Catalog

INTRODUCTION

A microprocessor is a computer processor which incorporates the functions of a computer's Central Processing Unit (CPU) on a single Integrated Circuit (IC), or at most a few integrated circuits. The microprocessor is a multipurpose, clock driven, register based, programmable electronic device which accepts digital or binary data as input, processes it according to instructions stored in its memory, and provides results as output. Microprocessors contain both combinational logic and sequential digital logic.

- Microprocessors operate on numbers and symbols represented in the binary numeral system Microprocessor is the controlling unit or CPU of a micro-computer, fabricated on a very small chip capable or performing ALU operations and communicating with the external word connected to it. It forms a micro-computer when combined with memory and Input/output devices.
- Microprocessors of different word size with varying decrease of capabilities are available. Microprocessor comprises of all the functional components of the central processing unit of a general purpose computer. In other words, functionally it is equivalent to a CPU.
- ***Cost:*** The most important characteristics of a microcomputer is its low cost. Because of the widespread use of microprocessors, the volume of production is very high. That is why, microprocessor chips are available at fairly low prices.
- ***Size:*** The second important features of a microprocessor is its small size. As a result of improvement in fabrication technology, VLSI, electronic circuitry has become so dense that a minute silicon chip can contain hundred and thousands of transistors constituting the microprocessor. Its size does not exceed a few inches on any side, even in the packaged form.
- ***Power Consumption:*** The important characteristics are its low power consumption microprocessors are normally manufactured by Metal-Oxide semiconductor technology.
- ***Versatility:*** The versatility of a microprocessor results from its stored program mode of operation. Keeping the same basic hardware, a microprocessor-based system can be

configured for a number of applications simplify altering the software program. This also makes it very flexible.

- ***Reliability:*** Another important property of VLSI devices which has also been in herniated by microprocessors is extreme reliability. It has been established that the failure rate of an IC is fairly uniform at the package level, regardless of its complexity.

THE 8086 MICROPROCESSOR

The 8086 is a 16-bit microprocessor chip designed by Intel between early 1976 and mid-1978, when it was released. The Intel 8088, released in 1979, was a slightly modified chip with an external 8-bit data bus (allowing the use of cheaper and fewer supporting ICs, and is notable as the processor used in the original IBM PC design, The 8086 gave rise to the x86 architecture which eventually turned out as Intel's most successful line of processors.

Features of 8086 Microprocessor:

- The 8086 is a 16-bit microprocessor which means its data handling capacity is 16-bits per clock. i.e. at any time any resources of 8086 systems can handle up to 16-bit of data for processing.
- It has 16-bit address bus.
- It has 20-bit data bus.
- Direct addressing capability 1MB of memory. (1MB=220 Bytes).
- It has fourteen 16-bit registers.
- 24 operand addressing modes.
- Supports Bit, Byte, Word and Block level operations.
- 8 and 16 bit signed and unsigned arithmetic operations including multiply and divide. (previously the multiply and divide operations were carried out using the iterative looping of addition and subtraction operations respectively).
- Four general purpose registers, each of 16 bit wide. AX, BX, CX and DX. These can be used as 8-bit as well as 16-bit registers.
    - ✓ AX (16-bit register) AL & AH (2 x 8 bit registers)
    - ✓ BX (16-bit register) BL & BH (2 x 8 bit registers)

- ✓ CX (16-bit register) CL & CH (2 x 8 bit registers)
- ✓ DX (16-bit register) DL & DH (2 x 8 bit registers)

- Two Index group registers available Source Index (SI) and Destination index (DI).
- There are four Segment registers in 8086: Code Segment (CS), Data Segment (DS), Stack Segment (SS), Extra Segment (ES).
- Six Status flags and three control flags.
- Memory is Byte addressable each stores and 8-bit value.
- Addresses can be upto 20-bits long, resulting up to 1MB of memory (216 Bytes=1MB)
- Ranges of Clock rates: 5MHz for 8086, 8 MHz for 8086-1 and 10 MHz for 8086-2.
- Multi-bus system compatible interface
- Available as a 40 Pins Plastic-DIP and Lead Cer-DIP.

The internal functions of the 8086 processor are partitioned logically as two functional units as shown in the figure 1.1. They are

- Bus Interface Unit (BIU)
- Execution Unit (EU)

The BIU and EU function independently. The BIU interfaces the 8086 to the outside world. It fetches the instructions, Reads data from memory and ports, and writes data to memory and I/O ports.

The EU receives the program instruction codes (OP-codes) and Data from the BIU, executes these instructions and stores the results in general registers/memory or send them out as outputs through ports using BIU. The EU has no connections to the system buses. It receives and outputs all its data through the BIU.

BIU contains

- Segment Registers
- Instruction Pointer
- Instruction Queue

EU contains

- Arithmetic and Logic Unit (ALU)

- General Purpose Registers
- Index Registers
- Pointers
- Flag registers

Bus Interface Unit (BIU)

The function of BIU is to:

- Fetch the instruction or data from memory.
- Write the data to memory.
- Write the data to the port.
- Read data from the port.

Instruction Queue

- The use of this queue is to hold next six instructions to be executed in FIFO manner.
- To increase the execution speed, BIU fetches as many as six instruction bytes ahead to time from memory.
- All six bytes are then held in first in first out 6-byte register called instruction queue.
- Then all bytes have to be given to EU one by one.
- This pre fetching operation of BIU may be in parallel with execution operation of EU, which improves the speed execution of the instruction.

Execution Unit (EU)

The functions of execution unit are:

- To tell BIU where to fetch the instructions or data from.
- To decode the instructions.
- To execute the instructions.
- The EU contains the control circuitry to perform various internal operations. A decoder in EU decodes the instruction fetched memory to generate different internal or external control signals required to perform the operation. EU has 16-bit ALU, which can perform arithmetic and logical operations on 8-bit as well as 16-bit.

General Purpose Registers of 8086:

These registers can be used as 8-bit registers individually as AL-AH, BL-BH, CL-CH and DL-DH or can be used as 16-bit in pair to have AX, BX, CX, and DX.

Note: Any Register RX can be 16-bit register, which can be used as 2 eight bit registers RL and RH, where RL contains lower order byte of that 16-bit word and RH contains the higher order byte of the word.

