The POP Instruction

- POP D
  - Copy the contents of the memory location pointed to by the SP to register E
  - Increment SP
  - Copy the contents of the memory location pointed to by the SP to register D
  - Increment SP

Operation of the Stack

- During pushing, the stack operates in a "decrement then store" style.
  - The stack pointer is decremented first, then the information is placed on the stack.
- During poping, the stack operates in a "use then increment" style.
  - The information is retrieved from the top of the the stack and then the pointer is incremented.
- The SP pointer always points to "the top of the stack".

LIFO

- The order of PUSHs and POPs must be opposite of each other in order to retrieve information back into its original location.

```
PUSH B
PUSH D
...
POP D
POP B
```

The PSW Register Pair

- The 8085 recognizes one additional register pair called the PSW (Program Status Word).
  - This register pair is made up of the Accumulator and the Flags registers.
- It is possible to push the PSW onto the stack, do whatever operations are needed, then POP it off of the stack.
  - The result is that the contents of the Accumulator and the status of the Flags are returned to what they were before the operations were executed.

Subroutines

- A subroutine is a group of instructions that will be used repeatedly in different locations of the program.
  - Rather than repeat the same instructions several times, they can be grouped into a subroutine that is called from the different locations.
- In Assembly language, a subroutine can exist anywhere in the code.
  - However, it is customary to place subroutines separately from the main program.

Subroutines

- The 8085 has two instructions for dealing with subroutines.
  - The CALL instruction is used to redirect program execution to the subroutine.
  - The RET instruction is used to return the execution to the calling routine.

The CALL Instruction

- CALL 4000H
  - Push the address of the instruction immediately following the CALL onto the stack
  - Load 4000H to PC

The RET Instruction

- RET
  - Retrieve the return address from the top of the stack
  - Load the program counter with the return address

Cautions

- The CALL instruction places the return address at the two memory locations immediately before where the Stack Pointer is pointing.
  - You must set the SP correctly BEFORE using the CALL instruction.
- The RET instruction takes the contents of the two memory locations at the top of the stack and uses these as the return address.
  - Do not modify the stack pointer in a subroutine. You will loose the return address.

Passing Data to a Subroutine

- In Assembly Language data is passed to a subroutine through registers.
  - The data is stored in one of the registers by the calling program and the subroutine uses the value from the register.
- The other possibility is to use agreed upon memory locations.
  - The calling program stores the data in the memory location and the subroutine retrieves the data from the location and uses it.

Call by Reference and Call by Value

- If the subroutine performs operations on the contents of the registers, then these modifications will be transferred back to the calling program upon returning from a subroutine.
  - Call by reference
- If this is not desired, the subroutine should PUSH all the registers it needs on the stack on entry and POP them on return.
  - The original values are restored before execution returns to the calling program.

Cautions with PUSH and POP

- PUSH and POP should be used in opposite order.
- There has to be as many POP's as there are PUSH's.
  - –If not, the RET statement will pick up the wrong information from the top of the stack and the program will fail.
- It is not advisable to place PUSH or POP inside a loop.

Conditional CALL and RET Instructions

- The 8085 supports conditional CALL and conditional RET instructions.
  - The same conditions used with conditional JUMP instructions can be used.
  - CC, call subroutine if Carry flag is set.
  - CNC, call subroutine if Carry flag is not set
  - RC, return from subroutine if Carry flag is set
  - RNC, return from subroutine if Carry flag is not set
  - Etc.

A Proper Subroutine

- According to Software Engineering practices, a proper subroutine:
  - Is only entered with a CALL and exited with an RET
  - Has a single entry point
    - Do not use a CALL statement to jump into different points of the same subroutine.
  - Has a single exit point
    - There should be one return statement from any subroutine.
- Following these rules, there should not be any confusion with PUSH and POP usage.

Interrupts

- Interrupt is a process where an external device can get the attention of the microprocessor.
  - The process starts from the I/O device
  - The process is asynchronous.
- Interrupts can be classified into two types:
  - Maskable (can be delayed)
  - Non-Maskable (can not be delayed)
- Interrupts can also be classified into:
  - Vectored (the address of the service routine is hard-wired)
  - Non-vectored (the address of the service routine needs to be supplied externally)

Interrupts

- An interrupt is considered to be an emergency signal.
  - The Microprocessor should respond to it as soon as possible.
- When the Microprocessor receives an interrupt signal, it suspends the currently executing program and jumps to an Interrupt Service Routine (ISR) to respond to the incoming interrupt.
  - Each interrupt will most probably have its own ISR.

Responding to Interrupts

- Responding to an interrupt may be immediate or delayed depending on whether the interrupt is maskable or non-maskable and whether interrupts are being masked or not.
- There are two ways of redirecting the execution to the ISR depending on whether the interrupt is vectored or non-vectored.
  - The vector is already known to the Microprocessor
  - The device will have to supply the vector to the Microprocessor

The 8085 Interrupts

- The maskable interrupt process in the 8085 is controlled by a single flip flop inside the microprocessor. This Interrupt Enable flip flop is controlled using the two instructions “EI” and “DI”.
- The 8085 has a single Non-Maskable interrupt.
  - The non-maskable interrupt is not affected by the value of the Interrupt Enable flip flop.

