DATE: 23-01-2024. DAY:

Computer Organization And Assembly Language (COAL)

Book:
Assembly Language Programing and organization of IBM-PC
(By Yath Yu)

Micro-Processor:

→ Processing Through system unit
SU has:
↳ CPU (Processing)
↳ RAM (Memory) (Temporary)
↳ Disk (storage) (Permanent)

3.0 GHz → Processing speed
3.0 G instructions in one second.

Low-Level Language:

→ Machine language (Using 0,1)
→ Assembly language (Using symbols)

(These languages are written appropriately if you have inside

DATE: DAY:

knowledge).

For Assembly ~~fito~~ language, you should have knowledge of hardware.

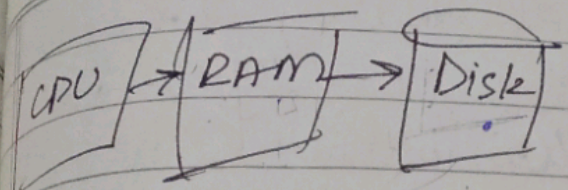
High level language is converted to low level language after compilation. The clock speed refers to the number of instructions in low language.

→ We will study intel 8086 ~~to~~ OR 8088 in this course.

→ Every CPU has its own assembly.

→ Only machine code is directly executable. Assembly code is also converted to machine code through assemblers. Like high level language is converted to machine code through compilers.

DATE: DAY:



RAM is faster bcz it has random/direct access and disk has sequential access.

Sequential Access: Access more than 1 locations in order to access a particular location.

Why RAM is called direct access?
Bcz we have used decoders, due to which only desired addressed is accessed.

RAM → 1 MB locations
→ 2^{20} number of locations
→ 1 Byte data at each locati
→ To assign unique address to each location, we will use 20 address bits bcz it will give us $(2)^{20}$ combinations.

$(2)^m$ → m → no. of address bits
→ no. of locations

DATE: DAY:

• 16 KB $= (2)^4 \cdot (2)^{10}$ B
$= 2^{14}$ Bytes

→ 14 addres bits.

• 2GB → 31 address bits

→ How many operations can be performed on RAM?

→ Always performed by CPU

→ 2 operations: Read/Write

Control Signals: (1 way CPU → RAM)
Sending R/W signal through control bus

Address bus: (1 way CPU → RAM)
Sending address of a particular location. (Address bus size is in accordance with RAM and CPU size).

Small address bus size and large RAM size will not work.

Data Bus: (2 way CPU ⟷ RAM)
Data to be read or written is sent through data bus.

DATE: 25-1-2024. DAY:

Components of CPU:

→ ALU (Arithmetic logic unit)

→ CU (Control Unit)

→ Registers (Small unit of memory)

→ Cache (Larger as compared to registers, stores data received from RAM)

→ FPU (floating point unit)

→ BIU (Bus interface unit)

Intel 8086 / 8088:
(16-bit) (8-bit)

Max. memory: 1 MB

No. of address bits = 20 bits

↳ Registers existing in these systems:

→ Data registers

→ Segment registers

→ Index and pointer registers

Register: Stores data in binary form

n-bit register:

signed unsigned → (Revise these concepts)

$(-2^{n-1} \leftrightarrow 2^{n-1}-1)$ $(0 \leftrightarrow 2^n-1)$

n=8

-128 / 127

-2^{n-1} ↔ $2^{n-1}-1$

n=4

0 - 15

0 - 2^n-1

0100
80
0180

DATE:

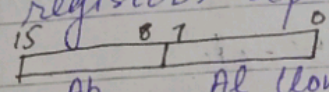
Any general data after calculations
↑
which data is stored here?

Data Register:

- i) Ax — Accumulator register
- ii) Bx — Base register
- iii) Cx — count register
- iv) Dx — Data register

↳ General purpose registers of size: 16-bit each

→ We can use half-half portions of these registers separately.



Ax:
↓
(complete portion)

Ah
↓
(upper half portion)

Al (lower half portion / least significant 8-bits)

Size of Ah/Al/Bh/Bl.... = 8-bits

→ They store data in binary form.

Segment Registers:

- i) DS — Data segment register
- ii) ES — Extra segment register
- iii) SS — Stack segment register
- iv) CS — Code segment register

↳ 16-bit each

↳ They stores memory address called segment address.

mov DS, AX

DATE: DAY:

Index & Pointer register:

SI — Source index register
DI — Destination index register
IP — instruction pointer register
BP — base pointer register
SP — stack pointer register.

→ 16-bit each
→ They store memory address called off-set address.

All I's and P's work with segment registers.

SI, DI ⟶ DS, ES
IP ⟶ CS
BP, SP ⟶ SS

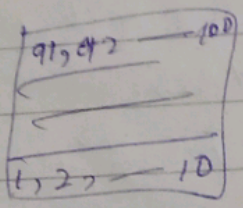
→ Registers don't have any address. They are accessed by their names and are a physical component.
→ Segment & index and pointer registers cannot be divided into two portions.

