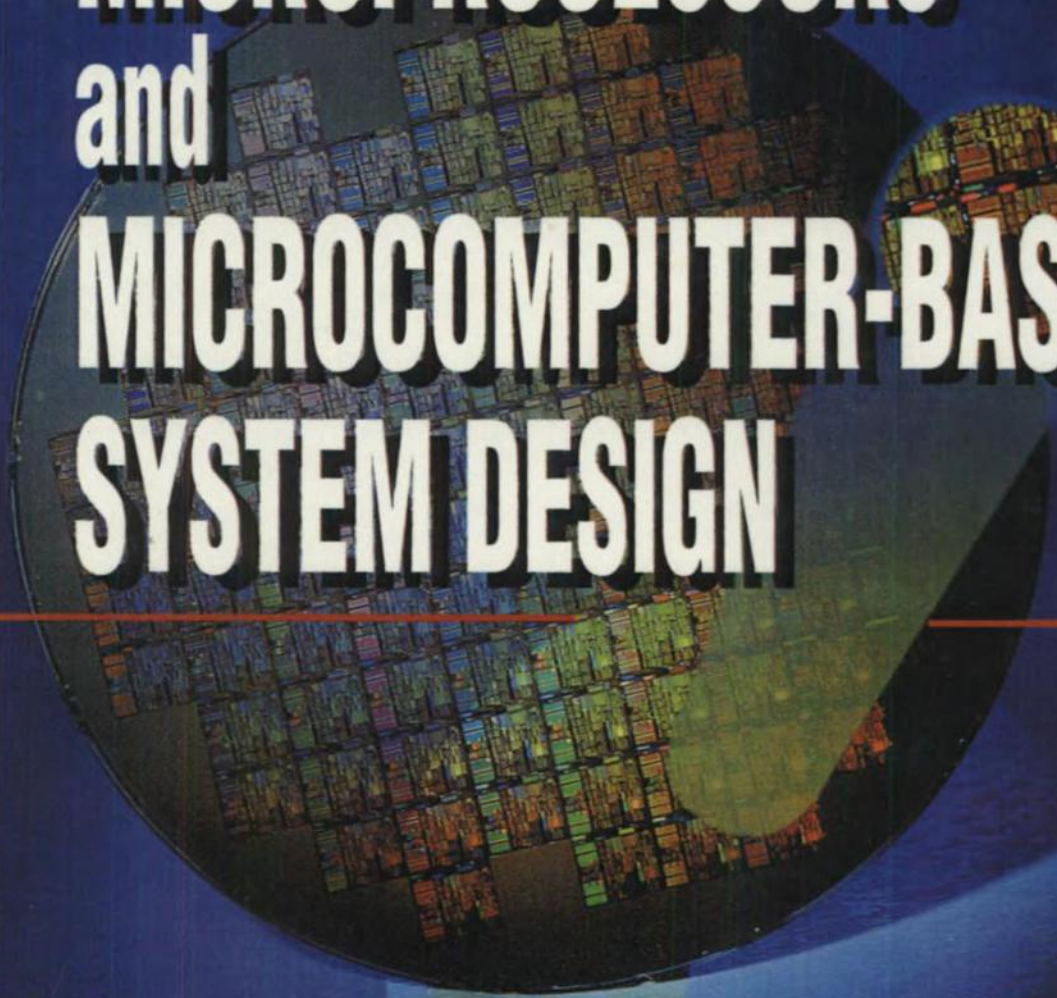
Second Edition

MICROPROCESSORS and MICROCOMPUTER-BASED SYSTEM DESIGN



MOHAMED RAFIQUZZAMAN

Second Edition

MICROPROCESSORS and MICROCOMPUTER-BASED SYSTEM DESIGN

MOHAMED RAFIQUZZAMAN, PH.D.

Professor
California State Polytechnic University
Pomona, California
and
Adjunct Professor
University of Southern California
Los Angeles, California

CRC Press
Boca Raton New York London Tokyo

Library of Congress Cataloging-in-Publication Data

Rafiquzzaman, Mohamed.
Microprocessors and microcomputer-based system design / Mohamed Rafiquzzaman. — 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-8493-4475-1
1. Microprocessors. 2. Intel 80xx series microprocessors. 3. Motorola 68000 series microprocessors.
4. Microcomputers. I. Title.
QA76.5.R27848 1995
004.16—dc20 95-7374
CIP

International Standard Book Number 0-8493-4475-1
Library of Congress Card Number 95-7374
Printed in the United States of America 2 3 4 5 6 7 8 9 0
Printed on acid-free paper

Preface

This book is based on the fundamental concepts associated with typical 8-, 16-, and 32-bit microprocessors and microcomputers. These concepts are related in detail to the Intel 8085/8086/80386/80486/80960 and Motorola 68000/68020/68030/88100. A brief coverage of Intel Pentium, Motorola 68040, Motorola/IBM/Apple PowerPC, and DEC's Alpha is also included.

With the growing popularity of both Intel and Motorola 32-bit microprocessors, it is now necessary to cover these processors at the undergraduate and graduate levels. Therefore, a thorough coverage of these processors is provided.

A detailed treatment of Intel 80386 and Motorola 68020/68030 along with more examples and system design concepts is included. Programming and system design concepts associated with other popular 32-bit microprocessors such as Intel 80486/80960 and Motorola 68040 are also covered in this book. Finally, an overview of Intel Pentium microprocessor is provided. Since the fundamental concept of 8-bit microprocessors, along with the Intel 8085, has proved its worth many times over in the intervening years, the 8085 has been retained in this edition.

This book is divided into ten chapters. Chapter 1 contains the basics of microprocessors, as in the first edition. New topics such as floating-point arithmetic, Program Array Logic (PAL) used for address decoding for 32-bit microprocessors, flash memories and an overview of various 32-bit microprocessors is also included.

Chapter 2 covers details of the 8085 microprocessor.

Chapters 3 through 8 provide detailed descriptions of the architectures, addressing modes, instruction sets, I/O and system design concepts of Intel's 8086, 80386, 80486, and 80960 and Motorola's 68000, 68020, 68030, 68040, and 88100 microprocessors. An overview of Intel 80186, 80286, Pentium and PowerPC microprocessors are also included.

Chapter 9 contains fundamentals of peripheral interfacing.

Chapter 10 includes system design concepts along with the applications of design principles covered in the preceding chapters. Three system design examples using the 8085, 8086, and 68000 are included in detail.

The appendices include materials on the HP 64000 microcomputer development systems, data sheets on various microprocessors and support chips, and a glossary.

The audience of this book can be college students or practicing microprocessor system designers in the industry. It can be used as an undergraduate or graduate text in electrical engineering, computer engineering or computer science. Practitioners of microprocessor system design in the industry will find greater detail and comparison considerations than are found in manufacturers' manuals. The book assumes a familiarity with digital logic and topics such as Boolean Algebra and K-maps.

The author wishes to express his sincere appreciation to his student Frank Lee for making constructive suggestions and typing the manuscript. The author is also grateful to Dr. W.C. Miller of University of Windsor, Canada, and others for their support throughout the writing effort.

Mohamed Rafiquzzaman
Pomona, California

The Author

Mohamed Rafiquzzaman obtained his Ph.D. in Electrical Engineering in Canada in 1974. He worked for Esso/Exxon and Bell Northern Research for approximately 5 years. Dr. Rafiquzzaman is presently a professor of electrical and computer engineering at California State Polytechnic University, Pomona. He was Chair of the department there from 1984 to 1985. Dr. Rafiquzzaman is also an adjunct professor of electrical engineering systems at the University of Southern California, Los Angeles. He consulted for ARCO, Rockwell, Los Angeles County, and Parsons Corporation in the areas of computer applications. He has published six books on computers, which have been translated into Russian, Chinese and Spanish, and has published numerous papers on computers.

Dr. Rafiquzzaman is the founder of Rafi Systems, Inc. a manufacturer of biomedical devices and computer systems consulting firm in California. In 1984, he managed the Olympic Swimming, Diving and Synchronized Swimming teams. He has also managed Swiss timing, scorekeeping, and computer systems.

From 1984 to 1989, he was the instructor for Motorola in Southern California teaching short courses on 68000, 68020, and 68030 for local industries.

Dr. Rafiquzzaman was an advisor to the President of Bangladesh on computers from 1988 to 1990. He is currently involved in research activities in both hardware and software aspects of typical 16- and 32-bit microprocessor-based applications. These activities include image processing, robotics control, and OCR (Optical Character Recognition).

To my parents,
my wife Kusum, son Tito,
and brother Elan

In memory of my brother, Dr. M. Kaisaruzzaman

Table of Contents

1

INTRODUCTION TO MICROPROCESSORS AND MICROCOMPUTER-BASED APPLICATIONS

This chapter provides a brief summary of the features of microprocessors and microcomputer-based applications.

The basic elements of a computer are the Central Processing Unit (CPU), the Memory, and Input/Output (I/O) units. The CPU translates instructions, performs arithmetic or logic operations, and temporarily stores instructions and data in its internal high-speed registers. The memory stores programs and data. The I/O unit interfaces the computer with external devices such as keyboard and display.

With the advent of semiconductor technology, it is possible to integrate the CPU in a single chip. The result is the microprocessor. Metal Oxide Semiconductor (MOS) technology is typically used to fabricate the standard off-the-shelf microprocessors such as those manufactured by Intel and Motorola. Appropriate memory and I/O chips are interfaced to the microprocessor to design a microcomputer. Single-chip microcomputers are also available in which the microprocessor, memory, and I/O are all fabricated in the same chip. These single-chip microcomputers offer limited capabilities. However, they are ideal for certain applications such as peripheral controllers.

Single chip microcomputers are also referred to as "microcontrollers". The microcontrollers are typically used for dedicated applications such as automotive systems, home appliances and home entertainment systems. Typical microcontrollers, therefore, include on-chip timers, A/D (Analog to Digital) and D/A (Digital to Analog) converters. Two popular microcontrollers are Intel 8751 (8-bit)/8096 (16-bit) and Motorola HC11 (8-bit)/HC16 (16-bit). The 16-bit microcontrollers include more on-chip ROM, RAM, and I/O compared to the 8-bit microcontrollers.

The efficient development of microprocessor-based systems necessitates the use of a microcomputer development system. The microcomputer development system is used for the design, debugging, and sometimes the documentation of a microprocessor-based system.

This chapter first covers the evolution of 8-, 16-, and 32-bit microprocessors along with an overview of programming languages, microcomputer hardware, and software. The attributes of typical microcomputer development systems features as well as some specific microprocessor applications are also included.

1.1 Evolution of the Microprocessor

Intel Corporation introduced the first microprocessor, the 4004 (4-bit), 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 4-bit 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 MOS Technology 6502, Motorola 6800 and 6809, Intel 8085, and Zilog Z80 evolved. These 8-bit 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 (16-bit ALU) and include typical processors such as Intel 8086/80186/80286 and Motorola 68000/68010. These microprocessors were designed using the HMOS (high-density MOS) technology. HMOS provides the following advantages over NMOS:

- Speed-Power-Product (SPP) of HMOS is four times better than NMOS:
  NMOS = 4 Picojoules (PJ)
  HMOS = 1 Picojoule (PJ)

  Note that Speed-Power-Product
  = speed * power
  = nanosecond * milliwatt
  = picojoules

- Circuit densities provided by HMOS are approximately twice those of NMOS:
  NMOS = 4128 gates/μm^2
  HMOS = 1852.5 gates/μm^2

  where 1 μm (micrometer) = 10^{-6} meter.

Later, Intel utilized the HMOS 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 68020/68030/68040/PowerPC, Intel's 80386/80486 and the Intel Pentium processor. These processors are fabricated using the low-power version of the HMOS 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. Both Intel and Motorola introduced 32-bit RISC (Reduced Instruction Set Computer) microprocessors, namely the Intel 80960 and Motorola MC88100/PowerPC with simplified instruction sets. Note that the purpose of RISC microprocessors is to maximize speed by reducing clock cycles per instruction. Almost all computations can be obtained from a simple instruction set.

RISC microprocessors have hardwired instruction sets like the non-RISC microprocessors such as the Intel 8085 and MOS 6502. This means that for every instruction, there exists actual, physical connections that provide the desired instruction decoding. The microprocessor,

therefore, does not use valuable clock cycles (machine cycles) in instruction decoding. Most RISC instructions require a maximum of only two clock cycles to complete. These instructions are restricted to register-to-register operations with load and store for memory access. Since RISC-type microprocessors are hardwired, instructions may be executed simultaneously (as long as the instructions do not share the same register.) This technique is known as pipelining.

Pipelining is the mechanism that actually enables simultaneous processing to occur. Instructions are fetched in sequential order. The processor continues to fetch instructions even though it has not executed the present instruction. This provides an input pipeline to the instruction cache. At this stage, the instructions are intelligently fed into the instruction execution unit for processing. When an instruction is executed, the registers used by the instruction clear respective bits in a scoreboard register. As the microprocessor is executing a given instruction, it checks the scoreboard register to see if the next instructions (residing in a fast read/write memory internal to the microprocessor called the instruction cache) use registers that are currently in use. Instructions not using the same registers can therefore be executed at the same time. The RISC microprocessors can execute as many as five instructions simultaneously.

The trend in microprocessors is implementation of more on-chip functions and for improvement of the speeds of memory and I/O devices. Some manufacturers are speeding up the processors for data crunching type applications. Digital Equipment Corporation's Alpha 21164 with 300 MHz clock, four instruction-per-cycle rate and RISC-based architecture is the fastest microprocessor available today.

1.2 Microprocessor Data Types

This section discusses the data types used by microprocessor. Typical data types include signed and unsigned binary integers, binary coded decimal (BCD), American Standard Code for Information Interchange (ASCII) and floating-point numbers. Note that binary integer numbers do not support fractions in microprocessors. Fractions and mixed numbers (numbers comprised of integers and fractions) use either binary or BCD floating-point formats. Floating-point numbers are often referred to as real numbers.

1.2.1 Unsigned and Signed Binary Integers

An unsigned binary integer has no arithmetic sign. Unsigned binary numbers are therefore always positive. An example is memory address which is always a positive number. An 8-bit unsigned binary integer represents all numbers from 00_{16} through FF_{16} (0_{10} through 255_{10}).

A signed binary integer, on the other hand, includes both positive and negative numbers. It is represented in true form for a positive number and in two's complement form for a negative number. For example, the decimal number +10 can be represented as a true form 8-bit number in a microprocessor as 0000 1010 (binary) or 0A (hexadecimal). The decimal number -10 can be represented in two's complement form as 1111 0110 (binary) or F6 (hexadecimal). The most significant bit of a signed binary number represents the sign of the number. For example, bit 7 of an 8-bit signed number, bit 15 of a 16-bit signed number and bit 31 of a 32-bit signed number represent the signs of the respective numbers. A 32-bit signed binary integer includes all numbers from -2,147,483,648 to 2,147,483,647 with 0 being a positive number.

Finally, note that the hexadecimal number FF_{16} is 255_{10} when represented as an unsigned number. On the other hand, FF_{16} is -1_{10} when represented as a signed number.

1.2.2 BCD (Binary Coded Decimal) Numbers

A BCD digit consists of four bits with a value ranging from 0000_2 to 1001_2 (0 through 9 decimal). A BCD digit greater than 9_{10} can be represented as two or more BCD digits.

Microprocessors store BCD numbers in two forms, packed and unpacked. The unpacked BCD number represents each BCD digit as a byte while the packed BCD number represents two BCD digits in a byte. For example, 23_{10} is represented as 0000 0010 0000 0011_2 as two unpacked BCD numbers, while it is represented as 0010 0011_2 as a packed BCD number.

Microprocessors normally input data from a keypad in unpacked BCD form. This data can then be converted by writing a program in the microprocessor to packed BCD form for arithmetic operations or for storing in memory. After processing, the packed BCD result is then converted to unpacked BCD form by another microprocessor program written by the user for displays. Typical displays use either unpacked BCD data or unpacked BCD data converted to seven segment code by the microprocessor's program.

1.2.3 ASCII

ASCII (American Standard Code for Information Interchange) is a code that represents alphanumeric (alpha characters and numbers) in a microcomputer's memory. ASCII also represents special symbols such as # and %. It is a 7-bit code. The most significant bit (bit 7) is sometimes used as a parity bit. The parity bit represents the number of ones in the byte. If the number of ones is odd, the parity is odd; otherwise, the parity is even.

Also, note that the hexadecimal numbers 30_{16} through 39_{16} are ASCII codes representing the decimal numbers 0 through 9. A listing of the ASCII codes is included in Appendix F.

1.2.4 Floating-Point Numbers

Floating-Point numbers contain three components. These are a sign, exponent and mantissa. For example, consider the decimal value -2.5×10^{-2}. The sign is negative, the exponent is -2 and the significand or mantissa is 2.5. In floating point numbers, it is possible to store the same number in several different ways. For example, 1 can be represented as 10.0×10^{-1}, 1.0×10^{0}, 0.1×10^{1} and 0.01×10^{2}. To make computations yield the maximum accuracy, the numbers are normalized. This means that the exponent is adjusted so that the mantissa always follows a specific format. A binary floating point number is represented as a normalized binary fraction raised to a power of 2. To convert a binary number to its standard floating-point form, the binary number is converted to a normalized floating-point number. Note that a normalized binary floating-point number is represented as 1.XXXXXX raised to some power of 2 where X can be 0 or 1. The 1. part is implied and is not stored by the microprocessor. First, the binary number is converted to a common 32-bit floating-point format. The most significant bit (bit 31) is the sign bit (S). If S is zero, the number is positive, while if S is one, the number is negative. The next 8 bits contain the biased exponent. This means that an 8-bit number $7F_{16}$ or 127_{10} is added to the exponent. Biased exponent makes numeric comparison (such as less than or greater than) easy. The bias is usually chosen such that the most negative number allowed in the exponent becomes zero and the most positive number becomes the largest value of the representation. For example, with an 8-bit exponent, and $+127_{10}$ bias, the smallest and the largest values of the bias exponent are 0_{10} and 255_{10} respectively. Note that the zero biased exponent is represented as 127_{10}, while the unbiased minimum and maximum values of the 8-bit exponent are -127_{10} and $+128_{10}$ respectively. The remaining 23 bits represent the fractional part of the number. Note that zero is stored as 32 zeros while infinity is stored as 32 ones. As an example, consider converting the decimal number 10 to the standard floating-point format as follows:

1. $10_{10} = 1010_2$
2. Normalize the binary number as $1.XXXXXX \times 2^n = 1.010 \times 2^3$
3. Sign, S = 0 for positive
4. Biased exponent = $7F_{16} + 3 = 82_{16}$

5. 23-bit Fraction = 0100 0000 0000 0000 0000 000
6. The floating-point equivalent of 10_{10} is

S	Exponent	Fraction
0	1000 0010	0100 0000 0000 0000 0000 0000

A special case of the floating-point format is called NaN (Not a Number). NaNs are results generated by floating-point operations that have no mathematical interpretations. These results may be generated by operations such as multiplication of infinity by infinity.

The BCD floating-point form represents a number in BCD scientific notation. The number is represented as normalized significand raised to some power of 10. Each BCD floating-point number is represented in typical microprocessors as 80 bits. The BCD fraction is 16 digits wide (64-bit) and is stored as packed BCD digits. The whole number portion of the significand is stored as one digit BCD from 0 to 9. The BCD exponent along with the sign is expressed as 12 bits.

Typical floating-point coprocessors such as 80387 (for the Intel 80386 microprocessor) and 68881/68882 (for the Motorola 68020/68030 microprocessor) support several data types. For example, the 80387 coprocessor supports seven data types. These are word integer (16-bits), short integer (32-bit), long integer (64-bit), packed BCD (80-bit), short real (32-bit), long real (64-bit), and temporary real (80-bit).

The 80387 integer data types are represented by the two's complement same as those used by the 80386. The only difference is that the 80386 supports an 8-bit integer while the 80387 supports a 64-bit integer.

The 80387 supports 80-bit packed BCD with 18 decimal digits (bits 0-71), bit 79 as the sign-bit and seven (bits 72-78) unused bits. With 18-digits representation, the COBOL standard (the High level language utilizing BCD) is followed.

The 80387 supports three real data formats. These are short real (32-bit with one sign bit, 8-bit exponent and 23-bit significand), long real (64-bit with one sign-bit, 11-bit for exponent and 62-bit for the significand), temporary real (80-bit with one sign-bit, 15-bit exponent, and 64-bit significand).

The 80387 uses the temporary real format internally. All data types are converted by the 80387 immediately into temporary real. This is done to maximum precision and range of computations.

The 80387 supports four special cases. These are zeros, infinities (both positive and negative), denormals, and NaNs (signalling and quiet).

Denormals represent very small numbers that are not normalized. Normally, numbers are required to be normalized by shifting to left until the most significant is one. Denormals do not have one as the most significant bit of the significand. Denormals permit a gradual underflow. That is the precision is lost gradually rather than abruptly. When the least normalizable number is reached, the next small representation is zero. Denormals provide gradual underflow of numbers that are not normalized. That is, denormals extend the range of very small numbers significantly, but with some loss in precision.

A signaling NaN causes an invalid operation exception when used in an operation. A quiet NaN, on the other hand, does not cause an invalid operation exception.

The floating-point data types supported by the Motorola 68881/68882 floating-point coprocessors are summarized next. Note that the 68881 and 68882 differ in execution speed. They are basically identical. The 68882 is an enhanced version of the 68881 in that it executes several floating-point instructions concurrently with the 68020/68030. The 68881/68882 supports integers, binary floating-point numbers, and packed floating-point BCD. Data are represented externally by using these formats. The 68881/68882 utilizes an 80-bit binary floating point form to represent all data internally.

The 68881/68882 supports three signed integer formats. These are 8-bit byte, 16-bit word, and 32-bit long word.

Binary floating-point format is also called binary real form. The 68881/68882 supports binary floating-point form which contains three fields. These are a sign, biased exponent and a significand. The 68881/68882 operates on these sizes, namely 32-bit single-precision, 64-bit double precision, and 96-bit extended precision.

For single precision (bit 31: sign bit, bits 23-30: 8-bit exponent, bits 0-22: 23-bit significand), 64-bit double precision (bit 63: sign bit, bits 62-52: 11-bit exponent, bits 0-51: 52-bit significand), and 96-bit extended-precision (bit 95: sign bit, bits 80-94: 15-bit exponent, bits 64-79: zero; sixteen unused bits, bits 0-63: 64-bit significand).

The biased exponent is used. The Single Precision adds a bias of 127_{10} ($7F_{16}$), double Precision uses a bias of 1023_{10} ($3FF_{16}$), and the extended-Precision uses a bias of 16383_{10} ($3FFFF_{16}$). The bias is added to the exponent before it is stored in this format and subtracted to convert to a true exponent when the number is interpreted.

A few special cases that do not conform to the floating-point form are also handled by the 68881/68882. For example, a zero is represented with all bits of the exponent and significand as zeros. The sign bit may be a one or a zero representing +0 or -0. The infinity, on the other hand is represented by all bits in the exponent and significand set to ones. The sign bit may be zero or one representing positive or negative infinity. The 68881/68882 supports BCD floating point form which represent each number as normalized significand raised to a power of 10. This format stores a number as 96 bits. The least significant 64 bits (8 bytes) contain the 16-digit BCD fraction. The next byte contains the whole number portion of the significand (0-9). The most significant bit (bit 95) contains the sign of significand while bit 94 includes the sign of the exponent. The exponent is represented by three digit BCD packed exponent (000-999) in bits 80–91. Similar to the 80387, the 68881/68882 also represents NaN's and also provides exceptions for signaling NaN's.

IEEE has established the standard for floating-point arithmetic specified by ANSI-IEEE754-1985. Typical 32-bit microprocessors use this standard.

1.3 Microcomputer Hardware

In this section, some unique features associated with various microcomputer components will be described.

The microcomputer contains a microprocessor, a memory unit, and an input/output unit. These elements are explained in the following in detail. Figure 1.1 shows a simplified block diagram of a microcomputer.

1.3.1 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 each of the memory and I/O elements so that information transfer between the microprocessor and any of the other elements can take place.

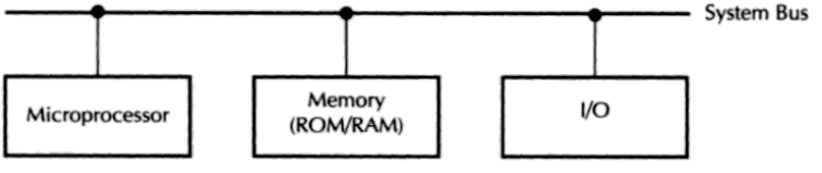


FIGURE 1.1 Simplified block diagram of a microprocessor.

On the address bus, information transfer normally takes place only in one direction, from the microprocessor to the memory or I/O elements. Therefore, this is called a unidirectional bus. This bus is usually 16 to 32 bits wide. The number of unique addresses that the microprocessor can generate on this bus depends on the width of this bus. For example, for a 16-bit address bus, the microprocessor can generate $2^{16} = 65{,}536$ different possible addresses. A different memory location or an I/O element can be represented by each one of these addresses.

The data bus is a bidirectional bus, that is, information can flow in both directions, to or from the microprocessor. This bus is normally 8, 16, or 32 bits wide.

The control bus is used to transmit signals that are used to synchronize the operation of the individual microcomputer elements. Typical control signals include READ, WRITE, and RESET. Some signals on the control bus such as interrupt signals are unidirectional, while some others such as RESET may be bidirectional.

1.3.2 The Microprocessor

The commercial microprocessor, fabricated using the MOS technology, is normally contained in a single chip. The microprocessor is comprised of a register section, one or more ALUs (Arithmetic Logic Units), and a control unit. Depending on the register section, the microprocessor can be classified either as an accumulator-based or a general-purpose register-based machine.

In an accumulator-based microprocessor such as the Intel 8085 and Motorola 6809, one of the operands is assumed to be held in a special register called the "accumulator". All arithmetic and logic operations are performed using this register as one of the data sources. The result after the operation is stored in the accumulator. One-address instructions are very predominant in this organization. Eight-bit microprocessors are usually accumulator-based.

The general-purpose register-based microprocessor is usually popular with 16- and 32-bit microprocessors, such as Intel 8086/80386/80486 and Motorola 68000/68020/68030/68040, and is called general-purpose, since its registers can be used to hold data, memory addresses, or the results of arithmetic or logic operations. The number, size, and types of registers vary from one microprocessor to another. Most registers are general-purpose registers, while some are provided with dedicated functions.

Typical dedicated registers include the Program Counter (PC), the Instruction Register (IR), Status Register (SR), the Stack Pointer (SP) and the Index Register. The 32-bit microprocessors include special on-chip combinational network called 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 normally initialized with the address of the first instruction. For example, the 80486, upon hardware reset, reads the first instruction from the 32-bit address $FFFFFFF0_{16}$. In order to execute the instruction, the microprocessor normally places the PC contents on the address bus and reads (fetches) the first instruction from external memory. The program counter contents are then automatically incremented by the ALU. The microcomputer thus executes a program sequentially unless it encounters a jump or branch instruction. The size of the PC varies from one microprocessor to another depending on the address size. For example, the 68000 has a 24-bit PC, while the 68040 contains a 32-bit PC.

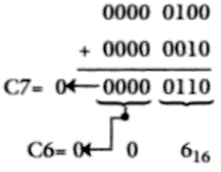
The instruction register (IR) contains the instruction to be executed. After fetching an instruction from memory, the microprocessor places it in the IR for translation.

The status register contains individual bits with each bit having a special meaning. The bits in the status register are called flags. Each flag is usually set or reset by an ALU operation. The flags are used by the Conditional Branch instructions. Typical flags include carry, sign, zero, and overflow.

The carry (C) flag is used to reflect whether or not an arithmetic operation such as ADD generates a carry. The carry is generated out of the 8th bit (bit 7) for byte operations, 16th bit (bit 15) for 16-bit, or 32nd bit (bit 31) for 32-bit operations. The carry is used as the borrow flag for subtraction. In multiple word arithmetic operations, any carry from a low-order word must be reflected in the high-order word for correct results.

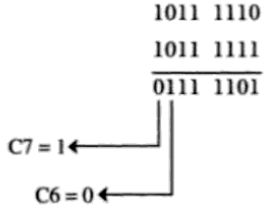
The zero (Z) flag is used to indicate whether the result of an arithmetic or logic operation is zero. Z = 1 for a zero result and Z = 0 for a non-zero result. The sign flag (sometimes also called the negative flag) indicates whether a number is positive or negative. S = 1 indicates a negative number if the most significant bit of the number is one; S = 0 indicates a positive number if the most significant bit of the number is zero.

The overflow (V) flag is set to one if the result of an arithmetic operation on signed (two's complement) numbers is too large for the microprocessor's maximum word size; the C flag is overflow for unsigned numbers. The overflow flag for signed 8-bit numbers can be shown as V = C7 ⊕ C6, where C7 is the final carry and C6 is the previous bit's carry. The ⊕ symbol indicates exclusive-OR operation. This can be illustrated by the numerical examples shown below:



04_{10}
$+ 02_{10}$
06_{10}

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:



$- 66_{10}$
$- 65_{10}$
$+ 125_{10(?)}$
result is incorrect

The result is incorrect. Since V = C6 ⊕ C7 = 0 ⊕ 1 = 1, the overflow flag is set. Note that this applies to signed two's complemented 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 items that go in last will come out first. This is because stacks perform all read (POP) and write (PUSH) operations from one end.

The stack is addressed by a register called the stack pointer (SP). The size of the SP is dependent on the microprocessor's address size. The stack is normally used by subroutines or interrupts for saving certain registers such as the program counter.

Two instructions, PUSH (stack write) and POP (stack read), can usually be performed by the programmer to manipulate the stack. If the stack is accessed from the top, the stack pointer is decremented before a PUSH and incremented after a POP. On the other hand, if the stack is accessed from the bottom, the SP is incremented before a PUSH and decremented after a POP. Typical microprocessors access the stack from the top. Depending on the microproces-

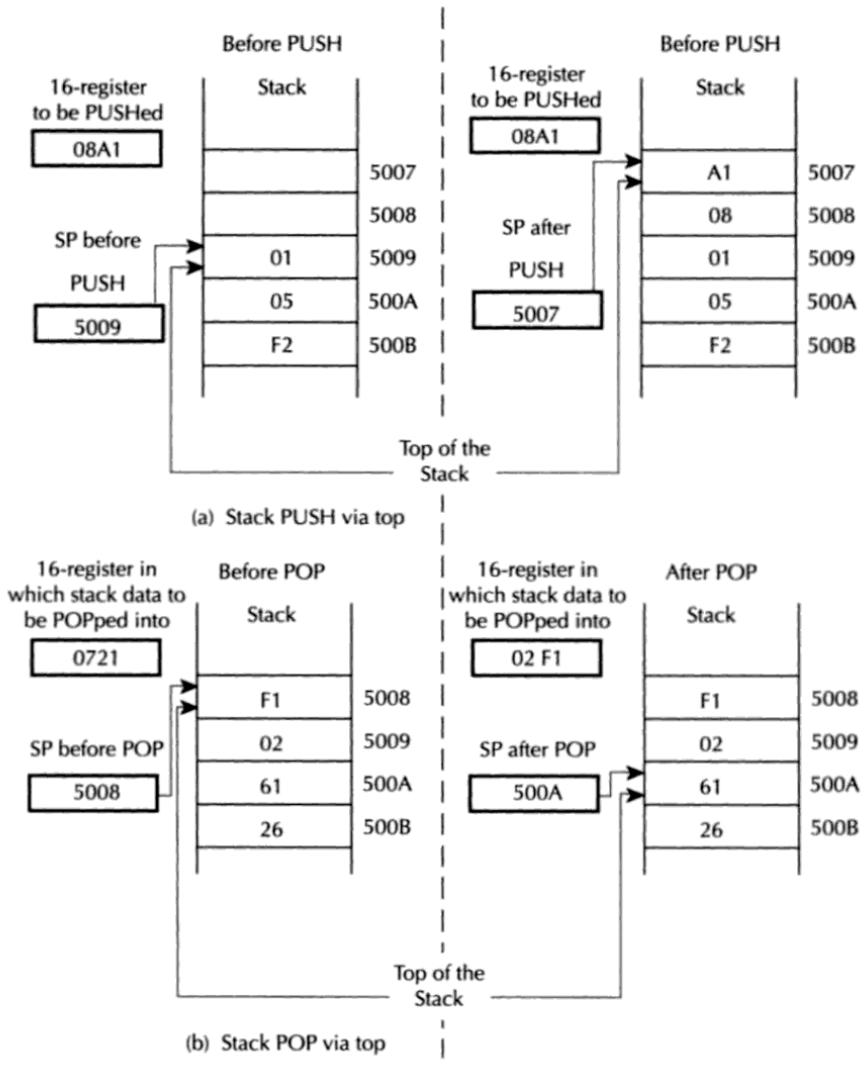


FIGURE 1.2 PUSH and POP operations via top.

sor, an 8-, 16-, or 32-bit register can be pushed onto or popped from the stack. The value by which the SP is incremented or decremented after POP or PUSH operations depends on the register size. For example, values of one for an 8-bit register, two for 16-bit registers, and four for 32-bit registers are used. Figure 1.2 shows the stack data when accessed from the top before and after PUSHing a 16-bit register onto the stack or POPping 16 bits from the stack into the 16-bit register. Note that stack items PUSHed must be POPped in reverse order. The item pushed last must be popped first.

Consider the PUSH operation in Figure 1.2a when the stack is accessed from the top. The SP is decremented by 2 after the PUSH. The SP is decremented since it is accessed from the top. A decrement value of 2 is used since the register to be pushed is 16 bits wide.

The POP operation shown in Figure 1.2b is the reverse of the PUSH. The SP is incremented after POP. The contents of locations 5008_{16} and 5009_{16} are assumed to be empty.

An index register is typically used as a counter for an instruction or for general storage functions. The index register is useful with instructions where tables or arrays of data are accessed. The general-purpose register-based microprocessor can use any general-purpose register as the index register.

Typical 32-bit microprocessors such as the Intel 80386/80486 and Motorola 68020/68030/68040 include a special type of shifter called barrel shifter for performing fast shift operations.

The barrel shifter is an on-chip combinational network for 32-bit microprocessors and provides fast shift operations. For example, the 80386 barrel shifter can shift a number from 0 through 64 positions in one clock period (clock rate is 16.67 MHz).

The ALU in the microprocessor performs all arithmetic and logic operations on data. The size of the ALU defines the size of the microprocessor. For example, Intel 8086 (or Motorola 68000) is a 16-bit microprocessor since its ALU is 16 bits wide. The Intel 8088 (or Motorola 68008) is also a 16-bit microprocessor since its ALU is 16 bits wide, even though its data bus is 8 bits wide. Motorola 68040 (or Intel 80486) is a 32-bit microprocessor since its ALU is 32 bits wide. The ALU usually performs operations such as binary addition and subtraction. The 32-bit microprocessors include multiple ALUs for parallel operations and thus achieve fast speed.

The control unit of the microprocessor performs instruction interpreting and sequencing. In the fetch phase, the control unit reads instructions from memory using the 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
- Nanoprogramming

The hardwired control unit is designed by physically connecting typical components such as gates and flip-flops. Typical 32-bit RISC microprocessors such as the Intel 80960 and Motorola 88100 are 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 68000/68020/68030/68040) 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, 68030, and 68040 are nanoprogrammed.

1.3.3 Memory Organization

1.3.3.a Introduction

A memory unit is an integral part of any microcomputer system and its primary purpose is to hold programs and data.

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 the microprocessor 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 or main 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.

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 or virtual memory.

Secondary memory stores programs and data in excess of the main memory. The microcomputer cannot directly execute programs stored in the secondary memory. In order to execute these programs, the microcomputer must transfer them to its main memory by a system program called the operating system. This topic is covered later in the chapter.

Data in disk memories are stored in tracks. A track is a concentric ring of data stored on a surface of a disk. Each track is further subdivided into several sectors. Each sector typically stores 512 or 1024 bytes of data. All disk memories use magnetic media except the optical disk memory which stores data on a plastic disk. Data is read or sometimes written on the optical disk with a laser beam. There are two types of optical disks. These are the CD-ROM (Compact Disk Read Only Memory) and the WORM (Write Once Read Many). The CD-ROM is inexpensive compared to the WORM drive. However it suffers from lack of speed and has limited software applications at the present time. The WORM drive is typically used in huge data storing applications such as insurance and banking since data can be written only once. The optical disk memory is currently becoming popular with microcomputer systems. One of the commonly used disk memories with microcomputer systems is the floppy or flexible disk. The floppy disk is a flat, round piece of plastic coated with magnetically sensitive oxide material. The disk is provided with a protective jacket to prevent fingerprints or foreign matter from contaminating the disk's surface. The floppy disk is available in three sizes. These are the 8 inch, 5.25 inch, and 3.5 inch. The 8 inch floppy disk is not used in present systems. These days, the 5.25 inch and 3.5 inch are very popular. Also, the 3.5 inch floppy is replacing the 5.25 inch floppy in newer systems since it is smaller in size and does not bend easily. All floppy disks are provided with an off-center index hole that allows the electronic system reading the disk to find the start of a track and the first sector.

Hard disk memory is also frequently used with microcomputer systems. The hard disk, also known as the fixed disk, is not removable like the floppy disk.

A comparison of the some of the features associated with the hard disk and floppy disk is provided below:

Characteristic	Hard Disk	Floppy Disk
Size	5 Mbytes to several Gbytes	1.2 Mbytes typical for 5.25 inch floppy. 1.44 Mbytes typical for 3.5 inch floppy.
Rotational Speed	3600 rpm	300 rpm
Number of heads	May have up to 8 disk surfaces with up to two heads per surface	Two heads; One head for the upper surface and the other head for the lower surface

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. Sometimes a user may have to program a ROM in the field. For instance, in a fusible-link ROM, programmable read-only memory (PROM) is available. The main disadvantage of a PROM is that it cannot be reprogrammed.

Some ROMs can be reprogrammed, these are called Erasable Programmable Read-Only Memories (EPROMs).

In an EPROM, programs are entered using electrical impulses and the stored information is erased by using ultraviolet rays. Usually an EPROM is programmed by inserting the EPROM chip into the socket of a PROM programmer and providing program addresses and voltage pulses at the appropriate pins of the chip. Typical erase times vary between 10 and 30 minutes.

With advances in IC technology, it is possible to achieve an electrical means of erasure. These new ROMs are called Electrically Alterable ROMs (EAROMs) or Electrically Erasable PROMs (EEPROMs or E^2PROMs) and these ROM chips can be programmed even when they are in the circuit board. These memories are also called Read Mostly Memories (RMMs), since they have much slower write times than read times. Random Access Memories (RAMs) are read/write memories.

Information stored in random access memories will be lost if the power is turned off. This property is known as volatility and, hence, RAMs are usually called volatile memories. RAMs can be backed up by batteries for a certain period of time and are sometimes called nonvolatile RAMs. Stored information in a magnetic tape or magnetic disk is not lost when the power is turned off. Therefore, these storage devices are called nonvolatile memories. Note that a ROM is a nonvolatile memory.

Some RAMs are constructed using bipolar transistors, and the information is stored in the form of voltage levels in flip-flops. These voltage levels do not usually drift away, or decay. Such memories are called static RAMs because the stored information remains constant for some period of time.

On the other hand, in RAMs that are designed using MOS transistors, the information is held in the form of electrical charges in capacitors. Here, the stored charge has the tendency to decay. Therefore, a stored 1 would become a 0 if no precautions were taken. These memories are referred to as dynamic RAMs. In order to prevent any information loss, dynamic RAMs have to be refreshed at regular intervals. Refreshing means boosting the signal level and writing it back. This activity is performed by a hardware unit called "refresh logic" which can either be a separate chip or is contained in the microprocessor chip.

Since the static RAM maintains information in active circuits, power is required even when the chip is inactive or in standby mode. Therefore, static RAMs require large power supplies. Also, each static RAM cell is about four times larger in area than an equivalent dynamic cell; a dynamic RAM chip contains about four times as many bits as a static RAM chip using the same or comparable semiconductor technology. Figure 1.3 shows the subcategories of ROMs, RAMs, and their associated technologies.

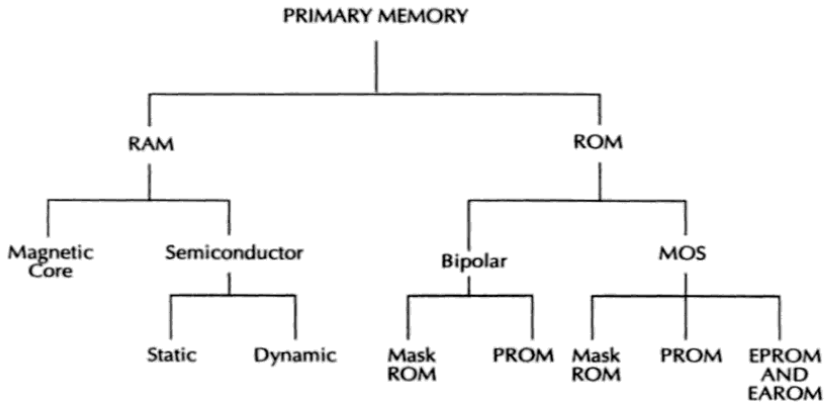


FIGURE 1.3 Subcategories of RAMs and ROMs.

Today, one megabit of data can be stored in an ordinary dynamic RAM chip. The data can be accessed in 80 nanoseconds or less. The RAM chip costs $5. In contrast, it takes 150 nanoseconds to access a one-megabit EEPROM which costs $150. Sixteen megabit DRAMs are very popular these days at a price of approximately 0.3 millicent per bit. Recently, IBM, Hitachi, Toshiba and others have introduced 64 mega-bit DRAMs. It is expected that giga-bit DRAMs will not be introduced until the next century.

In the mid 1980s, Toshiba Semiconductor invented flash memory. About the same time, Intel and Seeq Semiconductor were also working on flash memories. While each manufacturer implemented its flash memory differently, they operate in a similar way.

Like EPROMs and EEPROMs or EAROMs, flash memory is nonvolatile and reprogrammable. Flash memory is fabricated by using ETOX II (EPROM Tunnel Oxide) technology which is a combination of EPROM and EEPROM technologies. Flash memory is relatively inexpensive compared to EEPROM. A one megabit flash memory costs about $15. Flash memory can be reprogrammed electrically while embedded in the board. However, one can only change a sector or a block (consisting of multiple bytes) at a time.

Flash memory cells contain a single transistor like the EPROM cell. In contrast, a DRAM cell typically contains a transistor and a capacitor, an EEPROM cell contains two transistors while a static RAM cell requires four or six transistors.

The non-volatility and DRAM-like speed of flash memory are ideal for solid-state "disk" drives. Flash based disks do not have any disks or moving parts. Flash disks are very fast compared to most available disk drives.

Data can be accessed in 120 nanoseconds in flash memories while it takes 15 to 30 milliseconds to access data stored in today's typical hard disk. However, flash disks are limited to up to 40 megabytes in capacity whereas hard disk drives can store from 5 megabytes to several gigabytes.

A flash disk can be built from one or more flash-memory IC chips and some controlling logic devices. For example, to build a 512Kbyte flash disk, four one-megabit flash memory chips can be connected on a small card. An example of such a flash memory system is the Intel iMC004FLKA 4 Megabyte flash memory card. In addition to flash-disk hardware, software to manage files on a flash disk is required. The file system software handles creating and deleting files, changing the file sizes and formatting the flash disk. Microsoft offers flash file system software for the MS-DOS operating system.

The most severe limitation of flash disks has been its cost.However, the cost of flash ROM is significantly decreasing. In the future, high density flash memory is expected to be available at an inexpensive cost.

Flash memory can be programmed using either 5V or 12V. The 5V feature becomes more desirable for portable equipment where no 12V power is available. The speed, rugged construction, and lower power consumption of flash disks is ideal for laptop and notebook computers.

In summary, due to the high cost of flash disks, desktop computers will continue to use hard disk drives. Since flash memory combines the advantages of an EPROM's low cost with an EEPROM's ease of reprogramming, flash memories are being extensively used these days as a microcomputer's main non-volatile memory. An example of flash memory is the Intel 28F020 256K x 8 flash memory. By 1997, the cost of a megabyte of flash memory is expected to move from its current level of $120 to about $5. At that time, flash disks will be able to replace hard disks in many applications.

1.3.3.b 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. The memory map defining all memory addresses is determined. Note that the microprocessor's reset vector must be included in the memory map.

There are three types of techniques used for designing the main memory. These are linear decoding, full decoding/partial decoding and memory decoding using PALs. 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.

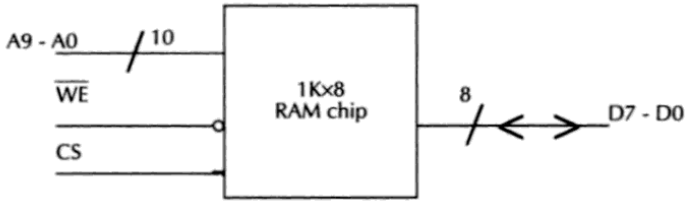


FIGURE 1.4 Typical Static RAM Chip.

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 specified 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: $\overline{WE}$ (write Enable) and CS (chip select). The truth table that describes the operation of this chip is shown in Table 1.1.

TABLE 1.1 Truth Table for 1K × 8 Static RAM

CS	$\overline{WE}$	MODE	Status of D7—D0	Power
L	X	Not selected	High impedance	Standby
H	L	Write	Acts as an input bus	Active
H	H	Read	Acts as an output bus	Active

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 $\overline{WE}$ 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.

1.3.3.b.i Linear Decoding. This technique uses the unused address lines of the microprocessor as chip selects for the memory chips. This method is used for small systems.

A simple way to connect an 8-bit microprocessor to a 6K 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 × 8 RAM chip. The remaining 6 high-order lines are used to select one of the 6 RAM chips. For example, if A15A14A13A12A11A10 = 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 have 64K bytes of RAM space, 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 programming 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 0000_{16} and address 8000_{16} are the same locations. This is called foldback and it wastes memory space.

1.3.3.b.ii Full/Partial Decoding. Difficulties such as the bus conflict and sparse address distribution are eliminated by the use of the full/partial 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:

A11	A10	Device Selected
0	0	RAM chip 0
0	1	RAM chip 1
1	0	RAM chip 2
1	1	RAM chip 3

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.

1.3.3.b.iii Memory Decoding by using Programmable Array Logic (PAL). A Programmable Array Logic (PAL) is similar to a ROM in concept except that it does not provide full decoding of the input lines. Instead, a PAL provides a partial sum of products which can be obtained via programming and saves a lot of space on the board. The PAL chip contains a fused programmable AND array and a fixed OR array. Note that in PLA (Programmable Logic Array) both AND and OR arrays are programmable. The AND and OR gates are fabricated inside the PAL without interconnections. The specific functions desired are implemented during programming via software. Programming of the PAL provides connections of the inputs of the AND gates and the outputs of the AND gates to the inputs of the OR gates. Therefore, the PAL implements sum of products of the inputs. PALs are used extensively these days with 32-bit microprocessors such as the Intel 80386/80486 and Motorola 68030/68040 for performing the memory decode function. PALs connect these microprocessors to memory, I/O and other chips without the use of any additional logic gates or circuits.

Each input has both true and complemented forms. A look at the NOR gate output $\varnothing_1$ indicates that there are two X connections at the inputs of this NOR gate. The three X inputs are wire-ANDed together with programming the PAL so that $\varnothing_1 = \left(I_0 \cdot \overline{I_1}\right) + I_2$.

The PAL chips are usually identified by a two-digit number followed by a letter and then a digit. The two-digit number specifies the number of input lines while the last digit defines the number of output lines. The fixed number of AND gates are connected to either an OR or a NOR gate. The letter 'H' indicates that the output gates are OR gates. The letter 'L' is used when the outputs are NOR gates.

As an example, the 10H8 provides eight OR gate outputs driven by ten AND gate inputs. The 10L8, on the other hand, is the same as the 10H8 except that the eight output gates are NOR gates.

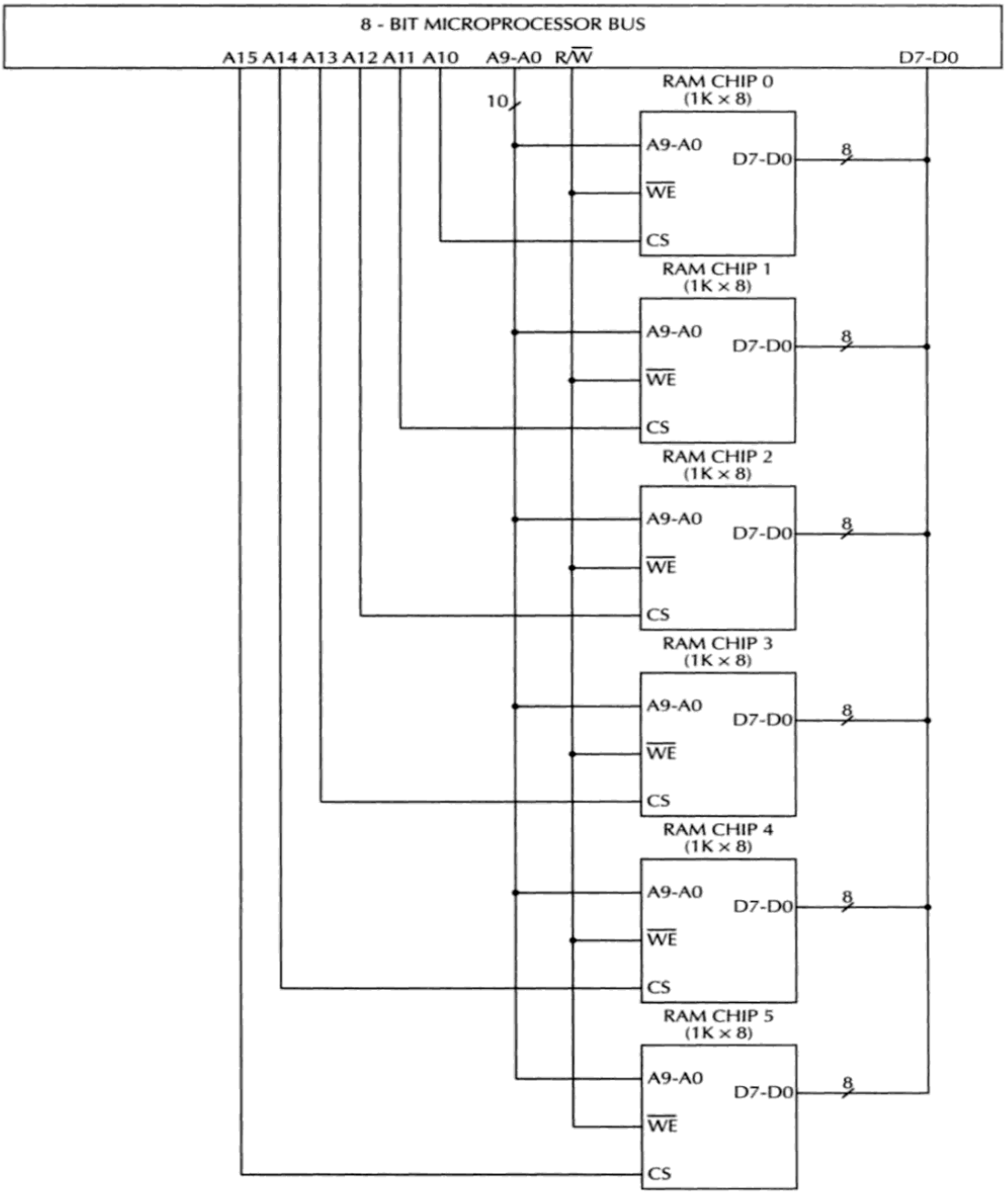


FIGURE 1.5 An 8-bit microprocessor interfaced to a 6K RAM system using the linear select decoding technique.

Some PALs provide additional features. The 16L8 includes tristate outputs. Each of the eight NOR gate outputs is driven internally by six AND gates. A seventh AND gate provides the enable signal for the tristate buffers.

The 16L8 is a popular PAL used with 32-bit microprocessors. The 16L8 is a 20-pin PAL manufactured using bipolar transistors. It has ten input pins (labeled I), two outputs (labeled O) and six programmable Input/Output (labeled I/O) lines. Using the programmable I/O lines, the number of input lines can be increased to a maximum of 16 and the number of output lines can be increased to 8.

Binary Address Pattern A15 A14 A13 A12 A11 A10 A9 A7 A6 A5 A4 A3 A2 A1 A0	Device Selected	Address Assignment in Hex
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 · 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1	RAM CHIP 0	0400 to 07FF
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 · 0 0 0 0 1 0 1 1 1 1 1 1 1 1 1	RAM CHIP 1	0800 to 0BFF
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 · 0 0 0 1 0 0 1 1 1 1 1 1 1 1 1	RAM CHIP 2	1000 to 13FF
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 · 0 0 1 0 0 0 1 1 1 1 1 1 1 1 1	RAM CHIP 3	2000 to 23FF
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 · 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1	RAM CHIP 4	4000 to 43FF
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 · 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1	RAM CHIP 5	8000 to 83FF

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

Programming PALs can be accomplished by first creating a file by using a text editor on a personal computer. The file should include information such as the pin assignments of the PAL and the boolean equation for the outputs. By inserting the PAL into the programming module included with the personal computer, the PAL can then be programmed with the PAL programming software provided with the personal computer.Note that PAL programming hardware and software are sold separately and not usually included with a personal computer.

1.3.3.c 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 very high. A combination of both off-board disk (secondary memory) and 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 mechanism must be able to transfer blocks of data efficiently, keep track of block usage, and replace them in a nonarbitrary way. The primary memory system must therefore be able to dynamically allocate memory space.

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, 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 (programmer's memory) and microprocessor physical (memory) address spaces. Since this device must manage memory use, 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 the memory management function 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. The addresses in a

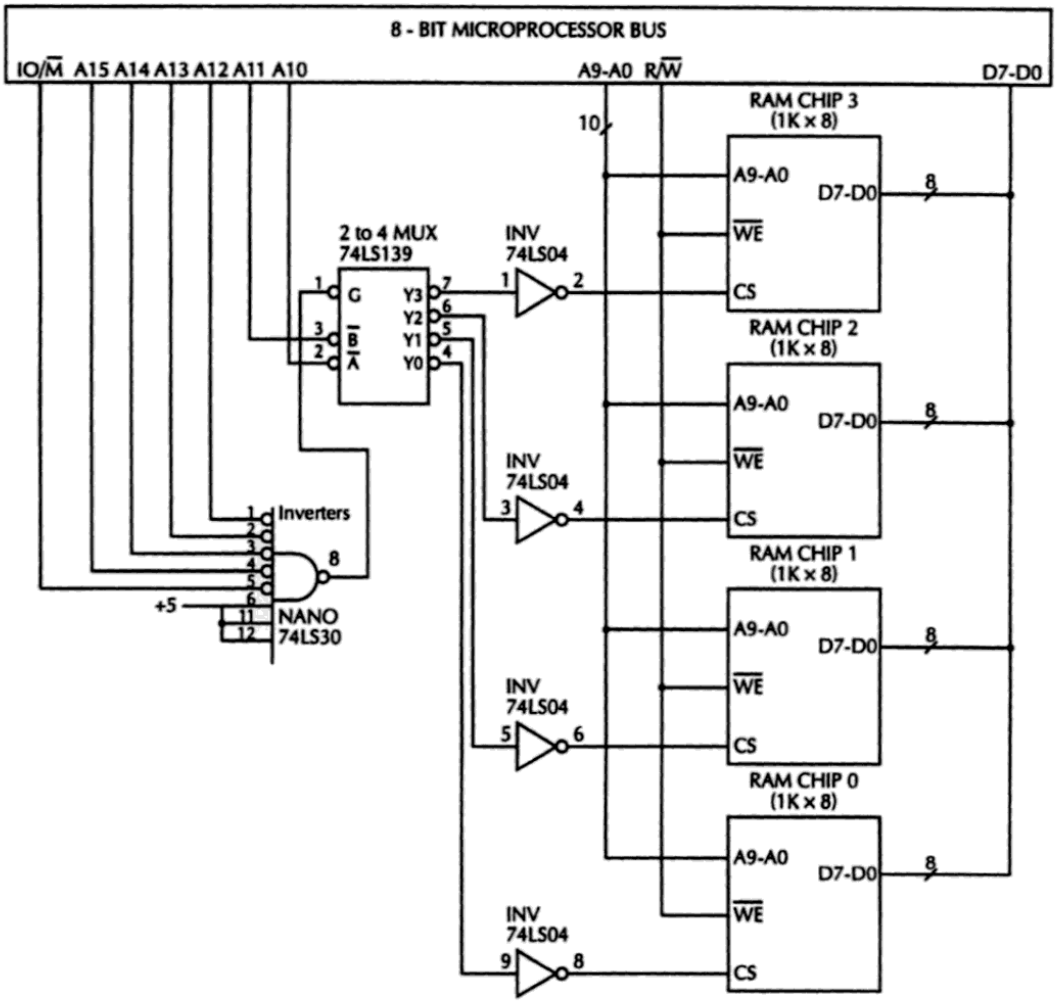


FIGURE 1.7 An 8-bit microprocessor interfaced to a 4K RAM system using a full/partial decoded addressing technique

Binary Address Pattern A15 A14 A13 A12 A11 A10 A9 A7 A6 A5 A4 A3 A2 A1 A0	Device Selected	Address Assignment in Hex
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	RAM CHIP 0	0000 to 03FF
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1		
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0	RAM CHIP 1	0400 to 07FF
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1		
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0	RAM CHIP 2	0800 to 0BFF
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1		
0 0 0 0 1 1 0 0 0 0 0 0 0 0 0	RAM CHIP 3	0C00 to 0FFF
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1		

FIGURE 1.8 Address map corresponding to the organization shown in Figure 1.7.

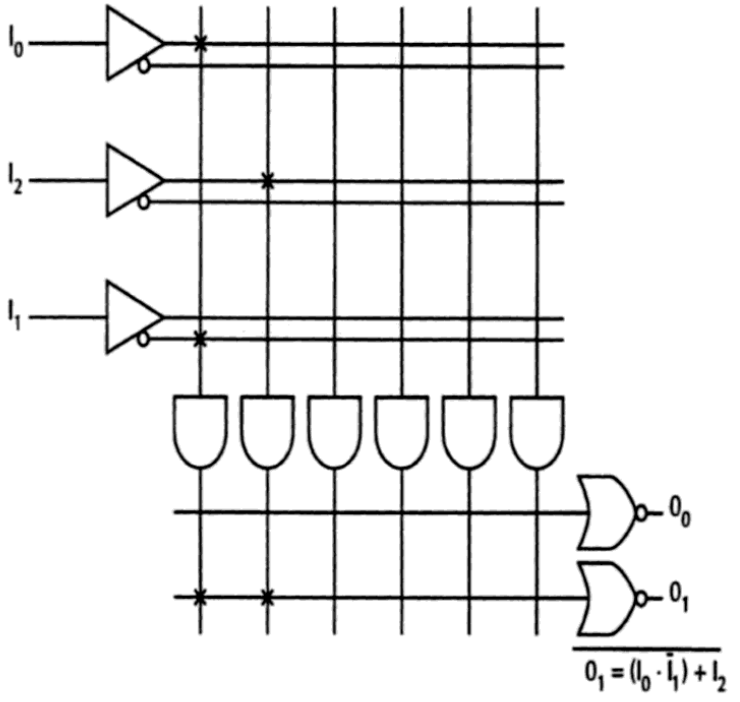


FIGURE 1.9 A typical PAL.

program are called logical addresses since they indicate the logical positions 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 1.10a
2. By adding an offset to each logical address to obtain the corresponding physical address as shown in Figure 1.10b

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 paged 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. Note that memory in excess of the main memory such as floppy disk storage is called virtual memory. The hardware included in the memory management 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.

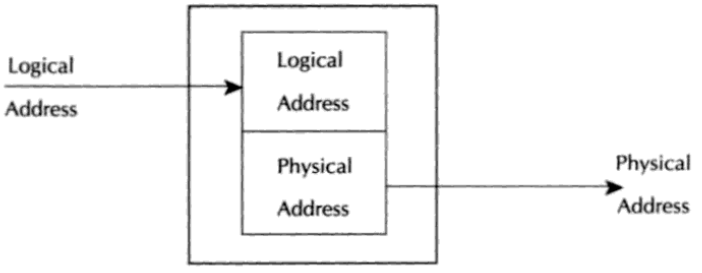


FIGURE 1.10a Address translation using substitution technique.

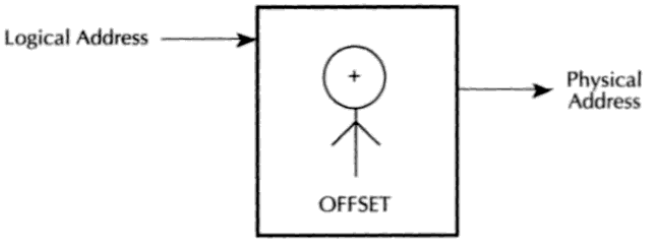


FIGURE 1.10b Address translation by offset technique.

If the main memory is full during transfer from secondary to main memory, it is necessary to remove a page from main memory to accommodate 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 program page to make room for a new page. This program 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 accessing the memory. On the other hand, 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 1.11. 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.

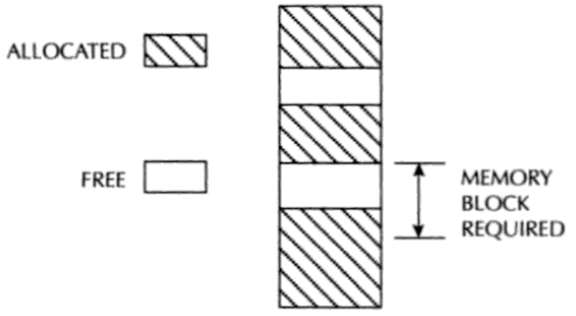


FIGURE 1.11 Memory fragmentation (external).

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 the program requesting access to a 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 permits the access. If the privilege level of the selector is lower than the descriptor, the MMU can interrupt the microprocessor informing of a privilege level violation. Therefore, the indirect technique of generating physical address provides a mechanism of protecting critical program sections in the operating system.

Paging divides the memory into equal-sized pages, it avoids the major problem of segmentation-external fragmentation. Since the pages are of the same size, when a new page is requested and an old one swapped out, the new one will always fit into the vacated space. However, a problem common to both techniques remains — internal fragmentation. Internal fragmentation is a condition where memory is unused but allocated due to memory block size implementation restrictions. This occurs when a module needs, say, 300 bytes and page is 1K bytes, as shown in Figure 1.12.

In the paged-segmentation method, each segment contains a number 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 mechanisms can operate 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.

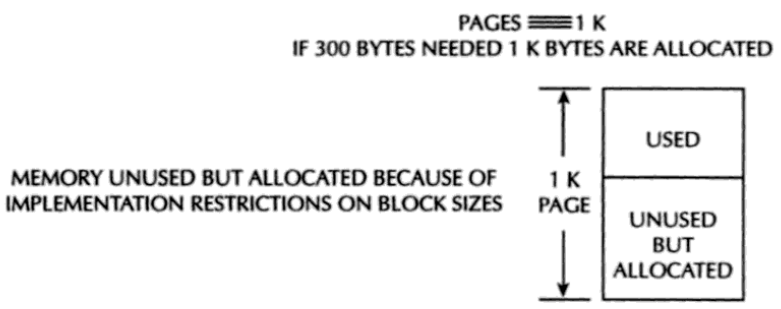


FIGURE 1.12 Memory fragmentation (internal).

1.3.3.d 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 microprocessor world. For example, an on-chip cache memory is implemented in Intel's 32-bit microprocessor, the 80486, and Motorola's 32-bit microprocessors, the MC 68020/68030/68040. The 80386 does not have on-chip cache but external cache memory can be interfaced to it. Studies have shown that typical programs spend most of their execution times in loops. This means that the addresses generated by a microprocessor have a tendency to cluster around a small region in the main memory. This phenomenon is known as locality of reference. The 32-bit microprocessor can execute the same instructions in a loop from the on-chip cache rather than reading them repeatedly from the external main memory. Thus the performance offered by 32-bit microprocessors is greatly improved.

The block diagram representation of a microprocessor system that employs an on-chip cache memory is shown in Figure 1.13. 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.

Cache hit means that the reference is found in the cache 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 bytes, 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). 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. Three 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 m bits wide and the cache memory address is n bits wide. Then the index field requires n bits and the tag field is (m — n) bits wide. The n-

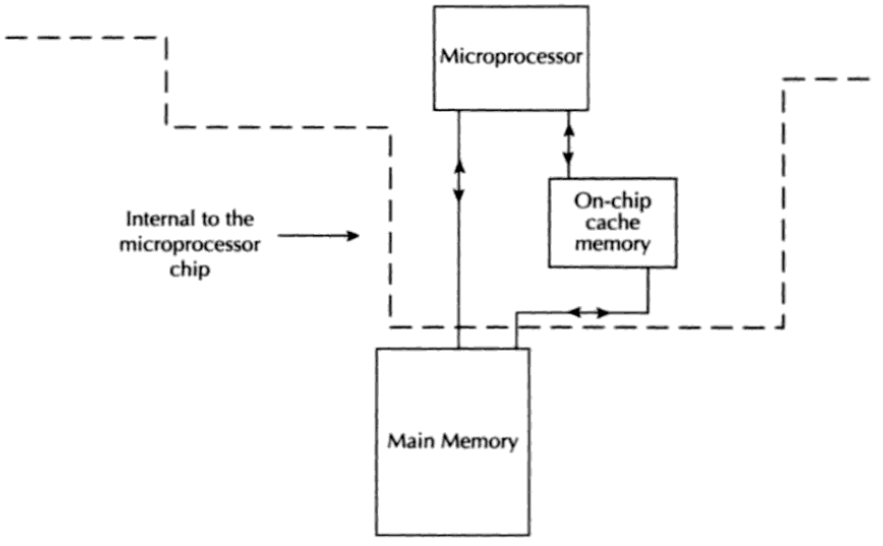


FIGURE 1.13 Memory organization of a computer system that employs a cache memory.

bit address accesses the code. Each word in the cache includes the data word and its associated tag. When the microprocessor generates an address for main memory, the index field is used as the address 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 then the required word is read from main 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 may occur if two or more words with addresses having the same index but different tags are accessed several times. This can be minimized by incorporating a larger cache.

The fastest cache memory utilizes an associative memory. The method is known as fully associative mapping. Each associative memory content contains main memory address and its content (data). When the microprocessor generates a main memory address, it is compared associatively (simultaneously) with all addresses in the associative memory. If there is a match, the corresponding data word is read from the associative cache memory and sent to the microprocessor. If a miss occurs, the main memory is accessed, and the address and its corresponding data are written to the 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-associative mapping is a combination of direct and associative mapping. Each cache word stores two or more main memory words using the same 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 address is then compared associatively (simultaneously) with all tags stored under the index. If a match occurs, the desired data word is read. If a miss occurs, the data word, along with its tag, is read from main 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. When a dirty word is to be replaced with a new word, the

dirty word is first copied into the main memory before it is overwritten by 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 cache data, the same alteration is made in the main memory copy of the altered cache data. 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 writes 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.

1.3.4 Input/Output (I/O)

This section describes the basic input and output techniques used by microcomputers to transfer data between the microcomputer and external devices. 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 circuitry between the microcomputer and I/O devices is necessary.

In a typical microcomputer system, the user gets involved with two types of I/O devices: physical I/O and logical 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, again by way of one or more I/O ports.

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

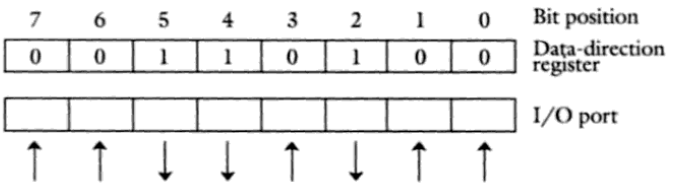
For a microcomputer with an operating system, the user works with virtual I/O devices. The user does not have to be familiar with the characteristics of the physical I/O devices. Instead, the user performs data transfers between the microcomputer and the physical I/O devices indirectly by calling the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

1.3.4.a 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 8355/8755 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 input or output. For the other type, all bits in the port can be set up as either all parallel input or output bits. Each port can be configured as an input or output port by another register called the command, or data-direction register. The port data register 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 (DDR). A bidirectional buffer (one input buffer and one output buffer) is connected at each bit of the port. A '1' written to a particular bit in DDR enables the output buffer while a '0' enables the input buffer connected at the corresponding bit of the port. As an example, if an 8-bit data-direction register contains 34_{16}, then the corresponding port is defined as follows:

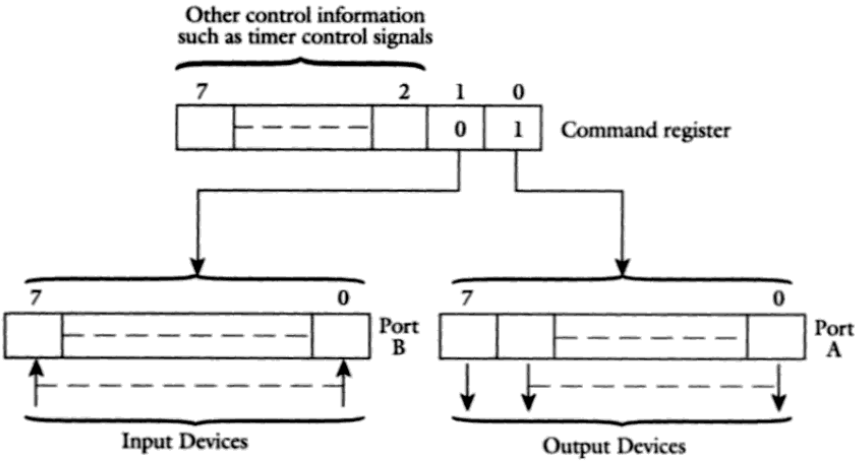


In the preceding example, since 34_{16} ($0011\ 0100_2$) is sent as an output into the data-direction register, bits 0, 1, 3, 6, and 7 of the port are set up as inputs, and bits 2, 4, and 5 of the port are defined as outputs. The microcomputer can then send outputs to external devices, such as LEDs, 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 8-bit microcomputer assumed here inputs the complete byte, including the bits to which LEDs are connected. While receiving input data from an I/O port, however, the microcomputer places a value, probably 0, at the bits 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 disregarded.

For parallel I/O, there is only one register, known as the command register, for all ports. A particular bit in the command register configures all bits in a 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,



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.

1.3.4.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 $IO/\overline{M}$ 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 $IO/\overline{M}$ HIGH, whereas memory-oriented instructions, such as LDA and STA, drive the IO/M to LOW. In standard I/O, the processor uses the $IO/\overline{M}$ pin to distinguish between I/O and memory. For 8-bit microprocessors, 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 microprocessor's $IO/\overline{M}$ control pin. The microprocessor uses the memory addresses (which may not exist in the microcomputer's physical memory) to represent I/O ports. The I/O ports are mapped into the microprocessor's main memory and, hence, are called *memory-mapped I/O.*

In memory-mapped I/O, the most significant bit (MSB) of the address may be 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. This reduces the microprocessor's addressing memory (main memory) by 50%. 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. Intel microprocessors can use either standard or memory-mapped I/O while Motorola microprocessors use only memory-mapped I/O. For example, with standard I/O, Intel 8086 uses IN AL, PortA, and OUT PortA, AL for inputing and outputing data. On the other hand, the 8086 uses memory-oriented instructions such as MOV AL, START, and MOV START, AL for inputing and outputing data. Note that START is an

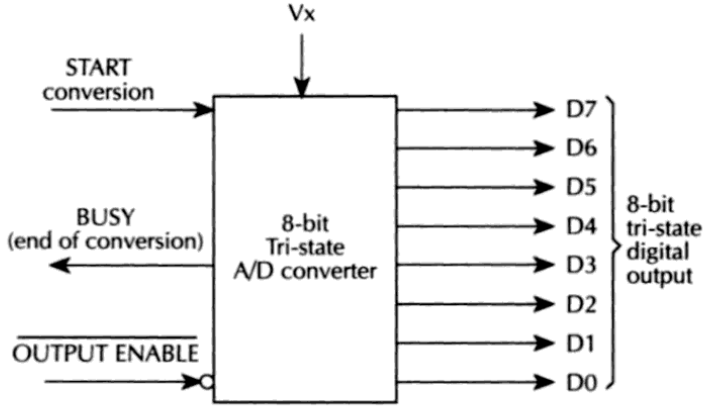
I/O port mapped as a memory address. Motorola, on the other hand, does not have any IN or OUT instructions and uses memory-oriented instructions for I/O operation.

1.3.4.c Unconditional and Conditional Programmed I/O

The microprocessor 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 microprocessor outputs data to an external device via handshaking. Data transfer occurs by the exchanging of control signals between the microprocessor and an external device. The microprocessor 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 microprocessor 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 transforms an analog voltage Vx 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 $(\overline{\text{OUTPUT ENABLE}})$ transfers the converter's output. A HIGH on the $(\overline{\text{OUTPUT ENABLE}})$ 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 1.14 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 the converter via bit 2 of port A. The microprocessor 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 microprocessor sends a LOW to the $(\overline{\text{OUTPUT ENABLE}})$ pin of the A/D converter. The microprocesor then inputs the converter's D0—D7 outputs via port B. If the conversion is not completed, the microprocessor waits in a loop checking for the BUSY signal to go HIGH.

1.3.4.d Typical Microcomputer Output Circuit

The microcomputer designer is often concerned with the output circuit because of the microcomputer's small output current drive capability. The tristate output circuit shown in Figure 1.15 typically is utilized by a microcomputer as the output circuit. This circuit uses totem pole-type output called a PUSH-PULL circuit providing low-output currents. Therefore, a current amplifier (buffer) is required to drive devices such as LEDs.

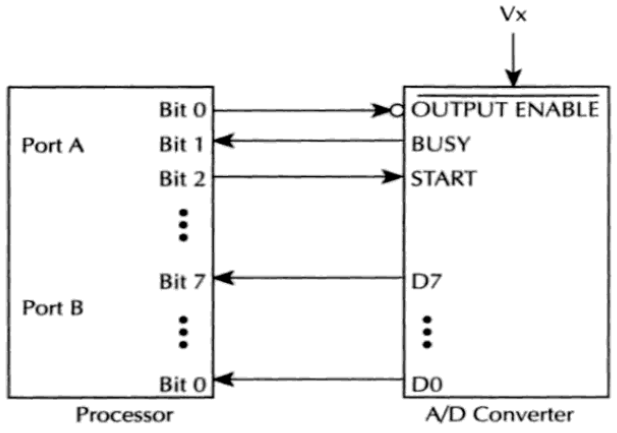


FIGURE 1.14 Interfacing an A/D converter to an 8-bit processor.

