Module 2: PC Hardware Features and Introduction to Assembly Language Programming

Lesson 1: Internal Memory, Register and Hardware Interrupts

Learning Outcomes

- LO 2.1.1 Identify the different parts of a computer system, and implement assembly and disassembling of the various components.
- LO 2.1.1 Explain the basic operating modes and internal units of an x86 Processor.

Introduction

The main component of the PC is its system board (or motherboard). It contains the processor, main memory, connectors, and expansion slots for optional cards. The slots and connectors provide access to such components as read only-memory (ROM), random access memory (RAM), hard disk, CD-ROM drives, additional main memory, video units, keyboard, mouse, parallel and serial devices, and sound synthesizers.

A bus with wires attached to the system board connects the components. It transfers data between the processor, memory, and external devices, in effect managing data traffic. When a program, for example, requests reading data from an external storage device, the processor determines the address in memory where it is to be delivered and places the address on an address bus. The memory unit then delivers the data to a data bus and notifies the processor that the data is ready. The processor now delivers the data from the data bus to addressed location in memory.

The power supply converts the 220-volt alternating current into direct current with voltage according to the computer requirements. Power supplies are commonly rated 300 watts.

The Processor

The brain of the PC is a processor (also known as central processing unit, or CPU) based on the Intel 8086 family that performs all executing of instruction and processing data. Processors vary in speed and capacity of memory, registers, and data bus. An internal clock synchronizes and controls all the processor operations. The basic time unit, the clock cycle, is rated in terms of megahertz (millions of cycles per second). Below is a brief description of Intel microprocessors.

The Intel 8008, introduced in 1972, was the first commercial 8-bit microprocessor (it transferred information eight bits at a time), and is still considered the foremost "first-generation" 8-bit microprocessor. Designed with a calculator-like architecture, the 8008 had an accumulator, six scratch pad registers, a stack pointer (a special address register for temporary storage), eight address registers, and special instructions to perform input and output.

The Intel 8088, has 16 bit registers and 8 bit data bus and can address up to 1 million bytes of internal memory. Although the registers can process two bytes at a time the data bus can transfer only one byte at a time. This processor runs in what is known a *real mode*, that is, one program at a time, with actual ("real") address in the segment registers.

The Intel 8086, is a 16-bit microprocessor chip designed by Intel and introduced on the market in 1978, which gave rise to the x86 architecture. It is 10x faster than its predecessor Intel 8080.This processor similar to the 8088, but has 16 bit data bus and runs faster.

The Intel 80286 Processor, Runs faster than the preceding processors, has additional capabilities, and can address up to 16 million bytes. This processor and its successors can operate in real mode or in protected mode, which enables an operating system like

Windows to perform multi-tasking (running more than one job concurrently) and to protect them from each other.
The Intel 80386 Has 32 bit registers and a 32-bit data bus and can address up to 4 billion bytes of memory. As well as protected mode, the processor supports virtual mode, whereby it can swap portions of memory onto disk, in this way, programs run concurrently have space to operate.
The Intel 80486 Also has 32-bit registers and a 32 bit data bus. As well, high speed *cache memory* connected to the processor bus enables the processor to store copies of the most recently used instruction and data. The processor can operate faster when using the cache directly without having to access the slower main memory.
The Intel Pentium has 32-bit registers, a 64-bit data bus, and separate caches of data and for memory. Adopted by the name Pentium in contrast to numbers, names can be copyrighted. Its superscalar design enables the processor to decode and execute more than one instruction per clock.
The Intel Pentium II and Pentium III, have dual Independent Bus design that provides separate paths to the system cache and to memory. Where the previous processors' connection to a storage cache on the system board caused delays, these processors are connected to a built-in storage cache by a 64-bit wide bus.
Processors up through the 80486 have what is known as a single-stage pipeline, which restricts them to completing one instruction before starting the next. Pipelining involves the way a processor divides an instruction into sequential steps using different resources. The Pentium has a five-stage pipelined structure, and the Pentium II has 12-stage super-pipelined structure. This feature enables many programs to run in parallel.
A problem faced by designers is that because the processor runs considerably faster than does memory, it has to wait for memory to deliver instructions. To handle this problem, each advanced processor in turn has more capability for dynamic execution. For example, by means of multiple branch prediction, the processor looks ahead a number of steps to predict what to process next.

2.1.1 Operating Modes:

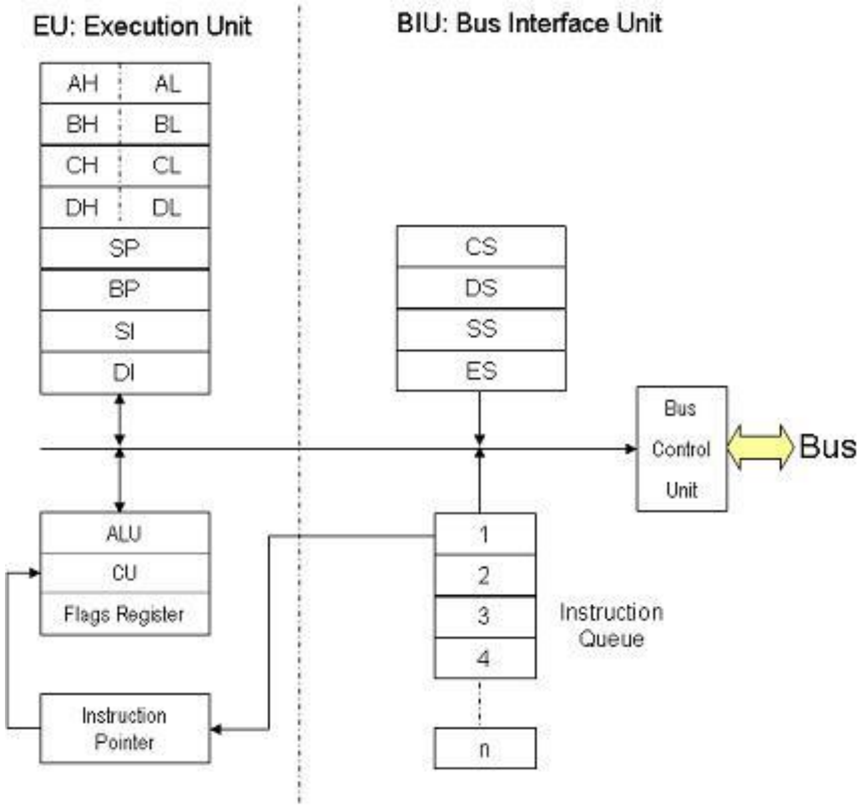
Real Address Mode - In the real address mode, the Intel x86 processor essentially operates as a high-performance 8086. Here, it supports all of the 8086 instructions (plus some of its own), but runs programs two to five times faster than an 8086. When you turn the computer's power on, the x86 Processor starts in the real address mode, it stays in this mode until a program explicitly switches it to protected mode.

