MODULE - 1

8086 Microprocessor

Madhuri. J
Asst. Professor
Dept of CSE

Salient features

(a) Intel 8086 is the first 16-bit processor released in the year 1978. 8086 was designed using HMOS (Hi speed) technology. and has 40 pins.

(b) 8086 requires an external clock source with 33% duty cycle.

(c) 8086 has 16 bit data bus and 20 bit address bus, it can directly address upto 1 MByte of memory space.

(d) It can generate 16-bit address for I/O devices. Hence it can generate 64K (2^{16}) I/O addresses.

(e) 8086 has fourteen 16-bit registers.

(f) 8086 operates in two modes maximum mode and minimum mode.

In minimum mode, the system is called uniprocessor system.

In maximum mode, the system is called multiprocessor system.

Pipelining

CPU processing can be made faster by:

→ Increasing working frequency.

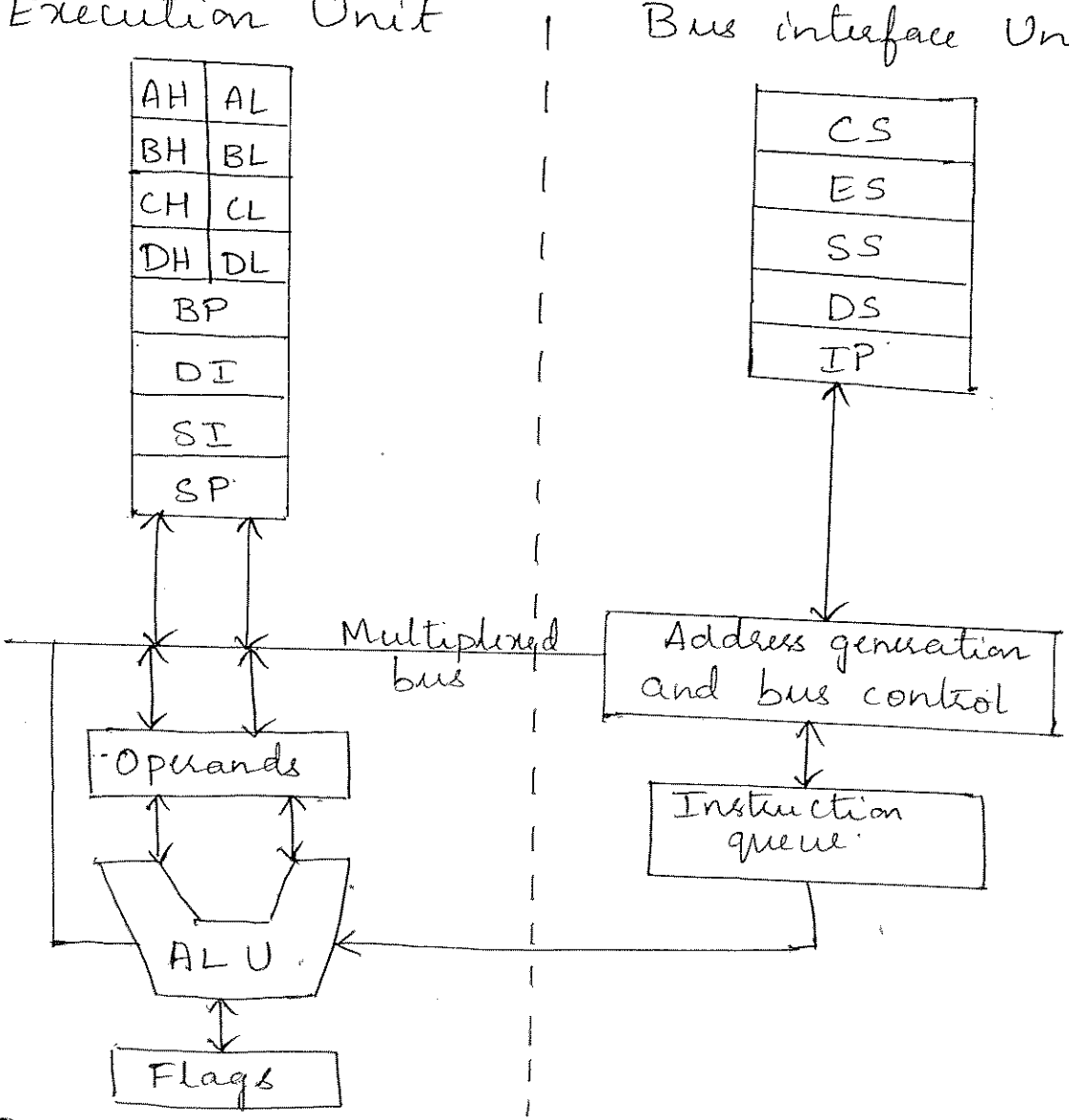
→ Changing internal architecture of the CPU.

The technology and the materials used in making IC's (Integrated circuits) determine the working frequency.

execute it.

In 8086, idea of pipelining is used where the CPU can fetch and execute at the same time.

Internal architecture of 8086 is as follows:
It consists of two sections: the execution unit (EU) and bus interface unit (BIU)



These sections work simultaneously.

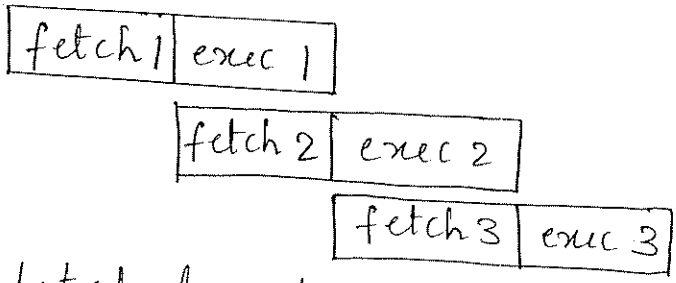
Bus interface unit (BIU) is the interface to the outside memory and peripherals.

1. It sends addresses to memory & I/O devices
2. It fetches instructions from memory.
3. It supports instruction queuing.

Instruction queue

Instruction queue of 8086 is 6-bytes long. The instructions are fetched ahead of execution unit from the memory and are queued up. This helps in speeding up of execution termed as pipelining.

Pipelining.



Instruction 2 is fetched when inst 1 is being executed and so on.

* Whenever a jump instructions are executed, the BIU flushes out the instruction queue and a new instruction is fetched from new location in memory. In this situation EU is idle. This is referred to as branch penalty.

Segment registers CS, ES, SS, DS holds the starting address of the segments used.

Execution Unit

→ EU executes the instructions in the instruction queue. ALU performs the arithmatic and logical operations. Registers stores the information temporarily. There are nine flags in 8086.

Registers

Information stored in the registers can be one or two bytes. The registers can be grouped into four categories.

