Internal Architecture of 8086

8086 has two blocks BIU and EU. The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands. The instruction bytes are transferred to the instruction queue. EU executes instructions from the instruction system byte queue. Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism which is called as Pipelining. This results in efficient use of the system bus and system performance. BIU contains Instruction queue, Segment registers, Instruction pointer, and Address adder. EU contains Control circuitry, Instruction decoder, ALU, Pointer, Index register and Flag register.

Bus Interface Unit:

It provides a full 16 bit bidirectional data bus and 20 bit address bus. The bus interface unit is responsible for performing all external bus operations.

Instruction fetches Instruction queuing, Operand fetch and storage, Address relocation and Bus control. The BIU uses a mechanism known as an instruction stream queue to implement pipeline architecture. This queue permits prefetching of up to six bytes of instruction code. Whenever the queue of the BIU is not full, it has room for at least two more bytes and at the same time the EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by prefetching the next sequential instruction. These prefetching instructions are held in its FIFO queue. With its 16 bit data bus, the BIU fetches two instruction bytes in a single memory cycle. After a byte is loaded at the input end of the queue, it automatically shifts up through the FIFO to the empty location nearest the output.

The EU accesses the queue from the output end. It reads one instruction byte after the other from the output of the queue. If the queue is full and the EU is not requesting access to operand in memory, these intervals of no bus activity, which may occur between bus cycles, are known as Idle state. If the BIU is already in the process of fetching an instruction when the EU request it to read or write operands from memory or I/O, the BIU first completes the instruction fetch bus cycle before initiating the operand read / write cycle. The BIU also contains a dedicated adder which is used to generate the 20bit physical address that is output on the address bus. This address is formed by adding an appended 16 bit segment address and a 16 bit offset address. For example**:** The physical address of the next instruction to be fetched is formed by combining the current contents of the code segment CS register and the current contents of the instruction pointer IP register. The BIU is also responsible for generating bus control signals such as those for memory read or write and I/O read or write.

Execution Unit

The Execution unit is responsible for decoding and executing all instructions. The EU extracts instructions from the top of the queue in the BIU, decodes them, generates operands if necessary, passes them to the BIU and requests it to perform the read or write bus cycles to memory or I/O and perform the operation specified by the instruction on the operands. If the queue is empty, the EU waits for the next instruction byte to be fetched and shifted to top of the queue. When the EU executes a branch or jump instruction, it transfers control to a location corresponding to another set of sequential instructions. Whenever this happens, the BIU automatically resets the queue and then begins to fetch instructions from this new location to refill the queue.