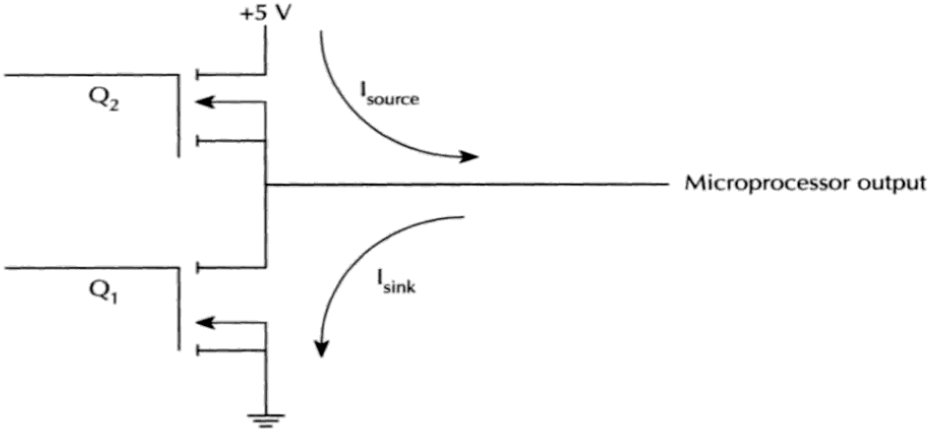


FIGURE 1.15 Digital microprocessor output circuit.

In the preceding figure, when Q_1 is ON, Q_2 is OFF and I_{sink} will flow from the external device into Q_1. Also, when Q_2 is ON, Q_1 is OFF and I_{source} will flow from Q_2 into the output device. Figure 1.16 shows a hardware interface circuit using the push-pull circuit for driving an external device such as LED.

Assume I_{source} to be 400 μA (usually represented by a negative sign such as $I_{OH} = -400$ μA; the negative sign indicates that the chip is losing current) with a minimum voltage V_A of 2.4 at point A, and that the LED requires 10 mA at 1.7 V. Therefore, a buffer such as a transistor is required at the output circuit to increase the current drive capability to drive the LED. In order to design the interface, the values of R_1,R_2 and minimum β of the transistor will be determined in the following:

$$R_1 = \frac{V_A - V_{BE}(Q_3)}{400\ \mu A} = \frac{2.4 - 0.7}{400\ \mu A} = 4.25\ k\Omega$$

Since $I_{LED} = 10$ μA at 1.7 V and assuming that V_{CE} (saturation) = 0 V,

$$R_2 = \frac{5 - 1.7 - V_{CE}(Q_3)}{10\ mA} = \frac{3.3}{10\ mA} = 330\ \Omega$$

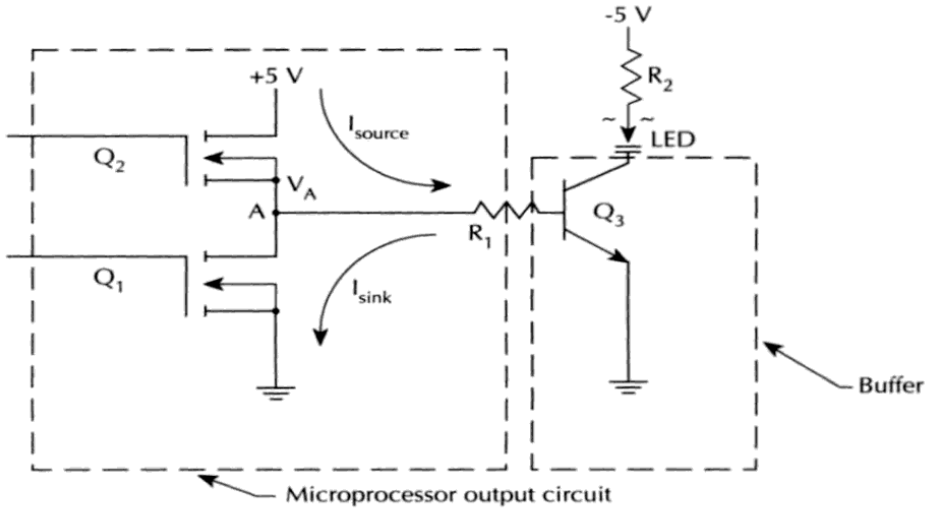


FIGURE 1.16 Microprocessor I/O interfaces for driving an LED.

Since $I_{source} = 400\ \mu A = I_B(Q_3)$, β for transistor Q_3 is

$$\beta = \frac{I_C(Q_3)}{I_B(Q_3)} = \frac{10\ mA}{400\ \mu A} = \frac{10 \times 10^{-3}}{400 \times 10^{-6}} = 25$$

Therefore, the interface design is complete, and a transistor with a mimimum of saturation β of 25 and $R_1 = 4.25\ k\Omega$ and $R_2 = 330\ \Omega$ is required. Note that a MOS outputs more sink current than source current. If sink current is used, the LED in Figure 1.16 can directly be connected to the microcomputer's output through an appropriate resistance. However, if the resistor value is not large enough, it may damage the transistor Q1.

1.3.4.e Interrupt Driven I/O

A disadvantage of conditional programmed 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 polled I/O and interrupt I/O techniques are 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 microprocessor chip. When the device needs an I/O transfer with the microcomputer, it activates the interrupt pin of the microprocessor 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 instruction 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 return address (saved in the stack before going to the service routine) in the program counter. Then, the microcomputer continues executing the main program. Note that subroutines and interrupts are handled in a similar way: subroutines are initiated via execution of an instruction such as subroutine CALL while interrupts are initiated via activation of the microprocessor's interrupt pin by the interrupting device.

1.3.4.e.i ***Interrupt Types.*** There are typically three types of interrupts: external interrupts, traps or internal interrupts, and software interrupts.

External interrupts are initiated through the microcomputer's interrupt pins by external devices such as A/D converters. 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 microprocessors, such as the Intel 8086, 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. Microprocessors 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 microprocessor. 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 microprocessors are provided with a maskable handshake interrupt. This interrupt is usually implemented by using two pins — INTR and $\overline{\text{INTA}}$. When the INTR pin is activated by an external device, the processor completes the current instruction, saves at least the current program counter onto stack, and generates an interrupt acknowledge ($\overline{\text{INTA}}$). In response to the $\overline{\text{INTA}}$, 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.

Internal interrupts, or traps, are activated internally by exceptional conditions such as overflow, division by zero, or execution of an illegal op-code. Traps are handled the same way as external interrupts. The user writes a service routine to take corrective measures and provide an indication to inform the user that an exceptional condition has occurred.

Many microprocessors include software interrupts, or system calls. When one of these instructions is executed, the microprocessor is interrupted and serviced similarly to external or internal interrupts. Software interrupt instructions are normally used to call the operating system. These instructions are shorter than subroutine calls, and no calling program is needed to know the operating system's address in memory. Software interrupt instructions allow the user to switch from user to supervisor mode. For some microprocessors, a software interrupt is the only way to call the operating system, since a subroutine call to an address in the operating system is not allowed.

1.3.4.e.ii ***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 microprocessor to another. With some microprocessors, 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.

1.3.4.e.iii ***Saving the Microprocessor Registers.*** When a microprocessor is interrupted, it saves at least the program counter on the stack so the microprocessor can return to the main program after executing the service routine. Some microprocessors save only one or two registers, such as the program counter and status register. Some other microprocessors save all microprocessor registers before going to the service routine. The user should know the specific registers the

microprocessor 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.

1.3.4.e.ivInterrupt Priorities. A microprocessor 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 interrupt by executing one general service routine for all devices. The priorities of devices are determined by the order in which the routine polls each device. The processor checks the status of each device in the general service routine, starting with the highest-priority device to service an interrupt. Once the processor determines the source of the interrupt, it branches to the service routine for the device.

In a daisy chain priority system, devices are connected in a daisy chain fashion to set up a priority system. Suppose one or more devices interrupt the processor. In response, the processor pushes at least the PC and generates an interrupt acknowledge ($\overline{\text{INTA}}$) signal to the highest-priority device. If this device has generated the interrupt, it will accept the $\overline{\text{INTA}}$. Otherwise it will pass the $\overline{\text{INTA}}$ onto the next device until $\overline{\text{INTA}}$ is accepted. Once accepted, the device provides a means for the processor to find an interrupt address vector by using external hardware. The daisy chain priority scheme is based on mostly hardware and is therefore faster than the polled interrupt.

1.3.4.f Direct Memory Access (DMA)

Direct Memory Access (DMA) is a technique that transfers data between a microcomputer's memory and an 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 lines 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 and releases the buses.

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 $\overline{\text{INHIBIT}}$ signal with the system clock. The system clock has the same frequency as the microprocessor clock.

The DMA controller controls the $\overline{\text{INHIBIT}}$ line. During normal operation, the $\overline{\text{INHIBIT}}$ line is HIGH, providing the microprocessor clock. When DMA operation is desired, the controller

makes the $\overline{\text{INHIBIT}}$ 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 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 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.

It should be mentioned that while using either block transfer or cycle stealing DMA in systems with dynamic RAMs, circuitry must be included for refreshing the dynamic RAM's during DMA transfer.

1.3.4.g Summary of Microcomputer I/O Methods

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

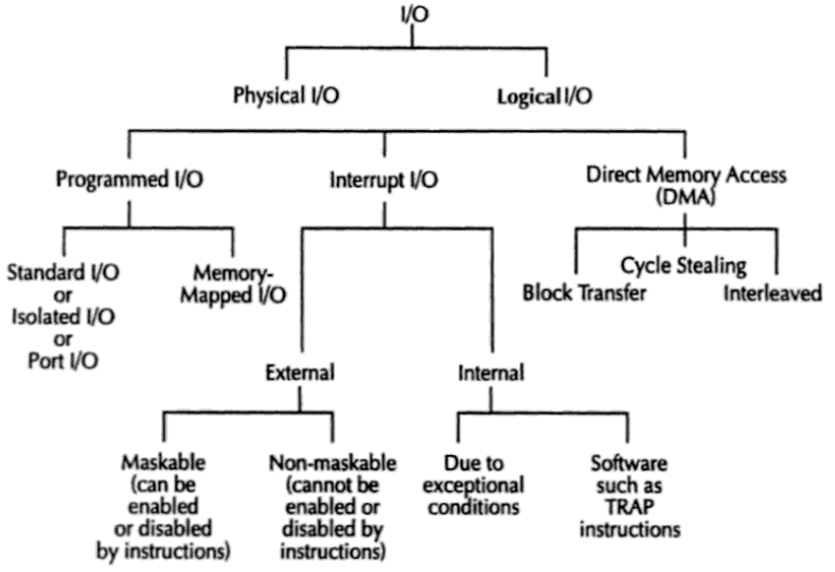


FIGURE 1.17 I/O structure of a typical microcomputer.

1.3.4.h Coprocessors

In typical 8-bit microprocessors such as the Intel 8085, technology places a limit on the chip area. As a 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 processor encounters one of these special instructions, it assumes the processing functions independent of the host microprocessor. The companion processor 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.

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 microprocessor. The coprocessor provides capabilities to the user without appearing to be hardware external to the main microprocessor.

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 microprocessor 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 microprocessor. The Intel 80387 floating-point coprocessor is of this type, as it monitors the instruction stream simultaneously with a main microprocessor such as the Intel 80386. 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 microprocessor to effect command and operand transfer.

In this approach, when the main microprocessor 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 microprocessor can read these data, and it tells the main microprocessor certain information such as whether additional information is required by the coprocessor to carry out the operation. If such data are required, the main microprocessor provides this; otherwise, the coprocessor carries out the operation concurrently with the main microprocessor 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.

Note that state-of-the-art 32-bit microprocessors such as the Intel 80486 and the Motorola 68040 implement coprocessor hardware such as floating point hardware and MMU on the microprocessor chip.

1.4 Microcomputer System Software and Programming Concepts

In this section, the basic concepts associated with system software and programming will be discussed.

1.4.1 System Software

Typical microcomputer system software provided in microcomputer development systems such as the HP64000 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 or a high-level language such as Pascal. The editor has commands to change, delete, or insert lines or characters. The text editor is a special type of editor that is used to enter 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 machine language in object code (binary).

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 into machine language. The advantage of high-level languages over assembly language are ease of readability and maintainability.

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 whole program into a sequence of machine instructions to be run later.

The debugger provides an interactive method of executing and debugging the user's software, one or a few instructions at a time, allowing the user to see the effects of small pieces of the program and thereby isolate programming errors.

An operating system performs resource management and human-to-machine translation functions. A resource may be the microprocessor, memory, or an I/O device. Basically, an operating system is another program that tells the machine what to do under a variety of conditions. Major operating system functions include efficient sharing of memory, I/O peripherals, and the microprocessor among several users. An operating system is

1. The interface between hardware and users
2. The manager of system resources in accordance with system policy to achieve system objectives

Operating systems for microcomputers became available when microcomputers moved from process control applications to the general-purpose computer applications. It was appropriate to write a process control program in assembly language because the microcomputer was required to perform dedicated real-time control functions. But when the microcomputers evolved to the point of controlling several I/O devices (disks, printers), an organized operating system was needed.

Note that many laboratory trainers such as the Intel SDK-86 include primitive operating systems called monitors to provide functions such as keyboard/display interfaces and program debugging features.

1.4.2 Programming Concepts

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

1.4.2.a 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 8086 assembly language program:

```
MOV   AL,5 ; Load AL reg with 5
ADD   AL,3 ; (AL) ← (AL) + 3
HLT              ; Halt processing
```

From this example, it is clear that the usage of mnemonics (in our example MOV, ADD, HLT are the mnemonics) improves 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 machine language (object) executable by the machine. This is illustrated in Figure 1.18.

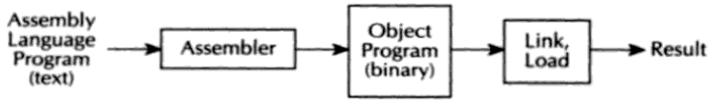


FIGURE 1.18 Assembly Process.

1.4.2.b High-Level Language Programming

Typical examples of high-level languages include FORTRAN, BASIC, Pascal, C 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 = N*(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.

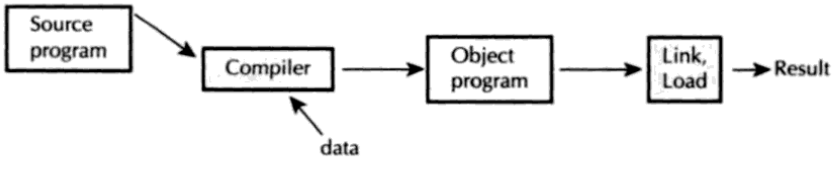


FIGURE 1.19 Compilation process.

1.4.2.c Which Programming Language to Choose?

Compilers normally provide inefficient machine codes because of the general guidelines which must be followed for designing them. It was found that a compiled high level language generates many more lines of machine code than an equivalent assembly language program. Therefore, the machine code program generated by the assembler will take up less memory space and also will execute much faster.

Therefore, although the C language includes I/O instructions, applications involving I/O are normally written in assembly language. One of the main uses of assembly language is in writing programs for real-time applications. Real time applications means that the task required by the application must be completed before any other input to the program can occur which will change its operation.

High level languages are normally used for applications which require extensive mathematical computations. These applications include optimizations and other non-real time control system applications.

1.5 Typical Microcomputer 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.

1.5.1 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
- Operand field(s)

The op-code field specifies how data are to be manipulated.The op-code field may contain data or a microprocessor register or a memory address. Consider the following instruction:

```
ADD          R1, R0
op-code field  operand field
```

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.

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

- Three-operand
- Two-operand
- One-operand
- Zero-operand

The 8-bit microprocessors include mostly one-operand instructions along with some zero- and two-operand instructions. In 16-bit microprocessors, two-operand instructions are predominant although some zero- and one-operand instructions are present. The 32-bit microprocessors include all four instruction formats.

1.5.2 Addressing Modes

The sequence of operations that a microprocessor 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 microprocessor accomplishes this task is by recognizing the *addressing mode* used in the instruction. 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 it is a zero-operand instruction. As an example, consider the zero-operand instruction CLC which clears the carry flag to zero.

Whenever an instruction contains data in the operand field, 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 MC68000.

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 in the operand field. For example, consider the following register mode intruction.

```
ADD R2, R3; R3 ← R2 + R3
```

This instruction uses register mode for both source and destination operands.

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 Effective Address (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]
```

Assume that the following configuration exists:

$$[R2] = 5000_{16}$$
$$[R3] = 4000_{16}$$
$$[5000] = 1256_{16}$$
$$[4000] = 4629_{16}$$

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_{16}.

1.5.3 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.

1.6 Basic Features of Microcomputer Development Systems

A microcomputer development system is a tool that allows the designer to develop, debug, and integrate error-free application software in microprocessor systems.

Development systems fall into one of two categories: systems supplied by the device manufacturer (non-universal systems) and systems built by after-market manufacturers (universal systems). The main difference between the two categories is the range of microprocessors that a system will accommodate. Non-universal systems are supplied by the microprocessor manufacturer (Intel, Motorola, RCA) and are limited to use for the particular microprocessor manufactured by the supplier. In this manner, an Intel development system may not be used to develop a Motorola-based system. The universal development systems (Hewlett-Packard, Tektronix) can develop hardware and software for several 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 central microcomputer. 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 microcomputer. 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).

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.

Each program (whether system software or user program) is stored in an ordered format on 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.

After the application system software has been completely developed and debugged, it needs to be permanently stored for execution in the target hardware. The EPROM programmer takes the machine code and programs it into an EPROM. Erasable/Programmable Read Only Memories (EPROMs) are more generally used in system development as they may be erased and reprogrammed if the program changes. EPROM 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 (ICEs). 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 emulator plugged into the microprocessor socket. The ICE will functionally and electrically act identically to the target processor with the exception that the ICE is under the control of development system 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. In a master/slave configuration, the master (host) processor controls all development functions such as editing, assembling, and so on. The master 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 slave microprocessor is typically connected to the user prototype via a 40-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 software executing on the target processor without stopping the emulation process. A benefit of the separate emulation facilities allows the master processor to be used for editing, assembling, and so on, while the slave processor continues the emulation. A designer may therefore start an emulation running, exit the emulator program, and at some future time return to the emulation program.

Another advantage of the separate bus architecture is that an operating system needs to be written only once for the master processor and will be used no matter what type of slave processor is being emulated. When a new slave processor is to be emulated, only the emulator probe needs to be changed.

A disadvantage of the master/slave architecture is that it is expensive. In single-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. The single processor systems are inexpensive.

The programs provided for 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 UNIX) handles I/O functions, memory management, and loading of programs from mass storage into RAM for execution.

The editor allows the user to enter the source code (either assembly language or some high-level language) into the development system.

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. 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.

Screen editors implement the editor commands in different fashions. 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.

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 or text 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 (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 8085 source code are

```
LABEL   | MNEMONIC | OPERAND | COMMENT
FIELD   | FIELD    | FIELD   | FIELD

        | MOV      | AL,5    | ; LOAD 5 INTO AL
        | ADD      | AL,2    | ; ADD 5 AND 2, STORE RESULT IN AL
```

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 a letter. After the letter, a combination of letters and numbers (called alphanumerics) may be used. For example, when grouping lines of code by function, a common alphabetic string may be used followed by a unique number for the label: L00P01, L00P02, L00P10, and so on.

A numeric quantity must start with a number, even though the number may be in hex (which may start with a letter). Most assemblers assume that a number is expressed in the decimal system and if another base is desired, a special code letter is used immediately following the number. The usual letter codes used are

B	binary
C	octal
H	hex (Motorola uses $ before the number)

To avoid confusion when hex quantities are used, a leading zero is inserted to tell the assembler that the quantity is a number and not a label (for example, the quantity FA in hex would be represented by 0FAH in the source code).

Assembler Directives

Assembler directives are instructions entered into the source code along with the assembly language. These directives do not get translated into object code but are used as special instructions to the assembler to perform some special functions. The assembler will recognize directives that assign memory space, assign addresses to labels, format the pages of the source code, and so on.

The directive is usually placed in the op-code field. If any labels or data are required by the directive, they are placed in the label or operand field as necessary.

Some common directives will now be discussed in detail.

a. ORIGIN (ORG). The ORG statement is used by the programmer when it is necessary to place the program in a particular location in memory. As the assembler is translating the source code, it keeps an internal counter (similar to the microprocessor program counter) that keeps track of the address for the machine code. The counter is incremented automatically and sequentially by the assembler. If the programmer wishes to alter the locations where the machine code is going to be located, the ORG statement is used.

For example, if it is desired to have a subroutine at a particular location in memory, such as 2000H, the statement ORG 2000H, would be placed immediately before the subroutine to direct the assembler to alter the internal program counter.

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 not at each of the ten locations where it is used in the program.

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.

For example, if a table of four values, 44H, 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   44
2001   34
2002   25
2003   D3
...
```

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 the 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,  2300H,  4BCAH
```

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

```
...
2500   00
2501   40
2502   00
2503   23
2504   CA
2505   4B
...
```

e. *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.

Several types of assemblers are available, the most common types are discussed below.

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 jumps forward), the assembler issues an error message.

b. ***Two-Pass Assembler.*** In the two-pass assembles, the source code is passed twice through the assembler. The first pass made through the source code is specifically for the purpose of assigning an address to all labels. When all labels have been stored in a table with the 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 *macros*. 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.

d. ***Cross Assemblers.*** A cross assembler may be of 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 8086 assembler may be written in FORTRAN and then the assembler may be executed on another machine such as the Motorola 6800.

e. ***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 EPROMs.

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 Out-Of-Circuit-Emulator (OCE) without having the final system hardware.

The usual software development tools provided with the development system are

- Single-step facility
- Breakpoint facility

A single-stepper simply allows the user to execute the program being debugged one instruction at a time. By examining the register and memory contents 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.

Single-stepper and breakpoint methods complement each other. The user may insert a breakpoint at the desired point and let the program execute up to that point. When the program stops at the breakpoint the user may use a single-stepper to examine the program one instruction at a time. Thus, the user can pinpoint the error in a 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.

PC-based workstations are extensively used as development systems.

1.7 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 1.20 shows a flowchart for the total development of a system. Notice that software and hardware development may occur in parallel to save time.

1.7.1 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 assembly language or high-level source code may now be written from the system flowchart.

The complete source code is then assembled. The assembler will check for syntax errors and print error messages to help in the correction of errors.

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 in debugging or it may be directed to the printer.

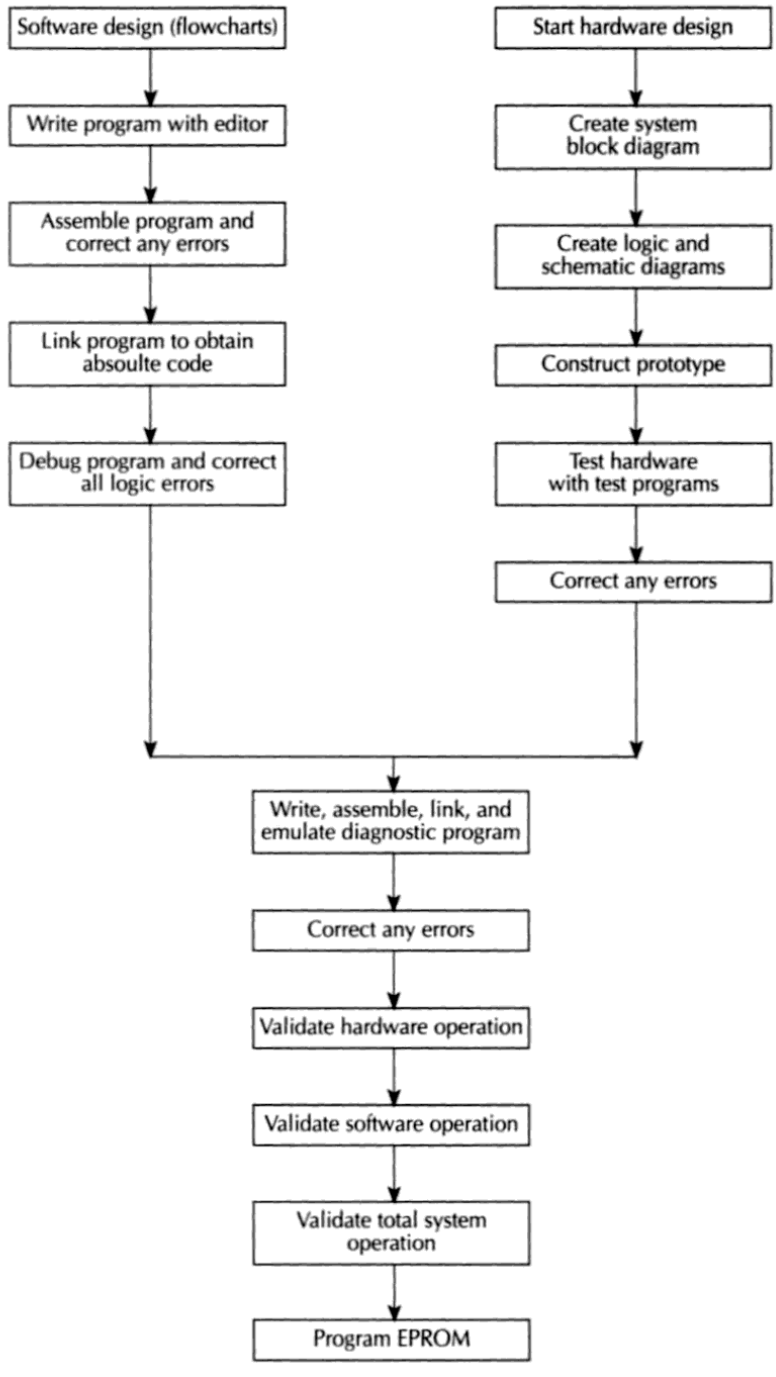


FIGURE 1.20 Microprocessor system development flowchart.

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 debugged using breakpoints or single-stepping.

1.7.2 Hardware Development

Working from the system specifications, a block diagram of the hardware must be developed.

The logic diagram and schematics may now be drawn using the block diagram as a guide.

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, all with test programs created for this purpose.

After the prototype has been debugged electrically, the development system in-circuit emulator may be used to check it functionally. The ICE will verify memory map, 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.

1.8 Typical Microprocessors

Intel and Motorola established the standard for future microprocessors.

Both Intel and Motorola introduced popular 8-bit microprocessors, namely, Intel 8080/8085, and Motorola 6800/6809 in the mid 1970's. These microprocessors are capable of directly addressing 64k of memory and were popular for several years. They are still being utilized in many inexpensive low-speed applications.

In 1978, Intel introduced its first 16-bit microprocessor, the Intel 8086. Several variations of the Intel 16-bit microprocessor family include the 8088, 80186/80188 and 80286. The 8088 is similar to the 8086 except that the 8088 has an 8-bit data bus. Both the 8086 and 8088 can run at 5 Mhz. They are packaged in 40 pins and can directly address one megabyte of memory. In 1982, Intel introduced an enhanced version of the 8086/8088 which is the Intel 80186/80188. The 80186 has a 16-bit data bus while the 80188 has an 8-bit data bus. The 80186/80188 can operate at 8 Mhz (80186), 6 Mhz (80186-6) and other frequencies. The 80186 integrates the 8086 and several new functional units into a single chip. The new on-chip components include a clock generator, two DMA channels, interrupt controller, address decoding, and three 16-bit programmable timers. Like the 8086, the 80186 can directly address one megabyte of memory. The 80186 is housed in a 68-pin leadless package.

The 80286 is Intel's other 16-bit microprocessor with on-chip memory protection capability primarily designed for multiuser/multitasking systems and introduced virtual memory.

Intel's 32-bit microprocessor family includes the 80386, 80486, and Pentium. The 80386 includes an on-chip memory management unit while the 80486 (DX and DX2) contains all the features of the 80386 along with floating-point hardware in the same chip. The Pentium, on the other hand, is a superscalar processor. This means that it includes dual pipelining and executes more than one instruction per cycle.

Soon after introduction of the Intel 8086, Motorola released its first 16-bit microprocessor, the 68000. It is housed in 64-pin or 68-pin package with a direct addressing capability of 16 megabytes. The 68008 is similar to the 68000 except that the 68000 has a 16-bit data bus while the 68008 has an 8-bit data bus. Also, the 68008 can directly address one megabyte of memory. The 68010 and 68012 are 16-bit microprocessors similar to the Intel 80286 and provide on-chip memory management support. Motorola CPU 32 is an enhanced 68000 along with many features of the 16-bit 68010 and the 32-bit 68020. The CPU 32 can support high level languages and is suited for controller applications. For example, in applications where power consumption is a consideration, the CPU32 forces the device into a low-power standby mode when immediate processing is not required upon execution of the LPSTOP instruction. Also, to maximize throughput for real-time applications, reference data is often precalculated and stored in memory for quick access upon execution of the TBL instruction.

Motorola's 32-bit microprocessor family includes the 68020, 68030, and 68040.

Since 1988, Intel, Morotola, and others have been introducing the RISC microprocessors. Some of these are the Intel 80960 family, the Motorola MC88100, the Apple/IBM/Motorola PowerPC, and Digital Equipment Corporation's Alpha 21164. The 80960 and 88100 are 32-bit microprocessors. They can directly address 4 gigabytes of memory. The PowerPC family, on the other hand, includes both 32-bit and 64-bit microprocessors. The 80960 and 88100 include a 32-bit data bus while the PowerPC contains a 64-bit data bus. The PowerPc is based on IBM PowerPC architecture and Motorola's 88100 bus interface design. These RISC microprocessors find extensive applications in embedded controls such as laser printers.

The PowerPC 601 outperforms and under-prices the Pentium but to be successful must build up its selection of software applications. Note that the Pentium had a flaw in its division algorithm caused by a problem with a Lookup table used in the division. Intel recently corrected this problem.

Tables 1.2a and 1.2b provide a brief description of Intel and Motorola's typical 16- and 32-bit microprocessors.

1.9 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 are described in the following.

1.9.1 Personal Workstations

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

The state-of-the-art workstations use 32-bit microprocessors to provide certain sophisticated functions such as IC layout, 3D graphics, and stress analysis.

1.9.2 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 reliable systems resulted in fault-tolerant systems. These systems use redundant computing units to provide reliable operation. However, the cost of fault-tolerant

Table 1.2a Intel 16-bit microprocessors

	8086	8088	80286
Introduced	June 1978	June 1978	Feb. 1982
Maximum clock speed	10 Mhz	10 Mhz	20 Mhz
On-chip cache	No	No	No
MIPS (Millions of Instructions Per Second)	0.33	0.33	1.2
Addressing Modes*	8	8	8
Transistors	29,000	29,000	134,000
Data Bus	16-bit	8-bit	16-bit
Address Bus	20-bit	20-bit	24-bit
Directly Addressable memory	One megabyte	One megabyte	Sixteen megabytes
Number of pins	40	40	68
Virtual Memory	No	No	Yes; One gigabyte per task
On-Chip memory management and protection	No	No	Yes
Math Coprocessor Interface	8087	8087	80287XL/XLT

*Addressing modes include register, immediate, and memory modes.

Table 1.2a Intel 32-bit microprocessors

	80386DX	80386SX	486DX	486SX	486DX2	Pentium
Introduced	October 1985	June 1988	April 1989	April 1991	March 1992	May 1993
Maximum clock speed	40 Mhz	33 Mhz	50 Mhz	25 Mhz	66 Mhz	100 Mhz
MIPS	6	2.5	20	16.5	40	112
Transistor	275,000	275,000	1.2 Millions	1.185 Millions	1.2 Millions	3.1 Millions
On-chip cache memory	Support chips avaliable	Support chips avaliable	Yes	Yes	Yes	Yes
Data Bus	32-bit	16-bit	32-bit	32-bit	32-bit	64-bit
Address Bus	32-bit	24-bit	32-bit	32-bit	32-bit	32-bit
Directly Addressable Memory	4 gigabytes	16 megabytes	4 gigabytes	4 gigabytes	4 gigabytes	4 gigabytes
Pins	132	100	168	168	168	273
Virtual Memory	Yes	Yes	Yes	Yes	Yes	Yes
On-chip Memory Management and Protection	Yes	Yes	Yes	Yes	Yes	Yes
Addressing Modes*	11	11	11	11	11	11
Floating-point	387DX	387SX	**	487SX	**	**

* Addressing modes include register, immediate, and memory modes.

** Indicates on-chip floating point.

Note that the 80386SL is also a 32-bit microprocessor with a 16-but data bus like the 80386SX (not listed in the table above). The 80386SL can run at a speed of up to 25 MHz and has a direct addressing capability of 32 megabytes. The 80386SL provides virtual memory support along with on-chip memory management and protection. It can be interfaced to the 80387SX to provide floating-point support. The 80386SL includes an on-chip disk controller hardware.

Table 1.2b Motorola 16-bit Microprocessors

	MC68000	MC68010	CPU32
Maximum Clock Speed	25 Mhz	25 Mhz	16.67 Mhz
Pins	64, 68	64,68	64, 68
Data Bus	16-bit	16-bit	16-bit
Virtual Memory	No	Yes	Yes
Directly Addressable Memory	16 Megabytes	16 Megabytes	16 Megabytes
Control Registers	None	3	3
Stack Pointers	2	2	2
Cache	No	No	No
Addressing Modes	14	14	16
Coprocessor Interface	No on-chip hardware	No on-chip hardware	No on-chip hardware

Table 1.2b Motorola 32-bit Microprocessors

	MC68020	MC68030	MC68040
Maximum Clock Speed	33 Mhz (8 Mhz min.)	33 Mhz (8 Mhz min.)	33 Mhz (8 Mhz min.)
Pins	114	118	179
Address Bus	32-bit	32-bit	32-bit
Addressing Modes	18	18	18
Maximum Addressable Memory	4 Gigabytes	4 Gigabytes	4 Gigabytes
Memory Management	By interfacing the 68851 MMU chip	On-chip MMU	On-chip MMU
Cache (on-chip)	Instruction cache	Instruction and data cache	Instruction and data cache
Floating Point	By interfacing 68881/68882 floating-point coprocessor chip	By interfacing 68881/68882 floating-point coprocessor chip	On-chip floating-point hardware

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, multiple 32-bit microprocessors in a redundant configuration outperform the VAXs. Thus, the 32-bit microprocessors provide efficient fault-tolerant systems.

1.9.3 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.

The state-of-the-art flight simulators 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 a car hood. These controllers are currently being designed using the small-sized 32-bit microprocessors to perform high-speed data manipulation and calculation.

1.9.4 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 manufactured the first 68020-based system robot control system.

1.9.5 Embedded Control

Embedded Control microprocessors, also called embedded controllers, are designed to be programmed to manage specific tasks. Once programmed, the embedded controllers can manage the functions of a wide variety of electronic products. Since the microprocessors are embedded in the host system, their presence and operation are basically hidden from the host system. Typical embedded control applications include office automation products such as copiers, laser printers, fax machines, and consumer electronics like VCRs, microwave ovens, and automative systems.

There are two types of embedded control applications, event control (real-time) and data control. Even control applications require distributed embedded controllers dedicated to single functions.

These applications include motor engine or instrument control system. In these applications, the controllers offer real-time response to support programs with small amounts of data. Typically, 8- and 16-bit single chip microcontrollers such as the Intel 8751/8096 and Motorola HC11/HC16 are used.

The 80186/80188 based microcontrollers are used for data control applications.

RISC microprocessor-based embedded controllers perform many different functions while handling large programs using large amounts of data. Applications such as laser printers require a high performance microprocessor with on-chip floating-point hardware. The RISC microprocessors are ideal for these types of applications.

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 Interpret FE_{16} as unsigned, signed, and floating-point numbers.

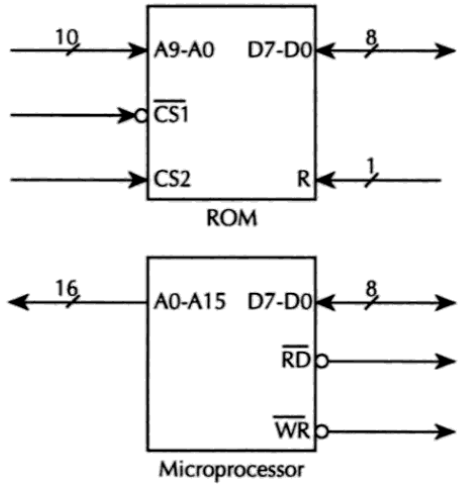
1.4 Determine the carry, sign, overflow, and zero flags for the following operation:

ADD 82A1H, 231FH

1.5 What are the basic difference between EPROM, EAROM, and flash ROM?

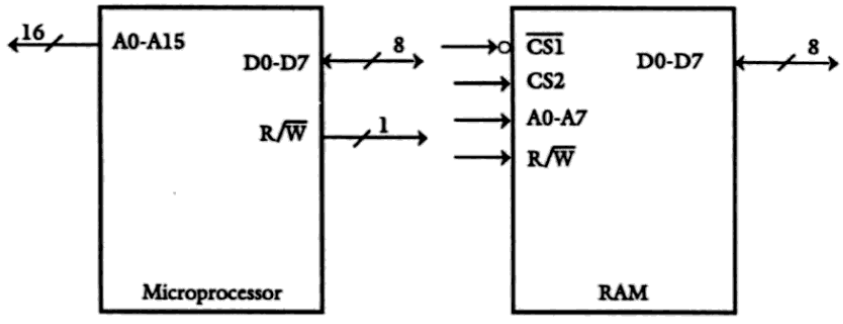
1.6 What is the difference between static and dynamic RAM?

1.7 Assume the following ROM chip and the microprocessor:



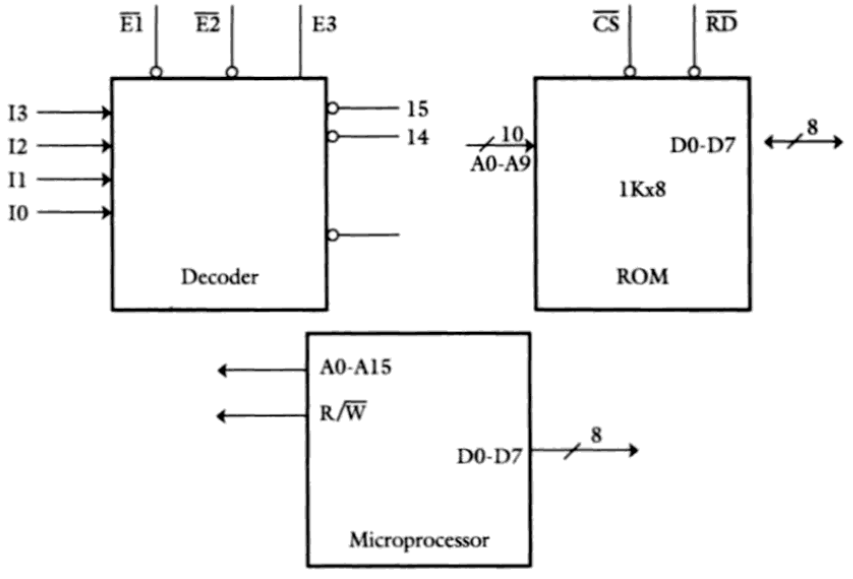
Connect the microprocessor to the ROM to obtain the following memory map: 0000_{16} through $03FF_{16}$. Use only the signals shown. Draw a neat logic diagram and analyze the memory map.

1.8 Assume the following microprocessor and the RAM chip:



Draw a neat logic diagram showing connections between the above microprocessor and the RAM chip using only the signals shown to include the memory map 0200_{16} through $02FF_{16}$. Use linear decoding.

1.9 Use the following chips to design a microcomputer memory to include the map 3000_{16} through $33FF_{16}$.



Draw a neat logic diagram.

1.10 What is meant by foldback in linear decoding?

1.11 Define the three basic types of I/O.

1.12 What is the difference between:

i) Standard and memory-mapped programmed I/O
ii) Maskable and nonmaskable interrupts
iii) Internal and external interrupts
iv) Block transfer and interleaved DMA
v) Polled and daisy chain
vi) One-pass and two-pass assemblers

1.13 Comment on the importance of the following architectural features in an operating system implementation:

a) Address translation
b) Protection
c) Program relocation

1.14 Explain clearly the differences between segmentation and paging. Can you think of a situation where it would be advantageous to define a virtual memory that is smaller than available physical memory?

1.15 What is the purpose of cache memory? Discuss briefly the various types of cache.

1.16 Discuss the main features of typical coprocessors.

1.17 What is the difference between software breakpoint and hardware breakpoint?

1.18 Discuss the basic features of microcomputer development systems.

1.19 Compare the typical features of the 16- and 32-bit microprocessors by Intel and Motorola.

1.20 What types of applications are the RISC microprocessors used for?

1.21 Discuss floating-point data formats supported by both Intel 80387 and Motorola 68881/6882.

2

INTEL 8085

This chapter describes hardware, software, and interfacing aspects of the Intel 8085. Topics include 8085 register architecture, addressing modes, instruction set, input/output, and system design.

2.1 Introduction

The Intel 8085 is an 8-bit microprocessor. The 8085 is designed using NMOS in 40-in DIP (Dual In-line Package). The 8085 can be operated from either 3.03 MHz maximum (8085A) or 5 MHz maximum (8085A-2) internal clock frequency.

The 8085 has three enhanced versions, namely, the 8085AH, 8085AH-2, and 8085AH-1. These enhanced processors are designed using the HMOS (High-density MOS) technology. Each is packaged in a 40-pin DIP like the 8085. These enhanced microprocessors consume 20% lower power than the 8085A. The internal clock frequencies of the 8085AH, 8085AH-2, and 8085AH-1 are 3, 5, 6 MHz, respectively. These HMOS 8-bit microprocessors are expensive compared to the NMOS 8-bit 8085A.

Figure 2.1 shows a simplified block diagram of the 8085 microprocessor. The accumulator connects to the data bus and the Arithmetic and Logic Unit (ALU). The ALU performs all data manipulation, such as incrementing a number or adding two numbers.

The temporary register feeds the ALU's other input. This register is invisible to the programmer and is controlled automatically by the microprocessor's control circuitry.

The flags are a collection of flip-flops that indicate certain characteristics of the result of the most recent operation performed by the ALU. For example, the zero flag is set if the result of an operation is zero. The zero flag is tested by the JZ instruction.

The instruction register, instruction decoder, program counter, and control and timing logic are used for fetching instructions from memory and directing their execution.

2.2 Register Architecture

The 8085 registers and status flags are shown in Figure 2.2.

The accumulator (A) is an 8-bit register. Most arithmetic and logic operations are performed using the accumulator. All I/O data transfers between the 8085 and the I/O devices are performed via the accumulator. Also, there are a number of instructions that move data between the accumulator and memory.

The B, C, D, E, H, and L are each 8 bits long. Registers H and L are the memory address register or data counter. This means that these two registers are used to store the 16-bit address

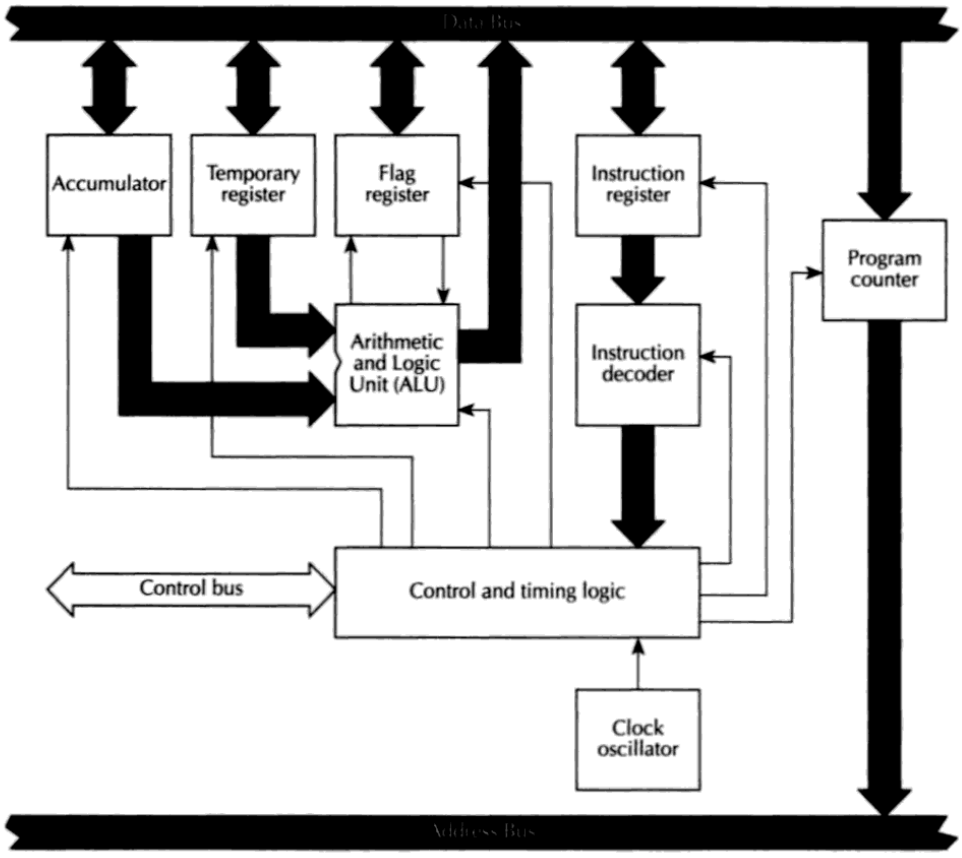


FIGURE 2.1 Simplified 8085 block diagram.

of 8-bit data being accessed from memory. This is the implied or register indirect addressing mode. There are a number of instructions, such as MOV reg, M, and MOV M, reg, which move data between any register and memory location addressed by H and L. However, using any other memory reference instruction, data transfer takes place between a memory location and the only 8085 register, the accumulator. The instruction LDAX B is a typical example.

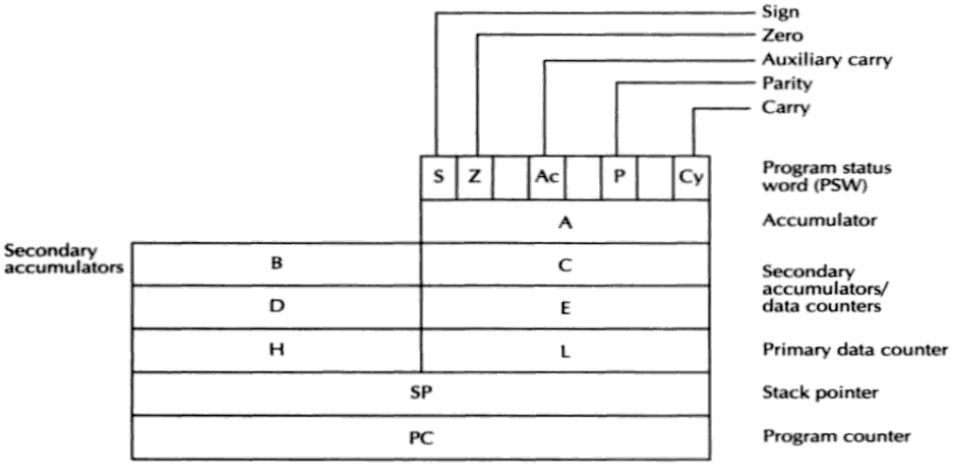


FIGURE 2.2 8085 microprocessor registers and status flags.

Registers B, C, D, and E are secondary accumulators or data counters. There are a number of instructions to move data between any two registers. There are also a few instructions that combine registers B and C or D and E, as a 16-bit data counter with high byte of a pair contained in the first register and low byte in the second. These instructions typically include LDAX B, LDAX D, STAX B, and STAX D, which transfer data between memory and the accumulator.

Each of these 8-bit registers can be incremented and decremented by a single byte instruction. There are a number of instructions which combine two of these 8-bit registers to form 16-bit register pairs as follows:

A	and	PSW
B	and	C
D	and	E
H	and	L
high-order byte		low-order byte

The 16-bit register pair obtained by combining the accumulator and the program status word (PSW) is used only for stack operations. Sixteen bit arithmetic operations use B and C, D and E, or H and L as 16-bit data registers.

The **program status word** consists of five status flags. These are described below.

The **carry flag** (Cy) reflects the final carry out of the most significant bit of any arithmetic operation. Any logic instruction resets or clears the carry flag. This flag is also used by the shift and rotate instructions. The 8085 does not have any CLEAR CARRY instruction. One way of clearing the carry will be by ORing or ANDing the accumulator with itself.

The **parity status flag** (P) is set to 1 if an arithmetic or logic instruction generates an answer with even parity, that is, containing an even number of 1 bits. This flag is 0 if the arithmetic or logic instruction generates an answer with odd parity, that is, containing an odd number of 1s.

The **auxiliary carry flag** (Ac) reflects any carry from bit 3 to bit 4 (assuming 8-bit data with bit 0 as the least significant bit and bit 7 as the most significant bit) due to an arithmetic operation. This flag is useful for BCD operations.

The **zero flag** (Z) is set to 1 whenever an arithmetic or logic operation produces a result of 0. The zero flag is cleared to zero for a nonzero result due to arithmetic or logic operation.

The **sign status flag** (S) is set to the value of the most significant bit of the result in the accumulator after an arithmetic or logic operation. This provides a range of -128_{10} to $+127_{10}$ (with 0 being considered positive) as the 8085's data-handling capacity.

The 8085 does not have an overflow flag. Note that execution of arithmetic or logic instructions in the 8085 affects the flags. All conditional instructions in the 8085 instruction set use one of the status flags as the required condition.

The **stack pointer** (SP) is 16 bits long. All stack operations with the 8085 use 16-bit register pairs. The stack pointer contains the address of the last data byte written into the stack. It is decremented by 2 each time 2 bytes of data are written or pushed onto the stack and is incremented by 2 each time 2 bytes of data are read from or pulled (popped) off the stack, that is, the top of the stack has the lowest address in the stack that grows downward.

The **program counter** (PC) is 16 bits long to address up to 64K of memory. It usually addresses the next instruction to be executed.

2.3 Memory Addressing

When addressing a memory location, the 8085 uses either register indirect or direct memory addressing. With register indirect addressing, the H and L registers perform the function of the memory address register or data counter; that is, the H, L pair holds the address of the data.

With this mode, data transfer may occur between the addressed memory location and any one of the registers A, B, C, D, E, H, or L.

Also, some instructions, such as LDAX B, LDAX D, STAX B, and STAX D, use registers B and C or D and E to hold the address of data. These instructions transfer data between the accumulator and the memory location addressed by registers B and C or D and E using the register indirect mode.

There are also a few instructions, such as the STA ppqq, which use the direct-memory addressing mode to move data between the accumulator and the memory. These instructions use 3 bytes, with the first byte as the OP code followed by 2 bytes of address.

The stack is basically a part of the RAM. Therefore, PUSH and POP instructions are memory reference instructions.

All 8085 JUMP instructions use direct or absolute addressing and are 3 bytes long. The first byte of this instruction is the OP code followed by a 2-byte address. This address specifies the memory location to which the program would branch.

2.4 8085 Addressing Modes

The 8085 has five addressing modes:

1. **Direct** — Instructions using this mode specify the effective address as a part of the instruction. These instructions contain 3 bytes, with the first byte as the OP code followed by 2 bytes of address of data (the low-order byte of the address in byte 2, the high-order byte of the address in byte 3). Consider LDA 2035H. This instruction loads accumulator with the contents of memory location 2035_{16}. This mode is also called the absolute mode.
2. **Register** — This mode specifies the register or register pair that contains data. For example, MOV B, C moves the contents of register C to register B.
3. **Register Indirect** — This mode contains a register pair which stores the address of data (the high-order byte of the address in the first register of the pair, and the low-order byte in the second). As an example, LDAX B loads the accumulator with the contents of a memory location addressed by B, C register pair.
4. **Implied or Inherent** — The instructions using this mode have no operands. Examples include STC (Set the Carry Flag).
5. **Immediate** — For an 8-bit datum, this mode uses 2 bytes, with the first byte as the OP code, followed by 1 byte of data. On the other hand, for 16-bit data, this instruction contains 3 bytes, with the first byte as the OP code followed by 2 bytes of data. For example, MVI B, 05 loads register B with the value 5, and LXI H, 2050H loads H with 20H and L with 50H.

A JUMP instruction interprets the address that it would branch to in the following ways:

1. **Direct** — The JUMP instructions, such as JZ ppqq, use direct addressing and contain 3 bytes. The first byte is the OP code, followed by 2 bytes of the 16-bit address where it would branch to unconditionally or based on a condition if satisfied. For example, JMP 2020 unconditionally branches to location 2020H.
2. **Implied or Inherent Addressing** — This JUMP instruction using this mode is 1 byte long. A 16-bit register pair contains the address of the next instruction to be executed. The instruction PCHL unconditionally branches to a location addressed by the H, L pair.

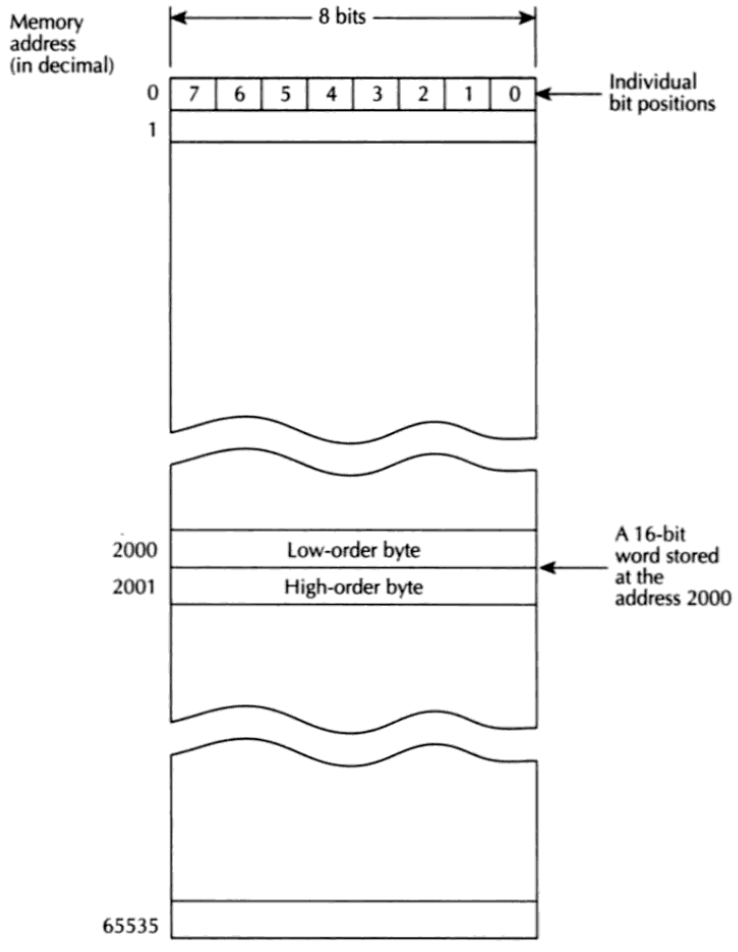


FIGURE 2.3 8085 addressing structures.

2.5 8085 Instruction Set

As mentioned before, the 8085 uses a 16-bit address. Since the 8085 is a byte-addressable machine, it follows that it can directly address 65,536 (2^{16}) distinct memory locations. The addressing structure of the 8085 processor is shown in Figure 2.3.

From this figure, we notice that two consecutive memory locations may be used to represent a 16-bit data item. However, according to the Intel convention, the high-order byte of a 16-bit quantity is always assigned to the high memory address.

The 8085 instructions are 1 to 3 bytes long and these formats are shown in Figure 2.4. The 8085 instruction set contains 74 basic instructions and supports conventional addressing modes such as immediate, register, absolute, and register indirect addressing modes.

Table 2.1 lists the 8085 instructions in alphabetical order; the object codes and instruction cycles are also included. When two instruction cycles are shown, the first is for "condition not met", while the second is for "condition met". Table 2.2 provides the 8085 instructions affecting the status flags. Note that not all 8085 instructions affect the status flags. The 8085 arithmetic and logic instructions normally affect the status flags.

In describing the 8085 instruction set, we will use the symbols in Table 2.3.

The 8085 move instruction transfers 8-bit data from one register to another, register to memory, and vice versa. A complete summary of these instructions is presented in Table 2.4.

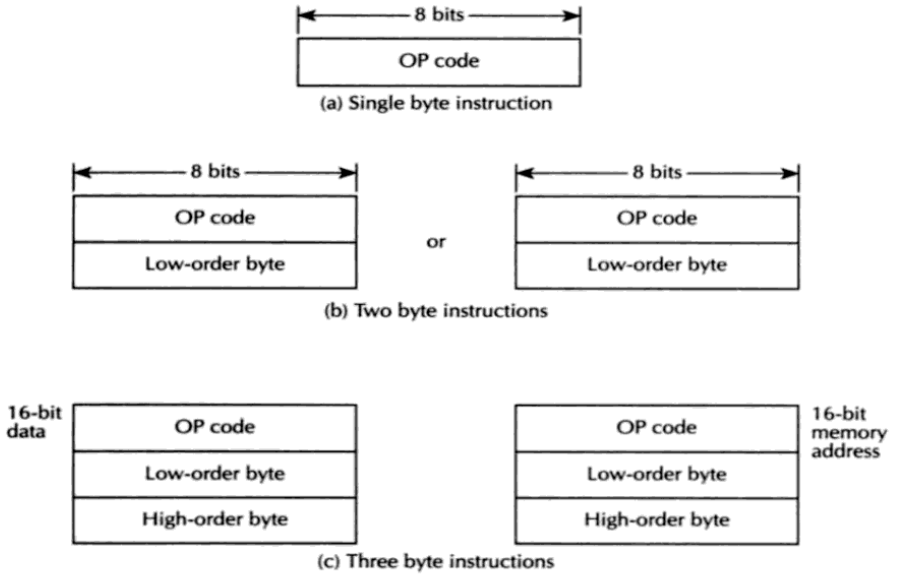


FIGURE 2.4 8085 instructions format.

Table 2.1 Summary of 8085 Instruction Set

Instruction	OP Code	Bytes	Cycles	Operations performed
ACI DATA	CE	2	7	[A] ← [A] + second instruction byte + [Cy]
ADC A	8F	1	4	[A] ←[A] + [A] + [Cy]
ADC B	88	1	4	[A] ← [A] + [B] + [Cy]
ADC C	89	1	4	[A] ← [A] + [C] + [Cy]
ADC D	8A	1	4	[A] ← [A] + [D] + [Cy]
ADC E	8B	1	4	[A] ← [A] + [E] + [Cy]
ADC H	8C	1	4	[A] ← [A] + [H] + [Cy]
ADC L	8D	1	4	[A] ← [A] + [L] + [Cy]
ADC M	8E	1	7	[A] ← [A] + [[H L]] + [Cy]
ADD A	87	1	4	[A] ← [A] + [A]
ADD B	80	1	4	[A] ← [A] + [B]
ADD C	81	1	4	[A] ← [A] + [C]
ADD D	82	1	4	[A] ← [A] + [D]
ADD E	83	1	4	[A] ← [A] + [E]
ADD H	84	1	4	[A] ← [A] + [H]
ADD L	85	1	4	[A] ← [A] + [L]
ADD M	86	1	7	[A] ← [A] + [[H L]]
ADI DATA	C6	2	7	[A] ← [A] + second instruction byte
ANA A	A7	1	4	[A] ← [A] ∧ [A]
ANA B	A0	1	4	[A] ← [A] ∧ [B]
ANA C	A1	1	4	[A] ← [A] ∧ [C]
ANA D	A2	1	4	[A] ← [A] ∧ [D]
ANA E	A3	1	4	[A] ← [A] ∧ [E]
ANA H	A4	1	4	[A] ← [A] ∧ [H]
ANA L	A5	1	4	[A] ← [A] ∧ [L]
ANA M	A6	1	4	[A] ← [A] ∧ [[H L]]
ANI DATA	E6	2	7	[A] ← [A] ∧ second instruction byte
CALL ppqq	CD	3	18	Call A subroutine addressed by ppqq
CC ppqq	DC	3	9/18	Call a subroutine addressed by ppqq if Cy = 1
CM ppqq	FC	3	9/18	Call a subroutine addressed by ppqq if S = 1
CMA	2F	1	4	[A] ← 1's complement of [A]
CMC	3F	1	4	[Cy] ← 1's complement of [Cy]
CMP A	BF	1	4	[A] – [A] and affects flags
CMP B	B8	1	4	[A] – [B] and affects flags

Table 2.1 Summary of 8085 Instruction Set (*continued*)

Instruction	OP Code	Bytes	Cycles	Operations performed
CMP C	B9	1	4	[A] – [C] and affects flags
CMP D	BA	1	4	[A] – [D] and affects flags
CMP E	BB	1	4	[A] – [E] and affects flags
CMP H	BC	1	4	[A] – [H] and affects flags
CMP L	BD	1	4	[A] – [L] and affects flags
CMP M	BE	1	7	[A] – [[H L]] and affects flags
CNC ppqq	D4	3	9/18	Call a subroutine addressed by ppqq if Cy = 0
CNZ ppqq	C4	3	9/18	Call a subroutine addressed by ppqq if Z = 0
CP ppqq	F4	3	9/18	Call a subroutine addressed by ppqq if S = 0
CPE ppqq	EC	3	9/18	Call a subroutine addressed by ppqq if P = 1
CPI DATA	FE	2	7	[A] – second instruction byte and affects flags
CPO ppqq	E4	3	9/18	Call a subroutine addressed by ppqq if P = 0
CZ ppqq	CC	3	9/18	Call a subroutine addressed by ppqq if Z = 1
DAA	27	1	4	Decimal adjust accumulator
DAD B	09	1	10	[HL] ← [HL] + [BC]
DAD D	19	1	10	[HL] ← [HL] + [DE]
DAD H	29	1	10	[HL] ← [HL] + [HL]
DAD SP	39	1	10	[HL] ← [HL] + [SP]
DCR A	3D	1	4	[A] ← [A] – 1
DCR B	05	1	4	[B] ← [B] – 1
DCR C	0D	1	4	[C] ← [C] – 1
DCR D	15	1	4	[D] ← [D] – 1
DCR E	1D	1	4	[E] ← [E] – 1
DCR H	25	1	4	[H] ← [H] – 1
DCR L	2D	1	4	[L] ← [L] – 1
DCR M	35	1	4	[[HL]] ← [[HL]] – 1
DCX B	0B	1	6	[BC] ← [BC] – 1
DCX D	1B	1	6	[DE] ← [DE] – 1
DCX H	2B	1	6	[HL] ← [HL] – 1
DCX SP	3B	1	6	[SP] ← [SP] – 1
DI	F3	1	4	Disable interrupts
EI	FB	1	4	Enable interrupts
HLT	76	1	5	Halt
IN PORT	DB	2	10	[A] ← [specified port]
INR A	3C	1	4	[A] ← [A] + 1
INR B	04	1	4	[B] ← [B] + 1
INR C	0C	1	4	[C] ← [C] + 1
INR D	14	1	4	[D] ← [D] + 1
INR E	1C	1	4	[E] ← [E] + 1
INR H	24	1	4	[H] ← [H] + 1
INR L	2C	1	4	[L] ← [L] + 1
INR M	34	1	4	[[HL]] ← [[HL]] +1
INX B	03	1	6	[BC] ← [BC] +1
INX D	13	1	6	[DE] ← [DE] + 1
INX H	23	1	6	[HL] ← [HL] + 1
INX SP	33	1	6	[SP] ← [SP] + 1
JC ppqq	DA	3	7/10	Jump to ppqq if Cy = 1
JM ppqq	FA	3	7/10	Jump to ppqq if S = 1
JMP ppqq	C3	3	10	Jump to ppqq
JNC ppqq	D2	3	7/10	Jump to ppqq if Cy = 0
JNZ ppqq	C2	3	7/10	Jump to ppqq if Z = 0
JP ppqq	F2	3	7/10	Jump to ppqq if S = 0
JPE ppqq	EA	3	7/10	Jump to ppqq if P = 1
JPO ppqq	E2	3	7/10	Jump to ppqq if P = 0
JZ ppqq	CA	3	7/10	Jump to ppqq if Z = 1
LDA ppqq	3A	3	13	[A] ← [ppqq]
LDAX B	0A	1	7	[A] ← [[BC]]
LDAX D	1A	1	7	[A] ← [[DE]]
LHLD ppqq	2A	3	16	[L] ← [ppqq],[H] ← [ppqq + 1]

Table 2.1 Summary of 8085 Instruction Set (*continued*)

Instruction	OP Code	Bytes	Cycles	Operations performed
LXI B	01	3	10	[BC] ← second and third instruction bytes
LXI D	11	3	10	[DE] ← second and third instruction bytes
LXI H	21	3	10	[HL] ← second and third instruction bytes
LXI SP	31	3	10	[SP] ← second and third instruction bytes
MOV A,A	7F	1	4	[A] ← [A]
MOV A,B	78	1	4	[A] ← [B]
MOV A,C	79	1	4	[A] ← [C]
MOV A,D	7A	1	4	[A] ← [D]
MOV A,E	7B	1	4	[A] ← [E]
MOV A,H	7C	1	4	[A] ← [H]
MOV A,L	7D	1	4	[A] ← [L]
MOV A,M	7E	1	7	[A] ← [[HL]]
MOV B,A	47	1	4	[B] ← [A]
MOV B,B	40	1	4	[B] ← [B]
MOV B,C	41	1	4	[B] ← [C]
MOV B,D	42	1	4	[B] ← [D]
MOV B,E	43	1	4	[B] ← [E]
MOV B,H	44	1	4	[B] ← [H]
MOV B,L	45	1	4	[B] ← [L]
MOV B,M	46	1	7	[B] ← [[HL]]
MOV C,A	4F	1	4	[C] ← [A]
MOV C,B	48	1	4	[C] ← [B]
MOV C,C	49	1	4	[C] ← [C]
MOV C,D	4A	1	4	[C] ← [D]
MOV C,E	4B	1	4	[C] ← [E]
MOV C,H	4C	1	4	[C] ← [H]
MOV C,L	4D	1	4	[C] ← [L]
MOV C,M	4E	1	7	[C] ← [[HL]]
MOV D,A	57	1	4	[D] ← [A]
MOV D,B	50	1	4	[D] ← [B]
MOV D,C	51	1	4	[D] ← [C]
MOV D,D	52	1	4	[D] ← [D]
MOV D,E	53	1	4	[D] ← [E]
MOV D,H	54	1	4	[D] ← [H]
MOV D,L	55	1	4	[D] ← [L]
MOV D,M	56	1	7	[D] ← [[HL]]
MOV E,A	5F	1	4	[E] ← [A]
MOV E,B	58	1	5	[E] ← [B]
MOV E,C	59	1	4	[E] ← [C]
MOV E,D	5A	1	4	[E] ← [D]
MOV E,E	5B	1	4	[E] ← [E]
MOV E,H	5C	1	4	[E] ← [H]
MOV E,L	5D	1	4	[E] ← [L]
MOV E,M	5E	1	7	[E] ← [[HL]]
MOV H,A	67	1	4	[H] ← [B]
MOV H,B	60	1	4	[H] ← [A]
MOV H,C	61	1	4	[H] ← [C]
MOV H,D	62	1	4	[H] ← [D]
MOV H,E	63	1	4	[H] ← [E]
MOV H,H	64	1	4	[H] ← [H]
MOV H,L	65	1	4	[H] ← [L]
MOV H,M	66	1	7	[H] ← [[HL]]
MOV L,A	6F	1	4	[L] ← [A]
MOV L,B	68	1	4	[L] ← [B]
MOV L,C	69	1	4	[L] ← [C]
MOV L,D	6A	1	4	[L] ← [D]
MOV L,E	6B	1	4	[L] ← [E]
MOV L,H	6C	1	4	[L] ← [H]
MOV L,L	6D	1	4	[L] ← [L]

Table 2.1 Summary of 8085 Instruction Set (*continued*)

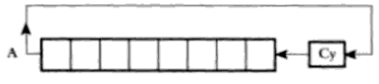
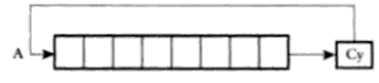
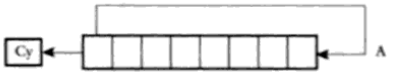
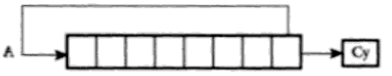
Instruction	OP Code	Bytes	Cycles	Operations performed
MOV L,M	6E	1	7	[L] ← [[HL]]
MOV M,A	77	1	7	[[HL]] ← [A]
MOV M,B	70	1	7	[[HL]] ← [B]
MOV M,C	71	1	7	[[HL]] ← [C]
MOV M,D	72	1	7	[[HL]] ← [D]
MOV M,E	73	1	7	[[HL]] ← [E]
MOV M,H	74	1	7	[[HL]] ← [H]
MOV M,L	75	1	7	[[HL]] ← [L]
MVI A, DATA	3E	2	7	[A] ← second instruction byte
MVI B, DATA	06	2	7	[B] ← second instruction byte
MVI C, DATA	0E	2	7	[C] ← second instruction byte
MVI D, DATA	16	2	7	[D] ← second instruction byte
MVI E, DATA	1E	2	7	[E] ← second instruction byte
MVI H, DATA	26	2	7	[H] ← second instruction byte
MVI L, DATA	2E	2	7	[L] ← second instruction byte
MVI M, DATA	36	2	10	[[HL]] ← second instruction byte
NOP	00	1	4	No operation
ORA A	B7	1	4	[A] ← [A]∨ [A]
ORA B	B0	1	4	[A] ← [A]∨ [B]
ORA C	B1	1	4	[A] ← [A]∨ [C]
ORA D	B2	1	4	[A] ← [A]∨ [D]
ORA E	B3	1	4	[A] ← [A]∨ [E]
ORA H	B4	1	4	[A] ← [A]∨ [H]
ORA L	B5	1	4	[A] ← [A] ∨ [L]
ORA M	B6	1	7	[A] ← [A] ∨ [[HL]]
ORI DATA	F6	2	7	[A] ← [A] ∨ second instruction byte
OUT PORT	D3	2	10	[specified port] ← [A]
PCHL	E9	1	6	[PCH]^a ← [H], [PCL]^a ← [L]
POP B	C1	1	10	[C] ← [[SP]], [SP] ← [SP] + 2 [B] ← [[SP] + 1]
POP D	D1	1	10	[E] ← [[SP]], [SP] ← [SP] + 2 [D] ← [[SP] + 1]
POP H	E1	1	10	[L] ← [[SP]], [SP] ← [SP] + 2 [H] ← [[SP] + 1]
POP PSW	F1	1	10	[A] ← [[SP] + 1], [PSW] ←[[SP]], [SP] ← [SP] + 2 [[SP]– 1] ← [B], [SP] ← [SP] – 2
PUSH B	C5	1	12	[[SP] – 2] ← [C]
PUSH D	D5	1	12	[[SP] – 1] ← [D], [[SP] – 2] ← [E] [SP] ← [SP] – 2
PUSH H	E5	1	12	[[SP] – 1] ← [H], [SP] ← [SP] – 2 [[SP] – 2] ← [L]
PUSH PSW	F5	1	12	[[SP] – 1] ← [A], [SP] ← [SP] – 2 [[SP] – 2] ← [PSW]
RAL	17	1	4	
RAR	1F	1	4	
RC	D8	1	6/12	Return if carry; [PC] ← [SP]]
RET	C9	1	10	[PCL]^a ← [[SP]], [SP] ← [SP] + 2 [PCH]^a ← [[SP] + 1]
RIM	20	1	4	Read interrupt mask
RLC	07	1	4	
RM	F8	1	6/12	Return if minus; [PC] ← [[SP]]

Table 2.1 Summary of 8085 Instruction Set (*continued*)

Instruction	OP Code	Bytes	Cycles	Operations performed
RNC	D0	1	6/12	Return if no carry; [PC] ← [[SP]]
RNZ	C0	1	6/12	Return if result not zero; [PC] ← [[SP]]
RP	F0	1	6/12	Return if positive; [PC] ← [[SP]], [SP] ← [SP] + 2
RPE	E8	1	6/12	Return if parity even; [PC] ← [[SP]], [SP] ← [SP] + 2
RPO	E0	1	6/12	Return if parity odd; [PC] ← [[SP]], [SP] ← [SP] + 2
RRC	0F	1	4	
RST0	C7	1	12	Restart
RST1	CF	1	12	Restart
RST2	D7	1	12	Restart
RST3	DF	1	12	Restart
RST4	E7	1	12	Restart
RST5	EF	1	12	Restart
RST6	F7	1	12	Restart
RST7	FF	1	12	Restart
RZ	C8	1	6/12	Return if zero; [PC] ← [[SP]]
SBB A	9F	1	4	[A] ← [A] – [A] – [Cy]
SBB B	98	1	4	[A] ← [A] – [B] – [Cy]
SBB C	99	1	4	[A] ← [A] – [C] – [Cy]
SBB D	9A	1	4	[A] ← [A] – [D] – [Cy]
SBB E	9B	1	4	[A] ← [A] – [E] – [Cy]
SBB H	9C	1	4	[A] ← [A] – [H] – [Cy]
SBB L	9D	1	4	[A] ← [A] – [L] – [Cy]
SBB M	9E	1	7	[A] ← [A] – [[HL]] – [Cy]
SBI DATA	DE	2	7	[A] ← [A] – second instruction byte – [Cy]
SHLD ppqq	22	3	16	[ppqq] ← [L], [ppqq + 1] ← [H]
SIM	30	1	4	Set interrupt mask
SPHL	F9	1	6	[SP] ← [HL]
STA ppqq	32	3	13	[ppqq] ← [A]
STAX B	02	1	7	[[BC]] ← [A]
STAX D	12	1	7	[[DE]] ← [A]
STC	37	1	4	[Cy] ← 1
SUB A	97	1	4	[A] ← [A] – [A]
SUB B	90	1	4	[A] ← [A] – [B]
SUB C	91	1	4	[A] ← [A] – [C]
SUB D	92	1	4	[A] ← [A] – [D]
SUB E	93	1	4	[A] ← [A] – [E]
SUB H	94	1	4	[A] ← [A] – [H]
SUB L	95	1	4	[A] ← [A] – [L]
SUB M	96	1	7	[A] ← [A] – [[HL]]
SUI DATA	D6	2	7	[A] ← [A] – second instruction byte
XCHG	EB	1	4	[D] ↔ [H], [E] ↔ [L]
XRA A	AF	1	4	[A] ← [A] ⊕ [A]
XRA B	A8	1	4	[A] ← [A] ⊕ [B]
XRA C	A9	1	4	[A] ← [A] ⊕ [C]
XRA D	AA	1	4	[A] ← [A] ⊕ [D]
XRA E	AB	1	4	[A] ← [A] ⊕ [E]
XRA H	AC	1	4	[A] ← [A] ⊕ [H]
XRA L	AD	1	4	[A] ← [A] ⊕ [L]
XRA M	AE	1	7	[A] ← [A] ⊕ [[HL]]
XRI DATA	EE	2	7	[A] ← [A] ⊕ second instruction byte
XTHL	E3	1	16	[[SP]] ↔ [L], [[SP] + 1]] ↔ [H]

[a]PCL — program counter low byte; PCH — program counter high byte.

Table 2.2 8085 Instructions Affecting the Status Flags

Instructions[a]	Status flags[b]				
	Cy	Ac	Z	S	P
ACI DATA	+	+	+	+	+
ADC reg	+	+	+	+	+
ADC M	+	+	+	+	+
ADD reg	+	+	+	+	+
ADD M	+	+	+	+	+
ADI DATA	+	+	+	+	+
ANA reg	0	1	+	+	+
ANA M	0	1	+	+	+
ANI DATA	0	1	+	+	+
CMC	+				
CMP reg	+	+	+	+	+
CMP M	+	+	+	+	+
CPI DATA	+	+	+	+	+
DAA	+	+	+	+	+
DAD rp	+				
DCR reg		+	+	+	+
DCR M		+	+	+	+
INR reg		+	+	+	+
INR M		+	+	+	+
ORA reg	0	0	+	+	+
ORA M	0	0	+	+	+
ORI DATA	0	0	+	+	+
RAL	+				
RAR	+				
RLC	+				
RRC	+				
SBB reg	+	+	+	+	+
SBB M	+	+	+	+	+
SBI DATA	+	+	+	+	+
STC	+				
SUB reg	+	+	+	+	+
SUB M	+	+	+	+	+
SUI DATA	+	+	+	+	+
XRA reg	0	0	+	+	+
XRA M	0	0	+	+	+
XRI DATA	0	0	+	+	+

[a]reg — 8-bit register; M — memory; rp — 16-bit register pair.

[b]Note that instructions which are not shown in the table do not affect the flags; + indicates that the particular flag is affected; 0 or 1indicates that these flags are always 0 or 1 after the corresponding instructions are executed.

Table 2.3 Symbols to be Used in 8085 Instruction Set

Symbol	Interpretation
r1, r2	8-bit register
rp	Register pair
data8	8-bit data
data16	16-bit data
M	Memory location indirectly addressed through the register pair H,L
addr16	16-bit memory address

Table 2.4 8085 MOVE Instructions

Instruction	Symbolic description	Addressing mode		Illustration	
		Source	Destination	Example	Comments
MOV r1, r2	(r1) ← (r2)	Register	Register	MOV A, B	Copy the contents of the register B into the register A
MOV r, M	(r) ← M((HL))	Register indirect	Register	MOV A, M	Copy the contents of the memory location whose address is specified in the register pair H,L into the A register
MVI r, data8	(r) ← data8	Immediate	Register	MVI A, 08	Initialize the A register with the value 08
MOV M, r	M((HL)) ← (r)	Register	Register indirect	MOV M, B	Copy the contents of the B register into the memory location addressed by H,L pair
MVI M, data8	M((HL)) ← data8	Immediate	Register indirect	MVI M, 07	Initialize the memory location whose address is specified in the register pair H,L with the value 07

Table 2.5 8085 Load and Store Instructions

Instruction	Symbolic description	Restriction	Addressing mode	Illustration	
				Example	Comments
LDA addr16	(A) ← M(addr16)	The destination is always the accumulator register	Absolute	LDA 2000H	Load the accumulator with the contents of the memory location whose address is 2000H
LHLD addr16	(L) ← M(addr16) (H) ← M(addr16+1)	The destination is always the register pair H,L	Absolute	LHLD 2000H	Load the H and L registers with contents of the memory locations 2001H and 2000H, respectively
LXI rp, data16	(rp) ← data16	rp may be HL, DE, BC, or SP	Immediate	LXI H, 2024H	H ← 20_{16} L ← 24_{16}
LDAX rp	(A) ← M((rp))	Destination is always the accumulator; also rp may be either B,C or D,E	Register indirect	LDAX B	Load the accumulator with the contents of the memory location whose address is specified with the register pair B,C
STA addr16	M(addr16) ← (A)	Source is always the accumulator register	Absolute	STA 2001H	Save the contents of the accumulator into the memory location whose address is 2001H
SHLD addr16	M(addr16) ← (L) M(addr16+1) ← (H)	The source is always the register pair H,L	Absolute	SHLD 2000	M(2000) ← (L) M(2001) ← (H)
STAX rp	M((rp)) ← (A)	The source is always the accumulator and the register pair may be B,C or D,E	Register indirect	STAX D	Save the contents of the accumulator register into the memory location whose address is specified with the register pair D,E
XCHG	[D] ↔ [H], [E] ↔ [L]		Inherent	XCHG	Exchanges [DE] with [HL]

Table 2.6 8085 Arithmetic Instructions

Operation	Instruction	Interpretation	Addressing mode	Illustration	
				Example	Comments
8-bit addition	ADD r	(A) ← (A) + (r)	Register	ADD B	(A) ← (A) + (B)
	ADI data8	(A) ← (A) + data8	Immediate	ADI 05	(A) ← (A) + 05
	ADD M	(A) ← (A) + M((HL))	Register indirect	—	—
8-bit addition with a carry	ADC r	(A) ← (A) + (r) + Cy	Register	ADC C	(A) ← (A) + (C) + Cy
	ACI data8	(A) ← (A) + data8 + Cy	Immediate	ACI 07	(A) ← (A) + 07 + Cy
	ADC M	(A) ← (A) + M((HL)) + Cy	Register indirect	—	—
8-bit subtraction	SUB r	(A) ← (A) – (r)	Register	SUB C	(A) ← (A) – (C)
	SUI data8	(A) ← (A) – data8	Immediate	SUI 03	(A) ← (A) – 03
	SUB M	(A) ← (A) – M((HL))	Register indirect	—	—
8-bit subtraction with a borrow	SBB r	(A) ← (A) – (r) – Cy	Register	SBB D	(A) ← (A) – (D) – Cy
	SBI data8	(A) ← (A) – data8 – Cy	Immediate	SBI 04	(A) ← (A) – 04 – Cy
	SBB M	(A) ← (A) – M((HL)) – Cy	Register indirect	—	—
16-bit addition	DAD rp	(HL) ← (HL) + (rp)	Register	DAD B	(HL) ← (HL) + (BC)
Decimal adjust	DAA	Convert the 8-bit number stored in the accumulator into BCD	Inherent	—	—
8-bit increment	INR r	(r) ← (r) + 1	Register	INR B	(B) ← (B) + 1
	INR M	M((HL)) ← M((HL)) + 1	Register indirect	—	—
16-bit increment[a]	INX rp	(rp) ← (rp) + 1	Register	INX D	(DE) ← (DE) + 1
8-bit decrement	DCR r	(r) ← (r) – 1	Register	DCR B	(B) ← (B) – 1
	DCR M	M((HL) ← M((HL)) – 1	Register indirect	—	—
16-bit decrement[a]	DCX rp	(rp) ← (rp) – 1	Register	—	

[a]rp = BC, DE, HL, or SP.

The 8085 instruction set also accomplishes the 8- and 16-bit data transfers using the load and store instructions. These instructions are summarized in Table 2.5.

From Table 2.5 notice that we adopt the following convention when we specify a register pair in the instruction.

Symbol	Register pairs used
B	B,C
D	D,E
H	H,L

Also, observe that in Table 2.5 the 8085 processor does not provide LDAX H instruction. This is because the same result can be obtained by using the MOV A,M instruction. The 8085 includes a one-byte exchange instruction, namely, XCHG. The XCHG exchanges the contents of DE with HL. That is, it performs the following operation:

$$[D] \leftrightarrow [H]$$
$$[E] \leftrightarrow [L]$$

The arithmetic instructions provided by the 8085 processor allow one to add (or subtract) two 8-bit data with or without carry (or borrow). The subtraction operation is realized by adding the two's complement of the subtrahend to the minuend. During the subtraction operation, the carry flag will be treated as the borrow flag. Table 2.6 lists these instructions. For some instructions such as ADD M, examples and comments are not included. This is due to limited space in the table.

As far as logical operations are concerned, the 8085 includes some instructions to perform traditional Boolean operations such as AND, OR, EXCLUSIVE-OR. In addition, instructions are available to complement the accumulator and to set the carry flag. The 8085 COMPARE instructions subtract the specified destination from the contents of the

Table 2.7 8085 Logical Instructions

Operation	Instruction	Interpretation	Addressing mode	Illustration	
				Example	Comments
Boolean AND	ANA r	(A) ← (A) ∧ (r)	Register	ANA B	(A) ← (A) ∧ (B)
	ANI data8	(A) ← (A) ∧ data8	Immediate	ANI 0FH	(A) ← (A) ∧ 00001111_2
	ANA M	(A) ← (A) ∧ M((HL))	Register indirect	—	—
Boolean OR	ORA r	(A) ← (A) ∨ (r)	Register	ORA C	(A) ← (A) ∨ (C)
	ORI data8	(A) ← (A) ∨ data8	Immediate	ORI 08H	(A) ← (A) ∨ 00001000_2
	ORA M	(A) ← (A) ∨ M((HL))	Register indirect	—	—
Boolean EXCLUSIVE-OR	XRA r	(A) ← (A) ⊕ (r)	Register	XRA A	(A) ← (A) ⊕ (A)
	XRI data8	(A) ← (A) ⊕ data8	Immediate	XRI 03H	(A) ← (A) ⊕ 00000011_2
	XRA M	(A) ← (A) ⊕ M((HL))	Register indirect	—	—
Compare	CMP r	(A) – (r) and affect flags	Register	CMP D	Compare (A) register with (D) register
	CPI data8	(A) – data8 and affect flags	Immediate	CPI 05	Compare (A) with 05
	CMP M	(A) – M((HL)) and affect flags	Register indirect	—	—
Complement	CMA	(A) ← ($\overline{A}$)	Inherent	—	—
Bit manipulation	STC	Cy ← 1 (set carry to 1)	Inherent	—	—
	CMC	Cy ← Cy′ (complement carry flag)	Inherent	—	—

accumulator and affect the status flags according to the result. However, in this case the result of the subtraction is not provided in the accumulator. All 8085 logical instructions are specified in Table 2.7. For some instructions in this figure, examples and comments are not provided. This is due to limited space in the table.

The AND instruction can be used to perform a masking operation. If the bit value in a particular bit position is desired in a word, the word can be logically ANDed with appropriate masking data to accomplish this. For example, the bit value at bit 3 of the word 1011 $X011_2$ can be determined as follows:

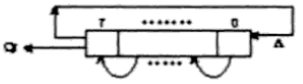
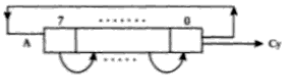
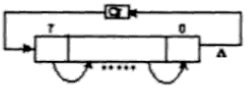
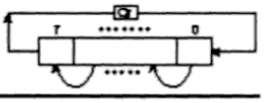
	1011	x011	Word
AND	0000	1000	Masking Data
	0000	x000	Result

If the bit value X at bit 3 is 1, then the result is nonzero (Z = 0); otherwise the result is zero (Z = 1). The Z-flag can be tested using JZ (jump if Z = 1) or JNZ (jump if Z = 0) to determine whether X = 0 or 1. The AND instruction can also be used to determine whether a binary number is odd or even by checking the least significant bit (LSB) of the number (LSB = 0 for even and LSB = 1 for odd). XRA instruction can be used to find ones compliment of a binary number by exclusive-ORing the number with all ones as follows:

	1010	1001	Original number
XOR	1111	1111	
	0101	x110	Ones complement

One of the applications of the compare instruction is to find a match in an array. The number to be matched can be loaded in the accumulator and compared with each element in the array. The JZ (jump if Z = 1) can then be executed to find the match. If the subtraction instruction is used in place of the COMPARE, the number to be matched in the accumulator

TABLE 2.8 8085 Rotate Instructions

Instruction	Interpretation	Illustration
RLC	Rotate left accumulator by one position without the carry flag Cy	
RRC	Rotate right accumulator by one position without the carry flag Cy	
RAR	Rotate right accumulator by one position through the carry flag	
RAL	Rotate left accumulator by one position through the carry flag	

will be lost after each subtraction, and therefore the number needs to be loaded for the next subtraction.

The 8085 instruction set includes rotating the contents of the A register to the left or right without or through the carry flag. These instructions are listed in Table 2.8.

In the 8085, only the absolute mode branch instruction is of the form JMP addr16. There is also a one-byte implied unconditional jump instruction, namely, PCHL. The PCHL loads [H] into PC high byte and [L] into PC low byte. That is, PCHL performs an unconditional JUMP to a location addressed by the contents of H and L. The general format of an 8085 conditional branch instruction is J <condition code> addr16 where the condition code may represent one of the following conditions:

Conditional jumps	Condition	Comment
JZ	Z = 1	Z flag is set (result equal to zero)
JNZ	Z = 0	Z flag is reset (result not equal to zero)
JC	Cy = 1	Cy flag is set
JNC	Cy = 0	Cy flag is reset
JPO	P = 0	The parity is odd
JPE	P = 1	The parity is even
JP	S = 0	S flag is reset (or the number is positive)
JM	S = 1	S flag is set (or the number is negative, or minus)

For example, the following instruction sequence causes a branch to the memory address 2000_{16} only if the contents of the A and B registers are equal:

```
CMP B
JZ  2000H
```

In the 8085, the subroutine call instruction is of the form

```
CALL addr16
```

The instruction RET transfers the control to the caller, and it should be the last instruction of the subroutine. Both instructions use the PC and SP for subroutine linkage.

For example, consider the following:

Main program		Subroutine
	SUB	—
—		—
—		—
CALL SUB		—
START —		—
—		—
—		RET
—		

The call SUB instruction pushes or saves the current PC contents (START which is the address of the next instruction) onto the stack and loads PC with the starting address of the Subroutine (SUB) specified with the CALL instruction. The RET instruction at the end of the subroutine pops or reads the address START (saved onto the stack by the CALL instruction) into PC and transfers control to the right place in the main program.

There are a number of conditional call instructions. These include:

```
CC  addr (call if Cy = 1)
CNC addr (call if Cy = 0)
CZ  addr (call if Z = 1)
CNZ addr (call if Z = 0)
CM  addr (call if S = 1)
CP  addr (call if S = 0)
CPE addr (call if P = 1)
CPO addr (call if P = 0)
```

Also, there are a number of conditional return instructions. These include:

```
RC  (Return if Cy = 1)
RNC (Return if Cy = 0)
RZ  (Return if Z = 1)
RNZ (Return if Z = 0)
RPE (Return if P = 1)
RPO (Return if P = 0)
RM  (Return if S = 1)
RP  (Return if S = 0)
```

There are eight one-byte call instructions (RST 0 to 7) which have predefined addresses. The format for these instructions is

```
11  XXX  111  =  000 for RST0
              =  001 for RST1
              =  010 for RST2
              =  011 for RST3
              =  100 for RST4
              =  101 for RST5
              =  110 for RST6
              =  111 for RST7
```

RSTs are one-byte call instructions used mainly with interrupts. Each RST has a predefined address. However, RST0 and the hardware reset vector have the same address 0000_{16}. Therefore, use of RST0 is not usually recommended. The RSTs cause the 8085 to push the PC onto the stack. The 8085 then loads the PC with a predefined address based on the particular RST being used.

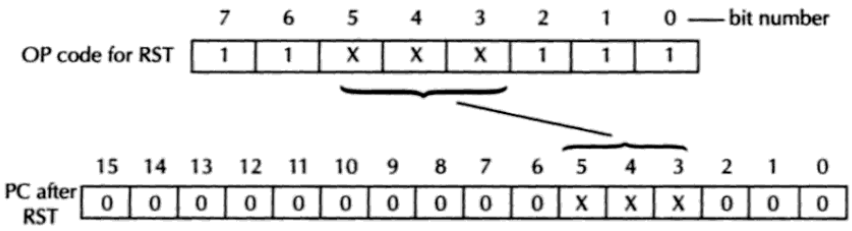


FIGURE 2.5 Execution of the RST instruction.

Table 2.9 RST0-RST7 Vector Addresses

Instruction	OP code (hexadecimal)	Vector address (hexadecimal)
RST0	C7	0000
RST1	CF	0008
RST2	D7	0010
RST3	DF	0018
RST4	E7	0020
RST5	EF	0028
RST6	F7	0030
RST7	FF	0038

A 3-bit code in the OP code for a particular RST determines the address to which the program would branch. This is shown in Figure 2.5.

The vector addresses for the RSTs are listed in Table 2.9. Note that a limited number of locations are available for each RST instruction. This may not provide enough locations for writing a complete subroutine. Therefore, one may place an unconditional JUMP at the vector address to JUMP to an address in RAM where the actual subroutine is written.

The 8085 stack manipulation instructions allow one to save and retrieve the contents of the register pairs into and from the stack, respectively. For example, the following instruction saves the register pair B,C into the stack:

```
PUSH B ; (SP) ← (SP) - 1
       ; M((SP)) ← (B)
       ; (SP) ← (SP) - 1
       ; M((SP)) ← (C)
```

Similarly, the instruction POP D retrieves the top two words of the stack and places them into the registers E and D in that order as follows:

```
POP D ; (E) ← M((SP))
      ; (SP) ← (SP) + 1
      ; (D) ← M((SP))
      ; (SP) ← (SP) + 1
```

This means that all 8085 registers can be saved onto the stack using the following instruction sequence:

```
PUSH PSW ; save the A and flags register
PUSH B   ; save the D,E pair
PUSH D   ; save the D,E pair
PUSH H   ; save the H,L pair
```

Similarly, the saved status can be restored by using the following sequence of POP instructions:

```
POP  H    ; restore H,L pair
POP  D    ; restore D,E pair
POP  B    ; restore B,C pair
POP  PSW  ; restore A and flags register
```

There are two other stack instructions: SPHL and XTHL. SPHL is a one-byte instruction. It moves the [L] to SP high byte and [H] to SP low byte. XTHL is also a one-byte instruction. It exchanges the [L] with the top of the stack addressed by SP and [H] with the next stack addressed by SP + 1. That is, XTHL performs the following:

$$[[SP]] \leftrightarrow [L]$$
$$[[SP+1]] \leftrightarrow [H]$$

The 8085 can use either standard or memory-mapped I/O. Using standard I/O, the input and output instructions have the following format:

```
IN   (8-bit port address)   ;      input instruction
OUT  (8-bit port address)   ;      output instruction
```

For example, the instruction IN 02H transfers the contents of the input port with address 02_{16} into the accumulator. Similarly, the instruction OUT 00H transfers the contents of the accumulator to the output port with address 00_{16}. Using memory-mapped I/O, LDA addr and STA addr can be used as input and output instructions, respectively.

The 8085 HLT instruction forces the 8085 to enter into the halt state. Similarly, the dummy instruction NOP neither achieves any result nor affects any CPU registers. This is a useful instruction for producing software delay routines and to insert diagnostic messages.

The 8085 8-bit increment (INR) and decrement (DCR) instructions affect the status flags. However, the 16-bit increment (INX) and decrement (DCX) instructions do not affect the flags. Therefore, while using these instructions in a loop counter value greater than 256_{10}, some other instructions must be used with DCX or INX to affect the flags after their execution. For example, the following instruction sequence will affect the flags for DCX:

```
        LXI  B,16-BIT DATA  ;  Load
Loop      —                 ;  initial
          —                 ;  16-bit loop
          —                 ;  count to BC
          —
          —
          —                 ;  Decrement
          DCX  B            ;  counter
          MOV  A,B          ;  Move B to A to
                            ;  test for zero
          ORA  C            ;  Logically or with A
          JNZ  Loop         ;  Jump if not zero
            —
            —
            —
```

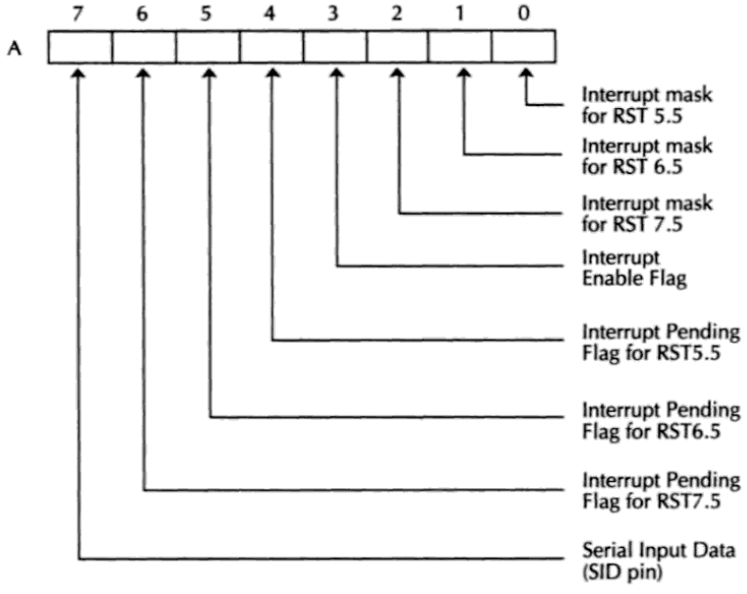


FIGURE 2.6 Accumulator data format after execution of RIM.

There are four one-byte 8085 interrupt instructions. These are DI, EI, RIM, and SIM.

DI disables the 8085's maskable interrupt capability. EI, on the other hand, enables the 8085 maskable interrupt capability.

RIM is a one-byte instruction. It loads the accumulator with 8 bits of data as shown in Figure 2.6.

Bits 0, 1, and 2 provide the values of the RST 5.5, RST 6.5, and RST 7.5 mask bits, respectively. If the mask bit corresponding to a particular RST is one, the RST is disabled; a zero in a specific RST (bits 0, 1, and 2) means that RST is enabled.

If the interrupt enable bit (bit 3) is 0, the 8085's maskable interrupt capability is disabled; the interrupt is enabled if this bit is one.

A "one" in a particular interrupt pending bit indicates that an interrupt is being requested on the identified RST line; if this bit is zero, no interrupt is waiting to be serviced. The serial input data (bit 7) indicate the value of the SID pin.

The SIM instruction outputs the contents of the accumulator to define interrupt mask bits and the serial output data line. The bits in the accumulator before execution of the SIM are defined as shown in Figure 2.7.

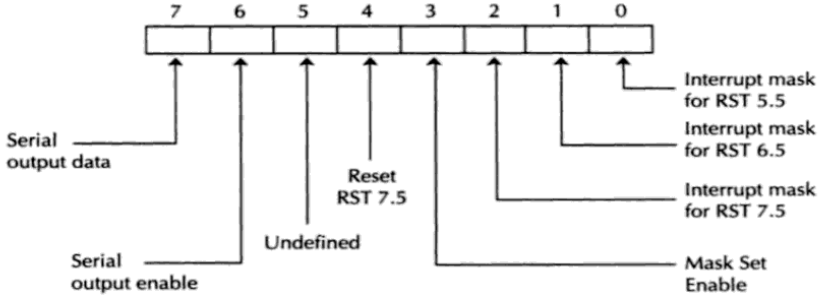


FIGURE 2.7 Accumulator data format before execution of the SIM.

If the mask set enable bit is set to one, interrupt mask bits for RST 7.5, RST 6.5, and RST 5.5 are sent out; a zero value at the mask set enable does not affect the interrupt mask bits. A one at a particular interrupt mask disables that interrupt and a zero enables it.

The RESET RST 7.5, if set to one, resets as internal flip-flop to zero in order to clear the 7.5 interrupt.

If the serial output enable is one, the serial output data are sent to the SOD pin.

The interrupt instructions will be covered in detail during discussion of the 8085 interrupts.

Example 2.1

Write an 8085 assembly language program to add a 16-bit number in locations 5000H (high byte) and 5001H (low byte) with another 16-bit number stored in 5002H (high byte) and 5003H (low byte). Store result in BC.

Solution

```
      ORG   2000H
      LDA   MEM2       ;  Load low byte of
      LXI   H,5003H    ;  number 1
      ADD   M          ;  Add with low
                       ;  byte of number 2
      MOV   C, A       ;  Store result in C
      LDA   MEM1       ;  Load high byte
      DCX   H          ;  of number 1
      ADC   M          ;  Add with
                       ;  high byte and Cy
      MOV   B, A       ;  Store result
                       ;  in B
      HLT              ;  stop
      ORG   5000H
MEM1  DB    DATA1
MEM2  DB    DATA2
MEM3  DB    DATA3
MEM4  DB    DATA4
```

Example 2.2

Write an 8085 assembly language program to perform a parity check on an 8-bit word in location 5000_{16}. If the parity is odd, store DD_{16} in location 5000_{16}. However, if the parity is even, store EE_{16} in location 5000_{16}.

Solution

```
      ORG 3000H
      LDA PARITY  ;  Load 8-bit
                  ;  data into A
      ADI 00H     ;  Add with
                  ;  00H to affect
      MVI A, 0DDH ;  flags
      JPO ODD     ;  Check for
                  ;  odd parity
      MVI A,0EEH  ;  If parity
                  ;  even, store
                  ;  EEH
```

```
ODD     STA PARITY  ;  in 5000H
        HLT         ;  Stop
        ORG 5000H
PARITY  DB DATA
```

Note that address 3000H at ORG 3000H is arbitrarily chosen.

Example 2.3

Write a program in 8085 assembly language to perform an unsigned 8-bit by 8-bit multiplication via repeated addition. Assume that the multiplicand is in "B" register and the multiplier is in "C" register. Store the product in HL.

Solution

```
        ORG 5000H
        LXI H,0000H  ;  Initialize the
                     ;  16-bit product to zero
        MVI C, MULT  ;  Move multiplier to C
        MOV E,B      ;  Move multiplicand
                     ;  to register E
        MVI D,OOH    ;  Convert multiplicand
                     ;  to unsigned 16-bit
START   DAD D        ;  Perform 16-bit addition
        DCR C        ;  Decrement multiplier
        JNZ START    ;  Jump if multiplier not zero
        HLT          ;  Stop. Product
                     ;  is in HL
```

Example 2.4

Write a subroutine in 8085 assembly language to check whether an 8-bit number in location 3000_{16} is odd or even. If the number is odd, store DD_{16} in location 3000_{16}; if the number is even, store EE_{16} in location 3000_{16}.

Also, write the main program to initialize stack pointer 5020_{16}, load the 8-bit number to be checked for odd or even into A, store DD or EE depending on the result, and stop.

Solution

Main Program

```
        ORG 4000H
        LXI SP, 5020H ;  Initialize SP
        LDA START     ;  Load number
        CALL CHECK    ;  Call subroutine
        STA START     ;  Store DD or EE
        HLT           ;  stop
        ORG 3000H
START   DB DATA
```

Subroutine

```
        ORG 7000H
CHECK   RAR           ;  Rotate 'A' for checking
        MVI A,0EEH    ;  Jump if even
```

```
        JNC RETURN     ; Store DD16 if
        MVI A,0DDH     ; odd and return
                       ; Store EE16
RETURN  RET            ; if even and return
```

Example 2.5

Write an 8085 assembly language program to clear 100_{10} consecutive bytes starting at 2050_{16}.

Solution

```
        ORG 2000H
        MVI A, 64H  ; Load 'A' with number of
                    ; bytes to be cleared
        LXI H,2050H ; Load HL with 2050H
LOOP    MVI M, 00H  ; Clear memory location
        INX H       ; Increment address
        DCR A       ; Decrement loop counter
        JNZ LOOP    ; Jump to loop if
                    ; loop counter zero
        HLT         ; Stop
```

2.6 Timing Methods

Timing concepts are very important in microprocessor applications. Typically, in sequential process control the microprocessor is required to provide time delays for on-off devices such as pumps or motor-operated valves. DELAY routines are used to provide such time delays.

A delay program typically has an input register that contains the initial count. The register pair D,E is used for this purpose. A typical delay subroutine is given below:

```
DELAY  DCX D      ; Decrement the D,E contents
       MOV A,D    ;
       ORA E      ; Are the contents zero?
       JNZ DELAY  ; Jump if not zero
       RET        ; Return
```

We now calculate the total time required by the DELAY routine using the following data:

Instruction	Number of cycles
CALL	18
DCX D	6
MOV A,D	4
ORA E	4
JNZ	7/10
RET	10

Note that in the above, if the JNZ condition is met (Z = 0), ten cycles are required and the program branches back to the DCX D instruction. However, if the JNZ condition is not met (Z = 1), the seven cycles are required, and the program executes the next instruction, that is, the RET instruction. Also, note that the CALL instruction is used in the main program written by the user, and the 3-byte instruction CALL DELAY is used.

For each iteration in which the JNZ condition is met (Z = 0), the number of cycles is equal to cycles for DCX D + cycles for MOV A,D + cycles for ORA E + cycles for JNZ = 6 + 4 + 4 + 10 = 24 cycles.

These 24 cycles will be performed (y – 1) times, where y is the initial contents of D,E. For the final iteration in which no jump is performed and the JNZ condition is not satisfied ($Z \neq 0$), the number of cycles is equal to cycles for DCX D + cycles for MOV A,D + cycles for ORA E + cycles for JNZ + cycles for RET = 6 + 4 + 4 + 7 + 10 = 31 cycles. Therefore, the time used, including a CALL instruction, is

$$18 + 31 + 24(y - 1) = 49 + 24(y - 1) \text{ clock cycles}$$

Suppose that in a program a delay time of 1/3 ms is desired. The DELAY routine can be used to accomplish this in the following way. Each cycle of the 8085 clock is 1/3 ms (3 MHz). The number of cycles required in the DELAY routine is

$$\frac{1/3 \text{ ms}}{1/3 \mu\text{s}} = \frac{10^{-3}}{10^{-6}} = 1000 \text{ cycles}$$

Therefore, the initial counter value y of the D,E register pair can be calculated:

$$\begin{aligned} 49 + 24\,(y - 1) &= 1000 \\ 24\,(y - 1) &= 951 \\ y = \frac{951}{24} + 1 &\cong 40_{10} = 28_{16} \end{aligned}$$

Therefore, in the program the D,E register pair can be loaded with 0028_{16} and the DELAY routine can be called to obtain 1/3 ms of time delay. Table 2.10 shows initial counts for various time delays.

Table 2.10 Time Intervals along with Initial Counts

3-MHz clock milliseconds	Initial count (hexadecimal)
1/3	0028
1	007C
2	00F9
10	04E1
100	30D3

The following program produces a delay of 10 ms:

```
LXI SP, 5000H  ;  Set stack pointer
LXI D, 04E1H   ;  Load D,E with initial count value of 04E1
               ;  to provide 10 ms of delay
CALL DELAY     ;  Call DELAY routine
HLT            ;  STOP
```

The delay times can be increased by using a counter. Suppose that a delay of 5 s is desired in a program. From Table 2.10 an initial count of $30D3_{16}$ produces a 100-ms delay. We can use a counter along with the 100-ms delay to obtain the 5-s delay as follows

$$(100 \text{ ms}) \times X = 5 \text{ s}$$

where X is the value of the counter. Then

$$X = \frac{5}{100 \times 10^{-3}} = \frac{5}{10^{-1}} = 50$$

Therefore, a counter of 50_{10} or 32_{16} is required. Now the program for the 5-s delay can be written as follows:

```
        LXI  SP,  5000H  ;  Set stack pointer
        MVI  C,   32H    ;  Do DELAY loop 50(10) times by loading C
                         ;  with count 32(16)
START   LXI  D,   30D3H  ;  Load initial count
        CALL DELAY       ;  Call DELAY loop
        DCR  C           ;  Decrement C and check if zero: if
                         ;  not, do another delay
        JNZ  START       ;  Loop back
        HLT              ;  STOP
```

In the above, since execution times of DCR C and JNZ START are very small compared to 5 s, they are not considered in computing the delay.

2.7 8085 Pins and Signals

The 8085 is housed in a 40-pin dual in-line package (DIP). Figure 2.8 shows the 8085 pins and signals.

The low-order address byte and data lines AD0 to AD7 are multiplexed. These lines are bidirectional. The beginning of an instruction is indicated by the rising edge of the ALE signal. At the falling edge of ALE, the low byte of the address is automatically latched by some of the 8085 support chips such as 8155 and 8355: AD0 to AD7 lines can then be used as data lines. Note that ALE is an input to these support chips. However, if the support chips do not latch AD0 to AD7, then external latches are required to generate eight separate address lines A7 to A0 at the falling edge of ALE.

Pins A8 to A15 are unidirectional and contain the high byte of the address.

Table 2.11 lists the 8085 pins along with a brief description of each.

The $\overline{RD}$ pin signal is output LOW by the 8085 during a memory or I/O READ operation. Similarly, the $\overline{WR}$ pin signal is output LOW during a memory or I/O WRITE.

Next, we explain the purpose of IO/$\overline{M}$, S0, and S1 signals. The IO/$\overline{M}$ signal is output HIGH by the 8085 to indicate execution of an I/O instruction such as IN or OUT. This pin is output LOW during execution of a memory instruction such as LDA 2050H.

The IO/$\overline{M}$, S0, and S1 are output by the 8085 during its internal operations, which can be interpreted as follows:

IO/$\overline{M}$	S1	S0	Operation performed by the 8085
0	0	1	Memory WRITE
0	1	0	Memory READ
1	0	1	I/O WRITE
1	1	0	I/O READ
0	1	1	OP code fetch
1	1	1	Interrupt acknowledge

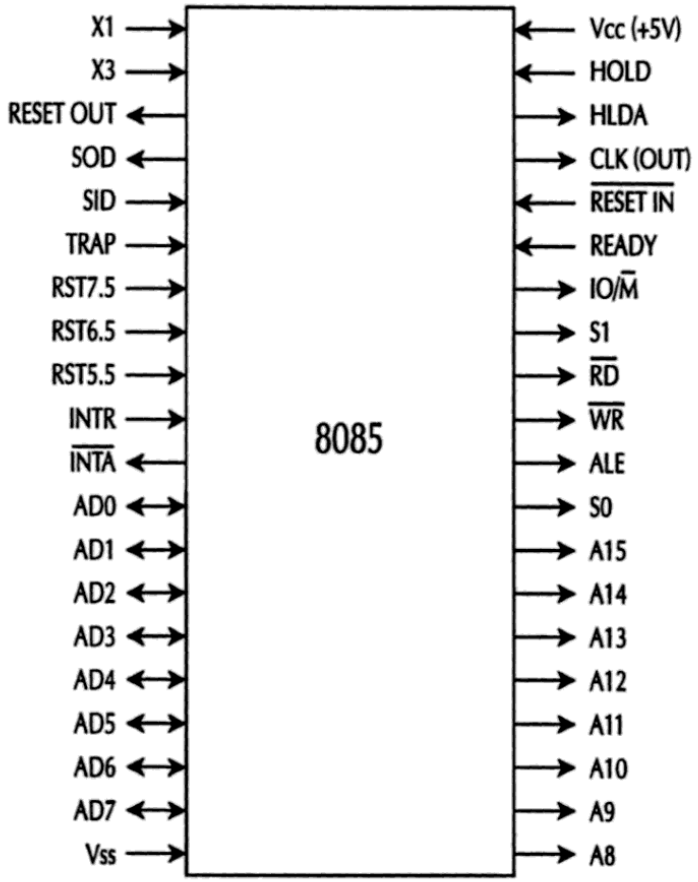
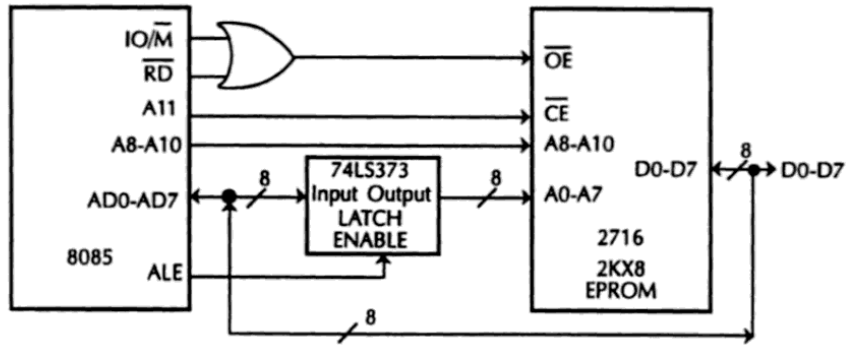


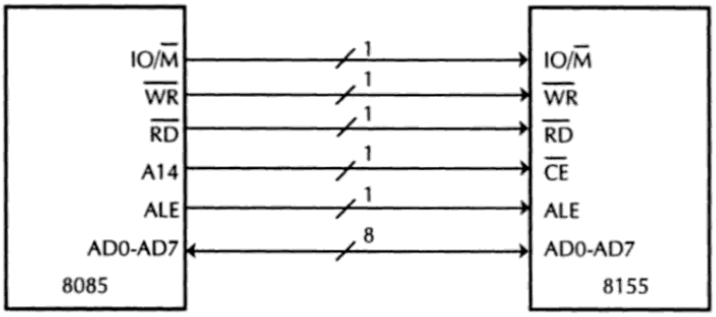
FIGURE 2.8 8085 microprocessor signals and pin assignments.

Table 2.11 8085 Signal Description Summary

Pin name	Description	Type
AD0-AD7	Address/data bus	Bidirectional, tristate
A8-A15	Address bus	Output, tristate
ALE	Address latch enable	Output, tristate
RD	Read control	Output, tristate
WR	Write control	Output, tristate
IO / M	I/O or memory indicator	Output, tristate
S0, S1	Bus state indicators	Output
READY	Wait state request	Input
SID	Serial data input	Input
SOD	Serial data output	Output
HOLD	Hold request	Input
HLDA	Hold acknowledge	Output
INTR	Interrupt request	Input
TRAP	Nonmaskable interrupt request	Input
RST5.5	Hardware vectored	Input
RST6.5	Hardware vectored interrupt request	Input
RST7.5	Hardware vectored	Input
INTA	Interrupt acknowledge	Output
RESET IN	System reset	Input
RESET OUT	Peripherals reset	Output
X1, X2	Crystal or RC connection	Input
CLK (OUT)	Clock signal	Output
Vcc, Vss	Power, ground	



(a) 8155 - 2716 interface using internal latches.



(b) 8085 - 8155 interface using ALE and AD0-AD7

FIGURE 2.9 8085's interface to external device using ALE and the multiplexed AD0 to AD7 pins.

Figure 2.9 illustrates the utilization of ALE and AD0 to AD7 signals for interfacing an EPROM and a RAM.

The 2716 is a 2K × 8 EPROM with separate address and data lines without any built-in latches. This means that a separate latch such as the 74LS373 must be used to isolate the 8085 low byte address and D0-D7 data lines at the falling edge of ALE (Figure 2.9a).

The 8155 contains 256-byte static RAM, three user ports, and a 14-bit timer. The 8155 is designed for 8085 in the sense that it has built-in latches with ALE as input along with multiplexed address (low byte) and data lines, AD0 to AD7. Therefore, as shown in Figure 2.9b, external latches are not required.

The READY input can be used by the slower external devices for obtaining extra time in order to communicate with the 8085. The READY signal (when LOW) can be utilized to provide wait-state clock periods in the 8085 machine. If READY is HIGH during a read or write cycle, it indicates that the memory or peripheral is ready to send or receive data. If not used, it must be tied high.

The Serial Input Data (SID) and Serial Output Data (SOD) lines are associated with the 8085 serial I/O transfer. The SOD line can be used to output the most significant bit of the accumulator. The SID signal can be input into the most significant bit of the accumulator.

The HOLD and HLDA signals are used for the Direct Memory Access (DMA) type of data transfer. The external devices place a HIGH on HOLD line in order to take control of the system bus. The HOLD function is acknowledged by the 8085 by placing a HIGH output on the HLDA pin and driving the tristate outputs high impedance.

The signals on the TRAP, RST7.5, RST6.5, RST5.5, INTR, and $\overline{\text{INTA}}$ are related to the 8085 interrupt signals. TRAP is a nonmaskable interrupt; that is, it cannot be enabled or disabled

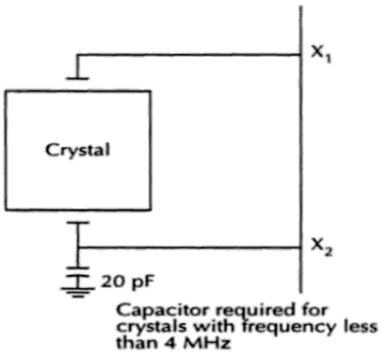


FIGURE 2.10 Crystal connection to X1 and X2 pins.

by an instruction. The TRAP has the highest priority. RST7.5, RST6.5, and RST5.5 are maskable interrupts used by the devices whose vector addresses are generated automatically. $\overline{\text{INTA}}$ is an interrupt acknowledge signal which is pulsed LOW by the 8085 in response to the interrupt INTR request. In order to service INTR, one of the eight OP codes (RST0 to RST7) has to be provided on the 8085 AD0-AD7 bus by external logic. The 8085 then executes this instruction and vectors to the appropriate address to service the interrupt.

All unused control pins such as interrupts and HOLD must be disabled by grounding them. (READY must be tied high).

The 8085 has the clock generation circuit on the chip and, therefore, no external oscillators need to be designed. The 8085A can operate with a maximum clock frequency of 3.03 MHz and the 8085A-2 can be driven with a maximum of 5 MHz clock. The 8085 clock frequency can be generated by a crystal, an LC tuned circuit, or an external clock circuit. The frequency at X1X2 is divided by 2 internally. This means that in order to obtain 3.03 MHz, a clock source of 6.06 MHz must be connected to X1X2. For crystals of less than 4 MHz, a capacitor of 20 pF should be connected between X2 and a ground to ensure the starting up of the crystal at the right frequency (Figure 2.10).

There is a TTL signal which is output on pin 37, called the CLK (OUT) signal. This signal can be used by other external microprocessors or support chips.

The $\overline{\text{RESET IN}}$ signal, when pulsed LOW then high, causes the 8085 to execute the first instruction at the 0000_{16} location. In addition, the 8085 resets instruction register, interrupt mask (RST5.5, RST6.5, and RST7.5) bits, and other registers. The $\overline{\text{RESET IN}}$ must be held LOW for at least three clock periods. A typical 8085 reset circuit is shown in Figure 2.11. In this circuit, when the switch is activated, $\overline{\text{RESET IN}}$ is driven to LOW with a large time constant providing adequate time to reset the system.

The 8085 requires a minimum operating voltage of 4.75 V. Upon applying power, the 8085A attains this voltage after 500 μs. The reset circuit of Figure 2.11 resets the 8085 upon activation

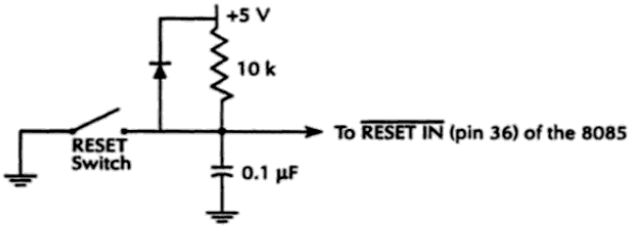


FIGURE 2.11 8085 reset circuit.

FIGURE 2.12 8085 machine cycles.

of the switch. The voltage across the 0.1-μF capacitor is zero on power-up. The capacitor then charges Vcc after a definite time determined by the time constant RC. The chosen values of RC in the figure will drive the $\overline{\text{RESET IN}}$ pin to low for at least three clock periods. In this case, after activating the switch, $\overline{\text{RESET IN}}$ will be low (assuming capacitance charge time is equal to the discharge time) for 10K*0.1 μF = 1 ms, which is greater than three clock periods (3*1/3 μs = 1 μs) of the 3-MHz 8085A. During normal operation of the 8085, activation of the switch will short the capacitor to ground and will discharge it. When the switch is opened, the capacitor charges and the $\overline{\text{RESET IN}}$ pin becomes HIGH. Upon hardware reset, the 8085 clears PC, IR, HALT flip-flop, and some other registers; the 8085 registers PSW, A, B, C, D, E, H, and L are unaffected. Upon activation of the $\overline{\text{RESET IN}}$ to low, the 8085 outputs HIGH at the RESET OUT pin which can be used to reset the memory and I/O chips connected to the 8085. Note that since hardware reset initializes PC to 0, the 8085 fetches the first instruction for address 0000_{16} after reset.

2.8 8085 Instruction Timing and Execution

An 8085's instruction execution consists of a number of machine cycles (MCs). These cycles vary from one to five (M1 to M5) depending on the instruction. Each machine cycle contains a number of 320-ns clock periods. The first machine cycle will be executed in either four or six clock periods, and the machine cycles that follow will have three clock periods. This is shown in Figure 2.12.

The shaded MCs indicate that these machine cycles are required by certain instructions. Similarly, the shaded clock periods (T5 and T6) mean that they are needed in M1 by some instructions.

The clock periods within a machine cycle can be illustrated as shown in Figure 2.13. Note that the beginning of a new machine cycle is indicated on the 8085 by outputting the Address Latch Enable (ALE) signal HIGH. During this time, lines AD0 to AD7 are used for placing the low byte of the address.

When the ALE signal goes LOW, the low byte of the address is latched so that the AD0 to AD7 lines can be used for transferring data.

We now discuss the timing diagrams for instruction fetch, READ, and WRITE.

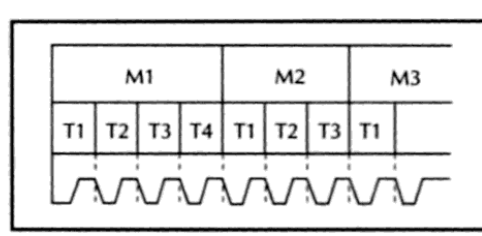


FIGURE 2.13 Clock period within a machine cycle.

2.8.1 Basic System Timing

Figure 2.14 shows the 8085 basic system timing. An instruction execution includes two operations: OP code fetch and execution.

The OP code fetch cycle requires either four (for one-byte instructions such as MOV A,B) or six cycles (for 3 byte instructions such as LDA 2030H). The machine cycles that follow will need three clock periods.

The purpose of an instruction fetch is to read the contents of a memory location containing an instruction addressed by the program counter and to place it in the instruction register. The 8085 instruction fetch timing diagram shown in Figure 2.14 can be explained in the following way:

1. The 8085 puts a LOW on the $IO/\overline{M}$ line of the system bus, indicating a memory operation.
2. The 8085 sets S0 = 1 and S1 = 1 on the system bus, indicating the memory fetch operation.
3. The 8085 places the program counter high byte on the A8 to A15 lines and the program counter low byte on the AD0 to AD7 lines of the system bus. The 8085 also sets the ALE signal to HIGH. As soon as the ALE signal goes to LOW, the program counter low byte on the AD0 to AD7 is latched automatically by some 8085 support chips such as 8155 (if 8085 support chips are not used, these lines must be latched using external latches), since these lines will be used as data lines for reading the OP code.
4. At the beginning of T2 in M1, the 8085 puts the $\overline{RD}$ line to LOW indicating a READ operation. After some time, the 8085 loads the OP code (the contents of the memory location addressed by the program counter) into the instruction register.
5. During the T4 clock period in M1, the 8085 decodes the instruction.

The Machine Cycle M2 of Figure 2.14 shows a memory (or I/O) READ operation as seen by the external logic, and the status of the S0 and S1 signals indicates whether the operation is instruction fetch or memory READ; for example, S1 = 1, S0 = 1 during instruction fetch and S1 = 1, S0 = 0 during memory READ provided $IO/\overline{M} = 0$.

The purpose of the memory READ is to read the contents of a memory location addressed by a register pair, such as the H,L pair, or a memory location specified with the instruction and the data placed in a microprocessor register such as the accumulator. In contrast, the purpose of the memory fetch is to read the contents of a memory location addressed by PC into IR. The machine cycle M3 of Figure 2.14 indicates a memory (or I/O) write operation. In this case, S1 = 0 and S0 = 1 indicate a memory write operation when $IO/\overline{M} = 0$ and an I/O write operation when $IO/\overline{M} = 1$.

2.8.2 8085 Memory READ ($IO/\overline{M}$ = 0, $\overline{RD}$ = 0) and I/O READ ($IO/\overline{M}$ = 1, $\overline{RD}$ = 0)

Figure 2.15a shows an 8085A clock timing diagram. The machine cycle of M2 of Figure 2.14 shows a memory READ timing diagram.

The purpose of the memory READ is to read the contents of a memory location addressed by a register pair, such as HL. Let us explain the 8085 memory READ timing diagram of Figure 2.15b along with the READ timing signals of Figure 2.14:

1. The 8085 uses machine cycle M1 to fetch and decode the instruction. It then performs the memory READ operation in M2.
2. The 8085 continues to maintain $IO/\overline{M}$ at LOW in M2 indicating a memory READ operation (or $IO/\overline{M} = 1$ for I/O READ).
3. The 8085 puts S1 = 1, S0 = 0, indicating a READ operation.
4. The 8085 places the contents of the high byte of the memory address register, such as the contents of the H register, on lines A8 to A15.

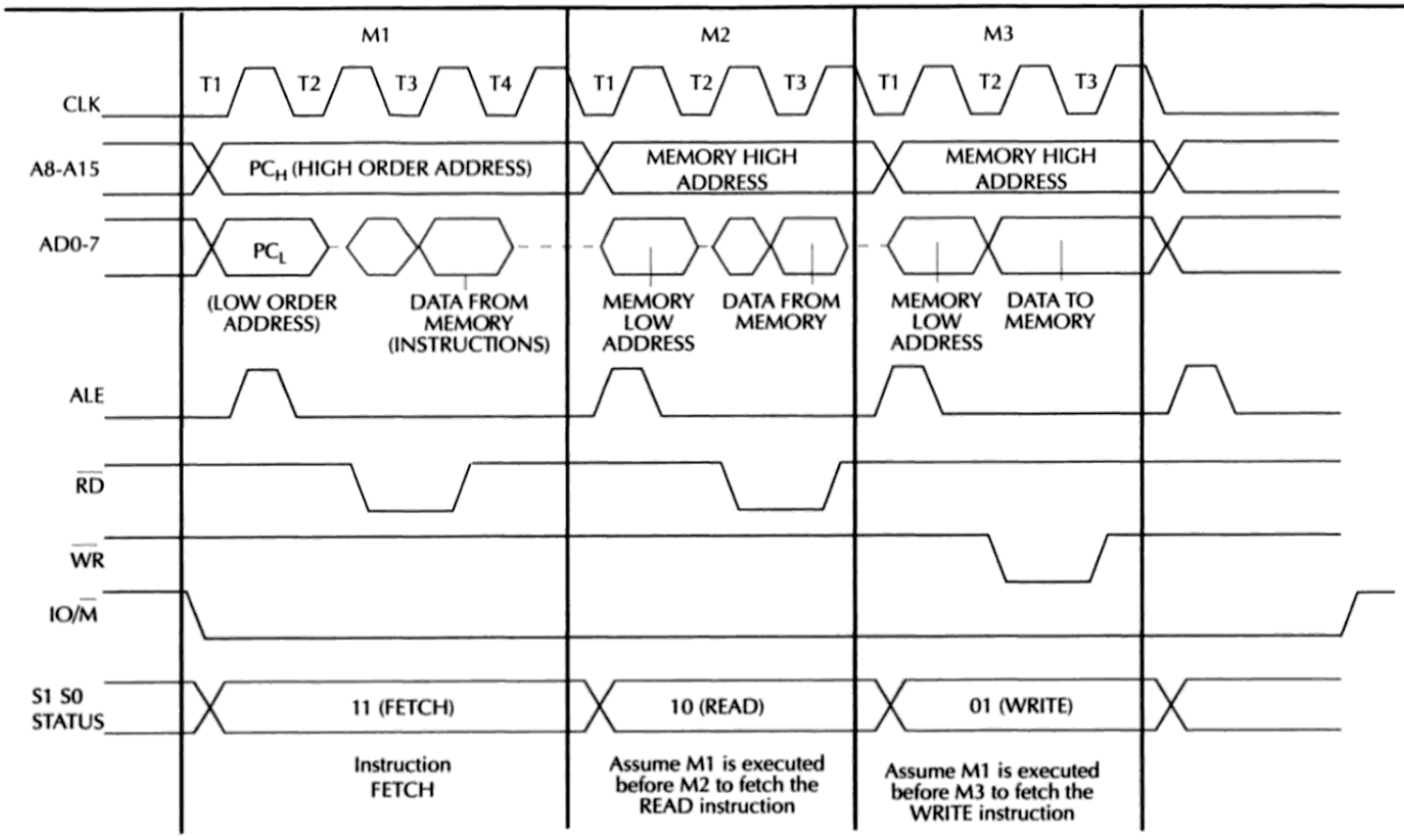


FIGURE 2.14 8085 Basic system timing.

5. The 8085 places the contents of the low byte of the memory address register, such as the contents of the L register, in lines AD0 to AD7.
6. The 8085 sets ALE to high, indicating the beginning of M2. As soon as ALE goes to low, the memory or support chip must latch the low byte of the address lines, since the same lines are going to be used as data lines.
7. The 8085 puts the $\overline{RD}$ signal to LOW, indicating a READ operation.
8. The 8085 gets the data from the memory location addressed by the memory address register, such as the H,L pair, and places the data into a register such as the accumulator. In case of I/O, the 8085 inputs data from the I/O port into the accumulator.

2.8.3 8085 Memory WRITE ($IO/\overline{M}$ = 0, $\overline{WR}$ = 0) and I/O WRITE ($IO/\overline{M}$ = 1, $\overline{WR}$ = 0)

The machine cycle M3 of Figure 2.14 shows a memory WRITE timing diagram. As seen by the external logic, the signals S1 = 0, S0 = 1, and $\overline{WR}$ = 0 indicate a memory WRITE operation.

The purpose of a memory WRITE is to store the contents of the 8085 register, such as the accumulator, into a memory location addressed by a pair, such as H,L.

The WRITE timing diagram of Figure 2.14 can be explained as follows:

1. The 8085 uses machine cycle M1 to fetch and decode the instruction. It then executes the memory WRITE instruction in M3.
2. The 8085 continues to maintain $IO/\overline{M}$ at LOW, indicating a memory operation (or $IO/\overline{M}$ = 1 for I/O WRITE).
3. The 8085 puts S1 = 0, S0 = 1, indicating a WRITE operation.
4. The 8085 places the Memory Address Register high byte, such as the contents of the H register, on lines A8 to A15.
5. The 8085 places the Memory Address Register low byte, such as the contents of L register, on lines AD0 to AD7.
6. The 8085 sets ALE to HIGH, indicating the beginning of M3. As soon as ALE goes to LOW, the memory or support chip must latch the low byte of the address lines, since the same lines are going to be used as data lines.
7. The 8085 puts the $\overline{WR}$ signal to LOW, indicating a WRITE operation.
8. It also places the contents of the register, say, accumulator, on data lines AD0 to AD7.
9. The external logic gets data from the lines AD0 to AD7 and stores the data in the memory location addressed by the Memory Address Register, such as the H,L pair. In case of I/O, the 8085 outputs [A] to an I/O port.

Figures 2.15a through c show the 8085A clock and read and write timing diagrams.

2.9 8085 Input/Output (I/O)

The 8085 I/O transfer techniques are discussed. The 8355/8755 and 8155/8156 I/O ports and 8085 SID and SOD lines are also included.

2.9.1 8085 Programmed I/O

There are two I/O instructions in the 8085, namely, IN and OUT. These instructions are 2 bytes long. The first byte defines the OP code of the instruction and the second byte specifies the I/O port number. Execution of the IN PORT instruction causes the 8085 to receive one byte of data into the accumulator from a specified I/O port. On the other hand, the OUT PORT instruction, when executed, causes the 8085 to send one byte of data from the accumulator into a specified I/O port.

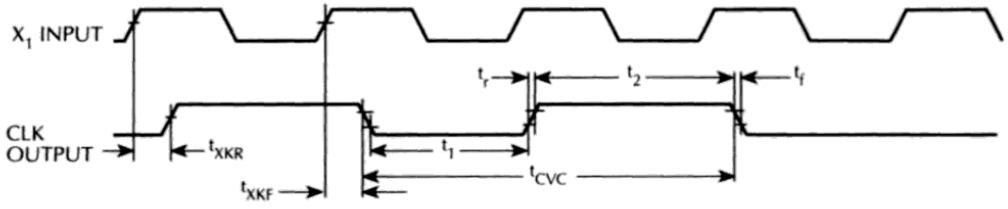


FIGURE 2.15a 8085A clock.

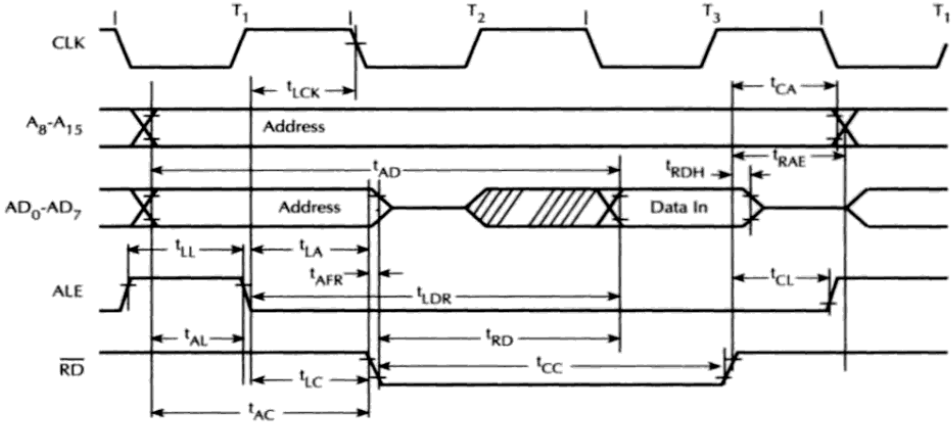


FIGURE 2.15b 8085 Read Timing diagram.

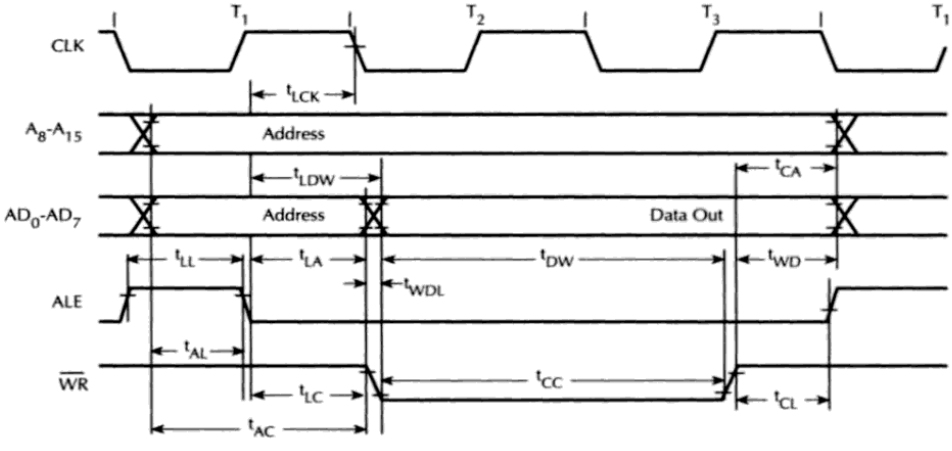


FIGURE 2.15c 8085 Write Timing diagram.

The 8085 can access I/O ports using either standard I/O or memory-mapped I/O.

In standard I/O, the 8085 inputs or outputs data using IN or OUT instructions.

In memory-mapped I/O, the 8085 maps I/O ports as memory addresses. Hence, LDA addr or STA addr instructions are used to input or output data to or from the 8085. The 8085's programmed I/O capabilities are obtained via the support chips, namely, 8355/8755 and 8155/8156. The 8355/8755 contains a 2K-byte ROM/EPROM and two 8-bit I/O ports (ports A and B).

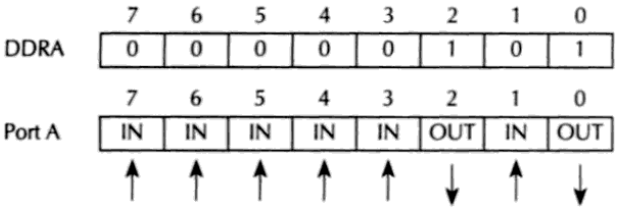
The 8155/8156 contains 256-byte RAM, two 8-bit and one 6-bit I/O ports, and a 14-bit programmable timer. The only difference between the 8155 and 8156 is that chip enable is LOW on the 8155 and HIGH on the 8156.

2.9.1.a 8355/8755 I/O Ports

Two 8-bit ports are included in the 8355/8755. These are ports A and B. Another 8-bit port, called the data direction register, is associated with each one of these ports. These registers (DDRA and DDRB) can be used to configure each bit in ports A or B as either input or output. For example, a "0" written into a bit position of the data direction register sets up the corresponding bit in the I/O port as input. On the other hand, a "1" written in a particular bit position in the data direction register sets up the corresponding bit in the I/O port as output. For example, consider the following instruction sequence:

```
MVI  A, 05H
OUT  DDRA
```

The above instruction sequence assumes DDRA as the data direction register for port A. The bits of port A are configured as follows:



The 8355/8755 uses the IO/$\overline{M}$ pin on the chip in order to distinguish between standard and memory-mapped I/O. This pin is controlled by the 8085 as shown in Figure 2.16.

The 8085 outputs a HIGH on the IO/$\overline{M}$ pin when it executes either an IN or OUT instruction. This means that IO/$\overline{M}$ in the 8355/8755 becomes HIGH during execution of IN or OUT. This, in turn, tells the 8355/8755 to decode the AD1 and AD0 lines in order to obtain the 8-bit address of various ports in the chip as follows:

AD1	AD0	
0	0	Port A
0	1	Port B
1	0	Data Direction Register A
1	1	Data Direction Register B

The other 6 bits of each 8-bit port address are don't care conditions. This means that these bits can be either one or zero. The 8085/8355/8755 standard I/O is illustrated in Figure 2.17.

Since the 8355/8755 is provided with 2K bytes of memory, 11 address lines A0 to A10 are required for memory addressing. Since the 8085 has 16 address pins A0 to A15, A11 to A15 will not be used for memory addressing. Note that the 8355/8755 includes two chip enables $\overline{CE}$ and CE. In Figure 2.17 these two chip enables are connected to Vcc and A11, respectively. It should be pointed out that the 8085 duplicates low and high bytes of the 16-bit address lines with the port address when it executes an IN or OUT instruction. This means that if the 8085 executes IN 01 instruction, it puts 0101_{16} on the 16 address lines. Note that in Figure 2.17, A11 = 0 for both memory and port addressing. This is because A11 = 0 enables this chip. When the 8085 executes an LDA addr or STA addr instruction, IO/$\overline{M}$ becomes LOW. This tells the 8355/8755 to interpret A0 to A10 as memory addresses. On the other hand, when the 8085 executes an IN PORT or OUT PORT instruction, the 8085 drives IO/$\overline{M}$ to HIGH. This tells the 8355/8755 to decode AD1 and AD0 for I/O port addresses. The port addresses are as follows:

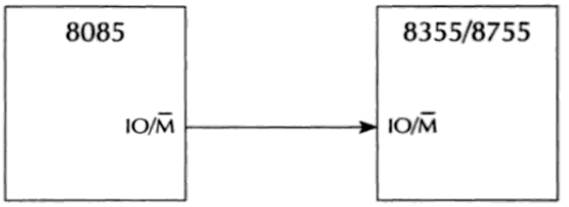


FIGURE 2.16 Interfacing the 8085 with 8355/8755 via the IO/$\overline{M}$ pin.

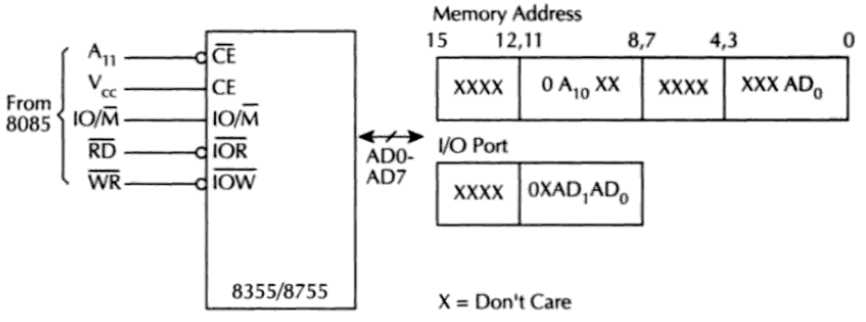


FIGURE 2.17 8355/8755 standard I/O.

	A15 AD7	A14 AD6	A13 AD5	A12 AD4	A11 AD3	A10 AD2	A9 AD1	A8 AD0	Address
Port A =	X	X	X	X	X	0	X	0	$= 00_{16}$
Port B =	X	X	X	X	0	X	0	1	$= 01_{16}$
DDRA =	X	X	X	X	0	X	1	0	$= 02_{16}$
DDRB =	X	X	X	X	0	X	1	1	$= 03_{16}$

X is don't care. Assume X is zero in the above.

Let us now discuss 8355/8755 memory-mapped I/O. Figure 2.18 provides such an example. In Figure 2.18, A11 must be zero for selecting the 8355/8755 and A15 is connected to IO/$\overline{M}$ of the 8355/8755. When A15 = 1, IO/$\overline{M}$ becomes HIGH. This tells the 8355/8755 chip to decode AD1 and AD0 for obtaining I/O port addresses. For example, if we assume all don't cares in the I/O port address are 1, then the I/O port addresses will be mapped into memory locations as follows:

Port name	16-bit memory address
I/O port A	$F7FC_{16}$
I/O port B	$F7FD_{16}$
DDRA	$F7FE_{16}$
DDRB	$F7FF_{16}$

Note that in Figure 2.18 memory addresses are mapped as 7000_{16} through $77FF_{16}$ assuming all don't cares to be ones.

Note that the above port addresses may not physically exist in memory. However, input or output operations with these ports can be accomplished by generating the necessary signals by executing LDA or STA instructions with the above addresses. For example, outputting to DDRA or DDRB can be accomplished via storing to locations $F7FE_{16}$ or $F7FF_{16}$, respectively. The instructions STA F7FEH or STA F7FFH will generate all of the required signals for OUT

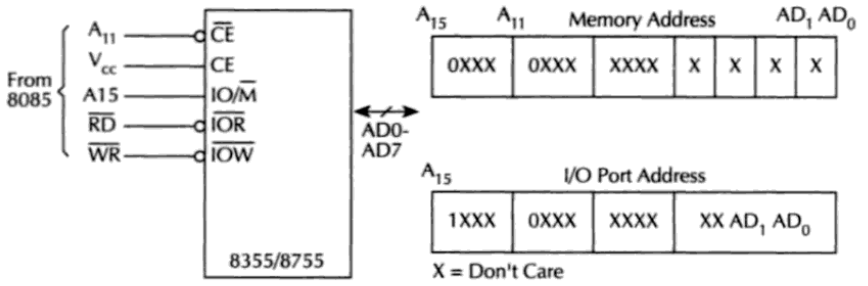


FIGURE 2.18 8355/8755 memory-mapped I/O.

DDRA or OUT DDRB, respectively. For example, upon execution of the STA F7FEH, the 8085 sends a LOW to the $\overline{WR}$ pin and places F7FEH on the address bus. This will make A15 = 1, A11 = 0, AD1 = 1, AD0 = 0, and thus will make the $IO/\overline{M} = 1$, $\overline{CE} = 0$, $\overline{IOW} = 0$ on the 8355/8755 in Figure 2.18.

When ALE goes to LOW, the 8085 places the contents of the accumulator on the AD7 to AD0 pins. The 8355/8755 then takes this data and writes into DDRA. Therefore, STA F7FEH is equivalent to OUT DDRA instruction.

2.9.1.b 8155/8156 I/O Ports

The 8155 or 8156 includes 256 bytes of static RAM and three parallel I/O ports. These ports are port A (8-bit), port B (8-bit), and port C (6-bit). By parallel it is meant that all bits of the port are configured as either all input or all output. Bit-by-bit configuration like the 8355/8755 is not permitted. The only difference between the 8155 and 8156 is that the 8155 has LOW chip enable ($\overline{CE}$), while the 8156 includes a HIGH chip enable (CE). The 8155/8156 ports are configured by another port called the command status register (CSR). When data are output to the CSR via the accumulator, each bit is interpreted as a command bit to set up ports and control timer as shown in Figure 2.19. Port C can be used as a 6-bit parallel port or as a control port to support data transfer between the 8085 and an external device via ports A and B using handshaking. Note that handshaking means data transfer via exchange of control signals. Two bits (bits 2 and 3) are required in CSR to configure port C. Note that port A interrupt and port B interrupt are associated with handshaking and are different from the 8085 interrupts. For example, port A interrupt is HIGH when data are ready to be transferred using handshaking signals such as port A buffer full and port A strobe. The port A interrupt (PC0 in ALT3) can be connected to an 8085 interrupt pin and data can be transferred to or from the 8085 via port A by executing appropriate instructions in the interrupt service routine.

When the 8085 reads the CSR, it accesses the status register and information such as status of handshaking signals and timer interrupt is obtained.

Three bits are used to decode the 8155 six ports (CSR, port A, port B, port C, timer high port, timer low port) as follows:

AD2	AD1	AD0	Port selected
0	0	0	CSR
0	0	1	Port A
0	1	0	Port B
0	1	1	Port C
1	0	0	Timer-low port
1	0	1	Timer-high port

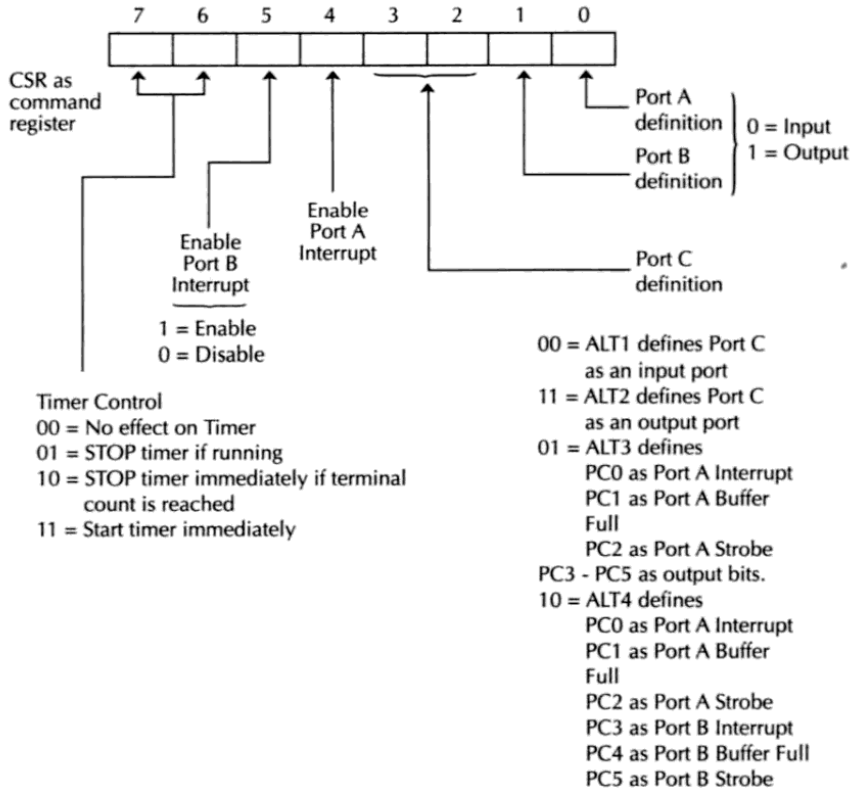
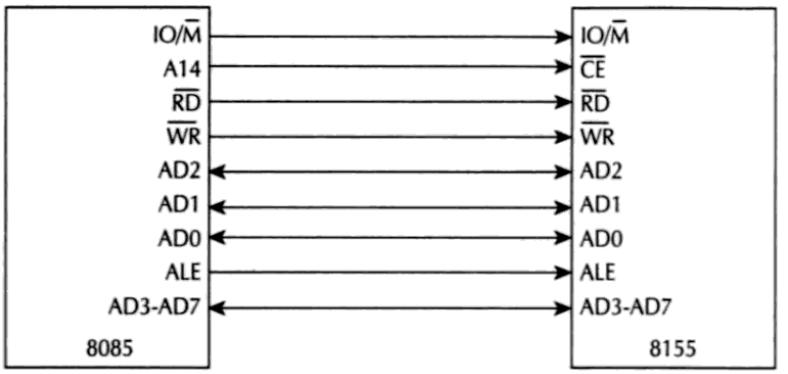


FIGURE 2.19 CSR format as command register.

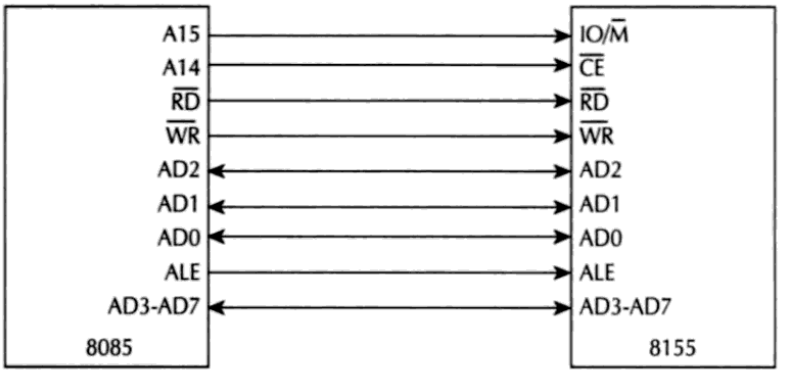
A typical interface between the 8085 and 8155 is shown in Figure 2.20. Consider Figure 2.20a. Since the 8085 duplicates low byte address bus with the high byte address (i.e., AD7 to AD0 same as A15 to A8 for 8085 standard I/O), the address pins AD2 to AD0 will be the same as A10 to A8. This means that the pins A10 to A8 must not be used as chip enables since they will be used for decoding of port addresses. Therefore, A14 is used as chip enable in the figure. The unused address lines A11 to A13 and A15 are don't cares and are assumed to be zero in the following. Therefore, the port addresses are

A15 AD7	A14 AD6	A13 AD5	A12 AD4	A11 AD3	A10 AD2	A9 AD1	A8 AD0	Address
0	0	0	0	0	0	0	0	= 00H = CSR
0	0	0	0	0	0	0	1	= 01H = Port A
0	0	0	0	0	0	1	0	= 02H = Port B
0	0	0	0	0	0	1	1	= 03H = Port C

For memory-mapped I/O, consider Figure 2.20b. In this case, the 8085 low byte and high byte address bus are not duplicated. The ports will have 16-bit addresses as follows. Assume the unused address pins A8 to AD3 to be zeros.



(a) 8085 – 8155 Interface using standard I/O



(b) 8085 – 8155 Interface using memory-mapped I/O

FIGURE 2.20 8085-8155 interface for I/O ports.

A15	A14	A13	A12	A11	A10	A9	A8	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	Port address
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 8000_{16} CSR
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	= 8001_{16} (Port A)
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	= 8002_{16} (Port B)
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	= 8003_{16} (Port C)

Like the 8355/8755 memory-mapped I/O, the above port addresses may not physically exist. However, read or write operations with them will generate the necessary signals for input or output transfer with the ports. Assuming all don't cares to be zeros, the memory map of either configuration of Figure 2.20a or b includes addresses 0000H through 00FFH.

Example 2.6

An 8085-8355-based microcomputer is required to drive an LED connected to bit 0 of port A based on the input conditions set by a switch on bit 1 of port A. The input/output conditions are as follows: if the input to bit 1 of port A is HIGH then the LED will be turned ON; otherwise

the LED will be turned OFF. Assume that a HIGH will turn the LED ON and a LOW will turn it OFF. Write an 8085 assembly language program starting at 5000H.

Solution

```
        ORG  5000H
PORT A  EQU  00H
DDRA    EQU  02H
        MVI  A, 01H   ;  Configure Port A
        OUT  DDRA
START   IN   PORT A   ;  Input Port A
        RAR           ;  Rotate switch to LED position
        OUT  PORT A   ;  Output to LED
        JMP  START    ;  Endless loop
```

Example 2.7

An 8085-8155-based microcomputer is required to drive an LED connected to bit 0 of port A based on two switch inputs connected to bits 6 and 7 of port A. If both switches are either HIGH or LOW, turn the LED on; otherwise turn it OFF. Assume that a HIGH will turn the LED ON and a LOW will turn it OFF. Use port addresses of CSR and port A as 20_{16} and 21_{16}, respectively.

Write an 8085 assembly language program to accomplish this starting at address 3000H.

Solution

