UNIT – 1: *Assembly Language Terms & Directives*

Introduction:

A microprocessor is the chip containing some control and logic circuits that is capable of a making arithmetic and logical decision based on input data and produces the corresponding arithmetic or logical output. The word 'processor' is the derivative of the word 'process' that means to carry out systematic operations on data. The computer we are using to write this page of the manuscript uses a microprocessor to do its work. The microprocessor is the heart of any computer, whether it is a desktop machine, a server or a laptop. The microprocessor we are using might be a Pentium, a K6, a PowerPC or any of the many other brands and types of microprocessors, but they all do approximately the same thing in approximately the same way. No logically enabled device can do anything without it. The microprocessor not only forms the very basis of computers, but also many other devices such as cell phones, satellites, and many other hand held devices. They are also present in modern day cars in the form of microcontrollers.

Microprocessor Evolution and types:

A common way of categorizing microprocessors is by the number of bits that their ALU can work with at a time. In other words, a microprocessor with a 4-bit ALU will be referred to as a 4-bit microprocessor, regardless of the number of address lines or the number of data bus lines that it has. The first commercially available microprocessor was the Intel 4004, produced in 1971. It contained 2300 PMOS transistors. The 4004 was a 4-bit device intended to be used with some other devices in making a calculator. Some logic designers, however, saw that this device could be used to replace PC boards full of combinational and sequential logic devices. Also, the ability to change the function of a system by just changing the programming, rather than redesigning the hardware, is very appealing. It was these factors that pushed the evolution of microprocessors.

In 1972 Intel came out with the 8008, which was capable of working with 8-bit words. The 8008, however, required 20 or more, additional devices to form a functional CPU. In 1974, Intel announced the 8080, which had a much larger instruction set than the 8008 and required only two additional devices to form a functional CPU. Also, the 8080 used NMOS transistors, so it operated much faster than the 8008. The 8080 is referred to as a second generation microprocessor.

Soon after Intel produced the 8080, Motorola came out with the MC6800, another 8-bit general-purpose CPU. The 6800 had the advantage that it required only a +5 – V supply rather than the -5-V, +5-V, and +12-V supplies required by the 8080. For several years, the 8080 and the 6800 were the top-selling 8-bit microprocessors. Some of their competitors were the MOS Technology 6502, used as the CPU in the Apple II microcomputer, and the Zilog Z80, used as the CPU in the Radio Shack TRS-80 microcomputer.

As designers found more and more applications for microprocessors, they pressured microprocessor manufactures to develop devices with architectures and features optimized for doing certain types of tasks. In response to the expressed needs, microprocessors have evolved in three majors directions during the last 15 years.

The 8086 microprocessor family overview:

The Intel 8086 is a 16-bit microprocessor that is intended to be used as the CPU in a microcomputer. The term 16-bit means that its arithmetic logic unit, its internal registers, and most of its instructions are designed to work with 16-bit binary words. The 8086 has a 16-bit data bus, so it can read data from or write data to memory and ports either 16 bits or 8 bits at a time. The 8086 has a 20-bit address bus, or it can address any one of 2^{20}, or 1,048,576 memory locations.

Each of the 1,048,576 memory addresses of the 8086 represents a byte-wide location. Sixteen-bit words will be stored in two consecutive memory locations. If the first byte of a word is at an even address, the 8086 can read the entire word in one operation. If the first byte of the word is at an old address, the 8086 can read the entire word in one operation.

The Intel 8088 has the same ALU, the same registers, and the same instructions set as the 8086. The 8088 also has a 20-bit address bus, so it can address any one of 1,048,576 bytes in memory. The 8088, however, has an 8-bit data bus, so it can only read data from or write data to memory and ports, 8-bits at a time.

The Intel 80186 is an improved version of the 8086, and the 80188 isan improved version of the 8088. In addition to a 16-bit CPU, the 80186 and 80188 each have programmable peripheral devices integrated in the same package. The instruction set of the 80186 and 80188 is a superset of the instructions set of the 8086.