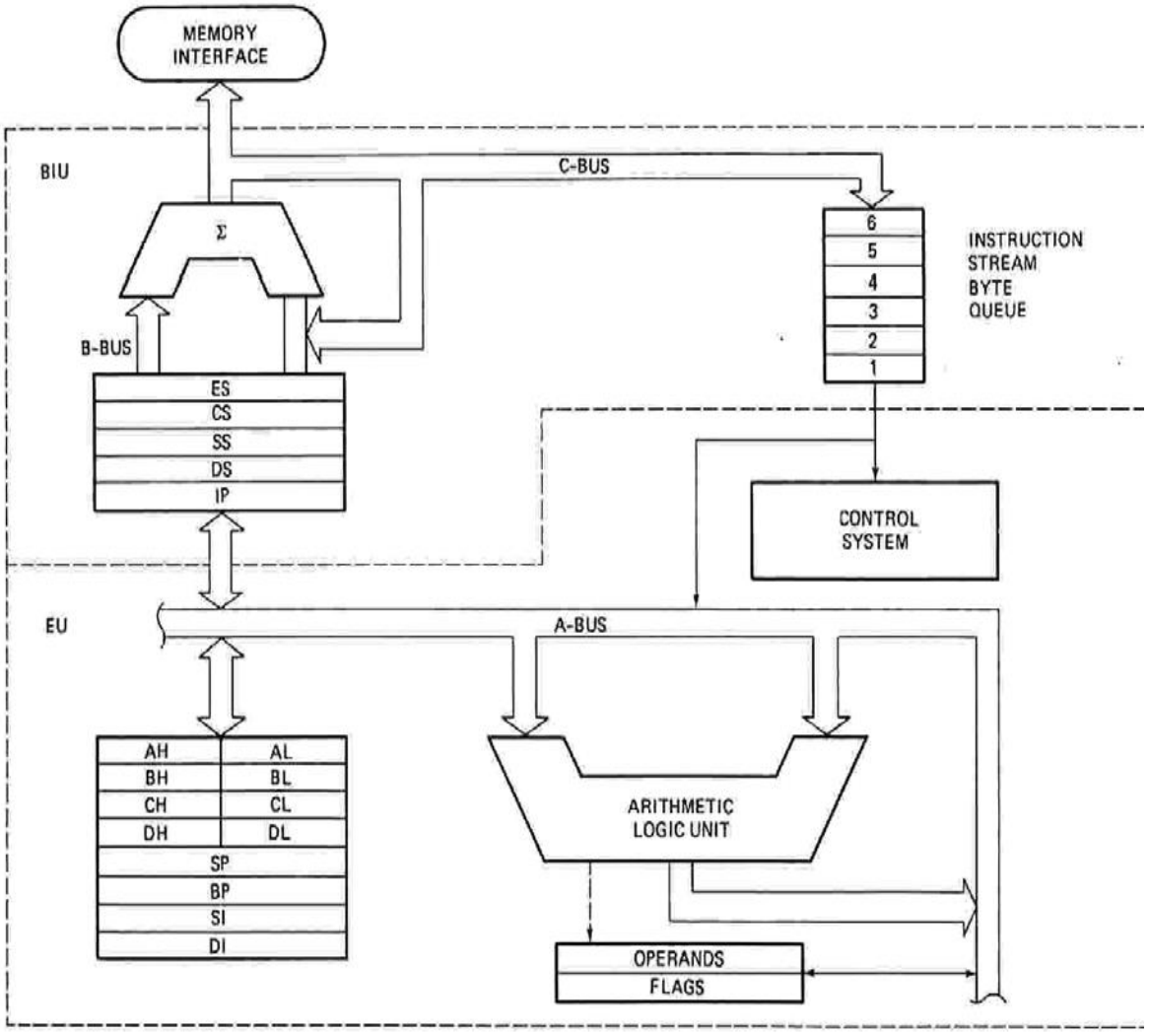

FIGURE 2-7 8086 internal block diagram. (*Intel Corp.*)

The function of BIU can be summarized as:

1. It sends an address of the memory or I/O
2. It fetches instruction from memory
3. It reads data from port / memory
4. It writes data into port / memory
5. It supports instruction queuing
6. It provides the address relocation facility