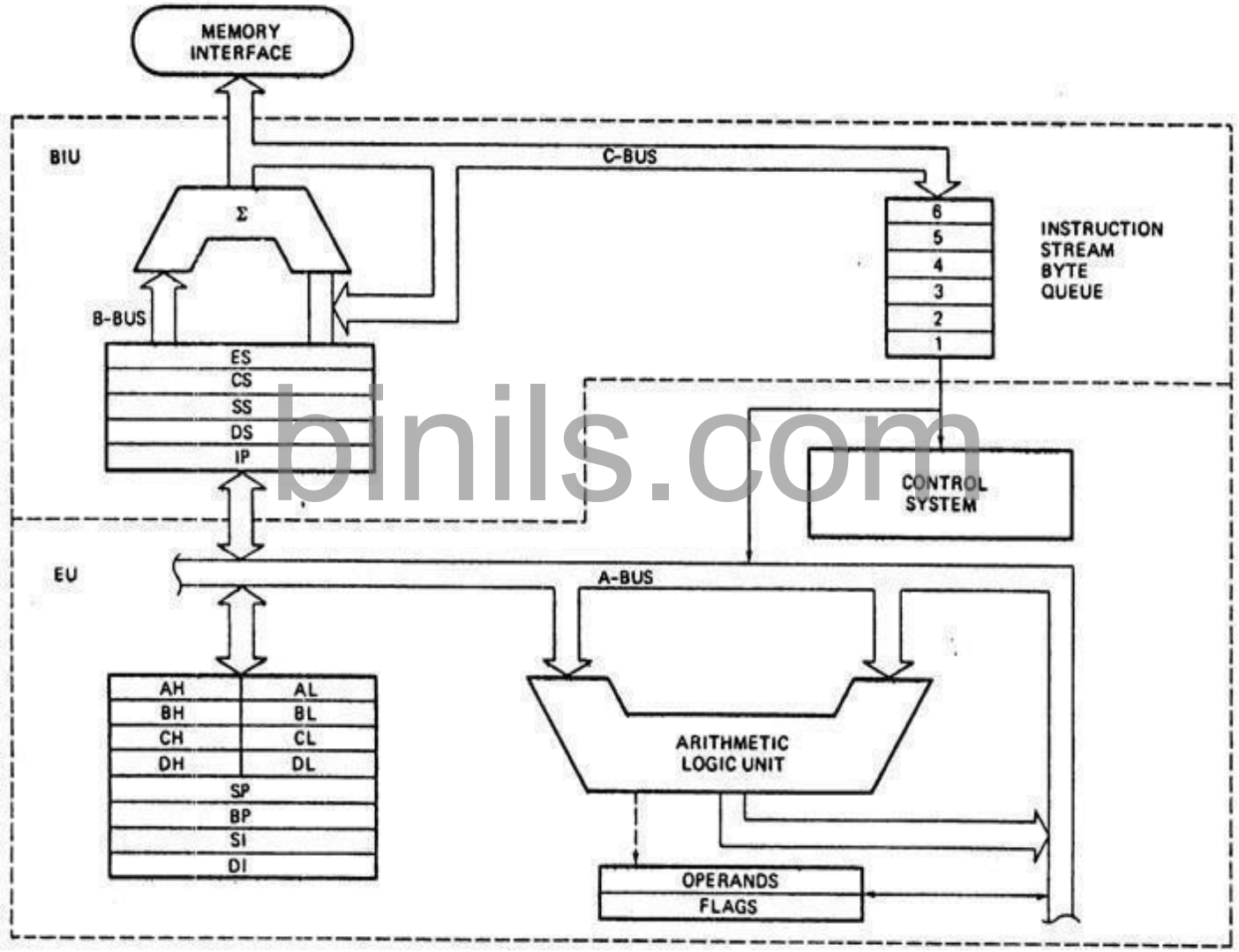

Figure 1.1.1 8086 Architecture

[Source: Advanced Microprocessors and Microcontrollersby A.K Ray & K.M. Bhurchandi]

Physical Address Formation: Generation of 20-bit Address

The 8086 addresses a segmented memory. The complete physical address which is 20-bits long is generated using segment and offset registers each of the size 16-bit. The content

of a segment register also called as segment address, and content of an offset register also called as offset address. To get total physical address, put the lower nibble0H to segment address and add offset address. The content of segment register is multiplied by 10H. i.e. shifted by 4 positions to the leftby inserting 4 zero bits and then the offset. i.e. the contents of IP register are added to the shifted contents of CS to generate physical address.

[Source: Advanced Microprocessors and Microcontrollersby A.K Ray & K.M. Bhurchandi]

Example: The contents of CS register are 348AH, therefore the shifted contents of CS register are 348A0H. When the BIU adds the offset of 4214H in IP to this starting address, we get 38AB4H as a 20-bit physical address of memory.

Register Organization of 8086:

All the registers of 8086 are 16-bit registers. The general purpose registers, can be used either 8- bit registers or 16-bit registers used for holding the data, variables and intermediate results temporarily or for other purpose like counter or for storing offset address for some particular addressing modes etc. The special purpose registers are used as segment registers, pointers, index registers or as offset storage registers for particular Addressing modes..

[Source: Advanced Microprocessors and Microcontrollersby A.K Ray & K.M. Bhurchandi]

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

BX Register: This register is mainly used as a **base register**. It holds the starting base location of a memory region within a data segment. It is used as offset storage for forming physical address in case of certain addressing mode.

CX Register: It is used as default counter - **count register** in case of string and loop instructions.

DX Register: Data register can be used as a port number in I/O operations and implicit operand or destination in case of few instructions. In integer 32-bit multiply and divide instruction the DXregister contains high-order word of the initial or resulting number.

Segment Registers:

1Mbyte memory is divided into 16 logical segments. The complete 1Mbyte memory segmentation is as shown in fig 1.4. Each segment contains 64Kbyte of memory. There are foursegment registers.

Code Segment (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by Instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions. It is used for addressing a memory location in the code segment of the memory, where the executableprogram is stored.

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

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

changed directly using POP and LDS instructions. It points to the data segment memory where the data is resided.

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

Pointers and Index Registers:

The pointers contain within the particular segments. The pointers IP, BP, SP usually containoffsets within the code, data and stack segments respectively:

Stack Pointer (SP) is a 16-bit register pointing to program stack in stack segment.

Base Pointer (BP) is a 16-bit register pointing to data in stack segment. BP register is usuallyused for based, based indexed or register indirect addressing.

Source Index (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirectaddressing, as well as a source data addresses in string manipulation instructions.

Destination Index (DI) is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data address in string manipulation instructions.

Flag Register:

[Source: Advanced Microprocessors and Microcontrollersby A.K Ray & K.M. Bhurchandi]

Flags Register determines the current state of the processor. They are modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program

8086 has 9 active flags and they are divided into two categories:

- Conditional Flags
- Control Flags

Conditional Flags

Carry Flag (CY): This flag indicates an overflow condition for unsigned integer **a**rithmetic. It isalso used in multiple-precision arithmetic.

Auxiliary Flag (AC): If an operation performed in ALU generates a carry/barrow from lower nibble (i.e. D0 – D3) to upper nibble (i.e. D4 – D7), the AC flag is set i.e. carry given by D3 bit to D4 is AC flag. This is not a general-purpose flag, it is used internally by the Processor to perform Binary to BCD conversion.

Parity Flag (PF): This flag is used to indicate the parity of result. If lower order 8-bits of the result contains even number of 1's, the Parity Flag is set and for odd number of 1's, the Parityflag is reset.

Zero Flag (ZF): It is set; if the result of arithmetic or logical operation is zero else it is reset.

Sign Flag (SF): In sign magnitude format the sign of number is indicated by MSB bit. If the result of operation is negative, sign flag is set.

Control Flags

Control flags are set or reset deliberately to control the operations of the execution unit. Control flags are as follows:

Trap Flag (TF): It is used for single step control. It allows user to execute one instruction of a program at a time for debugging. When trap flag is set, program can be run in single step mode.

Interrupt Flag (IF): It is an interrupt enable/disable flag. If it is set, the maskable interrupt of 8086 is enabled and if it is reset, the interrupt is disabled. It can be set by executing instruction sit and can be cleared by executing CLI instruction.

Direction Flag (DF): It is used in string operation. If it is set, string bytes are accessed from higher memory address to lower memory address. When it is reset, the string bytes are accessed from lower memory address to higher memory address.

1.2 ADDRESSING MODES

The set of mechanisms by which an instruction can specify how to obtain its operands is known as Addressing modes. The Addressing modes of 8086 can be broken into two categories such as,

1. Data related Addressing modes
2. Branch Addressing modes

The CPU can access the operands (data) in a number of different modes The 8086 has 12Addressing modes can be classified into five groups.

- Addressing modes for accessing immediate and register data (register and immediatemodes).
- Addressing modes for accessing data in memory (memory modes)
- Addressing modes for accessing I/O ports (I/O modes)
- Relative Addressing mode
- Implied Addressing mode

IMMEDIATE ADDRESSING MODE:

In this mode, 8 or 16 bit data can be specified as part of the instruction - OP Code Immediate Operand

Example 1:

MOV CL, 03 H: Moves the 8 bit data 03 H into CL

Example 2:

MOV DX, 0525 H: Moves the 16 bit data 0525 H into DX

In the above two examples, the source operand is in immediate mode and thedestination operand is in register mode.

A constant such as “VALUE” can be defined by the assembler EQUATE directivesuch as VALUE EQU 35H

Example:

MOV BH, VALUE

Used to load 35 H into BH

REGISTER ADDRESSING MODE:

The operand to be accessed is specified as residing in an internal register of 8086. Anyone internal registers can be used as a source or destination operand, however only the data registers can be accessed as either a byte or word.

Example 1: MOV DX,CX

MOV DX (Destination Register) , CX (Source Register) Which moves 16 bit content ofCS into DX.

Example 2: MOV CL, DL

Moves 8 bit contents of DL into CL

Example 3: MOV BX, CH is an illegal instruction.

The register sizes must be the same.

DIRECT ADDRESSING MODE:

The instruction Opcode is followed by an affective address, this effective Address is directly used as the 16 bit offset of the storage location of the operand from the location specified by the current value in the selected segment register. The default segment is always DS. The 20 bit physical Address of the operand in memory is normally obtained as PA = DS:EA

The data resides in a memory location in the data segment, whose effective Address may becomputed using 5000H as the offset Address and content of DS as segment address. The effective address, here, is 10H*DS+5000H.

Example 1: MOV AX, [5000H]

If DS = 1010H, OFFSET=5000, AX = 1000H then EA=15100H.DS:BX

1010H:5000H

10*HDS 10100

[BX] +5000

EA ❒ 12100H

Example 2:

MOV CH, START

If [DS] = 3050 and START = 0040

8 bit content of memory location 30540 is moved to CH.

