8085 Architecture &
Its Assembly language programming

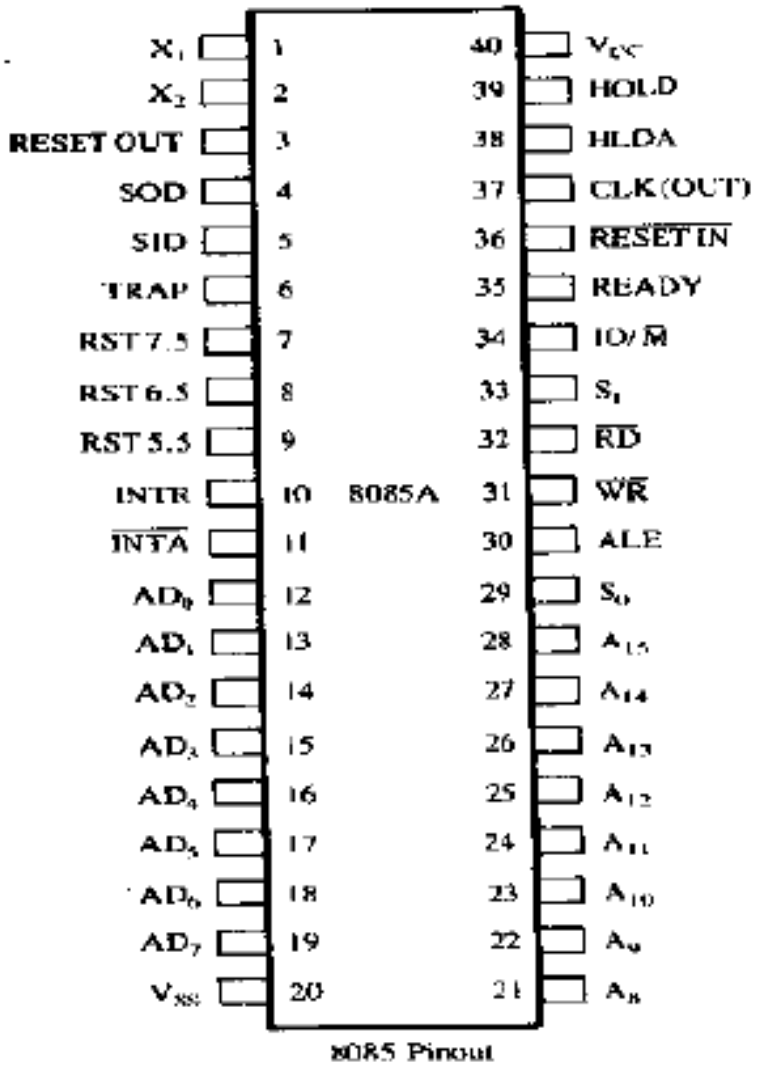
Outline

- 8085 Era and Features
- 8085
  - Block diagram (Data Path)
  - Bus Structure
  - Register Structure
- Instruction Set of 8085
- Sample program of 8085

8085 Microprocessor

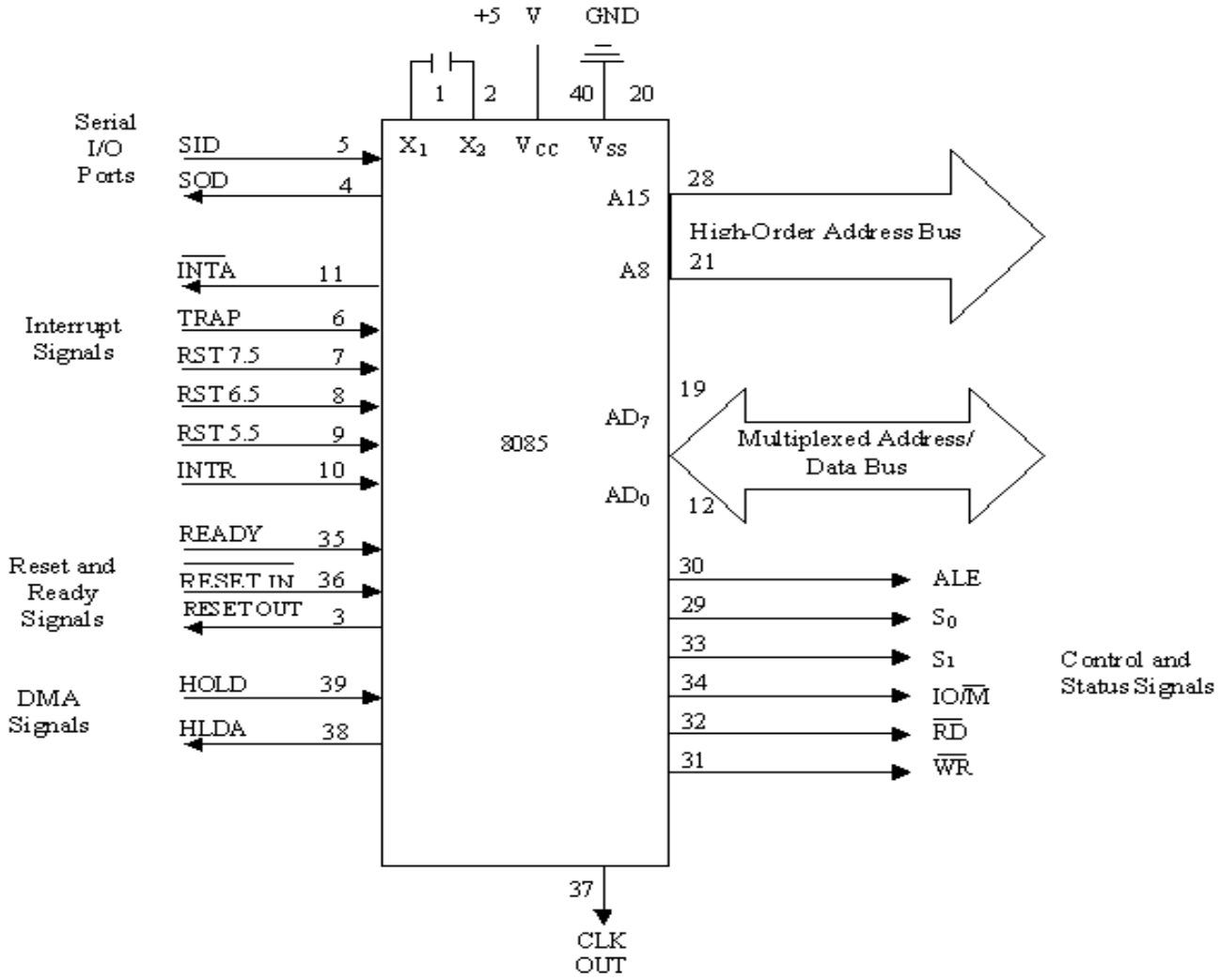
- 8 Bit CPU
- 3-6Mhz
- Simpler design
- ISA = Pre x86 design (Semi CISC)
- 40 Pin Dual line Package
- 16 bit address
- 6 registers: B, C, D, E, H,L
- Accumulator 8 bit

Pin-diagram

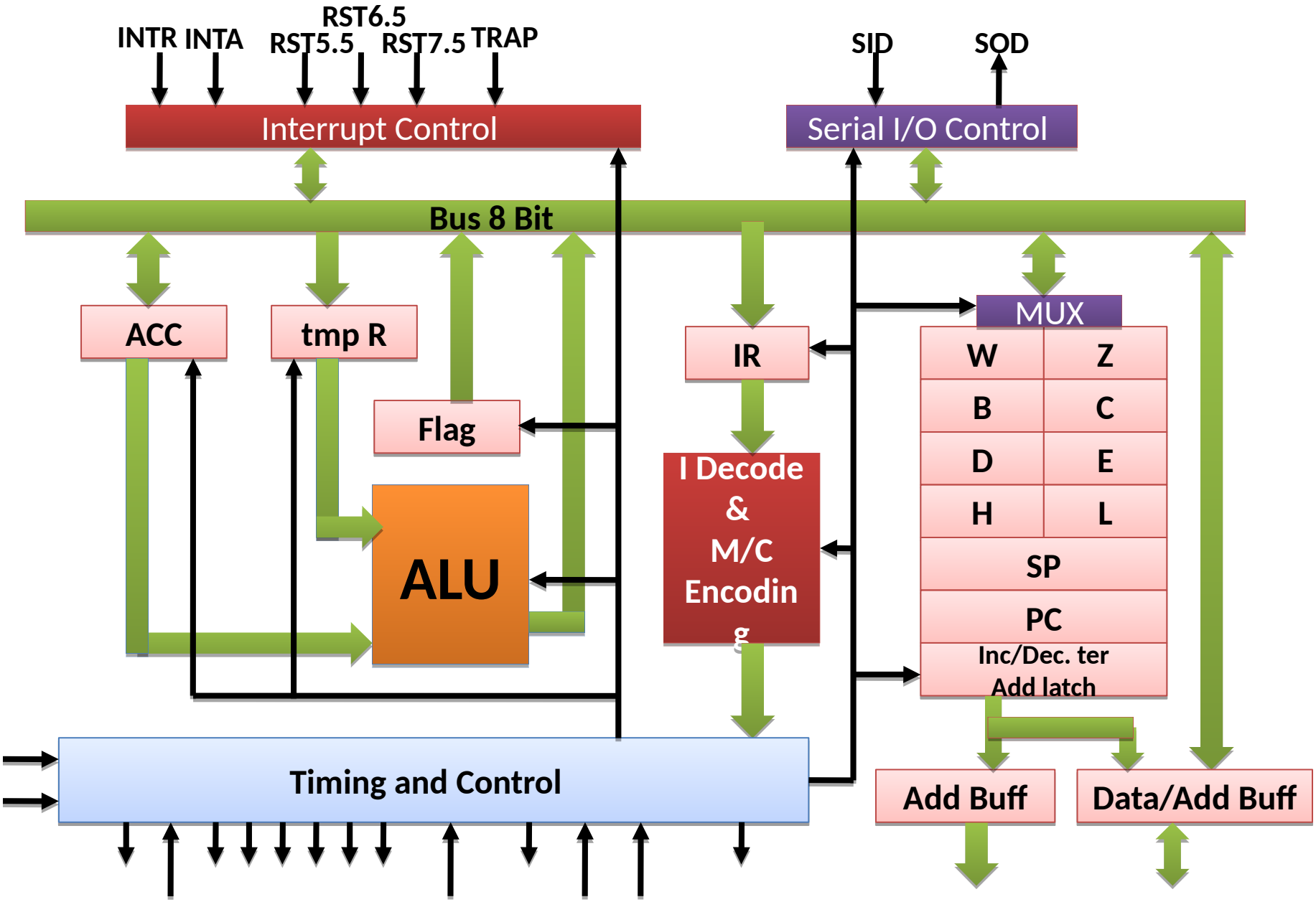


8085 Pinout

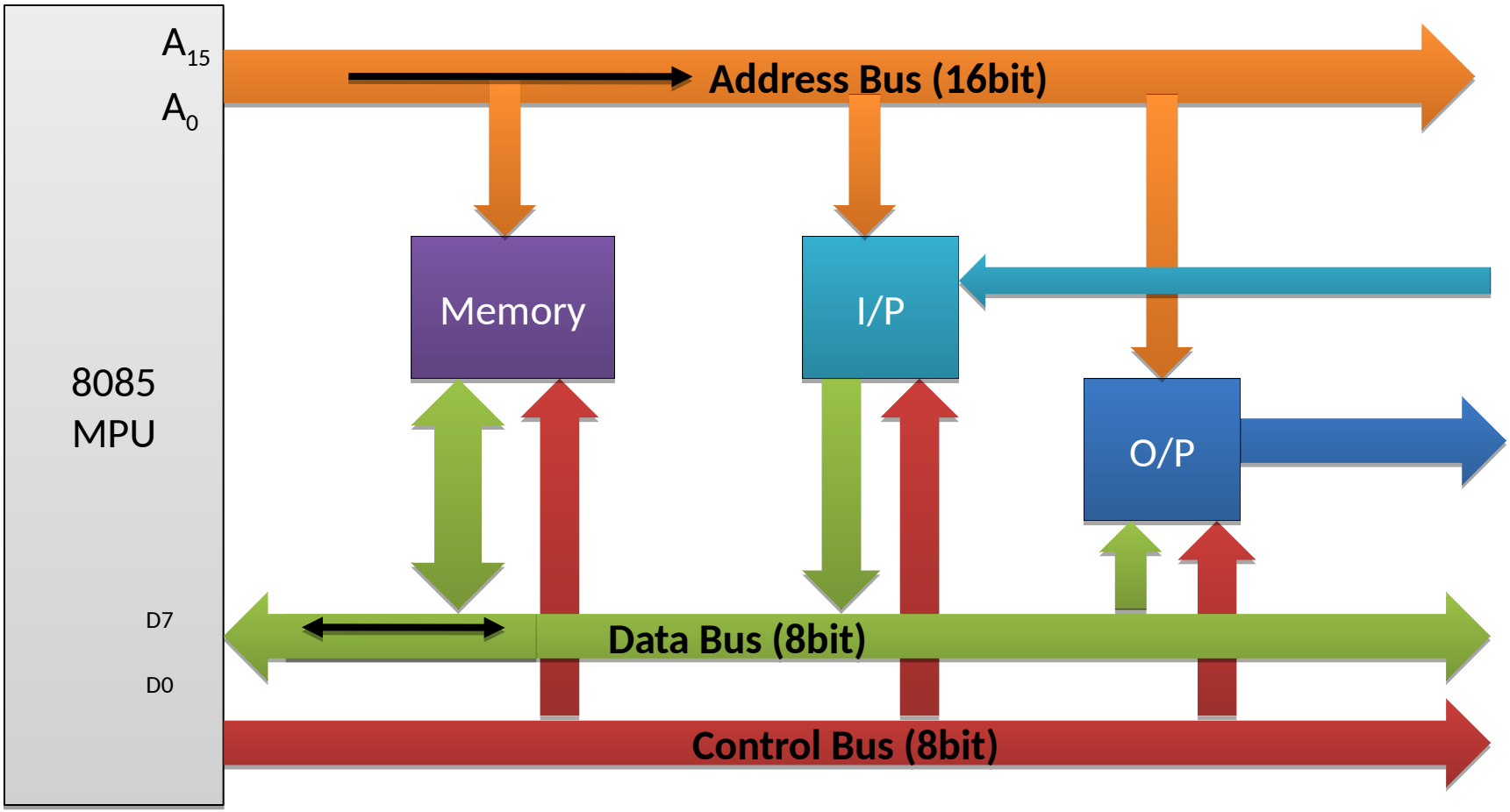
Functional block diagram



8085 Microprocessor Architecture



The 8085 Bus Structure

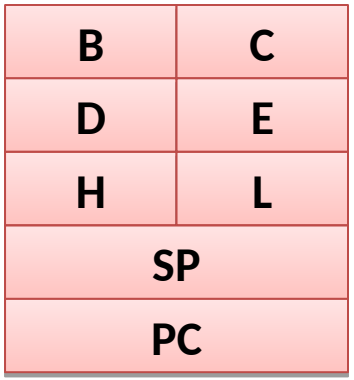


8085 Bus Structure

- Address Bus : Consists of 16 address lines: A_0 – A_{15}
  - Address locations: 0000 (hex) – FFFF (hex)
  - Can access 64K (= 2^{16}) bytes of memory, each byte has 8 bits
  - Can access 64K × 8 bits of memory
  - Use memory to map I/O, Same instructions to use for accessing I/O devices and memory
- Data Bus : Consists of 8 data lines: D_0 – D_7
  - Operates in bidirectional mode
  - The data bits are sent from the MPU to I/O & vice versa
  - Data range: 00 (hex) – FF (hex)
- Control Bus:
  - Consists of various lines carrying the control signals such as read / write enable, flag bits

8085 Registers

- Registers:
  - Six general purpose 8-bit registers: B, C, D, E, H,L
  - Combined as register pairs to perform 16-bit operations: BC, DE, HL
  - Registers are programmable (load, move, etc.)
- Stack Pointer (SP)
- Accumulator & Flag Register
  - (Zero, Sign, Carry, Parity, AuxCarry)
- Program Counter (PC)
  - Contains the memory address (16 bits) of the instruction that will be executed in the next step.

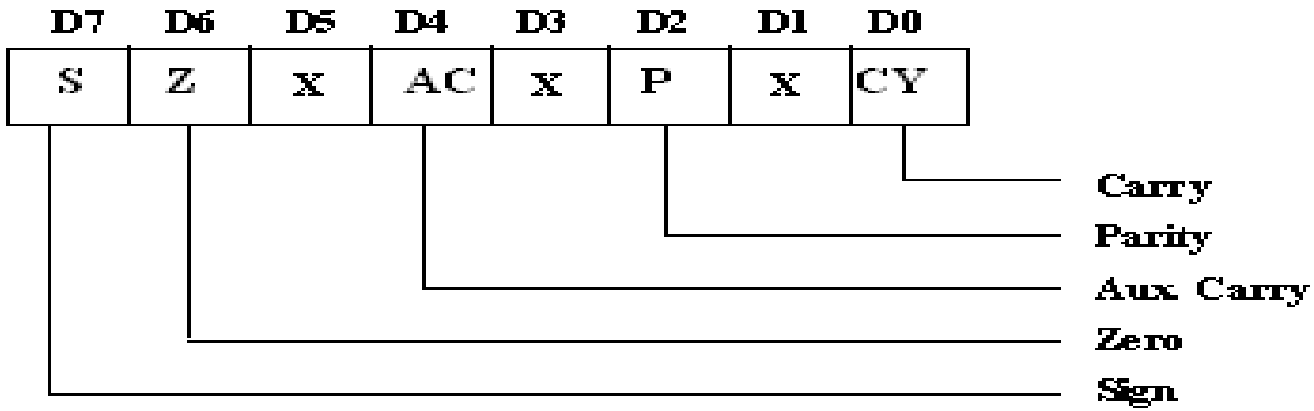


Register Addressing

- The most common form of data addressing.
  - once register names learned, easiest to apply.
- The microprocessor contains these 8-bit register names used with register addressing: AH, AL, BH, BL, CH, CL, DH, and DL.
- 16-bit register names: AX, BX, CX, DX, SP, BP, SI, and DI.

- In 80386 & above, extended 32-bit register names are: EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI.
- 64-bit mode register names are: RAX, RBX, RCX, RDX, RSP, RBP, RDI, RSI, and R8 through R15.
- Important for instructions to use registers that are the same size.
  - *never* mix an 8-bit with a 16-bit register, an 8- or a 16-bit register with a 32-bit register
  - this is not allowed by the microprocessor and results in an error when assembled

Flag Register



Status Signals

IO/M	S1	S0	OPERATION
0	1	1	Opcode fetch
0	1	0	Memory read
0	0	1	Memory write
1	1	0	I/O read
1	0	1	I/O write
1	1	0	Interrupt acknowledge
Z	0	1	Halt
Z	x	x	Hold
Z	x	x	Reset

Reset Signal

- **RESET IN :** When this signal goes low,
  - the program counter (PC) is set to Zero,
  - µp is reset
  - The data and address buses and the control lines are 3-stated during RESET
  - The CPU is held in the reset condition as long as RESET- IN is applied
- **RESET OUT:** This signal indicates that µp is being reset.
  - This signal can be used to reset other devices.
  - The signal is synchronized to the processor clock and lasts an integral number of clock periods.
  - .

Serial communication

- **Serial communication Signal**
  - **SID - Serial Input Data Line: The data on this line is loaded into accumulator bit 7 whenever a RIM instruction is executed.**
  - **SOD – Serial Output Data Line: The SIM instruction loads the value of bit 7 of the accumulator into SOD latch if bit 6 (SOE) of the accumulator is 1.**

DMA Signals

- **HOLD: Indicates** that another master is requesting the use of the address and data buses.
  - The CPU, upon receiving the hold request, will relinquish the use of the bus as soon as the completion of the current bus transfer.
  - Internal processing can continue. The processor can regain the bus only after the HOLD is removed.
  - When the HOLD is acknowledged, the Address, Data RD, WR and IO/M lines are 3-stated.

- **HLDA: Hold Acknowledge: Indicates that the CPU has received the HOLD request and that it will relinquish the bus in the next clock cycle.**
  - HLDA goes low after the Hold request is removed. The CPU takes the bus one half-clock cycle after HLDA goes low.

Ready Signal

- **READY: This signal Synchronizes the fast CPU and the slow memory, peripherals.**
- If READY is high during a read or write cycle, it indicates that the memory or peripheral is ready to send or receive data.
- If READY is low, the CPU will wait an integral number of clock cycle for READY to go high before completing the read or write cycle.
- READY must conform to specified setup and hold times.

Interrupts

- 8085 has 5 interrupts priority (from lowest to highest):

- **INTR:** (maskable and non-vectored interrupt). When the interrupt occurs, the processor fetches from the bus one instruction, usually one of these instructions:
  - One of the 8 RST instructions (RST0 - RST7). The processor saves current program counter into stack and branches to memory location N * 8 (where N is a 3-bit number from 0 to 7 supplied with the RST instruction).
  - **CALL: (3 byte instruction).** The processor calls the subroutine, address of which is specified in the second and third bytes of the instruction.

- **RST5.5: (**maskable interrupt). When this interrupt is received the processor saves the contents of the PC register into stack and branches to 2CH address.

- **RST6.5:** (maskable interrupt) When this interrupt is received the processor saves the contents of the PC register into stack and branches to 34H address
- **RST7.5** (maskable interrupt) When this interrupt is received the processor saves the contents of the PC register into stack and branches to 3CH (hexadecimal) address
- **TRAP**: (non-maskable interrupt) When this interrupt **is** received the processor saves the contents of the PC register into stack and branches to 24H address.

- All maskable interrupts can be enabled or disabled using EI and DI instructions. RST 5.5, RST6.5 and RST7.5 interrupts can be enabled or disabled individually using SIM instruction

8085 Non-Vectored Interrupt Process

- The interrupt process should be enabled using the EI instruction.

- The 8085 checks for an interrupt during the execution of every instruction.
- If INTR is high, Processor completes current instruction, disables the interrupt and sends INTA (Interrupt acknowledge) signal to the device that interrupted
- INTA allows the I/O device to send a RST instruction through data bus.
- MP saves the memory location of the next instruction, on the stack and the program is transferred to 'call' location (ISR Call) specified by the RST instruction
- Microprocessor Performs the ISR. ISR must include the 'EI' instruction to enable the further interrupt within the program.

- RET instruction at the end of the ISR allows the MP to retrieve the return address from the stack and the program is transferred back to where the program was interrupted.

