MICROPROCESSORS
AND
MICROCOMPUTER-BASED
SYSTEM DESIGN

Mohamed Rafiuzzaman, Ph.D.
Professor
California State Polytechnic University
Pomona, California
and
Adjunct Professor
University of Southern California
Los Angeles, California

UNIVERSAL BOOK STALL
NEW DELHI
opponent systems, a system development flowchart, and typical practical applications, such as personal computers, robotics, real-time controllers, and fault-tolerant systems.

Chapter 2 provides a detailed coverage of 8085 hardware, software, I/O, timing, and system design.

Chapters 3 through 8 provide detailed descriptions of the architectures, addressing modes, instruction sets, I/O, and system design concepts of Intel's 8086, 80186, 80286, and 80386 and Motorola's 68000, 68020, 68030, and 68310 microprocessors.

Chapter 9 covers the basics of peripheral interfacing. Topics include keyboard/display interfacing, DMA controllers, printer interfaces, CRT and graphics controllers, floppy disk interface, and coprocessors.

Chapter 10 offers two detailed design problems. The purpose of these is to offer an opportunity to apply some of the design principles covered in the preceding chapters.

The appendices include materials on the HP 64600 microcomputer development system, data sheets on the Motorola 68000 and support chips, data sheets on the Intel 8085, 8086 microprocessors and support chips, and a glossary.

Mohamed Rafiquzzaman
Pomona, California

ACKNOWLEDGMENTS

The author wishes to express his sincere appreciation to his students, Marc Langer, Clyde Soo, Laurie Polumbo, James Moses, Larry Keller, Mark Walker, Francis Thomson, and others; to his colleague Professor Rajan Chandra of California State Polytechnic University, Pomona; to Raja Truong of Hughes Aircraft; to Mehdi Mirfakhri of Rail Systems, Inc., Ontario, California; and to others for making constructive suggestions.

The author is also grateful to Dr. W. C. Miller of the University of Windsor, Canada, to his brother and his wife Captain and Mrs. Zaman, and to his friend Rick Heffer for their support and inspiration throughout the writing effort.

The reviewers of the manuscript are to be commended for their comments and constructive criticisms. They are Dr. Kenneth J. Breeding, Department of Electrical Engineering and Computer Science, Ohio State University; Dr. Darrow F. Dawson, Department of Electrical Engineering, University of Missouri-Rolla; Dr. David G. Green, Department of Electrical Engineering, University of Alabama at Birmingham; Dr. Martin E. Kalisi, Department of Electronic and Electrical Engineering, California State Polytechnic University, San Luis Obispo; Dr. David W. Knapp, Department of Computer Science, University of Illinois, Urbana Champaign; Dr. Michel Lemo, Department of Electrical Engineering, University of Florida; Dr. Alan B. Marcovitz, Department of Computer Engineering, Florida Atlantic University; Dr. Kay M. Purcell, Department of Electrical Engineering, Southern Illinois University at Carbondale; and Dr. Joseph G. Tront, Department of Electrical Engineering, Virginia Polytechnic Institute and State University.

The author is especially indebted to Janice Morey, Sandy Pearlman, and Russ Hall of CRC Press, Inc. for their personal commitment, dedication, and outstanding job in bringing this book to publication. To all the others on the staff of CRC Press who literally worked round the clock in order to complete the final stages of preparation, the author expresses his sincere appreciation.

Finally, the author is grateful to Nancy, Mary, Karen, Francis, and Melanie for typing all of the manuscript.
## CONTENTS

Chapter 1
Introduction to Microprocessors and Microcomputer-Based Design ........................................ 1
  1.1 Evolution of the Microprocessor .................................................................................. 1
  1.2 Microcomputer Hardware ......................................................................................... 2
      1.2.1 The System Bus .................................................................................................. 3
      1.2.2 The Microprocessor ......................................................................................... 4
      1.2.3 Memory Organization ....................................................................................... 4
          1.2.3.a Introduction ............................................................................................... 9
          1.2.3.b Main Memory Array Design ....................................................................... 9
          1.2.3.c Memory Management Concepts .................................................................. 12
          1.2.3.d Cache Memory Organization ..................................................................... 16
      1.2.4 Input/Output (I/O) ......................................................................................... 22
          1.2.4.a Programmed I/O ....................................................................................... 25
          1.2.4.b Standard I/O versus Memory-Mapped I/O ................................................. 26
          1.2.4.c Unconditional and Conditional Programmed I/O .................................... 29
          1.2.4.d Typical Microcomputer I/O Circuits ......................................................... 31
          1.2.4.e Interrupt Driven I/O .................................................................................. 33
          1.2.4.f Direct Memory Access (DMA) .................................................................... 36
          1.2.4.g Summary of Microcomputer I/O Methods ............................................... 37
  1.3 Microcomputer System Software and Programming Concepts .................................... 40
      1.3.1 System Software .............................................................................................. 40
      1.3.2 Programming Concepts .................................................................................... 41
          1.3.2.a Machine Language Programming ............................................................. 41
          1.3.2.b Assembly Language Programming ............................................................ 42
          1.3.2.c High-Level Language Programming .......................................................... 43
  1.4 Microcomputer Typical Addressing Modes and Instructions ....................................... 44
      1.4.1 Introduction ..................................................................................................... 44
      1.4.2 Addressing Modes ........................................................................................... 44
      1.4.3 Instruction Types ............................................................................................. 45
  1.5 Basic Features of Microcomputer Development Systems ............................................. 48
  1.6 System Development Flowchart ................................................................................ 48
      1.6.1 Software Development .................................................................................... 63
      1.6.2 Hardware Development ................................................................................... 64
  1.7 Typical Practical Applications .................................................................................... 66
      1.7.1 Two-Position Controller .................................................................................. 66

Chapter 2
Intel 8085 ............................................................................................................................. 73
  2.1 Introduction ................................................................................................................ 73
  2.2 Register Architecture ............................................................................................... 76
  2.3 Memory Addressing ................................................................................................... 77
  2.4 8085 Addressing Modes ........................................................................................... 78
  2.5 8085 Instruction Set ................................................................................................... 79
  2.6 Timing Methods ......................................................................................................... 112
  2.7 8085 Pins and Signals ............................................................................................... 115
  2.8 8085 Instruction Timing and Execution .................................................................... 122
      2.8.1 Basic System Timing ....................................................................................... 123
      2.8.2 8085 Memory Read (IO/M = 0, RD = 0) and I/O Read (IO/M = 1, RD = 0) .. 125
      2.8.3 8085 Memory Write (IO/M = 0, WR = 0) and I/O Write (IO/M = 1, WR = 0) . 125
  2.9 8085 Input/Output (I/O) ......................................................................................... 130
      2.9.1 8085 Programmed I/O ................................................................................... 130
      2.9.1.a 8255/8256 I/O Ports .................................................................................... 130
      2.9.1.b 8155/8165 I/O Ports .................................................................................... 134
      2.9.2 8085 Interrupt System ..................................................................................... 141
      2.9.3 8085 DMA ....................................................................................................... 153
      2.9.4 8085 SIO and SOD Lines ................................................................................ 156
      2.10 8085-Based System Design ................................................................................ 160
      Questions and Problems ............................................................................................. 160

Chapter 3
Intel 8086 ............................................................................................................................. 167
  3.1 Introduction .............................................................................................................. 167
  3.2 Architecture .............................................................................................................. 174
  3.3 8086 Addressing Modes ........................................................................................... 174
      3.3.1 Addressing Modes for Accessing Immediate and Register Data (Register and Immediate Modes) ................................................................. 177
      3.3.1.a Register Addressing Mode .......................................................................... 177
      3.3.1.b Immediate Addressing Mode ..................................................................... 177
      3.3.2 Addressing Modes for Accessing Data in Memory (Memory Modes) .......... 177
      3.3.2.a Direct Addressing Mode .............................................................................. 177
      3.3.2.b Register Indirect Addressing Mode .............................................................. 177

viii
Chapter 4

4.1 Intel 8086 and 80286

4.1.1 Intel 8086

4.1.2 Intel 80286

4.1.3 80286 Memory Management

4.1.4 Protection

4.2 Intel 80386

4.2.1 Basic 80386 Programmer's Model

4.2.2 Instruction Set

4.2.3 Arithmetic Instructions

4.2.4 Bit Instruction

4.2.5 Byte-Set-On Condition Instructions

4.2.6 Conditional Jumps and Loops

4.2.7 Data Transfer

4.2.8 Flag Control

4.2.9 Logical

4.2.10 Serial

4.2.11 Table Look-Up Translation Instruction

4.2.12 High-Level Language Instruction

4.3 Memory Organization

4.4 I/O System

4.4.1 I/O Ports

4.4.2 8088 Interrupts

4.4.3 80386 Input and Initialization

4.4.4 80386 Timing

4.4.5 Decoding

4.4.6 80386 Bus Transfer Technique

4.4.7 80186 Read and Write Cycles

4.4.8 80286 Timing

4.4.9 80186 Software Interface

4.4.10 Protected Mode

4.4.11 Virtual 8086 Mode

Questions and Problems

4.1

4.2

4.3

4.4

4.5

4.6

4.7

4.8

4.9

4.10

4.11

4.12
Chapter 10

Design Problems

10.1 Design Problem No. 1 .............................................. 911

10.1.1 Problem Statement ........................................... 911

10.1.2 Objective ......................................................... 911

10.1.3 Operation ......................................................... 911

10.1.4 Hardware ........................................................ 912

10.1.5 Software ........................................................ 913

10.2 Design Problem No. 2 .............................................. 913

10.2.1 Problem Statement ........................................... 921

10.2.2 Soln No. 1 ......................................................... 921

10.2.2.a Hardware ...................................................... 921

10.2.2.b Microcomputer Development System ................. 925

10.2.2.c Software ....................................................... 925

10.2.3 Solution No. 2 .................................................... 928

10.2.3.a Hardware ...................................................... 932

10.2.3.b Software ....................................................... 932

Questions and Problems .............................................. 934

Appendix A

HP 64000 ................................................................. 957

Appendix B

Motorola MC68000 and Support Chips — Data Sheets ......... 987

Appendix C

Intel 8085, 8086, and Support Chips — Data Sheets .......... 999
1.1 EVOLUTION OF THE MICROPROCESSOR

Intel Corporation introduced the first microprocessor, the 4004, in 1971. The 4004 evolved from a development effort while designing a calculator chip set.

Soon after the 4004 appeared in the commercial market, three other microprocessors were introduced. These were the Rockwell International PPS 4, the Intel 8-bit 8008, and the National Semiconductor 16-bit IMP. 16.

The microprocessors introduced between 1971 and 1973 were the first-generation systems. They were designed using the PMOS (P-type MOS) technology. This technology provided low cost, slow speed, and low output currents and was not compatible with TTL (Transistor Transistor Logic).

After 1973, second-generation microprocessors such as Motorola 6800 and 8089, Intel 8085, and Zilog Z80 evolved. These processors were fabricated using the NMOS (N-type MOS) technology. The NMOS process offers faster speed and higher density than PMOS and is TTL-compatible.

After 1978, the third-generation microprocessors were introduced. These processors are 16 bits wide and include typical processors such as Intel 8086/80186/80286 and Motorola 68000/68010. These microprocessors are designed using the HCMOS (high-density MOS) technology.

HCMOS provides the following advantages over NMOS:

- Speed-Power-Product (SPP) of HCMOS is four times better than NMOS:
  - NMOS = 4 Picowatts (PW)
  - HCMOS = 1 Picowatt (PW)

Note that Speed-Power-Product:
- speed × power
- nanosecond × milliwatt
- picowatts

- Circuit densities provided by HCMOS are approximately twice those of NMOS:
  - NMOS = 4128 gates/μm²
  - HCMOS = 1852.5 gates/μm²

where 1 μm (micrometer) = 10⁻⁶ meter.

1.2 MICROCOMPUTER HARDWARE

Recently, Intel utilized the HCMOS technology to fabricate the 8085A. Thus, Intel offers a high-speed version of the 8085A called 8085AH. The price of the 8085AH is higher than the 8085A.

In 1980, fourth-generation microprocessors evolved. Intel introduced the first commercial 32-bit microprocessor, the problematic Intel 432. This processor was eventually discontinued by Intel. Since 1985, more 32-bit microprocessors have been introduced. These include Motorola's MC 68020 and 68030 and Intel 80386. These processors are fabricated using the low-power version of the HCMOS technology called the HCMOS.

The performance offered by the 32-bit microprocessor is more comparable to that of superminicomputers such as Digital Equipment Corporation's VAX11/750 and VAX11/780. Recently, Motorola has introduced a 32-bit RISC (Reduced Instruction Set Computer) microprocessor with a simplified instruction set called the MC88100.

The trend in microprocessors is not toward introduction of 64-bit microprocessors. Extensive research is being carried out for implementation of more on-chip functions and for improvement of the speeds of memory and I/O devices.

1.2.1 THE SYSTEM BUS

The system bus contains three buses. These are the address bus, the data bus, and the control bus. These buses connect the microprocessor to:

![Simple block diagram of a microprocessor.](8.jpg)
Introduction

1.2 Microcomputer Hardware

A general-purpose register is also capable of manipulating the stored data by shift left or right operations. These registers are normally 8, 16, and 32 bits wide. For example, all general-purpose registers in the 8080 are 32 bits wide. Each register is a dedicated register that can be used at any location. Typical dedicated registers bits can be used at either 8, 16, or 32 bits. Typical dedicated registers include the Program Counter (PC), the Instruction Register (IR), Status Register (SR), the Stack Pointer (SP), the Index Register, and the Barrel Shifter.

The PC normally contains the address of the next instruction to be executed. Upon activating the microprocessor chip's RESET input, the PC is reloaded. When the instruction is executed, the microprocessor will normally store the address of the instruction in the address register. In order to execute the instruction, the microprocessor must select the address and read it from the address register and execute the instruction. The program counter is then incremented by one. The program counter contents are then automatically fetched from the next address. A program is stored in a single instruction. The instruction fetch is handled by a jump or branch instruction. The instruction fetch is handled by a jump or branch instruction. The microprocessor runs in a single instruction. The instruction fetch is handled by a jump or branch instruction. The microprocessor runs in a single instruction. The instruction fetch is handled by a jump or branch instruction. The microprocessor runs in a single instruction. The instruction fetch is handled by a jump or branch instruction.
numbers. The overflow flag for signed numbers can be shown as $V = C7 \oplus C6$, where $C7$ is the final carry and $C6$ is the previous carry. This can be illustrated by the numerical examples shown below:

\[
\begin{array}{c|c|c}
& 0000 & 0100 \\
0000 & 0010 & 0100 \\
C7 & 0 & 0100 \\
C6 & 0 & 0100 \\
& 0 & 1010 \\
\end{array}
\]

From the above, the result is correct when $C6$ and $C7$ have the same values (0 in this case). When $C6$ and $C7$ are different, an overflow occurs. For example, consider the following:

\[
\begin{array}{c|c|c|c}
-66 & 0100 & 0110 \\
1011 & 1111 & 0100 \\
C7 & 1 & 0110 \\
C6 & 0 & 0110 \\
\end{array}
\]

The result is incorrect. Since $V = C6 \oplus C7 = 0 \oplus 1 = 1$, the overflow flag is set. Note that this applies to signed numbers only.

The stack pointer (SP) register addresses the stack. A stack is Last-In First-Out (LIFO) read/write memory in the sense that that it stores in last (PUSH) operations from one end. The stack can be implemented by using hardware or software.

The hardware stack is designed by using a set of high-speed registers to provide a fast response. The disadvantage of a hardware stack is that the microprocessor uses hardware stacks. The software stack, on the other hand, is implemented by using a number of RAM locations. The software stack provides an unlimited stack size but is slower than the hardware stack, addressed by a register called the stack pointer (SP). The size of the SP is dependent on the microprocessor's logical address size. The stack is not the program counter.

The stack pointer (SP) is used as a counter for an instruction or for general storage functions. The index register is used with instructions where tables or arrays of data are accessed. In this operation, the index register is used to modify the address portion of the instruction. These instructions can also be used to access a global address. For example, the indexed addressing mode is determined by adding the address portion of the instruction to the contents of the index register. Some microprocessors such as the 8085 do not offer indexing. However, the indexed mode in 8085 can be achieved by using proper instructions. The general-purpose register-based microprocessor can use any general-purpose register as the index register.

Typical 32-bit microprocessors such as the Intel 80386 and Motorola 68020/68030 include a special type of 32-bit shifter called barrel shifter for performing fast shift operations.

The barrel shifter is an on-chip component for 32-bit microprocessors and provides fast shift operations. For example, the 80386 barrel shifter can shift a number from 0 to 31 in one clock period (clock rate is 16.67 MHz).

The ALU in the microprocessors performs all arithmetic and logic operations.
The control unit of the microprocessor performs instruction interpreting and sequencing. In the interpretation phase, the control unit reads instructions from memory using PC as a pointer. It then recognizes the instruction type, gets the necessary operands, and routes them to the appropriate functional units of the execution unit. Necessary signals are issued to the execution unit to perform the desired operations, and the results are routed to the specified destination.

In the sequencing phase, the control unit determines the address of the next instruction to be executed and loads it into the PC. The control unit is typically designed using one of three techniques:

- **Hardwired control**
- **Microprogramming**
- **Nanoprocessing**

The hardwired control unit is designed by physically connecting typical components such as gates and flip-flops. Zilog's 16-bit microprocessor 28000 is designed using hardwired control. The microprogrammed control unit includes a control ROM for translating the instructions, Intel 8086 is a microprogrammed microprocessor. Nanoprogramming includes two ROMs inside the control unit. The first ROM (microROM) stores all the addresses of the second ROM (nanoROM). If the microinstructions (which is the case with the 68000/68020/68030) repeat many times in a microprogram, use of two-level ROMs provides tremendous memory savings. This is the reason that the control units of the 68000, 68020, and 68030 are nanoprogrammed. The control unit generates certain control signals for other microcomputer elements or reads in control inputs and activates functions such as RESET and interrupts. The RESET input of a typical microprocessor is activated to initialize PC so that the microprocessor knows where to find the first instruction. For example, the 8085 loads PC with 0000, after activation of the RESET pin via an external circuit. This means that the 8085 executes the first instruction as 0000. Therefore, the user's memory map must include this address.

1.2.3 MEMORY ORGANIZATION

1.2.3.1 Introduction

A memory unit is an integral part of any microcomputer system and its primary purpose is to hold programs and data. The major design goal of a memory unit is to allow it to operate at a speed close to that of the processor. However, the cost of a memory unit is so prohibitive that it is practically not feasible to design a large memory unit with one technology that guarantees a high speed. Therefore, in order to seek a trade-off.
between the cost and operating speed, a memory system is usually designed with different technologies such as solid state, magnetic, and optical. In a broad sense, a microcomputer memory system can be logically divided into three groups:

- Processor memory
- Primary or main memory
- Secondary memory

Processor memory refers to a set of CPU registers. These registers are used to hold temporary results when a computation is in progress. Also, there is no speed disparity between these registers and the microprocessor because they are fabricated using the same technology. However, the cost involved in this approach forces a microcomputer architect to include only a few registers (usually 8 or 16) in the microprocessor.

Primary memory is the storage area in which all programs are executed. The microprocessor can directly access only those items that are stored in primary memory. Therefore, all programs and data must be within the primary memory prior to execution. MOS technology is normally used in these days in primary memory design. Usually the size of the primary memory is much larger than processor memory and its operating speed is slower than the processor registers by a factor of 25.

Secondary memory refers to the storage medium comprising slow devices such as magnetic tapes and disks. These devices are used to hold large data files and huge programs such as compilers and data base management systems which are not needed by the processor frequently. Sometimes secondary memories are also referred to as auxiliary or backup store.

Primary memory normally includes ROM (Read-only Memory) and RAM (Random Access Memory). As the name implies, a ROM permits only a read access. Some ROMs are custom made, that is, their contents are programmed by the manufacturer. Such ROMs are called mask programmable ROMs. A typical example for a mask programmable ROM is the character generator ROM 2513. Since these ROMs are mass produced, they are inexpensive. Sometimes a user may have to program a ROM in the PROM. For instance, in a fuse-link ROM, programmable read-only memory (PROM) is available. The main disadvantage of a PROM is that it cannot be reprogrammed.

In practice, when programs are in the development stage, it is necessary that they be altered before they become a marketable product. ROMs that allow reprogramming are called Erasable Programmable Read-Only Memories (EPROMs).
shows the subcategories of ROMs, RAMs, and their associated technologies.

One of the best known read/write memories is the ferite core memory. In this case, the reading process destroys the stored information. This property is called destructive readout (DRO) and it is unique with respect to ferite cores. Note that a semiconductor memory does not exhibit this property and for this reason solid-state memories are referred to as non-ferite core memory. They are first transferred to a buffer, and from this buffer, data are rewritten into the location from where they are originally read. It is interesting to note that the inclusion of the memory buffer register (MBR) as a part of a RAM unit is the direct consequence of the phenomenon of DRO.

1.2.3.2 Main Memory Array Design

In many applications, a memory of large capacity is often realized by interconnecting several small-size memory blocks. In this section, design of a large main memory using small-size memories as building blocks is presented.

There are two types of techniques used for designing the main memory. These are linear decoding and fully decoding. We will illustrate the concepts associated with these techniques in the following.

First, consider the block diagram of a typical static RAM chip shown in Figure 1.4.

The capacity of this chip is 8192 bits and these bits are organized as 1024 words with 8 bits/word. Each word has a unique address and this is

specification on 10-bit address lines A9—A0 (note that 2^10 = 1024). The inputs and outputs are routed through the 8-bit bidirectional data lines D7 through D0. The operation of this chip is governed by the two control inputs: WE (write Enable) and CS (chip select). The truth table that describes the operation of this chip is shown in Table 1.1.

<table>
<thead>
<tr>
<th>CS</th>
<th>WE</th>
<th>MODE</th>
<th>Status of the bidirectional data lines D7—D0</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>X</td>
<td>Not selected</td>
<td>High impedance</td>
<td>Standby</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>Write</td>
<td>Acts as an input bus</td>
<td>Active</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>Read</td>
<td>Acts as an output bus</td>
<td>Active</td>
</tr>
</tbody>
</table>

Note: H — high, L — low, X — don't care.

From this table, it is easy to see that when CS input is low, the chip is not selected and thus the lines D7 through D0 are driven to the high impedance state. When CS = 1 and WE is low, data on lines D7—D0 are written into the word addressed by A0 through A9. Similarly, when CS = 1 and WE is high, the contents of the memory word whose address is specified on address lines A9 through A0 will appear on lines D7 through D0. Note that when the chip select input CS goes to low, the device is disabled and the chip automatically reduces its power requirements and remains in this low-power standby mode as long as CS remains low. This feature results in system power savings as high as 85% in larger systems, where the majority of devices are disabled.
1.2.3.3 Linear Decoding

This technique uses the unused address lines of the microprocessor as chip selects for the memory chips.

A simple way to connect an 8-bit microprocessor to a 64K RAM system using linear decoding is shown in Figure 1.5. In this approach, the address lines A9 through A0 of the microprocessor are used as a common input to each 1K x 8 RAM chip. The remaining 6 high-order lines are used to select one of the 6 RAM chips. For example, if A15[A14:A13:A12:A11:A10 = 000010], then the RAM chip 1 is selected. The address map realized by this arrangement is summarized in Figure 1.6. This method is known as the linear select decoding technique. The principal advantage of this method is that it does not require any decoding hardware. However, this approach has some disadvantages:

- Although with a 16-bit address bus we connect 64K bytes of RAM, we are able to interface only 6K bytes of RAM. This means that this idea wastes address space.
- The address map is not contiguous; rather, it is sparsely distributed.
- If both A11 and A10 are high at the same time, both RAM chips 0 and 1 are selected and thus a bus conflict occurs. This can be avoided by proper memory map to select the desired memory chip and deselect the others.
- Also, if all unused address lines are not utilized as chip selects for memory, then these unused pins become don't cares (can be 0 or 1). This results in foldback, meaning that a memory location will have its image in the memory map. For example, if A15 is don't care in design and if A14 to A0 address lines are used, then address 00000 and address 80000 are the same locations. This is called foldback and it wastes memory space.

1.2.3.3.4 Fully Decoding (i.e., absolute)

The difficulties such as the bus conflict and sparse address distribution are eliminated by the use of the fully decoded addressing technique. To see this, consider the organization shown in Figure 1.7. In this setup, we use a 2-to-4 decoder and interface the 8-bit microprocessor with 4K bytes of RAM. In particular, the four combinations of the lines A11 and A10 select the RAM chips as follows:

<table>
<thead>
<tr>
<th>A11</th>
<th>A10</th>
<th>Device selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>RAM chip 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RAM chip 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RAM chip 2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RAM chip 3</td>
</tr>
</tbody>
</table>

![FIGURE 1.5 An 8-bit microprocessor interfaced to a 64K RAM system using the linear select decoding technique.](image)
Also observe that this hardware makes sure that the memory system is enabled only when the lines A15 through A12 are zero. The complete address map corresponding to this organization is summarized in Figure 1.8.

1.2.3.6 Memory Management Concepts

Due to the massive amount of information that must be saved in most systems, the mass storage is often a disk. If each access is to a disk (even a hard disk), then system throughput will be reduced to unacceptable levels.

An obvious solution is to use a large and fast locally accessed semiconductor memory. Unfortunately the storage cost per bit for this solution is on-board semiconductor main memory must be designed into a system. This requires a mechanism to manage the two-way flow of information between the primary (semiconductor) and secondary (disk) media. This block usage, and replace them in a non-linear way. The primary memory must therefore be able to dynamically allocate memory spaces.

An operating system must have resource protection from corruption or abuse by users. Users must be able to protect areas of code from each other,

FIGURE 1.6 Address map realized by the system shown in Figure 1.5.

FIGURE 1.7 An 8-bit microprocessor interfaced to a 4K RAM system using a fully decoded addressing technique.

while maintaining the ability to communicate and share other areas of code. All these requirements indicate the need for a device, located between the microprocessor and memory, to control accesses, perform address mappings, and act as an interface between the logical and microprocessor and physical (memory) address spaces. Since this device must manage the memory use configuration, it is appropriately called the memory management unit (MMU). Typical 32-bit microprocessors such as Motorola 68030 and Intel 80386 include on-chip MMU.
The MMU reduces the burden of memory management functions on
the operating system.

The basic functions provided by the MMU are address translation and
protection. The MMU translates logical program addresses to physical
memory addresses. Note that in assembly language programming, ad-
dresses are referred to by symbolic names. These addresses in a program
are called logical addresses since they indicate the logical position of
instructions and data. The MMU translates these logical addresses to
physical addresses provided by the memory chips. The MMU can perform address
translation in one of two ways:

1. By using the substitution technique as shown in Figure 1.9a
2. By adding an offset to each logical address to obtain the correspond-
ing physical address as shown in Figure 1.9b.

Address translation using substitution is faster than the offset method.
However, the offset method has the advantage of mapping a logical address
to any physical address as determined by the offset value.

Memory is usually divided into small manageable units. The terms
"page" and "segment" are frequently used to describe these units. Paging
divides the memory into equal-sized pages, while segmentation divides the
memory into variable-sized segments.

It is relatively easier to implement the address translation table if the
logical and main memory spaces are divided into pages. The term "page"
is associated with logical address space, while the term "block" usually
refers to a page in main memory space.

There are three ways to map logical addresses to physical addresses.
These are paging, segmentation, and combined paging/segmentation.

In a paging system, a user has access to a larger address space than
physical memory provides. The virtual memory system is managed by both
hardware and software. The hardware included in the memory manage-
ment unit handles address translation. The memory management software
in the operating system performs all functions including page replacement
policies in order to provide efficient memory utilization. The memory
management software performs functions such as removal of the desired
page from main memory to accommodate a new page, transferring a new
page from secondary to main memory at the right instant of time, and
placing the page at the right location in memory.

If the main memory is full during transfer from secondary to main
memory, it is necessary to remove a page from main memory to accom-
modate the new page. Two popular page replacement policies are first-in-first-
out (FIFO) and least recently used (LRU). The FIFO policy removes the
page from main memory that has been resident in memory for the longest
amount of time. The FIFO replacement policy is easy to implement. One
of the main disadvantages of the FIFO policy is that it is likely to replace
heavily used pages. Note that heavily used pages are resident in main
memory for the longest amount of time. Sometimes this replacement
policy might be a poor choice. For example, in a time-shared system, several users normally share a copy of the text editor in order to type and correct programs. The FIFO policy on such a system might replace a heavily used editor page to make room for a new page. This editor page might be recalled to main memory immediately. The FIFO, in this case, would be a poor choice.

The LRU policy, on the other hand, replaces that page which has not been used for the longest amount of time.

In the segmentation method, the MMU utilizes the segment selector to obtain a descriptor from a table in memory containing several descriptors. A descriptor contains the physical base address for a segment, the segment’s privilege level, and some control bits. When the MMU obtains a logical address from the microprocessor, it first determines whether the segment is already in the physical memory. If it is, the MMU adds an offset component to the segment base component of the address obtained from the segment descriptor table to provide the physical address. The MMU then generates the physical address on the address bus for selecting the memory. If the MMU does not find the logical address in physical memory, it interrupts the microprocessor. The microprocessor executes a service routine to bring the desired program from a secondary memory such as disk to the physical memory. The MMU determines the physical address using the segment offset and descriptor as above and then generates the physical address on the address bus for memory. A segment will usually consist of an integral number of pages, say, each 256 bytes long. With different-sized segments being swapped in and out, areas of valuable primary memory can become unusable. Memory is unusable for segmentation when it is sandwiched between already allocated segments and if it is not large enough to hold the latest segment that needs to be loaded. This is called external fragmentation and is handled by MMUs using special techniques. An example of external fragmentation is given in Figure 1.10.

![Figure 1.10](image1)

**FIGURE 1.10** Memory fragmentation (external).

The advantages of segmented memory management are that few descriptors are required for large programs or data spaces, and internal fragmentation (to be discussed later) is minimized. The disadvantages include external fragmentation, involved algorithms for placing data are required, possible restrictions on starting address, and longer data swap times are required to support virtual memory.

Address translation using descriptor tables offers a protection feature. A segment or a page can be protected from access by a program section of a lower privilege level. For example, the selector component of each logical address includes one or two bits indicating the privilege level of that segment. Each segment descriptor also includes one or two bits providing the privilege level of that segment. When an executing program tries to access a segment, the MMU can compare the selector privilege level with the descriptor privilege level. If the segment selector has the same or higher privilege level, then the MMU segment selector has the same or higher privilege level. Otherwise, the MMU can interpret the microprocessor informing of a descriptor, the MMU can interrupt the microprocessor informing of a descriptor, the MMU can interrupt the microprocessor informing of a descriptor. The MMU will then respond to the interrupt and perform the required operations.

A segment or a page can be protected from access by a program section of a lower privilege level. For example, the selector component of each logical address includes one or two bits indicating the privilege level of that segment. Each segment descriptor also includes one or two bits providing the privilege level of that segment. When an executing program tries to access a segment, the MMU can compare the selector privilege level with the descriptor privilege level. If the segment selector has the same or higher privilege level, then the MMU segment selector has the same or higher privilege level. Otherwise, the MMU can interpret the microprocessor informing of a descriptor, the MMU can interrupt the microprocessor informing of a descriptor. The MMU will then respond to the interrupt and perform the required operations.

A segment or a page can be protected from access by a program section of a lower privilege level. For example, the selector component of each logical address includes one or two bits indicating the privilege level of that segment. Each segment descriptor also includes one or two bits providing the privilege level of that segment. When an executing program tries to access a segment, the MMU can compare the selector privilege level with the descriptor privilege level. If the segment selector has the same or higher privilege level, then the MMU segment selector has the same or higher privilege level. Otherwise, the MMU can interpret the microprocessor informing of a descriptor, the MMU can interrupt the microprocessor informing of a descriptor. The MMU will then respond to the interrupt and perform the required operations.
of pages. The logical address is divided into three components: segment, page, and word. The segment component defines a segment number, the page component defines the page within the segment, and the word component provides the particular word within the page. A page component of n bits can provide up to $2^n$ pages. A segment can be assigned with one or more pages up to a maximum of $2^n$ pages; therefore, a segment size depends on the number of pages assigned to it.

Protection mechanism can be assigned either on physical address or logical address. Physical memory protection can be accomplished by using one or more protection bits with each block to define the access type permitted on the block. This means that each time a page is transferred from one block to another, the block protection bits must be updated. A more efficient approach is to provide a protection feature in logical address space by including protection bits in the descriptors of the segment table in the MMU.

1.2.3.3 Cache Memory Organization

The performance of a microcomputer system can be significantly improved by introducing a small, expensive, but fast memory between the microprocessor and main memory. This memory is called cache memory and this idea was first introduced in the IBM 360/85 computer. Later on, this concept was also implemented in minicomputers such as PDP-11/70. With the advent of VLSI technology, the cache memory technique is gaining acceptance in the microcomputer world. For example, a small on-chip cache memory is implemented in Intel's 32-bit microprocessor, the 80386, and Motorola's 32-bit microprocessor, the MC 68020/68030.

The block diagram representation of a microprocessor system that employs a cache memory is shown in Figure 1.12. Usually, a cache memory is very small in size and its access time is less than that of the main memory by a factor of 5. Typically, the access times of the cache and main memories are 100 and 500 ns, respectively.

If a reference is found in the cache, we call it a cache hit and the data pertaining to the microprocessor reference is transferred to the microprocessor from the cache. However, if the reference is not found in the cache, we call it a cache miss. When there is a cache miss, the main memory is accessed by the microprocessor and the data are then transferred to the microprocessor from the main memory. At the same time, a block of data containing the desired data needed by the microprocessor is transferred from the main memory to the cache. The block normally contains 4 to 16 words, and this block is placed in the cache using the standard replacement policies such as FIFO (First In First Out) or LRU (Least Recently Used).

![Diagram of cache memory organization](image)

**FIGURE 1.12** Memory organization of a computer system that employs a cache memory.

This block transfer is done with a hope that all future references made by the microprocessor will be confined to the fast cache.

The relationship between the cache and main memory blocks is established using mapping techniques. These widely used mapping techniques are:

- Direct mapping
- Fully-associative mapping
- Set-associative mapping

In direct mapping, the main memory address is divided into two fields: an index field and a tag field. The number of bits in the index field is equal to the number of address bits required to access the cache memory.

Assume that the main memory address is in bits wide and the cache memory address is in bits wide. Then the index field requires n bits and the memory address is n bits wide. The n-bit address accesses the cache. Each word tag field is $(n - n)$ bits wide. The n-bit address accesses the cache. When the tag field of the main memory address is accessed, the tag field of the main memory is used to access the cache. The tag field of the main memory is compared with the tag field in the word read from cache. A hit occurs if the tags match. This means that the desired data word is in cache. A miss occurs if there is no match and the required word is read from main memory.
memory. It is written in the cache along with the new tag. A random access memory is used as the cache memory.

One of the main drawbacks of direct mapping is that numerous misses different tags are accessed several times. This can be minimized by having such words far apart in the address range.

The fastest cache memory utilizes an associative memory. The method is known as fully associative mapping. Each associative memory contains a main memory address and its content (data). When the microprocessor (simultaneously) with all addresses in the associative memory. If there is a memory and sent to the microprocessor. If a tag occurs, the main memory associative cache memory. If the cache is full, certain policies such as FIFO (first-in, first-out) are used as replacement algorithm for the cache. The associative cache is expensive but provides fast operation.

The set-associate mapping is a combination of direct and associative mapping. Each cache word stores two or more main memory words using the index address. Each main memory word consists of a tag and its data word. An index with two or more tags and data words forms a set. When the microprocessor generates a memory request, the index of the main memory address is used as the cache address. The tag field of the main memory is stored under the index. If a match occurs, the desired data word is memory and also written into the cache. The hit ratio improves as the set size increases. This is because more words with the same index but different tags can be stored in cache.

There are two ways of writing into cache: the write-back and write-through methods. In the write-back method, whenever the microprocessor writes something into a cache word, a dirty bit is assigned to the cache word, and the dirty word is replaced with a new word, the dirty word is the incoming new word. The advantage of this method is that it avoids unnecessary writing into main memory. In the write-through method, whenever the microprocessor alters a cache address, the same alteration is made in the main memory copy of the altered cache address. This policy can be easily implemented and also it insures that the contents of the main memory are always valid. This feature is desirable in a multiprocessor system where the main memory is shared by several processors. However, this approach may lead to several unnecessary write to main memory.

One of the important aspects of cache memory organization is to devise a method that insures proper utilization of the cache. Usually, the tag directory contains an extra bit for each entry. This additional bit is called a valid bit. When the power is turned on, the valid bit corresponding to each cache block entry of the tag directory is reset to zero. This is done in order to indicate that the cache block holds invalid data. When a block of data is first transferred from the main memory to a cache block, the valid bit corresponding to this cache block is set to 1. In this arrangement, whenever the valid is a zero, it implies that a new incoming block can overwrite the existing cache block. Thus, there is no need to copy the contents of the cache block being replaced into the main memory.

### 1.2.4 INPUT/OUTPUT (I/O)

This section describes the basic input and output techniques used by microcomputers to transfer data between the microcomputer and an external device. The general characteristics of I/O are described. One communicates with a microcomputer system via the I/O devices interfaced to it. The user can enter programs and data using the keyboard on a terminal and execute the programs to obtain results. Therefore, the I/O devices connected to a microcomputer system provide an efficient means of communication between the computer and the outside world. These I/O devices are commonly called peripherals and include keyboards, CRT displays, printers, and disks.

The characteristics of the I/O devices are normally different from those of the microcomputer. For example, the speed of operation of the peripherals is usually slower compared to the microcomputer, and the word length of the microcomputer may be different from the data format of the peripheral device. To make the characteristics of the I/O devices compatible with those of the microcomputer, interface hardware circuits between the microcomputer and I/O devices is necessary. Interfaces provide all input and output transfers between the microcomputer and peripherals by using an I/O bus. An I/O bus carries three types of signals: device address, data, and command status.

For 16- and 32-bit microcomputers, a separate intelligent I/O processor (IOP) or data channel is provided to route all I/O transfers. To make 8-bit microcomputer systems inexpensive, a separate interface rather than a smart I/O processor is provided with each I/O device. I/O processors control all major I/O functions and relieve the microcomputer of these tasks.
The microprocessor uses the I/O bus when it executes an I/O instruction. A typical I/O instruction has three fields. When the microprocessor executes an I/O instruction, the control unit decodes the op-code field and identifies it as an I/O instruction. The microprocessor then places the device address and command from the respective fields of the I/O instruction onto the I/O bus. The interface for various devices connected to the I/O bus decodes this address, and appropriate interface is selected. The identified interface decodes the command lines and determines the function to be performed. Typical functions include receiving data from an input device into the microprocessor or sending data to an output device from the microprocessor.

In a typical microcomputer system, the user gets involved with two types of I/O devices: physical I/O and virtual I/O. When the microcomputer has no operating system, the user must work directly with physical I/O devices and perform detailed I/O design.

There are three ways of transferring data between the microcomputer and a physical I/O device:

- Programmed I/O
- Interrupt-driven I/O
- Direct memory access (DMA)

The microcomputer executes a program to communicate with an external device via a register called the I/O port for programmed I/O.

An external device requests the microcomputer to transfer data by activating a signal on the microcomputer's interrupt line during interrupt I/O. In response, the microcomputer executes a program called the interrupt-service routine to carry out the function desired by the external device.

Data transfer between the microcomputer's memory and an external device occurs without microprocessor involvement with direct memory access.

For a microcomputer with an operating system, the user works with virtual I/O devices. The user does not have to be familiar with the characteristics of the physical I/O devices. Instead, the user performs data transfers between the microcomputer and the physical I/O devices indirectly by calling the I/O routines provided by the operating system using virtual I/O instructions.

### 1.2.4.4 Programmed I/O

As described earlier, the microcomputer communicates with an external device via one or more registers called I/O ports using programmed I/O. These I/O ports are occasionally fabricated by the manufacturer in the same chip as the memory chip to achieve minimum chip count for small system applications. For example, the Intel 8035/8755 contains 2K bytes of ROM/EPROM with two I/O ports. The Motorola 6846 has 2K bytes of ROM and an 8-bit I/O port.

I/O ports are usually of two types. For one type, each bit in the port can be individually configured as either an input or output. For the other type, all bits in the port can be set as either all parallel input or output bits. Each port can be configured as an input or output port by another bit. Each port contains register called the command, or data-direction register. The port contains the actual input or output data. The data-direction register is an output register and can be used to configure the bits in the port as inputs or outputs.

Each bit in the port can usually be set up as an input or output by respectively writing a 0 or a 1 in the corresponding bit of the data-direction register. As an example, if an 8-bit data-direction register contains 34, then the corresponding port is defined as follows:

```
<table>
<thead>
<tr>
<th>Bit position</th>
<th>Data-direction register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 0 0</td>
<td>I/O port</td>
</tr>
</tbody>
</table>
```

In the preceding example, since 34 (0011 0100) is sent as an output into the data-direction register, bits 0, 1, 3, 6, and 7 of the port are set up as inputs, and bits 2, 4, and 5 of the port are defined as outputs. The microcomputer can then send outputs to external devices, such as LEDs, connected to bits 2, 4, and 5 through a proper interface. Similarly, the microcomputer can input the status of external devices, such as switches, through bits 0, 1, 3, 6, and 7. To input data from the input switches, the microcomputer assumes here inputs the complete byte, including the 8-bit microcomputer assumes here inputs the complete byte, including the bit to which LEDs are connected. While receiving input data from an I/O port, however, the microcomputer places a value, probably 0, at the bit of the data-direction register that is configured as outputs and the program must interpret them as "don't cares". At the same time, the microcomputer's outputs to bits configured as inputs are disabled.

For parallel I/O, there is only one data-direction register, known as the command register for all ports. A particular bit in the command register configures all bits in the port as either inputs or outputs.
Consider two I/O ports in an I/O chip along with one command register. Assume that a 0 or a 1 in a particular bit position defines all bits of ports A or B as inputs or outputs.

For example,

![Diagram showing I/O ports and command register](image)

Some I/O ports are called handshake ports. Data transfer occurs via these ports through exchanging of control signals between the I/O controller and an external device.

1.2.4.b Standard I/O Versus Memory-Mapped I/O

I/O ports are addressed using either standard I/O or memory-mapped I/O techniques. The standard I/O, also called isolated I/O, uses the IO/M control pin on the microprocessor chip. The processor outputs a HIGH on this pin to indicate to memory and the I/O chips that an I/O operation is taking place. A LOW output from the processor to this pin indicates a memory operation. Execution of IN or OUT instructions makes the IO/M HIGH, whereas memory-oriented instructions, such as LDA and STA, drive the IO/M to LOW. In standard I/O, the processor uses the IO/M pin to distinguish between I/O and memory. For 8-bit processors, an 8-bit address is typically used for each I/O port. This is because 8 bits are the basic data unit for these processors. Eight-bit processors are usually capable of directly addressing 64K bytes of memory using 16 address lines. With an 8-bit I/O port address, these processors are capable of addressing 256 ports. However, in a typical application, there are usually four or five I/O ports required. Some of the address bits of the microprocessor are normally decoded to obtain the I/O port addresses. With memory-mapped I/O,

the processor does not differentiate between I/O and memory and, therefore, does not use the IO/M control pin. The processor uses a portion of the memory addresses to represent I/O ports. The I/O ports are mapped into the processor's main memory and, hence, are called memory-mapped I/O. Each method has its advantages and disadvantages. For example, when standard I/O is used, an 8085 processor normally uses the 2-byte IN or OUT instruction as follows:

- **IN**
  - Port number
  - 2-byte instruction for inputting data from the specified I/O port into the processor's accumulator

- **OUT**
  - Port number
  - 2-byte instruction for outputting data from the accumulator into the specified I/O port

With memory-mapped I/O, an 8-bit processor normally uses 3-byte instructions, namely, LDA and STA, as follows:

- **LDA**
  - Port address
  - Port address mapped into memory
  - 3-byte instruction for inputting a byte into accumulator

- **STA**
  - Port address
  - Port address mapped into memory
  - 2-byte instruction for outputting data into the specified port

In memory-mapped I/O, the most significant bit (MSB) of the address is typically used to distinguish between I/O and memory. If the MSB of address is 1, an I/O port is selected. If the MSB of address is 0, a memory location is accessed. Sixteen- and thirty-two-bit microprocessors provide special control signals for performing memory-mapped I/O. Thus, these processors do not use MSB of the address lines. The processor can perform operations on port data without moving it into a microprocessor register through memory-mapped I/O. The port data must be moved into the accumulator for further processing while using standard I/O.

1.2.4.c Unconditional and Conditional Programmed I/O

The processor can send data to an external device at any time during
unconditional I/O. The external device must always be ready for data transfer. A typical example is when the processor outputs a 7-bit code through an I/O port to drive a seven-segment display connected to this port.

In conditional I/O, the processor outputs data to an external device via handshaking. Data transfer occurs by the exchanging of control signals between the processor and an external device. The processor inputs the status of the external device to determine whether the device is ready for data transfer. Data transfer takes place when the device is ready.

The concept of conditional I/O will now be demonstrated by means of data transfer between a processor and an analog-to-digital (A/D) converter. Consider, for example, the A/D converter shown in the accompanying figure.

The A/D converter just shown converts an analog voltage \( V_x \) into an 8-bit binary output at pins D7—D0. A pulse at the START conversion pin initiates the conversion. This drives the BUSY signal LOW. The signal stays LOW during the conversion process. The BUSY signal goes HIGH as soon as the conversion ends. Since the A/D converter's output is tristated, a LOW on the OUTPUT ENABLE pin initiates the converter's output. A HIGH on the OUTPUT ENABLE drives the converter's output to a high impedance state.

The concept of conditional I/O can be demonstrated by interfacing the A/D converter to an 8-bit processor. Figure 1.13 shows such an interfacing example.

The user writes a program to carry out the conversion process. When this program is executed, the processor sends a pulse to the START pin of

![Figure 1.13 Interfacing an A/D converter to an 8-bit processor.](image)

The processor then checks the BUSY signal by bit 1 of port A to determine if the conversion is completed. If the BUSY signal is HIGH (indicating the end of conversion), the processor sends a LOW to the OUTPUT ENABLE pin of the A/D converter. The processor then inputs the converter's D0—D7 outputs via port B. If the conversion is not completed, the processor waits in a loop checking for the BUSY signal to go HIGH.

1.2.4.4 Typical Microcomputer I/O Circuits

The tristate output circuit shown in Figure 1.14 utilizes totem-pole-type output called a PUSH-PULL circuit providing low-output currents. Therefore, current amplifier (buffer) is required to drive devices such as LED.

![Figure 1.14 Typical microprocessor output circuit.](image)
Chapter 1

1.2 Microcomputer Hardware

Since $I_{	ext{sat}} = 400 \mu A = I_f(Q_i), \beta$ for transistor $Q_i$ is

\[
\beta = \frac{I_C(Q_i)}{I_B(Q_i)} = \frac{10 mA}{400 \mu A} = 25
\]

Therefore, the interface design is complete, and a transistor with a minimum of saturation $\beta$ of 25 and $R_s = 4.25 \Omega$ and $R_p = 280 \Omega$ is required.

In the input mode of the tristate I/O pin configured by the data direction register, an input switch shown in Figure 1.16 can be interfaced.

1.2.4 Interrupt Driven I/O

A disadvantage of conditional programed I/O is that the microcomputer needs to check the status bit (BUSY signal for the A/D converter) by waiting in a loop. This type of I/O transfer is dependent on the speed of the external device. For a slow device, this waiting may slow down the capability of the microprocessor to process other data. The interrupt I/O technique is efficient in this type of situation.

Interrupt I/O is a device-initiated I/O transfer. The external device is connected to a pin called the interrupt (INT) pin on the processor chip. When the device needs an I/O transfer with the microcomputer, it activates the interrupt pin of the processor chip. The microcomputer usually completes the current instruction and saves at least the contents of the current program counter on the stack.

The microcomputer then automatically loads an address into the program counter to branch to a subroutine-like program called the interrupt-service routine. This program is written by the user. The external device wants the microcomputer to execute this program to transfer data. The last instruc-
of the service routine is a RETURN, which is typically the same instruction used at the end of a subroutine. This instruction normally loads the address (saved in the stack before going to the service routine) in the program counter. Then, the microcomputer continues executing the main program.

1.2.4.1 Interrupt Types

There are typically three types of interrupts: external interrupts, traps, and software interrupts.

External interrupts are initiated through the microcomputer’s interrupt pins by external devices such as A/D converters. A simple example of an external interrupt was given in the previous section.

External interrupts can further be divided into two types: maskable and nonmaskable. A maskable interrupt is enabled or disabled by executing instructions such as EI or DI. If the microcomputer’s interrupt is disabled, the microcomputer ignores the maskable interrupt. Some processors, such as the Intel 38086, have an interrupt-flag bit in the processor status register. When the interrupt is disabled, the interrupt-flag bit is 1, so no maskable interrupts are recognized by the processor. The interrupt-flag bit resets to zero when the interrupt is enabled.

The nonmaskable interrupt has higher priority than the maskable interrupt. If both maskable and nonmaskable interrupts are activated at the same time, the processor will service the nonmaskable interrupt first. The nonmaskable interrupt is typically used as a power failure interrupt. Processors normally use +5 V DC, which is transformed from 110 V AC. If the power falls below 90 V AC, the DC voltage of +5 V cannot be maintained. However, it will take a few milliseconds before the AC power can drop this low (below 90 V AC). In these few milliseconds, the power failure-sensing circuitry can interrupt the processor. An interrupt service routine can be written to store critical data in nonvolatile memory such as battery-backed CMOS RAM. The interrupted program can continue without any loss of data when the power returns.

Some processors are provided with a maskable handshake interrupt. This interrupt is usually implemented by using two pins: INTR and INTA. When the INTR pin is activated by an external device, the processor completes the current instruction, saves at least the current program counter on stack, and generates an interrupt acknowledge (INTA). In response to the INTA, the external device provides an instruction, such as CALL, using external hardware on the data bus of the microcomputer. This instruction is then read and executed by the microcomputer to branch to the desired service routine.

1.2.4.2 Interrupt Address Vector

The technique used to find the starting address of the service routine (commonly known as the interrupt address vector) varies from one processor to another. With some processors, the manufacturers define the fixed starting address for each interrupt. Other manufacturers use an indirect approach by defining fixed locations where the interrupt address vector is stored.

1.2.4.3 Saving the Microprocessor Registers

When a processor is interrupted, it saves at least the program counter on stack so the processor can return to the main program after executing the service routine. Some processors save only one or two registers, such as the program counter and status register. Other processors save all microprocessor registers before going to the service routine. The user should know the specific registers the processor saves prior to executing the service routine. This will enable the user to use the appropriate return instruction at the end of the service routine to restore the original conditions upon return to the main program.

1.2.4.4 Interrupt Priorities

A processor is typically provided with one or more interrupt pins on the chip. Therefore, a special mechanism is necessary to handle interrupts from several devices that share one of these interrupt lines. There are two ways of servicing multiple interrupts: polled and daisy chain techniques.

Polled interrupts are handled by software and therefore are slower when compared with daisy chaining. The processor responds to an inter-
1.2.4.f Direct Memory Access (DMA)

Direct Memory Access (DMA) is a technique that transfers data between a microcomputer's memory and I/O device without involving the microprocessor. DMA is widely used in transferring large blocks of data between a peripheral device and the microcomputer's memory. The DMA technique uses a DMA controller chip for the data transfer operation. The main functions of a typical DMA controller are summarized as follows:

- The I/O devices request DMA operation via the DMA request line of the controller chip.
- The controller chip activates the microprocessor HOLD pin, requesting the CPU to release the bus.
- The processor sends HLDA (hold acknowledge) back to the DMA controller, indicating that the bus is disabled. The DMA controller places the current value of its internal registers, such as the address register and counter, on the system bus and sends a DMA acknowledge to the peripheral device. The DMA controller completes the DMA transfer.

There are three basic types of DMA: block transfer, cycle stealing, and interleaved DMA.

For block-transfer DMA, the DMA controller chip takes the bus from the microcomputer to transfer data between the memory and I/O device. The microprocessor has no access to the bus until the transfer is completed. During this time, the microprocessor can perform internal operations that do not need the bus. This method is popular with microprocessors. Using this technique, blocks of data can be transferred.

Data transfer between the microcomputer memory and an I/O device occurs on a word-by-word basis with cycle stealing. Typically, the microprocessor clock is enabled by ANDing an INHIBIT signal with the system clock. The system clock has the same frequency as the microprocessor clock.

The DMA controller controls the INHIBIT line. During normal operation, the INHIBIT line is HIGH, providing the microprocessor with control of the system bus. When DMA operation is desired, the controller makes the INHIBIT line LOW for one clock cycle. The microprocessor is then stopped completely for one cycle. Data transfer between the memory and I/O takes place during this cycle. This method is called cycle stealing because the DMA controller takes away or steals a cycle without microprocessor recognition. Data transfer takes place over a period of time.

With interleaved DMA, the DMA controller chip takes over the system bus when the microprocessor is not using it. For example, the microprocessor does not use the bus while incrementing the program counter or performing an ALU operation. The DMA controller chip identifies these cycles and allows the transfer of data between the memory and I/O device. Data transfer takes place over a period of time for this method.

The DMA controller chip usually has at least three registers normally selected by the controller's register select (RS) line: an address register, a terminal count register, and a status register. Both the address and terminal count registers are initialized by the microprocessor. The address register contains the starting address of the data to be transferred, and the terminal count register contains the desired block to be transferred. The status register contains information such as completion of DMA transfer.

1.2.4.g Summary of Microcomputer I/O Methods

Figure 1.17 summarizes the I/O structure (explained so far) of typical microcomputers.

1.2.4.h Coprocessors

In typical 8-bit microprocessors such as the Intel 8085 and Z-80, technology places a limit on the chip area. In consequence, these microprocessors include no hardware or firmware for performing scientific computations such as floating-point arithmetic, matrix manipulation, and graphic data processing. Therefore, users of these systems must write these programs. Unfortunately, this approach is unacceptable in high-speed applications, since program execution takes a significant amount of time. To eliminate this problem, coprocessors are used.
In this approach, a single chip is built for performing scientific computations at high speed. However, the chip is regarded as a companion to the original or host microprocessor. Typically, each special operation is encoded as an instruction that can be interpreted only by the companion processor. When the companion microprocessor encounters one of these special instructions, it assumes the processing functions independent of the host microprocessor. The companion microprocessor that operates in this manner is called the coprocessor. Therefore, this concept not only extends the capabilities of the host microprocessor, but also increases the processing rate of the system. The coprocessor concept is widely used with typical 32-bit microprocessors such as the Motorola 68020 and Intel 80386.

Functionally, the coprocessor provides a logical extension of the programmer's model in the way of instructions, registers, and operand types. This extension is transparent to the programmer.

It is important to make the distinction between standard peripheral hardware and a coprocessor. A coprocessor is a device that has the capability of communicating with the main processor through the protocol defined as the coprocessor interface. As mentioned before, the coprocessor also adds additional instructions, registers, and data types that are not directly supported by the main processor. The coprocessor provides capabilities to the user without appearing to be hardware external to the main processor.

Standard peripheral hardware, on the other hand, is generally accessed through the use of interface registers mapped into the memory space of the main processor. The programmer uses standard processor instructions to access the peripheral interface registers and thus utilize the services provided by the peripheral. It should be pointed out that even though a peripheral can provide capabilities equivalent to a coprocessor for many applications, the programmer must implement the communication protocol between the main processor and the peripheral necessary to use the peripheral hardware. Two main techniques may be used to pass commands to a coprocessor. These are intelligent monitor interface and coprocessors using special signals.

In the intelligent monitor interface, the coprocessor monitors the instruction stream by obtaining commands directly from the bus at the same time as the main processor. The Intel 8087 floating-point coprocessor is of this type, as it monitors the instruction stream simultaneously with the main processor such as the Intel 8086. This has the obvious advantage of requiring no additional bus cycles to pass the content of the instruction word to the coprocessor. One of the main disadvantages of this approach is that each coprocessor in the system must duplicate the bus monitoring circuitry and instruction queue, tracking all branches, wait states, operand fetches, and instruction fetches.

In the second type, the coprocessor may be explicitly addressed by certain instructions, which initiate a special sequence of microinstructions in the main processor to effect command and operand transfer.

In this approach, when the main processor executes a coprocessor instruction, it decodes the instruction and writes a command in the command register (one of the interface registers) specifying the operation required by the coprocessor. In response, the coprocessor writes data back in a register, called the response register (one of the interface registers). The main processor can read these data, and it tells the main processor certain information such as whether additional information is required by the coprocessor to carry out the operation. If such data are required, the main processor provides this; otherwise, the coprocessor carries out the operation concurrently with the main and provides the result.

An advantage of this approach is that no special signals are required for the coprocessor interface.

One of the main disadvantages of this method is that once the main processor detects a coprocessor instruction, the main has to use bus bandwidth and timing to transmit the command to the appropriate coprocessor. The Motorola 68881 (floating-point coprocessor) is of this type.
1.3 MICROCOMPUTER SYSTEM SOFTWARE AND PROGRAMMING CONCEPTS

In the early days of computer system design, the high cost of hardware components completely overshadowed the labor costs of software design. Now, hardware costs have dropped so low that software development time has become the critical factor in achieving low-cost microcomputer-based products. The system software for a microcomputer allows one to develop application programs for microprocessor-based systems. In this section, the basic concepts associated with software and programming will be discussed.

1.3.1 SYSTEM SOFTWARE

Typical microcomputer system software includes editors, assemblers, compilers, interpreters, debuggers, and an operating system. The editor is used to create and change source programs. Source programs can be written in assembly language, high-level language such as Pascal, or in data tables. The editor has commands to change, delete, insert lines or characters. The text editor is a special type of editor that is used to edit and edit text in a general-purpose computer, whether the text is a report, a letter, or a program.

An assembler translates a source text that was created using the editor into a target language such as assembly or object code. High-level languages contain English-like commands that are readily understandable by the programmer. High-level languages normally combine a number of assembly-level statements into a single high-level statement. A compiler is used to translate the high-level languages such as Pascal, FORTRAN, or other languages into machine language. The advantages of high-level languages are ease of readability and maintainability. Also, the multiplicity of high-level languages increases the productivity of the programmer. This is assuming, of course, that the problem is better suited to being solved using such a language and not an assembly language.

Like a compiler, an interpreter usually processes a high-level language program. Unlike a compiler, an interpreter actually executes the high-level language program one statement at a time, rather than translating the entire program into a sequence of machine instructions to be run later. Interpreters may be microcoded firmware rather than software.

1.3.2 PROGRAMMING CONCEPTS

In general, programs are developed using machine, assembly, and high-level languages.

1.3.2.1 Machine Language Programming

One way to write a program is to assign a fixed binary pattern for each instruction and represent the program by sequencing these binary patterns. Such a program is called a machine language program or an object program. A typical example of a machine language program is shown on the following page. This program will readily run on a Z-80 based microcomputer. But the program is hard to understand and difficult to debug.
1.3.2 Assembly Language Programming

Program designers realized the importance of symbols in programming to improve readability and expedite the program development process. As a first step, they came up with the idea of giving a symbolic name for each instruction. These names are called mnemonics, and a program written using such mnemonics is called an assembly language program. Given below is a typical Z-80 assembly language equivalent for the previous machine language program:

```
LD A, 5 ; load A reg with 5
LD B, 10 ; load B reg with 10
ADD A, B ; (A) ← (A) + (B)
LD (100), A ; save the result in the location 100
HALT ; halt processing
```

From this example, it is clear that the usage of mnemonics (in our example LD, ADD, HALT are the mnemonics) has improved the readability of our program significantly.

An assembly language program cannot be executed by a machine directly, as it is not in binary form. Usually, we refer to a symbolic program as a source program. An assembler is needed in order to translate an assembly language (source) program into the object code executable by the machine. This is illustrated in Figure 1.18.

Assembly language provides improved readability, but the programmer needs to know the internal architecture of the microprocessor. The assembly language program written for one processor will not usually run another processor. The assembly language instructions include input/output instructions and, therefore, this programming method is normally used for writing I/O routines.

1.3.2c High-Level Language Programming

Typical examples of high-level languages include FORTRAN, BASIC, Pascal, and C. The program shown below is written in FORTRAN, in order to obtain the sum of the first N natural numbers:

```
READ (5, 10) N
10 FORMAT (I4)
NSUM = H*(N+1)/2
WRITE (2, 10) NSUM
STOP
END
```

A translator called a compiler is needed to translate a program written in a high-level language into binary form. Running a source program or a program written in a high-level language using a microcomputer is a two-stage process. This is explained in Figure 1.19.

In the first step, the compiler reads each line of the source program and produces an object program executable on a given target machine. In the second step, the object program is executed with the given data in order to produce results. Note that most compilers make several passes through the source code before translation is complete. In order to execute the same program many times with different data, there is no need to read the source program for the second run. In this way, a compiler improves the efficiency of the whole process. This idea works very well in a professional environment where one has to think of many production runs of the same program.

![Compilation process](image)

**FIGURE 1.19** Compilation process.
1.4 MICROCOMPUTER TYPICAL ADDRESSING MODES AND INSTRUCTIONS

In this chapter, some important characteristics and properties of microcomputer instruction sets are discussed. Topics include addressing modes and instruction types.

1.4.1 INTRODUCTION

An instruction manipulates the stored data, and a sequence of instructions constitutes a program. In general, an instruction has two components:

- Op-code field
- Address field(s)

The op-code field specifies how data are to be manipulated. The data items may reside within a microprocessor register or in the main memory.

The purpose of the address field is to indicate the data address. When operations require data to be read from or stored into two or more addresses, the address field may contain more than one address. For example, consider the following instruction:

```
ADD R1, R0
```

Assume that this microcomputer uses R1 as the source register and R0 as the destination register. The preceding instruction then adds the contents of registers R0 and R1 and saves the sum in register R0. The number and types of instructions supported by a microcomputer vary from one microcomputer to another and depend primarily on the architecture of a particular machine.

Depending on the number of addresses specified, one can have the following instruction formats:

- Three-address
- Two-address
- One-address
- Zero-address

1.4.2 ADDRESSING MODES

The sequence of operations that a processor has to carry out while executing an instruction is called its instruction cycle. One of the activities in an instruction cycle is the determination of the addresses of the operands involved in that instruction.

The way in which a processor accomplishes this task is called the addressing mode. Typical addressing modes supported by the instruction sets of popular processors will be examined.

An instruction is said to have an inherent addressing mode if its op-code indicates the address of the operand, which is usually the contents of a register. For example, consider the following instruction:

```
STC
```

Set the carry flag in the status register. Since the op-code implies the address of the operand, the processor does not have to compute the operand address. This mode is very common with 8-bit microprocessors such as the 8085, Z 80, and MC6809.

Whenever an instruction contains the operand value, it is called an
**Immediate mode instruction.** For example, consider the following instruction:

```
ADD $25, R1; R1 ← R1 + 25
```

In this instruction, the symbol $ indicates that it is an immediate-mode instruction. This convention is adopted in the assemblers for processors such as the MC6809 and MC68000. In these systems, the machine representation of this instruction occupies two consecutive memory words: the first word holds the op-code, whereas the next word holds the data value. (For the preceding case, it is 25.)

To execute this instruction, the processor has to access memory twice.

An instruction is said to have an **absolute addressing mode** if it contains the address of the operand. For example, consider the following move instruction:

```
MOVE 5000, R2; R2 ← (5000)
```

This instruction copies the contents of memory location 5000 in the register R2.

An instruction is said to have a **register mode** if it contains a register address as opposed to a memory address. For example, consider the following register mode and instruction:

```
ADD R2, R3; R3 ← R2 + R3
```

Whenever an instruction specifies a register that holds the address of an operand, the resulting addressing mode is known as the **register indirect** mode. From this definition, it follows that the EA of an operand in the register indirect mode is the contents of the register R. More formally, this result is written as follows:

```
EA = [R]
```

To illustrate this idea clearly, consider the following instruction:

```
MOVE (R2), (R3); [R3] ← [R2]
```

**1.4 Microcomputer Typical Addressing Modes and Instructions**

Assume that the following configuration exists:

```
[R2] = 5000\text{H}
[R3] = 4000\text{H}
[5000] = 1256\text{H}
[4000] = 4622\text{H}
```

This instruction copies the contents of the memory location, whose address is specified by the register R2, into the location whose address is specified by the register R3. Thus, after the execution of this instruction, the memory location 4000 will contain the value 1256.

**1.4.3 INSTRUCTION TYPES**

In general, instructions available in a processor may be broadly classified into five groups:

- Data transfer instructions
- Arithmetic instructions
- Logical instructions
- Program control instructions
- I/O instructions

Data transfer instructions are primarily concerned with data transfers between the microprocessor registers or between register and memory. An example is MOVE R0, R1 which transfers the contents of register R0 to register R1.

Typical arithmetic instructions include ADD and SUBTRACT instructions. For example, ADD R0, R1 adds the contents of R0 to R1 and stores the result in R1.

Logical instructions perform Boolean AND, OR, NOT, and EXCLUSIVE-OR operations on a bit-by-bit basis. An example is OR R0, R1 which logically ORs the contents of R0 with R1 and places the result in R1.

Typical program control instructions include unconditional and conditional branch and subroutine CALL instructions. For example, JMP 2035H unconditionally branches to the 16-bit address 2035H.

I/O instructions perform input and output operations. An example is IN PORTA which inputs the contents of an I/O port called port A into a microprocessor register such as the accumulator.
1.5 BASIC FEATURES OF MICROCOMPUTER DEVELOPMENT SYSTEMS

When the first 4- and 8-bit microprocessors appeared on the market, the hardware costs associated with a microprocessor system were so high that the labor costs (logic design, software, debugging) were of lesser impact on the total project cost. With the constantly decreasing cost of new microprocessor components, the labor costs of developing new products have taken on a more significant role. The microcomputer development system's main function is to simplify the product development stage and therefore obtain the maximum efficiency from the design team, whether it be a single engineer or a large team.

Development systems allow the parallel design of hardware and software so that the design can be tested and debugged before any actual hardware simulations occur. Some development systems allow the simulation of software without any of the final hardware being available to the software engineer.

Development systems fall into one of two categories: systems supplied by the device manufacturer and systems built by after-market manufacturers. The main difference between the two categories is the range of microprocessors that a system will accommodate. Systems supplied by the device manufacturer (Intel, Motorola, RCA) are limited to use for the particular chip set manufactured by the supplier. In this manner, an Intel development system may not be used to develop a Motorola-based system.

The other category contains systems that are more universal in usage. Software and emulation hardware are available for the universal systems (Tektronix, Hewlett-Packard, AMI) to develop most of the popular microprocessors.

Within both categories of development systems, there are basically three types available: single-user systems, time-shared systems, and networked systems. A single-user system consists of one development station that can be used by one user at a time. Single-user systems are low in cost and may be sufficient for small systems development. Time-shared systems usually consist of a "dumb"-type terminal connected by data lines to a centralized microcomputer-based system that controls all operations. A networked system usually consists of a number of smart Cathode Ray Tubes (CRTs) capable of performing most of the development work and can be connected over data lines to a larger central computer. The central microcomputer in a network system usually is in charge of allocating disk storage space and will download some programs into the user's work station computer. The networked development systems are becoming more popular because they allow more throughput of data which allows the user to develop the programs faster and more easily. The cost per station of networked systems is less than that of the single-user system as the number of stations increases.

A microcomputer development system is a combination of the hardware necessary for microprocessor design and the software to control the hardware. The basic components of the hardware are the central processor, the CRT terminal, mass storage device (floppy or hard disk), and usually an In-Circuit Emulator (ICE).

The central processor is the heart of the development system. Whether a microprocessor in a single-user system or a large processor in a timeshared multiuser system, the central processor is responsible for the overall control of the development system. In a single-user system, the central processor executes the operating system software, handles the Input/Output (I/O) facilities, executes the development programs (editor, assembler, linker), and allocates storage space for the programs in execution. In a large multiuser networked system the central processor may be responsible for mass storage allocation, while a local processor may be responsible for the I/O facilities and execution of development programs.

The CRT terminal provides the interface between the user and the operating system or program under execution. The user enters commands or data via the CRT keyboard and the program under execution displays data to the user via the CRT screen. In early development systems, CRTs were not used and a mechanical teletype was used to communicate with the development system. The current CRTs communicate with the operating system at speeds in excess of ten times the speed of the old teletype. The CRT of a single-user system may be connected directly to the central processor, while CRTs in time-shared systems may be connected to remote microcomputers via phone lines (using modems).

A disk drive of some kind is necessary for mass storage of user data. Some of the current complex operating systems and high-level development programs require large amounts of system Random Access Memory (RAM) to operate, and little memory is left available for user data such as source code for the program under development. For this reason, the operating system must allocate the available memory and store unused portions of the program on a disk. A common technique for memory management is the use of overlay files. An overlay is a portion of a program that is not always needed and is therefore not kept in the limited RAM
space. When the overlay is needed, it is read from the disk and overlays another program in memory that is no longer needed. The use of overlays and other memory management techniques allows the use of complex programs that would otherwise not be available for use on the smaller microcomputers in use today. The user's files, such as source code, are also stored on the disk.

Most of the single-user systems use a flexible (floppy) disk for mass storage, while the time-shared and networked systems tend to use a hard disk for mass storage. Cost and performance are directly related in the disk systems. The cost of flexible disk systems is low, but the storage capability is also low. The access time of a flexible disk (the delay time before the data are available after they have been requested) is also slow compared with hard disks or standard memory. The access time of a hard disk is much shorter than a flexible disk and the storage capability is much greater, but the higher cost of a hard disk system may be prohibitive to a small user. If a hard disk system is purchased, the cost can usually be returned in decreased development times.

Each program (whether system software or user program) is stored in an ordered format on the disk. Each separate entry on the disk is called a file. The operating system software contains the routines necessary to interface between the user and the mass storage unit. When the user requests a file by a specific file name, the operating system finds the program stored on disk by the file name and loads it into main memory. More advanced development systems contain memory management software that protects a user's files from unauthorized modification by another user. This is accomplished via a unique user identification code called USER ID. A user can only access files that have the user's unique code.

The equipment listed above comprises a basic development system, but most systems have other devices such as printers and PROM programmers attached. A printer is needed to provide the user with a hard copy record of the program under development. Many simple programs may be developed completely on the CRT, but a printer is needed to provide documentation necessary for advanced program development. After a program has been fully debugged and tested, the PROM programmer is used to load the machine code into a PROM for final prototype testing. Figure 1.21 shows a typical development system.

MODEM (Modulator-Demodulator) interfaces allow serial communication between systems using telephone lines. The modulator on the transmitting end converts the serial data into distinct tones which are demodulated on the receiver end back into serial data. MODEMs are also used to connect a user's CRT to a master development system at a remote location. Communication rates for MODEMs are usually limited to about 1200 bits per second and are therefore much slower than directly connected CRTs. Microcomputer development systems can be interfaced to a minicomputer via MODEMs to provide software development capabilities for many different microprocessors.

Program development and documentation call for the availability of printed listings of the programs. These listings are referred to as hard copy and are printed on the system line printer.

After the target system software has been completely developed and debugged, it needs to be permanently stored on a read-only memory (ROM or PROM. Erasable/Programmable Read Only Memories (EPROMs) are more generally used in system development as they can be erased and reprogrammed if the program changes. PROM programmers usually interface to circuits particularly designed to program a specific PROM. These interface boards are called personality cards and are available for all the popular PROM configurations.

Most development systems support one or more in-circuit emulators. The ICE is one of the most advanced tools for microprocessor hardware development. To use an ICE, the microprocessor chip is removed from the system under development (called the target processor) and the emulotor plugged into the socket that the processor was removed from. The ICE will functionally and electrically operate identically to the target processor with the exception that the ICE is under the control of development software. In this manner the development system may exercise the hardware that is being designed and monitor all status information available about the operation of the target processor. Using an ICE, processor register contents may be displayed on the CRT and operation of the hardware.
observed in a single-stepping mode. In-circuit emulators can find hardware and software bugs quickly that might take many hours using conventional hardware testing methods.

Architectures for development systems can be generally divided into two categories: the master/slave configuration and the single-processor configuration. A master/slave configuration, the master (host) processor controls all development functions such as editing, assembling, and so on. The slave processor controls the mass storage device and processes all I/O (CRT, printer).

The software for the development systems is written for the master processor which is usually not the same as the slave (target) processor. The pin connector (the number varies with the processor) which links the slave processor to the master processor.

Some development systems such as the HP 64000 completely separate the system bus from the emulation bus and therefore use a separate block of memory for emulation. This separation allows passive monitoring of the process. A benefit of the separate emulation facilities is that the test processor is used for editing, assembling, and so on, while the slave process continues the emulation. A designer may therefore start an return to the emulation program.

Another advantage of the separate bus architecture is that an operating system need not be written only once for the master processor and will be the slave processor is to be emulated, only the emulator needs to be changed.

A disadvantage of the master/slave architecture is that memory is not continuous because it is in isolated blocks. This separation makes it a higher cost associated with the master/slave architecture is a disadvantage, the HP 64000 system, the increased throughput may make the master/processor architecture, only one processor is used for system operation and target emulation.

The single processor does both jobs of executing system software as well as acting as the target processor. Since there is only one processor involved, the system software must be rewritten for each type of processor that is to be emulated. Since the system software must reside in the same memory used by the emulator, not all memory will be available to the emulation process, which may be a disadvantage when large prototypes are being developed.

System cost is lower because less hardware is required, but, as mentioned in the section on master/slave systems, the cost is a viable factor only if a single development facility is needed.

The hardware for a development system is necessary, but would be useless without the proper development software. As the developments become more and more advanced, the operating systems reduce many of the routine tasks from the user, making the user's time more productive. Many development systems are self-promoting, freeing the user from the responsibility of looking up commands and procedures in operating manuals.

The main programs necessary to microprocessor development are the operating system, editor, assembler, linker, compiler, and debugger.

The operating system is responsible for executing the user's commands. The operating system (such as CP/M, MP/M, UNIX) handles I/O functions, memory management, and loading of programs from mass storage into RAM for execution. The larger the development system, the larger and more capable the operating system.

The editor is usually the first program used in developing the software for the system being developed. The editor allows the user to enter the source code (either assembly language or some high-level language) into the development system. The editor includes commands that facilitate changing (additions or deletions) the source code whenever necessary.

The editor functions start by loading a segment of the input file into the development system RAM. The program segment is displayed to the user in some manner and the user edits the segment to make any needed changes. When the changes for a segment are complete, the segment is stored back on disk and the next segment loaded into RAM.

There are two types of editors currently in use: the line-oriented editor and the character-oriented editor.

The first editors developed were line-oriented editors. Line editors have fewer features than character editors and are therefore less expensive. Line editors are limited mostly to home computer systems and are rarely found on modern development systems. The main difference between line editors and character editors is that line editors do not display any changes in the file until the user prints the file, whereas character editors will display the change when the user makes the change.

Modern line-oriented editors allow the user to enter lines without line numbers. The user enters lines by positioning a pointer to the location in the program where the new line is desired and the new line is typed into
RAM. The user may also position the pointer to locations within a line, thus allowing characters to be added or deleted without retying the entire line. This type of line editor supports more advanced features than the line number-based line editor, but is still inferior to the character-oriented editor.

Almost all current microprocessor development systems use the character-oriented editor, more commonly referred to as the screen editor. The editor is called a screen editor because the text is dynamically displayed on the screen and the display automatically updates any edits made by the user. The screen editor uses the pointer concept to point to the character(s) that need editing. The pointer in a screen editor is called the cursor and special commands allow the user to position the cursor to any location displayed on the screen. Other commands allow the user to scroll the display forward or backward to allow display of any segment of the file. When the cursor is positioned, the user may insert characters, delete characters, or simply type over the existing characters. Complete lines may be added or deleted using special editor commands. By placing the editor in the insert mode, any text typed will be inserted at the cursor position when the cursor is positioned between two existing lines. If the cursor is positioned on a line to be deleted, a single command will remove the entire line from the file.

Search commands allow the user to search the file for occurrences of specific characters or text called strings. The user can specify a string of text and the editor will compare the file text for any matches with the string and then display the segment of text that holds the matched string. A similar function of find and replace is also available where the user can specify one string that is to be found and another string that is to replace the first string. This feature is extremely valuable for correcting misspelled words or phrases.

Other useful commands allow portions of other files to be merged into the current file, move segments to other positions in the file, delete old files, copy files, and perform many other functions that speed up the process of writing and debugging source code.

Screen editors implement the editor commands in different ways. Some editors use dedicated keys to provide some cursor movements. The cursor keys are usually marked with arrows to show the direction of cursor movement. Other special editor functions are accomplished with the use of control keys. A control key is activated by pressing a special key (usually labeled control or CTRL) simultaneously with a normal key. This combination of key strokes creates a new character that is sent to the editor program.

More advanced editors (such as the HP 64000) use soft keys. A soft key

is an unmarked key located on the keyboard directly below the bottom of the CRT screen. The mode of the editor decides what functions the keys are to perform. The function of each key is displayed on the screen directly above the appropriate key. The soft key approach is valuable because it frees the user from the problem of memorizing many different special control keys. The soft key approach also allows the editor to reassign a key to a new function when necessary.

The source code generated on the editor is stored as ASCII characters and cannot be executed by a microprocessor. Before the code can be executed, it must be converted to a form acceptable by the microprocessor. An assembler is the program used to translate the assembly-language source code generated with an editor into object code or machine code which may be executed by a microprocessor.

Assemblers recognize four fields on each line of source code. The fields consist of a variable number of characters and are identified by their position in the line. The fields, from left to right on a line, are the label field, the mnemonic or op-code field, the operand field, and the comment field. Fields are separated by characters called delimiters which serve as a flag to the assembler that one field is done and the next one is to start. Typical delimiters and their uses are:

- **space** used to separate fields
- **TAB** used to separate fields
- **;** used between addresses or data in the operand field
- **;** used before a comment statement
- **;** used after a label

A few typical lines of 8088 source code are:

<table>
<thead>
<tr>
<th>LABEL</th>
<th>OP CODE</th>
<th>OPERAND</th>
<th>COMMENT FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>A, 5</td>
<td>LOAD A 5 INTO A</td>
<td></td>
</tr>
<tr>
<td>DCR</td>
<td>A</td>
<td>DECREMENT THE COUNT</td>
<td></td>
</tr>
<tr>
<td>JNZ</td>
<td>LOOP</td>
<td>REPEAT LOOP IF COUNTER NOT ZERO</td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td></td>
<td>RETURN TO CALLER WHEN LOOP DONE</td>
<td></td>
</tr>
</tbody>
</table>

As can be seen in the above example, tab keys are used instead of spaces to separate the fields to give a more spread-out line which is easier to read during debugging.

In order for the assembler to differentiate between numbers and labels, specific rules are set up which apply to all assemblers. A label must start with
Introduction

Chapter 1

Diately before the subroutine to direct the assembler to alter the internal program counter:

ORG 2000H ; SET PROGRAM COUNTER TO 2000H
SUB MVI A, 5 ; SUBROUTINE PUTS A 5 IN 'A'
RET ; RETURN TO CALLER

Most assemblers will assume a starting address of zero if no ORG statement is given in the source code.

b. EQUATE (EQU). The EQU instruction is used to assign the data value or address in the operand field to the label in the label field. The EQU instruction is valuable because it allows the programmer to write the source code in symbolic form and not be concerned with the numeric value needed. In some cases, the programmer is developing a program without knowing what addresses or data may be required by the hardware. The program may be written and debugged in symbolic form and the actual data added at a later time. Using the EQU instruction is also helpful when a data value is used several times in a program. If, for example, a counter value was loaded at ten different locations in the program, a symbolic label (such as COUNT) could be used and the label count defined at the end of the program. By using this technique, if it is found during debugging that the value in COUNT must be changed, it need only be changed at the EQU instruction and at each of the ten locations where it is used in the program.

As an example of EQU, consider the following 8085 code:

PORTA EQU 00 ; ASSIGN A DUMMY VALUE
PORTB EQU 00 ; ANOTHER DUMMY VALUE
MVI A, 0FFH ; DATA TO BE SENT TO OUTPUTS
OUT PORTA ; INITIALIZE PORT A
OUT PORTB ; AND PORT B TOO

In the example, the programmer does not know the hardware addresses of output ports A and B, but may still write the source code. When the I/O addresses are known, they may be used to replace the 00 in the EQU statement.

c. DEFINE BYTE (DEFB or DB). The DB instruction is used to set a memory location to a specific data value. The DB instruction is usually used to create data tables or to preset a flag value used in a program. As the name implies, the DB instruction is used for creating an 8-bit value.
Introduction

For example, if a table of four values, 45H, 34H, 25H, and 0D3H, had to be created at address 2000H, the following code could be written:

```
ORG 2000H  ; SET TABLE ADDRESS
TABLE DB 44H, 34H, 25H, 0D3H  ; PRESET TABLE VALUES
```

The commas are necessary for the assembler to be able to differentiate between data values. When the code is assembled, the machine code would appear as follows:

```
... 2000 45
2001 34
2002 25
2003 0D3
... 
```

d. **DEFINE WORD (DEFW or DW).** Similarly to DB, DW defines memory locations to specific values. As the name implies, the memory allotted is in word lengths which are usually 16 bits wide. When assigning a 16-bit value to memory locations, two 8-bit memory locations must be used. By convention, most assemblers store the least significant byte of the 16-bit value in the first memory location and the most significant byte of the 16-bit value in the next memory location. This technique is sometimes referred to as Intel style, because the first microprocessors were developed by Intel, and this storage method is how Intel processors store 16-bit words.

Data tables may be created with the DW instruction, but care must be taken to remember the order in which the 16-bit words are stored. For example, consider the following table:

```
ORG 2500H
DATA DW 4000H, 2500H, 0BCH
```

The machine code generated for this table would appear as follows:

```
... 2500 00
2501 40
2502 00
2503 2B
2504 CA
2505 0B
...
```

Chapter 1

1.5 Basic Features of Microcomputer Development Systems

2. TITLE. TITLE is a formatting instruction that allows the user to name the program and have the name appear on the source code listing. Consider the following line:

```
TITLE 'MULTIPLICATION ROUTINE'
```

When the assembler generates the program listing, each time it starts a new page the title MULTIPLICATION ROUTINE appears at the top of each page.

3. PAGE (also called EJECT). PAGE is another formatting instruction that causes the assembler to skip to the next page. The PAGE instruction is used to keep subroutines or modules of source code on separate pages which makes the source code easier to read and debug.

4. SPACE. The SPACE instruction simply instructs the assembler to skip a line.

5. END. The END pseudoinstruction signals to the assembler that the source code is complete. Any lines after the END statement will be ignored by the assembler. Some assemblers require an END statement while some assemblers merely assume an END after the last line of the source code has been processed.

6. LIST. The LIST instruction is a direct command to the assembler and will cause the assembler to print the entire source code program.

Several types of assemblers are available, the most common of which are the one-pass assembler, the two-pass assembler, the macroassembler, cross assemblers, resident assemblers, and the metaassembler.

a. **One-Pass Assembler.** The one-pass assembler was the first type to be developed and is therefore the most primitive. Very few systems use a one-pass assembler because of the inherent problem that only backward references may be used.

In a one-pass assembler the source code is processed only once. As the source code is processed, any labels encountered are given an address and stored in a table. Therefore, when the label is encountered again, the assembler may look backward to find the address of the label. If the label has not been defined yet (for example, a jump instruction that skips forward), the assembler issues an error message.

Since only one pass is used to translate the source code, a one-pass
assembler is very fast, but because of the forward reference problem the one-pass assembler is seldom used.

b. Two-Pass Assembler. The two-pass assembler is similar in operation to the one-pass assembler with one important difference. The first pass made through the source code is specifically for the purpose of assigning an address to all labels. When all addresses have been stored in a table with their appropriate addresses, a second pass is made to actually translate the source code into machine code.

The two-pass style assembler is the most popular type of assembler currently in use.

c. Macroassembler. A macroassembler is a type of two-pass assembler that allows the programmer to write the source code in macro. A macro is a sequence of instructions that the programmer gives a name. Whenever the programmer wishes to duplicate the sequence of instructions, the macro name is inserted into the source code.

A macro routine is not the same as a subroutine call in assembly language. An assembly language call is a function of the microprocessor. When the CALL (8085) instruction is encountered in the machine code, the 8085 will save the address of the next sequential instruction to be executed and alter the program counter to the address given in the CALL instruction. When a macro is used, it is a function of the assembler and not the microprocessor. Every time the macro name is encountered in the source file, the macro assembler substitutes the macro sequence in place of the macro name.

Conditional assembly is included with most macroassemblers (as well as with most two-pass assemblers) and is very useful under certain conditions. In some cases a program is being written which will execute on two or more hardware systems which are similar, but have minor differences. Instead of writing a customized program for each system, the same program may be used with conditional assembly. A flag is defined in the source code as either true or false depending on which system is going to execute the program. A conditional statement is then inserted at the locations in the program where different actions must be taken, depending on the hardware.

d. Cross Assembler. A cross assembler may be any of the types already mentioned. The distinguishing feature of a cross assembler is that it is not written in the same language used by the microprocessor that will execute the machine code generated by the assembler.

Cross assemblers are usually written in a high-level language such as FORTRAN which will make them machine independent. For example, an 8085 assembler may be written in FORTRAN and then the assembler may be executed on another machine such as the Motorola 6800. Microprocessors will provide cross assemblers to develop machine code for one of their microprocessors using their own development system that uses another microprocessor.

e. Resident Assembler. A resident assembler is almost the complete opposite of the cross assembler because it is written to run on the same machine that will execute the source code. For example, an 8085 assembler that is written in 8085 assembly language is a resident assembler.

f. Metaassembler. The most powerful assembler is the metaassembler because it will support many different microprocessors. The programmer merely specifies at the start of the source code which microprocessor assembly language will be used and the metaassembler will translate the source code to the correct machine code.

The output file from most development system assemblers is an object file. The object file is usually relocatable code that may be configured to execute at any address. The function of the linker is to convert the object file to an absolute file which consists of the actual machine code at the correct address for execution. The absolute files thus created are used for debugging and finally for programming PROMs.

A compiler will take the source code written in a high-level language (such as FORTRAN, COBOL, PL/M, Pascal) and translate it into machine language.

Interpreters are another form of high-level language but they do not convert to assembly language. The interpreter (such as BASIC) is a program that is written for the system and the interpreter directly executes the high-level statements one at a time.

Debugging a microprocessor-based system may be divided into two categories: software debugging and hardware debugging. Both debug processes are usually carried out separately from each other because software debugging can be carried out on an emulator without having the final system hardware.

The usual software development tools provided with the development system are:

- Single-step facility
- Breakpoint facility
- Simulator program

37.jpg
A single-stepper simply allows the user to execute the program being debugged one instruction at a time. A single-stepper allows the user to follow program logic one step at a time, examining all processor information before allowing the next instruction to be executed. During debugging it is usually helpful to examine the state of the data bus, the address bus, the status flags, and the control lines. By examining the state of the processor during each step, the debugger can detect such program faults as incorrect jumps, incorrect addressing, erroneous op codes, and so on.

A breakpoint allows the user to execute an entire section of a program being debugged.

There are two types of breakpoint systems: hardware and software. The hardware breakpoint uses hardware to monitor the system address bus and detect when the program is executing the desired breakpoint location. When the breakpoint is detected, the hardware uses the processor control lines to either halt the processor for inspection or cause the processor to execute an interrupt to a breakpoint routine. Hardware breakpoints can be used to debug both ROM- and RAM-based programs. Software breakpoint routines may only operate on a system with the program in RAM because the breakpoint instruction must be inserted into the program that is to be executed.

Breakpoint routines are extremely useful for debugging programs because they allow the user to execute a large portion of the program before stopping execution. An entire subroutine may be executed and then halted to examine the processor registers to see if the subroutine is executed properly. When the breakpoint is no longer needed, it may be removed and the program will execute normally.

Most debuggers combine single-stoppers and breakpoint routine to provide a complete package. The user may insert a breakpoint (either manually or automatically, depending on the system) at the desired point and let the program execute up until that point. When the program stops at the breakpoint the user may use a single-stepper to then examine the program one instruction at a time.

A simulator is a program that allows the execution of a program to be simulated while keeping track of all parameters (address, data, flags) of the target processor. The processor that executes the simulator does not need to be of the same type as the target processor because the program to be debugged is not actually executed by the development system processor. The simulator operates by taking one instruction at a time and then simulating all actions that the target processor would perform for that instruction.

Simulators are usually very large programs which will include all the software debugging tools (single-stepper, breakpoints) necessary in a complete software development program.

There are two main hardware debugging tools: the logic analyzer and the in-circuit emulator.

Logic analyzers are usually used to debug hardware faults in a system. The logic analyzer is the digital version of an oscilloscope because it allows the user to view logic levels in the hardware.

In-circuit emulators can be used to debug and integrate software and hardware.

Two other features associated with a typical microcomputer development system are disassembly and command files.

Disassembly is the reverse process of assembly and is generally used during debugging to verify that the hex machine code being executed actually matches the assembly language that the user thinks is being executed. Most emulators and debuggers have a disassembler built into the program. A typical disassembly of a section of machine code may appear as follows:

```
0100 21 00 05 LXI H, 0500
0103 06 05 MVI, B05
0105 AF XRA A
```

By using the disassembly feature of a debugger, some bugs may be found without actually referring to the source code listing because the code may be displayed in assembly language form on the CRT.

A command file is a file created by the user that contains a sequence of commands that are to be executed. Instead of the programmer entering an identical sequence of commands over and over while debugging a program, the programmer uses a command file to perform the operations. A typical command file may hold the commands necessary to load a program into an emulator, set the emulator parameters, and execute the program. A command file saves the programmer time while at the same time providing repeatability by removing the possibility of operator error from command entry.

1.6 SYSTEM DEVELOPMENT FLOWCHART

The total development of a microprocessor-based system typically involves three phases: software design, hardware design, and program
diagnostic design. A systems programmer will be assigned the task of writing the application software, a logic designer will be assigned the task of designing the hardware, and typically both designers will be assigned the task of developing diagnostics to test the system. For small systems, one engineer may do all three phases, while on large systems several engineers may be assigned to each phase. Figure 1.22 shows a flowchart for the total development of a system. Notice that software and hardware development may occur in parallel to save time.

1.6.1 SOFTWARE DEVELOPMENT

The first step in developing the software is to take the system specifications and write a flowchart to accomplish the desired tasks that will implement the specifications. The flowchart should be properly structured to ensure efficient program flow.

The assembly language or high-level source code may now be written from the system flowchart. Individual software designers may be given portions of the flowchart to develop individual modules that will be linked together after they are debugged. As much hardware information as possible should be made available at this point so that hardware-dependent portions of the program may be entered with the proper data.

After the source code has been written, it may be entered into the editor. Labels should be used whenever possible and symbols should be used in place of direct data.

Once the complete source code has been entered, it may be assembled. The assembler will check for syntax errors and print error messages to help in the correction of errors. Many advanced assemblers will assemble source code for many different microprocessors. In order to identify which microprocessor is to be used, a special command is entered in the source code to tell the assembler what microprocessor is in use.

The normal output of an assembler is the object code and a program listing. The object code will be used later by the linker. The program listing may be sent to a disk file for use later or it may be directed to the printer I/O if a printer is available.

The linker can now take the object code generated by the assembler and create the final absolute code that will be executed on the target system. The emulation phase will take the absolute code and load it into the development system RAM. From here, the program may be tested using breakpoints or single-stepping. Single-stepping allows the designer to watch the program execute one instruction at a time and monitor program status. During single-stepping, registers may be displayed to detect any program problems that may occur.

FIGURE 1.22 Microprocessor system development flowchart.
1.6.2 HARDWARE DEVELOPMENT

Working from the system specifications, a block diagram of the hardware must be developed. The block diagram will include all major hardware sections and show how they interconnect. The logic diagram may now be drawn using the block diagram as a guide. Great care should be taken to design a system that uses the most efficient logic available. Minimum parts count may be important if space is limited. In a system where cost is a limiting factor, parts count may be increased if less expensive parts are available which will perform the same function.

A wiring list and parts layout diagram is then made from the logic diagram. Care should be taken to ensure that all pin connections are correct, for in most cases the design engineer will not be constructing the prototype and will therefore not be able to detect any errors during construction. A prototype may now be constructed and tested for wiring errors.

When the prototype has been constructed it may be debugged for correct operation using standard electronic testing equipment such as oscilloscopes, meters, logic probes, and logic analyzers.

After the prototype has been debugged electrically, the development system in-circuit emulator may be used to check it functionally. The ICE will verify addressing, correct I/O operation, and so on.

The next step in the system development is to validate the complete system by running operational checks on the prototype with the finalized application software installed. The EPROM is then programmed with the error-free programs.

1.7 TYPICAL PRACTICAL APPLICATIONS

Microprocessors are being extensively used in a wide variety of applications. Typical applications include dedicated controllers, personal workstations, and real-time robotics control. Some of these applications will be described in the following.

1.7.1 TWO-POSITION CONTROLLER

The control law of the two-position controller can be stated as:

\[
\text{Valve output} = \begin{cases} 
100\% & \text{for } E > 0 \\
0\% & \text{for } E < 0 
\end{cases}
\]

where the error \( E = PV - SP \), and \( PV \) is the process variable measurement, and \( SP \) is the set point.

The above relationship means that when the measured value is greater than the set point, the two-position controller must completely open the valve. When it is smaller than the set point, the controller must close the valve. An example is a home furnace which is either turned ON or OFF depending on the "opened" or "closed" status of a switch controlled by a thermostat, the temperature sensor. When the temperature drops below the set point, the switch is closed, the furnace is turned on, and the temperature rises. When the temperature is higher than the set point, the controller opens the switch and thus the furnace is turned OFF. The up and down cycling continues indefinitely. To avoid cycles from changing too fast and thus becoming destructive to the equipment, a gap called the neutral zone is built into the controller so that nothing happens until the temperature passes the set point and goes beyond the neutral zone boundaries. A microcomputer can be used as the two-position controller.

The following example illustrates this concept. Consider Figure 1.2.3.

![Two-Position Controller Diagram](image)

The cylindrical tank in the figure is emptied by a constant flow. This outflow is maintained by controlling the valve, \( V_1 \).

The microprocessor-based two-position controller opens the valve \( V_1 \) if the level measured by the level sensor drops below 19 meters and closes the valve if the level exceeds 21 meters. The neutral zone is 2 meters.

The level sensor measures the level and converts it to electrical analog...
voltage. The amplifier is used to amplify this voltage to make it compatible for analog-to-digital conversion. The program in the microcomputer inputs this digital (binary) data and compares it with the lower and upper bounds of the neutral zone. The microcomputer then outputs HIGH or LOW to the buffer. The buffer is used to amplify the current for making the microcomputer output compatible for turning the valve V₁ solenoid (relay) ON or OFF.

### 1.7.2 PERSONAL WORKSTATIONS

Personal workstations are designed using the high-performance 16- and 32-bit microprocessors. A dedicated single user (rather than multiple users sharing resources of a single microcomputer) can obtain significant computing power from these workstations.

The workstations currently under production are using 32-bit microprocessors to provide certain sophisticated functions such as IC layout, 3D graphics, and stress analysis. Typical workstations designed using an 8- and 16-bit microprocessor cost about $10,000 per machine. With 32-bit microprocessors, the cost of the workstations can be lowered since the functions performed by discrete components and special-purpose hardware will be performed by the high-performance 32-bit microprocessor.

### 1.7.3 FAULT-TOLERANT SYSTEMS

In many applications such as control of life-critical systems, control of nuclear waste, and unattended remote system operation, the reliability of the hardware is of utmost importance. The need for such a reliable system resulted in fault-tolerant systems. These systems use redundant computing units to provide reliable operation. However, the cost of fault-tolerant systems can be very high if the performance requirements of the application need high-performance VAX-type computers. Since the performance levels of 32-bit microprocessors are comparable to the VAX-type computer, use of multiple 32-bit microprocessors in a redundant configuration will outperform the VAX. Thus, the 32-bit microprocessors will provide efficient fault-tolerant systems.

### 1.7.4 REAL-TIME CONTROLLERS

Real-time controllers such as flight-control systems for aircraft, flight simulators, and automobile engine control require high-performance computers. Some of these applications were handled in the past by using mainframe computers which resulted in high cost and the controllers occupied large spaces.

Next-generation flight simulators currently under production use multiple 32-bit microprocessors to perform graphic manipulation, data gathering, and high-speed communications. Obviously, an application such as real-time automobile engine control using mainframe computers is not practical since these systems are not small enough to fit under the car hood. These controllers are currently being designed using the small-sized 32-bit microprocessors to perform high-speed data manipulation and calculation.

### 1.7.5 ROBOTICS

The processing requirements of complex robots attempting to emulate human activities exceed the capabilities of 8- and 16-bit microprocessors. With 32-bit microprocessors, it is now feasible to design these controllers at low cost. In many cases, the microprocessor is used as the brain of the robot. In a typical application, the microprocessor will input the actual arm angle measurement from a sensor, compare it with the desired arm angle, and will then send outputs to a motor to position the arm.

Mitsubishi recently announced the first 68020-based system robot control system.

### 1.7.6 SUMMARY

Microcomputers, therefore, are being used in diversified applications. With improvements in speed of memory and I/O devices in the future, high-speed real-time applications will be more feasible with 16- and 32-bit microprocessors.

## QUESTIONS AND PROBLEMS

1.1 What is the basic difference between the microprocessor and the microcomputer?

1.2 What is meant by an 8-bit microprocessor and a 16-bit microprocessor?

1.3 What is the main advantage of a barrel shifter?

1.4 Determine the carry, sign, overflow, and zero flags for the following operation:

\[ 23H_{16} \text{ ADD } 82A_{16} \]
Chapter 3

INTEL 8086

This chapter describes the internal architecture, addressing modes, instruction set, and I/O techniques associated with the 8086 microprocessor. Interfacing capabilities to typical memory and I/O chips such as the 2716, 2142, and 8255 are included.

A design technique is presented showing interconnection of the 8086 to 2716 EPROM, 2142 RAM, and 8255 I/O chips. The memory and I/O maps are then determined.

3.1 INTRODUCTION

The 8086 is Intel's first 16-bit microprocessor. Its design is based on the 8080 but it is not directly compatible with the 8080.

The 8086 is designed using the NMOS technology and contains approximately 29,000 transistors. The 8086 is packaged in a 40-pin DIP (Dual In-line Package) and requires a single 5V power supply. The 8086 can be operated at three different clock speeds. The standard 8086 runs at 5 MHz internal clock frequency, whereas the 8086-2 and 8086-4 run at internal clock frequencies of 8 and 4 MHz, respectively. An external clock generator/driver chip such as the Intel 8284 is needed to generate the 8086 clock input signal. The 8086 divides the external clock connected at the CLK pin internally by three. This means that for a 5-MHz internal clock, the 8284 must generate a 15-MHz clock output which should be connected at the 8086 CLK pin.

The 8086 has a 20-bit address and, hence, it can directly address up to one megabyte (2^20) of memory. The 8086 uses a segmented memory. An interesting feature of the 8086 is that it prefetches up to six instructions
bytes from memory and queues them in order to speed up instruction execution.

The memory of an 8086-based microcomputer is organized as bytes. Each byte can be uniquely addressed with 20-bit addresses of 00000h to 00007h, 00008h to 00015h, ..., EFFFFFFh. An 8086 16-bit word consists of any two consecutive bytes; the low-addressed byte is the low byte of the word and the high-addressed byte contains the high byte as follows:

<table>
<thead>
<tr>
<th>Low byte of the word</th>
<th>High byte of the word</th>
</tr>
</thead>
<tbody>
<tr>
<td>07h</td>
<td>16h</td>
</tr>
</tbody>
</table>

Address 00520h is 2607h. The 16-bit word stored at the even address 00520h is 2607h. Next consider a word stored at an odd address as follows:

<table>
<thead>
<tr>
<th>Low byte of the word</th>
<th>High byte of the word</th>
</tr>
</thead>
<tbody>
<tr>
<td>05h</td>
<td>1Fh</td>
</tr>
</tbody>
</table>

Address 01257h is 3F05h. The 16-bit word stored at the odd address 01257h is 3F05h. Note that for word addresses, the programmer uses the low-order address (odd or even) to specify the whole 16-bit word.

The 8086 always accesses a 16-bit word to or from memory. The 8086 can read a 16-bit word in one operation if the first byte of the word is at an even address. On the other hand, the 8086 must perform two memory accesses to two consecutive memory even addresses if the first byte of the word is at an odd address. In this case, the 8086 discards the unwanted bytes of each. For example, consider MOV BX, ADDR. Note that the X or H (or L) following the 8086 register name in an instruction indicates whether the transfer is 16-bit or 8-bit. For example, MOV ADDR, BX moves the contents of the 20-bit physical memory location addressed by ADDR into the 8086 16-bit register BX. Now, if ADDR is a 20-bit even address such as 30024h, then this MOV instruction loads the low (BL) and high (BH) bytes of the 8086 16-bit register BX with the contents of memory locations 30024h and 30025h, respectively, in a single access. Now, if ADDR is an odd address such as 40005h, then the MOV BX, ADDR instruction loads BL and BH with the contents of memory locations 40005h and 40006h, respectively, in two accesses. Note that the

8086 accesses locations 40004h and 40005h in the first operation but discards the contents of 40004h and in the second operation accesses 40006h and 40007h, but ignores the contents of 40007h. Next, consider a byte move such as MOV BH, ADDR. If ADDR is an even address such as 50002h, then this MOV instruction accesses both 50002h and 50003h, but loads BH with the contents of 50002h and ignores the contents of 50003h. However, if ADDR is an odd address such as 50003h, then this MOV loads BH with the contents of 50003h and discards the contents of 50002h.

The 8086 family consists of two types of 16-bit microprocessors — the 8086 and 8088. The main difference is how the processors communicate with the outside world. The 8088 has an 8-bit external data path to memory and I/O, while the 8086 has a 16-bit external data path. This means that the 8088 will have to do two read operations to read a 16-bit word into memory. In most other respects, the processors are identical. Note that the 8086 accesses memory in bytes. No alterations are needed to run software written for one microprocessor on the other. Because of similarities, only the 8086 will be considered here. The 8086 is used in designing the IBM Personal computer.

An 8086 can be configured as a small microprocessor system (minimum mode if the MN/MX pin is tied to HIGH) or as a multiprocessor system (maximum mode when MN/MX pin is tied to LOW). In a given system, the MN/MX pin is permanently tied to either HIGH or LOW. Some of the 8086 pins have dual functions depending on the selection of the MN/MX pin level. In the minimum mode (MN/MX pin high), these pins transfer control signals directly to memory and input/output devices. In the maximum mode (MN/MX pin low), these same pins have different functions which facilitate multiprocessor systems. In the maximum mode, the control functions normally present in minimum mode are assumed by a support chip, the 8288 bus controller.

Due to technological advances, Intel introduced the high performance 80186 and 80188 which are enhanced versions of the 8086 and 8088, respectively. The 8-MHz 80186/80188 provides two times greater throughput than the standard 5-MHz 8086/8088. Both have integrated several new peripheral functional units such as a DMA controller, a 16-bit timer unit, and an interrupt controller unit into a single chip. Just like the 8086 and 8088, the 80186 has a 16-bit data bus and the 80188 has a 8-bit data bus; otherwise, the architecture and instruction set of the 80186 and 80188 are identical. The 80186/80188 has an on-chip clock generator so that only an external crystal is required to generate the clock. The 80186/80188 can operate at either 6 or 8 MHz internal clock. Like the 8086, the crystal frequency is divided by 2 internally. In other words, external crystals
of 12 or 16 MHz must be connected to generate the 6- or 8-MHz internal clock frequency. The 80186/80188 is fabricated in a 68-pin package. Both processors have on-chip priority interrupt controller circuits to provide five interrupt pins. Like the 8086/8088, the 80186/80188 can directly address one megabyte of memory. The 80186/80188 is provided with 10 new instructions beyond the 8086/8088 instruction set. Examples of these instructions include INs and OUTs for inputting and outputting string byte or string word. The 80286, on the other hand, has added memory protection and management capabilities to the basic 8086 architecture. An 8-MHz 80286 provides up to six times greater throughput than the 5-MHz 8086. The 80286 is fabricated in a 68-pin package. The 80286 can be operated at 4, 6, or 8 MHz clock frequency. An external 82284 clock generator chip is required to generate the clock. The 80286 divides the external clock by two to generate the internal clock. The 80286 is typically used in a multitasking or multithreading system. The 80286 can be operated in two modes, real address and protected virtual address. The real address mode emulates a very high performance 8086. In this mode, the 80286 can directly address one megabyte of memory. The 80286, in the virtual address mode, can directly address 16 megabytes of memory. The virtual address mode provides (in addition to the real address mode capabilities) virtual memory management, task management, and protection. The programmer can select one of these modes by loading appropriate data in the 16-bit machine status word (MSW) register by using load and store instructions. Two examples of these instructions are LMSW (Load MSW register) and SMSW (Store MSW register). The 80286 is used as the CPU of the IBM PC/AT Personal computer. An enhanced version of the 80286 is at the 32-bit 80386 microprocessor which will be covered later. The 80386 is used as the CPU in the IBM's newest PC.

3.2 8086 ARCHITECTURE

Figure 3.1 shows a block diagram of the 8086 internal architecture. As shown in the figure, the 8086 microprocessor is internally divided into two separate functional units. These are the Bus Interface Unit (BIU) and the Execution Unit (EU). The BIU fetches instructions, reads data from memory and ports, and writes data to memory and I/O ports. The EU executes instructions that have already been fetched by the BIU. The BIU and EU function independently. The BIU interfaces the 8086 to the outside world. The BIU provides all external bus operations. The BIU contains segment registers, instruction pointer, instruction queue, and address generation/bus control circuitry to provide functions such as fetching and queuing of instructions, and bus control.

The BIU's instruction queue is a First-In First-Out (FIFO) group of registers in which up to six bytes of instruction code are prefetched from memory ahead of time. This is done in order to speed up program execution by overlapping instruction fetch with execution. This mechanism is known as pipelining. If the queue is full and the EU does not request BIU to access memory, the BIU does not perform any bus cycle. On the other hand, if the BIU is not full and if it can store at least two bytes and the EU does not request it to access memory, the BIU may prefetch instructions. However, if BIU is interrupted by EU for memory access

![Diagram](image_url)
while the BIU is in the process of fetching an instruction, the BIU first completes fetching and then services the EU; the queue allows the BIU to keep the EU supplied with prefetched instructions without tying up the system bus. If an instruction such as jump or subroutine call is encountered, the BIU will reset the queue and begin prefilling after passing the new instruction to the EU.

The BIU contains a dedicated adder which is used to produce the 20-bit address. The bus control logic of the BIU generates all the bus control signals such as read and write signals for memory and I/O.

The BIU has four 16-bit segment registers. These are the Code Segment (CS) register, the Data Segment (DS) register, the Stack Segment (SS) register, and the Extra Segment (ES) register. The 8086's one megabyte memory is divided into segments of up to 64K bytes each. The 8086 can directly address four segments (256K byte within the 1 Mbyte memory) at a particular time. Programs obtain access to code and data in the segments by changing the segment register contents to point to the desired segments. All program instructions must be located in memory pointing to the 16-bit CS register with a 16-bit offset in the segment contained in the 16-bit instruction pointer (IP). The BIU computes the 20-bit physical address internally using the program-issued logical address (16-bit contents of CS and IP) by logically shifting the contents of CS four bits to the left and then adding the 16-bit contents of IP. In other words, the CS is multiplied by 16, and the BIU computes the 20-bit physical address. This means that all instructions of a program are relative to the contents of the CS register multiplied by 16 and then offset is added to this value provided by the 16-bit contents of IP. For example, if [CS] = 4560H and [IP] = 1620H, then the 20-bit physical address is generated by the BIU as follows:

\[
\text{Four times logically shifted [CS] to left} = 45600H \\
+ \text{[IP] as offset} = 1620H \\
\text{20-bit physical address} = 46C00H
\]

The BIU always inserts four zeros for the lowest 4-bits of the 20-bit starting address (physical) of a segment. In other words, the CS contains the base or start of the current code segment, and IP contains the distance or offset from this address to the next instruction byte to be fetched. Note that immediate data are considered as part of the code segment.

The SS register points to the current stack. The 20-bit physical stack address is calculated from SS and SP for stack instructions such as PUSH and POP. The programmer can use the BP register instead of SP for accessing the stack using the based addressing mode. In this case, the 20-bit physical stack address is calculated from BP and SS.

The DS register points to the current data segment, operands for most instructions are fetched from this segment. The 16-bit contents of Source Index (SI) or Destination Index (DI) are used as offset for computing the 20-bit physical address.

The ES register points to the extra segment in which data (in excess of 64K pointed to by DS) is stored. String instructions always use ES and DI to determine the 20-bit physical address for the destination.

The segments can be continuous, partially overlapped, fully overlapped, or disjoint. An example of how five segments (segment 0 through segment 4) may be stored in physical memory is shown below:

In the above segments 0 and 1 are contiguous (adjacent), segments 1 and 2 are partially overlapped, segments 2 and 3 are fully overlapped, and segments 2 and 4 are disjoint. Every segment must start on 16-byte memory boundaries.

Typical examples of values of segments should then be selected based on physical addresses starting at 00000H, 00010H, 00020H, 00030H, ..., FEFF0H. A physical memory location may be mapped into (contained in) one or more logical segments. Many applications can be written to simply initialize the segment registers and then forget them.
A segment can be pointed to by more than one segment register. For example, DS and ES may point to the same segment in memory if a string located in that segment is used as a source segment in one string instruction and as a destination segment in another string instruction. Note that for string instructions, a destination segment must be pointed to by ES.

It should be pointed out that codes should not be written within 6 bytes of the end of physical memory. Failure to comply with this guideline may result in an attempted op code fetch from nonexistent memory, hanging the CPU if READY is not returned. One example of four currently addressable segments is shown below:

The EU decodes and executes instructions. A decoder in the EU control system translates instructions. The EU has a 16-bit ALU for performing arithmetic and logic operations.

The EU has eight 16-bit general registers. These are AX, BX, CX, DX, SP, BP, SI, and DI. The 16-bit registers AX, BX, CX, and DX can be used as two 8-bit registers (AH, AL, BH, BL, CH, CL, DH, DL). For example, the 16-bit register DX can be considered as two 8-bit registers DH (high byte of DX) and DL (low byte of DX). The general-purpose registers AX, BX, CX, and DX are named after special functions carried out by each one of them. For example, the AX is called the 16-bit accumulator while the AL is the 8-bit accumulator. The use of accumulator registers is assumed by some instructions. The Input/Output (IN or OUT) instructions always use AX or AL for inputting/outputting 16- or 8-bit data to or from an I/O port.

Multiplication and division instructions also use AX or AL. The AL register is the same as the 8085 A register.

BX register is called the base register. This is the only general-purpose register, the contents of which can be used for addressing 8086 memory.

All memory references utilizing these register contents for addressing use DS as the default segment register. The BX register is similar to 8085 HL register. In other words, 8086 BH and BL are equivalent to 8085 H and L registers, respectively.

The CX register is known as the counter register. This is because some instructions such as shift, rotate, and loop instructions use the contents of CX as a counter. For example, the instruction LOOP START will automatically decrement CX by 1 without affecting flags and will check if [CX] = 0. If it is zero, the 8086 executes the next instruction; otherwise the 8086 branches to the label START.

The data register DX is used to hold high 16-bit result (data) in 16 x 16 multiplication or high 16-bit dividend (data) before a 32 x 16 division and the 16-bit remainder after the division.

The two pointer registers, SP (stack pointer) and BP (base pointer), are used to access data in stack segment. The SP is used as an offset from the current SS during execution of instructions that involve stack segment in external memory. The SP contents are automatically updated (incremented or decremented) due to execution of POP or PUSH instruction.

The base pointer contains an offset address in the current SS. This offset is used by the instructions utilizing the base addressing mode.

The FLGS register in the EU holds the status flags typically after an ALU operation. Figure 3.2 shows the 8086 registers.
3.3 8086 ADDRESSING MODES

The 8086 has 12 basic addressing modes. The various 8086 addressing modes can be classified into five groups:

1. Addressing modes for accessing immediate and register data (register and immediate modes)

2. Addressing modes for accessing data in memory (memory modes)
3. Addressing modes for accessing I/O ports (I/O modes)
4. Relative addressing mode
5. Indirect addressing mode

3.3.1 ADDRESSING MODES FOR ACCESSING IMMEDIATE AND REGISTER DATA (REGISTER AND IMMEDIATE MODES)

3.3.1.a Register Addressing Mode
This mode specifies the source operand, destination operand, or both to be contained in an 8086 register. An example is MOV CX, DX which moves the 16-bit contents of CX into DX. Note that in the above both source and destination operands are in register mode. Another example is MOV CL, DL which moves 8-bit contents of DL into CL. MOV BX, CX is an illegal instruction; the register sizes must be the same.

3.3.1.b Immediate Addressing Mode
In immediate mode, 8- or 16-bit data can be specified as part of the instruction. For example, MOV CL, 02H moves the 8-bit data 02H into CL. On the other hand, MOV DX, 0502H moves the 16-bit data 0502H into DX. Note that in both of the above MOV instructions, the source operand is in immediate mode and the destination operand is in register mode.

A constant such as "VALUE" can be defined by the assembler EQUATE directive as VALUE EQU 35H. An 8086 instruction with immediate mode such as MOV BH, VALUE can then be used to load 35H into BH. Note that even though the immediate mode specifies data with the instruction, these immediate data must be located in the memory addressed by the 8086 CS and IP registers. This is because these data are considered as part of the instruction.

3.3.2 ADDRESSING MODES FOR ACCESSING DATA IN MEMORY (MEMORY MODES)

As mentioned before, the Execution Unit (EU) has direct access to all registers and data for register and immediate operands. However, the EU cannot directly access the memory operands. It must use the BIU in order to access memory operands. For example, when the EU needs to access a memory location, it sends an offset value to the BIU. This offset is also called the Effective Address (EA). Note that EA is displacement of the...
desired location from the segment base. As mentioned before, the BIU generates a 20-bit physical address after shifting the contents of the desired segment register four bits to the left and then adding the 16-bit EA to it. The 8086 must use a segment register whenever it accesses the memory. Also, every memory addressing instruction uses an Intel-defined standard default segment register. However, a segment override prefix can be placed before most of the memory addressing instructions whose default segment register is to be overridden. For example, INC BYTE PTR [START] will increment the 8-bit content of memory location in DS with offset START by one. However, segment DS can be overridden by ES as follows: INC ES; BYTE PTR [START], segments cannot be overridden for stack reference instructions (such as PUSH and POP). Destination segment of a string segment which must be ES (if a prefix is used with string instruction, only the source segment DS can be overridden). The Code Segment (CS) register used in program memory addressing cannot be overridden. There are six modes in this category. These are:

a. Direct addressing mode
b. Register indirect addressing mode
c. Based addressing mode
d. Indexed addressing mode
e. Based indexed addressing mode
f. String addressing mode

3.3.2.a Direct Addressing Mode
In this mode, the 16-bit effective address (EA) is taken directly from the displacement field of the instruction. The displacement (unsigned 16-bit or sign-extended 8-bit number) is stored in the location following the Instruction op code. This EA or displacement is the distance of the memory location from the current value in the data segment (DS) register in which the data are stored. The BIU shifts the [DS] four times to the left and adds the EA to generate the 20-bit physical address. The register can be overridden by the programmer by using a segment override instruction to tell the BIU to add EU to some other segment base for producing the 20-bit physical address. As an example of 16-bit read from memory, consider MOV CX, START. This instruction moves the contents of memory location which is offset by START from the current DS value into register CX.

Note that, in the above instruction, the source is in direct addressing mode. If the 16-bit value assigned to the offset START by the programmer using an assembler pseudoinstruction such as DW 0040H, then the BIU generates the 20-bit physical address 30540H on the 8086 address pins and then initiates a memory read cycle to read the 16-bit data from memory location starting at 30540H. The memory logic places the 16-bit contents of locations 30540H and 30541H on the 8086 data pins. The BIU transfers these data to the EU; the EU then moves these data to CSX [30540H] to CLX and [30541H] to CHX. Now, for 8-bit read from memory consider MOV CH, START. If [DS] = 3050H, the value of START is 0040H, then the MOV instruction moves [BL] and [BH] to locations 30540H, 30541H, respectively. Similarly, an example of 8-bit data write is MOV CX, START, BL. The 8086 does not have a MOVE instruction for moving 8-bit or 16-bit data from one memory location to another in which both source and destination operands are in direct addressing mode. String addressing mode (to be discussed later) allows string memory operation such as block move from source memory array to a destination memory array. Note that START in the above can be defined as an address by using the assembler DB (Define Byte) or DW (Define Word) pseudoinstructions.

3.3.2.b Register Indirect Addressing Mode
In this mode, the EA is specified to either a pointer register or an index register. The pointer register can be either base register BX or base pointer register BP and index register can be either Source Index (SI) register or Destination Index (DI) register. The 20-bit physical address is computed using DS and EA. For example, consider MOV [DI], BX. The destination operand of the above instruction is in register indirect mode, while the source operand is in register mode. The instruction moves the 16-bit content of BX into a memory location offset by the value of EA specified in DI from the current contents in DS. Now, if [DS] = 5000H, [DI] = 0040H, and [BX] = 2456H then after MOV [DI], BX, content of BX (2456H) is moved to memory locations 50040H and 50041H.

Using this mode, one instruction can operate on many different memory locations if the value in the base or index register is updated. The LEA (Load Effective Address) might be used to change the register value.

3.3.2.c Based Addressing Mode
In this mode, EA is obtained by adding a displacement (signed 8-bit or unsigned 16-bit) value to the contents of BX or BP. The segment registers used are DS and SS. When memory is accessed, the 20-bit physical address is computed from BX and DS. On the other hand, when the stack is accessed, the 20-bit physical address is computed from BP and SS.
allows the programmer to access the stack without changing the SP contents. As an example of this mode, consider MOV AL, START [BX]. Note that some assemblers use MOV AL, [BX + START] rather than MOV AL, START [BX]. The source operand in the above instruction is in base mode. EA is obtained by adding the value of START and [BX]. The 20-bit physical address is produced from DS and EA. The 8-bit content of this memory location is moved to AL. The displacement START can be either unsigned 16-bit or signed 8-bit. However, 1 byte is saved for the machine code representation of the instruction if 8-bit displacement is used. The 8086 sign-extends the 8-bit displacement and then adds it to [BX] in the above MOV instruction for determining EA. On the other hand, the 8086 adds an unsigned 16-bit displacement directly with [BX] for determining EA.

Based addressing provides a convenient way to address structure which may be stored at different places in memory.

For example, the element salary in record 0 of the employee NAME 0 can be loaded into an 8086 internal register such as AL using the instruction MOV AL, ALPFA [BX], where ALPFA is the 8-bit displacement 02H and BX contains the starting address of the record 0. Now, in order to access the salary of RECORD N, the programmer simply changes the contents of the base register to 3000H.

If BP is specified as a base register in an instruction, the 8086 automatically obtains the operand from the current SS (unless a segment override prefix is present). This makes based addressing with BP a very convenient way to access stack data. BP can be used as a stack pointer in SS to access local variables. Consider the following instruction sequence:

- PUSH BP : save BP
- MOV BP, SP : establish BP
- PUSH AX : save registers
- SUB SP, 4 : allocate 2 words of stack for accessing stack
- MOV [BP – 6], AX : Arbitrary instructions for accessing stack data using BP
- MOV [BP – 8], BX
- ADD SP, 4 : deallocate storage
- POP AX : restore
- POP DX : all registers
- POP BP : that were pushed before

This instruction sequence is arbitrarily chosen to illustrate the use of BP for accessing the stack.

Figure 3.4 shows the 8086 stack during various stages. Figure 3.4A shows the stack before execution of the instruction sequence.

The instruction sequence from PUSH BP to SUB SP, 4 pushes BP, DX, and AX and then subtracts 4 from SP, and this allocates 2 words of the stack. The stack at this point is shown in Figure 3.4B. Note that in 8086, SP is decremented by 2 for PUSH and incremented by 2 for POP. The [BP] is not affected by PUSH or POP. The instruction sequence MOV [SP – 6], AX saves AX in the stack location addressed by [BP – 6] in SS. The instruction MOV [BP – 8], BX writes the [BX] into the stack location [BP – 8] in SS. These instructions are arbitrarily chosen to illustrate how BP can be used to access the stack. These two local variables can be accessed by the subroutine using BP. The instruction ADD SP, 4 releases two words of the allocated stack. The stack at this point is shown in Figure 3.4C. The last three POP instructions restore the contents of AX, DX, and BP to their original values and return the stack as it was before the instruction sequence was executed. This shown in Figure 3.4D.

3.3.2.2 Indexed Addressing Mode

In this mode, the effective address is calculated by adding the unsigned 16-bit or sign-extended 8-bit displacement and the contents of SI or DI. As an example, MOV BH, START [SI] moves the contents of the 20-bit address computed from the displacement START, SI and DS into BH.
The 8-bit displacement is provided by the programmer using the assembler pseudoinstruction such as EQU. For 16-bit displacement, the EU adds this to SI to determine EA. On the other hand, for 8-bit displacement the EU sign-extends it to 16 bits and then adds to SI for determining EA.

Note that for both based and indexed modes, the 8086 obtains the Effective Address (EA) by using the equation \( EA = RA + M \), where RA is the Reference Address and M is the Modifier. In case of based addressing mode, RA is held in a register such as BX or BP and the modifier M is unsigned 16-bit or signed 8-bit displacement and is included as part of the instruction. The based mode is useful in accessing segmented memory in which BX or BP holds the base address of a segment. In indexed addressing mode, on the other hand, RA is included as part of the instruction and the 8086 register SI or DI contains the value of M. RA can be unsigned 16-bit or sign-extended 8-bit.

Indexed addressing mode can be used to access a single table. The displacement can be the starting address of the table. The content of SI or DI can then be used as an index from the starting address to access a particular element in the table.

3.3.2c Based Indexed Addressing Mode

In this mode, the EA is computed by adding a base register (BX or BP), an index register (SI or DI), and a displacement (unsigned 16-bit or sign-extended 8-bit). As an example, consider MOV ALPHA [SI] [BX], CL if [BX] = 0200H, value of ALPHA = 08H, [SI] = 1000H, and [DX] = 3000H, then 8-bit content of CL is moved to 20-bit physical address 31208H.
Based indexed addressing mode provides a convenient way for a subroutine to address an array allocated on a stack. Register BP can be loaded with the offset in segment SS (top of the stack after the subroutine has saved registers and allocated local storage). The displacement can be the value which is the difference between the top of the stack and the beginning of the array. An index register can then be used to access individual array elements as follows:

![Diagram of stack and displacement](51.jpg)

In the above, [BP] = top of the stack = 2005H; displacement difference between the top of the stack and start of the array = 04H; [SI or DI] = N = 16-bit number (0, 2, 4, 6 in the example). As an example, the instruction MOV DX, 4 [SI] [BP] with [SI] = 6 will read the array (3) which is the content of 200FH in SS into DX. Since in the based indexed mode, the contents of two registers such as IX and SI can be varied, two-dimensional arrays such as matrices can also be accessed.

3.3.2 String Addressing Mode

This mode uses index registers. The string instructions automatically assume SI to point to the first byte or word of the source operand and DI to point to the first byte or word of the destination operand. The contents of SI and DI are automatically incremented (by setting DF to 0 by CLD instruction) or decremented (by setting DF to 1 by STD instructions) to point to the next byte or word. The segment register for the source is DS and may be overridden.

The segment register for the destination must be ES and cannot be overridden. As an example, consider MOVS BYTE. If [DF] = 0, [DS] = 2000H, [SI] = 0500H, [ES] = 4000H, [DI] = 0300H, [20500H] = 38H, and [40500H] = 45H, then after execution of the MOVS BYTE [40500H] = 38H, [ES] = 0801H, and [DI] = 0301H. The contents of other registers and memory locations are unchanged. Note that SI and DI can be used in either the source or destination operand of a two-operand instruction, except for string instructions in which SI points to the source (may be overridden) and DI must point to the destination.

3.3.3 ADDRESSING MODES FOR ACCESSING I/O PORTS (I/O MODES)

Standard I/O uses port addressing modes. For memory-mapped I/O, memory addressing modes are used. There are two types of port addressing modes: direct and indirect.

In direct port mode, the port number is an 8-bit immediate operand. This allows fixed access to ports numbered 0 to 255. For example, OUT 05H, AL outputs [AL] to 8-bit port 05H. In indirect port mode, the port number is taken from DX allowing 64K 8-bit ports or 32K 16-bit ports. For example, if [DX] = 5040H, then IN AL, DX inputs the 8-bit content of port 5040H into AL. On the other hand, IN AX, DX inputs the 8-bit contents of ports 5040H and 5041H into AL and AH, respectively. Note that 8-bit and 16-bit I/O transfers must take place via AL and AH, respectively.

3.3.4 RELATIVE ADDRESSING MODE

Instructions using this mode specify the operand as a signed 8-bit displacement relative to PC. An example is JNC START. This instruction means that if carry = 0, then PC is loaded with current PC contents plus the 8-bit signed value of START; otherwise the next instruction is executed.

3.3.5 IMPLIED ADDRESSING MODE

Instructions using this mode have no operands. An example is CLC which clears the carry flag to zero.
3.4 8086 INSTRUCTION SET

The 8086 instruction set includes equivalents of the 8085 instructions plus many new ones. The new instructions contain operations such as signed and unsigned multiplication and division, bit manipulation instructions, string instructions, and interrupt instructions.

The 8086 has approximately 117 different instructions with about 300 op codes. The 8086 instruction set contains no operand, single operand, and two operand instructions. Except for string instructions which involve array operations, the 8086 instructions do not permit memory-to-memory operations. Table 3.1 lists a summary of 8086 instructions in alphabetical order. Tables 3A-1 through 3A-12 (supplied at the end of the chapter) provide a detailed description of the 8086 instructions.

### TABLE 3.1
Summary of 8086 Instructions

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Interpretation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>ASCII adjust (AL) after addition</td>
<td>This instruction has implied addressing mode; this instruction is used to adjust the content of AL after addition of two ASCII characters</td>
</tr>
<tr>
<td>AAD</td>
<td>ASCII adjust for division</td>
<td>This instruction has implied addressing mode; converts two unpacked BCD digits in AX into equivalent binary numbers in AL; AAD must be used before dividing two unpacked BCD digits by an unpacked BCD type</td>
</tr>
<tr>
<td>AAM</td>
<td>ASCII adjust after multiplication</td>
<td>This instruction has implied addressing mode; after multiplying two unpacked BCD numbers, adjust the product in AX to become an unpacked BCD result; CF, SF, and PF are affected</td>
</tr>
<tr>
<td>AAS</td>
<td>ASCII adjust (AL) after subtraction</td>
<td>This instruction has implied addressing mode used to adjust (AL) after subtraction of two ASCII characters</td>
</tr>
<tr>
<td>ADC mem/reg 1, mem/reg 2</td>
<td>Memory or register can be 8- or 16-bit; all flags are affected; no segment register are allowed; no memory-to-memory ADC is permitted</td>
<td></td>
</tr>
<tr>
<td>ADC mem, data</td>
<td>Data can be 8- or 16-bit; mem uses DS as the segment register; all flags are affected</td>
<td></td>
</tr>
<tr>
<td>ADD reg, data</td>
<td>[reg] = [reg] + data</td>
<td>Data can be 8- or 16-bit; register cannot be segment register; all flags are affected; reg is not normally used as AX or AL, 'A' can be AL or AX register; all flags are affected</td>
</tr>
<tr>
<td>ADD mem/reg 1, mem/reg 2</td>
<td>[mem/reg 1] = [mem/reg 2] + CY</td>
<td>Add two 8- or 16-bit data; no memory-to-memory ADD is permitted; all flags are affected; mem uses DS as the segment register; reg 1 or 2 cannot be segment register</td>
</tr>
<tr>
<td>ADD mem, data</td>
<td>[mem] = [mem] + data</td>
<td>Mem uses DS as the segment register; data can be 8- or 16-bit; all flags are affected</td>
</tr>
<tr>
<td>ADD A, data</td>
<td>[A] = [A] + data</td>
<td>Data can be 8- or 16-bit; no segment registers are allowed; all flags are affected; this instruction should not be used to add AL or AX with 8- or 16-bit immediate data</td>
</tr>
<tr>
<td>AND mem/reg 1, mem/reg 2</td>
<td>[mem/reg 1] = [mem/reg 2]</td>
<td>This instruction logically ANDS 8- or 16-bit data in [mem/reg 1] with 8- or 16-bit data in [mem/reg 2]; all flags are affected; OF and CF are cleared to zero; no segment registers are allowed; no memory-to-memory operation is allowed; mem uses DS as the segment register</td>
</tr>
<tr>
<td>AND mem, data</td>
<td>[mem] = [mem] &amp; data</td>
<td>Data can be 8- or 16-bit; mem uses DS as the segment register; all flags are affected with CF and OF always cleared to zero</td>
</tr>
<tr>
<td>AND reg, data</td>
<td>[reg] = [reg] &amp; data</td>
<td>Data can be 8- or 16-bit; reg cannot be segment register; this instruction should not be used to AND AL or AX with 8- or 16-bit immediate data; all flags are affected with CF and OF cleared to zero</td>
</tr>
<tr>
<td>CALL PROC (NEAR)</td>
<td>Call a subroutine in the same segment with signed 16-bit displacement to CALL a subroutine in ( \pm 320 )</td>
<td>Data can be 8- or 16-bit; 'A' must be AL or AX; all flags are affected with OF and CF cleared to zero</td>
</tr>
</tbody>
</table>

NEAR in the statement BEGIN PROC NEAR indicates that the subroutine "BEGIN" is in the same segment and BEGIN is 16-bit signed; CALL BEGIN instruction decrements SP by 2 and then pushes IP onto the stack and then adds the signed 16-bit value of BEGIN to IP and CS is unchanged; then a subroutine is called in the same segment (intra-segment direct)
### TABLE 3.1 (continued)

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Interpretation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL Reg 16</td>
<td>CALL a subroutine in the same segment addressed by the contents of a 16-bit general register.</td>
<td>The 8086 decrements SP by 2 and then pushes IP onto the stack, then specified 16-bit register contents (such as BX, SI, and DI) provide the new value for IP. CS is unchanged (intersegment indirect). The 8086 decrements SP by 2 and pushes IP onto the stack; the 8086 then loads the contents of a memory location pointed to by 8086 16-bit register such as BX, SI, and DI into IP; CS is unchanged (intersegment indirect). FAR in the statement BEGIN PROC FAR indicates that the subroutine 'BEGIN' is in another segment and the value of BEGIN is 32-bit wide. The 8086 decrements SP by 2 and pushes CS onto the stack and moves the low 16-bit value of the specified 32-bit number such as 'BEGIN' in CALL, BEGIN into CS, SP is again decremented by 2; IP is pushed onto the stack; IP is then loaded with high 16-bit value of BEGIN; thus, this instruction CALLs a subroutine in another code segment (intersegment direct). This instruction decrements SP by 2, and pushes CS onto the stack. CS is then loaded with the contents of memory locations addressed by [reg 16 + 2] and [reg 16 + 3]. If DS is 0, the SP is again decremented by 2; IP is pushed onto the stack; IP is then loaded with the contents of memory locations addressed by [reg 16 + 2] and [reg 16 + 3]. If DS is not 0, typical 8086 registers used for reg 16 are BX, SI, and DI (intersegment indirect). Extend the sign bit (bit 7) of AL register into AH. Clear carry to zero. Clear direction flag to zero. Clear interrupt enable flag to zero to disable maskable interrupts. One's complement carry. Memory can be 8- or 16-bit; no memory-to-memory comparison allowed; result of subtraction is not provided; all flags are affected.</td>
</tr>
<tr>
<td>CALL Mem 16</td>
<td>CALL a subroutine addressed by the content of a memory location pointed to by 8086 16-bit register such as BX, SI, and DI.</td>
<td></td>
</tr>
<tr>
<td>CALL PROC (FAR)</td>
<td>CALL a subroutine in another segment</td>
<td></td>
</tr>
<tr>
<td>CALL DWORD PIR [reg 16]</td>
<td>CALL a subroutine in another segment</td>
<td></td>
</tr>
<tr>
<td>CWD</td>
<td>Convert a word to 32 bits</td>
<td>Extend the sign bit of AX (bit 15) into DX. This instruction uses implied addressing mode; converts [AX] into BCD; DAA should be used after BCD addition.</td>
</tr>
<tr>
<td>DAA</td>
<td>Decimal adjust [AL] after addition</td>
<td></td>
</tr>
<tr>
<td>DAS</td>
<td>Decimal adjust [AL] after subtraction</td>
<td></td>
</tr>
<tr>
<td>DEC reg 16</td>
<td>[reg 16] – 1</td>
<td>This instruction uses implied addressing mode; converts [AL] into BCD; DAA should be used after BCD subtraction. This is a one-byte instruction; used to decrement a 16-bit register except segment register; does not affect the carry flag. Used to decrement a byte or a word in memory or an 8-bit register contents. Segment register cannot be decremented by this instruction; does not affect carry flag. Memory is 8-bit for 16-bit by 8-bit divide and 16-bit by 16-bit divisor; this is an unsigned division; no flags are affected; division by zero automatically generates an internal interrupt.</td>
</tr>
<tr>
<td>DEC mem/reg</td>
<td>16/32 bit divide:</td>
<td></td>
</tr>
<tr>
<td>DIV mem/reg</td>
<td>16/32 bit divide:</td>
<td></td>
</tr>
<tr>
<td>ESC external</td>
<td>ESC to external processes</td>
<td>This instruction is used to pass instructions to a coprocessor such as the 8087 floating point coprocessor which simultaneously monitors the system bus with the 8086; the coprocessor's CP codes are 8-bit wide, the</td>
</tr>
<tr>
<td>Instructions</td>
<td>Interpretation</td>
<td>Comments</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
<td>---------</td>
</tr>
<tr>
<td>ESC ext. OPC code, source (continued)</td>
<td>coprocessor treats normal 8086 instructions as NOPs; the 8086 fetches all instructions from memory; when the 8086 encounters in ESC instructions, it treats it as NOP; the coprocessor decodes this instruction and carries out the operation using the 6-bit OPC code independent of the 8086; for ESC OPC code, memory, the 8086 accesses data in memory for the coprocessor; for ESC data, registers, the coprocessor operates on 8086 registers; the 8086 treats this as an NOP.</td>
<td></td>
</tr>
<tr>
<td>HLT</td>
<td>HLT</td>
<td></td>
</tr>
<tr>
<td>IDIV mem/reg</td>
<td>Same as DIV mem/reg</td>
<td></td>
</tr>
<tr>
<td>IMUL mem/reg</td>
<td>For (x * b)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>([AX] \leftarrow [XL] + [XH])</td>
<td>Mem/Reg can be 8- or 16-bit; only CF and OF are affected; signed multiplication</td>
</tr>
<tr>
<td>IN AL, DX</td>
<td>([AL] \leftarrow \text{PORT DX})</td>
<td>Input AL with the 8-bit content of a port addressed by DX; this is a one-byte instruction.</td>
</tr>
<tr>
<td>IN AX, DX</td>
<td>([AX] \leftarrow \text{PORT DX})</td>
<td>Input AX with the 16-bit content of a port addressed by DX and DX + 1; this is a one-byte instruction.</td>
</tr>
<tr>
<td>IN AL, PORT</td>
<td>([AL] \leftarrow \text{PORT})</td>
<td>Input AL with the 8-bit content of a port addressed by the second byte of the instruction.</td>
</tr>
<tr>
<td>IN AX, PORT</td>
<td>([AX] \leftarrow \text{PORT})</td>
<td>Input AX with the 16-bit content of a port addressed by the second byte of the instruction.</td>
</tr>
<tr>
<td>INC reg 16</td>
<td>([\text{reg} 16] \leftarrow [\text{reg} 16] + 1)</td>
<td>This is a one-byte instruction used to increment a 16-bit register except the segment register; does not affect the carry flag.</td>
</tr>
<tr>
<td>INC mem/reg 8</td>
<td>([\text{mem}] \leftarrow [\text{mem}] + 1) or ([\text{reg} 8] \leftarrow [\text{reg} 8] + 1)</td>
<td>This is a two-byte instructions can be used to increment a byte or word in memory or an 8-bit register content; segment registers cannot be incremented by this instruction; does not affect the carry flag.</td>
</tr>
<tr>
<td>INT n (n can be zero thru 255)</td>
<td>([SP] \leftarrow [SP] - 2)</td>
<td>Software interrupts can be used as supervisor calls that is, for service from an operating system; a different interrupt type can be used for each type of service that</td>
</tr>
<tr>
<td></td>
<td>([SP] \leftarrow \text{Flags})</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF \leftarrow 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TF \leftarrow 0</td>
<td></td>
</tr>
</tbody>
</table>
TABLE 3.1 (continued)

Instructions | Interpretation | Comments
--- | --- | ---
JMP Label | Unconditionally jump to another segment | This instruction loads the contents of memory locations addressed by [reg 16] and [reg 16 + 1] in DS into IP; it then loads the contents of memory locations addressed by [reg 16 + 2] and [reg 16 + 3] in DS into CS; typical 8086 registers used for reg 16 are BX, SI, and DI (intersegment indirect).
JMP [mem 16] | Unconditionally jump to another segment | Same as JA/NE exception that the jump is taken if [IP] = 0; used for signed and unsigned comparison.
JMP [mem] (FAR) | Jump to an address specified by the contents of a 16-bit memory location addressed by 16-bit physical address | This instruction has implied addressing mode; it loads AH with the low byte of the flag register; no flags are affected.
JMP Reg 16 | Jump to an address specified by the contents of a 16-bit register such as BX, SI, and DI in the same code segment. | Load a 16-bit register (AX, BX, CX, DX, SP, BP, SI, DS) with the contents of specified memory and load DS with the contents of the location that follows; no flags are affected; DS is used as the segment registers for mem.
JMP [mem] | Jump to an address specified by the contents of a 16-bit memory location addressed by 16-bit physical address; | LEA load effective address loads the value of the source operand rather than its content to register (such as SI, DI, BX) which are allowed to contain offset for accessing memory; no flags are affected. DS is used as the segment register for mem; in the example LEA DX, [BX] DX is loaded with 16-bit value from a memory location addressed by 20-bit physical address computed from DS and BX; the 16-bit content of the next memory is loaded into ES; no flags are affected.
LOCK | LOCK bus during next Instruction | Lock is a one-byte prefix that causes the 8086 (configured in maximum mode) to assert its bus LOCK signal while following
### TABLE 3.1 (continued)

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Interpretation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCK (continued)</td>
<td></td>
<td>Instruction is executed; this signal is used in multiprocessor systems to the LOCK pin of the 8086 (or 8088) can be used to LOCK other processors off the system bus during execution of an instruction; in this way, the 8086 can be shielded from unauthorized access to common system resources such as shared RAM.</td>
</tr>
<tr>
<td>LODS byte or LODSB</td>
<td>FOR BYTE</td>
<td>S[i] ← [S[i]]</td>
</tr>
<tr>
<td>LODS word or LODSW</td>
<td>FOR WORD</td>
<td>[A][S][i] ← [S][i] + 1</td>
</tr>
<tr>
<td>LOOP disp 8</td>
<td>Loop if CX not equal to zero</td>
<td>Decrement CX by one, without affecting flags and loop with signed 8-bit displacement from 1 to 255, zero being positive if CX is not equal to zero.</td>
</tr>
<tr>
<td>LOOPZ disp 8</td>
<td>Loop while equal/loop while zero</td>
<td>Decrement CX by one without affecting flags and loop with signed 8-bit displacement if CX is not equal to zero and ZF = 0, which results from execution of the previous instruction.</td>
</tr>
<tr>
<td>LOOPNE/LOOPNZ disp 8</td>
<td>Loop while not equal/loop while not zero</td>
<td>Decrement CX by one without affecting flags and loop with signed 8-bit displacement if CX is not equal to zero and ZF = 0, which results from execution of the previous instruction.</td>
</tr>
<tr>
<td>MOV mem/reg, mem[reg], mem[reg]</td>
<td>mem[reg] ← [mem[reg]]</td>
<td>mem uses DS as the segment register; no memory-to-memory operation allowed; that is, MOV mem, mem is not permitted; segment register cannot be specified as reg or reg; no flags are affected; not usually used to load or store A' from or to memory.</td>
</tr>
<tr>
<td>MOV reg, data</td>
<td>[reg] ← [data]</td>
<td>Moves 8-bit data from memory to register: no flags are affected.</td>
</tr>
<tr>
<td>MOV data, mem</td>
<td>[data] ← [mem]</td>
<td>Takes lower bytes than reg; mem uses DS as the segment register; A's = AL or AX; no flags are affected.</td>
</tr>
</tbody>
</table>

### TABLE 3.1 (continued)

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Interpretation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV mem, A</td>
<td>mem ← [A]</td>
<td>mem uses DS as segment register; 'A' can be AL or AX; no flags are affected; needs lower bytes than MOV mem, reg</td>
</tr>
</tbody>
</table>
### TABLE 3.1 (continued)

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Interpretation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR reg, data</td>
<td>(reg) ← (reg) + data</td>
<td>Reg and data can be 8 or 16-bit; no segment registers are allowed; all flags are affected with CF and CF cleared to zero; should not be used with OR AL or AX with immediate data.</td>
</tr>
<tr>
<td>OR A, data</td>
<td>(A) ← (A) + data</td>
<td>Data can be 8- or 16-bit; A must be AL or AX; all flags are affected with CF and CF cleared to zero.</td>
</tr>
<tr>
<td>OUT DX, AL</td>
<td>[PORT] ← [AL] DX</td>
<td>Output the 8-bit contents of AL into an I/O Port addressed by the 16-bit content of DX; this is a one-byte instruction.</td>
</tr>
<tr>
<td>OUT DX, AX</td>
<td>[PORT] ← [AX] DX</td>
<td>Output the 8-bit contents of AX into an I/O Port addressed by the 16-bit content of DX; this is a one-byte instruction.</td>
</tr>
<tr>
<td>OUT PORT, AL</td>
<td>[PORT] ← [AL]</td>
<td>Output the 8-bit contents of AL into the Port specified in the second byte of the instruction.</td>
</tr>
<tr>
<td>OUT PORT, AX</td>
<td>[PORT] ← [AX]</td>
<td>Output the 16-bit contents of AX into the Port specified in the second byte of the instruction.</td>
</tr>
<tr>
<td>POP mem</td>
<td>[mem] ← [SP]</td>
<td>Mem uses DS as the segment register; no flags are affected. Cannot be used to POP segment registers or flag register. POP CS is illegal.</td>
</tr>
<tr>
<td>POP reg</td>
<td>[reg] ← [SP]</td>
<td>This instruction pops the top two stack bytes. In the 16-bit flag register, mem uses DS as segment register; no flags are affected; push 16-bit memory contents reg must be a 16-bit register; cannot be used to push segment register or flag register. POP CS is illegal.</td>
</tr>
<tr>
<td>POP segreg</td>
<td>[segreg] ← [SP]</td>
<td>This instruction pushes the 16-bit flag register onto the stack.</td>
</tr>
<tr>
<td>PUSH mem</td>
<td>[mem] ← [SP]</td>
<td>Mem uses DS as the segment register; no flags are affected; push 16-bit memory contents reg must be a 16-bit register; cannot be used to push segment register or flag register. PUSH CS is illegal.</td>
</tr>
<tr>
<td>PUSH reg</td>
<td>[reg] ← [SP]</td>
<td>This instruction pushes the top two stack bytes. In the 16-bit flag register, mem uses DS as segment register; no flags are affected; push 16-bit memory contents reg must be a 16-bit register; cannot be used to push segment register or flag register. PUSH CS is illegal.</td>
</tr>
<tr>
<td>PUSH segreg</td>
<td>[segreg] ← [SP]</td>
<td>This instruction pushes the 16-bit flag register onto the stack.</td>
</tr>
</tbody>
</table>

### 3.4 8086 Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Interpretation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCL mem/reg, CL</td>
<td>ROTATE through carry left byte or word in mem/reg by [CL]</td>
<td>Operation same as RCL mem/reg, 1 except the number of rotates is specified in CL for rotates up to 255; zero or negative rotates are illegal.</td>
</tr>
<tr>
<td>RCR mem/reg, 1</td>
<td>ROTATE through carry right once byte or word in mem/reg</td>
<td>FOR BYTE</td>
</tr>
<tr>
<td>RCR mem/reg, CL</td>
<td>ROTATE through carry right byte or word in mem/reg by [CL]</td>
<td>Operation same as RCR mem/reg, 1 except the number of rotates is specified in CL for rotates up to 255; zero or negative rotates are illegal.</td>
</tr>
<tr>
<td>RET</td>
<td>POPS IP for intersegment CALLS</td>
<td>The assembler generates an intersegment return if the programmer has defined the subroutine as NEAR; for intersegment return, the following operations take place: IP ← [IP]; CF ← [IP] + 2; on the other hand, the assembler generates an intersegment return if the subroutine has been defined as FAR; in this case, the following operations take place: IP ← [IP]; CF ← [IP] + 2; an optional 16-bit displacement &quot;START&quot; can be specified with the intersegment return such as RET START; in this case, the 16-bit displacement is added to the IP value; this feature may be used to discard parameter pushed onto the stack before the execution of the CALL instruction.</td>
</tr>
<tr>
<td>CALLS</td>
<td>POPS IP and CS for intersegment CALLS</td>
<td>FOR WORD</td>
</tr>
<tr>
<td>ROL mem/reg, 1</td>
<td>ROTATE left once byte or word in mem/reg</td>
<td>FOR BYTE</td>
</tr>
</tbody>
</table>

### Diagrams

- **FOR BYTE**
- **FOR WORD**
### TABLE 3.1 (continued)

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Interpretation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ROL</strong> mem/reg, CL</td>
<td>ROTATE left byte or word by the content of CL</td>
<td>(CL) contains rotate count up to 255; zero and negative shifts are illegal; CL is used to rotate count when the rotate is greater than one; mem uses DS as the segment register FOR BYTE</td>
</tr>
<tr>
<td><strong>ROR</strong> mem/reg, 1</td>
<td>ROTATE right once byte or word in mem/reg</td>
<td></td>
</tr>
<tr>
<td><strong>ROR</strong> mem/reg, CL</td>
<td>ROTATE right byte or word in mem/reg by CL</td>
<td>Operation same as ROR mem/reg, 1; (CL) specifies the number of rotates for up to 255; zero and negative rotates are illegal; mem uses DS as the segment register</td>
</tr>
<tr>
<td><strong>SAHF</strong></td>
<td>(Flags, low-byte) ← [AH]</td>
<td>This instruction has the implied addressing mode; the content of the AH register is stored into the low-byte of the flag register; no flags are affected FOR BYTE</td>
</tr>
<tr>
<td><strong>SAL</strong> mem/reg, 1</td>
<td>Shift arithmetic left once byte or word in mem reg</td>
<td>Mem uses DS as the segment register; reg cannot be segment registers; OF and SF are affected; if sign bit is changed during or after shifting, the OF is set to one</td>
</tr>
<tr>
<td><strong>SAL</strong> mem/reg, CL</td>
<td>Shift arithmetic left byte or word by shift count on CL</td>
<td>Operation same as SAL mem/reg, 1; CL contains shift count for up to 255; zero and negative shifts are illegal; (CL) is used as shift count when shift is greater than one; OF and SF are affected; if sign bit of [memreg] is changed during or after shifting, the OF is set to one; mem uses DS as segment register</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Interpretation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SAR</strong> mem/reg, 1</td>
<td>SHIFT arithmetic right once byte or word in mem/reg</td>
<td>FOR BYTE</td>
</tr>
<tr>
<td><strong>SAR</strong> mem/reg, CL</td>
<td>SHIFT arithmetic right byte or word in mem/reg by CL</td>
<td>Operation same as SAR mem/reg, 1; however, shift count is specified in CL for shifts up to 255; zero and negative shifts are illegal</td>
</tr>
<tr>
<td><strong>SBB</strong> mem/reg, 1</td>
<td>[mem/reg] ← [mem/reg] ← [mem/reg] ← [CL]</td>
<td>Same as SLB mem/reg, 1, mem/reg 2 except this is a subtraction with borrow</td>
</tr>
<tr>
<td><strong>SBB</strong> mem/reg, 2</td>
<td>SUB mem/reg, 1</td>
<td>Same as SLB mem/reg, 1, mem/reg 2 except this is a subtraction with borrow</td>
</tr>
<tr>
<td><strong>SBB</strong> mem, data</td>
<td>mem ← mem ← mem ← CY</td>
<td>Same as SLB mem, data except this is a subtraction with borrow</td>
</tr>
<tr>
<td><strong>SBB</strong> reg, data</td>
<td>[reg] ← [reg] ← data ← CY</td>
<td>Same as SLB reg, data except this is a subtraction with borrow</td>
</tr>
<tr>
<td><strong>SBB</strong> A, data</td>
<td>[A] ← [A] ← data ← CY</td>
<td>Same as SLB A, data except this is a subtraction with borrow</td>
</tr>
<tr>
<td><strong>SCAS BYTE</strong> or <strong>SCASB</strong></td>
<td>FOR BYTE [AL] ← [DI], flags are affected, [DI] = [DI] + 1</td>
<td>B or 16-bit data addressed by [DI]. Mem in ES is subtracted from 8- or 16-bit data in AX or AX and flags are affected without affecting [AL] or [AX] or storing data; LB cannot be overriden; if OF = 0, then DI is incremented by one for byte and two for word; if OF = 1, then DI is decremented by one for byte or decremented by two for word</td>
</tr>
<tr>
<td><strong>SCAS WORD</strong> or <strong>SCASW</strong></td>
<td>FOR WORD [AX] ← [DI], flags are affected, [DI] = [DI] + 2</td>
<td>Same as SCAS BYTE, CL except overflow is cleared to zero</td>
</tr>
<tr>
<td><strong>SHL</strong> mem/reg, 1</td>
<td>SHIFT logical left once byte or word in mem/reg</td>
<td>FOR BYTE</td>
</tr>
<tr>
<td><strong>SHL</strong> mem/reg, CL</td>
<td>SHIFT logical left byte or word in mem/reg by the shift count in CL</td>
<td></td>
</tr>
<tr>
<td><strong>SHR</strong> mem/reg, 1</td>
<td>SHIFT right logical once byte or word in mem/reg</td>
<td>FOR WORD</td>
</tr>
</tbody>
</table>
### Table 3.1 (continued)

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Interpretation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHL mem/reg, CL</td>
<td>SHIFT right logical byte or word in mem/reg by [CL]</td>
<td>Operation same as SHR mem/reg, 1; however, shift count is specified in CL for shifts up to 255; zero and negative shifts are illegal.</td>
</tr>
<tr>
<td>STC</td>
<td>CF ← 1</td>
<td>Set carry to one</td>
</tr>
<tr>
<td>STD</td>
<td>DF ← 1</td>
<td>Set direction flag to one</td>
</tr>
<tr>
<td>STI</td>
<td>IF ← 1</td>
<td>Set interrupt enable flag to one to enable maskable interrupts.</td>
</tr>
<tr>
<td>STOS BYTE or STOSB</td>
<td>FOR BYTE</td>
<td>Store 8-bit data from AL or 16-bit data from AX into a memory location addressed by DI in segment DS; segment register DS cannot be overridden; if DI = 0, then DI is incremented by one for byte or incremented by two for word after the store.</td>
</tr>
<tr>
<td>STOS WORD or STOSW</td>
<td>FOR WORD</td>
<td>No memory-to-memory SUB permitted; all flags are affected; mem uses DS as the segment register.</td>
</tr>
<tr>
<td>SUB mem/reg, 1, mem/reg, 2</td>
<td>[mem/reg] ← [mem/reg] - [mem/reg 2]</td>
<td>Data can be 8- or 16-bit; mem uses DS at the segment register; all flags are affected.</td>
</tr>
<tr>
<td>SUB mem, data</td>
<td>[mem] ← [mem] - [data]</td>
<td>Data can be 8- or 16-bit; this instruction is not usually used for subtracting data from AX or AL. SUB A, S data is used for this; all flags are affected.</td>
</tr>
<tr>
<td>SUB reg, data</td>
<td>[reg] ← [reg] - [data]</td>
<td>'A' can be AL or AX; data can be 8- or 16-bit; all flags are affected.</td>
</tr>
<tr>
<td>SUB A, data</td>
<td>[A] ← [A] - [data]</td>
<td>No memory-to-memory TEST is allowed; no result is provided; all flags are affected with CF and OF cleared to zero; [mem], [reg 2] or (reg 2) can be 8- or 16-bit; no segment registers are allowed; mem uses DS as the segment register.</td>
</tr>
<tr>
<td>TEST mem/reg, 1, mem/reg, 2</td>
<td>[mem/reg 1] ← [mem/reg 2] - [mem/reg 2]</td>
<td>Mem and data can be 8- or 16-bit; no result is provided; all flags are affected with CF and OF cleared to zero; mem uses DS as the segment register.</td>
</tr>
<tr>
<td>TEST reg, data</td>
<td>[reg] ← [data] - [reg]</td>
<td>Reg and data can be 8- or 16-bit; no result is provided; all flags are affected with CF and OF cleared to zero; reg cannot be segment register; should not be used to test AL or AX with immediate data.</td>
</tr>
<tr>
<td>TEST A, data</td>
<td>[A] ← [data]</td>
<td>'A' must be AL or AX; data can be 8- or 16-bit; no result is provided; all flags are affected with CF and OF cleared to zero.</td>
</tr>
</tbody>
</table>

### Table 3.1 (continued)

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Interpretation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>8086 enters wait state</td>
<td>Causes CPU to enter wait state if the 8086 TEST pin is high; while in wait state, the 8086 continues to check TEST pin for low; if TEST pin goes back to zero, the 8086 executes the next instruction.</td>
</tr>
<tr>
<td>XCHG AX, reg</td>
<td>(AX) ← [reg]</td>
<td>Reg must be 16-bit; no flags are affected; reg cannot be segment register.</td>
</tr>
<tr>
<td>XCHG mem, reg</td>
<td>[mem] ← [reg]</td>
<td>Reg and mem can be both 8- or 16-bit; mem uses DS as the segment register; reg cannot be segment register; no flags are affected.</td>
</tr>
<tr>
<td>XCHG reg, reg</td>
<td>[reg] ← [reg]</td>
<td>Reg not used to exchange reg with AX; reg can be 8- or 16-bit; reg cannot be segment register; no flags are affected.</td>
</tr>
<tr>
<td>XLAT</td>
<td>[AL] ← [AL + IBX]</td>
<td>This instruction is useful for translating characters from one code such as ASCII to another such as EBCDIC; this is a non-opercational instruction.</td>
</tr>
</tbody>
</table>

This instruction looks up the value of alpha. The instruction is valid for loading AL with the contents of a 20-bit physical address computed from DS, BX, and AH; this instruction can be used to load the elements in a table where BX can be loaded with a 16-bit value, point to the starting address (offset from DS) and AL can be loaded with the element number (0 being the first element number); no flags are affected; XLAT instruction is equivalent to MOV AL, [AL + BX]. |

| XOR mem/reg, 1, mem/reg, 2 | [mem/reg 1] ← [mem/reg 1] ⊕ [mem/reg 2] | Mem and data can be 8- or 16-bit; all flags are affected with CF and OF cleared to zero; mem uses DS as the segment register. |
| XOR reg, data | [reg] ← [mem] ⊕ [data] | Data and mem can be 8- or 16-bit; mem uses DS as the segment register; mem cannot be segment register; all flags are affected with CF and OF cleared to zero. |
| XOR A, data | [A] ← [A] ⊕ [data] | 'A' must be AL or AX; data can be 8- or 16-bit; all flags are affected with CF and OF cleared to zero. |
Explanation of Table 3.1-1 Instructions

- MOV CX, DX copies the 16-bit content of DX into CX. MOV AX, 0205H moves immediate data 0205H into 16-bit register AX. MOV CH, [BX] moves the 8-bit content of memory location addressed by BX in segment register DS into CH. If [BX] = 0050H, [DS] = 2000H, [20050H] = 08H, then after MOV CH, [BX], the content of CH will be 08H.
- MOV START [BP], CX moves the 16-bit (CL to first location and then CH) content of CX into two memory locations addressed by the sum of the displacement START and BP in segment register SS. For example, if [CX] = 5000H, [BP] = 0020H, [SS] = 3000H; START = 06H, then after MOV START [BP], CX physical memory location [30006H] = 9H and [30007H] = 50H. Note that the segment register SS can be overridden by CS using MOV CS: START [BP], CX.
- PUSH START [BX] pushes the 16-bit contents of two memory locations starting at the 20-bit physical address computed from START, BX, and DS after decrementing SP by 2.
- POP ES pops the top stack word into ES and then increments SP by 2.
- XCHG START [BX], AX exchanges the 16-bit word in AX with the contents of two consecutive memory locations starting at 20-bit physical address computed from START, BX, and DS. AL is exchanged with the content of the first location and AH is exchanged with the content of the next location.
- XLAT can be used to convert a code such as ASCII into another code such as EBCDIC.