↳ Memory-address = 20-bits
The actual address of memory

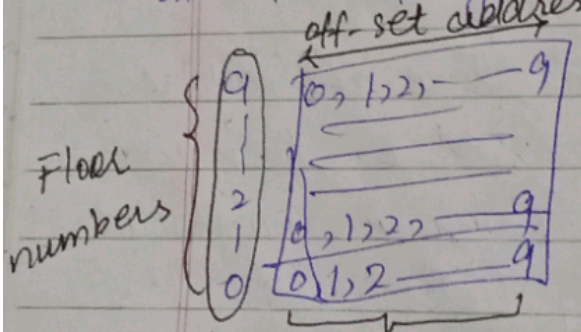
→ 1 byte is stored at each location in memory and we access complete location.

DATE: DAY:

is called physical address.
CPU can store memory addresses only in registers.
But each register is of 16-bits only. So, we use segmentation.
(Example of floor number, room number (& can't use 2-digits)
10-floors, 10-rooms, assign unique names.



Don't use 2-digits.



segment and off-set address works together to access memory's physical address.

e.g:
segment no=2 off-set no=7
Physical number = 27

DATE: DAY:

How to access physical address:

[PA = S * 10 + off]

This formula is only for above example.

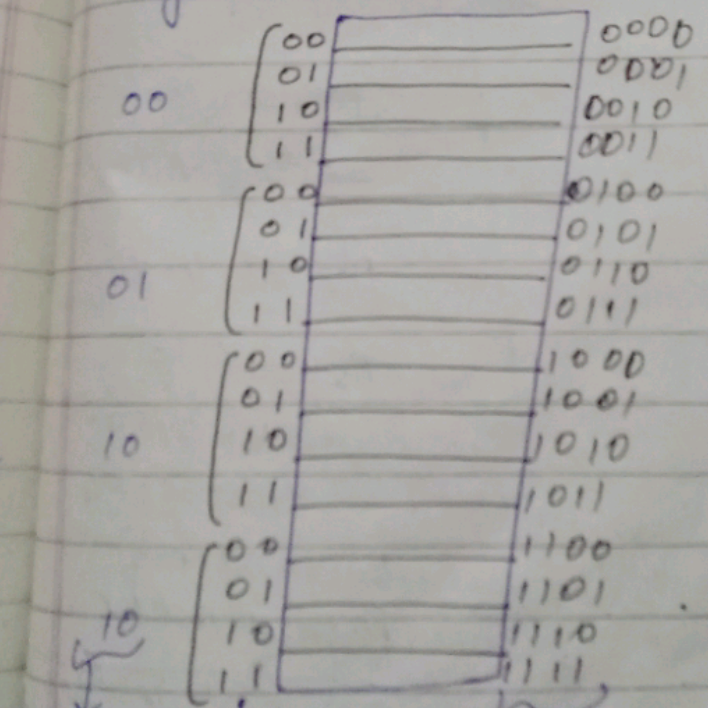
Memory Segmentation:

We have 16 B memory.

Registers → 2 bits

Physical address → 4-bits

so, we have to make segmentation of 2,2 bits.



How to access P.A? Just

A35
$16^2 \times 10^1 + 3 \times 16^1 + 5 \times 16^0$ 00000 ÷16 ÷16

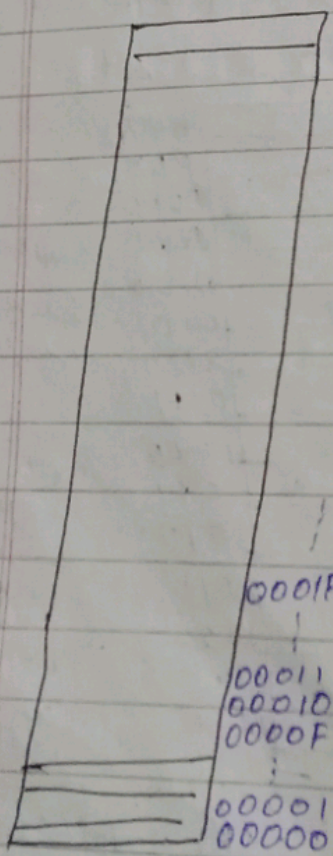
DATE: DAY:

concatenate both segment & off set address..

For 1-MB:

→ 1 hexa-digit has 4 binary digit.
→ So 5 hexa-digits has 20 binary digits.

So, we will write address in hexadecimal for 1MB memory to increase readability.



Registers: 16-bit

Segment = 4-digit hexa

Off-set = 4-digit hexa

~~How~~

S.A = 01FA

offset = 23FF

How to access Physical address:

P.A = S.A x 10 + offset

Result

DATE: DAY:

Result would be 5-digit hexa-decimal address. (But address bus will carry 20-digit binary number to access it).