```
        ORG  3000H
CSR     EQU  20H
PORT A  EQU  21H
START   MVI  A, 00H   ;  Configure Port A
        OUT  CSR      ;  as input
        IN   PORT A   ;  Input Port A
        ANI  0C0H     ;  Retain bits 6 and 7
                      ;  turn LED ON
        MVI  A, 01H   ;  Otherwise, configure
        OUT  CSR      ;  Port A as output
        JPE  LEDON
        MVI  A, 00H   ;  and turn LED OFF.
LEDON   OUT  PORT A   ;
        JMP  START    ;  Jump to START
```

Example 2.8

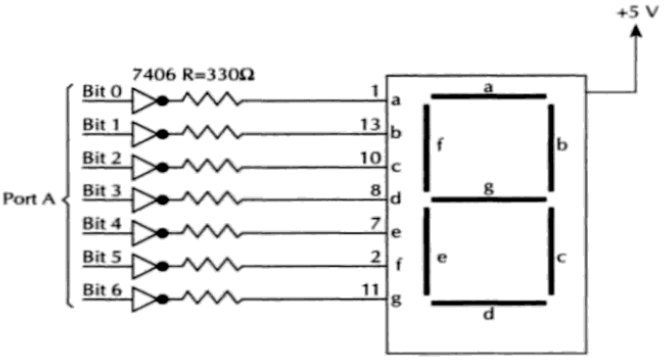
Write an 8085 assembly language program starting at address 5000H to turn on an LED connected to bit 4 of the 8155 I/O port B. Use address of port B as 22_{16}.

Solution

```
        ORG  5000H
Port B  EQU  22H
        MVI  A,02H    ;  Configure
        OUT  CSR      ;  Port B as output
        MVI  A,10H    ;  Output HIGH
        OUT  PORT B   ;  to LED
        HLT
```

Example 2.9

An 8085-8355-based microcomputer is required to drive a common anode seven-segment display connected to port A as follows:



Write an 8085 assembly language program starting at address at 5000H to display a single hexidecimal digit (0 to F) from location 4000_{16}. Use a look-up table. Note that the 7406 shown in the figure contains six inverting buffers on one 7406. Therefore, two 7406 or one 7406 and one transistor are required for the seven segments. Use port addresses of DDRA and port A as 42H and 40H, respectively.

Solution

The decode table can be obtained as follows:

	Bits								
Hex digit	7	6	5	4	3	2	1	0	Decode byte
		g	f	e	d	c	b	a	
0	0	0	1	1	1	1	1	1	3F
1	0	0	0	0	0	1	1	0	06
2	0	1	0	1	1	0	1	1	5B
3	0	1	0	0	1	1	1	1	4F
4	0	1	1	0	0	1	1	0	66
5	0	1	1	0	1	1	0	1	6D
6	0	1	1	1	1	1	0	1	7D
7	0	0	0	0	0	1	1	1	07
8	0	1	1	1	1	1	1	1	7F
9	0	1	1	0	0	1	1	1	67
A	0	1	1	1	0	1	1	1	77
B	0	1	1	1	1	1	0	0	7C
C	0	0	1	1	1	0	0	1	39
D	0	1	0	1	1	1	1	0	5E
E	0	1	1	1	1	0	0	1	79
F	0	1	1	1	0	0	0	1	71

```
        ORG     5000H
DDRA    EQU     42H
Port A  EQU     40H
MVI     A,7FH
OUT     DDRA    ; Port A
LXI     H,4015H ; Load HL with
                ; starting address of table
LXI     D,0000H ; Load 0000H to DE
```

```
        LDA     DIGIT    ; Load digit to be
                         ; Displayed into A
        MOV     E,A      ; Move digit to E
        DAD     D        ; Determine digit address
        MOV     A,M      ; Load decode byte to A
        OUT     Port A   ; Outport decode byte
        HLT              ; to display
        ORG     4015H
TABLE   DB      3FH, 06H, 5BH, 4FH
        DB      66H, 6DH, 7DH, 07H
        DB      7FH, 67H, 77H, 7CH
        DB      39H, 5EH, 79H, 71H
        ORG     4000H
DIGIT   DB      DATA
```

2.9.2 8085 Interrupt System

The 8085 chip has five interrupt pins, namely, TRAP, RST7.5, RST6.5, RST5.5, and INTR. If the signals on these interrupt pins go to HIGH simultaneously, then TRAP will be serviced first (i.e., highest priority) followed by RST7.5, RST6.5, RST5.5, and INTR. Note that once an interrupt is serviced, all the interrupts except TRAP are disabled. They can also be enabled or disabled simultaneously by executing the EI or DI instruction respectively. The 8085 interrupts are

1. TRAP — TRAP is a nonmaskable interrupt. That is, it cannot be enabled or disabled by an instruction. In order for the 8085 to service this interrupt, the signal on the TRAP pin must have a sustained HIGH level with a low to high transition. If this condition occurs, then the 8085 completes execution of the current instruction, pushes the program counter onto the stack, and branches to location 0024_{16} (interrupt address vector for the TRAP). Note that the TRAP interrupt is cleared by the falling edge of the signal on the pin.
2. RST7.5 — RST7.5 is a maskable interrupt. This means that it can be enabled or disabled using the SIM or EI/DI instruction. The 8085 responds to the RST7.5 interrupt when the signal on the RST7.5 pin has a low to high transition. In order to service RST7.5, the 8085 completes execution of the current instruction, pushes the program counter onto the stack, and branches to $003C_{16}$. The 8085 remembers the RST7.5 interrupt by setting an internal D flip-flop by the leading edge.
3. RST6.5 — RST6.5 is a maskable interrupt. It can be enabled or disabled using the SIM or EI/DI instruction. RST6.5 is HIGH level sensitive. In order to service this interrupt, the 8085 completes execution of the current instruction, saves the program counter onto the stack, and branches to location 0034_{16}.
4. RST5.5 — RST5.5 is a maskable interrupt. It can be enabled or disabled by the SIM or EI/DI instruction. RST5.5 is HIGH level sensitive. In order to service this interrupt, the 8085 completes execution of the current instruction, saves the program counter onto the stack, and branches to $002C_{16}$.
5. INTR — INTR is a maskable interrupt. It can be enabled or disabled by EI or DI instruction. This is also called the handshake interrupt. INTR is HIGH level sensitive. When no other interrupts are active and the signal on the INTR pin is HIGH, the 8085 completes execution of the current instruction, and generates an interrupt acknowledge, $\overline{\text{INTA}}$, LOW pulse on the control bus. The 8085 then expects either a 1-byte CALL (RST0 through RST7) or a 3-byte CALL on the data lines. This instruction must be provided by external hardware. In other words, the INTA can be used to enable a tristate

buffer. The output of this buffer can be connected to the 8085 data lines. The buffer can be designed to provide the appropriate op code on the data lines. Note that the occurrence of $\overline{\text{INTA}}$ turns off the 8085 interrupt system in order to avoid multiple interrupts from a single device. Also note that there are eight RST instructions (RST0 through RST7). Each of these RST instructions has a vector address. These were shown in Table 2.9.

In response to a HIGH on the INTR, the 8085 proceeds with the sequence of events described below. If INTR is the only interrupt and if the 8085 system interrupt is enabled by executing the EI instruction, the 8085 will turn off the system interrupt and then make the $\overline{\text{INTA}}$ LOW for about two cycles. This $\overline{\text{INTA}}$ signal can be used to enable an external hardware to provide an op code on the data bus. The 8085 can then read this op code. Typically, the 1-byte RST or 3-byte CALL instruction can be used as the op code. If the 3-byte CALL is used, then the 8085 will generate two additional $\overline{\text{INTA}}$ cycles in order to fetch all 3 bytes of the instruction. However, on the other hand, if RST is used, then no additional $\overline{\text{INTA}}$ is required. The call op code is normally placed on the data bus by the 8259 programmable interrupt controller. At this point, only the op code for the CALL (CD16) is fetched by the 8085. The 8085 executes this instruction and determines that it needs two more bytes (the address portion of the 3-byte instruction). The 8085 then generates a second $\overline{\text{INTA}}$ cycle followed by a third $\overline{\text{INTA}}$ cycle in order to fetch the address portion of the CALL instruction from the 8259. The 8085 executes the CALL instruction and branches to the interrupt service routine located at an address specified in the CALL instruction. Note that the recognition of any maskable interrupt (RST7.5, RST6.5, RST5.5, and INTR) disables all maskable interrupts to avoid multiple interrupts from the same device.

Therefore, in order that the 8085 can accept another interrupt, the last two instructions of the interrupt service routine will be EI followed by RET.

One can produce a single RST instruction, say RST7 (op code FFH in HEX), using 74LS244. The inputs I0 to I7 of the 74LS244 (Figure 2.21) are connected to HIGH and its enable line ($\overline{\text{OE}}$) is tied to an $\overline{\text{INTA}}$. In response to INTA LOW, the 74LS244 places FF in hex (RST7) on the data bus. Figure 2.21 shows a typical circuit. Figure 2.22 provides a schematic for eight priority interrupts.

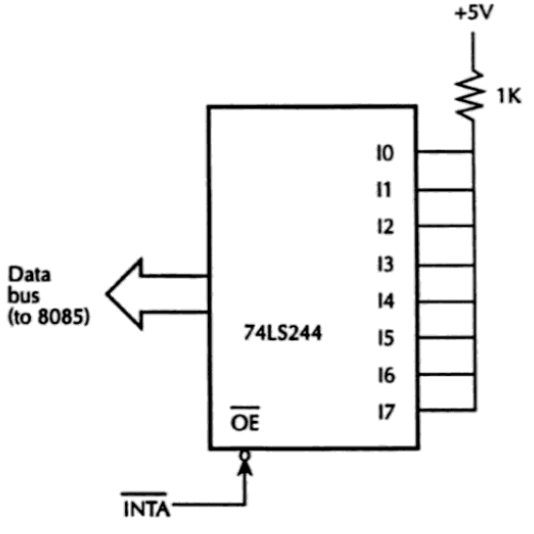


FIGURE 2.21 Using an octal buffer to provide RST7 instruction.

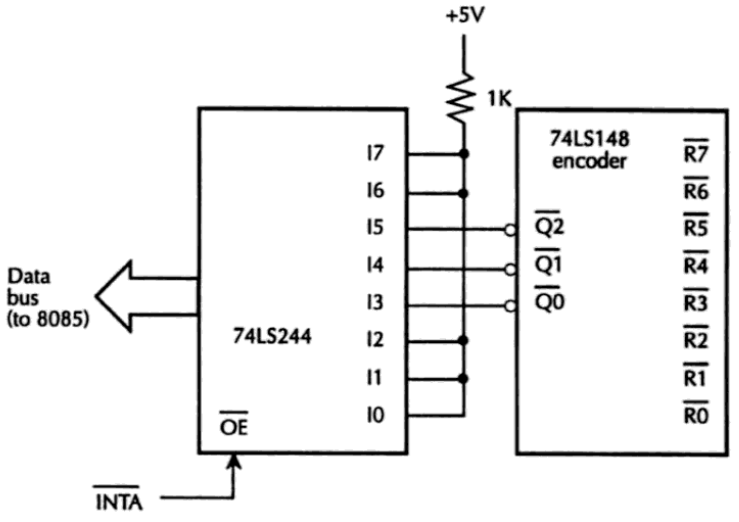


FIGURE 2.22 Forming eight RST instructions with a priority encoder.

Let us elaborate on Figure 2.23. Execution of the EI instruction sets the RS flip-flop of Figure 2.23 and makes one of the inputs to the AND gates #1 through #4 HIGH. Hence, in order for all the interrupts (except TRAP) to work, the interrupt system must be enabled. Execution of Disable Interrupts (DI) clears the RS flip-flop and disables all interrupts except TRAP. The SIM instruction outputs the contents of the accumulator which can be interpreted as shown in Figure 2.24.

The interrupt mask function is only executed if the mask set enable bit is 1. Suppose that if 06_{16} is stored in the accumulator and the SIM instruction is executed. A 1 will be sent to the interrupt mask for RST7.5 and RST6.5, and 0 will be sent to RST5.5. That is, in Figure 2.23, a 1 will be sent to the inputs of the AND gates #1 and #2, and a 0 will be sent to the AND gate #3, then inverted at the AND gate inputs (shown by circles), giving two LOW outputs disabling RST7.5, RST6.5, and a HIGH input to AND gate #3. Therefore, in order to enable RST7.5, RST6.5, or RST5.5, the interrupt system must be enabled by executing EI, the appropriate interrupt mask bit must be LOW by executing SIM, and the appropriate interrupt signal (leading edge or high level) at the respective pins must be available. For example, consider the RST7.5 interrupt. When the EI and SIM instructions are executed, the interrupt system can be enabled and also the interrupt mask bit for RST7.5 can be set to LOW, making the two inputs to AND gate #1 HIGH. The third input to this AND gate can be set to a HIGH by a leading edge at the RST7.5 pin. This sets the D flip-flop, thus making the output of the AND gate #1 HIGH, enabling RST7.5. The 8085 branches to location 003C16 where a 3-byte JMP instruction takes the program to the service routine. The RST5.5 and RST6.5 can similarly be explained from Figure 2.23.

The RIM instruction can be used to check whether one or more of the RST7.5, RST6.5, and RST5.5 interrupts are waiting to be serviced. The RIM instruction also provides the status of the mask bits for RST7.5, RST6.5, and RST5.5 and the status of the SID pin (HIGH or LOW). After execution of the RIM instruction, the 8085 loads the accumulator with 8 bits of data which can be interpreted as shown in Figure 2.25.

The bits are interpreted as follows:

Mask bits — Bits 0, 1, and 2 are the status of the mask bits for RST5.5, RST6.5, and RST7.5.

Interrupt Enable bit — This bit indicates whether one maskable interrupt capability is enabled (if this bit = 1) or disabled (if this bit = 0).

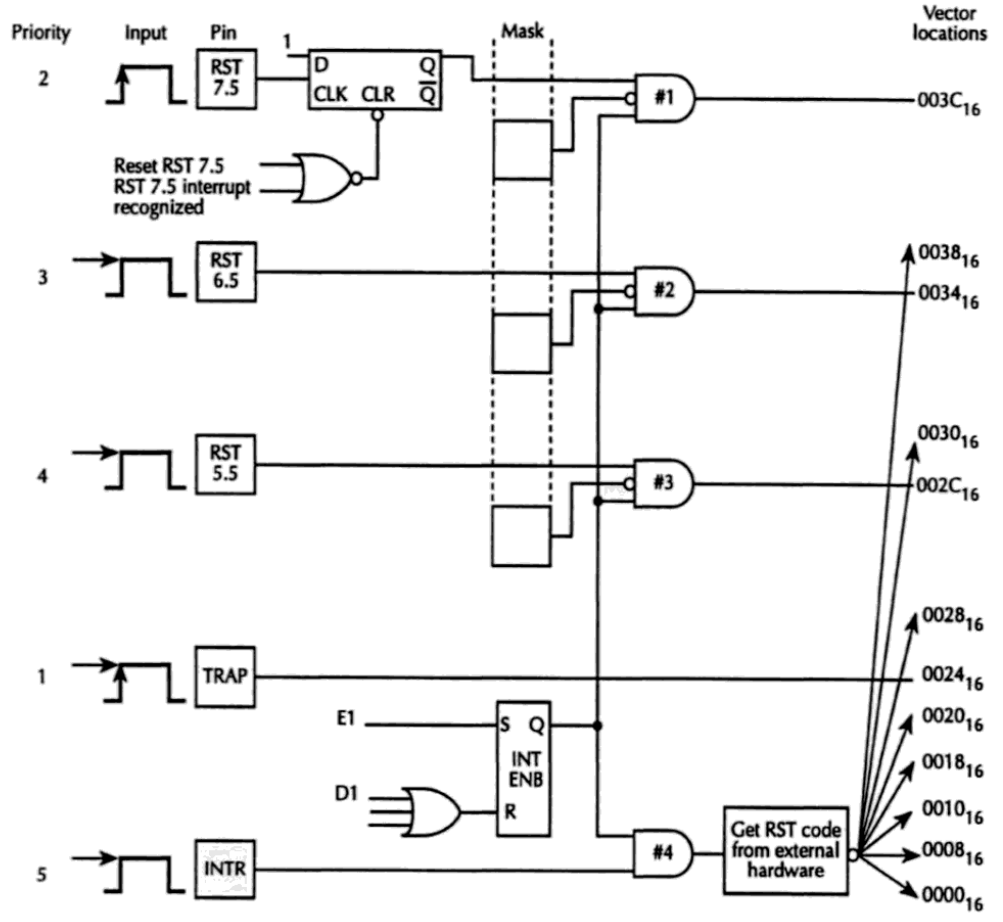


FIGURE 2.23 8085 interrupt structure.

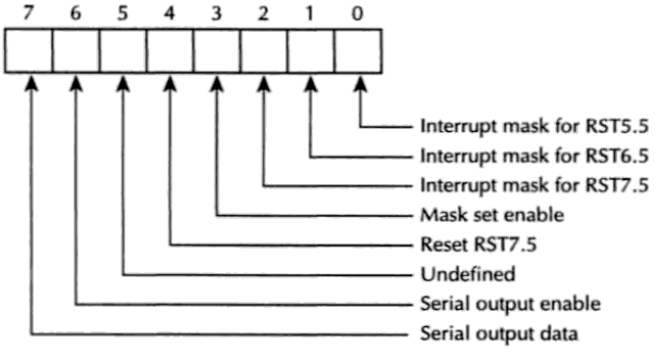


FIGURE 2.24 Interpretation of data output by the SIM instruction.

Interrupt Pending bits — A one in a particular bit position indicates that the particular RST is waiting to be serviced, while a 0 indicates that no interrupt is pending.

Serial Input data — These data indicate the status of the SID pin.

The RIM instruction can be used to read the interrupt pending bits in the accumulator. These bits can then be checked by software to determine whether any higher priority interrupts

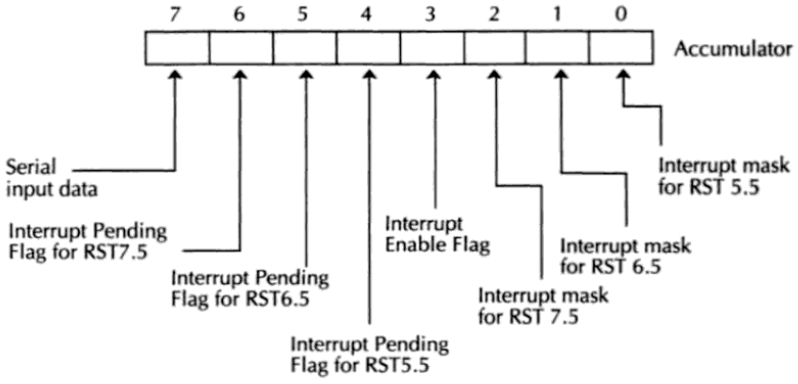


FIGURE 2.25 Execution of RIM instruction.

are pending. For example, suppose that RST6.5 and RST5.5 interrupts are used in a system. If RST5.5 interrupt first occurs, assuming no other higher level interrupts, the 8085 will execute the RST5.5 service routine. Note that while in this service routine, the RST6.5 interrupt (higher priority than RST5.5 interrupt) may occur. Since the 8085's maskable interrupt capability is disabled, the user can check whether the RST6.5 is pending while executing the RST5.5 service routine in one of two ways:

- By executing EI at the beginning of the RST5.5 service routine provided that the level at RST5.5 pin is LOW (RST5.5 disabled). This will service the RST6.5 immediately if pending and will then complete RST5.5 service routine (nested interrupts).
- By executing RIM and using rotate instruction to check whether RST6.5 is pending. The RIM instruction can be executed at several places in the RST5.5 service routine. If the RST6.5 is found to be pending, the interrupt can be enabled by executing EI in the RST5.5 service routine.

2.9.3 8085 DMA

The Intel 8257 DMA controller chip is a 40-pin DIP and is programmable. It is compatible with the 8085 microprocessor. The 8257 is a four-channel DMA controller with priority logic built into the chip. This means that the 8257 provides for DMA transfers for a maximum of up to four devices via the DMA request lines DRQ0 to DRQ3 (DRQ0 has the highest priority and DRQ3 the lowest). Associated with each DRQ is a DMA acknowledge ($\overline{\text{DACK0}}$ to $\overline{\text{DACK3}}$ for four DMA requests DRQ0 to DRQ3). Note that the $\overline{\text{DACK}}$ signals are active LOW. The 8257 uses the 8085 HOLD pin in order to take over the system bus. After initializing the 8257 by the 8085, the 8257 performs the DMA operation in order to transfer a block of data of up to 16,384 bytes between the memory and a peripheral without involving the microprocessor. A typical 8085-8257 interface is shown in Figure 2.26. An I/O device, when enabled by the 8085, can request a DMA transfer by raising the DMA request (DRQ) line of one of the channels of the 8257. In response, the 8257 will send a HOLD request (HRQ) to the 8085. The 8257 waits for the HOLD acknowledge (HLDA) from the 8085. On receipt of HLDA from the 8085, the 8257 generates a LOW on the DACK lines for the I/O device. Note that DACK is used as a chip select bit for the I/O device. The 8257 sends the READ or WRITE control signals, and data are transferred between the I/O and memory. On completion of the data transfer, the $\overline{\text{DACK0}}$ is set to HIGH, and the HRQ line is reset to LOW in order to transfer control of the bus to the 8085. The 8257 utilizes four clock cycles in order to transfer 8 bits of data.

The 8257 has three main registers. These are a 16-bit DMA address register, a terminal count register, and a status register. Both address and terminal count registers must be initialized

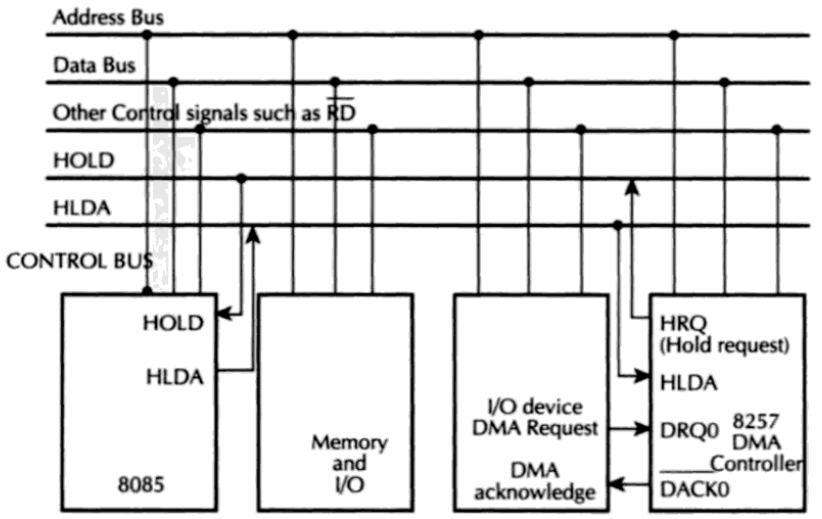


FIGURE 2.26 An 8085-8257 interface.

before a DMA operation. The DMA address register is initialized with the starting address of the memory to be written into or read from. The low-order 14 bits of the terminal count register are initialized with the value (n – 1), where n is the desired number of DMA cycles. A Terminal Count (TC) pin on the 8257 is set to HIGH in order to indicate to the peripheral device that the present DMA cycle is the last cycle. An 8-bit status register in the 8257 is used to indicate which channels have attained a terminal count.

2.9.4 8085 SID and SOD Lines

Serial I/O is extensively used for data transfer between a peripheral device and the microprocessor. Since microprocessors perform internal operations in parallel, conversion of data from parallel to serial and vice versa is required to provide communication between the microprocessor and the serial I/O. The 8085 provides serial I/O capabilities via SID (Serial Input Data) and SOD (Serial Output Data) lines.

One can transfer data to or from the SID or SOD lines using the instruction RIM and SIM. After executing the RIM instruction, the bits in the accumulator are interpreted as follows:

1. Serial input bit is bit 7 of the accumulator.
2. Bits 0 to 6 are interrupt masks, the interrupt enable bit, and pending interrupts.

The SIM instruction sends the contents of the accumulator to the interrupt mask register and serial output line. Therefore, before executing the SIM, the accumulator must be loaded with proper data. The contents of the accumulator are interpreted as follows:

1. Bit 7 of the accumulator is the serial output bit.
2. The SOD enable bit is bit 6 of the accumulator. This bit must be 1 in order to output bit 7 of the accumulator to the SOD line.
3. Bits 0 to 5 are interrupt masks, enables, and resets.

Example 2.10

An 8085/8155-based microcomputer is required to input a switch via the SID line and output the switch status to an LED connected to the SOD line. Write an 8085 assembly language program to accomplish this.

Solution

```
        ORG   5000H
START   RIM           ; Bit 7 of 'A' is SID
        ORI   40H     ; Set SOD Enable to one
        SIM           ; output to LED
        JMP   START   ; REPEAT
```

Example 2.11

Write an 8085 assembly language program to implement the following requirements. i) if V1 > V2, the 8085/8155-based microprocessor system will read the switch input from port B. If switch is open (input high), turn the LED off. If switch is closed (input low), turn the LED on. ii) Repeat i) by using a) TRAP, b) RST6.5 with SID/SOD, c) INTR.

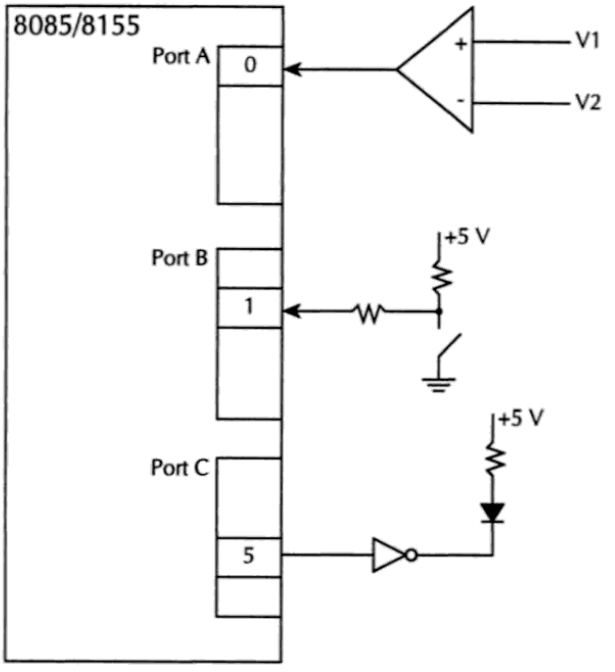
Assume

CSR = 00H
Port A = 01H
Port B = 02H
Port C = 03H

Write all programs starting at 2000H and service routines at 3000H. Intialize SP to 20C0H.

i) *Solution*

```
        ORG   2000H
CSR     EQU   00H      ; Define CSR address.
PORTA   EQU   01H      ; Define Port A address.
PORTB   EQU   02H      ; Define Port B address.
PORTC   EQU   03H      ; Define Port C address.
        MVI   A,0CH    ; Set Port A/Port B input and Port
                         C output.
        OUT   CSR      ; Write to CSR.
        OUT   PORTC    ; Initialize bit 5 of port C. Turn
                         LED off.
SCAN    IN    PORTA    ; Get the data from Port A.
        RAR            ; Move the bit0 of A to Cy.
        JNC   SCAN     ; Go to SCAN if V1 < V2. Go to
                         next if V1>V2.
        IN    PORTB    ; Get the switch input from Port B.
        XRI   02H      ; Invert the input data.
        RAL            ; Move the data in bit 1 of Port A
                         to bit 5.
        RAL            ;
        RAL            ;
        RAL            ;
        OUT   PORTC    ; Write to Port C.

        JMP   SCAN     ; Loop
```



ii) *Solution*

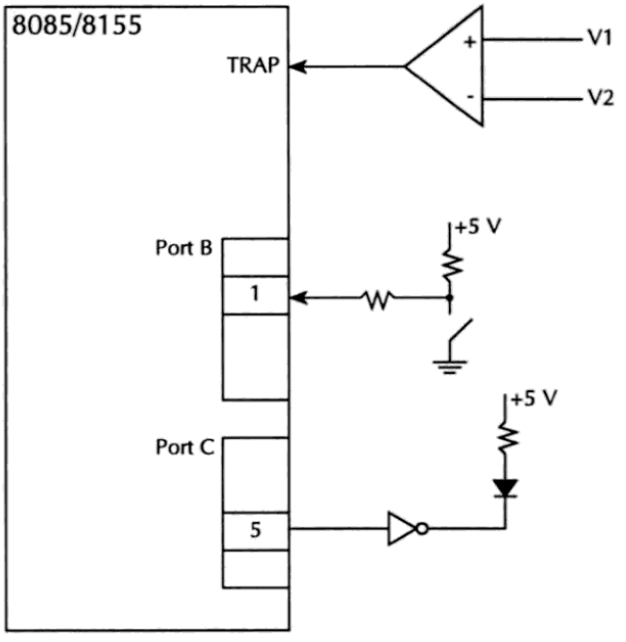
a)

```
        ORG   0024H      ; Trap vector address entry.
        JMP   3000H      ; Go to trap service routine.
        ORG   2000H
CSR     EQU   00H        ; Define CSR address.
PORTB   EQU   02H        ; Define Port B address.
PORTC   EQU   03H        ; Define Port C address.
START   LXI   SP,20C0H   ; Initialize SP.
        MVI   A,0CH      ; Set Port B input and Port C
                           output.
        OUT   CSR        ; Write to CSR.
        OUT   PORTC      ; Initialize bit 5 of Port C. Turn
                           LED off.
LOOP    NOP              ; Wait for TRAP request.
        JMP   Loop
        ORG   3000H      ; TRAP service routine entry.
        IN    PORTB      ; Get the switch input from Port
                           B.
        XRI   02H        ; Invert the input data.
        RAL              ; Move the data in bit 1 of A to
                           bit 5.
        RAL              ;
        RAL              ;
        RAL              ;
        OUT   PORTC      ; Write to Port C.
        JMP   START
```

ii) *Solution*

b)

```
        ORG   0034H      ; RST6.5 vector address entry.
```

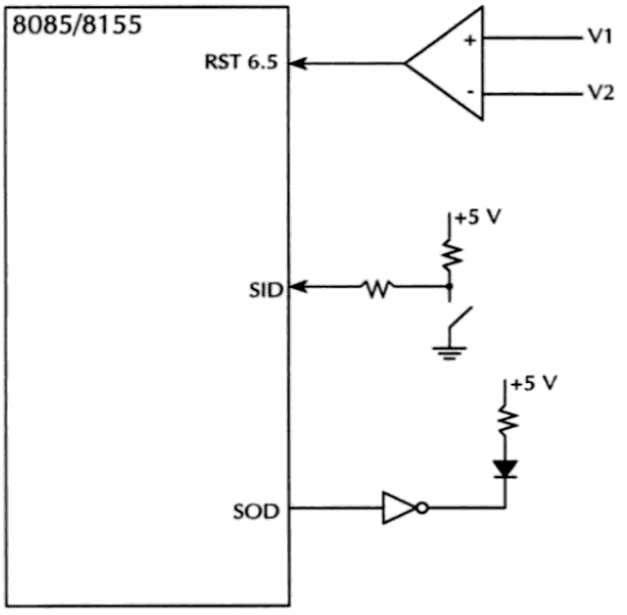


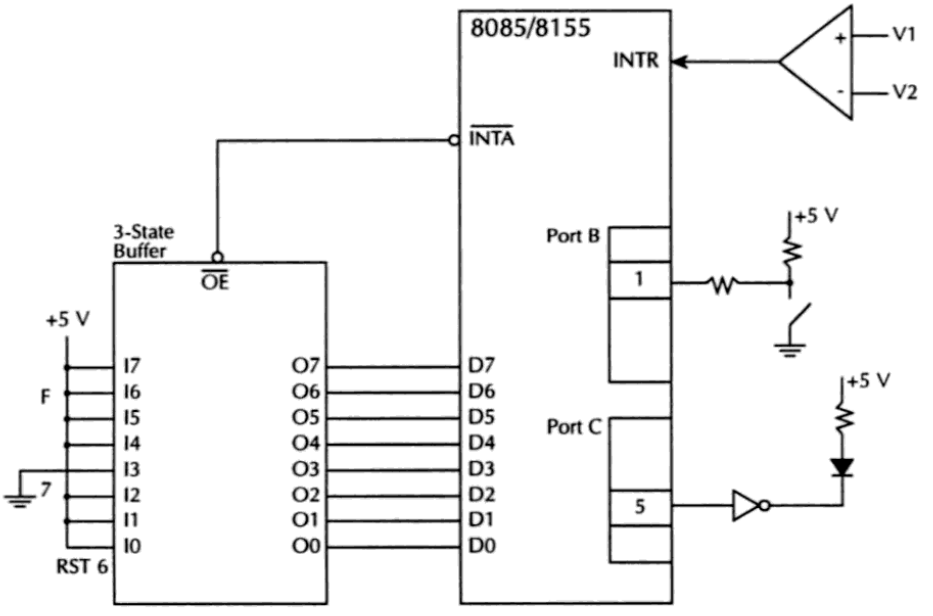

```
        JMP   3000H     ; Go to RST6.5 service routine.
        ORG   2000H
START   LXI   SP,20C0H  ; Initialize SP.
        MVI   A,40H     ; Initialize SOD. Turn LED off.
        SIM             ;
                        ;
        MVI   A,0DH     ; Set data for RST6.5.
        SIM             ; Enable RST6.5.
        EI              ; Enable interrupt.
LOOP    NOP
        JMP   LOOP      ; Wait for RST6.5 request.
        ORG   3000H     ; RST6.5 service routine entry.
        RIM             ; Get the data from SID>
        XRI   80H       ; Invert the input data in bit
                          7 of A.
        ORI   40H       ; Enable SOD.
        SIM             ; Write to SOD.
        JMP   START
```

ii) *Solution*

c)

```
        ORG   0030H     ; RST6 vector address entry
        JMP   3000H     ; Go to RST6 service routine
        ORG   2000H
CSR     EQU   00H       ; Define CSR address.
PORTB   EQU   02H       ; Define Port B address.
PORTC   EQU   03H       ; Define Port C address.
BEGIN   LXI   SP,20C0H  ; Initialize SP.
        MVI   A,0CH     ; Set Port B input and Port C
                          output.
        OUT   CSR       ; Write to CSR.
```






```
        OUT   PORTC     ; Initialize bit 5 of port C. Turn
                          LED off.
        EI              ; Enable interrupt.
LOOP    NOP             ; Wait for INTR request.
        JMP   LOOP
        ORG   3000H     ; RST6 service routine entry.
        IN    PORTB     ; Get the switch input from Port
                          B.
        XRI   02H       ; Invert the input data.
        RAL             ; Move the data in bit 1 of A to
                          bit 5.
```

```
RAL            ;
RAL            ;
RAL            ;
OUT    PORTC   ; Write to Port C.
JMP    BEGIN
```

2.10 8085-Based System Design

In order to illustrate the concepts associated with 8085-based system design, a microcomputer with 2K EPROM (2716), 256 byte RAM, and 3 ports (8155) is designed. A hardware schematic is included. Also, an 8085 assembly language program is provided to multiply 4-bit unsigned numbers entered via DIP switches connected to port A. The 8-bit product is displayed on two seven-segment displays interfaced via port B. Repeated addition will be used for multiplication. Figure 2.27 shows a schematic of the hardware design. Full decoding using the 74LS138 decoder is utilized. Texas Instruments TIL 311's displays with on-chip decoder are used. The memory and I/O maps are given in the following:

1. Memory map

2716

A15	A14	A13	A12	A11	A10	A9	A8	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0	0	0	0	0	{- - - - - - - - - - - - - - - - - all zeros to ones - - - - - - - - - - - -}										

Result 0000H-07FFH

8155

A15	A14	A13	A12	A11	A10	A9	A8	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0	0	0	0	1	0	0	0	{- - - - - - - - - - - all zeros to ones - - - - - - - - }							

Result 0800H-08FFH

2. I/O map using standard I/O

Ports

A15 AD7	A14 AD6	A13 AD5	A12 AD4	A11 AD3	A10 AD2	A9 AD1	A8 AD0		
0	0	0	0	1	0	0	0	CSR	08H
0	0	0	0	1	0	0	1	Port A	09H
0	0	0	0	1	0	1	0	Port B	0AH
0	0	0	0	1	0	1	1	Port C	0BH

A listing of the 8085 assembly language program for performing 4 bit × 4 bit multiplication is provided below:

```
CSR EQU   08H
PORT A    EQU    09H
PORT B    EQU    0AH
REPEAT    MVI    A,02H  ; Configure Port A as input
          OUT    CSR    ; and Port B as output
          MVI    L,00H  ; Initialize product to zero
          IN     Port A ; Input multiplier and
                          multiplicand
```

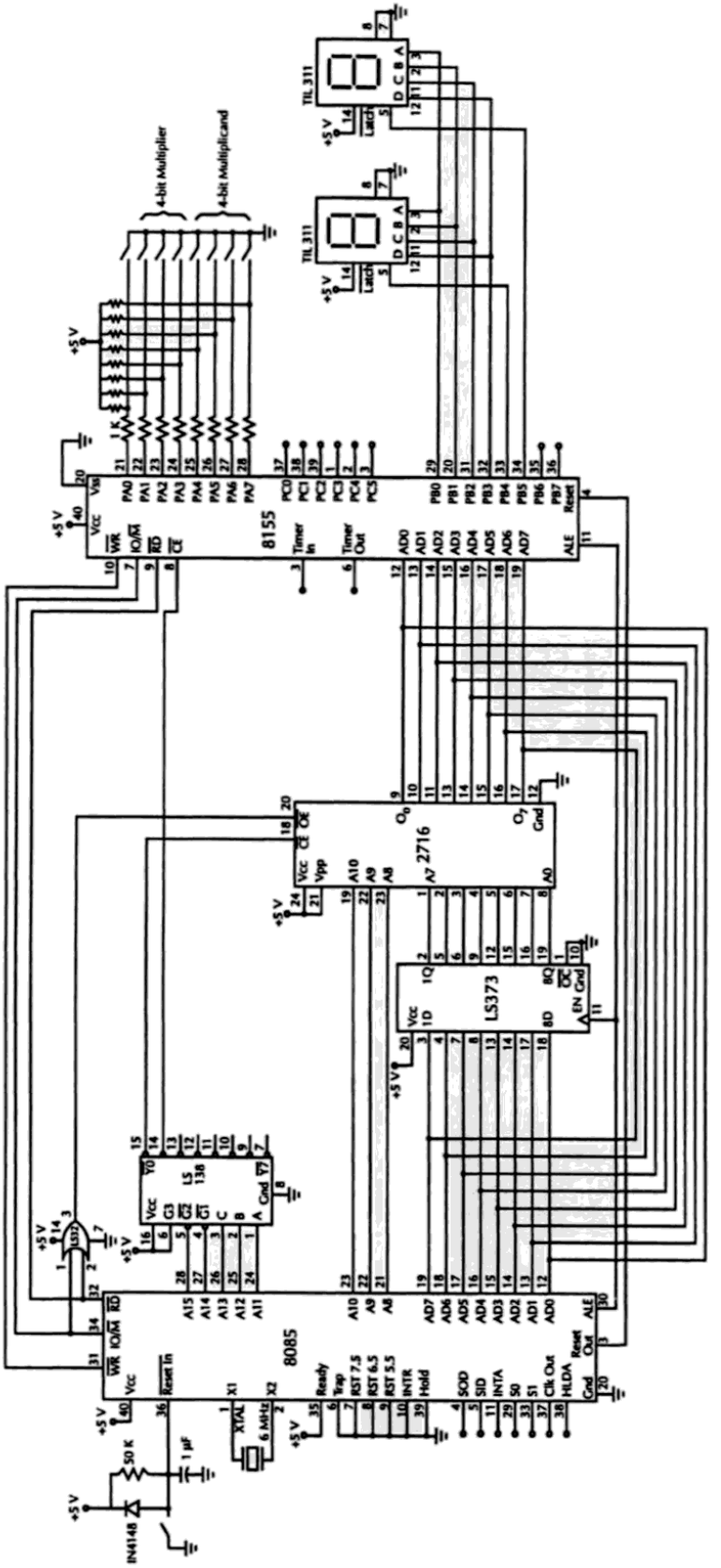


FIGURE 2.27 8085-based system design.

```
        MOV   B,A          ; Save multiplicand and
                             multiplier in B
        ANI   OFH          ; Mask multiplicand and retain
                             multiplier
        MOV   C,A          ; Save 4-bit multiplier in C
        MOV   A,B          ; Move multiplicand and
                             multiplier to A
        RAR                ; Move 4-bit multiplicand
        RAR                ; (upper nibble of A)
        RAR                ; into LOW nibble
        RAR                ; of
                           ; accumulator
        ANI   OFH          ; Mask high nibble and retain
        MOV   D,A          ; multiplicand
START   ADD   D            ; Perform repeated addition
        DCR   C            ; Decrement multiplier value
        MOV   L,A          ; Save sum
        JNZ   START        ; if Z = 0, repeat addition
                           ; else product in L
        MOV   A,L          ; Move product to A
        ANI   OFH          ; Retain product LOW nibble and
        MOV   H,A          ; Save in H
        MOV   A,L          ; Move product to A
        RAR                ; Move
        RAR                ; high nibble
        RAR                ; of product
        RAR                ; to LOW nibble
                           ; of A
        ANI   2FH          ; Retain high nibble of
                             product and enable latch
        OUT   PORTB        ; of high hex display and
                             disable low latch
                           ; Display high nibble of product
        MOV   A,H          ; Move LOW product nibble to A
        ANI   1FH          ; Enable LOW latch and disable
                             high latch
        OUT   PORTB        ; Display LOW nibble of product
        JMP   REPEAT       ; On low display and continue
```

The above program can be assembled. The 2716 can then be programmed by using an EPROM programmer with the machine code of the preceding program starting at location 000H. Then upon activation of the switch at the 8085 reset input, the dip switch low and high nibbles will be multiplied and the result of the multiplication will be displayed on the two TIL311s. By changing DIP switch inputs at port A, new results can be displayed.

Questions and Problems

2.1 Compare the main differences of 8085AH, 8085AH-2, and 8085AH-1 with the 8085.

2.2 What is the primary purpose of the 8085 H-L pair with respect to external memory? List two of its main functions.

2.3 Identify the addressing modes of the following instructions:

i) `MOV A, M`
ii) `RAR`
iii) `STAX D`
iv) `LDA START`

2.4 Assume register pair BC contains $7F02_{16}$. For the following 8085 assembly language program, determine the carry, zero, parity, and sign flags after execution of the MVI A,05H instruction:

```
ORG   4000H
LXI   SP, 2050H
PUSH  B
MVI   A,02H
ADI   03H
POP   PSW
MVI   A, 05H
HLT
```

2.5 What function is performed by each of the following 8085 instructions:

i) `XOR A`
ii) `MOV D,D`
iii) `DAD H`

2.6 i) What is the function of the 8085 ALE pin?
ii) If a crystal of 4 MHz is connected to the 8085 X1X2 pins, what is the 8085 internal clock period?

2.7 Write an 8085 assembly language program to divide an 8-bit unsigned number in accumulator by 4. Neglect remainder. Use minimum number of instructions. Store result in A.

2.8 Using the simplest possible algorithm, write an 8085 assembly language program with a minimum number of instructions to divide a 16-bit unsigned number in DE by 16. Neglect remainder. Store result in DE.

2.9 Assume the contents of DE are 2050H. Write an 8085 assembly language program to unconditionally branch to 2050H. Do not use any conditional or unconditional Jump or CALL or return instructions. Use three instructions maximum.

2.10 Write an 8085 assembly language program using a minimum number of instructions to add the 16-bit numbers in BC, DE, and HL. Store the 16-bit result in DE.

2.11 Write an 8085 assembly language program to add two 24-bit numbers located in memory address 2000_{16} through 2005_{16} with the most significant byte of the first number in 2000_{16} and the most significant byte of the second number in 2003_{16}. Store the 24-bit result in three consecutive bytes starting at address 2000_{16}.

2.12 Write a subroutine in 8085 assembly language to divide a 16-bit unsigned number in BC by 32_{10}. Neglect the remainder. Use the simplest possible algorithm. Also, write the main program in 8085 assembly language to perform all the initializations. The main program will then call the subroutine, store the 16-bit quotient in DE, and stop.

2.13 Write an 8085 assembly language program to shift the contents of the DE register twice to left without using any ROTATE instructions. After shifting, if the contents of DE are nonzero, store $FFFF_{16}$ in the HL pair. On the other hand, if the contents of DE pair are zero, then store 0000_{16} in the HL pair.

2.14 Write an 8085 assembly language program to check the parity of an 8-bit number in "A" without using any instructions involving the parity flag. Store EE in location 3000 if the parity is even; otherwise, store DD in location 3000_{16}.

2.15 Write an 8085 assembly language program to move a block of data of length 100_{10} from the source block starting at 2000_{16} to the destination block starting at 3000_{16}.

2.16 Write a subroutine in 8085 assembly language to divide an 8-bit unsigned number Xi by 2. Also, write the main program in 8085 assembly language which will call the subroutine to compute

$$\sum_{i=1}^{3} \frac{Xi}{2}$$

Store the result in location 5000_{16}. Use a minimum number of instructions.

2.17 Assume an 8085/8355-based microcomputer. Suppose that four switches are connected to bits 0 through 3 of port A, an LED to bit 4 of port A, and another LED to bit 2 of port B. If the number of low switches is even, turn the port A LED ON and port B LED OFF. If the number of low switches is odd, turn port A LED OFF and port B LED ON. Write an 8085 assembly language program to accomplish the above.

2.18 Draw a simplified diagram using an 8085, one 8156, and one 8355 to include the following memory map and I/O ports:

i) *8355* *8156*
2000_{16} thru $27FF_{16}$ 4000_{16} thru $40FF_{16}$

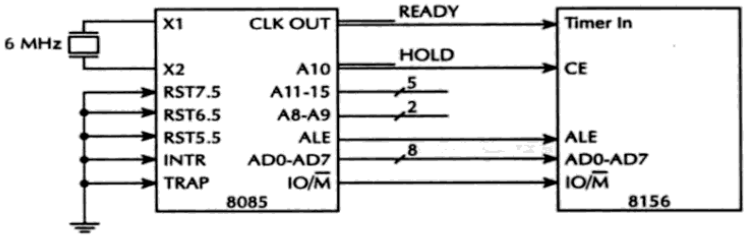
ii) Using memory-mapped I/O, configure ports A and B of the 8355 at addresses 8000_{16} and 8001_{16}. Show only connections for the lines which are pertinent.

2.19 It is desired to interface a pump to a 8085/8355/8156-based microcomputer. The pump will be started by the microcomputer at the trailing edge of a start pulse via the SOD pin. When the pump runs, a HIGH status signal from the pump called "Pump Running" will be used to interrupt the microcomputer via its INTR interrupt. In response, the microcomputer will turn an LED on connected to bit 7 of the 8355 port A. Assume DDRA as the data direction register for this port.

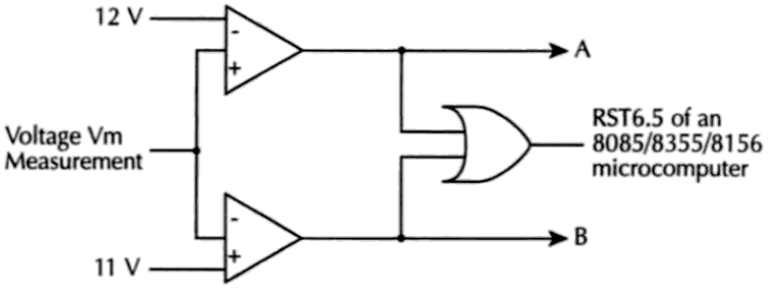
i) Draw a simplified schematic for accomplishing the above.

ii) Write the main and interrupt service routines using the 8085 assembly language programs.

2.20 Will the circuit shown work? If so, determine the memory and I/O maps in hex. If not, justify briefly, modify the circuit and then determine the memory and I/O maps. Use only the pins and signals as shown. Also, assume all don't cares to be zeros.



2.21

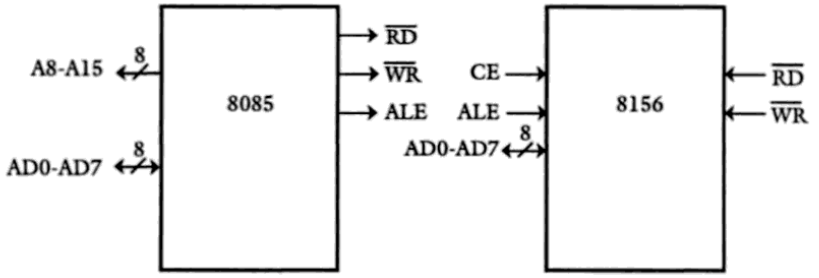


In the above, if Vm > 12 V, turn an LED ON connected at bit 3 of port A. On the other hand, if Vm < 11 V, turn the LED OFF. Use ports, registers, and memory locations as needed.

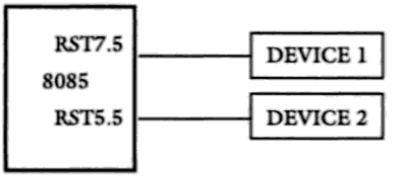
i) Draw a hardware block diagram showing the microcomputer and the connections of the above diagram to its ports.

ii) Write main program and service routines in 8085 assembly language.

2.22 Interface the following 8156 RAM chip to obtain the following memory map: 0400H thru 04FFH. Show only the connections for the pins which are shown. Assume all unused address lines to be zero.

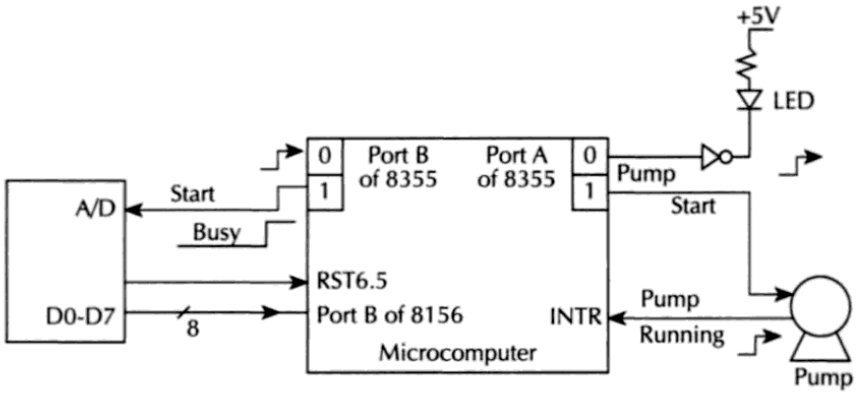


2.23



If device 2 is presently being serviced by the 8085 and the device 1 interrupt occurs, explain briefly what the user needs to do in the service routine of device 2 in order that device 1 will be serviced before device 2.

2.24 Assume an 8085/8355 microcomputer. Suppose that two switches are connected to bit 1 of Port A and the SID line. Also, an LED is connected to the SOD line. It is desired to turn the LED ON if both switches are HIGH; otherwise the LED is to be turned off. Write an 8085 assembly language program to input both switches and turn the LED ON or OFF based on the above conditions.



2.25 It is desired to interface a pump and an A/D converter to an 8085/8355/8156-based microcomputer as follows: The pump can be started by a HIGH at the pump start signal. If the pump starts, the pump-running signal goes to HIGH; otherwise the pump running stays LOW. The A/D converter can be started by a HIGH and the conversion is completed when the Busy signal is HIGH. Start the pump and A/D converter at nearly the same time. If the pump runs, the LED is to be turned on. On the other hand, if the Busy signal goes to HIGH, the A/D converter's output is to be read.

i) Draw a simplified hardware schematic to accomplish the above.
ii) Write main program and service routines in 8085 assembly language. Include the interrupt priority concept in the programs.

2.26 Assume an 8085/8355-based microcomputer. Write an 8085 assembly language program to turn an LED (LED 1) ON connected to bit 3 of port A if a 16-bit number in DE register pair is negative after shifting once to the left; if the number is positive after shifting, turn the LED (LED 1) OFF. Also, after shifting, if there is a sign change of the 16-bit number, turn another LED (LED 2) ON connected to bit 1 of port A; otherwise, turn the LED (LED 2) OFF.

2.27 Design an 8085/2716/8155-based microcomputer to include the following:

i) 4K EPROM, 512 bytes static RAM, four 8-bit and two 6-bit ports. Use standard I/O and linear decoding.
ii) Repeat i) except use fully decoding using a 3×8 decoder.
iii) Repeat i) except use memory-mapped I/O.
iv) Repeat i) except use memory-mapped I/O and full decoding.

Draw a neat schematic of each design and also determine memory and I/O maps in each case.

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, 6116, and 8255 are included.

A design technique is presented showing interconnection of the 8086 to 2716 EPROM, 6116 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.

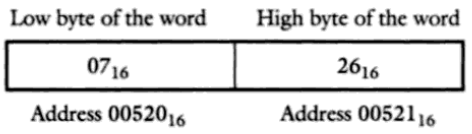
The 8086 is designed using the HMOS technology and contains approximately 29,000 transistors. The 80C86A is the low power version of the 8086 designed using HCMOS technology. The 8086 is packaged in a 40-pin CERDIP or plastic package and requires a single 5V power supply. The 8086 can be operated at three different clock speeds. The standard 8086 runs at 5 MHz internal clock frequency, whereas the 8086-1 and 8086-2 run at internal clock frequencies of 10 and 8 MHz, respectively. An external clock generator/driver chip such as the Intel 8284 is needed to generate the 8086 clock input signal. The 8284 divides the external crystal input internally by three. This means that for a 5-MHz 8086 internal clock, the 8284's X1 and X2 pins must be connected to a 15-MHz crystal. The 8284 will then generate a 5 MHz clock at its CLK pin which should be connected to the 8086 CLK input.

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 instruction bytes from memory and queues them in order to speed up instruction execution.

There are some advantages of working with the segmented memory. First of all, after initializing the 16-bit segment registers, the 8086 has to deal with only 16-bit effective addresses. That is, the 8086 has to manipulate and store 16-bit address components. Secondly, because of memory segmentation, the 8086 can be effectively used in time-shared systems. For example, in a time-shared system, several users share a microprocessor. The microprocessor works with one user's program for say, 10 milliseconds. After spending 10 milliseconds with each of the other users, the microprocessor returns to execute the first user's program. Each time the microprocessor switches from one user's program to the next, it must execute a new section of code and new sections of data. Segmentation makes it easy to switch from one user program (and data) to another.

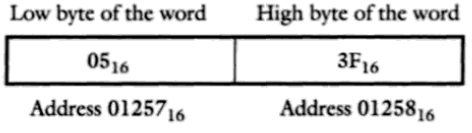
The memory of an 8086-based microcomputer is organized as bytes. Each byte can be uniquely addressed with 20-bit addresses of 00000_{16}, 00001_{16}, 00002_{16} ..., $FFFFF_{16}$. An 8086 16-

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:



The 16-bit word stored at the even address 00520_{16} is 2607_{16}.

Next consider a word stored at an odd address as follows:



The 16-bit word stored at the odd address 01257_{16} is $3F05_{16}$. 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 (for X) or 8-bit (for H or L). The instruction, MOV BX, [ADDR] moves the contents of a memory location addressed by ADDR into the 8086 16-bit register BX. Now, if ADDR along with the data segment register provides a 20-bit even address such as 30024_{16}, 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 30024_{16} and 30025_{16}, respectively, in a single access. Now, if ADDR is an odd address such as 40005_{16}, then the MOV BX, [ADDR] instruction loads BL and BH with the contents of memory locations 40005_{16} and 40006_{16}, respectively, in two accesses. Note that the 8086 accesses locations 40004_{16} and 40005_{16} in the first operation but discards the contents of 40004_{16}, and in the second operation accesses 40006_{16} and 40007_{16} but ignores the contents of 40007_{16}.

Next, consider a byte move such as MOV BH, [ADDR]. If ADDR is an even address such as 50002_{16}, then this MOV instruction accesses both 50002_{16} and 50003_{16}, but loads BH with the contents of 50002_{16} and ignores the contents of 50003_{16}. However, if ADDR is an odd address such as 50003_{16}, then this MOV loads BH with the contents of 50003_{16} and ignores the contents of 50002_{16}.

The 8086 family consists of two types of 16-bit microprocessors — the 8086 and 8088. 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 8088 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 8088 was used in designing the original IBM Personal computer.

An 8086 can be configured as a small uniprocessor system (minimum mode if the $\overline{\text{MN}}/\text{MX}$ 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/$\overline{MX}$ 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 performed 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 an 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 6 MHz, 8 MHz, and other frequencies. Like the 8085, 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 6, 8, 10, 12.5, 16.67, or 20 MHz clock frequency. The 80286 is typically used in a multiuser or multitasking system. The 80286 was used as the CPU of the IBM PC/AT Personal computer. Intel's 32-bit microprocessor family includes 80386, 80486 and Pentium microprocessors which will be covered later in this book.

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 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.

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 8086 contains the on-chip logical address to physical address mapping hardware. The programmer works with the logical address which includes the 16-bit contents of a segment register and a 16-bit displacement or offset value. The 8086 on-chip mapping hardware translates this logical address to 20-bit physical address which it then generates on its twenty addressing pins.

The BIU has four 16-bit segment registers. These are the Code Segment (CS), the Data Segment (DS), the Stack Segment (SS), and the Extra Segment (ES). 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

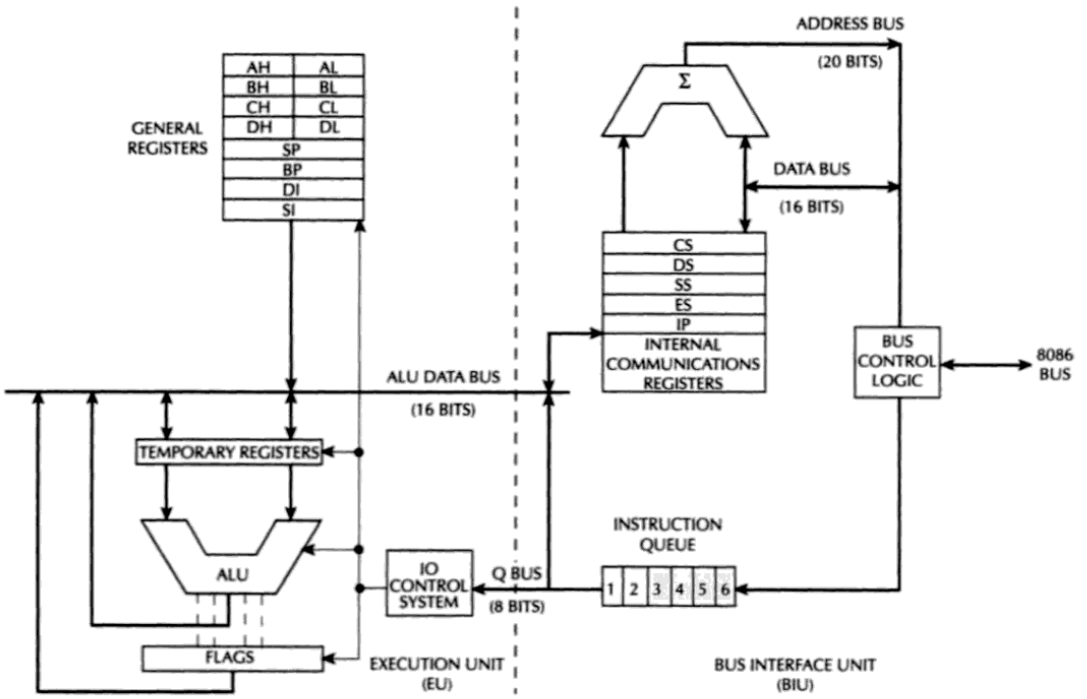


FIGURE 3.1 Internal architecture of the 8086.

desired segments. All program instructions must be located in main memory pointed to by 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 programmer-provided logical address (16-bit contents of CS and IP) by logically shifting the contents of CS four bits to left and then adding the 16-bit contents of IP. For example, if [CS] = $456A_{16}$ and [IP] = 1620_{16}, then the 20-bit physical address is generated by the BIU as follows:

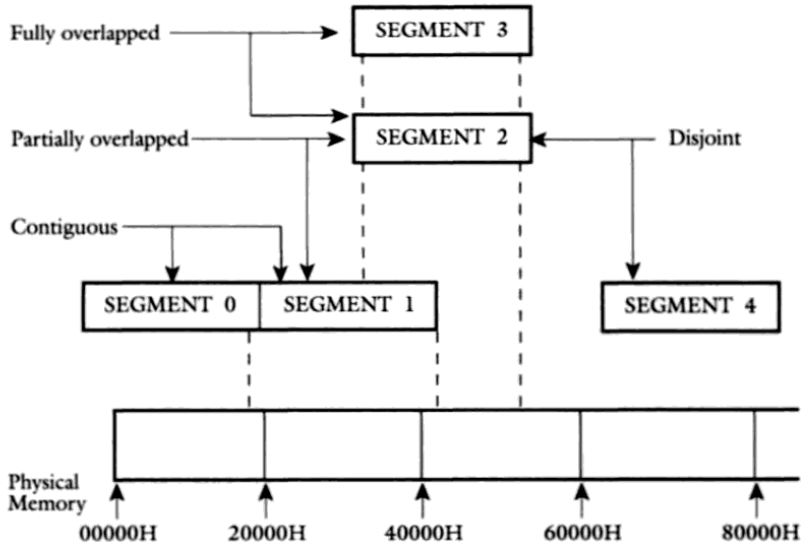
Four times logically shifted [CS] to left	$= 456A0_{16}$
+ [IP] as offset	$= 1620_{16}$
20-bit physical address	$= 46CC0_{16}$

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. A 16-bit offset (Effective Address, EA) along with the 16-bit contents of DS are used 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 use ES and DI to determine the 20-bit physical address for the destination, and DS and SI for the source address.

The segments can be contiguous, partially overlapped, fully overlapped, or disjoint. An example of how five segments (segment 0 through segment 4) may be stored in physical memory are 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 00000_{16}, 00010_{16}, 00020_{16}, 00030_{16} …, $FFFF0_{16}$. 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. One can then work with a 64K memory as with the 8085.

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 each 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.

The 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 16×16 multiplication or high 16-bit dividend (data) before a $32 \div 16$ division and the 16-bit remainder after the division.

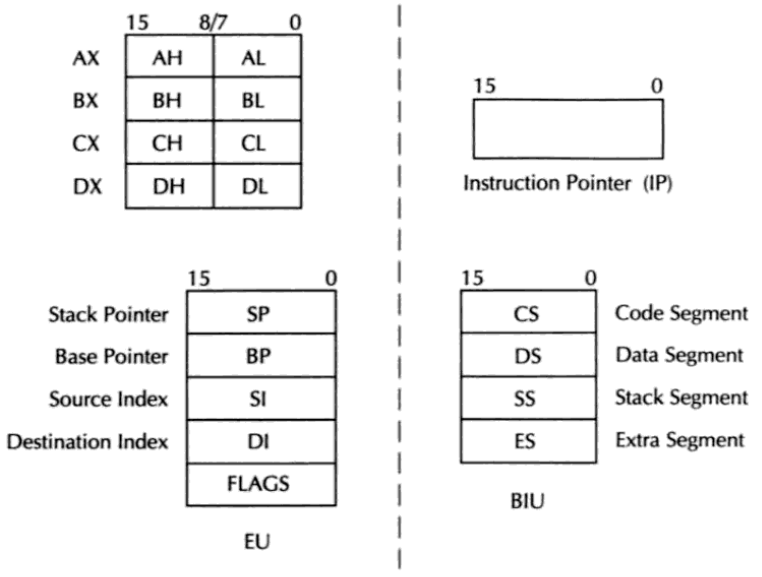


FIGURE 3.2 8086 registers.

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 based addressing mode.

The FLAG register in the EU holds the status flags after an ALU operation. Figure 3.2 shows the 8086 registers.

The 8086 has six one-bit flags. Figure 3.3 shows the flag register. The AF (Auxiliary carry Flag) is used by BCD arithmetic instructions. AF = 1 if there is a carry from the low nibble (4-bit) into the high nibble or a borrow from the high nibble into the low nibble of the low-order 8-bit of a 16-bit number. The CF (Carry Flag) is set if there is a carry from addition or borrow from subtraction. The OF (Overflow Flag) is set if there is an arithmetic overflow, that is, if the size of the result exceeds the capacity of the destination location. An interrupt on overflow instruction is available which will generate an interrupt in this situation. The SF (Sign Flag) is set if the most significant bit of the result is one (negative) and is cleared to zero for non-negative result. The PF (Parity Flag) is set if the result has even parity; PF is zero for odd parity of the result. The ZF (Zero Flag) is set if result is zero; ZF is zero for nonzero result.

The 8086 has three control bits in the flag register which can be set or reset by the programmer: Setting DF (Direction Flag) to one causes string instructions to autodecrement the appropriate index register(s), and clearing DF to zero causes string instructions to autoincrement. Setting IF (Interrupt Flag) to one causes the 8086 to recognize external maskable interrupts; clearing IF to zero disables these interrupts. Setting TF (Trace Flag) to one places the 8086 in the single-step mode. In this mode, the 8086 generates an internal interrupt

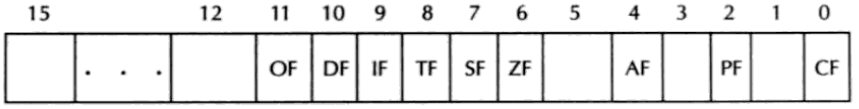


FIGURE 3.3 8086 flag register.

after execution of each instruction. The user can write a service routine at the interrupt address vector to display the desired registers and memory locations. The user can thus debug a program.

3.3 8086 Addressing Modes

The 8086 has 8 basic addressing modes (register, immediate, and memory). The basic and other 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. Implied addressing mode

The assembler directives for the Microsoft 8086 asembler written for IBM PC will be used to illustrate the above 8086 addressing modes.

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 DX, CX 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, CH 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, 03H moves the 8-bit data 03H 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 such 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 part of the program located in the code segment. That is, the memory must be addressed by the 8086 CS and IP registers. This is because these data are considered part of the instruction.

3.3.2 Addressing Modes for Accessing Data in Memory (Memory Modes)

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 which is DS in most cases. 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 some cases such as stack reference instructions (such as PUSH and POP). 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 directly included with the instruction. As an example, consider MOV CX, DS:START. This instruction moves the 16-bit contents of a 20-bit physical memory location computed from START and DS into 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 directive such as EQU is 0040_{16} and [DS] = 3050_{16}, then the BIU generates the 20-bit physical address 30540_{16} on the 8086 address pins and then initiates a memory read cycle to read the 16-bit data from memory location starting at 30540_{16} location. The memory logic places the 16-bit contents of locations 30540_{16} and 30541_{16} on the 8086 data pins. The BIU transfers these data to the EU; the EU then moves these data to CX; [30540_{16}] to CL and [30541_{16}] to CH.

3.3.2.b Register Indirect Addressing Mode

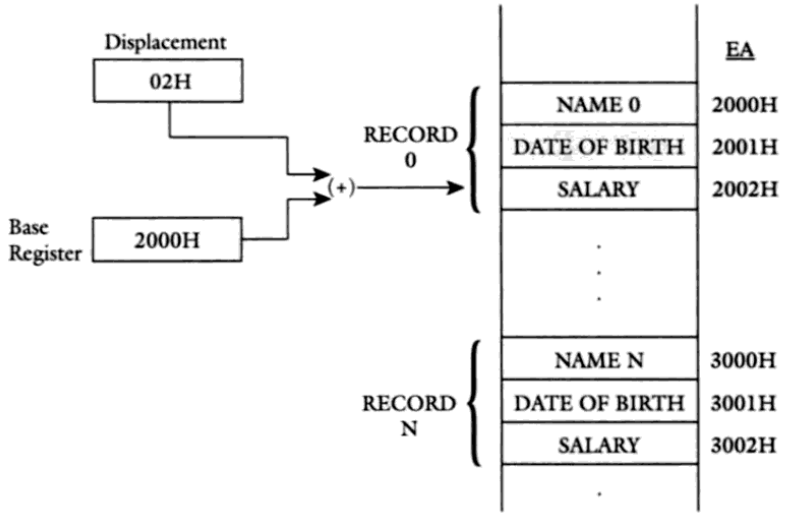
In this mode, the EA is specified in either a pointer register or an index register. The pointer register can be either base register BX or base pointer register BP and index register can be either the Source Index (SI) register or the Destination Index (DI) register. The 20-bit physical address is computed 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*16. Now, if [DS] = 5004_{16}, [DI] = 0020_{16}, and [BX] = 2456_{16}, then after MOV [DI], BX, contents of BL (56_{16}) and BH (24_{16}) are moved to memory locations 50060_{16} and 50061_{16}, respectively.

Using this mode, one instruction can operate on many different memory locations if the value in the base or index register is updated.

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 using BX, the 20-bit physical address is computed from BX and DS. On the other hand, when the user stack is accessed using BP, the 20-bit physical address is computed from BP and SS. This allows the programmer to access the stack without changing the SP contents. Note that SP is called the system stack pointer since some 8086 stack instructions such as PUSH and POP automatically use SP as the stack pointer while BP is called the user stack pointer. 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 based 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, a 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 a 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, ALPHA [BX], where ALPHA 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 DX              ;  Save
PUSH AX              ;  registers
SUB SP, 4            ;  Allocate 2 words of
⋮                    ;  stack for accessing stack
MOV [BP - 6], AX     ;  Arbitrary instructions for
MOV [BP - 8], BX     ;  accessing stack data using BP
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 [BP – 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

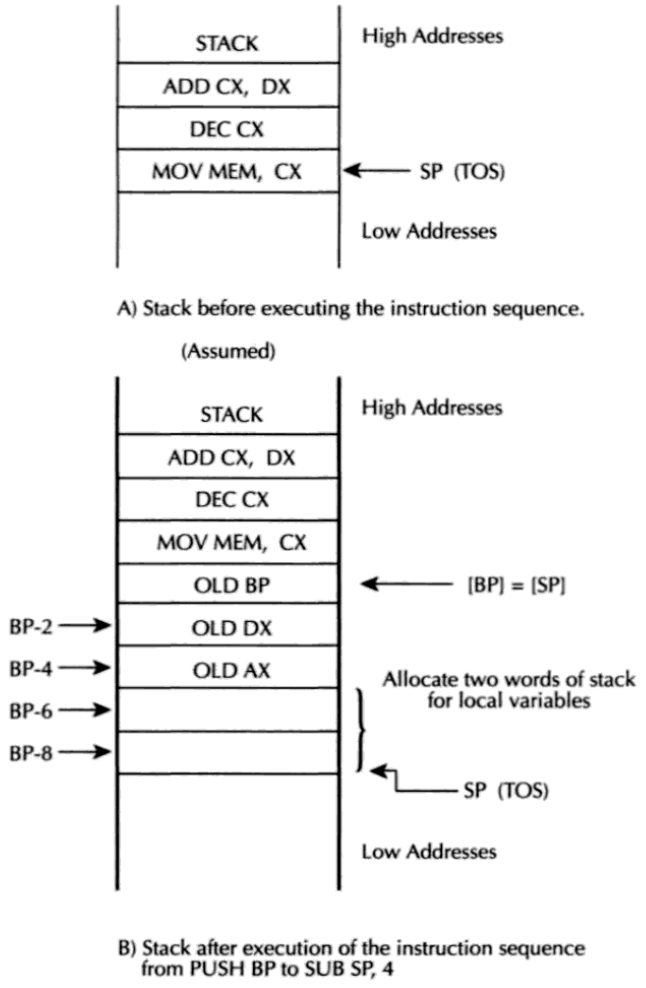


FIGURE 3.4 Accessing stack using BP.

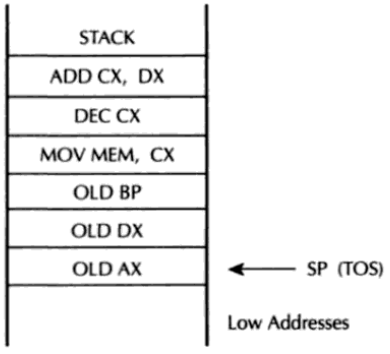
stack. The stack at this point is shown in Figure 3.4C. The last three POP instructions restore the contents of AX, DX, and BP to their original values and return the stack as it was before the instruction sequence was executed. This is shown in Figure 3.4D.

3.3.2.d Indexed Addressing Mode

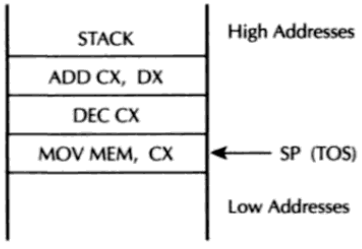
In this mode, the effective address is calculated by adding the unsigned 16-bit or sign-extended 8-bit displacement and the contents of SI or DI.

As an example, MOV BH, ARRAY[SI] moves the contents of the 20-bit address computed from the displacement ARRAY, SI and DS into BH. The 8-bit displacement is provided by the programmer using an assembler directive 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.

The 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.



C) Stack after execution of ADD SP, 4



D) Stack before execution of POP BP.

FIGURE 3.4 (continued).

3.3.2.e 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 [DS] = 3000 H, then 8-bit content of CL is moved to 20-bit physical address 31208_{16}.

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 shown in Figure 3.5.

In the following, [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 BX and SI can be varied, two-dimensional arrays such as matrices can also be accessed.

3.3.2.f String Addressing Mode

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

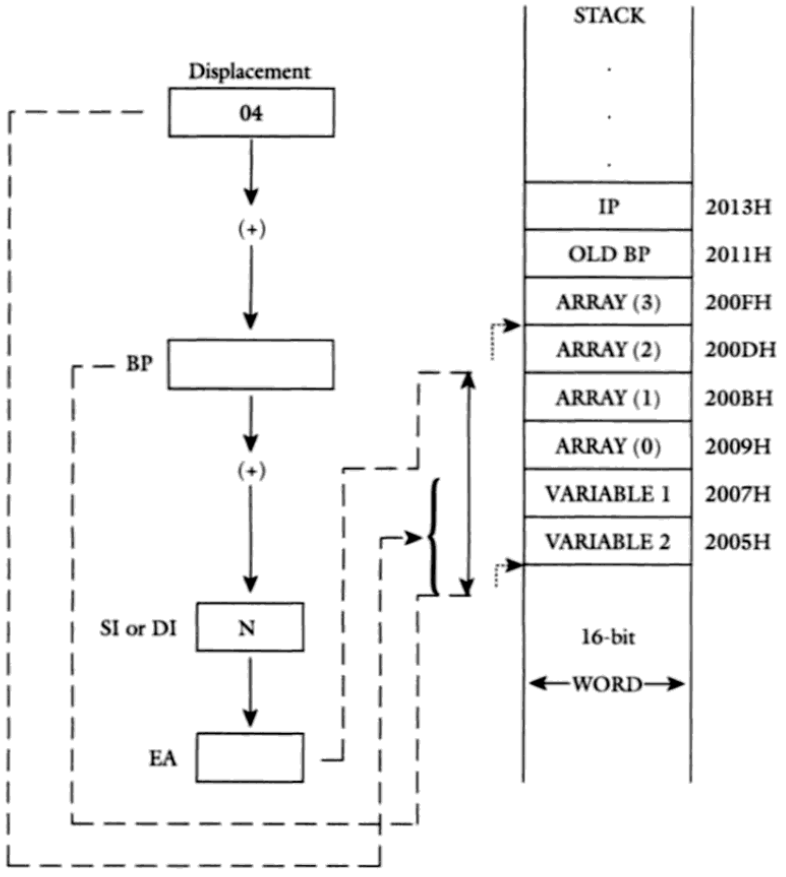


FIGURE 3.5 An example of Based Indexed Addressing.

The segment register for the destination must be ES and cannot be overridden. As an example, consider MOVS BYTE. If [DF] = 0, [DS] = 2000_{16}, [SI] = 0500_{16}, [ES] = 4000_{16}, [DI] = 0300_{16}, [$20500]_{16}$ = 38_{16}, and [40300_{16}] = 45_{16}, then after execution of the MOVS BYTE, [40300_{16}] = 38_{16}, [SI] = 0501_{16}, and [DI] = 0301_{16}. 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] = 5040_{16}, then IN AL, DX inputs the 8-bit content of port 5040_{16} into AL. On the other hand, IN AX, DX inputs the 8-bit contents of ports 5040_{16} and 5041_{16} into AL and AH, respectively. Note that 8-bit and 16-bit I/O transfers must take place via AL and AX, 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 (next instruction address) plus the 8-bit signed value of START; otherwise the next instruction is executed. Relative mode with signed 8-bit displacement provides a range of -128_{10} to $+127_{10}$ (0 being positive). If branching beyond this range is necessary, one must use the unconditional 8086 JUMP instruction which uses direct mode.

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 supplied in Appendix E provide a detailed description of the 8086 instructions including the instruction execution times.

Examples of some of the 8086 instructions are given in the following.

1. *Data transfer 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 and 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, and segment register SS. For example, if [CX] = 5009H, [BP] = 0030H, [SS] = 3000H, START = 06H, then after MOV START [BP], CX physical memory location [30036H] = 09H and [30037H] = 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. This instruction is equivalent to MOV AL, [AL][BX].

Table 3.1 Summary of 8086 Instructions

Instructions	Interpretation	Comments
AAA	ASCII adjust [AL] after addition	This instruction has implied addressing mode; this instruction is used to adjust the content of AL after addition of two ASCII characters
AAD	ASCII adjust for division	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 byte
AAM	ASCII adjust after multiplication	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
AAS	ASCII adjust [AL] after subtraction	This instruction has implied addressing mode used to adjust [AL] after subtraction of two ASCII characters
ADC mem/reg 1, mem/reg 2	[mem/reg 1] ← [mem/reg 1] + [mem/reg 2] + CY	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
ADC mem, data	[mem] ← [mem] + data + CY	Data can be 8- or 16-bit; mem uses DS as the segment register; all flags are affected
ADC reg, data	[reg] ← [reg] + data + CY	Data can be 8- or 16-bit; register cannot be segment register; all flags are affected
ADD mem/reg 2, mem/reg 1	[mem/reg 1] ← [mem/reg 2] + [mem/reg 1]	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
ADD mem, data	[mem] ← [mem] + data	Mem uses DS as the segment register; data can be 8- or 16-bit; all flags are affected
ADD reg, data	[reg] ← [reg] + data	Data can be 8- or 16-bit; no segment registers are allowed; all flags are affected
AND mem/reg 1, mem/reg 2	[mem/reg 1] ← [mem/reg 1] ∧ [mem/reg 2]	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
AND mem, data	[mem] ← [mem] ∧ data	Data can be 8- or 16-bit; mem uses DS as the segment register; all flags are affected with OF and CF always cleared to zero
AND reg, data	[reg] ← [reg] ∧ data	Data can be 8- or 16-bit; reg cannot be segment register; all flags are affected with OF and CF cleared to zero
CALL PROC (NEAR)	Call a subroutine in the same segment with signed 16-bit displacement (to CALL a subroutine in ±32K)	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; thus, a subroutine is called in the same segment (intrasegment direct)
CALL reg 16	CALL a subroutine in the same segment addressed by the contents of a 16-bit general register	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 (intrasegment indirect)

Table 3.1 Summary of 8086 Instructions (*continued*)

Instructions	Interpretation	Comments
CALL mem 16	CALL a subroutine addressed by the content of a memory location pointed to by 8086 16-bit register such as BX, SI, and DI	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 such as BX, SI, and DI into IP; [CS] is unchanged (intrasegment indirect)
CALL PROC (FAR)	CALL a subroutine in another segment	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)
CALL DWORDPTR [reg 16]	CALL a subroutine in another segment	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] in DS; 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] and [reg 16 + 1] in DS; typical 8086 registers used for reg 16 are BX, SI, and DI (intersegment indirect)
CBW	Convert a byte to a word	Extend the sign bit (bit 7) of AL register into AH
CLC	CF ← 0	Clear carry to zero
CLD	DF ← 0	Clear direction flag to zero
CLI	IF ← 0	Clear interrupt enable flag to zero to disable maskable interrupts
CMC	CF ← CF'	One's complement carry
CMP mem/reg 1, mem/reg 2	[mem/reg 1] – [mem/reg 2], flags are affected	mem/reg can be 8- or 16-bit; no memory-to-memory comparison allowed; result of subtraction is not provided; all flags are affected
CMP mem/reg, data	[mem/reg] – data, flags are affected	Subtracts 8- or 16-bit data from [mem or reg] and affects flags; no result is provided
CMPS BYTE or CMPSB	FOR BYTE [[SI]] – [[DI]], flags are affected [SI] ← [SI] ± 1 [DI] ← [DI] ± 1	8- or 16-bit data addressed by [DI] in ES is subtracted from 8- or 16-bit data addressed by SI in DS and flags are affected without providing any result; if DF = 0, then SI and DI are incremented by one for byte and two for word; if DF = 1, then SI and DI are decremented by one for byte and two for word; the segment register ES in destination cannot be overridden
CMPS WORD or CPSW	FOR WORD [[SI]] – [[DI]], flags are affected [SI] ← [SI] + 2 [DI] ← [DI] + 2	
CWD	Convert a word to 32 bits	Extend the sign bit of AX (bit 15) into DX
DAA	Decimal adjust [AL] after addition	This instruction uses implied addressing mode; this instruction converts [AL] into BCD; DAA should be used after BCD addition
DAS	Decimal adjust [AL] after subtraction	This instruction uses implied addressing mode; converts [AL] into BCD; DAS should be used after BCD subtraction
DEC reg 16	[reg 16] ← [reg 16] – 1	This is a one-byte instruction; used to decrement a 16-bit register except segment register; does not affect the carry flag
DEC mem/reg 8	[mem] ← [mem] – 1 or [reg 8] ← [reg 8] – 1	Used to decrement a byte or a word in memory or an 8-bit register content; segment register cannot be decremented by this instruction; does not affect carry flag

Table 3.1 Summary of 8086 Instructions (*continued*)

Instructions	Interpretation	Comments
DIV mem/reg	16/8 bit divide: [AX] [mem 8 / reg 8] [AH] ← Remainder [AL] ← Quotient 32/16 bit divide: [DX] [AX] [mem 16 / reg 16] [DX] ← Remainder [AX] ← Quotient	Mem/reg is 8-bit for 16-bit by 8-bit divide and 16-bit for 32-bit by 16-bit divide; this is an unsigned division; no flags are affected; division by zero automatically generates an internal interrupt
ESC external OP code, source	ESCAPE to external processes	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; when the 8086 encounters an ESC instruction, it usually treats it as NOP; the coprocessor decodes this instruction and carries out the operation using the 6-bit OP code independent of the 8086; for ESC OP code, memory, the 8086 accesses data in memory for the coprocessor; for ESC data, register, the coprocessor operates on 8086 registers; the 8086 treats this as an NOP
HLT	HALT	Halt
IDIV mem/reg	Same as DIV mem/reg	Signed division. No flags are affected.
IMUL mem/reg	For 8 × 8 [AX] ← [AL] * [mem 8/reg 8] FOR 16 × 16 [DX] [AX] ← [AX]* [mem 16/reg 16]	Mem/reg can be 8- or 16-bit; only CF and OF are affected; signed multiplication
IN AL, DX	[AL] ← [PORT DX]	Input AL with the 8-bit content of a port addressed by DX; this is a one-byte instruction
IN AX, DX	[AX] ← [PORT DX]	Input AX with the 16-bit content of a port addressed by DX and DX + 1; this is a one-byte instruction
IN AL, PORT	[AL] ← [PORT]	Input AL with the 8-bit content of a port addressed by the second byte of the instruction
IN AX, PORT	[AX] ← [PORT]	Input AX with the 16-bit content of a port addressed by the 8-bit address in the second byte of the instruction
INC reg 16	[reg 16] ← [reg 16] + 1	This is a one-byte instruction; used to increment a 16-bit register except the segment register; does not affect the carry flag
INC mem/reg 8	[mem] ← [mem] + 1 or [reg 8] ← [reg 8] + 1	This is a two-byte instruction; 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
INT n (n can be zero thru 255)	[SP] ← [SP] – 2 [[SP]] ← Flags IF ← 0 TF ← 0 [SP] ← [SP] – 2 [[SP]] ← [CS] [CS] ← 4n + 2 [SP] ← [SP] –2	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 or program; software interrupt instructions can also be used for checking interrupt service routines written for hardware-initiated interrupts

Table 3.1 Summary of 8086 Instructions (*continued*)

Instructions	Interpretation	Comments
	[[SP]] ← [IP] [IP] ← 4n	
INTO	Interrupt on Overflow	Generates an internal interrupt if OF = 1; executes INT 4; can be used after an arithmetic operation to activate a service routine if OF = 1; when INTO is executed and if OF = 1, operations similar to INT n take place
IRET	Interrupt Return	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
JA/JNBE disp 8	Jump if above/jump if not below or equal	Jump if above/jump if not below or equal with 8-bit signed displacement; that is, the displacement can be from -128_{10} to $+127_{10}$, zero being positive; JA and JNBE are the mnemonic which represent the same instruction; Jump if both CF and ZF are zero; used for unsigned comparison
JAE/JNB/JNC disp 8	Jump if above or equal/jump if not below/jump if no carry	Same as JA/JNBE except that the 8086 Jumps if CF = 0; used for unsigned comparison
JB/JC/JNAE disp 8	Jump if below/jump if carry/jump if not above or equal	Same as JA/JNBE except that the jump is taken CF = 1, used for unsigned comparison
JBE/JNA disp 8	Jump if below or equal/jump if not above	Same as JA/JNBE except that the jump is taken if CF = 1 or ZF = 0; used for unsigned comparison
JCXZ disp 8	Jump if CX = 0	Jump if CX = 0; this instruction is useful at the beginning of a loop to bypass the loop if CX = 0
JE/JZ disp 8	Jump if equal/jump if zero	Same as JA/JNBE except that the jump is taken if ZF = 1; used for both signed and unsigned comparison
JG/JNLE disp 8	Jump if greater/jump if not less or equal	Same as JA/JNBE except that the jump is taken if ((SF ⊕ OF) or ZF) = 0; used for signed comparison
JGE/JNL disp 8	Jump if greater or equal/ jump if not less	Same as JA/JNBE except that the jump is taken if (SF ⊕ OF) = 0; used for signed comparison
JL/JNGE disp 8	Jump if less/Jump if not greater nor equal	Same as JA/JNBE except that the jump is taken if (SF ⊕ OF) = 1; used for signed comparison
JLE/JNG disp 8	Jump if less or equal/ jump if not greater	Same as JA/JNBE except that the jump is taken if ((SF ⊕ OF) or ZF) = 1; used for signed comparison
JMP Label	Unconditional Jump with a signed 8-bit (SHORT) or signed 16-bit (NEAR) displacement in the same segment	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 (–128 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 and then adds it to IP; [CS] is unchanged; on the other hand, if the assembler finds the displacement to be signed 16-bit (±32 K), then the assembler uses three bytes for the instruction: one byte for the OP code followed by 2 bytes for the displacement; the assembler adds the signed 16-bit displacement to IP; [CS] is unchanged; therefore, this JMP provides a jump in the same segment (intrasegment direct jump)
JMP reg 16	[IP] ← [reg 16] [CS] is unchanged	Jump to an address specified by the contents of a 16-bit register such as BX, SI, and DI in the same code segment; in the example JMP BX, [BX] is loaded into IP and [CS] is unchanged (intrasegment memory indirect jump)
JMP mem 16	[IP] ← [mem] [CS] is unchanged	Jump to an address specified by the contents of a 16-bit memory location addressed by 16-bit register

Table 3.1 Summary of 8086 Instructions (*continued*)

Instructions	Interpretation	Comments
		such as BX, SI, and DI; in the example, JMP [BX] copies the content of a memory location addressed by BX in DS into IP; CS is unchanged (intrasegment memory indirect jump)
JMP Label (FAR)	Unconditionally jump to another segment	This is a 5-byte instruction: the first byte is the OP code followed by four bytes of 32-bit immediate data; bytes 2 and 3 are loaded into IP; bytes 4 and 5 are loaded into CS to JUMP unconditionally to another segment (intersegment direct)
JMP DWORDPTR [reg 16]	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)
JNE/JNZ disp 8	Jump if not equal/jump if not zero	Same as JA/JNBE except that the jump is taken if ZF = 0; used for both signed and unsigned comparison
JNO disp 8	Jump if not overflow	Same as JA/JNBE except that the jump is taken if OF = 0
JNP/JPO disp 8	Jump if no parity/jump if parity odd	Same as JA/JNBE except that the jump is taken if PF = 0
JNS disp 8	Jump if not sign	Same as JA/JNBE except that the jump is taken if SF = 0
JO disp 8	Jump if overflow	Same as JA/JNBE except that the jump is taken if OF = 1
JP/JPE disp 8	Jump if parity/jump if parity even	Same as JA/JNBE except that the jump is taken if PF = 1
JS disp 8	Jump if sign	Same as JA/JNBE except that the jump is taken if SF = 1
LAHF	[AH] ← Flag low-byte	This instruction has implied addressing mode; it loads AH with the low byte of the flag register; no flags are affected
LDS reg, mem	[reg] ← [mem] [DS] ← [mem + 2]	Load a 16-bit register (AX, BX, CX, DX, SP, 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
LEA reg, mem	[reg] ← [offset portion of 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
LES reg, mem	[reg] ← [mem] [ES] ← [mem + 2]	DS is used as the segment register for mem; in the example LES 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 instruction is executed; this signal is used in multiprocessing; the LOCK pin of the 8086 can be used to LOCK other processors off the system bus during execution of an instruction; in this way, the 8086 can be assured of uninterrupted access to common system resources such as shared RAM

Table 3.1 Summary of 8086 Instructions (*continued*)

Instructions	Interpretation	Comments
LODS BYTE or LODSB	FOR BYTE [AL] ← [[SI]] [SI] ← [SI] ± 1	Load 8-bit data into AL or 16-bit data into AX from a memory location addressed by SI in segment DS; if DF = 0, then SI is incremented by 1 for byte or
LODS WORD or LODSW	FOR WORD [AX] ← [[SI]] [SI] ← [SI] ± 2	incremented by 2 for word after the load; if DF = 1, then SI is decremented by 1 for byte or decremented by 2 for word; LODS affects no flags
LOOP disp 8	Loop if CX not equal to zero	Decrement CX by one, without affecting flags and loop with signed 8-bit displacement (from –128 to +127, zero being positive) if CX is not equal to zero
LOOPE/LOOPZ disp 8	Loop while equal/loop while zero	Decrement CX by one without affecting flags and loop with signed 8-bit displacement if CX is equal to zero, and if ZF = 1 which results from execution of the previous instruction
LOOPNE/LOOPNZ disp 8	Loop while not equal/loop while not zero	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 previous instruction
MOV mem/reg 2, mem/reg 1	[mem/reg 2] ← [mem/reg 1]	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; no flags are affected
MOV mem, data	[mem] ← data	mem uses DS as the segment register; 8- or 16-bit data specifies whether memory location is 8- or 16-bit; no flags are affected
MOV reg, data	[reg] ← data	Segment register cannot be specified as reg; data can be 8- or 16-bit; no flags are affected
MOV segreg, mem/reg	[segreg] ← [mem/reg]	mem uses DS as segment register; used for initializing CS, DS, ES, and SS; no flags are affected
MOV mem/reg, segreg	[mem/reg] ← [segreg]	mem uses DS as segment register; no flags are affected
MOVS BYTE or MOVSB	FOR BYTE [[DI]] ← [[SI]] [SI] ← [SI] ± 1	Move 8-bit or 16-bit data from the memory location addressed by SI in segment DS location addressed by DI in ES; segment DS can be overridden by a
MOVS WORD or MOVSW	FOR WORD [[DI]] ← [[SI]] [SI] ← [SI] ± 2	prefix but destination segment must be ES and cannot be overridden; if DF = 0, then SI is incremented by one for byte or incremented by two for word; if DF = 1, then SI is decremented by one for byte or by two for word
MUL mem/reg	FOR 8 × 8 [AX] ← [AL]* [mem/reg] FOR 16 × 16 [DX] [AX] ← [AX]* [mem/reg]	mem/reg can be 8- or 16-bit; only CF and OF are affected; unsigned multiplication
NEG mem/reg	[mem/reg] ← [mem/reg]′ + 1	mem/reg 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
NOP	No Operation	8086 does nothing
NOT reg	[reg] ← [reg]′	mem and reg can be 8- or 16-bit; segment registers are not allowed; no flags are affected; ones complement reg
NOT mem	[mem] ← [mem]′	mem uses DS as the segment register; no flags are affected; ones complement mem
OR Mem/reg 1, Mem/reg 2	[mem/reg 1] ← [mem/reg 1] ∨ [mem/reg 2]	No memory-to-memory operation is allowed; [mem] or [reg 1] or [reg 2] can be 8- or 16-bit; all flags are affected with OF and CF cleared to zero; no

Table 3.1 Summary of 8086 Instructions (*continued*)

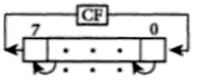
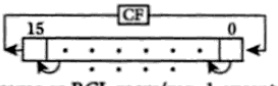
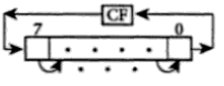
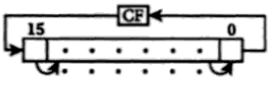
Instructions	Interpretation	Comments
		segment registers are allowed; mem uses DS as segment register
OR mem, data	[mem] ← [mem] ∨data	mem and data can be 8- or 16-bit; mem uses DS as segment register; all flags are affected with CF and OF cleared to zero
OR reg, data	[reg] ← [reg] ∨ data	reg and data can be 8- or 16-bit; no segment registers are allowed; all flags are affected with CF and OF cleared to zero
OUT DX, AL	[PORT] ← [AL] DX	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
OUT DX, AX	[PORT] ← [AX] DX	Output the 16-bit contents of AX into an I/O Port addressed by the 16-bit content of DX; this is a one-byte instruction
OUT PORT, AL	[PORT] ← [AL]	Output the 8-bit contents of AL into the Port specified in the second byte of the instruction
OUT PORT, AX	[PORT] ← [AX]	Output the 16-bit contents of AX into the Port specified in the second byte of the instruction
POP mem	[mem] ← [[SP]] [SP] ← [SP] + 2	mem uses DS as the segment register; no flags are affected
POP reg	[reg] ← [[SP]] [SP] ← [SP] + 2	Cannot be used to POP segment registers or flag register
POP segreg	[segreg] ← [[SP]] [SP] ← [SP] + 2	POP CS is illegal
POPF	[Flags] ← [[SP]] [SP] ← [SP] + 2	This instruction pops the top two stack bytes in the 16-bit flag register
PUSH mem	[SP] ← [SP] – 2 [[SP]] ← [mem]	mem uses DS as segment register; no flags are affected; pushes 16-bit memory contents
PUSH reg	[SP] ← [SP] – 2 [[SP]] ← [reg]	reg must be a 16-bit register; cannot be used to PUSH segment register or Flag register
PUSH segreg	[SP] ← [SP] – 2 [[SP]] ← [segreg]	PUSH CS is illegal
PUSHF	[SP] ← [SP] – 2 [[SP]] ← [Flags]	This instruction pushes the 16-bit Flag register onto the stack
RCL mem/reg, 1	ROTATE through carry left once byte or word in mem/reg	FOR BYTE  FOR WORD 
RCL mem/reg, CL	ROTATE through carry left byte or word in mem/reg by [CL]	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
RCR mem/reg, 1	ROTATE through carry right once byte or word in mem/reg	FOR BYTE  FOR WORD 
RCR mem/reg, CL	ROTATE through carry right byte or word in mem/reg by [CL]	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

Table 3.1 Summary of 8086 Instructions (*continued*)

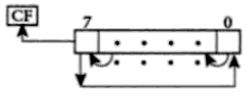
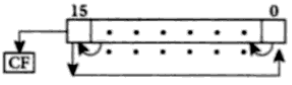
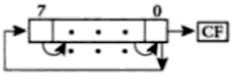
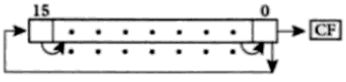
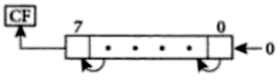
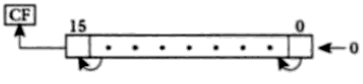
Instructions	Interpretation	Comments
RET	.POPS IP for intrasegment CALLS .POPS IP and CS for intersegment CALLS	The assembler generates an intrasegment return if the programmer has defined the subroutine as NEAR; for intrasegment return, the following operations take place: [IP] ← [[SP]], [SP] ← [SP] + 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] ← [[SP]], [SP] ← [SP] + 2, [CS] ← [[SP]], [SP] ← [SP] + 2; an optional 16-bit displacement 'START' can be specified with the intersegment return such as RET START; in this case, the 16-bit displacement is added to the SP value; this feature may be used to discard parameter pushed onto the stack before the execution of the CALL instruction
ROL mem/reg, 1	ROTATE left once byte or word in mem/reg	FOR BYTE  FOR WORD 
ROL mem/reg, CL	ROTATE left byte or word by the content of CL	[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 once; mem uses DS as the segment register
ROR mem/reg, 1	ROTATE right once byte or word in mem/reg	FOR BYTE  FOR WORD 
ROR mem/reg, CL	ROTATE right byte or word in mem/reg by [CL]	Operation same as ROR mem/reg, 1; [CL] specifics the number of rotates for up to 255; zero and negative rotates are illegal; mem uses DS as the segment register
SAHF	[Flags, low-byte] ← [AH]	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
SAL mem/reg, 1	Shift arithmetic left once byte or word in mem or reg	FOR BYTE  FOR WORD  Mem uses DS as the segment register; reg cannot be segment registers; OF and CF are affected; if sign bit is changed during or after shifting, the OF is set to one

Table 3.1 Summary of 8086 Instructions (*continued*)

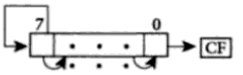
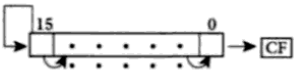
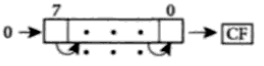
Instructions	Interpretation	Comments
SAL mem/reg, CL	Shift arithmetic left byte or word by shift count on CL	Operation same as SAL mem/reg, 1; CL contains shift count for up to 255; zero and negative shifts are illegal; [CL] is used as shift count when shift is greater than one; OF and SF are affected; if sign bit of [mem] is changed during or after shifting, the OF is set to one; mem uses DS as segment register
SAR mem/reg, 1	SHIFT arithmetic right once byte or word in mem/reg	FOR BYTE  FOR WORD 
SAR mem/reg, CL	SHIFT arithmetic right byte or word in mem/reg by [CL]	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
SBB mem/reg 1, mem/reg 2	[mem/reg 1] ← [mem/reg 1] – [mem/reg 2] – CY	Same as SUB mem/reg 1, mem/reg 2 except this is a subtraction with borrow
SBB mem, data	[mem] ← [mem] – data – CY	Same as SUB mem, data except this is a subtraction with borrow
SBB reg, data	[reg] ← [reg] – data – CY	Same as SUB reg, data except this is a subtraction with borrow
SBB A, data	[A] ← [A] – data – CY	Same as SUB A, data except this is a subtraction with borrow
SCAS BYTE or SCASB	FOR BYTE [AL] – [[DI]], flags are affected, [DI] [DI] ± 1	8- or 16-bit data addressed by [DI] in ES is subtracted from 8- or 16-bit data in AL or AX and flags are affected without affecting [AL] or [AX] or string data; ES cannot be overridden; if DF = 0, then DI is incremented by one for byte and two for word; if DF = 1, then DI is decremented by one for byte or decremented by two for word
SCAS WORD or SCASW	FOR WORD [AX] – [[DI]], flags are affected, [DI] ← [DI] ± 2	
SHL mem/reg, 1	SHIFT logical left once byte or word in mem/reg	Same as SAL mem/reg, 1
SHL mem/reg, CL	SHIFT logical left byte or word in mem/reg by the shift count in CL	Same as SAL mem/reg, CL except overflow is cleared to zero
SHR mem/reg, 1	SHIFT right logical once byte or word in mem/reg	FOR BYTE  FOR WORD 
SHR mem/reg, CL	SHIFT right logical byte or word in mem/reg by [CL]	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
STC	CF ← 1	Set carry to one
STD	DF ← 1	Set direction flag to one
STI	IF ← 1	Set interrupt enable flag to one to enable maskable interrupts
STOS BYTE or STOSB	FOR BYTE [[DI]] ← [AL] [DI] ← [DI] ± 1	Store 8-bit data from AL or 16-bit data from AX into a memory location addressed by DI in segment ES; segment register ES cannot be overridden; if DF = 0, then DI is incremented by one for byte or incremented by two for word after the store
STOS WORD or STOSW	FOR WORD [[DI]] ← [AX] [DI] ← [DI] ± 2	

Table 3.1 Summary of 8086 Instructions (*continued*)

Instructions	Interpretation	Comments
SUB mem/reg 1, mem/reg 2	[mem/reg 1] ← [mem/reg 1] – [mem/reg 2]	No memory-to-memory SUB permitted; all flags are affected; mem uses DS as the segment register
SUB mem, data	[mem] ← [mem] – data	Data can be 8- or 16-bit; mem uses DS as the segment register; all flags are affected
SUB reg, data	[reg] ← [reg] – data	Data can be 8- or 16-bit; all flags are affected
TEST mem/reg 1, mem/reg 2	[mem/reg 1] ∧ [mem/reg 2], no result; flags are affected	No memory-to-memory TEST is allowed; no result is provided; all flags are affected with CF and OF cleared to zero; [mem], [reg 1] or [reg 2] can be 8- or 16-bit; no segment registers are allowed; mem uses DS as the segment register
TEST mem, data	[mem] ∧ data, no result; flags are affected	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
TEST reg, data	[reg] ∧ data, no result; flags are affected	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;
WAIT	8086 enters wait state	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; this feature can be used to synchronize the operation of 8086 to an event in external hardware
XCHG mem, reg	[mem] ↔ [reg]	reg and mem can be both 8- or 16-bit; mem uses DS as the sement register; reg cannot be segment register; no flags are affected
XCHG reg, reg	[reg] ↔ [reg]	reg can be 8- or 16-bit; reg cannot be segment register; no flags are affected
XLAT	[AL] ← [AL] + [BX]	This instruction is useful 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 AL; 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 AL 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, [AL] [BX]
XOR mem/reg 1, mem/reg 2	[mem/reg 1] ← [mem/reg 1] ⊕ [mem/reg 2]	No memory-to-memory operation is allowed; [mem] or [reg 1] or [reg 2] 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 mem, 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 reg, data	[reg] ← [reg] ⊕ data	Same as XOR mem, data

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 (34H) 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 in the data segment of memory; with code for 0 stored at 3030H, code for 1 at 3031H, 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 the data segment 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.

2. *Examples of input/output 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 contained in the DX register. Consider ports addressed by 16-bit address contained in DX. Assume [DX] = 3124_{16} in all the following examples:

  IN AL, DX inputs 8-bit data from 8-bit port 3124_{16} into AL.

  IN AX, DX inputs 16-bit data from ports 3124_{16} and 3125_{16} into AL and AH respectively.

  OUT DX, AL outputs 8-bit data from AL into port 3124_{16} .

  OUT DX, AX outputs 16-bit data from AL and AH into ports 3124_{16} and 3125_{16} respectively.

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 for which data output is desired to register DX in the subroutine.

3. *Examples of address initialization instructions*

- LEA reg, mem loads an offset (mem) directly into the specified register. This instruction is useful when address computation is required. LEA BX, 5000H and MOV BX, 5000H accomplish the same task. That is, both of these instructions load 5000H into BX. On the other hand, LEA DI, [SI] [BX] loads the 16-bit value computed from BX and SI into DI whereas MOV DI, [SI] [BX] loads the 16 bit contents of a memory location computed from SI and BX into DI. 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 instructions.
- 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 the data segment into SI and the 16-bit contents of memory offset by [BX + 2] in the data segment 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 DI, [BX] loads the 16-bit contents of

memory offset by [BX] in the data segment to DI and then initializes ES with the 16-bit contents of memory offset by [BX + 2] in DS.

4. *Examples of flag register instructions*

- PUSHF pushes the 16-bit flag register onto the stack.
- LAHF loads AH with the condition codes from the low byte of the flag register.

5. *Explanation of arithmetic 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 (32_{16}) and 5 (35_{16}) 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] = 32H = ACSII for 2
             ;  [DL] = 35H = ASCII for 5
             ;  Result [CL] = 67H
MOV AL, CL   ;  Move ASCII result
             ;  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_{16} 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. If after the addition, the low 4-bit of the result in AL is greater than 9 (or if AF = 1), then the DAA adds 6 to the low 4 bits of AL. Then, 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] = 6DH
DAA          ;  Since low nibble
             ;  D = 1101₂ > 9, add i.e. 1101₂ + 0110₂ →
                                        1      0011₂
                                        ↑      ↑
                                      carry  3BCD
```