The 8085 Interrupts

- The 8085 has 5 interrupt inputs.
  - The INTR input.
    - The INTR input is the only non-vectored interrupt.
    - INTR is maskable using the EI/DI instruction pair.
  - RST 5.5, RST 6.5, RST 7.5 are all automatically vectored.
    - RST 5.5, RST 6.5, and RST 7.5 are all maskable.
  - TRAP is the only non-maskable interrupt in the 8085
    - TRAP is also automatically vectored

The 8085 Interrupts

Interrupt name	Maskable	Vectored
INTR	Yes	No
RST 5.5	Yes	Yes
RST 6.5	Yes	Yes
RST 7.5	Yes	Yes
TRAP	No	Yes

Interrupt Vectors and the Vector Table

- 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).
  - The purpose of the IVT is to hold the vectors that redirect the microprocessor to the right place when an interrupt arrives.
  - The IVT is divided into several blocks. Each block is used by one of the interrupts to hold its "vector"

The 8085 Non-Vectored Interrupt Process

1. The interrupt process should be enabled using the EI instruction.
2. The 8085 checks for an interrupt during the execution of every instruction.
3. If there is an interrupt, the microprocessor will complete the executing instruction, and start a RESTART sequence.
4. The RESTART sequence resets the interrupt flip flop and activates the interrupt acknowledge signal (INTA).
5. Upon receiving the INTA signal, the interrupting device is expected to return the op-code of one of the 8 RST instructions.

The 8085 Non-Vectored Interrupt Process

6. When the microprocessor executes the RST instruction received from the device, it saves the address of the next instruction on the stack and jumps to the appropriate entry in the IVT.
7. The IVT entry must redirect the microprocessor to the actual service routine.
8. The service routine must include the instruction EI to re-enable the interrupt process.
9. At the end of the service routine, the RET instruction returns the execution to where the program was interrupted.

The 8085 Non-Vectored Interrupt Process

- The 8085 recognizes 8 RESTART instructions: RST0 - RST7.
  - each of these would send the execution to a predetermined hard-wired memory location:

Restart Instruction	Equivalent to
RST0	CALL 0000H
RST1	CALL 0008H
RST2	CALL 0010H
RST3	CALL 0018H
RST4	CALL 0020H
RST5	CALL 0028H
RST6	CALL 0030H
RST7	CALL 0038H

Restart Sequence

- The restart sequence is made up of three machine cycles
  - In the 1st machine cycle:
    - The microprocessor sends the INTA signal.
    - While INTA is active the microprocessor reads the data lines expecting to receive, from the interrupting device, the opcode for the specific RST instruction.
  - In the 2nd and 3rd machine cycles:
    - the 16-bit address of the next instruction is saved on the stack.
    - Then the microprocessor jumps to the address associated with the specified RST instruction.

Restart Sequence

- The location in the IVT associated with the RST instruction can not hold the complete service routine.
  - The routine is written somewhere else in memory.
  - Only a JUMP instruction to the ISR's location is kept in the IVT block.

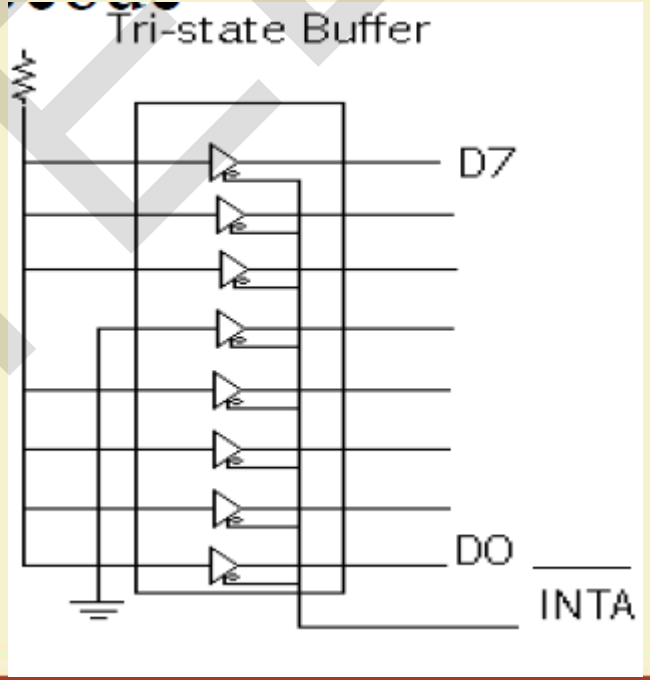
Hardware Generation of RST Opcode

- How does the external device produce the opcode for the appropriate RST instruction?
  - The opcode is simply a collection of bits.
  - So, the device needs to set the bits of the data bus to the appropriate value in response to an INTA signal.

