Memory Models and Addresses

Linear Memory Model

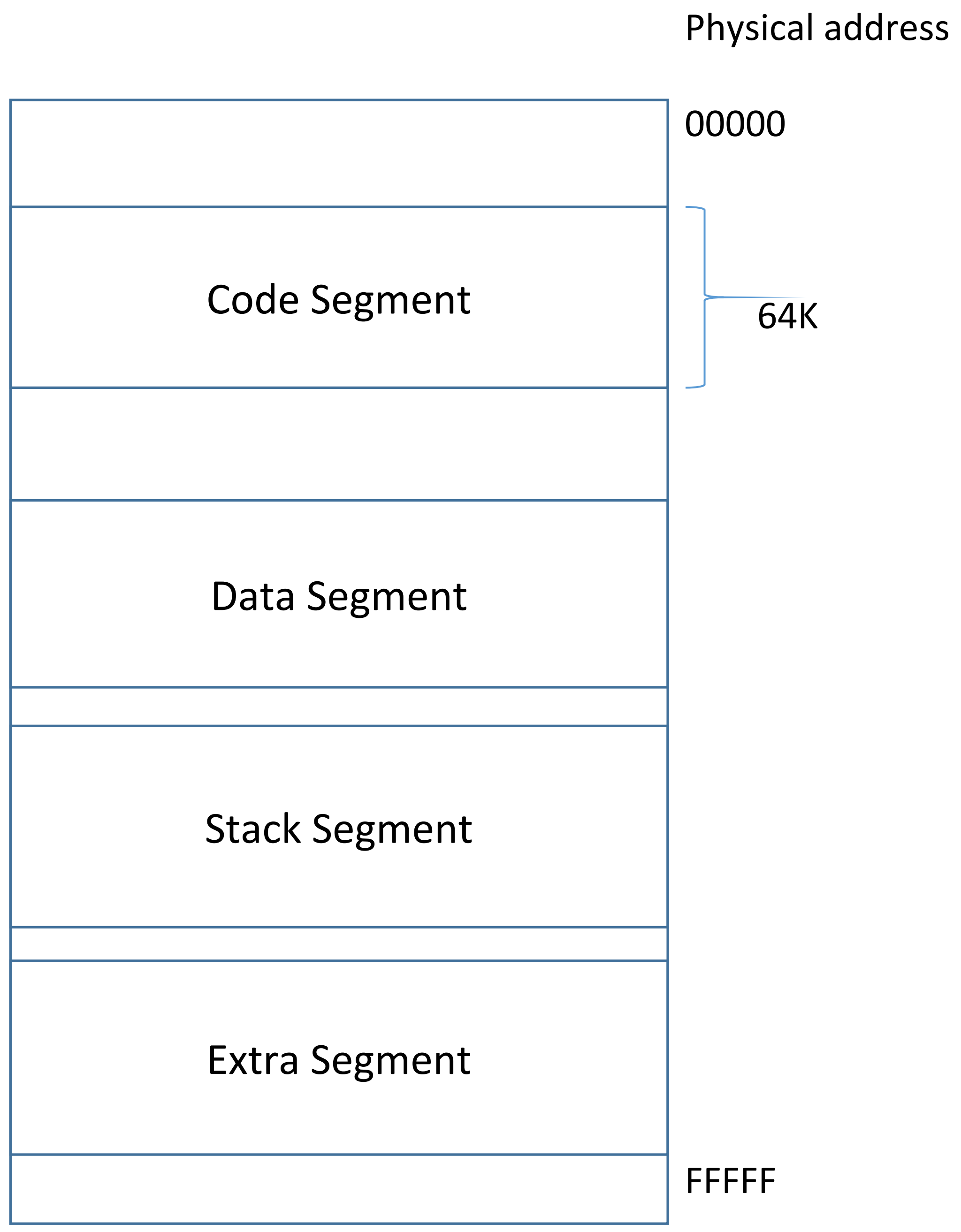
- In linear memory model the whole memory appears like a single array of data
- In earlier processors like 8080 and 8085 the linear memory model was used to access memory
- 8080 and 8085 could access a total memory of 64K using the 16 lines of their address bus.

Segmented Memory Model

- The segmented memory model allows multiple functional windows into the main memory, a code window, a data window etc.
- The processor sees code from the code window and data from the data window.
- For 16 bit processor the size of one window is restricted to 64K, (16 bit register 2^16=64K)
- The four segment registers point to the base of each window.

