MICROPROCESSORS

(**Common to CSE & ISE)**

Subject Code: 10CS45	**I.A. Marks : 25**
Hours/Week : 04	**Exam Hours: 03**
Total Hours : 52	**Exam Marks: 100**

PART A

UNIT – 1 **7 Hours**

Introduction, Microprocessor Architecture – 1: A Historical Background, The Microprocessor-Based Personal Computer Systems.

The Microprocessor and its Architecture: Internal Microprocessor Architecture, Real Mode Memory Addressing.

UNIT – 2 **7 Hours**

Microprocessor Architecture – 2, Addressing Modes: Introduction to Protected Mode Memory Addressing, Memory Paging, Flat Mode Memory

Addressing Modes: Data Addressing Modes, Program Memory Addressing Modes, Stack Memory Addressing Modes

UNIT – 3 **6 Hours**

Programming – 1: Data Movement Instructions: MOV Revisited, PUSH/POP, Load-Effective Address, String Data Transfers, Miscellaneous Data Transfer Instructions, Segment Override Prefix, Assembler Details.

Arithmetic and Logic Instructions: Addition, Subtraction and Comparison, Multiplication and Division.

UNIT - 4 **6 Hours**

Programming – 2: Arithmetic and Logic Instructions (continued): BCD and ASCII Arithmetic, Basic Logic Instructions, Shift and Rotate, String Comparisons.

Program Control Instructions: The Jump Group, Controlling the Flow of the Program, Procedures, Introduction to Interrupts, Machine Control and Miscellaneous Instructions.

PART B

UNIT - 5 **6 Hours**

Programming – 3: Combining Assembly Language with C/C++: Using Assembly Language with C/C++ for 16-Bit DOS Applications and 32-Bit Applications

Modular Programming, Using the Keyboard and Video Display, Data Conversions, Example Programs

UNIT - 6 **7 Hours**

Hardware Specifications, Memory Interface – 1: Pin-Outs and the Pin Functions, Clock Generator, Bus Buffering and Latching, Bus Timings, Ready and Wait State, Minimum versus Maximum Mode.

Memory Interfacing: Memory Devices

UNIT – 7 **6 Hours**

Memory Interface – 2, I/O Interface – 1: Memory Interfacing (continued): Address Decoding, 8088 Memory Interface, 8086 Memory Interface.

Basic I/O Interface: Introduction to I/O Interface, I/O Port Address Decoding.

UNIT 8 **7 Hours**

I/O Interface – 2, Interrupts, and DMA: I/O Interface (continued): The Programmable Peripheral Interface 82C55, Programmable Interval Timer 8254.

Interrupts: Basic Interrupt Processing, Hardware Interrupts: INTR and INTA/; Direct Memory Access: Basic DMA Operation and Definition.

Text Book:

1. Barry B Brey: The Intel Microprocessors, 8th Edition, Pearson Education, 2009.
   (Listed topics only from the Chapters 1 to 13)

TABLE OF CONTENTS

UNIT 1: INTRODUCTION TO MICOPROCESSOR & COMPUTER

The Microprocessor Age

• The world's first microprocessor was the Intel 4004.
• 4004 was a 4-bit microprocessor programmable controller on a chip.
• 4004 addressed a mere 4096, 4-bit-wide memory locations.
• The instruction-set contained only 45 instructions.
• 4004 was fabricated with P-channel MOSFET technology.
• 4004 executed instructions at the slow rate of 50 KIPs (kilo-instructions per second).
• When compared to 30-ton ENIAC computer, 4004 weighed much less than an ounce.
• Later in 1971, Intel released the 8008— an extended 8-bit version of the 4004 microprocessor.
• 8008 was the first of the modem 8-bit microprocessor.
• The 8008
  → addressed an expanded memory size (16K bytes) and
  → contained additional instructions (a total of 48)
• The main drawbacks of both 4004 and 8008:
  → slow speed
  → small word-width and
  → small memory-size
  → limited instruction set

What Was Special about the 8080?

• 8080 executed the instructions 10 times faster than the 8008.
• 8080 also addressed four times more memory (64Kbytes) than the 8008 (l6Kbytes).
• Also, 8080 was compatible with TTL, whereas the 8008 was not directly compatible. This made interfacing much easier and less expensive.

The 8085 Microprocessor

• In 1977, Intel introduced an updated version of the 8080—the 8085.
• Although only slightly more advanced than an 8080 microprocessor, the 8085 executed software at an even higher speed.
• The main advantages of the 8085:
  → internal clock generator
  → internal system controller and
  → higher clock frequency
• This higher level of component integration reduced the 8085's cost and increased its usefulness.

The Modern Microprocessor

• In 1978, Intel released the 8086 microprocessor; a year later, it released the 8088.
• Both devices are 16-bit microprocessors, which executed instructions in as little as 400 ns (2.5 MIPs).
• In addition, the 8086 and 8088 addressed 1Mbyte of memory, which was 16 times more memory than the 8085.
• This higher execution speed and larger memory size allowed the 8086 and 8088 to replace smaller minicomputers in many applications.
• One other feature found in the 8086/8088 was a small queue (or 6-byte instruction cache) that pre-fetched a few instructions before they were executed.
• The queue increased the speed of operation of many sequences of instructions.
• Improvements to the instruction-set included multiply and divide instructions, which were missing on earlier microprocessors.
• In addition, the number of instructions increased from 45 on the 4004 to well over 20,000 variations on the 8086 and 8088 microprocessors.
• The 16-bit microprocessor also provided more internal register storage space than the 8-bit microprocessor.
• The additional registers allowed software to be written more efficiently.
• The popularity of the Intel family was ensured in 1981, when IBM Corporation decided to use the 8088 microprocessor in its personal computer.

The 80286 Microprocessor

• The 80286 microprocessor was almost identical to the 8086 and 8088, except it addressed a 16 Mbyte memory system instead of a 1 Mbyte system.
• The instruction-set of the 80286 was almost identical to the 8086 and 8088, except for a few additional instructions that managed the extra 15 Mbytes of memory.
• The 80286 contains a memory management unit (MMU).
• The 80286 operates in both the real and protected modes.
→ In the real mode, the 80286 addresses a 1 MByte memory address space and is virtually identical to 8086
→ In the protected mode, the 80286 addresses a 16 MByte memory-space
• The 80286 is basically an 8086 that is optimized to execute instructions in fewer clocking periods than the 8086.

The 80386 Microprocessor

• The 80386 microprocessor is an enhanced version of the 80286 microprocessor.
• 80386 includes a MMU to provide memory paging.
• 80386 also includes 32-bit extended registers and a 32-bit address and data bus.
• 80386 has a physical memory size of 4GBytes that can be addressed as a virtual memory with up to 64 TBytes.
• 80386 is operated in the pipelined mode, it sends the address of the next instruction to the memory system prior to completing the execution of the current instruction.
• 80386 operates in both the real and protected modes.
→ In the real mode, the 80386 addresses a 1MByte memory address space and is virtually identical to 8086
→ In the protected mode, the 80386 addresses a 4 Gbytes memory space

TABLE 1–2 Many modern Intel and Motorola microprocessors.

Manufacturer	*Part Number*	*Data Bus Width*	*Memory Size*
Intel	8048	8	2K internal
	8051	8	8K internal
	8085A	8	64K
	8086	16	1M
	8088	8	1M
	8096	16	8K internal
	80186	16	1M
	80188	8	1M
	80251	8	16K internal
	80286	16	16M
	80386EX	16	64M
	80386DX	32	4G
	80386SL	16	32M
	80386SLC	16	32M + 8K cache
	80386SX	16	16M
	80486DX/DX2	32	4G + 8K cache
	80486SX	32	4G + 8K cache
	80486DX4	32	4G + 16 cache
	Pentium	64	4G + 16K cache
	Pentium OverDrive	32	4G + 16K cache
	Pentium Pro	64	64G + 16K L1 cache + 256K L2 cache
	Pentium II	64	64G + 32K L1 cache + 256K L2 cache
	Pentium III	64	64G + 32K L1 cache + 256K L2 cache
	Pentium 4	64	64G+32K L1 cache+ 512K L2 cache (or larger) (1T for 64-bit extensions)
	Pentium4 D (Dual Core)	64	1T + 32K L1 cache + 2 or 4 M L2 cache
	Core2	64	1T + 32K L1 cache + a shared 2 or 4 M L2 cache
	Itanium (Dual Core)	128	1T + 2.5 M L1 and L2 cache + 24 M L3 cache

The 80486 Microprocessor

• The 80486 is an improved version of the 80386 that contains
→ an 8K-byte cache and
→ an 80387 arithmetic co processor
• 80486 executes a few new instructions that control the internal cache memory.
• A new feature found in the 80486 is the BIST(builtin self-test) that tests the microprocessor, coprocessor, and cache at reset time.
• Additional test registers are added to the 80486 to allow the cache memory to be tested.
• These new test registers are TR3 (cache data), TR4 (cache status), and TR5 (cache control).

Pentium Microprocessor

• The Pentium microprocessor is almost identical to the earlier 80386 and 80486 microprocessors.
• The main difference is that the Pentium has been modified internally to contain
→ a dual cache (instruction and data) and → a dual integer unit
• The Pentium also operates at a higher clock speed of 66 MHz.
• The data bus on the Pentium is 64–bits wide and contains eight byte-wide memory banks selected with bank enable signals.
• Memory access time, without wait states, is only about 18 ns in the 66 MHz Pentium.
• The superscalar structure of the Pentium contains three independent processing units:
→ a floating point processor and → two integer processing units
• Pentium contains a new mode of operation called the System Memory Management (SMM) mode. It is intended for high-level system functions such as power management and security.
• Another feature found in the Pentium is the BIST(built-in self-test) that tests the microprocessor, coprocessor, and cache at reset time.
• The Pentium allows 4MByte memory pages instead of the 4Kbyte pages.

Pentium Pro Microprocessor

• The Pentium Pro is an enhanced version of the Pentium microprocessor that contains
→ level 1 caches found inside the Pentium
→ level 2 cache of 256 K or 512K found on most main boards
• The Pentium Pro operates using the same 66 MHz bus speed as the Pentium and the 80486.
• The only significant software difference between the Pentium Pro and earlier microprocessors is the addition of FCMOV and CMOV instructions.
• The only hardware difference between the Pentium Pro and earlier microprocessors is the addition of 2M paging and four extra address lines that allow access to a memory address space of 64G Bytes.
• The Pentium uses an internal clock generator to multiply the bus speed by various factors to obtain higher internal execution speeds.

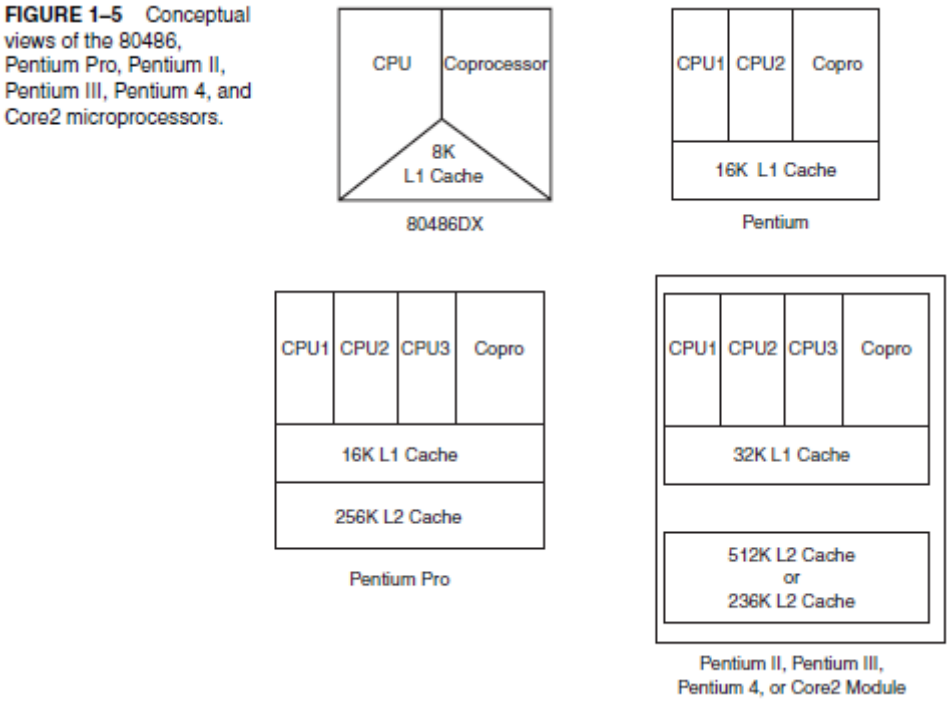


FIGURE 1–5 Conceptual views of the 80486, Pentium Pro, Pentium II, Pentium III, Pentium 4, and Core2 microprocessors.

Pentium II and Pentium Xeon Microprocessors

• Released in 1997, the Pentium II was an adaptation of the Pentium Pro aimed at the general public.
• Instead of being an integrated circuit as with prior versions of the microprocessor, Intel has placed the Pentium II on a small circuit board.
• The main reason for the change is that the L2 cache found on the main circuit board of the Pentium was not fast enough to function properly with the Pentium II.
• On the Pentium system, the L2 cache operates at the system bus speed of 60 MHz or 66 MHz.
• New features compared to the Pentium Pro were essentially MMX (SIMD) support and a doubling of the Level 1 cache.

Pentium III Microprocessor

• Released in 1999, the Pentium III microprocessor uses a faster core than the Pentium II, but it is still a P6 or Pentium Pro processor.
• It is also available in the slot 1 version mounted on a plastic cartridge and a socket 370 version called a flip-chip, which looks like the older Pentium package.
• Another difference is that the Pentium III is available with clock frequencies of up to 1 GHz.
• The slot 1 version contains a 512K cache and the flip-chip version contains a 256K cache.
• The speeds are comparable because the cache in the slot 1 version runs at one-half the clock speed, while the cache in the flip-chip version runs at the clock speed.
• Both versions use a memory bus speed of 100 MHz, while the Celeron7 uses memory bus clock speed of 66 MHz.

Pentium 4 and Core2 Microprocessors

• In late 2000, Intel announced its new processor, the Pentium 4.
• The most recent version of the Pentium is called the Core2 by Intel.
• The Pentium 4 and Core2, like the Pentium Pro through the Pentium III, use the Intel P6 architecture.
• The main difference is that
  → The Pentium 4 is available in speeds to 3.2 GHz and faster and
  → The chip sets that support the Pentium 4 use the RAMBUS or DDR memory technologies in place of once standard SDRAM technology.
• The Core2 is available at speeds of up to 3 GHz. These higher microprocessor speeds are made available by an improvement in the size of the internal integration, which at present is the 0.045 micron or 45 nm technology.
• It is also interesting to note that Intel has changed the level 1 cache size from 32K to 8K bytes and most recently to 64K.

The Microprocessor-Based Personal Computer System

• The block diagram of personal-computer is composed of 3 blocks that are interconnected by buses:
  1) Memory system
  2) Microprocessor and
  3) I/O system (Figure 1-6).
• A bus is the set of common connections that carry the same type of information.

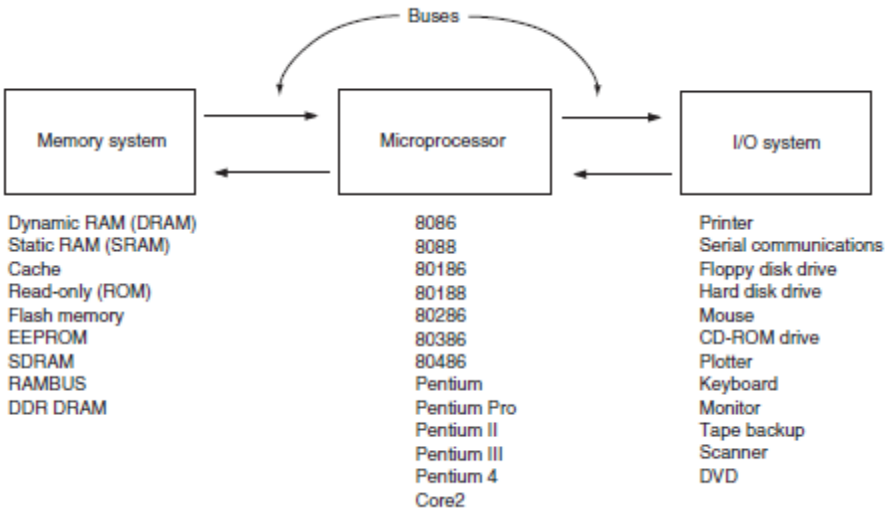


Figure 1-6: The block diagram of a microprocessor-based computer system

The Memory & I/O System

- Normally, the memory contains 3 main parts:
  1) TPA(Transient Program Area)
  2) System Area &
  3) XMS(Extended Memory System)
- If the computer is based upon 8086, the TPA and system-area exist, but there is no extended memory-area(Fig:1-7).
- The PC contains
  → 640KB of TPA &
  → 384KB of system memory
- The first 1Mbyte of memory is called real(conventional) memory because each microprocessor is designed to function in this area by using its real-mode of operation.
- If the computer is based upon 80286, then 1)TPA 2)System area 3)Extended memory exist. These machines are called AT class-machines. (Sometimes, these machines are also referred to as ISA machines <Industry Standard Architecture>)
- In the 80286, extended-memory contains up to 15MB (in addition to the first 1Mbyte of real memory).
- The ISA machine contains an 8-bit peripheral-bus. The bus is used to interface 8-bit devices to the 8086-based computer.
- In ATX class-systems, following 3 newer buses exist:
  1) USB(Universal Serial Bus) is used to connect peripheral-devices(such as keyboards and mouse) to the microprocessor through a serial data-path and a twisted-pair of wires.
  (Main idea: To reduce system cost by reducing the number of wires).
  2) AGP(Advanced Graphics Port) is used to transfer data between the video-card and the microprocessor at higher speeds(533 MB/s)
  3) SATA(Serial ATA) is used to transfer data from the PC to the hard-disk drive at rate of 150MB/s.

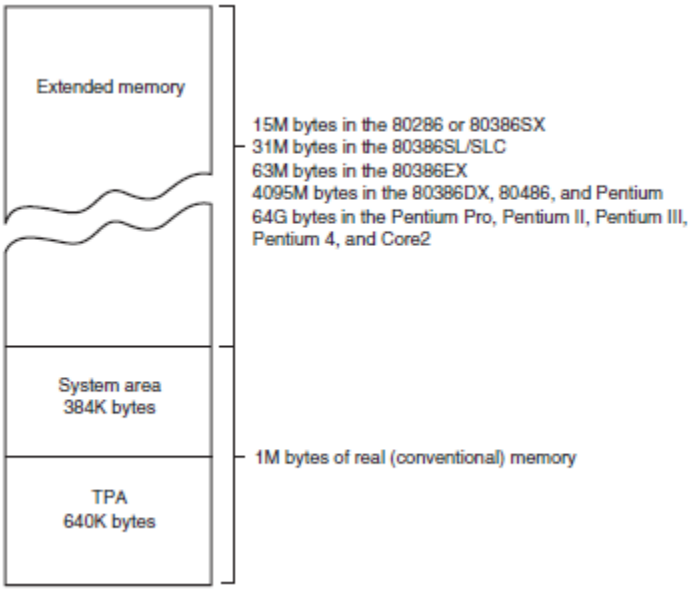


Figure 1-7: The memory map of a personal computer

TPA (Transient Program Area)

- This holds i)DOS, ii)Application-programs and iii)Drivers that control the computer.
- Length of TPA = 640KB (Fig: 1-8).
- The memory-map of TPA contain following parts:

1) The *Interrupt-vectors* access various features of the DOS, BIOS and applications.
2) The *system BIOS* is a collection of programs stored in either a ROM or flash memory. This operates many of the I/O devices connected to the computer.
3) The *DOS communications areas* contain transient-data used by programs to access
→ I/O devices and
→ internal features of the computer.
4) The *IO.SYS* contains programs that allow DOS to use the keyboard, video-display, printer and other I/O devices.
5) *Device-drivers* are programs that control installable I/O devices such as mouse, keyboard, CD-ROM memory, DVD.
6) The *COMMAND.COM program*(command processor) controls the operation of the computer from the keyboard when operated in the DOS mode.

The System Area

- This contains memory used for video-cards, disk-drives and the BIOS ROM.
- This contains
→ programs on either a ROM or flash memory &
→ areas of RAM for data-storage.
- The area at locations C8000H-DFFFFH is open(or free). This area is used for the expanded memory system(EMS) in the computer.
- The EMS allows a 64Kbyte page-frame of memory to be used by application-programs.
- Finally, the *system BIOS ROM* controls the operation of the basic I/O devices connected to the computer.

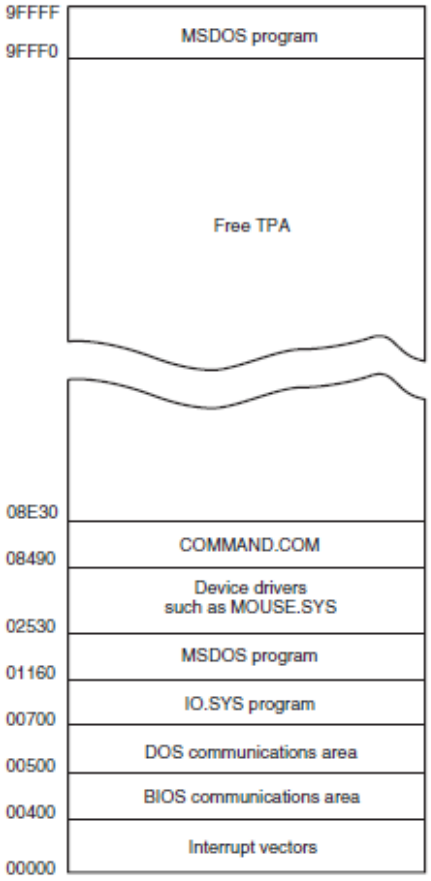


Figure 1-8:The memory map of the TPA in a PC

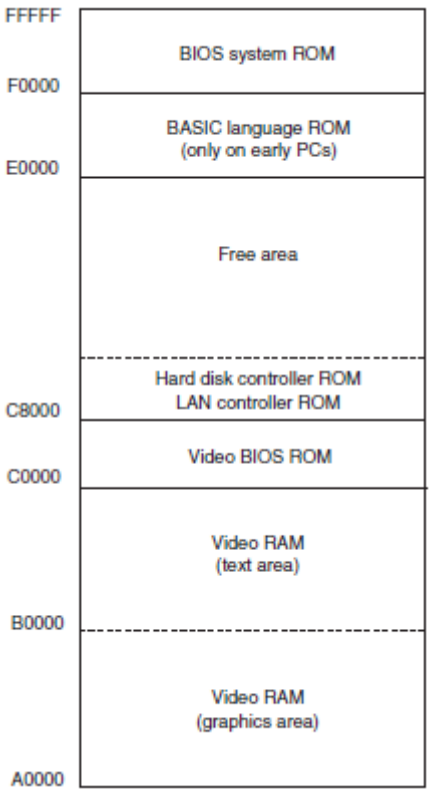


Figure 1-9:The system area of a typical PC

I/O Space

• An *I/O port* is similar to a memory-address, except that instead of addressing memory, it addresses an I/O device.
• The I/O devices allow the microprocessor to communicate between itself and the outside world.
• The I/O space extends from I/O port 0000H to port FFFFH.
• The I/O space allows the computer to access
→ up to 64 different 8-bit I/O devices or
→ up to 32K different 16-bit devices.

The Microprocessor

• The *microprocessor*(or CPU) is the controlling-element in a computer.
• The microprocessor controls memory and I/O through buses.
• Memory and I/O are controlled through instructions that are
→ stored in the memory and
→ executed by the microprocessor
• The buses
→ select an I/O or memory-device
→ transfer data between an I/O(or memory) and the microprocessor &
→ control the I/O and memory.
• The microprocessor performs 3 main tasks for the computer:
1) Data transfer between itself and the memory or I/O
2) Simple arithmetic & logic operations
3) Program flow via simple decisions
• The microprocessor executes programs(stored in the memory) to perform complex operations in short periods of time. (The arithmetic and logic operations are very basic, but through them, very complex problems are solved).
• Data are operated upon from the memory or internal registers. Data widths are variable and include a byte(8 bits) and word(16 bits).
• Microprocessor is powerful because of its ability to make simple decisions based upon numerical facts.
• Simple arithmetic and logic operations includes: Addition, Subtraction, Multiplication, Division, AND, OR, NOT, Shift & Rotate
• Decisions found in the 8086 microprocessor are
→ Zero: Test a number for zero or not-zero
→ Sign: Test a number for positive or negative
→ Carry: Test for a carry after addition or a borrow after subtraction
→ Parity: Test a number for an even or an odd number of 1's

Buses

- A bus is a common group of wires that interconnect components in a computer (Fig 1-12)
- The buses transfer address-, data- and control-information between
  - → microprocessor & its memory
  - → microprocessor & I/O
- Three types of buses are used for transfer of information: i)address, ii)data and iii)control.

Address Bus

- The address-bus is used to request
  - → a memory-location(from the memory) or
  - → an I/O location(from the I/O devices)
- If I/O is addressed, the address-bus contains a 16-bit I/O address. The 16-bit I/O address (or port number) selects one of 64K different I/O devices.
- If memory is addressed, the address-bus contains a memory-address (which varies in width with the different versions of the microprocessor).
- 8086 address 1M byte of memory using a 20-bit address that selects locations 00000H-FFFFFH.
  80286 address 16MB of memory using a 24-bit address that selects locations 000000H-FFFFFFH.

Data Bus

- The data bus is used to transfer information between
  - → microprocessor and memory
  - → microprocessor and I/O address-space.
- A 16-bit data bus transfers 16 bits of data at a time.
- The advantage of a wider data bus is speed in applications that use wide data.(For example, if a 32 bit number is stored in memory, it takes the 8086 microprocessor two transfer operations to complete because its data bus is only 16 bits wide).

Control Bus

- The control-bus
  - → controls the memory & I/O and
  - → requests reading or writing of data
- There are 4 control-bus connections:
  1) $\overline{MRDC}$ (memory read control),
  2) $\overline{MWTC}$ (memory write control),
  3) $\overline{IORC}$ (I/O read control) and
  4) $\overline{IOWC}$ (I/O write control) {The overbar indicates that the control signal is active-low i.e. it is active when a logic zero appears on the control line. For example, if $\overline{IOWC}$ =0, the microprocessor is writing data from the data bus to an I/O device whose address appears on the address bus}
- Steps to read (or fetch) data from memory:
  1) Firstly, microprocessor sends an address (of a memory-location) through the address bus to the memory
  2) Next, it sends the MRDC signal to cause the memory read data
  3) Finally, the data read from the memory are passed to the microprocessor through the data bus

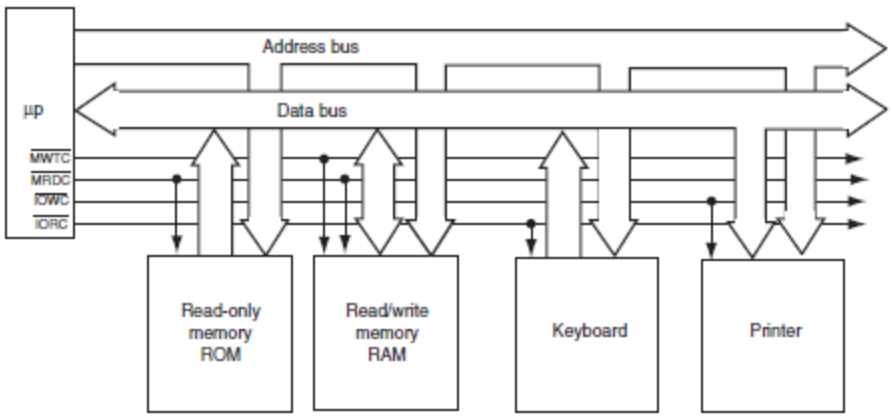


Figure 1-12: The block diagram of a computer system showing the address, data and control bus structure

The Intel family of microprocessor bus and memory sizes

Microprocessor	Data bus width	Address bus width	Memory size
8088	8	20	1M
8086	16	20	1M
80286	16	24	16M
80386DX	32	24	4G
Pentium	64	32	4G
Pentium Pro Core2	64	40	1T
Itanium	128	40	1T

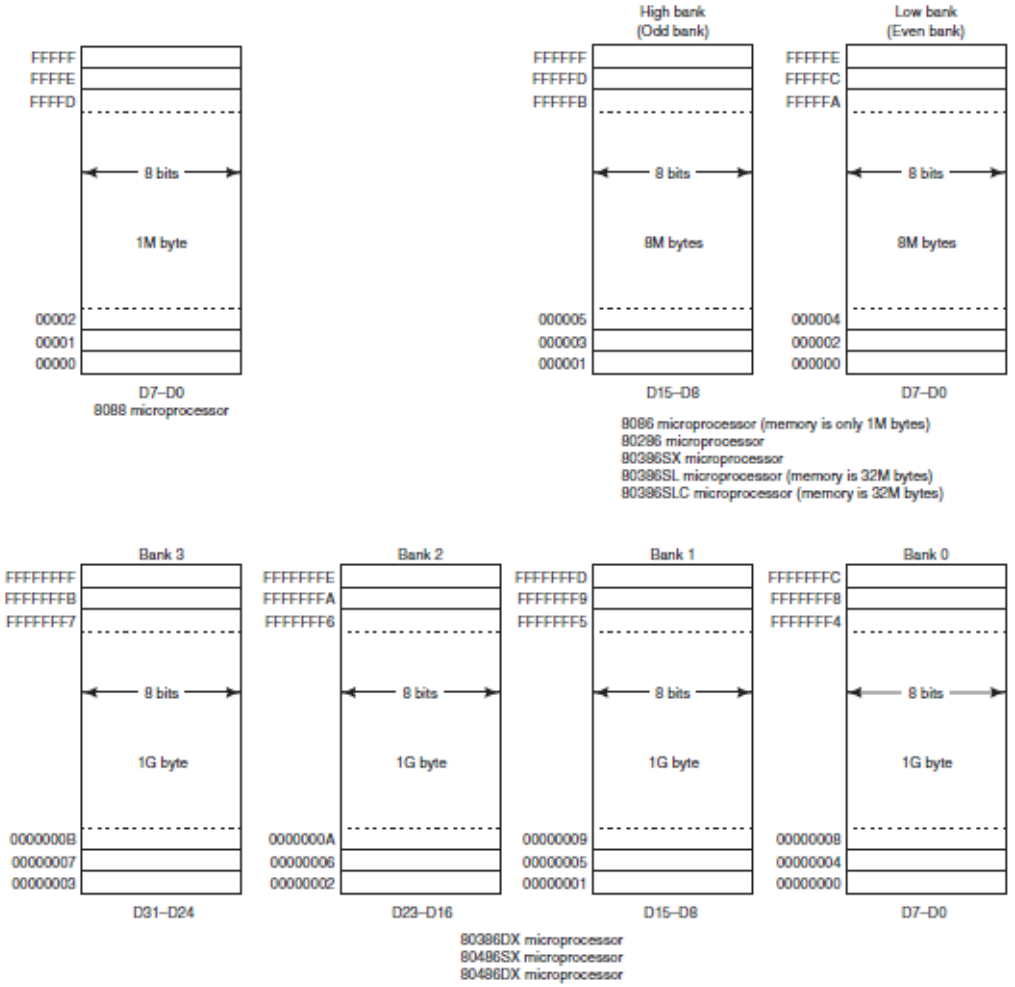


Figure 1-13: The physical memory systems of the 8086 and 80286

Binary Coded Hexadecimal (BCH)

- This is used to represent hexadecimal data in binary-code.
- A binary-coded hexadecimal number is a hexadecimal number written so that each digit is represented by a 4-bit number.
- The assembler is a program that is used to program a computer in its native binary machine language.
- Hexadecimal memory addresses (memory locations) are used to number each byte of the memory system. A hexadecimal number is a number represented in base 16, with each digit representing a value from 0 to 9 and A to F

Hexadecimal digit =>BCH code

0=> 0000	1=> 0001	2=> 0010	3=> 0011	4=> 0100
5=> 0101	6=> 0110	7=> 0111	8=>1000	9=>1001
A=>1010	B=>1011	C =>1100	D=>1101	E=> 1110

F=> 1111

Example 1:

2AC =>0010 1010 1100

1000 0011 1101 **.** 1110 =>83D.E

UNIT 1(CONT.): THE MICROPROCESSOR AND ITS ARCHITECTURE

Internal Microprocessor Architecture

The Programming Model

- The programming model of the 8086 is considered to be program-visible because its registers
  → are used during application programming and
  → are specified by the instructions.
- Other registers are considered to be program-invisible because
  they are not addressable directly during applications programming,
  but may be used indirectly during system programming.
- Only 80286 contain the program-invisible registers used to control and operate the protected memory system and other features of the microprocessor.
- The programming model contains 8-, 16- and 32-bit registers.
- The 8-bit registers are AH, AL, BH, BL, CH, CL, DH and Dl. (For example, an *ADD AL,AH* instruction adds the 8-bit contents of AH to AL)
- The 16-bit registers are AX, BX, CX, DX, SP, BP, DI and SI (AX contains AH and AL)
- The segment registers are CS, DS, ES and SS.
- The 32-bit extended registers are EAX, EBX, ECX, EDX, ESP, EBP, EDI and ESI.
- The 32-bit registers are called multipurpose registers because
  they hold various data-sizes (bytes, words) and are used for almost any purpose.
- The 64-bit registers are designated as RAX, RBX and so on.
- There are also additional 64-bit registers that are called R8 through R15.

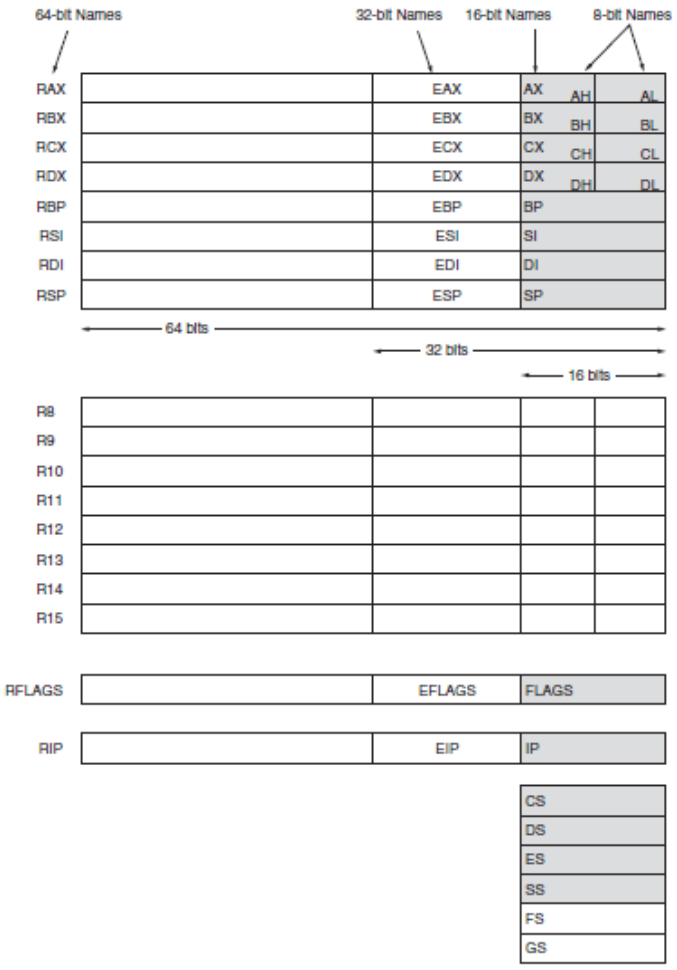


Figure 2-1: The programming model of 8086 through Core2 microprocessor

Multipurpose Registers

RAX(Accumulator)

- AX is used for instructions such as multiplication & division instructions (Figure 2-1).
- In 80386, the EAX may also hold the offset-address of a location in the memory.

RBX(Base Index)

- BX holds the offset address of a location in the memory.
- In 80386, EBX also can address memory-data.

RCX(Count)

- CX holds the count for various instructions.
- The shift & rotate instructions use CL as the count
  the repeated string & loop instructions use CX as the count.

RDX(Data)

- DX holds
  → a part of the result from a multiplication or
  → a part of the dividend before a division

RBP(Base Pointer)

- BP points to a memory-location for memory data-transfers.

RDI(destination Index)

- DI addresses string destination-data for the string instructions.

RSI(Source Index)

- SI addresses source string-data for the string instructions.

R8 through R15

- These are only found in the Pentium if 64-bit extensions are enabled.

Special Purpose Registers

RIP(Instruction Pointer)

- It is used by the microprocessor to find the next sequential instruction in a program located within the code-segment.
- It can be modified with a jump or a call instruction.

RSP(Stack Pointer)

- It addresses an area-of-memory called the stack.
- The stack-memory stores data through this pointer.

Segment Registers

- Segment-registers generate memory-addresses when combined with other registers.

CS(Code)

- The code-segment is a section-of-memory that holds the code(programs & procedures) used by the microprocessor.
- CS register contains the starting-address of the code-segment.
- In real-mode operation, it defines the start of a 64Kbyte code-segment.
  In protected-mode, it selects a descriptor that describes the starting-address and length of a code-segment memory.
- The code-segment is limited to
  → 64 KB in the 8086 and
  → 4 GB in the 80386

DS(Data)

- The data-segment is a section-of-memory that contains most data used by a program.
- Data are accessed in the data-segment by an offset-address (or the contents of other registers that hold the offset-address).

ES(Extra)

- The extra-segment is an additional data-segment that is used by some of the string instructions to hold destination-data.

SS(Stack)

- The stack-segment is a section-of-memory used for the stack.
- The stack entry-point is determined by the stack-segment(SS) and stack-pointer(SP) registers.

FS and GS

- The FS and GS segments allow 2 additional memory segments for access by programs in 80386 microprocessor.

RFLAGS

• This register indicates the condition of the microprocessor and controls its operation (Figure 2-2).

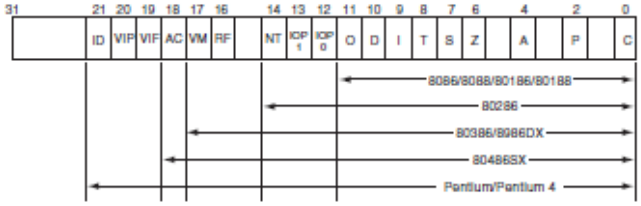


Figure 2-2:The EFLAG register of microprocessor family

C(Carry)

• Carry holds the carry after addition or the borrow after subtraction.

• The carry flag also indicates error conditions (as dictated by some programs and procedures).