General Purpose	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL DH, DL.
Pointer	16	SP (stack pointer) BP (Base pointer)
Index	16	SI (source index) DI (destination inde
Segment	16	CS, DS, ES, SS.
Instruction pointer	16	IP
Flag	16	FR (Flag register

The general purpose registers in 8086 microprocess can be accessed as either 16 bit or 8 bit registers.

AX → Accumulator

AH AL

BX → Base addressing regist

CX → Counter in loops

DX → Points to data I/O operations.

Program segments:

→ With 20 address lines, the maximum addres memory of 8086 is 2^{20} = 1MB.

→ The total memory of 1MB is divided into 16 64KB blocks. Each 64KB block of memory is referred to as a segment

→ Each segment begins on an address evenly divisible by 16

→ An assembly language program consists of atleast three segments.

: a code segment
data segment
stack segment.

Code segment consists of assembly language instructions that performs the tasks that the program was designed to accomplish.

Data segment is used to store information (data) that needs to be processed by the instructions in the code segment.

Stack segment is used by the CPU to store information temporarily.

→ These segments occupy atmost three of the memory segments of 64k each.

→ Hence 8086 can handle a maximum of 64KB of code, 64KB of data and 64KB of stack at any given time

Logical address and Physical address.

In 8086, there are three types of addresses
→ the physical address
→ the offset address
→ the Logical address.

Physical address is the 20 bit address on the address pins of 8086 MP. This address is in the range of 00000h to FFFFFh.

Offset address is a location within a 64-kbyte segment range. It ranges from 0000h to FFFFh.

Logical address consists of a segment value and

Code segment consists of the instructions th has to be executed.

→ 8086 fetches instructions from the code segme

→ Logical address consists of CS (code segment) a an IP (offset address). This is shown in the format CS:IP.

→ Physical address is generated by shifting th Code segment (CS) i.e the segment value to left and adding the value of IP to it.

→ The resulting physical address is 20 bit add

Consider, CS = 2500 IP = 95F3h.

Logical address = 2500 : 95F3 CS:IP.

Physical add^{n} is calculated as follows

CS | 2 | 5 | 0 | 0 |

Shift left CS | 2 | 5 | 0 | 0 | 0 |

Add IP | 9 | 5 | F | 3 |

$$\begin{array}{r} 25000 \\ +\ 95F3 \\ \hline 2E5F3 \end{array}$$

Physical address = 2E5F3.

CS holds the starting address of code segment

If CS = 2000

starting address of code segment, offset = 0000

i.e, 20000h + 0000h.

= 20000h.

Ending address of code segment, offset = FFFF

i.e, 20000h + FFFFh

= 2FFFFh.

Ending address offset is FFFFh because each each segment size is 64K (FFFFh)

IP is the default reg^{r} to carry the offset address of code segment.

Data segment

→ The area of the memory that stores only the data is called the data segment.

→ Data segment uses the register DS for starting address. Offset value or address is stored in BX DI, SI or 8, 16 bit number.

The term pointer is used to for a register holding an offset address.

Consider the example.

DS = 5000.

MOV BX, 0200h.

Add AL, [BX]

14	0202h
13	0201h
12	200h

Here Bx is enclosed in the square brackets. The brackets indicate, the operand represents the address of the data and not the data itself

AL = 0 after, add AL, [BX]

AL + 12

Therefore AL = 12h

Physical address of data segment.

→ Physical address is calculated by shifting DS one hex digit left and adding the offset value.

Mov ax, [BX] DS = 5000h BX = 200h

Physical address = 5000 x 10h + BX.

= 50000 + 200 = 50200h

Contents of address 50200h is moved to AX.

Little endian convention.

Mov ax, 53F5

mov [1500], AX.

Low byte of the data, goes to low memory. High byte goes to high memory address.

→ Stack is a section of read/write memory used by the CPU to store information temporarily. As the no of registers is limited, CPU needs this stor area to store data temporarily.

→ The disadvantage of stack is its access time.

→ Two main registers used to access stack are SS (stack segment) and SP (stack pointe

→ The stack pointer (SP) points at the current memory location used for the top of the stack After the data is pushed onto the stack, The stack pointer SP is incremented
When the data is popped off the stack, the stack pointer (SP) is decremented.

→ Any register ~~can be~~ Except SP & segment reg can be pushed or popped from the stack.
It has to be entire 16-bit register.

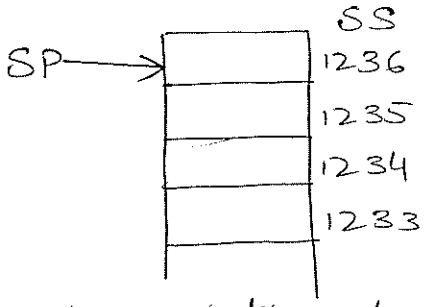
Eg Push AX is valid.
Push AL is invalid as AL is 8 bits.

→ The SP points to top of the section and it gr downwards from upper address to lower address. This is opposite to the instruction point which gets incremented after each instruction is executed. This is ensure that code section and stack section never write over each other, hence they are located at opposite ends of memory.

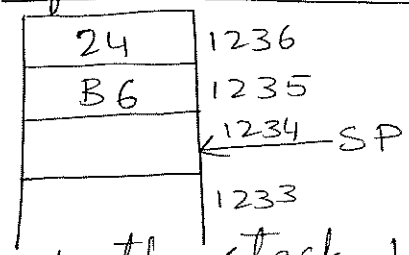
Pushing onto the stack.

Consider SP = 1236 , AX = 24B6, DI = 85C2.

1. PUSH AX

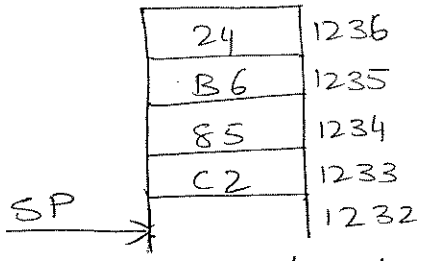


After PUSH AX.



Initially SP is pointing to top of the stack 1236 after PUSH AX, the SP is decremented by 2 and is pointing to 1234 (top of the stack).

2. PUSH DI

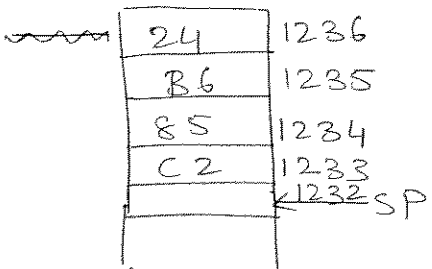


Initially SP is pointing to top of the stack 1234 after PUSH DI, SP decrements by two & points at 1232

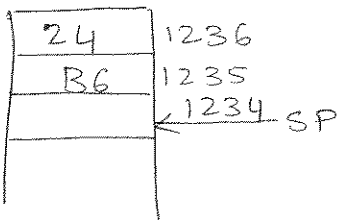
Popping of the stack

Consider SP = 1232

After every POP, the stack pointer decrements by 2 and the contents on 2 bytes on the top of the stack are copied to the register specified.



