

POWERUNIT

Chapter 1

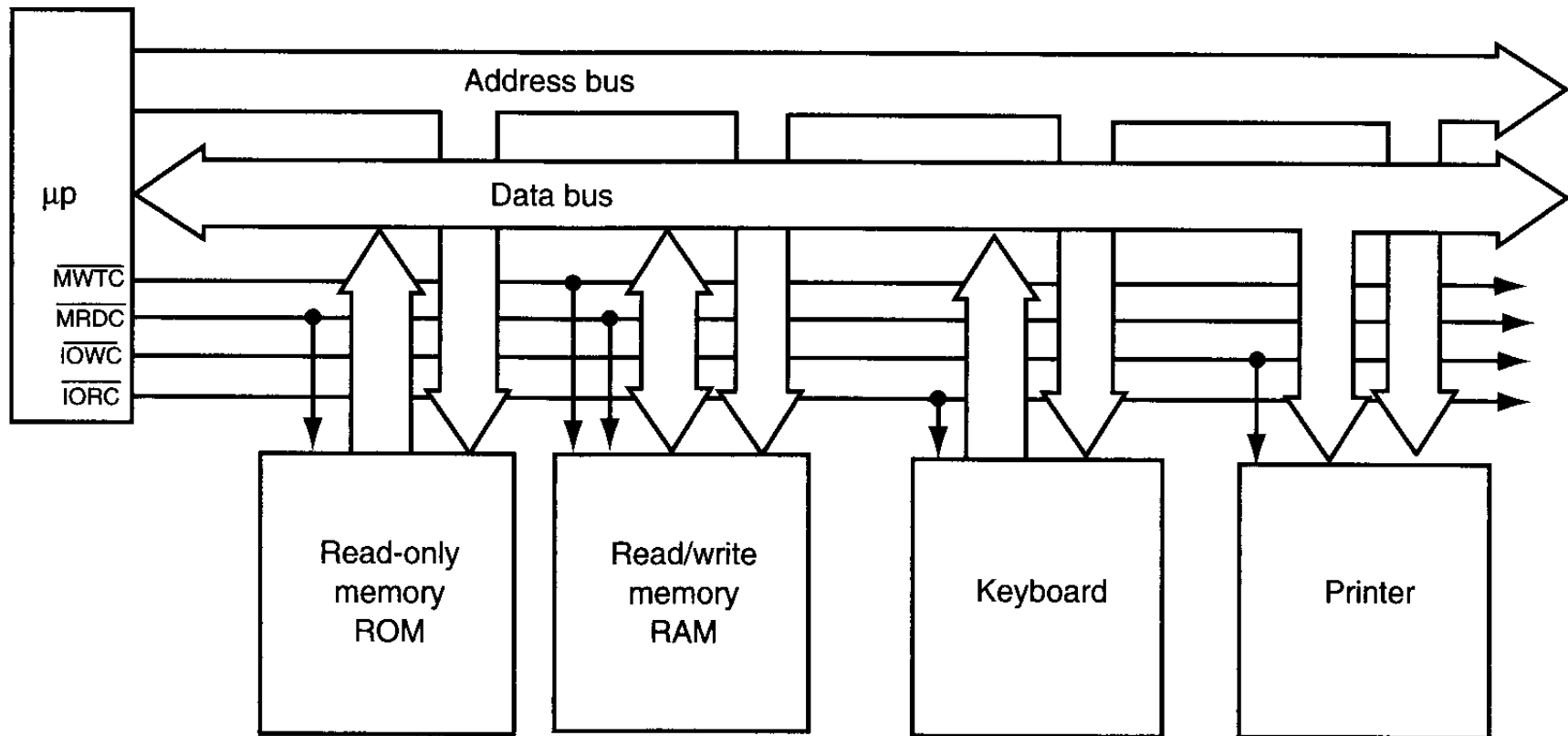
Introduction to Microprocessors

Part I

*Note: These slides are adapted from more than
a site/reference*

**Instructor:
Dr. Khalid A. Darabkh**

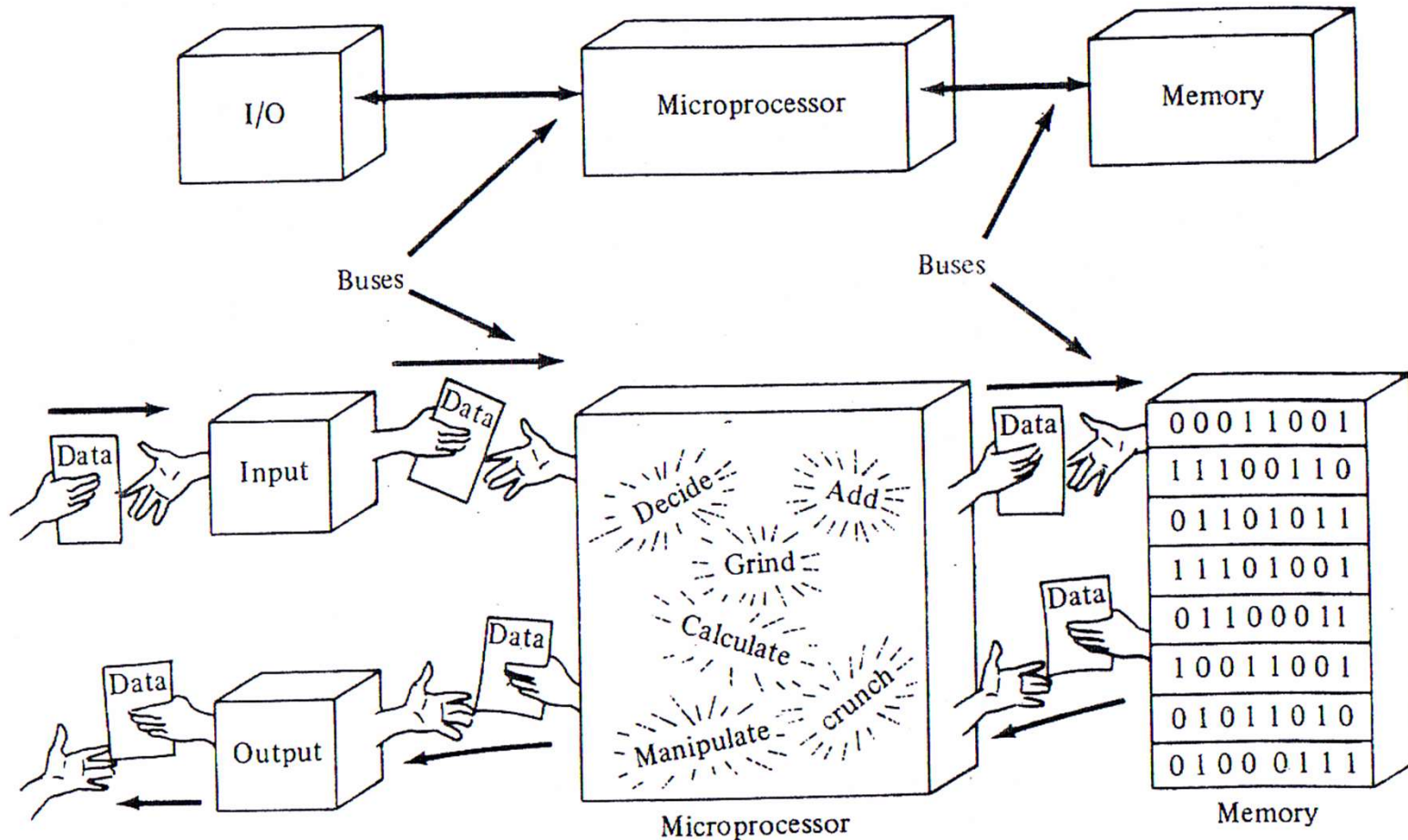
Computer Structure



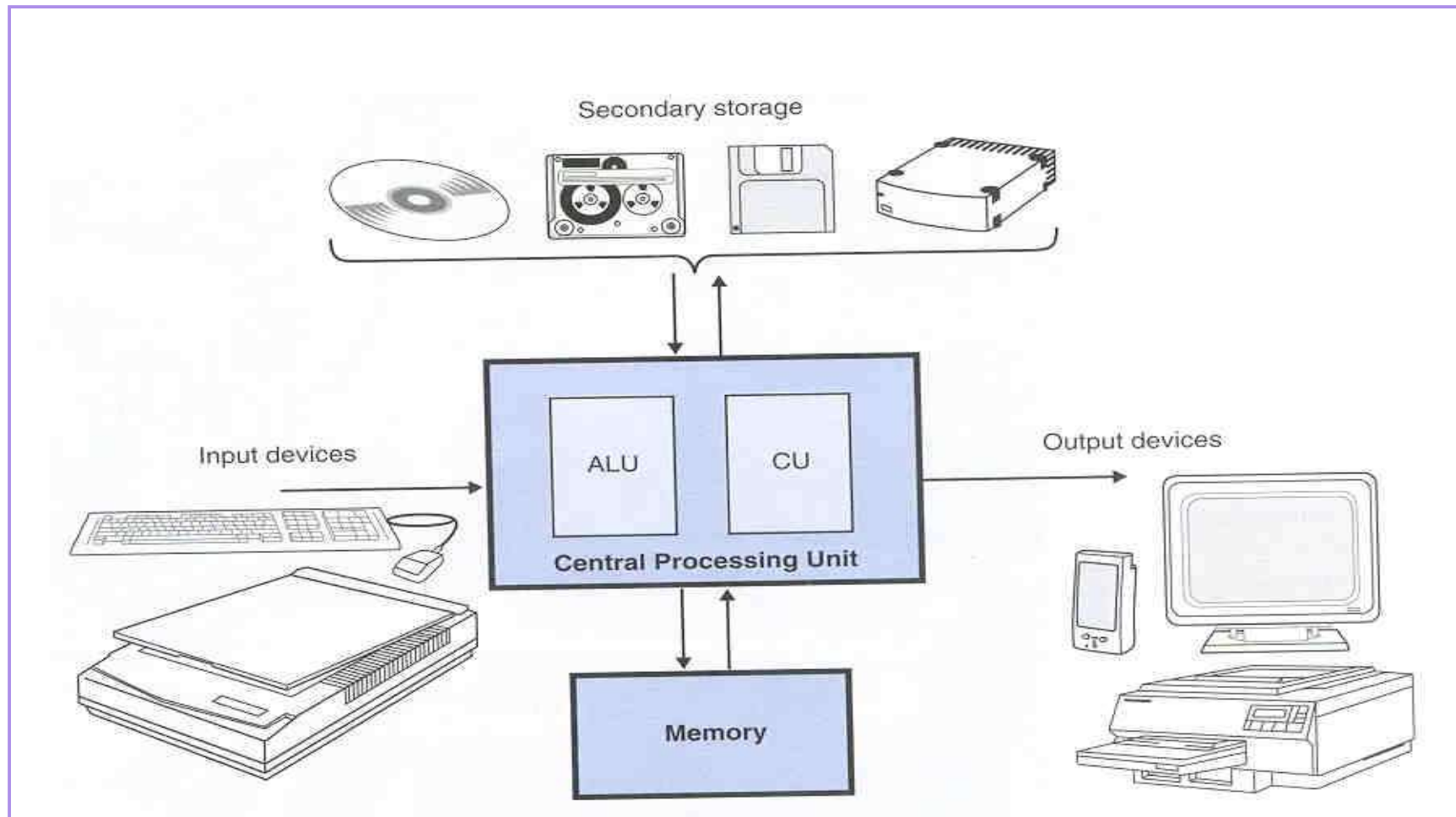
Peripheral device is a computer device, such as a CD-ROM drive or printer, that is not part of the essential computer, i.e., the memory and microprocessor. Peripheral devices can be external -- such as a mouse, keyboard, printer, monitor, external scanner -- or internal, such as a CD-ROM drive, CD-R/W drive

Block Diagram of Microprocessor System

FIGURE 10.1 Block Diagram



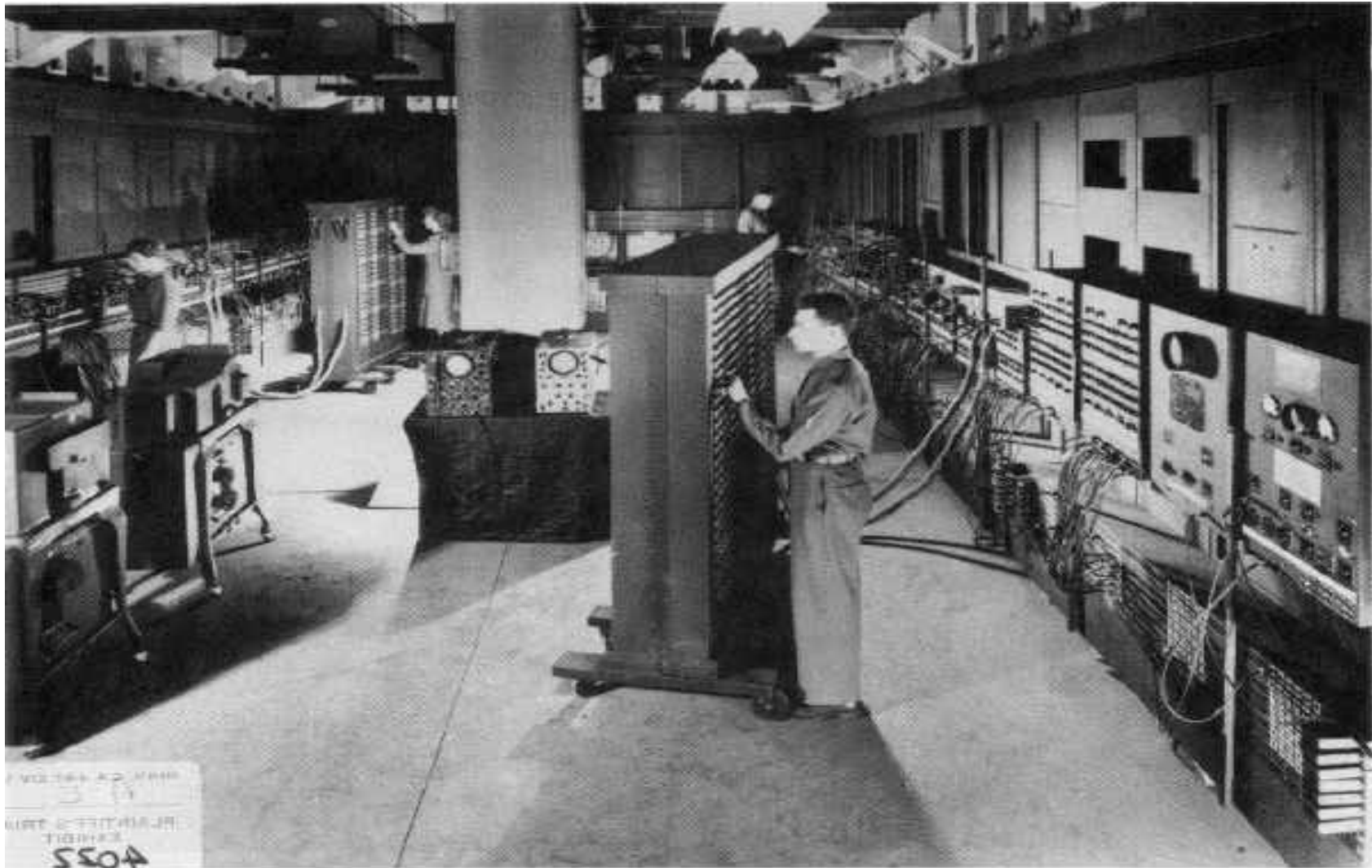
Block Diagram of Microprocessor System Cont.



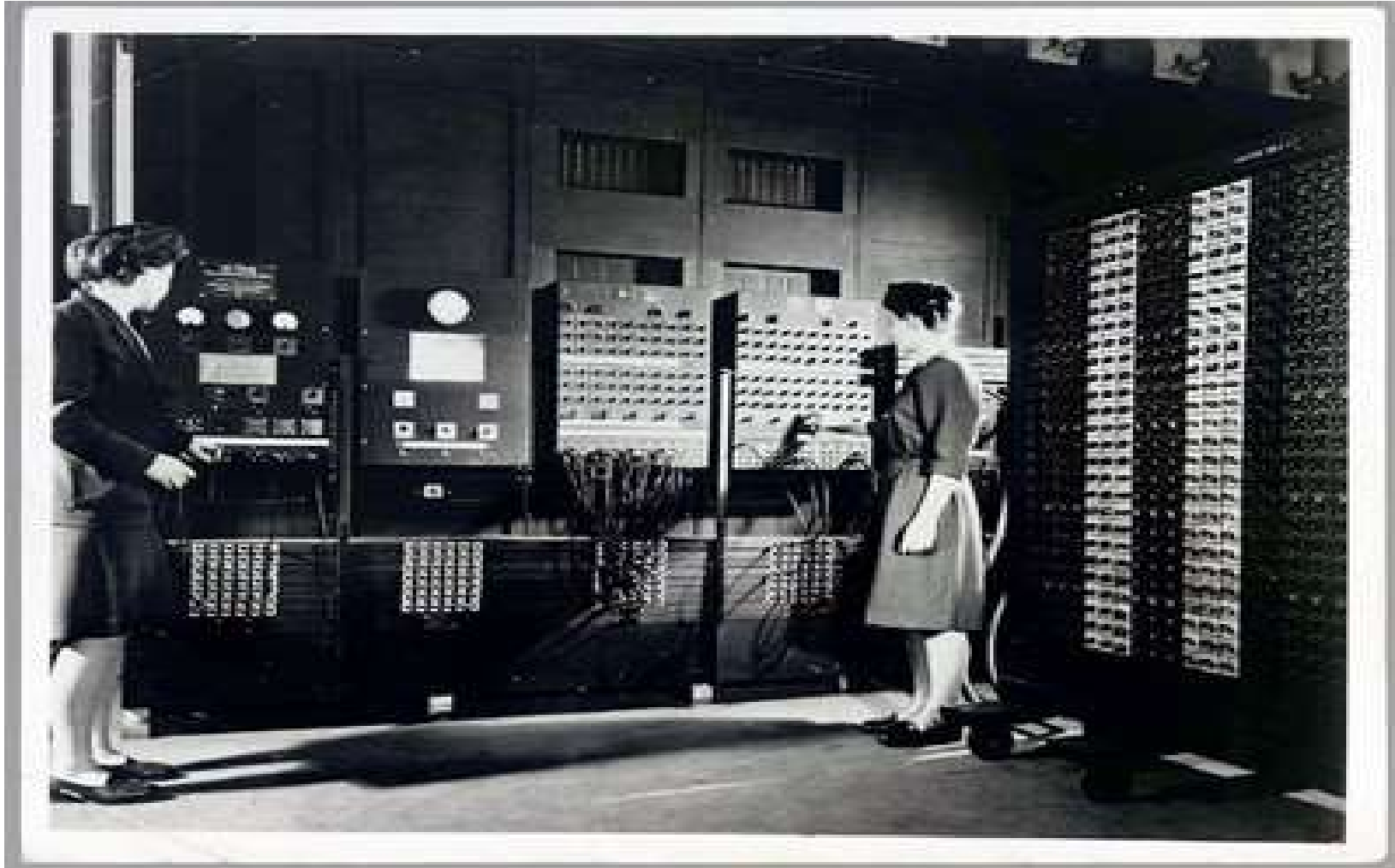
ENIAC - The first electronic computer

- **1945:** The first computer prototype using vacuum tubes, ENIAC (Electronic Numerical Integrator and Computer) was designed to calculate trajectory tables for the U.S. Army during World War II, but it was not completed until three months after the war.

ENIAC - The first electronic computer

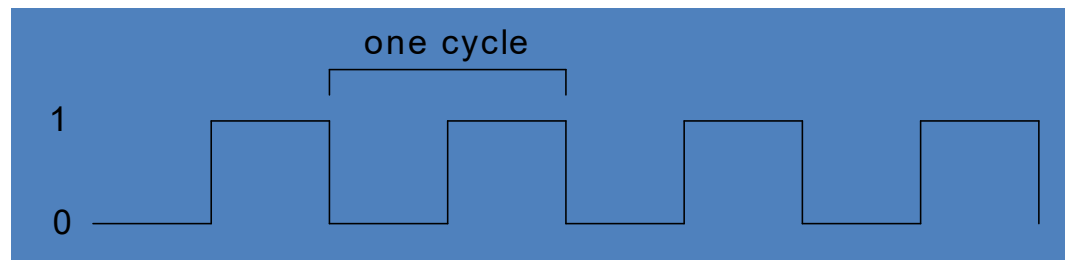


ENIAC - The first electronic computer



Clock

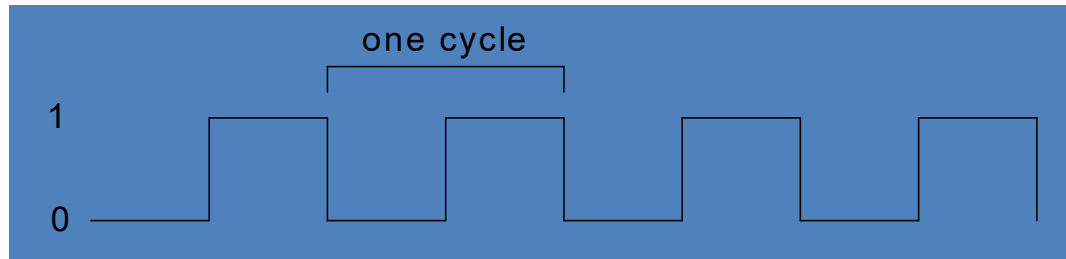
- Synchronizes all CPU and BUS operations
- Machine (clock) cycle measures time of a single operation
- Clock is used to trigger events



Clocking Methodology

- An edge-triggered clocking methodology means that any values stores in the machine are updated (changed) only on a clock edge.
- A level-triggered clocking methodology means that any values stores in the machine are updated only on the high level (level 1 not zero) of the clock cycle.

Clock Cont.



Cycle time = time between ticks = seconds per cycle

Clock rate (frequency) = cycles per second

1 HZ = 1 cycle/sec

1 MHZ = one million cycles per second

A 200 MHz clock has a cycle time = 5 nanoseconds

What is inside Computer

- **The internal working of every computer can be broken down into three parts:**
 - 1. CPU
 - 2. Memory
 - 3. I/O (input / output devices)
- The function of the CPU is to execute (process) information stored in memory.
- The function of the I/O devices such as the keyboard and video monitor is to provide a means of communicating with the CPU. The CPU is connected to memory and I/O through strips of wire called a bus.

What is inside Computer Cont.

- The bus inside a computer carries information from place to place just as a street bus carries people from place to place.
- Components of the computer are connected by Buses.
- A bus in the simplest form is a set of wires that used to carry information in the form of electrical signals between CPU and memory and CPU and I/O
- **Example:** VAX has a 32-bit address bus and a 32-bit data bus

What is inside Computer Cont.

In every computer there are three types of buses:

- 1. Address bus
- 2. Data bus
- 3. Control (Signal) bus
- For a device (memory or I/O) to be recognized by the CPU, it must be assigned an address. The address assigned to a given device must be unique; no two devices are allowed to have the same address.
- The CPU puts the address (of course in binary) on the address bus, and the decoding circuitry finds the device. Then the CPU uses the data bus either to get data from that device or to send data to it.
- The control buses are used to provide read or write signals to the device to indicate if the CPU is asking for information or sending information.

More about data bus:

- Since data buses are used to carry information in and out of a CPU, the more data buses available, the better the CPU.
- More data buses mean a more expensive CPU and computer.
- The average size of data buses in CPUs varies between 8 and 64.
- Data buses are bi-directional, since the CPU must use them either to receive or to send data.
- The processing power of a computer is related to the size of its buses, since an 8-bit bus can send out 1 byte a time, but a 16-bit bus can send out 2 bytes at a time, which is twice as fast.

More about data bus:

- There are two types of data bus :
- Internal data bus (back side bus): which is a bus between CPU and cache
- External data bus (front side bus): which is a bus between cache and memory
- Note: backside and front side buses are always different. The first runs at processor speed (connecting processor with cache) and the other is designed to match lower speed devices like memories and I/Os

More about address bus:

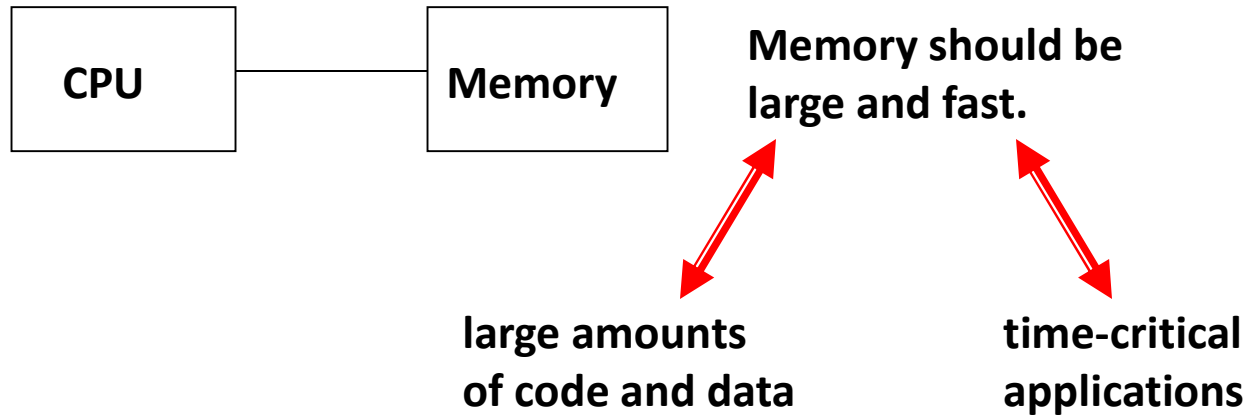
- The address bus is unidirectional bus which means that the CPU uses the address bus only to send out addresses.
- The total number of memory locations addressable by a given CPU is always equal to 2 to the power x , where x is the number of address bits, regardless of the size of the data bus.

Main Memory

- A linear list of memory cells. Each cell holds a data word.
- A word could be a byte (ex. VAX), 2 bytes, 4 bytes.....
- All memories share two organizational features:-
 - 1) Each information unit is the same size.
 - 2) An information unit has a numbered address associated with it by which it can be uniquely referenced.
- A memory cell is characterized by two things:-
 - 1) An address
 - 2) Content

Main Memory (Cont.)

Memory is used to store programs and data.



- Fast memories are expensive, slow memories are cheap.
- A fast and large memory results in an expensive system.

What is inside CPU?

- A program stored in memory provides instructions to the CPU to perform an action. It is the function of the CPU to fetch these instructions from memory and execute them. To perform the actions of fetch and execute, all CPUs are equipped with recourses such as the following:
- 1. Registers, the CPU uses registers to store information temporarily. The information could be two values to be processed, or the address of the value needed to be fetched from memory. Registers inside CPU can be 8-bit, 16-bit, 32-bit, or even 64-bit registers, depends on the CPU.
- **Two kinds of registers exist:**
- **A. Special purpose register: hold program state; they usually include the program counter (instruction pointer), stack pointer, and status register (processor status word).**

Inside CPU (Cont.)

- Note: the function of the program counter is to point to the address of the next instruction to be executed.
- **B. General purpose registers: Can store both data and addresses, i.e., they are combined Data/Address registers.**
- 2. The CPU also has what is called ALSU.
- 3. Control Unit (CU): decodes each machine instruction and sends signal to other components for carrying out the instruction.
- More Explanation about (3): The function of the instruction decoder is to interpret the instruction fetched into the CPU. One can think of the instruction decoder as a kind of dictionary, storing the meaning of each instruction and what steps the CPU should take upon receiving a given instruction.

Internal working of computers

- To demonstrate some of the concepts discussed above, a step-by-step analysis of the process a CPU would go through to add three numbers is given next.
- Assume that an imaginary CPU has registers called A, B, C and D. It has an 8-bit data bus and a 16-bit address bus. Therefore, the CPU can access memory from address 0000 to FFFF.

Action	Code	Data
Move value 21H into register A	B0H	21H
Add value 42H to register A	04H	42H
Add value 12H to register A	04H	12H

Internal working of computers (Cont.)

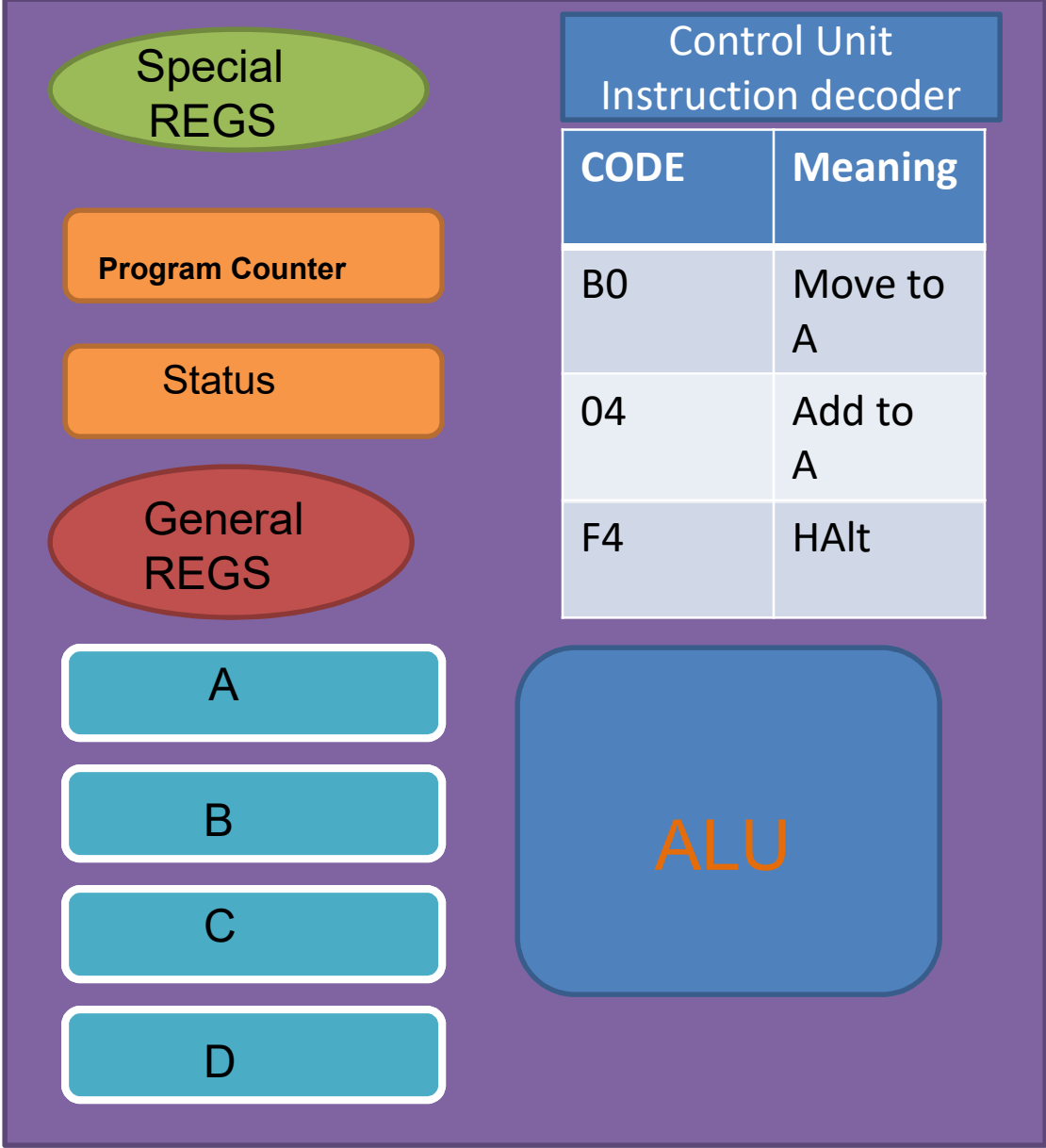
- If the program to perform the actions listed above is stored in memory locations starting at 1400H, the following would represent the contents for each memory address location:

Memory address	Contents of memory address
1400	B0 the code for moving a value to register A
1401	21 the value to be moved
1402	04 the code for adding a value to register A
1403	42 the value to be added
1404	04 the code for adding a value to register A
1405	12 the value to be added
1406	F4 the code for halt

RAM

Offset	Code	Code or data	Inst #
1400	B0	CODE	1
1401	21	DATA	1
1402	04	CODE	2
1403	42	DATA	2
1404	04	CODE	3
1405	21	DATA	3
1406	F4	CODE	4

MICRO PROCESSOR



Control Unit Instruction decoder	
CODE	Meaning
B0	Move to A
04	Add to A
F4	HALt

Internal working of computers (Cont.)

- The actions performed by the CPU to run the program above would be as follows:
- 1. The CPU's program counter can have a value between 0000 and FFFFH. The program counter must be set to the value 1400H, indicating the address of the first instruction code to be executed.
- 2. The CPU puts 1400H on the address bus and sends it out. The memory circuitry finds the location while the CPU activates the READ signal, indicating to memory that it wants the byte at location 1400H. This causes the contents of memory locations 1400H, which is B0, to be put on the data bus and brought into the CPU.

Internal working of computers (Cont.)

- 3. The CPU decodes the instruction B0 with the help of its instruction decoder dictionary. When it finds the definition for that instruction it knows it must bring into register A of the CPU the byte in the next memory location. Therefore, it commands its controller circuitry to do exactly that. When it brings in value 21H from memory location 1401H, it makes sure that the doors of all registers are closed except register A. Therefore, when value 21H comes into CPU it will go directly into register A. After completing one instruction, the program counter points to the address of the next instruction to be executed, which in this case is 1402H. Address 1402 is sent out on the address bus to fetch the next instruction.

Internal working of computers (Cont.)

- 4. From memory location 1402H it fetches code 04H. After decoding, the CPU knows that it must add to the contents of register A the byte sitting at the next address (1403). After it brings the value (in this case 42H) into the CPU, it provides the contents of register A along with this value to the ALU to perform the addition. It then takes the result of the addition from the ALU's output and puts it in register A. Meanwhile the program counter becomes 1404, the address of the next instruction.

Internal working of computers (Cont.)

- 5. Address 1404H is put on the address bus and the code is fetched into the CPU, decoded, and executed. This code is again adding a value to register A. the program counter is updated to 1406H.
- 6. Finally, the contents of address 1406 are fetched in and executed. This HALT instruction tells the CPU to stop incrementing the program counter and asking for the next instruction. In the absence of the HALT, the CPU would continue updating the program counter and fetching instructions.

Major Categories of Programming Languages

- **Machine Language** (First-generation Language)
- Most basic level of programming languages. Uses binary coded instruction.

- **Assembly Language** (Second-generation Language)
- They are sometimes called symbolic languages, because symbols are used to represent operation codes and storage locations.

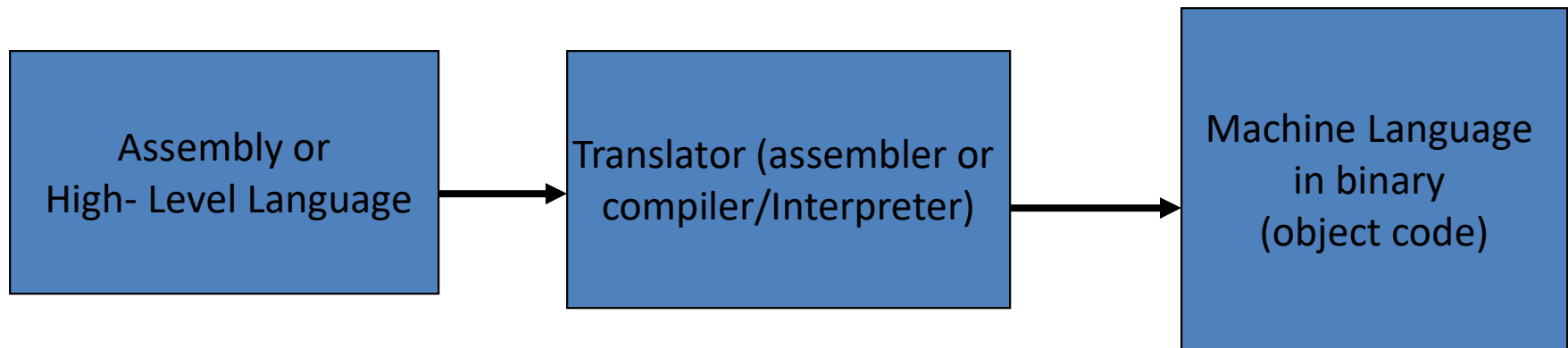
Major Categories of Programming Languages (Cont.)

- **Assembly Language:**
- 1. Is a symbolic code that allows mnemonics for machine language instructions and symbolic names for memory locations?
- Example:
- Instead of using “1100 000”, we write the symbol ADD to mean the same thing.
- 2. It is a programming language in which instructions correspond closely to the individual machine instructions that are carried by a particular computer. **So assembly language, like machine language, is machine dependent**

Programming Languages (Cont.)

- 3. Assembly language uses symbols to represent the operation codes while a special notation is used to represent the different addressing modes.
- 4. A translator is used to translate the assembly code into machine code. This translator is called an **assembler**.
- **High-Level Languages** (Third-generation Languages)
- Use instructions called statements that closely resemble human language or standard notation of mathematics.
- **Fourth-generation Languages:**
- A variety of programming languages that is more non-procedural and conversational. The programmers specify the result they want, while the computer determines the sequence of instructions that will accomplish these results. **Example: SQL**

Programming Languages (Cont.)



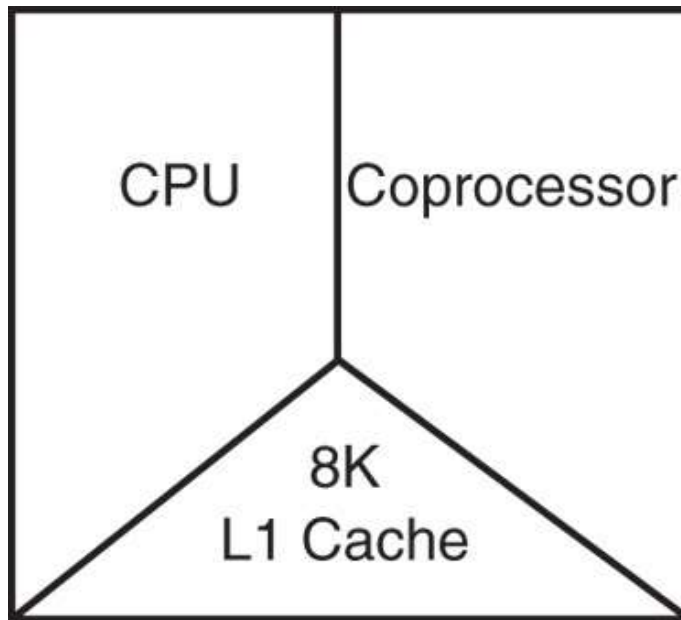
CHAPTER 1

TABLE 1–2 Many modern Intel and Motorola microprocessors.

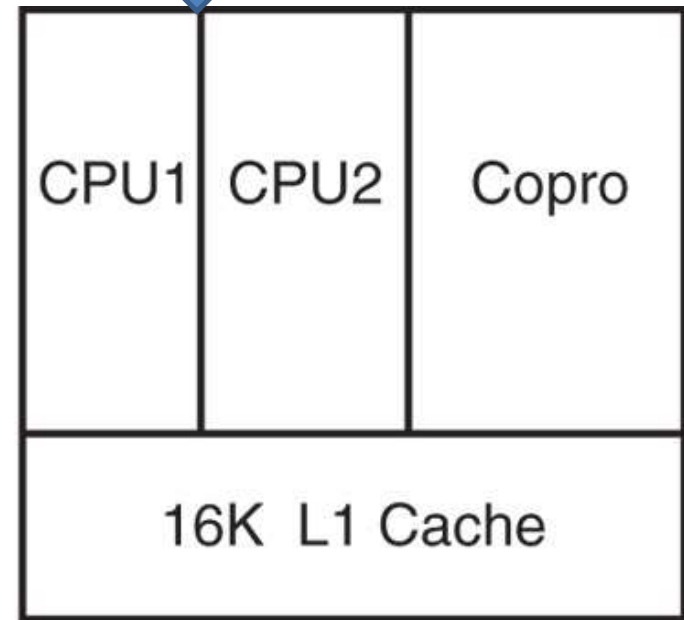
<i>Manufacturer</i>	<i>Part Number</i>	<i>Data Bus Width</i>	<i>Memory Size</i>
Intel	8048	8	2K internal
	8051	8	8K internal
	8085A	8	64K
	8086	16	1M
	8088	8	1M
	8096	16	8K internal
	80186	16	1M
	80188	8	1M
	80251	8	16K internal
	80286	16	16M
	80386EX	16	64M
	80386DX	32	4G
	80386SL	16	32M
	80386SLC	16	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	
Motorola	6800	8	64K
	6805	8	2K
	6809	8	64K
	68000	16	16M
	68008D	8	4M
	68008Q	8	1M
	68010	16	16M
	68020	32	4G
	68030	32	4G + 256 cache
	68040	32	4G + 8K cache
	68050	32	Proposed, but never released
	68060	64	4G + 16K cache
	PowerPC	64	4G + 32K cache

Conceptual views of the 80486, Pentium microprocessors.

Two execution engines.
It can execute up to two
instructions simultaneously.



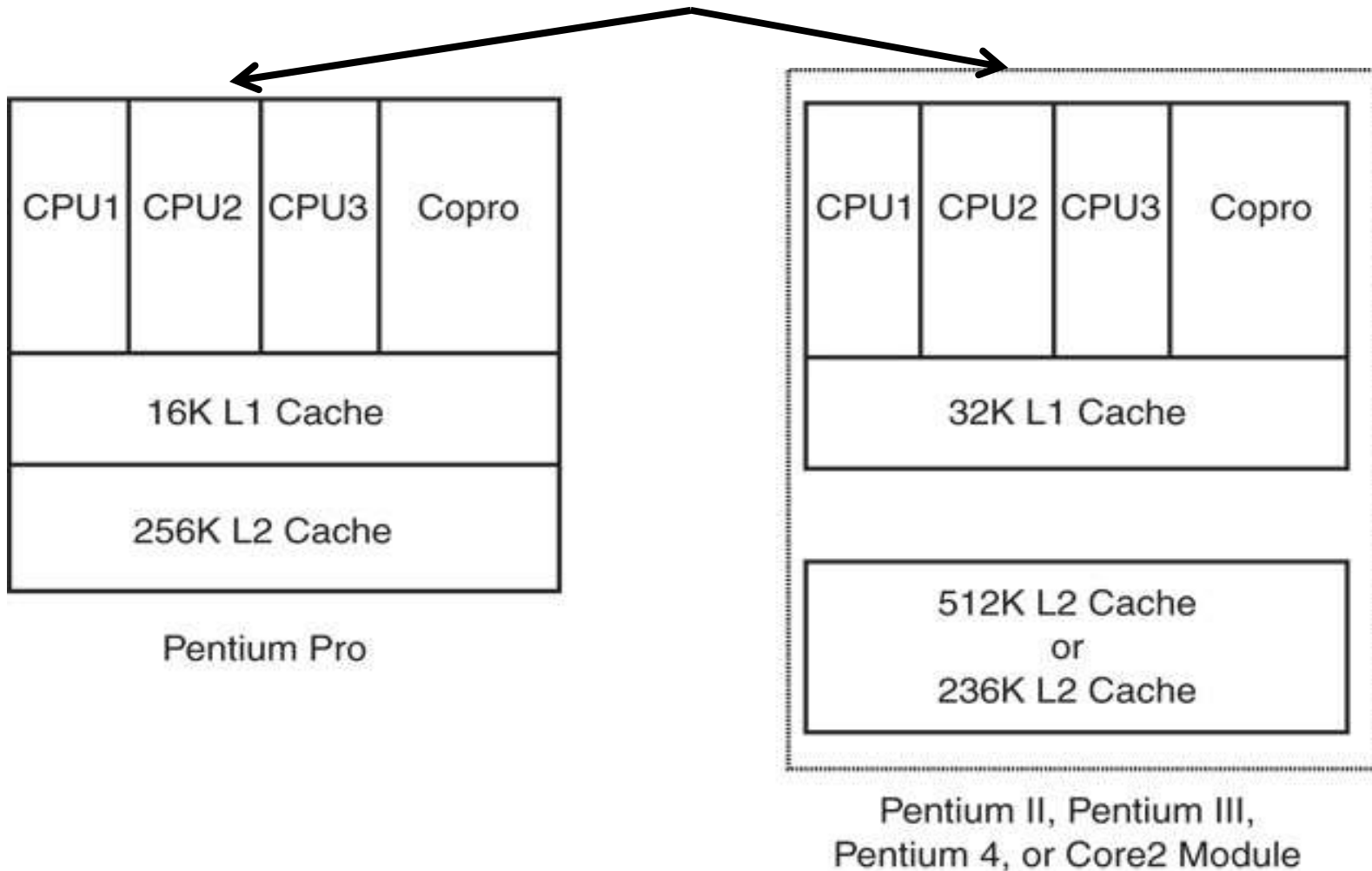
80486DX



Pentium

Conceptual views of the Pentium Pro, Pentium II, Pentium III, Pentium 4, and Core2 microprocessors.

Three execution engines. It can execute up to three instructions at a time.

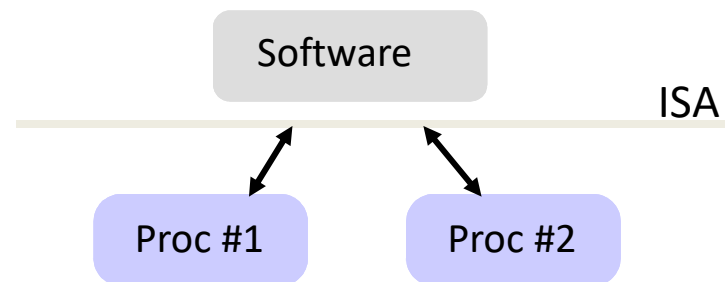


Structure of Pentium Microprocessors

- Beginning with the **80486**, the microprocessor contained a **numeric coprocessor** that allowed it to perform complex arithmetic using **floating-point arithmetic**. The numeric coprocessor, which is similar to a calculator chip, was an additional component in the 8086- through the 80386-based personal computers.
- **Pentium Pro** uses **three execution engines**, to execute up to three instructions at a time.
- Recent modifications to Pentium 4 and Core2 include a 64-bit core and multiple cores.
- 64-bit modification allows address of over 4G bytes of memory through a 64-bit address
- Biggest advancement is inclusion of multiple cores.
 - each core executes a separate task in a program
- Increases speed of execution if program is written to take advantage of multiple cores.
 - called **multithreaded** applications
- Intel manufactures dual and quad core versions; number of cores will likely increase to eight or even sixteen.

Instruction Set Architecture (ISA)

- The ISA is the interface between hardware and software.
- The ISA serves as an **abstraction layer** between the HW and SW
 - Software doesn't need to know how the processor is implemented
 - Any processor that implements the ISA appears equivalent



- An ISA enables processor innovation without changing software
 - This is how Intel has made billions of dollars.
- Before ISAs, software was re-written/re-compiled for each new machine.

RISC vs. CISC

- RISC = Reduced Instruction Set Computer
 - Term coined at Berkeley, ideas pioneered by IBM, Berkeley, Stanford
- RISC characteristics:
 - Load-store architecture
 - Fixed-length instructions (typically 32 bits)
 - Simple operations
- RISC examples: MIPS, SPARC, IBM/Motorola PowerPC, Compaq Alpha, ARM, SH4, HP-PA, ...

- CISC = Complex Instruction Set Computer
 - Term referred to non-RISC architectures
- CISC characteristics:
 - Register-memory architecture
 - Variable-length instructions
 - Complex operations
- CISC examples: Intel 80x86, VAX, IBM 360, ...

What is Superscalar?

- **What is Pipelining? A way of speeding up execution of instructions. Key idea: overlap execution of multiple instructions**
- The term **superscalar** refers to a machine that designed to improve the performance of the execution of scalar instructions.
- There are multiple independent instruction pipelines in a superscalar processor.
- Each pipeline consists of multiple stages, can handle multiple instructions at a time.
- Multiple pipelines introduce a new level of parallelism, enabling multiple streams of instructions to be processed at a time.

What is Superscalar? (Cont.)

- A **superscalar** CPU architecture implements a form of parallelism called instruction-level parallelism within a single processor.
- It thereby allows faster CPU throughput than would otherwise be possible at the same clock rate.
- Attempts to find nearby instructions that are independent of one another and can be executed in parallel.
- A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor. Each functional unit is not a separate CPU core but an execution resource within a single CPU such as an arithmetic logic unit, a bit shifter, or a multiplier.

Alternative Approach: VLIW

- **Alternative Approach: VLIW (Very Long Instruction Word):**
- VLIW architectures rely on compile-time detection of parallelism. The compiler analyzes the program and detects operations to be executed in parallel; such operations are packed into one “large” instruction.
- No hardware is needed for run-time detection of parallelism like the case of superscalar processor.

The microprocessor-Based Personal Computer System

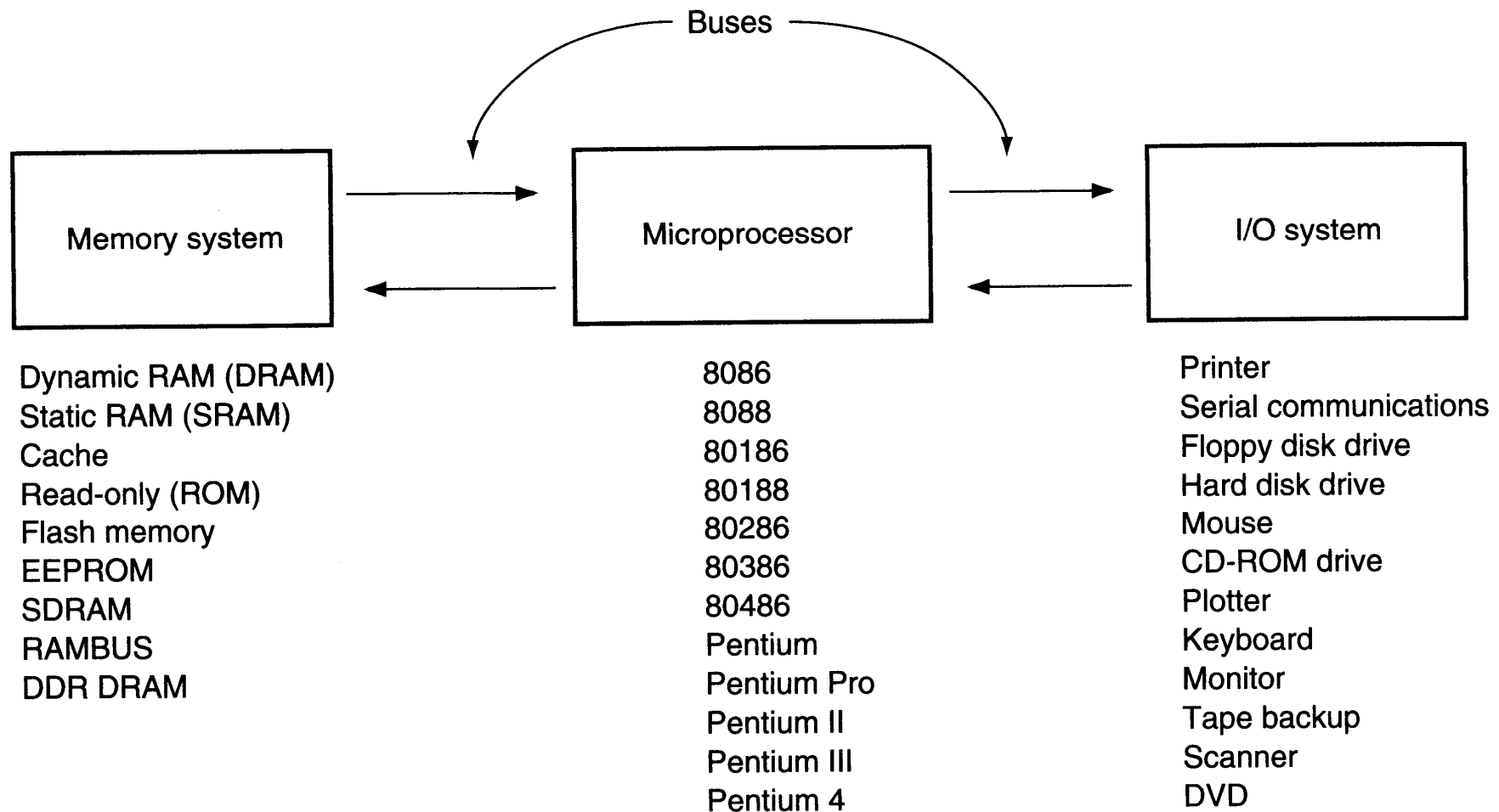


FIGURE 1-6 The block diagram of a microprocessor-based computer system.

What are I/O Devices ?