P(Parity)

• Parity is logic 0 for odd-parity and logic 1 for even-parity.

• Parity is the count of 1s in a binary-number expressed as even or odd.

For example, if a number contains three binary 1 bits, it has odd-parity.

A(Auxiliary Carry)

• The auxiliary-carry holds the carry after addition (or borrow after subtraction) between bit-positions 3 and 4 of the result.

• This flag bit is tested by the DAA or DAS instructions to adjust the value of AL after a BCD addition or subtraction.

Z(Zero)

• Zero flag shows that the result of an arithmetic or logic operation is zero.

If Z=1, the result is zero; if Z=0, the result is not zero.

S(Sign)

• Sign flag holds the sign of the result after an arithmetic or logic instruction executes.

If S=1, the sign bit is set(or negative); if S=0,the sign bit is cleared(or positive).

T(Trap)

• If T=1, the microprocessor interrupts the flow of the program on conditions as indicated by the debug registers and control registers. If the T=0, the trapping feature is disabled.

I(Interrupt)

• This flag controls the operation of the INTR(interrupt request) input pin.

If I=1, the INTR pin is enabled; if I=0, the INTR pin is disabled.

• The I flag is set with STI(set I flag) and cleared with the CLI(clear I flag) instructions.

D(Direction)

• The direction flag selects either the increment or decrement mode for the DI or SI registers during string instructions.

• If D=1, registers are automatically decremented; if D=0, registers are automatically incremented.

• The D flag is set with STD(set direction) and cleared with the CLI(clear direction) instructions.

O(Overflow)

• Overflows occur when signed-numbers are added or subtracted.

• An overflow indicates that the result has exceeded the capacity of the machine.

IOPL(I/O Privileged Level)

• IOPL is used to select the privilege-level for I/O devices.

• If current privilege-level is higher or more trusted than the IOPL, I/O executes without difficulty.

If current privilege-level is lower than the IOPL, an interrupt occurs, causing execution to suspend.

NT(Nested Task)

• This flag indicates that the current task is nested within another task.

RF(Resume)

• This flag is used with debugging to control the resumption of execution after the next instruction.

VM(Virtual Mode)

• This flag selects virtual mode operation.

AC(Alignment Check)

• This flag activates if a word is addressed on a non-word boundary.

VIF(Virtual Interrupt)

• This is a copy of the interrupt flag bit available to the Pentium4 microprocessor.

VIP(Virtual Interrupt Pending)

• This is used in multitasking environments to provide the operating-system with virtual interrupt flags and interrupt pending information.

ID(Identification)

• This flag indicates that the Pentium4 microprocessor supports the CPUID instruction.

Real Mode Memory Addressing

• Real-mode operation allows the microprocessor to address only first 1Mbyte of memory-space.
(The first 1M byte of memory is called the real memory, conventional memory or DOS memory system).

Segments & Offsets

• In real mode, a combination of a segment-address and an offset-address accesses a memory-location(Figure 2-3).

• The *segment-address* (located within one of the segment registers) defines the starting-address of any 64Kbyte memory-segment.

`The *offset-address* selects any location within the 64KB memory segment.

• Segments always have a length of 64KB.

• Each segment-register is internally appended with a 0H on its rightmost end. This forms a 20-bit memory-address, allowing it to access the start of a segment. (For example, when a segment register contains 1200H, it addresses a 64Kbyte memory segment beginning at location 12000H).

• Because of the internally appended 0H, real-mode segments can begin only at a 16byte boundary in the memory. This 16-byte boundary is called a *paragraph.*

• Because a real-mode segment of memory is 64K in length, once the beginning address is known, the ending address is found by adding FFFFH.

• The offset-address is added to the start of the segment to address a memory location within the memory-segment. (For example, if the segment address is 1000H and the offset address is 2000H,the microprocessor addresses memory location 12000H).

• In the 80286, an extra 64K minus 16bytes of memory is addressable when the segment is FFFFH and the HIMEM.SYS driver for DOS is installed in the system. This area of memory is referred to as *high memory*.

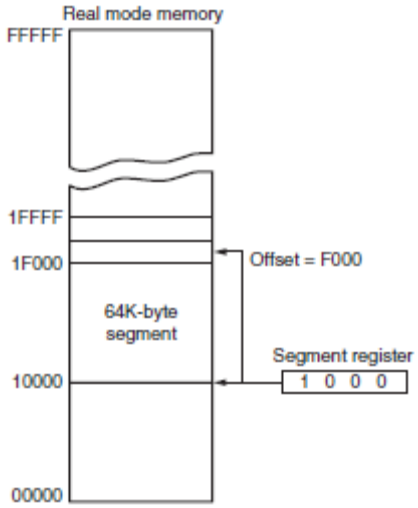


Figure 2-3: The real mode memory-addressing scheme using a segment address plus an offset

Default Segment & Offset Registers

• The microprocessor has a set of rules that apply to segments whenever memory is addressed. These rules define the segment-register and offset-register combination. For example, the CS register is always used with the IP to address the next instruction in a program.

• The CS register defines the start of the code-segment and the IP locates the next instruction within the code-segment. For example, if CS=1400H and IP=1200H, the microprocessor fetches its next instruction from memory location 14000H+1200H=15200H

Segment	*Offset*	*Special Purpose*
CS	IP	Instruction address
SS	SP or BP	Stack address
DS	BX, DI, SI, an 8- or 16-bit number	Data address
ES	DI for string instructions	String destination address

Table 2-3: Default 16-bit segment and offset combinations.

Segment & Offset Addressing Scheme Allows Relocation

- This scheme allows both programs and data to be relocated in the memory (Figure 2-4).
- This also allows programs written to function in the real-mode to operate in a protected-mode system.
- Relocatable-program can be placed into any area of memory and executed without change.
- Relocatable-data can be placed in any area of memory and used without any change to the program.

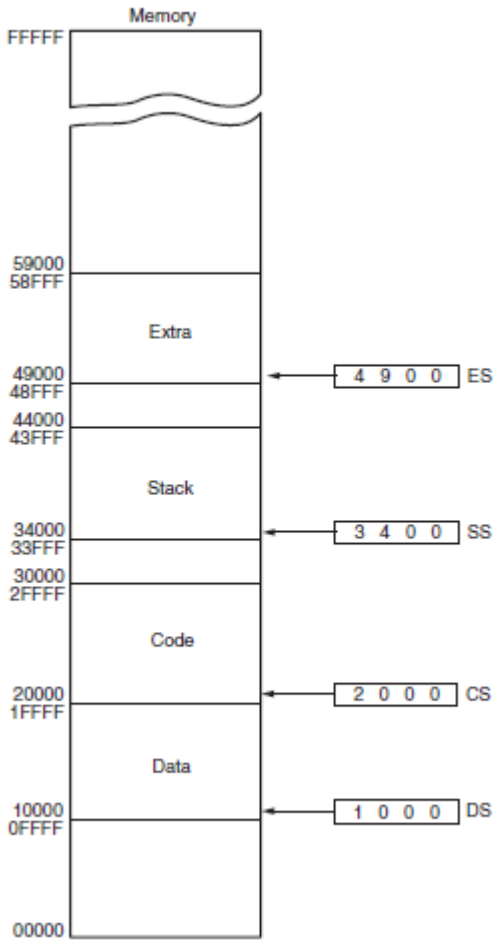


Figure2-4: A memory system showing the placement of four memory segments

Segment Register	*Starting Address*	*Ending Address*
2000H	20000H	2FFFFH
2001H	20010H	3000FH
2100H	21000H	30FFFH
AB00H	AB000H	BAFFFH
1234H	12340H	2233FH

Table 2-2: Example of real mode segment addresses

UNIT 2: THE MICROPROCESSOR AND ITS ARCHITECTURE(CONT.)

Introduction to Protected Mode Memory Addressing

- Protected-mode memory addressing(80286) allows access to data & programs located
  - → above first 1MB of memory &
  - → within first 1MB of memory.
- In place of segment-address, segment-register contains a selector that selects a descriptor from a descriptor-table. (The extended memory system is accessed via a segment-address plus an offset-address, just as in the real mode. The difference is that the segment-address is not held in the segment-register. In the protected-mode, the segment starting-address is stored in a descriptor that is selected by the segment-register).
- Descriptor describes
  - → memory-segment's location
  - → length &
  - → access rights
- Another difference in the 80386 is that the offset-address can be a 32-bit number instead of a 16-bit number.

Selectors & Descriptors

- *Selector*(located in the segment-register) selects one of 8192 descriptors from one of 2 tables of descriptors.
- *Descriptor* describes the location, length and access rights of the segment of memory.
- There are 2 descriptor-tables:
  1) Global descriptor table &
  2) Local descriptors table.
- Global-descriptors contain segment-definitions that apply to all programs
  whereas local-descriptors are usually unique to an application.
- Each descriptor-table contains 8192 descriptors, so a total of 16384 total descriptors are available to an application at any time.
- A descriptor contains
  1) Base-address locates starting-address of memory-segment
  2) Segment-limit contains last offset-address found in a segment (For example, if a segment begins at memory location F00000H and ends at location F000FFH, the base-address is F00000H and the limit is FFH).
  3) Access rights byte defines how the memory-segment is accessed via a program.
- For 80286 microprocessor, the base-address is a 24-bit address, so segments begin at any location in its 16MB of memory.
- In 80386, if G(granularity)=0, the limit specifies a segment-limit of 00000H to FFFFFH.
  If G=1, the value of the limit is multiplied by 4KB.
- In the 64-bit descriptor, if L=1, 64-bit address in a Pentium4 with 64-bit extensions is selected.
  if L=0, 32-bit compatibility mode is selected
- The AV bit is used by some operating-systems to indicate that the segment is available(AV=1) or not available(AV=0).
- D bit indicates how 80386 instructions access register & memory-data in protected- or real-mode.
  If D=0, the instructions are 16-bit instructions, compatible with the 8086 microprocessor. (This means that the instructions use 16-bit offset addresses and 16-bit register by default).
  If D=1, the instructions are 32-bit instructions.

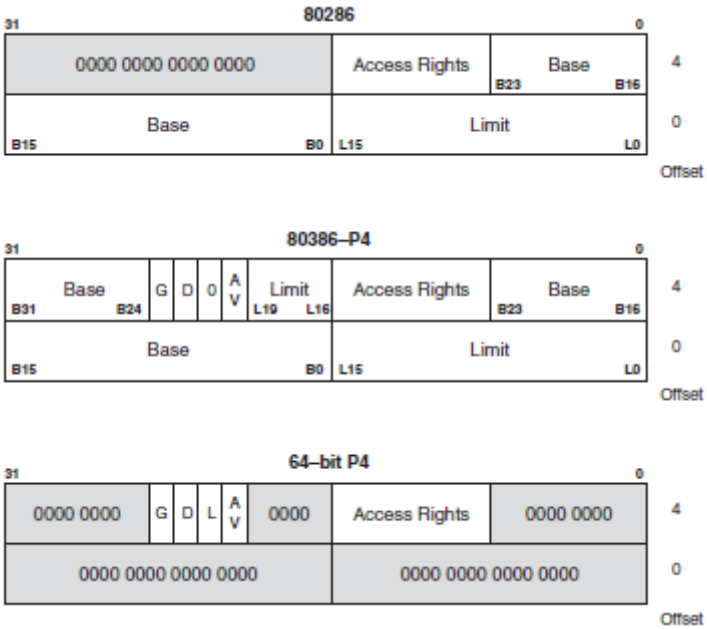


Figure 2-6: The 80286 through Core2 64-bit descriptors

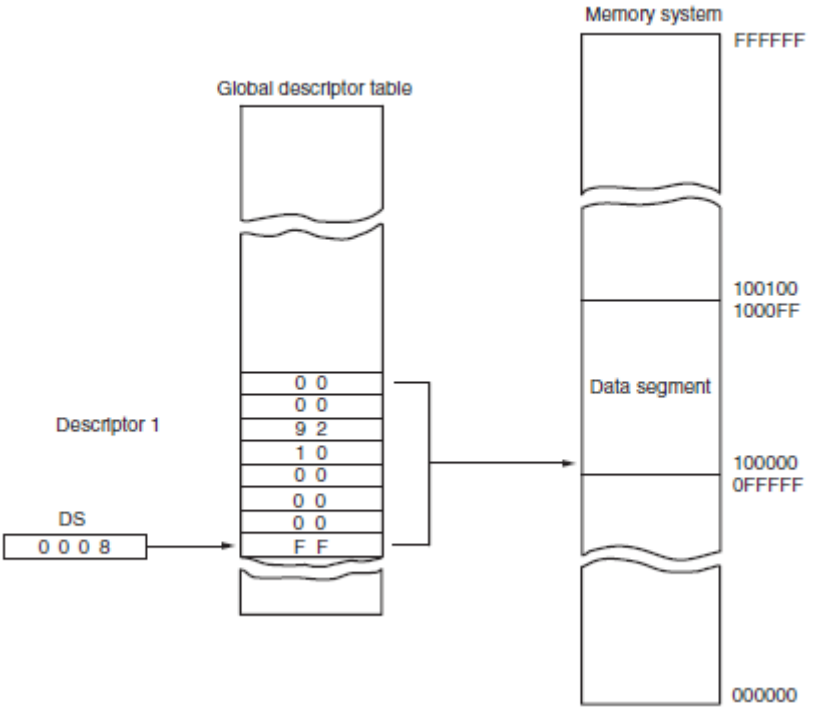


Figure 2-9: Using DS register to select a descriptor from the global descriptor table

Access Rights Byte

• This controls access to the protected-mode segment. This byte describes how the segment functions in the system.

• If the segment is a data-segment, the direction of growth is specified.
If the segment grows beyond its limit, the operating-system program is interrupt to indicate a general protection-fault.

• The RPL(request privilege level) requests the access privilege-level of a memory-segment.
If the RPL is higher than the privilege-level set by the access rights byte, access is granted.

• The segment-register contains 3 fields of information:
→ Selector(First 13 bits) address one of 8192 descriptors from a descriptor-table.
→ TI bit selects either global descriptor-table(TI=0) or local descriptor-table(TI=1).
→ RPL(Last 2 bits) select the requested priority level for the memory segment access.

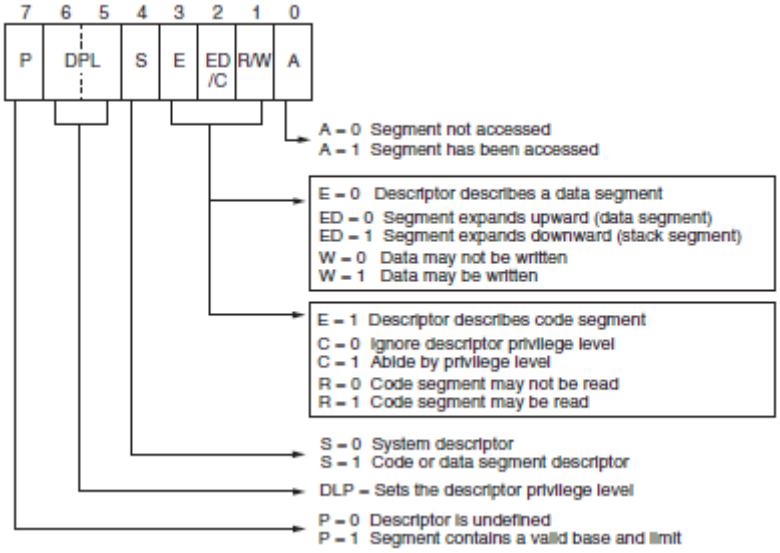


Figure 2-7:The access rights byte for the 80286 descriptor

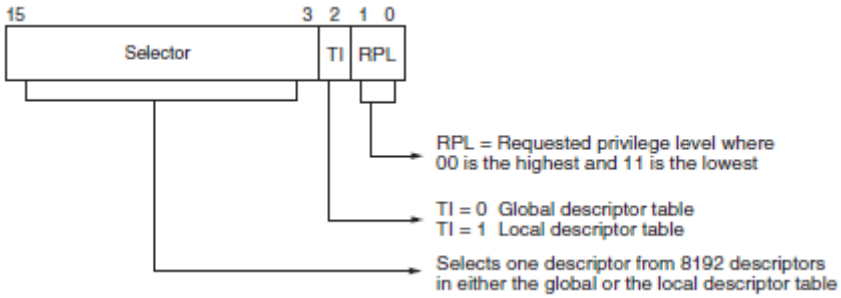


Figure 2-8: The contents of a segment register during protected mode operation of the 80286 microprocessor

Program Invisible Registers

• These registers are not directly addressed by software so they are given this name.

• These registers control the microprocessor when operated in protected-mode.

• The program-invisible portion(cache-memory) of the segment-register is loaded with base-address, limit and access rights each time the number segment-register is changed.

• When a new segment-number is placed in a segment-register, the microprocessor accesses a descriptor-table and loads the descriptor into the program-invisible portion of the segment-register. It is held there and used to access the memory-segment until the segment-number is again changed.

• The GDTR(global descriptor table register) & IDTR(interrupt descriptor table register) contain the base-address of the descriptor-table and its limit.(The limit of each descriptor-table is 16 bits because the maximum table length is 64KB)

• When the protected-mode operation is desired, the address of the global descriptor-table and its limit are loaded into the GDTR.

• The location of the local descriptor-table is selected from the global descriptor-table. One of the global descriptors is set up to address the local descriptor-table. To access the local descriptor-table, the LDTR(local descriptor table register) is loaded with a selector.

• This selector accesses the global descriptor-table and loads the address, limit and access rights of the local descriptor-table into the cache portion of the LDTR.

• The TR(task register) holds a selector, which accesses a descriptor that defines a task. (A task is most often a procedure or application program)

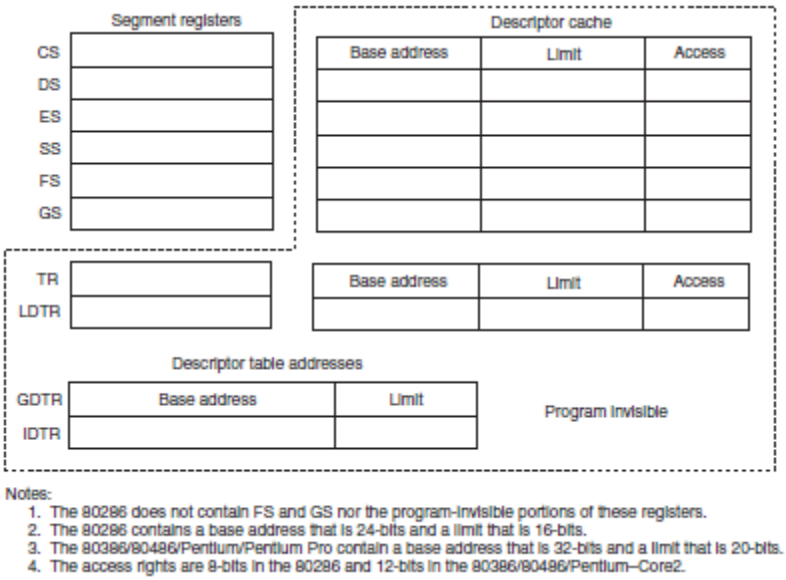


Figure 2-10:The program-invisible register within the 80286 microprocessor

Memory Paging

- Memory-paging mechanism allows any physical memory-location to be assigned to any linear-address.
- The linear-address is defined as the address generated by a program
  The physical-address is the actual memory-location accessed by a program.

Paging Registers

- Memory-paging is accomplished through control-registers CR0 and CR3.
- The paging-unit is controlled by the contents of the control-registers.
- The leftmost bit(PG) position of CR0 selects paging when placed at a logic 1 level.
  If PG bit is cleared(0), linear-address generated by program becomes physical-address used to access memory.
  If PG bit is set(1), linear-address is converted to a physical-address through paging-mechanism.
- The page directory base-address locates the directory for the page translation-unit.
- If PCD is set(1),the PCD pin becomes a logic one during bus cycles that are not paged.
- The page-directory contains upto 1024 entries that locate physical-address of a 4KB memory-page.
- The linear address is broken into 3 sections:
  1) Page directory selects a page table that is indexed by the next 10 bits of the linear-address
  2) Page table entry
  3) Offset part selects a byte in the 4KB memory-page
- In 80486, the cache (translation look aside buffer-TLB) holds the 32 most recent page translation addresses.

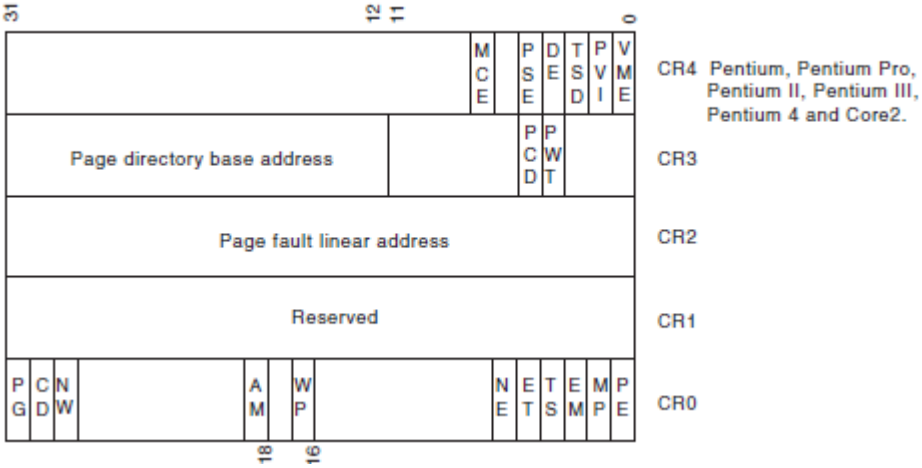


Figure 2-11:The control register structure of the microprocessor

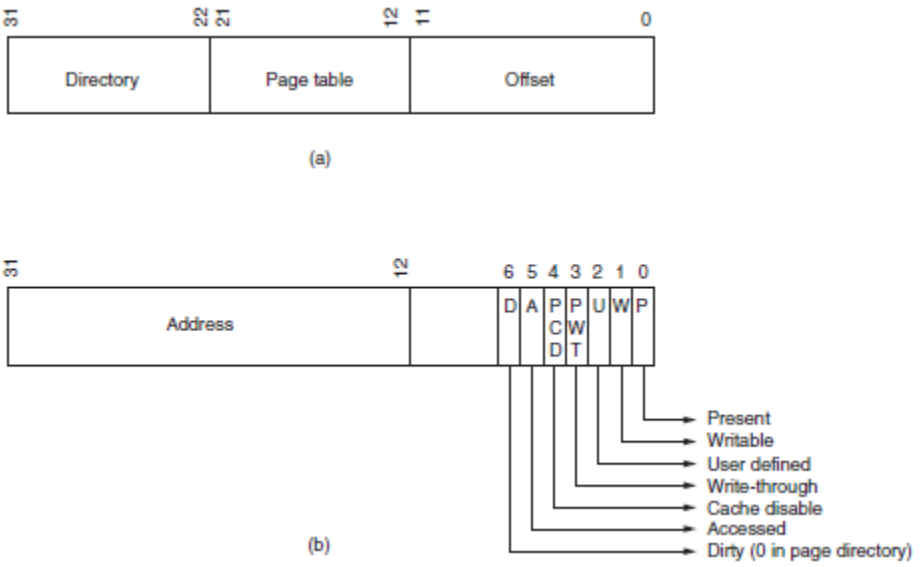


Figure 2-12: The format for the linear address

Flat Mode Memory

- In a Pentium-based computer, the memory-system that uses the 64-bit extensions uses a flat mode memory.
- A flat mode memory system is one in which there is no segmentation.
- The flat model does not use a segment-register to address a location in the memory.
- The CS segment-register is used to select a descriptor from the descriptor-table that defines the access rights of only a code-segment.
- The segment-register still selects the privilege-level of the software.
- The flat model does not select the memory-address of a segment using the base and limit in the descriptor.
- The flat mode memory contains 1TB of memory using a 40-bit address.

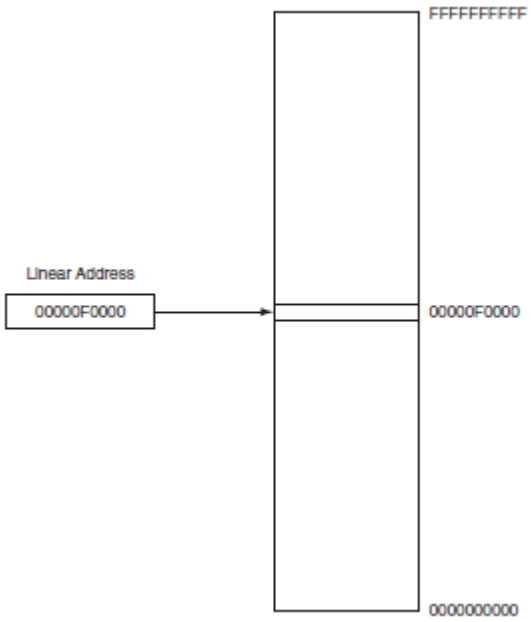


Figure 2-15: The 64-bit flat mode memory model

UNIT 2(CONT.): ADDRESSING MODES

Data Addressing Modes

• MOV AX,BX; This instruction transfers the word contents of the source-register(BX) into the destination-register(AX).
• The source never changes, but the destination always changes.
• This instruction always copies the source-data into the destination.
• This instruction never actually picks up the data and moves it.
• Memory-to-memory transfers are not allowed by any instruction except for the MOVS instruction.
• The source & destination are called operands (ex: contents of AX, BX, LOC)
• An opcode(operation code) tells microprocessor which operation to perform(ex ADD, MOV, SUB).

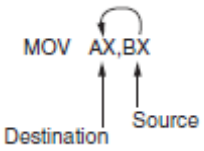
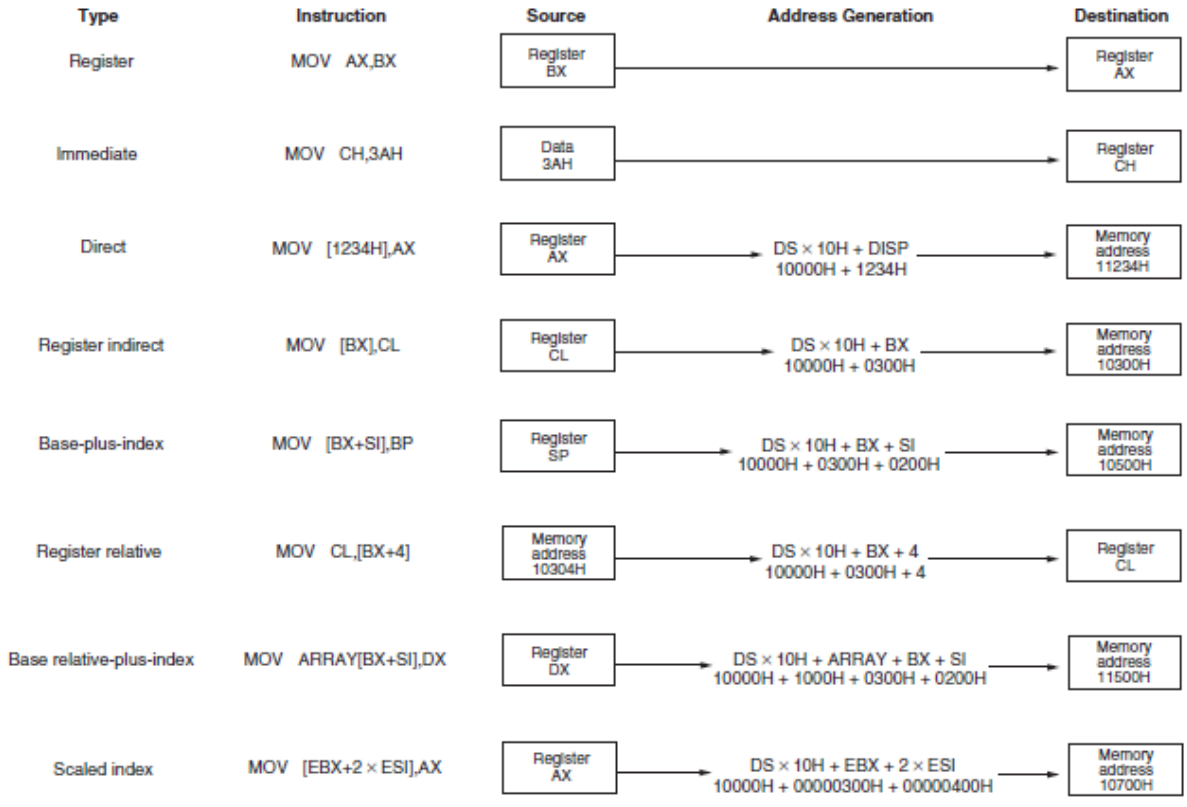


Figure 3-1: The MOV instruction showing the source, destination and direction of data flow



Notes: EBX = 00000300H, ESI = 00000200H, ARRAY = 1000H, and DS = 1000H

Figure 3-2:8086 data addressing modes

Addressing Modes

Register Addressing

• This transfers a copy of a byte(or word) from the source-register or contents of a memory-location to the destination-register or memory-location.

• Example: MOV CX, DX; This instruction copies word-sized contents of register DX into register CX.

Immediate Addressing

• This transfers an immediate byte/word of source-data into the destination-register or memory-location.

• For example, MOV AL, 22H ;This instruction copies a byte-sized 22H into register AL.

Direct Addressing

• This moves a byte(or word) between a memory-location and a register.

• For example, MOV CX, LIST ;This instruction copies the word-sized contents of memory-location LIST into register CX.

• Memory-to-memory transfers are not allowed by any instruction except for the MOVS instruction.

Register Indirect Addressing

• This transfers a byte between a register and a memory-location addressed by base-register.

• The index & base registers are BP, BX, DI and SI.

• For example, MOV [BP], DL ;This instruction copies the byte-sized contents of register DL into the memory-location addressed by BP.

Base Plus Index Addressing

• This transfers a byte between a register and the memory-location addressed by a base-register (BP or BX) plus an index-register (DI or SI).

• For example, MOV [BX+DI], CL ;this instruction copies the byte-sized contents of register CL into the memory-location addressed by BX plus DI.

Register Relative Addressing

• This moves byte between a register and memory-location addressed by an index-or base-register plus a displacement.

• For example, MOV AX, [BX+4] ;This instruction loads AX from memory-location addressed by BX+4.

Base-Relative-Plus Index Addressing

• This transfers a byte between a register and the memory-location addressed by a base- and an index-register plus a displacement.

• For example, MOV AX,[BX+DI+4] ;This instruction loads AX from memory-location addressed by BX+DI+4.

Scaled Index Addressing

• The second register of a pair of registers is modified by the scale factor of 2*, 4* to generate the operand memory address.

• For example, MOV EDX, [EAX+4*EBX] ;This instruction loads EDX from memory-location addressed by EAX plus four times EBX.

• Scaling allows access to word(2*), doubleword(4*) memory array-data.

RIP Relative Addressing

• This mode allows access to any location in the memory by adding a 32-bit displacement to the 64-bit contents of the 64-bit instruction pointer.

Register Addressing

• The microprocessor contains the following 8-bit registers used with register addressing: AH, AL, BH, BL, CH, CL, DH and DL.

• Also present are the following 16-bit registers: AX, BX, CX, DX, SP, BP, SI and DI.

• Some MOV instructions and the PUSH/POP instructions also use the 16-bit segment registers: CS, ES, DS, SS, FS and GS.

• Any instruction should use registers that are of same size {(AL,CL), (AX,CX)}. Never mix an 8-bit register with a 16-bit register because this is not allowed by the microprocessor and results in an error. A few instructions such as SHL DX, CL are exceptions to this rule.

• None of the MOV instructions affect the flag bits.
The flag bits are normally modified by arithmetic or logic instructions (ADD, SUB, INC)

• A segment-to-segment register MOV instruction is not allowed.

• The contents of the destination-register (or memory-location) change for all instructions except the CMP and TEST instructions.

Assembly Language	*Size*	*Operation*
MOV AL,BL	8 bits	Copies BL into AL
MOV CH,CL	8 bits	Copies CL into CH
MOV R8B,CL	8 bits	Copies CL to the byte portion of R8 (64-bit mode)
MOV R8B,CH	8 bits	Not allowed
MOV AX,CX	16 bits	Copies CX into AX
MOV SP,BP	16 bits	Copies BP into SP
MOV DS,AX	16 bits	Copies AX into DS
MOV BP,R10W	16 bits	Copies R10 into BP (64-bit mode)
MOV SI,DI	16 bits	Copies DI into SI
MOV BX,ES	16 bits	Copies ES into BX
MOV ECX,EBX	32 bits	Copies EBX into ECX
MOV ESP,EDX	32 bits	Copies EDX into ESP
MOV EDX,R9D	32 bits	Copies R9 into EDX (64-bit mode)
MOV RAX,RDX	64 bits	Copies RDX into RAX
MOV DS,CX	16 bits	Copies CX into DS
MOV ES,DS	—	Not allowed (segment-to-segment)
MOV BL,DX	—	Not allowed (mixed sizes)
MOV CS,AX	—	Not allowed (the code segment register may not be the destination register)

Table 3-1: Examples of register-addressed instructions

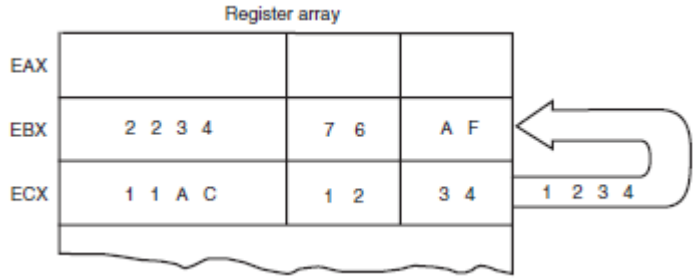


Figure 3-3: The effect of executing the MOV BX,CX instruction at the point just before the BX register changes. Note that only the rightmost 16bits of register EBX change

Example 3-1

```
MOV AX, BX      ; copy contents of BX into AX
MOV CL, DH      ; copy contents of DH into CL
MOV AX, CS      ; copy CS into DS (2 steps)
MOV DS, AX      ;
MOV CS, AX      ; copy AX into CS (causes problem)
```

Immediate Addressing

- The term 'immediate' implies that the data immediately follow the opcode in the memory.
- Immediate-data are constant-data, whereas
  data transferred from a register(or memory-location) are variable-data.
- This addressing operates upon a byte or word of data.
- MOV immediate instruction transfers a copy of immediate-data into a register or a memory-location.
- The symbolic assembler shows immediate-data in many ways.
  The letter H appends hexadecimal data.
  The letter B appends binary data.

Assembly Language	*Size*	*Operation*
MOV BL,44	8 bits	Copies 44 decimal (2CH) into BL
MOV AX,44H	16 bits	Copies 0044H into AX
MOV SI,0	16 bits	Copies 0000H into SI
MOV CH,100	8 bits	Copies 100 decimal (64H) into CH
MOV AL,'A'	8 bits	Copies ASCII A into AL
MOV AH,1	8 bits	Not allowed in 64-bit mode, but allowed in 32- or 16-bit modes
MOV AX,'AB'	16 bits	Copies ASCII BA* into AX
MOV CL,11001110B	8 bits	Copies 11001110 binary into CL
MOV EBX,12340000H	32 bits	Copies 12340000H into EBX
MOV ESI,12	32 bits	Copies 12 decimal into ESI
MOV EAX,100B	32 bits	Copies 100 binary into EAX
MOV RCX,100H	64 bits	Copies 100H into RCX

Table 3-2: Examples of immediate addressing using the MOV instruction

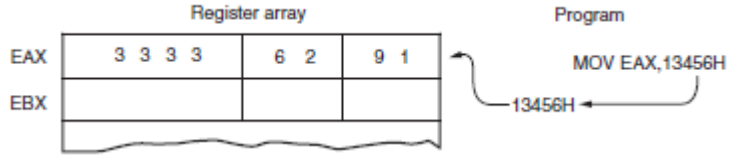


Figure 3-4: Operation of the MOV EAX,13456H instruction. This instruction copies immediate data(13456) into EAX

Example 3-3

Label	Opcode	Operand	Comment
DATA1	DB	23H	; defines DATA1 as a byte of 23H
DATA2	DW	1000H	;defines DATA2 as a word of 1000
START:	MOV	AL,BL	;copy BL into AL
	MOV	BH,AL	; copy AL into BH
	MOV	CX,200	;copy 200 into CX

- Each statement in an assembly language program consists of 4 parts:
  1) Label is used to store a symbolic-name for the memory-location that it represents. A label may be of any length from 1 to 35 characters.
  2) Opcode-field is used to hold the instruction or opcode
  3) Operand-field contains information used by the opcode
  4) Comment-field contains a comment about an instruction or a group of instructions. A comment always begins with a semicolon (;)

Example 3-2:

```
.MODEL TINY     ; choose single segment model
.CODE                       ; start of code segment
.STARTUP                    ; start of program
        MOV AX,0            ;place 0000H into AX
        MOV SI,AX           ;copy AX into SI
.EXIT                       ;exit to DOS
END                         ;end of program
```

- .MODEL TINY directs the assembler to assemble the program into a single code segment.
- .CODE indicates the start of the code segment
- .STARTUP indicates the start of the code.
- .EXIT causes the program to exit to DOS.
- .END indicates the end of the program file.

Example 3-6

```
.MODEL SMALL            ; choose small model
.DATA                   ; start data segment
        DATA1 DB 10H            ; place 10H into DATA1
        DATA2 DB 2H             ; place 02H into DATA2
        DATA3 DW 0              ; place 0000H into DATA3
        DATA4 DW 1234H          ; place 1234H into DATA4
.CODE                   ; start code segment
.STARTUP                ; start program
        MOV AL,DATA1            ; copy DATA1 into AL
        MOV AH,DATA2            ; copy DATA2 into AH
        MOV DATA3,AX            ; copy AX into DATA3
        MOV BX,DATA4            ; copy DATA4 into BX
.EXIT                   ; exit to DOS
END                     ; end program listing
```

- .DATA is used to inform the assembler where the data segment begins.
- .SMALL model allows one data segment and one code segment. A SMALL model program assembles as an execute(.EXE) program file.
- .STARTUP
  - -indicates the start of the code &
  - -loads the DS register with the segment address of the data segment memory

Direct Data Addressing

- There are 2 basic forms of direct data addressing:
  1) Direct addressing applies to a MOV between a memory-location and the AL or AX (accumulator)
  2) Displacement addressing applies to almost any instruction in the instruction set
- In either case, the effective-address(EA) is formed by adding the displacement to the default data-segment address.(ex: if DS=1000 and BP=200, then EA=DS*10+BP=1000*10+200=10200H)

Direct Addressing

- Direct addressing with a MOV instruction transfers data between a memory-location, located within the data-segment and the AL or AX register.
- A MOV instruction is usually a 3-byte long instruction.
- The MOV AL,DATA ;This instruction loads AL from the memory-location DATA(1234H)
- Memory location 'DATA' is a symbolic memory location,
  while the 1234H is the actual hexadecimal location.
- MOV AL, [1234H] ;This instruction transfers a copy of the byte-sized contents of memory-location 11234H into AL. The effective address is formed by adding 1234H(offset address) and 10000H(data segment address of 1000H ties 10H) in a system operating in the real mode.