Protected Mode - In the protected mode, the x86 processor provides everything it does in the real address mode plus some sophisticated features for data protection and memory management. The most significant aspect of the protected mode is that it allows the x86 Processor to access huge amounts of memory using a technique called "virtual addressing."

2.2.2 Internal Units

Many microprocessors execute a program by reading an instruction from memory, decoding it, executing it, then reading the next instruction, and so on. This plodding, one-at-a-time approach naturally slows the processor down, because it must wait until each new instruction has been read and decoded before executing it. The 80286 eliminates much of this delay by assigning these three tasks—reading, decoding, and executing instructions—to separate, special-purpose units within the chip. Intel calls them the Bus Unit, Instruction Unit, Execution Unit, and Address Unit. (The 8086 and 8088 have no address unit.)

Bus Unit (BU) - The Bus Unit (BU) is the 286's mail carrier; its job is to read instructions from memory and pass data between the processor and the "outside world." The BU puts each new instruction in a code queue, where it is available for access by the chip's Instruction Unit.
Instruction Unit (IU) - The Instruction Unit (IU) grabs instructions from the BU's code queue, decodes them, and places the code in its "instruction queue," or pipeline—the electronic equivalent of a vending machine. The pipeline can hold up to three decoded instructions.
Execution Unit (EU) - The Execution Unit (EU) executes instructions under control of its built-in "microcode" ROM (read-only memory). When it nears completion of the current instruction, the ROM signals the EU to take the next one out of the instruction queue.
Address Unit (AU) - The Address Unit (AU) performs memory management and protection functions by translating virtual addresses to physical addresses and checking protection rights, all simultaneously



Lesson 2: Internal Memory, Register and Hardware Interrupts

Learning Outcomes

- LO 2.2.1 Introduce the model of memory segments and addressing.
- LO 2.2.2 Explain the concepts on how data is addressed in the memory and calculate the specific memory address given the base and displacement.
- LO 2.2.3 Explain and define the uses of various x86 Internal register. (Segment, General Purpose, Index, Flag Registers)

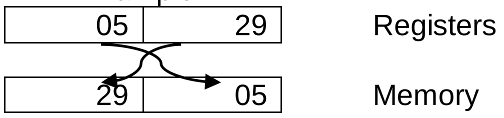
ROM: Consists of special memory chips that (as the full name suggests) can only be read. Because instructions and data are permanently "burned into" the chips, they cannot be altered. The ROM Basic Input/Output System (BIOS) begins at address 768K and handles input/output devices, such as a hard disk controller. ROM beginning at 960K controls the computer's basic functions, such as power-on self-test, dot-patterns for graphics, and the disk self-loader. When you switch on the power, ROM performs various check-outs and loads special system data from disk into RAM.

RAM: A programmer is mainly concerned with RAM, which would be better named "read-write memory". RAM is available as a "worksheet" for temporary storage and execution of programs. When you turn on the power, the ROM boot-up procedure loads a portion of the operating system into RAM. You then request it to perform actions such as loading a program from disk into RAM. Your program executes in RAM and normally produces output on the screen, printer, or disk. When finished you may ask the system to load another program into RAM, an action that overwrites the previous program.
Turning off the power erases the contents of RAM but does not affect the ROM. Consequently, if you have been changing data in document, you need to save it on disk before shutting down the PC.

2.1.2 Addressing Data in Memory

Depending on the model, the processor can access one or more bytes of memory at a time. Consider the decimal number 1,315. The hex representation of this value, 0529H, requires two bytes, or one word, of memory. It consists of a high-order (most significant) byte, 05, and a low-order (least significant) byte, 29. The processor stores the data in memory in reverse byte sequence: the low-order byte in the low memory address and the high order byte in the high memory address.

Example:



The processor expects numeric data in memory to be in reverse-byte sequence and processes the data accordingly. When the processor retrieves the word from memory, it reverses it again, restoring them correctly to as hex 0529.
There are 2 types of addressing schemes:

- An **absolute address**, such as 04A26H, is a 20-bit value that directly references a specific location in memory
- A **segment:offset** address combines the starting segment with an offset (displacement) value

Segments and Addressing

Segments are special areas defined in a program for containing the code, the data, and what is known as the stack. A segment begins on a paragraph boundary, that is, at a location evenly divisible by 16, or Hex 10. Although a segment may be located almost anywhere in memory and in real mode may be up to 64Kbytes, it requires only as much space as the program requires for its use; that is, data and instruction that process the data. There are 4 types of segment

- **Code segment** contains the machine instruction that is to execute. Typically, the first executable instruction is at the start of this segment, and the operating system links to that location to begin program execution. The Code segment (CS) register addresses the code segment
- **Data segment** contains the program's defined data, constants, and work areas. The Data segment (DS) register addresses the data segment.

- **Stack segment** holds a special kind of data structure called a "stack" that acts as a temporary depository for data and addresses. The program needs to save by your own called subroutines. The Stack segment (SS) register addresses the stack segment.
- **Extra segment** is like a spare data segment. It is primarily used in string operations. The Extra Segment (ES) register addresses the stack segment.