Register Organisation of 8086

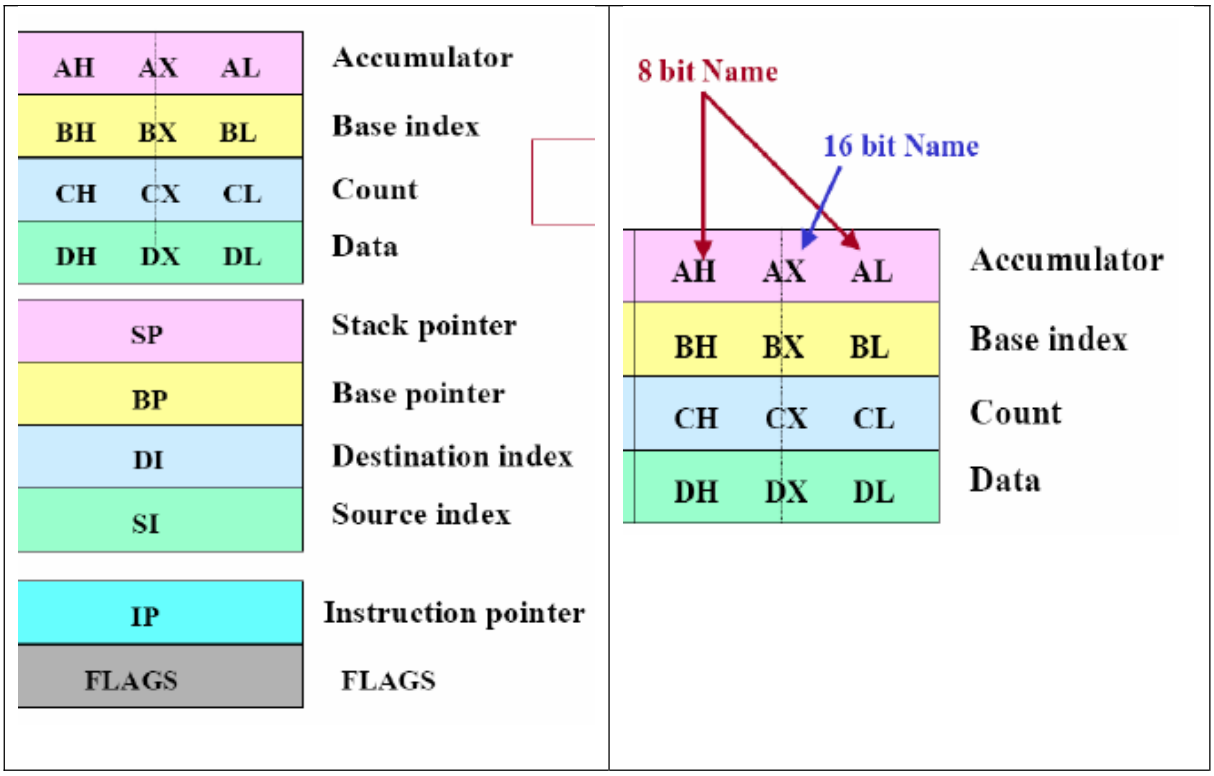

Fig: Register Organization of 8086

The 8086 has four groups of the user accessible internal registers. They are the instruction pointer, four data registers, four pointer and index register, four segment registers The 8086 has a total of fourteen 16-bit registers including a 16 bit register called the status register, with 9 of bits implemented for status and control flags. Most of the registers contain data/instruction offsets within 64 KB memory segment. There are four different 64 KB

segments for instructions, stack, data and extra data. To specify where in 1 MB of processor memory these 4 segments are located the processor uses four segment registers:

Code segment (CS) 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. It contains the starting address of a program's code segment. This segment address plus an offset value in the IP register indicates the address of an instruction to be fetched for execution.

Stack segment (SS) is a 16-bit register containing address of 64KB segment with program stack. It permits the implementation of a stack in memory .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. It stores the starting address of a program's stack segment the SS register. SS register can be changed directly using POP instruction.

Data segment (DS) 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, and DX) and index register (SI, DI) is located in the data segment. It contains the starting address of a program's data segment .Instruction uses this address to locate data. This address plus an offset value in an instruction causes a reference to a specific byte location in the data segment. DS register can be changed directly using POP and LDS instructions.

Extra Segment (ES) is a 16-bit register containing address of 64KB segment, usually with program data. By default, the processor assumes that the DI register points to the ES segment in string manipulation Instructions. ES register can be changed directly using POP and LES instructions.. It is used by some string operations to handle memory addressing.

Accumulator register consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX. AL in this case contains the low order byte of the word, and AH contains the high-order byte. Accumulator can be used for I/O operations and string manipulation.

Base register consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX. BL in this case contains the low-order byte of the word, and BH contains the high-order byte. BX register usually contains a data pointer used for based, based indexed or register indirect addressing.

Count register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit

register CX. When combined, CL register contains the low order byte of the word, and CH contains the high-order byte. Count register can be used in Loop, shift/rotate instructions and as a counter in string manipulation,.

Data register consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX. When combined, DL register contains the low order byte of the word, and DH contains the high-order byte. Data register can be used as a port number in I/O operations. In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

POINTER REGISTERS

The 16 bit Pointer Registers are IP,SP and BP respectively .SP and BP are located in EU whereas IP is located in BIU

Stack Pointer (SP) is a 16-bit register pointing to program stack. The 16 bit SP Register provides an offset value, which when associated with the SS register (SS: SP)

Base Pointer (BP) is a 16-bit register pointing to data in stack segment. The 16 bit BP facilitates referencing parameters, which are data and addresses that a program passes via the stack. The processor combines the addresses in SS with the offset in BP.BP can also be combined with DI and SI as a base register for special addressing. BP register is usually used for based, based indexed or register indirect addressing.

INDEX REGISTERS

The 16 bit Index Registers are SI and DI

Source Index (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data addresses in string manipulation instructions. SI is associated with the DS Register.

Destination Index (DI) is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data addresses in string manipulation instructions. In this context, DI is associated with the ES register.

Other registers:

Instruction Pointer (IP) is a 16-bit register. The 16 bit IP Register contains the offset address of the next instruction that is to execute .IP is associated with CS register as (CS:IP) .For each instruction that executes, the processor changes the offset value in IP so that IP in effect directs each step of execution.

The Queue

While the EU is decoding an instruction or executing an instruction which does not require use of the buses, the BIU fetches up to six instructions bytes for the following instructions. The BIU Stores pre- fetched bytes in First in First out register set called a queue. When the EU is ready for its next instruction, it simply reads the instruction bytes for the instruction from the queue in the BIU. This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction bytes or bytes. Fetching the next instruction while the current instruction executes is called pipelining.