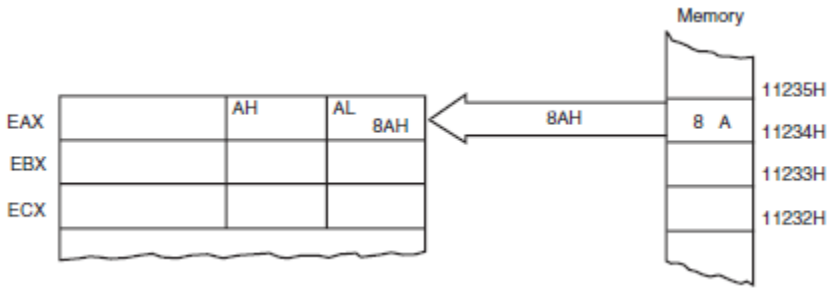


Figure 3-5: The operation of the MOV AL,[1234H] instruction when DS=1000H

Assembly Language	*Size*	*Operation*
MOV AL,NUMBER	8 bits	Copies the byte contents of data segment memory location NUMBER into AL
MOV AX,COW	16 bits	Copies the word contents of data segment memory location COW into AX
MOV EAX,WATER*	32 bits	Copies the doubleword contents of data segment location WATER into EAX
MOV NEWS,AL	8 bits	Copies AL into byte memory location NEWS
MOV THERE,AX	16 bits	Copies AX into word memory location THERE
MOV HOME,EAX*	32 bits	Copies EAX into doubleword memory location HOME
MOV ES:[2000H],AL	8 bits	Copies AL into extra segment memory at offset address 2000H
MOV AL,MOUSE	8 bits	Copies the contents of location MOUSE into AL; in 64-bit mode MOUSE can be any address
MOV RAX,WHISKEY	64 bits	Copies 8 bytes from memory location WHISKEY into RAX

Table 3-3: Direct addressed instructions using AX and AL

Register Indirect Addressing

- This transfers a byte between a register and a memory-location addressed by base-register.
- The index- and base-registers are BP, BX, DI and SI.
- The [] symbols denote indirect addressing.
- If BP addresses memory, the stack-segment is used by default.
  If BX, DI or SI addresses memory, the data-segment is used by default.

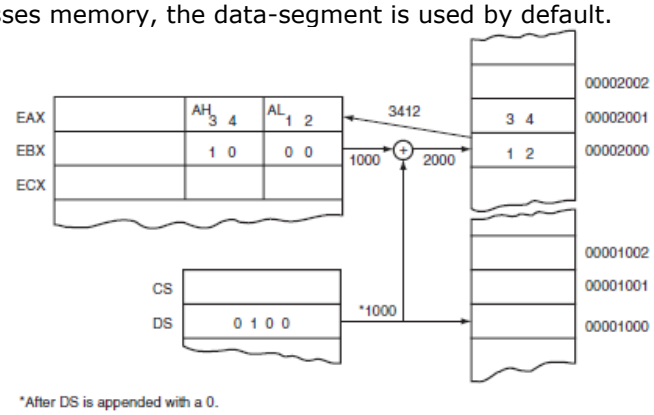


Figure 3-6: The operation of the MOV AX,[BX] instruction when BX=1000H and DS=0100H.

TABLE 3–5 Examples of register indirect addressing.

Assembly Language	*Size*	*Operation*
MOV CX,[BX]	16 bits	Copies the word contents of the data segment memory location addressed by BX into CX
MOV [BP],DL*	8 bits	Copies DL into the stack segment memory location addressed by BP
MOV [DI],BH	8 bits	Copies BH into the data segment memory location addressed by DI
MOV [DI],[BX]	—	Memory-to-memory transfers are not allowed except with string instructions
MOV AL,[EDX]	8 bits	Copies the byte contents of the data segment memory location addressed by EDX into AL
MOV ECX,[EBX]	32 bits	Copies the doubleword contents of the data segment memory location addressed by EBX into ECX
MOV RAX,[RDX]	64 bits	Copies the quadword contents of the memory location address by the linear address located in RDX into RAX (64-bit mode)

Table 3-5: Examples of register indirect addressing

- In some cases, indirect addressing requires specifying the size of the data. The size is specified by the special assembler directive BYTE PTR, WORD PTR. These directives indicate the size of the memory data addressed by the memory pointer(PTR).
- For example, the *MOV AL,[DI]* instruction is clearly a byte-sized move instruction, but the MOV [DI],10H instruction is ambiguous. Does the *MOV [DI],10H* instruction address a byte, word sized memory location? The assembler can't determine the size of the 10H. The instruction *MOV BYTE PTR[DI],10H* clearly designates the location addressed by DI as a byte-sized memory location.

Example 3-7: Program loads register BX with the starting address of the table and it initializes the count, located in register CX to 50.

```
MODEL SMALL                    ; select small model
.DATA                          ; start data segment
      DATAS DW 50 DUP(?)       ; setup array of 50 words
.CODE                          ; start code segment
.STARTUP                       ; start program
      MOV AX,0
      MOV ES,AX                ; address segment 0000 with ES
      MOV BX,OFFSET DATAS      ; address DATAS array with BX
      MOV CX,50                ; load counter with 50
      AGAIN:
      MOV AX,ES:[046CH]        ; get clock value
      MOV [BX],AX              ; save clock value in DATAS
      INC BX                   ; increment BX to next element
      LOOP AGAIN               ; repeat 50 times
.EXIT                          ; exit to DOS
END                            ; end program listing
```

- The LOOP instruction decrements the counter(CX); if CX is not zero, LOOP causes a jump to memory location AGAIN. If CX becomes zero, no jump occurs and this sequence of instruction ends.

Base Plus Index Addressing

- This addressing is similar to indirect addressing because it indirectly addresses memory-data.
- This uses one base-register(BP or BX) & one index-register(DI or SI) to indirectly address-memory.
- The base-register often holds the beginning location of a memory-array, whereas Index-register holds the relative position of an element in the array.

Assembly Language	*Size*	*Operation*
MOV CX,[BX+DI]	16 bits	Copies the word contents of the data segment memory location addressed by BX plus DI into CX
MOV CH,[BP+SI]	8 bits	Copies the byte contents of the stack segment memory location addressed by BP plus SI into CH
MOV [BX+SI],SP	16 bits	Copies SP into the data segment memory location addressed by BX plus SI
MOV [BP+DI],AH	8 bits	Copies AH into the stack segment memory location addressed by BP plus DI
MOV CL,[EDX+EDI]	8 bits	Copies the byte contents of the data segment memory location addressed by EDX plus EDI into CL
MOV [EAX+EBX],ECX	32 bits	Copies ECX into the data segment memory location addressed by EAX plus EBX
MOV [RSI+RBX],RAX	64 bit	Copies RAX into the linear memory location addressed by RSI plus RBX (64-bit mode)

Table 3-6: Examples of base-plus-index addressing

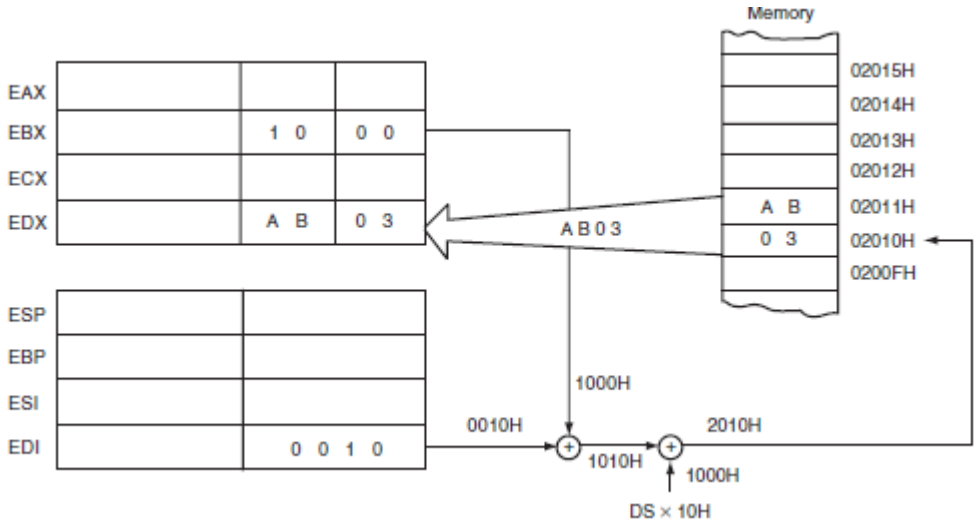


Figure 3-8: An example showing how the base-plus-index addressing mode functions for the MOV DX,[BX+DI instruction]. Memory address 02010H is accessed because DS=0100H, BX=100H and DI=0010H

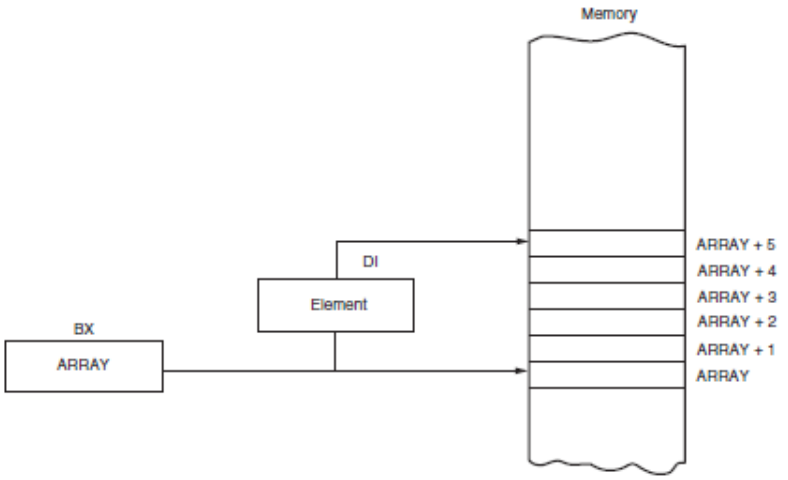


Figure 3-9:An example of the base-plus-index addressing mode. Here an element(DI) of an ARRAY(BX) is addressed.

Example 3-8: Program to move array element 10H into array element 20H.

```
.MODEL SMALL            ; select small model
.Data                   ; start data segment
        ARRAY DB 16 DUP(?)          ; setup array of 16 bytes
        DB 29H                      ; element 10H
        DB 20 dup(?)
.CODE                   ; start code segment
.STARTUP
        MOV BX,OFFSET ARRAY         ; address ARRAY
        MOV DI,10H                  ; address element 10H
        MOV AL,[BX+DI]              ; get element 10H
        MOV DI,20H                  ; address element 20H
        MOV [BX+DI],AL              ; save in element 20H
.EXIT                   ; exit to DOS
END                     : end program
```

Register Relative Addressing

• This moves byte/word between a register and the memory-location addressed by an index- or base- register plus a displacement (BP, BX, DI or SI).

Assembly Language	*Size*	*Operation*
MOV AX,[DI+100H]	16 bits	Copies the word contents of the data segment memory location addressed by DI plus 100H into AX
MOV ARRAY[SI],BL	8 bits	Copies BL into the data segment memory location addressed by ARRAY plus SI
MOV LIST[SI+2],CL	8 bits	Copies CL into the data segment memory location addressed by the sum of LIST, SI, and 2
MOV DI,SET_IT[BX]	16 bits	Copies the word contents of the data segment memory location addressed by SET_IT plus BX into DI
MOV DI,[EAX+10H]	16 bits	Copies the word contents of the data segment location addressed by EAX plus 10H into DI
MOV ARRAY[EBX],EAX	32 bits	Copies EAX into the data segment memory location addressed by ARRAY plus EBX
MOV ARRAY[RBX],AL	8 bits	Copies AL into the memory location ARRAY plus RBX (64-bit mode)
MOV ARRAY[RCX],EAX	32 bits	Copies EAX into memory location ARRAY plus RCX (64-bit mode)

Table 3-7: Examples of register relative addressing

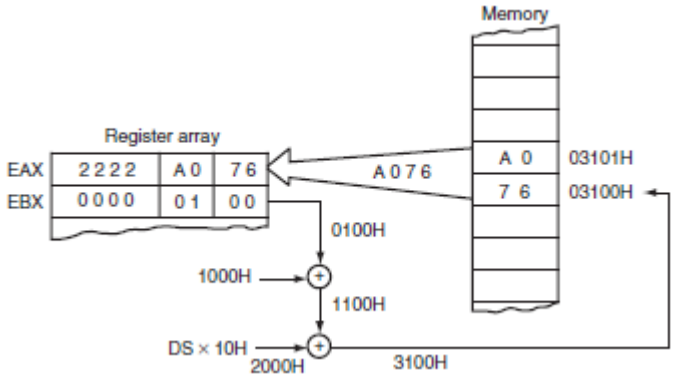


Figure 3-10:The operation of the MOV AX,[BX+1000H] instruction when BX=0100H and DS=0200H

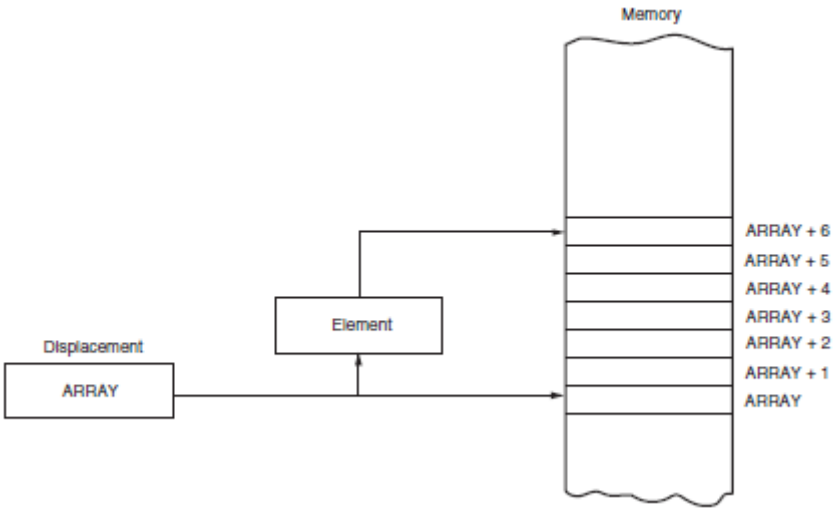


Figure 3-11: Register relative addressing used to address an element of ARRAY. The displacement addresses the start of ARRAY and DI accesses an element

Base Relative Plus Index Addressing

• This transfers a byte between a register and the memory-location addressed by a base-and an index-register plus a displacement.

• This often addresses a t2-dimensional array of memory-data.

Assembly Language	*Size*	*Operation*
MOV DH,[BX+DI+20H]	8 bits	Copies the byte contents of the data segment memory location addressed by the sum of BX, DI and 20H into DH
MOV AX,FILE[BX+DI]	16 bits	Copies the word contents of the data segment memory location addressed by the sum of FILE, BX and DI into AX
MOV LIST[BP+DI],CL	8 bits	Copies CL into the stack segment memory location addressed by the sum of LIST, BP, and DI
MOV LIST[BP+SI+4],DH	8 bits	Copies DH into the stack segment memory location addressed by the sum of LIST, BP, SI, and 4
MOV EAX,FILE[EBX+ECX+2]	32 bits	Copies the doubleword contents of the memory location addressed by the sum of FILE, EBX, ECX, and 2 into EAX

Table 3-8: Example base relative-plus-index instructions

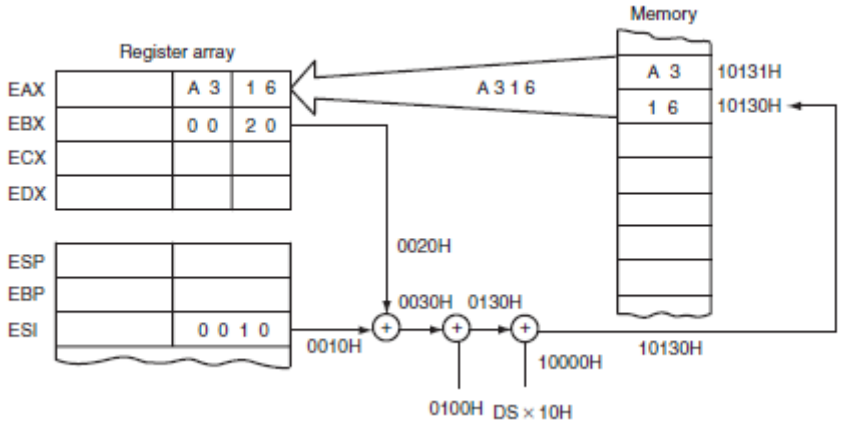


Figure 3-12: An example of base relative-plus-index addressing using a MOV AX,[BX+SI+100H] instruction. Note DS=1000H.

Program Memory-Addressing Modes

• This mode(used with the JMP and CALL instructions) consists of 3 distinct forms: direct, relative and indirect.

Direct Program Memory Addressing

• Many early microprocessors used this type of addressing for all jumps and calls.
• This is also used in high-level languages such as the BASIC language (GOTO instructions).
• The instructions store the address with the opcode.
• An *inter-segment jump* is a jump to any memory location within the entire memory system.
• The direct jump is often called a *far jump* because
it can jump to any memory location for the next instruction.
• In the real-mode, a far jump accesses any location within the first 1Mbyte of memory by changing both CS and IP.

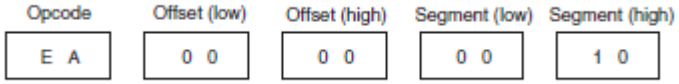


Figure 3-14: The 5-byte machine language version of a JMP[10000H] instruction

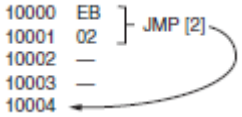


Figure 3-15: A JMP[2] instruction . This instruction skips over the 2-bytes of memory that follows the JMP instruction

Relative Program Memory Addressing

• The term relative means "relative to the instruction pointer(IP)".
• A 1-byte displacement is used in short jumps and
a 2-byte displacement is used with near jumps and calls.
• An *intra-segment jump* is a jump anywhere within the current code segment.

Assembly Language	*Operation*
JMP AX	Jumps to the current code segment location addressed by the contents of AX
JMP CX	Jumps to the current code segment location addressed by the contents of CX
JMP NEAR PTR[BX]	Jumps to the current code segment location addressed by the contents of the data segment location addressed by BX
JMP NEAR PTR[DI+2]	Jumps to the current code segment location addressed by the contents of the data segment memory location addressed by DI plus 2
JMP TABLE[BX]	Jumps to the current code segment location addressed by the contents of the data segment memory location address by TABLE plus BX
JMP ECX	Jumps to the current code segment location addressed by the contents of ECX
JMP RDI	Jumps to the linear address contained in the RDI register (64-bit mode)

Table 3-10: Examples of indirect program memory addressing

Indirect Program Memory Addressing

• If a 16-bit register holds the address of a JMP instruction, the jump is near. For example, if the BX register contains 1000H and a JMP BX instruction executes, the microprocessor jumps to offset address 1000H in the current code segment.
• If a relative register holds the address, the jump is also considered to be an indirect jump. For example, JMP[BX] refers to the memory location within the data segment at the offset address contained in BX. At this offset address is a 16-bit number that is used as the offset address in the intra-segment jump. This type of jumps is called an indirect-indirect or double-indirect jump.

Stack Memory Addressing Modes

- The stack holds data temporarily and stores the return-addresses used by procedures.
- The stack-memory is an LIFO memory.
- Data are placed onto the stack with a PUSH instruction and removed with a POP instruction.
- CALL instruction also uses the stack to hold the return-address for procedures.
  (RET instruction is used to remove the return address from the stack).
- The stack-memory is maintained by 2 registers: i) stack-pointer(SP) & 2) stack segment(SS).
- Whenever a data-word is pushed onto the stack,
  The high-order 8 bits are placed in the location addressed by SP-1
  The low-order 8 bits are placed in the location addressed by SP-2.
  The SP is then decremented by 2.
- The SP register always points to an area-of-memory located within the stack-segment.
- In real mode, the SP register adds to SS*10H to form the stack memory-address
- Whenever data are popped from the stack,
  The low-order 8 bits are removed from the location addressed by SP.
  The high-order 8 bits are removed from the location addressed by SP+1.
  The SP register is then incremented by 2.
- In 8086, PUSH & POP store or retrieve words of data but never bytes
- The PUSHA and POPA instructions either push or pop all the registers except segment-registers onto the stack.

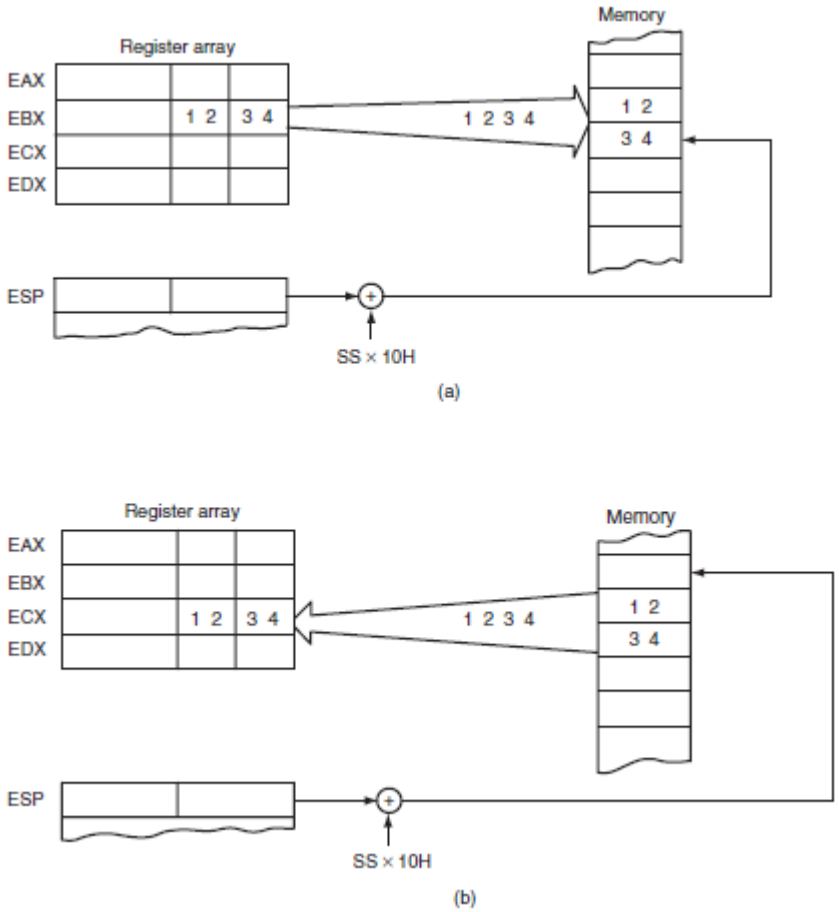


Figure 3-17:The PUSH & POP instructions: a)PUSH BX places the contents of BX into the stack b)POP CX removes data from the stack and places them into CX.

UNIT 3: DATA MOVEMENT INSTRUCTIONS

Machine Language

• Machine-language is the native binary-code that the microprocessor understands (and uses) as its instructions to control its operation.
• Length of Machine instructions= 1 to 13 bytes.
• In 8086, instructions are 16-bit mode-instructions (real-mode).
• In real-mode, 80386 & above assume that all instructions are 16-bit mode instructions (Fig 4-1).
In protected-mode, the upper-byte of the descriptor contains the D-bit. The bit selects either the 16- or 32-bit instruction-mode.
• In 32-bit instruction-mode format, first 2 bytes are called *override prefixes* because they are not always present.
1) First byte modifies the size of the operand-address used by the instruction &
2) Second byte modifies the register-size.
• If 80386 operate as 16-bit instruction-mode machines (real or protected mode) and a 32-bit register is used, the register-size prefix(66H) is appended to the front of the instruction (The address size prefix(67H) is used in the similar fashion)

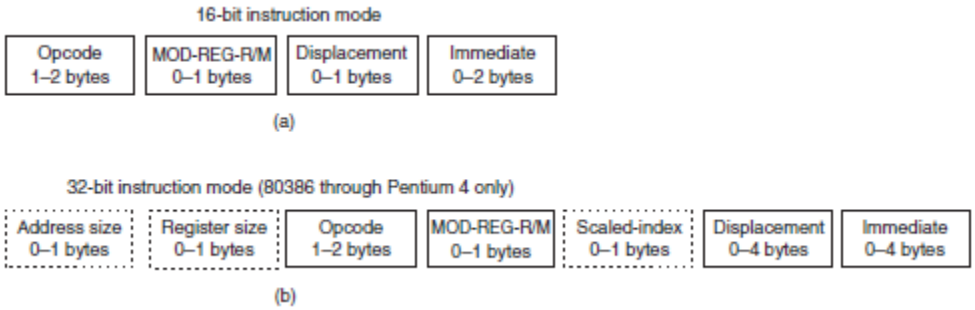


Figure 4-1: The formats of the 8086-Core2 instructions. a) The 16-bit form and b)The 32-bit form

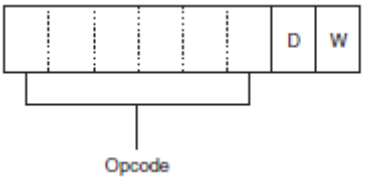


Figure 4-2: Byte-1 of many machine language instructions, showing the position of the D- and W-bits.

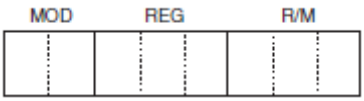


Figure 4-3: Byte of many machine language instructions, showing the position of the MOD, REG and R/M fields.

Opcode(Operation Code)

• This selects operation (addition, subtraction, move and so on) that is performed by the microprocessor.
• Opcode is either 1 or 2 bytes long for most machine language instructions (Figure 4-2).
• First 6 bits of first byte are the opcode. The remaining 2 bits are
i) If D=1, data flow from the R/M field to the register REG field. If D=0, data flow from the REG field to the R/M field.
ii) If W=1, the data size is a word; if W=0, the data size is always a byte. (The W-bit appears in most instructions, while the D-bit appears mainly with the MOV and some other instructions)

MOD Field

- This specifies addressing-mode(MOD) for the selected instruction (i.e. it selects the type of addressing and whether a displacement is present with the selected type) (Figure 4-3 & 4-3).
- If MOD=11, it selects the register-addressing mode (Table 4-1& 4-2).
- Register addressing uses the R/M field to specify a register instead of a memory-location.
- If MOD contains 00, 01 or 10, the R/M field selects one of the data memory-addressing modes.
- When MOD selects a data-memory addressing-mode, it indicates that the addressing-mode contains
  - → no displacement(00) {e.g. MOV AL, [DI]}
  - → 8-bit sign extended displacement(01) {e.g. MOV AL,[DI+2] }or
  - → 16-bit displacement(10) {e.g. MOV AL,[DI+1000H }

Table 4-1: MOD field for the 16-bit instruction mode

MOD	*Function*
00	No displacement
01	8-bit sign-extended displacement
10	16-bit signed displacement
11	R/M is a register

Table 4-2: MOD field for the 32-bit instruction mode

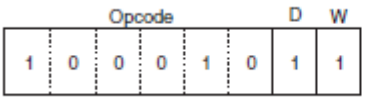
MOD	*Function*
00	No displacement
01	8-bit sign-extended displacement
10	32-bit signed displacement
11	R/M is a register

Table 4-3: REG and R/M when MOD=11 assignments.

Code	W = 0 *(Byte)*	W = 1 *(Word)*	W = 1 *(Doubleword)*
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

Table 4-4:16-bit R/M memory-addressing modes

R/M Code	*Addressing Mode*
000	DS:[BX+SI]
001	DS:[BX+DI]
010	SS:[BP+SI]
011	SS:[BP+DI]
100	DS:[SI]
101	DS:[DI]
110	SS:[BP]*
111	DS:[BX]



Opcode = MOV
D = Transfer to register (REG)
W = Word
MOD = R/M is a register
REG = BP
R/M = SP

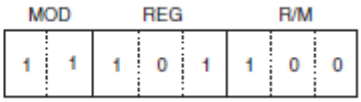


Figure 4-4: The 8BEC instruction placed into bytes 1 and 2 formats. This instruction is a MOV BP, SP

Special Addressing Mode

• This mode is used whenever memory-data are referenced by only the displacement mode of addressing for 16-bit instructions (Figure 4-5).

• Example:

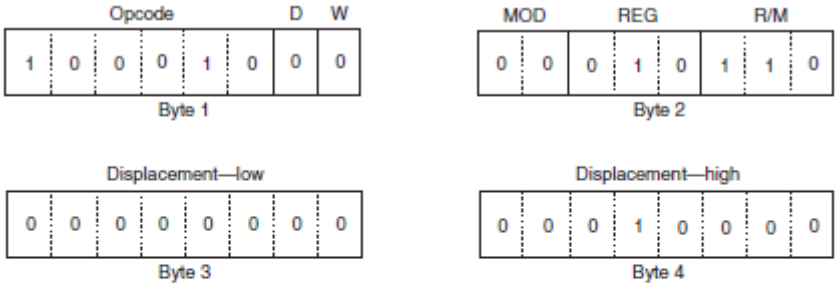
MOV [1000H],DL ;this instruction moves contents of DL into memory-location 1000H(Figure 4-6).
MOV NUMB, DL ;this instruction moves register DL into symbolic memory-location NUMB

• Whenever an instruction has only a displacement, the MOD field is always a 00 and the R/M field is always 110.

• If instruction contains no displacement and uses [BP] addressing mode, then you cannot actually use [BP] addressing mode without a displacement in machine language (Figure 4-7). The assembler takes care of this by using an 8-bit displacement(MOD=01) of 00H whenever the [BP] addressing mode appears in an instruction. (This means that the [BP] addressing mode assembles as a [BP+0], even though a [BP] is used in the instruction)

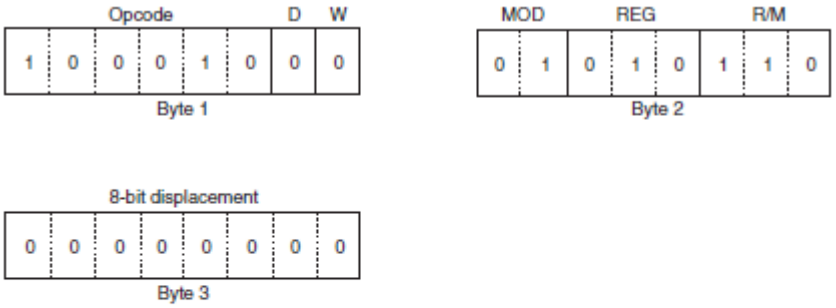
Opcode = MOV
D = Transfer to register (REG)
W = Byte
MOD = No displacement
REG = DL
R/M = DS:[DI]

Figure 4-5: A MOV DL, [DI] instruction converted to its machine language form



Opcode = MOV
D = Transfer from register (REG)
W = Byte
MOD = because R/M is [BP] (special addressing)
REG = DL
R/M = DS:[BP]
Displacement = 1000H

Figure 4-6: The MOV [1000H], DL instruction uses the special addressing mode



Opcode = MOV
D = Transfer from register (REG)
W = Byte
MOD = because R/M is [BP] (special addressing)
REG = DL
R/M = DS:[BP]
Displacement = 00H

Figure 4-7: The MOV [BP], DL instruction converted to binary machine language

32-bit Addressing Modes

• The 32-bit addressing modes are obtained by using the address-size prefix 67H (Table 4-5).

• The instruction MOV EAX, [EBX+4*ECX] is encoded as 67668B048BH. Both the address size(67H) and register size(66H) override appear in the instruction.

Table 4-5: 32-bit addressing modes selected by R/M

R/M Code	*Function*
000	DS:[EAX]
001	DS:[ECX]
010	DS:[EDX]
011	DS:[EBX]
100	Uses scaled-index byte
101	SS:[EBP]*
110	DS:[ESI]
111	DS:[EDI]

An Immediate Instruction

• MOV WORD PTR [BX+1000H],1234 ;this instruction moves a 1234H into word-sized memory-location addressed by the sum of 1000H, BX and DS*10H (Figure 4-9).

• WORD PTR directive indicates to the assembler that the instruction uses a word-sized memory pointer. (These directives are necessary only when it is not clear whether the operation is a byte or word. The MOV [BX], AL instruction is clearly a byte move; the MOV [BX], 9 instruction is not exact, and could therefore be a byte or word move. Here, the instruction must be coded as MOV BYTE PTR[BX],9 or MOV WORD PTR[BX],9. If not, the assembler flags it as an error because it cannot determine the intent of the instruction)

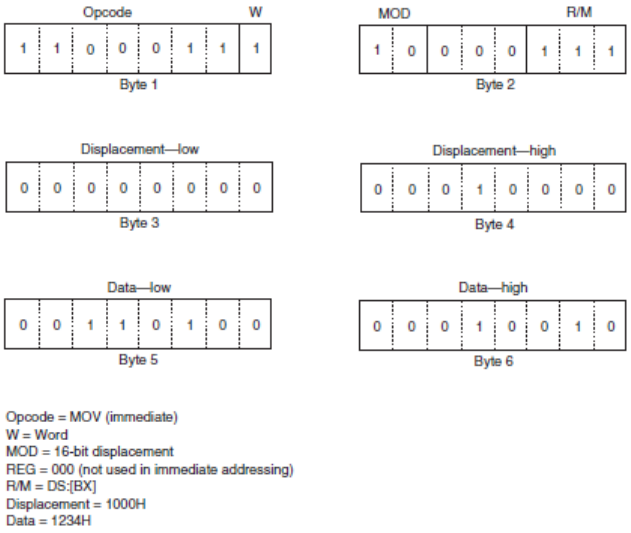


Figure 4-9: A MOV WORD PTR [BX+1000H],1234H instruction converted to binary machine language

Segment MOV Instructions

Table 4-6: Segment register selection

Code	*Segment Register*
000	ES
001	CS*
010	SS
011	DS
100	FS
101	GS

*Note: MOV CS,R/M and POP CS are not allowed.

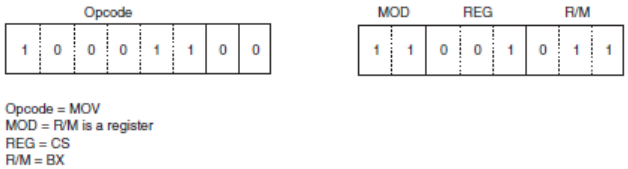


Figure 4-10: A MOV BX, CS instruction converted to binary machine language

The 64-bit Mode for the Pentium 4 & Core2

• In the 64-bit mode, an additional prefix called REX(register extension) is added (Figure 4-11).

• The REX prefix(which is encoded as a 40H-4Fh) follows other prefixes and is placed immediately before the opcode to modify it for 64-bit operation (Table 4-7).

• The purpose of the REX prefix is to modify the reg and r/m fields in the second byte of the instruction (Fig 4-11).

• REX is needed to be able to address registers R8 through R15.

