UNIT 1
THE 8086 MICROPROCESSOR

Introduction to 8086 – Microprocessor architecture – Addressing modes - Instruction set and assembler directives – Assembly language programming – Modular Programming - Linking and Relocation - Stacks - Procedures – Macros – Interrupts and interrupt service routines – Byte and String Manipulation.

1.1 INTRODUCTION TO 8086

Intel 8086 is a 16-bit microprocessor, contains approximately 29,000 transistors and is fabricated using the HMOS technology. Its throughput is a considerable improvement over that of the Intel 8080, its 8-bit predecessor. Although some attempt at compatibility with the 8080 CPU architecture was made, the designers decided not to sacrifice sophistication in order to attain compatibility. By increasing the number of address pins from 16 to 20, the memory addressing capacity was increased from 64K bytes to 2^{20}= 1 megabyte. The expanded memory capability made multiprogramming feasible and several multiprogramming features have been incorporated into the 8086's design. The 8086 also includes a number of features which enhance its multiprocessing capabilities, thus allowing it to be used with other processing elements such as the 8087 numeric data processor.

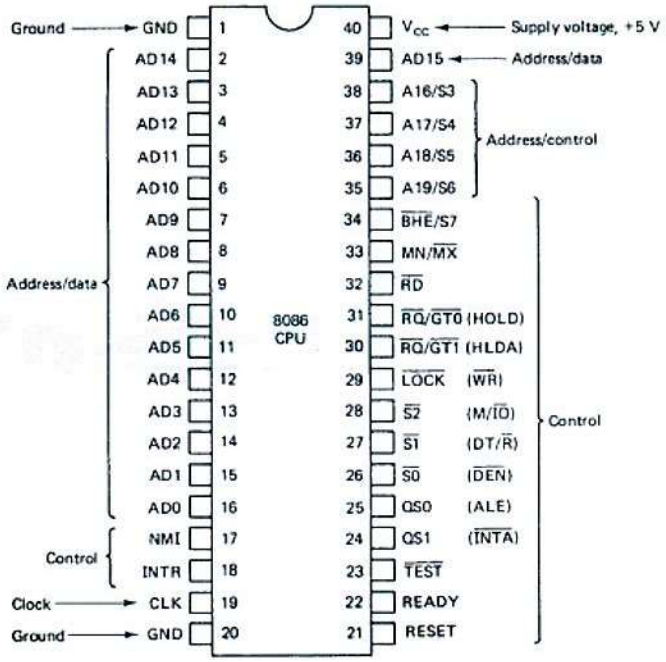


Fig 1.1:8086 Pin assignments

A pin assignment diagram for the 8086 is given in Fig. 1.1. The 8086 has 20 address pins, 16 of which are also used as data pins. The use of pins for both addresses and data means

that both an address and datum cannot be sent to the system bus at the same time. This multiplexing of addresses and data reduces the number of pins needed, but does slow down the transfer of data. However, because of the timing on the bus, the transfer rate is not decreased as much as one might expect. There are 16 control lines for providing handshaking signals during bus transfers and for permitting at least some external control of the CPU. The 8086 requires only one supply voltage, + 5 V, and one clock phase whose frequency can be up to 5 MHz. (There are actually two other versions of the 8086, the 8086-2, which permits a clock frequency of up to 8 MHz, and the 8086-1, which can handle up to 10 MHz.) Rounding out the 40-pin configuration are two grounds, pins 1 and 20.

1.2 MICROPROCESSOR ARCHITECTURE

8086 is a 16 bit microprocessor with 20 bit address bus and 16 bit data bus. Thus it can directly access 220=1,048,576(1MB) memory location and can read/write 8 bits or 16 bit data from /to memory or I/O. The internal architecture of 8086 has two functional units:

1. Bus interface unit
2. Execution unit

Bus interface unit:

The BIU contains Bus interface logic, Segment Registers, Memory addressing logic and a 6-byte Instruction Object Code Queue. The BIU performs all bus operations for the execution unit, and is responsible for executing all bus cycles. When the EU is busy in instruction execution, the BIU continues fetching instructions from memory and stores them in the instruction queue.

If the EU executes an instruction, which transfers the control of the program to another location, then the BIU

- Resets the queue
- Fetches the instruction from the new address
- Passes the instructions to the EU and
- Begins refilling the queue from the new location.

Execution Unit:

The execution unit (EU) contains the complete infrastructure required to execute an instruction, i.e. Instruction decoder, Arithmetic Logic Unit, General Purpose Registers, Pointers and Index Registers, Flag Register and Control Circuitry.

The EU is responsible for

- The execution of all instructions
- Providing address to the BIU for fetching data/instruction, and
- Manipulating various registers as well as the flag register.

1.2.1 Register Set:

The 8086 CPU has fourteen 16-bit registers. These registers are divided into Data group register, segment group register, pointer and index group register. The remaining two group are instruction register (program counter) and flag register.

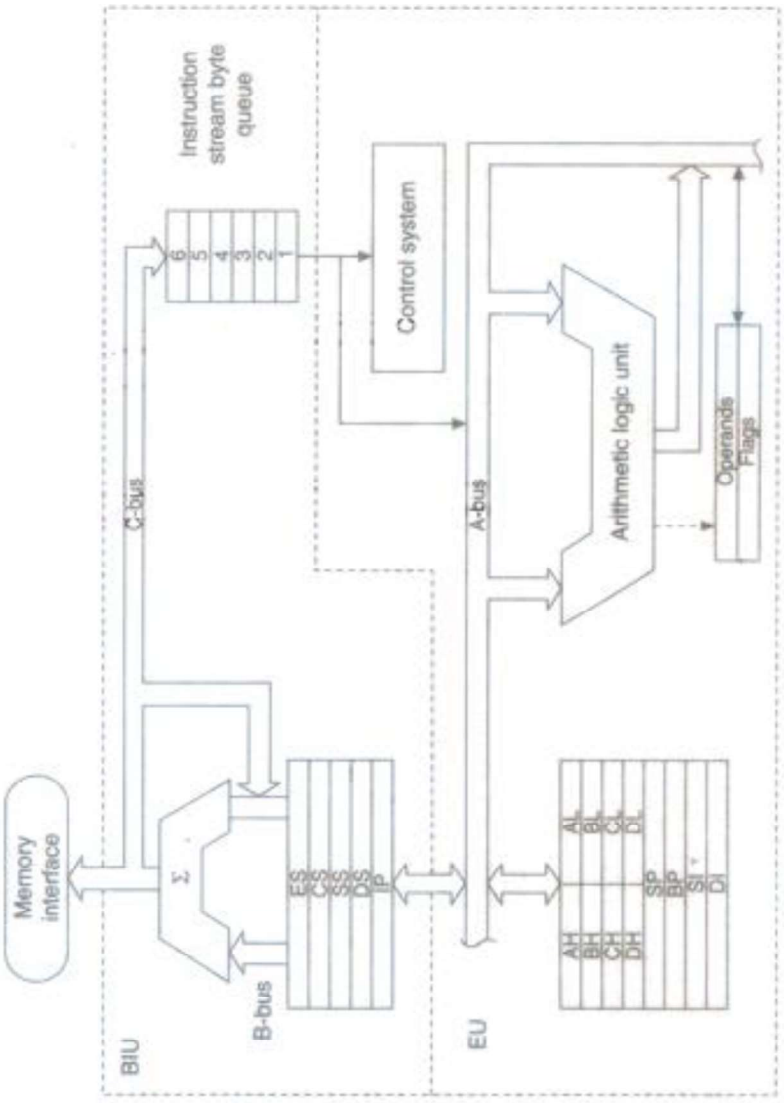


Fig 1.2: 8086 internal architecture

1.2.2 Data Registers

8086 has four 16 bit data registers AX, BX, CX, DX . These registers are unique since their upper and lower bytes can be addressed separately in addition to being single 16 bit register. They can be treated as four 16 bit registers or eight 8 bit registers.

AX Register:

The AX register serves as a primary accumulator. It is unique in the following ways.

- Input/output operations pass through AX (or AL)
- Instructions involving AX (or AL) and immediate data usually require less program memory than that required by other registers.
- Several powerful string primitive instructions require one of the operands to be AX(or AL)
- AX contains the one word operand and the result in 16 bit multiplies and divides instructions whereas AL is used for 8 bit operations. In 32 bit multiply and divide instruction AX is used to hold the lower order word operand.

BX register:

In addition to serving as a general purpose register, DX can be used as base register while computing the data memory address.

CX register:

In addition to serving as a general purpose register, it can be used to hold count in multi iteration instruction. Several 8086 instructions can be made to repeat or to loop. In such instruction CX holds the desired number of repetitions and is automatically decremented after each iteration. When CX becomes zero, the execution of the instruction is terminated.

DX register:

In addition to serving as a general-purpose data register, DX may be used in I/O instructions, multiply and divide instructions. DX contains the addresses of the I/O ports in certain types of I/O instructions. In 32-bit multiply and divide instructions, DX is used to hold the high-order word operand.

1.2.3 Segment registers

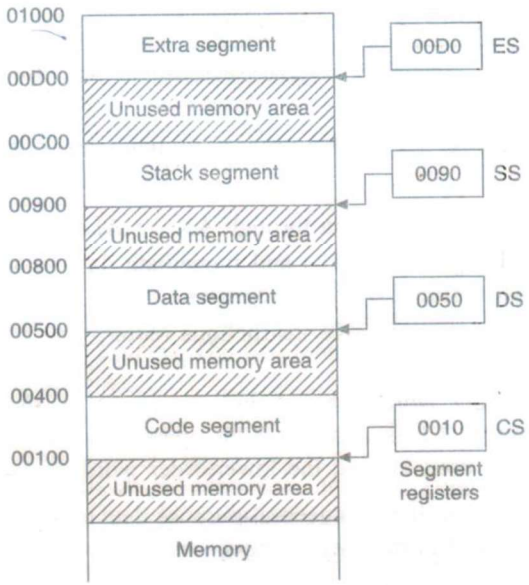


Fig 1.3: Memory segmentation

In the 8086, the 1 MB physical memory is divided into four segments—Code Segment, Data Segment, Stack Segment and Extra Segment. Each segment has memory space of 64 KB. Each segment is addressed by a 16-bit segment register as follows:

CS—Code Segment Register
DS--Data Segment Register
SS—Stack Segment Register
ES- Extra Segment Register

The 8086 memory address is 20 bits. The segment register supplies the higher-order 16 bits of the 20-bit memory address. All memory addresses of the 8086 are computed by summing the contents of the segment register and the offset address.

The offset address is calculated in four different ways for different addressing modes. The selected segment register contents are left shifted by 4 bits. This now represents the starting address of the segment in memory. The effective address, i.e. address basically represents the offset of the starting address of the segment. The offset address is added to segment register contents after 4 bit left-shift, to get the effective address, i.e. the actual memory location address.

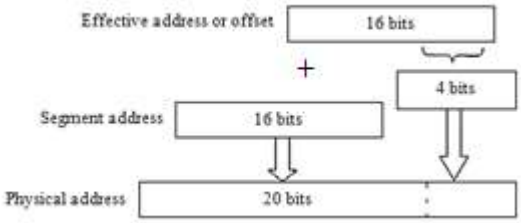


Fig 1.4 : Formation of physical address

For example

Code segment register = 010A (Hex)
Code segment register after 4 bit left-shift = 010A0 (Hex)
Offset address + Offset within code segment = Contents of the instruction pointer = F950 (Hex)

\

```
  010A0
+ F950
-------
  109F0   (Hex)
```

Effective address, i.e. address of the actual memory location

1.2.4 Pointer and index register

The register sin these groups are as follows:

Stack pointer (SP)
Base pointer (BP)
Source index (SI)
Destination index (DI)
Instruction pointer (IP)

Stack pointer (SP):

The stack pointer is used in instructions which use stack, i.e. PUSH, POP, CALL, RET, etc. It always points to a location in memory known as stack top. However, the complete address is formed by adding the contents of the stack segment register, after 4-bit left shift, to the stack pointer.

Base Pointer (BP):

The chief purpose of this register is to provide indirect access to data in stack register. It may also be used for general data storage

Source index (SI) and destination Index (DI):

These registers may be used for general data storage. However, the main purpose of this registers is to store offset in case of indexed, base indexed and relative base indexed addressing modes.

Instruction Pointer (IP):

This register is also referred as program counter. It is used for the calculation of actual memory addresses of instructions. It stores the offset for the instruction. During an instruction fetch, IP contents are added to the code segment register contents after 4 bit left-shift.

1.2.5 Flag Register:

The 8086's PSW contains 16 bits, but 7 of them are not used. Each bit in the PSW is called a flag. The 8086 flags are divided into the conditional flags, which reflect the result of the previous operation involving the ALU, and the control flags, which control the execution of special functions.

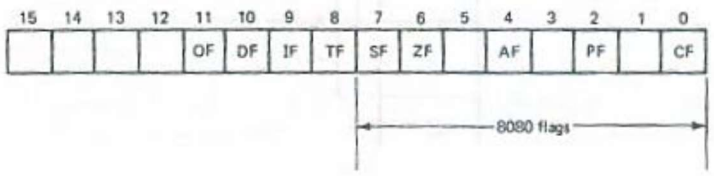


Fig 1.5: 8086 PSW

The condition flags are:

SF (Sign Flag) – After the arithmetic or logic operations, the sign flag is set if the MSB of the result is 1. It indicates the result is negative.

ZF (Zero Flag) – The zero flag is set to 1 if the result is zero and 0 if the result is nonzero.

PF (Parity Flag) - It is set to I if the low-order 8 bits of the result contain an even number of ls; otherwise it is cleared.

CF (Carry Flag) - An addition causes this flag to be set if there is a carry out of the MSB, and a subtraction causes it to be set if a borrow is needed.

AF (Auxiliary Carry Flag) - It is set if there is a carry out of bit 3 during an addition or a borrow by bit 3 during a subtraction. This flag is used exclusively for BCD arithmetic.

OF (Overflow Flag) - is set if an overflow occurs, i.e., a result is out of range. More specifically, for addition this flag is set when there is a carry into the MSB and no carry out of the MSB or vice versa. For subtraction, it is set when the MSB needs a borrow and there is no borrow from the MSB, or vice versa.

As an example, if the previous instruction performed the addition

```
    0010  0011  0100  0101
+   0011  0010  0001  1001
    0101  0101  0101  1110
```

Then following the instruction:

SF=0 ZF=0 PF=0 CF=0 AF=0 OF=0

If the previous instruction performed the addition

```
    0101  0100  0011  1001
+   0100  0101  0110  1010
    1001  1001  1010  0011
```

then the flags would be:

SF=1 ZF=0 PF=1 CF=0 AF=1 OF=1

The control flags are:

DF (Direction Flag) It is used with string instructions. When set causes the string instructions to auto decrement or to process the string from right to left. Otherwise the string instructions are auto incremented i.e. from left to right

IF (Interrupt Enable Flag) –This flag enables the 8086 to recognize the external interrupt requests. When IF=0, all maskable interrupts are disabled. It has no effect on either non-maskable interrupts or internally generated interrupts.

TF (Trap Flag) – If set, it puts the processor into single step mode for debugging..

1.3 MACHINE LANGUAGE INSTRUCTIONS

An instruction is divided into groups of bits, or fields, with one field, called the operation code (or op code), indicating what the computer is to do, and the other fields, called the operands, indicating the information needed by the instruction in carrying out its task. An operand may contain a datum, at least part of the address of a datum, an indirect pointer to a datum, or other information pertaining to the data to be acted on by the instruction. A general instruction format is shown in Fig. 1.6.

Opcode	Operand	…	Operand

Fig 1.6: General instruction format

Instructions may contain several operands, but the more operands and the longer these operands are, the more memory space they will occupy and the more time it will take to transfer each instruction into the CPU.

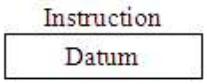
1.4 ADDRESSING MODES

The way in which an operand is specified is called its addressing mode. The addressing modes for the 8086 instructions are typical and are discussed below. They are broken into two categories, those for data and those for branch addresses.

1.Immediate–The datum is either 8 bits or 16 bits long and is part of the instruction.