After POP AX



AX = 85C2.

Initially SP is pointing to 1232, after POP AX the contentints on the top are transferred to AX and SP is decremented by 2.

→ Locations A0000h to BFFFh is set for vide
The amount of memory used and location vary
depending on the video board installed.

ROM

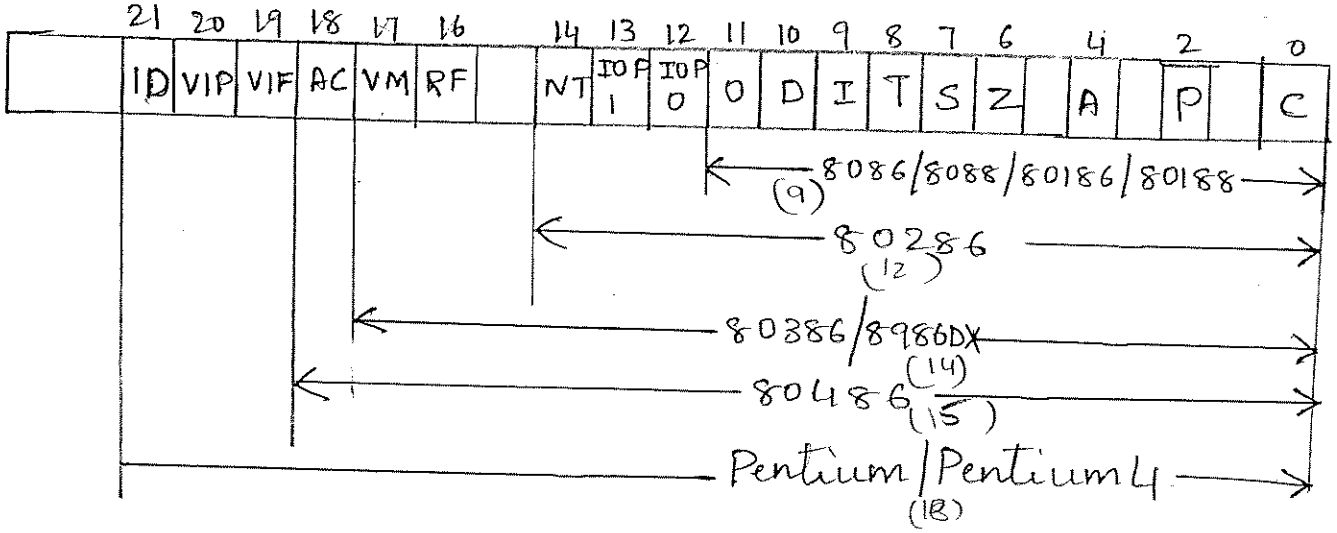
→ Total 256K bytes is assigned for ROM. Out of the
only 64K bytes are used by BIOS (basic i/p/o/p sy
ROM. Remaing space is used by various adapter
cards.
→ Permanent (non volatile) memory holds the proc
to tell the CPU what to do when the pow
is on.
→ BIOS RAM contains programs to test the compo
connected to CPU and allows the window
to communicate with peripheral devices suc
as keyboards, video, printer and disk.
→ BIOS tests all the devices connected to the PC
when the computer is turned on and reports
any errors.

Flag Register

→ Flag register is a 16-bit register also
referred as the status register.
→ There are nine flags in 8086
→ Six of the flags are conditional flags whic
indicates some condition that resulted after
an instruction execution.

·RSP (Stack Pointer): RSP also refferred as ESP, SP addrees the stack. The stack memory stores the data through this pointer.

RFlags: Flags indicate the condition of the microprocessor and control its operation.



C (Carry): Holds the carry after addition or the borrow after subtraction.

P (Parity): Parity is the count of number of ones in a number. Parity is 0 for odd parity and 1 for even parity Eg: 00011100 - It is odd parity as there are 3 ones.

A (Auxillary): It holds the carry and borrow for addition and subtraction between the bit positions 3 and 4.

Z (zero): Z=1 if the result of arithmetic or logical operation is zero and Z=0 if result is not zero.

is set or negative, if S=0, the sign bit is cleared or positive.

T (Trap): Trap flag enables traping through an on-chip debugging feature. CPUgm is debug indicated by
If T=1 — The microprocessor interrupts the flow of program as debug registers & cntl regis
If T=0 trapping is disabled.

I (Interrupt): Interrupt flag controls the operation of the INTR. If I=1, INTR is enabled. If I=0, INTR is disabled.
Eg: STI (set I), CLI (clear I).

D (Direction): Direction flag sets the DI / SI registers in increment or decrement mode during string instructions.
If D=1, registers are decremented.
D=0, registers are incremented.
Eg: STD (set D), CLD (clear D)

O (Overflow): An overflow indicates the result has exceeded the capacity of machine.
O=1, when overflow occurs.

IOPL (I/O privilege level): IOPL is used to set the privilege level for I/O devices.

BANGALORE INSTITUTE OF TECHNOLOGY

DATA SHEET

Name :

Title :

Branch :

Batch :

Date :

=Large register affected

MOV BH, 38h.

ADD BH, 2Fh.

```
 38      0011  1000        CF = 0      ZF = 0
+2F      0010  1111        AF = 1      SF = 0.
---------------------       PF = 0
         0110  0111
```

MOV AX, ~~2h~~ 34F5h.

ADD AX, 95EBh.

```
0011 0100 1111 0101
1001 0101 1110 1011
-------------------
1100 1010 1110 0000
```

CF = 0 ZF = 0

AF = 1 SF = 1.

PF = 0

Register
immediate
direct.
register indirect.

based relative
indexed relative.
based indexed relative.

mov bx, AAAAh

ADD bx, 5556h.

```
1010 1010 1010 1010
0101 0101 0101 0101
-------------------
0000 0000 0000 0000
```

CF = 1 AF = 1 ZF = 1

PF = 1 ZF = 1 SF = 0

Addressing modes.

Madhuri-J
Dept of CSE

Data addressing modes:

The data addressing modes are explained with the MOV instruction.
The data addressing modes are as follows:

Register addressing:

Transfers a copy of a byte or word from the source register ~~or memory~~ to destination register ~~or memory~~. The registers used must be of same size. Segment-to-segment register addressing is not allowed.

Examples.

MOV AL, BL → Copies BL into AL.
MOV AX, CX
MOV SP, BP
MOV DS, AX
MOV ECX, EBX (80386)

Immediate addressing.

Transfers the immediate byte or word of data into the destination register or memory. Immediate data are the constant data.