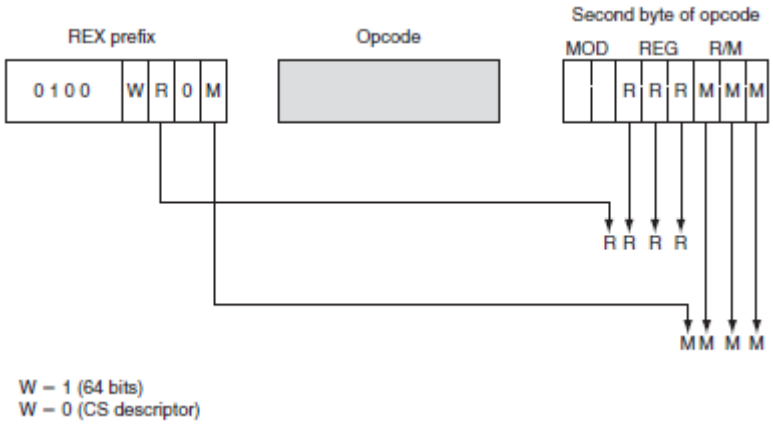


Figure 4-11: The application of REX without scaled index

Code	*Register*	*Memory*
0000	RAX	[RAX]
0001	RCX	[RCX]
0010	RDX	[RDX]
0011	RBX	[RBX]
0100	RSP	See note
0101	RBP	[RBP]
0110	RSI	[RSI]
0111	RDI	[RDI]
1000	R8	[R8]
1001	R9	[R9]
1010	R10	[R10]
1011	R11	[R11]
1100	R12	[R12]
1101	R13	[R13]
1110	R14	[R14]
1111	R15	[R15]

Table 4-7: The 64-bit register and memory designators for rrrr and mmmm

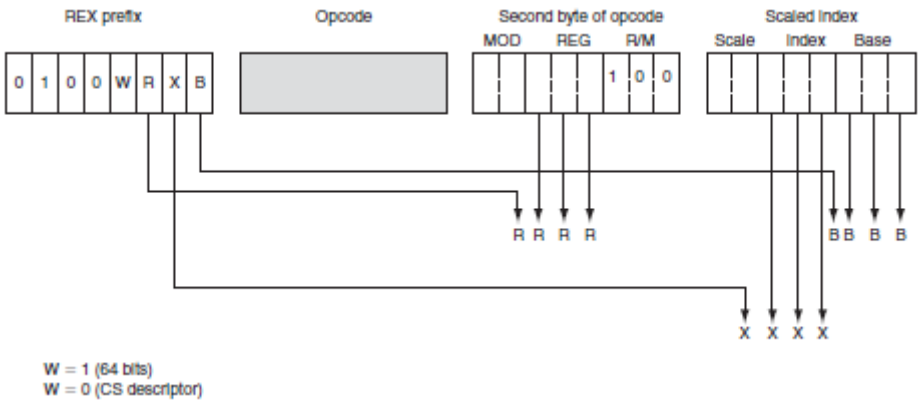


Figure 4-12: The scaled index byte and REX prefix for 64-bit operations

PUSH

- In 8086, PUSH always transfers 2 bytes of data to the stack (Figure 4-13).
- Source of data may be
  - → any 16-bit register
  - → immediate-data
  - → any segment-register or
  - → any 2 bytes of memory-data (Table 4-8)
- Whenever data are pushed onto stack,
  - → first data byte moves to the stack-segment memory location addressed by SP-1
  - → second data byte moves to the stack-segment memory location addressed by SP-2
- After the data are pushed onto stack, SP is decrement by 2. (SP=SP-2)
- PUSHF(push flags) copies the contents of flag register to the stack.
- PUSHA(push all) copies the contents of the internal register-set(except the segment registers) to the stack in the following order: AX, CX, DX, BX, SP, BP, SI and DI (Figure 4-14).
- After all registers are pushed, SP is decremented by 16. (SP=SP-16)

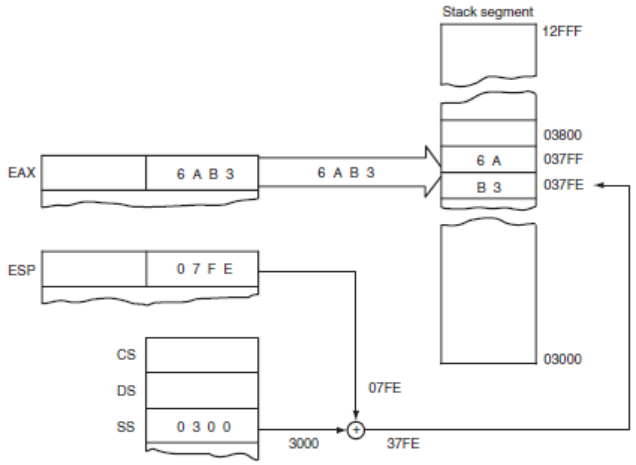


Figure 4-13: The effect of the PUSH AX instruction on ESP and stack memory locations 37FFH and 37FEH.This instruction is shown at the point after execution

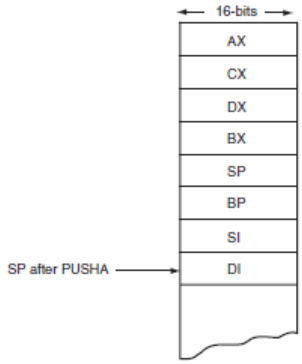


Figure 4-14: The operation of the PUSHA, showing location and order of stack data

Table 4-8: The PUSH instruction

Symbolic	*Example*	*Note*
PUSH reg16	PUSH BX	16-bit register
PUSH reg32	PUSH EDX	32-bit register
PUSH mem16	PUSH WORD PTR[BX]	16-bit pointer
PUSH mem32	PUSH DWORD PTR[EBX]	32-bit pointer
PUSH mem64	PUSH QWORD PTR[RBX]	64-bit pointer (64-bit mode)
PUSH seg	PUSH DS	Segment register
PUSH imm8	PUSH 'R'	8-bit immediate
PUSH imm16	PUSH 1000H	16-bit immediate
PUSHD imm32	PUSHD 20	32-bit immediate
PUSHA	PUSHA	Save all 16-bit registers
PUSHAD	PUSHAD	Save all 32-bit registers
PUSHF	PUSHF	Save flags
PUSHFD	PUSHFD	Save EFLAGS

POP

- POP removes data from the stack &
  places it into 16-bit register, segment-register or 16-bit memory-location (Fig4-15 & Table4-9).
- POPF(pop flags)
  → removes 2 bytes of data from the stack &
  → places it into the flag-register
- POPA(pop all)
  → removes 16 bytes of data from the stack &
  → places them into the following registers (in the order: DI, SI, BP, SP, BX, DX, CX and AX).
- This instruction is not available as an immediate POP.

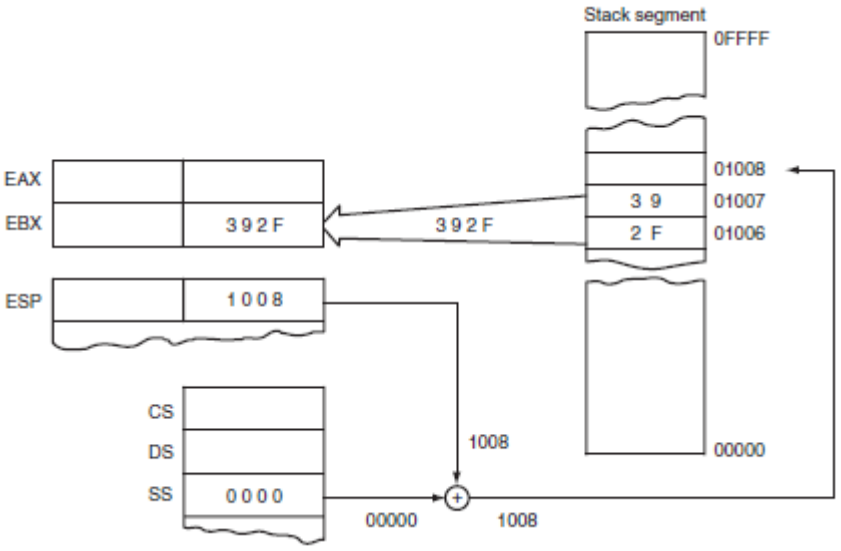


Figure 4-15: The POP BX instruction showing how data are removed from the stack. This instruction is shown after execution

Symbolic	*Example*	*Note*
POP reg16	POP CX	16-bit register
POP reg32	POP EBP	32-bit register
POP mem16	POP WORD PTR[BX+1]	16-bit pointer
POP mem32	POP DATA3	32-bit memory address
POP mem64	POP FROG	64-bit memory address (64-bit mode)
POP seg	POP FS	Segment register
POPA	POPA	Pops all 16-bit registers
POPAD	POPAD	Pops all 32-bit registers
POPF	POPF	Pops flags
POPFD	POPFD	Pops EFLAGS

Table 4-9: The POP instructions

LEA(Load-Effective Address)

- This loads a 16-bit register with the offset-address of the data specified by the operand(Table4-10).
- LEA BX,[DI] ;this loads offset-address specified by [DI] into register BX.
  Whereas MOV BX, [DI] ;this loads data stored at memory-location addressed by [DI] into BX.
- The OFFSET directive performs the same function as an LEA instruction if the operand is a displacement. For example, the MOV BX, OFFSET LIST performs the same function as LEA BX, LIST.
- Why is the LEA instruction available if the OFFSET directive accomplishes the same task?
  Ans: OFFSET only functions with simple operands such as LIST. It may not be used for an operand such as [DI], LIST [SI] and so on.
- OFFSET directive is more efficient than the LEA instruction for simple operands. Because it takes the microprocessor longer to execute the LEA BX, LIST instruction than the MOV BX, OFFSET LIST.
- The reason that the MOV BX, OFFSET LIST instruction executes faster is because the assembler calculates the offset address of LIST, whereas the microprocessor calculates the address for the LEA instruction.

Table 4-10: Load-effective address instructions

Assembly Language	*Operation*
LEA AX,NUMB	Loads AX with the offset address of NUMB
LEA EAX,NUMB	Loads EAX with the offset address of NUMB
LDS DI,LIST	Loads DS and DI with the 32-bit contents of data segment memory location LIST
LDS EDI,LIST1	Loads the DS and EDI with the 48-bit contents of data segment memory location LIST1
LES BX,CAT	Loads ES and BX with the 32-bit contents of data segment memory location CAT
LFS DI,DATA1	Loads FS and DI with the 32-bit contents of data segment memory location DATA1
LGS SI,DATA5	Loads GS and SI with the 32-bit contents of data segment memory location DATA5
LSS SP,MEM	Loads SS and SP with the 32-bit contents of data segment memory location MEM

LDS, LES, LFS, LGS and LSS

- The LDS, LES, LFS, LGS and LSS instructions load
  → any 16-bit register with an offset-address &
  → DS, ES, FS, GS or SS with a segment-address.
- For example, LDS BX,[DI] ;this instruction transfers 32-bit number(addressed by DI in the segment) into the BX and DS registers (Figure 4-17).
- These instructions cannot use the register addressing-mode (MOD=11).

(These instructions use any of the memory-addressing modes to access a 32-bit section of memory that contains both the segment and offset address. The 32-bit section of memory contains a 16-bit offset and 16-bit segment address)

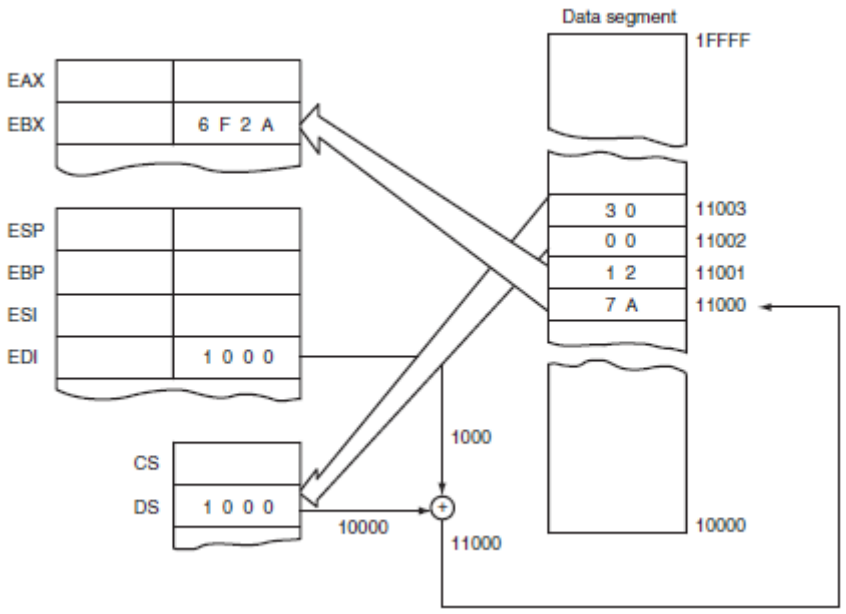


Figure 4-17:The LDS BX,[DI] instruction loads register BX from addresses 11000H and 11001H and register DS from locations 11002H and 11003H.This instruction is shown at the point just before DS changes to 3000H and BX changes to 127AH.

String Data Transfers (LODS, STOS, MOVS, INS & OUTS)

The Direction flag

- D flag (located in flag-register) is used only with the string instructions.
- D flag selects the auto-increment(D=0) or the auto-decrement(D=1) operation for the DI and SI registers during string operations.
- CLD instruction clears the D flag(D=0) &
  STD instruction sets the D flag(D=1).{.'. CLD instruction selects the auto-increment mode and STD selects the auto-decrement mode}
- Whenever a string instruction transfers a byte, DI and/or SI is incremented or decremented by 1.
  If a word is transferred, DI and/or SI is incremented or decremented by 2.
- For example, STOSB instruction uses the DI register to address a memory-location. When STOSB executes, only the DI register is incremented or decremented without affecting SI.

DI and SI

- The DI offset-address accesses data in the extra-segment for all string instructions that use it. The SI offset-address accesses data, by default, in the data-segment. (The DI segment assignment is always in the extra segment when a string instruction executes. This assignment cannot be changed)
- The reason that one pointer addresses data in the extra-segment and the other in the data-segment is so that the MOVS instruction can move 64KB of data from one segment of memory to another.

LODS(load string)

- This loads AL or AX with data stored at the data-segment offset-address indexed by SI(Table4-11).
- LODSB(loads a byte) instruction causes a byte to be loaded into AL
  LODSW(loads a word) instruction causes a word to be loaded into AX (Figure 4-18).
- After loading AL with a bytes, SI is incremented if D=0 or decremented if D=1.

Table 4-11: Forms of the LODS instruction

Assembly Language	*Operation*
LODSB	AL = DS:[SI]; SI = SI ± 1
LODSW	AX = DS:[SI]; SI = SI ± 2
LODSD	EAX = DS:[SI]; SI = SI ± 4
LODSQ	RAX = [RSI]; RSI = RSI ± 8 (64-bit mode)
LODS LIST	AL = DS:[SI]; SI = SI ± 1 (if LIST is a byte)
LODS DATA1	AX = DS:[SI]; SI = SI ± 2 (if DATA1 is a word)
LODS FROG	EAX = DS:[SI]; SI = SI ± 4 (if FROG is a doubleword)

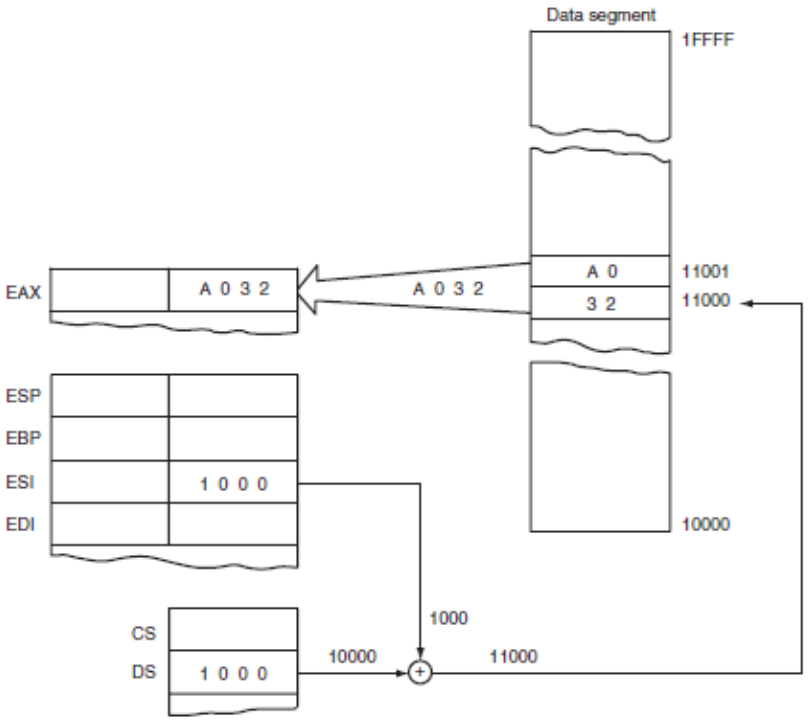


Figure 4-18: The operation of the LODSW instruction if DS=1000H, D=0, 11000H=32, and 11001H=A0.This instruction is shown after AX is loaded from memory but before SI increments by 2.

STOS(store string)

- This stores AL or AX at the extra-segment memory-location addressed by DI (Table 4-12).
- STOSB(stores a byte) instruction stores the byte in AL at the extra-segment memory-location addressed by DI
  STOSW(stores a word) instruction stores AX in extra-segment memory-location addressed by DI
- After the byte(AL) is stored, DI is incremented if D=0 or decremented if D=1.

STOS with a REP

- The repeat prefix(REP) is added to any string data transfer instruction except the LODS instruction.('.' it doesn't make any sense to perform a repeated LODS operation)
- REP prefix causes CX to decrement by 1 each time the string instruction executes.
  After CX decrement, the string instruction repeats.
  If CX reaches 0, the instruction terminates and the program continues with the next sequential instruction.

(For example, if CX is loaded with 100 and a *REP STOSB* instruction executes, the microprocessor automatically repeats the STOSB instruction 100 times)

Table 4-12: Forms of the STOS instruction

Assembly Language	*Operation*
STOSB	ES:[DI] = AL; DI = DI ± 1
STOSW	ES:[DI] = AX; DI = DI ± 2
STOSD	ES:[DI] = EAX; DI = DI ± 4
STOSQ	[RDI] = RAX; RDI = RDI ± 8 (64-bit mode)
STOS LIST	ES:[DI] = AL; DI = DI ± 1 (if LIST is a byte)
STOS DATA3	ES:[DI] = AX; DI = DI ± 2 (if DATA3 is a word)
STOS DATA4	ES:[DI] = EAX; DI = DI ± 4 (if DATA4 is a doubleword)

MOVS

- This transfers a bytes(or word) from the data-segment memory-location addressed by SI to the extra-segment memory-location addressed by DI (Table 4-14).
- SI/DI is incremented if D=0 or decremented if D=1.
- This is the only memory-to-memory transfer allowed in the 8086 microprocessor.
- The destination-operand must always be located in the extra-segment. While the source-operand located in the data-segment may be overridden so that another segment may be used.

Table 4-14: Forms of the MOVS instruction

Assembly Language	*Operation*
MOVSB	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (byte transferred)
MOVSW	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2 (word transferred)
MOVSD	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4 (doubleword transferred)
MOVSQ	[RDI] = [RSI]; RDI = RDI ± 8; RSI = RSI ± 8 (64-bit mode)
MOVS BYTE1, BYTE2	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (byte transferred if BYTE1 and BYTE2 are bytes)
MOVS WORD1,WORD2	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2 (word transferred if WORD1 and WORD2 are words)
MOVS TED,FRED	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4 (doubleword transferred if TED and FRED are doublewords)

INS

• This transfers a byte (or word) of data from an I/O device into the extra-segment memory-location addressed by DI.
• I/O address is contained in the DX register (Table 4-15).
• INSB inputs data from an 8-bit I/O device & stores it in byte-sized memory-location indexed by SI.
INSW inputs 16-bit I/O data and stores it in a word-sized memory-location.
• This instruction can be repeated using the REP prefix, which allows an entire block of input-data to be stored in the memory from an I/O device.

Table 4-15: Forms of the INS instruction

Assembly Language	*Operation*
INSB	ES:[DI] = [DX]; DI = DI ± 1 (byte transferred)
INSW	ES:[DI] = [DX]; DI = DI ± 2 (word transferred)
INSD	ES:[DI] = [DX]; DI = DI ± 4 (doubleword transferred)
INS LIST	ES:[DI] = [DX]; DI = DI ± 1 (if LIST is a byte)
INS DATA4	ES:[DI] = [DX]; DI = DI ± 2 (if DATA4 is a word)
INS DATA5	ES:[DI] = [DX]; DI = DI ± 4 (if DATA5 is a doubleword)

OUTS

• This transfers a byte (or word) of data from the data-segment memory-location address by SI to an I/O device.
• I/O device is addressed by DX register (Table 4-16).

Table 4-16: Forms of the OUTS instruction

Assembly Language	*Operation*
OUTSB	[DX] = DS:[SI]; SI = SI ± 1 (byte transferred)
OUTSW	[DX] = DS:[SI]; SI = SI ± 2 (word transferred)
OUTSD	[DX] = DS:[SI]; SI = SI ± 4 (doubleword transferred)
OUTS DATA7	[DX] = DS:[SI]; SI = SI ± 1 (if DATA7 is a byte)
OUTS DATA8	[DX] = DS:[SI]; SI = SI ± 2 (if DATA8 is a word)
OUTS DATA9	[DX] = DS:[SI]; SI = SI ± 4 (if DATA9 is a doubleword)

Miscellaneous Data Transfer Instructions

XCHG

• This exchanges contents of a register with contents of any other register or memory-location.
• This cannot exchange segment registers or memory-to-memory data(Table 4-17).
• This can exchange either byte or word.
• This can use any addressing-mode except immediate addressing.
• XCHG instruction, using the 16-bit AX register with another 16-bit register, is the most efficient exchange. This instruction occupies 1 byte of memory. Other XCHG instructions require 2 or more bytes of memory depending on the addressing-mode selected.
• When using a memory-addressing mode, it doesn't matter which operand addresses memory.
• XCHG AL,[DI] instruction is identical to XCHG [DI],AL instruction, as far as the assembler is concerned.

Table 4-17: Forms of the XCHG instruction

Assembly Language	*Operation*
XCHG AL,CL	Exchanges the contents of AL with CL
XCHG CX,BP	Exchanges the contents of CX with BP
XCHG EDX,ESI	Exchanges the contents of EDX with ESI
XCHG AL,DATA2	Exchanges the contents of AL with data segment memory location DATA2
XCHG RBX,RCX	Exchange the contents of RBX with RCX (64-bit mode)

LAHF and SAHF

• LAHF transfers the rightmost 8 bits of the flag-register into the AH register
SAHF instruction transfers the AH register into the rightmost 8 bits of the flag-register
• These instructions are used because they were designed as bridge-instructions.
• These instructions allowed 8085 software to be translated into 8086 software by a translation-program.

XLAT

- This converts the contents of the AL register into a number stored in a memory-table.
- This performs the direct table-lookup technique often used to convert one code to another.
- This first adds the contents of AL to BX to form a memory-address within the data-segment.
  This then copies the contents of this address into AL (Figure 4-19).
- This is the only instruction that adds an 8-bit number to a 16-bit number.

Example 4-11: Program to convert BCD to 7-segment code

```
TABLE  DB 3FH,06H,5BH,4FH,66H,6DH,7DH,27H,7FH,6FH     :lookup table
LOOK:  MOV AL,5                                       ;load AL with 5
       LEA BX, TABLE                                  ;address lookup table
       XLAT                                           ;convert
```

\

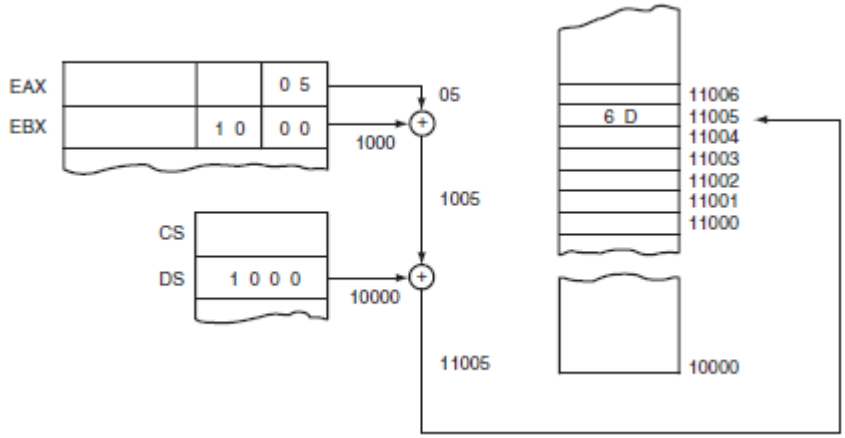


Figure 4-19: The operation of the XLAT instruction at the point just before 6DH is loaded into AL

IN and OUT

- IN transfers data from an external I/O device into AL or AX
  OUT transfers data from AL or AX to an external I/O device (Table 4-18).
- Two forms of I/O addressing are
  1) Fixed-port addressing allows data-transfer between AL/AX using an 8-bit I/O port-address.
     This is called fixed-port addressing because
     port-number follows the instruction's opcode(just as with immediate addressing).
  2) Variable-port addressing allows data-transfers between AL/AX and a 16-bit port-address.
     This is called variable-port addressing because
     the I/O port-number is stored in register DX, which can be changed(varied) during the execution of a program. (The 16-bit port I/O port addressing appears on the address bus pin connections A0-A15).

Table 4-18: IN and OUT instructions

Assembly Language	*Operation*
IN AL,p8	8 bits are input to AL from I/O port p8
IN AX,p8	16 bits are input to AX from I/O port p8
IN EAX,p8	32 bits are input to EAX from I/O port p8
IN AL,DX	8 bits are input to AL from I/O port DX
IN AX,DX	16 bits are input to AX from I/O port DX
IN EAX,DX	32 bits are input to EAX from I/O port DX
OUT p8,AL	8 bits are output to I/O port p8 from AL
OUT p8,AX	16 bits are output to I/O port p8 from AX
OUT p8,EAX	32 bits are output to I/O port p8 from EAX
OUT DX,AL	8 bits are output to I/O port DX from AL
OUT DX,AX	16 bits are output to I/O port DX from AX
OUT DX,EAX	32 bits are output to I/O port DX from EAX

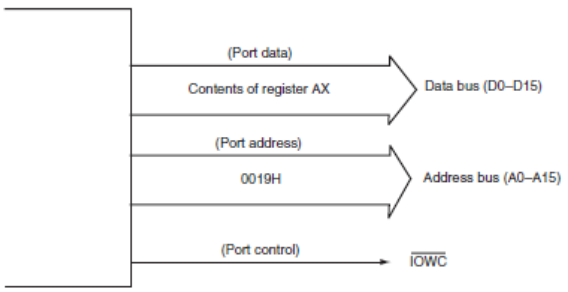


Figure 4-20: The signals found in the microprocessor-based system for an OUT 19H,AX instruction

MOVSX & MOVZX

- These instructions move data, & at the same time either sign or zero extends it (Table 4-19).
- A number is zero-extended when zero is copied into the most significant part. For example, if an 8-bit 34H is zero-extended into a 16-bit number, it becomes 0034H.
- Zero-extension is used to convert unsigned 8-bit numbers into unsigned 16-bit numbers by using the MOVZX.
- A number is sign-extended when its sign-bit is copied into the most significant part. For example, if an 8-bit 84H is sign-extended into a 16-bit number, it becomes FF84H.
- Sign-extension is used to convert 8-bit signed numbers into 16-bit signed numbers by using the MOVSX.

Table 4-19: The MOVSX and MOVZX instructions

Assembly Language	*Operation*
MOVSX CX,BL	Sign-extends BL into CX
MOVSX ECX,AX	Sign-extends AX into ECX
MOVSX BX,DATA1	Sign-extends the byte at DATA1 into BX
MOVSX EAX,[EDI]	Sign-extends the word at the data segment memory location addressed by EDI into EAX
MOVSX RAX,[RDI]	Sign-extends the doubleword at address RDI into RAX (64-bit mode)
MOVZX DX,AL	Zero-extends AL into DX
MOVZX EBP,DI	Zero-extends DI into EBP
MOVZX DX,DATA2	Zero-extends the byte at DATA2 into DX
MOVZX EAX,DATA3	Zero-extends the word at DATA3 into EAX
MOVZX RBX,ECX	Zero-extends ECX into RBX

BSWAP(Byte-Swap)

- This is available only in the 80486-Pentium 4 microprocessors.
- This takes the contents of any 32-bit register and swaps 1st byte with 4th, & 2nd with 3rd.
- For example, assume EAX=22334456H, then BSWAP EAX instruction swaps bytes resulting in EAX=56443322H
- This is used to convert data between the big- and little-endian forms.

CMOV(Conditional Move)

- This is available only in the Pentium Pro-Core2 microprocessors (Table 4-20).
- This moves the data only if the condition is true. For example, CMOVZ instruction moves data only if result from some prior instruction was a zero.
- The destination is limited to only a 16- or 32-bit register, but the source can be a 16- or 32-bit register or memory-location.

Table 4-20: The conditional move instructions

Assembly Language	*Flag(s) Tested*	*Operation*
CMOVB	C = 1	Move if below
CMOVAE	C = 0	Move if above or equal
CMOVBE	Z = 1 or C = 1	Move if below or equal
CMOVA	Z = 0 and C = 0	Move of above
CMOVE or CMOVZ	Z = 1	Move if equal or move if zero
CMOVNE or CMOVNZ	Z = 0	Move if not equal or move if not zero
CMOVL	S ! = O	Move if less than
CMOVLE	Z = 1 or S ! = O	Move if less than or equal
CMOVG	Z = 0 and S = O	Move if greater than
CMOVGE	S = O	Move if greater than or equal
CMOVS	S = 1	Move if sign (negative)
CMOVNS	S = 0	Move if no sign (positive)
CMOVC	C = 1	Move if carry
CMOVNC	C = 0	Move if no carry
CMOVO	O = 1	Move if overflow
CMOVNO	O = 0	Move if no overflow
CMOVP or CMOVPE	P = 1	Move if parity or move if parity even
CMOVNP or CMOVPO	P = 0	Move if no parity or move if parity odd

Segment Override Prefix

- This allows the programmers to deviate from the default-segment (Table 4-21).
- This may be added to almost any instruction in any memory addressing-mode.
- This is an additional byte that appends the front of an instruction to select an alternate segment-register.
- Only instructions that cannot be prefixed are JMP and CALL that must use the code-segment register for address generation.
- For example, MOV AX,ES ;[DI] this instruction addresses extra-segment instead of data-segment.

Table 4-21: Instructions that include segments override prefixes

Assembly Language	*Segment Accessed*	*Default Segment*
MOV AX,DS:[BP]	Data	Stack
MOV AX,ES:[BP]	Extra	Stack
MOV AX,SS:[DI]	Stack	Data
MOV AX,CS:LIST	Code	Data
MOV ES:[SI],AX	Extra	Data
LODS ES:DATA1	Extra	Data
MOV EAX,FS:DATA2	FS	Data
MOV GS:[ECX],BL	GS	Data

Directive

- This indicates how an operand or section of a program is to be processed by the assembler.
- Some directives generate and store information in the memory; others do not.

For e.g. DB(define byte) directive stores bytes of data in the memory whereas BYTE PTR directive never stores data. The BYTE PTR directive indicates size of data referenced by a pointer or index register.

Table 4.22: Common MASM directives

Directive	*Function*
.286	Selects the 80286 instruction set
.286P	Selects the 80286 protected mode instruction set
.386	Selects the 80386 instruction set
.386P	Selects the 80386 protected mode instruction set
.486	Selects the 80486 instruction set
.486P	Selects the 80498 protected mode instruction set
.586	Selects the Pentium instruction set
.586P	Selects the Pentium protected mode instruction set
.686	Selects the Pentium Pro–Core2 instruction set
.686P	Selects the Pentium Pro–Core2 protected mode instruction set
.287	Selects the 80287 math coprocessor
.387	Selects the 80387 math coprocessor
.CODE	Indicates the start of the code segment (models only)
.DATA	Indicates the start of the data segment (models only)
.EXIT	Exits to DOS (models only)
.MODEL	Selects the programming model
.STACK	Selects the start of the stack segment (models only)
.STARTUP	Indicates the starting instruction in a program (models only)
ALIGN n	Align to boundary n (n = 2 for words, n = 4 for doublewords)
ASSUME	Informs the assembler to name each segment (full segments only)
BYTE	Indicates byte-sized as in BYTE PTR
DB	Defines byte(s) (8 bits)
DD	Defines doubleword(s) (32 bits)
DQ	Defines quadwords(s) (64 bits)
DT	Defines ten byte(s) (80 bits)
DUP	Generates duplicates
DW	Define word(s) (16 bits)
DWORD	Indicates doubleword-sized, as in DWORD PTR
END	Ends a program file
ENDM	Ends a MACRO sequence
ENDP	Ends a procedure
ENDS	Ends a segment or data structure
EQU	Equates data or a label to a label
FAR	Defines a far pointer, as in FAR PTR
MACRO	Designates the start of a MACRO sequence
NEAR	Defines a near pointer, as in NEAR PTR
OFFSET	Specifies an offset address
ORG	Sets the origin within a segment
OWORD	Indicates octalwords, as in OWORD PTR
PROC	Starts a procedure
PTR	Designates a pointer
QWORD	Indicates quadwords, as in QWORD PTR
SEGMENT	Starts a segment for full segments
STACK	Starts a stack segment for full segments
STRUC	Defines the start of a data structure
USES	Automatically pushes and pops registers
USE16	Uses 16-bit instruction mode
USE32	Uses 32-bit instruction mode
WORD	Indicates word-sized, as in WORD PTR

Storing Data in a Memory Segment

• DB(define byte), DW(define word) and DD(define doubleword) directives are used to define & store memory-data.

• If a numeric-coprocessor executes software in the system, the DQ(define quadword) and DT(define ten bytes) directives can also be used.

• These directives label a memory-location with a symbolic-name and indicate its size.

• Memory is reserved for use in the future by using a question mark(?) as an operand for a DB or DW directive.

• When a ? is used in place of a numeric(ASCII) value, the assembler sets aside a location and does not initialize it to any specific value.

• 10 DUP(?) reserves 10 locations of memory, but stores no specific value in any of the 10 locations.

• If a number appears within the ( ) part, the assembler initializes the reserved section of memory with the data indicated. For example, LIST DB 10 DUP(2) reserves 10 bytes of memory for array LIST and initializes each location with a 2H.

• ALIGN directive makes sure that the memory-arrays are stored on word-boundaries.

ALIGN 2 places data on word-boundaries &

ALIGN 4 places them on doubleword-boundaries

• A word stored at an odd-numbered memory-location takes twice as long to access as a word stored at an even-numbered memory-location.

• ENDS directive indicates the end of the segment.

• The name of the segment can be anything that the programmers desire to call it.

Example 4-13: Program to illustrate usage of DB, DW, DUP, ALIGN, ENDS directives

```
LIST_SEG SEGMENT
        DATA1 DB 1,2,3      ;define bytes
        DB 45H              ;define hexadecimal
        DB 11110000B        ;define binary
        DATA2 DW 1234       ;define word
        DW LIST1            ;define symbolic
        LISTA DB ?          ;reserves 1 byte

        ALIGN 2             ;set word boundary

        LISTB DB 10 DUP(?)      ;reserves 10 bytes
        SIXES BB 10 DUP(6)      ;reserve 10 bytes initialized with 6

LIST_SEG ENDS
```

ASSUME, EQU and ORG

• EQU directive equates a numeric, ASCII or label to another label.

• Equates make a program clearer and simplify debugging.

• ORG(origin) statement changes the starting offset-address of the data in the data-segment to location 300H.

• ASSUME statement tells the assembler what names have been chosen for the code-, data-, extra- and stack-segments.

Example 4-14: Program to illustrate usage of ASSUME, EQU and ORG directives

```
DATA_SEGMENT SEGMENT
        ORG 300H
        TEN EQU 10
        NINE EQU 9
        RES DB ?
DATA_SEG ENDS

CODE_SEG SEGMENT
        ASSUME CS:CODE_SEG,DS:DATA_SEG
        MOV AL,TEN
        ADD AL,NINE
        MOV RES,AL
CODE_SEG ENDS
```

PROC & ENDP

- PROC and ENDP directives indicate the start and end of a procedure(subroutine).
- These directives force structure because the procedure is clearly defined.
- The PROC directive indicates the start of a procedure(PROC must be followed with a NEAR or FAR)
- A NEAR procedure is one that resides in the same code-segment as the program.
  A FAR procedure may reside at any location in the memory. (Often call NEAR procedure is considered to be local and the call FAR procedure is considered to be global)
- The USES statement indicates which registers are used by the procedures,
  so that the assembler can automatically
  → save them before your procedure begins and
  → restore them before the procedure ends with the RET instructions.

Example 4-16:Procedure that adds BX,CX and DX, and stores the sum in register AX.

```
ADDING PROC NEAR USES BX,CX,DX      ;start of procedure
        PUSH BX
        PUSH CX
        PUSH DX
        ADD BX,CX
        ADD BX,DX
        MOV AX,BX
       RET
        POP DX
        POP CX
        POP BX
        Ret 0000h
ADDING ENDP                         ;end of procedure
```

Memory Organization

- The assembler(MASM) for the microprocessor can be used in two ways:
  1) With *models* that are unique to a particular assembler &
  2) With *full-segment definitions* that allow complete control over the assembly process and are universal to all assemblers

Models

- Memory-models are unique to the MASM assembler-program.
- There are many models available to the MASM assembler, ranging from tiny to huge.
  1) TINY model requires that all software & data fit into one 64KB memory-segment, it is useful for many small programs.
  2) SMALL model requires that only one data-segment be used with one code-segment for a total of 128KB of memory.
- Models are easier to use for simple tasks.
- Models are also used with assembly language procedures that are used by high level languages such as C/C++.
- .EXIT 0 directive returns to DOS with an error code of 0.
- @DATA are used to identify various segments.
- If the .STARTUP directive is used, the MOV AX,@DATA followed by MOV DS, AX statements can be eliminated.

Example 4-18: Using model format, program to copy the contents of a 100 byte block of memory(LISTA) into a second 100 byte block of memory(LISTB)

```
.MODEL SMALL
.STACK 100H
.DATA
        LISTA DB 100 DUP(?)
        LISTB DB 100 DUP(?)
.CODE
        HERE:MOV AX,@DATA
        MOV ES,AX
        MOV DS,AX
        CLD
        LEA SI, LISTA
        LEA DI,LISTB
        MOV CX,100
        REP MOVSB
.       EXIT 0
        END HERE
```

Full Segment Definitions

• The full-segment definitions are common to most assemblers including the Intel assembler, and are often used for software development.

• The full-segment definitions offer better control over the assembly language task and are recommended for complex programs

• ASSUME statement tells the assembler and linker that the name used for the code-segment(CS) is CODE_SEG.

• END statement indicates the end of the program and the location of the first instruction executed.

Example 4-18: Using full segment definition, program to copy the contents of a 100 byte block of memory(LISTA) into a second 100 byte block of memory(LISTB)

```
STACK_SEG SEGMENT
        DW 100H DUP(?)
STACK_SEG ENDS

DATA_SEG SEGMENT
        LISTA DB 100 DUP(?)
        LISTB DB 100 DUP(?)
DATA_SEG ENDS

CODE_SEG SEGMENT
        ASSUME CS:CODE_SEG,DS:DATA_SEG
        ASSUME SS:STACK_SEG
    MAIN PROC FAR
        MOV AX,DATA_SEG
        MOV ES,AX
        MOV DS,AX
        CLD
        LEA SI, LISTA
        LEA DI,LISTB
        MOV CX,100
        REP MOVSB
        MOV AH,4C
        INT 21H
    MAIN ENDP
CODE_SEG ENDS
        END MAIN
```

Example 4-20: Full segment program that reads a key and displays it.

```
CODE_SEG SEGMENT
      ASSUME CS:CODE_SEG
  MAIN PROC FAR
        MOV AH,06H          ;read a key
        MOV DL,0FFH
        INT 21H
        JE MAIN             ;if no key typed
        CMP AL,'@'
        JE MAIN1            ;if an @ key
        MOV AH,06H          ;display key(echo)
        MOV DL,AL
        INT 21H
        JMP MAIN            ;repeat
 MAIN1:MOV AH,4CH           ;exit to DOS
        INT 21H
MAIN ENDP
        END MAIN
```

UNIT 3(CONT.): ARITHMETIC AND LOGIC INSTRUCTIONS

ADD(Addition)

- This is used to add byte/word of data of 2 registers or a register & a memory-location
- The only types of addition not allowed are memory-to-memory and segment register (Table 5-1).

Table 5-1: Example addition instructions

Assembly Language	*Operation*
ADD AL,BL	AL = AL + BL
ADD CX,DI	CX = CX + DI
ADD EBP,EAX	EBP = EBP + EAX
ADD CL,44H	CL = CL + 44H
ADD BX,245FH	BX = BX + 245FH
ADD EDX,12345H	EDX = EDX + 12345H
ADD [BX],AL	AL adds to the byte contents of the data segment memory location addressed by BX with the sum stored in the same memory location
ADD CL,[BP]	The byte contents of the stack segment memory location addressed by BP add to CL with the sum stored in CL
ADD AL,[EBX]	The byte contents of the data segment memory location addressed by EBX add to AL with the sum stored in AL
ADD BX,[SI+2]	The word contents of the data segment memory location addressed by SI + 2 add to BX with the sum stored in BX
ADD CL,TEMP	The byte contents of data segment memory location TEMP add to CL with the sum stored in CL
ADD BX,TEMP[DI]	The word contents of the data segment memory location addressed by TEMP + DI add to BX with the sum stored in BX
ADD [BX+D],DL	DL adds to the byte contents of the data segment memory location addressed by BX + DI with the sum stored in the same memory location
ADD BYTE PTR [DI],3	A 3 adds to the byte contents of the data segment memory location addressed by DI with the sum stored in the same location
ADD BX,[EAX+2*ECX]	The word contents of the data segment memory location addressed by EAX plus 2 times ECX add to BX with the sum stored in BX
ADD RAX,RBX	RBX adds to RAX with the sum stored in RAX (64-bit mode)
ADD EDX,[RAX+RCX]	The doubleword in EDX is added to the doubleword addressed by the sum of RAX and RCX and the sum is stored in EDX (64-bit mode)

Register Addition

- Whenever arithmetic and logic instructions execute, the contents of the flag-register changes.
- Any ADD instruction modifies contents of sign, zero, carry, auxiliary, carry, parity & overflow flags.