Interrupt vectors

- An interrupt vector is a pointer to where the ISR is stored in memory.
- All interrupts (vectored or otherwise) are mapped onto a memory area called the Interrupt Vector Table(IVT).
- The IVT is usually located in memory page 00 (0000H - 00FFH).
- Example:
  - Let a device interrupts the Microprocessor using the RST 7.5 interrupt line.
  - Because the RST 7.5 interrupt is vectored, Microprocessor knows in which memory location it has to go using a call instruction to get the ISR address.
  - RST7.5 is known as Call 003Ch to Microprocessor. Microprocessor goes to 003C location and will get a JMP instruction to the actual ISR address. The microprocessor will then, jump to the ISR location

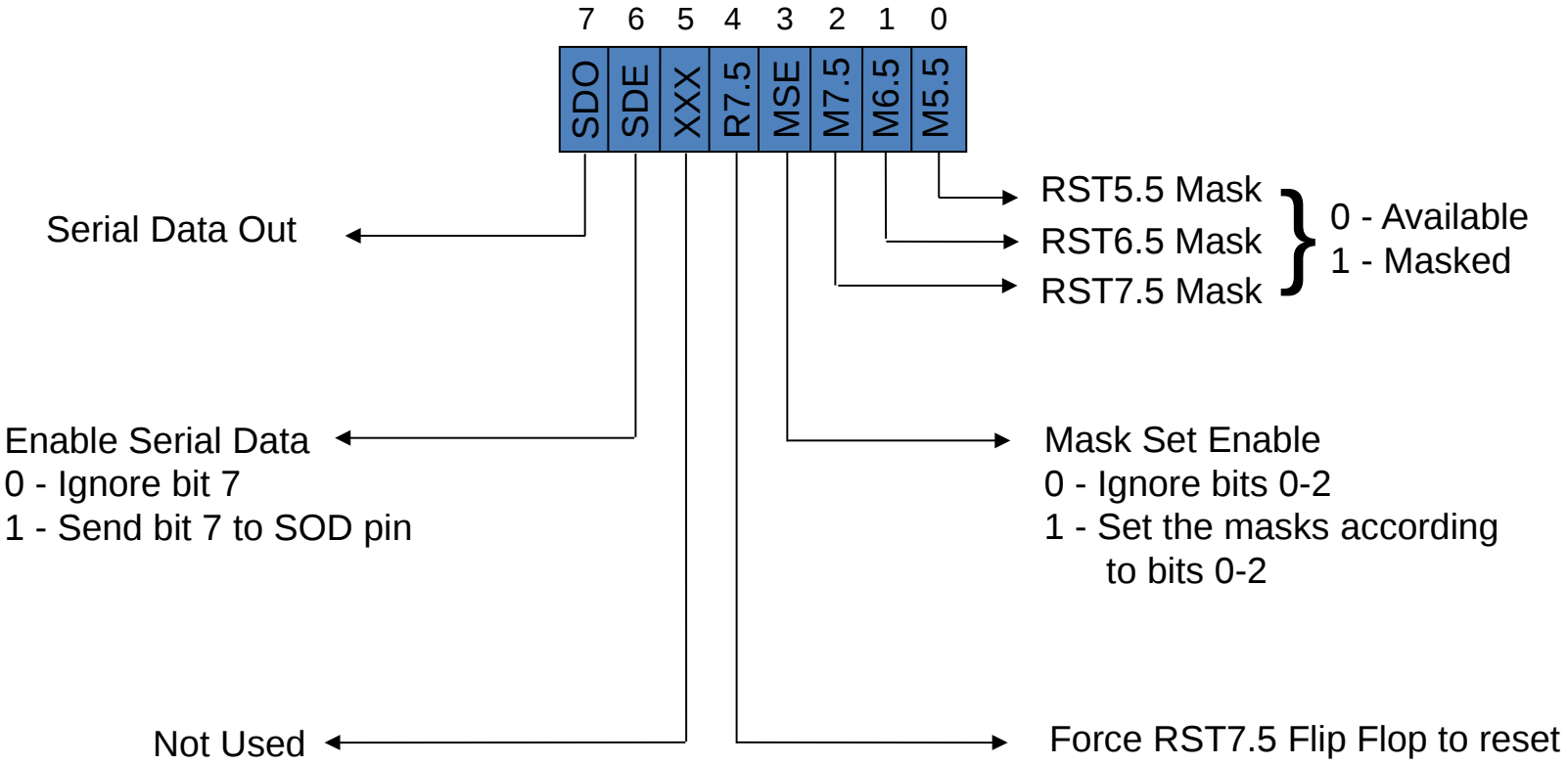
Vectored Interrupt

- The interrupt process should be enabled using the EI instruction.
- The 8085 checks for an interrupt during the execution of every instruction.
- If there is an interrupt, and if the interrupt is enabled using the interrupt mask, the microprocessor will complete the executing instruction, and reset the interrupt flip flop.
- The microprocessor then executes a call instruction that sends the execution to the appropriate location in the interrupt vector table.
- When the microprocessor executes the call instruction, it saves the address of the next instruction on the stack.
- The microprocessor jumps to the specific service routine.
- The service routine must include the instruction EI to re-enable the interrupt process.
- At the end of the service routine, the RET instruction returns the execution to where the program was interrupted.

MANIPULATING THE MASKS

- MANIPULATING THE MASKS
  - The Interrupt Enable flip flop is manipulated using the EI/DI instructions.
  - The individual masks for RST 5.5, RST 6.5 and RST 7.5 are manipulated using the SIM instruction.
  - The SIM instruction takes the bit pattern in the Accumulator and applies it to the interrupt mask enabling and disabling the specific interrupts.

How SIM Interprets the Accumulator

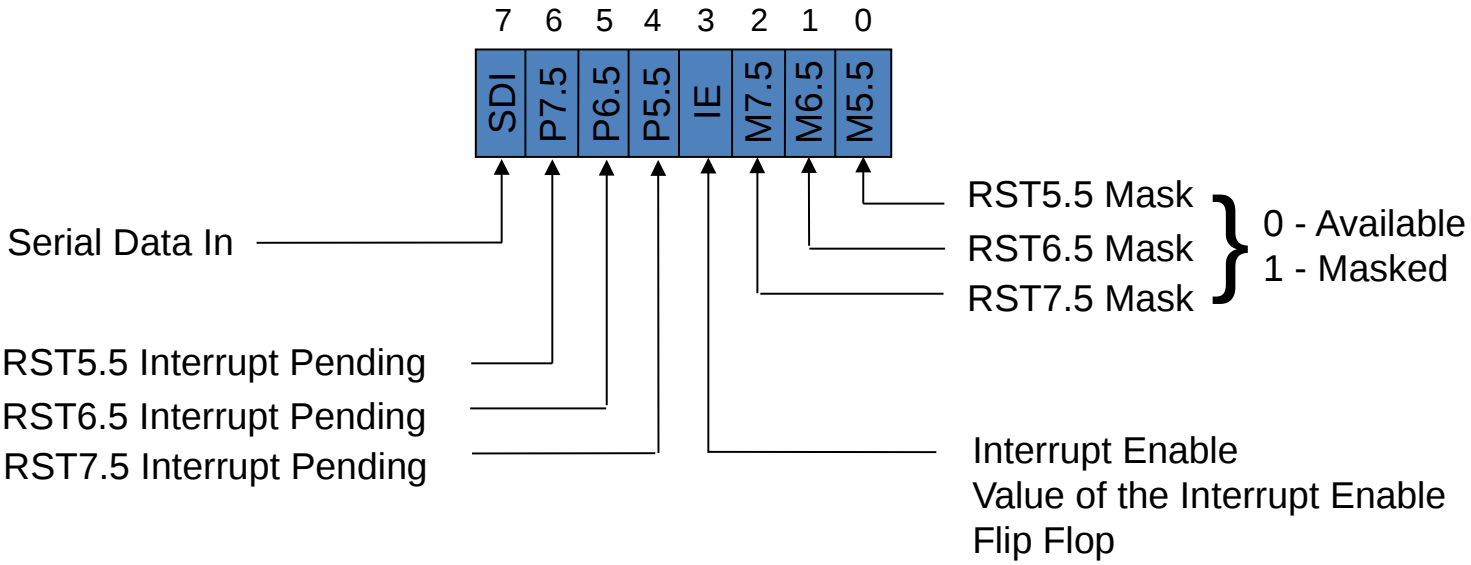


If MSE is 1

If the mask bit is 0, the interrupt is available.

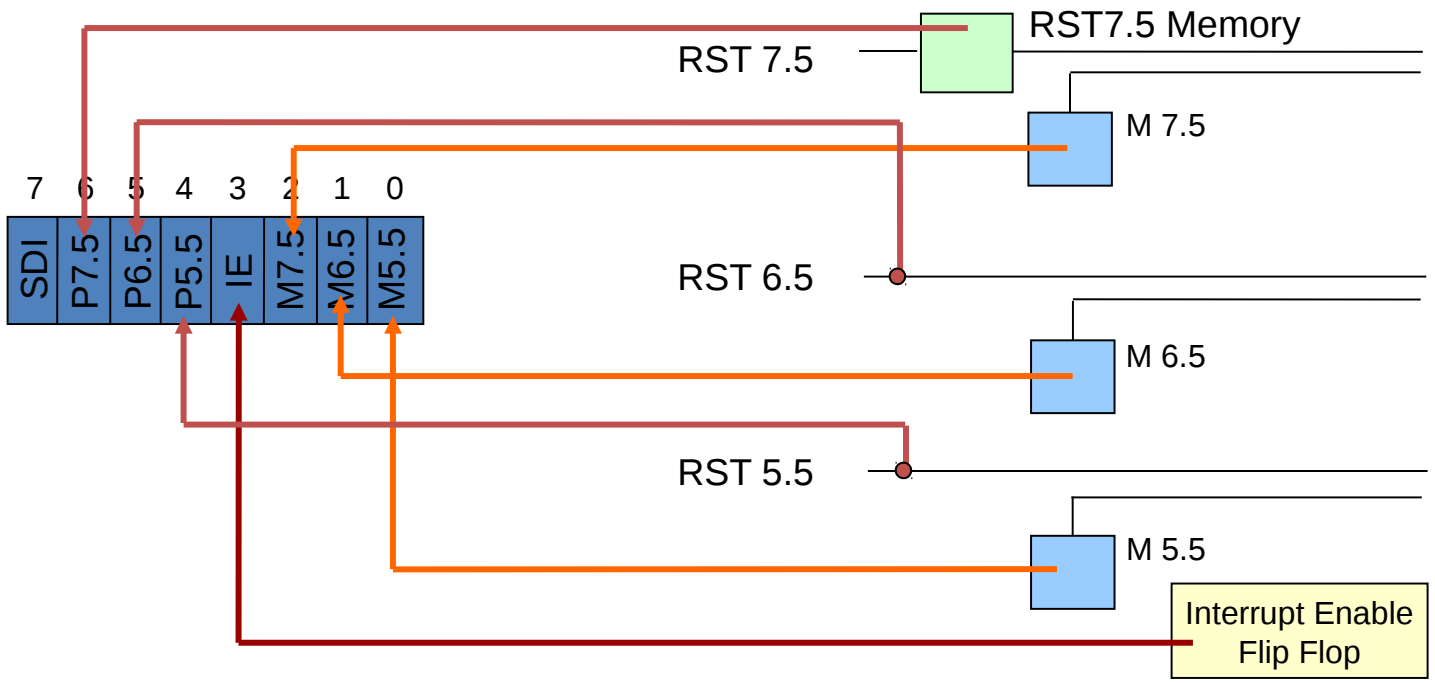
If the mask bit is 1, the interrupt is masked.

How RIM sets the Accumulator's different bits



Determining the Current Mask Settings

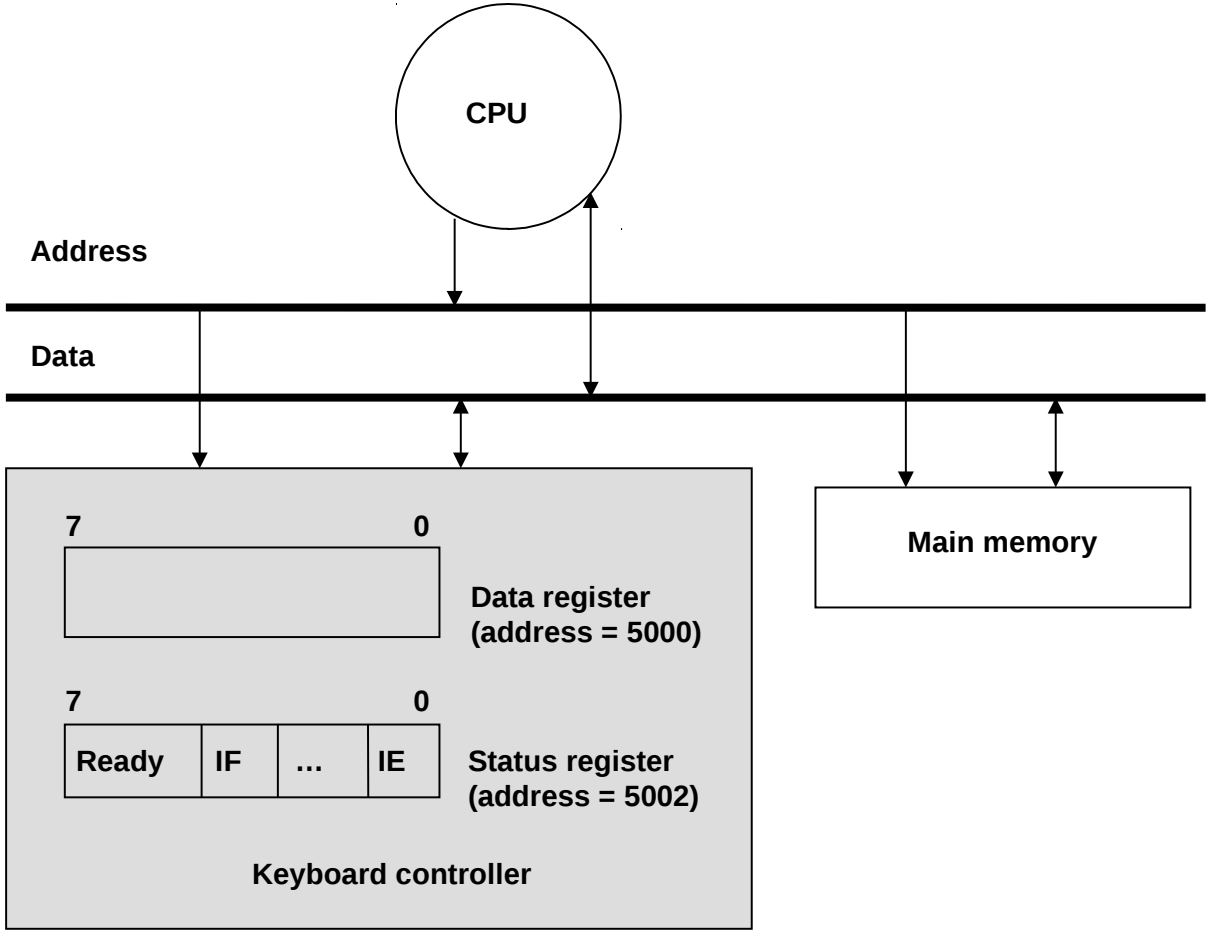
- RIM instruction: Read Interrupt Mask
  - Load the accumulator with an 8-bit pattern showing the status of each interrupt pin and mask.



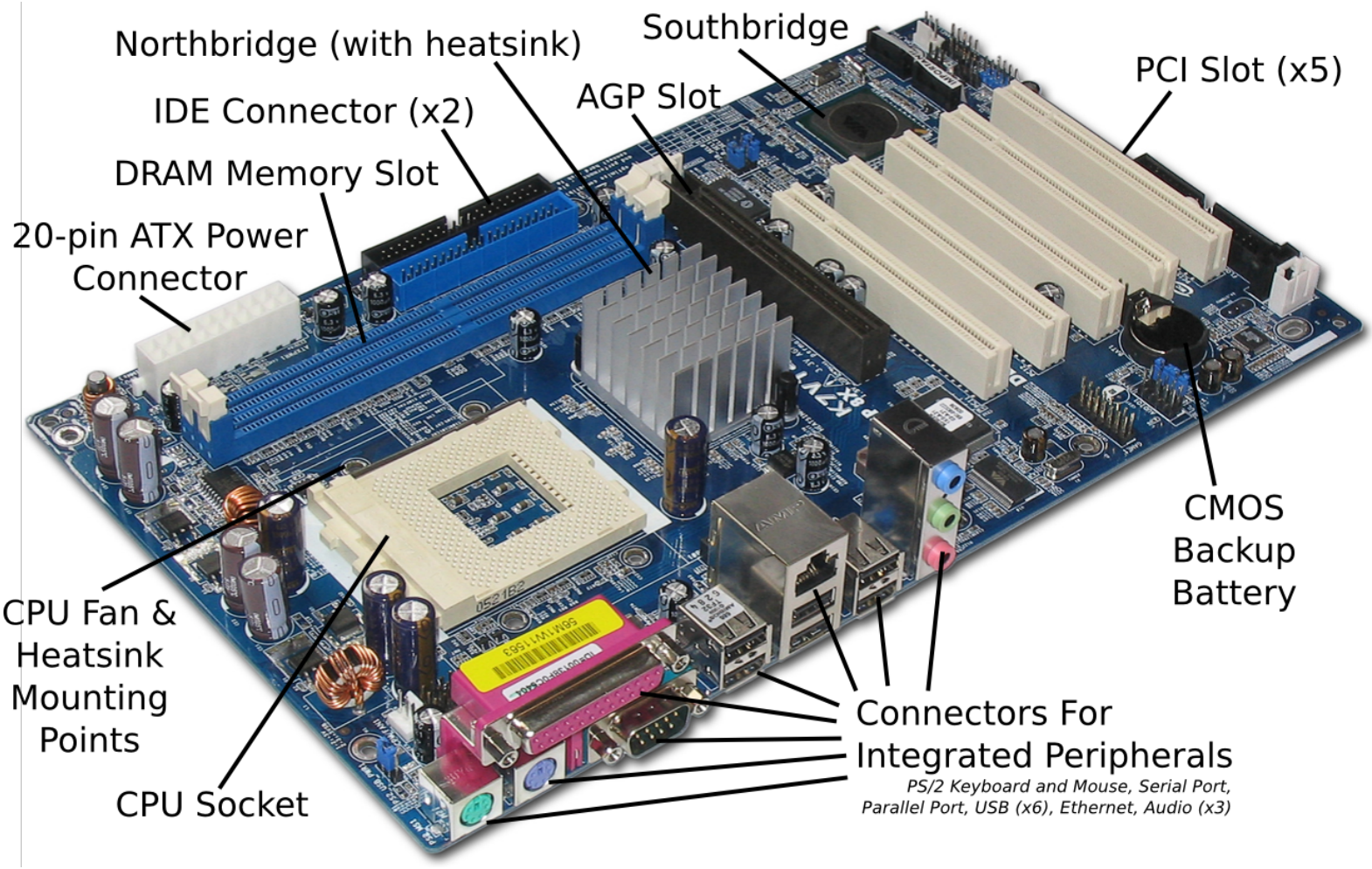
What is Memory Mapped I/O?

- Instead of having special methods for accessing the values to be read or written, just get them from memory or put them into memory.
- The device is connected directly to certain main memory locations.
- Two types of information to/from the device
  - Status
  - Value read/write

Memory Mapped I/O



Mother Board



comparison

Memory mapping	I/O mapping
16/20 bit address are provided for I/O devices	8-bit or 16-bit addresses are provided for I/O devices
The I/O ports or peripherals can be treated like memory locations and so all instructions related to memory can be used for data transmission between I/O device and processor	Only IN and OUT instructions can be used for data transfer between I/O device and processor
Data can be moved from any register to ports and vice versa	Data transfer takes place only between accumulator and ports
When memory mapping is used for I/O devices, full memory address space cannot be used for addressing memory. ⇒ Useful only for small systems where memory requirement is less	Full memory space can be used for addressing memory. ⇒ Suitable for systems which require large memory capacity
For accessing the memory mapped devices, the processor executes memory read or write cycle. ⇒ M / is asserted high	For accessing the I/O mapped devices, the processor executes I/O read or write cycle. ⇒ M / IO is asserted low
For accessing the memory mapped devices, the processor executes memory read or write cycle. ⇒ M / IO is asserted high	For accessing the I/O mapped devices, the processor executes I/O read or write cycle. ⇒ M / IO is asserted low

IO Mapped IO Device I/O Port Locations on PCs (partial)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Instruction format

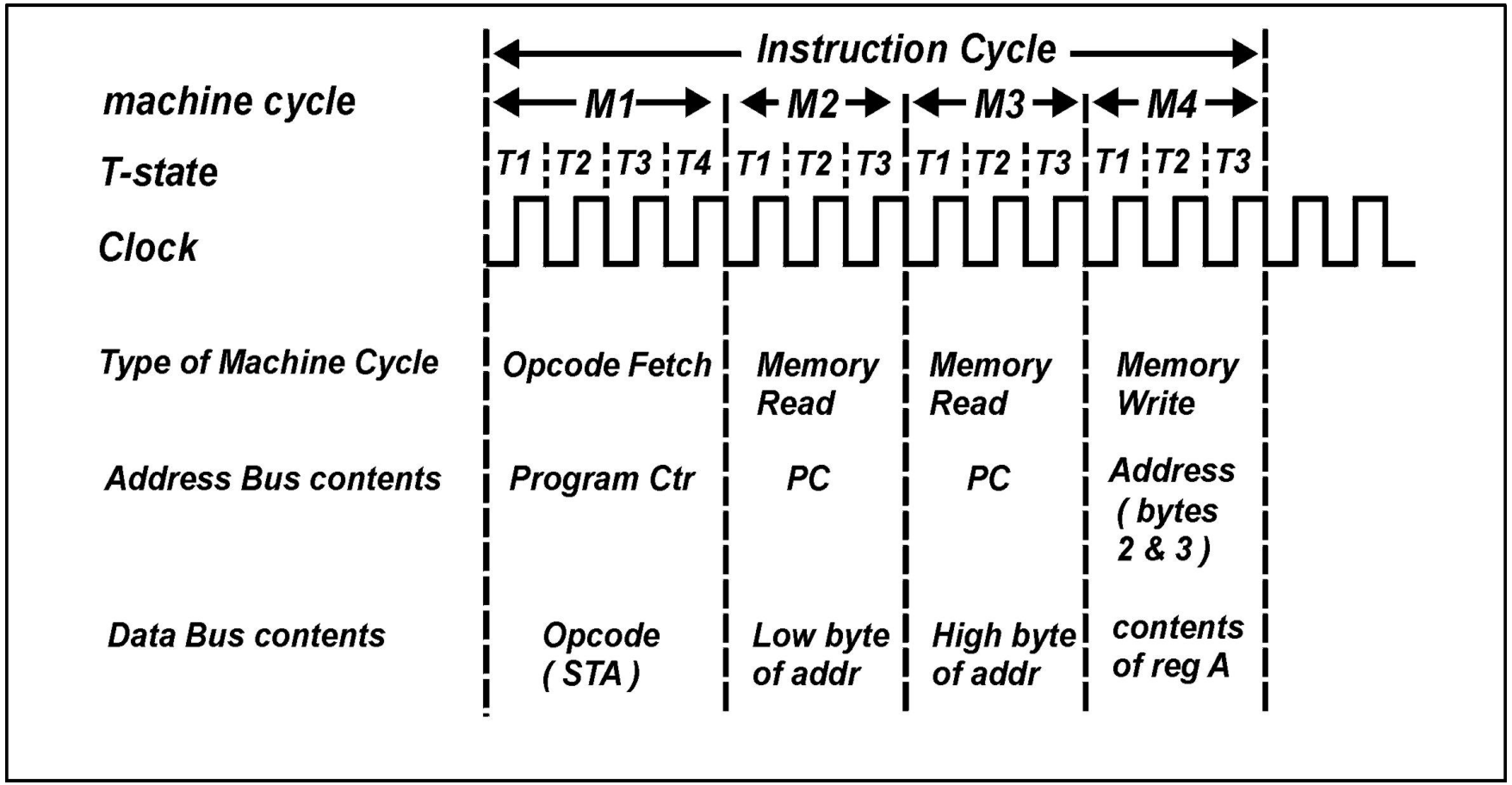
- Based on word size(8-bit processor so byte is same as word)
  - One byte instruction
  - Ex.: Mov rd,rs
    - Binary code: 01 ddd sss
    - Mov A,B: 01 111 000 = 78h
  - Two byte instruction
  - Ex.: mvi A, data
    - Binary code: 0011 1110 data
    - Mvi A, 30: 3E 30h
  - Three byte instruction
  - Ex.: Lxi rp,data_address //rp: register pair
  - Binary code: 21 lower_byte upper byte address
  - Lxi H, 2236: 21 36 22 h

MACHINE CYCLE

- The 8085 CPU can perform *seven* basic machine operations. All but the bus-idle cycle involve the transfer of data between the CPU and a peripheral device.
- The seven machine cycles are :
  - •Opcode Fetch fetch the opcode of an instruction from memory
  - •Memory Read read data stored at an addressed memory location
  - •Memory Write write data to an addressed memory location
  - •IO Read read data from an addressed input device
  - •IO Write write data to an addressed output device
  - •Interrupt Ack acknowledge an interrupt request
  - •Bus Idle no bus operation