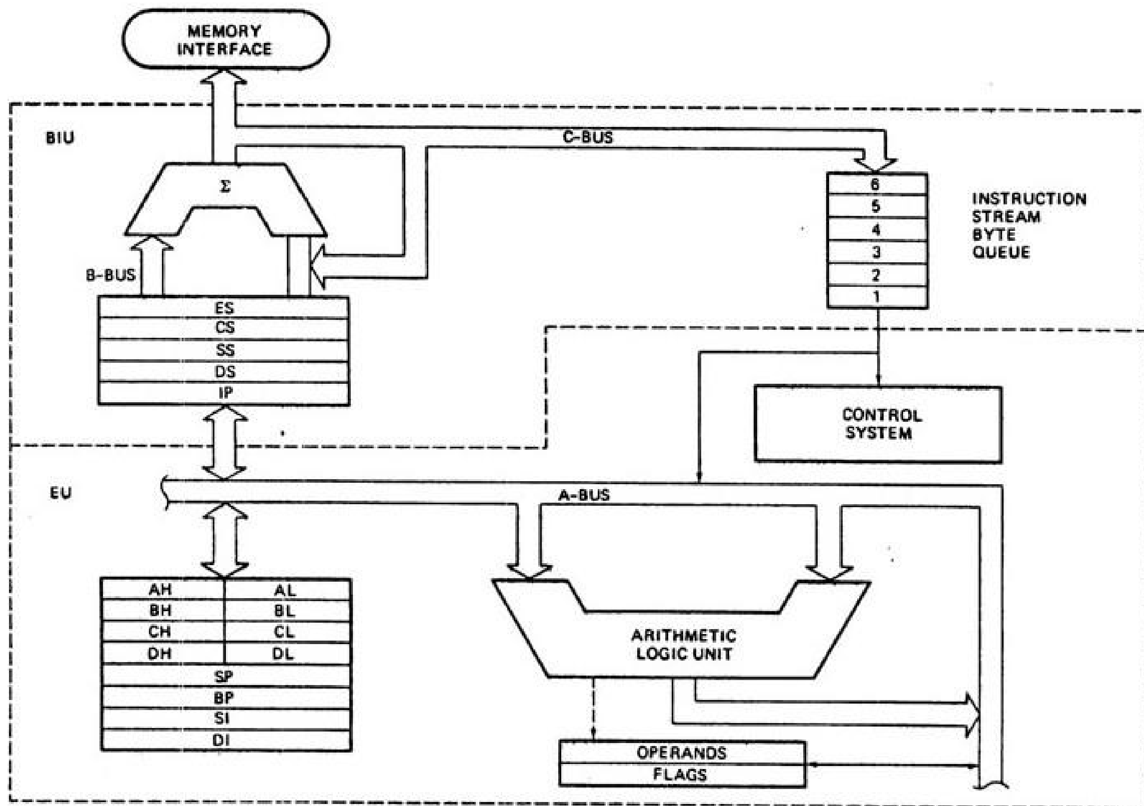
The Intel 80286 is a 16-bit, advanced version of the 8086 which was specifically designed for use as the CPU in a multiuser or multitasking microcomputer. When operating in its real address mode, the 80286 functions mostly as a fast 8086. Most programs written for an 8086 can be run on an 80286 operating in its real address mode.

With the 80386 processor, Intel started the 32-bit processor architecture, known as the IA-32 architecture. This architecture extended all the address and general purpose registers to 32-bits, which gave the processor the capability to handle 32-bit address with 32-bit data, and yet accommodating all the software designed for the earlier 16-bit processors, 8086, 8088, 80186, 80188 and 80286. It contains more sophisticated features for use in multiuser and multitasking environments.

Intel 80486 is the next member of the IA-32 architecture. This processor has the floating point processor integrated into the CPU chip itself. These processors are then followed

by different versions of the Pentium Processors, with different additional capabilities such as multimedia (MMX, SSE, SSE2 etc.), system power saving modes, hyper thread technology etc.

8086 internal architecture : the execution unit, the bus interface unit:



The Execution Unit:

As shown in fig. , the EU contains control circuitary which directs internal operations. A decoder in the EU translates instructions fetched from memory into a series of actions which the EU carries out. The EU has a 16-bit arithmetic logic unit which can add, subtract, AND, OR, XOR, increment, decrement, complement, or shift binary numbers.