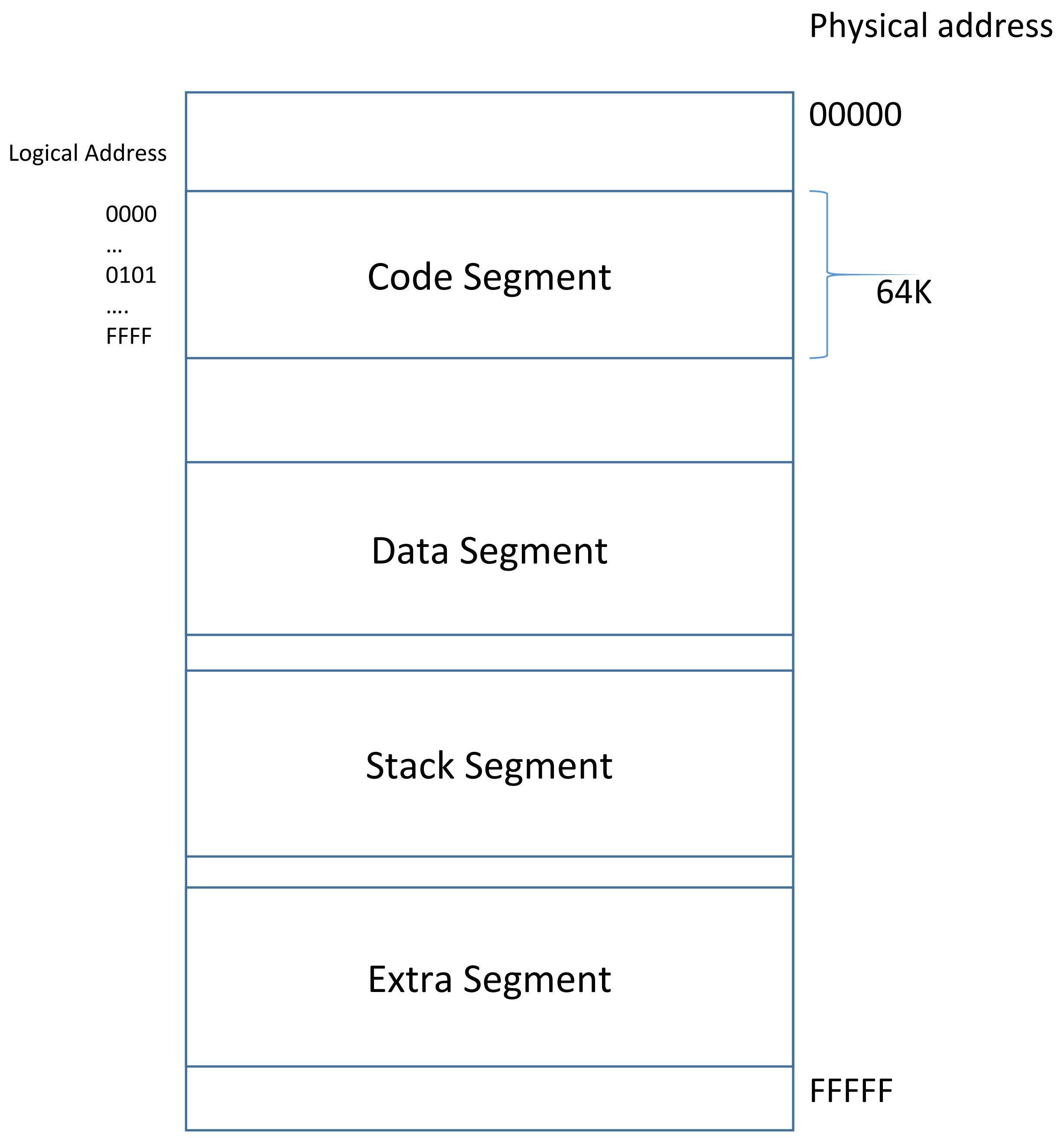
Physical Addresses

- As our memory is 1MB, physical addresses are 20 bit long
- Every address given in our program will be of 16 bit.
- We will need base address of segment and offset calculate physical address in memory.
- Each segment start at some address that is multiple of 16.
  - This is to ensure that the last 4 bits are 0
  - Now the base address of each segment can be represented using 16 bits (instead of 20)
  - This is also called Paragraph Boundaries
- Following registers hold the (16bit) base address of each segment.
  - CS holds the base address of code segment
  - DS holds the base address of data segment
  - SS holds the data
  - ES holds the base of extra segment



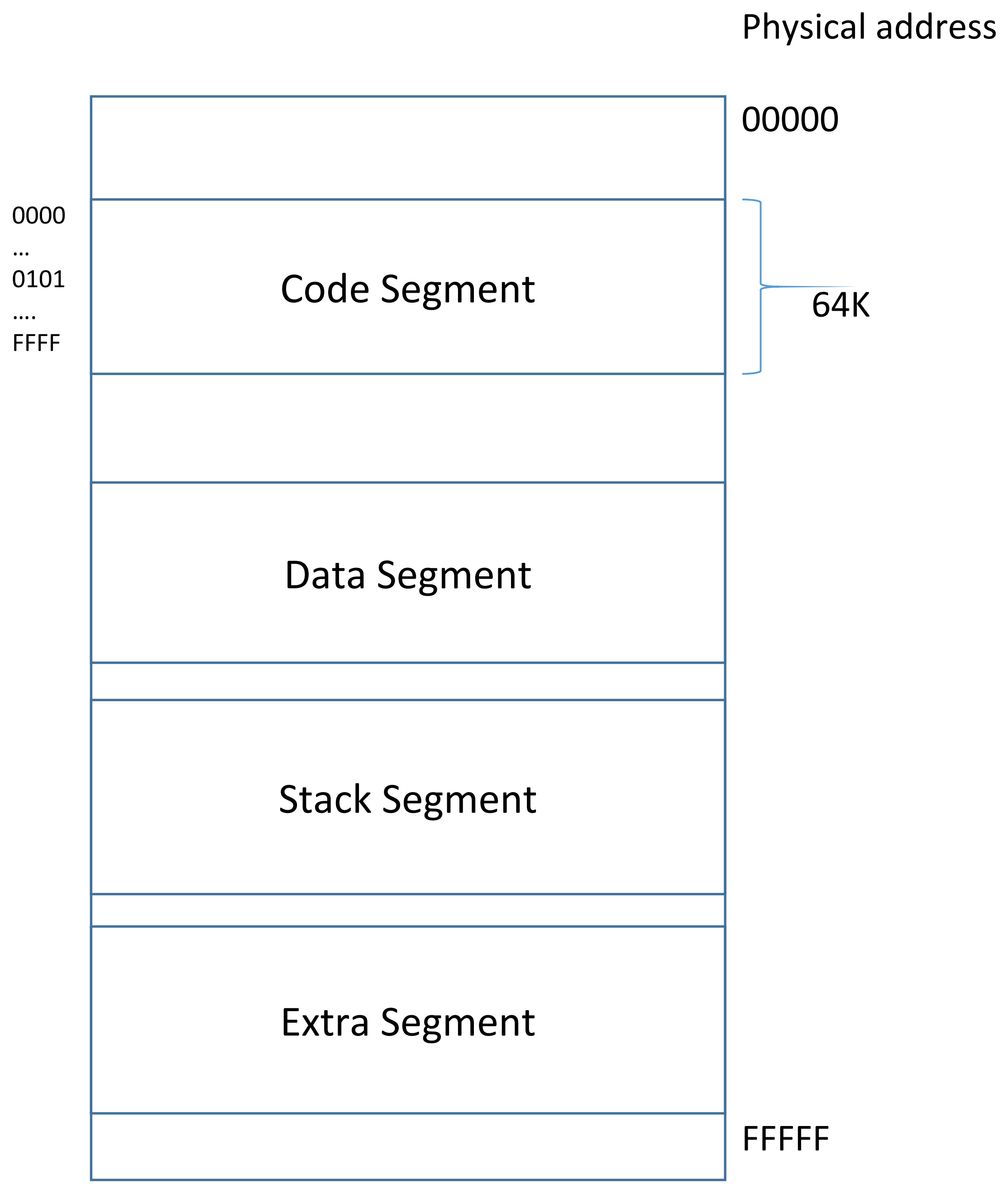
Logical Address

- **Logical Address** Space is the set of all **logical addresses** generated by CPU for a program
- The addresses you have seen in registers so far (e.g IP) are logical addresses.
- In 8088 each address we have seen in registers is of 16 bits, so possible logical addresses are 64k
- For example
    - When IP was equal to 0100 it was the logical address of next instruction to be fetched.
    - When num1 label was 0110 it was logical address starting address of data referred by label num1
- Logical address is converted to Physical address to access the data/instruction from memory