REGISTER INDIRECT ADDRESSINGMODE:

The EA is specified in either pointer (BX) register or an index (SI or DI) register.

Example: MOV AX, [BX]

Here, data is present in a memory location in DS whose offset Address is in BX. The effective Address of the data is given as 10H*DS+ [BX].

MOV AX, [BX]

If DS = 1010H, BX = 2000H, AX = 1000H

then EA=12100H. DS:BX

1010H:2000H

10*HDS 10100

[BX] +2000

EA 12100H

INDEXED ADDRESSING:

The offset of the operand is stored in one of the index registers. DS and ES are the defaultsegments for index registers SI and DI respectively.

Example: MOV AX, [SI]

Here, data is available at an offset Address stored in SI in DS. The effective address, in this case,is computed as 10H*DS+ [SI].

If DS = 1010H, SI = 3010H,

then EA=13110H. DS: SI

❒1010H:

3010H 10*HDS❒ 10100

[BX] +3010

EA 13110H

REGISTER RELATIVE ADDRESSING:

In this Addressing mode, the data is available at an effective Address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default (either DS or ES) segment. The example given before explains this mode.

Example: MOV AX, 5000H [BX]

Here, effective Address is given as 10H*DS+50H+ [BX].

If DS = 1010H, BX = 2000H, offset=5000

then EA=17100H.

DS:[5000+BX] 1010H:

5000+2000H

10*H 10100

DS

Offset 5000

[BX] +

2000

EA 17100H

BASED INDEXED:

The effective Address of data is formed, in this Addressing mode, by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI). The default segment register may be ES or DS.

Example: MOV AX, [BX] [SI]

Here, BX is the base register and SI is the index register. The effective Address is computed as10H*DS+ [BX] + [SI]

If DS = 1010H, BX = 2000H, SI=3010

then EA=15110H.

DS:[SI+BX]

1010H:[3010H: 2000H]

10*HDS 10100

[SI] 3010

[BX] +2000

EA 15110H

RELATIVE BASED INDEXED:

The effective Address is formed by adding an 8-bit or 16-bit displacement with the sum of contents of any one of the bases registers (BX or BP) and any one of the index registers, in a default segment.

Example: MOV AX, 5000H [BX] [SI]

Here, 50H is an immediate displacement, BX is a base register and SI is an index register. Theeffective address of data is computed as 160H*DS+ [BX] + [SI] + 50H.

If DS = 1010H, BX = 2000H, SI=3010 then

EA=1A110H.

DS:[5000+BX+SI] 1010H:

[5000+2000H+3010H]

10*H 10100

DS

[SI] 3010

[BX] 2000

Offset +5000

EA 1A110H

BRANCH RELATED ADDRESSING MODES:

These type of Addressing are related to whether the Addressing is within the same segment or to a different segment. Accordingly the Addressing modes in this category are knownas intrasegment and intersegment with direct or indirect

addressing. These are explained below:

INTRASEGMENT DIRECT ADDRESSING MODE:

The effective Address is the sum of the IP and 8 / 16 bit displacement. It leads to a short jump if displacement is 8 bit, and this Addressing may be used conditional or unconditional in a program.

INTRASEGMENT INDIRECT ADDRESSING:

In this Addressing mode the effective Address may be in a register or at a memory location as accessed by any data related addressingmode except the immediate and implied mode. This Addressing mode is called only unconditionally.

INTERSEGMENT DIRECT ADDRESSING MODE:

This Addressing mode when used replaces the content of the CSand IP with the offset and segment part of the instruction. Used to branch from one segment to another segment.

1.3 INSTRUCTION SET

The 8086 instructions are categorized into the following main types.

1. Data Copy / Transfer Instructions
2. Arithmetic and Logical Instructions
3. Shift and Rotate Instructions
4. Loop Instructions
5. Branch Instructions
6. String Instructions
7. Flag Manipulation Instructions
8. Machine Control Instructions

DATA COPY / TRANSFER INSTRUCTIONS:

The data transfer instructions move data between memory and the general-purpose and segment registers, and perform operations such as conditional moves, stack access, and data conversion.

There are four basic 8086 instructions for transferring quantities to and/or fromthe registers and memory such as,

- General purpose data transfer instructions
- I/O transfer instruction
- Special address transfer instruction
- Flag transfer instruction

General purpose data transfer instructions

- MOV
- PUSH
- POP
- XCHG
- XLAT

MOV:

This instruction copies a word or a byte of data from some source to a destination. The destination can be a register or a memory location. The source can be a register, a memory

location, or an immediate number.

Syntax:

```
MOV destination, source
```

Depending on the addressing modes it can transfer information from

SPECIAL ADDRESS TRANSFER

LEA: Load Effective Address

Load effective address of the operand into specified register

Eg: LEA BX,ADR :effective address of label ADR

LDS: Load DS register and other specified register from memory.

Eg. LDS BX,5000H

LES: Load ES register and other specified register from memory.

Eg. LES BX,5000H

Flag transfer instructions:LAHF**:**

Load (copy to) AH with the low byte the flag register. [AH] [Flags low byte]

SAHF:

Store (copy) AH registers to low byte of flag register. [Flags low byte] [AH]

PUSHF**:**

Copy flag register to top of stack.

POPF:

Copy word at top of stack to flag register.

Arithmetic Instructions:

The 8086 provides many arithmetic operations: addition, subtraction, negation, increment, decrement multiplication and comparing two values.

Addition Instruction**:**

- Add contents of two registers with or without carry

- Add contents of a registers and a memory with or without carry
- Add immediate data to a registers or a memory with or without carry
- Increment the content of a register or a memory location
- To perform ASCII adjustment after addition
- To perform decimal adjustment after addition

ADD:

The add instruction adds the contents of the source operand to the destination operand.Syntax:

ADD oper1, oper2

ADD AX, 0100H	Add immediate value to the content of AX
ADD AX, BX	Add contents of AX and BX and result in AX
ADD AX, [SI]	Add word from memory at offset [SI] inDS to the content of DX
ADD AX, [5000H]	Add content of data whose address is 5000H with AX andresult in AX
ADD [5000H], 0100H	Add immediate value to the content of data whose address is 5000H and result in 5000H

ADC: Add with Carry

This instruction performs the same operation as ADD instruction, but adds the carry flag tothe result.

ADC AX, 0100H	Add immediate value plus carry status to the content of AX
ADC AX, BX	Add contents of AX and BX plus carry status and result in AX
ADC AX, [SI]	Add word from memory at offset [SI] in DS plus carry status to the content ofDX
ADC AX, [5000H]	Add content of data whose address is 5000H plus carry status with AX and result in AX
ADC [5000H], 0100H	Add immediate value to the content of data whose addressis 5000H plus carry status and result in 5000H

INC: Increment

This instruction increases the contents of the specified Register or memory location.

Immediate data cannot be operand of this instruction. Eg. INCAX

INC [BX] INC [5000H]

AAA: ASCII Adjust After Addition

- The AAA instruction is executed after an ADD instruction that add two ASCII coded operand to give a byte of result in AL.
- The AAA instruction converts the resulting contents of AL to a unpacked decimal digits.
- After the addition it will check the lower 4 bits of AL is a valid BCD number in the range of 0 to 9
- If it is between 0 to 9 the AF is zero and AAA sets AH=0
- If lower digit of AL is between 0 to 9 AF is set,06 is added to AL. The upper 4 bits ofAL are cleared and AH is incremented by one
- If lower digit of AL greater than 9, then 06 is added to AL. The upper 4 bits of AL arecleared and AH is incremented by one.

DAA: Decimal Adjust After Addition

- The DAA instruction is executed after an ADD instruction that add two ASCII coded operand to give a byte of result in AL.
- The DAA instruction converts the resulting contents of AL to a unpacked decimal digits.
- If lower nibble is greater than 9, after addition it will add 06 to the lower nibble in AL.
- After adding 06 to lower nibble of AL, if upper nibble of AL is greater than 9, then adds 60H to AL.

Subtraction Instruction**:**

- Subtract contents of two registers with or without carry
- Subtract contents of a registers and a memory with or without carry
- Subtract immediate data to a registers or a memory with or without carry

- Decrement the content of a register or a memory location
- To perform ASCII adjustment after Subtract
- To perform decimal adjustment after Subtract

SUB: Subtract

The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand.