Hardware Generation of RST Opcode

RST 5's opcode is EF =

D							D
7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1



Hardware Generation of RST Opcode

- During the interrupt acknowledge machine cycle, (the 1st machine cycle of the RST operation):
  - The Microprocessor activates the INTA signal.
  - This signal will enable the Tri-state buffers, which will place the value EFH on the data bus.
  - Therefore, sending the Microprocessor the RST 5 instruction.

- The RST 5 instruction is exactly equivalent to CALL 0028H

Issues in Implementing INTR Interrupts

- How long must INTR remain high?
  - The microprocessor checks the INTR line one clock cycle before the last T-state of each instruction.
  - The interrupt process is Asynchronous.
  - The INTR must remain active long enough to allow for the longest instruction.
  - The longest instruction for the 8085 is the conditional CALL instruction which requires 18 T-states.

Therefore, the INTR must remain active for 17.5 T-states.

Issues in Implementing INTR Interrupts

- How long can the INTR remain high?
  - The INTR line must be deactivated before the EI is executed. Otherwise, the microprocessor will be interrupted again.
  - The worst case situation is when EI is the first instruction in the ISR.
  - Once the microprocessor starts to respond to an INTR interrupt, INTA becomes active (=0).

Therefore, INTR should be turned off as soon as the INTA signal is received.

Issues in Implementing INTR Interrupts

- Can the microprocessor be interrupted again before the completion of the ISR?
  - As soon as the 1st interrupt arrives, all maskable interrupts are disabled.
  - They will only be enabled after the execution of the EI instruction.

Therefore, the answer is: "only if you allow it to".

If the EI instruction is placed early in the ISR, other interrupt may occur before the ISR is done.

Multiple Interrupts & Priorities

- How do we allow multiple devices to interrupt using the INTR line?
  - The microprocessor can only respond to one signal on INTR at a time.
  - Therefore, we must allow the signal from only one of the devices to reach the microprocessor.
  - We must assign some priority to the different devices and allow their signals to reach the microprocessor according to the priority.

The Priority Encoder

The solution is to use a circuit called the priority encoder (74366).

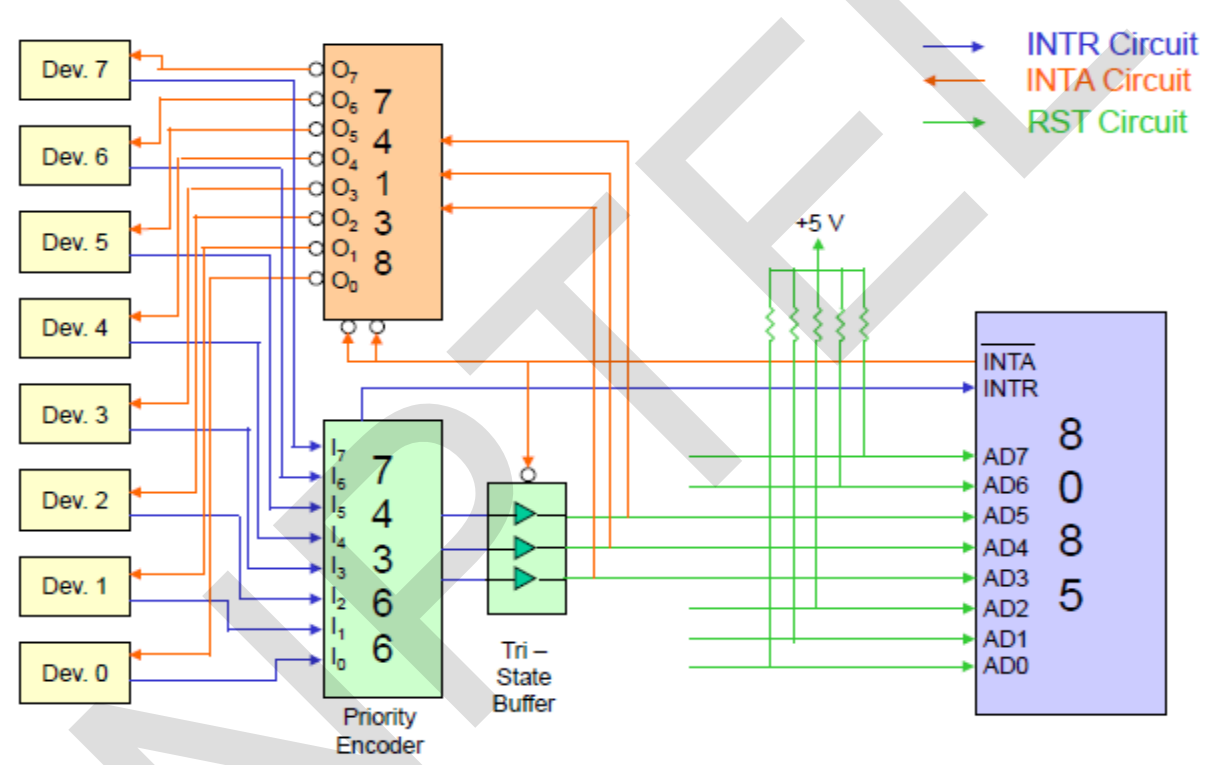
- This circuit has 8 inputs and 3 outputs.
- The inputs are assigned increasing priorities according to the increasing index of the input.
  - Input 7 has highest priority and input 0 has the lowest.
- The 3 outputs carry the index of the highest priority active input.