Segment Boundaries

A segment register is 16-bit in size and contains the starting address of a segment. The figure below presents and graphic view of the SS, DS, and CS registers and their relationship. As mentioned earlier, a segment begins on a paragraph boundary, which is an address evenly divisible by decimal 16 or Hex 10. Consider a data segment that begins at memory location 038E0H. Because in this and all other cases the rightmost hex digit is zero, the computer designers decided that it would be unnecessary to store the zero digit in the segment register. Thus 038E0H is stored in the register as 038E, with the right most four bits shifted off.

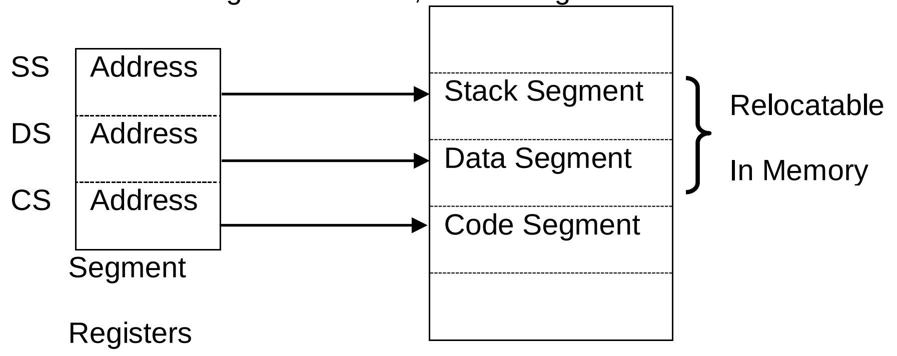


Figure 2.1 Segment and Registers

Memory Addressing

Once the x86 processor has obtained the segment number and offset, it combines them to form one large 20-bit address. It does this by adding the 16-bit offset to the contents of a segment register multiplied by 16. That is:

Physical Address = Offset + (16 x Segment Register)

In reality, the x86 processor doesn't actually multiply the segment register contents by 16, but instead uses the register as if it had four extra zero bits at the end. Adding zeros to the end is the same as multiplying; however, because each time you displace a binary number one bit position to the left, you double its value. Thus, displacing the segment register value four bit positions to the left "multiplied" it by 16, since 2x2x2x2 = 16.
One register contains the segment and another register contains the offset. If you put the two registers together you get a 20-bit address.
Example:

DS segment contents: multiplied by 16	038E[0] $\longrightarrow$	Segment register
Offset:	<u> 0032</u>	
Actual address:	03912H	

Internal Register

The processor's registers are used to control instruction being executed, to handle addressing of memory, and to provide arithmetic capability. A program references the registers by name, such as CS, DS, and SS. Bits in register (like bits in a byte) are conventionally numbered from right to left. Beginning with 0, as …15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0.

Segment Registers

A segment register provides for addressing an area of memory known as the current segment. Intel processor used by the PC series provides different addressing capabilities.

8086/8088 Addressing, the segment registers of these processors are 16 bits in length and operate in real mode. As earlier discussed, as segment address is stored in the segment register, and t he processor shifts off the rightmost four bits 0-bit, so the hex nnnn() becomes nnnn. The processor assumes the rightmost four bits, so effectively, a segment register is 20 bits. A maximum value of FFFF[0]H allows addressing up to 1,048,560 bytes.

80286 Addressing, In real mode, the 80286 processor handles addressing the same as 8086 does. In protected mode, the 24 bit addressing scheme provides for addresses up to FFFFFF[0], or 16 million bytes. The segment registers act as a selector for accessing 24-bit segment address from the memory. Protected mode allows the processor to do multi-tasking.

80386/486/Pentium addressing, in real mode, this processor also handles addressing much the same as 8086 does. In protected mode, the processor use up to 48-bit for addressing, this allows 4 billion bytes. The 16 bit segment registers act as a selector for accessing a 32-bit segment address.

CS Registers – Contains the starting address of a program code segment. This segment address, plus ad offset value in the Instruction Pointer (IP) register, indicates the address of an instruction to be fetched for execution.
DS Register – Contains the starting address of a program's data segment. Instructions use this address to locate a data; this address, plus and offset value in an instruction, causes a reference to a specific byte location in the data segment.
SS Register – Permits implementation of a stack in memory, which a program uses for temporary storage of address and data. The system stores the starting address

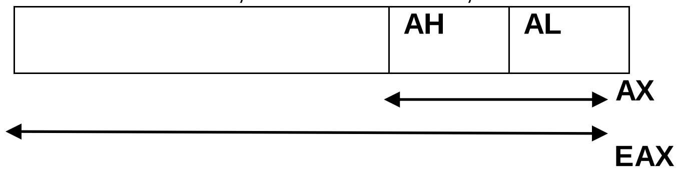
of a program's stack segment in the SS register. This segment address, plus the offset value in the Stack Pointer (SP) register, indicates the current word in the stack being addressed.
ES Register – Used by some string (character data) operations to handle memory addressing. In this context, the ES (Extra Segment) register is associated with the DI (Index) register. If the program requires the use of the ES, you must initialize it with an appropriate segment address.
FS and GS – Additional extra segment registers introduced on 80386 for handling storage requirements.
Pointer Register
Instruction pointer (IP) register, this 16-bit register contains the offset address of the next instruction that is to execute. IP is associated with CS register (CS:IP) in that IP indicates the current instruction within the currently executing code segment. The 80386 introduced the 32-bit IP called EIP.
Stack Pointer (SP) register – The 16-bit SP register provides an offset value which, when associated with SS register (SS:SP), refers to the current word being processed in the stack. The 80386 introduced an extended 32-bit stack pointer, the ESP register.
Base Pointer (BP) register – The 16-bit BP facilitates referencing parameters, which are data and addresses that a program passes via the stack. The processor combines the addresses SS with the offset in BP. BP can also combine with DI and with SI as a base register for special addressing. The 80386 introduced the 32-bit BP, the EBP register.