SUB AX, 0100H	Subtract immediate value to the content of AX
SUB AX, BX	Subtract contents of AX and BX and result in AX
SUB AX, [SI]	Subtract word from memory at offset [SI] in DS to the content of DX
SUB AX, [5000H]	Subtract content of data whose address is 5000H with AX and result in AX
SUB [5000H], 0100H	Subtract immediate value to the content of data whose address is 5000H and result in 5000H

SBB: Subtract with Borrow

The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand

SBB AX, 0100H	Subtract immediate value plus carry status to the content of AX
SBB AX, BX	Subtract contents of AX and BX plus carry status and result in AX
SBB AX, [SI]	Subtract word from memory at offset [SI] in DS plus carry status to thecontent of DX
SBB AX, [5000H]	Subtract content of data whose address is 5000H plus carry status with AX and result in AX
SBB [5000H], 0100H	Subtract immediate value to the content of data whose address is 5000H plus carry status and result in5000H

DEC: Decrement

The decrement instruction subtracts 1 from the contents of the specified register ormemory location.

Eg. DEC AX DEC [5000H]

AAS: ASCII Adjust After Subtraction

- The AAA instruction is executed after an SUB instruction that subtracts two ASCII coded operand to give a byte of result in AL.
- The AAA instruction converts the resulting contents of AL to a unpacked decimaldigits.
- After the addition it will check the lower 4 bits of AL is a valid BCD number in the range of 0 to9
- If it is between 0 to 9 the AF is zero and AAA sets AH=0
- If lower digit of AL is between 0 to 9 AF is set,06 is subtracted to AL. The upper 4bits of AL are cleared and AH is incremented by one.
- If lower digit of AL greater than 9, then 06 is subtracted to AL. The upper 4 bits ofAL are cleared and AH is incremented by one.

DAS: Decimal Adjust After Subtraction

- The DAA instruction is executed after an SUB instruction that subtract two ASCIIcoded operand to give a byte of result in AL.
- The DAA instruction converts the resulting contents of AL to a unpacked decimaldigits.
- If lower nibble is greater than 9, after subtraction it will subtract 06 to the lowernibble in AL.
- After subtracting 06 to lower nibble of AL, if upper nibble of AL is greater than9, then subtract 60H to AL.

NEG: Negate

The negate instruction forms 2's complement of the specified destination in the instruction. Thedestination can be a register or a memory location. This instruction can be implemented by inverting each bit and adding 1 to it.

Eg. NEG AL

AL = 0011 0101 35H

Replace number in AL with its 2's complementAL = 1100 1011 = CBH

CMP: Compare

This instruction compares the source operand, which may be a register or an immediate data ora memory location, with a destination operand that may be a register or a memory location

Eg. CMP BX, 0100H CMP AX, 0100HCMP [5000H], 0100H CMP BX, [SI] CMP BX, CX

Multiplication Instruction:

MUL: Unsigned Multiplication Byte or Word

This instruction multiplies an unsigned byte or word by the contents of AL.Eg. MUL BH; (AX) (AL) x (BH)

MUL CX; (DX)(AX) (AX) x (CX)

MUL WORD PTR [SI]; (DX)(AX) (AX) x ([SI])

IMUL: Signed Multiplication

This instruction multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by a signed word in AX.

Eg. IMUL BH IMUL CXIMUL [SI]

AAM: ASCII Adjust after Multiplication

This instruction, after execution, converts the product available In AL into unpackedBCD format.

Eg. MOV AL, 04; AL = 04MOV BL ,09; BL = 09

MUL BL; AX = AL*BL; AX=24H AAM;AH = 03, AL=06

Division Instruction: DIV: Unsigned division

This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word by a word.

Eg. DIV CL; Word in AX / byte in CL; Quotient in AL, remainder in AH

DIV CX; Double word in DX and AX / word; in CX, and Quotient in AX;remainder in DX

IDIV: Unsigned division

This instruction is used to divide an signed word by a byte or to divide an unsigned double word by a word.

Eg. IDIV CL; Word in AX / byte in CL; Quotient in AL, remainder in AH IDIV CX; Double word in DX and AX / word; in CX, and Quotient inAX;

remainder in DX

AAD: ASCII Adjust before Division

This instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. In the instruction sequence, this instruction appears before DIV instruction.

Eg. AX 05 08

AAD result in AL 00 3A 58D = 3A H in AL

The result of AAD execution will give the hexadecimal number 3A in AL and 00 in AH where 3A is the hexadecimal Equivalent of 58(decimal).

CBW: Convert Signed Byte to Word

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said tobe sign extension of AL.

Eg. CBW

AX= 0000 0000 1001 1000 Convert signed byte in AL signed word in AX. Result in AX

= 1111 1111 1001 1000

CWD: Convert Signed Word to Double Word

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said tobe sign extension of AL.

Eg. CWD

Convert signed word in AX to signed double word in DX: AX DX= 1111 1111 1111 1111

Result in AX = 1111 0000 1100 0001

Logical instructionsAND: Logical AND

This instruction bit by bit ANDs the source operand that may be an immediate register or a memory location to the destination operand that may a register or a memory location.The result is stored in the destination operand.

Syntax:

AND destination, source

Eg. AND AX, 0008HAND AX, BX

If the Content of AX is 3A0F

AX	0011	1010	0000	1111
AND				
0008	0000	0000	0000	1000
AX	0000	0000	0000	1000

OR: Logical OR

This instruction bit by bit ORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Syntax:

OR destination, source

NOT: Logical Invert

This instruction complements the contents of an operand register or a memory location,bit by bit.

Syntax:

```
NOT destination
```

XOR: Logical Exclusive OR

This instruction bit by bit XORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The resultis stored in the destination operand.

Syntax:

```
XOR destination, source
```

Eg. XOR AX, 0098HXOR AX,BX

TEST: Logical Compare Instruction

The TEST instruction performs a bit by bit logical AND operation on the two operands. The result of this ANDing operation is not available for further use, but flags are affected.

Syntax:

```
TEST destination, source
```

Eg. TEST [0500], 06H

Shift and Rotate InstructionsSAL/SHL:

SAL and SHL are two mnemonics for the same instruction.

- This instruction shifts each bit in the specified destination to the left and 0 is stored atLSB position.
- The MSB is shifted into the carry flag.
- The destination can be a byte or a word.
- It can be in a register or in a memory location.
- The number of shifts is indicated by count.Syntax:

```
SAL / SHL destination, count.
```

Eg. SAL CX, 1

SHL AX,CL

SHR: SHR destination, count

This instruction shifts each bit in the specified destination to the right and 0 is stored atMSB position.

- The LSB is shifted into the carry flag.
- The destination can be a byte or a word.
- It can be a register or in a memory location.
- The number of shifts is indicated by count.

Syntax:

SHR destination, count.

SAR: SAR destination, count

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. The LSB will be shifted into CF.

Syntax:

SAR destination, count

ROL Instruction: Rotate left without carry

This instruction rotates all bits in a specified byte or word to the left some number of bit positions. MSB is placed as a new LSB and a new CF.

Syntax: ROL destination, count.

ROR Instruction: Rotate right without carry

This instruction rotates all bits in a specified byte or word to the right some numberof bit

positions. LSB is placed as a new MSB and a new CF.

Syntax: ROR destination, count

RCL Instruction: Rotate left with carry

This instruction rotates all bits in a specified byte or word some number of bit positions to the left along with the carry flag. MSB is placed as a new carry and previous carry is placeas new LSB.

Syntax: RCL destination, count.

RCR Instruction: Rotate right with carry

This instruction rotates all bits in a specified byte or word some number of bit positions to the right *along with the carry flag*. LSB is placed as a new carry and previous carry is place as newMSB.

Syntax: RCR destination, count.

Loop Instructions:

Unconditional LOOP Instructions

LOOP: LOOP Unconditionally

This instruction executes the part of the program from the Label or Address specified in the instruction upto the LOOP instruction CX number of times. At each iteration, CX is decremented automatically and JUMP IF NOT ZERO structure.

Example: MOV CX, 0004H

Conditional LOOP Instructions

LOOPZ / LOOPE Label

Loop through a sequence of instructions from label while ZF=1 and CX=0.

LOOPNZ / LOOPENE Label

Loop through a sequence of instructions from label while ZF=1 and CX=0.

Branch Instructions:

Branch Instructions transfers the flow of execution of the program to a new Address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location to be transferred.The Branch Instructions are classified into two types

i. Unconditional Branch Instructions.

ii. Conditional Branch Instructions.