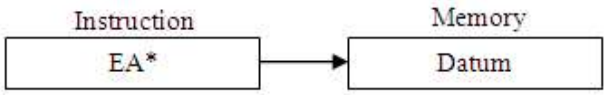
Eg: MOV AX, 1234

Copy into AX the 16-bit data 1234.
(AX) ← 1234



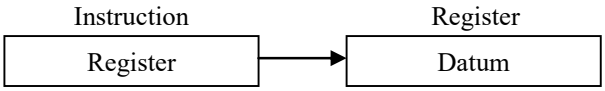
2.Direct-The 16-bit effective address of the datum is part of the instruction.
Eg: MOV (1234), AX

Copy the contents of AX register into memory locations calculated from data segment register and offset 1234.
((DS) x 10H + 1234) ← (AL)
((DS) x 10H + 1234 + 1) ← (AH)



3.Register-The datum is in the register that is specified by the instruction. For a 16-bit operand, a register may be AX, BX, CX, DX, SI, DI, SP, or BP, and for an 8-bit operand a register may be AL, AH, BL, BH, CL, CH, DL, or DH.
Eg: MOV AX, BX

Move the contents of BX register into AX register. The contents of BX register remains unchanged.
(AX) ← (BX)

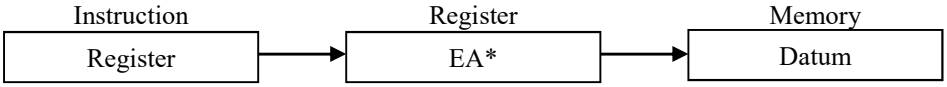


4.Register Indirect-The effective address of the datum is in the base register BX or an index register that is specified by the instruction, i.e.,

$$EA = \begin{Bmatrix} (BX) \\ (DI) \\ (SI) \end{Bmatrix}$$

Eg: MOV AX, (BX)

Copy into AX register the contents of memory location, whose address is calculated using offset as contents of BX register and the contents of DS register.
(AL) ← ((DS) x 10 + (BP))



5.Register Relative-The effective address is the sum of an 8- or 16-bit displacement and the contents of a base register or an index register, i.e.,

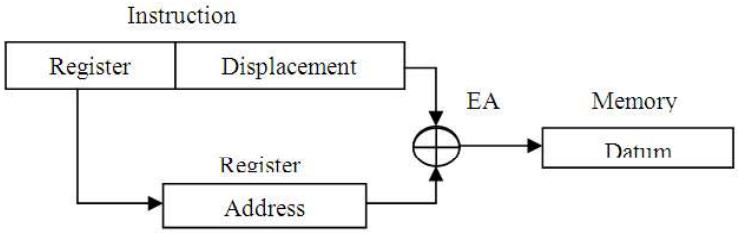
$$EA = \begin{Bmatrix} (BX) \\ (BP) \\ (SI) \\ (DI) \end{Bmatrix} + \begin{Bmatrix} 8-\text{bit displacement} \\ (\text{sign extended}) \\ 16-\text{bit displacement} \end{Bmatrix}$$

Eg: MOV AX, (BX + 06)

Copy the contents of memory location whose address is calculated using DS, DI register with displacement 06, and copy AH the contents of the next higher memory location.

(AL) ← ((DS) x 10H + (DI) +06)

(AH) ← ((DS) x 10H + (DI) +07)



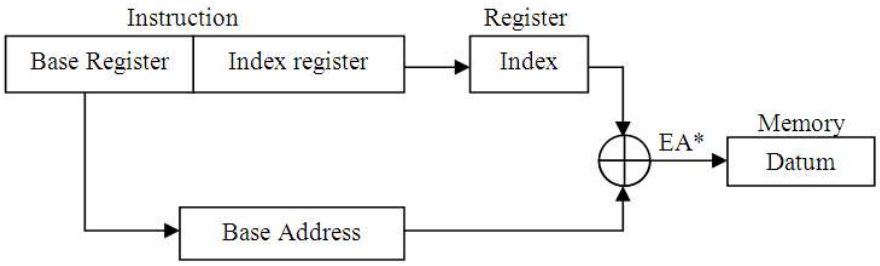
6.Based Indexed - The effective address is the sum of a base register and an index register, both of which are specified by the instruction, i.e.,

$$EA = \begin{Bmatrix} (BX) \\ (BP) \end{Bmatrix} + \begin{Bmatrix} (SI) \\ (DI) \end{Bmatrix}$$

Eg: MOV (BX + DI), AL

Copy the contents of AL register into memory location whose address is calculated using the contents of DS, BX and DI registers.

((DS) x 10H + (BX) + (DI)) ← (AL)



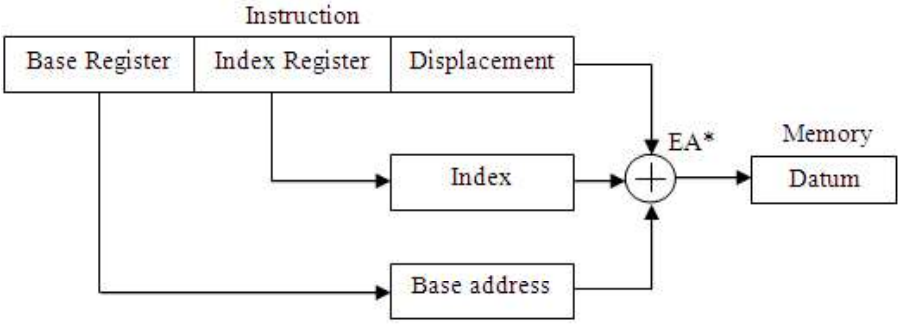
7.Relative Based Indexed -The effective address is the sum of an 8- or 16-bit displacement and a based indexed address, i.e.,

$$EA = \begin{Bmatrix} (BX) \\ (BP) \end{Bmatrix} + \begin{Bmatrix} (SI) \\ (DI) \end{Bmatrix} + \begin{Bmatrix} 8-\text{bit displacement} \\ (\text{sign extended}) \\ 16-\text{bit displacement} \end{Bmatrix}$$

Eg: MOV (BX + DI + 2), CL

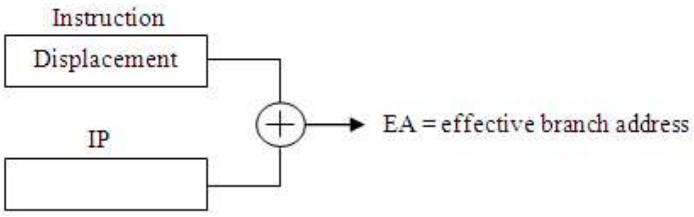
Copy the contents of CL register to the memory location whose address is calculated using DS, BX and DI registers and 02 as displacement.

((DS) x 10 + (BX) + (DI) + 2) ← (CL)

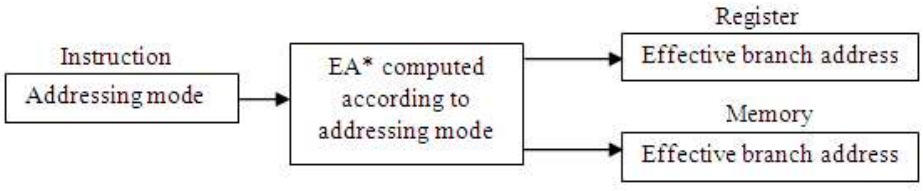


The addressing modes for indicating **branch addresses** are:

1.Intrasegment Direct-The effective branch address is the sum of an 8- or 16- bit displacement and the current contents of IP. When the displacement is 8 bits long, it is referred to as a short jump. Intrasegment direct addressing is what most computer books refer to as relative addressing because the displacement is computed "relative" to the IP. It may be used with either conditional or unconditional branching, but a conditional branch instruction can have only an 8-bit displacement.

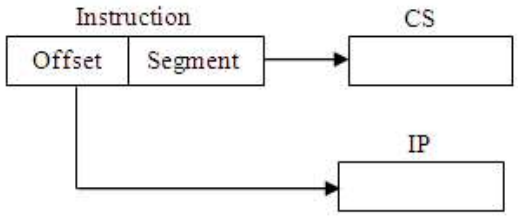


2.Intrasegment Indirect-The effective branch address is the contents of a register or memory location that is accessed using any of the above data-related addressing modes except the immediate mode. The contents of IP are replaced by the effective branch address. This addressing mode may be used only in unconditional branch instructions.

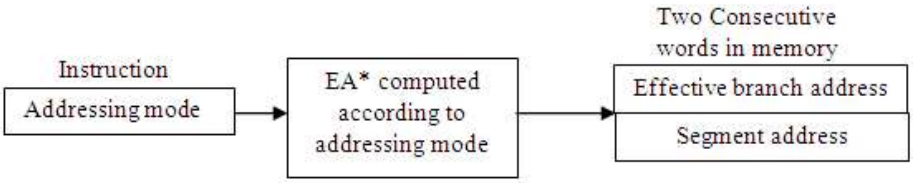


3.Intersegment Direct - Replaces the contents of IP with part of the instruction and the contents of CS with another part of the instruction. The purpose of this addressing mode is to

provide a means of branching from one code segment to another.



4.Intersegment Indirect-Replaces the contents of IP and CS with the contents of two consecutive words in memory that are referenced using any of the above data-related addressing modes except the immediate and register modes.



1.5 ASSEMBLER LANGUAGE PROGRAMMING

Other than machine language code, which consists of the 0-1 combinations that the computer decodes directly, assembler language code is the most primitive form a program can assume. There are two types of statements in an assembler language.

1. There are instructions, which are translated into machine instructions by the assembler, and
2. Directives, which give directions to the assembler during the assembly process but are not translated into machine instructions.

Although each assembler language instruction produces only one machine language instruction, assembler language instructions are easier to write because acronyms, called mnemonics, indicate the type of instruction and character strings, called identifiers, represent addresses and, perhaps, numbers. A typical assembler instruction,

```
ADD     AX, COST
```

adds the contents of the memory location associated with the identifier COST to the register AX. The abbreviation ADD is the instruction mnemonic. An example of a directive is

```
COST    DW      ?
```

which causes the assembler to reserve a word and associate the identifier COST with the word, but does not result in a machine language instruction.

Because an assembler is nothing more than a translating program, the format and syntax of the instructions and directives depend on how the assembler is written, not on the computer. The assembler assumed here is the ASM-86 assembler designed by Intel. It is representative of 8086 assemblers and its major features are representative of assemblers in general.

The features that are available in high-level languages must also be available in assembler languages, even though a single high-level language statement may require several assembler statements to implement it. In addition to including ways of performing branches, loops, I/O arithmetic operations, and assignments, an assembler needs to provide the equivalents of Pre-assignment, storage allocation, naming of constants, commenting, structuring data, calling procedures, statement functions, and global labeling.

Regardless of the level of the language in which programs are written, they involve inputting, processing, and outputting. Complex programs may require an intermixing of these functions as illustrated by the diagram in Fig. 1.7(a).

But simple programs tend to perform them in sequence as shown in Fig. 1.7(b).

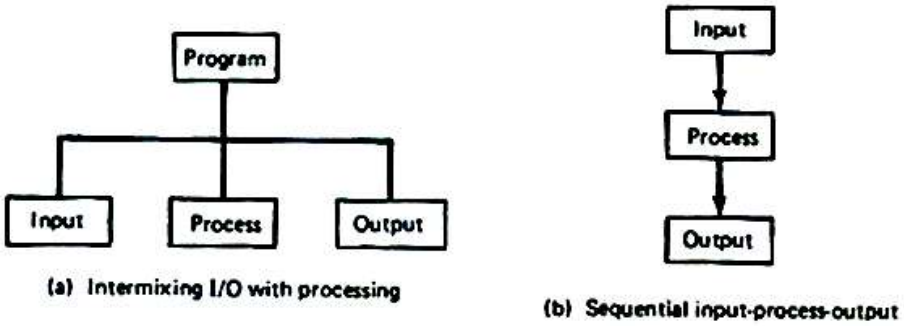


Fig 1.7 :General Program Structure

1.5.1 Assembler Instruction Format

The general format of an assembler instruction is

Label Mnemonic Operand, Operand ;Comments

where the inclusion of spaces is arbitrary, except that at least one space must be inserted if no space would lead to an ambiguity (e.g., between the mnemonic and first operand). Also, there can be no spaces within a mnemonic or identifier and spaces within string constants or comments will be included as space characters. A label is an identifier that is assigned the address of the first byte of the instruction in which it appears. The presence of a label in an instruction is optional, but, if present, the label provides a symbolic name that can be used in branch instructions to branch to the instruction. If there is no label, then the colon must be deleted. All instructions must contain a mnemonic. (Mnemonics may be modified by prefixes-these prefixes are introduced as the need arises.) The presence of the operands depends on the instruction. Some instructions have no operands, some have one, and some have two. If there are two operands, they are separated by a comma. The comments field is for commenting the program and may contain any combination of characters. It is optional and if it is deleted the semicolon may also be deleted. A comment may appear on a line by itself provided that the first character on the line is a semicolon. An instruction may be continued to other lines by placing an ampersand at the beginning of each of the continuation lines.

An assembler language instruction must have an operand for each machine language instruction operand and the notation for each operand must be sufficient to indicate the operand's addressing mode. If there are two operands, the destination operand appears first and

the source operand second. A typical assembler instruction, one which uses the register relative and register addressing modes to add a memory location to the register AX, is given in Fig. 1.8.

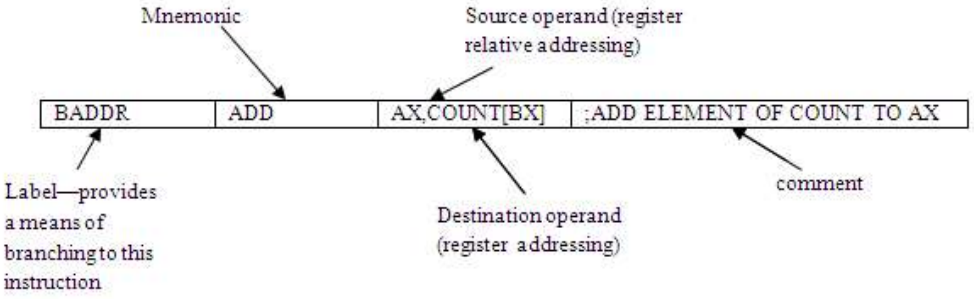


Fig 1.8 Representative assembler language instruction

When an operand is a word in memory the low-order byte of the word will have the lower address and the high-order byte the higher address, e.g., if the identifier COST is used as a word operand, then COST is associated with the low-order byte and COST + 1 with the high-order byte. The syntax for the various types of operands is summarized in Fig. 1.9.

Addressing mode	Format	Example
Data related		
Immediate	Constant expression	10110B 529 0A9H 'AB' YY-XX=5
Direct	Variable± Constant expression	WGT CNT-5 ARRY+5
Register	Register	AX DH
Register indirect	[Register]	[BX]
Register relative	Variable [Register ± Constant expression] (or) [Register ± Constant expression]	VAR X[BX] MESG [SI+10H] [BX-1]
Based indexed	[Base Register][Index Register]	[BP][DI]
Register based Indexed	Var.[Base reg. ± Constant exp][Index Reg ± Constant exp] (or) [Base Reg ± Constant exp][IndexReg ± Constant exp]	[BX+5][SI-2] DATA [BX][SI] [BP+2][DI-9]
Branch-related		
Intrasegment direct	Label ± Constant exp	OVFL ERROR+7
Intrasegment indirect	Same as data related formats except cannot be immediate	

Intersegment direct	Label ± Constant exp	BRANCH EXIT
Intrasegment indirect	Same as data related formats except must not be immediate or register	

Fig 1.9 Operand formats for the addressing modes

These forms may be extended by modifiers that are introduced later, but these basic forms are all that are needed at this point. A variable is an identifier that is associated with the first byte of a datum. An expression is a concatenation of symbols known as tokens, and a constant expression is an expression which can be evaluated to produce a number. The tokens may be variable identifiers or:

Constant –A number whose base is indicated by a suffix as follows:

B-binary
D-decimal
O-octal
H-hexadecimal

The default is decimal. The first digit in a hexadecimal number must be 0 through 9; therefore, if the most significant digit is a letter (A-F), then it must be prefixed with a 0. Examples are:

1011_2 = 1011B
223_{10} = 223D = 223
$B25A_{16}$ = 0B25AH

String Constant –A character string enclosed in single quotes (‘).

Arithmetic Operators –The operators“ + ” , “ – ” , “ *”, and “/”.

Logical Operators –The operators "AND", "OR", "NOT", and "XOR" (exclusive OR). The logical operations are performed by putting the operands in binary form and performing the operation on the corresponding pairs of bits.

Sub expressions –An expression that is part of another expression and is delimited from its parent expression by parentheses.

Name - an identifier that represents a constant, string constant or expression.

The order of precedence for operators is the same as it is in high level languages. An identifier, be it a label, variable, or name, consists of a string of up to 31 characters (actually, it may be longer but the ASM-86 assembler recognizes only the first 31 characters). The characters must be letters, digits, question marks, underscores, or "@" signs except that the first character cannot be a digit. Identifiers cannot be keywords used by the assembler, such as instruction mnemonics, register names, and special operators. Several complete ADD instruc-tions, along with the machine language instructions that they are translated into, are given in Fig. 1.10

It should be emphasized that a segment register designation can appear in an operand of only certain instructions. A segment register can be explicitly specified in only the MOV, PUSH, and POP instructions. Other instructions, such as the branch instructions, may implicitly reference the segment registers. Except for string operations, for a double-operand instruction one of the operands must be a register unless the source operand is immediate. In no case can IP

be specified as an operand because only a branch or other control transfer instruction can change the contents of this register.