Example 5-1: Program to compute AX=BX+CX+DX.

```
ADD AX,BX
ADD AX,CX
ADD AX,DX
```

Immediate Addition

- Immediate addition is employed whenever constant-data are added.

Example 5-2: Program to add immediate data

```
MOV DL,12H
ADD DL,33H
```

Memory-to-Register Addition

Example 5-3: Program to add two consecutive bytes of data(stored at the data segment offset locations NUMB and NUMB+1) to the AL register.

```
LEA DI,NUMB      ;address NUMB
MOV AL,0         ;clear sum
ADD AL,[DI]      ;add NMB
ADD AL,[DI+1]    ;add NUMB+1
```

Array Addition

Example 5-4: Program to add the contents of array element at indices 3, 5 and 7

```
MOV AL,0                ;clear sum
MOV SI,3                ;address element 3
ADD AL,ARRAY[SI]        ;add element 3
ADD AL,ARRAY[SI+2]      ;add element 5
ADD AL,ARRAY[SI+4]      ;add element 7
```

INC(Increment Addition)

• This adds 1 to any register or memory-location, except a segment-register (Table 5-2).

• With indirect memory increments, the size of the data must be described by using the BYTE PTR, WORD PTR directives.

The reason is that the assembler cannot determine if, for example, INC [DI] instruction is a byte or word sized increment.

The INC BYTE PTR[DI] instruction clearly indicates byte sized memory data.

Example 5-6: Using INC instruction, program to add two consecutive bytes of data(stored at the data segment offset locations NUMB and NUMB+1) to the AL register.

```
LEA DI,NUMB      ;address
MOV AL,0         ;clear sum
ADD AL,[DI]      ;add NUMB
INC DI           ;increment DI
ADD AL,[DI]      ;add NUMB+1
```

Table 5-2: Example increment instructions

Assembly Language	*Operation*
INC BL	BL = BL + 1
INC SP	SP = SP + 1
INC EAX	EAX = EAX + 1
INC BYTE PTR[BX]	Adds 1 to the byte contents of the data segment memory location addressed by BX
INC WORD PTR[SI]	Adds 1 to the word contents of the data segment memory location addressed by SI
INC DWORD PTR[ECX]	Adds 1 to the doubleword contents of the data segment memory location addressed by ECX
INC DATA1	Adds 1 to the contents of data segment memory location DATA1
INC RCX	Adds 1 to RCX (64-bit mode)

ADC(Addition with Carry)

- This adds the bit in the carry-flag(C) to the operand-data (Table 5-3).
- This mainly appears in software that adds numbers that are wider than 16 bits in the 8086.

Example 5-7: To add the 32-bit number in BX and AX to the 32-bit number in DX and CX (Figure 5-1)

```
ADD AX,CX
ADC BX,DX
```

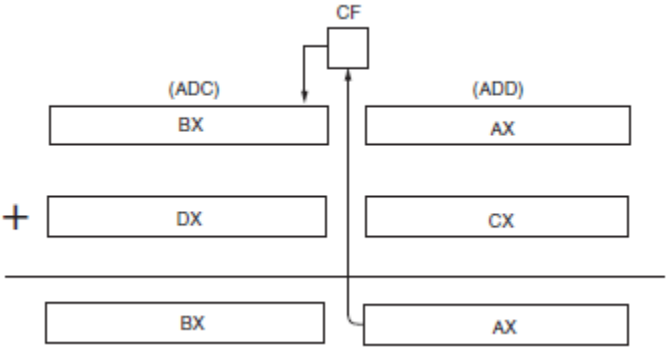


Figure 5-1: ADC showing how the carry flag(C) links the two 16-bit additions into one 32-bit addition

Table 5-3: Example add-with-carry instructions

Assembly Language	*Operation*
ADC AL,AH	AL = AL + AH + carry
ADC CX,BX	CX = CX + BX + carry
ADC EBX,EDX	EBX = EBX + EDX + carry
ADC RBX,0	RBX = RBX + 0 + carry (64-bit mode)
ADC DH,[BX]	The byte contents of the data segment memory location addressed by BX add to DH with the sum stored in DH
ADC BX,[BP+2]	The word contents of the stack segment memory location addressed by BP plus 2 add to BX with the sum stored in BX
ADC ECX,[EBX]	The doubleword contents of the data segment memory location addressed by EBX add to ECX with the sum stored in ECX

XADD(Exchange & Add)

- This instruction is used in 80486-Core2 microprocessors.
- This adds the source to the destination and stores the sum in the destination.
- The difference is that after the addition takes place, the original value of the destination is copied into the source-operand. For example, if BL=12H and DL=02H,

XADD BL,DL instruction executes,

BL register contains the sum 14H and DL becomes 12H

SUB(Subtraction)

- Only types of subtraction not allowed are memory-to-memory and segment register subtractions.
- This affects the flag bits (Table 5-4).

Table 5-4: Example subtraction instructions

Assembly Language	*Operation*
SUB CL,BL	CL = CL – BL
SUB AX,SP	AX = AX – SP
SUB ECX,EBP	ECX = ECX – EBP
SUB RDX,R8	RDX = RDX – R8 (64-bit mode)
SUB DH,6FH	DH = DH – 6FH
SUB AX,0CCCCH	AX = AX – 0CCCCH
SUB ESI,2000300H	ESI = ESI – 2000300H
SUB [DI],CH	Subtracts CH from the byte contents of the data segment memory addressed by DI and stores the difference in the same memory location
SUB CH,[BP]	Subtracts the byte contents of the stack segment memory location addressed by BP from CH and stores the difference in CH
SUB AH,TEMP	Subtracts the byte contents of memory location TEMP from AH and stores the difference in AH
SUB DI,TEMP[ESI]	Subtracts the word contents of the data segment memory location addressed by TEMP plus ESI from DI and stores the difference in DI
SUB ECX,DATA1	Subtracts the doubleword contents of memory location DATA1 from ECX and stores the difference in ECX
SUB RCX,16	RCX = RCX – 18 (64-bit mode)

Register Subtraction

Example 5-9:To subtract the 16-bit contents of registers CX and DX from the contents of register BX

```
SUB BX,CX
SUB BX,DX
```

Immediate Subtraction

Example 5-10:To subtract 44H from 22H

```
MOV CH,22H
SUB CH,44H
```

After the subtraction, the difference(0DEH) moves into the CH register. The flags change as follows for this subtraction:

Z=0(result not zero)
C=1(borrow)
A=1(half borrow)
S=1(result negative)
P=1(even parity)
O=0(no overflow)

DEC(Decrement Subtraction)

- This subtracts 1 from a register or the contents of a memory-location (Table 5-5).
- The decrement indirect memory-data instructions require BYTE PTR or WORD PTR because the assembler cannot distinguish a byte from a word when an index-register addresses memory.
- For example, DEC [SI] is unclear because the assembler cannot determine whether the location addressed by SI is a byte or word.
- Using DEC BYTE PTR[SI],DEC WORD PTR[DI] tells the size of the data to the assembler.

Table 5-5: Example decrement instructions

Assembly Language	*Operation*
DEC BH	BH = BH – 1
DEC CX	CX = CX – 1
DEC EDX	EDX = EDX – 1
DEC R14	R14 = R14 – 1 (64-bit mode)
DEC BYTE PTR[DI]	Subtracts 1 from the byte contents of the data segment memory location addressed by DI
DEC WORD PTR[BP]	Subtracts 1 from the word contents of the stack segment memory location addressed by BP
DEC DWORD PTR[EBX]	Subtracts 1 from the doubleword contents of the data segment memory location addressed by EBX
DEC QWORD PTR[RSI]	Subtracts 1 from the quadword contents of the memory location addressed by RSI (64-bit mode)
DEC NUMB	Subtracts 1 from the contents of data segment memory location NUMB

Subtraction-with-Borrow(SBB)

- This functions as a regular subtraction except that the carry flag(which holds the borrow) also subtracts from the difference (Table 5-6).

Example 5-11: Program to subtract BX-AX from SI-DI (Figure 5-2)

```
SUB AX,DI
SBB BX,SI
```

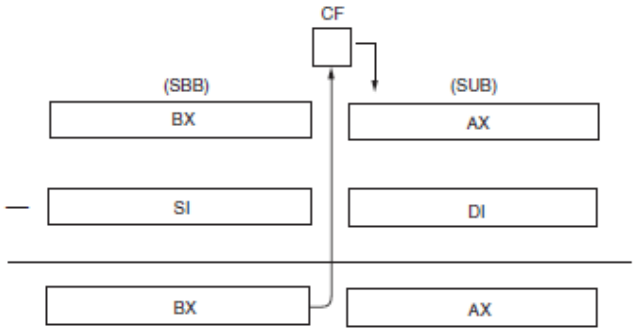


Figure 5-2: Subtraction-with-borrow showing how the carry flag propagates the borrow

Table 5-6: Example subtraction-with-borrow instructions

Assembly Language	*Operation*
SBB AH,AL	AH = AH – AL – carry
SBB AX,BX	AX = AX – BX – carry
SBB EAX,ECX	EAX = EAX – ECX – carry
SBB CL,2	CL = CL – 2 – carry
SBB RBP,8	RBP = RBP– 2 – carry (64-bit mode)
SBB BYTE PTR[DI],3	Both 3 and carry subtract from the data segment memory location addressed by DI
SBB [DI],AL	Both AL and carry subtract from the data segment memory location addressed by DI
SBB DI,[BP+2]	Both carry and the word contents of the stack segment memory location addressed by BP plus 2 subtract from DI
SBB AL,[EBX+ECX]	Both carry and the byte contents of the data segment memory location addressed by EBX plus ECX subtract from AL

CMP(Comparison)

- This is a subtraction that changes only the flag bits,
  destination-operand never changes (Table 5-7).
- This is normally followed by a conditional jump instruction (which tests condition of the flag-bits)
- Only types of comparison not allowed are memory-to-memory and segment register comparisons

Example 5-12:

```
CMP AL,10H      ;compare AL against 10H
JAE SUPER       ;if AL is 10H or above
```

Table 5-7: Example comparison instructions

Assembly Language	*Operation*
CMP CL,BL	CL – BL
CMP AX,SP	AX – SP
CMP EBP,ESI	EBP – ESI
CMP RDI,RSI	RDI – RSI (64-bit mode)
CMP AX,2000H	AX – 2000H
CMP R10W,12H	R10 (word portion) – 12H (64-bit mode)
CMP [DI],CH	CH subtracts from the byte contents of the data segment memory location addressed by DI
CMP CL,[BP]	The byte contents of the stack segment memory location addressed by BP subtracts from CL
CMP AH,TEMP	The byte contents of data segment memory location TEMP subtracts from AH
CMP DI,TEMP[BX]	The word contents of the data segment memory location addressed by TEMP plus BX subtracts from DI
CMP AL,[EDI+ESI]	The byte contents of the data segment memory location addressed by EDI plus ESI subtracts from AL

CMPXCHG(Compare & Exchange)

- This is used only in 80486-Core2 microprocessor.
- This compares the destination-operand with the accumulator(AX).
- If they are equal, the source-operand is copied into the destination;
  if they are not equal, the destination-operand is copied into the accumulator.
- For example, *CMPXCHG CX, DX;* this instruction first compares the contents of CX with AX.
  If CX=AX, DX is copied into AX;
  otherwise CX is copied into AX.

8-bit Multiplication(MUL)

- The multiplicand is always in the AL register.
  The multiplier can be any 8-bit register or any memory location (Table 5-8).
- The product after a multiplication is always a double-width product(AX).
- Immediate multiplication is not allowed.
- This contains one operand because
  it always multiplies the operand times the contents of register AL.
  For example, *MUL BL* ;this instruction multiplies the unsigned-contents of AL by the unsigned-contents of BL.
- For signed multiplication(IMUL),
  the product is in binary-form, if positive &
  in 2's complement forms if negative.

Example 5-13:Program to compute DX=BL*CL

```
MOV BL,5        ;load data
MOV CL,10
MOV AL,CL       ;position data
MUL BL          ;multiply
MOV DX,AX       ;position product
```

Table 5-8: Example 8-bit multiplication instructions

Assembly Language	*Operation*
MUL CL	AL is multiplied by CL; the unsigned product is in AX
IMUL DH	AL is multiplied by DH; the signed product is in AX
IMUL BYTE PTR[BX]	AL is multiplied by the byte contents of the data segment memory location addressed by BX; the signed product is in AX
MUL TEMP	AL is multiplied by the byte contents of data segment memory location TEMP; the unsigned product is in AX

16-bit Multiplication

- AX contains the multiplicand while any 16-bit general purpose register contains the multiplier.
- The 32-bit product is stored in DX-AX.
  DX always contains the most significant 16-bits of the product, and
  AX contains the least significant 16 bits.

A Special Immediate 16-bit Multiplication

- This instruction is available in 80286-Core2 microprocessors (Table 5-9).
- This contains 3 operands.
  → First operand is 16-bit destination-register;
  → Second operand is a register or memory-location that contains the 16-bit multiplicand &
  → Third operand is either 8-bit or 16-bit immediate-data used as the multiplier
- Example: IMUL CX,DX,12H ;this instruction multiplies 12H times DX and leaves a 16-bit signed product in CX.

Table 5-9: Example 16-bit multiplication instructions

Assembly Language	*Operation*
MUL CX	AX is multiplied by CX; the unsigned product is in DX–AX
IMUL DI	AX is multiplied by DI; the signed product is in DX–AX
MUL WORD PTR[SI]	AX is multiplied by the word contents of the data segment memory location addressed by SI; the unsigned product is in DX–AX

8-bit Division(DIV)

• AX register is used to store the dividend that is divided by the contents of any 8-bit register or memory-location (Table 5-12).

• After division, quotient appears in AL and
remainder appears in AH.

• For a signed division, quotient is positive or negative;
the remainder always assumes the sign of the dividend and is always an integer.
For example, if AX=0010(+16) and BL=0FDH(-3) and
IDIV BL instruction executes AX=01FBH (This represents a quotient of -5(AL) with a remainder of 1(AH))

Table 5-12: Example 8-bit division instructions

Assembly Language	*Operation*
DIV CL	AX is divided by CL; the unsigned quotient is in AL and the unsigned remainder is in AH
IDIV BL	AX is divided by BL; the signed quotient is in AL and the signed remainder is in AH
DIV BYTE PTR[BP]	AX is divided by the byte contents of the stack segment memory location addressed by BP; the unsigned quotient is in AL and the unsigned remainder is in AH

Example 5-14: Program to divide the unsigned byte contents of memory location NUMB by the unsigned contents of memory location NUMB1.

```
MOV AL,NUMB     ;get NUMB
MOV AH,0        ;zero extend
DIV NUMB1       ;divide by NUMB1
MOV ANSQ,AL     ;save quotient
MOV ANSR,AH     ;save remainder
```

16-Bit Division

• DX-AX register is used to store the dividend that is divided by the contents of any 16-bit register or memory-location (Table 5-13).

• After division, the quotient appears in AX and the remainder appears in DX.

Table 5-13: Example 16-bit division instructions

Assembly Language	*Operation*
DIV CX	DX–AX is divided by CX; the unsigned quotient is AX and the unsigned remainder is in DX
IDIV SI	DX–AX is divided by SI; the signed quotient is in AX and the signed remainder is in DX
DIV NUMB	DX–AX is divided by the word contents of data segment memory NUMB; the unsigned quotient is in AX and the unsigned remainder is in DX

UNIT 4: ARITHMETIC AND LOGIC INSTRUCTIONS(CONT.)

BCD Arithmetic

- Two arithmetic operation on BCD data are: addition and subtraction.
- The instruction-set provides 2 instructions that correct the result of a BCD addition and a BCD subtraction.
  1) DAA(decimal adjust after addition) instruction follows BCD addition.
  2) DAS(decimal adjust after subtraction) follows BCD subtraction.
  (Both instructions correct result of addition or subtraction so that it is a BCD number)
- For BCD data, the numbers always appear in the packed BCD form and are stored as 2 BCD digits per byte.
- These instructions use only AL as the source and as the destination.

DAA Instruction

- This follows the ADD or ADC instruction to adjust the result into a BCD result.

Example 5-18:Program to add the BCD numbers in DX and BX, and store the result in CX

```
MOV DX,1234H      ;load 1234 BCD
MOV BX,3099H      ;load 3099 BCD
MOV AL,BL         ;sum BL and DL
ADD AL,DL
DAA
MOV CL,AL         ;answer to CL
MOV AL,BH         ;sum BH,DH an carry
ADC AL,DH
DAA
MOV CH,AL         ;answer to CH
```

DAS Instruction

- This follows the SUB or SBB instruction to adjust the result into a BCD result.

Example 5-18:Program to subtract DX from BX, and store the result in CX

```
MOV DX,1234H      ;load 1234 BCD
MOV BX,3099H      ;load 3099 BCD
MOV AL,BL         ;subtract DL from BL
SUB AL,DL
DAS
MOV CL,AL         ;answer to CL
MOV AL,BH         ;subtract DH and carry
SBB AL,DH
DAS
MOV CH,AL         ;answer to CH
```

ASCII Arithmetic

- The ASCII arithmetic instructions function with ASCII coded-numbers.
- These numbers range in value from 30H to 39H for the numbers 0-9.
- There are 4 instructions used with ASCII arithmetic operations:
  - → AAA (ASCII adjust after addition)
  - → AAD (ASCII adjust before division)
  - → AAM (ASCII adjust after multiplication) &
  - → AAS (ASCII adjust after subtraction)
- These instructions use only AX as the source and as the destination.

AAA Instruction

- If 31H and 39H are added, the result is 6AH. This ASCII addition(1+9) should produce a two-digit ASCII result equivalent to a 10 decimal, which is a 31H and a 30H in ASCII code.
- If the AAA instruction is executed after this addition, the AX register will contain a 0100H. (Although this is not ASCII code, it can be converted to ASCII code by adding 3030H to AX which generates 3130H)
- AAA instruction clears AH if result is less than 10, and adds 1 to AH if the result is greater than 10.

Example 5-20: Program to illustrate ASCII addition

```
MOV AX,31H        ;load ASCII 1
ADD AL,39H        ;add ASCII 9
AAA               ;adjust sum
ADD AX,3030H      ;answer to ASCII
```

AAD Instruction

- This appears before a division.
- This requires that the AX register contain a two-digit unpacked BCD number before executing.
- After adjusting the AX register with AAD, it is divided by an unpacked BCD number to generate a single-digit result in AL & any remainder in AH.

Example 5-21: Program to divide 72 in unpacked BCD by 9 to produce a quotient of 8.

```
MOV BL,9          ;load divisor
MOV AL,0702H      ;load dividend
AAD               ;adjust
DIV BL            ;divide
```

AAM Instruction

- This follows the multiplication instruction after multiplying 2 one-digit packed BCD numbers.

Example 5-22: Program to multiply 5 times 3

```
MOV AL,5      ;load multiplicand
MOV CL,3      ;load multiplier
MUL CL
AAM           ;adjust
```

AAS Instruction

- This adjusts the AX register after an ASCII subtraction. For example, if 38H is subtracted from 37H, then AL will equal 09H and the number in AH will decrement by 1.

Basic Logic Instructions

AND

- This performs logical multiplication (Figure 5-3).
- The AND operation clears bits of a binary number. The task of clearing a bit in a binary-number is called *masking* (Figure 5-4).
- This uses any addressing-mode except memory-to-memory and segment register addressing.

A	B	T
0	0	0
0	1	0
1	0	0
1	1	1

(a)

(b)

Figure 5-3:a)The truth table for the AND operation & b)the logic symbol of an AND gate

```
  x x x x  x x x x   Unknown number
• 0 0 0 0  1 1 1 1   Mask
-----------------
  0 0 0 0  x x x x   Result
```

Figure 5-4: The operation of AND function showing how bits of a number are cleared to zero

Table 5-16: Example AND instructions

Assembly Language	*Operation*
AND AL,BL	AL = AL and BL
AND CX,DX	CX = CX and DX
AND ECX,EDI	ECX = ECX and EDI
AND RDX,RBP	RDX = RDX and RBP (64-bit mode)
AND CL,33H	CL = CL and 33H
AND DI,4FFFH	DI = DI and 4FFFH
AND ESI,34H	ESI = ESI and 34H
AND RAX,1	RAX = RAX and 1 (64-bit mode)
AND AX,[DI]	The word contents of the data segment memory location addressed by DI are ANDed with AX
AND ARRAY[SI],AL	The byte contents of the data segment memory location addressed by ARRAY plus SI are ANDed with AL
AND [EAX],CL	CL is ANDed with the byte contents of the data segment memory location addressed by ECX

Example 5-25: Program to convert the ASCII contents of BX into BCD

```
MOV BX,3135H     ;load ASCII
AND BX,0F0FH     ;mask BX
```

OR

- This performs logical addition (Figure 5-5).
- This uses any addressing-mode except memory-to-memory and segment register addressing.

A	B	T
0	0	0
0	1	1
1	0	1
1	1	1

(a)

(b)

Figure 5-5:a)The truth table for the OR operation & b)the logic symbol of an OR gate

```
  x x x x  x x x x   Unknown number
+ 0 0 0 0  1 1 1 1   Mask
-----------------
  x x x x  1 1 1 1   Result
```

Figure 5-6: The operation of the OR function showing how bits of a number are set to one

Table 5-17: Example OR instructions

Assembly Language	*Operation*
OR AH,BL	AL = AL or BL
OR SI,DX	SI = SI or DX
OR EAX,EBX	EAX = EAX or EBX
OR R9,R10	R9 = R9 or R10 (64-bit mode)
OR DH,0A3H	DH = DH or 0A3H
OR SP,990DH	SP = SP or 990DH
OR EBP,10	EBP = EBP or 10
OR RBP,1000H	RBP = RBP or 1000H (64-bit mode)
OR DX,[BX]	DX is ORed with the word contents of data segment memory location addressed by BX
OR DATES[DI + 2],AL	The byte contents of the data segment memory location addressed by DI plus 2 are ORed with AL

NOT

• NOT inverts all bits of a byte or word (Table 5-21).

• NEG 2'complements a number, which means that the arithmetic-sign of a signed-number changes from positive to negative or from negative to positive.

• NOT is considered logical &
NEG is considered an arithmetic operation.

• Both can use any addressing-mode except segment register addressing.

Table 5-21: Example NOT and NEG instructions

Assembly Language	*Operation*
NOT CH	CH is one's complemented
NEG CH	CH is two's complemented
NEG AX	AX is two's complemented
NOT EBX	EBX is one's complemented
NEG ECX	ECX is two's complemented
NOT RAX	RAX is one's complemented (64-bit mode)
NOT TEMP	The contents of data segment memory location TEMP is one's complemented
NOT BYTE PTR[BX]	The byte contents of the data segment memory location addressed by BX are one's complemented

Exclusive OR

• This differs from inclusive OR. The difference is that a 1,1 condition of OR function produces a 1;
the 1,1 condition of the exclusive OR operation produces a 0(Figure 5-7).

• This is sometimes called a *comparator.*

• This uses any addressing-mode except segment register addressing (Figure 5-8).

• Common use: to clear a register to zero. For example, XOR CH,CH instruction clears register CH to 00H and requires 2 bytes of memory to store the instruction. (Likewise, MOV CH,00H instruction also clears CH to 00H but requires 3 bytes of memory).

A	B	T
0	0	0
0	1	1
1	0	1
1	1	0

(a)

(b)

Figure 5-7:a)The truth table for the XOR operation & b)the logic symbol of an XOR gate

```
  x x x x  x x x x   Unknown number
⊕ 0 0 0 0  1 1 1 1   Mask
-------------------
  x x x x  x̄ x̄ x̄ x̄   Result
```

Figure 5-8:The operation of XOR function showing how bits of a number are inverted

Table 5-18:Example XOR instructions

Assembly Language	*Operation*
XOR CH,DL	CH = CH xor DL
XOR SI,BX	SI = SI xor BX
XOR EBX,EDI	EBX = EBX xor EDI
XOR RAX,RBX	RAX = RAX xor RBX (64-bit mode)
XOR AH,0EEH	AH = AH xor 0EEH
XOR DI,00DDH	DI = DI xor 00DDH
XOR ESI,100	ESI = ESI xor 100
XOR R12,20	R12 = R12 xor 20 (64-bit mode)
XOR DX,[SI]	DX is Exclusive-ORed with the word contents of the data segment memory location addressed by SI
XOR DEAL[BP+2],AH	AH is Exclusive-ORed with the byte contents of the stack segment memory location addressed by BP plus 2

Test

• This performs the AND operation. The difference is that the AND instruction changes the destination operand, whereas the TEST instruction does not (Table 5-19).

• This only affects the condition of the flag-register, which indicates the result of the test.

Example 5-28:Program to test the rightmost and leftmost bit positions of the AL register

```
TEST AL,1        ;test right bit
JNZ RIGHT        ;if set
TEST AL,128      ;test left bit
JNZ LEFT         ;if set
```

Table 5-19: Example TEST instructions

Assembly Language	*Operation*
TEST DL,DH	DL is ANDed with DH
TEST CX,BX	CX is ANDed with BX
TEST EDX,ECX	EDX is ANDed with ECX
TEST RDX,R15	RDX is ANDed with R15 (64-bit mode)
TEST AH,4	AH is ANDed with 4
TEST EAX,256	EAX is ANDed with 256

Bit Test

• This functions in the same manner as a CMP.
The difference is that the Bit Test normally tests a single bit
whereas the CMP tests the entire byte or word (Table 5-20).

• Z=1 if the bit under test is a zero &
Z=0 if the bit under test is not zero.

• Usually, this is followed by either the JZ(jump if zero) or JNZ(jump if not zero).

• The destination-operand is normally tested against immediate-data.

• The value of immediate-data is 1 to test the rightmost bit position, 2 to test the next bit, 4 for the next, and so on.

Example 5-29:

```
BTS CX,9        ;set bit 9
BTR CX,0        ;clear bit 0
BTC CX,12       ;complement bit 12
```

Table 5-20: Bit test instructions

Assembly Language	*Operation*
BT	Tests a bit in the destination operand specified by the source operand
BTC	Tests and complements a bit in the destination operand specified by the source operand
BTR	Tests and resets a bit in the destination operand specified by the source operand
BTS	Tests and sets a bit in the destination operand specified by the source operand

Shift

- This move numbers to the left or right within a register or memory-location (Figure 5-9).
- They also perform simple arithmetic such as
  → multiplication by powers of 2^{+n} (left side) &
  → division by powers of 2^{-n} (right shift).
- There are 4 different shift instructions: Two are logical-shifts and two are arithmetic-shifts.
- In a logical left-shift, a 0 is moved into the rightmost bit-position.
  In a logical right-shift, a 0 is moved into the leftmost bit-position.
- The arithmetic shift-left and logical left-shift are identical.
  The arithmetic right-shift and logical right-shift are different because
  the arithmetic right-shift copies the sign-bit through the number,
  whereas the logical right-shift copies a 0 through the number.
- A shift count can be
  → immediate or
  → located in register CL
- Logical shifts function with unsigned numbers &
  arithmetic shifts function with signed numbers.
- Logical shifts multiply or divide unsigned data &
  arithmetic shifts multiply or divide signed data.

(A shift left always multiplies by 2 for each bit position shifted, and a shift right always divides by 2 for each bit position shifted. Shifting a number 2 places, to left or right, multiplies or divides by 4).

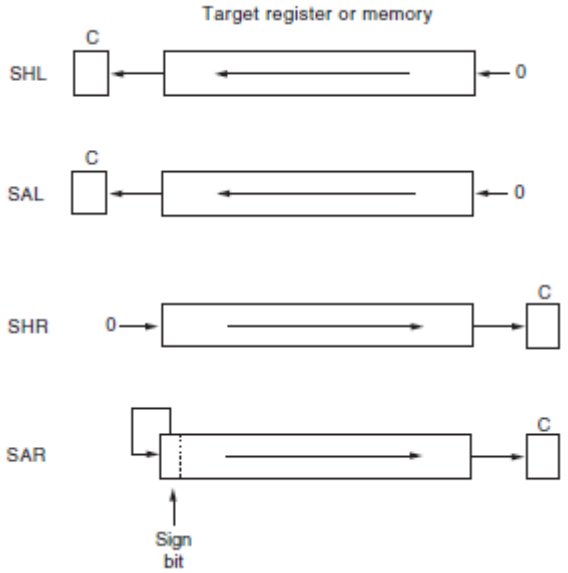


Figure 5-9: The shift instructions showing the operation and direction of the shift

Table 5-22: Example shift instructions

Assembly Language	*Operation*
SHL AX,1	AX is logically shifted left 1 place
SHR BX,12	BX is logically shifted right 12 places
SHR ECX,10	ECX is logically shifted right 10 places
SHL RAX,50	RAX is logically shifted left 50 places (64-bit mode)
SAL DATA1,CL	The contents of data segment memory location DATA1 are arithmetically shifted left the number of spaces specified by CL
SHR RAX,CL	RAX is logically shifted right the number of spaces specified by CL (64-bit mode)
SAR SI,2	SI is arithmetically shifted right 2 places
SAR EDX,14	EDX is arithmetically shifted right 14 places

Example 5-31: Program to multiply AX by 10(1010B)

```
SHL AX,1          ;AX times 2
MOV BX,AX
SHL AX,2          ;AX times 8
ADD AX,BX         ;AX times 10
```

Double-Precision Shifts

- SHLD(shift left) and SHRD(shift right) are available in only 80386-Core2 microprocessors.
- Each instruction contains 3 operands (instead of 2 found with the other shift instructions).
- SHRD AX,BX,12 ;this instruction logically shifts AX right by 12 bit positions.
  The rightmost 12 bits of BX shift into the leftmost 12 bits of AX.
  The contents of BX remain unchanged.

Rotate

- This positions binary-data by rotating the information in a register or memory-location, either from one end to another or through the carry-flag (Figure 5-10).
- The programmer can select either a left or a right rotate.
- A rotate count can be
  → immediate or
  → located in register CL.

Example 5-32: Program to shift 48-bit numbers in registers DX, BX and AX left one binary place

```
SHL AX,1
RCL BX,1
RCL DX,1
```

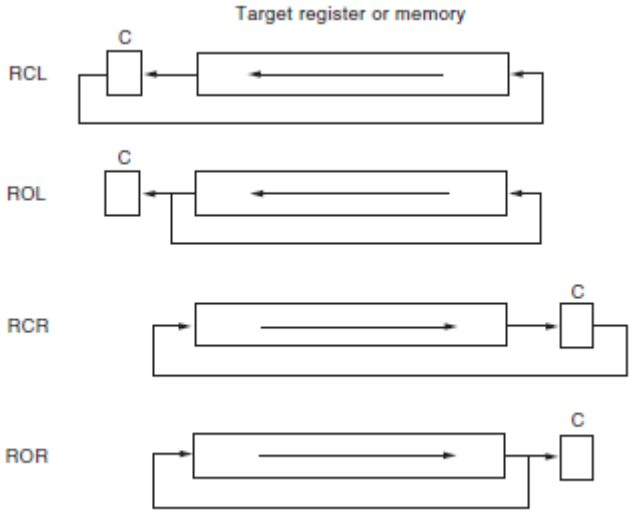


Figure 5-10: The rotate instructions showing the direction and operation of each rotate

Table 5-23: Example rotate instructions

Assembly Language	*Operation*
ROL SI,14	SI rotates left 14 places
RCL BL,6	BL rotates left through carry 6 places
ROL ECX,18	ECX rotates left 18 places
ROL RDX,40	RDX rotates left 40 places
RCR AH,CL	AH rotates right through carry the number of places specified by CL
ROR WORD PTR[BP],2	The word contents of the stack segment memory location addressed by BP rotate right 2 places

Bit Scan Instructions

- BSF(bit scan forward) & BSR(bit scan reverse) are available only in the 80386-Pentium4 processors.
- Both scan through the source-number, searching for the first 1-bit.
- BSF scans the number from the leftmost-bit toward the right &
  BSR scans the number from the rightmost-bit toward the left.
- If a 1-bit is encountered,
  zero-flag is set and the bit position-number of the 1-bit is placed into the destination-operand.
  If no 1-bit is encountered,
  zero flag is cleared.
- For example, if EAX=60000000H and the BSF EBX,EAX instruction executes,
  the number is scanned from the leftmost bit toward the right.
  The first 1-bit encountered is at bit position 30, which is placed into EBX & zero-flag bit is set.

String Comparisons

SCAS(string scan)

- This compares
  - → AL register with a byte block of memory addressed by DI in extra-segment memory or
  - → AX register with a word block of memory addressed by DI in extra-segment memory
- This subtracts memory from AL or AX without affecting either the register or the memory-location.
- The opcode used for byte comparison is SCASB,
  the opcode used for the word comparison is SCASW.
- This uses the direction-flag(D) to select ether auto-increment or auto-decrement operation for DI.

Example 5-33: Program to check whether any location contains 00H in a 100 byte of memory

```
LEA DI,BLOCK      ;address data
CLD               ;auto-increment
MOV CX,100        ;load counter
XOR AL,AL         ;clear AL
REPNE
```

CMPS(compare string)

- This always compares 2 section of memory-data as bytes(CMPSB) or words(CMPSW).
- The contents of the data-segment memory-location addressed by SI are compared with the contents of the extra-segment memory-location addressed by DI.
- This increment or decrement both SI and DI.
- This is normally used with either the REPE or REPNE prefix.

Example 5-35:Program to compare two section of memory searching for a match

```
LEA SI,LINE       ;address LINE
LEA DI,TABLE      ;address TABLE
CLD               ;auto-increment
MOV CX,10         ;load counter
REPE CMPSB        ;search
```

UNIT 6: 8086/8088 HARDWARE SPECIFICATIONS

Power Supply Requirements

- 8086 microprocessor requires +5.0V with a supply voltage tolerance of 10%.
- This uses a maximum supply current of 360mA.
- This operates in ambient temperatures of between 32'F and 180'F.

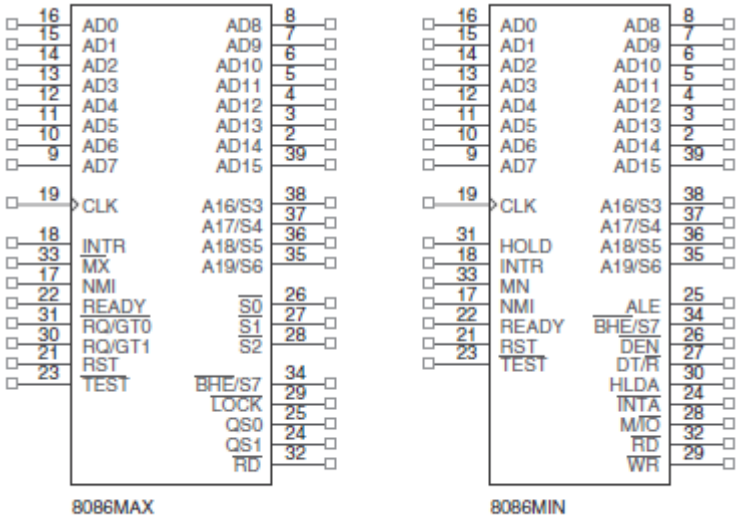


Figure 9-1:a)The pin-out of 8086 in maximum mode; b)The pin-out of 8086 in minimum mode

DC Characteristics

Table 9-1: Input characteristics of the 8086

Logic Level	*Voltage*	*Current*
0	0.8 V maximum	±10 µA maximum
1	2.0 V minimum	±10 µA maximum

Table 9-2: Output characteristics of the 8086

Logic Level	*Voltage*	*Current*
0	0.45V maximum	2.0 mA maximum
1	2.4 V minimum	–400 µA maximum

Table 9-3: Recommended fan-out for 8086 pin connection

Family	*Sink Current*	*Source Current*	*Fan-out*
TTL (74)	–1.6 mA	40 µA	1
TTL (74LS)	–0.4 mA	20 µA	5
TTL (74S)	–2.0 mA	50 µA	1
TTL (74ALS)	–0.1 mA	20 µA	10
TTL (74AS)	–0.5 mA	25 µA	10
TTL (74F)	–0.5 mA	25 µA	10
CMOS (74HC)	– 10 µA	10 µA	10
CMOS (CD)	– 10 µA	10 µA	10
NMOS	– 10 µA	10 µA	10

Pin Connections

AD_{15}-AD_0(Multiplexed Address/Data)

• These lines contain
→ memory-address or port-number if ALE=1 &
→ data if ALE=0.

• These pins are at their high impedance state during a hold acknowledge.

A_{19}/S_6-A_{16}/S_3(Multiplexed Address/Status)

• These lines provide address-signals A_{19}-A_{16} and status-bits S_6-S_3.

• S_6=0 always, S_3=IF(Interrupt Flag), S_4 & S_3 indicates which segment is accessed during the current bus cycle(Table 9-4).

Table 9-4: Function of status bits S_3 and S_4

S_4	S_3	Function
0	0	Extra segment
0	1	Stack segment
1	0	Code or no segment
1	1	Data segment

$\overline{RD}$ (Read)

• If $\overline{RD}$=0, data is read from memory(or I/O device) to data bus.

READY

• If READY=0, microprocessor enters into wait states & remains idle.
If READY=1, microprocessor executes normally.

INTR(Interrupt Request)

• If INTR=1 when IF=1, microprocessors enters an interrupt acknowledge cycle after executing the current instruction.

$\overline{TEST}$

• If TEST=0, WAIT instruction functions as an NOP.
If TEST=1, WAIT instruction waits for TEST to become 0.

NMI(Non Maskable Interrupt)

• NMI is similar to INTR except that it doesn't depend on IF.

• This is non-maskable which means that it cannot be disabled.

• If NMI=1, interrupt vector type-2 occurs. .

RESET

• If RESET=1 for a minimum of 4 clocking-periods, microprocessor resets itself.

