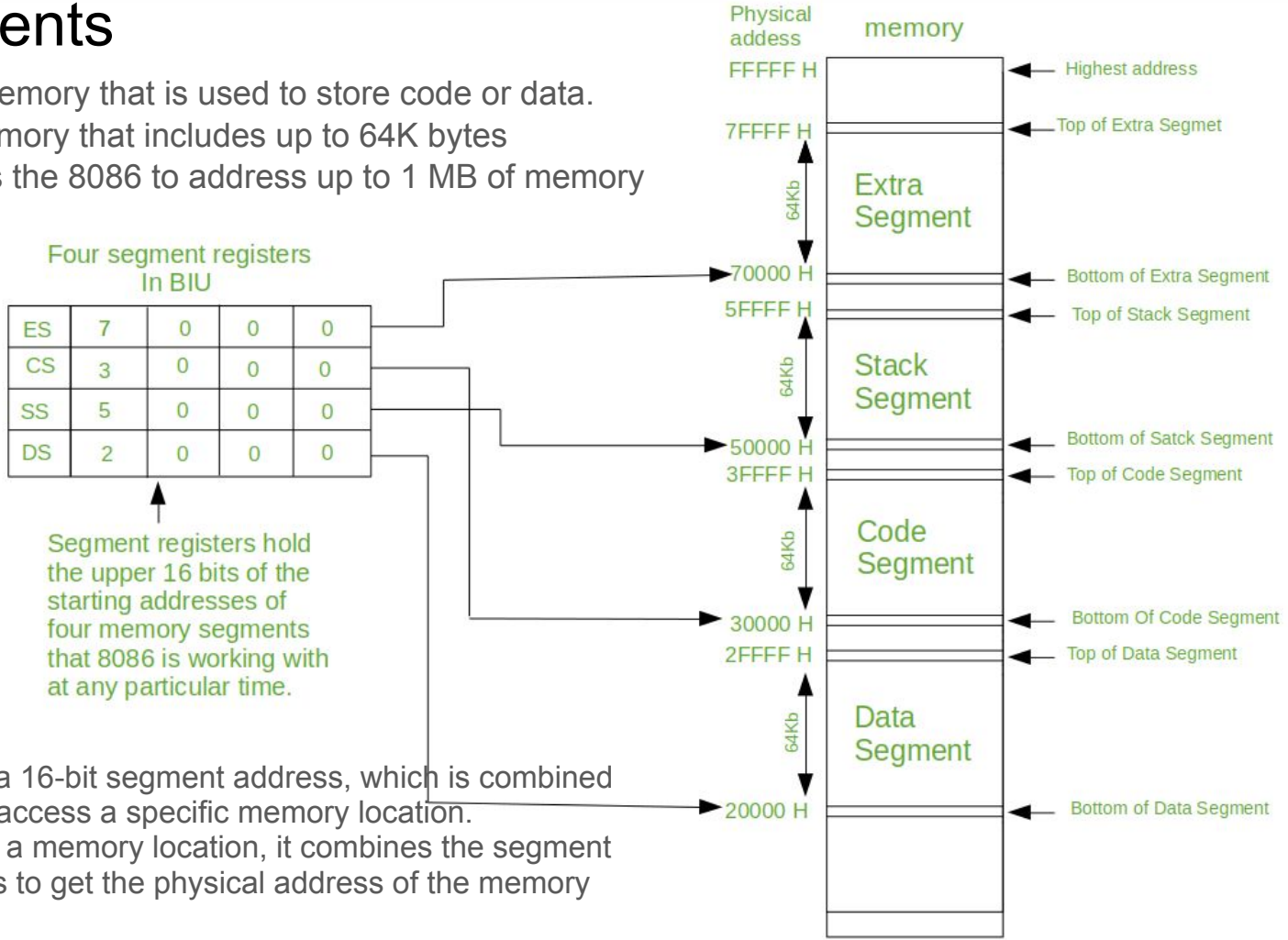
Microprocessors

Program Segments

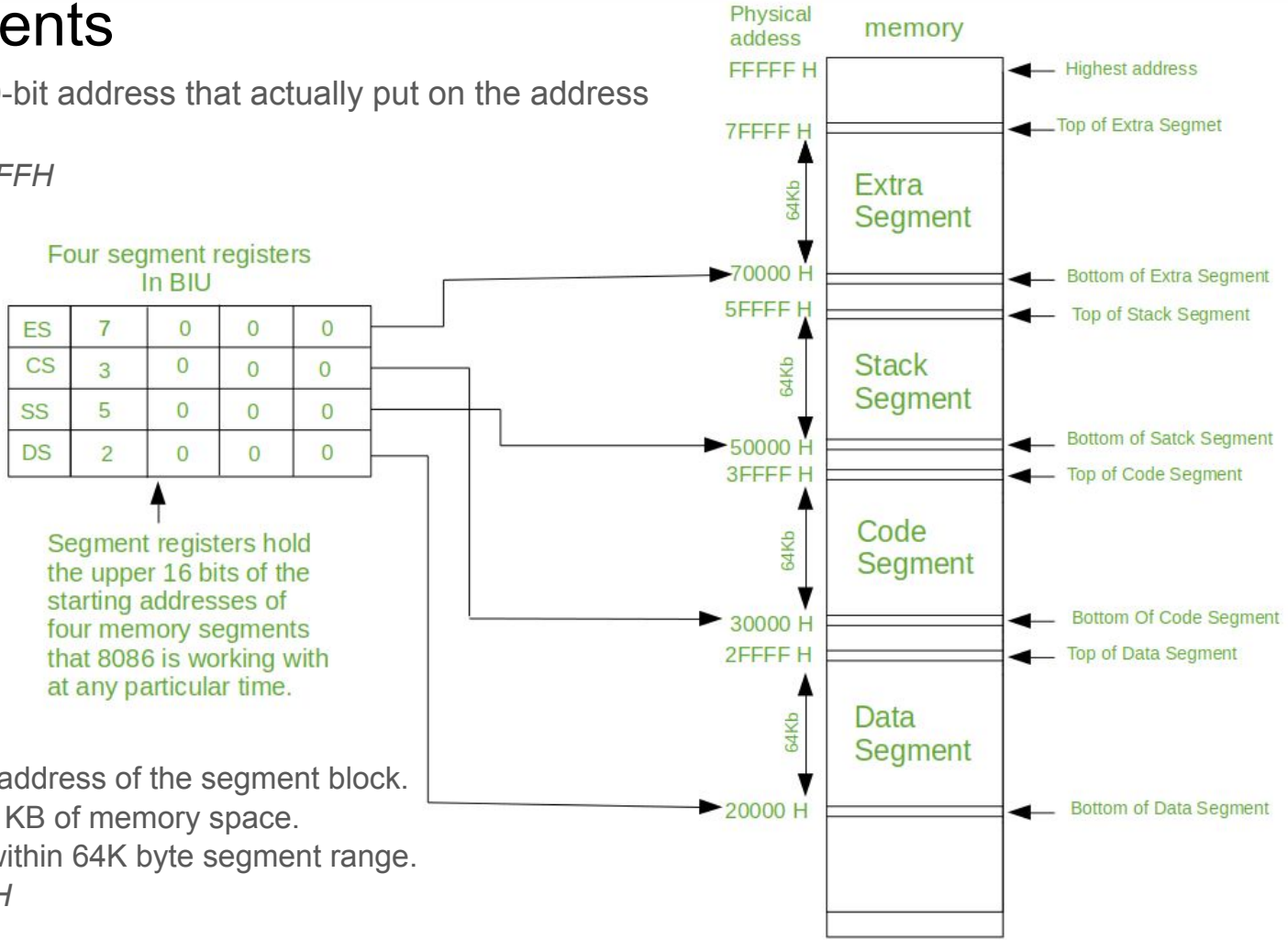
A segment is a section of memory that is used to store code or data.
A segment is an area of memory that includes up to 64K bytes
The use of segments allows the 8086 to address up to 1 MB of memory



Each segment is identified by a 16-bit segment address, which is combined with a 16-bit offset address to access a specific memory location.
When the processor accesses a memory location, it combines the segment address and the offset address to get the physical address of the memory location.