I/O Devices (Peripherals) : They are The Part of a computer that communication with the outside world .

Their Main Characteristics are :

1. Behavior: (Input , Output or Storage).
2. Partner : Either Human or Machine.
3. Data Rate : The peak rate at which data can be transferred between I/O device and the main memory.

What are I/O Devices? Cont.

Device	Behavior	Partner	Data Rate (KB/sec)
Keyboard	Input	Human	0.01
Mouse	Input	Human	0.02
Voice Input	Input	Human	0.02
Voice Output	Output	Human	0.60
Modem	Input / Output	Machine	2.00 – 8.00
Floppy Disk	Storage	Machine	100.00
Magnetic disk	Storage	Machine	2000.0-10,000.0

Chapter 1

Introduction to Microprocessors Part II

*Note: These slides are adapted from more than
a site/reference*

**Instructor:
Dr. Khalid A. Darabkeh**

The Memory and I/O System

- The memory structure of all Intel-based personal computers are similar
- The memory system is divided into three main parts:
 - **TPA** Transient program area (640 K bytes)
 - **System area** (384 K bytes)
 - **XMS** Extended memory system.

The type of microprocessor determines whether an extended memory system exists

INTRODUCTION TO THE MICROPROCESSOR AND COMPUTER

FIGURE 1-7 The memory map of the personal computer.

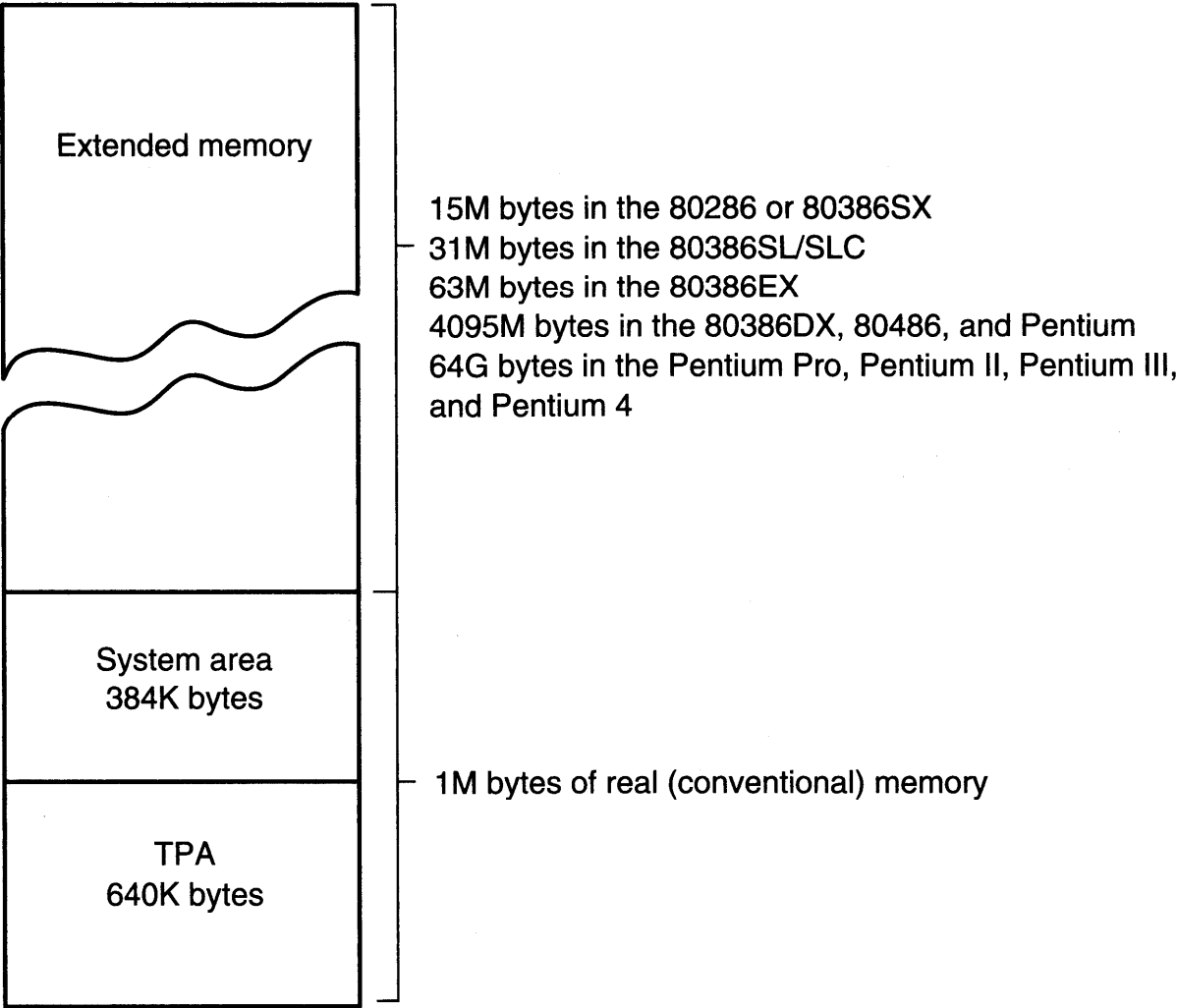
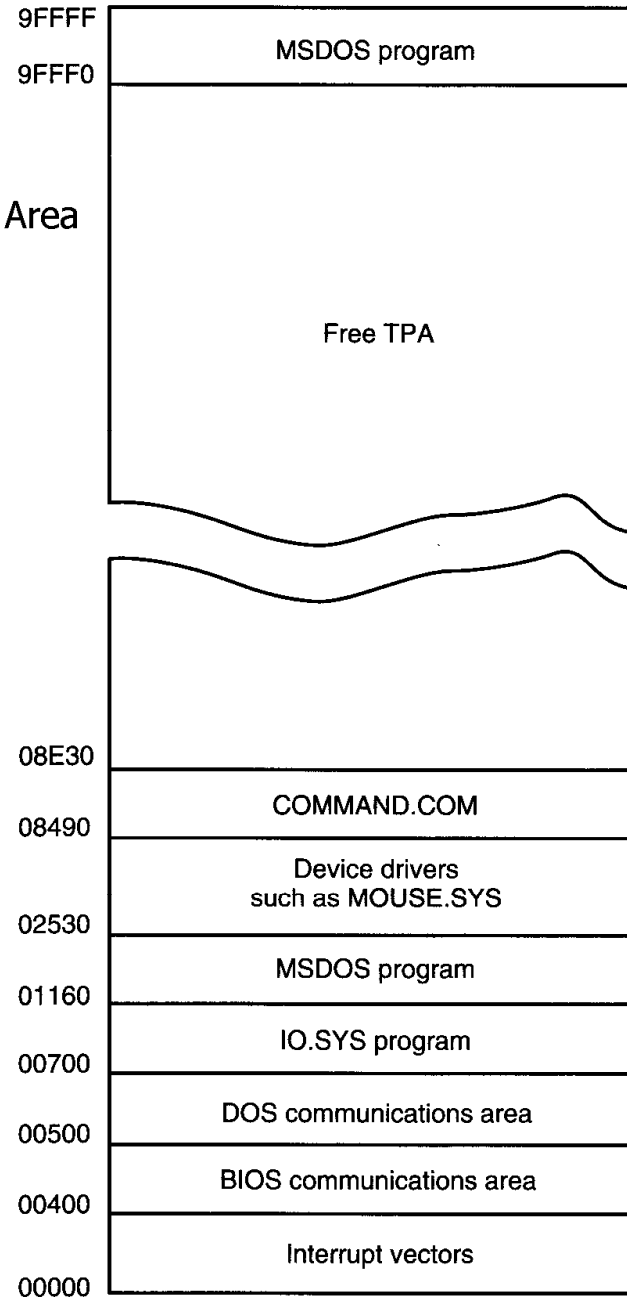


FIGURE 1-8 The memory map of the TPA in a personal computer. (Note that this map will vary between systems.)

TPA Transient Programming Area



How does the O/S stop program X?

Answer: It issues an *interrupt*.

The TPA

- The transient program area (TPA) holds the DOS (Disk Operating System) and other programs that control the computer system
- **BIOS** (Basic I/O System) is a collection of programs stored in either a ROM or flash memory that operates many of the I/O devices connected to your computer.
- The **IO.SYS** is a program that loads into the TPA from the disk whenever **an MSDOS system is started**. The IO.SYS contains programs that allow DOS to use the keyboard, Video display, printer, and other I/O devices often found in the computer system.
- The IO.SYS program links DOS to the programs stored in the system BIOS ROM found in the system area.
- A **device driver** is a program that controls a particular type of device that is attached to your computer. There are device drivers for printers, displays, CD-ROM readers, diskette drives, and so on. When you buy an operating system, many device drivers are built into the product. However, if you later buy a new type of device that the operating system didn't anticipate, you'll have to install the new device driver. A device driver essentially converts the more general input/output instructions of the operating system to messages that the device type can understand.
- **DOS Device drivers** are normally files that have an extension of .SYS, such as **MOUSE.SYS**.
- In DOS version 3.2 and later, the files have an extension of .EXE, such as EMM386.EXE. Note that even though these files are not used by Windows, they are still used to execute DOS applications, even with Windows XP.

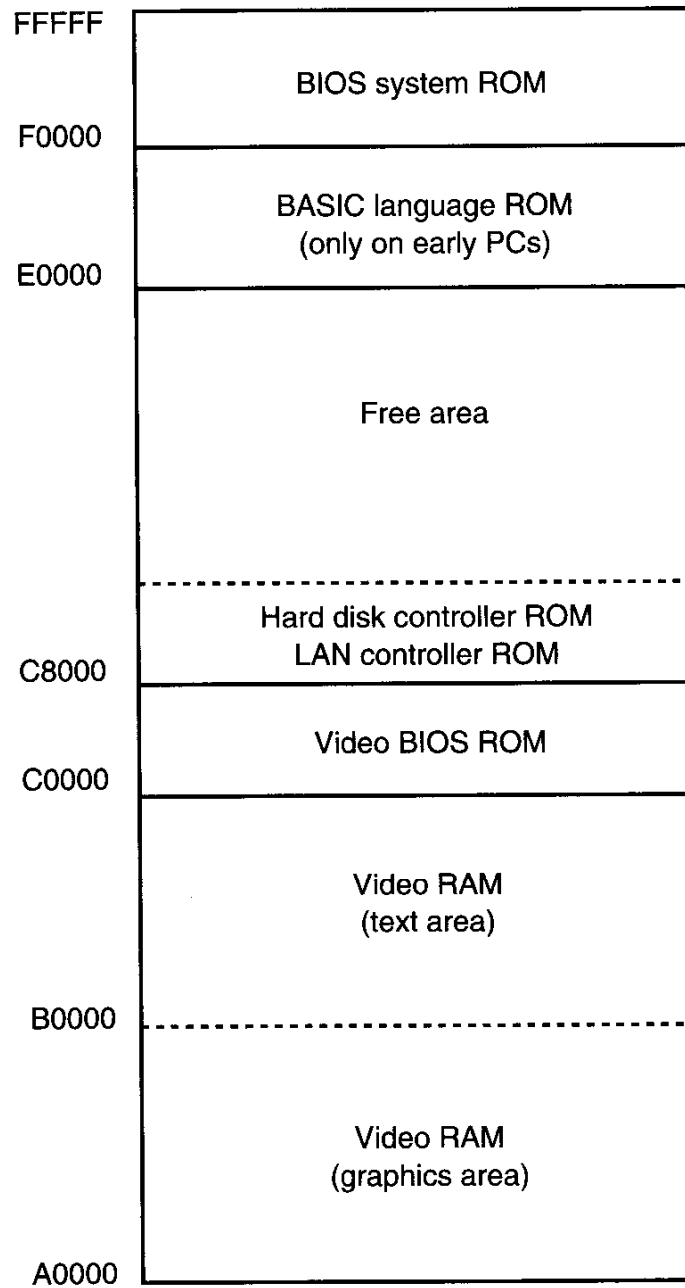
The TPA Cont.

- The **COMMAND.COM** program (command processor) controls the operation of the computer from the keyboard when operated in the DOS mode. The COMMAND.COM program processes the DOS commands as they are typed from the keyboard. For example, if DIR is typed, the COMMAND.COM program displays a directory of the disk files in the current disk directory.
- If COMMAND.COM is erased, the computer cannot be used from the keyboard in DOS mode.
- Windows uses a file called **SYSTEM.INI** to load drivers used by Windows.
- In Windows XP and newer, a registry is added to contain information about the system and drivers used.
- You can view the **registry** with the **REGEDIT** program.
- **Note:** The system BIOS and DOS communication areas contain transient data used by programs to access I/O devices and the internal features of the computer system.

The System Area

- The system area contains programs in either ROM or Flash memory, and areas for read/write (RAM) memory for data storage.
- The first area of the system space contains video display RAM and video control programs on ROM or flash memory.
- The size and amount of memory used depends on the type of the video display adapter.
- A **video card**, also known as a **graphics accelerator card**, **display adapter**, or **graphics card**. Your system's video card is the component responsible for producing the visual output from your computer. Virtually all programs produce visual output; the video card is the piece of hardware that takes that output and tells the monitor which of the dots on the screen to light up (and in what color) to allow you to see it.
- The video BIOS, located on a ROM or flash memory, contains programs that control the DOS video display.
- **System BIOS ROM is also located in the top of the system area.** This ROM controls the operation of the basic I/O devices connected to the computer system. It does not control the operation of the video system, which has its own BIOS ROM.

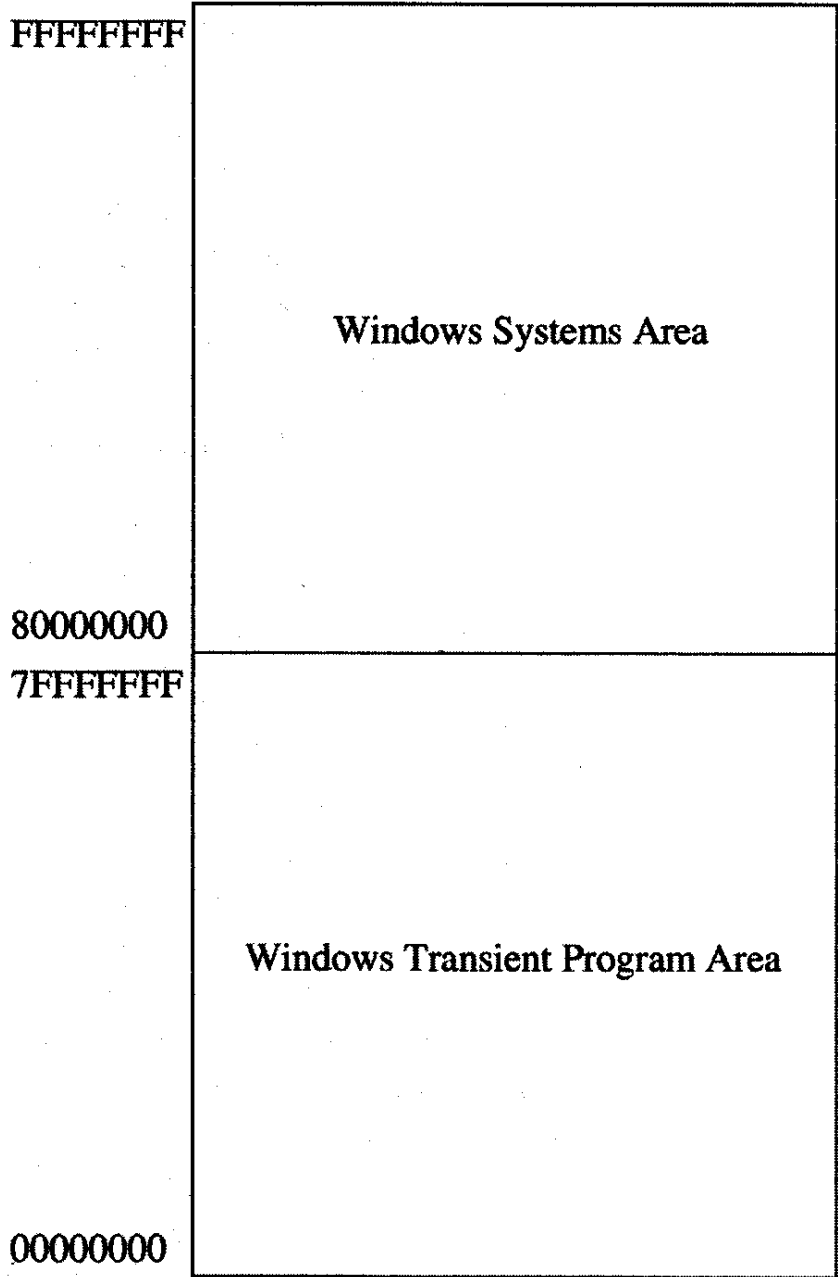
FIGURE 1-9 The system area of a typical personal computer.



Windows System

- The Windows memory map appears in Figure 1-10 and has two main areas, a TPA and a system area. The difference between it and the DOS memory map are the sizes and locations of these areas.
- The Windows **TPA** is the first 2G bytes from **00000000H to 7FFFFFFFH**
- The **windows system area** is the last 2G bytes of memory from **80000000H to FFFFFFFFH**

FIGURE 1-10 The memory map used by Windows XP.

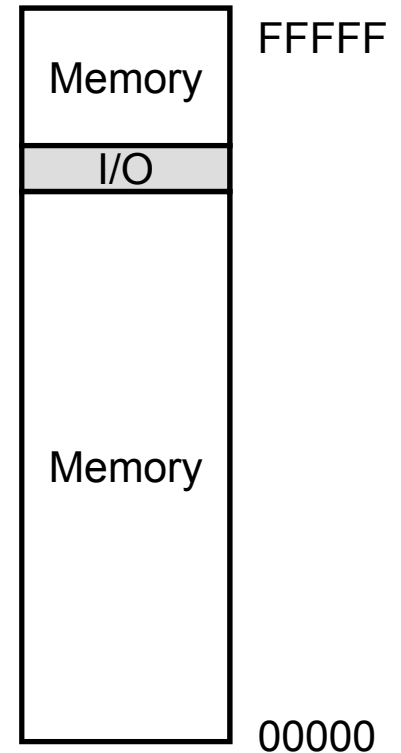


I/O Space

- ***I/O Space:*** The I/O (input/output) space in a computer system extends from I/O port 0000H to port FFFFH. (An I/O **port address** 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 allows the computer to access up to **64K** different **8-bit** I/O devices, 32K different 16-bit devices. or 16K different 32-bit devices. A great number of these locations are available for expansion in most computer systems.

Memory-mapped I/O

- With **memory-mapped I/O**, one address space is divided into two parts.
 - Some addresses refer to physical memory locations.
 - Other addresses actually reference peripherals.



The Microprocessor

- The microprocessor (**CPU** Central Processing Unit), is the controlling element in a computer system.
- The microprocessor **controls memory and I/O** through a series of connections called buses.
- The **buses select an I/O and memory device**, transfer data between an I/O device or memory and microprocessor, and control the I/O and memory system
- **Memory and I/O are controlled through instructions that are stored in the memory and executed by the microprocessor**

- The microprocessor perform **three main tasks** for the computer system:

1- **data transfer**

2- simple **arithmetic and logic operations**

3- **program flow via simple decision.**

The power of the microprocessor is its capability to **execute billions of millions of instructions per second**

Tables 1-4 and 1-5 shows the arithmetic and logic operations for Intel family of microprocessor.

TABLE 1-4 Simple arithmetic and logic operations.

<i>Operation</i>	<i>Comment</i>
Addition	
Subtraction	
Multiplication	
Division	
AND	Logical multiplication
OR	Logic addition
NOT	Logical inversion
NEG	Arithmetic inversion
Shift	
Rotate	

TABLE 1-5 Decisions found in the 8086–Pentium 4 microprocessors.

<i>Decision</i>	<i>Comment</i>
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 ones
Overflow	Test for an overflow that indicates an invalid result after a signed addition or a signed subtraction

Two's Comp. Subtraction

- To subtract two's complement numbers we first negate the second number and then add the corresponding bits of both numbers.

- For example:

$$\begin{array}{cccc}
 3 = 0011 & -3 = 1101 & -3 = 1101 & 3 = 0011 \\
 -2 = 0010 & --2 = 1110 & -2 = 0010 & --2 = 1110 \\
 \hline
 \end{array}$$

become

$$\begin{array}{cccc}
 3 = 0011 & -3 = 1101 & -3 = 1101 & 3 = 0011 \\
 + -2 = 1110 & + 2 = 0010 & + -2 = 1110 & + 2 = 0010 \\
 \hline
 1 = 0001 & -1 = 1111 & -5 = 1011 & 5 = 0101
 \end{array}$$

Overflow

- When adding or subtracting numbers, the sum or difference can go beyond the range of representable numbers.
- This is known as overflow. For example, for two's complement numbers,

$$\begin{array}{cccc}
 5 = 0101 & -5 = 1011 & 5 = 0101 & -5 = 1011 \\
 + 6 = 0110 & + -6 = 1010 & - -6 = 1010 & - +6 = 0110 \\
 \hline
 -5 = 1011 & 5 = 0101 & -5 = 1011 & 5 = 0101
 \end{array}$$

- Overflow creates an incorrect result that should be detected.

2's Comp - Detecting Overflow

- When adding two's complement numbers, overflow will only occur if
 - the numbers being added have the same sign
 - the sign of the result is different
- If we perform the addition

$$\begin{array}{r} a_{n-1} a_{n-2} \dots a_1 a_0 \\ + b_{n-1} b_{n-2} \dots b_1 b_0 \\ \hline = s_{n-1} s_{n-2} \dots s_1 s_0 \end{array}$$

- Overflow can be detected as

$$V = a_{n-1} \cdot b_{n-1} \cdot \overline{s_{n-1}} + \overline{a_{n-1}} \cdot \overline{b_{n-1}} \cdot s_{n-1}$$
- Overflow can also be detected as

$$V = c_n \oplus c_{n-1}$$

where c_{n-1} and c_n are the carry in and carry out of the most significant bit.

Unsigned - Detecting Overflow

- For unsigned numbers, overflow occurs if there is carry out of the most significant bit.

$$V = c_n$$

- For example,

$$\begin{array}{r} 1001 = 9 \\ +1000 = 8 \\ \hline =0001 = 1 \end{array}$$
- With the MIPS architecture
 - Overflow exceptions occur for two's complement arithmetic
 - » add, sub, addi
 - Overflow exceptions do not occur for unsigned arithmetic
 - » addu, subu, addiu

Buses Revisited

- A common group of wires that interconnect components in a computer system is called a Bus.
- Three buses exist for this transfer of information: address, data, and control.
- **Address Bus** – selects a location in the memory or a specific I/O device
- **Data Bus** – transfers data between the microprocessor and the memory or I/O
- **Control Bus** – selects I/O or memory and causes a read or a write
- Figure 1–12 shows how these buses interconnect various system components

Control BUS

- The control bus contains lines that select the memory or I/O and cause them to perform a read or write operation. In most computer systems, there are four control bus connections: *MRDC* (memory read control), *MWTC* (memory write control), *IORC* (I/O read control), and *IOWC* (I/O write control).
- Note that the overbar indicates that the control signal is active-low; that is, it is active when a logic zero appears on the control line. For example, *IOWC* = 0, the microprocessor is writing data from the data bus to an I/O device whose address appears on the address bus.

Unicode

- Many Windows-based applications use the Unicode system to store alphanumeric data.
- This system stores each character as a 16-bit data.
- The code 0000H to 00FFH are the same as standard ASCII code.
- The remaining codes, 0100H-FFFFH, are used to store all special characters from many worldwide character sets.

EXAMPLE 1-18

```
0000  42 61 72 72 79  NAMES  DB  'Barry B. Brey'  
      20 42 2E 20 42  
      72 65 79  
000D  57 68 65 72 65  MESS   DB  'Where can it be?'  
      20 63 61 6E 20  
      69 74 20 62 65  
      3F  
001D  57 68 61 74 20  WHAT   DB  'What is on first.'  
      69 73 20 6F 6E  
      20 66 69 72 73  
      74 2E
```

EXAMPLE 1-19

```
CString NAMES = "Barry B. Brey"  
  
CString MESS = "Where can it be?"  
  
CString WHAT = "What is on first."
```

If an earlier version of C++ is used, then the string is defined with a CString for MS Visual C++ instead of a String[^]. The [^] symbol indicates that String is a member of the garbage collection heap for managing the storage.

BCD (Binary-Coded Decimal) Data

- **BCD** is stored in either **Packed** (two digits per byte) or **unpacked** (one digit per byte)
- **BCD** are not suitable for complex arithmetic
- The range extends from 0-9 decimal.

TABLE 1-10 Packed and unpacked BCD data.

<i>Decimal</i>	<i>Packed</i>		<i>Unpacked</i>		
12	0001 0010		0000 0001	0000 0010	
623	0000 0110	0010 0011	0000 0110	0000 0010	0000 0011
910	0000 1001	0001 0000	0000 1001	0000 0001	0000 0000

- **Example 1-20 shows how to use assembler to define both packed and unpacked BCD data. Example 1-21 shows how to do this using Visual C++ and char or bytes.**

EXAMPLE 1-20

```

                                ;Unpacked BCD data (least-significant data first)
                                ;
0000  03 04 05      NUMB1 DB    3,4,5          ;defines number 543
0003  07 08      NUMB2 DB    7,8            ;defines number 87
                                ;
                                ;Packed BCD data (least-significant data first)
                                ;
0005  37 34      NUMB3 DB    37H,34H       ;defines number 3437
0007  03 45      NUMB4 DB    3,45H        ;defines number 4503

```

EXAMPLE 1-21

```

//Unpacked BCD data (least-significant data first)
//
char Numb1 = 3,4,5;          ;defines number 543
char Numb2 = 7,8            ;defines number 87
//
//Packed BCD data (least-significant data first)
//
char Numb3 = 0x37,0x34      ;defines number 3437
char Numb4 = 3,0x45         ;defines number 4503

```

Hexadecimal Integers

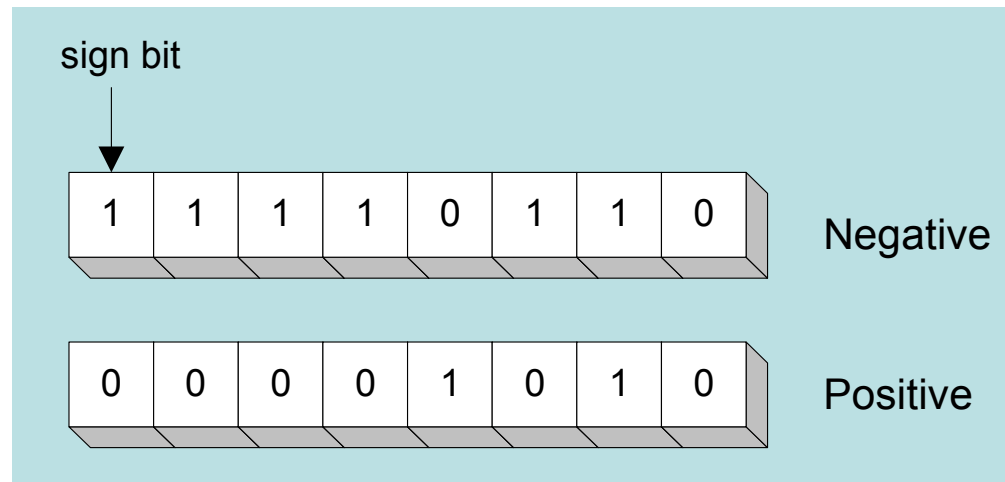
Binary values are represented in hexadecimal.

Table 1-5 Binary, Decimal, and Hexadecimal Equivalents.

Binary	Decimal	Hexadecimal	Binary	Decimal	Hexadecimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

Signed Integers

The highest bit indicates the sign. 1 = negative, 0 = positive



If the highest digit of a hexadecimal integer is > 7 , the value is negative.
Examples: 8A, C5, A2, 9D

Forming the Two's Complement

- Negative numbers are stored in two's complement notation
- Represents the additive Inverse

Starting value	00000001
Step 1: reverse the bits	11111110
Step 2: add 1 to the value from Step 1	11111110 +00000001
Sum: two's complement representation	11111111

Note that $00000001 + 11111111 = 00000000$

Ranges of Signed Integers

The highest bit is reserved for the sign. This limits the range:

Storage Type	Range (low–high)	Powers of 2
Signed byte	-128 to +127	-2^7 to $(2^7 - 1)$
Signed word	-32,768 to +32,767	-2^{15} to $(2^{15} - 1)$
Signed doubleword	-2,147,483,648 to 2,147,483,647	-2^{31} to $(2^{31} - 1)$
Signed quadword	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	-2^{63} to $(2^{63} - 1)$

Practice: What is the largest positive value that may be stored in 20 bits?

Integer Storage Sizes

Standard sizes:

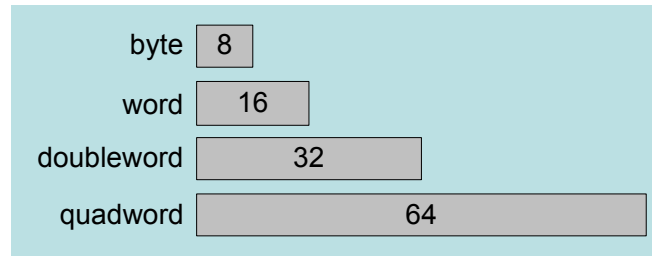


Table 1-4 Ranges of Unsigned Integers.

Storage Type	Range (low–high)	Powers of 2
Unsigned byte	0 to 255	0 to $(2^8 - 1)$
Unsigned word	0 to 65,535	0 to $(2^{16} - 1)$
Unsigned doubleword	0 to 4,294,967,295	0 to $(2^{32} - 1)$
Unsigned quadword	0 to 18,446,744,073,709,551,615	0 to $(2^{64} - 1)$

What is the largest unsigned integer that may be stored in 20 bits?

Byte-Sized Data

- Byte-sized data are stored as **unsigned** and **signed** integers
- The difference in this form is the **weight of the leftmost bit position**. Its value is **128** for the **unsigned** integer and **minus 128** for the **signed** integer.
- **Unsigned** integer range in value from 00H to FFH (0-255).
- **Signed** integers range in value from -128 to 0 to +127.

Byte-Sized Data: Example using assembler

EXAMPLE 1-24

```
                ;Unsigned byte-sized data
                ;
0000 FE        DATA1 DB      254      ;define 254 decimal
0001 87        DATA2 DB      87H      ;define 87 hexadecimal
0002 47        DATA3 DB      71       ;define 71 decimal
                ;
                ;Signed byte-sized data
                ;
0003 9C        DATA4 DB      -100     ;define -100 decimal
0004 64        DATA5 DB      +100     ;define +100 decimal
0005 FF        DATA6 DB      -1       ;define -1 decimal
0006 38        DATA7 DB      56       ;define 56 decimal
```

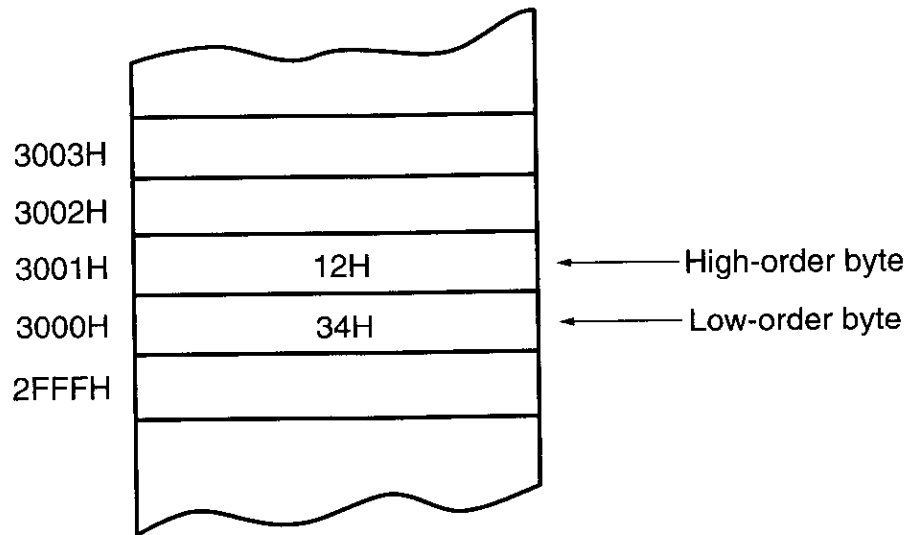
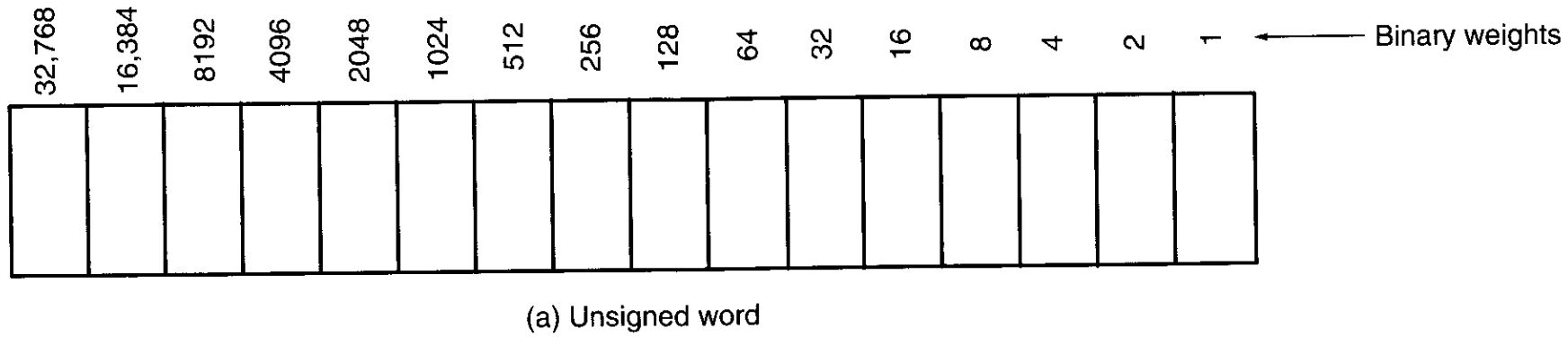
Byte-Sized Data: Example using MS Visual C++

EXAMPLE 1-25

```
//Unsigned byte-sized data
//
unsigned char Data1 = 254;      //define 254 decimal
unsigned char Data2 = 0x87;    //define 87 hexadecimal
unsigned char Data3 = 71      //define 71 decimal
//
//Signed byte-sized data
//
char Data4 = -100;            //define -100 decimal
char Data5 = +100;           //define +100 decimal
char Data6 = -1;             //define -1 decimal
char Data7 = 56;             //define 56 decimal
```

Word-Sized data

- A **word (16 bit)** is Formed with two bytes of data.
- The **least significant byte is always stored in the lowest-numbered** memory location, and the **most significant byte is stored in the highest.**
- This method of storing a number is called the **little endian** format. (used with Intel family of microprocessor)
- In the **big endian** format (**used with Motorola** family of microprocessor and MIPS machine) The lowest memory location contains the **most significant byte** .



(b) The contents of memory location 3000H and 3001H are the word 1234H.

FIGURE 1-15 The storage format for a 16-bit word in (a) a register and (b) two bytes of memory.

Word-Sized Data: Example using assembler and MS Visual C++

EXAMPLE 1-26

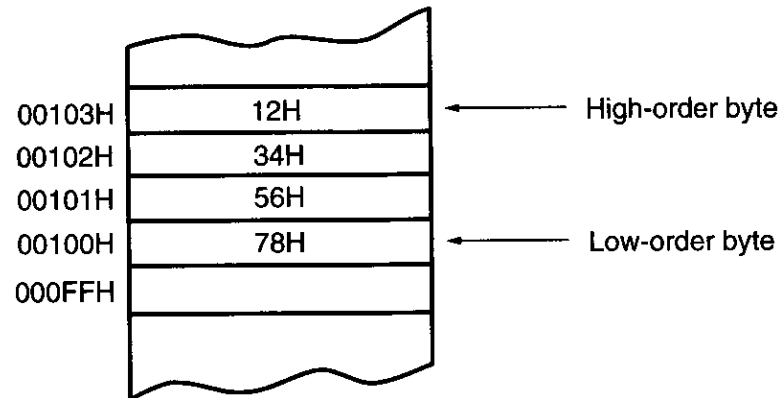
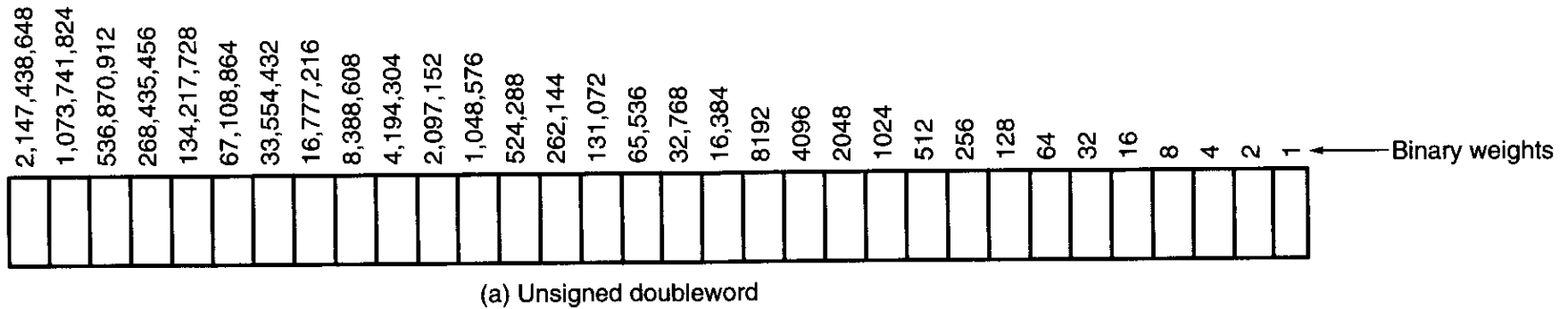
```
                ;Unsigned word-sized data
                ;
0000    09F0    DATA1 DW      2544            ;define 2544 decimal
0002    87AC    DATA2 DW      87ACH          ;define 87AC hexadecimal
0004    02C6    DATA3 DW      710              ;define 710 decimal
                ;
                ;Signed word-sized data
                ;
0006    CBA8    DATA4 DW     -13400         ;define -13400 decimal
0008    00C6    DATA5 DW      +198          ;define +198 decimal
000A    FFFF    DATA6 DW      -1           ;define -1 decimal
```

EXAMPLE 1-27

```
//Unsigned word-sized data
//
unsigned short Data1 = 2544;           //define 2544 decimal
unsigned short Data2 = 0x87AC         //define 87AC hexadecimal
unsigned short Data3 = 710;           //define 710 decimal
//
//Signed word-sized data
//
short Data4 = -13400;                 //define -13400 decimal
short Data5 = +198;                   //define +198 decimal
short Data6 = -1;                     //define -1 decimal
```

Doubleword-Sized Data

- Requires **four** bytes of memory because it is a **32-bit** number.
- Appear as a product after a multiplication and also as a dividend before a division.
- **Lsb** is stored in the **lowest** numbered memory location and **msb** is stored in the **highest**-numbered memory location using the **little endian** format.



(b) The contents of memory location 00100H–00103H are the doubleword 12345678H.

FIGURE 1–16 The storage format for a 32-bit word in (a) a register and (b) four bytes of memory.

Doubleword-Sized Data: Example using assembler (using DD or DWORD) and MS Visual C++

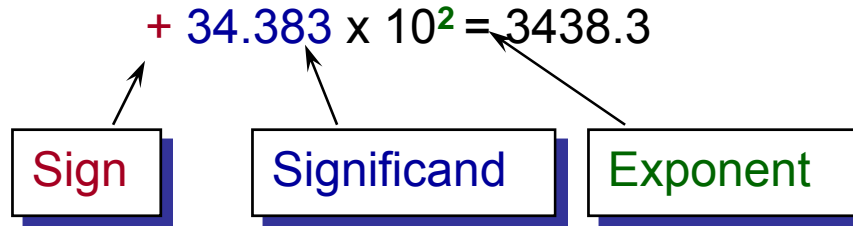
EXAMPLE 1-28

```
                                ;Unsigned doubleword-sized data
                                ;
0000    0003E1C0    DATA1 DD    254400    ;define 254400 decimal
0004    87AC1234    DATA2 DD    87AC1234H    ;define 87AC1234 hexadecimal
0008    00000046    DATA3 DD    70          ;define 70 decimal
                                ;
                                ;Signed doubleword-sized data
                                ;
000C    FFEB8058    DATA4 DD    -1343400    ;define -1343400 decimal
0010    000000C6    DATA5 DD    +198        ;define +198 decimal
0014    FFFFFFFF    DATA6 DD    -1          ;define -1 decimal
```

EXAMPLE 1-29

```
//Unsigned doubleword-sized data
//
unsigned int Data1 = 254400;           //define 254400 decimal
unsigned int Data2 = 0x87AC1234;      //define 87AC1234 hexadecimal
unsigned int Data3 = 70;              //define 70 decimal
//
//Signed doubleword-sized data
//
int Data4 = -1343400;                 //define -1342400 decimal
int Data5 = +198;                     //define +198 decimal
int Data6 = -1;                       //define -1 decimal
```


Real Floating-point Numbers Scientific Notation



$+ 3.4383 \times 10^3 = 3438.3$

Normalized form: Only one digit before the decimal point

$+3.4383000E+03 = 3438.3$

Floating point notation

8 digit significand can only represent 8 significant digits

Binary Floating Point Numbers

+ 101.1101

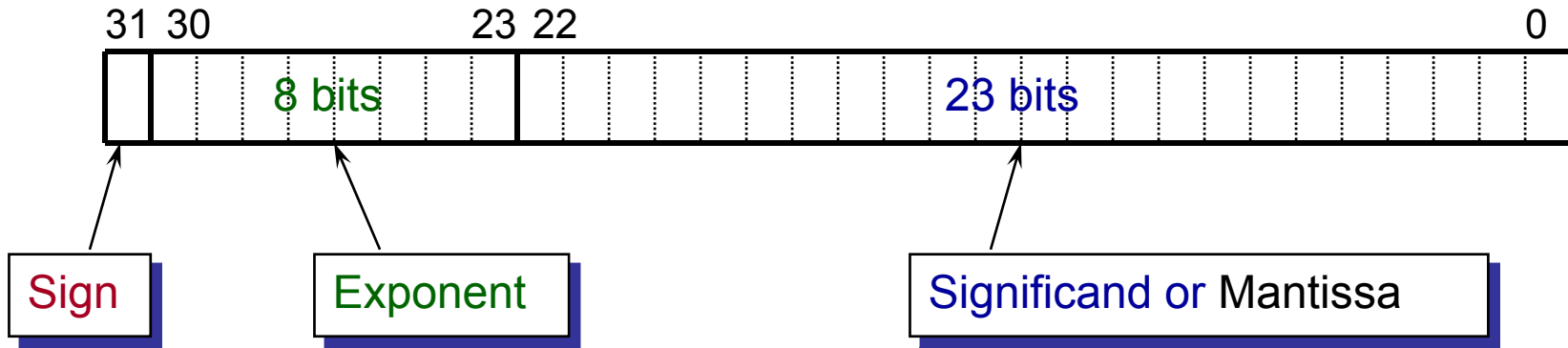
$$\begin{aligned} &= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} \\ &= 4 + 0 + 1 + 1/2 + 1/4 + 0 + 1/16 \\ &= 5.8125 \end{aligned}$$

+1.011101 E+2

Normalized so that the binary point immediately follows the leading digit

Note: First digit is always non-zero
--> First digit is always one.

IEEE Standard 754 For Floating Point Format



0: Positive
1: Negative

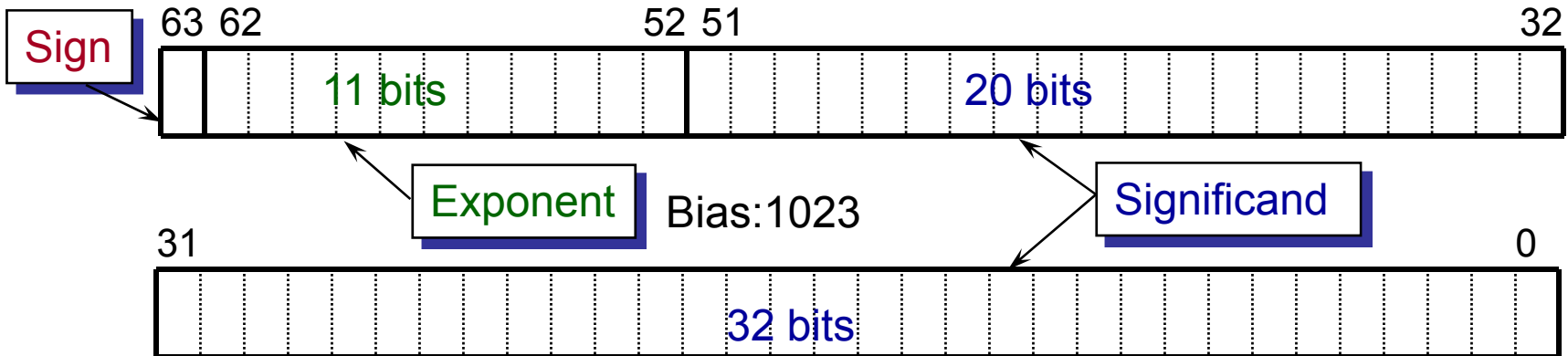
Biased by 127.

Leading '1' is implied, but not represented

$$\text{Number} = -1^{\text{S}} * (1 + \text{Sig}) \times 2^{\text{E}-127}$$

- Allows representation of numbers in range 2^{-127} to 2^{+128}
- Since the **significand** always starts with '1', we don't have to represent it explicitly
 - **Significand** is effectively 24 bits

IEEE 754 Double Precision Format



$$\text{Number} = -1^{\text{S}} * (1 + \text{Sig}) \times 2^{\text{E}-1023}$$

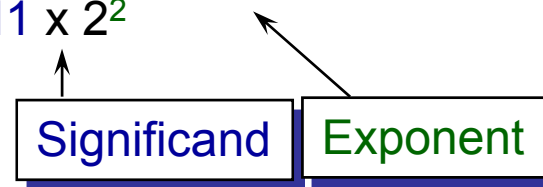
- Allows representation of numbers in range 2^{-1023} to 2^{+1024}
- Larger **significand** means more precision
- Takes two registers to hold one number

Conversion

Convert 5.75 to Single-Precision IEEE Floating Point

1. Convert 5.75_{10} to Binary $\rightarrow 101.11_2$

2. Normalize $\rightarrow 1.0111 \times 2^2$



3. **Sign = 0** (positive).

4. Add 127 (bias) to exponent. **Exponent = $129_{10} = 1000001_2$**

5. Express significand as 24 bits

Sig = 1.011100000000000000000000

6. Remove leading one from significand, leaving 23 bits

Sig = .011100000000000000000000

7. Put in proper bit fields

Number = 0 1000001 011100000000000000000000 = 0x40B80000

TABLE 1–11 Single-precision real numbers.

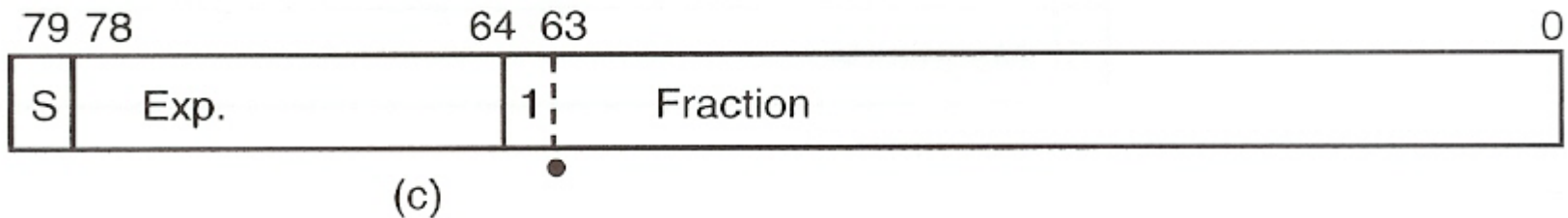
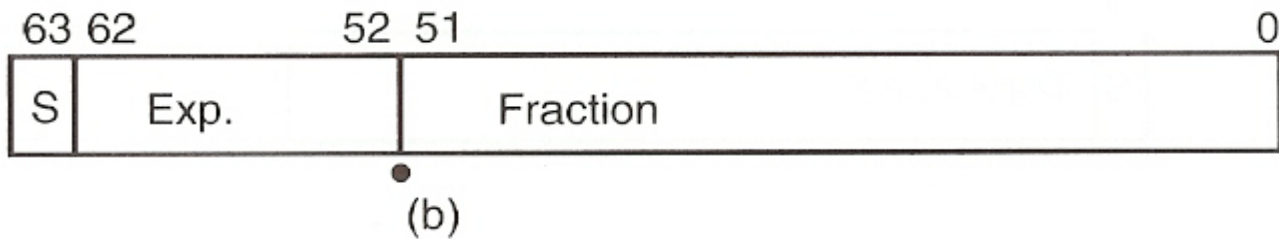
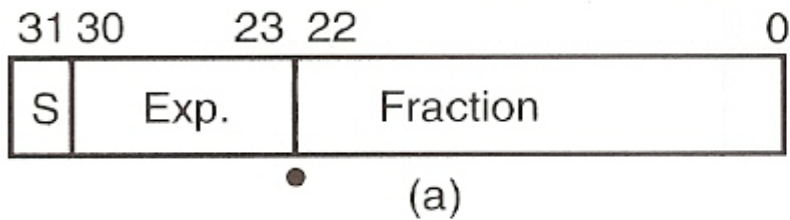
<i>Decimal</i>	<i>Binary</i>	<i>Normalized</i>	<i>Sign</i>	<i>Biased Exponent</i>	<i>Mantissa</i>
+12	1100	1.1×2^3	0	1000010	10000000 00000000 00000000
–12	1100	1.1×2^3	1	1000010	10000000 00000000 00000000
+100	1100100	1.1001×2^6	0	10000101	10010000 00000000 00000000
–1.75	1.11	1.11×2^0	1	01111111	11000000 00000000 00000000
+0.25	0.01	1.0×2^{-2}	0	01111101	00000000 00000000 00000000
+0.0	0	0	0	00000000	00000000 00000000 00000000

¹⁰IEEE is the Institute of Electrical and Electronic Engineers.

Directives for storing FP

- **Short** – single precision (32 bit)
 - **Real4** or Define Double (**DD**), **DWORD**
- **Long** – double precision (64 bits)
 - **Real8** or define Quadwords (**DQ**)
- **Extended** – extended precision (80 bits)
 - **Real10** or Define temporary (**DT**)

Floating-Point



Note: S = sign-bit and Exp. = exponent

Floating-Point Formats

- In the assembler

```
DATAA DD 23.4
DATAB DQ -345.0
DATAR DQ 3.5E2
```

- In C++

```
float DataC = 23.4;
double DataD = -345;
double DataE = 3.5e2;
```

- **DD** to define single precision 32-bit number
- **DQ** to define double precision 32-bit number

Chapter 2

The Microprocessor and its Architecture

Note: Adapted from Barry B. Brey (Author Slides)

Instructor:
Dr. Khalid A. Darabkeh

Introduction

- The microprocessor as a programmable device.
- The architecture of Intel microprocessors.
- Ways that the family members address the memory system.
- Addressing modes are described for the real, protected, and flat modes of operation.

INTERNAL MICROPROCESSOR ARCHITECTURE

- Before a program is written or instruction investigated, internal configuration of the microprocessor must be known.
- In a multiple core microprocessor each core contains the same programming model.
- Each core runs a separate task or thread simultaneously.