Eg: MOV EAX, 13456h → Copies the immediate data 13456 into the register EAX
MOV BL, 44h
MOV CH, 100 → Copies a 100 decimal (64H)

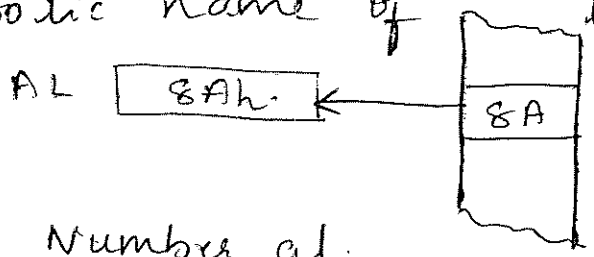
3. Direct addressing.

It moves a byte or word between memory location and the registie. The two form of direct data addressing is,

1. Direct addressing 2. Displacement addres

Direct addressing is used between the memory location and AL, AX or EAX.

Eg: MOV AL, DATA. → loads AL with data seg memory location data (1234h). Data is the symbolic name of the actual location (12



MOV Number, al.
MOV sum, EAX.

Displacement addressing.

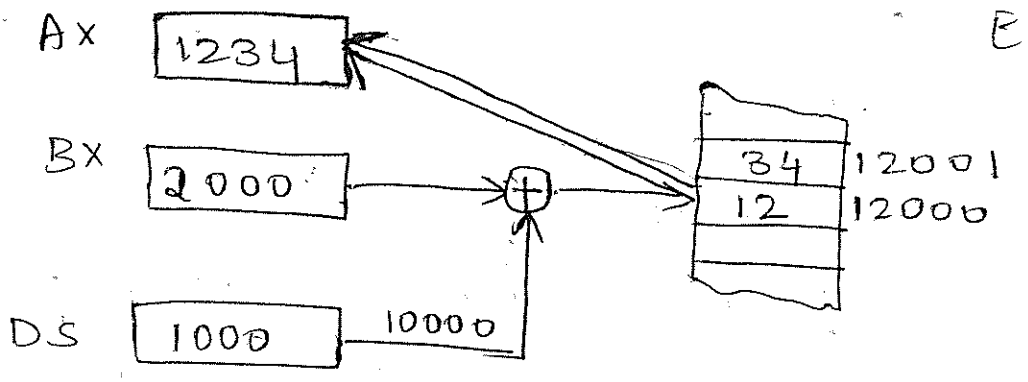
Displacement addressing is four byte wide. Eg: MOV CH, [1234h] → The data poi by DS x 10 + 1234h (disp) is transfered t CH registie. MOV CH, CDS: [1234h]

4. Registie Indirect addressing.

Registie indirect addressing all data to be addressed at any memory locati through an offset address held at any of the following registies: BP, BX, DI and SI.

Eg: MOV AX, [BX] → BX contains the offset addres 2000h. So the head address is 10000 + 2000h = 12000h. The contents of the DS = 1000h

location [12000h] is moved to AX.



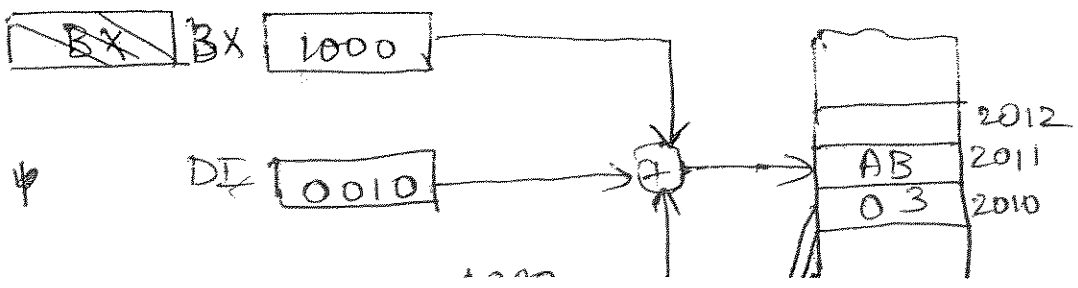
Eg: MOV CX, [BX]
MOV [BP], DI
MOV [DI], BH
MOV ECX, [EBX]

Data segment uses the default registers, BX, DI, SI with registe indirect addressing.
BP is used by the stack segment.

Base plus index Addressing.

Base plus index addressing because it addresses memory indirectly. It uses one base registe (BP or BX) and one index registe (DI or SI) to indirectly address memory. The base registe holds the beginning location of memory, while index registe holds relative position.

Eg: MOV DX, [BX + DI]. If BX = 1000h, DI = 0010h and DS = 0100h, the memory address is 01000 + 1000 + 0010 = 02010h. So the data from memory location 02010h is moved to DX.



Eg: MOV [BX+DI], SP

MOV CH, [BP+SI]

MOV [EAX+EBX], ECX.

6 Register Relative addressing.

In register relative addressing, d
data in the memory are addressed by addi
the displacement to the contents of base or a
index register

Eg: MOV BAX, [BX+1000h]

BX = 0100h, DS = 0200h. Addressed generated is

02000 + 0100 + 1000 = 03100h.

A displacement can also be the offset addres
appended to the front of [ ].

Eg: MOV al, data [DI] → Memory locati
is addressed by the sum of data + DI and
the contents are moved to al.

Eg: MOV AX, [DI+100h]

MOV List [SI+2], cl.

7. Base relative plus index addressing.

Here, the data is addressed by usin
base register + index register and the displac

Eg: MOV AX, [BX + SI + 100h] BX = 0020
SI = 0010
DS = 10000

∴ EA = 0020+0010+10000+100
= 10130h.

Eg: Mov dh, [BX+DI+20h]

MOV ax, File [BX+DI] → memory location is addressed by the sum of file + BX+DI and contents are moved to ax.

Scaled index addressing.

This addressing mode is present in 80386 through Pentium-4. Scaled index addressing uses a base register and the index register to access memory. The second register is multiplied by a scaling factor. The scaling factor can be 1x, 2x, 4x, 8x.

1x scaling factor is implied and need not be included Eg: MOV AL, [EBX+ECX]

Scaling factor 2x is used when memory is word sized. Eg: MOV AX, [EDI + 2*ECX]

4x is for double word sized memory arrays

8x is for quad word sized —11—

Program memory addressing modes.

Consists of three distinct forms: direct, relative and indirect.

Direct program memory addressing

The instructions with direct program memory addressing stores the address with the opcode. Usually the memory address is stored as the label. Eg: JMP NEXT

The term relative means "relative" to the instruction pointer (IP).

Eg: JMP [2] → The address in relation to the instruction pointer is a 2 that adds to the instruction pointer: One byte displacement is used in short jump. Two byte displacem is called near jump. Both types are consid intrasegment jumps.