Flag Register:

- A Flag is a flip-flop which 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 9 active flags.
- Figure below show shows the location of the nine flags in the flag register.

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

Figure: 8086 Flag Register Format

U = UNDEFINED

CONDITIONAL FLAGS

CF = CARRY FLAG [Set by Carry out of MSB]
PF = PARITY FLAG [Set if Result has even parity]
AF= AUXILIARY CARRY FLAG FOR BCD
ZF = ZERO FLAG [Set if Result is 0]
SF = SIGN FLAG [MSB of Result]
OF = OVERFLOW FLAG

CONTROL FLAG

TF = SINGLE STEP TRAP FLAG
IF = INTERRUPT ENABLE FLAG
DF = STRING DIRECTION FLAG

Overflow Flag (OF) - set if the result is too large positive number, or is too small negative number to fit into destination operand.

Direction Flag (DF) - if it is set then string manipulation instructions will auto-decrement index registers. If cleared

then the index registers will be auto-incremented.

Interrupt-enable Flag (IF) - setting this bit enables maskable interrupts.

Single-step Flag (TF) - if set then single-step interrupt will occur after the next instruction.

Sign Flag (SF) - set if the most significant bit of the result is set.

Zero Flag (ZF) - set if the result is zero.

Auxiliary carry Flag (AF) - set if there was a carry from or borrow to bits 0-3 in the AL register.

Parity Flag (PF) - set if parity (the number of "1" bits) in the low-order byte of the result is even.

Carry Flag (CF) - set if there was a carry from or borrow to the most significant bit during last result calculation.

MEMORY SEGMENTATION

The 8086 family of processors employs a segmented architecture - that is, each address is represented as a segment and an offset. Segmented addresses affect many aspects of assembly-language programming, especially addresses and pointers. Segmented architecture was originally designed to enable a 16-bit processor to access an address space larger than 64K. (The section "Segmented Addressing," later in this chapter, explains how the processor uses both the segment and offset to create addresses larger than 64K.) MS-DOS is an example of an operating system that uses segmented architecture on a 16-bit processor. With the advent of protected-mode processors such as the 80286, segmented architecture gained a second purpose. Segments can separate different blocks of code and data to protect them from undesirable interactions. Windows takes advantage of the protection features of the 16-bit segments on the 80286. Segmented architecture went through another significant change with the release of 32-bit processors, starting with the 80386. These processors are compatible with the older 16-bit processors, but allow flat model 32-bit offset values up to 4 gigabytes.

The 8086 BIU sends out 20 bit address so it can address any of 2^{20} or 1,048,576 bytes in memory. However at any given time the 8086 works with only four 65536 bytes (64 Kbyte) segment within this 1,048,576 byte (1M Byte) Range .Four segments are: Code Segment, Stack Segment, Data Segment and Extra Segment .Four segment registers in BIU are used to hold the upper 16 bits of the starting address of 4 memory segments that the 8086 is working with at a particular time. The 4 segment registers are code segment register (CS), stack segment register (SS), data segment register (DS) and the extra segment register (ES). For small programs which do not need all 64 Kbytes in each segment can overlap. For example, the code segment holds the upper 16 bits of the starting address for the segment from which the BIU is currently fetching instruction code bytes. The BIU always inserts zero for the lowest 4 bits of the 20 bit starting address. If the code segment register contains 348A H then the code segment

will start at address 348A0 H .A 64 Kbytes segment can be located anywhere within the 1 MByte address space, but the segment will always start at an address with zeros in the lowest 4 bits.