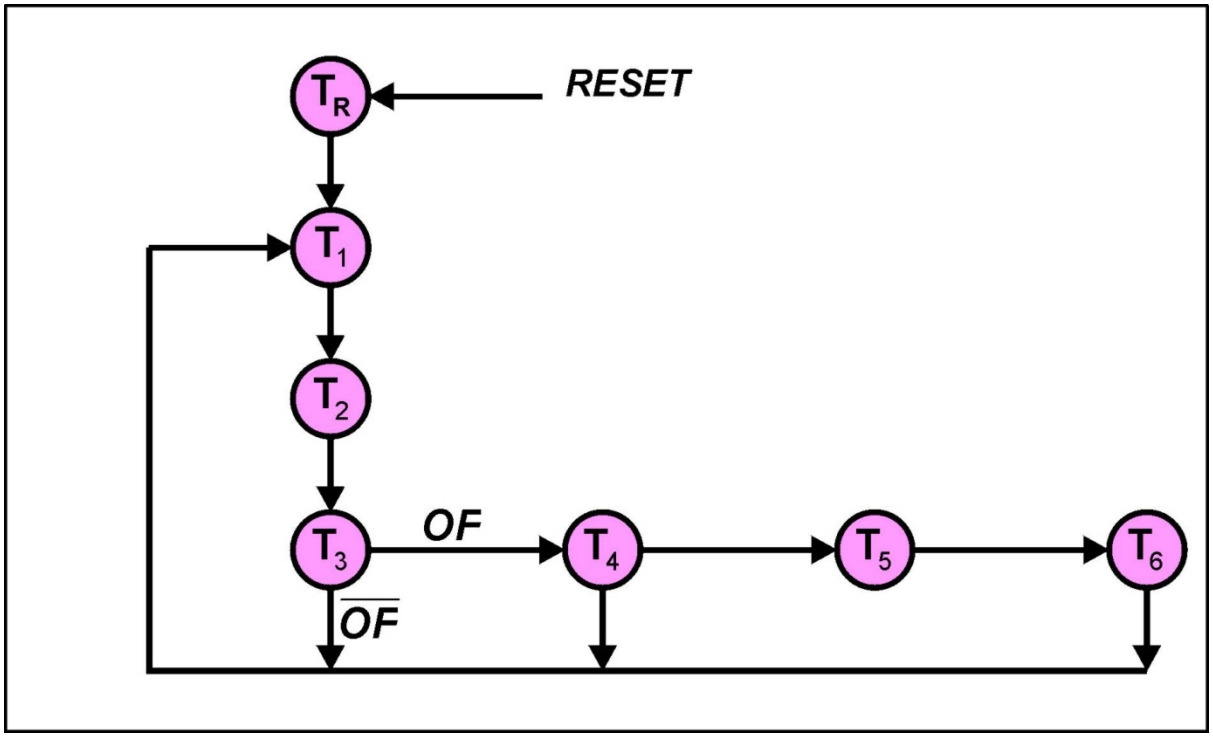
MACHINE CYCLE

- Example instruction: STA addr



T-STATE

- The operation of the 8085 can be described with respect to its state diagram (it is a synchronous state machine)



MACHINE CYCLE

- Opcode Fetch
  - The first operation in every instruction
  - Retrieves the opcode of the instruction that is being executed
  - Usually composed 4 clock cycles (T-state)
  - Some instruction required 6 T-state (CALL, INX, DCX)
  - IO/$\overline{M}$ signal is low (indicating a memory operation)

MACHINE CYCLE

- Memory Read
  - Machine cycle during which memory is read
  - Composed 3 clock cycles
  - IO/M signal is low (indicating a memory operation)
- Memory Write
  - This machine cycle is used when the 8085 needs to send data out from the accumulator or a specific register and then write it into memory
  - Composed 3 clock cycles
  - IO/M signal is low (indicating a memory operation)

MACHINE CYCLE

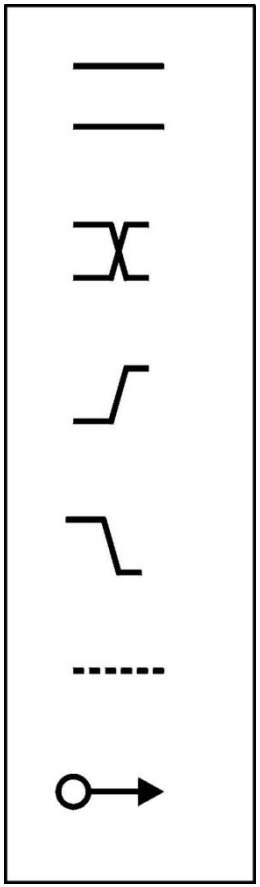
- I/O Read
  - Indicates that data is being read from an I/O device
  - Occurs when IN instruction is executed
  - Composed 3 clock cycles
  - IO/M signal is high (indicating an I/O operation)
- I/O Write
  - This machine cycle is used to write data out from accumulator in the microprocessor to the I/O device specified by the port address
  - Occurs when OUT instruction is executed
  - Composed 3 clock cycles
  - IO/M signal is high (indicating an I/O operation)

MACHINE CYCLE

- Interrupt Acknowledge
  - Special machine cycle that is used to place of the opcode fetch cycle in the RST (restart) instruction.
  - Similar to opcode fecth instruction except that it send out $\overline{\text{INTA}}$ signal instead of $\overline{\text{RD}}$ signal and IO/$\overline{\text{M}}$ signal is High
  - Composed 6 clock cycle

T-STATE

- Key to timing diagram



Indicates a bus. Whilst the lines remain parallel the individual bits of the bus remain unchanged.

Indicates a bus. Where the lines cross indicates a possible change in logic level of one or more bits of the bus.

Indicates a 0 -> 1 transition of a digital signal

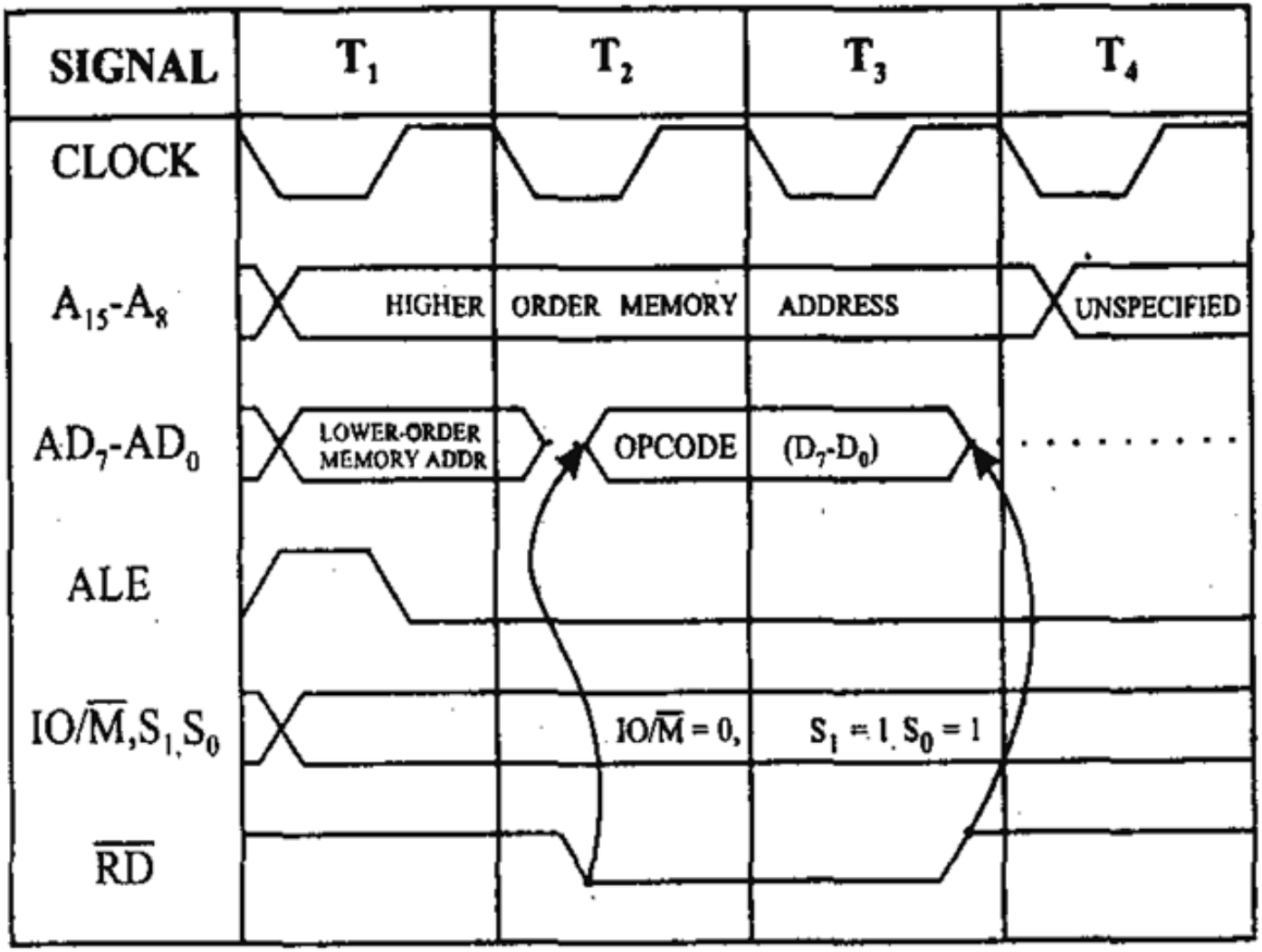
Indicates a 1 -> 0 transition of a digital signal

Indicates a bus or a bit being in the Hi-Z state (tri-state)

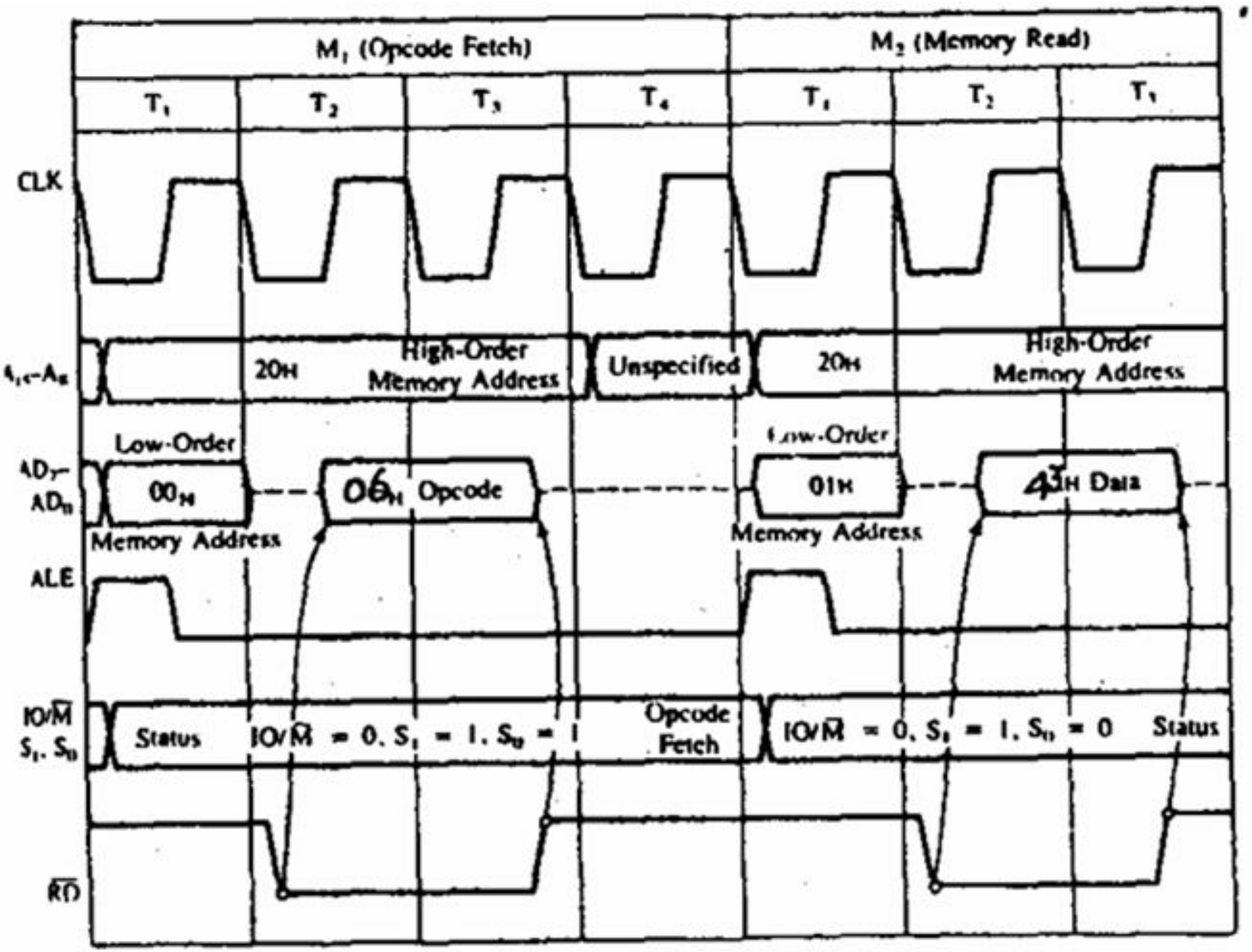
The tail of the arrow indicates the cause of a signal change.

The head of the arrow indicates the affected signal.

Opcode Fetch

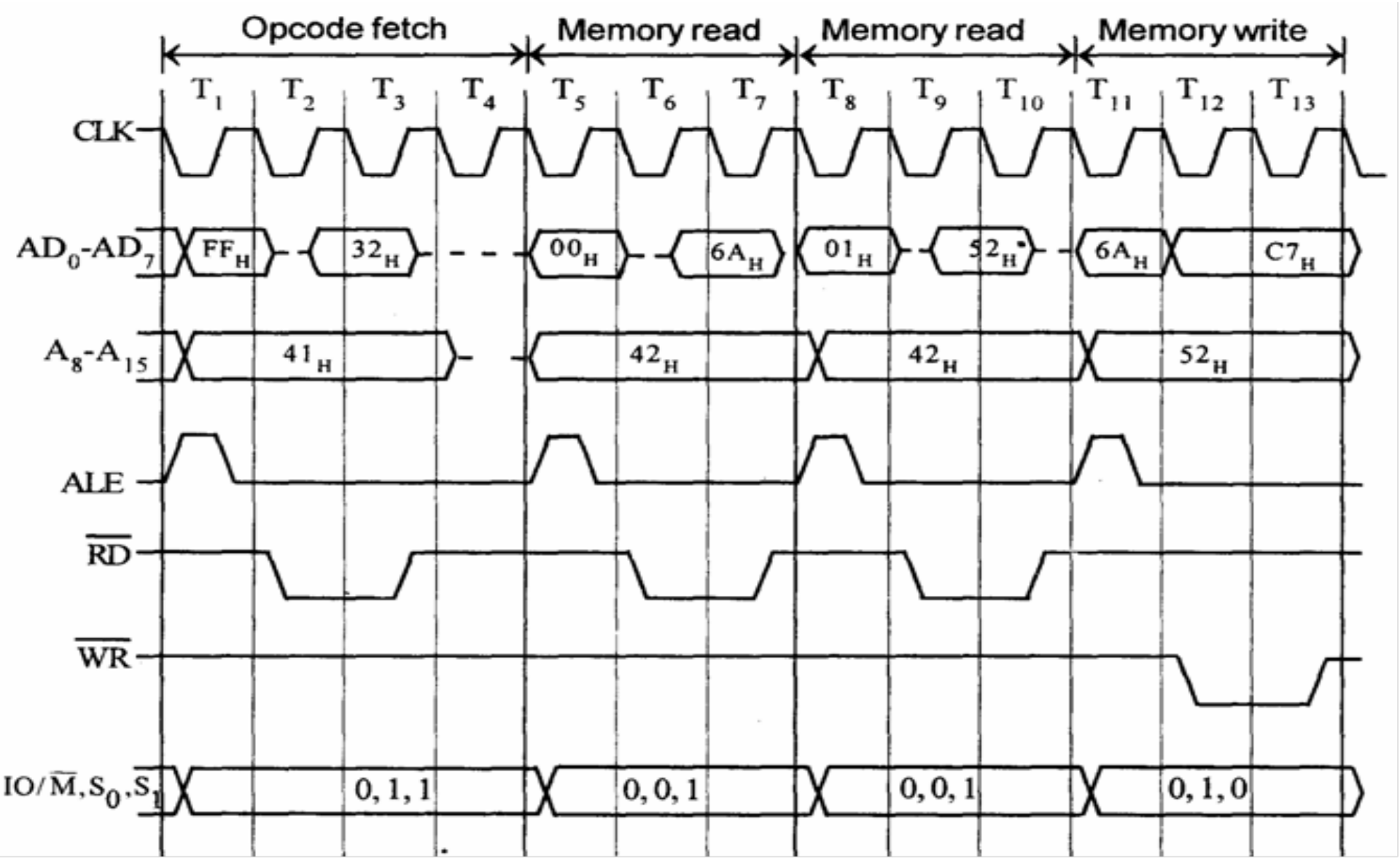


MVI

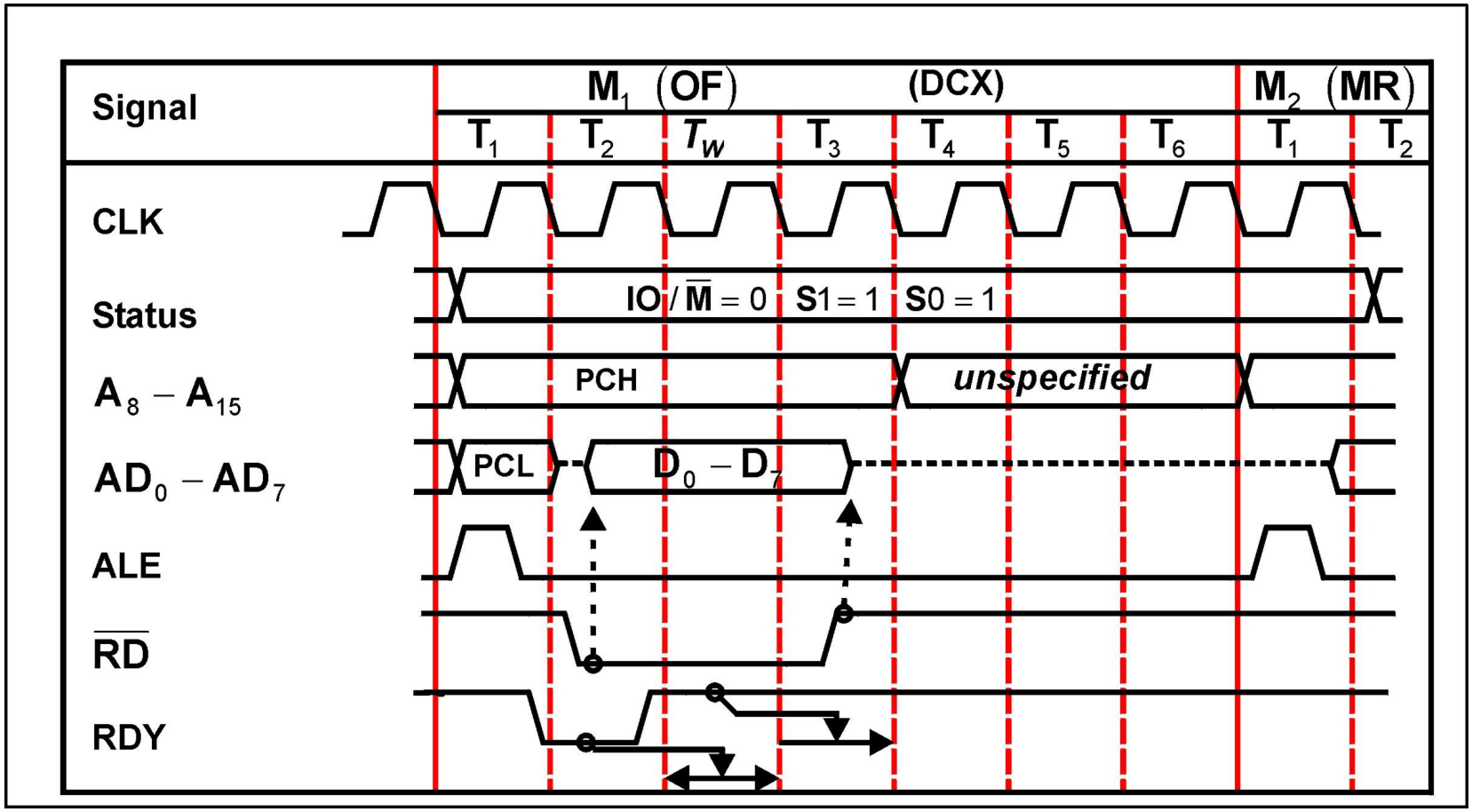

M1 (Opcode Fetch)
M2 (Memory Read)
T1
T2
T3
T4
T1
T2
T3
CLK
A15–A8
20H
High-Order Memory Address
Unspecified
20H
High-Order Memory Address
AD7–AD0
Low-Order
00H
Memory Address
06H Opcode
Low-Order
01H
Memory Address
43H Data
ALE
IO/M
S1, S0
Status
IO/M = 0, S1 = 1, S0 = 1
Opcode Fetch
IO/M = 0, S1 = 1, S0 = 0
Status
RD

STA 526A

(32, 6A, 52 (ACC-C7)



Wait-State


Signal
M1 (OF)
(DCX)
M2 (MR)
T1
T2
Tw
T3
T4
T5
T6
T1
T2
CLK
Status
IO / M = 0
S1 = 1
S0 = 1
A8 – A15
PCH
unspecified
AD0 – AD7
PCL
D0 – D7
ALE
RD
RDY

Instruction Set & Addressing mode

- Instruction Set:
  - 8085 instruction set consists of the following instructions:
  - Data moving instructions.
  - Arithmetic - add, subtract, increment and decrement.
  - Logic - AND, OR, XOR and rotate.
  - Control transfer - conditional, unconditional, call subroutine, return from subroutine and restarts.
  - Input/Output instructions.
  - Other - setting/clearing flag bits, enabling/disabling interrupts, stack operations, etc.

- **Addressing mode**
  - **Register - references the data in a register or in a register pair.**
  - **Register indirect - instruction specifies register pair containing address, where the data is located.**
  - **Direct,**
  - **Immediate - 8 or 16-bit data**

8085 Microprocessor Memory

- ***Memory:***

  Program, data and stack memories occupy the same memory space. The total addressable memory size is 64KB.

- **Program memory -** program can be located anywhere in memory.
  - Jump, branch and call instructions use 16-bit addresses, i.e. they can be used to jump/branch anywhere within 64 KB.
  - All jump/ branch instructions use absolute addressing.

- **Data memory -** the processor always uses 16-bit addresses so that data can be placed anywhere.
- **Stack memory** is limited only by the size of memory**.** Stack grows downward.