~~01FA0~~

↳ S.A = 01FA

Offset = 23FF

S.A + 10 = 01FA0

```
  01FA0
+ 23FF
-------
  0439F
```

Physical address: 0439F

F = 15

1 → carry

16 | 25
16

9

↳ S.A = 31FE

Off-set = 2FDB

```
  31FE0
+ 2FD3
-------
  34FB3
```

Physical address: 34FB3

1
16 | 31
16

15

A 10
B 11
C 12
D 13
E 14
F 15

1
1 | 27
11

11

DATE: DAY:

For 1MB, each segment has maximum size 2^{16} bytes.

bcz:

Range of offset: 0000 ——— FFFF

$= 16$ bytes in terms

$= 2^{16}$ bytes in binary

i.e 64K

↳ If S.A = 01FA, find starting & ending ~~location~~ address of this segment.

For starting: S.A = 01FA

offset = 0000

P.A = 01FA0

For ending: S.A = 01FA

offset = FFFF

P.A = 11F9F

Now calculate P.A by formula.

Bcz each segment has:

starting off-set = 0000

ending off-set = FFFF

↳ If only P.A is given, we cannot calculate S.A and off-set.

Bcz formula has 3 quantities,

DAY:

DATE:

at least 2 should be given.

30-1-2024.

Overlapped Locations:

A location in which a single physical address has mapping against more than one logical addresses.

1234

i) S.A=1234 Off= 0000

PA= 12340
0000
12340

ii) S.A= 1234 Off= FFFF

PA= 12340
FFFF
2233F

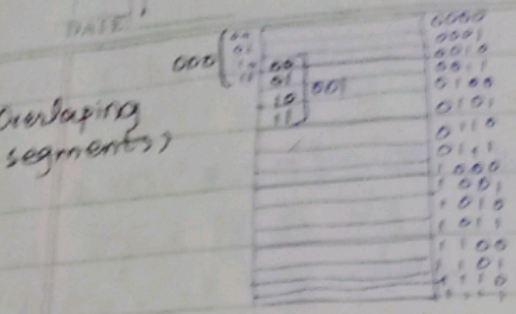
↳ 1235

i) S.A=1235 Off= FFFF

PA= 12350
FFFF
2234F

ii) S.A= 1235 Off= FFFF

PA= 12350



* How to find overlaping?
  - If ending address of first segment is greater than the starting address of 2nd segment, they are overlaping.

(To check no. of locations overlaped, find their difference and Add 1.)

Commands for debuging:

i) Registers:

command: r

↳ Debuger tool always work in hexadecimal.

→ r command shows information about register.

Assembler for 8086 → Masam MASM

DATE: DAY:

→ r ax command shows what is stored in AX and we can also store desired value.

→ r ah OR r al is not allowed.

→ q command quits debugger mode.

Dump command:

→ d

Shows data (i.e content of memory)

This shows data of 16 Bytes in one row.

i.e total 128 bytes.

The last portion shows corresponding ASCII character to that hexadecimal number.

Modes of d:

→ we can write off-set address with d. It will show data of next 128 bytes starting from that off-set.

41 in hexa = 65 in binary
61 in hexa = 97 in binary

DAY:

DATE:

→ we can also write specific interval of off-set addresses.

e.g d 0270 0275

→ we can also write specific segment address

e.g d 3000:0100

OR d 3000:200 20a

only this inverted will be shown

offset address should always be given with segment address.

Enter Command: (e)

To write data at a particular place.

→ Write off-set address with data.

→ Write data in spaces.

→ You can also write segment address.

→ Data can be written in form of a string in single quotes.

DATE: DAY:

Fill Command:

Also to enter data.

→ Needs starting + ending or off-set where you want to store a particular data.

e.g:

f 4000:0100 0120 44

44 will be stored in all these ~~segme~~ locations.

Data can also be stored like:

f 4000:100 130 41 42 43

ASSEMBLY INSTRUCTIONS:

→ Assembly language is not case sensitive.

(1) MOV

Transfer data from register to register, memory to register, register to memory → or a constant value to register.

AX=CX
CX=BX
BX=AX

DATE: (2 operands) DAY:

Syntax: MOV destination, source

AX=BX
CX=AX

MOV AX, BX

MOV AX, 123

CX=AX
AX=BX
BX=CX

MOV AL, 20

MOV AL, BX (Wrong bcz 8 bit, 16 bit)

MOV AL, FFFF (Wrong)

MOV AX, BL (X 16, 8 bit)

Both Operands should be of same size.

from memory to register.

MOV DX, [0100]

0100 → 4 bytes

4 bytes

I → offset address, segment address: By default of data segment.

[0100] has 1 byte data whereas DX has 16 bits.

0101 So the data of next location will also be copied.

byte = 8-bit

1 word = 2 byte

1 double word = 4 bytes

1 quad word = 8 bytes

DATE: DAY:

from register to memory:

MOV [0100], AX

If AX → 16 bits then data will be stored at next off-set address also i.e.