CLK(Clock)

• This provides the basic timing-signal to the microprocessor.

• The clock-signal must have a duty-cycle of 33%(high for 1/3 of the clocking period & low for 2/3) to provide proper internal timing for microprocessor.

VC(Power Supply)

• This provides a +5.0V, +10% signal to the microprocessor.

GND(Ground)

• This is the return for the power supply.

MN/$\overline{MX}$ (Minimum/Maximum)

• If MN/$\overline{MX}$ =+5.0V, minimum-mode operation is selected.
If MN/$\overline{MX}$ =GND, maximum-mode operation is selected.

$\overline{BHE}$ /S_7(Bus High Enable/Status)

• BHE is used to enable the most-significant data bus bits(D_{15}-D_8) during a read or a write operation.

• S_7= 1 always.

Minimum Mode Pins

• Minimum mode operation of microprocessor is obtained by connecting the MN/MX pin directly to +5.0 V.

$M/\overline{IO}$ (Memory/IO)

• If M/$\overline{IO}$ =1, address bus contains memory-address.

If M/$\overline{IO}$ =0, address bus contains I/O port-address.

$\overline{WR}$ (Write)

• If $\overline{WR}$ =0, data is written into memory(I/O) from data bus.

$\overline{INTA}$ (Interrupt Acknowledge)

• This is used to put the interrupt-vector number onto the data bus in response to an interrupt-request(INTR).

ALE(Address Latch Enable)

• This shows that multiplexed address/data lines contain address information.

$DT/\overline{R}$ (Data Transmit/Receive)

• This shows that the data-bus is transmitting(DT/$\overline{R}$ =1) or receiving (DT/$\overline{R}$ =0) data.

DEN(Data Bus Enable)

• This activates external data bus buffers.

HOLD

• If HOLD=1, μprocessor stops executing software & places its address, data & control bus at high-impedance state.

If HOLD=0, microprocessor executes software normally.

HLDA(Hold Acknowledge)

• This indicates that the microprocessor has entered the hold state.

$\overline{SS0}$

• This is combined with M/$\overline{IO}$ and DT/$\overline{R}$ to decode the function of the current bus cycle (Table 9-5).

Table 9-5: Bus cycle status using $\overline{SSO}$

IO/$\overline{M}$	DT/$\overline{R}$	$\overline{SS0}$	Function
0	0	0	Interrupt acknowledge
0	0	1	Memory read
0	1	0	Memory write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	I/O read
1	1	0	I/O write
1	1	1	Passive

Maximum Mode Pins

• Maximum mode operation of microprocessor is obtained by connecting MN/MX pin to GND.

$\overline{S2}$, $\overline{S1}$, and $\overline{S0}$ (Status bits)

• These bits indicate the function of the current bus cycle.

Table 9-6: Bus control function generated by the bus controller(8288)

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Function
0	0	0	Interrupt acknowledge
0	0	1	I/O read
0	1	0	I/O write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Passive

$\overline{RQ}$ /$\overline{GT1}$ and $\overline{RQ}$/$\overline{GT0}$ (Request/Grant)

• This is used to request DMA(direct memory access) operation.

$\overline{LOCK}$

• This is used to lock peripherals off the system.

QS_1 and QS_0(Queue Status)

•This shows the status of the internal instruction queue(Table 9-7).

Table 9-7: Queue status bits

QS_1	QS_0	Function
0	0	Queue is idle
0	1	First byte of opcode
1	0	Queue is empty
1	1	Subsequent byte of opcode

Pin Functions of Clock Generator (8284A)

- This provides the following basic functions or signals:
  1) Clock generation
  2) RESET synchronization
  3) READY synchronization and
  4) TTL-level peripheral clock signal.
- 82844A is an 18-pin IC designed specifically for use with the 8086 microprocessor

CLK(Clock)

- This provides clock signal to microprocessor & other components in the system.
- This has an output signal that is 1/3 of the crystal or EFI input frequency and has a 33% duty cycle.

PCLK(Peripheral Clock)

- This provides clock signal to peripheral-equipment in the system.
- This has an output signal that is 1/6 of the crystal or EFI input frequency and has a 50% duty cycle.

READY

- This is connected to READY input of microprocessor. This is synchronized with the RDY_1 and RDY_2 inputs.

RESET

- This is connected to the RESET input of microprocessor.

X_1 & X_2(Crystal Oscillator)

- This is used as timing source for the clock generator and all its functions. This is connected to an external crystal.

$F/\overline{C}$ (Frequency/Crystal)

- If $F/\overline{C}$ =1, external clock is provided to the EFI input pin.
  $F/\overline{C}$ =10, internal crystal oscillator provides the timing signal.

RDY_1 & RDY_2(READY)

- These are used to cause wait states in microprocessor.

$\overline{AEN1}$ and $\overline{AEN2}$ (Address Enable)

- These are used to enable the bus ready signals: RDY_1 & RDY_2.

$\overline{ASYNC}$ (Ready Synchronization)

- This selects either one or two stage of synchronization for the RDY_1 & RDY_2 inputs.

CSYNC(Clock Synchronization)

- This is used whenever the EFI input provides synchronization in systems with multiple processors.

OSC(Oscillator)

- This provides an EFI input to other 8284A clock generators in some multiple-processor systems.

GND

- This is connected ground.

VCC

- This is connected to +5.0V with a tolerance of +10%.

$\overline{RES}$ (Reset)

- This is an active low input to the 8284A.

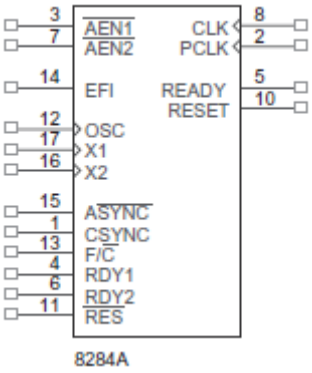


Figure 9-2: Pin out of 8284A Clock Generator

Operation of the 8284A

Operation of the Clock Section

• The crystal oscillator has 2 inputs: X_1 and X_2. If a crystal is attached to X_1 & X_2, the oscillator generates a square-wave signal at the same frequency as the crystal.

• If F/$\overline{C}$ =0, the oscillator output is steered through to the divide-by-3 counter.

If F/$\overline{C}$ =1, then EFI is steered through the divide-by-3 counter.

• The output of the divide-by-3 counter generates the timing for

→ ready synchronization

→ signal for divide-by-2 counter

→ CLK signal to the microprocessor

Operation of the Reset Section

• Reset section consists of a Schmitt trigger buffer & a single D-type flip-flop.

• D-type flip-flop ensures that the timing requirements of microprocessor's RESET input is met.

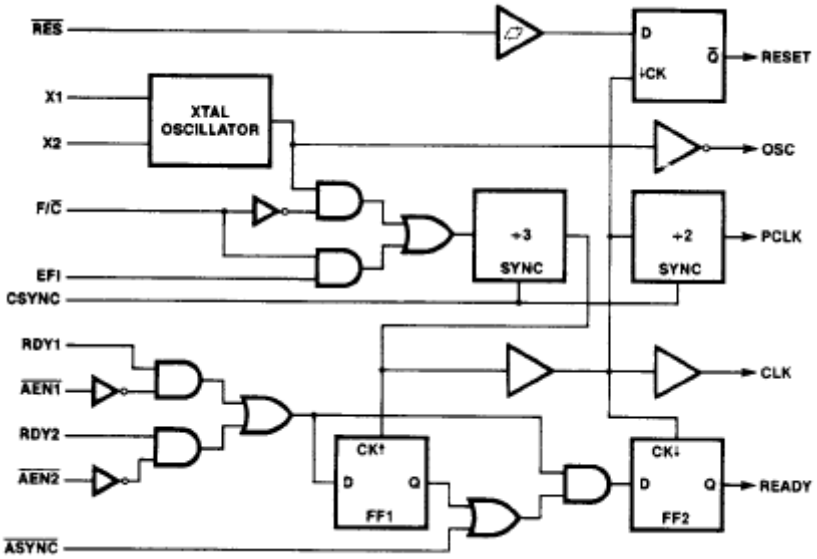


Figure 9-3: The internal block diagram of the 8284A clock generator

Bus Buffering and Latching

Demultiplexing the Buses

- The address/data bus is multiplexed(shared) to reduce the number of pins required for the microprocessor IC.
- All computer systems have 3 buses:
  - i) Address bus provides the memory and I/O with the memory address or the I/O port number
  - ii) Data bus transfers data between the microprocessor and the memory/IO in the system
  - iii) Control bus provides control signals to the memory and I/O.

Demultiplexing the 8088

- The multiplexed pins include AD_7-AD_0, A_{19}/S_6-A_{16}/S_3. All these signals must be demultipleed.
- Two 74LS343 transparent latches are used to demultiplex
  - → address/data lines connections AD_7-AD_0 &
  - → address/status lines connections A_{19}/S_6-A_{16}/S_3
- When ALE=1, these latches pass the inputs to the outputs.
- After a short time, ALE returns to 0, which causes the latches to remember the inputs at the time of the change to logic 0.
- A_7-A_0 are stored in the bottom latch and A_{19}-A_{16} are stored in the top latch.

Demultiplexing the 8086

- The multiplexed pins include AD_{15}-AD_0, A_{19}/S_6-A_{16}/S_3 and $\overline{BHE}/S_7$. All these signals must be demultiplexed.
- Three 74LS343 transparent latches are used to demultiplex
  - → address/data lines connections AD_{15}-AD_0 &
  - → address/status lines connections A_{19}/S_6-A_{16}/S_3 & $\overline{BHE}/S_7$
- When ALE=1, these latches pass the inputs to the outputs.
- After a short time, ALE returns to 0, which causes the latches to remember the inputs at the time of the change to logic 0.
- A_{15}-A_0 are stored in the bottom 2 latches and A_{19}-A_{16} are stored in the top latch.

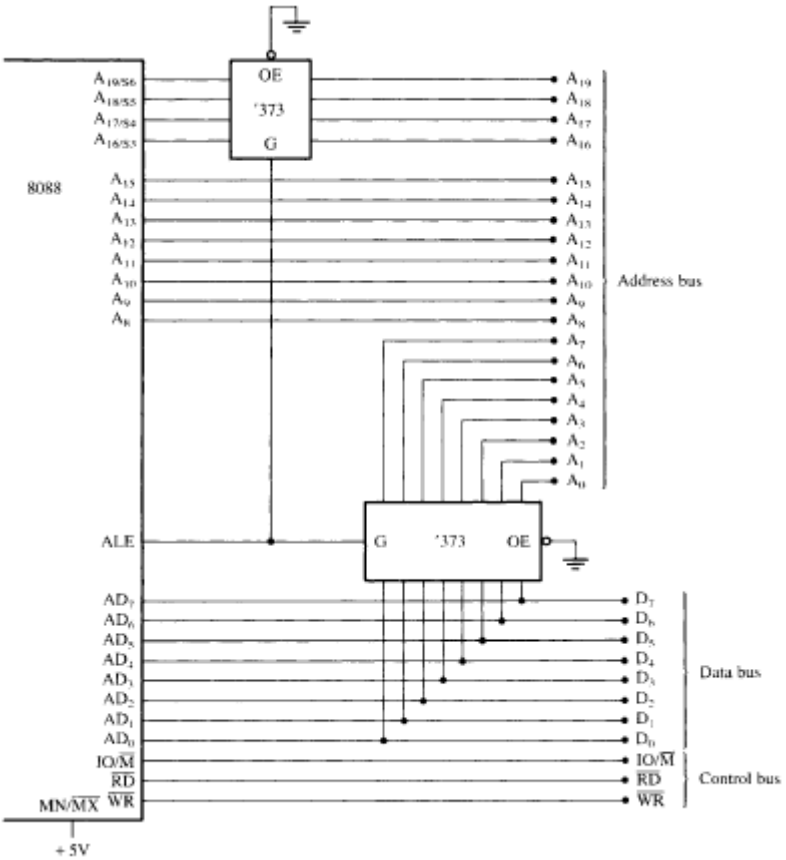


Figure 9-5: The 8086 microprocessor shown with a demultiplexed address bus.

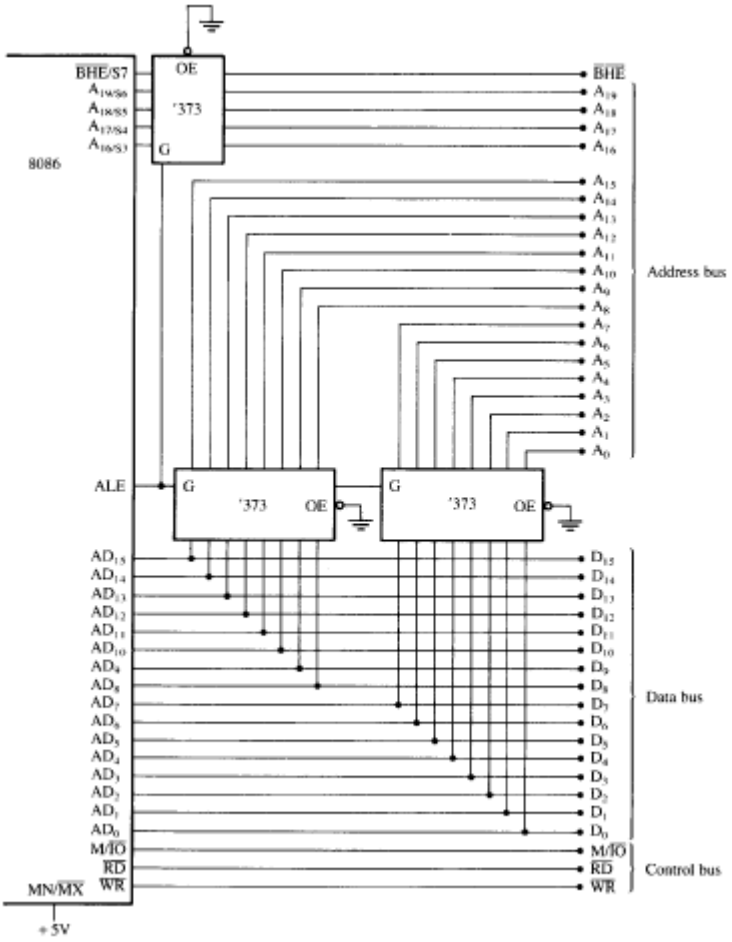


Figure 9-6: The 8086 microprocessor shown with a demultiplexed address bus.

The Buffered System

• If more than 10 unit loads are attached to any bus pins, the entire 8086 system must be buffered.

The Fully Buffered 8088

• The 8 address-pins A_{15}-A_8 use a 74LS244 octal buffer,
→ 8 data-bus pins D_7-D_0 use a 74LS245 octal bidirectional bus-buffer &
→ control-bus signals M/$\overline{IO}$, $\overline{RD}$ and $\overline{WR}$ use a 74LS244 buffer.

• The direction of the 74LS245 is controlled by the DT/$\overline{R}$ signal and is enabled & disabled by $\overline{DEN}$.

The Fully Buffered 8086

• The 16 address pins A_{15}-A_0 use two 74LS244 octal buffers,
→ the 16 data bus pins D_{15}-D_0 use two 74LS245 octal bidirectional bus buffers &
→ the control bus signals M/IO, RD and WR use a 74LS244 buffer.

• The direction of the 74LS245 is controlled by the DT/$\overline{R}$ signal and is enabled & disabled by $\overline{DEN}$.

• 8086 has a $\overline{BHE}$ signal that is buffered for memory bank selection.

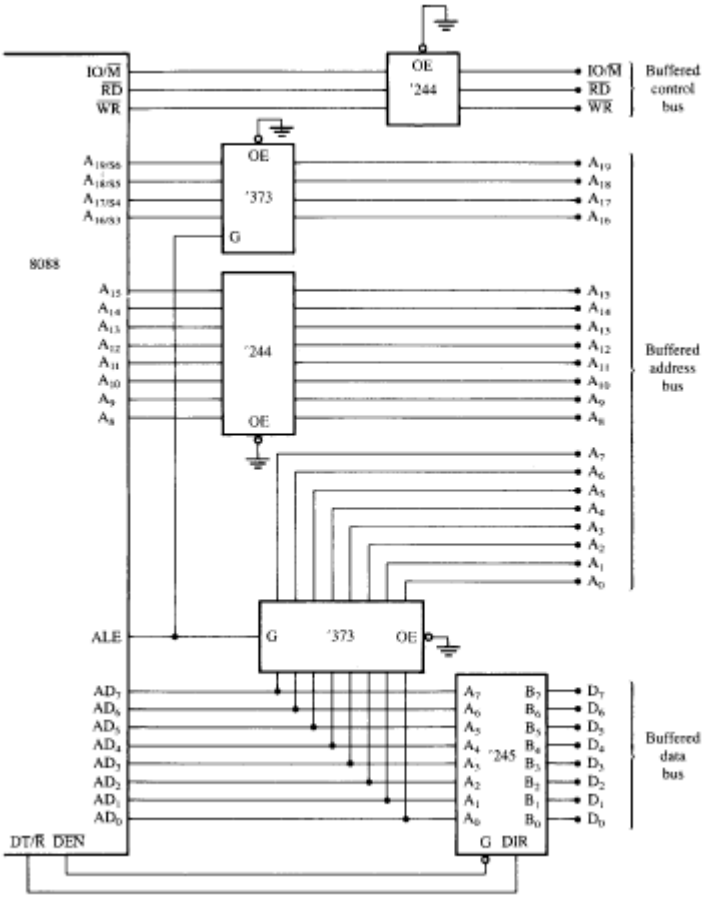


Figure 9-7: Fully buffered 8088 microprocessor

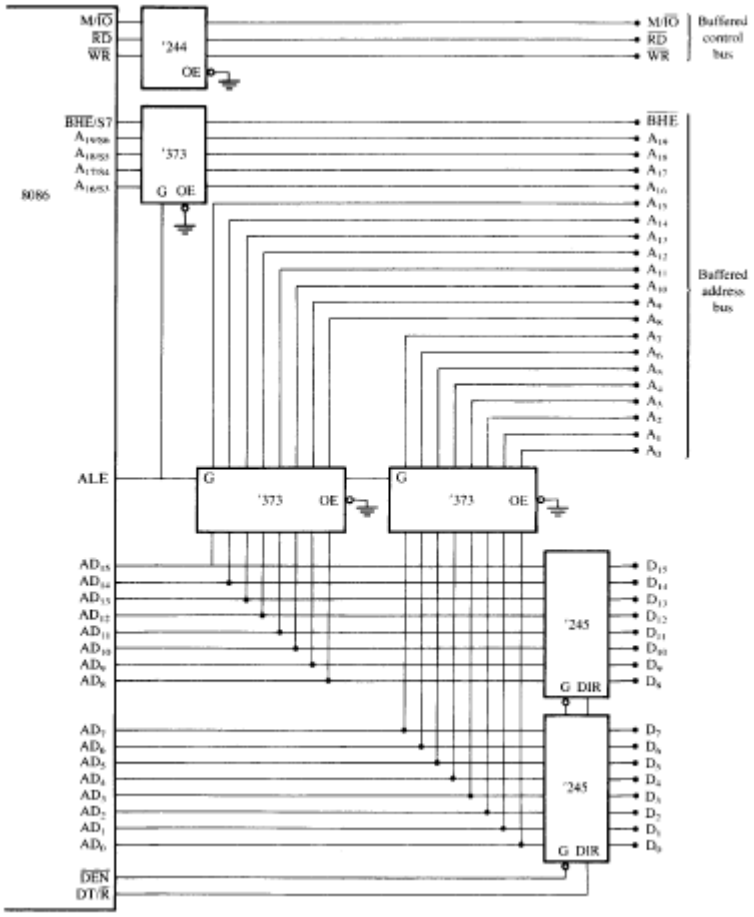


Figure 9-8: Fully buffered 8086 microprocessor

Bus Timing

Basic Bus Operation

- Read operation: The microprocessor
  - → outputs the memory address on address bus,
  - → issues a read memory signal($\overline{RD}$) &
  - → accepts the data via the data-bus.
- Write operation: The microprocessor
  - → outputs the memory address on address line,
  - → outputs the data to be written into memory on the data bus &
  - → issues a write($\overline{WR}$) to memory and sets M/ $\overline{IO}$ =1

Timing in General

- The 8086 microprocessor uses the memory and I/O in periods called bus cycle.
- Each bus cycle equals 4 system clocking periods(T states).
- If clock is operated at 5MHz, one 8086 bus cycle is complete in 800ns.(This means that microprocessor reads or writes data between itself & memory or I/O at maximum rate of 1.25 MIPS).

Steps for Read Operation

- During T1, the address of the memory(or I/O) location is sent out via the address-bus and the address/data bus connections. During same time, control signals ALE, DT/ $\overline{R}$ and M/ $\overline{IO}$ are also issued. (M/ $\overline{IO}$ signal indicates whether address-bus contains memory-address or I/O port-number).
- During T2, microprocessor issues $\overline{RD}$ & DEN. These events cause the memory or I/O device to begin to perform a read operation.
- During T3, data is transferred between the microprocessor and the memory or I/O.
  READY is sampled at the end of T2. If READY=0, T3 becomes a wait state(Tw).
- During T4, all bus signals are deactivated in preparation for the next bus cycle.

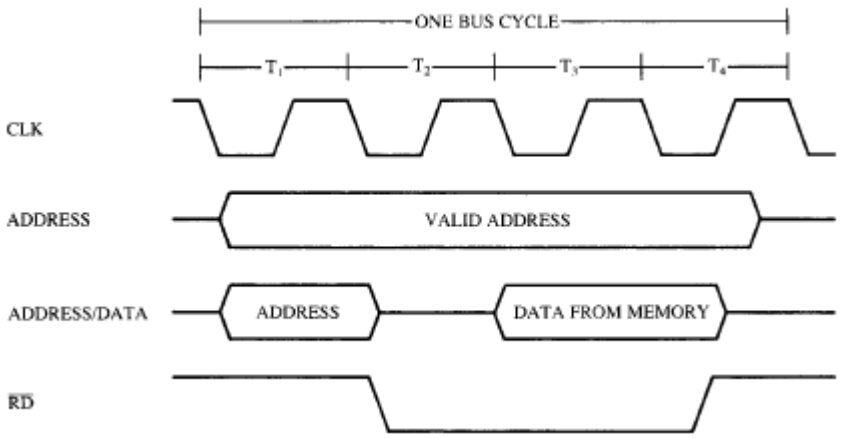


Figure 9-10: Simplified 8086 read bus cycle

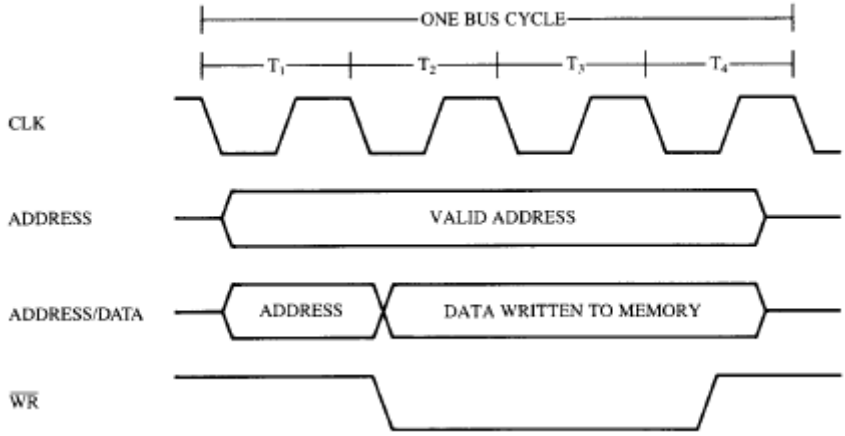


Figure 9-9: Simplified 8086 write bus cycle

Maximum Mode versus Maximum Mode

- Minimum-mode operation is obtained by connecting the mode selection pin MN/ $\overline{MX}$ to +5.0V & maximum mode is selected by grounding this pin.

Minimum Mode Operation

- Minimum mode operation costs less because all the control signals for the memory and I/O are generated by the microprocessor.

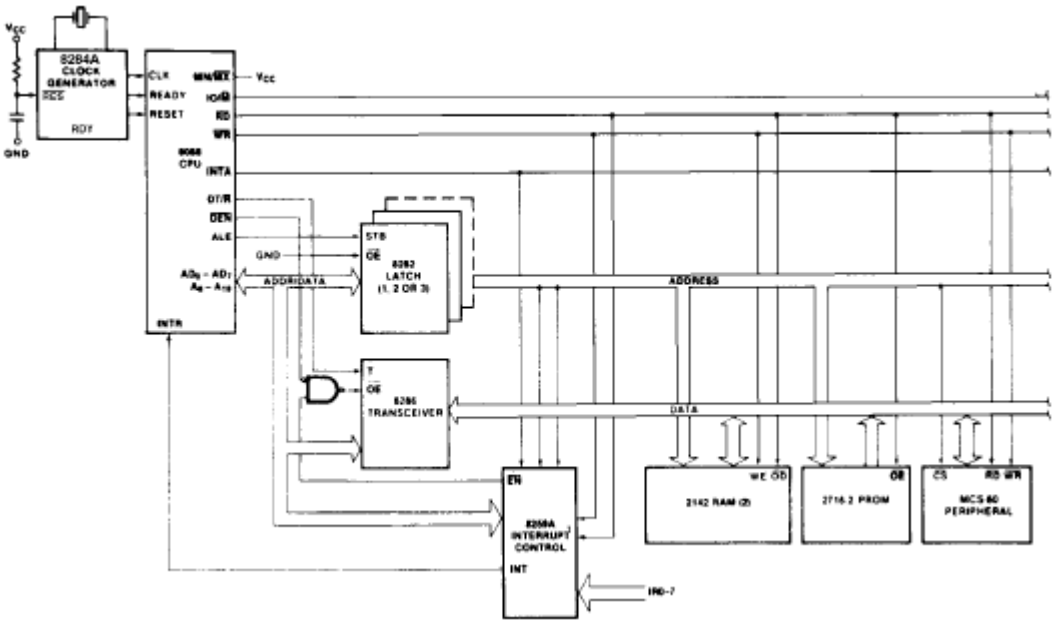


Figure 9-19: Minimum mode 8086 system

Maximum Mode Operation

• In maximum mode, 8288 bus controller must be used to provide the control bus signals to the memory and I/O. This is because maximum mode microprocessor removes some of the system's control signal lines in favor of control signals for the coprocessors.(Maximum mode is used only when the system contains external coprocessors such as the 8087 arithmetic coprocessor).

The 8288 Bus Controller

S2, S1 & S0(Status bits)

• These signals are decoded to generate the timing signals for the system.

CLK

• This provides internal timing and must be connected to CLK output-pin of 8284A clock generator.

ALE

• This is used to demultiplex the address/data bus.

DEN

• This controls the bidirectional data-bus buffers in the system.

$DT/\overline{R}$

• This is used to control the direction of the bidirectional data-bus buffers.

$\overline{AEN}$ (Address Enable)

• This is used to enable the memory control signals.

CEN(Control Enable)

• This is used to enable the command output pins on the 8288.

IOB(I/O Bus Mode)

• This selects either the I/O bus mode or system bus mode operation.

$\overline{AIOWC}$ (Advanced I/O Write Command)

• This is used to provide I/O with an advanced I/O write control signal.

$\overline{IORC}$ (I/O Read Command)

• This provides I/O with its read control signal.

$\overline{IOWC}$ (I/O Write Command)

• This provides I/O with its main write control signal.

$\overline{AMWT}$ (Advanced Memory Write)

• This provides memory with an advanced write signal.

$\overline{MWTC}$ (Memory Write Command)

• This provides memory with its normal write control signal.

$\overline{MRDC}$ (Memory Read Command)

• This provides memory with its normal read control signal.

$\overline{INTA}$

• This acknowledges an interrupt request input applied to the INTR pin.

$MCE/\overline{PDEN}$(Master Cascade/Peripheral Data)

• This selects cascade operating for an interrupt controller if IOB=0;
enables the I/O bus transceivers if IOB=1.

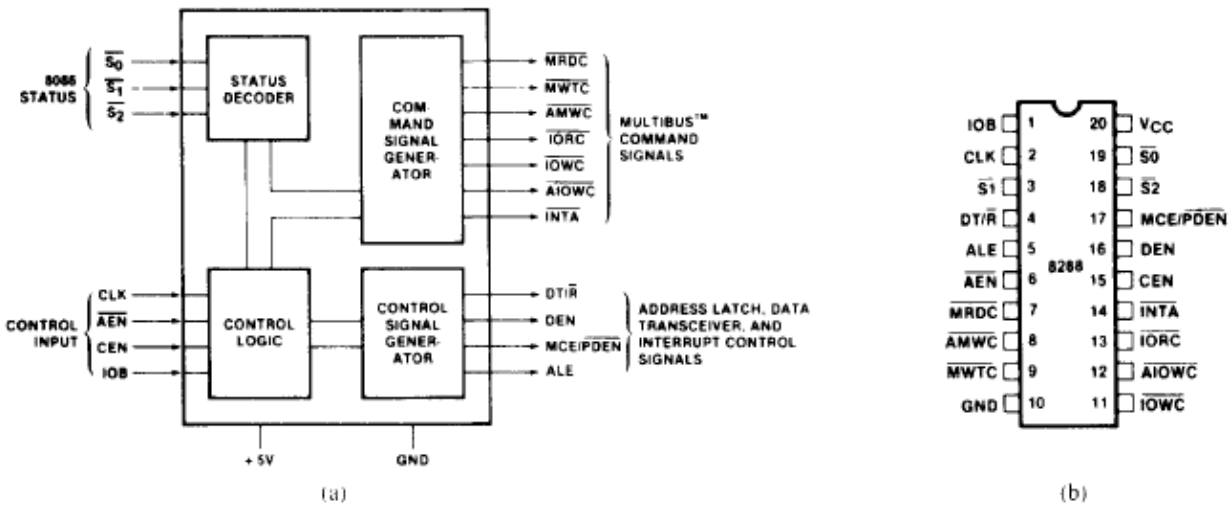


Figure 9-21: The 8288 bus controller a) block diagram b)pin out

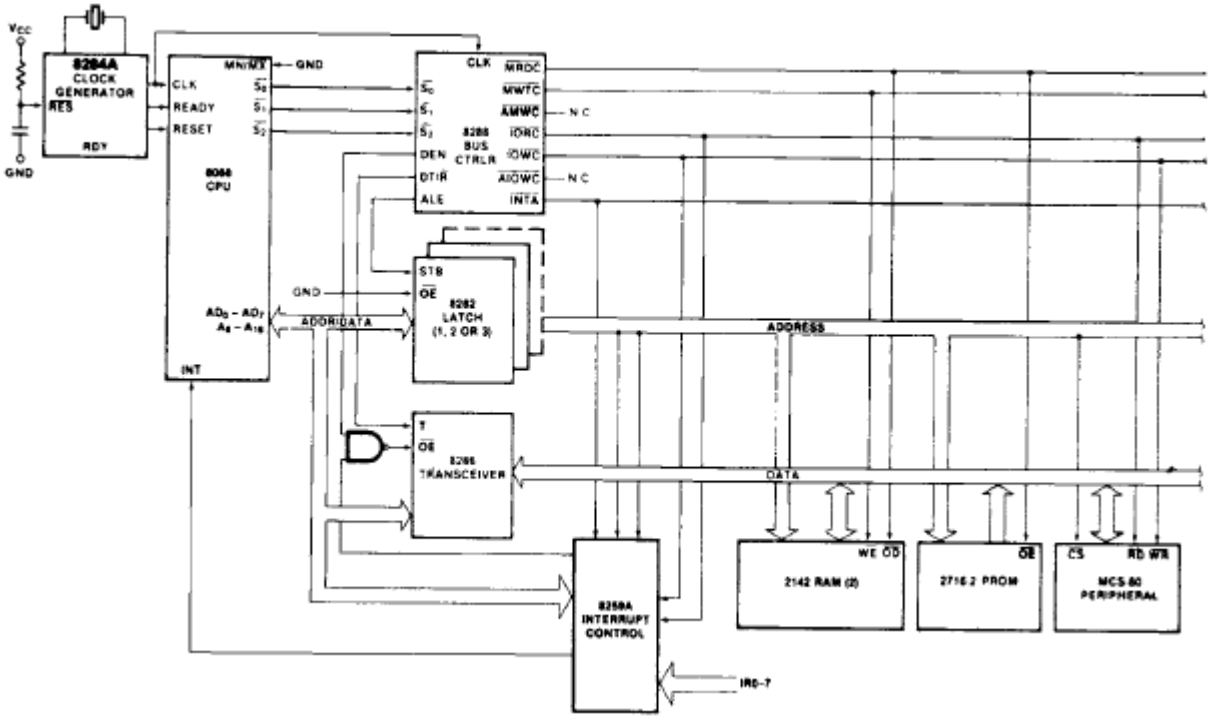


Figure 9-20: Maximum mode 8086 system

UNIT 7: MEMORY INTERFACE(CONT.)

Address Decoding

- The EPROM has 11 address-connections and the microprocessor has 20.
- This means that the microprocessor sends out a 20-bit memory-address whenever it reads or writes data.
- Because the EPROM has only 11 address-pins, there is a mismatch that must be corrected.
- The decoder corrects the mismatch by decoding the address pins that do not connect to the memory component.

Simple NAND Gate Decoder

- When the 2K*8 EPROM is used, address-connections A_{10}-A_0 of the 8088 are connected to address-inputs A_{10}-A0 of the EPROM (Figure 10-13).
- The remaining 9 address-pins(A_{19}-A_{11}) are connected to the inputs of a NAND gate decoder.
- The decoder selects the EPROM from one of the 2K-byte sections of the 1MB memory.
- A single NAND gate decodes the memory-address.
- The output of NAND gate is a logic 0 whenever the 8088 address-pins attached to its inputs(A_{19}-A_{11}) are all logic 1s.
- The active low, logic 0 output of the NAND gate decoder is connected to the $\overline{CE}$ input-pin that selects/enables the EPROM.
- Whenever $\overline{CE}$ is a logic 0, data will be read from the EPROM only if $\overline{OE}$ is also a logic 0.
- The $\overline{OE}$ pin is activated by the 8088 RD signal.

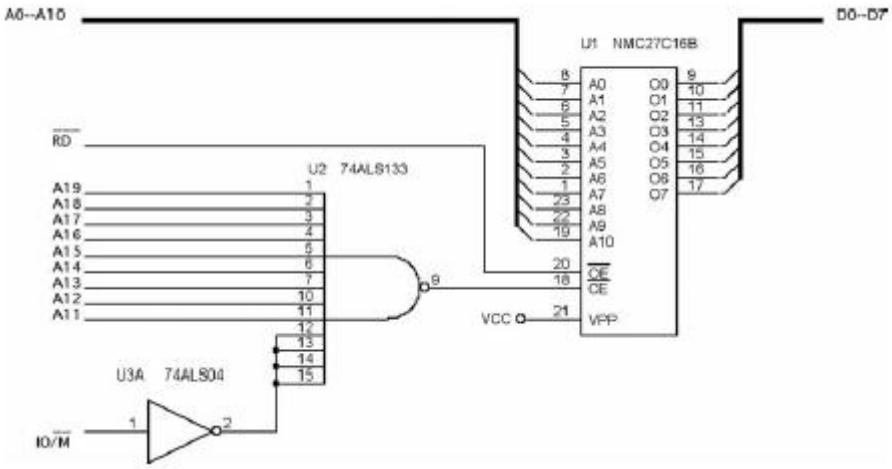


Figure 10-13:A simple NAND gate decoder that selects a 2716 EPROM for memory location FF800H-FFFFFH

The 3:8 Line Decoder (74LS138)

Sample Decoder Circuit

- The outputs of decoder are connected to eight different 2764 EPROM memory-devices (Fig: 10-15).
- The decoder selects eight 8KB blocks of memory for a total memory-capacity of 64KB.
- The decoder's outputs are connected to the $\overline{CE}$ inputs of the EPROMs, and
  the $\overline{RD}$ signal from the 8088 is connected to the $\overline{OE}$ inputs of the EPROMs.
  This allows only the selected EPROM to be enabled and to send its data to the microprocessor through the data-bus whenever $\overline{RD}$ becomes a logic 0.
- A 3-input NAND gate is connected to address-bits A_{19}-A_{17}.
- When all 3 address-inputs are high, the output of this NAND gate goes low and enables input $\overline{G2B}$ of the 74LS138.
- Input G1 is connected directly to A_{16}. (In other words, in order to enable this decoder, the first four address connections A_{19}-A_{16} must all be high).
- The address-inputs C, B and A connect to microprocessor address pins A_{15}-A_{13}. These 3 address-inputs determine which output pin goes low and which EPROM is selected whenever the 8088 outputs a memory-address within this range to the memory-system.

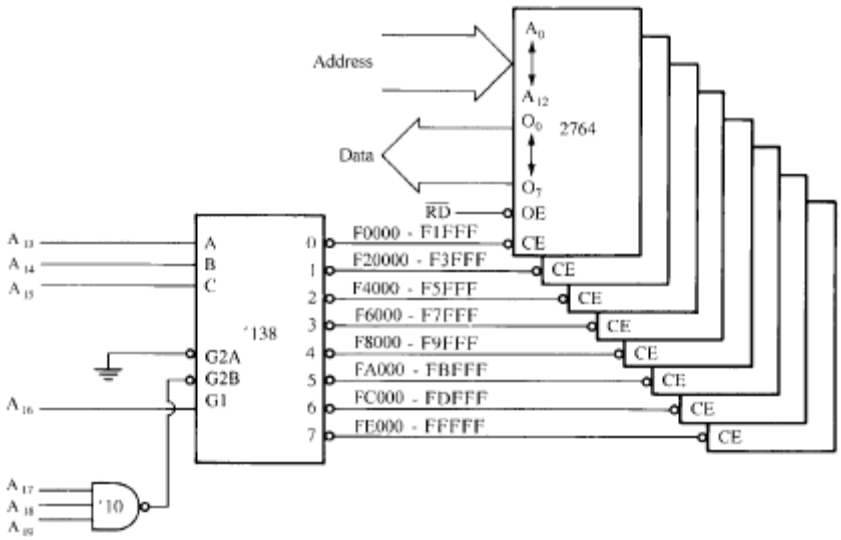


Figure 10-15: A circuit that uses eight 2764 EPROMs for a 64K*8 section of memory in an 8088 microprocessor-based system. The addresses selected in this circuit are F0000H-FFFFFH.