Relative JMP and CALL instructions conta either 8-bit or 16-bit displacements.

Indirect Program memory addressing.

Program Indirect JMP and CALL instruc uses any 16-bit register or any register relative with displacement.

Eg: JMP AX → jumps to current code segment location addressed by contents of

- JMP TABLE[BX]

STACK memory addressing modes.

Eg: PUSHF
POPF
Push AX
Pop BX
Push DX

Assembly language programming

→ Assembly language has the following fields:
Label : mnemonic [operands] [; comments]

→ Assembly language program is a series of lines or statements which are instructions or the directives.

→ The assembly language mnemonic (instruc^n) and operands field together perform the work of the program and accomplish the tasks for which program was written. Machine code is generated for the instructions.

→ Directives give directions to the assembler about how it should translate assembly language instructions into machine code. They are used by the assembler to organize the program as well as other output files. Machine code is not generated

→ Comment field begins with a ";".

Model definition

– The Model directive – selects the size of the memory model. MODEL selects among the following memory model are SMALL, MEDIUM, COMPACT, LARGE

· MODEL SMALL : This directive defines small model. The small model uses a maximum of 64K bytes of memory for code and another 64K bytes for data.

· MODEL TINY : Data and code must fit into 64K bytes

but code can exceed 64K Bytes

- MODEL LARGE : Both data and code can exceed 64K bytes

Segment definition

The X86 CPU has four segments registers. CS (code segment), DS (data segment), SS (stack segment), ES (extra segment)

→ Assembly language statements are grouped into segments in order to be recognized by the assembler and consequently by the CPU.

→ The program normally consists of at least three segments: The stack segment, code segment, data segment

- .STACK ; marks beginning of the stack segment
- .DATA ; marks beginning of data segment
- .CODE ; marks beginning of code segment

→ One can write an assembly language program that uses only one segment.

Assemble, Link and Run the program.

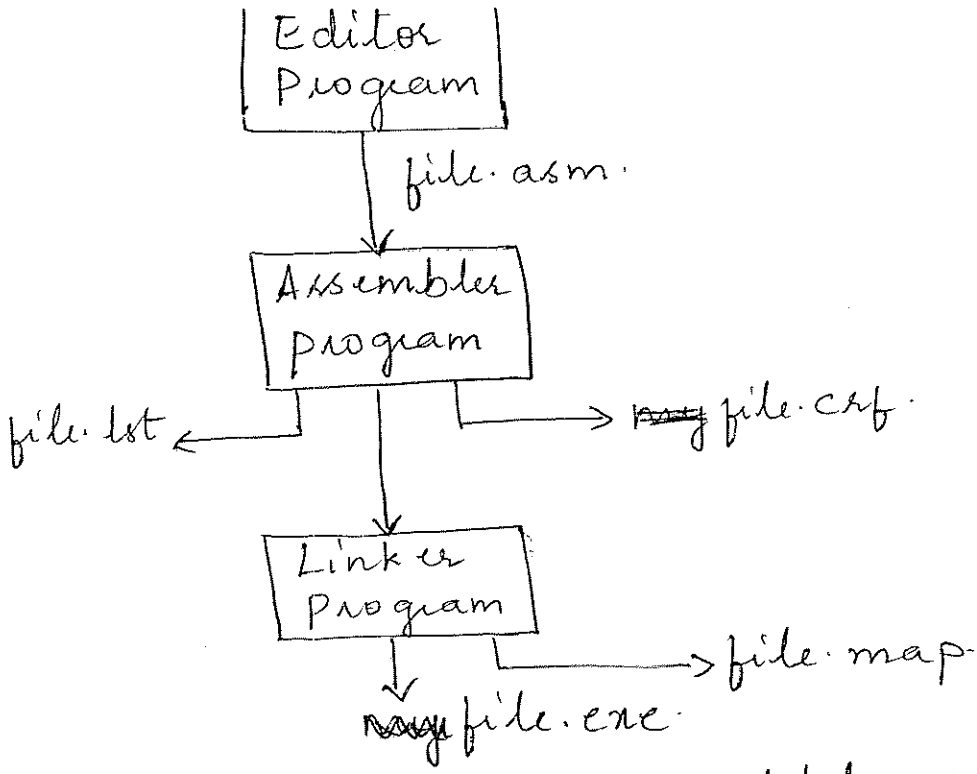
The three steps to create an executable assembly language program are outlined as follows.

Step	Program	output
1. Edit the program	Editor	File.asm
2. Assemble the program	TASM	File.obj
3. Link the program	TLINK	File.exe

· map file.

- When there are many segments for code and data, we can see where each segment is located in the memory and how many bytes are used by each segment.

Madhuri. I
Dept of CSE
BIT



→ .ASM source file is assembled using tasm, the assembler will produce an object file and a lst file along with other files that may be useful to the programmer.

→ Object file with the extension .obj is the input to the LINK program, that produces an executable program that ends in .exe

.lst file

— The .lst file is very useful to programmers because it lists all the opcodes and offset addresses as well as errors TASM detected.

e> myfile.lst

.crf file

— The .crf file provides an alphabetical list of all symbols and labels used in the program as well as program line numbers in which they are referenced.

Real mode and Protected mode.
With we have .486 & .486P for 80486 inst set
.586 and .586P for Pentium instruction set

1. Page & title.

This directive help to control format of assembled program.

Page: The page directive at the start of the program specifies the maximum number of lines to list on a page and maximum no of characters on each line.

Format: Page [length], [width]

Title: It is to write the title of the program on each Page.

Format: Title text

2. .Model.

There are six models ranging from tiny to huge. To designate a model use .model followed by the size of memory system

Tiny model – requires code and data fit into one 64kB memory segment.

Small model – required only one data segment and one code segment for a total of 128kB.

3. .Startup – If .startup directive is used, the mov ax, @data followed by mov ds, ax can be

4. Segment and ends. (Full segment definition)

It is more structured than the mod method. These directives are used to identify a group of data items or instructions. SEGMENT indicates start of segment & ENDS indicates end of segment.

Format: name SEGMENT
:
name ENDS

Eg: code SEGMEN
:
code ENDS

5. PROC & ENDP

→ This directive indicate the start and of the procedure. PROC and ENDP require a la to indicate the name of the procedure.

→ The PROC directive indicating start of procedure must be followed by NEAR OR F

Format: name PROC near/far

→ Near procedure resides in the same co segment. Far procedure may reside in any location in the memory system.

→ Often near procedure is called local an far procedure is called global.

6. ASSUME

→ ASSUME directive tells the assembler th what names have to be chosen for code, d stack, extra segments. A

→ ASSUME statement is only used with fu segment definitions.