FLAG Registers: A flag is a flip-flop that indicates some condition produced by the execution of an instruction or controls certain operations of the EU. A 16-bit flag register in the EU contains nine active flags. Six of the nine flags are used to indicate some condition produced by an instruction. For example, a flip-flop called the carry flag will be set to a 1 if the addition of two 16-bit binary numbers produces a carry out of the most significant bit position. If no carry out of the MSB is produced by the addition, then the carry flag will be a 0. The EU, thus effectively runs up a “flag” to tell you that a carry was produced.

The six conditional flags in this group are the carry flag (CF), the parity flag (PF), the auxiliary carry flag (AF), the zero flag (ZF), the sign flag (SF), and the overflow flag (OF). The names of these flags should give you hints as to what conditions affect them.

The three remaining flags in the flag register are used to control certain operations of the processor. The three control flags are the trap flag (TF), which is used for single stepping through a program; the interrupt flag (IF), which is used to allow or prohibit the interruption of a program; and the direction flag (DF), which is used with string instructions.

General Purpose Registers:

The EU has eight general-purpose registers, labeled AH, AL, BH, BL, CH, CL, DH and DL. These registers can be used individually for temporary storage of 8-bit data. The AL register is also called the accumulator; it has some features that the other general-purpose registers do not have.

Certain pairs of these general-purpose registers can be used together to store 16-bit data words. The acceptable register pairs are AH and AL, BH and BL, CH and CL and DH and CL. The AH-AL pair is referred to as the AX register, the BH-BL pair is referred to as the BX register, the CH-CL pair is referred to as the CX register, and the DH-DL pair is referred to as the DX register.

THE BIU:

The BIU has an instruction stream byte queue, a set of segment registers and instruction pointer.

Instruction Byte Queue

8086 instructions vary from 1 to 6 bytes. Therefore fetch and execution are taking place concurrently in order to improve the performance of the microprocessor. The BIU feeds the instruction stream to the execution unit through a 6 byte prefetch queue. This prefetch queue can be considered as a form of loosely coupled pipelining. Execution and decoding of certain instructions do not require the use of buses. While such instructions are executed, the BIU fetches up to six instruction bytes for the following instructions (the subsequent instructions). The BIU store these prefetched bytes in a first-in-first out register by name instruction byte queue. When the EU is ready for its next instruction, it simply reads the instruction byte(s) for the instruction from the queue in BIU. This process is much faster since it forms a pipeline.

Segment Registers

In 8086, program, data and stack memories occupy the same memory space. As the most of the processor instructions use 16-bit pointers the processor can effectively address only 64 KB of memory. To access memory outside of 64 KB the CPU uses special segment registers to specify where the code, stack and data 64 KB segments are positioned within 1 MB of memory.

Memory can be thought of as a vast collection of bytes. These bytes need to be organized in some efficient manner in order to be of any use. A simple scheme would be to order the bytes in a serial fashion and number them from 0 (or 1) to the end of memory. The numbers thus given to the individual positions in memory are called ADDRESSES. The problem with this approach is that towards the end of memory, the addresses become very large. For example, if a computer has 1 Megabyte of RAM, the highest address would be 1048575 (=1024*1024-1). This definitely would not fit in a 16-bit register and therefore addresses need to be stored in two registers. The scheme used in the 8086 is called segmentation. Every address has two parts, a SEGMENT and an OFFSET. The segment indicates the starting of a 64 kilobyte portion of memory, in multiples of 16. The offset indicates the position within the 64k portion.

Absolute address = (segment * 16) + offset

The memory of 8086 is divided into 4 segments namely code segment (program memory), data segment (data memory), stack memory (stack segment) and extra memory (extra segment).

Program memory – Program can be located anywhere in memory. Jump and call instructions can be used for short jumps within currently selected 64 KB code segment, as well as for far jumps anywhere within 1 MB of memory. All conditional jump instructions can be used to jump within approximately +127 - -127 bytes from current instruction.

Data memory – The processor can access data in any one out of 4 available segments, which limits the size of accessible memory to 256 KB (if all four segments point to different 64 KB blocks).

Stack memory – A stack is a section of the memory set aside to store addresses and data while a subprogram executes. The stack segment register is used to hold the upper 16 bits of the starting address for the program stack.