Suppose that an 8086-based microcomputer is interfaced to an ASCII keyboard and an IBM printer (EBCDIC code). In such a system, any number entered into the microcomputer will be in ASCII code which must be converted to EBCDIC code before outputting to the printer. If the number entered is 4, then the ASCII code for 4 (10H) must be translated to the EBCDIC code for 4 (F4H). A look-up table containing EBCDIC code for all the decimal numbers (0 to 9) can be stored in memory at the 16-bit starting address (offset) in DS, 3000H with the code for 0 stored at 3000H, code for 1 at 3001H, and so on. Now, if AL is loaded with the ASCII code 34H and BX is loaded with 3000H, then by using the XLAT instruction, content of memory location 3034H in DS containing the EBCDIC code for 4 (F4H) from the look-up table is read into AL, thus replacing the ASCII code for 4 with the EBCDIC code for 4.

Note that MOV A, mem can also be obtained from MOV mem/reg 2, mem/reg 1. This means that MOV A, mem has two different OP codes; one obtained from MOV mem/reg 2, mem/reg 1 and the other from the dedicated MOV A, mem instruction. The assembler usually provides the OP code for the MOV A, M since it takes fewer bytes. The same concept applies to other instructions involving “A” which have more than one OP code.

Symbolic names such as START can be defined as an address containing a byte or word by using the DB or DW assembler directive. To define the symbolic name as a constant for immediate data, one can use the EQU directive.

Explanation of Table 3.1-2 Instructions

- Consider fixed port addressing in which the 8-bit port address is directly specified as part of the instruction. IN AL, 38H inputs 8-bit data from port 38H into AL. IN AX, 38H inputs 16-bit data from ports 38H and 39H into AX. OUT 38H, AL outputs the contents of AL to port 38H. OUT 38H, AX, on the other hand, outputs the 16-bit contents of AX to ports 38H and 39H.
- For the variable port addressing, the port address is 16-bit and is specified in the DX register. Consider ports addressed by 16-bit address contained in DX. Assume [DX] = 3124H in all the following examples:
  - IN AL, DX inputs 8-bit data from 8-bit port 3124H into AL.
  - IN AX, DX inputs 16-bit data from ports 3124H and 3125H into AX.
  - OUT DX, AL outputs 8-bit data from AL into port 3124H.
  - OUT DX, AX outputs 16-bit data from AX into ports 3124H and 3125H.

Variable port addressing allows up to 65,536 ports with addresses from 0000H to FFFFH. The port addresses in the variable port addressing can be calculated dynamically in a program. For example, assume that an 8086-based microcomputer is connected to three printers via three separate ports. Now, in order to output to each one of the printers, separate programs are required if fixed port addressing is used. However, with variable port addressing one can write a general subroutine to output to the printers and then supply the address of the port for a particular printer in which data output is desired to register DX in the subroutine.
Explanations of Table 3A-3 Instructions

- LEA reg, mem loads an offset (mem) in DS directly into the specified register. The XLAT and string instructions assume that certain registers point to operands. LEA can be used to load these addresses. For example, LEA can be used to load the address of the table used by the XLAT instruction. Consider converting an ASCII code into EBCDIC code. Suppose that EBCDIC codes for the first seven symbols NUL (NULL), SOH (Start of Heading), STX (Start Text), and ETX (End Text), EOT (End of Transmission), ENQ (Enquiry), and ACK (Acknowledgement) are stored in memory.

Starting at location TABLE 1 as follows:

<table>
<thead>
<tr>
<th>TABLE 1:</th>
<th>00H: EBCDIC Code for NUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>01H:</td>
<td>Code for SOH</td>
</tr>
<tr>
<td>02H:</td>
<td>Code for STX</td>
</tr>
<tr>
<td>03H:</td>
<td>Code for ETX</td>
</tr>
<tr>
<td>07H:</td>
<td>Code for EOT</td>
</tr>
<tr>
<td>2EH:</td>
<td>Code for ENQ</td>
</tr>
<tr>
<td>2FH:</td>
<td>Code for ACK</td>
</tr>
</tbody>
</table>

Suppose that the ASCII codes for these characters are stored in memory starting at address TABLE 2 as follows:

<table>
<thead>
<tr>
<th>TABLE 2:</th>
<th>00H: ASCII Code for NUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>01H:</td>
<td>Code for SOH</td>
</tr>
<tr>
<td>02H:</td>
<td>Code for STX</td>
</tr>
<tr>
<td>03H:</td>
<td>Code for ETX</td>
</tr>
<tr>
<td>04H:</td>
<td>Code for EOT</td>
</tr>
<tr>
<td>05H:</td>
<td>Code for ENQ</td>
</tr>
<tr>
<td>06H:</td>
<td>Code for ACK</td>
</tr>
</tbody>
</table>

Now, in order to translate an ASCII Code for a symbol such as EOT into its EBCDIC Code, ASCII Code for EOT (04H) can be loaded into AL, BX can be initialized with the value of TABLE 1 using LEA, and XLAT can then be used to load the EBCDIC Code for EOT (27H) into AL as follows:

- LEA BX, TABLE 1; Load the value TABLE 1 into BX
- MOV AL, 04H; Load ASCII Code for EOT into AL
- XLAT; Load the content of memory location addressed by [TABLE 1 + 04H] into AL.

8086 Instruction Set

- LDS reg, mem can be used to initialize SI and DS to point to the start of the source string before using one of the string instructions. For example, LDS SI, [BX] loads the 16-bit contents of memory offset by [BX] in DS into SI and the 16-bit contents of memory offset by [BX + 2] in DS into DS.

- LES reg, mem can be used to point to the start of the destination string before using one of the string instructions. For example, LES DL, [BX] loads the 16-bit contents of memory offset by [BX] in DS to DI and then initializes ES with the 16-bit contents of memory offset by [BX + 2] in DS.

Table 3A-4 is self-explanatory.

Explanations of Table 3A-5 Instructions

- Numerical data received by an 8086-based microcomputer from a terminal is usually in ASCII code. The ASCII codes for numbers 0 to 9 are 30H through 39H. Two 8-bit data can be entered into an 8086-based microcomputer via a terminal. The ASCII codes for these data (with 3 as the upper middle for each type) can be added. AAA instruction can then be used to provide the correct unpacked BCD. Suppose that ASCII codes for 2 (32H) and 5 (35H) are entered into an 8086-based microcomputer via a terminal. These ASCII codes can be added and then the result can be adjusted to provide the correct unpacked BCD using AAA instructions as follows:

- ADD CL, DL; [CL] = 32, = ASCII for 2
- [DL] = 35, = ASCII for 5
- Result [CL] = 67
- MOV AL, CL; Move ASCII results
- Into AL since AAA
- Adjust only [AL]
- AAA; [AL] = 07, unpacked
- BCD for 7

Note that in order to send the unpacked BCD result 07 back to the terminal, [AL] = 07 can be ORed with 30H to provide 37H, the ASCII code for 7.

- DAA is used to adjust the result of adding two packed BCD numbers in AL to provide a valid BCD number. After the addition, the low 4-bit of the result in AL is greater than 9 (or d'AF = 1), then the DAA adds 6 to the low 4 bits of AL. On the other hand, if the high 4 bits
of the result in AL is greater than 9 (or if CF = 1), then DAA adds 60H to AL. As an example, consider adding two packed BCD digits 55 with 18 as follows:

**ADD AL, DL : [AL] = 55 BCD**  
**[DL] = 18 BCD**  
**Result = [AL] = 60H**

**DAA**  
Since low nibble  
\[ D = 1101_2 > 9, \text{ add } i.e. \ 1101_2 + 0110_2 = \ 10011_2 \]

\[ \text{carry} \]

The following instruction sequence provides the correct subtraction result:

**SUB AL, DL : [AL] = 1111 1110 \_2 = subtraction result**  
in 2\_s complement  
**[CF] = 1**

**AAS**  
**[AL] = BCD02**  
**[CF] = 1 means**  
**borrow to be used in multi BCD subtraction**

AAS adjusts [AL] and leaves zeros in the upper nibble. To output to the terminal from the microcomputer, BCD data can be ORed with 30H to produce the correct ASCII code.

**DAS**  
**[AL] = BCD20**  
**[CF] = 1 means**  
**borrow to be used in multi BCD subtraction**

DAS can be used to adjust the result of subtraction in AL of two packed BCD numbers to provide the correct packed BCD. If low 4-bit of AL is greater than 9 (or if AF = 1), then DAS subtracts 6 from the low 4-bit of AL. On the other hand, if the upper 4-bit of the result in AL is greater than 9 (or if CF = 1), DAS subtracts 60 from AL.

While performing these subtractions, any borrows from LOW and HIGH nibbles are ignored. For example, consider subtracting BCD 55 in DL from BCD 94 in AL.

**SUB AL, DL : [AL] = 3FH \rightarrow \text{Low nibble = 1111}**  
**DAS : CF = 0**  
**-6 = 1010 1001**  
**[AL] = 39 BCD**  
**Ignore**

- **IMUL mem/reg provides signed 8 \times 8 or signed 16 \times 16 multiplication.** As an example, if [CL] = FDH \rightarrow [AL] = FEH = 2\_w, then after IMUL CL, register AX contains 0006H.
- **Consider 16 \times 16 unsigned multiplication, MUL WORDPTR [BX].**  
  - **Consider DIV BL. **IF [AX] = 0009H, [BL] = 02H, then after DIV BL,
    - **[AH] = Remainder = 01H**  
    - **[AL] = Quotient = 04H**
- **Consider IDIV.WORDPTR [BX].**  
  WORDPTR [BX],
  \[ [DX] = \text{Remainder} = 0001H \]
  \[ [AX] = \text{Quotient} = 0004H \]
- **AAD converts two unpacked BCD digits in AH and AL to an equivalent binary number in AL. AAD must be used before dividing two unpacked BCD digits in AX by an unpacked BCD byte.** For example, consider dividing [AX] = unpacked BCD 0508 (38 decimal) by [DH] = 07H. [AX] must first be converted to binary by using AAD. The register AX will then contain 003AH = 58 decimal. After DIV DL, [AX] = quotient = 08 unpacked BCD, [AH] = remainder = 02 unpacked BCD.
- **Consider CBW. This instruction extends the sign from the AL register to AH register.** For example, if [AL] = E2H, then after CBW, AH will contain FFH since the most significant bit of E2H is one. Note that sign extension is useful when one wants to perform an arithmetic
operation on two signed numbers of different sizes. For example, the 8-bit signed number 02H can be subtracted from 16-bit signed number 2005H as follows:

\[ 2005H - 0010 0000 00000101 = 1111 1111 11111110 \]

Another example of sign extension is that in order to multiply a signed 8-bit number by a signed 16-bit number, one must first sign-extend the signed 8-bit into a signed 16-bit number and then the instruction IMUL can be used for \( 16 \times 16 \) signed multiplication.

- AAM adjusts the product of two unpacked BCD digits in AX. If [AL] = BCD2 = 00001001, and [CH] = BCD8 = 0001 0000, then after

\[ \text{MUL CH}, [AX] = 0000000000011000 = 0018H \]

and after using AAM, [AX] = 00000010 00000010 = unpacked 24. The following instruction sequence accomplishes this:

\[ \text{MUL CH}, \text{AAM} \]

Table 3.A-6 is self-explanatory.

Explanations of Table 3.A-7 Instructions:

- **LODS** can be represented in four forms:

  For Byte  LODS BYTE or LODSB
  For Word  LODS WORD or LODSW


- REP, a one-byte prefix, can be used with MOVSW to cause the instruction MOVSW to continue executing until CX = 0. Each time the instruction is then executed, REP MOVSB or REP MOVSW can be used to move string bytes until the string length (loaded into CX before the instructions REP MOVSB or REP MOVSW) is decremented for zero.

- REP, REPZ, REPNE, REPNZ, prefix can be used with CMPS or SCAS to cause one of these instructions to continue executing until ZF = 0 (for REP, REPNE, REPNZ prefixes) and CX = 0. Note that REP, REPZ, REPNZ also provide the same purpose. If CMPS is prefixed with REP, REPZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings are equal (ZF = 0)". If CMPS is preceded by REPNE or REPNZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings not equal (ZF = 0)". Thus, repeated CMPS can be used to find matching or differing string elements. If SCAS is prefixed with REP, REPZ, the operation is interpreted as "scan while not end-of-string (CX not zero) and string element = scan-value (ZF = 1)". This form may be used to scan for departure from a given value. If SCAS is prefixed with REPNE or REPNZ, the operation is interpreted as "scan while not end-of-string (CX not zero) and string element is not equal to scan-value (ZF = 0)". This form may be used to locate a value in a string. Repeated string instructions are interruptible; the processor recognizes the interrupt before processing the next string element. Upon return from the interrupt, the repeated operation is resumed from the point of interruption. When multiple prefixes (such as LOCK and segment override) are specified in addition to any of the repeat prefixes, program execution does not resume properly upon return from interrupt.

The processor remembers only one prefix in effect at the time of the interrupt, the prefix that immediately precedes the string instruc-
tions. Upon return from interrupt, program execution resumes at this point but any additional prefixes specified are not recognized. The multiple prefixes must be used with a string instruction; maskable interrupts should be disabled for the duration of the repeated execution. However, this will not prevent a nonmaskable interrupt from being recognized.

* Note that the segment register for destination for all string instructions is always ES and cannot be overridden. However, DS can be overridden for the source using a prefix. For example, ES: MOVSB instruction uses the segment register as ES for both source and destination.

Table 3.3.8 is self-explanatory.

**Explanation of Table 3.3.9 Instructions**

All 8086 conditional branch instructions use 8-bit signed displacement. That is, the displacement covers a branch range of $-128_{10}$ to $+127_{10}$ with 0 being positive. In order to branch out of this range, the 8086 unconditional jump instructions (having direct mode) must be used.

Conditional jumps are typically used with compare instructions to find the relationship (equal to, greater than, or less than) between two numbers. The use of conditional instructions depends whether the numbers to be compared are signed or unsigned. The 8-bit number 1111 1110, when considered as signed, has a value of $-2_{16}$; the same number has a value of $+254_{16}$ when considered as unsigned. This number, when considered signed, will be smaller than zero, and when unsigned will be greater than zero. Some new terms are used to differentiate between signed and unsigned conditional transfers. For the signed numbers, the terms are used are “less than” and “greater than”; for signed numbers, the terms “less than and greater than” are used. In the above, the number 1111 1110, when considered signed is greater than $0000_{16}$ while the same number 1111 1110, when considered unsigned is greater than or equal to $0000_{16}$.

**3.4.1 SIGNED AND UNSIGNED CONDITIONAL BRANCH INSTRUCTIONS**

The table below lists these instructions:

<table>
<thead>
<tr>
<th>Signed</th>
<th>Alternate name</th>
<th>Unsigned</th>
<th>Alternate name</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE disp8</td>
<td>ZE disp8</td>
<td>JE disp8</td>
<td>ZE disp8</td>
</tr>
<tr>
<td>JZ disp8</td>
<td>(JUMP if result equal)</td>
<td>JZ disp8 (JUMP if result</td>
<td>JE disp8 (JUMP if result</td>
</tr>
<tr>
<td>JBE disp8</td>
<td>(JUMP if result equal)</td>
<td>zero)</td>
<td>equal)</td>
</tr>
<tr>
<td>JNB disp8</td>
<td>INZ disp8</td>
<td>JNB disp8</td>
<td>INZ disp8</td>
</tr>
<tr>
<td>JNL disp8</td>
<td>(JUMP if not equal)</td>
<td>JNL disp8 (JUMP if not</td>
<td>JNL disp8 (JUMP if not</td>
</tr>
<tr>
<td>JLE disp8</td>
<td>(JUMP if not equal)</td>
<td>equal)</td>
<td>equal)</td>
</tr>
<tr>
<td>JG disp8</td>
<td>ING disp8</td>
<td>JG disp8 (JUMP if not</td>
<td>JG disp8 (JUMP if not</td>
</tr>
<tr>
<td>JB disp8</td>
<td>(JUMP if less than)</td>
<td>equal)</td>
<td>equal)</td>
</tr>
<tr>
<td>JNB disp8</td>
<td>INZ disp8</td>
<td>JNB disp8 (JUMP if less</td>
<td>JNB disp8 (JUMP if less</td>
</tr>
<tr>
<td>JNL disp8</td>
<td>(JUMP if less than)</td>
<td>than)</td>
<td>than)</td>
</tr>
<tr>
<td>JLE disp8</td>
<td>ING disp8</td>
<td>JLE disp8 (JUMP if less</td>
<td>JLE disp8 (JUMP if less</td>
</tr>
<tr>
<td>JG disp8</td>
<td>(JUMP if less than)</td>
<td>than or equal)</td>
<td>than or equal)</td>
</tr>
</tbody>
</table>

**3.4.2 CONDITIONAL JUMPS AFFECTING INDIVIDUAL FLAGS**

The JP and JNP instructions use alternate names. They are JPE (Jump if Parity Even) and JPO (Jump if Parity Odd), respectively.

64.jpg
Now, let us look at the flag settings for the conditional transfer instructions concerned with the relationship between two numbers. These are listed in the table below:

### Flag Settings for Instructions Concerned with Relationships between Two Numbers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Flag setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>JEAZ</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>JNE/JNZ</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JL/JNLE</td>
<td>SF = OF = 1</td>
</tr>
<tr>
<td>JL/JGGE</td>
<td>SF = OF = 0</td>
</tr>
<tr>
<td>JC/JLLE</td>
<td>(SF + OF) OR ZF = 0</td>
</tr>
<tr>
<td>JNC/JLLE</td>
<td>(SF x + OF) OR ZF = 1</td>
</tr>
<tr>
<td>JB/JNAE</td>
<td>CF = 1</td>
</tr>
<tr>
<td>JNB/JAE</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JA/JNBE</td>
<td>(CF OR ZF) = 0</td>
</tr>
<tr>
<td>JNA/JBE</td>
<td>(CF OR ZF) = 1</td>
</tr>
</tbody>
</table>

The meanings of JAE/JZ (ZF = 1) and JNE/JNZ (ZF = 0) are obvious.

If two signed numbers are compared and if the "less than" condition is satisfied (SF = 1) with no overflow (OF = 0), the processor after execution of the JL/JNLE instruction branches to a label with the specified displacement; otherwise, the next instruction is executed. In this case, SF + OF = 1. Similarly, the meaning of JNL/JGE can be explained.

The processor when executing the JC/JLLE instruction (after comparing two signed numbers) branches to the label with the specified displacement if OF = 0 and SF = 0 (i.e., result is greater than zero). In this case, (SF + OF) OR ZF = 0. Similarly, the meaning of JNG/JLLE can be explained.

The overflow flag OF is not involved while considering unsigned numbers. After comparing two unsigned numbers, the processor when executing the JB/JNAE instruction branches to a label with the specified displacement if result is below zero (i.e., CF = 1 indicating a borrow). Similarly, the meaning of JNB/JAE can be explained.

After comparing two unsigned numbers, the processor when executing the JA/JNBE instruction branches to a label with the specified displacement if result is above zero (CF = 0) or not equal to zero (ZF = 0). Similarly, the meaning of JNA/JBE can be explained.


---

The 8086 instruction sizes vary from one to six bytes. The general 8086 instruction format is shown in Figure 3.5. The op code, register direction bit (D) and data size bit (W) in byte 1 are defined by Intel as follows:

- Op code occupies six bits and it defines the operation to be carried out by the instruction.
- Register Direction bit (D) occupies one bit. It defines whether the register operand in byte 2 is the source or destination operand. D = 1 specifies that the register operand is the destination operand; on the other hand, D = 0 indicates that the register is a source operand.
- Data size bit (W) defines whether the operation to be performed is on 8- or 16-bit data. W = 0 indicates 8-bit operation while W = 1 specifies 16-bit operation.
- The second byte of the instruction usually identifies whether one of the three conditions is satisfied.
the operands is in memory or whether both are registers; this byte contains three fields. These are the Mode (MOD) field, the Register (REG) field, and the Register/Memory (R/M) field and are defined as follows:
The 2-bit MOD field specifies whether the operand is in register or memory as follows:

<table>
<thead>
<tr>
<th>MOD</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Memory mode with no displacement follows except for 16-bit displacement when R/M = 110</td>
</tr>
<tr>
<td>01</td>
<td>Memory mode with 8-bit displacement</td>
</tr>
<tr>
<td>10</td>
<td>Memory mode with 16-bit displacement</td>
</tr>
<tr>
<td>11</td>
<td>Register mode (no displacement)</td>
</tr>
</tbody>
</table>

REG field occupies 3 bits. It defines the register for the first operand which is specified as the source or destination by the D-bit (byte 1). The definition of REG and W fields are given below:

<table>
<thead>
<tr>
<th>REG</th>
<th>W = 0</th>
<th>W = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AL</td>
<td>AL</td>
</tr>
<tr>
<td>001</td>
<td>CL</td>
<td>CL</td>
</tr>
<tr>
<td>010</td>
<td>DL</td>
<td>DL</td>
</tr>
<tr>
<td>011</td>
<td>BL</td>
<td>BL</td>
</tr>
<tr>
<td>100</td>
<td>AH</td>
<td>AH</td>
</tr>
<tr>
<td>101</td>
<td>CH</td>
<td>CH</td>
</tr>
<tr>
<td>110</td>
<td>DH</td>
<td>DH</td>
</tr>
<tr>
<td>111</td>
<td>BH</td>
<td>BH</td>
</tr>
</tbody>
</table>

The R/M field occupies 3 bits. The R/M field along with the MOD field defines the second operand as shown below:

### MOD 11

<table>
<thead>
<tr>
<th>MOD</th>
<th>Effective address calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R/M</td>
</tr>
<tr>
<td>000</td>
<td>AL</td>
</tr>
<tr>
<td>001</td>
<td>CL</td>
</tr>
<tr>
<td>010</td>
<td>DL</td>
</tr>
<tr>
<td>011</td>
<td>BL</td>
</tr>
<tr>
<td>100</td>
<td>AH</td>
</tr>
<tr>
<td>101</td>
<td>CH</td>
</tr>
<tr>
<td>110</td>
<td>DH</td>
</tr>
<tr>
<td>111</td>
<td>BH</td>
</tr>
</tbody>
</table>

In the above, encoding of the R/M field depends on how the mode field is set. If MOD = 11 (register-to-register mode), then R/M identifies the second register operand. If MOD selects memory mode, then R/M indicates how the effective address of the memory operand is to be calculated.

- **Bytes 3 through 6** of an instruction are optional fields that normally contain the displacement value of a memory operand and/or the actual value of an immediate constant operand. As an example, consider the instruction MOV CH, BL. This instruction transfers the 8-bit content of BL into CH. We will determine the machine code of this instruction. The 6-bit op code for this instruction is 100010. The D-bit indicates whether the register specified by the REG field of byte 2 is a source or destination operand. Let us define the BL in the REG field of byte 2. D = 0 indicates that the REG field of the next byte is the source operand. The W-bit of byte 1 is 0 since this is a byte operation.

In byte 2, since the second operand is a register, MOD field is 31. The R/M field = 101, specifies that the destination register is CH and, therefore, R/M = 101. Hence the machine code for MOV CH, BL is 10001001 11011011

**BYTE 1**

<table>
<thead>
<tr>
<th>MOD</th>
<th>Effective address calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R/M</td>
</tr>
<tr>
<td>110</td>
<td>DH</td>
</tr>
<tr>
<td>111</td>
<td>BH</td>
</tr>
</tbody>
</table>

= 89D16
As another example, consider SUB BX, [DI]. This instruction subtracts the 16-bit content of memory location addressed by DI and DS from BX. The 6-bit op code for SUB is 001010. D = 1 so that the REG field of byte 2 is the destination operand and W = 1 indicates 16-bit operation. Therefore, byte 1 = 0010 1011

3.6 8086 ASSEMBLER-DEPENDENT INSTRUCTIONS

Some 8086 instructions do not define whether an 8-bit or 16-bit operation is to be executed. Instructions with one of the 8086 registers as an operand typically define the operation as 8-bit or 16-bit based on the register size. An example is MOV CL, [BX] which moves an 8-bit number with the offset defined by [BX] in DS into register CL; MOV CX, [BX], on the other hand, moves the number from offsets [BX] and [BX + 1] in DS into CX.

Instructions with single-memory operand may define 8-bit or 16-bit operation by adding B or W for word with the mnemonic. Typical examples are MULB [BX] and CMPB [ADDR]. The string instructions may define this in two ways. Typical examples are MOVSB or MOVSW or MOVSW or MOVSW WORD for 8-bit. Memory offsets can also be specified by including BYTE PTR for 8-bit and WORD PTR for 16-bit with the instruction. Typical examples are INC BYTE PTR [BX] and INC WORD PTR [BX].

3.7 ASM-86 ASSEMBLER PSEUDOINSTRUCTIONS

The ASM-86 is the assembler written by Intel for the 8086 microprocessor. The ASM-86 allows the programmer to assign the values of CS, DS, SS, and ES. One of the requirements of the ASM-86 assembler is that a variable's type must be declared as byte (8-bit), word (16-bit), or double word (4 bytes or 2 words) before using in a program. Some examples are included below:

START DB 0 ; START is declared ; as a byte offset ; and initialized to zero.

BEGIN DW 0 ; BEGIN is declared ; as a word offset ; and initialized to zero.

NAME DD 0 ; NAME is declared ; as a double word ; (4 bytes) offset ; and initialized to zero.

The EQU directive can be used to assign a name to constants. For example, the statement JOHN EQU 21H directs the assembler to assign the value 21H every time it finds JOHN in the program. This means that the assembler reads the statement MOV BH, JOHN as MOV BH, 21H. As mentioned before, DB, DW, and DD are the ASM-86 directives used to assign names and specify data types for variables in a program. For example, after execution of the statement MOV DS: [BX], the ASM-86 assembler assigns 50H to the offset name BOB and 20H to the offset name BOB + 1. This means that the program can use the instruction MOV BX, BOB to load the 16-bit content of memory starting at the offset BOB in DS into BX. The DW acts as a single word, and the DB acts as a single byte. The DS acts as a single byte, and the DS acts as a single word. The DS acts as a single word, and the DS acts as a single word.
MOV [DI], BH
; Source is in register
; mode and destination
; is in register
; indirect mode.

MOV BH, VALUE
; Source is in register
; indirect with
; displacement mode
; and destination is
; in register mode
; VALUE is typically
; defined by the
; EQU directive prior
; to this instruction.

MOV AX, [DI + 4]
; Source is in
; indexed with
; displacement mode and
; destination is in
; register mode.

MOV SI, [BP + 2]
; Source is in
; bused indexed
; with displacement
; mode and
; destination is in
; register mode.

OUT 30H, AL
; Source is in
; register mode and
; destination is
; in direct port
; mode.

IN AX, DX
; Source is in
; indirect port mode and
; destination is
; in register mode.

Note that in the above, START must be previously defined by a data allocation statement such as DB and DW.

In the following, typical ASM-86 assembler directives such as SEGMENT, ENDS, ASSUME, and DUP will be discussed.

### 3.7.1 SEGMENT AND ENDS
A section of a program or data array can be defined by the SEGMENT and ENDS directives as follows:

```assembly
SEGMENT
    X1 DB 0
    X2 DB 0
    X3 DB 0

ENDS
```

The segment name is JOHN. The assembler will assign a numeric value to JOHN corresponding to the base value of the data segment. The programmer must use the 8086 instructions to load JOHN into DS as follows:

```
MOV BX, JOHN
MOV DS, BX
```

Note that the segment registers must be loaded via a 16-bit register such as AX or by content of a memory location.

### 3.7.2 ASSUME DIRECTIVE
As mentioned before, the 8086, at any time, can directly address four physical segments which include a code segment, a data segment, a stack segment, and an extra segment. The 8086 may contain a number of logical segments containing codes, data, and stack. The ASSUME pseudo-instruction assigns a logical segment to a physical segment at any given time. That is, the ASSUME directive tells the ASM-86 assembler what addresses will be in the segment registers at execution time.

For example, the statement ASSUME CS: PROGRAM 1, DS: DATA 1, SS: STACK 1 directs the assembler to use the logical code segment PROGRAM 1 as CS containing the instructions, the logical data segment DATA 1 as DS containing data, and the logical stack segment STACK 1 as SS containing the stack.

### 3.7.3 DUP DIRECTIVE
The DUP directive can be used to initialize several locations to zero. For example, the statement START DW 4 DUP (0) reserves four words.
starting at the offset START in DS and initializes them to zero. The DUP directve can also be used to reserve several locations which need not be initialized. A question mark must be used with DUP in this case. For example, the statement BEGIN DB 100 DUP (?) reserves 100 bytes of uninitialized data space to an offset BEGIN in DS. Note that BEGIN should be typed in the label field, DB in the op field, and 100 DUP (?) in the operand field.

A typical example illustrating the use of these directives is given below:

```
DATA 1 SEGMENT
ADDR 1   DW 3005H
ADDR 2   DW 2003H
DATA 1 ENDS
STACK 1 SEGMENT
DW 60 DUP (0)  ; Assign 60 bytes of stack with zeros.
STACK-TOP LABEL WORD  ; Initialize Stack-Top
            ; to the next location after the top of the stack.
STACK 1 ENDS
CODE 1 SEGMENT
ASSUME CS: CODE 1, DS: DATA 1, SS:
STACK 1
MOV AX, STACK 1
MOV SS, AX
LEA SP, STACK-TOP
MOV AX, DTDATA 1
MOV DS, AX
LEA SI, ADDR 1
LEA DI, ADDR 2

{ Main Program
  Body
}

CODE 1 ENDS
```

Note that LABEL is a directive used to initialize STACK-TOP to the next location after the top of the stack. The statement STACK-TOP LABEL WORD gives the name STACK-TOP to the next address after the 60 words are set aside for the stack. The WORD in this statement indicates that PUSH into and POP from the stack are done in words.

In the above program, the ASSUME directive tells the assembler the

names of the logical segments to use as code segment, data segment, and stack segment. The extra segment can be assigned a name in a similar manner. When the instructions are executed, the displacements in the instructions along with the segment register contents are used by the assembler to generate the 20-bit physical addresses. The segment register, other than the code segment, must be initialized by instructions before they are used to access data.

When the ASM-86 assembler translates an assembly language program, it computes the displacement, or offset, of each instruction code byte from the start of a logical segment that contains it. For example, in the above program the CS: CODE 1 in the ASSUME statement directs the assembler to compute the offsets or displacements by the following instructions from the start of the logical segment CODE 1. This means that when the program is run, the CS will contain the 16-bit value where the logical segment CODE 1 is located in memory. The assembler keeps track of the instruction byte displacements which are loaded into IP. The 20-bit physical address generated from CS and IP are used to fetch each instruction.

Another example to store data bytes in a data segment and to allocate stack is given in the following:

```
DSEG SEGMENT
ARRAY DB 02H, 01H, A2H  ; Store 3 bytes
                     ; of data in an
DSEG ENDS
                     ; address defined
                     ; by DSEG as DS
                     ; and ARRAY as
                     ; offset
SSEG SEGMENT
DW 10 DUP (0)  ; Allocate
              ; 10 word stack
STACK-TOP LABEL WORD  ; Label initial
                ; TOS
SSEG ENDS

; Initialize
MOV AX, DSEG
MOV DS, AX
MOV AX, SSEG
MOV SS, AX
```

69.jpg
MOV SP, STACK-TOP ; Initialize SP

A logical segment is not normally given a physical starting address when it is declared. After the program is assembled, the programmer uses the linker to assign physical address.

Example 3.1
Determine the effect of each one of the following 8086 instructions:

i) PUSH [BX]
ii) DIV DH
iii) CWD
iv) MOVSB
v) MOV START [BX], AL

Assume the following data prior to execution of each one of the above instructions independently. Assume all numbers in hexadecimal.

| [DS]  | 3000H | | [SI]  | 0400H |
| [ES]  | 5000H | | [DI]  | 0500H |
| [DX]  | 0400H | | DF    | 0      |
| [SP]  | 5000H | | [BX]  | 6000H |
| [SS]  | 6000H | | Value of START = 05H |
| [AX]  | 00A9H |
| [BX]  | 36000H |
| [BX]  | 02H, [36001H] = 03H |
| [DX]  | 50500H |
| [BX]  | 02H, [30401H] = 03H |

Solution

i) 20-bit physical memory addressed by DS and BX = 36000H. 20-bit physical location pointed to by SP and SS = 65000H. PUSH [DX] pushes [36001H] and [36000H] into stack locations 64FFFH and 64FEFH, respectively. The SP is then decremented by 2 to contain the 20-bit physical address 64FEFH (SS = 6000H, SP = 4FFFH). Therefore, [64FFFH] = 03H and [64FEFH] = 02H.


Example 3.2
Write 8086 assembly program to clear 100 consecutive bytes. Assume CS and DS are already initialized.

Solution
LEA BX, ADDR ; Initialize BX
MOV CX, 100 ; Initialize loop count
START MOV [BX], 0H ; Clear memory byte
INC BX ; Update pointers
LOOP START ; Decrement CX and loop
HLT

Example 3.3
Write 8086 assembly program to compute \( \sum_{i=1}^{n} \) where \( Xi \) and \( Yi \) are signed 8-bit numbers. \( N = 100 \). Assume CS and DS are already initialized. Assume no overflow.

Solution
MOV CX, 100 ; Initialize loop count
LEA BX, ADDR 1 ; Load ADDR 1 into BX
LEA SI, ADDR 2 ; Load ADDR 2 into SI
MOV DX, 0000H ; Initialize sum to zero
START MOV AL, [BX] ; Load data into AL
IMUL [SI] ; Signed multiplication
AOR DX, AX ; Sum Xi Yi

70.jpg
Example 3.4
Write an 8086 assembly language program to add two words; each word contains four packed BCD digits. The first word is stored in two consecutive locations with the low byte at the offset pointed by SI at offset 0500H, while the second word is stored in two consecutive locations with the low byte pointed by BX at the offset 1000H. Store the result in BX. Assume DS and ES are already initialized.

**Solution**

```
MOV CX, 2       ; Initialize loop count
MOV SI, 0500H   ; Initialize SI
MOV BX, 1000H   ; Initialize BX
CLC              ; Clear carry
START
    MOV AL, [SI]      ; Move data
    ADC AL, [BX]      ; Perform addition
    DAA              ; BCD adjust
    MOV [BX], AL      ; Store result
    INC SI           ; Update
    INC BX           ; Pointers
    LOOP START      ; Decrement CX and loop
HALT
```

Example 3.5
Write an 8086 assembly language program to add two words; each contains two ASCII digits. The first word is stored in two consecutive locations with the low byte pointed to by SI at offset 0300H, while the second byte is stored in two consecutive locations with the low byte pointed to by DI at offset 0700H. Store the result in DI. Assume DS and ES are already initialized.

**Solution**

```
MOV CX, 2       ; Initialize loop count
MOV SI, 0300H   ; Initialize SI
```

Example 3.6
Write an 8086 assembly language program to compare a source string of 50H words pointed to by an offset of 2000H in DS with a destination string pointed to by an offset 3000H in DS. The program should be halted as soon as a match is found or the end of string is reached. Assume CS and ES are initialized.

**Solution**

```
MOV SI, 2000H    ; Initialize SI
MOV DI, 3000H    ; Initialize DI
MOV CX, 50       ; Initialize CX
CLD              ; DF is cleared
                ; so that SI and DI
                ; will autoincrement
                ; after compare
REPNE CMPSW      ; Repeat CMPSW
                ; unit CX = 0 or
                ; until compared
                ; words are equal
HALT             ; STOP
```

Example 3.7
Write a subroutine in 8086 assembly language which can be called by a main program in a different code segment. The subroutine will multiply a signed 16-bit number in CX by a signed 8-bit number in AL. The main program will call this subroutine, store the result in two consecutive memory words, and stop. Assume SI and DI contain the signed 8-bit and 16-bit data, respectively.
3.8 SYSTEM DESIGN USING 8086

This section covers the basic concepts associated with interfacing the 8086 to its support chips such as memory and I/O. Topics such as timing diagrams and 8086 pins and signals will also be included.

3.8.1 PINS AND SIGNALS

The 8086 pins and signals are shown in Figure 3.6. Unless otherwise indicated, all 8086 pins are TTL compatible. As mentioned before, the 8086 can operate in two modes. These are minimum mode (uniprocessor system — single 8086) and maximum mode (multiprocessor system — more than one 8086). MN/MX is an input pin used to select one of these modes. When MN/MX is HIGH, the 8086 operates in the minimum mode. In this mode, the 8086 is configured (that is, pins are defined) to support small, single processor systems using a few devices that use the system bus.

When MN/MX is LOW, the 8086 is configured (that is, pins are defined in the maximum mode) to support multiprocessor systems. In this case, the Intel 8288 bus controller is added to the 8086 to provide bus controls and compatibility with the multiprocessor architecture. Note that in a particular application, the MN/MX must be used to either HIGH or LOW. AD0-AD15 lines are a 16-bit multiplexed address/data bus. During the first clock cycle AD0-AD15 are the low order 16 bits of address. The 8086 has a total of 20 address lines. The upper four lines are multiplexed with the status signals for the 8086. These are the A16/S3, A17/S4, A18/S5, and A19/S6. During the first clock period of a bus cycle (read or write cycle), the entire 20-bit address is available on these lines. During all other clock cycles for memory and I/O operations, AD15-AD0 contain the 16-bit data, and S3, S4, S5, and S6 become status lines. S8 and S4 lines are decoded as follows:

<table>
<thead>
<tr>
<th>A17/S4</th>
<th>A16/S3</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Extra segment</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Stack segment</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Code or no segment</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Data segment</td>
</tr>
</tbody>
</table>

Note that in the above programs SEGMENT, ASSUME, ENDP, and ENDS are used as assembler directives.
System Design Using 8086

Therefore, after the first clock cycle of an instruction execution, the A17/S4 and A16/S3 pins specify which segment register generates the segment portion of the 8086 address. Thus, by decoding these lines and then using the decoder outputs as chip selects for memory chips, up to 4 megabytes (one megabyte per segment) can be provided. This provides a degree of protection by preventing erroneous write operations to one segment from overwriting another segment and destroying information in that segment. A18/S5 and A19/S6 are used as A18 and A19, respectively, during the first clock period of an instruction execution. If an I/O instruction is executed, they stay low during the first clock period. During all other cycles, A18/S5 indicates the status of the 8086 interrupt enable flag and A19/S6 becomes S6, and a low A19/S6 pin indicates that the 8086 is on the bus. During a “Hold acknowledge” clock period, the 8086 tristates the A19/S6 pin and thus allows another bus master to take control of the system bus.

The 8086 tristates A00-AD15 during Interrupt Acknowledge or Hold Acknowledge cycles.

BHE/S7 is used as BHE (Bus High Enable) during the first clock cycle of an instruction execution. The 8086 outputs a low on this pin during read, write, and interrupt acknowledge cycles in which data are to be transferred in a high-order byte (AD0-AD8) of the data bus. BHE can be used in conjunction with AD0 to select memory banks. A thorough discussion is provided later. During all other cycles BHE/S7 is used as S7 and the 8086 maintains the output level (BHE) of the first clock cycle on this pin.

RD is LOW whenever the 8086 is reading data from memory or an I/O location.

TEST is an input pin and is only used by the WAIT instruction. The 8086 enters a wait state after execution of the WAIT instruction until a LOW is seen on the TEST pin. This input is synchronized internally during each clock cycle on the leading edge of the CLK pin.

INTR is the maskable interrupt input. This line is not latched and, therefore, INTR must be held at a HIGH level until recognized to generate an interrupt.

NMI is the nonmaskable interrupt input activated by a leading edge. RESET is the system reset input signal. This signal must be high for at least four clock cycles to be recognized, except after power-on which requires a 50-μs reset pulse. It causes the 8086 to initialize registers DS, SS, ES, IP and flags to all zeros. It also initializes CS to FFH. Upon removal of the RESET signal from the RESET pin, the 8086 will fetch its next instruction from 20-hc physical address FFH (CS = FFH, IP = 000H).

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD15-AD0</td>
<td>Address/Data Bus</td>
<td>Bidirectional, 2-State</td>
</tr>
<tr>
<td>A19/S6</td>
<td>Address/Status</td>
<td>Output, 3-State</td>
</tr>
<tr>
<td>A16/S3</td>
<td>Bus High Enable</td>
<td>Output, 3-State</td>
</tr>
<tr>
<td>BHE/S7</td>
<td>Status</td>
<td>Input, 3-State</td>
</tr>
<tr>
<td>M/V/MX</td>
<td>Minimum/Maximum Mode</td>
<td>Input, 3-State</td>
</tr>
<tr>
<td>RD</td>
<td>Read Control</td>
<td>Output, 3-State</td>
</tr>
<tr>
<td>TEST</td>
<td>Wait On Test Control</td>
<td>Input, 3-State</td>
</tr>
<tr>
<td>READY</td>
<td>Wait State Control</td>
<td>Input, 3-State</td>
</tr>
<tr>
<td>RESET</td>
<td>System Reset</td>
<td>Input, 3-State</td>
</tr>
<tr>
<td>NMII</td>
<td>Non-Maskable</td>
<td>Input, 3-State</td>
</tr>
<tr>
<td>INTR</td>
<td>Interrupt Request</td>
<td>Input, 3-State</td>
</tr>
<tr>
<td>CLK</td>
<td>System Clock</td>
<td>Input, 3-State</td>
</tr>
<tr>
<td>VSS</td>
<td>Ground</td>
<td>Input, 3-State</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLD</td>
<td>Hold Request</td>
<td>Input, 3-State</td>
</tr>
<tr>
<td>HLC</td>
<td>Hold Acknowledge</td>
<td>Output, 3-State</td>
</tr>
<tr>
<td>WR</td>
<td>Write Control</td>
<td>Output, 3-State</td>
</tr>
<tr>
<td>WRD</td>
<td>Memory I/O Control</td>
<td>Output, 3-State</td>
</tr>
<tr>
<td>DT/R</td>
<td>Data Transfer/Read</td>
<td>Output, 3-State</td>
</tr>
<tr>
<td>DEN</td>
<td>Data Enable</td>
<td>Output, 3-State</td>
</tr>
<tr>
<td>ALE</td>
<td>Address Latch</td>
<td>Output, 3-State</td>
</tr>
<tr>
<td>INTA</td>
<td>Interrupt Acknowledge</td>
<td>Output, 3-State</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ/ST1</td>
<td>Request/Grant Bus</td>
<td>Bidirectional</td>
</tr>
<tr>
<td>LOCK</td>
<td>Access Control</td>
<td>Output, 3-State</td>
</tr>
<tr>
<td>BML</td>
<td>Bus Priority Lock Control</td>
<td>Output, 3-State</td>
</tr>
<tr>
<td>BS</td>
<td>Bus Cycle Status</td>
<td>Output, 3-State</td>
</tr>
<tr>
<td>OSL</td>
<td>Instruction Queue Status</td>
<td>Output, 3-State</td>
</tr>
</tbody>
</table>

FIGURE 3.8 Pin definition.
When the 8086 detects the positive going edge of a pulse on RESET, it stops all activities until the signal goes LOW. When the reset is low, the 8086 initializes the system as follows:

<table>
<thead>
<tr>
<th>8086 component</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flags</td>
<td>Clear</td>
</tr>
<tr>
<td>IP</td>
<td>0000H</td>
</tr>
<tr>
<td>CS</td>
<td>FFFFH</td>
</tr>
<tr>
<td>DS</td>
<td>0000H</td>
</tr>
<tr>
<td>SS</td>
<td>0000H</td>
</tr>
<tr>
<td>ES</td>
<td>0000H</td>
</tr>
<tr>
<td>Queue</td>
<td>Empty</td>
</tr>
</tbody>
</table>

The reset signal to the 8086 can be generated by the 8284. The 8284 has a Schmitt Trigger input (RES) for generating reset from a low active external reset.

To guarantee reset from power-up, the reset input must remain below 1.05 volts for 50 microseconds after Vcc has reached the minimum supply voltage of 4.5V. The RES input of the 8284 can be driven by a simple RC circuit as shown in Figure 3.7.

The values of R and C can be selected as follows:

\[ Vc(t) = V \left(1 - \exp \left(-t/(RC)\right)\right) \]

where \( t = 50 \) microseconds, \( V = 4.5 \) V, \( Vc = 1.05 \) V, and \( RC = 188 \) microseconds. For example, if \( C \) is chosen arbitrarily to be \( 0.1 \) \( \mu F \), then \( R = 1.88 \) KΩ.

![Figure 3.7 8086 reset and system reset.](image)

As mentioned before, the 8086 can be configured in either minimum or maximum mode using the MN/MX input pin. In minimum mode, the 8086 itself generates all bus control signals. These signals are:

- **DT/R (Data Transmit/Receive).** DT/R is an output signal required in minimum system that uses an 8286/8257 data bus transceiver. It is used to control direction of data flow through the transceiver.
- **DEN (Data Enable) is provided as an output enable for the 8286/8257 in a minimum system which uses the transceiver.**
- **DEN is active LOW during each memory and I/O access and for INTA cycles.**
- **ALE (Address Latch Enable) is an output signal provided by the 8286 and can be used to demultiplex the A0-A15 into A0-A15 and D0-D15 at the falling edge of ALE.** The 8086 ALE signal is same as the 8085 ALE.
- **M/IO.** This 8086 output signal is similar to the 8085 J0/M. It is used to distinguish a memory access (M/IO = HIGH) from an I/O access (M/IO = LOW). When the 8086 executes an I/O instruction such as INF or OUT, it outputs a LOW on this pin. On the other hand, the 8086 outputs HIGH on this pin when it executes a memory reference instruction such as MOVT AX, [SI].
- **WR.** The 8086 outputs LOW on this pin to indicate that the processor is performing a write memory or write I/O operation, depending on the M/IO signal.
- **INTA.** The 8086 INTA is similar to the 8085 INTA. For interrupt Acknowledge cycles (for INTR pin), the 8086 outputs LOW on this pin.
- **HOLD (input), HLDA (output).** These pins have the same purpose as the 8085 HOLD/HLDA pins and are used for DMA. A HIGH on the HOLD pin indicates that another master is requesting to take over the system bus. The processor receiving the HOLD request will output HLDA high as an acknowledgment. At the same time, the processor tristates the system bus. Upon receipt of LOW on the HOLD pin, the processor places LOW on the HLDA pin. HOLD is not an asynchronous input. External synchronization should be provided if the system cannot otherwise guarantee the setup time.
- **CLK (input) provides the basic timing for the 8086 and bus controller.**

The maximum clock frequencies of the 8086-4, 8086, and 8086-
2 are 4 MHz, 5 MHz, and 8 MHz, respectively. Since the design of these processors incorporates dynamic cells, a minimum frequency of 2 MHz is required to retain the state of the machine. The 8086-4, 8086, and 8086-2 will be referred to as 8086 in the following. Maximum frequency requirement, single stepping, or cycling of the 8086 may not be accomplished by disabling the clock. Since the 8086 does not have on-chip clock generation circuitry, an 8284 clock generator chip must be connected to the 8086 CLK pin as shown in Figure 3.8.

![Diagram of 8284 clock generator connections to the 8086.](image)

The crystal must have a frequency three times the 8086 internal frequency. That is, the 8284 divides the crystal clock frequency by 3. In other words, to generate 5 MHz 8086 internal clock, the crystal clock must be 15 MHz. To select the crystal inputs of the 8284 as the frequency source for clock generation, the E/C input must be strapped to ground. This strapping option allows either the crystal or the external frequency input as the source for clock generation. When selecting a crystal for use with the 8284, the crystal series resistance should be as low as possible. The oscillator delays in the 8284 appear as inductive elements to the crystal and cause the 8284 to run at a frequency below that of the pure series resistance. A capacitor CL should be placed in series with the crystal and the X2 pin of the 8284. This capacitor cancels the inductive element. The impedance of the capacitor XC = 1/2πfCL, where F is the crystal frequency. It is recommended that the crystal series resistance plus XCL be kept less than 1 kohm. As the crystal frequency increases, CL should be decreased. For example, a 12-MHz crystal may require CL = 24 PF, while 22 MHz may require CL = 8 PF. If very close correlation with the pure series resistance is not necessary, a nominal CL value of 12 to 15 PF may be used with a 15-MHz crystal. Two crystal manufacturers recommended by Intel are Crystek Corp. model CY 15A (15 MHz) and CTS Knight Inc. model CY 24A (24 MHz).

Note that the 8284 can be used to generate the 8086 READY input signal based on inputs from slow memory and I/O devices which are not capable of transferring information at the 8086 rate.

In the maximum mode, some of the 8086 pins in the minimum mode are redefined. For example, pins HOLD, HLD, WR, M/IO, DT/R, DEN, AL, and INTA in the minimum mode are redefined as RQ/GT0, RQ/GT1, LOCK, S2, S1, S0, QS0, and QS1, respectively. In maximum mode, the 8288 bus controller decodes the status information from S0, S1, S2 to generate bus timing and control signals required for a bus cycle. S1, S2, S0 are 8086 outputs and are decoded as follows:

<table>
<thead>
<tr>
<th>S2</th>
<th>S1</th>
<th>S0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Interrupt Acknowledge</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Read I/O port</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Write I/O port</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Halt</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Code access</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Read memory</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Write memory</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Inactive</td>
</tr>
</tbody>
</table>

RQ/GT0, RQ/GT1. These request/grant pins are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. Each pin is bidirectional, with RQ/GT0 having higher priority than RQ/GT1. These pins have internal pull-up resistors so that they may be left unconnected. The request/grant function of the 8086 works as follows:

1. A pulse (one clock wide) from another local bus master (RQ/GT0 or RQ/GT1 pins) indicates a local bus request to the 8086.
2. At the end of 8086 current bus cycle, a pulse (one clock wide) from the 8086 to the requesting master indicates that the 8086 has relinquished the system bus and tristated the outputs. Then the new bus master subsequently relinquishes control of the system bus by sending a LOW on RQ/GT0 or RQ/GT1 pins. The 8086 then regains bus control.

LOCK. The 8086 outputs LOW on the LOCK pin to prevent other bus masters from gaining control of the system bus. The LOCK signal...
is activated by the ‘LOCK’ prefix instruction and remains active until the completion of the instruction that follows.

- Q51, Q50. The 8086 outputs to Q51 and Q50 pins to provide status to allow external tracking of the internal 8086 instruction queue as follows:

<table>
<thead>
<tr>
<th>Q51</th>
<th>Q50</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No operation</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>First byte of op code from queue</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Empty the queue</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Subsequent byte from queue</td>
</tr>
</tbody>
</table>

Q50 and Q51 are valid during the clock period following any queue operation. The 8086 can be operated from a +5V to +10V power supply. There are two ground pins on the chip to distribute power for noise reduction.

### 3.8.2 8086 Basic System Concepts

This section describes basic concepts associated with 8086 bus cycles, address and data bus, system data bus, and multiprocessor environment.

#### 3.8.2.1 8086 Bus Cycle

In order to communicate with external devices via the system bus for transferring data or fetching instructions, the 8086 executes a bus cycle. The 8086 basic bus cycle timing diagram is shown in Figure 3.9. The minimum bus cycle contains four CPU clock periods called T States. The bus cycle timing diagram depicted in Figure 3.9 can be described as follows:

1. During the first T State (T1), the 8086 outputs the 20-bit address computed from a segment register and an offset on the multiplexed address/data/status bus.
2. For the second T State (T2), the 8086 removes the address from the bus and either tri-states or activates the AD15-AD0 lines in preparation for reading data via AD15-AD0 lines during the T3 cycle. In case of a write bus cycle, the 8086 outputs data on AD15-AD0 lines. Also, during T2, the upper four multiplexed bus lines switch from address (A19-A16) to bus cycle status (S0, S5, S4, S3). The 8086 outputs LOW on RD (for read cycle) or WR (for write cycle) during portions of T2, T3, and T4.

#### 3.8.2.2 Bus Cycle Timing

During T3, the 8086 continues to output status information on the four A19-A16/S6-S3 lines and will either continue to output write data or input read data to or from AD15-AD0 lines. If the selected memory or I/O device is not fast enough to transfer data at the 8086, the memory or I/O device activates the 8086's READY input line low by the start of T3. This will force the 8086 to insert additional clock cycles (wait states TW) after T3. Bus activity during TW is the same as T3. When the selected device has had sufficient time to complete the transfer, it must activate the 8086 READY PIN HIGH. As soon as the TW clock periods end, the 8086 executes the last bus cycle, T4. The 8086 will latch data on AD15-AD0 lines during the last wait state or during T4 if no wait states are requested.
4. During T4, the 8086 disables the command lines and the selected memory and I/O devices from the bus. Thus, the bus cycle is terminated in T4. The bus cycle appears to devices in the system as an asynchronous event consisting of an address to select the device, a register or memory location within the device, a read strobe, or a write strobe along with data.

5. DEN and DT/R pins are used by the 8286/8287 transceiver in a minimum system. During the read cycle, the 8086 outputs DEN LOW during part of T2 and all of T3 cycles. This signal can be used to enable the 8286/8287 transceiver. The 8086 outputs LOW on the DT/R pin from the start of T1 and part of T4 cycles. The 8086 uses this signal to receive (read) data from the receiver during T3-T4. During a write cycle, the 8086 outputs DEN LOW during part of T1, all of T2 and T3, and part of T4 cycles. The signal can be used to enable the transceiver. The 8086 outputs HIGH on DT/R through-out the four bus cycles to transmit (write) data to the transceiver during T3-T4.

3.8.2.8 8086 Address and Data Bus Concepts

The majority of memory and I/O chips capable of interfacing to the 8086 require a stable address for the duration of the bus cycle. Therefore, the address on the 8086 multiplexed address/data bus during T1 should be latched. The latched address is then used to select the desired I/O or memory location. Note that the 8086 has a 16-bit multiplexed address and data bus, while the 8085's 8-bit data lines and LOW address byte are multiplexed. Hence, the multiplexed bus components of the 8085 family are not applicable to the 8086. To demultiplex the bus, the 8086 provides an ALE (Address Latch Enable) signal to capture the address in either the 8282 (noninverting) or 8283 (inverting) 8-bit bistable latches. These latches propagate the address through to the outputs while ALE is HIGH and latch the address in the following edge of ALE. This only delays address access and chip select decoding by the propagation delay of the latch. Figure 3.10 shows how the 8086 demultiplexes the address and data buses.

The programmer views the 8086 memory address space as a sequence of one million bytes in which any byte may contain an eight-bit data element and any two consecutive bytes may contain an 16-bit data element. There is no constraint on byte or word addresses (boundaries). The address space is physically implemented on a 16-bit data bus by dividing the address space into two banks of up to 512K bytes as shown in Figure 3.11. These banks can be selected by BHE and A0 as follows:

<table>
<thead>
<tr>
<th>BHE</th>
<th>A0</th>
<th>Byte transferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Both bytes</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Upper byte to/from odd address</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Lower byte to/from even address</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>None</td>
</tr>
</tbody>
</table>

One bank is connected to D7-D0 and contains all even addressed bytes (A0 = 0). The other bank is connected to D15-D8 and contains odd addressed bytes (A0 = 1). A particular byte in each bank is addressed by A19-A11. The even addressed bank is enabled by LOW A0 and data bytes transferred over D7-D0 lines. The 8086 outputs HIGH on BHE (Bus High Enable) and thus disables the odd-addressed bank. The 8086 outputs
LOW on BHE to select the odd-addressed bank and HIGH on A0 to disable the even-addressed bank. This directs the data transfer to the appropriate half of the data bus. Activation of A0 and BHE is performed by the 8086 depending on odd or even addresses and is transparent to the programmer. As an example, consider execution of the instruction MOV DH, [BX]. Suppose the 20-bit address computed by BX and DS is even. The 8086 outputs LOW on A0 and HIGH on BHE. This will select the even-addressed bank. The content of the selected memory is placed on the D7-D0 lines by the memory chip. The 8086 reads this data via D7-D0 and automatically places it in DH. Next, consider accessing a 16-bit word by the 8086 with low byte at an even address as shown in Figure 3.12.

For example, suppose that the 8086 executes the instruction MOV [BX], CX. Assume [BX] = 0004H, [DS] = 2000H. The 20-bit physical address for the word is 20004H. The 8086 outputs LOW on both A0 and BHE, enabling both banks simultaneously. The 8086 outputs [CL] to D7-D6 lines and [CH] to D15-D8 lines with WR LOW and M/IO HIGH. The enabled memory banks obtain the 16-bit data and write [CL] to location 20004H and [CH] to location 20005H. Next, consider accessing an odd-addressed 16-bit word by the 8086. For example, suppose the 20-bit physical address computed by the 8086 is 20005H. The 8086 accomplishes this transfer in two bus cycles. In the first bus cycle, the 8086 outputs HIGH on A0, LOW on BHE, and thus enables the odd-addressed bank and disables the even-addressed bank. The 8086 also outputs LOW on RD and HIGH on M/IO pins. In this bus cycle, the odd memory bank places [20005H] on D15-D8 lines. The 8086 reads this data into CL. In the second bus cycle, the 8086 outputs LOW on A0, HIGH on BHE, and thus enables the even-addressed bank and disables the odd-addressed bank.

During a byte read, the 8086 floats the entire D15-D0 lines during portions of T2 cycle even though data are expected on the upper or lower half of the data bus. As will be shown later, this action simplifies the chip select decoding requirements for ROMs and EPROMs. During a byte write, the 8086 will drive the entire 16-bit data bus. The information on the half of the data bus not transferring data is indeterminate. These concepts also apply to I/O transfers.

If memory or I/O devices are directly connected to the multiplexed bus, the designer must guarantee that the devices do not corrupt the address on the bus during T1. To avoid this, the memory or I/O devices should have an output enable controlled by the 8086 read signal. This is shown in Figure 3.14.
The 8086 timing guarantees that the read is not valid until after the address is latched by ALE as shown in Figure 3.15.

All Intel peripherals, EPROMs, and RAMs for microprocessors provide output enable for read inputs to allow connection to the multiplexed bus. Several techniques are available for interfacing the devices without output enables to the 8086 multiplexed bus. However, these techniques will not be discussed here.

3.8.3 INTERFACING WITH MEMORIES

Figure 3.16 shows a general block diagram of an 8086 memory array. In Figure 3.16, the 16-bit word memory is partitioned into high and low 8-bit banks on the upper and lower halves of the data bus selected by BEH and A0.

3.8.3.1 ROM and EPROM

ROMs and EPROMs are the simplest memory chips to interface to the 8086. Since ROMs and EPROMs are read-only devices, A0 and BEH are not required to be part of the chip enable/select decoding (chip enable is similar to chip select except chip enable also provides whether the chip is in active or standby power mode). The 8086 address lines must be connected to the ROM/EPROM chips starting with A1 and higher to all the address lines of the ROM/EPROM chips. The 8086 unused address lines can be used as chip enable/select decoding. To interface the ROMs/RAMs directly to the 8086 multiplexed bus, they must have output enable signals. Figure 3.17 shows the 8086 interfaced to two 2716s.

Byte accesses are obtained by reading the full 16-bit word onto the bus with the 8086 discarding the unwanted byte and accepting the desired byte. If RD, WR, and M/IO are not decoded to generate separate memory and I/O commands for memory and I/O chips and the I/O space overlaps with the memory space of ROM/EPROM, then M/IO must be a condition of chip select decode.

3.8.3.2 Static RAMs

Since static RAMs are read/write memories, both A0 and BEH must
be included in the chip select/chip enable decoding of the devices and 
write timing must be considered in the compatibility analysis.

For each static RAM, the memory data lines must be connected to 
either the upper half AD15-AD0 or lower half AD7-AD0 of the 8086 data 
lines.

For static RAMs without output enable pins, read and write lines must 
be used as enables for chip select generation to avoid bus contention. If 
read and write lines are not used to activate the chip selects, static RAMs 
with common input/output data pins such as 2114 will face extreme bus 
contentions between chip selects and write active. The 8086 A0 and BHE 
pins must be used to enable the chip selects. A possible way of generating 
chip selects for high and low static RAM banks is given in Figure 3.18. 
Note that Intel #2083 has three enables E1, E2, and E3, three inputs A10- 
A2, and eight outputs O0-O7.

For devices with output enable such as 2142, one way to generate chip 
selects for the static RAMs is by gating the 8086 WR signal with BHE and 
A0 to provide upper and lower bank write strobes. A possible configuration 
is shown in Figure 3.19. Since the Intel 2142 is a 1624 x 4 bit static RAM, 
two chips for each bank with a total of 4 chips for 2K x 8 static RAM is 
required. Note that DATA is read from the 2142 when the output disable 
OD is low, WE is HIGH, and DATA is written into 2142. When the 
output disable OD is HIGH, WE is LOW, CS2 is HIGH, and CS1 is 
LOW. If multiple chip selects are available with the static RAM, BHE and 
A0 may be used directly as the chip selects. A possible configuration for 2K 
× 8 array is shown in Figure 3.20.

3.8.3 Dynamic RAM

Dynamic RAMs store information as charges in capacitors. Since ca-
capacitors can hold charges for a few milliseconds, refresh circuitry is 
necessary in dynamic RAMs for retaining these charges. Therefore, dynamic 
RAMs are complex devices to design a system. To relieve the designer of 
most of these complicated interfacing tasks, Intel provides the 8202 dy-
namic RAM controller as part of the 8086 family of peripheral devices. The 
8202 can be interfaced with the 8086 to build a dynamic memory system. 
A thorough discussion on this topic can be found in Intel manuals.

3.8.4 8086 PROGRAMMED I/O

The 8086 can be interfaced to 8- and 16-bit I/O devices using either 
standard or memory mapped I/O. The standard I/O uses the instructions 
IN and OUT, and is capable of providing 64K bytes of I/O ports. Using 
standard I/O, the 8086 can transfer 8- or 16-bit data to or from a 
peripheral device. The 64K byte I/O locations can then be configured as 
64K 8-bit ports or 32K 16-bit ports. All I/O transfer between the 8086 
and the peripheral devices take place via AL, for 8-bit ports (AH is not 
involved) and AX for 16-bit ports. The I/O port addressing can be done 
either directly or indirectly as follows:

DIRECT

- IN AL, PORTA or IN AX, PORTB inputs 8-bit 
  contents of port A into AL or 16-bit contents of port
B into AX, respectively. Port A and port B are assumed as 8- and 16-bit ports, respectively.

- OUT PORTA, AL or OUT PORT B, AX outputs 8-bit contents of AL into port A or 16-bit contents of AX into port B, respectively.

- INDIRECT IN AX, DX or IN AL, DX inputs 16-bit data addressed by DX into AX or 8-bit data addressed by DX into AL, respectively.
- OUT DX, AX or OUT DX, AL outputs 16-bit contents of AX into the port addressed by DX or 8-bit contents of AL into the port addressed by DX, respectively. In indirect addressing, register DX is used to hold the port address.
3.8.4.a Eight-Bit I/O Ports

Devices with 8-bit I/O ports can be connected to either the upper or lower half of the data bus. Bus loading is distributed by connecting an equal number of devices to the upper and lower halves of the data bus. If the I/O port chip is connected to the 8086 lower half of the data lines (AD9-AD7), the port addresses will be even (AO = 0). On the other hand, the port addresses will be odd (AO = 1) if the I/O port chip is connected to the upper half of the 8086 data lines (AD8-AD15). AO will always be one or zero for a partitioned I/O chip. Therefore, AO cannot be used as an address input to select registers within a particular I/O chip. If two chips are connected to the lower and upper halves of the 8086 address bus that differ only in AO (consecutive odd and even addresses), AO and BHE must be used as conditions of chip select decode to avoid a write to one I/O chip from erroneously performing a write to the other. Figure 3.21 shows two ways of generating chip selects for I/O port chips.

The method shown in Figure 3.21A uses separate 8265s to generate chip selects for odd- and even-addressed byte peripherals. If a 16-bit word transfer is performed to an even-addressed I/O chip, the adjacent odd-addressed I/O chip is also selected. Figure 3.21B generates chip selects for byte transfers only.

3.8.4.b Sixteen-Bit I/O Ports

For efficient bus utilization and simplicity of I/O chip selection sixteen-bit I/O ports should be assigned even addresses. Both AO and BHE should be the chip select conditions to ensure that the I/O chip is selected only for word operations. Figure 3.22 shows a method for generating chip for 16-bit ports. Note that in Figure 3.22, the 8086 will output 1 on both AO and BHE when it executes a 16-bit I/O instruction with an even port address such as IN AX, 0006H. This instruction inputs the 16 bit contents of ports 0006H and 0007H in AX.

3.8.5 8086-BASED MICROCOMPUTER

In this section, the 8086 will be interfaced in the minimum mode to provide 2K x 16 EPROM, 1K x 16 static RAM, and six 8-bit I/O ports. 2716 EPROM, 2142 static RAM, and 8255 I/O chips will be used for this purpose. Memory and I/O maps will also be determined. Figure 3.23 shows a hardware schematic for accomplishing this.
Three 8282 octal latches are used. These are numbered as 8282-1, 8282-2, and 8282-3. The STB input of the 8282 strobes the eight inputs at the D16-7 pins into the internal data latches. When OE is low, 8-bit data at the data latches are transferred to the output pins DOO-7. The 8086 ALE pin is used as the STB input for all the three latches and OE is tied to ground. The 8282-1 latches A16-A9 and BHE. Pins D15-7 and D00-7 pins are not used for the 8282-1 chip. The 8282-2 and 8282-3 chips provide the A15-A8 and A7-A0, respectively. The 2716 is a 2K x 8 ultraviolet EPROM with eleven address pins A0-A10 and eight data pins O0-7. Two 2716s numbered as 2716-1 and 2716-2 are used. The 2716-1 provides all the even-addressed data and the 2716-2 contains all the odd-addressed data. The 8086 A1-A11 pins are connected to the A0-A19 pins of these chips. The 2716-1 even EPROM’s O0-7 pins are connected to the 8086 D0-D7 pins. This is because the 8086 reads data via the D0-D7 pins for even addresses. On the other hand, the O0-7 pins of the 2716-2 are connected to the 8086 D8-D15 pins. The 8086 reads data via D8-D15 pins for odd addresses.

The 8205 is a 3 to 8 decoder with three enable pins E1, E3, and E3. E3 is tied to high. E1 and E3 are enabled by the 8086 A15 and A16 pins. The 8205 input pins A0-A2 are connected to the 8086 A12-A14. The 8205 is used to provide the fully decoding technique for addressing 2716s, 2142s, and 255s. The 8205 3 x 8 decoder is used for future expansion.

The memory map for the 2716s can be determined as follows. A19-A17 pins are don’t cares and are assumed to be high.

For 2716-1 (EVEN # EPROM)
A19 A18 A17 A16 A15 A14 A13 A12 A11 A10 A9 ... A1 A0
1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
F F 1 1 1 1 1 1

Figure 3.22 Chip selects for 16-bit parts.
For I/O ports, chips 8255-1 and 8255-2 are used. The 8255 is a general-purpose programmable I/O chip. The 8255 has three 8-bit I/O ports: ports A, B, and C. Ports A and B are latched 8-bit ports for both input and output. Port C is also an 8-bit port with latched output but the inputs are not latched. Port C can be used in two ways. It can either be used as a simple I/O port or as a control port for data transfer using handshaking via ports A and B.

The 8086 can configure the three ports by outputting appropriate data to the 8-bit control register. The ports can be decoded by two 8255 input pins A0 and A1 as follows:

<table>
<thead>
<tr>
<th>A1</th>
<th>A0</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Port A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Port B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Port C</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Control register</td>
</tr>
</tbody>
</table>

The structure of the control register is given below:

![Control Register Diagram]

The bit 7 (D7) of the control register must be one to send the above definitions for bits 0 through 6 (D0-D6).

In this format bits D0-D6 are divided into two groups: groups A and B. Group A configures all 8 bits of port B and upper 4 bits of port C, while
Group B defines all 8 bits of port A and lower 4 bits of Port C. All bits in a port can be configured as a parallel input port by writing a 1 at the appropriate bit in the control register by the 8086 OUT instructions, and a 0 in a particular bit position will configure the appropriate port as a parallel output port. Group A has three modes of operation. These are modes 0, 1, and 2. Group B has two modes: modes 0 and 1. Mode 0 for both groups A and B provides simple I/O operation for each of the three ports. No handshaking is required. Mode 1 for both groups A and B is the strobed I/O mode used for transferring I/O data to or from a specified port in conjunction with strobes or handshaking signals. Ports A and B use the lines on Port C to generate or accept these handshaking signals.

The mode 2 of Group A is the strobed bidirectional bus I/O and may be used for communicating with a peripheral device on a single 8-bit data bus for both transmitting and receiving data (bidirectional bus I/O). Handshaking signals are required. Interrupt generation and enable/disable functions are also available.

When $D7 = 0$, the bit set/reset control word format is used for the control register as follows:

```

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

```

This format is used to set or reset the output on a pin of port C or when enabling of the interrupt output signals for handshake data transfer is desired. For example, the 8-bits $0 \times \times X \times \times 0$ will clear bit 6 of port C to zero. Note that the control word format can be output to the 8255 control register by using the 8086 OUT instruction.

Next, the mode 0 of the 8255 will only be considered in the following discussion to illustrate 8086's programmed I/O capability.

Now, let us define the control word format for mode 0 more precisely.
Chapter 3

Example 3.8
Assume an 8086/8255 configuration with the following port addresses:

Port A = 0F9H
Port B = 0FBH
Control Register = FFH

Port A has three switches connected to bits 0, 1, and 2 and port B has an LED connected to bit 2 as follows:

Write an 8086 assembly language program to turn the LED ON if port A has an odd number of HIGH switch inputs; otherwise turn the LED OFF. Do not use any instructions involving the parity flag.

Solution

```assembly
PORT A EQU 0F9H
PORT B EQU 0FBH
CNTRL EQU 0FFH
MOV AL, 90H ; Configure Port A as input
OUT CNTRL, AL ; Port B as output
BEGIN IN AL, PORTA ; Input switches
AND AL, 07H ; Mask high five bits
CMP AL, 07H ; Are all three inputs HIGH?
JZ EOD ; If so, turn LED ON
CMP AL, 01H ; Is input 0 HIGH?
JZ EOD ; If so, turn LED ON
```

3.9 8086 Interrupt System

The 8086 interrupts can be classified into three types. These are
1. Predefined interrupts
2. User-defined software interrupts
3. User-defined hardware interrupts

The interrupt vector addresses for all the 8086 interrupts are determined from a table stored in locations 00000H through 003FFFFH. The starting addresses for the service routines for the interrupts are obtained by the 8086 using this table. Four bytes of the table are assigned to each interrupt: two bytes for IP and two bytes for CS. The table may contain up to 256 8-bit vectors. If fewer than 256 interrupts are defined in the system, the user need only provide enough memory for the interrupt pointer table for obtaining the defined interrupts.

The interrupt address vector (contents of IP and CS) for all the interrupts of the 8086 assigns every interrupt a type code for identifying the interrupt. There are 256 type codes associated with the 256 table entries. Each entry consists of two addresses, one for storing the IP contents and the other for storing the CS contents. Each 8086 interrupt physical address vector is 20 bits wide and is computed from the 16-bit contents of IP and CS.

For obtaining an interrupt address vector, the 8086 calculates two addresses in the pointer table where IP and CS are stored for a particular interrupt type.

For example, for the interrupt type nn (instruction INT nn), the table

```assembly
CMP AL, 02H ; Is input 1 HIGH?
JZ EOD ; If so, turn LED ON
CMP AL, 04H ; Is input 2 HIGH?
JZ EOD ; If so, turn LED ON
MOV AL, 00H ; Else turn LED
OUT PORTS, AL ; OFF
JMP BEGIN ; REPEAT
```

EOD MOV AL, 04H ; Turn LED
OUT PORTS, AL ; ON
JMP BEGIN ; REPEAT
address for IP = 4 * n + and the table address for CS = 4 * n + 2. For servicing the 8086's nonmaskable interrupt (NMI pin), the 8086 assigns the type code 2 to this interrupt. The 8086 automatically executes the INT2 instruction internally to obtain the interrupt address vector as follows:

Address for IP = 4 * 2 = 00000H
Address for CS = 4 * 2 + 2 = 0000AH

The 8086 loads the values of IP and CS from the 20-bit physical address 00000H and 0000AH in the pointer table. The user must store the desired 16-bit values of IP and CS in these locations. Similarly, the IP and CS values for other interrupts are calculated. The 8086 interrupt pointer table layout is shown in Table 3.2.

<table>
<thead>
<tr>
<th>Interrupt type code</th>
<th>16-bit Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000H</td>
</tr>
<tr>
<td>1</td>
<td>00004H</td>
</tr>
<tr>
<td>2</td>
<td>00008H</td>
</tr>
<tr>
<td>255</td>
<td>003FEH</td>
</tr>
</tbody>
</table>

In response to an interrupt, the 8086 pushes flags, CS, and IP onto the stack, clears TF and IF flags, and then loads IP and CS from the pointer table using the type code.

Interrupt service routine should be terminated with the IRET (Interrupt Return) instruction which pops the top three stack words into IP, CS, and flags, thus returning to the right place in the main program. The 256 interrupt type codes are assigned as follows:

- Types 0 to 4 are for the predefined interrupts.
- Types 5 to 31 are reserved by Intel for future use.
- Types 32 to 255 are available for maskable interrupts.

### 3.9.1 PREDEFINED INTERRUPTS (0 TO 4)

The predefined interrupts include DIVISION BY ZERO (type 0), SINGLE STEP (type 1) NONMASKABLE INTERRUPT pin (type 2), BREAKPOINT INTERRUPT (type 3), and INTERRUPT ON OVERFLOW (type 4). The user must provide the desired IP and CS values in the interrupt pointer table. The user may also imitate these interrupts through hardware or software. If a predefined interrupt is not used in a system, the user may assign some other function to the associated type.

The 8086 is automatically interrupted whenever a division by zero is attempted. This interrupt is nonmaskable and is implemented by Intel as part of the execution of the divide instruction. When the TF (TRAP flag) is set by an instruction, the 8086 goes into the single step mode. The TF can be cleared to zero as follows:

```assembly
PUSHF ; Save flags
MOV BP, SP ; Move [SP] → [BP]
AND [BP + 0], 0FFFF ; Clear TF
POPF ; Pop flags
```

Note that in the above [BP + 0] rather than [BP] is used since BP cannot be used without displacement. Now, to set TF, the AND instruction in the above should be replaced by OR [BP + 0], 0000H.

Once TF is set to one, the 8086 automatically generates a TYPE 1 interrupt after execution of each instruction. The user can write a service routine at the interrupt address vector to display memory locations and/or register to debug a program. Single step is nonmaskable and cannot be enabled by STI (enable interrupt) or CLI (disable interrupt) instruction. The nonmaskable interrupt is initiated via the 8086 NMI pin.

It is edge triggered (LOW to HIGH) and must be active for two clock cycles to guarantee recognition. It is normally used for catastrophic failures such as power failure. The 8086 obtains the interrupt vector address by automatically executing the INT2 (type 2) instruction internally.
Type 3 interrupt is used for breakpoint and is nonmaskable. The user inserts the one-byte instruction INT3 into a program by replacing an instruction. Breakpoints are useful for program debugging.

The INTERRUPT ON OVERFLOW is a type 4 interrupt. This interrupt occurs if the overflow flag (IF) is set and the INTO instruction is executed. The overflow flag is affected, for example, after execution of signed arithmetic such as MULS (signed multiplication) instruction. The user can execute the INTO instruction after the MULS. If there is an overflow, an error service routine written by the user at the type 4 interrupt address vector is executed.

3.9.2 USER-DEFINED SOFTWARE INTERRUPTS

The user can generate an interrupt by executing a two-byte interrupt instruction INT n. The INT n instruction is not maskable by the interrupt enable flag (IF). The INT n instruction can be used to test an interrupt service routine for external interrupts. Type codes 0 to 255 can be used. If predefined interrupt is not used in a system, the associated-type code can be utilized with the INT n instruction to generate software (internal) interrupts.

3.9.3 USER-DEFINED HARDWARE (MASKABLE INTERRUPTS)

The 8086 maskable interrupts are initiated via the INTR pin. These interrupts can be enabled or disabled by STI (IF = 1) or CLI (IF = 0), respectively. IF = 1 and INTR is active (HIGH) without occurrence of any other interrupts, the 8086, after completing the current instruction, generates INTA LOW twice, each time for about 2 cycles.

The state of the INTR pin is sampled during the last clock cycle of each instruction. In some instances, the 8086 samples the INTR pin at a later time. An example is execution of POP to a segment register. In this case, the interrupts are sampled until completion of the following instruction. This allows a 32-bit pointer to be loaded to SS and SP without the danger of an interrupt occurring between the two loads.

INTA is only generated by the 8086 in response to INTR, as shown in Figure 3.24. The interrupt acknowledge sequence includes two INTA cycles separated by two idle clock cycles. ALE is also generated by the 8086 and will load the address latches with indeterminate information. The ALE is useful in maximum systems with multiple 8259A priority interrupt controllers. During the INTA bus cycles, DTR and DEN are LOW (see 8086 minimum mode bus cycle). The first INTA bus cycle indicates that an interrupt acknowledge cycle is in progress and allows the system to be ready to place the interrupt type code on the next INTA bus cycle. The 8086 does not obtain the information from the bus during the first cycle. The system must place the type code on the lower half of the 16-bit data bus during the second cycle.

In the minimum mode, the M/IO line is low indicating I/O operation during the INTA bus cycles. The 8086 internal LOCK signal is also low from T2 of the first bus cycle until T2 of the second bus cycle to avoid the BIU from accepting a hold request between the two INTA cycles. Figure 3.25 shows an simplified interconnection between the 8086 and 74LS244 for servicing the INTR. INTA enables 74LS244 to place the type code on the 8086 data bus.

In the maximum mode, the status lines S0 to S2 will enable the INTR.
output for each cycle via the 8288. The 8086 LOCK output will be active from T2 of the first cycle until T2 of the second to prevent the 8086 from accepting a hold request on either RD/WT input and to prevent bus arbitration logic from releasing the bus between INTAs in multibus systems. The LOCK output can be used in external logic to lock other devices off the system bus, thus ensuring the INTA sequence to be completed without intervention.

Once the 8086 has the interrupt-type code (via the bus for hardware interrupts, from software interrupt instructions INT n or from the predefined interrupts), the type code is multiplied by four to obtain the corresponding interrupt vector in the interrupt vector table. The four bytes of the interrupt vector are least significant byte of the instruction pointer, most significant byte of the pointer, least significant byte of the code segment register, and most significant byte of the code segment register. During the transfer of control, the 8086 pushes the flags and current code segment and instruction pointer into the stack. The new CS and IP values are loaded, flags TF and PF are then cleared to zero. The CS and IP values are read by the 8086 from the interrupt vector table. No segment registers are used when accessing the interrupt pointer table. S453 has the value 10, to indicate no segment register selection.

As far as the 8086 interrupt priorities are concerned, single step interrupt has the highest priority, followed by NMI, followed by the software interrupts (all interrupts except single step, NMI, and INTR interrupts). This means that a simultaneous NMI and single step will cause the NMI service routine to follow simple step, a simultaneous software interrupt and single step will cause the software interrupt service routine to follow single step, and a simultaneous NMI and software interrupt will cause the NMI service routine to be executed prior to the software interrupt service routine. An exception to this priority scheme occurs if all three nonmaskable interrupts (single step, software, and NMI) are pending. For this case, software interrupt service routine will be executed first followed by NMI, and single step will be skipped. However, if NM service routine is in progress, and single stepping is not enabled. However, if NM service routine is in progress, and single stepping is not enabled, single stepping resumes upon execution of the instruction causing the software interrupt (the next instruction in the routine being single stepped).

The INTR is maskable and has the lowest priority. If the user does not wish to single step before INTR is serviced, the single-step routine must disable interrupts during execution of the program being single stepped, and reenable interrupts on return to the single-step routine. To avoid single stepping before the NMI service routine, the single-step routine must check the return address on the stack for the NMI service routine address and return control to that routine without single step enabled. Figure 3.26A and B illustrate this. In Figure 3.26A single step and NMI occur at the same time, whereas in Figure 3.26B NMI, INTR, and a divide error occur during a divide instruction being single stepped.

A priority interrupt controller such as the 8259A can be used with the 8086 INTR to provide eight levels of interrupts. The 8259A has built-in features for expansion of up to 64 levels with additional 8259As. The
3.10 8086 DMA

When configured in the minimum mode (MN/ MX pin HIGH), the 8086 provides HOLD (DMA request) and HLDA (DMA acknowledge) signals to take over the system bus for DMA applications. The Intel DMA controller chips 8237 and 8237D can be used with the 8086. The 8237 or 8237D can request DMA transfer between the 8086 memory and I/O device by activating the 8086 HOLD pin. The 8086 will complete the current bus cycle (if there is one presently in progress) and then output HLDA, relinquishing the system bus to the DMA controller. The 8086 will not try to use the HOLD pin until it is negated.

As mentioned before, the 8086 memory addresses are organized in two separate banks — one containing even-addressed bytes and the other containing odd-addressed bytes. An 8-bit DMA controller must alternate these two banks to access logically adjacent bytes in memory. The 8089 I/O processor can interface a high-speed 8-bit device to an 8086-based microcomputer via DMA. The 8089 I/O processor will be discussed in the next section.

3.11 8089 I/O Processor

An input-output processor (IOP) is a microprocessor system with DMA capabilities that can directly communicate with I/O devices. A microcomputer with IOP configuration typically consists of a memory unit and one or more IOPs. Each IOP performs all I/O functions and, therefore, off-loads the microprocessor of these functions. The Intel 8089 is an IOP and performs the functions of an intelligent DMA controller. The 8089 is primarily designed for the Intel 8086.

The 8089 contains two I/O channels. Each channel in the 8089 provides the basic features of a microprocessor along with those of a DMA controller.

The 8089 can be used in managing files and buffers for peripheral devices such as hard disk and floppy disk controllers. Other typical applications with the 8089 include CRT control functions, such as cursor control and autoscrolling keyboard control, and general I/O functions.

The 8086 initiates an I/O operation by forming a message in the main memory that specifies the function to be performed by the 8089. The 8089 reads this message from memory, performs the operation, and informs the 8086 upon completion of the task. All I/O device controller overheads such as DMA data block transfer functions are accomplished by the 8089 without any 8086 intervention.

The 8086 communicates with the 8089 in two distinct modes: initialization and command. The initialization sequence is usually performed when the system is powered up or reset. The 8086 initializes the 8089 by forming a series of message blocks in memory. Upon receipt of a command from the 8086, the 8089 reads these blocks and determines how the data buses are configured and how access to the buses is to be controlled.

Upon initialization, the 8086 directs all communications to either of the 8089's two channels. Each channel has the ability to execute programs and has its own set of registers and associated control logic that monitors its operation during data transfer. All 8086 to 8089 communication is based on the channel control block (CCB). The CCB is located in the 8089's memory space and its address is passed to the 8089 during initialization. The CCB contains information such as the busy flag and the Channel Command Word (CCW). The busy flag indicates whether the channel is in the midst of an operation or is available for a new 8086 command. Each channel maintains this flag. The 8086 sets the CCW to indicate the type of operation the 8089 is to perform. Six different commands allow the 8089 to start and stop programs, receive interrupt requests, etc.

If the 8086 is commanding a channel to run a program, it directs the channel to a parameter block (PB) and a task block (TB). The PB contains variable data that the channel program uses in carrying out its assignment. The PB may also contain space for the result that the channel is to return to the 8086. A TB is a channel program sequence of 8089 instructions that will perform the desired operation. A typical channel program might use the PB data to set up the 8089, and an I/O device controller for a transfer, to carry out the transfer, return the results, and then halt.

After initializing the CCW and linking the TB to a PB and the PB to the CCB, the 8086 normally activates the CA (channel attention) and SEL (channel select) pins of the 8089. The SEL pin at the falling edge of CA line selects channel 1 or channel 2.

An 8089 channel attention line (CA) is similar in concept to an 8086 interrupt. When the channel identifies the CA, it stops the current program.
and checks the command in the CCW. If the 8086 commands the 8089 to start a program, the 8089 reads the PB and TB into its internal registers, sets its busy flag, and starts executing the channel program. After sending the CA, the 8086 may perform other functions and the 8089 may carry on with the desired function in parallel.

Upon completion of the function, the 8089 informs the 8086 by clearing the busy flag in the CCW. Optionally, the 8089 may interrupt to inform the 8086 of its completion of the desired function.

The 8086/8089 communication structure is shown in Figure 3.27.

![Figure 3.27 8086/8089 communication.](image)

Almost all communications occur via messages in shared memory. The only direct hardware communications between the 8086 and the 8089 are channel attentions and interrupt requests.

The two 8089 channels can operate independently of each other. Each channel contains its own registers, channel attention, interrupt request, and DMA control signals.

A channel may be idle, may execute a program, carry out a DMA transfer, or respond to the 8086 channel attention at any particular time. The two channels cannot perform data transfer simultaneously. Only one of the two channels can be active at a time. However, in some cases, the two channels can be active concurrently operating in sequence. For example, during the successive DMA cycles of channel 1, instructions may be executed by channel 2. The 8089 assembly language programming (ASM-89) is used to write channel programs. The 8089 includes 50 instructions. These include data transfer, simple arithmetic, logical, and address manipulation operations, unconditional and conditional jumps, calls, bit manipulations, and DMA transfer instructions.

The 8089 DMA transfer instruction makes the channel ready for a DMA transfer. The 8089 completes execution of one additional instruction, then stops program execution and goes into DMA transfer mode. Before execution of the DMA transfer instruction, the user must initialize the channel register which monitors the DMA transfer.

### QUESTIONS AND PROBLEMS

3.1 What is the basic difference between the 8086 and 8088 microprocessors? Name one reason why these two microprocessors are included in the 8086 family by Intel.

3.2 List the 8086 minimum and maximum mode signals. How are these modes selected?

3.3 What are the functions provided by 8288 bus controller and 8289 bus arbiter in a maximum mode 8086 system?

3.4 Which bit of the 8086 FLAG register is used by the string instructions? How? Illustrate this by using the 8086 MOVSB instruction.

3.5 What is the relationship between the 8086 external and internal clocks? Does the 8086 have an on-chip clock circuitry? Comment.

3.6 Write an 8086 instruction sequence to set the TF bit in the FLAG register for single stepping.

3.7 If [BL] = 30H (ASCII code for 6) and [CL] = 33H (ASCII code for 3), write an 8086 assembly program which will add the contents of BL and CL, and then provide the result in decimal. Store result in CL.

3.8 What happens to the contents of the AH register after execution of the following 8086 instruction sequence:

```
MOV AX, 0F180H
CWE
CWD
```

3.9 Write an 8086 assembly program to implement the following Pascal segment:
Questions and Problems

3.15 Assume the memory and I/O maps of Figure 3.23. Interface the following A/D to the 8086/2716/2142/8255 of the figure:

Write an 8086 assembly program to input the A/D converter and turn an LED ON connected to bit 5 of port A of 8255. If the number read from A/D is odd, otherwise turn the LED OFF. Assume that the LED is turned ON by a HIGH and turned OFF by a LOW.

3.16 Repeat problem 3.15 using 8086 INTR interrupt.

3.17 Identify the 8086 and 8089 signals required to interface these two chips and discuss them.

3.18 Write 8086 assembly language program to add a 16-bit number stored in DX (bits 0 to 7 containing the high-order byte of the number and bits 8 to 15 containing the low-order byte) with another 16-bit number stored in BX (bits 0 to 7 containing the low-order 8 bits of the number and bits 8 through 15 containing the high-order 8 bits). Store the result in CX.
### TABLE 3.4-1 (continued)
#### 8086 Data Transfer Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Illustration</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>MOV mem, reg</td>
<td>[mem] ← [reg]</td>
<td>Immediate</td>
<td>Memory</td>
<td>MOV [BX], CX</td>
<td>mem uses DS as the segment register; no memory-to-memory operation allowed, that is, MOV mem, mem is not permitted Segment register cannot be specified as reg1 or reg2; no flags are affected; not usually used to load or store 'A' from or to memory mem uses DS as the segment register; 8- or 16-bit data specify whether memory location is 8- or 16-bit; no flags are affected Segment register cannot be specified as reg; data can be 8- or 16-bit; no flags are affected Takes fewer bytes than reg, mem; mem uses DS as segment register; 'A' can be AL or AX; no flags are affected mem uses DS as segment register; 'A' can be AL or AX; no flags are affected; needs fewer bytes than MOV mem, reg mem uses DS as segment register; used for initializing CS, DS, ES, and SS; no flags are affected mem uses DS as segment register; no flags are affected</td>
</tr>
<tr>
<td>MOV mem, A</td>
<td>MOV mem, A</td>
<td>[mem] ← [A]</td>
<td>Memory</td>
<td>Memory</td>
<td>MOV [BX], CX</td>
<td>mem uses DS as the segment register; no memory-to-memory operation allowed, that is, MOV mem, mem is not permitted Segment register cannot be specified as reg1 or reg2; no flags are affected; not usually used to load or store 'A' from or to memory mem uses DS as the segment register; 8- or 16-bit data specify whether memory location is 8- or 16-bit; no flags are affected Segment register cannot be specified as reg; data can be 8- or 16-bit; no flags are affected Takes fewer bytes than reg, mem; mem uses DS as segment register; 'A' can be AL or AX; no flags are affected mem uses DS as segment register; 'A' can be AL or AX; no flags are affected; needs fewer bytes than MOV mem, reg mem uses DS as segment register; used for initializing CS, DS, ES, and SS; no flags are affected mem uses DS as segment register; no flags are affected</td>
</tr>
<tr>
<td>MOV segreg, mem</td>
<td>MOV segreg, mem</td>
<td>[seg] ← [mem]</td>
<td>Memory or register</td>
<td>Memory or register</td>
<td>MOV [BX], CX</td>
<td>mem uses DS as the segment register; no memory-to-memory operation allowed, that is, MOV mem, mem is not permitted Segment register cannot be specified as reg1 or reg2; no flags are affected; not usually used to load or store 'A' from or to memory mem uses DS as the segment register; 8- or 16-bit data specify whether memory location is 8- or 16-bit; no flags are affected Segment register cannot be specified as reg; data can be 8- or 16-bit; no flags are affected Takes fewer bytes than reg, mem; mem uses DS as segment register; 'A' can be AL or AX; no flags are affected mem uses DS as segment register; 'A' can be AL or AX; no flags are affected; needs fewer bytes than MOV mem, reg mem uses DS as segment register; used for initializing CS, DS, ES, and SS; no flags are affected mem uses DS as segment register; no flags are affected</td>
</tr>
</tbody>
</table>
### Table 3.A.1 (continued)
8086 Data Transfer Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Illustration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translate table</td>
<td>XLAT</td>
<td>[AX] ← [AX] + [BX]</td>
<td>Source</td>
<td>Destination</td>
<td></td>
</tr>
</tbody>
</table>

This instruction is used for translating characters from one code such as ASCII to another such as EBCDIC; this is a no-operand instruction and is called an instruction with implied addressing mode; the instruction loads AL with the contents of a 20-bit physical address computed from DS, BX, and AX; this instruction can be used to read the elements in a table where BX can be loaded with a 16-bit value to point to the starting address (offset from DS) and AX can be loaded with the element number (0 being the first element number); no flags are affected; the XLAT instruction is equivalent to MOV AL, [AX] [BX]

### Table 3.A.2
8086 I/O Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Illustration</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td>IN AL, DX</td>
<td>[AL] ← [PORT DX]</td>
<td>Register (port)</td>
<td>Register (AL only)</td>
<td>IN AL, DX</td>
</tr>
<tr>
<td></td>
<td>IN AX, DX</td>
<td>[AX] ← [PORT DX]</td>
<td>Register (port)</td>
<td>Register (AX only)</td>
<td>IN AX, DX</td>
</tr>
<tr>
<td></td>
<td>IN AL, PORT</td>
<td>[AL] ← [PORT]</td>
<td>Register (port)</td>
<td>Register (AL only)</td>
<td>IN AL, PORT</td>
</tr>
<tr>
<td></td>
<td>IN AX, PORT</td>
<td>[AX] ← [PORT]</td>
<td>Register (port)</td>
<td>Register (AX only)</td>
<td>IN AX, PORT</td>
</tr>
<tr>
<td></td>
<td>OUT DX, AL</td>
<td>[PORT] ← [AL]</td>
<td>Register (port)</td>
<td>Register (AL only)</td>
<td>OUT DX, AL</td>
</tr>
<tr>
<td></td>
<td>OUT DX, AX</td>
<td>[PORT DX] ← [AX]</td>
<td>Register (port)</td>
<td>Register (AX only)</td>
<td>OUT DX, AX</td>
</tr>
<tr>
<td></td>
<td>OUT PORT, AL</td>
<td>[PORT] ← [AL]</td>
<td>Register (port)</td>
<td>Register (AL only)</td>
<td>OUT PORT, AL</td>
</tr>
<tr>
<td></td>
<td>OUT PORT, AX</td>
<td>[PORT] ← [AX]</td>
<td>Register (port)</td>
<td>Register (AX only)</td>
<td>OUT PORT, AX</td>
</tr>
</tbody>
</table>
### TABLE 3.A-3

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Illustration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load EA</td>
<td>LEA reg., offset</td>
<td>[reg] ← offset portion of address in DS</td>
<td>Source: Memory, Destination: Register</td>
<td>LEA SI, ADOR</td>
<td>LEA (load effective address) loads the value in the source operand rather than its content to register (such as SI, DI, BX), which are allowed to contain offset for accessing memory; no flags are affected</td>
</tr>
<tr>
<td>Load pointer using DS</td>
<td>LDS reg., mem.</td>
<td>[reg] ← [mem] [ES] ← [mem + 2]</td>
<td>Memory: Register</td>
<td>LDS BX, BEGIN</td>
<td>Load a 16-bit register (such as BX, BP, SI, DI) with the content of specified memory and load DS with the content of the location that follows; no flags are affected; DS is used as the segment register for mem; in the example, SI is loaded with 16-bit value from memory location addressed by 20-bit physical address computed from DS and BX. The 16-bit content of the next memory is loaded into ES; no flags are affected</td>
</tr>
<tr>
<td>Load pointer using ES</td>
<td>LES reg., mem.</td>
<td>[reg] ← [mem] [ES] ← [mem + 2]</td>
<td>Memory: Register</td>
<td>LES SI, (BX)</td>
<td></td>
</tr>
</tbody>
</table>

* Address object transfers — provides programmer with some control over the addressing mechanism.

### TABLE 3.A-4

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Illustration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load AH from flags</td>
<td>LAHF</td>
<td>[AH] ← (Flags low-byte)</td>
<td>Source: —, Destination: —</td>
<td>LAHF</td>
<td>This instruction has implied addressing mode; it loads AH with low byte of the flag register; no flags are affected</td>
</tr>
<tr>
<td>Store AH in flags</td>
<td>SAHF</td>
<td>(Flags low-byte) ← [AH]</td>
<td>Source: —, Destination: —</td>
<td>SAHF</td>
<td>This instruction has the implied addressing mode; the content of the AH register is stored into low-byte of the flag register; all flags are affected (low byte of SR)</td>
</tr>
<tr>
<td>Push flags into stack</td>
<td>PUSHF</td>
<td>[SP] ← [SP] − 2</td>
<td>Source: —, Destination: —</td>
<td>PUSHF</td>
<td>This instruction pushes the 16-bit flag register onto the stack; no flags are affected</td>
</tr>
<tr>
<td>Pop flags off stack</td>
<td>POPF</td>
<td>[Flags] ← [SP] [SP] ← [SP] + 2</td>
<td>Source: —, Destination: —</td>
<td>POPF</td>
<td>This instruction pops the top two stack bytes into the 16-bit flag register; all flags are affected (low byte of SR)</td>
</tr>
</tbody>
</table>
### TABLE 3.A-5

**Arithmetic Instructions**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Source</th>
<th>Destination</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADD</strong></td>
<td>ADD mem/reg 1, mem/reg 2</td>
<td>[mem/reg 1] ← [mem/reg 2] + [mem/reg 1]</td>
<td>Memory or memory</td>
<td>Memory or memory</td>
<td>ADD BL, 1FH</td>
<td>Add two 8- or 16-bit data; no memory-to-memory ADD is permitted; all flags are affected; mem uses DS as the segment register; reg 1 or reg 2 cannot be segment register; data can be 8- or 16-bit; all flags are affected.</td>
</tr>
<tr>
<td><strong>ADD mem, data</strong></td>
<td>[mem] ← [mem] + data</td>
<td>Immediate Memory</td>
<td>Immediate Memory</td>
<td>ADD START, 02H</td>
<td>Data can be 8- or 16-bit; no segment registers are allowed; all flags are affected; this instruction should not be used when an AX or AL with 8- or 16-bit immediate data.</td>
<td></td>
</tr>
<tr>
<td><strong>ADD reg, data</strong></td>
<td>[reg] ← [reg] + data</td>
<td>Immediate Register</td>
<td>Immediate Register</td>
<td>ADD BX, 0354H</td>
<td>Data can be 8- or 16-bit; 'A' can be AL or AX; all flags are affected.</td>
<td></td>
</tr>
<tr>
<td><strong>ADD A, data</strong></td>
<td>[A] ← [A] + data</td>
<td>Immediate Register</td>
<td>Immediate Register</td>
<td>ADD AH, 05H</td>
<td>Memory or register can be 8- or 16-bit; all flags are affected; no segment registers are allowed; no memory-to-memory ADC is permitted.</td>
<td></td>
</tr>
<tr>
<td><strong>ADD with carry</strong></td>
<td>ADC mem/reg 1, mem/reg 2</td>
<td>[mem/reg 1] ← [mem/reg 2] + [mem/reg 1] + CY</td>
<td>Memory or memory</td>
<td>Memory or memory</td>
<td>ADC [SI], BL</td>
<td>Data can be 8- or 16-bit; mem uses DS as the segment register; all flags are affected.</td>
</tr>
</tbody>
</table>

### TABLE 3.A-5 (continued)

**Arithmetic Instructions**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Source</th>
<th>Destination</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADD with carry</strong></td>
<td>ADC mem, data</td>
<td>[mem] ← [mem] + data + CY</td>
<td>Immediate Memory</td>
<td>ADC BXL, 05H</td>
<td>Data can be 8- or 16-bit; mem uses DS as the segment register; all flags are affected.</td>
<td></td>
</tr>
<tr>
<td><strong>ADC reg, data</strong></td>
<td>[reg] ← [reg] + data + CY</td>
<td>Immediate Register</td>
<td>ADC SL, 05FBH</td>
<td>Data can be 8- or 16-bit; register cannot be segment register; all flags are affected; reg is not usually used as AX or AL.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ADC A, data</strong></td>
<td>[A] ← [A] + data + CY</td>
<td>Immediate Register</td>
<td>ADC AH, 05H</td>
<td>'A' can be AL or AX register; all flags are affected.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Increment</strong></td>
<td>IN reg16</td>
<td>[reg16] ← [reg16] + 1</td>
<td>Register</td>
<td>INC BX</td>
<td>This is a one-byte instruction; used to increment a 16-bit register except the segment register; does not affect the carry flag.</td>
<td></td>
</tr>
<tr>
<td><strong>INC mem/reg 1</strong></td>
<td>[mem] ← [mem] + 1</td>
<td>Memory or memory</td>
<td>INC BYTE PTR ADDR</td>
<td>This is a two-byte instruction; can be used to increment a byte or word in memory or an 8-byte register contents; segment register cannot be incremented by this instruction; does not affect the carry flag.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 3.A-5 (continued)
#### Arithmetic Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII adjust after addition</td>
<td>AAA</td>
<td>ASCII adjust [AL] after addition</td>
<td>Source</td>
<td>Destination</td>
<td>Example</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AAA</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DAA</td>
<td>Decimal adjust [AL] after addition</td>
<td>AAA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtraction</td>
<td>SUB mem/leg 1, mem/leg 2</td>
<td>[mem/leg 1] + [mem/leg 2]</td>
<td>Memory or register</td>
<td>Memory or register</td>
<td>SUB CX, DX or SUB BH, BL</td>
</tr>
<tr>
<td></td>
<td>SUB mem, data</td>
<td>[mem] + [mem]</td>
<td>Immediate Memory</td>
<td>SUB ADOR, 03H</td>
<td>Data can be 8- or 16-bit; mem used DS as the segment register; all flags are affected</td>
</tr>
<tr>
<td></td>
<td>SUB reg, data</td>
<td>[reg] + [reg]</td>
<td>Immediate Memory</td>
<td>SUB CL, 03H</td>
<td>Data can be 8- or 16-bit; this instruction is not usually used for subtracting data from AX or AL; SUB A, data is used for this; all flags are affected</td>
</tr>
</tbody>
</table>

### TABLE 3.A-5 (continued)
#### Arithmetic Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtraction (continued)</td>
<td>SUB A, data</td>
<td>[A] + [A] - data</td>
<td>Memory or register</td>
<td>Memory or register</td>
<td>SUB CL, 03H</td>
</tr>
<tr>
<td></td>
<td>SBB mem/leg 1, mem/leg 2</td>
<td>[mem/leg 1] - [mem/leg 2]</td>
<td>Memory or register</td>
<td>Memory or register</td>
<td>SBB, BX, CX</td>
</tr>
<tr>
<td></td>
<td>SBB mem, data</td>
<td>[mem] - [mem]</td>
<td>Immediate Memory</td>
<td>SBB ADOR, 03H</td>
<td>Same as SUB mem, data except this is a subtraction with borrow</td>
</tr>
<tr>
<td></td>
<td>SBB reg, data</td>
<td>[reg] - [reg]</td>
<td>Immediate Register</td>
<td>SBB BX, 0302H</td>
<td>Same as SUB reg, data except this is a subtraction with borrow</td>
</tr>
<tr>
<td></td>
<td>SBB A, data</td>
<td>[A] - [A] - data</td>
<td>Immediate Register</td>
<td>SBB AL, 03H</td>
<td>Same as SUB A, data except this is a subtraction with borrow</td>
</tr>
<tr>
<td>Decrement</td>
<td>DEC reg16</td>
<td>[reg16] - [reg16] - 1</td>
<td>Memory or register</td>
<td>Register</td>
<td>DEC CX</td>
</tr>
<tr>
<td></td>
<td>DEC mem/leg 8</td>
<td>[mem] - [mem] - 1 or [reg8] - [reg8] - 1</td>
<td>Memory or register</td>
<td>DEC BYTE PTR ADOR or DEC BL</td>
<td>Used to decrement a byte of a word in memory or an 8-bit register content; segment register cannot be decremented by this instruction; does not affect carry flag</td>
</tr>
</tbody>
</table>
### TABLE 3.A-5 (continued)
#### Arithmetic Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Source</th>
<th>Destination</th>
<th>Example</th>
<th>Illustration</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal adjust [AL] after subtraction</td>
<td>DAS</td>
<td>Decimal adjust [AL] after subtraction</td>
<td></td>
<td></td>
<td>DAS</td>
<td>This instruction uses implied addressing mode; converts [AL] into BCD; DAS should be used after BCD subtraction.</td>
<td>memory can be 8- or 16-bit; performs two's complement subtraction of the specified operand from zero; that is, two's complement of a number is formed; all flags are affected except CF = 0 if [mem/reg] is zero; otherwise CF = 1</td>
</tr>
<tr>
<td>Negation</td>
<td>NEG mem/reg</td>
<td>[mem/reg] ← [mem/reg] + 1</td>
<td></td>
<td>Memory or register</td>
<td>NEG BL, NEG BYTE PTR ADDR</td>
<td></td>
<td>memory can be 8- or 16-bit; no memory comparison allowed; result of subtraction is not provided; all flags are affected</td>
</tr>
<tr>
<td>Compare</td>
<td>CMP mem/reg, mem/reg</td>
<td>Immediate</td>
<td>Memory or register</td>
<td>Memory or register</td>
<td>CMP BHL, CL</td>
<td></td>
<td>memory can be 8- or 16-bit; no memory comparison allowed; result of subtraction is not provided; all flags are affected</td>
</tr>
<tr>
<td>(continued)</td>
<td></td>
<td>Immediate</td>
<td>Memory or register</td>
<td>Memory or register</td>
<td>CMP ADOR, 05H, 0DH, 02H</td>
<td>Subtracts 8- or 16-bit data from [mem/reg] and affects flags; no result is provided</td>
<td></td>
</tr>
<tr>
<td>Multiplication (unsigned)</td>
<td>MUL mem/reg</td>
<td>Immediate</td>
<td>Memory or register</td>
<td>Memory or register</td>
<td>MUL BH, MUL WORD PTR, BX</td>
<td>memory can be 8- or 16-bit; only CF and OF are affected; unsigned multiplication</td>
<td></td>
</tr>
<tr>
<td>Multiplication (signed)</td>
<td>IMUL mem/reg</td>
<td>Immediate</td>
<td>Memory or register</td>
<td>Memory or register</td>
<td>IMUL CL, IMUL BYTE PTR</td>
<td>memory can be 8- or 16-bit; only CF and OF are affected; signed multiplication</td>
<td></td>
</tr>
<tr>
<td>ASCII adjust after multiplication</td>
<td>AAM</td>
<td>ASCII adjust after multiplication</td>
<td></td>
<td></td>
<td>AAM</td>
<td>This instruction has implied addressing mode; after multiplying two unpacked BCD numbers, adjust the product in AX to become an unpacked BCD result; ZF, SF, and PF are affected</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 3.A-5 (continued)
#### Arithmetic Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Illustration</th>
</tr>
</thead>
</table>
| Division (unsigned)| DIV mem/reg  | \[
|                   |              | \[AX] = \[mem\] % 8 or 16 \rightarrow\\   | Memory or         | DIV BL  | mem/reg is 8-bit for 16-bit   |
|                    |              | \[AH] = \text{Remainder} \rightarrow\\   | register         | or DIV  | by 8-bit divide and 16-bit   |
|                    |              | \[AL] = \text{Quotient} \rightarrow\\   | \text{---}       | WORD PTR| for 32-bit by divide; this  |
| Division (signed)  | IDIV mem/reg | Same as DIV mem/reg                                                                | Memory or         | ADDR   | is an unsigned division;     |
|                    |              |                                                                                   | register         |         | no flags are affected;      |
| ASCII adjust for   | AAD          | ASCII adjust for division                                                          | \text{---}       |         | division by zero             |
| division           |              |                                                                                   | \text{---}       |         | automatically generates    |
|                    |              |                                                                                   | \text{---}       |         | an internal interrupt       |
| Sign extension     | CBW          | Convert a byte to a word                                                           | \text{---}       | CBW    | Extend the sign bit (bit 7)  |
|                    | CWD          | Convert a word to a doubleword (32-bit)                                           | \text{---}       | CWD    | of AL register into AX       |

#### Comments
- This instruction has implied addressing mode; converts two unpacked BCD digits in AX into equivalent binary number in AL. AAD must be used before dividing two unpacked BCD digits by an unpacked BCD byte.
- Extend the sign bit of AX (bit 15) into DX.

### TABLE 3.A-6
#### Logical Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Illustration</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>AND mem/reg 1, mem/reg 2</td>
<td>[[mem] \rightarrow [mem] &amp; [reg]</td>
<td>Memory or register</td>
<td>AND BL, CH</td>
<td>This instruction logically |</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[mem] &amp; [reg]</td>
<td>Memory or register</td>
<td></td>
<td>&amp;nds 8- or 16-bit data in |</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[mem] &amp; [reg]</td>
<td>Memory or register</td>
<td></td>
<td>&amp;nds [mem] &amp; [reg]; all flags are affected; CF and |</td>
</tr>
<tr>
<td>AND</td>
<td>mem, data</td>
<td>[mem] &amp; [data]</td>
<td>Immediate Memory</td>
<td>AND START,</td>
<td>Data can be 8- or 16-bit; |</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[mem] &amp; [data]</td>
<td>Immediate Memory</td>
<td>02H</td>
<td>mem uses DS as the segment register; all flags are affected with CF and |</td>
</tr>
<tr>
<td>AND</td>
<td>reg, data</td>
<td>[reg] &amp; [reg]</td>
<td>Immediate Register</td>
<td>AND CX,</td>
<td>Data can be 8- or 16-bit; reg |</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[reg] &amp; [reg]</td>
<td>Immediate Register</td>
<td>027H</td>
<td>cannot be segment register; |</td>
</tr>
<tr>
<td>AND</td>
<td>A, data</td>
<td>[A] &amp; [data]</td>
<td>Immediate Register</td>
<td>AND AL,</td>
<td>Data can be 8- or 16-bit; |</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[A] &amp; [data]</td>
<td>Immediate Register</td>
<td>02H</td>
<td>A must be AL or AX; all flags are affected with CF and CF cleared to zero</td>
</tr>
</tbody>
</table>

#### Comments
- Data can be 8- or 16-bit; A must be AL or AX; all flags are affected with CF and CF cleared to zero.
### Table 3-A-6 (continued)

**Logical Instructions**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>NOT reg</td>
<td>(reg) ← [reg]'</td>
<td>Register</td>
<td>NOT BX</td>
<td>Keeps complement of a register; mem and reg can be 8- or 16-bit; segment registers are not allowed; no flags are affected.</td>
</tr>
<tr>
<td></td>
<td>NOT mem</td>
<td>[mem] ← [mem]'</td>
<td>Memory</td>
<td>NOT [SI]</td>
<td>mem uses DS as the segment register; no flags are affected.</td>
</tr>
<tr>
<td>OR</td>
<td>mem/reg 1,</td>
<td>[mem/reg 1] ←</td>
<td>Memory or</td>
<td>OR BX, CX</td>
<td>No memory-to-memory operation is allowed; (mem) or (reg) can be 8- or 16-bit; all flags are affected with CF and OF cleared to zero; no segment registers are allowed; mem uses DS as segment register.</td>
</tr>
<tr>
<td></td>
<td>mem/reg 2</td>
<td>[mem/reg 2]</td>
<td>register</td>
<td></td>
<td>mem and data can be 8- or 16-bit; mem uses DS as segment register; all flags are affected with CF and OF cleared to zero.</td>
</tr>
<tr>
<td>OR</td>
<td>mem, data</td>
<td>[mem] ← [mem] v data</td>
<td>Immediate Memory</td>
<td>OR [DI], 02H</td>
<td>No flags are affected with CF and OF cleared to zero.</td>
</tr>
<tr>
<td>OR</td>
<td>reg, data</td>
<td>[reg] ← [reg] v data</td>
<td>Immediate Register</td>
<td>OR BL, 03H</td>
<td>Data can be 8- or 16-bit; all flags are affected with CF and OF cleared to zero.</td>
</tr>
<tr>
<td>OR</td>
<td>A, data</td>
<td>[A] ← [A] v data</td>
<td>Immediate Register</td>
<td>OR AX, 2050H</td>
<td>Data can be 8- or 16-bit; A must be AL or AX; all flags are affected with CF and OF cleared to zero.</td>
</tr>
</tbody>
</table>

---

**Table 3-A-6 (continued)**

**Logical Instructions**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>TEST mem/reg 1, mem/reg 2</td>
<td>[mem/reg 1] ← [mem/reg 2]</td>
<td>Memory or register</td>
<td>TEST CL, BL</td>
<td>No memory-to-memory TEST is allowed; no result is provided; all flags are affected with CF and OF cleared to zero; mem and data can be 8- or 16-bit; no segment registers are allowed; mem uses DS as the segment register.</td>
</tr>
<tr>
<td></td>
<td>mem, data</td>
<td>[mem] ← [mem]</td>
<td>Immediate Memory</td>
<td>TEST START, 02H</td>
<td>No flags are affected with CF and OF cleared to zero; mem uses DS as the segment register.</td>
</tr>
<tr>
<td>TEST</td>
<td>reg, data</td>
<td>[reg] ← [reg]</td>
<td>Immediate Register</td>
<td>TEST CL, 03H</td>
<td>Reg and data can be 8- or 16-bit; no result is provided; all flags are affected with CF and OF cleared to zero; reg cannot be segment register.</td>
</tr>
<tr>
<td>TEST</td>
<td>A, data</td>
<td>[A] ← [A]</td>
<td>Immediate Register</td>
<td>TEST AL, 02H</td>
<td>A must be AX or AL; data can be 8- or 16-bit; no result is provided; all flags are affected with CF and OF cleared to zero.</td>
</tr>
</tbody>
</table>
### TABLE 3.A-6 (continued) Logical Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Source</th>
<th>Destination</th>
<th>Example</th>
<th>Illustration</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>XOR</strong></td>
<td>XOR mem/reg, 1, mem/reg 2</td>
<td>(mem/reg 1) e- (mem/reg 2) ⊕ (mem/reg 2)</td>
<td>Memory or register</td>
<td>Memory or register</td>
<td>XOR BL, CL</td>
<td>No memory-to-memory operation is allowed; (mem) or (reg) or (reg2) can be 8- or 16-bit; all flags are affected, with CF and OF cleared to zero; mem uses DS as the segment register</td>
<td>Data and mem can be 8- or 16-bit; mem uses DS as the segment register; mem cannot be segment registers; all flags are affected with CF and OF cleared to zero; should not be used for XORing AL or AX with immediate data A must be AL or AX; data can be 8- or 16-bit; all flags are affected with CF or OF cleared to zero</td>
<td>FOR BYTE</td>
</tr>
<tr>
<td></td>
<td>XOR mem/data</td>
<td>(mem) e- data</td>
<td>Immediate</td>
<td>Memory</td>
<td>XOR START</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XOR reg, data</td>
<td>(reg) e- (reg) @ data</td>
<td>Immediate</td>
<td>Register</td>
<td>XOR BL, 0FH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XOR A, data</td>
<td>[A] e- [A] @ data</td>
<td>Immediate</td>
<td>Register</td>
<td>XOR AL, 0FH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SHIFTS and ROTATES</strong></td>
<td><strong>SAL</strong> Mem/reg, 1</td>
<td>Shift arithmetic left on byte or word in mem or reg</td>
<td>Immediate</td>
<td>Memory or register</td>
<td>SAL BYTE PTR BL, CL or SAL BX, CL</td>
<td>Operation same as SAL mem/reg 1; CL contains shift count for up to 255; zero and negative shifts are illegal; [CL] is used as shift count when shift is greater than one; OF and CF are affected; if sign bit of [mem] is changed during or after shifting, the CF is set to one; mem uses DS as segment register</td>
<td>Same as SAL mem/reg, 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SAL</strong> mem/reg, Cl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SHL</strong> mem/reg, 1</td>
<td>SHIF logical left byte or word in memreg by the shift count in CL</td>
<td>Immediate</td>
<td>Memory or register</td>
<td>SHL BL, 1</td>
<td>Same as SHL mem/reg, Cl</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SHL</strong> mem/reg, Cl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 3A-6 (continued)
#### Logical Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SHIFTS</strong></td>
<td>SAR mem/reg, 1</td>
<td>SHIFT arithmetic right byte or word in mem/seg</td>
<td>Immediate Memory or register</td>
<td>SAR AX, 1</td>
<td>FOR BYTE</td>
</tr>
<tr>
<td>and <strong>ROTATES</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>(continued)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SAR mem/CL</strong></td>
<td></td>
<td></td>
<td>Register Memory or register</td>
<td>SAR DX, CL</td>
<td>FOR WORD</td>
</tr>
<tr>
<td><strong>SHR mem/1</strong></td>
<td></td>
<td></td>
<td>Immediate Memory or register</td>
<td>SHR DX, 1</td>
<td>FOR BYTE</td>
</tr>
<tr>
<td><strong>SHR mem/CL</strong></td>
<td></td>
<td></td>
<td>Register Memory or register</td>
<td>SHR BX, CL</td>
<td>FOR WORD</td>
</tr>
</tbody>
</table>

**Operation same as SHR mem/seg, 1; however, shift count is specified in CL for shifts up to 255, zero and negative shifts are illegal.**

### TABLE 3A-6 (continued)
#### Logical Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SHIFTS</strong></td>
<td>ROL mem/1</td>
<td>ROTATE left byte or word in mem/seg</td>
<td>Immediate Memory or register</td>
<td>ROL BX, 1</td>
<td>FOR BYTE</td>
</tr>
<tr>
<td>and <strong>ROTATES</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>(continued)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ROL mem/CL</strong></td>
<td></td>
<td></td>
<td>Register Memory or register</td>
<td>ROL DX, CL</td>
<td>FOR WORD</td>
</tr>
<tr>
<td><strong>ROL mem/1</strong></td>
<td></td>
<td></td>
<td>Immediate Memory or register</td>
<td>ROL AX, 1</td>
<td>FOR BYTE</td>
</tr>
<tr>
<td><strong>ROR mem/CL</strong></td>
<td></td>
<td></td>
<td>Register Memory or register</td>
<td>ROR DX, CL</td>
<td>FOR WORD</td>
</tr>
<tr>
<td>Operation</td>
<td>Instructions</td>
<td>Interpretation</td>
<td>Addressing mode</td>
<td>Example</td>
<td>Comments</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------</td>
<td>----------------</td>
<td>-----------------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>SHIFTS and</strong></td>
<td>ROR mem/reg, CL</td>
<td>Rotate right byte or word in mem/reg by [CL]</td>
<td>Register</td>
<td>Memory or register</td>
<td>ROR AX, CL</td>
</tr>
<tr>
<td><strong>ROTATES</strong></td>
<td>RCL mem/reg, 1</td>
<td>Rotate through carry left once byte or word in mem/reg</td>
<td>Immediate</td>
<td>Register</td>
<td>RCL BX, 1</td>
</tr>
<tr>
<td></td>
<td>RCL mem/reg, CL</td>
<td>Rotate through carry left byte or word in mem/reg by [CL]</td>
<td>Register</td>
<td>Memory or register</td>
<td>RCL DX, CL</td>
</tr>
<tr>
<td></td>
<td>RCR mem/reg, 1</td>
<td>Rotate through carry right once</td>
<td>Immediate</td>
<td>Memory or register</td>
<td>RCR AX, 1</td>
</tr>
</tbody>
</table>

```
TABLE 3.A-6 (continued)
Logical Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SHIFTS and</strong></td>
<td>RCR mem/reg, CL</td>
<td>Rotate right byte or word in mem/reg by [CL]</td>
<td>Register</td>
<td>Memory or register</td>
<td>RCR DX, CL</td>
</tr>
</tbody>
</table>
```
TABLE 3.A-7
String Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Source</th>
<th>Destination</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load string</td>
<td>LODS BYTE</td>
<td>Byte or word</td>
<td>—</td>
<td>—</td>
<td>LODS BYTE</td>
<td>Load 8-bit data into AL or 16-bit data</td>
</tr>
<tr>
<td>or LODSB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LODSB or LODSB</td>
<td>into AX from a memory location</td>
</tr>
<tr>
<td></td>
<td>LODS WORD</td>
<td>Byte or word</td>
<td>—</td>
<td>—</td>
<td>LODS WORD</td>
<td>if SI in segment DS; if DF = 0, then SI</td>
</tr>
<tr>
<td>or LODSW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LODSW or LODSW</td>
<td>is incremented by 1 for byte or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>incremented by 2 for word without load;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SI is decremented by 1 for byte or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>decremented by 2 for word; LODS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>affects no flags</td>
</tr>
<tr>
<td>MOVE string</td>
<td>MOVSB</td>
<td>Byte or word</td>
<td>—</td>
<td>—</td>
<td>MOVSB or MOVSB</td>
<td>Move 8-bit or 16-bit data from the</td>
</tr>
<tr>
<td>or MOVSB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>memory location addressed by SI in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>segment DS location addressed by DI in ES</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>segment DS can be overridden by a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>prefix but destination segment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>must be ES and cannot be overridden;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if DF = 0, then SI and DI are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>incremented by 1 for byte or 2 for</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>word; if DF = 1, then SI and DI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>are decremented by 1 for byte or by</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 for word</td>
</tr>
<tr>
<td>STORE string</td>
<td>STOS BYTE</td>
<td>Byte or word</td>
<td>—</td>
<td>—</td>
<td>STOS BYTE</td>
<td>Store 8-bit data from AL or 16-bit data</td>
</tr>
<tr>
<td>or STOSB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STOSB or STOSB</td>
<td>from AX into a memory location</td>
</tr>
<tr>
<td></td>
<td>STOS WORD</td>
<td>Byte or word</td>
<td>—</td>
<td>—</td>
<td>STOS WORD</td>
<td>addressed by DI in segment ES; segment</td>
</tr>
<tr>
<td>or STOSW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STOSW or STOSW</td>
<td>register ES cannot be overridden; if</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DF = 0, then SI and DI are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>incremented by one for byte or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>incremented by two for word; the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>segment register ES in destination</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cannot be overridden</td>
</tr>
</tbody>
</table>

TABLE 3.A-7 (continued)
String Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Source</th>
<th>Destination</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>CMPS BYTE</td>
<td>Byte or word</td>
<td>—</td>
<td>—</td>
<td>CMPS BYTE</td>
<td>8- or 16-bit data addressed by [DI]; ES</td>
</tr>
<tr>
<td>string</td>
<td>or CMPSB</td>
<td></td>
<td></td>
<td></td>
<td>or CMPSB</td>
<td>is subtracted from 8- or 16-bit data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addressed by SI in DS and flags are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>affected without providing any result;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if DF = 0, then SI and DI are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>incremented by one for byte and two</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>for word after this compare; if DF = 1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>then SI and DI are decremented by one for</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>byte and two for word; the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>segment register ES in destination</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cannot be overridden</td>
</tr>
<tr>
<td>Compare</td>
<td>CMPS WORD</td>
<td>Byte or word</td>
<td>—</td>
<td>—</td>
<td>CMPS WORD</td>
<td>8- or 16-bit data addressed by [DI]; ES</td>
</tr>
<tr>
<td>memory</td>
<td>or CMPSW</td>
<td></td>
<td></td>
<td></td>
<td>or CMPSW</td>
<td>is subtracted from 8- or 16-bit data</td>
</tr>
<tr>
<td>with AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addressed by SI in DS and flags are</td>
</tr>
<tr>
<td>or AX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>affected without providing any result;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if DF = 0, then SI and DI are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>incremented by one for byte and two</td>
</tr>
<tr>
<td>SCAS BYTE</td>
<td>SCAS BYTE</td>
<td>Byte or word</td>
<td>—</td>
<td>—</td>
<td>SCAS BYTE</td>
<td>8- or 16-bit data addresses by [DI]; ES</td>
</tr>
<tr>
<td>or SCASB</td>
<td>or SCASB</td>
<td></td>
<td></td>
<td></td>
<td>or SCASB</td>
<td>is subtracted from 8- or 16-bit data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>in AL or AX and flags are affected</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>without affecting [AL] or [AX] or string</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>data; ES cannot be overridden; if DF = 0,</td>
</tr>
<tr>
<td>SCAS WORD</td>
<td>SCAS WORD</td>
<td>Byte or word</td>
<td>—</td>
<td>—</td>
<td>SCAS WORD</td>
<td>then DI is incremented by one for byte</td>
</tr>
<tr>
<td>or SCASW</td>
<td>or SCASW</td>
<td></td>
<td></td>
<td></td>
<td>or SCASW</td>
<td>and two for word; if DF = 1, then DI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>is decremented by one for byte or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>decremented by two for word</td>
</tr>
</tbody>
</table>
### TABLE 3.A-B
Unconditional Transfers

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing mode</th>
<th>Example</th>
<th>Illustration</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL subroutine (insegment direct)</td>
<td>CALL (NEAR)</td>
<td>Call a subroutine in the same segment with signed 16-bit displacement to CALL a subroutine in 8205 16-bit</td>
<td>Relative</td>
<td>NEAR</td>
<td>A subroutine can be declared as NEAR by using an assembler directive; the subroutine can then be called in the same segment by using the CALL instruction</td>
<td></td>
</tr>
<tr>
<td>CALL subroutine (insegment indirect)</td>
<td>CALL reg16</td>
<td>CALL a subroutine in the same segment addressed by the contents of a 16-bit general register</td>
<td>Register</td>
<td></td>
<td>CALL BX; The 8086 decrements SP by 2 and then pushes IP onto the stack, then specified 16-bit register contents (such as BX, SI, and DI) provide the new value for IP; CS is unchanged (insegment indirect)</td>
<td></td>
</tr>
<tr>
<td>CALL mem16</td>
<td>CALL (FAR)</td>
<td>CALL a subroutine addressed by the content of a memory location pointed to by an 8086 16-bit</td>
<td>Memory indirect</td>
<td></td>
<td>CALL (BX); The 8086 decrements SP by 2 and pushes IP onto the stack; the 8086 then loads the contents of a memory location addressed by the content of a 16-bit register</td>
<td></td>
</tr>
</tbody>
</table>

### TABLE 3.A-B (continued)

Unconditional Transfers

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Source</th>
<th>Destination</th>
<th>Example</th>
<th>Illustration</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL subroutine (insegment indirect) (continued)</td>
<td>CALL PROC (FA)</td>
<td>CALL a subroutine in another segment</td>
<td>Memory</td>
<td></td>
<td>A subroutine can be declared as FAR by using the statement BEGIN PROC FAR; the subroutine BEGIN can be called from another segment by using CALL BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL a subroutine in another segment (insegment indirect)</td>
<td>CALL DWORD PTR (reg16)</td>
<td>CALL a subroutine in another segment</td>
<td>Memory indirect</td>
<td></td>
<td>Call DWORD PTR (BX) loads into IP the contents of memory</td>
<td>This instruction decrements SP by 2, and pushes CS onto the stack; CS is then loaded with the contents of memory locations addressed</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 3.A-8 (continued)

#### Unconditional Transfers

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Example</th>
<th>Illustration</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL a subroutine in another segment</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(intersegment indirect)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- POPS IP for insegment CALLS
- POPS IP and CS for intersegment CALLS

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Example</th>
<th>Illustration</th>
</tr>
</thead>
</table>
| Unconditional jump (insegment direct) | JMP label | Unconditional jump with a signed 8-bit (short) or signed 16-bit (NEAR) displacement in the same segment | JMP START | The label START can be signed 8-bit (called SHORT jump) or signed 16-bit (called NEAR jump) displacement; the assembler usually determines the displacement value, if the assembler finds the displacement value to be signed 8-bit -1 to +127, 0 being positive, then the assembler uses two bytes for the instruction, one byte for the OP code followed by a byte for the displacement; the assembler sign extends the 8-bit displacement.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Source</th>
<th>Destination</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconditional Jump</td>
<td>JMP</td>
<td>[IP] ← [reg 16]</td>
<td>Register</td>
<td></td>
<td>JMP BX</td>
<td>Jump to a non-address specified by the contents of a 16-bit memory location</td>
</tr>
<tr>
<td>(intrasegment direct)</td>
<td>reg16</td>
<td>[CS] is unchanged</td>
<td>Indirect</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unconditional Jump</td>
<td>JMP</td>
<td>[IP] ← [mem]</td>
<td>Memory</td>
<td></td>
<td>JMP [BX]</td>
<td>Jump to an address specified by the contents of a 16-bit memory location</td>
</tr>
<tr>
<td>(intrasegment indirect)</td>
<td>mem16</td>
<td>[CS] is unchanged</td>
<td>Indirect</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 3.A-8 (continued) Unconditional Transfers**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Source</th>
<th>Destination</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Jump</td>
<td>JMP</td>
<td>Unconditionally jump to another segment</td>
<td>Memory —</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(indirect jump)</td>
<td>label dARG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unconditional Jump</td>
<td>JMP</td>
<td>Unconditionally jump to another segment</td>
<td>Memory —</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(intrasegment direct)</td>
<td>label dARG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump unconditionally to another segment</td>
<td>JMP DWORD PTR [reg16]</td>
<td>Unconditionally jump to another segment</td>
<td>Memory —</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(intrasegment indirect)</td>
<td>PTR [reg16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This instruction loads the contents of memory locations addressed by [reg15] and [reg16 + 1] in DS into IP. It then loads the contents of memory.
### Table 3.A-8 (continued) Unconditional Transfers

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing Mode</th>
<th>Example</th>
<th>Illustration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump unconditionally to another segment (intersegment indirect) (continued)</td>
<td></td>
<td></td>
<td>Source Destination</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Jump if above
- Jump if below or equal
- Jump if not above or equal
- Jump if carry
- Jump if not carry
- Jump if overflow
- Jump if not overflow
- Jump if greater
- Jump if not greater
- Jump if equal
- Jump if not equal

- Locations addressed by [BX] and [BX + 1] in DS into DS; typical 8086
- Locations addressed by [BX + 2] and [BX + 3] in DS

### Table 3.A-9 Conditional Branch Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Source Destination</th>
<th>Example</th>
<th>Illustration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional branch instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Jump if above
- Jump if below or equal
- Jump if not above or equal
- Jump if carry
- Jump if not carry
- Jump if overflow
- Jump if not overflow
- Jump if greater
- Jump if not greater
- Jump if equal
- Jump if not equal

- Same as JA/NBE except that the jump is taken if CF = 0; used for unsigned comparison
- Same as JA/NBE except that the jump is taken if CF = 1; used for unsigned comparison
- Same as JA/NBE except that the jump is taken if CF = 1 or OF = 1; used for signed and unsigned comparison
- Same as JA/NBE except that the jump is taken if ZF = 0; used for signed comparison
- Same as JA/NBE except that the jump is taken if SF = 0; used for signed comparison
### TABLE 3.A-9 (continued)
Conditional Branch Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Interpretation</th>
<th>Source</th>
<th>Destination</th>
<th>Example</th>
<th>Illustration</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional branch instruction (continued)</td>
<td>jump if 'less' or 'less or equal' jump if not greater</td>
<td>Relative</td>
<td>Destination</td>
<td>IJNGE START</td>
<td>Same as JNJIE except that the jump is taken if (IF + OF) = 1; used for signed comparison</td>
<td></td>
</tr>
<tr>
<td>IJNGE disp 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IGE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>disp 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### TABLE 3.A-10
Loop Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing Mode</th>
<th>Example</th>
<th>Illustration</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>LOOP disp 8</td>
<td>Loop if CX ≠ 0</td>
<td>Relative</td>
<td>LOOP</td>
<td>START</td>
<td>Decrement CX by 1, without affecting flags and loop with signed 8-bit displacement from -128 to +127; zero being positive; if CX ≠ 0</td>
</tr>
<tr>
<td></td>
<td>LOOPZ</td>
<td>Loop while equal/loop while zero</td>
<td>Relative</td>
<td>LOOPZ</td>
<td>START</td>
<td>Decrement CX by one without affecting flags and loop with signed 8-bit displacement if CX ≠ 0 and ZF = 1, which results from execution of the previous instructions</td>
</tr>
<tr>
<td></td>
<td>LOOPNZ</td>
<td>Loop while not equal/loop while not zero</td>
<td>Relative</td>
<td>LOOPNZ</td>
<td>START</td>
<td>Decrement CX by one without affecting flags and loop with signed 8-bit displacement if CX ≠ 0 and ZF = 0, which results from execution of the previous instructions</td>
</tr>
<tr>
<td></td>
<td>JXXZ disp 8</td>
<td>Jump if CX ≠ 0</td>
<td>Relative</td>
<td>JXXZ</td>
<td>START</td>
<td>Jump if CX ≠ 0; this instruction is useful at the beginning of a loop to bypass the loop if CX = 0</td>
</tr>
</tbody>
</table>
### TABLE 3.4-11
Interrupt Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing Mode</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT n</td>
<td></td>
<td>Source</td>
<td>Destination</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 to 255</td>
<td>SP ← SP - 2</td>
<td>Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(SP) ← Flags</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF ← 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TF ← 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTO</td>
<td></td>
<td>Interrupt on overflow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRET</td>
<td></td>
<td>Interrupt return</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Software interrupts can be used as supervisor calls, that is, request for service from an operating system; a different interrupt type can be used for each type of service that the operating system could supply for an application program. Software interrupt instructions can also be used for checking interrupt service routines written for hardware-initiated interrupts.

Generates an internal interrupt if OF = 1 executes INT 14 can be used after an arithmetic or logic operation to activate a service routine if OF = 1; when INTO is executed and if OF = 1, operations similar to INT 14 take place.

POPS IP, CS and Flags from stack, IRET is used as return instruction at the end of a service routine for both hardware and software interrupts.

### TABLE 3.4-12
Processor Control Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instructions</th>
<th>Interpretation</th>
<th>Addressing Mode</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag operations, external synchronization, and no operation</td>
<td>STC</td>
<td>CF ← 1</td>
<td>Source</td>
<td>STC</td>
<td>Set carry to one</td>
</tr>
<tr>
<td></td>
<td>CLC</td>
<td>CF ← 0</td>
<td>Destination</td>
<td>CLC</td>
<td>Clear carry to zero</td>
</tr>
<tr>
<td></td>
<td>CMC</td>
<td>CF ← CF'</td>
<td></td>
<td>CMC</td>
<td>Ones complement carry</td>
</tr>
<tr>
<td></td>
<td>STD</td>
<td>DF ← 1</td>
<td></td>
<td>STD</td>
<td>Set direction flag to 1</td>
</tr>
<tr>
<td></td>
<td>CLD</td>
<td>DF ← 0</td>
<td></td>
<td>CLD</td>
<td>Clear direction flag to zero</td>
</tr>
<tr>
<td></td>
<td>STI</td>
<td>IF ← 1</td>
<td></td>
<td>STI</td>
<td>Set interrupt enable flag to one to enable maskable interrupts</td>
</tr>
<tr>
<td></td>
<td>CLI</td>
<td>IF ← 0</td>
<td></td>
<td>CLI</td>
<td>Clear interrupt enable flag to zero to disable maskable interrupts</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
<td>NO operation</td>
<td></td>
<td>NOP</td>
<td>8086 does nothing</td>
</tr>
<tr>
<td></td>
<td>HLT</td>
<td>HLT</td>
<td></td>
<td>HLT</td>
<td>Halt</td>
</tr>
<tr>
<td></td>
<td>WAIT</td>
<td>8086 enters wait state</td>
<td></td>
<td>WAIT</td>
<td>Causes CPU to enter wait state if the 8086 TEST pin is high; while in wait state, the 8086 continues to check TEST pin for low; if TEST pin goes back to zero, the 8086 executes the next instructions; this feature can be used to synchronize the operation of the 8086 to an event in external hardware. This instruction is used to pass instructions to a coprocessor such as the 8087 floating point coprocessor which simultaneously monitors the system bus with the 8086; the coprocessor OP codes are 6 bit wide; the coprocessor treats normal 8086 instructions as NOP's; the 8086 fetches all instructions from memory; where</td>
</tr>
</tbody>
</table>
## Chapter 4

### INTEL 80186/80286/80386

This chapter describes the internal architecture, addressing modes, instruction set, and I/O techniques associated with the 80186, 80286, and 80386 microprocessors. Interfacing capabilities to typical memory and I/O chips are also included. Finally, virtual memory concepts associated with the 80286 and 80386 are covered.

### 4.1 INTEL 80186 AND 80286

This section will cover the two enhanced versions of the 8086 microprocessor: Intel 80186 and 80286. The state-of-the-art technology has allowed fabrication of thousands of transistors on a single chip. This, in turn, has led to the Intel 80186 and 80286 microprocessors. The Intel 80186 includes the Intel 8086 and six separate functional units in a single chip, while the 80286 has integrated memory protection and management into the basic 8086 architecture.

#### 4.1.1 INTEL 80186

The Intel 80186 family contains two microprocessors: Intel 80186 and 80188. The only difference between them is that the 80186 has a 16-bit data bus, while the 80188 includes an 8-bit data bus. The 80186 is packaged in a 68-pin leadless package. The 80186 can be operated at two different clock speeds: 3 MHz (80186) and 6 MHz (80186-6). It can directly address one megabyte of memory. The 80186 contains the 8086-2 microprocessor and several additional functional units. The major on-chip circuits include a clock generator, two independent DMA channels, a
a programmable interrupt controller, three programmable 16-bit timers, and a chip select unit.

The 80186 provides double the performance of the standard 8086. The 80186 includes 10 new instructions beyond the 8086. The 80186 is completely object code compatible with the 8086. It contains all the 8086 registers and generates the 20-bit physical address from a 16-bit segment register and a 16-bit offset in the same way as the 8086. The 80186 does not have the N/P/MX pin. The 80186 has enough pins to generate the minimum mode type pins. S0-S3 status signals can be connected to external bus controller chips such as 8237 for generating the maximum mode type signals.

Figure 4.1 shows the 80186 functional block diagram. The DMA unit provides two DMA channels. Each DMA channel contains 20-bit source and destination pointers used to address the source and destination of the data transferred. Each of these pointers may independently address either memory or I/O. After each DMA cycle, these pointers are incremented, decremented, or kept constant. A 16-bit transfer count register is included in each channel which contains a number of DMA transfers to be performed. Data can be transferred either by the byte or by 16-bit words. Each data transfer takes 2 bus cycles (a minimum of 8 clocks); one cycle to read data and the other to store data. Each channel contains a 16-bit control register. This register specifies information such as: (1) whether 8- or 16-bit data are to be transferred; (2) whether DMA operation will be terminated when count register is zero; (3) whether source and destination pointers will be incremented, decremented, or maintained constant; (4) whether interrupts will be generated after the last transfer; and (5) whether source or destination pointers will address memory or I/O. No separate DMA acknowledge pulse is provided since both source and destination pointers are maintained. A read from a requesting source or a write to a requesting destination should be used as the DMA acknowledge signal. A chip select line can be used as the DMA acknowledge signal, since the chip select lines can be programmed to be active for a given block of memory or I/O space. The DMA channel may be programmed such that one channel has priority over the other. DMA cycles will take place whenever the ST/STOP bit in the control register is set to one.

The 80186 contains three independent 16-bit timers/counters. Two of these timers can be programmed to count external events. The third timer is not connected to any external pins. This timer only counts the 80186 clock cycles and can be used to interrupt the 80186 after a programmed amount of time to provide a count pulse to the DMA unit after a specific amount of time. The mode/control register allows the 80186 to operate in two modes. These are non-iRMX-86 and iRMX-86 modes. The non-iRMX-86 is also called the master mode. In the master mode, the interrupt controller acts as the interrupt controller for the system, while in iRMX-86 mode, the interrupt controller operates as a slave to an external interrupt controller, such as the 8259, for the system. The 8086 interrupt controller can be placed in the iRMX-86 mode by setting the iRMX mode bit in the peripheral control block relocation register. Some of the interrupt controller registers and interrupt controller pins change definition between these modes, but their functions are basically identical. Upon reset, the 80186 operates in the non-iRMX-86 (master) mode.

The 80186 interrupt controller allows the 80186 to receive interrupts from internal or external sources. Internal interrupt sources (timers and DMA channels) can be disabled by their own control registers or by mask bits within the interrupt controller. The interrupt controller has a special iRMX-86 compatibility that allows the use of the 80186 within the iRMX-86 operating system interrupt structure. The 80186 will accept external
Interrupts only in the master mode. The 80186 is provided with five dedicated pins for external interrupts. These pins are NMI, INT0, INT1, INT2/INTA0, and INT3/INTA1. NMI is the only non-maskable interrupt.

In the master mode, the interrupt controller provides three modes of operation. These are fully nested mode, cascade mode, and specially nested mode. In the fully nested mode, all four maskable interrupt pins are used as direct interrupt requests. The interrupt vectors are obtained by the 80186 internally. In the cascade mode, INT0 is used as the interrupt pin and can be connected to the 8259 INT pin (output) and the INT2/INTA0 is used as the interrupt acknowledge pin for the INT0. The INT2/INTA0 can be connected to the 8259 INTA input pin. Similarly, the INT3/INTA1 can be used as the interrupt acknowledge pin for the INT1 via the 8259. The use of dedicated acknowledge signal eliminates the need for external hardware to generate INTA and device select signals.

When the 8259 receives an interrupt from an external device, it activates the 80186 INT0 or INT1. The interrupt acknowledge pins INT2/INTA0 or INT3/INTA1 can then be used to enable an octal buffer to place the interrupt-type number on the 80186 low data byte. The 80186 thus determines the address vector. The special fully nested mode is entered by setting the proper bits in INT0 or INT1 control register. It enables complete nestedness with external 8259 masters. This allows multiple interrupts via a single pin. Upon acceptance of an interrupt (hardware, INT instructions, or instruction exceptions such as divide by 0), the 80186 pushes CS, IP, and status word onto the stack just like the 8086. Also, similar to the 8086, an interrupt pointer table with 256 entries provides interrupt address vectors (IP and CS) for each interrupt. This vector identifies the appropriate table entry. Non-maskable interrupts use an internally supplied vector, while the vectors for maskable interrupts are provided by the user via external hardware. The vectors for INT instructions and instruction exceptions are generated internally by the 80186. The 80186 includes an on-chip clock generator/crystal oscillator circuit. Like the 8085, a crystal connected at the 80186 X1 X2 pins is divided by 2 internally. The built-in chip select unit is an address decoder. This unit can be programmed to generate six memory chip select (LCS, UCS, and MCSO — 3 pins) and seven I/O or peripheral chip select (PCSO-4, PCS1-A1, and PCS2-A2 pins). This unit can be programmed to generate an active low chip select when a memory or port address in a particular range is sent out. For example, the 80186 outputs low on the LCS (lower chip select) pin when it accesses an address between 00000H and a higher address (in the range of 1K to 256K) programmable by the user via a control word. On the other hand, the 80186 outputs low on the UCS (upper chip select) pin when it accesses an address between a user programmable lower address (by placing some bits in a control word via an instruction) and upper fixed address FFFFFFFH. The four middle chip select pins (MC5O-3) are activated low by the 80186 when it accesses an address in the mid range. For peripheral chip selects, a base address can be programmed via a control word. The 80186 sends low on the PCSO when it accesses a port address located in a block from this base address to up to 128 bytes. The 80186 sends low on the other chip selects PCS1 - 6 when one of six contiguous 128-byte blocks above the block for PCSO. Like the 8086, memory for the 80186 is set up as odd (BEH = 0) and even (AO) memory banks.

The 80186 provides eight addressing modes. These include register, immediate, direct, register indirect (SI, DI, BX, or BP), based (BX or BP), indexed (SI or DI), based indexed, and indexed based with displacement modes.

Typical data types provided by the 80186 include signed integer, ordinal (unsigned binary number), pointer, string, ASCII, BCD, unpacked/packed BCD, and floating point. The 80186 instruction set is divided into seven types. These are data transfer, arithmetic, shift/rotate, string, control transfer, high level instructions (for example, the BOUND instruction detects values outside prescribed range), and processor control. As mentioned before, the 80186 includes 10 new instructions beyond the 8086. These 10 additional instructions are listed below:

**Data Transfer**
- PUSH — Push all registers onto stack
- POPA — Pop all registers from stack
- PUSH immediate — Push immediate numbers onto stack

**Arithmetic**
- IMUL destination register, source, immediate data means immediate data source → destination

**Logical**
- SHIFTR/ROTA T destination, immediate data shifts/rotates register or memory contents by the number of times specified in immediate data

**String Instructions**
- INS — Input string byte or string word
- OUTS — Output string byte or string word
High Level Instructions

ENTER — Format stack for procedure entry
LEAVE — Restore stack for procedure exit
BOUND — Detect values outside predefined range

Let us explain some of these instructions:

- IMUL destination, source, immediate data. This is a signed multiplication. This instruction multiplies signed 8- or 16-bit immediate data with 8- or 16-bit data in a specified register or memory location and places the result in a general-purpose register. As an example, IMUL DX, CX, 03H multiplies the contents of CX by 3 and places the lower 16-bit result in DX. Note that the immediate 8-bit data of 03H are sign extended to 16-bit prior to multiplication. A 32-bit result is obtained but only the lower 16-bit is saved by this instruction.

- ROL/ROR/SAL/SAR destination, immediate data. Like the 8086, data can be specified in CL up to a maximum of 32. Unlike the 8086, the immediate data of up to 32 can be specified as immediate data in the 80186 instruction. Note that the 8086 allows an immediate data of one only.

- INS/INSB/INSW inputs a byte of a word from a port addressed by DX to a memory location in ES pointed to by DI. If DF = 0, DI will automatically be incremented (by 1 for byte and 2 for word) after execution of this instruction. On the other hand, if DF = 1, DI is automatically decremented (by 1 for byte and 2 for word) after execution of this instruction. The instructions INSB (or INS BYTE PTR) for byte and INSW (or INS WORD PTR) for word are used.

- OUTS/OUTSB/OUTSWstring similarly provides outputting to a port addressed by DX from a source string in ES with offset in SI. A typical example of inputting 50 bytes of l/O data via a port into a memory location is given below (assume ES is already initialized):

```
STD ; Set DF to 1.
MOV DI, ADDR ; Initialize DI.
MOV DX, 0E124H ; Load port address.
MOV CX, 50 ; Initialize count.
REP INSB ; Input port until CX = 0.
STOP JMP STOP ; Halt.
```

The instructions OUTSB (or OUTS BYTE PTR) for byte and OUTSW (or OUTS WORD PTR) for word are used.

- The ENTER instruction is used at the beginning of an assembly language subroutine which is to be called by a high level language program such as Pascal. The main purpose of ENTER is to reserve space on the stack for variables used in the subroutine.

The ENTER instruction has two immediate operands:

```
ENTER imm16, imm8
```

The first operand imm16 specifies the total memory area allocated to the local variables, which is 16 bits wide (0 to 64K bytes). The second operand imm8, on the other hand, is 8 bits wide and specifies the number of nested subroutines.

For the main subroutine, imm8 = 0. Note that nested subroutines mean a subroutine calling another subroutine. For example, if there are three subroutines SUB1, SUB2, and SUB3 such that the main program M calls SUB1, SUB1 calls SUB2 and SUB2 calls SUB3, then imm8 = 2. ENTER can be used to allocate temporary stack space for local variables for the subroutines.

In the second operand, imm8 = 0, the ENTER instruction pushes the frame pointer BP onto the stack. ENTER then subtracts the first operand imm16 from the stack pointer and sets the frame pointer, BP, to the current stack pointer value.

The LEAVE instruction is used at the end of an assembly language subroutine (usually before the RET instruction) which is to be called by a high level language program. The LEAVE does not have any operand. The LEAVE instruction should be used with the ENTER instruction. The ENTER allocates space in stack for variables used in the subroutine, while the LEAVE instruction deallocates this space and ensures that SP and BP have the original values that they had prior to execution of the ENTER.

The RET instruction then returns to the appropriate address in the main program.

As an example of application of ENTER and LEAVE instructions, suppose that a subroutine requires 16 bytes of stack for local variables. The instructions ENTER 16, 0 at the subroutine's entry point and a LEAVE before the RET instruction will accomplish this. The 16 local bytes may be accessible. When the 80186 accesses an array, the BOUND instruction can be used to ensure that data outside the array are not accessed. When the BOUND is executed, the 80186 compares the content of a general-purpose register (initialized by the user with the offset of the array currently being accessed) with the lower and upper bounds of the array.
(loaded by the user prior to BOUND). The format for BOUND is
BOUND reg16, memory32. The first operand is the register containing
the array index and the second operand is a memory location containing
the array bounds. If the index value violates the array bounds, an exception
(maskable interrupt 5) takes place. A service routine can be executed by the
user to indicate that the array element being accessed is out of bounds. As
an example, consider BOUND SI, ADDR. The lower bound of the array
is contained in address ADDR and the upper bound is in address ADDR + 2. Both bounds are 16 bits wide. For a valid access [SI] must be greater
than or equal to [ADDR] and less than or equal to [ADDR + 2]; otherwise
interrupt 5 occurs. The BOUND instruction is normally placed just before
the array itself, making the array addressable via a constant from the start
of the array.