- The ASCII codes for two 8-bit numbers in an 8086-based microcomputer can be subtracted. Assume

```
[AL] = 35H = ASCII for 5
[DL] = 37H = ASCII for 7
```

The following instruction sequence provides the correct subtraction result:

```
SUB AL, DL   ;  [AL] = 1111 1110₂ = subtraction result
             ;  in 2's complement
             ;  CF = 1
AAS          ;  [AL] = BCD02
             ;  CF = 1 means
             ;  borrow to be
             ;  used in multibyte BCD subtraction
```

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

- 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 in AL is greater than 9 (or if AF = 1), then DAS subtracts 6 from the low 4-bit of AL. Then, 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 → Low nibble = 1111
DAS          ;  CF = 0                -6 = 1010
                                       1   1001
             ;  [AL] = 39 BCD           ↑
                                     ignore
```

- IMUL mem/reg provides signed 8 × 8 or signed 16 × 16 multiplication. As an example, if [CL] = FDH = -3_{10}, [AL] = FEH = -2_{10}, then after IMUL CL, register AX contains 0006H.
- Consider 16 × 16 unsigned multiplication, MUL WORDPTR [BX]. If [BX] = 0050H, [DS] = 3000H, [30050H] = 0002H, [AX] = 0006H, then after MUL WORDPTR [BX], [DX] = 0000H, [AX] = 000CH.
- Consider DIV BL. If [AX] = 0009H, [BL] = 02H, then after DIV BL,

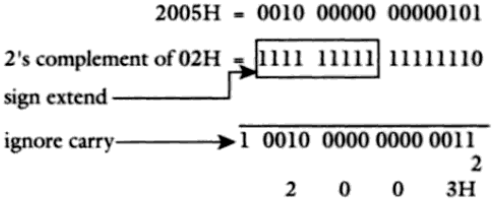
```
[AH] = Remainder = 01H
[AL] = Quotient = 04H
```

- Consider IDIV WORDPTR [BX]. If [BX] = 0020H, [DS] = 2000H, [20020H] = 0004H, [DX] [AX] = 00000011H, then after IDIV. WORDPTR [BX],

```
[DX] = Remainder = 0001H
[AX] = 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 (58 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 DH, [AL] = 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:



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 16×16 signed multiplication.

- AAM adjusts the product of two unpacked BCD digits in AX. If [AL] = BCD3 = 00000011_2 and [CH] = BCD8 = $0000\ 1000_2$, then after MUL CH, [AX] = 0000000000011000_2 = 0018H, and after using AAM, [AX] = $00000010\ 00000100_2$ = unpacked BCD 24. The following instruction sequence accomplishes this:

```
MUL CH
AAM
```

Note that the 8086 does not allow multiplication of two ASCII codes. Therefore, before multiplying two ASCII bytes received from a terminal, one must mask the upper 4-bits of each one of these bytes and then multiply them as two unpacked BCD digits and then use AAM for adjustment. In order to convert the unpacked BCD product back to ASCII, for sending back to the terminal, one must OR the product with 3030H.

6. *Examples of logical, shift and rotate instructions*

- TEST BL, 3 logically ANDs the contents of BL with 00000011_2 but does not store the result in BL. All flags are affected.
- All shift and rotate instructions include two operands. The destination operand specifies the register or memory to be shifted or rotated while the source operand specifies the number of times the register or memory contents are to be shifted. For example, SHL DX, 1 logically shifts the 16-bit contents of DX once to the left. On the other hand, SHL DX,CL, with CL = 5, logically shifts the 16-bit contents of DX five times to the left. For all 8086 shift and rotate instructions, a shift count of one must be represented in the source operand by '1' while shift counts from 2_{10} to 255_{10} must be represented in the source operand by the contents of CL.

7. *Example of string instructions*

- LODS can be represented in four forms:

```
For Byte   LODS BYTE
             or
           LODSB
```

```
For Word   LODS WORD
             or
           LODSW
```

If [SI] = 0020H, [DS] = 3000H, [30020H] = 05H, DF = 0, then after LODS BYTE or LODSB, [AL] = 05H, [SI] = 0021H.

- If [DS] = 2000H, [ES] = 3000H, [SI] = 0020H, [DI] = 0050H, DF = 0, [20020H] = 0205H, [30050H] 4071H, then after execution of MOVSW or MOVS WORD, memory location 30050H will contain 0205H. Since DF = 0, register SI will contain 0022H and register DI will contain 0052H.
- REP, a one-byte prefix, can be used with MOVS to cause the instruction MOVS to continue executing until CX = 0. Each time the instructions such as REP MOVSB or REP MOVSW are executed, CX is automatically decremented by 1, and if [CX] ≠ 0, MOVS is reexecuted and then CX is decremented by 1 until CX = 0; the next 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.
- REPE/REPZ or REPNE/REPNZ prefix can be used with CMPS or SCAS to cause one of these instructions to continue executing until ZF = 0 (for REPNE/REPNZ prefix) or CX = 0. Note that REPE and REPZ are two mnemonics for the same prefix byte. Similarly, REPNE and REPNZ also provide the same purpose.
- If CMPS is prefixed with REPE or REPZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings are equal (ZF = 1)". 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 REPE or REPZ, the operation is interpreted as "scan while not end-of-string (CX not 0) 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 0) 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 instructions. Upon return from interrupt, program execution resumes at this point but any additional prefixes specified are not recognized. If multiple prefix 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.

8. *Examples of unconditional transfers*

- There are two types of Jump instructions. These are intersegment Jumps (both IP and CS change; Jump to a different code segment) and intrasegment Jumps (IP changes and CS is fixed; Jump in the same code segment). For example, JMP FAR BEGIN (or some 8086 assemblers use JMP FAR PTR BEGIN) unconditionally branches to a label BEGIN in a different code segment. JMP START jumpts to a label START in the same code segment.
- CALL instructions can be intersegment and intrasegment. For example, CALL DWORD PTR [BX] pushes CS and IP onto the stack and loads IP and CS with the contents of four consecutive locations pointed to by BX. CALL BX, on the other hand, pushes IP onto the stack, the new value of IP is loaded from BX and CS in unchanged.

9. *Examples of conditional transfers*

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 on whether the numbers to be compared are signed or unsigned. The 8-bit number $1111\ 1110_2$, when considered as signed, has a value of -2_{10}; the same number has a value of $+254_{10}$ 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 unsigned numbers, the terms used are "below and above", while for signed numbers, the terms "less than" and "greater than" are used. The number $0111\ 1110_2$ when considered signed is greater than $0000\ 0000_2$, while the same number $0111\ 1110_2$ when considered unsigned is above $0000\ 0000_2$. The conditional transfer instructions for equality of two numbers are the same for both signed and unsigned numbers. This is because when two numbers are compared for equality irrespective of whether they are signed or unsigned, they will provide a zero result (ZF = 1) if equal or a nonzero result (ZF = 0) if not equal. Therefore, the same instructions apply for both signed and unsigned numbers for "equal to" or "not equal to" conditions, and the various signed and unsigned conditional branch instructions for determining the relationship between two numbers are as follows:

Signed		Unsigned	
Name	Alternate name	Name	Alternate name
JE disp8 (JUMP if equal)	JZ disp8 (JUMP if result zero)	JE disp8 (JUMP if equal)	JZ disp8 (JUMP if zero)
JNE disp8 (JUMP if not equal)	JNZ disp8 (JUMP if not zero)	JNE disp8 (JUMP if not equal)	JNZ disp8 (JUMP if not zero)
JG disp8 (JUMP if greater)	JNLE disp8 (JUMP if not less or or equal)	JA disp8 (JUMP if above)	JNBE disp8 (JUMP if not below or equal)
JGE disp8 (JUMP if greater or equal)	JNL disp8 (JUMP if not less)	JAE disp8 (JUMP if above or equal)	JNB disp8 (JUMP if not below)
JL disp8 (JUMP if less than)	JNGE disp8 (JUMP if not greater or equal)	JB disp8 (JUMP if below)	JNAE disp8 (JUMP if not above or equal)
JLE disp8 (JUMP if less or equal)	JNG disp8 (JUMP if not greater)	JBE disp8 (JUMP if below or equal)	JNA disp8 (JUMP if not above)

There are also conditional transfer instructions that are concerned with the setting of status flags rather than relationship between two numbers. The table below lists these instructions:

JC disp8	JUMP if carry, i.e., CF = 1
JNC disp8	JUMP if no carry, i.e., CF = 0
JP disp8	JUMP if parity, i.e., PF = 1
JNP disp8	JUMP if no parity, i.e., PF = 0
JO disp8	JUMP if overflow, i.e., OF = 1
JNO disp8	JUMP if no overflow, i.e., OF = 0
JS disp8	JUMP if sign, i.e., SF = 1
JNS disp8	JUMP if no sign, i.e., SF = 0
JZ disp8	JUMP if result zero, i.e., Z = 1
JNZ disp8	JUMP if result not zero, i.e., Z = 0

10. *Examples of LOOP instructions*

- LOOP BEGIN automatically decrements CX by 1 without affecting the flags and jumps to the label BEGIN if CX $\neq$ 0; goes to the next instruction if CX = 0 after autodecrement.
- LOOPNE NEXT automatically decrements CX by 1 and jumps to the label NEXT if CX $\neq$ 0 and ZF = 0. However, if CX = 0 after autodecrement or ZF = 1, execution will go on to the next instruction. Note that LOOPNE can be read as 'Loop while not equal and CX not 0.'

11. *Examples of interrupt instructions*

- INT n is a software interrupt instruction. This instruction pushes Flags, CS, and IP onto the stack and loads CS from the memory location 4n and IP from the memory location 4n+2. An interrupt service routine can be written at this location.
- IRET pops IP, CS, and flags from the stack.

12. *Examples of processor or control instructions*

- STD sets direction flag to one.
- CLI clears the IF flag in the status register to zero to disable maskable interrupts.

3.5 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 the data segment into register CL; MOV CX, [BX], on the other hand, moves the number from offsets [BX] and [BX + 1] in the data segment into CX.

The string instructions may define this in two ways. Typical examples are MOVSB or MOVS BYTE for 8-bit and MOVSW or MOVS WORD for 16-bit. Memory offsets can also be specified by including BYTEPTR for 8-bit and WORDPTR for 16-bit with the instruction. Typical examples are INC BYTEPTR [BX] and INC WORDPTR [BX].

3.6 ASM-86 Assembler Directives

The ASM-86 is the assembler written by Intel for the 8086 microprocessor. Other 8086 assemblers include the Microsoft 8086 assembler written for the IBM personal computer and the Hewllet Packard 8086 assembler written for the HP 64000 microcomputer development system. These assemblers allow the programmer to assign the values of CS, DS, SS, and ES.

These assemblers can be used to declare a variable's type as byte (8-bit), word (16-bit), or double word (4 bytes or 2 words) as follows:

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

BEGIN DW 0   ;  BEGIN is declared
             ;  as a word offset
             ;  and its content is 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.

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

3.6.1 SEGMENT and ENDS Directives

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

```
        ORG       5000H
JOHN    SEGMENT
X1      DB        0
X2      DB        0
X3      DB        0
JOHN    ENDS
```

The segment name is JOHN. The assembler will assign a numeric value 5000H to JOHN. 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 (except CS) must be loaded via a 16-bit register (AX, BX, CX, or DX).

3.6.2 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. An 8086 program may contain a number of logical segments containing code, data, and stack. The ASSUME pseudoinstruction assigns a logical segment to a physical segment at any given time. That is, the ASSUME directive tells the assembler what addresses will be in the segment registers at execution time.

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

SS containing the stack. But, it is the responsibility of the programmer to set these segment registers in the program.

3.6.3 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 directive 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 the data segment. 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        SEGMENT
ADDR        DW 3005H
ADDR        DW 2003FH
DATA        ENDS
STACK       SEGMENT
            DW 60 DUP (0)  ;  Assign 60(10) words.
                           ;  of stack with zeros.
STACK_TOP   LABEL WORD     ;  Initialize Stack_Top
                           ;  to the next
STACK       ENDS           ;  location after the
                           ;  top of the stack.
CODE        SEGMENT
            ASSUME CS: CODE, DS: DATA, SS: STACK
            MOV AX, STACK
            MOV SS, AX
            LEA SP, STACK_TOP
            MOV AX, DATA
            MOV DS, AX
            LEA SI, ADDR
            LEA DI, ADDRR
                 -          }
                 -          }  Main Program
                 -          }      Body
                 -          }
CODE        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 as words.

When the 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 in the ASSUME statement directs the assembler to compute the offsets or displacements of the following instructions from the start of the logical segment CODE. This means that when the program is run, the CS will contain the 16-bit value where the logical segment CODE was 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, F1H, 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
          -
          -
          -
          -
          MOV AX, DSEG       ;  Initialize
          MOV DS, AX         ;  DS
          MOV AX, SSEG       ;  Initialize
          MOV SS, AX         ;  SS
          MOV SP, STACK_TOP  ;  Initialize SP
          -
          -
          -
```

Note that typical 8086 assemblers on the Microsoft software and HP64000 system use the ORG directive to load CS and IP. For example, CS and IP can be initialized with 0500H and 0020H as follows:

For Microsoft 8086 assembler: ORG 05000020H
For HP64000 8086 assembler: ORG 0500H:0020H

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		

[36000H] = 02H, [36001H] = 03H
[50500H] = 05H
[30400H] = 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 [BX] pushes [36001H] and [36000H] into stack locations 64FFFH and 64FFEH, respectively. The SP is then decremented by 2 to contain the 20-bit physical address 64FFEH (SS = 6000H, SP = 4FFEH). Therefore, [64FFFH] = 03H and [64FFEH] = 02H.
ii) Before unsigned division, [DX] = 0400H, DH contains 04H, and [AX] = 00A9H = 169_{10}. After DIV DH, [AH] = Remainder = 01H and [AL] = Quotient = 2AH = 42_{10}.
iii) CWD sign extends AX register into the DX register. Since the sign bit of [AX] = 0, after CWD, [DX AX] = 000000A9H.
iv) MOVSB moves the content of memory addressed by the source [DS] and [SI] to the destination addressed by [ES] and [DI] and then it increments SI and DI by 1 for byte move. Since DF = 0, [DS] = 3000H, [SI] = 0400H, [ES] = 5000H, [DI] = 0500H, and the content of physical memory location 30400H is moved to physical memory 50500H. Since [30400H] = 02H, the location 50500H will also contain 02H. Since DF = 0 after MOVSB, [SI] = 0401H, [DI] = 0501H.
v) Since [BX] = 6000H, [DS] = 3000H, START = 05H, and the physical memory for destination = 36005H. After MOV START [BX], AL, memory location 36005H will contain A9H.

Example 3.2

Write 8086 assembly program to clear 100_{10} consecutive bytes.

Solution

```
0000                CODE SEGMENT AT 1000H          ;  Program to
                                                   ;  clear 100 bytes
                        ASSUME CS:CODE, DS:DATA
0000  B8 2000           MOV AX, 2000H              ;  Initialize
0003  8E D8             MOV DS, AX                 ;  Data Segment
0005  7D 1E 0000 R      LEA BX, ADDR               ;  Initialize BX
0009  B9 0064           MOV CX, 100                ;  Initialize loop
                                                   ;  count
000C  C6 07 00  START:MOV BYTEPTR[BX], 0           ;  Clear memory byte
000F  43                INC BX                     ;  Update pointer
0010  E2 FA             LOOP START                 ;  Decrement CX and
                                                      loop
0012  F4                HLT                        ;  Halt
0013                CODE ENDS
0000                DATA SEGMENT AT 2000H
0000  3000          ADDR DW 3000H                  ;  Store the initial
                                                   ;  address
0002                DATA ENDS                      ;  End program
                        END
```

```
Microsoft (R) Macro Assembler Version 5.10  5/13/92  13:12:12
                                                     Symbols-1

Segments and Groups:

Name                    Length   Align   Combine Class

CODE ..............     0013     AT      1000
DATA ..............     0002     AT      2000

Symbols:

Name                    Type     Value   Attr

ADDR ..............     L WORD   0000    DATA
START .............     L NEAR   000C    CODE
@CPU ..............     TEXT     0101h
@FILENAME .........     TEXT     ex32
@VERSION ..........     TEXT     510

    28 Source Lines
    28 Total Lines
     9 Symbols

47886 + 445965 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

Example 3.3

Write 8086 assembly program to computer $\sum_{i=1}^{N}$ XiYi where Xi and Yi are signed 8-bit numbers. N = 100. Assume no overflow.

Solution

```
0000                CODE SEGMENT AT 1000H
                         ASSUME CS:CODE, DS:DATA
0000  B8 2000            MOV AX, 2000H              ;  Initialize
0003  8E D8              MOV DS, AX                 ;  Data Segment
0005  B9 0064            MOV CX, 100                ;  Initialize loop
                                                    ;  count
0008  8D 1E 0000 R       LEA BX, ADDR1              ;  Load ADDR1
                                                    ;  into BX
000C  8D 36 0002 R       LEA SI, ADDR2              ;  Load ADDR2
                                                    ;  into SI
0010  BA 0000            MOV DX, 0000H              ;  Initialize sum
                                                    ;  to zero
0013  8A 07         START:MOV AL, [BX]              ;  Load data into
                                                    ;  AL
0015  F6 2C              IMUL BYTEPTR [SI]          ;  Signed 8x8
                                                    ;  multiplication
```

```
0017  03 D0          ADD DX, AX                ;  Sum XiYi
0019  43             INC BX                    ;  Update pointer
001A  46             INC SI                    ;  Update pointer
001B  E2 F6          LOOP START                ;  Decrement CX
                                               ;  and loop
001E            CODE ENDS
0000            DATA SEGMENT AT 2000H
0000  0000      ADDR1 DW 0000H                 ;  Location of Xi
0002  1000      ADDR2 DW 1000H                 ;  Location of Yi
0004            DATA ENDS
                     END                       ;  End program

Microsoft (R) Macro Assembler Version 5.10  5/13/92  13:26:33

Segments and Groups:

Name                    Length   Align  Combine Class

CODE ..............     001E     AT     1000
DATA ..............     0004     AT     2000

Symbols:

Name                    Type     Value  Attr

ADDR1 .............     L WORD   0000   DATA
ADDR2 .............     L WORD   0002   DATA
START .............     L NEAR   0013   CODE
@CPU ..............     TEXT     0101h
@FILENAME .........     TEXT     ex33
@VERSION ..........     TEXT     510

    39 Source Lines
    39 Total Lines
    10 Symbols

47848 + 445003 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

Example 3.4

Write 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 to by SI at 0500H, while the second word is stored in two consecutive locations with the low byte pointed to by BX at the offset 1000H. Store the result in memory pointed to by BX.

Solution

```
0000            CODE SEGMENT AT 1000H
                     ASSUME CS:CODE, DS:DATA
0000  BE 2000        MOV AX, 2000H             ;  Initialize
```

```
0003  8E D8          MOV DS, AX                ;  Data Segment
0005  B9 0002        MOV CX, 2                 ;  Initialize loop
                                               ;  count
0008  BE 0500        MOV SI, 0500H             ;  Initialize SI
000B  BB 1000        MOV BX, 1000H             ;  Initialize BX
000E  F8             CLC                       ;  Clear carry
000F  8A 04     START:MOV AL, [SI]             ;  Move data
0011  12 07          ADC AL, [BX]              ;  Perform addition
0013  27             DAA                       ;  BCD adjust
0014  88 07          MOV [BX], AL              ;  Store result
0016  46             INC SI                    ;  Update pointer
0017  43             INC BX                    ;  Update pointer
0018  E2 F5          LOOP START                ;  Decrement CX and
                                               ;  loop
001A  F4             HLT                       ;  Halt
001B            CODE ENDS
0000            DATA SEGMENT AT 2000H
0000            DATA ENDS
                     ENDS                      ;  End program

Microsoft (R) Macro Assembler Version 5.10

Segments and Groups:

Name                    Length   Align   Combine Class

CODE ..............     001B     AT      1000
DATA ..............     0000     AT      2000

Symbols:

Name                    Type     Value   Attr

START .............     L NEAR   000F    CODE
@CPU ..............     TEXT     0101h
@FILENAME .........     TEXT     ex34
@VERSION ..........     TEXT     510

    36 Source Lines
    36 Total Lines
     8 Symbols

47890 + 445961 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

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 memory pointed to by DI.

Solution

```
0000                CODE SEGMENT AT 1000H
                         ASSUME CS:CODE, DS:DATA
0000  B8 2000           MOV AX, 2000H              ;  Initialize
0003  8E D8             MOV DS, AX                 ;  Data Segment
0005  B9 0002           MOV CX, 2                  ;  Initialize loop
                                                   ;  count
0008  BE 0300           MOV SI, 0300H              ;  Initialize SI
000B  BF 0700           MOV DI, 0700H              ;  Initialize DI
000E  F8                CLC                        ;  Clear carry
000F  8A 04  START:     MOV AL, [SI]               ;  Move data
0011  12 05             ADC AL, [DI]               ;  Perform addition
0013  37                AAA                        ;  ASCII adjust
0014  88 05             MOV [DI], AL               ;  Store result
0016  46                INC SI                     ;  Update pointer
0017  47                INC DI                     ;  Update pointer
0018  E2 F5             LOOP START                 ;  Decrement CX and
                                                   ;  loop
001A                    HLT                        ;  Halt
001B                CODE ENDS
0000                DATA SEGMENT AT 2000H
0000                DATA ENDS
                         END                        ;  End program

Microsoft (R) Macro Assembler Version 5.10 10/05/93  22:25:10
                                                     Symbols-1

Segments and Groups:

Name                    Length   Align  Combine Class

CODE ..............     001B     AT     1000
DATA ..............     0000     AT     2000

Symbols:

Name                    Type     Value  Attr

START .............     L NEAR   000F   CODE
@CPU ..............     TEXT     0101h
@FILENAME .........     TEXT     ex35
@VERSION ..........     TEXT     510

    36 Source Lines
    36 Total Lines
     8 Symbols

47890 + 445961 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

Example 3.6

Write an 8086 assembly language program to compare a source string of 50_{10} words pointed to by an offset of 2000H in the data segment with a destination string pointed to by an offset 3000H in another segment. The program should be halted as soon as a match is found or the end of string is reached.

Solution

```
0000                CODE SEGMENT AT 1000H
                         ASSUME CS:CODE,DS:DATA,ES:DATAA
0000  B8 2000           MOV AX,2000H                ;  initialize
0003  8E D8             MOV DS,AX                   ;  Data segment
0005  B8 4000           MOV AX,4000H                ;  Initialize
0008  8E C0             MOV ES,AX                   ;  ES
000A  BE 2000           MOV SI,2000H                ;  Initialize SI
000D  BF 3000           MOV DI,3000H                ;  Initialize DI
0010  B9 0032           MOV CX,50                   ;  Initialize CX
0013  FC                CLD                         ;  DF is cleared so
                                                    ;  that SI and DI
                                                    ;  will autoincrement
                                                    ;  after compare
0014  F2/A7            REPNE CMPSW                 ;  Repeat CMPSW until
                                                    ;  CX=0 or until
                                                    ;  compared words are
                                                    ;  equal
0016  F4                HLT                         ;  Halt
0017                CODE ENDS
0000                DATA SEGMENT AT 2000H
0000                DATA ENDS
0000                DATAA SEGMENT AT 4000H
                         END                        ;  End program

Microsoft (R) Macro Assembler Version 5.10  10/05/92 11:45:41

Segments and Groups:

Name                    Length   Align   Combine Class

CODE ..............     0017     AT      1000
DATA ..............     0000     AT      2000
DATAA .............     0000     AT      4000

Symbols:

Name                    Type     Value   Attr

@CPU ..............     TEXT     0101h
@FILENAME .........     TEXT     TEST1
@VERSION ..........     TEXT     510

    20 Source Lines
```

```
    20 Total Lines
     8 Symbols

47884 + 433567 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

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 pointers to the signed 8-bit and 16-bit data, respectively.

Solution

```
0000                CODE SEGMENT AT 1000H
                         ASSUME CS:CODE,DS:DATA,SS:STACK
0000  B8 5000           MOV AX, 5000H              ;  Initialize
0003  8E D8             MOV DS, AX                 ;  Data Segment
0005  B8 6000           MOV AX, 6000H              ;  Initialize
0008  8E D0             MOV SS, AX                 ;  Stack Segment
000A  BC 0020           MOV SP, 0020H              ;  Initialize SP
000D  BB 2000           MOV BX, 2000H              ;  Initialize BX
0010  8A 04             MOV AL, [SI]               ;  Move 8-bit data
0012  8B 0D             MOV CX, [DI]               ;  Move 16-bit data
0014  9A 0000---R       CALL FAR PTR MULTI         ;  Call MULTI
                                                   ;  subroutine
0019  89 17             MOV [BX], DX               ;  Store high word of
                                                   ;  result
001B  89 47 02          MOV [BX+2], AX             ;  Store low word of
                                                   ;  result
001E  F4                HLT                        ;  Halt
001F                CODE ENDS
0000                SUBR SEMENT AT 7000H
                         ASSUME CS:SUBR
0000                MULTI PROC FAR                 ;  Must be called from
0000  51                PUSH CX                    ;  another code
                                                   ;  segment
0001  50                PUSH AX                    ;
0002  98                CBW                        ;  Sign extend AL
0003  F7 E9             IMUL CX                    ;  [DX][AX]←[AX]*[CX]
0005  58                POP AX                     ;
0006  59                POP CX                     ;
0007  CB                RET                        ;  Return
0008                MULTI ENDP                     ;  End of procedure
0004                SUBR ENDS                      ;  End subroutine
0000                DATA SEGMENT AT 5000H
0000                DATA ENDS
0000                STACK SEGMENT AT 6000H
0000                STACK ENDS
                         END                        ;  End program
```

```
Microsoft (R) Macro Assembler Version 5.10 10/05/93  22:40:32
                                                     Symbols-1

Segments and Groups:

Name                    Length   Align   Combine Class

CODE ..............     001F     AT      1000
DATA ..............     0000     AT      5000
STACK .............     0000     AT      6000
SUBR ..............     0004     AT      7000

Symbols:

Name                    Type     Value   Attr

MULTI .............     F PROC   0000    SUBR Length=0004
@CPU ..............     TEXT     0101h
@FILENAME .........     TEXT     EX37
@VERSION ..........     TEXT     510

    52 Source Lines
    52 Total Lines
    10 Symbols

47852 + 443951 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

Example 3.8

Write an 8086 assembly language program to subtract two 64-bit numbers. Assume SI and DI contain the starting address of the numbers. Store the result in memory pointed to by [DI].

Solution

```
0000                CODE SEGMENT AT 4000H
3000                     ORG 3000H
3000  05712FA2      DATAA DD 05712FA2H            ;  DATAA LOW
3004  0248A201            DD 0248A201H            ;  DATAA HIGH
3008  561A2604      DATAB DD 561A2604H            ;  DATAB LOW
300C  72A2B270            DD 72A2B270H            ;  DATAB HIGH
3010                DATA ENDS
0000                PROG SEGMENT
                         ASSUME CS:PROG,DS:DATA
0000  B8 4000             MOV AX, 4000H           ;  Initialize
0003  8E D8               MOV DS, AX              ;  DS
0005  BA 0004             MOV DX, 4               ;  Load 4 into DX
0008  8D 36 3000 R        LEA SI, DATAA           ;  Initialize SI
000C  8D 3E 30008 R       LEA DI, DATAB           ;  Initialize DI
0010  F8                  CLC                     ;  Clear carry
0011  8B 04 START:        MOV AX, [SI]            ;  Load DATAA
0013  19 05               SBB [DI], AX            ;  Perform subtraction
```

```
0015  83 C6 02        ADD SI, 2                ;  Update
0018  83 C7 02        ADD DI, 2                ;  pointers
001B  4A              DEC DX
001C  75 F3           JNZ START
001E  F4              HLT
001F              PROG ENDS
                      END

Microsoft (R) Macro Assembler Version 5.10  10/10/93  22:14:0
                                                      Symbols-1

Segments and Groups:

Name                    Length   Align   Combine Class

DATA ..............     3010     AT      4000
PROG ..............     001F     PARA    NONE

Symbols:

Name                    Type     Value   Attr

DATAA .............     L DWORD  3000    DATA
DATAB .............     L DWORD  3008    DATA
START .............     L NEAR   0011    PROG
@CPU ..............     TEXT     0101h
@FILENAME .........     TEXT     examp38
@VERSION ..........     TEXT     510

    25 Source Lines
    25 Total Lines
    10 Symbols

47518 + 428045 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

Example 3.9

Write an 8086 assembly language program that will perform the following operation:

$$5 * AL - 6 * BH + (BH/8) \rightarrow CX$$

Assume that the stack pointer is already initialized and all numbers are unsigned.

Solution

```
0000              PROG SEGMENT AT 5000H
0500                   ORG 0500H
                       ASSUME CS:PROG
0500  B3 05            MOV BL,5                 ;  Compute
0502  F6 E3            MUL BL                   ;  AX←5*AL
0504  8B C8            MOV CX,AX                ;  Store in CX
```

```
0506  B3 06          MOV BL,6                ;
0508  8A C7          MOV AL,BH               ;  Compute
050A  F6 E3          MUL BL                  ;  AX←6*BH
050C  29 C8          SUB CX,AX               ;  store in CX
050E  51             PUSH CX
050F  B1 03          MOV CL,3
0511  D2 EF          SHR BH,CL               ;  Compute BH/8
0513  8A DF          MOV BL,BH               ;  Convert BH/8 to
0515  B7 00          MOV BH,0                ;  16-bit unsigned
                                             ;  number in BX
0517  59             POP CX
0518  03 CB          ADD CX,BX               ;  Store final result
                                             ;  in CX
051A  F4             HLT
051B            PROG ENDS
                     END
```

```
Microsoft (R) Macro Assembler Version 5.10  1/16/80  12:07:44
                                                     Symbols-1

Segments and Groups:

Name                    Length   Align  Combine Class

PROG ..............     051B     AT     5000

Symbols:

Name                    Type     Value  Attr

@CPU ..............     TEXT     0101h
@FILENAME .........              TEXT   TEST1
@VERSION ..........              TEXT   510

    20 Source Lines
    20 Total Lines
     6 Symbols

47808 + 433643 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

Example 3.10

Write an 8086 assembly program that converts a number from Fahrenheit degrees to Celsius degrees. Each number is only one byte. The source byte is assumed to reside at offset 1000H in the data segment, and the destination at an offset of 2000H in the data segment.Discard the remainder of the result.

$$\text{use } C = \frac{(F-32)}{9} \times 5$$

Solution

```
0000                CODE SEGMENT
                         ASSUME CS:CODE,DS:DATA
0000  BE 1000            MOV SI, 1000H              ;  Initialize source
                                                    ;  pointer
0003  BF 2000            MOV DI, 2000H              ;  Initialize
                                                    ;  destination pointer
                                                    ;
0006  B4 00              MOV AH, 0                  ;  Clear AX high byte
0008  8A 04              MOV AL, [SI]               ;  Get degrees F
000A  98                 CBW                        ;  Sign extend
000B  2D 0020            SUB AX, 32                 ;  Subtract 32
                                                    ;
000E  B9 0005            MOV CX, 5                  ;  Get multiplier
0011  F7 E9              IMUL CX                    ;  Multiply by 5
                                                    ;
0013  B9 0009            MOV CX, 9                  ;  Get divisor
0016  F7 F9              IDIV CX                    ;  divide by 9 to
                                                    ;  get Celsius
                                                    ;
0017  88 05              MOV [DI], AL               ;  Put result in
                                                    ;  destination
                                                    ;
001A                CODE ENDS                      ;  End segment
                                                    ;
                         END                        ;

Microsoft (R) Macro Assembler Version 5.10  10/10/93 22:27:4
                                                     Symbols-1

Segments and Groups:

Name                    Length   Align   Combine Class

CODE ..............     0019     PARA    NONE

Symbols:

Name                    Type     Value   Attr

GO .................    L NEAR   0000    CODE
@CPU ...............    TEXT     0101h
@FILENAME ..........    TEXT     examp   310
@VERSION ...........             TEXT    510

    22 Source Lines
    22 Total Lines
    11 Symbols

47464 + 428099 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

3.7 System Design Using the 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.

3.7.1 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 multibus architecture. Note that in a particular application, the MN/MX must be tied to either HIGH or LOW.

The 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. S3 and S4 lines are decoded as follows:

A17/S4	A16/S3	Function
0	0	Extra segment
0	1	Stack segment
1	0	Code or no segment
1	1	Data segment

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 overlapping into 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 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.

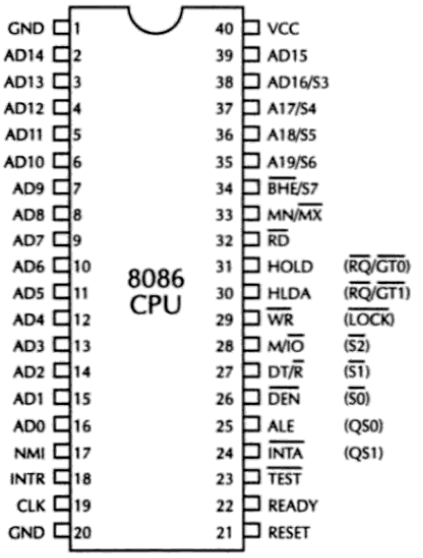
The 8086 tristates AD0-AD15 during Interrupt Acknowledge or Hold Acknowledge cycles.

$\overline{BHE}$/S7 is used as $\overline{BHE}$ (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 (AD15-AD8) of the data bus. $\overline{BHE}$ can be used in conjunction with AD0 to select memory banks. A thorough discussion is provided later. During all other cycles $\overline{BHE}$/S7 is used as S7 and the 8086 maintains the output level ($\overline{BHE}$) of the first clock cycle on this pin.

Common Signals		
Name	**Function**	**Type**
AD15-AD0	Address/Data Bus	Bidirectional, 3-State
A19/S6-A16/S3	Address/Status	Output, 3-State
$\overline{BHE}$/S7	Bus High Enable/Status	Output, 3-State
MN/$\overline{MX}$	Minimum/Maximum Mode Control	Input
$\overline{RD}$	Read Control	Output, 3-State
$\overline{TEST}$	Wait On Test Control	Input
READY	Wait State Control	Input
RESET	System REset	Input
NMI	Non-Maskable Interrupt Request	Input
INTR	Interrupt Request	Input
CLK	System Clock	Input
Vcc	+5V	Input
GND	Ground	

Mimimum Mode Signals (MN/MX = Vcc)		
Name	**Function**	**Type**
HOLD	Hold Request	Input
HLDA	Hold Acknowledge	Output
$\overline{WR}$	Write Control	Output, 3-State
M/$\overline{IO}$	Memory/IO Control	Output, 3-State
DT/$\overline{R}$	Data Transmit/Receive	Output, 3-State
$\overline{DEN}$	Data Enable	Output, 3-State
ALE	Address Latch Enable	Output
$\overline{INTA}$	Interrupt Acknowledge	Output

Maximum Mode Signals (MN/MX = GND)		
Name	**Function**	**Type**
RQ/GT1, 0	Request/Grant Bus Access Control	Bidirectional
LOCK	Bus Priority Lock Control	Output, 3-State
S2-S0	Bus Cycle Status	Output, 3-State
QS1, QS0	Instruction Queue Status	Output



Maximum mode pin functions (e.g., $\overline{LOCK}$) are shown in parenthesis

FIGURE 3.6 8086 pins and signals.

$\overline{RD}$ 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-ms reset pulse. It causes the 8086 to initialize registers DS, SS, ES, IP, and flags to all zeros. It also initializes CS to FFFFH. Upon

removal of the RESET signal from the RESET pin, the 8086 will fetch its next instruction from 20-bit physical address FFFF0H (CS = FFFFH, IP = 0000H).

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.

Since the reset vector is located at the physical address FFFF0H, there may not be enough locations available to write programs. Typical assemblers such as the Microsoft 8086 assembler use the following to jump to a different code segment to write programs:

```
ORG     0FFFF0H ;reset vector
        JMP     FAR BEGIN
        ORG     30000500H
BEGIN     -
          -       user
          -       program
          -
```

The above instruction sequence will allow the 8086 to jump to the offset BEGIN (offset 0500H) in code segment 3000H upon hardware reset where the user can write his/her programs.

To guarantee reset from power-up, the 8284 reset input must remain below 1.05 volts for 50 microseconds after Vcc has reached the minimum supply voltage of 4.5V. The $\overline{\text{RES}}$ 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:

$$Vc\ (t) = V\ (1 - \exp - (t/RC))$$

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

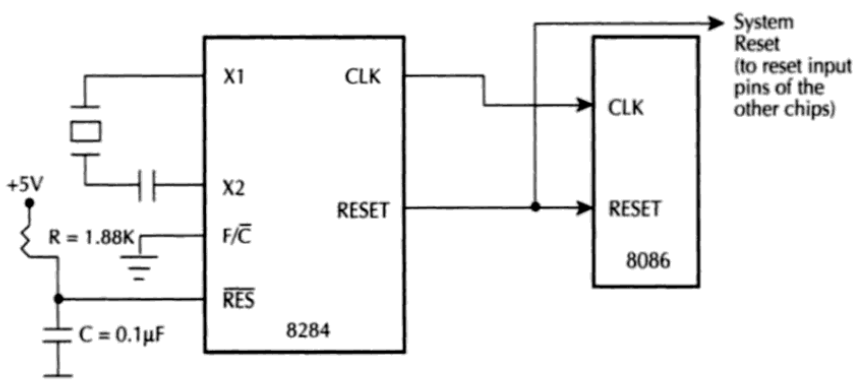


FIGURE 3.7 8086 reset and system resst.

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

- $\overline{DT}/R$ (Data Transmit/Receive). $\overline{DT}/R$ is an output signal required in minimum system that uses an 8286/8287 data bus transceiver. It is used to control direction of data flow through the transceiver.
- $\overline{DEN}$ (Data Enable) is provided as an output enable for the 8286/8287 in a minimum system which uses the transceiver.
- $\overline{DEN}$ is active LOW during each memory and I/O access and for $\overline{INTA}$ cycles.
- ALE (Address Latch Enable) is an output signal provided by the 8086 and can be used to demultiplex the AD0-AD15 into A0-A15 and D0-D15 at the falling edge of ALE. The 8086 ALE signal is similar to the 8085 ALE.
- M/$\overline{IO}$. This 8086 output signal is similar to the 8085 IO/$\overline{M}$. It is used to distinguish a memory access (M/$\overline{IO}$ = HIGH) from an I/O access (M/$\overline{IO}$ = LOW). When the 8086 executes an I/O instruction such as IN 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 MOVE AX, [SI].
- $\overline{WR}$. 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/$\overline{IO}$ signal.
- $\overline{INTA}$. The 8086 $\overline{INTA}$ is similar to the 8085 $\overline{INTA}$. 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.

The maximum clock frequencies of the 8086-1, 8086, and 8086-2 are 5 MHz, 10 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, 8086-1, and 8086-2 will be referred to as 8086 in the following. 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.

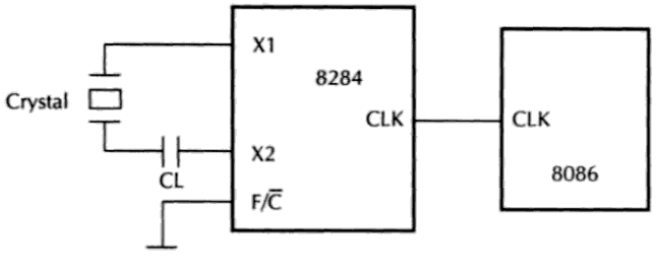


FIGURE 3.8 8284 clock generator connections to the 8086.

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 a 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 F/C input must be strapped to ground. This strapping option allows either the crystal or an 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. Two crystal manufacturers recommended by Intel are Crystle Corp. model CY 15A (15 MHz) and CTS Knight Inc. model CY 24 A (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, HLDA, WR, M/IO, DT/R, DEN, ALE, 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. S2, S1, S0 are 8086 outputs and are decoded as follows:

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	
0	0	0	Interrupt Acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Inactive

- $\overline{RQ}/GT0$, $\overline{RQ}/GT1$. 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 $\overline{RQ}/GT0$ having higher priority than $\overline{RQ}/GT1$. 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 ($\overline{RQ}/GT0$ or $\overline{RQ}/GT1$ 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 $\overline{RQ}/GT0$ or $\overline{RQ}/GT1$ pins. The 8086 then regains bus control.
- $\overline{LOCK}$. The 8086 outputs LOW on the $\overline{LOCK}$ pin to prevent other bus masters from gaining control of the system bus. The $\overline{LOCK}$ signal is activated by the 'LOCK' prefix instruction and remains active until the completion of the instruction that follows.
- QS1, QS0. The 8086 outputs to QS1 and QS0 pins to provide status to allow external tracking of the internal 8086 instruction queue as follows:

QS1	QS0	
0	0	No operation
0	1	First byte of op code from queue
1	0	Empty the queue
1	1	Subsequent byte from queue

QS0 and QS1 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.

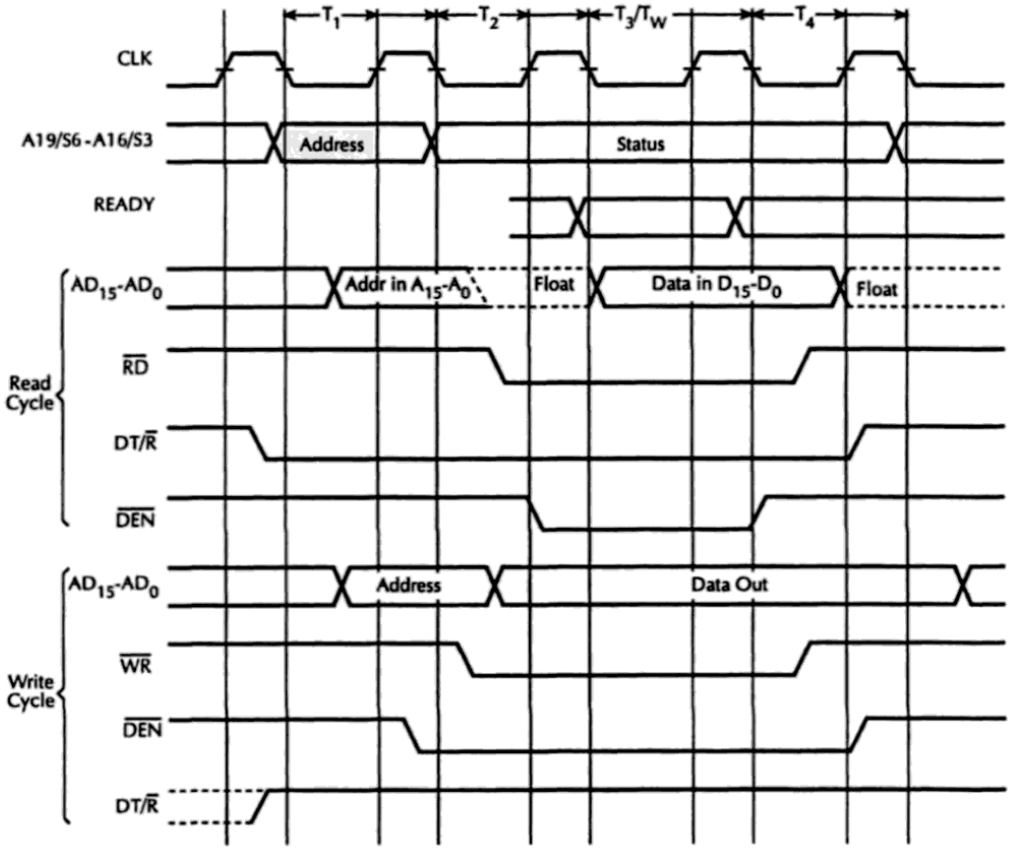


FIGURE 3.9 Basic 8086 bus cycle.

3.7.2 8086 Basic System Concepts

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

3.7.2.a 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 tristates 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 (S6, 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. 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

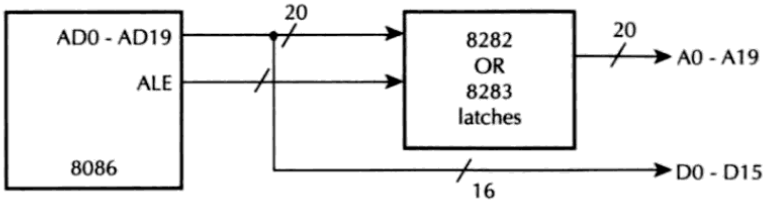


FIGURE 3.10 Separate address and data buses.

with 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 T3 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 $\overline{\text{DEN}}$ 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 throughout the four bus cycles to transmit (write) data to the transceiver during T3-T4.