- First 64 bytes in a zero memory page should be reserved for vectors used by RST instructions

8085 MP Instruction Set Architecture

- Contains several registers include B,C,D,E,H,L and an 8-bit accumulator register, A.
- The registers B,C,D,E,H,L can be accessed as pairs. Pairs are not arbitrary. B and C, D and E, H and L.
- SP is a 16 bit stack pointer register pointing to the top of the stack.
- PC is a 16-bit Program counter
- Contains five flags known as flag registers:

Instruction Set of 8085

- Arithmetic Operations
  - add, sub, inr/dcr
- Logical operation
  - and, or, xor, rotate, compare, complement
- Branch operation
  - Jump, call, return
- Data transfer/Copy/Memory operation/IO
  - MOV, MVI, LD, ST, OUT

Copy/Mem/IO operation

- MVI R, 8 bit *// load immediate data*
- MOV R1, R2 *// Example MOV B, A*
- MOV R M *// Copy to R from 0(HL Reg) Mem*
- MOV M R *// Copy from R to 0(HL Reg) Mem*

- LDA 16 bit *// load A from 0(16bit)*
- STA 16 bit *// Store A to 0(16bit)*
- LDAX Rp *// load A from 0(Rp), Rp=RegPair*
- STAX Rp *// Store A to 0(Rp)*
- LXI Rp 16bit *// load immediate to Rp*

- IN 8bit *// Accept data to A from port 0(8bit)*
- OUT 8 bit *// Send data of A to port 0(8bit)*

Arithmetic Operation

- ADD R *// Add A = A + B.reg*
- ADI 8bit *// Add A= A + 8bit*
- ADD M *// Add A=A + 0(HL)*

- SUB R *// Sub A = A -B.reg*
- SUI 8bit *// Sub A= A - 8bit*
- SUB M *// Sub A=A - 0(HL)*

- INR R *// R = R+1*
- INR M *// 0(HL)=0(HL)+1*
- DCR R *// R = R-1*
- DCR M *// 0(HL)=0(HL)-1*
- INX Rp *// Rp=Rp+1*
- DCX Rp *// Rp=Rp-1*

Other Operations

- Logic operations
  - ANA R ANI 8bit ANA M
  - ORA, ORI, XRA, XRI
  - CMP R // compare with R with ACC
  - CPI 8bit // compare 8 bit with ACC
- Branch operations
  - JMP 16bit, CALL 16 bit
  - JZ 16bit, JNZ 16bit, JC 16bit, JNC 16 bit
  - RET
- Machine Control operations
  - HLT, NOP, POP, PUSH

8085 Microprocessor Instruction Set

Contains a total of 74 different instructions.

R, R1, R2	8 bit registers representing A, B, C, D, E, H or L
M	Indicates memory location
RP	Indicates register pair such as BC, DE, HL, SP
Γ	16 bit address representing address or data value.
n	8-bit address or data value stored in memory immediately after the opcode
Cond	Condition for conditional instructions. NZ (Z = 0), Z (Z = 1),P (S = 0), N (S = 1), PO (P = 0), PE (P = 1), NC (CY = 0), C (CY=1)

Data movement instruction for the 8085 microprocessor

Instruction	Operation
NOP	No operation
MOV r1, r2	r1 = r2
MOV r, M	r1 = M[HL]
MOV M, r	M[HL] = r
MVI r, n	r = n
MVI M, n	M[HL] = n
LXI rp, Γ	rp = Γ
LDA Γ	A = M[Γ]
STA Γ	M[Γ] = A
LHLD Γ	HL = M[Γ], M[Γ + 1]
SHLD Γ	M[Γ], M[Γ + 1] = HL
LDAX rp	A = M[rp] (rp = BC, DE)
STAX rp	M[rp] = A (rp = BC, DE)
XCHG	DE ↔ HL
PUSH rp	Stack = rp (rp ≠ SP)
PUSH PSW	Stack = A, flag register
POP rp	rp = Stack (rp ≠ SP)
POP PSW	A, flag register = Stack
XTHL	HL ↔ Stack
SPHL	SP = HL
IN n	A = input port n
OUT n	Output port n = A

Data operation instruction for the 8085 microprocessor

Instruction	Operation	Flags
ADD r	A = A + r	All
ADD M	A = A + M[HL]	All
ADI n	A = A + n	All
ADC r	A = A + r + CY	All
ADC M	A = A + M[HL] + CY	All
ACI n	A = A + n + CY	All
SUB r	A = A - r	All
SUB M	A = A - M[HL]	All
SUI n	A = A - n	All
SBB r	A = A - r - CY	All
SBB M	A = A - M[HL] - CY	All
SBI n	A = A - n - CY	All
INR r	r = r + 1	Not CY
INR M	M[HL] = M[HL] + 1	Not CY
DCR r	r = r - 1	Not CY
DCR M	M[HL] = M[HL] - 1	Not CY
INX rp	rp = rp + 1	None
DCX rp	rp = rp - 1	None
DAD rp	HL = HL + rp	CY
DAA	Decimal adjust	All
ANA r	A = A $\wedge$ r	All
ANA M	A = A $\wedge$ M[HL]	All

Data operation instruction for the 8085 microprocessor

Instruction	Operation	Flags
ANI n	A = A ∧ n	All
ORA r	A = A ∨ r	All
ORA M	A = A ∨ M[HL]	All
ORI n	A = A ∨ n	All
XRA r	A = A ⊕ r	All
XRA M	A = A ⊕ M[HL]	All
XRI n	A = A ⊕ n	All
CMP r	Compare A and r	All
CMP M	Compare A and M[HL]	All
CPI n	Compare A and n	All
RLC	CY = A7, A = A(6-0), A7	CY
RRC	CY = A0, A = A0, A(7-1)	CY
RAL	CY, A = A, CY	CY
RAR	A, CY = CY, A	CY
CMA	A = A’	None
CMC	CY = CY’	CY
STC	CY = 1	CY

Program control instruction

Instruction	Operation
JUMP Γ	GOTO Γ
J cond Γ	If condition is true then GOTO Γ
PCHL	GOTO address HL
CALL Γ	Call subroutine at Γ
C cond Γ	If condition is true then call subroutine at Γ
RET	Return from subroutine
R cond	If condition is true then return from subroutine
RST n	Call subroutine at 8*n (n = 5.5, 6.5, 7.5)
RIM	A = IM
SIM	IM = A
DI	Disable interrupts
EI	Enable interrupts
HLT	Halt the CPU

A Simple 8085 Program

1: i = n, sum = 0

2: sum = sum + i, i = i - 1

3: IF i ≠ 0 then GOTO 2

4: total = sum

A Simple 8085 Program (contd)

	LDA n MOV B, A	i = n
	XRA A	*sum* = A ⊕ A = 0
Loop:	ADD B	*sum* = *sum* + i
	DCR B	*i* = *i* - 1
	JNZ *Loop*	IF i ≠ 0 THEN GOTO *Loop*
	STA *total*	*total = sum*

Execution trace

Instruction	1st Loop	2nd Loop	3rd Loop	4th Loop	5th Loop
LDA n MOV B, A	B = 5				
XRA A	A = 0				
ADD B	A = 5	A = 9	A = 12	A = 14	A = 15
DCR B	B = 4, Z = 0	B = 3, Z = 0	B = 2, Z = 0	B = 1, Z = 0	B = 0, Z = 1
JNZ *Loop*	JUMP	JUMP	JUMP	JUMP	NO JUMP
STA *total*					*total* = 15

Simple Assembly Program

```
MVI   A, 24H      // load Reg  ACC with 24H
MVI   B , 56H     // load Reg B with 56H
ADD   B           // ACC= ACC+B
OUT 01H           // Display ACC contents on port 01H
HALT              // End the program
```

Result: 7A (All are in Hex)

Code to multiply two number

```
    LDA   2000     // Load multiplicant to accumulator
    MOV  B,A       // Move multiplicant from A(acc) to B register
    LDA   2001     // Load multiplier to accumulator
    MOV  C,A       // Move multiplier from A to C
    MVI   A,00     // Load immediate value 00 to a
L:  ADD    B       // Add B(multiplier) with A
    DCR    C       // Decrement C, it act as a counter
    JNZ    L       // Jump to L if C reaches 0
    STA    2010    // Store result in to memory
    HLT            // End
```

TABLE 1–2 Many modern Intel and Motorola microprocessors.

Manufacturer	*Part*	*Data Bus Width*	*Memory Size*
Intel	8048	8	2K internal
	8051	8	8K internal
	8085A	8	64K
	8086	16	1M
	8088	8	1M
	8096	16	8K internal
	80186	16	1M
	80188	8	1M
	80251	8	16K internal
	80286	16	16M
	80386EX	16	64M
	80386DX	32	4G
	80386SL	16	32M
	80386SLC	16	32M + 1K cache
	80386SX	16	16M
	80486DX/DX2	32	4G + 8K cache
	80486SX	32	4G + 8K cache
	80486DX4	32	4G + 16K cache
	Pentium	64	4G + 16K cache
	Pentium Overdrive (P24T) (replaces 80486)	32	4G + 16K cache
	Pentium Pro processor	64	64G + 16K L1 cache + 256K L2 cache
	Pentium II	64	64G + 32K L1 cache + 512K L2 cache
	Pentium II Xeon	64	64G + 32K L1 cache + 512K or 1M L2 cache
	Pentium III, Pentium 4	64	64G + 32K L1 cache + 256K L2 cache
Motorola	6800	8	64K
	6805	8	2K
	6809	8	64K
	68000	16	16M
	68008Q	8	1M
	68008D	8	4M
	68010	16	16M
	68020	32	4G
	68030	32	4G + 256 cache
	68040	32	4G + 8K cache
	68050	32	Proposed, but never released
	68060	64	4G + 16K cache
	PowerPC	64	4G + 32K cache

Table 1.1 The Evolution of Intel Microprocessors[1]

Microprocessor	Year Introduced	Number of Transistors	Minimum Feature Size (microns)	External Data Bus Width	Internal Register Widths	Address Bus Width/ Memory Space	Estimated Processing Rate (MIPs)[2]	Onboard Coprocessor	Internal Cache Memory	V_{CC} (volts)	P_D (watts)
4004	1971	2,250	10.0	4	4	10/1K	.06 (.108MHz)	no	no		
8080	1974	6,000	6.0	8	8	16/64K	.2 (2 MHz)	no	no	±5, 12	1.2
8086	1978	29,000	3.0	16	16	20/1 MB	.47 (4.77 MHz)	no	no	5	1.7
8088	1979	29,000	3.0	8	16	20/1 MB	.33 (4.77 MHz)	no	no	5	1.7
80286	1982	134,000	1.5	16	16	24/16 MB	2 (8 MHz)	no	no	5	3
80386DX	1985	275,000	1.5	32	32	32/4 GB	5.5 (16 MHz)	no	no	5	1.95
80386SX	1988	275,000	1.5	16	32	24/16 MB	3.9 (16 MHz)	no	no	5	1.9
80486DX	1989	1.2 million	0.8	32	32	32/4 GB	20 (25 MHz)	yes	8K	5	5
80486SX	1991	1.2 million	0.8	32	32	32/4 GB	13 (16 MHz)	no	8K	5	3.4
80486DX2	1992	1.2 million	0.6	32	32	32/4 GB	41 (50 MHz)	yes	8K	5	4.8
80486DX4	1994	1.2 million	0.6	32	32	32/4 GB	60 (75 MHz)	yes	16K	3.3	
Pentium P60	1993	3.1 million	0.8	64	32	32/4 GB	100 (60 MHz)	yes	16K	5	14.6
Pentium P100	1994	3.1 million	0.6	64	32	32/4 GB	150 (100 MHz)	yes	16K	3.3	10.1
Pentium P120	1995	3.1 million	0.35	64	32	32/4 GB	185 (120 MHz)	yes	16K	3.3	12.8
Pentium Pro 150	1995	5.5 million[3]	0.6	64	32	36/64 GB	350 (150 MHz)	yes	16K/256K[4]	3.3	29.2
Pentium Pro 200	1996	5.5 million	0.35	64	32	36/64 GB	475 (200 MHz)	yes	16k/512K	3.3	35
P7[5]	1997–8	12 million	0.2	—	—	—	750	yes	—	—	—

[1]Specifications shown are for initial introduction of part.

[2]Millions of instructions per second (internal clock rate shown in parentheses).

[3]256K level two cache (separate die in same package) has 15.5 million transistors.

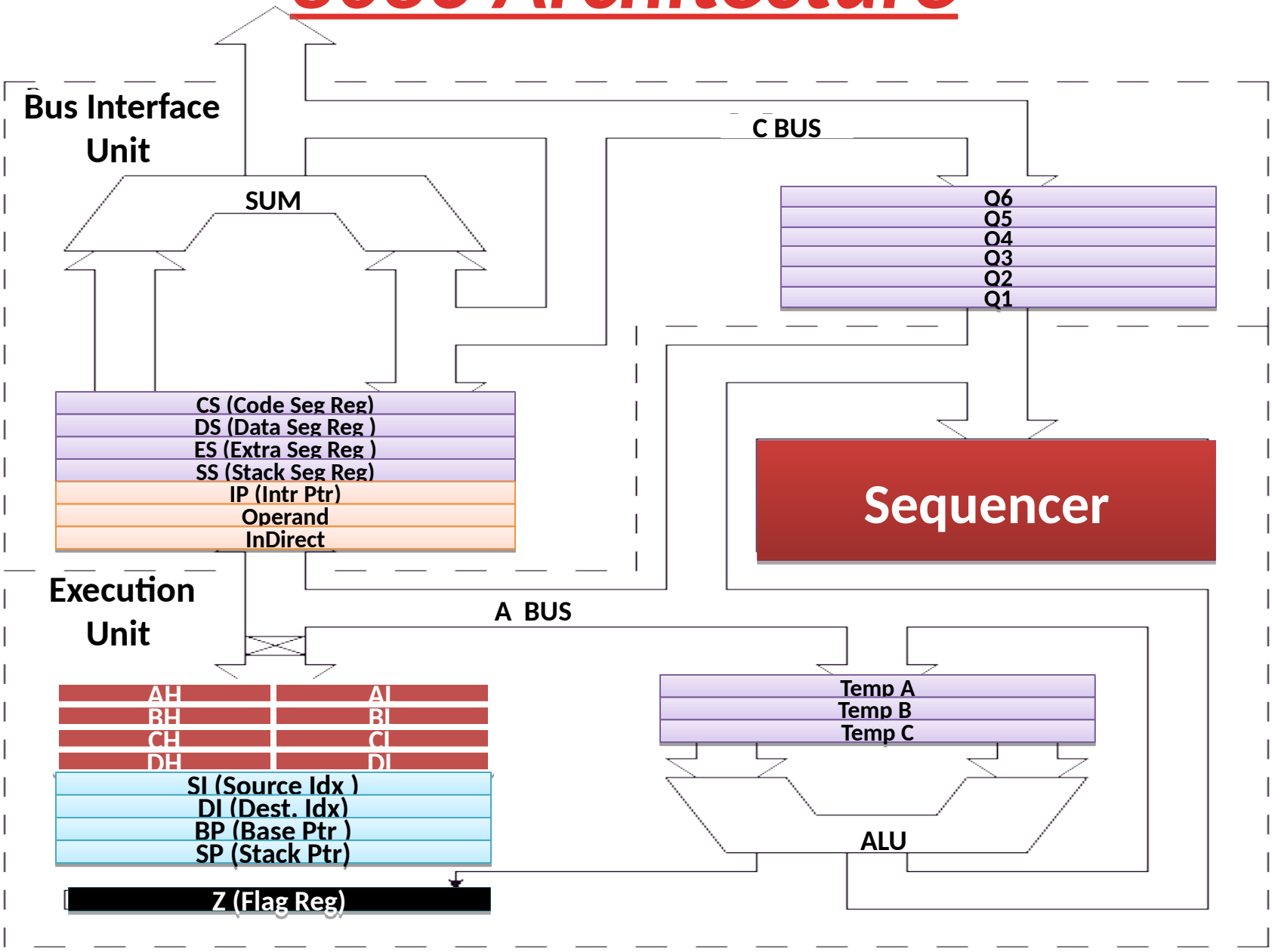
[4]16K data/code level one cache plus 256K level two cache.

[5]Best guess.

Next...

- 8086
  - Block diagram (Data Path), Registers
- Memory Model
  - Stack, Data and Code Segment
- Instruction Set of x86
- Addressing mode
- Procedure and subroutine
- Examples programs in C/C++ assembly
- Peripheral device and Assembly program

8086 Architecture



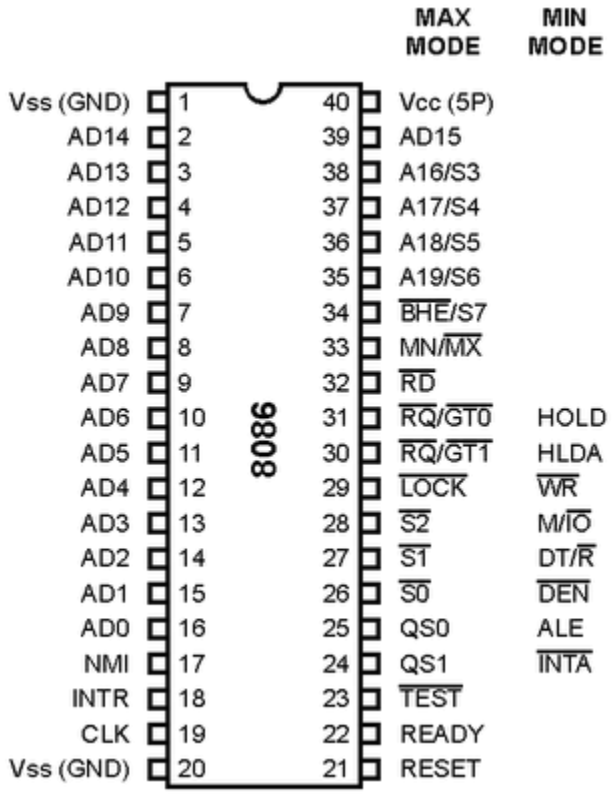
Minimum/Max Mode Configuration For 8086

The microprocessor 8086 is operated in minimum mode by strapping its MN/MX pin to logic 1.

In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system.

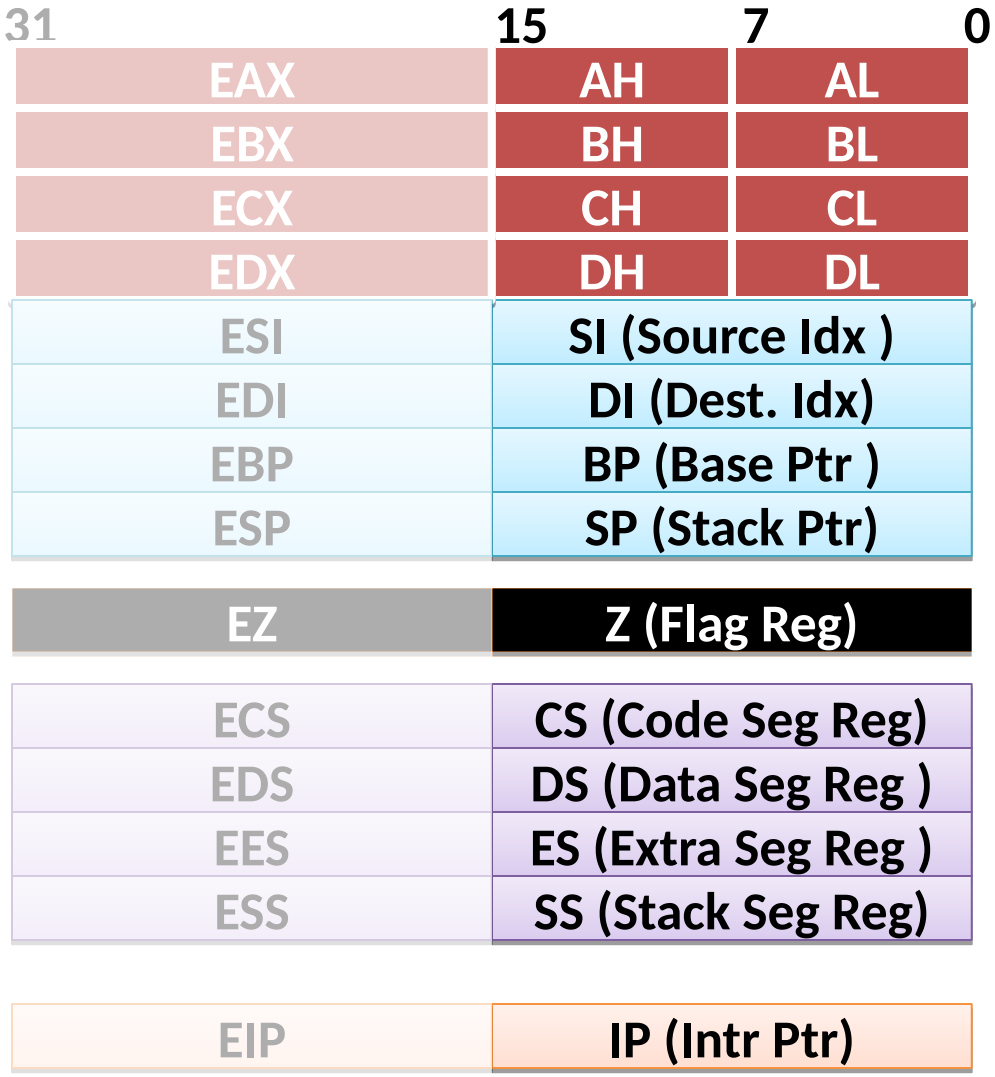
In the maximum mode, the 8086 is operated by strapping the MN/MX pin to ground.

In the maximum mode, there may be more than one microprocessor in the system configuration. The components in the system are same as in the minimum mode system.



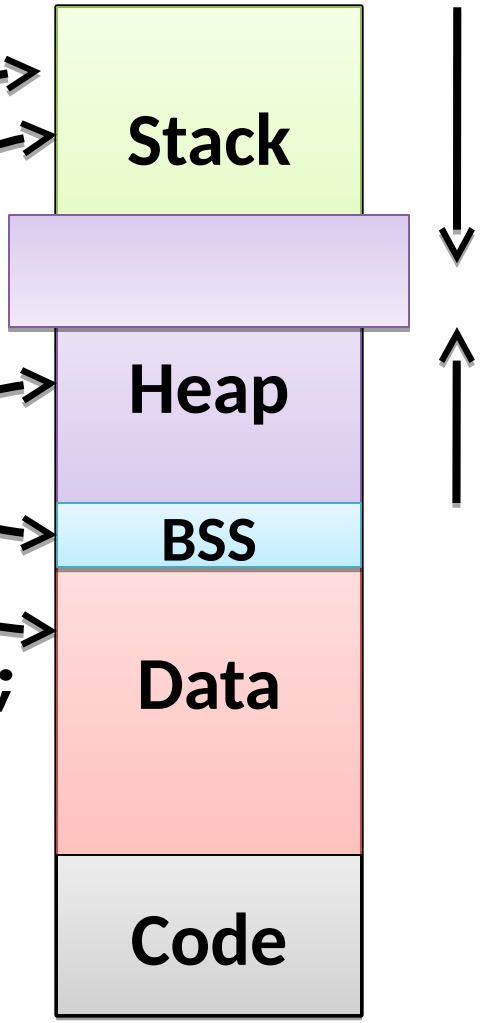
8086 & x86 Registers

- **AX** - accumulator reg
- **BX** - base address reg
- **CX** - count reg
- **DX** - data reg
- **SI** - source index reg
- **DI** - dest index reg
- **BP** - base pointer.
- **SP** - stack pointer.