4.1.2 INTEL 80286

The Intel 80286 is a high-performance 16-bit microprocessor with on-
chip memory protection capabilities primarily designed for multiuser/
multitasking systems. The IBM PC/AT and its clones capable of multi-
tasking operations use the 80286 as their CPU. The 80286 can address 16
megabytes (2^24) of physical memory and 1 gigabyte (2^30) of virtual memory
per task. The 80286 can be operated at three different clock speeds. These are
4 MHz (80286-4), 6 MHz (80286-6), and 8 MHz (80286-8).

The 80286 has two modes of operation. These are real address mode
and protected virtual address mode (PVAM). In the real address mode, the
80286 is object code compatible with the Intel iAPX 86/88 family. In
protected virtual address mode, the 80286 is source code compatible with
the iAPX 86/88 family and may require some software modification to use
virtual address features of the 80286.

The 80286 includes special instructions to support operating systems.
For example, one instruction can end a current task execution, save its
state, switch to a new task, load its state, and begin executing the new task.

The 80286's performance is up to six times faster than the standard 5-
MHz 8086. The 80286 is housed in a 28-pin leadless flat package. Figure
4.2 shows a functional diagram of the 80286. It contains four separate
processing units. These are the Bus Unit (BU), the Instruction Unit (IU),
the Address Unit (AU), and the Execution Unit (EU). The BU provides
all memory and I/O read and write operations. The BU also performs data
transfer between the 80286 and coprocessors such as the 80287. The
prefetcher in the BU prefetches instructions of up to 6 bytes and places
them in a queue.

The Instruction Unit (IU) translates or decodes up to 3 instructions
and places them in a queue for execution by the execution unit.
The following instruction sequence will accomplish this:

SMSW CX ; Store MSW into a general register such as CX
OR CX, 1 ; Set only the PE bit (bit 0 in MSW)
LMSW CX ; Load the new value back to MSW.

After the above instruction sequence is executed, the 80286 operates in PVAM with memory management capabilities. In the PVAM, the 80286 is compatible with the 8086/8088 at the source code level but not at the machine code level. This means that most 8086/8088 programs must be recompiled or reassembled.

When the 80286 is in the protected mode, the on-chip MMU is enabled which expects several address-mapping tables to exist in memory. The 80286, in this mode, will automatically access these tables for translating the virtual addresses used by the user to physical addresses. The 80286 supports the following data types:

- 8-bit or 16-bit signed binary numbers (integers)
- Unsigned 8- or 16-bit numbers (ordinal)
- A 32-bit pointer comprised of a 16-bit segment selector and 16-bit offset
- A contiguous sequence of bytes or words (strings)
- ASCII
- Packed and unpacked BCD
- Floating point

The 80286 provides 8 addressing modes. These include register, immediate, direct, register indirect, based, indexed, base-index, and based-indexed with displacement modes. The 80286 includes all 80186 instructions plus more. The new instructions are for supporting the PVAM of the 80286 via an operating system.

These instructions are listed in the following and are used by the operating system:

- CTST Clear task switch flag to zero located in the MSW register
- LCDT Load global descriptor table register from memory
- SCDT Store global descriptor table register into memory
- UIDT Load interrupt descriptor table register from memory
- LLDT Load selector and associated descriptor into LDTR (local descriptor table register)
- SLDT Store selector from LDTR in specified register or memory
- LTR Load task register and descriptor for TSS (task state segment)
supported by the 80286 since they may require fast task switching.
Note that an exception in a running task or an interrupt from a
peripheral device requires task switching.

The above 80286 features are not available in the 8086. Upon reset, the
80286 operates in real mode and provides all features of the 8086. An
80286 initialization program in the real mode can be executed to operate
the 80286 in virtual mode in which the functions of memory management,
protection, and task switching are available.

4.1.2 a 80286 Memory Management

The 80286 memory management is based on address translation. That
is, the 80286 translates logical addresses (addresses used in programs) to
physical addresses (addressing required by memory hardware). The 80286
memory pointer includes two 16-bit words: one word for a segment
selector and the other as an offset into the selected segment. The real and
virtual modes compute physical addresses from these selector and offset
values in different ways.

In the real mode address, the 80286 computes the physical address from
a 16-bit (selector) content and a 16-bit offset just like the 8086/
80186. It shifts the 16-bit selector four times to the left and then adds the
16-bit offset to determine the 20-bit physical address. As mentioned
before, even though the 80286 has 24 address pins (A0-A23), in the real
address mode pins A0-A19 are used and A20-23 pins are ignored.

In the Protected Virtual Address Mode (PVAM or virtual mode for short),
the 16-bit address is called a virtual address. Just like the logical
address, the virtual address includes a 16-bit selector and a 16-bit offset.
The 80286 determines the 24-bit physical address by first obtaining a 24-
bit value from a table in memory using the segment value (selector) as an
index (rather than shifting the segment value 4 times to left as in the real
mode) and then adding the 16-bit offset. Figure 4.3 shows the 80286
virtual address translation scheme.

The 16-bit selector is divided into a 13-bit index, one-bit Table
Indicator (TI), and two-bit Requested Privilege Level (RPL). The 13-bit
index is used as a displacement to access the selected table. Each entry in
the table is termed a descriptor. An index can start from a value of 0 to a
higher value. The index value refers to a descriptor in the table. For
easy, index value 0 refers to the descriptor 0. Each descriptor is 8 bytes
wide and contains the 24-bit base address required for physical address
calculation. This address only occupies three bytes of the 8-byte descriptor,
the meaning of the other bytes will be explained later. The single-bit TI
for implementing the 80286's protection features, RPL is not used in physical address calculation.

The 24-bit physical address is then generated by the 80286 by adding the 24-bit base address of the selected descriptor and the 16-bit offset.

Note that in the above, when TI = 0 (GDT selector) and Index = 0, a null selector is selected. The selector does not correspond to the 0th GDT descriptor. Null selectors can be loaded into a segment register, but use of null selectors in virtual address translation would generate an 80286 exception.

Figure 4.4 shows the 80286 address translation registers. The segment registers CS, DS, SS, and ES have already been discussed before. The GDT and LDT registers are only used in the 80286 virtual mode address translation. The GDT register stores the 24-bit base address and the length of the GDT (in bytes) minus one. During system initialization, the GDT is loaded and is usually kept unchanged after this. The 80286 generates an exception when indexing beyond the GDT limit is attempted.

![Segment registers](image)

The LDT register stores a 16-bit selector for a selector in the present LDT. A GDT descriptor is indexed by this selector. The LDT register makes task switching very fast. For example, the 80286 local address space can be modified during task switching by updating the LDT register.

Figure 4.5 shows a flowchart showing steps for accessing a memory byte. In the flowchart, it can be seen that when accessing memory the 80286 address translation scheme requires two 8-bit descriptors in memory (for LDT) and two 24-bit additions.

The 80286 has improved the speed of these memory accesses via additional hardware. This is accomplished by providing an internal register called the shadow register with the LDT register and four shadow registers with the four segment registers (one for CS, one for DS, one for SS, and
The shadow registers are internal to the 80286 and are not accessible via software. They are implemented in the 80286 hardware. The shadow registers store respective descriptors associated with the LDT registers and the segment register. Whenever the LDT register or a segment register is loaded with a selector, the associated descriptor is automatically moved into the corresponding shadow register. The 80286 internally performs steps 1 to 6 of the flowchart via hardware to load a shadow register with the associated descriptor. In order to compute the physical address, the 80286 obtains the segment base address from the descriptor, adds it to the offset, and then accesses memory with this physical address.

In summarizing the topic of shadow registers, these registers are provided to speed up memory references. The shadow registers are internal and cannot be accessed by instructions. Whenever a selector is moved into a segment register, the associated shadow register is updated with its descriptor automatically. Therefore, any memory accessed with respect to the segment register does not require referral to a look-up table, since the descriptor loaded into the shadow register contains the base address of the selected segment.

Note that in the virtual mode, the 80286 descriptor table can store a maximum of $2^{16}$ (16-bit index) descriptors and each segment can specify a segment of up to $2^{20}$ bytes. Therefore, a task can have its own address space of up to $2^{21} \times 2^{16} = 2^{27}$ bytes and can share $2^{20}$ bytes with all other tasks. Therefore, an address space of 1 gigabyte ($2^{30}$ bytes) can be assigned to a task.

The following 80286 memory management instructions include loading and storing the address translation register and checking the contents of descriptors:

- LGDT: Load GDT register
- SGDT: Store GDT register
- LLDT: Load LDT register
- SLDT: Store LDT register
- LAR: Load Access Rights
- LSL: Load Segment Limit

### 4.1.2.2 Protection

In this section, two topics will be covered. These are (1) protecting one 80286 task from another and (2) activating an operating system to protect itself.

The 80286 provides protection mechanisms for supporting multitasking and virtual memory features. The 80286 includes some basic protection features such as segment limit and segment usage checking.
protections are useful even though multitasking and virtual memory may not be available in a system. The basic protection mechanism also allows assignment of privilege levels to virtual memory space in a hierarchical manner.

The 80286 privilege level mechanism uses certain rules to define the hierarchical order. This allows protection of the operating system independent of the user. The 80286 includes special descriptor table entries named call gates to permit CALLS to higher privilege code segments at 80286 does not allow accessing a higher privilege entry point without an accessible gate.

The 80286 provides some instructions controlled by its I/OPL (Input/Output Privilege Level) feature to protect shared system resources. The L/O instructions are permitted at the lowest privilege level.

The 80286 on-chip protection features handle basic violations such as trying to access code segment instead of stack segment by trapping into the operating system's appropriate routine. Thus, the operating system recovers a faulty task by taking whatever actions are necessary. The 80286 protection hardware provides information such as stack status to inform the operating system of the fault type.

The 80286 on-chip protection hardware provides up to four privilege levels in a hierarchical manner which can be used to protect the operating system from unauthorized access. The 80286 can read specific bits in its segment descriptor to obtain privilege levels of each code and data segment. Figure 4.6 shows the format for a segment descriptor for code or data segment.

The descriptors include four words. Bit 3 of the 3-bit type field of the access rights byte is called the Executable (E) bit. E = 1 identifies a code segment descriptor, while E = 0 identifies the segment as a data segment descriptor. The two-bit DPL (Descriptor Privilege Level) provides the privilege level of the descriptor. The DPL field specifies a hierarchical privilege system of four levels 0 thru 3.

Level 0 is the highest level while level 3 is the lowest. Privilege level provides protection within a task. Operating system routines, interrupt handlers, and other system software can be included and protected within the virtual address space of each task using the four levels of privileges. Each task in the system has a separate stack for each of its privilege levels. Typical examples of privilege levels are summarized as follows:

A single privilege level can be assigned to the codes of a dedicated 80286. In this case, all load/store and I/O interactions are available. The 80286 can be initialized by setting the PE bit in MSW to one and
then loading the GDTR with appropriate values to address a valid global descriptor table.

- Privilege levels can be assigned to user and supervisor mode types of applications. In this case, all system software can be defined as level 0 (higher level) and all other programs at some lower privilege level.
- For large applications, the system software can be divided into critical and noncritical. All critical software can be defined as a kernel with the highest privilege level (level 0); the noncritical portion of the system software is defined with levels 1 and 2, while all user application programs are defined with the lowest level (level 3).

Upon enabling the protected mode bit, the 80286 basic protection features are available irrespective of a single 80286-type application or user/supervisor configuration or for a large application. The descriptors limit field (the maximum offset from the base) and the access rights byte provides the 80286's basic protection features.

Segment limit checking ensures that all memory accesses are physically available in the segment. For all read and write operations with memory, the 80286 in the protected mode automatically checks the offset of an effective address with the descriptor limit (predefined). This limit checking feature ensures that a software fault in a segment does not interfere with any other segments in the system.

The descriptor access rights byte complements the limit checking. It differentiates code segments from data segments. The access rights byte along with limit checking ensures proper usage of the segments. At least three types of segments can be defined using the access byte. Data segments can be defined as read/write or read-only. Code segments can be designated as execute-only or read-only. A code segment in a particular privilege level can be accessed by using 80286 CALL or JMP instruction without a privilege level transition. A segment of equal or lower privilege level than another segment (defined as conforming via the access byte) can access that segment.

The operating system can be protected from unauthorized accesses by malicious users by using the 80286's hierarchical privilege levels. The higher privilege level segments can be protected from the lower ones by using the 80286 protection features.

The hierarchical protection levels have four logical rules. These are summarized below:

- The Current Privilege Level (CPL) at any instant of time represents the level of the code segment presently being executed. This is provided by the privilege level in the access rights byte of the descriptor. The 80286 gives the value of CPL in its code register.
- Since every privilege level has its own stack, a stack segment rule is implemented in the 80286 to ensure using the proper stack. According to this rule, the stack segment (stack addressed by the stack segment register) and the current code segment must have the identical privilege level.
- As far as the data segments are concerned, the DPL (Descriptor Privilege Level) of an accessed data segment must be less than or equal to the CPL. This rule allows protection of privileged data segments from unprivileged code.
- The 80286 is provided with a rule which pertains to accessing data segments. The 80286 can access data segments of equal or less privilege with respect to the CPL. For example, if CPL is 1, the 80286 codes in current code segment can access data segments with privilege levels of 1, 2, or 3, but not 0.
- The 80286 allows CALLing a subroutine in a code segment with higher privilege level using CALL gates technique and returns to code with lower privilege code segments. This is called the flow control rule and it protects higher privilege code segments. For example, if the CPL is 2, all code segments with level 3 can be accessed by the 80286; code segments of levels 0 or 1 cannot be accessed directly. However, higher privilege level accesses can be done directly. Also, higher privilege level accesses can be controlled by special descriptor table entries called as CALL gates. A call gate is 8 bytes wide and is stored like a descriptor in a descriptor table.

The main difference between a descriptor and a call gate is that a descriptor's contents refer to a segment in memory. On the other hand, a gate refers to a descriptor.

A descriptor includes a 24-bit physical base address, while a gate contains a 32-bit virtual address. When the effective address of an instruction CALL references a call gate, the 80286 redirects control to the destination address defined within the gate. The 22-bit virtual address (selector and offset) of the gate can be used by the 80286 to access a higher privilege code segment.

The 80286 controls the use of I/O instructions. The user may choose the level at which these I/O instructions can be used. This level is called the IOPL (Input Output Privilege Level).

IOPL is a two-bit flag whose value varies from 0 to 3. In a user/supervisor configuration in which all supervisor code is at level 0 (highest)
and all user code at lower levels, the IOPL should be 0. This zero value of IOPL allows the supervisor code to carry out I/O operations but ensures that the user code cannot execute these I/O instructions. Protection of a task from unauthorized access by another is provided by the 80286’s multitasking feature via its virtual memory capabilities. The virtual memory space consists of two spaces: global and local. The local space is unique to the present task being executed. This uniqueness of the local spaces provides intertask protection in the virtual memory space. The 80286’s limit checking feature in the physical memory space avoids illegal accesses of segments beyond the defined segment limits and thus provides protection.

The 80286 is especially designed to execute several different tasks simultaneously (appears to be simultaneous). This is called multitasking. If the present task needs to wait for some external data, the 80286 can be programmed to switch to another task until such data are available. This mechanism of switching from one task to another is called task switching. The 80286 automatically performs all the necessary steps in order to properly switch from one task to another. When a task-switching takes place, the 80286 stores the state of the present task, typically the 80286 registers, loads the state of the new task, and starts executing the new task. If execution of the outgoing task is desired after completion of the incoming one, the 80286 can automatically go back to the right place where the task-switch took place.

Task switching may occur due to hardware or software reasons. For example, task switching may take place due to 80286 external interrupt requests (hardware reason) or due to the operating system’s desire to time-share the 80286 among multiple user tasks (software reason). The task to be executed due to interrupts is termed interrupt-scheduled, while the task to be executed due to time-sharing by the operating system is called software-scheduled.

As soon as an interrupt-scheduled or a software-scheduled task is ready to be run by the 80286, it becomes the currently active (incoming) task. All inactive tasks (outgoing) have code and data segments saved in memory or disk by the 80286. Each outgoing task has a Task State Segment (TSS) associated with it. The TSS holds the task register state of an inactive task.

The TSS includes a special access right byte in its descriptor in the GDT in order that the 80286 can identify it as code or data segments. TSSs are referenced by the 16-bit selector (each task has a unique selector) that identifies a TSS descriptor in the GDT. The 80286 stores the TSS selector of the presently active task in its Task State Segment register (TR). The first 44 bytes of a TSS store the complete state of a task. Information such as selectors and 80286 registers is saved.

In summary, the 80286 provides the capability of automatically performing all steps required for task switching and task state loading/unloading. The 80286 saves the state of a task in a Task State Segment (TSS). Each task has a unique TSS and an associated descriptor. The 80286 maintains a Task Register (TR) to store the TSS selector of the presently active task.

The 80286 switches from one task to another when it executes an intersegment CALL or JMP with the destination operand addressing a TSS or task gate. The 80286 stores registers in the present TSS and then reads in the new TSS selector into the task register. The 80286 also loads registers and flags from the new TSS and identifies the new task’s TSS descriptor-type code as busy. The processor automatically traps if the segments required in the new task’s TSS are not resident in the main memory. A trap service routine can be executed to load these segments by the operating system. The 80286 sets the Nested Task (NT) bit in the flag word of the incoming task to one if the task switching is due to CALL or interrupt. The back link field in the new TSS references the previous TSS. On the other hand, if the task switching occurs due to JMP or IRET, the 80286 identifies the outgoing task’s TSS descriptor-type code as not busy, indicating that the outgoing task is not busy anymore. The new task returns to the program at the right place as defined by the CS and IP values in the TSS.

Example 4.1
Discuss the 80286’s performance impact on memory management while executing the following program:

```
MOV DS, Segment selector ; Load data selector
MOV BX, DS:eip ; Load offset
MOV CX, Count ; Load loop count
BEGIN

MOV DX, data ; Move 16-bit data to DX
CMP DX, WORD PTR [BX] ; Find match
JZ DONE
JMP BEGIN
```

```
DONE
HLT
```

Solution
The 80286 memory management capabilities are only utilized when
4.2 INTEL 80386

The 80386 is a 32-bit microprocessor and is a logical extension of the Intel 80286. The 80386 provides multitasking support, memory management, pipelined architecture, address translation caches, and a high-speed bus interface in a single chip.

The 80386 is software compatible at the object code level with the Intel 8086, 80186, and 80286. The 80386 includes separate 32-bit internal and external data paths along with eight general-purpose 32-bit registers. The processor can handle 8-, 16-, and 32-bit data types. It has separate 32-bit data and address pins and generates a 32-bit physical address. The 80386 can directly address up to four gigabytes of physical memory and 64 terabytes (2^40) of virtual memory. The 80386 can be operated from a 12.5-, 16-, 20-, 25-, or 33 MHz clock. The chip has 132 pins and is housed in a Pin Grid Array (PGA) package. The 80386 is designed using high-speed CHMOS III technology.

The 80386 is highly pipelined and can perform instruction fetching, decoding, execution, and memory management functions in parallel. The on-chip memory management and protection hardware translates logical addresses to physical addresses and provides the protection rules required in a multitasking environment. The 80386 includes special hardware for task switching. A single instruction or an interrupt is required for the 80386 to perform complete task switching. A 16-MHz 80386 can save the state of one task (all registers, segment, and paging registers if needed), and resume execution in less than 16 microseconds.

The 80386 contains a total of 129 instructions. The 80386 protection mechanism, paging, and the instructions to support them are not present in the 80286. Also, the semantics of all instructions that affect segment registers (PUSH, POP, MOV, LES LDS) and those affecting program flow (CALL, INTO, INT, IRET, JMP, RET) are quite different than the 8086 on the 80386 is protected mode.

The main differences between the 80286 and the 80386 are the 32 bit addresses and data types and paging and memory management. To provide these features and other applications, several new instructions are added in the 80386 instruction set beyond those of the 80286.

The internal architecture of the 80386 includes six functional units (Figure 4.7) that operate in parallel. The parallel operation is known as pipelined processing. Fetching, decoding, execution, memory management, and bus access for several instructions are performed simultaneously. The six functional units of the 80386 are
- Bus interface unit
- Code prefetch unit
- Execution unit
- Segmentation unit
- Paging unit
- Decode unit

The bus interface unit interfaces between the 80386 with memory and I/O. Based on internal requests for fetching instructions and transferring data from the code prefetch unit, the 80386 generates the address, data, and control signals for the current bus cycles.

The code prefetch unit prefetches instructions when the bus interface unit is not executing bus cycles. It then stores them in a 16-byte instruction queue for execution by the instruction decode unit.

The instruction decode unit translates instructions from the prefetch queue into microcodes. The decoded instructions are then stored in an instruction queue (FIFO) for processing by the execution unit.

The execution unit processes the instructions from the instruction queue. It contains a control unit, a data unit, and a protection test unit.

The control unit contains microcode and parallel hardware for fast multiply, divide, and effective address calculation.

The data unit includes an ALU, 8 general-purpose registers, and a 64-bit barrel shifter for performing multiple bit shifts in one clock. The data unit carries out data operations requested by the control unit. The protection test unit checks for segmentation violations under the control of the microcode.

The segmentation unit translates logical addresses into linear addresses at the request of the execution unit.

The translated linear address is sent to the paging unit. Upon enabling of the paging mechanism, the 80386 translates these linear addresses into physical addresses. If paging is not enabled, the physical address is identical to the linear addresses and no translation is necessary.

Figure 4.8 shows a typical 80386 system block diagram.

The 80287 or 80387 numeric coprocessor extends the instruction set of the 80386 to include instructions such as floating point operations. These instructions are executed in parallel by the 80287 or 80387 with the 80386 and thus off-load the 80386 of these functions.

The 82584 clock generator provides system clock and reset signals. The 82584 generates both the 80386 clock (CLK2) and a half-frequency clock (CLK) to drive the 80286-compatible devices that may be included in the system. It also generates the 80386 RESET signal.

The 8259A interrupt controller provides interrupt control and man-
1/0 is not included as part of the basic programming model. This is because systems designers may select to use 1/0 instructions for application programs or may select to reserve them for the operating system. Therefore, 80386 1/0 capabilities will be covered during the discussion of systems programming.

4.2.1a Memory Organization and Segmentation

The 4-gigabyte physical memory of the 80386 is structured as 8-bit bytes. Each byte can be uniquely accessed by a 32-bit address.

The programmer can write assembly language programs without knowledge of physical address space.

The memory organization model available to applications programmers is determined by the system software designers. The memory organization model available to the programmer for each task can vary between the following possibilities:

- A "flat" address space includes a single array of up to 4 gigabytes. Even though the physical address space can be up to 4 gigabytes, in reality it is much smaller. The 80386 maps the 4-gigabyte flat space into the physical address space automatically by using an address translation scheme transparent to the applications programmers.

- A segmented address space includes up to 16,383 linear address spaces of up to 4 gigabytes each. In a segmented model, the address space is divided into logical address spaces and can be up to 2^16 bytes (64 terabytes). The processor maps this address space onto the physical address space (up to 4 gigabytes by an address translation technique).

To applications programmers, the logical address space appears as up to 16,383 one-dimensional subspaces, each with a specified length. Each of these linear subspaces is called a segment. A segment is a unit of contiguous address space with sizes varying from one byte up to a maximum of 4 gigabytes.

A pointer in the logical address space consists of a 16-bit segment selector identifying a segment and a 32-bit offset addressing a byte within a segment.

4.2.1b Data Types

Data types can be byte (8-bit), word (16-bit with low byte addressed n and high byte by address n + 1), and double word (32-bit with byte 0 addressed by address n and byte 3 by address n + 3). All three data types can start at any byte address. Therefore, the words are not required to be...
aligned at even-numbered addresses and double words need not be aligned at addresses evenly divisible by 4. However, for maximum performance, data structures (including stacks) should be designed in such a way that, whenever possible, word operands are aligned at even addresses and double-word operands are aligned at addresses evenly divisible by 4.

Depending on the instruction referring to the operand, the following additional data types are available: integer (signed 8-, 16-, or 32-bit), ordinal (unsigned 8-, 16-, or 32-bit), near pointer (a 32-bit logical address which is an offset within a segment), far pointer (a 48-bit logical address consisting of a 16-bit selector and a 32-bit offset), string (8-, 16-, or 32-bit from 0 bytes to $2^{32} - 1$ bytes), bit field (a contiguous sequence of bits starting at any bit position of any byte and may contain up to 32 bits), bit string (a contiguous sequence of bits starting at any position of any byte and may contain up to $2^{20} - 1$ bits), and packed/unpacked BCD. When the 80386 is interfaced to a coprocessor such as the 80287 or 80387, then floating point numbers (signed 32-, 64-, or 80-bit real numbers) are supported.

4.2.1. e 80386 Registers

Figure 4.9 shows 80386 registers. The 80386 has 16 registers classified as general, segment, status, and instruction.

The eight general registers are the 32-bit registers EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. The low-order word of each of these eight registers has the 8086/80186/80286 register names AX (AH or AL), BX (BH or BL), CX (CH or CL), DX (DH or DL), BP, SP, SI, and DI. They are useful for making the 80386 compatible with the 8086, 80186, and 80286 processors.

The six 16-bit segment registers (CS, SS, DS, ES, FS, and GS) allow systems software designers to select either a flat or segmented model of memory organization. The purpose of CS, SS, DS, and ES is obvious. Two additional data segment registers FS and GS are included in the 80386. The four data segment registers (DS, ES, FS, GS) can access four separate data areas and allow programs to access different types of data structures. For example, one data segment register can point to the data structures of the current module, another to the exported data of a higher level module, another to dynamically created data structure, and another to data shared with another task.

The flag register is a 32-bit register named EFLAGS. Figure 4.10 shows the meaning of each bit in this register. The low-order 16 bits of EFLAGS is named FLAGS and can be treated as a unit. This is useful when executing 8086/80186/80286 code, because this part of EFLAGS is the same as the FLAGS register of the 80286/80186/8086. The 80386 flags are grouped into three types: the status flags, the control flags, and the system flags.

The status flags include CF, PF, AF, ZF, SF, and OF like the 8086/80186/80286. The control flag DF is used by strings like the 8086/80186/80286. The system flags control I/O, maskable interrupts, debugging, task switching, and enabling of virtual 8086 execution in a protected,
multitasking environment. The purpose of IF and TF is identical to the 8086/80186/80286. Let us explain the other flags:

- **IPL (I/O Privilege Level)** — This is a 2-bit field and supports the 80386 protection feature. The IPL field defines the privilege level needed to execute I/O instructions. If the present privilege level is less than or equal to IPL (privilege level is specified by numbers), the 80386 can execute I/O instructions; otherwise it takes a protection exception.

- **NT (Nested Task)** — The NT bit controls the IRET operation. If NT = 0, a usual return from interrupt is taken by the 80386 by popping EFLAGS, CS, and EIP from the stack. If NT = 1, the 80386 returns from an interrupt via task switching.

- **RF (Resume Flag)** — If RF = 1, the 80386 ignores debug faults and does not take another exception so that an instruction can be restarted after a normal debug exception. If RF = 0, the 80386 takes another debug exception to service debug faults.

- **VM (Virtual 8086 Mode)** — When VM bit is set to one, the 80386 executes 8086 programs. When VM bit is zero, the 80386 operates in the protected mode.

The RF, NT, DF, and TF can be set or reset by an 80386 program executing at any privilege level. The VM and IOPL bits can be modified by a program running at only privilege level 0 (the highest privilege level).

An 80386 with I/O privilege level can only modify the IF bit. The IRET instruction or a task switch can set or reset the RF and VM bits. The other control bits can also be modified by the POPF instruction.

The instruction pointer register (IIP) contains the offset address relative to the start of the current code segment of the next sequential instruction to be executed. The EIP is not directly accessible by the programmer; it is controlled implicitly by control-transfer instructions, interrupts, and exceptions. The low-order 16 bits of EIP is called IP and is useful when the 80386 executes 8086/80186/80286 instructions.

### 4.2.1.d Instruction Format

The 80386 instruction is formatted to include information such as operation to be performed, operand types to be manipulated, and the location of these operands. In general, all 80386 instructions are subsets of a general instruction format which includes one or two primary op code bytes followed by two bytes (an address specified for mod r/m byte and a scaled index byte), a displacement (1, 2, or 4 bytes or none) if required, and an immediate data field (1, 2, or 4 bytes or none) if required.

The op code byte field varies depending on the class of operation. This field defines information such as direction of the operation, displacement sizes, register encoding, or sign extension.

Almost all instructions referring to an operand in memory have an addressing mode byte following the primary op code byte(s). This byte (the mod r/m byte) specifies the addressing mode to be used. Certain encodings of the mod r/m byte indicate a second addressing byte. For example, the scale-index-base byte that follows the mod r/m byte fully specifies the addressing mode. The scale-index-base byte specifies the scale factor for scale index addressing mode with a general register as index register and a general register as base register.

The displacement field can be 8, 16, or 32 bits if the addressing mode includes a displacement. The last field of the instruction is the immediate data which can be 8, 16, or 32 bits if the addressing mode is immediate.

### 4.2.1.e 80386 Addressing Modes

The 80386 has 11 addressing modes which are classified into register/immediate and memory addressing modes. Register/immediate type includes two addressing modes, while the memory addressing type contains the nine modes.

1. **Register/Immediate Modes** — Instructions using these register or immediate modes operate on either register or immediate operands.
   i) **Register Mode** — The operand is contained in one of the 8, 16,
or 32-bit general registers. An example is DEC ECX which decrements the 32-bit register ECX by one.

ii) Immediate Mode — The operand is included as part of the instruction. An example is MOV EDX, 5167812FH which moves the 32-bit data 5167812FH to EDX register. Note that the source operand in this case is in immediate mode.

2. Memory Addressing Modes — The other 9 addressing modes specify the effective memory address of an operand. These modes are used when accessing memory. An 80386 address consists of two parts: a segment base address and an effective address. The effective address is computed by adding any combination of the following four components:

- Displacement: 8- or 32-bit immediate data following the instruction; 16-bit displacements can be used by inserting an address prefix before instruction.
- Base: The contents of any general-purpose register can be used as base. Compilers normally use these base registers to point to the beginning of the local variable area.
- Index: The contents of any general-purpose register except ESP can be used as an index register. The elements of an array or a string of characters can be accessed via the index register.
- Scale: The index register's contents can be multiplied (scaled) by a factor of 1, 2, 4, or 8. Scaled index mode is efficient for accessing arrays or structures.

The nine memory addressing modes are a combination of the above four elements. Of these nine modes, eight of them are executed with the same number of clock cycles, since the Effective Address calculation is pipelined with the execution of other instructions; the mode containing base, index, and displacement components requires one additional clock.

As shown in Figure 4.11, the Effective Address (EA) of an operand is computed according to the following formula:

\[ EA = Base \text{ reg} + (\text{Index Reg} \times \text{Scaling}) + \text{Displacement} \]

1. Direct Mode — The operand's effective address is included as part of the instruction as an 8-, 16-, or 32-bit displacement. An example is DEC WORD PTR [4000H].

2. Register Indirect Mode — A base or index register contains the operand's effective address. An example is MOV EBX, [ECX].

3. Based Mode — The contents of a base register are added to a displacement to obtain the operand's effective address. An example is MOV [EDX + 16], EBX.

4. Index Mode — The contents of an index register are added to a displacement to obtain the operand's effective address. An example is ADD START [EDI], EBX.

5. Scaled Index Mode — The contents of an index register are multiplied by a scaling factor (1, 2, 4, or 8) which is added to a displacement to obtain the operand's effective address. An example is MOV START [EBX + 8], ECX.

6. Based Index Mode — The contents of a base register are added to the contents of an index register to obtain the operand's effective address. An example is MOV [ECX + 8], [EAX].

7. Based Scaled Index Mode — The contents of an index register are multiplied by a scaling factor (1, 2, 4, or 8) and the result is added to the contents of a base register to determine the operand's effective address. An example is MOV [ECX * 8], [EDX], EAX.

8. Based Index Mode with Displacement — The operand's effective address is obtained by adding a displacement to the contents of a base and index register.
address is obtained by adding the contents of a base register and an index register with a displacement. An example is MOV [EBX][EBP + 0F24792AH], ECX.

9. Based Scaled Index Mode with Displacement — The contents of an index register are multiplied by a scaling factor, and the result is added to the contents of a base register and a displacement to obtain the operand's effective address. An example is MOV [ESI*8][EBP + 60H], ECX.

The 80386 can execute 8086/80186/80286 16-bit instructions in real and protected modes. This is provided in order to make the 80386 software compatible with the 80286, 80186, and the 8086. The 80386 uses the D bit in the segment descriptor register (8 bytes wide) to determine whether the instruction size is 16 or 32 bits wide. If D = 0, the 80386 uses all operand lengths and effective addresses as 16 bits long. On the other hand, if D = 1, then the default length for operands and addresses is 32 bits. Note that in the protected mode, the operating system can set or reset the D bit using proper instructions. In real mode, the default size for operands and addresses is 16 bits. Note that real address mode does not use descriptors.

Irrespective of the D-bit definition, the 80386 can execute either 16- or 32-bit instructions via the use of two override prefixes such as operand size prefix and address length prefix. These prefixes override the D bit on an individual instruction basis. These prefixes are automatically included by Intel assemblers. For example, if D = 1 and the 80386 wants to execute INC.WORD PTR [BX] to increment a 16-bit memory location, the assembler automatically adds the operand length prefix to specify only a 16-bit value.

The 80386 uses either 8- or 32-bit displacements and any register as base or index register while executing a 32-bit code. However, the 80386 uses either 8- or 16-bit displacements with the base and index registers conforming to the 80286 while executing 16-bit code. The base and index registers utilized by the 80386 for 16- and 32-bit addresses are given in the following:

<table>
<thead>
<tr>
<th>16-bit addressing</th>
<th>32-bit addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base Register</strong></td>
<td>BX, BP</td>
</tr>
<tr>
<td><strong>Index Register</strong></td>
<td>SI, DI</td>
</tr>
<tr>
<td><strong>Scale Factor</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Displacement</strong></td>
<td>0, 8, 16 bits</td>
</tr>
</tbody>
</table>

### 4.2.2 80386 INSTRUCTION SET

The 80386 extends the 8086/80186/80286 instruction set in two ways: 32-bit forms of all 16-bit instructions are included to support the 32-bit data types and 32-bit addressing modes are provided for all memory reference instructions. The 32-bit extension of the 8086/80186/80286 instruction set is accomplished by the 80386 via the default bit (D) in the code segment descriptor and by having 2 prefixes to the instruction set. The 80386 instruction set is divided into nine types:

- **Data transfer**
  - Arithmetic
  - Shift/rotate
  - String manipulation
  - Bit manipulation
  - Control transfer
  - High-level language support
  - Operating system support
  - Processor control

These instructions are listed in Table 4.1.

#### TABLE 4.1

80386 Instructions

<table>
<thead>
<tr>
<th>General purpose</th>
<th>Data Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MOV</strong></td>
<td>Move operand</td>
</tr>
<tr>
<td><strong>PUSH</strong></td>
<td>Push operand</td>
</tr>
<tr>
<td><strong>POP</strong></td>
<td>Pop operand</td>
</tr>
<tr>
<td><strong>PUSHA</strong></td>
<td>Push all registers on stack</td>
</tr>
<tr>
<td><strong>POPA</strong></td>
<td>Pop all registers off stack</td>
</tr>
<tr>
<td><strong>XCHG</strong></td>
<td>Exchange operand, register</td>
</tr>
<tr>
<td><strong>XLAT</strong></td>
<td>Translate</td>
</tr>
<tr>
<td><strong>Conversion</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MOVCX</strong></td>
<td>Move Byte or Word, Dword, with zero extension</td>
</tr>
<tr>
<td><strong>MOVZX</strong></td>
<td>Move Byte or Word, Dword, sign extended</td>
</tr>
<tr>
<td><strong>CBW</strong></td>
<td>Convert Byte to Word, or Word to Dword</td>
</tr>
<tr>
<td><strong>CDW</strong></td>
<td>Convert Word to Dword</td>
</tr>
<tr>
<td><strong>CDVE</strong></td>
<td>Convert Word to Dword extended</td>
</tr>
<tr>
<td><strong>CDQ</strong></td>
<td>Convert Dword to Qword</td>
</tr>
</tbody>
</table>
### TABLE 4.1 (continued) 80386 Instructions

<table>
<thead>
<tr>
<th>Input/output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Input operand from I/O space</td>
</tr>
<tr>
<td>OUT</td>
<td>Output operand to I/O space</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA</td>
<td>Load effective address</td>
</tr>
<tr>
<td>LDS</td>
<td>Load pointer into D segment register</td>
</tr>
<tr>
<td>LES</td>
<td>Load pointer into E segment register</td>
</tr>
<tr>
<td>LFS</td>
<td>Load pointer into F segment register</td>
</tr>
<tr>
<td>LCS</td>
<td>Load pointer into G segment register</td>
</tr>
<tr>
<td>LSS</td>
<td>Load pointer into S (stack) segment register</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flag manipulation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAHF</td>
<td>Load A register from flags</td>
</tr>
<tr>
<td>SAHF</td>
<td>Store A register in flags</td>
</tr>
<tr>
<td>PUSHF</td>
<td>Push flags onto stack</td>
</tr>
<tr>
<td>POPF</td>
<td>Pop flags off stack</td>
</tr>
<tr>
<td>PUSHPFD</td>
<td>Push EFLAGS onto stack</td>
</tr>
<tr>
<td>POPFD</td>
<td>Pop EFLAGS off stack</td>
</tr>
<tr>
<td>CLC</td>
<td>Clear carry flag</td>
</tr>
<tr>
<td>CMC</td>
<td>Clear direction flag</td>
</tr>
<tr>
<td>STC</td>
<td>Complement carry flag</td>
</tr>
<tr>
<td>STD</td>
<td>Set direction flag</td>
</tr>
</tbody>
</table>

### Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add operand</td>
</tr>
<tr>
<td>ADC</td>
<td>Add with carry</td>
</tr>
<tr>
<td>INC</td>
<td>Increment operand by 1</td>
</tr>
<tr>
<td>AAA</td>
<td>ASCII adjust for addition</td>
</tr>
<tr>
<td>DAA</td>
<td>Decimal adjust for addition</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB</td>
<td>Subtract operand</td>
</tr>
<tr>
<td>SBB</td>
<td>Subtract with borrow</td>
</tr>
<tr>
<td>DEC</td>
<td>Decrement operand by 1</td>
</tr>
<tr>
<td>NEG</td>
<td>Negate operand</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare operands</td>
</tr>
<tr>
<td>AAS</td>
<td>ASCII adjust for subtraction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multiplication</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>Multiply double/single precision</td>
</tr>
<tr>
<td>IMUL</td>
<td>Integer multiply</td>
</tr>
<tr>
<td>AAA</td>
<td>ASCII adjust after multiply</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Division</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIV</td>
<td>Divide unsigned</td>
</tr>
<tr>
<td>IDIV</td>
<td>Integer divide</td>
</tr>
<tr>
<td>AAD</td>
<td>ASCII adjust after division</td>
</tr>
</tbody>
</table>

### String Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVSB</td>
<td>Move Byte or Word, Dword string</td>
</tr>
<tr>
<td>INSB</td>
<td>Input string from I/O space</td>
</tr>
<tr>
<td>OUTSB</td>
<td>Output string to I/O space</td>
</tr>
<tr>
<td>SCAS</td>
<td>Scan Byte or Word, Dword string</td>
</tr>
<tr>
<td>LODSB</td>
<td>Load Byte or Word, Dword string</td>
</tr>
<tr>
<td>STOS</td>
<td>Store Byte or Word, Dword string</td>
</tr>
<tr>
<td>REP</td>
<td>Repeat</td>
</tr>
<tr>
<td>REPE/REPZ</td>
<td>Repeat while equal/zero</td>
</tr>
<tr>
<td>RENZ/REPNZ</td>
<td>Repeat while not equal/not zero</td>
</tr>
</tbody>
</table>

### Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>“NOT” operand</td>
</tr>
<tr>
<td>AND</td>
<td>“AND” operand</td>
</tr>
<tr>
<td>OR</td>
<td>“Inclusive OR” operand</td>
</tr>
<tr>
<td>XOR</td>
<td>“Exclusive OR” operand</td>
</tr>
<tr>
<td>TEST</td>
<td>“Test” operand</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHL/SHR</td>
<td>Shift logical left or right</td>
</tr>
<tr>
<td>SAUL/SAR</td>
<td>Shift arithmetic left or right</td>
</tr>
<tr>
<td>SHLD/SHRD</td>
<td>Double shift left or right</td>
</tr>
<tr>
<td>Rotates</td>
<td>Rotate left/right</td>
</tr>
<tr>
<td>ROL/ROR</td>
<td>Rotate through carry left/right</td>
</tr>
</tbody>
</table>
### TABLE 4.1 (continued)

#### 80386 Instructions

**Bit Manipulation Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>Bit test</td>
</tr>
<tr>
<td>BTS</td>
<td>Bit test and set</td>
</tr>
<tr>
<td>BTR</td>
<td>Bit test and reset</td>
</tr>
<tr>
<td>BTC</td>
<td>Bit test and complement</td>
</tr>
<tr>
<td>BSF</td>
<td>Bit scan forward</td>
</tr>
<tr>
<td>BSR</td>
<td>Bit scan reverse</td>
</tr>
<tr>
<td>IBTS</td>
<td>Insert bit string</td>
</tr>
<tr>
<td>XBTS</td>
<td>Exact bit string</td>
</tr>
</tbody>
</table>

**Program Control Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETCC</td>
<td>Set byte equal to condition code</td>
</tr>
<tr>
<td>JA/NB</td>
<td>Jump if above/not below nor equal</td>
</tr>
<tr>
<td>JAE/NB</td>
<td>Jump if above or equal/not below</td>
</tr>
<tr>
<td>JB/NAE</td>
<td>Jump if below/not above nor equal</td>
</tr>
<tr>
<td>JBE/NA</td>
<td>Jump if below or equal/not above</td>
</tr>
<tr>
<td>JC</td>
<td>Jump if carry</td>
</tr>
<tr>
<td>JE/JZ</td>
<td>Jump if equal/zero</td>
</tr>
<tr>
<td>JCJNE/L</td>
<td>Jump if greater/not less nor equal</td>
</tr>
<tr>
<td>JCE/JNL</td>
<td>Jump if greater or equal/not less</td>
</tr>
<tr>
<td>JCL/JNGE</td>
<td>Jump if less/not greater nor equal</td>
</tr>
<tr>
<td>JLE/JNG</td>
<td>Jump if less or equal/not greater</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if not carry</td>
</tr>
<tr>
<td>JNE/JNZ</td>
<td>Jump if not equal/not zero</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if not overflow</td>
</tr>
<tr>
<td>JNP/JPO</td>
<td>Jump if not parity/parity odd</td>
</tr>
<tr>
<td>JNS</td>
<td>Jump if not sign</td>
</tr>
<tr>
<td>JO</td>
<td>Jump if overflow</td>
</tr>
<tr>
<td>JPE/JPOE</td>
<td>Jump if parity/parity even</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if sign</td>
</tr>
</tbody>
</table>

**Conditional transfers**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>Call procedure/task</td>
</tr>
<tr>
<td>RET</td>
<td>Return from procedure/task</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
</tr>
</tbody>
</table>

**Unconditional transfers**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>Loop</td>
</tr>
<tr>
<td>LOOPZ</td>
<td>Loop if equal/zero</td>
</tr>
<tr>
<td>LOOPNZ</td>
<td>Loop if not equal/not zero</td>
</tr>
<tr>
<td>JCXZ</td>
<td>JUMP if register CX = 0</td>
</tr>
</tbody>
</table>

**Interrupts**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>Interrupt</td>
</tr>
<tr>
<td>INTO</td>
<td>Interrupt if overflow</td>
</tr>
<tr>
<td>IRET</td>
<td>Return from interrupt</td>
</tr>
<tr>
<td>CLI</td>
<td>Clear interrupt enable</td>
</tr>
<tr>
<td>SSI</td>
<td>Set interrupt enable</td>
</tr>
</tbody>
</table>

**High Level Language Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOUND</td>
<td>Check array bounds</td>
</tr>
<tr>
<td>ENTER</td>
<td>Setup parameter block for entering procedure</td>
</tr>
<tr>
<td>LEAVE</td>
<td>Leave procedure</td>
</tr>
</tbody>
</table>

**Protection Model**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCDT</td>
<td>Store global descriptor table</td>
</tr>
<tr>
<td>SIE</td>
<td>Store interrupt descriptor table</td>
</tr>
<tr>
<td>STR</td>
<td>Store task register</td>
</tr>
<tr>
<td>SLDT</td>
<td>Store local descriptor table</td>
</tr>
<tr>
<td>LGDT</td>
<td>Load global descriptor table</td>
</tr>
<tr>
<td>LIDT</td>
<td>Load interrupt descriptor table</td>
</tr>
<tr>
<td>LTR</td>
<td>Load task register</td>
</tr>
<tr>
<td>LLDT</td>
<td>Load local descriptor table</td>
</tr>
<tr>
<td>ARPL</td>
<td>Adjust requested privilege level</td>
</tr>
<tr>
<td>LAR</td>
<td>Load access rights</td>
</tr>
<tr>
<td>LSL</td>
<td>Load segment limit</td>
</tr>
<tr>
<td>VERR/VERW</td>
<td>Verify segment for reading or writing</td>
</tr>
<tr>
<td>LMSW</td>
<td>Load machine status word (lower 16 bits of CR0)</td>
</tr>
<tr>
<td>SMSW</td>
<td>Store machine status word</td>
</tr>
</tbody>
</table>

**Processor Control Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLT</td>
<td>Halt</td>
</tr>
<tr>
<td>WAIT</td>
<td>Wait until BUSY # negated</td>
</tr>
<tr>
<td>ESC</td>
<td>Escape</td>
</tr>
<tr>
<td>LOCK</td>
<td>Lock bus</td>
</tr>
</tbody>
</table>
The 80386 instructions include zero-operand, single-operand, two-operand, and three-operand instructions. Most zero-operand instructions such as STC occupy only one byte. Single operand instructions are usually two bytes wide. The two-operand instructions usually allow the following types of operations:

- Register-to-register
- Memory-to-register
- Immediate-to-register
- Memory-to-memory
- Register-to-memory
- Immediate-to-memory

The operands can be either 8, 16, or 32 bits wide. In general, operands are 8 or 32 bits long when the 80386 executes the 32-bit code. On the other hand, operands are 8 or 16 bits wide when the 80386 executes the existing 80286 or 8086 code (16-bit code). Prefixes can be added to all instructions which override the default length of the operands. That is, 32-bit operands for 16-bit code or 16-bit operands for 32-bit code can be used.

The 80386 various instructions affecting the status flags are summarized in Table 4.2.

### TABLE 4.2

**Status Flag Summary**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CF</td>
<td>Carry flag — Set on high-order bit carry or borrow; cleared otherwise</td>
</tr>
<tr>
<td>2</td>
<td>PF</td>
<td>Parity flag — Set if low-order eight bits of result contain an even number of 1 bits; cleared otherwise</td>
</tr>
<tr>
<td>4</td>
<td>AF</td>
<td>Adjust flag — Set on carry from or borrow to the low-order four bits of AL; cleared otherwise; used for decimal arithmetic</td>
</tr>
<tr>
<td>6</td>
<td>ZF</td>
<td>Zero flag — Set if result is zero; cleared otherwise</td>
</tr>
<tr>
<td>7</td>
<td>SF</td>
<td>Sign flag — Set equal to high-order bit of result (0 is positive, 1 if negative)</td>
</tr>
<tr>
<td>11</td>
<td>OF</td>
<td>Overflow flag — Set if result is too large a positive number or too small a negative number (excluding sign-bit) to fit in destination operand; cleared otherwise</td>
</tr>
</tbody>
</table>

### TABLE 4.2 (continued)

#### Key to Codes

- **T** Instruction tests flag
- **M** Instruction modifies flag (either sets or resets depending on operands)
- **0** Instruction resets flag
- **—** Instruction's effect on flag is undefined
- **Blank** Instruction does not affect flag

#### Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>OF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>TM</td>
<td>—</td>
<td>M</td>
</tr>
<tr>
<td>AAS</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>TM</td>
<td>—</td>
<td>M</td>
</tr>
<tr>
<td>AAD</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AAM</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>TM</td>
<td>M</td>
<td>—</td>
</tr>
<tr>
<td>DAA</td>
<td>M</td>
<td>M</td>
<td>TM</td>
<td>M</td>
<td>TM</td>
<td>—</td>
</tr>
<tr>
<td>DAS</td>
<td>M</td>
<td>M</td>
<td>TM</td>
<td>M</td>
<td>TM</td>
<td>—</td>
</tr>
<tr>
<td>ADC</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>TM</td>
</tr>
<tr>
<td>ADD</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>SBB</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>—</td>
</tr>
<tr>
<td>SUB</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>—</td>
</tr>
<tr>
<td>CMP</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>—</td>
</tr>
<tr>
<td>CMPS</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>—</td>
</tr>
<tr>
<td>SCAS</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>—</td>
</tr>
<tr>
<td>NEG</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>—</td>
</tr>
<tr>
<td>DEC</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>—</td>
</tr>
<tr>
<td>INC</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>—</td>
</tr>
<tr>
<td>IMUL</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>—</td>
</tr>
<tr>
<td>MUL</td>
<td>M</td>
<td>M</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

- **RCL**/**RCR**
- **ROL**/**ROR**
- **SAL**/**SHL**/**SHR**
- **SALS**/**SAR**/**SHR**
- **SHLD**/**SHRD**
- **BSF**/**BSR**
- **BT**/**BTS**/**BTR**
- **AND**
- **OR**
- **TEST**
- **XOR**

132.jpg
Table 4.3 lists the various conditions referring to the relation between two numbers (signed and unsigned) for Jcond and SETcond instructions.

### TABLE 4.3
Condition Codes
(For Conditional Instructions Jcond, and SETcond)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Instruction subcode</th>
<th>Condition tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>Overflow</td>
<td>0000</td>
<td>OF = 1</td>
</tr>
<tr>
<td>NO</td>
<td>No overflow</td>
<td>0001</td>
<td>OF = 0</td>
</tr>
<tr>
<td>B</td>
<td>Below</td>
<td>0010</td>
<td>CF = 1</td>
</tr>
<tr>
<td>NAE</td>
<td>Neither above nor equal</td>
<td>0011</td>
<td>CF = 0</td>
</tr>
<tr>
<td>NB</td>
<td>Not below</td>
<td>0100</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>AE</td>
<td>Above or equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Equal</td>
<td>0101</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>Z</td>
<td>Zero</td>
<td>0110</td>
<td>(CF or ZF) = 1</td>
</tr>
<tr>
<td>NE</td>
<td>Not equal</td>
<td>0111</td>
<td>(CF or ZF) = 0</td>
</tr>
<tr>
<td>NZ</td>
<td>Not zero</td>
<td>1000</td>
<td>SF = 1</td>
</tr>
<tr>
<td>BE</td>
<td>Below or equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NA</td>
<td>Not above</td>
<td>1001</td>
<td>SF = 0</td>
</tr>
<tr>
<td>NBE</td>
<td>Neither below nor equal</td>
<td>1010</td>
<td>PF = 1</td>
</tr>
<tr>
<td>A</td>
<td>Above</td>
<td>1011</td>
<td>PF = 0</td>
</tr>
<tr>
<td>NS</td>
<td>Sign</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>Parity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE</td>
<td>Parity even</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP</td>
<td>No parity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>Parity odd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>Less</td>
<td>1100</td>
<td>(SF or OF) = 1</td>
</tr>
<tr>
<td>NGE</td>
<td>Neither greater nor equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NL</td>
<td>Not less</td>
<td>1101</td>
<td>(SF or OF) = 0</td>
</tr>
<tr>
<td>GE</td>
<td>Greater or equal</td>
<td>1110</td>
<td>(SF or OF) or ZF = 1</td>
</tr>
<tr>
<td>LE</td>
<td>Less or equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NG</td>
<td>Not greater</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The terms “above” and “below” refer to the relation between two unsigned values (neither SF nor OF is tested). The terms “greater” and “less” refer to the relation between two signed values SF and OF are tested.

All new 80386 instructions along with those which have minor variations from the 80286 are listed in alphabetical order below in Table 4.4.

### TABLE 4.4
80386 Instructions (New Instructions beyond Those of 8086/80186/80286)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC EAX, imm32</td>
<td>Add CF with sign-extended immediate byte and 32-bit data in reg32 or mem32</td>
</tr>
<tr>
<td>ADC reg32/mem32, imm32</td>
<td></td>
</tr>
<tr>
<td>ADC reg32/mem32, imm8</td>
<td></td>
</tr>
<tr>
<td>ADD reg32/mem32, imm32</td>
<td></td>
</tr>
<tr>
<td>ADD reg32/mem32, imm8</td>
<td></td>
</tr>
<tr>
<td>ADD reg32/mem32, reg32</td>
<td></td>
</tr>
<tr>
<td>ADD EAX, imm32</td>
<td></td>
</tr>
<tr>
<td>ADD reg32/mem32, reg32</td>
<td></td>
</tr>
<tr>
<td>ADD reg32, reg32/mem32</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 4.4 (continued)
80386 Instructions (New Instructions beyond Those of 8086/80186/80286)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AND EAX, imm32</strong></td>
<td></td>
</tr>
<tr>
<td><strong>AND reg32/mem32, imm32</strong></td>
<td></td>
</tr>
<tr>
<td><strong>AND reg32/mem32, imm8</strong></td>
<td></td>
</tr>
<tr>
<td><strong>AND reg32/mem32, reg32</strong></td>
<td></td>
</tr>
<tr>
<td><strong>AND reg32, reg32/mem32</strong></td>
<td></td>
</tr>
<tr>
<td><strong>BOUND reg32, mem64</strong></td>
<td>Check if reg32 is within bounds specified in mem64; the first 32 bits of mem64 contain the lower bound and the second 32 bits contain the upper bound.</td>
</tr>
<tr>
<td><strong>BSF</strong></td>
<td>Bit scan forward</td>
</tr>
<tr>
<td><strong>BSR</strong></td>
<td>Bit scan reverse</td>
</tr>
<tr>
<td><strong>BT</strong></td>
<td>Bit test</td>
</tr>
<tr>
<td><strong>BTC</strong></td>
<td>Bit test and complement</td>
</tr>
<tr>
<td><strong>BTR</strong></td>
<td>Bit test and reset</td>
</tr>
<tr>
<td><strong>BTS</strong></td>
<td>Bit test and set</td>
</tr>
<tr>
<td><strong>CALL label</strong></td>
<td>There are several variations of the label such as disp32, disp16, reg16, reg32, mem16, mem32, ptr16:16, ptr16:32, mem16:16, and mem16:32; NEAR CALLS use reg16/mem16, reg32/mem32, and disp16/disp32 and the CALL is the same segment with CS unchanged; CALL disp16 or disp32 adds a signed 16- or 32-bit offset to the address of the next instruction (current EIP) and the result is stored in EIP; when disp16 is used, the upper 16 bits of EIP are cleared to zero; CALL reg16/mem16 or reg32/mem32 specifies a register or memory location from which the offset is obtained; near-return instruction should be used for these.</td>
</tr>
<tr>
<td><strong>CDQ</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CMP reg32/mem32, imm32</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CMP EAX, imm32</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CMP reg32/mem32, imm8</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CMP reg32/mem32, reg32</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CMP reg32, reg32/mem32</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CMPS mem32, mem31</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CWDE</strong></td>
<td></td>
</tr>
<tr>
<td><strong>DEC reg32/mem32</strong></td>
<td></td>
</tr>
<tr>
<td><strong>DIV EAX, reg32/mem32</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ENTER imm16, imm8</strong></td>
<td></td>
</tr>
<tr>
<td><strong>EAX ← sign extend AX</strong></td>
<td></td>
</tr>
<tr>
<td><strong>EDX:EAX ← sign extend EAX</strong></td>
<td></td>
</tr>
<tr>
<td><strong>UNSIGNED divide EDX:EAX by reg32 or mem32 (EAX = quotient, EDX = remainder)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Create a stack frame before entering a procedure</strong></td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 4.4 (continued)
#### 80386 Instructions (New Instructions beyond Those of 8086/80186/80286)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV reg32, DR0/DR1/DR2/DR3</td>
<td>DRs are debug registers</td>
</tr>
<tr>
<td>MOV reg32, DR4/DR7</td>
<td></td>
</tr>
<tr>
<td>MOV DR0/DR1/DR2/DR3, reg32</td>
<td></td>
</tr>
<tr>
<td>MOV DR4/DR7, reg32</td>
<td></td>
</tr>
<tr>
<td>MOV reg32, TR6/TR7</td>
<td>TRs are test registers</td>
</tr>
<tr>
<td>MOV TR6/TR7, reg32</td>
<td></td>
</tr>
<tr>
<td>MOVS mem32, mem32</td>
<td>To be explained later</td>
</tr>
<tr>
<td>or MOVSD</td>
<td></td>
</tr>
<tr>
<td>MOVSX</td>
<td>Move with sign extend; to be discussed later</td>
</tr>
<tr>
<td>MOVZX</td>
<td>Move with zero extend; to be discussed later</td>
</tr>
<tr>
<td>MUL EAX, reg32/mem32</td>
<td>Unsigned multiply; EDX: EAX ← EAX * reg32 or mem32</td>
</tr>
<tr>
<td>NEG reg32/mem32</td>
<td>Two's complement negate 32 bits in reg32 or mem32</td>
</tr>
<tr>
<td>NOT reg32/mem32</td>
<td>Ones complement</td>
</tr>
<tr>
<td>OR d,s</td>
<td>The definitions for d and s are same as AND</td>
</tr>
<tr>
<td>OUT imm8, EAX</td>
<td>Output 32-bit EAX to immediate port number</td>
</tr>
<tr>
<td>OUT DX, reg32/mem32</td>
<td>To be explained later</td>
</tr>
<tr>
<td>or OUTSD</td>
<td></td>
</tr>
<tr>
<td>POP reg32</td>
<td>To be explained later</td>
</tr>
<tr>
<td>POP mem32</td>
<td>To be explained later</td>
</tr>
<tr>
<td>POP FS</td>
<td>To be explained later</td>
</tr>
<tr>
<td>POP GS</td>
<td>To be explained later</td>
</tr>
<tr>
<td>POPAD</td>
<td>To be explained later</td>
</tr>
<tr>
<td>POPOFD</td>
<td>To be explained later</td>
</tr>
<tr>
<td>PUSH reg32</td>
<td>To be explained later</td>
</tr>
<tr>
<td>PUSH mem32</td>
<td>To be explained later</td>
</tr>
<tr>
<td>PUSH F5</td>
<td>To be explained later</td>
</tr>
<tr>
<td>PUSH GS</td>
<td>To be explained later</td>
</tr>
<tr>
<td>PUSHAO</td>
<td>To be explained later</td>
</tr>
<tr>
<td>RCL reg32/mem32, 1</td>
<td>Rotate reg32 or mem32 thru CF once to left</td>
</tr>
<tr>
<td>RCL reg32/mem32, CL</td>
<td>Rotate reg32 or mem32 thru CF to left CL times</td>
</tr>
<tr>
<td>RCL reg32/mem32, imm8</td>
<td>d and n have same definitions as RCL/RCR/ROR/ROR</td>
</tr>
<tr>
<td>RCR reg32/mem32, 1</td>
<td>To be explained later</td>
</tr>
<tr>
<td>RCR reg32/mem32, CL</td>
<td>To be explained later</td>
</tr>
<tr>
<td>ROL reg32/mem32, 1</td>
<td>To be explained later</td>
</tr>
<tr>
<td>ROL reg32/mem32, CL</td>
<td>To be explained later</td>
</tr>
<tr>
<td>ROR reg32/mem32, 1</td>
<td>To be explained later</td>
</tr>
<tr>
<td>ROR reg32/mem32, CL</td>
<td>To be explained later</td>
</tr>
<tr>
<td>ROR reg32/mem32, imm8</td>
<td>d and s have same definitions as ADD d,s</td>
</tr>
<tr>
<td>SAL/SAR/SHL/SHR d, n</td>
<td>To be explained later</td>
</tr>
<tr>
<td>SBB d,s</td>
<td>To be explained later</td>
</tr>
<tr>
<td>SCAS mem32 or SCASD</td>
<td>To be explained later</td>
</tr>
<tr>
<td>SET cc</td>
<td>To be explained later</td>
</tr>
<tr>
<td>SHLD</td>
<td>To be explained later</td>
</tr>
<tr>
<td>SHRD</td>
<td>To be explained later</td>
</tr>
<tr>
<td>STOS mem32 or STOSD</td>
<td>To be explained later</td>
</tr>
<tr>
<td>SUB d,s</td>
<td>d and s have same definitions as ADD</td>
</tr>
<tr>
<td>TEST EAX, imm32</td>
<td>Exchange 32-bit register contents with EAX</td>
</tr>
<tr>
<td>TEST reg32/mem32, imm32</td>
<td></td>
</tr>
<tr>
<td>TEST reg32/mem32, reg32</td>
<td></td>
</tr>
<tr>
<td>XCHG reg32, EAX</td>
<td></td>
</tr>
<tr>
<td>XCHG EAX, reg32</td>
<td></td>
</tr>
<tr>
<td>XCHG reg32/mem32, reg32</td>
<td></td>
</tr>
<tr>
<td>XCHG reg32, reg32/mem32</td>
<td></td>
</tr>
<tr>
<td>XLATB</td>
<td></td>
</tr>
<tr>
<td>XOR d,s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d and s have same definitions as AND</td>
</tr>
</tbody>
</table>
A detailed description of most of the new 80386 instructions is given in the following.

4.2.2.a Arithmetic Instructions
There are two new instructions beyond those of 80286. These are CWDE and CDQ. CWDE sign extends the 16-bit contents of AX to a 32-bit doubleword in EAX. CDQ instruction sign extends a doubleword (32 bits) in EAX to a quadword (64 bits) in EDX: EAX.

4.2.2.b Bit Instructions
The following lists the 80386 six-bit instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSF</td>
<td>Bit scan forward</td>
</tr>
<tr>
<td>BSR</td>
<td>Bit scan reverse</td>
</tr>
<tr>
<td>BT</td>
<td>Bit test</td>
</tr>
<tr>
<td>BTC</td>
<td>Bit test and complement</td>
</tr>
<tr>
<td>BTZ</td>
<td>Bit test and reset</td>
</tr>
<tr>
<td>BTS</td>
<td>Bit test and set</td>
</tr>
</tbody>
</table>

The above instructions are discussed in the following:

* BSF (Bit Scan Forward)

BSF d, s
reg16, reg16
reg16, mem16
reg32, reg32
reg32, mem32

The 16-bit (word) or 32-bit (doubleword) number defined by s is scanned (checked) from right to left (bit 0 to bit 15 or bit 31). The bit number of the first one found is stored in d. If the whole 16-bit or 32-bit number is zero, the zero flag is set to one; if a one is found, the zero flag is reset to zero. For example, consider BSF EBX, EDX. If [EDX] = 01121240, then [EBX] = 000000006, and ZF = 0. This is because the bit number 6 in EDX (contained in second nibble of EDX) is the first one when EDX is scanned from the right.

* BSR (Bit Scan Reverse)

BSR d, s
reg16, reg16
reg16, mem16
reg32, reg32
reg32, mem32

BSR scans or checks a 16-bit or 32-bit number specified by s from the most significant bit (bit 15 or bit 31) to the least significant bit (bit 0). The destination operand d is loaded with the bit index (bit number) of the first set bit. If the bits in the number are all zero, the ZF is set to one and operand d is undefined; ZF is reset to zero if a one is found.

* BT (Bit Test)

BT d, s
reg16, reg16
mem16, reg16
reg16, imm8
mem16, imm8
reg32, reg32
mem32, reg32
reg32, imm8
mem32, imm8

BT assigns the bit value of operand d (base) specified by operand s (the bit offset) to the carry flag. Only the CF is affected. If operand s is an immediate data, only eight bits are allowed in the instruction. This operand is taken modulo 32, so the range of immediate bit offset is from 0 to 31. This permits any bit within a register to be selected. If d is a register, the bit value assigned to CF is defined by the value of the bit number defined by s taken modulo the register size (16 or 32). If d is a memory bit string, the desired 16-bit or 32-bit can be determined by adding s (bit index) divided by operand size (16 or 32) to the memory address of d. The bit within this 16- or 32-bit word is defined by d modulo the operand size (16 or 32). If d is a memory operand, the 80386 may access four bytes in memory starting at...
Effective address = \(4^* \text{[bit offset divided by 32]}\). As an example, consider \(\text{BT} \ CX, \ DX\). \(\text{IF} [\text{CX}] = 081h, [\text{DX}] = 0021h\), then since the content of \(\text{DX}\) is \(33_4\), the bit number one (remainder of \(33_{16} = 1\)) of \(\text{CX}\) (value 1) is reflected in the \(\text{CF}\) and therefore \(\text{CF} = 1\).

- **BTC** (Bit Test and Complement)

\[
\text{BTC \ d, s}
\]

\(d\) and \(s\) have the same definitions as the BT instruction. The bit of \(d\) defined by \(s\) is reflected in \(CF\). After \(CF\) is assigned, the same bit of \(d\) defined by \(s\) is ones complemented. The 80880 determines the bit number from \(s\) (whether \(s\) is immediate data or register) and \(d\) (whether \(d\) is register or memory bit-string) in the same way as the BT instruction.

- **BTR** (Bit Test and Reset)

\[
\text{BTR \ d, s}
\]

\(d\) and \(s\) have the same definitions as for the BT instruction. The bit of \(d\) defined by \(s\) is reflected in \(CF\). After \(CF\) is assigned, the same bit of \(d\) defined by \(s\) is reset to zero. Everything else that is applicable to the BT instruction also applies to BTR.

- **BTS** (Bit Test and Set)

\[
\text{BTS \ d, s}
\]

Same as BTR except the specified bit in \(d\) is set to one after the bit value of \(d\) defined by \(s\) is reflected into \(CF\). Everything else applicable to the BT instruction also applies to BTS.

### 4.2.2.c. Byte Set-On Condition Instructions

These instructions set a byte to one or reset a byte to zero depending on any of the 16 conditions defined by the status flags. The byte may be located in memory or in a one-byte general register. These instructions are very useful in implementing Boolean expressions in high level languages such as Pascal. The general structure of this instruction is SET\(_\text{cc}\) (set byte on condition \(cc\)) which sets a byte to one if condition \(cc\) is true; or else, reset the byte to zero. The following is a list of these instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETA/SETHBE reg8/mem8</td>
<td>CF = 0 and ZF = 0</td>
<td>Set byte if above or not equal</td>
</tr>
<tr>
<td>SETAE/SETHB/SETHC reg8/mem8</td>
<td>CF = 0</td>
<td>Set if above or equal, set if not below, set if not carry</td>
</tr>
<tr>
<td>SETNE/SETHL reg8/mem8</td>
<td>CF = 1</td>
<td>Set if below, set if not above, or set if not carry</td>
</tr>
<tr>
<td>SETBE/SETHNA reg8/mem8</td>
<td>CF = 1</td>
<td>Set if below/equal or set if not above</td>
</tr>
<tr>
<td>SETE/SETHZ reg8/mem8</td>
<td>ZF = 1</td>
<td>Set if equal or set if zero</td>
</tr>
<tr>
<td>SETG/SETHNLE reg8/mem8</td>
<td>ZF = 0 or SF = OF</td>
<td>Set if greater or set if not less/equal</td>
</tr>
<tr>
<td>SETL/SETHNL reg8/mem8</td>
<td>SF = OF</td>
<td>Set if greater/equal or set if not less</td>
</tr>
<tr>
<td>SETG0/SETHNGE reg8/mem8</td>
<td>SF # OF</td>
<td>Set if less or set if not greater/equal</td>
</tr>
<tr>
<td>SETL0/SETHNG reg8/mem8</td>
<td>ZF = 1 and SF # OF</td>
<td>Set if less/equal or set if not greater</td>
</tr>
<tr>
<td>SETNE/SETHNZ reg8/mem8</td>
<td>ZF = 0</td>
<td>Set if not equal or set if not zero</td>
</tr>
<tr>
<td>SETNO reg8/mem8</td>
<td>OF = 0</td>
<td>Set if no overflow</td>
</tr>
<tr>
<td>SETNP/SETPO reg8/mem8</td>
<td>PF = 0</td>
<td>Set if parity or set if parity odd</td>
</tr>
</tbody>
</table>
| SETS reg8/mem8 | SF # OF | Set if not sign |}

As an example, consider SETB BL. If [BL] = 52\(_8\), and CF = 1, then after this instruction is executed [BL] = 01\(_8\), and CF remains at 1; all other flags (OF, SF, ZF, AF, PF) are undefined. On the other hand, if CF = 0, then after execution of SETB BL, BL contains 00\(_8\), CF = 0 and ZF = 1; all other flags are undefined. Similarly, the other SET\(_\text{cc}\) instructions can be explained.

### 4.2.2.d. Conditional Jumps and Loops

JE CXZ disp18 jumps if ECX is zero. disp8 means a relative address
range from 128 bytes before the end of the instruction. JECXZ tests the contents of ECX register for zero and not the flags. If ECX = 0, then after execution of JECXZ instruction, the program branches with signed 8-bit relative offset (+127 to -128, with 0 being positive) defined by disp8.

JECXZ instruction is useful at the beginning of a conditional loop that terminates with a conditional loop instruction such as LOOPNPE label. The JECXZ prevents entering the loop with ECX = 0, which would cause the loop to execute up to 2^12 times instead of zero times.

**LOOP Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP disp8</td>
<td>Decrement CV/ECX by one and jump if CV/ECX # 0</td>
</tr>
<tr>
<td>LOOPNZ disp8</td>
<td>Decrement CV/ECX by one and jump if CV/ECX # 0 and ZF = 1</td>
</tr>
<tr>
<td>LOOPNZ/LOOPNE disp8</td>
<td>Decrement CV/ECX by one and jump if CV/ECX # 0 and ZF = 0</td>
</tr>
</tbody>
</table>

The 80386 LOOP instructions are similar to those of 8086/80186/80286, except that if the counter is more than 16 bits, ECX register is used as the counter.

### 4.2.2.6 Data Transfer

- **Move instructions description**

  - **MOVX d, s**  Move and sign extend
  - **MOVX d, s**  Move and zero extend

  The d and s operands are defined as follows:

  - **MOVX d, s**
    - reg16, reg8
    - reg16, mem8
    - reg16, mem32
    - reg32, mem8
    - reg32, reg16
    - reg32, mem16
  - **MOVX d, s**
    - reg16, mem64
    - reg32, mem64

  MOVX reads the contents of the effective address or register as a byte or a word from the source and sign-extends the value to the operand size of the destination (16 or 32 bits) and stores the result in the destination. No flags are affected. MOVZX, on the other hand, reads the contents of the effective address or register as a byte or a word and zero-extends the value to the operand size of the destination (16 or 32 bits) and stores the result in the destination. No flags are affected. For example, consider MOVZX BX, CL. If CL = 81H, and [BX] = 21AFH, then after execution of MOVZX BX, CL, register BX will contain FB1H, and CL contents do not change. Also, consider MOVZ BX, DH. If CX = F237H, and [DH] = 85H, then after execution of this MOVZ, CX register will contain 0003H and DH contents do not change.

- **PUSHAD and POPAD Instructions** — There are two new PUSH and POP instructions in the 80386 beyond those of 80286. These are PUSHAD and POPAD. PUSHAD saves all 32-bit general registers (the order is EAX, ECX, EDX, EBX, original ESP, EBP, ESP, EDI, and EDX) onto the 80386 stack. PUSHAD decrements the stack pointer (ESP) by 32H to hold the eight 16-bit values. No flags are affected. POPAD reverses a previous PUSHAD. It pops the eight 32-bit registers (the order is EDI, ESI, ESP, EBP, ESP, [BX, EDX], EAX, and EAX). The ESP value is discarded instead of loading onto ESP. No flags are affected. Note that ESP is actually popped but thrown away, so that [ESP] after popping all the registers, will be incremented by 32H.

- **Load Pointer Instruction** — There are five instructions in this category. These are LDS, LES, LFS, LGS, and LSS. The first two instructions LDS and LES are available in the 80286. However, the 80286 loads 32 bits from a specified location (16-bit offset and DS) into a specified 16-bit register such as BX and the other into DS for LDS or ES for LES. The 80386, on the other hand, can have four versions of these instructions as follows:

  - **LDS reg16, mem16: mem16**
  - **LDS reg32, mem16: mem32**
  - **LES reg16, mem16: mem16**
  - **LES reg32, mem16: mem32**

  Note that mem16 or mem32 defines a memory op-
4.2.2.2 Flag Control

There are two new 80386 instructions beyond those of the 80286. These are SHLD and SHRD. SHLD shifts the specified shift count to the contents of ds with the result stored back into d; d is shifted to the left by the shift count with the low-order bits of d being filled from the high-order bits of s. The bits in s are not altered after shifting. The carry flag becomes the value of the bit shifted out of the most significant bit of d.

If the shift count is zero, the instruction works as a NOP. For a specified shift count, the SF, ZF, and PF flags are set according to the result in d. CF is set to the value of the last bit shifted out. OF and AF are undefined.

SHRD, on the other hand, shifts the contents of ds by the specified shift count to the right with the result being stored back into d. The bits in d are shifted right by the shift count with the high-order bits being filled from the low-order bits of s. The bits in s are not altered after shifting.

If the shift count is zero, this instruction operates as a NOP. For the specified shift count, the SF, ZF, and PF flags are set according to the value of the result. CF is set to the value of the past bit shifted out. OF and AF are undefined.

As an example, consider SHLD BX, DX, 2. If [BX] = 183Fh, [DX] = 01F1h, then after this SHLD, [BX] = 600FCh, [DX] = 01F1h, CF = 0, SF = 0, ZF = 0, and PF = 1.

Similarly, the SHRD instruction can be illustrated.

4.2.2.2 String

- Compare String — There is a new instruction CMPS mem32, mem32 (or CMPSD) beyond the compare string instruction available with the 80286. This instruction compares 32-bit words ES:ED1 (second operand) with DS:ESI and affects the flags. The direction of subtraction of CMPS is [[ESI]] - [[ED1]]. The left operand ESI is the source and the right operand EDI is the destination. This is a reverse of the normal Intel convention in which the left operand is the destination and the right operand is the source. This is true for byte (CMPSB) or word (CMPSW) compare instructions. The result of subtraction is not stored; only the flags are affected. For the first operand (ESI), the DS is used as segment unless a segment override byte is present, while the second operand (EDI) must use ES as the segment register and cannot be overridden. ESI and EDI are incremented by 4 if DF = 0, while they are decremented by 4 if DF = 1. CMPSS can be preceded by the REP.E or REPNE prefix for block comparison. All flags are affected.
Load and Move Strings — There are two new 80386 instructions beyond those of 80286. These are LODS, mem32 (or LODSD) and MOVVS mem32, mem32 (or MOVSD). LODSD loads the doubleword (32-bit) from a memory location specified by DS:ESI into EAX. After the load, ESI is automatically incremented by 4 if DF = 0, while ESI is automatically decremented by 4 if DF = 1. No flags are affected. LODS can be preceded by REP prefix. LODS is typically used within a loop structure because further processing of the data moved into EAX is normally required. MOVSD copies the doubleword (32-bit) at memory location addressed by DS:ESI to the memory location at ES:EDI. DS is used as the segment register for the source and may be overridden. ES must be used as the segment register and cannot be overridden. After the move, ESI and EDI are incremented by four if DF = 0, while they are decremented by 4 if DF = 1. MOVVS can be preceded by the REP prefix for block movement of ECX doublewords. No flags are affected.

String I/O Instructions — There are two new 80386 string I/O instructions beyond those of the 80286. These are INS mem32, DX (or INSD) and OUTS DX, mem32 (or OUTSD). INS inserts a 16-bit data from a port addressed by the content of DX into a memory location specified by ES:EDI. ES cannot be overridden. After data transfer, EDI is automatically incremented by 4 if DF = 0, while it is decremented by 4 if DF = 1. INSD can be preceded by the REP prefix for block input of ECX doublewords. No flags are affected. OUTSD instruction outputs 32-bit data from a memory location addressed by DS:ESI to a port addressed by the content of DX. DS can be overridden. After data transfer, ESI is incremented by 4 if DF = 0 and decremented by 4 if DF = 1. OUTSD can be preceded by the REP prefix for block output of ECX doublewords.

Store and Scan Strings — There is a new 80386 STOS mem32 (or STOSD) instruction. STOS stores the contents of the EAX register to a doubleword addressed by ES and EDI. ES cannot be overridden. After storing, EDI is automatically incremented by 4 if DF = 0 and decremented by 4 if DF = 1. No flags are affected. STOS can be preceded by the REP prefix for a block fill of ECX doublewords. There is a new scan instruction called the SCAS mem32 (or SCASD) in the 80386. SCASD performs the 32-bit subtraction [EAX] — [memory addressed by ES and EDI]. The result of subtraction is not stored, and the flags are affected.

4.2.2.1 Table Look-Up Translation Instruction
There is a modified version of the 80286 XLAT instruction available in the 80386.

XLAT mem8 (or XLATB) replaces the AL register from the table index to the table entry. AL should be the unsigned index into a table addressed by DS:BX for 16-bit address (available in 80286 and 80386) and DS:EBX for 32-bit address (available only in 80386). CS can be overridden. No flags are affected. ES cannot be overridden. If DF = 0, EDI is incremented by 4 and if DF = 1, EDI is decremented by 4.

SCASD can be preceded by the REP or REPNE prefix for block search of ECX doublewords. All flags are affected.

4.2.2.2 High-Level Language Instructions

The three instructions ENTER, LEAVE, and BOUND (also available with 80186/80286) in this category have been enhanced in 80386.

The ENTER imm16, imm8 instruction creates a stack frame. The data imm8 defines the nesting depth of the subroutine and can be from 0 to 31. The value 0 specifies the first subroutine only. The data imm16 defines the number of stack frame pointers copied into the new stack frame from the preceding frame.

After the instruction is executed, the 80386 uses EBP as the current frame pointer and ESP as the current stack pointer. The data imm16 specifies the number of bytes of local variables for which the stack space is to be allocated.

If imm8 is zero, ENTER pushes the frame pointer EBP onto the stack; ENTER then subtracts the first operand imm16 from the ESP and sets EBP to the current ESP.

For example, a procedure with 28 bytes of local variables would have an ENTER 28, 0 instruction at its entry point and a LEAVE instruction before every RET. The 28 local bytes would be addressed as offset from EBP. Note that the LEAVE instruction sets ESP to EBP and then pops EBP. For 80186 and 80286, ENTER and LEAVE instructions use BP and SP instead of EBP and ESP. The 80386 uses BP (low 16 bits of EBP) and SP (low 16 bits of ESP) for 16-bit operands, and EBP and ESP for 32-bit operands.

The formal definition of the ENTER instruction is given in the following:
LEVEL denotes the value of the second operand, imm8:
Push EBP
Set a temporary value FRAME-PTR := ESP
If LEVEL > 0 then
  Repeat (LEVEL-1) times:
    ESP := ESP-4
    Push the doubleword pointed to by ESP
  End repeat
  Push FRAME-PTR
End if
ESP := FRAME-PTR
EBP := first operand, imm16

BOUND ensures that a signed array index is within the limits specified by a block of memory containing an upper and lower bound. The 80386 provides two forms of the BOUND instruction:

BOUND reg16, mem32
BOUND reg32, mem64

The first form is for 16-bit operands and is also available with the 80186 and 80286. The second form is for 32-bit operands and is included in the 80386 instruction set. For example, consider BOUND EDI, ADDR. Suppose [ADDR] = 32-bit lower bound, d, and [ADDR+4] = 32-bit upper bound du. If, after execution of this instruction, [EDI]<d, and >du, the 80386 traps to interrupt 5; otherwise the array is accessed.

Example 4.2
Determine the effect of each one of the following 80386 instructions:
i) CDQ
ii) BTC CX, BX
iii) MOVX ECX, E7H

i) After CDQ,
   [EAX] = [FFFFFF H
   [EDX] = [FFFFFF H

ii) After BTC CX, BX,
   [CX] = 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Bn no.
   0 0 0 0 1 1 1 1 1 0 1 0 0 0 1.

Hence, [CX] = 0F81H and [BX] = 0004H

iii) MOVX ECX, E7H copies the 8-bit data E7H into low byte of ECX and then sign-extends to 32 bits. Therefore, after MOVX ECX, E7H, [ECX] = FFFFFFF7H.

Example 4.3
Write an 80386 assembly language program to multiply a signed 8-bit number by a signed 32-bit number in ECX. Store result in EAX. Assume that the segment registers are already initialized and also that the result fits within 32 bits.

Solution
IMUL ECX, data8 ; Perform signed multiplication
MOV EAX, ECX ; Store result
HLT ; in EAX and stop

Example 4.4
Write an 80386 assembly program to move two columns of 10,000 32-bit numbers from A (i) to B (i). In other words, move A (1) to B (1), A (2) to B (2), and so on.
4.2.3 Memory Organization

Memory on the 80386 is structured as 8-bit (byte), 16-bit (word), or 32-bit (doubleword) quantities. Words are stored in two consecutive bytes with low byte at the lower address and high byte at the higher address. The byte address of the low byte addresses the word. Doublewords are stored in four consecutive bytes in memory with byte 0 at the lowest address and byte 3 at the highest address. The byte address of byte 0 addresses the doubleword.

Memory on the 80386 can also be organized as pages or segments. The entire memory can be divided into one or more variable length segments which can be shared between programs or swapped to disks. Memory can also be divided into one or more 4K-byte pages. Segmentation and paging can also be combined in a system. The 80386 includes three types of address spaces. These are logical or virtual, linear, and physical. A logical or virtual address contains a selector (contents of a segment register) and offset (effective address) obtained by adding the base, index, and displacement components discussed earlier. Since each task on the 80386 can have a maximum of 16K selectors and offsets can be 4 gigabytes (2^39 bits), the programmer views a virtual address space of 2^39 or 64 terabytes of logical address space per task.

The 80386 on-chip segment unit translates the logical address space to 32-bit linear address space. If the paging unit is disabled, then the 32-bit linear address corresponds to the physical address. On the other hand, if the paging unit is enabled, the paging unit translates the linear address space to the physical address space. Note that the physical addresses are generated by the 80386 on its address pins.

The main difference between real and protected modes is how the 80386 segment unit translates logical addresses to linear addresses. In real mode, the segment unit shifts the selector to the left four times and adds the result to the offset to obtain the linear address. In protected mode, every selector has a linear base address. The linear base address is stored in one of two operating system tables (local descriptor table or global descriptor table). The selector's linear base address is summed with the offset to obtain the linear address. Figure 4.12 shows the 80386 address translation mechanism.

![Address Translation Mechanism](image-url)

There are three main types of 80386 segments. These are code, data, and stack segments. These segments are of variable size and can be from 1 byte to 4 gigabytes (2^39 bits) in length.

Instructions do not explicitly define the segment type. This is done in order to obtain compact instruction encoding. A default segment register is automatically selected by the 80386 according to Table 4.5.

In general, DS, SS, and CS use the selectors for data, stack, and code. Segment override prefixes can be used to override a given segment register as per Table 4.5.
TABLE 4.5
Segment Register Selection Rules

<table>
<thead>
<tr>
<th>Type of memory reference</th>
<th>Implied (default) segment use</th>
<th>Segment override prefixes possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code fetch</td>
<td>CS</td>
<td>None</td>
</tr>
<tr>
<td>Destination of PUSH, PUSHA instructions</td>
<td>SS</td>
<td>None</td>
</tr>
<tr>
<td>Source of POP, POPA instructions</td>
<td>SS</td>
<td>None</td>
</tr>
<tr>
<td>Other data references, with effective address using base register of:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[EAX]</td>
<td>DS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[EBX]</td>
<td>DS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[ECX]</td>
<td>DS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[EDX]</td>
<td>DS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[EBX]</td>
<td>DS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[EDX]</td>
<td>DS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[EDI]</td>
<td>DS</td>
<td>CS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>[EBX]</td>
<td>SS</td>
<td>CS, DS, ES, FS, GS</td>
</tr>
<tr>
<td>[ESB]</td>
<td>SS</td>
<td>CS, DS, ES, FS, GS</td>
</tr>
</tbody>
</table>

4.2.4 I/O SPACE
The 80386 supports both standard and memory-mapped I/O. The I/O space contains 64K 8-bit ports, 32K 16-bit ports, 16K 32-bit ports, or any combination of ports up to 64K bytes. I/O instructions do not go through the segment or paging units. Therefore, the I/O space refers to physical memory. The M/IO pin distinguishes between the memory and I/O.

The 80386 includes IN and OUT instructions to access I/O ports with port address provided by DL, DX, or EDX registers. All 8- and 16-bit port addresses are zero-extended on the upper address lines. The IN and OUT instructions drive the 80386 M/IO pin to low.

I/O port addresses 000F8H through 00FFH are reserved by Intel. The coprocessors in the I/O space are at locations 800000F8H through 800000FFH.

4.2.5 80386 INTERRUPTS
Earlier, features and exceptions of interrupts and exceptions which are of interest to application programmers were discussed. In this section, details of these interrupts and exceptions are covered. The difference between interrupts and exceptions is that interrupts are used to handle asynchronous external events and exceptions handle instruction faults. The 80386 also treats software interrupts such as INT n as exceptions.

The 80386 interrupts and exceptions are similar to those of the 8086. There are three types of interrupts/exceptions. These are hardware interrupts, exceptions, and software interrupts.

Hardware interrupts can be of two types. The 80386 provides the NMI pin for the nonmaskable interrupt. When the NMI pin encounters a LOW to HIGH transition by an external device such as an A/D converter, the 80386 services the interrupt via the internally supplied instruction INT 2.

The INT 2 instruction does not need to be provided via external hardware. The 80386 services a maskable interrupt when its INTR pin is activated HIGH and the IF bit is set to one. An 8-bit vector can be supplied by the user via external hardware which identifies the interrupt source.

The IF bit in the EFLAGS register is reset when an interrupt is being serviced. This, in turn, disables servicing additional interrupts during an interrupt service routine.

When an interrupt occurs, the 80386 completes execution of the current instruction. The 80386 then pushes the EIP, CS and flags onto the stack. Next, the 80386 obtains an 8-bit vector via either external hardware (maskable) or internally (nonmaskable) which identifies the appropriate entry in the interrupt table. The table contains the starting address of the interrupt service routine. At the end of the interrupt service routine, IRET can be placed to resume the program at the appropriate place in the main program.

The software interrupt due to execution of INT n has the same effect as the hardware interrupt. A special case of the software interrupt INT n is the INT 3 or breakpoint interrupt. Like the 8086, the single step interrupt is enabled by setting the TF bit. The TF bit is set by altering the stack image and executing a POPF or IRET instruction. The single step uses INT 1. Exceptions are classified as faults, traps, or aborts depending on the way they are reported and whether or not the instruction causing the exception is restarted. Faults are exceptions that are detected and serviced before the execution of the faulting instruction. A fault can occur in a virtual memory system when the 80386 references a page or a segment not present in the main memory. The operating system can execute a service routine at the fault's interrupt address vector to fetch the page or segment from disk. Then the 80386 restarts the instruction traps and immediately reports the cause of the problem via the execution of the instruction.
Typical examples of traps are user-defined interrupts. Aborts are exceptions which do not allow the exact location of the instruction causing the exception to be determined. Aborts are used to report severe errors such as a hardware error or illegal values in system tables.

Therefore, upon completion of the interrupt service routine, the 80386 resumes program execution at the instruction following the interrupted instruction. On the other hand, the return address from an exception fault routine will always point to the instruction causing the exception. Table 4.6 lists the 80386 interrupts along with to where the return address points.

<table>
<thead>
<tr>
<th>Function</th>
<th>Interrupt number</th>
<th>Instruction which can cause exception</th>
<th>Return address points to faulting instruction</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide error</td>
<td>0</td>
<td>DIV, IDIV</td>
<td>Yes</td>
<td>Fault</td>
</tr>
<tr>
<td>Debug exception</td>
<td>1</td>
<td>Any instruction</td>
<td>Yes</td>
<td>Trap</td>
</tr>
<tr>
<td>NMI interrupt</td>
<td>2 INT 2 or NMI</td>
<td>No</td>
<td>No</td>
<td>NMI</td>
</tr>
<tr>
<td>One-byte interrupt</td>
<td>3 INT</td>
<td>No</td>
<td>No</td>
<td>Trap</td>
</tr>
<tr>
<td>Interrupt on overflow</td>
<td>4 INTD</td>
<td>No</td>
<td>No</td>
<td>Trap</td>
</tr>
<tr>
<td>Array bounds check</td>
<td>5 BOUND</td>
<td>Yes</td>
<td>Yes</td>
<td>Fault</td>
</tr>
<tr>
<td>Invalid OF-code</td>
<td>6 Any illegal instruction</td>
<td>Yes</td>
<td>Yes</td>
<td>Fault</td>
</tr>
<tr>
<td>Device not available</td>
<td>7 ESC, WAIT</td>
<td>Yes</td>
<td>Yes</td>
<td>Fault</td>
</tr>
<tr>
<td>Double fault</td>
<td>8 Any instruction that can generate an exception</td>
<td>Yes</td>
<td>Yes</td>
<td>Abort</td>
</tr>
<tr>
<td>Coprocessor segment</td>
<td>9 Coprocessor tries to access data past the end of a segment</td>
<td>No</td>
<td>No</td>
<td>Trap</td>
</tr>
<tr>
<td>Invalid TSS</td>
<td>10 JMP, CALL, IRET, INT</td>
<td>Yes</td>
<td>Yes</td>
<td>Fault</td>
</tr>
<tr>
<td>Segment not present</td>
<td>11 Segment register instructions</td>
<td>Yes</td>
<td>Yes</td>
<td>Fault</td>
</tr>
<tr>
<td>Stack fault</td>
<td>12 Stack references</td>
<td>Yes</td>
<td>Yes</td>
<td>Fault</td>
</tr>
<tr>
<td>General protection fault</td>
<td>13 Any memory reference</td>
<td>Yes</td>
<td>Yes</td>
<td>Fault</td>
</tr>
</tbody>
</table>

The 80386 can handle up to 256 different interrupts/exceptions. For servicing the interrupts, a table containing up to 256 interrupt vectors must be defined by the user. These interrupt vectors are pointers to the interrupt service routine. In real mode, the vectors contain two 16-bit words: the code segment and a 16-bit offset. In protected mode, the interrupt vectors are 8-byte quantities, which are stored in an interrupt descriptor table. Of the 256 possible interrupts, 32 are reserved by Intel and the remaining 224 are available to be used by the system designer.

If there are several interrupts/exceptions occurring at the same time, the 80386 handles them according to the following priorities:

1. (Highest) Exception faults
2. TRAP instructions
3. Debug traps for this instruction
4. Debug faults for next instruction
5. NMI
6. (Lowest) INTR

<table>
<thead>
<tr>
<th>Function</th>
<th>Interrupt number</th>
<th>Instruction which can cause exception</th>
<th>Return address points to faulting instruction</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page fault</td>
<td>14</td>
<td>Any memory access or code fetch</td>
<td>Yes</td>
<td>Fault</td>
</tr>
<tr>
<td>Coprocessor error</td>
<td>16</td>
<td>ESC, WAIT</td>
<td>Yes</td>
<td>Fault</td>
</tr>
<tr>
<td>Intel reserved</td>
<td>17—32</td>
<td>INT n</td>
<td>No</td>
<td>Trap</td>
</tr>
<tr>
<td>Two-byte interrupt</td>
<td>0—255</td>
<td>INT n</td>
<td>No</td>
<td>Trap</td>
</tr>
</tbody>
</table>
As an example, suppose an instruction causes both a debug exception (interrupt no. 1) and page fault (interrupt vector 14). According to the built-in priority mechanism, the 80386 will first service the page fault by executing the exception 14 handler. The exception 14 handler will be interrupted by the debug exception handler (1). An address in the page fault handler will be pushed onto the stack and the service routine for the debug handler (1) will be executed. After this, the exception 14 handler will be executed. This permits the system designer to debug the exception handler.

### 4.2.6 80386 RESET AND INITIALIZATION

Upon hardware reset, the 80380 registers contain the values as shown in Table 4.7.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag word</td>
<td>UUUU00002H</td>
</tr>
<tr>
<td>Machine status word (CR0)</td>
<td>UUUUUU00H</td>
</tr>
<tr>
<td>Instruction pointer</td>
<td>00000FF0H</td>
</tr>
<tr>
<td>Code segment</td>
<td>0000H</td>
</tr>
<tr>
<td>Data segment</td>
<td>0000H</td>
</tr>
<tr>
<td>Stack segment</td>
<td>0000H</td>
</tr>
<tr>
<td>Extra segment (ES)</td>
<td>0000H</td>
</tr>
<tr>
<td>Extra segment (FS)</td>
<td>0000H</td>
</tr>
<tr>
<td>Extra segment (GS)</td>
<td>0000H</td>
</tr>
<tr>
<td>All other registers</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

Note: U means undefined.

The 80386 executes instructions near the top of the physical memory at address FFFFFFF0H. When the 80386 executes the first instruction, it will go low and the 80386 executes instruction in the lower memory of physical memory. This allows the system designer to use a ROM at the top of physical memory to initialize the system and take care of resets. The 80386 is reset by activating its RESET pin high for at least 78 CLK2 periods.

### 4.2.7 TESTABILITY

The 80386 provides capability to perform self-test. The self-test checks all of the control ROM and the associate nonrandom logic inside the 80386. Self-test feature is performed when the 80386 RESET pin goes from HIGH to LOW and the BUSY # pin is LOW. The self-test takes about 50 milliseconds with a 16-MHz clock. After self-test, the 80386 performs reset and begins program execution. If the self-test is successful, the contents of both EAX and EDX are zero; otherwise the contents of EAX and EDX are not zero, indicating a faulty chip.

### 4.2.8 DEBUGGING

- In addition to the software breakpoint and single-stepping features, the 80386 also includes six program-accessible 32-bit registers for specifying up to four distinct breakpoints. Unlike the INT3 which only allows instruction breakpoints, the 80386 debug registers permit breakpoints to be set for data accesses. Therefore, a breakpoint can be set up if a variable is accidentally being overwritten. Thus, the 80386 can stop executing the program whenever the variable’s contents are being changed.

### 4.2.9 80386 PINS AND SIGNALS

As mentioned before, the 80386 is a 132-pin ceramic Pin Grid Array (PGA). Pins are arranged 0.1 inch (2.54 mm) center-to-center, in a 14 x 14 matrix, three rows around.

- A number of signals are available for low insertion force or zero insertion force mountings. These types of terminals include solderless, surface mount, or wire wrap. These options are available to the manufacturer by Amphenol Inc. of Harrisburg, PA, Advanced Interconnections of Warwick, RI, and Textool Products of Irving, TX.

- Figure 4.13 shows the 80386 pinout as viewed from the pin side of the chip. Table 4.8 provides the 80386 pinout functional grouping description.

- Figure 4.14 shows functional grouping of the 80380 pinout. A brief description of the 80386 pins and signals is provided in the following. The # symbol at the end of the signal name indicates the active or asserted state when the signal is low. When the symbol # is absent from the signal name, the signal is asserted when high.

- The 80386 has 20 Vcc and 21 GND pins for power distribution. These multiple power and ground pins reduce noise. Preferably, the circuit board should contain Vcc and GND planes.

- CLK2 pin provides the basic timing for the 86386. This clock is divided by 2 internally to provide the internal clock used for instruction execution.

- There are two phases (phase one and phase two) of the internal clock. Each CLK2 period defines a phase of the internal clock. Figure 4.15 shows the relationship. The 80386 is reset by activating the RESET pin for at least
TABLE 4.8
386XPGA Pinout Functional Grouping

<table>
<thead>
<tr>
<th>Pin/signal</th>
<th>Pin/signal</th>
<th>Pin/signal</th>
<th>Pin/signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>N5 A31</td>
<td>M5 D31</td>
<td>A1 Vcc</td>
<td>A2 Vss</td>
</tr>
<tr>
<td>P1 A30</td>
<td>P3 D30</td>
<td>A5 Vcc</td>
<td>A6 Vss</td>
</tr>
<tr>
<td>M2 A29</td>
<td>P4 D29</td>
<td>A7 Vcc</td>
<td>A9 Vss</td>
</tr>
<tr>
<td>L3 A28</td>
<td>M6 D28</td>
<td>A10 Vcc</td>
<td>B1 Vss</td>
</tr>
<tr>
<td>N1 A27</td>
<td>N5 D27</td>
<td>A14 Vcc</td>
<td>B5 Vss</td>
</tr>
<tr>
<td>M1 A26</td>
<td>P5 D26</td>
<td>C5 Vcc</td>
<td>B11 Vss</td>
</tr>
<tr>
<td>K3 A25</td>
<td>N6 D25</td>
<td>C12 Vcc</td>
<td>B14 Vss</td>
</tr>
<tr>
<td>L2 A24</td>
<td>P7 D24</td>
<td>D12 Vcc</td>
<td>C11 Vss</td>
</tr>
<tr>
<td>L1 A23</td>
<td>N8 D23</td>
<td>G2 Vcc</td>
<td>F5 Vss</td>
</tr>
<tr>
<td>K2 A22</td>
<td>P9 D22</td>
<td>G3 Vcc</td>
<td>F3 Vss</td>
</tr>
<tr>
<td>K1 A21</td>
<td>N9 D21</td>
<td>G12 Vcc</td>
<td>F14 Vss</td>
</tr>
<tr>
<td>J1 A20</td>
<td>M9 D20</td>
<td>G14 Vcc</td>
<td>J2 Vss</td>
</tr>
<tr>
<td>H3 A19</td>
<td>P10 D19</td>
<td>L12 Vcc</td>
<td>J3 Vss</td>
</tr>
<tr>
<td>H1 A18</td>
<td>N11 D18</td>
<td>M3 Vcc</td>
<td>J12 Vss</td>
</tr>
<tr>
<td>H1 A17</td>
<td>N10 D17</td>
<td>M7 Vcc</td>
<td>J13 Vss</td>
</tr>
<tr>
<td>G1 A16</td>
<td>N11 D16</td>
<td>M13 Vcc</td>
<td>M4 Vss</td>
</tr>
<tr>
<td>F1 A15</td>
<td>M11 D15</td>
<td>N4 Vcc</td>
<td>M8 Vss</td>
</tr>
<tr>
<td>E1 A14</td>
<td>P12 D14</td>
<td>N7 Vcc</td>
<td>M10 Vss</td>
</tr>
<tr>
<td>E2 A13</td>
<td>P13 D13</td>
<td>F2 Vcc</td>
<td>N3 Vss</td>
</tr>
<tr>
<td>E3 A12</td>
<td>N12 D12</td>
<td>P8 Vcc</td>
<td>P6 Vss</td>
</tr>
<tr>
<td>D1 A11</td>
<td>N13 D11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2 A10</td>
<td>M12 D10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D3 A9</td>
<td>N14 D9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1 A8</td>
<td>L13 D8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2 A7</td>
<td>K12 D7</td>
<td>E14 ADS#</td>
<td>B6 Vcc</td>
</tr>
<tr>
<td>C3 A6</td>
<td>L14 D6</td>
<td></td>
<td>B12 Vcc</td>
</tr>
<tr>
<td>B2 A5</td>
<td>K13 D5</td>
<td>B10 W/R#</td>
<td>C6 Vcc</td>
</tr>
<tr>
<td>B3 A4</td>
<td>K14 D4</td>
<td>A11 D/C#</td>
<td>C7 Vcc</td>
</tr>
<tr>
<td>A3 A3</td>
<td>H14 D3</td>
<td>A12 N/A#</td>
<td>E13 Vcc</td>
</tr>
<tr>
<td>C4 A2</td>
<td>H14 D2</td>
<td>G10 LOCK#</td>
<td>F13 Vcc</td>
</tr>
<tr>
<td>A13 BE3#</td>
<td>H13 D1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B13 BE2#</td>
<td>H12 D0</td>
<td>D13 NA#</td>
<td>C8 PEREQ</td>
</tr>
<tr>
<td>C13 BE1#</td>
<td>C14 B15#</td>
<td>B9 BUSY#</td>
<td></td>
</tr>
<tr>
<td>E12 BE0#</td>
<td>G13 READY#</td>
<td>A8 ERROR#</td>
<td></td>
</tr>
<tr>
<td>C9 RESET</td>
<td>M14 HIDA</td>
<td>B7 INTR</td>
<td>B8 NIA</td>
</tr>
</tbody>
</table>

FIGURE 4.13 386XPGA pinout view from pin side.

FIGURE 4.14 Functional signal groups.
15 CLK2 periods. The RESET signal is level-sensitive. When the RESET pin is asserted, the 80386 ignores all input pins and drives all output pins to idle bus state. The 82384 clock generator provides system clock and reset signals.

D0-D31 provides the 32-bit data bus. The 80386 can transfer 16- or 32-bit data via the data bus.

\[ \text{FIGURE 4.15 CLK2 signal and internal processor clock.} \]

The address pins A2-A31 along with the byte enable signals BE0# thru BE7# generate physical memory or I/O port addresses. Using these pins, the 80386 can directly address 4 gigabytes of physical memory (00000000H thru 0FFFFFFFH) and 64 kilobytes of I/O addresses (00000000H thru 0000000FHH). Therefore, coprocessor addresses range from 0000000FHH thru 0000000FFFFH. The coprocessor select signal is generated by the 80386 when M/IO # LOW and A21 HIGH.

The byte enable outputs, BE0# thru BE7# by the 80386, define which bytes of D0-D31 are utilized in the current data transfer. These definitions are given below:

BE0# is low when data is transferred via D0-D7
BE1# is low when data is transferred via D8-D15
BE2# is low when data is transferred via D16-D23
BE3# is low when data is transferred via D24-D31

The 80386 asserts one or more byte enables depending on the physical size of the operand being transferred (1, 2, 3, or 4 bytes).

When the 80386 performs a word memory write or word I/O write cycle via D16-D31 pins, it duplicates this data on D0-D15.

W/R#, D/C#, M/IO#, and LOCK# output pins specify the type of bus cycle being performed by the 80386. W/R# pin, when HIGH, identifies write cycle and, when LOW, indicates read cycle. D/C# pin, when HIGH, identifies data cycle and, when LOW, indicates control cycle. M/IO# differentiates between memory and I/O cycles. LOCK# distinguishes between locked and unlocked bus cycles. W/R#, D/C#, and M/IO# pins define the primary bus cycle. This is because these signals are valid when ADS# (address status output) is asserted. LOCK# output is valid as soon as the bus cycle begins, but due to address pipelining LOCK# may be valid later than ADS# asserted. Table 4.9 defines the bus cycle definitions.

The 80386 bus control signals include ADS# (address status), READY# (transfer acknowledge), NA# (next address request), and BS1# (bus size 16).

The 80386 outputs LOW on the ADS# pin to indicate a valid bus cycle (W/B#, D/C#, M/IO#) and address (BED#-BE3#, A2-A31) signals.

<table>
<thead>
<tr>
<th>M/IO#</th>
<th>D/C#</th>
<th>W/R#</th>
<th>Bus cycle type</th>
<th>Locked?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>INTERRUPT ACKNOWLEDGE</td>
<td>Yes</td>
</tr>
<tr>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Does not occur</td>
<td>—</td>
</tr>
<tr>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>I/O DATA READ</td>
<td>No</td>
</tr>
<tr>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>MEMORY CODE READ</td>
<td>No</td>
</tr>
<tr>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>HALT: SHUTDOWN:</td>
<td>No</td>
</tr>
</tbody>
</table>

Address = 2

Address = 0

BED# High (BE0# Low)

BE1# High (BE1# High)

BE2# Low (BE2# High)

BE3# High (BE3# High)

A2-A31 Low (A2-A31 Low)

A2-A31 Low (A2-A31 Low)

High

High

High

High

MEMORY DATA READ

MEMORY DATA WRITE

Some cycles

Some cycles
When READY# input is asserted during a read cycle or an interrupt acknowledge cycle, the 80386 latches the input data on the data pins and ends the cycle. When READY# is low during a write cycle, the 80386 ends the bus cycle.

The NA# input pin is activated low by an external hardware to request address pipelining. A low on this pin means that the system is ready to receive new values of BE0#-BE15#, A2-A31, W/R#, D/C#, and M/IO# from the 80386 even if the completion of the present cycle is not acknowledged on the READY# pin.

BS16# input pin permits the 80386 to interface to 32- and 16-bit data buses. When the BS16# input pin is asserted low by an external device, the 80386 uses the low-order half (D0-D15) of the data bus corresponding to BE0# and BE15# for data transfer. If the 80386 asserts BE2# or BE13# during a bus cycle, then assertion of BS16# by an external device in this cycle will automatically cause the 80386 to transfer the upper bytes via only D0-D15. For 32-bit data operands with BS16# asserted, the 80386 will automatically execute two consecutive 16-bit bus cycles to accomplish this.

HOLD (input) and HLDA (output) pins are 80386 bus arbitration signals. These signals are used for DMA transfers. PEREQ, BUSY#, and ERROR# pins are used for interfacing coprocessors such as 80287 or 80387 to the 80386. A HIGH on PEREQ (coprocessor request) input pin indicates that a coprocessor is requesting the 80386 to transfer data to or from memory. The 80386 thus transfers data between the coprocessor and memory. This signal is level-sensitive. A LOW on the BUSY# (coprocessor Busy) input pin means that the coprocessor is still executing an instruction and is not capable of accepting another instruction. The BUSY# pin avoids interference with a previous coprocessor instruction.

ERROR# (coprocessor error) input pin, when asserted LOW by the coprocessor, indicates that the previous coprocessor instruction generated a coprocessor error of a type not masked by the coprocessor's control register. This input pin is automatically sampled by the 80386 when a coprocessor instruction is encountered and, if asserted, the 80386 generates exception 7 for executing the error-handling routine.

There are two interrupt pins on the 80386. These are INTR (maskable) and NMI (nonmaskable) pins. INTR is level-sensitive. When INTR is asserted and IIF bit in the FFLAGS is 1, the 80386 (when ready) responds to the INTR by performing two interrupt acknowledge cycles and at the end of the second cycle latches an 8-bit vector on D0-D7 to identify the source of interrupt. To ensure INTR recognition, it must be asserted until the first interrupt acknowledge cycle starts.

NMI is leading-edge sensitive. It must be negated for at least 8 CLK2 periods and then be asserted for at least 8 CLK2 periods to assure recognition by the 80386. The servicing of NMI was discussed earlier.

Table 4.10 summarizes the characteristics of all 80386 signals.

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Signal function</th>
<th>Active state</th>
<th>Input/output</th>
<th>Input synch to CLK2</th>
<th>Output high impedance during HDLA?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK2</td>
<td>Clock</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>D0-D31</td>
<td>Data bus</td>
<td>High</td>
<td>I/O</td>
<td>S</td>
<td>Yes</td>
</tr>
<tr>
<td>BE0#-BE3#</td>
<td>Byte enables</td>
<td>Low</td>
<td>O</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>A2-A31</td>
<td>Address bus</td>
<td>High</td>
<td>O</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>W/R#</td>
<td>Write-read</td>
<td>High</td>
<td>O</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>D/C#</td>
<td>Data-control</td>
<td>High</td>
<td>O</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>M/IO#</td>
<td>Memory-I/O</td>
<td>High</td>
<td>O</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>LOCK#</td>
<td>Bus lock</td>
<td>Low</td>
<td>O</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ADS#</td>
<td>Address status</td>
<td>Low</td>
<td>O</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>N/A#</td>
<td>Next address</td>
<td>Low</td>
<td>I</td>
<td>S</td>
<td>—</td>
</tr>
<tr>
<td>BS16#</td>
<td>Bus size 16</td>
<td>Low</td>
<td>I</td>
<td>S</td>
<td>—</td>
</tr>
<tr>
<td>READY#</td>
<td>Transfer</td>
<td>Low</td>
<td>I</td>
<td>S</td>
<td>—</td>
</tr>
<tr>
<td>HOLD</td>
<td>Bus hold request</td>
<td>High</td>
<td>I</td>
<td>S</td>
<td>—</td>
</tr>
<tr>
<td>HLDA</td>
<td>Bus hold</td>
<td>High</td>
<td>O</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>PEREQ</td>
<td>Coprocessor</td>
<td>High</td>
<td>I</td>
<td>A</td>
<td>—</td>
</tr>
<tr>
<td>BUSY#</td>
<td>Coprocessor busy</td>
<td>Low</td>
<td>I</td>
<td>A</td>
<td>—</td>
</tr>
<tr>
<td>ERROR#</td>
<td>Coprocessor error</td>
<td>Low</td>
<td>I</td>
<td>A</td>
<td>—</td>
</tr>
<tr>
<td>INTR</td>
<td>Maskable interrupt request</td>
<td>Low</td>
<td>I</td>
<td>A</td>
<td>—</td>
</tr>
</tbody>
</table>
TABLE 4.10 (continued)  
80386 Signal Summary

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Signal function</th>
<th>Active state</th>
<th>Input sync or async</th>
<th>Output high impedance during HDLA?</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMI</td>
<td>Nonmaskable interrupt request</td>
<td>High</td>
<td>I</td>
<td>A</td>
</tr>
<tr>
<td>RESET</td>
<td>Reset</td>
<td>High</td>
<td>I</td>
<td>A (note)</td>
</tr>
</tbody>
</table>

Note: If the phase of the internal processor clock must be synchronized to external circuitry, RESET falling edge must meet setup and hold times $t_{su}$ and $t_{hh}$.

4.2.10 80386 BUS TRANSFER TECHNIQUE

The 80386 uses one or more bus cycles to perform all data transfers. The 32-bit address is generated by the 80386 from BE0#-BE3# and A2-A32 as follows:

<table>
<thead>
<tr>
<th>80386 address signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical base address</td>
</tr>
<tr>
<td>A31—A2 A1 A0 A31—A2 BE3# BE2# BE1# BE0#</td>
</tr>
<tr>
<td>A31—A2 0 0 A31—A2 X X X Low</td>
</tr>
<tr>
<td>A31—A2 0 1 A31—A2 X X Low High</td>
</tr>
<tr>
<td>A31—A2 1 0 A31—A2 X Low High High</td>
</tr>
<tr>
<td>A31—A2 1 1 A31—A2 Low High High</td>
</tr>
</tbody>
</table>

Dynamic bus sizing feature connects the 80386 with 32-bit or 16-bit data buses for memory or I/O. A single 80386 can be connected to both 16- and 32-bit buses. During each bus cycle, the 80386 dynamically determines bus width and transfers data to or from 32- or 16-bit devices. During each bus cycle, the 80386 B16# pin can be asserted for 16-bit ports or negate B16# for 32-bit ports by the external device. With B16# asserted, all transfers are performed via D0-D15 pins. Also, with B16# asserted, the 80386 automatically performs data transfers larger than 16 bits or misaligned 16-bits transfers in multiple cycles as needed.

Note that 16-bit memory or I/O devices must be connected on D0-D15 pins.

Assuming B16# only affects the 80386 when BE2# and/or BE3# are asserted during the cycle. Assertion of B16# does not affect the 80386 if data transfer is only performed via D0-D15. On the other hand, the 80386 is affected by assertion of the B16# pin, depending on which byte enable pins are asserted during the current bus cycle. For example, asserting B16# during “upper half only” reads the 80386 to read data on the D0-D15 pins and ignores data on the D16-D31. Data that would have been read from D16-D31 (as indicated by BE2# and BE3#) will instead be read from D0-D15.

A 32-bit-wide memory can be interfaced to the 80386 by utilizing its B16#, BE0#-BE3#, and A2-A31 pins. Each 32-bit memory word starts at a byte address that is a multiple of 4. B16# is connected to HIGH (negated) for all bus cycles for 32-bit transfers. A2-A31 and BE0#-BE3# are used for addressing the memory.

For 16-bit memories, each 16-bit memory word starts at an address which is a multiple of 2. The address is decoded to assert B16# only during bus cycles for 16-bit transfers.

A2-A31 can be used to address 16-bit memory also. A1 and two-byte enable signals are also required.

To obtain A1 and two-byte enables for 16-bit transfers, BE0#-BE3# should be decoded as in Table 4.11.

Figure 4.16 shows a block diagram interfacing 16- and 32-bit memories to 80386.

Finally, if an operand is not aligned such as a 32-bit doubleword operand beginning at an address not divisible by 4, then multiple bus cycles are required for data transfer.

4.2.11 80386 READ AND WRITE CYCLES

The 80386 performs data transfer during bus cycles (also called read or write cycle).

Two choices of address timing are dynamically selectable. These are nonpipelined and pipelined. One of these timing choices is selectable on a cycle-by-cycle basis with the Next Address (NA#) input.

After a bus idle state, the 80386 always uses nonpipelined address timing. However, the NA# may be asserted by an external device to select pipelined address timing, for the next cycle is made available before the present bus cycle is terminated by the 80386 by asserting READY#.
TABLE 4.11
Generating A1, BHE#, and BLE# for Addressing 16-Bit Devices

<table>
<thead>
<tr>
<th>80386 signals</th>
<th>16-bit bus signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE3#</td>
<td>BE2#</td>
</tr>
<tr>
<td>H*</td>
<td>H*</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>H*</td>
<td>L*</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L*</td>
<td>H*</td>
</tr>
<tr>
<td>L*</td>
<td>H*</td>
</tr>
<tr>
<td>L*</td>
<td>H*</td>
</tr>
<tr>
<td>L*</td>
<td>L*</td>
</tr>
<tr>
<td>L*</td>
<td>L*</td>
</tr>
</tbody>
</table>

Note: BLE# asserted when D0-D7 of 16-bit bus is active; BHE# asserted when D8-D15 of 16-bit bus is active; A1 low for all even words; A1 high for all odd words.

Key: x = don't care
H = high voltage level
L = low voltage level
* = a nonoccurring pattern of Byte Enables; either none are asserted, or the pattern has Byte Enables asserted for noncontiguous bytes.

In general, the 80386 samples NA# input during each bus cycle to select the desired address timing for the next bus cycle.

Physical data bus width (16- or 32-bit) is selected by the 80386 by sampling the B16# (bus size 16) input pin near the end of the bus cycle. Assertion of B16# indicates a 16-bit data bus, while negation of B16# means a 32-bit data bus.

A read or write cycle is terminated by the 80386 on a low READY# (assertion) from the external device. Until the READY# is asserted, the 80386 inserts wait states to permit adjustment for the speed of any external device when a read cycle is terminated, and the 80386 latches the information present at its data pins. When a write cycle is acknowledged, the 80386 write data remain valid throughout phase one of the next bus state, to provide write data hold time.

To illustrate the concept of 80386 bus cycle timing, a mixture of read and write cycles with nonpipelined address timing is shown in Figure 4.17.

This diagram shows the fastest possible cycles with nonpiped address timing having two bus states (T1 and T2) per bus cycle. In phase one T1, the address signals and bus control signals are valid and the 80386 activates AD5# low to indicate their availability.

During read cycle, the 80386 tristates its data signals to permit driving by the external device being addressed. During write cycle, the 80386...
memory. The address lines A2–A19, B0–B13 are used by the 80386 in this mode. Paging is not provided in real mode. Therefore, linear addresses are identical to physical addresses. The 20-bit physical address is formed by adding the shifted (four times to the left) segment registers to an offset as shown in Figure 4.18.

All segments in real mode are exactly 64K bytes wide. Segments can be overlapped in this mode. There are two memory areas which are reserved in real mode for system initialization and interrupt pointer table. Addresses 000000H thru 002FFH are reserved for the interrupt pointer table, while addresses FFFFEFFH thru FFFFFFFH are reserved for system initialization. Many of the exceptions listed in Table 4.6 are not applicable to real mode. Exceptions 10, 11, 12, and 14 will never occur in this mode. Also, other exceptions have minor variations as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Interrupt number</th>
<th>Related instructions</th>
<th>Return address location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt table</td>
<td>8</td>
<td>INT vector is not</td>
<td>Before instruction</td>
</tr>
<tr>
<td>Limit too small</td>
<td></td>
<td>within table limit</td>
<td></td>
</tr>
<tr>
<td>Segment overrun</td>
<td>13</td>
<td>Word memory reference with offset = FFFFH or an attempt to execute an instruction past the end of a segment</td>
<td>Before instruction</td>
</tr>
</tbody>
</table>

4.2.12 80386 MODES

The 80386 can be operated in real, protected, or virtual 80386 mode. These modes are described below.

4.2.12.1 80386 Real Mode

Upon reset or power-up, the 80386 operates in real mode. In real mode, the 80386 can access all the 8086 registers along with the 80386 32-bit registers. The memory addressing, memory size, and interrupts of 80386 in real mode are the same as those of the 80286 in real mode.

The 80386 can execute all the instructions in real mode. The main purpose of real mode is to initialize the 80386 for protected mode operation.

In real mode, the 80386 can directly address up to one megabyte of
4.2.12.2 Protected Mode

The total 80386 capabilities are available when the 80386 operates in protected mode. This mode increases the linear space to four gigabytes (2^32 bytes) and permits the execution of virtual memory programs of 64 terabytes (2^64 bytes). Also, in protected mode, the 80386 can run all existing 8086 and 80286 programs with on-chip memory management and protection features. The protected mode includes new instructions to support multitasking operating systems. The main difference between protected mode and real mode from a programmer's viewpoint is the increased memory space and a differing addressing mechanism. Similar to real mode, protected mode also includes two elements (16-bit selector for determining a segment’s base address and a 32-bit offset or effective address) to obtain a 32-bit linear address. This 32-bit linear address is either used as the 32-bit physical address or, if paging is enabled, the paging mechanism translates this 32-bit linear address to a 32-bit physical address. Figure 4.19 shows the protected mode addressing mechanism. The selector is used to specify an index into a table defined by the operating system. The table includes the 32-bit base address of a given segment. The physical address is obtained by summing the base address obtained from the table to the offset.

![Figure 4.19 Protected mode addressing.](image)

With the paging mechanism enabled, the 80386 provides additional memory management mechanisms. The paging feature manages large 80386 segments.

The paging mechanism translates the protected linear addresses from the segmentation unit into physical addresses. Figure 4.20 shows this translation scheme.

Let us now discuss 80386 segmentation, protection, and paging features.

Segmentation provides both memory management and protection. All information about the segments is stored in an 8-byte data structure called a descriptor. All the descriptors are stored in tables identified by the 80386 hardware. There are three types of tables holding 80386 descriptors: global descriptor table (GDT), local descriptor table (LDT), and interrupt descriptor table (IDT). These tables are memory arrays of variable lengths. Their sizes can vary from 8 bytes to 64K bytes. Each table can store up to 8192 8-byte descriptors. The upper 13 bits of a selector are used as an index into the descriptor table. The tables have associated registers which store a 32-bit linear base address and a 16-bit limit for each table. Each table has a set of registers, namely, GDTR (32-bit), LDTR (16-bit), and IDTR (32-bit), associated with it. The 80386 instructions LGDT, LLDT, and LIDT are used to load the base and 16-bit limit of the global, local, and interrupt descriptor tables into the appropriate register. The SGDT, SLDT, and SIDLDT instructions store the base and limit values.

The GDT contains descriptors which are available to all the tasks in the system. In general, the GDT contains code and data segments used by the operating system, task state segments, and descriptors for LDTs in a system. LDTs store descriptors for a given task. Each task has a separate LDT, while the GDT contains descriptors for segments which are common to all tasks.
The IDT contains the descriptors which point to the location of up to 256 interrupt service routines. Every interrupt used by a system must have an entry into the IDT. The IDT entries are referenced via INT instructions, external interrupt vectors, and exceptions.

The object to which the selector points to is called a descriptor. Descriptors are eight bytes wide containing attributes about a given segment. These attributes contain the 32-bit base linear address of the segment, segment length, protection level, read/write/execute privileges, the default operand size (16 or 32 bits), and segment type.

In order to provide operating system compatibility between the 80286 and 80386, the 80386 supports all of the 80286 segment descriptors. The only differences between the 80286 and 80386 formats are that the values of the type fields and the limit and base address fields have been expanded for the 80386.

The 80286 system segment descriptors contain a 24-bit base address and 16-bit limit, while the 80386 system segment descriptors have a 32-bit base address, a 20-bit limit field, and a granularity bit. Note that the segment length is page granular if the granularity bit is one; otherwise, the segment length is byte granular.

By supporting 80286 segments the 80386 is able to execute 80286 application programs on a 80386 operating system. This is possible because the 80386 automatically can differentiate between the 80286-type and 80386-type descriptors. In particular, if the high order word of a descriptor is zero, then that descriptor is a 80286-type descriptor.

The only other differences between the 80286 and 80386 descriptors are the interpretation of the word count field of call gates and the B bit. The word count field specifies the number of 16-bit quantities to copy for 80286 call gates and 32-bit quantities for 80386 call gates. The B bit controls the size of pushes when using a call gate. If B = 0, then pushes are 16 bits, while pushes are 32 bits for B = 1.

The 80386 provides four protection levels for supporting a multi-tasking operating system to isolate and protect user programs from each other and the operating system. The privilege level controls the use of privileged instructions, I/O instructions, and access to segments and segment descriptors. The 80386 includes the protection as part of its memory management unit. The 80386 also provides an additional type of protection when paging is enabled.

The four-level hierarchical privilege system is shown in Figure 4.21. It is an extension of the user/ supervisor privilege mode used by minicomputers. Note that the user/supervisor mode is supported by the 80386 paging mechanism. The Privilege Levels (PL) are numbered 0 thru 3. Level 0 is the most privileged level.

The 80386 provides the following rules of privilege to control access to both data and procedures between levels of a task:

- Data stored in a segment with a privilege level 0 can be accessed only by code executing at a privilege level at least as privileged as 0.
- A code segment/procedure with privilege level 0 can only be called by a task executing at the same or a lesser privilege level than 0.

The 80386 supports task gates (protected indirect calls) to provide a secure method of privilege transfers within a task.

The 80386 also supports a rapid task switch operation via hardware. It saves the entire state of the machine (all of the registers, address space, and a link to the previous task), loads a new execution state, performs protection checks, and commences execution in the new task in approximately 17 microseconds.

Paging is another type of memory management for virtual memory multitasking operating systems. The main difference between paging and segmentation is that paging divides programs/data into several equal-sized pages, while segmentation divides programs/data into several variable-sized segments.
There are three elements associated with the 80386 paging mechanism. These are page directory, page tables, and the page itself (page frame). Paging mechanism does not have memory fragmentation since all pages have the same size of 4K bytes. Figure 4.22 shows the 80386 paging mechanism.

![Page Directory Diagram](image)

**FIGURE 4.22 paging mechanism.**

There are four 32-bit control registers (CR0-CR3) associated with the paging mechanism. CR2 is the page fault linear address register and contains the 32-bit linear address which caused the last page fault detected. CR3 is the page directory physical base address register and contains the physical starting address of the page directory. The lower 12 bits of CR3 are always zero to ensure that the page directory is always page aligned. CR1 is reserved for future Intel processors. CR0 contains 6 defined bits for control and status purposes. The low-order 10 bits of CR0 are known as the machine status word and include special control bits such as the enable bit and the protection enable bit.

The page directory is 4K bytes wide and permits up to 1024 page directory entries. Each page directory entry contains the address of the next level of tables, page tables, and information about the page table. The upper 10 bits of the linear address (A22-A31) are used as an index to select the correct page directory entry.

Each page table is 4K bytes and holds up to 1024 page table entries. Page table entries contain the starting address of the page frame and statistical information about the page such as whether the page can be read or written in supervisor or user mode. Address bits A12-A21 are used as index to select one of the 1024 page table entries. The 20 upper-bit page frame address is concatenated with the lower 12 bits of the linear address to form the physical address. Page tables can be shared between tasks and swapped to disks.

The lower 12 bits of the page table entries and page directory entries contain statistical information about pages and page tables, respectively. As an example, the P (present) bit indicates whether a page directory or page table entry can be used in address translation. If P = 1, the entry can be used in address translation, and if P = 0, the entry cannot be used for translation and all other 31 bits are available for use by the software. These 31 bits can be used to indicate where on a disk the page is located.

The 80386 provides a set of protection attributes for paging systems. The paging mechanism provides two levels of protection: user and supervisor. The user level corresponds to level 3 of the segmentation-based protection and the supervisor level combines all of the other protection levels (0, 1, 2). Programs executing at level 0, 1, or 2 bypass the page protection, although segmentation-based protection is still enforced by hardware.

The 80386 takes care of the page address translation process, relieving the burden from an operating system in a demand-paged system. The operating system is responsible for setting up the initial page tables and the handling of any page faults. The operating system initializes the tables by loading CR3 with the address of the page directory and allocates space for the page directory and the page tables. The operating system also implements a swapping policy and handles all of the page faults.

### 4.2.12 Virtual 8086 Mode

The virtual 8086 mode permits the execution of 8086 applications while taking full advantage of the 80386 protection mechanism. In particular, the 80386 permits concurrent execution of 8086 operating systems and applications, an 80386 operating system, and both 80286 and 80386 application. For example, in a multipurpose 80386-based microcomputer, one person can run an MS-DOS spreadsheet, another person can use MS-DOS, and a third person can run multiple UNIX utilities and applications.

One of the main differences between 80386 real and protected modes is how the segment selectors are interpreted. In virtual 8086 mode, the segment registers are used in the same way as the real mode. The contents of the segment register are shifted 4 times to the left and added to the offset to obtain the linear address.
The paging hardware permits the simultaneous execution of several virtual mode tasks and provides protection.

The paging hardware allows the 20-bit linear address produced by a virtual mode program to be divided up into 256 pages. Each one of the pages can be located anywhere within the maximum 4-gigabyte physical address space of the 80386.

The paging hardware also permits sharing of the 8086 operating system code by several 8086 applications. All virtual mode programs execute at privilege level 3. Therefore, virtual mode programs are subject to all of the protection checks defined in protected mode. This is different from real mode which executes programs in level 0.

QUESTIONS AND PROBLEMS

4.1 Write an 80186 assembly program to multiply a 16-bit signed number in BX by 00F31H. Assume that the result is 16 bits wide.

4.2 Identify the peripheral functional blocks integrated into the 80186.

4.3 What is the purpose of the relocation register in the 80186 peripheral control block?

4.4 What is the relationship between internal and external clocks of the 80186?

4.5 Identify the basic differences between 8086 and 80186.

4.6 Identify the main differences between the 80186 and 80286.

4.7 How much physical and virtual memory can the 80286 address?

4.8 What is the difference between the 80286 real address mode and FVAMD? Explain how these two modes can be switched back and forth.

4.9 Explain how the 80286 determines where in memory the global descriptor table and the present local descriptor table are located.

4.10 Discuss briefly the 80286 protection mechanism.

Questions and Problems

4.11 What is meant by the 80286 task state segment for each active task? How can this be accessed?

4.12 Explain the meaning of 80286 call gates.

4.13 Discuss briefly the 80286 memory management features and task switching.

4.14 What is the purpose of 80286 CAP, COD/INTA pins?

4.15 Identify the 80286 pins used for interfacing it to a coprocessor.

4.16 Discuss the issues associated with isolating a user program from a supervisor program and then utilize the 80286's protection features for protection. Assume that no task switching is involved. Also, assume that the supervisor program will perform all I/O operations and be present in the virtual memory space.

4.17 Compare the features of the 80386 with those of the 80286 from the following point of view: registers, clock rate, number of pins, number of instructions, modes of operation, memory management, and protection mechanism.

4.18 What are the basic differences between the 80386 real, protected, and virtual 8086 modes?

4.19 Assume the following register contents:

- [EBX] = 0000 2000H
- [ECX] = 0500 0000H
- [EDX] = 5000 5000H

Prior to execution of each of the 80386 instructions listed below, determine the effective address after execution of each instruction and identify the addressing modes of both source and destination:

i) MOV [EBX + 2] [ECX], EDX

ii) MOV [EBX + 4] [ECX + 20H], EDX
4.20 Determine the effect of each of the following 80386 instructions:

i) \texttt{MOVZX ECX, BX}

\begin{verbatim}
assume [ECX] = F1250024H

[BX] = F130H
\end{verbatim}

prior to execution of the MOVZX instruction.

ii) \texttt{SHLD CX, BX, 0 IF}

\begin{verbatim}
\end{verbatim}

prior to execution of the SHLD instruction.

4.21 Write an 80386 assembly language program to divide a signed 64-bit number in EBX:EDX by a 16-bit signed number in AX. Store the 32-bit quotient and remainder in memory locations.

4.22 Write an 80386 assembly program to compute $X^2/N$ where $N=100$ and $X$'s are signed 32-bit numbers. Assume that $X^2$ can be stored as a 32-bit signed number without overflow.

4.23 Write an 80386 assembly program to input 100 32-bit string data via a port addressed by DX. The program will then store the data in memory locations addressed by [DS] and [ES].

4.24 Discuss how the following situation will be handled by the 80386: The 80386 executing an instruction causes both a general protection fault (interrupt 13) and coprocessor segment overrun (interrupt 9).

4.25 How does the 80386 generate the 32-bit physical address from A20-A31 and BE0#-BE3#?

4.26 What are the purposes of NA#, D/C#, BS16#, and ERROR# pins?

4.27 For 16- and 32-bit transfers, what is the logic level of the BS16# pin?

4.28 Discuss briefly 80386 segmentation unit, paging unit, and protection.

4.29 How many bits are required for the address in real and protected modes?

4.30 Discuss the basic differences between 80286 and 80386 descriptions.

4.31 What is the four-level hierarchical protection in protected mode?

4.32 Discuss briefly 80386 virtual mode.
Chapter 5

MOTOROLA MC68000

This chapter describes the details of the Motorola 68000 microprocessor. The basic architecture, addressing modes, instruction set, and interfacing features of the Motorola 68000 are included.

5.1 INTRODUCTION

The MC68000 is Motorola's first 16-bit microprocessor. All 68000 address and data registers are 32 bits wide and its ALU is 16 bits wide. The 68000 is designed using NMOS technology. The 68000 requires a single 5V supply. The processor can be operated from a maximum internal clock frequency of 25 MHz. The 68000 is available in several frequencies. These include 6 MHz, 8 MHz, 10 MHz, 12.5 MHz, 16.67 MHz, and 25 MHz. The 68000 does not have on-chip clock circuitry and, therefore, requires a crystal oscillator or external clock generator/driver circuit to generate the clock.

The 68000 has several different versions. These include 6800B, 68010, and 68012. The 68000 and 68010 are packaged in a 64-pin DIP (Dual In-line Package) with all pins assigned or in a 68-pin quad pack or Pin Grid Array (PGA) with some unused pins. The 68000 is also packaged in 68-terminal chip carrier. The 68008 is packaged in a 48-pin dual in-line package while the 68012 is packaged in 84-pin grid array. The 68008 provides the basic 68000 capabilities with inexpensive packaging. It has an 8-bit data bus which facilitates interfacing of this chip to inexpensive 8-bit peripheral chips.

The 68010 provides hardware-based virtual memory support and effi
cient looping instructions. Like the 68000, it has a 16-bit data bus and a 24-bit address bus.

The 68012 includes all the 68010 features with a 31-bit address bus. The clock frequencies of the 68008, 68010, and 68012 are the same as the 68000.

The following table summarizes the basic differences among the 68000 family members:

<table>
<thead>
<tr>
<th></th>
<th>68000</th>
<th>68008</th>
<th>68010</th>
<th>68012</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data size (bits)</td>
<td>16</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Address bus size (bits)</td>
<td>24</td>
<td>20</td>
<td>24</td>
<td>31</td>
</tr>
<tr>
<td>Virtual memory</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Control registers</td>
<td>None</td>
<td>None</td>
<td>Three</td>
<td>Three</td>
</tr>
<tr>
<td>Directly addressable memory</td>
<td>16 Mbytes</td>
<td>1 Mbyte</td>
<td>16 Mbytes</td>
<td>2 gigabytes</td>
</tr>
</tbody>
</table>

In order to implement operating systems and protection features, the 68000 can be operated in two modes. These are supervisor and user modes. The supervisor mode is also called the operating system mode. In this mode, the 68000 can execute all instructions. However, in the user mode, the 68000 operates in one of these modes based on the S-bit of the Status register. When the S-bit is one, the 68000 operates in the supervisor mode. On the other hand, the 68000 operates in the user mode when S = 0.

Table 5.1 lists the basic differences between 68000 user and supervisor modes.

From Table 5.1 it can be seen that the 68000 executing a program in supervisor mode can enter the user mode by modifying the S-bit of the Status register to zero via an instruction. Instructions such as MOVE to SR, ORI to SR, EORI to SR can be used to accomplish this. On the other hand, the 68000 executing a program in user mode can enter the supervisor mode only via recognition of a trap, reset, or interrupt. Note that upon hardware reset, the 68000 operates in the supervisor mode and can execute all instructions. An attempt to execute privileged instructions (instructions that can only be executed in supervisor mode) in user mode will automatically generate an internal interrupt (trap) by the 68000.

The logical level in the 68000 Function Code pins: FC2 indicates to the external devices whether the 68000 is currently operating in user or supervisor mode. The 68000 has three function code pins (FC2, FC1, and FC0) which indicate to the external devices whether the 68000 is accessing supervisor program/data, user program/data, or performing an interrupt acknowledge cycle. These three pins are used for memory protection and for enabling an external chip such as the 74LS244 to provide an interrupt address vector.

The 68000 can operate on five different data types. These are bit, 4-bit BCD digit, 8-bit byte, 16-bit word, and 32-bit long word.

The 68000 provides 56 basic instructions. With 14 addressing modes, 56 instructions, and 5 data types, the 68000 includes more than 1,600 op codes. The fastest instruction is MOVE reg, reg and is executed in 500 ns at 8-MHz clock. The slowest instruction is 32-bit by 16-bit divide, which is executed in 21.25 µs at 8-MHz clock.

Like the 8-bit Motorola microprocessors such as Motorola 6800 and 6809, the 68000 supports memory-mapped I/O. Thus, the 68000 instruction set does not include any IN or OUT instruction.

The 68000 is a general-purpose register-based microprocessor since any data register can be used as an accumulator or as a scratch pad register. Even though the 68000 program counter is 32 bits wide, only the low-order 24 bits are used for PC. With 24 bits as address, the 68000 can directly address 16 megabytes (2^24) of memory.
5.2 68000 PROGRAMMING MODEL

The register architecture of the 68000 is shown in Figure 5.1. The 68000 chip contains eight 32-bit data registers (D0-D7) and nine 32-bit address registers (A0-A7, A7'). The 68000 uses A7 or A7' as the user or supervisor stack pointer, depending on the mode of operation. Data items such as bytes (8 bits), words (16 bits), long words (32 bits), and BCD numbers (4 bits) are usually stored in the data registers. On the other hand, the address of the operand is usually stored in an address register. Since the address sizes used by 68000 instructions can be either 16 or 24 bits, the address registers can only be used as 16- or 32-bit registers. While using the address registers as 32-bit, the 68000 discards the uppermost eight bits (bits 24 thru 31).

The 68000 status register consists of two bytes. These are a user byte and a system byte (Figure 5.2). The user byte includes the usual condition codes such as C, V, N, Z, and X. The meaning of C, V, N, and Z flags is obvious. However, the X-bit (extend bit) has a special meaning. The 68000 does not have any ADDC or SUBC instructions; rather, it has ADDX or SUBX instructions. For arithmetic operations, the carry flag C and the extend flag X are affected in an identical manner. This means that one can use ADDX or SUBX to include carries or borrows while adding or subtracting high-order long words in multiprecision additions or subtractions.

The system byte contains a 3-bit interrupt mask (12, 11, 10), a supervisor flag (S), and a trace flag (T). The interrupt mask bits (12, 11, 10) provide the status of the 68000 interrupt pins IEI2, IEI1, and IEI0. 12 11 10 = 000 indicates no interrupt, while 12 11 10 = 111 means nonmaskable interrupt. The other combinations of 12, 11, and 10 provide the 68000 maskable interrupt levels. It should be pointed out that signals on IEI2, IEI1, and IEI0 pins are inverted and then reflected on 12, 11, and 10, respectively. When the S-bit in SR is 1, the 68000 operates in the supervisor mode. When the S-bit = 0, the 68000 assumes user mode operation. When the TF (trace flag) is set to one, the 68000 generates an internal interrupt (trap) after execution of each instruction. A debugging routine can be written at the interrupt address vector to display registers and/or memory after execution of each instruction. This provides single-stepping facility. The 68000 can be placed in the single-step mode by setting the TF bit in SR so one by executing a logical privileged instruction such as ORI # $8000, SR in the supervisor mode.

5.3 68000 ADDRESSING STRUCTURE

The 68000 supports 8-bit bytes, 16-bit words, and 32-bit long words as shown in Figure 5.3. Byte addressing contains both odd and even
and can occupy one to five words. The first word is the operation word and it specifies the type of operation, sizes of operands, and their addressing modes. The remaining words contain other information such as immediate data or extensions to the effective addressing mode specified in the operation word. Figure 5.4 shows the 68000 instruction format structure.

FIGURE 5.4 68000 instruction format.

The 68000 contains more than 18 different instruction formats. Table 5.2 provides a description of some of them.

As an example, consider the instruction format number 1. The two-bit operand-type field specifies the operand size (01 for byte, 11 for word, and 10 for long word). The 6-bit operand field is divided into a 3-bit addressing mode field and a 3-bit register field. In 68000, all registers cannot be used in all modes. This means that a specific mode allows only a particular set of registers. For example, with an 8-bit MOVE instruction, an address register cannot be used as destination. Because of this limitation, a register field of three bits is used to address 16 registers (D0-D7 and A0-A7).

Further, several two-address instructions require one of the operands in the register mode. For example, the ADD (add) instruction (Format 2 of Table 5.2) provides a total of three combinations of source and destination. These are register-to-register, memory-to-register, and register-to-memory. The ADD instruction does not perform memory-to-memory addition. A few 68000 instructions contain multiple formats. For example, the ROL (rotate left) instruction includes two different forms: one for the memory rotate and the other for the register rotate.

Format numbers 4 and 5 allow the instructions to use small constants. For example, format number 4 permits one to move an 8-bit 2's comple-
TABLE 5.2
Some 68000 Instruction Formats

<table>
<thead>
<tr>
<th>Format number</th>
<th>Number of fields</th>
<th>Name of the field</th>
<th>Field size (in bits)</th>
<th>Instructions that use this format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>OP code</td>
<td>2</td>
<td>MOVE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operand type</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operand 1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operand 2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>OP code</td>
<td>4</td>
<td>ADD, AND, CMP, SUB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mode</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operand</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>OP code 1</td>
<td>4</td>
<td>MOVEP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register 1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mode</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OP code 1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>OP code 1</td>
<td>4</td>
<td>MOVEQ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OP code 2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>OP code 1</td>
<td>4</td>
<td>ADDQ, SUBQ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data (small constant)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OP code 2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operand type</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operand</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

The instruction MOVEQ $0A1,D5 will move the 8-bit immediate number $A1_{16}$ into low byte of D5 and sign-extends it so that D5 will finally contain $FFFFFFA1_{16}$.

Using format number 5, byte, word, or long word in a register or memory location can be incremented or decremented by as much as 7. The instruction ADDQ $\#8$, D1 will increment the low byte of D1 by 5 without affecting the upper bytes of D1.

5.5 68000 ADDRESSING MODES

Table 5.3 lists the 14 addressing modes of the 68000. The addressing modes are divided into six basic groups. These are register direct, address register indirect, absolute, program counter relative, immediate, and implied.

As mentioned in the last section, the 68000 contains three types of instructions: zero-operand, single-operand, and two-operand. The zero-operand instructions have no operands in the operand field. A typical example is CLC (Clear carry) instruction. The single-operand instructions contain the effective address, EA, in the operand field. The EAs of these instructions are calculated by the 68000 using the addressing mode specified for this operand. For two-operand instructions, one of the operands usually contains the EA and the operand is usually a register or memory location. The EAs in these instructions are calculated by the 68000 based on the addressing modes used for the EAs. Some two-operand instructions have EA in both operands. This means that the operands in these instructions can use two different addressing modes.

As mentioned before, the 68000 address registers do not support byte-sized operands. Therefore, when an address register is used as a source operand, either the low-order word or the entire long-word operand is used depending on the operation size. On the other hand, when an address register is used as the destination operand, the entire register is affected regardless of the operation size. If the operation size is a word, and the destination operand is an address register, then the 68000 performs the 16-bit operation, places the result in the low 16 bits of the address register, and then sign-extends the address register to 32 bits. An example is MOVELW $38050,A5. In this case, the source operand is 16-bit immediate datum $38050_{16}$, which is moved to the low 16 bits of A5, and the result is then sign-extended to 32 bits so that $[A5] = FFFF8050_{16}$. Data registers support data operands of byte, word, and long word size.

5.5.1 REGISTER DIRECT ADDRESSING

In this mode, the eight data registers (D0-D7) or seven address registers (A0-A6) contain the data operand. For example, consider ADD $30050000,D0$. The destination operand of this instruction is in data register direct mode.
Now, if \([005000] = [00023a], [D0] = [00063a]\), then after execution of
ADD 5050000, D0 the contents of D0 = 0002 + 0003 = 0005. Note that
in the above instruction, the $ symbol is used to represent hexadecimal
numbers by Motorola. Also, note that instructions using address registers
are not available for byte operations. In addition, in the 68000, the first
operand of a two-operand instruction is the source and the second operand
is the destination. Recall that in the 8086, the first operand is the destina-
tion while the second operand is the source.

### 5.5.2 ADDRESS REGISTER INDIRECT ADDRESSING

There are five different types of address register indirect mode. In the
register indirect mode, an address register contains the effective address.
For example, consider CLR (A1). If \([A1] = 5003000\), then after execution
of CLR (A1), the contents of memory location 5003000 will be cleared
to zero.

The postincrement address register indirect mode increments an ad-
dress register by 1 for byte, 2 for word, and 4 for long word after it is used.
For example, consider CLR.L (A0) + if \([A0] = 005000\), then after ex-
cution of CLR.L (A0) +, the contents of locations 005000 through
005004 are cleared to zero and \([A0] = 005004\). The postincrement mode
is typically used with memory address stored from LOW to HIGH memory
locations. For example, in order to clear 1000 words starting at memory
location 003000, the following instruction sequence can be used:

```
MOV W #1000, D0 ; Load length of data
INTO D0
MOVEA L #3000, AO ; Load starting address
INTO A0
REPEAT CLR.W (AO) + ; Clear a location
pointed to by AO and
increment AO by 2
SUBL D0, 1 ; Decrement D0 by 1
BNE REPEAT ; Branch to REPEAT if Z
             ; = 0, else go to next
             ; instruction
```

Note that in the above, CLR.W (AO)+ automatically points to the next
location by incrementing AO by 2 after clearing a memory location.

The predecrement address register indirect mode, on the other hand,
decrements an address register by 1 for byte, 2 for word, and 4 for long word before using a register. For example, consider CLR.W A0. If [A0] = 002004, then after execution of CLR.W A0, the content of A0 is first decremented by 2, that is, [A0] = 002002, and the 16-bit contents of memory location 002002 are then cleared to zero.

The predecrement mode is used with arrays stored from HIGH to LOW memory locations. For example, in order to clear 1000 words starting at memory location 4000 and below, the following instruction sequence can be used:

```
MOVE.W $210000, D0
MOVE.L #$4002, AO
REPEAT CLR W (AO)
SUBQ1, D0
BNE REPEAT
```

In the above, CLR.W (AO) first decrements A0 by 2 and then clears the location. Since the starting address is 004000, A0 must initially be initialized with 004000.

It should be pointed out that the predecrement and postincrement modes can be combined in a single instruction. A typical example is MOVE.W (AS), (A3). The two other address register modes provide accessing of the tables by allowing offsets and indices to be included to an indirect address pointer. The address register indirect with offset mode determines the effective address by adding a 16-bit signed integer to the contents of an address register. For example, consider MOVE.W $10 (AS), D3 in which the source operand is in address register indirect with offset mode. If [AS] = 0002000, [002001] = 0014, then after execution of MOVE.W $10 (AS), D3, register D3 will contain 0014.

The indexed register indirect with offset determines the effective address by adding an 8-bit signed integer and the contents of a register (data or address register) to the contents of an address (base) register. This mode is usually used when the offset from the base address register needs to be varied during program execution. The size of the index register can be a 16-bit integer or a 32-bit value.

5.5 ADDRESSING MODES

As an example, consider MOVE.W $10 (A4, D3.W), D4 in which the source is in the indexed register indirect with offset mode. Note that in this instruction A4 is the base register and D3.W is the 16-bit index register (sign extended to 32 bits). This register can be specified as 32 bits by using D3.L in the instruction, and D1.W is the 8-bit offset which is sign-extended to 32 bits. If [A4] = 00003000, [D3] = 002001, [003210] = 0024, then the above MOVE instruction will load 0024 into low 16 bits of register D4.

The address register indirect with offset mode can be used to access a single table where the offset (maximum 16 bits) can be the starting address of the table (fixed number) and the address register can hold the index number in the table to be accessed. Note that the starting address plus the index number provides the address of the element to be accessed in the table. For example, consider MOVE.W $3400 (A5), D1. If A5 contains 04, then this move instruction transfers the contents of 3404 (i.e., the fifth element, 0 being the first element) into low 16 bits of D1. The indexed register indirect with offset, on the other hand, can be used to access multiple tables, where the offset (maximum 9 bits) can be the element number to be accessed. The address register pointer can be used to hold the starting address of the table containing the lowest starting address, and the index register can be used to hold the difference between the starting address of the table being accessed and the table with the lowest starting address. For example, consider three tables with Table 1 starting at 002000, Table 2 at 003000, and Table 3 at 004000. Now, in order to transfer element 7 (0 being the first element) in Table 2 to the low 16 bits of register D0, the instruction MOVE.W $106 (A2, D1.W), D0 can be used, where [A2] = starting address of the table, with the lowest address = 002000, in this case and [D1.W] = difference between the starting address of the table being accessed and the starting address of the table, with the lowest address = 003000, = 002000, = 1000. Therefore, the above MOVE instruction will transfer the contents of address 003006, (seventh element in Table 2) to register D0.

5.5.3 ABSOLUTE ADDRESSING

In this mode, the effective address is part of the instruction. The 68000 has two absolute addressing modes: absolute short addressing in which a 16-bit address is used (the address is sign-extended to 32 bits before use) and absolute long addressing in which a 24-bit address is used. For example, consider ADD $2000, D2 as an example of absolute-short mode. If [$2000] = 001210, [D2] = 001010, then after execution of ADD $2000, D2, the address $2000 is sign-extended to 32 bits, whose low 24 bits are used as the address, and register D2 will therefore contain 002101.
5.5.4 PROGRAM COUNTER RELATIVE ADDRESSING

The 68000 has two program counter relative addressing modes: relative with offset, and relative with index and offset. In relative with offset, the effective address is obtained by adding the contents of the current PC with a sign-extended 16-bit displacement. This mode can be used when the displacement needs to be fied during program execution. Typical branch instructions such as BRQ, BRA, and BLE use relative mode with offset. This mode can also be used by some other instructions. For example, consider ADDW+320, D5 in which the source operand is relative to the offset mode. Note that typical assemblers use the symbol * to indicate offset. Now suppose that the current PC contents are 002000, the contents of 002030 are 0005, and the low 16 bits of D5 contain 0010. After execution of the ADDW instruction, D5 will contain 0015.

In relative with index and offset, the effective address is obtained by adding the contents of the current PC, a signed 8-bit displacement (sign-extended to 32 bits), and the contents of an index register (address or data register). The size of the index register can be 16 or 32 bits wide. For example, consider ADDW 00000012, [PC+002000] or [D0]. If [D2] = 00000012 and [002000] = 10010, after this ADDW instruction, [D2] will be 00110. This mode is used when the displacement needs to be changed during program execution.

5.5.5 IMMEDIATE DATA ADDRESSING MODE

There are two immediate modes available with the 68000. These are the immediate and quick immediate modes. In the immediate mode, the operand data are constant data, which is part of the instruction. For example, consider ADD #0005, D0. If [D0] = 0002, then after this ADD instruction, [D0] = 0002 + 0005 = 0007. Note that the # symbol is used by Motorola to indicate the immediate mode.

The quick immediate mode allows one to increment or decrement a register by a number from 0 to 7. For example, ADDQ 1, D0 increments the contents of D0 by 1. Note that the data is inherent in the op code with the op code length of one word (16-bit). Data 0 to 7 are represented by three bits in the op code.

5.5.6 IMPLIED ADDRESSING

There are two types of implied addressing modes: implicit and explicit. The instructions using the implicit mode do not require any operand, and registers such as PC, SP, or SR are implicitly referenced in these instructions. For example, RET returns from an exception routine to the main program by using implicitly the PC and SR. The JMP instruction, on the other hand, allows loading a value into the PC, although the PC is not explicitly defined in the instruction.

All 68000 addressing modes of Table 5.3 can further be divided into four functional categories as follows:

- Data Addressing Mode. An addressing mode is said to be a data addressing mode if it references data objects. For example, all 68000 addressing modes, except the address register direct mode, fall into this category.
- Memory Addressing Mode. An addressing mode that is capable of accessing a data item stored in the memory is classified as memory addressing mode. For example, the data and address register direct addressing modes cannot satisfy this definition.
- Control Addressing Mode. This refers to an addressing mode that has the ability to access a data item stored in the memory without the need to specify its size. For example, all 68000 addressing modes except the following are classified as control addressing modes:
  - Data register direct
  - Address register indirect
  - Address register indirect with postincrement
  - Address register indirect with predecrement
  - Immediate
- Alterable Addressing Mode. If the effective address of an addressing mode is written into, then that mode is called alterable addressing mode. For example, the immediate and the program counter relative addressing modes will not satisfy this definition.