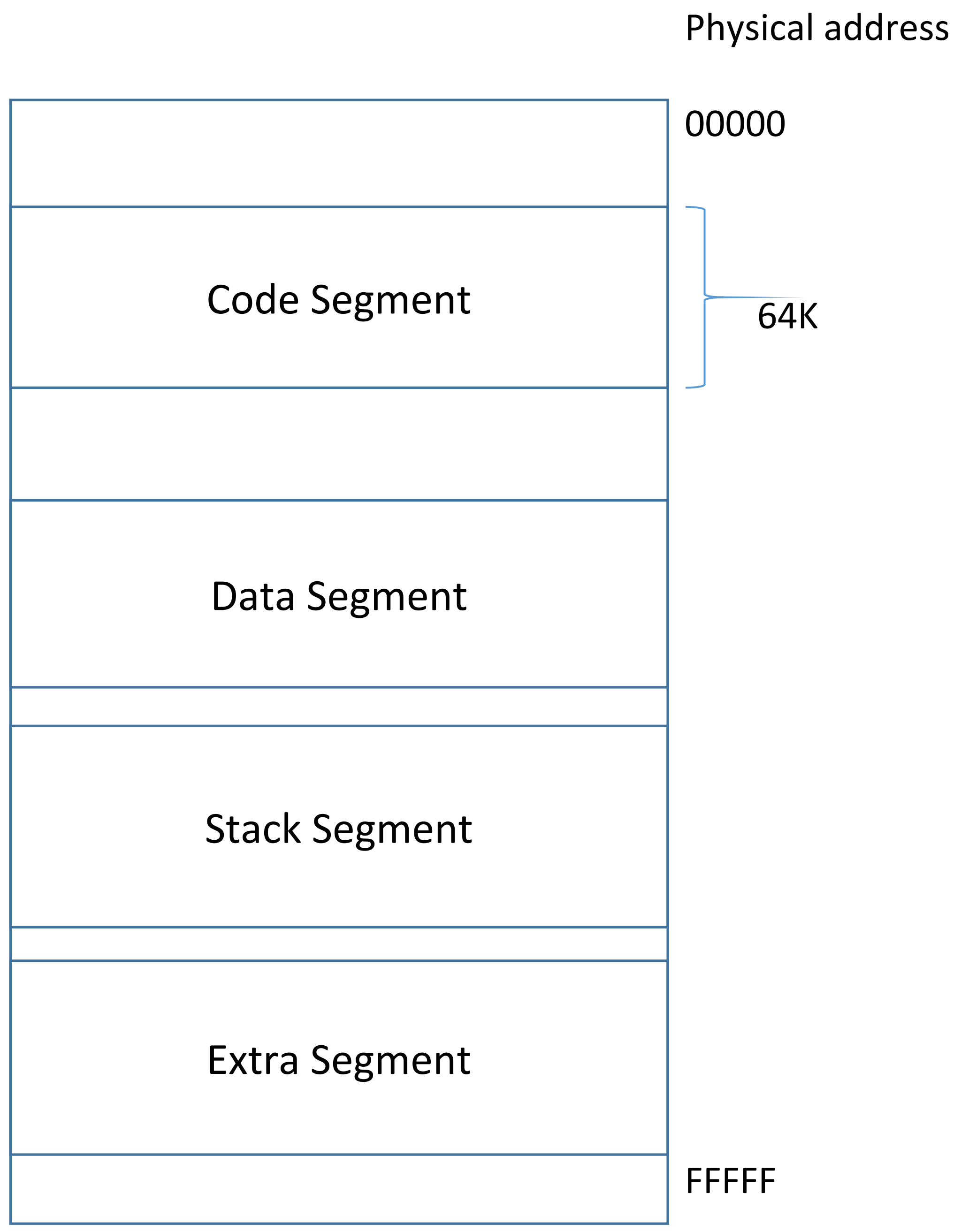
Calculating Physical Addresses

- Lets say that IP is 0100h.
- This mean the instruction to be fetched is at offset 0100h in code segment.
- The 20 bit physical address of this instruction will be calculated by following formula
  - CS*16 + IP
- Here IP is just an offset in Code segment.



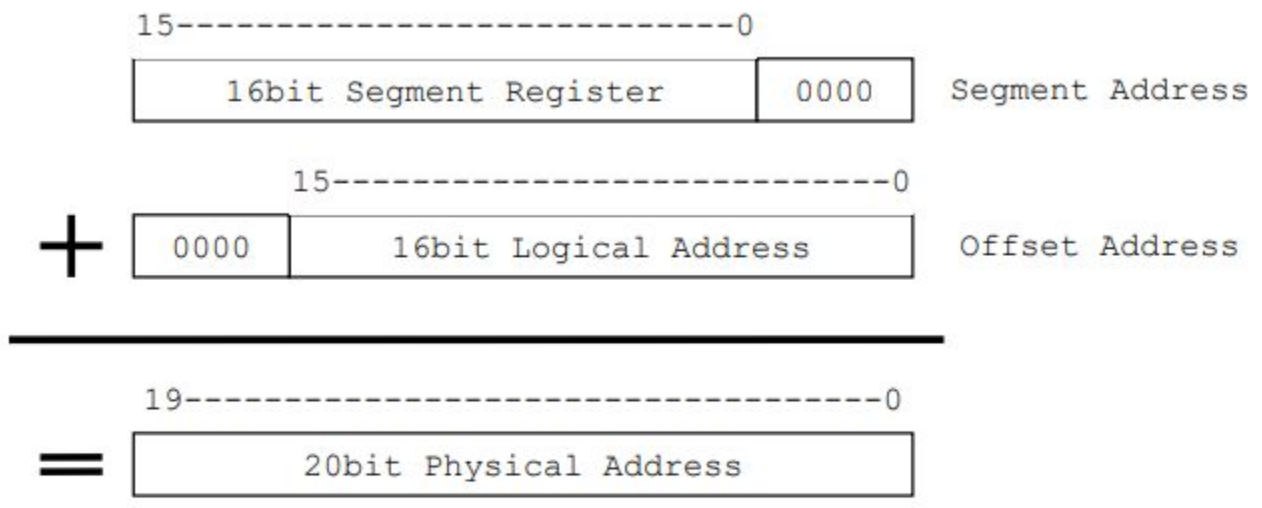
Calculating Physical Addresses

- Similarly if you want to access some data with logical address 0131h.
- Physically it will be located at
  - DS*16+ 0131h
- Here used DS as base because data is stored in data segment.