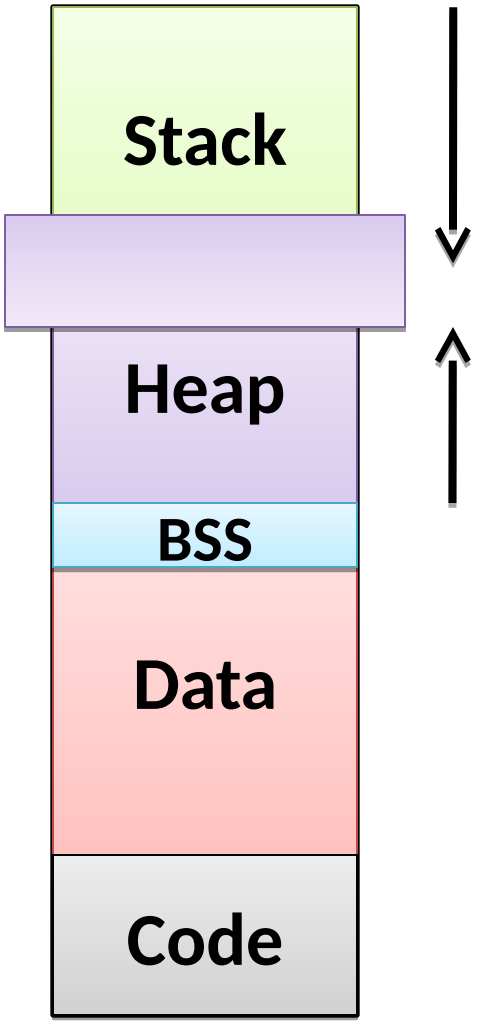
Memory layout of C program

```
int A;
int B=10;
main(){
   int Alocal;
   int *p;
   p=(int*)malloc(10);
}
```

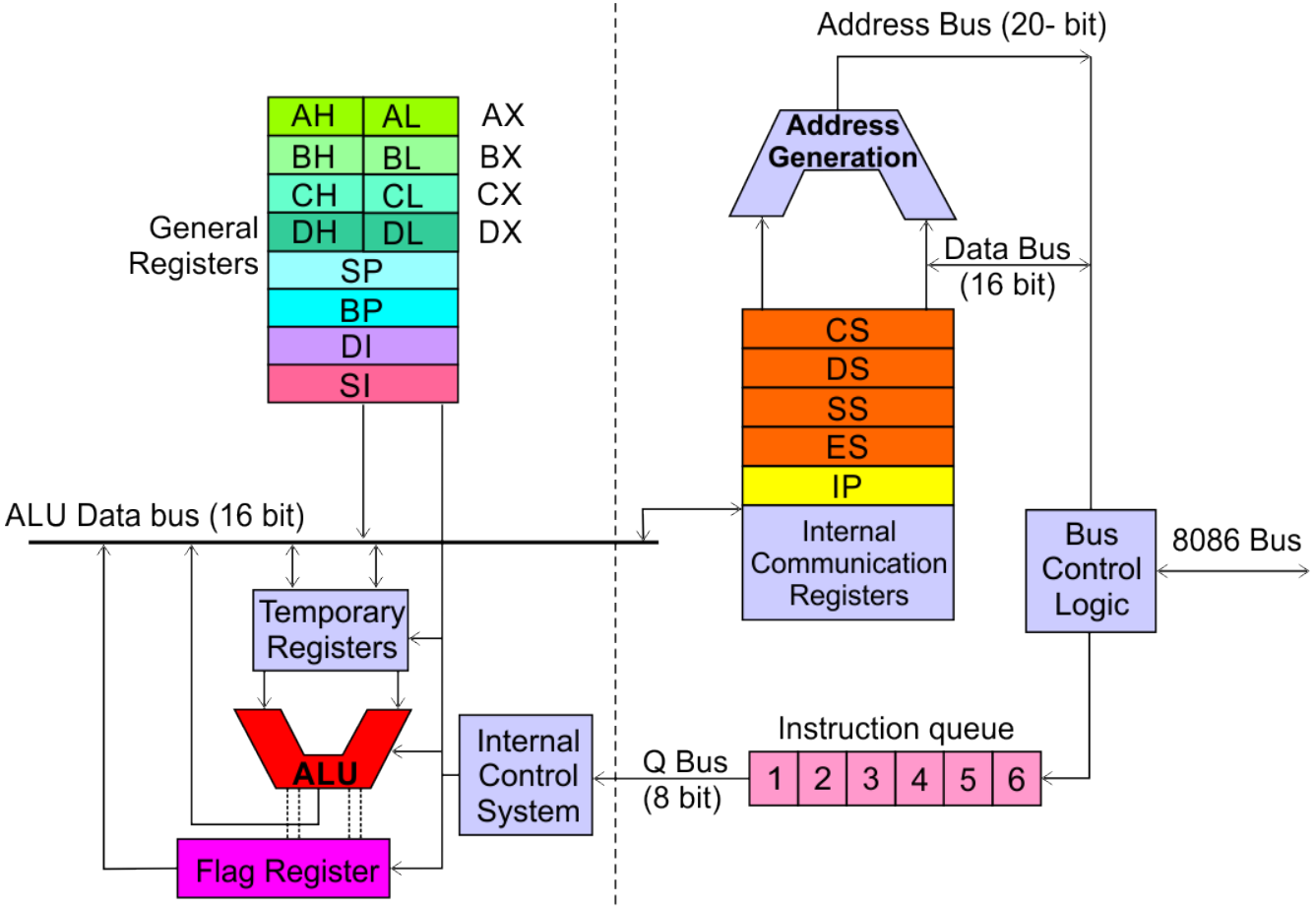


Memory layout of C program

- Stack
  - automatic (default), local
  - Initialized/uninitialized
- Data
  - Global, static, extern
  - BSS: Block Started by Symbol
- Code
  - program instructions
- Heap
  - malloc, calloc



Architecture



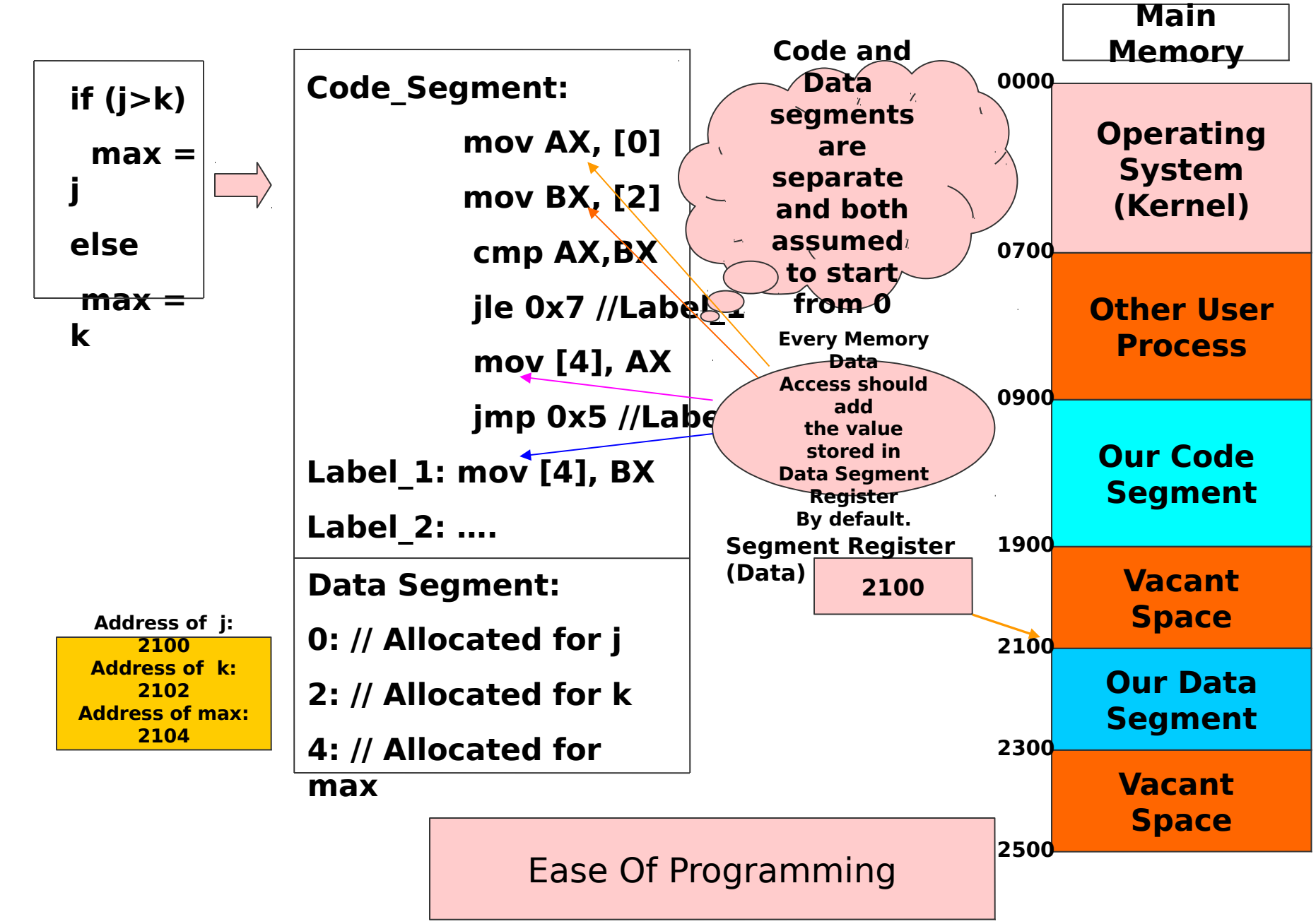
Execution Unit (EU)

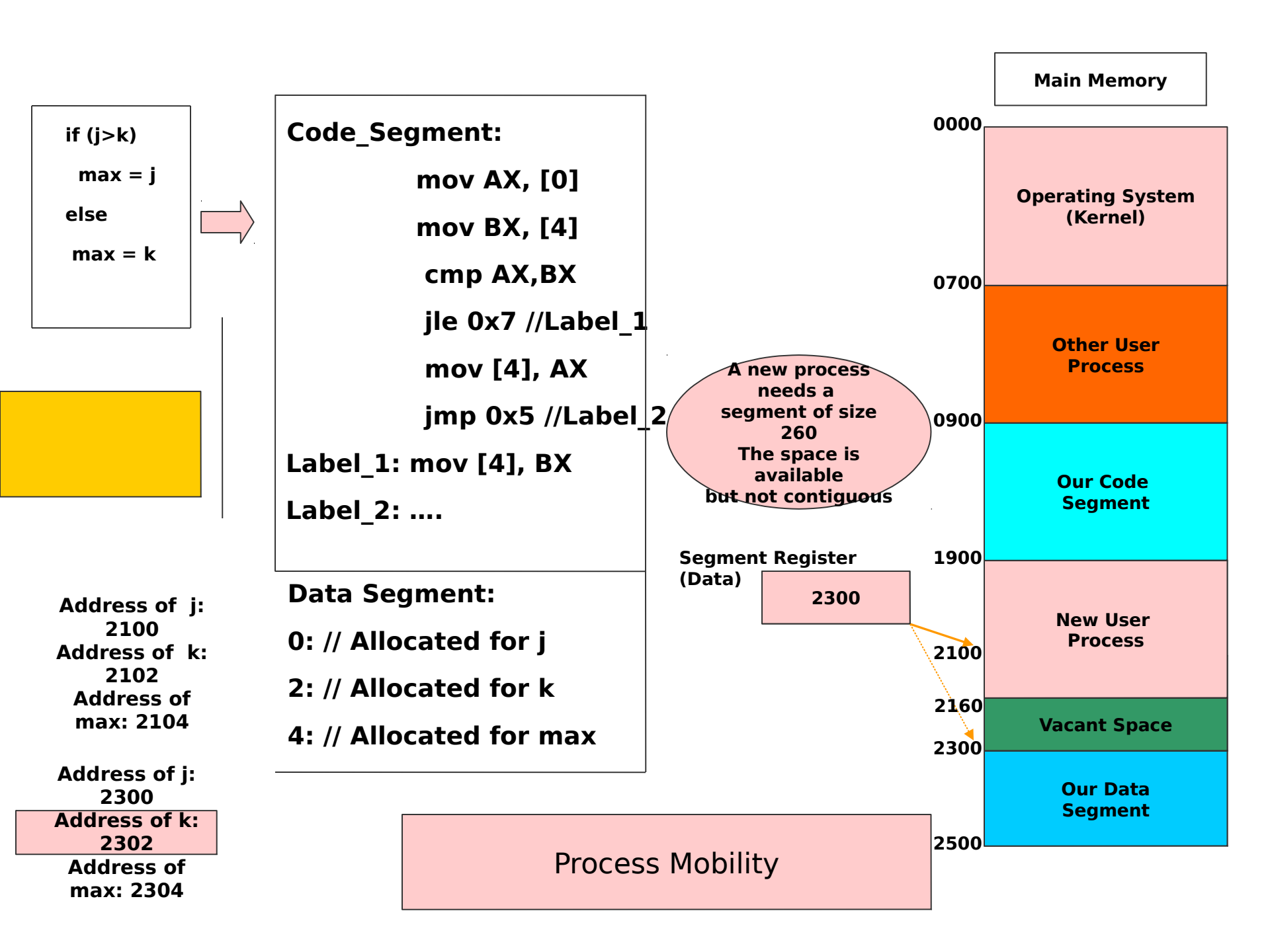
EU executes instructions that have already been fetched by the BIU.

BIU and EU functions separately.

Bus Interface Unit (BIU)

BIU fetches instructions, reads data from memory and I/O ports, writes data to memory and I/ O ports.


if (j>k)
max = j
else
max = k
Code_Segment:
mov AX, [0]
mov BX, [2]
cmp AX,BX
jle 0x7 //Label_1
mov [4], AX
jmp 0x5 //Label_2
Label_1: mov [4], BX
Label_2: ....
Data Segment:
0: // Allocated for j
2: // Allocated for k
4: // Allocated for max
Address of j: 2100
Address of k: 2102
Address of max: 2104
Code and Data segments are separate and both assumed to start from 0
Every Memory Data Access should add the value stored in Data Segment Register By default.
Segment Register (Data)
2100
Main Memory
0000
Operating System (Kernel)
0700
Other User Process
0900
Our Code Segment
1900
Vacant Space
2100
Our Data Segment
2300
Vacant Space
2500
Ease Of Programming


if (j>k)
max = j
else
max = k
Code_Segment:
mov AX, [0]
mov BX, [4]
cmp AX,BX
jle 0x7 //Label_1
mov [4], AX
jmp 0x5 //Label_2
Label_1: mov [4], BX
Label_2: ....
Data Segment:
0: // Allocated for j
2: // Allocated for k
4: // Allocated for max
Address of j: 2100
Address of k: 2102
Address of max: 2104
Address of j: 2300
Address of k: 2302
Address of max: 2304
A new process needs a segment of size 260
The space is available but not contiguous
Segment Register (Data)
2300
Main Memory
0000
Operating System (Kernel)
0700
Other User Process
0900
Our Code Segment
1900
New User Process
2100
2160
Vacant Space
2300
Our Data Segment
2500
Process Mobility

Multiple Segments

- The segment register can change its values to point to different segments at different times.
- X86 architecture provides additional segment registers to access multi data segments at the same time.
  - DS, ES, FS and GS
- X86 supports a separate Stack Segment Register (SS) and a Code segment Register (CS) in addition.
- By default a segment register is fixed for every instruction, for all the memory access performed by it. For eg. all data accessed by MOV instruction take DS as the default segment register.
- An **segment override prefix** is attached to an instruction to change the segment register it uses for memory data access.

mov [10], ax

- this will move the contents of ax register to memory location 0510

Opcode: 0x89 0x05 0x10

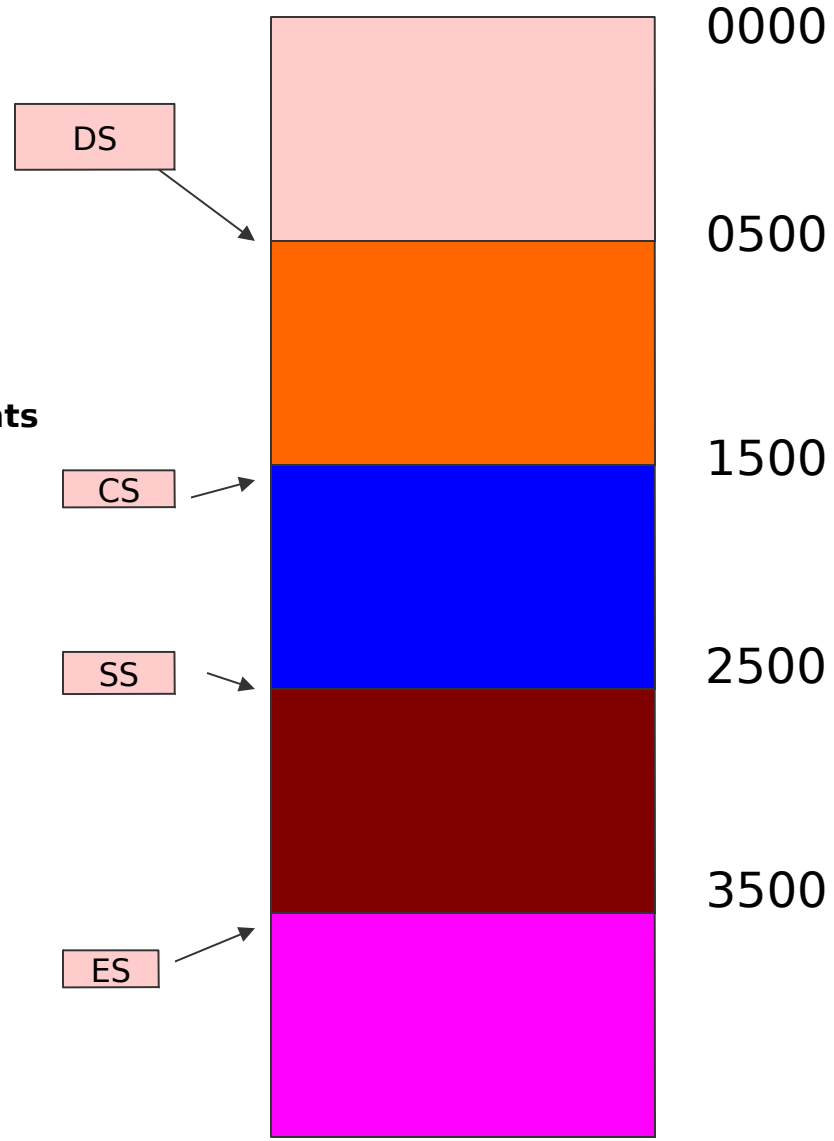
mov [ES:10], ax

-this will move the contents of ax register to memory location 3510

Opcode

0x26 0x89 0x05 0x10

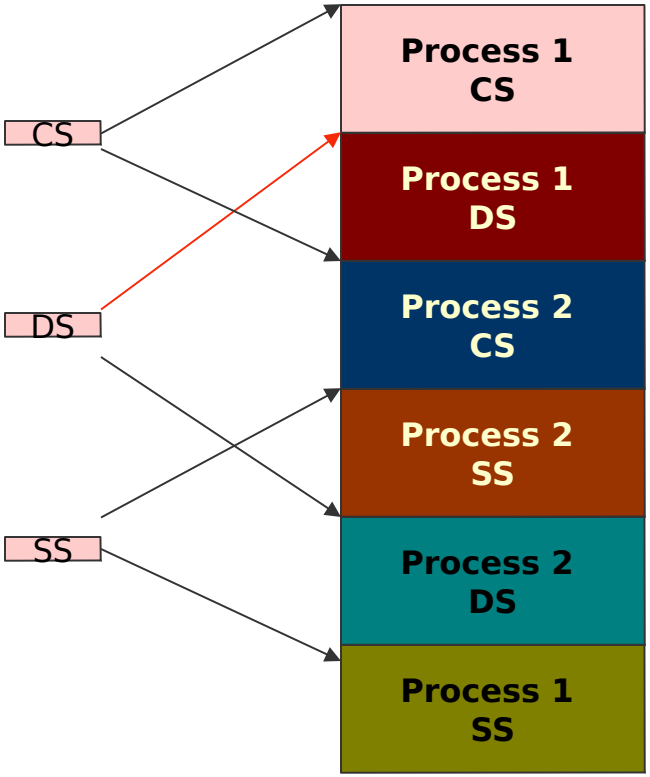
“0x26” is the segment override prefix.



Multiple Segments

Process 1 in Execution

Process 2 in Execution



Multiprocess Context switching

- Three salient features of using Segmentation

- Three Features
  - Code Mobility
  - Logically every segment can start with zero
  - Inter and Intra process protection ensuring data integrity.

Real Mode - Memory Addressing

- Segment << 4 + offset = 20 bit EA
- Segment size is a fixed 64K

DS = 0x1004

```
mov  [0x1000], AX
```

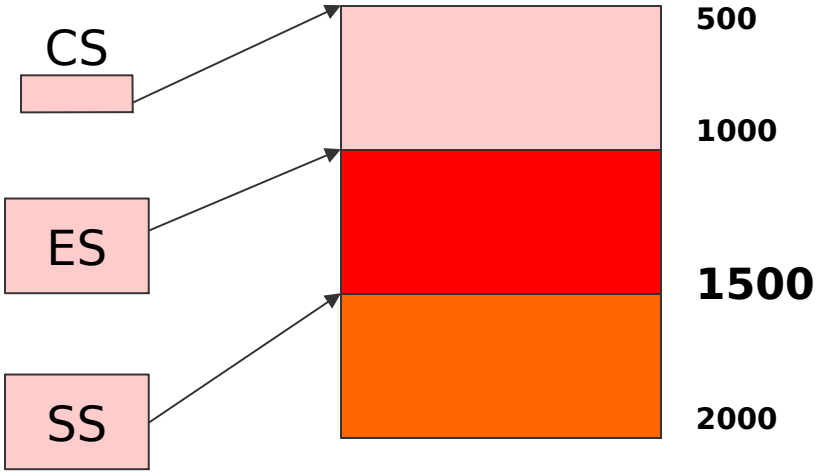
The mov will store the content of AX in

0x10040 + 0x1000 = 0x11040

Why this stuff? - To get 1 MB addressing using 16-bit Segment Registers

•A process always executes from Code segment. It should not execute by accessing from adjoining Data or stack area or any other code area too.

•A stack should not overgrow into adjoining segments



Every segment is specified a start address and limit.

Architecture checks if limit is not exceeded.

POP AX //Let SP be 0, Violation!!!

Intra and Inter process Protection

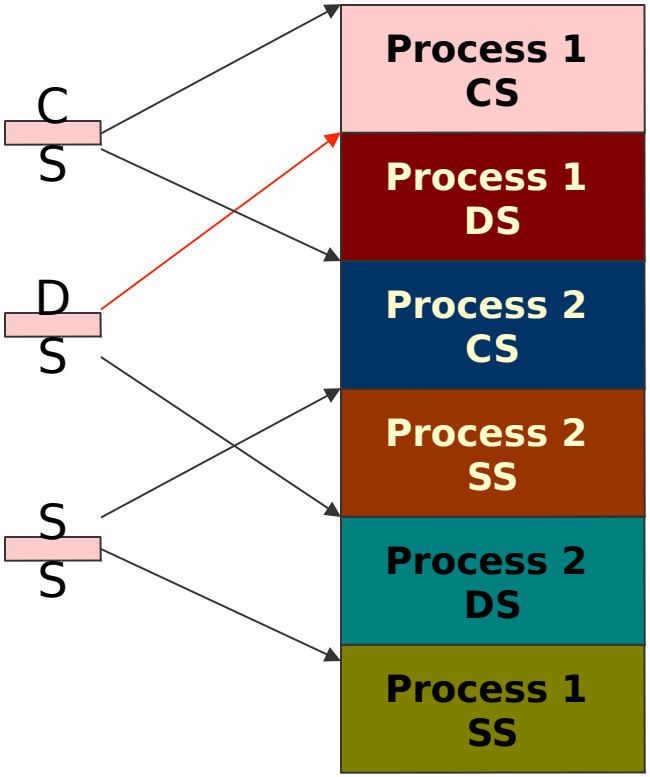
Process 1 should be prevented from loading CS, such that it can access the code of Process 2

Similarly for the DS,SS, ES, FS and GS

Privilege levels: [0-3] assigned to each segment.