Multiple Interrupts & Priorities

- Note that the opcodes for the different RST instructions follow a set pattern.
  - Bit D5, D4 and D3 of the opcodes change in a binary sequence from RST 7 down to RST 0.
  - The other bits are always 1.
  - This allows the code generated by the 74366 to be used directly to choose the appropriate RST instruction.

- The one draw back to this scheme is that the only way to change the priority of the devices connected to the 74366 is to reconnect the hardware.

Multiple Interrupts and Priority



The 8085 Maskable/Vectored Interrupts

The 8085 has 4 Masked/Vectored interrupt inputs.

- RST 5.5, RST 6.5, RST 7.5
  - They are all maskable.
  - They are automatically vectored according to the following table:

Interrupt	Vector
RST 5.5	002CH
RST 6.5	0034H
RST 7.5	003CH

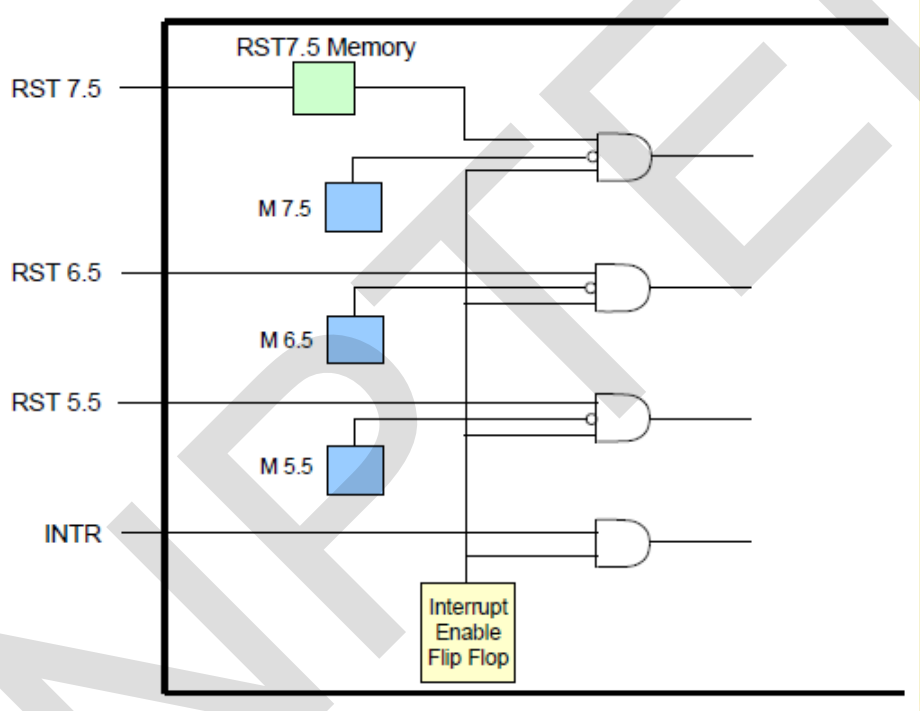
  - The vectors for these interrupt fall in between the vectors for the RST instructions. That's why they have names like RST 5.5 (RST 5 and a half).

Masking RST 5.5, RST 6.5 and RST 7.5

These three interrupts are masked at two levels:

- Through the Interrupt Enable flip flop and the EI/DI instructions.
  - The Interrupt Enable flip flop controls the whole maskable interrupt process.
- Through individual mask flip flops that control the availability of the individual interrupts.
  - These flip flops control the interrupts individually.

Maskable Interrupts



The 8085 Maskable/Vectored Interrupt Process

1. The interrupt process should be enabled using the EI instruction.
2. The 8085 checks for an interrupt during the execution of every instruction.
3. 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.
4. The microprocessor then executes a call instruction that sends the execution to the appropriate location in the interrupt vector table.