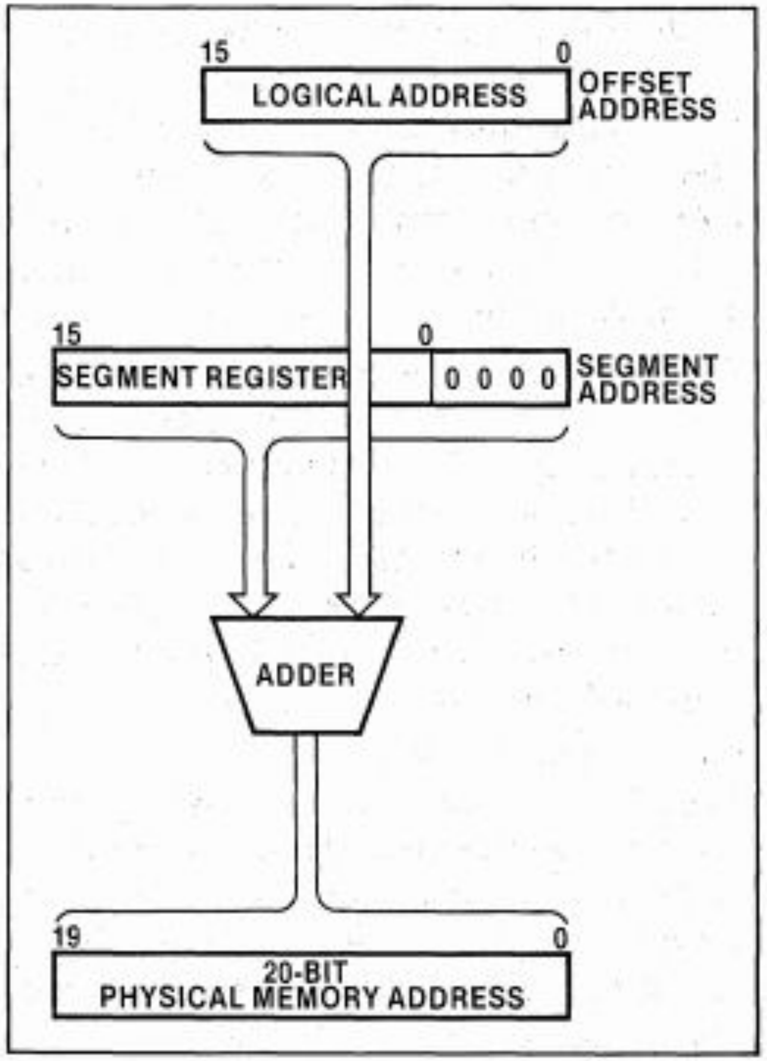


Calculating Physical Address.

- The formula to find the 20 bit physical address given segment's 16 base address and 16 bit offset is
  - BASE *16 + OFFSET
  - This can be seen in figure as



Another figure



Calculating Physical Address.

- Example 1:
  - If IP=0103h and CS=0200h then what is the physical address of the instruction to be fetched?
- Solution:
  - 02000h+00103h =02103h
  - The same calculation can be done in binary
  - 0000 0010 0000 0000 0000 + 0000 0000 0001 0000 0011 = 0000 0010 0010 0000 0010
- Example 2:
  - If a label named num1 has logical address of FFA0h in data segment and DS register is equal to 1BAA then what is the physical address of num1?
- Solution:
  - 1BAA0h + 0FFA0h = 2BA40h
  - The same calculation can be don in binary
  - 0001 1011 1010 1010 0000 + 0000 1111 1111 1010 0000= 0010 1011 1010 0100 0000

Calculating Physical Address.

- Example 3:
  - If a BX logical address of FFA0h stored in it and DS register is equal to 1BAA the which physical address will be used on following statement
    - Mov ax, [bx+1]?
- Solution:
  - 1BAA0h + 0FFA1h = 2BA41h
  - The same calculation can be don in binary
  - 0001 1011 1010 1010 0000 + 0000 1111 1111 1010 0001 = 00101011101001000001

Calculating Physical Address.

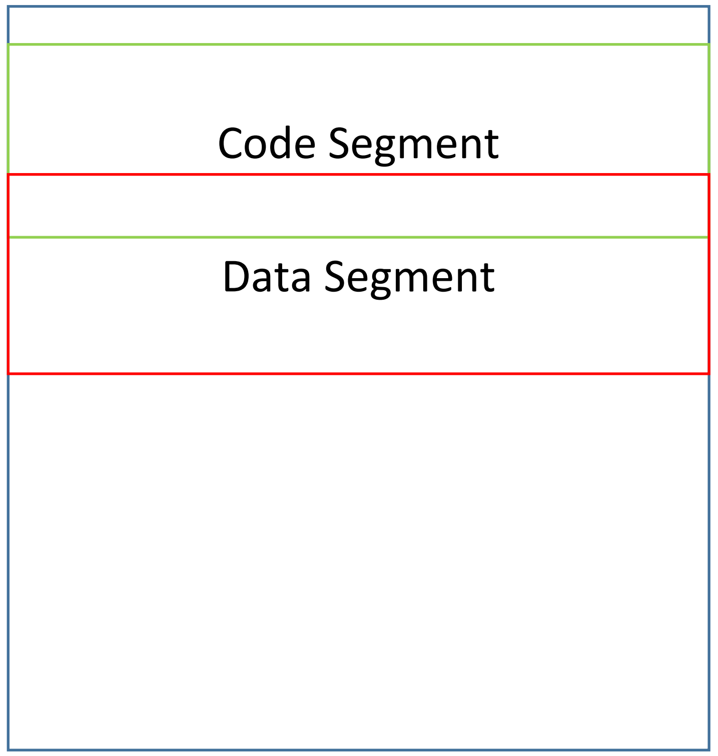
- Note that all these calculation to find physical address will be done by processor.
- The programmers can work with logical addresses.
  - For example when you write

```
mov [num1+10], ax
```

  - Programmer is just telling the logical address to processor
  - Physical location in memory will be calculated by processor.