Program Segments

Physical Address is the 20-bit address that actually put on the address bus. (in 8086)
Has a range of 00000H - FFFFFH

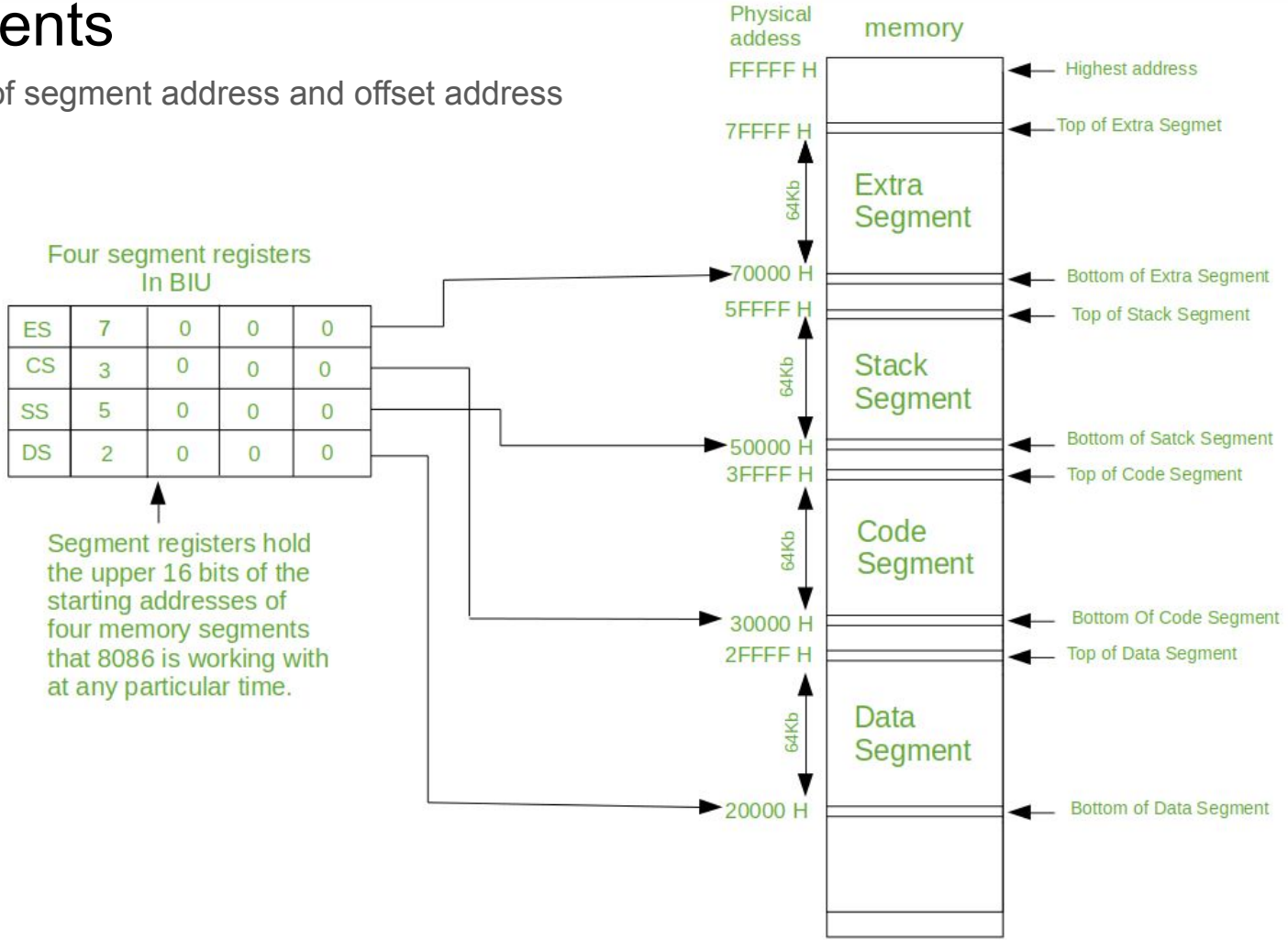


Segment Address is a 16-bit address of the segment block. Each segment is a block of 64 KB of memory space.
Offset Address is a location within 64K byte segment range.
Has a range of 0000H - FFFFH

Program Segments

Logical Address consists of segment address and offset address

For example: CS:IP



Program Segments **Logical and Physical Address**

Calculating physical address:

We want to access the memory location with **offset address** 0x1234 in the **data segment (DS)**. (0x means that the number is hexadecimal format. 0x1234 and 1234h are the same hex numbers)

The DS segment register contains the segment address 0x5000.