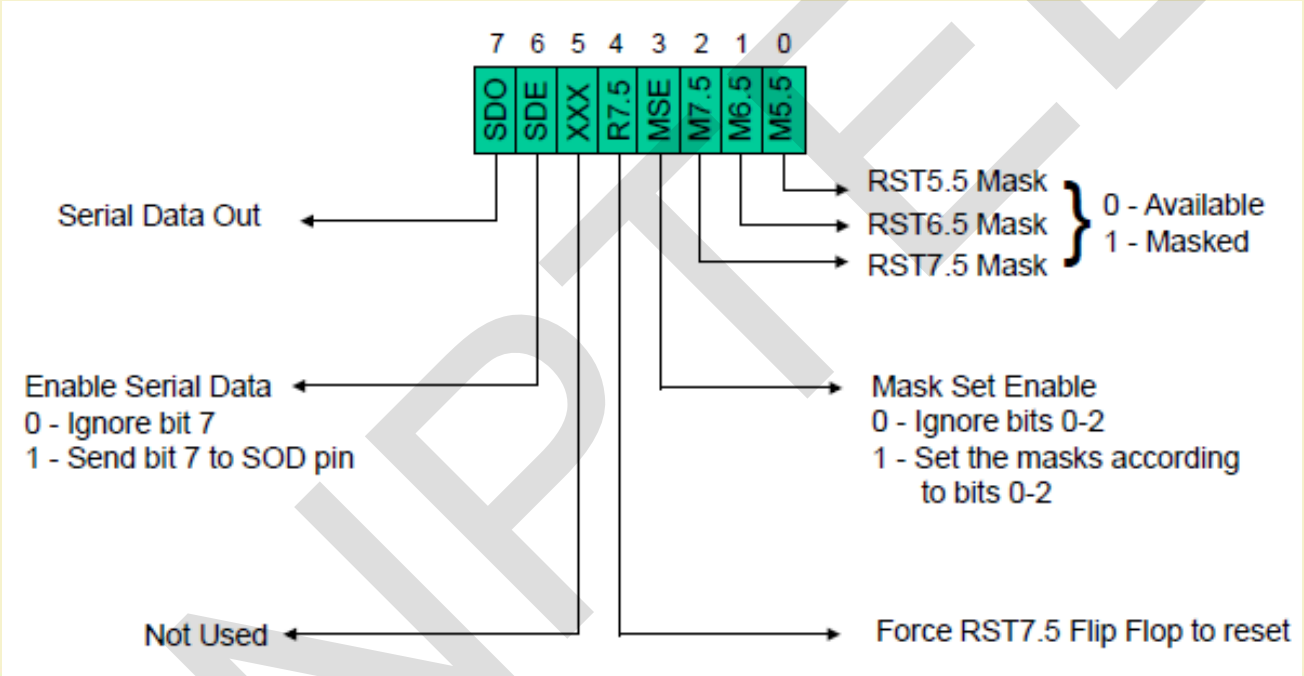
The 8085 Maskable/Vectored Interrupt Process

5. When the microprocessor executes the call instruction, it saves the address of the next instruction on the stack.
6. The microprocessor jumps to the specific service routine.
7. The service routine must include the instruction EI to re-enable the interrupt process.
8. At the end of the service routine, the RET instruction returns the execution to where the program was interrupted.

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.
  - This 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



SIM and the Interrupt Mask

- Bit 0 is the mask for RST 5.5, bit 1 is the mask for RST 6.5 and bit 2 is the mask for RST 7.5.
  - If the mask bit is 0, the interrupt is available.
  - If the mask bit is 1, the interrupt is masked.

- Bit 3 (Mask Set Enable - MSE) is an enable for setting the mask.
  - If it is set to 0 the mask is ignored and the old settings remain.
  - If it is set to 1, the new setting are applied.
  - The SIM instruction is used for multiple purposes and not only for setting interrupt masks.
    - **It is also used to control functionality such as Serial Data Transmission.**
    - **Therefore, bit 3 is necessary to tell the microprocessor whether or not the interrupt masks should be modified**

SIM and the Interrupt Mask

- The RST 7.5 interrupt is the only 8085 interrupt that has memory.
  - If a signal on RST7.5 arrives while it is masked, a flip flop will remember the signal.
  - When RST7.5 is unmasked, the microprocessor will be interrupted even if the device has removed the interrupt signal.
  - This flip flop will be automatically reset when the microprocessor responds to an RST 7.5 interrupt.
- Bit 4 of the accumulator in the SIM instruction allows explicitly resetting the RST 7.5 memory even if the microprocessor did not respond to it.

SIM and the Interrupt Mask

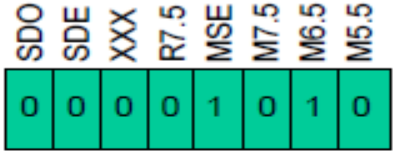
- The SIM instruction can also be used to perform serial data transmission out of the 8085's SOD pin.
  - One bit at a time can be sent out serially over the SOD pin.
- Bit 6 is used to tell the microprocessor whether or not to perform serial data transmission
  - If 0, then do not perform serial data transmission
  - If 1, then do.
- The value to be sent out on SOD has to be placed in bit 7 of the accumulator.
- Bit 5 is not used by the SIM instruction

Using SIM Instruction to Modify Interrupt Masks

Example: Set the interrupt masks so that RST5.5 is enabled, RST6.5 is masked, and RST7.5 is enabled.