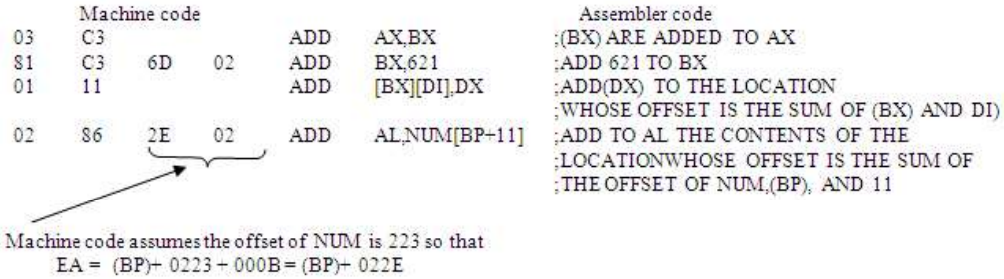


Fig 1.10: Typical assembler language instructions

Many instructions can operate on either bytes or words, depending on whether or not the W-bit is set. Normally, at least one of the operands is such that a byte or a word operation is implied by its notation or by how a memory operand is defined, and the assembler sets or clears the W-bit accordingly, e.g., because of the destination operand the assembler will assume that

ADD AX,[BX] is a word addition and

ADD AL,[BX] a byte addition.

Also, if appropriate directives are used to define COST to be a word variable and COUNT to be a byte variable, then

INC COST will be a word operation, and

INC COUNT will be a byte operation.

For some situations, however, it is impossible for the assembler to deduce the operand type. The instruction INC [BX]increments the quantity whose address is in BX, but should it increment a byte or a word? One of the purposes of the PTR operator is to specify the length of a quantity in this and other ambiguous situations. It is applied by writing the desired type followed by PTR. For the above INC instruction the PTR operator would be used to modify the operand as follows:

INC BYTE PTR [BX] if a byte is to be incremented, or

INC WORD PTR [BX] if a word is to be incremented.

1.6 Glossary of Symbols and Abbreviations

Abbreviation	Meaning	Symbol	Meaning
OPR	Operand	←	Replaced by
SRC	Source operand	↔	Interchanged with
DST	Destination operand	(X)	Contents of X
REG	Register	Λ	Logical AND
RSRC	Source register	V	Logical OR
RDST	Destination register	∀	Logical exclusive OR
CNT	Count	$\bar{X}$	Complement of X
DISP	Displacement (general)		
DB	8-bit displacement		

D16	16-bit displacement		
ADDR	Address		
EA	Effective address		
SEG	Segment		
DATA	Immediate operand (general)		
DATA8	8-bit immediate operand		
DATA16	16-bit immediate operand		
AX,BX,CX,DX, AL,AH,BL,BH, CL,CH,DL,DH, IP,BP,SP,SI,DI	8086 register names		
CS,SS,DS,ES	8086 segment register names		
DF,IF,TF	8086 control flags name		
OF,SF,ZF,AF,PF, CF	8086 condition flags name		

1.7 INSTRUCTION SET

1.7.1 Data Transfer Instructions

All computers need a variety of instructions solely for moving data, addresses, and immediate operands into register or memory locations.

There are four basic 8086 instructions for transferring quantities to and/or from the registers and memory; they are the MOV, LEA, LDS, and LES instructions and are defined below:

MOV DST, SRC

(DST) ← (SRC)

The MOV instruction is for moving a byte or word within the CPU or between the CPU and memory. Depending on the addressing modes it can transfer information from a:

Register to a register.
Immediate operand to a register.
Immediate operand to a memory location.
Memory location to a register.
Register to a memory location.
Register/memory location to a segment register (except CS).
A segment register to a register/memory location.

None of the flags are changed by the execution of a move instruction.

LEA Load effective address

Transfers the offset of src to the destination register.

LEA REG, SRC

(REG) ← (SRC)

LDS Load DS with pointer

Loads the 16- bit pointer from memory source to destination register and DS. The offset is placed in destination register and the segment is placed in DS.

LDS REG, SRC

(REG) ← (SRC)

(DS) ← (SRC + 2)

LES Load ES with pointer

Loads the 16- bit pointer from memory source to destination register and ES. The offset is placed in destination register and the segment is placed in ES.

LES REG, SRC

(REG) ← (SRC)

(ES) ← (SRC + 2)

XCHG Exchange

Exchanges the contents of source and destination

XCHG OPR1, OPR2

(OPR2) ↔ (OPR1)

PUSH Push word onto stack

Decrements SP by 2 and transfers one word from source to stack top

PUSH SRC

(SP) ← (SP) – 2

((SP)) ← (SRC)

POP Pop word off stack

Transfers the word at the current stack top to the destination and then increments SP by 2 to point to the new stack top.

POP SRC

((SP)) ← (SRC)

(SP) ← (SP) + 2

LAHF Load AH from flags

Copies bit 0-7 of the flag register into AH. This includes flags AF, CF, PF, SF and ZF.

AH = SF ZF xx AF xx PF xx CF

SAHF Store AH into flags

Copies bit 0-7 of the flag register into AH. This includes flags AF, CF, PF, SF and ZF.

AH = SF ZF xx AF xx PF xx CF

XLAT translate

Replaces the byte in Al with the byte from the user table addressed by BX.

(AL) ← ((AL) + (BX))

IN Input Byte or word from port

A byte or word is read from port and placed in AL or AX.

IN AL, PORT

(AL) ← (PORT)

OUT Output data to port

A byte or word is read from AL or AX and placed in port.

OUT AL, PORT

(PORT) ← (AL)

1.7.2 Arithmetic Instructions

The 8086 instruction set includes instructions for performing arithmetic operations in binary, packed BCD, and unpacked BCD.

Binary Arithmetic

The binary addition and subtraction instructions are defined below:

ADD Arithmetic addition

Adds source to destination and replaces the original contents of destination. Both the operands are binary.

ADD DST, SRC

(DST) ← (SRC) + (DST)

ADC Add with carry

Sums two binary operands placing the result in destination. If CF is set, a 1 is added to the destination

ADC DST, SRC

(DST) ← (SRC) + (DST) + (CF)

SUB Subtract

The source is subtracted from the destination and the result is stored in destination.

SUB DST, SRC

(DST) ← (DST) - (SRC)

SBB Subtract with borrow

The source is subtracted from the destination and the result is stored in destination.

SBB DST, SRC

(DST) ← (DST) - (SRC) – (CF)

MUL Unsigned multiply

Unsigned multiply of the accumulator by the source.

MUL SRC

$(AX) \leftarrow (AL) * (SRC)$

Word operands

$(DX: AX) \leftarrow (AL) * (SRC)$

IMUL SRC Signed multiply

Signed multiply of the accumulator by the source.

IMUL SRC

$(AX) \leftarrow (AL) * (SRC)$

Word operands

$(DX: AX) \leftarrow (AL) * (SRC)$

DIV divide

Unsigned binary division of the accumulator by the source.

DIV SRC

$(AL) \leftarrow \text{Quotientof } (AX)/(SRC)$

$(AH) \leftarrow \text{Remainderof } (AX)/(SRC)$

Word divisor

$(AX) \leftarrow \text{Quotientof } (DX: AX)/(SRC)$

(DX) ← Remainderof (DX: AX)/(SRC)

IDIV Signed integer division

Unsigned binary division of the accumulator by the source.

IDIV SRC

(AL) ← Quotientof (AX)/(SRC)

(AH) ← Remainderof (AX)/(SRC)

Word divisor

(AX) ← Quotientof (DX: AX)/(SRC)

(DX) ← Remainderof (DX: AX)/(SRC)

INC Increment

Adds 1 to the destination unsigned binary operand.

INC OPR

(OPR) ← (OPR) + 1

DEC Decrement

Unsigned binary subtraction of 1 from the destination.

DEC OPR

(OPR) ← (OPR) - 1

NEG Negate

NEG OPR

(OPR) ← - (OPR)

CMP Compare

Subtracts the source from destination and updates the flag but does not save the result.

CMP OPR1, OPR2

(OPR1) – (OPR2)

Sign Extension instructions

CBW Convert byte to word

Extend sign of AL to AH

CWD Convert word to double word

Extend sign of AX to DX

Packed BCD Arithmetic

Packed BCD numbers are stored two digits to a byte, in 4-bit groups referred to as nibbles. The ALU is capable of performing only binary addition and subtraction, but by adjusting the sum or difference the correct result in packed BCD format can be obtained. The correction rule for addition is:

For example:

Carry from	→	0	0	
Previous digit		7	0111	
	+	6	+ 0110	
		13	1101	
			110	Add 6 because 1101 is not a valid BCD digit
Carry to next digit		→ 1	0011	

Carry from	→	1		1	
Previous digit		9		1001	
	+	9	+	1001	
		19	1	0011	
				110	Add 6 because of carry to next digit
Carry to next digit		→ 1		1001	

DAA Decimal Adjust for Addition

(AL) ← Sum in AL adjusted to packed BCD format

DAS Decimal Adjust for subtraction

(AL) ← Difference in AL adjusted to packed BCD format

Unpacked BCD Arithmetic

AAA Unpacked BCD adjust for addition

(AL) ← Sum in AL adjusted to unpacked BCD format

(AH) ← (AH) + Carry from adjustment

AAS Unpacked BCD adjust for subtraction

(AL) ← Difference in AL adjusted to unpacked BCD format

(AH) ← (AH) - borrow from adjustment

AAM Unpacked BCD adjust for multiplication

(AL) ← Product in AL adjusted to unpacked BCD format with AH containing higher order digit.

(AH) : = (AL) /10

(AL) : = (AL) mod 10

AAD Unpacked BCD adjust for division

Used before dividing the unpacked decimal numbers.

(AL) ← (AH) X 10 + (AL)

(AH) ← 0

1.7.3 Branch Instructions

A branch instruction transfers control from the normal sequence of instructions by putting the address of the instruction to be branched to in the program counter. If the branch is conditional, then whether or not the contents of the program counter are replaced with the branch address depends on the flags. For the 8086, since a physical address is comprised of both an offset and a segment address, some branch (or jump) instructions modify both the IP and CS registers while others modify only the IP register. Those instructions that change the contents of the CS register are referred to as intersegment branches and the remaining branch instructions are called intrasegment branches. Only unconditional branch instructions can cause intersegment branches.

Conditional Branch Instructions

All conditional branch instructions have the following 2-byte machine code format:

where the second byte gives an 8-bit signed (2's complement) displacement relative to the address of the next instruction in sequence. The effective branch address is determined by sign extending D8 and adding it to the contents of IP after IP has been incremented to point to the next instruction. Therefore, a negative D8 means a backward branch from the next instruction and a positive D8 means a forward branch from the next instruction. The limitation of D8 to 8 bits restricts the distance between the address of the next instruction and the branch address to the range -128 to 127 bytes.

Name	Mnemonic and format	Alternate mnemonic	Test condition*
Branch on zero, or equal	JZ OPR	JE	ZF=1
Branch on non zero, or not equal	JNZ OPR	JNE	ZF=0
Branch on sign set	JS OPR		SF=1
Branch on sign clear	JNS OPR		SF=0
Branch on overflow	JO OPR		OF=1
Branch on no overflow	JNO OPR		OF=0
Branch on parity set, or even parity	JP OPR	JPE	PF=1
Branch on parity clear, or odd parity	JNP OPR	JPO	PF=0
Branch on below, or not above or equal (unsigned)	JB OPR	JNAE,JC	CF=1
Branch on not below, or above or equal (unsigned)	JNB OPR	JAE,JNC	CF=0
Branch on below or equal, or not above (unsigned	JBE OPR	JNA	CF V ZF = 1
Branch on not below or equal, or above (unsigned)	JNBE OPR	JA	CF V ZF = 0
Branch on less, or not greater or equal (signed)	JL OPR	JNGE	SF ∀ OF =1
Branch on not less, or greater or equal (signed)	JNL OPR	JGE	SF ∀ OF =0
Branch on less or equal, or not greater (signed)	JLE OPR	JNG	(SF ∀ OF) V ZF=1
Branch on not less or equal, JNLE OPR or greater (signed)	JNLE OPR	JG	(SF ∀ OF) V ZF=0

Some of them have alternate mnemonics, e.g., JNL (branch on not less than) is equivalent to JGE (branch on greater than or equal to).

Unconditional Branch Instructions

There are five unconditional branch instructions. Their machine code formats are summarized below:

Name	Mnemonic and format			Description
Intrasegment direct short branch	JMP SHOPT OPR	(IP)	←	(IP) + sign extended D8 determined by OPR
Intrasegmentdirect near branch	JMP NEAR PTR OPR	(IP)	←	(IP)+16 bit displacement determined by OPR

Intrasegment indirect branch	JMP	OPR*	(IP)	←	(EA) where EA is determined by OPR
Intrasegment direct (far) branch	JMP	FAR PTR OPR	(IP)	←	Offset of OPR within segment
			(CS)	←	Segment address of segment containing OPR
Intrasegment indirect branch	JMP	OPR*	(IP)	←	(EA) where EA is determined by OPR
			(CS)	←	(EA+2) where EA is determined by OPR

Three of these instructions are for branching to a point in the current segment and two of them are for branching to a different segment. The former type of branching is referred to as an intrasegment branch and the latter as an intersegment branch. All five instructions have the same mnemonic, JMP, and include a single operand.

1.7.4 Loop Instructions

The loop instructions are designed to simplify the decrementing, testing, and branching portion of the loop. In the above case this portion has required two instructions, but in more complicated situations may require more than two instructions.

The loop instructions for the 8086 all have the form

where D8 is a 1-byte displacement from the current contents of IP. Their similarity to the conditional branch instructions is evident and they must obey the same limitations, including the one that restricts the branch address to the range -128 to 127 bytes from the next instruction. The loop instructions are defined below:

Name	Mnemonic and Format		Alternate Mnemonic	Test Condition*
Loop	LOOP	OPR		(CX) ≠0
Loop while zero, or equal	LOOPZ	OPR	LOOPE	ZF =1 and (CX) ≠ 0
Loop while nonzero, or not equal	LOOPNZ	OPR	LOOPNE	ZF = 0 and (CX) ≠ 0
Branch on CX	JCXZ	OPR		(CX) = 0

1.7.5 NOP and HLT instructions

An instruction that is quite often used in conjunction with a branch instruction is the no operation, or no op, instruction. The no op instruction for the 8086 has the mnemonic NOP and is defined below:

Name	Mnemonic and Format	Description
No operation	NOP	Cause no action
Halt	HLT	Cause the operation of the computer to cease

1.7.6 Flag Manipulation Instructions

Many instructions set or clear the flags depending on their results. Sometimes, however, it is necessary to have direct control of the flags.

Name	Mnemonic and format	Description
Clear carry	CLC	$CF \leftarrow 0$
Complement carry	CMC	$CF \leftarrow \overline{CF}$
Set carry	STC	$CF \leftarrow 1$
Clear direction	CLD	$DF \leftarrow 0$
Set direction	STD	$DF \leftarrow 1$
Clear interrupt	CLI	$IF \leftarrow 0$
Set interrupt	STI	$IF \leftarrow 1$
Load AH from flags	LAHF	$(AH) \leftarrow (LOW - ORDER\ BYTE\ OF\ PSW)$
Store AH to flags	SAHF	$(LOW - ORDER\ BYTE\ OF\ PSW) \leftarrow (AH)$

The STC, CLC, and CMC instructions, which respectively set, clear, and complement the carry flag CF, are typically used in multiple-precision arithmetic. STD and CLD set and clear the direction flag DF, which affects the operation of the string manipulation instructions, and STI and CLI set and clear the interrupt flag IF, which controls the maskable interrupts. The other two instructions defined are designed to allow the efficient translation into 8086 programs of programs written for the 8080. The LAHF and SAHF instructions transfer the low-order byte of the PSW to and from the AH register. Except for the overflow flag (the flag with no 8080 counterpart), this byte includes all of the condition flags. LAHF and SAHF can be used with the logical instructions to initialize or compare the SF, ZF, AF, PF, and CF flags with a given bit pattern.

1.7.7 Logical Instructions

AND Logical AND

Performs the logical AND of the two operands replacing the destination with the result.

AND DST, SRC

(DST) ← (DST) Λ (SRC)

OR Logical OR

Performs the logical OR of the two operands replacing the destination with the result.

OR DST, SRC

(DST) ← (DST) V (SRC)

XOR Exclusive OR

Performs the exclusive OR of the two operands replacing the destination with the result.

OR DST, SRC

(DST) ← (DST) ∀ (SRC)

TEST Test for Bit pattern

Performs a logical AND of the two operands updating the flag register without saving the result.

TEST OPR1, OPR2

(OPR1) $\wedge$ (OPR2)

NOT 1's complement negation

Inverts the bit in destination operand forming the 1's complement.

NOT OPR

$(OPR) \leftarrow \overline{OPR}$

NEG 2's complement negation

Inverts the bit in destination operand forming the 1's complement and then adds a 1 along with it.