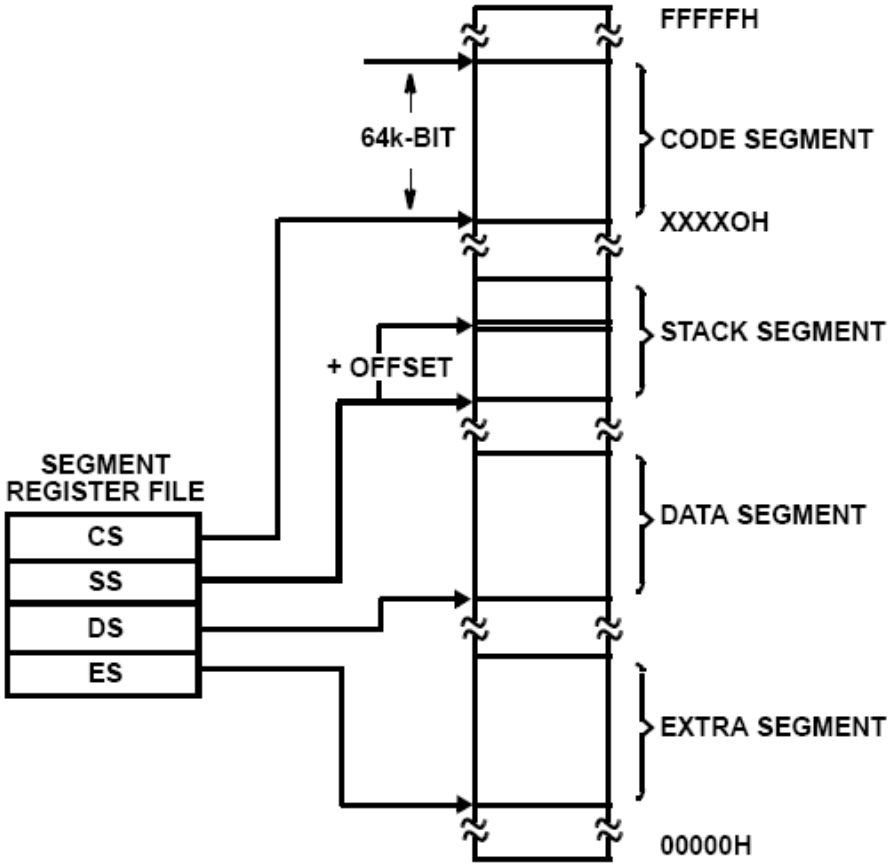

Fig: Memory segmentation of 8086

Instruction Sets of 8086

1. DATA TRANSFER INSTRUCTIONS

1.1 GENERAL PURPOSE BYTE OR WORD TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS

MOV Destination, Source
MOV
MOV CX,04H
	Copy byte or word from specified source to specified destination.

PUSH PUSH Source PUSH BX	Copy specified word to top of stack.
POP POP Destination POP AX	Copy word from top to stack to specified location.
XCHG XCHG Destination, Source XCHG AX,BX	Exchange word or byte.
XLAT	Translate a byte in AL using a table in memory. It first adds AL + BX to form memory address. It then copies the content into AL

1.2 SIMPLE INPUT AND OUTPUT PORT TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS
IN IN AX, Port_Addr IN AX,34H	Copy a byte or word from specified port to accumulator.
OUT OUT Port_Addr, AX OUT 2CH,AX	Copy a byte or word from accumulator to specified port.

1.3 SPECIAL ADDRESS TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS
LEA LEA Register, Source LEA BX,PRICE	Load effective address of operand into specified register.

LDS
LDS Register, Source
LDS BX,[4326H]
	Load DS register and other specified register from memory.

LES	Load ES register and other specified register from memory.

2. ARITHMETIC INSTRUCTIONS

1.4 FLAG TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS
LAHF	Copy to AH with the low byte of the flag register.
SAHF	Stores AH register to low byte of flag register.
PUSHF	Copy flag register to top of stack.
POPF	Copy word at top of stack to flag register.

2.1 ADDITION INSTRUCTIONS

INSTRUCTIONS	COMMENTS

ADD
ADD Destination, Source
ADD AL,74H
	Add specified byte to byte or word to word.

ADC ADC Destination, Source ADC CL,BL	Add byte + byte + carry flag Add word +word + carry flag
INC INC Register INC CX	Increment specified byte or word by 1.
AAA	ASCII adjust after addition.
DAA	Decimal adjust after addition.

2.2 SUBTRACTION INSTRUCTIONS

INSTRUCTIONS	COMMENTS

SUB
SUB Destination, Source
SUB CX,BX
	Subtract byte from byte or word from word.

SBB SBB Destination, Source SBB CH,AL	Subtract byte and carry flag from byte. Subtract word and carry flag from word.
DEC DEC Register DEC CX	Decrement specified byte or word by 1.
NEG NEG Register NEG AL	Form 2's complement.

CMP
CMP Destination, Source
CMP CX,BX
CF ZF SF
CX = BX 0 1 0
CX > BX 0 0 0
CX < BX 1 0 1
	Compare two specified bytes or words.

AAS	ASCII adjusts after subtraction.
DAS	Decimal adjusts after subtraction.