– First, determine the contents of the accumulator

- Enable 5.5	bit 0 = 0
- Disable 6.5	bit 1 = 1
- Enable 7.5	bit 2 = 0
- Allow setting the masks	bit 3 = 1
- Don't reset the flip flop	bit 4 = 0
- Bit 5 is not used	bit 5 = 0
- Don't use serial data	bit 6 = 0
- Serial data is ignored	bit 7 = 0



Contents of accumulator are: 0AH

```
EI          ; Enable interrupts including INTR
MVI A, 0A   ; Prepare the mask to enable RST 7.5, and 5.5, disable 6.5
SIM         ; Apply the settings RST masks
```

Triggering Levels

RST 7.5 is positive edge sensitive.

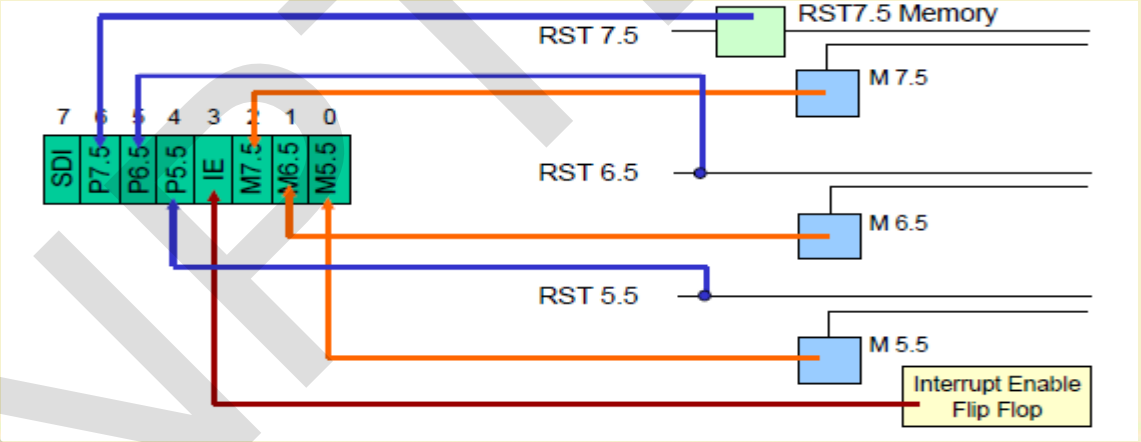
- When a positive edge appears on the RST7.5 line, a logic 1 is stored in the flip-flop as a "pending" interrupt.
- Since the value has been stored in the flip flop, the line does not have to be high when the microprocessor checks for the interrupt to be recognized.
- The line must go to zero and back to one before a new interruptis recognized.

RST 6.5 and RST 5.5 are level sensitive.

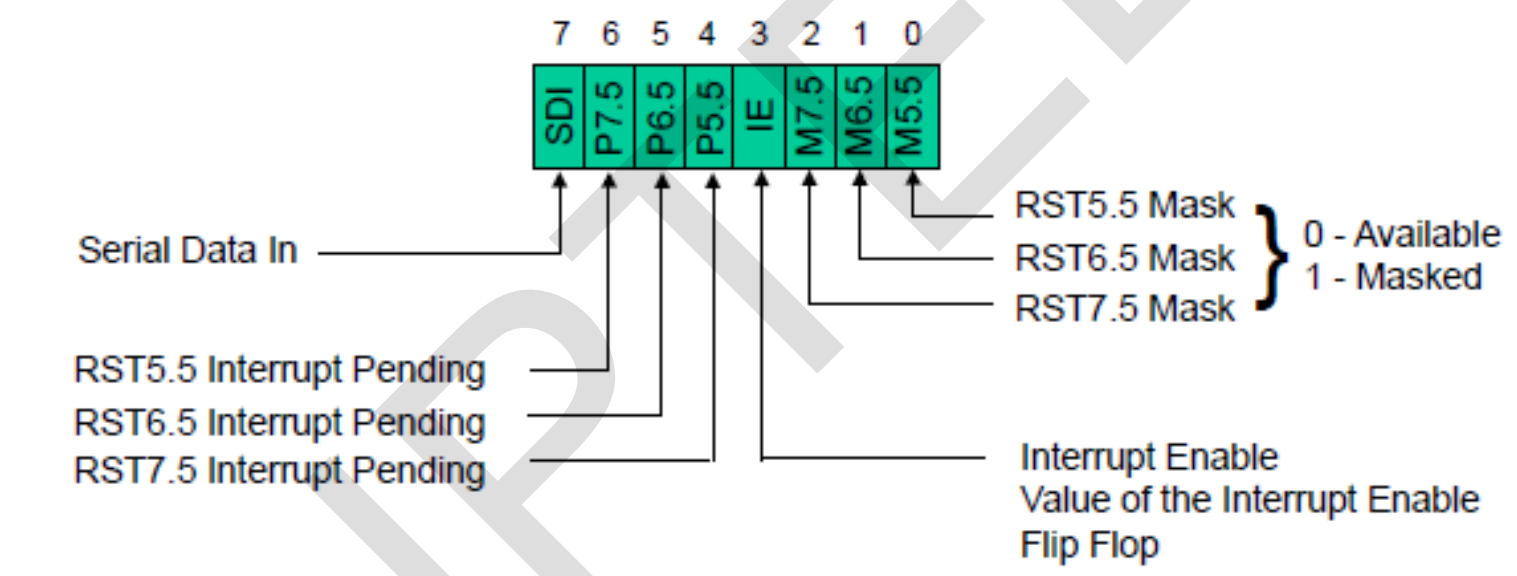
- The interrupting signal must remain present until the microprocessor checks for interrupts.