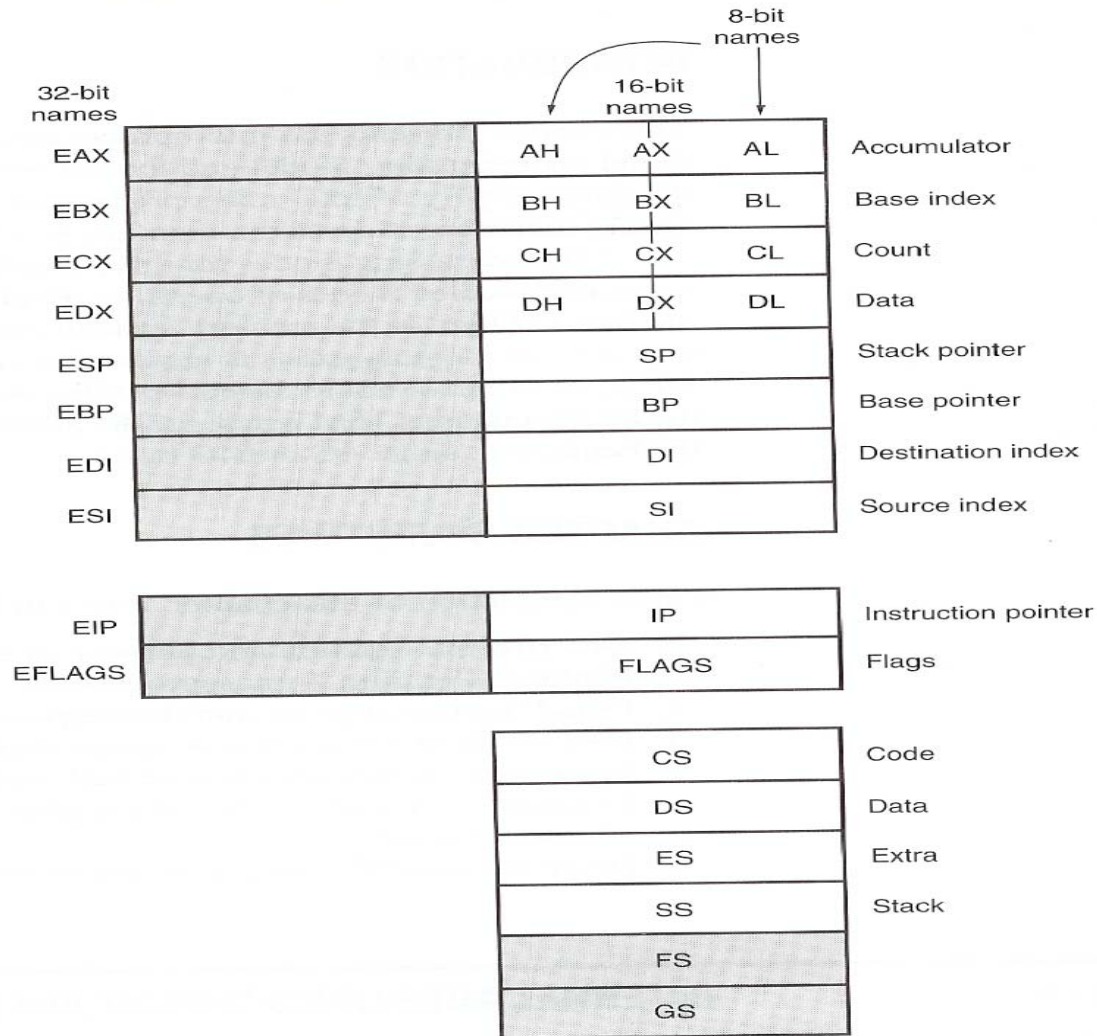
The Programming Model

- 8086 through Core2 consider two types of registers:
- **program visible.**
 - registers are used during programming and are specified by the instructions
- **program invisible.**
 - not addressable directly during applications programming

The Programming Model Cont.

- 80286 and above contain program-invisible registers to control and operate protected memory.
 - and other features of the microprocessor
- 80386 through Core2 microprocessors contain full 32-bit internal architectures.
- 8086 through the 80286 are fully upward-compatible to the 80386 through Core2.

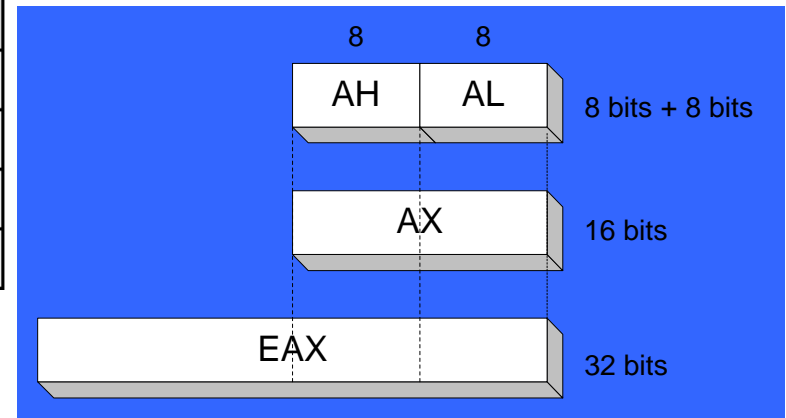
Figure 2–1 The programming model of the 8086 through the Core2 microprocessor including the 64-bit extensions.



Accessing Parts of Registers

- Use 8-bit name, 16-bit name, or 32-bit name
- Applies to EAX, EBX, ECX, and EDX

32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL



Multipurpose (General) Registers

- The top portion of the programming model contains the general purpose registers: EAX, EBX, ECX, EDX, EBP, ESI, and EDI.
- These registers, although general in nature, each have special purposes and names.

- **RAX** - a 64-bit register (RAX), a 32-bit register (**accumulator**) (EAX), a 16-bit register (AX), or as either of two 8-bit registers (AH and AL).
- The accumulator is used for instructions such as multiplication, division, and some of the adjustment instructions.
- Intel plans to expand the address bus to 52 bits to address 4P (peta) bytes of memory.

- **RBX**, addressable as RBX, EBX, BX, BH, BL.
 - BX register (**base index**) sometimes holds offset address of a location in the memory system in all versions of the microprocessor
- **RCX**, as RCX, ECX, CX, CH, or CL.
 - a (**count**) general-purpose register that also holds the count for various instructions
- **RDX**, as RDX, EDX, DX, DH, or DL.
 - a (**data**) general-purpose register
 - holds a part of the result from a multiplication or part of dividend before a division

- **RBP**, as RBP, EBP, or BP.
 - points to a memory (**base pointer**) location for memory data transfers
- **RDI** addressable as RDI, EDI, or DI.
 - often addresses (**destination index**) string destination data for the string instructions
- **RSI** used as RSI, ESI, or SI.
 - the (**source index**) register addresses source string data for the string instructions
 - like RDI, RSI also functions as a general-purpose register

- **R8 - R15** found in the Pentium 4 and Core2 if 64-bit extensions are enabled.
 - data are addressed as 64-, 32-, 16-, or 8-bit sizes and are of general purpose
- Most applications will not use these registers until 64-bit processors are common.
 - the 8-bit portion is the rightmost 8-bit only
 - bits 8 to 15 are not directly addressable as a byte

FIGURE 2-1 The programming model of the 8086 through the Core2 microprocessor including the 64-bit extensions.



Index and Base Registers

- Some registers have only a 16-bit name for their lower half:

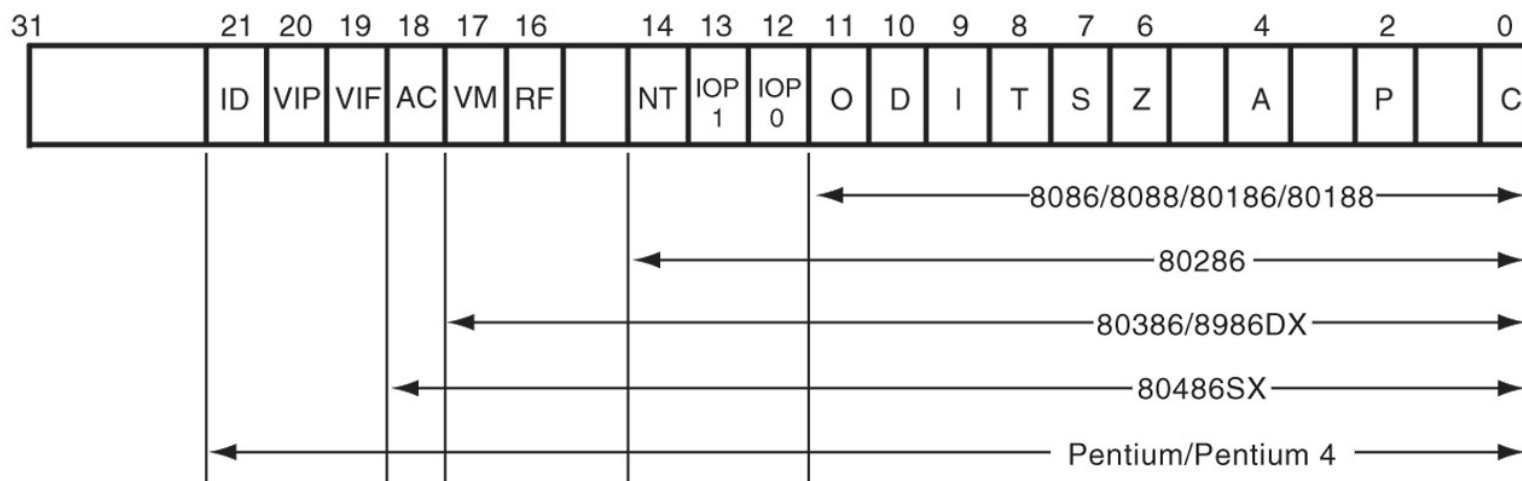
32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Special-Purpose Registers

- Include **RIP, RSP, and RFLAGS**
 - segment registers include CS, DS, ES, SS, FS, and GS
 - **Note:** Although it is theoretically possible to store data in the segment registers, this is never a good idea. The segment registers have a very special purpose - pointing at accessible blocks of memory.
- **RIP** addresses the next instruction in a section of memory.
 - defined as (**instruction pointer**) a code segment
- **RSP** addresses an area of memory called the stack.
 - the (**stack pointer**) stores data through this pointer

- **RFLAGS** indicate the condition of the microprocessor and control its operation.
- Figure 2–2 shows the flag registers of all versions of the microprocessor.
- Flags are upward-compatible from the 8086/8088 through Core2 .
- The rightmost five and the overflow flag are changed by most arithmetic and logic operations.
 - although data transfers do not affect them

Figure 2–2 The EFLAG and FLAG register counts for the entire 8086 and Pentium microprocessor family.



- Flags never change for any data transfer or program control operation.
- Some of the flags are also used to control features found in the microprocessor.

- Flag bits, with a brief description of function.
- **C (carry)** holds the carry after addition or borrow after subtraction.
 - also indicates error conditions
- **P (parity)** is the count of ones in a number expressed as even or odd. Logic 0 for odd parity; logic 1 for even parity.
 - if a number contains three binary one bits, it has odd parity
 - if a number contains no one bits, it has even parity

- **A (auxiliary carry)** holds the carry (half-carry) after addition or the borrow after subtraction between bit positions 3 and 4 of the result.
- **Z (zero)** shows that the result of an arithmetic or logic operation is zero.
- **S (sign)** flag holds the arithmetic sign of the result after an arithmetic or logic instruction executes.
- **T (trap)** The trap flag enables trapping through an on-chip debugging feature.

- **I (interrupt)** controls operation of the INTR (interrupt request) input pin.
- **D (direction)** selects increment or decrement mode for the DI and/or SI registers.
- **O (overflow)** occurs when signed numbers are added or subtracted.
 - an overflow indicates the result has exceeded the capacity of the machine

- **IOPL** used in protected mode operation to select the privilege level for I/O devices.
- **NT (nested task)** flag indicates the current task is nested within another task in protected mode operation.
- **RF (resume)** used with debugging to control resumption of execution after the next instruction.
- **VM (virtual mode)** flag bit selects virtual mode operation in a protected mode system.

- **AC, (alignment check)** flag bit activates if a word or doubleword is addressed on a non-word or non-doubleword boundary.
- **VIF** is a copy of the interrupt flag bit available to the Pentium 4—**(virtual interrupt)**
- **VIP (virtual)** provides information about a virtual mode interrupt for **(interrupt pending)** Pentium.
 - used in multitasking environments to provide virtual interrupt flags

- **ID (identification)** flag indicates that the Pentium microprocessors support the CUID instruction.
 - CUID instruction provides the system with information about the Pentium microprocessor

Segment Registers

- Generate memory addresses when combined with other registers in the microprocessor.
- Four or six segment registers in various versions of the microprocessor.
- A segment register functions differently in real mode than in protected mode.
- Following is a list of each segment register, along with its function in the system.

- **CS (code)** segment holds code (programs and procedures) used by the microprocessor.
- **DS (data)** contains most data used by a program.
 - Data are accessed by an offset address or contents of other registers that hold the offset address
- **ES (extra)** an additional data segment used by some instructions to hold destination data.

- **SS (stack)** defines the area of memory used for the stack.
 - stack entry point is determined by the stack segment and stack pointer registers
 - the BP register also addresses data within the stack segment

- **FS and GS** segments are supplemental segment registers available in 80386–Core2 microprocessors.
 - allow two additional memory segments for access by programs
- Windows uses these segments for internal operations, but no definition of their usage is available.

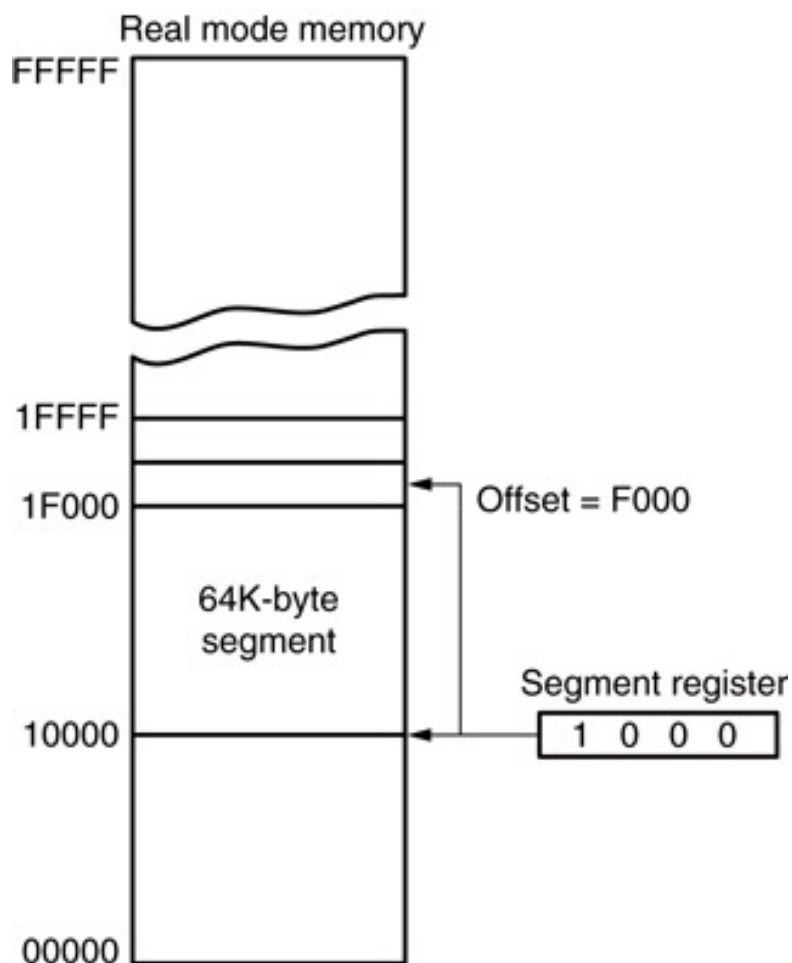
REAL MODE MEMORY ADDRESSING

- 80286 and above operate in either the real or protected mode.
- **Real mode operation** allows addressing of **only the first 1M byte** of memory space—even in Pentium 4 or Core2 microprocessor.
 - the first 1M byte of memory is called the **real memory, conventional memory, or DOS memory system**

Segments and Offsets

- All real mode memory addresses must consist of a segment address plus an offset address.
 - **segment address** defines the beginning address of any 64K-byte memory segment
 - **offset address** selects any location within the 64K byte memory segment
- Next slide shows how the **segment plus offset addressing** scheme selects a memory location.

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

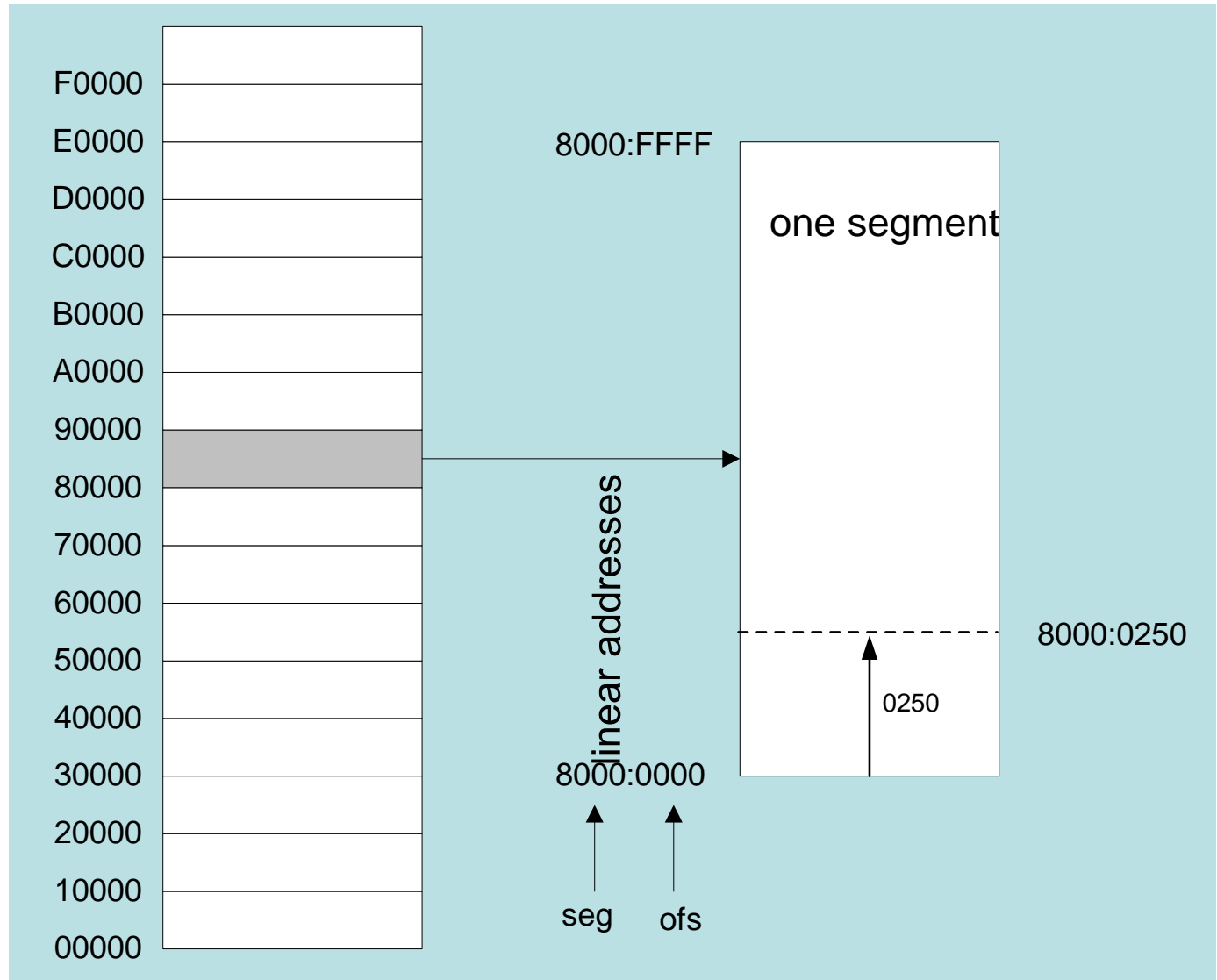


- this shows a memory segment beginning at 10000H, ending at location 1FFFFH
 - 64K bytes in length
- also shows how an offset address, called a **displacement**, of F000H selects location 1F000H in the memory

- Once the beginning address is known, the **ending address** is found by adding **FFFFH**.
 - because a **real mode segment of memory is 64K in length**
- The offset address is always added to the segment starting address to locate the data.
- Segment and offset address is sometimes written as 1000:2000.
 - a segment address of 1000H; an offset of 2000H

Segmented Memory

Segmented memory addressing: absolute (linear) address is a combination of a 16-bit segment value added to a 16-bit offset



Calculating Linear Addresses

- Given a segment address, multiply it by 16 (10H) (add a hexadecimal zero to the right), and add it to the offset
- Example: convert 08F1:0100 to a linear address

Adjusted Segment value:	0	8	F	1	0		
Add the offset:			0	1	0	0	
Linear address:			0	9	0	1	0

Effective Address Calculations

- EA = segment x 10H plus offset
 - (a) 10023 = 10000 + 0023
 - (b) ABC34 = AAF00 + 0134
 - (c) 21FF0 = 12000 + FFF0

Example (a) contained 1000 in the segment register, example (b) contained a AAF0 in the segment register, and example (c) contained a 1200 in the segment register.

Effective Address Calculations Cont.

What linear address corresponds to the segment/offset address 028F:0030?

$$028F0 + 0030 = 02920$$

Always use hexadecimal notation for addresses.

Effective Address Calculations

Cont.

What segment addresses correspond to the linear address 28F30h?

Many different segment-offset addresses can produce the linear address 28F30h. For example:

28F0:0030, 28F3:0000, 28B0:0430, . . .

Default Segment and Offset Registers

- The microprocessor has rules that apply to segments whenever memory is addressed.
 - these define the segment and offset register combination
- The **code segment** register defines the start of the code segment.
- The **instruction pointer** locates the next instruction within the code segment.

- Another of the default combinations is the **stack**.
 - stack data are referenced through the stack segment at the memory location addressed by either the **stack pointer** (SP/ESP) or the pointer (BP/EBP)
- Figure 2–4 shows a system that contains four memory segments.
 - a memory segment can touch or overlap if 64K bytes of memory are not required for a segment

TABLE 2-3 Default 16-bit segment and offset combinations.

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

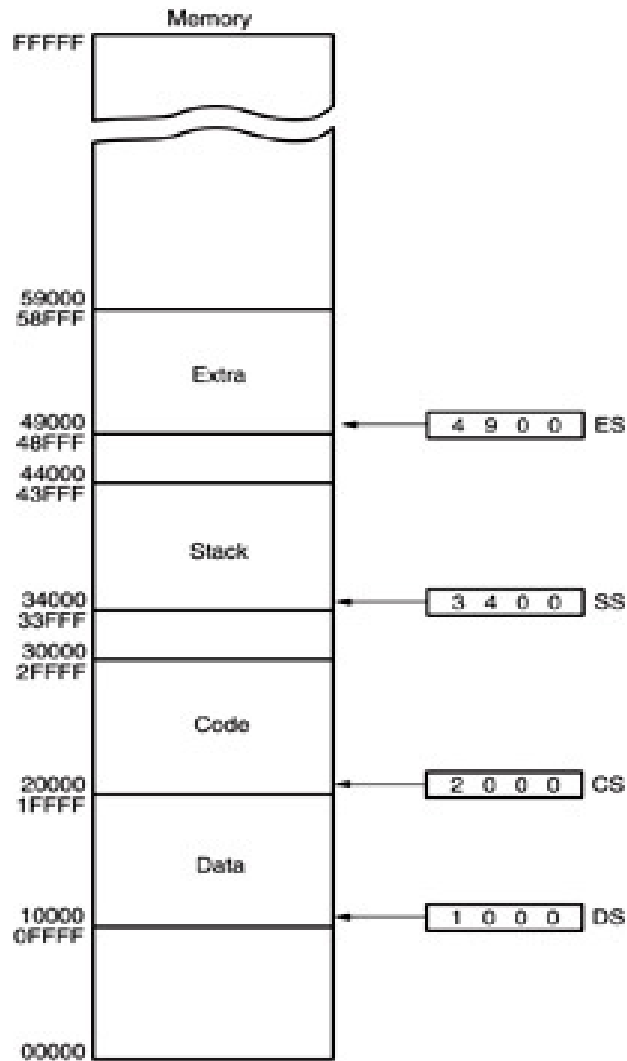
THE MICROPROCESSOR AND ITS ARCHITECTURE

TABLE 2-4 Default 32-bit segment and offset combinations.

<i>Segment</i>	<i>Offset</i>	<i>Special Purpose</i>
CS	EIP	Instruction address
SS	ESP or EBP	Stack address
DS	EAX, EBX, ECX, EDX, ESI, EDI, an 8- or 32-bit number	Data address
ES	EDI for string instructions	String destination address
FS	No default	General address
GS	No default	General address

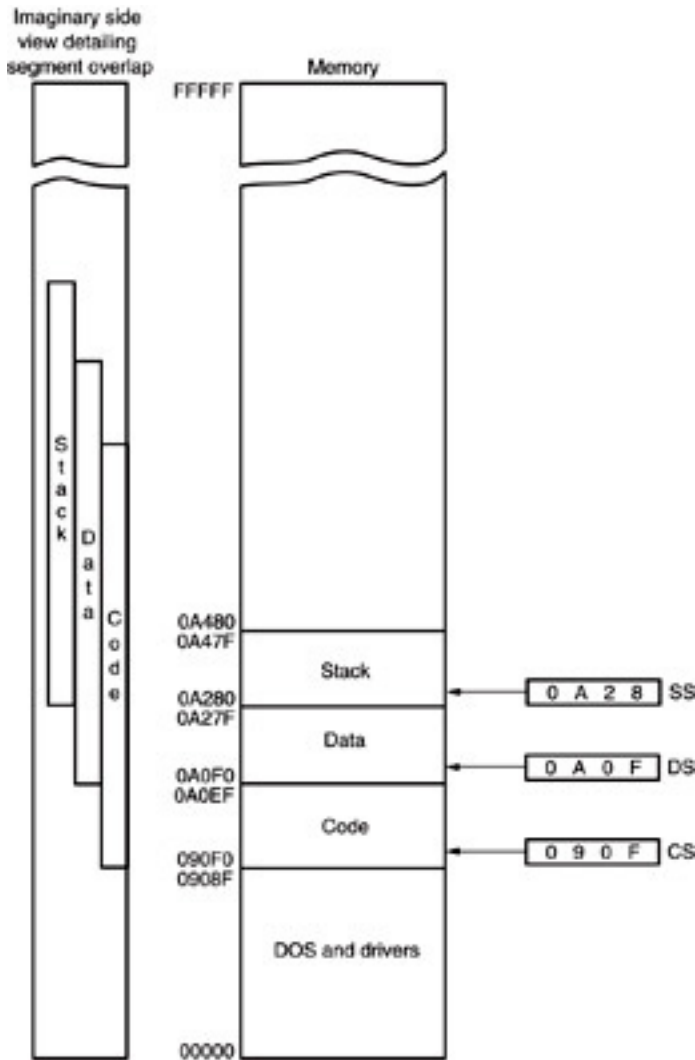
allows DOS programs to be relocated in the memory system. It also allows programs written to function in the real mode to operate in a protected mode system. A **relocatable program** is one that can be placed into any area of memory and executed without change. **Relocatable data** are data that can be placed in any area of memory and used without any change to

Figure 2–4 A memory system showing the placement of four memory segments.



- think of segments as windows that can be moved over any area of memory to access data or code
- a program can have more than four or six segments,
 - but only access four or six segments at a time

Figure 2–5 An application program containing a code, data, and stack segment loaded into a DOS system memory.



- a program placed in memory by DOS is loaded in the TPA at the first available area of memory above drivers and other TPA programs
- area is indicated by a **free-pointer** maintained by DOS
- program loading is handled automatically by the **program loader** within DOS

Segment and Offset Addressing Scheme Allows Relocation

- Segment plus offset addressing allows DOS programs to be relocated in memory.
- A **relocatable program** is one that can be placed into any area of memory and executed without change.
- **Relocatable data** are data that can be placed in any area of memory and used without any change to the program.

- Because memory is addressed within a segment by an offset address, the memory segment can be moved to any place in the memory system without changing any of the offset addresses.
- Only the contents of the segment register must be changed to address the program in the new area of memory.
- Windows programs are written assuming that the first 2G of memory are available for code and data.

INTRODUCTION TO PROTECTED MODE MEMORY ADDRESSING

- Allows access to data and programs located within & above the first 1M byte of memory.
- **Protected mode** is where *Windows* operates.
- In place of a segment address, the **segment register** contains a **selector** that selects a descriptor from a descriptor table.
- The **descriptor** describes the **memory segment's location, length, and access rights.**

Selectors and Descriptors

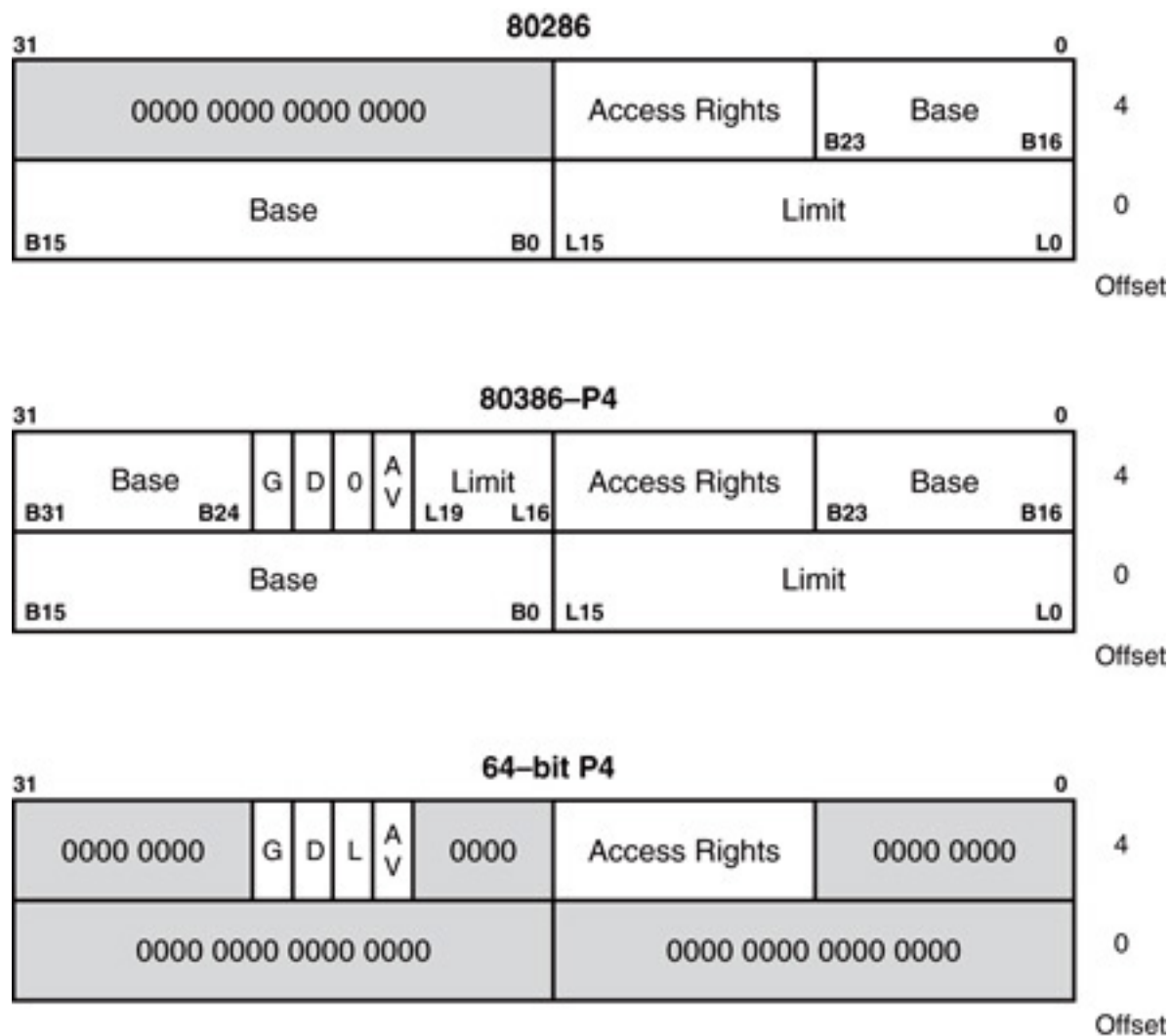
- The **descriptor** is located in the segment register & describes the location, length, and access rights of the segment of memory.
 - it selects one of 8192(64k/8) descriptors from one of two tables of descriptors
- In protected mode, this segment number can address any memory location in the system for the code segment.
- Indirectly, the register still selects a memory segment, but not directly as in real mode.

- **Global descriptors** contain segment definitions that **apply to all programs**.
- **Local descriptors** are usually **unique to an application**.
 - a global descriptor might be called a **system descriptor**, and local descriptor an **application descriptor**
- Figure 2–6 shows the format of a descriptor for the 80286 through the Core2.
 - **each descriptor is 8 bytes in length**
 - global and local **descriptor tables** are a maximum of **64K bytes in length**

Table 1–6 The Intel family of microprocessor bus and memory sizes.

<i>Microprocessor</i>	<i>Data Bus Width</i>	<i>Address Bus Width</i>	<i>Memory Size</i>
8086	16	20	1M
8088	8	20	1M
80186	16	20	1M
80188	8	20	1M
80286	16	24	16M
80386SX	16	24	16M
80386DX	32	32	4G
80386EX	16	26	64M
80486	32	32	4G
Pentium	64	32	4G
Pentium Pro–Core2	64	32	4G
Pentium Pro–Core2 (if extended addressing is enabled)	64	36	64G
Pentium 4 and Core2 with 64-bit extensions enabled	64	40	1T
Itanium	128	40	1T

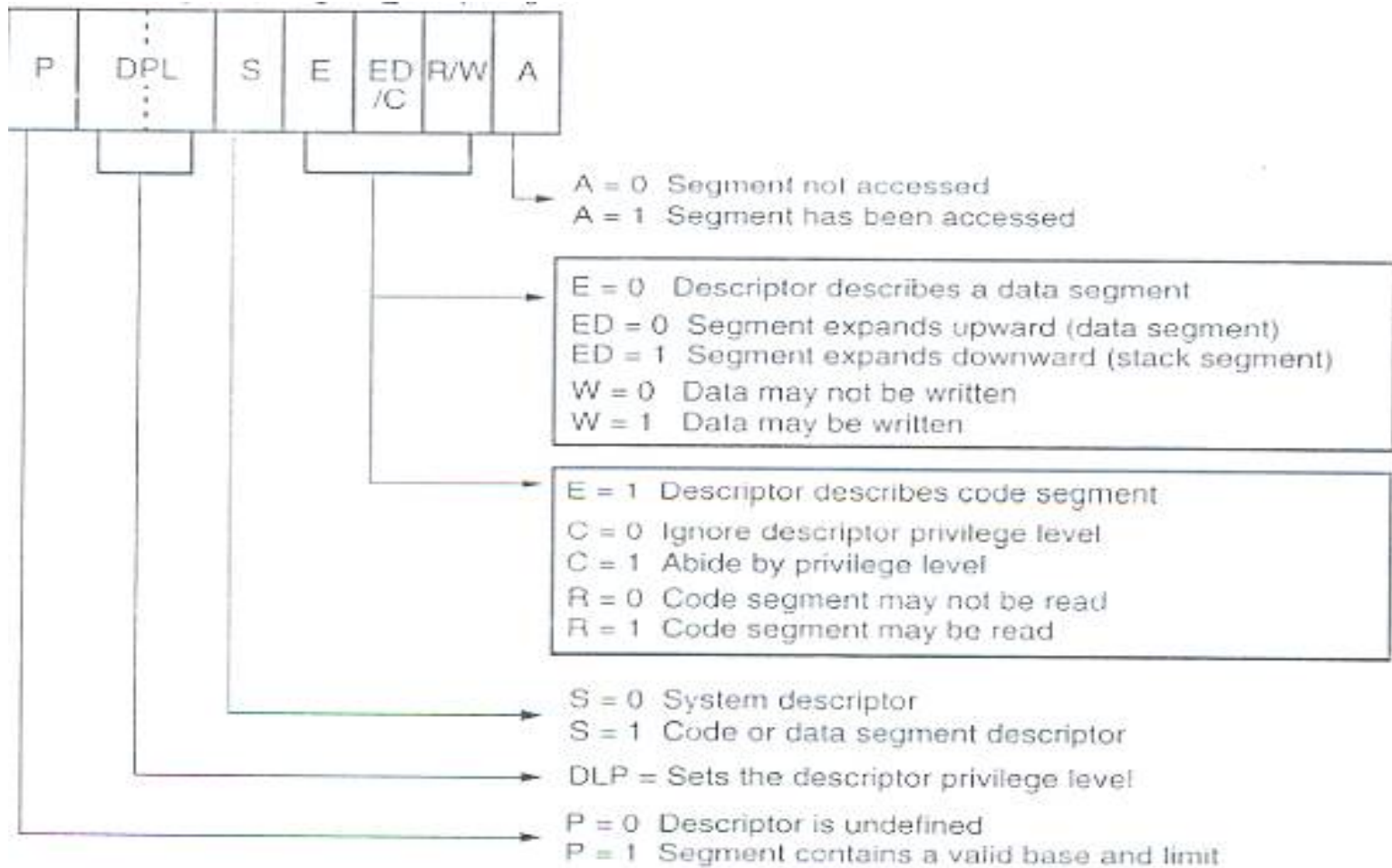
Figure 2–6 The 80286 through Core2 64-bit descriptors.



- The **base address** of the descriptor indicates the **starting location of the memory segment**.
- The **segment limit** contains the **last address found in a segment**.
- The **G**, or **granularity bit** allows a segment length of 4K to 4G bytes.
 - G=0; limit is 00000H to FFFFFFFH
 - G=1; limit is appended with FFF; 00000FFF to FFFFFFFF

- The **AV** bit, in the 80386 and above descriptor, is used by some operating systems to indicate that the segment is available (**AV=1**) or not available (**AV=0**). The **D** bit indicates how the 80386 through the Core2 instructions access register and memory data in the protected or real mode. If **D=0**, the instructions are 16-bit instructions, compatible with 8086-80286 microprocessor. This means that the instructions use 16-bit offset addresses and 16-bit register by default. This mode is often called the 16-bit instruction mode or DOS mode. If **D=1**, the instructions are 32-bit instructions. By default, the 32-bit instruction mode assumes that all offset addresses and all registers are 32-bit.

- In the 64-bit descriptor, the L bit (probably means large, but Intel calls it the 64-bit) selects 64-bit addresses in a Pentium 4 or Core2 with 64-bit extensions when L=1 and 32-bit compatibility mode when L=0.
- In 64-bit protected operation, the code segment register is still used to select a section of code from the memory.
- Notice that the 64-bit descriptor has no limit or base address. It only contains an access rights byte and the control bits.
- In 64-bit mode, there is no segment or limit in the descriptor and the base address of the segment, although not placed in the descriptor, is 00 0000 0000H. This means that all code segments start at address zero for 64-bit operation. There is no limit checks for a 64-bit code segment.



Note: Some of the letters used to describe the bits in the access rights bytes vary in Intel documentation.

RE 2-7 The access rights byte for the 80286 through Core2 descriptor.

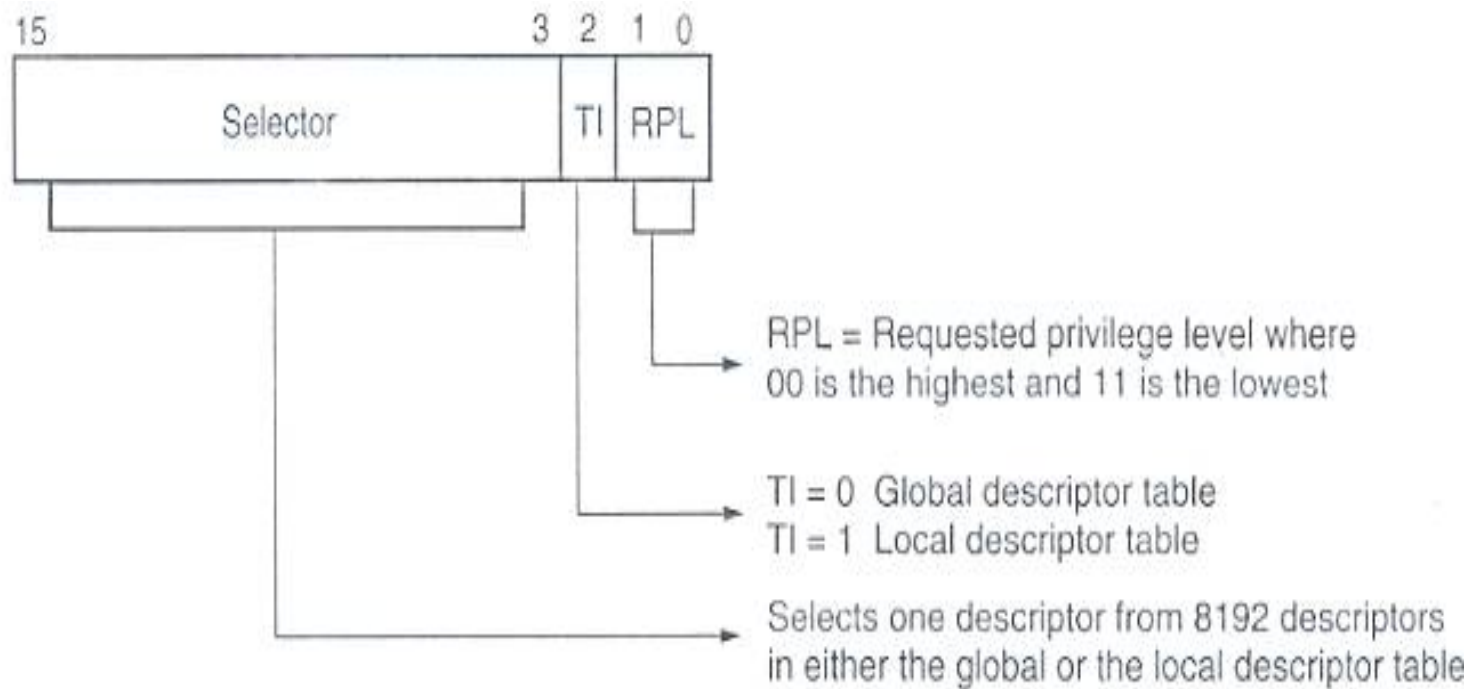


FIGURE 2-8 The contents of a segment register during protected mode operation of the 80286 through Core2 microprocessors.

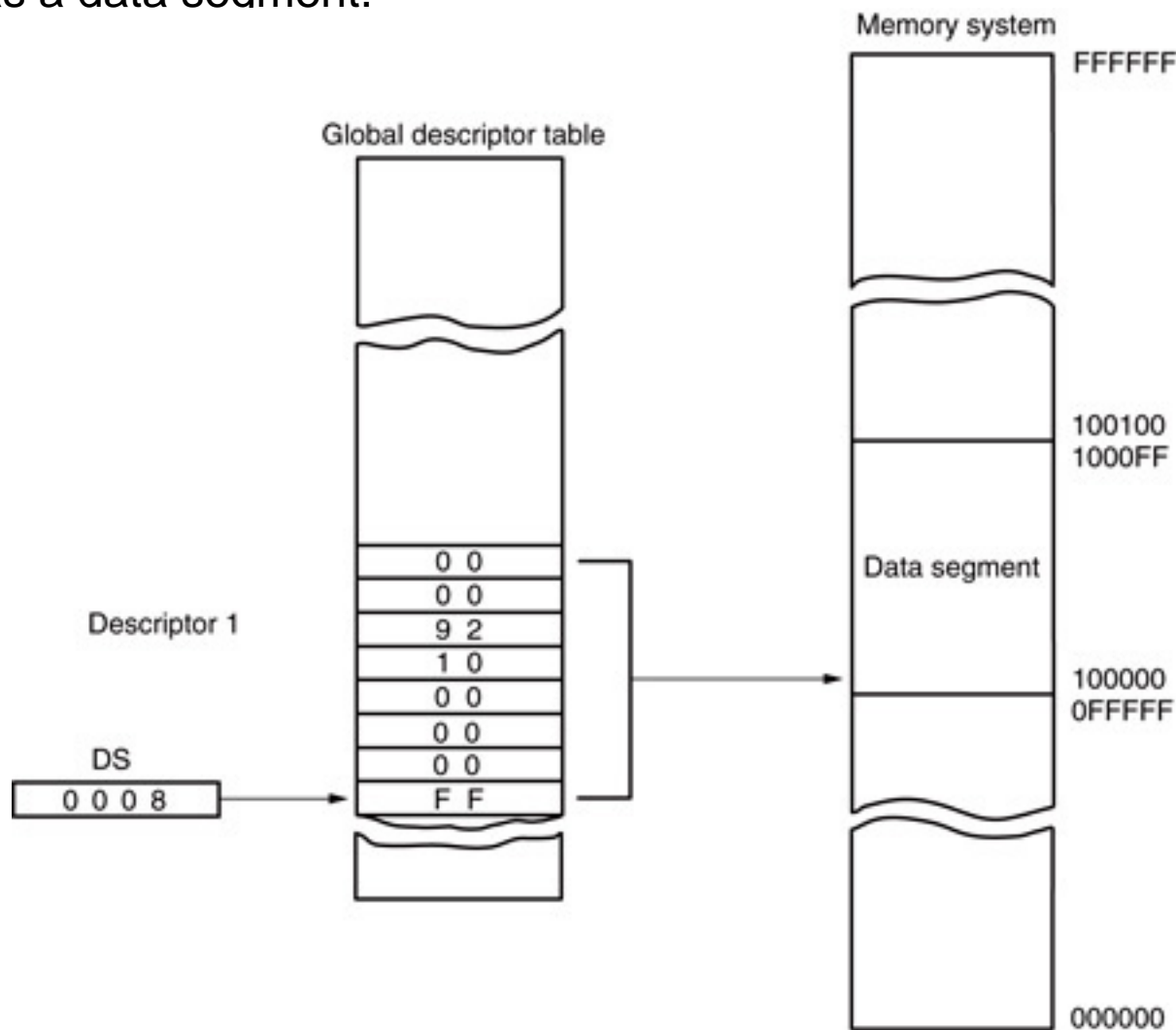
- The **access rights byte** controls access to the protected mode segment. **This byte describes how the segment functions in the system.** The access rights byte allows complete control over the segment. If the segment is a data segment, the direction of growth is specified. If the segment grows beyond its limit, the microprocessor's operating system program is interrupted, indicating a general protection fault. You can even specify whether a data segment can be written or is write-protected. The code segment is also controlled in a similar fashion.

- **Figure 2-8 shows how the segment register functions in the protected mode system.**
- **The segment contains a 13-bit selector field, a table selector bit, and a requested privilege field. The 13-bit selector chooses one of the 8192 descriptors from the descriptor table. The T1 bit selects either the global descriptor table (T1=0) or the local descriptor table (T=1). The requested privilege level (RPL) requests the access privilege level of a memory segment. The highest privilege level is 00 and the lowest is 11. If the requested privilege level matches or is higher in priority than the privilege level set by the access rights byte, access is granted. For example, if the requested privilege level is 10 and the access rights byte sets the segment privilege level at 11, access is granted.**

- Operating systems operate in a 16- or 32-bit environment.
- DOS uses a 16-bit environment.
- Most Windows applications use a 32-bit environment called **WIN32**.
- MSDOS/PCDOS & Windows 3.1 operating systems require 16-bit instruction mode.
- Instruction mode is accessible only in a protected mode system such as Windows Vista.

- Figure 2–9 shows how the segment register, containing a selector, chooses a descriptor from the global descriptor table.
- The entry in the global descriptor table selects a segment in the memory system.
- Descriptor zero is called the null descriptor, must contain all zeros, and may not be used for accessing memory.

Figure 2–9 Using the DS register to select a description from the global descriptor table. In this example, the DS register accesses memory locations 00100000H–001000FFH as a data segment.



Flat Mode Memory

- The memory system in a Pentium-based computer (Pentium 4 or Core 2) that uses 64-bit extensions uses a flat mode memory system.
- A *flat mode memory* system is one in which there is **no segmentation**.
 - does not use a segment register to address a location in the memory
 - The segment register is used to select a descriptor from the descriptor table that defines the access bytes and control bits.
- First byte address is at 00 0000 0000H; the last location is at FF FFFF FFFFH.
 - address is 40-bits
- The segment register still selects the privilege level of the software.
- **The offset address is the actual physical address in 64-bit mode.**

- Real mode system is *not* available if the processor operates in the 64-bit mode.
- Protection and paging are allowed in the 64-bit mode.
- The CS register is still used in the protected mode operation in the 64-bit mode.
- Most programs today are operated in the IA32 compatible mode.
 - current software operates properly, but this will change in a few years as memory becomes larger and most people have 64-bit computers

Program-invisible registers

- The global and local descriptor tables are found in the memory. In order to access and specify the address of these tables, the 80286-Core2 contain program-invisible registers. These registers control the microprocessor when operated in the protected mode.
- When the protected mode operation is desired, the address of the global descriptor and its limit are loaded into the **GDTR (global descriptor table register)**.

Program-invisible registers Cont.

- 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, just as a segment register is loaded with a selector.

SUMMARY

- The programming model of the 8086 through 80286 contains 8- and 16-bit registers.
- The programming model of the 80386 and above contains 8-, 16-, and 32-bit extended registers as well as two additional 16-bit segment registers: FS and GS.

SUMMARY

(cont.)

- 8-bit registers are AH, AL, BH, BL, CH, CL, DH, and DL.
- 16-bit registers are AX, BX, CX, DX, SP, BP, DI, and SI.
- The segment registers are CS, DS, ES, SS, FS, and GS.
- 32-bit extended registers are EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI.

SUMMARY

(*cont.*)

- The 64-bit registers in a Pentium 4 with 64-bit extensions are RAX, RBX, RCX, RDX, RSP, RBP, RDI, RSI, and R8 through R15.
- In addition, the microprocessor contains an instruction pointer (IP/EIP/RIP) and flag register (FLAGS, EFLAGS, or RFLAGS).
- All real mode memory addresses are a combination of a segment address plus an offset address.

SUMMARY

(*cont.*)

- The starting location of a segment is defined by the 16-bit number in the segment register that is appended with a hexadecimal zero at its rightmost end.
- The offset address is a 16-bit number added to the 20-bit segment address to form the real mode memory address.
- All instructions (code) are accessed by the combination of CS (segment address) plus IP or EIP (offset address).

SUMMARY

(*cont.*)

- Data are normally referenced through a combination of the DS (data segment) and either an offset address or the contents of a register that contains the offset address.
- The 8086-Core2 use BX, DI, and SI as default offset registers for data if 16-bit registers are selected.
- The 80386 and above can use the 32-bit registers EAX, EBX, ECX, EDX, EDI, and ESI as default offset registers for data.

SUMMARY

(*cont.*)

- Protected mode operation allows memory above the first 1M byte to be accessed by the 80286 through the Core2 microprocessors.
- This extended memory system (XMS) is accessed via a segment address plus an offset address, just as in the real mode.
- In the protected mode, the segment starting address is stored in a descriptor that is selected by the segment register.

SUMMARY

(*cont.*)

- A protected mode descriptor contains a base address, limit, and access rights byte.
- The base address locates the starting address of the memory segment; the limit defines the last location of the segment.
- The access rights byte defines how the memory segment is accessed via a program.

SUMMARY

(*cont.*)

- The 80286 microprocessor allows a memory segment to begin at any of its 16M bytes of memory using a 24-bit base address.
- The 80386 and above allow a memory segment to begin at any of its 4G bytes of memory using 32-bit base address.

SUMMARY

(*cont.*)

- The limit is a 16-bit number in the 80286 and a 20-bit number in the 80386 and above. This allows an 80286 memory segment limit of 64K bytes, and an 80386 and above memory segment limit of either 1M bytes (G=0) or 4G bytes (G=1). The L bit selects 64-bit address operation in the code descriptor.