To access this memory location, the 8086 would perform the following steps:

1. Shift the segment address 0x5000 left by 4 bits to get 0x50000.
2. Add the offset address 0x1234 to 0x50000 to get the physical address 0x51234.
3. Use the **physical address** 0x51234 to access the memory location.

Logical Address DS:(offset address) → 5000:1234

Physical Address 0x51234

Program Segments

Addressing in Code Segment

- To execute a program, the 8086 fetches the instructions from the **code segment**.
- The **logical address** of an instruction consists CS (Code Segment) and IP (instruction pointer).

CS:IP (16 bit CS and 16 bit IP making total of 32 bits)

Example: If CS register contains 2500H and IP register contains 95F3H. What is the **Logical Address** in the code segment?

CS:IP → **2500:95F3** **(default in adressing is hex. You don't need H)**

Example: If CS register contains 1980H and IP register contains 78FEH. What is the **Physical Address** in the code segment?

Logical address: CS:IP → 1980:78FE

Start with CS	Shift left CS	Add IP	**Physical Address**
1980	19800	19800 + 78FE = 210FE	**210FE**

The microprocessor will retrieve the instruction from the memory locations starting from **210FE** (20 bit address).

Program Segments

Example: If CS=24F6H and IP=634AH, determine:

a) The logical address

b) The offset address

c) The physical address

d) The lower range of the code segment

e) The upper range of the code segment

Program Segments

Addressing in Data Segment

• The area of memory allocated strictly for data is called **data segment.**

• Data segment contains **variables** containing single values and arrays of values, where **code segment** only contain **program instructions**.

▪ **Logical Address** in Data Segment is represented by using **segment address in DS register** and **Offset Address in BX, SI or DI registers**

DS:BX

DS:SI

DS:DI

Program Segments

Example: If DS=7FA2H and the offset is 438EH, determine:

a) The physical address

b) The lower range of the data segment

c) The upper range of the data segment

d) Show the logical address

Program Segments: Overlapping segments

Segments can overlap. For example:

DS = 1000h, Offset = 0010h → Physical = 10010h

ES = 1001h, Offset = 0000h → Physical = 10010h

This allows flexible memory access but requires careful management to avoid data corruption.

Example: Segment register S1 = 2000h, Offset O1 = 0500h. Another segment register S2 = 1FF0h, Offset O2 = ?. Find the value of O2 such that the physical address calculated from (S2:O2) is the same as that from (S1:O1).

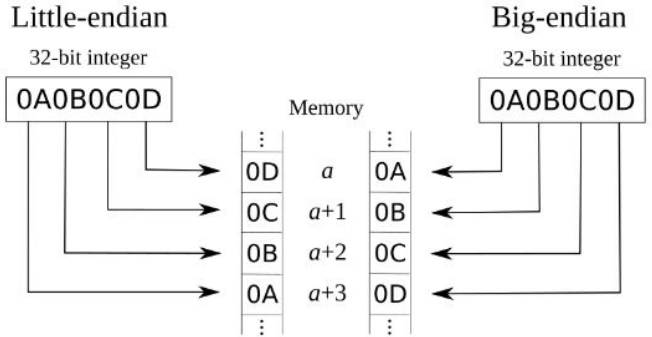
Program Segments: Segment Wrapping

Since each segment is 64 KB, the offset ranges from 0000h to FFFFh. If the offset exceeds this range, it wraps around, which can lead to unexpected behavior if not handled properly.

Example: Compute the physical address of FFFF:0010 before and after applying the 20-bit wrap-around.

Example: Compute the physical address of F800:9000. Does it exceed 1 MB? If yes, what is the wrapped address?