AX = 4A32

MOV [0100], AX

0100 → 32 0110 → 4A

⇒ MOV 32, AX (Wrong)

② **INC:** (increment)

1 operand instruction
↳ only a register

INC CX → Increment of 1 in that register

③ **DEC:**

Same as DEC

→ Decrement of 1 in that register

④ **ADD**

2 operands

i) Both registers
ii) 1 register, constant
iii) 1 register & 1 memory

DATE: DAY:

ADD AX, BX
(AX ← Destination, BX ← sources)

Add content of AX and BX and store in AX - BX is unchanged.

⑤ SUB

SUB AX, BX

AX - BX and store in AX.

Following operands are not allowed:

i) memory, constant

1-2-2024.

→ Assemble Command:

→ By default, there appears 1 segment address (of Code Segment) and off set address (of IP).

→ Assembly instructions can be written under this command.

DATE: DAY:

e.g:

a
CS : IP
073F:0100 mov ax,1234
073F:0103 mov bx,1222
073F:0106 add ax,bx
073F:0108 inc ax
073F:0109 (Just press enter to come out of this mode)

→ To view code:

Unassemble command:

-u

Format of its output:

Seg. Add: Off-set | Machine code of every assembly instruction | Assembly instruction
(CS) (IP)

Diff of off-set addresses is in accordance with size of instruction.

Size: MOV → 3 bytes
SUB/ ADD → 2 bytes
INC/DEC → 1 byte

0102 0103 0104 0105

↳ Not specific everytime. So, we will calculate it by machine code.

RISC Architecture: Less registers, same size of each instruction.
CISC Architecture: More registers, different size of each instruction.
↳ Our system has CISC architecture.

Unassemble command can be use only starting offset or an interval of offset.

e.g -U0100
OR -U 0100 0108

After r command:
Third row shows instruction present at current IP address.
This instruction is first executable.

Trace command: line by line
Executes assembly instructions.
It by default shows current status of register.
We don't need to enter register command.

DATE: DAY:

After execution of 1st instruction, the IP address + 3rd row is updated. Now ~~2nd~~ next executable instruction is present in 3rd row and next IP is off set (by adding bytes of instruction i.e for MOV → 3, ADD → 2 etc.

This can be told by number of digits of machine code.

e.g 6 hexa = 24 binary = 3 bytes

Enter command:

-e 4000:0100 'Hello'

Output:

4000:0100 48 65 6C 6C 6F 00

⇒ We cannot store constant value directly in segment registers. First store it in some general purpose register and then move to segment register.

DATE: DAY:

However, constant value can be stored in data & IP index/pointer registers.

2-2-2024

Flags: (each flag is of 1 bit)

Flags change only on arithmetic operations (not on MOV)

i) Zero Flag: (not on MOV)

Possible states: 0 (Reset)
1 (set)

If the answer of an arithmetic operation (Add/Sub) is zero, zero flag will set itself (ZR) and if result is 1, it will reset itself (NZ)

0 → NZ (Not zero)

1 → ZR

ii) Sign Flag:

It tells whether result is +ve or -ve.

If most significant bit (and carry) is zero, it will reset, otherwise it set.

Possible states:

+ve → reset(0) → PL

-ve → set(1) → NG

(zero is considered positive)

iii) Carry Flag:

0 → Reset → NC

1 → set → CY

DATE: DAY:

If the computation is performed considering unsigned numbers, it will set on overflow and reset when result is within range.

```
  56          FF
+ 23          01
  79          00
```

→ There would be a carry.

No carry
so → (Reset)

so → (Set)

For subtraction, if you need Carry in, then set otherwise reset.

iv) Overflow Flag: 0 → Reset → NV
1 → Set → OV

Checks range like carry flag. Diff is → numbers are considered signed numbers.

Example:

```
MOV AL, FF
MOV BL, 01
Add AL, BL
```

```
 11111111
 00000001
(1)00000000
```

Zero flag → set sign flag → reset
carry flag → set Overflow flag → reset

If you are performing computations on AL, BL etc, then check status of flags within range of 8-bits.
But if you are performing on AX, BX, check status within range of 16-bits.

Examples:

1) MOV AL, 80
MOV BL, 80
ADD AL, BL

80 zero-flag: set sign-flag: set
80 carry-flag: set overflow flag: set
1000 0000
1000 0000
(1) 0000 0000

80 → -128
80 → -128
-256

Range of 8-bit = -128 ↔ +127
so not in range.

80 = 1000 0000
++
0001

0111 1111
+1
1000 0000
= -128

By inspection:
Sum of 2 -ve numbers should be -ve. But our answer is +ve. It means there is overflow.