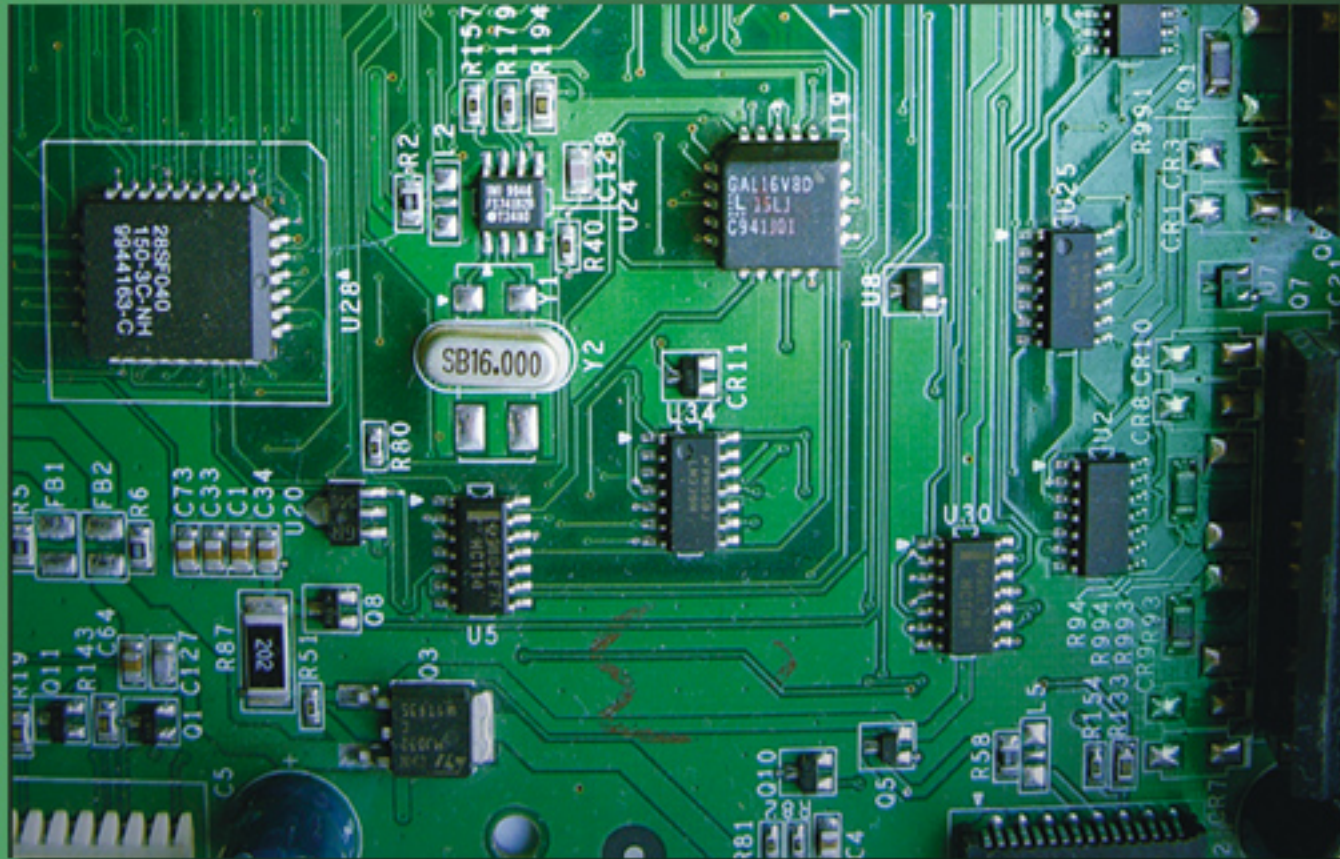
SUMMARY

- The flat mode memory contains 1T byte of memory using a 40-bit address.
- In the future, Intel plans to increase the address width to 52 bits to access 4P bytes of memory.
- The flat mode is only available in the Pentium 4 and Core2 that have their 64-bit extensions enabled.

The Intel Microprocessors

8086/8088, 80186/80188, 80286, 80386, 80486 Pentium, Pentium Pro Processor, Pentium II, Pentium 4, and Core2 with 64-bit Extensions

Architecture, Programming, and Interfacing



EIGHTH EDITION

Barry B. Brey

PEARSON

Chapter 3: Addressing Modes

Chapter 3

Addressing Modes

***Note:** Adapted from Barry B. Brey (Author Slides)*

Instructor:
Dr. Khalid A. Darabkeh

Introduction

- Efficient software development for the microprocessor requires a complete **familiarity** with the **addressing modes** employed by each instruction.
- This chapter explains the operation of the stack memory so that the PUSH and POP instructions and other stack operations will be understood.

Assembly Language

Each statement in an assembly language program consists of **four parts** or **fields**.

LABEL	OPCODE	OPERAND	COMMENT
DATA1	DB	23H	;define DATA1 as a byte of 23H
START:	MOV	AL,BL	;copy BL into AL

- The leftmost field is called the *label*.
 - used to **identify** the **name of a memory** location used for storing data and for other purposes
- All **labels** must **begin** with a **letter** or one of the following special characters: @, \$, -, or ?.
 - a **label** may any length from **1 to 35** characters .

- The next field to the right is the **opcode field** or **operation code**.
 - designed to hold the instruction, or opcode
 - the MOV part of the move data instruction is an example of an opcode
- Right of the opcode field is the **operand field**.
 - contains information used by the opcode
 - the **MOV AL,BL** instruction has the opcode MOV and **operands AL** and **BL**
- The **comment field**, the **final field**, contains a comment about the instruction(s).
 - **comments** always begin with a **semicolon (;)**

Example of MASM program

EXAMPLE 3-2

```
0000          .MODEL TINY          ;choose single segment model
          .CODE                    ;start of code segment
          .STARTUP                  ;start of program

0100  B8 0000  MOV AX,0             ;place 0000H into AX
0103  BB 0000  MOV BX,0             ;place 0000H into BX
0106  B9 0000  MOV CX,0             ;place 0000H into CX

0109  8B F0    MOV SI,AX            ;copy AX into SI
010B  8B F8    MOV DI,AX            ;copy AX into DI
010D  8B E8    MOV BP,AX            ;copy AX into BP

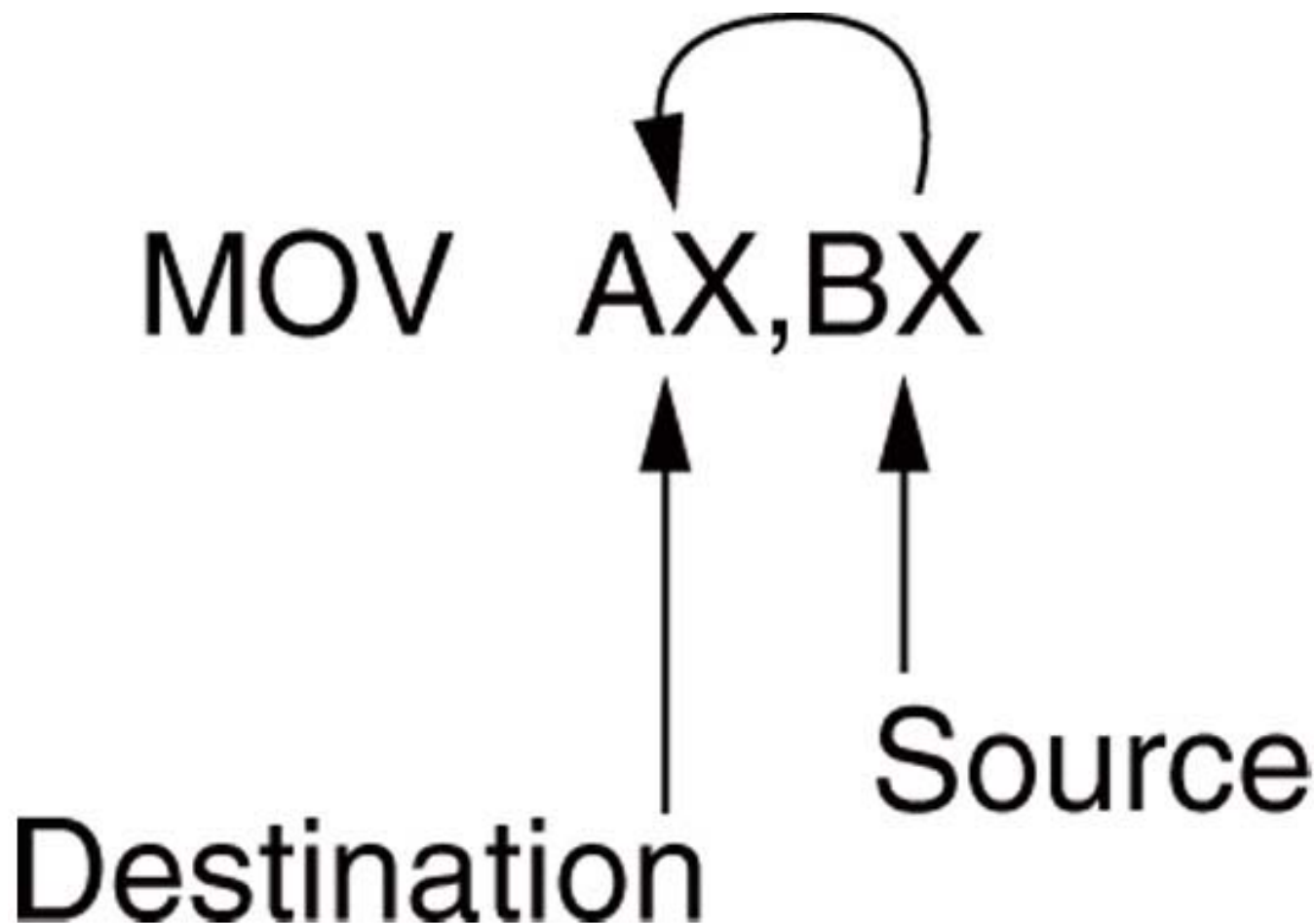
          .EXIT                    ;exit to DOS
          END                      ;end of program
```

Tiny program always assembled as a command (**.COM**) program

3–1 DATA ADDRESSING MODES

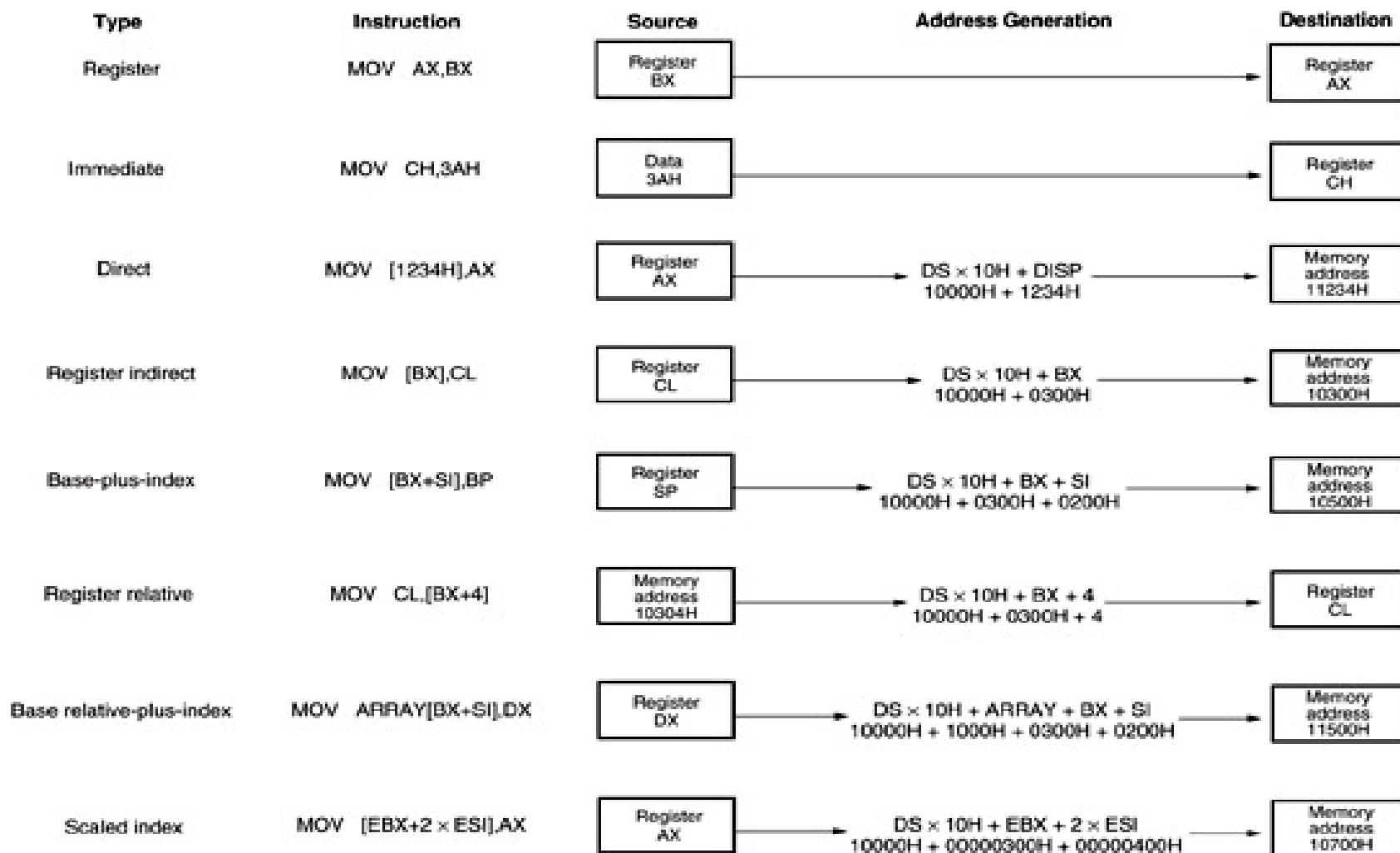
- **MOV** instruction is a common and flexible instruction.
 - provides a basis for explanation of data-addressing modes
- Figure 3–1 illustrates the MOV instruction and defines the direction of data flow.
- **Source** is to the right and **destination** the left, next to the opcode MOV.
 - an **opcode**, or **operation code**, tells the microprocessor which operation to perform

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



- **MOV** really moves nothing. **MOV copies** the **source** into the **destination**. It probably should be named COP for copy, but it is not.
- Figure 3–2 shows all possible variations of the data-addressing modes using MOV.
- These data-addressing modes are found with all versions of the Intel microprocessor.
 - except for the scaled-index-addressing mode, found only in 80386 through Core2

Figure 3–2 8086–Core2 data-addressing modes.



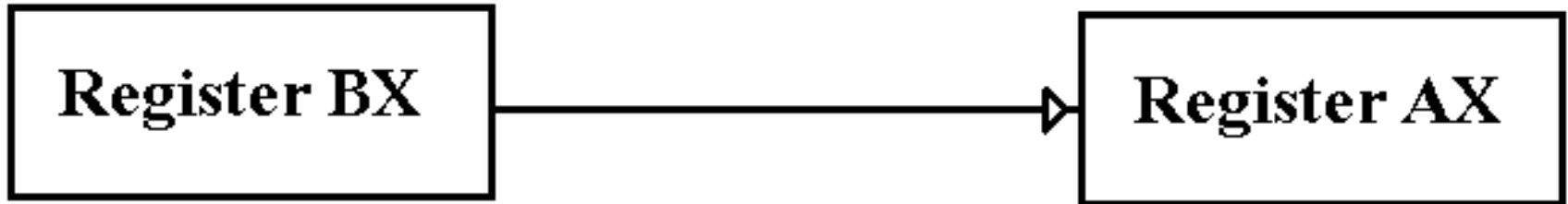
Notes: EBX = 00000300H, ESI = 00000200H, ARRAY = 1000H, and DS = 1000H

I- Register Addressing

- The most common form of data addressing.
 - once register names learned, easiest to apply.
- The microprocessor contains these 8-bit register names used with register addressing: AH, AL, BH, BL, CH, CL, DH, and DL.
- 16-bit register names: AX, BX, CX, DX, SP, BP, SI, and DI.

Source

Destination



Type
Register

Instruction
MOV AX,BX

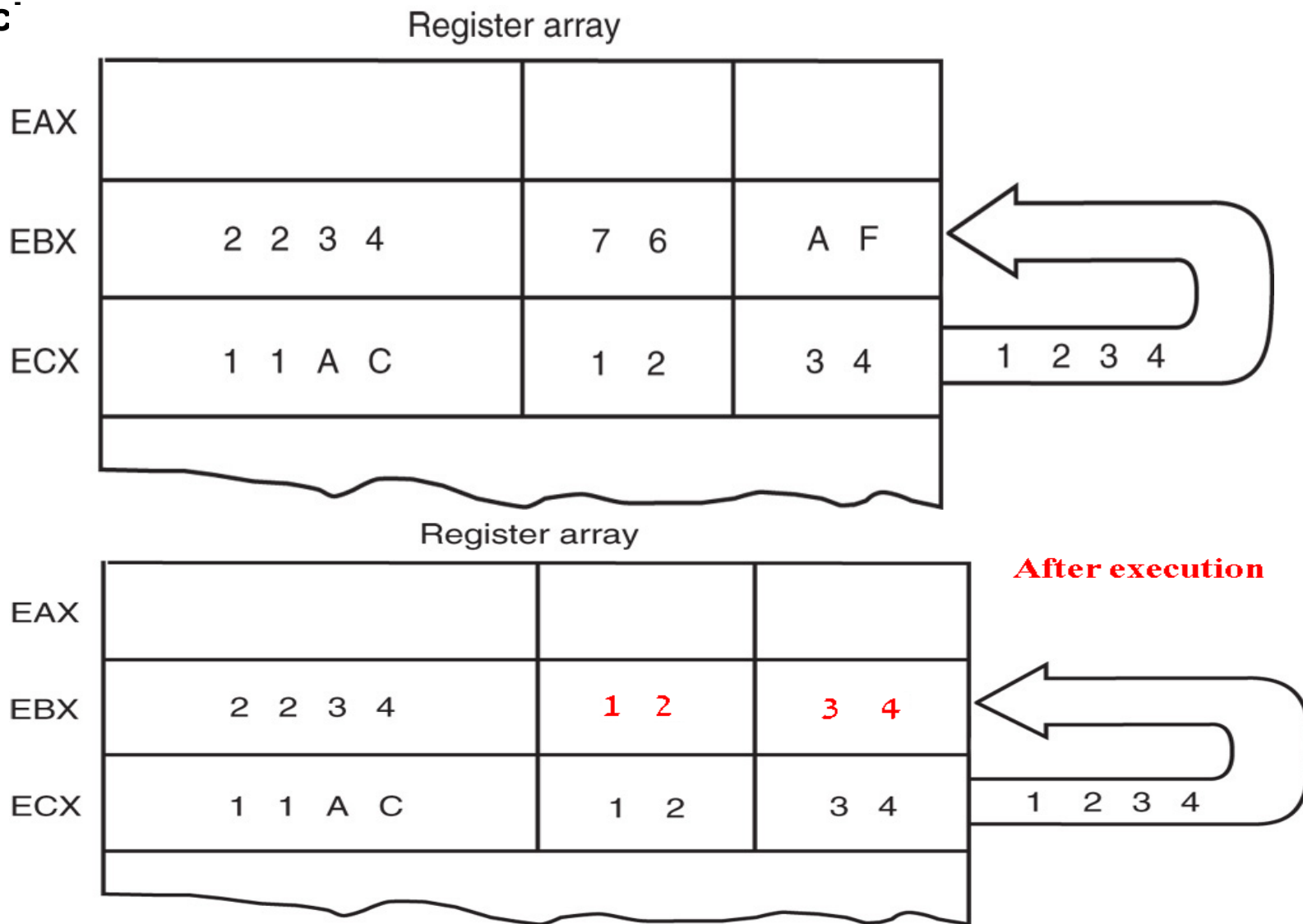
Register Addressing Mode

- In 80386 & above, extended 32-bit register names are: EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI.
- Important for instructions to use registers that are the **same size**.
 - *never* mix an 8-bit with a 16-bit register, an 8- or a 16-bit register with a 32-bit register
 - this is not allowed by the microprocessor and results in an error when assembled

Examples of registered-addressed instructions

<i>Assembly Language</i>	<i>Size</i>	<i>Operation</i>
MOV AL,BL	8 bits	Copies BL into AL
MOV CH,CL	8 bits	Copies CL into CH
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 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 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)

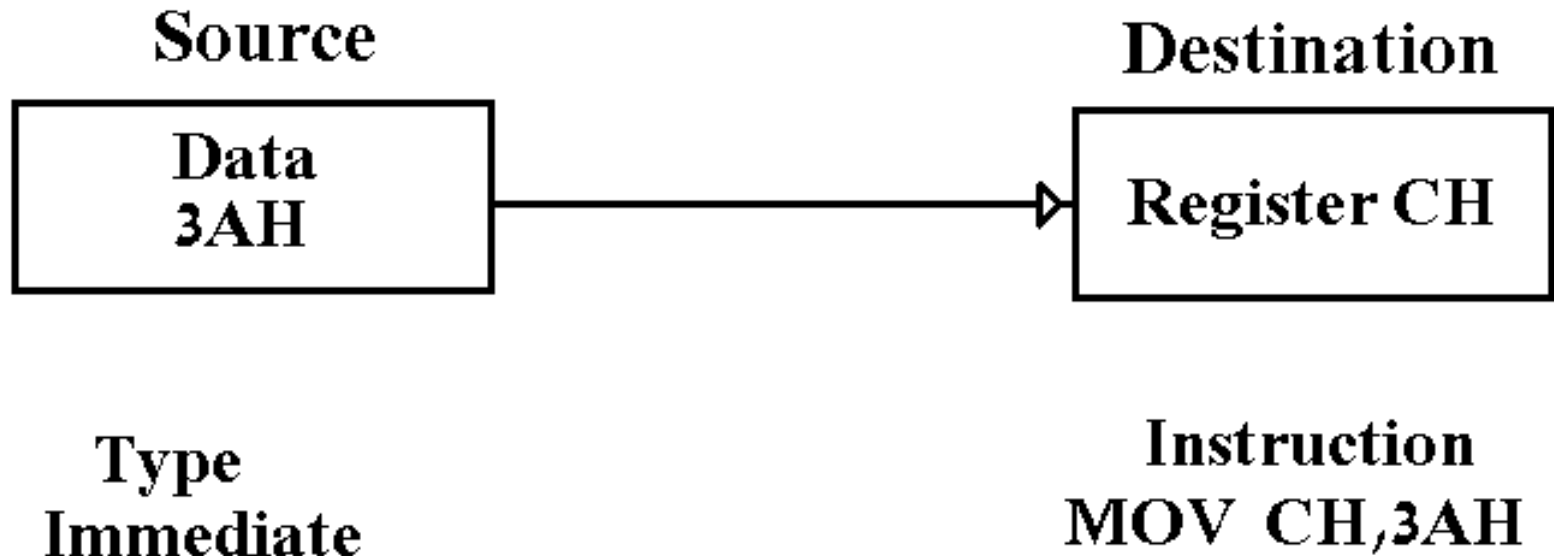
Figure 3–3 The effect of executing the **MOV BX, CX** instruction at the point just before the BX register changes. Note that only the rightmost 16 bits of register EBX c



- Figure 3–3 shows the operation of the MOV BX, CX instruction.
- The **source** register's contents **do not change**.
 - the destination register's contents do change
- The contents of the **destination** register or destination memory location **change** for all instructions **except** the **CMP** and **TEST** instructions.
- The MOV BX, CX instruction **does not affect the leftmost** 16 bits of register EBX.

II- Immediate Addressing

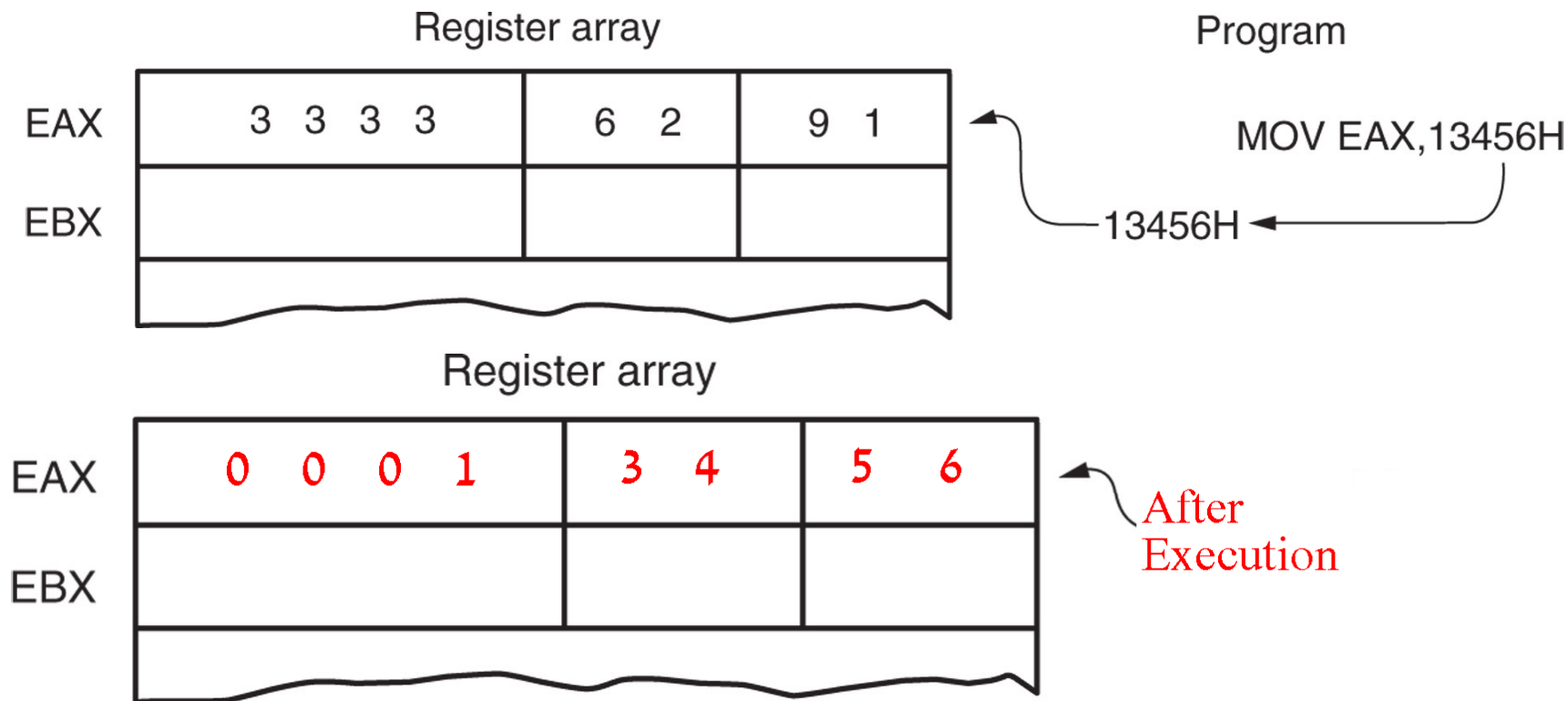
- Term *immediate* implies that **data immediately** follow the hexadecimal opcode in the memory.



Immediate Addressing Mode

- immediate data are **constant** data
- data transferred from a register or memory location are **variable** data
- Immediate addressing operates upon a **byte** or **word** of data.
- Figure 3–4 shows the operation of a MOV EAX,13456H instruction.

Figure 3–4 The operation of the **MOV EAX,13456H** instruction. This instruction copies the immediate data (13456H) into EAX.



- As with the MOV instruction illustrated in Figure 3–3, the source data overwrites the destination data.

- In **symbolic** assembly language, the symbol **#** precedes immediate data in some assemblers.
 - MOV AX,**#**3456H instruction is an example
- Most assemblers **do not** use the **#** symbol, but represent immediate data as in the MOV AX,3456H instruction.
 - an older assembler used with some Hewlett-Packard logic development does, as may others
 - **in this text, the # is not used for immediate data**

- The symbolic assembler portrays immediate data in **many ways**.
- The letter **H** appends hexadecimal data.
- If hexadecimal data begin with a letter, the assembler requires the data start with a **0**.
 - to represent a hexadecimal **F2**, **0F2H** is used in assembly language
- **Decimal** data are represented **as it is** and require no special codes or adjustments.
 - an example is the **100** decimal in the **MOV AL,100** instruction

- An **ASCII**-coded character or characters may be depicted in the immediate form if the ASCII data are enclosed in apostrophes.
 - be careful to use the **apostrophe** (‘) for ASCII data and not the single quotation mark (‘)
- **Binary** data are represented if the binary number is followed by the letter **B**.
 - in some assemblers, the letter **Y**

Examples of immediate addressing using the MOV instruction

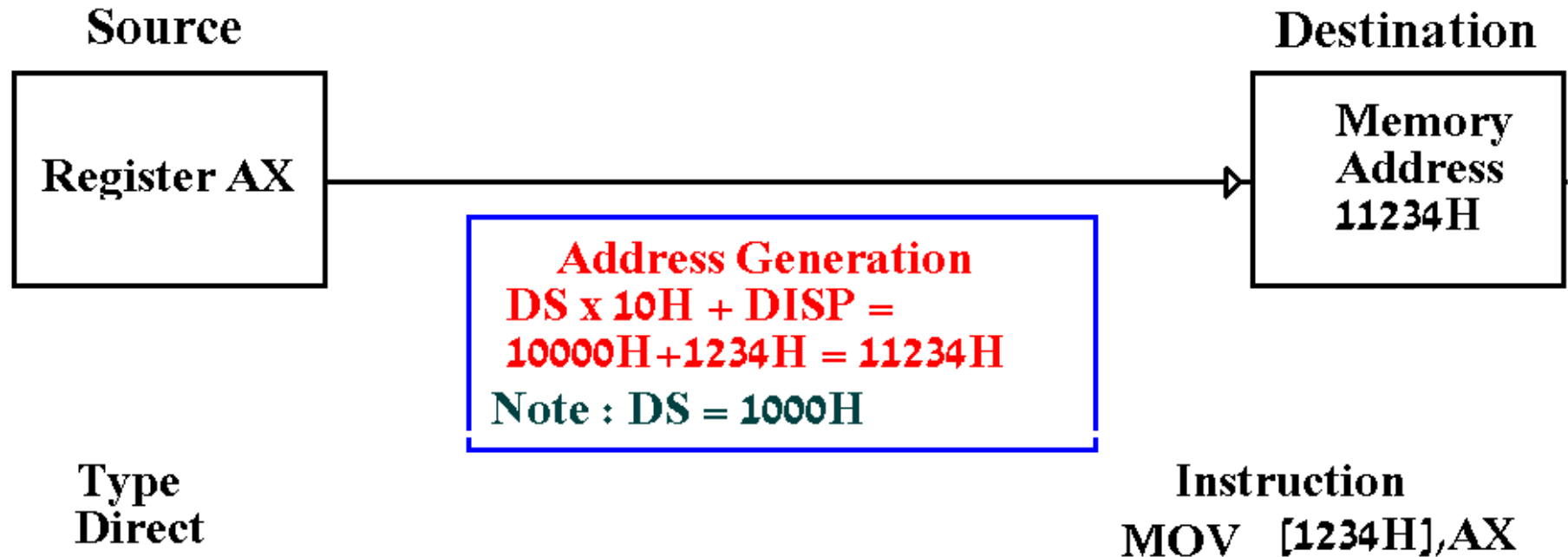
<i>Assembly Language</i>	<i>Size</i>	<i>Operation</i>
MOV BL,44	8 bits	Copies 44 decimal (2CH) into BL
MOV AX,44H	16 bits	Copies 0044H into AX
MOV SI,0	16 bits	Copies 0000H into SI
MOV CH,100	8 bits	Copies 100 decimal (64H) into CH
MOV AL, 'A'	8 bits	Copies ASCII A into AL
MOV AX, 'AB'	16 bits	Copies 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

III- Direct Data Addressing

- Direct addressing moves a **byte** or **word** between a **memory location** and a **register**.
- Applied to many instructions
- **Two** basic forms of direct data addressing:
 1. **Direct addressing**, which applies to a MOV between a memory location and **AL**, **AX**, or **EAX**
 2. **Displacement addressing**, which applies to almost any instruction in the instruction set

- Both forms of addressing are identical except that direct addressing is used to transfer data between **EAX**, **AX**, or **AL** and memory; displacement addressing is used with **any register-memory** transfer.
- Direct addressing requires **3 bytes** of memory, whereas displacement addressing requires **4 bytes**.

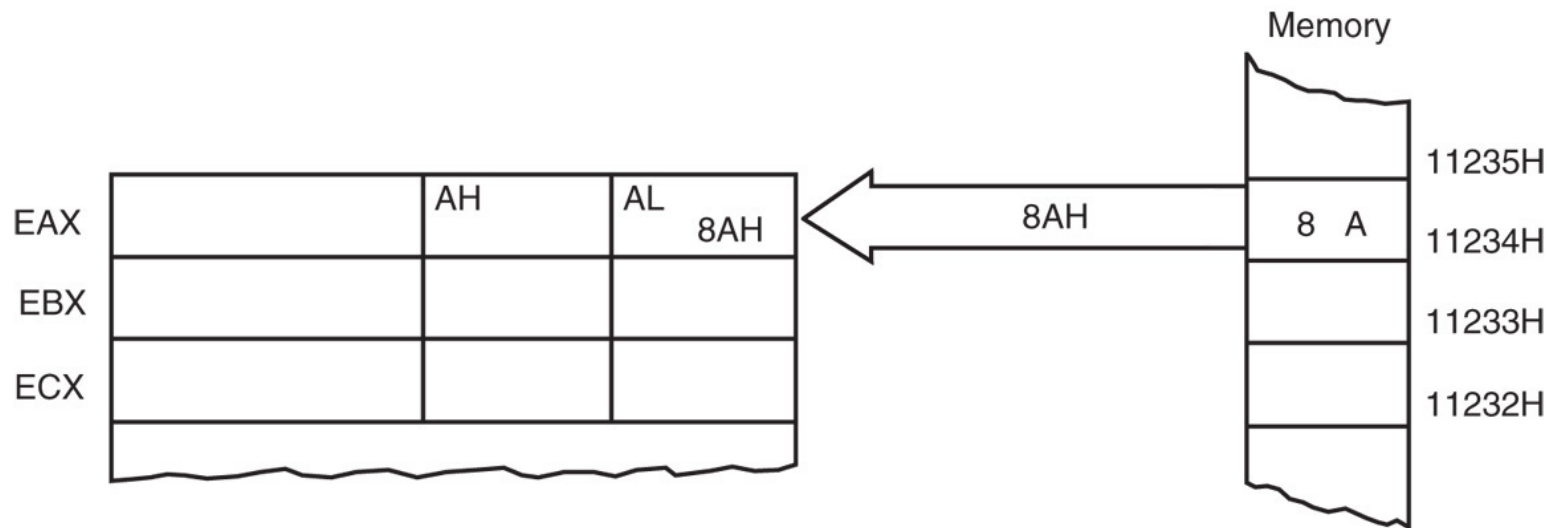
- Address is formed by adding the **displacement** to the default **data segment (DS)** address or an **alternate segment** address.



a. Direct Addressing

- Direct addressing with a MOV instruction transfers data **between** a memory location, located within the **data segment**, and the AL (8-bit), AX (16-bit), or EAX (32-bit) **register**.
 - usually a **3-byte** long instruction
- **MOV AL,DATA** loads **AL** from the data segment memory location **DATA** (1234H).
 - **DATA** is a **symbolic memory** location, while 1234H is the actual hexadecimal location

Figure 3–5 The operation of the **MOV AL,[1234H]** instruction when **DS=1000H** .



- This instruction transfers a copy contents of memory location **11234H** into **AL**.
 - the effective address is formed by adding 1234H (the **offset address**) and 10000H (the **data segment** address of 1000H times 10H) in a system operating in the **real mode**

- **Note the difference between**

MOV AX, 1234H

and

MOV AX,[1234H]

TABLE 3–3 Direct-addressed instructions using EAX, AX, and AL.

<i>Assembly Language</i>	<i>Size</i>	<i>Operation</i>
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

b. Displacement Addressing

- Almost **identical** to direct addressing, except the instruction is **4 bytes** wide instead of **3**.
- In 80386 through Pentium 4, this instruction can be up to **7 bytes** wide if a **32-bit** register and a **32-bit displacement** are specified.
- This type of direct data addressing is much more **flexible** because most instructions use it.

TABLE 3–4 Examples of direct data addressing using a displacement.

<i>Assembly Language</i>	<i>Size</i>	<i>Operation</i>
MOV CH,DOG	8 bits	Copies the byte contents of data segment memory location DOG into CH
MOV CH,DS:[1000H]*	8 bits	Copies the byte contents of data segment memory offset address 1000H into CH
MOV ES,DATA6	16 bits	Copies the word contents of data segment memory location DATA6 into ES
MOV DATA7,BP	16 bits	Copies BP into data segment memory location DATA7
MOV NUMBER,SP	16 bits	Copies SP into data segment memory location NUMBER
MOV DATA1,EAX	32 bits	Copies EAX into data segment memory location DATA1
MOV EDI,SUM1	32 bits	Copies the doubleword contents of data segment memory location SUM1 into EDI

EXAMPLE 3-6

```
                                .MODEL SMALL           ;choose small model
0000                            .DATA                ;start data segment

0000 10                        DATA1 DB 10H         ;place 10H into DATA1
0001 00                        DATA2 DB 0           ;place 00H into DATA2
0002 0000                      DATA3 DW 0           ;place 0000H into DATA3
0004 AAAA                      DATA4 DW 0AAAAH      ;place AAAAH into DATA4

0000                            .CODE                ;start code segment
                                .STARTUP             ;start program

0017 A0 0000 R                 MOV AL,DATA1     ;copy DATA1 into AL
001A 8A 26 0001 R              MOV AH,DATA2     ;copy DATA2 into AH
001E A3 0002 R                 MOV DATA3,AX    ;copy AX into DATA3
0021 8B 1E 0004 R              MOV BX,DATA4     ;copy DATA4 into BX

                                .EXIT                ;exit to DOS
                                END                  ;end program listing
```

- Note the difference in number of bytes used
- The first instruction is direct addressing uses only **3** bytes with **AL** register
- The second instruction is displacement addressing uses **4** bytes with **CL** register

EXAMPLE 3–5

0000	A0	1234	R	MOV AL, DS: [1234H]
0003	BA	0E 1234	R	MOV CL, DS: [1234H]

IV- Register Indirect Addressing

- Allows **data** to be addressed at **any memory location** through an **offset address** held in any of the following registers: **BP**, **BX**, **DI**, and **SI**.
- In addition, 80386 and above **allow register indirect addressing** with any **extended register** except ESP.

MOV [BX],CX

Address = DS x 10H + BX

Memory location = DS x 10 + BX = 01000+0300 = 10300

Source

Register CL

Destination

**Memory
Address
10300H**

Address Generation
DS x 10H + BX =
10000H + 0300H = 10300H
Note : DS = 1000H

Type
Register indirect

Instruction
MOV [BX],CL

Note the difference between

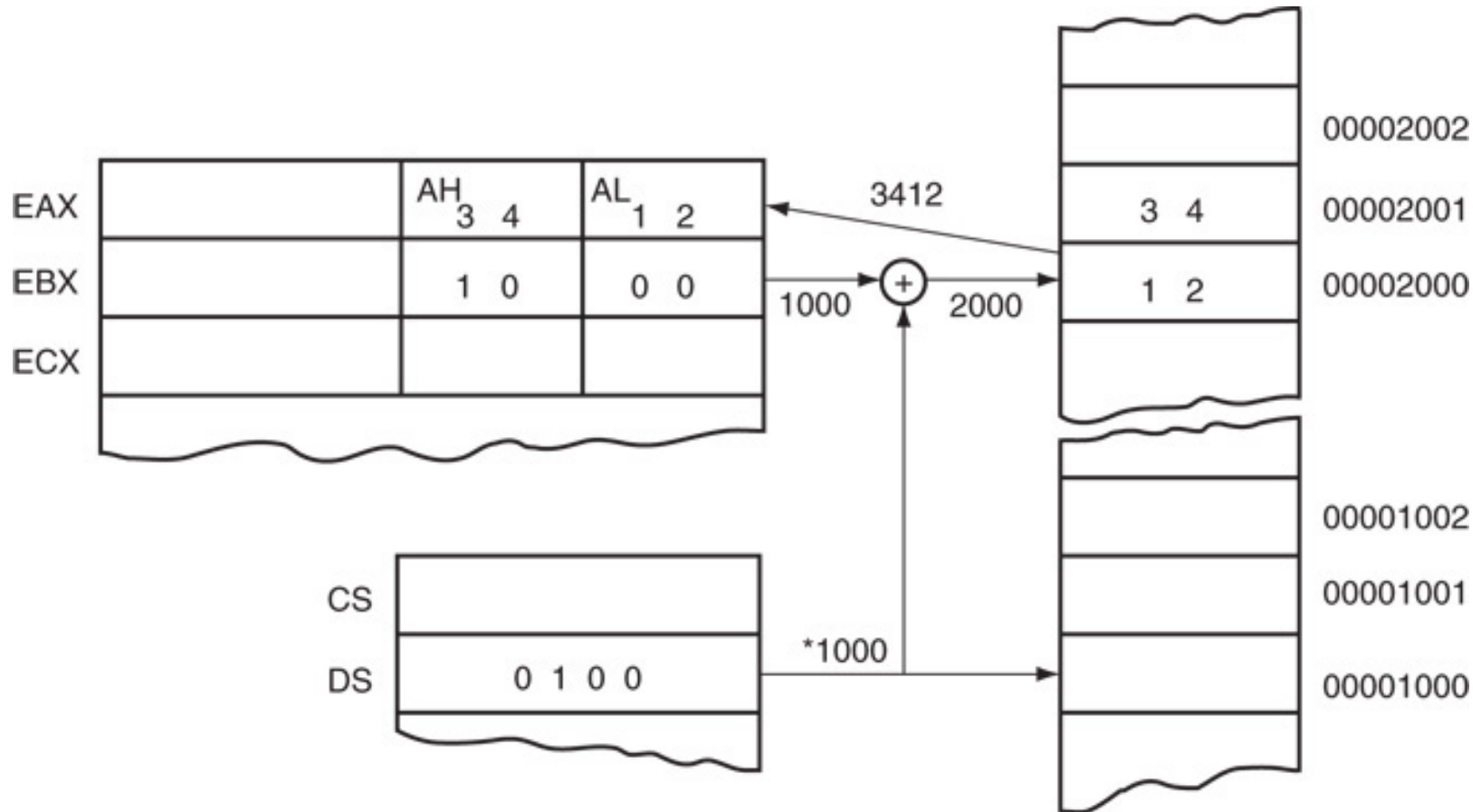
MOV BX , CX

and

MOV [BX],CX

Figure 3–6 The operation of the **MOV AX,[BX]** instruction when **BX = 1000H** and **DS = 0100H**. Note that this instruction is shown after the contents of memory are transferred to AX.

Memory location = DS x 10 + BX = 01000+1000 = 00002000 and 00002001



*After DS is appended with a 0.

- The **data segment (DS)** is used by **default** with register indirect addressing or any other mode that uses **BX, DI, or SI** to address memory.
- If the **BP** register addresses memory, the **stack segment (SS)** is used by default.
 - these settings are considered the **default** for these four index and base registers
- For the 80386 and above, **EBP** addresses memory in the **stack segment** by **default**.
- **EAX, EBX, ECX, EDX, EDI, and ESI** address memory in the data segment by **fault**.

- When using a 32-bit register to address memory in the **real mode**, contents of the register must never exceed 0000FFFFH (**Why?**).
- In the **protected mode**, any value can be used in a 32-bit register that is used to indirectly address memory.
 - as long as it does not access a location outside the segment, dictated by the access rights byte

- Note that the instruction **MOV [DI],10H** is ambiguous.
- In some cases, indirect addressing requires **specifying** the **size** of the data by the **special assembler directive** **BYTE PTR**, **WORD PTR**, **DWORD PTR**, or **QWORD PTR**.
 - these directives indicate the size of the memory data addressed by the memory **pointer** (**PTR**)
- ***The directives are with instructions that address a memory location through a pointer or index register with immediate data.***
- **MOV BYTE PTR[DI],10H is totally clear.**

- Indirect addressing often allows a program to refer to **tabular data** located in memory.
- Figure 3–7 shows the table and the **BX** register used to sequentially address each location in the table.
- To accomplish this task, **load** the **starting location** of the **table** into the **BX** register with a **MOV immediate** instruction.
- After initializing the starting address of the table, use **register indirect addressing** to store the **50 samples sequentially**.

Figure 3–7 An array (TABLE) containing 50 bytes that are indirectly addressed through register BX.

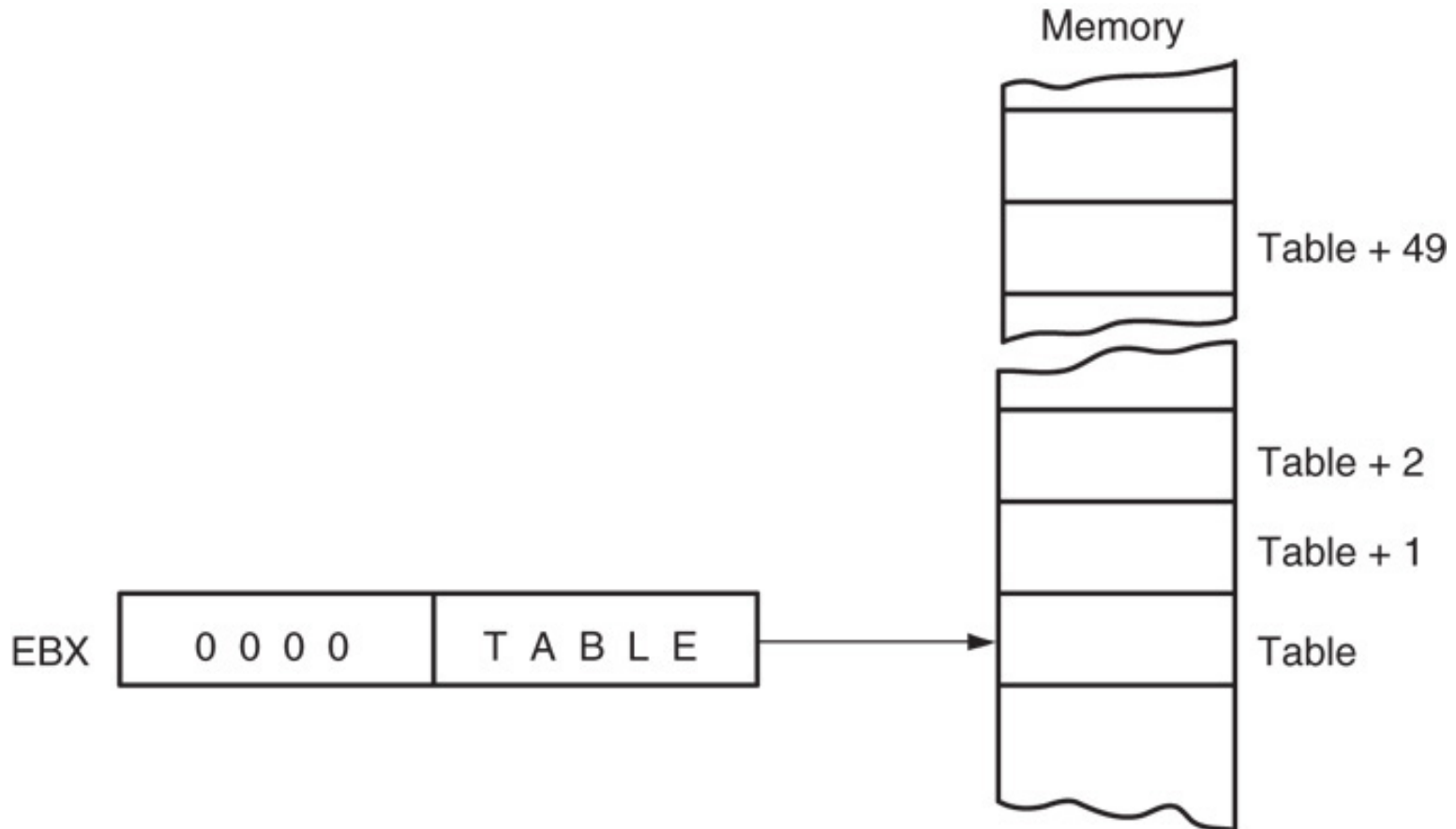


TABLE 3–5 Examples of register indirect addressing.

<i>Assembly Language</i>	<i>Size</i>	<i>Operation</i>
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

*Data addressed by BP or EBP are in the stack segment by default, while other indirect addressed instructions use the data segment by default.

EXAMPLE 3-7

```
                                .MODEL SMALL           ;select small model
0000                            .DATA                 ;start data segment

0000 0032 [                      DATAS DW 50 DUP(?) ;setup array of 50 words
                                0000
                                ]

0000                            .CODE                 ;start code segment
                                .STARTUP              ;start program

0017 B8 0000                    MOV AX,0
001A 8E C0                      MOV ES,AX           ;address segment 0000 with ES
001C B8 0000 R                  MOV BX,OFFSET DATAS ;address DATAS array with BX
001F B9 0032                    MOV CX,50          ;load counter with 50
0022                            AGAIN:
0022 26:A1 046C                 MOV AX,ES:[046CH]  ;get clock value
0026 89 07                      MOV [BX],AX        ;save clock value in DATAS
0028 43                          INC BX             ;increment BX to next element
0029 43                          INC BX
002A E2 F6                      LOOP AGAIN         ;repeat 50 times

                                .EXIT                 ;exit to DOS
                                END                   ;end program listing
```

- The **table** information contains **50** samples (taken from memory location **0000:046C**. Location **0000:046C** contains a counter in **DOS** maintained by the **real-time clock**)
- The **LOOP** instruction **repeats** the loop and **decrements** (subtracts 1 from) 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 instructions ends.

V- Base-Plus-Index Addressing

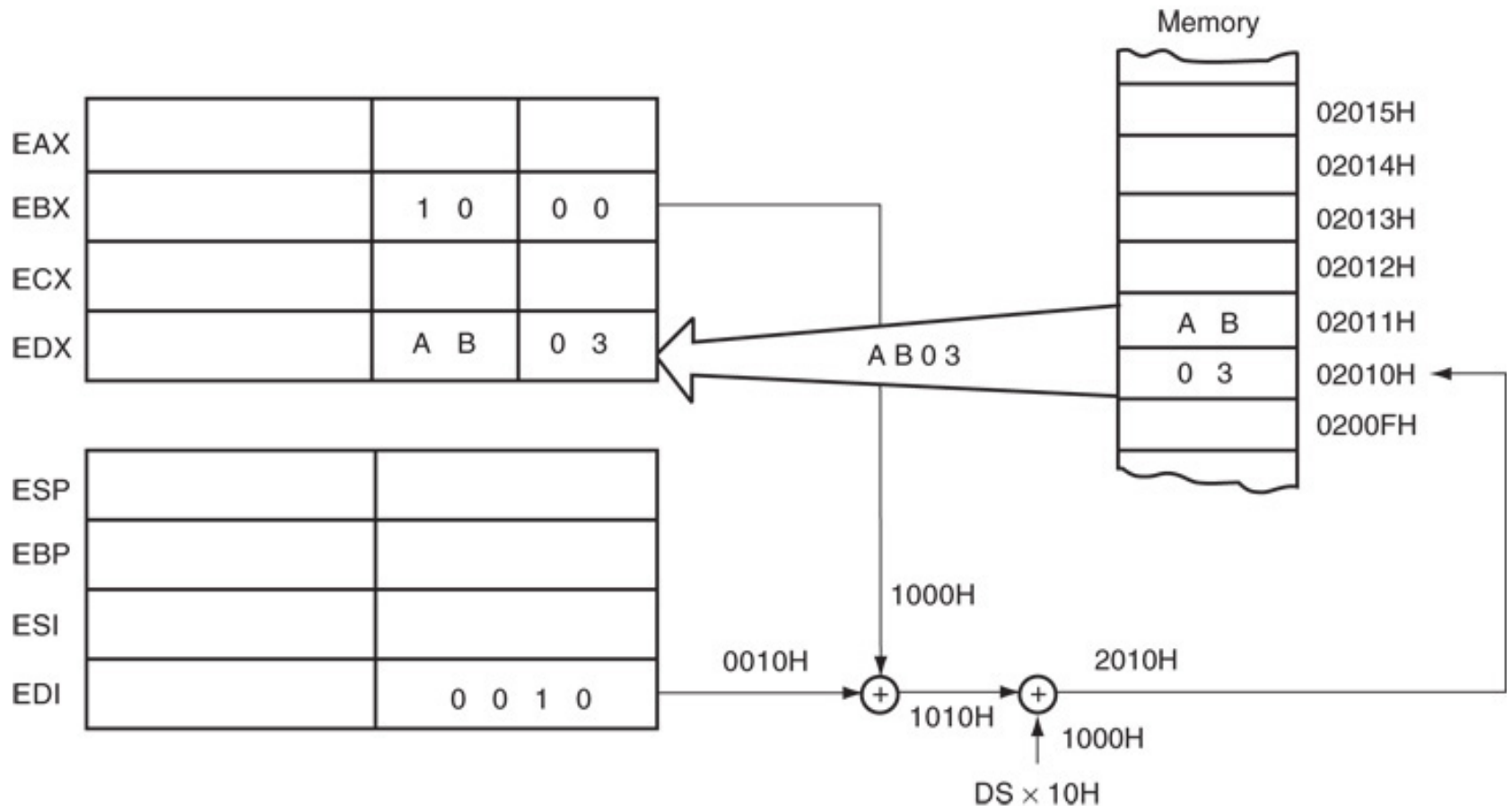
- Similar to indirect addressing because it **indirectly** addresses memory data.
- The **base register** often holds the **beginning** location of a memory **array**.
 - the **index register** holds the **relative position** of an element in the **array**
 - whenever **BP** addresses memory data, **both** the **stack segment** register and **BP** generate the effective address

Locating Data with Base-Plus-Index Addressing

- Figure 3–8 shows how data are addressed by the **MOV DX,[BX + DI]** instruction when the microprocessor operates in the real mode.
- The Intel assembler requires this addressing mode appear as **[BX][DI]** instead of **[BX + DI]**.
- The **MOV DX,[BX + DI]** instruction is **MOV DX,[BX][DI]** for a program written for the Intel ASM assembler.

Figure 3–8 An example showing how the **base-plus-index** addressing mode functions for the **MOV DX,[BX + DI]** instruction. Notice that memory address **02010H** is accessed because **DS=0100H**, **BX=1000H** and **DI=0010H**.