Extra segment – This segment is also similar to data memory where additional data may be stored and maintained. This area is very often used for string related operations.

Accessing data from the Data, Code, Stack or Extra segments can be usually done by prefixing instructions with the DS:, CS:, SS: or ES: (some registers and instructions by default may use the ES or SS segments instead of DS segment). Word data can be

located at odd or even byte boundaries. The processor uses two memory accesses to read 16-bit word located at odd byte boundaries. Reading word data from even byte boundaries requires only one memory access.

Stack memory can be placed anywhere in memory. The stack can be located at odd memory addresses, but it is not recommended for performance reasons.

Code Segment (CS) register is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

Stack Segment (SS) register is a 16-bit register containing address of 64KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction.

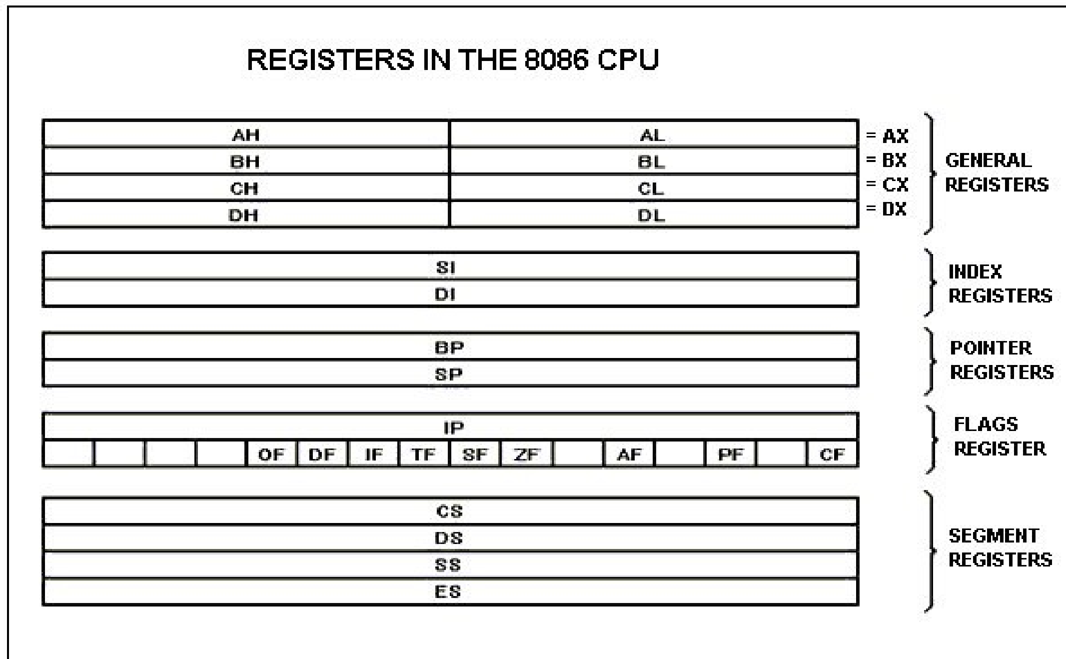
Data Segment (DS) register is a 16-bit register containing address of 64KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment. DS register can be changed directly using POP and LDS instructions.

Extra Segment (ES) register is a 16-bit register containing address of 64KB segment, usually with program data. By default, the processor assumes that the DI register references the ES segment in string manipulation instructions. ES register can be changed directly using POP and LES instructions.

Instruction Pointer

The instruction pointer contains a 16-bit offset which tells where in that 64-Kbyte code segment the next instruction byte is to be fetched from. The actual physical address sent to memory is produced by adding the offset contained in the IP register to the segment base represented by the upper 16 bits in the CS register.

Overview of 8086 register set:



General Purpose Register:

8086 CPU has 8 general purpose registers; each register has its own name:

AX - the accumulator register (divided into **AH / AL**):

1. Generates shortest machine code
2. Arithmetic, logic and data transfer
3. One number must be in AL or AX
4. Multiplication & Division
5. Input & Output

BX - the base address register (divided into **BH / BL**).