Unconditional Branch Instructions:

In Unconditional control transfer instructions, the execution control is transferred to the specifiedlocation independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

CALL: Unconditional Call

This instruction is used to call a Subroutine (Procedure) from a main program. Addressof procedure may be specified directly or indirectly. There are two types of procedure depending upon whether it is available in the same segment or in another segment.

i. Near CALL i.e., ±32Kdisplacement.

ii. For CALL i.e., anywhere outside the segment.

On execution this instruction stores the incremented IP & CS onto the stack and loads theCS & IP registers with segment and offset Addresses of the procedure to be called.

RET: Return from the Procedure.

At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with Flags are retrieved into the CS, IP and Flag registers from the stack and execution of the main program continues further.

INT N: Interrupt Type N.

In the interrupt structure of 8086, 256 interrupts are defined corresponding to the types from 00H to FFH. When INT N instruction is executed, the type byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from memory block in 0000 segment.

INTO: Interrupt on Overflow

This instruction is executed, when the overflow flag OF is set. This is equivalent to aType 4 Interrupt instruction.

JMP: Unconditional Jump

This instruction unconditionally transfers the control of execution to the specified Address using an 8-bit or 16-bit displacement. No Flags are affected by this instruction.

IRET: Return from ISR

When it is executed, the values of IP, CS and Flags are retrieved from the stack tocontinue the

execution of the main program.

Conditional Branch Instructions

When this instruction is executed, execution control is transferred to theAddressspecified relatively in the instruction, provided the condition implicit in the Opcode is satisfied. Otherwise execution continues sequentially.

JZ/JE Label

Transfer execution control to Address 'Label', if ZF=1.

JNZ/JNE Laber

Transfer execution control to Address 'Label', if CF=0.

JCXZ Label

Transfer execution control to Address 'Label', if CX=0

String Manipulation Instructions

A series of data byte or word available in memory at consecutive locations, to be referred as Byte String or Word String. A String of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent. The 8086

supports a set of more powerful instructions for string manipulations for referring to a string, two parameters are required.

- Starting and End Address of the String.
- Length of the String.

The length of the string is usually stored as count in the CX register. The incrementing or decrementing of the pointer, in string instructions, depends upon the Direction Flag (DF) Status. Ifit is a Byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation, the index registers are updated bytwo.

REP: Repeat Instruction Prefix

This instruction is used as a prefix to other instructions, the instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one).

- REPE / REPZ - repeat operation while equal / zero.
- REPNE / REPNZ - repeat operation while not equal / not zero.

These are used for CMPS, SCAS instructions only, as instruction prefixes.

MOVSB / MOVSW: Move String Byte or String Word

Suppose a string of bytes stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of source string is located in the memory location whose Address may be computed using SI (Source Index) and DS (Data Segment) contents. The starting Address of the destination locations where this string has to be relocated isgiven by DI (Destination Index) and ES (Extra Segment) contents.

CMPS: Compare String Byte or String Word

The CMPS instruction can be used to compare two strings of byte or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero Flag is set.

The REP instruction Prefix is used to repeat the operation till CX (counter) becomes zero or thecondition specified by the REP Prefix is False.

SCAN: Scan String Byte or String Word

This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The String is pointed to by ES: DI register pair. The length of the string stored in CX. The DF controls the mode for scanning of the string. Whenever a match to the

specified operand is found in the string, execution stops and the zero Flag is set. If no match is found, the zero flag is reset.

LODS: Load String Byte or String Word

The LODS instruction loads the AL / AX register by the content of a string pointed to by DS: SI register pair. The SI is modified automatically depending upon DF, If it is a byte transfer (LODSB), the SI is modified by one and if it is a word transfer (LODSW), the SI is modified by two. No other Flags are affected by this instruction.

STOS: Store String Byte or String Word

The STOS instruction Stores the AL / AX register contents to a location in the string pointer by ES: DI register pair. The DI is modified accordingly, No Flags are affected by this instruction.

The direction Flag controls the String instruction execution, the source index SI and Destination Index DI are modified after each iteration automatically. If DF=1, then the execution follows auto decrement mode, SI and DI are decremented automatically after each iteration. If DF=0, then the execution follows auto increment mode. In this mode, SI and DI are incremented automatically after each iteration.

Flag Manipulation and Processor Control Instructions

These instructions control the functioning of the available hardware inside theprocessor chip. These instructions are categorized into two types:

- Flag Manipulation instructions.
- Machine Control instructions.

Flag Manipulation instructions

The Flag manipulation instructions directly modify some of the Flags of 8086.

i. CLC – Clear Carry Flag.
ii. CMC – Complement Carry Flag.
iii. STC – Set Carry Flag.
iv. CLD – Clear Direction Flag.
v. STD – Set Direction Flag.
vi. CLI – Clear Interrupt Flag.
vii. STI – Set Interrupt Flag.

Machine Control instructions

The Machine control instructions control the bus usage and execution

i. WAIT – Wait for Test input pin to go low.
ii. HLT – Halt the process.
iii. NOP – No operation.
iv. ESC – Escape to external device like NDP
v. LOCK – Bus lock instruction prefix.

1.4 ASSEMBLER DIRECTIVES

Assembler directives help the assembler to correctly understand the assembly language programs to prepare the codes. Another type of hint which helps the assembler to assign a particular constant with a label or initialize particular memory locations or labels with constants is called an operator. Rather, the operators perform the arithmetic and logical tasks unlike directives that just direct the assembler to correctly interpret the program to code it appropriately. The following directives are commonly used in the assembly language programming practice using Microsoft Macro Assembler (MASM) or Turbo Assembler (TASM).

DB: Define Byte

The DB directive is used to reserve byte or bytes of memorylocations in the available memory.

LIST DB 0lH, 02H, 03H, 04H

This statement directs the assembler to reserve four memory locations for a list named LIST andinitialize them with the above specified four values.

DW: Define Word

It makes the assembler reserve the number of memory words (16-bit) instead of bytes. Some examples are given to explain this directive.

Examples WORDS DW 1234H, 4567H, 78ABH, 045CH

DQ: Define Quad word

This directive is used to direct the assembler to reset 4words (8 bytes) of memory for the specified variable and may initialize it with the specified values.

DT: Define Ten Bytes

The DT directive directs the assembler to define the specified variable requiring l -bytes for its storage and initialize the 10bytes with the specified values.

ASSUME: **Assume Logical Segment Name**

The ASSUME directive is used to inform the assembler, the names of the logical segments to be assumed for different segments used in the program. In the assembly language program, each segment is given aname.

For example, the code segment may be given the name CODE, data segment may be given the name DATA etc.

ASSUME CS:

CODE directs the assembler that the machine codes are available in a segment named CODE, and hence the CS register is to be loaded with the Address(segment) allotted by the operating system for the label CODE, while loading.

ASSUME DS:

DATA indicates to the assembler that the data items related to theprogram, are available in a logical segment named DATA, and the DS register is to be initialized by the segment Address value decided by the operating system for thedata segment, while loading.

END: END of Program

The END directive marks the end of an assembly language program. When the assembler comes across this END directive, it ignores the source lines available later on. Hence, it should be ensured that the END statement should be the last statement in the file and should not appear in between.

ENDP: END of Procedure.

In assembly language programming, the subroutines arecalled procedures. Thus, procedures may be independent program modules which return particular results or values to the calling programs. The ENDP directive is used toindicate the end of a procedure.

```
PROCEDURE STAR
.
.
STAR ENDP
```

ENDS: END of Segment

This directive marks the end of a logical segment.

DATA SEGMENT

.

.

.

DATA ENDS

ASSUME CS: CODE, DS:DATA CODE SEGMENT.

.

.

.

C

O

D

E

E

N

D

S

E

N

D

The above structure represents a simple program containing two segments named DATA and CODE. The data related to the program must lie between the DATA SEGMENT and DATA ENDS statements. Similarly, all the executable instructions must lie between CODE SEGMENT and CODE ENDS statements.

EVEN: Align on Even Memory Address

The EVEN directive updates the location counterto the next even Address if the current location counter contents are not even, and assigns the following routine or variable or constant to that Address.

EQU: Equate

The directive EQU is used to assign a label with a value or a symbol. Theuse of this directive is just to reduce the recurrence of the numerical values or constants in a program code.

Example

```
LABEL
EQU
0500H
ADDITI
ON EQU
ADD
```

The first statement assigns the constant 500H with the label LABEL, while the second statement assigns another label ADDITION with mnemonic ADD.

EXTRN: External and Public

The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly language modules.

GROUP: Group the Related segment

The directive is used to form logical groups of segments with similar purpose or type. This directive is used to inform the assembler to form a logical group of the following segment names.