Address = DS x 10 + BX + DI = 01000+01000+0010 = 02010H and 02011H

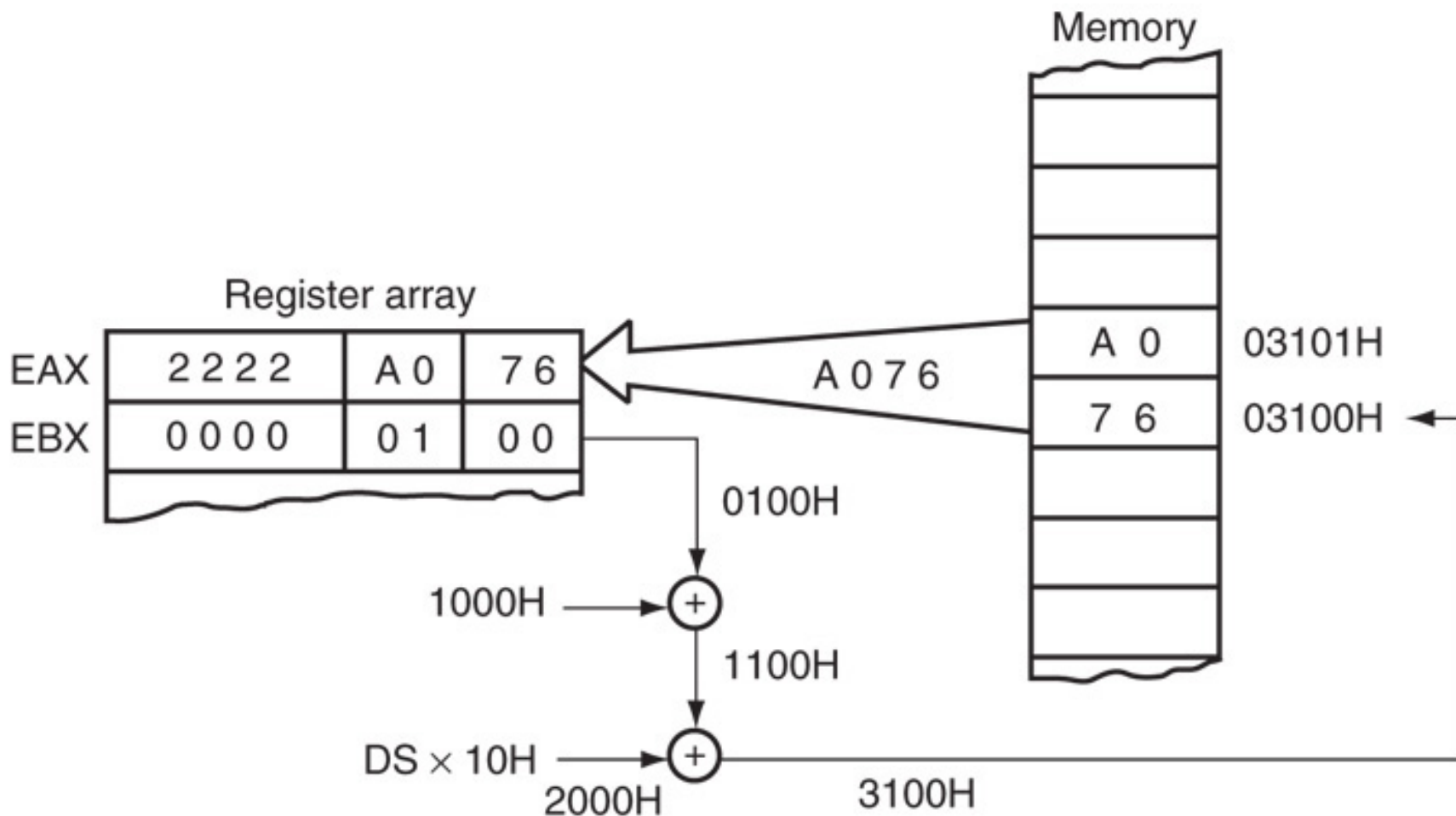


VI- Register Relative Addressing

- **Similar** to base-plus-index addressing and displacement addressing.
 - data in a segment of memory are addressed by **adding** the **displacement** to the contents of a **base** or an **index register** (BP, BX, DI, or SI)
- Figure 3–10 shows the operation of the **MOV AX,[BX+1000H]** instruction.
- A real mode segment is 64K bytes long.

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

Address Generation = DS X 10H + [BX + offset] = 02000+100+1000=3100H and 3101H



VII- Base Relative-Plus-Index Addressing

- Similar to base-plus-index addressing.
 - adds a **displacement**
 - uses a **base register** and an **index register** to form the **memory address**
- This type of addressing mode often addresses a two-dimensional array of memory data.

Addressing Data with Base Relative-Plus-Index

- Least-used addressing mode.
- Figure 3–12 shows how data are referenced if the instruction executed by the microprocessor is **MOV AX,[BX + SI + 100H]**.
 - displacement of **100H** adds to **BX** and **SI** to form the offset address within the data segment
- This addressing mode is too complex for frequent use in programming.

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

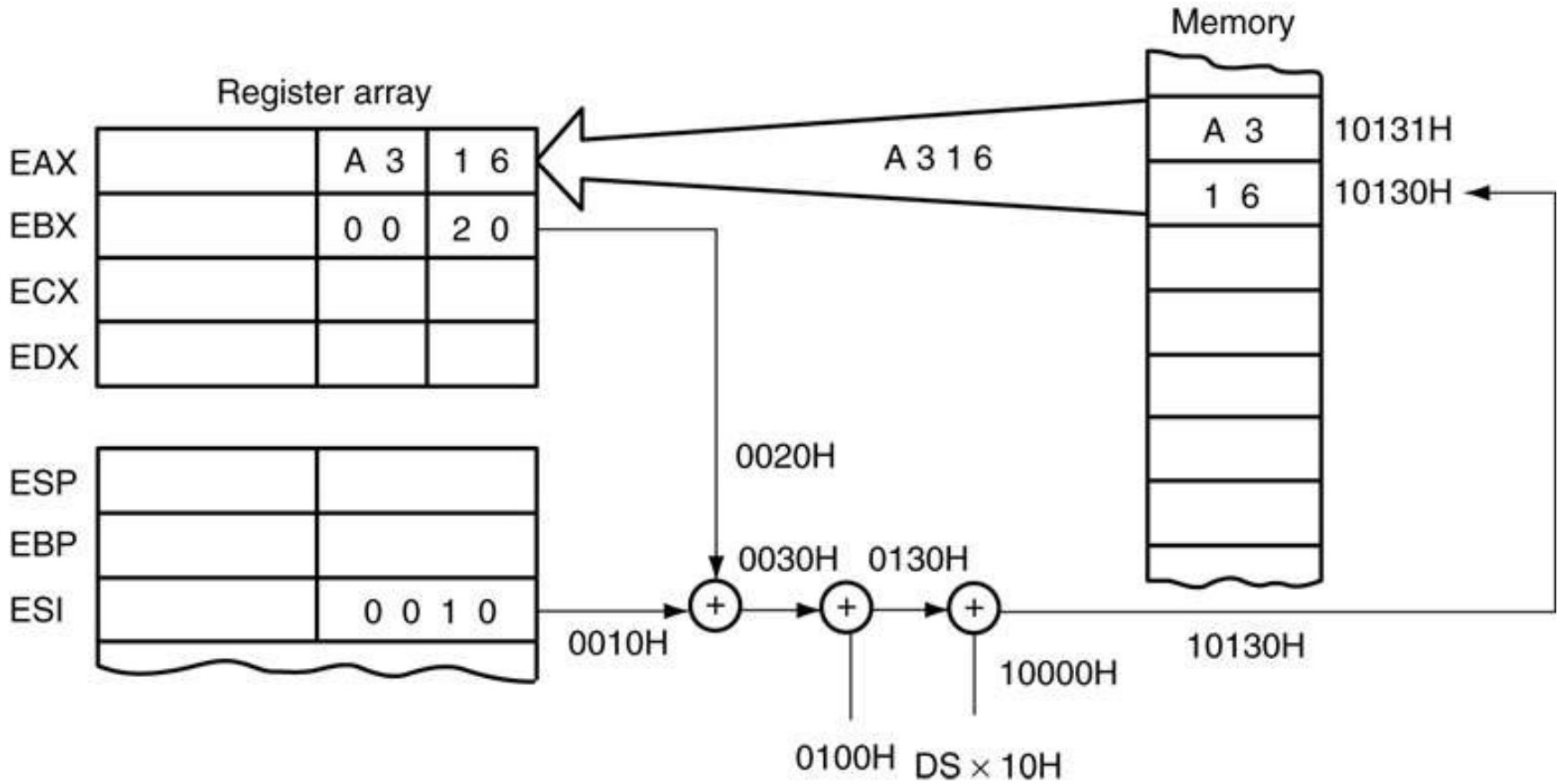


TABLE 3–8 Example base relative-plus-index instructions.

<i>Assembly Language</i>	<i>Size</i>	<i>Operation</i>
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

Scaled-Index Addressing

- Unique to 80386 - Core2 microprocessors.
 - uses two 32-bit registers (a base register and an index register) to access the memory
- The second register (**index**) is **multiplied** by a **scaling factor**.
 - the **scaling factor** can be **1x, 2x, 4x, 8x**
- A scaling factor of 1 is implied and need not be included in the assembly language instruction (**MOV AL,[EBX + ECX]**).

TABLE 3-9 Examples of scaled-index addressing.

<i>Assembly Language</i>	<i>Size</i>	<i>Operation</i>
MOV EAX,[EBX+4*ECX]	32 bits	Copies the doubleword contents of the data segment memory location addressed by the sum of 4 times ECX plus EBX into EAX
MOV [EAX+2*EDI+100H],CX	16 bits	Copies CX into the data segment memory location addressed by the sum of EAX, 100H, and 2 times EDI
MOV AL,[EBP+2*EDI+2]	8 bits	Copies the byte contents of the stack segment memory location addressed by the sum of EBP, 2, and 2 times EDI into AL
MOV EAX,ARRAY[4*ECX]	32 bits	Copies the doubleword contents of the data segment memory location addressed by the sum of ARRAY and 4 times ECX into EAX

3–3 STACK MEMORY-ADDRESSING MODES

- The stack plays an important role in all microprocessors.
 - holds **data** temporarily and stores **return addresses** used by procedures
- Stack memory is LIFO (**last-in, first-out**) memory
 - describes the way data are stored and removed from the stack

- Data are placed on the stack with a **PUSH instruction**; and removed with a **POP instruction**.
- The **Call** instruction also uses the stack to **hold the return address** for the procedures and a **RET** (return) instruction to **remove the return address** from the stack.
- Stack memory is maintained by **two** registers:
 - the **stack pointer** (SP or ESP)
 - the **stack segment** register (SS)
- Whenever a word of data 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**

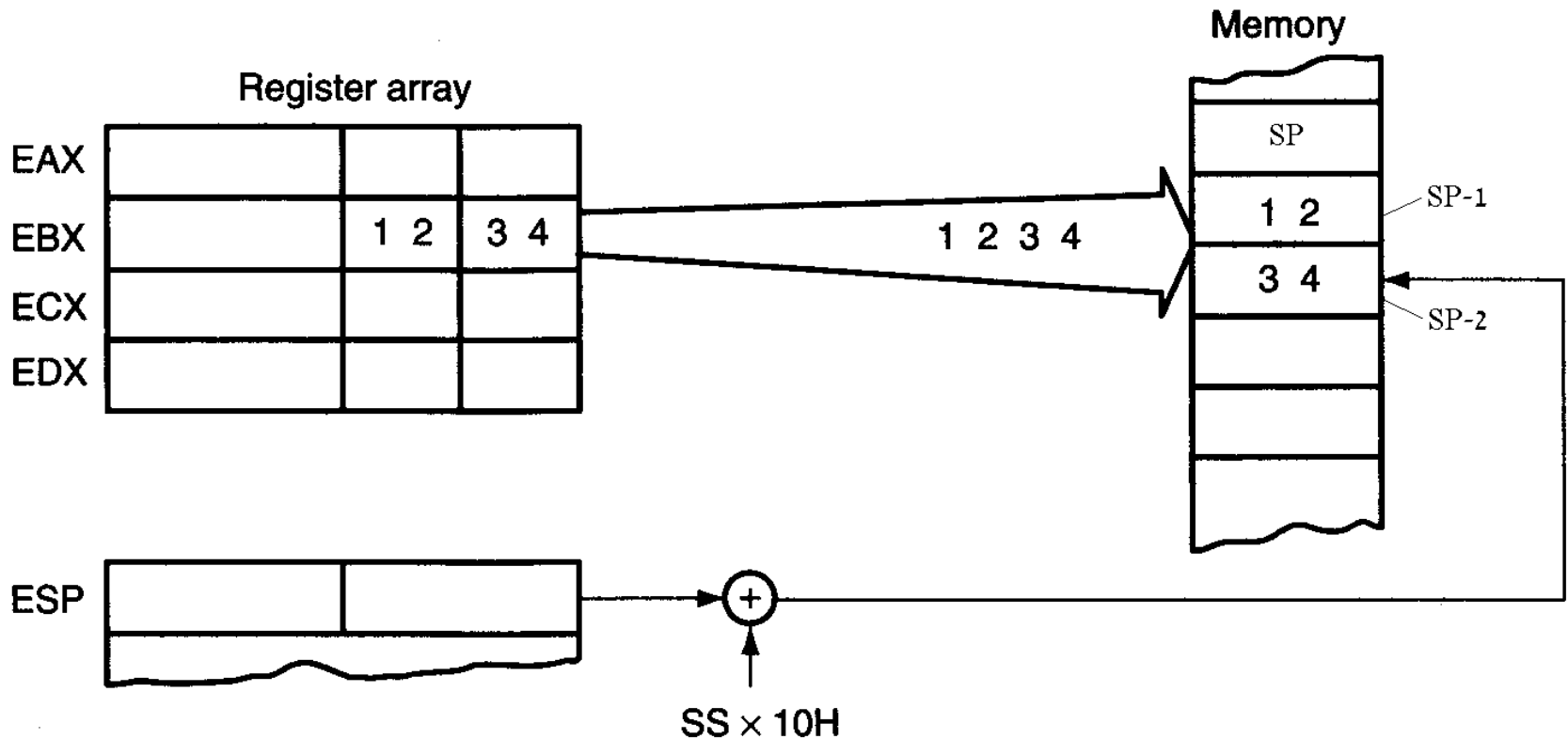
- The **SP** is **decremented** by **2** so the next word is stored in the next available stack location.
 - the **SP/ESP** register always points to an area of memory located within the stack segment.
- In protected mode operation, the **SS** register holds a **selector** that accesses a **descriptor** for the base address of the stack segment.

(a) PUSH BX places the contents of BX onto the stack;

Whenever a word of data 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**

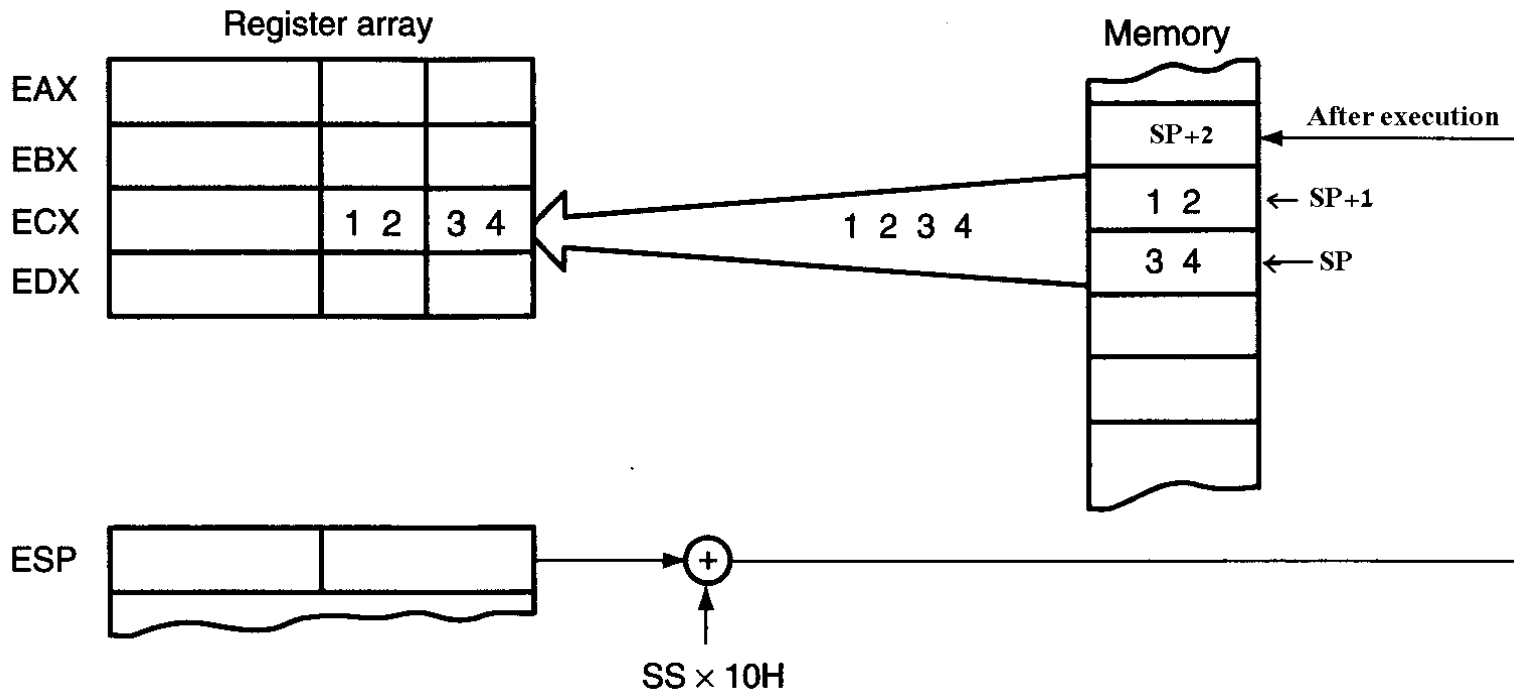
after the data are stored by a PUSH, the contents of the SP register decrement by two



(b) POP CX removes data from the stack and places them into **CX**. Instruction is shown after execution.

When 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 incremented by **2**



- Note that PUSH and POP store or retrieve **words** of data—**never bytes**—in 8086 - 80286.
- 80386 and above allow **words** or **doublewords** to be transferred to and from the stack.
- **Data** may be **pushed** onto the **stack** from **any 16-bit** register or **segment** register.
 - in 80386 and above, from any **32-bit** extended register
- **Data** may be **popped** off the **stack** into **any** register or **any segment** register except CS.

- **PUSHA** and **POPA** instructions push or pop **all** of the **registers**, except segment registers, on the stack.
- These instructions are not available on early 8086/8088 processors.
- 80386 and above allow extended registers to be pushed or popped.
 - 64-bit mode for Pentium and Core2 does **not contain** a **PUSHA** or **POPA** instruction

TABLE 3–11 Example PUSH and POP instructions.

<i>Assembly Language</i>	<i>Operation</i>
POPF	Removes a word from the stack and places it into the flag register
POPFD	Removes a doubleword from the stack and places it into the EFLAG register
PUSHF	Copies the flag register to the stack
PUSHFD	Copies the EFLAG register to the stack
PUSH AX	Copies the AX register to the stack
POP BX	Removes a word from the stack and places it into the BX register
PUSH DS	Copies the DS register to the stack
PUSH 1234H	Copies a word-sized 1234H to the stack
POP CS	This instruction is illegal
PUSH WORD PTR[BX]	Copies the word contents of the data segment memory location addressed by BX onto the stack
PUSHA	Copies AX, CX, DX, BX, SP, BP, DI and SI to the stack
POPA	Removes the word contents for the following registers from the stack: SI, DI, BP, SP, BX, DX, CX, and AX
PUSHAD	Copies EAX, ECX, EDX, EBX, ESP, EBP, EDI, and ESI to the stack
POPAD	Removes the doubleword contents for the following registers from the stack: ESI, EDI, EBP, ESP, EBX, EDX, ECX, and EAX
POP EAX	Removes a doubleword from the stack and places it into the EAX register
PUSH EDI	Copies EDI to the stack

EXAMPLE 3-15

```
0000          .MODEL TINY          ;select tiny model
              .CODE                ;start code segment
              .STARTUP             ;start program
0100 B8 1000  MOV    AX,1000H      ;load test data
0103 BB 2000  MOV    BX,2000H
0106 B9 3000  MOV    CX,3000H

0109 50      PUSH   AX            ;1000H to stack
010A 53      PUSH   BX            ;2000H to stack
010B 51      PUSH   CX            ;3000H to stack

010C 58      POP    AX            ;3000H to AX
010D 59      POP    CX            ;2000H to CBX
010E 5B      POP    BX            ;1000H to BX
              .exit              ;exit to DOS
              end                 ;end program
```

Data may be popped off the stack into any register or any segment register except CS.

Example

If $SS = 3500H$ and the SP is $FFFFEH$,

- (a) Calculate the physical address of the stack. (b) Calculate the lower range.
(c) Calculate the upper range of the stack segment. (d) Show the logical address of the stack.

Solution:

(a) $44FFE$ ($35000 + FFFE$)

(b) 35000 ($35000 + 0000$)

(c) $44FFF$ ($35000 + FFFF$)

(d) $3500:FFFFE$

Example

Assuming that $SP = 1236$, $AX = 24B6$, $DI = 85C2$, and $DX = 5F93$, show the contents of the stack as each of the following instructions is executed:

PUSH AX
PUSH DI
PUSH DX

Solution:

SS:1230

SS:1231

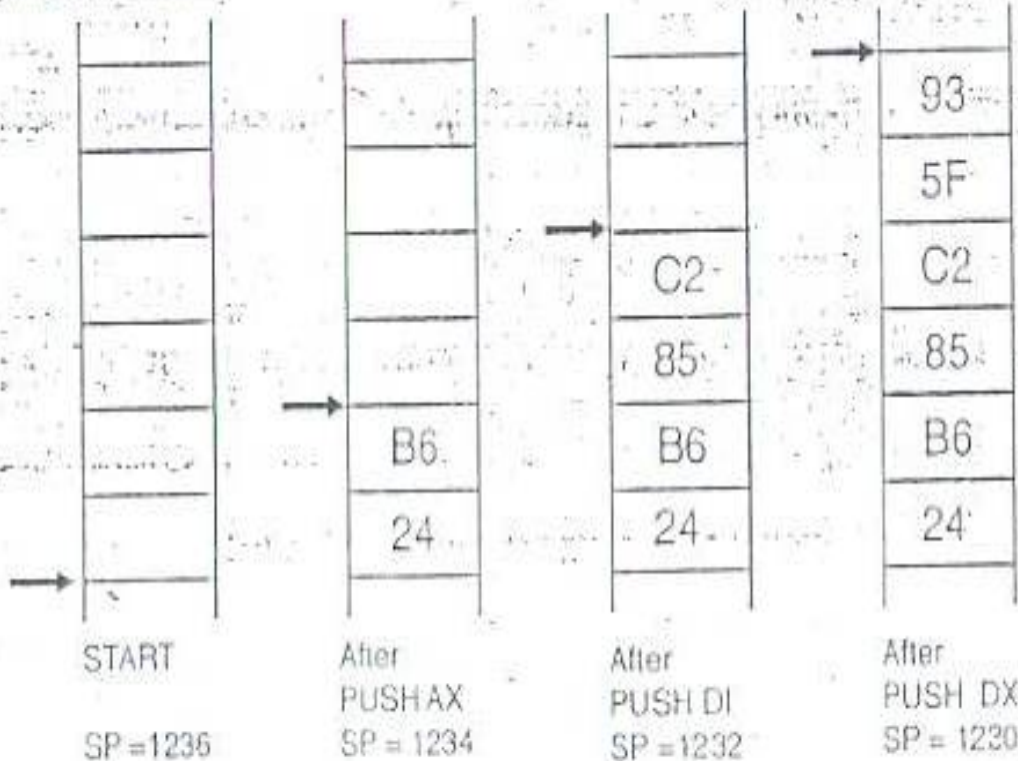
SS:1232

SS:1233

SS:1234

SS:1235

SS:1236

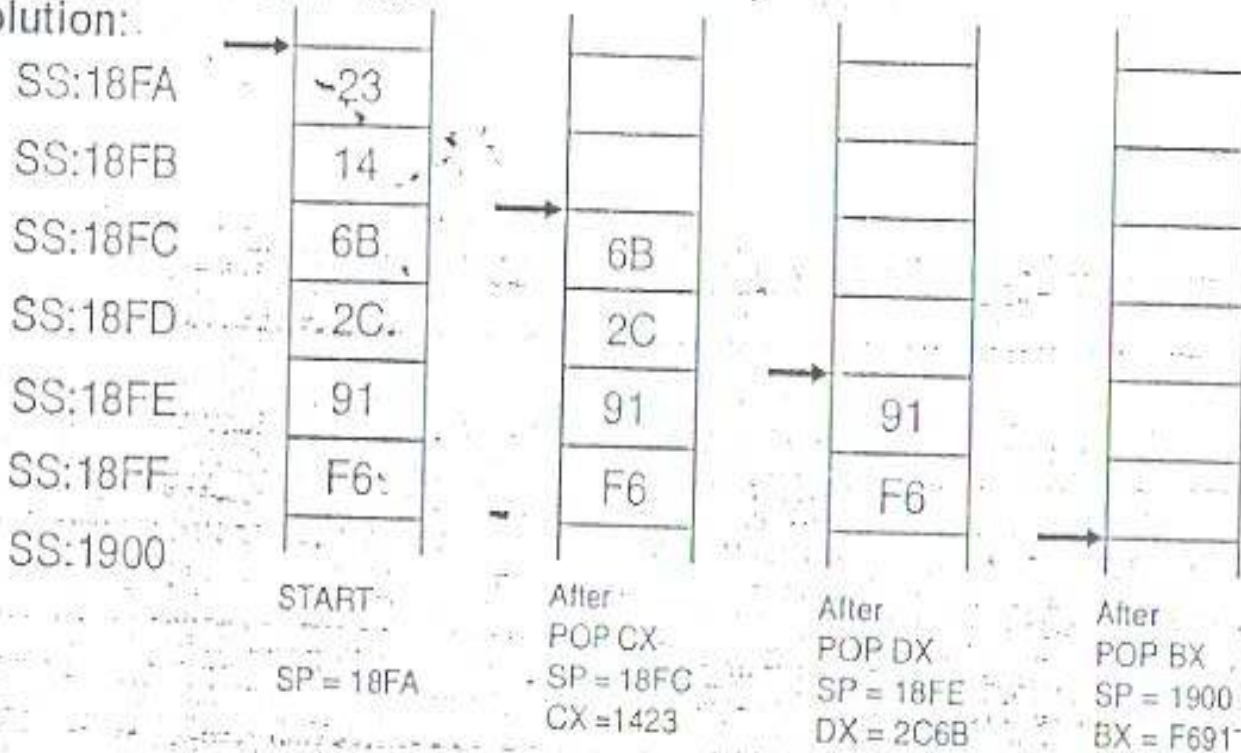


Example

Assuming that the stack is as shown below, and $SP = 18FA$, show the contents of the stack and registers as each of the following instructions is executed:

POP CX
POP DX
POP BX

Solution:



Note about DATA ALLOCATION

Example:

- ❑ To declare and initialize an integer array of 8 elements:
marks DW 0,0,0,0,0,0,0,0
- ❑ What if we want to declare and initialize to zero an array of 200 elements?
- ❑ **Assembler provides a directive to do this (DUP directive)**
 - ❑ **Marks DW 200 DUP (0)**
 - ❑ **Table1 DW 10 DUP(?) ; 10 words uninitialized**
 - ❑ **Name1 DB 30 DUP ('?') ; 30 bytes each initialized to ?**

Multiple initializations

(*cont.*)

- ❑ The DUP directive may also be nested
- ❑ Examples

stars DB 4 DUP (3 DUP (*), 2 DUP (?), 5 DUP (!))

Reserves 40 bytes space and initializes it as

*****??!!!!!!***??!!!!!!***??!!!!!!***??!!!!!!**

matrix DW 10 DUP (5 DUP (0))

Chapter 4

Data Movement Instructions

Note: These slides are prepared from more than a site/reference

Instructor:
Dr. Khalid A. Darabkeh

Introduction

- ❑ This chapter concentrates on the data movement instructions.
- ❑ The data movement instructions include MOV, MOVSX, MOVZX, PUSH, POP, BSWAP, XCHG, XLAT, IN, OUT, LEA, LDS, LES, LFS, LGS, LSS, LAHF, SAHF.
- ❑ String instructions: MOVS, LODS, STOS, INS, and OUTS.

Instruction Format

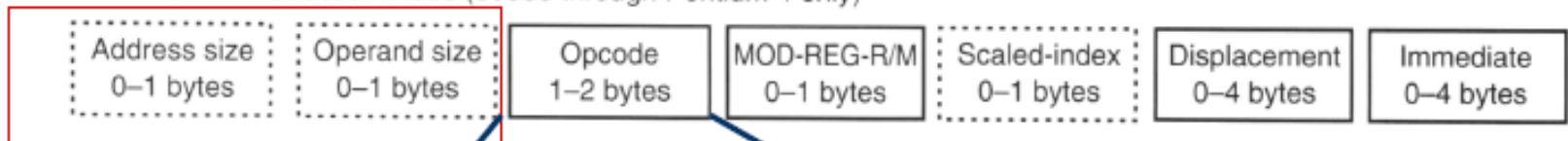
16-bit instruction mode



(a)

Override prefixes

32-bit instruction mode (80386 through Pentium 4 only)

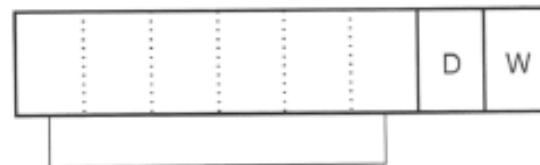


(b)

FIGURE 4-1 The formats of the 8086-Pentium 4 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.

Opcode



Opcode

D – Data flow
 D=1; R/M → REG
 D=0; REG → R/M
W – Weight
 W=0; 8-bit
 W=1; 16 or 32 bit

Register-size Prefixes

- **Default size**
 - 16-bit instruction mode (either real or protected mode) uses 8- and 16-bit register and addressing modes by default
 - 32-bit instruction mode (protected mode only) uses 8- and 32-bit register and addressing modes by default
- **Toggle register size**
 - operate in 16-bit instruction mode and
 - a 16-bit register is used
 - ⇒ register-size prefix is absent
 - a 32-bit register is used
 - ⇒ register-size prefix (66H) is appended
 - operate in 32-bit instruction mode and
 - a 32-bit register is used
 - ⇒ register-size prefix is absent
 - a 16-bit register is used
 - ⇒ register-size prefix (66H) is appended

The address size-prefix (67H) is used in a similar fashion.

Byte 1 : The Opcode



- **Opcode:** selects the operation performed by the μP
 - 1 or 2 bytes long for most (*not all*) machine instructions
 - addition, subtraction, move, and so on
 - **Direction (D) of data flow**
 - D=0: REG field \rightarrow R/M field
 - D=1: REG field \leftarrow R/M field
 - **Word (W) flag:** whether the data are a byte or others
 - W=0: byte
 - W=1:
 - below 80386 (16-bit instruction mode): word
 - In 80386 and above (32-bit instruction or protected mode) :
 - word (if with register-size prefix, 66H)
 - double word (if no register-size prefix)
-

Byte 2 : MOD + REG + R/M



MOD(mode), REG(register) and R/M(register/memory)

- MOD: specify the addressing mode

- All 8-bit displacements are sign-extended into 16-bit displacements (length=2 bytes)
 - 00H-7FH (positive) → 0000H-007FH
 - 80H-FFH (negative) → FF80H-FFFFH

16-bit instruction mode

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

32-bit instruction mode (80386–Pentium 4)

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

REG and R/M (when MOD = 11)

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

- REG and R/M register assignments

Figure 4-4: MOV BP, SP (=8BEC)



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 Byte 1 and 2 formats from Figures 4-2 and 4-3. This instruction is a MOV BP,SP.

References

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

REG and R/M (when MOD = 11)			
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

16-bit R/M Memory-Addressing Modes

- MOD \neq 11
 - MOD=00, and R/M=101
 - [DI]
 - MOD=01 or 10, and R/M=101
 - [DI + displacement]

MOD	REG	R/M
...

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]

*Note: See text section, Special Addressing Mode.

- Special addressing mode: only a displacement
 - MOD=00 and R/M=110 (no displacement and SS:[BP]*)
since we cannot use addressing mode [BP] without a displacement

Figure 4-5: MOV DL, [DI] (=8A15)



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.

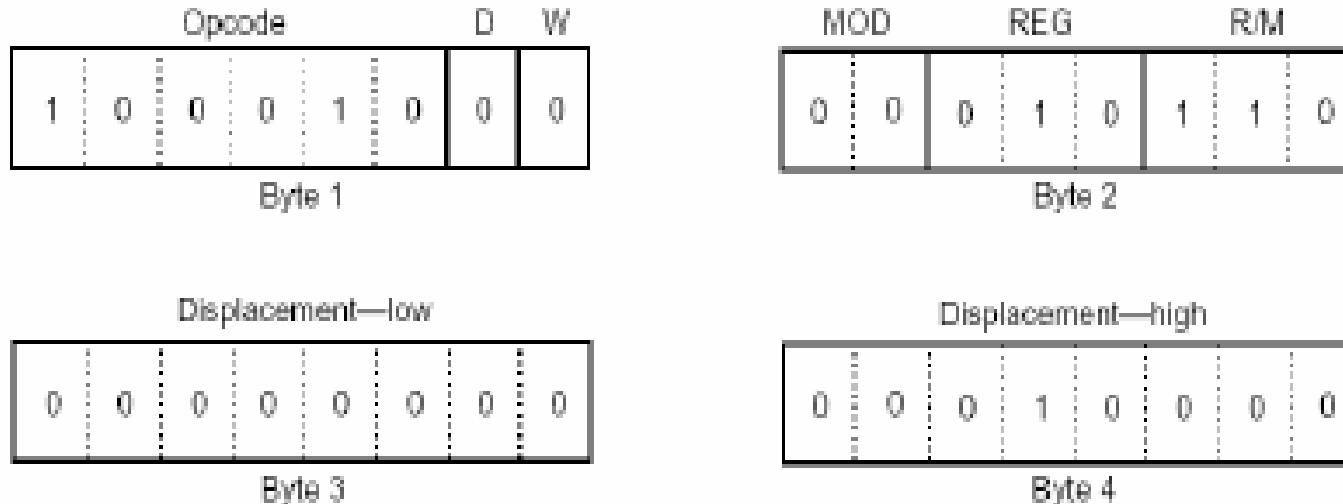
References

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

REG		R/M Code	Addressing Mode
Code	W = 0 (Byte)		
000	AL	000	DS:[BX+SI]
001	CL	001	DS:[BX+DI]
010	DL	010	SS:[BP+SI]
011	BL	011	SS:[BP+DI]
100	AH	100	DS:[SI]
101	CH	101	DS:[DI]
110	DH	110	SS:[BP]*
111	BH	111	DS:[BX]

*Note: See text section, Special Addressing Mode.

Figure 4-6: MOV [1000H], DL (=88161000H)



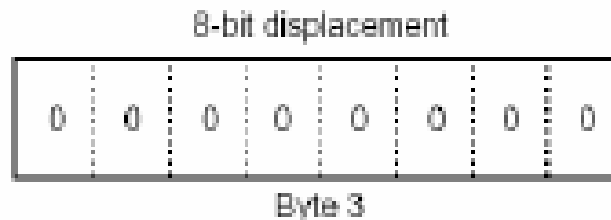
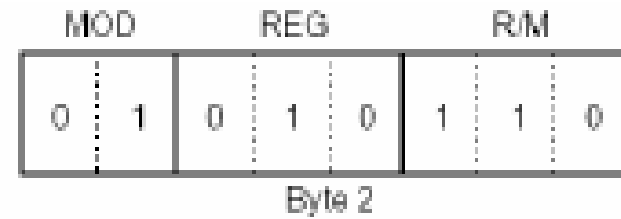
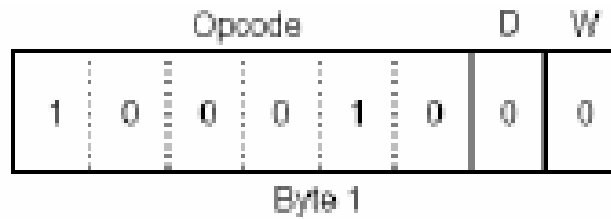
Opcode = MOV
D = Transfer from register (REG)
W = Byte
MOD = because R/M is [BP] (special addressing)
REG = DL
R/M = DS:[BP]
Displacement = 1000H

Special addressing mode:

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

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

Figure 4-7: MOV [BP], DL (=885600H)



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

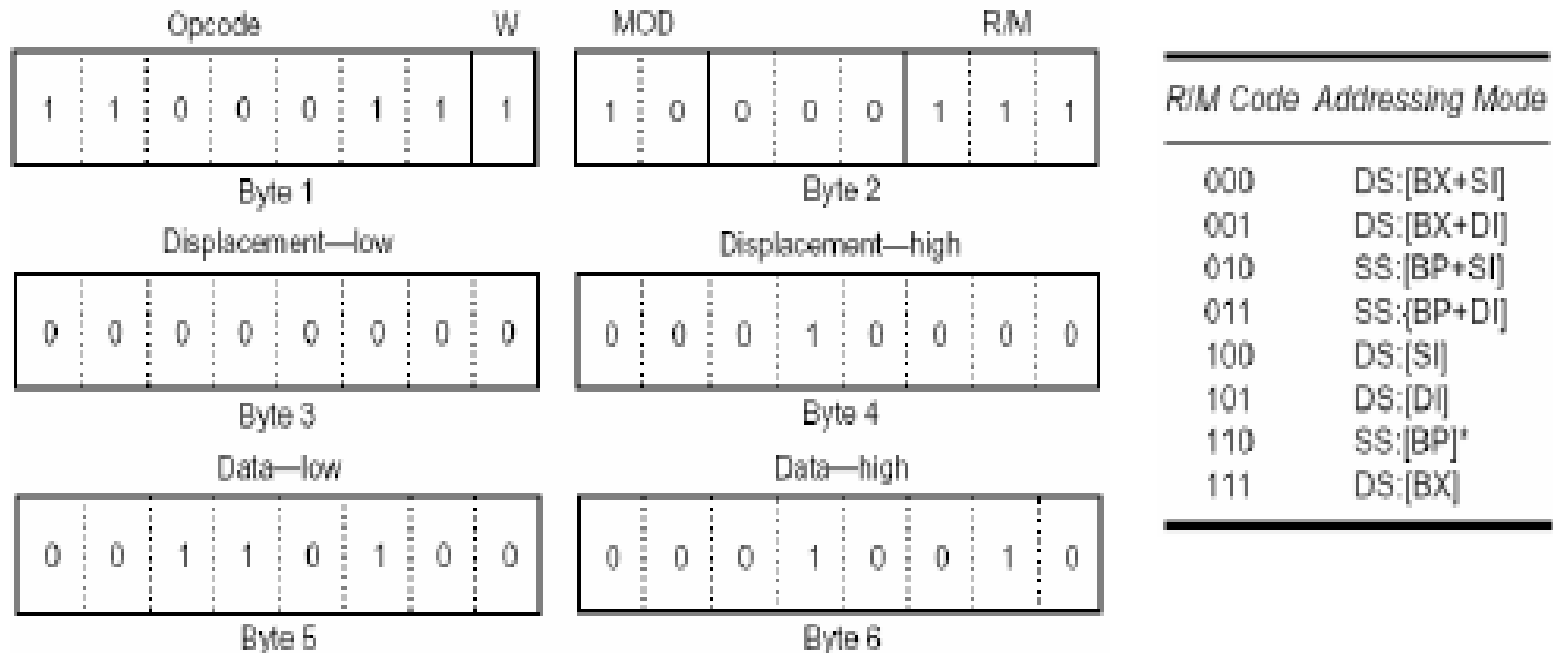
Special addressing mode:

You cannot actually use address mode [BP] without a displacement in machine language.

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

Immediate instruction

- Figure 4-9: MOV WORD PTR [BX+1000H], 1234H



Opcode = MOV (immediate)

W = Word

MOD = 16-bit displacement

REG = 000 (not used in immediate addressing)

R/M = DS:[BX]

Displacement = 1000H

Data = 1234H

Segment MOV Instructions

- a special set of register bits (REG) selects the segment register

TABLE 4-6 Segment register selection.

Note: MOV CS,R/M(16) and POP CS are not allowed by the microprocessor. The FS and GS segments are only available to the 80386–Pentium 4 microprocessors.

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

- Figure 4-10: MOV BX, CS (8C CBH)



Opcode = MOV
MOD = R/M is a register
REG = CS
R/M = BX

FIGURE 4-10 A MOV BX,CS instruction converted to binary machine language.

32-bit Addressing Mode

- 32-bit instruction mode, or 16-bit instruction mode by using address-size prefix 67H

- Example: 80386 and above operated in the 16-bit instruction
`MOV EAX, [EBX+4*ECX]`
`= 67 66 8B 04 8B H`

67H: address size

66H: register size

8BH: opcode=100010, D=1, W=1

04H: MOD=00, REG=000, R/M=100

8BH: ss=10, index=001, Base=011

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]

*Note: See text section, Special Addressing Mode.



index and base both contain register numbers

ss
 00 = ×1 10 = ×4
 01 = ×2 11 = ×8

W = 1 (Doubleword)
EAX
ECX
EDX
EBX
ESP
EBP
ESI
EDI

4-3 Load Effective Address

TABLE 4-9 Load-effective address instructions.

<i>Assembly Language</i>	<i>Operation</i>
LEA AX,NUMB	Loads AX with the address of NUMB
LEA EAX,NUMB	Loads EAX with the address of NUMB
LDS DI,LIST	Loads DS and DI with the 32-bit contents of data segment memory location LIST
LDS EDI,LIST	Loads DS and EDI with the 48-bit contents of data segment memory location LIST
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 memory location MEM

LDS, LES, LFS, LGS and LSS

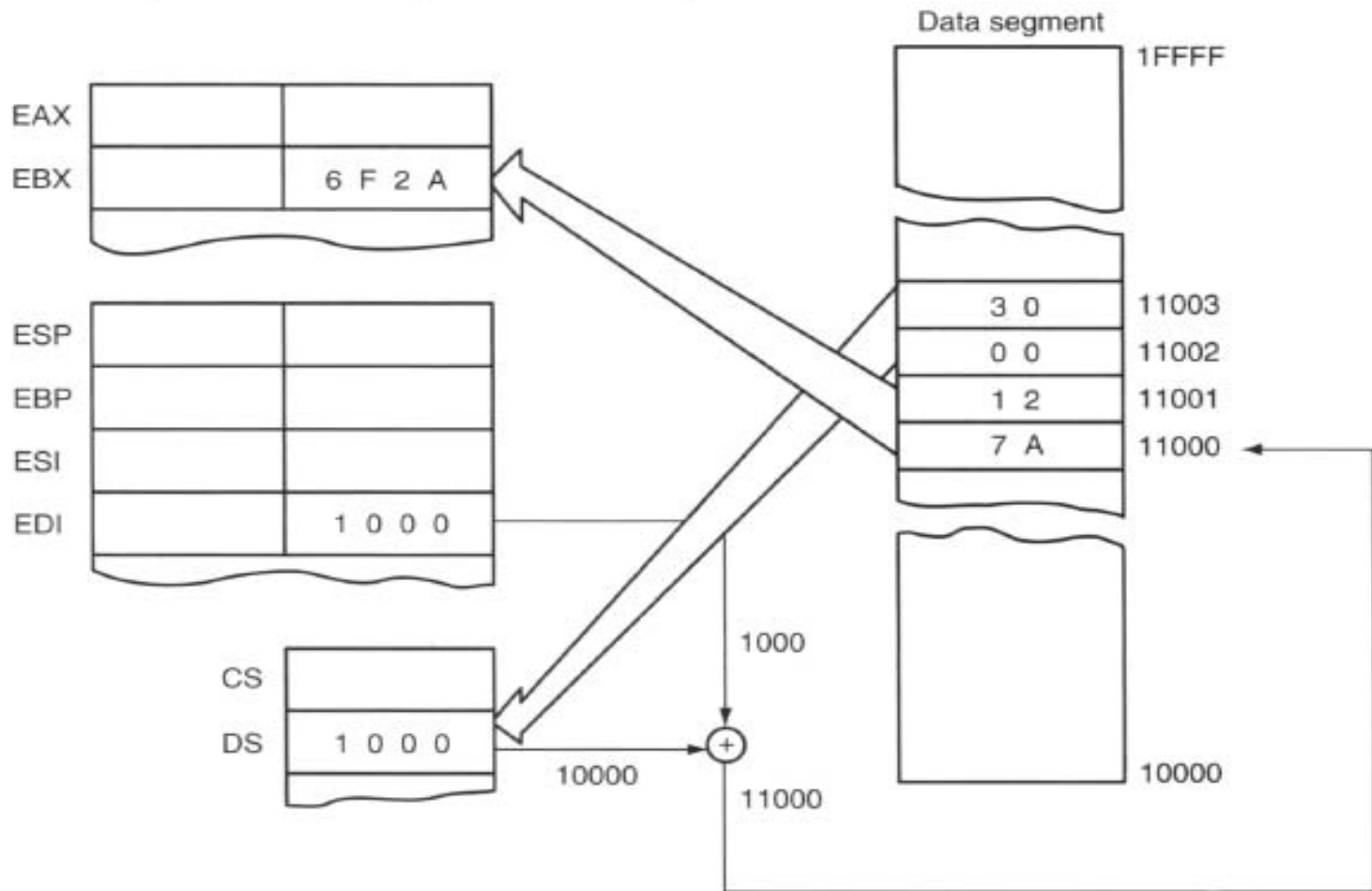


FIGURE 4-15 The `LDS BX,[DI]` instruction loads register BX from addresses 11000H and 11001H and register DS from locations 11002H and 11003H. This instruction is shown at the point just before DS changes to 3000H and BX changes to 127AH.

4-4 STRING DATA TRANSFERS

- Five string data transfer instructions: LODS, STOS, MOVS, INS, and OUTS.
- Each allows data transfers as a single byte, word, or doubleword.
- Before the string instructions are presented, the operation of the D flag-bit (direction), DI, and SI must be understood as they apply to the string instructions.

The Direction Flag

- The **direction flag** (D, located in the **flag register**) selects the **auto-increment** or the **auto-decrement** operation **for the DI and SI** registers during string operations.
 - used only with the string instructions
- The **CLD** instruction **clears the D flag** and the **STD** instruction **sets it**.
 - **CLD** instruction selects the **auto-increment** mode and **STD** selects the **auto-decrement** mode

DI and SI

- During execution of string instruction, memory accesses occur through DI and SI registers.
 - DI offset address accesses data in the **extra segment** for all string instructions that use it
 - SI offset address accesses data by default in the **data segment**
- Operating in 32-bit mode EDI and ESI registers are used in place of DI and SI.
 - this allows string using any memory location in the entire 4G-byte protected mode address space

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
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)

Note: The segment can be overridden with a segment override prefix as in LODS ES:DATA4.

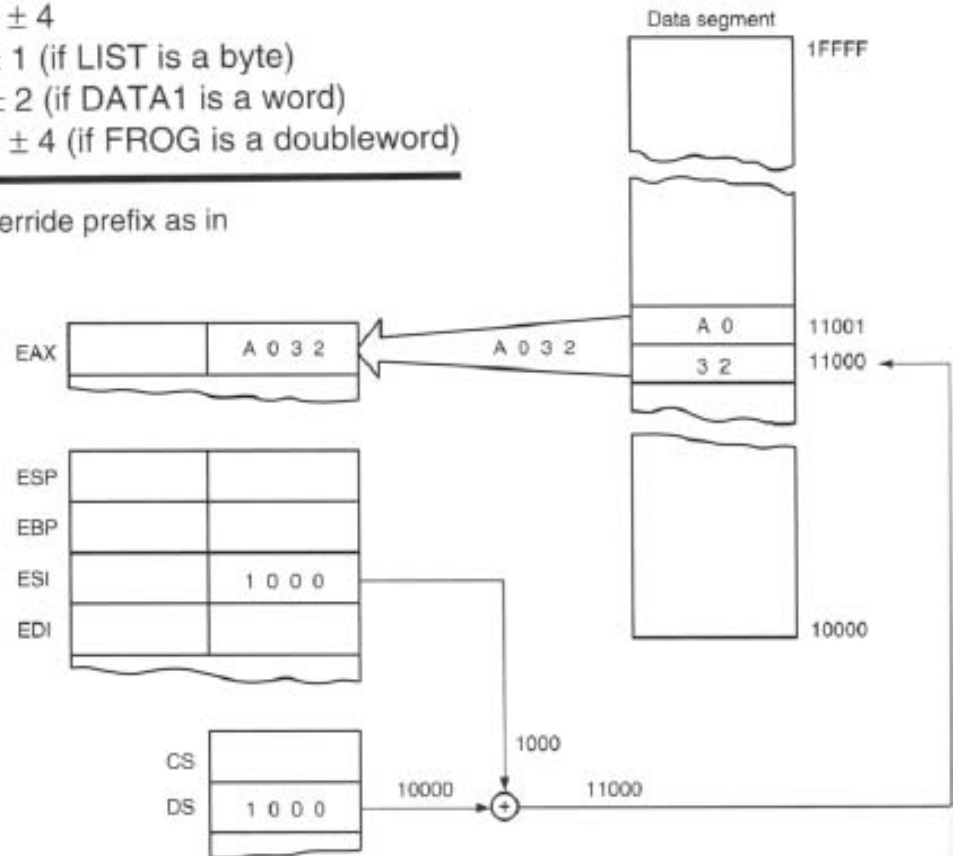


FIGURE 4-16 The operation of the LODSW instruction if DS = 1000H, D = 0, 11000H = 32, and 11001H = A0. This instruction is shown after AX is loaded from memory, but before SI increments by 2.

STOS

- Stores AL, AX, or EAX at the extra segment memory location addressed by the DI register.
- **STOSB (stores a byte)** stores the byte in AL at the extra segment memory location addressed by DI. $DI=DI\pm 1$
- **STOSW (stores a word)** stores AX in the memory location addressed by DI. $DI=DI\pm 2$
- After the byte (AL), word (AX), or doubleword (EAX) is stored, contents of DI increment or decrement.

STOS with a REP

- The **repeat prefix (REP)** is added to any string data transfer instruction except **LODS**. 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** decrements, the string instruction repeats
- If **CX** reaches a value of **0**, the instruction **terminates** and the program continues.
- If **CX** is loaded with **100** and a REP **STOSB** instruction executes, the microprocessor automatically **repeats the STOSB 100** times.

STOS: summary

STOS Instruction

<i>Assembly Language</i>	<i>Operation</i>
STOSB	ES:[DI] = AL; DI = DI \pm 1
STOSW	ES:[DI] = AX; DI = DI \pm 2
STOSD	ES:[DI] = EAX; DI = DI \pm 4
STOS LIST	ES:[DI] = AL; DI = DI \pm 1 (if list is a byte)
STOS DATA3	ES:[DI] = AX; DI = DI \pm 2 (if DATA3 is a word)
STOS DATA4	ES:[DI] = EAX; DI = DI \pm 4 (if DATA4 is a doubleword)

MOVS

- Transfers a byte, word, or doubleword from data segment addressed by SI to extra segment location addressed by DI.
 - pointers are incremented or decremented, as dictated by the direction flag
- Only the source operand (SI), located in the data segment may be overridden so another segment may be used.
- The destination operand (DI) must always be located in the extra segment.
- The only memory-to-memory transfer allowed.

- **MOVS** transfers **byte** from data segment to extra segment.
- **MOVSW** transfers **word** from data segment to extra segment.

MOVS: Summary

MOVS Instructions

- The only instruction for memory to memory movement

TABLE 4–13 Forms of the MOVS instruction.

<i>Assembly Language</i>	<i>Operation</i>
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)
MOVS BYTE1,BYTE2	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (if BYTE1 and BYTE2 are bytes)
MOVS WORD1,WORD2	ES:[DI] = DS:[SI]; DI = DI ± 2, SI = SI ± 2 (if WORD1 and WORD2 are words)
MOVS DWORD1, DWORD2	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4 (if DWORD1 and DWORD2 are doublewords)

Example

- DATSEG segment
- DATA1 db 'ABCDEFGHJKLMNOPQRST'
- DATA2 db 20 DUP(?)
- DATSEG ENDS

- CODE_SEG SEGMENT
- ASSUME CS:CODE_SEG, DS:DATSEG
- MAIN PROC FAR
- MOV AX, DATSEG
- MOV ES, AX
- MOV DS, AX
- CLD
- MOV SI, OFFSET DATA1
- MOV DI, OFFSET DATA2
- MOV CX, 20
- REP MOVSB
- MAIN ENDP
- CODE_SEG ENDS
- END MAIN

INS Instruction (not on 8086/88)

- Transfers a byte, word or doubleword of data from an I/O device into the ES:DI

<i>Assembly Language</i>	<i>Operation</i>
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)

Note: [DX] indicates that DX contains the I/O device address. These instructions are not available on the 8086/8088 microprocessors.