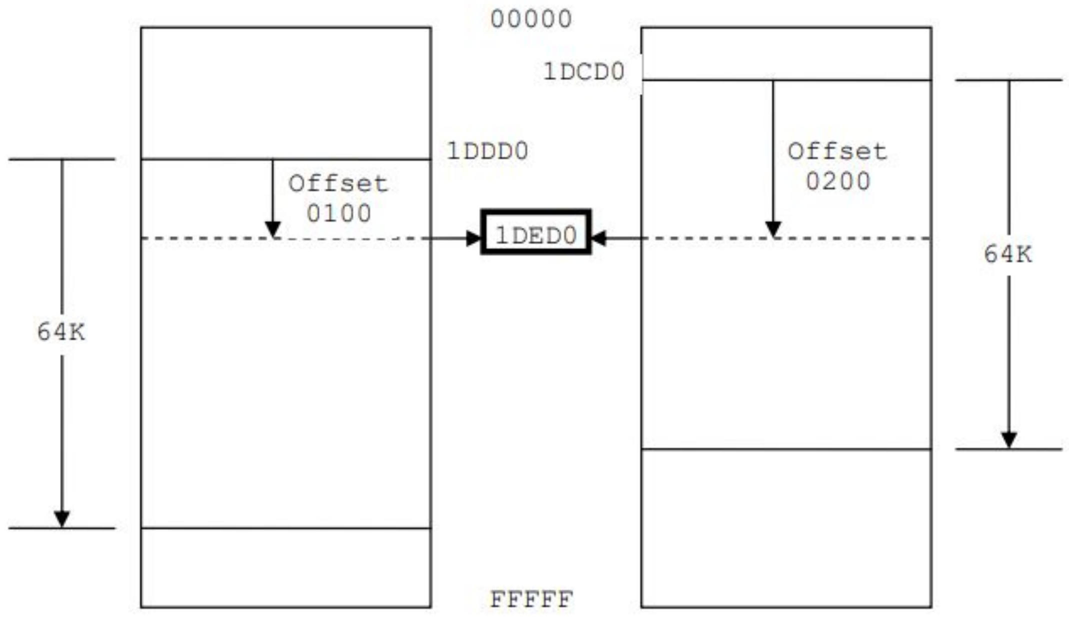
Overlapping Segments

- Consider if data segment starts from 01000 and code segment starts from 01010, and both are 64k long, then both segments will overlap. As shown in figure.
- It is possible to produce same physical address using different combination of segment base and offset as shown in next example.



Overlapping Segments

- The base address of two segments shown in figure are   1DDD0 and 1DCD0 resp.
- Same physical address will be access if offset of 0100 is used with 1DDD0 (also written as 1DDD:0100) and offset 0200 with 1DCD (IDCD:0200)



Segment Association

- There is a default segment associated to every register which accesses memory.
    - Note that in example 1 on slide 3, CS was used as base to find physical address, so CS is associated to IP by default (unchangeable)
    - In example 3, DS was used as base to calculate physical address. So BX is associate with DS by default. Can be changed
    - The list of all these associations is given in next slide

Segment Association

Register	Default associated segment	Flexible
BX, SI, DI	DS	yes
IP	CS	No
BP	SS	Yes
SP	SS	No

- To override the association for one instruction of one of the registers BX, BP, SI or DI, we use the segment override prefix. For example “mov ax, [cs:bx]” associates BX with CS for this one instruction. For the next instruction the default association will come back to act.

ADDRESS WRAPAROUND

- Memory is like a circle, if you start from 0000 an keep on going to next byte by adding 1 then on reaching FFFF adding 1 will take you again to 0000.
- This happens because when adding offset in address, if carry is generated it is dropped.
- Wraparound occurs in while calculating effective as well as physical address. Examples are given in next slide

ADDRESS WRAPAROUND Example

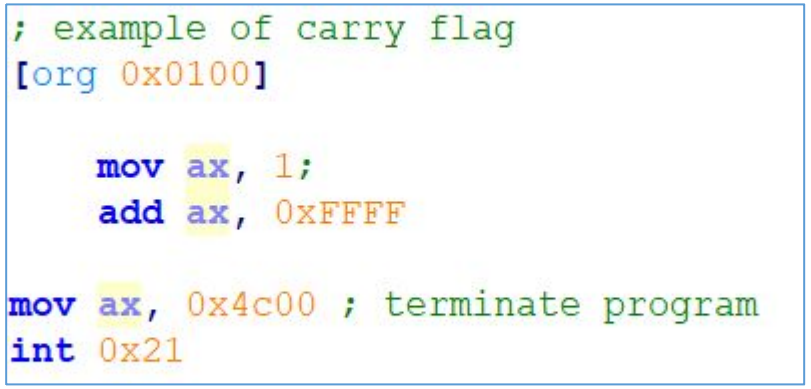
- Effective address:
  - If bx= FFFE then [bx+0003h] will result in effective logical address is 0001. Note that FFFE+0003=1 0001 so carry 1 is dropped and addresses was wrapped around within a segment.

- Physical address:
  - If BX=0100h (some logical address in Data Segment), DS=FFF0h (base address of data segment)
  - Then physical address generate by [bx+0x0100] will be FFF00+(0100+0100)=00100
  - Note that the effective address is calculate before physical address.
  - Note that the carry was dropped and physical address was wrapped around.

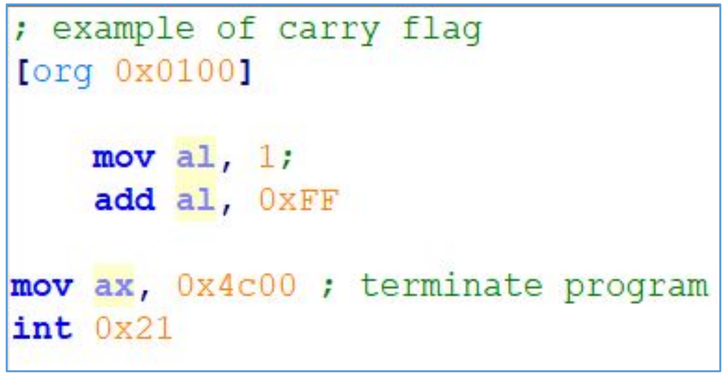
Flag Register with Examples

Carry Flag

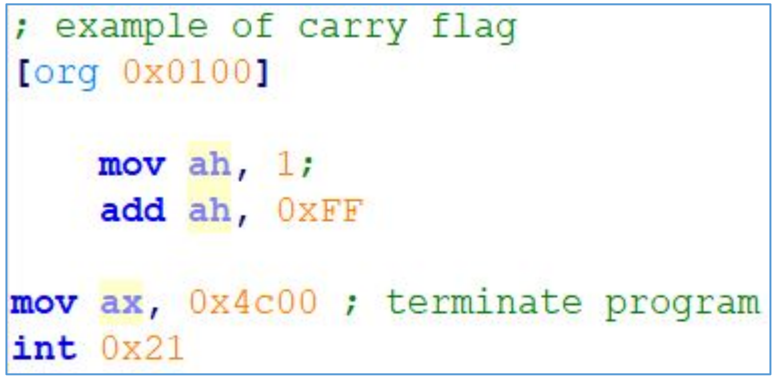
- When two 16bit numbers are added the answer can be 17 bits long or when two 8bit numbers are added the answer can be 9 bits long. This extra bit that won’t fit in the target register is placed in the carry flag where it can be used and tested.
- Examples: in all of the following examples CF should be 1 after addition



```
; example of carry flag
[org 0x0100]

    mov ax, 1;
    add ax, 0xFFFF

mov ax, 0x4c00 ; terminate program
int 0x21
```