General Purpose Registers

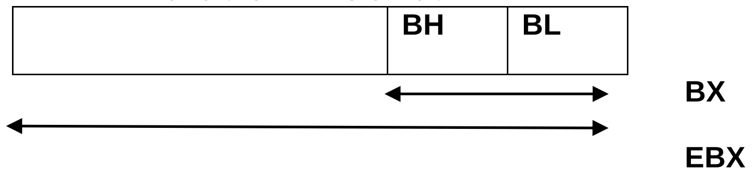
All General Purpose Registers are available for general programming use, but certain instructions also use them implicitly. The assembly code below illustrates moving zeros to AX,BH, and ECX registers respectively.

```
MOV AX, 00

MOV BH, 00

MOV ECX, 00
```

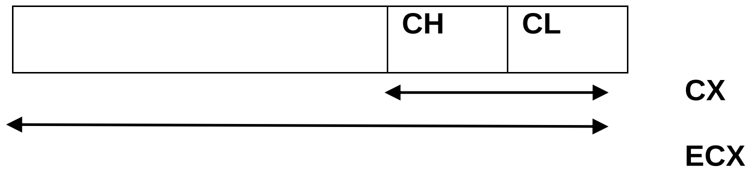
AX register. AX, the primary accumulator, is used for operations involving input/output and most arithmetic. For example, the multiply, divide, and translate instructions assume the use of AX. Also, some instruction generates more efficient machine code if they reference AX rather than another register. The AH and AL are 8 bits, the AX is 16 bits, and the EAX is 32-bit.



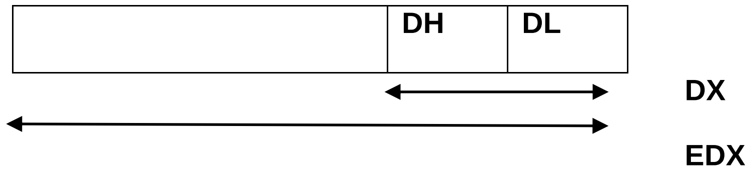
BX register. BX is known as the base register since it is the only general-purpose register that can be used as an index register to extend addressing. Another common purpose of BX is for computations. BX can also be combined with DI or SI as base register for special addressing. The BH and BL are 8 bits, the BX is 16 bits, and the EBX is 32-bit.



CX register. CX is known as the count register. It may contain a value to control the number of times a loop is repeated or a value to shift bits left or right. You may also, use the loop for computations. The CH and CL are 8 bits, the CX is 16 bits, and the ECX is 32-bit.



DX register. DX is known as the data register. Some input/output operations require its use, and multiply and divide operations that involve large values assume the use of DX and AX together as a pair. The DH and DL are 8 bits, the DX is 16 bits, and the EDX is 32-bit.



2.4.4 Index Register

These registers are available for indexed addressing and for some addition and subtraction

SI register – The 16-bit source index register is required for some string (character) handling operations. In this context, SI is associated with the DS register. The 80386 introduced the 32 bit extended register, ESI.

DI Register. The 16-bit destination index register is also required for some string operations. In this context, DI is associated with the ES register. The 80386 introduced the 32 bit extended register, EDI.

Flag Registers

Many instructions involving comparisons and arithmetic change the status of the flags, which some instruction may test to determine subsequent action. The following briefly describes the flags

Bit 0, the Carry Flag (CF), is 1 if an addition produces a carry or a subtraction produces a borrow; otherwise, it is 0. CF also holds the value of a bit that has been shifted or rotated out of a register or memory location, and reflects the result of a compare operation. Finally, CF also acts as a result indicator for multiplications. See the description of bit 11 (OF) for details

Bit 2, the Parity Flag (PF), is 1 if the result of an operation has an even number of 1 bits; otherwise, it is 0. PF is used primarily in data communications.

Bit 4, the Auxiliary Carry Flag (AF), is similar to the CF bit, except AF reflects the presence of a carry or borrow out of bit 3. CF is useful for operating on "packed" decimal numbers.

Bit 6, the Zero Flag (ZF), is 1 if the result of an operation is zero; a nonzero result clears ZF to 0.

Bit 7, the Sign Flag (SF), is meaningful only during operations on signed numbers. SF is 1 if an arithmetic, logical, shift, or rotate operation produces a negative result; otherwise, it is 0. In other words, SF reflects the most-significant (sign) bit of the result, regardless of whether the result is 8 or 16 bits long.

Bit 8, the Trap Flag (TF), makes the x86 Processor "single-step" through a program for debugging purposes.

Bit 9, the Interrupt Enable Flag (IF), allows the x86 Processor to recognize interrupt requests from external devices in the system. Clearing IF to 0 makes the x86 Processor ignore interrupt requests (until IF becomes 1).

Bit 10, the Direction Flag (DF), makes the x86 Processor decrement (DF=1) or increment (DF = 0) the index register(s) after executing a string instruction. If DF is 0, the x86 Processor progresses forward through a string (toward higher addresses, or "left to right"). If DF is 1, it progresses backward (toward lower addresses, or "right to left").

Bit 11, the Overflow Flag (OF), is primarily an error indicator during operations on signed numbers. OF is 1 if adding two like-signed numbers or subtracting two opposite-signed numbers produces a result that the operand can't hold; otherwise, it is 0. OF is also 1 if the most-significant (sign) bit of the operand changed during an arithmetic shift operation; otherwise, it is 0.

The OF flag, in combination with the CF flag, also indicates the length of a multiplication result. If the upper half of the product is nonzero, OF and CF are 1; otherwise, both flags are 0.

Flag:

→					OF	DF	IF	TF	SF	ZF		AF		PF		CF
→	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Summary of 16-bit registers

General Purpose Registers	AX	AH	AL	Accumulator
	BX	BH	BL	Base
	CX	CH	CL	Count
	DX	DH	DL	Data
Pointer and Index Registers	SP			Stack Pointer
	BP			Base Pointer
	SI			Source Index
	DI			Destination Index
Segment Registers	CS			Code Segment
	DS			Data Segment
	SS			Stack Segment
	ES			Extra Segment
Instruction Pointer	IP			Instruction Pointer
Status Flag				

Lesson 3: Assembly Language Features

Learning Outcomes

- LO 2.3.1 Distinguish between high-level and low level language.
- LO 2.3.1 Explore the basic features of assembly language programs (Program comments, Reserved words, identifiers, statements and directives).