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.



How RIM sets the Accumulator's different bits



The RIM Instruction and the Masks

- Bits 0-2 show the current setting of the mask for each of RST 7.5, RST 6.5 and RST 5.5
  - They return the contents of the three mask flip flops.
  - They can be used by a program to read the mask settings in order to modify only the right mask.
- Bit 3 shows whether the maskable interrupt process is enabled or not.
  - It returns the contents of the Interrupt Enable Flip Flop.
  - It can be used by a program to determine whether or not interrupts are enabled.

The RIM Instruction and the Masks

- Bits 4-6 show whether or not there are pending interrupts on RST 7.5, RST 6.5, and RST 5.5
  - Bits 4 and 5 return the current value of the RST5.5 and RST6.5 pins.
  - Bit 6 returns the current value of the RST7.5 memory flip flop.
- Bit 7 is used for Serial Data Input.
  - The RIM instruction reads the value of the SID pin on the microprocessor and returns it in this bit.

Pending Interrupts

- Since the 8085 has five interrupt lines, interrupts may occur during an ISR and remain pending.
  - Using the RIM instruction, the programmer can read the status of the interrupt lines and find if there are any pending interrupts.
- The advantage is being able to find about interrupts on RST 7.5, RST 6.5, and RST 5.5 without having to enable low level interrupts like INTR.

Using RIM and SIM to set Individual Masks

Example: Set the mask to enable RST6.5 without modifying the masks for RST5.5 and RST7.5.

- In order to do this correctly, we need to use the RIM instruction to find the current settings of the RST5.5 and RST7.5 masks.
- Then we can use the SIM instruction to set the masks using this information.
- Given that both RIM and SIM use the Accumulator, we can use some logical operations to masks the un-needed values returned by RIM and turn them into the values needed by SIM.

Using RIM and SIM to set Individual Masks

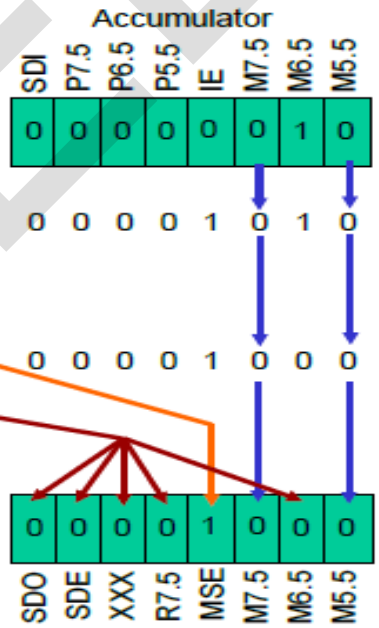
– Assume the RST5.5 and RST7.5 are enabled and the interrupt process is disabled.

```
RIM          ; Read the current settings.

ORI 08H      ; 0 0 0 0 1 0 0 0
             ; Set bit 4 for MSE.

ANI 0DH      ; 0 0 0 0 1 1 0 1
             ; Turn off Serial Data, Don't reset
             ; RST7.5 flip flop, and set the mask
             ; for RST6.5 off. Don't cares are
             ; assumed to be 0.

SIM          ; Apply the settings.
```



TRAP

- TRAP is the only non-maskable interrupt.
  - It does not need to be enabled because it cannot be disabled.
- It has the highest priority amongst interrupts.
- It is edge and level sensitive.
  - It needs to be high and stay high to be recognized.
  - Once it is recognized, it won't be recognized again until it goes low, then high again.
- TRAP is usually used for power failure and emergency shutoff.

Internal Interrupt Priority

- Internally, the 8085 implements an interrupt priority scheme.
  - The interrupts are ordered as follows:
    - TRAP
    - RST 7.5
    - RST 6.5
    - RST 5.5
    - INTR
  - However, TRAP has lower priority than the HLD signal used for DMA.