Little Endian vs Big Endian: Byte Ordering



Little endian convention

Given 8-bit (1-byte) data, bytes are stored one after the other in the memory. However given 16-bit (2-bytes) of data how are data stored?

Low byte goes to low address

Example: Storing `0x1234` (16-bit value)

- Memory[0] = `0x34` (LSB)
- Memory[1] = `0x12` (MSB)

This convention is used by **Intel**.

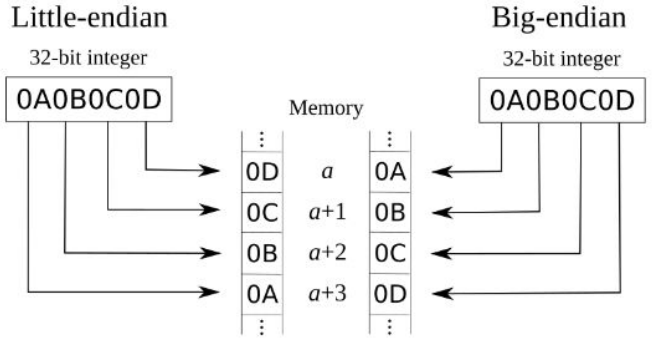
Advantages:

- Easier to incrementally read values (e.g., reading LSB first)
- Common in low-level programming and embedded systems

Disadvantages:

- Less intuitive for human readability
- Can cause confusion in cross-platform data exchange

Little Endian vs Big Endian: Byte Ordering



Big endian convention

Big endian convention is the opposite, where the high byte goes to the low address and low byte goes to the high address.

Example: Storing `0x1234`

- Memory[0] = `0x12` (MSB)
- Memory[1] = `0x34` (LSB)

Motorolla microprocessor uses this convention.

Advantages:

- More intuitive for humans (matches how we write numbers)
- Standard in network communication (known as network byte order)

Disadvantages:

- More complex to handle in some low-level operations
- Less efficient for certain processor architectures

Little Endian vs Big Endian: Byte Ordering

Ex-1: Show how the following values are stored in little-endian memory:

a) DX = 0xABCD

b) 32-bit value 0x89ABCDEF

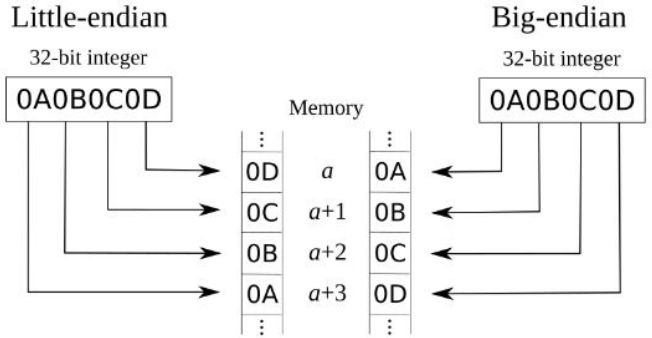
Ex-2: Memory dump (in increasing addresses): 78 56 34 12. Interpret these 4 bytes as:

a) A 32-bit value in little-endian

b) A 32-bit value in big-endian

Ex-3: The number 0x1340F5AD0 is sent to the address with DS:0700 and the offset address is 0x0100. What is the physical address where the MSB side of the number is stored in little endian?

Little Endian vs Big Endian: Byte Ordering



Why Does It Matter?

- **Cross-platform compatibility**: Data exchanged between systems with different endianness must be converted.
- **File formats**: Some binary formats specify endianness (e.g., WAV, BMP).
- **Networking**: Protocols like IP and TCP use big endian, so little endian systems must convert data before transmission.

Little Endian vs Big Endian: Byte Ordering

Example:

A 32-bit integer `0x12345678` is stored in memory on an **Intel x86 system (little-endian)** and then transmitted over a network using **big-endian (network byte order)**.

1. Show the byte order in memory before transmission.
2. Show the byte order as it appears on the network.
3. If the receiving system is also little-endian and does not convert the byte order, what value will it interpret?
4. Explain why protocols like TCP/IP enforce big-endian for transmission.