Overview

Assembly language is a set of words that tell the computer what to do. However, the words in the assembly-language instruction set refer to computer components directly. It's like the difference between telling someone to walk to the mailbox and telling them precisely how to move their muscles and maneuver past obstacles. Obviously, a simple command is sufficient most of the time; only athletes or mountain climbers need the more detailed instructions.

You write assembly language program according to strict rules, use an editor or word processor for keying it into the computer as a file, and then use the assembler translator program to read the file and to convert it into machine code. Assembly-language programs give the computer detailed commands, such as "load 32 into the AX register," "transfer the contents of the CL register into the DL register," and "store the number in the DL register into memory location 3,456." Precise form is essential to achieving maximum performance.

Because assembly language requires you to operate on the computer's internal components, you must understand the features and capabilities of the integrated circuit (or "chip") that holds these components, the computer's microprocessor.

Programming languages are classified into two classes the high-level and low level. Programmers writing in high level language, such as C or BASIC use powerful commands, each of which may generate machine language instructions. Programmers writing in a low level assembly language, on the other hand, code symbolic instructions, each of which generates one machine instruction. Despite the fact that coding in a high-level language is more productive, some advantages to coding assembly language are that in general:

- Provides more control over handling particular hardware requirements
- Generates smaller, more compact executables
- Results in faster execution

A common practice is to combine the benefits of both programming levels: Code the bulk of a project in a high-level language, and code critical modules (those that cause noticeable delays) in assembly language.

Assembly Language Features

The source program you enter into the computer is a sequence of statements designed to perform a specific task. A source statement (a line in the program) can be either be a program comment, reserved word, identifier, statements, and directive.

Program Comments

The use of comments throughout a program can improve its clarity, especially in assembly language, where the purpose of a set of instructions is often unclear. For example, it is obvious that the instruction MOV AH, 10H to AH, but the reason for doing this may be unclear. A comment begins with a semicolon (;) and where you code it, the assembler assumes that all characters on the line to its right are comments. A comment may contain printable characters including a blank. Example:

```
ADD AX, BX ; Accumulate total quantity
```

Because a comment appears only on a listing of an assembled source program and generates no machine code, you may include any number of comments without affecting the assembled to program’s size execution.

Reserved Words

Certain names in assembly language are reserved for their own purposes, to be used only under special condition. Reserved words, by category include:

- ***Instructions**,* such as MOV and ADD, which operations that the computer can execute.

- ***Directives,*** such as END or SEGMENT, which you use to provide information to the assembler.
- ***Operators,*** such as FAR and SIZE, which you use in expression
- ***Predefined symbols,*** such as @Data and @Model, which return information to your program during assembly

Identifiers

An identifier (or symbol) is a name that you apply to an item in your program that you expect to reference. The two types of identifiers are name and label

1. ***Name*** refers to the address of data item, such as counter in COUNTER in
   ```
   COUNTER DB 0
   ```

2. ***Label*** refers to the address of an instruction, procedure, or segment, such as MAIN and B30: in the following statements
   ```
   MAIN PROC FAR

   B30: ADD BL, 25
   ```

The same rules apply to both names and labels. An identifier can use the following characters:

- Alphabet Letters: A through Z and a through z.
- Digits: 0 through 9 (not the first character)
- Special character: Question mark (?), break, or underline (_), dollar ($), at (@) period or dot (.). (not the first character)

The first character of an identifier must be an alphabetic letter or a special character, except for the dot. Because the assembler uses some special words that begin with the @ symbol, you should avoid using it for your own definitions. Reserved words cannot be used as identifiers.

Statements

An assembly program consists of a set of statements. The two types of statements are:

1. ***Instruction***, such as MOV and ADD, which the assembler translates to object code
2. ***Directive,*** which tell the assembler to perform a specific action, such as define a data item.

Below is the format for a statement, where square brackets indicate optional entries:

[identifier]	Operation	[operand(s)]	[;comment]

An identifier (if any), operation, and operand (if any) are separated by at least one blank or tab character. Most programmers prefer to stay within 80 characters because that is the maximum number of most screens can accommodate. Two examples of statements are the following:

	Identifier	Operation	Operand	Comment
Directive	`COUNT`	`DB`	`1`	`; Name, operation, operand`

Instruction `L30:        MOV        AX,0        ;   Label,   operation,   2 operands`

As defined earlier "Identifiers", are the name applied to the name of a defined item or directive, whereas the term label applies to the name of an instruction.

The operation, which must be coded, is most commonly used for defining data areas and coding instructions. For a data item, an operation such as DB or DW defines a field, work area, or constant. For an instruction, an operation such as MOV and ADD indicates an action to perform.

The operand (if any) provides information for the operation to act on. For a data item, the operand defines its initial value. For example, in the following definition of a data item named COUNTER, the operation DB means "define byte" and the operand initializes its contents with zero value.

Name	Operation	Operand	Comment
COUNTER	DB	0	; Define byte with initial value

For an instruction, an operand indicates where to perform the action. An instruction's operand may contain one, two, or even no entries. Example

Operation	Operand	Comment
RET		; Return from procedure
INC	BX	; Increment BX register by 1
ADD	CX, 25	; Add 25 to CX register

Directives

Assembly language supports a number of statements that enable you to control the way in which a source program assembles and lists. These statements, called directives, act only during the assembly of a program and generate no machine-executable code.