UNIT 7(CONT.): BASIC I/O INTERFACE

Isolated I/O

• The term isolated describes how the I/O locations are isolated from the memory-system in a separate I/O address-space (Figure 11-1).
• The addresses of I/O devices (called ports) are separate from the memory.
• Separate control-signals for the I/O space are used which indicates an I/O read($\overline{IO}\overline{RC}$) or an I/O write($\overline{IO}\overline{WC}$) operation.
• An 8-bit port-address is used to access devices located on the system-board (e.g. timer and keyboard interface). while a 16-bit port is used to access serial & parallel ports as well as video & disk drive systems.
• Disadvantage: Data transferred between I/O & microprocessor must be accessed by the IN, INS, OUT and OUTS instructions.

Memory Mapped I/O

• The I/O device is treated as a memory-location in the memory-map.
• Advantage: i) Any memory transfer instruction can be used to access the I/O device
ii) $\overline{IO}\overline{RC}$ and $\overline{IO}\overline{WC}$ signals have no function in a memory-mapped I/O system and may reduce the amount of circuitry required for decoding.
• Disadvantage: Portion of the memory system is used as the I/O map (This reduces the amount of memory available to applications).

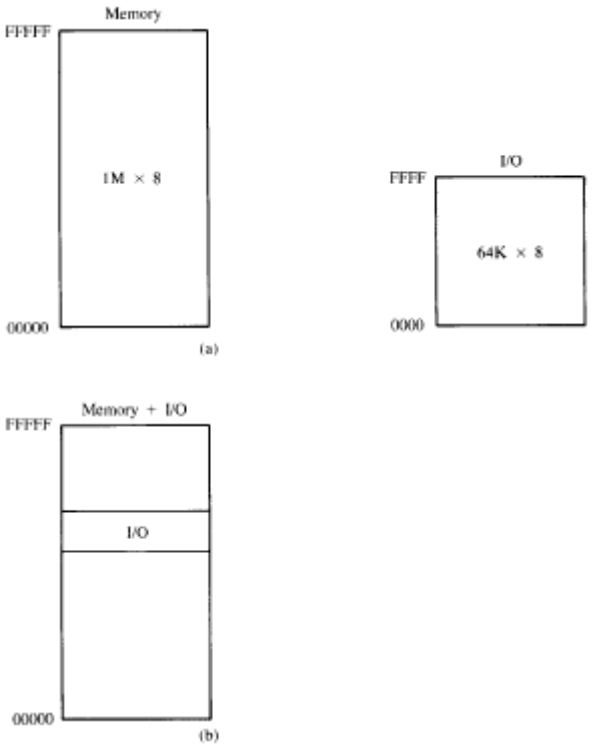


Figure 11-1:The memory and I/O maps for the 8086 microprocessor
a)isolated I/O b)memory mapped I/O

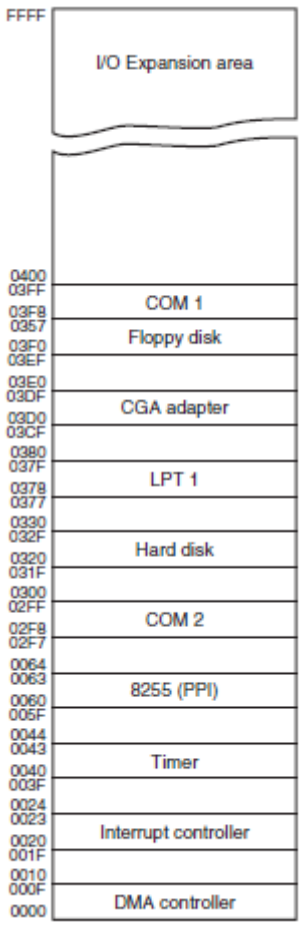


Figure 11-4:The I/O map of a personal computer illustrating many of the fixed I/O areas

Personal Computer I/O Map

- I/O space between ports 0000H & 03FFH is normally reserved for computer-system & the ISA bus.
  The I/O ports located at 0400H-FFFFH are generally available for user-applications, main-board functions and the PCI bus (Figure 11-4).
- The I/O ports located between 0000H & 00FFH are accessed via the fixed-port I/O instructions;
  the ports located above 00FFH are accessed via the variable-port I/O instructions.

Basic Input & Output Interfaces

The Basic Input Interface

- Three-state buffers are used to construct the 8-bit input port (Figure 11-3).
- The external TTL data are connected to the inputs of the buffers.
  While the outputs of the buffers connect to the data-bus.
- The circuit allows the microprocessor to read the contents of the 8 switches when the select-signal $\overline{SEL}$ becomes a logic 0.
- When the microprocessor executes an IN instruction,
  → the contents of the switches are copied into the AL register &
  → the I/O port-address is decoded to generate the logic 0 on $\overline{SEL}$.
- A 0 placed on the output control-inputs(1G & 2G) of the buffer causes the data-input connections(A) to be connected to the data-output(Y) connections.
  A 1 placed on the output control-inputs(1G & 2G) of the buffer causes the device to enter the three-state high-impedance mode that effectively disconnects the switches from the data-bus.

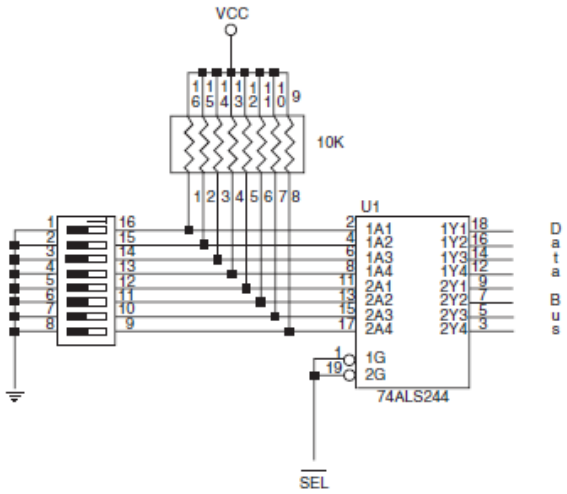


Figure 11-3: The basic input interface illustrating the connection of 8 switches.

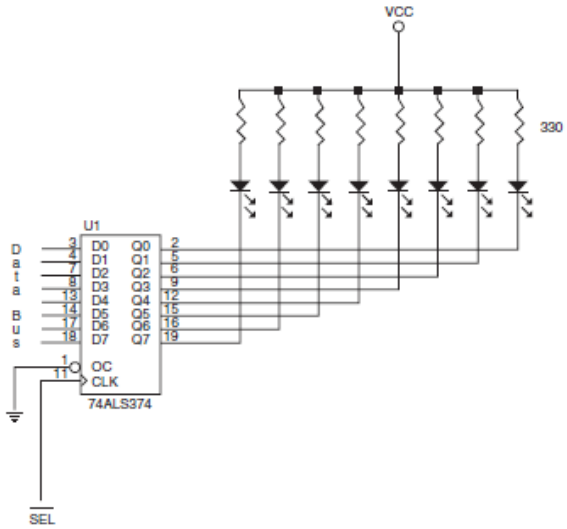


Figure 11-4:The basic output interface connected to a set of LED displays

The Basic Output Interface

- The basic output-interface receives data from the microprocessor and usually must hold it for some external-device (Figure 11-4).
- Latches are needed to hold data because
  when microprocessor executes an OUT instruction, data are only present on data-bus for less than 1.0 μs.
- When the OUT instruction executes, the data from accumulator is transferred to the latch via the data-bus.
- The D inputs of a latch are connected to the data-bus to capture the output-data, and the Q outputs of the latch are attached to the LEDs.
- When a Q output becomes a logic 0, the LED lights.
- Each time that the OUT instruction executes, the $\overline{SEL}$ signal to the latch activates, capturing the data-output to the latch from any 8-bit section of the data-bus. The data are held until the next OUT instruction executes.

UNIT 8: BASIC I/O INTERFACE(CONT.)

The Programmable Peripheral Interface(82C55)

- This is a low-cost interfacing-component.
- This has 24 pins that are programmable in groups of 12 pins.
- This can operate in 3 different modes.
- This can interface any TTL-compatible I/O device to the microprocessor.

Basic Description of the 82C55

- This has three I/O ports(labeled A,B and C) which can be programmed as group (Figure 11-18).
- Group A connections consist of port A(PA_7-PA_0) and the upper nibble of port C(PC_7-PC_4) and Group B consists of port B(PB_7-PB_0) and the lower nibble of port C(PC_3-PC_0).
- This is selected by its $\overline{CS}$ (chip select) pin for programming and for reading or writing to a port.
- Register selection is accomplished through the A_1 and A_0 input-pins that select an internal-register for programming or operation.

Table 11-2:I/O port assignments for the 82C55

A_1	A_0	Function
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Command register

- All the 82C55 pins are direct connections to the 80386SX except for the $\overline{CS}$ pin. The $\overline{CS}$ pin is decoded and selected by a 74ALS138 decoder (Figure 11-19).
- The RESET input of the 82C55 initializes the device whenever the microprocessor is reset. RESET input to 82C55 causes all ports to be setup as simple input-ports (usually mode 0 operation).

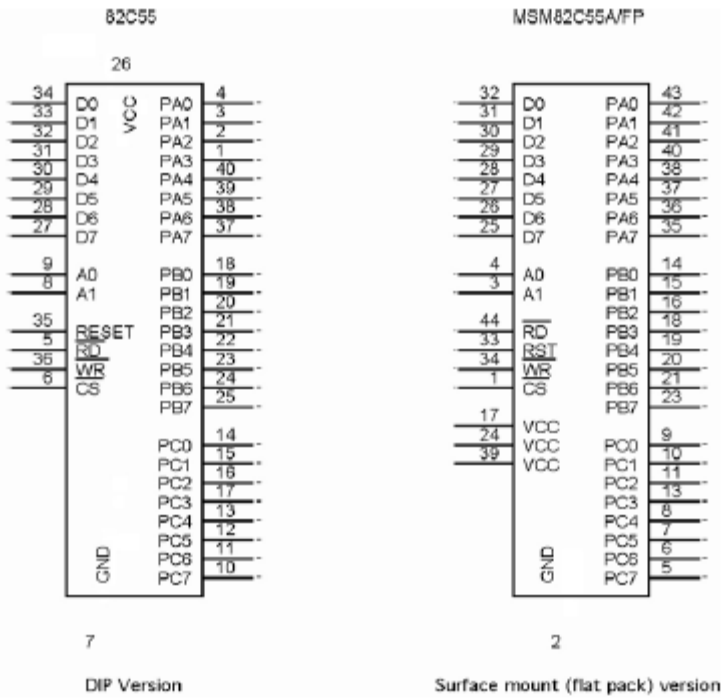


Figure 11-18:The pin-out of 82C55 PPI

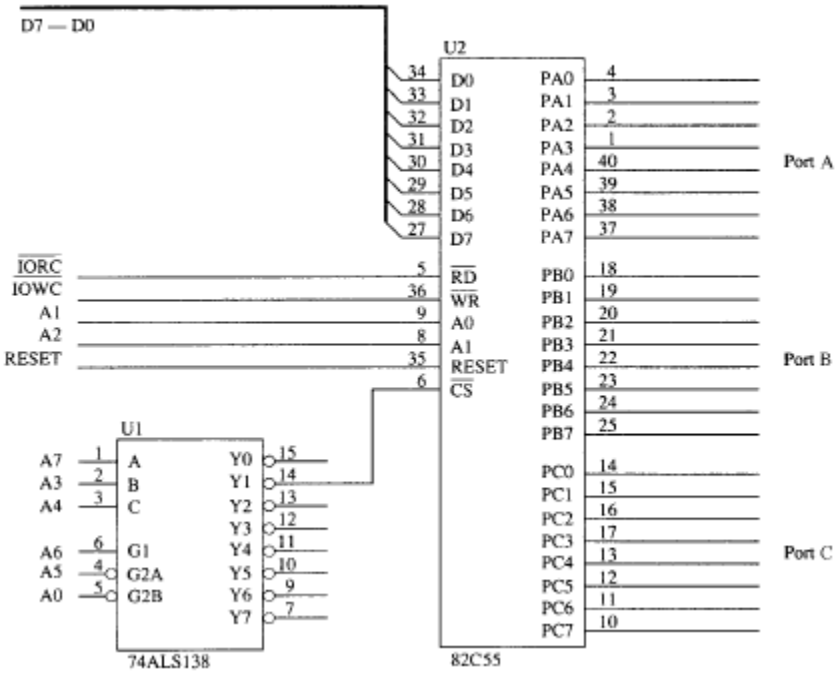


Figure 11-19:The 82C55 interface to the low bank of the 80386SX microprocessor

Programming the 82C55

- The 82C55 is programmed through the 2 internal command-registers (Figure 11-20).
- Command-byte A programs the function of group A and B,
  whereas command-byte B sets(1) or resets(0) bits of port C(only if 82C55 is programmed in mode 1 or 2)
- 82C55 can operate in following 3 modes:
  i) Mode 0 is the basic input/output mode that allows the pins of group A/B to be programmed as simple input- and latched output-connections.
  ii) Mode 1 operation is the strobed operation for group A/B connections, where data are transferred through port A/B and handshaking-signals are provided by port C.
  iii) Mode 2 operation is a bidirectional mode of operation for port A.
- Group B pins
  → are programmed as either input or output pins.
  → operates in either mode 0 or mode 1.
- Group A are programmed as either input- or output-pins. The difference is that group A can operate in mode 0, 1 and 2.
- If a 0 is placed in bit-position of the command-byte A, command-byte B is selected.

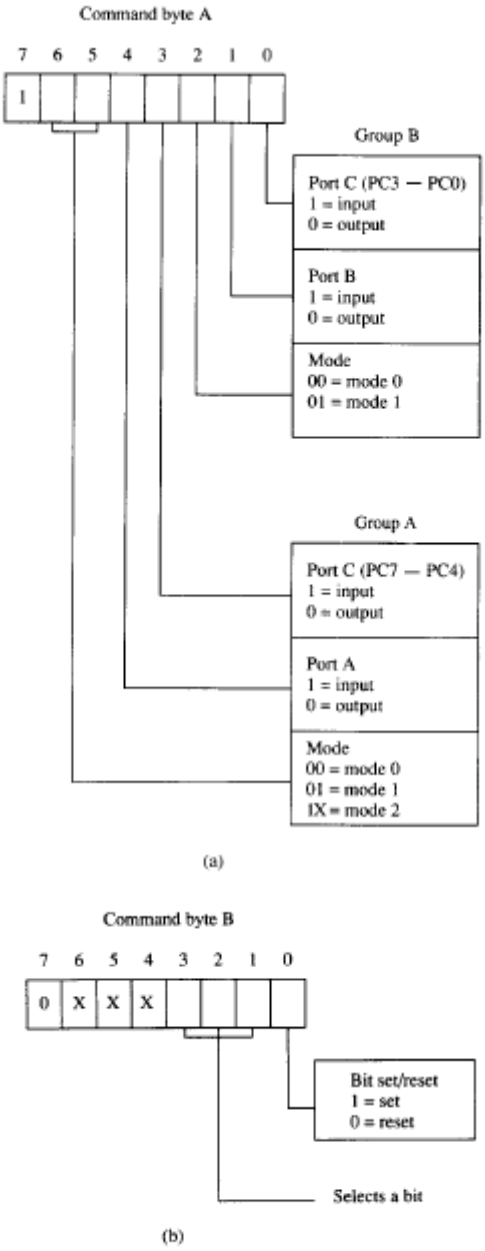


Figure 11-20:The command byte of the command register in the 82C55 a)Programs ports A,B and C b)sets or resets the bit indicated in the select a bit field

Mode 0 Operation

LED Display Interfaced to the Microprocessor through an 82C55 PIA

- Both ports A and B are programmed as simple latched output-ports (Figure 11-21).
- Port A provides the segment data-inputs to the display and
  Port B provides a means of selecting one display position at a time for multiplexing the displays.

Example 11-10: An assembly language procedure that multiplexes the 8-digit display.

```
DISP PROC NEAR
        MOV BX,8                ;load counter
        MOV AH,7FH              ;load selection pattern
        LEA SI,TABLE            ;address display data
        MOV DX,701H             ;address port B

        .REPEAT
                MOV AL,AH               ;send selection pattern to port B
                OUT DX,AL
                DEC DX
                MOV AL,[BX+SI]          ;send data to port A
                OUT DX,AL
                CALL DELAY              ;wait 1.0 ms
                ROR AH,1                ;adjust selection pattern
                INC DX
                DEC BX                  ;decrement counter
        .UNTIL BX==0
        RET
DISP ENDP

DELAY PROC NEAR
        MOV CX,0FFFFH
        L1:LOOP L4
        RET
DELAY ENDP
```

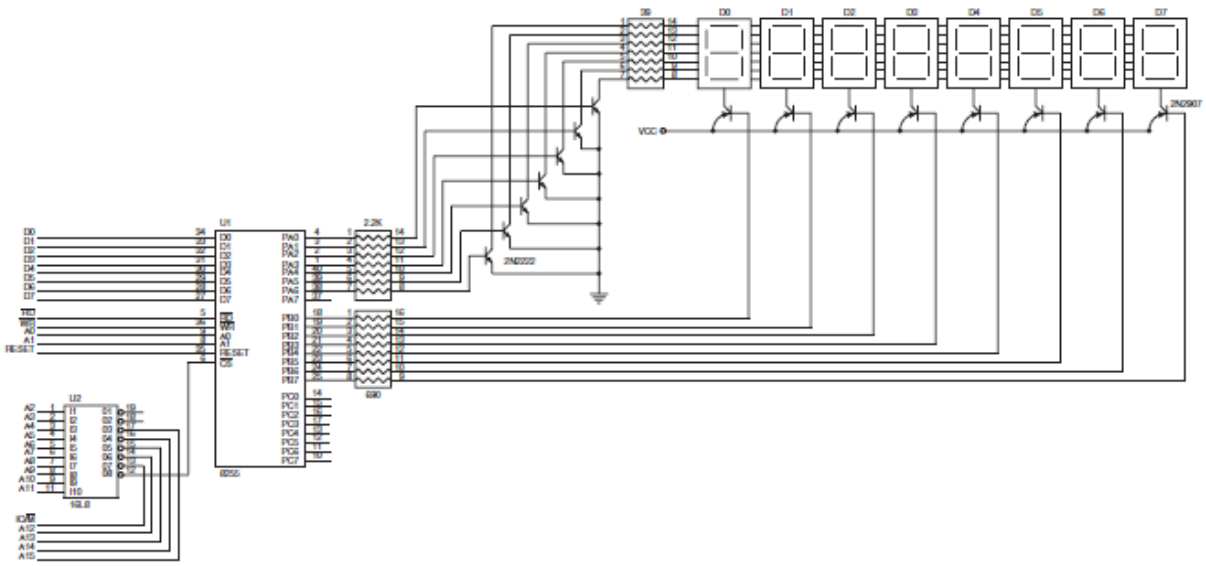


Figure 11-21:An 8-digit LED display interfaced to the 8088 microprocessor through an 82C55 PIA

An LCD Display Interfaced to the 82C55

- LCD display is a 4-line by 20-characters-per-line display that accepts ASCII code as input-data(Figure 11-22).
- The data-connections(D_0-D_7) are used to input display data and to read information from the display.
- There are 4 control-pins on the display :
    1) V_{EE} connection is used to adjust the contrast of the LED display
    2) RS(register select) selects data(RS=1) or instructions(RS=0)
    3) E(enable) must be a logic 1 for the LCD to read or write information.
    4) R/W selects a read or write operation.
- In order to program the LCD, it has to be initialized by following steps
    1) Wait at least 15ms after Vcc rises to 5.0V
    2) Output the function set command(30H),and wait for at least 4.1 ms.
    3) Output the function set command(30H) a second time, and wait for at least 100 μs.
    4) Output the function set command(30H) a third time, and wait for at least 40 μs.
    5) Output the function set command(38H) a third time, and wait for at least 40 μs.
    6) Output 08H to display the display, and wait at least 40 μs.
    7) Output a 01H to home the cursor ad clear the display, and wait at least 1.64 ms
    8) Output the enable display cursor off, and wait at least 40 μs.
    9) Output 06H to select auto-increment, shift the cursor, and wait at least 40 μs.

Table 11-3: Instructions for most LCD displays

Instruction	*Code*	*Description*	*Time*
Clear display	0000 0001	Clears the display and homes the cursor	1.64 ms
Cursor home	0000 0010	Homes the cursor	1.64 ms
Entry mode set	0000 00AS	Sets cursor movement direction (A = 1, increment) and shift (S = 1, shift)	40 μs
Display on/off	0000 1DCB	Sets display on/off (D = 1, on) (C = 1, cursor on) (B = 1, cursor blink)	40 μs
Cursor/display shift	0001 SR00	Sets cursor movement and display shift (S = 1, shift display) (R = 1, right)	40 μs
Function set	001L NF00	Programs LCD circuit (L = 1, 8-bit interface) (N = 1, 2 lines) (F = 1, 5 × 10 characters) (F = 0, 5 × 7 characters)	40 μs
Set CGRAM address	01XX XXXX	Sets character generator RAM address	40 μs
Set DRAM address	10XX XXXX	Sets display RAM address	40 μs
Read busy flag	B000 0000	Reads busy flag (B = 1, busy)	0
Write data	Data	Writes data to the display RAM or the character generator RAM	40 μs
Read data	Data	Reads data from the display RAM or character generator RAM	40 μs

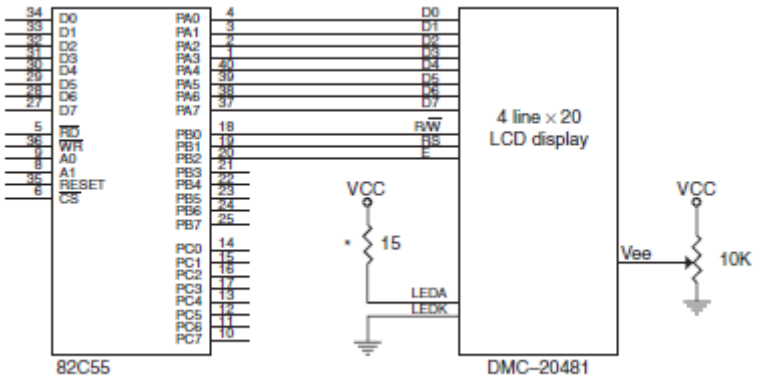


Figure 11-22:The DMC-20481 LCD display interfaced to the 82C55

A Stepper Motor Interfaced to the 82C55

- A stepper-motor
  - → is a digital-motor because it is moved in discrete-steps as it traverses through 360'
  - → has 4 coils and an armature (Fig11-23)
  - → is rotated by energizing the coils
  - → is driven by using NPN Darlington amplifier pairs to provide a large current to each coil
- 82C55 provides motor with the drive-signals that are used to rotate the armature in either right-hand or left-hand direction (Figure 11-24).
- Stepper-motor can also be operated in the half-step mode, which allows 8 steps per sequence.
- Half-stepping allows the armature to be positioned at 0', 90', 180' and 270'. (The half step position codes are 11H, 22H, 44H and 88H).

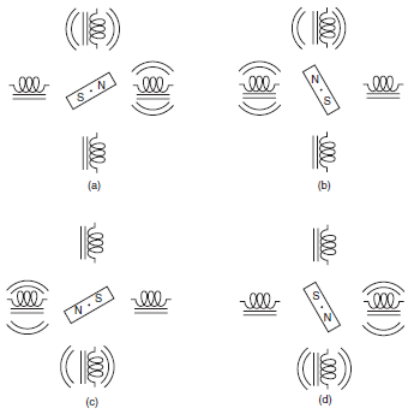


Figure 11-23: The stepper motor showing full-step operation a)45' b)135' c)225' d)315'

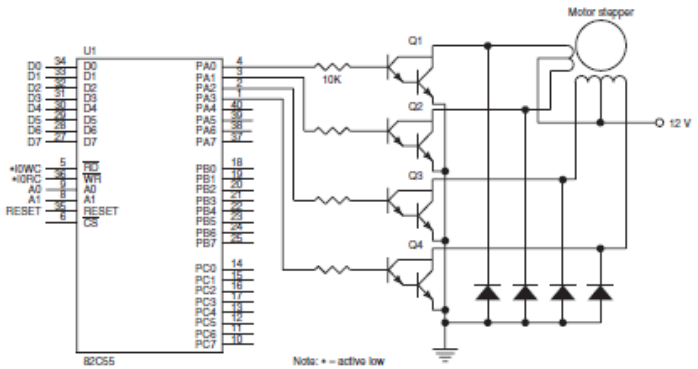


Figure 11-24: A stepper motor interface to the 82C55

Example 11-16:An assembly language procedure that controls the stepper motor

```
STEP PROC NEAR
        MOV AL,POS                          ;get position
        OR CX,CX                            ;set flag bits
        IF !ZERO?
                .IF !SIGN?
                        .REPEAT             ;if no sign
                                ROL AL,1
                                OUT PORT,AL     ;rotate step left
                                CALL DELAY      ;wait 1ms
                        .UNTILCXZ
                        .ELSE
                                AND CX,7FFFH        ;make CX positive
                                .REPEAT
                                        ROR AL,1        ;rotate step right
                                        OUT PORT,AL
                                        CALL DELAY      ;wait for 1ms
                                .UNTILCXZ
                .ENDIF
        .ENDIF
        MOV POS,AL
        RET
STEP ENDP
```

Key Matrix Interface

• Key-matrix contains 16 switches interfaced to ports A and B of an 82C55 (Figure 11-25).

• The switches are formed into a 4*4 matrix. Keys are organized into 4 rows(ROW_0-ROW_3) and 4 columns(COL_0-COL_3).

• Each row is connected to 5.0V through a 10KΩ pull-up resistor to ensure that the row is pulled high when no push-button switch is closed.

• Port A is programmed as an input port to read the rows and port B is programmed as an output port to select a column.(For example, if 1110 is output to port B pins PB_3-PB_0, column 0 has a logic 1, so the 4 keys in column 0 are selected. With a logic 0 on PB_0, the only switches that can place a logic 0 onto port A are switches 0-3.If switches 4-F are closed, the corresponding port A pins remain a logic 1).

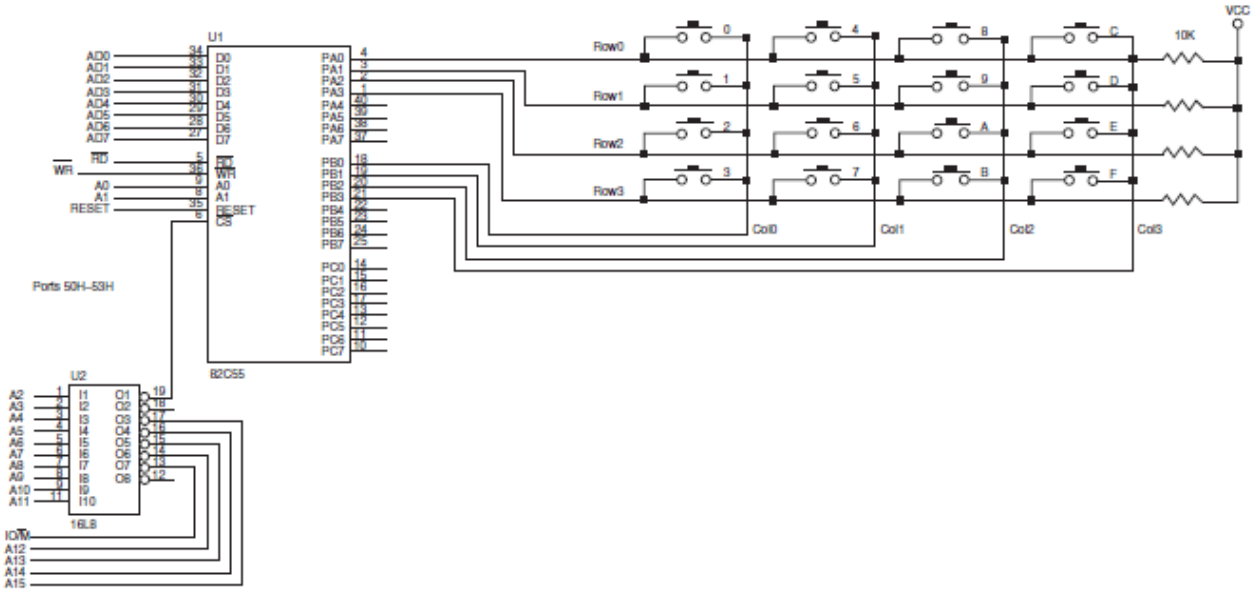


Figure 11-25:A 4*4 keyboard matrix connected to an 8088 microprocessor through the 82C55 PIA

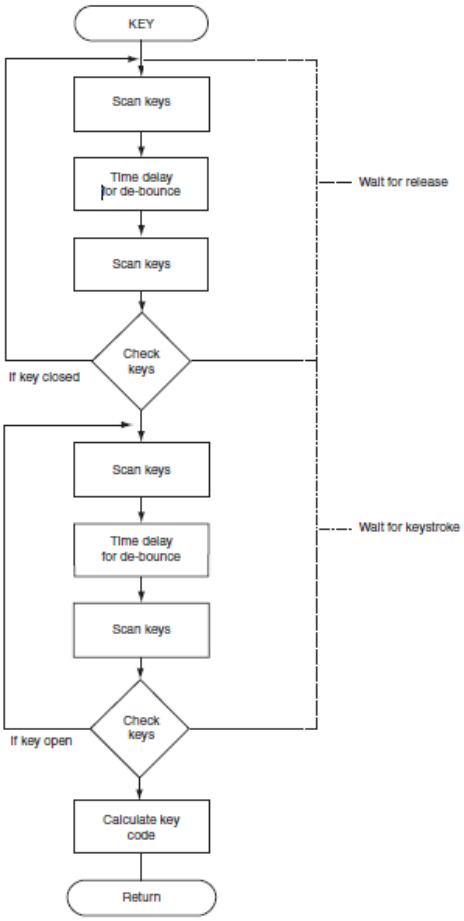


Figure 11-26:The flowchart of a keyboard scanning procedure

Example 11-17:KEY scans the keyboard and returns the key code in AL

```
COLS EQU 4
ROWS EQU 4
PORTA EQU 50
PORTB EQU 51H

KEY PROC NEAR
        MOV BL,FFH                      ;compute row mask
        SHL BL,ROWS

        MOV AL,0
        OUT PORTB,AL                    ;place zeros on port B

        .REPEAT                         ;wait for release
                .REPEAT
                        CALL SCAN
                .UNTIL ZERO?
                CALL DELAY
                CALL SCAN
        .UNTIL ZERO?

        .REPEAT                          ;wait for key
                .REPEAT
                        CALL SCAN
                .UNTIL !ZERO?
                CALL DELAY
                CALL SCAN
        .UNTIL !ZERO?

        MOV CX,00FEH
        .WHILE 1                        ;find column
                MOV AL,CL
                OUT PORTB,AL
                CALL SHORTDELAY          ;see text
                CALL SCAN
                .BREAK !ZERO?
                ADD CH,COLS
                ROL CL,1
        .ENDW

        .WHILE 1                        ;find row
                SHR AL,1
                .BREAK .IF !CARRY?
                INC CH
        .ENDW

        MOV AL,CH                      ;get key code
        RET
KEY ENDP

SCAN PROC NEAR
        IN AL,POSTA                     ;read rows
        OR AL,BL
        CMP AL,0FFH                     ;test for no keys
        RET
SCAN ENDP
```

Mode 1 Strobed Input

- Mode 1 operation causes port A &/or port B to function as latching input-devices (Figure 11-27).
- This allows external-data to be stored into the port until the microprocessor is ready to retrieve it.
- Port C is used for control(or handshaking) signals that help operate either or both port A and port B as strobed input-ports.
- Steps for input operation:
  1) When $\overline{STB}$ signal is activated, external interface sends data into the port.
  2) The IBF bit is tested with software to decide whether data have been strobed into the port.
     If IBF=, then data is input using the IN instruction.
  3) When IN instruction executes, IBF bit is cleared and data in the port are moved into AL.

Signal Definition for Mode 1 Strobed Input

$\overline{STB}$ (Strobe)

- This loads data into the port-latch which holds the information until it is input to the microprocessor via the IN instruction.

IBF(Input Buffer Full)

- This indicates that the input-latch contains information.

INTR(Interrupt Request)

- This is used to interrupt the microprocessor to perform read operation.

INTR=1 when $\overline{STB}$ =1. INTR=0 when data is read from the port by the microprocessor.

INTE(Interrupt Enable)

- This is used to enable/disable the INTR pin. INTE A bit is programmed using the PC_3 bit and INTE B is programmed using the PC_0.

PC_7, PC_6

- These are general-purpose I/O pins. The bit set & reset command is used to set or reset these 2 pins.

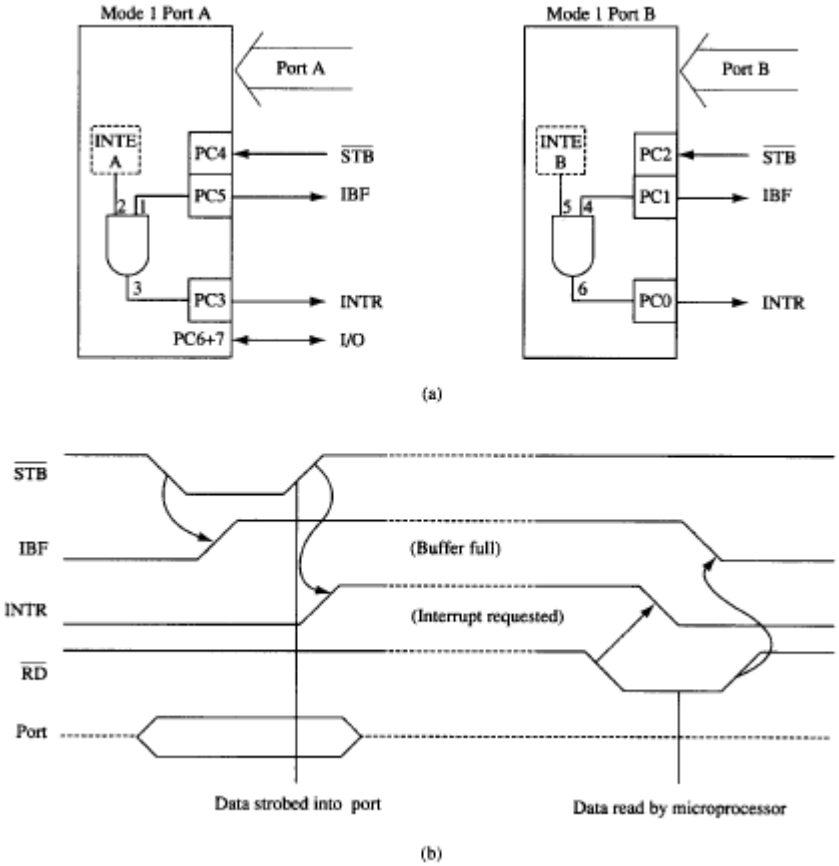


Figure 11-27:Strobed input operation of the 8255 a)internal structure and b)timing diagram

Strobed Input Example (Keyboard)

• The keyboard encoder debounces the key-switches and provides a strobe-signal whenever a key is depressed and the data-output contain the ASCII coded key-code(Figure 11-28).

Example 11-18:A procedure that reads the keyboard encoder and returns the ASCII key code in AL

```
BIT5 EQU 20H
PORTC EQU 22H
PORTA EQU 24H

READ PROC NEAR
        .REPEAT                 ;poll IBF bit
                IN AL,PORTC
                TEST AL,BIT5
        .UNTIL !ZERO?
        IN AL,PORTA             ;get ASCII data
        RET
READ ENDP
```

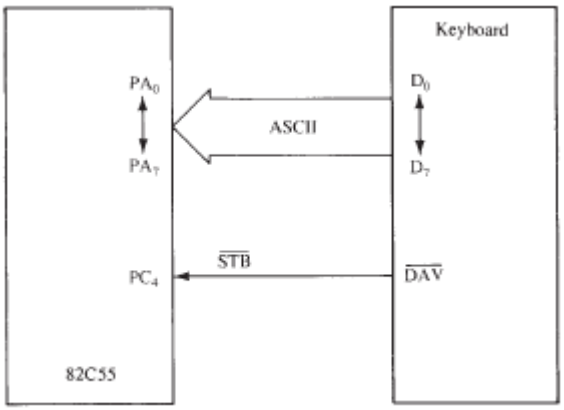


Figure 11-28:Using the 82C55 for strobed input operation of a keyboard

Mode 1 Strobed Output

- Steps for write operation
  1) When $\overline{OBF}$ is activated, data is written to the output-port
  2) Then, external I/O device removes the data by sending the $\overline{ACK}$ signal to the output-port
  3) Finally, ACK signal sets $\overline{OBF}$ =1 to indicate that the buffer is not full (Figure 11-29)

Signal Definitions for Mode 1 Strobed Output

$\overline{\text{OBF}}$ ***(Output Buffer Full)***

- $\overline{OBF}$ =0 whenever data is output to the port A or port B latch.
  $\overline{OBF}$ =1 whenever $\overline{ACK}$ pulse returns from the external device.

$\overline{\text{ACK}}$***(Acknowledge)***

- This is a response from an external-device, indicating that it has received the data from the 82C55 port.

INTR(Interrpt Request)

- This is used to interrupt the microprocessor to perform write operation.
  INTR=1 when $\overline{OBF}$ =1. INTR=0 when data is written into the port by the microprocessor.

INTE(Interrupt Enable)

- This is used to enable/disable the INTR pin. INTE A bit is programmed using the PC_3 bit and INTE B is programmed using the PC_0.

PC_4, PC_5

- These are general-purpose I/O pins.