NEG OPR

$(OPR) \leftarrow \overline{OPR} + 1$

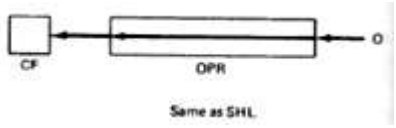
1.7.8 Shift and Rotate Instructions

The shift and rotate instructions for the 8086 are defined below:.

SHL Shift logical left/ Shift arithmetic left

Shifts left the destination by count bits with zeros shifted to the right. The carry flag contains the last bit shifted out.

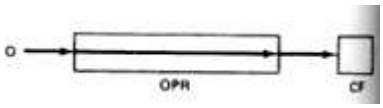
SHL OPR,CNT / SAL OPR, CNT



SHR Shift logical right

Shift right the destination by count bits with zeros shifted to the left. The carry flag contains the last bit shifted out.

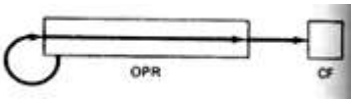
SHR OPR,CNT



SAR Shift arithmetic right

Shift right the contents of the data memory location by count bits with the current sign bit replicated in the leftmost bit. The carry flag contains the last bit shifted out.

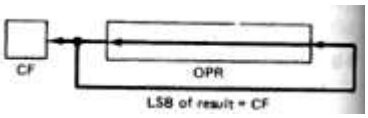
SAR OPR,CNT



ROL Rotate left

Rotates the bits in the destination to the left count times with all the data pushed out from the left side, reentering on the right. The carry flag will contain the value of the last bit rotated out.

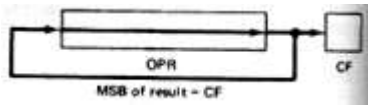
ROL OPR,CNT



ROR Rotate right

Rotates the bits in the destination to the right count times with all the data pushed out from the right side, reentering on the left. The carry flag will contain the value of the last bit rotated out.

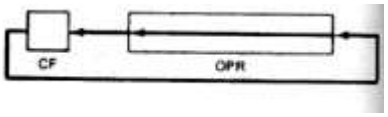
ROR OPR,CNT



RCL Rotate through carry left

Rotates the bits in the destination to the right count times through the carry flag with all the data pushed out from the right side, reentering on the left. The carry flag holds the last bit rotated out.

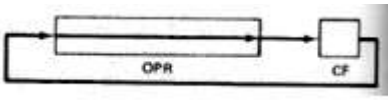
RCR OPR, CNT



RCR Rotate through carry right

Rotates the bits in the destination to the right count times through the carry flag with all the data pushed out from the right side, reentering on the left. The carry flag holds the last bit rotated out.

RCR OPR, CNT



These instructions shift all of the bits in the operand to the left or right by the specified count, CNT. For the shift left instructions, 0s are shifted into the right end of the operand and the MSBs are shifted out the left end and lost, except that the least significant of these MSBs is retained in the CF flag. The shift right instructions similarly shift bits to the right; however, SAR (shift arithmetic right) does not automatically insert 0s from the left; it extends the sign of the operand by repeatedly inserting the MSB. The rotate instructions differ from the shift instructions in that the operand is treated like a circle in which the bits shifted out of one end are shifted into the other end. The RCL and RCR instructions include the carry flag in the circle and the ROL and ROR do not, although the carry flag is affected in all cases.

1.8 ASSEMBLER DIRECTIVES

There are certain instructions in assembly language program which are not the actual instructions of the processor and they are not translated to machine language instructions

1.8.1 Data Definition and Storage Allocation

Statements that pre-assign data and reserve storage have the form:

Variable Mnemonic Operand, ... , Operand ;Comments

where the variable is optional, but if it is present it is assigned the offset of the first byte that is reserved by the directive. Unlike the label field, a variable must be terminated by a blank, not a colon. The mnemonic determines the length of each operand and is one of the following:

DB (Define Byte)-Each operand datum occupies one byte.

DW (Define Word)-Each operand datum occupies one word, with its loworder part being in the first byte and its high-order part being in the second byte.

DD (Define Double Word)-Each operand datum is two words long with the low-order word followed by the high-order word.

The operands indicate the data to be pre-assigned or the amount of space to be reserved. Comments may be used to describe the data assignment or allocation or may be omitted.

For example,

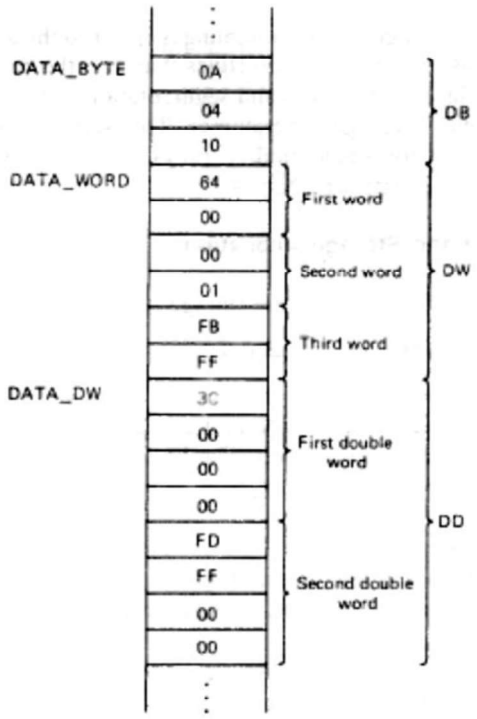


Fig 1.11: Typical preassignment of data using the DB,DW and DD directives

```
DATA_BYTE   DB      10,4, 10H
DATA_ WORD  DW      100,100H,-5
DATA_DW     DD      3*20,0FFFDH
```

will cause a sequence of bytes to be pre-assigned as shown in Fig. 1.11.

As another example, an 8-digit number 12345678can be defined in packed BCD form by

```
PACKED      DB      78H,56H,34H,12H
```

and in unpacked BCD form by

```
UNPACKED DB 8H,7H,6H,5H,4H,3H,2H,1H.
```

The statements

```
ABC DB 0,?,?,?,0
DEF DW ?,52,?
```

assemble the bytes shown in Fig. 1.12.

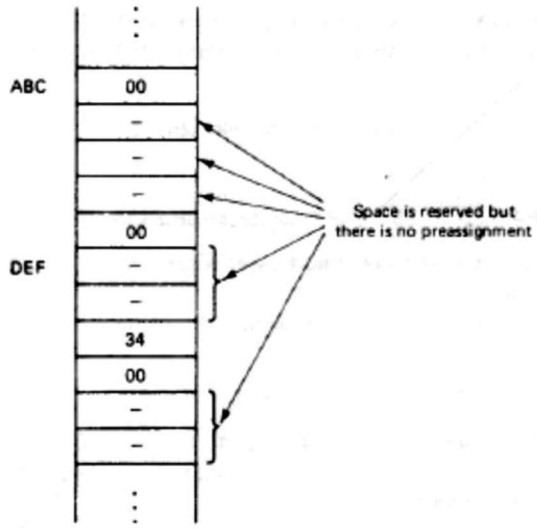


Fig 1.12: Application of "?" operand

The statements

```
ARRAY1      DB      2DUP(0,1,2, ?)
ARRAY2      DB      100 DUP(?)
```

would cause the pre-assignment and allocation shown in Fig. 1.13(a).

There is, however, a way of overriding the implicit type given to an operand, and that is to used PTR attribute operator in a form such as

Type PTR Variable ± Constant expression

where the type may be BYTE, WORD, or DWORD (for double word). In the above sequence

```
MOV WORD PTR OPER1+1,AX
```

would allow a word MOV instruction to be assernbled without generating an error.

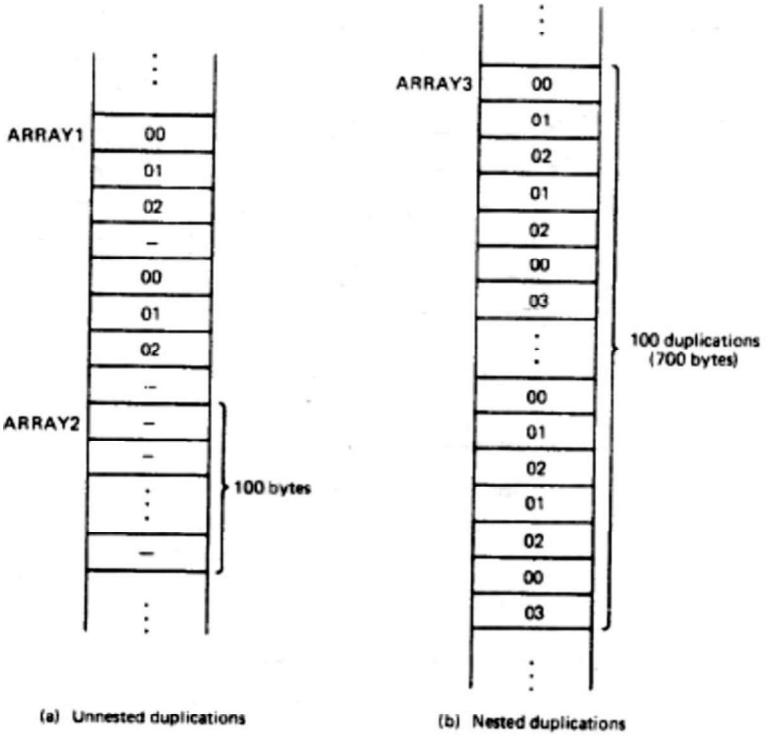


Fig 1.13: Application of the DUP operator

Also, MOV BYTE PTR OPER2,AL would replace the first byte of the first word with the contents of AL. On the other hand, the instruction

MOV BYTE PTR OPER2+1,AL

would cause the high-order byte of the first word of OPER2 to be replaced.

Since it may become tedious to use the PTR operator to frequently reference variables that are accessed as two different types, a method has been provided for associating a location with two different variables. The LABEL directive, which has the form

Variable LABEL Type

causes the variable to be typed and assigned to the current offset.

1.8.2 Structures

A structure definition gives the pattern of the structure and may have the simplified form Structure name STRUC

:

:

:Sequence of DB, DW, and DD directives

:

Structure name ENDS

All elements allocated by a single storage definition statement must be of the same type (bytes, words, or double words). 1t is desirable, especially in business data processing

applications, for a variable to have several fields, with each field having its own type. For instance, the data structure for a personnel record might be as shown in Fig. 1.14.

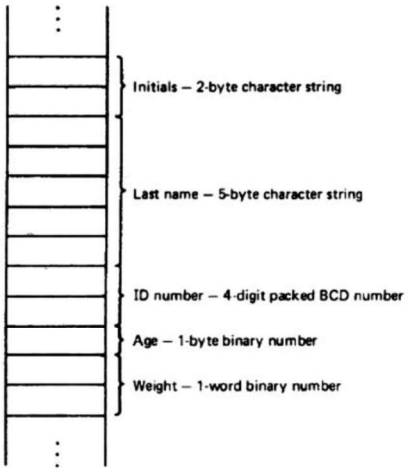


Fig 1.14: Fields in a typical personnel record data structure

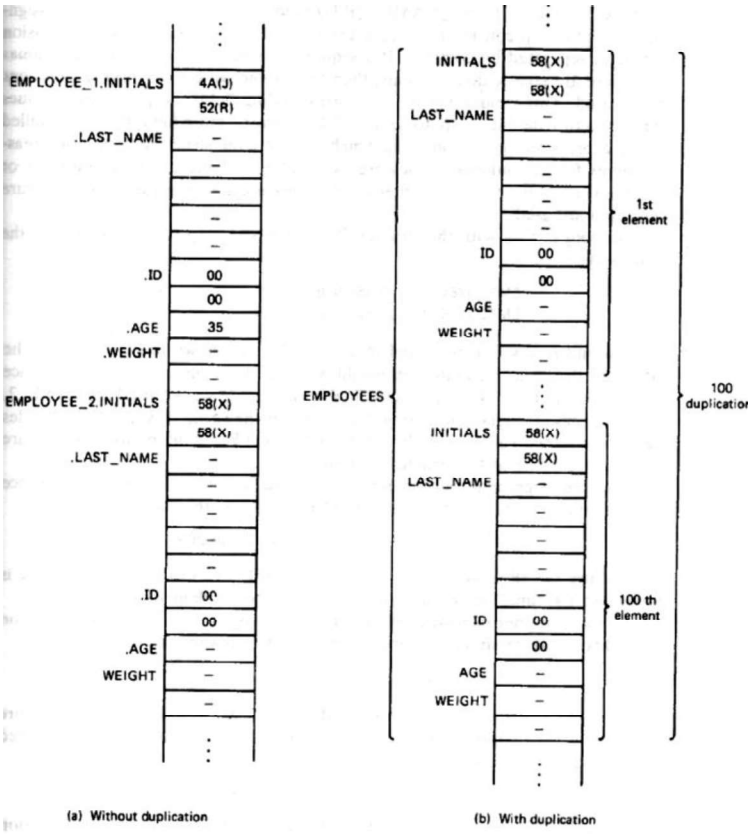


Fig 1.15: Allocation and preassignment of structures

The structure for the personnel record shown in Fig 1.15 could be defined

```
PERSONNEL_DATA STRUC
        INITIALS        DB 'XX'
        LAST_NAME       DB 5 DUP(?)
        ID              DB 0,0
        AGE             DB ?
        WEIGHT          DW ?
PERSONNEL_DATA ENDS
```

1.8.3 Records

The RECORD directive is for defining a bit pattern within a word or byte. RECORD creates the pattern and it is left to other statements to invoke the pattern while reserving and preassigning storage. It has the form

Record name RECORD Field specification, ... ,Field specification

For example,

PATTERN RECORD OPCODE:5,MODE:3,0PR1:4=8,0PR2:4=5

would break a word into four fields and give them the names OPCODE, MODE, OPRl, and OPR2. The lengths of the fields in bits would be 5, 3, 4, and 4, respectively. The statement

INSTRUCTION PATTERN <..,5>

would actually reserve the word, associate it with the variable INSTRUCTION, and preassignOPRl to 8 and OPR2 to 5 as shown in Fig. 1.16.

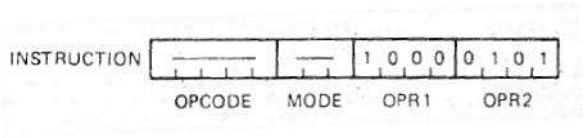


Fig 1.16: Typical use of RECORD to subdivide a word

1.8.4 Assigning Names to Expressions

The statement that assigns a name to an expression has the form

Expression name EQU Expression

where the expression name may be any valid identifier and the expression may have the format of any valid operand, be any expression that evaluates to a constant (the expression name is then a constant name), or be any valid mnemonic. Several examples of expressions are given in Fig. 1.17.

Another important reason for using names in place of instructions is that they permit a program to be easily modified. For example, suppose that the instruction

ADD BX,6

appears several times within a program and it is known that in some applications of the program the 6 will need to be replaced with a different number. BX may contain an address and the purpose of the instruction may be to increment this address between fields in a structure. If the program must be rewritten to accommodate a different length, then the instructions would need to be changed.

Directive			Description
CONSTANT	EQU	256	Assigns 256 the name CONSTANT
DATA	EQU	HEIGHT+12	Assigns the direct operand HEIGHT+12 the name DATA
FIELD	EQU	[BX].ETA[SI]	Assigns the structure referencing operand [BX].ETA[SI] to the name FIELD
ALPHA	EQU	7	This combination would give 7 the name
BETA	EQU	ALPHA-2	ALPHA, 7-2=5 the name BETA, and the
ADDR	EQU	VAR+BETA	direct operand VAR+BETA = VAR+5 the name ADDR

Fig 1.17: Examples of EQU directive

However, by including the statement

NUM EQU 6

at the beginning of the program and writing the ADD instructions as follows:

ADD BX,NUM

the increment could everywhere be changed to, say 8, by simply replacing the EQU statement with NUM EQU 8

This application of the EQU statement is particularly useful in addressing I/O ports because, when the system configuration is changed, these addresses may change.

1.8.5 Segment Definition

A data, extra data, or stack segment normally has the form

Segment name SEGMENT

Storage definition, allocation, and alignment directives

Segment name ENDS

and a code segment normally has the form

Segment name SEGMENT

Instructions and instruction-related directives

Segment name ENDS

The assignments of the segments to the segment registers are made with directives which are written

ASSUME Assignment, ... , Assignment

where each assignment is written

Segment register name:Segment name

The statement ASSUME CS:CODE_SEG,DS:SEG1,ES:SEG2 would inform the assembler that it is to assume that the segment address of CODE_SEG is in CS, of SEG1 is in DS, and of SEG2 is in ES. An assignment is not made for SS, presumably because either the stack is not used or the-assignment for SS is in a separate ASSUME statement.

```
DATA_SEG1   SEGMENT
            .
            .  } Directives
            .
DATA_SEG1   ENDS
DATA_SEG2   SEGMENT
            .
            .  } Directives
            .
DATA_SEG2   ENDS
CODE_SEG    SEGMENT
   START:   .
            .  } Instructions and directives
            .
CODE_SEG    ENDS
            END  START
```

Fig 1.18: Representative program structure

It is important to note that the ASSUME directive does not load the segment addresses into the corresponding segment registers. For all of the segment registers except CS, which is usually loaded by an intersegment branch when the code segment is initiated, this must be done by explicit transfers, normally by MOV instructions. An ASSUME directive makes a promise to the assembler and it must be accompanied by MOV instructions that fulfill the promise.