CX - the count register (divided into **CH / CL**):

1. Iterative code segments using the LOOP instruction
2. Repetitive operations on strings with the REP command
3. Count (in CL) of bits to shift and rotate

DX - the data register (divided into **DH / DL**):

1. DX:AX concatenated into 32-bit register for some MUL and DIV operations

2. Specifying ports in some IN and OUT operations

SI - source index register:
1. Can be used for pointer addressing of data
2. Used as source in some string processing instructions
3. Offset address relative to DS

DI - destination index register:
1. Can be used for pointer addressing of data
2. Used as destination in some string processing instructions
3. Offset address relative to ES

BP - base pointer:
1. Primarily used to access parameters passed via the stack
2. Offset address relative to SS

SP - stack pointer:
1. Always points to top item on the stack
2. Offset address relative to SS
3. Always points to word (byte at even address)
4. An empty stack will had SP = FFFEh

Segment Registers:

CS - points at the segment containing the current program.

DS - generally points at segment where variables are defined.

ES - extra segment register, it's up to a coder to define its usage.

SS - points at the segment containing the stack.

Although it is possible to store any 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. Segment registers work together with general purpose register to access any memory value.

The address formed with 2 registers is called an **effective address**. By default **BX, SI** and **DI** registers work with **DS** segment register; **BP** and **SP** work with **SS** segment register. Other general purpose registers cannot form an effective address. Also, although **BX** can form an effective address, **BH** and **BL** cannot.

Special Purpose Registers

IP - the instruction pointer:

1. Always points to next instruction to be executed
2. Offset address relative to CS

IP register always works together with **CS** segment register and it points to currently executing instruction.

FLAGS REGISTER

Flags Register - determines the current state of the processor. They are modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program. Generally you cannot access these registers directly.

1. **Carry Flag (CF)** - this flag is set to **1** when there is an **unsigned overflow**. For example when you add bytes **255 + 1** (result is not in range 0...255). When there is no overflow this flag is set to **0**.
2. **Parity Flag (PF)** - this flag is set to **1** when there is even number of one bit in result, and to **0** when there is odd number of one bit.
3. **Auxiliary Flag (AF)** - set to **1** when there is an **unsigned overflow** for low nibble (4 bits).
4. **Zero Flag (ZF)**- set to **1** when result is **zero**. For non-zero result this flag is set to **0**.
5. **Sign Flag (SF)** - set to **1** when result is **negative**. When result is **positive** it is set to **0**. (This flag takes the value of the most significant bit.)
6. **Trap Flag (TF)** - Used for on-chip debugging.
7. **Interrupt enable Flag (IF)** - when this flag is set to **1** CPU reacts to interrupts from external devices.
8. **Direction Flag (DF)** - this flag is used by some instructions to process data chains, when this flag is set to **0** - the processing is done forward, when this flag is set to **1** the processing is done backward.
9. **Overflow Flag (OF)** - set to **1** when there is a **signed overflow**. For example, when you add bytes **100 + 50** (result is not in range -128...127).

The concept of Assembler:

UNIT – 1: *Assembly Language Terms & Directives*

An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. Some people call these instructions assembler language and others use the term *assembly language*.

Here's how it works:

- Most computers come with a specified set of very basic instructions that correspond to the basic machine operations that the computer can perform. For example, a "Load" instruction causes the processor to move a string of bits from a location in the processor's memory to a special holding place called a register.
- The programmer can write a program using a sequence of assembler instructions.
- This sequence of assembler instructions, known as the source code or source program, is then specified to the assembler program when that program is started.
- The assembler program takes each program statement in the source program and generates a corresponding bit stream or pattern (a series of 0's and 1's of a given length).
- The output of the assembler program is called the object code or object program relative to the input source program. The sequence of 0's and 1's that constitute the object program is sometimes called machine code.
- The object program can then be run (or executed) whenever desired.

In the earliest computers, programmers actually wrote programs in machine code, but assembler languages or instruction sets were soon developed to speed up programming. Today, assembler programming is used only where very efficient control over processor operations is needed. It requires knowledge of a particular computer's instruction set, however. Historically, most programs have been written in "higher-level" languages such as COBOL, FORTRAN, PL/I, and C. These languages are easier to learn and faster to write programs with than assembler language. The program that processes the source code written in these languages is called a compiler. Like the assembler, a compiler takes higher-level language statements and reduces them to machine code.