-128 → 127

-128 - (-128)
= -128 + 128
= 0

DATE: DAY:

ii) MOV Al, 80
MOV Bl, 80
SUB Al, Bl

1000 0000
1000 0000
→ 0000 0000

-128
+128
0 -256 0

Z.F: set SiF: reset
C.F: reset O.F: reset

→ MOV ki instruction does not change status of flags.
→ INC ki instruction does not affect carry flag.

13-2-2024.

Jump instructions:

Jumps from one instruction to another.

→ i) conditional
→ ii) unconditional jump (JMP)

JMP ± offset address
space

It functions as a loop.

DAY:

DATE:

unconditional → No testing here

```
_ :0100   ADD  AX,BX
_ :0102   SUB  AX,BX
_ :0104   INC  Al
_ :0105   JMP  0100
```

↓ This moves to 1st instruction (0100)

JMP → IP address updated to address written after JMP. In this way, that instruction executes.
→ Infinite loop

Conditional Jump:

i) **JZ:** (or JE)
Jump if zero flag is set.

ii) **JNZ:** (or JNE)
Jump if zero flag is not set.

iii) **JC:** Jump if carry flag is set.

iv) **JNC:** Jump if carry flag is not set.

DATE: DAY:

Syntax:

Any general purpose register not used in loop body

MOV CX, 5

:XXXX → Loop Body

DEC CX

JNZ XXXX

No. of times you want to run the loop.

Working:

CX = 5

→ loop body

DEC → CX = 4

JNZ: (true) Bcz 4 in CX

So jumps again to first line of loop body whose ~~IP add~~ offset address we have written after JNZ.

If JNZ → false,

loop terminates.

Example:

e 4000:0100 40 41 42 43 44

Write a program to copy 5 bytes data from 0100 to 0200.

MOV [0100], [0200] ✗
Bcz do not clear how much data should be copied.

Hard code:

```
MOV SI, 0100
MOV DI, 0200
MOV AL, [0100] / [SI]      (Not efficient)
MOV [DI], AL
```

[And do this 5 times. by changing SI's, DI's i.e 0102, 0202 - -

Loop:

```
1   MOV   AX, 4000
2   MOV   DS, AX
3   MOV   SI, 0100
4   MOV   DI, 0200
5   MOV   CX, 5
6   MOV   AL, [SI]
7   MOV   [DI], AL
8   INC   SI
9   INC   DI
10  DEC   CX
11  JNZ   6
```

Let offsets be:

Why we used SI & DI?
Bcz for DS, these two are used.

DATE: DAY:

IP value:

If JNZ → true (that one written after JNZ)

If JNZ → false (2 bytes + current IP) → depends on machine code

Write above program but store data in reverse orders:

Line# 4 → MOV DI, 0204

Line# 9 → DEC DI

Write above program to copy data of 5 words (10 bytes)

Line# 6 → MOV AX, [SI]

Line# 7 MOV [DI], AX

Line# 8 : Add SI, 2

Lin# 9 Add DI, 2

→ OR INC 2 times

→ 2 bytes data copying (Do carefully)

DATE:

20-2-2024

File name for assembly: .asm

Assembler → MASM

Code format:

- .Model Small
- .stack 100h
- .data

≡

- .Code

Assembly code

Main Proc

≡

Main endp

last line of each code ← end main

In assembly, numbers are by default considered decimal as in debugger they are considered hexa. So we have to write "h" for hexa.

NOTE

We can also write .code first and .data afterwards.

Everything written in .data moves to DS and .code moves to CS.

Example: (for copying 5 byte data from 1 array to other)

- .Model Small
- .stack 100h
- .data

array1 → same name conventions as in high level language

array1 db 41,42,43,44,45

array2 db 5 dup(?)

db: defined byte → duplicate

db	dw	dd	dq
defined byte	defind word	defined double	defined quard

↳ due to ? → array will be empty, otherwise write whatever you want to store.

⇒ Data is in sequence.

i.e if 41 → 0100, 42 → 0200 ...

Segment address of DS changes ~~for~~ every time used.

```
.data
array1 db 41h,42h,43h,44h,45h
array2 db 5 dup(?)

.code
main proc
mov AX, @data
mov DS, AX
mov SI, offset array1
mov DI, offset array2
mov CX, 5
aa: mov al, [si]
mov [Di], al
INC SI
DEC DI
DEC CX
JNZ aa

mov Ah, 4ch
INT 21h
main endp
end main
```

DATE:

In every code (changing) ↑

mov AX, @data } → initializing data segment register

keyword

Tag ← aa:

↓

Naming conventions same as variable name

These 2 ending instructions ← are must for every code. as (works same returns)

file extension

DATE: DAY:

⇒ Use Notepad → Create .asm file in 8086 directory.

Assembler

Execution: (in DOSBOX)

masm prog.asm → (After this instruction, code is assembled)

→ object file formed here

for execution

link prog.obj; and enter

OR

link prog.obj and Enter 3 times

→ executable file formed here

for execution:

prog.exe

↳ Done now

→ execution done here

To import file in debugger mode,

debug prog.exe

Interrupt Instruction,

for its execution, don't use trace, use proceed (p).