2.3 MULTIPLICATION INSTRUCTIONS

INSTRUCTIONS	COMMENTS
MUL MUL Source MUL CX	Multiply unsigned byte by byte or unsigned word by word. When a byte is multiplied by the content of AL, the result is kept into AX. When a word is multiplied by the content of AX, MS Byte in DX and LS Byte in AX.
IMUL IMUL Source IMUL CX	Multiply signed byte by byte or signed word by word.
AAM	ASCII adjusts after multiplication. It converts packed BCD to unpacked BCD.

2.4 DIVISION INSTRUCTIONS

INSTRUCTIONS	COMMENTS
DIV DIV Source DIV BL DIV CX	Divide unsigned word by byte Divide unsigned double word by byte. When a word is divided by byte, the word must be in AX register and the divisor can be in a register or a memory location. After division AL (quotient) AH (remainder) When a double word is divided by word, the double word must be in DX: AX pair and the divisor can be in a register or a memory location. After division AX (quotient) DX (remainder)
AAD	ASCII adjust before division BCD to binary convert before division.
CBW	Fill upper byte of word with copies of sign bit of lower byte.
CWD	Fill upper word of double word with sign bit of lower word.

3. BIT MANIPULATION INSTRUCTIONS

3.1 LOGICAL INSTRUCTIONS

INSTRUCTIONS	COMMENTS
NOT NOT Destination NOT BX	Invert each bit of a byte or word.
AND AND Destination, Source AND BH,CL	AND each bit in a byte/word with the corresponding bit in another byte or word.
OR OR Destination, Source OR AH,CL	OR each bit in a byte or word with the corresponding bit in another byte or word.
XOR XOR Destination, Source XOR CL,BH	XOR each bit in a byte or word with the corresponding bit in another byte or word.
TEST TEST Destination, Source TEST AL,BH	AND operands to update flags, but don't change the operands.

3.2 SHIFT INSTRUCTIONS

INSTRUCTIONS	COMMENTS
SHL/SAL SAL Destination, Count SHL Destination, Count CF←MSB←LSB←0	Shift Bits of Word or Byte Left, Put Zero(s) in LSB.
SHR SHR Destination, Count 0→MSB→LSB→CF	Shift Bits of Word or Byte Right, Put Zero(s) in MSB.
SAR SAR Destination, Count MSB→MSB→LSB→CF	Shift Bits of Word or Byte Right, Copy Old MSB into New MSB.

3.3 ROTATE INSTRUCTIONS

INSTRUCTIONS	COMMENTS
ROL	Rotate Bits of Byte or Word Left, MSB to LS and to CF.
ROR	Rotate Bits of Byte or Word Right, LSB to MSB and to CF.
RCL	Rotate Bits of Byte or Word Left, MSB to CF and CF to LSB.
RCR	Rotate Bits of Byte or Word Right, LSB TO CF and CF TO MSB.

4. PROGRAM EXECUTION TRANSFER INSTRUCTIONS

4.1 UNCONDITIONAL TRANSFER INSTRUCTION

INSTRUCTIONS	COMMENTS
CALL	Call a Subprogram/Procedure.
RET	Return From Procedure to Calling Program.
JMP	Go to Specified Address to Get Next Instruction (Unconditional Jump to Specified Destination).

4.2 CONDITIONAL TRANSFER INSTRUCTION

INSTRUCTIONS	COMMENTS
JA/JNBE	Jump if Above/Jump if Not Below or Equal.
JAE/JNB	Jump if Above or Equal/Jump if Not Below.
JB/JNAE	Jump if Below/Jump if Not Above or Equal.
JBE/JNA	Jump if Below or Equal/Jump if Not Above.
JC	Jump if Carry Flag CF=1.
JE/JZ	Jump if Equal/Jump if Zero Flag (ZF=1).
JG/JNLE	Jump if Greater/Jump if Not Less than or Equal.
JGE/JNL	Jump if Greater than or Equal/Jump if Not Less than.
JL/JNGE	Jump if Less than/Jump if Not Greater than or Equal.
JLE/JNG	Jump if Less than or Equal/Jump if Not Greater than.
JNC	Jump if No Carry i.e. CF=0
JNE/JNZ	Jump if Not Equal/Jump if Not Zero(ZF=0)
JNO	Jump if No Overflow.
JNP/JPO	Jump if Not Parity/Jump if Parity Odd.
JNS	Jump if Not Sign(SF=0)
JP/JPE	Jump if Parity/Jump if Parity Even (PF=1)
JS	Jump if Sign (SF=1)

4.3 ITERATION CONTROL INSTRUCTIONS

INSTRUCTIONS	COMMENTS
LOOP	Loop Through a Sequence of Instructions Until CX=0.
LOOPE/LOOPZ	Loop Through a Sequence of Instructions While ZF=1 and CX!=0.