The Page and Title Listing Directive: the Page and Title directive help to control the format of a listing and an assembled program. This is their only purpose, and they have no effect on subsequent execution of the program. At the start of a program, the PAGE directive designates the maximum number of characters on a line. Its format is

```
PAGE [length][,width]
```

Under a typical assembler, the number of lines per page may range from 10 through 255, and the number of characters per line may range from 60 through 132. Omission of a PAGE statement causes the assembler to default to PAGE 50, 80.
You can use the TITLE directive to cause a title for a program to print on line 2 of each page of the program listing. You may code TITLE once, at the start of this program. The format is

```
TITLE text comment
```

For text, a common practice is to use the name of the program as cataloged on disk.
SEGMENT Directive: In the previous chapter it is indicated that in .EXE consists of one or more segments. In real mode, the Stack segment is used for storage, Data segment defines data items and Code segment contains the instruction that would be executed. The directives for defining in a segment, SEGMENT and ENDS, have the following format:

Name	**Operation**	**Operand**
Segment-name	SEGMENT	[align][combine][‘class’]
Segment-name	ENDS	

The segment statement defines the start of a segment. The segment-name must be present, must be unique, and must follow assembly language naming convention. The ENDS statement indicates the end of the segment and contains the same name as the SEGMENT statements. The maximum size of a segment in real mode is 64K. The operand of a segment statement may contain three types of options: alignment, combine, and class.

- The align option indicates the boundary on which the segment is to begin. The typical requirement is PARA, which causes the segment to align on a paragraph boundary so that the starting address is evenly divisible by 16 or 10H. Omission of the align operand causes the assembler to default to PARA.
- The combine option indicates whether to combine the segment with other segments when they are linked after assembly
- You may use PUBLIC and COMMON where you intend to combine separately assembled programs when linking them. Otherwise, where a program is not to be combined with other programs, you may omit this option or code NONE
- The class option, enclosed in apostrophes, is used to group related segments when linking.

The program below illustrates the SEGMENT statements with various operations.

ASSUME Directive: An .EXE program uses the SS register to address the stack, DS to address the data segment, and CS to address the code segment. To this end, you have to tell the assembler the purpose of each segment in the program. The required directive is ASSUME, coded in the code segment as follows:

```
ASSUME      SS: stackname, DS:datasegname, CS:codesegname
```

SS:stackname means that the assembler is to associate the stack segment with the SS register, and similarly to the data segment and code segment. The operands may appear in any sequence. Assume may contain an entry for the ES register, such as ES:datasegname; ES won’t be used omit the ES.
PROC Directive: The code segment contains the executable code for a program which consists of one or more procedure, defined initially with the PROC directive and ended in ENDP directive. The format is shown below

NAME	OPERATION	OPERAND	COMMENT
Procedure-name	PROC	FAR	;Begin Procedure
Procedure-name	ENDP		;End procedure

The procedure-name must be present, must be unique, and must follow assembly language naming conventions. The operand, FAR in this case, is related to program execution. When you request program execution, the program loader uses this procedure as the entry point for the first instruction to execute.

The ENDP directive indicates the end of a procedure and contains the same name as the PROC statement to enable the assembler to relate the end to the start. Because procedure must be fully contained within a segment, ENDP defines the end of a procedure before ENDS defines the end of the segment.

END Directive: As indicated before ENDS ends a segment, ENP ends a procedure, definitely END directive ends the entire program and appears as the last statement. The format is shown below.

```
END [procedure-name]
```

The operand may be blank if the program is not to execute. In most programs, the operand contains the name of the first or only PROC designated as FAR, where program execution begins.

Processor Directives: Most assemblers assume that the source program is to run on a basic 8086-level computer. As a result, when you use instructions or features introduced by later processors, you have to notify the assembler by means of a processor directive such as .286, .386, .486, or .586. The directive may appear immediately before the instruction, code segment, or even the start of the source program. But the 8086 emulator that you would be using is purely supporting 8086 instruction set, thus these processor directives are useless.

Skeleton of an .EXE program using Conventional segments

```
PAGE        60,132

TITLE MYASMPROG  Skeleton of an .EXE program

STACK       SEGMENT PARA     STACK 'Stack'

      …
      STACK       ENDS
      ;---------------------------------------------------------------
      DATASEG     SEGMENT PARA 'Data'
      …
      DATASEG     ENDS
      ;---------------------------------------------------------------
      CODESEG     SEGMENT     PARA        'Code'
      MAIN PROC FAR
                  ASSUME SS:STACK, DS:DATASEG, CS:CODESEG
                  MOV AX, DATASEG   ;Set address of data segment
                  MOV DS, AX        ;in DS
                  …
                  MOV AX, 4C00H           ;End processing
```

```
                INT 21H
        MAIN ENDP                           ;End procedure
        CODESEG     ENDS                    ;End segment
                    END MAIN                ;End program
```

The source code above indicates end processing. This instruction is an INT 21H common DOS interrupt that uses a function code in the AH register to specify an action to be performed. The many functions on INT 21H include keyboard input, screen handling, disk I/O, and printer output. The function that concerns us here is 4CH, which INT 21H recognizes as a request to end program execution. You can also use this operation to pass a return code in AL for subsequent testing in a batch file, as shown below.

```
MOV AH,4CH       ;request end processing
MOV AL, retcode  ; Optional return code
INT 21H                ; Call interrupt service
```

The return code for normal completion of a program is usually 0 (zero). You may also recode two MOVs as one statement.

```
MOV AX, 4C00H         ; request normal exit
```

Simplified Segment Directives: The assembler provides some shortcuts in defining data segments. To use them, you have to initialize the memory model before defining any segment. The different models tell the assembler how to use segments, to provide enough space for the object code, and to ensure optimum execution speed. The format (including the leading dot) is

```
.MODEL memory-model
```

The memory model may be Tiny, Small, Medium, Compact, Large, Huge, or Flat. The tiny model is intended for the use of .COM programs, which have their data, code and stack segment in on 64K segment. The flat model defines one area up to 4 gigabytes for both code and data; the program uses 32-bit addressing and runs under Windows in protected mode. The requirements of other models are:

MODEL	**Number Of Code Segments**	**Number of Data Segments**
Small	1<=64K	1<=64K
Medium	Any Number, Any Size	1<=64K
Compact	1<=64K	Any Number, Any Size
Large	Any Number, Any Size	Any Number, Any Size
Huge	Any Number, Any Size	Any Number, Any Size