PROGRAM GROUP CODE, DATA, STACK

The above statement directs the loader/linker to prepare an EXE file such that CODE, DATA and STACK segment must lie within a 64kbyte memory segment that is named as PROGRAM. Now, for the ASSUME statement, one can use the label PROGRAM rather than CODE, DATA and STACK as shown.

ASSUME CS: PROGRAM, DS: PROGRAM, SS: PROGRAM.

LABEL: Label

The Label directive is used to assign a name to the current content of the location counter. At the start of the assembly process, the assembler

initializes a location counter to keep track of memory locations assigned to the program.

LENGTH: Byte Length of a Label

This directive is not available in MASM. This isused to refer to the length of a data array or a string.

MOV CX, LENGTH ARRAY

LOCAL:

The labels, variables, constants or procedures declared LOCAL in a module areto be used only by thatmodule.

NAME: Logical Name of a Module

The NAME directive is used to assign a name to an assembly language program module. The module may now be referred to by its declared name.

OFFSET: Offset of a Label

When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement (also called as offset interchangeably) of the particular label, and replaces the string 'OFFSET LABEL' by thecomputed displacement.

ORG: Origin

The ORG directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared Addressing. The ORG statement while starting the assembly process for a module, the assembler initializes a location counter to keep track of the allotted addresses for the module. If the ORG statement is not written in the program, the location counter is initialized to 0000. If an ORG 200H statement is present at the starting of the code segment of that module, then the code will start from 200H Addressing code segment.

PROC: Procedure

The PROC directive marks the start of a named procedure in the statement.

PTR: Pointer

The pointer operator is used to declare the type of a label, variable ormemory operand. The operator PTR is prefixed by either BYTE or WORD. If the prefix is BYTE, then the particular label, variable or memory operand is treated as an 8-bit quantity while if WORD is the prefix, then it is treated as a 16- bit quantity.

Example:

MOV AL, BYTE PTR [SI];

Moves content of memory location addressed by SI (8-bit) toAL

SEG: Segment of a Label

The SEG operator is used to decide the segment Address of the label, variable, or procedure and substitutes the segment base Address in place of 'SEG label'. The example given below explains the use of SEG operator.

Example MOV AX, SEG ARRAY;

This statement moves the segment address

SEGMENT: Logical Segment

The SEGMENT directive marks the starting of alogical segment. The started segment is also assigned a name, i.e. label, by this statement. The SEGMENT and ENDS directive must bracket each logical segment of a program.

TYPE :

The TYPE operator directs the assembler to decide the data type of the specified label and replaces the 'TYPE label' by the decided data type. For the word type variable, the data type is 2, for double word type, it is 4, and for byte type, it is 1.

GLOBAL:

The labels, variables, constants or procedures declared GLOBAL may be used by other modules of the program. Once a variable is declared GLOBAL, it can be used by any module in the program. The following statement declares the procedure ROUTINE as a global label.

1.5 MODULAR PROGRAMMING

Complex programs are divided into many parts and each sub-part are known as modules. All the modules perform a well-defined task. Formulation of computer code using a module is known as modular programming.

The reasons for breaking a program into small parts are

- Modules are easier to understand.
- Different modules can be assigned to different programmers.
- The debugging and testing can be done in a more orderly fashion.
- Documentation can be more easily understood.
- Modifications may be localized.

Most assembler languages are used in modularization process in three ways such as,

1. Allow data to be structured so that they can be accessed by several modules.

2. Provide for procedures or subroutines.

3. Permit sections of code known as macros.

To perform modular programming, the following tasks must be performed.

- Linking and relocation
- Stack Operation
- Procedures
- Interrupt process

LINKING AND RELOCATION

The assembly language program can be written with an ordinary text editors such as word star, editor etc. The assembly language program text is an input to the assembler. The assembler translates assembly language statements to their binary equivalent known as object code. During assembling process assembler checks for syntax errors and displays them before giving object code module. The object code module contains the information about where the program or module to beloaded in memory. If the object module is to be linked with other separate modules then it contains additional linkage information.

At link time, separately assembled modules are combined into one single load module by the linker. The linker also SUBs any required initialization or finalization code

to allow the OS to start the program running and to return control to OS after the program is completed. At load time, the program loader copies the program into computer main memory and at execution time, the program execution begins. If the modules in the program they are assembled separately, then there is one main module and other modules. This main module has the first instruction to be executed and it is terminated by an END statement with entry point satisfied. Other modules are terminated by an END statement with no operand.

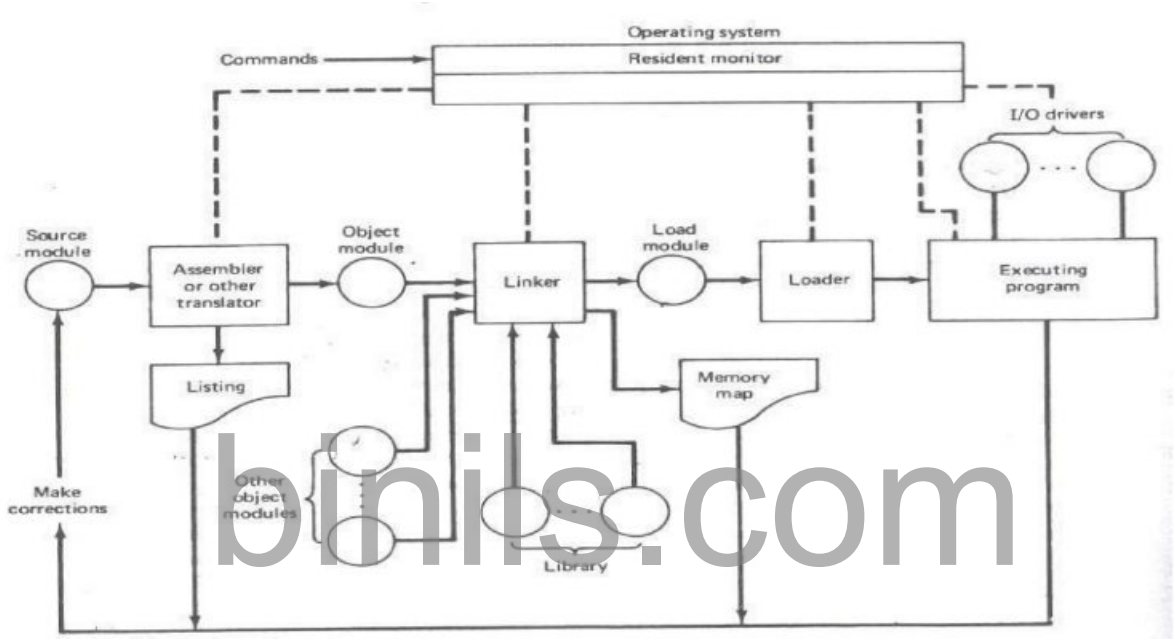

Figure 1.5.1 Creation and Execution of Assembly language program

[Source: "Microcomputer Systems: The 8086 / 8088 Family - Architecture, Programming and Design" by Yu-Cheng Liu, Glenn A.Gibson]

SEGMENT COMBINATION

In addition to the linker commands, the assembler provides a means of regulating the waysegments in different object modules are organized by the linker. Segments with same name are joined together by using the modifiers attached to theSEGMENT directives. SEGMENT directive may have the form Segment name SEGMENT Combination-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 the segments in different modules have the same name and combine- type

PUBLIC, then they are concatenated into a single element 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 starting 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 individuallyspecified 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-type.

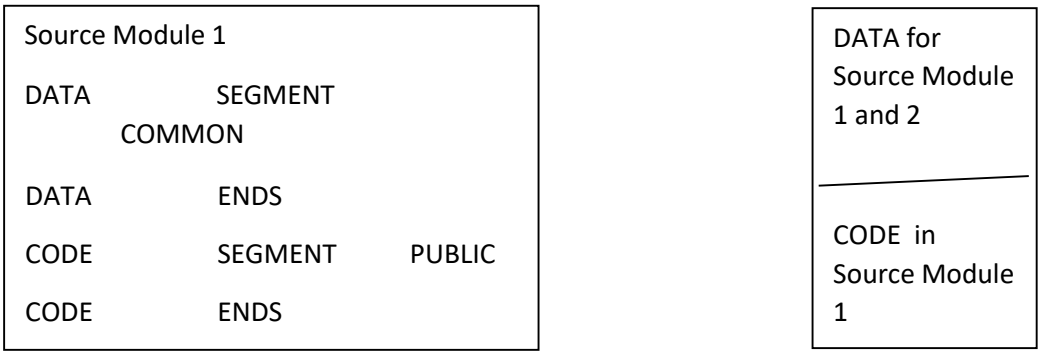

Figure 1.5.2 Segment combinations resulting from the PUBLIC and Common Combination types

[Source: "Microcomputer Systems: The 8086 / 8088 Family - Architecture, Programming and Design" by Yu-Cheng Liu, Glenn A.Gibson]

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. 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* relativeto the module. Two lists are implemented by the EXTRN and PUBLIC directives, which have the forms:

EXTRN Identifier Type….. Identifier Type And

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

The assembler must know the type of all external identifiers before it can generate the proper machine code; a type specifier must be associated with each identifier in an EXTRN statement. For a variable, the type may be BYTE, WORD, or DWORD and for a label it may be NEAR or FAR.