EXAMPLE 4-7

```
                                ;Using the REP INSB to input data to a memory array
                                ;
0000  BF 0000 R                MOV    DI,OFFSET LISTS    ;address array
0003  BA 03AC                  MOV    DX,3ACH          ;address I/O
0006  FC                      CLD                    ;auto-increment
0007  B9 0032                  MOV    CX,50           ;load count
000A  F3/6C                    REP   INSB             ;input data
```

INS: *More*

- Transfers a byte, word, or doubleword of data from an I/O device into the extra segment memory location addressed by the DI register.
 - I/O address is contained in the DX register
- Useful for inputting a block of data from an external I/O device directly into the memory.
- One application transfers data from a disk drive to memory.
 - disk drives are often considered and interfaced as I/O devices in a computer system

- Three basic **forms** of the INS.
- **INSB** inputs data from an **8-bit** I/O device and stores it in a memory location indexed by DI.
- **INSW** instruction inputs 16-bit I/O data and stores it in a word-sized memory location.
- **INSD** instruction inputs a doubleword.
- These instructions **can be repeated** using the **REP** prefix
 - allows an entire block of input data to be stored in the memory from an I/O device

OUTS Instruction (not on 8086/88)

- Transfers a byte, word or doubleword of data from DS:SI to an I/O device

<i>Assembly Language</i>	<i>Operation</i>
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)

Note: [DX] indicates that DX contains the I/O device address. These instructions are not available on the 8086/8088 microprocessors.

EXAMPLE 4-8

```
                                ;Using the REP OUTS to output data from a memory array
                                ;
0000 BE 0064 R                   MOV    SI,OFFSET ARRAY    ;address array
0003 BA 03AC                     MOV    DX,3ACH          ;address I/O
0006 FC                           CLD                      ;auto-increment
0007 B9 0064                     MOV    CX,100          ;load count
000A F3/6E                       REP   OUTSB
```


OUTS : *More*

- Transfers a byte, word, or doubleword of data from the data segment memory location address by SI to an I/O device.
 - I/O device addressed by the DX register as with the INS instruction
- OUTSB
- OUTSW
- OUTSD
- INS and OUTS instructions not available on 8086/8088 microprocessors.

4-5 Miscellaneous Data Tx Instructions

■ XCHG (Exchange) Instruction

TABLE 4-16 Forms of the XCHG instruction.

<i>Assembly Language</i>	<i>Operation</i>
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

■ XLAT (Translate) Instruction

EXAMPLE 4-9

```
                                ;Using an XLAT to convert from BCD to 7-segment code
                                ;
                                ;.MODEL SMALL                                ;select SMALL model
                                ;.DATA                                    ;start of DATA segment
0000                                TABLE DB 3FH,6,5BH,4FH                ;7-segment lookup table
0000 3F 06 5B 4F                    DB 66H,6DH,7DH,27H
0004 66 6D 7D 27                    DB 7FH,6FH
0008 7F 6F
000A 00                                CODE7 DB ?                                ;reserve for result
0000                                .CODE                                ;start of CODE segment

                                ;.STARTUP                                ;start of program
0017 B0 04                                MOV AL,4                                ;load test data
0019 BB 0000 R                            MOV BX,OFFSET TABLE                    ;address lookup table
001C D7                                    XLAT                                    ;convert to 7-segment
001D A2 000A R                            MOV CODE7,AL                            ;save 7-segment code
                                ;.EXIT                                ;exit to DOS
                                END                                    ;end of file
```

XCHG : *More*

- Exchanges contents of a register with any other register or memory location.
 - cannot exchange segment registers or memory-to-memory data
- Exchanges are byte-, word-, or doubleword and use any addressing mode except immediate addressing.
- XCHG using the 16-bit AX register with another 16-bit register, is most efficient exchange.
- XCHG AL,[DI] identical to XCHG [DI], AL
- Example: XCHG AL, CL; XCHG CX, BP; XCHG AL,DATA2

XLAT : *More*

- Converts the contents of the AL register into a number stored in a memory table.
 - performs the **direct table lookup** technique often used to **convert one code to another**
- An XLAT instruction first **adds** the contents of **AL to BX** to form a memory **address** within the **data segment**.
 - copies the **contents** of this address **into AL**
 - The **only** instruction that **adds an 8-bit to a 16-bit number**

Example

- There is often a need in computer applications for a table that holds some important information. To access the elements of the table, 8088/86 –Core2 microprocessors provide the XLAT (translate) instruction.
- The table is commonly referred to as a look-up table.
- Assume that one needs a table for the values of x^2 , where x is between 0 and 9.
- First the table is generated and stored in memory:
- `SQUR_TABLE DB 0,1,4,9,16,25,36,49,64,81`
- Now one can access the square of any number from 0 to 9 by the use of XLAT. To do that, the register BX must have the offset address of the look-up table, and the number whose square is sought must be in AL register.
- Then after the execution of XLAT, the AL register will have the square of the number.
- The following shows how to get the square of 5 from the table:

```
MOV  BX, OFFSET SQUR_TABLE
MOV  AL,05
XLAT
```

- After execution of this program, the AL register will have 25 (19H), the square of 5.
- In fact, XLAT is equivalent to the following code:

```
SUB AH,AH ; AH=0
MOV SI,AX ; SI=00X
MOV AL,[BX+SI] ; Get the SIth entry from beginning of the table pointed at by BX.
```

IN and OUT

- **IN & OUT** instructions perform I/O operations.
- Contents of **AL, AX, or EAX** are transferred only **between I/O device and microprocessor**.
 - an **IN** instruction transfers data **from** an external **I/O device into AL, AX, or EAX**
 - an **OUT** transfers data **from AL, AX, or EAX to** an external **I/O device**
- Only the 80386 and above contain EAX

- Two forms of I/O device (port) addressing:
- *Fixed-port addressing* allows data transfer between AL, AX, or EAX using an 8-bit I/O port address.
 - port number follows the instruction's opcode
- *Variable-port addressing* allows data transfers between AL, AX, or EAX and a 16-bit port address.
 - the I/O port number is stored in register DX, which can be changed (varied) during the execution of a program.
- The port address appears on the address bus during an I/O operation. And extended by zeros in the case of 8-bit port address.

In and OUT (2)

Microprocessor-based system

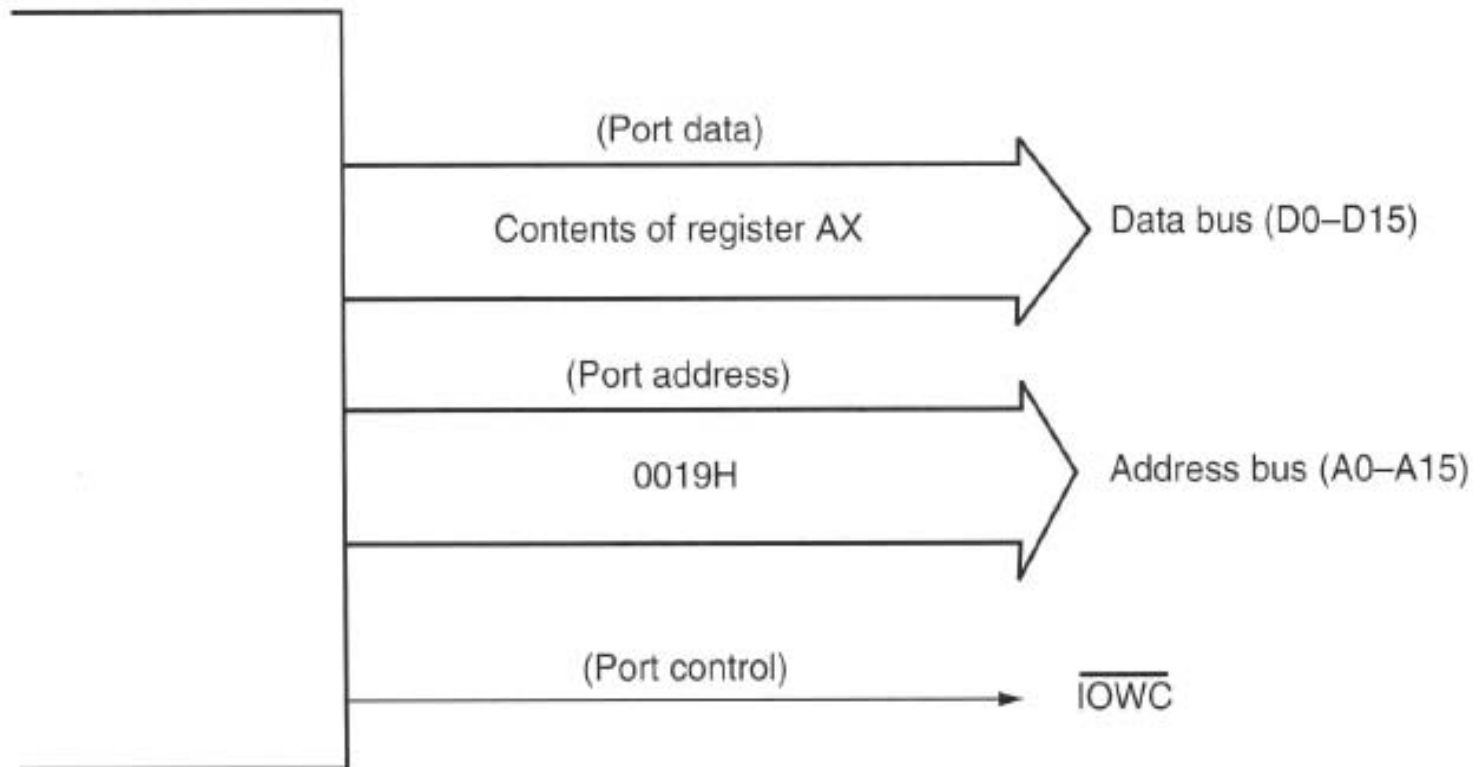


FIGURE 4–18 The signals found in the microprocessor-based system for an OUT 19H,AX instruction.

IN and OUT

- For data exchange between a microprocessor and I/O devices

<i>Assembly Language</i>	<i>Operation</i>
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 from AL to I/O port p8
OUT p8,AX	16-bits are output from AX to I/O port p8
OUT p8,EAX	32-bits are output from EAX to I/O port p8
OUT DX,AL	8-bits are output from AL to I/O port DX
OUT DX,AX	16-bits are output from AX to I/O port DX
OUT DX,EAX	32-bits are output from EAX to I/O port DX

Note: p8 = an 8-bit I/O port number and DX = the 16-bit port address held in DX.

Miscellaneous Data Tx Instructions

- **MOVSX** (move with sign extend) – in x386+
- **MOVZX** (move with zero extend) – in x386+
- **BSWAP** (byte swap; 1 \Leftrightarrow 4, 2 \Leftrightarrow 3) – in x486+
- **CMOV** (conditional move) – in Pentium+

MOVSX: Move and sign extend

MOVZX: Move and zero extend

Found only in 386 and above

Examples:

MOVSX CX,AL; **MOVSX** BX,DATA1;

MOVZX EBP,DI; **MOVZX** EAX, DATA3;

BSWAP :Takes the contents of any 32-bit register and swaps the first byte with the fourth, and the second with the third.

BSWAP (byte swap) is available only in 80486–Pentium 4 microprocessors

Example: **BSWAP** EAX

TABLE 4-19 The MOVSX and MOVZX instructions.

<i>Assembly Language</i>	<i>Operation</i>
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

CMOV (Conditional Move)

- Many variations of the CMOV instruction.
 - these move the data only if the condition is true
- New to the Pentium-above
- CMOVZ instruction moves data only if the result from some prior instruction was a zero.
 - 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
- Because this is a new instruction, you cannot use it with the assembler unless the .686 switch is added to the program

TABLE 4-20 The conditional move instructions.

<i>Assembly Language</i>	<i>Flag(s) Tested</i>	<i>Operation</i>
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 if 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 != 0	Move if less than
CMOVLE	Z = 1 or S != 0	Move if less than or equal
CMOVG	Z = 0 and S = 0	Move if greater than
CMOVGE	S = 0	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

CMOV_{cc}

- CMOVA *r16, r/m16* Move if above (CF=0 and ZF=0)
- CMOVA *r32, r/m32* Move if above (CF=0 and ZF=0)

- CMOVAE *r16, r/m16* Move if above or equal (CF=0)
- CMOVAE *r32, r/m32* Move if above or equal (CF=0)

- CMOVB *r16, r/m16* Move if below (CF=1)
- CMOVB *r32, r/m32* Move if below (CF=1)

- CMOVBE *r16, r/m16* Move if below or equal (CF=1 or ZF=1)
- CMOVBE *r32, r/m32* Move if below or equal (CF=1 or ZF=1)

- CMOVC *r16, r/m16* Move if carry (CF=1)
- CMOVC *r32, r/m32* Move if carry (CF=1)

- CMOVE *r16, r/m16* Move if equal (ZF=1)
- CMOVE *r32, r/m32* Move if equal (ZF=1)

CMOVcc: Example

- .Model Tiny
- .686
- .code
- .startup
-
- MOV BX,0AA00H
- MOV DX,0BB00H
- CMOVAE DX,BX ; Now: DX =0AA00H
- .exit
- END

4-6 Segment Override Prefix

<i>Assembly Language</i>	<i>Segment Accessed</i>	<i>Default Segment</i>
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 AX,ES:[SI]	Extra	Data
LODS ES:DATA1	Data	Extra
MOV EAX,FS:DATA2	Data	FS
MOV BL,GS:[ECX]	Data	GS

- ❑ The segment override prefix, which may be added to almost any instruction in any memory addressing mode, allows the programmer to deviate from the default segment.
- ❑ The segment override prefix is an additional byte that appends the front of an instruction to select an alternate segment register.
- ❑ The only instructions that cannot be prefixed are the jump and call instructions that must use the code segment register for address generation.

4-7 Assembler Directives

- **Processor Specific:** .286, .286P... .586, .586P
- **Co-processor Specific:** .287, .387
- **ASM Specific:** .model, .startup, .exit
- **Data definition:** DB, DW, DWord, DD, DQ, DT
- **Start definition:** macro, proc, segment, stack, struc
- **End definition:** endm, endp, ends, end
- **Classifiers:** byte, word, ptr, near, far, equ, offset
- **Coding Classifiers:** org, use16, use32

4–7 ASSEMBLER DETAIL

- *The assembler can be **used in two ways**:*
 - **Models: unique** to a particular assembler
 - **full-segment definitions** that allow complete control over the assembly process and are **universal** to all assemblers
- In most cases, the inline assembler found in Visual is used for developing assembly code for use in a program
 - occasions require separate assembly modules using the assembler

Directives

- **Pseudo-operations**
- Indicate **how** an operand or section of a program is to be **processed** by the **assembler**.
 - some generate and store information in the memory; others do not
- The DB directive stores bytes of data in the memory.
- BYTE PTR indicates the size of the data referenced by a pointer or index register.
- **Inline assembler** which is a part of VC++ does not use directives
- Complex sections of assembly code are still written using MASM.
- By **default** the assembler accepts **8086/8088** instructions, unless the program is processed by **MP selection switches**

Storing Data in a Memory Segment

- **DB**, **DW**, and **DD** are most often used with MASM to **define and store memory data**.
- If a **numeric coprocessor** executes software in the system, the **DQ (define quadword)** and **DT (define ten bytes)** directives are also common.
- 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**, **DW**, or **DD** directive.
 - when ? is used **in place of a numeric or ASCII value**, the assembler sets aside a location and **does not initialize** it to any specific value
 - **DUP: creates array** with or without initial values
- It is important that word-sized data are placed at word boundaries and doubleword-sized data are placed at doubleword boundaries.
 - if not, the microprocessor spends additional time accessing these data types

ASSUME, EQU, and ORG

- Equate directive (**EQU**) equates a numeric, ASCII, or **label** to another **label**.
 - equates **make a program clearer** and simplify debugging
 - EX: TEN EQU 10
 MOV AL,TEN

- The **ORG (origin)** statement **changes the starting offset address of the data or code segments.**
- At times, the origin of data or the code must be assigned to an absolute offset address with the ORG statement.
- **ASSUME** tells the assembler what **names** have been **chosen** for the **code, data, extra, and stack** segments.
 - Used only with full-segment definition

Example

- DATSEG segment
- DATA1 db 'ABCDEFGHJKLMNOPQRST'
- **ORG 30H**
- DATA2 db 20 DUP(?)
- DATSEG ENDS

- CODE_SEG SEGMENT
- ASSUME CS:CODE_SEG, DS:DATSEG
- MAIN PROC FAR
- MOV AX, DATSEG
- MOV ES, AX
- MOV DS, AX
- CLD
- MOV SI, OFFSET DATA1
- MOV DI, OFFSET DATA2
- MOV CX, 20
- REP MOVSB
- MAIN ENDP
- CODE_SEG ENDS
- END MAIN

PROC and ENDP

- Indicate **start and end of a procedure** (subroutine).
 - they *force structure* because the procedure is clearly defined
- Both the PROC and ENDP directives **require** a **label** to indicate the **name** of the **procedure**.
- **RET** instruction executed the end of the proc.
- **USES** directive indicates which registers are used by the proc.
 - The assembler automatically save and restore them using the stack instructions.
 - EX: PROC1 PROC USES AX BX CX
 - Use .LISTALL directive to view all instruction generated by assembler

- The PROC directive, which indicates the start of a procedure, must also be followed with a NEAR or FAR.
 - A NEAR procedure is one that resides in the same code segment as the program, often considered to be *local*
 - A FAR procedure may reside at any location in the memory system, considered *global*
- The term *global* denotes a procedure that can be used by any program.
- *Local* defines a procedure that is only used by the current program.

```

CODE_SEG  SEGMENT
          ASSUME CS:CODE_SEG, DS:DATSEG
MAIN      PROC          FAR
          MOV AX, DATSEG
          MOV ES, AX
          MOV DS, AX
          CALL SUBR1
          CALL SUBR2
          CALL SUBR3
MAIN      ENDP

```

PROC and ENDP: Summary

```

;-----
SUBR1     PROC
          ....
          ....
          RET
SUBR1     ENDP
;-----
SUBR2     PROC
          ....
          ....
          RET
SUBR2     ENDP
;-----
SUBR3     PROC
          ....
          ....
          RET
SUBR3     ENDP
CODE_SEG ENDS
END
MAIN

```

If version 6.x of the Microsoft MASM assembler Program is available, the PROC directive specifies And automatically saves any registers used within the Procedure. The USES statement indicates which Registers are used by the procedure, so that the Assembler can automatically save them before your Procedure begins and restore them before the Procedure ends with the RET instruction.

For example, the **PROC1 PROC USES AX BX CX** statement automatically pushes AX, BX, and CX on the stack before the procedure begins and pops them from the stack before the RET instruction executes at the end of the procedure.

Memory Organization

- The assembler uses two basic formats for developing software:
 - one method uses **models**; the other uses **full-segment definitions**
- **Memory models are unique to MASM.**
- The **models** are **easier** to use for **simple tasks**.
- The **full-segment** definitions offer better control over the assembly language task and are recommended for **complex programs**.

Models

- There are many models available to the MASM assembler, ranging from tiny to huge.
- **.MODEL *memsize***
 - *TINY: all software and data fit into 64kb memory segment. Useful for small programs. assembled as a command (.COM) program*
 - *SMALL: one data segment with one code segment for a total of 128kb of memory. assembled as an execute (.EXE) program*
- Start of segments: **.CODE, .DATA, .STACK**
- Start of instructions and load segment registers with segment addresses: **.STARTUP**
- Exit to DOS: **.EXIT**
- End of file: **END**
- MP selection : **.386, .486, .586, .686 ..**

Full-Segment Definitions

- Group names: 'STACK', 'CODE', and 'DATA' are used so that CodeView effectively used to debug the program.
- **Use assume directive before the program begins.**
- The program loader does not automatically initialize DS and ES. These registers must be loaded in the program.

- To access CodeView, type CV, followed by the file name at the DOS command line; if operating from Programmer's WorkBench, select Debug under the Run menu.
- If the group name is not placed in a program, CodeView can still be used to debug a program, but the program will not be debugged in symbolic form.

Examples

EXAMPLE 3-2

```
0000          .MODEL TINY          ;choose single segment model
             .CODE                ;start of code segment
             .STARTUP             ;start of program

0100  B8 0000  MOV AX,0            ;place 0000H into AX
0103  BB 0000  MOV BX,0            ;place 0000H into BX
0106  B9 0000  MOV CX,0            ;place 0000H into CX

0109  8B F0    MOV SI,AX           ;copy AX into SI
010B  8B F8    MOV DI,AX           ;copy AX into DI
010D  8B E8    MOV BP,AX           ;copy AX into BP

             .EXIT                ;exit to DOS
             END                  ;end of program
```



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

```
DATA_SEG SEGMENT 'DATA'  
    LISTA DB      100 DUP(?)  
    LISTB DB      100 DUP(?)  
DATA_SEG ENDS
```

```
CODE_SEG SEGMENT 'CODE'  
        ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG  
MAIN    PROC      FAR  
        MOV AX, DATA_SEG  
        MOV ES, AX  
        MOV DS, AX  
        CLD  
        MOV SI, OFFSET LISTA  
        MOV DI, OFFSET LISTB  
        MOV CX, 100  
        REP MOVSB  
MAIN    ENDP  
CODE_SEG ENDS  
        END      MAIN
```

<pre> ;FULL SEGMENT DEFINITION ;---- stack segment ---- name1 SEGMENT DB 64 DUP (?) name1 ENDS ;---- data segment ---- name2 SEGMENT DATA1 DW 2345H DATA2 DW 98F4H RESULT DW ? name2 ENDS ;---- code segment ---- name3 SEGMENT MAIN PROC FAR ASSUME ... MOV AX,name2 MOV DS,AX ... MAIN ENDP name3 ENDS END MAIN </pre>	<pre> ;SIMPLIFIED FORMAT .MODEL SMALL .STACK 64 ; ;----- .DATA DATA1 DW 2345H DATA2 DW 98F4H RESULT DW ? ; ;----- .CODE MAIN: MOV AX,@DATA MOV DS,AX END MAIN </pre>
--	--

Figure 2-3. Full vs. Simplified Segment Definition.

Intel and MASM documentation

Very Useful Link:

<http://web.sau.edu/LillisKevinM/csci240/masmdocs/>

SUMMARY

- Data movement instructions transfer data between registers, a register and memory, a register and the stack, memory and the stack, the accumulator and I/O, and the flags and the stack.
- Memory-to-memory transfers are allowed only with the MOVS instruction.

SUMMARY

(cont.)

- Data movement instructions include MOV, PUSH, POP, XCHG, XLAT, IN, OUT, LEA, LOS, LES, LSS, LGS, LFS, LAHF, SAHF, and the following string instructions: LODS, STOS, MOVS, INS, and OUTS.
- The first byte of an instruction contains the opcode, which specifies the operation performed by the microprocessor.
- The opcode may be preceded by one or more override prefixes.

SUMMARY

(cont.)

- The D-bit, located in many instructions, selects the direction of data flow.
- The W-bit, found in most instructions, selects the size of the data transfer.
- MOD selects the addressing mode of operation for a machine language instruction's R/M field.
- A 3-bit binary register code specifies the REG and R/M fields when the MOD = 11.

SUMMARY

(cont.)

- The 8-bit registers are AH, AL, BH, BL, CH, CL, DH, and DL.
- The 16-bit registers are AX, BX, CX, DX, SP, BP, DI, and SI.
- The 32-bit registers are EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI.
- To access the 64-bit registers, a new prefix is added called the REX prefix that contains a fourth bit.

SUMMARY

(cont.)

- By default, all memory-addressing modes address data in the data segment unless BP or EBP addresses memory.
- The BP or EBP register addresses data in the stack segment.
- The segment registers are addressed only by the MOV, PUSH, or POP instructions.
- The instruction may transfer a segment register to a 16-bit register, or vice versa.

SUMMARY

(cont.)

- The 80386 through the Pentium 4 include two additional segment registers, FS & GS.
- Data are transferred between a register or a memory location and the stack by the PUSH and POP instructions.
- Variations of these instructions allow immediate data to be pushed onto the stack, the flags to be transferred between the stack; all 16-bit registers can transfer between the stack and registers.

SUMMARY

(cont.)

- Opcodes that transfer data between the stack and the flags are PUSHF and POPF.
- Opcodes that transfer all the 16-bit registers between the stack and the registers are PUSHA and POPA.
- In 80386 and above, PUSHFD and POPFD transfer the contents of the EFLAGS between the microprocessor and the stack, and PUSHAD and POPAD transfer all the 32-bit registers.

SUMMARY

(cont.)

- The PUSHA and POPA instructions are invalid in the 64-bit mode.
- LEA, LDS, and LES instructions load a register or registers with an effective address.
- The LEA instruction loads any 16-bit register with an effective address; LDS and LES load any 16-bit register and either DS or ES with the effective address.

SUMMARY

(cont.)

- In 80386 and above, additional instructions include LFS, LGS, and LSS, which load a 16-bit register and FS, GS, or SS.
- String data transfer instructions use either or both DI and SI to address memory. .
- The DI offset address is located in the extra segment, and the SI offset address is located in the data segment.
- If 80386-Core2 operates in protected mode, ESI & EDI are used with string instructions.

SUMMARY

(cont.)

- The direction flag (D) chooses the auto-increment or auto-decrement mode of operation for DI and SI for string instructions.
- To clear D to 0, use the CLD instruction to select the auto-increment mode; to set D to 1, use the STD instruction to select the auto-decrement mode.
- Either/both DI and SI increment/decrement by 1 for a byte operation, by 2 for a word operation, and 4 for doubleword operation.

SUMMARY

(cont.)

- LODS loads AL, AX, or EAX with data from the memory location addressed by SI; STOS stores AL, AX, or EAX in the memory location addressed by DI; and MOVS transfers a byte, a word, or a doubleword from the memory location addressed by SI into the location addressed by DI.

SUMMARY

(cont.)

- INS inputs data from an I/O device addressed by DX and stores it in the memory location addressed by DI.
- The OUTS instruction outputs the contents of the memory location addressed by SI and sends it to the I/O device addressed by DX.

SUMMARY

(cont.)

- The REP prefix may be attached to any string instruction to repeat it.
- The REP prefix repeats the string instruction the number of times found in register CX.
- Arithmetic and logic operators can be used in assembly language.
- An example is `MOV AX,34*3`, which loads AX with 102.

SUMMARY

(cont.)

- Translate (XLAT) converts the data in AL into a number stored at the memory location addressed by BX plus AL.
- IN and OUT transfer data between AL, AX, or EAX and an external I/O device.
- The address of the I/O device is either stored with the instruction (fixed-port addressing) or in register DX (variable-port addressing).

SUMMARY

(cont.)

- The Pentium Pro-Core2 contain a new instruction called CMOV, or conditional move.
- This instruction only performs the move if the condition is true.
- The segment override prefix selects a different segment register for a memory location than the default segment.

SUMMARY

(cont.)

- Assembler directives DB (define byte), DW (define word), DD (define doubleword), and DUP (duplicate) store data in the memory system.
- The EQU (equate) directive allows data or labels to be equated to labels.
- The SEGMENT directive identifies the start of a memory segment and ENDS identifies the end of a segment when full-segment definitions are in use.

SUMMARY

(cont.)

- The ASSUME directive tells the assembler what segment names you have assigned to CS, DS, ES, and SS when full-segment definitions are in effect.
- In the 80386 and above, ASSUME also indicates the segment name for FS and GS.
- The PROC and ENDP directives indicate the start and end of a procedure.

SUMMARY

(cont.)

- The assembler assumes that software is being developed for the 8086/8088 microprocessor unless the .286, .386, .486, .586, or .686 directive is used to select one of these other microprocessors.
- This directive follows the .MODEL statement to use the 16-bit instruction mode and precedes it for the 32-bit instruction mode.

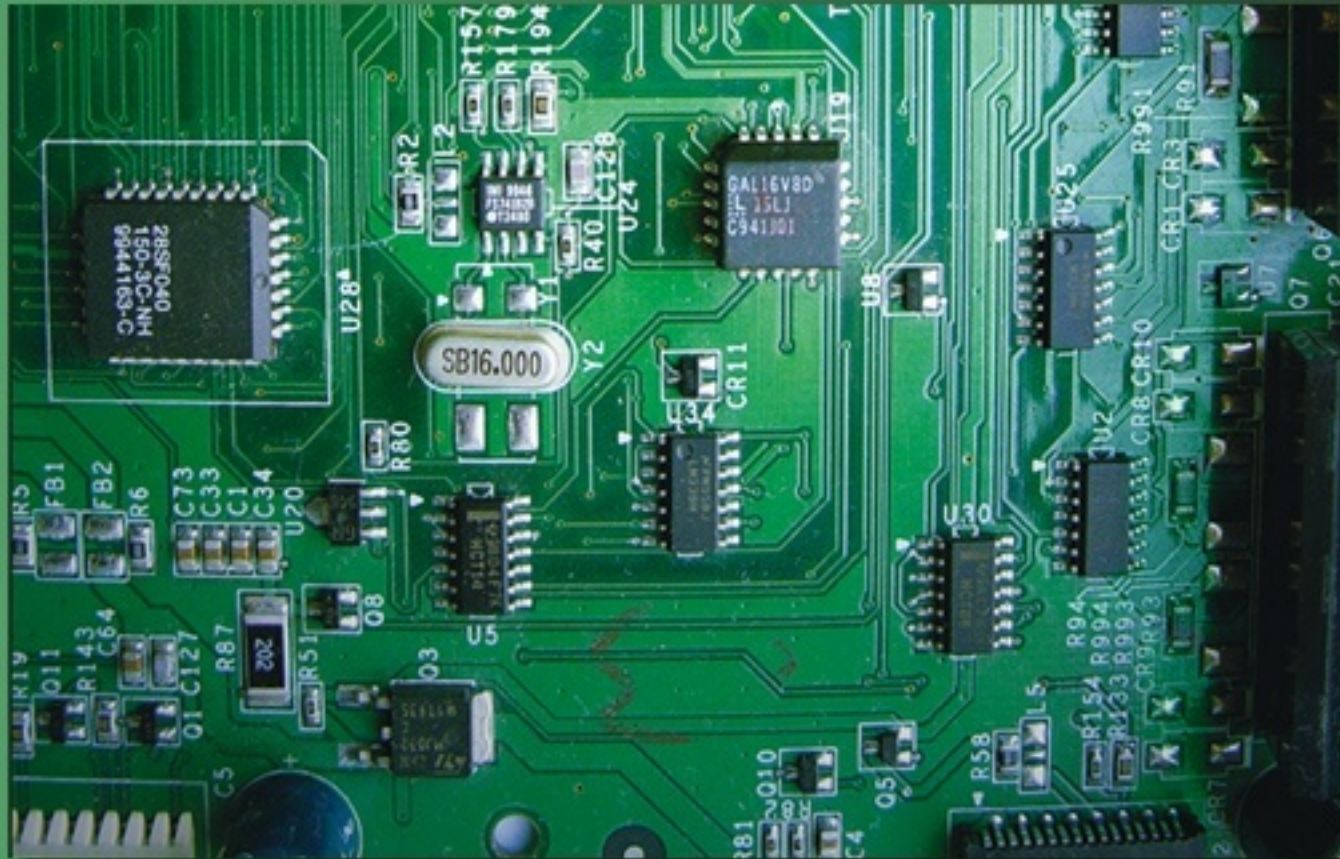
SUMMARY

- Memory models can be used to shorten the program slightly, but they can cause problems for larger programs.
- Also be aware that memory models are not compatible with all assembler programs.

The Intel Microprocessors

8086/8088, 80186/80188, 80286, 80386, 80486 Pentium, Pentium Pro Processor, Pentium II, Pentium 4, and Core2 with 64-bit Extensions

Architecture, Programming, and Interfacing



EIGHTH EDITION

Barry B. Brey

PEARSON

Chapter 5: Arithmetic and Logic Instructions

Chapter 5

Arithmetic and Logic Instructions

***Note: Most of slides are adapted from Barry B.
Brey (Author Slides)***

**Instructor:
Dr. Khalid A. Darabkeh**

Introduction

- We examine the arithmetic and logic instructions. The arithmetic instructions include addition, subtraction, multiplication, division, comparison, negation, increment, and decrement.
- The logic instructions include AND, OR, Exclusive-OR, NOT, shifts, rotates, and the logical compare (TEST).

Chapter Objectives

Upon completion of this chapter, you will be able to:

- Use arithmetic and logic instructions to accomplish simple binary, BCD, and AS-CII arithmetic.
- Use AND, OR, and Exclusive-OR to accomplish binary bit manipulation.
- Use the shift and rotate instructions.

Chapter Objectives

(*cont.*)

Upon completion of this chapter, you will be able to:

- Explain the operation of the 80386 through the Core2 exchange and add, compare and exchange, double-precision shift, bit test, and bit scan instructions.
- Check the contents of a table for a match with the string instructions.

5-1 ADDITION, SUBTRACTION AND COMPARISON

- The bulk of the arithmetic instructions found in any microprocessor include addition, subtraction, and comparison.
- Addition, subtraction, and comparison instructions are illustrated.
- Also shown are their uses in manipulating register and memory data.

Addition

- Addition (**ADD**) appears in many forms in the microprocessor.
- A second form of addition, called **add-with-carry**, is introduced with the **ADC** instruction.
- The only types of addition not allowed are **memory-to-memory** and **segment register**.
 - segment registers can only be moved, pushed, or popped
- Increment instruction (**INC**) is a special type of addition that **adds 1** to a number.
- See table 5.1

Register Addition

- Add the content of several registers.
- When arithmetic and logic instructions execute, contents of the **flag register change**.
 - interrupt, trap, and other flags do not change
- Any ADD instruction **modifies** the contents of the **sign, zero, carry, auxiliary carry, parity, and overflow** flags.
- EX: ADD AX, BX
ADD AX, CX
ADD AX, DX

Immediate Addition

- Immediate addition is employed whenever **constant or known data are added**.
- EX: MOV DL, 12H
ADD DL, 33H ;

The sum **45H** is stored in **DL**. **Flags** changes, as follows:

Z = 0 (result not zero)

S = 0 (result positive)

C = 0 (no carry)

P = 0 (odd parity)

A = 0 (no half carry)

O = 0 (no overflow)

Memory-to-Register Addition

- Moves **memory data to be added to a register.**
- EX: MOV DI, OFFSET NUMB
MOV AL, 0
ADD AL, [DI]
ADD AL, [DI+1]

Array Addition

- Memory arrays are sequential lists of data.
- EX: MOV AL, 0
MOV SI, 3
ADD AL, ARRAY[SI]
ADD AL, ARRAY[SI+2]
ADD AL, ARRAY[SI+4]

Array Addition

- Memory arrays are sequential lists of data.
- **Ex:** Suppose we want to add elements 3, 5, and 7 of an area of memory called **ARRAY**.

EXAMPLE 5-4

```
0000 B0 00          MOV    AL,0          ;clear sum
0002 BE 0003        MOV    SI,3          ;address element 3
0005 02 84 0000 R   ADD    AL,ARRAY[SI]  ;add element 3
0009 02 84 0002 R   ADD    AL,ARRAY[SI+2] ;add element 5
000D 02 84 0004 R   ADD    AL,ARRAY[SI+4] ;add element 7
```

- **EX:** Suppose that an array of data contains 16-bit number, to add elements 3, 5, and 7 of an area of memory called **ARRAY** a **scaled-index** form addressing is used.

EXAMPLE 5-5

```
0000 66|BB 00000000 R    MOV  EBX,OFFSET ARRAY    ;address ARRAY
0006 66|B9 00000003      MOV  ECX,3                ;address element 3
000C 67&8B 04 4B        MOV  AX,[EBX+2*ECX]      ;get element 3
0010 66|B9 00000005      MOV  ECX,5                ;address element 5
0016 67&03 04 4B        ADD  AX,[EBX+2*ECX]      ;add element 5
001A 66|B0 00000007      MOV  ECX,7                ;address element 7
0020 67&03 04 4B        ADD  AX,[EBX+2*ECX]      ;add element 7
```

- **EBX** is loaded with the address **ARRAY**, and **ECX** holds the array **element number**.
- The scaling factor is used to multiply the contents of the **ECX** register by **2** to address **words** of data.

TABLE 5–1 Example addition instructions.

<i>Assembly Language</i>	<i>Operation</i>
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

Assembly Language**Operation**

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+DI],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

Increment Addition

- The INC instruction adds 1 to any register or memory location, except a segment register.
- The size of the data must be described by using the BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR directives.
- The assembler program cannot determine if the INC [DI] instruction is a byte-, word-, or doubleword-sized increment.
- EX: INC BL, INC SP, INC EAX, INC BYTE PTR[BX], INC data1; see also ex 5.6.
- Affect the same flags except the C flag.
- Increment twice or add 2, the same but use INC if you want to preserve the C flag

TABLE 5–2 Example increment instructions.

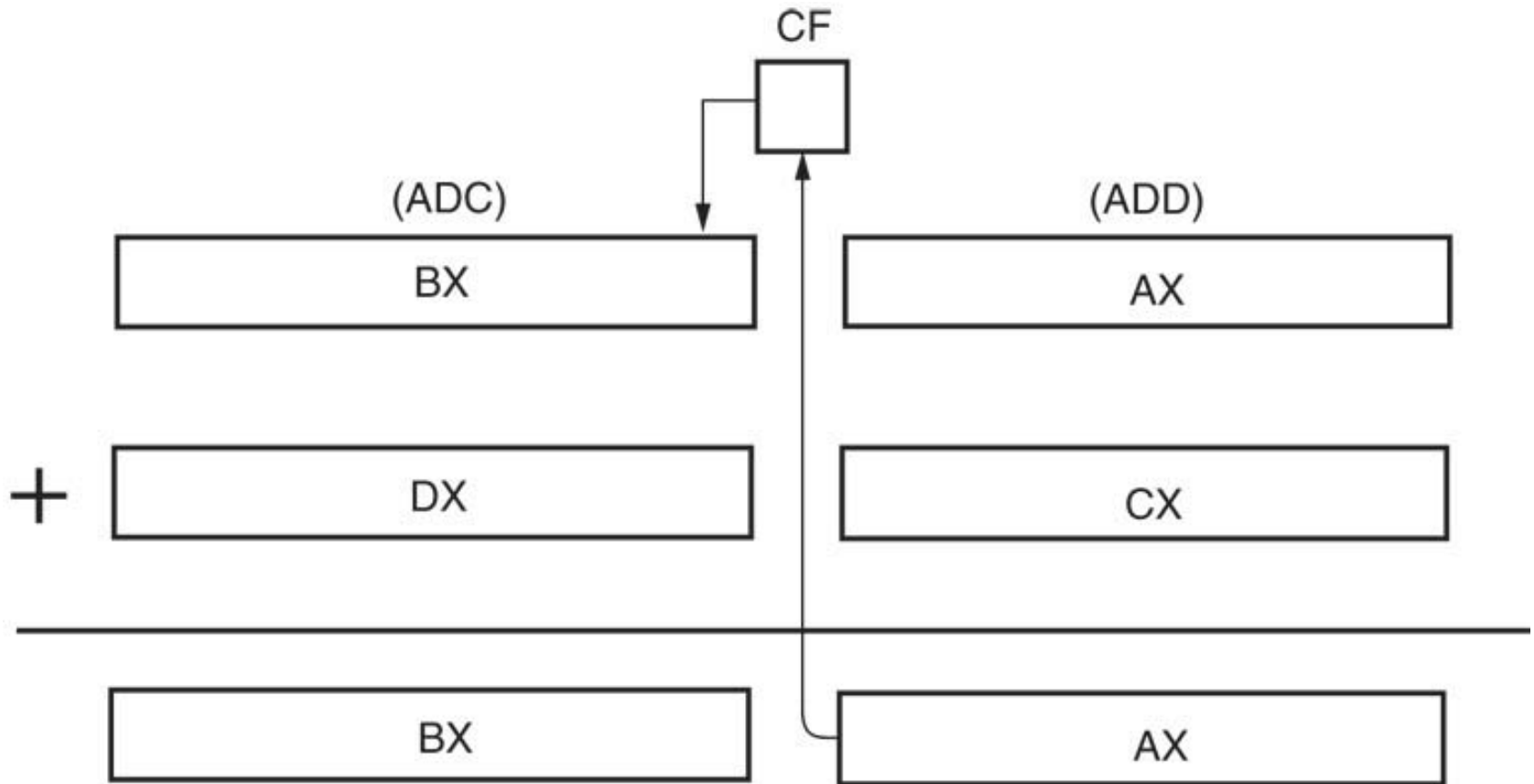
<i>Assembly Language</i>	<i>Operation</i>
INC BL	BL = BL + 1
INC SP	SP = SP + 1
INC EAX	EAX = EAX + 1
INC BYTE PTR[BX]	Adds 1 to the byte contents of the data segment memory location addressed by BX
INC WORD PTR[SI]	Adds 1 to the word contents of the data segment memory location addressed by SI
INC DWORD 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

Addition-with-Carry

- ADC adds the bit in the carry flag (C) to the operand data.
 - mainly appears in software that adds numbers wider than 16 or 32 bits in the 80386–Core2
 - like ADD, ADC affects the flags after the addition
- EX: ADC AL, AH; ADC CX, BX; ADC DH, [BX]
- Fig 5.1 illustrate an example:
 - cannot be easily performed without adding the carry flag bit because the 8086–80286 only adds 8- or 16-bit numbers

Figure 5–1 Addition-with-carry showing how the carry flag (C) links the two 16-bit additions into one 32-bit addition.

ADD AX,CX
ADC BX,DX



Exchange and Add for the 80486– Core2 Processors

- Exchange and add (**XADD**) appears in 80486 and continues through the Core2.
- XADD instruction adds the source to the destination and stores the sum in the destination, as with any addition.
 - **after the addition takes place**, the **original value** of the **destination** is **copied** into the **source** operand
- One of the **few instructions** that **change** the **source**.

Subtraction

- Many forms of subtraction (**SUB**) appear in the instruction set.
 - these use **any addressing mode** with 8-, 16-, or 32-bit data
 - a **special form** of subtraction (decrement, or **DEC**) subtracts 1 from any register or memory location
- Numbers that are wider than 16 bits or 32 bits must occasionally be subtracted.
 - the **subtract-with-borrow instruction (SBB)** performs this type of subtraction

Register Subtraction

- After each subtraction, the microprocessor modifies the contents of the flag register.
 - flags change for most arithmetic/logic operations
- Ex: SUB BX, CX

Immediate Subtraction

- The microprocessor also allows immediate operands for the subtraction of constant data.
- Ex: MOV CH, 22H
SUB CH,44H; flags: z=0, c=1; A=1; s=1; P=1; O=0

TABLE 5-4 Example subtraction instructions.

<i>Assembly Language</i>	<i>Operation</i>
SUB CL,BL	CL = CL – BL
SUB AX,SP	AX = AX – SP
SUB ECX,EBP	ECX = ECX – EBP
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

Decrement Subtraction

Subtracts **1** from a **register/memory** location.

TABLE 5–5 Example decrement instructions.

<i>Assembly Language</i>	<i>Operation</i>
DEC BH	BH = BH – 1
DEC CX	CX = CX – 1
DEC EDX	EDX = EDX – 1
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 NUMB	Subtracts 1 from the contents of data segment memory location NUMB

Subtraction-with-Borrow

- A subtraction-with-borrow (SBB) instruction functions as a regular subtraction, except that the carry flag (C), which holds the borrow, also subtracts from the difference.
 - most common use is subtractions wider than 16 bits in the 8086–80286 microprocessors or wider than 32 bits in the 80386–Core2.
 - wide subtractions require borrows to propagate through the subtraction, just as wide additions propagate the carry

Ex: SBB AH, AL; SBB AX, BX; SBB CL, 2;
SBB BYTE PTR[DI], 3

Figure 5–2 Subtraction-with-borrow showing how the carry flag (C) propagates the borrow.

SUB AX,DI
SBB BX,SI

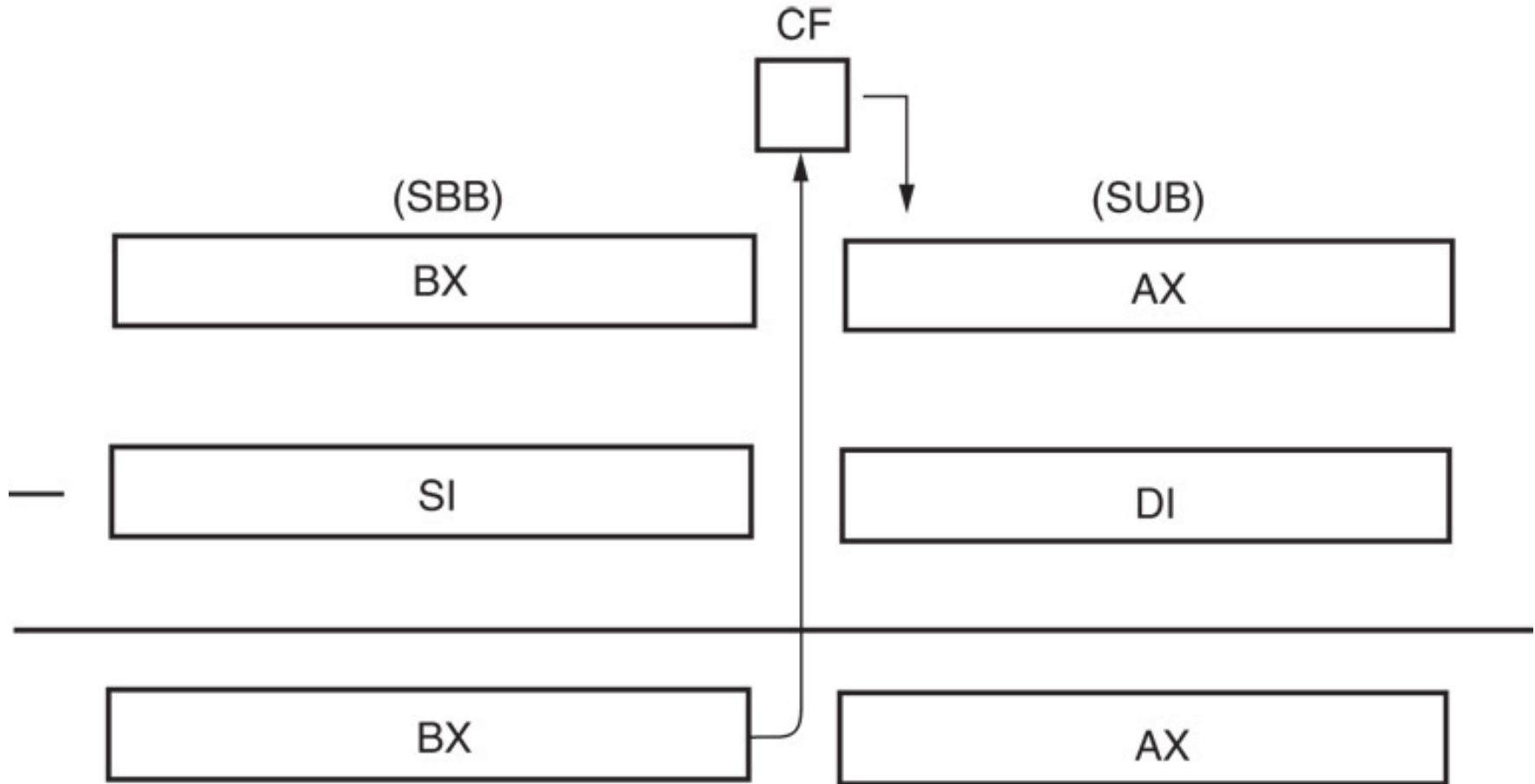


TABLE 5-6 Example subtraction-with-borrow instructions.

<i>Assembly Language</i>	<i>Operation</i>
SBB AH,AL	$AH = AH - AL - \text{carry}$
SBB AX,BX	$AX = AX - BX - \text{carry}$
SBB EAX,ECX	$EAX = EAX - ECX - \text{carry}$
SBB CL,2	$CL = CL - 2 - \text{carry}$
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

Comparison

- The comparison instruction (**CMP**) is a **subtraction that changes only the flag bits**.
 - **destination** operand never changes
- Useful for **checking the contents** of a **register or a memory location against another value**.
- A **CMP** is normally **followed** by a conditional **jump** instruction, which **tests** the condition of the **flag** bits.
- See table 5-7 for examples.
- EX: **CMP AL, 10H**
JAE NEXT ; jump if above or equal

TABLE 5-7 Example comparison instructions.

<i>Assembly Language</i>	<i>Operation</i>
CMP CL,BL	CL – BL
CMP AX,SP	AX – SP
CMP EBP,ESI	EBP – ESI
CMP AX,2000H	AX – 2000H
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

Compare and Exchange (80486–Core2 Processors Only)

- Compare and exchange instruction (CMPXCHG) compares the destination operand with the accumulator.
 - found only in 80486 - Core2 instruction sets
- If they are equal, the source operand is copied to the destination; if not equal, the destination operand is copied into the accumulator.
 - instruction functions with 8-, 16-, or 32-bit data

- **CMPXCHG CX,DX** instruction is an example of the compare and exchange instruction.
 - this compares the contents of CX with AX
 - if CX equals AX, DX is copied into AX; if CX is not equal to AX, CX is copied into AX
 - also compares AL with 8-bit data and EAX with 32-bit data if the operands are either 8- or 32-bit
- This instruction has a bug that will cause the operating system to crash.
 - more information about this flaw can be obtained at www.intel.com

5-2 MULTIPLICATION AND DIVISION

- Earlier 8-bit microprocessors could not multiply or divide without the use of a program that multiplied or divided by using a series of shifts and additions or subtractions.
 - manufacturers were aware of this inadequacy, they incorporated multiplication and division into the instruction sets of newer microprocessors.
- Pentium–Core2 contains special circuitry to do multiplication in as few as one clocking period.
 - over 40 clocking periods in earlier processors

Multiplication

- Performed on bytes, words, or doublewords,
 - can be signed (IMUL) or unsigned integer (MUL)
- Affected flags are C, and O.
 - Set: if higher byte of result not zero
 - Reset: the result fit exactly the lower half.
- Product after a multiplication always a double-width product.
 - two 8-bit numbers multiplied generate a 16-bit product; two 16-bit numbers generate a 32-bit; two 32-bit numbers generate a 64-bit product
 - in 64-bit mode of Pentium 4, two 64-bit numbers are multiplied to generate a 128-bit product

8-Bit Multiplication

- With 8-bit multiplication, the **multiplicand** is always in the **AL** register, **signed or unsigned**.
 - **multiplier** can be **any 8-bit register or memory location**
- **Immediate** multiplication is **not allowed unless** the special signed immediate multiplication instruction appears in a program.
- The multiplication instruction contains **one operand** because it always multiplies the operand times the contents of register **AL**.

TABLE 5–8 Example 8-bit multiplication instructions.

<i>Assembly Language</i>	<i>Operation</i>
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

EXAMPLE 5–13

```
0000 B3 05      MOV    BL, 5           ;load data
0002 B1 0A      MOV    CL, 10
0004 8A C1      MOV    AL, CL         ;position data
0006 F6 E3      MUL    BL             ;multiply
0008 8B D0      MOV    DX, AX        ;position product
```

AX = BL X CL after multiplication

16-Bit Multiplication

- Word multiplication is very similar to byte multiplication.
- **AX** contains the multiplicand instead of AL.
 - 32-bit product appears in **DX–AX** instead of AX
- The **DX** register always contains the **most significant** 16 bits of the product; **AX** contains the **least significant** 16 bits.
- As with 8-bit multiplication, the choice of the **multiplier is up to the programmer.**

32-Bit Multiplication

- In **80386 and above**, 32-bit multiplication is allowed because these microprocessors contain 32-bit registers.
 - can be signed or unsigned by using **IMUL** and **MUL** instructions
- Contents of **EAX** are multiplied by the operand specified with the instruction.
- The 64 bit product is found in **EDX–EAX**, where **EAX** contains the least significant 32 bits of the product.

TABLE 5–9 Example 16-bit multiplication instructions.

<i>Assembly Language</i>	<i>Operation</i>
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

TABLE 5–10 Example 32-bit multiplication instructions.