It will show: Program terminated normally.

Service numbers of INT:
- 4ch → Terminate
- 09h → Print string
- 01h → Console input (only one character)

To use masm & link simultaneously:
ml prog.asm

22-2-24.

Console Input/Output:

→ Assembly language only deals with ASCII codes.

'A'	65	41h
'a'	97	61h
'0'	48	30h
space	32	20
enter	13	0Dh
return	10	0Ah

(enter + return = next line) → used simultaneously for next line

Defining String:
- data → we can use single/double quotes

```
str1 db "Hello World $"
```

String is not stored in memory, rather their ASCII code is saved.

→ Add $ sign at the end of string to avoid any garbage value. Dollar sign shows that string ends here.

DATE: DAY:

```
.code
main proc
MOV AX, @Data
MOV DS, AX
MOV DX, offset str1
MOV Ah, 09h
INT 21h
```

To display string

String will be displayed upto 1st dollar sign and if no dollar sign is there, some garbage data will also be printed.

INT: (depends on service number of Ah)
It checks what is stored in Ah. If it is 4Ch, it terminates program normally. But if it is 09h, it will print the string whose offset address is present in DX.

Printing a single character

Service number
↳ 02h

Which character is printed? whose ASCII code is present in DL.

```
.code
// 1st 2 lines
MOV DL, 'A'
MOV Ah, 02h
INT 21h
```

Or
~~MOV DL, 65~~

DATE: DAY:

To print 'A'.

→ MOV DL, 'A'
→ mov DL, 41h
→ MOV DL, 0100 0001b
→ mov DL, 65
→ mov DL, 65d

Assembly does not have any data type. Everything is just a character.

If you want to read character from memory; use its offset address.

mov DL, [-]

DL → 8-bits → Range → 0-255
so characters corresponding to these ASCII codes will be displayed.

By default, Printing is on same line. for next line:

Sequence:
1st 10
Then 30

```
mov dl,10
mov ah,2   → because we are dealing with enter as a "character"
int 21h

mov dl,13
mov ah,2
int 21h
```

DAY:

DATE:

If you don't want to write these instructions for next line, then do this:

```
str1 db "Hello World.$"
str2 db 10,13,"Bye World$"
```

This will store these two characters in string and memory. On giving its offset address, first these two are executed.

10 → stores binary of 10

10 → stores binary of 1 and 0

↳ If .model small

1 segment → data 2 segment → code

→ If we write data that exceeds our segment, use medium/large. If still exceeds, it will show error.

Print →

```
.data
str1 db "*$"
str2 db 10,13,"**$"
str3 db 10,13,"***$"
str4 db 10,13,"****$"
.code
main proc
    MOV AX, @Data
    MOV DX, AX
    MOV DX, offset str1
    MOV Ah, 02
    INT 21h.
    MOV DX, offset str2
    MOV Ah, 02
    INT 21h
    MOV DX, offset str3
    MOV Ah, 02
    INT 21h
    MOV DX, offset str4
    MOV Ah, 02
    INT 21h
```

for (i=0; i<4; i++) {
for (j=1; j<=i; j++)
printf (*)
printf ("\n")
}

DATE: DAY:

Through Loop:

cl → behaves as i
ch → behaves as j

~~MOV cl,4~~
~~MOV ch,1~~

```
      MOV  cl,1
bb:   MOV  ch,cl
aa:   MOV  DL,'*'
      MOV  AH,2
      INT  21h
      DEC  ch
      JNZ  aa
      MOV DL,10
      INT 21h

      MOV DL,13
      MOV AH,02
      INT 21h

      INC  cl
      CMP  cl,4
      JBE  bb
      MOV  AH,4Ch
      INT  21H
```

DATE: DAY:

Input:

```
mov Ah,01
INT 21h
```

→ Just a single character from keyboard
→ We dont need to press enter
→ The ASCII code of entered character is stored in Al.
→ The cursor will blink showing that assembler is asking for an input (just a single number).

for more characters:
use loop, but store each character in an empty string.

DATE: DAY:

Convert above program by taking no. of times by user:

```
INC Cl
MOV Ah,01
INT 21h

SUB
MOV Bl,Al
SUB Bl,30h
CMP Cl,Bl
```

→ why we used Bl? Bcz INT can change contents of Al, Ah.

DATE: 26-2-2024. DAY:

Toggle uppercase & lowercase:
Upper case has bit 5 '0' while lower case has bit '1'. Just toggle this bit by using ⊕ XOR.

(i) XOR Al, 00100000b
OR XOR Al, 20h
OR XOR Al, 32d

0→0
1→0
→ Just toggle lowercase:
AND Al, 11011111b
OR AND Al, DFh

→ Just toggle uppercase:
OR Al, ~~01111111b~~ 00100000
OR ~~XOR Al~~

0→1

next → Data section always give consecutive addresses to variables / arrays.
(1) e.g in above example, if V1 ends at 0209, V2 will start at 0203.