The addressing modes are classified into the four functional categories as shown in Figure 5.5.
### 5.6 68000 INSTRUCTION SET

The 68000 instruction set contains 56 basic instructions. Table 5-4 lists them in alphabetical order. Table 5-5 lists those affecting the condition codes. The repertoire is very versatile and offers an efficient means to handle high-level language structures (such as arrays and linked lists). Note that in order to identify the operand size of an instruction, the following is placed after a 68000 mnemonic: B for byte, W or none for word, and L for long word. For example:

- **ADD.B**
  \[ \text{DD}, \text{D1} \rightarrow [\text{D1}]_{8} \leftarrow [\text{DD}]_{8} + [\text{D1}]_{8} \]
- **ADD.W or ADD.L**
  \[ \text{DD}, \text{D1} \rightarrow [\text{D1}]_{16} \leftarrow [\text{DD}]_{16} + [\text{D1}]_{16} \]

### TABLE 5.4
68000 Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Length (words)</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDC - (Ax), -(Ax)</td>
<td>B</td>
<td>1</td>
<td>(-[Ax]<em>{16} = -[Ax]</em>{16} + X \rightarrow [Ax]_{16})</td>
</tr>
<tr>
<td>ABDC Dy, Dx</td>
<td>B</td>
<td>1</td>
<td>([Dy]<em>{16} + [Dx]</em>{16} + X \rightarrow Dx_{16})</td>
</tr>
<tr>
<td>ADD (EA), (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>([EA]<em>{16} \rightarrow [EA]</em>{16})</td>
</tr>
<tr>
<td>ADDA (EA), An</td>
<td>W, L</td>
<td>2 or 3</td>
<td>([EA]<em>{16} \rightarrow An</em>{16})</td>
</tr>
<tr>
<td>ADDI #data, (EA)</td>
<td>B, W, L</td>
<td>2 or 3</td>
<td>([EA]<em>{16} \rightarrow [EA]</em>{16})</td>
</tr>
<tr>
<td>ADDQ #data, (EA)</td>
<td>B, W, L</td>
<td>2 or 3</td>
<td>([EA]<em>{16} \rightarrow [EA]</em>{16})</td>
</tr>
<tr>
<td>ADDX - (Ax), -(Ax)</td>
<td>B, W, L</td>
<td>1</td>
<td>(-[Ax]<em>{16} = -[Ax]</em>{16} + X \rightarrow [Ax]_{16})</td>
</tr>
<tr>
<td>ADDX Dy, Dx</td>
<td>B, W, L</td>
<td>1</td>
<td>([Dy]<em>{16} + [Dx]</em>{16} + X \rightarrow Dx_{16})</td>
</tr>
<tr>
<td>AND (EA), (EA)</td>
<td>B, W, L</td>
<td>2 or 3</td>
<td>([EA]<em>{16} \rightarrow [EA]</em>{16})</td>
</tr>
<tr>
<td>ANDI #data, (EA)</td>
<td>B, W, L</td>
<td>2 or 3</td>
<td>([EA]<em>{16} \rightarrow [EA]</em>{16})</td>
</tr>
<tr>
<td>ANDI #data, CCR</td>
<td>B</td>
<td>2</td>
<td>([data]<em>{16} \rightarrow [CCR]</em>{16})</td>
</tr>
<tr>
<td>ANDI #data, SR</td>
<td>W</td>
<td>2</td>
<td>([data]<em>{16} \rightarrow [SR]</em>{16})</td>
</tr>
<tr>
<td>ASL Dy, Dy</td>
<td>B, W, L</td>
<td>1</td>
<td>number of bits determined by ([Dy]_{16})</td>
</tr>
<tr>
<td>ASL #data, Dy</td>
<td>B, W, L</td>
<td>1</td>
<td>number of bits determined by ([Dy]_{16})</td>
</tr>
<tr>
<td>ASL (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>number of bits determined by ([EA]_{16})</td>
</tr>
<tr>
<td>ASR Dy, Dy</td>
<td>B, W, L</td>
<td>1</td>
<td>number of bits determined by ([Dy]_{16})</td>
</tr>
<tr>
<td>ASR #data, Dy</td>
<td>B, W, L</td>
<td>1</td>
<td>number of bits determined by ([Dy]_{16})</td>
</tr>
</tbody>
</table>
### TABLE 5.4 (continued)

#### 68000 Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Length (words)</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>![Diagram]</td>
</tr>
<tr>
<td>BCC d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if carry = 0; else next instruction</td>
</tr>
<tr>
<td>BCHC Dn, (EA)</td>
<td>B, L</td>
<td>1</td>
<td>(bit of [EA], specified by Dn) → Z</td>
</tr>
<tr>
<td>BCHC #data, (EA)</td>
<td>B, L</td>
<td>2</td>
<td>Same as BCHC Dn, [EA] except bit number is specified by immediate data</td>
</tr>
<tr>
<td>BCLR Dn (EA)</td>
<td>B, L</td>
<td>1</td>
<td>(bit of [EA]) → Z</td>
</tr>
<tr>
<td>BCLR #data, (EA)</td>
<td>B, L</td>
<td>2</td>
<td>0 → bit of [EA] specified by Dn</td>
</tr>
<tr>
<td>BCS d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if carry = 1; else next instruction</td>
</tr>
<tr>
<td>BEQ d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if Z = 1; else next instruction</td>
</tr>
<tr>
<td>BGE d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if greater than or equal; else next instruction</td>
</tr>
<tr>
<td>BGT d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if greater than; else next instruction</td>
</tr>
<tr>
<td>BHI d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if higher; else next instruction</td>
</tr>
<tr>
<td>BLE d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if less or equal; else next instruction</td>
</tr>
<tr>
<td>BLS d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if low or same; else next instruction</td>
</tr>
<tr>
<td>BLT d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if less than; else next instruction</td>
</tr>
<tr>
<td>BMI d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if N = 1; else next instruction</td>
</tr>
<tr>
<td>BNE d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if Z = 1; else next instruction</td>
</tr>
<tr>
<td>BPL d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if N = 0; else next instruction</td>
</tr>
<tr>
<td>BRA d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch always to PC + d</td>
</tr>
</tbody>
</table>

### TABLE 5.4 (continued)

#### 68000 Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Length (words)</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSET Dn, (EA)</td>
<td>B, L</td>
<td>1</td>
<td>bit of [EA] → Z</td>
</tr>
<tr>
<td>BSET #data, (EA)</td>
<td>B, L</td>
<td>2</td>
<td>Same as BSET Dn, [EA] except bit is specified by immediate data</td>
</tr>
<tr>
<td>BSR d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>PC → PC – d</td>
</tr>
<tr>
<td>BTST Dn, (EA)</td>
<td>B, L</td>
<td>1</td>
<td>[bit of [EA] specified by Dn] → Z</td>
</tr>
<tr>
<td>BTST #data, (EA)</td>
<td>B, L</td>
<td>2</td>
<td>Same as BTST Dn, [EA] except bit is specified by data</td>
</tr>
<tr>
<td>BVC d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if V = 0; else next instruction</td>
</tr>
<tr>
<td>BVS d</td>
<td>B, W</td>
<td>1 for B, 2 for W</td>
<td>Branch to PC + d if V = 1; else next instruction</td>
</tr>
<tr>
<td>CHK (EA), Dn</td>
<td>W</td>
<td>1</td>
<td>If Dn &lt; 0 or Dn &gt; [EA], then trap</td>
</tr>
<tr>
<td>CLR (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>0 → EA</td>
</tr>
<tr>
<td>CMP (EA), Dn</td>
<td>B, W, L</td>
<td>1</td>
<td>Dn – [EA] → Affect all condition codes except X</td>
</tr>
<tr>
<td>CMP (EA), An</td>
<td>W, L</td>
<td>1</td>
<td>An – [EA] → Affect all condition codes except X</td>
</tr>
<tr>
<td>CMPI #data, (EA)</td>
<td>B, W, L</td>
<td>2 for B, 3 for L</td>
<td>[EA] → data → Affect all flags except X-bit</td>
</tr>
<tr>
<td>CMPM (A) → (A)</td>
<td>B, W, L</td>
<td>1</td>
<td>[A] = [A] → Affect all flags except X; update A and A'</td>
</tr>
<tr>
<td>DBCC Dn, d</td>
<td>W</td>
<td>2</td>
<td>If condition false, i.e., C = 1, then Dn – 1 → Dn if Dn ≠ 1, then PC + d → PC; else PC + 2 → PC</td>
</tr>
<tr>
<td>DBCS Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition is C = 1</td>
</tr>
<tr>
<td>DBEQ Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition is Z = 1</td>
</tr>
<tr>
<td>DBH Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition is always false</td>
</tr>
<tr>
<td>DBGE Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition is greater or equal</td>
</tr>
<tr>
<td>DBGT Gn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition is greater than</td>
</tr>
<tr>
<td>DBHI Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition is high</td>
</tr>
<tr>
<td>DBLE Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition is less than or equal</td>
</tr>
<tr>
<td>DBLS Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition is low or same</td>
</tr>
</tbody>
</table>
TABLE 5.4 (continued)
68000 Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Length (words)</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLT Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition is less than</td>
</tr>
<tr>
<td>DBML Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition is N = 1</td>
</tr>
<tr>
<td>DBNE Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition Z = 0</td>
</tr>
<tr>
<td>DBPL Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition N = 1</td>
</tr>
<tr>
<td>DBT Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition is always true</td>
</tr>
<tr>
<td>DBVC Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition is V = 0</td>
</tr>
<tr>
<td>DBVS Dn, d</td>
<td>W</td>
<td>2</td>
<td>Same as DBCC except condition is V = 1</td>
</tr>
<tr>
<td>DIV (EA), Dn</td>
<td>W</td>
<td>1</td>
<td>Signed division</td>
</tr>
<tr>
<td>DIVU (EA), Dn</td>
<td>W</td>
<td>1</td>
<td>Dn ÷ [EA] → EA</td>
</tr>
<tr>
<td>EOR Dn, (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>Dn @ [EA] → EA</td>
</tr>
<tr>
<td>EORI #data, (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>data @ [EA] → EA</td>
</tr>
<tr>
<td>EORI #d6, CCR</td>
<td>B</td>
<td>2</td>
<td>d16 @ CCR → CCR</td>
</tr>
<tr>
<td>EORI #d16, SR</td>
<td>W</td>
<td>2</td>
<td>Rx ← Ry</td>
</tr>
<tr>
<td>EXT Dn</td>
<td>W, L</td>
<td>1</td>
<td>Extend sign bit of Dn from 8-bit to 16-bit or from 16-bit to 32-bit depending on whether operand size is B or W</td>
</tr>
<tr>
<td>JMP (EA)</td>
<td>Unsize</td>
<td>1</td>
<td>Jump to subroutine using address in operand</td>
</tr>
<tr>
<td>JSR (EA)</td>
<td>Unsize</td>
<td>1</td>
<td>Unconditional jump using address in operand</td>
</tr>
<tr>
<td>LIA (EA), An</td>
<td>L</td>
<td>1</td>
<td>An ← [EA]</td>
</tr>
<tr>
<td>LINK An, # -d</td>
<td>Unsize</td>
<td>2</td>
<td>No operation</td>
</tr>
<tr>
<td>LSL Dn, Dxy</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as LSL Dn, Dxy except immediate data specify the number of shifts from 0 to 7</td>
</tr>
<tr>
<td>LSL #data, Dxy</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as LSL Dxy, Dxy except immediate data specify the number of shifts from 0 to 7</td>
</tr>
<tr>
<td>LSL (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as LSL Dn, Dxy except left shift is performed only once</td>
</tr>
</tbody>
</table>

TABLE 5.4 (continued)
68000 Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Length (words)</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSR Dn, Dxy</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as LSR except immediate data specifies the number of shifts from 0 to 7</td>
</tr>
<tr>
<td>LSR (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as LSR, Dn, Dxy except the right shift is performed once only</td>
</tr>
<tr>
<td>MOVE (EA), (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>[EA] ← [EA]</td>
</tr>
<tr>
<td>MOVE (EA), CCR</td>
<td>W</td>
<td>1</td>
<td>[EA] ← CCR</td>
</tr>
<tr>
<td>MOVE (EA), SR</td>
<td>W</td>
<td>1</td>
<td>If S = 1, then [EA] ← SR, else TRAP</td>
</tr>
<tr>
<td>MOVE (EA), SRCR</td>
<td>W</td>
<td>1</td>
<td>SR ← [EA]</td>
</tr>
<tr>
<td>MOVE An, USP</td>
<td>L</td>
<td>1</td>
<td>If S = 1, then An → USP, else TRAP</td>
</tr>
<tr>
<td>MOYEM register</td>
<td>W, L</td>
<td>2</td>
<td>Register list → [EA]</td>
</tr>
<tr>
<td>MOYEM (EA), register list</td>
<td>W, L</td>
<td>2</td>
<td>[EA] ← register list</td>
</tr>
<tr>
<td>MOYEV Dn, d (EA)</td>
<td>W, L</td>
<td>2</td>
<td>Dn → [EA]</td>
</tr>
<tr>
<td>MOYEV d (Ay), Dn</td>
<td>W, L</td>
<td>2</td>
<td>d[Ay] → Dn</td>
</tr>
<tr>
<td>MOYEVQ #d8, Dn</td>
<td>L</td>
<td>1</td>
<td>Signed 16 x 16 multiplication [EA]16</td>
</tr>
<tr>
<td>MULS (EA)16, (Dn16)</td>
<td>W</td>
<td>1</td>
<td>Signed 16 x 16 multiplication [EA]16</td>
</tr>
<tr>
<td>MULU (EA)16, (Dn16)</td>
<td>W</td>
<td>1</td>
<td>Unsigned 16 x 16 multiplication [EA]16</td>
</tr>
<tr>
<td>NBEQ (EA)</td>
<td>B</td>
<td>1</td>
<td>0 ← [EA]10 - X → EA</td>
</tr>
<tr>
<td>NEQ (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>0 ← [EA] → EA</td>
</tr>
<tr>
<td>NECX (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>0 ← [EA] - X → EA</td>
</tr>
<tr>
<td>NOP</td>
<td>Unsize</td>
<td>1</td>
<td>No operation</td>
</tr>
<tr>
<td>NOT (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>[EA] ← EA</td>
</tr>
<tr>
<td>OR #data, (EA)</td>
<td>B, W, L</td>
<td>2 for B, W</td>
<td>data v [EA] → EA</td>
</tr>
<tr>
<td>OR #d8, CCR</td>
<td>B</td>
<td>2</td>
<td>d8 v CCR → CCR</td>
</tr>
</tbody>
</table>
### TABLE 5.4 (continued) 68000 Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Length (words)</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORL #d16, SR</td>
<td>W</td>
<td>2</td>
<td>If S = 1, then d16 v SR -&gt; SR; else TRAP</td>
</tr>
<tr>
<td>PEA (EA)</td>
<td>L</td>
<td>1</td>
<td>[EA][16 sign extend to 32 bits] -&gt; [SP]</td>
</tr>
<tr>
<td>RESET</td>
<td>Unsized</td>
<td>1</td>
<td>If S = 1, then assert RESET line; else TRAP</td>
</tr>
<tr>
<td>ROL Dx, Dy</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as ROL Dx, Dy except immediate data specifies number of times to be rotated from 0 to 7</td>
</tr>
<tr>
<td>ROL #data, Dy</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as ROL Dx, Dy except immediate data specifies number of times to be rotated from 0 to 7</td>
</tr>
<tr>
<td>ROL (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as ROL Dx, Dy except [EA] is rotated once</td>
</tr>
<tr>
<td>ROR Dx, Dy</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as ROR Dx, Dy except the number of rotates is specified by immediate data from 0 to 7</td>
</tr>
<tr>
<td>ROR #data, Dy</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as ROR Dx, Dy except the number of rotates is specified by immediate data from 0 to 7</td>
</tr>
<tr>
<td>ROR (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as ROR Dx, Dy except [EA] is rotated once</td>
</tr>
<tr>
<td>ROXL Dx, Dy</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as ROXL Dx, Dy except immediate data specifies number of rotates from 0 to 7</td>
</tr>
<tr>
<td>ROXL #data, Dy</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as ROXL Dx, Dy except immediate data specifies number of rotates from 0 to 7</td>
</tr>
<tr>
<td>ROXL (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as ROXL Dx, Dy except [EA] is rotated once</td>
</tr>
</tbody>
</table>

### TABLE 5.4 (continued) 68000 Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Length (words)</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROXR Dx, Dy</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as ROXR Dx, Dy except immediate data specifies number of rotates from 0 to 7</td>
</tr>
<tr>
<td>ROXR (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>Same as ROXR Dx, Dy except [EA] is rotated once</td>
</tr>
<tr>
<td>RTE</td>
<td>Unsized</td>
<td>1</td>
<td>If S = 1, then [SP]+ -&gt; SR; [SP]++ -&gt; PC, else TRAP</td>
</tr>
<tr>
<td>RTR</td>
<td>Unsized</td>
<td>1</td>
<td>[SP] + CC; [SP] + PC</td>
</tr>
<tr>
<td>RTS</td>
<td>Unsized</td>
<td>1</td>
<td>[SP] + PC</td>
</tr>
<tr>
<td>SBED (A), (AX)</td>
<td>B</td>
<td>1</td>
<td>[AX][10] -&gt; [AX][10] -&gt; [AX]</td>
</tr>
<tr>
<td>SBED Dy, Dx</td>
<td>B</td>
<td>1</td>
<td>[Dy][10] -&gt; [Dy][10] -&gt; [Dx]</td>
</tr>
<tr>
<td>SCC (EA)</td>
<td>B</td>
<td>1</td>
<td>If C = 0, then 1s -&gt; [EA] else 0s -&gt; [EA]</td>
</tr>
<tr>
<td>SCS (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except the condition is C = 0</td>
</tr>
<tr>
<td>SEQ (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except if Z = 0</td>
</tr>
<tr>
<td>SF (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except condition is always false</td>
</tr>
<tr>
<td>SCE (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except if greater than or equal</td>
</tr>
<tr>
<td>SGT (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except if greater than</td>
</tr>
<tr>
<td>SHI (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except if high</td>
</tr>
<tr>
<td>SLE (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except if less or equal</td>
</tr>
<tr>
<td>SLS (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except if low or same</td>
</tr>
<tr>
<td>SLT (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except if less than</td>
</tr>
<tr>
<td>SMI (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except if N = 0</td>
</tr>
<tr>
<td>SNE (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except if Z = 0</td>
</tr>
<tr>
<td>SPL (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except if N = 0</td>
</tr>
<tr>
<td>ST (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except condition always true</td>
</tr>
<tr>
<td>STOP #data</td>
<td>Unsized</td>
<td>2</td>
<td>If S = 1, then data -&gt; SR and stop; TRAP if executed in user mode</td>
</tr>
<tr>
<td>SUB (EA), (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>EA - [EA] -&gt; EA</td>
</tr>
<tr>
<td>SUBA (EA), An</td>
<td>W, L</td>
<td>1</td>
<td>An - [EA] -&gt; An</td>
</tr>
</tbody>
</table>
TABLE 5.4 (continued)
68000 Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Length (words)</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBQ #data, (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>[EA] = data → EA</td>
</tr>
<tr>
<td>SUBX – (Ay), – (Ax)</td>
<td>B, W, L</td>
<td>1</td>
<td>–(Ay) – X → [Ax]</td>
</tr>
<tr>
<td>SUBX Dy, Dx</td>
<td>B, W, L</td>
<td>1</td>
<td>Dy – Dx – X → Dx</td>
</tr>
<tr>
<td>SVC (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except if V = 0</td>
</tr>
<tr>
<td>SVS (EA)</td>
<td>B</td>
<td>1</td>
<td>Same as SCC except if V = 1</td>
</tr>
<tr>
<td>SWAP Dn</td>
<td>W</td>
<td>1</td>
<td>Dn [31:16] → Dn [15:0]</td>
</tr>
<tr>
<td>TAS (EA)</td>
<td>B</td>
<td>1</td>
<td>[EA] tested; N and Z are affected accordingly; 1 = bit 7 of [EA]</td>
</tr>
<tr>
<td>TRAP #vector</td>
<td>Unsize</td>
<td>1</td>
<td>PC → -[SSP], SR → -[SSP], (vector) → PC; 16 TRAP vectors are available</td>
</tr>
<tr>
<td>TRAPV</td>
<td>Unsize</td>
<td>1</td>
<td>If V = 1, then TRAP</td>
</tr>
<tr>
<td>TST (EA)</td>
<td>B, W, L</td>
<td>1</td>
<td>[EA] – 0 → condition codes affected; no result provided</td>
</tr>
<tr>
<td>UNLK #n</td>
<td>Unsize</td>
<td>1</td>
<td>An → SP; [SP] + → An</td>
</tr>
</tbody>
</table>

All 68000 instructions may be classified into eight groups, as follows:

i) Data movement instructions
ii) Arithmetic instructions
iii) Logical instructions
iv) Shift and rotate instructions
v) Bit manipulation instructions
vi) Binary-coded decimal instructions
vii) Program control instructions
viii) System control instructions

5.6.1 DATA MOVEMENT INSTRUCTIONS

These instructions allow data transfers from register to register, register to memory, memory to register, and memory to memory. In addition,

<table>
<thead>
<tr>
<th>Instruction</th>
<th>X</th>
<th>N</th>
<th>Z</th>
<th>Y</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCD</td>
<td>+</td>
<td>U</td>
<td>+</td>
<td>U</td>
<td>-</td>
</tr>
<tr>
<td>ADD, ADDI, ADDQ, ADDX</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>AND, ANDI</td>
<td>-</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>+</td>
</tr>
<tr>
<td>ASL, ASR</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>BCHG, BCLR, BSET, BTST</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CHK</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CLR</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>CMP CMPA, CMPI, CMPPM</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>DIVS, DIVU</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>EOR, EORI</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>EXT</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LSL, LSR</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MOVE (EA), (EA)</td>
<td>-</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MOVE TO CC</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>MOVE TO SR</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>MOVEQ</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MULS, MULU</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NBCD</td>
<td>-</td>
<td>U</td>
<td>+</td>
<td>U</td>
<td>+</td>
</tr>
<tr>
<td>NEG, NEGX</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NOT, OR, ORI</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ROL, ROR</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ROXL, ROXR</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RLE, RTR</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SBCD</td>
<td>+</td>
<td>U</td>
<td>+</td>
<td>U</td>
<td>+</td>
</tr>
<tr>
<td>STOP</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SUB, SUBI, SUBQ, SUBX</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SWAP</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TAS</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TST</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: +, affected; —, not affected; U, undefined.

There are also special data movement instructions such as MOVELM (Move multiple registers). Typically, byte, word, or long-word data can be transferred. Table 5.6 lists the 68000 data movement instructions.
### 5.6 68000 Instruction Set

#### TABLE 5.6.1 (continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVEM reg list, (EA) W, L or (EA), reg list</td>
<td>Specified registers are transferred to or from consecutive memory locations starting at the location specified by the effective address.</td>
<td></td>
</tr>
<tr>
<td>MOVEP Dn, d (Ay) or Dn, d (Ay)</td>
<td>Two (W) or four (L) bytes of data are transferred between a data register and alternate bytes of memory.</td>
<td></td>
</tr>
</tbody>
</table>

#### 5.6.1a MOVE Instructions

The format for the basic MOVE instruction is MOVE.P (EA), (EA),
where \( S = L, W, \) or B. \( (EA) \) can be a register or memory location depending on the addressing mode used. Consider MOVE.B D2, D0
which uses a data register direct mode for both the source and destination.
Now if $[D1] = 03_{H} → [D0] = 01_{H}$, then after execution of this MOVE instruction $[D2] = 03_{H}$ and $[D0] = 03_{H}$.

There are several variations of the MOVE instruction. For example, MOVE.W CCR, (EA) moves the content of the low-order byte of the SR, i.e., CCR, to the low-order byte of the destination operand, and the upper byte of SR is considered as zero. The source operand is a word. Similarly, MOVE.W (EA), CCR moves a 16-bit immediate number or low-order 8-bit data from a memory location or register into the condition code register; the upper byte is ignored. The source operand is a word. Data can also be transferred between (EA) and SR or USP using the following instructions:

\[
\begin{align*}
\text{MOVE.W (EA), } & \text{SR} \\
\text{MOVE.W SR, (EA)} & \\
\text{MOVE.L USP, } & \text{An} \\
\text{MOVE.L An, USP} & \\
\end{align*}
\]

MOVEA.W or L (EA), An can be used to load an address into an address register. Word size source operands are sign-extended to 32 bits. Note that (EA) is obtained using an addressing mode. As an example, MOVEA.W $\#80000$, A0 moves the 16-bit word $8000_{H}$ into the low 16 bits of A0 and then sign-extends $8000_{H}$ to the 32-bit number $FFFF8000_{H}$.

Note that sign extension means extending bit 15 of $8000_{H}$ from bit 16 through bit 31. As mentioned, sign extension is required when an arithmetic operation between two signed binary numbers of different sizes is performed. (EA) in MOVEA can use all addressing modes. MOVEM instruction can be used to PUSH or POP multiple registers to or from the stack. For example, MOVEM.L D0-D7/A0-A6, (SP) saves the contents of all of the 8 data registers and 7 address registers in the stack. This instruction stores address registers in the order A6-A0 first, followed by data registers in the order D7-D0, regardless of the order in the list. MOVEM.L (SP)+, D0-D7/A0-A6 restores the contents of the registers in the order D0-D7/A0-A6 regardless of the order in the list. MOVEM instruction can also be used to save a set of registers in memory. In addition to the above predecrement and postincrement modes for the effective address, MOVEM instruction allows all the control modes. If the effective address is in one of the control modes, such as absolute short, then the registers are transferred starting at the specified address and up through higher addresses. The order of transfer is from D0 to D7 and then from A0 to A7. For example, MOVEM.W A4/D1/D3/A0-A2, $\#8000$ transfers the low 16-bit contents of D1, D3, A0, A1, A2, and A4 to locations $\#8000$, $\#8002$, $\#8004$, $\#8006$, $\#8008$, and $\#800A$, respectively.

The MOVEQ. L #D8, Dn moves the immediate 8-bit data into the low byte of Dn. The 8-bit data are then sign-extended to 32 bits. This is a one-word instruction. For example, MOVEQ. L #FF, D0 moves $FFFFFF$ into D0. In order to transfer data between the 68000 data registers and 68000 (8-bit) peripherals, the MOVEP instruction can be used. This instruction transfers two to four bytes of data between a data register and alternate byte locations in memory, starting at the location specified in increments of 2.

Register indirect with displacement is the only addressing mode used with this instruction. If the address is even, all the transfers are made on the high-order half of the data bus. The high-order byte from the register is transferred first and the low-order byte is transferred last. For example, consider MOVEP.L $\#00050$, D0, D10. If $[A10] = 06003000_{H}$, then after the execution of the above MOVEP instruction, D10 will contain $01003000_{H}$.

5.6.1.b EXG and SWAP Instructions

The EXG Rs, Rx instruction exchanges the 32-bit contents of Rx with that of the Rs. The exchange is between two data registers, two address registers, or between an address and a data register. The EXG instruction exchanges only 32-bit-long words. The data size (L) does not have to be specified after the EXG instruction since this instruction has only one data size. No flags are affected.

The SWAP Dn instruction, on the other hand, exchanges the low 16 bits of Dn with the high 16 bits of Dn. All condition codes are affected.

5.6.1.c LEA and PEA Instructions

The LEA (EA), An moves an effective address (EA) into the specified address register. The (EA) can be calculated based on the addressing mode of the source. For example, LEA $\#00456074$, A1 moves $\#00456074$ to A1. This instruction is equivalent to MOVEA.L $\#00456074$, A1. Note that $00456074_{H}$ is contained in the PC.

LEA instruction is very useful when address calculation is desired during program execution. (EA) in LEA specifies the actual data to be loaded into An, whereas (EA) in MOVEA specifies the address of actual data. For example, consider LEA $\#06$, (A2, D5.W), A0. If $[A2] = 00003000_{H}$, then the LEA instruction moves $00003000_{H}$ into A0. On the other hand, MOVEA $\#06$, (A2.D5.W), A0 moves the contents of $00003000_{H}$ into A0. Therefore, it is obvious that if address calculation is required, the instruction LEA is very useful.

The PEA (EA) computes an effective address and then pushes it onto the stack. This instruction can be used when the 16-bit address used in
absolute short mode is required to be pushed onto the stack. For example, consider PEA S8000 in the user mode. If [USP] = 500005004, then S8000 is sign-extended to 32 bits and pushed to stack. The low-order 16 bits (S8000) are pushed at 50005002, and the high-order 16 bits (FFFF) are pushed at 00500016.

5.6.1.d LINK and UNLK Instructions
Before calling a subroutine, the main program quite often transfers values of certain parameters to the subroutine. It is convenient to save these variables onto the stack before calling the subroutine. These variables can then be read from the stack and used by the subroutine for computations. The 68000 LINK and UNLK instructions are used for this purpose. In addition, the 68000 LINK instruction allows one to reserve temporary storage for the local variables of a subroutine. This storage can be accessed as needed by the subroutine and be released using UNLK before returning to the main program. The LINK instruction is usually used at the beginning of a subroutine to allocate stack space for storing local variables and parameters for nested subroutine calls. The UNLK instruction is usually used at the end of subroutine before the RETURN instruction to release the local area and restore the stack pointer contents so that it points to the return address.

The LINK A0, #-100 instruction causes the current contents of the specified A0 to be pushed onto the system (user or supervisor) stack. The updated SP contents are then loaded into A0. Finally, a sign-extended 2's complement displacement value is added to the SP. No flags are affected. For example, consider LINK A0, #-100. If [A0] = 00005100 [USP] = 00002164 then after execution of the LINK instruction the situation shown in Figure 5.6 occurs.

This means that after the LINK instruction, [A0] = 00005100 is pushed onto the stack, the [updated USP] = 002100 is loaded into A0. USP is then loaded with 002000 and, therefore, 100 locations are allocated to subroutine at the beginning of which the above LINK instruction can be used. Note that A0 cannot be used in the subroutine.

The UNLK instruction at the end of this subroutine before the RETURN instruction releases the 100 locations and restores the contents of A0 and USP to those prior to using the LINK instruction. For example, UNLK A0 will load [A0] = 00002100 into USP, the two stack words 00005100 into A0, and USP is then incremented by 4 to contain 00002104. Therefore, the contents of A0 and USP prior to using the LINK are restored. In the above example, after execution of the LINK, addresses 001999 and below can be used as the stack. One hundred locations starting at 002000 and above can be reserved for storing the local variables of the subroutine. These variables can then be accessed with an address register such as A0 as a base pointer using the address register indirect with displacement mode such as MOVES d(A0), D2 for read and MOVES d(A0), d(A0) for write.

5.6.2 ARITHMETIC INSTRUCTIONS
These instructions allow:
- 8-, 16-, or 32-bit additions and subtractions
- 16-bit by 16-bit multiplication (both signed and unsigned)
- Compare, clear, and negate instructions
- Extended arithmetic instructions for performing multiprecision arithmetic
- Test (TST) instruction for comparing the operand with zero
- Test and set (TAS) instruction which can be used for synchronization in multiprocessor system

The 68000 arithmetic instructions are summarized in Table 5.7.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD (E1, EA)</td>
<td>B, W, L</td>
<td>[EA] + [EA] → EA</td>
</tr>
<tr>
<td>ADDI #data, (EA)</td>
<td>B, W, L</td>
<td>[EA] + data → EA</td>
</tr>
<tr>
<td>ADDQ #d8, (EA)</td>
<td>B, W, L</td>
<td>[EA] + d8 → EA</td>
</tr>
</tbody>
</table>

d8 can be an integer from 0 to 7
### Table 5.7 (continued) 68000 Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDA (EA), An</td>
<td>W, L</td>
<td>An + [EA] → An</td>
</tr>
<tr>
<td>SUBA (EA), An</td>
<td>W, L</td>
<td>An - [EA] → An</td>
</tr>
<tr>
<td>MULS (EA), Dn</td>
<td>W</td>
<td>[Dn]16 * [EA]16 → [Dn]32 (signed multiplication)</td>
</tr>
<tr>
<td>MULU (EA), Dn</td>
<td>W</td>
<td>[Dn]16 * [EA]16 → [Dn]32 (unsigned multiplication)</td>
</tr>
<tr>
<td>DIVS (EA), Dn</td>
<td>W</td>
<td>[Dn]32 / [EA]16 → [Dn]32 (signed division, high word of Dn contains remainder and low word of Dn contains quotient)</td>
</tr>
<tr>
<td>DIVU (EA), Dn</td>
<td>W</td>
<td>[Dn]32 / [EA]16 → [Dn]32 (unsigned division, remainder is in high word of Dn and quotient is in low word of Dn)</td>
</tr>
</tbody>
</table>

#### Compare, Clear, and Negate Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP (EA), Dn</td>
<td>B, W, L</td>
<td>Dn - [EA] → No result; affects flags</td>
</tr>
<tr>
<td>CMPI #data, (EA)</td>
<td>B, W, L</td>
<td>[EA] - data → No result; affects flags</td>
</tr>
<tr>
<td>CMPM (Ay) +, (Ax) +, (Ay)</td>
<td>B, W, L</td>
<td>[Ax] + [Ay] → No result; affects flags; Ax and Ay are incremented depending on operand size</td>
</tr>
<tr>
<td>CLR (EA)</td>
<td>B, W, L</td>
<td>0 → EA</td>
</tr>
<tr>
<td>NEG (EA)</td>
<td>B, W, L</td>
<td>0 - [EA] → EA</td>
</tr>
</tbody>
</table>

#### Extended Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDX Dy, Dx</td>
<td>B, W, L</td>
<td>Dx + Dy + X → Dx</td>
</tr>
<tr>
<td>ADDX - (Ay), - (Ax)</td>
<td>B, W, L</td>
<td>-(Ax) + -(Ay) + X →</td>
</tr>
</tbody>
</table>

### Table 5.7 (continued) 68000 Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXT Dn</td>
<td>W, L</td>
<td>If size is W, then sign-extend low byte of Dn to 16 bits; if size is L, then sign extend low 16 bits of Dn to 32 bits</td>
</tr>
<tr>
<td>NEGX (EA)</td>
<td>B, W, L</td>
<td>0 - [EA] - X → EA</td>
</tr>
<tr>
<td>SUBX Dy, Dx</td>
<td>B, W, L</td>
<td>Dx - Dy - X → Dx</td>
</tr>
<tr>
<td>SUBX - (Ay), - (Ax)</td>
<td>B, W, L</td>
<td>-(Ax) - -(Ay) - X →</td>
</tr>
</tbody>
</table>

#### Test Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TST (EA)</td>
<td>B, W, L</td>
<td>[EA] - 0 → flags affected</td>
</tr>
</tbody>
</table>

#### Test and Set Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAS (EA)</td>
<td>B</td>
<td>If [EA] = 0, then set Z = 1; else Z = 0, N = 1 and then always set bit 7 of [EA] to 1</td>
</tr>
</tbody>
</table>

Note: If [EA] in the ADD or SUB instruction is an address register, the operand length is WORD or LONG WORD. [EA] in any instruction is calculated using the addressing mode used. All instructions except ADDA and SUBA affect condition codes.

- Source [EA] in the above ADDA and SUBA can use all modes.
- Destination [EA] in ADDI and SUBI can use all modes except An relative and immediate.
- Destination [EA] in ADDQ and SUBQ can use all modes except relative and immediate.
- [EA] in all multiplication and division instructions can use all modes except An.
- Source [EA] in CMP and CMPI instructions can use all modes.
- Destination [EA] in CMPI can use all modes except An, relative, and immediate.
- [EA] in CLR and NEG can use all modes except An, relative, and immediate.
- [EA] in NEG can use all modes except An, relative, and immediate.
- [EA] in TST can use all modes except An, relative, and immediate.
- [EA] in TAS can use all modes except An, relative, and immediate.
5.6.2 Addition and Subtraction Instructions

Consider ADD.W $245000, D0. If $245000 = 0101, 0000, and $D0 = 1004, 0000, then after execution of this ADD instruction, the low 32 bits of D0 will contain 3018, 0000.

ADDI instruction can be used to add immediate data to register or memory location. The immediate data follows the instruction word. For example, consider ADDIW $500000, $500000. If $500000 = 0000, 0000, then after execution of this ADDI instruction, memory location 500000, will contain 1062, 0000. ADDQ, on the other hand, adds a number from 0 to 7 to the register or memory location in the destination operand. This instruction occupies 16 bits and the immediate data 0 to 7 is specified by 3 bits in the instruction word. For example, consider ADDQ #044, D0. If $D0 = 0004, 0000, then after execution of this ADDQ, low byte of register D0 will contain 06, 0000.

For ADD or SUB, if the destination is Dn, then the source register (EA) can use any mode; if the destination is a memory location (all modes except Dn, A,, An, immediate), then the source must be Dn.

All subtraction instructions subtract source from destination. For example, consider SUB.W D0, $500000. If $D0 = 0000, 0000, and $500000 = 1000, 0000, then after execution of this SUB instruction, memory location 500000, will contain 2002, 0000.

Consider SUB.W $500000, D0. If $D0 = 1000, 0000, then after execution of this SUB instruction, D0 will contain 0000, 0000. Note that the same result can be obtained by using a SUB.W $4, D0. However, in this case, the data is inherent in the instruction word.

5.6.2.b Multiplication and Division Instructions

The 68000 instruction set includes both signed and unsigned multiplication of integer numbers.

- MULS (EA), Dn multiplies two 16-bit signed numbers and provides a 32-bit result. For example, consider MULS #2, D0. If $D0 = 0004, 0000, then after this MULS, D0 will contain the 32-bit result FFFFD9, 0000, which is –8 in decimal. MULU (EA), Dn, on the other hand, performs unsigned multiplication. Consider MULU (A), D2. If $A = 00000000, [0000] = 0000, 0000, and $D2 = 0000, 0000, then after this MULU, D2 will contain 32-bit result 00000000.

- Consider DIVS #4, D0. If $D0 = –9, 0000, then after this DIVS, register D0 will contain FFFFFFFF, 0000.

Note that in 68000, after DIVS, the sign of the remainder is always the same as the dividend unless the remainder is equal to zero. Therefore, in the example above, since the dividend is negative (–9), the remainder is negative (–1). Also, division by zero causes an internal interrupt automatically. A service routine can be written by the user to indicate an error, N = 1 if the quotient is negative, and V = 1 if there is an overflow. The DIVU instruction is the same as the DIVS instruction except that the division is unsigned. For example, consider DIVU #2, D0. If $D0 = 11, 0000, 0000, then after execution of this DIVU, register D0 contains:

\[
\begin{array}{c|c|c}
D & 0001 & 0001 \\
16-bit remainder & 1062 & 0000 \\
16-bit quotient & 06 & 0000 \\
\end{array}
\]

As with the DIVS, division by zero using DIVU causes trap. Also, V = 1 if there is an overflow.

5.6.2.c Compare, Clear, and Negate Instructions


- CLEAR D3 clears all 32 bits of D3 to zero.

- Consider NEG.W (A1). If $[A1] = 00500000, [500000] = 0000, then after this NEG instruction, the low 16 bits of location 500000, will contain FFFFE9, 0000.

5.6.2.d Extended Arithmetic Instructions

ADDX and SUBX instructions can be used in performing multiple precision arithmetic since there are no ADDC (add with carry) or SUBB (subtract with borrow) instructions. For example, in order to perform a 64-bit addition, the following two instructions can be used:
5.6 68000 Instruction Set

(a address strobe) pin which can be used to lock out other processors from accessing the semaphore.

Let us explain the application of the TAS instruction. TAS(EA) is usually used to synchronize two processors in multiprocessor data transfers. For example, consider two 68000-based microcomputers with shared RAM shown in Figure 5.7.

Suppose that it is desired to transfer the low byte of D0 from processor 1 to the low byte of D2 in processor 2. A memory location, namely, TRDATA, can be used to accomplish this. First, processor 1 can execute the TAS instruction to test the byte in the shared RAM with address TEST for zero value. If it is zero, the processor 1 can be programmed to move the low byte of D0 into location TRDATA in the shared RAM. The processor 2 can then execute an instruction sequence to move the contents of TRDATA from the shared RAM into the low byte of D2. The following instruction sequence will accomplish this:

Processor 1 Routine

Proc 1 TAS TEST
BNE Proc 2
MOVE.B D0, TRDATA
CLR.B TEST

Processor 2 Routine

Proc 2 TAS TEST
BNE Proc 1
MOVE.B TRDATA, D2
CLR.B TEST

Note that in the above, TAS TEST checks the byte addressed by TEST for zero. If [TEST] = 0, then Z is set to one; otherwise Z = 0 and N = 1. After this, bit 7 of [TEST] is set to 1. Note that a zero value of TEST indicates that the shared RAM is free for use and the Z bit indicates this after the TAS is executed. In each of the above instruction sequences, after a data transfer using the MOVE instruction, TEST is cleared to zero so that the shared RAM is free for use by the other processor. Note that bit 7 of memory location TEST is called the semaphore.

In order to avoid testing of the TEST byte simultaneously by two pro-

---

FIGURE 5.7 Two 68000s interfaced via shared RAM.
5.6.3 LOGICAL INSTRUCTIONS

These instructions include logical OR, EOR, AND, and NOT as shown in Table 5.8.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDI #data, (EA)</td>
<td>B, W, L</td>
<td>[EA] AND #data → EA; [EA] cannot be address register</td>
</tr>
<tr>
<td>ANDI #data8, CCR</td>
<td>B</td>
<td>CCR AND #data → CCR</td>
</tr>
<tr>
<td>ANDI #data16, SR</td>
<td>W</td>
<td>SR AND #data → SR</td>
</tr>
<tr>
<td>EOR On, (EA)</td>
<td>B, W, L</td>
<td>Dh + [EA] → EA; [EA] cannot be address register</td>
</tr>
<tr>
<td>EORI #data, (EA)</td>
<td>B, W, L</td>
<td>[EA] OR #data → EA; [EA] cannot be address register</td>
</tr>
<tr>
<td>NOT (EA)</td>
<td>B, W, L</td>
<td>One's complement of [EA] → EA</td>
</tr>
<tr>
<td>ORI #data, (EA)</td>
<td>B, W, L</td>
<td>[EA] OR #data → [EA]; [EA] cannot be address register</td>
</tr>
<tr>
<td>ORI #data8, CCR</td>
<td>B</td>
<td>CCR OR #data8 → CCR</td>
</tr>
<tr>
<td>ORI #data16, SR</td>
<td>W</td>
<td>SR OR #data → SR</td>
</tr>
</tbody>
</table>

- Source [EA] in AND and OR can use all modes except An.
- Destination [EA] in AND or OR or EOR can use all modes except Dn, An, relative, and immediate.
- Destination [EA] in ANDI, ORI, and EORI can use all modes except An, relative, and immediate.
- [EA] in NOT can use all modes except An, relative, and immediate.
- Consider AND, W D2, D6. If [D2] = 0006H and [D6] = 000FH, then after execution of this AND, the low 16 bits of D6 will contain 00016.
- Consider ANDL #80, CCR. If [CCR] = 0F16, then after this ANDI, register CCR will contain 0616.
- Consider EOR, W D2, D6. If [D2] = 0000H and [D6] = 0006H, then after execution of this EOR, register D6 will contain 0000H, and D2 will remain unchanged at 000F6.
- Consider NOT, B D0. If [D0] = 04H, then after execution of this NOT instruction, the low byte of D0 will contain FBH.
- Consider ORI #1008, SR. If [SR] = A01H, then after execution of this ORI, register SR will contain B019H. Note that this is a privileged instruction since the high byte of SR containing the control bits is changed and therefore can only be executed in the supervisor mode.

5.6.4 SHIFT AND ROTATE INSTRUCTIONS

The 68000 shift and rotate instructions are listed in Table 5.9.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASL, Dx, Dy</td>
<td>B, W, L</td>
<td>Shift [Dy] by the number of times to left specify in Dx; the low 6 bits of Dx specify the number of shifts from 0 to 63</td>
</tr>
<tr>
<td>ASL #data, Dn</td>
<td>B, W, L</td>
<td>Same as ASL, Dx, Dy except that the number of shifts is specified by immediate data from 0 to 7</td>
</tr>
<tr>
<td>ASL (EA)</td>
<td>B, W, L</td>
<td>[EA] is shifted one bit to left; the most significant of [EA] goes to x and c, and zero moves into the least significant bit</td>
</tr>
<tr>
<td>ASR, Dx, Dy</td>
<td>B, W, L</td>
<td>Arithmetically shift [Dy] to the right by rotating the sign bit; the low 6 bits of Dx specify the number of shifts from 0 to 63</td>
</tr>
<tr>
<td>ASR #data, Dn</td>
<td>B, W, L</td>
<td>Same as above except the number of shifts is from 0 to 7</td>
</tr>
</tbody>
</table>
### TABLE 5.9 (continued)

#### 68000 Shift and Rotate Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR (EA)</td>
<td>B, W, L</td>
<td>Same as above except [EA] is shifted once to the right</td>
</tr>
<tr>
<td>LSL Dx, Dy</td>
<td>B, W, L</td>
<td></td>
</tr>
<tr>
<td>LSL #data, Dn</td>
<td>B, W, L</td>
<td>Same as above except the number of shift is specified by immediate data from 0 to 7</td>
</tr>
<tr>
<td>LSL [EA]</td>
<td>B, W, L</td>
<td>[EA] is shifted one bit to the left</td>
</tr>
<tr>
<td>LSR Dx, Dy</td>
<td>B, W, L</td>
<td>Same as LSL Dx, Dy except the rotate is to the right</td>
</tr>
<tr>
<td>LSR #data, Dn</td>
<td>B, W, L</td>
<td>Same as LSL #data, Dn except shift is to the right by immediate data from 0 to 7</td>
</tr>
<tr>
<td>ROL Dx, Dy</td>
<td>B, W, L</td>
<td>Same as above except that the immediate data specify that [Dy] to be shifted from 0 to 7</td>
</tr>
<tr>
<td>ROL #data, Dn</td>
<td>B, W, L</td>
<td></td>
</tr>
<tr>
<td>ROR Dx, Dy</td>
<td>B, W, L</td>
<td>[EA] is rotated once to the left</td>
</tr>
<tr>
<td>ROR #data, Dn</td>
<td>B, W, L</td>
<td>Same as ROL #data, Dn except the shift is to the right by immediate data from 0 to 7</td>
</tr>
</tbody>
</table>

Note: [EA] in ASL, ASR, LSL, LSR, ROL, ROR, ROXL, and ROXR can use all modes except Dn, An, relative, and immediate.

- All the instructions in Table 5.9 affect N and Z flags according to the result. V is reset to zero except for ASL.
- Note that in the 68000 there is no true arithmetic shift left instruction. In true arithmetic shifts, the sign bit of the number being shifted is retained. In the 68000, the instruction ASL does not retain the sign bit, whereas the instruction ASR retains the sign bit after performing the arithmetic shift operation. Consider ASLW D3, D0. If \( [D3]_{low} = [D3]_{high} = 0001 \), \( [D0]_{low} = 87FF \), then after this ASL instruction \( [D0]_{low} = FF\), \( [D0]_{high} = 00 \), \( C = 0 \), and \( X = 0 \). Note that the sign of the contents of D0 is changed from 1 to 0, and therefore the overflow is set. ASL sets the overflow bit to indicate sign change during the shift. In the example, the sign bit of D0 is changed after
shifting. A 3 times. ASR, on the other hand, retains the sign bit. For example, consider ASR W #2, D0. If [D5] = 8FE2, then after this ASR, the low 16 bits of [D5] = E3F8, C = 1, and X = 1. Note that the sign bit is retained. ASL (EA) or ASR (EA) shifts (EA) one bit to the left or right, respectively. For example, consider ASL W (A2). If [A2] = 000004000, and [04000] = F000, then after execution of the ASL, [004000] = F000, X = 1, and C = 1. On the other hand, ASR W (A2), memory location 004000, will contain 004000, C = 0, and X = 0. Note that only memory alterable modes are allowed for (EA). Also, only 16-bit operands are allowed for (EA) when the destination is memory location.

- LSL and ASL instructions are the same in the 68000 except that with the ASL, V is set to 1 if there is a sign change of the number during the shift.
- Consider LSR.W #0002, D0. If [D0] = F000, then after the LSR, [D0] = 3C00, X = 0, and C = 0.
- Consider ROL.B #02, D2. If [D2] = 0110, and C = 1, then after execution of the ROL, the low byte of [D2] = C7, and C = 0. On the other hand, with [D2] = 0110, and C = 1, consider ROR.B #02, D2. After execution of this ROR, register D2 will contain 0C, and C = 0.
- Consider ROXL.W D2, D1. If [D2] = 0000, [D1] = F201, C = 0, and X = 1, then the low 16 bits after execution of this ROXL are [D1] = 9000, C = 1, and X = 1.

5.6.5 BIT MANIPULATION INSTRUCTIONS

The 68000 has four bit manipulation instructions, and these are listed in Table 5.10.

- In all the above instructions, the 1's complement of the specified bit is reflected in the Z flag. The specified bit is then 1's complemented, cleared to zero, set to one, or unchanged by BCHG, BCLR, BSET, or BTST, respectively. In all the instructions in Table 5.10, if [EA] is Dn, then length of Dn is 32 bits, otherwise, the length of the destination is one byte.
- Consider BCHG.B #2, 003000. If [002000] = 05, then after execution of this BCHG instruction, Z = 0 and [03000] = 01.
- Consider BCLR.L #3, D1. If [D1] = F210E120, then after execution of this BCLR, register D1 will contain 0F10E120, and Z = 0.
- Consider BSET.B #0, A1. If [A1] = 000003000, [03000] = 00, then after execution of this BSET, memory location 003000 will contain 01, and Z = 1.

5.6.6 BINARY-CODED DECIMAL INSTRUCTION

The 68000 instruction set contains three BCD instructions, namely

- ABCD for adding, SBCD for subtracting, and NBCD for negating. These instructions always include the extra (X) bit in the operation. The BCD instructions are listed in Table 5.11.

- Consider ABCD D1, D2. If [D1] = 35 and [D2] = 15, X = 0, then after execution of the ABCD instruction, [D2] = 40, X = 0, and Z = 0.

- Consider SBCD - (A2). If [A2] = 00000200, [A3] =
TABLE 5.11
68000 Binary-Coded Decimal Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCD Dy, Dx</td>
<td>B</td>
<td>(Dx)10 + [Dy]10 + X → Dx</td>
</tr>
<tr>
<td>ABCD – (Ay), – (Ax)</td>
<td>B</td>
<td>–[Ax]10 + –[Ay]10 + X → [Ax]</td>
</tr>
<tr>
<td>SBCD Dy, Dx</td>
<td>B</td>
<td>(Dx)10 – (Dy)10 – X → Dx</td>
</tr>
<tr>
<td>NBCD (EA)</td>
<td>B</td>
<td>0 – [EA]10 – X → EA</td>
</tr>
</tbody>
</table>

Note: [EA] in NBCD can use all modes except An, relative, and immediate.

00003003H, [00200] = 05W, 003002H = 06W, then after execution of this SBCD, [002002] = 00W, X = 0, and Z = 1.

Consider NBCD (A1). If [A1] = [00003003H], [002000H] = 05W, X = 1, then after execution of the NBCD, [003000] = EA1W.

5.6.7 PROGRAM CONTROL INSTRUCTIONS

These instructions include branches, jumps, and subroutine calls as listed in Table 5.12.

TABLE 5.12 (continued)
68000 Program Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bcc d</td>
<td>B,W</td>
<td>(continued)</td>
</tr>
<tr>
<td>BRA d</td>
<td>B,W</td>
<td>Branch always to PC + d where PC value is current instruction location plus 2; as with Bcc, d can be signed 8 or 16 bits; this is an unconditional branching instruction with relative mode; note that the PC contents are even since the instruction length is either one word or two words</td>
</tr>
<tr>
<td>BSR d</td>
<td>B,W</td>
<td>PC → [SR]</td>
</tr>
<tr>
<td>DBcc Dn d</td>
<td>W</td>
<td>If condition code cc is true, then PC + d → PC; the PC value is current instruction location plus 2; d can be 8- or 16-bit signed displacement; if 8-bit displacement is used, then the instruction size is 16 bits with the 8-bit displacement as the low byte of the instruction word; if 16-bit displacement is used, then the instruction size is two words with the 8-bit displacement field (low byte) in the instruction word as zero and the second word following the instruction word as the 16-bit displacement</td>
</tr>
<tr>
<td>JMP (EA)</td>
<td>Unsize</td>
<td>[EA] → PC</td>
</tr>
<tr>
<td>JSR (EA)</td>
<td>Unsize</td>
<td>PC → [SP]</td>
</tr>
<tr>
<td>RTR</td>
<td>Unsize</td>
<td>[SP] + → CCR</td>
</tr>
</tbody>
</table>

There are 14 conditions such as BCC (Branch if Carry Clear), BEQ (Branch if result equal to zero, i.e., Z = 1), and BNE (Branch if not equal, i.e., Z = 0).

Note that the PC contents will always be even since the instruction length is either one word or two words depending on the displacement widths.

Branch always to PC + d where PC value is current instruction location plus 2; as with Bcc, d can be signed 8 or 16 bits; this is an unconditional branching instruction with relative mode; note that the PC contents are even since the instruction length is either one word or two words.

The address of the next instruction following PC is pushed onto the stack; PC is then loaded with PC + d; as before, d can be 8 or 16 bits; this is a subroutine call instruction using relative mode.

If cc is false, then Dn – 1 → Dn, and if Dn = –1, then PC + 2 → PC; if Dn = –1, then PC + d → PC; if cc is true, PC + 2 → PC.

This is an unconditional jump instruction which uses control addressing mode.

This is a subroutine call instruction which uses control addressing mode.

Return and restore condition codes.
TABLE 5.12 (continued)
68000 Program Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>Unsized</td>
<td>Return from subroutine [SPI + → PC]</td>
</tr>
<tr>
<td>Scc (EA)</td>
<td>B</td>
<td>If cc is true, then the byte specified by [EA] is set to all ones, otherwise the byte is cleared to zero</td>
</tr>
</tbody>
</table>

- [EA] in JMP and JSR can use all modes except Dn, An, (An) +, -(An), and immediate.
- [EA] in Scc can use all modes except An, relative, and immediate.

Consider Bcc d. There are 14 branch conditions. This means that cc in Bcc can be replaced by 14 conditions providing 14 instructions. These are BCC, BCS, BEQ, BGE, BGT, BHI, BLE, BLS, BLT, BMI, BNE, BPL, BVC, and BVS. It should be mentioned that some of these instructions are applicable to both signed and unsigned numbers, some can be used with only signed numbers, and some instructions are applicable to only unsigned numbers as shown below:

For both signed and unsigned numbers

<table>
<thead>
<tr>
<th>For signed numbers</th>
<th>For unsigned numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCC d (Branch if C = 0)</td>
<td>BGE d (Branch if greater or equal)</td>
</tr>
<tr>
<td>BCS d (Branch if C = 1)</td>
<td>BGT d (Branch if greater than)</td>
</tr>
<tr>
<td>BEQ d (Branch if Z = 0)</td>
<td>BLE d (Branch if less than or equal)</td>
</tr>
<tr>
<td>BNE d (Branch if Z = 0)</td>
<td>BLT d (Branch if less than)</td>
</tr>
<tr>
<td>BMI d (Branch if N = 1)</td>
<td>BVC d (Branch if V = 0)</td>
</tr>
<tr>
<td>BPL d (Branch if N = 0)</td>
<td>BVS d (Branch if V = 1)</td>
</tr>
</tbody>
</table>

For signed numbers

| BHI d (Branch if high) |

For unsigned numbers

| BSE d (Branch if high) |

After signed arithmetic operations such as ADD or SUB the instructions such as BEQ, BNE, BVS, BVC, BMI, and BPL can be used. On the other hand, after unsigned arithmetic operations, the instructions such as BCC, BCS, BEQ, and BNE can be used.

If V = 0, BMI and BGE have the same meaning. Likewise, if V = 0, BMI and BLT perform the same function.

The conditional branch instructions can be used after typical arithmetic instructions such as subtraction to branch to a location if cc is true. For example, consider SUB.W D1, D2. Now if D1 and D2 are unsigned numbers, then

BCC d can be used if [D2] > [D1]
BCS d can be used if [D2] < [D1]
BEQ d can be used if [D2] = [D1]
BNE d can be used if [D2] ≠ [D1]
BHI d can be used if [D2] > [D1]
BLT d can be used if [D2] < [D1]
BLS d can be used if [D2] ≤ [D1]

On the other hand, if D1 and D2 are signed numbers, then after SUB.W D1, D2, the following branch instructions can be used:

BEQ d can be used if [D2] = [D1]
BNE d can be used if [D2] ≠ [D1]
BLT d can be used if [D2] < [D1]
BLS d can be used if [D2] ≤ [D1]
BGT d can be used if [D2] > [D1]
BGE d can be used if [D2] ≥ [D1]

Now, as an example consider BEQ* = $20$. If $[PC] = 000200\_H$, then after execution of the BEQ instruction, program execution continues at $000220\_H$, if Z = 1; if Z = 0, program execution continues at $000200\_H$. Note that * is used by some assemblers to indicate displacement.

- The instruction BRA and JMP are unconditional JUMP instructions. The BRA instruction uses the relative addressing mode, whereas the JMP uses only control addressing modes. For example, consider BRA $\_S$ = $20\_H$. If $[PC] = 000200\_H$, then after execution of the BRA, program execution starts at $000220\_H$.

Now, consider JMP (A1). If $[A1] = 00000220\_H$, then after execution of the JMP, program execution starts at $000220\_H$.

- The instructions BSR and JSR are subroutine CALL instructions. BSR uses relative mode, whereas JSR uses control addressing mode. Consider the following program segment:
Main program
- SUB MOVEM.L D0-D7/AO-A6, -(SP)
- JSR SUB
- START
- MOVEM.L (SP)+, D0-D7/A0-A6
- RTS

In the above, JSR SUB instruction calls the subroutine called SUB. In response to JSR, the 68000 pushes the current PC contents called START onto the stack and loads the starting address SUB of the subroutine into PC. The first MOVEM in the SUB pushes all registers onto the stack and after the subroutine is executed, the second MOVEM instruction pops all the registers back. Finally, RTS pops the address START from the stack into PC, and the program control is returned to the main program. Note that BSR SUB could have been used instead of JSR SUB in the main program. In that case, the 68000 assembler would have considered the SUB with BSR as a displacement rather than as an address with the JSR instruction.

The 68000 can pass data between a main program and a subroutine using the following method:
1. Using 68000 registers
2. Using 68000 stack
3. Using parameter areas in memory
4. By coding argument values after subroutine CALL.

Note that the information required by a subroutine is called parameters. In high-level languages, parameters provide general information for the subroutine. The subroutine can then be called by specific values called arguments in the main program. The arguments replace the parameters with specific values and provide results. For example, in FORTRAN, a subroutine SUMSQ with two parameters (Q, R) can be defined as follows:

```
subroutine SUMSQ (Q, R)
```

This subroutine can be called with various arguments in the main program as long as the arguments and parameters are the same data types such as real, integer, and floating-point. The subroutine can be called by the main program by using CALL (5.0, RESULT). The subroutine uses the value 5.0 for Q and provides the answer in a location called RESULT. The specific value can be changed to get other results. In the above, Q and R are parameters, while 5.0 and RESULT are arguments.

Now let us discuss the methods of passing the arguments between the main program and the subroutine.

5.6.7.a Using 68000 Registers

The eight data registers (D0-D7) and seven address registers (A0-A6) can be used to pass arguments. Data can be passed via D0-D7, while addresses can be passed via A0-A6. For example, the following instruction sequence in the main program loads a 32-bit value in D5 and an indirect pointer in A4; the subroutine TRIG can use the values directly from the registers D5 and A4 to accomplish the desired function:

```
MOVE.L &DATA, D5
LEA.L ADDR, A4
BRR TRIG
```

This method is simple, requires small memory, but the number of arguments to be passed is limited by the number of registers.

5.6.7.b Using 68000 Stack

A 68000 stack can be formed by using A0-A6 registers as stack pointers. This stack can then be used to pass arguments by pushing the arguments onto the stack before calling the subroutine. The main program can use postincrement or predecrement modes for pushing the arguments. The following instruction sequence initializes A5 as the stack pointer and passes four arguments:

```
MOVE.L # STACKPTR, A5 ; Define A5 as SP
MOVE .L DATA1, (A5)+ ; Push first data
MOVE .L DATA2, (A5)+ ; Push second data
MOVE .L DATA3, (A5)+ ; Push third data
MOVE .L DATA4, (A5)+ ; Push fourth data
BRR START
```

In the above, STACKPTR defines the bottom of the stack and the stack grows from low to high addresses.

The four arguments can be retrieved by the subroutine START as follows:
Chapter 5

MOVE.L (.A5), DATA 4 ; Get DATA 4
MOVE.L (.A5), DATA 3 ; Get DATA 3
MOVE.L (.A5), DATA 2 ; Get DATA 2
MOVE.L (.A5), DATA 1 ; Get DATA 1

The stack pointer, now .A5, contains the value STACKPTR.
The system stack can also be used to pass arguments. The arguments can be pushed by the main program before calling the subroutine as follows:

MOVE.L DATA 1, -(USP) ; Push DATA 1
MOVE.L DATA 2, -(USP) ; Push DATA 2
PRA.W LOC ; Push LOC
BSR START ; Call Subroutine
; START

Since BSR .START pushes PC onto the stack top, this must be popped from the stack temporarily before popping the arguments as follows:

MOVE.L (USP)+, AO ; Save PC temporarily
MOVE.L (USP)+, A1 ; Obtain LOC
MOVE.L (USP)+, D0 ; Obtain DATA 2
MOVE.L (USP)+, D1 ; Obtain DATA 1

MOV.E.L AO, -(USP) ; RESTORE PC
RTS
; Return to
; Main Program

This method is normally used by the user program and, hence, USP is used as the stack pointer.

5.6.7.c Using Parameter Areas in Memory
A parameter area is usually created when several parameters need to be passed. This parameter area contains the data or the addresses to be accessed by subroutine after the subroutine has been passed the beginning address of the parameter area by the main program.

The main program can create a parameter area of three parameters as follows:

MOVEA.L (USP), .A5 ; Obtain PC into .A5
MOVE.W (.A5)+, D2 ; Obtain 5 into D2
MOVE.W (.A5)+, D3 ; Obtain 8 into D3
MOVE.L .A5, (USP) ; Save .START+4 in stack

; ; ;

RTS
Chapter 5

The MOVEA instruction loads the PC value START in A6. The first MOVE loads the content (5) of memory location START to low word of D2 and increments A5 to START+5. The second MOVE moves the content (8) of location START+2 to low word of D3 and updates A5 to START+4. The MOVEA instruction then pushes START+4 to stack. RTS instruction at the end of the subroutine pops START+4 to PC and returns control to the right place in the main program.

The use of stack to save temporary variables during subroutine execution can be facilitated by utilizing a stack frame. The stack frame is a set of locations in stack used for saving return addresses, local variables, and I/O parameters. Local variables such as loop counters used in the subroutine are not returned to the main program. The stack frame can be utilized by the subroutine to access a new set of parameters each time the main program calls it.

In 68000, the stack frame can be created by the main program and the UNLK instructions. The subroutine can utilize the LINK and UNLK instructions. The instructions can create a stack frame on the stack by using displacements from a base register called a frame pointer. The 68000 LINK instruction can be used to create a stack frame and define the frame pointer.

As an example of LINK/UNLK, the instruction LINK A5, #-100 creates a stack frame of 100 words, with A5 as the frame pointer. The instruction MOVEM.W (A5), D1 and MOVEM.W D1,.A5 can then be used to access the stack frame.

A typical instruction sequence illustrating the use of LINK and UNLK is given below:

**Main Program**
- MOVE.L Const,-(USP) ; Push address of 32-bit constant to be passed
- PEA.W ADDR ; Push starting address of a table to be passed
- JSR START ; Jump to the subroutine

**Subroutine**
- LINK A5, #-12 ; Allocate 12 bytes

- MOVE.L DATA 1,-4 (A3) ; Push local variable 1
- MOVE.L DATA 2,-6 (A3) ; Push local variable 2

The following illustrates the stack contents at various points:

1. **Before executing the main program**
2. **Stack after JSR**
3. **Stack after UNLK**
4. **Stack after RTS**

In the main program, it is assumed that the 68000 is in user mode and, therefore, USP is used in the program. It is assumed that the main program passes the subroutine a 32-bit constant, and the starting address ADDR of a table to be accessed by the subroutine. The subroutine allocates 12 bytes to save three 32-bit local variables by using the LINK instruction. Several instructions are arbitrarily chosen (to illustrate the concept) to be executed by the subroutine to read data from the table, decrement the local variable 3 by 5, and read the 32-bit constant so that they can be used in the instruction sequence that follows (not shown in the subroutine). The
5.6.8 SYSTEM CONTROL INSTRUCTIONS

The 68000 contains some system control instructions which include privileged instructions, trap instructions, and instructions that use or modify SR. Note that the privileged instructions can only be executed in the supervisor mode. The system control instructions are listed in Table 5.13.

- The RESET instruction, when executed in supervisor mode, outputs a signal on the RESET pin of the 68000 in order to initialize the external peripheral chips. The 68000 reset pin is bidirectional. A bidirectional buffer circuitry can be connected to the reset pin. The 68000 can be reset by asserting the reset pin using hardware, whereas the peripheral chips can be reset using the software RESET instruction.
- MOVE, USP, (An) or MOVE.L (An), USP can be used to save, restore, or change the contents of USP in supervisor mode. The USP must be loaded in supervisor mode since MOVE USP is a privileged instruction.
- Consider TRAP # n. There are 16 TRAP instructions with n ranging from 0 to 15. The hexadecimal vector address is calculated using the following equation:

   Hexadecimal vector address = 80 + 4*n

TABLE 5.13

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIVileged Instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESET</td>
<td>Unsized</td>
<td>If supervisory state, then assert reset line; else TRAP</td>
</tr>
<tr>
<td>RTE</td>
<td>Unsized</td>
<td>If supervisory state, then restore SR and PC; else TRAP</td>
</tr>
<tr>
<td>STOP #data</td>
<td>Unsized</td>
<td>If supervisory state, then load immediate data to SR and then STOP; else TRAP</td>
</tr>
<tr>
<td>ORI to SR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVE USP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANDI to SR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EORI to SR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVE (EA) to SR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Trap and Check Instructions

<table>
<thead>
<tr>
<th>TRAP # vector</th>
<th>Unsized</th>
<th>PC → [SSP]</th>
<th>SR → [SSP]</th>
<th>Vector address → PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAPV</td>
<td></td>
<td>W</td>
<td>If Dn &lt; 0 or Dn &gt; [EA], then TRAP</td>
<td></td>
</tr>
<tr>
<td>CHK (EA), Dn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Status Register

| ANDI to CCR   |      | Already explained earlier |
| EORI to CCR   |      |                           |
| MOVE (EA) to CCR |    |                           |
| ORI to CCR    |      |                           |
| MOVE SR to (EA) |  |                           |

Note: (EA) in CHK can use all modes except An.
The TRAP instruction first pushes the contents of PC and then the
SR onto the system (user or supervisor) stack. The hexadecimal vector
address is then loaded into PC. The TRAP is basically a software
interrupt.

One of the 16 trap instructions can be executed in the user mode to
evaluate a supervisory program located at the specified trap routine.
Using the TRAP instruction, control can be transferred to the super-
visor mode from the user mode.

There are other traps which occur due to certain arithmetic errors. For
example, division by zero automatically traps to location 140h on the
other hand, an overflow condition, i.e., if V = 1, will trap to address
1C0h if the instruction TRAP is executed.

The CHK (EA) instruction compares [Dn] with (EA).
If [Dn]16h is less than 0 or if [Dn]16h > (EA), then a trap to location
0018h is generated. Also, if [Dn]16h = 0 and N is reset
to zero if [Dn]16h = (EA). (EA) is treated as a 16-bit 2's comple-
ment integer. Note that program execution continues if [Dn]16h lies
between 0 and (EA).

Consider CHK (A5), D2. If [D2]16h is less than 0200h, [A5] = 00003000h,
[003000h] = 0100h, then after execution of the CHK, the 68000 will trap
since [D2] = 0200h is greater than [003000h] = 0100h.

The purpose of the CHK instruction is to provide boundary checking
by testing if the content of a data register is in the range from zero to an
upper limit. The upper limit used in the instruction can be set equal to the
length of the array. Then every time the array is accessed, the CHK
instruction can be executed to make sure that the array bounds have not
been violated.

The CHK instruction is usually placed following the computation of an
index value to ensure that the limits of this index value are not violated.

This allows checking whether or not an address of an array belonging
is within the array boundaries when the address register indicates the index
mode is used to access an element in the array. For example, the instruction
sequence shown below will allow accessing of an array with base address in
A8 and array length of 125h bytes:

| CHK | [124, D5] |
| MOVE.B 0 (A3, D5.W), D4 |

In the above, if low 16 bits of D5 is greater than 124, the 68000 will trap
to location 0018h.

In the above, it is assumed that D5 is computed prior to execution of
the CHK instruction. Also, the 68000 assembler requires that a displace-
ment value of 0 be specified as in the instruction MOVE.B 0 (A3, D5.W),
D4.

5.6.9 STACKS AND QUEUES

A stack is a last-in-first-out (LIFO) memory, while the queue is first-
in-first-out (FIFO) memory. When data are added to a stack or a queue,
they are "pushed" onto the structure; when they are removed, they are
"popped" from the structure.

The 68000 supports stacks and queues with the address register indirect
postincrement and predereference addressing modes. A queue can be
implemented using a linked list. Each entry in the linked list contains the
data to the next entry in the list, along with data for each element.

A linked list with four elements is shown below:

```
<table>
<thead>
<tr>
<th>Pointer to beginning of list (HEAD)</th>
<th>Element 0</th>
<th>Element 1</th>
<th>Element 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer to Element 1</td>
<td>Pointer to Element 2</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>Data for Element 0</td>
<td>Data for Element 1</td>
<td>Data for Element 2</td>
<td></td>
</tr>
</tbody>
</table>
```

An advantage of linked list is that the items or elements in the list need not
be stored in memory in a sequential order, since each element includes the
pointer to the next element. This means that in order to change the order
of one or more elements, the address pointers must be moved, while the
data associated with each element are not required to be moved. Addition,
deletion, and insertion of elements is easier with linked lists. The element
2 (last element) in the above figure includes a special symbol identified as
NULL to indicate the end of the list. In 68000, a NULL value of 0000h
can be used, since no data will be stored at address 0000h. If the address
HEAD contains 0000h, the list is assumed to be empty.

5.6.9.9a Stacks

In addition to SP, all seven address registers A0-A6 can be used as stack
points at using appropriate addressing modes.

Subroutine calls, traps, and interrupts automatically use the system
stack pointers: USP when S = 0 and SSP when S = 1. Subroutine calls push
PC onto the stack, while RTE pops PC from the system stack. Traps and interrupt push both PC and SR onto the system, while RTE pops PC and SR from the system stack.

These stack operations fill data from high memory to low memory. This means that the system SP is decremented by 2 (word) or 4 (long word) after push and incremented by 2 (for word) or 4 (for long word) after pop. As an example, suppose that a 68000 call instruction (SR or BSR) is executed when PC = 50031 F200; then after execution of the subroutine call the stack will push the PC as follows:

| USP-4 or SSP-4 | 0031 (H) | Low Address |
| USP-2 or SSP-2 | F200 (L) | High Address |
| USP or SSP | VALID DATA |

Note that the system stack pointer always points to valid data.

Stacks can be created by the user by using address register indirect with postincrement or predecrement modes. Using one of the seven (A0-A6) address registers, the user may create stacks which can be filled from either high memory to low memory or vice versa.

Stack from high to low memory is implemented with predecrement mode for push and postincrement mode for pop. On the other hand, stack from low to high memory is implemented with postincrement for push and predecrement for pop. For example, consider the following stack growing from high to low memory addresses in which A5 is used as the stack pointer:

| 72 2000016 | A5 | 2000004 |
| 2000006 | 2000008 | 2000010 | 2000012 | 2000014 | 2000016 |
| ~TOP | DATA 2 | DATA 1 | DATA 6 | BOTTOM |

In order to push 16-bit content 0504₁₆ of memory location 205010₁₆, the instruction MOVE.W $205010, - (A5) can be used as follows:

| STK | 0504₁₆ | 2000020₁₆ | 2000018₁₆ | 2000016₁₆ |
| TST | TOP |
| 2000014₁₆ | 2000012₁₆ | 2000010₁₆ | 2000008₁₆ |
| 2000006₁₆ | 2000004₁₆ | 2000002₁₆ | 2000000₁₆ |
| BOTTOM |

The 16-bit data 0504₁₆ can be popped from the stack into low 16 bits of D0 by using MOVE.W (A5)+, D0. A5 will contain 2000010₁₆ after the POP. Note that in this case, the stack pointer A5 points to valid data.

Next, consider the stack growing from low to high memory addresses in which, say, A6 is used as the stack pointer:

| 23 0 30500016 | A6 | 3050004 |
| 3050006 | 3050008 | 3050010 | 3050012 | 3050014 | 3050016 |
| ~TOP | DATA 0 | DATA 1 | DATA 6 | BOTTOM |

In order to PUSH 16-bit contents 2070₁₆ of the low 16 bits of D5, the instruction MOVE.W D5, (A6)+ can be used as follows:

| STK | 30500016 | 305000016 | 2070₁₆ |
| BOTTOM |
| 30500014₁₆ | 30500012₁₆ | 30500010₁₆ | 30500008₁₆ |
| DATA 0 |
| 2070₁₆ |
| TOP |
| FREE |
5.6.9.b Queues

Queues are used to store data in the order in which they will be used. Operating systems store tasks in queues so that they will be executed in the right order.

User queues can be created with the address register indirect with postincrement or predecrement mode. Using two address registers of A0-A6, the user may implement queues which are filled from either high memory to low memory addresses or vice versa. Since queues are pushed from one end and popped from the other, two address registers (Put Pointer and Get Pointer) are used.

For low to high memory addresses, data are placed in the queue using one of A0-A6 such as A5 with postincrement mode as the put pointer, while data are obtained from the queue using a different A0-A6 register such as A6 with postincrement mode. After a put operation, the put address register addresses the next free space in the queue, while the unchanged get register addresses the next data to be read from the queue. After a get operation, the get address register addresses the next data to be read from the queue and the unchanged put address register addresses the next free space in the queue. This situation is shown in the following:

To put data from, for example, low word of D1 into the FREE space of low memory, the instruction MOVE.W D1, (A5)+ can be used. To get data into a 24-bit memory location 451724, the instruction MOVE.W (A3)+, $451724$ can be used.

For high to low memory queue, data are placed in the queue using one of A0-A6 with predecrement mode as the put pointer. Data can be obtained from the queue using a different A0-A6 with predecrement mode. After a put operation, the put address register addresses the last data placed in the queue and the unchanged get address register addresses the last data read from the queue. After a get operation, the get address register addresses the last data read from the queue and the unchanged put address register addresses the last data placed in the queue. This situation is shown below:

Now, to put 32-bit data from, for example, D2 into the free space of high memory, the instruction MOVE.L D2, -(A6) can be used. To get data to low 16-bit D3, the instruction MOVE.W -(A2), D3 can be used.
Example 5.1
Determine the effect of each one of the following 68000 instructions:

- CLR D0
- MOV L D1, D0
- CLRL (A0)+
- MOVE - (A0), D0
- MOVE 20 (A0), D0
- MOVQ L #$D7, D0
- MOVE 21 (A1, A0.L), D0

Assume the following initial configuration before each instruction is executed. Also, assume all numbers in hex.

- [D0] = 22220000
- [D1] = 55556666
- [A0] = 00002222
- [A1] = 00003333
- [000220] = 8888
- [000212] = 7777
- [000214] = 5555
- [000216] = 5555
- [000218] = AAAA
- [00056C] = FF00

Solution

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effective address</th>
<th>Net effect (HEX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR D0</td>
<td>Destination EA = D0</td>
<td>[D0] = 22220000</td>
</tr>
<tr>
<td>MOV L D1, D0</td>
<td>Destination EA = D0</td>
<td>[D0] = 55556666</td>
</tr>
<tr>
<td>CLRL (A0)+</td>
<td>Source EA = D1</td>
<td></td>
</tr>
<tr>
<td>MOVE - (A0), D0</td>
<td>Source EA = [A0] + 2</td>
<td>[A0] = 00002222</td>
</tr>
<tr>
<td>MOVE 20 (A0), D0</td>
<td>Source EA = [A0] + 20 (or 14)</td>
<td>[A0] = 00002222</td>
</tr>
</tbody>
</table>

Example 5.2
Write a 68000 assembly program segment that implements each one of the following Pascal segments. Assume the following information about the variables involved in this problem.

Variable | Comments
---|---
X | Address of a 16-bit signed integer of array 1 of 10 elements
Y | Address of a 16-bit signed integer of array 2 of 10 elements
SUM | Address of the sum

(a) If X ≥ Y, then X := X + 10; else Y := Y + 12.

(b) Sum := 0; for i := 0 to 9, do sum := sum + A[i].

Solution

(1) LEA X, A0 ; Point A0 to X
LEA Y, A1 ; Point A1 to Y
MOVE (A0), D0 ; MOVE [X] into D0
CMP (A1), D0 ; COMPARE [Y] WITH
BGE THRU7
5.6 68000 Instruction Set

unconditional JUMP to some location such as FINISH JMP FINISH in the above program must be used. Since DBF is a word instruction and considers D0's low 16-bit word as that loop count, D0 must be initialized by a word MOVE rather than byte MOVE, even though $\Sigma_{i=1}^{N}$ can be accommodated in a byte. Also, one should be careful about using MOVEQ, since MOVEQ sign extends a byte to a word.

Example 5.4

Write 68000 assembly program to compute

$$\sum_{i=1}^{N} x_i y_i$$

where $x_i$ and $y_i$ are signed 16-bit numbers, $N = 100$. Assume no overflow.

Solution

```
ORG $1000
MOVE L#$99, D0
LEA P, A0
LEA Q, A1
CLR L D1
LOOP MOVE (A0)+, D2
MULS (A1)+, D2
ADCL D2, D1
DBF D0, LOOP
FINISH JMP FINISH
```

Example 5.5

Write a 68000 subroutine to compute

$$y = \sum_{i=1}^{N} \frac{x_i}{x_i}$$

Assume the $x_i$'s are 16-bit signed integers and $N = 100$. The numbers are stored in consecutive locations. Assume A0 points to the $x_i$'s and SP is already initialized. Assume no overflow.
Solution

```assembly
SQR   MOVEM.L D2/D3/A0, -(SP) ; Save registers
CLR   L D1
MOVEM.L #99, D2 ; Clear sum
LOOP  MOVE.W (A0)+, D3 ; Initialize loop count
MULS D3, D3
ADDS L D3, D1
DBF D2, LOOP
DIVS #130, D1
MOVEM.L (SP)+, D2/D3/A0 ; Store Xi into D1
RTS ; Compute Σ Xi/N

Example 5.6
Write a 68000 assembly language program to move block of data length 100, from the source block starting at location 002000H to the destination block starting at location 003000H.

Solution

```assembly
MOVEA.L #82000, A4 ; Load A4 with source address
MOVEA.L #30000, A5 ; Load A5 with destination address
MOVEM.L #99, D0 ; Load D0 with count -1 = 98H
START MOVE.W (A4)+, (A5)+ ; MOVE source data to destination
DBF D0, START
END JMP END
HALT
```

Example 5.7
Assume the following linked list:

```
2000H 2020H 2000H 2020H
HEAD
```

Solution

```
ADDR1 EQU $2010
HEAD EQU $2000
ORG $5000
MOVEA.W HEAD, A5 ; MOVE old head to A5
MOVE.W A5, ADDR1 ; MOVE starting address of old item 0 to ADDR1
MOVE.W A5, ADDR1 ; Item 0 first content
STOP JMP STOP
```

5.7 68000 PINS AND SIGNALS

The 68000 is housed in one of the following packages:
- 64-pin dual in-line package (DIP)
- 68-Terminal Chip Carrier
- 68-pin Quad Pack
- 68-pin Grid Array

Figure 5.8 shows the pin diagrams of the 68-pin grid array. Pin diagrams for the other three packages are shown in Appendix B.

The 68000 is provided with several Vcc and ground pins. Power is thus distributed in order to reduce noise problems at high frequencies.

In order to build a prototype to demonstrate that the paper design for
The 8000-based microcomputer is correct, one must use either wire-wrap or solder for the actual construction. Prototype board must not be used. This is because at high frequencies above 4 MHz, there will be noise problems due to stray capacitances.

D0-D15 is the 16-bit data bus. All transfers to and from memory and I/O devices are conducted over the 16-bit bus. A1-A23 are the 23 address lines. A0 is obtained by encoding UDS (Upper Data Strobe) and LDS (Lower Data Strobe) lines. The 68000 operates on a single-phase TTL level clock at 4 MHz, 6 MHz, 8 MHz, 10 MHz, 12.5 MHz, 16.67 MHz, or 25 MHz. The 68000 also has a lower-power HCMOS version called the MC68HC000 which can run at 8 MHz, 10 MHz, 12.5 MHz, and 16.67 MHz.

There is no on-chip clock generator/clock circuit and therefore the clock must be generated externally. This clock input is utilized internally by the 68000 to generate additional clock signals for synchronizing the 68000's internal operation.

Figure 5.9 shows the 68000 CLK waveform and clock timing specifications.
The clock is at TTL compatible voltage. The clock timing specifications provides data for three different clock frequencies: 8 MHz, 10 MHz, and 12.5 MHz.

The 68000 CLK input can be provided by a crystal oscillator or by designing an external circuit. Figure 5.10 shows a simple oscillator to generate the 68000 CLK input. The above circuit uses two inverters connected in series. Inverter 1 is biased in its transition region by the resistor R. Inverter 1 inverts the output (sinusoidal) to produce logic pulse train at the output of inverter 1. Inverter 2 sharpens the wave and drives the crystal. For this circuit to work, HCMOS logic (74HCO0, 74HC02, or 74HC04) must be used and a coupling capacitor should be connected across the supply terminals to reduce the ringing effect during high-frequency switching of the HCMOS devices. Additionally, the output of this oscillator is fed to the clock input of a D-flip-flop (74LS74) to further reduce the ringing. Hence, a clock signal of 50% duty cycle at a frequency of 1/2 the crystal frequency is generated.

The 68000 consumes about 1.5 watts of power. The 68000 signals can be divided into five functional categories. These are

1. Synchronous and asynchronous control lines
2. System control lines
3. Interrupt control lines
4. DMA control lines
5. Status lines

5.7.1 SYNCHRONOUS AND ASYNCHRONOUS CONTROL LINES

The 68000 bus control is asynchronous. This means that once a bus cycle is initiated, the external device must send a signal back in order to complete it. The 68000 also contains three synchronous control lines that facilitate interfacing to synchronous peripheral devices such as Motorola’s inexpensive MC68000 family. Note that synchronous operation means that bus control is synchronized or clocked using a common system clock signal. In 68000 family peripherals, this common clock is phase 2 (P2) or an E clock signal depending on the particular chip used.

With synchronous control, all READ and WRITE operations must be synchronized with the common clock. However, this may create problems when interfacing slow peripheral devices. This problem does not arise with asynchronous bus control.

Asynchronous operation is not dependent on a common clock signal. The 68000 utilizes the asynchronous control lines to transfer data between the 68000 and peripheral devices via handshaking. Using asynchronous operation, the 68000 can be interfaced to any peripheral chip regardless of the speed. The 68000 has three control lines to transfer data over its bus in a synchronous manner. These are E (enable), VPA (valid peripheral address), and VMA (valid memory address).

The E clock corresponds to phase 2 clock of the 68000. The E clock is output at a frequency that is 1/10th of the 68000 input clock. The VPA is an input which tells the 68000 that a 68000 device is being addressed and therefore data transfer must be synchronized with the E clock. The VMA is processor's response to VPA. VPA is asserted when the memory address is valid. This also tells the external device that the next data transfer over the data bus will be synchronized with the E clock. VPA can be generated by decoding the address pins and address strobe (AS). Note that the 68000 asserts AS low when the address on the address bus is valid. VMA is typically used as chip select of the 68000 peripheral. This ensures that the 68000 peripherals are selected and deselected at the correct time. The 68000 peripheral interfacing sequence is provided in the following:

1. The 68000 initiates a cycle by starting a normal read or write cycle.
2. The 68000 peripheral defines the 68000 cycle by asserting the 68000 VPA input. If the VPA is asserted as soon as possible after the assertion of AS, then VPA will be recognized as being asserted on the falling edge of AS. If the VPA is not asserted at the falling edge of AS, the 68000 inserts wait states until VPA is recognized by the 68000 as asserted. DTACK should not be asserted while VPA is asserted. The 68000 peripheral must remove VPA within one clock after AS is negated.
3. The 68000 monitors enable (E) until it is low. The 68000 then
synchronizes all read and write operations with the E clock. VMA output pin is asserted low by the 68000.

4. The 68000 peripheral waits until E is active (HIGH) and then transfers the data.

5. The 68000 waits until E goes low (on a read cycle the data is latched as E goes low internally). The 68000 then negates VMA, AS, UDS, and LDS. The 68000 thus terminates the cycle and starts the next cycle.

The 68000 provides five lines to control address and data transfers asynchronously. These are AS (Address Strobe), R/W (Read/Write), DTACK (Data Acknowledge), UDS (Upper Data Strobe), and LDS (Lower Data Strobe).

The 68000 outputs AS to notify the peripheral device when data are to be transferred. AS is active LOW when the 68000 provides a valid address on the address bus. The R/W is HIGH for read and LOW for write. The DTACK is used to tell the 68000 that a transfer is to be performed.

When the 68000 wants to transfer data asynchronously, it first activates the AS line and, at the same time, the 68000 generates the required address on the address lines in order to select the peripheral device.

Since the AS line tells the peripheral chip when to transfer data, the AS line should be part of the address decoding scheme. After enabling the AS, the 68000 enters the wait state until it receives the DTACK from the selected peripheral device. On receipt of the DTACK, the 68000 knows that the peripheral device is ready for data transfer. The 68000 then utilizes the R/W and data lines to transfer data.

UDS and LDS are defined as follows:

<table>
<thead>
<tr>
<th>UDS</th>
<th>LDS</th>
<th>Data Transfer occurs via</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>D0-D7 pins for byte</td>
<td>Odd</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>D8-D15 pins for byte</td>
<td>Even</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>D0-D15 pins for word or long word</td>
<td>Even</td>
</tr>
</tbody>
</table>

UDS and LDS are used to segment the memory into bytes instead of words. When the UDS is asserted, contents of even addresses are transferred on the high-order 8 lines of the data bus, D8-D15. The 68000 internally shifts these data to the low byte of the specified register. When LDS is asserted, contents of odd addresses are transferred on the low-order 8 lines of the data bus, D0-D7. During word and long word transfers, both UDS and LDS are asserted and information is transferred on all 16 data lines, D0-D15. Note that during byte memory transfers, A0 corresponds to UDS for even addresses (A0 = 0) and to LDS for odd addresses (A0 = 1). The circuit in Figure 5.11 shows how even and odd addresses are interfaced to the 68000.

**FIGURE 5.11 Interfacing of the 68000 to even and odd addresses.**

### 5.7.2 SYSTEM CONTROL LINES

The 68000 has three control lines, namely, BERR (Bus Error), HALT, and RESET, that are used to control system-related functions.

The BERR is an input to the 68000 that is used to inform the processor that there is a problem with the instruction cycle currently being executed. With asynchronous operation, this problem may arise if the 68000 does not receive DTACK from a peripheral device. An external timer can be used to activate the BERR pin if the external device does not send DTACK within a certain period of time. On receipt of the BERR, the 68000 does one of the following:

- Renews the instruction cycle which caused the error
- Executes an error service routine

The troubled instruction cycle is renounced by the 68000 if it receives a
HALT signal along with the BERR signal. On receipt of LOW on both HALT and BERR pins, the 68000 completes the current instruction cycle and then goes itself into the high impedance state. On removal of both HALT and BERR (that is, when both HALT and BERR are HIGH), the 68000 returns the troubled instruction cycle. The cycle can be reset repeatedly if both BERR and HALT are enabled/disabled continually.

On the other hand, an error service routine is executed only if the BERR is received without HALT. In this case, the 68000 will branch to a bus error vector address where the user can write a service routine. If two simultaneous bus errors are received via the BERR pin without HALT, the 68000 automatically goes into the HALT state until it is reset.

The HALT line can also be used by itself to perform single-stepping or to provide DMA. When HALT input is activated, the 68000 completes the current instruction and goes into a high impedance state until HALT is returned to HIGH. By enabling disabling the HALT line continually, the single-stepping debugging can be accomplished. However, since most 68000 instructions consist of more than one instruction cycle, single-stepping using HALT is not normally used. Rather, the trace bit in the status register is used to single-step the complete instruction.

One can also use HALT to perform microprocessor-halt DMA. Since the 68000 has separate DMA control lines, DMA using the HALT line will not normally be used.

The HALT pin can also be used as an output signal. The 68000 will assert the HALT pin LOW when it goes into a HALT state as a result of a catastrophic failure. The double bus error (activation of BERR twice) is an example of this type of error. When this occurs, the 68000 goes into a high impedance state until it is reset. The HALT line informs the peripheral devices of the catastrophic failure.

The RESET line of the 68000 is also bidirectional. In order to reset the 68000, both the RESET and HALT pins must be asserted at the same time. The 68000 executes a reset service routine automatically for loading the PC with the starting address of the program. The 68000 RESET pin can also be used as an output line. A LOW can be sent to this output line by executing the RESET instruction in the supervisor mode in order to reset external devices connected to the 68000. The execution of the RESET instruction does not affect any data, address, or status register. Therefore, the RESET instruction can be placed anywhere in the program whenever the external devices need to be reset. An external bidirectional buffer circuit is needed in order to use the RESET pin as both input and output.

The RESET line of the 68000 is bidirectional. This means that the 68000 RESET pin can be used as an input or an output line. For hardware reset, this RESET pin must be activated LOW by an external circuit. This will initialize 68000 PC, SP, and status register. The RESET pin can also be used as an output pin. The RESET instruction can be executed by the 68000 in supervisor mode to output a LOW on the RESET pin to reset the memory and I/O chips. In order to use the RESET pin as an I/O pin, bidirectional buffer will be required. However, the reset input pins of the memory and I/O chips can be connected to the 68000 RESET pin along with the reset circuit. Activation of the 68000 RESET pin via the reset circuit will reset the 68000, memory, and I/O chips. In this case, bidirectional buffers are not required.

In order to reset the 68000, both the RESET and HALT pins must be asserted simultaneously by an external circuit. Figure 5.12 shows the timing diagram for the 68000 reset operation.

Upon hardware reset, the 68000 performs the following:

1. The 68000 reads four words from addresses $000000, $000002, $000004, and $000006. The 68000 loads the supervisor stack pointer high and low words with the contents of locations $000000 and $000002, respectively. Also, the program counter high and low words are loaded with the contents of locations $000004 and $000006, respectively.
2. The 68000 initializes the status register to an interrupt level of seven.
3. No other registers are affected by hardware reset.
When a RESET instruction is executed, the 68000 drives the RESET pin low for 124 clock periods. In this case, the processor is trying to reset the rest of the system. Therefore, there is no effect on the internal state of the 68000. All of the 68000 internal registers and the status register are unaffected by the execution of a reset instruction. All external devices connected to the RESET line will be reset at the completion of the reset instruction.

Asserting the RESET and HALT lines for 10 clock cycles will cause a processor reset, except when Vcc is initially applied to the 68000. In this case, an external reset must be applied for at least 100 milliseconds.

The reset circuit (used for 8085) depicted in Figure 5.13a satisfies the 68000 reset requirements mentioned above.

---

**Figure 5.13a** 68000 RESET circuit (simple)

The above circuit is similar to the 8085 reset circuit except that the output goes to both RESET and HALT lines of the 68000. A more accurate RESET circuit is shown in Figure 5.13b.

The Motorola MCl455 in Figure 5.13b is a timer chip that provides accurate time delays or oscillation. The timer is precisely controlled by external resistors and capacitors. The timer may be triggered by an external trigger input (falling waveform) and can reset by external reset input (falling waveform).

The reset circuit in Figure 5.13b will assert the 68000 RESET pin for at least 10 clock cycles. From the MCl455 data sheet, the internal block diagram of the MCl455 is shown in Figure 5.13c.

When the input voltage (Vcc) to the trigger comparator (COMPB) falls below 1/3 Vcc, the comparator output (COMP2A) triggers the flip-flop so that its output becomes low. This turns the capacitor discharge transistor OFF and drives the digital output to the HIGH state. This condition permits the capacitor to charge at an exponential rate set by the...
output is HIGH for 5.17 s and the output of the debounce circuit is HIGH when the push button is activated. This will generate a LOW at the HALT pin for 5.17 s (greater than 1.25 μs). The timing requirements of the 68000 RESET and HALT pins will be satisfied by the reset circuit of Figure 5.13b.

Note that when the reset circuit is not activated, the top input of NAND gate #1 is HIGH and the bottom input of NAND gate #2 is LOW (HIGH is grounded to LOW, see Figure 5.13b). Since a NAND gate always produces a HIGH output when one of the inputs is LOW, the output of NAND gate #2 will be HIGH. This will make the bottom input of NAND gate #1 HIGH and thus the output of NAND gate #1 will be LOW, which in turn will make the output of NAND gate #3 HIGH. Therefore, the 68000 will not be reset when the push button is not activated. Upon activation of the push button, the top input of NAND gate #1 is LOW (HIGH is grounded to LOW); this will make the output of NAND gate #1 HIGH. Hence, both inputs of NAND gate #3 will be HIGH providing a LOW at the HALT pin for 5.17 s (greater than 1.25 μs).

5.7.3 INTERRUPT CONTROL LINES

IPL0, IPL1, and IPL3 are interrupt control lines. These lines provide for seven interrupt priority levels (IPL3, IPL2, IPL1, IPL0 = 111 means no interrupt and IPL0 = 000 means nonmaskable interrupt). IPL2, IPL1, IPL0 = 001 through 110 provides six maskable interrupts. The 68000 interrupts will be discussed later in this chapter.

5.7.4 DMA CONTROL LINES

BR (Bus Request), BG (Bus Grant), and BGR (Bus Grant Acknowledge) lines are used for DMA purposes. The 68000 DMA will be discussed later in this chapter.

5.7.5 STATUS LINES

The 68000 has three output lines called the function code pins (FC2, FC1, and FC0). Table 5.14 shows how these lines tell external devices whether user data, user program, supervisor data, or supervisor program is being addressed. These lines can be decoded to provide user or supervisor programs and/or data, and interrupt acknowledge as shown in Table 5.14.

The FC2, FC1, and FC0 pins can be used to partition memory into four functional areas: user data memory, user program memory, supervisor data memory, and supervisor program memory. Each memory partition can directly access up to 16 megabytes, and thus the 68000 can be used to directly address up to 64 megabytes of memory. This is shown in Figure 5.14.
TABLE 5.14
Function Code Lines

<table>
<thead>
<tr>
<th>FC2</th>
<th>FC1</th>
<th>FC0</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Unassigned</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>User data</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>User program</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Unassigned</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Unassigned</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Supervisor data</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Supervisor program</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Interrupt acknowledge</td>
</tr>
</tbody>
</table>

FIGURE 5.14 Partitioning 68000 address space using FC2, FC1, and FC0 bits.

5.8 68000 SYSTEM DIAGRAM

Figure 5.15 shows a simplified version of the 68000 basic system diagram.

5.9 TIMING DIAGRAMS

The 68000 family of processors (68000, 68008, 68010, and 68012) uses a handshaking mechanism to transfer data between the processors and the peripheral devices. This means that all these processors can transfer data asynchronously to and from peripherals of varying speeds.

Figure 5.16 shows 68000 read and write cycle timing diagrams.

During the read cycle, the 68000 obtains data from a memory location or an I/O port. If the instruction specifies a word such as MOVE.W0,250,04, D1 or a long word such as MOVE.L 0030808, D0, the 68000 reads both upper and lower bytes at the same time by asserting UDS and LDS pins. When the instruction is for a byte operation the 68000 utilizes an internal bit to find which byte to read and then outputs the data strobe required for that byte. For byte operations, when the address is even (A0 = 0), the 68000 asserts UDS and reads data via D8-D15 pins into low byte of the
specified data register. On the other hand, for reading data byte from an odd address (A0 = 1), the 68000 outputs LOW on LDS and reads data byte via D8-D7 pins into low byte of the specified data register. For example, consider MOVE.B $507144, D5. The 68000 outputs LOW on UDS (since A0 = 0) and high on LDS. The memory chip’s 8 data lines must be connected to the 68000 D8-D15 pins. The 68000 reads the data byte via D8-D15 pins into low byte of D5. Note that for reading a data byte from an odd location by executing an instruction such as MOVE.B $507145, D5, the 8 data lines of the memory chip must be connected to 68000 D0-D7 pins. The 68000, in this case, outputs low on LDS (A0 = 0) and high on UDS, then reads the data byte into low byte of D5.

Now, let us discuss the read timing diagram of Figure 5.16. Consider Figure 5.16 for read word timing. During SO, address and data signals are in high impedance state. At the start of S1, the 68000 outputs the address on its address pins (A1-A23). During S0, the 68000 outputs FC2-FC0 signals. AS can be used at this point to latch the signals on the address pins. The 68000 asserts UDS and LDS pins to indicate a word transfer. The 68000 also outputs high on the R/W pin to indicate a read operation.

The 68000 now waits for the peripheral device to assert DTACK. Upon placing data on the data bus, the peripheral device asserts DTACK. The 68000 samples DTACK signal at the end of S4. If DTACK is not asserted by the peripheral device, the processor automatically inserts wait states (W).

However, upon assertion of DTACK, the 68000 negates AS, UDS, and LDS signals and then latches the data from data bus into an internal register at the end of the next cycle. Once the selected peripheral device senses that the 68000 has obtained data from the data bus (by recognizing the negation of AS, UDS, or LDS), the peripheral device must negate DTACK immediately, so that it does not interfere with the start of the next cycle.

If DTACK is not asserted by peripherals at the end of state 4 (Figure 5.16), the 68000 inserts wait states. The 68000 outputs valid addresses on the address pins and keeps asserting AS, UDS, and LDS until the peripheral asserts DTACK. The 6800 always inserts an even number of wait states if DTACK is not asserted by the peripheral, since all 68000 operations are performed using the clock with two states per clock cycle. Note that in Figure 5.16, the 6800 inserts 6 wait states or 3 cycles.

As an example of word read, consider that the 68000 is ready to execute the MOVE.W $602122, D0 instruction. The 68000 performs the following:

1. The 68000, at the end of S0, places the upper 23 bits of the address $602122\text{\_}w on A1-A23.
2. At the end of S1 state, the 68000 asserts AS, UDS, and LDS.
3. The 68000 continues to output high on R/W pin from the beginning of the read cycle, indicating a READ operation.
4. At the end of S0, the 68000 places appropriate outputs on FC2-FC0 pins to indicate whether supervisor or user read.
5. If the peripheral asserts DTACK at the end of S4, the 68000 reads the contents of $602122\text{\_}w and $602122\text{\_}\text{\_}w via D8-D15 pins and D0-D7 pins, respectively, into high and low bytes of D0 at the end of S6. If the peripheral does not assert DTACK at the end of S4, the 68000 continues to insert wait states.

Consider Figure 5.16 for 68000 write word timing. The 68000 outputs the address of the location to be written into the address bus at the start of S1. During S0, the 68000 places the proper function code values at the FC2, FC1, and FC0 pins. If the 68000 uses the data bus in the previous cycle, then it places all data pins in the high impedance state and then outputs LOW on AS and R/W pins. At the start of S3, the 68000 places data on D0-D15 pins. The 6800 then asserts UDS and LDS pins at the beginning of S4.

For the memory or I/O device, if DTACK is not asserted by memory or I/O device by the end of S4, the 68000 automatically inserts wait states into the write cycle.

The 68000 provides a special cycle called the read-modify-write cycle.
during execution of only the TAS instruction. This instruction reads a data byte, sets condition codes according to the byte value, sets bit 7 of the byte, and then writes the byte back into memory. The TAS instruction can be used in providing data transfer between two 68000 processors using shared RAM. The data byte mentioned above is held in the shared RAM. The read/modify/write cycle is indivisible. That is, it cannot be interrupted by any other bus request.

5.10 68000 MEMORY INTERFACE

One of the advantages of the 68000 is that it can easily be interfaced to memory chips. This is because the 68000 goes into a wait state if DTACK is not asserted by the memory devices at the end of S4.

A simplified schematic showing an interface of two 2716s and two 6116s is shown in Figure 5.17. The 2716 is a 2 K bit static RAM. For a 4 MHz clock, each cycle is 250 ns. The 68000 samples DTACK at the falling edge of S4 (third clock cycle) and latches data at the falling edge of S6 (fourth clock cycle). A5 is used to assert DTACK. A5 goes to LOW after 500 ns (two clock cycles). The time delay between A5 going LOW and the falling edge of S6 is 500 ns.

Since the access times of the 2716 and 6116 are, respectively, 450 and 120 ns, delay devices for DTACK are not required. Note that LDS and UDS must be used as chip selects as in the figure. They must not be connected to A0 of the memory chips, since in that case half of the memory in each chip will be wasted. For example, consider Figure 5.18 in which A0 of the even 2716 is connected to 68000 UDS pin via an inverter. Assume that the 2716's OE and CE are enabled (not shown in the figure).

The 2716 has eleven address pins. These 11-bit locations accessed by the 68000 in the 2716s can be obtained as follows:

Even 2716 (when UDS = 0, A0 is 0)

A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0 Hex Address
0 0 0 0 0 0 0 0 0 0 0 0000 0 0 0 0 0 0 0 0 0 0 0002
0 0 0 0 0 0 0 0 0 0 0 0010
0 0 0 0 0 0 0 0 0 0 0 0011
1 1 1 1 1 1 1 1 1 1 1 1110 57FE
Odd 2716 (when \( \text{LDS} = 0 \), \( A0 = 1 \))

A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
0 0 0 0 0 0 0 0 0 0 1 = $001
0 0 0 0 0 0 0 0 0 1 1 = $003

The memory map includes the addresses $000000$, $000002$, $000004$, \ldots $0000FF$.

Memory map for odd 2716 (A12 must be 0 to select odd 2716 and A13 must be 0 to deselect odd 6116)

A23 A22 A21 A20 A19 A18 A17 A16 A15 A14 A13 A12 A11 \ldots A1 A0
0 0 0 0 0 0 0 0 0 0 0 0 1 \ldots 1

The memory map includes the addresses $000001$, $000003$, $000005$, \ldots $0000FF$.

Memory map for even 6116 (A12 must be one to deselect even 2716 and A13 must be one to select even 6116)

A23 A22 A21 A20 A19 A18 A17 A16 A15 A14 A13 A12 A11 \ldots A1 A0
0 0 0 0 0 0 0 0 0 0 0 1 1 \ldots 1

The memory map includes the addresses $003000$, $003002$, \ldots $0030FF$.

Memory map for odd 6116 (A12 must be one to deselect odd 6116 and A13 must be one to deselect even 6116)

A23 A22 A21 A20 A19 A18 A17 A16 A15 A14 A13 A12 A11 \ldots A1 A0
0 0 0 0 0 0 0 0 0 0 0 1 \ldots 1

The memory map includes the addresses $003001$, $003003$, \ldots $0030FF$.

In summary, the memory for the schematic in Figure 5.16 is shown in the following:

\[
\begin{array}{lcccc}
\text{EVEN 2716 EPROM} & \text{000000} & \text{000002} & \text{000004} & \ldots & \text{0000FF} \\
\text{ODD 2716 EPROM} & \text{000001} & \text{000003} & \text{000005} & \ldots & \text{0000FF} \\
\text{EVEN 6116 RAM} & \text{003000} & \text{003002} & \text{003004} & \ldots & \text{0030FF} \\
\text{ODD 6116 RAM} & \text{003001} & \text{003003} & \text{003005} & \ldots & \text{0030FF}
\end{array}
\]
In the following, some examples will be considered to illustrate the use of memory map of the schematic of Figure 5.17 due to execution of 68000 MOVE instruction.

Consider MOVE.B S000004, D. Upon execution of this MOVE instruction, the 68000 reads the [S000004] from even 2716 to low byte of D1. In order to execute the instruction, the processor places the upper 23 bits of S000004 on its A23-A1 pins, and asserts UDS and AS. Since A12 = 0, the output of AND gate 2 = 1, the output of AND gate 3 generates a LOW, making CE and OE of the even 2716 LOW. When the 68000 samples DTACK (asserted by AS in Figure 5.17), data placed on the 68000 D8-D15 lines by the even 2716 are read by the processor into low byte of D1. Similarly, a byte read operation from the odd 2716 can be explained.

Consider MOVE.W S000004, D1. The 68000 asserts both UDS and LDS in this case. The outputs of AND gates 3 and 5 are LOW. Both the even 2716 and the odd 2716 are selected. Data placed on D0-D7 pins and D8-D15 pins of the 68000 from locations S000004 and S000005 of the even 2716 and odd 2716 are read by the processor, respectively, into bits 8-15 and bits 0-7 of D0.

Consider MOVE.B D2, S000001. The 64000 asserts LDS, AS and outputs HIGH on A12 and A13. The output of AND gate 6 is LOW and thus selects odd 6116. Since R/W = 0, the odd 6116 writes the low byte of D2 from 68000 D0-D7 pins into location S000001. Similarly, other byte operations from the RAMs can be illustrated.

Consider MOVE.L D3, S000002. The 68000 asserts AS, and both UDS and LDS. It also outputs HIGH on A12 and A13. Both RAMs are selected. The high 16 bits of the 32-bit data placed on D0-D7 pins by the 68000 are written into locations S000002 (byte 0) and S000003 (byte 1), and the low 16 bits of the 32-bit data placed on D8-D15 pins by the 68000 are written into locations S000004 (byte 2) and S000005 (byte 3), respectively, of the even and odd 6116. Similarly, the other long word operations can be illustrated.

5.11 68000 PROGRAMMED I/O

As mentioned before, the 68000 uses memory-mapped I/O. Programmed I/O can be achieved in the 68000 using one of the following ways:

1. By interfacing the 68000 synchronously with its own family of peripheral devices such as the MC68230, Parallel Interface/Timer chip.
2. By interfacing the 68000 synchronously with 6800 peripherals such as the MC6821 (note that synchronization means that every READ or WRITE operation is synchronized with the clock).

5.11.1 68000-68230 INTERFACE

The MC68230 parallel interface/timer (PI/T) provides double buffered parallel interfaces and a timer for 68000 systems. Note that double buffering means that the ports have dual latches. Double buffering allows simultaneous reading of data from a port by the microprocessor and placing of data onto the same port by an external device via handshaking. Double buffering is most useful in situations where a peripheral device and the processor are capable of transferring data at nearly the same speed. If there is a large difference in speed between the microprocessor and the peripheral, little or no benefit of double buffering is achieved. In these cases, however, there is no penalty for using double buffering. Double buffering permits the fetch operation of the data transmitter to be overlapped with the store operation of the data receiver.

The parallel interfaces provided by the 68230 can be 8 or 16 bits wide with unidirectional or bidirectional modes. In the unidirectional mode, a data direction register configures each port as an input or output. In the bidirectional mode, the data direction registers are ignored and the direction is determined by the state of four handshake pins.

The 68230 allows use of interrupts, and also provides a DMA request pin for connection to a DMA controller chip such as the MC68450. The timer contains a 24-bit-wide counter. This counter can be clocked by the output of a 5-bit (divide by 32) prescaler or by an external timer input pin (TEN).

Table 5.15 provides the signal summary and Figure 5.19 shows the 68230 pin diagram. The 68230 is a 48-pin device.

The purpose of D0-D7, R/W, and CS pins is obvious. RS1-RS5 are five register select input pins for selecting the 32 internal registers.

During read or interrupt acknowledge cycles, DTACK is asserted after data have been provided on the data bus and during write cycles it is asserted after data have been accepted at the data bus. A pullup resistor is required to maintain DTACK high between bus cycles.

Upon activation of the RESET input, all control and data direction registers are cleared and most internal operations are disabled by the assertion of RESET LOW.
TABLE 5.15
Signal Summary of the MC68230

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Input/ output</th>
<th>Active state</th>
<th>Edge/level sensitive</th>
<th>Output states</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK</td>
<td>Input</td>
<td>Low</td>
<td>Falling and rising edge</td>
<td>High, low, high impedance</td>
</tr>
<tr>
<td>CS</td>
<td>Input</td>
<td>Low</td>
<td>Level</td>
<td>High, low, high impedance</td>
</tr>
<tr>
<td>D0-D7</td>
<td>Input/output</td>
<td>High = 1, low = 0</td>
<td>Level</td>
<td>High, Low, high impedance</td>
</tr>
<tr>
<td>RAMBREQ</td>
<td>Output</td>
<td>Low</td>
<td>Level</td>
<td>High, Low, high impedance</td>
</tr>
<tr>
<td>DTACK</td>
<td>Output</td>
<td>Low</td>
<td>Asserted edge</td>
<td>High, low, high impedance</td>
</tr>
<tr>
<td>H1</td>
<td>Input/output</td>
<td>Low or high</td>
<td>Asserted edge</td>
<td>High, low, high impedance</td>
</tr>
<tr>
<td>H2</td>
<td>Input/output</td>
<td>Low or high</td>
<td>Asserted edge</td>
<td>High, low, high impedance</td>
</tr>
<tr>
<td>PA0-P6, PA7</td>
<td>Input/ output</td>
<td>High = 1, low = 0</td>
<td>Level</td>
<td>High, low, high impedance</td>
</tr>
<tr>
<td>PB0-PB7, PC0-PG7</td>
<td>Input/ output</td>
<td>High = 1, low = 0</td>
<td>Level</td>
<td>High, low, high impedance</td>
</tr>
<tr>
<td>PASC</td>
<td>Input</td>
<td>Level</td>
<td>Low</td>
<td>Low, high impedance</td>
</tr>
<tr>
<td>IRCG</td>
<td>Input</td>
<td>High = 1, low = 0</td>
<td>Level</td>
<td>Low, high impedance</td>
</tr>
<tr>
<td>RST, RSR</td>
<td>Input</td>
<td>Low</td>
<td>Level</td>
<td>Low, high impedance</td>
</tr>
<tr>
<td>~RST</td>
<td>Input</td>
<td>Low</td>
<td>Rising edge</td>
<td>High, low</td>
</tr>
<tr>
<td>TACQ</td>
<td>Input</td>
<td>Low or high</td>
<td>Rising edge</td>
<td>Low, High, low</td>
</tr>
<tr>
<td>TIN (external clock)</td>
<td>Input</td>
<td>Low</td>
<td>Rising edge</td>
<td>High, Low</td>
</tr>
<tr>
<td>TIN (software)</td>
<td>Input</td>
<td>Low</td>
<td>Rising edge</td>
<td>High, Low</td>
</tr>
<tr>
<td>TOUT (tacq)</td>
<td>Output</td>
<td>Low</td>
<td>Level</td>
<td>Low, High, high impedance</td>
</tr>
<tr>
<td>TOUT (tacq)</td>
<td>Output</td>
<td>Low</td>
<td>Level</td>
<td>Low, High, high impedance</td>
</tr>
</tbody>
</table>

* Pullup resistors required.
* The clock pin has the same specifications as the 68000.

PAD-P7 and PB0-PB7 pins provide two 8-bit ports that may be concatenated to form a 16-bit port in certain modes. The ports may be controlled in conjunction with handshake pins H1-H4. A simple example of a handshake operation for input of data is for the I/O device to indicate to the port that new data are available at the port by activating H1 to HIGH. After input of data by the 68000, the H2 output of the 68230 is set to HIGH to indicate to the I/O device that data have been read and it may now provide another data byte to the port.

Handshake pins H1-H4 are multipurpose pins that provide an interlocked handshake, a pulsed handshake, an interrupt input, or simple I/O pins. The interlocked and pulsed I/O handshake modes are selected via port A control register (PACR) and port B control register (PBCR). For pulsed or interlocked mode, H1 and H2 are used with port A, while H3 and H4 are used with port B.

In the interlocked I/O handshake protocol, the handshake pins H2 and H4 are defined as output pins. For interlocked input protocol, H2 and H4 are asserted automatically by the 68230 when the input lines are ready to accept new data. When both double-buffered latches are full, H2 (or H4) remain negated until data are removed by a read of port A (or port B) data register. Therefore, any time the H2 (or H4) output is asserted, new data may be input by asserting H1 (or H3).

In the interlocked output protocol, H2 and H4 are also output pins. H2 (or H4) is asserted two clock cycles after data are transferred to the double-buffered output latches. The data remain stable and H2 (or H4) stays asserted until the next asserted edge of H1 (or H3). At that time, H2 (or H4) is synchronously negated. As soon as the next data are available, they are output to the latches, and H2 (or H4) is asserted. H1 (or H3) is used by the external device to receive data.

For pulsed input handshake protocol, H2 (or H4) is asserted exactly as
the interlocked input handshake and output protocol up to four clock cycles. As soon as the selected edge (HI or LO) occurs, the handshake protocol begins. HI (or LO) is an optional feature that allows the user to select the edge (HI or LO) that triggers the handshake protocol.

Table 5.11 provides the 68230 register addressing assignments. Table 5.17 shows each of the 68230 registers and the functions of each of the bits. These registers control the input and output pins to or from the I/O device and provide data distribution of the ports and the

<table>
<thead>
<tr>
<th>Register Address</th>
<th>Register Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000h</td>
<td>Port A</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0001h</td>
<td>Port B</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0002h</td>
<td>Port C</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0003h</td>
<td>Port D</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0004h</td>
<td>Port E</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0005h</td>
<td>Port F</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0006h</td>
<td>Port G</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0007h</td>
<td>Port H</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0008h</td>
<td>Port I</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0009h</td>
<td>Port J</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>000Ah</td>
<td>Port K</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>000Bh</td>
<td>Port L</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>000Ch</td>
<td>Port M</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>000Dh</td>
<td>Port N</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>000Eh</td>
<td>Port O</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>000Fh</td>
<td>Port P</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0010h</td>
<td>Port Q</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0011h</td>
<td>Port R</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0012h</td>
<td>Port S</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0013h</td>
<td>Port T</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0014h</td>
<td>Port U</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0015h</td>
<td>Port V</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0016h</td>
<td>Port W</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0017h</td>
<td>Port X</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0018h</td>
<td>Port Y</td>
<td>Data In/Out</td>
</tr>
<tr>
<td>0019h</td>
<td>Port Z</td>
<td>Data In/Out</td>
</tr>
</tbody>
</table>

Table 5.16 shows the relationship between the data in the 68230 registers and the handshake protocol. It also provides the function of each bit.
### Table 5.17
68230 Register Modes

<table>
<thead>
<tr>
<th>Register select bits</th>
<th>Register value after RESET (hex value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0</td>
<td>Port Mode Control</td>
</tr>
<tr>
<td>0 0 0 0 0 1</td>
<td>SVCRCQ Select</td>
</tr>
<tr>
<td>0 0 0 0 1 0</td>
<td>Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0</td>
</tr>
<tr>
<td>0 0 0 1 1 1</td>
<td>Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0</td>
</tr>
<tr>
<td>0 0 1 0 0 0</td>
<td>Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0</td>
</tr>
<tr>
<td>0 1 0 1 0 1</td>
<td>Internal Vector Number</td>
</tr>
<tr>
<td>0 1 1 0 0 0</td>
<td>Port A Submode</td>
</tr>
<tr>
<td>0 1 1 1 1 1</td>
<td>Port B Submode</td>
</tr>
<tr>
<td>0 1 0 0 0 0</td>
<td>Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0</td>
</tr>
<tr>
<td>0 1 0 0 0 1</td>
<td>Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0</td>
</tr>
<tr>
<td>0 1 0 1 0 1</td>
<td>Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0</td>
</tr>
</tbody>
</table>

### Table 5.17 (continued)
68230 Register Modes

<table>
<thead>
<tr>
<th>Register select bits</th>
<th>Register value after RESET (hex value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0</td>
<td>Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0</td>
</tr>
<tr>
<td>0 0 0 0 1 1 1</td>
<td>Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0</td>
</tr>
<tr>
<td>0 0 0 1 0 1 1</td>
<td>Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0</td>
</tr>
<tr>
<td>0 0 1 0 0 0 0</td>
<td>Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0</td>
</tr>
<tr>
<td>0 1 0 1 0 0 0</td>
<td>Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0</td>
</tr>
<tr>
<td>0 1 1 0 1 1 1</td>
<td>Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0</td>
<td>Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0</td>
</tr>
</tbody>
</table>
TABLE 5.17 (continued)
68230 Register Modes

<table>
<thead>
<tr>
<th>Register select bits</th>
<th>Register value after RESET (hex value)</th>
<th>** Count Register (Low)</th>
<th>** Timer Status Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1</td>
<td>1 1 0 0 1</td>
<td>0 0</td>
<td>0 0 (Null)</td>
</tr>
<tr>
<td>1 1 0 1 0</td>
<td>1 1 0 1 0</td>
<td>0 0</td>
<td>0 0 (Null)</td>
</tr>
<tr>
<td>1 1 1 0 1</td>
<td>1 1 1 0 1</td>
<td>0 0</td>
<td>0 0 (Null)</td>
</tr>
<tr>
<td>1 1 1 0 0</td>
<td>1 1 1 0 0</td>
<td>0 0</td>
<td>0 0 (Null)</td>
</tr>
<tr>
<td>1 1 1 1 0</td>
<td>1 1 1 1 0</td>
<td>0 0</td>
<td>0 0 (Null)</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
<td>1 1 1 1 1</td>
<td>0 0</td>
<td>0 0 (Null)</td>
</tr>
</tbody>
</table>

Note: *Unused, read as zero; ** value before RESET; *** current value on pins; **** undetermined value.

The 68230 ports can be configured for various modes of operation. For example, consider ports A and B. Bits 6 and 7 of the port general control register, PGCR (R6) are used for configuring ports A and B in one of four modes as follows:

- **PGCR**
  - bits 6 7
  - 00 Mode 0 (unidirectional 8-bit mode)
  - 01 Mode 1 (unidirectional 16-bit mode)
  - 10 Mode 2 (bidirectional 8-bit mode)
  - 11 Mode 3 (bidirectional 16-bit mode)

The other pins of PGCR are defined as follows:

The modes 0 and 2 configure ports A and B as unidirectional or bidirectional 8-bit ports. Modes 1 and 3, on the other hand, combine ports A and B together to form a 16-bit unidirectional or bidirectional port. Ports configured as unidirectional must further be programmed as sub-modes of operation using bits 7 and 6 of PACR (R6) and PBCR (R7) as follows:

For unidirectional 8-bit mode (mode 0)

- Bit 7 of PACR or PBCR
- Bit 6 of PACR or PBCR

<table>
<thead>
<tr>
<th>Submode</th>
<th>00</th>
<th>01</th>
<th>0X</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

Pin-definable double-buffered input or single-buffered output
Pin-definable double-buffered output or non-latched input
Bit I/O (pin-definable single-buffered output or non-latched input)
Note that in the above X means don’t care. Nonlatched inputs are latched internally but the bit values are not available at the port.

The submodes define the ports as parallel input ports, parallel output modes, or bit-configurable l/O ports. In addition to these, the submodes further define the ports as latched input ports, interrupt-driven ports, DMA ports, and with various l/O handshake operations. From Table 5.17 it can be seen that R0 through R13 registers of the 68230 configure and control the operation of these ports. Therefore, some of these registers are considered in detail in the following.

As mentioned earlier, PGCR is register R0. Bits 7 and 6 of R0 configure ports A and B in one of four modes of operation. The other bits of PGCR are used to enable and sense the handshake lines H1 through H4. For example, bit 5 of PGCR is the H3 enable bit. It must be set to HIGH to enable the H3 and H4 lines. Similarly, bit 12 is used to enable or disable the H1 and H2 lines. Bits 6 through 3 of PGCR are sense bits for H1 through H4. These bits are programmable. For example, setting one or more of these bits to one will result in raising the active level of the corresponding handshake pin (H1-H4). On the other hand, making one or more of these bits to 0 will set the low voltage level as the active state.

Register R1 is the port service request register (PSRR). This control port is used to interface 68230 to 68000 for interrupt-driven operation of I/O. The bits in this port are used

1. To define the bits of port C as interrupt request and interrupt acknowledge signals instead of I/O lines
2. To set up a priority scheme for the handshake lines H1-H4
3. To operate ports A and B in DMA mode

In Figure 5.20, the format for R1 is shown. Bits 5 and 6 of PSRR are identified as SVCRQ (service request) select. These bits define whether the PC4/Dmareq pin of port C is used as an I/O pin PC4 or as the DMA request pin, DMAreq. Outputting 0 to bit 6 of PSRR configures the PC4/DMAreq pin as an I/O pin and outputting 1 makes it the DMAreq pin. The DMA mode, bit 6 of PSRR defines whether DMA operations with the selected port are with respect to H1 or H3. For example, bits 6, 5 of PSRR as L1 will choose DMA operation with H3 and port B.

Bits 3 and 4 configure the operation of the PC5/PIreq and PC6/PIack pins of the 68230. For example, outputting 10 to these bits will select PC5/PIreq as an I/O pin and PC6/PIack as the PIack pin.

Finally, bits 0, 1, and 2 define H2 as the highest priority, H4 next highest, H3 next, and H4 as the lowest.
Registers R2, R3, and R4 are, respectively, the port A data direction register (PADRR), port B data direction register (PBDDR), and port C data direction register (PCCDDR). The bits in these control ports configure the corresponding bits in the associated ports as input or output for unidirectional mode of operation. A '0' in a particular bit position in a port DDR is used to configure the corresponding bit in the associated port as input and a '1' is used to program the bit as output. For example, to set all bits of port A as input, 000 must be output to PADRR.

R5 is the interrupt vector register (PIVR). It is used with the interrupt-driven mode for the ports. The upper 6 bits of an interrupt vector number are loaded into the bits 2-7 of PIVR.

Bits 0 and 1 of PIVR are provided internally by the 68230 for indicating the priority of the active handshake line. R6 and R7 are, respectively, the port A control register (PACR) and port B control register (PBCR). As mentioned earlier, bits 7 and 6 of these ports define the submodes. The formats for PACR and PBCR are shown in Figure 5.21. Figure 5.22 provides port mode layout and Table 5.18 shows port mode control summary. In the figure, for mode 0 (defined by PGCR) with submode 00, port A is defined as a 8-bit latched double-buffered input port. By port A latching, it is meant that at the logic level transition of H1 data at port A pins are latched internally by the 68230 flip-flops. Note that the active level of H1 can be set to 1 or 0 by the sense bit in PGCR. As mentioned before, double buffering means that there are dual latches in the 68230 ports. This allows simultaneous reading of port A data by the 68090 and placing of new data into port A by the I/O device.

The other bits of PACR and PBCR are for handshake operations. R8 and R9 are port A and port B data registers, PADR and PABDR. Each bit in these registers refers to one of the bits in the corresponding 1/0 port.

In the following, an example is given to illustrate interfacing of 68230 to a 68090 in a simplified manner. The 68230 ports A and B are used along with mode 0 (bits 7, 6 of PGCR = 00).

In mode 0, bit 1/0 is available by programming submode LX in PACR or PBCR for port A or port B. This submode can be used in applications where several independent devices are to be controlled or monitored. Data output to the specified port is single buffered. If a bit in the data direction register is one (output), the output buffer is enabled. If it is zero (input), the register is not available at the pin. Data read from the data register are the instantaneous value of the pin or what was written to the data register, depending on the contents of the data direction register.
Chapter 5

5.11 68000 Programmed I/O

FIGURE 5.21 (continued)

FIGURE 5.22 Port mode layout.
<table>
<thead>
<tr>
<th>TABLE 5.18</th>
<th>Port Mode Control Summary</th>
</tr>
</thead>
</table>

**Mode 0 (unidirectional 8-bit mode)**
- **Port A**
  - Submode 00 — pin-definable double-buffered input or single-buffered output
    - H1 — latches input data
    - H2 — status/interrupt generating input, general purpose output, or operation with H1 in the interlocked or pulsed handshake protocols
- Submode 01 — pin-definable double-buffered output or nonlatched input
  - H1 — indicates data received by peripheral
  - H2 — status/interrupt generating input, general purpose output, or operation with H1 in the interlocked or pulsed handshake protocols
- Submode 1X — pin-definable single-buffered output or nonlatched input
  - H1 — status/interrupt generating input
  - H2 — status/interrupt generating input or general purpose output
- **Port B**
  - H3 and H4 — identical to port A, H1 and H2

**Mode 1 (unidirectional 16-bit mode)**
- **Port A** — most significant data byte or nonlatched input or single-buffered output
  - Submode XX — (not used)
    - H1 — status/interrupt generating input
    - H2 — status/interrupt generating input or general purpose output
- **Port B** — least significant data byte
  - Submode X0 — pin-definable double-buffered input or single-buffered output
    - H3 — latches input data
    - H4 — status/interrupt generating input, general purpose output, or operation with H3 in the interlocked or pulsed handshake protocols
  - Submode X1 — pin-definable double-buffered output or nonlatched input

**Mode 2 (bidirectional 8-bit mode)**
- **Port A** — bit 0/1
  - Submode XX — (not used)
- **Port B** — double-buffered bidirectional data
  - Submode XX — (not used)
    - H1 — indicates output data received by the peripheral and controls output drivers
    - H2 — operation with H1 in the interlocked or pulsed output handshake protocols
    - H3 — latches input data
    - H4 — operation with H3 in the interlocked or pulsed input handshake protocols

**Mode 3 (bidirectional 16-bit mode)**
- **Port A** — double-buffered bidirectional data (most significant data byte)
  - Submode XX — (not used)
- **Port B** — double-buffered bidirectional data (least significant data byte)
  - Submode XX — (not used)
    - H1 — indicates output data received by peripheral and controls output drivers
    - H2 — operation with H1 in the interlocked or pulsed output handshake protocols
    - H3 — latches input data
    - H4 — operation with H3 in the interlocked or pulsed input handshake protocols

---

Figure 5.23 shows a simplified schematic for the 68000-68230 interface. A23 is chosen to be HIGH to select the 68230 chip so that the port addresses are different than the 68000 reset vector addresses 0000000 through 000006F. The configuration in the figure will provide even port addresses, since PTE is used for enabling the 68230 CS.
From the figure, addresses are for registers PGCR (R0), PADDR (R2), PBDTR (R3), PACR (R6), PBCTR (R7), PADDR (R8), and PBDTR (R9).

Consider PGCR.

\[
\begin{align*}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 &= 800000 \\
\end{align*}
\]

Similarly,

- address for PADDR = 800000
- address for PBDTR = 800004
- address for PACR = 800006
- address for PBCTR = 800008
- address for PADDR = 80000C
- address for PBCTR = 80000E
- address for PBDTR = 800010
- address for PBDTR = 800012

As an example, the following instruction sequence will select mode 0, submode 1x and configure bits 0-3 of port A as outputs, bits 4-7 as inputs, and port B as an output port:

- MOV B #600, PGCR ; Select mode 0
- MOV B #8FF, PADDR ; Port A bit I/O submode
- MOV B #8FF, PBCTR ; Port B bit I/O submode
- MOV B #800, PBDTR ; Configure Port A bits 0-3 as outputs and bits 4-7 as inputs
- MOV B #8FF, PBDTR ; Configure Port B as an output port.

5.11.2 MOTOROLA 68000-6821 INTERFACE

The Motorola 6821 is a 40-pin peripheral interface adapter (PIA) chip. It is provided with an 8-bit bidirectional data bus (D0-D7), two register select lines (RS0, RS1), read/write line (R/W), reset line (RESET), an enable line (E), two 8-bit I/O ports (PA0-PA7) and PB0-PB7, and other pins.

There are six 6821 registers. These include two 8-bit ports (ports A and B), two data direction registers, and two control registers. Selection of these registers is controlled by the RS0 and RS1 inputs together with bit 2 of the control register. Table 5.19 shows how the registers are selected.

In Table 5.19 bit 2 in each control register (CRA-2 and CRB-2) determines selection of either an I/O port or the corresponding data direction register when the proper register select signals are applied to RS0 and RS1. A 1 in bit 2 allows access of I/O ports, while a 0 selects the data direction registers.

Each I/O port bit can be configured to act as an input or output. This is accomplished by setting a 1 in the corresponding data direction register bit for those bits which are to be output and a 0 for those bits which are to be inputs.

A RESET signal sets all PIA registers to 0. This has the effect of setting PA0-PA7 and PB0-PB7 as inputs.
### TABLE 5.19
6821 Register Definition

<table>
<thead>
<tr>
<th>Control register bits 2</th>
<th>Register selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS1 RS0 CRA-2 CRB-2</td>
<td></td>
</tr>
<tr>
<td>0 0 1 X</td>
<td>I/O port A</td>
</tr>
<tr>
<td>0 0 0 X</td>
<td>Data direction register A</td>
</tr>
<tr>
<td>0 1 X X</td>
<td>Control register A</td>
</tr>
<tr>
<td>1 0 X 1</td>
<td>I/O port B</td>
</tr>
<tr>
<td>1 0 X 0</td>
<td>Data direction register B</td>
</tr>
<tr>
<td>1 1 X X</td>
<td>Control register B</td>
</tr>
</tbody>
</table>

Note: X = Don’t care.

There are three built-in signals in the 68000 which provide the interface with the 6821. These are the Enable (E), Valid Memory Access (VMA), and Valid Peripheral Access (VPA).

The Enable signal (E) is an output from the 68000. It corresponds to the E signal of the 6821. This signal is the clock used by the 6821 to synchronize data transfer. The frequency of the E signal is one tenth of the 68000 clock frequency. Therefore, this allows one to interface the 68000 (which operates much faster than the 6821) with the 6821. The Valid Memory Address (VMA) signal is output by the 68200 to indicate to the 6800 peripherals that there is a valid address on the address bus.

The Valid Peripheral Address (VPA) is an input to the 68200. This signal is used to indicate that the device addressed by the 68000 is a 6800 peripheral. This tells the 68000 to synchronize data transfer with the Enable signal (E).

Let us now discuss how the 68000 instructions can be used to configure the 6821 ports. As an example, bit 7 and bits 0-6 of port A can be configured, respectively, as input and outputs using the following instruction sequence:

```
BCLR.B #62, CRA ; ADDRESS DDRA
MOVE.B #57F, DDRA ; CONFIGURE PORT A
BSET.B #62, CRA ; ADDRESS PORT A
```

Once the ports are configured to the designer’s specification, 6821 can be used to transfer data from an input device to the 68000 or from the 68000 to an output device by using the MOVE.B instruction as follows:

```
MOVE.B (EA), Dn TRANSFER 8-bit data from an input port to the specified data register Dn.
MOVE.B Dn,(EA) TRANSFER 8-bit data from the specified data register Dn to an output port.
```

Figure 5.23 shows a block diagram of how two 6821s are interfaced to the 68000 in order to generate four 8-bit I/O ports. Note that the least significant bit, A0, of the 68000 address pins is internally encoded to generate two signals called Upper Data Strobe (UDS) and Lower Data Strobe (LDS). For byte transfers UDS is asserted if an even numbered byte is being transferred and LDS is asserted for odd numbered byte.

In Figure 5.24, 1/O port addresses can be obtained as follows. When A23 is HIGH and A2 is LOW, the OR gate output will be LOW. This OR gate output is used to provide VPA. The inverted OR gate output, in turn, makes CS1 HIGH on both the chips. Note that A23 is arbitrarily chosen. A25 is chosen to be HIGH to enable CS1 so that the addresses for the ports and the reset vector are not the same. Assuming the don’t care address lines A22 A3 to be zero, the addresses for the I/O ports, control registers, and the data direction registers for the even 6821 can be obtained as follows:

<table>
<thead>
<tr>
<th>Port name</th>
<th>Memory address</th>
</tr>
</thead>
<tbody>
<tr>
<td>A23 A22</td>
<td>A3 A2 A1 A0</td>
</tr>
<tr>
<td>DDRA</td>
<td>0 0 0 0 = 800000h</td>
</tr>
<tr>
<td>CRA</td>
<td>0 0 0 1 = 800001h</td>
</tr>
<tr>
<td>DDBR</td>
<td>0 0 1 0 = 800003h</td>
</tr>
<tr>
<td>CRB</td>
<td>0 1 1 0 = 800006h</td>
</tr>
</tbody>
</table>

Note that in the above, A0 = 0 for even addressing. Also, from Table 5.19, for accessing DDRA, bit 2 of CRA with memory address 800002h must be set to 1 and then port A can be configured with appropriate data in 800000h. Note that port A and its data direction register, DDRA, have the same address, 800000h. Similarly, port B and DDBR have the same address 800004h. Bit 2 in CRA or CRB identifies whether address 800000h or 800004h is an I/O port or a data direction register.

Similarly, the addresses for the ports, control registers, and the data direction register for the odd 6821 (A0 = 1) can be determined as follows:

| Port A/DDRA (800001h), CRA (800003h), port B/DDBR (800005h), and CRB (800007h). |
Figure 5.25a shows the schematic of a 68000-based microcomputer with 4K EPROM, 4K Static RAM, and four 8-bit I/O ports.

Let us explain the various sections of the hardware schematic. Two 2716 and two 6116 chips are required to obtain the 4K EPROM and 4K RAM. The LDS and UDS pins are ORed with the memory select signal to enable the chip selects for the EPROMs and the RAMs.

Address decoding is accomplished by using a 3 x 8 decoder. The decoder enables the memory or I/O chips depending on the status of A12-A14 address lines and AS line of the 68000. AS is used to enable the decoder. Selects the EPROMs, II selects the RAMs, and II selects the I/O ports.

When addressing memory chips, DTACK input of the 68000 must be asserted for data acknowledge. The 68000 clock in the hardware schematic is 10 MHz. Therefore, each clock cycle is 100 nanoseconds. In Figure 5.25a, AS is used to enable the 3 x 8 decoder. The outputs of the decoder are gated to assert 68000 DTACK. This means that AS is indirectly used to assert DTACK. From the 68000 read timing diagram of Figure 5.16, AS goes to LOW after approximately two cycles (200 ns for 10 MHz clock) from the beginning of the bus cycle. With no wait states, the 68000 samples DTACK at the falling edge of S4 (300 ns) and, if recognized, the 68000 latches data at the falling edge of S6 (400 ns). If the DTACK is not recognized at the falling edge of S4, the 68000 inserts one cycle (100 ns in this case) wait state, samples DTACK at the end of S6, and, if recognized, latches data at the end of S8 (500 ns), and the process continues. Since the access time of 2716 is 480 ns, DTACK recognition by the 68000 at the falling edge of S6 (400 ns) and, hence, latching of data at the falling edge of S8 (500 ns) will satisfy the timing requirement. This means that the decoder output IO for ROM select must go LOW at the end of S6. Therefore, AS must be delayed by 200 ns, i.e., two cycles (S2 through S6).

A delay circuit (Figure 5.25b) is designed using a 74LS75 D-Flip-Flop. AS activates the delay circuit. The input is then shifted right two bits to obtain a two-cycle wait state to allow sufficient time for data transfer. DTACK assertion and recognition are delayed by two cycles during data transfer with the EPROMs. A timing diagram for the DTACK delay circuit is shown in Figure 5.25c.

When ROM is not selected by the decoder, then clear pin is asserted.
68000/2716/6116/6821-Based Microcomputer

**Figure 5.23b Delay circuit for DTACK.**

- **CLK** (10kHz)
  - ROM Set: not selected
  - ROM is selected
  - CLR
  - Q1
  - Q2

**Figure 5.23c Timing diagram for the DTACK delay circuit.**

(output of inverter). So, Q is forced LOW and Q̅ is high. Therefore, DTACK is not asserted. When the processor is addressing the ROMs, then the output of the inverter is LOW so clear pin is not asserted. Now, the D flip-flop will accept a high at the input, and Q will output high and Q̅ will output low. Now that Q is low, it can assert DTACK. Q1 will provide one
wait state and Q2 will provide two wait states. Since the 2716 EPROM has a 450-ns access time and the microprocessor is operating at 10 MHz (100 ns clock cycle), two wait states are inserted before asserting DTACK (2*100 = 200 ns). Therefore, Q2 can be connected to DTACK pin.

No wait state is required for RAMS as the access time for the RAMs is only 120 nanoseconds.

Four 8-bit I/O ports are obtained by using two 6821 chips. When the I/O ports are selected, the VPA pin is asserted instead of DTACK. This will acknowledge to the 68000 that it is addressing a 6800-type peripheral. In response, the 68000 will synchronize all data transfer with the E clock.

The memory and I/O maps for the schematic are shown below:

### Memory Mapping

<table>
<thead>
<tr>
<th>A23-A15 A14 A13 A12 A11-A1 A0</th>
<th>UDS or LDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-0 0 0 0 0-0 0</td>
<td>ROM (EVEN) = 2K</td>
</tr>
<tr>
<td>0-0 0 0 0 1-1 0</td>
<td>$000000, $000002, $000004</td>
</tr>
<tr>
<td>0-0 0 0 0 0-0 1</td>
<td>$000000, $000002, $000004</td>
</tr>
<tr>
<td>0-0 0 0 0 1-1 1</td>
<td>$000001, $000003, $000005</td>
</tr>
<tr>
<td>0-0 0 0 1 0-0 0</td>
<td>RAM (even) = 2K</td>
</tr>
<tr>
<td>0-0 0 0 1 1-1 0</td>
<td>$001000, $001002, ... $001FFE</td>
</tr>
<tr>
<td>0-0 0 0 1 0-0 1</td>
<td>RAM (odd) = 2K</td>
</tr>
<tr>
<td>0-0 0 0 1 1-1 1</td>
<td>$001001, $001003, ... $001FFF</td>
</tr>
</tbody>
</table>

### Memory Mapped I/O

<table>
<thead>
<tr>
<th>A23-A15 A14 A13 A12 A11-A1 A0</th>
<th>RS1 RS0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-0 0 1 0 0-0 0</td>
<td>Port A or DDRA = $002000</td>
</tr>
<tr>
<td>0-0 0 1 0 0-0 1</td>
<td>Port B or DDRB = $002004</td>
</tr>
<tr>
<td>0-0 0 1 0 0-0 1</td>
<td>CRB = $002006</td>
</tr>
</tbody>
</table>

Note that upon hardware reset, the 68000 loads the supervisor SP high and low words, respectively, from addresses $000000 and $000002 and the PC high and low words, respectively, from locations $000004 and $000006. The memory map of Figure 5.25a contains these reset vector addresses in the even and odd 2716.

### Example 5.8

Assume (assume EPROM/RAM in the system) that in the configuration of Figure 5.26, port A has three inputs and an LED connected to bits 0-3. Port B has an LED connected to bit 3. Write 68000 assembly program to:

1. Turn the port A LED ON and port B LED OFF if port A has an even number of high switch inputs.
2. Turn the port A LED OFF and port B LED ON if port A has an odd number of high switch inputs.
3. Turn both LEDs off if there are no high switch inputs.

Figure for Problem 5.8.
Solution

PCGR EQU $800000
PAADR EQU $800004
PBDDR EQU $800005
PACR EQU $80000C
PSCR EQU $80000E
PADR EQU $800013
PRDR EQU $800012

MOVE.B #600, PCGR ; Select mode 0
MOVE.B #687, PACR ; Port A bit 1/0 submode
MOVE.B #587, PSCR ; Port B bit 1/0 submode
MOVE.B #5C8, PBDDR ; Configure port A
MOVE.B #5C8, PDADR ; Configure port B
MOVE.B PADR, D1 ; Get port A switches
ANDI.B #5C7, D1 ; Mask high five bits
                   ; Are all three inputs low
BEQ LOW           ; If so, turn both LEDs off.

CMPI.B #363, D1 ; Are high switch inputs even
BEQ EVEN          ; If so, turn port A LED on and port B LED OFF.

CMPI.B #845, D1 ; Are high switch inputs even
BEQ EVEN          ; If so, turn port A LED on and port B LED OFF.

CMPI.B #846, D1 ; Are high switch inputs even
BEQ EVEN          ; If so, turn port A LED on and port B LED OFF.

CMPI.B #837, D1 ; Are high switch inputs even
BEQ ODD           ; If so, turn port A LED off and port B LED ON.

CMPI.B #834, D1 ; Are high switch inputs even
BEQ ODD           ; If so, turn port A LED off and port B LED ON.

5.13 68000 Interrupt I/O

The 68000 services interrupts in the supervisor mode. The 68000 interrupt I/O can be divided into two types: external interrupts and internal interrupts.

5.13.1 EXTERNAL INTERRUPTS

The 68000 provides seven levels of external interrupts, 1 through 7. The external hardware provides an interrupt level using the pins IPED, IPDL, IPDR. Like other processors, the 68000 checks for and accepts interrupts only between instructions. It computes the value of inverted IPDL-IPDR with the current interrupt mask contained in the bits 10, 9, and 8 of the status register.

If the value of the inverted IPDL-IPDR is greater than the value of the current interrupt mask, then the processor acknowledges the interrupt and initiates interrupt processing. Otherwise, the 68000 continues with the current interrupt. Interrupt request level zero (IPDL=IPDR=HIGH) indicates that no interrupt service is requested. An inverted IPDL, IPDL, IPDL of 7 is always acknowledged and has the highest priority. Therefore,
interrupt level 7 is "nonmaskable." Note that the interrupt level is indicated by the interrupt mask bits (inverted IPE, IP1, IPEL). To ensure that an interrupt will be recognized, the following interrupt rules should be considered:

1. The incoming interrupt request level must be at a higher priority level than the mask level set in the interrupt mask bits (except for level 7, which is always recognized).
2. The IPE, IPEL, IP1 bits must be held at the interrupt request level until the 68000 acknowledges the interrupt by initiating an interrupt acknowledge (IACK) bus cycle.

Interrupt level 7 is edge-triggered. On the other hand, the interrupt levels 1 to 6 are level sensitive. As soon as one of them is acknowledged, the processor updates its interrupt mask to the same level.

The 68000 does not have any EI (Enable Interrupt) or DI (Disable Interrupt) instructions. Instead, the level indicated by ILI 10 at the SR disables all interrupts below or equal to this value and enables all interrupts above this. For example, in the supervisor mode, ILI 10 and ILI can be modified by using instructions such as AND with SR. If ILI 10 and ILI are modified to contain 100, then interrupt levels 1 to 4 are disabled and levels 5 to 7 are enabled. Note that ILI 10 = 111 disables all interrupts.

Upon hardware reset, the 68000 operates in supervisor mode and sets ILI 10 to 111, and disables the interrupt levels 1 through 6. Note that if ILI 10 is modified to 110, the 68000 also disables levels 1 through 6 and level 7, of course, always enabled.

Once the 68000 has decided to acknowledge an interrupt request, it pushes PC and SR onto the stack, enters supervisor state by setting 8-bit to 1, clears TF to inhibit tracing, and updates the priority mask bits and also the address lines A1-A1 with the interrupt level. The 68000 then asserts AS to inform the external devices that AS-A1 has the interrupt level. The processor sets EC2 EC1 FC0 to 111 to run an IACK cycle for 8-bit vector number acquisition. The 68000 multiplies the 8-bit vector by 4 to determine pointer to locations containing the starting address of the service routine. The 68000 then branches to the service routine. The last instruction of the service routine should be RTE which pops PC and SR back from the stack. In order to explain how the 68000 interrupt priorities work, assume that ILI 10 = 100 in SR has the value 101. This means that levels 1, 2, and 3 are disabled and levels 4, 5, 6, and 7 are enabled. Now, if the 68000 is interrupted with level 5, the 68000 pushes PC and SR, updates ILI 10 in SR with 101, and loads PC with the starting address of the service routine.

Now, while in the service routine of level 5, if the 68000 is interrupted by level 6, the 68000 pushes PC and SR onto the stack. The 68000 then completes execution of the level 6 interrupt. The RTE instruction at the end of the level 6 service routine pops old PC and old SR and returns control to the level 5 interrupt service routine at the right place and continues with the level 5 service routine.

External logic can respond to the interrupt acknowledge in one of the following ways: by requesting automatic vectoring or by placing a vector number on the data bus (nonautovector), or by indicating that no device is responding (Sporadic Interrupt). If the hardware asserts VPA to terminate the IACK bus cycle, the 68000 directs itself automatically to the proper interrupt vector corresponding to the current interrupt level. No external hardware is required for providing interrupt address vector. This is known as autovectoring. The vectors for the seven autovector levels are given below:

<table>
<thead>
<tr>
<th></th>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
<th>Level 4</th>
<th>Level 5</th>
<th>Level 6</th>
<th>Level 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>IACK</td>
<td>vector</td>
<td>IACK</td>
<td>vector</td>
<td>IACK</td>
<td>vector</td>
<td>IACK</td>
</tr>
<tr>
<td>11</td>
<td>vector</td>
<td>IACK</td>
<td>vector</td>
<td>IACK</td>
<td>vector</td>
<td>IACK</td>
<td>vector</td>
</tr>
<tr>
<td>10</td>
<td>vector</td>
<td>vector</td>
<td>vector</td>
<td>vector</td>
<td>vector</td>
<td>vector</td>
<td>vector</td>
</tr>
</tbody>
</table>

During autovectoring, the 68000 asserts VMA after assertion of VPA and then completes a normal 68000 read cycle as shown in Figure 5.26. The interrupting device uses external hardware to place a vector number on data lines D0-D7, and then performs a DIACK handshake to terminate the IACK bus cycle. The vector numbers allowed are 040 to SFF, but Motorola has implemented a protection on the first 64 entries so that user-interrupt vectors may overlap at the discretion of the system designer. This is known as nonautovectoring. The 68000 multiplies this vector by 4 and determines the pointers to interrupt address vector.

During autovectoring, the 68000 always checks the VPA line for LOW, and if VPA is asserted, the 68000 obtains the interrupt vector address using autovectoring. If VPA is not asserted, the 68000 checks DIACK for LOW. If DIACK is asserted, the 68000 obtains the interrupt address vector using autovectoring.

Another way to terminate an interrupt acknowledge bus cycle is with
the BErr (Bus Error) signal. Even though the interrupt control pins are synchronized to enhance noise immunity, it is possible that external system interrupt circuitry may initiate an IACK bus cycle as a result of noise. Since no device is requesting interrupt service, neither DTACK nor VPA will be asserted to signal the end of the nonexistent IACK bus cycle. When there is no response to an IACK bus cycle after a specified period of time (monitored by the user by an external timer), the BErr can be asserted by an external timer. This indicates to the processor that it has recognized a spurious interrupt. The 68000 provides 18H as the vector to fetch for the starting address of this exception handling routine.

The 68000 determines the interrupt address vector for each of the above cases as follows. After obtaining the 8-bit vector n, the 68000 reads the long word located at memory 4^n. This long word is the address of the service routine. Therefore, the address is found using indirect addressing. Note that the spurious interrupt and bus error interrupt due to troubled instruction (when no DTACK is received by the 68000) have different vectors. Spurious interrupt occurs when the BUS ERROR pin is asserted during interrupt processing.

Figure 5.27a shows a flowchart for the interrupt acknowledge sequence, a timing diagram is given in Figure 4.27b, and the interrupt processing sequence is shown in Figure 5.28. In Figure 5.27b, during IACK cycle, FC2 FC1 FC0 = 111. A1-A3 has the interrupt level. The vector number is provided on the D0-D7 pins by external hardware. Note that during nonautovectoring, if VPA goes to LOW, the 68000 ignores it.

**PROCESSOR**

- Grant the interrupt
- Compare interrupt level in Status Register and Wake for Current Instructions to Compute
- Place Interrupt level in A1, A2, A3
- Set Function Code to interrupt
- Acknowledge
- Assert Address Strobe (AS)
- Assert Data Strobe (DS* and DS)

**INTERRUPTING DEVICE**

- Request the Interrupt
- Provide the Vector Number
  - Place Vector Number on DS0-63
  - Assert Data Transfer Acknowledge (DTACK)
- Acquire the Vector Number
  - Load Vector Number
  - Negate DS and DS*
  - Negate AS
- Start Interrupt Processing

**FIGURE 5.27a** Vector acquisition flow chart (nonautovectoring).

**FIGURE 5.27b** Interrupt acknowledge cycle timing diagram (nonautovectoring).
5.13.2 INTERNAL INTERRUPTS

The internal interrupt is a software interrupt. This interrupt is generated when the 68000 executes a software interrupt instruction called TRAP or by some undesirable events such as division by zero or execution of an illegal instruction.

5.13.3 68000 EXCEPTION MAP

Figure 5.29 shows an interrupt map of the 68000. Vector addresses $000 through $2C include vector addresses for reset, bus error, trace, divide by 0, etc., and addresses $30 through $4C are unassigned. The RESET vector requires four words (addresses 0, 2, 4, and 6) and other vectors require only two words. As an example of how the 68000 determines the interrupt address vector, consider autovector 1. After VFA is asserted, if ID 1 10 = 001, the 68000 automatically obtains the 8-bit vector 25,149, and multiplies 19,4 by 4 to obtain the 24-bit address 0000064,149. The 68000 then loads the 16-bit contents of location 0000064,149 and the next location 0000065,149 into PC. For example, if the user wants to write the service routine for autovector 1 at address 271452,149, then XX27,149 and 1452,149 must, respectively, be stored at 0000064,149 and 0000065,149. Note that XX in XX27 are two don't care nibles.

After hardware reset, the 68000 loads the supervisor SP high and low words, respectively, from addresses 000000,0 and 000002,0, and the PC high and low words, respectively, from 000004,0 and 000006,0. Typical assembler directive, Define Constant (DC) can be used to load PC and SSP. For example, the following instruction sequence loads SSP with $004100 and PC with $001000:

```
ORG $000000
DC.L $00004100
DC.L $00001000
```

5.13.4 68000 INTERRUPT ADDRESS VECTOR

Suppose that the user decides to write a service routine starting at address $123456 using autovector 1. Since the autovector 1 uses addresses $0000064 and $0000066, the numbers $0012 and $3456 must be stored in locations $0000064 and $0000066, respectively. The DC.L assembler directive can be used to load $123456 into location $0000064 as follows:

```
ORG $0000064
DC.L $00123456
```
5.13.5 AN EXAMPLE OF AUTOVECTOR AND NONAUTOVECTOR INTERRUPTS

As an example to illustrate the concept of autovector and nonautovector interrupts, consider Figure 5.30. In this figure, I/O device 1 uses nonautovector and I/O device 2 uses autovector interrupts. The system is capable of handling interrupts from eight devices, since an 8-to-3 priority encoder such as the 74LS148 is used. Suppose that I/O device 2 drives the I/O line LOW in order to activate line 3 of this encoder. This, in turn, interrupts the processor. When the 68000 decides to acknowledge the interrupt, it drives FC0-FC2 HIGH. The interrupt level is reflected on A1-
acknowledgment of the interrupt, the 68000 will service the interrupt as level 3 autovector interrupt. Note that the encoder in Figure 5.31 is used for illustrative purposes. This encoder is not required for a single device as the A/D converter in the above example.

Figure 5.32 shows interfacing of a typical A/D converter to the 68000-based microcomputer using the nonautovector interrupt.

In the figure, the 68000 starts the A/D converter as before. Also, the DV signal is used to interrupt the microcomputer using line 5 (IPL2, IPL1, IPL0 = 101 which is level 2 interrupt) of the encoder. The DV can be used to assert DTACK so that after acknowledgment of the interrupt, FC2, FC1, FC0 becomes 111, which can be ANDed and inverted to enable an octal buffer such as the 74LS244 in order to transfer an 8-bit vector from the input of the buffer to the D0-D7 lines of the 68000. The 68000 can then multiply this vector by 4 to determine the interrupt address vector. As before, the encoder in Figure 5.32 is not required for the single A/D converter.

5.14 68000 DMA

Three DMA control lines are provided with the 68000. These are BR (Bus Request), BG (Bus Grant), and BGACK (Bus Grant Acknowledge).

The BR line is an input to the 68000. The external device activates this line to tell the 68000 to release the system bus. At least one clock period after receiving the BR, the 68000 will enable its BG output line to acknowledge the DMA request. However, the 68000 will not relinquish the bus until it has completed the current instruction cycle. The external device must check AS (Address Strobe) line to determine the completion of instruction cycle by the 68000. When AS becomes HIGH, the 68000 will tristate its address and data lines and will give up the bus to the external device.

After taking the bus, the external device must enable BGACK line. The BGACK line tells the 68000 and other devices connected to the bus that the bus is being used. The 68000 stays in a tristate condition until BGACK becomes HIGH.
5.15 68000 EXCEPTION HANDLING

A 16-bit microcomputer is usually capable of handling unusual or exceptional conditions. These conditions include situations such as execution of illegal instruction or division by zero. In this section, exception handling capabilities of a typical microprocessor such as the MC68000 are described.

The 68000 exceptions can be divided into three groups, namely, groups 0, 1, and 2. Group 0 has the highest priority and group 2 has the lowest priority. Within the first two groups, there are additional priority levels. A list of 68000 exceptions along with individual priorities is shown below:

Group 0: Reset (highest level in this group), Address Error (next level), and Bus Error (lowest level)
Group 1: Trace (highest level), Interrupt (next level), Illegal op code (next level), and Privilege Violation (lowest level)
Group 2: TRAP, TRAPV, CHK, and ZERO DIVIDE (no individual priorities assigned in group 2)

Exceptions from group 0 always override an active exception from group 1 or group 2. Group 0 exception processing begins at the completion of the current bus cycle (two clock cycles). Note that the number of cycles required for a READ or WRITE operation is called a bus cycle. This means that during an instruction fetch if there is a group 0 interrupt, the 68000 will complete the instruction fetch and then service the interrupt.

Group 1 exception processing begins at the completion of the current instruction.

Group 2 exceptions are initiated through execution of an instruction. Therefore, there are no individual priority levels within group 2. Exception processing occurs when a group 2 interrupt is encountered, provided there are no group 0 or group 1 interrupts.

When an exception occurs, the 68000 saves the contents of the program counter and status register onto the stack and then executes a new program whose address is provided by the exception vector. Once this program is executed, the 68000 returns to the main program using the stored values of program counter and status register.

Exceptions can be of two types: internal or external.

The internal exceptions are generated by situations such as division by zero, execution of illegal or unimplemented instructions, and address errors.

As mentioned before, internal interrupts are called traps.

The external exceptions are generated by bus error, reset, or interrupts.

The basic concepts associated with interrupts, relating them to the 68000, have already been described. In this section we will discuss the other exceptions.

In response to an exceptional condition, the processor executes a user-written program. In some microcomputers, one common program is provided for all exceptions. The beginning section of the program determines the cause of the exception and then branches to the appropriate routine. The 68000 utilizes a more general approach. Each exception can be handled by a separate program.

As mentioned before, the 68000 has two modes of operation: user state and supervisor state. The operating system runs in supervisor mode and all other programs are executed in user mode. The supervisor state is, therefore, privileged. Several privileged instructions such as MOVE to SR can only be executed in supervisor mode. Any attempt to execute them in user mode causes a trap.

We will now discuss how the 68000 handles exceptions which are caused by external reset, instructions causing traps, bus and address errors, tracing, execution of privileged instructions in user mode, and execution of illegal/unimplemented instructions.

The reset exception is generated internally. In response to this exception, the 68000 automatically loads the initial starting address into the processor.

The 68000 has a TRAP instruction which always causes an exception. The operand for this instruction varies from 0 to 15. This means that there are 16 TRAP instructions. Each TRAP instruction is normally used to call subroutines in an operating system. Note that this automatically places the 68000 in supervisor state. TRAP can also be used for inserting breakpoints in a program. Two other 68000 instructions cause traps if a particular condition is true. These are TRAPV and CHK. TRAPV generates an exception if the overflow flag is set. The TRAPV instruction can be inserted after every arithmetic operation in a program for causing a trap whenever there is the possibility of an overflow. A routine can be written at the vector address for the TRAPV to indicate to the user that an overflow has occurred. The CHK instruction is designed to ensure that access to an array in memory is within the range specified by the user. If there is a violation of this range, the 68000 generates an exception.

A bus error occurs when the 68000 tries to access an address which does not belong to the device connected to the bus. This error can be detected by asserting the BERR pin on the 68000 chip by an external device when no DTACK is received from the device after a certain period of time.
5.16 Multiprocessing with 68000 Using the TAS Instruction and AS (Address Strobe) Signal

Earlier, the 68000 TAS instruction was discussed. The TAS instruction supports the software aspects of interfacing two or more 68000s via shared RAM. When TAS is executed, an indivisible read-modify-write cycle is performed. The timing diagram for this specialized cycle is shown in Figure 5.33. During both the read and the write portions of the cycle, the AS remains LOW, and the cycle starts as a normal read cycle.

However, in the normal read the AS going inactive indicates the end of the read. During execution of the TAS, the AS stays LOW throughout the cycle, and therefore AS can be used in the design as a bus locking circuit.

Due to bus locking, only one processor at a time can perform a TAS operation in a multiprocessor system. The TAS instruction supports semaphore operations (globally shared resources) by checking a resource for availability and reserving or locking it for use by a single processor. The TAS instruction can, therefore, be used to allocate memory space reservations.

The TAS instruction execution flow for allocating memory is shown in Figure 5.34a and b. The shared RAM of Figure 5.34b is divided into M sections. The first byte of each section will be pointed to by (EA) of TAS.

---

**Figure 5.33** MC68000 read-modify-write cycle for TAS.

**Figure 5.34** Memory allocation using TAS.
(EA) instruction. In the flow chart, (EA) first points to the first byte of section 1. The instruction TAS (ea) is then executed.

The TAS instruction checks the most significant bit (N bit) in (EA). N = 0 indicates that the section 1 is free; N = 1 means section 1 is busy. If N = 0, then section 1 will be allocated for use. On the other hand, if N = 1, section 1 is busy; a program will be written to subtract one section length from (EA) to check the next section for availability. Also, (EA) must be checked with the value TASLOCM. If (EA) < TASLOCM, then no space is available for allocation. However, (EA) > TASLOCM, TAS is executed and the availability of that section is determined.

In a multiprocessor environment, the TAS instruction provides software support for interfacing two or more 68000 via shared RAM. The X5 signal can be used to provide the bus-locking mechanism.

Figures 5.35a and 5.35b show an MC68000 multiprocessor system with shared RAM using the TAS instruction.

In Figure 5.35a, the system is composed of two 68000 minimum systems: a shared memory subsystem and buffer control logic and an arbiter.

5.16.1 68000 MINIMUM SUBSYSTEM
The subsystem utilizes minimum 68000 configuration with ROM and RAM.

5.16.2 SHARED MEMORY AND BUFFER CONTROL LOGIC
The system takes arbiter output Q, Q and subdivides bus transceivers on-line and off-line as dictated by the arbiter. Note that 74LS244 and 74LS245 are, respectively, unidirectional and bidirectional buffer. This logic, after some delay, also asserts DTACK to the microprocessors after buffers have been switched on line, and data are valid. This is accomplished by digital delay lines. This is necessary because a processor will "throw away" its bus cycle while waiting for access to shared memory. Also, when the arbiter terminates a bus cycle, the R/W line is actively pulled HIGH several time constants before the tri-state bus changes. This insures that spurious writes will not occur at random locations due to the tri-state condition. The circuit is accomplished as shown on the opposite page.

5.16.3 ARBITER
In the system of Figure 5.35a, DTACK must be given to a processor whenever a processor is accessing local memory of when it is the only processor accessing shared RAM. When both processors request shared RAM, then only one processor should receive it. In this system, the
processors are given equal priority to access the shared RAM. The bus arbiter performs all the control functions.

Figure 5.15b shows the details of the bus arbiter. Shared RAM will be accessed when address line A23 (A23A or A23B) is HIGH. For each processor, when A23 is LOW and when A5 is asserted, shared RAM DTACK will be given immediately (gates USA1 and USA2 for processor A, gates USB1 and USB2 for processor B).

A crucial criterion for the shared RAM system is that when a processor is given access to the shared bus, it must be allowed to complete the bus cycle without being interrupted. Thus, when a bus cycle is initiated by A5 being asserted, the end of the cycle must be detected by A5 being negated.

In addition to detecting the completion of the bus cycle, another processor must be inhibited from changing the state of the arbiter. This is accomplished by NOR gates U1A and U1B which act as the bus arbiter. These gates act as follows: when both processors are not accessing shared memory, both inputs are HIGH and the arbiter assumes some state. The inverter at the processor's CLK input ensures that only one processor has access to the bus if both processors access it at the same time. The arbiter will remain in a locked condition until A5 is negated. When this occurs and if the second processor requests access, then the arbiter will change state. If it does not request access then the arbiter will remain idle until either processor requests shared RAM access. After completion of a bus cycle, the bus master negates A5 and the bus arbiter flips state. Finally, the new output of the arbiter is propagated to the delay lines and the bus control
may be offered to the next processor. After some delay time, DTACK is
given if the other processor is requesting shared RAM.

In Figure 5.35a the buffers 74LS244 (1) or 74LS244 (3), respectively,
transfer All-A15 lines of processor A or B upon enabling of the OE by the
bus arbiter. Since 74LS244 is an 8-bit unidirectional buffer, two 74LS244s
are required for 16-bit transfer. The buffers 74LS244 (2) and 74LS244 (4)
are used to transfer R/W, UDS, LDS lines of processor A or B to the
shared memory upon enabling of OE by DTACKA or DTACKB. The
buffers 74LS245 (5) and 74LS245 (6) are used for bidirectional transfer
of D0-D15 (from processor to RAM and vice versa).

For read or write operation consider Figure 5.35c. Assume that pro-
cessor A has access to the shared RAM. This means that DTACKA is LOW.
During read cycle, processor A R/W = 1 will make DIR (direction) input
of the 74LS245 (5) HIGH. This will transfer data from the shared RAM
to processor A via 74LS245 (5). On the other hand, during write cycle, R/
W will make DIR (direction) input of 74LS245 (5) LOW. This will transfer
data from processor A to shared RAM via 74LS245 (5). Note that since
74LS245 is an 8-bit bidirectional buffer, two 74LS245s are required for
16-bit transfer.

Similarly, when processor B has access to the shared RAM, buffer
74LS245 (6) is used to transfer data between the shared RAM and pro-
cessor B.

5.16.4 DIGITAL DELAY LINES

The digital delay lines can delay the propagation of a digital signal by
a known amount. Furthermore, by the use of taps, the digital signal can be
"picketed" at various delay times. This can be very useful in "pre-event"
signal switching. Examine the following theoretical case, where we wish to
route several signals down some path that is several levels deep, and a
control line falls LOW when we are to actually accomplish the task:

![DELAY LINE Diagram]

There are several major manufacturers of digital delay lines. Listed
below are some major manufacturers of delay lines.

Data Delay Devices
383 Lakeview Ave.
Clifton, NJ 07011
Stock number of 120-nS device: DDU-66U-120

Engineering Components Company
3580 Sacramento Drive
San Luis Obispo, CA 93401
Stock number of 120-nS device: TTLDM-120
The 68000 Control Unit is designed using nanomemories. This method provides significant savings in memory when a group of microoperations occur several times in a microprogram. Consider the microprogram of Figure 5.36. The microprogram in this figure contains a microinstruction B bits wide.

![Microprogram](image)

The size of the control memory to store this microprogram is AB bits. Assume that the microprogram has n (n ≤ A) unique microinstructions. These n microinstructions can be held in a separate memory called the nanomemory of size nB bits. Each of these n instructions occurs once in the nanomemory. Each microinstruction in the original microprogram is replaced with the address that specifies the location of the nanomemory in which the original B-bit-wide microinstructions are held. Since the nanomemory has n addresses, only the upper integer of log₂ n bits is required to specify a nanomemory address. This is illustrated in Figure 5.37.

The operation of microprocessor employing a nanomemory can be explained as follows. The microprocessor's control unit reads an address from the microprogram. The contents at this address in the nanomemory is the desired control word. The bits in the control word are used by the control unit to accomplish the desired operation. Note that a control unit employing nanomemories (two-level memory) is slower than the one using a conventional control memory (single memory). This is because the nanomemories require two memory reads (one for the control memory and the other for the nanomemory). For a single conventional control memory, only one memory fetch is necessary. This reduction in control unit speed is offset by the cost of the memory when the same microinstructions occur many times in the microprogram.

Consider a 7 x 4-bit microprogram stored in the single control memory of Figure 5.38. This simplified example is chosen to illustrate the nanomemories concept even though this is not a practical example.

![7 x 4-bit single control memory](image)

In this program, 3 out of 7 microinstructions are unique. Therefore, the size of the microcontrol store is 7 x 2 bits and the size of the nanomemory is 3 x 4 bits. This is shown in Figure 5.39.

Memory requirements for the single control memory = 7 x 4 = 28 bits.
Memory requirements for nanomemories = (7 x 2 + 3 x 4) bits = 26 bits.
Therefore, saving using nanomemories = 28 - 26 = 2 bits. For a
 example like this, 2 bits are saved.

The 68000 control unit nanomemory includes a 640 x 9-bit microcontrol store and a 280 x 70 nanocontrol store as shown in Figure 5.40.

![FIGURE 5.39 Two-level store (nanomemory).](image)

In Figure 5.40, out of 640 microinstructions, 280 are unique. If the 68000 were implemented using a single control memory, memory requirements would have been 640 x 70 bits. Therefore, memory savings are:

- 640 x 70 - (640 x 9 + 280 x 70) bits
- 44,800 - 25,360
- 19,440 bits

This is a tremendous memory saving for the 68000 control unit.

5.18 BASIC DIFFERENCES BETWEEN THE 68000, 68008, 68010, AND 68012

The 68000 has the same register set as the 68000. The 68010 and 68012, on the other hand, include all 68000 registers plus a 32-bit vector base register (VBR) and two 3-bit alternate function code registers, SFC (source function code) and DFC (destination function code).

The 68008, like the 68000, provides an exception vector table starting at a fixed address 0. The 68010 and 68012 compute the address of the exception table by multiplying the 8-bit vector number by 4 and then adding it to VBR. This provides the user with the flexibility to provide exception vector address anywhere in memory.

The 68000 family of processors provides three function code output lines FC2, FC1, and FC0. These pins indicate to any peripherals the type of access currently in progress by the processors. The SFC and DFC registers on the 68010 and 68012 permit a system-level program to provide its own function code outputs for the source or destination during the execution of the MOVES instruction (to be discussed in Chapter 6).

The 68000, 68008, 68010, and 68012 include the same instruction set. The processors use a two-word instruction fetch. The processors begin by fetching the instruction word. As decoding starts, the processors fetch the following word from memory so that when they start executing the instruction, two words are already available. This speeds up instruction execution. However, if the processors encounter a branch instruction, the word following the instruction is discarded.

The 68010 and 68012 provide an enhancement over the two-word pipelining scheme just mentioned. In some cases, the processors can execute the looped instruction without refetching the instruction from memory. In the special case where the second word of the two prefetched words branches back to that single word instruction, the processors need not do any instruction fetches from memory while inside the loop. The pins and signals associated with the 68000, 68008, 68010, and 68012 are shown in Figure 5.41.

The 68008 has an 8-bit-wide data bus and this is the only processor in the family which has the least significant address line A0. The 68008 is also the only processor in the family which includes the single data strobe DS signal. The 68008 places a LOW on DS to indicate that its data bus is in...
use. The 68000/684010 has A1-A23 as the address lines, while the 68012 has A1-A29 and A31 with A0 on these processors encoded from UDS and LDS in the same way as the 68000.

AS, R/W, DTACK, HALT, RESET, and CLK pins on the 68008, 68010, and 68012 have similar functions as the 68000.

The function code outputs (FC2, FC1, and FC0) on the 68008, 68010, and 68012, like those on the 68000, are valid when AS is low and identify the type of bus activity (user vs. supervisor) with the same encodings as the 68000 except for the 68010 and 68012 when FC2, FC1, and FC0 are all high. For the 68010 and 68012, the machine cycle with FC2, FC1, FC0 = 111 is called the CPU space cycle. In this cycle, the processors 68010 and 68012 can perform certain functions such as breakpoints. Note that in the 68000, the function code outputs FC2, FC1, FC0 = 111 indicate the interrupt acknowledge cycle. BERR input signal provides slightly different results on the virtual memory processors 68010 and 68012. In these processors, the total addressing space of the task may be greater than the physical memory space. The operating system may choose to store certain programs on disk while executing other programs. Since the processors only execute programs in physical memory, the operating system must be able to read the programs from disk into physical memory when they need to be executed. Therefore, to implement virtual memory effectively, the processors must be able to recognize the absence of required memory, read the data from disk, and continue execution of the program. External logic should recognize the nonexistence of the required memory and assert BERR. The 68010 and 68012 save a detailed stack frame in initiating bus error exception. As part of the exception routine, the 68010 and 68012 may analyze the stored information and determine if the error was caused because of a request for data stored on disk. If so, it can read data from disk into memory.

The bus arbitration signals BR, BG, and BGAck are available on the 68000, 68010, and 68012. The 68008 implements fewer handshaking signals and does not include the BGAck signal. Also, the 68008 does not implement all the 6800-type signals. For example, the 68008 does not implement the VMA signal.

The 68000 read and write timing diagrams are similar to those of 68010 and 68012. The 68008 read and write timing diagrams are a bit different from those of the other processors, since the 68008 has DS (data strobe) pin instead of UDS and LDS pins with the 68000/68010/68012, 8 data pins D0-D7, and 20 address pins A0-A19. The details of 68008 timing can be found in Motorola manuals.

The read/modify/write cycles of all the 68000 family of processors are identical except that the 68012 provides an additional pin called the RMC (read/modify/write cycle). The pin is asserted by the 68012 throughout the entire read/modify/write cycle. The RMC pin can be used by memory management schemes that require advanced indication of read/modify/write cycle. The 68020 and 68030 include the RMC pin.

External interrupts on all processors (except for the 68008) are handled via the IP2, INT1, and IFLO pin in the same way. The 68008 combines IFLO and IF2 into one signal (IFLO/2) and provides four interrupt levels.

**QUESTIONS AND PROBLEMS**

5.1 Assume that D0 = 2577441. What will be the contents of D0 after execution of each of the following instructions:

i) CLR B, D0
ii) CLR D0
iii) CLR L, D0

5.2 Determine the contents of registers and the locations affected by each of the following instructions:

i) MOVE L - (A2), (A3) +

Assume the following data prior to execution of the MOVE:

\[ [A2] = 33000504, \quad [A3] = 5510718, \]
\[ [3000500] = 01, \quad [3000501] = 0F1, \]
\[ [3000502] = 172, \quad [3000503] = 8A1, \]
\[ [5510718] = 253, \quad [5510719] = 20, \]
\[ [551071B] = 33, \quad [510718] = 27 \]

ii) MOVE W D1, A4

Assume the following data prior to execution of the MOVEA:

\[ [D1] = 371588470, \]
\[ [A4] = 581212834 \]

iii) MOVE L A2, A3

Assume the following data prior to execution of the MOVEA:

\[ [A2] = 1234567, \]
\[ [A3] = 20718714 \]
5.3 Identify the following 68000 instructions as privileged or nonprivileged:
   i) MOV E, (A2)
   ii) MOV CR, (A0)
   iii) LEA L, (A2), A5
   iv) MOV L A2, USP

5.4 What are the contents of register D1 after execution of the following two instructions? Assume [D1] = $34758480 prior to the execution of the instructions:
   EXT L D1
   MOVEQ L $52F, D1

5.5 Find the contents of D1 after execution of the following DIVS instructions:
   DIVS (A1), D1

5.6 Write a 68000 assembly program to divide an 8-bit signed number in low byte of D1 by an 8-bit signed number in low byte of D2. Store quotient and remainder in D1.

5.7 Write a 68000 assembly language program to add two 128-bit numbers. Assume that the first number is stored in consecutive memory locations starting at $65014. The second number is stored in consecutive memory locations starting at $76020. Store the result in memory locations beginning at $708020.

5.8 Write a 68000 assembly program to add two top two 32 bits of the stack. Store the 32-bit result onto the stack. Assume user mode.

5.9 Write a 68000 assembly program to multiply an 8-bit signed number in low byte of D1 by a 16-bit signed number in high word of D5 store the result in D3.

5.10 Write a 68000 assembly program to add twenty 32-bit numbers stored in consecutive memory locations starting at address $502000. Store the 32-bit result onto the stack. Assume that for each 32-bit number, the lowest address stores the highest byte of the number.

5.11 Write 68000 assembly language to find the minimum value of a string of ten signed 16-bit numbers using indexed addressing.

5.12 Write a 68000 assembly program to compare two strings of twenty ASCII characters. The first string is stored starting at $500000 followed by the second string is stored starting at $604000. The ASCII character in location $500000 of string 1 will be compared with the ASCII character in location $5004000 of string 2, [S5003001] to be compared with [S5003002], and so on. Each time there is a match, store SEEEE onto the stack, otherwise store S0000.

5.13 Write a 68000 assembly program to divide a 27-bit unsigned number in high 27 bits of D0 by 16. Do not use any divide instruction. Neglect remainder. Store quotient in low 27-bit of D0.

5.14 Write a subroutine in 68000 assembly language to compute

$$ z = \frac{\sum_{i=1}^{n} (x_i - y_i)}{10} $$

Assume that Xi's and Yi's are signed 16-bit and stored in consecutive locations starting at $020524 and $305116, respectively. Assume A0 and A1 point to Xi's and Yi's, respectively, and SP is already initialized.

5.15 Write a subroutine in 68000 assembly language to subtract two unsigned eight-digit BCD numbers. The BCD number 1 is stored at location starting from $300000 thru $300007, with the least significant digit at $300007 and the most significant digit at $300000. Similarly, the BCD number 2 is stored at location starting from $400000 through $400007, with the least significant digit at $400007 and the most significant digit at $400000. The BCD number 2 is to be subtracted from BCD number 1. Store result in D1.
5.16 Write a subroutine in 68000 assembly to convert a 3-digit unsigned BCD number to binary. The most significant digit is stored in memory location starting at $003000$, the next digit is stored at $003001$, and so on. Store the binary result in D3. Use the value of the 3-digit BCD number in $V = D2 \times 10^2 + D1 \times 10^1 + D0 = (D2 \times 10 + D1) \times 10 + D0$.

5.17 Using LINK and UNLK instructions, write recursive subroutine (A subroutine calling itself) in 68000 assembly language to find the factorial of an 8-bit number $n$ by using $n! = n(n-1)(n-2) \ldots 1$. Store the result in D0.

5.18 Determine the status of LDS, UDS, AS, FC2-FC0, and address lines immediately after execution of the following instruction sequence (before the 68000 transits these lines to fetch the next instruction):

\[
\text{MOVE} \ #20000, \ \text{SR} \\
\text{MOVT.B} \ D2, \ #030002 \\
\]

Assume the 68000 is in supervisor mode prior to execution of the above instructions.

5.19 Write a 68000 assembly program to output the contents of memory locations $003000$ and $003001$ to two seven-segment displays connected to two 8-bit ports A and B.

5.20 Assume that in the configuration of Figure 5.23, port A and port B each has three switches and an LED connected to bits 0 through 3. Write 68000 assembly program to

i) Turn the port A LED ON and port B LED OFF if port A has an even number of high switch inputs.

ii) Turn the port A LED OFF and port B LED ON if port A has an odd number of high switch inputs.

iii) Turn both LEDs ON if both ports A and B have an even number of high switch inputs.

iv) Turn both LEDs OFF if both ports A and B have odd number of high switch inputs.

5.21 Interface a 68000 to 2716s, 6116s, and a 68330 to provide 4K EPROM, 4K RAM, and two 8-bit I/O ports. Draw a neat schematic and determine memory and I/O maps. Assume 16.67 MHz internal clock for the 68000.

5.22 If the IPL3, IPL2, IPL1 pins are interrupted by an external device with the code 001, when the interrupt mask value 12110 is $3_{16}$, will the interrupt be serviced immediately or ignored by the 68000?

5.23 Discuss briefly the various 68000 exceptions.

5.24 Write a service routine for reset that will initialize all data and address registers to zero, supervisor SP to $3F0728$, user SP to $3F0524$, and then jump to $300200$.

5.25 Assume the following stack and register values before occurrence of an interrupt:

\begin{center}
\begin{tabular}{c|c}
\hline
Stack & R12606 \\
& R12652 \\
& R12664 \\
& R12668 \\
& SPF13480 \hline
\end{tabular}
\end{center}

If an external device requests an interrupt by asserting the IPL3, IPL2, IPL1 pins with the value 000, determine the contents of SPF and SR during interrupt and after execution of RTE at the end of the service routine of the interrupt. Draw the memory layouts showing where SPF points to and the stack contents during and after interrupt. Assume that the stack is not used by the service routine.

5.26 Suppose that two pumps (P1, P2) and two LEDs (L1, L2) are to be connected to a 68000-based microcomputer. Each pump has a 'pump running' output to indicate the ON/OFF status. The microcomputer starts the pumps via bits 2 and 3 of 8-bit port A and tries to turn the pumps ON. Two LEDs L1 (for P1) and L2 (for P2) are connected to bits 9 and 1 of 8-bit port B to indicate the pump statuses. If P1 runs, turn L1 ON.
On the other hand, if P2 runs, turn L2 ON. If both pumps run, turn both LEDs ON; otherwise, turn one LED (L1 or L2 depending on the pump) or both LEDs OFF if one or both pumps do not run. Assume that the pump can be turned ON by HIGH and turned OFF by LOW.

i) Using programmed I/O, draw a block diagram and write a 68000 assembly program to accomplish the above.

ii) Using interrupt I/O, draw a block diagram. Write the main program and service routine in 68000 assembly language to accomplish the above. The main program will perform all initializations and then start the pumps.

5.27 Compare the basic features of the 68000 with those of 68008, 68010, and 68012.

5.28 Write a 68000 assembly language program to add a 32-bit number stored in D0 (bits 0 through 15 containing the high-order 16 bits of the number and bits 16 through 31 containing the low-order 16 bits) with another 32-bit number stored in D1 (bits 0 through 15 containing the low-order 16 bits of the number and bits 16 through 31 containing the high-order 16 bits). Store the result in D0.

5.29 Write a subroutine in 68000 assembly language using the TAS interaction to find, reserve, and lock a memory segment for the main program. The memory is divided into four segments (0, 1, 2, 3) of 8 bytes each. The first byte of each segment includes a flagbyte to be used by the TAS instruction. In the subroutine, a maximum of four 8-byte memory segments must be checked for a free segment. Once a free segment is found, the TAS instruction is used to set the flagbyte. The starting address of the free segment must be stored in A5 and D5 and must be cleared to zero to indicate a free segment. If no free block is found, a nonzero value must be stored in D5.

5.30 Write a 68000 assembly language program to delete item 0 from the following linked list:

<table>
<thead>
<tr>
<th>ITEM 0</th>
<th>ITEM 1</th>
<th>ITEM 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000H</td>
<td>5000H</td>
<td>5000H</td>
</tr>
<tr>
<td>5000H</td>
<td>5000H</td>
<td>5000H</td>
</tr>
<tr>
<td>HEAD</td>
<td>ITEM</td>
<td>ITEM</td>
</tr>
</tbody>
</table>

Chapter 6

MOTOROLA MC68020

This chapter describes in detail the hardware, software, and interfacing features associated with the MC68020.

Topics include MC68020 architecture, addressing modes, instruction set, I/O, coprocessor, and system design.

6.1 INTRODUCTION

The MC68020 is a 32-bit microprocessor. The MC68020 is designed to execute all user object code written for previous members of the MC 68000 family.

The MC68020 is manufactured using HCMOS (combining E-MOS and CMOS on the same device). The MC68020 consumes a maximum of 1.75 watts. It contains 200,000 transistors on a 3/4" piece of silicon. The chip is packaged in a square (1.345" x 1.345") pin grid array (PGA) and contains 169 pins (114 pins used) arranged in a 13 x 13 matrix.

The processor speed of the MC68020 can be 12.5 MHz, 16.67 MHz, 20 MHz, 25 MHz, or 33 MHz. The chip must be operated from a minimum frequency of 8 MHz. Like the MC68000, it does not have any on-chip clock generation circuitry. The MC68020 contains 18 addressing modes and 101 instructions. All addressing modes and instructions of the MC68000 are included in the MC68020. The MC68020 supports coprocessors such as the MC68881/MC68882 floating-point and MC68881 memory management unit (MMU) coprocessors.

The features of MC68020 are compared with those of MC68000 in the following:
### Chapter 6

#### 6.1 Introduction

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>68000</th>
<th>68020</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>NMOS</td>
<td>HCMOS</td>
</tr>
<tr>
<td>Size</td>
<td>3-1/2&quot; x 1-1/4&quot;</td>
<td>1.345&quot; x 1.345&quot;</td>
</tr>
<tr>
<td>Number of pins</td>
<td>64, 66</td>
<td>169 (13 x 13 matrix; pins come out at the bottom of the chip; 114 pins currently used)</td>
</tr>
<tr>
<td>Control unit</td>
<td>Nanomemory (two-level control memory)</td>
<td>Nanomemory (two-level control memory)</td>
</tr>
<tr>
<td>Clock</td>
<td>6 MHz, 8 MHz, 10 MHz, 12.5 MHz, 16.67 MHz (no minimum requirements)</td>
<td>12.5 MHz, 16.67 MHz, 20 MHz, 25 MHz, 33 MHz (must be 8 MHz minimum)</td>
</tr>
<tr>
<td>ALU</td>
<td>One 16-bit ALU</td>
<td>Three 32-bit ALUs</td>
</tr>
<tr>
<td>Address bus size</td>
<td>24 bits with A0 encoded from LDS and LDSs</td>
<td>32 bits; no encoding of A0 required</td>
</tr>
<tr>
<td>Data bus size</td>
<td>Uses D0-D7 for odd addresses and D8-D15 for even addresses during byte transfers; for word and long word use D0-D15</td>
<td></td>
</tr>
<tr>
<td>Instruction and data access</td>
<td>All word and long word accesses must be at even addresses for both instructions and data; for byte, instruction must be at even addresses, and data can be at either odd or even addresses</td>
<td>Instructions must be accessed at even addresses; data accesses can be at any address for byte, word, and long word</td>
</tr>
<tr>
<td>Instruction cache</td>
<td>128-entry 16-bit word cache; at the start of an instruction fetch, the 68020 always outputs LOW on the FC5 (external cycle start pin and accesses the cache); if the instruction is found in the cache, the 68020 inhibits outputting LOW on the X3 pin; otherwise the 68020 sends LOW on the X3 pin and reads the instruction from the main memory</td>
<td>4-gigabyte (4, 294, 964, 296 bytes)</td>
</tr>
<tr>
<td>Directly addressable memory</td>
<td>16 megabytes</td>
<td>16-bit SR</td>
</tr>
<tr>
<td>Registers</td>
<td>8 32-bit data registers</td>
<td>8 32-bit data registers</td>
</tr>
<tr>
<td></td>
<td>7 32-bit address registers</td>
<td>7 32-bit address registers</td>
</tr>
<tr>
<td></td>
<td>2 32-bit SPs</td>
<td>3 32-bit SPs</td>
</tr>
<tr>
<td></td>
<td>1 32-bit PC (all bits used)</td>
<td>1 16-bit SR</td>
</tr>
<tr>
<td>Addressing modes</td>
<td>14</td>
<td>16-bit SR</td>
</tr>
<tr>
<td>Stack pointers</td>
<td>56 instructions</td>
<td>1 2-bit VBR (vector base register)</td>
</tr>
<tr>
<td>Stack pointer</td>
<td>USP, SSU</td>
<td>2 3-bit function code registers (ISFC and ISFC)</td>
</tr>
<tr>
<td>Status register</td>
<td>T, S, T0, I2, X, N, Z, V, C</td>
<td>132-bit CAAR (cache address register)</td>
</tr>
<tr>
<td></td>
<td>T, T0, S, M, I0, I1, I2, X, N, Z, V, C</td>
<td>32-bit CACR (cache control register)</td>
</tr>
<tr>
<td></td>
<td>T0, T1, S, M, I0, I1, I2, X, N, Z, V, C</td>
<td>18</td>
</tr>
<tr>
<td>Coprocessor interface</td>
<td>Emulated in software; that is, by writing subroutines, coprocessor functions such as floating-point arithmetic can be obtained</td>
<td>Can directly be interfaced to coprocessor chips and coprocessor functions, such as floating-point arithmetic can be obtained via 68020 instructions</td>
</tr>
<tr>
<td>FC2, FC0, FC1 pins</td>
<td>FC2, FC0, FC1 = 111 means interrupt acknowledge</td>
<td>FC2, FC0, FC1 = 111 means CPU space cycle and then by decoding A16-A19, one can obtain breakpoints, coprocessor functions, and interrupt acknowledge</td>
</tr>
</tbody>
</table>
Some of the 68020 characteristics tabulated above will now be explained:

- The three independent ALUs are provided for data manipulation and address calculations.
- A 32-bit barrel shift register (occupies 7% of silicon) is included in the 68020 for very fast shift operations regardless of the shift count.
- The 68020 has three SPSs. In the supervisor mode (when \( S = 1 \)), two SPSs can be accessed. These are MSP (when \( M = 1 \)) and ISP (when \( M = 0 \)). The ISP can be used to simplify and speed up task switching for operating systems.
- The vector base register (VBR) is used in interrupt vector computation. For example, in the 68000 the interrupt address vector is obtained by multiplying an 8-bit vector by 4. In the 68020, on the other hand, the interrupt address vector is obtained by using VBR + 4*8-bit vector.
- The SFC (source function code) and DFC (destination function code) registers are 3 bits wide. These registers allow the supervisor to move data between address spaces. In supervisor mode, 3-bit addresses can be written into SFC or DFC using instructions such as MOVEC, MOVES. The upper 29 bits of SFC are assumed to be zero. The MOVES.W(A0), D0 can then be used to move a word from a location within the address space specified by SFC and (A0) to D0. The 68020 outputs [SFC] to the FC2, FC1, and FC0 pins. By decoding these pins via external decoder, the desired source memory location addressed by (A0) can be moved to D0. Now, if this data in D0 is to be moved to another space, then the following instructions will accomplish this:

```
MOVES.W D0, (A5)
```

Note that there is no MOVES mem, mem instruction. SFC and DFC allow one to move data from one space to another. Since in the above, MOVES.W D0, (A5) outputs [DFC] to FC2, FC1, and FC0 pins which can be used to enable the chip containing the memory location addressed by (A5). [D0] is then moved to this location. The new addressing modes in the 68020 include scaled indexing, 32-bit displacements, and memory indirect. In order to illustrate the concept of scaling, consider moving the contents of memory location 500 to A1. Using the 68000, the following instruction sequence will accomplish this:

```
MOVES.W #10, A0  ; Load starting address of a table to A0
MOVES.W #10, D0  ; Load index value to D0
ASL #2, D0       ; Scale index
MOVES.L 0(A0, D0), A1  ; Access data
```

The scaled indexing can be used with the 68020 to perform the same as follows:

```
MOVES.W #10, A0  ; Load starting address of a table to D0
MOVES.W #10, D0  ; Load index value to D0
MOVES.L (0, A0, D0*W4), A1  ; Access data
```

Note that [D0] in the above is scaled by 4. Scaling 1, 2, 4, or 8 can be obtained.

- The new 68020 instructions include bit field instructions to better support compilers and certain hardware applications such as graphics, 32-bit multiply and divide instructions, pack and unpack instructions for 3CD, and coprocessor instructions. Bit field instructions can be used to input A/D converters and eliminate wasting main memory space when the A/D converter is not in-use.
- FC2, FC1, FC0 = 111 means CPU space cycle. The 68020 makes CPU space access for breakpoints, coprocessor operations, or interrupt acknowledge. The CPU space classification is generated by the 68020 based upon execution of breakpoint instructions, coprocessor instructions, or during interrupt acknowledge cycle. The 68020 then decodes A19-A16 to determine the type of CPU space. For example, FC2, FC1, FC0 = 111 and A19, A18, A17, A16 = 0010 mean coprocessor instruction.
- For performing floating-point operations, the 68000 user must write subroutines using the 68000 instruction set. The floating point capability in the 68020 can be obtained by connecting the floating-point coprocessor chip such as the Motorola 68881. The MC68020 currently has two coprocessor chips. These are the 68881 (floating-point) and 68881i (memory management). The 680020 can have up to eight coprocessor chips. When a coprocessor is connected to the 68020, the coprocessor instructions are added to the 68020 instruction set automatically, and this is transparent to the user. For example, when the 68881 floating-point coprocessor is added to the 68020,
instructions such as FADD (floating-point ADD) are available to the user. The programmer can then execute the instruction:

\[ \text{FADD FDO, FD1} \]

Note that registers FDO and FD1 are in the 68881. When the 68020 encounters the FADD instruction, it writes a command in the command register in the 68881, indicating that the 68881 has to perform this operation. The 68881 then responds to this by writing in the 68881 response register. Note that all coprocessor registers are memory-mapped. The 68020 thus can read the response register and obtain the result of the floating-point ADD from the appropriate location.

### 6.2 PERIPHERAL SUPPORT AND APPLICATIONS

The 68020 is supported by an array of peripheral devices that use standard microcomputer functions. These functions include MMUs, synchronous and asynchronous data communication, local area network (LAN) interface, local/global bus interface and arbitration, general-purpose I/O (serial and parallel), floppy and hard disk controller, and direct-memory access (DMA) devices.

The 68020 derives its power from its variety of features. However, the degree to which each feature boosts performance depends on the specific application. For example, suppose that the 68020 is being used as a communications processor. The instruction cache — which helps most in executing branches and loops — would be of limited benefit. A signal processor’s job of coordinating the flow of communications suggests a linear program flow with few loops. On the other hand, new 68020 features like scaled index, extended addressing modes, and bit-field manipulations are well-suited for communication processing.

The 68020’s bit-field instructions are valuable tools in graphics applications. The cyclical nature of a typical graphics processing algorithm would benefit from the chip’s instruction cache. Robotics and numeric control applications would also make good use of the cache memory. Operating systems control is supported by privileged instructions (user/supervisor split), memory management, the interrupt and trap structures, and other specific 68020 instructions. Multiprocessing benefits from on-chip bus arbitration logic for shared bus and memory systems.

A popular application of the MC68020 microprocessor has been as a general processor in UNIX-based supermicrocomputers and engineering computer-aided engineering design, and manufacturing (CAE/CAD/CAM) workstations that use multithread real-time software to produce complex graphics. Additional possible applications include: next-generation personal computers that can emulate mainframe operation, large-scale small business computing systems that feature UNIX business packages, and fault-tolerant multiprocessor/multitasking systems.

### 6.3 FUNCTIONAL BLOCK DESCRIPTION

Figure 6.1 shows a block diagram of the MC68020. The processor can be divided into two main sections — the bus controller and the microprocessor. These blocks operate independently so that the two blocks can operate concurrently, yet synchronize instruction execution and bus operation.

The bus controller loads instructions from the 68020 data bus into the instruction cache. The instruction pipe in the microprocessor section inputs these instructions. The instruction decoder, sequencer, microprogram/transfer, and the control section in the microprocessor perform the instruction decode and generate the control signals.

![Figure 6.1 MC68020 block diagram](image)
6.4 PROGRAMMER’S MODEL

The MC68020 programmer’s model is based on sequential, nonconcurrent instruction execution. This implies that each instruction is completely executed before the next instruction is executed. Although instructions might operate concurrently in actual hardware, they do not operate concurrently in the programmer’s model.

Figure 6.2 shows the MC68020 programming models. The user model has sixteen 32-bit general-purpose registers (D0-D7 and A0-A7), a 32-bit program counter (PC), and a condition code register (CCR) contained within the supervisor status register (SR). The supervisor model has two 32-bit supervisor stack pointers (ISP and MSP), a 16-bit status register (SR), a 32-bit vector base register (VBR), two 8-bit alternate function code registers (SFC and DPC), and two 32-bit cache handling (address and control) registers (CAAR and CACR). General-purpose registers D0-D7 are used as data registers for operation on all data types. General-purpose registers A0-A6, user stack pointer (USP) A7, interrupt stack pointer (ISP) A7", and master stack pointer (MSP) A7" are address registers that may be used as software stack pointers or base address registers.

The status register (Figure 6.3) consists of a user byte (condition code register CCR) and a system byte. The system byte contains control bits to indicate that the processor is in the trace mode (T1, T0), supervisor/user state (S), and master/interrupt state (M). The user byte consists of the following condition codes: carry (C), overflow (V), zero (Z), negative (N), and extend (X).

Table 6.1 shows the conditional tests available to the MC68020.

The bits in 68020 user byte are set to reset in the same way as 68000 user byte. The bits T2, T1, 10, and S have the same meaning as 68000. In the 68020, two trace bits (T1, T0) are included as opposed to one trace bit (t) in the 68000. These two bits allow the 68000 to trace on both normal instruction execution and jumps. The 68020 M-bit is not included in the 68000 status register.

The vector base register (VBR) is used to locate the exception processing vector table in memory. The VBR supports multiple vector tables so each process can properly manage independent exceptions.

The MC68020 distinguishes address spaces as supervisor/user program/data. To support full access privileges in the supervisor mode, the alternate function code registers (SFC and DPC) allow the supervisor to
access any address space by preloading the SF/DFC registers appropriately.

The cache registers (CACR and CAAR) allow software manipulation of the instruction cache. The CACR provides control and status accesses to the instruction cache, while the CAAR holds the address for those cache control functions that require an address.

6.5 DATA TYPES, ORGANIZATION, AND CPU SPACE CYCLE

The MC68000 family supports data types of bits, byte integers (8 bits), word integers (16 bits), long word integers (32 bits), and binary coded decimal (BCD) digits. In addition to these, four new data types are supported by the MC68020: variable-width bit field, packed BCD digits, quad words (64 bits), and variable-length operands.

Data stored in memory are organized on a byte-addressable basis, where the lower addresses correspond to higher-order bytes. The MC68020

---

**TABLE 6.1**

**MC68020 Conditional Tests**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Condition</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>T*</td>
<td>True</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>False</td>
<td>0</td>
</tr>
<tr>
<td>HI</td>
<td>High</td>
<td>C . Z</td>
</tr>
<tr>
<td>LS</td>
<td>Low or same</td>
<td>C + Z</td>
</tr>
<tr>
<td>CC(15)</td>
<td>Carry clear</td>
<td>C</td>
</tr>
<tr>
<td>CS(13)</td>
<td>Carry set</td>
<td>C</td>
</tr>
<tr>
<td>NE</td>
<td>Not equal</td>
<td>Z</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal</td>
<td>Z</td>
</tr>
<tr>
<td>VC</td>
<td>Overflow clear</td>
<td>V</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow set</td>
<td>V</td>
</tr>
<tr>
<td>PL</td>
<td>Plus</td>
<td>N</td>
</tr>
<tr>
<td>MI</td>
<td>Minus</td>
<td>N</td>
</tr>
<tr>
<td>CE</td>
<td>Greater or equal</td>
<td>N . V + N . V</td>
</tr>
<tr>
<td>LT</td>
<td>Less than</td>
<td>N . V + N . V</td>
</tr>
<tr>
<td>LE</td>
<td>Less or equal</td>
<td>Z + N . V + N . V</td>
</tr>
</tbody>
</table>

*Note: * = Boolean AND, + = Boolean OR, and ̅ = Boolean NOT N.
* Not available for the Bcc instruction.

---

The MC68000 family supports data types of bits, byte integers (8 bits), word integers (16 bits), long word integers (32 bits), and binary coded decimal (BCD) digits. In addition to these, four new data types are supported by the MC68020: variable-width bit field, packed BCD digits, quad words (64 bits), and variable-length operands.

Data stored in memory are organized on a byte-addressable basis, where the lower addresses correspond to higher-order bytes. The MC68020
does not require data to be aligned on even byte boundaries, but data that are not aligned are transferred less efficiently. Instruction words must be aligned on even byte boundaries. Figure 6.4 shows how data are organized in memory.

---

Table 6.2 shows decoding of the function code pins. The function code pins define the user/supervisor program and data spaces in the same way as the MC68000, except that FC2 FC1 FC0 = 111 for MC68020 defines a new cycle called the CPU space cycle. Note that for MC68000, FC2, FC1, FC0 = 111 provides the interrupt acknowledge cycle. CPU space is not intended for general instruction execution, but is reserved for processor functions. The CPU space has been subdivided into 16 types of access. The type of CPU access is indicated by address bits (A19-A16) in combination with the CPU space function code (FC2 FC1 FC0 = 111).

Table 6.3 defines the four different types of CPU accesses. The definition of regions in the CPU space makes it possible to acknowledge break...
points and interrupts and to communicate with coprocessors and other special devices (such as the MMU) without dictating memory organization for user- and supervisor-related activity.

The MC68020 has three stack pointers: the user stack pointer (USP) register A7, the interrupt stack pointer (ISP) register A7', and the master stack pointer (MSP) register A7''. During normal operation, most codes will be executed in user space and programs will use A7 stack for temporary storage and parameter passing between software routines (modules). ISP register is only used when an exception occurs, such as an external interrupt.

TABLE 6.2

<table>
<thead>
<tr>
<th>Function code</th>
<th>FC2</th>
<th>FC1</th>
<th>FC0</th>
<th>Address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(Undefined, reserved)*</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>User data space</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>User program space</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(Undefined, reserved)*</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Supervisor data space</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Supervisor program space</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CPU space</td>
</tr>
</tbody>
</table>

* Address space 3 is reserved for user definition, while 0 and 4 are reserved for future use by Motorola.

TABLE 6.3

<table>
<thead>
<tr>
<th>CPU space address types.</th>
<th>Function Code</th>
<th>Address bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Address</td>
<td>11</td>
<td>31</td>
</tr>
<tr>
<td>Access Level</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>Processor Status</td>
<td>11</td>
<td>31</td>
</tr>
<tr>
<td>Internal Acknowledge</td>
<td>11</td>
<td>31</td>
</tr>
<tr>
<td>CPU Space Type Field</td>
<td>15</td>
<td>19</td>
</tr>
</tbody>
</table>

The MC68020 uses these stack pointer registers to manage different levels of program execution and interrupt handling.
when control is passed to supervisor mode and the relevant exception process is performed. The (MSP) holds process-related information for the various tasks and allows for the separation of task-related and non-task-related exception tracking. When the master stack is enabled through bit (M) in the SR, all noninterrupting exceptions, such as divide by zero, software traps, and privilege violation, are placed in the user's process control block on the master stack.

6.6 MC68020 ADDRESSING MODES

Figure 6.5 lists the MC68020’s 18 addressing modes. Table 6.4 compares the addressing modes of the MC68000 with those of the MC68020. All index Xn in the table can be scaled 1, 2, 4, or 8. Since MC68000 addressing modes are covered in detail with examples, the MC68020 modes which are not available in the MC68000 will be covered in the following discussion.

6.6.1 ADDRESS REGISTER INDIRECT (ARI) WITH INDEX (SCALED) AND 8-BIT DISPLACEMENT

Assembler syntax: \( (d8, An, Xn, \text{size} \times \text{SCALE}) \)

EA = (An) + (Xn.size \times \text{scale value}) + d8

Xn can be W or L.

If index register (An or Dn) is 16 bits, then it is sign-extended to 32 bits and then multiplied by 1, 2, 4, or 8 prior to being used in EA calculation. d8 is also sign-extended to 32 bits prior to EA calculation. An example is

\[
\text{MOV} \ W \ (0, A2, D2, W^2), D1
\]

Suppose that \([A2] = 8500000000, [D2.W] = 10000, \) and \([8500000000] = 1577, \) then after execution of the \text{MOV}, \([D1.W^2] = 1577, \) since EA = \(8500000000 + 10000 \times 2 = 8500200000\).

6.6.2 ARI WITH INDEX (BASE DISPLACEMENT, bd: VALUE 0 OR 16 BITS OR 32 BITS)

Assembler syntax: \( (bd, An, Xn, \text{size} \times \text{SCALE}) \)

EA = (An) + (Xn.size \times \text{SCALE}) + bd

<table>
<thead>
<tr>
<th>Addressing Modes</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Direct</td>
<td>Dn</td>
</tr>
<tr>
<td>Data Register Direct</td>
<td>An</td>
</tr>
<tr>
<td>Address Register Direct</td>
<td>An</td>
</tr>
<tr>
<td>Register indirect</td>
<td>(An)</td>
</tr>
<tr>
<td>Address Register Indirect with Post Increment</td>
<td>(An) + 1</td>
</tr>
<tr>
<td>Address Register Indirect with Predecrement</td>
<td>(An) - 1</td>
</tr>
<tr>
<td>Address Register Indirect with Displacement</td>
<td>(An) + offset</td>
</tr>
<tr>
<td>Register indirect with index</td>
<td>(bd, An, Xn)</td>
</tr>
<tr>
<td>Address Register Indirect with Index (8-bit Displacement)</td>
<td>(bd, An, Xn) + offset</td>
</tr>
<tr>
<td>Address Register Indirect with Index</td>
<td>(bd, An, Xn)</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>(bd, An, Xn)</td>
</tr>
<tr>
<td>Memory indirect with displacement</td>
<td>(bd, An, Xn) + offset</td>
</tr>
<tr>
<td>Program counter</td>
<td>(bd, An, Xn)</td>
</tr>
<tr>
<td>Program counter indirect with displacement</td>
<td>(bd, An, Xn) + offset</td>
</tr>
<tr>
<td>Program counter indirect with index (8-bit Displacement)</td>
<td>(bd, Xn)</td>
</tr>
<tr>
<td>Program counter indirect with index</td>
<td>(bd, Xn)</td>
</tr>
<tr>
<td>Program counter indirect with index (base displacement)</td>
<td>(bd, Xn)</td>
</tr>
<tr>
<td>Absolute</td>
<td>(bd, Xn)</td>
</tr>
<tr>
<td>Absolute short</td>
<td>(bd, Xn)</td>
</tr>
<tr>
<td>Absolute long</td>
<td>(bd, Xn)</td>
</tr>
<tr>
<td>Immediate</td>
<td>#data</td>
</tr>
</tbody>
</table>

NOTES:
- Dn = Data Register, Dn:07
- An = Address Register, An:10
- Xn = Address or data register used as an index register; form is Xn,SCALE, where SCALE = 1 indicates index register size and SCALE = 2, 4, or 8 (index register multiplied by SCALE) use of SCALE and/or SCALE is optional.
- bd = A base/displacement base displacement; when present, size can be 16 or 12 bits.
- offset = Offset displacement; added as part of effective address calculation after any memory indirection; size is optional with a size of 16 or 32 bits.
- PC = Program Counter
- (bd) = Immediate value of 8, 16, or 32 bits.
- (1) = Effective Address.
- (2) = Use as indirect address to long word address.

FIGURE 6.5 MC68020 addressing modes.
TABLE 6.4
Addressing Modes, MC68000 Vs. MC68020

<table>
<thead>
<tr>
<th>Addressing modes available</th>
<th>68000</th>
<th>68020</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Register Direct</td>
<td>Dn</td>
<td>Yes</td>
</tr>
<tr>
<td>Address Register Direct</td>
<td>An</td>
<td>Yes</td>
</tr>
<tr>
<td>Address Register Indirect</td>
<td>(An)</td>
<td>Yes</td>
</tr>
<tr>
<td>(ARI)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARI with Postincrement</td>
<td>(An)+</td>
<td>Yes</td>
</tr>
<tr>
<td>ARI with Predecrement</td>
<td>-(An)</td>
<td>Yes</td>
</tr>
<tr>
<td>ARI with Displacement (16-bit displ)</td>
<td>(d, An)</td>
<td>Yes</td>
</tr>
<tr>
<td>ARI with Index (8-bit displ)</td>
<td>(d, An, Xn)</td>
<td>Yes*</td>
</tr>
<tr>
<td>ARI with Index (Base Displ: 0, 16, 32)</td>
<td>(bd, An, Xn)</td>
<td>No</td>
</tr>
<tr>
<td>Memory Indirect (Postindexed)</td>
<td>(bd, An, Xn, od)</td>
<td>No</td>
</tr>
<tr>
<td>Memory Indirect (Preindexed)</td>
<td>(bd, An, Xn, od)</td>
<td>No</td>
</tr>
<tr>
<td>PC Indirect with Displ. (16-Bit)</td>
<td>(d, PC)</td>
<td>Yes</td>
</tr>
<tr>
<td>PC Indirect with Index (8-Bit Displ)</td>
<td>(d, PC, Xn)</td>
<td>Yes*</td>
</tr>
<tr>
<td>PC Indirect with Index (Base Displ)</td>
<td>(bd, PC, Xn)</td>
<td>No</td>
</tr>
<tr>
<td>PC Memory Indirect (Postindexed)</td>
<td>(bd, PC, Xn, od)</td>
<td>No</td>
</tr>
<tr>
<td>PC Memory Indirect (Preindexed)</td>
<td>(bd, PC, Xn, od)</td>
<td>No</td>
</tr>
<tr>
<td>Absolute Short</td>
<td>xxxL/W</td>
<td>Yes</td>
</tr>
<tr>
<td>Absolute Long</td>
<td>xxxL</td>
<td>Yes</td>
</tr>
<tr>
<td>Immediate</td>
<td>#data</td>
<td>Yes</td>
</tr>
</tbody>
</table>

* 68000 has no scaling capability; 68020 can scale Xn by 1, 2, 4 or 8

The figure below shows the use of ARI with index, Xn and base displacement, bd for accessing tables or arrays:

<table>
<thead>
<tr>
<th>An</th>
</tr>
</thead>
<tbody>
<tr>
<td>bd</td>
</tr>
<tr>
<td>Xn * SCALE</td>
</tr>
</tbody>
</table>

An example is MOVE.W ($3000, A2, D1.W * 4), D5. If [A2] = $3000 0000, [D1.W] = $0200, and [$3000 0000] = $0174, then after this move, [D5]$_{hex} =$0174, since EA = $3000 + $3000 0000 + $0200 * 4 = $3000 5800.

6.6.3 MEMORY INDIRECT

Memory indirect is distinguished from address register indirect by use of square brackets ([ ] ) in the assembler notation.

The concept of memory indirect mode is depicted below:

In the above, register A5 points to the effective address $2000 0501. Since CLR ([A5]) is a 16-bit clear instruction, two bytes in location $2000 0501 and $2000 0502 are cleared to zero.

Memory indirect mode can be indexed with scaling and displacements.

There are two types of memory indirect with scaled index and displacement: postindexed memory indirect mode and preindexed memory indirect mode.

For postindexed memory indirect, an indirect memory address is first calculated using the base register (An) and base displacement (bd). This address is used for an indirect memory access of a long word followed by adding a scaled indexed operand and an optional outer displacement (od) to generate the effective address. bd and od can be zero, 16 bits, or 32 bits.

In postindexed memory indirect mode, indexing occurs after memory indirect.

Assembler syntax:

```
(EA) = (bd + An) + Xn.size * Scale + od
```

The concept is depicted on the following page.

An example is MOVE.W ([$0004, A1], D1.W * 2, D2). If [A1] = $2000 0000, [$2000 0004] = $5000 3000, [D1.W] = $5002, [$0000 3006] = $1A40, then after execution of the above MOVE, intermediate pointer = (4 * $2000 0000) = $2000 0004, [$2000 0004], which is $5000 3000 used as a pointer. Therefore, EA = $5000 3000 + $5000 0004 + $2 = $5000 3006; hence, [D2]$_{low 16}$ = $1A40.
6.6 MC68020 Addressing Modes

As an application of memory indirect preindexed mode, consider several I/O devices in a system. The addresses of these devices can be held in a table pointed to by An, bd, and Xn. The actual programs for the devices can be stored in memory pointed to by the respective device addresses and od.

As an example of memory indirect preindexed mode, consider MOVE.W 
(\{10004, A2, D1.W * 2\}, D5). If \{A2\} = $A000 0000, [D1.W] = $0062, [S3000 000C] = $0024 1782, [S0024 1784] = $9F270, then after execution of the above MOVE, intermediate pointer = $3000 0006 + 4 - S0002 * 4 = $3000 000C. Therefore, [S2000 000C] which is $0024 1782 is used as a pointer to memory. EA = $0024 1782 + 2 = $0024 1784. Hence, [DS]_{add}= $9F270. Note that in the above, bd, Xn, and od are sign-extended to 32 bits if one (or more) of them is 16 bits wide before the calculation.

A summary of MC68020 memory indirect mode is provided below:

6.6.4 MEMORY INDIRECT WITH PC

In this mode, PC (program counter) is used to form the address rather than an address register. The effective address calculation is similar to address register indirect. A summary of PC indirect mode is given in the following.

Assemblr syntax:

\[(bd, An, Xn.size * Scale), od\]

\[EA = (bd + An + Xn.size * Scale) + od\]
6.6.4.a PC Indirect with Index (8-Bit Displacement)

The effective address is obtained by adding the PC contents, the sign-extended displacement, and the scale indexed (sign-extended to 32 bits if it is 16 bits before calculation) register.

Assembler syntax:

\[(d3, PC, Xn.size \times Scale)\]

\[EA = (PC) + (Xn.size \times SCALE) + d8\]

For example, consider \texttt{MOVE.W D2, (2, PC, D1.W + 2)}. If \([PC] = 04000 \ 0020, [D1.W] = 0020, [D2.W] = 020A2\), then after this \texttt{MOVE},

\[EA = 2 + 04000 \ 0020 + 0020 \times 2 = 04000 \ 0062.\]

Hence, \([04000 \ 0062] = 020A2\).

6.6.4.b PC Indirect with Index (Base Displacement)

This address of the operand is obtained by adding the PC contents, the scaled index register contents, and the base displacement.

Assembler syntax:

\[(bd, PC, Xn.size \times Scale)\]

\[EA = (PC) + (Xn.size \times SCALE) + bd\]

\(Xn\) and \(bd\) are sign-extended to 32 bits if either or both are 16 bits.

6.6.4.c PC Indirect (Postindexed)

An intermediate memory pointer in program space is calculated by adding PC (used as a base register) and \(bd\). The 32-bit contents of this address is used in EA calculation. EA is obtained by adding the 32-bit contents with a scaled index register and \(od\). Note that \(bd, od, or index register is sign-extended to 32 bits before using in calculation if one (or more) is 16 bits.

Assembler syntax:

\[(bd, PC, Xn.size \times Scale, od)\]

\[EA = (bd + PC) + (Xn.size \times Scale + od)\]

Consider another example \texttt{MOVE.W (2, PC, D1.W*4, 0)}, \(D1, [PC] = 03000 \ 0060, [D1.W] = 0010, [03000 \ 0024] = 02040 \ 0090, [02040 \ 0090] = 0A240\), then after this \texttt{MOVE}, \([D1.W] = 0A240\).

6.6.4.d PC Indirect (Preindexed)

The scaled index register is added to the PC and \(bd\). This sum is then used as an indirect address into the program space. The 32-bit value at this address is added to \(od\) to find EA.

Assembler syntax:

\[(bd, PC, Xn.size \times Scale, od)\]

\[EA = (bd + PC + Xn.size \times Scale) + od\]

\(od, bd, or the index register is sign-extended to 32 bits if one (or more) of them is 16 bits before the EA calculation.

As an example, consider \texttt{MOVE.W (14, PC, D1.W*2, 4)}, \(D5, [PC] = 05000 \ 0000, [D1.W] = 0010, [05000 \ 0024] = 02050 \ 7004, [02050 \ 7004] = 050708\), then after this \texttt{MOVE}, \([D5.W] = 050708\).

A summary of PC modes is provided on the following page.

Example 6.1

Show the contents of registers \(A2, D4, A5,\) and the affected memory location(s), after execution of the following instruction:

\texttt{MOVEA.W (0, A2, D4.L), A5}
Example 6.2

The following MC68000 instruction sequence:

\[
\text{MOVEA.L 6 (USP), A1} \\
\text{MOVE.W (A1), D5}
\]

is used by a subroutine to access a parameter whose address has been passed into A1 and then moves the parameter to D5.

Find the equivalent MC68020 instruction.

Solution

\[
\text{MOVE.W (16, USP), D5}
\]

Example 6.3

Find a MC68020 compare instruction with the appropriate addressing mode to replace the following MC68000 instruction sequence:

\[
\text{ASL.L} \quad \#3, \text{D6} \\
\text{CMP.L} \quad 0, (\text{A1, D6.L}), \text{D3}
\]

Solution

The equivalent MC68020 instruction is

\[
\text{CMP.L (0, A1, D6.L * 8), D3}
\]

Example 6.4

Find the contents of register A1, D0, D5 and the affected memory locations after execution of:

\[
\text{MOVE.B (0, A1, D0.W * 2), D5}
\]

Assume the following data prior to execution of the MOVE instruction:

\[
\begin{align*}
\text{A1} & = \$0000 \quad \text{D0} = \$0000 \\
\text{D5} & = \$7124 \\
\text{D0000 2006} & = \$51 \\
\text{D0000 2007} & = \$74 \\
\text{D0000 2008} & = \$82 \\
\text{D0000 2009} & = \$70
\end{align*}
\]

Effective address:

\[
\begin{align*}
\text{D4.L} & = \$3072 \quad \text{D400} \\
\text{A2.L} & = \$0571 \quad \text{6660} \\
\text{d} & = \$0000 \quad \text{0000} \\
\text{EA} & = \$35E2 \quad \text{E2 60}
\end{align*}
\]

Therefore, after execution of the MOVEA instruction:

\[
\begin{align*}
\text{[A2]} & = \$0571 \quad \text{6660} \\
\text{[D4]} & = \$3072 \quad \text{8400} \\
\text{[A5]} & = \$0000 \quad \text{0000}
\end{align*}
\]
6.7 INSTRUCTIONS

All MC68020 instructions are at least one word, and up to 11 words long. Figure 6.5 shows the general format of an instruction. The first word in the instruction format (the operation word) determines the length of the instruction and the operation to be performed. The remaining words in the instruction are called extension words and further specify the instruction and operands. These extension words can be immediate operands, extensions to the EA mode specified in the operation word, branch displacements, bit numbers or bit field specifications, special register specifications, trap operands, pack/unpack constants, argument counts, or coprocessor condition codes.

Operation Word
(One Word, Specifies Operation and Modes)

Special Operand Specifiers
(If Any, One or Two Words)

Immediate Operand or Source Effective Address Extension
(If Any, One to Five Words)

Destination Effective Address Extension
(If Any, One to Five Words)

FIGURE 6.6 MC68020 instruction word general format.

The MC 68020 instruction set (Table 6.5) includes all 68000 instructions, plus some new ones. Some of the 68000 instructions are enhanced. These are listed in Table 6.6. Over 20 new instructions have been added to provide new functionality. These instructions are listed below:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFBG</td>
<td>Bit field change</td>
</tr>
<tr>
<td>BFCR</td>
<td>Bit field clear</td>
</tr>
<tr>
<td>BFEXT</td>
<td>Bit field extended</td>
</tr>
<tr>
<td>BFEXTU</td>
<td>Bit field unsigned extract</td>
</tr>
<tr>
<td>BFFFF</td>
<td>Bit field find first set</td>
</tr>
<tr>
<td>BFINS</td>
<td>Bit field insert</td>
</tr>
<tr>
<td>BFSET</td>
<td>Bit field set</td>
</tr>
<tr>
<td>BFTST</td>
<td>Bit field test</td>
</tr>
<tr>
<td>CALLM</td>
<td>Call module</td>
</tr>
<tr>
<td>CAS</td>
<td>Compare and swap</td>
</tr>
<tr>
<td>CAS2</td>
<td>Compare and swap (two operands)</td>
</tr>
<tr>
<td>CHK2</td>
<td>Check register against upper and lower bounds</td>
</tr>
<tr>
<td>CMP2</td>
<td>Compare register against upper and lower bounds</td>
</tr>
<tr>
<td>copA</td>
<td>Coprocessor branch on coprocessor condition</td>
</tr>
<tr>
<td>copB</td>
<td>Coprocessor test condition, decrement, and branch</td>
</tr>
<tr>
<td>copC</td>
<td>Coprocessor general function</td>
</tr>
<tr>
<td>copR</td>
<td>Coprocessor restore internal state</td>
</tr>
<tr>
<td>copS</td>
<td>Coprocessor save internal state</td>
</tr>
<tr>
<td>copT</td>
<td>Coprocessor set according to coprocessor condition</td>
</tr>
<tr>
<td>copTRAPcc</td>
<td>Coprocessor trap on coprocessor condition</td>
</tr>
<tr>
<td>PACK</td>
<td>Pack BCD</td>
</tr>
<tr>
<td>RTM</td>
<td>Return from module</td>
</tr>
<tr>
<td>UNFJ</td>
<td>Unpack BCD</td>
</tr>
</tbody>
</table>

In the following sections, the 68020 instructions listed below will be discussed in detail:

1. 68020 new privileged MOVE instructions
2. TRAP instruction
3. CHK/CHK2 and CMP/CMP2 instructions
4. TRAPcc instructions
5. Bit field instructions
6. PACK and UNFJ instructions
7. Multiplication and division instructions
8. 68000 enhanced instructions
### TABLE 6.5
MC68020 Instruction Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCD</td>
<td>Add decimal with extend</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
</tr>
<tr>
<td>ADDA</td>
<td>Add address</td>
</tr>
<tr>
<td>ADDI</td>
<td>Add immediate</td>
</tr>
<tr>
<td>ADDQ</td>
<td>Add quick</td>
</tr>
<tr>
<td>ADDX</td>
<td>Add with extend</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>ANDI</td>
<td>Logical AND immediate</td>
</tr>
<tr>
<td>ASL, ASR</td>
<td>Arithmetic shift left and right</td>
</tr>
<tr>
<td>Bcc</td>
<td>Branch conditionally</td>
</tr>
<tr>
<td>BCHG</td>
<td>Test bit and change</td>
</tr>
<tr>
<td>BCLR</td>
<td>Test bit and clear</td>
</tr>
<tr>
<td>BFCHG</td>
<td>Test bit field and change</td>
</tr>
<tr>
<td>BFCLR</td>
<td>Test bit field and clear</td>
</tr>
<tr>
<td>BFEXTS</td>
<td>Signed bit field extract</td>
</tr>
<tr>
<td>BFEXTU</td>
<td>Unsigned bit field extract</td>
</tr>
<tr>
<td>BFFIO</td>
<td>Bit field find first one</td>
</tr>
<tr>
<td>BFINS</td>
<td>Bit field insert</td>
</tr>
<tr>
<td>BFSET</td>
<td>Test bit field and set</td>
</tr>
<tr>
<td>BFTST</td>
<td>Test bit field</td>
</tr>
<tr>
<td>BKPT</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>BRA</td>
<td>Branch</td>
</tr>
<tr>
<td>BSET</td>
<td>Test bit and set</td>
</tr>
<tr>
<td>BSR</td>
<td>Branch to subroutine</td>
</tr>
<tr>
<td>BTST</td>
<td>Test bit</td>
</tr>
<tr>
<td>CALM</td>
<td>Call module</td>
</tr>
<tr>
<td>CAS</td>
<td>Compare and swap operands</td>
</tr>
<tr>
<td>CAS2</td>
<td>Compare and swap dual operands</td>
</tr>
<tr>
<td>CHK</td>
<td>Check register against bound</td>
</tr>
<tr>
<td>CHK2</td>
<td>Check register against upper and lower bounds</td>
</tr>
<tr>
<td>CLR</td>
<td>Clear</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
</tr>
<tr>
<td>CMPA</td>
<td>Compare address</td>
</tr>
<tr>
<td>CMPI</td>
<td>Compare immediate</td>
</tr>
<tr>
<td>CMPM</td>
<td>Compare memory to memory</td>
</tr>
<tr>
<td>CMP2</td>
<td>Compare register against upper and lower bounds</td>
</tr>
<tr>
<td>DBcc</td>
<td>Test condition, decrement, and branch</td>
</tr>
<tr>
<td>DIVS, DIVSL</td>
<td>Signed divide</td>
</tr>
<tr>
<td>DIVU, DIVUL</td>
<td>Unsigned divide</td>
</tr>
<tr>
<td>EOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>EORI</td>
<td>Logical exclusive OR immediate</td>
</tr>
<tr>
<td>EXG</td>
<td>Exchange registers</td>
</tr>
<tr>
<td>EXT, EXTB</td>
<td>Sign extend</td>
</tr>
<tr>
<td>ILLEGAL</td>
<td>Take illegal instruction trap</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
</tr>
<tr>
<td>JSR</td>
<td>Jump to subroutine</td>
</tr>
<tr>
<td>LEA</td>
<td>Load effective address</td>
</tr>
<tr>
<td>LINK</td>
<td>Link and allocate</td>
</tr>
<tr>
<td>LSL, LSR</td>
<td>Logical shift left and right</td>
</tr>
<tr>
<td>MOVE</td>
<td>Move</td>
</tr>
<tr>
<td>MOVEA</td>
<td>Move address</td>
</tr>
<tr>
<td>MOVE CCR</td>
<td>Move condition code register</td>
</tr>
<tr>
<td>MOVE SR</td>
<td>Move status register</td>
</tr>
<tr>
<td>MOVE USP</td>
<td>Move user stack pointer</td>
</tr>
<tr>
<td>MOVEC</td>
<td>Move control register</td>
</tr>
<tr>
<td>MOVEM</td>
<td>Move multiple registers</td>
</tr>
<tr>
<td>MOVEP</td>
<td>Move peripheral</td>
</tr>
<tr>
<td>MOVEQ</td>
<td>Move quick</td>
</tr>
<tr>
<td>MOVES</td>
<td>Move alternate address space</td>
</tr>
<tr>
<td>MULS</td>
<td>Signed multiply</td>
</tr>
<tr>
<td>MULLU</td>
<td>Unsigned multiply</td>
</tr>
<tr>
<td>NBCD</td>
<td>Negate decimal with extend</td>
</tr>
<tr>
<td>NEG</td>
<td>Negate</td>
</tr>
</tbody>
</table>
TABLE 6.5 (continued)
MC68020 Instruction Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NECl</td>
<td>Negate with extend</td>
</tr>
<tr>
<td>NOCl</td>
<td>No operation</td>
</tr>
<tr>
<td>NOTCl</td>
<td>Logical complement</td>
</tr>
<tr>
<td>ORCl</td>
<td>Logical inclusive OR</td>
</tr>
<tr>
<td>ORCl</td>
<td>Logical inclusive OR immediate</td>
</tr>
<tr>
<td>PACC</td>
<td>Pack BCD</td>
</tr>
<tr>
<td>PEAc</td>
<td>Push effective address</td>
</tr>
<tr>
<td>RESETc</td>
<td>Reset external devices</td>
</tr>
<tr>
<td>ROLc, RORc</td>
<td>Rotate left and right</td>
</tr>
<tr>
<td>ROXLc, ROXRc</td>
<td>Rotate with extend left and right</td>
</tr>
<tr>
<td>RTDc</td>
<td>Return and destinate</td>
</tr>
<tr>
<td>RTEc</td>
<td>Return from exception</td>
</tr>
<tr>
<td>RTMc</td>
<td>Return from module</td>
</tr>
<tr>
<td>RT Rc</td>
<td>Return and restore codes</td>
</tr>
<tr>
<td>RTS c</td>
<td>Return from subroutine</td>
</tr>
<tr>
<td>SBCDc</td>
<td>Subtract decimal with extend</td>
</tr>
<tr>
<td>Scc</td>
<td>Set conditionally</td>
</tr>
<tr>
<td>STOPc</td>
<td>Stop</td>
</tr>
<tr>
<td>SUBc</td>
<td>Subtract</td>
</tr>
<tr>
<td>SUBAc</td>
<td>Subtract address</td>
</tr>
<tr>
<td>SUBIc</td>
<td>Subtract immediate</td>
</tr>
<tr>
<td>SUBQc</td>
<td>Subtract quick</td>
</tr>
<tr>
<td>SUBXc</td>
<td>Subtract with extend</td>
</tr>
<tr>
<td>SWAPc</td>
<td>Swap register words</td>
</tr>
<tr>
<td>TASc</td>
<td>Test operand and set</td>
</tr>
<tr>
<td>TRAPc</td>
<td>Trap</td>
</tr>
<tr>
<td>TRAPcc</td>
<td>Trap conditionally</td>
</tr>
<tr>
<td>TRAPVc</td>
<td>Trap on overflow</td>
</tr>
<tr>
<td>TSTc</td>
<td>Test operand</td>
</tr>
<tr>
<td>UNLKc</td>
<td>Unlink</td>
</tr>
<tr>
<td>UNPKc</td>
<td>Unpack BCD</td>
</tr>
</tbody>
</table>

6.7 Instructions

COPROCESSOR INSTRUCTIONS

cpBCC     | Branch conditionally       |
cpDBcc    | Test coprocessor condition, decrement, and branch |
cpGEN     | Coprocessor general instruction |
cpRESTORE | Restore internal state of coprocessor |
cpSAVE    | Save internal state of coprocessor |
cpSc      | Set conditionally           |
cpTRAPcc  | Trap conditionally         |

TABLE 6.6
MC68020 Instruction Enhancements

MULS, MULU, DIVS, DIVU, Bcc, BRA, BSR, LINK, MOVEC, BKPT

Operations extended to 32-bit operands
Displacements extended to 32 bits
New control registers may be accessed
Op code substitution supported

The instructions listed below are of an advanced nature and will be
discussed in Chapter 7:

- BKPT instructions
- CALLM/RTM instructions
- TAS and CAS/CAS2 instructions
- Coprocessor instructions

6.7.1 NEW PRIVILEGED MOVE INSTRUCTION

The new privileged move instructions are executed by the 68000 in the
supervisor mode.

The MOVE instructions are summarized in Table 6.7. The MOVEC
instruction was added to allow the new supervisor registers (Re) to be
accessed. Since these registers are used for system control, they are gener-
ally referred to as control registers and include the vector base register
(VBR), the source function and destination code registers (SFC, DFC), the
master, interrupt, and user stack pointers (MSP, ISP, USP), and the cache
control registers (CACS, CAAR). Register (Rn) can be either an address register or data register.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand size</th>
<th>Operation</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>16</td>
<td>SRC → destination</td>
<td>MOVE SR, (EA)</td>
</tr>
<tr>
<td>MOVC</td>
<td>32</td>
<td>RC → Rn</td>
<td>MOVECL RC, Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rn → RC</td>
<td>MOVECL Rn, RC</td>
</tr>
<tr>
<td>MOVES</td>
<td>8, 16, 32</td>
<td>Rn → destination using DFC</td>
<td>MOVES, Rn, (EA)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Source using SFC → Rn</td>
<td>MOVES, Rn, (EA), Rn</td>
</tr>
</tbody>
</table>

The operand size indicates that these MOVC operations are always long-word. Notice only register-to-register operations are allowed.

A control register (RC) can be copied to an address or data register (Rn), or vice versa. When copying the 3-bit SFC, or DFC register into Rn, all 32 bits of the register are overwritten and the upper 29 bits are "0".

The MOVE to alternate space instruction (MOVES) allows the operating system to access any addressed space defined by the function codes.

It is typically used when an operating system running in the supervisor mode must pass a pointer or value to a previously defined user program or data space.

The MOVES instruction allows register-to-memory or memory-to-register operations. When a memory-to-register occurs, this instruction causes the contents of the source function code register to be placed on the external function hardware pins.

For a register-to-memory move, the processor places the destination function code register on the external function code pins.

The MOVES instruction can be used to MOVE information from one space to another. For example, in order to move the 16-bit content of a memory location addressed by A0 in supervisor data space (FC2 FCI FC0 = 101) to a memory location addressed by A1 in user data space (FC2 FCI FC0 = 001), the following instruction on sequence can be used:

MOVEQ.L #5, D0 ; Move source space 5 to D0
MOVE.L #1, D1 ; Move dest space 1 to D1
MOVC.L D0, SFC ; Initialize SFC
MOVC.L D1, DFC ; Initialize DFC
MOVES.W (A0), D2 ; Move memory location addressed by (A0) and SFC to D2
MOVES.W D2, (A1) ; Move D2 to a memory location addressed by (A1) and DFC

In the above, the first four instructions initialize SFC to 101 and DFC to 001. Since there is no MOVES mem, mem instruction, the register D2 is used as a buffer for the memory-to-memory transfer. MOVES.W (A0), D2 transfers [SFC] to FC2 FCI FC0 and also reads the content of a memory location addressed by SFC and (A0) to D2. The 68020 FC2 FCI FC0 pins can be decoded to enable the appropriate memory chip containing the memory location addressed by (A0). Next, MOVES.W D2, (A1) outputs [DFC] to FC2 FCI FC0 and then moves [D2] to a memory location addressed by (A1) contained in a memory chip which can be enabled by decoding FC2, FC1, and FC0 pins.

Example 6.5
Find the content of memory location $5000 2000$ after execution by MOVES.W SR, [A6]. Assume the following data prior to execution of the MOVE instruction:

\[
[A6] = \{5000 2000\} = \{5000 2000\}
\]

Also, assume supervisor mode.

Solution
SR is moved to a memory location pointed to by $5000 2000$. After execution:

\[
\]

Example 6.6
Find the content of DFC after execution of MOVC.L A5, DFC. Assume the following data prior to execution of the instruction:
Solution
After execution of the MOVEC, \([\text{DPC}] = 101\) \(_\text{z}\), \([\text{A5}] = \$2000 \ 0105\).

Example 6.7
Find the contents of D5 and the function code pins FC2, FC1, and FC0 after execution of MOVES B D5, (A5). Assume the following data prior to execution of the MOVES:

\[
\begin{align*}
[\text{SFC}] &= 101 \_z, \ [\text{DPC}] = 101 \_z, \\
[\text{A5}] &= \$7000 \ 0023, \\
[\text{D5}] &= \$718F \ 2A05, \\
[\$7000 \ 0029] &= \$01, \ [\$7000 \ 0021] &= \$F1, \\
[\$7000 \ 0022] &= \$20, \\
[\$7000 \ 0023] &= \$2A
\end{align*}
\]

Solution
After execution of the above MOVES:

\[
\begin{align*}
\text{FC2} & \rightarrow \text{FC1} \rightarrow \text{FC0} = 100 \_z, \\
[\$7000 \ 0022] &= \$005
\end{align*}
\]

6.7.2 RETURN AND DELOCATE INSTRUCTION

Return and delocate instruction RTD is useful when a subroutine has the responsibility to remove parameters off the stack by the calling routine. Note that the calling routine’s JSR (jump to subroutine) or BSR (branch to subroutine) instructions do not automatically push parameters onto the stack prior to the call, as the CALLM instructions. Rather, the pushed parameters must be placed there using the MOVE instruction. Table 6.8 shows the format of the RTD instruction.

<table>
<thead>
<tr>
<th>Instruction size</th>
<th>Operation</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTD</td>
<td>(SP) \to \text{PC}, SP + 4 + d \to SP</td>
<td>\text{RTD} # \langle\text{displacement}\rangle</td>
</tr>
</tbody>
</table>

Table 6.8
RTD Instruction

The RTD instruction operates as follows:
1. Read the long word from memory pointed to by the stack pointer.
2. Copy it into the program counter.
3. Increment the stack pointer by 4.
4. Sign extend the 16-bit immediate data displacement to 32 bits.
5. Add it to the stack pointer.

Since parameters are pushed onto the stack to lower memory locations, only a positive displacement should be added to the SP when removing parameters from the stack. The displacement value (16 bits) is sign-extended to 32 bits.

Example 6.8
Write a 68020 instruction sequence to illustrate the use of RTD instruction by using BSR instruction and pushing three 32-bit parameters onto the stack.

Solution

\[
\begin{align*}
\text{MOVE.L PAR1}, -(\text{SP}) \\
\text{MOVE.L PAR2}, -(\text{SP}) \\
\text{MOVE.L PAR3}, -(\text{SP}) \\
\text{CALLM} \\
\text{RTD#12} \\
\text{BSR SUBR} \\
\text{RTD#12} \\
\text{RTD#12}
\end{align*}
\]

Calling routine pushes parameters on stack:
- this causes the stack pointer to be decremented by 12

Calling routine calls subroutine and the PC stacked;
- the subroutine then accesses the parameters to perform the task

Last instruction of the subroutine returns and delocates the parameters off stack by adding (12) to the stack pointer
6.7.3 CHK/CHK2 AND CMP/CMP2 INSTRUCTIONS

The enhanced MC68020 check instruction (CHK) now compares a 32-bit two's complement integer value residing in a data register (Dn) against a lower bound (L.B.) value of zero and against an upper bound (U.B.) value of the programmer's choice. These bounds are located beginning at the effective address (EA) specified in the instruction format.

The CHK instruction has the following format:

\[
\text{CHK (EA), Dn}
\]

where (S) is the operand size designator which is either word (.W) or long word (.L).

If the data register value is less than zero (Dn < 0) or if the data register is greater than the upper bound (Dn > UB), then the processor traps through exception vector 6 (offset $18$) in the exception vector table. Of course, the operating system or the programmer must define a check service handler routine at this vector address. After completion of the CHK instruction, the negative (N) bit is the only condition code register (CCR) bit that is defined or affected. If the compared register (Dn) is less than zero, then the N-bit is set to one. If the data register exceeds the upper bound, then the N-bit is cleared to zero. If the CHK instruction finds that the compared register value is within bounds (i.e., 0 < Dn < UB value), then all CCR bits except X (X is unaffected) are undefined (U), and program execution resumes with the next instruction in the instruction flow. This instruction can be used for maintaining array subscripts since all subscripts can be checked against an upper bound (i.e., UB = array size minus one). If the compared subscript is within the array bounds (0 < subscript value < UB value), then the subscript is valid, and the program continues normal instruction execution. If the subscript value is out of array limits (i.e., 0 > subscript value, or the subscript value > UB value), then the processor traps through the CHK exception.

**Example 6.9**

Find the effects of execution of the MC68020 CHK instruction: CHKL (A5), D3, where A5 represents a memory pointer to the array's upper bound value. Register D3 contains the subscript value to be checked against the array bounds. Assume the following data prior to execution of the CHK instruction:

\[
\begin{align*}
[D3] &= \text{S}0150 \text{7126}, & [A5] &= \text{S}00730004, \\
[\text{S}0071 \text{0004}] &= \text{S}0150 0000
\end{align*}
\]

This table shows the CHK instruction operation:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand size</th>
<th>Operation</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHK</td>
<td>16, 32</td>
<td>If Dn &lt;0 or Dn&gt;</td>
<td>TRAP CHK (EA), Dn</td>
</tr>
</tbody>
</table>

The 68020 CMP.B (EA), Dn subtracts

\[
\text{or} \quad \text{or} \quad \text{or} .W .L
\]

(EA) from Dn and affects the condition codes without any result.

Both the CHK2 and the CMP2 instructions have similar formats (CHK2.S (EA),R0) and (CMP2.S (EA),Rn). They compare a value contained in a data or address register designated by (Rn) against two (2)
bounds chosen by the programmer. The size of the data to be compared (S) may be specified as either byte (B), word (W), or long word (L). As shown in the figure below, the lower-bound value (LB) must be located in memory at the effective address (EA) specified in the instruction, and the upper-bound value (UB) must follow immediately at the next higher memory address [i.e., UB addr = LB addr + SIZE where SIZE = B (+1), W (+2), or L = (+4)].

If the compared register is a data register (i.e., \( R_n = D_n \)) and the operand size (S) is a byte or word, then only the appropriate low-order part of the data register is checked. If the compared register is an address register (i.e., \( R_n = A_n \)) and the operand size (S) is a byte or word, then the bound operands are sign extended to 32 bits, and the extended operands are compared against the full 32 bits of the address register. The CHK2 and CMP2 instructions both set the CCR carry bit (C) to 1 if the compared register’s data are out of bounds. Likewise, the zero bit (Z) is set to 1 if the data are equal to either bound or otherwise cleared to zero. In the case where an upper bound equals the lower bound, the valid range for comparison becomes a single value. The only difference between the CHK2 and CMP2 instructions is that for comparisons determined to be out of bounds, CHK2 causes exception processing utilizing the same exception vector as the CHK instructions, whereas the CMP2 instruction execution only affects the condition codes.

In both instructions, the compare is performed for either signed or unsigned bounds. The MC68020 automatically evaluates the relationship between the two bounds to determine which kind of comparison to employ. If the programmer wishes to have the bounds evaluated as signed values, the arithmetically smaller value should be the lower bound. If the bounds are to be evaluated as unsigned values, the programmer should make the logically smaller value the lower bound. The following CHK2 and CMP2 instruction examples are identical in that they both utilize the same registers, comparison data, and bound values. The difference is how the upper and lower bounds are arranged.

**Example 6.10**

Determine the effects of CMP2.W (A2), D1. Assume the following
by clearing the CCR carry bit if it is in bounds and setting the carry bit if it is out of bounds.

**Example 6.11**

Find the effects of execution of CHK2W (A2), D1. Assume the following data prior to execution of the CHK2:

\[ [D1] = 05000\text{H} 0200, \hspace{1em} [A2] = 00000\text{H} 7000, \hspace{1em} [30000\text{H} 7000] = 05000 \hspace{1em} [30000\text{H} 7002] = 02000 \]

**Solution**

This time, the value 05000 is located in memory as the lower bound, and the value 02000 as the upper bound. Now, since the lower bound contains the logically smaller value, the programmer is indicating to the 68020 to interpret the bound values as unsigned numbers, representing only a magnitude. Therefore, the instruction evaluates the word value contained in register D1, 02000, to determine if it is greater than or equal to lower bound 05000 or less than or equal to the upper bound 02000. Since the compared value 02000 is less than 05000, the carry bit is set to indicate an out-of-bounds condition and to program traps to the CHK/CHK2 exception vector service routine. Also, since 02000 is not equal to either bound, the zero bit (Z) is cleared. The range of valid values that D1 could contain is shown below:

\[
\begin{array}{c|c|c}
\text{CHK2W(A2,D1)} & \text{D1 W} & \text{D1 W SFF} \\
\hline
\text{D1} & 05000 & 02000 \\
\text{D1 W} & 05000 & 02000 \\
\text{SFF} & 0 & 0 \\
\hline
\end{array}
\]

A typical application for the CHK2 instruction would be to cause a trap exception to occur if a certain subscript value is not within the bounds of some defined array. Using the CHK2 example format just given, if we define an array of 100 elements with subscripts ranging from 50 to 40, and if the two words located at (A2) and (A2+2) contain 50 and 49, respectively, and register D1 contains 100, then execution of the CHK2 instruction would cause a trap through the CHK/CHK2 exception vector.

The operation of the CMP2 and the CHK2 instructions is summarized in Figure 6.7:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Operation</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP2</td>
<td>0, 16, 32</td>
<td>Compare Bn &lt; source - lower bound or Bn &gt; source - upper bound</td>
<td>CMP2 &lt;Bn, Bn&gt;</td>
</tr>
<tr>
<td>CHK2</td>
<td>0, 16, 32</td>
<td>If Bn &lt; source - lower bound or Bn &gt; source - upper bound</td>
<td>CHK2 &lt;Bn, Bn&gt;</td>
</tr>
</tbody>
</table>

**6.7.4 TRAP ON CONDITION INSTRUCTIONS**

The new trap on condition (Trap cc) instruction has been added to allow a conditional trap exception on any of the following conditional conditions, as shown in Table 6.10.

**TABLE 6.10** Conditions for TRAPcc

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>Carry clear</td>
</tr>
<tr>
<td>CS</td>
<td>Carry set</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal</td>
</tr>
<tr>
<td>NE</td>
<td>Never true</td>
</tr>
<tr>
<td>GE</td>
<td>Greater or equal</td>
</tr>
<tr>
<td>GT</td>
<td>Greater than</td>
</tr>
<tr>
<td>HI</td>
<td>High</td>
</tr>
<tr>
<td>LE</td>
<td>Less or equal</td>
</tr>
<tr>
<td>LS</td>
<td>Low or same</td>
</tr>
<tr>
<td>LT</td>
<td>Less than</td>
</tr>
<tr>
<td>MI</td>
<td>Minus</td>
</tr>
<tr>
<td>NE</td>
<td>Not equal</td>
</tr>
<tr>
<td>PL</td>
<td>Plus</td>
</tr>
<tr>
<td>T</td>
<td>Always true</td>
</tr>
<tr>
<td>VC</td>
<td>Overflow clear</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow set</td>
</tr>
</tbody>
</table>

**FIGURE 6.7 Operation of CMP2 and CHK2.**

252.jpg
These are the same conditions that are allowed for the set on condition (SetC) and the branch on condition (BranchC) instructions. The TRAPc instruction evaluates the selected test condition based on the state of the condition code flags, and if the test is true, the MC68020 initiates exception processing by trapping through the same exceptional vector as the TRAPV instruction (vector 7, offset $1C$, VBR = VBR = offset). The trap on cc instruction format is TRAPc (or TRAPc (S) # <data>), where (S) is the operand size designator, which is either word (W) or long word (L).

If either a word or long word operand is specified, a 1- or 2-word immediate operand is placed following the instruction word. The immediate operand(s) consist of arguments that are passed to the trap handler to further define requests or services it should perform. If cc is false, the 68020 does not interpret the immediate operands, but instead adjusts the program counter to the beginning of the following instruction. The exception handler can access this immediate data as an offset to the stack PC. The stack PC is the next instruction to be executed.

A summary of the TRAPc instruction operation is shown in Figure 6.8.

![Figure 6.8 TRAPc operation](image)

### 6.7.5 BIT FIELD INSTRUCTIONS

The bit field instructions allow an operation such as clear, set, one's complement, input, insert, and test one or more bits in a string of bits (bit field).

Table 6.11 lists all the bit field instructions. Note that the condition codes are affected according to the value in field before execution of the instruction. All bit field instructions affect the N and Z bits as shown for BFTST. C and V are always cleared, X is always unaffected.

For all instructions: Z = 1, if all bits in a field prior to execution of the instruction are zero; Z = 0 otherwise. N = 1 if the most significant bit of the field prior to execution of the instruction is one; N = 0 otherwise.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand size</th>
<th>Operation</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAPc</td>
<td>None</td>
<td>If cc then TRAPc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>TRAPc W # &lt;data&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>TRAPc L # &lt;data&gt;</td>
<td></td>
</tr>
</tbody>
</table>

#### TABLE 6.11

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand size</th>
<th>Operation</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFTST 1-32</td>
<td>Field MSB → N, Z = 1 if all bits in field are zero; Z = 0 otherwise</td>
<td>BFTST (EA) [offset + width]</td>
<td></td>
</tr>
<tr>
<td>BFCLR 1-32</td>
<td>0's → field</td>
<td>BFCLR (EA) [offset + width]</td>
<td></td>
</tr>
<tr>
<td>BFSET 1-32</td>
<td>1's → field</td>
<td>BFSET (EA) [offset + width]</td>
<td></td>
</tr>
<tr>
<td>BFCHG 1-32</td>
<td>Field' → field</td>
<td>BFCHG (EA) [offset + width]</td>
<td></td>
</tr>
<tr>
<td>BFEXTS 1-32</td>
<td>Field → Dn; sign extended</td>
<td>BFEXTS (EA) [offset + width], Dn</td>
<td></td>
</tr>
<tr>
<td>BFEXTU 1-32</td>
<td>Field → Dn; zero extended</td>
<td>BFEXTU (EA) [offset + width], Dn</td>
<td></td>
</tr>
<tr>
<td>BFINS 1-32</td>
<td>Dn → field</td>
<td>BFINS Dn, (EA) [offset + width]</td>
<td></td>
</tr>
<tr>
<td>BFFFO+ 1-32</td>
<td>Scan for first bit set in field</td>
<td>BFFFO (EA) [offset + width], Dn</td>
<td></td>
</tr>
</tbody>
</table>

*The offset of the first bit set in bit field is placed in Dn. If no set bit is found, Dn contains the offset plus field width.*

(EL) address of the byte that contains bit 0 of the array
offset #0 – #31 or Dn – 2, (O, 2, 2, 2, 1, 1)
width #1 – #32 or Dn (1 – #32, mod 32)

Immediate offset is from 0 to 31, while offset in Dn can be specified from -2, to 2, 2, 1, 1. All instructions are unsigned. They are useful for memory conservation, graphs, and communications.

As an example, consider BFCLR $5002$ [4:12]. Assume the following memory contents:

```
$5001: 1 1 0 1 0 0 0 0 0 1
$5002: 1 1 0 0 1 1 1 1 0 0
$5003: 0 1 1 0 0 0 0 1
$5004: 0 0 0 1 1 0 0 1 0 1
```

![Example](image)
Bit 7 of the base address $S002 has the offset 6. Therefore, bit 3 of $S002 has offset value 4. Bit 0 of location $S001 has offset value 1, bit 1 of $S001 has the offset value 2, and so on. The above BFCLR instruction clears 12 bits starting with bit 3 of $S002. Therefore, bits 0—3 of location $S002 and bits 0—7 of location $S003 are cleared to zero. Therefore, the above memory contents are as follows:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S001</td>
<td>1010101</td>
</tr>
<tr>
<td>$S002</td>
<td>1001000</td>
</tr>
<tr>
<td>$S003</td>
<td>0101010</td>
</tr>
<tr>
<td>$S004</td>
<td>0001110</td>
</tr>
</tbody>
</table>

The use of bit field instructions may result in memory savings. For example, assume that an input device such as a 12-bit A/D converter interfaced via a 16-bit port of a MC68020-based microcomputer. Now, suppose that one million pieces of data are to be collected from this port. Each 12 bits can be transferred to a 16-bit memory location or bit field instructions can be used.

Using 16-bit location for each 12-bit:
- Memory bytes required:
  - $2 \times 1$ million
  - $2$ million bytes

Using bit fields:
- 12 bits = 1.5 bytes
- Memory requirements = $1.5 \times 1$ million
  - = 1.5 million bytes
- Savings = $2$ million bytes - 1.5 million bytes
  - = $500,000$ bytes

Example 6.12
Find the effects of:

- **BFCHG** $S004 (D5 : D6)
- **BFEXTU** $S004 (2 : 4), D5
- **BFINS** D4, (A0) (D5 : D6)
- **BFZERO** $S004 (D6 : 4), D6

Assume the following data prior to execution of each of the above instructions:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>0000 0001</td>
</tr>
<tr>
<td>D5</td>
<td>1111 1110</td>
</tr>
<tr>
<td>D6</td>
<td>0000 0001</td>
</tr>
<tr>
<td>CCR</td>
<td>0000 0000</td>
</tr>
<tr>
<td>$S004</td>
<td>1111 1111</td>
</tr>
</tbody>
</table>

Register contents are given in hex, CCR and memory contents in binary, and offset to the left of memory in decimal.

**Solutions**

- **BFCHG** $S004 (D5 : D6)
  
<table>
<thead>
<tr>
<th>CCR</th>
<th>X N Z V C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S004</td>
<td>1111 1111</td>
</tr>
</tbody>
</table>

- **BFEXTU** $S004 (2 : 4), D5
  
<table>
<thead>
<tr>
<th>CCR</th>
<th>X N Z V C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S004</td>
<td>1111 1111</td>
</tr>
</tbody>
</table>

- **BFINS** D4, (A0) (D5 : D6)
  
<table>
<thead>
<tr>
<th>CCR</th>
<th>X N Z V C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S004</td>
<td>1111 1111</td>
</tr>
</tbody>
</table>

- **BFZERO** $S004 (D6 : 4), D6
  
<table>
<thead>
<tr>
<th>CCR</th>
<th>X N Z V C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S004</td>
<td>1111 1111</td>
</tr>
</tbody>
</table>
6.7.6 PACK AND UNPACK INSTRUCTIONS

Table 6.12 lists the details of PACK and UNPK instructions. Both instructions have three operands and are unsized. They do not affect the condition codes. The PACK instruction converts two unpacked BCD digits to two packed BCD digits. The UNPK instruction reverses the process and converts two packed BCD digits to the unpacked BCD digits. Immediate data can be added to convert numbers from one code to another. That is, these instructions can be used to translate codes such as ASCII or EBCDIC to BCD and vice versa.

**TABLE 6.12**

<table>
<thead>
<tr>
<th>Operand size</th>
<th>Operation</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PACK 16 → 8</td>
<td>Unpacked source + # data → packed destination</td>
<td>PACK -(An), -(An), #&lt;data&gt;</td>
</tr>
<tr>
<td>UNPK 8 → 16</td>
<td>Packed source → unpacked source</td>
<td>UNPK -(An), -(An), #&lt;data&gt;</td>
</tr>
<tr>
<td></td>
<td>unpacked source + # data → unpacked destination</td>
<td>UNPK Dn, Dr, #&lt;data&gt;</td>
</tr>
</tbody>
</table>

Note: Condition codes are not affected.

The PACK and UNPK instructions are useful when an I/O device such as an ASCII keyboard is interfaced to a MC68020-based microcomputer. Data can be entered into the microcomputer via the keyboard in ASCII codes. The PACK instruction can be used with appropriate adjustments to convert these ASCII codes into BCD. Arithmetic operations can be performed inside the microcomputer and the result will be in PACKED BCD. The UNPK instruction can similarly be used with appropriate adjustment to convert packed BCD to ASCII codes.

**Example 6.13**

Find the effects of execution of the following PACK and UNPK instructions:

1. PACK D0, D5, # $0000
2. PACK -(A1), -(A1), # $0000
3. UNPK D4, D6, # $3030
4. UNPK -(A3), -(A2), # $3030

Assume the following data:

<table>
<thead>
<tr>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>D7</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**MEMORY**

<table>
<thead>
<tr>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>D7</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

255.jpg
This instruction with the assumed data converts two packed BCD digits to their equivalent ASCII digits.

### 6.7.7 MULTIPLICATION AND DIVISION INSTRUCTIONS

The MC68020 includes the following signed and unsigned multiplication instructions:

- **MULS.W (EA)**, \(D_n \times 16 \rightarrow 32\), \((EA)16 \times Dn16 \rightarrow Dn32\)
- **MULU**

**MULS.L (EA)**, \(D_1 \times 32 \rightarrow 32\), \((EA) \times D_1 \rightarrow D_1\)

**MULU**

- **MULS.L (EA)**, \(Dh:D1 \times 32 \rightarrow 64\), \((EA) \times Dh:D1 \rightarrow Dh:D1\)
- **MULU**

Holds 32-bit multiplicand before multiplication

Holds low 32 bits of the result after multiplication, upper 32 bits of the result are discarded

Holds high 32 bits of the product after multiplication

Holds 32-bit multiplier before multiplication

(EA) in the above can be all modes except An. The condition codes N, Z, V are affected, C is always cleared to zero, and X is unaffected for both MULS and MULU. For signed multiplication, overflow (V = 1) can only occur for \(32 \times 32\) multiplication producing a 32-bit result if the high-order 32 bits of the product are not the sign extension of the low-order 32 bits. In the case of unsigned multiplication, overflow (V = 1) can occur for \(32 \times 32\) multiplication producing a 32-bit result if the high-order 32 bits of the 64-bit product are not zero.

Both MULS and MULU have a word form and a long-word form. For the word form \((16 \times 16)\) the multiplier and multiplicand are both 16 bits and the result is 32 bits. The result is saved in the destination data register. For \(32 \times 32\) bit, the multiplier and multiplicand are both 32 bits and the result is either 32 bits or 64 bits. When the result is 32 bits for a 32-bit \(\times 32\)-bit operation, the low-order 32 bits of the 64-bit product is provided.

The signed and unsigned division instructions of the MC68020 in-
Consider the following. Source is divisor and destination is dividend. Result (remainder and quotient) is stored in destination.

\[
\text{DIVS.W (EA), Dn} \quad 32/16 = \text{16r:16q}
\]
or

\[
\text{DIVU}
\]

\[
\text{DIVS.L (EA), Dn} \quad 32/32 = \text{32q}\quad \text{(no remainder is provided)}
\]
or

\[
\text{DIVU}
\]

\[
\text{DIVS.L (EA), Dr:Dq} \quad 64/32 = \text{32r:32q}
\]
or

\[
\text{DIVU}
\]

\[
\text{DIVS.L (EA), Dr:Dq} \quad 32/32 = \text{32r:32q}
\]
or

\[
\text{DIVU}
\]

Destination contains dividend and (EA) contains divisor (EA) in the above instructions can use all nodes except An.

The condition codes for either signed or unsigned division are affected as follows: N = 1 if the quotient is negative; N = 0 otherwise. N is undefined if overflow or divide by zero. Z = 1 if the quotient is zero; Z = 0 otherwise. Z is undefined for overflow or divide by zero. V = 1 for division overflow; V = 0 otherwise. X is unaffected.

Division by zero causes a trap. If overflow is detected before completion of the instruction, V is set to one, but the operands are unaffected.

Both signed and unsigned division instructions have a word form and three long-word forms.

For the word form, the destination operand is 32 bits and the source operand is 16 bits. The 32-bit result in Dn contains the 16-bit quotient in the low and the 16-bit remainder in the high word. The sign of the remainder is the same as the sign of the dividend.

\[
\text{For DIVS.L (EA), Dq or DIVU}
\]

Both destination and source operands are 32 bits. The result in Dq contains the 32-bit quotient and the remainder is discarded.

**Example 6.14**

Find the effects of the following multiplication and division instructions:

1. \[
\text{MULS.L} \# \$2, \text{D5 if [D5] = } \text{FFFFFEEF}
\]
2. \[
\text{MULS.L} \# \$2, \text{D5 if [D5] = } \text{FFFFFEEF}
\]
3. \[
\text{MULS.L} \# \$2, \text{D5:D2 if [D5] = } \text{FFFFFEEF}
\]
4. \[
\text{DIVS.L} \# \$2, \text{D5 if [D5] = } \text{FFFFFEEF}
\]
5. \[
\text{DIVS.L} \# \$2, \text{D2:D0 if [D2] = } \text{FFFFFEEF}
\]
6. \[
\text{DIVS.L} \# \$2, \text{D2:D1 if [D1] = } \text{FFFFFEEF}
\]

**Solution**

1. \[
\text{MULS.L} \# \$2, \text{D5, [D5] = } \text{FFFFFEEF}
\]

\[
\text{FFFFFEEF}
\]

\[
100000000
\]

\[
00000000 \text{FFFFFEEF}
\]

\[
\text{Y = 1 Low 32-bit since result in D5 is nonzero}
\]

Therefore, [D5] = FFFFFFEE, N = 0 since the most significant bit of result is 0, Z = 0 since the result is non-zero, V = 1 since the high 32 bits of the 64-bit product is not zero, C = 0 (always), and X = not affected.
2. \texttt{MULS.L} \# \$2, D5 if [D5] = $FFFFFE

\[
\begin{array}{c}
\text{SFFFFFE} \\
\text{10000002}
\end{array}
\]

Therefore, [D5] = $FFFFFE, X = unaffected, C = 0, N = 1, V = 0, and Z = 0.

3. \texttt{MULU.L} \# \$2, D5:2 if [D5] = $2ABC 1800 and [D2] = $FFFFFE

\[
\begin{array}{c}
\text{SFFFFFE} \\
\text{10000002} \\
\text{D5} \\
\text{D2}
\end{array}
\]

N = 0, Z = 0, V = 1 since high 32 bits of the 64-bit product is not zero, C = 0, and X = not affected.

4. \texttt{DIVS.L} \# \$2, D5, D5 if [D5] = $FFFFFE

\[
\begin{array}{c}
\text{SFFFFFE} \\
\text{FFFFFE} \\
\text{FFFFEF}
\end{array}
\]

[D5] = $FFFFFE, X = unaffected, N = 1, Z = 0, V = 0, and C = 0 (always).

5. \texttt{DIVU.L} \# \$2, D2:D0 if [D2] = $FFFF FEFF and [D0] = $FFFF FFFC

\[
\begin{array}{c}
\text{FFFFFE} \\
\text{FFFFFE} \\
\text{FFFFFC} \\
\text{FFFFFC}
\end{array}
\]

6. \texttt{DIVS.L} \# \$2, D6:D1 if [D1] = $0000 1224 and [D5] = $FFFF FFF0

\[
\begin{array}{c}
\text{FFFFFE} \\
\text{FFFFFE} \\
\text{FFFFFE} \\
\text{FFFFFE}
\end{array}
\]

[D6] = $FFFFFE = remainder, [D1] = $FFFFFE = quotient, X = unaffected, Z = 0, N = 1, V = 0, and C = 0 (always).

6.7.8 MC68000 ENHANCED INSTRUCTIONS

The MC68020 includes the enhanced version of the 68000 instructions listed in Table 6.13.

**Table 6.13** Enhanced Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRA label</td>
<td>8, 16, 32</td>
<td>PC + d → PC</td>
</tr>
<tr>
<td>Bcc label</td>
<td>8, 16, 32</td>
<td>If cc is true, then PC + d → PC; else next instruction</td>
</tr>
<tr>
<td>BSR label</td>
<td>8, 16, 32</td>
<td>PC ← (SP); PC + d → PC</td>
</tr>
<tr>
<td>CMPIS #</td>
<td>8, 16, 32</td>
<td>Destination – # data → CCR is affected</td>
</tr>
<tr>
<td>data (EA)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TST.S (EA)</td>
<td>8, 16, 32</td>
<td>Destination – 0 → CCR is affected</td>
</tr>
<tr>
<td>LINKS An, –d</td>
<td>16, 32</td>
<td>An ← (SP); SP ← An, SP + d → SP</td>
</tr>
<tr>
<td>EXTBL Dn</td>
<td>32</td>
<td>Sign extend byte to long word</td>
</tr>
</tbody>
</table>

Note: S can be B, W, L. In addition to 8- and 16-bit signed displacements for BRA, Bcc, BSR like the 68000, the 68020 also allows signed 32-bit displacements. Link is unsigned in 68000. (EA) in CMPIS and TST support all MC68000 modes plus PC relative. Examples are CMPIS.W#2000, (START, PC). In addition to EXT.L and EXTBL, Dn like the 68000, the 68020 also provides EXTBL instruction.
Example 6.15
Write a program in MC68020 assembly language to find the first one in a bit field which is greater than 16 bits and less than or equal to 512 bits. Assume the number of bits to be checked is divisible by 16. If no ones are found, store $0000 0000 in D3. Assume A2 points to start of the array and D2 contains the number of bits in array.

Solution

```
CLRL D3
DIVU $16, D2
SUBQW $1, D2
MOVEQ $16, D5
START BEFFO (A2) (D3:D5), D3
DBNE D2, START
BEQ STOP
CLR L D3
STOP JMP STOP
```

Example 6.16
Write a program in MC68020 assembly language to convert 20 packed BCD digits to their ASCII equivalent and store the result to memory location $F1002004. The data bytes start at $5000.

Solution

```
MOVEA.L # $5000, A5
```

Example 6.17
Write a program in MC68020 assembly language to divide a signed 32-bit number in D9 by a signed 8-bit number in D1 by storing the division result in the following manner:
1. Store 32-bit quotient in D9 and neglect remainder.
2. Store 32-bit remainder in D1 and 32-bit quotient in D9.

Assume dividend and divisor are already in D0 and D1, respectively.

Solution

```
1. EXTB.L D1
DIVS.L D1, D9
STOP JMP STOP
2. EXTB.L D1
```
6.8 MC68020 PINS AND SIGNALS

Figure 6.9a and 6.9b show the MC68020 functional signal groups and pin diagram. Tables 6.14a and 6.14b list these signals along with a summarized description of each, and also the pin assignments.

There are 10 VCC (+5V) and 13 ground pins to distribute the power in order to reduce the noise.

Figure 6.9a MC68020 functional signal groups.
<table>
<thead>
<tr>
<th>Signal name</th>
<th>Mnemonic</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-modify-write cycle</td>
<td>RMCE</td>
<td>Provides an indicator that the current bus cycle is part of an indivisible read-modify-write operation.</td>
</tr>
<tr>
<td>External cycle start</td>
<td>ECS</td>
<td>Provides an indication that a bus cycle is beginning identical operation to that of ECS except that OCS is asserted only during the first bus cycle of an operand transfer.</td>
</tr>
<tr>
<td>Operand cycle start</td>
<td>OCS</td>
<td>Indicates that valid data is to be placed on the data bus by an external device or has been placed on the data bus by the MC68020.</td>
</tr>
<tr>
<td>Address strobe</td>
<td>AS</td>
<td>Indicates that a valid address is on the bus.</td>
</tr>
<tr>
<td>Data strobe</td>
<td>DS</td>
<td>Indicates that valid data is to be placed on the data bus.</td>
</tr>
<tr>
<td>Read/write</td>
<td>R/W</td>
<td>Defines the bus transfer as an MPU read or write.</td>
</tr>
<tr>
<td>Data buffer enable</td>
<td>DBEN</td>
<td>Provides an enable signal for external data buffers.</td>
</tr>
<tr>
<td>Data transfer and size acknowledge</td>
<td>DSACK/DSACK1</td>
<td>Bus response signals that indicate the requested data transfer operation are completed; in addition, these two lines indicate the size of the external data.</td>
</tr>
<tr>
<td>Cache disable</td>
<td>CDIS</td>
<td>Bus port on a cycle-by-cycle basis.</td>
</tr>
<tr>
<td>Interrupt priority level</td>
<td>IPLD-IPL2</td>
<td>Dynamically disables the on-chip cache to assist emulator support.</td>
</tr>
<tr>
<td>Autovector</td>
<td>AVEC</td>
<td>Provides an encoded interrupt level to the processor.</td>
</tr>
<tr>
<td>Interrupt pending</td>
<td>IPEND</td>
<td>Requests an autovector during an interrupt acknowledge cycle.</td>
</tr>
<tr>
<td>Bus request</td>
<td>BR</td>
<td>Indicates that an external device requires bus mastership.</td>
</tr>
<tr>
<td>fBus grant</td>
<td>BG</td>
<td>Indicates that an external device may assume bus mastership.</td>
</tr>
<tr>
<td>Bus grant acknowledge</td>
<td>BCACK</td>
<td>Indicates that an external device has assumed bus mastership.</td>
</tr>
<tr>
<td>Reset</td>
<td>RESET</td>
<td>System reset.</td>
</tr>
<tr>
<td>Halt</td>
<td>HALT</td>
<td>Indicates that the processor should suspend bus activity.</td>
</tr>
<tr>
<td>Bus error</td>
<td>BERR</td>
<td>Indicates an invalid or illegal bus operation is being attempted.</td>
</tr>
<tr>
<td>Clock</td>
<td>CLK</td>
<td>Clock input to the processor.</td>
</tr>
<tr>
<td>Power supply</td>
<td>VCC</td>
<td>+5 volt ± 5% power supply.</td>
</tr>
<tr>
<td>Ground</td>
<td>GND</td>
<td>Ground connection.</td>
</tr>
</tbody>
</table>
### TABLE 6.14b
MC68020 Pin Assignment

<table>
<thead>
<tr>
<th>Pin number</th>
<th>Function</th>
<th>Pin number</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>BGACK</td>
<td>C9</td>
<td>A16</td>
</tr>
<tr>
<td>A2</td>
<td>A1</td>
<td>C10</td>
<td>A12</td>
</tr>
<tr>
<td>A3</td>
<td>A31</td>
<td>C11</td>
<td>A9</td>
</tr>
<tr>
<td>A4</td>
<td>A28</td>
<td>C12</td>
<td>A7</td>
</tr>
<tr>
<td>A5</td>
<td>A26</td>
<td>C13</td>
<td>A5</td>
</tr>
<tr>
<td>A6</td>
<td>A23</td>
<td>D1</td>
<td>Vcc</td>
</tr>
<tr>
<td>A7</td>
<td>A22</td>
<td>D2</td>
<td>Vcc</td>
</tr>
<tr>
<td>A8</td>
<td>A19</td>
<td>D3</td>
<td>Vcc</td>
</tr>
<tr>
<td>A9</td>
<td>Vcc</td>
<td>D4-D11</td>
<td>—</td>
</tr>
<tr>
<td>A10</td>
<td>GND</td>
<td>D12</td>
<td>A4</td>
</tr>
<tr>
<td>A11</td>
<td>A14</td>
<td>D13</td>
<td>A3</td>
</tr>
<tr>
<td>A12</td>
<td>A11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A13</td>
<td>A8</td>
<td>E1</td>
<td>FC0</td>
</tr>
<tr>
<td>B1</td>
<td>GND</td>
<td>E2</td>
<td>RMC</td>
</tr>
<tr>
<td>B2</td>
<td>BG</td>
<td>E3</td>
<td>Vcc</td>
</tr>
<tr>
<td>B3</td>
<td>BR</td>
<td>E12</td>
<td>A2</td>
</tr>
<tr>
<td>B4</td>
<td>B5</td>
<td>H12</td>
<td>A2</td>
</tr>
<tr>
<td>B5</td>
<td>A27</td>
<td>H13</td>
<td>OCS</td>
</tr>
<tr>
<td>B6</td>
<td>A24</td>
<td>F1</td>
<td>SIZ0</td>
</tr>
<tr>
<td>B7</td>
<td>A20</td>
<td>F2</td>
<td>FC2</td>
</tr>
<tr>
<td>B8</td>
<td>A18</td>
<td>F3</td>
<td>FC1</td>
</tr>
<tr>
<td>B9</td>
<td>GND</td>
<td>F12</td>
<td>GND</td>
</tr>
<tr>
<td>B10</td>
<td>H15</td>
<td>F13</td>
<td>IPEND</td>
</tr>
<tr>
<td>B11</td>
<td>A13</td>
<td>G1</td>
<td>FC5</td>
</tr>
<tr>
<td>B12</td>
<td>A10</td>
<td>G2</td>
<td>SIZT</td>
</tr>
<tr>
<td>B13</td>
<td>A6</td>
<td>G3</td>
<td>DBEN</td>
</tr>
<tr>
<td>C1</td>
<td>RESET</td>
<td>H1</td>
<td>Vcc</td>
</tr>
<tr>
<td>C2</td>
<td>CLOCK</td>
<td>H12</td>
<td>Vcc</td>
</tr>
<tr>
<td>C3</td>
<td>GND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>A0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td>A29</td>
<td>H2</td>
<td>AVEC</td>
</tr>
<tr>
<td>C6</td>
<td>A25</td>
<td>H3</td>
<td>DSACK0</td>
</tr>
<tr>
<td>C7</td>
<td>A21</td>
<td>H12</td>
<td>IPL2</td>
</tr>
<tr>
<td>C8</td>
<td>A17</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Vcc and GND pins are separated into three groups to provide individual power supply connections for the address bus buffers, data bus buffers, and all other output buffers and internal logic.

### TABLE 6.14b (continued)
MC68020 Pin Assignment

<table>
<thead>
<tr>
<th>Pin number</th>
<th>Function</th>
<th>Pin number</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>H13</td>
<td>GND</td>
<td>M1</td>
<td>D5</td>
</tr>
<tr>
<td>J1</td>
<td>DSACK1</td>
<td>M2</td>
<td>D29</td>
</tr>
<tr>
<td>J2</td>
<td>BERR</td>
<td>M3</td>
<td>D26</td>
</tr>
<tr>
<td>J3</td>
<td>GND</td>
<td>M4</td>
<td>D24</td>
</tr>
<tr>
<td>J12</td>
<td>IPL0</td>
<td>M5</td>
<td>D21</td>
</tr>
<tr>
<td>J13</td>
<td>IPL1</td>
<td>M6</td>
<td>D18</td>
</tr>
<tr>
<td>K1</td>
<td>GND</td>
<td>M7</td>
<td>D16</td>
</tr>
<tr>
<td>K2</td>
<td>HALT</td>
<td>M8</td>
<td>Vcc</td>
</tr>
<tr>
<td>K3</td>
<td>GND</td>
<td>M9</td>
<td>D13</td>
</tr>
<tr>
<td>K12</td>
<td>D1</td>
<td>M10</td>
<td>D10</td>
</tr>
<tr>
<td>K13</td>
<td>D0</td>
<td>M11</td>
<td>D6</td>
</tr>
<tr>
<td>L1</td>
<td>A5</td>
<td>M12</td>
<td>D3</td>
</tr>
<tr>
<td>L2</td>
<td>R/W</td>
<td>M13</td>
<td>D4</td>
</tr>
<tr>
<td>L3</td>
<td>D30</td>
<td>N1</td>
<td>D31</td>
</tr>
<tr>
<td>L4</td>
<td>D27</td>
<td>N2</td>
<td>D28</td>
</tr>
<tr>
<td>L5</td>
<td>D23</td>
<td>N3</td>
<td>D25</td>
</tr>
<tr>
<td>L6</td>
<td>D19</td>
<td>N4</td>
<td>D22</td>
</tr>
<tr>
<td>L7</td>
<td>GND</td>
<td>N5</td>
<td>D20</td>
</tr>
<tr>
<td>L8</td>
<td>D15</td>
<td>N6</td>
<td>D17</td>
</tr>
<tr>
<td>L9</td>
<td>D11</td>
<td>N7</td>
<td>GND</td>
</tr>
<tr>
<td>L10</td>
<td>D7</td>
<td>N8</td>
<td>Vcc</td>
</tr>
<tr>
<td>L11</td>
<td>GND</td>
<td>N9</td>
<td>D14</td>
</tr>
<tr>
<td>L12</td>
<td>D3</td>
<td>N10</td>
<td>D12</td>
</tr>
<tr>
<td>L13</td>
<td>D2</td>
<td>N11</td>
<td>D9</td>
</tr>
</tbody>
</table>

#### Group
- **Address bus**: A9, D3, A10, B9, C3, F12
- **Data bus**: M8, N8, N13, L7, L11, N7, K3
- **Logic**: D1, D2, E3, G11, G13, G12, H13, J3, K1
- **Clock**: B1
Both the 32-bit address (A0-A31) and data (D0-D31) buses are nonmultiplexed. Like the MC68000, the three function code signals FC2, FC1, and FC0 identify the processor state (supervisor or user) and the address space of the bus cycle currently being executed as follows:

<table>
<thead>
<tr>
<th>FC2</th>
<th>FC1</th>
<th>FC0</th>
<th>Cycle type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(Undefined, reserved)*</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>User data space</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>User program space</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>(Undefined, reserved)*</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Supervisor program space</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Supervisor data space</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>CPU space</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CPU space</td>
</tr>
</tbody>
</table>

* Address space 3 is reserved for user definition, while 0 and 4 are reserved for future use by Motorola.

Note that in MC68000, FC2, FC1, FC0 = 111 indicates interrupt acknowledge cycle. In the MC68020, this means CPU space cycle. In this cycle, by decoding the address lines A19-A16, the MC68020 can perform various types of functions such as coprocessor communication, breakpoint acknowledge, interrupt acknowledge, and module operations as follows:

<table>
<thead>
<tr>
<th>A19</th>
<th>A17</th>
<th>A18</th>
<th>A16</th>
<th>Function performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Breakpoint acknowledge</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Module operations</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Coprocessor communication</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Interrupt acknowledge</td>
</tr>
</tbody>
</table>

Note that A19, A18, A17, A16 = 0011, to 1110, is reversed by Motorola. In the coprocessor communication CPU space cycle, the MC68020 determines the coprocessor type by decoding A15-A13 as follows:

<table>
<thead>
<tr>
<th>A15</th>
<th>A14</th>
<th>A13</th>
<th>Coprocessor type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>MC68851 paged memory management unit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>MC68881 floating-point coprocessor</td>
</tr>
</tbody>
</table>

The MC68020 offers a feature called dynamic bus sizing which enables designers to use 8- and 16-bit memory and I/O devices without sacrificing system performance.

The key elements used to implement dynamic bus sizing are the data multiplexer, the SIZE output (SIZE0 and SIZE1 pins), and the DSACK0 and DSACK1 inputs. The MC68020 uses these signals dynamically to interface to the various-sized devices (8-, 16-, or 32-bit) on a cycle-by-cycle basis. For example, if the MC68020 executes an instruction that reads a long-word operand, it will attempt to read all 32 bits during the first bus cycle. The MC68020 always assumes the memory or I/O size to be 32 bits when starting the bus cycle. Hence, it always transfers the maximum amount of data on all bus cycles. If the device responds that it is 32 bits, the processor latches all 32 bits of data and continues to the next operand. If the device responds that it is 16 bits wide, the MC68020 generates two bus cycles, obtaining 16 bits of data each time; an 8-bit transfer is handled similarly, but four bus cycles are required, obtaining 8 bits of data each time. Each device (8-, 16-, or 32-bit) assignment is fixed to particular sections of the data bus to minimize the number of bus cycles needed to transfer data. For example, the 8-bit devices transfer data via D31-D24, the 16-bit devices via D31-D16 pins, and the 32-bit devices via D31-D0 pins.

A routing and duplication multiplexer takes the four byte of 32 bits and routes them to their required positions depending on its bus size, the positioning of bytes is determined by SIZE1, SIZE0, and address outputs A15 and A0 pins.

The MC68020 dynamic bus sizing feature for 8-, 16-, and 32-bit devices is shown in Figure 6.10. The “OP26’s” define the various operand bytes with OPJ as the most significant byte. The multiplexer in the figure routes the four bytes of data (OP0-OP3) via the external data bus. For example, OP0 can be routed to D31-D24 in the normal case, or it can be routed to any other byte in order to support misaligned transfer. Note that alignment and misalignment mean that the MC68020’s data transfer is with an even and odd address, respectively.

In summary, the four signals added to support dynamic bus sizing are DSACK0, DSACK1, SIZE0, and SIZE1. Data transfer and device size acknowledge signals (DSACK0 and DSACK1) are used to terminate the bus cycle and to indicate the external size of the data bus (see Table 6.15). As the MC68020 steps through memory during the data operand transfer process, the two size line outputs (SIZE0 and SIZE1) indicate how many bytes are still to be transferred during a given bus cycle (Table 6.15).

The MC68020 has no restrictions on the alignment of operands in
TABLE 6.15 Dynamic Sizing Control Signals

<table>
<thead>
<tr>
<th>DSACK1</th>
<th>DSACK0</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>H</td>
<td>Insert wait states in current bus cycle</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>Complete — data bus port size is 8 bits</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>Complete — data bus port size is 32 bits</td>
</tr>
</tbody>
</table>

SIZE Output Encodings

<table>
<thead>
<tr>
<th>SIZ1</th>
<th>SIZ0</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Word</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3 bytes</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Long word</td>
</tr>
</tbody>
</table>

Note: To adjust the size of the physical bus interface, external circuits must issue strobe signals to gate data bus buffers. By using data strobe control, external logic can enable the proper section of the data bus.

The multiplexer routes and/or duplicates one or more bytes in the 32-bit data to permit any combination of aligned or misaligned transfers. The remaining number of bytes to be transferred during the second and subsequent cycles if required is defined by the SIZ0 and SIZ1 outputs. The address lines A0 and A1 define the byte position in Figure 6.10. For example, A1A0 = 00, A1A0 = 01, A1A0 = 10, and A1A0 = 11, indicate byte 0 (OP0), byte 1 (OP1), byte 2 (OP2), and byte 3 (OP3), respectively.
of the 32-bit operand, A2-A31 indicate the long-word base address of that portion of the operand to be accessed.

Table 6.16 defines the data path along with SIZE1, SIZE0, A1, and A0 of the MC68020's internal multiplexer to the external data bus D31-D0.

### TABLE 6.16
Internal to External Data Bus Multiplexor

<table>
<thead>
<tr>
<th>Transfer size</th>
<th>SIZE1</th>
<th>SIZE0</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Word</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>3 bytes</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Long word</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: X = don't care; * = byte ignored on read, this byte output on write.

In each cycle, the MC68020 outputs to SIZE1 SIZE0 pins to indicate to the external device the number of bytes remaining to be transferred. The DSACK1 and DSACK0 inputs to the MC68020 from the device terminate the bus cycle and in the subsequent cycles (if required) indicate the device size. The address pins A1 and A0 outputs to the device from the MC68020 indicate which data pins are to be used in the data transfer. For example, an 8-bit device always transfers data via D31-D16 pins. However, in the first cycle, if the address is even (A1A0 = 00, or 10), 16-bit data transfer takes place using D31-D16 pins with the data byte addressed by the odd address via D31-D16 pins. On the other hand, if the starting address is odd (A1A0 = 01, or 11) for a 16-bit device, only a byte is transferred in the first cycle via D31-D16 pins. For a 32-bit device, in the first cycle, if A1A0 = 00, all 32 bits are transferred via D0-D31 pins; if A1A0 = 01, three bytes are transferred via D31-D24 pins; if A1A0 = 10, two bytes are transferred via D31-D24 pins; if A1A0 = 11, only one byte is transferred via D7-D0 pins.

The MC68020 always starts transferring data with the most significant byte first. As an example, consider MOVE.L D2, $5000170. Since the address is even, this is an aligned transfer from the 32-bit data register D3 to an even memory address. In the first bus cycle, the MC68020 does not know the size of the external device and hence outputs all combinations of data on D31-D0 pins, taking into consideration that the device size may be byte, word, or long word. The MC68020 outputs OP0, OP1, OP2, and OP3, respectively, on D31-D24, D23-D16, D15-D8, and D7-D0 pins. If the device is 8-bit, it will take the data OP0 from the D31-D24 pins and write to locations $50000170 in the first cycle. However, by the second cycle, the device asserts DSACK1 and DSACK0 as 10, indicating an 8-bit device; the MC68020 then transfers the remaining 24 bits via D31-D24 in three consecutive cycles. If the device is 32-bit, it obtains data bytes from OP0-OP3 in one cycle. Now, let us consider a 16-bit device. During the first cycle, the MC68020 outputs A1A0 = 00, indicating an aligned transfer, and SIZE1 SIZE0 = 00, indicating 32-bit transfer. Therefore, in the first cycle, the device obtains OP0 and OP1 from D31-D24 and D23-D16, respectively. The device then asserts DSACK1 and DSACK0 as 01, to terminate the cycle and to indicate to the MC68020 that it is a 16-bit device. In the second cycle, the MC68020 outputs the A1A0 as 10, indicating that 16-bit data to be obtained by the device via D31-D16 pins and SIZE1 and SIZE0 as 10, indicating that two more bytes remain to be transferred. The MC68020 places the low two bytes (OP2 and OP3) from register D3 via the multiplexer on D31-D16 pins. The device takes these data and places them into locations $50000172 and $50000173, respectively, by activating DSACK1 and DSACK0 as 01, indicating completion of the cycle.

If [D31] = $03F1 2517, [OP0] = $03, [OP1] = $F1, [OP2] = $25, and [OP3] = $17, then data transfer for MOVE.L D3, $50000170 takes place as shown in the following:
Now, let us consider a misaligned transfer to a 16-bit device. For example, consider MOV.E D4, $60172421. Assume [D4] = $7126E214, that is, OPO = $71, OP1 = 26, OP2 = $E2, OP3 = $14. Now, suppose that the device is 16-bit. In the first cycle, the MC68020 outputs $7126E212 via the multiplexor; the multiplexor places these data on D31-D0 pins considering that the device may be 8-, 16-, or 32-bit as follows:

This is because the device accepts $71 if it is 8-bit via D31-D24 pins, $71 via D24-D16 pins if it is 16-bit, and 24-bit data $7126E212 via D23-D0 pins if it is 32-bit.

For 8-bit and 32-bit devices, four and two cycles are required, respectively, to complete the long-word transfer. Now, let us consider the 16-bit device for this example in detail.

In the first cycle, the MC68020 outputs $71, $7126E214 as 80, indicating a 22-bit transfer, and A1A0 as 01, indicating that a byte transfer is to take place via D23-D16 pins in the first cycle. The memory device obtains $71 from D23-D16 and writes these data to $60172421 and then activates DSACK1 and DSACK0 as 01, indicating that the memory device is 16-bit device. In the second cycle, the MC68020 outputs $7126E214 as 111, indicating that three more bytes remain to be transferred, and A1A0 as 10, indicating a 16-bit transfer is to take place via D31-D16.

The memory device activates DSACK1 and DSACK0 as 01, to write $26E2 to locations $60172422 and $60172423. The MC68020 then terminates the cycle.

Figure 6.12 shows a functional block diagram for MC68020 interfaces to 8-, 16-, and 32-bit memory or I/O devices.

Aligned long-word transfer to 8-, 16-, and 32-bit devices is shown in Figure 6.13.

MC68020 byte addressing is summarized in Figure 6.14.
6.8 MC68020 Pins and Signals

Alignment: LONWORD port – A1 = 0 and A0 = 0 (mod 4) or
WORD port – A0 = 0 (mod 2)

MC68020 REGISTER


ROUTING & Duplication
MUX

CPU DATA PINS

<table>
<thead>
<tr>
<th>BUS</th>
<th>DATA</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-BIT SLAVE</td>
<td>BYTE 8</td>
<td>BYTE 7</td>
<td>BYTE 6</td>
<td>BYTE 5</td>
<td></td>
</tr>
<tr>
<td>16-BIT SLAVE</td>
<td>BYTE 4</td>
<td>BYTE 3</td>
<td>BYTE 2</td>
<td>BYTE 1</td>
<td></td>
</tr>
<tr>
<td>8-BIT SLAVE</td>
<td>BYTE 0</td>
<td>BYTE 1</td>
<td>BYTE 2</td>
<td>BYTE 3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BUS</th>
<th>DATA</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-BIT SLAVE</td>
<td>BYTE 8</td>
<td>BYTE 7</td>
<td>BYTE 6</td>
<td>BYTE 5</td>
<td></td>
</tr>
<tr>
<td>16-BIT SLAVE</td>
<td>BYTE 4</td>
<td>BYTE 3</td>
<td>BYTE 2</td>
<td>BYTE 1</td>
<td></td>
</tr>
<tr>
<td>8-BIT SLAVE</td>
<td>BYTE 0</td>
<td>BYTE 1</td>
<td>BYTE 2</td>
<td>BYTE 3</td>
<td></td>
</tr>
</tbody>
</table>

* Size pins indicate number of bytes remaining to complete the operand transfer.

FIGURE 6.12 MC68020 dynamic bus sizing block diagram.

Figure 6.15 shows misaligned long-word transfer to 8-, 16-, and 32-bit devices.

Now, let us explain the other MC68020 pins.

The ECS (external cycle start) pin is a MC68020 output pin. The MC68020 asserts this pin during the first one half clock of every bus cycle to provide the earliest indication of the start of a bus cycle. The use of ECS must be validated later with A5, since the MC68020 may start an instruction fetch cycle and then assert it if the instruction is found in the cache. In the case of a cache hit, the MC68020 does not assert A5, but provides A31-A0, SIZ1, SIZ0, and FC2-FC0 outputs.

The MC68020 asserts the OCS (operand cycle start) pin only during the first bus cycle of an operand transfer or instruction prefetch.
Longword (32 bit) port

<table>
<thead>
<tr>
<th>A31 - A0</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZX = # of bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example:
- If SIZX = 0, then: 1st bus cycle, bytes 2, 3 will be accessed
- If SIZX = 0, then: 2nd bus cycle, bytes 4, 5 will be accessed

Word (16 bit) port

<table>
<thead>
<tr>
<th>A0 = 0</th>
</tr>
</thead>
</table>

Example:
- If SIZX = 0, then: 1st bus cycle, byte 1 will be accessed
- If SIZX = 0, then: 2nd bus cycle, bytes 2, 3 will be accessed
- If SIZX = 0, then: 3rd bus cycle, byte 4 will be accessed

Byte (8 bit) port

| A0 = 0 |

The MC68020 asserts the RMC (read-modify-write) pin to indicate that the current bus operation is an indivisible read-modify-write cycle. RMC should be used as a bus lock to ensure integrity of instructions which use read-modify-write operations such as TAS and CAS.

In a read cycle, the MC68020 asserts the DS (data strobe) pin to indicate that the slave device should drive the bus. During a write cycle, it indicates that the MC68020 has placed valid data on the data bus.

DEN (data buffer enable) is output by the MC68020 which may be used to enable external data buffers.

The CDIS (cache disable) input pin to the MC68020 dynamically disables the cache when asserted.

The interrupt pending (IPEND) input pin indicates that the value of the IPL2-IPL0 pins is higher than the current I21110 in 8R or that a nonmaskable interrupt has been recognized.

The MC68020 AVEC input is activated by an external device to service an interrupt vector interrupt. The AVEC has the same function as the MC68000 VPA.

The functions of the other signals such as AS, R-W, IPL2-IPL0, ER, BL, and BGACK are similar to those of the MC68000.
The MC68020 system control pins are functionally similar to those of the MC68000. However, there are some minor differences. For example, for hardware reset, RESET and HALT pins need not be asserted simultaneously. Therefore, unlike the MC68000, RESET and HALT pins are not tied together in the MC68020 system. RESET and HALT pins are bi-directional, open-drain (external pull-up resistances are required), and their functions are independent.

When HALT input is asserted by an external device, the following activities take place:

- All control signals become inactive.
- Address lines, R/W line, and function code lines remain driven with last bus cycle information.
- All bus activities stop after current bus cycle completion.

Assertion of HALT stops only external bus activities and the processor execution continues. That is, the MC68020 can continue with instruction execution internally if cache hits occur and if the external bus is not required.

The MC68020 asserts the HALT output for double bus fault. The BERR input pin, when asserted by an external device, causes the bus cycle to be aborted and strobos negated. If the BERR is asserted during operand read or write (not prefetch), exception processing occurs immediately. The BERR pin can typically be used to indicate a nonresponding device (no DASACK received from the device), vector acquisition failure, illegal access determined by memory management unit hardware such as access fault (protected memory scheme), and page fault (virtual memory system).

Figure 6.16 shows the MC68020 reset characteristics.

The RESET signal is a bi-directional signal. The RESET pin, when asserted by an external circuit for a minimum of 620 clock periods, resets the entire system including the MC68020. Upon hardware reset, the MC68020 completes any active bus cycle in an orderly manner and then performs the following:

- Reads the 32-bit contents of address $00000000 and loads it into ISP (contents of $00000000 to most significant byte of ISP and so on)
- Reads the 32-bit contents of address $00000004 into PC (contents of $00000000 to most significant byte of PC and so on)
- Sets I21110 bits of SR to 111, sets S-bit in SR to 1, and clears T1, T0, M bits in SR
- Clears VBR to $00000000

### Figure 6.16 MC68020 reset characteristics (higher level exception)

- Clears the cache enable bit in the CACR
- All other registers are unaffected by hardware reset
When the RESET instruction is executed, the MC68020 asserts the RESET pin for 512 clock cycles and the processor resets all the external devices connected to the RESET pin. Software reset does not affect any internal registers.

Figure 6.17 shows a MC68020 reset circuit.

The Motorola MC3456 contains a dual timing circuit. The MC3456 uses an external resistor-capacitor network as its timing elements. Like the MC1455 timer used in the 68000 reset circuit, the MC3456 includes comparators and an R-S flip-flop. From the MC3456 data sheet, the RC values connected at the TRG input of the MC3456 will make output OP HIGH for $T = 1.1 R C$, seconds, where $R = 1$ Mohm, and $C =$ capacitor connected at $TSH = 1$ mF, and $C =$ capacitor connected at $TSH = 0.47$ mF. This means that OP will be HIGH for 517 mics (1.1 m * 0.47 mF), the OP will then go back to LOW state. Therefore, the output of the inverter (connected at OP) will be LOW for 517 mics.

The push button connected to the input of the debouncing circuit, when not activated, will generate HIGH at the top input of AND gate #2 and LOW at the top input of AND gate #1 (HIGH is grounded by the switch as shown in Figure 6.17).

Since a NAND gate generates a HIGH with one of the inputs as LOW, the output of NAND gate #1 will be HIGH. This means that both inputs of NAND gate #2 will be HIGH. Therefore, the output of NAND gate #2 will be LOW. This LOW level is inverted and connected to a WIRED-OR circuit along with the inverted OP of the MC3456 at the MC68020 reset pin. The output of the debouncing circuit will be HIGH when the push button is activated. For example, activation of the push button will generate a HIGH at the top input of NAND gate #1 and a LOW at the bottom input of NAND gate #2. The AND gate #2 will generate a HIGH at its output. Therefore, the inverted output will be LOW which is presented at the WIRED-OR circuit of the 68020 reset pin. A LOW will be provided at the 68020 reset pin only when both inputs are LOW, that is, when the push button is activated and inverted OP of MC3456 is LOW.

The MC3456 timer will keep this reset pin signal LOW for 517 mics. As mentioned before, the 68020 requires the reset pin to stay LOW for at least 520 clock cycles. For a 60 ns (16-MHz) clock, the 68020 must then be LOW for at least 31.2 ns (520 * 60 ns). Since the reset circuit of Figure 6.17 outputs a LOW for 517 mics (>31.2 ns) upon activation of the push button, the 68020 reset pin will be asserted properly.

Example 6.18

Determine the number of bus cycles, bytes written to memory (in Hex), and signal levels of A1, A0, SIZE1, and SIZE0 pins that would occur when the instruction MOVEL D1, (A0) with [D1] = $5012$ $6124$ and $[A0] =$ $2000$ $2103$ is executed by the MC68020.

Assume
1. 32-bit memory
2. 16-bit memory
3. 8-bit memory

Indicate the bus cycles in which UCS is asserted.
Motorola MC68020

Chapter 6

6.8 MC68020 Pins and Signals

Example 6.19

Determine the contents of PC, SR, MSP, ISP after MC68020 hardware reset. Assume 32-bit memory with the following data prior to the reset:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>S5000134</td>
</tr>
<tr>
<td>00000004</td>
<td>S72153614</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSP</td>
<td>S27140124</td>
</tr>
<tr>
<td>ISP</td>
<td>S6171420</td>
</tr>
<tr>
<td>PC</td>
<td>S552061271</td>
</tr>
<tr>
<td>SR</td>
<td>S00201</td>
</tr>
</tbody>
</table>

Solution

After hardware reset, the following are memory and register contents:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>S5000134</td>
</tr>
<tr>
<td>00000004</td>
<td>S72153614</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSP</td>
<td>S27140124</td>
</tr>
<tr>
<td>ISP</td>
<td>S5000134</td>
</tr>
<tr>
<td>PC</td>
<td>S72153614</td>
</tr>
<tr>
<td>SR</td>
<td>S2701</td>
</tr>
</tbody>
</table>

Note that SR = $2701 = 01111011010 

T1 T0 S M 12 E D K W Z V C

271.jpg
Therefore, T1/TO = 00, S = 1, M = 0, and 121110 = 111. Other bits are unaffected.

6.9 MC68020 TIMING DIAGRAMS

The MC68020 always activates all data lines. The MC68020 can perform either synchronous or asynchronous operation. Synchronous operation permits interfacing the devices which use the MC68020 clock to generate DSACKX and other asynchronous inputs. The asynchronous input setup and hold times must be satisfied for the assertion or negation of these inputs. The MC68020 then guarantees recognition of these signals on the current falling edge of the clock.

On the other hand, asynchronous operation provides clock frequency independence for generating DSACKX and other asynchronous inputs. This operation requires utilization of only the bus handshake signals (AS, DS, DSACKX, BERR, and HALT), in asynchronous operation. AS indicates the beginning of a bus cycle and DS validates data in a write cycle. The SIZ1, SIZ0, A1, and A0 signals are decoded to generate strobe signals. These strobes indicate which data bytes are to be used in transfer. The memory or I/O devices then place data on the right portion of the data bus for a read cycle or latch data in a write cycle. The selected device finally activates the DSACKX lines according to the device size to terminate the cycle. If no DSACKX is received by the MC68020 or the access is invalid, the external device can assert BERR to abort or BERR and HALT to retry the bus cycle. There is no limit on the time from assertion of AS to the assertion of DSACKX, since the MC68020 keeps inserting wait states in increments of one cycle until DSACKX is recognized by the processor.

For synchronization, the MC68020 uses a time delay to sample an external asynchronous input for high or low and then synchronizes this input to the clock.

Figure 6.18 shows an example of synchronization and recognition of asynchronous inputs.

Note that for all inputs, there is a sample window of 20 ns during which the MC68020 latches the input level. In order to guarantee the recognition of a certain level on a particular falling edge of the clock, the input level must be held stable throughout this sample window of 20 ns. If an input changes during the sample window, the level recognized by the MC68020 is unknown or illegal. One exception to this rule is the delayed assertion of BERR where the signal must be stable through the window or the MC68020 may exhibit erratic behavior.

Note that if the BERR is asserted during an instruction prefetch, the MC68020 delays bus error exception processing until the faulted data are required for execution. Bus error checking will take place for faulty access if change in program flow such as branching occurs, since the faulty data are not required. Also, after satisfying the setup and hold times, all input signals must meet certain protocols. For example, when DSACKX is
asserted it must remain asserted until AS is negated. Figures 6.19a and b show timing of DSACKX input recognition and the MC68020's reading of data satisfying the required protocol.

In the timing diagrams of Motorola's 68020 manuals, parameter 47 (47a and 47b) provides the asynchronous input setup time of 20 ns. All numbers circled in the timing diagrams are the timing parameters provided in Motorola manuals.

In Figure 6.19b (synchronous operation), assertion of DSACKX is recognized on the falling edge of S2; the MC68020 latches valid data on the falling edge of S4. For asynchronous operation, data are latched 50 ns (parameter 31) after assertion of DSACKX. If DSACKX or ERR is not asserted by the external device during the 20 ns window of the falling edge of S2, the 68020 enters wait states until one of these input signals is asserted. A minimum of three clock cycles is required for a read operation. DSACKX remains asserted until AS negation is satisfied in Figure 6.19a.

Figure 6.20a shows asynchronous bus cycle timing along with various parameters. Figures 6.20b, c, and d show typical MC68020 read and write timing diagrams (general form) along with their AC specifications. Note that in Figures 6.20b and c signals such as SIZ2, SIZ2, DSACKX, D0-D31, A1, and A0, which precisely distinguish data transfers between 8-, 16-, and 32-bit devices, are kept in general form.

**6.10 EXCEPTION PROCESSING**

The MC68020 exceptions are functionally similar to those of the MC68000 with some minor variations. The MC68020 exceptions can be generated by external or internal causes. Externally-generated exceptions include interrupts, bus errors, reset, and coprocessor-detected errors. Internally-generated exceptions are caused by certain instructions, address errors, tracing, and breakpoints. Instructions that may cause internal exceptions as part of their instruction execution are CHK, CHK2, CALLM, RTM, RTE, DIV, and all variations of the TRAP instruction. In addition, illegal instructions, privilege violations, and coprocessor violations cause exceptions. Table 6.17 lists the priority and characteristics of all MC68020 exceptions.

MC68020 exception processing is similar in concept to the MC68000 with some minor variations. In the MC68020 exception processing occurs in four steps and varies according to the cause of the exception. The four steps are summarized below:

1. During the first step, an internal copy is made of the SR, and the SR is set for exception processing. This means that the status register enters the supervisor state and tracing is disabled.
FIGURE 6.20a Asynchronous bus cycle timing. All times in nanoseconds.

FIGURE 6.20b Read cycle timing diagram. Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted. The voltage swing through this range should start outside and pass through the range such that the rise or fall will be linear between 0.8 and 2.0 volts.
Exception Processing

6.10 Exception Processing

FIGURE 6.29c Write cycle timing diagram. Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted. The voltage swing through this range should start outside and pass through the range such that the rise or fall will be linear between 0.8 and 2.0 volts. ‘Note 5’ refers to Figure 6.28a.

FIGURE 6.29d Read and write cycle specifications.
<table>
<thead>
<tr>
<th>Table 6.17 Exception Priorities and Recognition Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception priorities</td>
</tr>
<tr>
<td>Group 0</td>
</tr>
<tr>
<td>Group 1</td>
</tr>
<tr>
<td>Group 2</td>
</tr>
<tr>
<td>Group 3</td>
</tr>
</tbody>
</table>
### TABLE 6.17 (continued)
**Exception Priorities and Recognition Times**

<table>
<thead>
<tr>
<th>Group</th>
<th>Exception</th>
<th>Time of recognition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>cp post-instruction</td>
<td>End of instruction cycle</td>
</tr>
<tr>
<td>1</td>
<td>Trace</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Interrupt</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Halt and bus arbitration are recognized at end of bus cycle. 0.0 is highest priority; 4.2 is lowest.

### TABLE 6.18
**MC68020 Exception Vector Assignments**

<table>
<thead>
<tr>
<th>Vector number(s)</th>
<th>Vector offset</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>SP</td>
</tr>
<tr>
<td>1</td>
<td>004</td>
<td>SP</td>
</tr>
<tr>
<td>2</td>
<td>008</td>
<td>SD</td>
</tr>
<tr>
<td>3</td>
<td>00C</td>
<td>SD</td>
</tr>
<tr>
<td>4</td>
<td>010</td>
<td>SD</td>
</tr>
<tr>
<td>5</td>
<td>014</td>
<td>SD</td>
</tr>
<tr>
<td>6</td>
<td>018</td>
<td>SD</td>
</tr>
<tr>
<td>7</td>
<td>01C</td>
<td>SD</td>
</tr>
<tr>
<td>8</td>
<td>020</td>
<td>SD</td>
</tr>
<tr>
<td>9</td>
<td>024</td>
<td>SD</td>
</tr>
<tr>
<td>10</td>
<td>028</td>
<td>SD</td>
</tr>
<tr>
<td>11</td>
<td>02C</td>
<td>SD</td>
</tr>
<tr>
<td>12</td>
<td>030</td>
<td>SD</td>
</tr>
<tr>
<td>13</td>
<td>034</td>
<td>SD</td>
</tr>
<tr>
<td>14</td>
<td>038</td>
<td>SD</td>
</tr>
<tr>
<td>15</td>
<td>03C</td>
<td>SD</td>
</tr>
<tr>
<td>16</td>
<td>040</td>
<td>SD</td>
</tr>
</tbody>
</table>

**Note:** SP = supervisor program space; SD = supervisor data space.

### Table 6.18 (continued)
**MC68020 Exception Vector Assignments**

<table>
<thead>
<tr>
<th>Vector number(s)</th>
<th>Hex</th>
<th>Space</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>068</td>
<td>SD</td>
<td>Level 2 interrupt auto vector</td>
</tr>
<tr>
<td>27</td>
<td>06C</td>
<td>SD</td>
<td>Level 3 interrupt auto vector</td>
</tr>
<tr>
<td>28</td>
<td>070</td>
<td>SD</td>
<td>Level 4 interrupt auto vector</td>
</tr>
<tr>
<td>29</td>
<td>074</td>
<td>SD</td>
<td>Level 5 interrupt auto vector</td>
</tr>
<tr>
<td>30</td>
<td>078</td>
<td>SD</td>
<td>Level 6 interrupt auto vector</td>
</tr>
<tr>
<td>31</td>
<td>07C</td>
<td>SD</td>
<td>Level 7 interrupt auto vector</td>
</tr>
<tr>
<td>32</td>
<td>080</td>
<td>SD</td>
<td>TRAP #0-15 instruction vectors</td>
</tr>
<tr>
<td>33</td>
<td>0BC</td>
<td>SD</td>
<td>FPCP branch or set on unordered condition</td>
</tr>
<tr>
<td>34</td>
<td>0CD</td>
<td>SD</td>
<td>FPCP inexact result</td>
</tr>
<tr>
<td>35</td>
<td>0CE</td>
<td>SD</td>
<td>FPCP divide by zero</td>
</tr>
<tr>
<td>36</td>
<td>0DF</td>
<td>SD</td>
<td>FPCP underflow</td>
</tr>
<tr>
<td>37</td>
<td>0E0</td>
<td>SD</td>
<td>FPCP operand error</td>
</tr>
<tr>
<td>38</td>
<td>0E4</td>
<td>SD</td>
<td>FPCP overflow</td>
</tr>
<tr>
<td>39</td>
<td>0E8</td>
<td>SD</td>
<td>FPCP signal exception</td>
</tr>
<tr>
<td>40</td>
<td>0EC</td>
<td>SD</td>
<td>Unassigned, reserved</td>
</tr>
<tr>
<td>41</td>
<td>0F0</td>
<td>SD</td>
<td>PMMU configuration</td>
</tr>
<tr>
<td>42</td>
<td>0F4</td>
<td>SD</td>
<td>PMMU illegal operation</td>
</tr>
<tr>
<td>43</td>
<td>0FA</td>
<td>SD</td>
<td>PMMU access level violation</td>
</tr>
<tr>
<td>44</td>
<td>0FC</td>
<td>SD</td>
<td>Unassigned, reserved</td>
</tr>
<tr>
<td>45</td>
<td>100</td>
<td>SD</td>
<td>User defined vectors (192)</td>
</tr>
</tbody>
</table>

**Note:** Figure 6.21 shows an overview of exception processing. Exception processing saves certain information on the top of the supervisor stack. This information is called the exception stack frame. The
Exception Processing

The MC68020 provides six different stack frames. These frames include the following:

1. Normal four-word stack frame
2. Normal six-word stack frame
3. Four-word throwaway stack frame
4. Coprocessor misinstruction exception stack frame
5. Short bus fault stack frame
6. Long bus fault stack frame

The normal four-word stack frame (FORMAT S0) is generated by interrupts, format errors (when instructions such as RTE find an invalid stack format code), TRAPn instruction, illegal instructions, emulator traps, privilege violations, and coprocessor preinstruction exceptions.

The throwaway four-word stack frame (FORMAT S1) is generated on the interrupt stack during exception processing for an interrupt when a transition from the master to the interrupt state occurs.

The normal six-word stack frame (FORMAT S2) is generated by instruction-related exceptions such as CHK, CHX, CPTRAPcc, TRAPcc, TRAPV, trace, and zero divide. The fifth and sixth stack words contain the instruction address that caused the exception. The other four words have the same format as the general stack format.
word stack frame; the other four words contain contents of internal registers.

The short bus cycle fault stack frame (FORMAT $A$) is generated whenever a bus cycle fault is detected and the MC68020 identifies it as an instruction boundary and it can use this reduced version of the bus fault stack frame. This stack frame contains 16 words and words 6-16 include information such as internal registers and instruction pipes.

The long bus cycle fault stack frame (FORMAT $B$) is generated whenever the MC68020 identifies bus cycle fault and finds that it is not on an instruction boundary. This stack frame contains 46 words. Words 6-46 include information such as internal registers, instruction pipes, and data input/output buffers.

The MC68020 provides the concept of two supervisor stacks pointed to by MSP and ISP. The M-bit (when $S = 1$) determines the active supervisor stack pointer. The MC68020 accesses MSP when $S = 1$, $M = 0$. The MSP can be used for program traps and other exceptions, while the ISP can be used for interrupts. The use of two supervisor stacks allows isolation of user processes or tasks and asynchronous supervisor I/O tasks. IT2, IT1, IT0, AVEC, and IE2 pins are used as the MC68020 hardware interrupt control signals (Figure 6.24). The MC68020 supports seven levels of prioritized interrupts encoded by using IT2, IT1, IT0 pins (Table 6.19).

The coprocessor midinstruction exception stack frame (FORMAT $S$) is generated by coprocessor-related operations such as protocol violation. This frame contains ten words. The first six words are the same as the six-
TABLE 6.19
MC68020 Interrupt Priority

<table>
<thead>
<tr>
<th>Requested interrupt level</th>
<th>State of pins IPL2 IPL1 IPL0</th>
<th>Mask level required for recognition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (no interrupt)</td>
<td>hi hi hi</td>
<td>N/A</td>
</tr>
<tr>
<td>1</td>
<td>hi hi lo</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>hi lo hi</td>
<td>1 or lower</td>
</tr>
<tr>
<td>3</td>
<td>hi lo lo</td>
<td>2 or lower</td>
</tr>
<tr>
<td>4</td>
<td>lo hi hi</td>
<td>3 or lower</td>
</tr>
<tr>
<td>5</td>
<td>lo hi lo</td>
<td>4 or lower</td>
</tr>
<tr>
<td>6</td>
<td>lo lo hi</td>
<td>5 or lower</td>
</tr>
<tr>
<td>7</td>
<td>lo lo lo</td>
<td>Not masked</td>
</tr>
</tbody>
</table>

- Interrupts allowed only above mask level.
- Level 7 may not be masked out — becomes NMI.
- Levels applied to the pins are inverted with respect to corresponding interrupt mask level.

In Figure 6.24, when an interrupting priority level 1 through 6 is requested, the MC68020 compares the interrupt level to the interrupt mask to determine whether the interrupt should be processed. Interrupt recognized as valid does not force immediate exception processing; a valid interrupt causes IPEND to be asserted, signaling to external devices that the MC68020 has an interrupt pending. Exception processing for a pending interrupt that begins at the next instruction boundary of a higher priority exception is not also currently valid. The DESKEW logic in Figure 6.24 continuously samples the IPL3-IPL0 pins on every falling edge of the clock, but despews or latches an interrupt request when it remains at the same level for two consecutive falling edges of the input clock. Figure 6.25 gives an example of the MC68020 interrupt despew logic.

Whenever the processor reaches an instruction execution boundary, it checks for a pending interrupt. If it finds one, the MC68020 begins an exception processing and executes an interrupt acknowledge cycle (IACK) with FC2 FC1 FC0 = 111, and A19 A18 A17 A16 = 1111. The MC68020 basic hardware interrupt sequence is shown in Figure 6.26a. Figures 6.26b and c show the interrupt acknowledge flowchart and timing diagram. Before the interrupt acknowledge cycle is completed, the MC68020 must receive either AVEC, DSACKX, or BERR otherwise it will execute wait states until one of these input pins is activated externally.

![Image](280.jpg)
interrupt logic. Finally, if the SERR pin is asserted, the interrupt is considered to be spurious and the MC68020 assigns the appropriate vector number for handling this.

### 6.11 MC68020 MEMORY AND I/O INTERFACES

In this section, simplified block diagrams illustrating interfacing of memory chips (such as 2716 and 6116) and I/O chips (68220) will be provided. Finally, an A/D converter will be interfaced to the MC68020-based microcomputer using both autovector and nonautovector techniques. These block diagrams are self-explanatory.

Figure 6.28 shows interfacing of MC68020 to a 2716 EPROM and a 6116 RAM. Only pertinent signals are shown. Table 6.20 shows the memory selection scheme.

The delay circuit in Figure 6.28 delays DSACK0 by an appropriate
amount of time (determined by the access times of 2716 and 6116) so that data will be accessed when they are valid. For example, the access time of 2716 is 450 ns. The MC68020 samples DSACKR at the falling edge of S2 (two cycles; see Figure 6.20b). For an 8-bit device, DSACK1 and DSACK0 must have the values 1 and 0, respectively. In Figure 6.28, DSACK1 = 1 and DSACK0 = 0 after two cycles with no wait states.

If DSACK0 is asserted by the external devices within two cycles and the 68020 recognizes this at the falling edge of S2, then the 68020 latches data at the falling edge of S4 (three clock cycles).

For 16.67 MHz (60 ns per cycle), the 68020 latches data at 60 ns * 3 = 180 ns with no wait states. If DSACKX is not recognized at the falling edge of S2, the MC689020 inserts wait states.

Since the 6116 has an access time of 120 ns, the 68020 is required to insert wait states. This can be explained as follows: the 68020 generates AS and DS LOW after 62 ns (one cycle until the falling edge of S0 plus parameter 9 in Figure 6.20b) from the start of the bus cycle. The 68020 recognizes this at the falling edge of S2. The 68020 latches invalid data at the falling edge of S4 (180 ns). Since AS is used to provide CE for the...
6116, time delay between the falling edge of \( \overline{A} \)S and the falling edge of \( S \)1 is approximately 70 ns (parameter 14, 125 ns \( -30 \) ns \( = \) 70 ns), which is smaller than the 6116 access time of 110 ns. Therefore, the 6116 will not be ready with valid data unless 6116 CE (used for DSACK0 recognition) is asserted at and recognized at the falling edge of \( S \)4 (70 ns) and data are latched at the falling edge of \( S \)6 (60 ns \( +70 \) ns \( = \) 130 ns from \( A \)S going LOW).

Therefore, the 6116 CE (output of the inverter) must be delayed by, say, 50 ns (during \( S \)4 and before the falling edge of \( S \)4) so that DSACK0 is asserted and recognized at the falling edge of \( S \)4 and data are latched at the falling edge of \( S \)6.

As far as the EPROM is concerned, the 2716 has an access time of 450 ns and will not work with the 16.67 MHz 68020 unless the DSACK0 in Figure 6.28 is delayed. DS is used to enable the 2716 and also to assert DSACK0. \( \overline{A} \)S and \( \overline{A} \)S have almost identical timing characteristics (Figure 6.29b). This means that DS goes to LOW after 63 ns from the start of the bus cycle. In order to satisfy the timing requirements, time delay between the DS going LOW and data latched must be greater than 450 ns. As with the \( \overline{A} \)S signal, time delay between DS LOW and the falling edge of \( S \)4 is 70 ns. DS must be delayed by about 430 ns from DS going LOW (DS going LOW to the falling edge of \( S \)4 plus 6 cycles \( = \) 70 ns \( +360 \) ns), so that DSACK0 is asserted and recognized at the falling edge of \( S \)4, and then the 6820 will latch data at the falling edge of \( S \)4 (490 ns from DS going LOW). This will provide enough time for the 2716 to valid data on the data bus.

Figure 6.29a shows a simplified block diagram of the 68020 interface to the 68230 for I/O ports.

![Figure 6.29a](image)

**FIGURE 6.29a** MC68020/68230 block diagram.

In Figure 6.29b the MC68020 always reads data from or writes data to the 8-bit device containing both odd and even addresses via D31-D24. For a 16-bit device, the MC68020 communicates via D31-D24 pins for even addresses and D23-D16 for odd addresses. Therefore, the 8-bit data lines of the even memory byte chip must be connected to D31-D24 pins; the 8-bit data lines of the odd memory byte chip must be connected to D23-D16 pins. For long-word transfer, four 2716s are required, each containing a byte, OP0 through OP3. OP0 is transferred via D31-D24, OP1 via D23-D16, OP2 via D15-D8, and OP3 via D0-D7 pins of the MC68020.

The autovector and nonautovector features of the MC68020 interrupts are illustrated in Figures 6.30a and b. Note that in Figure 6.30b A19A18A17A16 \( = \) 1111 means interrupt acknowledge. A3A2A1 has interrupt level 001 (inverted IPEL).

![Figure 6.30a](image)

**FIGURE 6.30a** MC68020/AD interface using autovector 1.
6.12 MC68020 SYSTEM DESIGN

The following MC68020 system design will use a 128-KB, 32-bit wide memory and 8-bit parallel I/O port. The memory system is partitioned into four unique address space encodings: user data, user program, supervisor data, and supervisor program. This design uses RAMs for memory accesses and EPROMs for program memory accesses. Each address has 32 KB of memory available for use. Data I/O port space is appended to both user and supervisor data spaces (see MEMORY MAP); this is done by decoding the user/supervisor data space and address line A18 signal state.

The 32-bit-wide system memory consists of 4-byte-wide memories, each connected to its associated portion of the system data bus (D24-D31, D16-D23, D8-D15, and D0-D7). To manipulate this memory configuration, 32-bit data bus control logic is incorporated to generate byte addresses (DBBE44, DBBE43, DBBE32, and DBBE11) (Table 6.21).
The byte strobe state table shows the necessary individual byte strobe states as dictated by the MC68020's size (SIZ1, SIZ0) and address offset (A2, A0) encodings.

Karnaugh Maps (Table 6.22) for each data strobe signal have been created to identify the logic required to implement its state table requirement. The logic created for each data strobe is then combined into a complete 32-bit control logic schematic and connected to the memory structure as shown in the system hardware schematic diagram (Figure 6.31).

The system hardware design also identifies the required interconnections between the MC68020 MPU, the 74LS138 address space decoder, the CY7C198 user/supervisor data RAMs, the NM27C256 user/supervisor program EPROMs, the 32-bit port control logic, and the MC68230 parallel I/O interface.

Since each memory is 32 K x 8, only address lines A0-A14 are connected. The 74LS138 selects memory banks to enable, as dictated by the decoder FG2-FC0 signals. Control logic generated data strobes (DBBE4-DBBE1) select which byte-wide portion of the data bus to activate. The 8-bit parallel I/O interface (MC68230) provides three bidirectional 8-bit ports as well as asynchronous handshake signals necessary for communication protocols.

TABLE 6.22
K Maps for Strobe Signals for 32-Bit Devices

![Image of K Maps for Strobe Signals for 32-Bit Devices]
TABLE 6.22 (continued)
K Maps for Strobe Signals for 32-Bit Devices

![Diagram of K Maps for Strobe Signals for 32-Bit Devices]

D8BE44 = SIZE1 + SIZE2 + A1 * A0 + SIZE1 * A1 + SIZE1 * SIZE2 + A0

D8BE4 = SIZE1 * SIZE2 * A1 * A0 + SIZE1 * A1 + SIZE1 * SIZE2 + A0

K-Maps
6.1 Find the contents of 68020 registers that are affected and the condition codes after execution of

i) ADD 5 D2, D3
ii) ADD W D5, D6
iii) ADDA, Z A2, A4

Assume the following data prior to execution of each of the above instructions:

\[
\begin{align*}
[D2] &= \$01F46281 \\
[D3] &= \$01D01110 \\
[D5] &= \$00008210 \\
[D6] &= \$00001010 \\
[A2] &= \$71240010 \\
[A4] &= \$21040100
\end{align*}
\]

6.2 i) How many ALUs does the 68020 have? Comment on the purpose of each.
ii) What is the purpose of the 68020 32-bit barrel shifter?

6.3 a) Summarize the basic differences between the 68000 and 68020.
b) Discuss the differences between 68000 and 68020 debug capabilities implemented in their status registers. Will the instructions listed below cause trace or change of flow:

i) MOVE SR, D5
ii) TRAP EQ, START

when Z = 1?

6.4 Determine the number of bus cycles, bytes written to memory (in Hex), and signal levels of A1, A0, SIZ, and SIZ0 pins that would occur when the following 68020 instruction
Chapter 6

6.7 A subroutine in the supervisor mode is required to read a parameter from the stack configuration given below:

\[
\begin{array}{c|c|c|c|c}
A7 & 00002000 & \rightarrow & c & 2 \\
A6 & 00002000 & & 9 & 0 \\
A5 & 00003000 & & 8 & 0 \\
A4 & 00003500 & \text{PARAMETER} & & \\
\end{array}
\]

Write a 68000 instruction sequence to read the parameter into D5 and then write the equivalent 68020 instruction. Assume that the A7 and the offset of the parameter (6) are known.

6.8 The 68000 instruction sequence below searches a table of 10,32-bit elements for a match. The address register A5 points to element 0, D3 contains the length (10,32) to be searched, and D2 contains the number to be matched. Find the 68020 single instruction which can replace lines 3, 4, and 5.

<table>
<thead>
<tr>
<th>Line</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MOVE.B #10, D3</td>
</tr>
<tr>
<td>2</td>
<td>SUBQ.B #1, D3</td>
</tr>
<tr>
<td>3</td>
<td>START MOVE D3, D5</td>
</tr>
<tr>
<td>4</td>
<td>ASL.L #2, D5</td>
</tr>
<tr>
<td>5</td>
<td>CMP.L 0 (A5, D5.L), D2</td>
</tr>
<tr>
<td>6</td>
<td>DBH.Q D3, START</td>
</tr>
</tbody>
</table>

6.9 Find the contents of D1, D2, A4, CCR, and the memory locations after execution of the following 68020 instructions:

i) BFINSTS $5000 (8:16), D4
ii) BFINST D2, A4 (D1:D4)
iii) BFINST $5000 (D1:10)
Questions and Problems

Assume the following data prior to execution of CHK2:

\[ \begin{align*}
[A1] &= 0000 \quad F200 \\
\end{align*} \]

6.12 Fill in the missing hex values for the following 68020 instructions:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Label</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0200 \quad 0200$</td>
<td>$60 \quad -$</td>
<td>START 0</td>
<td>BRAW START2</td>
</tr>
<tr>
<td>$0200 \quad 0206$</td>
<td>$60 \quad -$</td>
<td>START 1</td>
<td>BRAW START10</td>
</tr>
<tr>
<td>$0200 \quad 020F$</td>
<td>$-$</td>
<td>$-$</td>
<td>START 2</td>
</tr>
</tbody>
</table>

6.13 Identify the following 68020 instructions as valid or invalid. Comment if an instruction is invalid.

i) BSET (A0) (-2:5), D7
ii) DIVS D5, D5
iii) CHK.B D2, (A1)

6.14 Determine the values of Z and C flags after execution of each of the following 68020 instructions:

i) CHK2.W (A5), D3
ii) CMP2.L $2001$, A5

Assume the following data prior to execution of each of the instructions:

Assume the following data prior to execution of each of the instructions:

\[ \begin{align*}
[D1] &= 0000 \quad 0004 \\
[D4] &= 0000 \quad 3000 \\
[D5] &= 1234 \quad 5678 \\
[A4] &= 0000 \quad 5000 \\
\end{align*} \]

6.10 Find the 68020 condition codes after execution of CMP2.W (A2), D7. Determine the range of valid values. Indicate whether the comparison is signed or unsigned. Also, indicate the register values along with upper and lower bounds on the following:

Assume the following data prior to execution of CMP2:

\[ \begin{align*}
[D7] &= SF271 \quad 1020 \\
\end{align*} \]

6.11 Find the 68020 condition codes and also determine if an exception occurs due to execution of

\[ \text{CHK2.W} \quad 85002, \quad A1 \]

Motorola MC68020

Chapter 6

Assume the following data prior to execution of each of the above instructions:

\[ \begin{align*}
[D1] &= 0000 \quad 0004 \\
[D4] &= 0000 \quad 3000 \\
[D5] &= 1234 \quad 5678 \\
[A4] &= 0000 \quad 5000 \\
\end{align*} \]
6.15 Write a 8020 assembly language program to test the 10 bit adder. Assume 2 adders a and b are inputs, and the output is the sum at 3000.

6.16 Write a 8020 assembly language program to test the 10 bit subtractor. Assume 2 inputs a and b, and the output is the difference at 4000.

6.17 What are the minimum times for a read cycle and write cycle for a 68020 processor?

6.18 What are the functions of 8020 A, B, C, and D? (Refer to the manufacturer's data sheet for a detailed description.)

6.19 Determine the functions of 8020 A, B, C, and D. (Refer to the manufacturer's data sheet for a detailed description.)

6.20 Determine the length of time the data has been valid during a read cycle. Assume the data is available at 3000.

6.21 Determine the length of time the data has been valid during a write cycle. Assume the data is available at 3000.

6.22 What happens when the 8020 is not a valid memory address? What happens when the 8020 is not a valid memory address?

6.23 Draw a logic circuit to interface a 4K EPROM, 4K RAM, and 2 K RAM using the 8276, 8176, and 84270 to 8020. Determine memory size.

6.24 What are the basic assumptions that must be made to use the 8020 in a microprocessor system?
Chapter 7

MC68020 — ADVANCED TOPICS

This chapter describes the advanced topics associated with the Motorola MC68020. Topics include the 68020 advanced instructions (such as CAS/CAS2 and CALLM/RTM instructions), cache memory, and coprocessors.

7.1 68020 ADVANCED INSTRUCTIONS

This section provides a detailed description of the 68020 advanced instructions including BKPT, CAS/CAS2, TAS, CALLM/RTM, and coprocessor instructions.

7.1.1 BREAKPOINT INSTRUCTION

A breakpoint is a debugging tool that allows the programmer to check or pass over an entire section of a program. Execution of a breakpoint usually results in exception processing. Hence, the programmer can use any of the TRAP vectors as breakpoints. Also, any of the seven interrupt levels can be used by external hardware to cause a breakpoint.

Included with the MC68020 is the breakpoint instruction (BKPT), used to support the program breakpoint function for debug monitors and real-time hardware in-circuit emulators. Real-time hardware emulators require acknowledgment of breakpoint occurrences to transfer control (i.e., trigger map switching logic) from a task to a monitor. External system hardware must be present if the breakpoint function is to be employed. The breakpoint instruction uses a special breakpoint acknowledgment bus cycle to notify external hardware that an illegal instruction (i.e., breakpoint) has been encountered. This breakpoint hardware has been included in Motorola's Paged Memory Management Unit, MC68851. The notation for the break-
point instruction is BKPT # data. The immediate data ranges from 0 to 7 indicate eight separate breakpoint op codes, $4848$ through $484F$. These breakpoint op codes are contained in the dynamically relocatable Vector Base Register (VBR).

Upon detecting one of the breakpoint op codes, the processor informs the external hardware of its occurrence by issuing a breakpoint bus cycle to access CPU space \#0. The breakpoint instruction format of Figure 7.1 shows how CPU space \#0 is encoded during a breakpoint instruction. The three-bit breakpoint identifier on address lines A2, A3, and A4 differentiates between the eight possible breakpoints permitted.

<table>
<thead>
<tr>
<th>FC2</th>
<th>FC1</th>
<th>FC0</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 7.1** BKPT instruction format.

When a BKPT instruction is executed, the MC68020 reads an op word from the CPU space address corresponding to the breakpoint number. If the external hardware terminates the read cycle by asserting the DSACKX signals, then the processor replaces the breakpoint instruction in its internal pipeline with the instruction present on data bus and begins execution with that instruction. No exception processing occurs when the op code is returned along with the appropriate DSACKX signals. The data read from CPU space are the 16-bit op code that was displaced to make room in program memory for the breakpoint instruction. If the breakpoint acknowledge cycle is terminated by the BERR input control signal, then the processor performs exception processing for an illegal instruction by trapping through the illegal instruction vector contained in the vector base table. Execution of the breakpoint instruction does not affect the Condition Code Register (CCR).

The breakpoint instruction flowchart is shown in Figure 7.2.

Example 7.1
Explain the breakpoint operation shown in Figure 7.3.

Solution
In Step 1 the illegal instruction BKPT is inserted into the program flow by external hardware as requested by a user. In this example, the 16-bit op code for ADD.L D2, D3 is replaced with the 16-bit op code for BKPT #4.
In Step 2, the op code ADD.L.D2, D3 is stored by the MC68020 in the CPU space long-word memory location corresponding to breakpoint number five, address $10$.

Step 3 shows how long-word memory locations $006$ to $010$ in CPU space $0$ are used to temporarily store the op codes replaced by breakpoint instructions $0$ through $7$. The displaced op code is stored in the upper 16 bits, and an optional count value can be loaded into the lower 16 bits through external hardware control.

In Step 4, each time BKPT $\#4$ is executed, the MC68020 accesses CPU space $9$ location $010$. If the count value (bits $0:15$) is zero, the replaced op code is placed on the data bus (bits $16:31$), the counter is decremented by one, and the appropriate DACK lines are asserted by the external hardware. The MC68020 reads this op word and executes it. If the count value is zero, the 68020 EERR pin can be asserted by external hardware to take exception.

The breakpoint function is summarized in Table 7.1. A hardware implementation to take advantage of the BKPT instruction's looping feature is shown in Figure 7.4. A debug monitor maintains a small amount of breakpoint memory. It is used to hold up to eight replacement op codes and eight counters. When a user wants a breakpoint to be encountered after say "10" passes of the op code, the debugger initiates a breakpoint counter to $10$, replaces the user op code with a breakpoint instruction, saves the user op code in the breakpoint data memory, and executes the user's program. When the breakpoint op code is encountered, the MC68020 generates the breakpoint acknowledge bus cycle. The 74LS138 8-to-8
decoder shown on the right decodes the function codes as all ones (as validated by the AS signal) and asserts its bottom output indicating a CPU bus space cycle. This output enables one of the inputs of the 74LS188 shown on the left. The top enable output is enabled when address lines A16 through A19 are all zero, indicating a type 0 CPU space cycle. This decoder enables one of eight breakpoint counters.

If the counter = 0, then it is decremented by the external hardware and the replacement op code is returned on the data bus to the MC68020 with DBACK asserted. In this case, we started with a count of 10, so it is decremented to 9. When the breakpoint acknowledge cycle occurs on the 11th pass, the counter = 0 and this time bus error ERR signal is asserted by the external hardware. This forces illegal instruction processing and subsequent servicing of the breakpoint. With this type of hardware, operating system support is not required until the count has been exhausted. This reduces the operating system overhead significantly. The MC68020 breakpoint instruction flow diagram is shown in Figure 7.5.

![Figure 7.5 Breakpoint Instruction Flowchart](image)

### 7.1.2 CALL MODULE/RETURN FROM MODULE INSTRUCTIONS

For large programs, flowcharting does not provide an efficient software design tool. The flowcharts, however, can assist the programmer in dividing a large program into subprograms called modules (typically 20 to 50 lines). The task of dividing a large program into modules is called modular programming. Typical problems of modular programming include how to modularize a large program and then how to combine the individual modules.

One of the advantages of modular programming includes ease in writing, debugging, and testing a module rather than a large program.

One of the disadvantages of modular programming is the difficulty in modularizing a program efficiently. Some guidelines for modularization include limiting the amount of information shared by modules and controlling the access of one module to another.

As an example of modular programming, consider that a line printer and a keyboard are to be interfaced to a microcomputer. The entire task can be divided into four sections:

- Printing module
- Data reception module
- Keyboard module
- Data transmission module

Modular programming is supported by the MC68020 call module (CALLM) and the return from module (RTM) instructions.

The CALLM instruction creates a module stack frame on the stack (similar to an interrupt), stores the module state in that frame, and points to the address of a module descriptor (the effective address) which contains control information for the entry into the called module.

The CALLM instruction leads the processor with the data provided by the module descriptor. Thus the CALLM instruction does not directly access the program module. Rather, it indirectly calls the routine via the module descriptor whose contents are maintained by the operating system. The module descriptor can be thought of as a gateway through which the calling program must gain access. The module being called can be thought of as a subroutine that is to act upon the arguments passed to it.

The MC68020 concept of a module is different from a subroutine in the following ways. Stack parameter passing and argument removal are handled automatically. The operating system can dynamically define the starting address of the called module, and the called module can be granted access rights that could be at a higher privilege level than those of the program calling the module. Access rights define what portions of memory can be accessed. Since all MC68020 I/O devices are memory-mapped, access to specific areas of memory are required to perform I/O functions. In the MC68020, access rights are tied to the concept of address spaces, where the function codes (FC2-FC0) define one of eight memory space access levels. The module instructions support a hierarchical resource protection mechanism which allows external hardware to implement up to 256 privileged access levels. This is accomplished through the use of an 8-bit access level number contained in each module descriptor.
When a module is called, a request can be made to change the current access level. The external hardware could be designed to interpret this access level field for level checking or level modification purposes. During execution of the module, the processor can generate CPU space I bus cycles (i.e., FC2FC, FC0 = 001 and A19 – A16 = 0001), called access level control cycles to which the external hardware responds. Motorola's Paged Memory Management Unit (PMMU) has this hardware built in.

The CALLM instruction syntax is