1. **Memory on little-endian system (Intel x86):**
   Little-endian stores LSB first: Address ↑: 78 56 34 12

   So the order in memory is: `78h 56h 34h 12h`

Little Endian vs Big Endian: Byte Ordering

Example:

A 32-bit integer `0x12345678` is stored in memory on an **Intel x86 system (little-endian)** and then transmitted over a network using **big-endian (network byte order)**.

1. Show the byte order in memory before transmission.
2. Show the byte order as it appears on the network.
3. If the receiving system is also little-endian and does not convert the byte order, what value will it interpret?
4. Explain why protocols like TCP/IP enforce big-endian for transmission.

2. Network transmission (big-endian):
Big-endian sends MSB first: Network order: 12 34 56 78

3. Receiving system (little-endian, no conversion):
It reads the bytes as if they were little-endian: Interpreted value = 0x78563412 This is incorrect because the byte order was not swapped back.

Little Endian vs Big Endian: Byte Ordering

Example:

A 32-bit integer `0x12345678` is stored in memory on an **Intel x86 system (little-endian)** and then transmitted over a network using **big-endian (network byte order)**.

1. Show the byte order in memory before transmission.
2. Show the byte order as it appears on the network.
3. If the receiving system is also little-endian and does not convert the byte order, what value will it interpret?
4. Explain why protocols like TCP/IP enforce big-endian for transmission.

4. Why TCP/IP uses big-endian: Big-endian (network byte order) is a **standardized convention** to ensure interoperability across different architectures. Without this, systems with different endianness would misinterpret multi-byte values like IP addresses and port numbers.

Addressing modes

❖ **Addressing mode: The Various formats of specifying the operands are called addressing modes.**

✓ Immediate Addressing Mode

❑ In this Addressing mode, data (1byte/2bytes) specified in instruction is directly transferred into register.

Example:

```
MOV CL,15H        ; 15H will gets transferred to CL
MOV BX,1000H      ; 1000H will gets transferred to BX
```

✓ Register Addressing Mode

❑ In this Addressing mode, operands are specified in registers only.

Example:

```
MOV CL,DL         ; DL will gets copied into CL
MOV AX,BX         ; BX will gets copied into AX
```

✓ Direct Addressing Mode

❑ In this Addressing mode, address of operand is specified in instruction.

Example:

```
MOV CL,[1000H]    ; CL will get data from 1000H address
MOV CX,[1000H]    ; CL & CH will get data from 1000H &
                    1001H, respectively.
```

✓ Implied/Implicit Addressing Mode

❑ In this Addressing mode, the operand is implied in instruction.

Example:

```
STC               ; Set the carry flag
CLD               ; Clear Directional flag
```

Addressing modes

✓ Indirect Addressing Mode

- ❑ In this Addressing mode, the address of operand is stored in registers.

- ❖ In 8086, with Indirect Addressing Mode following categories are there:
  1. Register Indirect Addressing Mode
  2. Register Relative Addressing Mode
  3. Base Index Addressing Mode
  4. Base Relative Plus Index Addressing Mode

1. Register Indirect Addressing Mode

- ❑ In this Addressing mode, operand address will be given by memory pointer.

Example:

```
MOV CL,[BX]     ; CL will take data pointed by BX Address
MOV [BP],CL     ; CL will be copied at Address pointed by BP (On stack)
```

2. Register relative Addressing Mode

- ❑ In this Addressing mode, operand address will be given by memory pointer + 8 bits or 16 bits displacement.

Example:

```
MOV CL,[BX+4]   ; CL will take data pointed by [BX+4] Address
```

3. Base Indexed Addressing Mode

- ❑ In this Addressing mode, operand address will be given by Base Register + Index register.

Example:

```
MOV CL,[BX+SI]  ; CL will take data pointed by [BX+SI] Address
```

4. Base Relative Plus Indexed Addressing Mode

- ❑ In this Addressing mode, operand address will be given by Base Register + Index register + 8 bits or 16 bits displacement .