You may use any of these models for a standalone program (that is, a program that is not linked to another program). The small model is suitable for most of the examples when using 8086 Emulator, the assembler assumes that addresses are near (within 64K) and generates a 16-bit offset. The .MODEL directive automatically generates the ASSUME statement for all models. The format (including the leading dot) for the directives that define the stack, data, and code segments are:

```
.STACK [size]

.DATA
```

```
.CODE [segment-name]
```

Each of these directives causes the assembler to generate the required SEGMENT statement and it's matching ENDS. The default segment names (which you don't have to define) are _STACK, _DATA, and _TEXT. The default stack size is 1,024 bytes, which you may override, coding also in this format, you can also override the segment-name for the code segment. You use these directives to identify where in the program the three segments are to be located. However you are still to initialize the addresses of the data segment in DS.

```
MOV AX, @data          ; Initialize DS with

MOV DS, AX       ; address of data segment
```

Skeleton of an .EXE program using Simplified segments

```
PAGE        60,132
TITLE MYASMPROG   Skeleton of an .EXE program
;----------------------------------------------------------------
.MODEL SMALL
.STACK      64
.DATA
…
;----------------------------------------------------------------
.CODE
MAIN PROC FAR
            MOV AX,@data            ;Set address of data segment
            MOV DS, AX        ;in DS
            …
            MOV AX, 4C00H           ;End processing
            INT 21H
MAIN ENDP                           ;End procedure
            END MAIN                      ;End program
```

Lesson 4: Defining Data Types Equate Directives

Lesson Outcomes

- LO 2.4.1 Apply these features on constructing the basic framework of an assembly source code on a particular memory model.
- LO 2.4.2 Explain the basic data types in assembly, and trace each of these data types on its specific data segment (memory) location.

The data segment in an .EXE program contains constants, work areas and input/output areas. The assembler provides a set of directives that permit definitions of items by various types and lengths; for example, DB defines byte and DW defines word. A data item may contain an undefined (that is, uninitialized) value, or it may contain an initialized constant, defined either as a character string or as an numeric value.

[name]	Dn	Expression

Name: A program that references a data item does so by means of a name. The name is otherwise optional, as indicated by square brackets.

Directive (DN): The directive that defines data item are DB (byte), DW (word), DD (Doubleword), DF(farword), DQ(Quadword), and DT (tenbytes), each of which explicitly indicates the length of the defined item.

Expression: The expression in an operand may specify an uninitialized value or a constant value. To indicate an uninitialized item, define the operand with a question mark, such as

```
DATAX DB ?  ;Uninitialized item
```

In this case, when your program begins execution, the initial value of DATAX is unknown to you. The normal practice before using this item is to move some value into it, but it must fit the defined size. You can use the operand to define a constant, such as

```
DATAY DB 25 ; Initialized item
```

An expression may contain multiple constants separated by commas and limited only by the length of the line:

```
DATAZ DB 21, 22, 23, 24, 25, 26,
```

The assembler defines these constants in adjacent bytes, from left to right. A reference to DATAZ is to the first byte constant, 21 (you could think of the first byte as DATAZ+0), and a reference to DATAZ+1 is to the second constant, 22. For example

```
MOV AL, DATAZ+2
```

Would move decimal number 23 to AL register. The expression also permits duplication of constants in a statement of the format

[name]	Dn	Repeat-count DUP (expression)

The following examples illustrate duplication:

```
DW 10 DUP(?)     ; Ten words, uninitialized

DB 5 DUP (12)    ; Five bytes containing hex 0C0C0C0C0C

DB 3 DUP (5 DUP(4))   ; Fifteen 4s
```

An expression may define and initialize a character and numeric constant

Character String

Character Strings are used for descriptive data such as people's names and product descriptions. The string is defined within single quotes, such as 'PC', or within double quotes, such as "PC". The assembler stores the contents of the quotes as object code in normal ASCII format, without apostrophes. If a string must contain a single or double quote, you can define it in one of these ways

```
  DB "Crazy Sam's CD Emporium"     ; Double quotes for string,

                                   Single quote for apostrophe

  DB 'Crazy Sam's CD Emporium'     ;Single  quote  for  string,
two

                                   Single quote for apostrophe
```

3.1.1 Numeric Constant

Numeric constants are used to define arithmetic values and memory addresses. The constant is not defined within quotes, but is followed by optional radix specifier, such as H in hexadecimal value 12H. For most of the data definition directives, the assembler stores the generated bytes in object code in reverse sequence, from left to right. Below are various numeric formats

Binary: Binary format uses the binary digits 0 and 1, followed by the radix specifier B. A common use for binary format is to distinguish values for the bit-handling instructions AND, OR, XOR, TEST.

Decimal: Decimal format uses the decimal digits 0-9, optionally followed by the radix specifier D, such as 125 or 125D. Although the assembler allows you to define values in decimal format as a coding convenience, it converts your decimal values to binary object code and represent them in hexadecimal.

Hexadecimal: Hex format uses the hex digit 0 – F, followed radix specifier H. Because the assembler expects that a reference beginning with a letter is a symbolic name, the first digit of a hex constant must be 0 to 9. Examples 3DH and 0DE8H which the assembler stores as 3D and E80D, respectively.

Real: The assembler converts a given real value (a decimal or hex constant followed by the radix specifier R) into floating-point format for use with a numeric coprocessor.

Be sure to distinguish between the use of character and numeric constants. For example, a character constant defined as DB '24' generates ASCII characters, represented in hex 3234. A numeric defined as DB 24 generates a binary number, represented in hex 18.

