* Segmentation of Memory →

Direction

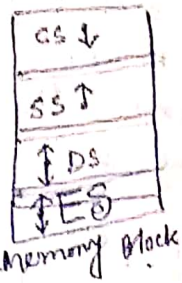


fig 1

Code goes on to new increasing addresses. →↓
Stack goes on to new decreasing addresses. →↑
Data can go up/down randomly.

Now if we consider a memory block without any segmentation, then there is possibility of overwriting b/n code, data, stack. Hence segmentation is essential so that we can do avoid this problem & the searching space of code, data, stack become more easy to access.

- In 8086 µp we have increased address bus by 4 bit. So we get 20 bit address bus. This is done so that we can interface 1 MB memory location with 8086. This is good advantage. But problem is this requires $2\frac{1}{2}$ byte size bus. That means $\frac{1}{2}$ byte is waste here. Bcoz we know computer works w.r. to byte (8bit)
- Hence a method is introduced by which 16 bit addresses are used to calculate Physical address (20bit).

$$\text{Physical Add. (20bit)}_H = \left[(\text{Segment Add})_H \times 10_H\right] + \underset{(16\text{ bit})}{\text{Offset Add}}$$

** Here the segment addresses are not fixed. Programmer can change this address while writing program. It must be included at begning of program. ** Then, just using 16 bit offset address programmer can write program like 8085.

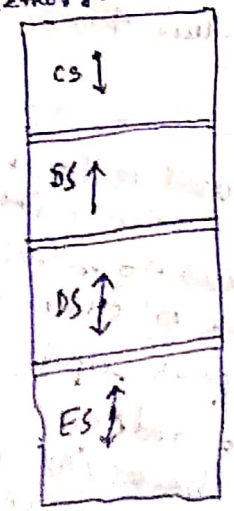


fig2

We can initilize any segments with diff. values. The reg. of 8086 are used to put initial location value of each segment.

fig3 shows diff registers.

The max. size of any seg. is 2^{16} = 64KB
∴ 0000 → FFFF possible offset-reg values.
or, possible location of each seg

The min. size of any seg is 16 byte.
Say CS reg. is 5134, & offset can be 0000 to 000F
What will be next segment address?
Ans: [5134 × 10] + 0000 = 51340
[5134 × 10] + 000F = 5134F
∴ Next seg. can start from 51350

[16 byte is min. size of any segments]

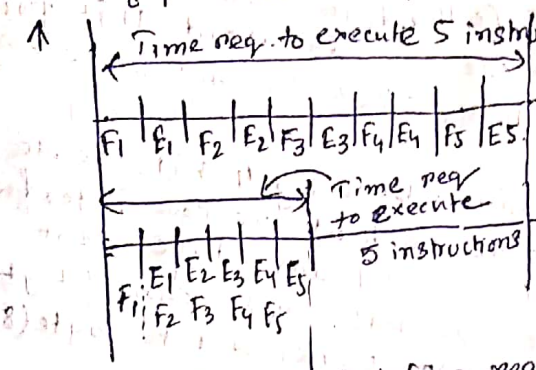
fig3

** How overwriting problem is solved here??
Ans. As we can see here, after initializing of diff. segments in our program we just need to vary the offset value. So, offset value can be 0000-FFFF(64KB) that's how we can't go beyond this offset value, Hence we can't cross any segments, there will be no overwriting problem.

— * —

8086 up.

① Pipelining → To make faster processor, we can increase speed of processor by increasing (i) clock freq. (3.2 MHz 8085 to some GHz now a days), (ii) increasing the data bus size (8 bit in 8085 to 64 bit in nowadays). So what is the next process to improved processing speed of processor. That is : Pipelining Process.



F → Fetching of opcode & Decoding.
E → Execution of instruction.

Basic : Fetching required to handle memory
Execution " to use up (ALU Section).

** This pipelining technique is further used by breaking the instruction operation into different units like 2 stage, 3 stage 4 stage ---

** As 8086 architecture is divided into two units (BIU + EU) so we can apply this pipelining process. However we can't apply this concept in 8085 as it has only single unit in architecture.

** Drawbacks of pipelining: (i) Data dependency - If one ins. requires data from other instruction then prog execution will dependent on data.

(ii) Pipelining fails when branching ins. comes into any program. "Branch Prediction Algorithm" (Momentary failure)

As we increase the stage of pipelining then failure time increase due to branching operation.

** Aligned & Misaligned data. (3)

A_{19} .... A_4	A_3	A_2	A_1	A_0	Location	
0 ——— 0	0	0	0	0	0	Aligned (AD) data
	0	0	0	1	1	
	0	0	1	0	2	A.D.
	0	0	1	1	3	
	0	1	0	0	4	A.D
	0	1	0	1	5	
	0	1	1	0	6	A.D
	0	1	1	1	7	

A_0 - is not used to locate any address location in RAM/ROM. Then, it is used to select the even bank or odd bank.