A newer idea in program preparation and portability is the concept of a virtual machine. For example, using the Java programming language, language statements are compiled into a generic form of machine language known as bytecode that can be run by a virtual machine, a kind of theoretical machine that approximates most computer operations. The bytecode can then be sent to any computer platform that has previously downloaded or built in the Java virtual machine. The virtual machine is aware of the specific instruction lengths and other particularities of the platform and ensures that the Java bytecode can run.

Assembly Language Comments:

The use of comments throughout a program can improve clarity, especially in assembly language where the purpose of a set of instructions is often unclear. A comment always begins with a semicolon (;), and wherever r you code it, the assembler assumes that all characters to its right are comments. A comment may contain any printable character, including a blank.

A comment may appear on a line by itself or following an instruction on the same line, as the following two examples illustrate.

; This entire line is a comment

ADD AX, BX ; Comment on same line as instruction

CODING FORMAT

In general, a name refers to the address of a data item, whereas a label refers to the address of an instruction. Since the same rules apply to both names and labels, this section uses the term name to mean either anme or lable. Here is the general format for an instruction.

[name]	Operation	[operand(s)]

A name (if any), operation, and operand (if any) are separated by at least one blank or tab character. There are a maximum of 132 characters on a line, although most people prefer to stay within 80 characters because of the screen width. Name, operation, and operand may begin at any column. However, consistently starting at the same column for these entries makes a more readable program.

Name

A name or label can use the following characters:

Alphabetic letters: A through Z and a through z
Digits: 0 through 9
Special Characters: ?, ., @, _, $

The first character of a name must be an alphabetic letter or a special character. The assembler treats uppercase and lowercase letters the same. The maximum length is 31 characters.

Operation

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 or ADD indicates an action.

Operand

For a data item, an operand defines its initial value. In the following definition of a data item named COUNTER, the operand initializes its contents with 0.

Name	**Operation**	**Operand**	
COUNTER	DB	0	; Dine byte (DB) with 0 value

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

	Operation	**Operand**	**Comment**
No operand	RET		; Return
One operand	INC	CX	; Increment CX Register
Two operand	ADD	AX, 12	; Add 12 to AX Register

Data Definition Directives

The directives that define data items are DB (byte), DW (word), DD(Doubleword), DT (tenbytes), DQ (quadword) each of which indicates the length of the defined item.

Expression: The expression in an operand may contain a question mark to indicate an uninitialized item, such as

FLD1 DB ?

An expression may contain multiple constants separted by commas and limited only by the length of a line, as follows:

FLD3 DB 11, 12, 13, 14, 15…

An expression may contain a character string or a numeric constant.

Character String is used for descriptive data such as people's names and page titles. The string is contained within single quotes as 'PC' or within double quotes as "PC". DB is only format that defines a character string exceeding two characters and stores them in normal left-to-right sequence.

Numeric Constants are used for arithmetic values and for memory addresses. The constant is not stored within quotes. The assembler converts numeric constants to hexadecimal and stores the bytes in object code in reverse sequence – right to left. Following are the various numeric formats.

Decimal format permits the decimal digits 0 through 9 optionally followed by the letter D, as 125 or 125D. Although the assembler allows decimal format as a coding convenience, it converts decimal to binary object code and represents it in hex. For example, decimal 125 becomes hex 7D.

Hexadecimal. Hex format permits the hex digits 0 through F followed by the letter H, which you can use to define binary values.

Binary format permits the binary digits 0 and 1 followed by the letter B. the normal use for binary format is to clearly distinguish bit values for the Boolean instructions AND, OR, XOR, and TEST.

Octal This format permits the octal digits 0 through 7 followed by the letter Q or O, such as 253Q. Octal has specialized uses.

Real: The assembler converts the given real value, a decimal or hex constant followed by the letter R, into floating-point format.

Assembler Directives

ASSUME

The ASSUME directive is used to tell the assembler the name of the logical segment it should use for a specified segment. The statement ASSUME CS: CODE, for example, tells the assembler that the instructions for a program are in a logical segment named CODE. The statement ASSUME DS: DATA tells the assembler that for any program instruction which refers to the data segment, it should use the logical segment called DATA. If, for example, the assembler reads the statement MOV AX, [BX] after it reads this ASSUME, it will know that the memory location referred to by [BX] is in the logical segment DATA.