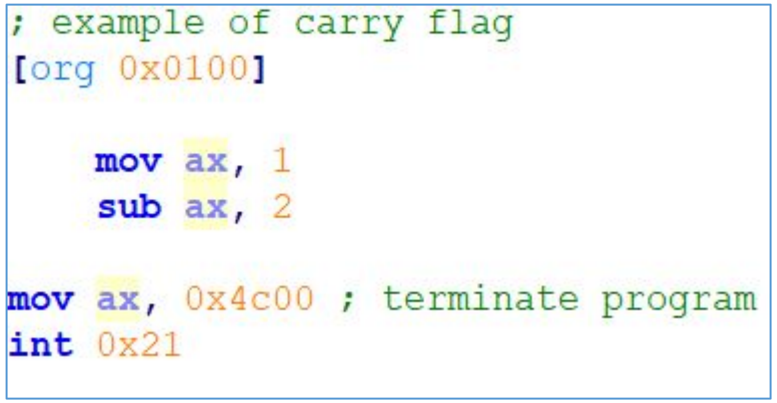
```
; example of carry flag
[org 0x0100]

    mov al, 1;
    add al, 0xFF

mov ax, 0x4c00 ; terminate program
int 0x21
```



```
; example of carry flag
[org 0x0100]

    mov ah, 1;
    add ah, 0xFF

mov ax, 0x4c00 ; terminate program
int 0x21
```

Carry Flag

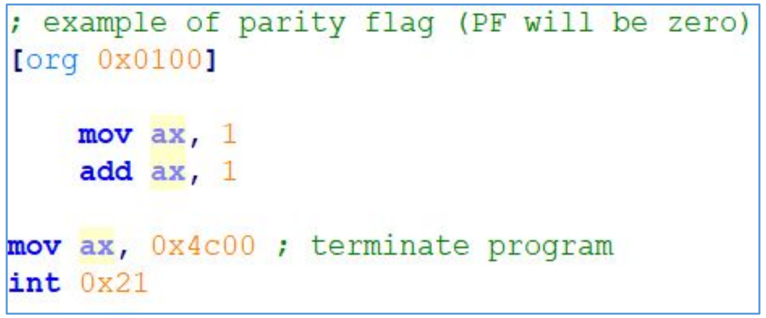
- The carry flag plays the role of borrow during the subtraction operation. And in this condition the carry flag will be set.
- Example: in following example carry flag will be set as 1



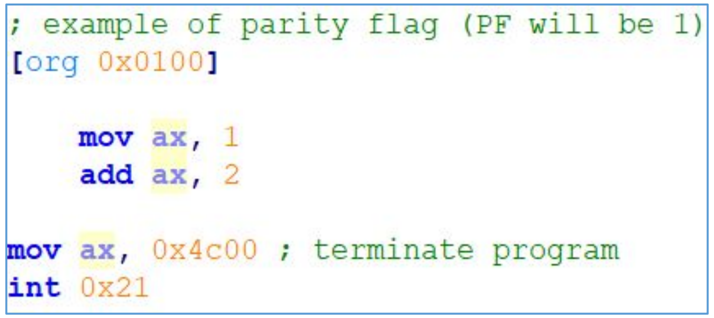
```
; example of carry flag
[org 0x0100]

    mov ax, 1
    sub ax, 2

mov ax, 0x4c00 ; terminate program
int 0x21
```

Parity Flag

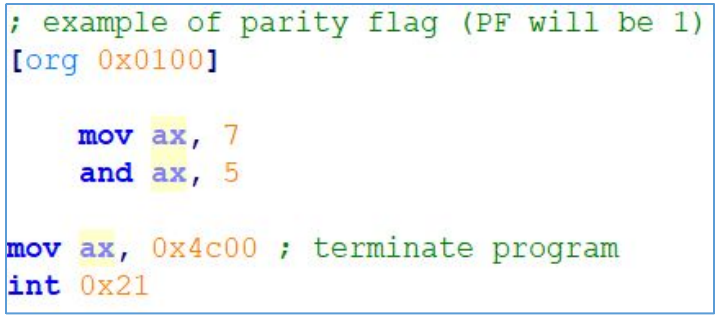
- **Parity flag** indicates if the numbers of set bits is odd or even in the binary representation of the result of the last operation.
- It is only effected by arithmetic and logical operations.
- This information is normally used in communications to verify the integrity of data sent from the sender to the receiver.



```
; example of parity flag (PF will be zero)
[org 0x0100]

    mov ax, 1
    add ax, 1

mov ax, 0x4c00 ; terminate program
int 0x21
```



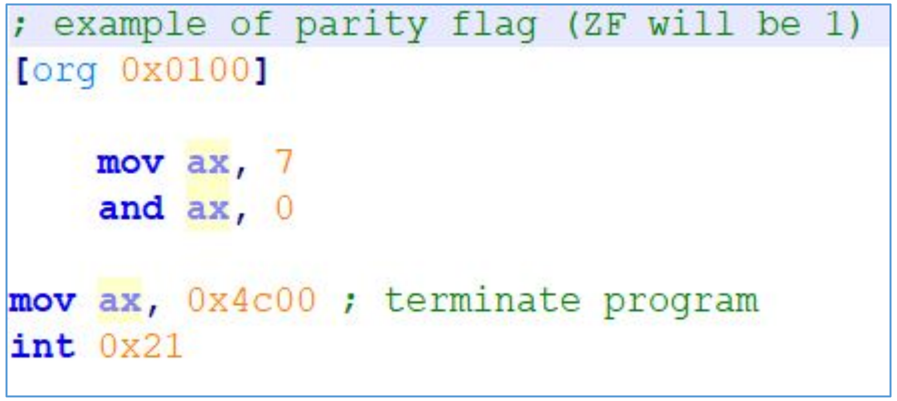
```
; example of parity flag (PF will be 1)
[org 0x0100]

    mov ax, 1
    add ax, 2

mov ax, 0x4c00 ; terminate program
int 0x21
```



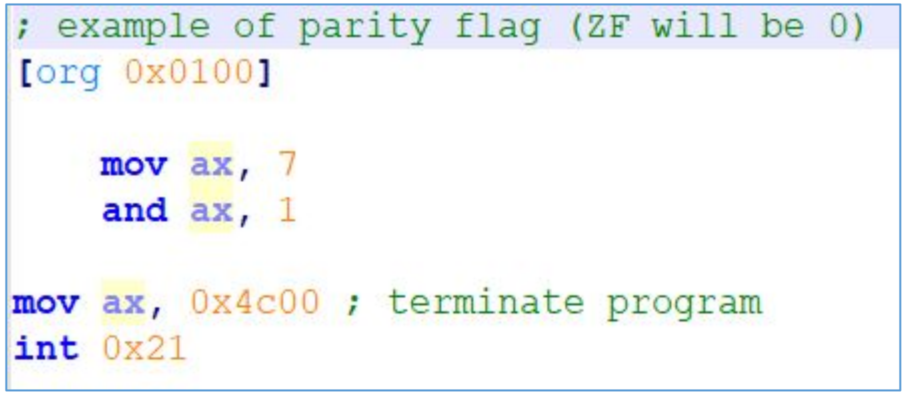
```
; example of parity flag (PF will be 1)
[org 0x0100]

    mov ax, 7
    and ax, 5

mov ax, 0x4c00 ; terminate program
int 0x21
```