DATE: DAY:

.data

V1	byte (or db)	12h, 65, 0101b	
V2	Word (or dw)	1234h	(lowest significant byte) ← lower byte, upper byte
V3	Sbyte (signed byte)	-5	
V4	Dword (double word)	5 dup(0)	
V5	WORD	V2 (its offset address will be saved)	starting

/* V db 41h, 42h, 43h, $ (0100 0101 0102 - ( ) → will also be given address
db 45h, 46h, 47h (0103 0104 0105)
db 5 dup(0) (0106 — 010A)

→ No variable declared, but still it will be given memory address

.code:
Now DX, offset V
mov ah, 9
INT, 21h
Output: ABC */

→ MOV SI, offset V1
~~MOV~~ MOV CL, [SI]
OR MOV CL, V1
→ 1st value of V1

OR MOV CL, V1+2 → 3rd value of V1

To store 1234 at 0101, 0102

1234 → 0101 (least significant byte stored first)
→ 0102

DAY:

To copy into AX:

AL → 34
AH → 12

→ If you are storing an offset in a variable, its data type should always be "word", i.e 2 bytes.

```
.data                          0000 0001 0002
    v1 byte 12h, 65, 0101b
    v2 sbyte -5                0003
    v3 dword 5 dup(0)          0004 — 0017
    v4 word v2                 0018, 0019
    v5 word 1234h              001A - 001B
    v6 word 1235h              001C - 001D
.code
main proc
    mov ax, @data
    mov ds, ax
    mov si, offset v1          si, 0000
    mov cl, v1                 cl, [0000]
    mov dl, v2                 dl, [0003]
    mov bx, offset v3          bx, 0004
    mov cx, v5+2               cx, [001C]
    sub v5, 2                  [001A], 2
mov ah, 4ch
```

1234 − 2 = 1232

1232

DATE: DAY:

→ If we move one variable into another, its offset address will be moved.

→ If we move one variable in a register, the value at its starting offset will be moved to register. If you want to move that offset only, then write:

mov ax, offset v1

→ To move, consider data type, i.e for byte use AL/AH and for word use AX/BX —

→ If you are moving a hexa number starting with an alphabet, write zero at its start.

e.g mov AX, FFASh → error
MOV AX, 0FFASh → true

DATE: 27-2-2024 DAY:

Shift & Rotate Instructions:

Shift
- Logical shift
  - i) SHL (left)
  - ii) SHR (right)
- Arithmetic shift
  - i) SAL
  - ii) SAR

=> All have 2 operands.

SHL: SHL AL, 2 → [kitna shift?]

If AL = A2h

A2h = 1010 0010

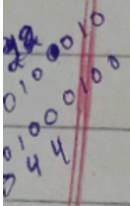
SHL AL, 1 → [C.F]

01000100 → (0 at empty bit in case of SHL)

Where msb moved?
↳ To carry flag
if it zero C.F = rest, if 1, C.F = set.

For SHL AL, 2
- i) 01000100
- ii) 10001000

Now C.F will have most latest shifted bit.



DATE: DAY:

AL, 20h

SHL AL, 2

i) 0010 0000

ii) 1000 0000

C·F = reset

Single shift left, magnitude × 2

2 times shift: magnitude × 4

3 times shift = magnitude × 8

SAL:

Same behaviour as SHL.

SHR:

AL = 1010 0001

SHR AL, 1

C·F

(zero at empty bit in SHR) 0101 0000

→ Single shift right = Number/2

Two times = Number/4

SAR: AL = 10100001

SAR AL, 1

1101 0000

(Same value which was already in msb)

DATE: DAY:

If,
AL, 1000 0001
SAR, 3
1 1 1 1 0000
[msb (1) at each place]

(That's why zero at empty bits) Logical Shift: Unsigned number
(That's why msb at empty bits) Arithmetic Shift: Signed number

= 1000 0000 (80h) (128) Sgn (-128)
SHR: 0100 0000 (40h) (64)
SAR: 1100 0000 (C0h) (-64)

So, Right shift divides by 2 no matter signed or unsigned.
Overflow → Above rule does not apply

In both left & Right Shifts, the bit which is ~~shifted~~ out is moved to carry flag.

DAY:

DATE:

Read a string from user (having small letters). Find count of a's (assuming there are not more than 9 a's entered by user)

for "enter" comparison:
(i.e end string when user press enter)

```
str1 db 50 dup('$')
.code:
aa   MOV, AH, 1      → Now add 30h at the end, otherwise MOV BL, '0'
     INT 21H
     CMP AL, 13
     JE exist
     MOV [SI], AL     CMP AL, 'a'
     INC SI           JNE aa
     JMP aa1          INC bl
                      Add BL, 30h
                      MOV dl, bl
                      MOV AH, 2
                      INT 21h
exit: MOV Dx, offset str1   → only for displaying string
      AH, 9
      INT 21H
```

DATE: DAY:

In case of termination on taking 50 characters

```
MOV CX,0049
aa:
   |
   |
   |
INC SI
DEC CX
JNZ aa
```

if a etc register has a digit, to print it, you have to add 30h or 48h

5-3-2024.