Eg: ASSUME CS: code ; CS link code area

7. EQU

→ Equate (EQU) directiv equates a numeric, ASCII to labd. Equate makes the program clear & simple.

Eg.: PI EQU 3.1415.

8. ORG

→ ORG assemble directive is used to set the starting of the memory allotment of the segment.

Eg: Data segment
ORG 300h
Data ends.

→ Here ORG ensures that the data in the data segment originates from the offset 300h onwards.

9. DUP

→ Used to initialise several locations to duplicate basic data definition n no of times

format: name datatype no DUP (value)

Eg: Array dB 10 dup(10).

→ Data definition directives — They create storage space at the assembly time.

DB → define byte
DW → define word.
DD → Double wad.

10. EXTRN.

→ This directive infams the assemblu that the name a label following this directive is present in some othe module.

11. PUBLIC.

→ So when we [illegible] label can be seen (used) by other modules.

Eg: for PUBLIC & EXTERN.

```
.model small
.data
  EXTERN array :FAR
.code
  EXTERN readkB : FAR.
   mov ax, @data
   mov ds, ax
   mov cx, 0Ah
   mov si, offset array.
next: Call readkB
   mov [SI], al
   inc SI
   loop next
   mov
END
```

```
.model sma
.data
  Public arra
  array dB 100 dup C
.code
Public readkB
readkB proc
  mov ah, 07h
  int 21h
ReadkB endP
END.
```

TLink 1a+1b

12. MACRO and ENDM:

→ A macro is a group of instructions that perform one task.

→ All instructions defined in a macro is inserted in the program at the point of usage

→ MACRO and ENDM marks the starting and end of the macro sequence.

Eg: Name1 MACRO A, B
:
ENDM.

Macro name is Name1, parameters used - A & B.
No label is placed before & after ENDM.

13. OFFSET

→ It is used to refer to the memory indirectly.

Eg: mov Bx, offset data

The above instruction copies the offset address of data into BX.

14. LENGTH.

→ This is used to refer to the length of the data array or string.

Eg: MOV cx, length array → This on execution will substitute the length of the array in bytes to Cx register.

15. LOCAL

Variables/constants or procedures declared local in a module are to be used only by that module.

Eg: LOCAL a, b, data, array.

16. SHORT.

→ Indicates that only one byte is required to code the displacement for a jump (−128 to +127 bytes) otherwise assembler may reserve two bytes for displacement Eg: JMP SHORT label.

17. PTR

→ The operator PTR is prefixed by byte or word.

→ If prefix is byte then that variable or memory operand is 8 bit quantity.

Eg: Mov al, byte PTR[SI] ; moves memory location addressed by SI (8 bit) to al.

INC byte PTR[BX] ; Increments by a byte.

	Procedure.	Macro
1.	Accessed by call and RET instruction.	Accessed during assembly time.
2.	Machine code is generated only one and placed in memory	Machine code is generated each time macro is called.
3.	Less memory	Occupies more memory.
4.	Execution is slow since it involves PUSH & POP	Execution is fast.
5.	Uses stack & cannot define memory locations locally.	Do not use stack & can define memory location locally.

Program control Instructions.

- These are also called program execution transfer instructions or branch instructions.

1. JUMP. (jump)

JMP allows the programmer to skip section of a program and branch to any part of the memory for the next instruction.

(a) Unconditional jump (JMP)

↔ unconditionally jumps to a specified addr to get the next instruction.

→ Three types of JMP instructions

(a) Short jump - It is a 2-byte instruction that allows jumps or branches to memory locations within +127 to -128 bytes from the address following the jo

(b) Near jump — Allows branch or jump within ±32kB from the instruction in the current code segment. 3 byte instruction.

(c) FAR jump — Allows jumps to any memory location within real memory system. 5 byte instruction

```
Eg: EXTRN up: FAR.  ; Different module.
    Start: ADD AX, 1
           JMP UP.
```

Near & short jumps are intrasegment, FAR jump is called inter segment jump.

Eg: JMP AX → Copies the contents of Ax register to IP when JMP occurs.

JMP CONTINUE → Jumps to label continue. Jump can be foward and backward.

Conditional Jumps.

→ Conditional jumps are always short jumps in 8086.

→ Conditional jumps, jump to a specified address with some conditions in flag bits.

→ The following flag bits are tested

sign (S), zero (Z), Carry (C), Parity (P), overflow (O)

→ Conditional jumps are used after CMP, ADD, SUB, INC, DEC instructions.

Eg: JC → will jump if carry bit is set.

Some of Conditional jump instructions.

(i) JA/JNBE : jump if above / jump if not below or equal.
i.e, CF=0, ZF=0

(ii) JAE/JNB : Jump if above or equal / jump if not

(3) JB / JNAE : CF = 1 and ZF = 0 " " or equal

(4) JBE / JNA : jump if below or equal / Jump if not above
CF = 0 and ZF = 1.

(5) JC : jump if CF = 1.

(6) JE / JZ : jump if equal i.e, ZF = 1.

(7) JNE : Jump is not equal i.e, ZF = 0.

(8) JO : Jump if overflow occurs : OF = 1.

(9) JP : Jump if parity flag is set P = 1.

(10) JG / JNLE : Jump on greate ZF = 0, CF

(11) JS jump if sign flag is set SF = 0.

(12) JL jump if less than SF !=

2. LOOP

→ Loop though a sequence of instructions unt CX = 0.

→ Loop instruct is the combination of decrement and JNZ conditional jump.

→ No of times loop iteration is required to be ex is placed in CX registe, everytime loop is executed CX is decremented by 1. Loop effects no flags.

Format : LOOP label.

Conditional loops

LOOP instruction has two conditional forms: LOOPE and LOOPNE.

LOOPE → Jumps if if CX != 0. It will exit the loop condition is not equal or if CX = 0.

LOOPNE → Exits if condition is equal or if CX=0

Procedures

→ Procedures are group of instructions that usually performs one task.

→ It is the reusable section of the software that is stored in memory once, but used as often as necessary.

→ The disadvantage is it takes some amount of time to link to the procedure and return from it.

CALL instruction and RET instruction

→ The CALL instruction transfers the flow of the program to the procedure.

→ It also saves the return address on to stack.

Near CALL → If the call is made to the procedure in the same segment as the calling program, then it is near call.

FAR CALL → If the call is to the procedure in the segment other than calling program, it is far call

Near CALL → 3 bytes long instruction. When call executes it pushes offset address of next instruction stored in IP to stack and transfers control to procedure

FAR CALL → 5 byte instruction. When executed places contents of both IP and CS on the stack.

Calls with reg operands: CALL BX → It pushes IP into stack and jumps to the offset address located in register BX.

Near RET — removes 16-bit number from the sta & places in IP

FAR RET — removes 32-bit no from stack an places them in IP and CS.