<i>Assembly Language</i>	<i>Operation</i>
MUL ECX	EAX is multiplied by ECX; the unsigned product is in EDX–EAX
IMUL EDI	EAX is multiplied by EDI; the signed product is in EDX–EAX
MUL DWORD PTR[ESI]	EAX is multiplied by the doubleword contents of the data segment memory location address by ESI; the unsigned product is in EDX–EAX

A Special Immediate 16-Bit Multiplication

- **80186 - Core2 processors** can use a special version of the multiply instruction.
 - **immediate** multiplication must be **signed**;
 - instruction format is different because it contains **three operands**
- First operand is 16-bit destination register; the second a register/memory location with 16-bit multiplicand; the third 8- or 16-bit immediate data used as the multiplier.
 - Ex: **IMUL BX, NUMBER, 1000H**

IMUL CX,DX,12H

Multiplies 12H times DX and leaves a 16-bit signed product in CX

CX = 12 x DX

Another example

IMUL BX,NUMBER,1000H

Multiplies NUMBER times 1000H and leaves the product in BX

BX = 1000H x NUMBER

IMUL—Signed Multiply

Opcode	Instruction	Description
F6 /5	IMUL <i>r/m8</i>	AX ← AL * <i>r/m</i> byte
F7 /5	IMUL <i>r/m16</i>	DX:AX ← AX * <i>r/m</i> word
F7 /5	IMUL <i>r/m32</i>	EDX:EAX ← EAX * <i>r/m</i> doubleword
0F AF /r	IMUL <i>r16,r/m16</i>	word register ← word register * <i>r/m</i> word
0F AF /r	IMUL <i>r32,r/m32</i>	doubleword register ← doubleword register * <i>r/m</i> doubleword
6B /r ib	IMUL <i>r16,r/m16,imm8</i>	word register ← <i>r/m16</i> * sign-extended immediate byte
6B /r ib	IMUL <i>r32,r/m32,imm8</i>	doubleword register ← <i>r/m32</i> * sign-extended immediate byte
6B /r ib	IMUL <i>r16,imm8</i>	word register ← word register * sign-extended immediate byte
6B /r ib	IMUL <i>r32,imm8</i>	doubleword register ← doubleword register * sign-extended immediate byte
69 /r iw	IMUL <i>r16,r/m16,imm16</i>	word register ← <i>r/m16</i> * immediate word
69 /r id	IMUL <i>r32,r/m32,imm32</i>	doubleword register ← <i>r/m32</i> * immediate doubleword
69 /r iw	IMUL <i>r16,imm16</i>	word register ← <i>r/m16</i> * immediate word
69 /r id	IMUL <i>r32,imm32</i>	doubleword register ← <i>r/m32</i> * immediate doubleword

Division

- Occurs on **8- or 16-bit and 32-bit numbers** depending on microprocessor.
 - **signed (IDIV)** or **unsigned (DIV)** integers
- Dividend is always a double-width dividend, divided by the operand.
- There is **no immediate** division instruction available to any microprocessor.

- A division can result in two types of **errors**:
 - attempt to **divide by zero**
 - other is a **divide overflow**, which occurs when a **small number divides into a large number**. (ex: $AX=1300 / 2$ the result 1500 in AL cause and overflow)
- In either case, the microprocessor generates an **interrupt** if a divide error occurs.
- In most systems, a divide error interrupt **displays an error message on the video screen**.
- **No affected flags are defined**

8-Bit Division

- Uses AX to store the dividend divided by the contents of any 8-bit register or memory location.
- **Quotient** moves into **AL** after the division with **AH containing** a whole number **remainder**.
 - quotient is positive or negative; **remainder** always **assumes sign** of the **dividend**; always an **integer**
- Numbers usually 8 bits wide in 8-bit division .
 - the dividend must be **converted** to a 16-bit wide number in **AX** ; accomplished differently for signed and unsigned numbers.
- For the unsigned number, the most significant 8 bits must be cleared to zero (zero-extended). The MOVZX instruction can be used to zero-extend a number in the 80386 through the Core2 processors.
- For singed numbers, the least significant 8 bits are sign-extended into the most 8 bits. In the micro-processor, a special instruction sign-extends AL to AH, or convert an 8-bit singed number in AL into a 16-bit singed number in AX. **CBW (convert byte to word)** instruction performs this conversion.
- In 80386 through Core2, **MOVSX** sign-extends a number.

TABLE 5–11 Example 8-bit division instructions.

<i>Assembly Language</i>	<i>Operation</i>
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

```
0000 A0 0000 R      MOV    AL, NUMB      ;get NUMB
0003 B4 00          MOV    AH, 0         ;zero-extend
0005 F6 36 0002 R   DIV    NUMB1        ;divide by NUMB1
0009 A2 0003 R      MOV    ANSQ, AL       ;save quotient
000C 88 26 0004 R   MOV    ANSR, AH      ;save remainder
```

16-Bit Division

- Sixteen-bit division is similar to 8-bit division
 - instead of dividing into **AX**, the 16-bit number is divided into **DX–AX**, a 32-bit dividend
- As with 8-bit division, numbers must often be converted to the proper form for the dividend.
- If a 16-bit unsigned number is placed in **AX**, **DX** must be **cleared to zero**
- In the 80386 and above, the number is **zero-extended** by using the **MOVZX** instruction.
- If **AX** is a 16-bit signed number, the **CWD** (convert word to doubleword) instruction sign-extends it into a signed 32-bit number.
- If the 80386 and above are available, the **MOVSX** instruction can also be used to sign-extend a number.

TABLE 5–12 Example 16-bit division instructions.

<i>Assembly Language</i>	<i>Operation</i>
DIV CX	DX–AX is divided by CX; the unsigned quotient is in 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

EXAMPLE 5–15

```
0000 B8 FF9C      MOV  AX,-100      ;load a -100
0003 B9 0009      MOV  CX,9         ;load +9
0006 99           CWD              ;sign-extend
0007 F7 F9       IDIV CX
```

The **CWD** (convert word to doubleword) converts the **-100** in AX to **-100** in **DX-AX** before the division.

After the division, the result appear in DX-AX as a quotient of **-11** in **AX** and a remainder of **-1** in **DX**.

32-Bit Division

- 80386 - Pentium 4 perform 32-bit division on signed or unsigned numbers.
 - 64-bit contents of EDX–EAX are divided by the operand specified by the instruction
 - leaving a 32-bit quotient in EAX
 - and a 32-bit remainder in EDX
- Other than the size of the registers, this instruction functions in the same manner as the 8- and 16-bit divisions.
- The **CDQ (convert doubleword to quadword) instruction** is used before a signed division to convert the 32-bit contents of EAX into a 64-bit signed number in EDX-EAX.

TABLE 5–13 Example 32-bit division instructions.

<i>Assembly Language</i>	<i>Operation</i>
DIV ECX	EDX–EAX is divided by ECX; the unsigned quotient is in EAX and the unsigned remainder is in EDX
IDIV DATA4	EDX–EAX is divided by the doubleword contents in data segment memory location DATA4; the signed quotient is in EAX and the signed remainder is in EDX
DIV DWORD PTR[EDI]	EDX–EAX is divided by the doubleword contents of the data segment memory location addressed by EDI; the unsigned quotient is in EAX and the unsigned remainder is in EDX

The Remainder

- Could be used to **round** the quotient or dropped to truncate the quotient.
- If division is unsigned, **rounding requires the remainder be compared with half the divisor** to decide whether to round up the quotient
- The remainder could also be **converted to a fractional remainder**.

```
DIV BL
ADD AH,AH
CMP AH, BL
JB NEXT
INC AL
NEXT:
```

```
MOV AX, 13
MOV BL 2
DIV BL
MOV ANSQ, AL
MOV AL, 0
DIV BL
MOV ANSR, AL
```

Suppose that a fractional remainder is required instead of an integer remainder. A fractional remainder is obtained by saving the quotient. Next, the AL register is cleared to zero. The number remaining in AX is now divided by the original operand to generate a fractional remainder.

Example

- DATSEG segment
- DATA1 db +13,-10,+19,+14,-18,-9,+12,-9,+16
- ORG 0010H
- AVERAGE dw ?
- REMAINDER dw ?
- DATSEG ENDS
- CODE_SEG SEGMENT
- ASSUME CS:CODE_SEG, DS:DATSEG
- MAIN PROC FAR
- MOV AX, DATSEG
- MOV ES, AX
- MOV DS, AX
- MOV CX,9 ; Load counter
- SUB BX,BX ; Clear BX, Used as accumulator
- MOV SI,OFFSET DATA1
- BACK: MOV AL,[SI]
- CBW ; sign extend into AX
- ADD BX,AX
- INC SI
- LOOP BACK
- MOV AX,BX
- CWD
- IDIV CX
- MOV AVERAGE,AX
- MOV REMAINDER,DX
- MAIN ENDP
- CODE_SEG ENDS
- END MAIN

5-3 BCD and ASCII Arithmetic

- The microprocessor allows arithmetic manipulation of both **BCD** (binary-coded decimal) and **ASCII** (American Standard Code for Information Interchange) **data**.
- **BCD** operations occur in systems such as point-of-sales terminals (e.g., cash registers) and others that seldom require complex arithmetic.

BCD Arithmetic

- Two arithmetic techniques operate with BCD data: addition and subtraction.
- **DAA (decimal adjust after addition)** instruction follows BCD addition,
- **DAS (decimal adjust after subtraction)** follows BCD subtraction.
 - both correct the result of addition or subtraction so it is a BCD number

DAA Instruction

- DAA follows the ADD or ADC instruction to adjust the result into a BCD result.
- After adding the **BL** and **DL** registers, the result is adjusted with a DAA instruction before being stored in **CL**.

DAS Instruction

- Functions as does DAA instruction, except it follows a subtraction instead of an addition.

EXAMPLE 5-18

```
0000 BA 1234          MOV    DX,1234H      ;load 1234 BCD
0003 BB 3099          MOV    BX,3099H      ;load 3099 BCD
0006 8A C3            MOV    AL,BL         ;sum BL and DL
0008 02 C2            ADD    AL,DL
000A 27               DAA
000B 8A C8            MOV    CL,AL         ;answer to CL
000D 9A C7            MOV    AL,BH         ;sum BH, DH an carry
000F 12 C6            ADC    AL,DH
0011 27               DAA
0012 8A E8            MOV    CH,AL         ;answer to CH
```

Processing Packed BCD Numbers

- Two instructions to process packed BCD numbers
 1. DAA – Decimal Adjust after addition
Used after ADD or ADC instruction
 2. DAS – Decimal adjust after subtraction
Used after SUB or SBB instruction
- No support for multiplication or division

Examples

Ex:

27H = 00100111

34H = 00110100

5BH = 01011011

Should be 61H (add 6)

Ex:

52H = 01010010

61H = 01100001

B3H = 10110011 should be 113 (add 60H)

Ex:

29H = 00101001

69H = 01101001

92H = 10010010

Should be 98H (add 6)

Summary: The DAA instruction works as follows:

```
MOV AL,71H
```

```
ADD AL,43H ; AL = B4H
```

```
DAA ; AL = 14 H and CF = 1
```

- If the least significant four bits in AL are >9 or if $AF=1$, it adds 6 to AL and sets AF
- If the most significant four bits in AL are >9 or if $CF=1$, it adds 60 to AL and sets the CF

Summary: The DAS instruction works as follows:

```
MOV AL,71H
```

```
SUB AL,43H ; AL = 2EH
```

```
DAS ; AL = 28 H
```

- If the least significant four bits in AL are >9 or if AF=1, it subtracts 6 from AL and sets AF
- If the most significant four bits in AL are >9 or if CF=1, it subtracts 60 from AL and sets the CF

ASCII Arithmetic

- ASCII arithmetic instructions function with coded numbers, value 30H to 39H for 0–9.
- Four instructions in 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 register **AX** as the **source** and as the **destination**.

BCD Number System

- **BCD number system:**
- In computer literature one encounters two terms for BCD numbers:
- **Unpacked BCD**
- In unpacked BCD, the lower 4 bits of the number represent the BCD number and the rest of the bits are 0.
- Example: 0000 1001 and 0000 0101 are unpacked BCD for 9 and 5, respectively.
- **Packed BCD**
- In the case of packed BCD, a single byte has two BCD numbers in it, one in the lower 4 bits and one in the upper 4 bits.
- Example: 0101 1001 is packed BCD for 59. It takes only a byte of memory to store the packed BCD operands.

ASCII numbers

ASCII to BCD conversion:

ASCII to unpacked BCD conversion:

To convert ASCII data to BCD, the programmer must get rid of the tagged “011” in the highest 4 bits of the ASCII. To do that, each ASCII number is ANDed with “0000 1111” (0FH).

ASCII to packed BCD conversion:

To convert ASCII to packed BCD, it is first converted to unpacked BCD (to get rid of 3) and then combined to make packed BCD. For example, for 9 and 5 the keyboard gives 39 and 35 respectively. The goal is to produce 95H or “1001 0101”, which is called packed BCD.

Key	ASCII (hex)	Binary	BCD unpacked
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

Key	ASCII	Unpacked BCD	Packed BCD
4	34	0000 0100	0100 0111
7	37	0000 0111	Or 47H

AAA Instruction

- ASCII adjust AL after addition.
- Adjusts the sum of two unpacked BCD values to create an unpacked BCD result.
- The AL register is the implied source and destination operand for this instruction.
- only useful when it follows an ADD instruction

- Ex1: **MOV AL, '5' ; AL=35**
- **ADD AL, '2' ; add to AL 32 the ASCII of 2**
- **AAA ; changes 67H to 07H**
- **OR AL, 30 ; OR AL with 30 to get ASCII**

- Ex2: 31 + 39 = 6A this ASCII addition should produce two two-digit ASCII result 10 which is 31 30 H.

```
MOV AX, 31H                            ; AX = 0031H
ADD AL, 39H                            ; AX = 006AH
AAA                                     ; AX = 0100H
ADD AX, 3030H                         ; AX = 3130 which is the ASCII for 10H
```

```
Ex3: SUB AH,AH ;AH=00
MOV AL,'7' ;AL=37H
MOV BL, '5' ;BL=35H
ADD AL,BL ;37H+35H=6CH therefore AL=6C
AAA ; changes 6CH to 02 in AL and AH=CF=1
OR AX, 3030H ;AX=3132 which is the ASCII for 12H
```

AAS Instruction

- Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result.
- The AL register is the implied source and destination operand for this instruction.
- only useful when it follows an SUB instruction.

EX: 38 – 39 = FF, the result should be FF 39H

```
MOV AH, 0           ; AH=0
MOV AL, '8'         ; AX=0038H
SUB AL, '9'         ; AX=00FFH
AAS                 ; AX=FF09H
OR AL, 30h          ; AX=FF39H
```

AAA

```
IF ((AL AND 0FH) > 9) OR (AF = 1)
  THEN
    AL ← (AL + 6);
    AH ← AH + 1;
    AF ← 1;
    CF ← 1;
  ELSE
    AF ← 0;
    CF ← 0;
FI;
AL ← AL AND 0FH;
```

AAS

```
IF ((AL AND 0FH) > 9) OR (AF = 1)
  THEN
    AL ← AL - 6;
    AH ← AH - 1;
    AF ← 1;
    CF ← 1;
  ELSE
    CF ← 0;
    AF ← 0;
FI;
AL ← AL AND 0FH;
```

Example:

- Sub AH,AH ; clear AH
- Mov AL,'6' ; AL =36H
- Add AL, '7' ; AL =36H+37H = 6DH
- AAA ; AX = 0103H
- Or AX,3030H ; AL=3133H

- **Example 1: Positive result**

```
Sub AH,AH    ; clear AH
Mov  AL,'9'  ; AL =39H
Sub  AL, '3'  ; AL =39H-33H = 06H
AAS          ; AX = 0006H
Or   AL,30H  ; AL=36
```

- **Example 2 : Negative result**

```
Sub AH,AH    ; clear AH
Mov  AL,'3'  ; AL =33H
Sub  AL, '9'  ; AL =33H-39H = FAH
AAS          ; AX = FF04H
Or   AL,30H  ; AL=34
```


AAM Instruction

- Adjusts the result of the multiplication of two unpacked BCD values to create a pair of unpacked BCD values.
- The AX register is the implied source and destination.
- The AAM instruction is only useful when it follows an MUL instruction.

- Ex:
• MOV AL,'7' ;AL=37H
• AND AL,0F ;AL=07 unpacked BCD
• MOV DL,'6' ;DL=36H
• AND DL,0FH ;DL=06 unpacked BCD
• MUL DL ;AX=ALXDL=07*06=002AH=42
• AAM ;AX=0402 (7x6=42 unpacked BCD)
• OR AX,3030H ;AX=3432H result in ASCII

- Ex:
MOV BL, 5
MOV AL, 6
MUL BL ; AX= 001EH
AAM ; AX= 0300H

Exceptional case of division regarding where to save remainder and quotient.

The AAM instruction is used to adjust the content of the AL and AH registers after the AL register has been used to perform the multiplication of two unpacked BCD bytes. The CPU uses the following simple logic:
 $al = al \bmod 10$
 $ah = al/10$

The AAM instruction accomplish this conversion by dividing AX by 10. the remainder is found in AL, and the quotient is in AH. Note that the second byte of the instruction contains 0A (page 174 of your book). If the 0AH is changed to 0BH, the AAM instruction divides by 11. Thus, be careful.

AAD Instruction

- Appears before a division.
- The **AAD** instruction requires the **AX** register contain a two-digit unpacked **BCD** number (**not ASCII**) before executing.
- Before dividing the unpacked BCD by another unpacked BCD, AAD is used to convert it to HEX. By doing that the quotient and remainder are both in unpacked BCD.

- **Ex:**
- **MOV AX,3539H ;AX=3539 ASCII for 59**
- **AND AX,0F0FH ; AH=05, AL=09 unpacked BCD data**
- **AAD ; AX=003BH hex equivalent of 59**
- **MOV BH,08H ; divide by 08**
- **DIV BH ; 3B/08 gives AL=07, AH=03**
- **OR AX,3030H ; AL=37H (quotient) AH=33H (rem)**

Ex:

MOV AX, 0307 ; AX=0307H

AAD ;AX=0025H

MOV BL, 5

DIV BL ;AX=0207

Convert from Two-Byte Unpacked BCD to Binary: To convert from a two-byte unpacked BCD to a binary number, multiply the most significant byte of the BCD by decimal ten (0AH), then add the product to the least significant byte.

For example,

$00001001\ 00000010_{(\text{unpacked BCD})} = 01011100_{(\text{base 2})}$

5-4 BASIC LOGIC INSTRUCTIONS

- Include **AND**, **OR**, **Exclusive-OR**, and **NOT**.
 - also **TEST**, a special form of the AND instruction
 - **NEG**, similar to the NOT instruction
- Logic operations provide **binary bit control** in low-level software.
 - **allow bits to be set, cleared, or complemented**
- Low-level software appears in machine language or assembly language form and often controls the I/O devices in a system.

- All logic instructions affect the flag bits.
- Logic operations always clear the carry and overflow flags
 - other flags change to reflect the result
- When binary data are manipulated in a register or a memory location, **the rightmost bit position is always numbered bit 0.**
 - **position numbers increase from bit 0 to the left**, to bit 7 for a byte, and to bit 15 for a word
 - a doubleword (32 bits) uses bit position 31 as its leftmost bit and a quadword (64-bits) position 63

AND

- Performs **logical multiplication**, illustrated by a truth table.
- AND can **replace discrete AND gates** if the speed required is not too great
 - normally reserved for embedded control applications
- In 8086, the AND instruction often executes in about a microsecond.
 - with newer versions, the execution speed is greatly increased

Figure 5–3 (a) The truth table for the AND operation and (b) the logic symbol of an AND gate.

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

(a)



(b)

- AND clears bits of a binary number.
 - called **masking**
- AND uses any mode except memory-to-memory and segment register addressing.
- An **ASCII** number can be converted to **BCD** by using AND to **mask off** the leftmost four binary bit positions.
 - Ex: `MOV BX, 3135H`
`AND BX, 0F0FH`

Figure 5–4 The operation of the AND function showing how bits of a number are cleared to zero.

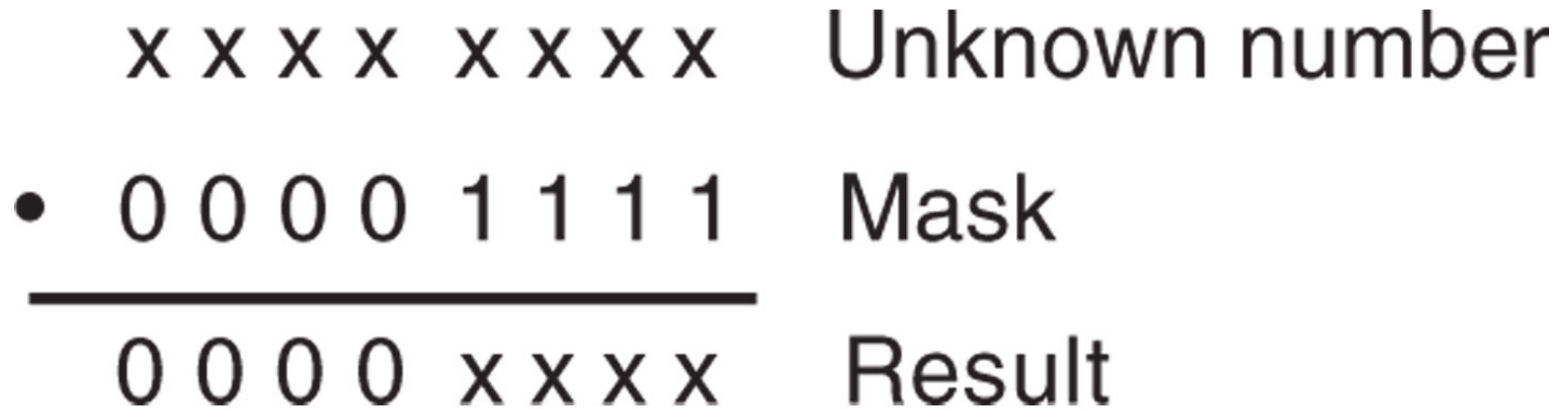


TABLE 5-16 Example AND instructions.

<i>Assembly Language</i>	<i>Operation</i>
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

OR

- Performs **logical addition**
 - often called the *Inclusive-OR* function
- The OR function generates a logic 1 output if any inputs are 1.
 - a 0 appears at output only when all inputs are 0
- Figure 5–6 shows how the OR gate **sets (1)** **any bit** of a binary number.
- The OR instruction uses **any addressing mode except segment register addressing.**

Figure 5–5 (a) The truth table for the OR operation and (b) the logic symbol of an OR gate.

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

(a)



(b)

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

	X X X X	X X X X	Unknown number
+	0 0 0 0	1 1 1 1	Mask
<hr/>			
	X X X X	1 1 1 1	Result

Example 5.26

```
MOV    AL, 5
MOV    BL, 7
MUL    BL
AAM
OR     AX, 3030H
```

TABLE 5–17 Example OR instructions.

<i>Assembly Language</i>	<i>Operation</i>
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

Exclusive-OR

- Differs from Inclusive-OR (OR) in that the 1,1 condition of Exclusive-OR produces a 0.
 - a 1,1 condition of the OR function produces a 1
- The **Exclusive-OR operation *excludes* this condition**; the Inclusive-OR *includes* it.
- If inputs of the Exclusive-OR function are both 0 or both 1, the output is 0; if the **inputs are different, the output is 1.**
- Exclusive-OR is **sometimes called a comparator.**

Figure 5–7 (a) The truth table for the Exclusive-OR operation and (b) the logic symbol of an Exclusive-OR gate.

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

(a)



(b)

- XOR uses any **addressing mode except segment register addressing**.
- Exclusive-OR is useful if **some bits** of a register or memory location **must be inverted**.
- Figure 5–8 shows how just part of an unknown quantity can be inverted by XOR.
 - when a 1 Exclusive-ORs with X, the result is X
 - if a 0 Exclusive-ORs with X, the result is X
- A common use for the Exclusive-OR instruction is to **clear a register to zero** (XOR CH,CH)
 - Takes 2 bytes
 - Mov takes 3 bytes

Figure 5–8 The operation of the Exclusive-OR function showing how bits of a number are inverted.

x x x x	x x x x	Unknown number
⊕ 0 0 0 0	1 1 1 1	Mask
<hr/>		
x x x x	\bar{x} \bar{x} \bar{x} \bar{x}	Result

Example 5.27

OR	CX,0600H	;Set bits 9 and 10
AND	CX,0FFFCH	;Clear bits 0 and 1
XOR	CX,1000H	;Invert bit 12

TABLE 5–18 Example Exclusive-OR instructions.

<i>Assembly Language</i>	<i>Operation</i>
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 and Bit Test Instructions

- **TEST** performs the **AND** operation.
 - only affects the condition of the flag register, which indicates the result of the test
 - functions the same manner as a CMP
 - Normally tests a **single bit or multiple bits**
- Usually followed by either the **JZ** (jump if zero) or **JNZ** (jump if not zero) instruction.
 - Z=0 if the bit under test is not zero
 - Z=1 if the bit under test is a zero
- The **destination** operand is **normally tested against immediate** data (indicating the bit weight).

- 80386 - Pentium 4 contain additional test instructions that test single bit positions.
 - four different bit test instructions available
- All forms test the bit position in the destination operand selected by the source operand.
(study table 5-18)

- Ex 5-28

TEST AL,1

JNZ RIGHT

TEST AL,128

JNZ LEFT

TABLE 5–19 Example
TEST instructions.

<i>Assembly Language</i>	<i>Operation</i>
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

NOT and NEG

- The **NOT** instruction **inverts all bits of a byte, word, or doubleword. One's complement**.
 - None of flags affected.
- **NEG two's complements a number**.
 - the **arithmetic sign of a signed number changes** from positive to negative or negative to positive
 - The **CF flag cleared to 0 if the source operand is 0**; otherwise it is set to 1. other flags are set according to the result.
- The **NOT** function is considered **logical**, **NEG** function is considered an **arithmetic** operation.
- **NOT and NEG can use any addressing mode except segment register addressing.**

TABLE 5–21 Example NOT and NEG instructions.

<i>Assembly Language</i>	<i>Operation</i>
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

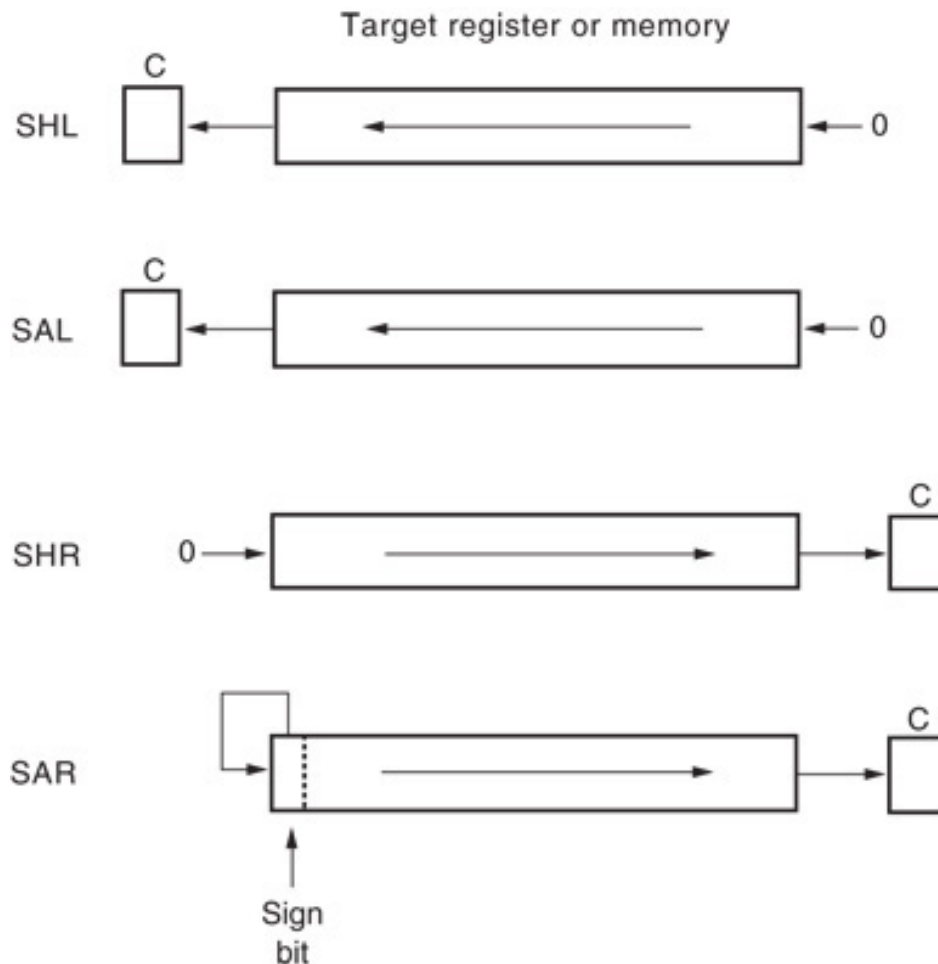
Shift and Rotate

- Shift and rotate instructions manipulate binary numbers at the binary bit level.
 - as did AND, OR, Exclusive-OR, and NOT
- Common applications in low-level software used to control I/O devices.
- The microprocessor contains a complete complement of shift and rotate instructions that are used to shift or rotate any memory data or register.

Shift

- Position or **move numbers to the left or right within a register or memory location.**
 - also perform **simple arithmetic as multiplication** by powers of 2^{+n} (left shift) and **division** by powers of 2^{-n} (right shift).
- The microprocessor's instruction set contains **four different shift instructions:**
 - **two are logical; two are arithmetic** shifts
- All four shift operations appear in Figure 5–9.

Figure 5–9 The shift instructions showing the operation and direction of the shift.



- logical shifts **move 0 in the rightmost bit** for a **logical left shift**;
- The **arithmetic shift left** is identical to the logical shift left.
- 0 to the leftmost bit position for a **logical right shift**
- **arithmetic right shift** copies the sign-bit through the number.

- **Logical** shift function with **unsigned** numbers
- **Arithmetic** shift 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
 - a shift right always divides by 2 for each position
 - shifting a two places, multiplies or divides by 4
- **Segment shift not allowed**
- Two forms:
 - **Immediate shift count** (SHR BX, 12)
 - **CL holds the shift count** (SAL DATA, CL)

Examples

- Multiply the contents of AX by 10 (1010) :
SHL AX,1 ; AX times 2
MOV BX,AX
SHL AX,2 ; AX times 8
ADD AX,BX ; AX times 10
- Multiply the contents of AX by 18 (10010) :
SHL AX,1 ; AX times 2
MOV BX,AX
SHL AX,3 ; AX times 16
ADD AX,BX ; AX times 20
- Multiply the contents of AX by 5 (101) :
MOV BX,AX
SHL AX,2 ; AX times 4
ADD AX,BX ; AX times 5
- **Multiply by constant** using shift left is faster than using MUL operation
- The CF flag contains the value of the last bit shifted out of the destination operand. The SF, ZF, and PF flags are set according to the result.

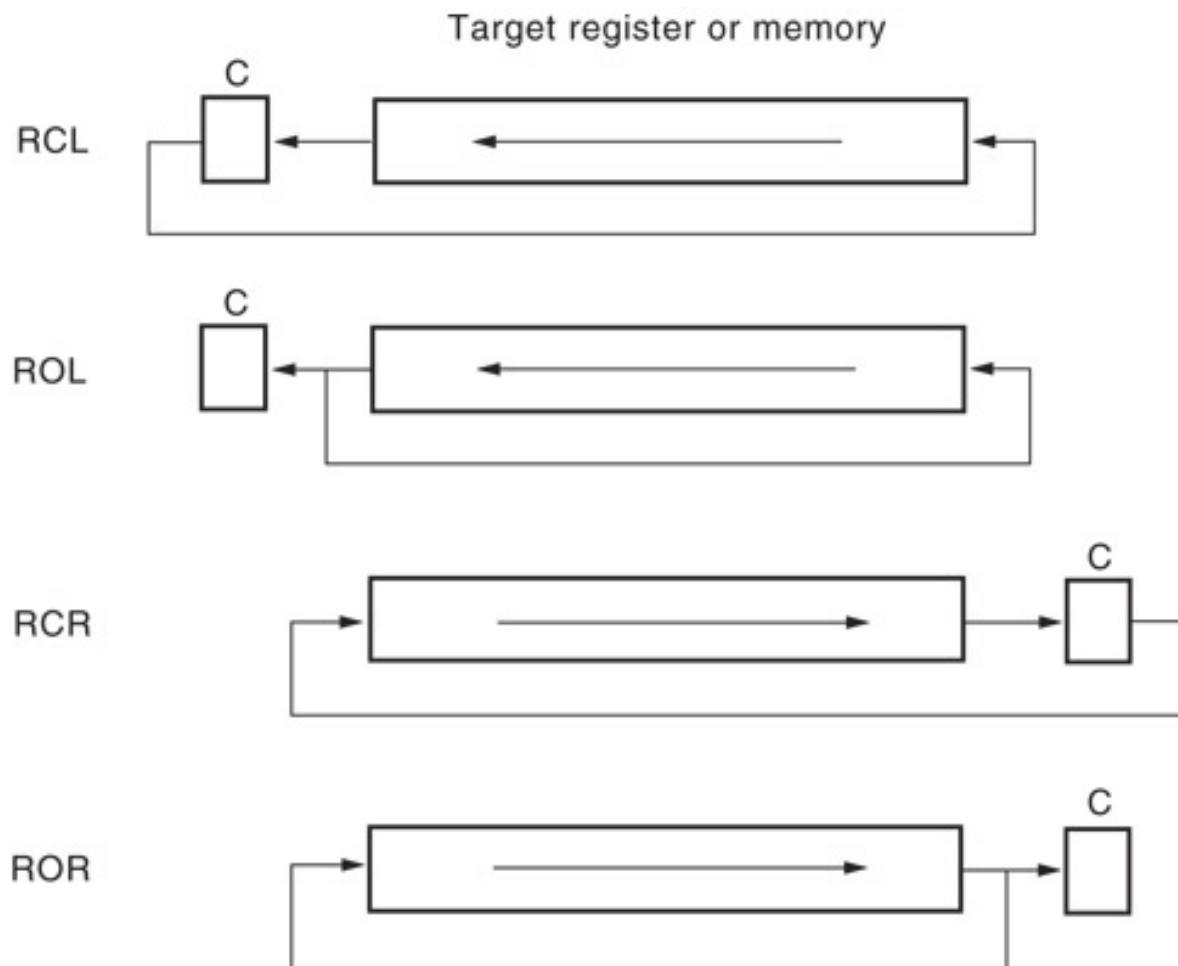
TABLE 5–22 Example shift instructions.

<i>Assembly Language</i>	<i>Operation</i>
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

Rotate

- Positions binary data by **rotating information** in a **register or memory location**, either from **one end to another** or **through the carry** flag.
 - used to shift/position numbers wider than 16 bits
- With either type of instruction, the programmer can select either a left or a right rotate.
- **Addressing modes** used with rotate are the **same as** those used with **shifts**.
- Rotate instructions appear in Figure 5–10.

Figure 5–10 The rotate instructions showing the direction and operation of each rotate.



- A **rotate count** can be **immediate** or located in register **CL**.
 - if CL is used for a rotate count, it does not change
- Rotate instructions are often used to shift wide numbers to the left or right.
 - Ex: shift the 48 binary number left one position

SHL AX, 1

RCL BX, 1

RCL DX, 1

TABLE 5–23 Example rotate instructions.

<i>Assembly Language</i>	<i>Operation</i>
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

- Scan through a number **searching for the first 1-bit**.
 - accomplished by shifting the number
 - available in 80386–Pentium 4
- **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**, the **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** the **zero flag is cleared**
 - Ex:
 - If `EAX = 60000000H` and `BSF EBX,EAX` instruction executes, the number is scanned from the leftmost bit toward the right. The first 1-bit encountered is at position 30, which is placed into `EBX` and the zero flag is set. If the same value of `EAX` is used for the `BSR` instruction, the `EBX` register is loaded with 29 and the zero flag bit is set.

5-6 STRING COMPARISONS

- String instructions are powerful because they allow the programmer to manipulate large blocks of data with relative ease.
- Block data manipulation occurs with MOVS, LODS, STOS, INS, and OUTS.
- Additional string instructions allow a section of memory to be tested against a constant or against another section of memory.
 - **SCAS** (string scan); **CMPS** (string compare)

SCAS

- Compares the AL register with a byte block of memory, AX with a word block, or EAX with a doubleword block of memory.
- Opcode used for byte comparison is **SCASB**; for word comparison **SCASW**; doubleword comparison is **SCASD**
- In all cases the content of **ES** memory location addressed by **DI** compared with the accumulator (AL, AX, or EAX).
- SCAS uses **direction flag (D)** to select auto-increment or auto-decrement operation for DI.
 - also **repeat if prefixed by conditional repeat prefix**
 - **REPNE** : cause the SCAS instruction to repeat until either CX reaches 0, or until an equal condition exists as the outcome of the comparison. Another conditional repeat prefix is REPE (repeat while equal)

Example

- In the data segment:
 - `DATA1 DB 'Mr. Gones'`
- And in the Code Segment:
 - `CLD ; DF=0 for increment`
 - `MOV DI, Offset DATA1 ; DI=Array offset`
 - `MOV cx,09 ; Length of array`
 - `MOV AL,'G' ; Scanning for letter 'G'`
 - `REPNE SCANSB ; Repeat the scanning if not equal or`
until CX becomes zero.
- In the example above, the letter 'G' is compared with 'M'. Since they are not equal, DI is incremented and CX is decremented, and the scanning is repeated until the letter 'G' is found or the CX register is zero. In that example, since 'G' is found, ZF is set to 1 (ZF=1), indicating that there is a letter 'G' in the array.

Example (Cont.)

- In the data segment:

- `DATA1 DB 'Mr. Gones'`

- And in the Code Segment:

- `CLD ; DF=0 for increment`

- `MOV DI, Offset DATA1 ; DI=Array offset`

- `MOV cx,09 ; Length of array`

- `MOV AL,'G' ; Scanning for letter 'G'`

- `REPNE SCANSB ; Repeat the scanning if not equal or`
until CX becomes zero.

- `JNE OVER; JUMP if ZF=0`

- `DEC DI`

- `MOV BYTE PTR [DI], 'J'`

- `OVER:`

- The above program scans the name Mr. Gones and replaces the 'G' with the letter 'J'.

CMPS

- Always compares two sections of memory data as bytes (CMPSB), words (CMPSW), or doublewords (CMPSD).
 - contents of the data segment memory location addressed by SI are compared with contents of extra segment memory addressed by DI
 - CMPS instruction increments/decrements SI & DI
- Normally used with REPE or REPNE prefix.
 - alternates are REPZ (repeat while zero) and REPNZ (repeat while not zero)
 - REPE: cause the CMPS instruction to repeat until either CX reaches 0, or until an unequal condition exists as the outcome of the comparison

Example

```
DATSEG segment
DATA1 DB 'Europe'
DATA2 DB 'Euorope'
MESSAGE1 DB 'The spelling is correct' ; try this: 'The spelling is correct','$'
MESSAGE2 DB 'Wrong Spelling' ; try this: 'Wrong Spelling','$'
DATSEG ENDS

CODE_SEG          SEGMENT
ASSUME CS:CODE_SEG, DS:DATSEG

MAIN      PROC      FAR
            MOV AX, DATSEG
            MOV ES, AX
            MOV DS, AX

            MOV CX,6 ; Load counter
            CLD ; DF=0 for increment
            MOV SI,OFFSET DATA1
            MOV DI,OFFSET DATA2
            REPE CMPSB ; Repeat as long as equal or until CX=0
            JE OVER ; If ZF=1 then display message 1
            MOV DX, Offset MESSAGE2 ; If ZF=0 then display message 2
            JMP DISPLAY

OVER:      MOV DX, Offset MESSAGE1
DISPLAY:   MOV AH,09
            INT 21H

MAIN      ENDP
CODE_SEG          ENDS
END      MAIN
```


DOS Interrupt 21H

- **INT 21H option 09: outputting a string of data to the monitor:** INT 21H can be used to send a set of ASCII data to the monitor. To do that, the following registers must be set: AH=09 and DX = the offset address of the ASCII data to be displayed. Then INT 21H is invoked. The address of the DX register is an offset address and DS is assumed to be the data segment.
- INT 21H option 09 will display the ASCII data string pointed by DX until it encounters the dollar sign '\$'. In the absence of encountering a dollar sign, DOS function call 09 will continue to display any garbage that it can find in subsequent memory locations until it finds '\$'.
- **INT 21H option 02: outputting a single character to the monitor:** to do that, 02 is put in AH, and DL is loaded with the character to be displayed and then INT 21H is invoked.

MOV AH,02

MOV DL,'J'

INT 21H

- This option can also be used to display '\$' on the monitor since the string display option (option 09) will not display '\$'.

DOS Interrupt 21H

- **INT 21H option 0AH: Inputting a string of data from the keyboard:**

- DATSEG segment
- **ORG 0010H**
- **DATA1 DB 6,?,6 DUP (0FFH)**
- DATSEG ENDS
- CODE_SEG SEGMENT
- ASSUME CS:CODE_SEG, DS:DATSEG
- MAIN PROC FAR
- MOV AX, DATSEG
- MOV ES, AX
- MOV DS, AX
- **MOV AH,0AH**
- **MOV DX,offset DATA1**
- **INT 21H**
- MAIN ENDP
- CODE_SEG ENDS
- END MAIN

Assuming the data that was entered through the Keyboard was "USA" <RETURN>, the content of Memory locations starting at 0010H would be like this:

0010	0011	0012	0013	0014	0015	0016	0017
06	03	55	53	41	0D	FF	FF
		U	S	A	CR		

Step-by-step analysis:

0010H=06 DOS requires the size of the buffer in the first location.

0011H =03 number of letters read excluding CR key

0012H: this the ASCII hex for letter U

0013H: this the ASCII hex for letter S

0014H: this the ASCII hex for letter A

0015H: this the ASCII hex for letter CR (carriage return)

DOS Interrupt 21H

- **INT 21H option 01H: Inputting a single character:**
- **This functions waits until a character is input from the keyboard. After the interrupt, the input character will be in AL.**

- **MOV AH, 01 ; Option 01 inputs one character**
- **INT 21H ; After this interrupt, AL=input character**

SUMMARY

- Addition (ADD) can be 8, 16, 32, or 64 bits.
- The ADD instruction allows any addressing mode except segment register addressing.
- Most flags (C, A, S, Z, P, and O) change when the ADD instruction executes.
- A different type of addition, add-with-carry (ADC), adds two operands and the contents of the carry flag (C).

SUMMARY

(*cont.*)

- The 80486 through the Core2 processors have an additional instruction (XADD) that combines an addition with an exchange.
- The increment instruction (INC) adds 1 to the byte, word, or doubleword contents of a register or memory location.
- The INC instruction affects the same flag bits as ADD except the carry flag.

SUMMARY

(*cont.*)

- BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR directives appear with the INC instruction when contents of a memory location are addressed by a pointer.
- Subtraction (SUB) is a byte, word, doubleword, or quadword and is performed on a register or a memory location.
- The only form of addressing not allowed by the SUB instruction is segment register addressing.

SUMMARY

(*cont.*)

- The subtract instruction affects the same flags as ADD and subtracts carry if the SBB form is used.
- The decrement (DEC) instruction subtracts 1 from the contents of a register or a memory location.
- The only addressing modes not allowed with DEC are immediate or segment register addressing.

SUMMARY

(*cont.*)

- The DEC instruction does not affect the carry flag and is often used with BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR.
- The comparison (CMP) instruction is a special form of subtraction that does not store the difference; instead, the flags change to reflect the difference.

SUMMARY

(*cont.*)

- Comparison is used to compare an entire byte or word located in any register (except segment) or memory location.
- An additional comparison instruction (CMPXCHG), which is a combination of comparison and exchange instructions, is found in the 80486-Core2 processors.
- In the Pentium-Core2 processors, the CMPXCHG8B instruction compares and exchanges quadword data.

SUMMARY

(*cont.*)

- Multiplication is byte, word, or doubleword; can be signed (IMUL) or un-signed (MUL).
- The 8-bit multiplication always multiplies register AL by an operand with the product found in AX.
- The 16-bit multiplication always multiplies register AX by an operand with the product found in DX-AX.

SUMMARY

(*cont.*)

- The 32-bit multiply always multiplies register EAX by an operand with the product found in EDX-EAX.
- A special IMUL immediate instruction exists on the 80186-Core2 processors that contains three operands.
- In the Pentium 4 and Core2 with 64-bit mode enabled, multiplication is 64 bits.

SUMMARY

(*cont.*)

- Division is byte, word, or doubleword, and it can be signed (IDIV) or unsigned (DIV).
- For an 8-bit division, the AX register divides by the operand, after which the quotient appears in AL and the remainder appears in AH.
- In the 16-bit division, the DX-AX register divides by the operand, after which the AX register contains the quotient and DX contains the remainder.

SUMMARY

(*cont.*)

- In the 32-bit division, the EDX-EAX register is divided by the operand, after which the EAX register contains the quotient and the EDX register contains the remainder.
- The remainder after a signed division always assumes the sign of the dividend.
- BCD data add or subtract in packed form by adjusting the result of the addition with DAA or the subtraction with DAS.

SUMMARY

(*cont.*)

- ASCII data are added, subtracted, multiplied, or divided when the operations are adjusted with AAA, AAS, AAM, and AAD.
- These instructions do not function in the 64-bit mode.
- The AAM instruction has an interesting added feature that allows it to convert a binary number into unpacked BCD.

SUMMARY

(*cont.*)

- This instruction converts a binary number between 00H-63H into unpacked BCD in AX.
- The AAM instruction divides AX by 10, and leaves the remainder in AL and quotient in AH.
- These instructions do not function in the 64-bit mode.

SUMMARY

(cont.)

- The AND, OR, and Exclusive-OR instructions perform logic functions on a byte, word, or doubleword stored in a register or memory location.
- All flags change with these instructions, with carry (C) and overflow (O) cleared.
- The TEST instruction performs the AND operation, but the logical product is lost.
- This instruction changes the flag bits to indicate the outcome of the test.

SUMMARY

(*cont.*)

- The NOT and NEG instructions perform logical inversion and arithmetic inversion.
- The NOT instruction one's complements an operand, and the NEG instruction two's complements an operand.

SUMMARY

(*cont.*)

- There are eight different shift and rotate instructions.
- Each of these instructions shifts or rotates a byte, word, or doubleword register or memory data.
- These instructions have two operands: The first is the location of the data shifted or rotated, and the second is an immediate shift or rotate count or CL.

SUMMARY

(*cont.*)

- If the second operand is CL, the CL register holds the shift or rotate count.
- In the 80386 through the Core2 processors, two additional double-precision shifts (SHRD and SHLD) exist.
- The scan string (SCAS) instruction compares AL, AX, or EAX with the contents of the extra segment memory location addressed by DI.

SUMMARY

(*cont.*)

- The string compare (CMPS) instruction compares the byte, word, or doubleword contents of two sections of memory.
- One section is addressed by DI in the extra segment, and the other is addressed by SI in the data segment.

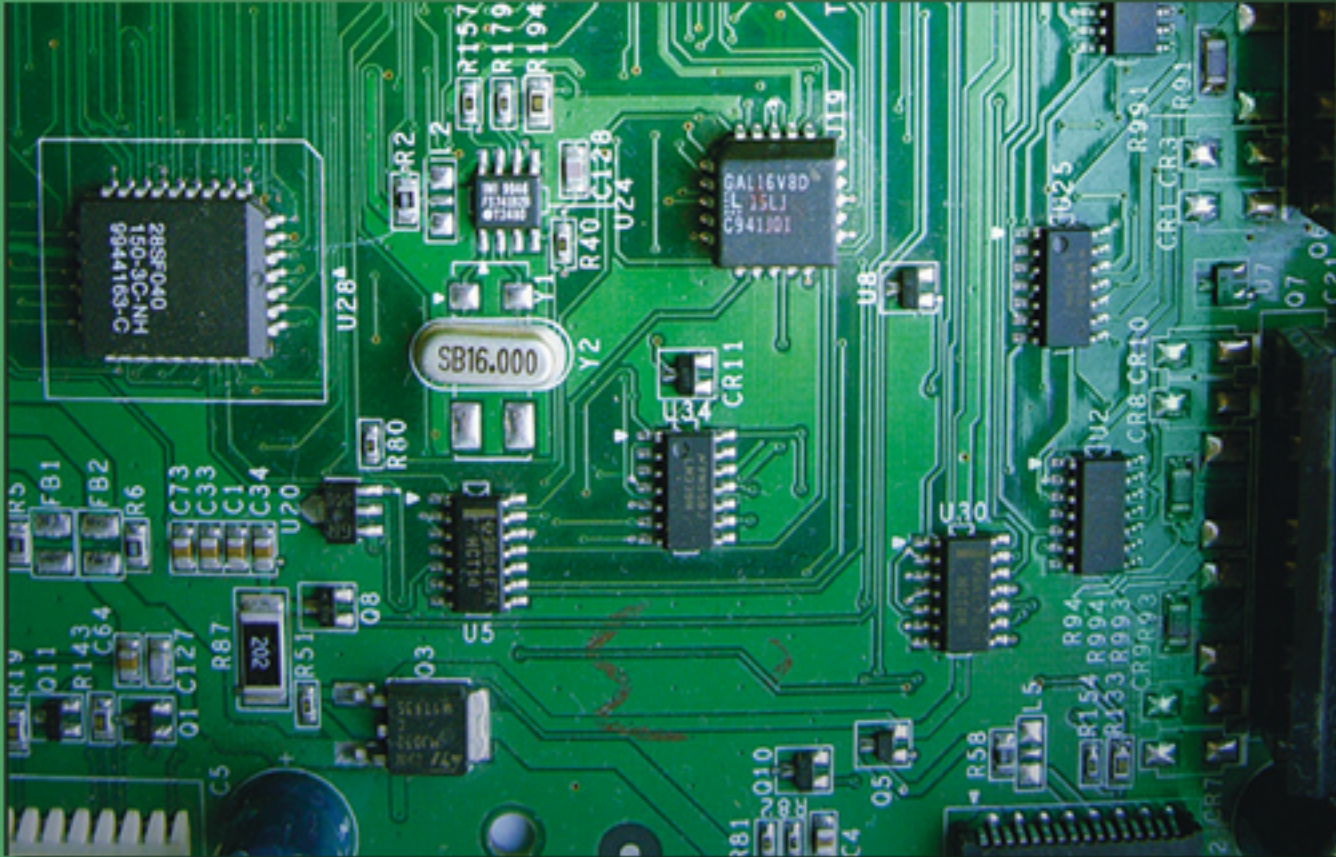
SUMMARY

- The SCAS and CMPS instructions repeat with the REPE or REPNE prefixes.
- The REPE prefix repeats the string instruction while an equal condition exists, and the REPNE repeats the string instruction while a not-equal condition exists.

The Intel Microprocessors

8086/8088, 80186/80188, 80286, 80386, 80486 Pentium, Pentium Pro Processor, Pentium II, Pentium 4, and Core2 with 64-bit Extensions

Architecture, Programming, and Interfacing



EIGHTH EDITION

Barry B. Brey



Chapter 6: Program Control Instructions

Chapter 6

Program Control Instructions

*Note: Most of slides are adapted from Barry B. Brey
(Author Slides)*

Instructor:
Dr. Khalid A. Darabkeh

Introduction

- This chapter explains the **program control instructions**, including the **jumps**, **calls**, and **returns** instructions.
- This chapter also presents the **relational assembly language statements** (.IF, .ELSE, .ELSEIF, .ENDIF, .WHILE, .ENDW, .REPEAT, and .UNTIL) that are available in version 6.xx and above of MASM or TASM, with version 5.xx set for MASM compatibility.

Chapter Objectives

Upon completion of this chapter, you will be able to:

- Use both conditional and unconditional jump instructions to control the flow of a program.
- Use the relational assembly language statements .IF, .REPEAT, .WHILE, and so forth in programs.
- Use the call and return instructions to include procedures in the program structure.