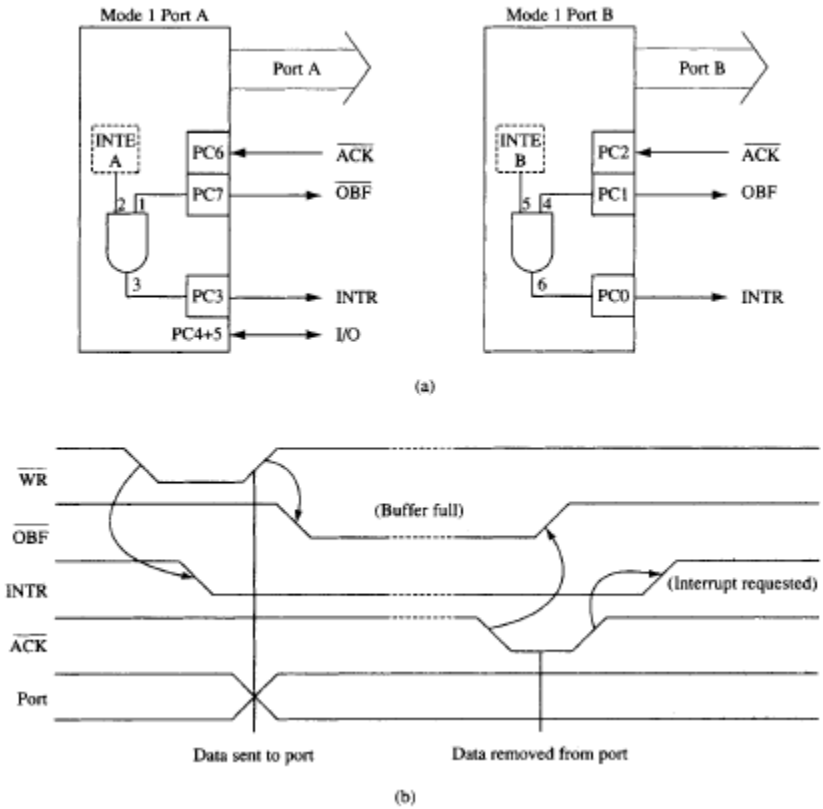


Figure 11-29:Strobed output operation of the 82C55 a)internal structure and b)timing diagram

Strobed Output Example(Printer)

- Port B is connected to a parallel-printer with 8 data-inputs for receiving ASCII coded-data.
- $\overline{DS}$ (data strobe) input is used to strobe data into the printer. $\overline{ACK}$ output is used to acknowledge the receipt of the ASCII character (Figure 11-30).
- PC4 is used with software to generate the $\overline{DS}$ signal.
- The $\overline{ACK}$ signal that is returned from the printer acknowledges the receipt of the data and is connected to the $\overline{ACK}$ input of the 82C55.

Example 11-19: A procedure that transfers an ASCII character from AH to the printer connected to port B .

```
BIT1 EQU 2
PORTC EQU 22H
PORTA EQU 24H
CMD EQU 26H
PRINT PROC NEAR
        .REPEAT                 ;wait for printer ready
                IN AL,PORTC
                TEST AL,BIT1
        .UNTIL !ZERO?
        MOV AL,AH               ;send ASCII
        OUT PORTB,AL
        MOV AL,8
        OUT CMD,AL
        MOV AL,9
        OUT CMD,AL
        RET
PRINT ENDP
```

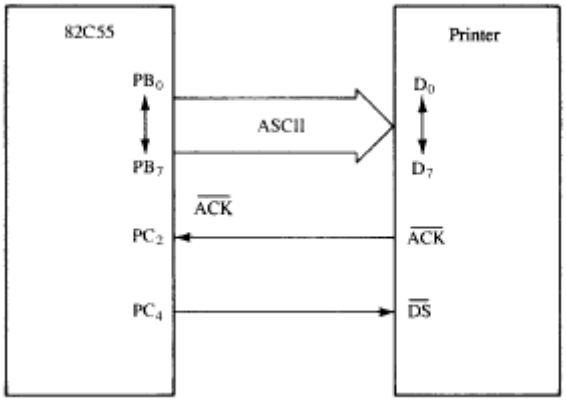


Figure 11-30:The 82C55 connected to a parallel printer interface that illustrates the strobed output mode of oeration for the 82C55

Mode 2 Bidirectional Operation

- In mode 2, port A is bidirectional i.e. data can be transmitted and received over the same 8 wires
- Bidirectional bused data are useful when interfacing two computers (Figure 11-31).

Signal Definitions for Bidirectional Mode 2

INTR(Interrpt Request)

- This is used to interrupt the microprocessor for both input- and output-conditions.

$\overline{\text{OBF}}$ ***(Output Buffer Full)***

- $\overline{OBF}$ =0 whenever data is output to the port A or port B latch.
  $\overline{OBF}$ =1 whenever $\overline{ACK}$ pulse returns from the external device.

$\overline{\text{ACK}}$***(Acknowledge)***

- This is a response from an external-device, indicating that it has received the data from the 82C55 port.

$\overline{\text{STB}}$ ***(Strobe)***

- This loads data into the port latch which holds the information until it is input to the microprocessor via the IN instruction.

IBF(Input Buffer Full)

- It indicates that the input latch contains information.

INTE(Interrupt Enable)

- It is used to enable/disable the INTR pin.

PC_0, PC_1 and PC_2

- These are general-purpose I/O pins.

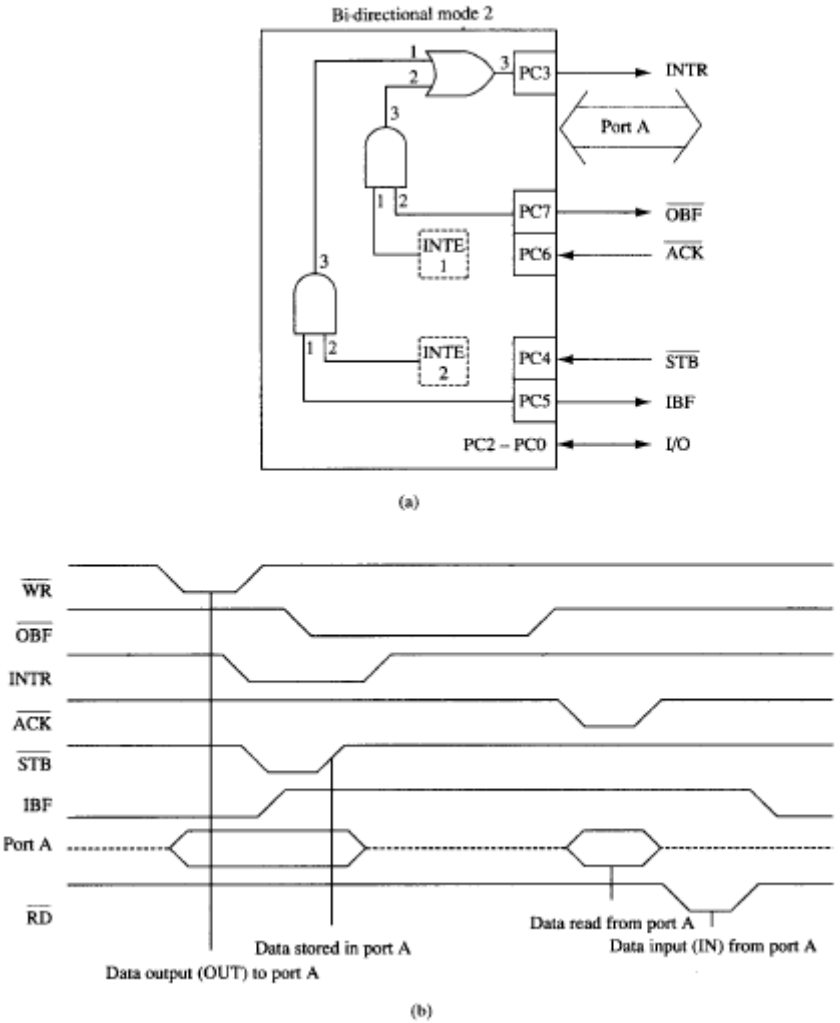


Figure 11-31:Mode 2 operation of the 82C55 a)internal structure b)timing diagram

The Bidirectional Bus

• Steps for input operation:

1) When $\overline{STB}$ signal is activated, external interface sends data into the port (Figure 11-32)
2) The IBF bit is tested with software to decide whether data have been strobed into the port
   If IBF=, then data is input using the IN instruction
3) When the IN instruction executes, the IBF bit is cleared and the data in the port are moved into AL

Example 11-21: A procedure that reads data from the bidirectional bus into AL

```
BIT5 EQU 20H
PORTC EQU 22H
PORTA EQU 24H
READ PROC NEAR
        .REPEAT                 ;test IBF
                IN AL,PORTC
                TEST AL,BIT5
        .UNTIL !ZERO?
        IN AL,PORTA
        RET
READ ENDP
```

• Steps for write operation

1) When $\overline{OBF}$ is activated, data is written to the output port.
2) Then, external I/O device removes the data by sending the $\overline{ACK}$ signal to the output port.
3) Finally, $\overline{ACK}$ signal sets $\overline{OBF}$ =1 to indicate that the buffer is not full.

Example 11-20: A procedure transmits AH through the bidirectional bus

```
BIT7 EQU 80H
PORTC EQU 22H
PORTA EQU 24H
TRANS PROC NEAR
        .REPEAT                 ;test OBF
                IN AL,PORTC
                TEST AL,BIT7
        .UNTIL !ZERO?
        MOV AL,AH               ;send ASCII
        OUT PORTA,AL
        RET
TRANS ENDP
```

		Mode 0		Mode 1		Mode 2
Port A		IN	OUT	IN	OUT	I/O
Port B		IN	OUT	IN	OUT	Not used
Port C	0	IN	OUT	$\overline{INTR_B}$	$\overline{INTR_B}$	I/O
	1			IBF_B	$\overline{OBF_B}$	I/O
	2			$\overline{STB_B}$	$\overline{ACK_B}$	I/O
	3			$\overline{INTR_A}$	$\overline{INTR_A}$	INTR
	4			$\overline{STB_A}$	I/O	$\overline{STB}$
	5			IBF_A	I/O	IBF
	6			I/O	$\overline{ACK_A}$	$\overline{ACK}$
	7			I/O	$\overline{OBF_A}$	$\overline{OBF}$

Figure 11-32:A summary of the port connections for the 82C55 PIA

8254 Programmable Interval Timer

- This consists of 3 independent 16-bit programmable counters (or timers)
- Each counter is capable of counting in binary or BCD (Figure 11-34).
- This is useful wherever the microprocessor must control real-time events.
  Example: Real time clock, events counter
- This is used to do the following
  1) Generate a basic timer interrupt that occurs at approximately 18.2 Hz.
  2) Cause the DRAM memory system to be refreshed.
  3) Provide a timing source to the internal speaker and other devices.

8254 Functional Description (Pin Definitions)

A0,A1(Address)

- These select 1 of 4 internal registers.

Table 11-4: Address selection inputs of the 8254

A_1	A_0	*Function*
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Control word

CLK(Clock)

- This is the timing source for each of the internal counters.

$\overline{CS}$ ***(Chip Select)***

- This enables the 8254 for programming and reading or writing a counter.

G(Gate)

- This controls the operation of the counter in some modes of operation.

GND(Ground)

- This is connected to the system ground bus.

OUT(Output)

- This where the waveform generated by the timer is available.

$\overline{RD}$ ***(Read)***

- This causes data to be read from the 8254 and is connected to the $\overline{IO}\overline{RC}$ signal.

Vcc(Power)

- This is connected to the +5.0V power supply.

$\overline{WR}$ ***(Write)***

- This causes data to be written to the 8254 and is connected to the $\overline{IO}\overline{WC}$ signal.

Programming the 8254

- Each counter is individually programmed by writing a control-word, followed by the initial count.
- The control-word allows the programmer to select the counter, mode of operation and type of operation(read/write).
- Each counter may be programmed with a count of 1 to FFFFH. A count of 0 = FFFFH+1 or 10,000 in BCD. A minimum count of 1 applies to all modes of operation except modes 2 and 3(which have a minimum count of 2)

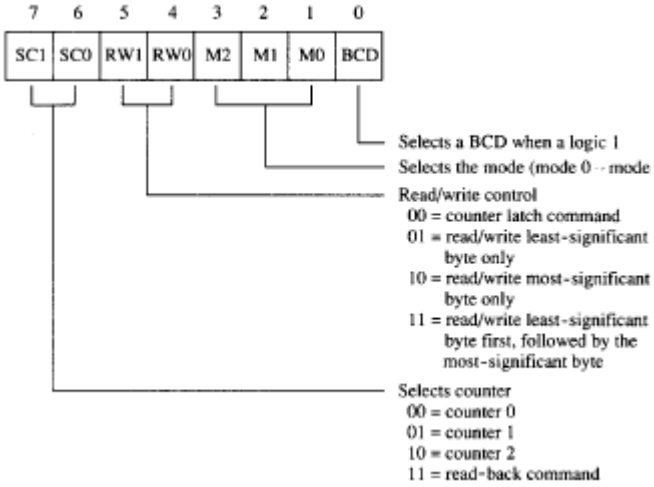


Figure 11-34: The control word for the 8254-2 timer

Modes of Operation

Mode 0

• Allows the counter to be used as an events counter (Figure 11-35)

• The output becomes a logic 0 when the control-word is written and remains there until N plus the number of programmed counts. (For example, if a count of 7 is programmed, the output will remain a logic 0 for 8 counts beginning with N)

Mode 1

• Causes the counter to function as a retriggerable, monostable multivibrator.

• The G input triggers the counter so that it develops a pulse at the OUT connection that became a logic 0 for the duration of the count.(If the count is 5,then the OUT connection goes low for 5 clocking periods when triggered).

Mode 2

• Allows the counter to generate a series of continuous pulses that are one clock pulse wide. The separation between pulses is determined by the count. (For example, for a count of 5, the output is a logic 1 for four clock periods and 0 for one clock period)

Mode 3

• Generates a continuous square wave at the OUT connection, provided that the G pin is a logic 1.

• If the count is even, the output is high for one half of the count and low for one half of the count.
If the count is odd, the output is high for one clocking period longer than it is low(for example, if the counter is programmed for a count of 6,the output is high for 3 clocks and low for 3 clocks)

Mode 4

• Allows the counter to produce a single pulse at the output.(For example, if the count is programmed as a 8,the output is high for 8 clocking periods and low for 1 clocking period)

Mode 5

• A hardware triggered one-shot that functions as mode 4 except that it is started by a trigger pulse on the G pin instead of by software.(This mode is also similar to mode 1 because it is retriggerable).

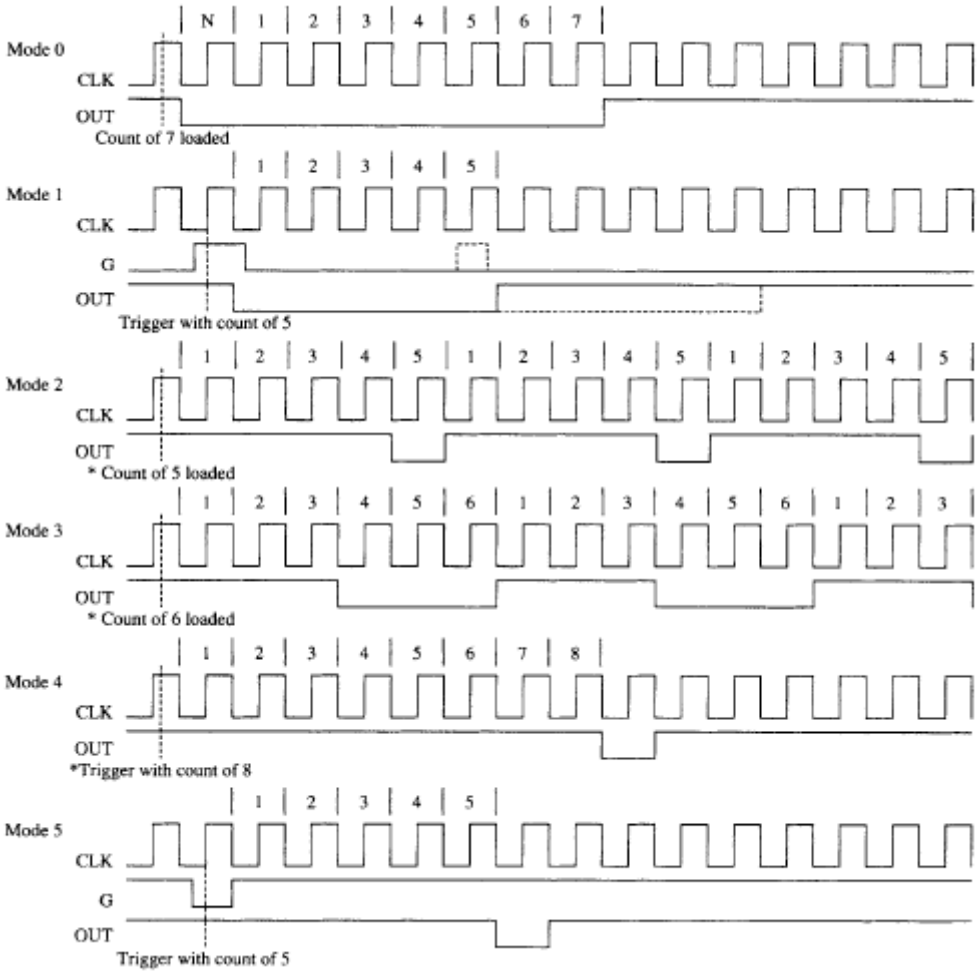


Figure 11-35:The six modes of operation for the 8254-2 Programmable interval timer

Reading a Counter

• Read-back control word is used when it is necessary for the contents of more than one counter to be read at the same time (Figure 11-38).

• Status register shows whether the counter is at its null state(0),& how the counter is programmed(Figure 11-39).

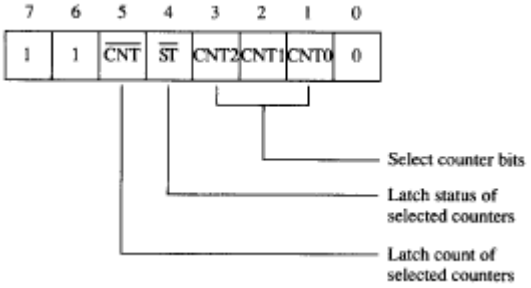


Figure 11-38:The 8254-2 read-back control word

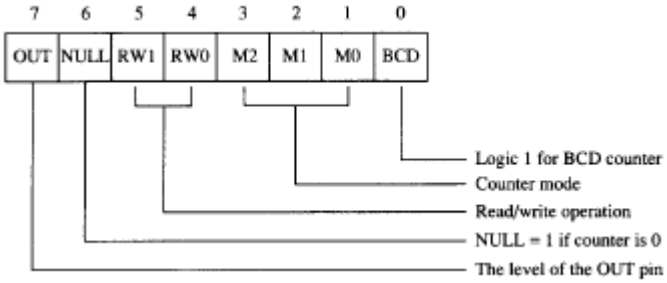


Figure 11-39:The 8254-2 status register

UNIT 8(CONT.): INTERRUPTS

Interrupt Vectors

- The interrupt-vector table
  - → is located in first 1024 bytes of memory at addresses 000000H-0003FFH
  - → contains 256 different four-byte interrupt-vectors
- An interrupt-vector contains address of the ISP(Interrupt Service Procedure).
- First 32 interrupt-vectors are reserved. Remaining 224 vectors are available as user interrupt vectors.
- In real mode, each vector is 4 bytes long and contains the starting-address of the ISP.

List of Interrupt Vectors

TYPE 0(Divide Error)

- This interrupt occurs whenever
  - → an attempt is made to divide by zero or
  - → the result from a division overflows. (Figure 12-2)

TYPE 1(Single Step/Trap)

- This interrupt occurs after the execution of each instruction if TF=1.

TYPE 2(NMI)

- This interrupt occurs when NMI=1.This is non-maskable which means that it cannot be disabled.

TYPE 3

- This is used by INT 3 instruction to access its ISP (INT 3 is used to store a breakpoint in program for debugging)..

TYPE 4

- This is used by INTO instruction to interrupt the program if OF=1.

TYPE 5

- BOUND instruction compares a register with boundaries stored in the memory.
- If register contents are out of bounds, a type 5 interrupt occurs.
  If *(first word)<(register contents)<(second word),* no interrupt occurs.

TYPE 6(Invalid Opcode)

- This interrupt occurs whenever an undefined opcode is encountered in a program.

TYPE 7(Coprocessor not Available)

- If an ESC/WAIT instruction executes and the coprocessor is not found, a type 7 interrupt occurs.

TYPE 8(Double Fault)

- This interrupt occurs whenever 2 separate interrupts occur during the same instruction.

TYPE 9(Coprocessor Segment Overrun)

- This interrupt occurs if the ESC instruction memory operand extends beyond offset address FFFFH in real mode.

TYPE 10(Invalid Task State Segment)

- This interrupt occurs if the TSS is invalid(because segment limit field is not 002BH or higher).

TYPE 11(Segment not Present)

- This interrupt occurs when the protected mode P bit(P=0) in a descriptor indicates that the segment is not present or not valid.

TYPE 12(Stack Segment Overrun)

- This interrupt occurs if the limit of the stack segment is exceeded.

TYPE 13(General Protection Fault)

- This interrupt occurs for any of the following protection violations
  1) Segment limit exceeded
  2) Descriptor table limit exceeded
  3) Privilege rules violated
  4) Write to protected code segment
  5) Write to read-only data segment

TYPE 14(Page Fault)

- This interrupt occurs for any page fault memory/code access in the Pentium-Core2 microprocessors.

TYPE 16(Coprocessor Error)

- This interrupt occurs whenever a coprocessor ERROR=0 is encountered for the ESC or WAIT instructions in the Pentium-Core2 microprocessors.

TYPE 17(Alignment Check)

- This interrupt indicates that word-data are addressed at an odd memory location.

TYPE 18(Machine Check)

- This activates a system memory management mode interrupt in Pentium-Core2 microprocessors.

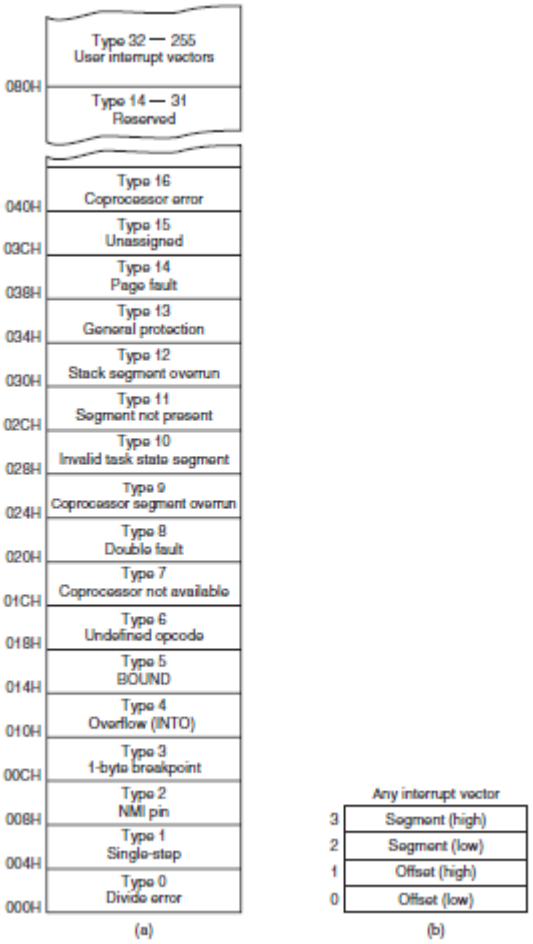


Figure 12-2: a)interrupt vector table b)contents of an interrupt vector

Interrupt Instructions: BOUND, INTO, INT, INT 3 & IRET

BOUND

- This instruction compares a register with boundaries stored in the memory.
- For e.g. *BOUND AX, DATA*

;If register contents are out of bounds, a type 5 interrupt occurs.
;If (DATA+DATA1) < AX < (DATA2+DATA3) , no interrupt occurs.

INTO

- This instruction checks the overflow-flag(OF).
- If OF=1, this instruction calls procedure whose address is stored in interrupt-vector type number 4.
  If OF=0, this instruction performs no operation & next sequential instruction in program is executed.

INT n

- This instruction calls the ISP that begins at the address represented in vector-number n.
  For e.g. INT 5 calls the ISP whose address is stored in the vector type number 5.
  (To determine the vector address, just multiply the vector type number 'n' by 4,which gives the beginning address of the 4-byte long interrupt vector. INT 5=4*5 or 20)
- Each INT instruction is stored in 2 bytes of memory:
  1) The first byte contains the opcode, and
  2) The second byte contains the interrupt type number.
- The only exception: INT 3 instruction which is a one-byte instruction.

INT 3

- This instruction is often used to store a breakpoint in a program for debugging.
  This is because it is easy to insert a one-byte instruction into a program.

IRET

- This instruction is a used to return for both software and hardware interrupts.
- This restores 6 bytes from the stack: 2 for IP, 2 for CS and 2 for flags.

The Purpose of Interrupts

- An interrupt is a hardware-initiated procedure that interrupts whatever program is currently executing.
- Interrupts are particularly useful when interfacing I/O devices that provide or require data at relatively low data transfer rates.
- Interrupt processing allows the microprocessor to execute other software while the keyboard operator is thinking about what key to type next. (Figure 12-1)

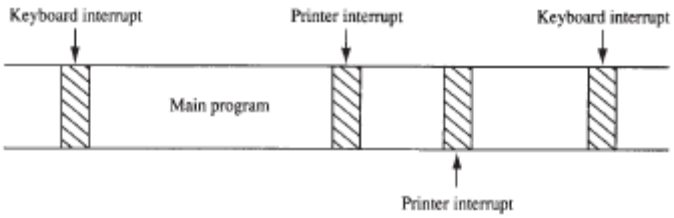


Figure 12-1: A time line that indicates interrupt usage in a typical system

Operation of a Real Mode Interrupt

• When the microprocessor completes executing the current instruction, it determines whether an interrupt is active by checking (in the order presented):

1) Instruction executions
2) Single-step or Trap
3) NMI
4) Coprocessor segment overrun
5) INTR
6) INT instructions

• If any one of the condition is true, the following sequence of events occurs:

1) The contents of the flag-register are pushed onto the stack
2) Both the IF(interrupt) and TF(trap) flags are cleared. This disables the INTR pin and the trap feature.
3) The contents of the CS register are pushed onto the stack.
4) The contents of the IP register are pushed onto the stack.
5) The interrupt-vector contents are fetched
   then placed into both IP and CS
   so that the next instruction executes at the ISP.
6) When IRET instruction is encountered at the end of ISP, 6 bytes are removed from the stack:2 for IP,2 for CS and 2 for flags.

Operation of a Protected Mode Interrupt

• Protected mode uses a set of 256 interrupt descriptors (in place of interrupt-vectors) that are stored in a IDT(Interrupt Descriptor Table).
• IDT is 256*8(2K) bytes long, with each descriptor containing 8 bytes. (Figure 12-3)
• IDT is located at any memory location in the system by the IDTR(interrupt-descriptor table address register).
• Each entry in the IDT contains the address of the ISP.
• The address is in the form of segment selector and a 32-bit offset address.
• This also contains the P bit(present) and DPL bits to describe the privilege level of the interrupt.
• In the 64-bit mode of the Pentium4-Core2, an IRETQ must be used to return from an interrupt.

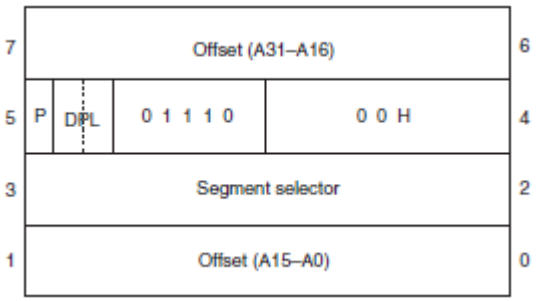


Figure 12-3: Protected Mode Interrupt descriptor

Interrupt flag bits

- If IF=1, INTR pin is enabled to cause an interrupt.
  If IF=0, INTR pin is disabled from causing an interrupt.
- If TF=1, trap interrupt occurs after each instruction is executed.
  If TF=0, normal program execution occurs.
- The interrupt flag is set & cleared by the STI & CLI instructions respectively (Figure 12-4).

Example 12-1: A procedure that sets the TRAP flag bit to enable trapping

```
TRON PROC FAR USES AX BP
        MOV BP,SP               ;get SP
        MOV AX,[BP+8]           ;retrieve flags from stack
        OR AH,1                 ;set trap flag
        MOV [BP+8],AX
        IRET
TRON ENDP
```

Example 12-2: A procedure that clears the TRAP flag bit to disable trapping

```
TROFF PROC FAR USES AX BP
        MOV BP,SP               ;get SP
        MOV AX,[BP+8]           ;retrieve flags from stack
        AND AH,0FEH             ;clear trap flag
        MOV [BP+8],AX
        IRET
TROFF ENDP
```

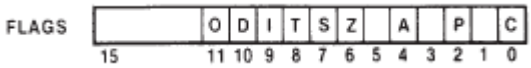


Figure 12-4 : Flag register

Hardware Interrupts

- Whenever the NMI input is activated, a type 2 interrupt occurs.
- NMI is often used for parity error and other major system faults such as power failures.
- Power failures are easily detected by monitoring the AC power line and causing an NMI interrupt whenever AC power drops out.
- An optical isolator provides isolation from the AC power line. (Figure 12-6)
- The output of the isolator is shaped by a Schmitt-trigger inverter that provides a 60Hz pulse to the trigger input of the 74LS122 retriggerable, monostable multivibrator.
- Because 74LS122 is retriggerable, as long as AC power is applied, the Q output remains triggered at logic 1 and Q remains logic 0.
- If the AC power fails, the 74LS122 no longer receives trigger pulses from the 74LS14, which means that Q becomes a logic 0 and Q becomes a logic 1,interrupting the microprocessor through the NMI pin.
- The ISP stores the contents of all internal registers into a battery-backed-up memory. (Figure 12-7)

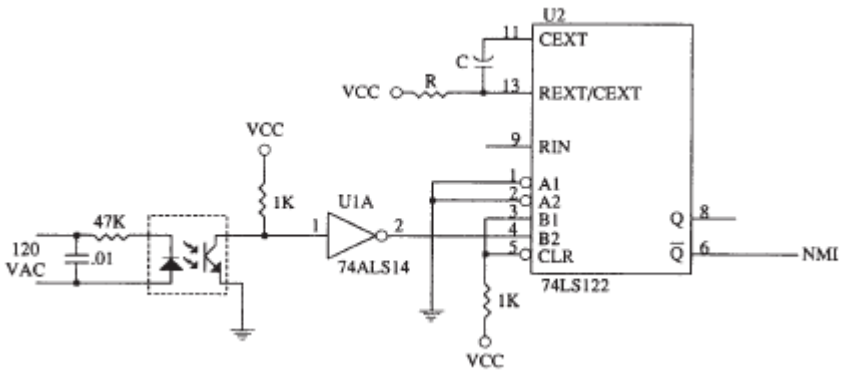


Figure 12-6: A power failure detection circuit

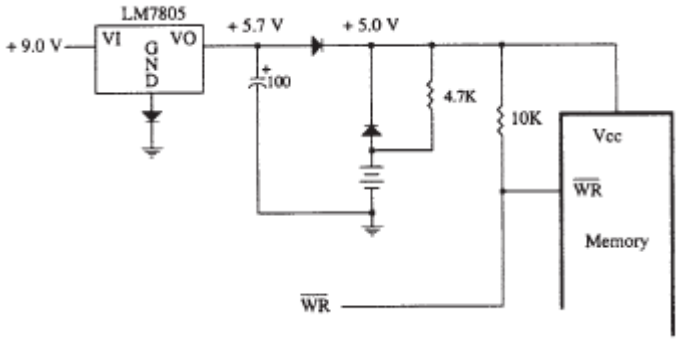


Figure 12-7: A battery-backed-up memory system using a NiCad or gel cell

INTR and INTA

- INTR is level-sensitive which means that it must be held at logic 1 level until it is recognized.
- INTR pin is set by an external event and cleared inside the ISP.
- INTR pin is automatically disabled once it is accepted by the microprocessor and re-enabled by the IRET instruction at the end of the ISP.
- The microprocessor responds to the INTR input by pulsing the INTA output in anticipation of receiving an interrupt vector type number on data bus connections D_7-D_0.
- Following methods can be used for generating interrupt vector type number:

  1) Method -1: As shown in figure 12.9, INTA pin is not connected in this circuit. Because resistors are used to pull the data bus connections(D_0-D_7) high, the microprocessor automatically sees vector type number FFH in response to the INTR input.

  2) Method -2 (Using a Three-State buffer for INTA): In response to the INTR, the microprocessor outputs the INTA that is used to enable a 74LS244 three-state octal buffer. The octal buffer applies the interrupt vector type number to the data bus in response to the INTA pulse.

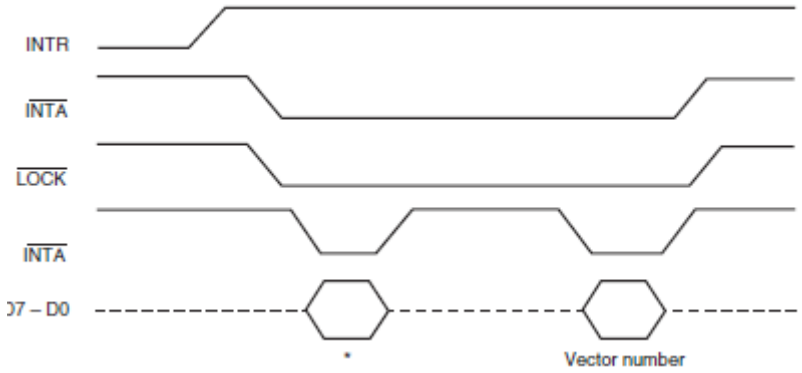


Figure 12-8: The timing of the INTR input and INTA output.

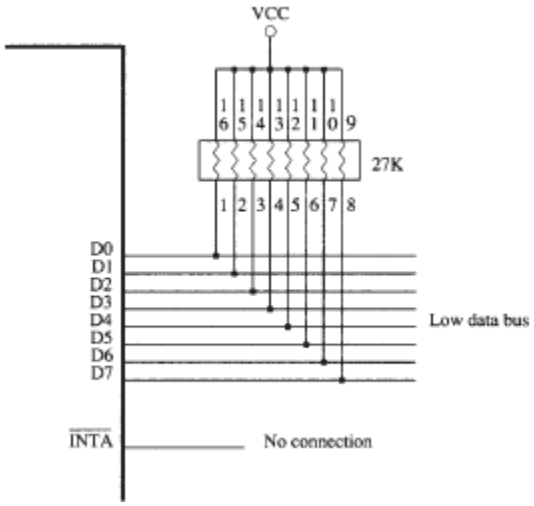


Figure 12-9:A simple method of generating interrupt vector type number FFH in response to INTR

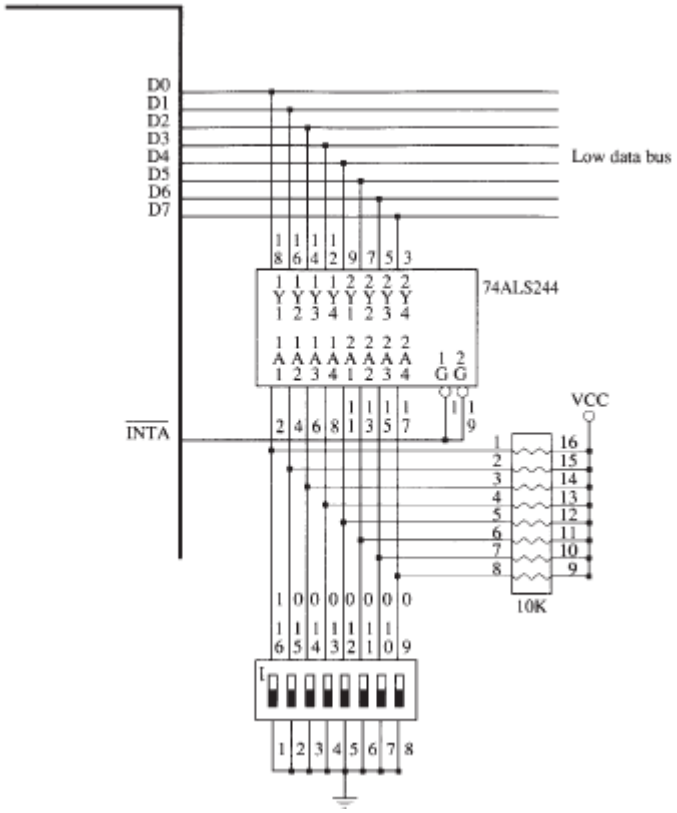


Figure 12-10: A circuit that applies any interrupt vector type number in response to INTA.

UNIT 8(CONT.): DIRECT MEMORY ACCESS

Basic DMA Operation

- Two control signals are used to request & acknowledge a DMA transfer in the microprocessor-based system.
  1) HOLD pin is an input that is used to request a DMA action &
  2) HLDA pin is an output that acknowledges the DMA action.
- Whenever HOLD is placed at logic 1, DMA action(hold) is requested.
- The microprocessor responds
  → by suspending the execution of the program &
  → by placing its address, data and control bus at their high-impedance states
- External I/O device uses the system buses to access the memory directly.
- HOLD has a higher priority than INTR or NMI.
  On the other hand, RESET has a higher priority than NMI.
- Interrupt take effect at the end of an instruction, whereas a HOLD takes effect in the middle of an instruction.

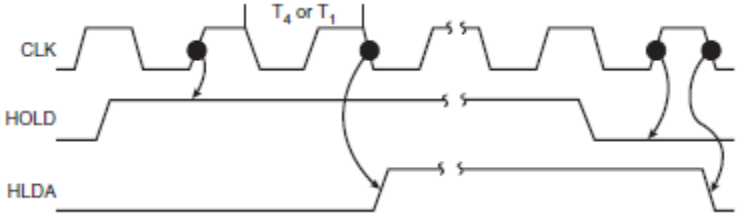


Figure 13-1: HOLD and HLDA timing for the microprocessor

Basic DMA Definitions

- Direct memory accesses occur between an I/O device & memory without the use of microprocessor.
- A DMA read transfers data from memory to I/O device.
  A DMA write transfers data from I/O device to memory.
- In both operations, memory and I/O are controlled simultaneously.
- DMA read causes both $\overline{MRDC}$ & $\overline{IOWC}$ to activate simultaneously,
  transferring data from memory to I/O device.
  DMA write causes both $\overline{MWTC}$ & $\overline{IORC}$ to activate simultaneously,
  transferring data from I/O device to memory.
- These control bus signals are available to all microprocessor in the Intel family except the 8086 system.
- The 8086 require system controller to generate the control bus signals.

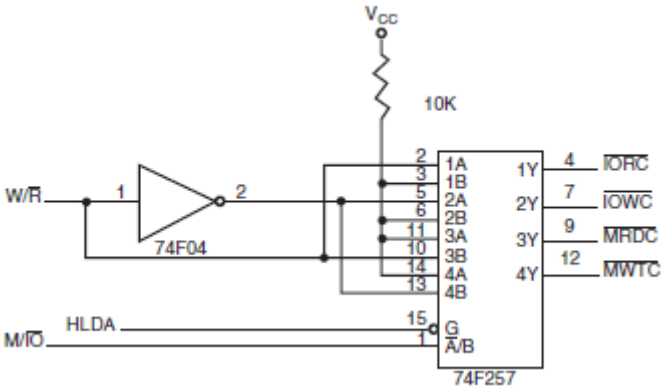


Figure 13-2: A circuit that generates system control signals in a DMA environment