3.7.2.b 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 falling 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 a 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 $\overline{\text{BHE}}$ (Bus High Enable) and A0 as follows:

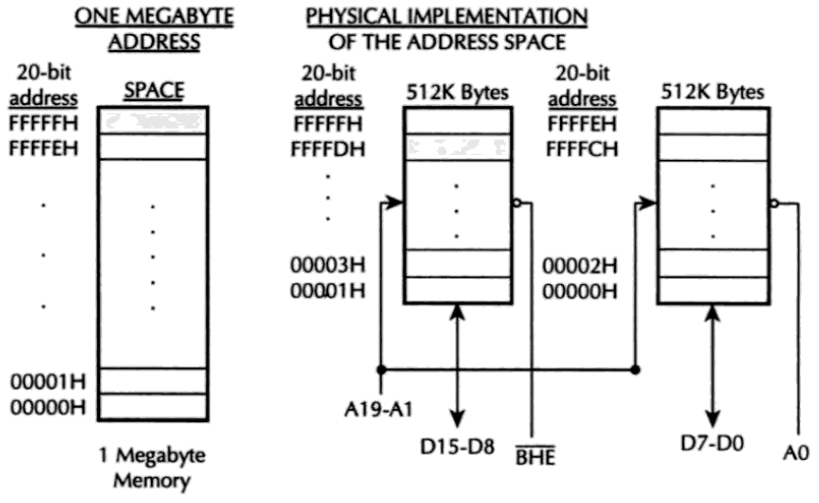


FIGURE 3.11 8086 memory.

$\overline{BHE}$	A0	Byte transferred
0	0	Both bytes
0	1	Byte to/from odd address via D_{15}–D_8
1	0	Byte to/from even address via D_7–D_0
1	1	None

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-A1. 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

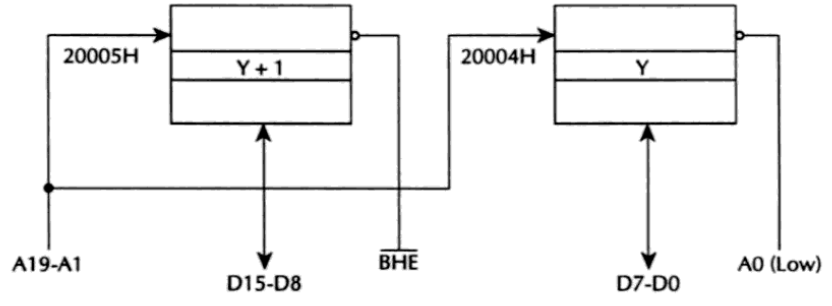


FIGURE 3.12 Even-addressed word transfer.

LOW on both A0 and $\overline{BHE}$, enabling both banks simultaneously. The 8086 outputs [CL] to D7-D0 lines and [CH] to D15-D8 lines with $\overline{WR}$ 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 (A19-A1 = 20005H in the first cycle, and 20006H in the second cycle). In the first bus cycle, the 8086 outputs HIGH on A0, LOW on $\overline{BHE}$, and thus enables the odd-addressed bank and disables the even-addressed bank. The 8086 also outputs LOW on $\overline{RD}$ 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 $\overline{BHE}$, and thus enables the even-addressed bank and disables the odd-addressed bank. The 8086 also outputs LOW on $\overline{RD}$ and HIGH on M/IO pins. The selected even-addressed memory bank places [20006H] on D7-D0 lines. The 8086 reads this data into CH.

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 of 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.

3.7.3 Interfacing with Memories

Figure 3.13 shows a general block diagram of an 8086 memory array. In Figure 3.13, 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 $\overline{BHE}$ and A0.

3.7.3.a 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 $\overline{BHE}$ 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

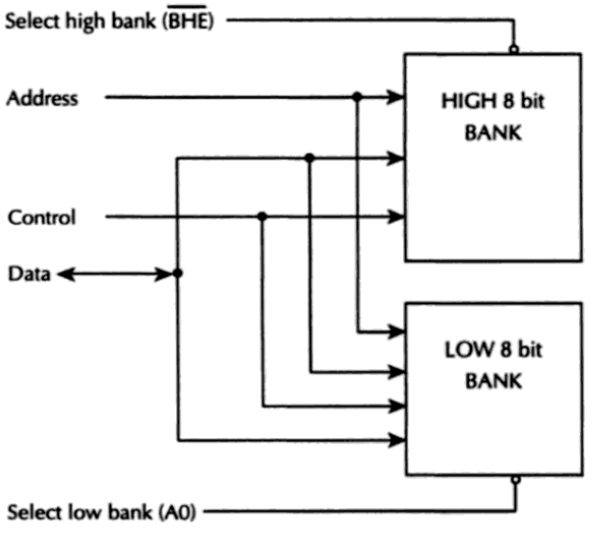


FIGURE 3.13 8086 memory array.

the chip is in active or standby power mode). The 8086 address lines must be connected to the address pins of 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.

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.

3.7.3.b Static RAMs

Since static RAMs are read/write memories, both A0 and $\overline{\text{BHE}}$ 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 (containing odd or even addresses), the memory data lines must be connected to either the upper half AD15-AD8 for static RAM with odd addresses or lower half AD7-AD0 of the 8086 data lines for static RAM with even addresses.

3.7.3.c Dynamic RAMs

Dynamic RAMs store information as charges in capacitors. Since 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 into a system. To relieve the designer of most of these complicated interfacing tasks, Intel provides the 8202 dynamic 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 the Intel manuals.

3.7.4 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.

Data transfer using the memory-mapped I/O is accomplished by using memory-oriented instructions such as MOV reg 8 or reg 16, [BX] and MOV [BX], reg 8 or reg 16 for inputting and outputting 8- or 16-bit data from or to an 8-bit register or a 16-bit register addressed by the 20-bit memory-mapped port location computed from DS and BX.

Note that the indirect I/O transfer method is desirable for service routines that handle more than one device such as multiple printers by allowing the desired device (a specific printer) to be passed to the procedure as a parameter.

Devices with I/O ports can be connected to either the upper or lower half of the data bus. If the I/O port chip is connected to the 8086 lower half of the data lines (AD0-AD7), the port addresses will be even (A0 = 0). On the other hand, the port addresses will be odd ($\overline{BHE}$ = 0) if the I/O port chip is connected to the upper half of the 8086 data lines (AD8-AD15).

3.8 8086-Based Microcomputer

In this section, an 8086 will be interfaced in the minimum mode to provide 2K × 16 EPROM, 2K × 16 static RAM, and six 8-bit I/O ports. 2716 EPROM, 6116 static RAM, and 8255 I/O chips will be used for this purpose. Memory and I/O maps will also be determined. Figure 3.14 shows a hardware schematic for accomplishing this.

Three 74LS373 latches are used. The 2716 is a 2K × 8 ultraviolet EPROM with eleven address pins A0-A10 and eight data pins O0-7. Two 2716s are used. The 8086 A1-A11pins are connected to the A0-A10 pins of these chips. The 2716 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 odd 2716 are connected to the 8086 D8-D15 pins. The 8086 reads data via D8-D15 pins for odd addresses.

The 6116 is a 2KX8 SRAM. Two 6116s are used.

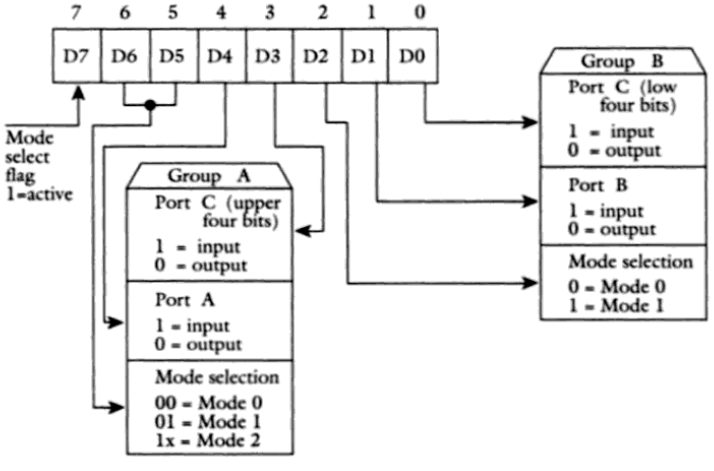
Table 3.2 shows memory and I/O maps of the 8086-based microcomputer of Figure 3.14. Note that the reset vector FFFF0H (CS = FFFFH, IP = 0000H) is included in the 2716.

For I/O ports, two 8255 chips 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:

A1	A0	
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Control register

The structure of the control register is given below:



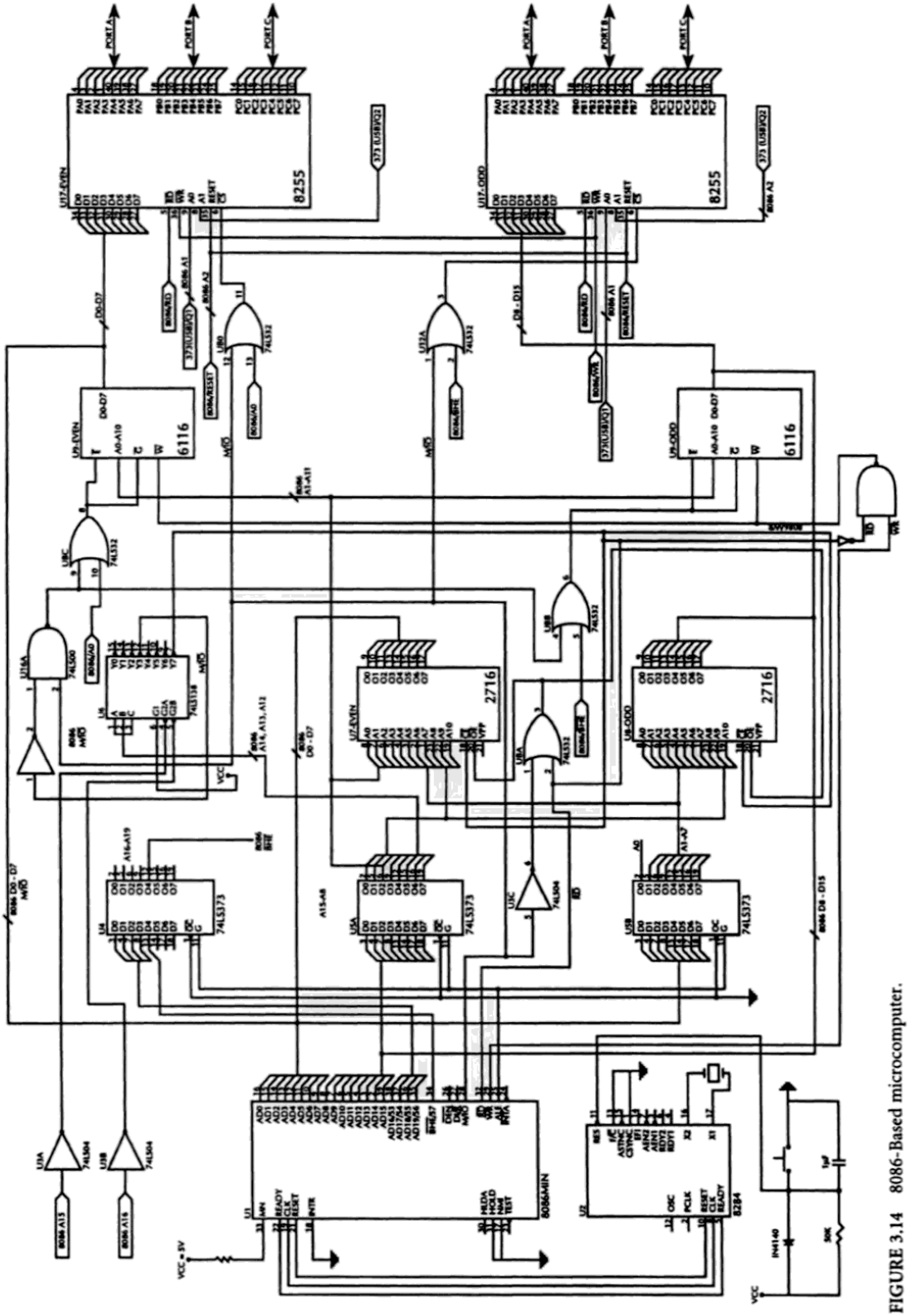


FIGURE 3.14 8086-Based microcomputer.

TABLE 3.2 Memory and I/O Maps

(a) Physical Memory Map

2716 (U7) EVEN # EPROM																			
A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
TO																			
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
don't cares asumed high			to enable the 74LS138		output line 7 of the 74LS138														
FF000H, FF002H, ... FFFFEH																			

2716 (U8) ODD # EPROM																			
A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1
TO																			
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
don't cares asumed high			to enable the 74LS138		output line 7 of the 74LS138														
FF001H, FF003H, ... FFFFFH																			

6116 (U9) EVEN # RAM																			
A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
TO																			
1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0
don't cares asumed high			to enable the 74LS138		output line 3 of the 74LS138														
FB000H, FB002H, ... FBFFEH																			

6116 (U10) ODD # RAM																			
A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1
TO																			
1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
don't cares asumed high			to enable the 74LS138		output line 3 of the 74LS138														
FB001H, FB003H, ... FBFFFH																			

TABLE 3.2 Memory and I/O Maps (*continued*)

(b) Logical Memory Map

Chip	Segment Value	Offset
Even 2716	FF00H	0000H, 0002H, 0004H, ..., 0FFEH
Odd 2716	FF00H	0001H, 0003H, 0005H, ..., 0FFFH
Even 6116	FB00H	0000H, 0002H, 0004H, ..., 0FFEH
Odd 6116	FB00H	0001H, 0003H, 0005H, ..., 0FFFH

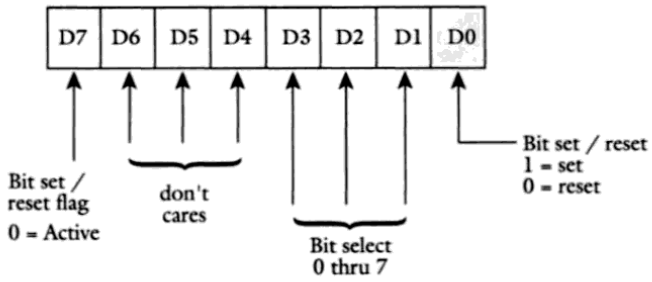
I/O Map	
Even 8255	Port A = F8H
	Port B = FAH
	Port C = FCH
	CSR = FEH
Odd 8255	Port A = F9H
	Port B = FBH
	Port C = FDH
	CSR = FFH

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 A and upper 4 bits of port C, while Group B defines all 8 bits of port B 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 to 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 conjuction 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:



This format is used to set or reset the output on a pin of port C or when enabling the interrupt output signals for handshake data transfer. For example, the following 8-bits

0	X X X	1 1 0	0
↑	don't	bit 6	↑
bit set/ reset mode	care		clear

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, only 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 by means of numerical example. Consider that the control word format is 1 000 0 010_2. With this data in the control register, all 8 bits of port A are configured as outputs, 8 bits of port C are also configured as outputs, but all 8 bits of port B are defined as inputs. On the other hand, outputting 1 0 011 011_2 into the control register will configure all 3 8-bit ports (ports A, B, and C) as inputs.

Let us decode the I/O port addresses. One 8255 will contain the odd-addressed ports since it is enabled by $\overline{BHE}$, while the other 8255 will include the even-addressed ports since it is enabled by A0.

Since the 8086 A1 and A2 pins are utilized in addressing the ports, bits A3-A7 are don't cares and are assumed to be ones here. Note that A0 = 1 for odd-addressed ports, while A0 = 0 for even-addressed ports.

For 8255-1 (Odd-Addressed Ports)

	Address								
Port name	A7	A6	A5	A4	A3	A2	A1	A0	
Port A	1	1	1	1	1	0	0	1	= F9H
Port B	1	1	1	1	1	0	1	1	= FBH
Port C	1	1	1	1	1	1	0	1	= FDH
Control Register	1	1	1	1	1	1	1	1	= FFH

For 8255-2 (Even-Addressed Ports)

	Address								
Port name	A7	A6	A5	A4	A3	A2	A1	A0	
Port A	1	1	1	1	1	0	0	0	= F8H
Port B	1	1	1	1	1	0	1	0	= FAH
Port C	1	1	1	1	1	1	0	0	= FCH
Control Register	1	1	1	1	1	1	1	0	= FEH

In the above, standard I/O technique is used. The 8255s can also be interfaced to the 8086 using memory-mapped I/O. In this case the 8086 M/IO pin will not be used. The 20-bit physical addresses for the ports can be determined in a similar way by considering any unused 8086 address bits (A3-A19) as don't cares.

From the above discussion, the following points can be summarized:

1. For ROMs and EPROMs, $\overline{\text{BHE}}$ and A0 are not required to be part of chip enable/select decoding.
2. For RAMs and I/O chips, both $\overline{\text{BHE}}$ and A0 must be used in chip select logic.
3. For ROMs/EPROMs and RAMs, both odd and even addressed chips are required. However, for I/O chips, either an odd-addressed I/O chip or even-addressed I/O chip or both can be used depending on the number of ports required in an application.
4. For interfacing ROMs/EPROMs and RAMs to the 8086, the same chip select logic must be used for both even and its corresponding odd memory bank. For example, the even memory chip containing address 00000H, 00002H, 00004H, ..., must have the same chip select logic as its odd counterpart containing address 00001H, 00003H, 00005H,.... The memory map of the two memory banks will be distinguished by the value of A0 (A0 = 0 for even addresses and A0 = 1 for odd addresses).

Example 3.11

Assume 8086/8255 based microcomputer. Write an 8086 assembly language program to drive an LED connected to bit 2 of Port B based on two switch inputs at bits 6 and 7 of Port A. If both switch inputs are HIGH or LOW, turn the LED "ON"; otherwise turn the LED "OFF".

Solution

```
                TITLE   PROGRAM
1000            ORG     1000H
0000            PROG    SEGMENT
                        ASSUME CS:PROG
=0001           PORTA   EQU 01H
```

```
=0003          PORTB    EQU 03H
=0007          CNTRL    EQU 07H
0000 B0 90               MOV AL,90H
0002 E6 07               OUT CNTRL,AL
0004 E4 01     BEGIN:    IN AL,PORTA
0006 24 C0               AND AL,0C0H
0008 7A 06               JPE LEDON
000A B0 00               MOV AL,00H
000C E6 03               OUT PORTB,AL
000E EB F4               JMP BEGIN
0010 B0 04     LEDON:    MOV AL,04H
0012 E6 03               OUT PORTB,AL
0014 EB EE               JMP BEGIN
0016           PROG      ENDS
                         END

Microsoft (R) Macro Assembler Version 5.10  10/23/93 21:58:39
PROGRAM                                              Symbols-1

Segments and Groups:

Name                    Length   Align   Combine Class

PROG ..............     0016     PARA    NONE

Symbols:

Name                    Type     Value   Attr

BEGIN .............     L NEAR   0004    PROG
CNTRL .............     NUMBER   0007
LEDON .............     L NEAR   0010    PROG
PORTA .............     NUMBER   0001
PORTB .............     NUMBER   0003
@CPU ..............     TEXT     0101h
@FILENAME .........              TEXT    test1
@VERSION ..........              TEXT    510

    19 Source Lines
    19 Total Lines
    11 Symbols

47884 + 433567 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

Example 3.12

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

Port A = F9H
Port B = FBH
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

```
1000                ORG     1000H
0000                CODE    SEGMENT
                            ASSUME CS:CODE
=00F9               PORTA   EQU 0F9H
=00FB               PORTB   EQU 0FBH
=00FF               CNTRL   EQU 0FFH
0000 B0 90                  MOV AL, 90H     ;  Configure Port A as
                                               input
0002 E6 FF                  OUT CNTRL, AL   ;  Port B as output
0004 E4 F9          BEGIN:  IN AL, PORTA    ;  Input switches
0006 24 07                  AND AL, 07H     ;  Mask high five bits
0008 3C 07                  CMP AL, 07H     ;  Are all three inputs
                                            ;  HIGH?
000A 74 12                  JZ ODD          ;  If so, turn LEDON
000C 3C 01                  CMP AL, 01H     ;  Is only input 0 HIGH?
000E 74 0E                  JZ ODD          ;  If so, turn LED ON
0010 3C 02                  CMP AL, 02H     ;  Is only input 1 HIGH?
0012 74 0A                  JZ ODD          ;  If so, turn LED ON
0014 3C 04                  CMP AL, 04H     ;  Is only input 2 HIGH?
0016 74 06                  JZ ODD          ;  If so, turn LED ON
0018 B0 00                  MOV AL, 00H     ;  Else turn LED
001A E6 FB                  OUT PORTB, AL   ;  OFF
001C EB E6                  JMP BEGIN       ;  REPEAT
001E B0 04          ODD:    MOV AL, 04H     ;  Turn LED
0020 E6 FB                  OUT PORTB, AL   ;  ON
0022 EB E0                  JMP BEGIN       ;  REPEAT
0024                CODE    ENDS
                            END

Microsoft (R) Macro Assembler Version 5.10   11/14/93 16:40:4
                                                    Symbols-1

Segments and Groups:

Name                    Length   Align   Combine Class

CODE ..............     0024     PARA    NONE

Symbols:

Name                    Type     Value   Attr

BEGIN .............     L NEAR   0004    CODE
CNTRL .............     NUMBER   00FF
ODD ...............     L NEAR   001E    CODE
PORTA .............     NUMBER   00F9
```

```
PORTB ..........    NUMBER   00FB
@CPU ...........    TEXT     0101h
@FILENAME ......             TEXT    examp17
@VERSION .......             TEXT    510

   26 Source Lines
   26 Total Lines
   11 Symbols

47576 + 427923 Bytes symbol space free

    0 Warning Errors
    0 Severe Errors
```

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 003FEH. 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 32-bit vectors. If fewer than 256 interrupts are defined in the system, the user is required to provide enough memory for the interrupt pointer table for obtaining the defined interrupts.

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 word 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 uses the 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 address for IP = 4 * nn and the table address for CS = (4 * nn) + 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:

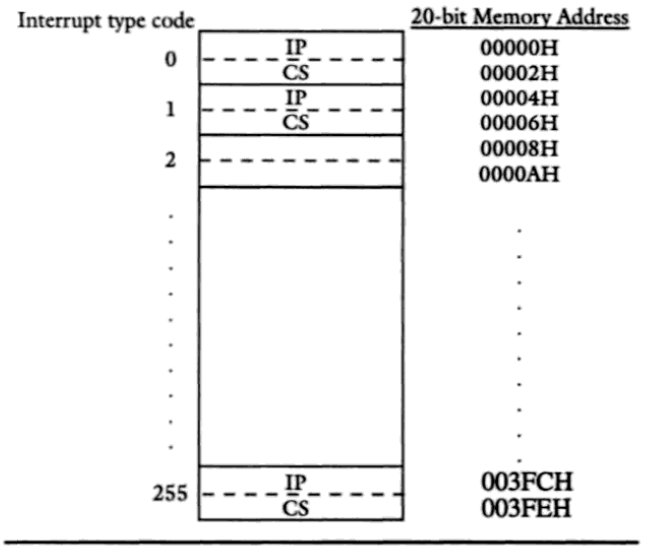
```
Address for IP = 4 * 2 = 00008H
Address for CS = (4 * 2) + 2 = 0000AH
```

The 8086 loads the values of IP and CS from the 20-bit physical addresses 00008H 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 3.3.

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 routines must 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:

TABLE 3.3 8086 Interrupt Pointer Table



- 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 set to one as follows:

```
PUSHF                    ;  Save flags
MOV BP, SP               ;  Move [SP] to [BP]
OR [BP + 0], 0100H       ;  Set TF
POPF                     ;  Pop flags
```

Note that in the above [BP + 0] rather than [BP] is used since BP cannot be used without displacement.

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 disabled by 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 (OF) is set and the INTO instruction is executed. The overflow flag is affected, for example, after execution of signed arithmetic such as IMUL (signed multiplication) instruction. The user can execute the INTO instruction after the IMUL. 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 nn. The INT nn instruction is not maskable by the interrupt enable flag (IF). The INT nn 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 nn instruction to generate software (internal) interrupts.

3.9.3 User-Defined Hardware (Maskable Interrupts, Type Codes 32_{10}–255_{10})

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 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 3.15. 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

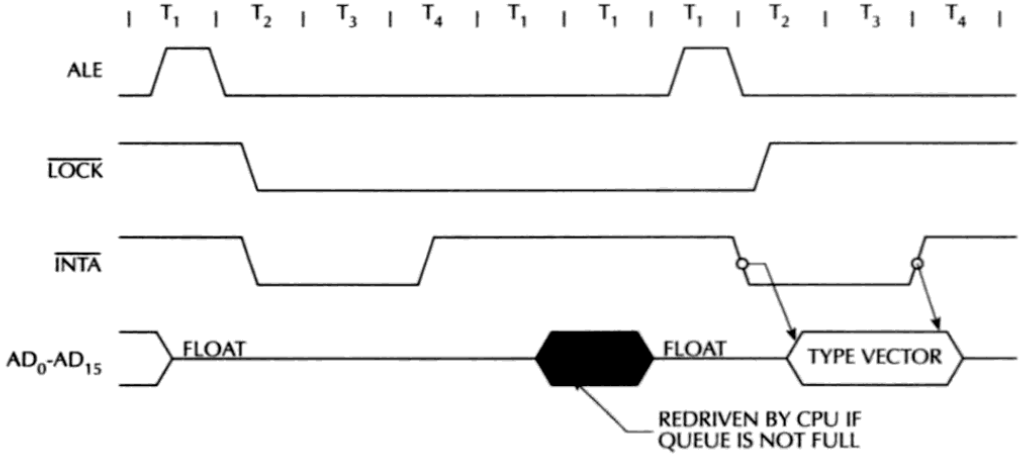


FIGURE 3.15 Interrupt acknowledge sequence.

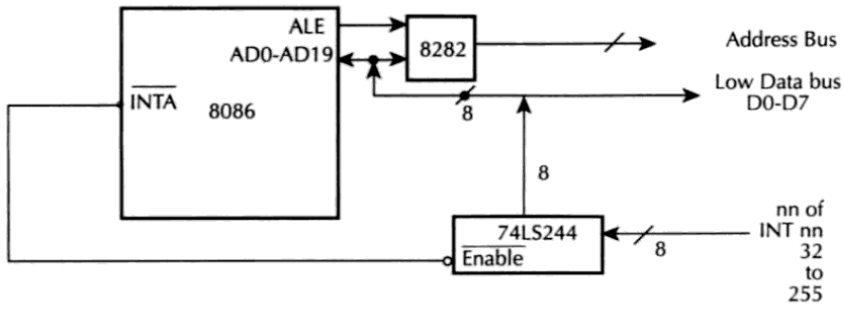


FIGURE 3.16 Servicing the INTR in the minimum mode.

information. The ALE is useful in maximum systems with multiple 8259A priority interrupt controllers. During the INTA bus cycles, DT/R 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 external hardware must place the type code on the lower half of the 16-bit data bus during the second cycle.

In the minimum mode, the $\overline{M/IO}$ is low indicating I/O operation during the $\overline{INTA}$ 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 3.16 shows a simplified interconnection between the 8086 and 74LS244 for servicing the INTR. INTA enables 74LS244 to place the type code nn on the 8086 data bus.

In the maximum mode, the status lines S0 to S2 will enable the INTA 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 RQ/GT input and to prevent bus arbitration logic from releasing the bus between INTA s in multimaster 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 nn 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 register and instruction pointer into the stack. Flags TF and IF 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. S4, S3 has the value 10_2 to indicate no segment register selection.

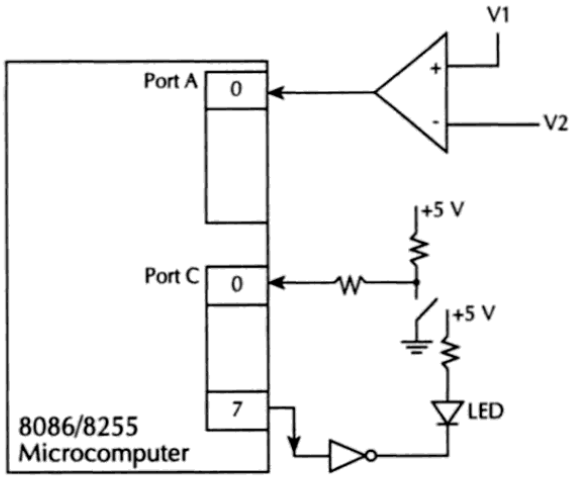
As far as the 8086 interrupt priorities are concerned, single-step interrupt has the highest priority, followed by NMI, followed by the software interrupts (all interrupts except single step, NMI, and INTR interrupts). This means that a simultaneous NMI and single step will cause the NMI service routine to follow single 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 the NMI service routine, and single stepping will not be serviced. However, if software interrupt and single stepping are pending,

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 entry 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.

A priority interrupt controller such as the 8259A can be used with the 8086 INTR to provide eight levels of hardware interrupts. The 8259A has built-in features for expansion of up to 64 levels with additional 8259As. The 8259A is programmable and can be readily used with the 8086 to obtain multiple interrupts from the single 8086 INTR pin.

Example 3.13



i) In the above, the 8086/8255 microcomputer is required to perform the following:

If V1 > V2, turn the LED "ON" if the switch is open. If V1 ≤ V2, loop and test again.

Write an 8086 assembly language program to accomplish the above.

ii) Repeat part i) using NMI and INTR interrupts. Write service routine at IP = 4000H, CS = 3000H. Assume that the stack pointer and stack segment are already initialized.

Solution

i)

```
0000            JOHN   SEGMENT AT 1000H
                       ASSUME CS: JOHN
=0001           PORTA  EQU 01H
=0005           PORTC  EQU 05H
=0007           CNTRL  EQU 07H
0000 B0 91             MOV AL, 91H          ;  Configure
0002 E6 04             OUT CNTRL,AL         ;  Port A and Port C
```

```
0004 E4 01   START:  IN AL, PORTA        ;  Input comparator
0006 24 01           AND AL, 01H         ;  Check if high
0008 74 FA           JZ START            ;  Jump back if low
000A E4 03           IN AL, PORTC        ;  Input switch
000C B1 02           MOV CL, 2
000E D2 D8           RCR AL, CL
0010 E6 03           OUT PORTC, AL       ;  Output to LED
0012 F4              HLT
0013         JOHN    ENDS
                     END

Microsoft (R) Macro Assembler Version 5.10   11/17/93 22:35:4
                                                  Symbols-1

Segments and Groups:

Name                 Length   Align  Combine Class

JOHN .............   0013     AT     1000

Symbols:

Name                 Type     Value  Attr

CNTRL ............   NUMBER   0007
PORTA ............   NUMBER   0001
PORTC ............   NUMBER   0005
START ............   L NEAR   0004   JOHN
@CPU .............   TEXT     0101h
@FILENAME ........   TEXT     EXP318A
@VERSION .........   TEXT     510

    17 Source Lines
    17 Total Lines
    10 Symbols

47576 + 427939 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

ii)

Using NMI

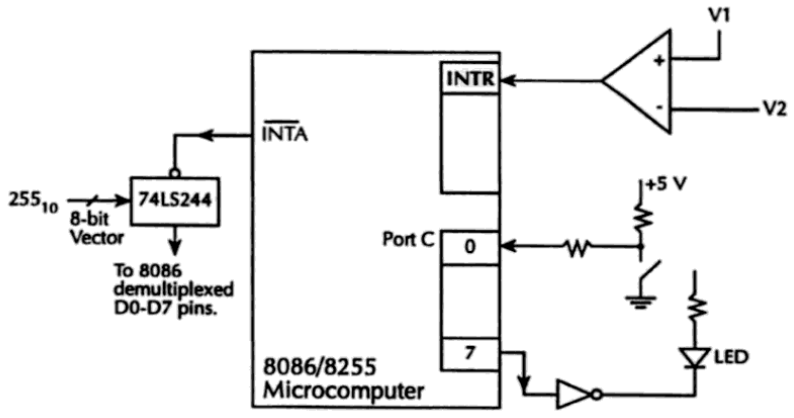
Main program

```
0000         JOHN    SEGMENT AT 2000H
                     ASSUME CS: JOHN
=0001        PORTA EQU 01H
=0005        PORTC EQU 05H
=0007        CNTRL EQU 07H
0000 B0 91           MOV AL, 91H         ;  Configure
0002 E6 07           OUT CNTRL,AL        ;  Port A and Port C
0004 9B      WAIT:   JMP WAIT
0005 F4              HLT                 ;  Wait for interrupt
0006         JOHN    ENDS
                     END
```

Service routine

```
0008                            ORG             00000008H
0008  40003000                  DD              40003000H
30004000                        ORG             30004000H
30004000  E4  03    START:      IN AL, PORTC    ;  Input switch
30004002  B1  02                MOV CL, 2
30004004  D2  D8                RCR AL, CL
30004006  E6  03                OUT PORTC, AL   ;  Output to LED
30004008  CF                    IRET

Microsoft (R) Macro Assembler Version 5.10  11/17/93 22:18:20
                                                    Symbols-1

Segments and Groups:

Name                    Length   Align   Combine Class
JOHN ..............     0006     AT      2000

Symbols:

Name                    Type     Value   Attr
CNTRL .............     NUMBER   0007
PORTA .............     NUMBER   0001
PORTC .............     NUMBER   0005
@CPU ..............     TEXT     0101h
@FILENAME .........              TEXT    test
@VERSION ..........              TEXT    510

    11 Source Lines
    11 Total Lines
     9 Symbols

47884 + 433567 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

Using INTR (vector 256_{10})



Main program

```
0000            JOHN    SEGMENT AT 3000H
                        ASSUME CS: JOHN
=0001           PORTA   EQU 01H
=0005           PORTC   EQU 05H
=0007           CNTRL   EQU 07H
0000 FB                 STI                     ;  Enable maskable
                                                ;  interrupt
0001 B0 91              MOV AL, 91H             ;  Configure Port A and
0003 E6 04              OUT CNTRL, AL           ;  Port C
                WAIT:   JMP WAIT                ;  Wait for interrupt
0005 F4                 HLT
0006            JOHN    ENDS
                        END
```

Service routine

```
03FC                            ORG   000003FCH
03FC 40003000                   DD    40003000H
30004000                        ORG   30004000H
30004000 E4 03                  IN AL, PORTC    ;  Input switch
30004002 B1 02                  MOV CL, 2
30004004 D2 D8                  RCR AL, CL
30004006 E6 03                  OUT PORTC, AL   ;  Output to LED
30004008 CF                     IRET
```

```
Microsoft (R) Macro Assembler Version 5.10   11/17/93 22:55:4
                                                Symbols-1

Segments and Groups:

Name                    Length   Align   Combine Class

JOHN .............      0006     AT      3000

Symbols:

Name                    Type     Value   Attr

CNTRL .............     NUMBER   0007
PORTA .............     NUMBER   0001
PORTC .............     NUMBER   0005
@CPU ..............     TEXT     0101h
@FILENAME .........              TEXT    EXP318C
@VERSION ..........              TEXT    510

    11 Source Lines
    11 Total Lines
     9 Symbols

47576 + 427939 Bytes symbol space free
```

```
0 Warning Errors
0 Severe Errors
```

3.10 8086 DMA

When configured in the minimum mode (MN/$\overline{\text{MX}}$ 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 8257 and 8237 can be used with the 8086. The 8257 or 8237 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 system bus until the HOLD pin 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 alternately select these two banks to access logically adjacent bytes in memory.

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 i APX 86 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 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 and 8284 input clocks? Does the 8086 have an on-chip clock circuitry? Comment.

3.6 What is the purpose of the TF bit in the FLAG register?

3.7 If [BL] = 36_{16} (ASCII code for 6) and [CL] = 33_{16} (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 AX register after execution of the following 8086 instruction sequence:

```
MOV AX, 0F180H
CBW
CWD
```

3.9 Determine the addressing modes for the following instructions:

(a) `MOV CH, 8`
(b) `MOV AX, DS:START`
(c) `MOV [SI], AL`

(d) `MOV SI, BYTEPTR[BP+2][DI]`

3.10 Consider MOV BX, DS: BEGIN. How many memory accesses are required by the 8086 to execute the above instruction if BEGIN = 0401H.

3.11 Write an 8086 assembly program to implement the following Pascal segment:

```
SUM: =0; for i: = 0 to 15 do
SUM: = SUM + A(i)
```

Assume CS and DS are already initialized. A(i)s are 16-bit numbers.

3.12 Write and 8086 assembly program to add two 128-bit numbers stored in memory in consecutive locations.

3.13 What are the remainder, quotient, and registers containing them after execution of the following instruction sequence?

```
MOV DX, 0
MOV AX, -5
MOV BX, 2
IDIV BX
```

3.14 Write an 8086 assembly language program to divide A5721624H by F271H. Store the remainder and quotient onto the stack. Assume that the numbers are signed and stored in the stack as follows:

Assume that the stack segment and stack pointer are already initialized.

3.15 Write an 8086 assembly language program to compute X = Y + Z – 12FFH where X, Y, Z are 64-bit variables. The lower 32 bits of Y and Z are stored respectively at offset 5000H and 5008H followed by the upper 32 bits. Store the lower 32 bits of the 64-bit result at offset 6000H followed by the upper 32 bits.

3.16 Assume that registers AL, BX and DX CX contain a signed byte, a signed word, and a signed 32-bit number respectively. Write an 8086 assembly language program that will compute the signed 32-bit result:

$$AL + BX - DXCX \rightarrow DXCX$$

3.17 Write an 8086 assembly language program to compute X = 5 * Y + (Z/W) where offsets 5000H, 5002H, and 5004H respectively contain the 16-bit signed integers Y, Z, and W. Store the 32-bit result in memory starting at offset 5006H. Discard the remainder of Z/W.

3.18 Write an 8086 instruction sequence to clear the trap flag to zero.

3.19 Write an 8086 subroutine to compute

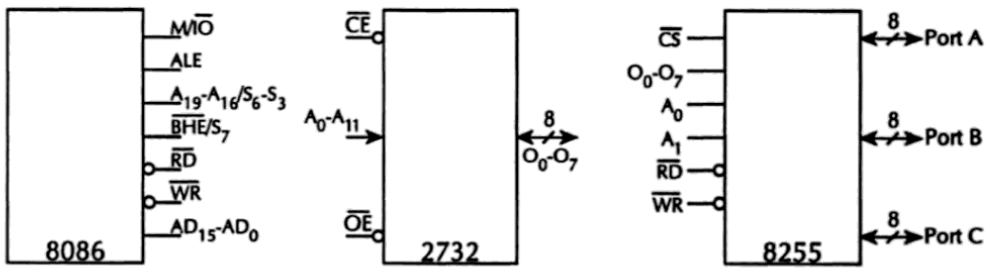
$$Y = \sum_{i=1}^{N} \frac{Xi^2}{N}$$

where Xis are signed 8-bit integers and N = 100. The numbers are stored in consecutive locations. Assume SI points to Xis and SP, DS, SS are already initialized.

3.20 Write an 8086 assembly program to move a block of data bytes of length 100_{10} from the source block starting at location 2000H in ES = 1000H to the destination block starting at location 3000H in the same extra segment.

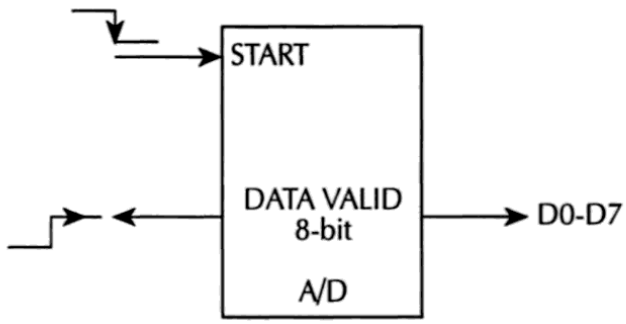
3.21 Write an 8086 assembly program to logically shift a 128-bit number stored in location starting at 4000H in DS twice to the right. Store the result in memory location starting at 5000H in the data segment. Assume DS is initialized.

3.22 Connect one 2732, and one 8255 to an 8086 to obtain even 2732 locations and odd addresses for the 8255's Port A, Port B, Port C, and the Control register. Show only the connections for the pins shown in the figure below. Assume all unused address lines to be zeros. Use latches and gates as required.



3.23 Determine the number of 2732 4K × 8 EPROMs and 6116 2K × 8 static RAMs to provide a 4K ×16 EPROM and a 2K ×16 word RAM. Draw a neat schematic connecting these chips to the 8086 and determine the map.

3.24 Assume the memory and I/O maps of Table 3.2. Interface the following A/D to the 8086/2716/6116/8255 of Figure 3.14:



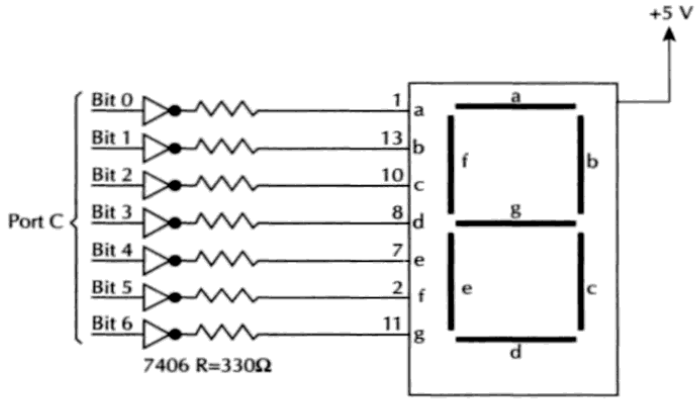
Write an 8086 assembly program to input the A/D converter and turn an LED ON connected to bit 5 of port A of the even 8255 if the number read from the A/D is odd; otherwise turn the LED OFF. Assume that the LED is turned ON by a HIGH and turned OFF by a LOW.

3.25 Repeat problem 3.24 using the 8086 INTR interrupt.

3.26 Write an 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.

3.27 Assume an 8086/8255 based system. Write an 8086 assembly language program to input 16-bit data via Port B and Port C, and then divide this by the 8-bit input data at Port A. Assume all numbers to be signed.

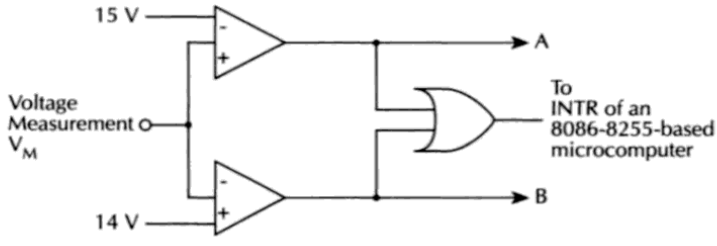
3.28 An 8086/8255 based microcomputer is required to drive a common anode seven-segment display connected to Port C as follows:



Write an 8086 assembly language program to display a single hexadecimal digit (0 to F) from offset 3000H at DS = 1000H. Use a look-up-table.

3.29 Assume an 8086/8255 based microcomputer. Suppose that four switches are connected to bits 0 through 3 of Port A, an LED is connected at bit 4 of Port B and another LED is connected at bit 2 of Port C. If the number of low switches is even, turn the Port B LED "ON" and Port C LED "OFF". If the number of low switches is ODD turn Port B LED "OFF" and Port C LED "ON". Write an 8086 assembly language program to accomplish the above.

3.30



In the above figure, if $V_M > 15V$, turn an LED "ON" connected at bit 2 of Port C. On the other hand if $V_M < 14V$, turn the LED "OFF". Use registers and memory locations of your choice.

Draw a block diagram showing the microcomputer and the connections of the figure to its ports. Also, write 8086 assembly language programs to accomplish the above using:

a) Polled I/O by inputting one or more outputs in the figure.
b) INTR
c) NMI

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 covers the two enhanced versions of the 8086 microprocessor: Intel 80186 and 80286. 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 is fabricated using HMOS technology and 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 three different clock speeds: 8 MHz, 10 MHz, 12.5 MHz, and other frequencies. The 80C186 and 80C188 are the Low-Power (HCMOS) versions of the 80186 and 80188 respectively. The 80C186 and the 80186 can be operated at the same frequencies while the 80C188, like the 80188 can be operated at 8- or 10-MHz. The 80186 can directly address one megabyte of memory. It contains the 8086 microprocessor and several additional functional units. The major on-chip circuits include a clock generator, two independent DMA channels, 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 8086's MN/$\overline{\text{MX}}$ 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 8288 for generating the maximum mode type signals. Like the 8086, the 80186 fetches the first instruction from physical address FFFF0H upon hardware reset.

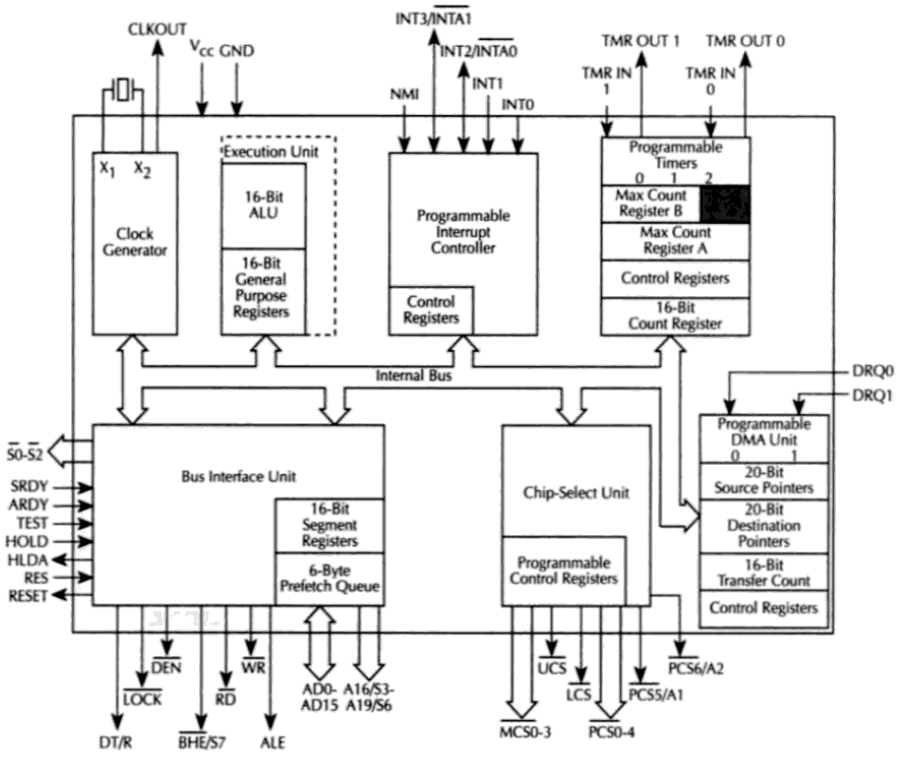


FIGURE 4.1 80186 functional block diagram.

Figure 4.1 shows the 80186 functional block diagram. The DMA unit provides two DMA request input signals, DRQ0 and DRQ1. One of these input signals can be used by external devices such as disk controller to request a data transfer between the memory and disk via direct memory access technique. Each 80186 DMA channel contains two 20-bit registers and a 16-bit counter. One of the 20-bit registers holds the destination address of the DMA transfers while the 16-bit counter stores the number of words or bytes to be transferred. DMA transfers can take place between memory and I/O or from memory to memory or from I/O to I/O. The DMA channels may be programmed such that one channel has priority over the other. DMA transfers 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 namely counter 0, 1, and 2. Counters 0 and 1 can be programmed to count external events. The inputs and outputs of these two counters are available on the 80186 pins. The third timer, counter 2, is not connected to any external pins. This timer only counts the 80186 clock cycles. Counter 2 is decremented every four 80186 clocks. One can connect the output of counter 2 to a DMA unit or to an interrupt input or to the input of counter 1 and/or 0 by setting or clearing the appropriate bits in a control word. Counter 2 can, therefore, be used to interrupt the 80186 after a programmed amount of time or to provide a pulse to the DMA unit after a specific amount of time.

The priorities of the four interrupt pins, INT0, INT1, INT2/ INTA0 and INT3/ INTA1 are programmable. If these four interrupt inputs are programmed in their internal mode, then the activation of one of them by an external signal will cause the 80186 to push the return address on the stack and vector directly to the interrupt address vector for that interrupt. The INT2/ INTA0 and INT3/ INTA1 pins have dual functions. They can be programmed as interrupt inputs or as interrupt acknowledge output signals (INTA0 for INT0 and INTA1 for INT1). These interrupt pins can be used to connect the 80186 to an external priority interrupt controller such as the 8259A. For example, suppose that the interrupt request line from an

external 8259A is connected to the 80186 INT1 input pin and the 80186 INT3/ INTA1 pin is connected to the 8259A interrupt acknowledge input pin. When the 8259A receives an interrupt request, it activates the 80186 INT1 pin. In response, the 80186 sends the interrupt acknowledge signal via its INT3/ INTA1 pin. The 8259A then places the desired interrupt type code on the 80186 data bus. The 80186 obtains the CS and IP values based on the type code and branches to the service routine.

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 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 nonmaskable interrupt.

In the master mode, the interrupt controller provides three modes of operation. These are fully nested mode, cascade mode, and special fully 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.

Upon acceptance of an interrupt (hardware, INT instructions, or instruction exceptions such as divide by 0), the 80186 pushes CS, IP, and the 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 type. This type identifies the appropriate table entry. Nonmaskable interrupts use an internally supplied type, while the types for maskable interrupts are provided by the user via external hardware. The types 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 selects (LCS, UCS, and MCS0 — 3 pins) and seven I/O or peripheral chip selects (PCS0-4, PCS5/ A1, and PCS6/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 FFFFFH. The four middle chip select pins (MCS0-3) are activated low by the 80186 when it accesses an address in the mid range. Both the starting address and the size of the four blocks can be specified via the control word. The specified size of blocks can be from 2K to 128K. The memory areas assigned to different chip selects must not overlap; otherwise, two chip selects will be asserted and bus contention will occur. The built-in decoder allows the 80186 to select large memory blocks. For peripheral chip selects, a base address can be programmed via a control word. The 80186 sends low on the PCS0 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 PCS0 is selected. Like the 8086, memory for the 80186 is set up as odd (BHE = 0) and even (A0 = 0) 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 based indexed with displacement modes.

Typical data types provided by the 80186 include signed integer, ordinal (unsigned binary number), pointer, string, ASCII, 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

PUSHA	—	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 register.

Logical

SHIFT/ROTATE destination, immediate data or CL shifts/rotates register or memory contents by the number of times specified in immediate data or by the contents of CL.

String Instructions

INSB or INSW	—	Input string byte or string word
OUTSB or OUTSW	—	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 register, 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 source register or memory location and places the result in a general-purpose destination register. As an example, IMUL DX, CX, –3 multiplies the contents of CX by –3 and places the lower 16-bit result in DX. Note that the immediate 8-bit data of –3 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 or CL. Shift count can be specified by immediate data (one) or in CL up to a maximum of 32_{10}.
- INSB DX or INSW DX respectively inputs a byte or 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 for byte and INSW for word are used. A typical example of inputting 50 bytes of I/O data via a port into a memory location is given below (assume ES is already initialized):

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

- OUTSB DX or OUTSW DX respectively provides outputting to a port addressed by DX from a source string in DS with offset in SI.
- 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 = 0 for SUB1, 1 for SUB2, and 2 for SUB3. ENTER can be used to allocate temporary stack space for local variables for each subroutine.

In the second operand, if imm8 = 0, the ENTER instruction pushes the frame printer 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 each subroutine (usually before the RET instruction). 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 accessed.

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 element 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 content of SI must be greater than or equal to the content of the memory location with offset ADDR and less than or equal to the contents of the memory location with offset 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.

The BOUND instruction is normally placed following the computation of an offset value to ensure that the limits of the array boundaries are not violated. This permits checking whether or not the offset of an array being accessed is within the boundaries when the based addressing mode is used to access an element in the array. For example, the instruction segment shown below will allow accessing of an array with base address in BX and array length of 50_{10} bytes:

```
MOV     DS:ADDR, 1000;
MOV     DS:ADDR+2, 1049;
BOUND   BX, ADDR;
MOV     CL, [BX];
```

In the above, it is assumed that the offset of the lowest array element is 1000. With an array length of 50, the offset of the highest array element is 1049. It is assumed that BX contains the

offset of the array element currently being worked on. The BOUND instruction checks whether the contents of BX is between 1000 and 1049. If it is, the MOV instruction accesses the desired array element into CL; otherwise type 5 interrupt is generated.

The 80186/80188 is used in embedded control. In these applications, the microcomputer performs a dedicated control function. Embedded control applications are divided into two types. These are event control and data control.

In embedded control applications involving event control, the microprocessor initiates a timed sequence of events. An example of such an application is the industrial process control.

In embedded control applications involving data control, the microprocessor transfers volumes of data to be processed from secondary memory such as disk to main memory.

The 80186/80188 is, therefore, highly integrated to satisfy the requirements of data control applications. The 80186/80188 is also provided with added features such as string I/O instructions and DMA channels to better handle fast movement of data.

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 multitasking 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 several different clock speeds. These are 8 MHz (80286-4), 10 MHz (80286-6), 12.5 MHz, 16.67 MHz, and 20 MHz (80286).

The 80286 has two modes of operations. 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 8086/8088/80186/80188. 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. Note that the protected virtual address mode is not used by PCDOS.

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 68-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

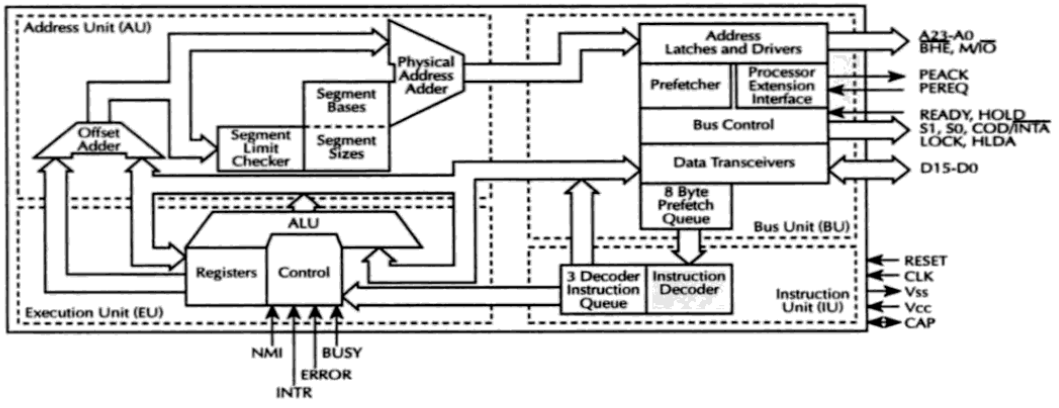


FIGURE 4.2 80286 internal block diagram.

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 prefetched instructions and places them in a queue for execution by the execution unit.

The Execution Unit (EU) executes instructions from the IU sequentially. The EU contains a 16-bit ALU, an 8086 flag register, general-purpose registers, pointer registers, index registers, and one 16-bit additional register called the machine status word (MSW) register. The lower four bits of the MSW are used. One bit places the 80286 into PVAM mode while the other three bits control the processor extension (coprocessor) interface. The LMSW and SMSW instructions can load and store MSW in real address mode.

The Address Unit (AU) calculates a 20-bit physical address based on the 16-bit contents of a segment register and a 16-bit offset just like the 8086. In this mode, the 80286 addresses one megabyte of physical memory. The 80286 has 24 address pins. However, in the real address mode, pins A23-A20 are ignored and A19-A0 pins are used. In the protected virtual address mode (PVAM), the AU operates as a memory management unit (MMU) and utilizes all 24 address lines to provide 16 megabytes of physical memory. The BU outputs memory or I/O addresses to devices connected to the 80286 after receiving them from the AU.

The 80286 does not have on-chip clock generator circuitry. Therefore, an external 82284 chip is required. The 80286 has a single CLK pin for a single-phase clock input. The 80286 divides its input clock by 2 internally and then provides the processor clock. The 82284 also provides the 80286 RESET and READY signals.

The 80286 external memory is configured as odd ($\overline{BHE}$ = 0) and even (A0 = 0) memory banks just like the 8086. The 80286 operates in a mode similar to the 8086 maximum mode. Some of the 80286 pins such as M/IO, S0, S1, HOLD, HLDA, READY, and LOCK have identical functions as the 8086. Two external interrupt pins (NMI and INTR) are provided. The nonmaskable interrupt NMI is serviced in the same way as the 8086. The INTR and COD/$\overline{INTA}$ are used together to provide an interrupt-type code on the data bus via external hardware. Note that the 80286 $\overline{INTA}$ is multiplexed with another function called the COD (code). This pin distinguishes instruction fetch cycles from memory data read cycles. Also, it distinguishes interrupt acknowledge cycles from I/O cycles. M/$\overline{IO}$ = HIGH and COD/$\overline{INTA}$ = HIGH define instruction fetch cycle. On the other hand, M/IO = LOW and COD/$\overline{INTA}$ = LOW specify interrupt acknowledge cycle.

A new pin called the CAP is provided on the chip. The 80286 MOS substrate must be applied with a negative voltage for maximum speed. The negative voltage is obtained from the +5V. An external capacitor must be connected to the CAP pin for filtering this bias voltage.

Four pins are provided to interface the 80286 with a coprocessor. These are PEREQ, PEACK, BUSY, and ERROR. The PEREQ (Processor Extension Request) input pin can be activated by the coprocessor to tell the 80286 to perform data transfer to or from memory for it. When the 80286 is ready, it activates the PEACK (Processor Extension Acknowledge) signal to inform the coprocessor of the start of the transfer. The 80286 BUSY signal input, when activated LOW by the coprocessor, stops 80286 program execution on WAIT and some ESC instructions until BUSY is HIGH. If a coprocessor finds some error during processing, it will activate the 80286 ERROR input pin. This will generate an interrupt. A service routine for this interrupt can be written to provide an indication to inform the user of the error.

Upon hardware reset, the 80286 operates in real (physical) address mode and starts executing programs at physical address FFFF0H like the 8086. In this mode, 20-bit physical addresses are generated by adding a 16-bit offset to the shifted (4 times to the left) segment register just like the 8086. Note that after hardware reset, the 80286 sets A23-A20 to all ones, CS = F000H, DS = 0000H, ES = 0000H, and SS = 0000H.

The 80286 on-chip MMU is disabled in the real address mode. In this mode, the 80286 acts functionally as a high-performance 8086. This mode averages 2 $^1/_2$ times the performance of an 8086 running at the same clock frequency. All instructions of the 8086, 80186, plus a few

more such as LMSW (Load Machine Status Word) and SMSW (Store Machine Status Word) are available with the 80286 in the real address mode. In this mode, the 80286 supports only the 8086 data types and can directly execute 8086 machine code programs with minor modifications. When interfaced with an 80287 floating point coprocessor, the 80286 supports 8087 floating point data types also. Upon hardware reset, the 80286 operates in real address mode unless the user sets a bit in the Machine States Word (MSW) register by using the LMSW instruction to change the 80286 mode to Protected Virtual Address Mode (PVAM). Note that before changing to PVAM, descriptor tables must be in memory. The 80286, in real address mode, can run 8086 or 8088 software. Now, to change the 80286 mode from real address to PVAM, the user should read the contents of MSW, set just the Protection Enable (PE) bit to 1 without changing the other bits, and then load the new data into the MSW. 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. Note that the real address mode is normally used to initialize peripheral devices, transfer the main portion of the operating system from disk to main memory, initialize some registers, enable interrupts and place the 80286 into PVAM.

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 logical addresses used by the user to physical addresses. Once the 80286 is in PVAM, the only way to get back to the real mode is via hardware reset. This is intentionally done so that a malicious programmer cannot switch the mode from PVAM to real mode and thus the protection feature in PVAM is maintained.

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, based index, and based indexed with displacement modes. The new 80286 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
LGDT	Load global descriptor table register from memory
SGDT	Store global descriptor table register into memory
LIDT	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)

STR	Store selector from task register in register or memory
LMSW	Load MSW register from register or memory
SMSW	Store MSW register in register or memory
LAR	Load access rights byte of descriptor into register or memory
LSL	Load segment limit from descriptor into register or memory
ARPL	Adjust register privilege Level of selector
VERR	Determine if segment addressed by a selector is readable
VERW	Determine if segment pointed to the selector is writable

Next, the 80286 will be considered from an operating systems point of view. In this context, the memory management capabilities protection and task switching features of the 80286 will be covered. Using these on-chip hardware features, a multitasking operating system can be implemented in the 80286-based microcomputer system.

The 80286 memory management features provide the operating system with the following capabilities:

- An operating system which can separate tasks from each other. This avoids an 80286 system failure due to task errors.
- As tasks begin and end, the operating system can optimize memory usage by moving them around, a process referred to as dynamic relocation. This is because a program can be executed in different parts of memory without being reassembled or recompiled.
- Use of virtual memory becomes easy. Note that virtual memory is a method for executing programs larger than the main memory by automatically transferring parts of the programs between main memory and disk.
- Controlled sharing of information between tasks.

The 80286 protection features allow:

- An operating system to protect itself from malicious users in a multiuser environment
- Critical subsystems such as disk I/O from being destroyed by program bugs under development

The 80286 task switching provides:

- Fast task switching due to 80286's hardware implementation for accomplishing this feature. This permits the 80286 to spend more time on task execution than switching. Real-time systems can thus be 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.

4.1.2.a 80286 Memory Management

The 80286 logical segments may be called virtual segments because all of them may not be resident in physical memory at the same time. A 80286 logical segment can be of any length from 1 byte to 64K bytes. During creation of each segment, a size or limit value is defined. This makes it easier for the 80286 to determine if a memory access is within bounds of a segment. The segments presently used by a task are stored in physical memory. In PVAM, all 24 address pins are used and therefore, the directly addressable (physical) memory is 16 megabytes. While writing 80286 programs in PVAM, one can refer to the current segment by the assigned names. For example, if a segment called JOHN is to be used as the current data segment, then the following instructions should be used to load JOHN into DS:

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

When the 80286 executes the above instruction segment, it will use JOHN as the current data segment. If the segment JOHN is not currently resident in Physical memory, the 80286 will generate an interrupt. A service routine is written to load this segment from disk to physical memory and then resume execution of the main program.

In PVAM, when a program is assembled, a descriptor (8-byte wide) is assigned to each segment. The descriptor contains information such as segment length in bytes, 24-bit base address where the segment is located in physical memory and the privilege level.

The descriptors are held in tables in main memory and are read into the 80286 as needed. There are two main types of descriptor tables. These are the global and the local descriptor tables. A system can contain only one global descriptor table. The global descriptor table includes information such as the descriptors for the operating system segments and the descriptors for segments which are accessed by user tasks. A local descriptor is created in the system for each task. All tasks share a global descriptor table with the memory areas specified by its descriptors. Also, each task contains its own local descriptor table with the memory areas specified by its descriptors.

In PVAM, all programs are written using segments. Each segment is assigned with an 8-byte descriptor which includes the length, starting address, and access rights for that segment. Segment descriptors for programs are stored in memory either in the global descriptor table or in a local descriptor table.

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 address mode, 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. Also, A20-A23 pins are only used at reset for CS and are zero otherwise.

In the Protected Virtual Address Mode (PVAM or virtual mode for short), the 32-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 example, index value K refers to the descriptor K. 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 tells the 80286 to select one of two tables: Global Descriptor Table (GDT) and Local Descriptor Table (LDT). All tasks in the 80286 share a common single table called the GDT, while each task has its own LDT. Therefore, the 24-bit base addresses required in physical address calculation for segments to be shared by all tasks are stored in the GDT and the base addresses for segments dedicated to a particular task are stored in its LDT. When TI = 0, the GDT is used as the look-up table, and when TI = 1, the LDT is used as the look-up table. The 2-bit RPL is used by the operating system 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.

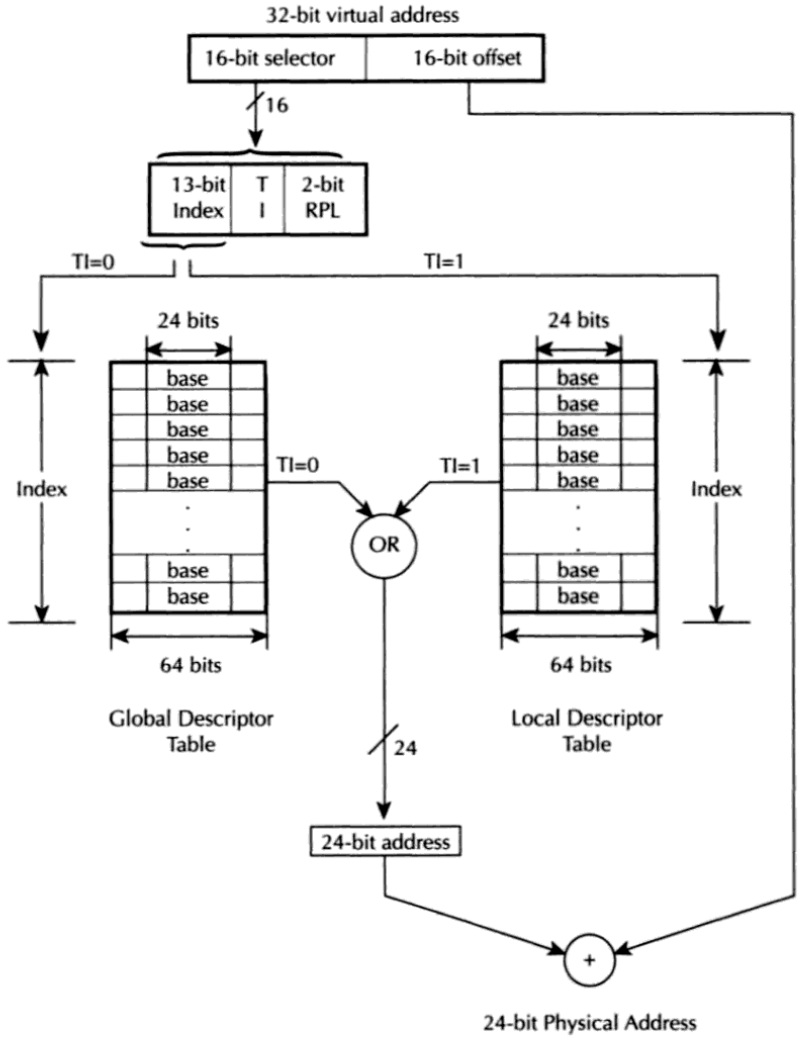


FIGURE 4.3 80286 virtual address translation.

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.

The LDT register stores a 16-bit selector for a descriptor in the GDT which defines the location of the present LDT. The 80286 task switch operation is invoked by executing an inter-segment JMP or CALL instruction which refers to a task state segment (TSS) or task gate descriptor in the GDT or LDT. Each task must have a TSS associated with it. The current TSS is identified by a special register in the 80286 called the Task Register (TR). The TR register makes task switching automatic and very fast. For example, the 80286 local address space can be modified during task switching by updating the LDT register. Figure 4.5 shows flowchart for accessing memory.

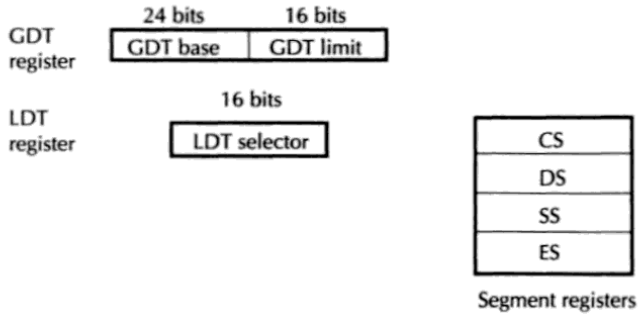


FIGURE 4.4 80286 address translation registers.

The 80286 contains a number of special registers called shadow registers which 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^{13} (13-bit index) descriptors and each segment can specify a segment of up to 2^{16} bytes. Therefore, a task can have its own LDT address space of up to $2^{13} \times 2^{16} = 2^{29}$ bytes and can share 2^{29} bytes (GDT) 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 registers 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.b Protection

In PVAM, the 80286 has built-in features for the following protection schemes:

1. Protecting system software such as the operating system from user programs.
2. Protecting one user task from another.
3. Protecting portions of memory from accidental access.

The 80286 protection mechanism is implemented by using the contents of the descriptors. Any access to memory is validated by checking the information in segment descriptors. The 80286 generates an interrupt if the memory access is invalid.

The 80286 provides protection mechanism for supporting multitasking and virtual memory features. The 80286 includes certain basic protection features such as segment limit and segment usage checking. These basic 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

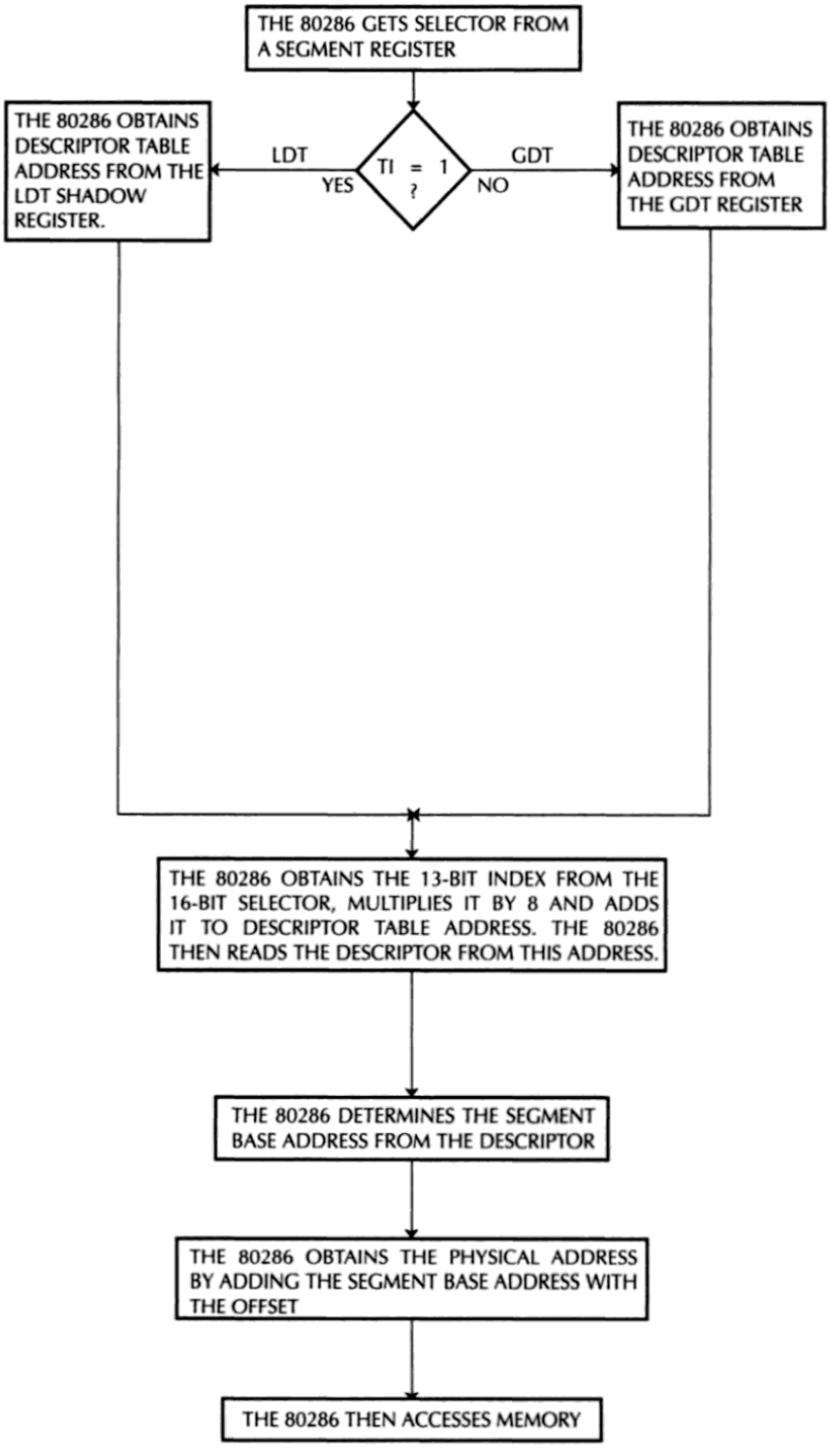
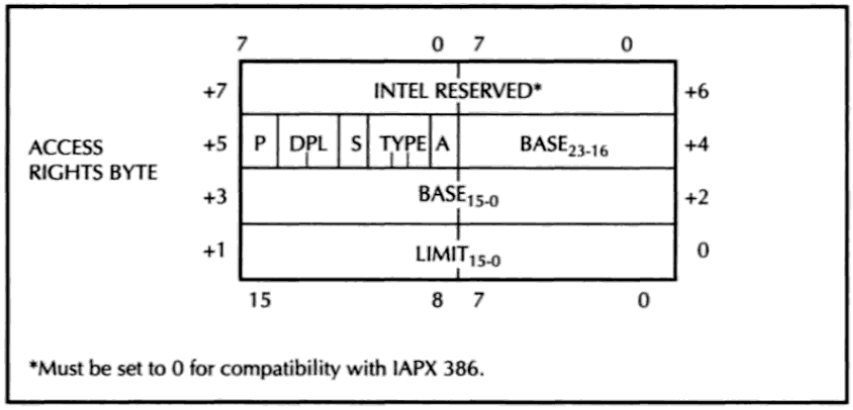


FIGURE 4.5 Flowchart showing steps for accessing memory.

Code or Data Segment Descriptor



Access Rights Byte Definition

Bit Position	Name	Function	
7	Present (P)	P = 1 Segment is mapped into physical Memory. P = 0 No mapping to physical memory eexists, base and limit are not used.	
6-5	Decriptor Privilege Level (DPL)	Segment privilege attribute used in privilege test.	
4	Segment Descriptor (S)	S = 1 Code or data (includes stacks) segment descriptor S = 0 System Segment Descriptor or Gate Descriptor	
3	Executable (E)	E = 0 Data segment descriptor type is:	If Data Segment (S =1, E = 0)
2	Expansion Direction (ED)	ED = 0 Expand up segment, offsets must be ≤ limit. ED = 1 Expand down segment, offsets must be > limit.	
1	Writeable (W)	W = 0 Data segment may not be written into. W = 1 Data segment may be written into.	
3	Executable (E)	E = 1 Code Segment Descriptor type is:	If Code Segment (S =1, E = 0)
2	Conforming (C)	C = 1 Code segment may only be executed when CPL ≥ DPL and CPL remains unchanged.	
1	Readable (R)	R = 0 Code segment may not be read. R = 1 Code segment may be read.	
0	Accessed (A)	A = 0 Segment has not been accessed. A = 1 Segment selector has been loaded into segment register or used by selector test instructions.	

FIGURE 4.6 Code and data segment descriptor contents.

at specific valid entry points. In order to protect the operating system, the 80286 does not allow accessing a higher privilege entry point without its access gate.

The 80286 provides some instructions controlled by its IOPL (Input/Output Privilege Level) feature to protect shared system resources. The I/O instructions are permitted at the highest 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 systems'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 system software such as 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 a 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 all the code of a dedicated 80286. In this case, all load/store and I/O instructions 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 (highest 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 a large application. The descriptor's 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 right 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 and can be defined as conforming segments. 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 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 lower than or equal to the CPL. This rule allows protection of privileged data segments from unprivileged codes.

- The 80286 is provided with a rule which pertains to accessing data segments. The 80286 can access data segments of equal or lower 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 by using call gates, 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 levels 2 and 3 can be accessed by the 80286; code segments of levels 0 or 1 cannot be accessed directly. However, lower or equal privilege level accesses can be done directly. Also, higher privilege level accesses can be controlled by special descriptor table entries known 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 another 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 intersegment CALL references a call gate, the 80286 redirects control to the destination address defined within the gate. The 32-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 is 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 both in virtual and physical memory spaces. The 80286 provides a multitasking feature via its virtual memory capabilities. As mentioned before, 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 most of 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 16-bit selectors (each task has a unique selector) that identify 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.

The 80286 provides protection for portions of memory from accidental access in the following ways. When a segment selector is to be loaded into a segment register, the 80286 automatically verifies whether the descriptor table indexed by the selector contains a valid descriptor for that selector. An interrupt is automatically generated by the 80286 if a valid descriptor is not present. However, if the descriptor is valid, the shadow registers are loaded with the base, limit, and access rights byte of the descriptor. The 80286 then verifies whether the segment for that descriptor is resident in physical memory. An interrupt is generated by the 80286 if it is not present (as indicated by the P-bit of the access rights byte of the descriptor). An interrupt service routine can be written to move the desired segment into physical memory and return execution to the main program. The 80286 also verifies whether the segment descriptor is of the right type to be loaded into the appropriate segment register. For example, the descriptor for a read-only data segment cannot be moved into a stack segment register since the stack is a read/write memory. After a segment selector and descriptor are loaded into a segment register, checks are made each time an address in the actual segment is accessed. For example, an attempt to write to a read-only data segment will generate an error. Also, the limit value in the segment descriptor is used to ensure that an address generated by program instructions is within the limit specified for the segment.

Example 4.1

Discuss the 80286's performance impact on memory management while executing the following program:

```
        MOV DS, Segmentselector    ;  Load data selector
        MOV BX, Displ              ;  Load offset
        MOV CX, Count              ;  Load loop count
BEGIN   MOV DX, data               ;  Move 16-bit data to
                                      DX
        CMP DX, WORDPTR[BX]        ;  Find match
        JZ DONE                    ;  If match
        JMP BEGIN                  ;  found, stop
                                   ;  else compare
DONE    HLT
```

Solution

The 80286 memory management capabilities are only utilized when loading the selector value (first instruction in the above program). By loading the selector value into DS, the 80286 chooses a descriptor from a descriptor table and then automatically determines the physical address of that segment using the descriptor. Thus, by executing the first instruction MOV DS, segmentselector, the 80286 automatically determines the segment's physical address (transparent to the user).

After determining the segment's physical address, the 80286 executes all other instructions that follow. The 80286 reads data from the data segment every time it goes through the BEGIN loop. The 80286 does not refer to the descriptor table because of its on-chip cache.

Like any other memory management system, the 80286 will have some overhead for performing the virtual address to the physical address translation. This is transparent to the user's application programs and is automatically carried out by the 80286. The memory management overhead (virtual to physical address translation) is minimized due to on-chip cache. One of the main characteristics of on-chip MMU is that after the descriptor is read into the on-chip cache memory, no overhead occurs. Therefore, the overhead is kept at a minimum, thus providing good performance.

4.1.2.c 80286 Exceptions

In the real address mode, the 80286 exception mechanism is similar to the 8086 except a few more exceptions such as the 'invalid opcode exception' is included in the 80286. External interrupts such as the maskable (INTR) and nonmaskable (NMI) interrupts in the 80286 are serviced in the real address mode by using interrupt vector table with 256 vectors similar to the 8086.

In the protected mode, the 80286 detects exceptions essential to its protection model, and its support for multitasking and virtual memory. Some examples of the non-real mode exceptions include 'exception for code, data, or extra segment not present' and 'Privilege violation'. The protected mode pointers are actually gates, since gates in protected mode serve as a redirection mechanism. The interrupt table can contain task gates, interrupt gates, and trap gates. The interrupt table, in protected mode, is known as the IDT (Interrupt Descriptor Table). A special register called the IDTR (Interrupt Descriptor Table register) is used by the 80286 to locate the interrupt table both in real and protected modes. The IDTR is typically initialized by the system programmer once, to point at the desired area of physical memory.

4.2 INTEL 80386

In 1985, Intel introduced its first 32-bit microprocessor, the 80386DX. It initially ran at a clock frequency of 16 MHz and contained 275,000 transistors. In 1988, Intel introduced the 80386SX which was a 16-bit external data bus version of the 80386DX (32-bit data bus). The 80386DX is a full 32-bit microprocessor and is available in 16-, 20-, 25- and 33-Mhz speed versions. It can directly address a maximum of 4 gigabytes of memory.

The 80386SX, on the other hand, can operate at 16- and 20-Mhz. The 80386SL is similar to the 80386SX with 16-bit data bus and can operate at 20- and 25-Mhz. The 80386SX and 80386SL can directly address up to 16 megabytes and 32 megabytes of memory respectively. Compaq was the first major OEM (Original Equipment Manufacturer) to use the 80386DX in 1986 in its PC. The 80386SL, on the other hand, is used in notebook computers with built-in power management options.

The 80386DX will be covered in detail in this section. The 80386 family of microprocessors will be referred to as the 80386 in the following.

The 80386 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 generates a 32-bit physical address. 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 protec-

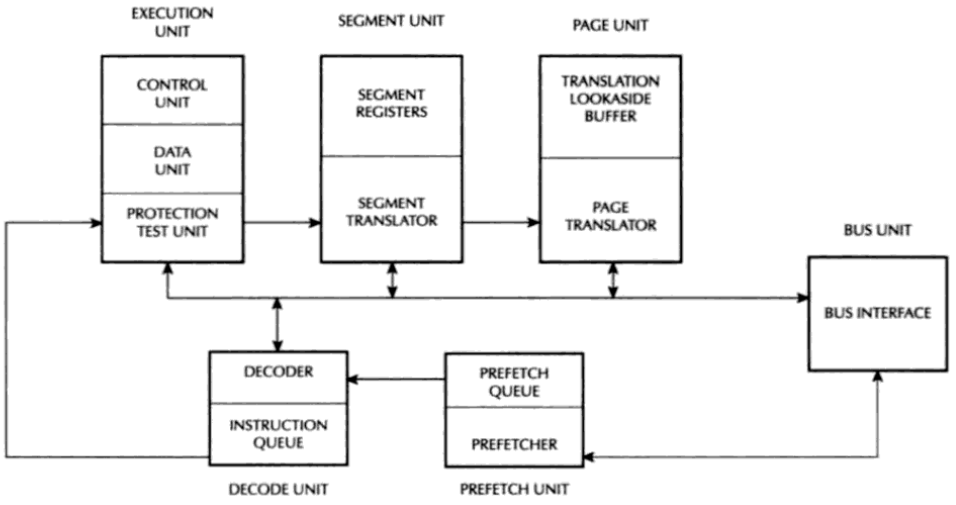


FIGURE 4.7 80386 functional units.

tion 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), load the state of another 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 8086. 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 in 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
- Decode unit
- Execution unit
- Segmentation unit
- Paging unit

The bus interface unit connects 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 decoding 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 a 32-bit 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 can be interfaced to the 80386 to extend the 80386 instruction set 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 82384 clock generator provides system clock and reset signals. The 82384 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 internal frequency of the 80386 is 1/2 the frequency of CLK2.

The 8259A interrupt controller provides interrupt control and management functions. Interrupts from as many as eight external sources are accepted by one 8259A and up to 64 interrupt requests can be handled by connecting several 8259A chips. The 8259A manages priorities between several interrupts, then interrupts the 80386 and sends a code to the 80386 to identify the source of the interrupt.

The 82258 Advanced DMA (ADMA) controller performs DMA transfers between the main memory and the I/O device such as a hard disk or floppy disk without involving the 80386. It provides four channels and all signals necessary to perform DMA transfers.

The 80386 has three processing modes: protected mode, real-address mode, and virtual 8086 mode.

Protected mode is the normal 32-bit application of the 80386. All instructions and features of the 80386 are available in this mode.

Real-address mode (also known as the "real mode") is the mode of operation of the processor upon hardware RESET. This mode appears to programmers as a fast 8086 with a few new instructions. This mode is utilized by most applications for initialization purposes only.

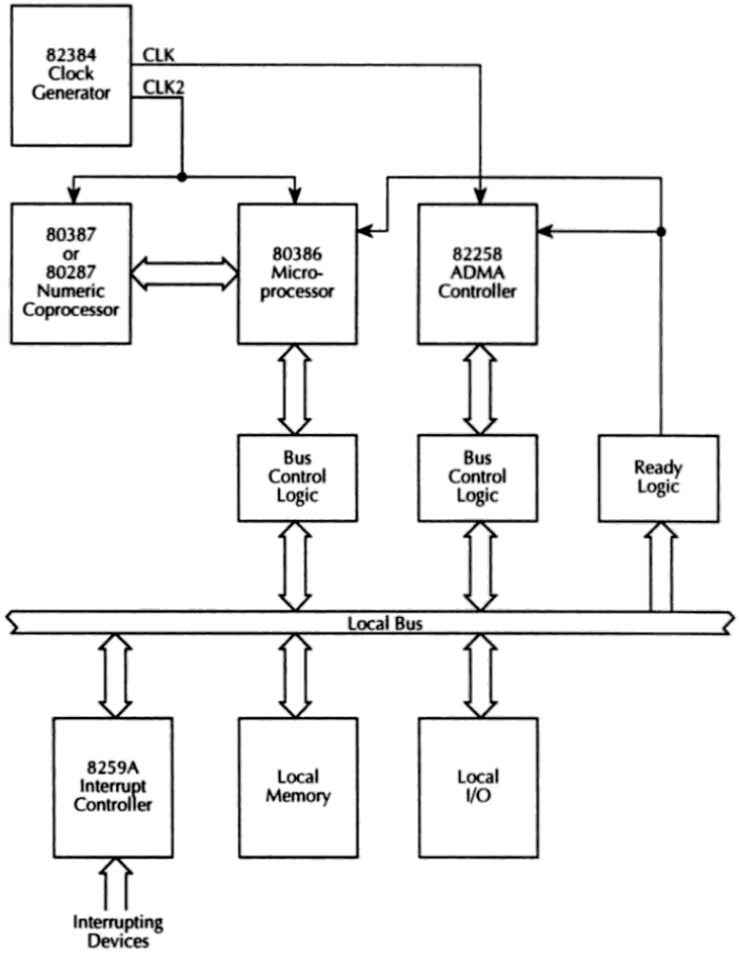
Virtual 8086 mode (also called V86 mode) is a mode in which the 80386 can go back and forth repeatedly between V86 mode and protected mode at a fast speed. The 80386, when entering into the V86 mode, can execute an 8086 program. The processor can then leave V86 mode and enter protected mode to execute an 80386 program.

As mentioned before, the 80386 enters real address mode upon hardware reset. In this mode, the Protection Enable (PE) bit in a control register called the Control Register 0 (CR0) is cleared to zero. Setting the PE bit in CR0 places the 80386 in protected mode. When in protected mode, setting the VM (Virtual Machine) bit in the flag register (called the EFLAGS register) will place the 80386 in V86 mode. Details of these modes are discussed later.

4.2.1 Basic 80386 Programming Model

The 80386 basic programming model includes the following aspects:

a) Memory organization and segmentation
b) Data types



Component	Description
80386 Microprocessor	32-bit high-performance microprocessor with on-chip memory management and protection; no on-chip cache
80287 or 80387 Numeric Coprocessor	Performs numeric instruction in parallel with 80386; expands instruction set
82384 Clock Generator	Generates system clock and RESET signal
8259A Programmable Interrupt Controller	Provides interrupt control and management
82258 Advanced DMA	Performs direct memory controller access (DMA)

FIGURE 4.8 80386 system block diagram.

c) Registers
d) Addressing modes

I/O is not included as part of the basic programming model. This is because systems designers may select to use I/O instructions for application programs or may select to reserve

them for the operating system. Therefore, 80386 I/O capabilities will be covered during the discussion of systems programming.

4.2.1.a 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 a 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 called the logical address space and can be up to 2^{46} bytes (64 tetrabytes). 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.

4.2.1.b Data Types

Data types can be byte (8-bit), word (16-bit with low byte address 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 speed 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 data 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 $2^{32}-1$ bits), and packed/unpacked BCD and ASCII-type data. 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.c 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

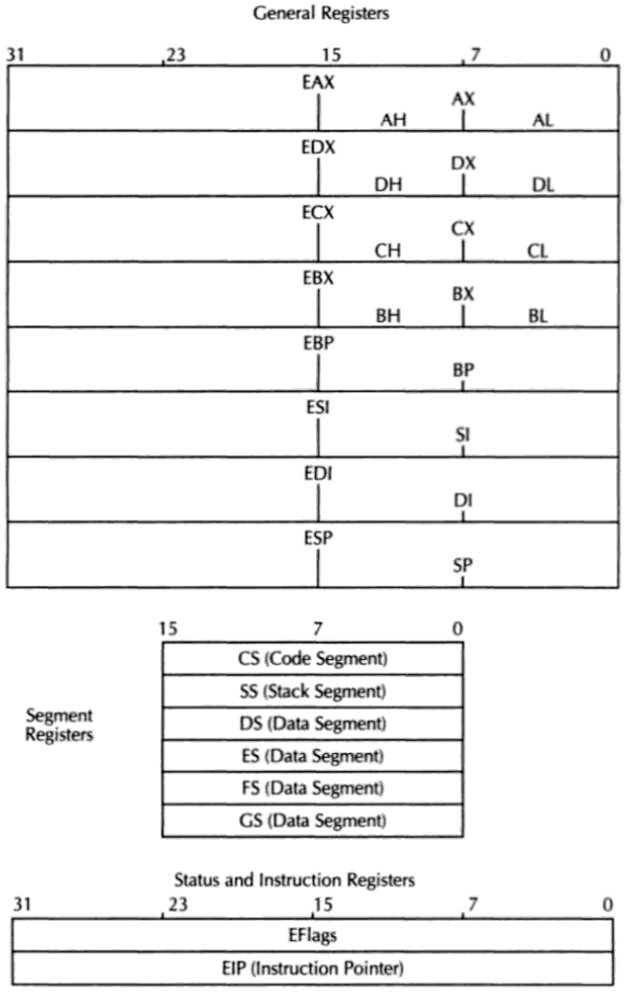


FIGURE 4.9 80386 applications register set.

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 a 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

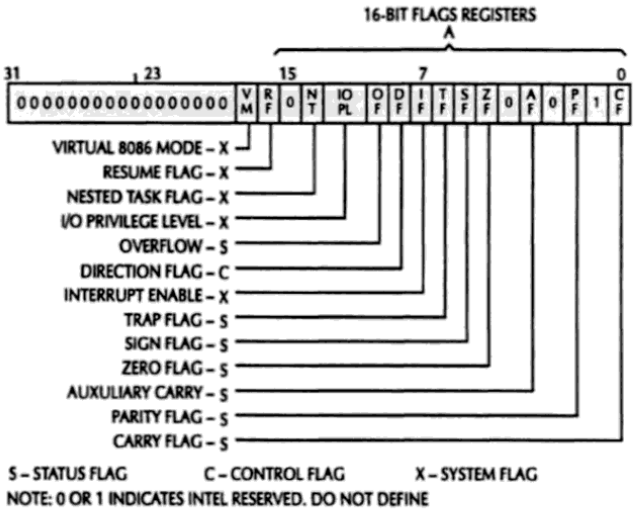


FIGURE 4.10 EFLAGS register.

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 80286/80186/80286. 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 as in the 8086/80186/80286. The control flag DF is used by strings as in 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:

- IOPL (I/O Privilege Level) — This is a 2-bit field and supports the 80386 protection feature. The IOPL field defines the privilege level needed to execute I/O instructions. If the present privilege level is less than or equal to IOPL (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 (EIP) 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 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 other 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 $5167812F_{16}$ 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 the 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 additional clocks.

As shown in Figure 4.11, the Effective Address (EA) of an operand is computed according to the following formula:

$$EA = \text{Base reg} + (\text{Index Reg} * \text{Scaling}) + \text{Displacement}$$

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 WORDPTR [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, [ESI] [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 * 4] [EDX], EAX.

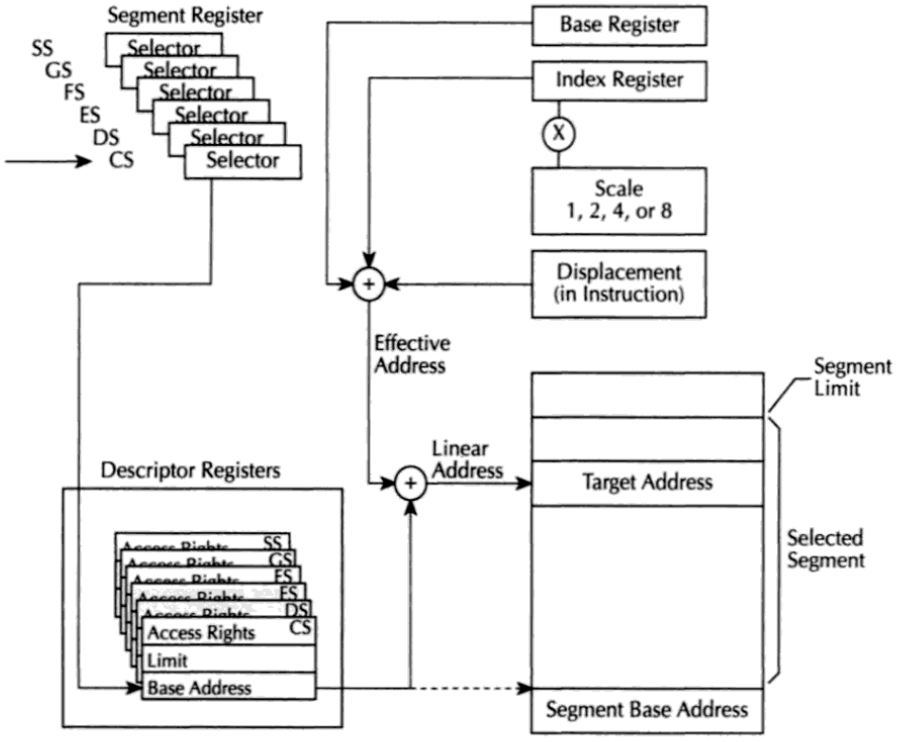


FIGURE 4.11 Addressing mode calculations.

8. Based Index Mode with Displacement — The operand's effective address is obtained by adding the contents of a base register and an index register with a displacement. An example is MOV [EBX] [EBP + 0F24782AH], 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:

	16-bit addressing	32-bit addressing
Base Register	BX, BP	Any 32-bit general-purpose register
Index Register	SI, DI	Any 32-bit general-purpose register except ESP
Scale Factor	None	1, 2, 4, 8
Displacement	0, 8, 16 bits	0, 8, 32 bits

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
String
Logical
Bit manipulation
Program Control
High-level language
Protection Model
Processor control

These instructions are listed in Table 4.1.

TABLE 4.1 80386 Instructions

	Data Transfer Instructions
General purpose	
MOV	Move operand
PUSH	Push operand onto stack
POP	Pop operand off stack
PUSHA	Push all registers on stack
POPA	Pop all registers off stack
XCHG	Exchange operand, register
XLAT	Translate
Conversion	
MOVZX	Move Byte or Word, Dword, with zero extension
MOVSX	Move Byte or Word, Dword, sign extended
CBW	Convert Byte to Word, or Word to Dword
CDW	Convert Word to Dword
CDWE	Convert Word to Dword extended
CDQ	Convert Dword to Qword
Input/output	
IN	Input operand from I/O space
OUT	Output operand to I/O space
Address object	
LEA	Load effective address
LDS	Load pointer into D segment register
LES	Load pointer into E segment register
LFS	Load pointer into F segment register
LGS	Load pointer into G segment register
LSS	Load pointer into S (stack) segment register
Flag manipulation	
LAHF	Load AH register from flags

TABLE 4.1 80386 Instructions (*continued*)

	Data Transfer Instructions
SAHF	Store AH register in flags
PUSHF	Push flags onto stack
POPF	Pop flags off stack
PUSHFD	Push Eflags onto stack
POPFD	Pop Eflags off stack
CLC	Clear carry flag
CLD	Clear direction flag
CMC	Complement carry flag
STC	Set carry flag
STD	Set direction flag
	Arithmetic Instructions
Addition	
ADD	Add operand
ADC	Add with carry
INC	Increment operand by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
Subtraction	
SUB	Subtract operand
SBB	Subtract with borrow
DEC	Decrement operand by 1
NEG	Negate operand
CMP	Compare operands
AAS	ASCII adjust for subtraction
Multiplication	
MUL	Multiply double/single precision
IMUL	Integer multiply
AAM	ASCII adjust after multiply
Division	
DIV	Divide unsigned
IDIV	Integer divide
AAD	ASCII adjust after division
	String Instructions
MOVS	Move Byte or Word, Dword string
INS	Input string from I/O space
OUTS	Output string to I/O space
CMPS	Compare Byte or Word, Dword string
SCAS	Scan Byte or Word, Dword string
LODS	Load Byte or Word, Dword string
STOS	Store Byte or Word, Dword string
REP	Repeat
REPE/REPZ	Repeat while equal/zero
RENE/REPNZ	Repeat while not equal/not zero
	Logical Instructions
Logicals	
NOT	"NOT" operand
AND	"AND" operand
OR	"Inclusive OR" operand
XOR	"Exclusive OR" operand
TEST	"Test" operand
Shifts	
SHL/SHR	Shift logical left or right
SAL/SAR	Shift arithmetic left or right
SHLD/SHRD	Double shift left or right

TABLE 4.1 80386 Instructions (*continued*)

Rotates	
ROL/ROR	Rotate left/right
RCL/RCR	Rotate through carry left/right
	Bit Manipulation Instructions
Single bit instructions	
BT	Bit test
BTS	Bit test and set
BTR	Bit test and reset
BTC	Bit test and complement
BSF	Bit scan forward
BSR	Bit scan reverse
Bit string instructions	
IBTS	Insert bit string
XBTS	Exact bit string
	Program Control Instructions
Conditional transfers	
SETCC	Set byte equal to condition code
JA/JNBE	Jump if above/not below nor equal
JAE/JNB	Jump if above or equal/not below
JB/JNAE	Jump if below/not above nor equal
JBE/JNA	Jump if below or equal/not above
JC	Jump if carry
JE/JZ	Jump if equal/zero
JG/JNLE	Jump if greater/not less nor equal
JGE/JNL	Jump if greater or equal/not less
JL/JNGE	Jump if less/not greater nor equal
JLE/JNG	Jump if less or equal/not greater
JNC	Jump if not carry
JNE/JNZ	Jump if not equal/not zero
JNO	Jump if not overflow
JNP/JPO	Jump if not parity/parity odd
JNS	Jump if not sign
JO	Jump if overflow
JP/JPE	Jump if parity/parity even
JS	Jump if sign
Unconditional transfers	
CALL	Call procedure/task
RET	Return from procedure/task
JMP	Jump
Iteration controls	
LOOP	Loop
LOOPE/LOOPZ	Loop if equal/zero
LOOPNE/ LLOPNZ	Loop if not equal/not zero
JCXZ	JUMP if register CX = 0
Interrupts	
INT	Interrupt
INTO	Interrupt if overflow
IRET	Return from interrupt
CLI	Clear interrupt enable
STI	Set interrupt enable
	High Level Language Instructions
BOUND	Check array bounds

TABLE 4.1 80386 Instructions (*continued*)

ENTER	Setup parameter block for entering procedure
LEAVE	Leave procedure
	Protection Model
SGDT	Store global descriptor table
SIDT	Store interrupt descriptor table
STR	Store task register
SLDT	Store local descriptor table
LGDT	Load global dedcriptor table
LIDT	Load interrupt descriptor table
LTR	Load task register
LLDT	Load local descriptor table
ARPL	Adjust requested privilege level
LAR	Load access rights
LSL	Load segment limit
VERR/VERW	Verify segment for reading or writing
LMSW	Load machine status word (lower 16 bits of CR0)
SMSW	Store machine status word
	Processor Control Instructions
HLT	Halt
WAIT	Wait until BUSY # negated
ESC	Escape
LOCK	Lock bus

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 4.3 lists the various conditions referring to the relation between two numbers (signed and unsigned) for Jcond and SETcond instructions.

All new 80386 instructions along with those which have minor variations from the 80286 are listed in alphabetical order below in Table 4.4.

A detailed description of most of the new 80386 instructions is given in the following.

TABLE 4.2 Status Flag Summary

Status Flag Functions

Bit	Name	Function
0	CF	Carry flag — Set on high-order bit carry or borrow; cleared otherwise
2	PF	Parity flag — Set if low-order eight bits of result contain an even number of 1 bits; cleared otherwise
4	AF	Adjust flag — Set on carry from or borrow to the low-order four bits of AL; cleared otherwise; used for decimal arithmetic
6	ZF	Zero flag — Set if result is zero; cleared otherwise
7	SF	Sign flag — Set equal to high-order bit of result (0 is positive, 1 is negative)
11	OF	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

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	OF	SF	ZF	AF	PF	CF
AAS	—	—	—	TM	—	M
AAD	—	M	M	—	M	—
AAM	—	M	M	—	M	—
DAA	—	M	M	TM	M	TM
DAS	—	M	M	TM	M	TM
ADC	M	M	M	M	M	TM
ADD	M	M	M	M	M	M
SBB	M	M	M	M	M	TM
SUB	M	M	M	M	M	M
CMP	M	M	M	M	M	M
CMPS	M	M	M	M	M	M
SCAS	M	M	M	M	M	M
NEG	M	M	M	M	M	M
DEC	M	M	M	M	M	
INC	M	M	M	M	M	
IMUL	M	—	—	—	—	M
MUL	M	—	—	—	—	M
RCL/RCR 1	M					TM
RCL/RCR count	—					TM
ROL/ROR 1	M					M
ROL/ROR count	—					M
SAL/SAR/SHL/SHR 1	M	M	M	—	M	M
SAL/SAR/SHL/SHR count	—	M	M	—	M	M
SHLD/SHRD	—	M	M	—	M	M
BSF/BSR	—	—	M	—	—	—
BT/BTS/BTR/BTC	—	—	—	—	—	M
AND	0	M	M	—	M	0
OR	0	M	M	—	M	0
TEST	0	M	M	—	M	0
XOR	0	M	M	—	M	0

TABLE 4.3 Condition Codes
(For Conditional Instructions Jcond, and SETcond)

Mnemonic	Meaning	Condition tested
O	Overflow	OF = 1
NO	No overflow	OF = 0
B	Below	
NAE	Neither above nor equal	CF = 1
NB	Not below	CF = 0
AE	Above or equal	
E	Equal	ZF = 1
Z	Zero	
NE	Not equal	ZF = 0
NZ	Not zero	
BE	Below or equal	(CF or ZF) = 1
NA	Not above	
NBE	Neither below nor equal	(CF or ZF) = 0
A	Above	
S	Sign	SF = 1
NS	No sign	SF = 0
P	Parity	PF = 1
PE	Parity even	
NP	No parity	PF = 0
PO	Parity odd	
L	Less	(SF ⊕ OF) = 1
NGE	Neither greater nor equal	
NL	Not less	(SF ⊕ OF) = 0
GE	Greater or equal	
LE	Less or equal	((SF ⊕ OF) or ZF) = 1
NG	Not greater	
NLE	Neither less nor equal	((SF ⊕ OF) or ZF) = 0
G	Greater	

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).

TABLE 4.4 80386 Instructions (New Instructions beyond Those of 8086/80186/80286)

Instruction	Comment
ADC EAX, imm32	Add CF with sign-extended immediate byte and 32-bit data in reg32 or mem32
ADC reg32/mem32, imm32	
ADC reg32/mem32, imm8	
ADC reg32/mem32, reg32	
ADC reg32, reg32/mem32	
ADD EAX, imm32	
ADD reg32/mem32, imm32	Immediate data byte is sign-extended before addition
ADD reg32/mem32, imm8	
ADD reg32/mem32, reg32	
ADD reg32, reg32/mem32	
AND EAX, imm32	
AND reg32/mem32, imm32	
AND reg32/mem32, imm8	
AND reg32/mem32, reg32	
AND reg32, reg32/mem32	
BOUND reg32, mem64	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
BSF	Bit scan forward
BSR	Bit scan reverse

TABLE 4.4 80386 Instructions (New Instructions beyond Those of 8086/80186/80286) (*continued*)

Instruction	Comment
BT	Bit test
BTC	Bit test and complement
BTR	Bit test and reset
BTS	Bit test and set
CALL label	There are several variations of the label such as disp16, disp32, 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 in 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 CALL instructions; the far calls CALL ptr16:16 or ptr16:32 uses a four-byte or six-byte operand as a long pointer to the procedure called; the CALL mem16:16 or mem16:32 reads the long pointer from the memory location specified (indirection); in real-address mode or virtual 86 mode, the long pointer provides 16 bits for CS and 16 or 32 bits for EIP depending on operand size; the far-call instruction pushes CS and IP (or EIP) and return addresses; in protected mode, both long pointers have different meaning; consult Intel manuals for details
CDQ	Convert doubleword to quadword
CMP reg32/mem32, imm32	EDX : EAX ← sign extend EAX $\begin{bmatrix} \text{reg32} \\ \text{or} \\ \text{mem32} \end{bmatrix}$ – imm32 ⇒ affects flags
CMP EAX, imm32	
CMP reg32/mem32, imm8	Compare sign extended 8-bit data with reg32 or mem32
CMP reg32mem32, reg32	
CMP reg32, reg32/mem32	
CMPSD	
CMPS mem32, mem31	
CWDE	EAX ← sign extend AX
DEC reg32/mem32	
DIV EAX, reg32/mem32	Unsigned divide EDX:EAX by reg32 or mem32 (EAX = quotient, EDX = remainder)
ENTER imm16, imm8	Create a stack frame before entering a procedure
IDIV EAX, reg32/mem32	Signed divide; everything else is same as DIV
IMUL reg32/mem32	Signed multiply EDX: EAX ← EAX * reg32 or mem32
IMUL reg32, reg32/mem32	reg32 ← reg32 *reg32 or mem32; upper 32 bits of the product are discarded
IMUL reg16, reg16/mem16	reg16 ← reg16* reg16/mem16; result is low 16 bits of the product
IMUL reg16, reg16/mem16, imm8	reg16 ← reg16/mem16 * (sign extended imm8); result is low 16 bits of the product
IMUL reg32, reg32/mem32, imm8	reg32 ← reg32/mem32* (sign extended imm8); result is low 32 bits of the product
IMUL reg16, imm8	In all IMUL instructions, size of the result is defined by the size of the destination (first) operand.
IMUL reg32, imm8	
IMUL reg16, reg16/mem16,	
IMUL reg32, reg32/mem32,	
IMUL reg16, imm16	
IMUL reg32, imm32	
IN EAX, imm8	Input 32 bits from immediate port into EAX
IN EAX, DX	Input 32 bits from port DX into EAX
INC reg32/mem32	
INS reg32/mem32, DX or INSD	

TABLE 4.4 80386 Instructions (New Instructions beyond Those of 8086/80186/80286) (*continued*)

Instruction	Comment
IRET/IRETD	In real-address mode, IRET pops IP, CS, and FLAGS, and IRETD POPS EIP, CS, and EFLAGS; for protection mode, consult Intel manuals
Jcc label	cc can be any of the 31 conditions, including the flag settings and less, greater, above, equal, etc.; label can be disp8 as with the 80286; in 80386, however, one can have disp16 and disp32 as label; all these are signed displacements; the 80386 includes the 80286 JCXZ disp8 instruction; furthermore, the 80386 includes a new instruction called JECXZ disp8; (Jump if ECX = 0)
JMP label	The label can be specified in the same way as the CALL label instruction; see CALL label explanation for near-jump and far-jump explanations
LAR reg32, reg32/mem32	Load access right byte; reg32 ← reg32 or mem32 masked by 00FXFF00H
LEA reg32, m	Calculate the effective address (offset part) m and store it in reg32
LEAVE	LEAVE releases the stack space used by a procedure for its local variables; LEAVE reverses the action of ENTER instruction; LEAVE sets ESP to EBP and then POPS EBP
LGS/LSS/LDS/LES/LFS	Load full pointer; explained later
LODS mem32 or LODSD	To be explained later
LOOP disp8	To be explained later
LOOPcond disp8	To be explained later
MOV reg32/mem32, reg32	
MOV reg32, reg32/mem32	
MOV EAX, mem32	Move 32 bits from mem32 to EAX
MOV mem32, EAX	
MOV reg32, imm32	
MOV reg32/mem32, imm32	
MOV reg32, CR0/CR2/CR3	CRs are control registers
MOV CR0/CR2/CR3, reg32	
MOV reg32, DR0/DR1/DR2/D3	DRs are debug registers
MOV reg32, DR6/DR7	
MOV DR0/DR1/DR2/DR3, reg32	
MOV DR6/DR7, reg32	
MOV reg32, TR6/TR7	TRs are test registers
MOV TR6/TR7, reg32	
MOVS mem32, mem32 or MOVSD	To be explained later
MOVSX	Move with sign extend; to be discussed later
MOVZX	Move with zero extend; to be discussed later
MUL EAX, reg32/mem32	Unsigned multiply; EDX: EAX ← EAX * (reg32 or mem32)
NEG reg32/mem32	Two's complement. Negate 32 bits in reg32 or mem32
NOT reg32/mem32	Ones complement
OR d,s	The definitions for d and s are same as AND
OUT imm8, EAX	Output 32-bit EAX to immediate port number
OUT DX, reg32/mem32 or OUTSD	To be explained later
POP reg32	To be explained later
POP mem32	To be explained later
POP FS	To be explained later
POP GS	To be explained later
POPAD	To be explained later
POPFD	To be explained later
PUSH reg32	To be explained later
PUSH mem32	To be explained later
PUSH FS	To be explained later
PUSH GS	To be explained later
PUSHAD	To be explained later
PUSHFD	To be explained later
RCL reg32/mem32, 1	Rotate reg32 or mem32 thru CF once to left

TABLE 4.4 80386 Instructions (New Instructions beyond Those of 8086/80186/80286) (*continued*)

Instruction	Comment
RCL reg32/mem32, CL	Rotate reg32 or mem32 thru CF to left CL times
RCL reg32/mem32, imm8	
RCR reg32/mem32, 1	
RCR reg32/mem32, CL	
RCR reg32/mem32, imm8	
ROL reg32/mem32, 1	
ROL reg32/mem32, CL	
ROL reg32/mem32, imm8	
ROR reg32/mem32, 1	
ROR reg32/mem32, CL	
ROR reg32/mem32, imm8	
SAL/SAR/SHL/SHR d, n	d and n have same definitions as RCL/RCR/ROL/ROR
SBB d,s	d and s have same definitions as ADD d,s
SCAS mem32 or SCASD	To be explained later
SET cc	To be explained later
SHLD	To be explained later
SHRD	To be explained later
STOS mem32 or STOSD	To be explained later
SUB d,s	d and s have same definitions as ADD
TEST EAX, imm32	
TEST reg32/mem32, imm32	
TEST reg32/mem32, reg32	
XCHG reg32, EAX	Exchange 32-bit register contents with EAX
XCHG EAX, reg32	
XCHG reg32/mem32, reg32	
XCHG reg32, reg32/mem32	
XLATB	Set AL to memory byte DS: [EBX + unsigned AL]. DS cannot be overridden. In XLAT, DS can be overridden.
XOR d,s	d and s have same definitions as AND

Note: All MUL operands are same as IMUL operands.

4.2.2.a Sign-Extension 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 Manipulation Instructions

The following lists the 80386 six-bit manipulation instructions:

```
BSF   Bit scan forward
BSR   Bit scan reverse
BT    Bit test
BTC   Bit test and complement
BTR   Bit test and reset
BTS   Bit test and set
```

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] = 01241240_{16}, then [EBX] = 00000006_{16} 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* [bit offset divided by 32]). As an example, consider BT CX, DX. If [CX] = $081F_{16}$, [DX] = 0021_{16}, then since the content of DX is 33_{10}, the bit number one (remainder of 33/16 = 1) of CX (value 1) is reflected in the CF and therefore CF = 1.

- BTC (Bit Test and Complement)

```
BTC  d , s
```

d and s have the same definitions as the BT instruction. The bit of d defined by s is reflected in the CF. After CF is assigned, the same bit of d defined by s is ones

complemented. The 80386 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)

```
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)

```
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 SETcc (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:

Instruction	Condition codes	Description
SETA/SETNBE reg 18/mem8	CF = 0 and ZF = 0	Set byte if above or not below/equal
SETAE/SETNB/SETNC reg8/mem8	CF = 0	Set if above/equal, set if not below, or set if not carry
SETB/SETNAE/SETC reg8/mem8	CF = 1	Set if below, set if not above/equal, or set if carry
SETBE/SETNA reg8/mem8	CF = 1 or ZF = 1	Set if below/equal or set if not above
SETE/SETZ reg8/mem8	ZF = 1	Set if equal or set if zero
SETG/SETNLE reg8/mem8	ZF = 0 or SF = OF	Set if greater or set if not less/equal
SETGE/SETNL reg8/mem8	SF = OF	Set if greater/equal or set if not less
SETL/SETNGE reg8/mem8	SF ≠ OF	Set if less or set if not greater/equal
SETLE/SETNG reg8/mem8	ZF = 1 and SF ≠ OF	Set if less/equal or set if not greater
SETNE/SETNZ reg8/mem8	ZF = 0	Set if not equal or set if not zero
SETNO reg8/mem8	OF = 0	Set if no overflow
SETNP/SETPO reg8/mem8	PF = 0	Set if no parity or set if parity odd
SETNS reg8/mem8	SF = 0	Set if not sign
SETO reg8/mem8	OF = 1	Set if overflow
SETP/SETPE reg8/mem8	PF = 1	Set if parity or set if parity even
SETS reg8/mem8	SF = 1	Set if sign

As an example, consider SETB BL. If [BL] = 52_{16} and CF = 1, then after this instruction is executed [BL] = 01_{16} 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_{16}, CF = 0 and ZF = 1; all other flags are undefined. Similarly, the other SETcc instructions can be explained.

4.2.2.d Conditional Jumps and Loops

JECXZ disp8 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 ($+127_{10}$ to -128_{10} 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 LOOPNE label. The JECXZ prevents entering the loop with ECX = 0, which would cause the loop to execute up to 2^{32} times instead of zero times.

LOOP Instructions

Instruction	Description
LOOP disp8	Decrement CX/ECX by one and jump if CX/ECX ≠ 0
LOOPE/LOOPZ disp8	Decrement CX/ECX by one and jump if CX/ECX ≠ 0 and ZF = 1
LOOPNE/LOOPNZ disp8	Decrement CX/ECX by one and jump if CX/ECX ≠ 0 and ZF = 0

The 80386 LOOP instructions are similar to those of 8086/80186/80286, except that if the counter is more than 16 bits, ECX rather than CX register is used as the counter.

4.2.2.e Data Transfer

- Move instructions description

```
MOVSX  d,s  Move and sign extend
MOVZX  d,s  Move and zero extend
```

The d and s operands are defined as follows:

```
MOVSX  d,s
or
MOVZX  reg16,  reg8
       reg16,  mem8
       reg32,  reg8
       reg32,  mem8
       reg32,  reg16
       reg32,  mem16
```

MOVSX 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 MOVSX BX, CL. If CL = 81_{16} and [BX] = $21AF_{16}$, then after execution of MOVSX BX, CL, register BX will contain $FF81_{16}$ and CL contents do not change. Also, consider MOVZX CX, DH. If CX = $F237_{16}$ and [DH] = 85_{16}, then after execution of this MOVZX, CX register will contain 0085 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, ESI, and EDI) onto the 80386 stack. PUSHAD decrements the stack pointer (ESP) by 32_{10} to hold the eight 32-bit values. No flags are affected. POPAD reverses a previous PUSHAD. It pops the eight 32-bit registers (the order is EDI, ESI, EBP, ESP, EBX, EDX, ECS, 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 32_{10}.

- 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: mem16 or mem16: mem32 defines a memory operand containing four pointers composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to the offset. These instructions read a full pointer from memory and store it in the selected segment register: specified register. The instruction loads 16 bits into DS (for LDS) or into ES (for LES). The other register loaded is 32 bits for 32-bit operand size and 16 bits for 16-bit operand size. The 16- and 32-bit registers to be loaded are determined by reg16 or reg32 register specified.

The three new instructions LFS, LGS, and LSS associated with segment registers FS, GS, and SS can similarly be explained.

4.2.2.f Flag Control

There are two new 80386 instructions beyond those of the 80286. These are PUSHFD and POPFD. PUSHFD decrements the stack pointer by 4 and saves the 80386 EFLAGS register to the new top of the stack. No flags are affected. POPFD, on the other hand, pops the 32-bit (doubleword) from the stack-top and stores the value in EFLAGS. All flags except VM and RF are affected.

4.2.2.g Logical

There are two new 80386 logical instructions beyond those of 80286. These are SHLD and SHRD.

Instruction	Description
SHLD d,s, count	Shift left double
SHRD d,s, count	Shift right double

The operands are defined as follows:

d	s	count
reg16	reg16	imm8
mem16	reg16	imm8
reg16	reg16	CL
mem16	reg16	CL
reg32	reg32	CL
mem32	reg32	imm8
reg32	reg32	CL
mem32	reg32	CL

For both SHLD and SHRD, the shift count is defined by the low five bits and, therefore, shifts up to 0 to 31 can be obtained.

SHLD shifts by the specified shift count the contents of d:s 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 last 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 d:s 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 last bit shifted out. OF and AF are undefined.

As an example, consider SHLD BX, DX, 2. If [BX] = $183F_{16}$, [DX] = $01F1_{16}$, then after this SHLD, [BX] = $60FC_{16}$, [DX] = $01F1_{16}$, CF = 0, SF = 0, ZF = 0, and PF = 1.

Similarly, the SHRD instruction can be illustrated.

4.2.2.h 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:EDI (second operand) with DS:ESI and affects the flags. The direction of subtraction of CMPS is [[ESI]] – [[EDI]]. 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. CMPSD can be preceded by the REPE 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 MOVS 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. MOVS 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). INSD inputs 32-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. SCASD can be preceded by the REPE or REPNE prefix for block search of ECX doublewords. All flags are affected.

4.2.2.i 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). DS can be overridden. No flags are affected.

4.2.2.j High-Level Language Instructions

The three instructions ENTER, LEAVE, and BOUND (also available with 80186/80286) in this category have been enhanced in the 80386.

Before a subroutine is called by a main program, it is required quite often to pass some parameters to the subroutine by the main program. Normally these parameters are pushed onto the stack before calling the subroutine and then they are used by the subroutine by popping them from stack during its execution. In the 80386, a portion of the stack called the stack frame is used to store these parameters. Two 80386 instructions, namely, ENTER and LEAVE, are included for allocating and deallocating stack frames.

The ENTER imm16, imm8 instruction creates a stack frame. The data imm8 defines the nesting depth (also called the lexical level) 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 imm8 specifies the number of bytes of local variables for which the stack space is to be allocated.

ENTER can be used for either nested or non-nested subroutines or procedures. For example, if the lexical level imm8 is zero, the non-nested form is used. 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 the 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.

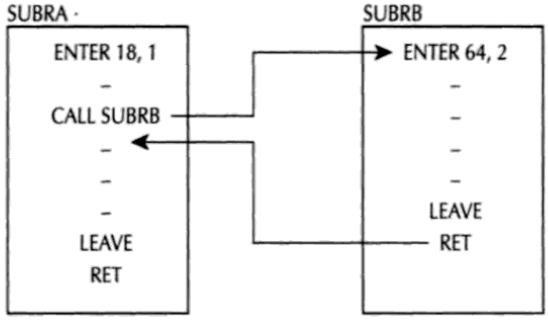
The formal definition of the ENTER instruction is given in the following:

```
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:
              EBP: = EBP-4
              Push all EBP for the previous subroutines
                 and then EBP for the present subroutine.
      End repeat
      Push FRAME-PTR
End if
EBP:  = FRAME-PTR
ESP:  = ESP - first operand, imm16
```

The LEAVE instruction performs the following:

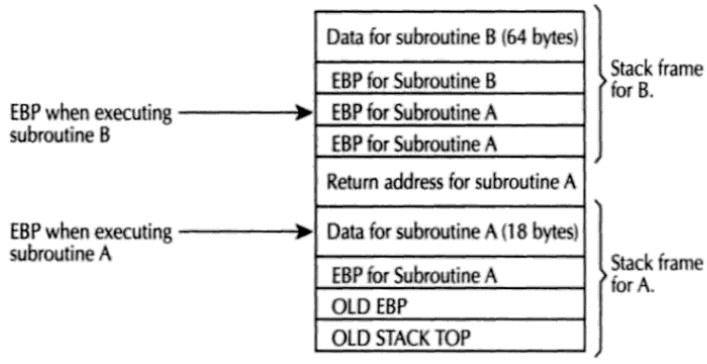
- (ESP) ← (EBP)
- POP into EBP.

In order to illustrate the Enter and Leave instructions, consider the following:



In the above, subroutine A (SUBRA) calls subroutine B. It is assumed that the nesting depth for these subroutines are 1 and 2 respectively.

The stack frames created after execution of the ENTER instructions in the two subroutines are shown below:



As the ENTER instruction in subroutine A is executed, the old EBP from the subroutine that called SUBRA is pushed onto the stack. EBP is loaded from ESP to point to the location of the

old EBP. The lexical level for the ENTER for subroutine A is pushed onto the stack. Finally, in order to allocate 18 bytes, 12H is subtracted from the current ESP.

After entering into the subroutine B, a second ENTER instruction ENTER 64, 2 is executed. Since the lexical level in this case is 2, the Old EBP (EBP for subroutine A) is first pushed onto the stack. The EBP for subroutine A previsouly stored on the stack frame is pushed onto the stack. Finally, the current EBP for subroutine B is pushed onto the stack. This mechanism provides access to the stack frame for subroutine A from subroutine B. Next, local storage of 64 bytes as specified in the ENTER instruction are allocated for local storage.

The BOUND instruction checks to determine if the contents of a register called the array index lie within the minimum (lower bound) and maximum (upper bound) limits of an array.

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_L and [ADDR + 4] = 32-bit upper bound d_U. If, after execution of this instruction, [EDI] < d_L and > du, the 80386 traps to interrupt 5; otherwise the array is accessed.

The BOUND instruction is usually placed following the computation of an index value to ensure that the limits of the index value are not violated. This allows checking whether or not the address of an array being accessed is within the array boundaries when the address register indirect with index mode is used to access an element in the array. For example, the following instruction segment will allow accessing an array with base address in ESI, the index value in EDI, and an array length of 200_{10} bytes. Assume the 32-bit contents of memory location 52070422_{16} and 52070426_{16} are 0 and 199_{10}, respectively:

```
-
-
-
BOUND   EDI, 52070422H
MOV     EAX, [EDI][ESI]
```

In the above, if the contents of EDI are not within the array boundaries of 0 and 199_{10}, the 80386 will trap to interrupts and the MOV instruction will not be executed.

In the following 80386 programming examples, the 80386 is assumed to be real mode. The 80386 assembler directives are not included. These directives are similar to those of the 8086.

Example 4.2

Determine the effect of each one of the following 80386 instructions:

i) **CDQ**
ii) **BTC CX, BX**
iii) **MOVSX ECX, E7H**

Assume [EAX] = FFFFFFFFH, [ECX] = F1257124H, [EDX] = EEEEEEEEH, [BX] = 0004H, [CX] = 0FA1H, prior to execution of each of the above instructions.

i) After CDQ,
[EAX] = FFFF FFFFH
[EDX] = FFFF FFFFH
ii) After BTC CX, BX, bit 4 of register CX is reflected in the ZF and then ones complemented in CX.

Before

Hence, [CX] = 0FB1H and [BX] = 0004H

ii) MOVSX ECX, E7H copies the 8-bit data E7H into low byte of ECX and then sign-extends to 32 bits. Therefore, after MOVSX ECX, E7H, [ECX] = FFFFFFE7H.

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. Initialize DS to 2000H, ES to 3000H, ESI to 0100H and EDI to 0200H.

Solution

```
MOV ECX, 10000      ;  Initialize counter
MOV BX, 2000H       ;  Initialize DS
MOV DS, BX          ;  register
MOV BX, 3000H       ;  Initialize ES
MOV ES, BX          ;  register
MOV ESI, 0100H      ;  Initialize ESI
MOV EDI, 0200H      ;  Initialize EDI
CLD                 ;  Clear DF to Autoincrement
REPMOVSD            ;  MOV A (i) to
                    ;  B (i) until ECX = 0
HLT
```

Example 4.5

Identify the addressing modes and the size of the operation for the following instructions:

i) `MOV EAX, [EBX*4]`
ii) `ADD AH, [EBX+35][ESI]`

Solution

i)

Source	Destination
Scaled Index 32-bit	Register

ii)

Source	Destination
Based Indexed 8-bit	Register

Example 4.6

Compute the physical address for the specified operands of the following instructions. Assume DS = 0300_{16}, ESI = 00005000_{16}, EBX = 00000200_{16}.

i) `MOV BH, [SI]`
ii) `ADD [BX+50H], CX`

Solution

i) Physical address for the source = $03000_{16} + 5000_{16} = 08000_{16}$
Physical address for the source = $03000_{16} + 0250_{16}$
$= 03250_{16}$

Example 4.7

Write an 80386 instruction sequence to compute the following:

$$\texttt{INTEGER} = (\texttt{INTEGER1} \oplus \overline{\texttt{EDX}}) \vee (\texttt{ECX} \cdot \texttt{EDX})$$

Assume that the contents of locations INTEGER and INTEGER1 are 32-bit wide.

Solution

```
MOV EAX, EDX
NOT EDX                 ;  EDX ← NOT EDX
XOR EDX, [INTEGER1]     ;  EDX ← (INTEGER1) ⊕ NOT EDX
AND ECX, EAX            ;  ECX ← ECX · EDX
OR ECX, EDX             ;  ECX ← EDX ∨ ECX
MOV [INTEGER], ECX      ;  [INTEGER] ← ECX
```

Example 4.8

Write an 80386 assembly language program to add two 64-bit binary numbers. The numbers are pointed to by ESI and EDI, respectively. Store result in location pointed to by EDI. Assume data are already loaded in memory locations.

Solutions

```
MOV EAX, [ESI]    ;  Load first 64-bit
MOV EBX, [ESI+4]  ;  number
ADD [EDI], EAX    ;  Add with
ADC [EDI+4], EBX  ;  second 64-bit number
HLT
```

Example 4.9

Write an 80386 assembly language program to check whether the 64-bit number stored in memory pointed to by ESI is zero. If it is zero, store 0 in AL; otherwise store 1 in AL.

Solution

```
        MOV         EBX, [ESI]    ;  Move the upper 32-bit into
                                  ;  EBX
        OR          EBX, [ESI+4]  ;  Check whether both halves
                                  ;  are zero.
        JNZ         ZERO
        MOV         AL, 1         ;  The number is not zero
        HLT
ZERO:   MOV AL, 0                 ;  The number is zero
        HLT
```

Example 4.10

Assume the content of physical memory location 21010_{16} is $2F_{16}$, [ADDR] = 2000_{16} and double word stored at location ADDR+2 is 02340110_{16}. Find the contents of EAX register after execution of the following 80386 instruction sequence:

```
LDS   EAX, [ADDR]
MOV   EBX, 00001000H
XLAT
```

Solution

LDS instruction in the above loads DS with 2000_{16} and EAX with 02340110_{16}. MOV loads EBX with $00001000H_{16}$. The XLAT instruction loads AL with the contents of memory location addressed by DS + BX + AL which is $20000_{16} + 1000_{16} + 10_{16} = 21010_{16}$. Therefore, [EAX] = $0234012F_{16}$.

Example 4.11

Write an 80386 assembly language program to multiply an unsigned 32-bit number in EBX by an unsigned 16-bit number in CX. Store result in EDX:EAX.

Solution

```
MOVZX ECX, CX   ;  Zero Extend CX.
MOV EAX, EBX    ;  Move to EAX for multiplication
MUL EAX, ECX    ;  Unsigned multiplication
HLT
```

Example 4.12

Write an 80386 assembly language instruction sequence to extract the bit field EAX[31:24] and store it in EBX[7:0], so that EBX[31:8] is zero and the original EAX is not affected.

Solution

```
MOV   EBX, EAX
MOV   CL, 8
ROL   EBX, CL
AND   EBX, 000000FFH
```

Example 4.13

Write 80386 assembly language program to compute the following: $X = Y + Z - 1F20_{16}$ where X, Y, and Z are 64-bit variables. The lower 32 bits of Y and Z are stored starting at locations NUMBER and NUMBER + 8, each followed by the upper 32 bits. Store the lower 32-bit of the 64-bit result in EAX followed by the upper 32 bits in ECX.

Solution

```
MOV   EAX, [NUMBER]      ;  Load EAX with low 32-bit of Y
MOV   EBX, [NUMBER+4]    ;  Load EBX with high 32-bit of Y
ADD   EAX, [NUMBER+8]    ;  Add low 32 bits
MOV   ECX, [NUMBER+12]   ;  Load ECX with high 32-bit of Z
ADC   ECX, EBX           ;  Add high 32 bits and carry
SUB   EAX, 1F20H         ;  Subtract 1F20H from 32-bit of
                            result
MOV   EBX, 0             ;  If borrow,
SBB   ECX, EBX           ;  Subtract from high 32-bit of result
HLT
```

The 80386 assembly language programs can be assembled by using 80386 assemblers on IBM PC's.

A typical program structure is given below:

```
        PAGE 55, 132              ;  Set page dimensions
        .386
STACK   SEGMENT 'STACK' STACK
        DW 50 DUP(?)
STACK   ENDS
PROG    SEGMENT PARA 'CODE' PUBLIC USE16
        ASSUME CS:PROG, DS:DATA, SS:STACK
BEGIN:  MOV BX, 3000H
        MOV DS, BX
(Specify the constants and variables here)
        (Enter program here)
        MOV AX, 4C00H             ;  RETURN
        INT 21H                   ;  to DOS
PROG    ENDS
        END BEGIN
```

In the above, the PAGE directive in the first line specifies the number of lines on a page and the width of each line for the assembler. The notation .386 in the second line indicates to the assembler that the program includes the 80386 instructions. The next three lines define the stack segment with 50 bytes.

The code segment where the actual program starts is defined next. The logical name of the code segment in the above is PROG. The code segment is public and tells the assembler to use 16-bit registers.

The 80386 uses a new directive USE16 or USE32. Programs that are developed to run on the 8086/80286 must use the USE16 to specify that all operand and address sizes are 16 bits. This automatically limits segment size to 64K. With the USE32, programs are assembled with an operand and address sizes of 32 bits. This allows access to up to 4 gigabytes of memory.

INT 21H with 4C00H in AX returns control to DOS operating system. The 80386 assembly language programs can be assembled with an assembler such as the PHAR LAP '386' assembler. All examples in this chapter are written without the complete 80386 directives. The main purpose is to illustrate use of 80386 instructions for writing assembly language programs.

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^{32} bits), the programmer views a virtual address space of 2^{46} or 64 tetrabytes 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.

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^{32} 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.

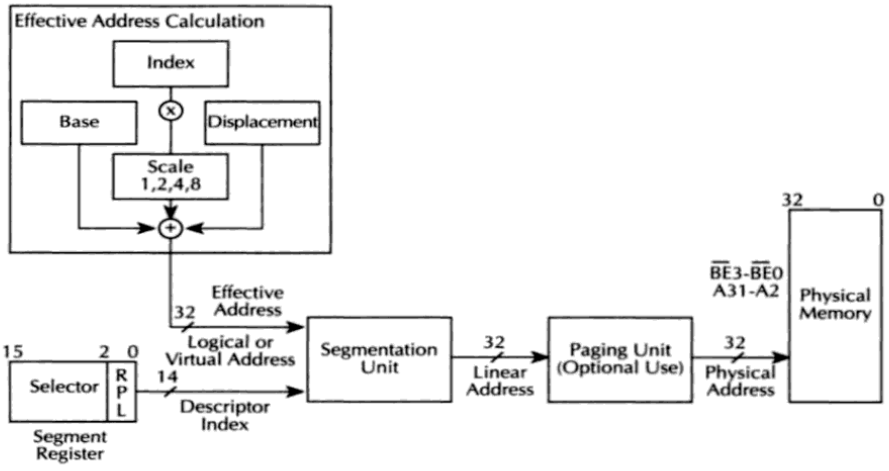


FIGURE 4.12 Address translation mechanism.

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 00F8H 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 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.

TABLE 4.5 Segment Register Selection Rules

Type of memory reference		Implied (default) segment use	Segment override prefixes possible
Code Fetch		CS	None
Destination of PUSH, PUSHA instructions		SS	None
Source of POP, POPA instructions		SS	None
Other data references, with effective address			
using base register of:	[EAX]	DS	CS,SS,ES,FS,GS
	[EBX]	DS	CS,SS,ES,FS,GS
	[ECX]	DS	CS,SS,ES,FS,GS
	[EDX]	DS	CS,SS,ES,FS,GS
	[EBX]	DS	CS,SS,ES,FS,GS
	[ESI]	DS	CS,SS,ES,FS,GS
	[EDI]	DS	CS,SS,ES,FS,GS
	[EBP]	SS	CS,DS,ES,FS,GS
	[ESP]	SS	CS,DS,ES,FS,GS

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 INT2. The INT2 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 EFLAG registers 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 INT3 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 INT1. 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.

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:

Priority	Interrupt/exception
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 4.6 Interrupt Vector Assignments

Function	Interrupt number	Instruction which can cause exception	Return address points to faulting instruction	Type
Divide error	0	DIV, IDIV	Yes	Fault
Debug exception	1	Any instruction	Yes	Trap[a]
NMI interrupt	2	INT 2 or NMI	No	NMI
One-byte interrupt	3	INT	No	Trap
Interrupt on overflow	4	INTO	No	Trap
Array bounds check	5	BOUND	Yes	Fault
Invalid OP-code	6	Any illegal instruction	Yes	Fault
Device not available	7	ESC, WAIT	Yes	Fault
Double fault	8	Any instruction that can generate an exception		Abort
Coprocessor segment	9	Coprocessor tries to access data past the end of a segment	No	Trap[b]
Invalid TSS	10	JMP, CALL, IRET, INT	Yes	Fault
Segment not present	11	Segment register instructions	Yes	Fault
Stack fault	12	Stack references	Yes	Fault
General protection fault	13	Any memory reference	Yes	Fault
Page fault	14	Any memory access or code fetch	Yes	Fault
Coprocessor error	16	ESC, WAIT	Yes	Fault
Intel reserved	17—32			
Two-byte interrupt	0—255	INT n	No	Trap

[a]Some debug exceptions may report both traps on the previous instruction and faults on the next instruction.
[b]Exception 9 no longer occurs on the 80386 due to the improved interface between the 80386 and its coprocessors.

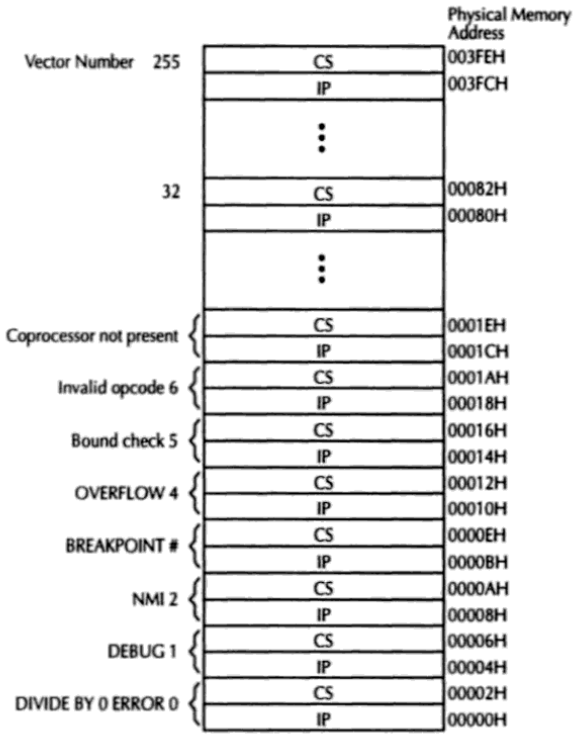
As an example, suppose an instruction causes both a debug exception (interrupt no. 1) and page fault (interrupt vector 14). According to the built-in priority mechanism, the 80386 will first service the page fault by executing the exception 14 handler. The exception 14 handler will be interrupted by 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 completed. After this, the exception 14 handler will be executed. This permits the system designer to debug the exception handler.

In real mode, the 80386 obtains the values of IP and CS similar to the 8086 by using a table called the interrupt pointer table. In protected mode, this table is called Interrupt descriptor table. The 80386 real-mode interrupt pointer table is shown on the following page.

4.2.6 80386 Reset and Initialization

When the 80386 is initialized and reset, the 80386 will start executing instructions near the top of physical memory, at location FFFFFFF0H. When the first Intersegment Jump or call is executed, address lines A20-A31 will drop low, and the 80386 will only execute instructions in the lower one megabyte of physical memory. This allows the system designer to use a ROM at the top of physical memory to intialize the system and take care of Resets. Driving the RESET input pin HIGH for at least 78 CLK2 periods resets the 80386.

Upon hardware reset, the 80386 registers contain the values as shown in Table 4.7.



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. The 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 above 30 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 breakpointing, 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 14×14 matrix, three rows around.

A number of sockets are available for low insertion force or zero insertion force mountings. Three types of terminals include soldertail, surface mount, or wire wrap. These application

TABLE 4.7 Register Values after Reset

Flag word(EFLAGS)	00000002H	Note 1
Machine status word (CR0)	UUUUUUU0H	Note 2
Instruction pointer (IP)	0000FFF0H	
Code segment	F000H	Note 3
Data segment	0000H	
Stack segment	0000H	
Extra segment (ES)	0000H	
Extra segment (FS)	0000H	
Extra segment (GS)	0000H	
All other registers	Undefined	

Note: U means undefined.

Note 1: The upper 14 bits of the EFLAGS registers are zero, VM (Bit 17) and RF (Bit 16) and other defined flag bits are zero.

Note 2: All of the defined fields in the CR0 are 0 (PG Bit 31, TS Bit 3, EM Bit 2, MP Bit 1, and PE Bit 0) except for ET Bit 4 (processor Extension type). The ET Bit is set during Reset according to the type of coprocessor in the system. If the coprocessor is an 80387 then ET will be 1, if the coprocessor is an 80287 or no coprocessor is present then ET will be 0. All other bits are undefined.

Note 3: The code segment Register (CS) will have its base address set to FFF00000H and Limit set to 0FFFFH. All undefined bits are reserved and should not be used.

sockets are manufactured by Amp, 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 80386 pins. A brief description of the 80386 pins and signals is provided in the following. The # symbol at the end of the signal name or the — symbol above a signal name indicates the active or asserted state when it is low. When the symbol # is absent after the signal name or the symbol — is absent above a 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 80386. This clock is divided by 2 internally to provide the internal clock used for instruction execution. The CLK2 signal is specified at CMOS-compatible voltage levels and not at TTL levels.

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 15 CLK2 periods. The RESET signal is level-sensitive. When the RESET pin is asserted, the 80386 ignores all input pins and drives all other 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.

The address pins A2-A31 along with the byte enable signals BE0# thru BE3# to generate physical memory or I/O port addresses. Using these pins, the 80386 can directly address 4

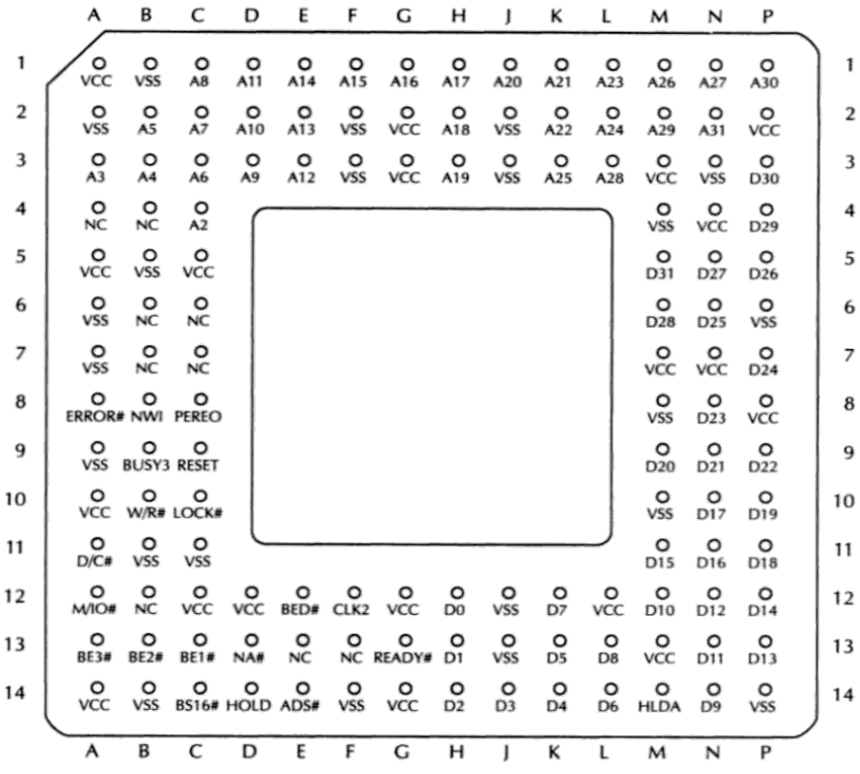


FIGURE 4.13 80386 PGA pinout view from pin side.

gigabytes by physical memory (00000000H thru FFFFFFFFH) and 64 kilobytes of I/O addresses (00000000H thru 0000FFFFH). The coprocessor addresses range from 800000F8H thru 800000FFH. Therefore, coprocessor select signal is generated by the 80386 when M/IO # is LOW and A31 is HIGH.

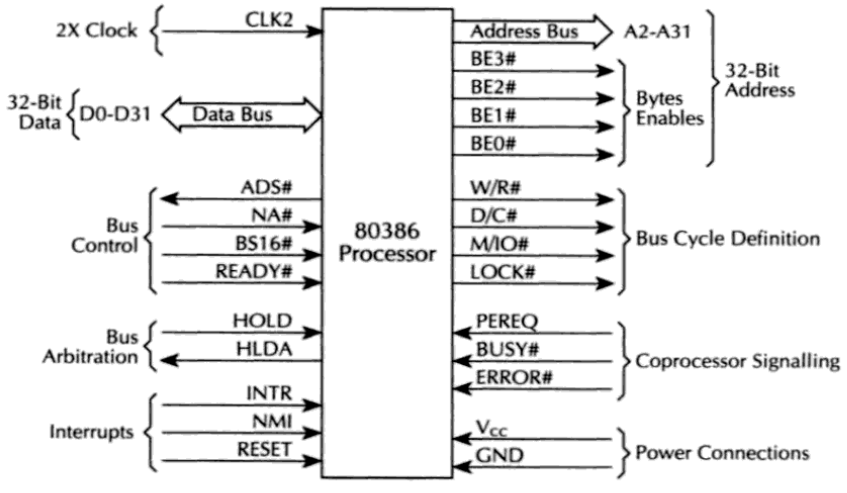


FIGURE 4.14 Functional signal groups.

TABLE 4.8 80386 PGA Pinout Functional Grouping

Pin/signal		Pin/signal		Pin/signal		Pin/signal	
NS	A31	M5	D31	A1	Vcc	A2	Vss
P1	A30	P3	D30	A5	Vcc	A6	Vss
M2	A29	P4	D29	A7	Vcc	A9	Vss
L3	A28	M6	D28	A10	Vcc	B1	Vss
N1	A27	N5	D27	A14	Vcc	B5	Vss
M1	A26	P5	D26	C5	Vcc	B11	Vss
K3	A25	N6	D25	C12	Vcc	B14	Vss
L2	A24	P7	D24	D12	Vcc	C11	Vss
L1	A23	N8	D23	G2	Vcc	FS	Vss
K2	A22	P9	D22	G3	Vcc	F3	Vss
K1	A21	N9	D21	G12	Vcc	F14	Vss
J1	A20	M9	D20	G14	Vcc	J2	Vss
H3	A19	P10	D19	L12	Vcc	J3	Vss
H2	A18	P11	D18	M3	Vcc	J12	Vss
H1	A17	N10	D17	M7	Vcc	J13	Vss
G1	A16	N11	D16	M13	Vcc	M4	Vss
F1	A15	M11	D15	N4	Vcc	M8	Vss
E1	A14	P12	D14	N7	Vcc	M10	Vss
E2	A13	P13	D13	P2	Vcc	N3	Vss
E3	A12	N12	D12	P8	Vcc	P6	Vss
D1	A11	N13	D11			P14	Vss
D2	A10	M12	D10				
D3	A9	N14	D9	F12	CLK2	A4	N.C.
C1	A8	L13	D8			D4	N.C.
C2	A7	K12	D7	E14	ADS#	B6	N.C.
C3	A6	L14	D6			B12	N.C.
B2	A5	K13	D5	B10	W/R#	C6	N.C.
B3	A4	K14	D4	A11	D/C#	C7	N.C.
A3	A3	J14	D3	A12	M/IO#	E13	N.C.
C4	A2	H14	D2	C10	LOCK#	F13	N.C.
A13	BE3#	H13	D1				
B13	BE2#	H12	D0	D13	NA#	C8	PEREQ
C13	BE1#			C14	BS16#	B9	BUSY#
E12	BE0#			G13	READY#	A8	ERROR#
		D14	HOLD				
C9	RESET	M14	HLDA	B7	INTR	B8	NMI

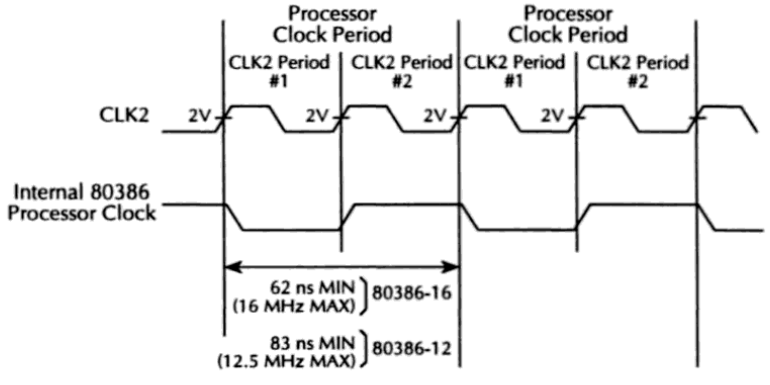


FIGURE 4.15 CLK2 signal and internal processor clock. The 82384 generates the 80386 CLK2 which is then divided by 2 by the 80386 internally.

The byte enable outputs, BE0# thru BE3# 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 when ADS# is 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 BS16# (bus size 16).

The 80386 outputs LOW on the ADS# pin to indicate a valid bus cycle (W/R#, D/C#, M/IO#) and address (BE0#-BE3#, A2-A31) signals.

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 external hardware to request address pipelining. A low on this pin means that the system is ready to receive new values of BE0#-BE3#, 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 BE1# for data transfer. If the 80386 asserts BE2# or BE3# during a bus cycle, then assertion of BS16# by an external device in this cycle

TABLE 4.9 Bus Cycle Definition

M/IO#	D/C#	W/R#	Bus cycle type		Locked?
Low	Low	Low	INTERRUPT ACKNOWLEDGE		Yes
Low	Low	High	Does not occur		—
Low	High	Low	I/O DATA READ		No
Low	High	High	I/O DATA WRITE		No
High	Low	Low	MEMORY CODE READ		No
High	Low	High	HALT:	SHUTDOWN:	No
			Address = 2	Address = 0	
			(BE0# High	(BE0# Low	
			BE1# High	BE1# High	
			BE2# Low	BE2# High	
			BE3# High	BE3# High	
			A2-A31 Low)	A2-A31 Low)	
High	High	Low	MEMORY DATA READ		Some cycles
High	High	High	MEMORY DATA WRITE		Some cycles

will automatically cause the 80386 to transfer the upper byte(s) 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 if the IF bit in the EFLAGS 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 4.10 80386 Signal Summary

Signal name	Signal function	Active state	Input/output	Input synch or asynch to CLK2	Output high impedance during HDLA?
CLK2	Clock	—	I	—	—
D0-D31	Data bus	High	I/O	S	Yes
BE0#-BE3#	Byte enables	Low	O	—	Yes
A2-A31	Address bus	High	O	—	Yes
W/R#	Write-read indications	High	O	—	Yes
D/C#	Data-control indication	High	O	—	Yes
M/IO#	Memory-I/O indication	High	O	—	Yes
LOCK#	Bus lock indication	Low	O	—	Yes
ADS#	Address status	Low	O	—	Yes
NA#	Next address request	Low	I	S	—
BS16#	Bus size 16	Low	I	S	—
READY#	Transfer acknowledge	Low	I	S	—
HOLD	Bus hold request	High	I	S	—
HLDA	Bus hold acknowledge	High	O	—	No
PEREQ	Coprocessor request	High	I	A	—
BUSY#	Coprocessor busy	Low	I	A	—
ERROR#	Coprocessor error	Low	I	A	—
INTR	Maskable interrupt request	High	I	A	—
NMI	Nonmaskable interrupt request	High	I	A	—
RESET	Reset	High	I	A (note)	—

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_{25} and t_{26}.

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:

80386 address signals							
Physical base address			80386 address pins and BE0#-BE3# signals				
A31—A2	A1	A0	A31—A2	BE3#	BE2#	BE1#	BE0#
A31—A2	0	0	A31—A2	X	X	X	Low
A31—A2	0	1	A31—A2	X	X	Low	High
A31—A2	1	0	A31—A2	X	Low	High	High
A31—A2	1	1	A31—A2	Low	High	High	High

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 then transfers data to or from 32- or 16-bit devices. During each bus cycle, the 80386 BS16# pin can be asserted for 16-bit ports or negate BS16# for 32-bit ports by the external device. With BS16# asserted all transfers are performed via D0-D15 pins. Also, with BS16# 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.

Asserting BS16# only affects the 80386 when BE2# and/or BE3# are asserted during the cycle. Assertion of BS16# 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 BS16# pin, depending in which byte enable pins are asserted during the current bus cycle. For example, asserting BS16# during "upper half only" reads causes 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 BS16#, BE0#-BE3#, and A2-A31 pins. Each 32-bit memory word starts at a byte address that is a multiple of 4. BS16# 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 BS16# 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 cycles).

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

80386 signals				16-bit bus signals			
BE3#	BE2#	BE1#	BE0#	A1	BHE#	BLE# (A0)	Comments
H*	H*	H*	H*	x	x	x	x — no active bytes
H	H	H	L	L	H	L	
H	H	L	H	L	L	H	
H	H	L	L	L	L	L	
H	L	H	H	H	H	L	
H*	L*	H*	L*	x	x	x	x — not contiguous bytes
H	L	L	H	L	L	H	
H	L	L	L	L	L	L	
L	H	H	H	H	L	H	
L*	H*	H*	L*	x	x	x	x — not contiguous bytes
L*	H*	L*	H*	x	x	x	x — not contiguous bytes
L*	H*	L*	L*	x	x	x	x — not contiguous bytes
L	L	H	H	H	L	L	
L*	L*	H*	L*	x	x	x	x — not contiguous bytes
L	L	L	H	L	L	H	
L	L	L	L	L	L	L	

Note: BLE# asserted when D0-D7 of 16-bit bus is active; BHE# asserted when D8-D15 of 16-bit bus is active; A0 low for all even words; A0 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 BS16# (bus size 16) input pin near the end of the bus cycle. Assertion of BS16# indicates a 16-bit data bus, while negation of BS16# 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

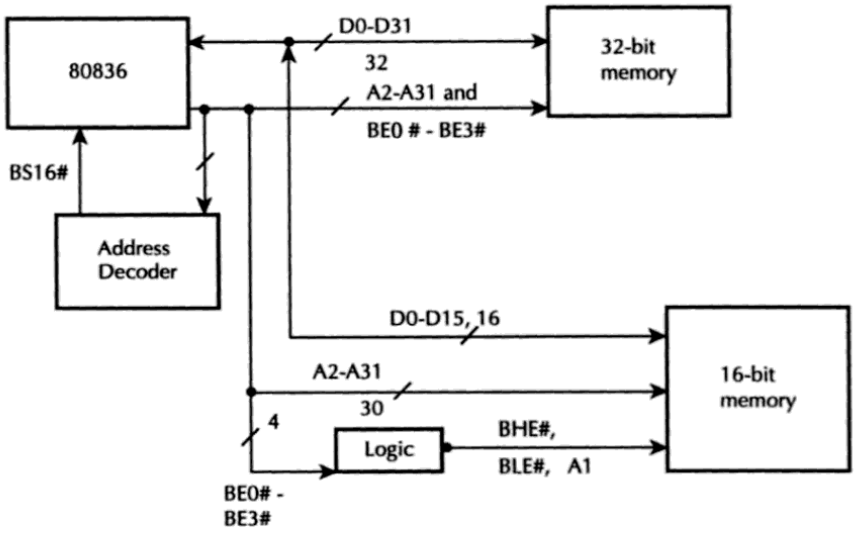


FIGURE 4.16 Interfacing 80386 16- and 32-bit memories.

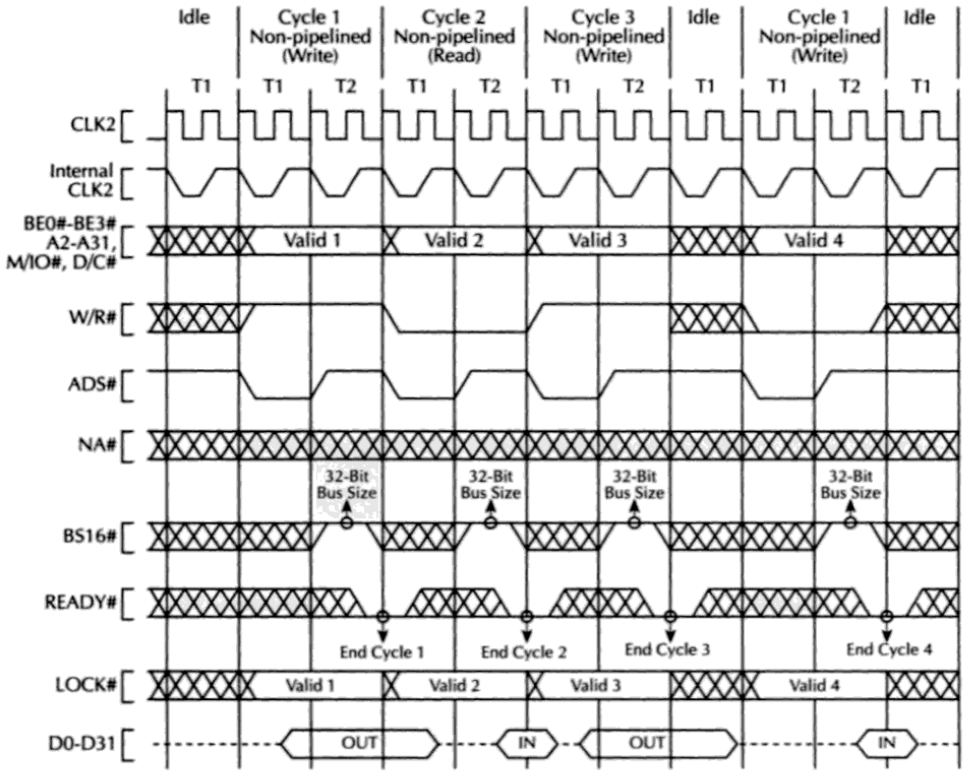


FIGURE 4.17 Bus cycles with nonpipelined address (zero wait states).

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 nonpipelined 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 ADS# 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 places data on the data bus starting in phase two of T1 until phase one of the bus state following cycle acknowledgment.

4.2.12 80386 Modes

The 80386 can be operated in real, protected, or virtual 8086 mode. These modes are described below.

4.2.12.a 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 this 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 memory. The address lines A2-A19, BE0#-BE3# are used by the 80386 in this mode. Paging is not provided in real

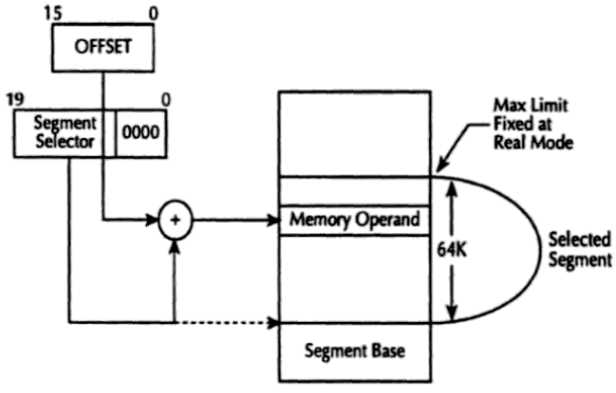


FIGURE 4.18 Real address mode addressing.

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 00000H thru 002FFH are reserved for the interrupt pointer table, while addresses FFFFFFF0H thru FFFFFFFFH 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:

Function	Interrupt number	Related instructions	Return address location
Interrupt table Limit too small	8	INT vector is not within table limit	Before instruction
Segment overrun exception	13	Word memory reference with offset = FFFFH or an attempt to execute an instruction past the end of a segment	Before instruction

4.2.12.b 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 tetra-bytes (2^{46} 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 with the offset.

With the paging mechanism enabled, the 80386 provides an additional memory management mechanism. 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.

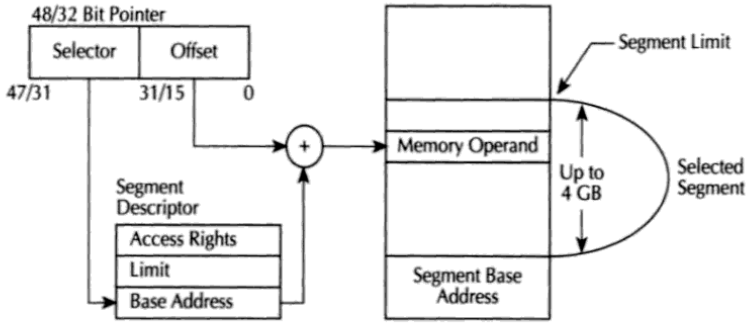


FIGURE 4.19 Protected mode addressing.

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 registers. The SGDT, SLDT, and SIDT 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.

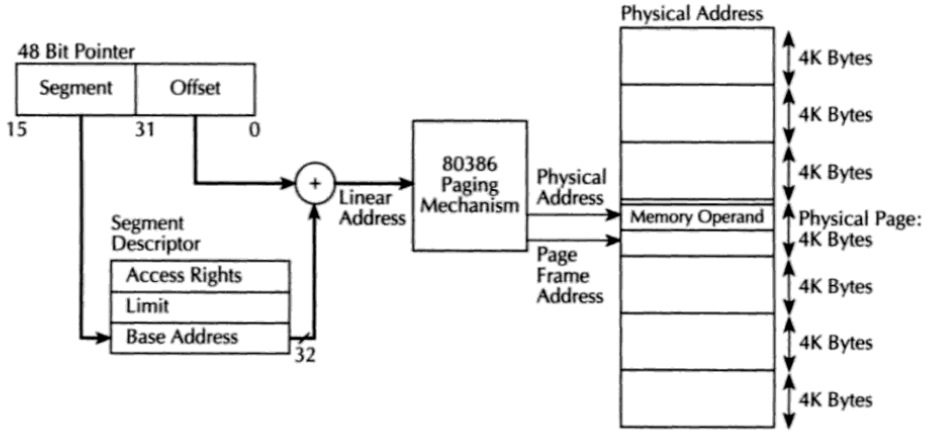


FIGURE 4.20 Paging and segmentation.

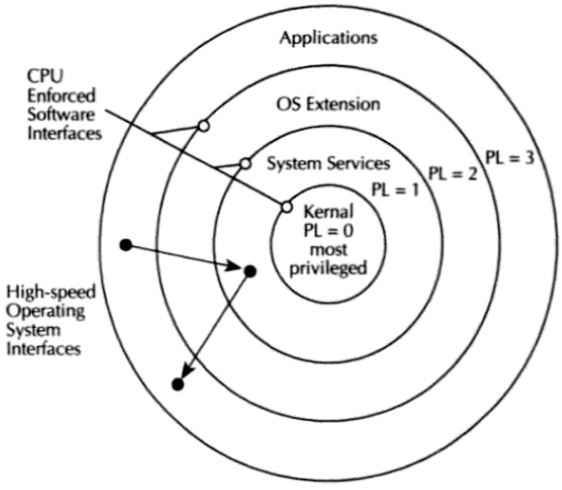


FIGURE 4.21 Four-level hierarchical protection.

The object to which the selector points 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 an 80386 operating system. This is possible because the 80386 automatically can differentiate between the 80286-type and 80386-type descriptors. In particular, if the upper word of a descriptor is zero, then that descriptor is an 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 multitasking 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:

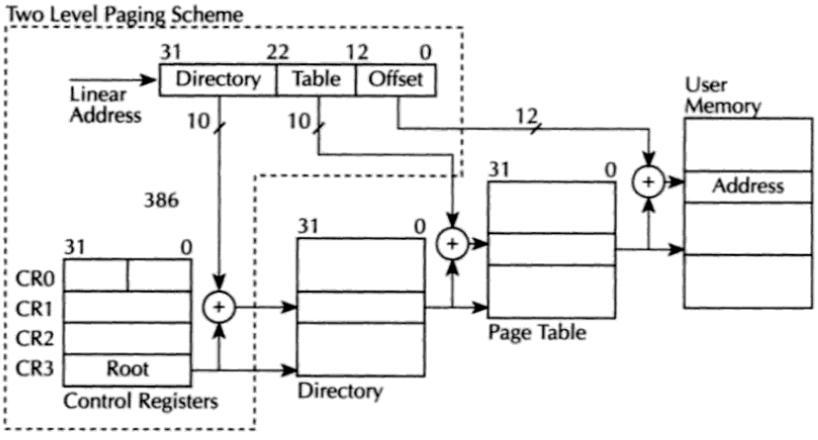


FIGURE 4.22 Paging mechanism.

- Data stored in a segment with a privilege level x can be accessed only by code executing at a privilege level at least as privileged as x.
- A code segment/procedure with privilege level x can only be called by a task executing at the same or a higher privilege level than x.

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). The 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.

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 16 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 long 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 long 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.c 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 applications. For example, in a multiuser 80386-based microcomputer, one person can run an MD-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.

4.3 80386 System Design

In this section, the 80386 is interfaced to typical memory and I/O chips. As mentioned in the last section the 80386 address and data lines are not multiplexed. There is a total of thirty address pins (A2-A31) on one chip. A0 and A1 are decoded internally to generate four byte enable outputs, BE0#, BE1#, BE2# and BE3#. In real mode, the 80386 utilizes 20-bit addresses and A2 through A19 address pins are active and the address pins A20 through A31 are used in real mode at reset, high for CS-based accesses, low for others, and always low after CS changes. In the protected mode, on the other hand, all address pins A2 through A31 are active.

In both modes, A0 and A1 are decoded internally. In all modes, the 80386 outputs on the byte enable pins to activate appropriate portions of the data bus to transfer byte (8-bit), word (16-bit), and double-word (32-bit) data as follows:

Byte Enable Pins	Data Bus
BE0#	D0-D7
BE1#	D8-D15
BE2#	D16-D23
BE3#	D24-D31

The 80386 supports dynamic bus sizing. This feature connects the 80386 with 32-bit or 16-bit data buses for memory or I/O. The 80386 32-bit data bus can be dynamically switched to a 16-bit bus by activating the BS16# input from high to low by a memory or I/O device. In this case, all data transfers are performed via D0-D15 pins. 32-bit transfers take place as two consecutive 16-bit transfers over data pins D0 through D15. On the other hand, the 32-bit memory or I/O device can activate the BS16# pin HIGH to transfer data over D0-D31 pins.

For reading a byte, the 80386 makes one of BE0#-BE3# active. For a word read (aligned:even address), the 80386 makes two byte enable outputs (BE0#-BE3#) active. On the other hand, for a 32-bit aligned read, the 80386 activates all byte enable outputs BE0#-BE3#.

The 80386 duplicates data on some 16-bit write operations in order to enhance performance of the data bus. This is illustrated in the table below:

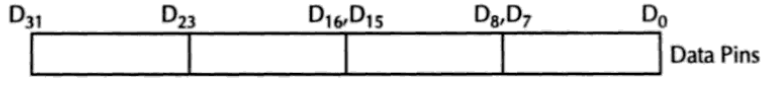
Transfer							Data Pins			
Size	A1	A0	$\overline{BE3}$	$\overline{BE2}$	$\overline{BE1}$	$\overline{BE0}$	D_{31}-D_{24}	D_{23}-D_{16}	D_{15}-D_8	D_7-D_0
Byte	0	0	1	1	1	0				X
Byte	0	1	1	1	0	1			X	
Byte	1	0	1	0	1	1		X		DD
Byte	1	1	0	1	1	1	X		DD	
Word	0	0	1	1	0	0			X	X
Word	0	1	1	0	0	1		X	X	
Word	1	0	0	0	1	1	X	X	DD	DD
Dword*	0	0	1	0	0	0		X	X	X
Dword*	0	1	0	0	0	1	X	X	X	
Dword	0	0	0	0	0	0	X	X	X	X

Note: DD = Data Duplication. Assumes 32-bit bus.

*For 32-bit misaligned transfers for addresses 3, 7, ... or addresses 1, 5, ..., these three-byte transfers along with one-byte from above are used to complete the 32-bit transfers in two cycles.

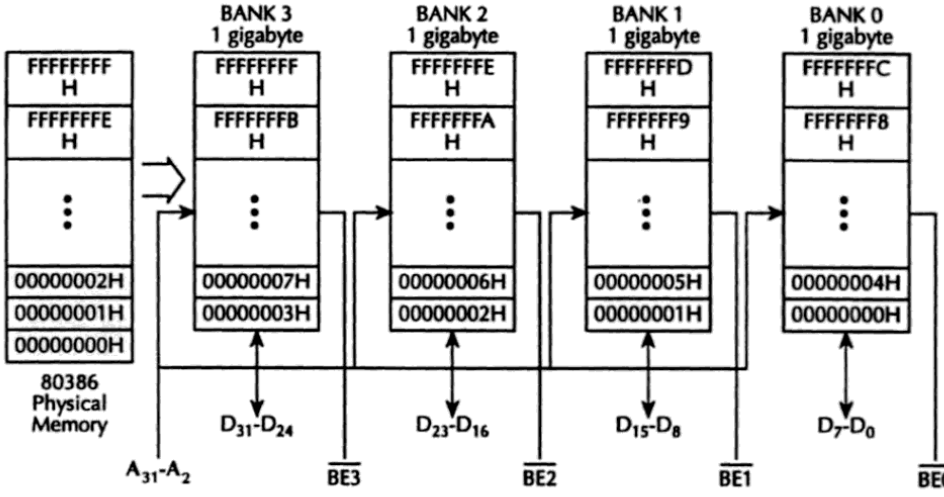
In the above table, the 80386 duplicates data for three cases. For example, in case of write cycle for byte when data is written on D_{23}-D_{16} pins, the same data is duplicated on D_7-D_0 pins. On the other hand, writing a byte on D_{31}-D_{24} pins, the same data is written on D_8-D_{15} pins. Finally when 16-bit data is written on D_{31}-D_{16} pins, the same data is duplicated by the 80386 on D_{15}-D_0 pins.

The 80386 address pins A1 and A0 specify the four addresses of a four byte (32-bit) word. Consider the following:



The contents of the memory addresses which include 0, 4, 8, ... with A1A0 = 00_2 are transferred by D_0-D_7. Similarly, the contents of addresses which include 1, 5, 9, ..., with A1A0 = 01_2 are transferred over D_{15}-D_8. On the other hand, the contents of memory addresses 2, 6, 10, ... with A1A0 = 10_2 are transferred over D_{16}-D_{23} while contents of addresses 3, 7, 11, ... with

A1A0 = 11_2 are transferred over D_{24}-D_{31}. Note that A1A0 is encoded from BE3 - $\overline{BE0}$ pins. The following figure depicts this:



In each bank, a byte can be accessed by enabling one of the byte enables, $\overline{BE0}$ - $\overline{BE3}$. For example, in response to execution of a byte-MOVE instruction such as MOV (00000006H), BL, the 80386 outputs low on BE2 and high on BE0, BE1 and BE3 and the content of BL is written to address 00000006H. On the other hand, when the 80386 executes a MOVE instruction such as MOV (00000004H), AX, the 80386 drives BE0 and BE1 to low. The contents of 00000004H and 00000005H are transferred from AL and AH via D_0-D_7 and D_8-D_{15} respectively. For 32-bit transfer, the 80386 executing a MOVE instruction from an aligned address such as MOV (00000004H), EAX, the 80386 drives all bus enable pins (BE0 - BE3) to low and written to the contents of four bytes (00000004H through 00000007H) from EAX. Byte (8-bit), aligned word (16-bit), and aligned double-word(32-bit) are transferred by the 80386 in a single bus cycle.

The 80386 performs misaligned transfers in two bus cycles. For example, the 80386 executing a misaligned word MOVE instruction such as MOV(00000003H), AX drives BE3 to low in the first bus cycle and writes the contents of 00000003H from bank 3 into AL in the first bus cycle. The 80386 then drives BE0 to low in the second bus cycle and writes the contents of 00000004H from bank 0 into AH in the second bus cycle. This transfer takes two bus cycles.

A 32-bit misaligned transfer such as MOV (00000002H), EAX, on the other hand, takes two bus cycles. In the first bus cycle, the 80386 enables BE2 and BE3, and writes the contents of address 00000002H and 00000003H from banks 2 and 3 respectively into low 16-bits of EAX. In the second cycle, the 80386 enables BE0 and BE1 to low and then writes the contents of address 00000004H and 00000005H into the upper 16 bits of EAX.

4.3.1 80386 Memory Interface

Figure 4.23 shows the basic memory interface block diagram.

The bus control logic provides the control signals for READY#, NA#, address latches, data buffers, and memory chips. The bus control logic activates READY# to LOW to signal the completion of the 80386 bus cycle and also outputs low on NA# (Next Address Request) to

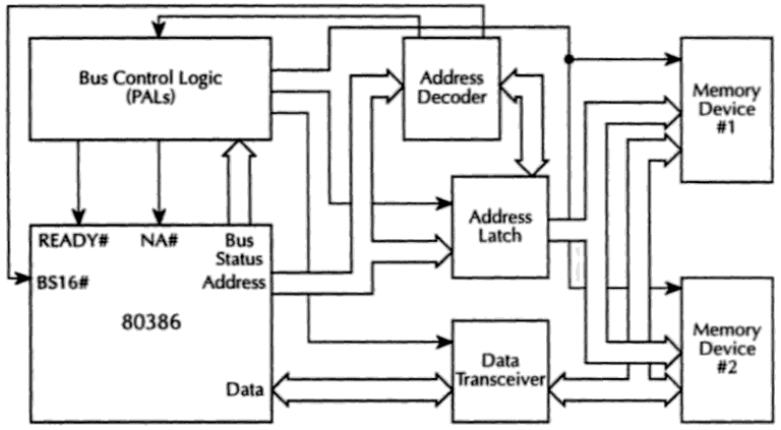


FIGURE 4.23 Basic memory interface block diagram.

activate address pipelining. The address decoder generates chip select signals and BS16# signal based on the address outputs of the 80386. These conventional memory interfacing concepts can be used to interface the 80386 to ROMs, EPROMs, and static RAMs.

The bus control logic decodes the 80386 status outputs (W/R#, M/IO#, and D/C#) and sends a command signal for the type of bus cycle according to Table 4.9 as per the following:

1. Memory read command (MRDC#) signal is generated from memory data read (M/IO#=1, D/C#=1, W/R#=0) or memory code read (M/IO#=1, D/C#=0, W/R#=0) cycle. $\overline{\text{MRDC}}$ commands the selected memory device to output data.
2. I/O read cycle (M/IO#=0, D/C#=1, W/R#=0) generates the I/O read command (IORC#) output. IORC# commands the selected I/O device to output data.
3. Memory write command (MWTC#) is obtained from memory write cycle (M/IO#=1, D/C#=1, W/R#=1). MWTC# tells the selected memory device to receive data on the data bus.
4. I/O write command (IOWC#) is obtained from the IO write cycle (M/IO#=0, D/C#=1, W/R#=1). IOWC# tells the selected I/O device to receive data on the data bus.
5. INTA# is generated from Interrupt Acknowledge cycle (M/IO#=0, D/C#=0, W/R#=0). INTA# is sent back to the Interrupt controller such as the 80259A. A second INTA# cycle tells a device such as the 8257A to place the interrupt vector on the bus. PALs can be used to design bus control logic.

Address latches maintain the 80386 address for the duration of the bus cycle and are required to pipeline address since the address for the next bus cycle appears on the address lines before the end of the current bus cycle. Latches such as 74 × 373 can be used. Note that the 80386 can be run without address pipelining to eliminate the need for address latching but the system will run inefficiently.

Standard 8-bit transceivers (74 × 245) provide isolation and additional drive currents for the 80386 data bus. Transceivers are necessary to prevent the contention on the data bus that occurs if some devices are slow to remove data from the data bus after a read cycle. Transceiv-

ers can be omitted if the data float time of the device is short enough and the load on the 80386 data pins meets device specifications. Figure 4.24 shows memory interface of a typical 80386-based microcomputer.

Three 74AS373 octal latches are used to maintain the address for the duration of the bus cycle. The 74AS373 latch Enable (LE) input is controlled by the 80386 $\overline{ALE}$ output. The 74AS373 output Enable (OE#) is always active.

Two 74F139 decoders (2 to 4) are used. The top 74F139 decoder has $\overline{A31}$ and $M/\overline{IO}$ as inputs. When the 80386 executes a memory-oriented instruction, it outputs HIGH on the M/IO pin and if A31 is one, then the Y2 output of the decoder goes to LOW. The Y2 output of the decoder is connected to the chip select 1 wait state (CS1WS) of the PAL 16R8B. The CS1WS is also latched by the 74AS373 and is used as the chip select for the 27128 EPROMs. Note that the 80386 reset vector is included in high memory (FFFFFFF0H). Therefore, the memory maps for EPROMS must include these high memory addresses. Also, when $M/\overline{IO}$=1 and A31=0, the upper decoder generates chip select 0 wait state (CS0WS). This signal is latched and used as chip select for 651628, 2KX8 static RAMs.

The address pins, byte enable signals (BE3-BE0), chip select outputs of the decoder and W/R are latched by the 74AS373 at the falling edge of ALE. Note that 80386 address lines A1 through A14 are used to address each EPROM.

The PAL 1 and PAL2 devices are used to design the bus control logic. PAL1 follows the 80386 bus cycles and generates the overall bus cycle timing. PAL2 generates most of the control signals required for memory interface. The PAL equations are given in Intel 80386 hardware reference manual. These equations can be assembled by a PAL assembler program. The assembled code to the PAL is then applied by using a standard PROM programmer with a special PAL enhancement. The write control logic for the SRAMs are implemented by using 74F32 quad OR gates. The memory write command (MWTC) is applied to one input of each one of the OR gates. The other input to each one of the OR gates is the appropriate byte enable (BE3-BE0) signal. For example, when both MWTC and BE3 are low, then $\overline{WE}$ input of the left most static RAM is enabled and data can then be written to that RAM.

Four 74AS245 transceivers are used to provide buffering of the data lines. Then $\overline{DEN}$ output generated by PAL2 is used to enable the transceivers and the latched 80386 W/R is used to control the direction of data transfer.

Two 27128's are used to provide 16K × 16 of EPROM. These chips are connected to the 80386 D0-D15 pins. The 27128 requires 14-bit address, A0–A13. MRDC (Memory Read Command) output generated by PAL2 is used to enable the 27128 $\overline{OE}$ Chip select signal (CS1WS) from the input of PAL1 is latched and then converted to $\overline{CS}$ inputs of the 27128's. CS1WS is also connected to the 80386 BS16 input to indicate to the 80386 that all read cycles are 16-bit.

The 27128 memory map can be determined as follows:

Memory Map for 27128-1 ODD

A31 A30 . . . . . A15 A14 . . . . . A1 A0
1 1

don't cares assume 1's — all zeros to ones

= FFFF8001H, FFFF8003H, ... , FFFFFFFFH.

Memory Map for 27128-1 EVEN

FFFF8000H, FFFF8002H, ... , FFFFFFFEH

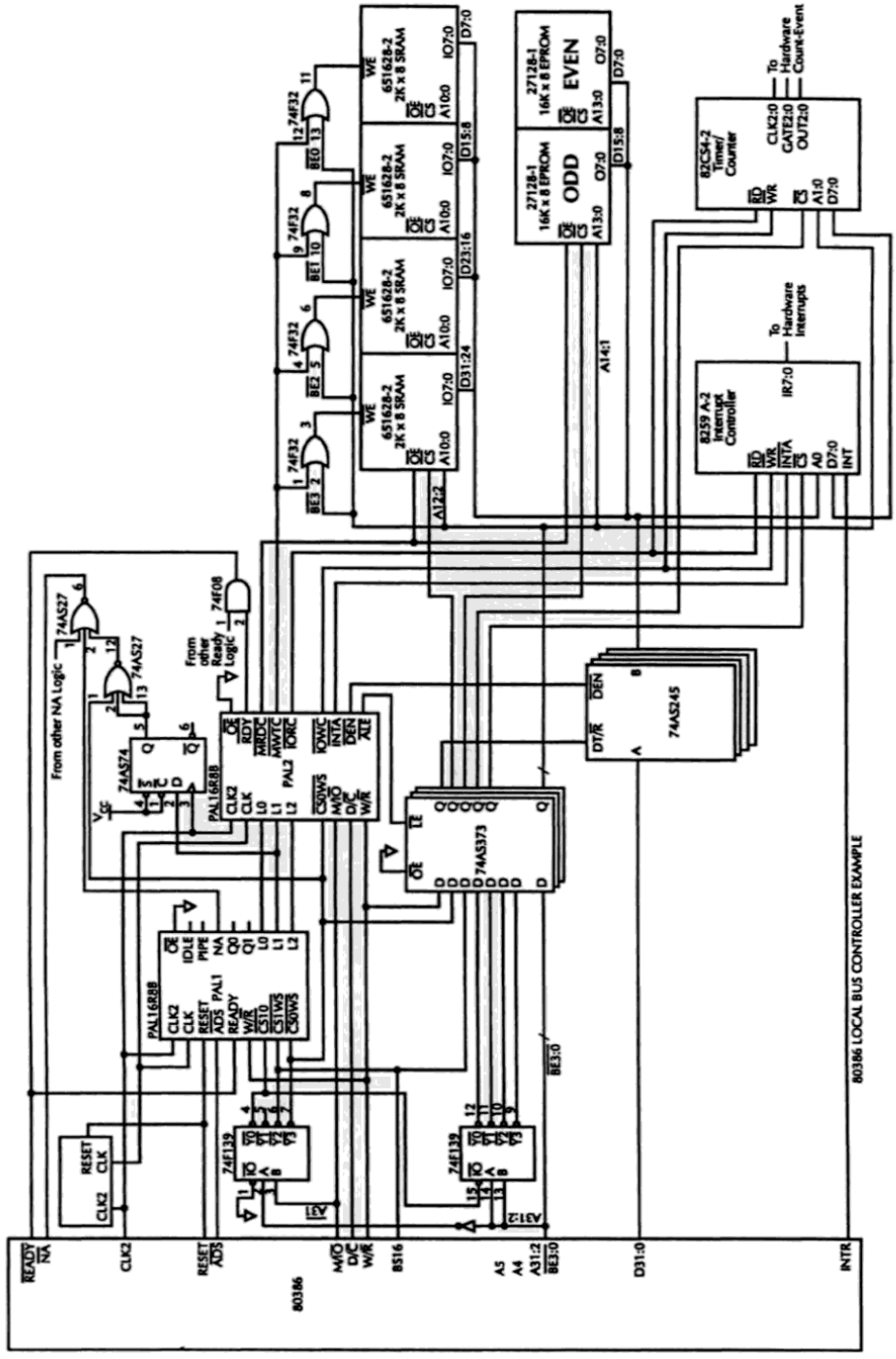


FIGURE 4.24 Basic memory interface for an 80386-based microcomputer. Note: A1 is generated for 27128 EPROMs according to Table 4.11.

Four 2K × 8 static RAM chips are used to provide a total of 8K bytes of RAM storage. Each RAM has 11-bit address input connected to the 80386 A12-A2 pins. When data is read from SRAMs, the MRDC signals activate all four RAMs and the 32-bit data from the selected location is placed on the 80386 D_{31}-D_0 pins. The 80386 then reads a byte, word, or double word from the bus.

During write to SRAMs, the 80386 may not output a 32-bit word. The 80386 in this case outputs the appropriate data size (byte, word or double word). It also sends appropriate BE0-BE3 signals to indicate which portion of the data bus carries the data. MWTC and WE signals ensure that data are written into the appropriate SRAM. Note that SRAMs are enabled by the CS0WS signal. This means that data read and write bus cycle do not require any wait states.

Let us determine the memory map for RAMs:

BANK 0 A31 A30 . . . A13 A12 . . . A2 A1 A0
0 (don't cares, assume 1's) (all zeros to ones) 0 0
= 7FFFE000H, 7FFFE004H, ... ,
BANK 1 7FFFE001H, 7FFFE005H, ...
BANK 2 7FFFE002H, 7FFFE006H, ...
BANK 3 7FFFE003H, 7FFFE007H, ...

When an 80386-based microcomputer system uses a large main memory of several megabytes, it is usually designed with high capacity, slow-speed dynamic RAMs and EPROMs. Although access times of DRAMs and EPROMs can be as fast as 60ns and 120ns respectively, the 80386 microcomputer system with these chips is expensive and too slow for running with zero wait states. The details of the 80386's interfacing to DRAMs can be found in Intel manuals and is beyond the scope of this chapter.

An 80386 microprocessor at a speed of 25MHz would require DRAMs at 40ns access time for zero wait-state. This is why, for large memory, wait states are introduced in all bus cycles while accessing memory. These degrade the overall performance of the 80386-based microcomputer.

A cache memory subsystem can be implemented in the 80386-based microcomputer to improve overall system performance while utilizing inexpensive, slow-speed DRAMs in main memory.

A cache memory subsystem includes a small amount of fast static RAM and a large amount of DRAM. The cache memory subsystem contains a fast SRAM between the 80386 and the slower main memory (DRAMs) along with a cache controller. The cache controller such as Intel 82385 includes the logic to implement the cache memory. Cache sizes are either 32K bytes or 64K bytes.

One of the two most widely used cache organizations, namely the direct-mapped cache or two-way set associative cache, is utilized in an 80386 system. Details of the 80386 cache implementation are provided in the 80386 hardware reference manual.

4.3.2 80386 I/O

The 80386 can use either a standard I/O or a memory-mapped I/O technique.

The address decoding required to generate chip selects for devices using standard I/O is often simpler than that required for memory-mapped devices. But, memory-mapped I/O offers more flexibility in protection than standard I/O does.

The 80386 can operate with 8-, 16-, and 32-bit peripherals.

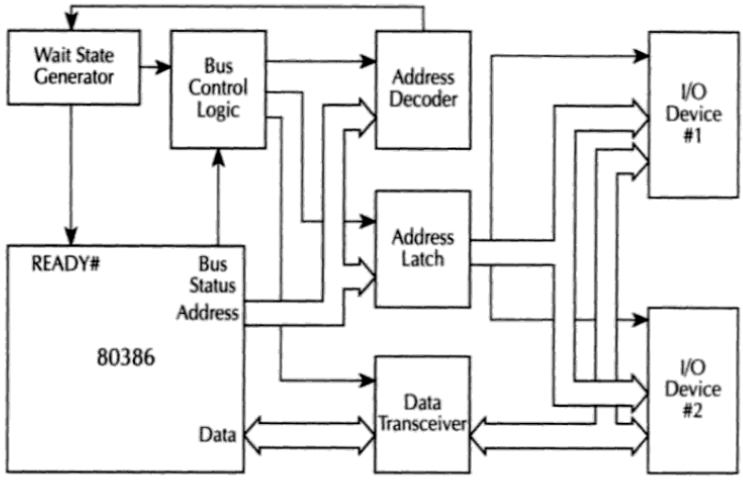


FIGURE 4.25 Basic I/O interface block diagram.

Eight-bit I/O devices can be connected to any of the four 8-bit sections of the data bus. Figure 4.25 shows 80386 I/O interface block diagram.

In the figure, the 80386's 32-bit data bus is multiplexed onto an 8-bit bus.

For efficient operation, 32-bit I/O devices should be assigned to addresses that are even multiples of four. If I/O devices are located on adjacent word boundaries, address decoding must generate the BS16# signal so that the 80386 performs a 16-bit bus cycle.

The block diagram is very similar to the 80386 memory interface block diagram. The purpose of various blocks in the figure has already been explained earlier in this section.

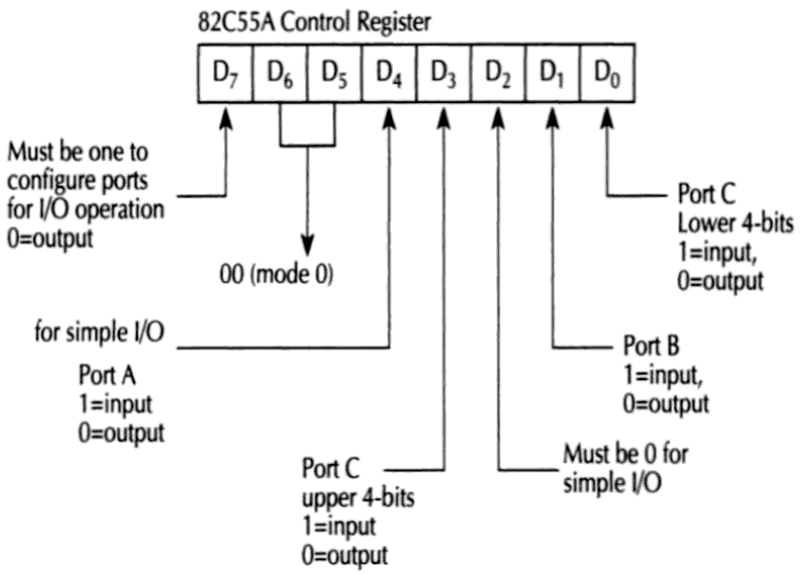
For standard I/O, the 80386 includes three types of I/O instructions. These are direct, indirect, and string I/O instructions which include the following:

```
Direct
  For 8-bit:    IN AL, PORT
                OUT PORT, AL
  For 16-bit:   IN AX, PORT
                OUT PORT, AX
  For 32-bit:   IN EAX, PORT
                OUT PORT, EAX
Indirect
  For 8-bit:    IN AL, DX
                OUT DX, AL
  For 16-bit:   IN AX, DX
                OUT DX, AX
  For 32-bit:   IN EAX, DX
                OUT DX, EAX
String
  For 8-bit:    INSB,   (ES:DI) ← ((DX))
                        DI ← DI ± 1
                OUTSB   ((DX)) ← (ES:SI)
                        SI ← SI ± 1
  For 16-bit:   INSW,   (ES:DI) ← ((DX))
                        (DI) ← DI ± 2
                OUTSW   (ES:SI) ← ((DX))
```

```
                    (SI) ← SI ± 2
For 32-bit:  INSD,  (ES:EDI) ← ((DX))
                    EDI ← EDI ± 4
             OUTSD, ((DX)) ← (ES:ESI)
                    ESI ← ESI ± 4
```

The 82C55A Programmable Peripheral Interface (PPI) can be interfaced with the 80386 for obtaining parallel ports for either standard or memory-mapped I/O. The pin diagram and the features provided by the 82C55A are same as the 8255 which were described in Chapter 3. The 82C55A will be interfaced to the 80386 for simple I/O operation.

In summary, the 82C55A contains three 8-bit parallel ports namely Port A, Port B, and Port C. These ports can be configured as input or output ports by writing 1 or 0 respectively in the corresponding bits in the control register. The **control register** bits can be defined as follows:



For example, outputting 89H to the control register will configure Ports A and B as output ports and Port C as an input port.

The 82C55A ports and control register are selected by the two-bit register select inputs (A0 and A1 inputs of the 82C55A) as follows:

A1	A0	
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Control Register

Figure 4.26 shows 82C55A-80386 interface using standard I/O.

Now let us determine the I/O port addresses in the 82C55A-4.

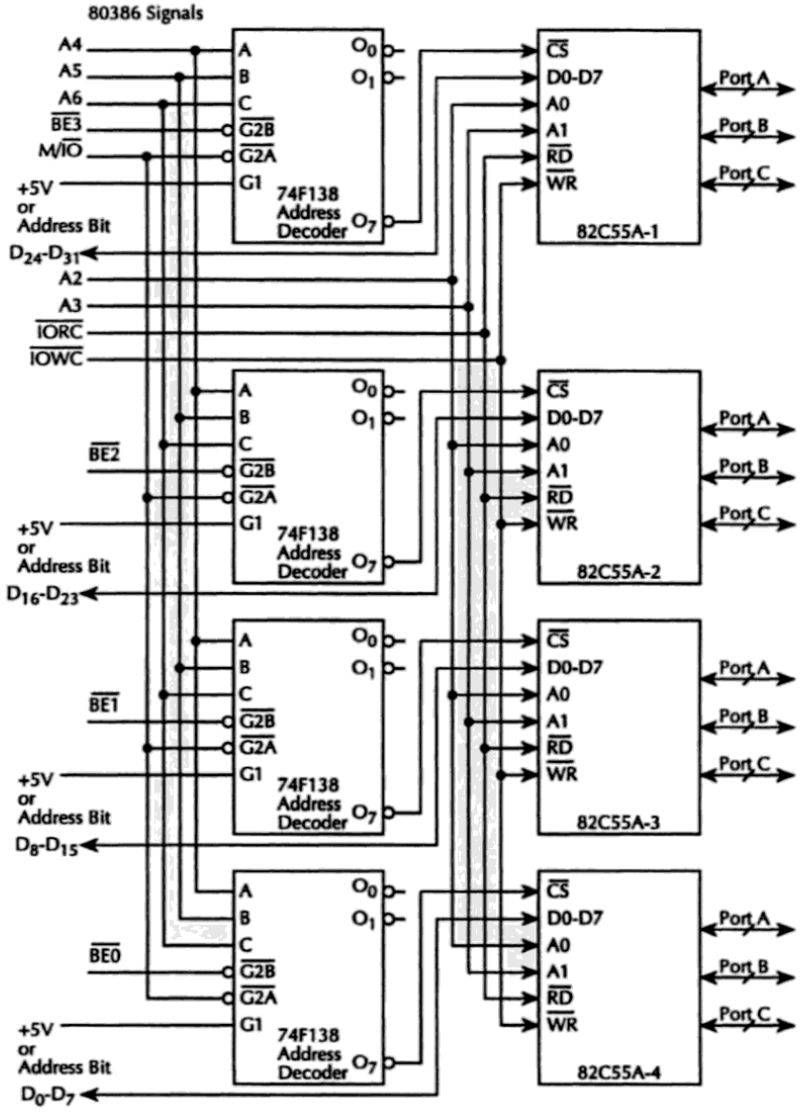


FIGURE 4.26 80386-82C55A interface using standard I/O.

A7	A6	A5	A4	A3	A2	A1	A0
X	1	1	1	0	0	0	0
↑							
don't care				Port A			
assume 1					=F0H		

Note that, since $\overline{\text{BE0}}$ is used to select the 82C55A-4, 80386 A_1 and A_0 pins will be zeros.

Therefore in the 82C55A-4 the port addresses can be obtained as follows:

8-bit Address	Port Name
F0H	Port A
F4H	Port B
F8H	Port C
FCH	Control Register

The seven other unused outputs of each address decoder can be used to enable three 82C55As in each category. Similarly, the port addresses for the other 82C55A's can be obtained as follows:

	8-bit Address	Port Name
	F1H	Port A
82C55A-3	F5H	Port B
	F9H	Port C
	FDH	Control Register
	F2H	Port A
82C55A-2	F6H	Port B
	FAH	Port C
	FEH	Control Register
	F3H	Port A
82C55A-1	F7H	Port B
	FBH	Port C
	FFH	Control Register

Note that $\overline{\text{IORC}}$ and $\overline{\text{IOWC}}$ are the outputs of PAL2 of the 80386 memory interface of Figure 4.24.

Using memory-mapped I/O, a read-mode 80386 can be interfaced to 82C55's in Figure 4.26 by connecting MRDC and MWTC outputs of PAL2 of Figure 4.24 to RD and WR of the 82C55A respectively. The G2A input line of the 74F138 can be connected through an inverter to unused 80386 address line such as A19, G1 can be connected to the 80386 M/IO pin through an inverter so that when M/IO=0 and A19=1, I/O ports are selected. The unused outputs 0_0 through 0_6 of the 74F138 decodes can be connected to other peripherals. The schematic of Figure 4.26 can be changed to memory-mapped I/O by connecting each of the 82C55's as above. Note that M/IO=1 and A19=0 can be used to select memory chips.

Example 4.14

Write an 80386 assembly language program to input two 32-bit data via Ports 5274H and 5270H, logically OR them together and then output the 32-bit result to Port 5270H.

Solution

```
MOV   DX, 5274H   ;  Initialize DX
IN    EAX, DX     ;  Input first 32-bit data
MOV   EBX, EAX    ;  Save data
MOV   DX, 5270H   ;  Initialize DX
IN    EAX, DX     ;  Input second 32-bit data
```

```
OR    EAX, EBX    ; Or the two data
OUT   DX, EAX     ; Output to Port 5270H
HLT
```

Example 4.15

Prior execution of the 80386 INSD instruction, assume the following data:

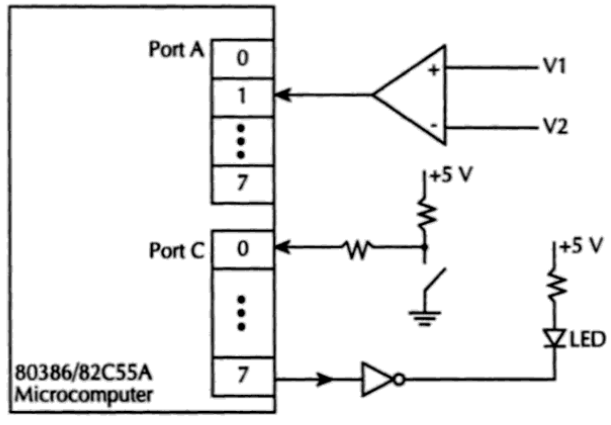
```
EDI = 00000032H
Contents of address [ES:EDI] = 37124426H
DX = 0020H, DF = 0
Contents of PORT 0020H = EEEF2752H
```

What are the contents of EDI, [ES:EDI], DX, DF, and PORT 0020H after execution of the INSD?

Solution

```
DF=0, EDI=00000036H
Contents of [ES:EDI]=EEEF2752H
DX=0020H
Contents of PORT 0020H=EEEF2752H
```

Example 4.16



For the above, write an 80386 assembly language program such that if $V_1 > V_2$, input the switch via bit 0 of 82C55A Port C and if the switch is closed, turn the LED ON.

Solution

```
MOV   AL, 90H     ; Configure Port A as input, Port C
                  ; lower
```

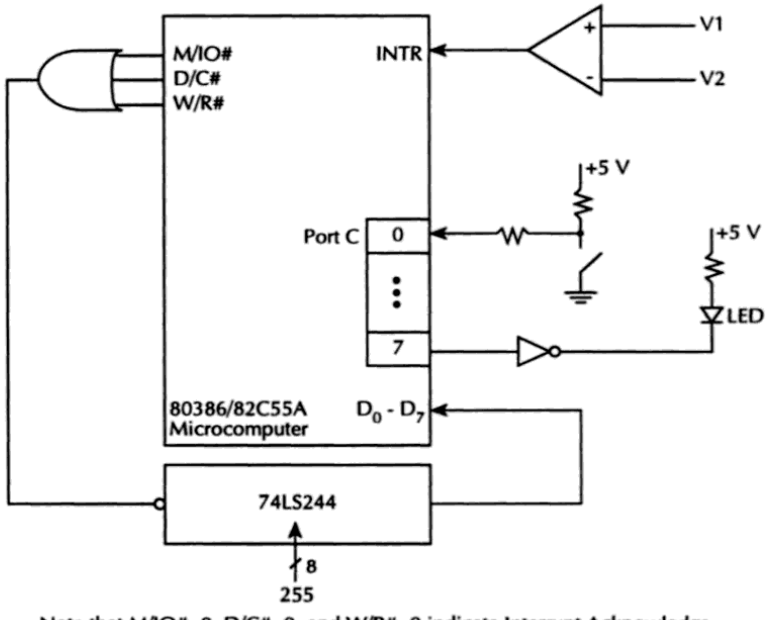
```
        OUT   CNTRL, AL    ; 4-bit as input and upper 4-bit as
                           ; output
CHK:    IN    AL, PORTA    ; Input Port A
        AND   AL, 02H      ; check V1 > V2
        CMP   AL, 02H      ; check if V1 > V2
        JNZ   CHK          ; Loop if V1 < V2
        IN    AL, PORT C   ; Input switch
        BTC   AL, 0        ; complement bit 0 of Port C
        RCR   AL,2
        OUT   PORT C, AL   ; output to Port C
        HLT
```

Example 4.17

Repeat Example 4.16 using interrupt I/O. Assume that the comparator output is connected to 80386 INTR pin. Use INT255.

Solution

The block diagram for servicing interrupt is as follows:



Note that M/IO#=0, D/C#=0, and W/R#=0 indicate Interrupt Acknowledge

Assume all segment registers are initialized. Write service routine at IP=3000H and CS=0200H.

Main Program

```
        MOV   SP, VALUE    ; Initialize SP
        STI                ; Enable Interrupts
        MOV   AL, 90H      ; Configure
```

```
        OUT   CNTRL, AL    ;  Port C lower 4-bit as input,
                           ;  upper 4-bit as output
WAIT    JMP   WAIT         ;  wait for interrupt
        HLT
Service Routine
        ORG   30000200H
BEGIN:  IN    AL, PORT C   ;  Input switch
        BTC   AL, 0        ;  complement switch
        RCR   AL, 2
        OUT   PORT C, AL   ;  output to LED
        IRET
        ORG   000003FCH
        DD    02003000H    ;  Initialize IP = 0200H and
                           ;  CS = 3000H
```

4.4 Coprocessor Interface

The performance of an 80386 system is enhanced by addition of a numeric coprocessor such as the 80287 or the 80387. The numeric coprocessor executes numeric instructions in parallel with the 80386. The 80386 automatically passes on these instructions to the coprocessor as it encounters them through I/O ports 800000F8H and 800000FCH.

The 80287 performs 16-bit transfers with a 16-bit data bus while the 80387 performs 32-bit transfers with a 32-bit data bus. The 80387 provides additional enhancements to the 80287 and includes full compatibility with the IEEE Floating-Point Standard draft 10.

The 80387 can utilize seven types of data using any 80386 addressing mode. These 80387 data types are listed in the following:

1. 32-bit Short real
2. 64-bit Long real
3. 80-bit Temporary real
4. 16-bit Word Integer
5. 32-bit Short Integer
6. 64-bit Long Integer
7. 80-bit Packed decimal

The 32-bit short real (single precision) format contains the 23-bit significand (bits 0-22), 8-bit biased exponent (bits 23-30), and the sign bit (bit 31). The 64-bit Long real (double precision) includes the 52-bit significand (bits 0-51), the 11-bit biased exponent (bits 52-62), and the sign bit (bit 63). The 80-bit Temporary real (extended precision) contains the 64-bit significand (bits 0-63), the 15-bit biased exponent (bits 64-78), and the sign bit (bit 79).

The 80-bit packed decimal contains 18 decimal digits (bits 0-71) and the sign bit (bit 79). Bits 72 thru 78 are undefined.

The 16-bit word integer includes the 15-bit integer (bits 0-14) along with the sign bit (bit 15) while the 32-bit short integer contains the 31-bit integer (bits 0-30) and the sign bit (bit 31). Finally, the 64-bit Long integer contains the 63-bit integer (bits 0-62) along with the sign bit (bit 63).

4.4.1 Coprocessor Hardware Concepts

The 80386 samples its ERROR# input during initialization to determine which coprocessor is present. The 80287 and 80387 require different interfaces and therefore, slightly different protocols.

For coprocessor cycles, the address pin A_{31} is HIGH. I/O addresses 800000F8H and 800000FCH are used for transferring data to and from a coprocessor. The 80386 automatically generates these addresses for coprocessor instructions. Also, the 80386 provides chip-select signals for the coprocessor using A_{31}=1 and M/IO#=0.

The 80386 utilizes three input signals for controlling data transfers with coprocessors. These are BUSY#, Coprocessor Request (PEREQ), and ERROR#.

The BUSY# indicates that the coprocessor is presently executing an instruction and therefore cannot accept another instruction. A new instruction, therefore, cannot overrun the execution of the current coprocessor instruction.

PEREQ indicates that the coprocessor needs to perform data transfer to or from memory. Since the coprocessor is never a bus master, all I/O transfers are performed by the 80386.

If a coprocessor math instruction results in an error that is not masked by the coprocessor's control register, the ERROR# signal is asserted. The coprocessor data sheets describe these errors and explain how to mask them by writing programs.

The interfacing characteristics of the 80386 with the 80387 Numeric coprocessor is described in the following.

The 80387 runs at an internal clock frequency of up to 16MHz. It is designed to run either fully synchronously or pseudo synchronously with the 80386. In the pseudo synchronous mode, the interface logic of the 80387 runs with the 80386 clock signal while internal logic runs with a different clock signal.

Figure 4.27 shows a typical 80386-80387 interface schematic. The main interfaces are described below:

- The 80386 and 80387 BUSY#, ERROR#, and PEREQ signals are directly connected.
- The 82384 RESET output is connected to the 80386 RESET and 80387 RESET IN signals.
- The 80387 chip select inputs, NPS1# and NPS2 are respectively connected to the 80386 M/IO# and A31. With M/IO#=0 and A31=1, the 80386 selects the coprocessor.
- The command input (CMD#) of the 80387 distinguishes data from commands. The 80386 A2 is connected directly to the 80387 CMD#. The 80386 outputs address 800000FCH when reading or writing data while address 800000F8H is used when writing a command or reading status.
- The 80387 uses READY# and ADS# pins to track bus activity and determine when W/R#, NPS1#, NPS2 and status Enable (STEN) can be sampled. STEN is an 80387 chip select and is pulled HIGH. If multiple 80387's are used by one 80386, STEN can be used to activate one 80387 at a time.
- Ready out (READY0#) is an optional signal that can be used to generate the wait states required by a coprocessor. When the 80386 encounters a coprocessor instruction, it automatically generates one or more I/O cycles to addresses 800000F8H and 800000FCH. The 80386 performs all bus cycles to memory and transfers data to and from the 80387. All 80387 transfers are 32-bit wide. For 16-bit memories, the 80386 automatically performs the necessary conversion before transferring data with the 80387.

Read cycles (transfer from 80387 to the 80386) require at least one wait state while write cycles to the 80386 require no wait states. This requirement is automatically reflected in the state of the 80387 READY0# output which can be used to generate the required wait states.

Upon hardware reset, the 80386 before executing the first instruction, checks its ERROR# input to determine the type of coprocessor present. If the 80386 samples ERROR# low, it assumes that an 80387 is present. On the other hand, a high ERROR# input indicates either an 80287 is present or no coprocessor is used.

The coprocessor type can be determined via software by checking the ET bit in the machine status word. The 80387 is present if ET=1.

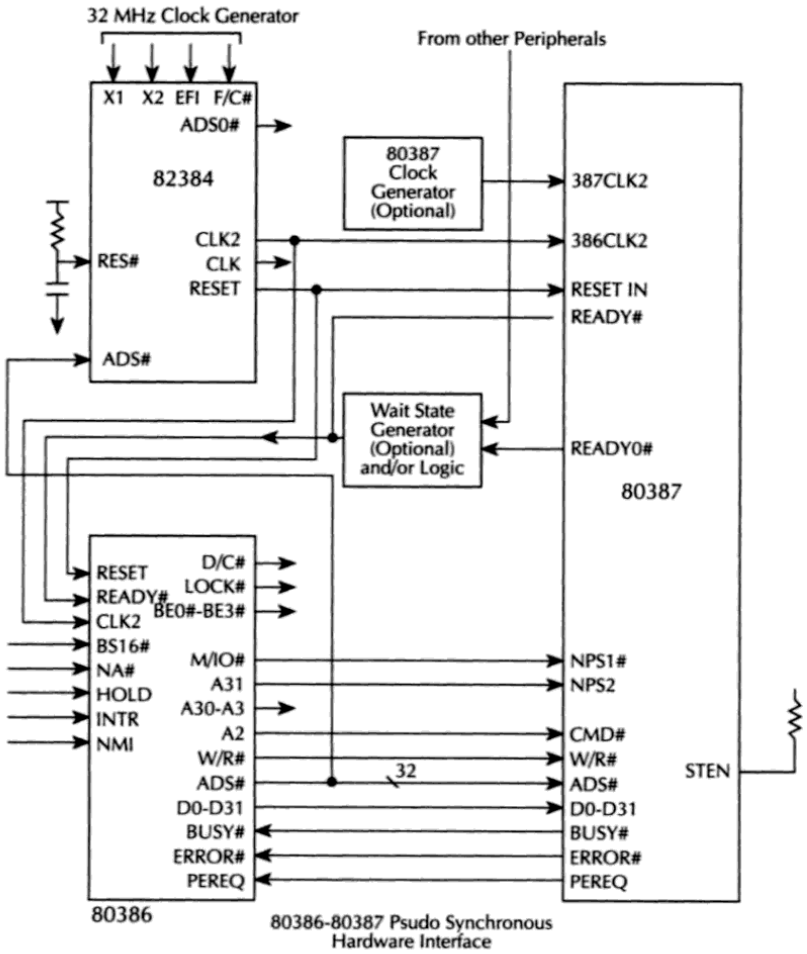


FIGURE 4.27 80386 system with 80387 coprocessor.

If the 80386 finds that an 80387 is present, an 80386 program must be written to execute the FNINT instruction to reset 80387 ERROR# output before any coprocessor transaction occurs.

When 80386 finds that either an 80287 is present or a coprocessor is not used, an 80386 program must be executed to determine the presence of an 80287 in order to set an internal status. An example of an 80386 instruction sequence to check the presence of an 80287 is given below:

```
;Initialization Routine to Detect An 80287 Numeric Processor
FND_287:    FNINIT              ;  INITIALIZE NUMERIC
                                ;  PROCESSOR
```

```
            FSTSW   AX          ;  RETRIEVE 80287 STATUS WORD
            OR      AL,AL       ;  TEST LOW-BYTE 80287
                                ;  EXCEPTION FLAGS. IF ALL
                                ;  ZERO, THEN 80287 PRESENT AND
                                ;  PROPERLY INITIALIZED.
                                ;  IF NOT ALL ZERO, THEN 80287
            JZ      GOT_287     ;  ABSENT. BRANCH IF
                                ;  80287 PRESENT
            SMSW    AX          ;  NO NUMERIC PROCESSOR
            OR      AX,04H      ;  SET EM BIT IN MACHINE
            LMSW    AX          ;  STATUS WORD TO ENABLE
                                ;  SOFTWARE EMULATION OF 80287
            JMP     CONTINUE
GOT_287:    SMSW    AX          ;  NUMERIC PROCESSOR PRESENT
            OR      AX,02H      ;  SET MP BIT IN MACHINE
            LMSW    AX          ;  STATUS WORD TO PERMIT
                                ;  NORMAL 80287 OPERATION
CONTINUE:                       ;  AND OFF WE GO ....
```

In the above instruction sequence, the 80386 assumes that the 80287 is present. Therefore, it executes an FNINT instruction. Next, the 80386 reads the 80287 status word. If an 80287 is present, the lower 8 bits of this word (the exception flags) are all zeros. If an 80287 is not present, these data lines are floating. If a pull-up resistor is connected to at least one of these lines, the absence of the 80287 is confirmed by at least one high bit in the lower eight bits of the status word. The routine then sets or resets the Emulate Coprocessor (EM) bit of the CR0 register of the 80386, depending on whether or not the 80287 is present.

4.4.2 Coprocessor Registers

Figure 4.28 shows the 80387 registers.

The 80387 contains three types of registers:

1. Eight 80-bit floating point stack registers
2. One 16-bit status word, one 16-bit control word and one 16-bit tag register
3. Four 32-bit error-pointer registers namely FIP, FCS, FOO, and FOS which store the instruction and memory operand causing an exception

The 80386 floating-point instructions consider the eight 80-bit registers as a stack of accumulators. The current stack top is called ST or ST(0). After a LOAD or PUSH, ST(0) becomes ST(1) and all other ST(i)'s are incremented by one. After a STORE or POP, ST(1) before the POP becomes the new ST(0), and all ST(i)'s are decremented by one. A 3-bit field name TOP in the status word defines the register number ST(0) of the current stack top. If TOP=000_2, a push will decrement TOP to 111_2 and store a new value into ST(7). ST(7) is the present stack top. On the other hand, if TOP=111_2, a POP will read a value from ST(7) and then increment top to 000_2 so that ST(0) becomes the new top of stack.

The 16-bit tag register includes eight 2-bit fields—one for each physical floating point register. This two-bit field indicates whether the corresponding physical floating-point register (0-7)

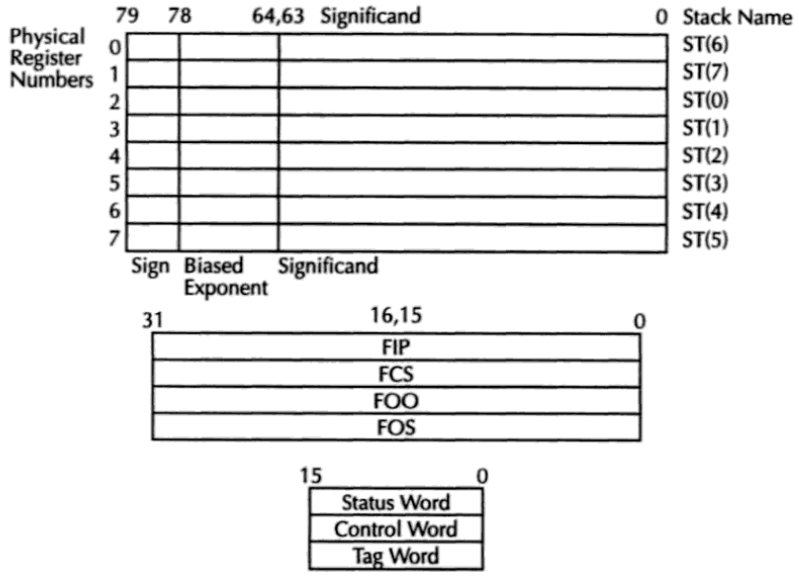


FIGURE 4.28 80387 registers.

rather than ST-relative names contains a valid, zero or special floating-point number(infinity, NAN) or is empty. The tag fields are used to indicate overflow or underflow of ST(i)'s. A stack overflow occurs if a PUSH decrements TOP to point to an ST(i) that is not empty. A stack underflow occurs if an empty ST(i) is popped. Stack underflow or overflow generates an invalid operation exception.

The control word register can be written by the program to control the 80387 operation. The control word contains exception mask bits for situations such as invalid operation, zero divide, overflow, and underflow. If the 80387 encounters an exception, the mask bit for that exception is checked to determine if the exception should be sent to a program error handler if the mask bit=0 or handled by an 80387 error handler if the mask bit=1.

The control word also contains information such as precision control and rounding control. The two-bit precision control field specifies that the results of arithmetic operations are rounded to 24-bit (short real), 53 bits (long real), or 64-bits (temporary real) precision before storing into the destination. All other operations use temporary real precision or a precision specified in the instruction.

On the other hand, the two bit rounding control field specifies that the results of floating-point operations are approximated or rounded to nearest toward minus infinity, plus infinity, or truncate toward zero.

The 80387 updates the bits of the status word register that can be checked by the program to check for special conditions. Bits 11, 12, and 13 specify the TOP field containing the three bit address of the physical register stack (0-7) corresponding to ST(0). The other bits in the status register include information such as indication of exception due to floating-point operations and floating-point condition code bits.

The four 32-bit 80387 Error Pointer register (FIP, FCS, FOO, and FOS) hold pointers to the last 80387 instruction executed along with its data. FCS stores the CS selector while FIP stores the IP offset. FOS, on the other hand, stores operand selector while FOO contains the operand offset.

4.4.3 80387 Instructions

The 80387 floating point instructions can be classified into six groups. These are data transfer, arithmetic, comparison, transcendental, loading, and control instructions. Some of these instructions are described in the following:

i) **Data Transfer Instructions**

-FLD mem or ST(n) — mem can be short, long or temporary real. This instruction first decrements stack pointer by one and then loads the specified real number onto the new stack top.

Example:

Consider FLD mem32. If prior to execution of this FLD, contents of mem32 = $4.1572 * 10^3$ with TOP = 4 then after FLD, contents of mem32 = $4.1572 * 10^3$, ST = $4.1572 * 10^3$ with TOP = 3

-FXCH ST(n) — This instruction exchanges the contents of ST(n) with the stack top.

Example:

Consider FXCH ST(5). If prior to execution of this instruction, ST = $5.71252 * 10^{-87}$, ST(5) = $-2.78164 * 10^{82}$ with TOP = 5 then after this FXCH, ST = $-2.78164 * 10^{82}$, ST(5) = $5.71252 * 10^{-87}$ with TOP = 5.

-FST mem or ST(n) — The stack top is stored into specified memory or ST(n). FTSP mem or ST(n) is identical to FST except that the stack is popped after transfer.

ii) **Arithmetic Instructions**

-FABS — This instruction converts the stack top to its absolute value.

Example:

If prior to execution of FABS, ST = $-5.372 * 10^{-300}$ then after FABS, ST = $5.372 * 10^{-300}$

-FADD
FADD ST(n) — Performs real addition.

- FADD; ST ← ST + ST(1)
- FADD ST(n); ST ← ST + ST(n)

Example:

If prior to execution of FADD,
ST=$3.17300 * 10^4$, ST(1) = $2.167 * 10^3$
then after FADD,
ST=$3.38970 * 10^4$, ST(1) = $2.167 * 10^3$

-FDIV — Performs real division.

FDIV: ST ← ST/ST(1)

FDIVP ST(n),ST: ST(n) ← ST(n)/ST. Stack is popped.

FDIV ST(n): ST ← ST/ST(n)

FIDIV src: ST ← ST/src. Integer divide
src may be ST(n) or integer

Example:

If prior execution of FDIV,
ST = $4.240 * 10^{-4}$, ST(1) = $8.480 * 10^{+3}$
then after FDIV,
ST = $5.000 * 10^{-8}$, ST(1) = $8.480 * 10^{+3}$

-FIMUL src — Integer multiply. ST ← ST * src

-FMUL: — Performs multiplication of real numbers.

FMUL: — ST ← ST(1) * ST

FMUL src — ST ← ST * src; src may be ST(n) or a real number

FMUL ST(n),src — ST(n) ←ST(n)*src; src may be ST(n) or a real number

Example:

If prior to execution of FMUL,
$ST = 2.500 * 10^{-10}$, $ST(1) = 2.500 * 10^{+7}$
then after FMUL,
$ST = 6.250 * 10^{-3}$, $ST(1) = 2.500 * 10^{7}$

-FSUB — Performs real subtraction

FSUB: — ST(1) ← ST(1) – ST

FSUB ST(n): — ST ← ST – ST(n)

Example:
If prior execution of FSUB,
$ST = 7.140 * 10^{10}$, $ST(1) = 8.217 * 10^{11}$
then after FSUB,
$ST(1) = 7.503 * 10^{11}$

-FSQRT — ST ← sqrt(ST)

Example
If prior execution of FSQRT,
$ST = 2.25 * 10^{6}$
then after FSQRT,
$ST = 1.5 * 10^{3}$

iii) Comparison Instructions

-FCOM ST(n) — Compares ST(n) numerically with top of stack. The condition codes C3, C2, and C0 (bits 14,10, and 8 in the status word) are set according to the following:

	C3	C2	C0
ST > ST(n)	0	0	0
ST < ST(n)	0	0	1
ST = ST(n)	1	0	0

iv) Transcendental Instructions

-F2XMI: — $ST \leftarrow 2^{ST} - 1$
The range of the values of ST prior execution of F2XMI is –0.5 to +0.5.

-FCOS — This instruction computes the cosine of ST. ST is assumed to contain real numbers in radians. The result replaces the original ST.

-FSIN — This instruction computes the sine of ST. ST is assumed to contain real number in radians. The result replaces ST.

-FSINCOS — ST ← COS(ST)
ST(1) ← SIN(ST)

-FYL2X — $ST = ST(1) * LOG_2(ST)$
The stack is popped and the new stack top is replaced with the result.

-FYL2XP1 — $ST = ST(1) * LOG_2(ST + 1.0)$
The stack is popped and the new stack top is replace with the result of this computation.

v) **Loading Instructions**

-FLD n — The stack is pushed. The constant value, n is loaded into the new top of stack (ST) according to the following:

FLD1	Load 1.0
FLDL2E	Load Log_2e
FLDL2T	Load Log_210
FLDLG2	Load $Log_{10}2$
FLDLN2	Load Log_e2
FLDPI	Load π
FLDZ	Load 0.0

vi) **Control Instructions**

-FLDCW mem16 — Loads the control word with the contents of mem 16.

Example:

If prior execution of FLDCW mem16,
CW = 2547H, mem16 = 25F2H
then after FLDCW mem16,
CW = 25F2H, mem16 = 25F2H.

Example 4.18

Write an 80386 assembly language program using 80387 floating point instructions to compute $Y = \sqrt{X^2 - Z^2}$.

Solution

```
FLD     Z          ; Load Z onto stack top
FMUL    ST, ST     ; Compute Z²
FLD     X          ; Load X onto stack
FMUL    ST, ST     ; compute X²
FSUB    ST, ST(1); Compute X² - Z²
FSQRT              ; Y=SQRT(X² - Z²)
HLT
```

Example 4.19

Write an 80386 assembly language program using 80387 floating point instructions to compute the volume of a sphere = (4/3) * π * r^3 where r is the radius of the sphere.

Solution

```
FLD     r          ;  Load r to stack top
FST     ST(1)      ;  Make a copy of r on stack
FMUL    ST, ST     ;  compute r²
FMUL    ST, ST(1)  ;  compute r³
FMUL    NUM        ;  Compute 4*r³
FDIV    NUM1       ;  Compute (4/3) * r³
FLDPI              ;  Load π
FMUL    ST, ST(1)  ;  Compute volume
HLT
```

The above program assumes NUM and NUM1, respectively store real numbers 4.0 and 3.0. Also, memory location r stores the radius.

QUESTIONS AND PROBLEMS

4.1 Write an 80186 assembly program to multiply a 16-bit signed number in BX by 00F3H. Assume that the result is 16 bits wide.

4.2 Identify the peripheral functional blocks integrated into the 80186.

4.3 What is the relationship between internal and external clocks of the 80186?

4.4 Identify the basic differences between 8086 and 80186.

4.5 Identify the main differences between the 80186 and 80286.

4.6 How much physical and virtual memory can the 80286 address?

4.7 What is the difference between the 80286 real address mode and PVAM? Explain how these two modes can be switched back and forth.

4.8 Explain how the 80286 determines where in memory the global descriptor table and the present local descriptor table are located.

4.9 Discuss briefly the 80286 protection mechanism.

4.10 Explain the meaning of 80286 call gates.

4.11 What is the purpose of 80286 CAP, COD/$\overline{\text{INTA}}$ pins?

4.12 Identify the 80286 pins used for interfacing it to a coprocessor.

4.13 Discuss the issues associated with isolating a user program from a supervisor program and then describe 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.

4.14 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.

4.15 What are the basic differences between the 80386 real, protected, and virtual 8086 modes?

4.16 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
```

4.17 Determine the effect of each of the following 80386 instructions:

i)
```
MOVZX ECX, BX
assume [ECX] = F1250024H
       [BX] = F130H
```

prior to execution of the MOVZX instruction.

ii)
```
SHLD CX, BX, 0
if [CX] = 0025H, [BX] = 0F27H
```

prior to execution of the SHLD instruction.

4.18 Given the following data prior to the execution of each of the following instructions, determine the contents of the specified registers and carry flag after execution:

i)
```
Prior to execution: [EAX]=0527AF12H
                    [EBX]=0071FFD1H
                    CF=1
After execution of BTC EAX, EBX
```

ii)
```
Prior execution: [BX]=F216H
                 CF=1
After execution of BTR BX, 20H
```

iii)
```
Prior to execution: [EDX]=F214721FH
                    CF=1
After execution of: BTS EDX, 35H
```

4.19 Write an 80386 assembly language program to divide a signed 64-bit number in EBX:EAX by a 16-bit signed number in AX. Store the 32-bit quotient and remainder in memory locations.

4.20 Write an 80386 assembly program to compute Xi^2/N where N = 100 and Xi's are signed 32-bit numbers. Assume that Xi^2 can be stored as a 32-bit signed number without overflow.

4.21 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 [ESI].

4.22 Find a single 80386 MOV instruction with the appropriate addressing mode to replace the following 80386 instruction sequence:

```
MOV CL, 3
SAL ESI, CL
MOV EAX, [ESI]
```

4.23 Write an 80386 assembly language program to compute the following: X = Y + Z – 20EEH where X, Y, and Z are 64-bit variables. The upper 32 bits of Y and Z are stored at 3000H and 3008H each followed by the lower 32 bits. Store the upper 32-bit of the 64-bit result at location at 4000H followed by the lower 32 bits.

4.24 Assume that the 80386 registers BL, CX, and EDX contain a signed byte, a signed word, and a signed 32-bit word in two's complement form respectively. Write an 80386 assembly language program that will generate the signed result of operation BL + CX – EDX → EDX.

4.25 Write an 80386 assembly language program to compute:

```
ECX=6*BX+EDX/EAX
```

where all numbers are signed numbers. Discard the remainder of EDX/EAX.

4.26 Write an 80386 assembly language program that checks all 32 bits of EAX from right to left (bit 0 to bit 31). The program will set the corresponding bit in EBX based on the location of the first one bit found in EAX. If the entire 32 bits of EAX are zero, clear EBX to all zeros.

4.27 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).

4.28 How does the 80386 generate the 32-bit physical address from A2-A31 and BE0#-BE3#?

4.29 What are the purposes of NA#, D/C#, BS16#, and ERROR# pins?

4.30 For 16- and 32-bit transfers, what is the logic level of the BS16# pin?

4.31 Discuss briefly the 80386 segmentation unit, paging unit, and protection.

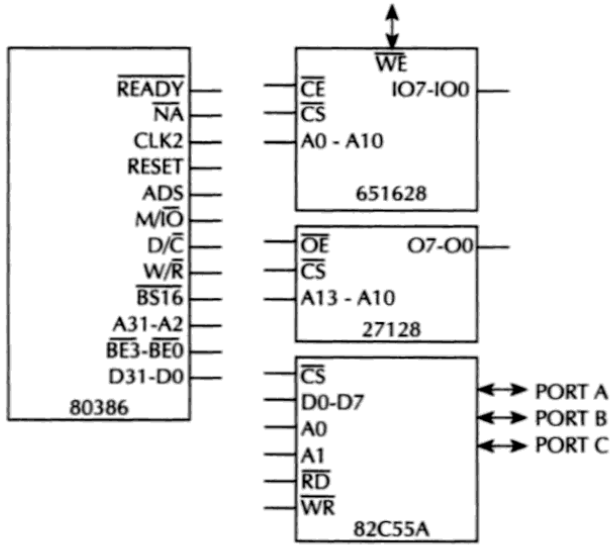
4.32 How many bits are required for the address in real and protected modes?

4.33 Discuss the basic differences between the 80286 and 80386 descriptors.

4.34 What is the four-level hierarchical protection in protected mode?

4.35 Discuss briefly the 80386 virtual mode.

4.36 Consider the following pins and signals:



Draw a simplified diagram connecting the above chips to include the following:

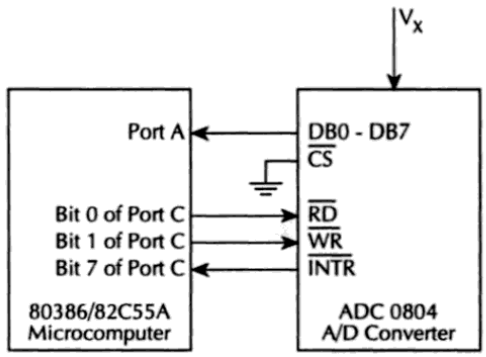
-The 651628 to contain addresses
XXXXXXX1H, XXXXXXX5H, XXXXXXX9H . . . . .
. . . . . .

-The 27128 to include addresses
XXXXXXX0H, XXXXXXX4H, XXXXXXX8H, ...
. . . . . .

-The 82C55A to contain port addresses
Port A = X1H
Port B = X5H
Port C = X9H

Assume all don't cares (X) to be zeros. Use PALs, latches, and other components as required. Determine memory and I/O maps.

4.37



The ADC 0804 can be started by sending a HIGH to LOW transition at the $\overline{WR}$ pin. The conversion is completed when INTR is LOW. The 8-bit data at the DB0-DB7 pins can be read by the microcomputer by sending a LOW at the RD pin.

Write an 80386 assembly language program to start the 0804 by the microcomputer and input the converted data via Port A.

4.38 Repeat problem 4.37 by using:

i) **NMI**
ii) **INTR**

4.39 Assume 80386/80387 system. Write an 80386 assembly language program using floating point instructions to compute the mass of a sphere with radius r and density of Log_e^2. Mass is equal to density times volume.

4.40 Assume 80386/80387 system. Write an 80386 assembly language program to compute the logarithm with a base other than 2; or log base 'n' of X (Log_nX).

MOHAMED RAFIQUZZAMAN

MICROPROCESSORS and MICROCOMPUTER-BASED SYSTEM DESIGN

Second Edition

Features

- New, detailed descriptions of the architectures, addressing modes, instruction sets, I/O, and system design concepts of the Intel 80486/80960 and Motorola 68040 microprocessors
- Overview of the Pentium and PowerPC microprocessors
- New topics such as floating-point arithmetic, Program Array Logic, and flash memories
- Sample design problems and solutions

CRC
4475
ISBN 0-8493-4475-1