0: Highest privilege

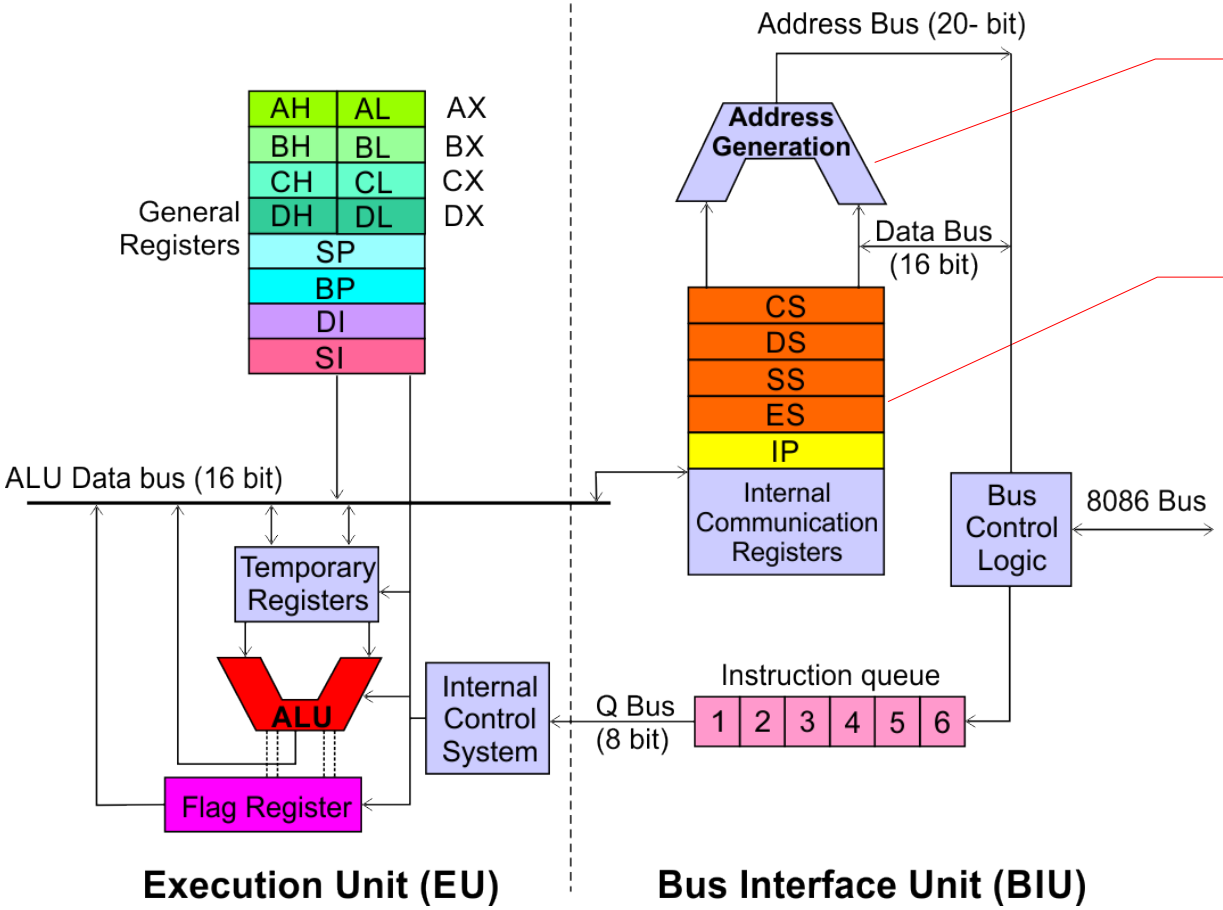
3: Lowest privilege



Interprocess Protection

8086 Microprocessor Architecture

Bus Interface Unit (BIU)



Dedicated Adder to generate 20 bit address

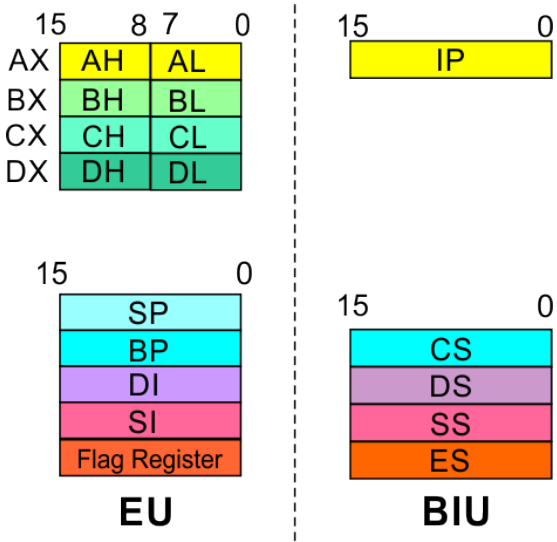
Four 16-bit segment registers

Code Segment (CS)
Data Segment (DS)
Stack Segment (SS)
Extra Segment (ES)

Segment Registers

Code Segment Register

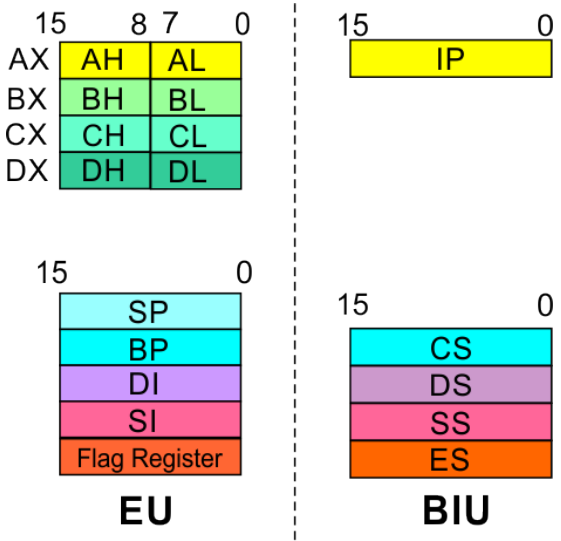
- 16-bit
- CS contains the base or start of the current code segment; IP contains the distance or offset from this address to the next instruction byte to be fetched.
- BIU computes the 20-bit physical address by logically shifting the contents of CS 4-bits to the left and then adding the 16-bit contents of IP.
- That is, all instructions of a program are relative to the contents of the CS register multiplied by 16 and then offset is added provided by the IP.



Segment Registers

Data Segment Register

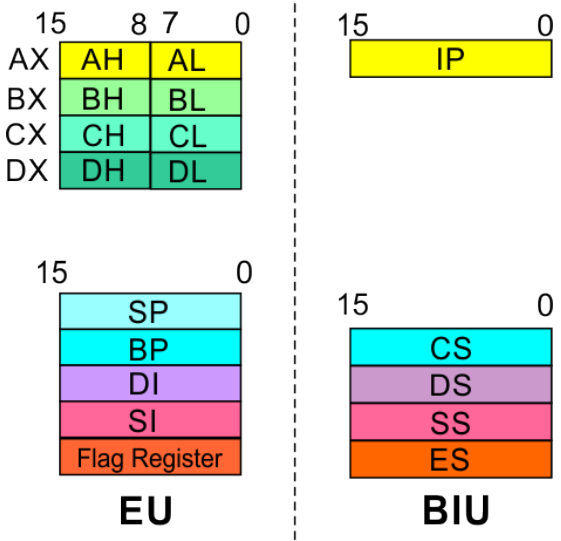
- 16-bit
- Points to the current data segment; operands for most instructions are fetched from this segment.
- The 16-bit contents of the Source Index (SI) or Destination Index (DI) or a 16-bit displacement are used as offset for computing the 20-bit physical address.



Segment Registers

Stack Segment Register

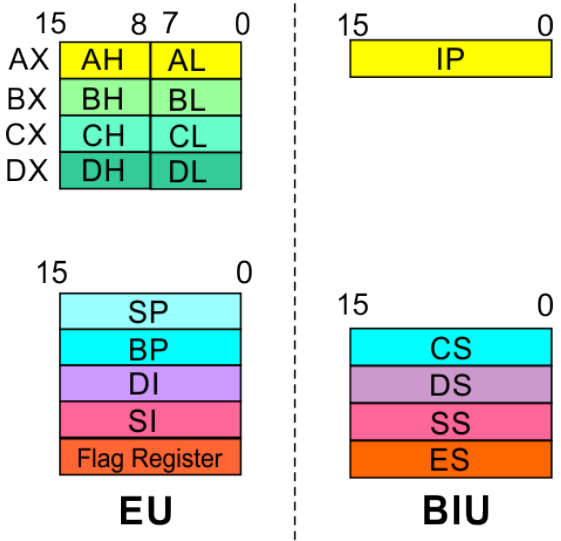
- 16-bit
- Points to the current stack.
- The 20-bit physical stack address is calculated from the Stack Segment (SS) and the Stack Pointer (SP) for stack instructions such as PUSH and POP.
- In based addressing mode, the 20-bit physical stack address is calculated from the Stack segment (SS) and the Base Pointer (BP).



Segment Registers

Extra Segment Register

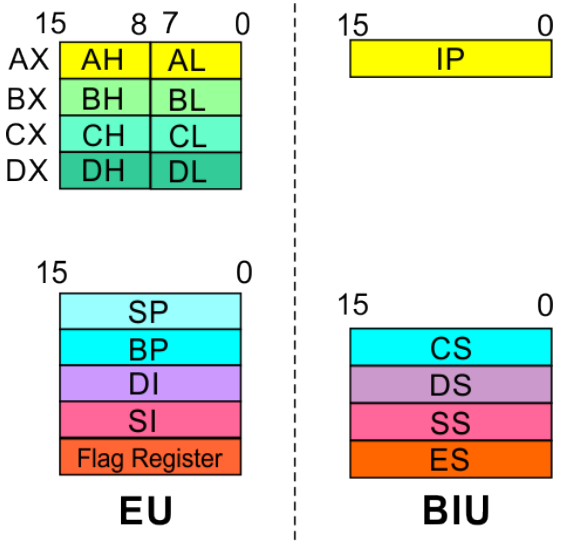
- 16-bit
- Points to the extra segment in which data (in excess of 64K pointed to by the DS) is stored.
- String instructions use the ES and DI to determine the 20-bit physical address for the destination.



Segment Registers

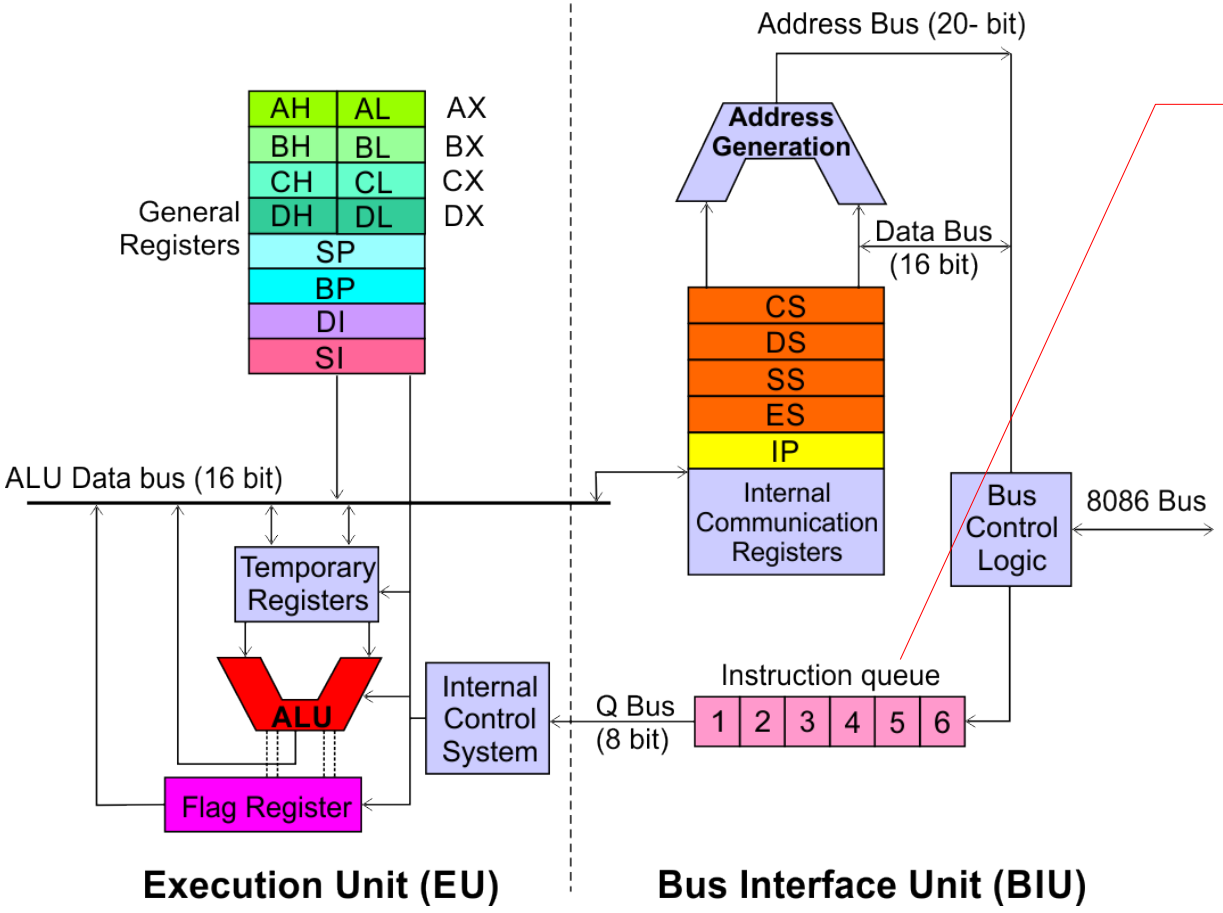
Instruction Pointer

- 16-bit
- Always points to the next instruction to be executed within the currently executing code segment.
- So, this register contains the 16-bit offset address pointing to the next instruction code within the 64Kb of the code segment area.
- Its content is automatically incremented as the execution of the next instruction takes place.



8086 Microprocessor Architecture

Bus Interface Unit (BIU)



Instruction queue

- **A group of First-In-First-Out (FIFO) in which up to 6 bytes of instruction code are pre fetched from the memory ahead of time.**
- **This is done in order to speed up the execution by overlapping instruction fetch with execution.**
- **This mechanism is known as pipelining.**

Architecture

Execution Unit (EU)

EU decodes and executes instructions.

A decoder in the EU control system translates instructions.

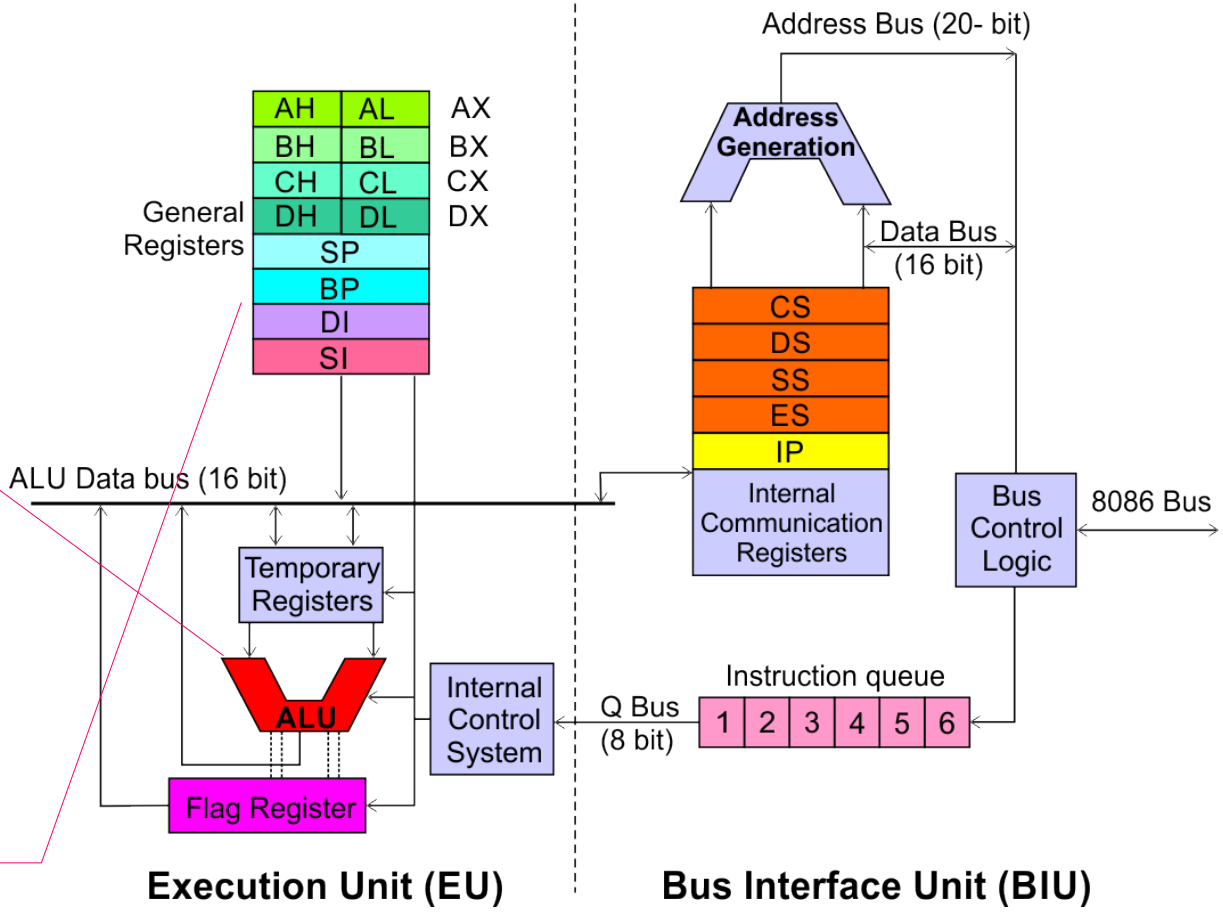
16-bit ALU for performing arithmetic and logic operation

Four general purpose registers(AX, BX, CX, DX);

Pointer registers (Stack Pointer, Base Pointer);

and

Index registers (Source Index, Destination Index) each of 16-bits



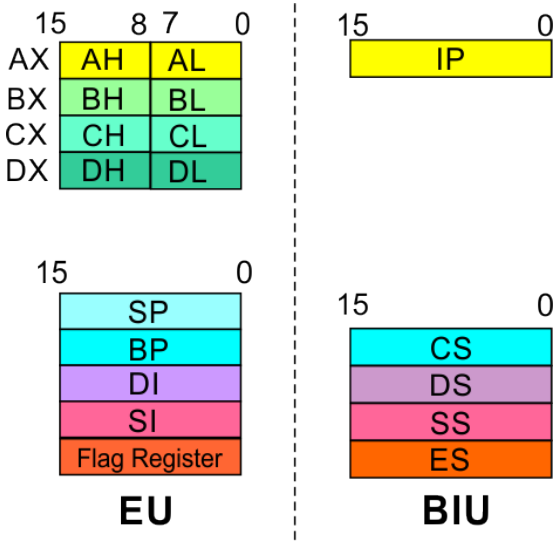
Some of the 16 bit registers can be used as two 8 bit registers as :

AX can be used as AH and AL
BX can be used as BH and BL
CX can be used as CH and CL
DX can be used as DH and DL

EU Registers

Accumulator Register (AX)

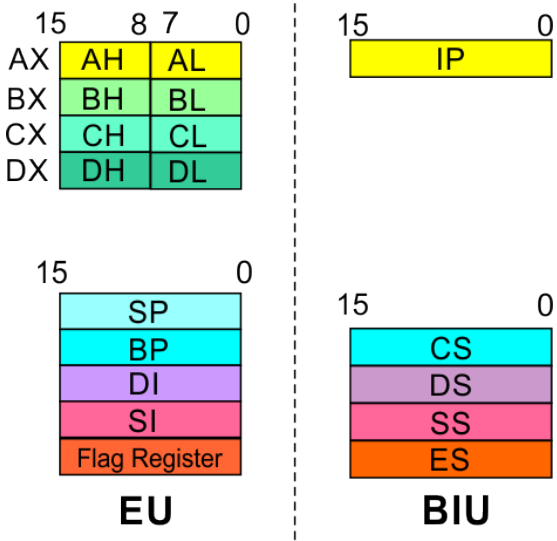
- Consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX.
- AL in this case contains the low order byte of the word, and AH contains the high-order byte.
- The I/O instructions use the AX or AL for inputting / outputting 16 or 8 bit data to or from an I/O port.
- Multiplication and Division instructions also use the AX or AL.



EU Registers

Base Register (BX)

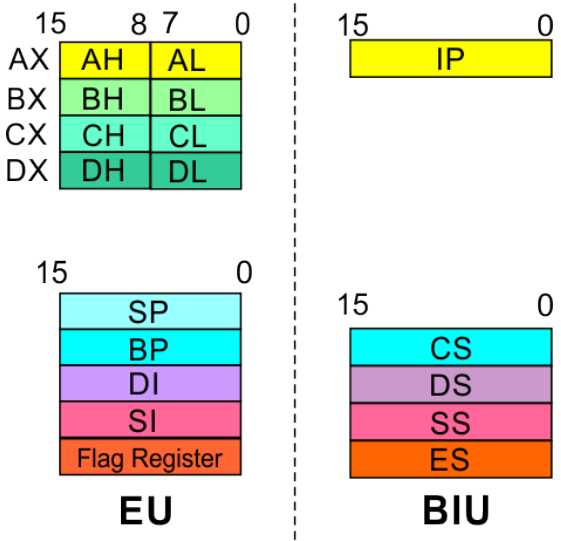
- Consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX.
- BL in this case contains the low-order byte of the word, and BH contains the high-order byte.
- This is the only general purpose register whose contents can be used for addressing the 8086 memory.
- All memory references utilizing this register content for addressing use DS as the default segment register.



EU Registers

Counter Register (CX)

- Consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX.
- When combined, CL register contains the low order byte of the word, and CH contains the high-order byte.
- Instructions such as **SHIFT**, **ROTATE** and **LOOP** use the contents of CX as a counter.



Example:

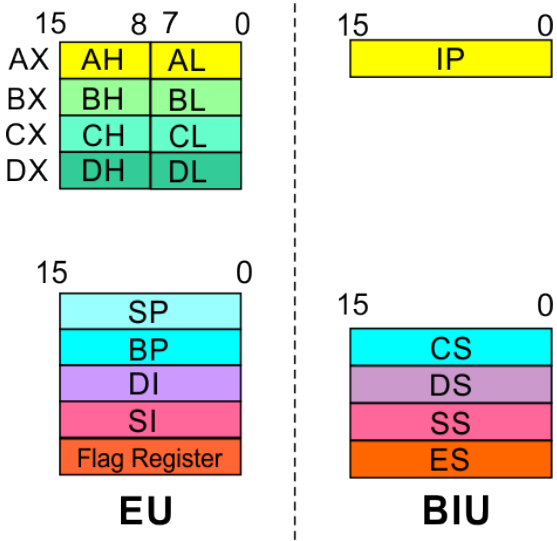
The instruction LOOP START automatically decrements CX by 1 without affecting flags and will check if [CX] = 0.

If it is zero, 8086 executes the next instruction; otherwise the 8086 branches to the label START.

EU Registers

Data Register (DX)

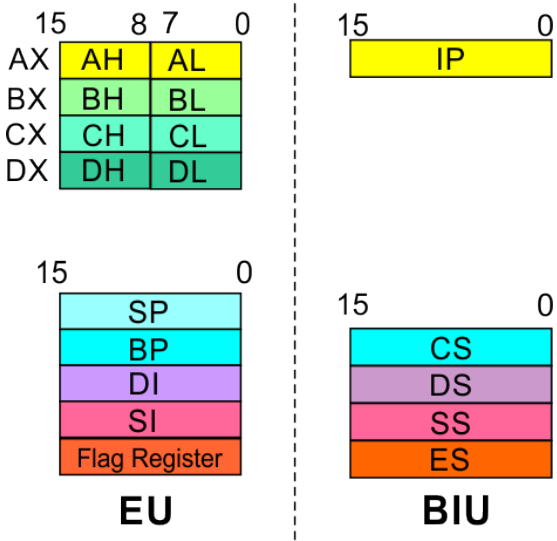
- Consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX.
- When combined, DL register contains the low-order byte of the word, and DH contains the high-order byte.
- Used to hold the high 16-bit result (data) in 16 X 16 multiplication or the high 16-bit dividend (data) before a 32 16 division and the 16-bit reminder after division.



