

## COAL LECTURE 3

Virtual machine → for 16 bit environment

*The command MOUNT C /home/name/ee213 in DOSBox would attempt to mount the directory /home/name/ee3 on your local system as the C drive within the DOS environment of DOSBox. This means that the files and folders within /home/name/ee3 would be accessible as if they were located on the C drive within DOSBox.*

```
setup.sh
1  #!/bin/bash
2  # setting up a 16-bit environment for learning
3
4  sudo apt install dosbox
5  dosbox
6  mount c /home/nam/ee213
7
8  nasm c.asm -o c.com
9  afd c.com
```

```
1  [org 0x0100]
2
3  ; start of code
4
5  mov ax, 5          ; move the constant 5 into register ax
6  mov bx, 10
7
8  add ax, bx         ; add value of bx into the value of ax
9
10 mov bx, 15         ; add constant 15 into the value of bx
11 add ax, bx
12
13 mov ax, 0x4c00     ; exit ..
14 int 0x21           ; .. is what the OS should do for me
15
16
17
```

. ax → register

. A bit is a single binary digit, a 0 or 1. A nibble is defined as half of a byte. A byte is two nibbles. A word is two bytes. We also have the double-word/DWORD, which is two words, and quad-word/QWORD. In Intel/AMD architecture, that works out to the following values:

**Bit: 1 bit**

**Nibble: 4 bits**

**Byte: 8 bits**

**Word: 16 bits**

**DWORD: 32 bits**

**QWORD: 64 bits**

1. **mov ax, 5:** This line moves the constant value 5 into the 16-bit register ax. Here, ax is being used as a general-purpose register to hold a value.
2. **mov bx, 10:** Similarly, this line moves the constant value 10 into the 16-bit register bx.
3. **add ax, bx:** This line adds the value stored in register bx to the value stored in register ax and stores the result back in register ax. This instruction effectively performs  $ax = ax + bx$ .
4. **mov bx, 15:** This line moves the constant value 15 into register bx, overwriting the previous value.
5. **add ax, bx:** Again, this line adds the value stored in register bx (which is now 15) to the value stored in register ax and stores the result back in register ax. (30)

MOV AX → OP CODE → B8 , 0500 → 5 (0500 OPERANDS B8 OPERATION CODE OF AX)

MOV BX → BB 0A00 → 10

WHERE MY PROGRAM LOAD IN RAM → BASE ADDRESS

SO 000000000 IS NOT RAM 0 ADDRESS BUT MY PROGRAM

In this case, MOV instructions typically take up 3 bytes - 1 for the opcode and 2 for the operand), the next instruction would be stored at the next memory address after the current instruction.

if MOV AX, 5 is stored at memory address 0x00, then MOV BX, 10 would be stored at memory address 0x03 (assuming MOV AX, 5 takes up bytes 0x00, 0x01, and 0x02).

1	1	[org 0x0100]	
2	2		
3	3		; start of code
4	4		
5	5	00000000 B80500	mov ax, 5 ; move
6	6	00000003 BB0A00	mov bx, 10
7	7		
8	8	00000006 01D8	add ax, bx ; add v
9	9		
10	10	00000008 BB0F00	mov bx, 15 ; add c
11	11	0000000B 01D8	add ax, bx
12	12		
13	13	0000000D B8004C	mov ax, 0x4c00 ; exit
14	14	00000010 CD21	int 0x21 ; .. is
15	15		
16	16		
17	17		

Dosbox start addresses from 100 thats why we write it org 0x0100 my program if load at 1000 then it starts from 1100.

When using DOSBox, memory addresses start from 100h. Therefore, if a program is loaded at address 1000h, it starts executing from 1100h, as indicated by the `ORG` directive in assembly language.

8086/8088, CPU speed: 5000 cycles, Frameskip: 0, Program: AX

AX 0000	SI 0000	CS 19F5	IP 0100	Stack +0 0000	Flags 7202
BX 0000	DI 0000	DS 19F5		+2 20CD	
CX 0012	BP 0000	ES 19F5	HS 19F5	+4 9FFF	OF DF IF SF ZF
DX 0000	SP FFFE	SS 19F5	FS 19F5	+6 EA00	0 0 1 0 0

M1 addr: seg\_reg:

CMD >m1 0100

0100 B80500	MOV	AX,0005
0103 BB0A00	MOV	BX,000A
0106 01D8	ADD	AX,BX
0108 BB0F00	MOV	BX,000F
010B 01D8	ADD	AX,BX
010D B8004C	MOV	AX,4C00
0110 CD21	INT	21
0112 0000	ADD	[BX+SI],AL

	0	1	2	3	4
DS:0100	B8	05	00	B8	0A
DS:0108	BB	0F	00	01	D8
DS:0110	CD	21	00	00	00
DS:0118	00	00	00	00	00
DS:0120	00	00	00	00	00
DS:0128	00	00	00	00	00
DS:0130	00	00	00	00	00
DS:0138	00	00	00	00	00
DS:0140	00	00	00	00	00
DS:0148	00	00	00	00	00

.m1 0100 showing the same address here B80500

```

12
13  mov ax, 0x4c00    ; exit ..
14  int 0x21          ; .. is what the OS should do for me

```

-Application execution via OS:

- Our application relies on the operating system for all tasks.