4.4 INTERRUPT INSTRUCTIONS

INSTRUCTIONS	COMMENTS
INT	
INT0	Interrupt Program Execution if OF=1
IRET	Return From Interrupt Service Procedure to Main Program.

5 PROCESSOR CONTROL INSTRUCTIONS

5.1 FLAG SET/CLEAR INSTRUCTION

INSTRUCTIONS	COMMENTS
STC	Set Carry Flag CF to 1.
CLC	Clear Carry Flag to 0.
CMC	Complement the State of CF.
STD	Set Direction Flag to 1.
CLD	Clear Direction Flag to 0.
STI	Set Interrupt Flag to 1. (Enable INTR Input).
CLI	Clear Interrupt Enable to 0

5.2 NO OPERATION INSTRUCTION

INSTRUCTIONS	COMMENTS
NOP	No Action Except Fetch and Decode.

5.3 EXTERNAL HARDWARE SYNCHRONIZATION INST.

INSTRUCTIONS	COMMENTS
HLT	Halt (Do Nothing) Until Interrupt or Reset.
WAIT	Wait Until Signal On the TEST Pin is Low.
ESC	Escape to External Coprocessor Such as 8087 or 8089.
LOCK	Prevents Another Processor From Taking the Bus While the Adjacent Instruction Executes.

6 STRING INSTRUCTIONS

INSTRUCTIONS	COMMENTS
REP	Repeat Instruction Until CX=0.
REPE/REPZ	Repeat if Equal/Repeat if Zero
REPNE/REPNZ	Repeat if Not Equal/Repeat if Not Zero.
MOVS/MOVSB/MOVSW	Move Byte or Word From One String to another.
COMPS/COMPSB/COMPSW	Compare Two String Bytes or Two String Words.
SCAS/SCASB/SCASW	Compares a Byte in AL or Word in AX With a Byte or Word Pointed By DI in ES.

Addressing modes of 8086

Addressing modes are the way in which operands are specified within the instruction.
The 8086 has several addressing modes. They are as follows:

1. Register Addressing mode:
   In this addressing mode, on 8-bit or 16-bit general purpose register contains an operand.
   For e.g.: MOV BX, CX
   ADD AL, CH
   ADD CX, DX

2. Immediate Addressing mode:
   In this addressing mode, the operand is contained in the instruction itself.
   The operand forms the part of the instruction.
   For e.g.: MOV AL, 58h
   MOV BX, 0340h
   MOV [1234], 58h

3. Direct Addressing mode:
   In this addressing mode, an effective address (offset) is given in the instruction itself.
   Or the addressing mode, in which the effective address of the memory location at which the data operand is stored is given in the instruction, is known as "Direct Addressing Mode".
   For e.g.: MOV AL, [0300h]
   MOV [0401h], AX

4. Register Indirect Addressing mode:
   The operand's offset is the base register BX or BP or in an (SI or DI) specified in the instruction. For e.g.: ADD CX, [BX]
   MOV DX, [SI]

Note
Displacement: It is an 8 bit or 16 nit immediate values given in the instruction (Relative)
Base: It is the content of the base register BX or BF
Index: It is the content of the index register

5. Based Addressing mode:
   The operand's offset address is the sum of the content of base register BX or BP and an 8-bit or 16 bit displacement. i.e. offset = [BX or BP + 8-bit / 16 bit displacement]
   For e.g.: ADD AL, [BX + 04h]
   ADD AL, [BX + 1234h]

6. Indexed Addressing mode:
   The operand's offset is computed by adding an 8-bit / 16-bit displacement to the context of an index register SI or DI. i.e. offset = [SI or DI + 8-bit / 16-bit displacement]

For e.g.: ADD AX, [SI + 08h]
MOV CX, [SI + 1234h]

7. Based Indexed Addressing mode:
The operand's offset is computed by adding the content of base register to the content of index register. i.e. offset = [BX or BP] + [SI or DI]
BX is used as base register for DS. BF is used as base register for SS.
For e.g.: MOV A, [BX + SI]
ADD CX, [BX + SI]

8. Based Indexed with Displacement mode:
The operand's offset is computed by adding base register's content, an index register content and 8-bit or 16-bit displacement. i.e. offset = [BX or SP] + [SI or DI] + Displacement
For e.g.: MOV AX, [BX + SI + 04h]
ADD CX, [BX + SI + 1234h]

Assembly Language Programming

Introduction

Assembly language is a low level programming language. An Assembly Language program consists of a series of lines that are assembly language instructions. These instructions consist of a mnemonic, which is a command, and an operand, which is the data to be manipulated. The programs usually include comments which are written at the end of a line or in a separate line beginning with a ';' and are ignored by the assembler. Assembly Language uses two, three or 4 letter mnemonics to represent each instruction type.