EU Registers

Stack Pointer (SP) and Base Pointer (BP)

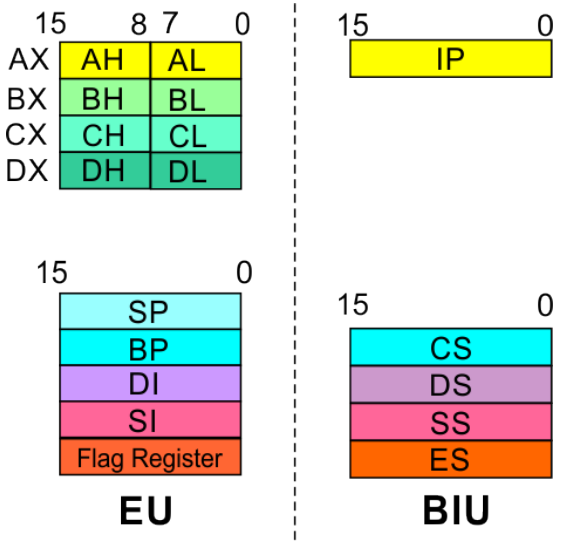
- SP and BP are used to access data in the stack segment.
- SP is used as an offset from the current SS during execution of instructions that involve the stack segment in the external memory.
- SP contents are automatically updated (incremented/ decremented) due to execution of a POP or PUSH instruction.
- BP contains an offset address in the current SS, which is used by instructions utilizing the based addressing mode.



EU Registers

Source Index (SI) and Destination Index (DI)

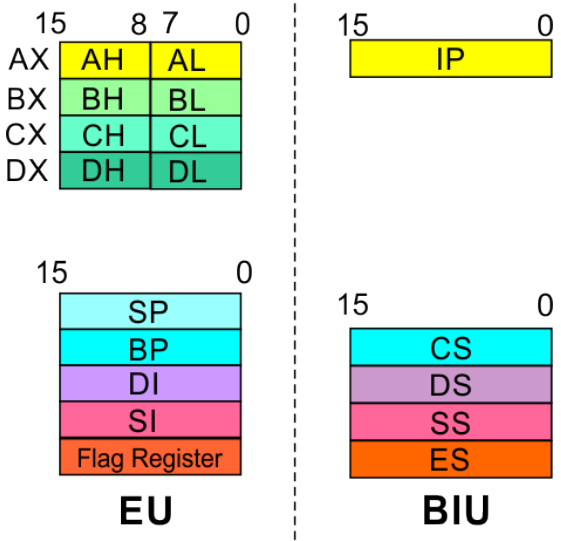
- Used in indexed addressing.
- Instructions that process data strings use the SI and DI registers together with DS and ES respectively in order to distinguish between the source and destination addresses.



EU Registers

Source Index (SI) and Destination Index (DI)

- **Used in indexed addressing.**
- **Instructions that process data strings use the SI and DI registers together with DS and ES respectively in order to distinguish between the source and destination addresses.**



8086 Microprocessor

Architecture

Execution Unit (EU)

Flag Register

Auxiliary Carry Flag

This is set, if there is a carry from the lowest nibble, i.e, bit three during addition, or borrow for the lowest nibble, i.e, bit three, during subtraction.

Carry Flag

This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

Sign Flag

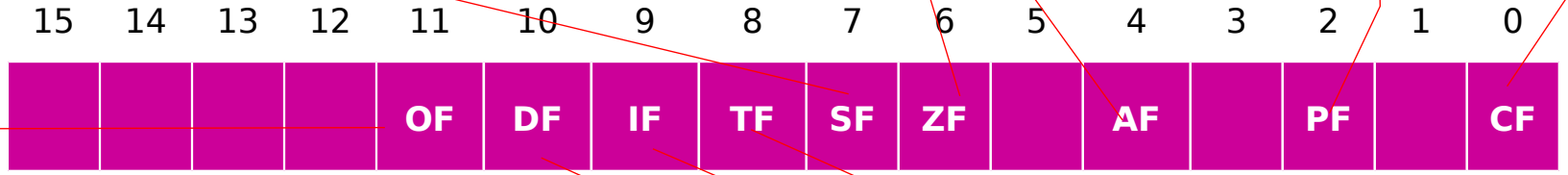
This flag is set, when the result of any computation is negative

Zero Flag

This flag is set, if the result of the computation or comparison performed by an instruction is zero

Parity Flag

This flag is set to 1, if the lower byte of the result contains even number of 1's ; for odd number of 1's set to zero.



Over flow Flag

This flag is set, if an overflow occurs, i.e, if the result of a signed operation is large enough to accommodate in a destination register. The result is of more than 7-bits in size in case of 8-bit signed operation and more than 15-bits in size in case of 16-bit sign operations, then the overflow will be set.

Trap Flag

If this flag is set, the processor enters the single step execution mode by generating internal interrupts after the execution of each instruction

Direction Flag

This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e., auto incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto incrementing mode.

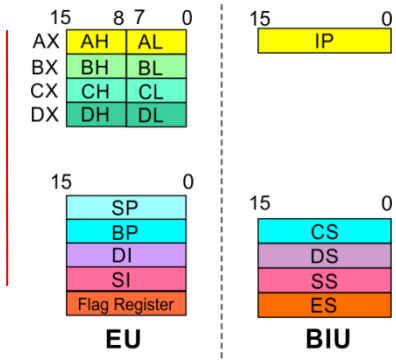
Interrupt Flag

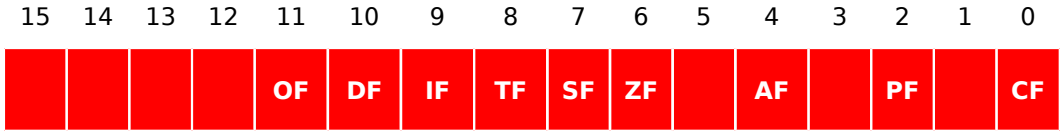
Causes the 8086 to recognize external mask interrupts; clearing IF disables these interrupts.

8086 Microprocessor

Architecture

8086 registers categorized groups





Sl.No.	Type	Register width	Name of register
1	General purpose register	16 bit	AX, BX, CX, DX
		8 bit	AL, AH, BL, BH, CL, CH, DL, DH
2	Pointer register	16 bit	SP, BP
3	Index register	16 bit	SI, DI
4	Instruction Pointer	16 bit	IP
5	Segment register	16 bit	CS, DS, SS, ES
6	Flag (PSW)	16 bit	Flag register

Register	Name of the Register	Special Function
AX	16-bit Accumulator	Stores the 16-bit results of arithmetic and logic operations
AL	8-bit Accumulator	Stores the 8-bit results of arithmetic and logic operations
BX	Base register	Used to hold base value in base addressing mode to access memory data
CX	Count Register	Used to hold the count value in SHIFT, ROTATE and LOOP instructions
DX	Data Register	Used to hold data for multiplication and division operations
SP	Stack Pointer	Used to hold the offset address of top stack memory
BP	Base Pointer	Used to hold the base value in base addressing using SS register to access data from stack memory
SI	Source Index	Used to hold index value of source operand (data) for string instructions
DI	Data Index	Used to hold the index value of destination operand (data) for string operations

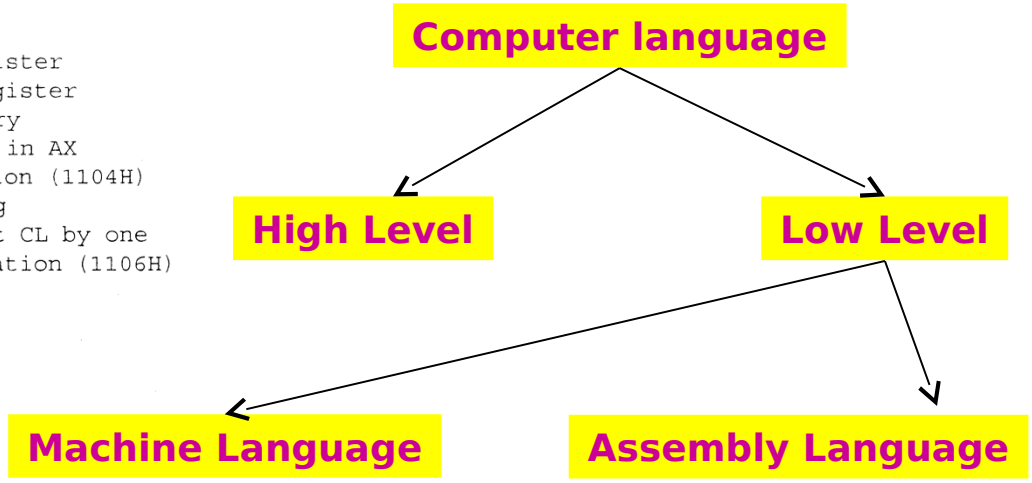
ADDRESSING MODES
&
Instruction set

Introduction

```
;PROGRAM TO ADD TWO 16-BIT DATA (METHOD-1)

DATA SEGMENT            ;Assembler directive

        ORG 1104H       ;Assembler directive
        SUM DW 0        ;Assembler directive
        CARRY DB 0      ;Assembler directive

DATA ENDS               ;Assembler directive

CODE SEGMENT            ;Assembler directive

        ASSUME CS:CODE  ;Assembler directive
        ASSUME DS:DATA  ;Assembler directive
        ORG 1000H       ;Assembler directive

        MOV AX,205AH    ;Load the first data in AX register
        MOV BX,40EDH    ;Load the second data in BX register
        MOV CL,00H      ;Clear the CL register for carry
        ADD AX,BX       ;Add the two data, sum will be in AX
        MOV SUM,AX      ;Store the sum in memory location (1104H)
        JNC AHEAD       ;Check the status of carry flag
        INC CL          ;If carry flag is set,increment CL by one
AHEAD:  MOV CARRY,CL    ;Store the carry in memory location (1106H)
        HLT

CODE ENDS               ;Assembler directive
END                     ;Assembler directive
```

Program
A set of instructions written to solve a problem.

Instruction
Directions which a microprocessor follows to execute a task or part of a task.



- **Binary bits**

- **English Alphabets**
- **'Mnemonics'**
- **Assembler**
  Mnemonics → Machine Language

ADDRESSING MODES

Addressing Modes

- Every instruction of a program has to operate on a data.
- The different ways in which a source operand is denoted in an instruction are known as addressing modes.

1. Register Addressing
2. Immediate Addressing

Group I : Addressing modes for register and immediate data

3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing

Group II : Addressing modes for memory data

9. Direct I/O port Addressing
10. Indirect I/O port Addressing

Group III : Addressing modes for I/O ports

11. Relative Addressing

Group IV : Relative Addressing mode

12. Implied Addressing

Group V : Implied Addressing mode

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

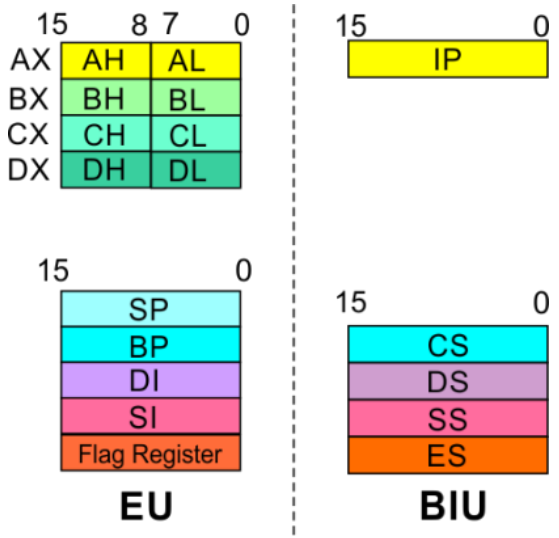
The instruction will specify the name of the register which holds the data to be operated by the instruction.

Example:

MOV CL, DH

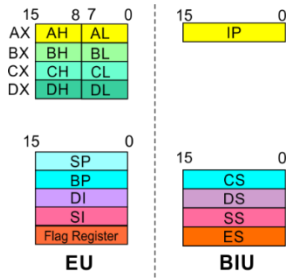
The content of 8-bit register DH is moved to another 8-bit register CL

(CL) ← (DH)



Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In immediate addressing mode, an 8-bit or 16-bit data is specified as part of the instruction

Example:

MOV DL, 08H

The 8-bit data (08_H) given in the instruction is moved to DL

(DL) ← 08_H

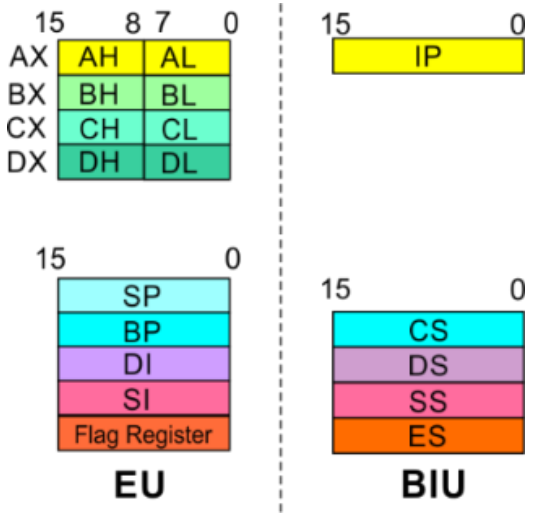
MOV AX, 0A9FH

The 16-bit data ($0A9F_H$) given in the instruction is moved to AX register

(AX) ← $0A9F_H$

Addressing Modes : Memory Access

- **20 Address lines ⇒ 8086 can address up to 2^{20} = 1M bytes of memory**
- **However, the largest register is only 16 bits**
- **Physical Address will have to be calculated**
  Physical Address : Actual address of a byte in memory. i.e. the value which goes out onto the address bus.
- **Memory Address represented in the form –**
  Seg : Offset (Eg - 89AB:F012)
- **Each time the processor wants to access memory, it takes the contents of a segment register, shifts it one hexadecimal place to the left (same as multiplying by 16_{10}), then add the required offset to form the 20- bit address**

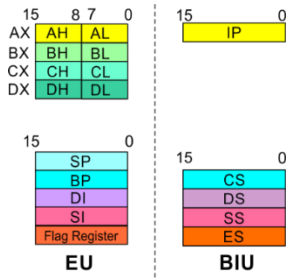


16 bytes of contiguous memory

89AB : F012 → 89AB → 89AB0 (Paragraph to byte → 89AB x 10 = 89AB0)
F012 → 0F012 (Offset is already in byte unit)
+ -------
98AC2 (The absolute address)

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



Here, the effective address of the memory location at which the data operand is stored is given in the instruction.

The effective address is just a 16-bit number written directly in the instruction.

Example:

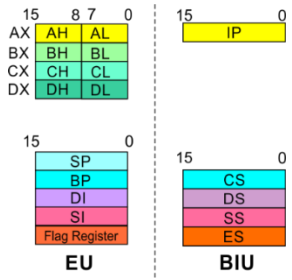
```
MOV   BX, [1354H]
MOV   BL,  [0400H]
```

The square brackets around the 1354_H denotes the contents of the memory location. When executed, this instruction will copy the contents of the memory location into BX register.

This addressing mode is called direct because the displacement of the operand from the segment base is specified directly in the instruction.

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Register indirect addressing, name of the register which holds the effective address (EA) will be specified in the instruction.

Registers used to hold EA are any of the following registers:

BX, BP, DI and SI.

Content of the DS register is used for base address calculation.

Example:

MOV CX, [BX]

Note : Register/ memory enclosed in brackets refer to content of register/ memory

Operations:

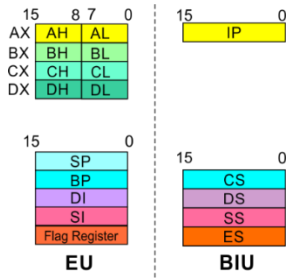
EA = (BX)
BA = (DS) x 16_{10}
MA = BA + EA

(CX) ← (MA) or,

(CL) ← (MA)
(CH) ← (MA +1)

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Based Addressing, BX or BP is used to hold the base value for effective address and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

In case of 8-bit displacement, it is sign extended to 16-bit before adding to the base value.

When BX holds the base value of EA, 20-bit physical address is calculated from BX and DS.

When BP holds the base value of EA, BP and SS is used.

Example:

MOV AX, [BX + 08H]

Operations:

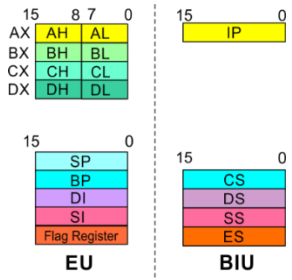
$0008_H \leftarrow 08_H$ (Sign extended)
EA = (BX) + 0008_H
BA = (DS) x 16_{10}
MA = BA + EA

(AX) ← (MA) or,

(AL) ← (MA)
(AH) ← (MA + 1)

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



SI or DI register is used to hold an index value for memory data and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

Displacement is added to the index value in SI or DI register to obtain the EA.

In case of 8-bit displacement, it is sign extended to 16-bit before adding to the base value.

Example:

MOV CX, [SI + 0A2H]

Operations:

$FFA2_H \leftarrow A2_H$ **(Sign extended)**

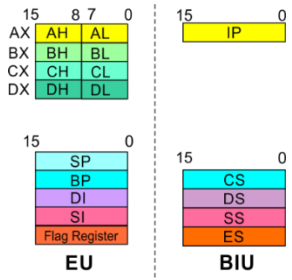
$EA = (SI) + FFA2_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

$(CX) \leftarrow (MA)$ **or,**

$(CL) \leftarrow (MA)$
$(CH) \leftarrow (MA + 1)$

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Based Index Addressing, the effective address is computed from the sum of a base register (BX or BP), an index register (SI or DI) and a displacement.

Example:

MOV DX, [BX + SI + 0AH]

Operations:

$000A_H \leftarrow 0A_H$ **(Sign extended)**

$EA = (BX) + (SI) + 000A_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

$(DX) \leftarrow (MA)$ **or,**

$(DL) \leftarrow (MA)$
$(DH) \leftarrow (MA + 1)$

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

Note : Effective address of the Extra segment register

Employed in string operations to operate on string data.

The effective address (EA) of source data is stored in SI register and the EA of destination is stored in DI register.

Segment register for calculating base address of source data is DS and that of the destination data is ES

Example: MOVS BYTE

Operations:

Calculation of source memory location:

$EA = (SI)$ $BA = (DS) \times 16_{10}$ $MA = BA + EA$

Calculation of destination memory location:

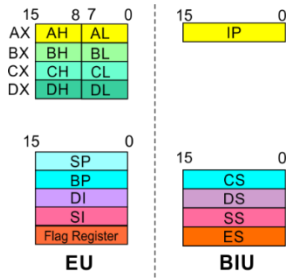
$EA_E = (DI)$ $BA_E = (ES) \times 16_{10}$ $MA_E = BA_E + EA_E$

$(MAE) \leftarrow (MA)$

If DF = 1, then $(SI) \leftarrow (SI) - 1$ and $(DI) = (DI) - 1$
If DF = 0, then $(SI) \leftarrow (SI) + 1$ and $(DI) = (DI) + 1$

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



These addressing modes are used to access data from standard I/O mapped devices or ports.

In direct port addressing mode, an 8-bit port address is directly specified in the instruction.

Example: **IN AL, [09H]**

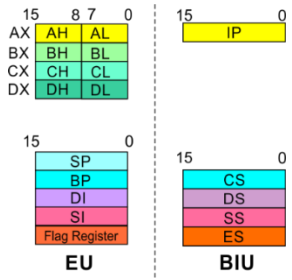
Operations: $PORT_{addr} = 09_H$

$(AL) \leftarrow (PORT)$

Content of port with address 09_H is moved to AL register

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In this addressing mode, the effective address of a program instruction is specified relative to Instruction Pointer (IP) by an 8-bit signed displacement.

Example: JZ 0AH

Operations:

$000A_H \leftarrow 0A_H$ (sign extend)

If ZF = 1, then

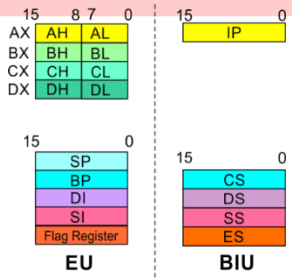
$EA = (IP) + 000A_H$
$BA = (CS) \times 16_{10}$
MA = BA + EA

If ZF = 1, then the program control jumps to new address calculated above.

If ZF = 0, then next instruction of the program is executed.

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



Instructions using this mode have no operands. The instruction itself will specify the data to be operated by the instruction.

Example: **CLC**

This clears the carry flag to zero.

INSTRUCTION SET

Instruction Set

8086 supports 6 types of instructions.

1. **Data Transfer Instructions**
2. **Arithmetic Instructions**
3. **Logical Instructions**
4. **String manipulation Instructions**
5. **Process Control Instructions**
6. **Control Transfer Instructions**

Instruction Set

1. Data Transfer Instructions

Instructions that are used to transfer data/ address in to registers, memory locations and I/O ports.

Generally involve two operands: Source operand and Destination operand of the same size.

Source: Register or a memory location or an immediate data
Destination : Register or a memory location.

The size should be a either a byte or a word.

A 8-bit data can only be moved to 8-bit register/ memory and a 16-bit data can be moved to 16-bit register/ memory.