Example:

```
MOV CL,[BX+SI+4] ; CL will take data pointed by [BX+SI+4] Address.
```

Program Segments

Offset Registers for various Segments

Segment register	CS	DS	ES	SS
Offset register(s)	IP	SI, DI, BX	SI, DI, BX	SP, BP

Segment Override

```
MOV AL,[BX]       ; normally points DS:BX
MOV AL,ES:[BX] ; you can force to point ES:BX

MOV AX,[BP]       ; normally points SS:BP
MOV AX,DS:[BP] ; you can force to point DS:BP
```

Program Segments

Example: DS = 1200h, SS = 1300h, ES = 1400h BX = 0100h, SI = 0020h, DI = 0004h, BP = 0010h

Compute the physical addresses for each instruction and explain which segment is used:

1. `MOV AX, [BX+SI+0010h]`
2. `MOV AX, [BP+DI+0020h]`
3. `MOV AX, ES:[BX+DI+0010h]`
4. `MOV AX, SS:[SI+BP+0008h]`

Segment register	CS	DS	ES	SS
Offset register(s)	IP	SI, DI, BX	SI, DI, BX	SP, BP

Program Segments

Ex.1: What are the segment and physical addresses used for the following two commands? DS = 1200h, SS = 1300h

a) `MOV AL, [BP+SI+10h]`

b) `MOV AL, DS:[BP+SI+10h]`

Program Segments

Ex.2:

```
; Assume: DS=2000h

ORG 0100h

MOV BX, 0020h

MOV SI, 0010h

MOV AX, [BX+SI]
```

```
2000:0029 → 34

2000:0030 → 78

2000:0031 → 56
```

[BX+SI] goes to which physical address? Which two bytes come to AX?

Program Segments

Addressing Stack segment

The **stack** is a special memory area used for temporary storage of data such as return addresses, local variables, and register contents during subroutine calls or interrupts.

In the **8086 architecture**, the stack is implemented in the **Stack Segment (SS)**.

The **SS register** holds the base address of the stack segment.

The **SP (Stack Pointer)** register points to the current top of the stack within the stack segment.

logical address → **SS:SP** or **SS:BP**

The stack in 8086 grows **downward** in memory:

- When you **PUSH** data, the SP is **decremented**.
- When you **POP** data, the SP is **incremented**.

Each entry in the stack is **word-sized (16 bits)** because 8086 is a 16-bit processor.

Program Segments

Addressing Stack segment

Address Calculation for Stack:

Physical Address = SS × 10H + SP

Example:

If SS = 2000H and SP = FFFEH, then:

Physical Address = 2000H × 10H + FFFEH = 20000H + FFFEH = **2FFFEH**

Push & Pop instructions

- PUSH and POP instructions uses Stack segment [Stack Memory].
- With 8086, Top of stack is indicated by Stack pointer.
- Stack works as per Last in 1st out {LIFO}.
- PUSH and POP does not Support Immediate Addressing Mode.
- PUSH and POP operates with 16bits register. {It means BL or BH is not allowed, but you can use BX.}

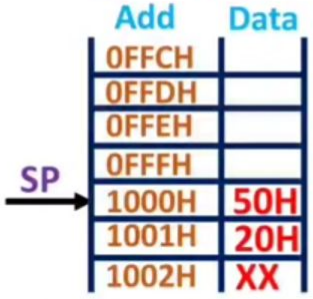
Example:

```
MOV BX, 2050H
PUSH BX
POP CX
```

BH	BL
20H	50H

STACK initially

SS: [SP-1] ← BH
SS: [SP-2] ← BL
SP ← SP-2



STACK after PUSH

CL ← SS: [SP]
CH ← SS: [SP+1]
SP ← SP+2

STACK after POP

CH	CL
20H	50H

- So, After PUSH instruction, Stack Pointer decrement by 2 and it will load data of given register on stack as per Little Endian.
- So, After POP instruction, Stack Pointer increment by 2 and it will load data of stack in given register as per Little Endian.