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 reference and a linker error will occur. The offsets for the local identifier willbe inserted by the assembler, but the offsets for the external identifiers and all segment addresses must be inserted by the linking process. The offsets associated with all external references can be assigned once all of the object modules have been found and their external symbol tables have been examined. The assignment of the segment addresses is called *relocation* and is done after the linking process has determined exactly where each segment is to be put in memory.

STACKS

The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU. It is a top-down data structure whose elements are accessed using the stack pointer (SP) which gets decremented by two as we store a data word into the stack and gets incremented by two as we retrieve a data word from the stack back to the CPU register.

The process of storing the data in the stack is called **'pushing into'** the stack and the reverse process of transferring the data back from the stack to the CPU register is known as **'popping off'** the stack. The stack is essentially *Last-In-First -Out* (LIFO) data segment. This means that the data which is pushed into the stack last will be on top of stack and will be popped off the stack first.

The stack pointer is a 16-bit register that contains the offset Address of the memory locationin the stack segment. The stack segment, like any other segment, may have a memory block of a maximum of 64 Kbytes locations, and thus may overlap with any other segments. Stack Segment register (SS) contains the base Address of the stack segment in the memory.

The Stack Segment register (SS) and Stack pointer register (SP) together Address the stack- top. For a selected value of SS, the maximum value of SP=FFFFH and the segment can have maximum of 64K locations. If the SP starts with an initial value of FFFFH, it will be decremented by two whenever a 16-bit data is pushed onto the stack. After successive push operations, when the stack pointer contains 0000H, any attempt to further push the data to the stack will result in stack overflow.

After a procedure is called using the CALL instruction, the IP is incremented to the next instruction. Then the contents of IP, CS and flag register are pushed automatically to the stack. The control is then transferred to the specified Addressing the CALL instruction i.e. starting Address of the Procedure. Then the procedure is executed.

PROCEDURES

A procedure 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. The branch to 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.

CALLS, RETURNS, AND PROCEDURE DEFINITIONS

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 registers used by the procedure need to be stored before their contents are changed, andthen restored just before their contents are changed, and then restored just before the procedure is excited.

A CALL may be direct or indirect and intrasegment or intersegment. If the CALL is intersegment, the return must be intersegment. Intersegment call must push both (IP) and (CS) onto the stack. The return must correspondingly pop two words from the stack. In the case of intrasegment call, only the contents of IP will be saved and retrieved when call and return instructions are used.

Procedures are used in the source code by placing a statement of the form at the beginning of the procedure

Procedure name PROC Attribute

Procedure name ENDP

The attribute that can be used will be either NEAR or FAR. 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:

- The same code segment as the statement that calls it.
- A code segment that is different from the one containing the statement that calls it, but in thesame source module as the calling statement.
- A different source module and segment from the calling statement.

In the first case, the attribute could be NEAR provided that all calls are in the same code segment as the procedure. For the latter two cases the attribute must be FAR. If the procedure is given a FAR attribute, then all calls to it must be intersegment calls even if the call is from the same code segment. For the third case, the procedure name must be declared in EXTRN and PUBLIC statements.

SAVING AND RESTORING REGISTERS

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

MSK PROC NEARPUSH AX PUSH BX PUSHCX

POP CXPOP BXPOP AXRET

MSK ENDP

PROCEDURE COMMUNICATION

There are two general types of procedures, those operate on the same set of data and those that may process a different set of data each time they are called. If a procedure is in the same source module as the calling program, then the procedure can refer tothe variables directly. When the procedure is in a separate source module it can still refer to the source moduledirectly provided that the calling program contains the directive PUBLIC ARY, COUNT, SUM EXTRN ARY: WORD, COUNT: WORD, SUM: WORD

RECURSIVE PROCEDURES

When a procedure is called within another procedure it called recursive procedure. To make sure that the procedure does not modify itself, each call must store its set of parameters, registers, and all temporary results in a different place in memory

Eg. Recursive procedure to compute the factorial

Disadvantages of Procedure

a. Linkage associated with them.
b. It sometimes requires more code to program the linkage than is needed to perform the task. Ifthis is the case, a procedure may not save memory and execution time is considerably increased.

MACROS

Macros are needed for providing the programming ease of a procedure while avoiding the linkage. 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 reference.

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. The prototype code along with the statements for referencing and terminating is called the macro definition.

Once a macro is defined, it can be inserted at various points in the program by using macrocalls. When a macro call is encountered by the assembler, the assembler replaces the call with the macro code. Insertion of the macro code by the assembler for a macro call is referred to as a macro expansion. During a macro expansion, the first actual parameter replaces the first dummy parameter in the prototype code, the second actual parameter replaces the second dummy parameter, and soon.

A macro call has the form

%Macro name (Actual parameter list) with the actual parameters being separated bycommas.

%MULTIPLY (CX, VAR, XYZ[BX]

Above macro call results in following set of codes.

```
PUSH DX
    PUSH AX MOV AX,CXIMUL VAR
    MOV XYZ[BX],AXPOP AX
    POPDX
```

NESTED MACROS

It is possible for a macro call to appear within a macro definition. This is referred to as **Macro nesting.** The limitation of nested macros is that all macros included in the definition of a given macro must be defined before the given macro is called.

1.6 INTERRUPT AND INTERRUPT SERVICE ROUTINES

INTERRUPT AND ITS NEED

The microprocessors allow normal program execution to be interrupted in order to carry out a specific task/work.

The processor can be interrupted in the following ways

- by an external signal generated by a peripheral,
- by an internal signal generated by a special instruction in the program,
- by an internal signal generated due to an exceptional condition which occurs while executing an instruction. (For example, in 8086 processors, divide by zero is an exceptional condition which initiates type 0 interrupt and such an interrupt is also called execution).

In general, the process of interrupting the normal program execution to carry out a specific task/work is referred to as interrupt.

When a microprocessor **receives an interrupt signal it stops executing current normal** program, **save**the status (or content) of various registers (IP, CS and flag registers in case of 8086) in stack and then the processor executes a **subroutine/procedure** in order to perform the specific task/work requested by the interrupt. The subroutine/procedure that is executed in response to an interrupt is also called Interrupt Service Subroutine. (ISR). At the end of ISR, the stored status of registers in stack is **restored to respective registers**, and the processor resumes the normal program execution from the point {instruction) where it was interrupted.

The **external interrupts are used to implement interrupt driven data transfer scheme**. The interrupts generated by special instructions are called **software interrupts** and they are used to implement system services/calls (or monitor services/calls). The system/monitor services are procedures developed by system designer for various operations and stored in memory. The user can call these services through software interrupts. The interrupts generated by exceptional conditions are used to implement error conditions in the system.

INTERRUPT DRIVEN DATA TRANSFER SCHEME

The interrupts are useful for efficient data transfer between processor and peripheral. When a peripheralis ready for data transfer, it interrupts the processor by sending an appropriate signal. Upon receiving an interrupt signal, the processor suspends the current program execution, save the status in stack and executes an ISR to perform the data transfer between the peripheral and processor. This type of data transfer scheme is called interrupt driven data transfer scheme.

The data transfer between the processor and peripheral devices can be implemented either by polling technique or by interrupt method. In polling technique, the processor has to periodically poll or check the status/readiness of the device and can perform data transfer only when the device 'is ready. In polling technique the processor time is wasted, because the processor has to suspend its work and check the statusof the device in predefined intervals. Alternatively, if the device interrupts the processor to initiate a data transfer whenever it is ready then the processor time is effectively utilized because the processor need not suspend its work and check the status of the device in predefined intervals.

CLASSIFICATION OF INTERRUPTS

In general the interrupts can be classified in the following three ways:

- Hardware and software interrupts
- Vectored and Non Vectored interrupt
- Maskable and Non Maskable interrupts.