1.8.6 Program Termination

Just as an END statement is needed to signal the end of a high-level language program, an END directive of the form

END Label

is needed to indicate the end of a set of assembler language code. The label appears only in the termination of the main program in a program structure, although all separately assembled program modules must conclude with an END directive.

1.8.7 Alignment Directives

There are two directives that are used for alignment purposes. The directive

EVEN

forces the address of the next byte to be even

The directive

ORG Constant expression

causes the next byte to be then n+ 1th byte within a segment, where n is the value of the constant expression.

1.8.8 Value-Returning Attribute Operators

The value-returning attribute operators

LENGTH, SIZE, OFFSET, SEG, and TYPE

fall into this category. All of these operators return values related to the expressions which follow them within their operands.

The LENGTH operator returns the number of units assigned to a-variable.
The SIZE operator is the same as the LENGTH operator except that it returns the number of bytes instead of the number of units,

The OFFSET operator returns the value of the offset of a label or variable. The instruction MOV BX,OFFSET OPER_ONE
results in the offset of OPER_ONE being assembled as an immediate operand and, at execution time, the offset of OPER_ONE is loaded into BX.

The SEG operator similarly causes the segment address of the variable or label to be inserted as an immediate operand (although the actual insertion is made by the linker). If DATA_SEG is assigned to the block of memory beginning at 05000 and OPER1 is in DATA_SEG, then the instruction

MOV BX,SEG OPER1

would put 0500 in BX. The TYPE operator is used primarily with variables and structure names and returns the number of bytes associated with the variable or structure.
TYPE appears with a label, but when it does - 1 is returned for a NEAR label and - 2 for a FAR label

1.9 MODULAR PROGRAMMING

The formulation of complex programs from numerous small sequences, called program modules (or simply modules) each of which performs a well-defined task. Such formulation of computer code is referred to as modular programming. Assembler language programs are developed by essentially the same procedure as high-level language programs, that is, by:

1. Precisely stating what the program is to do.
2. Breaking the overall problem into tasks.
3. Defining exactly what each task must do and how it is to communicate with the other tasks.
4. Putting the tasks into assembler language modules and connecting the modules together to form the program.
5. Debugging and testing the program.
6. Documenting the program.

Steps 2, 3, and 4 are similar to those followed by a hardware designer, who first breaks the design into circuit modules, which may correspond to printed circuit (PC) boards, implements the modules, and then integrates them into the desired system.

The primary aid used in subdividing a program into modules is the hierarchical diagram, which is a block-oriented figure that summarizes the relationships between the modules (tasks) and submodules (subtasks). A typical hierarchical diagram is shown in Fig1.19.

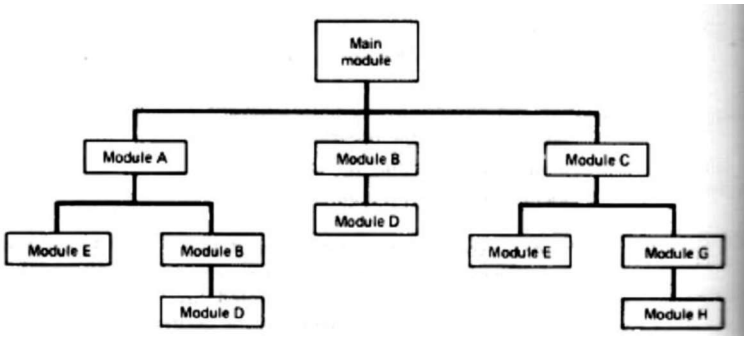


Fig 1.19: Typical hierarchical diagram

These diagrams look like, and serve the same purpose as, organizational charts for corporations. The main module corresponds to the president of the corporation, the principal submodules, A, B, and C, to the vice presidents, and so on. A hierarchical diagram shows the chain of subordination that exists among the modules and their submodules just as an organizational chart indicates the chain of command within a corporation. It helps the programmer and others to visualize the functional structure of the program. Note that a submodule may be subordinate to more than one module just as a person can appear more than once on an organizational chart.

The reasons for breaking a program into small parts are that:

1. Modules are easier to comprehend.
2. Different .modules can be assigned to different programmers.
3. Debugging and testing can be done in a more orderly fashion.
4. Documentation can be more easily understood.
5. Modifications may be localized.
6. Frequently used tasks can be programmed into modules that are stored in libraries and used by several programs.

With regard to item 1, the human mind is capable of concentrating on only a limited amount of material at one time. Therefore, it is necessary for us to subdivide our thinking no matter what kind of problem we are confronted with. Modular programming is a natural extension of this phenomenon to programming.

When designing the modules of a program one must be concerned with how and under what circumstances the modules are entered and exited, the control coupling, and how information is communicated between the modules, the data coupling. The coupling between two modules depends on several factors, including whether they are assembled together or separately and the organization of the data. In general, the task selection and module design should be such that control coupling is kept simple and data coupling is minimized.

Most assembler languages are designed to aid the modularization process in three ways. One is to allow data to be structured so that they can be readily accessed by the various modules, another is to provide for procedures (or subroutines), and the third is to permit

sections of code, known as macros, to be inserted by the appearance of single statements, each of which contains a name and a set of arguments.

1.10 LINKING AND RELOCATION

In constructing a program some program modules may be put in the same source module and assembled together; others may be in different source modules and assembled separately.

In any event, the resulting object modules, some of which may be grouped into libraries, must be linked together to form a load module before the program can be executed. In addition to outputting the load module, normally the linker prints a memory map that indicates where the linked object modules will be loaded into memory. After the load module has been created it is loaded into the memory of the computer by the loader and execution begins. Although the I/O can be performed by modules within the program, normally the I/O is done by I/O drivers that are part of the operating system. All that appears in the user's program are references to the I/O drivers that cause the operating system to execute them.

The linker is divided into two parts, one being called the linker and the other the locator, and the loading is done by the operating system. Regardless of the system, however, the linker/loader combination must make the entire segment and address assignments needed to allow the program to work properly.

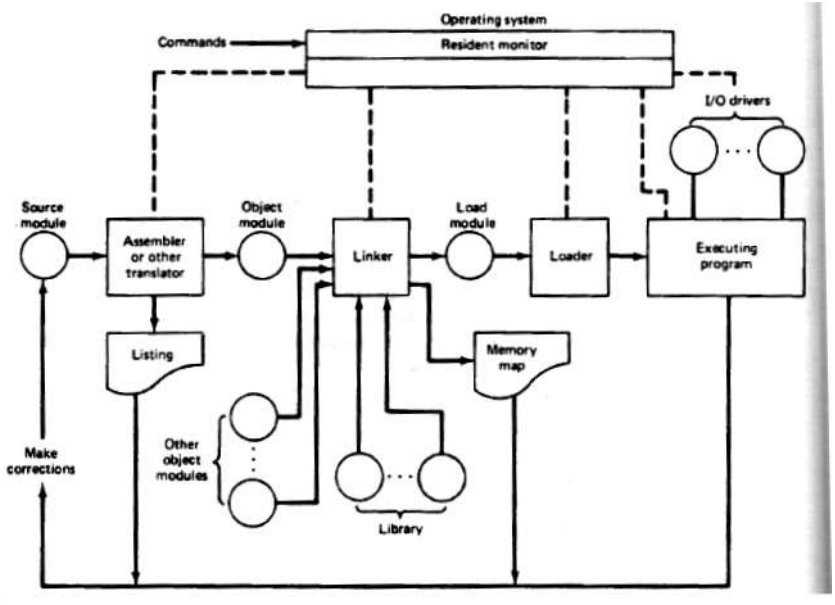


Fig 1.20: Creation and execution of a program

More specifically, this combination must:

1. Find the object modules to be linked.
2. Construct the load module by assigning the positions of all of the segments in all of the object modules being linked.

3. Fill in all offsets that could not be determined by the assembler.
4. Fill in all segment addresses.
5. Load the program for execution.

The general process for creating and executing a program is illustrated in Fig. 1.20.

The object modules to be linked are determined by naming them in the command to the linker and by the operating system searching through libraries.

The order in which the object modules appear in the linker command may determine the order in which they are stacked together to form the load module, or the arrangement of the load module may be decided by the operating system according to a set of fixed rules.

1.10.1 Segment Combination

In addition to the linker commands, the assembler provides a means of regulating the way segments in different object modules are organized by the linker. Sometimes segments with the same name are concatenated and sometimes they are overlaid. Just how the segments with the same name are joined together is determined by modifiers attached to the SEGMENT directives. A SEGMENT directive may have the form

Segment name SEGMENT Combine-type

where the combine-type indicates how the segment is to be located within the load module. Segments that have different names cannot be combined and segments with the same name but no combine-type will cause a linker error. The possible combine-types are:

PUBLIC-If segments in different object modules have the same name and the combine-type PUBLIC, then they are concatenated into a single segment in the load module. The ordering in the concatenation is specified by the linker command.

COMMON-If the segments in different object modules have the same name and the combine-type is COMMON, then they are overlaid so that they have the same beginning address. The length of the common segment is that of the longest segment being overlaid.

STACK-If segments in different object modules have the same name and the combine-type STACK, then they become one segment whose length is the sum of the lengths of the individually specified segments. In effect, they are combined to form one large stack.

AT-The AT combine-type is followed by an expression that evaluates to a constant which is to be the segment address, It allows the user to specify the exact location of the segment in memory.

MEMORY-This combine-type causes the segment to be placed at the last of the load module. If more than one segment with the MEMORY combine type is being linked, only the first one will be treated as having the MEMORY combine-type; the others will be overlaid as if they had COMMON combine types.

Figure 1.21 shows how the PUBLIC and COMMON combine-types affect the construction of a load module. By causing two or more code segments to be put in a single segment, the use of PUBLIC eliminates the need to change the contents of the CS register as the program passes between sets of instructions within the code segment, i.e., it allows intersegment branches to be replaced by intrasegment branches. Data segments can be given the PUBLIC combine-type to cause several sets of data to be combined into one larger set. The

COMMON combine-type makes it possible to create a pool of locations that can be shared by several modules.

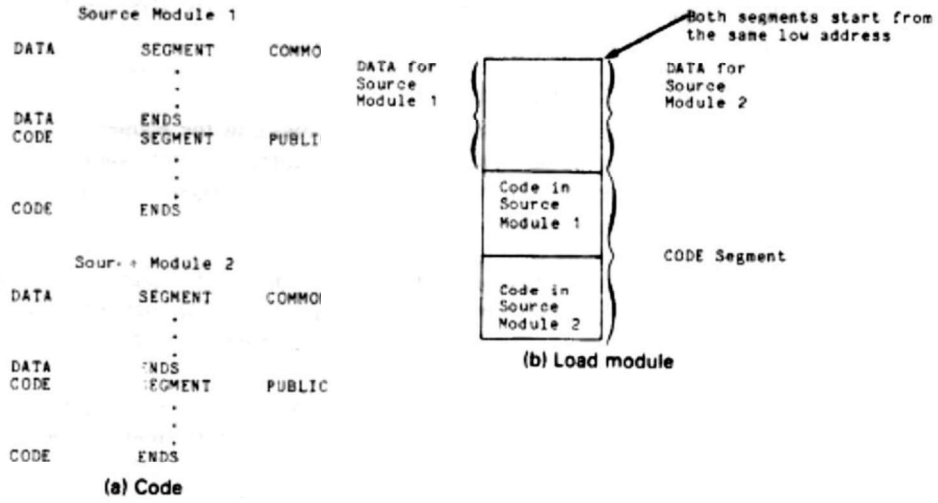


Fig: 1.21 segment combinations resulting from the PUBLIC and COMMON

STACK provides a means of obtaining one stack that can be shared by several modules while allowing each module to contribute to the overall stack length. An example for STACK modifier is provided in fig 1.22

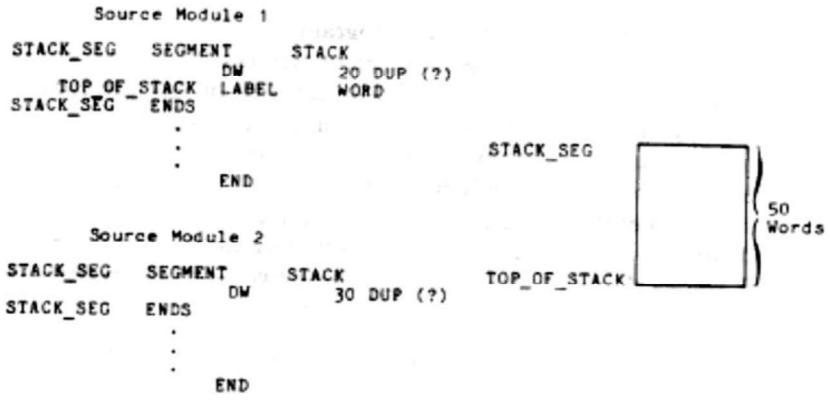


Fig 1.22: Formation of a stack from two segments

The AT combine-type permits the user to specify the exact beginning location of a segment of code or data. It is used primarily by systems programmers or someone who is writing the software for a special purpose system. The final physical addresses of the variables and labels defined in segments not given the AT combine type are set by the linker/loader process and these variables and labels are said to be relocatable. On the other hand, the addresses of the variables and labels defined in segments having the AT combine-type are set

by the programmer and assembler and are referred to as absolute variables and labels. AT is most often applied in connection with I/O handling, which involves special routines and data buffers and other memory areas that must be directly accessible by a variety of programs.

Sometimes a programmer needs to force one segment to be located in a higher part of memory than the other segments. The MEMORY combine-type can be applied to such situations.

1.10.2 Access to External Identifiers

. If an identifier is defined in an object module, then it is said to be a local (or internal) identifier relative to the module, and if it is not defined in the module but is defined in one of the other modules being linked, then it is referred to as an external (or global) identifier relative to the module.

For single-object module programs all identifiers that are referenced must belocally defined or an assembler error will occur. For multiple-module programs, the assembler must be informed in advance of any externally defined identifiers that appearing a module so that it will not treat them as being undefined.

Each module may contain two lists, one containing the external identifiers it references and one containing the locally defined identifiers that can be referred to by other modules. These two lists are implemented by the EXTRN and PUBLIC directives, which have the forms:

EXTRN Identifier:Type, ... , Identifier:Type

and PUBLIC Identifier, . . . , Identifier

where the identifiers are the variables and labels being declared as external or as being available to other modules.

If VARl is external and is associated with a word, then the module containing the statement must also contain a directive such as EXTRN ... ,VAR1 :WORD, ... and the module in which VARl is defined must contain a statement of the form PUBLIC .. .,VAR1, ...

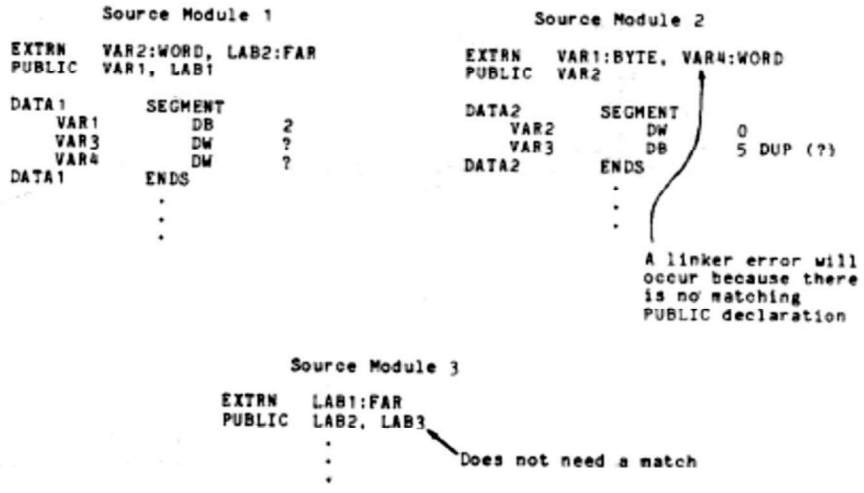


Fig 1.23: Illustration of the matching verified by the linker

One of the primary tasks of the linker is to verify that every identifier appearing in an EXTRN statement is matched by one in a PUBLIC statement. If this is not the case, then there will be an undefined external reference and a linker error will occur.