Zero Flag

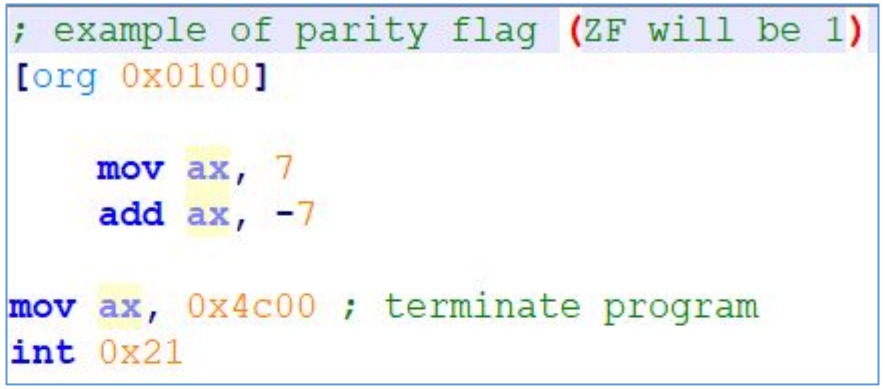
- The Zero flag is set if the last mathematical or logical instruction has produced a zero in its destination.



```
; example of parity flag (ZF will be 1)
[org 0x0100]

    mov ax, 7
    and ax, 0

mov ax, 0x4c00 ; terminate program
int 0x21
```



```
; example of parity flag (ZF will be 0)
[org 0x0100]

    mov ax, 7
    and ax, 1

mov ax, 0x4c00 ; terminate program
int 0x21
```

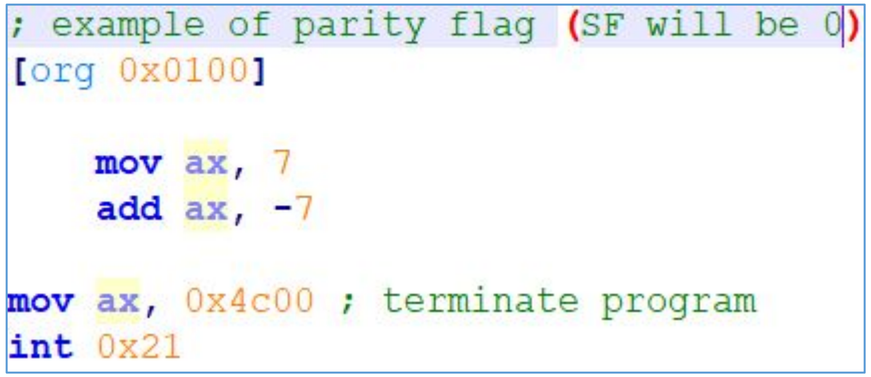


```
; example of parity flag (ZF will be 1)
[org 0x0100]

    mov ax, 7
    add ax, -7

mov ax, 0x4c00 ; terminate program
int 0x21
```

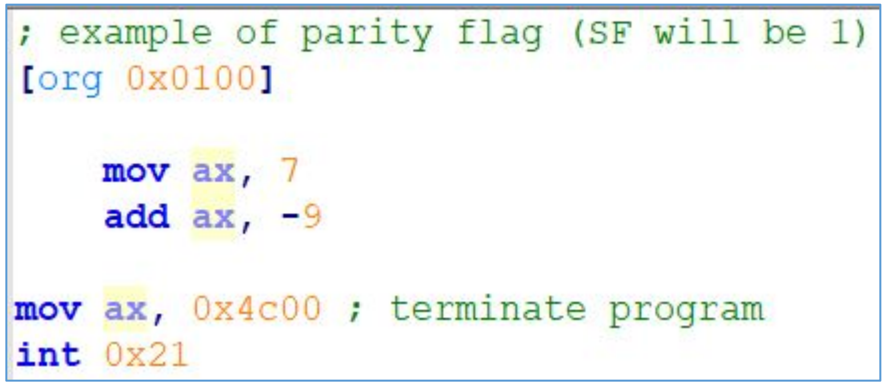
Note the values of -7 in afd or listing file

Sign Flag

- A signed number is represented in its two’s complement form in the computer.
- The most significant bit (MSB) of a negative number in this representation is 1 and for a positive number it is zero.
- The sign bit of the last mathematical or logical operation’s destination is copied into the sign flag.



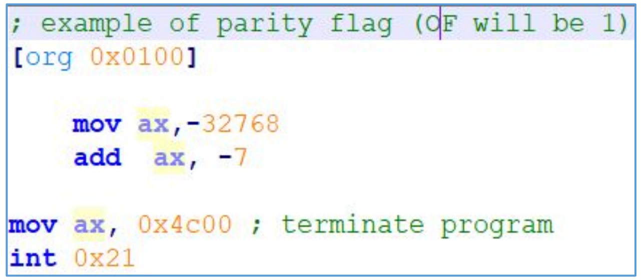
```
; example of parity flag (SF will be 0)
[org 0x0100]

    mov ax, 7
    add ax, -7

mov ax, 0x4c00 ; terminate program
int 0x21
```



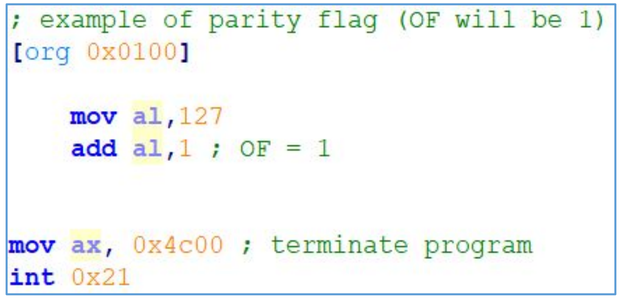
```
; example of parity flag (SF will be 1)
[org 0x0100]

    mov ax, 7
    add ax, -9

mov ax, 0x4c00 ; terminate program
int 0x21
```