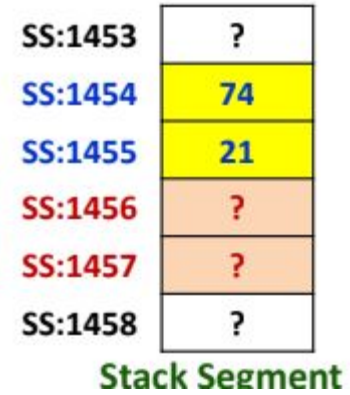
Push & Pop instructions

Example:

Given that SP=1456H, what are the contents of AX, top of the stack and SP after the execution of the following instruction.

```
MOV  AX,2174H

PUSH AX
```



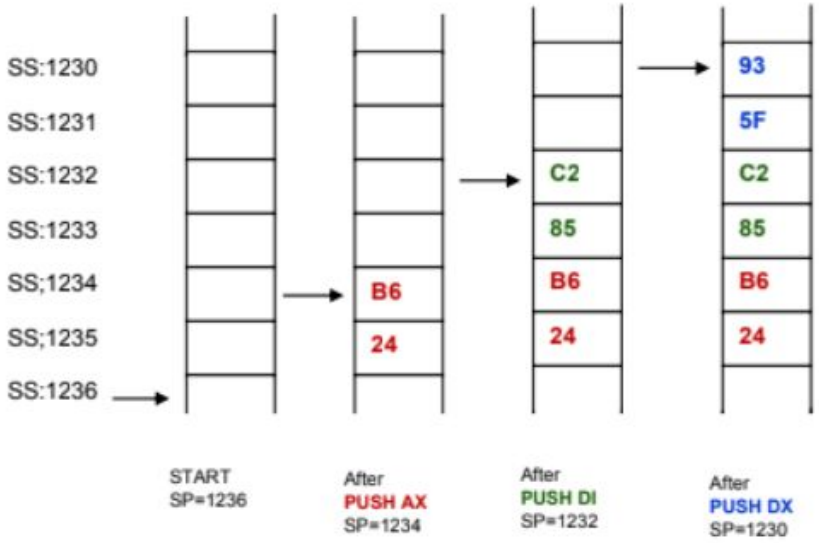
Solution: AX=2174H (stays the same), SP=1454H (decremented by 2),

Push & Pop instructions

Example: SP=1236H, AX=24B6H, DI=85C2H, and DX=5F93H, show the contents of the stack as each of the following instructions is executed.

```
MOV   AX,24B6H
MOV   DI,85C2H
MOV   DX,5F93H
PUSH  AX
PUSH  DI
PUSH  DX
```

```
0700:122F:   00
0700:1230:   93
0700:1231:   5F
0700:1232:   C2
0700:1233:   85
0700:1234:   B6
0700:1235:   24
0700:1236:   00
```



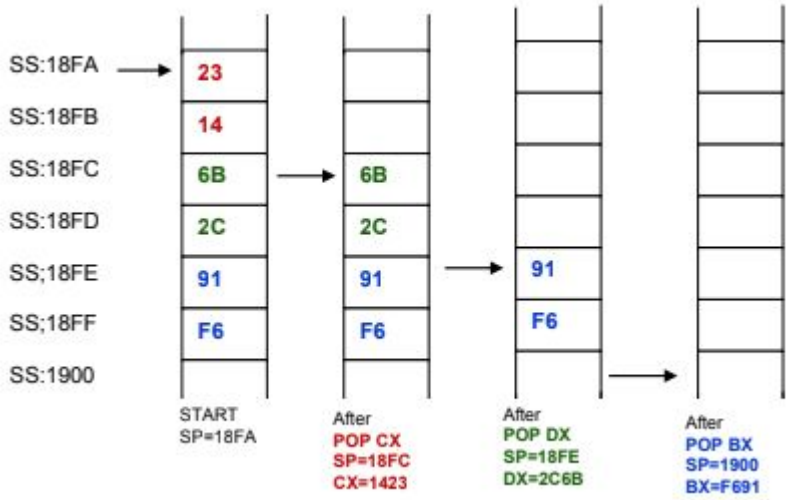
Push & Pop instructions

Example: Assume that the stack is shown below, and SP=18FAH, show the contents of the stack and registers as each of the following instructions is executed.

```
POP CX

POP DX

POP BX
```



Push & Pop instructions

Example: SP=1236H, What will be the content of AX and SP after the execution of the following instruction.

```
MOV  AX,24B6H

MOV  DI,85C2H

MOV  DX,5F93H

PUSH AX

PUSH DI

PUSH DX

POP AX
```