The new location is the address of the instruction that immediately follows the CALL instruction.

RET 6 — Here the stack ptr increments by the number after popping of IP from the stak. i.e, after executing RET 6, SP valu will be SP+6. This is to remove co local parameters created.

INTRODUCTION TO INTERRUPTS

An interupt is generated in three ways:

(a) Hardware interrupt — Derived from a hard signal. NMI, INTR line

(b) Software interrupt — Derived from execution an instruction. int xx

(c) Exceptions — result of execution of some instructions like divide by zero et

Interrupt is said to occur when currently exec program is interrupted.

Interrupt vector table

→ CPU processes the interupts using interrupt vector table (IVT) which has the addresses of interrupt service routines for different interrup

→ A interrupt vector is a four byte number stor in first 1024 bytes of memory in real mode.

→ In protected mode interrupt descriptor table used which has eight byte descriptors.

Types of Instructions. MODULE - 2

All instructions of 8086 can be put under one of the following types.

A) Data transfer instructions.

(i) General purpose byte/word transfer instructions
MOV, PUSH, POP, XCHG, XLAT

(ii) Special address transfer instructions.
LEA, LDS, LES.

(iii) Flag transfer instructions
LAHF, SAHF, PUSHF, POPF

(iv) Simple i/p o/p transfer instructions.
IN, OUT.

B) Arithmetic instructions.

(i) Addition
ADD, ADC, INC, AAA, DAA.

(ii) Subtraction.
SUB, SBB, DEC, NEG, DAS, AAS, CMP, CBW, CWD

(iii) Multiply.
MUL, IMUL, AAM.

(iv) Division.
DIV, IDIV, AAD

C) Bit manipulation instructions

(i) Logical
AND, NOT, OR, XOR, TEST

(ii) Shift.
SAL, SHL, SAR, SHR.

(Control transfer instructions).

(i) Conditional - JC, JNC, JZ, JNZ, JP, JPO
JG, JCXZ, JO, JS, Loop.

(ii) Unconditional - JMP, call, ret

E) Processor control instructions.

CLC, CMC, STC, CLD, STD, CLI, STI, HLT,
WAIT, LOCK, NOP

F) String manipulation instructions.

- Used with the prefix REP, REPE, REPNE
MOVS, CMPS, LODS, STOS, SCAS.

G) Interrupt instructions.

int xx, IRET, INT 3, INTO.

Unsigned addition and subtraction

Addition of unsigned numbers.

ADD distination, source ; distination = distination + source.

→ Distination operand can be a registre or in memory

→ Source operand can be registre, memory or immediate

→ ZF, SF, AF, CF, PF are the flags affected

Eg: ADD AX, BX.
ADD AL, 0Bh.

Add with carry (ADC)

ADC - add the bit in the carry flag (CF) to the operand data and result is stord in distination.

```
.model small
.data
count equ 05
data db 125, 235, 197, 91, 48.
SUM dw ?
.code
mov ax, @data
mov ds, ax
mov cx, count
mov si, offset data
mov ax, 00
back: add al, [si]
      JNC over
      INC AH
over: INC si
      DEC cx
      JNZ back
      MOV SUM, AX
      MOV AH, 4CH
      INT 21
      END
```

Replace by

```
ADD AL, [SI]
ADC AH, 00
```

Multiplication of Unsigned numbers.

→ In multiplication, we consider three cases

1. byte x byte.
2. word x word.
3. word x byte.

byte x byte — MUL operand1

→ One operand must be AL register and the second operand must be in a register or in memory as addressed by one of the addressing modes. After multiplication the result is in AX.

Eg:
```
mov al, 25h
mov bl, 65h
mul bl        ; AX = 25x65.
mov res, ax   ; product is saved.
```

word x word.

→ one operand must be in Ax and second can be in any reg or memory. The product will be stored in AX and DX registers.

→ Since the product can be 32-bit result, Ax will be hold the lower word and Dx the higher word.

Eg: Eg: MUL CX → [Ax * CX, product is in DX-Ax]

Unsigned multiplication summary.

Multiplication	Operand1	Operand2	Result
byte x byte	AL	reg/Mem	AX.
word x word	AX.	Reg/Mem	DX-AX
word x byte	AL=byte AH=0	Reg/Mem	Dx-Ax

Unsigned division Summary.

Division	Numerator	Denominator	Quotient	Remainder
Byte/Byte	AL=byte, AH=0	Reg or Mem	AL	AH
word/Word	AX = word	Reg or mem	AX	DX.
word/Byte	Ax = word	Reg or Mem	AL	AH

Logical Instructions

AND, OR, XOR, SHIFT, COMPARE

AND

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Syntax: AND destination, source.

This instruction performs logical AND on the operands and results are placed in destinat^n.

AND affects the following ~~registers~~: flags:
CF, OF, PF, ZF, SF

AND instruction is used to mask certain bits in the operand. It is also used to test for a zero operand.

1. AND DH, DH
   ZF=1
   (If answer is zero, then the operand in DH is zero).

2. MOV BL, 35H.
   AND BL, 0FH.
   This can be used to mask the higher nibble or MSB.

35 → 0011 0101
0F → 0000 1111
0000 0101

Division occurs on 8 or 16-bit numbers
In division we consider,
(a) Byte over byte
(b) Word over word.
(c) Word over byte
(d) Doubleword over word.

The could be cases where CPU cannot perform. The are refered to as an exception.
Exception can occur in following situations.
1. If the denominator is zero (divide by zero).
2. If quotient is too large for the assigned regist

byte / byte

DIV operand1.

The numerator must be in AL register and AH is zero.

DIV CL → AX is divided by CL

After division quotient is stored in AL and th remainder is stored in AH.

DIV NUM1 → Ax is divided by content of memory location NUM1

Word / word.

Numerator is in Ax.

DIV cx → Ax is divided by cx.

After division quotient is stored in Ax and the remainder is stored in Dx.

None of the flags are affected during division

OR

syntax: OR destination, source

The destination and source operands are ORed and result is placed in destination. Flags affected are same as AND instruction.
OR can be used for setting certain bits of the operand or to test for zero operand.

Eg: MOV AL, 35h
OR AL, 0Fh. (Set the lower nibble)

35 → 0011 0101
0F → 0000 1111
0011 1111 (All the bits in lower nibble are set).

XOR.

syntax: XOR dest, src.

The destination and src operands are exclusive-ORed.
XOR sets the bits if they are not equal.
Otherwise they are reset to zero.

Eg: XOR AL, 44.

AL → 0011 0101
44 → 0100 0100
0111 0001

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

XOR can be used to initialize a registers to zero as XOR BX, BX. This is faster than MOV BX, 00h.

There are two kinds of shift: Logical shift and arithmetic shift.
Logical shift is for unsigned operands
Arithmetic shift is for signed operands.
This section discusses logical shifts.