```
CALLM & data, (RA)
```

To use this instruction, an immediate data value (data 0-255) must be specified to indicate the number of bytes of argument parameters to be passed to the called module. The effective address (EA) that points to the external module descriptor must also be included. The program containing the CALLM instruction must define a RAM storage area for the module descriptor to reside in. The address of the module descriptor is known to the operating system at "run time". It is the operating system that loads the starting address of the library routine to be called into the module descriptor.

The return from module (RTM) instruction is the complement of the CALLM instruction. The RTM syntax is RTM Rn.

Register Rn (address or data) represents the module data area pointer. The RTM instruction recovers the previous module state from the stack frame and returns program execution in the calling module. The processor state (program counter, status word) comes from the previously stacked data. The operation of the module instructions is shown in Table 7.2.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand size</th>
<th>Operation</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALLM</td>
<td>None</td>
<td>Save current module state on stack; load new module data area descriptor</td>
<td>CALLM data, (RA)</td>
</tr>
<tr>
<td>RTM</td>
<td>None</td>
<td>Relaxed saved module state in stack frame; place module data area pointer in Rn</td>
<td>RTM Rn</td>
</tr>
</tbody>
</table>

Figure 7.6 shows an overview of the CALLM and RTM operation. When the CALLM instruction executes, it creates a module call stack frame and saves the module state in that frame.

![Diagram of CALLM and RTM operation](image)

**FIGURE 7.6 Overview of the CALLM and RTM operation.**

It then loads the new module state from the module descriptor. The start address fetched from the module descriptor points to the module entry word of the library routine. The second word of the routine is its first op word. The last instruction of the routine is the RTM instruction.

The RTM instruction reloads the saved module state in the created stack frame, and user program execution continues with the next instruction.

During the execution of the module instructions, three components are utilized: the module descriptor, the module stack frame, and the called module. The location of the module descriptor is determined by the programmer through a descriptor declaration which ultimately must be known to the Operating System (OS). It is the operating system that maintains the contents of the module descriptor. The module descriptor format and its relation to the called module is shown in Figure 7.7.

The CALLM instruction interprets the module descriptor's information to determine how the arguments are to be passed, the type of...
descriptor, whether access level changes will be made, and the location of the called module, data area, and optional stack area. The first long word of the descriptor contains control information used during the execution of the CALLM instruction. The remaining locations contain the descriptor pointers which may be loaded into three processor registers by the CALLM instruction after first being pushed to the stack frame. The descriptor OPT field specifies how arguments are to be passed to the called module and is summarized below:

Options

- **00**:
  - The called module expects to find arguments from the calling module on the stack stack just below the module stack frame; for stack pointer changes, the MC68020 copies the arguments from old to new stack.

- **10**:
  - Called module accesses arguments through an indirect pointer on the stack of the called module; the MC68020 puts the calling module stack pointer value in the module stack frame; hence, the arguments are not copied.

All other options cause a format exception. The descriptor type field specifies the type of descriptor and is summarized below:

<table>
<thead>
<tr>
<th>Type</th>
<th>Type of descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No change in module access rights or stack pointer; the called module builds its stack frame on the top of the stack used by the calling module.</td>
</tr>
<tr>
<td>01</td>
<td>Possible change in module access rights, or stack pointer; the called module is allowed to have a stack area independent of the caller's stack area; if the stack changes, arguments may be copied automatically.</td>
</tr>
</tbody>
</table>

All other types cause a format exception. The CALLM instruction flow diagrams for type 0 and type 1 forms are shown in Figure 7.8. The descriptor's access level field (used only for type 01 descriptors) is passed to external hardware to change or monitor access control. The descriptor's module entry word pointer specifies the entry address of the called module. The first word at the entry address of the called module shown in Figure 7.8 specifies which data or address register is to be saved in the module stack frame, and subsequently loaded with the module.
Thus, the CALLM instruction reads this word to determine which register is to be used as the data area pointer. The descriptor's module data area pointer field contains the address of the called module data area. The second word of the called module is the module routine's first op word. The last instruction of the called module is the RTM Rn instruction. The RTM Rn instruction loads the register specified (Rn) with the saved stack module data area pointer value and restores the program counter and optionally the stack pointer. The Rn register is usually chosen to match the register specified by the module entry word. The RTM instructions also read the argument byte count and increments the stack pointer by the number of arguments specified in the CALLM instructions, which remove the argument from the stack.

The module stack frame is shown in Figure 7.10.
This stack frame is constructed by the CALLM instruction and is removed by the RTM instruction. The first two long words contain information passed by the CALLM instruction to the RTM instruction. The contents of the Condition Code Register (CCR), as well as the OPT and type fields, are copied and saved by the CALLM instruction and removed by the RTM instruction. The access level contains control information saved from external hardware during CALLM instruction execution. The argument count field is set by the CALLM instruction and is used to remove arguments from the stack of the calling module by the RTM instruction. The module descriptor pointer contains the address of the descriptor used during the module call. The program counter is the saved address of the instruction following the CALLM instruction. The remaining locations (saved module data area pointer, saved stack pointer, optional argument) contain information to be restored on return to the calling module.

Type 01 module descriptor indicates a possible request to change access levels. While processing a type 01 descriptor, the module instructions communicate with external access control hardware (like an MMU via accesses in CPU space. Figure 7.11 shows the MMU Access Level Control (ALC) bus registers used as an address map for CPU space accesses.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Address Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>CALM</td>
<td>0000000000</td>
</tr>
<tr>
<td>01</td>
<td>STOP</td>
<td>0000000001</td>
</tr>
<tr>
<td>02</td>
<td>DM</td>
<td>0000000010</td>
</tr>
<tr>
<td>03</td>
<td>DM</td>
<td>0000000011</td>
</tr>
<tr>
<td>04</td>
<td>JUMP</td>
<td>0000000100</td>
</tr>
<tr>
<td>05</td>
<td>JUMP</td>
<td>0000000101</td>
</tr>
<tr>
<td>06</td>
<td>JUMP</td>
<td>0000001000</td>
</tr>
<tr>
<td>07</td>
<td>JUMP</td>
<td>0000001001</td>
</tr>
<tr>
<td>08</td>
<td>JUMP</td>
<td>0000001010</td>
</tr>
<tr>
<td>09</td>
<td>JUMP</td>
<td>0000001011</td>
</tr>
<tr>
<td>0A</td>
<td>JUMP</td>
<td>0000010000</td>
</tr>
<tr>
<td>0B</td>
<td>JUMP</td>
<td>0000010001</td>
</tr>
<tr>
<td>0C</td>
<td>JUMP</td>
<td>0000010010</td>
</tr>
<tr>
<td>0D</td>
<td>JUMP</td>
<td>0000010011</td>
</tr>
<tr>
<td>0E</td>
<td>JUMP</td>
<td>0000010100</td>
</tr>
<tr>
<td>0F</td>
<td>JUMP</td>
<td>0000010101</td>
</tr>
<tr>
<td>10</td>
<td>JUMP</td>
<td>0000010110</td>
</tr>
<tr>
<td>11</td>
<td>JUMP</td>
<td>0000010111</td>
</tr>
<tr>
<td>12</td>
<td>JUMP</td>
<td>0000100000</td>
</tr>
<tr>
<td>13</td>
<td>JUMP</td>
<td>0000100001</td>
</tr>
<tr>
<td>14</td>
<td>JUMP</td>
<td>0000100010</td>
</tr>
<tr>
<td>15</td>
<td>JUMP</td>
<td>0000100011</td>
</tr>
<tr>
<td>16</td>
<td>JUMP</td>
<td>0000100100</td>
</tr>
<tr>
<td>17</td>
<td>JUMP</td>
<td>0000100101</td>
</tr>
<tr>
<td>18</td>
<td>JUMP</td>
<td>0000100110</td>
</tr>
<tr>
<td>19</td>
<td>JUMP</td>
<td>0000100111</td>
</tr>
<tr>
<td>1A</td>
<td>JUMP</td>
<td>0001000000</td>
</tr>
<tr>
<td>1B</td>
<td>JUMP</td>
<td>0001000001</td>
</tr>
<tr>
<td>1C</td>
<td>JUMP</td>
<td>0001000010</td>
</tr>
<tr>
<td>1D</td>
<td>JUMP</td>
<td>0001000011</td>
</tr>
<tr>
<td>1E</td>
<td>JUMP</td>
<td>0001000100</td>
</tr>
<tr>
<td>1F</td>
<td>JUMP</td>
<td>0001000101</td>
</tr>
<tr>
<td>20</td>
<td>JUMP</td>
<td>0001000110</td>
</tr>
<tr>
<td>21</td>
<td>JUMP</td>
<td>0001000111</td>
</tr>
<tr>
<td>22</td>
<td>JUMP</td>
<td>0010000000</td>
</tr>
<tr>
<td>23</td>
<td>JUMP</td>
<td>0010000001</td>
</tr>
<tr>
<td>24</td>
<td>JUMP</td>
<td>0010000010</td>
</tr>
<tr>
<td>25</td>
<td>JUMP</td>
<td>0010000011</td>
</tr>
<tr>
<td>26</td>
<td>JUMP</td>
<td>0010000100</td>
</tr>
<tr>
<td>27</td>
<td>JUMP</td>
<td>0010000101</td>
</tr>
<tr>
<td>28</td>
<td>JUMP</td>
<td>0010000110</td>
</tr>
<tr>
<td>29</td>
<td>JUMP</td>
<td>0010000111</td>
</tr>
<tr>
<td>2A</td>
<td>JUMP</td>
<td>0010010000</td>
</tr>
<tr>
<td>2B</td>
<td>JUMP</td>
<td>0010010001</td>
</tr>
<tr>
<td>2C</td>
<td>JUMP</td>
<td>0010010010</td>
</tr>
<tr>
<td>2D</td>
<td>JUMP</td>
<td>0010010011</td>
</tr>
<tr>
<td>2E</td>
<td>JUMP</td>
<td>0010010100</td>
</tr>
<tr>
<td>2F</td>
<td>JUMP</td>
<td>0010010101</td>
</tr>
<tr>
<td>30</td>
<td>JUMP</td>
<td>0010010110</td>
</tr>
<tr>
<td>31</td>
<td>JUMP</td>
<td>0010010111</td>
</tr>
</tbody>
</table>

The Current Access Level Register (CAL) specifies the access level rights of the currently executing module. The increase access register (IAL) and the decrease access register (DAL) are the registers through which the MC68020 requests increased or decreased access rights, respectively.

The access status register ("STATUS") in the MMU allows the MC68020 to consult external hardware (68851 MMU) as to the legitimacy of intended access level transitions. Table 7.3 gives the valid code values of the access status register.

**Table 7.3**

<table>
<thead>
<tr>
<th>Value</th>
<th>Validity</th>
<th>Processor action</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Invalid</td>
<td>Format error</td>
</tr>
<tr>
<td>01</td>
<td>Valid</td>
<td>No change in access rights</td>
</tr>
<tr>
<td>02-03</td>
<td>Valid</td>
<td>Change access rights with no change of stack pointer</td>
</tr>
<tr>
<td>04-07</td>
<td>Valid</td>
<td>Change access rights and change stack pointer</td>
</tr>
<tr>
<td>Other</td>
<td>Undefined</td>
<td>Undefined — take format error exception</td>
</tr>
</tbody>
</table>

**Example 7.2**

Write an MC68020 instruction sequence to illustrate the use of CALLM and RTM instructions.

**Solution**

```
CALL
PEA (EA)
PEA (EA)
CALL #8, (A0)
MOVE.L 818(SP), A1
MOVE.L 81C(SP), A2
RTM
```

*Figure 7.7*
7.1.3 COMPARE, SWAP, AND CAS INSTRUCTIONS

The MC68020 compare and swap (CAS and CAS2) instructions provide support for multitasking and multiprocessing. The compare and swap instructions are used when several processors must communicate through a common block of memory, when globally shared data structures (such as counters, stack pointers, or queue pointers) must be securely updated, or when multiple bus cycles may be required. A typical application is a counting semaphore (shared incrementer). Another application of the compare and swap instructions would be to manipulate pointers for system stacks and queues that use linked lists when a new item (element) is inserted or an existing element is deleted.

When data structures are manipulated in a multiprocessor environment and a processor updates elements of the data structure can be corrupted by another processor between the time an element is obtained for update and the time the element is written back to memory. To ensure the integrity of an update operation, the CAS and CAS2 instructions check the original value of each element before it is updated, to make sure that it matches the values that were originally fetched. If the values match, elements in the data structure are updated. If not (that is, if another processor changed the value of the element since the value was fetched), the CAS and CAS2 compare registers receive the new values instead of updating them. The new values are then used to repeat the compare and swap process until the update operation is successful.

The CAS instruction uses two registers (DC and Du) and the address (EA) of the globally shared operand variable (or pointer) to be protected. CAS operates on byte, word, and long word operands. The assembly level programming notation for the CAS instruction is:

```
CAS.S (.B or .W or .L) DC, Du, (EA)
```

The CAS instruction first compares the old “fetched” starting pointer in register (DC) with the present starting pointer in the variable’s original location (EA) to see if another task accessed the variable pointer and changed it while the first task was using it; if the two compared pointer values DC and (EA) are equal (that is, pointer contents (EA) remain unchanged by interrupting tasks), then CAS passes the updated pointer value (located in Du) to the destination operand (EA) and sets the equal condition code flag in the CCR to Z = 1.

If the two compared values are not equal the (EA) pointer has been changed by an interrupting task, then CAS copies the new changed pointer value contents of (EA) to register (DC) and clears the equal condition code flag in the CCR to Z = 0.

A condensed version of the CAS operation is shown below:

```
CAS DC, Du, (EA)
```

1. (EA) → DC → ea
2. IF (EA) = DC
   THEN Du → ea
   ELSE (EA) → DC

Next, application of the CAS for queue insertion will be discussed.

Figure 7.12a shows how to insert a new entry in a queue using the MOVE instruction.

![Figure 7.12a](image)

In a single user/single task environment the above can be accomplished by the following instruction sequence:

```
MOVE.L HEAD, (NEXT, A1)
MOVE.L A1, HEAD
```

But in a multitask/multitask environment, the above instruction sequence may not accomplish the task. For example, more than one user may attempt to insert a new entry in an existing queue. Suppose that after user 1 executes MOVE.L HEAD, (NEXT, A1), user 2 executes the instruction sequence:

```
MOVE.L HEAD, (NEXT, A1)
MOVE.L A1, HEAD
```

and inserts user 2 new entry in the existing queue before user 1 gets to...
MOVE.L A1, HEAD. This situation is depicted in Figures 7.12b and 7.12c. This situation can be avoided by using the CAS instruction. User 2 entry gets lost if the MOVE instruction is used for insertion. In Figure 7.12c, the HEAD (known to user 1 at the start) gets changed between the time user 1 established the forward link and the time user 1 updated HEAD.

![Diagram](attachment:image.png)

**Figure 7.12b** Two users attempting to insert a new entry into an existing queue.

**Figure 7.12c** Insertion of entries into an existing queue in a multiuser environment using MOVE (situation to avoid).

The following instruction sequence uses CAS to insert a new entry in a queue in a multiuser/multiprocess environment.

MOVE.L HEAD, D0 Capture current HEAD
MOVE.L A1, D1 Need value in D1 for CAS

**Figure 7.12d** Inserting an element in a queue using CAS without intervention.

```assembly
MOVE.L D0, (NEXT, A1) Establish forward link
CAS.L D0, D1, HEAD If HEAD unchanged, update
BNE LOOP You have new HEAD in
D0, try again
```

Figures 7.12d and 7.12e illustrate the use of CAS in the above instruction sequence, respectively, in a system with no intervention and simultaneous insertion by multusers.

The CAS instruction cannot be interrupted. This ensures secure updates of variables in a multiprocess system. Executing CAS causes the read-modify-write (RMW) signal to be asserted, which locks the bus. Other bus masters in the system must wait for RMW to be deasserted before they can take control of the bus. The CAS instructions read, compare, and store operations are performed while the bus is locked. This prevents other processors from interfering with the instruction while it performs its compare and swap operation.

To illustrate the read-modify-write operation invoked during execution
of a 32-bit port CAS instruction, the following text will review the asynchronous bus signals involved and show an RMC operational flowchart with a corresponding timing diagram.

Read/Write (R/W) — A high output indicates a read from an external device. A low output indicates a write to an external device.

Read-Modify-Write (RMC) — Asserted indicates that the current bus cycle operation is indivisible and used a bus lock to ensure the integrity of an entire BLC sequence.

External Cycle Start (ECS) — Asserted provides the earliest indication that the 68020 has accessed the cache for the instruction. In case of a cache hit, the 68020 does not assert ECS; otherwise the 68020 asserts ECS and completes the fetch cycle.

Address Strobe (AS) — Assertion indicates that the function codes (FC0-FC2), address lines (A0-A31), bus size indicators (SIZ0-SIZ1), and the read/write (R/W) signal all contain valid information.

Data Strobe (DS) — During a read cycle, assertion indicates the data bus is free for slave device operations during a write cycle, and the MC68020 has placed valid data on the data bus.

Data Transfer and Size Acknowledge (DSACK0, DSACK1) — Assertion indicates that data transfer is complete and indicates port size of the attached external device. During a read cycle, processor recognition of

**Example 7.3**


**Solution**

<table>
<thead>
<tr>
<th>CAS.B D3, D5, (A1) Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3</td>
<td>25125551</td>
</tr>
<tr>
<td>D5</td>
<td>701152652</td>
</tr>
<tr>
<td>A1 - 50000004</td>
<td>MEMORY</td>
</tr>
<tr>
<td>A1 - 50000004</td>
<td>0.8</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>N</td>
<td>0</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
</tr>
<tr>
<td>V</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
</tr>
<tr>
<td>D0</td>
<td>0</td>
</tr>
</tbody>
</table>

DSACKX causes the data to be latched before termination of the bus cycle. During a write cycle, processor recognition of DSACKX causes the bus cycle to be terminated. The DSACKX signals are also used with the SIZE output encoding (SIZ0, SIZ1) and the addressing encoding (A0, A1) to implement the dynamic bus sizing feature.

Data Buffer Enable (DBEN) — Assertion provides an enable to external data bus buffers.

The read-modify-write cycle operational flowchart for the CAS instruction along with its corresponding timing diagram is shown in Figures 7.13 and 7.14.

In a single-processor multitasking interrupt-driven system, the noninterruptible CAS update operations provide security. In a multiprocessor system, an indivisible operand bus cycle (generated by inserting RMC) provides the security mechanism.
The CAS instruction is an extension of the test and set instruction. TAS is primarily used in a multiprocessor environment and is also performed in an indivisible non-atomic manner. In the MC68000 system, AS stays low throughout the execution of the TAS instruction. On the other hand, in the MC68020 system, RMC is output low by the 68020 throughout the execution of the TAS instruction.

Therefore, the main difference between the 68000 and 68020 TAS implementations is that the 68000 uses AS as the bus locking control signal input, while the 68020 RMC pin performs the bus locking control signal input. Other differences include the asynchronous bus control and data transfer signals used by the two MPUs.

Figure 7.14 shows a 68020 shared RAM implementation.

Shared memory bus reservation logic active outputs of the OR gates 1 and 2 indicate that the associated MPU wants exclusive use of the shared RAM. NAND gates 5 and 6 constitute an inverting latch for the outputs of NAND gate 3 and 4.

Now, if MPU-B first requests RAM, the outputs of OR gate 2 and NAND gate 4 force the MPU-A request RAM output of NAND 3 to be its “request denied” state. At the same time, the MPU-B request RAM
output of NAND 4 is inverted and latched on the output of NAND 5. The complementary outputs of latches 5 and 6 select which MPU system bus to connect with the shared memory bus by enabling the appropriate data/address/control bus interface logic. If MPU-A also requests use of the RAM, the outputs of NAND gates 3 and 4 both become active, but the latch does not change its previously recorded MPU-B RAM request state. Thus, the MPU-A RAM request becomes pending upon the removal of the MPU-B request. When this occurs, the changed output of NAND 4 forces the MPU-A RAM request output of NAND 3 to activate the latch into the MPU-A RAM request state. ICs 7, 8, and 9 provide bidirectional read/write data bus control when enabled by the active RAM bus request latch output of NAND gates 5 and 6.

Example 7.4
Write a 68020 instruction sequence for counting a semaphore. That is, the instruction sequence will increment a count in a shared location.

**Solution**

```
CLR.W CNT
JER INIT
AGAIN JSR D0:JOB
MOVE.W CNT, D0
GET a current value of the counter

LOOP MOVE.W D0, D1
ADDQ.W $1, D1
CAS.W D0, D1, CNT
IF the counter value is the same, update it
AND increment it

BNE LOOP
IF not, try again using new counter value in D0

BRA AGAIN
```

Since any system processor can update the counter, the loop guarantees that the variable is updated properly. If a processor increments a shared counter while a second processor is using the counter, the processor using the counter will continue its count sequence using the updated count value.

Instruction CAS2 is identical to CAS except that it can be used to compare and update dual operands within the same indivisible cycle. The CAS2 instruction operates on word or long word operands. The notation for the CAS2 instruction is CAS2.W De1: De2, Du1: Du2, (Rn1):(Rn2).

or

L
With the CAS2 instruction, both comparators must show a match for the contents of the update registers Du1 and Du2 to be stored at the operands' destination addresses in memory pointed to by the registers Rn1 and Rn2. If either comparator fails to match, both destination operands obtained from memory are copied to the compare registers Dc1 and Dc2. The CAS2 instruction memory references to operand destination addresses must be specified using register indirect addressing with either a data or address register used as a pointer to memory.

The condensed version of CAS2 DC1: Dc2 Du1: Du2, (EA1): (EA2) operation is given below:

If (EA1) = Dc1 and (EA2) = Dc2, then
Du1 → (EA1) and Du2 → (EA2);
else (EA1) → Du1 and (EA2) ;→ Du2

Example 7.5
Determine the effects of CAS2 L: D4: D5: D6: D4, (A5): (A6). Assume the following data prior execution of the CAS2:

\[
[D4] = 00000 7000 \\
[D5] = 00000 8000 \\
[D6] = 00000 9000 \\
[A5] = 12345 0000 \\
[A6] = 50000 0200
\]

Solution

<table>
<thead>
<tr>
<th>BEFORE</th>
<th>OPERATION</th>
<th>AFTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGISTERS</td>
<td></td>
<td>REGISTERS</td>
</tr>
<tr>
<td>D4</td>
<td>0 0 0 0 0 0 0</td>
<td>D4</td>
</tr>
<tr>
<td>D5</td>
<td>0 0 0 0 8 0 0 0</td>
<td>D5</td>
</tr>
<tr>
<td>D6</td>
<td>0 0 0 0 9 0 0 0</td>
<td>D6</td>
</tr>
<tr>
<td>A5 = 12345000</td>
<td>A5 = 0 0 0 0 7 0 0 0</td>
<td></td>
</tr>
<tr>
<td>A6 = 50000200</td>
<td>A6 = 0 0 0 0 7 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Solution

Example 7.6
Write a 68020 instruction sequence to delete an element from a linked list using CAS2 instruction.

Solution

```
LEA HEAD, A0 ; Load address of head pointer into A0
MOVE.L (A0), D0 ; Move value of head pointer into D0
LOOP TST.L D0 ; Check for null pointer
  BEQ EMPTY ; If empty, no deletion required
  LEA (Next, D0, X), A1 ; Load address forward of link into A1
  MOVE.L (A1), D1 ; Put value of forward link in D1
  CAS2.L D0: D1, D1: D1, (A0) : (A1) ; If no change, update head and forward pointer
  RNE LOOP
```

EMPTY -

After Deleting an Element:
Table 7.4 summarizes the operation of the CAS, CAS2, and TAS instructions.

**TABLE 7.4**

<table>
<thead>
<tr>
<th>Operand</th>
<th>Instruction</th>
<th>Operation</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAS</td>
<td>8</td>
<td>Destination -0 → CCR</td>
<td>TAS (EA)</td>
</tr>
<tr>
<td>CAS</td>
<td>8,16,32</td>
<td>Destination -Dc → CCR</td>
<td>CASs Dc, Du, (AS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if Z, then Du → destination; else destination → Dc</td>
<td></td>
</tr>
<tr>
<td>CAS2</td>
<td>16,32</td>
<td>dest1 - Dc1 → CCR; if Z, dest2 - Dc2 → CCR; if Z, Du1 → dest 1; Du2 → dest2; else Dest 1 → DC1, dest 2 → Dc2</td>
<td>CAS2s.Dc1.Dc2.Du1.Du2. (Ro1:Ro2)</td>
</tr>
</tbody>
</table>

| Note: s = B, W, or L. |

7.1.4 COPROCESSOR INSTRUCTIONS

Table 7.5 lists the MC68020 coprocessor instructions. These instructions are available on the MC68020 system when a coprocessor such as the MC68881 (floating point) or MC68881 (paged memory management unit) is interfaced to the system. Note that cp in these instructions is replaced by F for floating point (MC68881) or P for paged (MC68881), depending on the coprocessor.

In this instruction F replaces cp and MOVE replaces GEN in the cpGEN format. This instruction moves 4 bytes from the MC68020-based microcomputer memory, starting with a location addressed by the contents of A0 to low 32 bits of the 80-bit register (FPO) in the MC68881; A0 is then incremented by 4.

**TABLE 7.5**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand</th>
<th>Operation</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpGEN</td>
<td>User defined</td>
<td>Pass command word to coprocessor and respond to coprocessor primitives</td>
<td>cpGEN (parameters defined by coprocessors)</td>
</tr>
<tr>
<td>cpBcc</td>
<td>16,32</td>
<td>If cpcc true, then PC + d → PC</td>
<td>cpBcc.Dn, (label)</td>
</tr>
<tr>
<td>cpDBcc</td>
<td>16</td>
<td>If cpcc false, then (Dn - 1 → Dn; if Dn = -1, then PC + d → PC)</td>
<td>cpDBcc.Dn, (label)</td>
</tr>
<tr>
<td>cpScc</td>
<td>8</td>
<td>If cpcc true, then 1's → destination, else 0's → destination</td>
<td>cpScc (EA)</td>
</tr>
<tr>
<td>cpTRAPcc</td>
<td>None</td>
<td>If cpcc true, then TRAP</td>
<td>cpTRAPcc (EA)</td>
</tr>
<tr>
<td>cpSAVE</td>
<td>16,32</td>
<td>Save internal state of coprocessor</td>
<td>cpSAVE (EA)</td>
</tr>
<tr>
<td>cpRESTORE</td>
<td>None</td>
<td>Restore internal state of coprocessor</td>
<td>cpRESTORE (EA)</td>
</tr>
</tbody>
</table>
TABLE 7.5 (continued)

Coprocessor Instructions

Privileged instructions for O.S. context switching (task switching) support.

- MAIN and CO each do what they know how to do best
  - MAIN: Tracks instruction stream
    - Takes exceptions
    - Takes branches
  - CO: Does graphics manipulations
    - Calculates transcendental, floating point
    - Does matrix manipulations

When MAIN obtains an F-line op word, it cooperates with a coprocessor to complete execution of the instruction.

**Co-Processor F-line Instruction**

<table>
<thead>
<tr>
<th>000-101</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>110-111</td>
<td>User Defined</td>
</tr>
</tbody>
</table>

| 000 = 68851 |
| (Pagd MMD) |
| 001 = 68881 |
| (Floating Point) |

In the above table, each type (bits 6-8), type dependent (bits 0-5) specifies effective address, conditional predicate, etc. For example, for general instructions (bits 5, 7, 6 = 00), type dependent (bits 0-5) specifies (ea).

The instruction FMOVE.L (A0) -, FP0 contains two words in memory as follows:

![Diagram of FMOVE.L instruction]

Therefore, the FMOVE.L (A0) -, FP0 contains two words in memory with the first word SF218 as the op word and the second word $4000$ as the command word for the coprocessor.

The "cmpBc displacement" is the coprocessor conditional branch instruction. If the specified coprocessor condition is satisfied, program execution continues at location (PC) + displacement; otherwise, the next instruction is executed. The displacement is a two's complement integer which may be either 16 or 32 bits. The coprocessor determines the specific condition from the condition field in the operation word. A typical example of this instruction is FSEQ.L.

**FSEQ.L**

The cmpBc (label) instruction works as follows. If qpc is false, then Dn - 1 -> Dn and if Dn ≠ -1, then PC + d → PC, or if Dn = -1, then next instruction is executed. On the other hand, if qpc is true, then the next instruction is executed.
The coprocessor determines the specific condition from the condition word which follows the operation word. A typical example of this instruction is

**FDNIE.W START**

The operand size is 16 bits.

The `cplcc.B (EA)` instruction tests a specific condition. If the condition is true, the byte specified by (EA) is set to true (all ones); otherwise that byte is set to false (all zeros). The coprocessor determines the specific condition from the condition word which follows the operation word. (EA) in the instruction can use all modes except Aa, immediate, (A16, PC, (d8, PC, Xn), (bd, PC, Xn), and (bd, PC, Xn, od). An example of this instruction is `FSEQ.B $8000 0F20`.

The `cplTRAPc` or `cplTRAPcc #` data checks the specific condition on a coprocessor. If the selected coprocessor condition is true, the MC68020 initiates exception processing. The vector number is generated to reference the `cplTRAPc` exception vector, the stacked PC is the address of the next instruction. If the selected condition is false, no operation is performed and the next instruction is executed. The coprocessor determines the specific condition from the (EA) which follows the operation word. The user-defined option of immediate data (third word of the machine code for `cplTRAPc #`) data) is used by the trap handler routine. A typical example of this instruction is

**FTRAPEQ**

`cplTRAPcc` is unsized and `cplTRAPc #` data has word and long word operands. `cplSAVE` and `cplRESTORE` are privileged instructions. Both instructions are unsized. `cplSAVE (EA)` saves the internal state of a coprocessor. (EA) can be predecrement on all alterable control addressing modes. This instruction is used by an operating system to save the context (internal state) of a coprocessor. The MC68020 initiates a `cplSAVE` instruction by reading an internal register.

The `cplRESTORE (EA)` instruction restores the internal state of a coprocessor. (EA) can use postincrement or control addressing mode. This instruction is used by an operating system to restore the context of a coprocessor for both the user-visible and the user-invisible state.

Caches have been used for years in larger machines to increase performance without greatly increasing system cost and complexity. As microprocessor clock speeds increase (the latest version of the MC68020 can run at 25 MHz), memory access time is often the limiting factor in system performance. It is possible to design a cache that is only a fraction of the size of the main memory store and yet significantly decrease the average access time to the main store.

Figure 7.16 illustrates the concept of cache memory.

![Figure 7.16 Cache memory concept](image)

This concept has been incorporated in the MC68020, which contains...
a 256-byte on-chip instruction cache. The cache improves performance by reducing the number of fetches required to external memory.

In a local memory, the address bits directly specify the location of the data. With a cache memory, however, the address does not specify a particular data location in the cache. Rather, the address information stored in the cache is called a tag. All cache tags are compared simultaneously to determine which cache data location is to be accessed.

The cache interface to the processor data paths within the MC68020 allows complete overlap of instruction fetches with data operand accesses. If simultaneous instruction and data operand requests are generated by the MC68020 microprocessor, a “hit” in the instruction cache allows concurrent fetches to take place. Also, increasing system performance is a two-clock-cycle access time for an access that “hits” the cache. This provides a 30% improvement over external three-clock memory accesses.

The memory access read operation flow diagram is shown in Figure 7.17.

In systems without an instruction cache, when the processor fetches an instruction from memory, no processing can be performed until the instruction has been decoded.

When the MC68020 instruction cache is enabled and an instruction fetch can be performed from the cache, the processor spends less time waiting for information from external memory. Because the MC68020 uses the bus for a small amount of time, it increases system performance by providing more bus bandwidth for other bus masters such as an MMU, DMA devices, or coprocessors. The MC68020 processor’s new three-clock bus cycles (as opposed to the four-clock cycles required by earlier MC68000 family members) also help to free the system bus for other traffic.

The MC68020 instruction cache is a 256-byte direct-mapped cache organized as 64 long word entries. Each cache entry consists of a tag field made up of the upper 24 address bits, the FC2 (user/supervisor) value, one valid bit (V), and 32 bits (two words) of instruction data. Figure 7.18 shows the MC68020 on-chip cache organization. The 68020 cache only stores instructions, data or operands are fetched directly from main memory as needed.

A processor tends to execute its program from several small areas in

FIGURE 7.17 Memory access (read).
memory. Therefore, the cache helps the processor use fewer bus cycles by storing recently executed instructions in anticipation of using them again. Further, when an instruction is supplied from a cache rather than from memory, it reaches the processor without delay and it requires zero clock cycles (parallel operation). A timing diagram showing parallel cache and external bus access is given in Figure 7.19.

![Timing Diagram](image)

**FIGURE 7.19** Parallel cache and external bus access.

When the processor fetches an instruction, it checks the cache first to determine if the word required is in the cache. First, one of 64 entries is selected by using the index field (A2-A7) of the access address as an index into the cache. Next, the 68020 compares address bits A11-A8 and function code bit FC2 with the 24-bit tag of the selected entry. If the function code and the address bit match, the 68020 sets the valid bit for the cache entry. This is called a cache hit. Finally, the 68020 uses address bit A1 to select the proper instruction word from the cache memory. If there is no match or valid bit is clear, a cache miss occurs and the instruction is fetched from external memory and put into the cache.

The MC68020 uses a 32-bit data bus and fetches instructions on long word address boundaries. Hence, each 32-bit instruction brings in two 16-bit instruction words which are then written into the on-chip cache. Subsequent prefetches will find the next 16-bit instruction word already present in the cache, and the related bus cycle is saved. Even when the cache is disabled, the subsequent prefetch will find the bus controller still holds the two instruction words and can satisfy the prefetch, again saving the related bus cycle. The bus controller provides an instruction hit rate of up to 50% even with the on-chip cache disabled.

Only the CPU uses the internal cache, so users have no direct access to the entries. However, several instructions allow the user to control the cache or dynamically disable it through the external hardware cache disable pin (CDIS). Typically, it is used by an emulator or bus state analyzer to force all bus cycles to be external cycles. The processor's two cache registers (CACR, CARR) can be programmed while in supervisor mode by using the MOVEC instruction. Enabling, disabling, freeing, or clearing the cache is carried out by the cache control register (CACR). The CACR also allows the operating system to maintain and optimize the cache. The cache control (CACR) is shown in Figure 7.20.

![Cache Control Register](image)

**FIGURE 7.20** Cache control register format. C = clear cache, CE = clear entry, F = freeze cache, and E = enable cache.

The clear cache (C) bit of the CACR is used to invalidate all entries in the cache. This is termed as “flashing the cache”. Setting the (C) bit causes all valid bits (V) in the cache to be cleared, thus invalidating all entries.

The Clear Entry (CE) of the CACR is used in conjunction with the address specified in the cache address register (CAAR). When writing to and setting the (CE) bit, the processor uses the CAAR index field to locate the selected address in the CAAR and invalidate the associated entry by clearing the valid bit.

The freeze cache (F) bit of the CACR keeps the cache enabled, but cache misses are not allowed to update the cache entries. This bit can be used by emulators to freeze the cache during emulation execution. It could be used to lock a critical region of the code in the cache after it has been executed, providing the cache is enabled and the freeze bit is cleared. The enables cache (E) bit is used for system's debug and emulation. This bit allows the designer to operate the processor with the cache disabled as long as the (E) bit remains cleared.

The cache address register (CAAR) format is shown in Figure 7.21. The CAAR is used by the MC68020 to provide an address for the Clear Entry (CE) function as implemented in the CACR. The index portions
this register is used to specify which one of 64 entries to invalidate by clearing the associated cache entry valid bit (V).

If the cache is enabled and encounters a breakpoint instruction, it may supply the replacement op code (N) times. On the Nth execution, the breakpoint op code is fetched from the cache, and the associated loop counter is decremented to zero. At this point, an illegal instruction exception routine is evoked which clears the breakpoint entry in the cache, and the saved op code is restored in memory.

Although the microarchitecture of the MC68020 is highly pipelined, the predominant pipeline mechanism is a three-stage instruction pipeline. Figure 7.22 shows the MC68020 pipeline organization. The pipeline is completely internal to the processor and is used as part of the instruction fetching and decoding circuitry. Instructions from the on-chip cache, or from external memory (if the cache is disabled), go into the first stage of the pipeline and synchronously pass through the following two stages.

The pipeline output gives the 68020's control and execution unit a completely decoded instruction. The 68020 loads data and other operands into the pipeline so they are ready for immediate use. The pipeline speeds 68020 operation by making information available immediately. The benefit of the pipeline is to allow concurrent operations (parallelism) for up to three words of a single instruction or for up to three consecutive one-word instructions. Therefore, the performance benefits of a pipeline are maximized during the execution of in-line code.

7.3 MC68020 VIRTUAL MEMORY

Virtual memory is a technique that allows all user programs executing on a processor to behave as if each had the entire 4-GB addressing range of the MC68020 at its disposal, regardless of the amount of physical memory actually present in the system. Virtual memory can be supported by providing a limited amount of high-speed physical memory that can be accessed directly by the processor while maintaining an image of a much larger "virtual" memory on a secondary storage device, such as high-capacity disk drives. Figure 7.23 shows a minimal system configuration for a typical virtual memory system. Also, any given instruction must be able to be aborted and restarted. When a processor attempts to access a location in the virtual memory map that is not resident in physical memory (this is called a "page fault"), the access to that location is temporarily suspended while data are fetched from secondary storage and placed into physical memory. Page faults force a trap to the bus error exception vector.

The MC68020 processor has the abort capability via the bus error (BERR) input to the processor. When BERR is asserted, exception processing causes the processor to save sufficient information to allow complete restoration of the faulted instruction. The faulted instruction will be recovered by the instruction continuation method, in which the faulted instruction is allowed to complete execution from the point of the fault. Instruction continuation is crucial for supporting virtual I/O devices in memory.
main general-purpose processor cannot accommodate. The coprocessor concept allows the enhancement of general-purpose processor performance and capabilities for a particular application, without overloading the main processor architecture.

Coprocessors are DMA or non-DMA, according to the way they use the bus. A coprocessor is a Direct Memory Access (DMA) coprocessor if it can control the bus independent of the main processor. A non-DMA coprocessor cannot control the bus, so it operates as a bus slave.

The MC68020 coprocessor is not a standard peripheral device. A MC68020 coprocessor communicates with the main processor through the protocol defined at the coprocessor interface. Dedicated coprocessor instructions are provided to use the coprocessor capabilities.

The interface between the main processor and a coprocessor is transparent to the user. The programmer does not have to be aware that a separate piece of hardware is executing some of the program code sequence. Hardware-implemented microcode within the MC68020 handles coprocessor interfacing so that a coprocessor can provide its capabilities to the programmer without appearing as an external hardware, but rather as a natural extension to the main processor architecture.

In contrast, standard peripheral hardware is generally accessed through the use of an interface register mapped into the memory space of the main processor. The programmer uses standard processor instructions to access the peripheral interface registers and thus use the peripheral's capabilities. Accessing coprocessors over the coprocessor interface is straightforward, because the interface uses standard MC68020 asynchronous bus structure, without any special signals.
The coprocessor does not have to be architecturally similar to the main processor, but can be designed so that it best suits its particular application. The only requirement is that it adheres to the coprocessor's interface protocol. A coprocessor can be implemented as a Very Large Scale Integration (VLSI) device to solve some of the more common special-purpose processing needs (such as floating-point computations) or to support virtual memory/machine requirements (such as MMUs, DMA controllers, or master/slave bus arbitration devices). A board-level design particular to some limited but important application, or even a separate microcomputer system, can become a system coprocessor. The MC68020 coprocessor interface is designed to provide full support for all sequential operations necessary for nonconcurrent operation between the main processor and its associated coprocessors. The MC68020 coprocessor interface does allow concurrency in coprocessor execution, but it is the responsibility of the coprocessor designer to implement this concurrency. At the same time, the designer must maintain the MC68020 programmer's model based on sequential, nonconcurrent instruction execution.

When communicating with a coprocessor, the MC68020 executes bus cycles in CPU memory space to access a set of Coprocessor Interface Registers (CIRs). Table 7.6 shows how the separate coprocessor interface registers are located in CPU space. Within this interface register set, the various registers are allocated to specific functions required for operating the coprocessor interface. There are registers specifically for passing information such as commands, operands, and EAs. Other registers are allocated for use during a context switch operation.

### Table 7.6
**Coprocessor Interface Register**

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response</td>
<td>Requests action from CPU</td>
<td>R/W</td>
</tr>
<tr>
<td>Control</td>
<td>CPU directed control</td>
<td>R</td>
</tr>
<tr>
<td>Save</td>
<td>Initiate save of internal state</td>
<td>R</td>
</tr>
<tr>
<td>Restore</td>
<td>Initiate restore of internal state</td>
<td>R</td>
</tr>
<tr>
<td>Operation word</td>
<td>Current coprocessor instruction</td>
<td>W</td>
</tr>
<tr>
<td>Command word</td>
<td>Coprocessor specific command</td>
<td>W</td>
</tr>
<tr>
<td>Condition word</td>
<td>Condition to be evaluated</td>
<td>W</td>
</tr>
<tr>
<td>Operand</td>
<td>32-bit operand</td>
<td>W</td>
</tr>
<tr>
<td>Register select</td>
<td>Specifies CPU register or mask</td>
<td>R</td>
</tr>
<tr>
<td>Instruction address</td>
<td>Pointer to coprocessor instruction</td>
<td>R/W</td>
</tr>
<tr>
<td>Operand address</td>
<td>Pointer to coprocessor operand</td>
<td>R/W</td>
</tr>
</tbody>
</table>

Figure 7.25 shows the coprocessor interface register set map and communication protocol between the main processor and the coprocessor necessary to execute a coprocessor instruction. The MC68020 implements the CIR communication protocol automatically, so the programmer is only concerned with the coprocessor's instruction and data type extensions to the MC68020 programmer's model.

![Figure 7.25 Coprocessor interface register set map.](image)

Figure 7.26 shows the CIR set(s) (that is, more than one coprocessor) address map in CPU space.

![Figure 7.26 Coprocessor address map in CPU space.](image)

The MC68020 indicates that it is accessing CPU memory space by encoding the function code lines high (FC0-FC2 = 111). Thus, the CIR.
set is mapped into CPU space the same way that a peripheral interface register set is generally mapped into data space. The address bus then selects the desired coprocessor chip.

Encoding the address bus during coprocessor communication is shown in Figure 7.27. By using the cp-ID field on the address bus, up to eight separate coprocessors can be interfaced concurrently to the MC68020. Figure 7.27 also shows how simply this can be done. Interfacing to these separate coprocessors is just a matter of decoding the relevant cp-ID field (address lines A13-A15) and the corresponding CPU space type field (address lines A16-A9) encoded within the coprocessor instruction, so that the MC68020 communicates with the relevant register set in CPU space.

Figure 7.28 shows a block level diagram of the MC68020 coprocessor interface signals. System designers can use multiple coprocessors of the same type (for example, several MC68881 floating-point coprocessors operating concurrently) by assigning a uniquely encoded cp-ID to each one.

Coprocessors define their own instruction set to support their specific design functions. Executing a coprocessor instruction uses an "E-line operation word" (an operation word is the first word of any instruction) to specify the type of coprocessor instruction that will be executed. Figure 7.29 shows the E-line coprocessor instruction operation word format.

The E-line operation word can be appended by extension words that provide additional instruction execution information. The E-line operation word (coprocessor instruction) contains the encoded coprocessor identification code (cp-ID) that the MC68020 uses to identify which of eight possible coprocessors to access for a given instruction. This decouples the main processor from the definition of a particular coprocessor instruction set, making the interface more general in nature. After evaluating the instruction, the coprocessor determines the service required from the main processor (if any), so that it can execute the instruction.

These service requirement requests are transmitted to the main processor in the form of an encoded "response primitive" word. The response primitive word is then evaluated and executed by the main processor. The response register of the CIR set communicates the coprocessor's service requests to the main processor. Thus, the response register contains the primitive response instruction word. Figure 7.30 shows the coprocessor response primitive format.

The MC68020's use of primitive responses minimizes coprocessor interface overhead. For example, the main processor can perform (EA) calculations and pass the result to the coprocessor or fetch a variable length operand and pass it on. By performing such complicated services for the coprocessor, the main processor shows the coprocessor design to be much simpler.
The branch instruction (cpCC) allows branching based upon a coprocessor condition. Both 16- and 32-bit displacement are supported. The conditional instructions (cpcc, cpTRAPc, and cpDBc) allow evaluation of one of 64 coprocessor conditions, and based upon the result, they conditionally set (cpScC), trap (cpTRAPc), or decrement and branch (cpDBc). For conditional instructions, the coprocessor is passed a condition for evaluation and then informs the main processor of the result.

The save and restore instructions are used to support operating system context switching. These two instruction types allow the saving and restoring of the state of a coprocessor in a virtual machine environment. During a context switch in a multitasking environment, the coprocessor state is treated as a part of the overall system state (along with the main processor's general purpose register, PC, and SR) and must be switched out when the current task is suspended and another task takes over. The cpSAVE and cpRESTORE commands simplify this swap and reduce the amount of information that has to be saved during the context switch. Once a coprocessor instruction has been passed from the main processor to the coprocessor for evaluation, the coprocessor has a number of main processor-recognized primitives (Table 7.7) from which to implement a coprocessor instruction. Response primitives are instructions the coprocessor uses to communicate status information and processing service requests to the main processor. The main processor supports 18 basic primitives, some with several variations. Primitives can be grouped into five basic categories: processor/coprocessor synchronization, instruction stream manipulation, exception handling, general operand transfer, and register transfer.

Processor/coprocessor synchronization primitives allow the coprocessor to inform the main processor of its status. The coprocessor can perform the following functions:

- Signal that it is busy with a previous or current instruction
- Allow the main processor to process any pending interrupts while it is busy
- Request the main processor to pass the current PC value
- Direct the main processor to proceed with the next instruction, depending on whether the instruction tracing bits (TI, TO) located in the SR (see Table 7.8) are enabled
- Indicate the result of a conditional test

Instruction stream manipulation primitives allow the coprocessor to request the transfer of a coprocessor instruction to itself. The transfer instruction primitive allows the coprocessor to request the main processor to transfer up to 356 bytes from the instruction stream to the coprocessor.
TABLE 7.7
MC68020 Coprocessor Primitives

Processor synchronization
Busy with current instruction
Proceed with next instruction, if no trace
Service interrupts and re-query, if trace enabled
Proceed with execution, condition true/false

Instruction manipulation
Transfer operation word
Transfer words from instruction stream

Exception handling
Take privilege violation if S bit not set
Take pre-instruction exception
Take mid-instruction exception
Take post-instruction exception

General operand transfer
Evaluate and pass (EA)
Evaluate (EA) and transfer data
Write to previously evaluated (EA)
Take address and transfer data
Transfer to/from top of stack

Register transfer
Transfer CPU register
Transfer CPU control register
Transfer multiple CPU registers
Transfer multiple coprocessor registers
Transfer CPU SR and/or ScanPC

TABLE 7.8
Instruction Tracing Control

<table>
<thead>
<tr>
<th>T1</th>
<th>T0</th>
<th>Tracing function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No tracing</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Trace on change of flow (BRA, JMP, etc.)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Trace on instruction execution (any instruction)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Undefined, reserved</td>
</tr>
</tbody>
</table>

and to update the PC appropriately. This means that coprocessor instructions can be any number of words in length.

Exception handling primitives force the main processor to check for and/or take an exception. The coprocessor can request the main processor to perform a supervisor check before executing a privileged coprocessor instruction. If the main processor is not in the supervisor state, a privilege violation exception is taken, and the main processor may be forced to save enough state information to recover from the exception later.

General operand transfer primitives are used to request the transfer of operands from or to the coprocessor. These primitives include evaluating an (EA) for the coprocessor, transferring the evaluated (EA) to the coprocessor, transferring data from/to the (EA), and transferring data to a previously evaluated (EA). Also, the coprocessor may pass an address to the main processor, as well as request an operand transfer to or from the address passed. An additional primitive allows transfer of operands to or from the top of the active system stack.

Register operand transfer primitives allow transfer of single main processor control registers, single and multiple main processor address/data registers, and multiple coprocessor registers between the main processor and the coprocessor. Additionally, the coprocessor may request a transfer to or from the main processor SR and PC.

7.4.1 MC68881 FLOATING-POINT COPROCESSOR

The MC68881 HCMOS floating-point coprocessor implements IEEE standards for binary floating-point arithmetic. When interfaced to the MC68020, the MC68881 provides a logical extension to the MC68020 integer processing capabilities. The MC68882 is an upgrade of the MCC68881 and provides in excess of 15 times the performance of the MC68881. It is a pin and software compatible upgrade of the MC68881. A summary of the MC68881 features is listed below:

- Eight general purpose floating-point data registers, each supporting an 80-bit extended-precision real data format (a 64-bit mantissa, plus a sign bit and a 15-bit signed exponent)
- A 67-bit arithmetic unit
- A 67-bit barrel shifter for fast shift operations
- 46 instructions with 35 arithmetic operations
- Supports trigonometric and transcendental functions
- Supports seven data types: bytes, word, long word integer; single, double, and extended precision real numbers; and packed binary-coded decimal string real numbers
- 22 constants including π, e, and powers of 10
The MC68881 is a non-DMA-type coprocessor which uses a subset of the general-purpose coprocessor interface supported by the MC68020.

The MC68881 programming model is shown in Figure 7.32.

The MC68881 programming model includes the following:

- Eight 80-bit floating-point registers (FP0-FP7) (These general-purpose registers are analogous to the MC68020 D0-D7 registers.)
- A 32-bit control register containing enable bits for each class of exception trap and mode bits, to set the user-selectable rounding and precision modes
- A 32-bit status register containing floating-point condition codes, quotient bits, and exception status information
- A 32-bit Floating-Point Instruction Address Register (FPIAR) containing the MC68020 memory address of the last floating-point instruction that was executed (This address is used in exception handling to locate the instruction that caused the exception.)

The MC68881 can be interfaced as a coprocessor to the MC68020.

Figure 7.33 provides the MC68020/68881 block diagram.

The MC68881 is internally divided into three sections: the Bus Interface Unit (BIU), the Execution Control Unit (ECU) and the Microcode Control Unit (MCU).

The BIU communicates with the MC68020, the ECU and MCU execute all MC68881 instructions. The BIU contains the Coprocessor Interface Registers (CIRs) and the 32-bit control, status, and instruction address registers. The register select and DSACKx timing control logic are included in the BIU. The CIRs are addressed in the same way as memory by the MC68020.

The MC68020 implements the coprocessor interface protocol in hardware and microcode. When the MC68020 encounters a typical MC68881 instruction, the MC68020 writes the instruction to the memory-mapped command CIR and reads the response CIR. In this response, the BIU transmits any additional action required of the MC68020 by the MC68881. Upon satisfying the coprocessor requests, the MC68020 can fetch and execute subsequent instructions.

The MC68881 supports the following data formats:

- Byte Integer (B)
- Word Integer (W)
- Long Word Integer (L)
- Single Precision Real (S)
Double Precision Real (D)
Extended Precision Real (X)
Packed Decimal String Real (P)

The capital letters included in parentheses denote the suffixes added to instructions in the assembly language source to specify the data format to be used.

Figure 7.34 shows the MC68811 data formats. These data types are supported: integer, floating-point, and packed decimal string real data. These data formats are defined by IEEE standards. Integer data types do not include any fractional part of the number. The real formats contain an exponent part and a fractional (mantissa) part. The single-real and double-real formats provide sign of a fraction with 8- and 11-bit exponents and 23- and 52-bit fractional parts, respectively. The extended real format is 96 bits wide with a 16-bit exponent, 64-bit mantissa, and a sign bit for the mantissa. The packed decimal real is 96 bits wide which provides sign for both mantissa and exponent parts. It includes a 3-digit (12 bits for BCD) base-10 exponent and a 17-digit (68 bits) base-10 mantissa. This format contains two bits to indicate infinity or non-a-number (NAN) representations. Note that NAN is a symbolic representation of a special number or situation in floating-point format. NAN include all numbers with nonzero fractions with format's maximum exponent. The infinity data types include zero fraction and maximum exponent.

Whenever an integer is used in a floating-point operation, the integer is automatically converted to an extended-precision floating-point number.
before being used. For example, the instruction FADD.W #2, FP0 converts
the constant 2 to the floating-point number format in FP0, adds the two
numbers, and then stores the result in FP0. This allows integer in floating-
point operations and also saves user memory, since an integer representa-
tion of a number is normally smaller than the equivalent floating-point
representation.

The floating-point representation contains single-precision (32 bits),
double-precision (64 bits), and extended-precision numbers (96 bits) as
specified by the IEEE format. The single-precision and double-precision
data types should be used in most calculations involving real numbers. The
coefficient is biased and the mantissa is in sign and magnitude form. Single
and double precision require normalized numbers. Note that a normalized
number has the most significant bit of the mantissa positioned such that the
one lies to the left of the radix point. Therefore, only the fractional part of
the mantissa is stored in memory, which means that the most significant bit
is implied and equal to one.

Extended-precision numbers are 96 bits wide but only 80 bits are used,
and the unused 16 bits are for future expansion. Extended-precision
numbers are not for use as temporary variables, intermediate variables, or in
situations where extra precision is required.

For example, a compiler might select extended-precision arithmetic for
determining the value of the right side of an equation with mixed sized
data, and then convert the answer to the data type on the left side of the
equation. Extended precision data should not be stored in large arrays due
to the amount of memory required by each number. As with other data
types, the packed BCD strings are automatically converted to extended-
precision real values when they are input to the MC68881. This permits
packed BCD number to be used as input to any operation such as FADD P
# = 2.012E+18, FP1.

The MC68881 does not include any addressing modes. If the 68020
requests the 68020 to transfer an operand via the coprocessor interface, the
68020 provides the addressing mode calculations requested in the instruc-
tion.

Floating-point data registers FP0-FP7 always contain extended-precision
values. Also, all data used in an operation are converted to extended
precision by the MC68881 before the operation is performed. The
MC68881 provides all results in extended precision. The MC68881 in-
suctions can be grouped into six types:

1. MOVE instructions between the MC68881 and memory on the
MC68020
2. MOVE multiple registers
3. Monadic operations
4. Dyadic operations
5. Branch, set, or trap conditionally
6. Miscellaneous

7.4.1.a MOVE

All MOVE instructions from memory or from a MC68020 data reg-
ister to the MC68881 convert data from the source format to the internal
extended precision.

Also, all MOVE instructions from the MC68881 to memory or a
MC68020 data register convert from the internal extended precision for-
mat to the destination data.

Typical examples of MOVE instructions include:

F MOVE . (fmt) (EA), FPn
F MOVE . (fmt) FPn, (EA)
F MOVE . (fmt) FPn, FPn

7.4.1.b MOVE Multiple Registers

Any floating-point register FP0 through FP7 can be moved to or from
memory with one instruction. These registers are always moved as 96-bit
extended data with no conversion. Typical examples of FMOVEM include:

FMOVEM (EA), FP0-FP4/FP6
FMOVEM FP0/FP1/FP5, (EA)

Any combination of FP0-FP7 can be moved. (EA) can be control
modes, predecrement, or postincrement mode. For control or postinc-
crement mode, the order of transfer is from FP7-FP0. For predecrement
mode, the order of transfer is from FP0-FP7. Any combination of FP CR,
FPSR, and FPIAR can also be moved by FMOVEM instruction. These
registers are always moved in the order FPCR, FPSR, and FPIAR.

7.4.1.c Monadic

Monadic instructions have a single input operand. This operand may
be in a floating-point data register, memory, or in a MC68020 data
register. The result is always stored in a floating-point data register. Typical
examples include:

F TAN . (fmt) (EA), FPn
or
F TAN . X FPn, FPn
or
F TAN . X FPn
Chapter 7

MC68020 — Advanced Topics

7.4 MC68020 Coprocessor Interface

TABLE 7.9c
MC68881 — Monadic Instructions

Nontranscendental functions

FABS
ABSOLUTE VALUE
FINV
INTEGER PART
FNEG
NEGATE
FSQRT
SQUARE ROOT
FNOP
NO OPERATION (SYNCHRONIZE)
FGETEXP
GET EXPONENT
FGETMAN
GET MANTISSA
FIST
TEST

7.4.1.d Dyadic Instructions

Dyadic instructions have two input operands. The first input operand comes from a floating-point data register, memory, or a MC68020 data register. The second input operand comes from a floating-point data register. The second input is also the destination floating-point data register. Typical examples include:

FCPF. L (fpn) (fr), FPn

Tables 7.10a flowcharts the dyadic function and Table 7.10b lists dyadic instructions.

TABLE 7.10a
MC68881 Dyadic Functions

SOURCE
MC68020 REGISTER OR
MC68881 REGISTER OR
MEMORY

FUNCTION

DESTINATION
MC68881 FP REGISTER

TABLE 7.9a
Monadic Functions

<table>
<thead>
<tr>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY OR</td>
</tr>
<tr>
<td>MIREG REGISTER OR</td>
</tr>
<tr>
<td>MC68881 REGISTER</td>
</tr>
</tbody>
</table>

TABLE 7.9b
MC68881 — Monadic Instructions

Transcendental functions

<table>
<thead>
<tr>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>FACS</td>
</tr>
<tr>
<td>FASIN</td>
</tr>
<tr>
<td>FATAN</td>
</tr>
<tr>
<td>FATANH</td>
</tr>
<tr>
<td>FCOS</td>
</tr>
<tr>
<td>FCOSH</td>
</tr>
<tr>
<td>FETOX</td>
</tr>
<tr>
<td>FETOXM1</td>
</tr>
<tr>
<td>FLOG10</td>
</tr>
<tr>
<td>FLOG2</td>
</tr>
<tr>
<td>FLOGN</td>
</tr>
<tr>
<td>FLOGNP1</td>
</tr>
<tr>
<td>FSIN</td>
</tr>
<tr>
<td>FSINCOS</td>
</tr>
<tr>
<td>FSINH</td>
</tr>
<tr>
<td>FTAN</td>
</tr>
<tr>
<td>FTANH</td>
</tr>
<tr>
<td>FTENX</td>
</tr>
<tr>
<td>FTWOTOX</td>
</tr>
</tbody>
</table>

Source operand to extended precision (if necessary), computes the tangent of that number, and stores the result in the destination floating-point data register.

Table 7.9a flowcharts the monadic function. Tables 7.9b and c list all MC68881 monadic instructions.
TABLE 7.10b
MC 68881 Dyadic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>ADD</td>
</tr>
<tr>
<td>FCMP</td>
<td>COMPARE</td>
</tr>
<tr>
<td>FDIV</td>
<td>DIVIDE</td>
</tr>
<tr>
<td>FMOD</td>
<td>MOD</td>
</tr>
<tr>
<td>FMUL</td>
<td>MULTIPLY</td>
</tr>
<tr>
<td>FREM</td>
<td>IEE REMAINDER</td>
</tr>
<tr>
<td>FSCALE</td>
<td>SCALE EXPONENT</td>
</tr>
<tr>
<td>FSUB</td>
<td>SUBTRACT</td>
</tr>
</tbody>
</table>

7.4.1.e BRANCH, Set, or Trap-On Condition

These instructions are similar to those of the MC68020 except that move conditions exist due to the special values in IEEE floating-point arithmetic. When the MC68020 encounters a floating-point conditional instruction, it passes the instruction to the MC68881 for performing the necessary condition checkup. The MC68881 then checks the condition and tells the MC68020 whether the condition is true or false. The MC68020 then takes the appropriate action.

The MC68881 conditional instructions are:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBBcc.W</td>
<td>displ Branch</td>
</tr>
<tr>
<td>or L</td>
<td></td>
</tr>
<tr>
<td>FDBcc.W</td>
<td>displ Decrement and branch</td>
</tr>
<tr>
<td>or L</td>
<td></td>
</tr>
<tr>
<td>FScc.W</td>
<td>displ Set byte according to condition</td>
</tr>
<tr>
<td>or L</td>
<td></td>
</tr>
<tr>
<td>FTRAPcc.W</td>
<td>displ Trap-on condition</td>
</tr>
<tr>
<td>or L</td>
<td></td>
</tr>
</tbody>
</table>

All the above instructions can have 16- or 32-bit displacement. cc is one of the 32 floating-point conditional test specifiers as shown in Table 7.11.

7.4.1.f Miscellaneous Instructions

These instructions include moves to and from the status, control, and instruction address registers. The virtual memory instructions FSAVE and FRESTORE that save and restore the internal state of the MC68881 also fall into this category. These instructions include:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>Signaling false</td>
</tr>
<tr>
<td>SEQ</td>
<td>Signaling equal</td>
</tr>
<tr>
<td>GT</td>
<td>Greater than</td>
</tr>
<tr>
<td>GE</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>LT</td>
<td>Less than</td>
</tr>
<tr>
<td>LE</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>GL</td>
<td>Greater or less</td>
</tr>
<tr>
<td>GLE</td>
<td>Greater or less or equal</td>
</tr>
<tr>
<td>NGLE</td>
<td>Not (greater, less, or equal)</td>
</tr>
<tr>
<td>NGL</td>
<td>Not (greater or less)</td>
</tr>
<tr>
<td>NLE</td>
<td>Not (less or equal)</td>
</tr>
<tr>
<td>NLT</td>
<td>Not (less than)</td>
</tr>
<tr>
<td>NGE</td>
<td>Not (greater or equal)</td>
</tr>
<tr>
<td>NGT</td>
<td>Not (greater than)</td>
</tr>
<tr>
<td>SNE</td>
<td>Signaling not equal</td>
</tr>
<tr>
<td>ST</td>
<td>Signaling true</td>
</tr>
</tbody>
</table>

Note: The preceding conditional tests set the ESUN bit in the status register exception byte under any circumstances.
Chapter 7

MC68020 — Advanced Topics

INSTRUCTION: FADD X, FPx, FPy, FPz

FPMOVE (EA), FPCR
FPMOVE FPc, (EA)
FPRESTORE (EA)

FPCR means floating-point control register. The MC68881 does not perform addressing mode calculations. The MC68020 carries out this calculation specified in the instruction.

Typical addressing modes include immediate, postincrement, preincrement, direct, and the indexed/indirect modes of the MC68020. Some addressing modes are restricted for some instructions. For example, PC relative mode is not permitted for a destination operand.

The MC68881 can execute an instruction concurrently or nonconcurrently with the MC68020 depending on the instruction being executed. Figure 7.35 shows examples of concurrency and nonconcurrency.

FIGURE 7.35a MC68020/MC68881 concurrency example.

Figure 7.36 shows the MC68881 pins and signals.

The 68881 is included either in a 54-pin DIP or in a 68-PGA (Pin Grid Array) package. There are 7 Vcc (+5V) and 15 grounds for power distribution to reduce noise.

The five address lines A0−A4 are used by the MC68020 to select the coprocessor interface registers mapped in the MC68020 address space. These address pins select the registers as listed in Table 7.12.

When the MC68881 is configured to operate over an 8-bit data bus for processors such as MC68308, A0 pin is used as address signal for byte accesses of the MC68881 interface register. When the MC68881 is configured to operate over a 16-bit data bus (68000) or 32-bit data bus (68020), both A0 and SIZE pins are used, according to Table 7.13.

FIGURE 7.35b MC68020/MC68881 nonconcurrency example.

SIZE and A0 pins are used to configure the MC68881 for operation over 8-, 16-, or 32-bit data bus.

The address strobe AS is when LOW indicates that there is a valid address on the address bus, and both CS and R/W are valid.
TABLE 7.12
Coprocessor Interface Register Selection

<table>
<thead>
<tr>
<th>A4-A0</th>
<th>Offset</th>
<th>Width</th>
<th>Type</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000x</td>
<td>$00</td>
<td>16</td>
<td>Read</td>
<td>Response</td>
</tr>
<tr>
<td>0001x</td>
<td>$02</td>
<td>16</td>
<td>Write</td>
<td>Control</td>
</tr>
<tr>
<td>0010x</td>
<td>$04</td>
<td>16</td>
<td>Read</td>
<td>Save</td>
</tr>
<tr>
<td>0011x</td>
<td>$06</td>
<td>16</td>
<td>R/W</td>
<td>Restore</td>
</tr>
<tr>
<td>0100x</td>
<td>$08</td>
<td>16</td>
<td>—</td>
<td>(reserved)</td>
</tr>
<tr>
<td>0101x</td>
<td>$0A</td>
<td>16</td>
<td>Write</td>
<td>Command</td>
</tr>
<tr>
<td>0110x</td>
<td>$0C</td>
<td>16</td>
<td>—</td>
<td>(reserved)</td>
</tr>
<tr>
<td>0111x</td>
<td>$0E</td>
<td>16</td>
<td>Write</td>
<td>Condition</td>
</tr>
<tr>
<td>1000x</td>
<td>$10</td>
<td>32</td>
<td>R/W</td>
<td>Operand</td>
</tr>
<tr>
<td>1001x</td>
<td>$12</td>
<td>16</td>
<td>Read</td>
<td>Register select</td>
</tr>
<tr>
<td>1100x</td>
<td>$18</td>
<td>32</td>
<td>Read</td>
<td>Instruction address</td>
</tr>
<tr>
<td>1110x</td>
<td>$1C</td>
<td>32</td>
<td>R/W</td>
<td>Operand address</td>
</tr>
</tbody>
</table>

TABLE 7.13
Data Bus Configuration

<table>
<thead>
<tr>
<th>A0</th>
<th>Size</th>
<th>Data bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>Low</td>
<td>8-bit</td>
</tr>
<tr>
<td>Low</td>
<td>High</td>
<td>16-bit</td>
</tr>
<tr>
<td>High</td>
<td>High</td>
<td>32-bit</td>
</tr>
</tbody>
</table>

A low on DS indicates that there is valid data on the data bus during a write cycle.

If the bus cycle is a MC68020 read from MC68881, the MC68881 asserts DSACK1 and DSACK0 to indicate that the information on the data bus is valid. If the bus cycle is a MC68020 write to the MC68881, DSACK1 and DSACK0 are used to acknowledge acceptance of the data by the MC68881.

The MC68881 also uses DSACK0 and DSACK1 to dynamically indicate the device size on a cycle-by-cycle basis.

A low on MC68881 RESET pin clears the floating-point control, status, and instruction address registers. When performing power-up reset, external circuitry should keep the RESET pin asserted for a minimum of four clock cycles after Vcc is within tolerance. After Vcc is written tolerance

for more than the initial power-up time, the RESET pin must be asserted for at least two clock cycles.

The MC68881 clock input is a TTL-compatible signal that is internally buffered for generation of the internal clock signals. The clock input should be a constant frequency square wave with no stretching or shaping techniques required. The MC68881 can be operated from a 12, 16.67, or 20-MHz clock.

The SENSE pin may be used as an additional ground pin for more noise immunity or as an indicator to external hardware that the MC68881 is present in the system. This signal is internally connected to the GND of the die, but it is not necessary to connect it to the external ground for correct device operation. If a pullup resistor (larger than 10 K ohm) is connected to this pin location, external hardware may sense the presence of the MC68881 in a system.

Figure 7.37 shows the MC68020/MC68881 interface.

The A0 and SIZE pins are connected to Vcc for 32-bit operation. Note that A19, A18, A17, A16 = 0010, (indicating coprocessor function), FC1 = 111, (meaning CPU space cycle), and A15 A14 A13 = 001, (indicating 6881 floating-point coprocessor) are used to enable 68881 CS. The 68020 A19 and A18 pins are not used in the chip select decode since their values are 00.

The BERR pin is asserted for low CS and high SENSE signals. A trap routine can be executed to perform the coprocessor operations.

The protocols between the MC68020 and MC68881 are shown in Figures 7.38a through 7.38b.

Let us now provide a MC68881 interface example: consider FMOVE.P (A0) → FPO. Note that P means “packed decimal” (12 bytes).

The above instruction moves 12 bytes from MC68020 main memory addressed by the MC68020 address register A0 to the MC68881 floating-point register, FPO. This MOVE instruction converts packed BCD numbers to extended-precision real values. The machine code equivalent of FMOVE.P (A0) → FPO includes two words as follows:

```
$F218
$4C00
```

The first word $F218 in Figure 7.38a specifies that the instruction is F-line, coprocessor type is MC68881, and that the instruction is general type with source operand having postincrement mode indirectly addressed by A0. The second word $4C00 in Figure 7.38a defines the op code type (FMOVE), source data format (.P), and destination register (FPO).

When the MC68020 executes the MOVE.P (A0) → FPO instruction,
it places 5000200A on the address bus A31-A0. In this data, A19, A18, A17, A16 = 0010, indicates coprocessor communications and bits A15, A14, A13 = 001 specifies that the coprocessor is MC68881. Bits B1-20 and bits 5-11 are always zeros during a coprocessor access and FC2, FC1, FC0 = 111, indicates CPU space cycle. Figure 7.38b provides the MC68020 address bus encodings for MC68881 access (see Figure 7.38b). Bits 0-4 provide 50A which is the address for the MC68881 command register. The information encoded on the FC2-FC0 lines and address lines A17-A13 (A19, A18 always zero) of the MC68020 during the MC68881 access can be used to generate the chip select signal (CS) for the MC68881 (see Figure 7.37). The MC68020 address lines A14-A0 are used to specify that the command register (address 50A) in the command interface register set is being accessed during operand transfers between the MC68020 and the MC68881. The MC68020 also writes $4C00 [second word of the instruction FMOVE.P(A0) +, FP0] on the data bus. The MC68881 asserts DSACKX, receives this datum $4C00, and places it in its command register. The MC68881 responds to the MC68020 command in the command register by writing appropriate data in the response register (address 50H). The MC68881 places these data on the data bus and then asserts DSACK. The MC68020 places 111 on FC2-FC0 and 50002000 on A31-A0 (A4A3A2A1A0 = 0000) on a reset. The MC68020 also asserts R/W pin to read the contents of the MC68881 response register (3960C in this example) as shown in Figure 7.38c.
Chapter 7

7.4 MC68020 Coprocessor Interface

INSTRUCTION | (PC) ADDRESS | BUS R/W | DATA BUS | COPROCESSOR RESPONSE
-------------|--------------|--------|----------|---------------------
FMOVE.P (A0)+, FP0 | SF218 | $4C00 | 16110 2200A | W | S4C00 | DSACK (accept the command)

CPU Space

Address Bus

Bits 31-24 are zeros

FIGURE 7.34b MC68801 interface example.

INSTRUCTION | (PC) ADDRESS | BUS R/W | DATA BUS | COPROCESSOR RESPONSE
-------------|--------------|--------|----------|---------------------
FMOVE.P (A0)+, FP0 | SF218 | $4C00 | 1111102200A | W | S4C00 | DSACK (accept the command)

Valid encoders for Source operand:
110 = Memory

FIGURE 7.34c MC68801 interface example.

INSTRUCTION | (PC) ADDRESS | BUS R/W | DATA BUS | COPROCESSOR RESPONSE
-------------|--------------|--------|----------|---------------------
FMOVE.P (A0)+, FP0 | SF218 | $4C00 | 1111102200A | R | $5900C | Place Response Register on Data Bus and DSACK

Valid encoders for Source operand:
110 = Memory

FIGURE 7.34c MC68801 interface example.
The MC68881 response register contents of $960C in Figure 7.38c can be interpreted as follows: bit 15 in the coprocessor response primitive format, denoted by CA (call or come again), is used to specify call again operation of the MC68020. Whenever the MC68020 receives a response primitive from the response register with the CA bit set to one, it will carry out the service indicated by the primitive and then come back to read the response register.

Bit 14 in the response primitive format, denoted by PC, is used to pass program counter. If the PC bit is set to one, the MC68020 will immediately pass the current PC contents to the coprocessor instruction address interface register.

As the first operation in servicing, the primitive request bit 13 of the coprocessor response primitive format, denoted by DR, is the direction bit and it is used in conjunction with operand transfers between the MC68020 and the coprocessor. If DR = 0, the direction of transfer is from the main processor to the coprocessor (MC68020 write). If DR = 1, the direction of transfer is from the coprocessor to the MC68020 (MC68020 read). If the operation does not involve operand transfer, the bit value is part of the particular primitive encoding.

In Figure 7.38c, CA = 1 for call again, PC = 0 for don’t pass, and PC and DR = 0 for transfer from the MC68020 to the MC68881 as required in the FMOVE.P instruction. Bits 12 and 11 are always zero; bits 10, 9, and 8 contain 110 indicating value (EA) mode for the source; length byte = 12, indicating 12 bytes (P) or 96 bits to be transferred.

Upon receipt of this primitive, the MC68020 calculates the effective address and then transfers the number of bytes (12 bytes in this case) specified in the response primitive.

As shown in Figure 7.38d, the MC68020 outputs P01 to FC2FC1FC0 and the address $00000000 on A31-A0 pins along with R/W = 1. The memory device places the first four bytes from memory on D31-D0 lines. In the last step of Figure 7.38d, the MC68020 outputs 111 on FC2FC1FC0.
to be in CPU space and then outputs S00022010 on A31-A0 lines along with R/W line LOW to access the MC68881 operand and register (low 5 bits of the address lines A0-A4 specify the coprocessor interface register and A4-A0 = 10 indicates MC68881 operand register). The MC68881 asserts DSACKX, takes the first 4 bytes of the total 8 bytes, and then stores it in the operand register. Note that the MC68881 operand register is a 32-bit read/write register and is used by the MC68020 to transfer data to and from the MC68881.

In Figure 7.38c the MC68020 transfers the remaining 8 bytes of data from (A0) + into the MC68881 operand register. Since the 12-byte data transfer is complete, the MC68881 places the null primitive in its response register. The format for the null primitive is given in Figure 7.40.

![Figure 7.40](326.jpg)

In Figure 7.40 CA = 0 indicates done, FC = 0 means don’t pass, and PC and PF = 1 indicates the MC68881 has finished this instruction. The MC68020 places 1111 on PCF2FC0 and 00002000 on A31-A0 to select the MC68881 response register, along with R/W HIGH.

After asserting DSACKX, the MC68881 places the response register contents S0802 on data bus. The MC68020 reads these data and determines that the coprocessor has completed all processing associated with this instruction.

### Example 7.7
For the following figure, write MC68020 assembly program using floating-point coprocessor instructions: determine X and θ.

![Figure](326.jpg)

Assume X, Y, and Z are 32 bits wide.

#### Solution

- \( \text{MOVE.L } Y, \text{FP2} \) : MOVE Y TO FP2
- \( \text{MOVE.L } Z, \text{FP3} \) : MOVE Z TO FP3
- \( \text{FMADD.L } \text{FP2, FP0} \) : MULT Y TO FP0
- \( \text{FMADD.L } \text{FP3, FP1} \) : MOVE Z TO FP1
- \( \text{FMUL.L } \text{FP0, FP0} \) : \( Y^2 \)
- \( \text{FADD.L } \text{FP1, FP1} \) : \( Z^2 + Y^2 \)
- \( \text{FSQRT.L } \text{FP1} \) : \( Z = \sqrt{Y^2 + Z^2} \)
- \( \text{FDIV.L } \text{FP3, FP2} \) : \( \theta = \tan(Y/Z) \)

STOP

JMP STOP

#### 7.4.2 MC68881 MMU

The MC68881 coprocessor is a demand Paged Memory Management Unit (PMMU) designed to support the MC68020-based virtual memory system. The 68851 is included in a 132-TGA package and can be operated at a frequency of either 12.5 or 16.67 MHz.

The main functions of the 68851 are to provide logical-to-physical address translation, protection mechanism, and to support the 68020 breakpoint operations.

The 68851 translates a logical address comprised of a 32-bit address and a 4-bit function code. The 68851 has four function code pairs, FC0-FC3, issued by the 68020 into a corresponding 32-bit physical address in
The CPU root pointer points at the translation table tree for the currently executing task; the supervisor root pointer points to the operating system translation table; and the DMA root pointer points to a DMA controller’s (if present in the system) translation table.

All addresses contained in the translation tables are physical addresses. In Figure 7.41, table entries at the higher levels of the tree (pointer tables) contain pointers to other tables. Entries at the leaf level (page tables) contain page descriptors. The pointer table lookup normally uses the function code as the index, but they may be suppressed. The 68851 includes the 4-bit bidirectional function code pins, FC0-FC3, which indicate the address space of the current bus cycle. The 4-bit function code consists of the three function code outputs (FC0-FC2) of the 68020 and a fourth bit that indicates that a DMA access is in progress. When the 68851 is busy, it drives the function code pins as outputs with a constant value of FC3-FC0 = 05, indicating the supervisor data space.

The 68851 hierarchical protection mechanism monitors and enforces the protection/privilege mechanism. These may be up to 8 levels with privilege hierarchies, and the upper 3 bits of the incoming logical address define these levels, with level 0 as the highest privilege in the hierarchy and level 7 as the lowest level. Privilege levels of 0, 2, or 4 can also be implemented with the 68851, in which case the access level encoding is included in the upper zero, one, or two logical address lines, respectively. The 68851 access level mechanism, when enabled, compares the access level of the memory logical address with the current access level as defined in the Current Access Level (CAL) register. The current access level defines the highest privilege level that a task may execute at that time.

If the privilege level provided by the bus cycle is more privileged than allowed, then the 68851 terminates this access as a fault.

In the 68851 protection mechanism, the privilege level of a task is defined by its access level. Smaller values for access levels specify higher privilege levels. In order to access program and/or data requiring a higher privilege level than the level of the current task, the 68851 provides CALLM (call module) and RTM (return from module) instructions. These instructions allow a program to call a module operating at the same or higher privilege level and to return from that module after completing the module function.

The 68851 provides a breakpoint acknowledge facility to support the 68020. When the 68020 executes a breakpoint instruction, it creates a breakpoint acknowledge cycle and reads a predefined address in the CPU space cycle. The 68851 decodes this address and responds by either providing a replacement op code for the breakpoint op code and asserting DS/BEKR inputs or by asserting 68020 BEXR input to execute illegal...
instruction exception processing. The 68851 can be programmed to provide (1) the replacement op code n times \(1 \leq n \leq 255\) in a loop and then assert BERR or (2) assert BERR on every breakpoint.

The 68851 instructions provide access to the 68020 instruction set via the coprocessor interface. These instructions provide:

1. Loading and storing of MMU registers
2. MNUs control functions
3. Access rights and conditionals checking

For example, the PMOVE instruction moves data to or from the 68851 registers using all 68020 addressing modes. PVALID compares the access level bits of an incoming logical address with those of the Valid-Access Level (VAL) register and traps if the address bits are less. PTEST searches the ATC and translation tables for an entry corresponding to a specific address and function code. The results of the test are placed in the 68851 status register which can be tested by various conditional branch and set instructions.

Optionally, the PTEST instruction can obtain the address of the page descriptor. A companion instruction, PLOAD, takes a logical address and function code, searches the translation table, and loads the ATC with an entry to translate the logical address.

PLOAD can be used to load the ATC before starting the memory transfer. This can speed up a DMA operation. PFLUSH and its variations clear the ATC of either all entries, entries with a specified function code, or those limited to a particular function code and logical address.

PSAVE saves the contents of any register that reflects the current task's state and the internal state of the 68851 dealing with coprocessor and module call operations. PRESTORE restores the information saved by PSAVE. PSAVE and PRESTORE permit the context of the 68851 to be switched.

The PBcc, PDcc, PDcc, and PTRAPcc instructions have the same meaning as those of the 68020 except that the conditions are based on the 68851 condition codes.

Figure 7.42 shows a 68020-based microcomputer system which interfaces 68831 and 68851 chips to the 68020. The 68851 MMU is placed between the logical and physical address buses. The 68851 allows interfacing memory, disk controller, and serial I/O devices to the 68020.

Questions and Problems

FIGURE 7.42 A 68020-based microcomputer with coprocessors.

QUESTIONS AND PROBLEMS

7.1 What is the purpose of affected 68020 CALL and RTM instructions?

7.2 Determine the contents of 68020 register, memory locations, and condition code register after execution of CAS.B D2, D4, (A0). Assume the following data prior to execution of the instruction:

\[
\begin{align*}
[D2] &= 0 \times 0001002 \\
[D4] &= 0 \times 0001000 \\
[AS] &= 0 \times 0001004 - 0 \times 0004 \\
X, N, Z, V, C &= 1, 1, 1, 1, 0 \\
OCR &= \text{111, 010} \\
\end{align*}
\]
7.3 Determine the contents of 68020 registers and memory locations after execution of CAS2.W D5:6, D0:20, (A5): (A6). Assume the following data prior to execution of CAS2:

\[
\begin{align*}
[D0] &= 00000000 \\
[D5] &= 00000000 \\
[D6] &= 00000000 \\
[D1] &= 00000000
\end{align*}
\]

Memory

\[
\begin{array}{cccc}
15 & 14 & 13 & 12 \\
11 & 10 & 9 & 8 \\
7 & 6 & 5 & 4 \\
6 & 5 & 4 & 3 \\
5 & 4 & 3 & 2 \\
4 & 3 & 2 & 1 \\
3 & 2 & 1 & 0 \\
2 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

7.4 i) Name two exception vectors for the MC68851 and the MC68881.  
ii) What is the size of the 68020 on-chip cache?  
iii) What is the 68020 cache access time?

7.5 What are the values of SIZ1, SIZ0, FC2, FC1, FC0, R/W, and A31-A0 pins after execution of the 68020 BKPT #3 instruction?

7.6 Determine the values of FC2, FC1, FC0, and A31-A0 pins for a 68020 CPU space cycle with a floating-point coprocessor command register being accessed.

7.7 Assume a 68020/68881 system. Write a program in 68020 assembly language to find

\[
\sqrt{XY}
\]

Assume X and Y are 32-bit integers.

7.8 Determine the values of 68020 operation word, command register, function code, R/W, and that address bus when the 68881 asserts DSACK to accept the command after execution of the following instruction:

\[
FADD.L D2, D0
\]

Questions and Problems

Use the following data:

\[
\begin{align*}
\text{FADD} & & \text{op class} & & \text{010010}_2 \\
\text{L} & & \text{operand mode} & & \text{000}_2
\end{align*}
\]

7.9 i) Write a 68020 instruction sequence using CAS2 to insert an element in a double-linked list.  
ii) Write a 68020 instruction sequence using CAS2 to delete the last entry in a linked list. Note that a double-linked list needs two controlled locations which point to the last element inserted in the list and the next element to be removed. If the list is empty, both pointers are NULL (0). Therefore, a double-linked list maintains a forward link and a reverse link.

7.10 i) What are the main functions of the 68851 MMU?  
ii) Summarize the address translation and protection mechanism of the 68851.  
iii) How does the 68851 support the 68020 breakpoint function?
Chapter 8

MOTOROLA MC68030 AND MC88100

This chapter provides an overview of the hardware, software, and interfacing features associated with the MC68030 and MC88100.

The MC68030 is a 32-bit virtual memory microprocessor based on the MC68020, while the MC88100 is a 32-bit RISC (Reduced Instruction Set Computer) microprocessor.

8.1 MOTOROLA MC68030

The MC68030 is a virtual memory microprocessor based on an MC68020 core with additional features. The MC68030 is designed by using ECMOS technology and can be operated at 16.67, 20, and 33-MHz clocks.

The MC68030 contains all features of the MC68020, plus some additional features. The basic differences between the MC68020 and MC68030 are listed below:

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>MC68020</th>
<th>MC68030</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-chip cache</td>
<td>256-byte instruction cache</td>
<td>256-byte instruction cache and 256-byte data cache</td>
</tr>
<tr>
<td>On-chip Memory</td>
<td>None</td>
<td>Paged data memory management (demand page of the MC68851)</td>
</tr>
<tr>
<td>Management Unit (MMU)</td>
<td></td>
<td>105 (four new instructions are for the chip MMU)</td>
</tr>
<tr>
<td>Instruction set</td>
<td>101</td>
<td></td>
</tr>
</tbody>
</table>
Like the MC68020, the MC68030 also supports 7 data types and 18 addressing modes. The MC68030 I/O is identical to the MC68020. The MC68020 enhancements such as instruction cache and MMU, along with the basic MC68030 features, are described in the following section.

8.1.1 MC68030 BLOCK DIAGRAM

Figure 8.1 shows the MC68030 block diagram and includes the major sections of the processor.

The bus controller includes all the logic for performing bus control functions and also contains the multiplexer for dynamic bus sizing. It controls data transfer between the MC68030 and memory or I/O device at the physical address.

The control section contains the execution unit and associated logic. Program Logic Arrays (PLAs) are utilized for instruction decode and sequencing.

The instruction and data cache units operate independently. They obtain information from the bus controller for future use. Each cache has its own address and data buses and thus permits simultaneous access. Both the caches are organized as 64-long word entries (256 bytes) with a block size of four long words. The data cache uses a write-through policy with no write allocation on cache misses.

The memory management unit maps address for page sizes from 256 bytes to 32K bytes. Mapping information stored in descriptors resides in translation tables in memory that are automatically searched by the MC68030 on demand. Most recently used descriptors are maintained in a 22-entry

fully associative cache called the Address Translation Cache (ATC) in the MMU, permitting address translation and other MC68020 functions to occur simultaneously. The MMU utilizes the ATC to translate the logic address generated by the MC68030 into a physical address.

8.1.2 MC68030 PROGRAMMING MODEL

Figure 8.2 shows the MC68030 programming model.

The additional registers implemented in the 68030 beyond those of the 68020 are for supporting the MMU features. All common registers in the 68030 are the same as the 68020, except the cache control register which has additional control bits for the data cache and other new cache functions.

The 68030 additional registers are:

- 32-bit translation control register (TC)
- 64-bit CPU root pointer (CRP)
- 64-bit supervisor root pointer (SRP)
- 32-bit transparent translation registers T0 and T1
- 16-bit MMU status register, MMUSR

The TC includes several fields that control address translation. These fields enable and disable address translation, enable and disable the use of SRP for the supervisor address space, and select or ignore the function codes in translating addresses. Other TC fields specify memory page sizes, the number of address bits used in translation, and the translation table structure.

The CRP holds a pointer to the root of the translation tree for the currently executing 68030 task. This tree includes the mapping information for the task's address space.

The SRP holds a pointer to the root of the translation tree for the supervisor's address space when the 68030 is configured to provide a separate address space for the supervisor programs.

Registers T0 and T1 can each specify separate memory blocks as directly-addressable without address translation. Logic addresses in these areas are the same as physical addresses for memory access. Therefore, registers T0 and T1 provide fast movement of larger blocks of data in memory or I/O space, since delays associated with the translation scheme are not encountered. This feature is useful in graphics and real-time applications.

The MMUSR register includes memory management status information resulting from a search of the address translation cache or the translation tree for a particular logical address.
8.1.3 MC68030 DATA TYPES, ADDRESSING MODES, AND INSTRUCTIONS

Like the MC68020, seven basic data types are supported on the MC68030:

- Bits
- Bit fields
- BCD digits
- Byte integers (8 bits)
- Word integers (16 bits)
- Long word integers (32 bits)
- Quad word integers (64 bits)

The 18 addressing modes of the MC68020 are also supported by the MC68030.

The MC68030 includes all MC68020 instructions, plus a subset of the MC68851 (PMU) instructions. These instructions (Table 8.1) include PMOVE, PTTEST, PLOAD, and PFLUSH, and they are compatible with the corresponding instructions in the MC68851 PMU. The MC68020 requires the MC68851 coprocessor interface to execute these instructions. These instructions are executed by the MC68030 just like all other instructions. All the MMU instructions are privileged and do not affect the condition codes. These new instructions are explained in the following sections.

**TABLE 8.1**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opernd syntax</th>
<th>Opernd size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFLUSH</td>
<td>(FC),#mask</td>
<td>None</td>
<td>Invalidate ATC entries at effective address</td>
</tr>
<tr>
<td></td>
<td>[EA]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLOAD</td>
<td>(FC),[EA],[RAW]</td>
<td>None</td>
<td>Create ATC entry for effective address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMOVE</td>
<td>Rn,[EA]</td>
<td>16, 32</td>
<td>Register n → destination</td>
</tr>
<tr>
<td></td>
<td>(EA),Rn</td>
<td>16, 32</td>
<td>Source → register n</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Information about logical address → PMU status</td>
</tr>
<tr>
<td>PTST</td>
<td>(FC),[EA],#level</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[RW],[An]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8.1.3a PMOVE Rn, (EA) or (EA), Rn
Rn can be any MMU register. (EA) uses control alterable addressing mode. The operand size depends on the MMU registers used as follows:

| CRP, SRP | Quad word (64-bit) |
| QC, TT  | Long word (32-bit)  |
| MMUSR  | Word (16-bit)       |

The PMOVE instruction moves data to and from the MMU registers. This instruction is normally used to initialize the MMU registers and to read the contents of the MMUSR for determining a fault.

As an example, consider PMOVE (AS), SRP. This instruction moves a 64-bit word pointed to by AS to the supervisor root pointer.

8.1.3b PTEST
The PTEST has four forms:

- PTESTR (function code), (EA), #level
- PTESTR (function code), (EA), #level, An
- PTESTW (function code), (EA), #level
- PTESTW (function code), (EA), #level, An

The PTEST instruction interrogates the MMU about a logical address and is normally used to determine the cause of a fault. The PTEST instruction executes a table search for the ATC or the translation tables to a specified level for the translation descriptor corresponding to the (EA) and indicates the results of the search in the MMU status register.

The PTESTR or PTESTW means that search is to be done as if the bus cycle is a read or a write. The details of the operand are given below:

- The function code is specified in one of the following ways:
  - Immediate (three bits in the command word)
  - Data register (three least-significant bits of the data register specified in the instruction)
  - Source function code register
  - Destination function code register

- The (EA) operand provides the address to be tested.
- The level operand defines the maximum depth of the table or number of descriptors to be searched. Level 0 means searching ATC only while levels 1 to 7 indicate searching the translation tables only.

Execution of the PTEST instruction continues to the registry level until the descriptor, limit violation, or bus error assertion occurs.

When the address register, An, is specified for a translation table search, the physical address of the last table entry (last descriptor) fetched is loaded into the address register.

The MMUSR includes the results of the search. Table 8.2 shows the values in the fields of the MMUSR for an ATC search.

As an example of PTEST, consider

PTEST SFC, (A3), #4, A5

The function code for the page is in the Source Function Code register, SFC. The content of A3 is the logical address and the search is to be extended to 4 levels. Search is to be done as if the bus cycle is a read and the physical address of the last entry checked is to go to A5.

The PTEST instruction is unsized and the condition codes are unaffected, but the MMUSR contents are changed.

8.1.3c PLOAD
The PLOAD instruction loads an entry into the ATC. This is normally used in demand paging systems to load the descriptors into the ATC before returning to execute the instruction that caused the page fault.

The two forms for the instruction are:

- PLOADR (function code), (EA)
- PLOADW (function code), (EA)

The function code is specified in one of the following ways:

- Immediate
- Data register
- Source function code register
- Destination function code register

(EA) can be control alterable addressing modes only. The PLOAD instruction searches the ATC for (EA) and also searches the translation table for (EA). It is used to load a descriptor from the translation tables to the address translation cache.

The condition codes and MMUSR contents are unaffected.

As an example, consider PLOADR SFC, (A5). The function code for
TABLE 8.2
MMUSER Fields for an ATC Search

<table>
<thead>
<tr>
<th>MMUSER bit</th>
<th>PTTEST, Level 0</th>
<th>PTTEST, Level 1-7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus error (B)</td>
<td>This bit is set if the bus error bit is set in the ATC entry for the specified logical address.</td>
<td>This bit is set if a bus error is encountered during the table search for the PTTEST instruction.</td>
</tr>
<tr>
<td>Limit (L)</td>
<td>This bit is cleared.</td>
<td>This bit is set if an index exceeds a limit during the table search.</td>
</tr>
<tr>
<td>Supervisor violation (S)</td>
<td>This bit is cleared.</td>
<td>This bit is set if the 5th bit of a long format table descriptor or long format page descriptor encountered during the search is set, and the FC0 bit of the function code specified by the PTTEST instruction is not equal to one. The 5th bit is undefined if the 1 bit is set.</td>
</tr>
</tbody>
</table>

Write protected (W) | The bit is set if the WP bit of the ATC entry is set; it is undefined if the 1 bit is set. | This bit is set if a descriptor or page descriptor is encountered with the WP bit set during the table search, the W bit is undefined if the 1 bit is set. |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid (I)</td>
<td>This bit indicates an invalid translation; the 1 bit is set if the translation for the specified logical address is not resident in the ATC, or if the B bit of the corresponding ATC entry is set.</td>
<td>This bit indicates an invalid translation, the 1 bit is set if the DT field of a table or a page descriptor encountered during the search is set to invalid, or if either the B or L bits of the MMUSER are set during the table search.</td>
</tr>
<tr>
<td>Modified (M)</td>
<td>This bit is set if the ATC entry corresponding to the specified address has the modified bit set; it is undefined if the 1 bit is set.</td>
<td>This bit is set if the page descriptor for the specified address has the modified bit set; it is undefined if the 1 bit is set.</td>
</tr>
<tr>
<td>Transparent (T)</td>
<td>This bit is set if a match occurred in either (or both) of the transparent translation registers (700 or T7).</td>
<td>This bit is set to zero.</td>
</tr>
<tr>
<td>Number of levels (N)</td>
<td>This 3-bit field is cleared to zero.</td>
<td>This 3-bit field contains the actual number of tables accessed during the search.</td>
</tr>
</tbody>
</table>

The desired page is in SFC. The logical address for which the descriptor is desired is in A5 and the descriptor is to be loaded as for a read bus cycle.

8.1.3d PFLUSH
The PFLUSH instruction invalidates cache entries. The forms of PFLUSH are

- **PFLUSH (function code), mask**
- **PFLUSH (function code), mask, (EA)**

The operands can be specified as follows:

- (Function code) can be immediate (3 bits), data register (least significant 3 bits), SFC, or DFC.
- mask when set to one indicates corresponding bit in function code for matching.
- (EA) can be any one of the control alterable addressing modes.

The instruction is unsized and condition codes and MMUSER contents are unaltered.

The PFLUSH instruction invalidates all ATC entries. When (function code) and mask are specified in the PFLUSH instruction, the instruction invalidates all entries for the specified function code or codes. When the PFLUSH instruction also specifies (EA), the instruction invalidates the page descriptor for that effective address entry in each specified function code.

The mask operand includes three bits corresponding to the function code bits. Each bit in the mask that is set to one means that the corresponding bit by the function code operand applies to the operation. Each bit in the mask with zero value means that a bit of function code operand and the function code that is ignored. As an example, consider PFLUSH #1, 4. The mask operand of 100, causes the instruction to consider only the most significant bit of the function code operand. Since the function code is 001, function codes 000, 001, 010, and 011 are selected.

8.1.4 MC68030 CACHE
The 68030 cache is placed between the processor and the rest of the memory system. The effective memory access time

\[ t_{ac} = h \cdot t_{cache} + (1 - h) \cdot t_{bus} \]

where \( t_{ac} \) = effective system access time, \( t_{cache} \) = cache access time, \( t_{bus} \) =
access time of the rest of the system, and this is found in the cache.

The instruction cache in the 68030 is a 256-byte direct-mapped cache with 16 blocks. Each block contains four long words. Each long word can be accessed independently and thus 64 entries are possible with A1 selecting a correct word during an access. Figure 8.3 shows the 68030 instruction cache.

The index or a line is addressed by address lines A4 through A7 and each entry is selected by A2 and A3. The tag includes address lines A8 through A31 along with FC2.

A0 is not used since instructions must be at even addresses (A0 is always zero).

An entry means that a line has occurred and the valid bit (four valid bits, one for each long word) for the selected entry is set. If an entry miss occurs, the cache entry can be updated with the instructions read from memory. If the cache is disabled, no cache hits or updates will take place, and if the cache is enabled but frozen, hits can occur without updates.

The CACR can be used to clear cache entries and enable or disable the caches. The system hardware can disable the on-chip caches at any time by asserting the CDIS input pin.

Figure 8.4 shows the organization of the 68030 data cache. The data cache is organized in the same way as the instruction cache except that all three function code bits are used to determine a line hit. The data cache can be updated for both read and write. If the data cache is disabled, no hits or updates will occur. However, when the data cache is enabled but frozen, hits can occur and updates for write but not read can take place. The data cache can be updated for both read and write if enabled but not frozen.

The data cache utilizes a write-through policy with no write allocation of data. This means that if a cache hit takes place on a write cycle, both the data cache and the external device are updated with the new data. If a write cycle generates a miss in the data cache, only the external device is updated and data cache entry is replaced or allocated for that address.

Let us now consider some examples of the 68030 cache. Consider Figure 8.5 showing an instruction cache entry hit. The figure shows the CLR.W(A) instruction with op code 4231 at PC + 11CCA29Eh, put in user space (FC2 = 0). The instruction is stored in PC2 bit which is 1 for supervisor space and 0 for user space. The 68030 outputs 0 on FC2 and 11CCA29Eh, on its address pins. Assume that the instruction cache is enabled. Since A3-A0 = 1110, the address bits A3-A2 are 11. This means LW3 in cache is accessed. A3-A6

![Figure 8.3 MC68030 Instruction Cache](image-url)
Chapter 8

Motorola MC68030 and MC88100

3.1 Motorola MC68030

Figure 8.4 MC68030 data cache.

A31 through A8 and FC2 = 0. Therefore, the valid bit is set to one. The
op code 4251₁₄₁₄ is thus read into the cache output register.

Figure 8.5 illustrates instruction cache miss. In this case, the op code
4251₁₄₁₄ for CLR.W(A1) stored at the PC value of 276F₁₄₁₄₄ in supervisor

is 11CCA₂₁₄, which is used as the tag field with A7 through A₄ (₉₄₄₄) as the
index field. A line hit occurs since the tag at index 9 in the cache matches

User Space

3.1 Motorola MC68030
space (FC2 = 1) is to be accessed. The 68030 outputs one on the FC2 pin and 276F1A64 on the A31-A0 pins. A miss occurs since the tag in line 6/057CD2/1 does not match the address bits A31-A8 (276F1A64) and FC2 = 1 does not match the function code bit (0), the most significant bit above the tag field. Assume the cache is enabled and not frozen. The 68030 reads 4251 µ into L/W1 from external memory and updates the FC2 field and tag field with 1 and 276F1A64, respectively. The valid bits are updated as 0100, Figure 8.7 shows an example of data cache hit. The 68030 executes MOVE.W(A1), D0 to read data C28F32 at address 75B4A176A0 pointed to by A1 in user data space FC2FC1FC00 = 001. Note that the most significant column shows the value on the function code pins. Since A3A2 = 01, L/W1 is accessed with tag value of 75B4A176A0 and index value of 7. Assume that the cache is enabled. Since the tag value in the cache matches A31-A8 and the function code values (FC2 FC1 FC0 = 001) match the function code field value of 1 in the cache, a cache hit occurs. The valid bit is set to one and datanum C28F32 is placed in the cache output register. Figure 8.8 shows an example of data cache miss. The 68030 executes the instruction MOVE.W(A1), D0 to read the contents of 01F37638A0 pointed to by A1 into D0 in the supervisor data space (FC2 FC1 FC0 = 101). Since A3A2 = 10A0, L/W2 in the cache is accessed. Assume the cache is enabled but not frozen. Since the function code pins in cache do not match the 68030 function codes, cache miss occurs. Valid bit 2 is set to one. The 68030 obtains datum 1576C1 from external memory into low word of D0 and then updates the function code and tag fields of the cache. The 68030 then invalidates LW0, LW1, LW3 and validates LW2 by writing 0010 in the valid bits.

8.1.5 68030 PINS AND SIGNALS

The 68030 is housed in a 13 x 13 PGA package for 16.67 MHz and RC SoftX Package for 20 MHz.

Figure 8.9 shows the 68030 pin diagram. Figure 8.10 shows the 68030 functional signal groups. Table 8.8 summarizes the signal descriptions.

8.1.6 MC68030 READ AND WRITE TIMING DIAGRAMS

The MC68030 provides three ways of data transfer between itself and the peripherals. These are:

- Asynchronous transfer
- Synchronous transfer
- BURST mode transfer
DSACKs, BERR and HALT. The synchronous bus cycles of the MC68030 are similar to those of the MC68020. The MC68030 can transfer data in a minimum of three clock cycles. The dynamic bus sizing using the DSACKs signals can determine the amount of data transferred on a cycle-by-cycle basis.

Synchronous bus cycles are terminated with the STERM (synchronous termination) signal and always transfer 32-bit data in a minimum of two clock cycles. The cycles terminated with the DSACKs signals can also be classified as synchronous if the signals are interpreted relative to clock.
### TABLE 8.3 Signal Description

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Mnemonic</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function codes</td>
<td>FCO-FC2</td>
<td>3-bit function code used to identify the address page of each bus cycle</td>
</tr>
<tr>
<td>Address bus</td>
<td>A0-A31</td>
<td>32-bit address bus used to address any of 4,294,967,296 bytes</td>
</tr>
<tr>
<td>Data bus</td>
<td>D0-D31</td>
<td>32-bit data bus used to transfer 8, 16, 24, or 32 bits of data per bus cycle</td>
</tr>
<tr>
<td>Size</td>
<td>SIZE/AL1</td>
<td>Indicates the number of bytes remaining to be transferred for this cycle; these signals, together with A0 and A1, define the active sections of the data bus</td>
</tr>
<tr>
<td>Operand cycle start</td>
<td>OCS</td>
<td>Identical operation to that of ECS except that OCS is asserted only during the first bus cycle of an operand transfer</td>
</tr>
<tr>
<td>External cycle start</td>
<td>ECS</td>
<td>Provides an indication that a bus cycle is beginning</td>
</tr>
<tr>
<td>Read/Write</td>
<td>R/W</td>
<td>Defines the bus transfer as an MPU read or write</td>
</tr>
<tr>
<td>Read-modify-write cycle</td>
<td>R/M/W</td>
<td>Provides an indication that the current bus cycle is part of an indivisible read-modify-write operation</td>
</tr>
<tr>
<td>Address strobe</td>
<td>AS</td>
<td>Indicates that a valid address is on the bus</td>
</tr>
<tr>
<td>Data strobe</td>
<td>DS</td>
<td>Indicates that valid data is to be placed on the data bus by an external device or has been placed on the data bus by the MC68020</td>
</tr>
<tr>
<td>Data buffer enable</td>
<td>DBE/N</td>
<td>Provides an enable signal for external data buffers</td>
</tr>
<tr>
<td>Data transfer and size acknowledge</td>
<td>DACKDV DACKI</td>
<td>Bus response signals that indicate the requested data transfer operation is completed; in addition, these two lines indicate the size of the external bus port on a cycle-by-cycle basis and are used for asynchronous transfers</td>
</tr>
<tr>
<td>Cache inhibit in</td>
<td>CTIN</td>
<td>Prevents data from being loaded into the MC68030</td>
</tr>
<tr>
<td>Cache inhibit out</td>
<td>COUT</td>
<td>Reflects the CI bit in ATC entries; indicates that external caches should ignore these accesses</td>
</tr>
<tr>
<td>Cache burst request</td>
<td>CBREQ</td>
<td>Indicates a burst request for the instruction or data cache</td>
</tr>
<tr>
<td>Cache burst acknowledge</td>
<td>CBACK</td>
<td>Indicates that accessed device can operate in burst mode</td>
</tr>
<tr>
<td>Interrupt priority level</td>
<td>IPL0-IPL2</td>
<td>Provides an encoded interrupt level to the processor</td>
</tr>
<tr>
<td>Interrupt pending</td>
<td>IPEND</td>
<td>Indicates that an interrupt is pending</td>
</tr>
<tr>
<td>Autovector</td>
<td>AVEC</td>
<td>Requests an autovector during an interrupt acknowledge cycle</td>
</tr>
</tbody>
</table>

### TABLE 8.3 (continued) Signal Description

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Mnemonic</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus request</td>
<td>BR</td>
<td>Indicates that an external device requires bus mastership</td>
</tr>
<tr>
<td>Bus grant</td>
<td>BGR</td>
<td>Indicates that an external device may assume bus mastership</td>
</tr>
<tr>
<td>Bus grant acknowledge</td>
<td>BGRACK</td>
<td>Indicates that an external device has assumed bus mastership</td>
</tr>
<tr>
<td>Reset</td>
<td>RESET</td>
<td>System reset; same as 68020</td>
</tr>
<tr>
<td>Halt</td>
<td>HALT</td>
<td>Indicates that the processor should suspend bus activity</td>
</tr>
<tr>
<td>Bus error</td>
<td>BERR</td>
<td>Indicates an invalid or illegal bus operation is being attempted</td>
</tr>
<tr>
<td>Synchronous termination</td>
<td>STERR</td>
<td>Bus response signal that indicates a port size of 32 bits and that data may be latched on the next falling clock edge</td>
</tr>
<tr>
<td>Cache disable</td>
<td>CDIS</td>
<td>Dynamically disables the on-chip cache to assist emulator support</td>
</tr>
<tr>
<td>MMU disable</td>
<td>MMUDIS</td>
<td>Dynamically disables the translation mechanism of the MMU</td>
</tr>
<tr>
<td>Clock</td>
<td>CLK</td>
<td>Clock input to the processor</td>
</tr>
<tr>
<td>Power supply</td>
<td>Vcc</td>
<td>+5 volt ± 5% power supply</td>
</tr>
<tr>
<td>Ground</td>
<td>GND</td>
<td>Ground connection</td>
</tr>
</tbody>
</table>

The cycles that use the synchronous cycles must synchronize the responses to the MC68030 clock in order to be synchronous.

The synchronous cycles terminated with STERR are for 32-bit ports only, while the synchronous cycles terminated by DSACKI can be for 8-, 16-, or 32-bit ports. The main difference between the use of STERR and DSACKI is that STERR can be asserted and data can be transferred earlier than for a synchronous cycle terminated with DSACKI. Wait cycles can be inserted by delaying the assertion of STERR if required.

BURST mode transfer can be used to fill blocks of the instruction and data caches when the MC68030 asserts CBREQ (cache burst request).

BURST mode transfer takes place in synchronous operation and requires assertion of STERR to terminate each of its cycles. BURST mode is enabled by bits in the cache control register (CACR).

As an illustration of 68030 read/time timing diagrams, 68030 longword read and write cycles for synchronous operation will be considered. Figure 8.11 depicts two write cycles (between two read cycles, with no idle time in-between) for a 32-bit device. The timing diagram in Figure 8.11 for read can be explained as follows:

1. The read cycle for an instruction such as MOVE.L (A0), D1 starts in state 0 (S0). The MC68030 drives the external cycle start (ECS) pin
LOW indicating the start of an external cycle. When the cycle is the first external cycle of a read operand, the MC68030 outputs low on the operand cycle start (OCS) pin. The MC68030 then places a valid address (content of register A0 in this case) on A0-A31 pins and valid function codes on FC0-FC2 pins. The MC68030 drives R/W HIGH for read and drives data buffer enable (DBEN) inactive to disable data buffers. SIZ0 and SIZ1 signals become valid, indicating the number of bytes to be transferred. Cache inhibit out (CIOUT) becomes valid, indicating the state of the MMUC bit in the address translation descriptor or in the selected TT main.

2. During S1, the MC68030 activates AS LOW, indicating a valid address on A0-A31. The MC68030 then activates DS LOW and negates ECS and OCS (if asserted).

3. During S2, the Mc68030 activates the DBEN pin to enable external data buffers. The selected device such as a memory chip utilizes DS, SIZ0, SIZ1, R/W, and CIOUT to place data on the data bus. The MC68030 outputs LOW on CIIN if appropriate. Any or all bytes on D0-D31 are selected by SIZ0, SIZ1, A0, and A1. At the same time, the selected device asserts the DSACKs signals. The MC68030 samples DSACKs at the falling edge of the S2 and if DSACKs is recognized, the MC68030 inserts no wait states.

As long as at least one of the valid DSACKs is asserted by the end of S2 (satisfying the asynchronous setup time requirement), the MC68030 latches data on the next falling edge of the clock and ends the cycle. If DSACKs is not recognized by the MC68030 by the beginning of S3, the MC68030 inserts wait states and DSACKs must remain HIGH throughout the asynchronous input setup and hold times around the end of the S2. During MC68030’s wait states, the MC68030 continually samples DSACKs on the falling edge of each of the subsequent cycles until one DSACKs is asserted.

4. With no wait states, at the end of S4, the MC68030 latches data.

5. During S5, the MC68030 negates AS, DS, and DBEN. The MC68030 keeps the address valid during S5 to provide address hold time for memory systems R/W, SIZ0, and SIZ1, and FC0-FC2 also remain valid during S5.

The timing diagram in Figure 8.11 for write can be explained as follows:

1. The MC68030 outputs LOW on both ECS and OCS and places a valid address and function codes on A0-A31 and FC2-FC0, respectively. The MC64030 also places LOW on R/W and validates SIZ0, SIZ1, and CIOUT.

2. In S1, the MC68030 asserts AS and DBEN and negates ECS and OCS (if asserted).

3. During S2, the MC68030 outputs data to be written on D6-D31 and samples DSACKs at the end of S2.

4. During S3, the MC68030 outputs LOW on DS, indicating that the data are stable on data bus. As long as at least one of the valid DSACKs is recognized by the end of S2, the cycle terminates after one cycle. If DSACKs is not recognized by the start of S3, the MC68030 inserts wait states. If wait states are inserted, the MC68030 continues to sample the DSACKs signals on the subsequent falling edge of the
clock, until one of the DSACKs is recognized. The selected device such as memory utilizes R/W, D3, SIZ0, SIZ1, and A0 and A1 to latch data from the appropriate portion of D0-D31 pins. The MC68030 generates no new control signals during S4.

5. During S5, the MC68030 negates AS and D3. The MC68030 holds the address and data valid to provide address hold time for memory systems. The processor also keeps R/W, SIZ0, SIZ1, FC0-FC2, and DBEN valid during S5.

8.1.7 MC68030 ON-CHIP MEMORY MANAGEMENT UNIT

8.1.7a MMU Basics

A Memory Management Unit (MMU) translates addresses from the microprocessor (logical) to physical addresses. Logical addresses are assigned to the task when it is linked, while physical addresses are assigned at the time the task is loaded into memory based on free physical addresses. The MMU keeps a task in its own address space. If a task attempts to go out of its own address space, the MMU asserts bus error. Besides protection, this feature is valuable in demand paging systems.

An operating system must know the free physical memory addresses available so that it can load the next task to these spaces. One way of accomplishing this is by dividing the memory into contiguous blocks of equal size. These are called pages on the logical side of the MMU and page frames on the physical side. The system memory will contain page descriptions which point to the page frames and status of the page frames.

The page size determines the number of address bits to be translated by the MMU and the number which addresses memory directly. The lower bits of the logical address related to the page size are not translated by the MMU and directly go to the physical address bus as shown in Figure 8.12. Note that A8-A3 bits provide the page number and A0-A7 bits define the 256-byte page size in this case.

![Figure 8.12 Logical to physical translation.](image)

A translation table can be used to translate pages to page frames. Figure 8.13 shows an example of table translation of pages. The task control block contains a pointer to the starting address of the translation table. A logical address is translated by accessing the location in the translation table determined by starting address of translation + (page number * entry size).

The entry accessed contains the page frame number. Note that entry size is 4 for 32 bits (4 bytes). Each task has a translation table. Single or multiple translation tables can be used. Figure 8.14 shows an example of a single pointer level translation. The TCB contains $00010000$ containing the starting address of the translation table. The 12-bit page size ($$CS$$ for P2) replaces the low 12 bits of the physical address directly. The logical address for P2 is translated by accessing the translation table at the location.
Logical address translation example.

**Logical Address**

\[ \text{Page Number} + \text{Page Size} \]

**Example**

- Logical Address
  - Physical Address

**Translation Examples**

- Logical: $1000$
  - Physical: $E678$
- Logical: $1234$
  - Physical: $F678$
- Logical: $5678$
  - Physical: $E678$

If a task has only 2 pages, $0$ and $FFFF$, the page descriptor table must be the same size as if the task used all pages. Size of a page frame is 4 bytes, $0$ to $FFFF x 4 = 4194304$ bytes.

Each task has an allocation table.

**Figure 8.14** Single-pointer level translation.

- $10000 + \text{(Page number} \times \text{entry size})$
- $10000 + (00001 \times 4)$
- $100004$

Therefore, location $100004$ in the translation table is accessed and its content $30271$ obtained as the page frame number. The 12-bit page size $30271$ is concatenated with the page frame number to obtain the physical address for page $P2$ as $E678$.

One of the disadvantages of the single-level translation is that for most systems, the required sizes would be too large. Therefore, multilevel translation tables are used. For example, consider the double pointer level translation table of Figure 8.15. In this case, the TCB contains a pointer to the starting address of the level 1 translation table. Level 1 descriptors contain pointers to level 2 page descriptors.

A logical address is translated in two steps:

1. A pointer is obtained from level 1 translation table as follows: pointer from level 1 table = starting address of level 1 translation table + (level 1 pointer size of level 1 table).
2. This pointer is used to access a location in level 2 table by adding the pointer obtained from level 1 table with (level 2 pointer size of level 2 table).
The location in table 2 thus obtained contains the page frame number. Consider the logical address:

```
<table>
<thead>
<tr>
<th>Page Number</th>
<th>11</th>
<th>22.21</th>
<th>12.11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>001</td>
<td>003</td>
<td>ABC</td>
<td></td>
</tr>
</tbody>
</table>
```

Level 1: 000000000012
Level 2: 000000000013

Page Number

If TCB contains ($10000) point to the starting address of table 1, level 1 location

\[
\text{location} = \$10000 + \text{level 1} \times \text{entry size of level 1} = \$10000 + 1 \times 4 = \$10004
\]

Therefore, if the content of $10004 is $21000 pointing to the starting address of level 2 table, then the accessed location in table 2

\[
\text{location} = \$21000 + (36 \times 4) = \$21000 + 144 = \$2100C
\]

Assume the content of $2100C is $25719. Then $25719 is concatenated with the page size $ABCD to obtain the 32-bit physical address $25719ABCD. An example of the double-pointer level translation is shown at the top of the next page.

Assume a page size of 4K bytes. Let us translate the following logical addresses to physical addresses for the task shown above:

\[
\text{level 1 location: } \$0000072A \text{ and } \$00401A59
\]

Consider the logical address $0000072A. This address has level 1 and level 2 page numbers of 0. This means that level 1 location

\[
\text{location} = \$00780000 + 0 \times 4 = \$00780000
\]

The content of location $00800000 ($176A5) is concatenated with the page size $72A so that the physical address is $176A572A.

Next, consider the logical address $00401A59. The uppermost 10 bits (0000 000001) define the level 1 page number as one. The next 10 bits (0000 000001) define the level 2 page number as one. The page size is $A59, Level 1 location

\[
\text{location} = \$7800 0000 + (1 \times 4) = \$7800 0004
\]
The content of location $7800 0004 (S00801000) points to the starting address of level 2 table. Level 2 location

\[ \text{Level 2 location} = S00800100 + 1 \times 4 \]
\[ = S00801004 \]
\[ = S0080 1004 \]

The content of location $0080 1004 (S91024) is concatenated with the page size A59 to obtain the physical address S91024 A59.

The main advantage of the multiple translation table is that it reduces the required translation table size significantly. However, multiple memory access (two in the example of two-level translation) is required to obtain a descriptor. However, address translation cache can be used to speed up memory access.

An MMU provides three basic functions:

- Translates page number to page frame number
- Restricted access, i.e., a task, is restricted to its own address space
- Provides write protection by not allowing write access to write-protected pages

The status information is included in the low bits of the translation table entry data to provide protection information.

8.1.7.8 68030 On-chip MMU

Figure 8.16 provides a block diagram of the MC68030 and identifies the on-chip MMU.

The 68030 on-chip MMU translates logical addresses to physical addresses. If the desired information is in the address translation cache, an

time delay occurs due to MMU address translation. The page descriptor is obtained by the MMU by searching the translation tables if needed.

The pins used by the 68030 on-chip MMU are A31-A0, CIOUT, and MMUDIS.

The MMU outputs the translated physical addresses (from logical addresses) or the addresses for fetching translation data (when a table is searched) on the A31-A0 pins. The MMU asserts CIOUT pin for pages which are defined as noncacheable. The MMUDIS pin, when asserted by an external emulator, disables the MMU.

The 68030 executes a normal bus cycle when the address translation information is in the ATC. However, if the address translation information is not in the ATC, the 68030 executes additional bus cycles to obtain the desired address translation information.

The 68030 includes three main elements in its MMU: a set of registers, ATC, and table search logic.

Figure 8.17 provides the MMU registers.

![Figure 8.16 MC68030 on-chip MMU block diagram.](image)

![Figure 8.17 MC68030 MMU registers.](image)
The CRF or SRP points to the beginning of the task translation table and supervisor table (if enabled), respectively. The translation control register controls the MMU functions such as MMU enable.

The transparent translation registers output specified logical addresses on the A31-A0 pins.

The MMUSR provides the results of execution of the MMU instructions PMOVE (EA), MMUSR and PTEST. The MMUSR stores memory management status information resulting from a search of the address translation cache, or the translation tree, for a particular logical address.

As mentioned before, four MMU instructions — PMOVE, PTEST, PLOAD, and PFLUSH — are included in the 68030 instruction set. The MMU provides up to 5 levels of address translation tables. Address translation starts with the contents of the root pointers CRF or SRP.

Figure 8.18 provides a typical 5-level table translation scheme.

Figure 8.19 provides the ATC block diagram along with a simplified flowchart for the physical address translation.

The ATC is a content-addressable, fully associative cache with up to 22 descriptors. The ATC stores recently used descriptors so that table search is not required if future accesses are required.

There are six types of descriptors used in MMU operation. These are ATC page descriptors, table descriptors, initial termination page descriptors, invalid descriptor, and indirect descriptor. Some descriptors have a long and a short form.

Figure 8.20 shows the page descriptor summary.

Each ATC entry consists of a logical address and information from a corresponding page descriptor. The 28-bit logical or tag portion of each entry consists of three fields. These are the valid bit field (bit 27), function code field (bits 26-24), and 24-bit logical address field (bits 0-23). The V-bit indicates that the entry is valid if V = 1. The function code field includes the function code bits (FC0-FC2) corresponding to the logical address in this entry. The 24-bit logical address includes the most significant logical address bits for this entry. All 24 bits are used in comparing the entry to an incoming logical address when the page size is 256 bytes. For larger page sizes, some least significant bits of this field are ignored.

Table descriptors have short (32-bit) and long (64-bit) forms. In the 32-bit short form, the table descriptor includes four status bits (bits 0-3) and a 28-bit table address. The status bits provide information such as write protection and descriptor type.

The table address contains the 28-bit physical base address of a table of descriptors. The long table descriptor includes a 28-bit table address, a 16-bit status information, and a 16-bit limit field. The status bits in the long form include additional information such as whether the table is a supervisor-only table. The limit field includes a limit to which the index portion of an address is compared to detect an out-of-bounds index.

The page descriptors also have short and long formats. The short page format is identical to the short table format except that for page tables, page table and status information related to pages are used. The long page format...
The address translation cache (ATCC) and flowchart for translation:

- This is a simplified flowchart.

**FIGURE 8.19** Address translation cache (ATCC) and flowchart for translation.

descriptor contains a 24-bit page address and 16-bit status information. The status bits provide information such as write protect, descriptor-type, and identification of a modified page.

The early termination page descriptor (both short and long form) includes the page descriptor identification bits in the descriptor-type field of the status bits, but the descriptor resides in a pointer table. This means that the table in which an early termination page descriptor is located is not at the bottom level of the address translation tree. The invalid descriptors can also be short and long forms. These descriptors only contain the two descriptor-type status bits and are used with long or short page and table descriptors.

The indirect descriptors also include short and long forms. They contain physical address of a page descriptor and two descriptor-type bits which identify an indirect descriptor that point to a short or long format page descriptor.

Tables 8.4 summarizes differences between 68030 on-chip MMU and 68881.

Now let us discuss the details of the 68030 MMU registers. The CRP points to the first level translation table. It is normally loaded during a context (task) switch. The ATC is automatically flushed when the CRP is loaded. Figure 8.21 shows the details of the CRP.

The limit field in the CRP limits the size of the next table. That is, it limits the size of the table index field. If the limit is exceeded during a table search, an ATC entry is created with bus error set. Two fields are assigned to the limit. These are L/U and limits. L/U = 1 means a lower range limit, while L/U = 0 indicates an upper range limit. If L/U = 0 and limit = 57FF or L/U = 1 and limit = 0, the limit function is suppressed. The translation control register (TC) permits the user to control the translation process.
TABLE 8.4
68030 On-Chip MMU vs. 68851

<table>
<thead>
<tr>
<th>Feature</th>
<th>68030</th>
<th>68851</th>
</tr>
</thead>
<tbody>
<tr>
<td>No access level (no CALLM or RTM Instructions)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No breakpoint registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No DMA root pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No task aliasing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22 entry ATC instead of 64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No lockable content-addressable memory (CAM) entries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No shared globally entities</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions supported</td>
<td>PFLUSH, PFLUSH, PMOVE, PTEST, PLOAD</td>
<td></td>
</tr>
<tr>
<td>Instructions not supported (F-line trap)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PVALID, PFLUSHR, PFLUSHES, P8cc, PDBcc, PSScc, PTRAPcc, PSAVE, PRESTORE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control alterable effective addresses only for the MMU Instructions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CRP</th>
<th>Limit</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>15</td>
</tr>
</tbody>
</table>

* Status Field:

| 15   | 21   | 0       |

* CT Values:

| 00   | Not allowed, if loaded with this value, an MMU configuration error exception occurs. |

Table address field is a page descriptor.

Table address field points to descriptors which are 4 bytes long.

Table address field points to descriptors which are 8 bytes long.

FIGURE 8.21 CRP details.

of the access. TC is also used to enable and disable the main function of the MMU. Figure 8.22 shows the details of TC:

FIGURE 8.22 TC details.

The SRP is similar to the CRP except that it is used only for supervisor access. This register must be enabled in the TC before use. The format of SRP is shown in Figure 8.23.

FIGURE 8.23 SRP form.

If a task is required to access physical memory directly, such as in graphics applications, TTD or TTI can be used to specify the physical areas of memory to be accessed.

Figure 8.24 provides formats for TTD and TTI.
The information in the MMU status register is the result of execution of the PTEST instruction.

- Level = 0 Search the ATC only
- Level = 0 Search translation tables only

<table>
<thead>
<tr>
<th>Bit Nianing</th>
<th>Level = 0</th>
<th>Level = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Bus error</td>
<td>Bus error</td>
</tr>
<tr>
<td></td>
<td>is set in</td>
<td>occurred</td>
</tr>
<tr>
<td></td>
<td>matching ATC</td>
<td>during table work</td>
</tr>
<tr>
<td>L</td>
<td>Limit bit</td>
<td>Always cleared</td>
</tr>
<tr>
<td></td>
<td>is set in</td>
<td>is descriptor</td>
</tr>
<tr>
<td></td>
<td>descriptor</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Supervisor violation</td>
<td>Supervisor bit is set in descriptor</td>
</tr>
<tr>
<td>W</td>
<td>Write protect</td>
<td>Address is write protected</td>
</tr>
<tr>
<td></td>
<td>is set in the memory table</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>Invalid bit</td>
<td>No entry in ATC or B is set</td>
</tr>
<tr>
<td></td>
<td>Page decoder has</td>
<td>No translation in table, or B or L is set</td>
</tr>
<tr>
<td>M</td>
<td>Modified bit</td>
<td>ATC entry has M bit set</td>
</tr>
<tr>
<td></td>
<td>Page decoder has</td>
<td>M bit set</td>
</tr>
<tr>
<td>T</td>
<td>Transparent</td>
<td>Address is within range of TTO or TFI</td>
</tr>
<tr>
<td></td>
<td>is set in decoder</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Number of tables used in translation</td>
<td>Number of tables used in translation</td>
</tr>
</tbody>
</table>

in the table indexed by the logical address can be set from one to four and up to 15 logical address bits can be used as an index at each level. One main advantage of this tree structure is to deallocate large portions of the logical address space with a single entry at the higher levels of the tree.

The entries in the translation tables include status information with respect to the pointer for the next level of lookup or the pages themselves. These bits can be used to designate certain pages or blocks of pages as register-only, write-protected, or noncacheable. The 68030 MMU exceptions include the following:

- Bus error
- F-line
- Privilege violation
- Configuration error

**Figure 8.24** TTO and TFI details.

The ATC entries include two (logical and physical) 28-bit fields. The logical field includes a valid bit, function codes, and page number. The physical field contains four control/status bits and page frame numbers.

Figure 8.26 includes the ATC entry structure.

The translation tables supported by the 68030 contain a tree structure. The root of a translation table tree is pointed to by one or two root pointers.

Table entries at higher levels of the tree contain pointers to other tables, while entries at the leaf level (page tables) contain page descriptors. The technique used for table searches utilizes portions of the logical address as index for each level of the lookup. All addresses contained in the translation table entries are physical addresses.

**Figure 8.27** shows the 68030 MMU translation table tree structure.

The function codes are usually used as an index in the first level of lookup in the table. However, this may be suppressed. In table searching up to 15 of the logical address lines can be ignored. The number of levels

**FIGURE 8.25** AVMSR details.
All of the exceptions except configuration error exception are included in other ways besides the MMU. Table 8.5 summarizes the 68030 MMU exceptions.

**TABLE 8.5**

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Typical cause</th>
<th>Possible response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus error</td>
<td>Bit set in ATC entry due to normal tablewalk or LOAD</td>
<td>Page not in memory</td>
</tr>
<tr>
<td></td>
<td>Invalid descriptor</td>
<td>Load page to memory and return cycle</td>
</tr>
<tr>
<td>Limit violation or range</td>
<td>Page not in memory</td>
<td>Load page to memory and return cycle</td>
</tr>
<tr>
<td>violation</td>
<td>Descriptor fetch</td>
<td>Abort task</td>
</tr>
<tr>
<td>Bus error asserted</td>
<td>accessed a location at which there was no device</td>
<td>Restart task</td>
</tr>
<tr>
<td>User attempt to write to</td>
<td>Bad task</td>
<td>Abort task</td>
</tr>
<tr>
<td>supervisor space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write to a write-protected</td>
<td></td>
<td></td>
</tr>
<tr>
<td>page</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F-line exception</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F-line code with cr1D = 0 was attempted and was not equal to PFLUSH, PLOAD, PMOVE, or PTEST</td>
<td>MC68651 instruction(s)</td>
<td></td>
</tr>
<tr>
<td>Privilege violation</td>
<td></td>
<td>Rewrite instructions or write exception handler to emulate 68851 instructions</td>
</tr>
<tr>
<td>User attempts to execute</td>
<td>Bad task</td>
<td>Abort task</td>
</tr>
<tr>
<td>PFLUSH, PLOAD, PMOVE, or PTEST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration error</td>
<td>System program error</td>
<td>Crash system</td>
</tr>
<tr>
<td>Result of loading invalid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>data into CR, SRP, and TC using</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMOVE CRP, SRP, or TC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid data may be due to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>invalid descriptors</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example 8.1
Assume user program space. The MC68030 executes CLR.W(A1) with PC contents and instruction cache contents as follows.

Given the op code for CLR.W(A1) is 425110, [PC] = 502513080.

Instruction Cache Contents

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid bit</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>256130</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>255130</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>254130</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>253130</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>252130</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

When the instruction CLR.W(A1) is fetched by the 68030, will a cache hit or miss occur? Why?

Solution
A miss will occur since the valid bit is zero.

Example 8.2
Write an instruction sequence to freeze the data cache and disable the instruction cache.

Solution

MOVEC CACR,D1 : Get current CACR
BCLR.L #0,D1 Disable instruction cache
BSET.L #9,D1 Freeze data cache
MOVEC D1,CACR Write to CACR

Example 8.3
Assume a page size of 4K bytes. Determine the physical address from the logical addresses of the task shown below:

\[ \text{TCB} \]

Logical addresses to be translated are 000001A5, 00000270.

Solution
From logical addresses 000001A5, the page number is upper 20 bits, i.e., page number is zero. This is because the page size is given as 4K bytes and hence the lower 12 bits (2^{12} = 4K) are used as the page size. Since TCB address is 0002000, $12345$ is concatenated with the lower 12 bits (51A5) of the logical address $000001A5$ to obtain the physical address $123451A5$.

Similarly, the physical address for the logical address $00000270$ is $11BCA7370$.

Example 8.4
Determine the contents of SRP when enabled in TC to describe a page descriptor table located at 005721050 and limited to logical address 0 - 57900.

Solution
The format for SRP is:

```
31 30 16 15 2 1 0
L/U Limit 0 ... DT
Table Address Unaligned
```

L/U is 0 for upper limit range. LIMIT is $7000$. DT must be 01 for page descriptor. Table address is 005721050. Hence, SRP is