* Aligned Data means up is producing single address for both of the banks (A_{19} ... A_1) so, address 00000_H & 00001_H Aligned location. But, 00001 & 00010_H is Misaligned data.

Ex: 30004, 30005 - Aligned location.
30005, 30006 - Misaligned location.

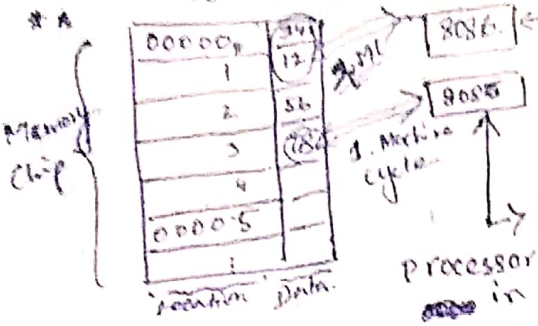
$\overline{BHE}$	A_0	operation
0	0	R/W 16 bit from HB LB
0	1	R/W 8 bit from HB
1	0	R/W 8 bit from LB
1	1	None

** MOV CX [2000]
MOV AL [2000]

2. Memory Banking (Aligned & Misaligned)

** 8086 up is 16 bit processor. So, the size of data bus is 16 bit But this 16 bit data is not stored in a single location. Rather it is located in two different bank of memory. But in 8086 up the data bus is 8 bit size. So, only in single location data can be stored. So, the memory banking/segmentation is required.

8086 - 20 bit Addr Bus

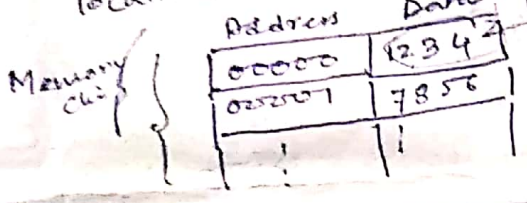


It will access 16 bit no at a time. But this actually requires 2 Machine cycles by up & 8 bit processor. So, in single cycle it can read only 8 bit data. Hence 2 machine cycles are required to read 16 bit data from memory by up.

Here the story is diff. As it is 8 bit processor so, it can read 8 bit data (78) from location in one machine cycle.

* But why should we waste one machine cycle for reading 16 bit data? Why can't we read it in one machine cycle.

** Is it really possible? Ans: YES. If we make memory locations which can contain 16 bit data not 8 bit.



In our M.C we can read 16 bit data & fed it to up 8086.

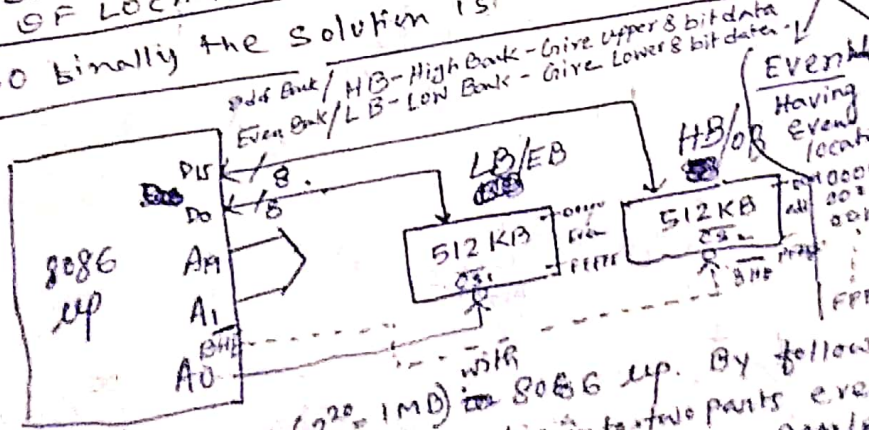
But this is not practical case. Here we waste data location. When we store 8 bit data into these locations.

& processor will be used to handle 16 bit data only.

ONE SINGLE MEMORY LOCATION SHOULD BE AS SMALL AS POSSIBLE. THAT IS 1 BYTE. IN SIZE.

BUT IN TOTALITY WE MAKE HUGE NO. OF LOCATIONS (count) not increase size.

** So finally the solution is Memory Banking



Connect memory (2^{20} = 1MB) with 8086 up. By following memory banking process, we divide the memory chip into two parts even bank & odd bank. So, instead of connecting single 1 MB RAM/ROM with 8086 up, we connect two 512 KB RAM/ROM chips with 8086 up. So that we can create memory banking. Using this two chip we can read 16 bit data at a time by enabling both chip by $\overline{CS}_1$ & $\overline{CS}_2$. Or we can access data of 8 bit from even or odd bank and work with them. So, this makes memory banking process more powerful.

Even Bank - stores even addresses
Odd Bank - Stores odd addresses
EB - Holds lower byte data & OB holds upper byte data