Directives For Defining Data

The assembled program below provides example of DB, DW, DD and DQ directives to define characters strings and numeric constants. The generated object code is listed on the left. Note that uninitialized values appear as hex zeros, because there is no code segment.

```
                              TITLE DEFINEDATATYPES
                              .MODEL SMALL
                              .DATA
                              ; DB-Define Bytes:
                              ;----------------------------------
0000 00                       BYTE1 DB ?              ;
Uninitialized
0001 30                       BYTE2 DB 48       ; Decimal constant
0002 30                       BYTE3 DB 30H            ;        Hex
constant
0003 7A                       BYTE4 DB 0111101B ; Binary constant
0004 000A [00]                BYTE5 DB 10 DUP (0)     ; Ten zeros
000E 50 43 20 45 6D 70  BYTE6 DB 'PC Emporium'; Character string
     6F 72 69 75 6D
0019 31 32 33 34 25           BYTE7 DB '12345'        ; Number as
chars
001E 01 4A 61 6E 02 46  BYTE8 DB 01, 'Jan', 02, 'Feb', 03, 'Mar'
     65 62 03 4D 61 72                          ; Table of months
                              ; DW-Define Words:
                              ;----------------------------------
```

```
002A FFF0                     WORD1 DW 0FFF0H         ;        Hex
constant
002C 007A                     WORD2 DW 01111010 ; Binary Constant
002E 001E R             WORD3 DW BYTE8          ; Address constant
0030 0002 0004 0006 0007      WORD4 DW 2,4,6,7,9      ; Table of 5
constant
     0009
003A 0008 [0000]              WORD5 DW 8 DUP(0) ; Eight zeros

                              ; DD-Define Doublewords:
                              ;----------------------------------
004A 00000000                 DWORD1 DD ?       ; Uninitialized
004E 00000A25A                DWORD2 DD 41562         ;    Decimal
value
0052 00000018 00000030  DWORD3 DD 24, 48        ; Two constants
005A 00000001                 DWORD4 DD BYTE3-BYTE2   ; Difference
                                                      ;        bet
addresses
                              ; DQ-Define Quadwords:
                              ;----------------------------------
005E 0000000000000000         QWORD1 DQ 0       ; Zero constant
0066 0000000000005E39         QWORD2 DQ 05E39H        ;        Hex
constant
006E 000000000000A25A         QWORD3 DQ 41562         ;    Decimal
constant
                              END
```

DB Define Byte: A DB numeric expression may define one or more 1-byte constants, each consisting of two hex digits. For unsigned numeric data the range of values is 0 to 255; for signed data, the range of values is -128 to +127. The assembler converts numeric constants to binary object code (represented in hex). The DEFINEDATATYPES sample code, uses "?" to specify an uninitialized value. Numeric DB constants are BYTE2, BYTE3, BYTE4 and BYTE5. For example, the assembler has converted the defined value 48 to hex 30.

A DB character expression may contain a string of any length up to the end of the line, for examples BYTE6 and BYTE7 in the code. The hex object code shows the ASCII character of each byte in normal left-to-right sequence, where 20H represents a blank character. BYTE8 shows a mixture of numeric and string constants suitable for defining a table.

DW Define Word: The DW directive items that are one word (two bytes in length). A DW numeric expression may define one or more one word constants. For unsigned numeric data, the range of values is 0 to 65,535; for signed data items, the range of values is -32,768 to + 32, 767. The assembler converts DW numeric constant to binary object code (represented in hex), but stores the bytes in reverse byte sequence. Consequently, a decimal value defined as 12345 converts to hex 3039, but stored in 3930.

WORD1 and WORD2 defined in DW numeric constants. WORD3 defines the operand as an address-in this case, the offset address of BYTE8. The generated object code is 00IE (the R to the right means relocatable), and check of the figure shows that the offset address of BYTE8 (the leftmost column) is indeed 001E. WORD4 defines a table of five numeric constants each with 2 bytes in length. WORD5 defines a table initialize with eight zeros.

DD Define Doubleword: The DD directive defines items that are a doubleword (four bytes) in length. A DD numeric expression may define on or more constants, each with a maximum of four bytes (eight hex digits). For unsigned numeric data, the range of values is 0 to 4,294,967,295; for signed data, the range of values is -2,147,483,648 to +2,147,483,647. The assembled data is also converted into reverse byte order.

The DWORD2 defines a DD numeric constant, DWORD3 defines two numeric constants. DWORD4 generates the numeric difference between two defined address, in this case the result is one byte.

DQ Define Quadword: The DQ directive defines items that are four words (eight bytes) in length. A DQ numeric expression may define one or more constants, each with a maximum of eight bytes, or 16 hex digits. The largest positive quadword hex number is 7 followed by 15Fs.

The assembler handles DQ numeric values and character strings just as it does DD and DW numeric values. QWORD1, QWORD2, QWORD3 illustrate numeric values.

Equate Directives

The assembler provides Equal-Sign, and EQU directives for redefining symbolic names with other names and numeric values with names. These directives do not generate any data storage. Instead, the assembler uses the defined value to substitute in other statements.

The Equal Sign-Directive: The Equal-Sign directive enables you to assign the value of an expression to a name, and may do so any number of times in a program. The following examples illustrate its use:

```
VALUE_OF_PI  =  3.1416
RIGHT_COL = 79
SCREEN_POSITIONS  = 80 *25
```

Examples of the use of the preceding directives are:

```
IMUL AX, VALUE_OF_PI
CMP BL,RIGHT_COL
MOV CX,SCREEN_POSITION
```

When using this directive for defining a doubleword value, first use the .386 directive to notify the assembler:

```
.386

DBLWORD1 = 42A3B05CH
```

The EQU Directive: Consider the following EQU statement coded in the data segment:

```
FACTOR EQU 12
```

The name, in this case FACTOR, may be any name acceptable to the assembler. Now, whenever the word FACTOR appears in an instruction or another directive, the assembler substitute the value 12. For example, the assembler converts the directive

```
TABLEX DB FACTOR DUP(?)
```

To its equivalent value

```
TABLEX DB 12 DUP(?)
```

An instruction may contain an equated operand, as in the following:

```
MOV CX, FACTOR
```

You may also equate symbolic names, as in the following code:

```
AT    EQU   FACTOR
MPY   EQU  MUL
```

The first EQU equates the AT to the defined item FACTOR. For any instruction that contains the operand AT, the assembler replaces it with the address of FACTOR. The second EQU enables a program to use the word MPY in place of the regular symbolic instruction MUL.