DB – DEFINE BYTE

The DB directive is used to declare a byte-type variable, or to set aside one or more storage locations of type byte in memory. The statement CURRENT_TEMPERATURE DB 42H, for example, tells the assembler to reserve 1 byte of memory for a variable named CURRENT_TEMPERATURE and to put the value 42H in that memory location when the program is loaded into RAM to be run.

DD – DEFINE DOUBLEWORD

The DD directive is used to declare a variable of type doubleword or to reserve memory locations which can be accessed as type doubleword. The statement ARRAY_POINTER DD 25629261H, for example, will define a doubleword named ARRAY_POINTER and initialize the doubleword with the specified value when the program is loaded into memory to be run.

DT – DEFINE TEN BYTES

DT is used to tell the assembler to define a variable which is 10 bytes in length or to reserve 10 bytes of storage in memory.

DW – DEFINE WORD

The DW directive is used to tell the assembler to define a variable of type word or to reserve storage locations of type word in memory. The statement MULTIPLIER DW 437AH, for example, declares a variable of type word named MULTIPLIER. The statement also tells the assembler that the variable MULTIPLIER should be initialized with the value 437AH when the program is loaded into memory to be run.

END – END PROGRAM

The END directive is put after the last statement of a program to tell the assembler that this is the end of the program module.

ENDP – END PROCEDURE

This directive is used along with the name of the procedure to indicate the end of a procedure to the assembler.

ENDS – END SEGMENT

This directive is used with the name of a segment to indicate the end of that logival segment. ENDS is used with the SEGMENT directive to ‘bracket’ a logical segment containing instructions or data.

LABEL

As the assembler assembles a section of data declarations or instructions statements, it uses a location counter to keep track of how many bytes it is from the start of a segment at any time. The LABEL directive is used to give a name to the current value in the location counter. The LABEL directive must be followed by a term which specifies the type you want associated with that name. If the label is going to be used as the destination for a jump or a call, then the label must be specified as type near or type far.

If the label is going to be used to reference a data item, then the label must be specified as type byte, type word, or type double word.

OFFSET

OFFSET is an operator which tells the assembler to determine the offset or displacement of a named data item (variable) or procedure from the start of the segment which contains it.

PROC – PROCEDURE

The PROC directive is used to identify the start of a procedure. The PROC directive follows a name you give the procedure. After the PROC directive, the term near or the term far is used to specify the type of the procedure.

PTR - POINTER

The PTR operator is used to assign a specific type to a variable or to a label. It is necessary to do this in any instruction where the type of the operand is not clear. The PTR operator can be used to override the declared type of a variable. It is also used to clarify our intentions when we use indirect Jump instructions.

Public

Large programs are usually written as several separate modules. Each module is individually assembled, tested, and debugged. When all the modules are working correctly, their object code files are linked together to form the complete program. In order for the modules to link together correctly, any variable name or label referred to in other modules must be declared **public** in the module in which it is defined.

SEGMENT

The SEGMENT directive is used to indicate the start of a logical segment. Preceding the SEGMENT directive is the name you want to give the segment. The statement CODE SEGMENT, for example, indicates to the assembler the start of a logical segment called CODE. The SEGMENT and ENDS directives are used to ‘bracket’ a logical segment containing code or data.

EQU - EQUATE

EQU is used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it will replace the name with the value or symbol you equated with that name. Suppose, for example, you write the statement CORRECTION_FACTOR EQU 03H at the start of your program, and later in the program you write the instruction statement ADD AL, CORRECTION_FACTOR. When it

codes this instruction statement, the assembler will code it as if you had written the instruction ADD AL, 03H.

EXTRN

The EXTRN directive is used to tell the assembler that the names or labels following the directive are in some other assembly module. For example, if you want to call a procedure which is in a program module assembled at a different time form that which contains the CALL instruction, you must tell the assembler that the procedure is external. The assembler will then put information in the object code file so that the linker can connect the two modules together.

INCLUDE – INCLUDE SOURCE CODE FROM FILE

This directive is used to tell the assembler to insert a block of source code form the named file into the current source module. This shortens the source code. An alternative is to use the editor block commands to copy the file into the current source module.