SHR: Logical shift right

Syntax: SHR dest, count.

The operand is shifted right for number of time specified in the count.

For every shift LSB will go to carry flag and MSB is filled with 0.

0 → MSB → LSB → CF.

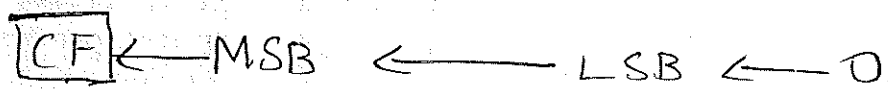
Eg. MOV CL, 03
SHR AL, CL AL = 9AH.

AL → 10011010 (Initial.
1st shift → 01001101 CF = 0.
2nd shift → 00100110 CF = 1
3rd shift → 00010011 CF = 0 AL = 13h.

This instruction produces the effect of dividing an unsigned number by power of 2
SHR is faster than DIV instruction.

SHL: Logical shift left.

Shift left is the reverse of the shift right SHR. After every shift, LSB is filled with 0 and MSB goes to CF



Eg: MOV dh, 4

SHL dh, 01h.

Madhuri. J
Dept of CSE
BIT

Initial	0000	0100
Ist shift	0000	1000

SHL instruction produces the effect of multi-plication by the powers of 2.

SHL is faster than MUL instruction.

Rotate Instructions

→ Rotate instructions perform the bit wise rotation of an operand. The rotation instructions are:

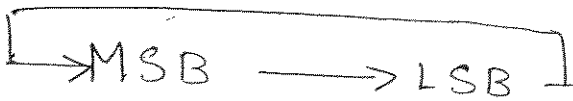
ROR, ROL, RCL, RCR.

→ The operand can be in register or memory.

ROR rotate right

→ In rotate right bits are shifted from left to right the exit from right end (LSB) and enter the left end (MSB)

→ The bit exiting the LSB, a copy of it is given to carry flag.



ROR dest, count

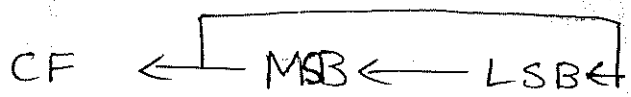
ROR AL, 1 (If operand has to be rotated

ROR AL, CL (If it has to be rotated more than one, register CL holds the count)

ROL (Rotate left)

→ In rotate left, bits are shifted from right to left end. and They exit the left end (MSB) and enter the right end (LSB).

→ Every bit that leaves MSB is copied to carry f



ROL op1, count

ROL AL, 1

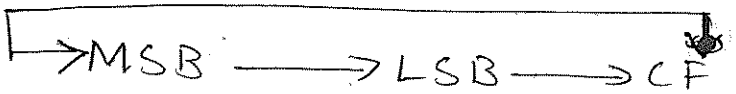
ROL AL, CL (If rotation is more than one, then register CL holds the count

RCR (rotate right through carry)

→ In RCR, bits are shifted from left to righ they exit LSB to the carry flag. The carry fl enters the MSB.

MOV al, 97h

RCR al, 1



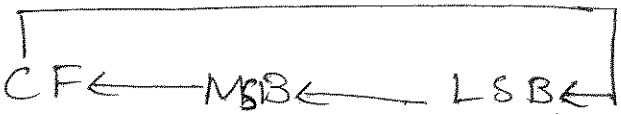
1001 0111 CF = 0

01001011 CF = 1

RCL (Rotate left through carry)

→ In RCL, bits are shifted from right to left. bit at the MSB enters the CF and carry fl enter the LSB.

RCL al, 1



Compare of unsigned numbers

Syntax: CMP dest, src.

→ The CMP instruction compares two operands and changes the flags according to the result of comparison. The operands themselves remain unchanged.

→ Although, all the flags are affected, only CF and ZF are used.

	CF	ZF
dest > src	0	0
dest = src	0	1
dest < src	1	0

Compare (CMP) instructions are followed by JMP instructions.

Eg: CMP CX, AX. [If contents of CX is above AX, then
JA Label. jump to Label]

→ CMP instruction is same as SUB instruction except that the values of the operands do not change.

Write an ALP to find the highest number in the array stored in memory.

BCD and ASCII conversion.

Two terms of BCD numbers:

(1) Unpacked BCD (2) Packed BCD.

Unpacked BCD

→ The low 4 bits of the number represents the BCD number and rest of the bits are 0.

Eg: 09 and 05 are unpacked BCD [illegible]

→ A single byte has two BCD numbers
Eg: 59 is a packed BCD number.

→ All the values entered from keyboard are in the form of ASCII Eg: When key '0' is pressed, 30h is provided to the computer

→ To process the data in BCD, first the ASCII data provided by the keyboard must be converted into BCD.

DAA (Decimal adjust after addition)

→ DAA instruction is used for the purpose of correcting the problem associated with BCD addit
→ DAA makes sure that the result of adding 2 packed BCD no's is adjusted to legal BCD no
DAA works only on AL and it can work only after ADD instruction.
Since (A - F) is not allowed in BCD, they have to be converted into BCD numbers.

Eg:
```
mov AL, 29h
mov BL, 18h
ADD AL, BL

DAA
```

```
  0010 1001
  0001 1000
-------------
  0100 0001 = 41h is incorrect

  0100 0001
     0110
-------------
  0100 0111 = 47h correct
```

DAA works as follows
1. After addition if Auxillary carry (AF) is set if lower nibble is greater than 9. Then add 6 to the lower nibble. In the above example, AF is set.

2. If higher nibble is greater than 9 or if the carry flag is set, Add 60 to the number or add 6 to higher nibble.

Eg. 53 + 75 = D8 + 6 = 128

0 1 0 1	0 0 1 1	
0 1 1 1	0 1 0 1	
1 1 0 1	1 0 0 0	As upper nibble is greater than 9, DAA adds 6 to upper nibble.
0 1 1 0	0 0 0 0	
CF = 1	0 0 1 0	1 0 0 0

1 28

DAS (Decimal adjust after subtraction)

↔ This instruction is used after subtraction of packed BCD operands.

→ DAS instruction can be used after SUB instruction and operand must be in AL register.

If AL = 86 BH = 57

SUB AL, BH.

1 0 0 0	0 1 1 0
0 1 0 1	0 1 1 1
0 0 1 0	1 1 1 0	CF = 0, AF = 1

2 F is incorrect.

Add 06 to LSB.

0 0 1 0	1 1 1 1
	0 1 1 0
0 0 1 0	1 0 0 1

2 9 is correct.

Summary

1. If after SUB or SBB if lower nibble is greater than 9, or if AF = 1 subtract 0110 from lower nibble.
2. If upper nibble is greater than 9 or CF = 1, subtract 0110 from upper nibble