Overflow flag

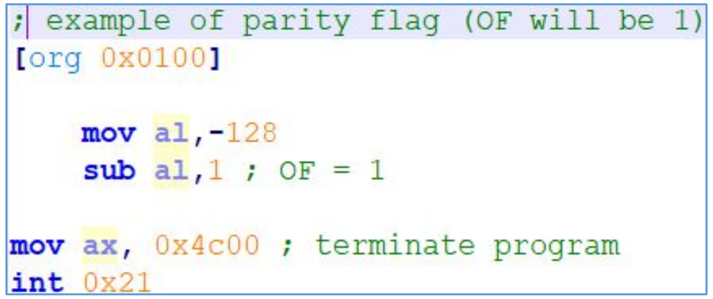
- Set when the result of a signed arithmetic operation is too large or too small to fit into the destination.
- Example 1, if an instruction has a 16- bit destination operand but it generates a negative result smaller than -32,768 decimal, the Overflow flag is set.
- Example 2, we know that the largest possible integer signed byte value is 127; adding 1 to it causes overflow:
- Example 3: Similarly, the smallest possible negative integer byte value is 128. Subtracting 1 from it causes underflow. The destination operand value does not hold a valid arithmetic result, and the Overflow flag is set.



```
; example of parity flag (OF will be 1)
[org 0x0100]

    mov ax,-32768
    add  ax, -7

mov ax, 0x4c00 ; terminate program
int 0x21
```



```
; example of parity flag (OF will be 1)
[org 0x0100]

    mov al,127
    add al,1 ; OF = 1


mov ax, 0x4c00 ; terminate program
int 0x21
```

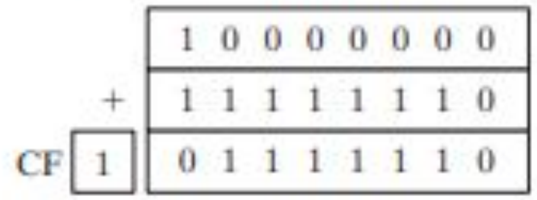


```
; example of parity flag (OF will be 1)
[org 0x0100]

    mov al,-128
    sub al,1 ; OF = 1

mov ax, 0x4c00 ; terminate program
int 0x21
```

How the Hardware Detects Overflow

- The CPU uses an interesting mechanism to determine the state of the Overflow flag after an addition or subtraction operation.
- The value that carries out of the highest bit position is exclusive ORed with the carry into the high bit of the result.
- The resulting value is placed in the Overflow flag.
- In Figure 4-5, shows that adding the 8-bit binary integers 10000000 and 11111110 produces CF = 1, with carryIn(bit7) = 0. In other words, 1 XOR 0 produces OF = 1.

Figure 4–5 Demonstration of how the Overflow flag is set.



- Reference KI

Trick to know value of OF

- Another indication to see the OF will be set is when the MSB of the destination first operand is not same as result.
- Ex

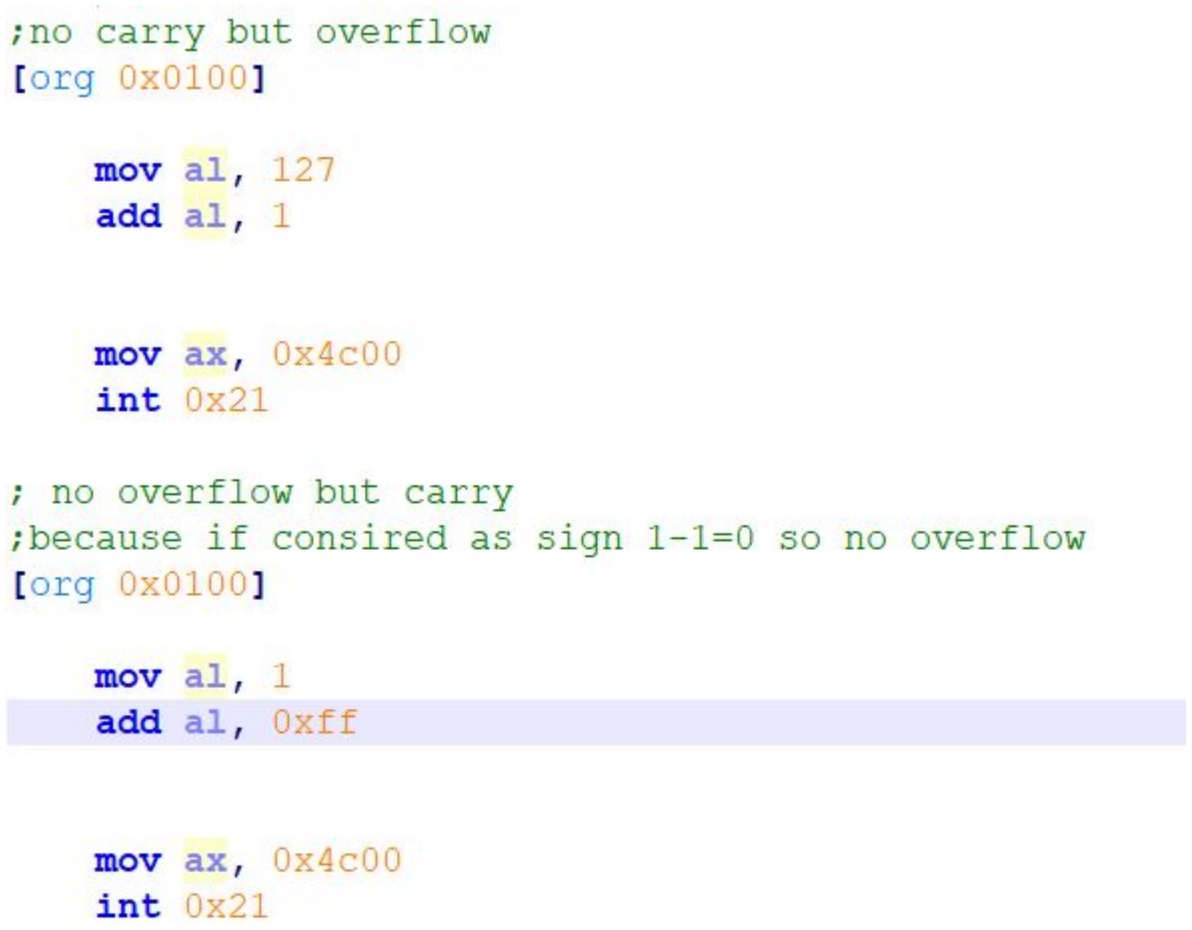
0000 0001 - 1111 1111 = 0000 0010

OF= 0

0000 0001 + 0111 1111 = 1000 0000

OF=1

More examples



```
;no carry but overflow
[org 0x0100]

    mov al, 127
    add al, 1


    mov ax, 0x4c00
    int 0x21

; no overflow but carry
;because if consired as sign 1-1=0 so no overflow
[org 0x0100]

    mov al, 1
    add al, 0xff


    mov ax, 0x4c00
    int 0x21
```