SHL AL,2 → Error

Solution?

```
MOV BL,2
SHL AL,BL
```

Another solution:

- model small
- 386
- stack 100h

SHL AL,2 ✓

DATE: DAY:

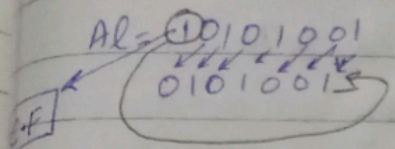
Rotate Instructions:

Without Carry — With Carry

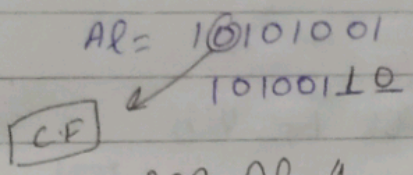
ROL — ROR — RCL — RCR

Al = 1010 1001

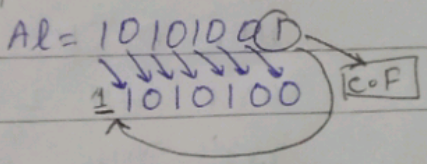
ROL Al, 1



ROL Al, 2

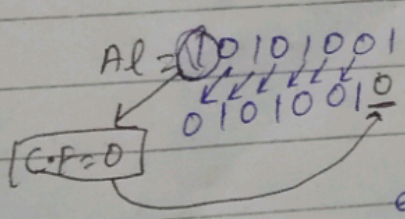


ROR Al, 1



With Carry:

RCL Al, 1 (Assuming C.F = 0)



Jo bit out hogi, wo C.F mn jaye gi, aur jo kuch C.F mn already hoga, wo empty bit pe jaye ga.

DATE: DAY:

RCL AL, 2

AL = 0100 0101

1000 1010

0001 0100

C.F=0

↳ C.F=1 after this example.

Example:

MOV BL, 5Ch

Display value of BL on console in binary:

5c

0101 1100

Let,

MOV BL, 41h

Now 41h

MOV DL, BL

MOV AH, 2

INT 21h

→ It will only display 'A' not binary of 41h.

code:

```
MOV BL, 41h  → 0100 0001
MOV CX, 8

again:  SHL BL,1
        JC aa
        MOV DL, '0'
        MOV AH, 2
        INT 21H
        JMP bb
aa:     MOV DL,'1'
        MOV AH,2
        INT 21H
bb:     DEC CX
        JNZ again
```

For reverse binary: SHR BL,1

For BX:
```
MOV CX,16
SHL BX,1
```

After execution:
BL will be 0000 0000

what if you don't want to change contents of BL? ROL BL,1

C→99 4→ 'c' → 20h loop loop loop

DATE: loop DAY:

An array has 5 elements. Display binary of each at separate line.

```
arr1 db 40h, 41h, 42h, 43h, 44h

.code:
        mov si, offset arr1
        mov cl, 5
cc:     mov bl, [si]
        mov ch, 8
again:  shl bl, 1
        jc aa
        mov dl, '0'
        mov ah, 2h
        int 21h
        jmp bb
aa:     mov dl, '1'
        -
        -
bb:     dec ch
        jnz again
        dec si
        inc si
        dec cl
        jnz cc
```

for dw:

```
mov bx, [si]
mov ch, 16
shl bx, 1
;
inc si
inc si
.
```

jnz again → dl, '10' → Next line

dec si → dl, '43' → Next line

cq24

BL (24)

DAY:

⇒ Display BX in hexa-form.

```
      MOV BX, 4C92h
      MOV CX, 4
again: ROL  BX, 4      BX = C92 4 924C
      MOV  Dl, Bl      Dl = B4C
      AND  Dl, 0Fh     Dl = 0C (12)
      CMP  DL, 9
      JBE  aa
      Add  Dl, 7       12+7 = 19
aa:   Add  dl, 48/30h  19+48 = 67
      MOV  AH, 2
      INT  21H
      DEC  CX                C
      JNZ  again
```

⇒ An array has 5 elements. Display hexa-form of each element.

DATE: DAY:

- data

  arr1 db 41h, 42h, 43h, 44h, 45h

- code

```
        mov si, offset arr(1)
        mov cl, 5
again2: mov bx, [si]
        mov ch, 4
        ROL bx, 4
        |
        |
        |
        |
        |
        DEC cx
        JNZ again
        Next line
        inc si
        dec cl
        JNZ again2
```

for dw:

```
MOV BX, [SI]
MOV CH, 4
ROL BX, 4
|
|
|
INC SI
INC SI
```