Instruction Set

1. Data Transfer Instructions

Mnemonics: MOV, XCHG, PUSH, POP, IN, OUT …

MOV reg2/ mem, reg1/ mem **MOV reg2, reg1** **MOV mem, reg1** **MOV reg2, mem**	 **(reg2) ← (reg1)** **(mem) ← (reg1)** **(reg2) ← (mem)**
MOV reg/ mem, data **MOV reg, data** **MOV mem, data**	 **(reg) ← data** **(mem) ← data**

XCHG reg2/ mem, reg1 **XCHG reg2, reg1** **XCHG mem, reg1**	 **(reg2) ↔ (reg1)** **(mem) ↔ (reg1)**

Instruction Set

1. Data Transfer Instructions

Mnemonics: MOV, XCHG, PUSH, POP, IN, OUT …

PUSH reg16/ mem	
PUSH reg16	**(SP) ← (SP) – 2** **MA_S = (SS) x 16_{10} + SP** **(MA_S ; MA_S + 1) ← (reg16)**
PUSH mem	**(SP) ← (SP) – 2** **MA_S = (SS) x 16_{10} + SP** **(MA_S ; MA_S + 1) ← (mem)**
POP reg16/ mem	
POP reg16	**MA_S = (SS) x 16_{10} + SP** **(reg16) ← (MA_S ; MA_S + 1)** **(SP) ← (SP) + 2**
POP mem	**MA_S = (SS) x 16_{10} + SP** **(mem) ← (MA_S ; MA_S + 1)** **(SP) ← (SP) + 2**

Instruction Set

1. Data Transfer Instructions

Mnemonics: MOV, XCHG, PUSH, POP, IN, OUT …

IN A, [DX]	
IN AL, [DX]	**$PORT_{addr}$ = (DX)** **(AL) ← (PORT)**
IN AX, [DX]	**$PORT_{addr}$ = (DX)** **(AX) ← (PORT)**
IN A, addr8	
IN AL, addr8	**(AL) ← (addr8)**
IN AX, addr8	**(AX) ← (addr8)**

OUT [DX], A	
OUT [DX], AL	**$PORT_{addr}$ = (DX)** **(PORT) ← (AL)**
OUT [DX], AX	**$PORT_{addr}$ = (DX)** **(PORT) ← (AX)**
OUT addr8, A	
OUT addr8, AL	**(addr8) ← (AL)**
OUT addr8, AX	**(addr8) ← (AX)**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

ADD reg2/ mem, reg1/mem **ADC reg2, reg1** **ADC reg2, mem** **ADC mem, reg1**	 **(reg2) ← (reg1) + (reg2)** **(reg2) ← (reg2) + (mem)** **(mem) ← (mem)+(reg1)**
ADD reg/mem, data **ADD reg, data** **ADD mem, data**	 **(reg) ← (reg)+ data** **(mem) ← (mem)+data**
ADD A, data **ADD AL, data8** **ADD AX, data16**	 **(AL) ← (AL) + data8** **(AX) ← (AX) +data16**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

ADC reg2/ mem, reg1/mem **ADC reg2, reg1** **ADC reg2, mem** **ADC mem, reg1**	 **(reg2) ← (reg1) + (reg2)+CF** **(reg2) ← (reg2) + (mem)+CF** **(mem) ← (mem)+(reg1)+CF**
ADC reg/mem, data **ADC reg, data** **ADC mem, data**	 **(reg) ← (reg)+ data+CF** **(mem) ← (mem)+data+CF**
ADC A, data **ADC AL, data8** **ADC AX, data16**	 **(AL) ← (AL) + data8+CF** **(AX) ← (AX) +data16+CF**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

SUB reg2/ mem, reg1/mem **SUB reg2, reg1** **SUB reg2, mem** **SUB mem, reg1**	 **(reg2) ← (reg1) - (reg2)** **(reg2) ← (reg2) - (mem)** **(mem) ← (mem) - (reg1)**
SUB reg/mem, data **SUB reg, data** **SUB mem, data**	 **(reg) ← (reg) - data** **(mem) ← (mem) - data**
SUB A, data **SUB AL, data8** **SUB AX, data16**	 **(AL) ← (AL) - data8** **(AX) ← (AX) - data16**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

SBB reg2/ mem, reg1/mem **SBB reg2, reg1** **SBB reg2, mem** **SBB mem, reg1**	 **(reg2) ← (reg1) - (reg2) - CF** **(reg2) ← (reg2) - (mem)- CF** **(mem) ← (mem) - (reg1) –CF**
SBB reg/mem, data **SBB reg, data** **SBB mem, data**	 **(reg) ← (reg) – data - CF** **(mem) ← (mem) - data - CF**
SBB A, data **SBB AL, data8** **SBB AX, data16**	 **(AL) ← (AL) - data8 - CF** **(AX) ← (AX) - data16 - CF**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

INC reg/ mem	
INC reg8	**(reg8) ← (reg8) + 1**
INC reg16	**(reg16) ← (reg16) + 1**
INC mem	**(mem) ← (mem) + 1**
DEC reg/ mem	
DEC reg8	**(reg8) ← (reg8) - 1**
DEC reg16	**(reg16) ← (reg16) - 1**
DEC mem	**(mem) ← (mem) - 1**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

MUL reg/ mem	
MUL reg	**For byte : (AX) ← (AL) x (reg8)** **For word : (DX)(AX) ← (AX) x (reg16)**
MUL mem	**For byte : (AX) ← (AL) x (mem8)** **For word : (DX)(AX) ← (AX) x (mem16)**
IMUL reg/ mem	
IMUL reg	**For byte : (AX) ← (AL) x (reg8)** **For word : (DX)(AX) ← (AX) x (reg16)**
IMUL mem	**For byte : (AX) ← (AX) x (mem8)** **For word : (DX)(AX) ← (AX) x (mem16)**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

DIV reg/ mem	
DIV reg	**For 16-bit :- 8-bit :** **(AL) ← (AX) :- (reg8) Quotient** **(AH) ← (AX) MOD(reg8) Remainder** **For 32-bit :- 16-bit :** **(AX) ← (DX)(AX) :- (reg16) Quotient** **(DX) ← (DX)(AX) MOD(reg16) Remainder**
DIV mem	**For 16-bit :- 8-bit :** **(AL) ← (AX) :- (mem8) Quotient** **(AH) ← (AX) MOD(mem8) Remainder** **For 32-bit :- 16-bit :** **(AX) ← (DX)(AX) :- (mem16) Quotient** **(DX) ← (DX)(AX) MOD(mem16) Remainder**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

IDIV reg/ mem	
IDIV reg	**For 16-bit :- 8-bit :** (AL) ← (AX) :- (reg8) Quotient (AH) ← (AX) MOD(reg8) Remainder **For 32-bit :- 16-bit :** (AX) ← (DX)(AX) :- (reg16) Quotient (DX) ← (DX)(AX) MOD(reg16) Remainder
IDIV mem	**For 16-bit :- 8-bit :** (AL) ← (AX) :- (mem8) Quotient (AH) ← (AX) MOD(mem8) Remainder **For 32-bit :- 16-bit :** (AX) ← (DX)(AX) :- (mem16) Quotient (DX) ← (DX)(AX) MOD(mem16) Remainder

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

CMP reg2/mem, reg1/ mem	
CMP reg2, reg1	**Modify flags ← (reg2) – (reg1)** **If (reg2) > (reg1) then CF=0, ZF=0, SF=0** **If (reg2) < (reg1) then CF=1, ZF=0, SF=1** **If (reg2) = (reg1) then CF=0, ZF=1, SF=0**
CMP reg2, mem	**Modify flags ← (reg2) – (mem)** **If (reg2) > (mem) then CF=0, ZF=0, SF=0** **If (reg2) < (mem) then CF=1, ZF=0, SF=1** **If (reg2) = (mem) then CF=0, ZF=1, SF=0**
CMP mem, reg1	**Modify flags ← (mem) – (reg1)** **If (mem) > (reg1) then CF=0, ZF=0, SF=0** **If (mem) < (reg1) then CF=1, ZF=0, SF=1** **If (mem) = (reg1) then CF=0, ZF=1, SF=0**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

CMP reg/mem, data	
CMP reg, data	**Modify flags ← (reg) – (data)** **If (reg) > data then CF=0, ZF=0, SF=0** **If (reg) < data then CF=1, ZF=0, SF=1** **If (reg) = data then CF=0, ZF=1, SF=0**
CMP mem, data	**Modify flags ← (mem) – (mem)** **If (mem) > data then CF=0, ZF=0, SF=0** **If (mem) < data then CF=1, ZF=0, SF=1** **If (mem) = data then CF=0, ZF=1, SF=0**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

CMP A, data	
CMP AL, data8	**Modify flags ← (AL) – data8** **If (AL) > data8 then CF=0, ZF=0, SF=0** **If (AL) < data8 then CF=1, ZF=0, SF=1** **If (AL) = data8 then CF=0, ZF=1, SF=0**
CMP AX, data16	**Modify flags ← (AX) – data16** **If (AX) > data16 then CF=0, ZF=0, SF=0** **If (mem) < data16 then CF=1, ZF=0, SF=1** **If (mem) = data16 then CF=0, ZF=1, SF=0**

Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …

AND A, data AND AL, data8	(AL) ← (AL) & data8
AND AX, data16	(AX) ← (AX) & data16

AND reg/mem, data AND reg, data	(reg) ← (reg) & data
AND mem, data	(mem) ← (mem) & data

Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …

OR reg2/mem, reg1/mem OR reg2, reg1	(reg2) ← (reg2) | (reg1)
OR reg2, mem	(reg2) ← (reg2) | (mem)
OR mem, reg1	(mem) ← (mem) | (reg1)

OR reg/mem, data OR reg, data	(reg) ← (reg) | data
OR mem, data	(mem) ← (mem) | data

OR A, data OR AL, data8	(AL) ← (AL) | data8
OR AX, data16	(AX) ← (AX) | data16

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

XOR reg2/mem, reg1/mem	
XOR reg2, reg1	(reg2) ← (reg2) ^ (reg1)
XOR reg2, mem	(reg2) ← (reg2) ^ (mem)
XOR mem, reg1	(mem) ← (mem) ^ (reg1)

XOR reg/mem, data	
XOR reg, data	(reg) ← (reg) ^ data
XOR mem, data	(mem) ← (mem) ^ data

XOR A, data	
XOR AL, data8	(AL) ← (AL) ^ data8
XOR AX, data16	(AX) ← (AX) ^ data16

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

TEST reg2/mem, reg1/mem	
TEST reg2, reg1	Modify flags ← (reg2) & (reg1)
TEST reg2, mem	Modify flags ← (reg2) & (mem)
TEST mem, reg1	Modify flags ← (mem) & (reg1)

TEST reg/mem, data	
TEST reg, data	Modify flags ← (reg) & data
TEST mem, data	Modify flags ← (mem) & data

TEST A, data	
TEST AL, data8	Modify flags ← (AL) & data8
TEST AX, data16	Modify flags ← (AX) & data16

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

SHR reg/mem

SHR reg

i) SHR reg, 1

ii) SHR reg, CL

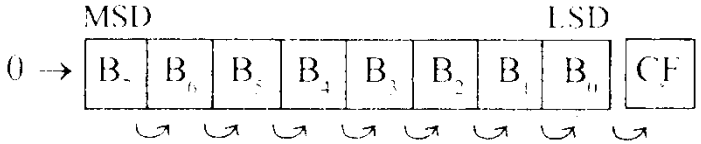
SHR mem

i) SHR mem, 1

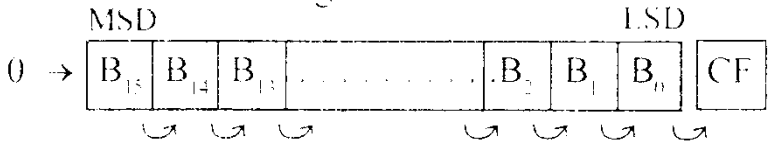
ii) SHR mem, CL

$$CF \leftarrow B_{LSD} ; B_n \leftarrow B_{n+1} ; B_{MSD} \leftarrow 0$$

reg 8 / mem 8



reg 16 / mem 16



Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

SHL reg/mem or SAL reg/mem

SHL reg or SAL reg

i) SHL reg, 1 or SAL reg, 1

ii) SHL reg, CL or SAL reg, CL

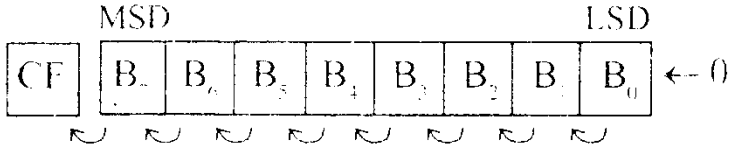
SHL mem or SAL mem

i) SHL mem, 1 or SAL mem, 1

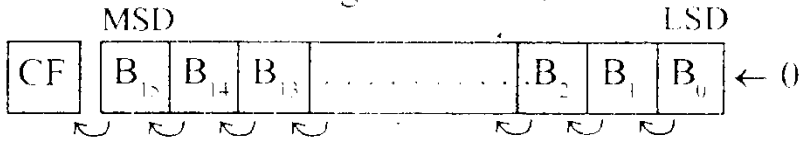
ii) SHL mem, CL or SAL mem, CL

$CF \leftarrow B_{MSD} ; B_{n+1} \leftarrow B_n ; B_{LSD} \leftarrow 0$

reg 8 / mem 8



reg 16 / mem 16



Instruction Set

3. Logical Instructions

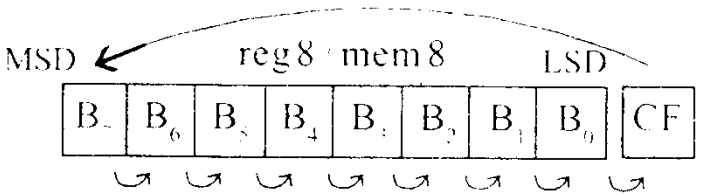
Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

RCR reg/mem

RCR reg

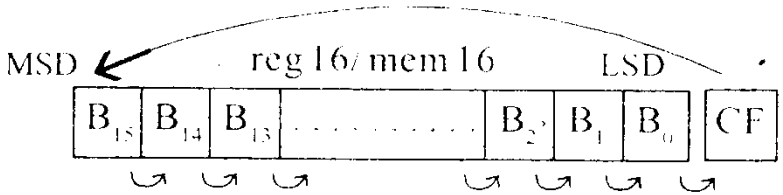
i) RCR reg, 1
ii) RCR reg, CL

$B_n \leftarrow B_{n-1}$; $B_{MSD} \leftarrow CF$; $CF \leftarrow B_{LSD}$



RCR mem

i) RCR mem, 1
ii) RCR mem, CL



Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...

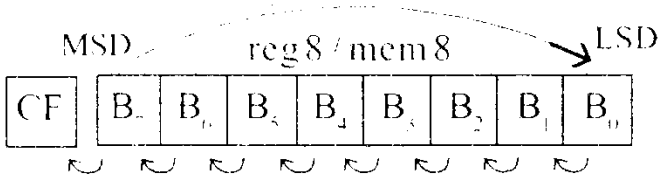
ROL reg/mem

ROL reg

i) ROL reg, 1

ii) ROL reg, CL

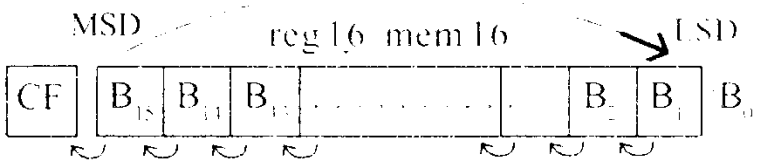
$B_{n+1} \leftarrow B_n : CF \leftarrow B_{MSD} ; B_{LSD} \leftarrow B_{MSD}$



ROL mem

i) ROL mem, 1

ii) ROL mem, CL



Instruction Set

4. String Manipulation Instructions

- String : Sequence of bytes or words
- 8086 instruction set includes instruction for string movement, comparison, scan, load and store.
- REP instruction prefix : used to repeat execution of string instructions
- String instructions end with S or SB or SW.
  S represents string, SB string byte and SW string word.
- Offset or effective address of the source operand is stored in SI register and that of the destination operand is stored in DI register.
- Depending on the status of DF, SI and DI registers are automatically updated.
- DF = 0 ⇒ SI and DI are incremented by 1 for byte and 2 for word.
- DF = 1 ⇒ SI and DI are decremented by 1 for byte and 2 for word.

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

REP **REPZ/ REPE** **(Repeat CMPS or SCAS until ZF = 0)**	**While CX ≠ 0 and ZF = 1, repeat execution of string instruction and (CX) ← (CX) – 1**
REPNZ/ REPNE **(Repeat CMPS or SCAS until ZF = 1)**	**While CX ≠ 0 and ZF = 0, repeat execution of string instruction and (CX) ← (CX) - 1**

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

MOVS	
MOVSB	**MA = (DS) x 16_{10} + (SI)** **MA_E = (ES) x 16_{10} + (DI)** **(MA_E) ← (MA)** **If DF = 0, then (DI) ← (DI) + 1; (SI) ← (SI) + 1** **If DF = 1, then (DI) ← (DI) - 1; (SI) ← (SI) - 1**
MOVSW	**MA = (DS) x 16_{10} + (SI)** **MA_E = (ES) x 16_{10} + (DI)** **(MA_E ; MA_E + 1) ← (MA; MA + 1)** **If DF = 0, then (DI) ← (DI) + 2; (SI) ← (SI) + 2** **If DF = 1, then (DI) ← (DI) - 2; (SI) ← (SI) - 2**

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

Compare two string byte or string word

CMPS	
CMPSB	**MA = (DS) x 16_{10} + (SI)** **MA_E = (ES) x 16_{10} + (DI)**
	Modify flags ← (MA) - (MA_E)
CMPSW	**If (MA) > (MA_E), then CF = 0; ZF = 0; SF = 0** **If (MA) < (MA_E), then CF = 1; ZF = 0; SF = 1** **If (MA) = (MA_E), then CF = 0; ZF = 1; SF = 0**
	<u>For byte operation</u> **If DF = 0, then (DI) ← (DI) + 1; (SI) ← (SI) + 1** **If DF = 1, then (DI) ← (DI) - 1; (SI) ← (SI) - 1**
	<u>For word operation</u> **If DF = 0, then (DI) ← (DI) + 2; (SI) ← (SI) + 2** **If DF = 1, then (DI) ← (DI) - 2; (SI) ← (SI) - 2**

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

Scan (compare) a string byte or word with accumulator

SCAS	

SCASB	$MA_E = (ES) \times 16_{10} + (DI)$
	Modify flags ← (AL) - (MA_E)
	
	If (AL) > (MA_E), then CF = 0; ZF = 0; SF = 0
	If (AL) < (MA_E), then CF = 1; ZF = 0; SF = 1
	If (AL) = (MA_E), then CF = 0; ZF = 1; SF = 0
	
	If DF = 0, then (DI) ← (DI) + 1
	If DF = 1, then (DI) ← (DI) – 1

SCASW	$MA_E = (ES) \times 16_{10} + (DI)$
	Modify flags ← (AL) - (MA_E)
	
	If (AX) > $(MA_E ; MA_E + 1)$, then CF = 0; ZF = 0; SF = 0
	If (AX) < $(MA_E ; MA_E + 1)$, then CF = 1; ZF = 0; SF = 1
	If (AX) = $(MA_E ; MA_E + 1)$, then CF = 0; ZF = 1; SF = 0
	
	If DF = 0, then (DI) ← (DI) + 2

Instruction Set

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Load string byte in to AL or string word in to AX

LODS	
LODSB	**MA = (DS) x 16_{10} + (SI)** **(AL) ← (MA)** **If DF = 0, then (SI) ← (SI) + 1** **If DF = 1, then (SI) ← (SI) – 1**
LODSW	**MA = (DS) x 16_{10} + (SI)** **(AX) ← (MA ; MA + 1)** **If DF = 0, then (SI) ← (SI) + 2** **If DF = 1, then (SI) ← (SI) – 2**

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

Store byte from AL or word from AX in to string

STOS	
STOSB	**MA_E = (ES) x 16_{10} + (DI)** **(MA_E) ← (AL)** **If DF = 0, then (DI) ← (DI) + 1** **If DF = 1, then (DI) ← (DI) – 1**
STOSW	**MA_E = (ES) x 16_{10} + (DI)** **(MA_E ; MA_E + 1) ← (AX)** **If DF = 0, then (DI) ← (DI) + 2** **If DF = 1, then (DI) ← (DI) – 2**

Instruction Set

5. Processor Control Instructions

Mnemonics	Explanation
STC	Set CF ← 1
CLC	Clear CF ← 0
CMC	Complement carry CF ← CF/
STD	Set direction flag DF ← 1
CLD	Clear direction flag DF ← 0
STI	Set interrupt enable flag IF ← 1
CLI	Clear interrupt enable flag IF ← 0
NOP	No operation
HLT	Halt after interrupt is set
WAIT	Wait for TEST pin active
ESC opcode mem/ reg	Used to pass instruction to a coprocessor which shares the address and data bus with the 8086
LOCK	Lock bus during next instruction

Instruction Set

6. Control Transfer Instructions

- Transfer the control to a specific destination or target instruction
- Do not affect flags

- 8086 Unconditional transfers

Mnemonics	Explanation
CALL reg/ mem/ disp16	Call subroutine
RET	Return from subroutine
JMP reg/ mem/ disp8/ disp16	Unconditional jump

Instruction Set

6. Control Transfer Instructions

- 8086 signed conditional branch instructions
- 8086 unsigned conditional branch instructions

- Checks flags
- If conditions are true, the program control is transferred to the new memory location in the same segment by modifying the content of IP

Instruction Set

6. Control Transfer Instructions

- 8086 signed conditional branch instructions

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JG disp8 Jump if greater	JNLE disp8 Jump if not less or equal
JGE disp8 Jump if greater than or equal	JNL disp8 Jump if not less
JL disp8 Jump if less than	JNGE disp8 Jump if not greater than or equal
JLE disp8 Jump if less than or equal	JNG disp8 Jump if not greater

- 8086 unsigned conditional branch instructions

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JA disp8 Jump if above	JNBE disp8 Jump if not below or equal
JAE disp8 Jump if above or equal	JNB disp8 Jump if not below
JB disp8 Jump if below	JNAE disp8 Jump if not above or equal
JBE disp8 Jump if below or equal	JNA disp8 Jump if not above

Instruction Set

6. Control Transfer Instructions

- 8086 conditional branch instructions affecting individual flags

Mnemonics	Explanation
JC disp8	Jump if CF = 1
JNC disp8	Jump if CF = 0
JP disp8	Jump if PF = 1
JNP disp8	Jump if PF = 0
JO disp8	Jump if OF = 1
JNO disp8	Jump if OF = 0
JS disp8	Jump if SF = 1
JNS disp8	Jump if SF = 0
JZ disp8	Jump if result is zero, i.e, Z = 1
JNZ disp8	Jump if result is not zero, i.e, Z = 1

Assembler directives

Assemble Directives

- Instructions to the Assembler regarding the program being executed.

- Control the generation of machine codes and organization of the program; but no machine codes are generated for assembler directives.

- Also called ‘pseudo instructions’

- Used to :
  - specify the start and end of a program
  - attach value to variables
  - allocate storage locations to input/ output data
  - define start and end of segments, procedures, macros etc..

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **Define Byte**
- **Define a byte type (8-bit) variable**
- **Reserves specific amount of memory locations to each variable**
- **Range : 00_H – FF_H for unsigned value; 00_H – $7F_H$ for positive value and 80_H – FF_H for negative value**
- **General form : variable DB value/ values**

Example:

```
LIST DB 7FH, 42H, 35H
```

Three consecutive memory locations are reserved for the variable LIST and each data specified in the instruction are stored as initial value in the reserved memory location

Assemble Directives

- **Define Word**
- **Define a word type (16-bit) variable**
- **Reserves two consecutive memory locations to each variable**
- **Range : 0000_H – $FFFF_H$ for unsigned value;**
  0000_H – $7FFF_H$ for positive value and
  8000_H – $FFFF_H$ for negative value
- **General form : variable DW value/ values**

Example:

```
ALIST DW 6512H, 0F251H, 0CDE2H
```

Six consecutive memory locations are reserved for the variable ALIST and each 16-bit data specified in the instruction is stored in two consecutive memory location.

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **SEGMENT : Used to indicate the beginning of a code/ data/ stack segment**
- **ENDS : Used to indicate the end of a code/ data/ stack segment**
- **General form:**

```
Segnam SEGMENT
   ...
   ...
   ...
   ...
   ...
   ...
Segnam ENDS
```

Program code
or
Data Defining Statements

User defined name of the segment

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

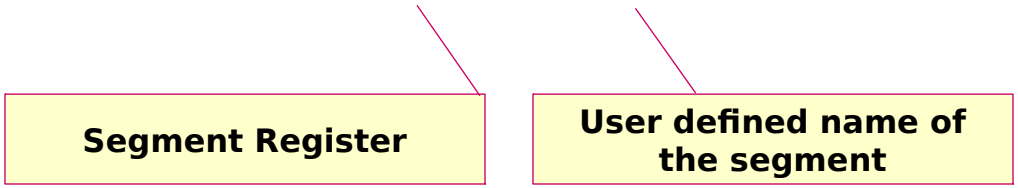
PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- Informs the assembler the name of the program/ data segment that should be used for a specific segment.
- General form:

ASSUME segreg : segnam, .. , segreg : segnam



Example:

ASSUME CS: ACODE, DS:ADATA	Tells the compiler that the instructions of the program are stored in the segment ACODE and data are stored in the segment ADATA

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **ORG** (Origin) is used to assign the starting address (Effective address) for a program/ data segment
- **END** is used to terminate a program; statements after END will be ignored
- **EVEN** : Informs the assembler to store program/ data segment starting from an even address
- **EQU** (Equate) is used to attach a value to a variable

Examples:

ORG 1000H	Informs the assembler that the statements following ORG 1000H should be stored in memory starting with effective address 1000_H
LOOP EQU 10FEH	Value of variable LOOP is $10FE_H$
_SDATA SEGMENT ORG 1200H A DB 4CH EVEN B DW 1052H _SDATA ENDS	In this data segment, effective address of memory location assigned to A will be 1200_H and that of B will be 1202_H and 1203_H.

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **PROC** Indicates the beginning of a procedure
- **ENDP** End of procedure
- **FAR** Intersegment call
- **NEAR** Intrasegment call
- General form

```
procname PROC[NEAR/ FAR]
        ...
        ...
        ...
        RET

procname ENDP
```

Program statements of the procedure

Last statement of the procedure

User defined name of the procedure

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

Examples:

```
ADD64 PROC NEAR
        ...
        ...
        ...
        RET
ADD64 ENDP
```

The subroutine/ procedure named ADD64 is declared as NEAR and so the assembler will code the CALL and RET instructions involved in this procedure as near call and return

```
CONVERT PROC FAR
        ...
        ...
        ...
        RET
CONVERT ENDP
```

The subroutine/ procedure named CONVERT is declared as FAR and so the assembler will code the CALL and RET instructions involved in this procedure as far call and return

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **Reserves one memory location for 8-bit signed displacement in jump instructions**

Example:

JMP SHORT AHEAD	**The directive will reserve one memory location for 8-bit displacement named AHEAD**

Assemble Directives

- **MACRO** Indicate the beginning of a macro
- **ENDM** End of a macro
- General form:

```
macroname MACRO[Arg1, Arg2 ...]
          ...
          ...
          ...
macroname ENDM
```

Program statements in the macro

User defined name of the macro