```
31 30 16 15 2 1 0
0 7900 0001
```

Example 8.5
What happens upon execution of the ITEST instruction with level ≠ 0 (i.e., search translation tables only)? Assume that the MMUSR contains
8.2 MOTOROLA MC88100 32-BIT RISC (REDUCED INSTRUCTION SET COMPUTER) MICROPROCESSOR

8.2.1 BASIC FEATURES OF RISC

RISC is an acronym for Reduced Instruction Set Computer. Motorola MC88100 is a 32-bit RISC microprocessor. This type of microprocessor emphasizes simplicity and efficiency. RISC designs start with a necessary and sufficient instruction set. The purpose of using RISC architecture is to maximize speed by reducing clock cycles per instruction. Almost all computation can be obtained from a few simple operations. The goal of RISC architecture is to maximize the effective speed of a design by performing infrequent operations in software and frequent functions in hardware, thus obtaining a net performance gain.

The following summarizes the typical features of a RISC microprocessor:

1. The microprocessor is designed using hardwired control with little or no microcode. Note that variable length instruction formats generally require microcode design. All RISC instructions have fixed formats, and therefore, microcode design is not necessary.
2. RISC microprocessor executes most instructions in a single cycle.
3. The instruction set of RISC microprocessor typically includes only register-to-register, load, and store. All instructions involving arithmetic operations use registers, while load and store operations are utilized to access memory.
4. The instructions have simple fixed format with few addressing modes.
5. RISC microprocessor has several general-purpose registers and large cache memories.
6. RISC microprocessor processes several instructions simultaneously and thus includes pipelining.
7. Software can take advantage of more concurrency. For example, jumps occur after execution of the instruction that follows. This allows fetching of the next instruction during execution of the current instruction.

8.2.2 BASIC FEATURES OF THE 88100 RISC MICROPROCESSOR

MC88100 is a 32-bit microprocessor designed using HCMOS technology. The main features of the 88100 include the following:

- Hardwired control design with no microcodes
- 20 or 25 MHz internal clock frequency
- Housed in 17 x 17 (180 pins used) Pin Grid Array (PGA) with a maximum of 1.78" x 1.78"
- Supports integers, bit field, and floating-point data types
- Includes 51 instructions which include integer arithmetic, floating-point, logical, bit-field, load/store/exchange, and branch
- Four fully parallel execution units (pipelined)
- Thirty two 32-bit general-purpose registers
- User and supervisor modes
- 32-bit combinational multiplier
- Separate data and instruction buses that include 32-bit data bus, 32-bit instruction address bus, and 32-bit instruction bus (fixed instruction length of 32 bits)
- Directly interfaces to memory or to 88200 cache/memory management unit
- 4 gigabytes of physical memory

The 88100 performs register-to-register operation for all data manipulation instructions. Source operands are contained in source registers or are included as an immediate value inherent in the instruction. A separate destination register stores the results of an instruction. This means that source operand registers can be reused in the subsequent instructions. Register contents that can be read from or written to memory only are & (load) and @ (store) instructions. A xmem (memory exchange) instruction is included for semaphore testing and multiprocessor application.

The 88100 contains 51 instructions. All instructions are executed in one cycle. The instructions requiring more than one cycle are executed in effectively one cycle via pipelining. All instructions are decoded by hardware and no microcode is used.

The 88100 includes all data manipulation instructions as register-to-register or register plus immediate value instructions. This eliminates memory access delays in data manipulation. Only 10 memory addressing modes are provided: three modes for data memory, four modes for instruction memory, and three modes for registers.
All 88100 instructions are 32 bits wide. This fixed instruction format minimizes instruction decode time and eliminates the need for alignment. All instructions are fetched in a single memory access. The 88100 implements delayed branching to minimize pipeline delay. For pipelined architecture, branching instructions can slow down execution speed due to the time required to flush and refill the pipeline. The 88100 delayed branching feature allows fetching of the next instruction before the branch instruction is executed.

The 88100 provides two modes: supervisor and user. The supervisor mode is used by the operating system, while the application programs are executed in user mode.

The 88100 includes four execution units which operate independently and concurrently. The 88100 can perform up to five operations in parallel.

Scoreboard bits are associated with each of the general-purpose registers. When an instruction is executed or dispatched, the scoreboard bit of the destination register is set, reserving that register for that instruction. Other instructions are executed or dispatched as long as their source and destination operands have clear scoreboard bits. When an instruction completes execution, the scoreboard bit of the destination is cleared, freeing that register to be used by other instructions.

The 88100 memory devices can interface directly to memory. Most 88100 designs implement at least two 88200 CMRRUs (one for data memory and one for instruction memory). The P-bus provides the interface to the 88200/memory system. The 88200 is an optional external chip that provides page-able memory support and data/instruction cache memory.

Conditional test results are provided to any specified, general-purpose register instead of a dedicated condition code register. Conditions are computed at the explicit request of the programmer using compare instructions. This eliminates contention between concurrent execution units accessing a dedicated condition code register.

8.2.3 88100/88200 INTERFACE

Figure 8.28 shows typical 88100 interfaces to several 88200s. The Pibus (processor bus) contains logical addresses, while MBUS (memory bus) contains all physical addresses. Up to 4 88200s can reside on each Pibus. Note that in the figure, the MC88000 includes the entire RISC microprocessor family, with 88100 being the first microprocessor.

Figure 8.29 shows the 88100/88200 block diagram. Each unit in the 88100 can operate independently and simultaneously. Each unit may be pipelined.

The integer unit performs 32-bit arithmetic, logic, bit field, and address operations. All operations are performed in one clock cycle. The integer unit includes 21 control registers. The floating-point unit supports IEEE 754-1985 floating-point arithmetic, integer multiply, and divide. This unit contains 11 control registers with five-stage add pipeline and six-stage multiply pipeline. Six optional SFUs (special function units) are reserved in the architecture. The SFUs can be added to or removed from a given system with no impact on the architecture.

The data unit performs address calculations and data access and includes a three-stage pipeline. The instruction unit fetches instruction codes and contains a three-stage pipeline.

The register file includes 32 32-bit general-purpose registers. The sequencer uses a scoreboard to control register reads/writes. It dispatches instructions and recognizes exceptions.
program data including instruction operand and results and provide address and bit field information. Internal registers are used by the 88100 to track instruction.

Three types of registers are included:

- 32 general-purpose registers, r0-r31, containing program data (source operand and instruction results). All of these registers except r0 (constant 0) have read/write access.
- Internal registers control instruction execution and data transfer
- Control registers in the various execution units containing status, execution control, and exception processing information

The internal registers cannot be directly accessible in software, while
most control registers can be accessed in supervisor mode. The internal registers can only be modified and used indirectly.

The control registers include shadow registers and exception time
registers, integer-unit control registers, and floating-point unit control
registers.

The shadow registers are associated with several internal registers.
Shadowing is utilized by the 88100 to keep track of the internal pipeline
registers at each stage of the instruction execution. The shadowed registers
include the three instruction pointers, the scoreboard register, the data unit
pipeline registers, and the floating-point unit source operand registers.
Shadowing these registers generates a copy of the instruction pipeline, the
memory access pipeline, and the floating-point source operands at the end
of each cycle.

Exception time registers are updated with copies of internal registers,
but these registers are only written when an exception occurs. The PSR
Corresponds to one of the exception time registers.

There are 21 control registers (cr0 through cr20) in the integer unit.
Fourteen of these registers provide exception information for integer unit
or data unit exceptions. The other seven registers include status information,
the base address of the exception vector table, and general-purpose
storage.

The floating point includes 11 control registers (fcr0-fcr8, fcr62, and
fcr63). fcr0 through fcr8 contain exception information such as the
exception type, source operands and results, and the instruction in progress.
These registers can only be accessed in supervisor mode. Registers fcr62
and fcr63 are not privileged. These two registers can be used to enable
user-supplied exception handler software and to report exception causes in
user mode.

The supervisor programmer's model contains all general-purpose and
control registers. The general-purpose registers provides data and address
information, while the integer unit control registers provide exception
recovery and status information for the integer unit. The integer unit
registers are loaded to and from the general-purpose registers using ldr (load
from control register), stcr (store to control register), and xcr (exchange
control register) instructions. The floating-point unit control registers
provide exception recovery, status, and control information for the
floating-point unit. These registers are loaded to and from the general-purpose
registers using the lldr (load from floating-point control register), stcr
(store to floating-point control register), and xcr (exchange floating-point
control register) instructions.

In user mode, all general-purpose registers can be accessed.
control registers (floating-point control and status) can be accessed in the user mode.

Tables 8.6a through 8.6d summarize the 88100 general-purpose, internal, and control registers.

### TABLE 8.6a
**General-Purpose Register Summary**

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
<th>Convention</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>Constant zero</td>
<td>Hardware</td>
<td>Read only</td>
</tr>
<tr>
<td>r1</td>
<td>Subroutine return pointer</td>
<td>Hardware</td>
<td>Read/write</td>
</tr>
<tr>
<td>r2-r9</td>
<td>Called procedure parameter registers</td>
<td>Hardware</td>
<td>Read/write</td>
</tr>
<tr>
<td>r10-r13</td>
<td>Called procedure temporary registers</td>
<td>Software</td>
<td>Read/write</td>
</tr>
<tr>
<td>r14-r25</td>
<td>Called procedure reserved registers</td>
<td>Software</td>
<td>Read/write</td>
</tr>
<tr>
<td>r26-r29</td>
<td>Linker registers</td>
<td>Software</td>
<td>Read/write</td>
</tr>
<tr>
<td>r30</td>
<td>Frame pointer</td>
<td>Software</td>
<td>Read/write</td>
</tr>
<tr>
<td>r31</td>
<td>Stack pointer</td>
<td>Software</td>
<td>Read/write</td>
</tr>
</tbody>
</table>

### TABLE 8.6b
**Internal Register Summary**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XIP</td>
<td>Pointer to currently executing instruction</td>
</tr>
<tr>
<td>NIP</td>
<td>Pointer to next instruction to execute</td>
</tr>
<tr>
<td>FIP</td>
<td>Pointer to instruction being fetched from memory</td>
</tr>
<tr>
<td>SB</td>
<td>Register file scoreboard</td>
</tr>
</tbody>
</table>

### TABLE 8.6c
**Integer-Unit Control Register Summary**

<table>
<thead>
<tr>
<th>Register</th>
<th>Mnemonic</th>
<th>Function</th>
<th>Shadow/exception time</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>cr0</td>
<td>PID</td>
<td>Processor ID field and PCE bits</td>
<td>—</td>
<td>Read only</td>
</tr>
<tr>
<td>cr1</td>
<td>PSR</td>
<td>Processor status and control bits</td>
<td>—</td>
<td>Read/write</td>
</tr>
</tbody>
</table>

### TABLE 8.6c (continued)
**Integer-Unit Control Register Summary**

<table>
<thead>
<tr>
<th>Register</th>
<th>Mnemonic</th>
<th>Function</th>
<th>Shadow/exception time</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>cr2</td>
<td>TPSR</td>
<td>Trapped processor status registers</td>
<td>Exception</td>
<td>Read/write</td>
</tr>
<tr>
<td>cr3</td>
<td>SSBR</td>
<td>Shadow scoreboard register</td>
<td>Shadow</td>
<td>Read/write</td>
</tr>
<tr>
<td>cr4</td>
<td>SXIP</td>
<td>Shadow execute instruction pointer</td>
<td>Shadow</td>
<td>Read/write</td>
</tr>
<tr>
<td>cr5</td>
<td>SNIP</td>
<td>Shadow next instruction pointer</td>
<td>Shadow</td>
<td>Read/write</td>
</tr>
<tr>
<td>cr6</td>
<td>SFIP</td>
<td>Shadow fetch instruction pointer</td>
<td>Shadow</td>
<td>Read/write</td>
</tr>
<tr>
<td>cr7</td>
<td>VBR</td>
<td>Exception vector table base register</td>
<td>Shadow</td>
<td>Read/write</td>
</tr>
<tr>
<td>cr8</td>
<td>DMT2</td>
<td>Transaction information for faulted data memory transaction</td>
<td>Shadow</td>
<td>Read only</td>
</tr>
<tr>
<td>cr9</td>
<td>DMD2</td>
<td>Data for faulted data memory transaction</td>
<td>Shadow</td>
<td>Read only</td>
</tr>
<tr>
<td>cr10</td>
<td>DMA2</td>
<td>Address for faulted data memory transaction</td>
<td>Shadow</td>
<td>Read only</td>
</tr>
<tr>
<td>cr11</td>
<td>DMT1</td>
<td>Transaction information for active data memory transaction when fault occurred</td>
<td>Shadow</td>
<td>Read only</td>
</tr>
<tr>
<td>cr12</td>
<td>DMD1</td>
<td>Data for active data memory transaction when fault occurred</td>
<td>Shadow</td>
<td>Read only</td>
</tr>
<tr>
<td>cr13</td>
<td>DMA1</td>
<td>Address for active data memory transaction when fault occurred</td>
<td>Shadow</td>
<td>Read only</td>
</tr>
<tr>
<td>cr14</td>
<td>DMT0</td>
<td>Transaction information for data memory transaction being prepared when fault occurred</td>
<td>Shadow</td>
<td>Read only</td>
</tr>
<tr>
<td>cr15</td>
<td>DMD0</td>
<td>Data for data memory transaction being prepared when fault occurred</td>
<td>Shadow</td>
<td>Read only</td>
</tr>
<tr>
<td>cr16</td>
<td>DMA0</td>
<td>Address for data memory transaction being prepared when fault occurred</td>
<td>Shadow</td>
<td>Read only</td>
</tr>
</tbody>
</table>
TABLE 8.6d
Floating-Point Control Register Summary

<table>
<thead>
<tr>
<th>Register</th>
<th>Mnemonic</th>
<th>Function</th>
<th>Shadow/exception time</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcr0</td>
<td>FPICR</td>
<td>Exception cause indicator</td>
<td>—</td>
<td>Read only</td>
</tr>
<tr>
<td>fcr1</td>
<td>FPICR</td>
<td>Upper bits of floating-point source 1 operand when exception occurred</td>
<td>Shadow</td>
<td>Read only</td>
</tr>
<tr>
<td>fcr2</td>
<td>FPICR</td>
<td>Lower bits of floating-point source 1 operand or integer source 1 operand when exception occurred</td>
<td>Shadow</td>
<td>Read only</td>
</tr>
<tr>
<td>fcr3</td>
<td>FPICR</td>
<td>Upper bits of floating-point source 2 operand when exception occurred</td>
<td>Shadow</td>
<td>Read only</td>
</tr>
<tr>
<td>fcr4</td>
<td>FPICR</td>
<td>Lower bits of floating-point source 2 operand or integer source 1 operand when exception occurred</td>
<td>Shadow</td>
<td>Read only</td>
</tr>
<tr>
<td>fcr5</td>
<td>FPICR</td>
<td>Information about instruction that caused a precise exception</td>
<td>Exception time</td>
<td>Read only</td>
</tr>
<tr>
<td>fcr6</td>
<td>FPICR</td>
<td>Status information and upper 21 bits of partial floating-point result when imprecise exception occurred</td>
<td>Exception time</td>
<td>Read only</td>
</tr>
<tr>
<td>fcr7</td>
<td>FPICR</td>
<td>Low-order bits of partial floating-point result when imprecise exception occurred</td>
<td>Exception time</td>
<td>Read only</td>
</tr>
<tr>
<td>fcr8</td>
<td>FPICR</td>
<td>Information about instruction that caused an imprecise exception</td>
<td>Exception time</td>
<td>Read only</td>
</tr>
<tr>
<td>fcr9</td>
<td>FPICR</td>
<td>Flags that indicate which floating-point exception occurred</td>
<td>—</td>
<td>Read only</td>
</tr>
<tr>
<td>fcr10</td>
<td>FPICR</td>
<td>Flags that specify rounding mode and that enable user-supplied exception handlers</td>
<td>—</td>
<td>Read/Write</td>
</tr>
</tbody>
</table>

As the 32 general-purpose registers, r0 always contains the value 0, r1 is loaded with the subroutine return address, and r2 through r31 are general-purpose.

Figure 8.32 shows the 88000 register data formats. The 88100 supports two types of data formats, namely, integer (signed or unsigned) and floating-point real numbers. The integers can be byte, half-word (16-bit), and word (32-bit). All operations affect all 32 bits of a general-purpose register. The half-word or byte pads the sign bit; a signed byte writes the sign bit. That is, a signed byte writes the sign bit from bit 8 through bit 31. An unsigned byte, on the other hand, writes 0 from bit 8 through bit 31.

The floating-point data can be single precision and double precision. Figure 8.33 shows formats for Fcr2 and Fcr3. The reserved bits in Fcr2 and Fcr3 are always read as zero. The FPCR defines the desired rounding mode and which exceptions are handled by user exception handlers. The FPSR indicates which floating-point exceptions have occurred but were not processed by a user exception handler.

The 88100 general-purpose register convention is shown in Figure 8.34. r31 addresses the top of the stack. r30 contains the address of the current data frame in the stack.
Figure 8.35 shows the 88100 stack operation. The SP must always be 32-bit aligned. The stack grows from high memory to low memory addresses.

Next, consider the supervisor programmer model:

- The VBR contains the address of the exception vector table.
- The 88100 does not automatically use SRO-SR3. They are reserved for operating system use.

Figure 8.36 shows the 88100 processor status register format. In Figure 8.36, Big Endian means the most significant byte at the highest byte address. Serial instruction is to complete before the next one begins. Note that not all adds/subtracts affect C. The SFDI bit enables or disables the floating-point unit. When SFDI = 1, attempted execution of any floating-point or integer multiply/divide instructions cause floating-point precise exceptions.
8.2.5 88100 DATA TYPES, ADDRESSING MODES, AND INSTRUCTIONS

Tables 8.7a and 8.7b list the data types and addressing modes supported by the 88100. Table 8.8 summarizes the 88100 instructions.

The 51 instructions listed in Table 8.8 of the MC88100 can be divided into 6 classes: integer arithmetic, floating-point arithmetic, logical, bit field, load/store/exchange, and flow control.

These simple instructions must be used to obtain complex operations.

TABLE 8.7a

<table>
<thead>
<tr>
<th>Data type</th>
<th>Represented as</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit fields</td>
<td>Signed and unsigned bit fields from 1 to 32 bits</td>
</tr>
<tr>
<td>Integer</td>
<td>Signed and unsigned byte (8 bits)</td>
</tr>
<tr>
<td>Floating point</td>
<td>Signed and unsigned half-word (16 bits)</td>
</tr>
<tr>
<td></td>
<td>Signed and unsigned word (32 bits)</td>
</tr>
<tr>
<td></td>
<td>IEEE P754 single precision (32 bits)</td>
</tr>
<tr>
<td></td>
<td>IEEE P754 double precision (64 bits)</td>
</tr>
</tbody>
</table>

TABLE 8.7b

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register indirect with unsigned immediate</td>
<td>rD,rS1,imm16</td>
</tr>
<tr>
<td>Register indirect with index</td>
<td>rD,rS1,i52</td>
</tr>
<tr>
<td>Register indirect with scaled index</td>
<td>rD,rS1,i52</td>
</tr>
</tbody>
</table>

TABLE 8.8

<table>
<thead>
<tr>
<th>Instruction Set Summary</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Arithmetic</td>
<td>add</td>
<td>Add</td>
</tr>
<tr>
<td></td>
<td>addu</td>
<td>Add unsigned</td>
</tr>
<tr>
<td></td>
<td>cmp</td>
<td>Compare</td>
</tr>
<tr>
<td></td>
<td>div</td>
<td>Divide</td>
</tr>
</tbody>
</table>
### TABLE 8.8 (continued)
#### Instruction Set Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Integer Arithmetic Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>divu</td>
<td>Divide unsigned</td>
</tr>
<tr>
<td>mul</td>
<td>Multiply</td>
</tr>
<tr>
<td>sub</td>
<td>Subtract</td>
</tr>
<tr>
<td>subu</td>
<td>Subtract unsigned</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Floating-Point Arithmetic Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>fadd</td>
<td>Floating-point add</td>
</tr>
<tr>
<td>fcmp</td>
<td>Floating-point compare</td>
</tr>
<tr>
<td>fdiv</td>
<td>Floating-point divide</td>
</tr>
<tr>
<td>flcr</td>
<td>Load from floating-point control register</td>
</tr>
<tr>
<td>fli</td>
<td>Convert integer to floating point</td>
</tr>
<tr>
<td>fmul</td>
<td>Floating-point multiply</td>
</tr>
<tr>
<td>fstore</td>
<td>Store to floating-point control register</td>
</tr>
<tr>
<td>fsub</td>
<td>Floating-point subtract</td>
</tr>
<tr>
<td>fsr</td>
<td>Exchange floating-point control register</td>
</tr>
<tr>
<td>frt</td>
<td>Round floating point to integer</td>
</tr>
<tr>
<td>fntr</td>
<td>Floating-point round to nearest integer</td>
</tr>
<tr>
<td>fnccn</td>
<td>Truncate floating point to integer</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Logical Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>AND</td>
</tr>
<tr>
<td>mask</td>
<td>Logical mask immediate</td>
</tr>
<tr>
<td>or</td>
<td>OR</td>
</tr>
<tr>
<td>xor</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bit-Field Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>cir</td>
<td>Clear bit field</td>
</tr>
<tr>
<td>ext</td>
<td>Extract signed bit field</td>
</tr>
<tr>
<td>estu</td>
<td>Extract unsigned bit field</td>
</tr>
<tr>
<td>fdo</td>
<td>Find first bit clear</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bit-Field Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>cir</td>
<td>Clear bit field</td>
</tr>
<tr>
<td>ext</td>
<td>Extract signed bit field</td>
</tr>
<tr>
<td>estu</td>
<td>Extract unsigned bit field</td>
</tr>
<tr>
<td>fdo</td>
<td>Find first bit clear</td>
</tr>
</tbody>
</table>

**Bit-Field Instructions**

- **Find first bit set**: Find first bit set
- **Make bit field**: Make bit field
- **Rotate register**: Rotate register
- **Set bit field**: Set bit field

**Load/Store/Exchange Instructions**

- **ld**: Load register from memory
- **lda**: Load address
- **ldcr**: Load from control register
- **st**: Store register to memory
- **stcr**: Store to control register
- **xcr**: Exchange control register

**Flow Control Instructions**

- **bb0**: Branch on bit clear
- **bb1**: Branch on bit set
- **bcnd**: Conditional branch
- **br**: Branch to subroutine
- **jcr**: Unconditional jump
- **jst**: Jump to subroutine
- **rne**: Return from exception
- **tb0**: Trap on bit clear
- **tb1**: Trap on bit set
- **tbnd**: Trap on bounds check
- **tcnd**: Conditional trap

Shift and rotate operations are special cases of bit field instructions. Only `ld`, `st`, and `xmem` can access memory. Also, only compare instructions affect condition codes. Most MC88100 instructions can have one of the three formats.
1. Triadic register instructions (three operands). The general format is

```
    mnemonic   <opt> r0, r1, r2
    option     destination register
    destination register
    source 1 register
```

An example is `add.c, r2, r7, r4`. Note that Motorola's assembler expects the 88100 instructions in lowercase. In the example, the mnemonic is `add` and the option is `c`, meaning use 'carry in' in the operation. The source registers r7 and r4 remain unchanged unless one of them is used as destination. This add instruction adds [r7] and [r4] with carry and stores result in r2.

2. Triadic register instructions with 16-bit field instruction. The general format is mnemonic: <opt> D, r1, Imm16. Consider

```
    add.C0 r2, r4, 0x123
```

Immediate data

0X before data A125 means that A125 is in hex. This notation is used by the Motorola assembler.

3. Dyadic register instructions (two operands). The general format is mnemonic: <opt> r0, r2. An example is `fdiv r5, r0` — which converts the integer source word in r0 into floating-point in r5.

Table 8.9 lists the 88100 load, store, and exchange instructions.

- `ld` loads a general-purpose register from data memory. There are three registers with this instruction. Two-source operands are used to calculate the address. Three forms of `ld` use the three addressing modes available. The (<opt>) for `ld` specifies the size of data read from memory.
- The st instructions are similar to ld instructions, except they are used to store source data.
- The exchange instructions (`xmem`, `xcr`, `fxcr`) swap the content of a general-purpose register with data memory or with a control register. 
- Consider `ld r7, r31, 0x4`. The mode used here is register indirect with unsigned immediate. If [r31] = 00005000, the effective address is 00005000.

Table 8.9: MC88100 Load, Store, Exchange Instructions

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ld</code> (&lt;opt&gt;)</td>
<td>Data access</td>
</tr>
<tr>
<td><code>ld</code> (&lt;opt&gt;)</td>
<td>Misaligned access</td>
</tr>
<tr>
<td><code>ld</code> (&lt;opt&gt;)</td>
<td>Privilege violation</td>
</tr>
<tr>
<td><code>st</code> (&lt;size&gt;)</td>
<td>Data access</td>
</tr>
<tr>
<td><code>st</code> (&lt;size&gt;)</td>
<td>Misaligned access</td>
</tr>
<tr>
<td><code>st</code> (&lt;size&gt;)</td>
<td>Privilege violation</td>
</tr>
<tr>
<td><code>xmem</code> (&lt;bu&gt;)</td>
<td>Data access</td>
</tr>
<tr>
<td><code>xmem</code> (&lt;bu&gt;)</td>
<td>Misaligned access</td>
</tr>
<tr>
<td><code>xmem</code> (&lt;bu&gt;)</td>
<td>Privilege violation</td>
</tr>
<tr>
<td><code>fdiv</code></td>
<td>Privilege violation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opns</th>
<th>For</th>
<th>Id</th>
<th>&lt;size&gt;</th>
<th>For</th>
<th>st</th>
</tr>
</thead>
<tbody>
<tr>
<td>.b</td>
<td>Signed byte</td>
<td>b</td>
<td>Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.bu</td>
<td>Unsigned byte</td>
<td>h</td>
<td>Halfword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.h</td>
<td>Signed halfword</td>
<td>none</td>
<td>Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.hu</td>
<td>Unsigned halfword</td>
<td>d</td>
<td>Doubleword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.d</td>
<td>Word</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.dx</td>
<td>Doubleword</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.usr</td>
<td>Access user space regardless of mode bit in PSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.none</td>
<td>Access space indicated by PSR MODE bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* rs = Source register.
* crCRS = Source control register.
TABLE 8.9 (continued)  
MC68100 LOAD, STORE, Exchange Instructions

- crCRD = Destination control register.  
- crCRS/D = Source/destination control register.  
- crCFRS = Source floating-point control register.  
- crFCRD = Destination floating-point control register.  
- crFCRS/D = Source/destination floating-point control register.  
- Memory accesses for xmem are indivisible.

Compared by adding rs (r31) with unsigned 16-bit immediate data. Therefore, the 88100 loads the register r7 with 32-bit data from a memory location addressed by 00005004, r31 is the SP. Therefore, the access occurs within the stack. Since the immediate data are unsigned, the accessed address cannot be less than r31. This means that the stack grows toward the lower address.

- Consider st.b rs, r1, r2. This instruction has register indirect with index mode.

The access address is r31 + rs where r31 is the base register and rs is the index register. For example, consider st.b r1, r0, r5. If [r5] = 00010200, then since r0 is always 0, the low 8-bit content of r1 is stored at address 00001200. The 88100 ignores any carry generated during address calculation. Note that in the above r0 is the base address and r5 is the index register.

Finally, consider st.hl rs, r1 [rs2]. The mode is register indirect with index. The access address is r31 + rs2 (operand size). The scaling is specified by surrounding the index register rs2 by square brackets. Operand size is 1 for byte, 2 for halfword, 4 for word, and 8 for double word.

As an example, consider st. r5, r31 [r1]. If [r1] = 00000004, r31 = 00005000, then the effective address is 00005000 + 4 * 00000004 = 0000500C, scaled by 4 for word since the instruction without any option specified means 32-bit word.

Therefore, the above store instruction stores the 32-bit contents of r5 into a memory location addressed by 0000500C.

Table 8.10 shows the integer arithmetic instructions.

- Consider add [<opt>] r0, r1, r2. Three options can be used with this instruction as follows:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>rD.rS1.rS2</td>
</tr>
<tr>
<td>addu[&lt;opt&gt;]</td>
<td>rD.rS1.&lt;imm16&gt;</td>
</tr>
<tr>
<td>sub[&lt;opt&gt;]</td>
<td>rD.rS1.rS2</td>
</tr>
<tr>
<td>sub</td>
<td>rD.rS1.&lt;imm16&gt;</td>
</tr>
<tr>
<td>mul</td>
<td>rD.rS1.rS2</td>
</tr>
<tr>
<td>div</td>
<td>rD.rS1.rS2</td>
</tr>
<tr>
<td>divu</td>
<td>rD.rS1.&lt;imm16&gt;</td>
</tr>
<tr>
<td>divu</td>
<td>rD.rS1.rS2</td>
</tr>
<tr>
<td>lda[&lt;size&gt;</td>
<td>rD.rS1.rS2</td>
</tr>
<tr>
<td>lda[&lt;size&gt;</td>
<td>rD.rS1.&lt;imm16&gt;</td>
</tr>
<tr>
<td>lda[&lt;size&gt;</td>
<td>rD.rS1.rS2</td>
</tr>
<tr>
<td>lda[&lt;size&gt;</td>
<td>rD.rS1.&lt;imm16&gt;</td>
</tr>
<tr>
<td>lda[&lt;size&gt;</td>
<td>rD.rS1.rS2</td>
</tr>
</tbody>
</table>

<opt> FOR add/addu/sub/subu

- none: No carry
- .ci: Use carry in
- .co: Propagate carry out
- .clo: Use carry in and propagate carry out

<size> FOR lda

- h: Scale rS2 by 1
- .h: Scale rS2 by 2
- none: Scale rS2 by 4
- .d: Scale rS2 by 8

* mul yields correct signed and unsigned results.  
* Division by zero signals the integer divide exception.  
* An integer divide exception occurs when either source operand is negative for div.  
* Unscaled lda is functionally equivalent to addu.
363.jpg
The 88100 allows single and double precision operands. Sizes specify the operand sizes of rD, rS1, and rS2 as single or double precision. For example, word means that rD and rS1 are single precision, while rS2 is a double precision.

The operations trnc, nint, int, and fit provide conversions between integer and floating-point values. These instructions have two operands with one operand having floating-point value and the other having integer value. trnc, nint, and int convert a floating-point format to an equivalent format. The difference between them is the type of rounding performed. trnc rounds toward zero, and nint rounds to the nearest method int rounds specified by the RM field in FPCR.

Two exceptions are provided for floating-point instructions. Integer conversion overflow exception occurs when the operand value cannot be expressed as a high word. Reserved operand exception occurs by certain floating-point values.

fit converts a signed 32-bit number into a floating-point format. The integer operand size is always specified by “sizes” indicating signed word size. A “d” for double or “s” for single defines the floating-point operand’s precision.

Table 8.12 lists the MC88100 logical instructions.

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>and [c]</td>
<td>rD,rS1,rS2</td>
</tr>
<tr>
<td>and [u]</td>
<td>rD,rS1,&lt;imm16&gt;</td>
</tr>
<tr>
<td>mask [u]</td>
<td>rD,rS1,&lt;imm16&gt;</td>
</tr>
<tr>
<td>or [c]</td>
<td>rD,rS1,rS2</td>
</tr>
<tr>
<td>or [u]</td>
<td>rD,rS1,&lt;imm16&gt;</td>
</tr>
<tr>
<td>xor [c]</td>
<td>rD,rS1,rS2</td>
</tr>
<tr>
<td>xor [u]</td>
<td>rD,rS1,&lt;imm16&gt;</td>
</tr>
</tbody>
</table>

- Option c ones complement the contents of rS2 before performing the operation.
- Option u performs the specified logical operation between 16-bit immediate data <imm16> and high 16 bits of rS1.

Table 8.13 lists the 88100 bit field instructions.

- WS is five bit width and <05> is five bit offset.
- The number of bits in a bit field is called width. The size of width b can be from 1 to 32. The least significant bit in a bit field is called the offset. When the sum of width and offset is greater than 32, the bit field may be imagined to extend beyond the most significant bit of the register.

Isolated bit field ends in the least significant bit of a register with an implicit offset of zero. Bit field may contain signed and unsigned values. For unsigned isolated bit field, the high-order bits in a register are all zero and for signed isolated bit field, they are 2's complement sign bit. In either case, the entire register contains the word value equivalent to the isolated bit field value. The 88100 includes instructions to isolate embedded bit field for arithmetic manipulation. The 88100 instruction moves isolated bit field back into embedded bit field. This is illustrated in the following:
Two bit field formats are used. These are literal width with offset and register width with offset. The literal width with offset uses immediate width and offset values. An example is: $r3, r1, 3 <7>$. The destination register is $r5$; the source bit field is 3 bits wide with an offset of 7 in $r3$. With Motorola assembler, the offset must be included in angle brackets $< >$.

The register width with offset uses three operands. An example is: $dr r1, r3, r6$. The source bit field is in $r3$ with offset and width determined from $r6$. The MC88100 obtains the offset from bits 0-4 of $r4$ and the width from bits 5-9 of $r4$. The upper 22 bits of $r4$ are don't cares. Both formats use an offset of 0 to 31 and width of 1 to 32 with 32 encoded as 0. The destination register ($r1$ in this case) stores the final result. The content of the source register ($r3$) does not change after the operation.

- ext (signed) and extu (unsigned) instructions extract the register value from $r1$ and convert to an isolated bit field in $rD$. For a bit field width of 32 (encoded as 0), ext and extu perform shift right operation.
- mak creates an imbedded bit field in $rD$ with an offset specified by an immediate value or by the content of $r2$. The MC88100 stores the least significant bits of $r1$ in the imbedded bit field. The bits outside the imbedded bit field in $rD$ are cleared to zero. The mak is the inverse operation of ext and extu.
- The shift left operation may be performed by a bit field width of 32. The offset specifies the number of positions to be shifted.
- rot reads $r1$ and rotates it to the right by the number of bits specified in $<0>$. If in bits 0-4 of $r2$. The result is stored in $rD$.
- ffl finds the most significant set bit in $r2$ and stores the bit number in $rD$.
- If all bits are cleared, the 88100 loads 32 into $rD$. ffl0 operates similarly but finds the most significant clear bit.

Table 8.14 summarizes the 88100 Integer Compare instructions.
TABLE 8.14
MC88100 Integer Compare

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp rD,rS1,imm16</td>
<td>None</td>
</tr>
<tr>
<td>cmp rD,rS1,rS2</td>
<td></td>
</tr>
</tbody>
</table>

Predicate Bit String:

- cmp tests the content of rS1 for = 0, ≠ 0, <0, >0, and ≤0 and branches with 16-bit signed displacement if the condition is true. The 16-bit signed displacement is sign-extended to 32 bits, shifted twice to the left, and adds to address of branch to branch with a displacement of (2^16 - 4) bytes. i indicates 'execute next'. If i is present and the condition is true, the branch executes the next instruction before taking the branch. The 'execute next' allows the 88100 to branch without flushing the execution pipeline and thus provides faster execution.

- tcomb also tests the content of rS1 for the condition but traps if the condition is true. The 88100 includes 512 vectors in the vector table. <veco> specifies a 9-bit vector number from 0 to 511. bb1 (branch on bit set) tests the content of rS1 for a set bit. If it is set, the 88100 takes a branch. <bb5> specifies a bit number from 0 to 31. bb1 usually follows cmp and fcmp instructions. tbb1 is similar to bb1 except a trap is taken if the bit is set. bb0 (branch on bit clear) is similar to bb1 except a branch is taken if the specified bit is 0. tbb0 is similar to tbb1 except a trap is taken if the specified bit is 0. tbb0 is similar to tbb1 except a trap is taken if the specified bit is 0. The bound (trap on bound check) generates bound check violation. rS1 contains out of bounds; 0 is the implicit lower bound. The upper bound is either an assigned 16 bits or contained in rS2. The value of the upper bound is treated as an assigned number. If the array limit is less than 0 or greater than the upper value, an exception is taken.
### Table 8.16
MC88100 Conditional Flow Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Conditions</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bcond [.n]</code></td>
<td><code>&lt;cond&gt;, r51, &lt;d16&gt;</code></td>
<td>None</td>
</tr>
<tr>
<td><code>tcnd</code></td>
<td><code>&lt;cond&gt;, r51, &lt;vec9&gt;</code></td>
<td>Trap vector 9, privilege violation</td>
</tr>
<tr>
<td><code>bb1 [.n]</code></td>
<td><code>&lt;b5&gt;, r51, &lt;d16&gt;</code></td>
<td>None</td>
</tr>
<tr>
<td><code>bb0 [.n]</code></td>
<td><code>&lt;b5&gt;, r51, &lt;d16&gt;</code></td>
<td>Trap vector 9, privilege violation</td>
</tr>
<tr>
<td><code>tbi</code></td>
<td><code>&lt;b5&gt;, r51, &lt;vec9&gt;</code></td>
<td>Privilege violation</td>
</tr>
<tr>
<td><code>tb0</code></td>
<td><code>&lt;b5&gt;, r51, &lt;vec9&gt;</code></td>
<td>Privilege violation</td>
</tr>
<tr>
<td><code>tbnd</code></td>
<td><code>r51, r52</code></td>
<td>Simulated check violation</td>
</tr>
<tr>
<td><code>tbnd</code></td>
<td><code>r51, cimm16</code></td>
<td>Sounds check violation</td>
</tr>
</tbody>
</table>

An instruction using the `.n` option must not be followed by another flow control instruction. (Error undetected by the MC88100.)

### Table 8.17
MC88100 Unconditional Flow Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Conditions</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>br [.n]</code></td>
<td><code>&lt;d26&gt;</code></td>
<td>None</td>
</tr>
<tr>
<td><code>bsr [.n]</code></td>
<td><code>&lt;d26&gt;</code></td>
<td>Privilege violation</td>
</tr>
<tr>
<td><code>jrp [.n]</code></td>
<td><code>r52</code></td>
<td>None</td>
</tr>
<tr>
<td><code>jsr [.n]</code></td>
<td><code>r52</code></td>
<td>Privilege violation</td>
</tr>
<tr>
<td><code>ite</code></td>
<td><code>&lt;d26&gt;</code></td>
<td>Signed 26-bit displacement</td>
</tr>
<tr>
<td><code>.n</code></td>
<td><code>execute next</code></td>
<td></td>
</tr>
<tr>
<td><code>bsr</code> and <code>jsr</code></td>
<td>Save the return address in r1</td>
<td></td>
</tr>
<tr>
<td><code>jrp</code></td>
<td>Performs return from subroutine</td>
<td></td>
</tr>
<tr>
<td>The last instruction of typical exception handlers is <code>ite</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `br` always unconditionally branches with signed 26-bit displacement with a range of $2^{-30}$ to $2^{-28} - 4$ bytes.
- `bsr` is an unconditional subroutine call and saves the return address in r1. When `.n` is specified, the return address is the address of `bsr` plus 8.
- `jrp` branches to the address specified by `r52` content. The 88100 rounds the least 2 bits of `r52` to 0 before branching for alignment. However, the contents of `r52` are unchanged by the instruction. `jrp` is similar to `jrp` except that it is a subroutine jump to an address specified by `r52` content and also saves the return address in r1.
- The 88100 does not provide any return from subroutine instruction. Jump r1 fetches the next instruction from the return address saved by `bsr` or `jsr`.
- `ite` provides an orderly termination of an exception handler. It uses the shadow registers to restore the state that existed before the prior exception. `ite` can only clear the mode bit in PSR and ensures that the instruction is executed in user mode.

**Example 8.6**

Show the contents of registers and memory after the 88100 executes the following instructions:

1. `sh r2, r3, r5`
2. `xmem r2, [r3]`
Chapter 8

Motorola MC68030 and MC68100

Assume the following data:

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000 FF 25 71 03</td>
</tr>
<tr>
<td>2004 04 81 03 03</td>
</tr>
<tr>
<td>2008 A2 71 36 25</td>
</tr>
<tr>
<td>8000 01 85 A2 71</td>
</tr>
<tr>
<td>8004 81 11 26 05</td>
</tr>
</tbody>
</table>

**Solution**

i) \([r3] + [r5] = 2002_{16}\). Consider sth \(r2, r3, r5\) where \(h)\) stands for half-word (16 bits). The low 16 bits of \(r2\) is stored in \(2002_{16}\) and \(2003_{16}\). Therefore

\[
\begin{array}{c}
2002 \\
2003
\end{array}
\]

ii) \(\text{movem } r2, r2[r3]\). This is a 32-bit word operation. Hence, the scale factor is 4. The effective address

\[
= [r2] + 4 \times [r3] \\
= 0000 0004_{16} + 4 \times 00002000_{16} \\
= 0000 0004_{16}
\]

Therefore, after the movem, \([r2] = B111 2605_{16}\) and \([00008004] = 0000 0004_{16}\).

Example 8.7

Write an instruction to logically shift right by 3 bits the value of \(r2\) into \(r5\).

**Solution**

Since \(r2\) is 32 bits wide, extru performs logical right shift and the width 32 is encoded as 0. extru \(r6, r2, 0\times3\).

Example 8.8

Write an MC68100 instruction sequence to logically AND the 32-bit content of \(r7\) with \(F2710562_{16}\) and store the result in \(r5\).

**Solution**

and \(r5, r7, 0x0562\); and logically ANDs \(0562_{16}\); with low 16 bits of \(r7\) and stores result in low 16 bits of \(r5\).

and.u \(r5, r7, \text{ Ox271}\); and.u logically ANDs \(271_{16}\) with high 16 bits of \(r7\) and stores result in high 16 bits of \(r5\).

Example 8.9

Find an MC68100 instruction to load register \(r5\) with the constant value 9.

**Solution**

add \(r5, r0, 9\).

Example 8.10

Find an MC68100 instruction to branch to the instruction with label \(\text{START}\) if the value in \(r5\) is equal to zero.

**Solution**

bcd eq0, r5, START.

8.2.6 88100 PINS AND SIGNALS

Figure 8.37 shows the 88100 pin diagram. Figure 8.38 shows these pins by functional group. Table 8.18 provides a brief description of the functions of these pins.

8.2.7 88100 EXCEPTION PROCESSING

The 88100 includes the following exceptions:

- Reset the hardware interrupts which are activated externally via the respective input pins
- Externally activated errors such as a memory access fault
- Internally generated errors such as divide by zero
- Trap instructions

Exceptions are processed by the 88100 after completion of the current instructions. When an exception is acknowledged, the 88100 freezes the
Chapter 8

Motorola MC68030 and MC68100

8.2 Motorola MC88100 32-Bit Microprocessor

FIGURE 8.36 MC88100 signal functional diagram.

TABLE 8.18 Signal Index

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Mnemonic</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data address bus</td>
<td>DA2-DA31</td>
<td>Provides the 36-bit word address to the data memory space; an entire data word (32 bits) is always addressed: individual bytes or half words are selected using the data byte strobe signals</td>
</tr>
<tr>
<td>Data bus</td>
<td>DX-D31</td>
<td>32-bit bidirectional data bus interfacing the MC88100 to the data memory space</td>
</tr>
<tr>
<td>Data supervisor/user</td>
<td>DS/U</td>
<td>This signal selects between the supervisor select data address space and the user data address space: DS/U is determined by the value of the MODE bit in the processor status register, or by the user option of the ID and st instructions</td>
</tr>
<tr>
<td>Data read/write</td>
<td>DR/W</td>
<td>Indicates whether the memory transaction is a read (DR/W = 1) or a write (DR/W = 0)</td>
</tr>
</tbody>
</table>
### TABLE 8.18 (continued) Signal Index

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Mnemonic</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data bus lock</td>
<td>DLOCK</td>
<td>The memory lock pin is used by the xmem instruction in conjunction with the CMMU; (x\text{mem} ) instruction allows control of the memory bus during the two xmem accesses; data are guaranteed to be unaccessed between the read and write accesses of the xmem instruction.</td>
</tr>
<tr>
<td>Data byte enable</td>
<td>DBE0-DBE3</td>
<td>Used during memory access; these signals indicate which bytes are accessed at the addressed location; DBE0-DBE3 are always valid during memory write cycles; a memory read is always 4 bytes wide, and the processor uses the enables to extend the valid data; if during an 16-bit instruction, the memory system should drive all 32 data signals, regardless of whether 1, 2, or 4 bytes enable are asserted; when DBE0-DBE3 are negated, the transaction is a null; otherwise, the transaction is a valid load or store operation.</td>
</tr>
<tr>
<td>Data reply</td>
<td>DRO-OR1</td>
<td>Indicates the status of the data memory transaction.</td>
</tr>
<tr>
<td>Code address bus</td>
<td>CA2-CA31</td>
<td>Provides the 30-bit word address to the instruction memory space; all instructions are 32 bits wide and are aligned on 4-byte boundaries; therefore, the lower two bits of the address space are not required and are implied to be zero.</td>
</tr>
<tr>
<td>Code bus</td>
<td>C0-C11</td>
<td>This read-only, 32-bit data bus interfaces the MC68030 to the instruction memory space; instructions are always 32 bits wide.</td>
</tr>
<tr>
<td>Code supervisor/user select</td>
<td>CS/UT</td>
<td>Selects between the user and supervisor instruction memory spaces; when asserted, selects supervisor memory and when negated, user memory; this signal is determined by the value of the MODE bit in the processor status register.</td>
</tr>
</tbody>
</table>

### TABLE 8.18 (continued) Signal Index

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Mnemonic</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code fetch</td>
<td>CFETCH</td>
<td>When asserted, signals that an instruction fetch is in progress; when negated, the instruction is a null transaction (code: P bus idle).</td>
</tr>
<tr>
<td>Code reply</td>
<td>CRO-CRI1</td>
<td>Signals the status of the instruction memory transaction.</td>
</tr>
<tr>
<td>Error</td>
<td>ERR</td>
<td>Asserted when a bus comparator error occurs; ERR indicates that the desired signal level was not driven on the output pin; ERR is used in systems implementing a master/slave configuration of MC88100.</td>
</tr>
<tr>
<td>Clock</td>
<td>CLK</td>
<td>Internal clock normally phase locked to minimize skew between the external and internal signals; since CLK is applied to peripherals such as CAMU devices, exact timing of internal signals is required to properly synchronize the device to the P bus.</td>
</tr>
<tr>
<td>Phase lock enable</td>
<td>PLEN</td>
<td>Asserted during reset to select phase lock; PLEN controls the internal phase lock circuit that synchronizes the internal clocks to CLX.</td>
</tr>
<tr>
<td>Reset</td>
<td>RST</td>
<td>Used to perform an orderly reset of the processor; when asserted, the instructions pipeline is cleared and certain internal registers are cleared or initialized; when negated, the reset vector is fetched from memory, with execution beginning in supervisor mode.</td>
</tr>
<tr>
<td>Interrupt</td>
<td>INT</td>
<td>Indicates that an interrupt request is in progress; when asserted, the processor saves the execution context and begins execution at the interrupt exception vector; software is responsible for handling all recognized interrupts those between instructions when no higher priority exception occurs.</td>
</tr>
</tbody>
</table>
### TABLE 8.18 (continued)
**Signal Index**

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Mnemonic</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>P bus checker enable</td>
<td>PCE</td>
<td>Used in systems incorporating two or more MC68010s redundantly; when enabled, the processor operates normally and when asserted, the processor monitors (but does not drive) all of its outputs except ERR as inputs. +5 volt power supply.</td>
</tr>
<tr>
<td>Power supply</td>
<td>Vcc</td>
<td>Ground connections</td>
</tr>
<tr>
<td>Ground</td>
<td>GND</td>
<td></td>
</tr>
</tbody>
</table>

Contents of shadow and exception time control registers, disables interrupts and all SFUs, and enters supervisor mode.

Figure 8.39 shows the 88100 general exception (other than reset) flowchart. Note the following in the flowchart:

- The instruction pipeline shadow registers and register scoreboard are frozen by setting the SFZ bit in PSR to one.
- The floating-point unit is disabled by setting the SFZ bit in PSR to one.
- Interrupts are disabled by setting the IND bit to PSR to one.
- The MODE bit in PSR is set to one by the 88100 to operate in the supervisor mode.

For another exception, state information cannot be saved if shadow registers are frozen. Hence, for nested exceptions, an exception handler must be able to save shadow registers to memory and clear SFZ in PSR to zero.

Figures 8.40 and 8.41, respectively, show the 88100 reset flowchart and timing diagrams. When the 88100 RST pin is asserted, all outputs go into high impedance state except ERR, which indicates no error. Upon hardware reset, the 88100 initializes PSR with appropriate data (SFZ, SFZ, SFZ, SFZ, SFZ, SFZ, SFZ, SFZ, SFZ, SFZ), VBR with zero value, and fetches two 32-bit instructions from the reset vector 0. When reset, in timing diagrams indicates push button reset.

PLLEN is the phase lock enable which controls phase locking to the internal clock of the 88100 to the external clock.

When SFZ = 1, preemption context and data such as instruction pointer, register scoreboard, data and pipeline registers and floating point source operand registers are associated with the registers (shadowed). When SFZ = 0, no context state is saved.
Reset, the assertion of phase lock enable (PLLEN) signal determines the type of phase locking performed. Figure 8.41a describes power-up reset with phase locking enabled. After power stabilizes, RST asserts and PLLEN negates together for more than 8 clock cycles. PLLEN asserts while RST remains asserted for at least 64 clock cycles. During this time, the 88100 internal clock synchronizes to the external clock input using the phase locking technique. After RST negates, the 88100 enters the exception state.

Figure 8.41b shows power-up reset with phase locking disabled. Operating the 88100 without phase locked clock is useful for debugging since this allows the 88100 to operate at clock frequencies lower than possible with phase locking enabled. After power stabilizes, RST asserts and PLLEN negates for more than 8 clock cycles. PLLEN remains negated after RST negates. The 88100 then enters the exception state.

Figure 8.41c shows a timing diagram for a warm reset after power-up reset without any change in phase locking. PLLEN remains either asserted or negated throughout the duration of the reset. RST asserts for more than 8 clock cycles. After RST negates, the 88100 enters exception.

**Figure 8.41 88100 reset exception timing diagrams.**

Figure 8.41d shows the timing diagram for a warm reset with the clock relocked. RST asserts and PLLEN negates for more than 8 clock cycles. PLLEN asserts while RST remains asserted for at least 64 clock cycles. During this time, the 88100 internal clock re synchronizes to the external clock input. After RST negates, the 88100 goes into exception.

The 88100 can be interrupted when the INT pin is asserted HIGH. The 88100 completes the current instruction, then performs the following before branching to the service routine:

- Internally synchronizes INT for one clock cycle
- Fetches instruction of vector
- Propagates instruction through code pipeline

The 88100 code access faults are handled by asserting CR0 and CR1 input pins HIGH. A high on CR0 and CR1 pins indicates a fault. Table 8.19 lists the 88100 program error exceptions.
TABLE 8.19
88100 Program Error Exceptions

Mislaid access faults
Data memory addresses misaligned for 16, 32, or 64 bit
Disabled when MXM = 1 in the PSR
Unimplemented opcode
Unimplemented integer unit instruction execution
Privilege violation
User level attempted access to control registers
User level used of unm option
User level traps to vectors 0-127
Bounds check violation
Invalid instruction operand failure
Integer divide
Zero divide or div, divu
Negative operands for div, divu
Integer overflow
2's complement overflow for signed integer arithmetic
Traps

Table 8.20 lists the 88100 floating-point unit exceptions.

Table 8.20
88100 Floating-Point Unit Exceptions

Floating-point precise exceptions
Floating-point unit disabled
Floating-point integer conversion overflow
Floating-point unimplemented op code
Floating-point privilege violation
Floating-point reserved operand
Floating-point divide by zero
Floating-point imprecise exceptions
Floating-point underflow
Floating-point overflow
Floating-point invalid

VBR contains the address of the 88100 vector table. The vector table contains 512 vectors. Each vector corresponds to an exception. Each vector address contains the first two instructions of its exception routine. The instruction stored in the first vector is usually a branch instruction such as ‘br START’ (delayed branch) where START is the starting address of the

exception routine. The second vector is normally used to save the current SP, and therefore the instruction such as ‘ldr r31, cr17’ is stored at the second vector. The first instruction of the exception routine should load the new SP to be used in the exception routine by using an instruction such as ‘ldr r31, cr18’. The last instruction of the exception handling routine should be ‘ret’ which restores the preexception state.

Table 8.21 shows the 88100 exception vector table.

Table 8.21
88100 VBR, Vector Table
The 88100 determines the starting address of the service routine by reading two consecutive 32-bit words pointed to by VBR. The VBR contents are automatically computed by the 88100 by adding VBR to (8 * vector number). For each exception type, there is a predefined vector number. The 88100 multiplies this vector number by 8 to find the offset. This offset is added to the initial VBR contents to find the VBR contents for the particular exception.

QUESTIONS AND PROBLEMS

8.1 i) Identify the 68030 registers that are not included in the 68020.
     ii) Name a 68030 register which is included in the 68020 but formatted in a different way from the 68020. Why is it structured differently?

8.2 What are the functions of 68030 REFILL, STATUS, and STEAM pins?

8.3 i) What is the difference between 68030 DSACK and STEAM?
     ii) What are the minimum bus access times in clock cycles for 68030 synchronous and asynchronous operations?

8.4 What conditions must be satisfied before a cache hit occurs for either instruction or data cache in the 68030?

8.5 Assume user data space. The 68030 executes the instruction CLR.W (A1) with [A1] = $20507020 and [20507002] = $1234 with the following information in data cache:

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>LW0</th>
<th>LW1</th>
<th>LW2</th>
<th>LW3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>12345678</td>
<td>12345678</td>
<td>12345678</td>
<td>12345678</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>12345678</td>
<td>12345678</td>
<td>12345678</td>
<td>12345678</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>12345678</td>
<td>12345678</td>
<td>12345678</td>
<td>12345678</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>12345678</td>
<td>12345678</td>
<td>12345678</td>
<td>12345678</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>12345678</td>
<td>12345678</td>
<td>12345678</td>
<td>12345678</td>
</tr>
</tbody>
</table>

Will a cache hit occur? Why or why not?

8.6 What are the functions of 68030 CBREQ and CBACK?

8.7 Write a 68030 instruction sequence to clear the entry for instruction cache for address $00000000.

8.8 For a page size of 4K bytes, translate the 68030 logical address $00000240 to physical address. Assume the following data:

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td>0002</td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td>0004</td>
<td>0005</td>
</tr>
<tr>
<td></td>
<td>0006</td>
<td>0007</td>
</tr>
</tbody>
</table>

8.9 Identify the inputs and outputs that affect the 68030 MMU.

8.10 i) How many levels of translation tables does the 68030 MMU provide?
      ii) What is the maximum number of page descriptor entries in the ATC?

8.11 Assume the following data contents of the 68030 CRU:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

i) Is this a page or table descriptor?
ii) What is the address of the next table?
iii) What is the allowed range of logical addresses for the next table?
8.13 Determine the contents of the 68030 TC which will enable MMU, SRF, disable function code look-up, and will permit 2 levels of equal size look-up with a page size of 1K byte for a 30-bit address.

8.14 Show the contents of the 68030 TTO for translating addresses $20000000-$21FFFFFFF in supervisor data space. These addresses should be read-only and cacheable.

8.15 Write a 68030 instruction sequence for initializing TC, CRP, and SRF registers from memory pointed to by A2. Can you initialize all MMU registers by this instruction? If not, list the register or registers.

8.16 Compare 68030 on-chip MMU features with those of the 68851 MMU.

8.17 What is meant by configuration error exception in the 68030?

8.18 In the 68030 MMU (demand paging system), what happens when a task attempts to access a page not in memory? What action should be taken by the demand paging operating system?

8.19 Summarize the basic features of RISC microprocessors. Identify how some of these features are implemented in the 88100.

8.20 Describe briefly the functional blocks included in the 88200. What does the 88200 provide to a 88100 system?

8.21 What is the maximum number of the 88200 that can be present in one 88100 processing mode?

8.22 How many pipelines are in the 88100?

8.23 What operations are controlled by the 88100 register scoreboard?

8.24 Discuss briefly the 88100 exceptions.

Questions and Problems

8.25 Since there is no return from subroutine instruction, how does the 88100 return from subroutine?

8.26 What 88100 floating-point control registers can be accessed by the user mode program?

8.27 Identify the stack pointer register in the 88100.

8.28 Show the contents of registers after the 88100 executes the following instructions:
   i) `str h r1, r2, 0x0A`
   Assume `[r1] = 0000 0020h`
   `[r2] = 0000 0000h`
   Memory

   0800 1A 87 1F 25
   0804 01 00 04 30
   0808 1F 12 26 18
   080C 68 58 AA FF

   All numbers are in hexadecimal.

8.29 Find the contents of r5 after execution of the following 88100 instructions:

   i) `mask u r5, r2, 0xFFFF`
   Assume `[r5] = AAAA 0100h`
   `[r2] = 0020 05FFh`
   `[r6] = 7777 7777h`

   prior to execution of each of the above instructions.

8.30 Write an 88100 instruction sequence to logically shift the content of r2 into r1 to the right by 8 bits.

8.31 What is the effect of the 88100 `tbl r0, r1, 200` instruction?

8.32 What are the functions of 88100 CS/UC, BE0-BE3, C0, and C2 pins?

8.33 What 88100 registers are affected by hardware reset?

8.34 Discuss briefly the 88100 exceptions.
Chapter 9

PERIPHERAL INTERFACING

This chapter describes interfacing characteristics of a microcomputer with typical peripheral devices such as hexadecimal keyboard and display, DMA controller, printer, CRT (Cathode Ray tube) terminal, floppy disk, and coprocessor.

9.1 BASICS OF KEYBOARD AND DISPLAY INTERFACE TO A MICROPROCESSOR

A common method of entering programs into a microcomputer is via a keyboard. A popular way of displaying results by the microcomputer is by using seven segment displays. The main functions to be performed for interfacing a keyboard are

1. Sense a key actuation.
2. Debounce the key.
3. Decode the key.

Let us now elaborate on the keyboard interfacing concepts. A keyboard is arranged in rows and columns. Figure 9.1 shows a $2 \times 2$ keyboard interfaced to a typical microcomputer. In Figure 9.1, the columns are normally at a HIGH level. A key actuation is sensed by sending a LOW to each row one at a time via PA0 and PA1 of port A. The two columns can then be input via PB2 and PB3 of port B to see whether any of the normally HIGH columns are pulled LOW by a key actuation. If they are, the rows...
can be checked individually to determine the row in which the key is down. The row and column code in which the key is pressed can thus be found.

The next step is to debounce the key. Key bounce occurs when a key is pressed or released — it bounces for a short time before making contact. When this bounce occurs, it may appear to the microcomputer that the same key has been actuated several times instead of just once. This problem can be eliminated by reading the keyboard after 20 ms and then verifying to see if it is still down. If it is, then the key actuation is valid.

The next step is to translate the row and column code into a more popular code such as hexadecimal or ASCII. This can easily be accomplished by a program.

There are certain characteristics associated with keyboard actuations which must be considered while interfacing a microcomputer. Typically, these are two-key lockout and N-key rollover. The two-key lockout takes into account only one key pressed. An additional key pressed and released does not generate any codes. The system is simple to implement and most often used. However, it might slow down the typing since each key must be fully released before the next one is pressed down. On the other hand, the N-key rollover will ignore all keys pressed until only one remains down.

Now let us elaborate on the interfacing characteristics of typical displays. The following functions are to be typically performed for displays.

1. Output the appropriate display code.
2. Output the code via right entry or left entry into the displays if there is more than one display.

The above functions can easily be realized by a microcomputer program. If there is more than one key, they are typically arranged in rows. A row of four displays is shown in Figure 9.2. Note that in Figure 9.2, one has the option of outputting the display code via right entry or left entry. If it is entered via left entry, then the code for the most significant digit of the four-digit display should be output first, then the next digit code, and so on. Note that the first digit will be shifted three times, the next digit twice, the next digit once, and the last digit (least significant digit in this case) does not need to be shifted. The shifting operations are so fast that visually all four digits will appear on the display simultaneously. If the displays are entered via right entry, then the least significant digit must be output first and the rest of the sequence is similar to the left entry.

Two techniques are typically used to interface a hexadecimal display to the microcomputer. These are nonmultiplexed and multiplexed. In nonmultiplexed methods, each hexadecimal display digit is interfaced to the microcomputer via an I/O port. Figure 9.3 illustrates this method.

ECD to seven-segment conversion is done in software. The microcomputer can be programmed to output to the two display digits in sequence. However, the microcomputer executes the display instruction sequence so fast that the displays appear to human eyes at the same time.

Figure 9.4 illustrates the multiplexing method of interfacing the two hexadecimal displays to the microcomputer.
The keyboard/display functions are typically performed by the microprocessor. The keyboard/display functions are then performed by the chip independent of the microprocessor.

The amount of keyboard/display functions performed by the controller chip varies from one manufacturer to another. However, these functions are usually shared between the controller chip and the microprocessor.

9.1.1 HEXADECIMAL KEYBOARD/DISPLAY INTERFACE TO THE 8085

In order to illustrate the hexadecimal keyboard/display interfacing concepts, the 8085/8156/2716-based microcomputer of Figure 2.32 will be utilized. The I/O map of the microcomputer is provided in the following:

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSR</td>
<td>08H</td>
</tr>
<tr>
<td>Port A</td>
<td>09H</td>
</tr>
<tr>
<td>Port B</td>
<td>0AH</td>
</tr>
<tr>
<td>Port C</td>
<td>0BH</td>
</tr>
</tbody>
</table>

The 2716 and 8156 memory maps are provided in the following:

### 2716 Memory Map
- 0000H through 07FFH
- 0800H through 08FFH

Figure 9.5 shows the microcomputer’s interface to a hexadecimal keyboard and two seven-segment hexadecimal displays.

The keyboard contains a 4 x 4 matrix keypad. There are eight 10K resistors which connect all columns and rows of the keypad to 5 volts such that all bits of port A will be HIGH when no key is activated.

The display includes two hexadecimal displays (TIL811). The TIL811s have on-chip decoders.

A look-up table will be used to store row and column codes for each hexadecimal digit in the 2716 EPROM.

In the software, port A is first read and checked to determine whether the previous key has been released. This will eliminate any problems when someone holds the key for a long time. When the key is released, it is debounced by calling a subroutine providing a 20-ms delay. The 8156 RAM will be used as the stack. The program will then input the rows and columns of the keyboard via port A and check for key closure. If the key...
closure is sensed, it is debounced again by waiting 20 ms, and the binary code corresponding to the key pressed is determined by using the look-up table. Rows and columns are connected to input port A. Port C is configured as an output port. The rows are also connected to port C. The rows are grounded by sending zeros to all rows via port C and key closure is sensed by inputting the column data via port A.

In order to display the keyin data, all displays are cleared first. The least significant display is the first keyin data and is stored at memory location 0800H. When the second key is pressed, the first data is displayed in the left display and the second data is displayed on the right display. The second display is also stored in location 0800H. When the third key is pressed, the leftmost display is discarded. The second key from location 0800EH is displayed in the left display and the third key is displayed on the right display. The third key is also stored in location 0800H and the process continues.

The assembly language program listing is given in Figure 9.6. The data in TABLE of Figure 9.6 are obtained by inspecting the keys of Figure 9.5. For example, the code for the E key is 1011 0111. This is because when a LOW is sent on the top row and the E key is pressed, the top row and the column connected to bit 6 of port A (normally HIGH column is grounded by activating the E key) becomes zero. This makes port A data B7H. Similarly, the codes for the other keys in TABLE can be determined.

```
1  CSR    EQU DBH
2  PORTA EQU 00H
3  PORTB EQU 01H
4  PORTC EQU 02H
5  START  MVX A, DBH
6  OUT CSR
7  LVI SP, 0B00H
8  MVX A, SP
9  OUT PORTB
10  NOP
11  MVX A, 0DH
12  OUT PORTB
13  STA 0B00H
```

**FIGURE 9.6 Assembly language program for hex keyboard and displays.**
842 Peripheral Interfacing

Chapter 9

843 Basics of Keyboard and Display Interface

14 REPEAT XEA A : Clear A
15 OUT PORTC : Send zero to rows
16 KEY OPEN IN PORTA : Input Port A
17 CPI OFH : Compare with OFH
18 RJNZ KEY OPEN : All keys are open
19 IF Z = 1, otherwise
20 RJNZ CALL DEBOUNCE : Wait for previous
21 NEXT ROW CFI OFH : Key to open
22 CALL DEBOUNCE : Delay 20ms
23 KEY CLOSE IN PORTA : Input Port A
24 NEXT ROW CFI OFH : Compare with OFH
25 RJNZ KEY CLOSE : Check for
26 CALL DEBOUNCE : Key closure
27 MJW A, OFTH : Set A to all ones
28 ORA A : Clear carry
29 RJNZ CALL DEBOUNCE : Rotate zero to first row
30 MOV B, A : Save row mask in B
31 OUT PORTC : Output to all rows
32 IN PORTA : Read row and column
33 MOV C, A : Save row and column code
34 ANI OFTH : Mask row code
35 CPI OFH : Check column for HIGH
36 CALL DECODE : If Z = 0, decode key
37 STC : IF Z = 1, key not found
38 MOV A, B : Move one to next
39 JMP NEXT ROW : row
40 DECODE LXH, TABLE : Load starting address
41 JMP NEXT ROW : of lookup table
42 MOV B, OFH : Initialize character counter
43 MOV A, C : Get row and column code
44 OUT PORTC : Compare row and column code
45 JZ DONE : with lookup table
46 INX H : If key found, go to DONE
47 DCR B : Increment character counter

FIGURE 9.6 continued.
In order to explain the program logic of Figure 9.6, let us display E<sub>16</sub> in the two-digit display of Figure 9.5. The above display will take place in the following order: the E key will have to be pressed first and will be displayed on the right display. The E key code will then be saved in location 0800H until the second key '2' is pressed. When the key '2' is activated, the contents of 0800<sub>16</sub> will be sent to the left display and '2' will be sent to right display, and '2' is saved in 0800<sub>16</sub> until the next key is pressed.

In Figure 9.6, lines 1–13 initialize ports, output zeros to the two displays, and store zeros in location 0800<sub>16</sub>. Note that initially zeros are output to displays. Lines 14–23 detect whether the previous key is released. If it is, then the key is debounced for 20 msec. This will eliminate any problems if someone holds a key for a long time. The program then detects a key closure. If a key is pressed, it is debounced for 20 ms; otherwise, the program stays in the KEYCLOSE loop.

Lines 24–32 determine exactly which key is pressed. This is done by moving all ones into the accumulator in Line 24. The CARRY is cleared and a zero is moved from this carry to a particular bit position in the accumulator. These data are used to output zero to each row in sequence. The rows and columns are read via Port A. If a key is pressed in a particular row, the column connected to that row via the depressed key will also be zero. For example, if the E key is pressed, Code B7<sub>16</sub> for E will appear at Port A. Since the E key is pressed in this example, the program branches from line 33 to line 37 and decodes the key. The program then branches to the label DONE; otherwise, the program loops through NEXTROW until the key is found. Once the E key is found, appropriate data for the key and ones to the two LATCH lines are sent via Port B (lines 46 and 47). Instructions in lines 49 and 50 enable the right LATCH line and display E (in this case) in the right-most display.
If the 8279 is to be memory mapped, that is, to appear as a group of memory locations, we will have to demultiplex the address/data lines to provide the entire 16-bit address bus. This can be done by enabling an octal latch such as the Intel 8282 or a TTL 74153 by using the 8085 ALE (Address Latch Enable). With ALE, the octal latch can store the lower byte of the address bus, thereby providing the entire 16-bit address bus for decoding. This is shown in Figure 9.9.

The 8279 has an 8-bit data bus which supplies control information and the exchange of data. This must be connected to the 8085’s address/data bus.

The 8279 has two signals which control the direction of data flow on the data bus. These signals are RD and WR. When RD is true, the 8085 is reading data from the 8279. When WR is true, the 8085 is writing data to the 8279. To simplify the chip select circuitry, these signals should only be active during either an I/O cycle or memory cycle, depending on the mapping selected for the 8279. The 8085 does not generate these two signals directly, but they can be generated easily.

The 8085 provides three signals which define the operation in progress. They are WR, RD, and I/O/M. The RD and WR signals specify the direction of data flow, while the I/O/M signal specifies the operation in
Peripheral Interfacing

Chapter 9

Basics of Keyboard and Display Interface

I/O devices, a more complete decoding of the address is needed to protect against bus conflicts. But in a simple system with few devices, no real decoding is needed. Each device can occupy many I/O locations.

The reset pin on the 8279 serves to reset the device during power on or reset events. This pin can be connected to the 8085 reset out pin.

The CLK pin provides the clock for the internal timing of the 8279. This clock must be greater than 200 kHz for the internal timers to provide the correct delays. This can be connected to the 8085's CLK out pin or other frequency source.

The 8279's IRQ pin signals the 8085 that the 8279 has data available and needs to be read. In an interrupt-driven system, the IRQ signal can be connected to the 8085's RST5.5, RST6.5, or RST7.5 pins to interrupt the 8085 when data are available. This will cause the 8085 to execute an interrupt service routine to read the 8279. In noninterrupt-driven systems this line is not used; the 8085 can determine if the 8279 needs service by reading the status register of the 8279.

A complete interface of the 8279 to an 8085 can be seen in Figure 9.11. This implementation is I/O mapped, interrupt driven, and uses simple decoding of the address bus.

9.1.2.2 Interfacing to Keyboard Display Hardware

The 8279 interfaces typical microprocessors such as 8048 and 8086 to a keyboard and a display device. Since these two sections are separate, we first discuss the display interface.

The display interface is relatively simple. The 8279 provides the buffers,
Peripheral Interfacing

Chapter 9

the multiplexing logic, and the refreshing needed to drive the display. This greatly simplifies the necessary hardware. In fact, if you only need a four-character display, you only need to add the display driver necessary for the particular display you are using.

![Diagram of 8279 interface]

**Figure 9.1** A complete 8085/8279 interface.

The 8279 provides refreshing of multiplexed displays. Seven-segment codes for the digits to be displayed are sent to the FIFO RAM in the 8279. The 8279 then automatically outputs the code for one digit and displays it. After some time delay (about one millisecond) the 8279 outputs code for the next digit and displays it. The process is repeated until all digits are displayed. The 8279 then goes back to the first digit.

The 8279 has two buffers: A buffer and B buffer. These buffers can be used together to form two 4-bit x 16-word displays, or they can be used together to form an 8-bit x 16-word display. Writing data into or reading data from the buffer is controlled by the mode selected, and will be discussed in the next section on software interfacing.

The 8297 provides data from the internal buffer on the out A and out B pins. Each of these ports is 4 bits wide. These data are synchronized with the scan lines, S0 through S3, to provide multiplexing. These scan lines are either encoded or decoded. In the decoded mode, only one scan line is active at one time. Thus, only four characters are displayed. In the encoded mode, the scan lines form a binary sequence counting from 0 to either 8 or 16, depending on the options selected. In this mode, an external decoder, such as a 4- to 16-line decoder, would have to be used. In the decoded mode, the scan lines are active low, while in the encoded mode they are active high.

There is one additional signal provided by the 8279 for the display interface. This is BD or Blank Display. This active low line goes true when the display should be blanked, either by command or between successive words. Note that the display is blanked to prevent ghosting of displays from one digit to the next when the display is switched to the next one.

If you use the two outputs together to form an 8-bit x 16-word display, you can directly drive a multiplexed LED display. Seven of the bits control the seven segments, and the eighth bit can control the decimal points. With this setup, you can display limited alphanumericics, or even special graphics. (A display much like the simple handheld games could be fashioned in this way.) For this application, you would only need to add mode and cathode drivers to buffer the signals from the 8279 and decoder chip. A sample of this display is shown in Figure 9.12.

The purpose of the keyboard section of the 8279 is to scan a keyboard, detect a key closure, and store the code for the pressed key in the internal FIFO RAM where it can be read by the microprocessor.

The keyboard interface is organized as either an 8 x 8 matrix or a 4 x 8 matrix of SPST switches. The keyboard uses the same four scan lines that the display uses (SLO-3).

The keyboard can operate in two different modes: decoded (4 x 8) or encoded (8 x 8). The scan lines behave exactly as described for the display interface. In the decoded mode, the scan lines directly do the scanning, while in the encoded mode, the scan lines must be externally decoded. Note that in the encoded mode only the first 2 bits or eight lines should be used for the keyboard. The keyboard section and the display section must be in the same scan mode, meaning that if the keyboard scan is encoded, the display scan must be encoded.

Two additional keys are provided for by the 8279. These are for the
SHIFT and CONTROL keys. These keys do not trigger the 8279 themselves, but are recorded in the FIFO along with a key in the matrix when that key is pressed. This gives you an effective 356 codes which can be generated by the 8279. A simple circuit for 64 keys, SHIFT, CONTROL, and a 16 character display is shown in Figure 9.13.

FIGURE 9.12 8279 display interface using OUT A and OUT B together.

In the scanned sensor mode, the FIFO buffer is converted into an 8 x 8-bit map. Each bit in the map represents the state of the switch at the same location in the keyboard matrix. If the switch is closed, then the RAM location corresponding to that switch will be a logical one. No key debouncing is performed by the 8279, and the SHIFT or CONTROL status is not recorded. In this mode, the 8279 signals any change in the matrix by raising IRQ. A bit is also set in the status register.

In the scanned sensor mode, the sensors do not have to be switches, but can be anything that produces a 1 or a 0 in response to the scan lines.

FIGURE 9.13 8279 interface to a keyboard and display.
A third mode is available for entering data into the FIFO of the 8279. This is the strobed input mode. In this mode the data present on the Return Lines (RLs) are loaded into the FIFO by the rising edge of the CNTL/STB pulse. In this mode the scan lines do not mean anything to the keyboard section, they only control the display hardware.

### 9.1.2.4 Software Interface

For the 8279 to work properly, it must be sent commands programming it for the specific hardware configuration. These commands are sent to the command port of the 8279 (A0 = high). The data are read from or sent to the data port of the 8279 (A0 = low). Status information is read from the command port.

The keyboard/display mode set command defines the mode the 8279 will operate in. This command is diagrammed below:

```
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>K</td>
<td>K</td>
</tr>
</tbody>
</table>
```

DD

- **0 0**: 8-bit character display — left entry
- **0 1**: 16-bit character display — left entry
- **1 0**: 8-bit character display — right entry
- **1 1**: 16-bit character display — right entry

KKK

- **0 0 0**: Encoded scan keyboard — two-key lockout
- **0 0 1**: Decoded scan keyboard — two-key lockout
- **0 1 0**: Encoded scan keyboard — N-key lockout
- **0 1 1**: Decoded scan keyboard — N-key lockout
- **1 0 0**: Encoded scan sensor matrix
- **1 0 1**: Decoded scan sensor matrix
- **1 1 0**: Strobed input, encoded display scan
- **1 1 1**: Strobed input, decoded display scan

The DD field sets the display length and the entry mode. The length of the display can be 8 or 16 words long, and we can enter data from the left or the right.

9.1 Basics of Keyboard and Display Interface

In left-entry mode, characters are entered from the left as on a typewriter. When in the autoincrementing mode, each new character will appear to the right of the old character. If you are using a 16-word display, the seventeenth character will replace the first character entered.

In right-entry mode, the characters are entered at the right side of the display and scrolled to the left with each new character, much like on a calculator. In this mode, the seventeenth character causes the first character entered to scroll off the display.

The KKK field sets the keyboard and scan mode. The scan modes have been discussed in the hardware section, along with the scanned sensor and strobed input modes.

The keyboard mode is either two-key lockout or N-key lockout. These modes determine how the 8279 deals with the problem of rollover, that is, when two or more keys are depressed at once. In two-key lockout the 8279 recognizes the first key pressed, and additional keys pressed are ignored until the first key is released. In N-key lockout, keys pressed simultaneously are entered into the FIFO in the order that the 8279 finds them. In this mode, a special error mode can be set. This error mode will consider a simultaneous key depression as an error. A bit is set in the status register, any further writing into the FIFO is inhibited, and an interrupt is requested. This error mode is set by the end interrupt/error mode set command, described later.

The 8279 has programmable counters to divide the CLK signal to 100 kHz for the internal timing. The value is set by the use of the program clock command. The P field contains the divisor necessary to give a 100-kHz internal clock. This value is in the range 2 to 31. For a 2-MHz CLK signal, the divisor would be 20,040 or 14.

Program clock

```
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P</td>
<td>P</td>
</tr>
</tbody>
</table>
```

When the microprocessor sends a read FIFO/sensor RAM command, the 8279 responds to each successive read from the FIFO. This command is shown below:

```
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
</tr>
</tbody>
</table>
```

When the read FIFO/sensor RAM 10 command is not used, the 8279 will output the current data from the FIFO.
In the scanned sensor mode, AAA selects one of the eight rows of the RAM to read. AI is the autoincrement bit. If this is true, the RAM address will automatically increment after the read. If the AI bit is set, reading the RAM will not clear the interrupt. The AAA and AI fields are ignored during any other mode.

The read display RAM command enables the microprocessor to read the data from the display buffer. This command is diagrammed below:

Read display RAM

\[
\begin{array}{c|cccc}
\text{MSB} & 0 & 1 & \text{AI} & \text{A} & \text{A} & \text{A} \\
\text{LSB} & & & & & & \\
\end{array}
\]

The AAA field sets the address for this read, and the AI field controls the autoincrement mode. Since the same counter is used for reading and writing to the display RAM, this command also sets the address and increment mode for both reading and writing.

The write display RAM command sets up the 8279 for a write to the display buffer. The addressing and autoincrement modes are identical to the Read Display RAM command, described before.

Write display RAM

\[
\begin{array}{c|cccc}
\text{MSB} & 1 & 0 & \text{AI} & \text{A} & \text{A} & \text{A} \\
\text{LSB} & & & & & & \\
\end{array}
\]

This command does not affect the source of subsequent data reads; the CPU will read from whichever data source (FIFO/sensor RAM or display RAM) was last selected. If the source was the display RAM, the write display RAM command will change the address of the next read location.

The display inhibit/blanking command is used to inhibit writing to the display RAM or in blanking the display. This command makes using two 4-bit ports easier in that the inhibit and blanking functions can be performed on each of the two ports independently.

Display inhibit/blanking

\[
\begin{array}{c|cccc}
\text{MSB} & 1 & 0 & \text{X} & \text{IW} & \text{IW} & \text{BL} & \text{BL} \\
\text{LSB} & & & & & & & \text{X = don't care} \\
\end{array}
\]

The IW bits are used to inhibit writing to the A port or B port. If we are using BCD decoding for the two 4-bit displays, we can inhibit writing to one display while writing to the other. Port B is the lower nibble of the data written to the display, with port A being the upper nibble.

The BL bits blank the display of the selected port. The code that is sent to the display during blanking is specified by the clear command described later. When using the display as a single 8-bit display, both BL bits must be set to blank the display.

The clear command determines the codes sent to the display when it is blanked, and can also clear the display RAM and reset the device. This command, shown below,

Clear

\[
\begin{array}{c|cccc}
\text{MSB} & 1 & 1 & \text{X} & \text{EN} & \text{CD} & \text{CD} & \text{CD} & \text{CA} \\
\text{LSB} & & & & & & & & \\
\end{array}
\]

consists of one field and three enable bits. The CD field determines the blanking code for the displays.

<table>
<thead>
<tr>
<th>CD</th>
<th>CD</th>
<th>Blanking code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>All zeros (X = don't care)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Hex 20 (ASCII space)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>All ones</td>
</tr>
</tbody>
</table>

- The EN bit enables clearing of the display when true. During display clear, the display RAM may not be written into. This state is recorded in the status register, and remains true until the display is finished clearing.
- The CF bit, if true, clears the FIFO status, the interrupt line is reset, and the sensor RAM pointer is reset to 0.
- The CA bit, when set, has the combined effect to the EN and CF bits; in addition, it resynchronizes the internal timing chain.

The end interrupt/error mode set command lowers the interrupt request line and enables further writing into RAM. This command also sets the special error mode for the N-key rollover mode described earlier.

End interrupt/error mode set

\[
\begin{array}{c|cccc}
\text{MSB} & 1 & 1 & \text{X} & \text{X} & \text{X} & \text{X} & \text{X} \\
\text{LSB} & & & & & & & \text{X = don't care} \\
\end{array}
\]
Peripheral Interfacing

Chapter 9

When the E bit is set and the keyboard mode is set to N-key rollover, the special error mode is set.

Now that we have discussed all the commands, we will now look at the data formats of the various registers contained within the 8279. The FIFO status word contains information as to the state of the 8279. This word is diagrammed below:

```
FIFO status word
  MSB  |  LSB
------|------
  D4   |  S/E  |  O  |  U  |  F  |  N  |  N  
```

The NNN field indicates the number of characters contained in the FIFO during keyboard or strobed input modes. This field has no meaning when the scanned sensor mode.

- The E bit indicates the FIFO is full and needs to be read before another character is entered. This bit has no meaning during the scanned sensor mode.
- The U bit indicates an underrun error has occurred. That is, the CPU has read an empty FIFO. This bit has no meaning during the scanned sensor mode.
- The O bit indicates a FIFO overrun has occurred. That is, a character was entered into a full FIFO. This bit has no meaning during the scanned sensor mode.
- The S/E bit has two functions, determined by the mode the 8279 is operating in. During the scanned sensor mode, this bit is set to show that at least one sensor closure indication is contained in the sensor RAM. In the N-key rollover mode, if the special error mode has been set, this bit indicates a simultaneous multiple error has occurred.
- The D4 bit indicates the display is unavailable because a clear display or clear all command has not completed.

The data format for the FIFO during keyboard mode is shown below:

```
Keyboard data
  |  MSB  |  LSB
  |------|------
  C   |  S   |  SC  |  SC  |  RL  |  RL  |  RL  
```

This byte represents the position of the switch matrix, along with the control and shift lines.

9.1.3 AN 8085-BASED MICROCOMPUTER USING THE 8279 FOR KEYBOARD/DISPLAY INTERFACE

In order to illustrate the application of the 8279, an example is given in the following to design an 8085-based microcomputer.

The 8085 microcomputer design uses two 8155 RAM and 1/O, a 2716 EPROM, a 6116 RAM, 8279 keyboard controllers, a multiplexed 16-key keypad, six T1211 displays, and other support chips. The EPROM provides the system with 2048 bytes of ROM. 2560 bytes of RAM are available to the system through one 6116 and two 8155 chips. Memory-mapped I/O is used in this design. Two I/O ports are used by the system for displays, leaving four ports for the user. All interrupts are tied to ground.

The monitor routine consists of four control functions to change the address field, change the data field, enter data to a memory location specified by the address field, and send the microprocessor to an address specified by the address field. The address field is automatically incremented with each data entry, and the new address and data are updated on the displays. User programs may call on monitor subroutines to display information or implement a delay. The four monitor functions are accessed by using a shift key along with the hex keypad. User programs can do a graceful halt by returning control to the monitor routine which then resumes polling of the keyboard controller.

A block diagram (Figure 9.14) illustrates the main components of the system. Figure 9.15 shows a detailed hardware schematic of the microcomputer.

9.1.3.1 Statement of the Problem

The objective of this project is to build an experimentee's microcomputer that will illustrate the application of the 8279. A 74LS138 with its eight select lines provides for future expansion. (Currently only five lines are being used. Also with A14 of the 8085 unused, the addition of another LS138 selected by A14 could provide another eight select lines.) A memory-
mapped I/O scheme is arbitrarily chosen. The 8279 keyboard controller was chosen because the shift and control/strobe lines triple the functions of a 16-key entry pad. TI311 displays are used because of on-chip decodes. Since these displays are not compatible with the output display ports of the 8279, they are driven by 8155 I/O ports.

9.1.3.b Hardware Description
9.1.3.6i Power-On Reset

When power is applied to the 8085 it is reset by the power-on reset circuitry. A manual reset is included so that the program counter can be set
to zero without turning power off. Schmitt-triggered inverters were included for increased reliability. Thus, the circuit functions so that in the case of a power transient, the diode discharges the capacitor quickly, ensuring the generation of a reset pulse with the reappearance of power.

### 9.1. Basics of Keyboard and Display Interface

The 8085AHI is an enhanced 8085A microprocessor implemented in NMOS, depletion load, silicon gate technology (SiGe MOS). At 3 MHz, ICC (power supply current) is 135 mA, 20% lower than the 8085A. The instruction cycle is 1.8 µs.

A15, along with RD, WR, and ALE, is propagated through an LS244 line driver. A15 serves as IO/M for memory mapping and it is also tied to A0 of the 8279. Configuration commands to the 8279 and status information require a high A0; data transfer a low A0.

A11, A12, A13 are decoded through an LS138 to enable the various chips. The resulting memory addresses are contained in the table below.

### 9.1.3. Memory Map

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000H - 007FH</td>
<td>ROM MONITOR ROUTINE</td>
</tr>
<tr>
<td>8800H, RD</td>
<td>FIFO STATUS (LOA 8800H)</td>
</tr>
<tr>
<td>8800H, WR</td>
<td>COMMAND (SA 8800H)</td>
</tr>
<tr>
<td>A = 2 x H</td>
<td>CMD = PROGRAM CLOCK</td>
</tr>
<tr>
<td>A = 4 x H</td>
<td>CMD = READ FIFO</td>
</tr>
<tr>
<td>0080H</td>
<td>ETC.</td>
</tr>
<tr>
<td>8155 - 1</td>
<td>DATA READ OR WRITE</td>
</tr>
<tr>
<td>1000H - 10FFH</td>
<td>SYSTEM, RAM, STACK, FLAGS, DATA</td>
</tr>
<tr>
<td>9000H</td>
<td>CSR</td>
</tr>
<tr>
<td>9001H</td>
<td>PORT A</td>
</tr>
<tr>
<td>9002H</td>
<td>PORT B</td>
</tr>
<tr>
<td>9003H</td>
<td>PORT C</td>
</tr>
<tr>
<td>9004, 5FH</td>
<td>TIMER</td>
</tr>
<tr>
<td>8155 - 2</td>
<td>1800H - 18FFH</td>
</tr>
<tr>
<td>9800H</td>
<td>RAM</td>
</tr>
<tr>
<td>9800H</td>
<td>CSR</td>
</tr>
<tr>
<td>9801H</td>
<td>PORT A: DATA TO T1311</td>
</tr>
<tr>
<td>9802H</td>
<td>PORT B: LATCH DISPLAYS</td>
</tr>
<tr>
<td>6116</td>
<td>MAIN USER RAM</td>
</tr>
</tbody>
</table>
9.1.3.b.iv Keyboard and Shift

The method to generate compatible key codes is a mixture of hardware and software. When the 8279 is configured for a decoded scan, the four scan lines have active low outputs so that a low is placed on each line in turn at a rate determined by the programming of the 8279 clock. The return lines of the 8279 have internal pull-ups which remain high until a switch closure pulls them low. For a 16-key keypad, these scan lines are driven at a low on each row in rapid succession (8.1 ms scan cycle). Each column of the keypad is connected to a Return line. If a switch is closed this will pull a particular Return line low and this data will be recorded in the 8279 FIFO/SENSOR RAM. The status of this RAM is also recorded in an 8279 STATUS register which may be polled to determine if data are available.

Eight bits of data in the FIFO RAM are available in the scanned keyboard mode. The two most significant bits (MSBs) correspond to the status of the control and shift keys, respectively. These are high until pulled low. The next three bits correspond to the (scan) row of a key closure and the three least significant bits (LSBs) to the (return) column of the key closure. In this case, these bits will vary from 000 to 011. The monitor routine checks on the scan of the shift pin and then determines which key was depressed. The 8279 automatically debounces the key depression with a 10 ms debounce (for a clock programmed to 100 kHz).

A shift key is implemented using a 556 timer chip to output a two-second pulse. Thus, simultaneous depression of the shift key with a keypad key is not required to obtain a shifted key response. The keypad key only needs to be depressed within the allotted two seconds of depressings the shift key.

9.1.3.b.v 8279 Keyboard Controller

On reset the monitor routine configures the 8279 as a decoded scan with a clock scale of 100 kHz. These parameters can be reconfigured experimentally by a user program and control can then be returned to the monitor routine to examine the results. Since the keyboard only uses four return lines, there are four more return lines that are available to the user. A user program can reconfigure the 8279 into Sensor/Matrix mode and any logic that can be triggered by the scan lines can also read data from the return line inputs. Thus, the 8279 can function as another user port.

The 8279 can be polled or an interrupt output can signal the 8885 that data are available. The monitor routine polls the 8279. Depressing a key for more than 20 ms results in another entry into the FIFO 8 x 8 RAM. Since the RAM only has room for 8 entries an overrun error (entry of another character into a full FIFO) can easily occur. The monitor routine takes care of this by polling the FIFO and clearing it if there is an overrun. If a key is continuously depressed over a number of debounce cycles, data are lost. In fact, there has been an overrun, but it has been cleared by the presence of 1.5 sec. For the membrane-type keypad that this system uses, this delay data.

A typical code sequence to program the 8279 involves two basic steps. First, a command must be sent to inform the 8279 what type of data transfer is desired, such as read/write to a RAM and programming of the clock; execution is executed which will select the 8279 with A0 high. The actual data low.

9.1.3.b.vi TIL311 Displays

The TIL311 displays with their latch, decoder, and drivers are not directly compatible with the outputs of the display ports of the 8279. The TIMING waveform of the 8279 (Figure 9.16) shows that the 30 µs before ports are a blanking code consisting of all zeros or all ones or 20H. Thus, using the scan lines to latch the TIL311 displays, a solution to this might be to decode TIL311 latch pins low and use the 8279 BD signal, inverted, output from the 8279 display ports. For this system, data to the displays and a latched signal is sent via 8155 I/O ports.

9.1.3.b.vii RAM, EPROM, and I/O

The decoding scheme for selecting and addressing RAM and I/O has
already been described and a table for the locations of the ports provided. The 2716 and 6116 have their addresses latched by an LS373. Since these two chips are basically pin compatible (with pin 21 switched from high to low for the 6116), user programs can be stored in a 2716 and the 2716 used in place of the 6116.

9.1.3.3 Software Description

   When the system is first turned on the monitor routine loads address 2000H to the address display field. The data displayed in the data display field are unpredictable. The user may now start loading data. Data will be entered from the right and the LSB of the data display will become the MSB. Once the correct data are entered a shift-F will modify the data at the memory location indicated by the address display. The address display will be incremented and the data associated with that location will be displayed in the data field. A shift-C at any point allows the user to modify the address field. Once the correct address is displayed a shift-D will allow modification of the data field. Or if the user desires to load the program counter with the address displayed a shift-E accomplishes this function. The four functions may be summarized as:

   Shift-C: modify the address field
   Shift-D: load the program counter with the address field
   Shift-E: modify the data field
   Shift-F: enter data to memory; update address and data fields

9.1.3.3 Monitor Routines: Summary

   Figure 9.17 shows the flowcharts of the monitor routine. An 8085 assembly language program for the monitor is given in Figure 9.18.

1. Start: This routine initializes the stack pointer, configures the 8279 and 8155, sets the flag for data entry, and loads the address display field with 2000H. It then goes to the polling routine.

2. Poll: The 8279 FIFO status is polled. If the status indicates an error condition, the FIFO is reset. If the status indicates no data entered, then the monitor goes to the repolling routine. If data are available, then the FIFO is read.

3. Reset Fifo: This routine resets the FIFO and then goes to the repolling routine. After processing the keyboard input most routines return to this point. User programs can return control to the monitor.
by sending the PC to this point. Results of a user program can be stored in the address and data image locations and then the user program can jump to this point. The results will be displayed and the keyboard will be ready to accept new input.

4. Repoll: In this routine, a 1.5-second delay is implemented. After the delay the subroutines to display the address image and data image are called. From here, the monitor goes back to poll, that is, to check the status of the 8279.

5. ReadFifo: This routine analyzes the data in the FIFO, first determining whether a shift function is being invoked or if the input is hex key data. If no shift is found then a determination is made as to which key was pressed. Then, based on the flag indicating whether the entry was to the data field or address field, a left shift is done and the field updated. From here the routine goes to ReadFifo. If, instead, a shift is detected, then ReadFifo determines which shifted key was depressed and then goes to the corresponding routine. Invalid shifted keys are treated as a normal hex key input. The functions of the shifted keys have already been described.

6. ADisplay: Calling this subroutine causes the data stored in locations 1002H and 1003H, the address image fields, to be displayed. DDis-
play similarly sends the contents of $1004\text{H}$ to the TI111 data displays.

7. **Delay**: This subroutine may be called to implement a delay proportional to the magnitude of the contents of the DE registers.

### 9.1.3.c.ii A Sample Program
The following program adds to 8-bit numbers in locations 2000 and 2001 and may be keyed in as follows:

1) **power-on**: 2000 is displayed in addr field
2) **data**: any hex key pressed modifies data field
3) **Shift-F**: 2001 is displayed in addr field & [2001] in data
4) **data**: next 8-bit number to add
5) **Shift-F**: this is entered and now 2002 in addr field
6) **continue to enter in program**.

```
LDA 2000H
MOV B,A
LDA 2001H
ADD B
STA 1004H: 1004H is data image location
CD 0000H: Call Display
CJ 0000H: Jmp to Reset/Fifo of monitor routine
```

7) **Shift-F**: after entering last data with a Shift-F load the starting address of the program is 2002H
8) **Shift-C**: keys pressed after this modify address display
9) 2002: enter starting address
10) **Shift-D**: this executes the program

The results of the addition should be displayed in the data field. At this point the address field still shows 2002. Press Shift-C and load 2000. Now Shift-E. The first 8-bit data now are displayed in the data field. Pressing any key will modify it so that a new number may be added with the data in 2001. Loading 2002 to the address field and Shift-D will execute the program again. If the program was terminated with a Halt instruction instead of returning control to the monitor, the manual reset switch would have to be depressed to reinitiate polling of the keypad.

### 9.2 DMA Controllers

As mentioned before, direct memory access (DMA) is a type of data transfer between the microcomputer's main memory and an external device such as disk without involving the microprocessor. The DMA controller is an LSI (Large-Scale Integration) chip in a microcomputer system which supports DMA-type data transfer. The DMA controller can send commands to the memory in the same way as the microprocessor, and therefore, the DMA controller can be considered as a second microprocessor in the system, except that its function is to perform I/O transfers. The DMA controllers perform data transfer at a very high rate. This is because several functions for accomplishing the transfer are implemented in hardware. The DMA controller is provided with a number of I/O ports. A typical microcomputer system with a DMA controller is shown in Figure 9.19.

The DMA controller in the figure connects one or more ports directly to memory so that data can be transferred between these ports and memory without going through the microprocessor. Therefore, the microprocessor is not involved in the data transfer.

The DMA controller in the figure has two channels (Channel 0 and 1). Each channel contains an address register, a control register, and a counter for block length. The purpose of the DMA controller is to move a string of data between the memory and an external device. In order to accomplish this, the microprocessor writes the starting address of memory where transfer is to take place in the address register, and controls information such as the direction of transfer in the control register and the length of data to be transferred in the counter.

The DMA controller then completes the transfer independent of the microprocessor. However, in order to carry out the transfer, the DMA controller must not start the transfer until the microprocessor relinquishes the system bus and the external device is ready.

The interface between an I/O port and each channel has typically a number of control signals which include DMAX/RQ, DMACK, and I/O read/write signals. When the I/O port is ready with an available buffer to receive data or has data ready to write into memory location, it activates the

---

397.jpg
DMAREQ line of the DMA controller. In order to accomplish the transfer, the DMA controller sends the DMACK to the port, telling the port that it can receive data from memory or send data to memory.

**FIGURE 5.19** A microcomputer system with a DMA controller.

DMACK is similar to a chip select. This is because when the DMACK signal on the port is activated by the DMA controller, the port is selected to transfer data between the I/O device and memory. The main difference between a normal and DMA transfer is that the read or write operations have opposite meanings — that is, if the DMA controller activates the read line of the port, then data are read from a memory location to the port. However, this is a write operation to the port as far as the memory is concerned. This means that a read from a memory location is a write to the port. Similarly, a write to a memory location is equivalent to a read from the port. The figure shows two types of R/W signals. These are the usual memory R/W signal and the I/O R/W for external devices. The DMA controller activates both of these lines at the same time in opposite directions. That is, for reading data from memory and writing into a port, the DMA controller activates the memory R/W HIGH and I/O R/W LOW. The I/O ports are available with two modes of operation: non-DMA and DMA.

For non-DMA (microprocessor-controlled transfers), the ports operate in a normal mode. For DMA mode, the microprocessor first configures the port in the DMA mode and then signals the DMA controller to perform the transfer. The R/W line is complemented for providing proper direction of the data transfer during DMA transfer.

The DMA controller has a HOLD output signal and a HOLD ACK input signal. For each byte transfer, the microprocessor enables the I/O port for a DMA transfer. The port, when ready, generates the DMAREQ's signal for the DMA controller. The DMA controller then activates the HOLD input signal of microprocessor, requesting the microprocessor to relinquish the bus, and waits for a HOLD acknowledge back from the microprocessor.

After a few cycles, the microprocessor activates the HOLD acknowledge and transmits the output drivers to the system bus. The DMA controller then takes over the bus. The DMA controller:

1. Outputs the starting address in the system bus
2. Sends DMACK to the I/O port requesting DMA
3. Outputs normal R/W to memory and complemented R/W to the I/O port

The I/O port and memory then complete the transfer. After the transfer, the DMA controller disables all the signals including the HOLD on the system bus and transmits all its bus drivers. The microprocessor then takes over the bus and continues with its normal operation.

For efficient operation, the DMA controller is usually provided with a burst mode in which it has control over the bus until the entire block of data is transferred.

In addition to the usual address, control, and counter registers, some
DMA controllers are also provided with data-chain registers which contain an address register, a control register, a counter, and a channel identification. These data-chain registers store the information for a specific channel for the next transfer. When the specified channel completes a DMA transfer, its registers are reloaded from the data-chain registers and the next transfer continues without any interruption from the microprocessor. In order to reload the data-chain registers for another transfer, the microprocessor can check the status register of the DMA controller to determine whether the DMA controller has already used the contents of the data-chain registers. In case it has, the microprocessor initializes the data-chain registers with appropriate information for the next block transfer and the process continues.

In order to illustrate the functions of a typical DMA controller just described, Motorola’s MC68440 dual channel DMA controller will be considered in the following.

The MC68440 is designed for the MC68000 family microprocessors to move blocks of data between memory and peripheral using DMA.

The MC68440 includes two independent DMA channels with built-in priorities that are programmable. The MC68440 can perform two types of DMA cycle stealing and burst. In addition, it can provide noncontinuous block transfer (continue mode) and block transfer restart operation (reload mode).

Figure 9.20 shows a typical block diagram of the MC68000/68440/68230 interface to a disk.

Data transfer between the disk and the memory takes place via port A of the MC68230, using handshaking signals H1-H4.

The A8/D0 through A23/D15 lines are multiplexed. The MC68440 multiplex control signals OWN, UAS (upper address strobe), DBEN (data buffer enable), and DDIR (data direction) are used to control external demultiplexing devices such as 74LS245 bi-directional buffer and 74LS73 latches to separate address and data information on the A8/D0-A23/D15 lines. The MC68440 has 17 registers plus a general control register for each of the two channels and is selected by the lower address lines (A1-A7) in the MPU mode. A1-A7 also provides the lower 7 address outputs in the DMA mode.

A1-A7 lines can select 128 (2^7) registers; however, with A1-A7 lines, only seventeen registers with addresses are defined in the range from 60H through FFH, and some addresses are not used. An example, the addresses of the channel status register and the channel priority register are, respectively, 00H and 2DH.

The MC68440 registers contain information about the data transfer such as:
Peripheral Interfacing

Chapter 9

9.2 DMA Controllers

1. Source and destination addresses along with function codes
2. Transfer count
3. Operand size and device port size
4. Channel priority
5. Status and error information on channel activity

The processor service request register (PSRR) of the MC68230 defines how the DMAREQ pin should be used and how the DMA transfer should take place, whether via handshaking or ports.

A data block contains a sequence of bytes or words starting at a particular address with the block length defined by the transfer count register. Figure 9.21 shows the data block format.

![Figure 9.21 Data block format.](image)

There are three phases of a DMA transfer. These are channel initialization, data transfer, and block termination. During channel initialization,

the MC68000 loads the MC68440 registers with control information, address pointers, and transfer counts, and then starts the channel.

During the transfer phase, the MC68440 acknowledges data transfer requests and performs addressing and bus controls for the transfer. Finally, the block termination phase takes place when the transfer is complete.

During this phase, the 68440 informs the 68000 of the completion of data transfer via a status register. During the three phases of a data transfer operation, the MC68440 will be in one of the three modes of operation: these are idle, MPU, and DMA. The MC68440 goes into the idle mode when it is reset by an external device and waits for initialization by the MC68000 of an operand transfer request from a peripheral.

The MPU mode is assumed by the MC68440 when its CS (chip select) is enabled by the MC68000. In this mode, the MC68440 internal registers can be read or written for controlling channel operation and for checking the status of a block transfer.

The MC68440 assumes the DMA mode when it takes over the bus to perform an operand transfer.

In Figure 9.20, upon reset, the MC68440 goes into idle mode. In order to initialize the MC68440 registers, the MC68000 outputs appropriate register addresses on the bus. This will enable the MC68440 CS line and places the MC68440 in the MPU mode. The MC68000 initializes the MC68440 registers in this mode. The MC68000 then executes the RESET instruction to place the MC68440 back to the idle mode.

The MC68000 now waits for a transfer request from the 68230. When the 68000 desires a DMA transfer between the disk and memory, it enables the CS line of the 68230. The 68230, when reset, activates the DMAREQ line low, which in turn drives the REQ0 line of the MC68440 to low. The MC68440 then outputs low on its BR line requesting the MC68000 to relieve the bus. The MC68000, when ready, sends a low on its BS pin. This tells the MC68440 to take over the bus. The MC68440 then enters the DMA mode and sends low on its BGACK pin to inform the MC68000 of its taking over the bus. The MC68440 transfers data between the disk and memory (inside the MC6800-based microcomputer) via the MC68230. Each time a byte is transferred, the MC68440 decrements the transfer counter register and increments the address register. When the transfer is completed, the MC68440 updates a bit in the status register to indicate this. It also asserts the DTI (data transfer complete) flag to indicate completion of the transfer.

The MC68440 DTI pin can be connected to the MC68230 PI REQ pin. The MC68230 then outputs high on the MC68440 REQ pin, which

400.jpg
in turn places a HIGH on the MC68000 BR, and the MC68000 takes over the bus and goes back to normal operation.

9.3 PRINTER INTERFACE

Microprocessors are typically interfaced to two types of printers: serial and parallel.

Serial printers print one character at a time, while parallel printers print a number of characters on a single line so fast that they appear to be printed simultaneously. Depending on the character generation technique used, printers can be classified as impact or nonimpact. In impact printers, the print head strikes the printing medium, such as paper, directly, in order to print a character. In nonimpact printers, thermal or electrostatic methods are used to print a character.

Printers can also be classified based on the character formation technique used. For example, character printers use completely formed characters for character generation, while matrix printers use dots or lines to create characters.

The inexpensive serial dot matrix impact printer is very popular with microcomputers. An example of such a printer is the LRC7040 manufactured by LRC, Inc. of Riverton, Wyoming. The LRC7040 can print up to 40 columns of alphanumeric characters. The printer includes four major parts. These are the frame, the printhead, the main drive, and the paper handling components. The LRC7040 provides 8 inputs in the basic configuration. One input turns the main drive motor ON or OFF, while the other seven inputs control the print solenoids for the printhead, using TTL drivers.

The LRC7040 utilizes a 5 x 7 matrix of dots to generate characters. The columns are labeled T0 through T4 and rows are labeled S0 through S6. Each row corresponds to one of the solenoids. The entire printhead assembly is moved from left to right across the paper so that at some time the printhead is over the column T0, then it's over column T1, and so on.

A character is generated by energizing the proper solenoids at each of the columns T0 through T4. Figure 9.22 shows how the character C is formed.

At T0, solenoids S0 through S6 are ON and at T1 through T4 solenoids S0 and S6 are active to form the character C. A number of characters can be formed by the microcomputer by sending appropriate data to the printhead to generate the correct pattern of active solenoids for each of the five instants of time. The code for the character C consists of 5 bytes of data in the sequence 7F, 41, 41, 41, 41, as follows:

<table>
<thead>
<tr>
<th>S6</th>
<th>S5</th>
<th>S4</th>
<th>S3</th>
<th>S2</th>
<th>S1</th>
<th>S0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Note that in the above, it is assumed that S1 will turn a solenoid ON and a 0 will turn it OFF. Also it is assumed that S7 is zero.

![Figure 9.22 5 x 7 dot matrix pattern for generating the character C.](image)

The interface signals to the printer include a pair of wires for each solenoid, a pair of wires for each motor (main drive motor and line feed motor), a pair of wires indicating the state of the HOME microswitch, and a pair of wires indicating the state of the LINEFEED microswitch.

Paper feed is accomplished by activating the line feed motor. The LINEFEED microswitch is activated by the print logic when the actual paper feed takes place. The control logic can use the trailing edge of the signal generated by the LINEFEED microswitch to turn the line feed motor OFF. The LRC7040 also has an automatic line feed version.

The HOME microswitch is activated when the printhead is at the left-hand edge of the paper. When the printhead is over the print area and moves from left to right, the HOME microswitch is deactivated to zero.

The solenoids must be driven from 4 ± 4 volts with a peak current of 8.6 A. An interface circuit is required at the microcomputer's output to provide this drive capability.
There are two ways of interfacing the printer to a microcomputer. These are

1. Direct microcomputer control
2. Indirect microcomputer control using a special chip called the Printer Controller

The direct microcomputer control interfaces the printer via its I/O ports and utilizes mostly software. The microcomputer performs all the functions required for printing the alphanumeric characters.

Indirect microcomputer control, on the other hand, utilizes a printer control chip such as the Intel 8295 Dot Matrix Printer Controller. The benefits of each technique depend on the specific application.

Direct microcomputer approach provides an inexpensive interface and can be appropriate when the microcomputer has a light load. The indirect microcomputer approach, on the other hand, may be useful when the microcomputer has a heavy load and cost constraint is not of concern.

9.3.1 LRC7040 PRINTER INTERFACE USING DIRECT MICROCOMPUTER CONTROL

The steps involved in starting a printing sequence by the microcomputer are provided below:

1. The microcomputer must turn the Main Drive motor (MDM) ON by sending a HIGH output to the MDM.
2. The microcomputer is required to detect a HIGH at the HOME microswitch. This will ensure that the printhead is at the left-hand margin of the print area.
3. The microcomputer is then required to send five bytes of data for an alphanumeric character in sequence to energize the solenoids. Each solenoid requires a pulse of about 400 μs to generate a dot on the paper. A pause of about 900 μs is required between these pulses to provide a space between dots.

Figure 9.23 shows a block diagram interfacing the LRC7040 printer to an MC68000/6821/6116/2716-based microcomputer.

For printing the character C is shown in Figure 9.24 assuming 68000 user mode So that USP can be initialized.

The program assumes a look-up table which stores the 8-byte code for the character C starting at 8003000. Furthermore, the program assumes that the delay routines DELAY400 for 400 μs and DELAY900 for 900 μs are available. The program prints only one character C and then stops. The program is provided for illustrating the direct microcomputer control technique for printing.

9.3.2 LRC7040 PRINTER INTERFACE TO A MICROCOMPUTER USING THE 8295 PRINTER CONTROLLER CHIP

With direct microcomputer control, the microcomputer spends time in a "wait" loop for polling the status of the HOME signal from the LRC7040 printer. In order to unlatch the microcomputer of polling the printer status and other functions, typical LSI printer controller chips such as the Intel 8295 can be used.

The 8295 is a dot matrix printer controller. It provides an interface for
Chapter 9

Peripheral Interfacing

9.3 Printer Interface

The 8295 includes a 40-character buffer. When the buffer is full or a carriage return is received, a line is printed automatically. The 8295 has the buffering capability of up to 40 characters and contains a 7 × 7 matrix character generator, which includes 64 ASCII characters. The mode selection (serial or parallel) is not software programmable and is inherent in system hardware. For example, by connecting the 8295 IRQ/SER pin to ground, the serial mode is enabled; otherwise, the parallel mode is enabled. The two modes cannot be mixed in a single application. Note that IRQ/SER pin is also on 8295 interrupt request to the processor in the parallel mode.

9.3.2.1 8295 Parallel Interface

Two 8295 registers (one for input and the other for output) can be accessed by the processor in the parallel mode. The registers are selected as follows:

<table>
<thead>
<tr>
<th>RD</th>
<th>WR</th>
<th>CS</th>
<th>Register selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Input data register</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Output status register</td>
</tr>
</tbody>
</table>

Two types of data can be written in the input data register by the processor:

1. A command to be executed. The command can be 0XH or 1XH. For example, the command 08H will enable DMA mode. On the other hand, the command 1FH will enable normal left-to-right printing for printers whose print head is on the right.

2. A character data (defined in the 8295 data sheet) such as 37H for ‘7’ or 41H for ‘A’ to be stored in the character buffer for printing.

The 8295 status is available in the output status register at all times. Typical status bits indicate whether the input buffer is full or DMA is enabled. For example, the IBF (Input buffer full; bit 1 of the status register) is set to one whenever data are written to the input data register. When IBF = 1, no data should be written to the 8295. The DE bit (DMA Enabled; bit 4 of the status register) is set to one whenever the 8295 is in DMA mode. Upon completion of the DMA transfer, the DE is cleared to zero.

microprocessors such as the 8085 and 8086 to dot matrix impact printers such as the LRC7040. The 8295 is packaged in a 40-pin DIP and can operate in either a parallel or serial communication with the 8085 or 8086. In parallel mode, command and data transfer to the printer by
The 8295 IRQ/SER pin is used for interrupt driven systems. This output is activated HIGH when the 8295 is ready to receive data. Using polling in parallel mode, the 8295 IRQ/SER pin can be input via the processor I/O port and data can be sent to the 8295 input data register.

Using interrupt in parallel mode, the 8295 IRQ/SER pin can be connected to the processor's interrupt pin to provide an interrupt-driven system.

Using polled or interrupt technique in parallel mode, the processor typically communicates with the 8295 by performing the following sequence of operations in the main program (polled) or service routine (interrupt):

- The processor reads the 8295 status register and checks the IBF flag for HIGH.
- If IBF = 1, the processor waits in a wait loop until IBF = 0. The processor writes data to be printed to the 8295 input data register. IBF flag is then set to one indicating no data should be written.

Data can also be transferred from the main memory to the 8295 via the DMA method using a DMA controller such as 8257.

The processor initializes the 8257 by sending a starting address and a block length. The processor also enables the 8295 DMA channel by sending the "ENABLE DMA" command (08H) followed by two bytes specifying the block length to be transferred (low byte first). The 8295 will then activate the DMA request line of the 8257 without any further involvement by the main processor. The DMA enable (DE) flag in the status register will be HIGH until the data transfer is completed. As soon as the data transfer is completed, the DE flag is cleared to zero and IRQ/SER is set to HIGH. The 8295 then goes back to the non-DMA mode of operation.

Figure 9.25 shows a block diagram of the 8295 DMA transfer.

Typical control signals between the 8257 and the processor include HOLD, HLD*, RD, and WR. The 8295 control signals for the processor include CS, RD, WR, RESET, and IRQ/SER. CS, RD, and WR pins are used to select either 8295 input or output register. The 8295 control signals for the printer include MOT, PFM, STB, PFEED, and HOME.

The 8295 MOT output pin, when LOW, drives the motor. The MOT output is automatically in LOW on power-up. This will make the 8295 HOME input pin HIGH, indicating that the printhead of the printer is in HOME position.

The PFM signal, when LOW, drives the paper feed motor, and this is LOW on power-up. The PFEED is an 8295 input and indicates status of paper feed. A LOW on the PFM indicates that the paper feed mechanism is disabled and a zero indicates that the S1 through S7 signals, when LOW, drive the seven solenoids of the printer. Each character datum, when written into the 8295 input data register, is automatically converted to the five-byte code by the 8295 and provides the proper ON/OFF sequence for the solenoids. The STB output is used to determine duration of solenoid activation and is automatically provided by the 8295.

9.3.2.8 8295 Serial Mode

The 8295 serial mode is enabled by connecting the IRQ/SER pin to LOW. The serial mode is enabled immediately upon power-up. The serial baud rate is programmed by the D2, D1, D0 data lines. For example, D2 D1 D0 = 001 means 150 baud rate (bits/sec) and is used to set the serial transfer data rate. In this mode, RD must be tied high and CS and WR must be tied to ground. The processor needs a UART (Universal Asynchronous Receiver Transmitter) such as the 8251. The 8295 DACK/SIN signal (data input for serial mode) must be connected to the 8251 RXD and 8151 transmit data output pin. Also, the 8295 DACK/CTS (clear to send in serial mode) must be connected to the 8251 CTS output. Note that a UART chip converts parallel to serial data and vice versa.
9.4 CRT (CATHODE RAY TUBE) CONTROLLER AND GRAPHICS CONTROLLER CHIPS

The CRT terminal is extensively used in microcomputer systems as an efficient man-machine interface. The user communicates with the microcomputer system via the CRT terminal. It basically consists of a typewriter keyboard and a CRT display. In order to relieve the microprocessor from performing the tedious tasks of CRT control, manufacturers have designed an LSI chip called the CRT Controller. This chip simplifies and minimizes the cost of interfacing the CRT terminal to a microcomputer.

The CRT controller supports all the functions required for interfacing a CRT terminal to a microprocessor. The microprocessor and the CRT controller usually communicate via a shared RAM. The microprocessor writes the characters to be displayed in this RAM; the CRT controller reads this memory using DMA and then generates the characters on the video display. The CRT controller provides functions such as clocking and timing, cursor placement, and scrolling. The CRT controller chip includes several registers that can be programmed to generate timing signals and video interface signals required by the terminal. The display functions are driven by clock pulses generated from a master clock. The CRT controller chip normally produces a special symbol such as a blinking signal or an underline on the CRT. This signal is commonly called the 'cursor'. It can be moved on the screen to a specific location where data need to be modified. The scrolling function implemented in the CRT controller moves currently displayed data to the top of the screen as new data are entered at the bottom.

In this section, fundamentals of CRT, character generation techniques, and graphics controllers will be discussed.

A typical CRT controller such as the Intel 8275 will then be considered to illustrate its basic functions. Finally, the graphics functions provided by Intel 82786 will be covered.

9.4.1 CRT FUNDAMENTALS

A CRT consists of an evacuated glass tube, a screen with an inner fluorescent coating, and an electron gun for producing electron beams. When the electrons generated by the gun are focused on the fluorescent inner coating of the screen, an illuminated phosphor dot is produced.

The position of the dot can be controlled by deflecting the electron beam by using an electromagnetic deflection technique. A complete display is produced by moving the beam horizontally and vertically across the entire surface of the screen and at the same time by changing its intensity.

Most modern CRT terminals generate the display by using horizontal and vertical scans. In the horizontal scan, the beam moves from the upper left-hand corner to the extreme right-hand of the line and then travels across the screen. After several horizontal scans, the beam reaches the bottom of the screen to complete one vertical scan. The beam then disappears from the screen and begins another vertical scan from the top. This type of scan is also called 'raster' scan. This is because the display is produced on the screen by continuously scanning the beam across the screen for obtaining a regular pattern of closely spaced horizontal lines, or raster covering the entire screen. One of the most common examples of a raster display is the home TV set. The typical bandwidth used in these TV sets is 4.5 MHz. The raster displays used with microcomputers include a wider bandwidth from 10 MHz to 20 MHz for displaying detailed information. In most modern CRT terminals, each sweep field contains the entire picture or text to be displayed.

In order to display characters, the screen is divided by horizontal and vertical lines into a dot matrix. A matrix of 5 x 7 or 7 x 9 dots is popular for representing a character. For example, a 5 x 7 dot matrix can be used to represent the number '0' as shown in Figure 9.26.

![Figure 9.26] Generation of '0' using 5 x 7 dot matrix.

To provide space around the character, one top, one left, one right, and one bottom line are left blank. Each character is generated using 5 x 7 dot matrix. Therefore, each character requires 35 dots, which can be turned
ON or OFF depending on the dot pattern required by the character. The pattern of dots is usually stored in ROM. A ROM pattern for '0' is shown in Figure 9.27.

One character requires a 35-bit word. Each row is addressed by three bits. After reading each row data, it is transferred to a parallel to serial shift register. These data are then shifted serially by a clock to the CRT. For a standard 64-character set with each character represented by a 5 x 7 dot matrix, a total of 2240-bit (64 x 7 x 5) ROM is required. Each character in the 64 (2^6) character set can be addressed using 6 address lines and three row select or scan lines (row select counter typically used) are required to identify the dot row of the character. The ROM addressing logic and parallel to serial shift register are referred to as a character generator. Also, a memory known as screen memory is required in the CRT to store the character data to be displayed. When a character is entered via the CRT keyboard, it is stored in the screen memory. The character generator compares data with the screen memory and then sends the 35-bit pattern for the matched character to the CRT.

The character generator technique described above, when combined with VLSI technology, results in graphics generation.

--

9.4 CRT Controller and Graphics Controller Chips

Graphics capability can display any figure on the CRT screen. An example of such a VLSI chip is the Intel 82786. In this chip, linked lists are used to update the display and can thus generate displays at a high speed.

Most modern graphics use the bit mapping technique rather than character generation. In order to understand bit mapping, consider a CRT screen divided into 512 by 128 dots. Each dot is called a pixel, or picture element, which can be illuminated by an electron beam. Each dot is a single bit in a 64K (512 x 128 = 65,536) by 1 RAM and is called a bit plane. If a '1' is stored at a specific bit location, the associated pixel is turned ON (black). On the other hand, if a '0' is stored, the corresponding pixel is turned OFF (white). The video refresh circuitry implemented in the VLSI chip converts the ones and zeros in the bit plane to white and black on the CRT screen.

Resolution is an important factor to be considered in graphics. In order to provide various colors and intensity, more than one bit is utilized to represent a pixel. For example, Apple's 68000-based Lisa microcomputer uses four-bit pixels on a 512 x 720 Pixel screen. Therefore, a high-speed RAM of over 1 megabit (564 x 720 x 4 = 1,048,520) is required to support such a resolution.

Therefore, graphics generation requires the bit-mapped RAM array and the LSIG video interface chip. The software involves determining the information written to the bit plane array to generate the desired graphics display. Most graphics systems generate figures by combinations of straight line segments. The software is required to generate a straight line by identifying each pixel and write information to its corresponding bit-map position.

The concepts associated with CRT controllers and graphics described above will be illustrated by using the Intel 8275 and Intel 82786 in the following.

9.4.2 INTEL 8275 CRT CONTROLLER

The INTEL 8275 is a single chip (40-pin) CRT controller. It provides the functions required to interface CRT raster scan displays with Intel microcomputer systems using 8051, 8085, 8086, and 8088. It refreshes the display by storing (buffering) the information to be displayed from memory and control the display position on the screen. The 8275 provides raster timing, display row buffering, visual attribute decoding, cursor timing, and light pen detection. The 8275 can be interfaced with the Intel...
Peripheral Interfacing

8257 DMA controller and character generator ROM for dot matrix decoding.

Figure 9.28 shows the 8275's interface to a microcomputer system and the 8257.

![Diagram of 8275 Interface](image)

The 8275 obtains display characters from memory and displays them on a row-by-row basis. There are two row buffers in the 8275. It uses one row for display, and at the same time fills the other row with the next row of characters to be displayed. The number of display characters per row and the number of character rows are software programmable.

The 8275 utilizes the 8257 DMA controller to fill the row buffer that is not being used for display. It displays character rows one line at a time.

The 8275 controller provides visual attribute codes such as graphics symbols, without the use of the character generator, and blinking, highlighting, and underlining of characters. The raster timing is controlled by the 8275. This is done by generating the horizontal retrace and vertical retrace signals on the HRTC and the VRTC pins.

The 8275 provides the light pen input and associated registers. The light pen input is used to read the registers. A command can be used to read the light pen registers. The light pen consists of a microswitch and a tiny light sensor. When the light pen is pressed against the CRT screen, the microswitch enables the light sensor. When the raster sweep reaches the output of the light pen is presented to the 8275 LPEN pin, the row and character position coordinates are stored in a pair of registers. These registers can be read by a command. A bit in the status register in the 8275 is set, indicating detection of the light pen signal. The 8275 can generate a cursor. The cursor location is determined by a cursor row register and a character position register which are added by a command to the controller.

The cursor can be programmed to appear on the display in many forms such as a blinking underline and a nonblinking underline. The 8275 does not provide scrolling function.

The 8275 outputs the line count (LCD-LC3) and character code (CC0-CC6) signals for the character generation. The LCD-LC3 signals are contents of the 8275 line counter which are used to address the character generator for the line position on the screen. The CC0-CC6 outputs of the 8275 are the contents of the row buffers used for character selection in the character generator.

The 8275 video control signals typically include line attribute codes, highlight, and video suppression. The two line attribute codes (LA0 and LA1 pin outputs) must be decoded by the dot timing logic to produce the horizontal and vertical line combinations for the graphic display defined by the attribute codes. The video suppress (VSP pin) output signal is used to blank the video signal to the CRT. The highlighted (HLGT) output signal is used to intensify the display at a specific position on the screen, as defined by the attribute codes.

The dot timing and interface logic must provide the character clock (CCLK pin) input of the 8275 for proper timing.

9.4.3 INTEL 82786 GRAPHICS CONTROLLER

The Intel 82786 is a single VLSI chip providing bitmapped graphics. It is designed for microcomputer graphics applications, including personal computers, engineering workstations, terminals, and laser printers. The 82786 is designed using Intel's CMOS III process. It is capable of both drawing and refreshing raster displays. It supports high-resolution displays.
Peripheral Interfacing

Chapter 9

with a 25-MHz Pixel clock and can display up to 256 colors simultaneously. It can be interfaced to all Intel microcomputers such as 80186, 80286, and 80386. Figure 9.29 shows a block diagram of the 80186/80286/80386 interfaces to the 82786.

![Block diagram of 80186/80286/80386 interfaces to 82786](image)

The 82786 includes three basic components. These are a display processor (DP), a graphics processor (GP), and a bus interface unit (BIU), with DRAM/VRAM controller.

The display processor controls the CRT timing and provides the serial video data stream for the display. It can assemble several windows (portions of bitmaps) on the screen from different bitmaps scattered across the memory accessible to it.

The graphics processor executes commands from a graphics command block placed in memory by the 80186/80286/80386 and updates the bitmap memory for the display processor. The graphics processor has high level video display interface-like commands and can draw graphical objects and text at high speeds.

The BIU controls all communication between the 82786, 80186/80286/80386, and memory. The BIU contains a DRAM/VRAM controller that can perform block transfers. The display processor and graphics processor use the BIU to access the bitmaps in memory.

The system bus connects the 80186/80286/80386 and system memory to the 82786. The video interface connects the 82786 to CRT or other display. The video interface is controlled directly by the display processor. The 82786 can be programmed to generate all the CRT signals for up to 8 bits/Pixel (256 colors) displays. The other interfaces are controlled by the BIU. The BIU interfaces the graphics and display processors to the 80186/80286/80386 and system memory as well as the graphics memory via the internal DRAM/VRAM controller.

The dedicated graphics DRAM/VRAM memory provides the 82786 with fast access to memory without contention with the microprocessor and system memory.

Usually, the bitmaps to be drawn and displayed, the characters, and commands for the 82786 are all stored in this memory. The 82786 DRAM/VRAM controller interfaces directly with a number of dynamic RAMs without external logic.

Figure 9.29 shows the most common configuration. The microprocessor can access the system memory, while the 82786 accesses its dedicated graphics memory simultaneously. However, when the microprocessor accesses the graphics memory, the 82786 cannot access the system memory. Also, when the 82786 accesses the system memory, the microprocessor cannot access the graphics memory.

If DMA capability is provided, the 82786 can operate in either slave or master mode. In the slave mode, the microprocessor or DMA controller can access the 82786 internal registers or dedicated graphics memory through the 82786. In the master mode, the 82786 can access the system memory.

The microprocessor software can access both system and graphics memory in the same way. When the microprocessor accesses the 82786, the 82786 runs in slave mode.

In slave mode, the 82786 appears like an intelligent DRAM/VRAM controller to the microprocessor. The microprocessor can chip-select the 82786 and the 82786 will acknowledge when the cycle is completed by asserting a READY signal for the microprocessor.

The 82786 graphics and display processor accesses both system memory and graphics memory in the same way. When the 82786 accesses system memory, the 82786 must run in master mode.

In the master mode, the 82786 acts as a second microprocessor controlling the local bus. The 82786 activates HOLD line to take control of the system bus. When the microprocessor asserts HILDA line, the 82786 takes over the bus. When the 82786 is finished with the bus, it will disable the HOLD line and the microprocessor can remove HILDA to take over the bus.

The 82786 provides two different video interfaces when using standard
Peripheral Interfacing

9.5 FLOPPY DISK INTERFACE

Most microcomputer systems use inexpensive removable magnetic media for storing information. There are two types of removable storage devices. These are magnetic tape and floppy disk. The floppy disk is faster than the magnetic tape, since it provides direct access to stored information, rather than serial access.

Information access on a floppy disk requires less than 1 second, while tape movement often requires several minutes. Because of faster speed, the floppy disk is used as an on-line mass storage device, while the magnetic tape is used as back-up storage due to its large capacity.

Since floppy disk is very popular as an on-line secondary storage device with microcomputer systems, the structures of floppy disk and floppy disk drive will be covered in the following. Also, typical features associated with floppy disk controllers relating them to the Intel 82072 floppy disk controller will be included.

9.5.1 THE FLOPPY DISK

The floppy disk is a circular piece of thin plastic material provided with a magnetic coating and contained in a protective jacket. The floppy disk rotates at a fixed speed of approximately 360 rpm (revolutions per minute) within its jacket. Two standard sizes of the floppy disk are currently available. These are the 8 inches square and 5 1/4 inches square floppy disks. Information can either be stored on one side (single-density) or on both sides (double-density).

Information is stored on the floppy disk in concentric circles called tracks. The outermost track is defined with the lowest track number 0 and the innermost track has the highest track number. A read/write head is moved radially across the disk by a stepper motor in increments of the distance between two tracks each time the stepper motor is stepped. In order to access a specific track (seek operation), the head is moved to track 0 and then moved to the desired track by pulsing the stepper motor an appropriate number of times. When the head is over the desired track, it is brought in contact with the floppy disk. Data are then read from or written to the disk. The head is then released from the disk in order to minimize wear on both the disk and the head. The physical beginning of a track is located by means of a small hole (physical index mark) punched through the plastic near the center of the disk. This hole is optically sensed by the disk drive on every revolution of the disk.

Each track is divided into a number of sectors. Sectors are generally 128, 256, 512, or 1024 bytes in length. Track sectoring can be achieved in two ways: hard sectoring and soft sectoring. Hard sectoring divides each track into a maximum of 32 sectors. The start of each sector is indicated by a sector hole punched in the disk plastic. Soft sectoring, the standard for IBM, uses software to select sector sizes. In this method, each data sector is preceded by a unique sector identifier which is read or written by the disk controller.

The floppy disk often contains a write-protect notch punched at the edge of the outer jacket. This notch is sensed by the disk drive and sent to the disk controller as a write protection signal.

9.5.2 THE FLOPPY DISK DRIVE

The floppy disk drive is an electromechanical assembly that reads data from or writes data into the floppy disk. The disk drive includes all the electronics for controlling the head movement in increments toward the center of the disk (forward) or toward the edge of the disk (backward). The disk drive controller connects the head to the floppy data access and disconnects the head from floppy after data access. The drive provides certain information regarding the status of the drive and the floppy. These include status information such as drive ready (indicates that the drive door is closed and the floppy disk is inserted), track zero (indicates that the head is at track zero), and write protect (indicates that the floppy in the drive is write-protected).

9.5.3 THE FLOPPY DISK CONTROLLER

The floppy disk controller normally converts high level disk commands issued by software executed by the main microprocessor into disk drive commands. In order to accomplish this, the floppy controller performs functions such as disk drive selection, track selection, sector selection, head loading, error-checking, and data separation. The disk controller allows the main microprocessor to define a specific drive (in a system with multiple drives) to be used in a particular operation.

The disk controller positions the head from its present location to the desired track from which data are to be read from or written into.
The disk controller senses the requested sector. It also connects the head to the disk surface in order to read or write data.

Information recorded on a floppy disk is subject to both hardware and software errors. Hardware errors are due to media defects. These errors are permanent. Software errors, on the other hand, are temporary and are caused by electromagnetic noise or mechanical interference. Floppy disk controllers use a standard error checking technique known as cyclic redundancy check (CRC). As data are written to a disk, a 16-bit CRC character is computed and stored on the disk. When this data is read, the CRC character permits the controller to detect errors. When CRC errors are detected, the controlling software retries the failed operation several times. If data cannot be reliably read or written, the system software reports the error to the operator.

The actual signal recorded on floppy disk is a combination of timing information (clock) and data. The serial read data input from the disk drive is converted by the controller into two signal streams: clock and data. The controller also assembles the serial data into 8-bit bytes before transferring them to the microcomputer main memory. Note that not all floppy disk controllers such as Intel 82072 provide the data separation feature.

New floppy disks must be written with a fixed format by the floppy controller before these disks may be used to store data. Formatting is a technique of taking raw media and adding the necessary information to allow the controller to read and write data without error. All formatting is performed by the disk controller on a track-by-track basis under the direction of the main microprocessor.

The floppy disk controller functions described above are implemented in a single VLSI chip such as Intel 82072. The functions provided by the 82072 will be covered in the following.

9.5.4 INTEL 82072 FLOPPY DISK CONTROLLER

The 82072 interfaces a microcomputer system based on Intel microprocessors such as 80386, 80286, and 80186 to disk drives compatible with IBM single- and double-density formats. The 82072 is designed using Intel’s CMOS III technology and is housed in a plastic 40-pin DIP or a plastic 44-pin plastic carrier package.

The 82072 includes a 16-byte FIFO to reduce the timing constraints. The point at which the 82072 generates a request for a data transfer is selectable within the 16-byte range of the FIFO.

There are three ports accessible from the microprocessor's point of view: the FIFO, the Main Status Register, and the Data Rate Select Register. Communication between the microprocessor and the 82072 is done by reading the main status register to determine if the 82072 is ready. If it is ready, a command followed by the correct parameters is sent to the 82072 through the FIFO (data port). The main status register can be read by the microprocessor at any time. The microprocessor can also write to the data rate select register at any time.

Command handling in the 82072 can be divided into three phases. These are command, execution, and result.

Upon hardware reset, the 82072 goes into the command phase and is ready to receive a command from the main microprocessor. Typical commands include recalibrate, read track, format track, read data, write data, and seek. During the command phase, the microcomputer writes a predefined set of command code bytes and parameter bytes to the 82072. These bytes of data must occur in the sequence specified by Intel. For example, the recalibrate command causes the read/write head within the floppy disk drive to retract to the track 0 position. Two bytes are provided for the recalibrate command as follows:

D7 D6 D5 D4 D3 D2 D1 D0
0 0 0 0 0 1 1 1 First byte to the 82072
0 0 0 0 0 0 DS1 DS0 Second byte to the 82072

Note that DS1 and DS0 bits select the disk drives as follows:

<table>
<thead>
<tr>
<th>DS1</th>
<th>DS0</th>
<th>Drive selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Drive 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Drive 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Drive 2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Drive 3</td>
</tr>
</tbody>
</table>

Before writing the above data to the 82072, the microcomputer must check certain bits in the main status register. After the command phase, the 82072 automatically enters the next phase as specified by the command definition.

During the execution phases, all data transfers to or from the 82072 occur. Data are transferred in DMA or non-DMA mode as indicated in the SPECIFY command.

If DMA mode is selected, DMA transfers are initiated by the 82072 when it activates its DRQ (DMA request) pin. When ready, the DMA controller responds by activating DACK (DMA acknowledge) pin of the
82072. If the FIFO is enabled by the CONFIGURE command, the DBQ pin of the 82072 will stay active until the FIFO is filled (write) or cleared (read). If FIFO is disabled, each byte to be transferred will request a cycle.

The 82072 informs the microcomputer of the start of the result phase by activating its INT (interrupt) pin. For each of the commands, a definite set of result bytes must be read by the microcomputer before completing the result phase.

These bytes of data must be read out by the microcomputer for another command to start. Among all the Intel microprocessors, the 80186 provides the most of the required functions for interfacing to the 82072. This is because the 80186 contains on-chip DMA controller. Figure 9.30 shows a typical 80186/82072.

9.6 COPROCESSORS

The 82072 data lines are connected to the 80186 ADO-AD7 lines via the 74LS245 buffer. The 80186 on-chip DMA controller does not provide any DMA acknowledge or terminal count signal required by the 82072. This is because the 80186 performs a read or write directly with the device requesting DMA. The DMA acknowledge signal can be obtained by using one of the 80186-generated chip select lines.

The 80186 can generate PCS0-PCS7 chip select lines for the peripheral devices. If the system does not use all the chip select lines for a specific application, one of the spare chip select lines can be used to generate the DTACK signal for the 82072.

The TC line is used to terminate the data transfer. The TC (terminal count) signal can be generated by connecting the DTACK signal to one of the 80186 timers and program the timer to output a pulse after execution of a certain number of DMA cycles.

In Chapters 1 and 7, basics of coprocessors, along with functions provided by Motorola coprocessors such as MC68851 and MC68881, are covered. In this section, a brief overview of the Intel coprocessors will be provided.

Intel offers a number of coprocessors which include numeric coprocessors such as 8087/80187/80287, DMA coprocessors such as 81258, and graphic coprocessors such as 82286. In the following, a brief overview of Intel's numeric coprocessors, which include 8087, 80287, and 80387, will be included.

9.6.1 INTEL 8087

Intel 8087 numeric data coprocessor is designed using HMOS III technology and is packaged in a 40-pin DIP. When an 8087 is present in a microcomputer system, it adds 68 numeric processing instructions and eight 80-bit registers to the microprocessor's register set. The 8087 can be interfaced to Intel microprocessors such as 8086/8088 and 80186/80188.

The 8087 supports seven data types which include 16-, 32-, and 64-bit integers, 32-, and 16-bit floating point, and 18-digit BCD operands. The 8087 is compatible with IEEE floating point format. It includes
Peripheral Interfacing

Chapter 9

9.6 Coprocessors

memory references for numeric data or status information follow the 80286 memory management and protection rules and thus the 80287
extends the 80286 protected mode.

The 80287 receives instructions and data via the data channel control
signaling (PEEQ — Processor Extension Data Channel Operat- er Transfer
Request), PEACK — Processor Extension Data Channel Operat- er
Acknowledgememt; BUSY, NPMD — Numeric Processor RD, NPWR — Numeric
Processor WR). When in protected mode, all information received by the
80287 is validated by the 80286 memory management and protection
unit. The 80287 can operate in parallel with the 80286. When the 80287
detects an exception, it will indicate this to the 80286 by asserting the
ERROR signal.

The 80286/80287 is programmed as a single processor. All memory
addressing modes, physical memory, and virtual memory of the 80286 are
available in the 80287.

9.6.3 INTEL 80387

Intel 80387 is a numeric coprocessor that extends the 80386 architecture
with floating-point, extended integer, and BCD data types. It is
compatible with IEEE floating point. The 80387 includes 32-, 64-, and
80-bit floating point, 32- and 64-bit integers, and 18-digit BCD operands.
It extends the 80386 instruction set to include trigonometric, logarithmic,
exponential, and arithmetic instructions of all data types. The 80387 can
operate in real, protected, or virtual 8086 modes of the 80386. It is
designed using CHMOS III technology and is packaged in a 68-pin PGA
(Pin Grid Array).

The 80387 operates in the same manner whether the 80386 is executing
in real address mode, protected mode, or virtual 86 mode. All memory
access is handled by the 80386; the 80387 operates on instructions and
values passed to it by the 80386. Therefore, the 80387 is independent of
the 80386 mode.

The 80387 includes three functional units that can operate in parallel.
The 80386 can be transferring commands and data to the 80387 bus
control logic for the next instruction while the 80387 floating-point unit
is performing the current numeric instruction. This parallelism improves
system performance.

The 80387 adds to an 80386 system additional data types, registers,
instructions, and interrupts. All communication between the 80386 and
80387 is transparent to application software. Thus, the 80387 greatly
enhances the 80386 capabilities.
QUESTIONS AND PROBLEMS

9.1 Interface a hexadecimal keyboard and seven-segment display to an 8085/8058-based microcomputer.
   i) Draw a hardware schematic of the design. Show only the pertinent signals.
   ii) Write an 8085 assembly language program to display the hexadecimal digit 0-F each time a digit is pressed on the keyboard.

9.2 Repeat Problem 9.1 using 8086/8279 configuration.

9.3 What is the purpose of A6, B6, and IRQ pins on the 8279?

9.4 What are the functions of the 8279 SL0 through SL3 lines? How would you connect an eight-digit hexadecimal display and an 8 x 8 keyboard via these lines? Draw a functional block diagram.

9.5 Describe the basic functions of a DMA Controller. How does it control the I/O R/W and memory R/W signals? Why is the DMA Controller faster than the microprocessor for data transfer?

9.6 Describe briefly the main features of Motorola's MC68440 DMA controller.

9.7 Draw a functional block diagram showing pertinent signals of the MC68020/68230/68440 interface.

9.8 Define the MC68440 modes of operation.

9.9 Which mode and which address lines are required by the MC68440 to decode the register addressed? Why does the MC68440 require more address lines than it requires for register address decoding?

9.10 Draw a functional block diagram of the MC68440/68005 interface.

9.11 What is the difference between the following?

9.12 Assume an LRC7940 printer. Draw a functional block diagram of the LRC7940 printer to an 8085-based microcomputer. Write an 8085 assembly language program to print the hexadecimal digit '0' on the printer.

9.13 Draw a functional block diagram of the 8295 printer controller interface to an 8085-based microcomputer.

9.14 How are the 8295 input data register and output status register accessed? What are the functions of these registers?

9.15 How are the 8295 serial and parallel modes of operation selected?

9.16 In 8295 parallel mode, describe briefly how printers are interfaced via polled, interrupt, or DMA.

9.17 Summarize the basics of CRT. What is the main difference between CRT displays and graphics displays?

9.18 What are the typical functions of a CRT controller? Relate these typical functions to the Intel 8275.

9.19 Draw a functional block diagram showing an 8085-based microcomputer interface to 8275. Show only its pertinent signals.

9.20 What do you mean by bitmapping? How is it applied in graphics?

9.21 Describe briefly the functions provided by the Intel 82786 graphics controller.

9.22 Draw a functional block diagram showing 8088/82786 interface. Show only the pertinent signals.

9.23 What is the basic difference between floppy disk and magnetic tape?

9.24 Summarize the main functions provided by a floppy disk controller.
Chapter 10

DESIGN PROBLEMS

This chapter includes a number of design problems that utilize external hardware. The systems are based on typical microprocessors such as the 8085 and 68000. The concepts presented can be extended to other microprocessors.

10.1 DESIGN PROBLEM NO. 1

10.1.1 PROBLEM STATEMENT
An 8085-based digital voltmeter is designed which will measure a maximum of 5 V DC via an A/D converter and then display the voltage on two BCD displays. The upper display is the integer part (0 to 5 V DC) and the lower display is the fractional part (0.0 to 0.9 V DC).

10.1.2 OBJECTIVE
A digital voltmeter capable of measuring DC voltage up to and including 5 V will be built and tested. The voltmeter is to be implemented using the Intel 8085 microprocessor and an analog-to-digital converter of the designer’s choice. The measured voltage is to be displayed on two seven-segment LEDs.

10.1.3 OPERATION
Figure 10.1 shows a block diagram of the digital voltmeter. It is composed of the microprocessor, 2K bytes of EPROM, 256 bytes of RAM with I/O, the A/D converter, and the display section.
The Intel 8085 microprocessor provides control over all address, data, and control information involved in program execution. It also provides for manipulation of data as taken from A/D and sent to the display section.

The EPROM is a memory unit which stores the instructions necessary for system operation. The RAM and I/O section is a memory unit which provides for data as well as data transfer to and from the A/D and display. The A/D converter takes the voltage measured across Vin(+) and Vin(-) pins and converts it to an 8-bit binary message. The binary information is taken into the microprocessor via I/O and converted to its decimal equivalent. The display section takes the converted binary information from the processor so that it may be read over two seven-segment LED displays. The leftmost display provides the integer portion of the measured voltage, while the rightmost provides the fractional portion.

10.1.4 HARDWARE

Figure 10.2 shows the detailed hardware schematic. The system uses standard I/O and memory map of 0800H-68FFH. The I/O ports of the 8155 are all used. Configured as an input port, port B is connected to the output of the 0804 A/D chip. Configured as an output port, port C is connected to the TTL311 displays. Bits 0-3 are the data outputs while bits 4 and 5 are connected to the latches of the TTL311. Only three bits of port A are used, configured as an output port, to control the select, read, and write lines of the A/D chip.

Using the fully decoded memory addressing, the 74LS318, a decoder, is used to select either the RAM or the EPROM. Also, a 74LS373 is used to latch the address lines to the EPROM. The RAM does not require such a chip because the 8155 RAM has its own internal latches. The ALE line of the 8085 microprocessor controls the latches as seen in the schematic.

The EPROM contains the instructions and the algorithms for converting the binary representation of the analog voltage (applied to the A/D converter) back to decimal representation. The instructions are used to control the system operation. The algorithm uses repeated subtraction to obtain the correct voltage in decimal form. The left display is the integer part and the right display is the decimal part.

The displays, as stated before, are TTL311 hexadecimal displays. In addition, the displays have their own latches which are active low. In the 8085 microprocessor, the interrupt RST 6.5 is used to jump to the address with the algorithm to convert and display the voltage. The INTR pin of the A/D is connected directly to the RST 6.5 pin of the microprocessor. First, an active low is sent to the chip select pin, and then the write pin of the A/D converter is toggled.

Upon completion of the A/D conversion, the 8085 is interrupted. The service routine outputs an active low onto the read pin of the A/D, which latches the data. After inputting the data via the port, the read pin is toggled which then initiates the A/D output.

10.1.5 SOFTWARE

An important part of the software is to convert the A/D's 8-bit binary data into its decimal equivalent for the display. The decimal data will have two digits: one integer part and one fractional part. Two approaches can be used to accomplish this as follows.

Approach 1

Since the maximum decimal value that can be accommodated in 8 bits
Design Problem No. 1

is 255.10 (FF16), the maximum voltage of 5 V will be equivalent to 255.10.

This means that the display in decimal is given by:

\[
D = 5 \times (\text{Input}/255) \\
= \text{Input}/5 \\
= \text{Quotient} + (\text{Remainder}/5)
\]

Integer part

The fractional part in decimal is

\[
F = (\text{Remainder}/5)\times 10 \\
= \text{Remainder}/5
\]

Approach 2

In the second approach, the equivalent of 1 V (255/5 V = 51.10 = 31.10) is subtracted from the input data. If the input data are greater than 1 V, a counter initially cleared to zero is incremented by one. This process continues until the measured data are less than 1 V. The register keeps count of how many subtractions take place with a remainder greater than 1 V and thus contains the integer portion of the measured voltage in decimal.

The decimal portion of the fractional part is obtained in the same way except that if the input data are less than 1 V, then they are compared with the decimal equivalent (51/10 = 5) of 0.1 V. If the measured data are greater than 0.1 V, a counter initially cleared to zero is incremented by one and the process continues until the input data are less than 0.1 V. The counter contains the fractional part of the display.

Approach 2 is used as a solution to this problem. A complete listing of the 8085 assembly language program to control the digital voltmeter is given below. The program is used to begin and end the A/D conversion process as well as to manipulate the binary data into their decimal form so that they can be displayed in an easily readable format.

FILE: LST1.BAT001

HEWLETT-PACKARD: 8085 Assembler

LOCATION OBJECT

CODE LINE SOURCE LINE

\[
<0000> \\
<0001> \\
<0002> \\
<0003> \\
<0004> \\
<0005> \\
<0006> \\
<0007> \\
<0008> \\
<0009> \\
<000A> \\
<000B> \\
<000C> \\
<000D> \\
<000E> \\
<000F> \\
<0010> \\
<0011> \\
<0012> \\
<0013> \\
<0014> \\
<0015> \\
<0016> \\
<0017> \\
<0018> \\
<0019> \\
<001A> \\
<001B> \\
<001C> \\
<001D> \\
<001E> \\
<001F> \\
<0020> \\
<0021> \\
<0022> \\
<0023> \\
<0024> \\
<0025> \\
<0026> \\
<0027> \\
<0028> \\
<0029> \\
<002A> \\
<002B> \\
<002C> \\
<002D> \\
<002E> \\
<002F> \\
<0030> \\
<0031> \\
<0032> \\
<0033> \\
<0034> \\
<0035> \\
<0036> \\
<0037> \\
<0038> \\
<0039> \\
<003A> \\
<003B> \\
<003C> \\
<003D> \\
<003E> \\
<003F> \\
\]

2 0045

SOURCE LINE

<0000> \\
<0001> \\
<0002> \\
<0003> \\
<0004> \\
<0005> \\
<0006> \\
<0007> \\
<0008> \\
<0009> \\
<000A> \\
<000B> \\
<000C> \\
<000D> \\
<000E> \\
<000F> \\
<0010> \\
<0011> \\
<0012> \\
<0013> \\
<0014> \\
<0015> \\
<0016> \\
<0017> \\
<0018> \\
<0019> \\
<001A> \\
<001B> \\
<001C> \\
<001D> \\
<001E> \\
<001F> \\
<0020> \\
<0021> \\
<0022> \\
<0023> \\
<0024> \\
<0025> \\
<0026> \\
<0027> \\
<0028> \\
<0029> \\
<002A> \\
<002B> \\
<002C> \\
<002D> \\
<002E> \\
<002F> \\
\]
### Design Problem No. 1

If we are to return to a current program after an interrupt has been serviced, although this program is not necessarily under control, it is a good habit to initialize it by setting the stack pointer to the correct value. This may be a convenient time to set the stack pointer to the beginning of the memory used for the program.

Lines 12–14 set the stack pointer and enable interrupt RS76.5. Line 15 defines port A as output, port B as input, and port C as output. Note that the data to configure ports were already in the accumulator as per line 12.

Lines 16–17 send an active high to each display's data latch enable pin. This insures that the display will output the correct data on the next high-to-low transition at the latch enable pins.

Lines 18–19 send an active high to the chip's select (CS), write (WR), and read (RD) pins on the A/D converter. This insures proper startup of the converter.

Lines 20–21 send an active low to the converter's CS pin. Next, lines 22–25 toggle the WR, pin that conversion starts. The combination of CS and WR active low resets A/D internally and sets its up for the start of the conversion. By setting WR in active high, the conversion starts. Figure 10.3 shows the timing diagram for the A/D.

Line 26 is a halt which is placed as a delay to wait for the interrupt request. This is necessary since it may take from 1 to 276 us for the interrupt to be asserted. This equivalent to approximately 380 clock cycles for the 8085 operating at 1MHz.

Line 29 continues the program at the interrupt vector for interrupt RS76.5.

Line 30 loads the H1 register with a memory address to be used later in the program.

Lines 31–32 initialize the E port. E register is to hold the integer portion of the measured voltage, while C register holds a hexadecimal equivalent to 1 V for this system.

Lines 33–34 send an active low to the RD pin of the A/D converter so that the binary information corresponding to the measured voltage may be read by the microprocessor.

Lines 35–37 take the data in the A/D converter and store them into register B.

Lines 38–39 toggle the RDs back to active high.

Lines 40–41 move the 8-bit input to memory location 09000H and then into the accumulator.

Lines 42–46 convert the binary data into its decimal equivalent so that the integer portion may be removed. First the equivalent of 1 V is subtracted from the input data. If the measured voltage is less than 1 V, the program jumps to line 47. If the range is greater than one, the program continues.
10.2 Design Problem No. 2

continues at line 44 where register D is incremented by one. The remainder from the subtraction is temporarily stored in memory. The program then unconditionally jumps back to line 42 so that another subtraction takes place. This loop occurs until the remainder from the subtraction is less than 1 V. Register D keeps count of how many subtractions took place with a remainder greater than 1 V and thus counts the integer number of volts measured.

Lines 47—50 send the contents of register D to the leftmost display. The AND operation unblocks the data at the display.
Lines 51—52 again initialize registers D and C, but this time register D will be counting the fractional portion of the measured voltage and register C will hold the hex equivalent of 0.1 V.
Line 53 moves the last positive remainder from memory into the accumulator.
Lines 54—57 perform the same function as lines 42—46 but with the fractional portion of the measured data.
The remaining lines output the contents of register D into the rightmost display and then halt the program.

10.2 DESIGN PROBLEM NO. 2

10.2.1 PROBLEM STATEMENT

A 68000-based system is designed to drive three seven-segment displays and monitor three key switches. The system starts by displaying 600. If the increment key is pressed, it will increment the display by one. Similarly, if the decrement key is pressed, it will decrement the display by one. The display goes from 00-FF in the hex mode and from 000-255 in the BCD mode. The system will count correctly in either mode. The change mode key will cause the display to change from hex to decimal or vice versa, depending on its present mode. Figure 10.4 depicts the block diagram.

Two solutions are provided for this problem. Solution one uses programmed I/O and no interrupts, while solution two utilizes interrupt I/O but no programmed I/O.

10.2.2 SOLUTION NO. 1

The simplest and the most straightforward system possible is built to obtain the required results. This means that there will be no RAM in the
system; therefore, no subroutine will be used in the software and only programmed I/O (no interrupt) is used.

10.2.2.2a Hardware

Figure 10.5 shows the detailed hardware schematic. The circuit is divided into the following sections.

10.2.2.2a.i Reset Circuit

The reset circuit for the system is basically the same as the one used for the 8085. The circuit has a 0.1-µF capacitor and 1K resistor to provide an RC time constant of 10^4 s for power on reset. The RESET and HALT pins of the 68000 and the RESET pin of the 6821 are tied together for complete and total reset of the system.

10.2.2.2a.ii Clock Signal

An external pulse-generator is used to generate the clock signal for the system. The system is driven up to 3 MHz, the limit of the generator, without any problems.

10.2.2.2a.iii Buffering

Because the 68000 is interfaced to other devices (6821 and 2716s), the outputs of the 68000 which are used to drive these chips must be buffered in order to be certain that there is enough drive current. The buffering is done by a TTL open-collector inverter buffer chip (7406). Since this chip is of the open-collector type, 1K pull-up resistor is also needed.
10.2.2.1 Address Mapping

The system has two 2K EPROM (2716s) and one 68000's peripheral I/O chip (6821). The 68000 address lines A11 through A13 are needed to address the EPROM. So A12 is used to select between the 2716s and the 6821 (0 for 2716s and 1 for 6821). Memory access for the EPROM is asynchronous, while the 6821 is synchronized with the E-clock. A12 is inverted, through the buffer, so the output of the inverter goes to CS2 of the 6821 and also to VFA of the 68000 for synchronization. The 68000 VMA pin is buffered and inverted and it goes to CS0 of the 6821. The 6821 is chosen to be odd, so CS1 is activated by inverted EOS line. Finally, address lines A1 and A2 are connected to RS0 and RS1, respectively.

The address lines to the 2716s come from the NAND gates. The results of the inverted A12 NANDed with the inverted EOS or the inverted ULSD, depending on whether the EPROM is odd or even. The DTACK pin of the 68000 and the OE pins of the 2716s are activated by the signal of R/W inverted. When the 68000 wants to read the EPROM this signal will be high, so its inverted signal will provide a low to DTACK. This does not cause any problem because when the 68000 accesses the 6821, VFA is activated and the 68000 will not look for DTACK.

The configuration above causes the memory map to be as follows:

| A13...A0 | 16 | A12...A1 | 16 | A11 | 16 | A10 | 16 | A9 | 16 | A8 | 16 | A7 | 16 | A6 | 16 | A5 | 16 | A4 | 16 | A3 | 16 |
|-----------------|---|-----------|---|-----|---|-----|---|-----|---|-----|---|-----|---|-----|---|-----|---|-----|---|
| 4K of EPROM Memory | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| PA/DDRA | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| CRA | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| TB/DDRB | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| CRB | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

10.2.2.a I/O

There are 3 seven-segments displays in the system (T1311), an LED, and 3 switches. The 3 displays have internal latches and hex decoders. So the two least significant displays are connected directly to port B of the 6821 chip, and the most significant line is connected to the upper 4 bits of port A. The latch will enable, for the displays are tied to ground so as to enable them at all times. The LED, when ON, will indicate that the display is in the BCD mode. Each of the three switches, double-pole single throw type, with LED indicator, goes through two inverters (7404) for the hardware debounce, and the outputs of the inverters are connected to the lowest 3 bits of port A.

10.2.2.a.ii Unused-Pins Connection

For the 68000 there are 6 unused, active low, input pins which must be disabled by connecting them to 5 V. These are TPLB, TP1, TP2, TPLL, PERK, BR, and BGACK. Two of the 6821 unused pins (TRQA and TRQB) are also disabled this way, while CAI and CBI are disabled by connecting them to ground.

10.2.2.b Microcomputer Development System

Hewlett-Packard (HP) 64000 is used to design, develop, debug, and emulate the 68000-based system. Some details are given in this section.

The emulator is a very important part of the development of the software and hardware of the system. The 68000 emulator has most of the functions for emulation such as display memory or registers, modify memory or registers. But there is no single step function. The HP 64000 emulator is divided into three modes of operations: initialization, emulation, and EPROM programming.

10.2.2.b.i Initialization

Edit

The edit function is used to create the application program in mnemonic form. The first line of the program must be "68000!" to indicate that the program is to be assembled by the 68000 assembler, and that a 68000 microprocessor is to be used for emulation. Also, the special assembler mnemonic PROG should be placed before the first instruction for linking the program later.

Assemble

When the application program has been completed and properly edited, the file is then assembled into a relocatable object code file. All errors indicated by the assembler should be corrected at this point.

In order to use the 68000 emulator special functions, a special monitor program is required. This can be copied as follows:

COPY Mon_68K:HP:source to MON_68K

Note upper and lower case. Once this monitor is copied, it must be assembled for no errors.
Design Problems

Chapter 10

10.2 Design Problem No. 2

Note: Usually external clock requires external DTACK; however, since the system only has EPROMs and for the purpose of emulation these EPROMs are not used, the external DTACK is not required.

Once the required files and memory maps are loaded, the system is ready for emulation. The monitor program must be running before the application program is executed. To run the monitor program, the following is used:

```
run from 0100H <cr>
```

Then the application program is run using:

```
run from 1100H <cr>
```

Another important part that the user should keep in mind is the processor status. There are three messages for the processor status which indicate that the emulator is not generating any bus cycles. They are:

1. Reset — Indicates that the user's hardware is asserting the Reset input. The condition can only be terminated by releasing the user's hardware.
2. Wait — Indicates that the 68000 is waiting for a DTACK or other memory response. The condition can be terminated by asserting DTACK, BERR, VPA or entering "reset" from the keyboard.
3. No memory cycle — Indicates that the 68000 has executed a STOP instruction. The condition can be terminated by asserting "break" or "reset" from the keyboard.

10.2.2.kii EPROM Programming

After the software and hardware have been emulated and they work properly, the final step is to program the EPROM and put the final circuit together. But before this, the program must be changed to include the addresses for the stack pointer and the initial PC. This is done by using the "ORC" and "DC" assembler directives. Then this new program is assembled and linked again. The EPROM is then programmed with the contents of this final "absolute file."

Programming EPROMs with the HP 64000 for 68000 is done by odd and even EPROMs. To program the lower 8 bits of data (odd ROM), the option bit 0 is selected, and bit 8 for the upper 8 bits (even ROM) is chosen as follows:

- **Linking**

  The two relocatable files must be linked together to create an absolute file for the emulation process. The files can be linked as follows:

  ```
  Link <cr>
  Object files? Mon_68k
  Library file? <cr>
  Prog, Data, Comm = 000100H, 000000H, 000000H
  More files? yes
  Object file? (name of application program file)
  Library file? <cr>
  Prog, Data, Comm = 001100H, 000000H, 000000H
  More files? no
  Absolute file? (name of absolute file)
  ```

  There are reasons why the two files were linked this way. The monitor program must be stored at 0100H through 0FFFH, and since this I/O port is mapped starting at 1001H, the application program must be stored starting at a different address, so 1100H was used. Also, address 000H through 0FFH is used for exception vectors by the 68000 microprocessor.

- **10.2.2.kii Emulation**

  The emulator was used as a replacement for the actual 68000 chip to test the software logic and hardware before it was actually installed into the circuit.

  To start the emulation process, the following soft-key parameters were entered:

  - **RUNLOAD**
    - Processor clock? external
    - Restrict to real time? no
    - Memory block size? 256
    - Significant bits? 20
    - Break on write to ROM? yes
    - Memory map: 0000H thru 0FFFH emulation RAM (monitor & exception vector)
    - 1000H thru 10FFH user RAM (I/O PORT addresses)
    - 1100H thru 11FFH emulation ROM (application program)
    - Modify simulated I/O? no
    - Reconfigure pod? no
    - Command file name? (name of emulation command file)
10.2.2.2 Software

The program consists of three major functions: initialize I/O ports and data registers, monitor and debounce key switches, and increment, decrement, or change mode. The program configures port B of the 6821 as an output port which will be used to display the two lower significant nibbles of data. Higher 4 bits of port A are configured as output to display the most significant nibble of the data. Bit 3 is also an output bit which turns ON and OFF the mode of the LEDs. The lowest 3 bits of port A are configured as inputs to detect the positions of three key switches. Register D3 is used to store the data in hex. Registers D4 and D7 are used to store the data in BCD mode with the low order byte in D4 and the high order byte in D7. Bit 3 of D0 contains a logic 1 representing BCD mode and logic 0 representing hex mode. Register D5 contains a 1 which will be used for incrementing BCD data, since ABCD doesn’t have immediate mode. Register D6 contains 999 which is used for decrementing BCD.

The program monitors the three switches and stores the three input bits into register D0 if any of the keys is pressed. The processor then waits until the depressed key is released and then checks the input data one by one. The processor then increments, decrement, or mode change routine according to the depressed key. After execution, the processor will display the result on the seven-segment displays.

Figure 10.6 shows the software flowchart.

The assembly language program is listed below:

```
*LDRXU, 16 5000 8000 8000 Assembler

1 "8000"
2 ****************************
3 *THIS PROGRAM STARTS DISPLAYING 000 AND MONITORS THREE KEYS *
4 *SWITCHES TIGHT DECREMENT, DECREMENT, OR CONVERT HEX TO BCD *
5 *OR VICE VERSA, DEPENDING ON WHICH KEY IS DEPRESSED, THE *
6 *DISPLAY GOES FROM 00-FF IN HEX MODE OR 00-255 IN BCD MODE *
7 ****************************

FILE: LAB2.RIGA22

 Hewlett-Packard: 68000 Assembler

LOCATION OBJECT
CODE LINE

1024: 8 PA  EQU 00101010
1025: 10 EORA EQU 00101011
1026: 11 PS EQU 00101100
1027: 12 DBH EQU 00101101
1028: 13 DBL EQU 00101110
1029: 14 ORC EQU 00101111
006000 FFFF FFFF
006004 0000 0000
006004 "START"
18 "CONVERT THE INPUT AND OUTPUT PORTS,"
19 "DISPLAY 000 ON THE 7-SEGMENT DISPLAYS,"
20 "AND INITIALIZE ALL THE DATA REGISTERS,"
21 "THE HEX MODE IS STORED IN D7 AND THE"
22 "BCD MODE IS STORED IN D7 AND D4."
000000 4238 1003
000000 11FC 00FF
23 "START"
24 "CLR.B"
25 "CLR.B"
26 "CLR.B"
27 "CLR.B"
28 "CLR.B"
29 "CLR.B"
30 "CLR.B"
31 "CLR.B"
32 "CLR.B"
33 "CLR.B"
34 "CLR.B"
35 "CLR.B"
36 "CLR.B"
37 "CLR.B"
38 "DISPLAY THE KEY SWITCHES"
39 "SCAN"
40 "MOVE.B 00, PA"
41 "SCAN"
```

423.jpg
Design Problems

10.2 Design Problem No. 2

FILE: LAB2.K80A22  HEWLETT-PACKARD: 68000 Assembler
(continued)

LOCATION OBJECT

CODE LINE

SOURCE LINE

000046 67F6 42

REQ SCAN

: IF NO KEY IS DEPRESSED GO TO SCAN

000048 1438 1001 43

MOVE.B PA, D2

: READ THE DATA AGAIN

00004C 0202 0007 44

ANDI.B $078, D2

: CHECK TO SEE IF THE DATA REMAIN UNCHANGED

000050 8401 45

CMP.B D1, D2

: IF IT CHANGES GO TO SCAN

000052 66EA 46

BNE SCAN

: CHECK TO MAKE SURE THAT THE KEY IS

000054 1438 1001 51

KEEPL MOVE.B PA, D2

: RELEASED BEFORE THE NEXT KEY CAN BE

000058 0202 0007 52

ANDI.B $078, D2

: KEPT KEY

00005C 66F6 53

BNE KEYP

: CHECK TO SEE WHICH KEY HAS BEEN KE-EN-

000060 C801 0000 58

BSET.B $000, D1

: TERNED. BITS 0, 1, AND 2 OF D1 REPRESENT

000062 6600 003C 59

BNE INCK

: INCREMENT BOTH BJE AND BCD AT THE SAME

000064 C801 0001 60

BSET.B #1, D1

: TIME.

000066 8700 0044 61

BRQ MODE

: IF BIT-1 IS 1

000068 C003 0000 65

CMP.T.B $008, D3

: DECREMENT OTHERWISE GOTO MODE

000072 67CA 66

BRQ SCAN

: IF THE NUMBER IS

000074 5903 67

SUBQ.B #1, E3

: GOTO SCAN

000076 C603 68

AND.B D3, D3

: DECREMENT BJE BY 1

000078 C906 69

ABCD.B D6, D4

: CLEAN THE CARRY

00007A C606 70

ABCD.B D6, D7

: DECREMENT BCD BY 1 BY ADDING IT

00007C 0600 0033

DISPLAY BSET.B $3H, D0

000080 6704 0014

BRQ HEX

: IF BIT-3 OF D0 IS 0, GOTO BHE

000084 11C1 1095

MOVE.B D4, PB

: OUTPUT THE LSB TO PORT B

000088 194F 70

LSEL.W #1, D7

: SHIFT LEFT 4 TIMES

00008A 80C7 9013

BSET.B $3H, D7

: TURN OFF THE LED

00008E 11C7 10F1 86

MOVE.B D7, PA

: OUTPUT THE MSB TO OUTPUT 8 BITS OF PORT A

000092 284F 81

LSEL.W #6, D7

: BJA SCAN

000094 60AB 82

BRA SCAN

: OUTPUT TO PORT A

000096 11C0 1001 83

HEX MOVE.B D0, PA

: NUMBER TO PORT B

000098 11C3 1005 84

MOVE.B D3, PB

: OUTPUT THE HEX

00009E 609E 85

BRA SCAN

: INCREMENT BOTH BJE AND BCD.

0000A0 3C03 00FF

INCR CMP.T.B $008, D3

: INCREMENT HEX NUMBER BY 1

0000A4 6798 90

BRQ Scan

: IF THE NUMBER IS

0000A6 5203 92

ADDQ.B $1, D3

: DECREMENT NUMBER OF

0000A8 4202 92

CLR.B $2

: INCREMENT MSB OF BCD BY 1

0000AA C205 93

ABCD.B $1, D4

: INCREMENT LSB OF

0000AC CF02 94

ABCD $2, D7

: BCD BY 1 IF CARRY

0000AE 60CC 95

BRA DISPLAY

: IS 1

0000B0 0540 0003

BRA INCR $18, D0

: EXCHANGE MODE BY

0000B4 60CE 100

BRA DISPLAY

Errors = 0
10.2.3 SOLUTION NO. 2

The second solution approach uses interrupt I/O but no I/O ports.

### 10.2.3.a Hardware

The system includes a 3-digit and three momentary function switches (increment, decrement, and mode select). In order to minimize the complexity of the project, no I/O chips are used. Instead, a buffer and some latches at the I/O ports are used. The buffer is used to hold the status of the momentary switches and the latches are used to hold the information coming from the data bus. To further the design, three TIL311 displays are used because they contain internal data latches. Because the 68000 has 23 address lines (not including A0), the memory is linearly decoded. The even and odd memory chips are enabled by decoding pins UDS, LDS, and X8.

To display the three-digit number, the data lines are connected to the inputs of the three TIL311 displays (D0-D3 - LSD, D4-D7 - middle digit, D8-D11 - MSD). The address strobe (X5) is NANDed with the address line A14 to latch the data onto the three displays. The memory map for the displays is given in Table 10.1. Because of linear decoding, the problem of foldback exists.
Design Problems

Chapter 10

10.2 Design Problem No. 2

Two 2716s are used for the EPROM and two 6116s are used for the RAM. Both the RAM and the EPROM chips are divided into even and odd memory. The configuration enables the 68000 to access an even or an odd data byte or a complete word in one bus cycle. The even and odd select lines are generated by ANDing the UDS and AS pins and the LDS and AS pins, respectively. To access a word, both the even and the odd enable signals are asserted. These signals are then NANDed with address lines A12 and A13 to select the EPROM and the RAM, respectively (see Figure 10.7). The odd memory chip data lines are connected to D0-D7 of the 68000. The even memory chip data lines are connected to D8-D15. Table 10.1 shows the memory map.

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$000000-$0000FF</td>
<td>EPROM</td>
</tr>
<tr>
<td>$000300-$0003FF</td>
<td>RAM</td>
</tr>
<tr>
<td>$005000-$0055FF</td>
<td>DISPLAYS</td>
</tr>
<tr>
<td>$009000-$0095FF</td>
<td>SWITCHES</td>
</tr>
</tbody>
</table>

In the system, the interrupt pins are implemented by ANDing the status of the momentary switches and connecting the output of the gate to IPF3. To achieve a level 6 interrupt, IPF1 and IPF0 pins are connected to Vcc and ground, respectively (see Figure 10.7). To reduce the number of components, the 68000 is instructed to generate an internal autovector to service the interrupt. This is accomplished by asserting VPA and IPF2 at the same time. If an interrupt occurs (switch pressed), the 68000 will compute the autovector number $1E and the vector address $78. The processor will then go to a service routine that will find the switch that was pressed.

A 4-MHz crystal oscillator is used to clock the processor. Since the 68000 is operating at 4 MHz, A5 is directly connected to DTACK. This gave the EPROMs (459 ns access time) about 500 ns to provide valid data. A reset circuit similar to the one used in the 8085 system is used for the 68000-based microcomputer. However, on the 68000, both the RESET and HALT pins are tied together (see Figure 10.7). Figure 10.8 shows the board layout of all the chips.

10.2.3.6 Software

The first major feature of the software is the inclusion of a start-up routine. The advantage of the start-up routine is to visibly verify the system.

FIGURE 10.7 68000-based system for design problem no. 2.
three displays count F down to 0 (in parallel). This routine uses a DBF loop in which the counter's value is duplicated to the two higher hex digits. The following is the actual start-up routine implemented in the program:

```
MOVEQ #7FF, D0    ; INITIALIZE LOOP COUNTER
                ; TO $00000000F
LOOP          ; COPY D0 TO D1
    MOVE.W D0, D1
    ASL.W #4, D1   ; SHIFT DI LEFT 4 TIMES
    ADD.W D0, D1   ; ADD D0 TO D1
    ASL.W #4, D1   ; SHIFT DI LEFT 4 TIMES
```

ADD.W D0, D1   ; ADD D0 TO D1
MOV.W D1, DISPADDR ; SEND RESULT TO DISPLAY
MOV.L #VISIBLE, D6 ; LOAD DELAY TIME
JSR DELAY ; CALL DELAY SUBROUTINE
DBF.W D0, LOOP ; DEC BRANCH IF D0 != -1,
                ; NOT TO THE LOOP
CLR.L D0 ; INITIALIZE COUNTER TO ZERO
CLR.L D7 ; INITIALIZE MODE TO DECIMAL
MOV.W D0, DISPADDR ; INITIALIZE DISPLAYS TO
```
Design Problem No. 2

A service routine exists at location $8010$. In response to the interrupt, the software directs the 68000 to move in the status switches to the low word of D0. A "C"-type priority case statement executes.