Figure 1.23 shows three modules and how the matching is done by the linker while joining them together. Relative to Module 1, VARl and LABl are local and declared public and VAR3 and VAR4 are local but not declared public. Therefore, VAR3 arid VAR4 are ignored by the linker. Module 1 refers to VAR2 and LAB2, which are not locally defined but are available to Module 1 because they have been declared PUBLIC in at least one of the other modules. Module2 contains the definition of VAR2 and Module 3 defines LAB2. Module 2 references VARl and VAR4, which are defined in Module 1, but because VAR4 does not appear in a PUBLIC statement a linker error will occur. There is no ambiguity in defining VAR3 in Module 2 as well as Module 1 because there is no cross-reference involving VAR3. LAB3, which is defined in Module 3, is made PUBLIC but does not require a match; it is simply not used by the other modules to which Module 3 is currently being linked.

1.11 STACKS

There are many situations in which a program needs to temporarily store information and then retrieve it in reverse order. One example of such a situation is saving and restoring the counters when nesting loops. On the 8086 the CX register and the loop instructions can be conveniently used to provide the counting, testing, and branching needed in a loop, but because the loop instructions are designed to use only the CX register a problem arises when loops are nested. However, if as shown in Fig. 1.24 there were an efficient means of saving the loop counters in order and then restoring them, by retrieving the last stored counter first, at least part of the problem would be alleviated.

Most often a data storage situation such as the one mentioned above is resolved by designing into the computer last-in/first-out (LIFO) stack (or push down stack) facilities, in which the stack itself is simply a part of memory. Stacks are also frequently used for other storage manipulations, but the most important applications of stacks are related to procedures.

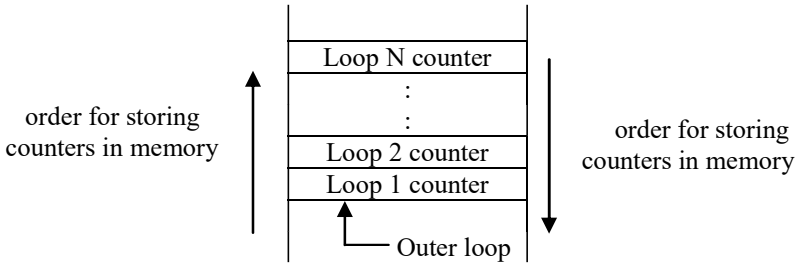


Fig:1.24 :Storing and retrieving counters for nested loops

Stack facilities normally involve the use of indirect addressing through a special register, the stack pointer, that is automatically decremented as items are put on the stack and incremented as they are retrieved. Putting something on the stack is called a push and taking it off is called a

pop. The address of the last element pushed onto the stack is known as the top of the stack (TOS).

The 8086 instructions for directly pushing and popping the stack are given in Fig. 1.25. Only words can be pushed or popped and the data cannot be immediate but the PUSH and POP instructions can use all of the other addressing modes. Only the POPF instruction, which pops the top of the stack into the PSW, affects the flags.

Stack Instructions:

Mnemonics and Format	Name	Description
PUSH SRC	Push onto stack	(SP) ← (SP)-2 ((SP) + 1: (SP)) ← (SRC)
POP DST	Pop from stack	(DST) ← ((SP) + 1: (SP)) (SP) ← (SP) + 2
PUSHF	Push the flags	(SP) ← (SP)-2 ((SP) + 1: (SP)) ← (PSW)
POPF	Pop the flags	(PSW) ← ((SP) + 1: (SP)) (SP) ← (SP) + 2

The flags are affected by the POPF instruction

Fig.1.25:Stack Instructions

As illustrated in Fig.1.26 the stack pointer register points to the top of the stack and a word is pushed on to the stack by decrementing (SP) by 2 so that it points to the next lower word in memory, and then storing the source operand into the address indicated by (SP).

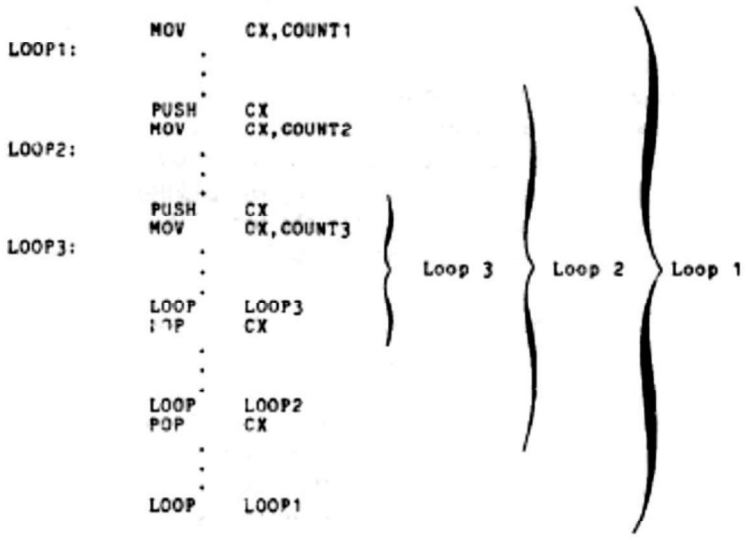


Fig 1.26 :8086 Stack

A pop consists of loading the top of the stack into the destination operand and then incrementing (SP) by 2. Thus if the stack were pushed and popped the same number of times the top of the stack would return to its original position. Figure 1.27 shows how nested loops could be implemented using 8086 instructions.

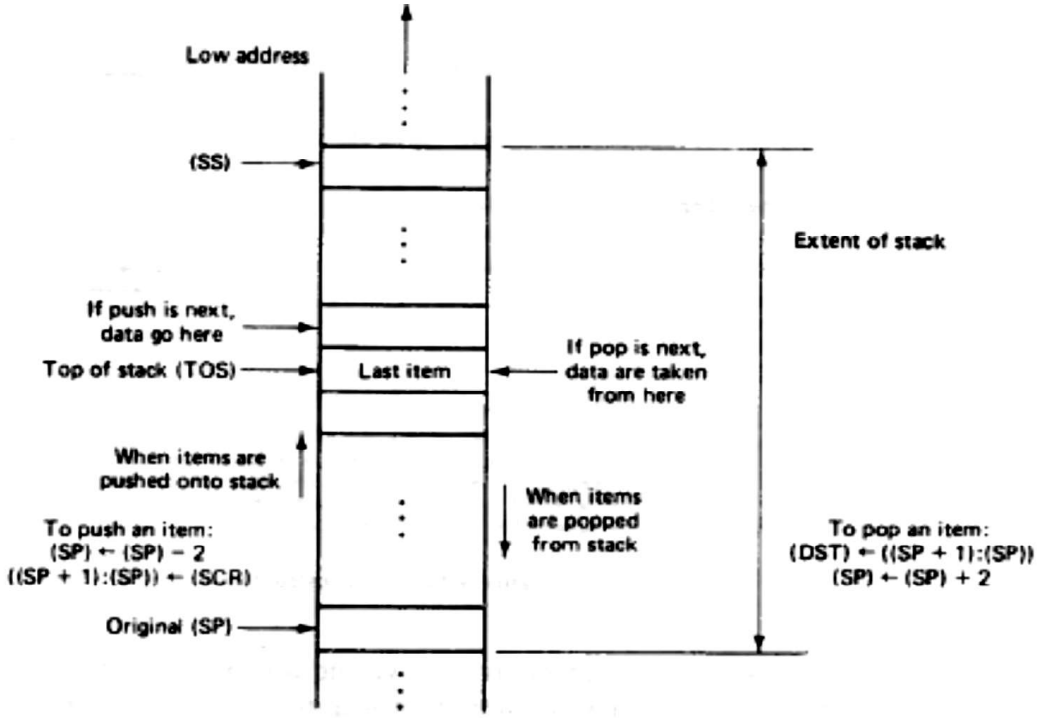


Fig 1.27: Using the stack for nested loops

On the 8086, the physical stack address is obtained from both (SP) and (SS) or (BP) and (SS), with SP being the implied stack pointer register for all push and pop operations and SS being the stack segment register. The (SS) are the lowest address in (i.e., limit of) the stack area and may be referred to as the base of the stack. The original contents of the SP are considered to be the largest offset the tack should attain.

Therefore, the stack is considered to occupy the memory locations from 16 times (SS) to 16 times (SS) plus the original (SP). The locationof the stack must, of course, be initially set by software, which may be a code sequence in either the operating system or the user's program. This is done by loading the (SS) and (SP) as shown in Fig. 1.28.

The BP register is primarily for allowing random accesses to the stack. For example,

MOV AX,[BP][SI]

will load AX from a location in the stack segment, the offset of the location being the sum of (BP) and (SI). On the other hand, for the instruction

MOV AX,[BX][SI]

the source operand is from the data segment. Stack facilities are more efficient than ordinary memory in two ways. The push and pop instructions are shorter because one of the

operands is indirectly addressed through the SP register, and the (SP) are automatically incremented or decremented to point to a new address.

```
STACK_SEG   SEGMENT
            DW      30 DUP(?)
    TOS     LABEL   WORD
STACK_SEG   ENDS
CODE_SEG    SEGMENT
  ASSUME  CS:CODE_SEG, SS:STACK_SEG
    START:  MOV     AX,STACK_SEG
            MOV     SS,AX           ;INITIALIZES SS
            MOV     SP,OFFSET TOS   ;INITIALIZES SP
              .
              .
              .
CODE_SEG    ENDS
```

Fig 1.28: Setting up a stack by loading the SS and SP registers

For example, the registers ex and DX could be stored in memory beginning at SAVE by the following instructions, which occupy a total of 8 bytes:

```
MOV   SAVE, CX
MOV   SAVE+2, DX
```

If the offset of SAVE is in SI, they could be saved by the sequence

```
MOV   [SI], CX
INC   SI
INC   SI
MOV   [SI], DX
```

which occupies only 6 bytes but contains four instructions. By saving CX and DX on the stack, only the instructions

```
PUSH  CX
PUSH  DX
```

Which require only 2 bytes of memory and can be executed very quickly, are needed: The contents of CX and DX could be retrieved by the sequence

```
POP   DX
POP   CX
```

1.12 PROCEDURES

A procedure (or subroutine) is a set of code that can be branched to and returned from in such a way that the code is as if it were inserted at the point from which it is branched to.

Procedures provide the primary means of breaking the code in a program into modules. Although not all modules are procedures, most of them are because procedures can easily be individually designed, tested, and documented. They can also be stored in libraries and used by a variety of programs. Procedures have one major disadvantage in that extra code is needed to join them together in such a way that they can communicate with each other. This extra code is referred to as *linkage.*

1.12.1 Calls, Returns, and Procedure Definitions

The branch to a procedure is referred to as the call, and the corresponding branch back is known as the return. The return is always made to the instruction immediately following the call regardless of where the call is located. If, as shown in Fig. 1.29(a), several calls are made to the same procedure, the return after each call is made to the instruction following that call.

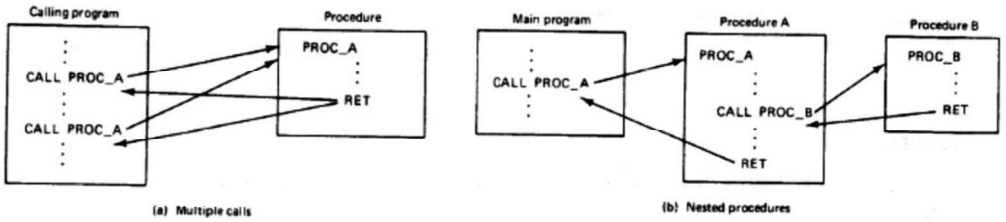


Fig 1.29: Use of procedures

Therefore, only one copy of the procedure needs to be stored in memory even though the procedure may be called several times. When procedures are nested as shown in Fig. 1.29(b), each return is made to the corresponding calling procedure, not to a procedure higher in the hierarchy.

The following three requirements must be satisfied when calling a procedure:

l. Unlike other branch instructions, a procedure call must save the address of the next instruction so that the return will be able to branch back to the proper place in the calling program.
2. The registers used by the procedure need to be stored before their contents are changed, and then restored just before the procedure is exited.
3. A procedure must have a means of communicating or sharing data with the routine that calls it and other procedures.

The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. The RET instruction simply pops the return address from the stack.

The addressing modes for the CALL instruction are the same as those for the JMP instruction except that a short CALL is not available. A CALL may be direct or indirect and intrasegment or intersegment.

If a call state is a forward reference, an attribute operator NEAR PTR or FAR PTR may be added before the operand to indicate an intrasegment direct call or an intersegment direct call, respectively.

However, if the CALL is intrasegment (intersegment), then the return must be intrasegment (intersegment), This is because an intrasegment call pushes only (IP), the offset of the return address, onto the stack, but an intersegment call must push both (IP) and (CS) onto the stack.

Call and Return instruction

Mnemonic and format	Name	Description
CALL DST	Intrasegment direct call	(SP) ← (SP)-2 ((SP) + 1: (SP)) ← (IP) (IP) ← (IP) + D16
CALL DST	Intrasegment indirect call	(SP) ← (SP)-2 ((SP) + 1: (SP)) ← (IP) (IP) ← (EA)
CALL DST	Intersegment direct call	(SP) ← (SP)-2 ((SP) + 1: (SP)) ← (CS) (SP) ← (SP)-2 ((SP) + 1: (SP)) ← (IP) (IP) ← D16 (CS) ← SEGMENT ADDRESS
CALL DST	Intersegment indirect call	(SP) ← (SP)-2 ((SP) + 1: (SP)) ← (CS) (SP) ← (SP)-2 ((SP) + 1: (SP)) ← (IP) (IP) ← (EA) (CS) ← (EA +2)
RET	Intrasegment Return	(IP) ← ((SP) + 1: (SP)) (SP) ← (SP) + 2
RET	Intersegment Return	(IP) ← ((SP) + 1: (SP)) (SP) ← (SP) + 2 (CS) ← ((SP) + 1: (SP)) (SP) ← (SP) + 2

Procedures are delimited within the source code by placing a statement of the form

Procedure name PROC Attribute

at the beginning of the procedure and by terminating the procedure with a statement

Procedure name ENDP

The procedure name is the identifier used for calling the procedure and the attribute is either NEAR or FAR. The attribute is needed to determine the type of RET statement that is to be assembled. If the attribute is NEAR, the RET instruction will only pop a word into the IP register, but if it is FAR, it will also pop a word into the CS register.

A procedure may be in:

1. The same code segment as the statement that calls it.
2. A code segment that is different from the one containing the statement that calls it,

but in the same source module as the calling statement.

3. A different source module and segment from the calling statement.

Figure 1.30 shows a situation in which procedure SUBT is defined in code segment SEGX and is called from both code segments SEGX and SEGY. Because SUBT is called from both SEGX and SEGY it must be given the FAR attribute. Since SUBT has been given the FAR attribute, the calls from within SEGX must push both (IP) and (CS) onto the stack as well as the call from SEGY. Since SUBT is defined before it is called, the assembler could actually implicitly translate the calls properly without using the FAR PTR attribute operator, but for clarity it is suggested that the attribute operator be included.

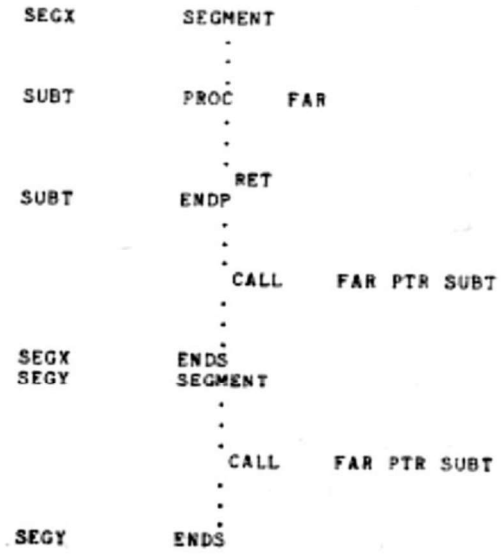


Fig 1.30: Procedure declaration

1.12.2 Saving and Restoring Registers

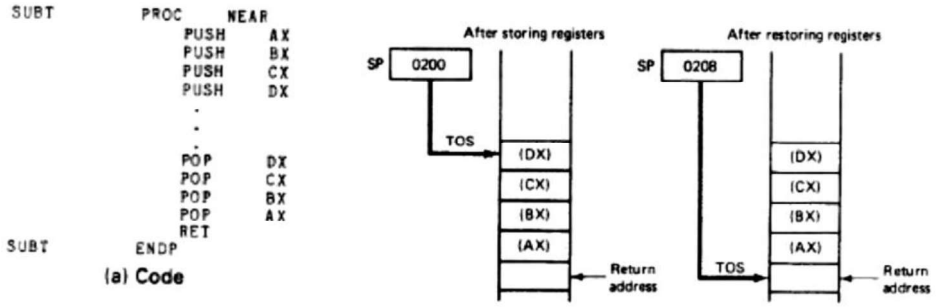


Fig 1.31: Storing and restoring the registers

Since both calling program and procedure share the same set of registers, it is necessary to save the registers when entering a procedure, and to restore them before returning to the calling program. Saving and retrieving the contents of the registers is

important because procedure will change its register value. Stack is used for this purpose. Figure 1.31(a) gives the code needed to store and restore the contents of the AX, BX, CX, and DX register and Fig. 1.31(b) shows how the stack and stack pointer are affected.