Advantages of Assembly Language

- Assembly Language Program requires less Memory and execution time compared to High Level Language. Assembly Language is useful for implementing system software.
- Assembly Language gives a programmer the ability to program small embedded system applications. Firmware (that resides in memory while other programs execute) and Interrupt Service Routine (that handles input and output) are developed in Assembly Language.
- It helps to understand sources of program inefficiency and helps in tuning program performance by providing more control over handling particular hardware requirements.
- Helps to write smaller and compact executable codes.

Types of assembler

Assembler can be categorized according to passes.

One Pass Assembler:

A one pass assembler passes over the source file exactly once, in the same pass collecting the labels, resolving future references and doing the actual assembly. The difficult part is to resolve future label references and assemble code in one pass. If each statement of an assembly language program is expressed only in terms of symbols previously defined in the program then the assembler can be implemented in one pass.

One pass assembler can be categorized as *load-and-go* type which produces executable modules and the other one which can produce load, or even objects that could be linked.

All the labels of the instructions are symbols and symbol tables has entry for symbol name, address value .Symbols that are defined in the later part of the program are called forward referencing. The main problem of one pass assembler is forward referencing.

Two pass assembler

A two pass assembler does two passes over the source file (the second pass can be over a file generated in the first pass). In the first pass all it does is looks for label definitions and introduces them in the symbol table. In the second pass, after the symbol table is complete, it does the actual assembly by translating the operations and similar operations.

A two-pass assembler performs two sequential scans over the source code:

First Pass: symbols and literals are defined

Second Pass: object program is generated

First Pass

On the first pass, the assembler performs the following tasks:

•Checks to see if the instructions are legal in the current assembly mode.

•Allocates space for instructions and storage areas you request.

•Fills in the values of constants, where possible.

•Builds a symbol table, also called a cross-reference table, and makes an entry in this table for every symbol it encounters in the label field of a statement

Second Pass

On the second pass, the assembler:

•Examines the operands for symbolic references to storage locations and resolves these symbolic references using information in the symbol table.

•Ensures that no instructions contain an invalid instruction form.

•Translates source statements into machine code and constants, thus filling the allocated space with object code.

•Produces a file containing error messages, if any have occurred.

Assembly Language Program Development Tools

1. EDITOR:

An Editor is a Program which allows user to create (write), modify and store a group of instructions or text under a filename containing the Assembly Language statements for your Program. The saved file must be in ASCII format for assembler to recognize. The file is called source file. It allows user to inputting or modifying text that is stored in mass storage device. Example: Notepad, MS DOS Edit.

2. ASSEMBLER

It translates a program written in assembly language into machine language or object code. An Assembler Program is used to translate the assembly language mnemonics for instruction to the corresponding binary codes.

3. LINKER

A linker is a program that combines object files to create an single "executable" file. Linking is the process of resolving symbols between independent object files of different modules. It resolves all references (ie. Program contains all parts needed to run). It is used to join several files into one large .obj file. It produces .exe file so that the program becomes executable.

4. LOCATOR

A Locator is a program used to assign the specific address of where the segment of object code is to be loaded into memory. It usually converts .exe file to .bin file which has physical address. A Locator program called EXE2BIN converts .exe files to .bin file.

5. DEBUGGER

A Debugger is a program which allows you to load your .obj code program into system memory, execute program and troubleshoot or debug it. Loader is the part of debugger that loads executable files into memory, and may initialize some registers (e.g. IP) and starts it going. Debugger loads but controls the execution of the program to start or stop execution, to view and modify state variables. It allows us to look at the content of registers and memory locations after we run program. It allows to set the breakpoint.

6. EMULATOR

An Emulator is a mixture of hardware and software. It is used to test and debug the hardware and software of an external system such as the prototype of a Microprocessor based instrument. It also allows us to look at the content of registers and memory locations after we run program and take snapshots. We can either use emulator or debugger to develop the program.