The interrupts initiated by external hardware by sending an appropriate signal to the interrupt pin of theprocessor is called hardware interrupt.

HARDWARE AND SOFTWARE INTERRUPTS

The 8086 processor has two interrupt pins INTR and NMI. The interrupts initiated by applying appropriate signal to these pins are called hardware interrupts of 8086.The software interrupts are program instructions. These instructions are inserted at desired locations in a program. While running a program, if software interrupt instruction is encountered then the processor initiates an interrupt. The 8086 processor has 256 types of software interrupts. The software interrupt instruction is INT n, where n is the type

number in the range 0 to 255.

VECTORED AND NON VECTORED INTERRUPT

When an interrupt signal is accepted by the processor, if the program control automatically branches to a specific address (called vector address) then the interrupt is called vectored interrupt. The automatic branching to vector address is predefined by the manufacturer of processors. (In these vector addresses the interrupt service subroutines (ISR) are stored). In non-vectored interrupts the interrupting device should supply the address of the ISR to be executed in response to the interrupt.

All the 8086 interrupts are **vectored interrupts**. The vector address for an 8086 interrupt is obtained from a vector table implemented in the first 1kb memory space (00000h to 03FFFh).The processor has the facility for accepting or rejecting hardware interrupts.

MASKABLE AND NON MASKABLE INTERRUPTS

Programming the processor to reject an interrupt is referred to as masking or disabling and programming the processor to accept an interrupt is referred to as unmasking or enabling. In 8086 the interrupt flag (IF) can be set to one to unmask or enable all hardware interrupts and IF is cleared to zero to mask or disable a hardware interrupts except NMI.

The interrupts whose request can be either accepted or rejected by the processor are called maskable interrupts. The interrupts whose request has to be definitely accepted (or cannot be rejected) by the processor are called non-maskable interrupts. Whenever a request is made by non-maskable interrupt, the processor has to definitely accept that request and service that interrupt by suspending its current program andexecuting an ISR. In 8086 processor all the hardware interrupts initiated through INTR pin are maskable by clearing interrupt flag (IF). The interrupt initiated through NMI pin and all software interrupts are non-maskable.

SOURCES OF INTERRUPTS IN 8086

An interrupt in 8086 can come from one of the following three sources.

1. One source is from an external signal applied to NMI or INTR input pin of the processor. The interrupts initiated by applying appropriate signals to these input pins are called hardware interrupts.

2. A second source of an interrupt is execution of the interrupt instruction "INT n", where n is the type number. The interrupts initiated by "INT n" instructions are called software interrupts.

3. The third source of an interrupt is from some condition produced in the 8086 by the execution of an instruction. An example of this type of interrupt is divide by zero interrupt. Program execution will be automatically interrupted if you attempt to divide an operand by zero. Such conditional interrupts are also known as exceptions.

The 8086 microprocessor has 256 types of interrupts. The type numbers are in the range of 0 to 255. The 8086 processor has dual facility of initiating these 256 interrupts. The interrupts can be initiated either by executing "INT n" instruction where n is the type number or the interrupt can be initiated by sending an appropriate signal to INTR input pin of the processor.

For the interrupts initiated by software instruction" INT n ", the type number is specified by theinstruction itself. When the interrupt is initiated through INTR pin, then the processor runs an interrupt acknowledge cycle to get the type number. (i.e., the interrupting device should supply the type number through D0- D7 lines when the processor requests for the same through interrupt acknowledge cycle). The kinds of interrupts and their designated types are summarized in the figure below, by illustrating the layout of their pointers within the memory. Only the first five types have explicit definitions; the other types may be used by interrupt instructions or external interrupts.

From the figure it is seen that the type associated with a division error interrupt is 0. Therefore, if a division by 0 is attempted, the processor will push the current contents of the PSW, CS and IP into the stack, fill the IP and CS registers from the addresses 00000 to 00003, and continue executing at the addressindicated by the new contents of IP and CS. A division error interrupt occurs any time a DIV or IDIV instruction is executed with the quotient exceeding the range, regardless of the IF (Interrupt flag) and TF (Trap flag) status.

The type 1 interrupt is the single-step interrupt (Trap interrupt) and is the only interrupt controlled by the TF flag. If the TF flag is enabled, then an interrupt will occ

at the end of the next instruction that will causea branch to the location indicated by the contents of 00004H to 00007H.The single step interrupt is used primarily for debugging which gives the programmer a snapshot of his program after each instruction is executed

.

[Source: Advanced Microprocessors and Microcontrollersby A.K Ray & K.M. Bhurchandi]

The type 2 interrupt is the nonmaskable 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 nonmaskable interruptpin.

The remaining interrupt types correspond to interrupts instructions imbedded in the interrupt program orto external interrupts. The interrupt instructions are summarized below and their interrupts are not controlled bythe IF flag. IRET is used to return from an interrupt service routine. It is similar to the RET instruction except that it pops the original contents of the PSW from the stack as well as the return address. The INT instruction has one of the forms INT or INT Type. The INT instruction is also often used as a debugging aid in cases where single stepping provides more detail than is wanted. By inserting INT instructions at key points, called breakpoints. Hence the 1 byte INT instruction (Type 3 interrupt) is also referred to as breakpoint interrupt.

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 condition does not cause an interrupt automatically; the interrupt must be explicitly specified by the INTO instruction. The remaining interrupt types correspond to interrupts instructions imbedded in the interrupt program or to external interrupts.

1.7 STRING MANIPULATION INSTRUCTIONS

A series of data byte or word available in memory at consecutive locations, to be referred as Byte String or Word String. A String of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent. The 8086 supports a set of more powerful instructions for string manipulations for referring to a string, two parameters are required.

I. Starting and End Address of the String.
II. Length of the String.

The length of the string is usually stored as count in the CX register. The incrementing or decrementing of the pointer, in string instructions, depends upon the Direction Flag (DF) Status. If it is a Byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation, the index registers are updated by two.

REP: Repeat Instruction Prefix

This instruction is used as a prefix to other instructions, the instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one). i. REPE / REPZ - repeat operation while equal / zero. ii. REPNE / REPNZ - repeat operation while not equal / not zero. These are used for CMPS, SCAS instructions only, as instruction prefixes.

MOVSB / MOVSW: Move String Byte or String Word

Suppose a string of bytes stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of source string is located in the memory location whose address may be computed using SI (Source Index) and DS (Data Segment) contents. The starting address of the destination locations where this string has to be relocated is given by DI (Destination Index) and ES (Extra Segment) contents.

Example: Block Transfer program using the move string instruction

```
MOV AX, DATA SEG ADDR
 MOV DS, AX
MOV ES, AX
MOV SI, BLK 1 ADDR
 MOV DI, BLK 2 ADDR
MOV CX, Count
CDF
REP MOVSB
HLT
```

CMPS: Compare String Byte or String Word

The CMPS instruction can be used to compare two strings of byte or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero Flag is set. The REP instruction Prefix is used to repeat the operation till CX (counter) becomes zero or the condition specified by the REP Prefix is False.

SCAN: Scan String Byte or String Word

This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The String is pointed to by ES: DI register pair. The length of the string stored in CX. The DF controls the mode for scanning of the string. Whenever a match to the specified operand is found in the string, execution stops and the zero Flag is set. If no match is found, the zero flag is reset.

LODS: Load String Byte or String Word

The LODS instruction loads the AL / AX register by the content of a string pointed to by DS: SI register pair. The SI is modified automatically depending upon DF, If it is a

byte transfer (LODSB), the SI is modified by one and if it is a word transfer (LODSW), the SI is modified by two. No other Flags are affected by this instruction.

STOS: Store String Byte or String Word

The STOS instruction Stores the AL / AX register contents to a location in the string pointer by ES: DI register pair. The DI is modified accordingly, No Flags are affected by this instruction. The direction Flag controls the String instruction execution, The source index SI and Destination Index DI are modified after each iteration automatically. If DF=1, then the execution follows auto decrement mode, SI and DI are decremented automatically after each iteration. If DF=0, then the execution follows auto increment mode. In this mode, SI and DI are incremented automatically after each iteration.

AUTO INDEXING FOR STRING INSTRUCTIONS:

SI & DI addresses are either automatically incremented or decremented based on the settingof the direction flag DF. When CLD (Clear Direction Flag) is executed DF=0 permits auto increment by 1. When STD (Set Direction Flag) is executed DF=1 permits auto decrement by 1.