The registers that are altered by the procedure should be saved. DX register is not saved because it is not accessed, but later the procedure is modified so that DX is used, then PUSH DX and POP DX would have to be properly inserted in the procedure.

1.12.3 Procedure Communication

There are two general types of procedures,

- Procedure that always operate on the same set of data and
- Those that may process a different set of data each time they are called.

For example, a procedure may be written so that it always adds the first COUNT elements in an array ARY and returns the result m a variable SUM, or it may be written so that it can add the elements in an arbitrary array and return the results to an arbitrary variable.

When the procedure is in a separate source module it can still refer to the variables directly, provided that the calling program contains the directive

```
PUBLIC      ARY, COUNT.SUM
```

and the procedure source module contains the directive

```
EXTRN       ARY:WORD, COUNT:WORD, SUM:WORD
```

ARY, COUNT, and SUM could be put in a common area by placing the code

```
DATA        SEGMENT     COMMON
            ARY         DW          100 DUP(?)
            COUNT       DW          ?
            SUM         DW          ?
DATA        ENDS
```

at the beginning of both source modules. This would cause both source modules to share the segment DATA. If the calling program uses different variable names, say NUM for ARY, N for COUNT, and TOTAL for SUM, then the DATA segment in its source module would need to be defined as follows:

```
DATA        SEGMENT     COMMON
            NUM         DW          100 DUP(?)
            N           DW          ?
            TOTAL       DW          ?
DATA        ENDS
```

In either case the segments DATA would be overlaid by the linking process and the first 100 words of DATA would contain the numbers being summed, the next word would contain the count, and the last word would hold the result.

Procedures are communicated by passing a message using parameters are variable. The variables whose addresses are passed are called *parameters.* There are two principal ways of passing parameter addresses:

(1) to construct a parameter address table (or array) and pass the address of the table via a register, and
(2) to push the parameter addresses onto the stack.

1.12.4 Recursive Procedures

Recursive is a method used to execute a same task again and again until certain conditions gets failed. It is needed in systems programming for performing searches, decoding the syntax of a high level language and executing other tasks.

As an example, Homer's rule for evaluating the polynomial

$((\ldots ((a_n x + a_{n-1})x + a_{n-2})x + \ldots)x + a_1)x + a_0$

involves multiplying x and $a_n$adding a_{n-1} to the result, multiplying x times this result, adding a_{n-2} to this result, and so on. The next result is obtained by multiplying the previous result by x and adding the next lower coefficient. A single application of the algorithm consists of a multiplication and an addition and the overall result could be obtained by nesting the simple multiply-add algorithm within itself. Such algorithms are said to be recursive and are sometimes needed in systems programming for performing searches, decoding the syntax of a high-level language, and executing other tasks.

Recursive algorithms may be implemented by having a procedure call itself, but care must be taken to assure that each successive call does not destroy the parameters and results generated by the previous call and to make sure that the procedure does not modify itself.

This means that each call must store its set of parameters, registers, and all temporary results in a different place in memory. To guarantee separate areas of memory stack is normally employed. By pushing the data onto the stack as shown in Fig. 1.32, it can be conveniently retrieved in reverse order as the process emerges from its nesting.

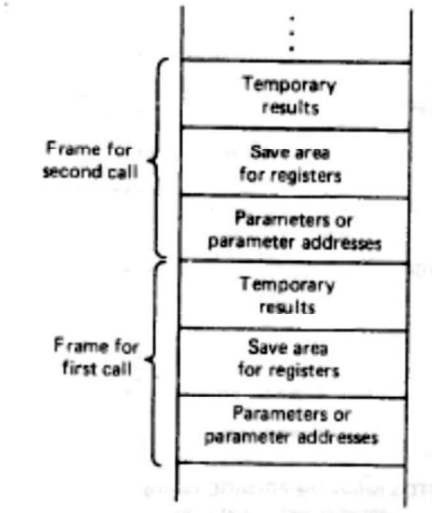


Fig 1.32: Use of stack to dynamically provide storage during recursive calls

The data stored by an application of the procedure is called a frame, and the 8086's BP register can be used to permit ready access to the individual items within a frame. Normally, a frame contains the saved register contents, the parameter addresses, and a temporary storage area. Properly storing the information on the stack provides for an orderly exit. To prevent

indefinite nesting of a recursive algorithm, clearly the algorithm must include a conditional branch that will eventually allow the nest to be exited.

1.13 MACROS

MACROS and procedures are used for same purpose, when compared to procedure macros have number of advantages. So macros are mainly used in any assembly language program.

***Advantages of Procedure*:**

- They save memory and programming time by allowing code to be reused
- It provides a modularity that makes it easier to debug and modify a program.

***Disadvantages of Procedure*:**

- It sometimes requires more code to program the linkage than is needed to perform the task.
- The execution time is increased.

A macro is a segment of code that needs to be written only once but whose basic structure can be caused to be repeated several times within a source module by placing a single statement at the point of each appearance. A macro is unlike a procedure in that the machine instructions are repeated each time the macro is referenced; therefore, no memory is saved, but programming time is conserved (no linkage is required) and some degree of modularity is achieved.

The code that is to be repeated is called the prototype code, and the prototype code along with the statements for referencing and terminating it is called the macro definition. Included in the first statement of a macro definition is the macro's name. The procedure for using a macro is to give the macro definition and then cause the macro to be inserted at various points within a program by placing a statement that includes ,the macro's name at these points. These statements are known as macro calls. Then a macro call is encountered by the assembler, the assembler replaces the call with the macro's code. This replacement action is referred to as a macro expansion.

1.13.1 Defining a MACRO:

A macro can be defined anywhere in a program using the directives MACRO and ENDM. The label prior to MACRO is the macro name which is used in the actual program. The ENDM directive means the end of the instructions or statements sequence assigned with th macro name.

A macro definition is constructed as follows:

```
%*DEFINE (Macro name (Dummy parameter list))
(       :
        :       Prototype code
        :
)
```

where the macro name (which must be an identifier that begins with a letter and contains only letters, numbers, and underscore characters) is used to reference the macro, and the dummy parameters in the parameter list are separated by commas. Each dummy parameter appearing in

the prototype code be preceded by a % character.

A macro call has the form % Macro name (Actual parameter list) with the actual parameters being separated by commas.

Example of the use of a macro

```
%*DEFINE (MULTIPLY (OPR1, OPR2, RESULT))
(       PUSH DX
        PUSH AX
        MOV AX, OPR1
        IMUL OPR2
        MOV RESULT, AX
        POP AX
        POP DX
)       :
        :
        :                                  PUSH DX
                                           PUSH AX
                                           MOV AX, CX
% (MULTIPLY (CX, VAR, XYZ(BX))      }       IMUL VAR
                                           MOV XYZ(BX), AX
                                           POP AX
                                           POP DX
```

Fig 1.33: Example of macro

1.13.2 Local Labels

A macro definition has to include labels. For example, consider a macro ABSOL, which replaces an operand by its absolute value. This macro might be defined as

```
%*DEFINE (ABSOL (OPER))
(       CMP %OPER,0
        JGE NEXT
        NEG %OPER
NEXT: NOP
)
```

After the macro ABSOL is called for the first time, the label NEXT will appear in the program and, therefore, it becomes defined. Any subsequent call for ABSOL will cause NEXT to again be defined. This will generate an error during the assembly process because NEXT has been associated with more than one location.

One solution to this problem would be to have NEXT replaced by a dummy parameter. Then each call for ABSOL could use a different actual parameter for the label.

To avoid the problems associated with including the label in the parameter list, some assemblers permit the designation of special labels, called local labels, that have suffixes that increment each time the macros are called.

These suffixes are two-digit numbers that increment by 1 starting from zero; thus a different label is generated for each expansion.

Labels are declared to be local by attaching

```
LOCAL        List of Local Labels
```

to the end of the first statement in the macro definition. In the above example, ABSOL could be redefined as follows:

```
%*DEFINE     (ABSOL (OPER)) LOCAL NEXT
              CMP %OPER,O
             JGE %NEXT
             NEG %OPER
%NEXT        : NOP
)
```

Assuming this definition of ABSOL, the first two calls, %ABSOL (VAR) and %ABSOL (BX), would result in the following expansions:

```
             CMP    VAR,0
             JGE    NEXT00
             NEG    VAR
NEXT00:      NOP
             :
             :
             CMP BX,0
             JGE NEXT01
             NEG BX
NEXT01:      NOP
```

1.13.3 Nested Macros

It is possible for a macro call to appear within a macro definition. This is referred to as macro nesting and has the limitation that all macros included in the definition of a given macro must be defined before the given macro is called. Any macro, assuming that its dummy parameters are properly defined and used, can be called directly or from within one or more other macros. An example of macro nesting is shown below:

```
%*DEFINE (DIFSQR (OPR1, OPR2, RESULT))
(            PUSH DX
             PUSH AX
%DIF (OPR1, OPR2))
             IMUL AX
             MOV R4ESULT, AX
             POP AX
             POP DX
)
%*DEFINE (DIF (X,Y))
             MOV AX, X
             SUB AX,Y
```

```
        :
        :
%DIFSQR (VAR1, VAR2, ERROR)
        :
        :
```

```
PUSH DX
PUSH AX
MOV AX, VAR1
SUB AX, VAR2
IMUL AX
MOV ERROR, AX
POP AX
POP DX
```

In this example the outer macro DIFSQR is for computing

$$(OPR1-OPR2)^2$$

and it uses the macro dif for computing the difference. It is also possible to include macro definitions within macro definitions.

1.13.4 Controlled Expansion and Other Functions

Format	Description
% IF (expr) THEN (: } Code block 1) ELSE (: } Code block 2) FI	If expr evaluates to an odd number or true, code block 1 is included in the expansion: otherwise code block 2 is included
% IF (expr) THEN (: } Code block) FI	If expr evaluates to an odd number or true, code block 1 is included in the expansion: otherwise code block 2 is included
% REPEAT (expr) (: } Code block)	The code block is duplicated "expr" times
% WHILE (expr) (: } Code block) % EXIT	The code block is repeated until expr evaluates to an even number or FALSE

The capability of being able to select the code that is to be assembled is called controlled expansion (or conditional assembly) and for the assembler this capability is built into the preprocessor.

To prevent the code block associated with a %WHILE control function from being expanded indefinitely it must contain statements that change the value of the expression in the

%WHILE statement. This is normally done by using the %SET function to change the value of a name that appears in the expression in the %WHILE statement.

Format	Description
%SET (Name, Value)	The name is associated with the value
%LEN (Character string)	If LEN is part of the expression in a IF, REPEAT or WHILE function, then the length of the string is returned as a number; otherwise the returned value is the ASCII representation.

1.14 INTERRUPTS AND INTERRUPT ROUTINES

When microprocessor is executing the normal program if any interrupt arises, microprocessor executes the current instruction execution and saves the address and current status in the stack and then executes the interrupt service routine.

The program that is executed to service the interrupt is called interrupt service routine. Instructions that initiate interrupts are called software interrupts. There are two general classes of interrupts. There are:

1. Internal interrupts that are initiated by the state of the CPU or by an instruction
2. External interrupts that are caused by a signal being sent to the CPU from elsewhere in the computer system.

An interrupt routine is similar to a procedure in that it may be branched to from any other program and a return branch is made to that program after the interrupt routine has executed. The interrupt routine must be written so that, except for the lapse in time, the interrupted program will proceed just as if nothing had happened. This means that the PSW and the registers used by the routine must be saved and restored and the return must be made to the instruction following the last instruction executed before the interrupt. An interrupt routine is unlike a procedure in that, instead of being linked to a particular program, it is sometimes put in a fixed place (absolutely located) in memory. Because it is not linked to other segments, it can use only common areas that are absolutely located to communicate with other programs. Because some kinds of interrupts are initiated by external events, they occur at random points in the interrupted program. For such interrupts, no parameters or parameter addresses can be passed to the interrupt routine. Instead, data communication must be made through variables that are directly accessible by both routines.

Regardless of the type of interrupt, the actions that result from an interrupt are the same and are known as the interrupt sequence. The interrupt sequence for the 8086 is shown in Fig. 1.34.

Some kinds of interrupts are controlled by the IF and TF flags and in those cases these flags must be properly set or the interrupt action is blocked. If the conditions for an interrupt are met and the necessary flags are set, the instruction that is currently executing is completed and the interrupt sequence proceeds by pushing the current contents of the PSW, CS, and IP onto the stack, inputting the new contents of IP and CS from a double word whose address is determined by the type of interrupt, and clearing the IF and TF flags.

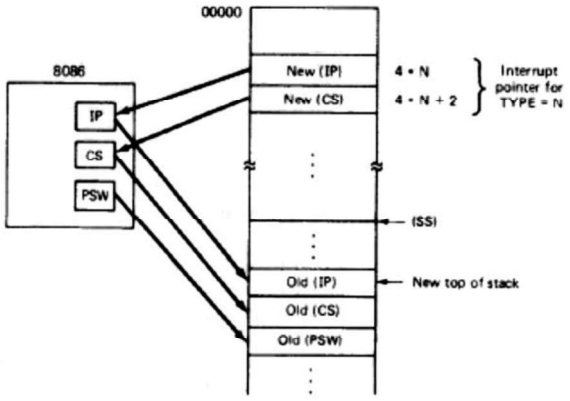


Fig 1.34: Interrupt sequence

The new contents of the IP and CS determine the beginning address of the interrupt routine to be executed. After the interrupt has been executed the return is made to the interrupted program by an instruction that pops the IP, CS, and PSW from the stack.

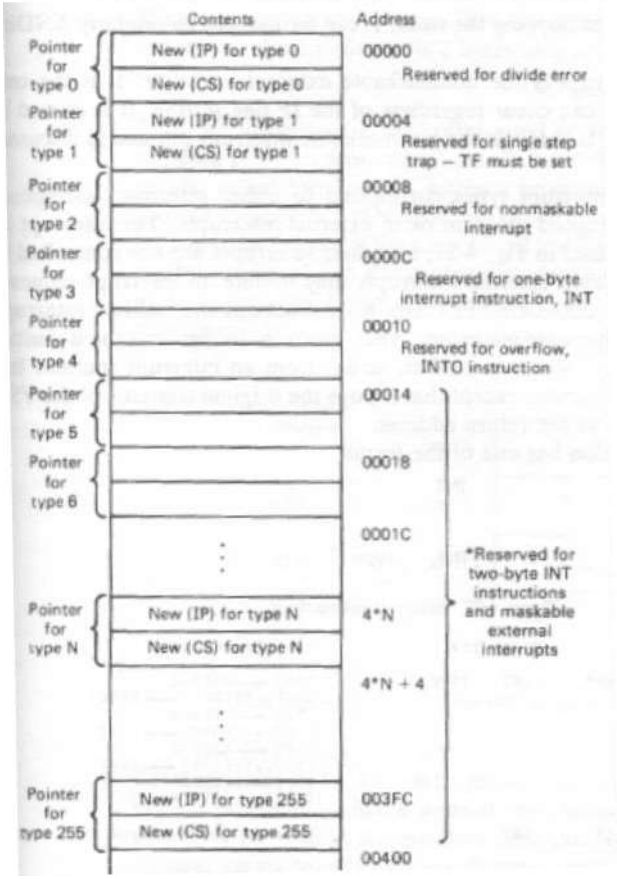


Fig 1.35: Layout of interrupt pointers

The double word containing the new contents of IP and CS is called an interrupt pointer (or vector). Each interrupt type is given a number between 0 and 255, inclusive, tnd the address of the interrupt pointer is found by multiplying the type by 4. If the type is 9, then the interrupt pointer will be in bytes 00024 through 00027. Since it takes 4 bytes to store a double word, the interrupt pointers may occupy the first 1024 bytes of memory and these bytes should never be used for other purposes. Some of the 256 interrupt types may be reserved by the operating system and may be initialized when the computer system is first brought up. Users may tailor the other interrupt types according to their particular applications.

The kinds of interrupts and their designated types are summarized in Fig. 1.35 by illustrating the layout of their pointers within memory.

Only the first five types have explicit definitions; the other types may be used by interrupt instructions or external interrupts.

Type 0: Division error

During the execution of a program, if the quotient from either DIV or IDIV instruction is too large to fit in the quotient register, 8086 will generate the Divide-by-zero interrupt request. Therefore, if a division by 0 is attempted, the computer will push the current contents of the PSW, CS, and IP onto the stack, fill the IP and CS registers from addresses 00000 and 00002, and continue executing at the address indicated by the new contents of IP and CS

Type 1: Single step trap

When Trap flag is set, 8086 executes the single step interrupt after the execution of each instruction. This causes a branch to the location indicated by the contents of 00004 through 00007. The single step trap is used for debugging. This gives the programmer a snap shot of his program after each instruction is executed.