The case statement has the priority of up, down, then mode. Implementation of the case statement eliminates uncertainties when multiple keys are depressed. In the following, the case statement is shown.

```
RESPONSE NOP
MOVE.W STATUS, D1 ;MOVE IN BUTTON STATUS WORD
BTST.W #UPBIT, D1 ;TEST INCREMENT BIT
BEQ.W INCREASE ;IF UPBIT=0 BRANCH TO INCREASE
BTST.W #DOWNBIT, D1 ;TEST DECREMENT BIT
BEQ.W DECREASE ;IF DOWNBIT=0 BRANCH TO DECREASE
BTST.W #MODEBIT, D1 ;TEST MODEBIT
BEQ.W CHANGE MODE ;IF MODEBIT=0 BRANCH TO CHANGE MODE
BRA.B RESPONSE ;NO RESPONSE, THEN SEARCH AGAIN
```

This segment utilizes the test bit facilities of the 68000. The algorithm first loads the status switches. The status word is then tested by the BTST instruction. The first bit test is the upbit. If the bit is found to be 0, the program will branch to an increase-update routine. If the downbit is found to be low, then the program reacts to decrement the displays. If the mode bit is found low, then the response is the base conversion of the displayed output.

The user may be tempted to indefinitely press a button or press multiple buttons. The habit is permissible. The program implements a 0.4-second wait loop at the end of any press of a key. This is a post-debounce. Without this feature, the 68000 will either count or change modes at speeds beyond recognition. The debounce routine also contains a priority. If the user constantly depresses multiple keys, the 68000 will service the input with the highest priority.

At this point, a deviation of the problem was made. The deviation was the case of checking out the project. During checkout, when one wants to see a rollover, the increment or decrement key must be pressed 255 times. This is futile. At the end of the service routine, the software will not lock out a key entry, but rather the 68000 will immediately go to the wait state.
where the next interrupt may take place. To the user, it will appear that the 68000 is either autodecrementing, autodecrementing, or automatically changing modes. The post-debounce segment is displayed below.

```
VIEBKR NOP ;ENTRY NO OPERATION
MOVE.1 #VISIBLE.D6 ;PLACE DELAY INTO D6
JSR DELAY ;JUMP TO DELAY SUBROUTINE
RTS ;RETURN FROM SUBROUTINE

DELAY NOP ;ENTRY NO OPERATION
DEF.W D6,DELAY ;DECREMENT FOR WAIT
RTS ;RETURN FROM SUBROUTINE
```

The debounce routine implements a dummy loop that utilizes a large loop count. The routine is initialized by an immediate move long to D6. The debounce routine is called via the jump subroutine command. The delay loop contains a no-operation to increase loop time. After the NOP, the DEF will decrement D6 and branch if D6 is not equal to negative 1.

The software uses a hex base for counting, that is, all numbers whether decimal or hex will originate from a hex byte in data register 0 (D0). The display status exists in data register 7 (D7). If the contents of D7 are zero, this informs the program to display a decimal number on the next update. Otherwise, the program will send a hex value to the display. A typical decision-making segment (below) uses the 68000's ability to update flags on a move operation.

```
MOVE. D7,D7 ;MOVE TO UPDATE FLAGS
JNB.B HEX ;IF Z=0 THEN SEND HEX TO DISPLAYS
BRA.B DECI \AL ;OTHERWISE, DECIMAL TO DISPLAYS
```

To convert a hex number to decimal format, the program uses the division/modulo algorithm shown in the following.

```
DECIMAL NOP ;ENTRY NO OPERATION
CLR.L D2 ;INITIALIZE D2 TO ZERO
CLR.L D1 ;INITIALIZE D1 TO ZERO
MOVE.B D0,D1 ;COPY COUNT
DIVU $10.D1 ;DIVIDE D1 BY 10 MSD HEX -> DECIMAL
SWAP D1 ;PLACE REMAINDER IN LOW WORD D1
```

The algorithm exploits the DIVU (unsigned division) facilities of the 68000. The hex byte is moved to a long word register with zero-extend (assumed by CLR.L, followed by a MOVE.B operation). The number is then divided by 10. The quotient remains in the low word of the destination register (D1); the remainder lies in the high word. With the use of SWAP, the remainder and quotient words are swapped. The remainder is moved (MOVE.W D1,D2) to another register D2 (initialized to zero). At this point, the remainder is cleared in D1, and swap is used to replace the quotient in the low word of D1. The next lower significant digit is extracted. Again, DIVU uses an immediate source of 10. The remainder in D1 is swapped into the low word, shifted four times, then added to D2. The quotient is swapped back to the low word, shifted left eight times, then added to D2. The result of this routine is (at most) a three-digit BCD number which is suitable to send to the displays.

After the update of the displays, a time delay subroutine allows execution delays from the upper to microseconds to the order of seconds. The time delay subroutine is shown below.

```
DELAY NOP ;ENTRY NO OPERATION
DEF.W D6,DELAY ;DECREMENT FOR WAIT
RTS ;RETURN FROM SUBROUTINE
```
The NOP serves to increase the delay time of the loop. The NOP takes 4 clock cycles. DBF, W (decrement and branch on false) takes 10 clocks on a branch and 14 clocks on a skip. JSR (jump subroutine) to the delay takes 23 cycles. The time analysis is simplified when consideration is taken into the duration of the delay. A suitable delay for this project is about 0.4 s. This equates to (4-MHz clock) 1.6 million clock cycles. Because of the high number of cycles required, the "calls" and "returns" can be avoided because of their insignificance when compared to the massive number of clock delays required. A delay of about 0.4 s is used, which requires about 100,000 loops in the delay routine.

Some mention should be made of the mode features of the software. The change mode allows the user to liberally change the viewing format from hex to BCD or vice versa. The activation of this software feature simply complements D7 and then updates the displays via the previously mentioned methods. The increment facility increments D0 and updates the displays. Similarly, the decrement facility decrements D0 and updates the displays.

Expansion of the system is possible. Maybe for user entertainment, an upown or downown can be implemented. The tone can be generated through a variable delay routine. One of the address lines may be tied in series to a small speaker and every time the address is accessed, the speaker will "tick". Otherwise, the software, as it is, is suitable for the project. A listing of the assembly language program is provided below:

```
"6000"
/
Microrcomputer Applications
M. Rajagopalan
November 9, 1987

This is the software routine for Design Problem
No. 2
THE BCD<->HEX COUNTER
Language is 68000 MACRO

NAME BCD<->HEX_COUNTER ;SERVIE ROUTINE ADDRESS
LOCATIONS
AUTO6 EQU 0000074H ;STACK INITIALIZATION
TSTACK EQU 000077CH ;STACK INITIALIZED
RESPONSE EQU 00009500H ;Interrupt vector
ADDRESS
```

```
10.2 Design Problem No. 2

DISPADER EQU 00000000H ;ADDRESS OF DISPLAY
PCINIT EQU 00004000H ;PC STARTUP ADDRESS
VISIBLE EQU 00009000H ;DELAY TIME APPROX. 0.4
SECOND
UPBIT EQU 0FH ;INCREMENT BIT LOCATION
DOWNBIT EQU 0FH ;DECREMENT BIT LOCATION
MOBEXIT EQU 0FH ;MODE BIT LOCATION
STATUS EQU 00008000H ;STATUS WORD LOCATION
INSTMARK EQU 2500H ;INTERRUPT MASK, LEVEL 7

;Top of the stack, program origin, and interrupt service
;location

ORG 00006000H ;STARTUP
DC L TSTACK ;INITIAL SUPERVISOR
DC L PCINIT ;FIRST PROGRAM INSTR LOC
ORG AUTO6 ;LOCATION OF AUTOVECTOR
DC L RESPONSE ;ADDRESS OF SERVICE
DC L RESPONSE ;ROUTINE 5
DC L RESPONSE ;ADDRESS OF SERVICE
DC L RESPONSE ;ROUTINE 6
DC L RESPONSE ;ADDRESS OF SERVICE
DC L RESPONSE ;ROUTINE MNI

; Startup routine

ORG PCINIT ;BOOTUP AND TEST ROUTINE
NEXEQ $0FH, DO ;INITIALIZE LOOP
COUNT TO 00000000F

LOOP MOVE W DO, D1 ;COPY DO TO D1
ADD W DO, D1 ;SHIFT D1 LEFT FOUR TIMES
ADD W DO, D1 ;ADD D0 TO D1
ADD W D1, D1 ;ADD D0 TO D1
MOVE W D1, DISPADOR ;SEND RESULT TO DISPLAY
MOVE L 4VISIBLE, D6 ;LOAD DELAY TIME
JSR DELAY ;CALL DELAY SUBROUTINE
DBF D0, LOOP ;DEC BRANCH IF D0 = 0, NOT TO LOOP

CLR L D0 ;INITIALIZE COUNTER TO ZERO
CLR L D7 ;INITIALIZE MODE TO ZERO
MOVE W D0, DISPADOR ;INITIALIZE DISPLAYS TO ZERO
MOV W #INSTMARK, SR ;SET INT AT 5 AND SUPER
```
CHAP 10

Design Problem No. 2

This routine sends hex contents of D0 to the displays.

```
ENTRY NO OPERATION
MOVE.W D0, DISPADDR
VIEWER
ENTRY NO OPERATION
ENTRY DATA IS SENT TO
DISPLAYS
GO TO DELAY BRANCH

Hex -> Decimal converter

ENTRY NO OPERATION
CLR.L D1
INITIALIZE D1 TO ZERO
ENTRY NO OPERATION
COPY COUNT
ENTRY NO OPERATION
DIVIDE D1 BY 10
ENTRY NO OPERATION
HEX -> DECIMAL
ENTRY NO OPERATION
PLACE REMAINDER IN LOW
ENTRY NO OPERATION
WORD D1
ENTRY NO OPERATION
MOVE REMAINDER TO D2
ENTRY NO OPERATION
CLEAR REMAINDER
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPERATION
ENTRY NO OPER
Design Problems

Chapter 10

10.2 Design Problem No. 2

Figure 10.9 shows the flowchart for the start-up routine.

- Initialization of startup PC and SSP based
- Initialize for startup routine
- DS Int count
- Copy counter to lower three MSB of D1
- Send D1 to displays
- NO
- Decrease DS = -1?
- YES
- Stamp Completed
- Wait for interrupt
- 84 brk 84

For 0.45 s delay with a 4-MHz clock, 1.8 million clock cycles are required (0.45 s was chosen for ease of calculations).

Therefore,

\[ 18n = 1,800,000 \]
\[ n = 100,000 \]
QUESTIONS AND PROBLEMS

Design and develop the software and hardware for the following using a particular microprocessor and its support chips with a microcomputer development system of your choice.

10.1 i) Design and develop the hardware and software for a microprocessor-based system that would measure, compute, and display the Root-Mean-Square (RMS) value of a sinusoidal voltage. The system is required to:
   1. Sample a 60-Hz sinusoidal voltage 128 times.
   2. Digitize the sampled value through a microprocessor-controlled analog-to-digital converter.
   3. Input the digitized value to the microprocessor using interrupt.
   4. Compute the RMS value of the waveform using the equation \( \text{RMS value} = 1.11 \times \text{average value} \).
   5. Display the RMS value using two digits.

ii) Repeat Problem i) using the algorithm

\[
\text{RMS value} = \sqrt{\frac{\sum x_i^2}{N}}
\]

where \( x_i \)'s are the samples and \( N \) is the total number of samples.

10.2 Design a microcomputer-based capacitance meter using the following RC circuit:

The voltage across the capacitor is \( V(t) = \frac{k}{e^{t/RC}} \). In one time constant \( RC \), this voltage is discharged to the value \( k/e \). For a specific value
of R, the value of the capacitor C = π/β, where π is the time constant that can be counted by the microcomputer. Design the hardware and software for a microprocessor to charge a capacitor by using a pulse to a voltage of up to 10 V peak voltage via an amplifier. The microcomputer will then stop charging the capacitor, measure the discharge time for one time constant, and compare the capacitor value.

10.3 Design and develop the hardware and software for a microprocessor-based system to drive a four-digit seven-segment display for displaying a number from 0000H to FFFFH.

10.4 Design a microprocessor-based digital clock to display time in hours, minutes, and seconds on six-digit seven-segment displays in decimal.

10.5 Design a microcomputer-based temperature sensor. The microcomputer will measure the temperature of a thermistor. The thermistor controls the timing pulse duration of a monostable multivibrator. By using a counter to convert the timing pulse to a decimal count, the microcomputer will display the temperature in degrees Celsius.

10.6 Design a microprocessor-based system to test five different types of IC, namely OR, NOR, AND, NAND, and XOR. The system will apply inputs to each chip and read the output. It will then compare the output with the truth table stored in the memory. If the comparison passes, a red LED will be turned ON. If the comparison fails, the red LED will be turned OFF.

10.7 Design a microprocessor-based system that reads a thermistor via an A/D converter and then displays the temperature in degrees Celsius on three seven-segment displays.

10.8 Design a microprocessor-based system to measure the power absorbed by the 1k resistor. The system will input the voltage V across the 1k resistor and then compute the power using $V^2/R$.

Questions and Problems

10.9 It is desired to design a priority vectored interrupt system using a daisy-chain structure for a microcomputer. Assume that the system includes four interrupt devices DEVO, DEV1, ..., DEV3, which, during the interrupt sequence, place the respective instructions RST0, RST1, ..., RST3 on the data bus. Also assume that DEVO, ..., DEV3 are Telecine 8708 A/D converters (DEV3 highest, DEV0 lowest priority) or equivalent.

(i) Flowchart the problem to provide service routines for inputting the A/D converters' outputs.
(ii) Design and develop the hardware and software.

10.10 It is desired to drive a six-digit display through six output lines of a microcomputer system. Use eight Texas Instruments TIL311, 14-pin MSI hexadecimal displays or equivalent:
(i) Design the interface with minimum hardware.
(ii) Flowchart the software.
(iii) Convert the flowchart to the assembly language program.
(iv) Implement the hardware and software.

10.11 Design a microcomputer-based combination lock which has a combination of five digits. The five digits are entered from a hexadecimal keyboard and they are to be entered within 10 s. If the right combination is entered within the same limit, the lock will open. If after 10 s either all five digits are not entered or a wrong combination is entered, then the display will show an error signal by displaying "E". The system will allow 5 s for the first digit to be entered the second time. If after this time the digit is not entered, the system will turn ON the alarm. If the second try fails, the alarm is also turned ON. When the alarm is ON, in order to reset the system, power has to be turned OFF.

10.12 Design a microcomputer-based stopwatch. The stopwatch will operate in the following way: the operator enters three digits (two digits for minutes and one digit for tenths of minutes) from a keyboard and then presses the GO key. The system counts down the remaining time on three seven-segment LED displays.

10.13 Design a microcomputer-based system as shown in the following diagram. The system scans a 16-key keyboard and drives three seven-segment displays. The keyboard is scanned in a 4 x 4 X-Y matrix. The system will take each key pressed and scroll them in from the right side of the displays and keep scrolling as each key is pressed. The leftmost digit is just discarded. The system continues indefinitely.
10.14 Design a microcomputer-based smart scale. The scale will measure the weight of an object in the range of 0-5 lb. The scale will use a load cell as a sensor such as the one manufactured by transducer, Inc. (Model 4 c162-108#-10p) strain-gage load cell. This load cell converts a weight in the range of 0-10 lb. into analog electrical voltage in the range 0-20 mV. The weight in lbs. should be displayed onto two BCD displays.

10.15 Design a microcomputer-based EPROM programmer to program 2716.

10.16 Design a microcomputer-based system to control a stepper motor.

10.17 Design a microcomputer-based sprinkler control system.

10.18 Design a phone call controller. The controller will allow the user to pass only ten random phone numbers chosen by the user. The controller will use the touch-tone frequencies to encode the user information code numbers. A device will be used to decode the touch-tone signals and convert them into a seven-bit word. A microprocessor will then interpret this word and see if it is a match with one of the ten different numbers chosen by the user. The ten numbers are inputted by the user via the * button from the touch-tone system. The user will have a manual override via the # button from the touch-tone system.

10.19 Design a microprocessor-based appointment reminder system with a clock. The system will alert the user before the present appointment time. The user has to set the appointments into fixed slots; for example, 9 AM or 2 PM. The system will deliver a voice message such as "Your next appointment is five minutes away" five minutes before the appointment time. A real time is to be included in the system to display the current time and will show the appointment time slots. You may use the Radio Shack SP256 narrated speech processor.

10.20 Design a microcomputer-based autoranged ohmmeter with a range of 1 ohm to 999 kohms as follows: the microcomputer generates a pulse to charge a capacitor up to 10 V peak voltage through an amplifier and then stops charging the capacitor. The microcomputer measures the discharge time of the capacitor for one time constant and then computes the value of the resistor.

10.21 Interface two microcomputers to a pair 2K x 8 dual-port RAMs (HD71732) without using any bus locking mechanism. Two seven-segment displays will serve as an indicator. A program will be written to verify the dual-ported RAM contents. One processor will write some known data to the dual-ported RAM and the other processor will read and verify this data against the known data.

10.22 Design a microcomputer-based low frequency (1 Hz to 10 kHz) sine wave generator. One cycle of a sine wave will be divided into a certain amount of equal intervals. Each interval is defined as a phase increment. The precalculated sine values corresponding to the intervals are stored in ROM. The frequency of the signal will be set up by switches. When the system is started, the microprocessor will read the switches and will determine the time delay corresponding to the phase increment. The microprocessor will follow the time increments to send data to a D/A converter to convert the digital signal to an analog signal.

10.23 Design a microcomputer-based automobile alarm system. The purpose of this system is to prevent intruders from stealing a car or having enough time to steal a stereo or other valuable items in a car.

10.24 Design a microcomputer-based three-axis robot controller. The microcomputer will perform the calculations and the I/O to control the movement of the arm. The microcomputer will receive destination data from an external source and perform coordinate transformations and boundary checking on the external data. It will then provide motor commands to the motor controllers to move the arm to the desired position.
10.25 Design a microcomputer-based home controller system. The system will simultaneously control six sprinkler stations, a heater, an air conditioner, and a burglar alarm. The system will contain a 12-hour clock and a temperature reader. The user will program the system through a keypad. The time and temperature will be entered to control the sprinklers, the heater, and the A/C. The alarm will be armed or disarmed by entering a 4-digit code.

10.26 Design a microcomputer-based FM modulator. The microcomputer will read an analog input, convert the signal to digital, and perform several data manipulations to generate a digital representation of the FM signal. Finally, the microcomputer will convert the FM value to an analog signal.

10.27 Design and develop a microcomputer-based system for FFT (Fast Fourier Transform) computation. The microcomputer will sample eight data points using an A/D converter and compute the time-domain FFT. After computation of FFT, the result will be stored in system RAM where it can be used by another program for signal processing.

APPENDICES

Appendix A
HP 64000

Appendix B
Motorola MC68000 and Support Chips — Data Sheets

Appendix C
Intel 8085, 8086, and Support Chips — Data Sheets

Appendix D
Glossary
APPENDIX A

THE HEWLETT-PACKARD (HP) 64000

A.1 System Description

The HP 64000 Microprocessor Logic Development System is a universal development system which provides all of the necessary tools to create, develop, modify, and debug software for microprocessor-based systems. In-circuit emulation provides the capability of performing an in-depth analysis of hardware and software interfacing during the integration phase of the development process.

The HP 64000 Microprocessor Logic Development System is a multi-user development system, allowing up to as many as six users to operate on the system simultaneously. All users of the system share a line printer and a common data base in the form of a 12-megabyte Winchester Technology Disc Drive or a selection of one to eight Multi-Access Controller (MAC) disk drives connected to the system via the HP Interface Bus, commonly referred to as HP-IB. Eight disk drives can provide up to 960 megabytes of HP-formatted storage space.

957
A.2 Development Station Description

Figure A.1 shows the front view of the HP 64000 Development Station. The keyboard (Figure A.2) is divided into four areas: (1) an ASCII-encoded typewriter-type keyboard; (2) a group of edit keys, which facilitate movement of text or cursor when in the edit mode; (3) special function keys, for system reset or pause or to access a command recall buffer; (4) the all important system “soft keys.” Eight unobtrusive large key pads just beneath the bezel which surrounds the display.

The soft keys provide a quick and easy means to invoke system commands, virtually eliminating the typographical errors one usually has to contend with when having to enter commands character by character. The definition of each soft key is written on the display just above the keys. The soft key system changes depending on the mode of operation and the position of the cursor. This greatly enhances the ease of use of the system since it provides a list of alternatives available and guides the operator to use the system. In cases where the form of the input required is unknown, brackets surrounding a key word, a syntactical variable will prompt the user with the correct form of input the system expects.

The system display is a Raster Scan CRT which provides a display of 18 lines of text entry, a status line which always displays the system’s status and date and time, three lines for command entry, and the soft key label line which indicates the function of each key. The display is 80 columns wide, but with the edit keys the display can be relocated to show text or data out to 240 columns. This is convenient for adding comments and really enhances the program documentation.

Other external station hardware includes RS232 ports for communication with either Data Communications Equipment (DCE) or Data Terminal Equipment (DTE). The RS232 port has a selectable baud rate up to 9,600 and uses the X-ON X-OFF convention for handshaking at baud rates 2,400 and above. There is a 20-mA current loop for TTY interfacing and two ports for triggering of external devices such as an oscilloscope during a logic trace. As system options, the front panel hosts a PROM programmer (Figure A.1) to the immediate right of the keyboard and a tape drive for file back-up. The tape drive performs a high-speed read and write and each cassette holds 250K bytes of data.
Figure A.3 shows special function keys and Table A.1 summarizes their functions.

Figure A.3  Special function keys. Source: Courtesy of Hewlett-Packard.

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR/LINE</td>
<td>Press to clear the current line containing cursor on the CRT.</td>
</tr>
<tr>
<td>RECALL</td>
<td>Used to recall, to the command line, previous commands from a stack. The commands are displayed one at a time for each time the RECALL key is pressed. The number of recallable commands is variable. Only valid commands are pushed into the stack. If the RECALL key is pressed and the buffer is empty, the system responds with “Recall buffer is empty” message.</td>
</tr>
<tr>
<td>CAPS/Lock</td>
<td>Used to lock keyboard in all uppercase letters. A message is presented on the CRT indicating “CAPS LOCK on” or “CAPS LOCK off.” At the next key stroke, the message is erased, but the mode remains in effect.</td>
</tr>
<tr>
<td>RES/SET</td>
<td>Pressing RES/SET once initiates a pause in system operation. A flashing “PAUSED” message, in inverse video, is presented on the status line. To continue operation, press any key except RES/SET.</td>
</tr>
<tr>
<td>Pressing the RES/SET key the second time will clear the CRT and return the system to the system monitor.</td>
<td></td>
</tr>
<tr>
<td>Holding the SHIFT key down and pressing RES/SET initiates a complete system reboot. This function should be regarded as a last resort when the system does not respond.</td>
<td></td>
</tr>
<tr>
<td>CNTRL RES/SET</td>
<td>Holding CNTRL key down and pressing RES/SET initiates system performance verification.</td>
</tr>
</tbody>
</table>

Source: Courtesy of Hewlett-Packard.

The following summarizes the HP 64000 soft keys, commands, assembler error codes, and other features.

**System Monitor Soft Keys**

The following provides a description of the system monitor soft keys:

- **userid**
  - The userid or user identification identifies each user as being unique within the system. This facilitates file management in that once the userid command is invoked all future references to files will be to files under that userid unless explicitly stated otherwise.
  - The HP 64000 uses six characters and must begin with an uppercase alpha character.

- **time**
  - HH:MM. Allows the user to enter the correct time on the 24-hour clock displayed on the status line. This also facilitates file management since files can be referenced by time and date.

- **date**
  - DD/MM/YY. (Day/Month/Year) Allows the user to enter the correct date into the system. This aids the file management system since files can be referenced by date and time.

- **store**
  - This command will transfer files from the disk to the tape cartridge. The user specifies the file name and file type or all files. If all files are specified the system will store only the source files, linker command files, and simulator command files. Other file types may be stored but the file type must be specified. Other file types can readily be regenerated. This command will overwrite any previous contents of the tape cartridge.
Editor Commands

The 64000 editor commands are listed below:

- **append**
  - Allows files to be appended to files previously stored on tape.

- **verify**
  - Verify compares a file on the disk to a file resident on the tape cartridge. The user has the option of specifying a single file or all files on the tape assigned to the current user.

- **restore**
  - This command will transfer files from the tape cartridge to the disk. The user can specify the file name or names and file type.

- **purge**
  - This command will remove specified files from the active file list. Purged files can be recovered providing they have not been written over.

- **recover**
  - Recover is used to recover files which have been purged. Files, if not written over, will be returned to the active file list.

- **rename**
  - Allows the user to rename files. This is used to rename a file before recovering a previous file with the same name. This command also allows the user to transfer a file from one user to another user.

- **copy**
  - Copy allows a disk file to be copied to, or from, the tape. This copy may be copied to the printer.

- **directory**
  - This command provides a listing of those files on the disk, the tape cartridge, and those recoverable files.

- **library**
  - This command is used to build libraries of relocatable files for use by the linker. These library files consist of relocatable files that can be selectively loaded by the linker.

- **log**
  - This command creates a command file for all legal keystrokes. The log function is either toggled on or off by the log soft key.

- **(CMD_FILE)**
  - This soft key represents a syntactical variable to be supplied by the user. This variable is a file name consisting of system commands which the development system will execute. A command file can be generated through the use of the editor or by using the log soft key.

- **(line #)**
  - This command causes the line to become the current line of text.

- **end**
  - This command terminates the editor session and directs it to a specific destination. Usually this destination is a new file name. If no new file name is specified, the editor session terminates by purging the original file and replacing it with the edited file.

- **merge**
  - Merge allows the user to merge an entire file or portions of it into the file being edited. Any text added to the file being edited will be added after the current line. Delimiters can be specified to determine the amount to be merged.

- **copy**
  - Copy places specified text into a temporary storage buffer on disk for future use. The copy command will overwrite any text previously stored in the buffer. This is avoided by selecting the append option. The default value for (limit) is the current line only.

- **extract**
  - This command removes the specified lines and places them into temporary storage space. If the append option is not selected, the extracted text will overwrite previously stored text. If (limit) is not specified, the current line will be extracted.
This command retrieves the text from temporary storage and inserts it into the program following the current line. The user has the option regarding the number of times the text is to be retrieved.

**insert**
This allows insertion of a combination of ASCII characters after the current line of text. Insert is executable in the command mode, repeat and insert mode.

**list**
This allows the user to list a file to another file or to a printer in numbered or unnumbered format. The listing will be exactly like the file text. There is also a (limit) option available.

**renumber**
This command renumbers the edited text starting from line one.

**repeat**
Repeat allows the user to duplicate the current line of text and add it immediately after the current line. The user can specify the number of times the repeat command is executed.

**tabset**
This command allows the user to set tabs in the desired column. The user has the choice of all 240 columns. Any character can be used to set tabs in any desired location.

**range**
Restricts the columns to which find and replace commands are constrained. Columns 1 through 240 can be specified. The range function is toggled ON and OFF. When ON, the label range displays in inverse video.

**autocab**
This function provides an automatic tab function that is based on the first nonblank column of the present line of text. Depressing the shift and the tab keys simultaneously allows tab back from autotab position.

### Assembler Soft Key Definitions

The following provides the definitions of the 64000 assembler soft keys.

#### Key Label

- **(FILE)**
  This indicates the name of the source file that will be assembled.

- **list**
  This soft key specifies the destination of the assembler's output. The options available are listing the output to a specified file, to the display, to the printer, or to null (no generation of a list). If no list file option is specified, the assembler output inherits defaults to the device previously specified by the user when the userid was declared.

---

### Assembler Pseudoinstructions

Pseudoinstructions are instructions used only by the assembler. They produce no executable code for the processor and normally do not take up any memory locations. They are used by the assembler to make programming easier. The following list contains those pseudo ops and their definitions supported by the HP 64000 assembler.

<table>
<thead>
<tr>
<th>Op Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASC</td>
<td>Stores data in memory in ASCII format.</td>
</tr>
<tr>
<td>BIN</td>
<td>Stores data in memory in binary format.</td>
</tr>
<tr>
<td>COMN</td>
<td>Assigns common block of data or code to a specific location in memory.</td>
</tr>
<tr>
<td>DATA</td>
<td>Assigns data to a specific location in memory.</td>
</tr>
<tr>
<td>DEC</td>
<td>Stores data in memory in decimal format.</td>
</tr>
<tr>
<td>END</td>
<td>Terminates the logical end of a program module. Operand field can be used to indicate starting address in memory for program execution.</td>
</tr>
<tr>
<td>EQU</td>
<td>Defines label field with operand field value. Symbol cannot be redefined.</td>
</tr>
<tr>
<td>EXPAND</td>
<td>Causes an output listing of all source and macro generated codes.</td>
</tr>
<tr>
<td>ENT</td>
<td>Indicates symbol defined in another program module.</td>
</tr>
<tr>
<td>GLB</td>
<td>Defines a global symbol that is used by other modules.</td>
</tr>
<tr>
<td>HEX</td>
<td>Stores data in memory in hexadecimal format.</td>
</tr>
<tr>
<td>LIST</td>
<td>Used to modify output listing of program.</td>
</tr>
</tbody>
</table>
Appendix A

MASK
Performs logical operations on designated ASCII string.

NAME
Permits user to add comments for reference in the linker list.

NOLIST
Suppresses output listings (except error messages).

ORG
Sets program counter to specific memory address for absolute programming.

PROG
Assigns source statements to a specific location in memory.

Assembler default condition is "FROG" storage area.

REPT
Enables user to repeat a source statement any given number of times.

SKIP
Enables user to skip to a new page to continue program listing.

SPC
Enables user to generate blanks in program listing.

TITLE
Enables user to create a title line at the top of each page listing for the source program.

The following pseudo ops are for the 8080 and 8085 assembler.

DB
Stores data in consecutive memory locations staring with the current setting of the program counter.

DS
Reserves the number of bytes of memory as indicated by the value in the operand field.

DW
The define word pseudo stores each 16-bit value in the operand field as an address with the least significant byte stored at the current setting of the program counter. The most significant byte stored at the next higher location.

Assembler Error Codes

The following provides a description of the 64000 assembler error codes.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>ASCII string; the length of the ASCII string was not valid or the string was not terminated properly.</td>
</tr>
<tr>
<td>CL</td>
<td>Conditional label; syntax of a conditional macro source statement requires a conditional label that is missing.</td>
</tr>
<tr>
<td>DE</td>
<td>Definition error; indicated symbol must be defined prior to its being referenced. Symbol may be defined later in the program sequence.</td>
</tr>
<tr>
<td>DS</td>
<td>Duplicate Symbol: Indicates that the noted symbol has been previously defined in the program. This occurs when the same symbol is equated to two values (using EQU directive) or when the same symbol labels two instructions.</td>
</tr>
<tr>
<td>DZ</td>
<td>Division by zero; invalid mathematical operation resulting in the assembler trying to divide by zero.</td>
</tr>
<tr>
<td>EG</td>
<td>External Global: Externals cannot be defined as globals.</td>
</tr>
<tr>
<td>EO</td>
<td>External Overflow: Program module has too many external declarations (512 maximum).</td>
</tr>
<tr>
<td>ES</td>
<td>Expanded Source: Indicates insufficient input buffer area to perform macro expansion. It could be the result of too many arguments being specified for a parameter substitution, or too many symbols being entered into the macro definition.</td>
</tr>
<tr>
<td>ET</td>
<td>Expression Type: The resulting type of expression is invalid. Absolute expression was expected and not found or expression contains an illegal combination of relocatable types (refer to Chapter 2 of the Assembler Manual for rules and conventions).</td>
</tr>
<tr>
<td>IC</td>
<td>Illegal Constant: Indicates that the assembler encountered a constant that is not valid.</td>
</tr>
<tr>
<td>IE</td>
<td>Illegal Expression: Specified expression is either incomplete or an invalid term was found within the expression.</td>
</tr>
<tr>
<td>IO</td>
<td>Invalid Operand: Specified operand is either incomplete or incorrectly used for this operation. This occurs when an unexpected operand is encountered or the operand is missing. If the required operand is an expression, the error indicates that the first item in the operand field is illegal.</td>
</tr>
<tr>
<td>IS</td>
<td>Illegal Symbol: Syntax expected an identifier and encountered an illegal character or token.</td>
</tr>
<tr>
<td>LR</td>
<td>Legal Range: Address or displacement causes the location counter to exceed the maximum memory locations of the instruction's addressing capability.</td>
</tr>
<tr>
<td>MC</td>
<td>Macro Condition: Relation (conditional) operator in macro is invalid.</td>
</tr>
<tr>
<td>MD</td>
<td>Macro Definition: Macro is called before being defined in the source file. Macro definition must precede the call.</td>
</tr>
<tr>
<td>ML</td>
<td>Macro Label: Label not found within the macro body.</td>
</tr>
<tr>
<td>MM</td>
<td>Missing Mere: Indicates that a macro definition with a missing macro directive was included in the program.</td>
</tr>
<tr>
<td>MO</td>
<td>Missing Operator: An arithmetic operator was expected but was not found.</td>
</tr>
<tr>
<td>MP</td>
<td>Mismatched Parenthesis: Missing right or left parenthesis.</td>
</tr>
<tr>
<td>MS</td>
<td>Macro Symbol: A local symbol within a macro body was not found.</td>
</tr>
</tbody>
</table>
NM
Nested Macro. A macro definition is not permitted within another macro.

PC
Parameter Call: Invalid parameter in macro header.

PE
Parameter Error: An error has been detected in the macro parameter list in the source statement.

RC
Repeat Call: Repeat cannot precede a macro call.

RM
Repeat Macro: The repeat pseudo-operation code cannot precede a macro definition.

SE
Stack Error: Indicates that a statement or expression does not conform to the required syntax.

TR
Text Replacement: Indicates that the specified text replacement string is invalid.

UC
Undefined Conditional: Conditional operation code is undefined.

UO
Undefined Operation code: Operation code encountered is not defined for the microprocessor, or the assembler disallows the execution of the macro in its current context. This occurs when the operation code is misspelled or an invalid delimiter follows the label field.

UP
Undefined Parameter: The parameter found in macro body was not included in the macro header.

US
Undefined Symbol: The indicated symbol is not defined as a label or declared as an external.

Linker Commands

The 64000 linker commands are defined below:

<table>
<thead>
<tr>
<th>Key</th>
<th>Label</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>link</td>
<td>(CMDFILE)</td>
<td>Initiates the link process. A syntactical variable supplied by the user. Would be the name of linker command file previously established.</td>
</tr>
<tr>
<td>listfile</td>
<td></td>
<td>Allows the user to select a destination other than the system default for the linker output listing.</td>
</tr>
<tr>
<td>display</td>
<td></td>
<td>Using this command designates the display as the output destination for the linker output listing.</td>
</tr>
<tr>
<td>(FILE)</td>
<td></td>
<td>Synthetical variable supplied by the user. Would be the name of a disk file to which the output of the linker would be directed.</td>
</tr>
<tr>
<td>null</td>
<td></td>
<td>Using this command suppresses the output listing. Error messages will still be output to the default destination as previously selected by the user.</td>
</tr>
</tbody>
</table>

Soft Key Definitions

The 64000 emulator soft key definitions are given below:

<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>run</td>
<td>This starts program execution in the emulation processor. Execution begins at the location specified by &quot;from&quot;, and ending under the conditions specified by &quot;until.&quot; If no limits are specified, emulation will begin at the current address until halted by a &quot;stop run&quot; or by a boundary specified by &quot;until.&quot; Syntax: run from (ADDRESS OR SYMBOL) until (ADDRESS OR SYMBOL).</td>
</tr>
<tr>
<td>step</td>
<td>This function causes the emulation processor to execute one instruction at a time. Once in the step mode, each expression of the return key will cause another instruction to be executed and displayed. The user can specify the number of steps to be executed each time the return key is pressed and the address from which stepping occurs. If these parameters are not specified, the system defaults to stepping from the current program counter location, executing one instruction each time the return key is pressed. Syntax: step # of (STATES) from (ADDRESS).</td>
</tr>
<tr>
<td>trace</td>
<td>This key is used to control the analysis function of the system, allowing the triggering and capturing of data of the emulation data bus. Syntax: trace in sequence—permits tracing on a sequence of events. trace after—catures and displays data after the trigger qualifier word is satisfied. trace before—captures and displays data before and after the trigger qualifier. trace only—permits explicit definition of the information to be captured in the trace.</td>
</tr>
</tbody>
</table>
trace continuous—allows continuous monitoring of trace information without reentering the trace command.

display This command causes the system to display a variety of data types on the development station's screen. Data types can be specified as global symbols, local symbols, and last active trace specification (valid only with the analysis card), at last active run specification, the trace buffer (valid only with analysis card), contents of pre-emulation microprocessor registers, absolute or relative time display (valid only with analysis card), or contents of user or emulation memory.
Syntax: display trace
Syntax: display register (REGISTER NAME)
Syntax: display memory (ADDRESS)
Syntax: display trace specification
Syntax: display run specification
Syntax: display count
Syntax: display global symbols
Syntax: display local symbols

The mode option for the trace, register, and memory display provides the user with a choice of how the data will be presented on the screen. The following modes are defined:

static The system will display the current conditions or contents one time only. No update will be shown.
dynamic The system will continually update the display as data are changed in the emulation system.
absolute The system displays data in absolute numeric code (i.e., hexadecimal or octal).
mnemonic The system presents the data in the appropriate assembly language.
effect by The system displays program modules so that the address values are offset by a specified value.
no offset The system displays all addresses in program modules with those values assigned by the linking loader.
packed The system displays operands and operands on the same line.
block The system displays more data on the development station by displaying multiple columns of data.
modify This command allows the user to change the contents of the emulation memory or processor registers to correspond to data entered from the console keyboard.
Syntax: modify (ADDRESS) to (VALUE)
Following are the monitor level soft keys which will be in effect after December 1981:

```
edit compile assemble link simulate prnt prog run etc...
directory pargen rename copy library recover log etc...
send datetime opt test terminal (CMDFILE) -TAPE-- etc
```

"-TAPE-" Soft Key
After -TAPE-- is returned the following soft keys are available.

```
more restore append verify tension directory etc...
```

"date&time" Soft Key
After date&time is depressed the following soft keys are available.

```
(DATE) (TIME)
```

"opt...test" Soft Key
This executes option test, which provides performance verification tests for options that are present.

**Terminal Mode**

"terminal Soft Key"
This puts the station in an RS232 terminal mode which allows it to be a terminal to another system.

**Passwords**

The capability to have increased file security using passwords has been added. Following is the new syntax for user id.

"userid" Soft Key
After userid is depressed the following soft keys are available.

```
(USERID) listfile
```

After USERID is entered the following soft keys are available.

```
lstfile password
```

After password is entered the following soft keys are available.

```
(PASSWD)
```

The user types in his password. This is not printing so he will not see on the display what he entered.

**“HOST” PASCAL**

"HOST” PASCAL consists of a compiler to allow users of the 64000 system to write programs that will execute on the internal host processor. In order to execute these programs the following syntax is used.

"run" Soft Key
After run is depressed the following soft keys are available.

```
(FILE)
```

After a file is specified the following soft keys are available.

```
input output
```

After input is depressed the following soft keys are available.

```
(FILE) keyboard
```

After output is depressed the following soft keys are available.

```
(FILE) display display printer null
```
Summary of the HP 64000 Development System

Example A-1
This example shows how to create a new file and edit it. The file to be created is listed below:

```
; This program starts at 0 and adds location 100H to
; location 101H and stores the result in location 102H
NAME ADD_WORKSHOP
ORG 200H
START LXI H,100H
MOV A,M
INX H
ADD M
INX H
MOV M,A
JMP START
```

Procedure
Step 1: Press soft key “userid” and type in your USERID. The HP doesn’t ask for the time and date after you enter your USERID. Rather, you have to press the soft key “Date&Time” to change it.

Step 2: To enter the edit mode, you have to create a new file. We’ll call this new file “ADD”.

edit into ADD (RETURN)

Step 2.5: It is a good idea to set up tabs so that if you want to move, you may jump to the opcode, operand, or comment. You do this by pressing the softkey “Tabset”. The editor will then display a tab row in which you can type “TT” at the current cursor position. Then when you want to move faster, the “tab” key will jump to where you set your tabs. The way we did it was:

```
tabset 7 17 27 37
```

To save your tab sets, type the inverse softkey text “tabset” again.

Step 3: The first line of the program is the assembler directive, which lets the assembler know what microprocessor you wish to emulate.

```
; This is a comment (The “;” is at the leftmost edge of column 1);
; This is a comment (The “;” can be at any position).
```

Note: Your comments must come after you type “;” or “;”.

Step 4: To enter comments, type a “;” in column 1 and then start with your comments. If you want to start it anywhere else on that line, type a “;” and enter your comment.

Step 5: Enter “Name” and a brief explanation of the file. This lets you know what a particular file does in case you have to link many files. This is optional.

```
(TAB) NAME (TAB) "ADD_WORKSHOP" (RETURN)
```

Step 6: Enter “ORG” to let the assembler know the starting location of your program — in this case, at 2000H.

```
(TAB) ORG (TAB) 2000H (TAB) ; COMMENTS (RETURN)
```

Step 7: Enter the label “START” at column 1 of the next line along with the first instruction. This label helps the user remember the English word rather than what the number was, if the user wants to loop or jump to that part of the program again.

```
START (TAB) LXI (TAB) H,100H (TAB) ; COMMENTS (RETURN)
```

Step 8: If you use only a few labels in your program, it is wise to use the softkey “AUTOTAB”. This softkey jumps to the next line and moves the cursor right under the first word of the previous line. This saves time. Also note that the “TAB” key can also do this if you specified the tab sets.

Step 9: Enter the rest of the program. This is the program listing called “ADD”.

Step 10: To list your file to the printer, make sure the printer is on-line (the light that’s adjacent to the word should be on; if not, press the “on-line” button on the printer). Then type:

```
list printer all (RETURN)
```

Step 11: To save the file, type

```
end (RETURN)
```

Note: If your file wasn’t named when you entered the edit mode, then type

```
and ADD (RETURN)
```

Step 12: The file is stored onto the hard drive or disk. To see your file on it, type:

directory (RETURN)

446.jpg
You should then see the file "ADD" with the type "SOURCE". You will also see when you last modified it and accessed it. This information is important, so that you know how updated your file is. The directory listing will only show those files under your USERID.

Step 15: To reedit the file, type

```
edit ADD (RETURN)
```

Notice that you don't type "edit into ADD". If you want to load your file from a disk, you have to specify the drive number. For disk drive X, type "ADD-X", where X is the disk drive number. If no drive number is specified, then the default is 0.

Step 16: To use the insert softkey, type a "NOP" after line 9 by

```
9 (RETURN)
insert (TAB) NOP (TAB) (TAB) ; NO OPERATION (RETURN)
```

Note: If you get an error, go to Step 15 and then back to 14. This may be because the editor was trying to find line 9, but your file has line numbers reading "NEW" instead.

Step 17: To remember your file in order to give your editor a way to find what line to edit, type

```
renumber (RETURN)
```

Step 18: If your file is very large and you want to search your file for a particular word like "NOP", type

```
find 'NOP' all (RETURN)
```

Note: Remember to enclose all strings with double quotes. If not, the editor will think it is a softkey command.

Step 19: To insert more text, type

```
insert (RETURN)
```

then move the cursor up, down, or sideways and begin typing the new line.

Step 20: To revise a line, enter

```
revise (RETURN)
```

This edits the line that the cursor is on. If that line isn't what you want, then move the cursor using the cursor keys.

Step 21: To move the display to allow for viewing all of the columns, depress the SHIFT and LEFT arrow keys simultaneously. Hitting SHIFT and

```
RIGHT keys will scroll the text right. To scroll the text up or down, hit the edit key ROLL UP or ROLL DOWN, respectively.
```

Step 22: To insert or delete character(s) when revising, hit the edit key INSERT CHAR or DELETE CHAR, respectively.

Step 23: To delete a line at the current cursor position, hit DELETE (RETURN). If you want to delete a line somewhere else, type

```
extract (RETURN)
```

Then move the cursor to where you want to insert that line and type

```
retrieve (RETURN)
```

Note: If you want to insert many copies of that line at the current cursor position, then type retrieve # (RETURN), where # is the number of copies.

Step 24: To abort the editor and not save your file, press the special function key RESET twice. Pressing it once will pause a running listing or program.

Step 25: To replace a word with another word, type

```
replace "word1" with "word2" all
```

This will replace all words "word1" by "word2". You can also specify when you want to stop replacing by using thru or until a certain line number.

Step 26: The "copy" command lets you copy a group of lines without erasing those lines, like "extract" does. First, place the cursor at the starting location line and then type

```
copy thru line # (RETURN)
```

where # is the last of your lines to copy. Next, move the cursor to the place where you want it inserted, and type retrieve.

Step 27: The "merge" command lets you insert an entire file or copies a block of lines like the "copy" command does. To merge a file, type

```
merge ADD (RETURN)
```

This lets you insert the file ADD at the current cursor position.

**Example A-2**

This example goes through the steps in assembling a file. Upon completion, it will create a "relloc" file to be later used for linking purposes.

**Procedure**

Step 1: Enter your userid, and optionally enter the time and date. The HP 64000 already has the current date and time, so entering isn't necessary.
These files are useful for the assembler and linker programs. The "reloc" file is an object file containing the hex values of your program. It then must be made into an "absolute" file so it can run by itself.

Example A-3
This example goes through the steps in linking the "reloc" file to create an "absolute" file. This new file can then be run independently, emulated, or tested by the PROM programmer.

Procedure
Step 1: Initialize the linker and show the results to the display by

```
link listfile display (RETURN)
"Object files ?"
```

Step 2: This new message asks you what file(s) you want to be linked, so type

```
ADD (RETURN)
"Library files ?"
```

Step 3: There are no library routines in "ADD" so skip it by

```
(RETURN)
"Load addresses: PROG,DATA,COMN = 0000H,0000H,0000H"
```

Step 4: This command allows you to specify different memory areas for the program, data, and common modules. No memory assignment is needed because the "ADD" file already has an ORG statement, so skip it by

```
(RETURN)
"More files ?"
```

Step 5: Since there is only one file to be linked, respond by

```
no (RETURN)
"LIST,XREF,overlap_check,compo db = on off on off"
```

Step 6: The linker is prompting the user to specify the output and declaring the default for the output listing. It then checks to see if your memory assignments overlap as well. Ignore this and type

```
(RETURN)
"Absolute file name ?"
```

You will then see two more "ADD" files with the type "reloc" and "xref".
Step 7: The linker wants you to enter the file name to be assigned to the absolute file.

ADD (RETURN)

Step 8: You will then see the linker examining your file

"STATUS: Linker: HP 64000S linker: Pass 1"
"STATUS: Linker: HP 64000S linker: Pass 2"
"STATUS: Linker: HP 64000S linker: End of link"

Note that the above display may be on a single line.

Step 9: The output will display the start and end location of your program, the current date and time, the assembler pseudo name

"ADD_WORKSHOP1" and extra data like XFER address and the total bytes loaded.

Step 10: To view your new files on the directory, type

directory (RETURN)

You should then see three new files with types "link_sym", "link_com", and "absolute". The absolute file is used for the emulator or the PROM program.

"link_sym" is a command file that holds all of the data that you need. The "link_com" is a command file that holds the data that you entered from Steps 1 to 7. This is good if you want to keep the same link configuration, but want to change or re-edit your source file(s).

If you want to save a linker listing to a file for later viewing type

link ADD listfile ADD.L (RETURN)

This will create a link listing similar to that of Step 9. Specifying "link ADD" will tell the linker to link it using its "link_com" file.

Example A-4

This example goes through the steps of emulating an absolute file without external hardware. This gives you a good idea of the importance of a development system like the HP 64000.

Procedure

Step 1: Before beginning, you must go through Examples 1 through 3. Also, the HP 64000 must have an I/O set up. If it is configured or has a 6000 emulator, emulation program will use only it, so be wary of this.

Step 2: To enter the emulation mode and load the absolute file do this

emulate load ADD (RETURN)

"Processor 0 clock?"

Step 3: This question asks if you want the source of the processor clock to be internal or external. Since there is no external hardware being used, type

internal (RETURN)

"Restrict processor to real-time runs?"

Step 4: This question asks if you want to restrict the processor to real-time runs which will limit the analysis functions that can be performed. Such as debugging your program. An example of this would be "display registers blocked". So answer

no (RETURN)

"Stop processor on illegal opcodes?"

Step 5: Specify "yes" so that the emulator will stop if an illegal opcode is detected.

yes (RETURN)

Step 6: The emulator will then want to specify the memory range for your emulation environment and user environment. Since all memory is internal (no user external hardware for this file), address 100H to 103H is used for storing the variables used in the program. Define this to be emulation RAM. To protect your program, define the memory to be emulation ROM. Thus, if something writes to your program, it will generate an error.

100H thru 103H emulation ram
2000H thru 20FFH emulation rom

Also notice that the ram and rom ranges are from 100H-3FFH and 2000H-20FFH. This may be because the emulator can only provide a range of memory area rather than a specified one.

Step 7: To keep your defined memory area, type

and (RETURN)

"Modify simulated I/O?"

Step 8: Since we are not using any I/O ports, type

no (RETURN)

"Modify interactive measurement specification?"

Step 8.5: This question is not in the book so type


(RETURN)
"Command file name ?"

Step 9: The emulator will then load your absolute file to the memory
areas you specified. You should then see

"STATUS: 8085--Program loaded"

If you want to make any modifications, you have to start again by re-editing,
assemblying, linking and then emulating. If you want the same emulating con-
figuration that you specified in Steps 1 to 9, then type emulate ADD load
ADD.

Step 11: To display your program with mnemonic, type

display memory 2000H mnemonic (RETURN)

You should then see the locations with their corresponding instructions of your
program.

Step 12: To change the values that your program uses to add two numbers,
you have to modify the emulation RAM by

modify memory 100H thru 122H to 02H (RETURN)

display memory 100H blocked (RETURN)

You should now see a display of your edited bytes. The memory block map will
show you the address, data, and ASCII translations of each byte.

Step 13: To run your program, you can type either

run from 2000H (RETURN)

or

run from glob_sym START (RETURN)

"STATUS: 8085--Running"

Your program will keep running, so you can now modify the memory locations
from Step 12 to something else, and then see the changes.

Step 14: You can also single-step the program to execute a single instruction
at a time. This is a good debugging tool on the 6400.

break (RETURN)

"STATUS: 8085--break in background"

display registers (RETURN)

step from 2000H (RETURN)

or

step from glob_sym START (RETURN)

(RETURN)

Continuously pressing return will execute the "step" command again. Remember
that pressing "return" will execute anything on the command prompt, no
matter where the cursor is. You should then see a display of each instruction
being executed with its corresponding register values. This is really good be-
cause you can trace any program, and scan the instructions, registers, flags,
stack pointer, and the next IP.

Step 15: You can also set up breakpoints to stop the program when it reaches
a certain argument. An example of this would be "run" from 2000H until
address 2006H.

Step 16: To set up a breakpoint at address 2005H, type

run from 2000H until address 2005H (RETURN)

Step 17: Now say you want to halt the program after a memory write:

run from 2000H until status memory_write (RETURN)

Step 18: To end the emulation session, type

end (RETURN)

You can also press "RESET" twice too.

Step 19: If you want to get back to the emulation and keep the same
emulator configuration, type

emulate ADD load ADD (RETURN)

If you want to change the emulator configuration type

modify_configuration (RETURN)

Note that the HP 64000 can perform in-circuit emulation with or without a
large-system hardware.
Operation of the 68000 Emulator

In order to use the 68000 emulator, a monitor program must be included in the linking of the user’s program. The purpose of the monitor is to provide special functions during emulation (including register display, software breakpoint setting, etc.).

The steps in the emulation process are as follows:

1. Create a program using the 64000’s text editor.
   A. Make sure that "68000", including quotations, is the first line in the editor.
   B. Make the second line in the application program “PROG”. (This will cause the monitor program to be successfully linked with the application.)
   C. Use “H” for Hex instead of “$” signs.
   D. Write your application program.
   Locate the program between 0000H and 10000H application size.
2. Assemble the program you write by typing:

   <Assemble> MON_68K

3. Make a copy of the assembly program “MON_68K” by typing:

   COPY Mon_68K-HP-source TO MON_68K

   (Note upper and lower case. Type it exactly as shown.)
4. Now assemble this program by typing:

   SOFTKEY
   <Assemble> MON_68K

5. Now, the application and monitor program must be linked together. Type:

   SOFTKEY
   <LINK>

   Object File? MYFILE
   Library Files? <CR>
   Prog,Data,Comm,A5= 000000H,04H,04H where XXX=100H to 10000H
   More Files? <yes> Soft Key
   Object File? MON_68K
   Library Files? <CR>
   Prog,Data,Comm,A5= 10000H,04H,04H
   More Files? <No> Soft Key
   Absolute File Name? MYFILE

A. Notes: The monitor program is position independent. Since the TARGET SYSTEM has limited address space, it is suggested that you locate your program in THAT address range. Furthermore, this allows the user to specify the monitor program’s address in upper address space and in emulation RAM.

B. Care should be taken so that address ranges 000H-0FFFH are reserved for vectors, and that user program addresses do not conflict with the monitor program whose size is about 1001 bytes. I locate the monitor at address 1000H-10FFFH.

6. Finally, the emulator is started by answering questions with their default values (unless the user wishes otherwise).

   A. For the memory map, the following should be entered:

   0000H-0FFFH Emulation ROM (Vectors, Interrupts, User Prog)
   10000H-10FFFH Emulation RAM (Monitor Program)
   01000H-01FFFH User RAM (other space for ports, tables, etc. that exist in either software or the target system)

   B. Some helpful commands during emulation (<xxxx> = Soft Key):

   1. <Load> MYFILE

      Loads both your file and the monitor program (linked).

   2. <Modify><Config>

      Lets you change the emulator configuration.

   3. <Modify><Software_bkpt>

      Modifies software breakpoints so that you can stop program execution anywhere.

   NOTE: The 68000 emulator DCES NOT allow single-stepping.

   4. <BREAK>

      Enter the monitor program.

   5. <DISPLAY><MEMORY>

      Display disassembled code.

   6. <MODIFY><REGISTER>

      Modify address space and registers.

   DO NOT put ORG statements in your program except for the interrupt vectors.
**MC68230 Parallel Interface/Timer**

The MC68230 Parallel Interface/Timer provides versatile double buffered parallel interface and an interrupt system aimed at MC68000 systems. The parallel interface provides an unbuffered or bidirectional mode, where 8 or 16 bits wide, in the unbuffered mode, an enabled data direction register determines whether the port pins are inputs or outputs. In the bidirectional mode, the data direction register is ignored and the direction is determined electrically by the state of the data pins. The data pins can be configured as push-pull or tri-state output drivers. These programmable interface lines facilitate easy connection to a wide variety of load resistors. A dedicated interrupt request line is provided, and an interrupt acknowledge line is also provided. The interrupt acknowledge line is asserted if the interrupt request line is high. An external interrupt is generated when a mode transition occurs, as shown in the timing diagram. The interrupt is active low and can be used to signal the occurrence of a specific event.

- **Pinout Information:**
  - Port0 Address: 40.4H
  - Port1: 40.0H

- **Pin Assignment:**
  - D1: 40.0H
  - D2: 40.1H
  - D3: 40.2H
  - D4: 40.3H
  - D5: 40.4H
  - D6: 40.5H
  - D7: 40.6H

- **Features:**
  - 48-bit Parallel Interface
  - 8-bit or 16-bit Data Bus
  - Programmable Control Register
  - Timer with 1024-page Addressable Counter
  - Interrupts on Timer Overflow, Data Ready, and Port 0/1

- **Applications:**
  - General-purpose microcontroller
  - Digital signal processing
  - Communication systems

- **Pin Descriptions:**
  - D0-D7: Data Inputs
  - D4H-D7H: Data Outputs
  - D1H-D3H: Data Outputs
  - D0H-D1H: Data Inputs
  - D4H-D7H: Data Outputs
PERIPHERAL INTERFACE ADAPTER (PIA)

The MC6821 Peripheral Interface Adapter provides the universal means of interfacing peripheral equipment to the 6800 family of microprocessors. The device is capable of interfacing the MPU to communication through two 8-bit bidirectional parallel data buses and two control lines. An external logic is required for interfacing to most peripheral devices.

The functional configuration of the PIA is programmed by the MPU through its internal control registers. Each of the peripheral data interfaces can be programmed to act as an input or output, and each of the four control signals may be programmed for one of several control modes. This allows a high degree of flexibility in the overall operation of the interface.

- 8-bit Bidirectional Data Bus for Communication with the MPU
- Two Bidirectional 8-bit Buses for Interface to Peripherals
- Two Programmable Data Direction Registers
- Four Individually-Controllable Interrupt Lines: Two Used as Peripheral Control Outputs
- Handshake Control Logic for Interface to Output Peripheral
- High-impedance, Three-State and Direct Translator Drive
- Program Controlled Interrupt and Interrupt Disable Capability
- CMOS Drive Capability on Side A Peripherals
- Two TTL Drive Capabilities on Side A and Side B Interface
- TTL Compatibility
- Static Operation

MAXIMUM RATINGS

<table>
<thead>
<tr>
<th>Condition</th>
<th>Symbol</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply Voltage</td>
<td>VCC</td>
<td>5</td>
<td>V</td>
</tr>
<tr>
<td>Operating Temperature Range</td>
<td>TMIN</td>
<td>0</td>
<td>°C</td>
</tr>
<tr>
<td></td>
<td>TMAX</td>
<td>70</td>
<td>°C</td>
</tr>
<tr>
<td>Storage Temperature Range</td>
<td>TSTG</td>
<td>-40</td>
<td>°C</td>
</tr>
</tbody>
</table>

THERMAL CHARACTERISTICS

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Symbol</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thermal Resistance</td>
<td>RθJA</td>
<td>50</td>
<td>°C/W</td>
</tr>
<tr>
<td>Junction to Case</td>
<td>TJC</td>
<td>40</td>
<td>°C</td>
</tr>
</tbody>
</table>

The device is designed to operate within its rated limits without excessive damage due to high excursions of electrical factors. However, it is advisable that control specifications be followed. In general applications, higher than nominal logic levels are desirable to the high-voltage circuit. Because of differences in the approach to signal and output circuits, there is a nominal logic voltage range, which is 2V to 5V. (11)

FIGURE B.1 EXPANDED BLOCK DIAGRAM
Motorola Data Sheets

Appendix B

1. INTERFACE SIGNALS FOR MPU

The interface to the 8510 MPU bus with its 6-bit bidirectional data lines, two clock selects, two negative enable lines, and three internal state lines is used in most microcomputer systems. To avoid problems with the interfering state lines, the 6-405, 8086, or 8088 microprocessors, such as the 6502, may be preferred. An exception to this would be the state lines, which should be considered as part of the address decoding.

2. Bidirectional Data Lines (DS0-D7): The bidirectional data lines do not show a marked data between the MPU and the state lines. The data lines are used to read and write data to the MPU. These lines can be used to map the state lines to the data transfer signals and the state lines to the data lines. The state lines are used to select the state lines that are valid.

3. Enable (E): This signal is generated by the MPU to control the data transfer from the MPU to the state lines. This signal is pulled high and the state lines are used to control the state lines that are valid.

4. Reset (RST): This signal is used to reset the MPU to a known state. When the MPU is reset, it will perform a reset operation and will set the state lines to a known state.

5. Clock Select (CS1, CS2): These signals are used to select the state lines that are valid. The state lines are selected by the state lines that are valid.

6. Typical State Lines (S0-S7): These signals are used to control the state lines that are valid. The state lines are selected by the state lines that are valid.

The interface to the 8510 MPU bus with its 6-bit bidirectional data lines, two clock selects, and three internal state lines is used in most microcomputer systems. To avoid problems with the interfering state lines, the 6-405, 8086, or 8088 microprocessors, such as the 6502, may be preferred. An exception to this would be the state lines, which should be considered as part of the address decoding.

The interface to the 8510 MPU bus with its 6-bit bidirectional data lines, two clock selects, and three internal state lines is used in most microcomputer systems. To avoid problems with the interfering state lines, the 6-405, 8086, or 8088 microprocessors, such as the 6502, may be preferred. An exception to this would be the state lines, which should be considered as part of the address decoding.

The interface to the 8510 MPU bus with its 6-bit bidirectional data lines, two clock selects, and three internal state lines is used in most microcomputer systems. To avoid problems with the interfering state lines, the 6-405, 8086, or 8088 microprocessors, such as the 6502, may be preferred. An exception to this would be the state lines, which should be considered as part of the address decoding.

The interface to the 8510 MPU bus with its 6-bit bidirectional data lines, two clock selects, and three internal state lines is used in most microcomputer systems. To avoid problems with the interfering state lines, the 6-405, 8086, or 8088 microprocessors, such as the 6502, may be preferred. An exception to this would be the state lines, which should be considered as part of the address decoding.

The interface to the 8510 MPU bus with its 6-bit bidirectional data lines, two clock selects, and three internal state lines is used in most microcomputer systems. To avoid problems with the interfering state lines, the 6-405, 8086, or 8088 microprocessors, such as the 6502, may be preferred. An exception to this would be the state lines, which should be considered as part of the address decoding.

The interface to the 8510 MPU bus with its 6-bit bidirectional data lines, two clock selects, and three internal state lines is used in most microcomputer systems. To avoid problems with the interfering state lines, the 6-405, 8086, or 8088 microprocessors, such as the 6502, may be preferred. An exception to this would be the state lines, which should be considered as part of the address decoding.

The interface to the 8510 MPU bus with its 6-bit bidirectional data lines, two clock selects, and three internal state lines is used in most microcomputer systems. To avoid problems with the interfering state lines, the 6-405, 8086, or 8088 microprocessors, such as the 6502, may be preferred. An exception to this would be the state lines, which should be considered as part of the address decoding.
APPENDIX C

intel 8085

<table>
<thead>
<tr>
<th>N</th>
<th>1</th>
<th>49</th>
<th>Y</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>2</td>
<td>30</td>
<td>30</td>
<td>HOLD</td>
</tr>
<tr>
<td>00</td>
<td>4</td>
<td>33</td>
<td>CLI/DIR</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>55</td>
<td>33</td>
<td>SET</td>
<td>IN</td>
</tr>
<tr>
<td>TR</td>
<td>6</td>
<td>35</td>
<td>E</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>33</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure C.1 8085 pinout

Figure C.1 shows 8085 pins and signals. The following table describes the function of each pin:

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Address bus: The most significant 8 bits of the memory address or the 1 bits of the I/O address. Multiplexed address/data bus: Lower 8-bits of the memory address (or I/O address) appear on the bus during the first clock cycle (T1 state) of a machine cycle. It then becomes the data bus during the second and third clock cycles. Address (Latch Enable): It occurs during the first clock state of a machine cycle and enables the address to get latched into the on-chip latch. Machine cycle status:</td>
</tr>
<tr>
<td>I/O</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### Appendix C

#### Symbol: FUNCTION

<table>
<thead>
<tr>
<th>I/O/M</th>
<th>S1</th>
<th>S2</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>I/O read</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Op code fetch</td>
</tr>
<tr>
<td>*</td>
<td>0</td>
<td>1</td>
<td>Interrupt acknowledge</td>
</tr>
<tr>
<td>*</td>
<td>X</td>
<td>X</td>
<td>Hold</td>
</tr>
<tr>
<td>*</td>
<td>X</td>
<td>X</td>
<td>Reset</td>
</tr>
</tbody>
</table>

* = 3-state (high impedance)
X = unspecified

S1 can be used as an advanced R/W status. I/O/M, S1, and S2 become valid at the beginning of a machine cycle and remain stable throughout the cycle. The falling edge of ALR may be used to latch the state of these lines.

**RD** (Output, three-state)

WHITE control. A low level on RD indicates the data bus is to be written into the selected memory or I/O location.

**WR** (Output, three-state)

If WR is high during a read or write cycle, it indicates that the memory or peripheral is ready to send or receive data. If WR is low, the CPU will wait an integral number of clock cycles for WR to go high before completing the read or write cycle.

**READY** (Input)

HIGH indicates that another master is requesting the use of the address and data buses. The CPU, upon receiving the REady signal, relinquishes the use of the bus as soon as the completion of the current bus transfer is completed. All normal processing can continue. The processor can regain the bus only after the READY signal is removed. When the CPU is not busy, the READY, RD, WR, and ID M lines are three-stated.

**HOLD** (Input)

HOLD indicates that another master is requesting the use of the address and data bus. The CPU, upon receiving the HOLD request, will relinquish the use of the bus as soon as the completion of the current transfer is accomplished. All normal processing can continue. The processor can regain the bus only after the HOLD signal is removed. When the CPU is not busy, the HOLD, RD, WR, and ID M lines are three-stated.

**HOLD ACKNOWLEDGE** (Output)

HOLD ACKNOWLEDGE indicates that the CPU has received the HOLD request and that it will relinquish the bus in the next clock cycle. If the CPU does not receive the HOLD ACKNOWLEDGE signal after the CPU has relinquished the bus, the CPU will assert a bus error.

**INTR** (Input)

Interrupt request. See the description in "Appendix A" for further information.

**INTA** (Output)

Interrupt acknowledge. See the description in "Appendix A" for further information.

**RST 5**

Reset 5. For use in the reset circuit of the processor. This input is driven by the processor's internal circuitry when the processor is reset.

**RST 6**

Reset 6. For use in the reset circuit of the processor. This input is driven by the processor's internal circuitry when the processor is reset.

**TRAP**

Trap Interrupt is a non-maskable interrupt. It is recognized at the same time as INTR or RST 5-7. It is unaffected by any mask or interrupt disable. It has the highest priority of any interrupt.

**RESET IN**

Indicates CPU is being reset. Can be used as a system reset.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, X'</td>
<td>X and X' are connected to a crystal, L/C, or RC network to drive the internal clock generator. X can also be an external clock input from a logic gate. The input frequency is divided by 2 to give the processor's internal operating frequency.</td>
</tr>
<tr>
<td>CLK</td>
<td>Clock output for use as a system clock. The period of CLK is twice the X, X' input period.</td>
</tr>
<tr>
<td>SID</td>
<td>Serial input data line. The data on this line is loaded into accumulator bit 7 whenever a SID instruction is executed.</td>
</tr>
<tr>
<td>SOC</td>
<td>Serial output data line. The output SOC is set or reset as specified by the SIM instruction.</td>
</tr>
<tr>
<td>Vcc</td>
<td>5 V supply.</td>
</tr>
<tr>
<td>Vss</td>
<td>Ground reference.</td>
</tr>
</tbody>
</table>
**8086/8086-2/8086-4**

16-BIT HMOS MICROPROCESSOR

- Direct Addressing Capability to 1 MByte of Memory
- Assembly Language Compatible with 8086/6808
- 14 Word, By 16-Bit Register Set with Symmetrical Operations
- 24 Operand Addressing Modes
- 8-bit, Byte, Word, and Block Operations
- 8-and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal Including Multiply and Divide
- 5 MHz Clock Rate (8 MHz for 8086-2)
- 4 MHz for 8086-4
- MULTIBUS™ System Compatible Interface

The Intel® 8086 is a new generation, high performance microprocessor implemented in n-channel, depletion load, silicon gate technology and is packaged in a 40-pin DIP package. The processor has 16 data bits and 16 address bits, allowing it to address memory as a sequence of 32-bit bytes, but has 16-bit wide physical path to memory for high performance.

---

**18284**

CLOCK GENERATOR AND DRIVER FOR 8086, 8088, 8089 PROCESSORS

- Generates the System Clock for the 8086, 8088 and 8089
- Uses a Crystal or a TTL Signal for Frequency Source
- Single ±5V Power Supply
- 18-Pin Package
- Generates System Reset Output from Schmitt Trigger Input
- Provides Local Ready and MULTIBUS™ Ready Synchronization
- Capable of Clock Synchronization with other 8284’s
- Industrial Temperature Range -40°C to +85°C

The 8284 is a single-chip clock generator designed to provide clock signals for the 8086, 8088 and 8089 and other peripherals. It also contains READY logic for operation with two MULTIBUS™ systems and provides the processors required READY synchronization and timing. Reset logic with hysteresis and synchronization is also provided.

---

**8086 Pin Diagram**

**18284 Pin Configuration**

**18284 Block Diagram**
**8288**
**BUS CONTROLLER**
**FOR 8086, 8088, 8089 PROCESSORS**

- Bipolar Drive Capability
- Provides Advanced Commands
- Provides Wide Flexibility in System Configurations
- 3-State Command Output Drivers
- Configurable for Use with an I/O Bus
- Facilitates Interface to One or Two Multi-Master Buses

The Intel® 8288 Bus Controller is a 26-pin bipolar component for use with medium-to-large 8086 processing systems. The bus controller provides continuous and controlled signal generation as well as bipolar bus drive capability while optimizing system performance.

A stepping action on the bus controller configures it for use with a multi-master system bus and separate I/O bus.

---

**2142**
**1024 X 4 BIT STATIC RAM**

- High Density 20 Pin Package
- Access Time Selections from 200-450ns
- Identical Cycle and Access Times
- Low Operating Power Dissipation
  - 1mW/Bit Typical
- Single +5V Supply
- No Clock or Timing Strobe Required
- Completely Static Memory
- Directly TTL Compatible: All Inputs and Outputs
- Common Data Input and Output Using Three-State Outputs

The Intel® 2142 is a 1024-bit static Random Access Memory organized as 1024 words by 4 bits using Nonintegrated Silicon technology. It uses fully CMOS enabling remarkably low static power dissipation — 1mW/Bit typical — and fast access times. The signal is a product of noninverted and has the same polarity as the output data. Common input/output pins are important for memory applications where high performance, low cost, and plug-in interchangeability are required.

The 2142 is pin-compatible with the industry-standard 20-pin AT2142. The 2142 is fabricated with Intel's Nonintegrated Silicon technology, a technology providing excellent performance without the use of low-cost plastic packages.
2716
16K (2K x 8) UV ERASABLE PROM

- Fast Access Time:
  - 550 ns Max. 2716-1
  - 390 ns Max. 2716-2
  - 450 ns Max. 2716
  - 650 ns Max. 2716-6

- Single +5V Power Supply

- Low Power Dissipation:
  - 525 mW Max. Active Power
  - 132 mW Max. Standby Power

- Completely Static

The Intel® 2716 is a 16,384-bit ultraviolet erasable and electrically programmable read-only memory (EPROM). The 2716 operates from a single +5V supply, has a static memory mode, and features fast single address location programming. It makes designing with EPROMs faster, easier, and more economical.

The 2716, when used with a single 5-V supply and with an access time up to 550 ns, is ideal for use in the newer high-performance 5-V microprocessors such as Intel's 8088 and 9300. The 2716 is also the first EPROM with a static memory mode which eliminates the power dissipation without increasing access time. The maximum active power dissipation is 525 mW while the maximum standby power dissipation is only 132 mW, a 75% saving.

The 2716 has the simplest and fastest method yet devised for programming EPROMs — single word TTL level programming. No need for high-voltage pulsing because all programming controls are handled by TTL signals. Program any location at any time—either irreversibly, sequentially, or in random, with the 2716's single address location programming. Total programming time for all 16,384 bits is only 108 seconds.

MODE SELECTION

PIN NAMES
8355/8355-2
16,384-BIT ROM WITH I/O

- 2048 Words x 8 Bits
- Single 5V Power Supply
- Directly compatible with 8085A and 8086 Microprocessors
- 2 General Purpose 8-Bit I/O Ports
- 40-Pin DIP

The Intel 8355 is a ROM and I/O chip to be used in the 8085A and 8086 microprocessor systems. The ROM portion is organized as 2048 words by 8 bits. It has a maximum access time of 200 ns to parallel use with the 8085A CPU. The I/O portion consists of 2 general purpose I/O ports. Each I/O port has 8 bit lines and each I/O port line is individually programmable as input or output.

The 8355-2 has a 300ns access time for compatibility with the 8085A-2 and high speed 6 MHz 8086 microprocessors.

**Pin Configuration**

<table>
<thead>
<tr>
<th>Pin</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CE</td>
<td>Chip Enable.</td>
</tr>
<tr>
<td>A0-A15</td>
<td>Address inputs.</td>
</tr>
<tr>
<td>R/W</td>
<td>Read/Write.</td>
</tr>
<tr>
<td>RD</td>
<td>Data output.</td>
</tr>
<tr>
<td>RS1-RS0</td>
<td>Chip Select.</td>
</tr>
</tbody>
</table>

**Block Diagram**

- **ALE** (Address Latch Enable): High, A0, A1, A2, CE, and CE' enter address latch. The inputs AD, ADE, A16, A17, CE, CE', are latched in at the trailing edge of ALE.
- **Bidirectional Address/Data bus**: The lower 8 bits of the ROM or I/O address are applied to the bus tree when ALE is high. During the I/O cycle, Port A or B are selected based on the high order bits of the ROM address. They do not affect I/O operations.

- **CE**/**CE'** (Input): Chip Enable is active low and CE' is active high. The 8355 can be accessed only when **CE** Chip Enable is active at the time the ALE signal becomes high. If either Chip Enable input is not active, the A0-A15 and READY outputs will be in a high impedance state.

- **RD** (Input): When the latched I/O is high, the output data comes from an I/O port. If it is low, the output data comes from the ROM.

- **RE** (Input): If the latched Chip Enable is active low on the selected I/O port, the output data comes from the ROM.

- **READY** (Output): The READY signal is always high.

- **Power Supply**: +5V for logic and 10V for the output drivers.

- **Addressable I/O Port**: Port A or B selected, active Chip Enable, and AD low are required to write operations. When the Chip Enable is active low, the output data comes from the selected I/O port.

- **Ground Reference**: 0V for power supply.
8755A/8755A-2
16,384-BIT EPROM WITH I/O

- 2048 Words x 8 Bits
- Single 5V Power Supply (Vcc)
- Each I/O Port Line Individually Programmable as Input or Output
- Multiplexed Address and Data Bus
- UV, Erasable and Electrically Reprogrammable
- Internal Address Latch

The Intel 8755A is an erasable and electrically programmable ROM (EPROM) and I/O chip to be used in the 8086A and 8088 microprocessor systems. This EPROM portion is organized as 2048 words by 8 bits. It has a maximum access time of 400 ns to permit use with a 4 MHz bus in an 8085A CPU.

The I/O portion consists of 2 general purpose I/O ports. Each I/O port has 8 port lines, and each I/O port line is individually programmable as input or output.

The 8755A-2 is a high-speed select version of the 8755A compatible with the 5 MHz 8085A-2 and the 5 MHz 8086A.

PIN CONFIGURATION

ECL BLOCK DIAGRAM

8755A FUNCTIONAL PIN DEFINITION

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALE</td>
<td>When Address Latch Enable goes high, ALE-0, IOMA, As-0, CIE, and CE are all latched at the trailing edge of ALE.</td>
</tr>
<tr>
<td>AD-2</td>
<td>Bidirectional Address/Data bus. The lower 8-bits of the PROM or I/O address are applied to the bus lines when ALE is high.</td>
</tr>
<tr>
<td>As-10</td>
<td>Input-Output.</td>
</tr>
<tr>
<td>CE-0</td>
<td>Chip Enable inputs. CE0 is active low and CE1 is active high. The 8755A can be guaranteed only when both Chip Enable inputs are high.</td>
</tr>
<tr>
<td>PROG/CE-0</td>
<td>Is also used as a programming pin (See section on programming.)</td>
</tr>
<tr>
<td>IG-0</td>
<td>If the latch of IOR is high when RD is low, the output data comes from the I/O port. If the latch of IOR is low, the output data is not affected.</td>
</tr>
<tr>
<td>RS</td>
<td>If the latch of Chip Enable is active, the RD output buffers are enabled and output either the selected PROM location or I/O port. When both RD and IOR are active, the RD output buffers are disabled.</td>
</tr>
<tr>
<td>IOW-0</td>
<td>Low on IOW causes the output port to be driven by the latch value of the selected I/O port.</td>
</tr>
<tr>
<td>CLK-0</td>
<td>The CLK is used to force the READY into its high impedance state after it has been forced low by CE0, CE1, or ALE high.</td>
</tr>
</tbody>
</table>

READY-0 | is a 3-state output controlled by CE0, CE1, ALE and CLK. READY is forced low when the Chip Enables are active during the time ALE is high, and remains low until the rising edge of the next CKL. (See Figure 6.) |

PA-1 | These are general purpose I/O pins. |

Port A is selected for input/output when the Chip Enables are active and IOH is low and ALE was previously latched from A0, A16. |

Address latch is enabled with either A0, A16, and active Chip Enables, or A0, A16, low, or 1016 high. RD low, active Chip Enables, and A0, A16 low. |

RESET-0 | When the Chip Enables are active, a low on IOR will output the selected I/O port onto the A0 bus. IOR low performs the same function as the combination of IG0 high and RD low. When IOR is not used as a system, IOR should be tied to Vcc (15V). |

VCC-0 | 15 volt supply. |

VSS-0 | Ground Reference. |

VP0-0 | Vpp is the programming voltage, and must be tied to +5V when the 8755A is being used. |

For programming a high voltage is supplied with Vpp=5V typical. (See section on programming.)
2048 BIT STATIC MOS RAM WITH I/O PORTS AND TIMER

- 256 Word x 8 Bits
- Single +5V Power Supply
- Completely Static Operation
- Internal Address Latch
- 2 Programmable 8 Bit I/O Ports
- 1 Programmable 6-Bit I/O Port
- Programmable 14-Bit Counter/Timer
- Compatible with 8086A and 8088 CPU
- Multiplexed Address and Data Bus
- 40 Pin DIP

The 8155 and 82156 are RAM and I/O chips to be used in the 8086A and 8088 microprocessor systems. The RAM portion is designed with 2048 static cells organized as 256 x 8. They have a maximum access time of 400 ns and a minimum access time of 330 ns for use with the 8086A-2 and the fast speed 8086 D8088 CPU.

The I/O portion consists of three general purpose I/O ports. One of the three ports can be programmed to be a special pins, thus allowing the other two ports to be operated in handshake mode.

A 14-bit programmable counter/timer is also included on chip to provide either a square wave or terminal count pulse for the CPU system depending on timer mode.

**PIN CONFIGURATION**

**BLOCK DIAGRAM**

**8155/8156 PIN FUNCTIONS**

<table>
<thead>
<tr>
<th>Pin</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESET</td>
<td>Pulse provided by the 50mA to initialize the system connect to 50mA. If the line is less than 50mA, it will not initialize the I/O pins. The RESET line should normally be kept high to avoid unintended initialization.</td>
</tr>
<tr>
<td>A0 [5]</td>
<td>Input/output. Data lines that interface with the CPU. Address/Data lines.</td>
</tr>
<tr>
<td>A0 [4]</td>
<td>Input. Address latch input. A0 is the address latch input. A0 is the falling edge of ALE.</td>
</tr>
<tr>
<td>A0 [3]</td>
<td>Input/output. Address latch input. A0 is the address latch input. A0 is the falling edge of ALE.</td>
</tr>
<tr>
<td>A0 [2]</td>
<td>Input/output. Address latch input. A0 is the address latch input. A0 is the falling edge of ALE.</td>
</tr>
<tr>
<td>A0 [1]</td>
<td>Input/output. Address latch input. A0 is the address latch input. A0 is the falling edge of ALE.</td>
</tr>
<tr>
<td>A0 [0]</td>
<td>Input/output. Address latch input. A0 is the address latch input. A0 is the falling edge of ALE.</td>
</tr>
</tbody>
</table>

**Address Latch Enable (ALE)**

- Address Latch Enable: This signal is active high and is used to address the A0-A15 lines and the data on the Chip Enable and I/O lines. ALE is the falling edge of ALE.

- Selects memory if low and I/O and command/status registers if high.

- Three 8 pins are general purpose I/O pins. The output direction is selected by programming the command register.

- These 8 pins are general purpose I/O pins. The output direction is selected by programming the command register.

- These 8 pins can function as either input, output, or in parallel. Signals for RA and RB are issued from the command register.

- When RA is used as a command/status register, it will provide the following:

  - PC: A INTR (Port A Interrupt) / A DTR (Port A Data Terminal Ready)
  - PC: B INTR (Port B Interrupt) / B DTR (Port B Data Terminal Ready)
  - PC: A STB (Port A Slave Select) / PC: B STB (Port B Slave Select)
  - PC: EEP (Port B Data Terminal Ready)
  - PC: B IN (Port B Input) / PC: B OUT (Port B Output)
  - PC: B STB (Port B Slave Select)

- The output of the counter/timer is input to the counter/timer. The output can be either a square wave or a pulse depending on the timer mode.

- VCC: +5 volt supply.

- GND: Ground Reference.
8255A/8255A-5
PROGRAMMABLE PERIPHERAL INTERFACE

- MCS-86™ Compatible 8255A-5
- 24 Programmable I/O Pins
- Completely TTL Compatible
- Fully Compatible with Intel® Microprocessor Families
- Improved Timing Characteristics
- Improved DC Driving Capability

The Intel® 8255A is a general purpose programmable I/O device designed for use with Intel® microprocessors. It has 24 I/O pins which may be individually programmed in 2 groups of 12 and used in 3 major modes of operation: in the first mode (MODE 0), each group of 12 I/O pins may be programmed in sets of 4 to be input or output. In MODE 1, the second mode, each group may be programmed to have 8 lines of input or output. Of the remaining 4 pins, 3 are used for handshake and interrupt control signals. The third mode of operation (MODE 2) is a bidirectional bus mode which uses 8 lines for a bidirectional bus, and 5 lines, borrowing one from the other group, for handshaking.

PIN CONFIGURATION

8255A BLOCK DIAGRAM

The display portion provides a scanned display interface for LED, incandescent, and other popular display technologies. Both numeric and alphanumeric segment displays may be used as well as simple indicators. The 8279 has fix display RAM which can be organized into dual fields. The RAM can be loaded or interrogated by the CPU. Both read, write, and display controllers can be done with auto-increment of the display RAM address.
## HARDWARE DESCRIPTION

The 8279 is packaged in a 40-pin DIP. The following is a functional description of each pin.

### Table 1. Pin Description

<table>
<thead>
<tr>
<th>Syencoding</th>
<th>Pin No.</th>
<th>Name and Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0–D7</td>
<td>1-12</td>
<td>BI-DIRECTIONAL DATA BUS: All data and commands between the CPU and the 8279 are transmitted on these lines.</td>
</tr>
<tr>
<td>CLK</td>
<td>3</td>
<td>CLODGE: Clock from system used to generate internal timing.</td>
</tr>
</tbody>
</table>
| RESET      | 9      | RESET: A high signal on this pin resets the 8279. After being reset the 8279 is placed in the following mode:
  1) 16-bit character display—left entry.
  2) Encoded scan keyboard—2 key backlog.
  Along with this the program clock prescaler is set to 31. |
| CS         | 22     | CHIP SELECT: A low on this pin enables the interface functions to receive or transmit. |
| Ao         | 21     | BUFFER ADDRESS: A high on this pin indicates the signals in or out are interpreted as a command or status. A low indicates that they are data. |
| RD, WR     | 10-11  | INPUT/OUTPUT READ AND WRITE: These signals enable the data buffers to either send data to the external bus or receive it from the external bus. |
| IRQ        | 4      | INTERRUPT REQUEST: In a keyboard mode, the interrupt line is high when there is data in the FIFO/Sensor RAM. The interrupt line goes low with each FIFO/Sensor RAM read and returns high if there is still information in the RAM. In a sensor mode, the interrupt line goes high whenever a change in a sensor is detected. |
| Vcc, Vcc   | 20, 40 | GROUND AND POWER SUPPLY PINS.                                                      |
| SL0–SL3    | 32-35  | SCAN LINES: Scan lines which are used to scan the key switch or sensor matrix and the display digits. These lines can be either encoded (1 of 16) or decoded (1 of 4). |
| RL0–RL7    | 35, 39 | RETURN LINE: Return line inputs which are connected to the scan lines through the ways or sensor switches. They have active internal pullups to keep them high until a switch closure pulls them low. They also serve as an l-bit input to the strobed input mode. |
| SHFT      | 26     | SHIFT: The shift input status is stored along with the key position on key closure, in the Scan RAM. It has an active internal pullup to keep it high until a switch closure pulls it low. |
| CTL/STD    | 37     | CONTROL/STROBED INPUT MODE: For keyboard modes this line is used as a control input and strobed like status on a key closure. The line is also the strobe line that enters the data into the FIFO in the Strobed input mode. Rising edges of this active internal pullup to keep it high until a switch closure pulls it low. |
| OUT A0–OUT A7 | 27-34 | OUTPUTS: These 8 bits are output lines for the 16 x 4 display refresh registers. The data from these outputs is synchronized to the scan lines (SL0–SL3) for multiplexed digit displays. The two 4-bit ports may be scanned independently. These two ports may also be considered as one 8-bit port. |
| OUT B0–OUT B7 | 31-38 |                                                                                          |
| BD         | 23     | BLANK DISPLAY: The output is used to blank the display during digit switching or by a display blanking command. |

### APPENDIX C

#### Absolute Addressing
The specific identification number (address) permanently assigned to a storage location, device, or register by the machine designer. Used to locate information and assist in circuit fault diagnosis.

#### Accumulator
Used for storing the result after most ALU operations; 8 bits long for an 8-bit microprocessor.

#### Address
A unique identification number (or locator) of some source or destination of data. That part of an instruction which specifies the register or memory location of an operand involved in the instruction.

#### Addressing Mode
The manner in which a microprocessor determines the operand and destination addresses in an instruction cycle.

#### Address Register
A register used to store the address (label for a memory location) of data being fetched or stored; a sequence of instructions to be executed, or the location to which control will be transferred.

#### Address Space
The number of storage locations that can be located by the addressing technique used by the CPU. The addressing range is determined by the number of bits capable of being held in the address register (that is, 2^n combinations).

#### American Standard Code for Information Interchange (ASCII)
An 8-bit code commonly used with microprocessors for representing alphanumeric codes.

#### Analog-to-Digital (A/D) Converter
Transforms an analog voltage into its digital equivalent.

#### Architecture
The organizational structure or hardware configuration of a computer system.

#### Arithmetic and Logic Unit (ALU)
A digital circuit which performs arithmetic and logic operations on two n-bit digital words.

#### Assembler
A program that translates an assembly language program into a machine language program.
Assembly Language: A type of microprocessor programming language that uses a semi-English language statement.

Asynchronous Operation: The execution of a sequence of steps such that each step is initiated upon completion of the previous step. For bus structures this implies a timing protocol that uses no clock and has no period; hence system operation proceeds at a rate governed by the time-constants of the enabled circuitry.

Asynchronous Serial Data Transmission: The transmitting device does not need to be synchronized with the receiving device.

Autodecrement Addressing Mode: The contents of the specified microprocessor register are first decremented by $K$ (1 for byte, 2 for 16-bit, and 4 for 32-bit) and then the resulting value is used as the address of the operand.

Autoincrement Addressing Mode: The contents of a specified microprocessor register are used as the address of the operand first and then the register contents are automatically incremented by $K$ (1 for byte, 2 for 16-bit, and 4 for 32-bit).

Bandwidth: Bandwidth of a bus or memory is a measure of communications throughput and can be represented as the product of the maximum number of transactions per second and number of data bits per transaction.

Barrel Shifter: A specially configured shift register that allows multiple shifting binary-encoded operands in one clock cycle.

Base Address: An address that is used to convert all relative addresses in a program to absolute (machine) addresses.

Base Page Addressing: This instruction uses two bytes: the first byte is the op code, and the second byte is the low-order address byte. The high-order address byte is assumed to be the base page number.

Baud Rate: Rate of data transmission in bits per second.

Binary-Coded Decimal (BCD): The representation of 10 decimal digits, 0 through 9, by their corresponding 4-bit binary numbers.

Bit: An abbreviation for a binary digit. A unit of information equal to one binary decision or one of two possible states (one or zero, on or off, true or false) and represents the smallest piece of information in a binary notation system.

Bit-Slice Microprocessor: Divides the elements of a central processing unit (ALU, registers, and control unit) into several ICs. The registers and ALU are usually contained in a single chip. These microprocessors can be cascaded to produce microprocessors of variable word lengths such as 8, 12, 16, 32. The control unit of a bit-slice microprocessor is typically microprogrammed.

Block Transfer DMA: A peripheral device requests the DMA transfer via the DMA request line, which is connected directly or through a DMA controller chip to the microprocessor. The DMA controller chip completes the DMA transfer and transfers the control of the bus to the microprocessor.

Branch: A computer instruction that marks a decision point in the program or that point at which the path followed for the execution of instructions diverges. The branch instruction allows the computer to skip or jump out of program sequence to a designated instruction.

Breakpoint: Allows the user to execute the section of a program until one of the breakpoint conditions is met. It is then halted. The designer may then single step or examine memory and registers. Typically breakpoint conditions are program counter address or data references.

Buffer: A temporary memory storage device designed to compensate for the different data rates between a transmitting device and a receiving device (for example, between a CPU and a peripheral).

Bus: A collection of parallel unbroken electrical signal lines that interconnect or link computer modules. The typical microcomputer interface includes separate buses for address, data, control, and power functions.

Bus Arbitration: Bus operation protocols that guarantee conflict-free access to a bus. Arbitration is the process of selecting one respondent from a collection of several candidates that concurrently request service.
Bus Cycle: The period of time in which a CPU carries out all the necessary bus communications to implement a standard operation.

Byte: An 8-bit word.

Cache Memory: An ultra-high-speed, directly accessible, relatively small semiconductor memory block used to store data/instructions that the computer may need in the immediate future. Increases system bandwidth by reducing the number of external memory fetches required by the processor.

Cathode Ray Tube (CRT): Evacuated glass tube with a fluorescent coating on the inner side of the screen.

Central Processing Unit (CPU): The portion of a computer containing the ALU, register section, and control unit.

Clock: Timing signals providing synchronization among the various components in a microcomputer system.

Code: A system of symbols or set of rules for the representation of data in a digital computer. Some examples include binary, BCD, and ASCII.

Compiler: A software program which translates the source code written in a high-level programming language into machine language that is understandable to the processor.

Complementary Metal Oxide Semiconductor (CMOS): Provides low power density and high noise immunity.

Concurrency: The occurrence of one or more operations at a time (see Parallel Operation).

Conditional Branching: Conditional branch instructions are used to change the order of execution of a program based on the conditions set by the status flags.

Condition Code Register: Contains information about the condition of functional units or peripheral devices. It is contained within the supervisor model's status register.

Control Register: A register which contains the address of the next instruction in the sequence of operations (that is, a program counter).

Control Store: Used to contain microcode (usually in ROM) in order to provide for microprogrammed “firmware” control functions. An integral part of a microprogrammed system controller.

Control Unit: Part of the microprocessor; its purpose is to read and decode instructions from the memory.

Controller/Sequencer: The hardware circuits which provide signals to carry out selection and retrieval of instructions from storage in sequence, interpret them, and initiate the required operation. The system functions may be implemented by hardware control, firmware control, or software control.

Coprocessor: A specialized microprocessor that performs specific functions independently from the CPU to speed up overall operations.

CPU Space: Protected memory space addressable only by the CPU itself; it is used for a processor's internal functions or vectorized exception processing.

CRT Controller: Provides all logic functions for interfacing the microprocessor to a CRT.

Cycle Stealing DMA: The DMA controller transfers a byte of data between the microprocessor and a peripheral device by stealing a clock cycle of the microprocessor.

Daisy Chain Interrupt: Priorities of interrupting devices are defined by connecting them in a daisy chain.

Data: Basic elements of information represented in binary form (that is, digits consisting of bits) that can be processed or produced by a computer. Data represent any group of operands made up of numbers, letters, or symbols denoting any condition, value, or state. Current typical computer operand sizes include: a word, which contains 2 bytes or 16 bits; a long word, which contains 4 bytes or 32 bits; a quad word, which contains 8 bytes or 64 bits.
Data Counter (DC): Also known as Memory Address Register (MAR). Stores the address of data; typically, 16 bits long for 8-bit microprocessors.

Data Register: A register used to temporarily hold operational data being sent to and from a coprocessor peripheral device.

Debugger: A program that executes and debugs the object program generated by the assembler or compiler. The debugger provides a single stepping, breakpoints, and program tracing.

Decoder: A device capable of deciphering encoded signal. It interprets input instructions and initiates the appropriate control operations as a result.

Direct Address: An address that specifies the location in memory of an operand.

Direct Memory Access (DMA): A type of input/output technique in which data can be transferred between the microcomputer memory and external devices without the microprocessor's involvement.

Dynamic RAM: Stores data in capacitors and, therefore, must be refreshed; uses refresh circuitry.

EAROM (Electrically Alterable Read-Only Memory): Can be programmed without removing the memory from its sockets. This memory is also called read-mostly memory since it has much slower write times than read times.

Editor: A program that produces an error-free source program, written in assembly or high-level languages.

Effective Address: The final address used to carry out an instruction.

Emulator: A hardware device that allows a computer system to emulate (that is, mimic the procedures or protocols) of another computer system.

Encode: To apply the rules governing a specific code. For example, the selection of which hardware devices to enable during an operation can occur automatically by encoding individual device identifications into the instructions themselves. Hence, to encode is to convert data from its natural form into a machine-readable code usable to the computer.

EPROM (Erasable Programmable Read-Only Memory): Can be programmed and erased using ultraviolet light. The chip must be removed from the microcomputer system for programming.

Exception Processing: The CPU processing state associated with interrupts, trap instructions, tracing, and other exceptional conditions, whether they are initiated internally or externally.

Extended Binary-Coded Decimal Interchange Code (EBCDIC): An 8-bit code commonly used with microprocessors for representing character codes.

Firmware: Permanently stored, unalterable program instructions contained in the ROM section of a computer's memory (see Control Store).

Flag(s): An indicator, often a single bit, to indicate some condition or mark information for processing attention. Sometimes referred to as a tag, mark, switch, or sentinel.

Flowchart: Representation of a program in a schematic form. It is convenient to flowchart a problem before writing the actual programs.

Global Bus: A computer bus system that is available to and shared by a number of processors connected together in a multiprocessor system environment.

Handshaking: Exchange of control signals between the microprocessor and an external device.

Hardware: The physical electronic circuits and electromechanical devices that make up the computer system.

HCMOS: Low-power NMOS.

Hexadecimal Number System: Base-16 number system.

Hierarchical Memory: A memory organization or informational structure in which functional relationships are associated with different levels.

High-Level Language: A type of programming language that uses a more understandable human-oriented language.
Glossary

HMOS: High-performance MOS reduces the channel length of the NMOS transistor and provides increased density and speed in LSI and VLSI circuits.

Immediate Address: An address that is used as an operand by the instruction itself.

Implied Address: An address not specified, but contained implicitly in the instruction.

In-Circuit Emulation: The most powerful hardware debugging technique, especially valuable when hardware and software are being debugged simultaneously.

Index: A symbol used to identify or place a particular quantity in an array (list) of similar quantities. Also, an ordered list of references to the contents of a larger body of data such as a file or record.

Indexed Addressing: Typically uses 3 bytes: the first byte for the op code and the next 2 bytes for the 16-bit address. The effective address of the instruction is determined by the sum of the 16-bit address and the contents of the index register.

Index Register: A register used to hold a value used in indexing data, such as when a value is used in indexed addressing to increment a base address contained within an instruction.

Indirect Address: An address that informs the CPU of the location of a direct address or another indirect address.

Input/Output Processor (IOP): Performs most of the I/O functions for the 16- and 32-bit microprocessors and unburdens the microprocessors of these I/O functions.

Instruction: A program statement (step) that causes the computer to carry out an operation, and specifies the values of locations of all operands.

Instruction Cycle: The sequence of operations that a microprocessor has to carry out while executing an instruction.

Instruction Register (IR): A register storing instructions; typically 8 bits long for an 8-bit microprocessor.

Instruction Set: Describes a computer's capability by listing all the instructions (available in machine code) that the computer can execute.

Interleaved DMA: Using this technique, the DMA controller takes over the system bus when the microprocessor is not using it.

Internal Interrupt: Activated internally by exceptional conditions such as overflow and division by zero.

Interpreter: A program that executes a set of machine language instructions in response to each high-level statement in order to carry out the function.

Interrupt I/O: An external device can force the microcomputer system to stop executing the current program temporarily so that it can execute another program known as the interrupt service routine.

Interrupt: A temporary break in a sequence of a program, initiated externally, causing control to pass to a routine, which performs some action while the program is stopped.

Invisible: See Transparent.

I/O (Input/Output): Describes that portion of a computer system that exchanges data between the computer system and the external world, or the data itself.

I/O Port: A module that contains control logic and data storage used to connect a computer to external peripherals.

Keyboard: Has a number of pushbutton-type switches configured in a matrix form (rows x columns).

Keybounce: When a mechanical switch opens or closes, it bounces violently for a small period of time (about 10-20 ms) before settling down.

Large-Scale Integration (LSI): An LSI chip contains more than 100 gates.

Linkage Editors: Connect the individual programs together which are assembled or compiled independently.
Linked Programming: The process of joining a subprogram with a main program or joining two separate programs together to form a single program.

Local Area Network: A collection of devices and communication channels that connect a group of computers and peripheral devices together so that they can communicate with each other.

Logic Analyzer: A hardware development aid for microprocessor-based design; gathers data on the fly and displays it.

Logical Address Space: All storage locations with a CPU's addressing range.

Loops: A programming control structure where a sequence of computer instructions are executed repeatedly (looped) until a terminating condition (result) is satisfied.

Machine Code: The binary code (composed of bit patterns) that a computer can sense, read, interpret, recognize, and manipulate.

Machine Language: A type of microprocessor programming language that uses binary or hexadecimal numbers.

Macroinstruction: Commonly known as an instruction; initiates execution of a complete microprogram.

Macroprogram: The assembly language program.

Mask: A pattern of bits used to specify (or mask) which bits of another bit pattern are to be operated on and which bits are to be ignored or "masked" out.

Mask ROM: Programmed by a masking operation performed on the chip during the manufacturing process; its contents cannot be changed by the user.

Maskable Interrupt: Can be enabled or disabled by executing typically the instructions EI and DI, respectively. If the microprocessor's interrupt is disabled, the microprocessor ignores the interrupt.

Memory: Any storage device which can accept, retain, and read back data. Usually refers to a computer subsystem of internal RAM- or ROM-based storage devices.

Memory Access Time: Average time taken to read a unit of information from the memory.

Memory Address Register (MAR): Also known as the Data Counter (DC). Stores the address of the data; typically 16 bits long for 8-bit microprocessors.

Memory Cycle Time: Average time lapse between two successive read operations.

Memory Map: A representation of the physical location of software within a computer's addressable main storage.

Memory-Mapped I/O: A microprocessor communications methodology (addressing scheme) where the data, address, and control buses extend throughout the system, with every connected device treated as if it were a memory location with a specific address. Manipulation of I/O data occurs in "interface registers" (as opposed to memory locations); hence there are not input (read) or output (write) instructions used in memory-mapped I/O.

Microcode: A set of "subcommands" or "pseudocommands" built into the hardware (usually stored in ROM) of a computer (that is, firmware) to handle the decoding and execution of higher-level instructions such as arithmetic operation.

Microcomputer: Consists of a microprocessor, a memory unit, and an input/output unit.

Microinstruction: Most microprocessors have an internal memory called control memory. This memory is used to store a number of codes called microinstructions. These microinstructions are combined to design the instruction set of the microprocessor.

Micro/Nano ROM: See Hierarchical Memory.
Microprocessor: The Central Processing Unit (CPU) of a microcomputer.

Microprocessor Development System: A tool for designing and debugging both hardware and software for microcomputer-based systems.

Microprocessor-Halt DAM: Data transfer is performed between the microprocessor and a peripheral device either by completely stopping the microprocessor or by a technique called cycle stealing.

Microprogramming: The microprocessor can use microprogramming to design the instruction set. Each instruction in the instruction register initiates execution of a microprogram in the control unit to perform the operation required by the instruction.

Module: (1) Any single hardware arrangement (device or component) within a computer system. (2) Any software, routine, or subroutine.

Monitor: Consists of a number of subroutines grouped together to provide “intelligence” to a microcomputer system. This intelligence gives the microcomputer system the capabilities for debugging a user program, system design, and displays.

Multiplexer: A hardware device which allows a CPU to be physically connected to a number of communication channels to receive or transmit data.

Multiprocessing: The process of executing two or more programs in parallel, handled by multiple processors all under common control. Typically each processor will be assigned specific processing tasks.

Multitasking: Operating system software that permits more than one program to run on a single CPU. Even though each program is given a small time slice in which to execute, the user has the impression that all tasks (different programs) are executing at the same time.

Multitasking: Describes a computer operating system that permits a number of users to access the system on a time-sharing basis.

Nested Subroutine: A commonly used programming technique that includes one subroutine entirely embedded within the "scope" of another subroutine.

Nibble: A 4-bit word.

NMOS: Less dense and faster in comparison to PMOS. Most 8-bit microprocessors and some 16-bit microprocessors are fabricated using this technology.

Noncontiguous: Noncontiguous in nature. Refers to breaks in the linear sequential flow of any information structure.

Nonmaskable Interrupt: Occurrence of this type of interrupt cannot be ignored by the microprocessor, even though the interrupt capability of the microprocessor is disabled. Its effect cannot be disabled by instruction.

Non-multiplexed: A non-multiplexed system indicates a direct single communication channel (that is, electrical wires) connection to the CPU.

Object Code: The binary (machine) code into which a source program is translated by a compiler, assembler, or interpreter.

Octal Number System: Base-8 number system.

One-Pass Assembler: This assembler goes through the assembly language program once and translates the assembly language program into a machine language program. This assembler has the problem of defining forward references. See Two-Pass Assembler.


Operand: A datum or information item involved in an operation from which the result is obtained as a consequence of defined actions (that is, data which is operated on by an instruction). Various operand types contain information, such as source address, destination address, or immediate data.

Operating System: Consists of a number of program modules to provide resource management. Typical resources include microprocessors, disks, and printers.

Operation: (1) Means by which a result is obtained from an operand(s). (2) An action defined by a single instruction or single logical element.
Page: Some microprocessors, such as the Motorola 6800 and the MOS 6502, divide the 65,536 memory locations into 256 blocks. Each of these blocks is called a page and contains 256 addresses.

Parallel Operation: Any operation carried out simultaneously with a related operation.

Parallel Transmission: Each bit of binary data is transmitted over a separate wire.

Parity: The number of 1's in a word is odd for odd parity and even for even parity.

Peripheral: An I/O device capable of being operated under the control of a CPU through communication channels. Examples include disk drives, keyboards, CRTs, printers, modems, etc.

Peripheral: An I/O device capable of being operated under the control of a CPU through communication channels. Examples include disk drives, keyboards, CRTs, printers, modems, etc.

Personal Computer: Low-cost, affordable computer used by an individual or a small group for video games, daily schedules, and industrial applications.

Physical Address Space: Includes all internal storage.

Pipeline: A technique that allows a computer processing operation to be broken down into several steps (dictated by the number of pipeline levels or stages) so that the individual step outputs can be handled by the computer in parallel. Often used to fetch the processor's next instruction while executing the current instruction, which considerably speeds up the overall operation of the computer.

Pointer: A storage location (usually a register within a CPU) that contains the address of (or points to) a required item of data or subroutine.

Polling Interrupt: A software approach for determining the source of interrupt in a multiple interrupt system.

POP Operation: Reading from the top or bottom of the stack.

Port: An access point for a computer through which communication data may be passed to peripheral devices.

Primary Memory Store: That memory storage which is considered main, integral, or internal to the computer system. It is that storage which is physically most closely associated with the CPU and is directly controlled by it.

Primitives: A basic or fundamental unit; often refers to the lowest level of machine instruction or the lowest unit of programming language instruction.

Privileged Instructions: An instruction which is reserved for use by a computer's operating system, which will determine the range of system resources that the user is allowed to exploit.

Processor Memory: A set of microprocessor registers for holding temporary results when a computation is in progress.

Program: A self-contained sequence of computer software instructions (source code) that, when converted into machine code, directs the computer to perform specific operations for the purpose of accomplishing some processing task.

Program Counter (PC): See Control Register.

Programmed I/O: The microprocessor executes a program to perform all data transfers between the microcomputer system and external devices.

PROM (Programmable Read-Only Memory): Can be programmed by the user by using proper equipment. Once programmed, its contents cannot be altered.

Protocol: A list of data transmission conventions or procedures that encompass the timing, control, formatting, and data representations by which two devices are to communicate. Also known as hardware “handshaking”, which is used to permit asynchronous communication.

Pseudo-Static RAM: Dynamic RAM with internal refresh circuitry.

PUSH Operation: Writing to the top or bottom of the stack.
Random Access Memory (RAM): A memory in which any addressable operand, disk sector, etc. can be read from and written to, and whose input-to-output access time is asynchronous in nature (that is, the access time is dictated by the time delays of the chip's internal circuitry). Internal semiconductor RAM (static or dynamic) is volatile in nature (in other words, information is lost when power is removed). External RAM examples include tape and disk formats.

Read-Only-Memory (ROM): A memory in which any addressable operand, disk sector, etc. can be read from, but not written to, after initial programming. It is an asynchronous device whose access time is dictated by its internal circuit rise delays. Internal semiconductor ROM storage is non-volatile (information is not lost when power is removed).

Real-Time Software: Computer code that allows processes to be performed during the actual time that a related physical I/O action takes place.

Register: A one-word, high-speed memory device usually constructed from flip-flops (electronic switches) that are directly accessible to the processor. It can also refer to a specific location in memory that contains word(s) used during arithmetic, logic, and transfer operations.

Register Indirect: Uses a register pair which contains the address of data.

Relative Address: An address used to designate the position of a memory location in a routine or program.

Rollover: Occurs when more than one key is pushed simultaneously.

Routine: A group of instructions for carrying out a specific processing operation. Usually refers to part of a larger program. A routine and subroutine have essentially the same meaning, but a subroutine could be interpreted as a self-contained routine nested within a routine or program.

Sample and Hold Circuit: When connected to the input of an A/D converter, it keeps a rapidly varying analog signal fixed during the A/D conversion process by storing it in a capacitor.

Scaling: To adjust values to bring them into a range that is acceptable to a computer.

Secondary Memory Storage: An auxiliary data storing device that supplements the main (primary) internal memory of a computer. It is used to hold programs and data that would otherwise exceed the capacity of the main memory. Although it has a much slower access time, secondary storage is less expensive. Common devices include magnetic disk (floppy and hard), cassette tape, and videodisk.

Serial Transmission: Only one line is used to transmit the complete binary data bit by bit.

Single-Chip Microcomputer: Microcomputer (CPU, memory, and input/output) on a chip.

Single-Chip Microprocessor: Microcomputer CPU (microprocessor) on a chip.

Single Step: Allows the use to execute a program one instruction at a time and examine memory and registers.

Software: Programs in a computer.

Source Code: The high-level language code used by a programmer to write computer instructions. This code must be translated to the object (machine) code to be usable to the computer.

Stack: An area of memory reserved to hold information about the status of a computer the instant an interrupt occurs so that the computer can continue processing after the interrupt has been handled. Another common use is in handling the accessing sequence of "nested" subroutines. The stacks are the last in/first out (LIFO) devices that are manipulated by using PUSH or POP instructions.

Stack Pointer: A counter or register used to keep track of the storage and retrieval of each byte of information in the system stack.

Standard I/O: Utilizes a control pin on the microprocessor chip called the IO/BI pin, in order to distinguish between input/output and memory; typically, IN and OUT instructions are used for performing input/output operations.

Static RAM: Stores data in flip-flops; does not need to be refreshed.
Status Register: A register which contains information concerning the activity within the CPU or about the condition of a functional unit or peripheral device.

Subroutine: A program carrying out a particular function and which can be called by another program known as the main program. A subroutine needs to be placed only once in memory and can be called by the main program as many times as the programmer wants.

Supervisor: Provides the procedures or instructions for coordinating the use of system resources and maintaining the flow of operations through a CPU to perform I/O operations.

Supervisor State: When internal CPU system processing operations are conducted at a higher privilege level, it is usually in the supervisor state. An operating system typically executes in the supervisor state to protect the integrity of "basic" system operations from user influences.

Synchronous Operation: Operations that occur at intervals directly related to a clock period. Also, a bus protocol in such data transactions is controlled by a master clock and is completed within a fixed clock period.

Synchronous Serial Data Transmission: Data is transmitted or received based on a clock signal.

Trending: A dynamic diagnostic technique in which a record of internal counter events is made to permit analysis (debugging) of the program's execution.

Transparent: Processes and operations that proceed automatically in hardware while being "invisible" or transparent to the programmer.

Tritate Buffer: Has three output states: logic 0, 1, and a high-impedance state. It is typically enabled by a control signal to provide logic 0 or 1 outputs. This type of buffer can also be disabled by the control signal to place it in a high-impedance state.

2's Complement: The 2's complement of a binary number is obtained by replacing each 0 with a 1 and each 1 with a 0 and adding to the resulting number.

Two-Pass Assembler: This assembler goes through the assembly language program twice. In the first pass, the assembler defines the labels with the addresses. In the second pass, the assembler translates the assembly language program to the machine language. See One-Pass Assembler.

UART (Universal Asynchronous Receiver Transmitter): A chip that provides all the interface functions when a microprocessor transmits or receives data to or from a serial device.

User State: Typical microprocessor operations processing conducted at the user level. The user state is usually at lower privilege level than the supervisor state. This protects basic system operation resources (the operating system).

Vector Base Register (VBR): A register used during interrupts to point to predefined memory locations in CPU space which contain interrupt processing routines.

Vectored Intermittents: A device identification technique in which the highest priority device with a pending interrupt request forces program execution to branch to an interrupt routine to handle exception processing for the device.

Very Large Scale Integration (VLSI): A VLSI chip contains more than 1000 gates.

Virtual Machine: A computer whose hardware and software architecture is specifically designed to support virtual storage techniques. The virtual machine concept is widely used within multiprogramming environments.

Virtual Memory: A memory management operating system technique that allows programs or data to exceed the physical size of the main, internal, directly accessed memory. Program or data segments/pages are swapped from external disk storage as needed. The swapping is invisible (transparent) to the programmer. Therefore the programmer need not be concerned with the actual physical size of internal memory while writing the code.

Words: The bit size of a microprocessor refers to the number of bits that can be processed simultaneously by the basic arithmetic circuits of the microprocessor. A number of bits taken as a group in this manner is called a word.
BIBLIOGRAPHY


Bibliography


Bibliography


Credits

The following material was reprinted by permission of the sources indicated below:

Motorola Corporation: Chapter 1: Figures 1.9a and b; 1.10, 1.11; Chapter 3: Figures 3.5; Figures 5.8, 5.9, 5.12, 5.13, 5.14, 5.16, 5.19 to 5.22, 5.26 to 5.30, 5.32, 5.34, Table 6.1, 6.3, 6.14 to 6.19, table on page 495; Chapter 6: Examples 6.1 and 6.8, all figures, tables, and graphics except the tables on pages 550 to 551. Chapter 7: Examples 7.2, 7.4, 7.6, table on page 678, all figures, tables, and graphics except Figures 7.3, 7.15, 7.21, and 7.27. Chapter 8: all figures, tables, and graphics, Section 8.1, equation on page 791. Chapter 9: Figures 9.20, 9.21. Appendix B: data sheets.

Intel Corporation: Chapter 1: Figures 1.2, 1.3, 1.4, 1.5 to 1.8, 1.11 to 1.13, 1.16, 1.17, 2.21, 2.23a and b, 2.25a to 2.28, Tables 2.9, 2.12, 2.18; Chapter 3: Figures 3.1 to 3.3, 3.5 to 3.16, 3.18, 3.21a and b, 3.22, 3.24, 3.26a and b, 3.27, Tables 3.2, 3.3, 3.4, 3.12, structure on page 231. Chapter 4: Figures 4.1, 4.2, 4.6 to 4.12, Tables 4.1 to 4.3, 4.8 to 4.11, table on page 382; Chapter 9: Figures 9.7, 9.18 to 9.30, tables on page 855. Appendix C: figures and data sheets.

All microprocessors and microcontrollers in Tables 1.1 and 3.1 are courtesy of Intel Corporation.

The 8088 microprocessor referred to in text is the i886", a trademark of Intel Corporation.

Rajgopalan, M., *Microcomputer Theory and Application with the Intel 8086", John Wiley & Sons, Inc., New York, 1986, reprinted by permission of John Wiley & Sons, Inc., Chapter 1: Tables 1.1 to 2.4, Examples 1.1 to 1.4, Figures 2.1 to 2.4, Table 3.1 to 3.3, Figures 4.1 to 4.3, Table 5.1 to 5.3, Figures 6.1 to 6.3, Table 7.1 to 7.3, Figure 8.1 to 8.3, Table 9.1 to 9.3, Table 10.1 to 10.3, Appendix A; Appendix C (excluding figures and data sheets).

1043
CHIP INDEX

Digital Equipment Corporation chips:
VAX11/750, 3
VAX11/780, 3

Hewlett-Packard (HP) Corporation
64000, 48, 52, 54, 925, 957—
986
assembly error codes, 966—968
assembler pseudoinstructions, 968—
966
assembler soft keys, 964—965
description, 965
description, 967
editor commands, 963—966
emulates, 984—986
emuatax soft keys, 969—972
host Pascal, 973
linker commands, 968—969
monitor soft keys, 961—962
passwords, 972—973
terminal mode, 972

Instruction Corporation chips, 1, 2, 48
423, 3
2142, 167, 1005
2513, 10
2716, 117, 157, 866, 1006
2732, 1000
4004, 2
4040, 6
6116, 859, 866
8008, 2
8080, 73, 75
8086, 4, 5, 7, 9, 73—165, 170

8080 compared to, 73
8279 interface to, 815—849, 859—
877
addressing modes, 78—79
architecture, 75—77
arithmetic logic unit, 73—74
assemblers and, 61
black diagram of, 71
CALL, 90
clock frequency, 119
code, 75
control logic, 75
corporation, 46
data sheets, 999—1001
description, 72—75
design, 156—158
design problem, 911—921
direct memory access (DMA), 77,
119, 154—155
enhanced versions, 71, see also
specific chips
flag, 75—77
input/output, 130—156
instruction decoder, 74
instruction execution, 122—129
instruction register, 74
instructions, 75, 79—112, 185
arithmetic, 96—97
logical, 98—99
interruption, 122—129
interrupt, 141—154
introduction of, 2
keyboard/display interface, 839—
845
logic operations, 95
memory addressing, 77—78
GENERAL INDEX

A

Absolute addressing mode, see Direct addressing mode
Access level control (ALC), 678
Access level field, 675
Access status register, 679
Accumulator-based microprocessor, 4, 75, see also specific types
A/D, see Analog-to-digital
Addition, 5, 95
binary, 8
Intel 8086, 176
Motorola 68000, 429—430, 432
Address bus, 3, 4
Address error, 528
Addresses, see also Addressing modes; specific types
data, 45
device, 25
effective, 338
of first instruction, 5
Intel 8086, 226—240
interrupt vector, 258
logical, 18, 19, 317
mapping of, see Mapping most significant bit (MSB) of, 29
of operands, 45
physical, see Physical addresses protection of, 18
Address field, 44, 45
Addressing modes, 45—47, see also Addresses; specific types
alterable, 413
based, 179—181, 309, 338—339
based indexed, 183—184, 309, 339, 346
based scaled indexed, 339, 340
count, 413
data, 45, 412—413
direct, see Direct addressing mode
fully decoding, 12, 14—16
indexed, 7, 181—183, 309, 339
indirect, 244—245, 565—567
inherent, see Implicit addressing mode
input/output, 185
Intel 8085, 75, 76—79
Intel 8086, 176—185, 243—245
Intel 80186, 309
Intel 80286, 315
Intel 80386, 337—340
linear decoding, 12, 14
memory, see Memory addressing mode
Motorola 68000, 401, 403—414, 564
absolute, see Direct addressing mode
immediate, 46, 78, 176, 177, 338, 412—413
implied, see Implicit addressing mode
program counter relative, 412
register direct, 407—409
register indirect, see Register indirect addressing mode
Motorola 68020, 562—572, 718
Motorola 68030, 747—751
Motorola 68811, 718
Motorola 68160, 800—819
| Motorola 68020, 697—702 | Arithmetic instructions, 47, see also specific types extended, 430—431, 432—434 Intel 8085, 95—97 Intel 8086, 176—178 Intel 80186, 299 Intel 80386, 342—344, 356 Motorola 68000, 429—436 Motorola 88100, 807, 809 Arithmetic logic unit (ALU), 4, 5, 7—37 Intel 8085, 73—74 Intel 80386, 380 Motorola 68020, 552 multiple, 8 see also 8 Arithmetic operations, 4, 7—38, see also specific types Intel 8085, 75, 76 multivolt, 5 AS, see Address strobe ASCU characters, 85 Assembler-dependent instructions, 216 Assembler pseudoinstructions Hewlett-Packard 64000, 965—966 Intel 8086, 215—226 Assemblers, 40, 53, 55, 56, 84, see also specific types cross, 60—61 error codes, 965—968 Hewlett-Packard 64000, 964—968 Motorola 68000, 925 one-pass, 59—60 resident, 61 soft keys, 964—965 two-pass, 60 types of, 59—63 Assembly language, 40, 55, 66 Assembly language programming, 18, 42—43, 333 Associative memory, 24 Assume directive, 219 Asynchronous control lines, 464—471 ATC, see Address translation cache AU, see Address unit Automobile engine control, 68, 69 Avionics, 517, 522—524, 643, 649 Auxiliary carry flag, 75, 176 Auxiliary store, 10, 16 B Back link field, 327 Backup store, 10, 16 Backward references, 59 Based shifters, 5, 7, 582 Based addressing mode, 179—181, 309, 338—339 Based indexed addressing mode, 183—184, 309, 339, 340 Based scaled indexed addressing mode, 339, 340 Base pointer (BP), 173, 175 Base register, 174 BASIC, 43, 61 Basic elements of computers, 1, see also specific elements BART, see Block address translation cache BCD, see Binary coded decimal BD, see Blank display BERR, see Bus error BG, see Bus grant BGACK, see Bus grant acknowledge BHE, see Bus high enable Bidirectional bus, 4 Binary addition, 8 Binary coded decimal (BCD), 441—442, 553, 558 Binary subtraction, 8 Bipolar transistors, 11 Bit instructions, 344 Intel 80386, 356—358 Motorola 68000, 440—441 Motorola 68020, 554, 588—592 Motorola 88100, 811, 813 Bit mapping, 895 Bit plane, 895 BIU, see Bus interface unit BKPT, see Breakpoint Black display (BD), 351 Block address translation cache (BART), 759 Blocks, 18 Branching, 5, 47, 211 Built-in self-test (BIST), 235, 253, 260 Buses, 6, 7, 8, 9, 244, 246, 333 Bus controller, 744, 1004 Bus error (BERR) Motorola 68000, 471, 518, 527 Motorola 68020, 525, 627, 644, 703, 727, 738 Motorola 68030, 769, 781 Bus grant (BG), 524, 541 Bus grant acknowledge (BGACK), 524, 533, 541 Bus high enable (BHE), 229, 237, 238, 241, 246, 253 Bus interface unit (BIU), 898
General Index

**Intel 8086, 170—172, 177, 178**
**Intel 80386, 330**
**Motorola 68020, 714**
**Bus request (BR), 524, 525, 541**
**Bus unit (BU), 312, 313**
**Busy flag, 264**
**BX, see Base register**
**Bye set-on condition instructions, 358—359**

**C**
- (programming language), 43, 44
- CA, see Channel attention
- CAAR, see Cache address register
- Cache address register (CAAR), 701
- Cache bit, 22
- Cache control register (CCR), 577—578, 701, 763
- Cache disable (CDIS), 619, 701, 752
- Cache inhibit out (CIOU), 764
- Cache match, 24
- Cache miss, 22, 24
- Cache registers, 558, 701
- Caches, 22—28, see also specific types
- Address translation, see Address translation cache (ATC)
- Block address translation (BATC), 790
- Direct mapped, 699
- Motorola 68020, 577—702
- Motorola 68030, 744, 751—757
- Page address translation (PATC), 790
- RAM and, 24
- CACR, see Cache control register
- CAD, see Computer-aided design
- CAE, see Computer-aided engineering
- CALL, see Current access level
- CALL, 47, 60, 101, 102, 449—452
- Call gates, 322, 325
- CALL, see Call module
- Call module (CALLM), 670—679, 737
- CAM, see Computer-aided manufacturing
- CAP, 314
- Carry flag, 5, 76, 95, 126, 346
- CAS, 618, 680—692
- Cascade mode, 388
- Cathode ray tube (CRT), 25, 50, 692—900, see also Displays; Monitors
- CCR, see Channel control block
- CCR, see Condition code register
- CCM, see Channel command word
- CDS, see Cache disable
- CEB, see Clear entry
- Central processing unit (CPU), 1, 49
- Motorola 68008, 577—578, 701, 752
- Space access and, 553
- Space cycle, 608
- Central processing unit (CPU) registers, 701, 763
- Central processing unit (CPU) root pointer (CRP), 745, 774, 777, 779
- Channel attention (CA), 263
- Channel command word (CCW), 263, 264
- Channel control block (CCB), 263
- Channel select, 263
- Character generation, 895
- Character generator ROMs, 10
- Character-oriented editors, 53, 54
- Check instructions, 582—587
- Chips, see specific chips in Chip Index
- Chip select, 13
- CHS, see Check instructions
- CIOU, see Cache inhibit out
- CIR, see Coprocessor interface registers
- Circuit density, 2
- Clear, 430
- Clear cache, 701
- Clear entry (CE), 701
- CLK (OUT) signal, 120
- Clock, 37
- Intel 8085, 119
- Intel 8086, 167, 221, 1003
- Intel 8088, 1003
- Intel 8089, 1003
- Intel 80186, 205
- Intel 80286, 312, 313

**Intel 8088, 530**
**Motorola 68000, 466, 468, 541**
**Motorola 68811, 727**
**CMOS RAM, 54**
**CPI see Compare**
**COBOL, 61**
**C0D, 314**
**Code perfect unit, 330**
**Code segment (CS), 172, 178**
**Command file, 63**
**Command registers, 27, 28, 130**
**Command signals, 25**
**Command status register (CSR), 134**
**Comment field, 65**
**Compare, 100, 582—587**
**Motorola 68000, 430**
**Motorola 68020, 577—692**
**Compare string, 53**
**Compressors, 40, 43, 53, 61, see also specific types**
**Computer-aided design (CAD), 555**
**Computer-aided engineering (CAE), 555**
**Computer-manufactured (CML), 555**
**Conditional branching, 5, 47, 211, 299—300**
**Conditional call instructions, 102**
**Conditional instructions, 722, see also specific types**
**Conditional jumps, 101, 211—212, 359—360**
**Conditional loops, 359—360**
**Conditional program counter (CCP), 29—31**
**Conditional return instructions, 102**
**Condition code register (CCR), 556, 668, 678**
**Condition codes, see Flags**
**Configuration error, 781**
**Control addressing mode, 413**
**Control bus, 127**
**Control bus, 3, 4**
**Control flags, 385**
**Control words, see also specific types**
**bus, 744, 1004**

**CRT, 892—900**
**dedicated, 66**
**DMA, see Direct memory access (DMA) controllers**
**floppy disk, 903—905**
**graphics, 897—900**
**input/output, 1**
**interrupts, 487, 488, 330**
**keyboard/display, 845—859, see also specific types**
**NIC, interfacing to, 845—849**
**hardware interface, 845—854**
**software interface, 854—859**
**peripheral, 1, see also specific types**
**real-time, 68—69**
**two-position, 65—68**
**Control logic, 75**
**Control registers, 251, 332**
**cache, 577—578**
**Motorola 68020, 556**
**Motorola 68100, 791, 793, 794**
**port A (TACR), 489, 495**
**port B (TCBR), 489, 495**
**Control signals, 4, see also specific types**
**Control unit, 4, 9, 26**
**hardwired, 9**
**Intel 80386, 380**
**microprogramming, 9**
**nanoprocessing, 9**
**Coprocessor interface registers (CIR), 706—708, 714**
**Coprocessors, 37—39, 905—907, see also specific types**
**DMA, 705**
**flitting point, 39, 713—735**
**Intel 80286 interface with, 314**
**Motorola 68020, 608, 692—696, 704—728**
**non-DMA, 705**
**peripheral hardware vs., 38—39**
**synchronization of with processor, 711**
**Counter register, 37, 175**
**CPL, see Current privilege level**
**CP/M, 53**
**CPU, see Central processing unit**
Displays, 855–877, see also Cathode ray tube (CRT); Monitors; specific types
Intel 8085-based microcomputer using the 8279, 859–877
Intel 8085 interface to, 859–845
Intel 8279 controller chip, see Controllers, keyboard/display
TTL, 865
Division, 174
Intel 8086, 186
Motorola 68000, 430, 432–433
Motorola 68020, 595–599
DMA, see Direct memory access
Documentation, 1
Double bus error, 472
Double bus fault, 620
Doubly linked list, 691
DP, see Display processor
DPC, 751
DPL, see Descriptor privilege level
DRQ, see Direct memory access request lines
DRO, see Destructive readout
DX, see Data strobe
DSACK, see Data transfer size acknowledge
DTACK, see Data acknowledge
DT/R, 226
Dual in-line package (DIP), 73, 115
Intel 8086, 167
Motorola 68000, 399, 465
Dual timing circuit, 622
DUP directive, 219–226
DX, see DEFINE WORD
DX, see Data register
Dyadic instructions, 721–722, 804, see also specific types
Dynamic bus sizing, 283, 609, 760
Dynamic RAM, 11, 243
Dynamic relocation, 816
EA, see Effective address
EAROM, see Electrically alterable ROM
ECS, see External cycle start
ECU, see Execution control unit
Editors, 40, 53, 54, 64, 963–964, see also specific types
EE PRO, see Electrically erasable PROM, 11
Effective address (EA), 178, 338
EFlags, 332, 334, see also Flags
Eight-bit microprocessors, 4, 6, 7, see also specific types
A/D converter and, 30
Coprocessors and, 87
input/output processor in, 25
memory-mapped I/O in, 29
workstations using, 86
EJECT, 89
Electrically alterable ROMs (EAROMs), 11
Electrically erasable PROMs (EEPROMs), 11
Electronic test equipment, 66
Elements of computers, 1, see also specific types
Emulators, see also specific types
Hardware, 665
Hewlett-Packard 64600, 969–972
984–986
in-circuit, see in-circuit emulator
(ICE)
Motorola 68000, 925
real-time hardware, 655
soft keys, 969–972
Enable interrupts (EI), 141, 150, 516
END, 59
Engineering, 64, 555
Enhanced instructions, 599–602, see also Advanced instructions; specific types
ENTER, 311
EPROM, see Erasable PROM
EQUATE (EQU), 87
Erasable PROM (EPROM), 10, 11, 51, 66
F
Fields, 55, see also specific types
address level, 675
address, 44, 45
back link, 427
comment, 55
descriptor type, 675
index, 23
label, 55
opcode, 44, 45, 56, 237
operand, 55
FIFO, see First-in first-out
File names, 50
Files, 50
command, 63
overlays, 48, 50
protection of, 50
Find and replace commands, 54
Firmware, 40
First-in first-out (FIFO), 19, 20, 22
Intel 8086, 171
Intel 80286, 171
Intel 8086, 351
Intel 80286, 330
Intel 8086, 455
Motorola 68000, 333
Flag address space, 333
Flags, 5, 175, 176, 273, 334, see also Registers; specific type
adjust, 346
auxiliary carry, 76, 176
borrow, 176
carry, 5, 76, 95, 176, 346
carry, 5, 76, 95, 176, 346
direction, 176
Intel 8086, 75, 76
Intel 8086, 175, 176
Intel 8086, 332, 334–337, 346–349, 362
interrupts, 34, 176, 334
overflow, 5, 6, 176, 346–347
parity, 76, 176, 346
resume, 336, 337
Interrupt systems, 68
Pentium core memory, 12
standard peripheral, 38–39
Hardware breakpoint system, 62
Hardware emulators, 665
Hardware interrupts, 255, 258—262,
271, 281, 641, 642
Hardware lock mechanisms, 454
Hardware reset
Intel 80286, 314
Motorola 68000, 473–474, 516,
712
Hardware stack, 6
Hardwired control unit, 9
HCMOS technology, 3, 549
Hewlett-Packard, 48, see also in Chip
Index
High-density MOS (HMOS) technology,
2, 3, 73, 167
High-level instructions, 310, 246,
355–368, see also specific types
High-level language programming,
43–44
High-level languages, 40, 60–61, see
also specific types
High to low memory quaste, 418, 459
History of microprocessor, 2–3
HLEDA, see HOLD acknowledge
HMOS, see High-density MOS
HOLD, 36, 119, 890
DMA controllers and, 879
Intel 8086, 221, 231, 232, 242
HOLD acknowledge (HLEDA), 36,
119, 154, 231, 233, 262, 890
HOLD request (HRQ), 184
HRQ, see HOLD request

I
IBM 360/85 computer, 22
IBM PC/AT, 312
ICE, see In-circuit emulator
IC layout, 68
IDT, 390
IDTR, 389
IF, see Interrupt flag
Immediate addressing mode, 46, 73,
176, 177, 388, 412–413
Immediate data addressing mode,
412–413
Impact printers, 864
Implicit implied addressing, 413
Implicit addressing mode, 45, 75, 78–
81, 185, 413–414
Inactive tasks, 326
In-circuit emulator (ICE), 49, 51, 51,
63, 66
Increment
8-bit (IN8), 105
16-bit (IN16), 108
Indirect addressing mode, 7, 181–
183, 391
Indexed addressing mode, 391
Index register indirect addressing,
mode, 410, 411
Index field, 28
Index register, 5, 7
Indirect addressing mode, 244–245
565–567
Indirect I/O transfer, 246
Indirect port mode, 185
Input/Output, 4
Interrupt organization, 4
Interrupt addressing mode, see implicit
addressing mode
INHIBIT, 37
Input/output, 1, 25–30, see also
Cardboard ray tube (CRT); Coprocessor; Disk drives; Display;
Keyboards; specific types
addressing modes, 185
but, 25, 26
characteristics of, 25
circuits, 31–33
controllers, 1
design of, 26
Direct transfer of, 246
instructions, 26, 47
Intel 8086, 271
Intel 80286, 125
Intel 80386, 180–186
DMA, 153–155
interrupt system, 141–153
programmed, see Programmed I/O
SID and SOD lines, 155–156
Intel 8279, 865–866
Intel 80386, 370
interface, 944–649
interrupts, 26, 31–34, 515–624
memory-mapped, see Memory-mapped I/O
Motorola 68020, 554–555
parallel, 27
physical, 26
ports, 26–29
8151–8156, 134–141
8451–8755, 130–134
8-bit, 246
16-bit, 246
privileged level (IOP), 325, 326, 336
processor, 262–265, see also specific types
speed, 25
standard, 28–29, 130, 185, 243
string, 564
virtual, 26
[IR], see Increment
insert mode in editors, 54
inception cycle, 45
Instruction decoders, 74
Instruction decode unit, 330
Instruction pointer (IP), 172
Instruction queue, 171, 436
Instruction register, 5, 74, 384, 337
Instructions, see also specific types
address of first, 3
advanced, see Advanced instructions
arithmetics, see Arithmetic instructions
components of, 44
conditional, 722
defined, 44
dynamic, 721–722
enhanced, 599–602
executability of, 122–129
format of, 213–215
high-level, 310, 365–368
immediate-mode, 46
Intel 8085, 78, 79–112, 122–129,
186
Intel 8086
format of, 213–215
size of, 186–212
tables, 268–204
Intel 80186, 369
Intel 80286, 315
Intel 80386, 329, 537, 341–368
interpretation of, 9
logical, see Logical instructions
mosaic, 719–721
Motorola 68000, 401, 404–405,
414–456
Motorola 68020, 572–602
Motorola 68030, 141–154
Motorola 68881, 728
Motorola 68681, 718
Motorola 88100, 800–819
one-address, 4
program-control, 47, 344–345
pseudo-, see Pseudoinstructions
sequencing of, 9
single-operands, 404, 407
timing of, 122–129
two-operands, 404, 407
types of, 74
zero-operands, 404, 407
Instruction unit (IU), 312, 789
INT, see Interrupt
INTA, see Interrupt acknowledge
Integer unit, 789
Intel Corporation, 1, 48
chip sets, see under Intel Corporation
Chip Index
history of microprocessors, 2
Intel I/O processor (IOP), 25, 36
Intelligent monitor interface, 39
Intel style, 58
Interface registers, 39, 724
Interleaved DMA, 36, 37
Internal Fragmentation, 21
Internal interrupts, see Traps
Internal registers, 791, 794
Interpreters, 40, 44, 61, see also specific types
Interrupt acknowledge (INTA), 34, 36,
119, 143, 149
Intel 8086, 231, 259
Motorola 68020, 608, 642
Interrupt address vector, 35, 521
Interrupt controller, 197, 308, 336
Interrupt deskewing logic, 642
Interrupt driven I/O, 26, 33–34
Interrupt enable bit, 149
Interrupt flag, 34, 376, 386
Interrupt I/O, 515–524
Interrupt pending (TPEND), 149, 619
Interrupt pin, 83
Interrupts, 9, 176, see also specific types
disable, 147
enable, 141, 150, 516
external, see External interrupts
handshake, 34
hardware, see Hardware interrupts
Intel 8085, 141–154
Intel 8086, 186, 256–262, 302
Intel 80186, 397
Intel 80386, 370–374
internal, see Traps
mountable, 54, 142, 258–262, 308
Motorola 68000, 477, 515–524,
527
Motorola 68020, 641–644, 649,
665, 666
non-mountable, 34, 142, 308
on overflow, 258
packed, 35, 36
predefined, 255, 257–258
printer interrupting and, 990
priority in, 35–36
software, 34, 35, 371
spurious, 517
timer, 124
types of, 34–35
user-defined hardware, 255, 258–
262
user-defined software, 255, 258
Interrupts-scheduled table, 324
Interrupt-service routine, 26, 28
Interrupt stack pointer (ISP), 341, 577,
635, 641
Interrupt vector addresses, 256
Interrupt vector assignment, 372–
373
INT, 34, 141–143, 151–154, 229,
235–241
INX, see Increment
I/O, see Input/output
IOP, see Intelligent I/O processor
IOP, see Input/output privilege level
IP, see Instruction pointer
TPEND, see Interrupt pending
IB, see Instruction register
Isolated I/O, see Standard I/O
ISP, see Interrupt stack pointer
IU, see Instruction unit
J
Jumps, 101, 211–212, 359–366
K
Keyboards, 25, 49, 835–877
actuation of, 836
description of, 835–836
Intel 8085 interface to, 839–845
Intel 8279 interface to, see Intel 8279
in Chip Index
key bounce on, 836
L
Label field, 55
LAN, see Local area network
Languages, see Programming languages
Large-scale integration (LSI), 877
Last-in-first-out (LIFO), 6, 685
LDA, 28, 29
LDS, see Lower data strobe
LDI, see Local descriptor table
General Index

POPAD, 361
Port A control register (PMCR), 489, 495
Port B control register (PSCR), 489, 495
Postindexed address register indirect mode, 509
Postindexed memory indirect mode, 565
Power supplies, 11
Practical applications of microprocessors, 66–69, see also specific applications
Preindexed address register indirect mode, 509
Predefined interrupts, 255, 257–258
Preindexed memory indirect mode, 565
Primary memory, see Main memory
Printer controllers, 887–891, see also specific types
Printers, 85, 80, 81
interface, 884
interfacing off, 884–891
LocalArea, 884
parallel, 884
serial, 884
Privilege levels, 21, 322, 390, 391
Privilege violations, 527, 528, 781
Processor extension acknowledge (PEACK), 314
Processor memory, 10
Processor register, 51
Processor service request register (PSRR), 882
Program control instructions, 47, 544–545, 642–642
Program counter (PC), 8, 9, 33
Intel 8085, 76, 77
Motorola 68020, 556, 567–572, 685, 675
saving of, 35
Program counter (PC) relative addressing, 412
Program diagnostic design, 64
Program error exceptions, 827, 828
Program logic array (PLA), 744
Programmable read-only memory (ROM), 10, 11, 50, 51
Programmed I/O, 26–28
conditional, 29–31
Intel 8085, 130
8155/8156 I/O ports, 134–141
8355/8355 I/O ports, 130–134
Intel 8086, 243–246
Motorola 68000
-6821 interface, 508–508
68230 interface, 486–507
unconditional, 29–31
Programming, 25, 40–44, see also Software: specific types
application, 41, 66, 68, 233
assembly language, 10, 42–43, 133
debugging, 62
development, 49
EPROM, 927–928
high-level language, 42–43
machine language, 42–43
modular, 670, 671
stimulator, 61–63
source, 40, 43
Programming languages, see also specific types
assembly, 40, 55, 60
high-level, 40, 60–61
Programming models
Intel 80386, 332–340
Motorola 68000, 402–404
Motorola 68020, 556–656
Motorola 68030, 745–747
Motorola 88100, 793, 798
Program status word (PSW), 76
PRISM, see Programmable read-only memory
Protected indirect calls, 391
Protected mode, 315, 388–393
Protected virtual address mode (PVAM), 179, 185–187
Intel 80286, 312, 315, 328
Intel 80386, 594
Protection, see also specific types
address, 18
block, 22
file, 50
Intel 80286, 312, 321–322
Intel 80386, 349, 349
mechanisms of, 22
Motorola 68851, 737
operating system, 316, 322, 324
physical memory, 22
Protection enable (PE), 314, 323, 332
Protection test unit, 330
Prototypes, 66
Pseudoinstructions, 56–63, see also specific types
Hewlett-Packard 64000, 968–968
Intel 8086, 216–220
PSR, see Processor service request register
PSW, see Program status word
PTEST, 748–749
P-type MII, 2
PUSH, 6, 7, 456, 457, see also WRITE
PUSHAD, 361
PUSH-FULL, circuit, 51
PVAM, see Protected virtual address mode
Q
Quad pack, 465
Quates, 455, see also specific types
instructions, 171, 380
Motorola 68000, 468–468
R
RAM, see Random access memory
Random access memory (RAM), 6, 10–12, 49
at cache memory, 24
GMCS, 34
development system, 64–65
dynamic, 11, 243
Intel 8279, 865–866
Motorola 68020, 687
nonvolatile, 49
read display, 856
refreshing of, 11
sensor, 858
shared, 857
static, 11, 12, 241–242
write display, 856
Rater scan, 893
RCA Corporation, 48
RD, 125, 125, 890
READ, 4, 6, 875, see also POP
Intel 80286, 282–286
memory, 125
Motorola 68000, 479
Motorola 68030, 757–756
Read display RAM, 856
Read modify-write, 618
Read mostly memories (MAMs), 11
Read-only memory (ROM), 5, 10, 12
character generator, 10
flash link, 10
Intel 8006, 241
in 8355/8355, 27
mask programmable, 10
programmable (ROM), 10, 11, 50, 51
two-level, 9
Read/write, 12, 684, 685
Read/write head for disk drives, 900
Read-write-modify (RWM), 841, 683,
684, 686, 687
READY, 111, 313
Real address mode, 170, 314, 317, 322
Intel 80286, 312
Intel 80386, 386–387
Real mode, see Real-address mode
Real-time controllers, 68–69
Real-time hardware emulators, 665
Real-time robotics control, 66
Redesigned instruction set computers (RISC), 3, 786–787, see also
Motorola 88100 in Chip Index: specific types