6-1 THE JUMP GROUP

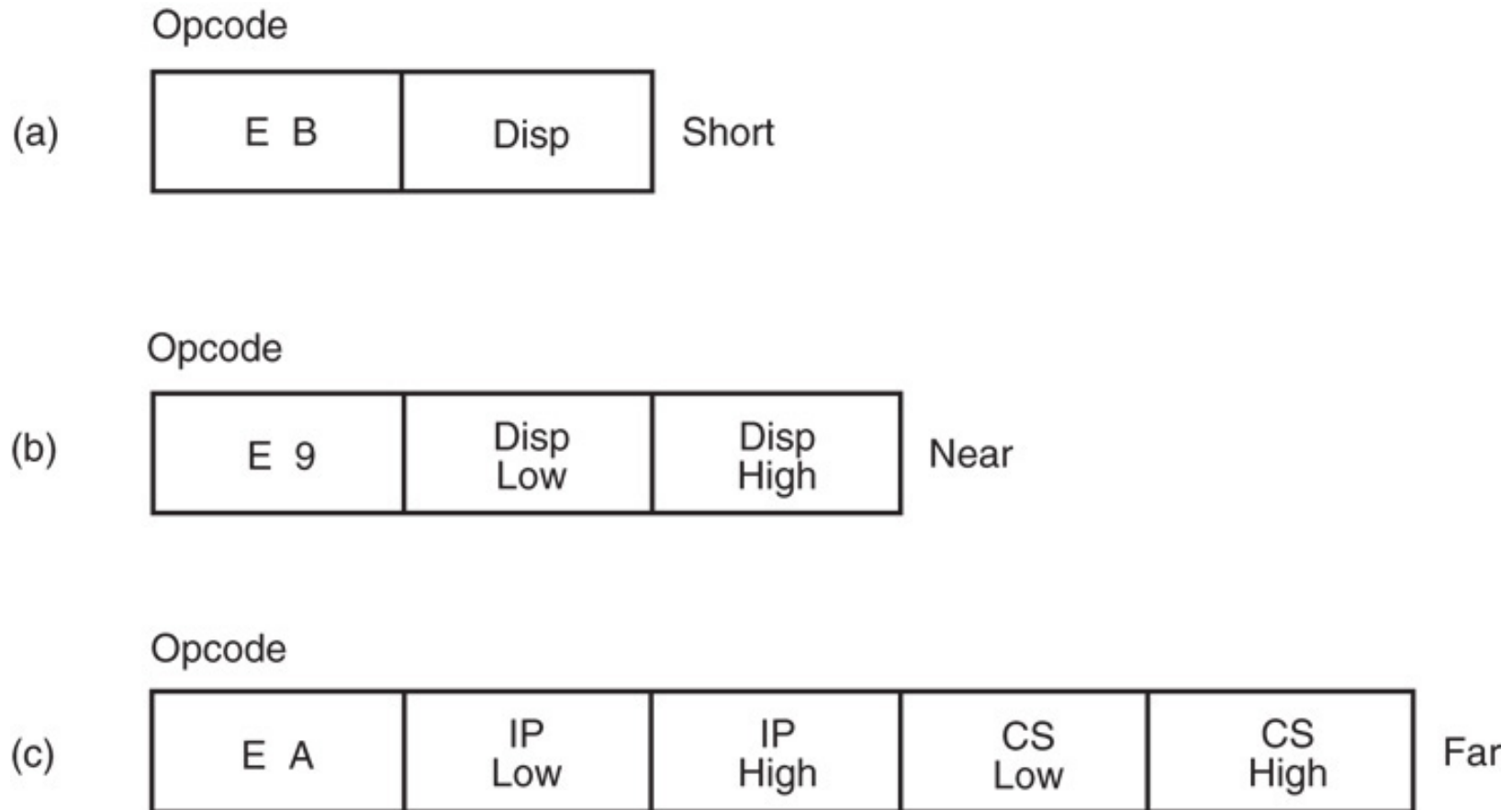
- Allows programmer to skip program sections and branch to any part of memory for the next instruction.
- A conditional jump instruction allows decisions based upon numerical tests.
 - results are held in the flag bits, then tested by conditional jump instructions
- LOOP and conditional LOOP are also forms of the jump instruction.

Unconditional Jump (**JMP**)

- Three types: **short** jump, **near** jump, **far** jump.
- **Short jump** is a **2-byte instruction** that allows jumps or branches to memory locations **within +127 and -128 bytes**.
 - from the address following the jump
- **3-byte near jump** allows a branch or jump **within ±32K bytes** from the instruction in the current code segment.

- **5-byte far jump** allows a jump to any memory location within the real memory system.
- The **short and near** jumps are often called **intra-segment jumps**.
- **Far** jumps are called **inter-segment jumps**.

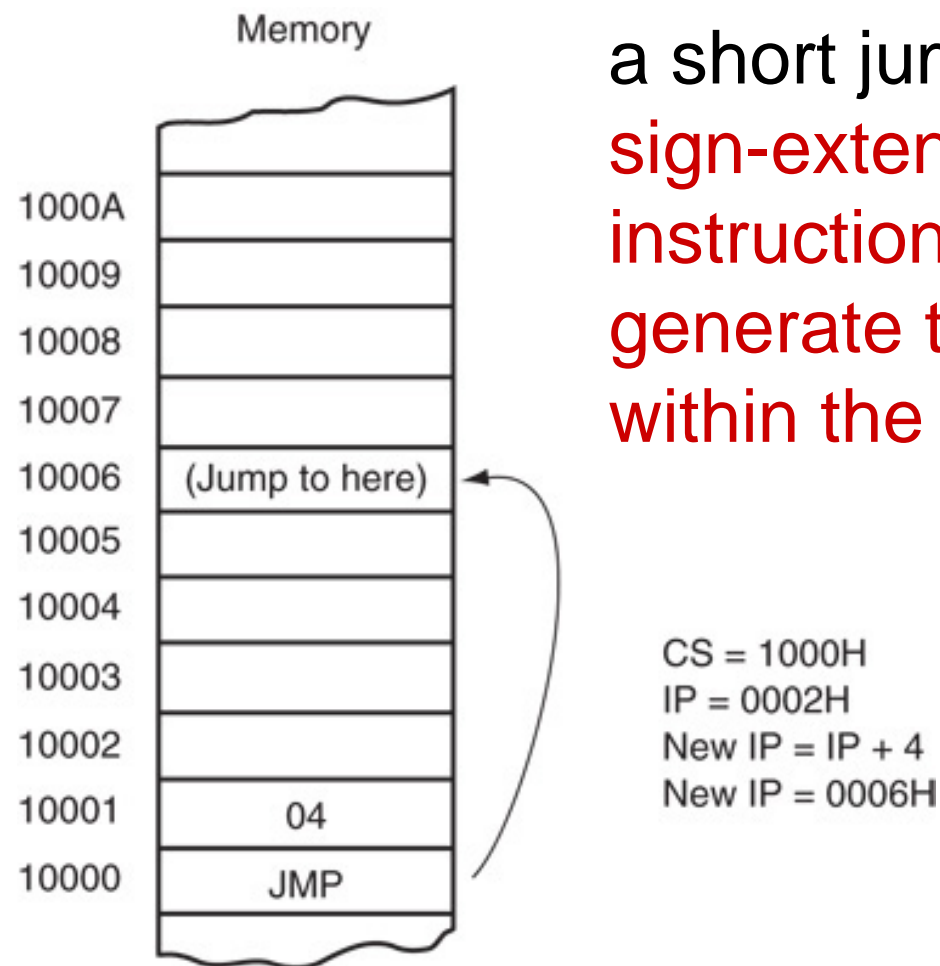
Figure 6–1 The three main forms of the JMP instruction. Note that Disp is either an 8- or 16-bit signed displacement or distance.



Short Jump

- Called **relative jumps** because they can be moved, with related software, to any location in the current code segment without a change.
 - jump address is not stored with the opcode
 - a **distance**, or **displacement**, follows the opcode
- The short jump displacement is a **distance represented by a 1-byte signed number** whose value ranges between +127 and –128.
- Short jump instruction appears in Figure 6–2.

Figure 6–2 A short jump to four memory locations beyond the address of the next instruction.



– when the microprocessor executes a short jump, the **displacement is sign-extended and added to the instruction pointer (IP/EIP) to generate the jump address within the current code segment**

– The instruction branches to this new address for the next instruction in the program

- When a jump references an address, a **label normally identifies the address**.
- The **JMP NEXT** instruction is an example.
 - it jumps to label NEXT for the next instruction
 - very rare to use an actual hexadecimal address with any jump instruction
- The **label NEXT** must be **followed by a colon** (NEXT:) to allow an instruction to reference it
 - if a colon does not follow, you cannot jump to it
- The only time a colon is used is when the label is used with a jump or call instruction.

Example

```
0000 33 db          xor bx, bx
0002 b8 0001      start: mov ax, 1
0005 03 c3        add ax, bx
0007 Eb 17        jmp short next
```

The Short directive force a short jump

<skipped memory locations>

```
0020 8b d8        next:  mov bx,ax
0022 eb de        jmp start
```

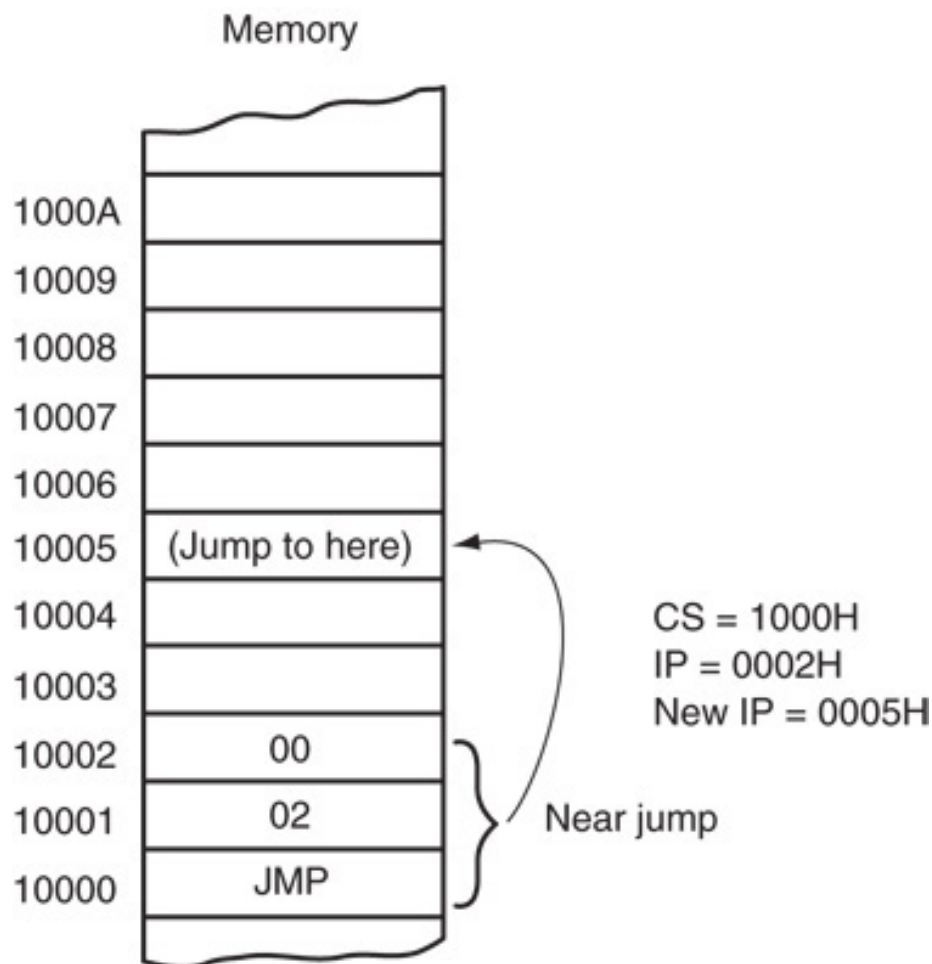
Assembles as short also; most assemblers choose the best form of the JUMP instruction

Near Jump

- A near jump passes control to an instruction in the **current code segment** located **within $\pm 32K$** bytes from the near jump instruction.
 - distance is $\pm 2G$ in 80386 and above when operated in protected mode
- Near jump is a **3-byte instruction** with opcode followed by a **signed 16-bit displacement**.
 - 80386 - Pentium 4 displacement is 32 bits and the near jump is 5 bytes long

- **Signed displacement adds** to the instruction pointer (**IP**) to generate the jump address.
 - because signed displacement is $\pm 32\text{K}$, **a near jump can jump to any memory location within the current real mode code segment**
- The protected mode code segment in the 80386 and above can be 4G bytes long.
 - 32-bit displacement allows a near jump to any location within $\pm 2\text{G}$ bytes
- Figure 6–3 illustrates the operation of the real mode near jump instruction.

Figure 6–3 A near jump that adds the displacement (0002H) to the contents of IP.



- The near jump is also **relocatable** because it is also a relative jump.
- This feature, along with the relocatable data segments, Intel microprocessors ideal for use in a general-purpose computer system.
- Software can be written and loaded anywhere in the memory and function without modification because of the relative jumps and relocatable data segments.

Example

```
0000 33 db      xor bx, bx
0002 b8 0001    start: mov ax, 1
0005 03 c3     add ax, bx
0007 E9 0200 R  jmp next
```

<skipped memory locations>

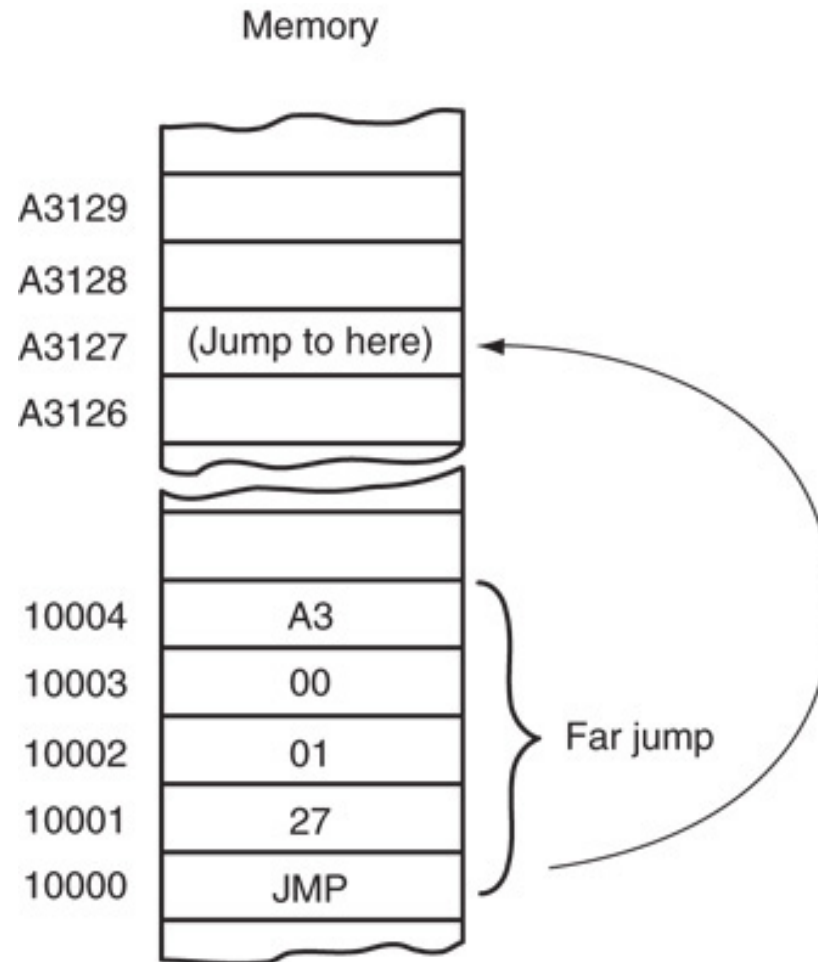
```
0200 8b d8     next: mov bx,ax
0202 e9 0002 R  jmp start
```

- The letter R denotes relocatable jump address of 0200.
- The relocatable address of 0200H is for the assembler program's internal use only. The actual machine instruction assembles as E9 F6 01, which does not appear in the assembler listing. The actual displacement is 01F6H for this jump. The assembler lists the jump address as 0200 R, so the address is easier to interpret as software is developed. If the linked execution file (.EXE) or command file (.COM) is displayed in hexadecimal code, the jump instruction appears as E9 F6 01.

Far Jump

- Obtains a **new segment and offset address** to accomplish the jump:
 - **bytes 2 and 3** of this 5-byte instruction contain the new **offset** address
 - **bytes 4 and 5** contain the new **segment** address
 - in protected mode, the segment address accesses a descriptor with the base address of the far jump segment
 - offset address, either 16 or 32 bits, contains the offset address within the new code segment

Figure 6–4 A far jump instruction replaces the contents of both CS and IP with 4 bytes following the opcode.



- The far jump instruction sometimes appears with the **FAR PTR directive**.
- Another way to obtain a far jump is to **define a label as a far label**
- A label is **far only** if it is **external to the current code segment or procedure**
- The JMP UP instruction references a far label.
- The label UP is defined as a far label by the **EXTRN UP:FAR** directive
- **External labels** appear in programs that contain more than one program file.

- Another way of defining a label as global is to use a *double colon* (LABEL::)
- When the **program files are joined**, the **linker inserts the address** for the **UP** label into the **JMP UP** instruction.
- Also **inserts segment address** in **JMP START** instruction.
- The segment address in JMP FAR PTR START is listed as ---- R for relocatable; the segment address in JMP UP is listed as ----E for external. In both cases, the ---- is filled in by the linker when it links or joins the program files.

Example

```
0000 33 db          extern up:far
0002 b8 0001      start: xor bx, bx
0005 03 c3        add ax, bx
0007 E9 0200 R    jmp next
```

<skipped memory locations>

```
0200 8b d8        next: mov bx,ax
0202 e9 0002 ---- R    jmp far ptr start
0207 ea 0000 ---- E    jmp up
```

The ---- is filled in by the linker when it links or joins the program files

Jumps with Register Operands

- Jump can also use a 16- or 32-bit register as an operand.
 - automatically sets up as an **indirect jump**
 - **address** of the jump is **in the register specified** by the jump instruction
- Unlike displacement associated with the near jump, **register contents are transferred directly into the instruction pointer.**
- An indirect jump does not add to the instruction pointer.

- **JMP AX**, for example, **copies** the contents of the **AX** register **into** the **IP**.
 - allows a jump to any location within the current code segment
- In 80386 and above, **JMP EAX** also jumps to any location within the current code segment;
 - in protected mode the code segment can be 4G bytes long, so a 32-bit offset address is needed
- Study example 6-4

Indirect Jumps Using an Index

- Jump instruction may also use the [] form of addressing to directly access the jump table.
- The jump **table** can contain **offset addresses** for near indirect jumps, **or segment and offset addresses** for far indirect jumps.
 - also known as a *double-indirect jump* if the register jump is called an *indirect jump*
- The assembler **assumes** that the jump is **near** unless the **FAR PTR** directive indicates a far jump instruction.

- Mechanism used to access the jump table is identical with a normal memory reference.
 - **JMP TABLE [SI]** instruction points to a jump address stored at the code segment offset location addressed by SI
- Both the register and indirect indexed jump instructions usually address a 16-bit offset.
 - both types of jumps are near jumps
- If **JMP FAR PTR [SI]** or **JMP TABLE [SI]**, with **TABLE** data defined with the **DD** directive:
 - microprocessor assumes the jump table contains doubleword, **32-bit addresses (IP and CS)**

Unconditional Jump (**JMP**) : SUMMARY

- The unconditional jump can take the following forms:
- 1. SHORT JUMP, which is specified by format “JMP SHORT label”. This is a jump in which the address of the target location is within -128 to +127 bytes of memory relative to the address of the current IP. The target address can be just direct addressing mode.
- 2. NEAR JUMP, **which is the default**, has the format “JMP label”. The target address can be any of the addressing modes of direct, register indirect, or memory indirect:
- Direct JUMP is exactly like the short jump explained earlier, except that the target address can be anywhere in the segment within the range +32767 to -32768 of the current IP.
- **NOTE: Most of current assemblers choose the best form of the JUMP instruction (i.e., either short or near)**

Unconditional Jump (**JMP**) : SUMMARY

- Register Indirect JUMP; the target address is in register. For example, in “JUMP BX”, IP takes the value of BX.
- Memory Indirect JUMP; Example, “JUMP [DI]” will replace the IP with the contents of memory locations pointed by DI and DI+1.
- 3. FAR JUMP which has the format “ JUMP FAR PTR label”. This is a jump out of the current code segment, meaning that not only the IP but also the CS is replaced with new values. The target address can be any of the addressing modes of direct or memory indirect (**JMP FAR PTR [SI]**).

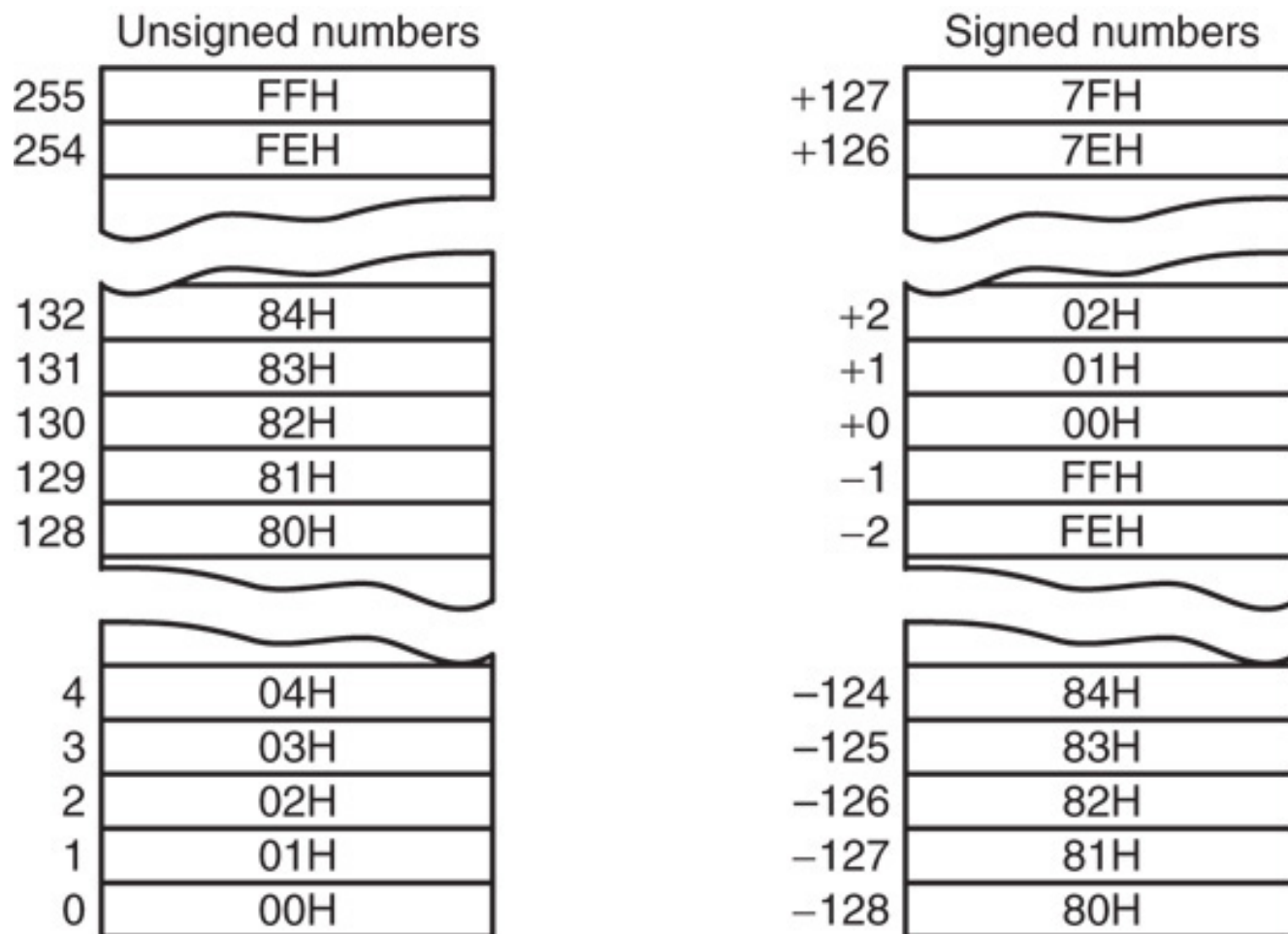
Conditional Jumps

- Always short jumps in 8086 - 80286.
 - limits range to within +127 and -128 bytes from the location following the conditional jump
- In 80386 and above, conditional jumps are either short or near jumps ($\pm 32\text{K}$).
 - in 64-bit mode of the Pentium 4, the near jump distance is $\pm 2\text{G}$ for the conditional jumps
- Allows a conditional jump to any location within the current code segment.

- Conditional jump instructions **test flag bits**:
 - sign (**S**), zero (**Z**), carry (**C**)
 - parity (**P**), overflow (**O**)
- If the condition under **test is true**, a **branch** to the label associated with the jump instruction **occurs**.
 - if **false**, **next sequential step in program executes**
 - for example, a **JC will jump if the carry bit is set**
- Most conditional jump instructions are straightforward as they often test one flag bit.
 - although some test more than one

- Because both signed and unsigned numbers are used in programming.
- Because the order of these numbers is different, there are **two sets of conditional jump instructions for magnitude comparisons.**
- 16- and 32-bit numbers follow the same order as 8-bit numbers, except that they are larger.
- Figure 6–5 shows the order of both signed and unsigned 8-bit numbers.

Figure 6–5 Signed and unsigned numbers follow different orders.



- When signed numbers are compared, use the **JG, JL, JGE, JLE, JE, and JNE** instructions.
 - terms greater than and less than refer to signed numbers
- When unsigned numbers are compared, use the **JA, JB, JAE, JBE, JE, and JNE** instructions.
 - terms above and below refer to unsigned numbers
- **Remaining conditional jumps test individual flag bits**, such as overflow and parity.

- Remaining conditional jumps test individual flag bits, such as overflow and parity.
 - notice that **JE** has an **alternative** opcode **JZ**
- **All instructions have alternates**, but many aren't used in programming because they don't usually fit the condition under test.

Opcode	Instruction	Description
77 <i>cb</i>	<i>JA rel8</i>	Jump short if above (CF=0 and ZF=0)
73 <i>cb</i>	<i>JAE rel8</i>	Jump short if above or equal (CF=0)
72 <i>cb</i>	<i>JB rel8</i>	Jump short if below (CF=1)
76 <i>cb</i>	<i>JBE rel8</i>	Jump short if below or equal (CF=1 or ZF=1)
72 <i>cb</i>	<i>JC rel8</i>	Jump short if carry (CF=1)
E3 <i>cb</i>	<i>JCXZ rel8</i>	Jump short if CX register is 0
E3 <i>cb</i>	<i>JECXZ rel8</i>	Jump short if ECX register is 0
74 <i>cb</i>	<i>JE rel8</i>	Jump short if equal (ZF=1)
7F <i>cb</i>	<i>JG rel8</i>	Jump short if greater (ZF=0 and SF=OF)
7D <i>cb</i>	<i>JGE rel8</i>	Jump short if greater or equal (SF=OF)
7C <i>cb</i>	<i>JL rel8</i>	Jump short if less (SF<>OF)
7E <i>cb</i>	<i>JLE rel8</i>	Jump short if less or equal (ZF=1 or SF<>OF)
76 <i>cb</i>	<i>JNA rel8</i>	Jump short if not above (CF=1 or ZF=1)
72 <i>cb</i>	<i>JNAE rel8</i>	Jump short if not above or equal (CF=1)
73 <i>cb</i>	<i>JNB rel8</i>	Jump short if not below (CF=0)
77 <i>cb</i>	<i>JNBE rel8</i>	Jump short if not below or equal (CF=0 and ZF=0)
73 <i>cb</i>	<i>JNC rel8</i>	Jump short if not carry (CF=0)
75 <i>cb</i>	<i>JNE rel8</i>	Jump short if not equal (ZF=0)
7E <i>cb</i>	<i>JNG rel8</i>	Jump short if not greater (ZF=1 or SF<>OF)
7C <i>cb</i>	<i>JNGE rel8</i>	Jump short if not greater or equal (SF<>OF)
7D <i>cb</i>	<i>JNL rel8</i>	Jump short if not less (SF=OF)
7F <i>cb</i>	<i>JNLE rel8</i>	Jump short if not less or equal (ZF=0 and SF=OF)
71 <i>cb</i>	<i>JNO rel8</i>	Jump short if not overflow (OF=0)
7B <i>cb</i>	<i>JNP rel8</i>	Jump short if not parity (PF=0)
79 <i>cb</i>	<i>JNS rel8</i>	Jump short if not sign (SF=0)
75 <i>cb</i>	<i>JNZ rel8</i>	Jump short if not zero (ZF=0)
70 <i>cb</i>	<i>JO rel8</i>	Jump short if overflow (OF=1)
7A <i>cb</i>	<i>JP rel8</i>	Jump short if parity (PF=1)
7A <i>cb</i>	<i>JPE rel8</i>	Jump short if parity even (PF=1)
7B <i>cb</i>	<i>JPO rel8</i>	Jump short if parity odd (PF=0)
78 <i>cb</i>	<i>JS rel8</i>	Jump short if sign (SF=1)
74 <i>cb</i>	<i>JZ rel8</i>	Jump short if zero (ZF = 1)
0F 87 <i>cw/cd</i>	<i>JA rel16/32</i>	Jump near if above (CF=0 and ZF=0)
0F 83 <i>cw/cd</i>	<i>JAE rel16/32</i>	Jump near if above or equal (CF=0)
0F 82 <i>cw/cd</i>	<i>JB rel16/32</i>	Jump near if below (CF=1)
0F 86 <i>cw/cd</i>	<i>JBE rel16/32</i>	Jump near if below or equal (CF=1 or ZF=1)
0F 82 <i>cw/cd</i>	<i>JC rel16/32</i>	Jump near if carry (CF=1)
0F 84 <i>cw/cd</i>	<i>JE rel16/32</i>	Jump near if equal (ZF=1)
0F 84 <i>cw/cd</i>	<i>JZ rel16/32</i>	Jump near if 0 (ZF=1)
0F 8F <i>cw/cd</i>	<i>JG rel16/32</i>	Jump near if greater (ZF=0 and SF=OF)

LOOP

- A combination of a **decrement CX** and the **JNZ** conditional jump.
- In 8086 - 80286 LOOP decrements CX.
 - if **CX != 0**, it **jumps** to the address indicated by the label
 - If **CX becomes 0**, the **next sequential instruction** executes
- In **80386 and above**, LOOP decrements either **CX or ECX**, depending upon instruction mode.

- In 16-bit instruction mode, LOOP uses CX; in the 32-bit mode, LOOP uses ECX.
 - default is changed by the LOOPW (using CX) and LOOPD (using ECX) instructions 80386 - Core2
- In 64-bit mode, the loop counter is in RCX.
 - and is 64 bits wide
- There is no direct move from segment register to segment register instruction.
- Study example 6-7

Conditional *LOOPS*

- LOOP instruction also has conditional forms:
LOOPE and **LOOPNE**
- LOOPE (**loop while equal**) instruction jumps if CX \neq 0 while an equal condition exists.
 - will **exit loop if the condition is not equal** or the **CX register decrements to 0**
- LOOPNE (**loop while not equal**) jumps if CX \neq 0 while a not-equal condition exists.
 - will **exit loop if the condition is equal** or the **CX register decrements to 0**

- In 80386 - Core2 processors, conditional LOOP can use CX or ECX as the counter.
 - LOOPEW/LOOPED or LOOPNEW/LOOPNEED override the instruction mode if needed
- Under 64-bit operation, the loop counter uses RCX and is 64 bits in width
- Alternates exist for LOOPE and LOOPNE.
 - LOOPE same as LOOPZ
 - LOOPNE instruction is the same as LOOPNZ
- In most programs, only the LOOPE and LOOPNE apply.

EXAMPLE:

```
DATSEG segment
DATA1 DB '123485559BCDEFGHIJK'
MESSAGE1 DB 'FOUND','$'
MESSAGE2 DB 'NOT FOUND','$'
DATSEG ENDS
CODE_SEG          SEGMENT
                  ASSUME CS:CODE_SEG, DS:DATSEG
MAIN             PROC FAR
                  MOV AX, DATSEG
                  MOV ES, AX
                  MOV DS, AX

                  MOV AH, 01; Option 01 inputs one character
                  INT 21H ; After this interrupt, AL=input character
                  MOV CX, 19
                  MOV SI, offset DATA1
GO:              MOV BL, [SI]
                  INC SI
                  CMP AL, BL
                  LOOPNE GO
                  JE FOUND ;ZF=1
                  MOV DX, Offset MESSAGE2
                  JMP NOTFOUND
                  ;-----
FOUND:           MOV DX, Offset MESSAGE1
NOTFOUND:       MOV AH, 09
                  INT 21H
MAIN             ENDP
CODE_SEG        ENDS
END             MAIN
```

6–2 CONTROLLING THE FLOW OF THE PROGRAM

- Easier to use assembly language statements **.IF**, **.ELSE**, **.ELSEIF**, and **.ENDIF** to control the flow of the program than to use the correct conditional jump statement.
 - these statements always indicate a special assembly language command to MASM
- Control flow assembly language statements beginning with a period available to MASM version 6.xx, and not to earlier versions.

- Other statements developed include **.REPEAT–.UNTIL** and **.WHILE–.ENDW**.
 - **the dot commands** do not function using the Visual C++ inline assembler
- Never use uppercase for assembly language commands with the inline assembler.
 - some of them are reserved by C++ and will cause problems

EXAMPLE 6-8(a)

```
.IF AL >= 'A' && AL <= 'F'  
    SUB AL, 7  
.ENDIF  
SUB AL, 30H
```

<i>Operator</i>	<i>Function</i>
==	Equal or the same as
!=	Not equal
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
&	Bit test
!	Logical inversion
&&	Logical AND
	Logical OR
	OR

WHILE Loops

- Used with a condition to begin the loop.
 - the `.ENDW` statement ends the loop
- The `.BREAK` and `.CONTINUE` statements are available for use with the `while` loop.
 - `.BREAK` is often followed by `.IF` to select the break condition as in `.BREAK .IF AL == 0DH`
 - `.CONTINUE` can be used to allow a `DO–.WHILE` loop to continue if a certain condition is met
- The `.BREAK` and `.CONTINUE` commands function the same manner in C++.

REPEAT-UNTIL Loops

- A **series** of instructions is **repeated** until some condition occurs.
- The **.REPEAT** statement defines the **start** of the loop.
 - **end** is defined with the **.UNTIL** statement, which contains a condition
- An **.UNTILCXZ** instruction uses the **LOOP** instruction to **check CX** for a repeat loop.
 - **.UNTILCXZ** uses the **CX** register as a counter to repeat a loop a **fixed** number of **times**

6–3 PROCEDURES

- A procedure is a group of instructions that usually performs one task.
 - subroutine, method, or **function** is an important part of any system's architecture
- A procedure is a **reusable** section of the software stored in memory once, used as often as necessary.
 - **saves memory** space and makes it easier to develop software

- Disadvantage of procedure is **time** it **takes** the computer to link to, and return from it.
 - **CALL** links to the procedure; the **RET** (**return**) instruction returns from the procedure
- **CALL** pushes the address of the instruction following the **CALL** (**return address**) on the stack.
 - the stack stores the return address when a procedure is called during a program
- **RET** instruction removes an address from the stack so the program returns to the instruction following the **CALL**.

- A procedure begins with the **PROC** directive and ends with the **ENDP** directive.
 - each directive appears with the procedure name
- **PROC** is followed by the **type** of procedure:
 - **NEAR** or **FAR**
- In MASM version 6.x, the **NEAR** or **FAR** type can be followed by the **USES** statement.
 - **USES** allows **any** number of **registers** to be automatically **pushed** to the stack and **popped** from the stack within the procedure

- Procedures that are to be used by all software (**global**) should be written as **far** procedures.
- Procedures that are used by a given task (**local**) are normally defined as **near** procedures.
- Most procedures are near procedures.
- **Near RET** pop a 16-bit number from the stack and save it into the IP.
- **Far RET** pop a 32-bit number from the stack and save it into the IP and CS.

EXAMPLE 6-14

0000		SUMS	PROC NEAR				
0000	03	C3	ADD	AX, BX			
0002	03	C1	ADD	AX, CX			
0004	03	C2	ADD	AX, DX			
0006	C3		RET				
0007		SUMS	ENDP				
0007		SUMS1	PROC FAR				
0007	03	C3	ADD	AX, BX			
0009	03	C1	ADD	AX, CX			
000B	03	C2	ADD	AX, DX			
000D	CB		RET				
000E		SUMS1	ENDP				
000E		SUMS3	PROC NEAR	USE	BX	CX	DX
0011	03	C3	ADD	AX, BX			
0013	03	C1	ADD	AX, CX			
0015	03	C2	ADD	AX, DX			
			RET				
001B		SUMS	ENDP				

CALL

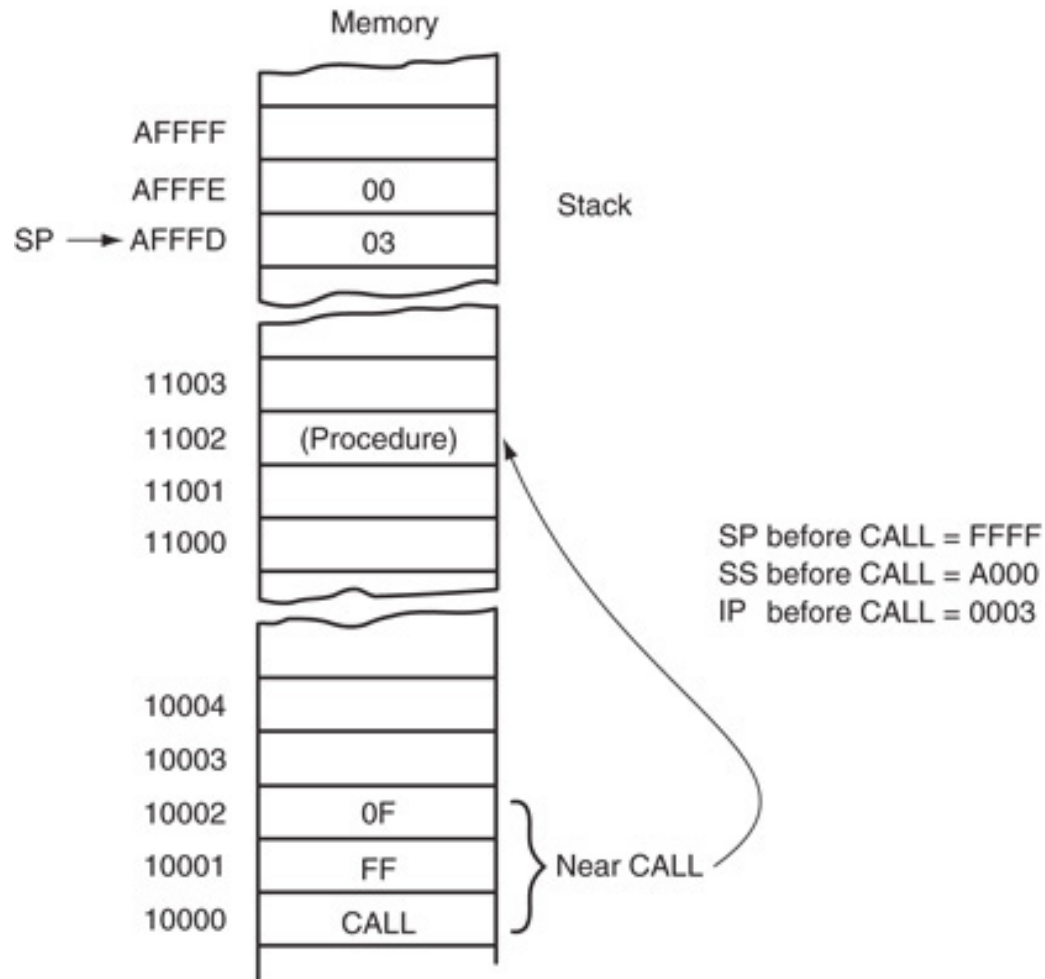
- **Transfers** the flow of the program to the **procedure**.
- **CALL** instruction **differs** from the **jump** instruction because a **CALL** **saves** a **return** address on the stack.
- The return address returns control to the instruction that immediately follows the **CALL** in a program when a **RET** instruction executes.

Near CALL

- **3 bytes** long.
 - the first byte contains the **opcode**; the second and third bytes contain the **displacement**
- When the near CALL executes, it **first pushes** the **offset** address of the next instruction onto the stack.
 - offset address of the next instruction appears in the instruction pointer (IP or EIP)
- It **then adds displacement** from bytes 2 & 3 **to the IP** to transfer control to the procedure.

- **Why save the IP or EIP on the stack?**
 - the instruction pointer always points to the next instruction in the program
- For the CALL instruction, the contents of IP/EIP are pushed onto the stack.
 - program control passes to the instruction following the CALL after a procedure ends
- Figure 6–6 shows the return address (IP) stored on the stack and the call to the procedure.

Figure 6–6 The effect of a near **CALL** on the stack and the instruction pointer.

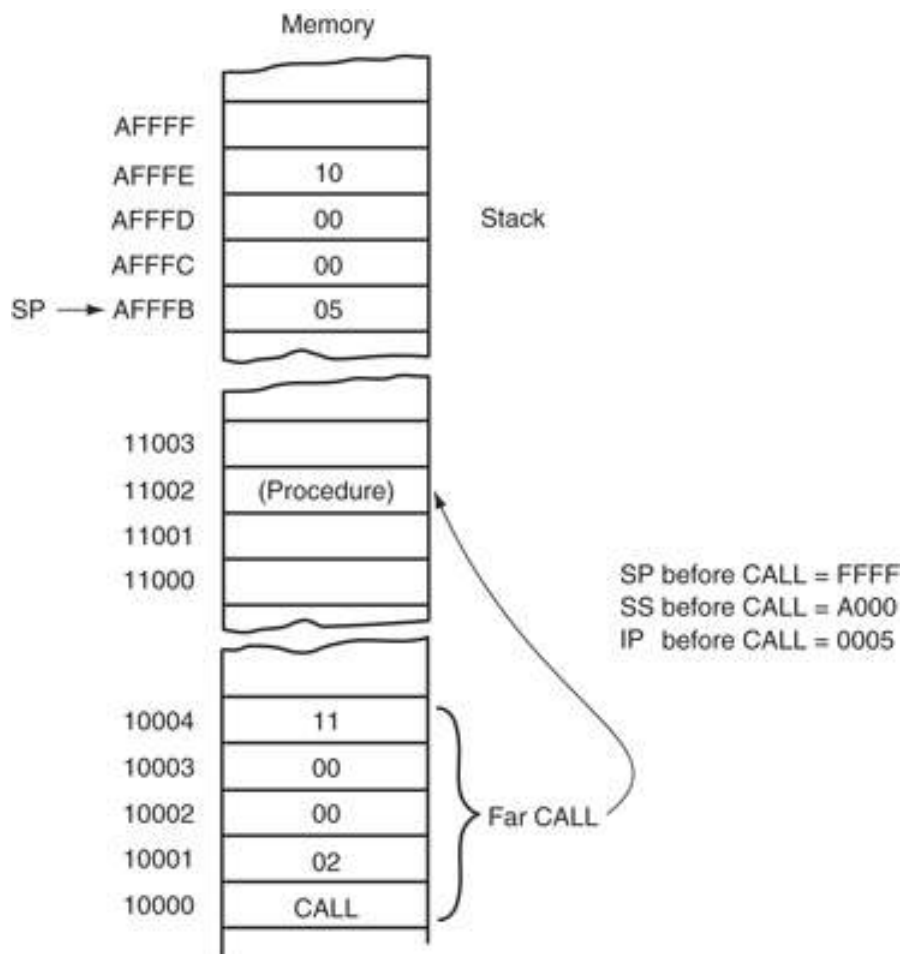


Far CALL

- **5-byte** instruction contains an **opcode** followed by the next value for the **IP** and **CS** registers.
 - bytes 2 and 3 contain new contents of the IP
 - bytes 4 and 5 contain the new contents for CS
- **Far CALL** places the contents of both **IP** and **CS** on the stack before jumping to the address indicated by bytes 2 through 5.
- This allows far CALL to call a procedure located **anywhere** in the **memory** and **return** from that procedure.

- Figure 6–7 shows how far CALL calls a far procedure.
 - contents of IP and CS are pushed onto the stack
- The program branches to the procedure.
 - A variant of far call exists as CALLF, but should be avoided in favor of defining the type of call instruction with the PROC statement
- In 64-bit mode a far call is to any memory location and information placed onto the stack is an 8-byte number.
 - the far return instruction retrieves an 8-byte return address from the stack and places it into RIP

Figure 6–7 The effect of a far CALL instruction.



CALLs with Register Operands

- An example **CALL BX**, which pushes the contents of **IP** onto the **stack**.
 - then jumps to the offset address, located in register **BX**, in the current **code segment**
- Always uses a 16-bit offset address, stored in any 16-bit register **except** segment registers.

EXAMPLE 6-15

```
                                ;A DOS program that displays OK using the DISP procedure.
                                ;
                                .MODEL TINY                                ;select tiny model
0000                                .CODE                                ;start code segment
                                .STARTUP                                ;start program
0100 BB 0110 R                    MOV  BX,OFFSET DISP                ;load BX with offset DISP
0103 B2 4F                        MOV  DL,'O'                            ;display O
0105 FF D3                        CALL BX
0107 B2 4B                        MOV  DL,'K'                            ;display K
0109 FF D3                        CALL BX

                                .EXIT
                                ;
0110                                DISP  PROC  NEAR
0110 B4 02                        MOV  AH,2                                ;select function 2
0112 CD 21                        INT  21H                                ;execute DOS function 2
0114 C3                        RET
0115                                DISP  ENDP
                                END
```

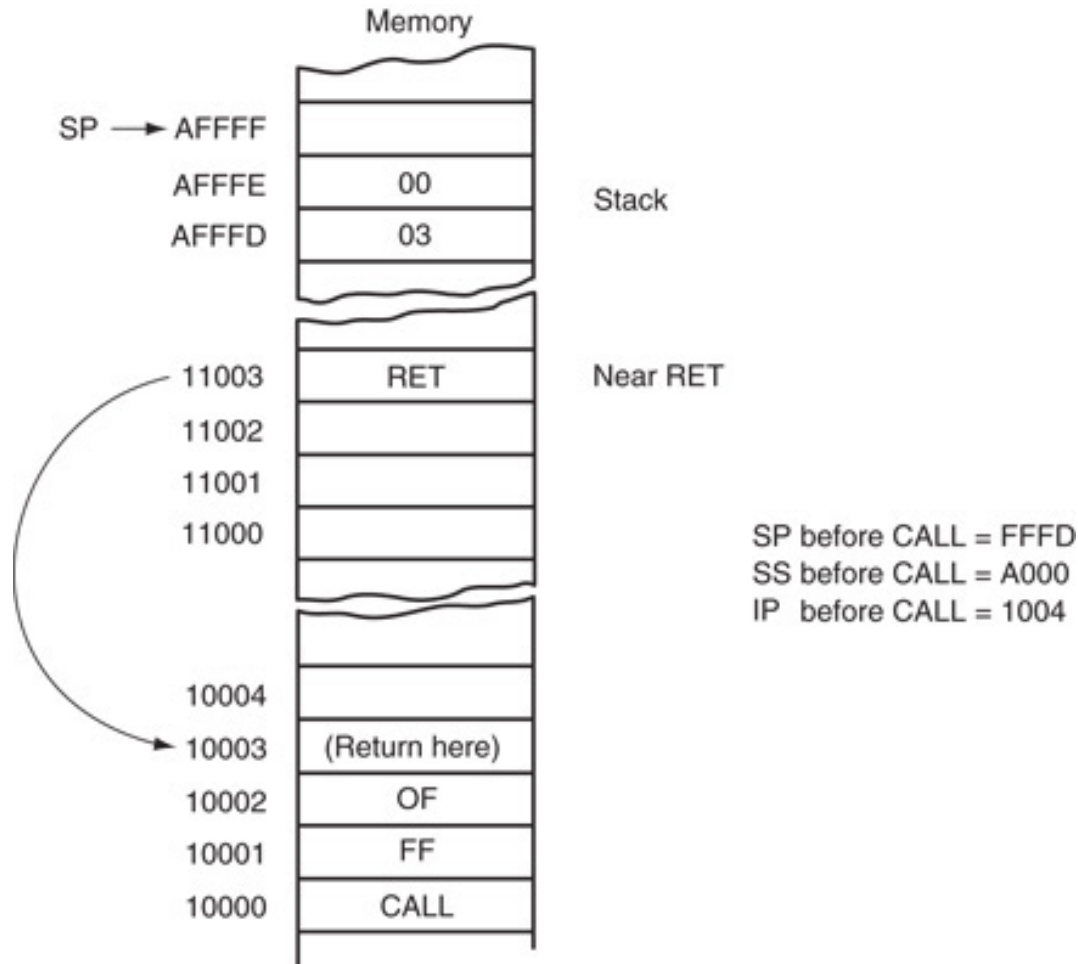

CALLs with Indirect Memory Addresses

- Particularly useful when **different subroutines need to be chosen in a program.**
 - selection process is often keyed with a number that addresses a CALL address in a **lookup table**
- Essentially the **same as the indirect jump** that used a lookup table for a jump address.

RET

- Removes a **16-bit number (near return)** from the stack placing it in **IP**, or removes a **32-bit number (far return)** and places it in **IP & CS**.
 - near and far return instructions in procedure's PROC directive
 - **automatically selects the proper return instruction**
- Figure 6–8 shows how the CALL instruction links to a procedure and how RET returns in the 8086–Core2 operating in the real mode.

Figure 6–8 The effect of a near return instruction on the stack and instruction pointer.



- Another form of return adds a number to the contents of the stack pointer (SP) after the return address is removed from the stack.
- A return that uses an **immediate** operand is ideal for use in a system that uses the C/C++ or PASCAL calling conventions.
 - these conventions **push parameters on the stack before calling a procedure**
- If the parameters are discarded upon return, the return instruction contains the **number of bytes pushed to the stack as parameters.**

- Parameters are addressed on the stack by using the BP register, which by default addresses the stack segment.
- Parameter stacking is common in procedures written for C++ or PASCAL by using the C++ or PASCAL calling conventions.
- Variants of the return instruction:
 - RETN and RETF
- Variants should also be avoided in favor of using the PROC statement to define the type of call and return.

EXAMPLE 6-17

```
0000 B8 001E      MOV  AX, 30
0003 BB 0028      MOV  BX, 40
0006 50           PUSH AX                ;stack parameter 1
0007 53           PUSH BX                ;stack parameter 2
0008 E8 0066      CALL ADDM              ;add stack parameters

0071             ADDM  PROC NEAR
0071 55           PUSH BP                ;save BP
0072 8B EC        MOV  BP, SP            ;address stack with BP
0074 8B 46 04     MOV  AX, [-BP+4]      ;get parameter 1
0077 03 46 06     ADD  AX, [BP+6]      ;add parameter 2
007A 5D           POP  BP                ;restore BP
007B C2 0004      RET  4                ;return, dump parameters
007E             ADDM  ENDP
```

SUMMARY

(*cont.*)

- There are three types of unconditional jump instructions: short, near, and far.
- The short jump allows a branch to within +127 and -128 bytes. The near jump (using a displacement of $\pm 32\text{K}$) allows a jump to any location in the current code segment (intra-segment). The far jump allows a jump to any location in the memory system (inter-segment).

SUMMARY

(*cont.*)

- Whenever a label appears with a JMP instruction or conditional jump, the label, located in the label field, must be followed by a colon (LABEL:). For example, the JMP DOGGY instruction jumps to memory location DOGGY:.
- The displacement that follows a short or near jump is the distance from the next instruction to the jump location.

SUMMARY

(*cont.*)

- Indirect jumps are available in two forms:
(1) jump to the location stored in a register
and (2) jump to the location stored in a
memory word (near indirect) or doubleword
(far indirect).
- Conditional jumps are all short jumps that
test one or more of the flag bits: C, Z, O, P,
or S. If the condition is true, a jump occurs;
if the condition is false, the next sequential
instruction executes.

SUMMARY

(*cont.*)

- The 80386 and above allow a 16-bit signed displacement for the conditional jump instructions.
- In 64-bit mode, the displacement is 32 bits allowing a range of $\pm 2G$.
- A special conditional jump instruction (LOOP) decrements CX and jumps to the label when CX is not 0.

SUMMARY

(*cont.*)

- The `.IF` and `.ENDIF` statements are useful in assembly language for making decisions.
- The instructions cause the assembler to generate conditional jump statements that modify the flow of the program.
- The `.WHILE` and `.ENDW` statements allow an assembly language program to use the `WHILE` construction, and the `.REPEAT` and `.UNTIL` statements allow use of the `REPEAT-UNTIL` construct.

SUMMARY

(*cont.*)

- Procedures are groups of instructions that perform one task and are used from any point in a program.
- The CALL instruction links to a procedure and the RET instruction returns from a procedure. In assembly language, the PROC directive defines the name and type of procedure.
- The ENDP directive declares the end of the procedure.

SUMMARY

(*cont.*)

- The CALL construction is a combination of a PUSH and a JMP instruction.
- When CALL executes, it pushes the return address on the stack and then jumps to the procedure.
- A near CALL places the contents of IP on the stack, and a far CALL places both IP and CS on the stack.

SUMMARY

(*cont.*)

- The RET instruction returns from a procedure by removing the return address from the stack and placing it into IP (near return), or IP and CS (far return).
- Interrupts are either software instructions similar to CALL or hardware signals used to call procedures. This process interrupts the current program and calls a procedure.
- After the procedure, a special IRET instruction returns control to the software.