TF flag can be enabled by pushing the PSW onto the stack, ORing the top of the stack with 0 1 0 0 , and then popping the stack. It can be disabled by similarly ANDing PSW with FEFF.

Type 2: Non maskable interrupt

The type 2 interrupt is the non maskable external interrupt. It is the only external interrupt that can occur regardless of the IF flag setting. It is caused by a signal sent to the CPU through the non maskable interrupt pin.

Type 3: INT

The INT instruction has one of the forms

INT or INT Type

If the type does not appear, then the instruction is 1 byte long and has type 3: otherwise, the instruction is 2 bytes long and the interrupt pointer begins at 4* Type. The INT instruction is lso used as a debugging aid in cases where single-stepping provides more detail than it is wanted. By inserting INT instructions at key points, called breakpoints, within a program a programmer can use an interrupt routine to print out messages and other information at these points. If the information to be printed is same for all the breakpoints, then the 1-byte INT instruction can be employed, but the information is to vary, then a type needs to be included in the instruction. Fig 1.36 shows how the 2-byte INT instruction can be applied for debugging.

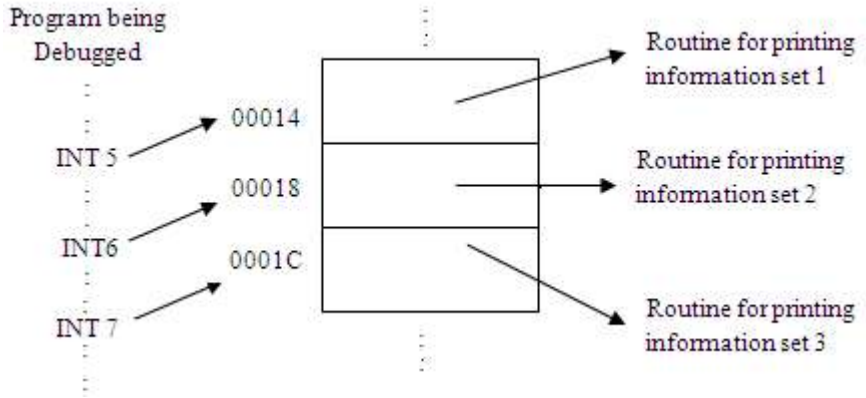


Fig 1.36 : Use of INT instruction for debugging a program

Type 4: INTO

The INTO instruction has type 4 and causes an interrupt if and only if the OF flag is set to 1. It is often placed just after an arithmetic instruction so that special processing will be done if the instruction causes an overflow. Unlike a divide-by-zero fault, an overflow does not cause an interrupt automatically; the interrupt must be explicitly specified by the INTO instruction.

1.15 BYTE AND STRING MANIPULATION

One very important computer application is text processing, which is the manipulation of sequences of bytes that contain the alphanumeric codes for characters, i.e., character strings. String is a collection of text used to represent character based information. The string oriented instructions performs any operation in a time consumed manner.

1.15.1 String Instructions

The string instructions are summarized below.

Mnemonic and format	Name	Description
MOVSB MOVSW	Move byte string Move word string	((DI)) ← ((SI)) Byte operands (SI) ← (SI) ± 1, (DI) ← (DI) ± 1 Word operands (SI) ← (SI) ± 2, (DI) ← (DI) ± 2
CMPS SRC, DST	Compare string	((SI)) – ((DI)) Byte operands (SI) ← (SI) ± 1, (DI) ← (DI) ± 1 Word operands (SI) ← (SI) ± 2, (DI) ← (DI) ± 2
SCASB SCASW	Scan byte string Scan word string	Byte operand (AL) – ((DI)), (DI) ← (DI) ± 1 Word operand (AX) – ((DI)), (DI) ← (DI) ± 2

LODSB LODSW	Load byte string Load word string	Byte operand (AL) – ((SI)), (SI) ← (SI) ± 1 Word operand (AX) – ((SI)), (SI) ← (SI) ± 2
STOSB STOSW	Store byte string Store word string	Byte operand ((DI)) ← (AL), (DI) ← (DI) ± 1 Word operand ((DI)) ← (AX), (DI) ← (DI) ± 2

Because the string instructions can operate on only a single byte or word unless they are used with the REP prefix. They are often referred to as string primitives, or simply primitives. All of the primitives are 1 byte long, with bit 0 indicating whether a byte (bit 0= 0) or a word (bit 0=1) is being manipulated.

There are five basic primitives and each may appear in one of the following three forms:

	Operation	Operand(s)
or	OperationB	
or	OperationW	

If the first form is used, whether bytes or words are to be operated on is determined implicitly by the type of the operand(s). The second and third forms explicitly indicate byte and word operations, respectively.

For a source operand the address of the operand is the sum of (SI) and (DS) x 16_{10}, and for the destination operand the address is always the sum of (DI) and (ES) x 16_{10}.

SI and/or DI to be automatically incremented or decremented. Auto-incrementing or auto-decrementing is used is dictated by the DF flag in the PSW. If DF = 1, the index registers are auto-decremented, and if DF = 0.

When working with strings, the advantages of the MOVS and CMPS instructions over the MOV and CMP instructions are:

1. They are only 1 byte long.
2. Both operands are memory operands.
3. Auto-indexing operations will decreases overall processing time.

MOVS instruction:

MOVS instruction is used to move the content of bytes or words from one string to another string. Here both source operand and destination operand can be a memory operand.

```
        MOV     OFFSET   STRING1
        MOV     OFFSET   STRING2
        MOV     LENGTH   STRING1
        CLD
MOV:    MOVS    STRING2, STRING1
        LOOP    MOVE     MOVE
```

Above program shows moving a block of data from one string to another using MOVS instruction.

CMPS instruction:

The CMPS primitive can be used to compare strings of bytes or words of arbitrary length. If two strings are equal, then it return SAME otherwise continues at EXIT.

The other three string primitives, SCAS, LODS, and STOS, have single memory operands, only SCAS affects the condition flags.

1.15.2 Rep Prefix

Generally string operations are involved in looping process, the 8086 machine language includes a prefix that considerably simplifies the use of string primitives with loops. This prefix has the machine code:

1	1	1	1	0	0	1	Z

where, for the CMPS and SCAS primitives, the Z bit helps control the loop. By prefixing MOVS, LODS, and STOS, which do not affect the flags, with the REP prefix 11110011,they are repeated the number of times indicated by the ex register according to the following steps:

1. If (CX) = 0, exit the REP operation.
2. Perform the specified primitive.
3. Decrement CX by 1.
4. Repeat steps 1 through 3.

For the CMPS and SCAS primitives, which do affect the flags, the prefix causes them to be repeated the number of times indicated by the CX register or until the Z-bit does not match the ZF flag.

In assembler language the prefix is invoked by placing the (REP) mnemonic before the primitive. The REP mnemonics are defined as follows:

Mnemonic and format	Name	Termination condition
REP string primitive	Repeat string operation until CX=0	(CX) = 0
REPE string primitive or REPZ	Repeat string operation while equal or zero	(CX) = 0 or (ZF) = 0
REPNE string primitive or REPNZ	Repeat string operation while not equal or not zero	(CX) = 0 or (ZF) = 1

1.15.3 Text Editor Example

In the design of a text editor or text formatting program it is necessary to have program sequences for moving and comparing character strings, and for inserting strings into and deleting them from other strings. It is often necessary to search a string for a given substring or to replace a substring with a different substring. A text editor is a requirement on any general purpose computer system, and features which improve the ability of such a system to work with character strings can be a definite advantage. A flowchart of the EDITOR module is shown in Fig. 1.37.

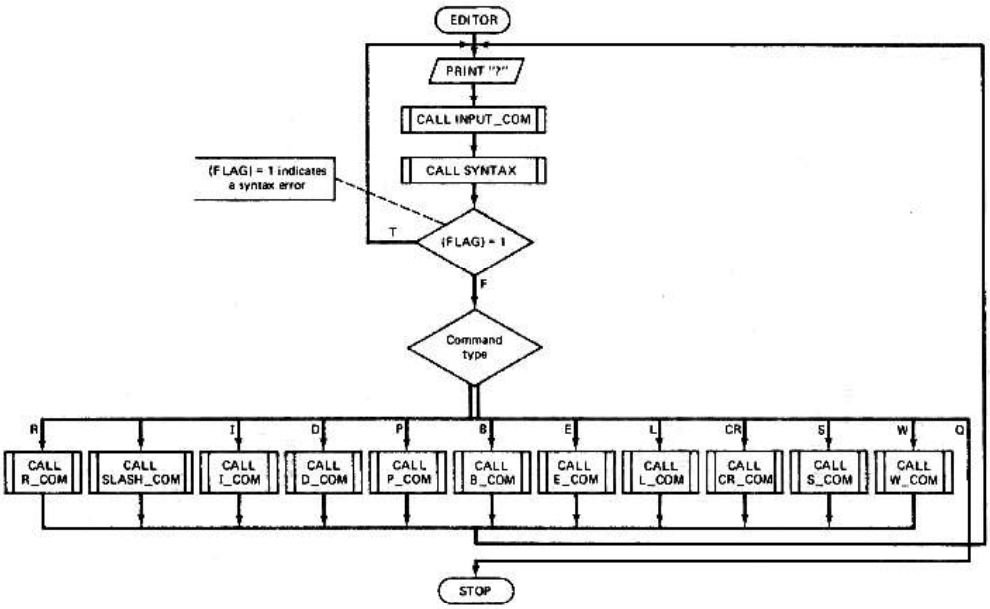


Fig 1.37: Flowchart of the editor module

R Filename - Reads the file Filename into the linked list (i.e., text buffer) in memory. The file is to have the linked list structure indicated above and the text buffer is to be a common area named TEXT_BUFF. After this command is executed, the cursor is to point to the first line of text.

/ String-Searches the text beginning with the current line until a substring that exactly matches String is found. The cursor is changed to point to this line and the line is printed. If String is not found, then the cursor is to return to its original value and NOT FOUND is to be printed by the subprogram MESSAGE.

I String- If the length of String is less than or equal to 80, this command is to cause String, concatenated with a carriage return/line feed combination, to be put in a line immediately following the current line. The cursor is to point to the new line and the .new line is to be printed. If the length of String is greater than 80, the command is to be ignored.

D- If the current line is not the first element in the list, this command causes the current line to be deleted, the cursor to point to the next line, and the next line is to be printed; otherwise, the command is ignored.

P-Causes the current line to be printed unless the current line is the first element in the list, in which case the command is ignored.

B-Causes the cursor to point to the first element in the list.

E- Causes the cursor to point to the last element in the list.

L-Causes all lines to be printed and the cursor to point to the last line.

(Carriage Return) -Causes the cursor to move to the next line and the next line to be printed.

S /String 1/String 2-If the length of String 2 minus the length of String 1 plus the length of the current line is less than or equal to 82, this command causes the current line to be searched for String 1 and, if it is found, to replace String 1 by String 2. Also, the changed line is to be printed. If the resulting line is too long or String 1 is not found, then no action is to be taken. In any case the cursor is to be left unchanged.

W Filename- Causes the linked list to be copied into an array so that the lines are in their proper order, and then outputs the ordered array to the file Filename.

Q- Causes the program to terminate.

A blank is to appear after the command symbol in those commands that contain a filename or string. The SYNTAX module is to check the syntax of the commands. If the syntax is correct and the command symbol is I, /, or S, SYNTAX is to call the module LENGTH to determine the length(s) of the string(s) involved in the command. The ORGANIZE module is to copy the linked list into an ordered array before it is written into a file by W_COM.

1.15.4 Table Translation

It is sometimes necessary to translate from one code to another. A terminal may communicate with the computer using the EBCDIC alphanumeric code even though the computer's software is designed to work with the ASCII code, or vice versa.

Code conversions involving fewer than 8 bits (which accommodates up to 256 distinct entities) can be performed most easily by storing the desired code in an array of up to 256 bytes and letting the original code be the index within the array of the desired code values. If the EBCDIC code were being converted to the ASCII code, then the EBCDIC code value for "A", which is 11000001,would be added to the address of the beginning of the array. Then, by putting the ASCII code for A, which is 01000001, in the array element having the address of the array plus 00Cl, the code conversion is readily accomplished.

The 8086 has an instruction specifically designed for executing this procedure. It is the XLAT instruction and is defined below:

Mnemonic and format	Name	Description
XLAT OPR XLATB	Translate	(AL) ← ((BX) + (AL))

1.15.5 Number Format Conversions

Terminals, printers, card readers, and other devices communicate with a computer system through a code, often the ASCII code. However, the internal arithmetic operations are not normally done with numbers in their coded formats, but usually operate on numbers in their binary or packed BCD formats. Therefore, it is necessary to convert from the external form of the data which the I/O equipment is familiar with, to the internal form, which can easily be operated on by the machine language instructions of the computer.

Although the 8086 is capable of performing arithmetic on ASCII-coded numbers directly. Arithmetic using the packed BCD format is acceptable if only addition and subtraction are involved. Binary arithmetic is, of course, the easiest and fastest of the three choices. Unfortunately, to use binary arithmetic all numbers must be converted from their ASCII form to their binary form.

Therefore, there is a trade-off between taking the extra time to perform the conversions to and from binary, and using more time in carrying out the arithmetic operations. The packed BCD format is a good compromise for some problems since the conversion from ASCII to packed BCD is relatively simple and addition and subtraction on packed BCD numbers can be performed fairly quickly. The decision as to which format to use is most often arrived at after weighing the amount of I/O against the complexity of the arithmetic.

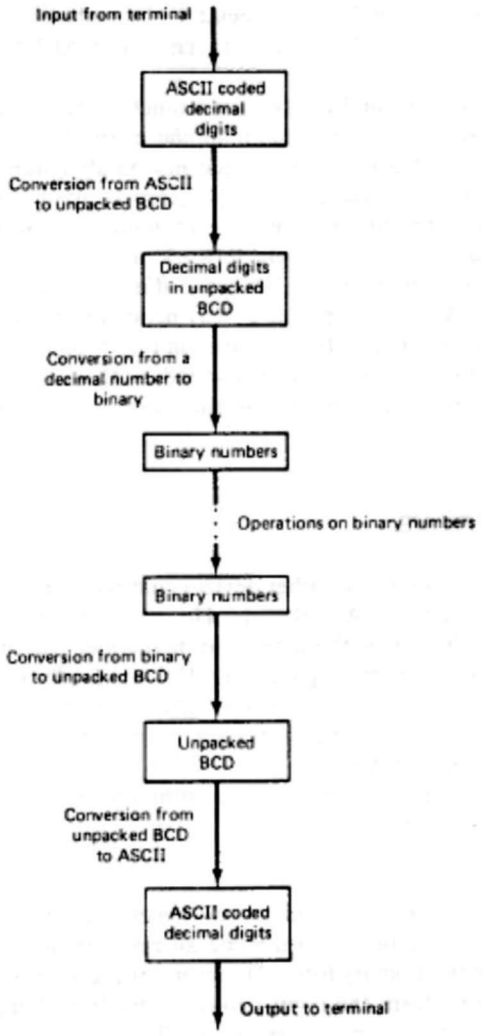


Fig 1.38: Conversion process required for inputting or outputting decimal numbers

The entire process of inputting ASCII character strings, converting them to binary numbers, performing the necessary arithmetic, and reconverting the results to ASCII strings is shown in Fig. 1.38.

The conversion to binary is a two-step process. First each ASCII character is converted into an unpacked BCD form with the four MSBs set to zeros, and then Homer's rule is applied to the string of unpacked digits to produce the binary number.
If the ASCII string is

$$d_n \ldots d_1 d_0$$

where $d_n \ldots d_1 d_0$ represent the decimal digits, then the binary number can be found by evaluating

$$((\ldots ((10d_n + d_{n-1})10 + d_{n-2}) \ldots)10 + d_1)10 + d_0$$

LIST OF QUESTIONS
PART A

1. What is microprocessor?

Microprocessor is a multipurpose programmable clock driven register based electronic device that fetches the instruction from the storage device called memory accepts the input from the user process the instruction and produce the result as output.

2. List the advantages of microprocessor.

- It simplifies system design.
- It reduces development time
- It reduces cost and size
- It has flexible operation

3. List few applications of microprocessor.

- It is used for speed control of machines.
- Used for traffic control and industrial tool control.

4. How many memory locations can be accessed by 11 address lines in 8086 processor?

The number of memory locations that can be accessed is 2^{11}

5. What are the advantages of using segment registers?

The advantages of using segment registers are that they:

1. Allow the memory capacity to be 1 megabyte even though the addresses associated with the individual instructions are only 16 bits wide.
2. Allow the instruction, data, or stack portion of a program to be more than 64K bytes long by using more than one code, data, or stack segment.
3. Facilitate the use of separate memory areas for a program, its data, and the stack.
4. Permit a program and/or its data to be put into different areas of memory each time the program is executed.