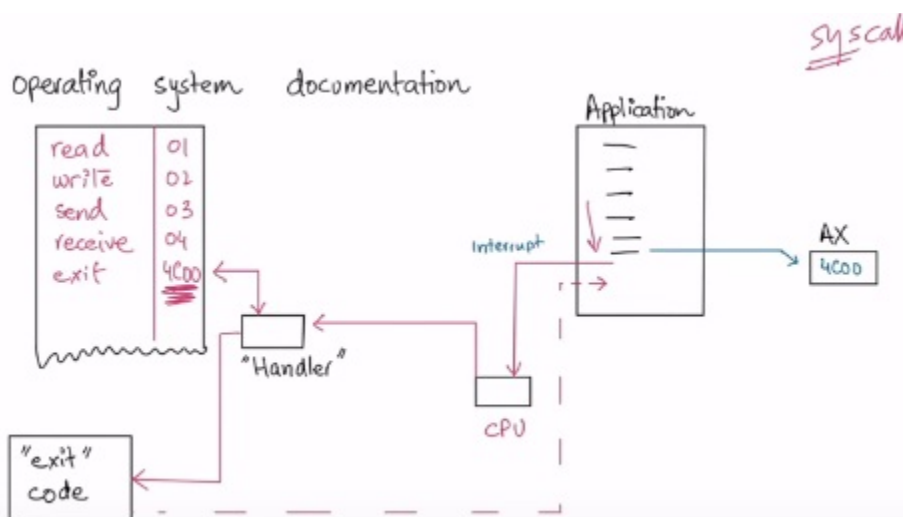
- System Call Handling:

- Occasionally, the OS interrupts our execution and assigns CPU tasks.

- Interrupt 0x21:

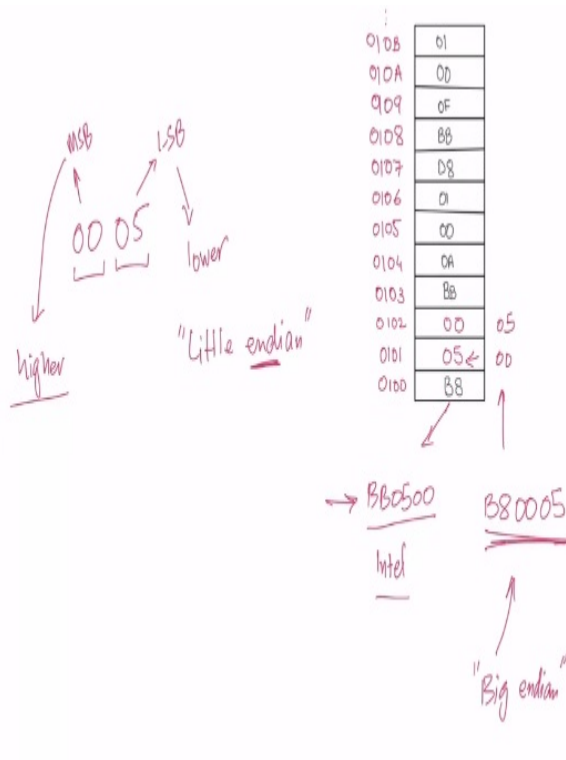
- This interruption prompts the CPU to perform specific work.

- Exit Operation:- Tasked with the work code 4C00, which signifies program termination.



### - Data Storage in Memory:

- According to Intel's convention, when storing data like `AX` with a value of `0005h`:
- The LSB (Least Significant Byte) `05` occupies the lower memory address.
- The MSB (Most Significant Byte) `00` occupies the higher memory address.
- Hence, in memory, `AX` with the value `0005h` would be stored as `0500h`.



### Little Endian and Big Endian:

- **Little Endian:**
  - In little-endian systems, the least significant byte (LSB) is stored at the lowest memory address.
  - The most significant byte (MSB) is stored at the highest memory address.
- **Big Endian:**
  - In big-endian systems, the most significant byte (MSB) is stored at the lowest memory address.
  - The least significant byte (LSB) is stored at the highest memory address.