The 8085 Interrupts

Interrupt Name	Maskable	Masking Method	Vectored	Memory	Triggering Method
INTR	Yes	DI / EI	No	No	Level Sensitive
RST 5.5 / RST 6.5	Yes	DI / EI SIM	Yes	No	Level Sensitive
RST 7.5	Yes	DI / EI SIM	Yes	Yes	Edge Sensitive
TRAP	No	None	Yes	No	Level & Edge Sensitive

Direct Memory Access

This is a process where data is transferred between two peripherals directly without the involvement of the microprocessor.

– This process employs the HOLD pin on the microprocessor

The external DMA controller sends a signal on the HOLD pin to the microprocessor.

- The microprocessor completes the current operation and sends a signal on HLDA and stops using the buses.
- Once the DMA controller is done, it turns off the HOLD signal and the microprocessor takes back control of the buses.

Basic Concepts in Serial I/O

- Interfacing requirements:
  - Identify the device through a port number.
    - Memory-mapped.
    - Peripheral-mapped.
  - Enable the device using the Read and Write control signals.

    Read for an input device.

    Write for an output device.
  - Only one data line is used to transfer the information instead of the entire data bus.

Basic Concepts in Serial I/O

- Controlling the transfer of data:
  - Microprocessor control.
    - Unconditional, polling, status check, etc.
  - Device control.
    - Interrupt.

Synchronous Data Transmission

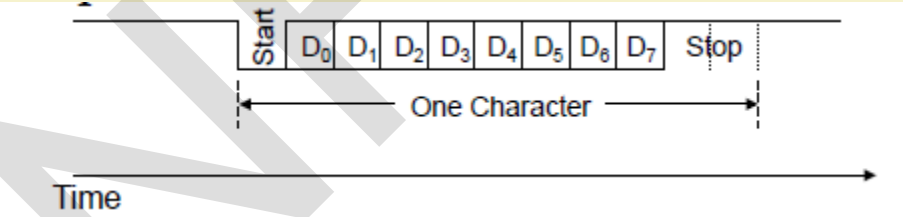
- The transmitter and receiver are synchronized.
  - A sequence of synchronization signals is sent before the communication begins.
- Usually used for high speed transmission.
  - More than 20 K bits/sec.
- Message based.
  - Synchronization occurs at the beginning of a long message.

Asynchronous Data Transmission

- Transmission occurs at any time.
- Character based.
  - Each character is sent separately.
- Generally used for low speed transmission.
  - Less the 20 K bits/sec.

Asynchronous Data Transmission

- Follows agreed upon standards:
  - The line is normally at logic one (mark).
    - Logic 0 is known as space.
  - The transmission begins with a start bit (low).
  - Then the seven or eight bits representing the character are transmitted.
  - The transmission is concluded with one or two stop bits.



Simplex and Duplex Transmission

- Simplex.
  - One-way transmission.
  - Only one wire is needed to connect the two devices
  - Like communication from computer to a printer.
- Half-Duplex.
  - Two-way transmission but one way at a time.
  - One wire is sufficient.
- Full-Duplex.
  - Data flows both ways at the same time.
  - Two wires are needed.
  - Like transmission between two computers.

Rate of Transmission

- For parallel transmission, all of the bits are sent at once.
- For serial transmission, the bits are sent one at a time.
  - Therefore, there needs to be agreement on how "long" each bit stays on the line.
- The rate of transmission is usually measured in bits/second or baud.

Length of Each Bit

- Given a certain baud rate, how long should each bit last?
  - Baud = bits / second.
  - Seconds / bits = 1 /baud.
  - At 1200 baud, a bit lasts 1/1200 = 0.83 m Sec.

Transmitting a Character

- To send the character A over a serial communication line at a baud rate of 56.6 K:
  - ASCII for A is 41H = 01000001.
  - Must add a start bit and two stop bits:
    - 11 01000001 0
  - Each bit should last 1/56.6K = 17.66 μ Sec. Known as bit time.
  - Set up a delay loop for 17.66 μ Sec and set the transmission line to the different bits for the duration of the loop.

Error Checking

- Various types of errors may occur during transmission.
  - To allow checking for these errors, additional information is transmitted with the data.
- Error checking techniques:
  - Parity Checking.
  - Checksum.
- These techniques are for error checking not correction.
  - They only indicate that an error has occurred.
  - They do not indicate where or what the correct information is

Parity Checking

Make the number of 1's in the data Odd or Even.

- Given that ASCII is a 7-bit code, bit D the parity information.
- Even Parity

  The transmitter counts the number of ones in the data. If there is an odd number of 1's, bit D is set to 1 to make the total number of 1's even.

  • The receiver calculates the parity of the received message, it should match bit D .
    - If it doesn't match, there was an error in the transmission.

Checksum

Used when larger blocks of data are being transmitted.

The transmitter adds all of the bytes in the message without carries. It then calculates the 2's complement of the result and send that as the last byte.

The receiver adds all of the bytes in the message including the last byte. The result should be 0.

RS 232

A communication standard for connecting computers to printers, modems, etc.

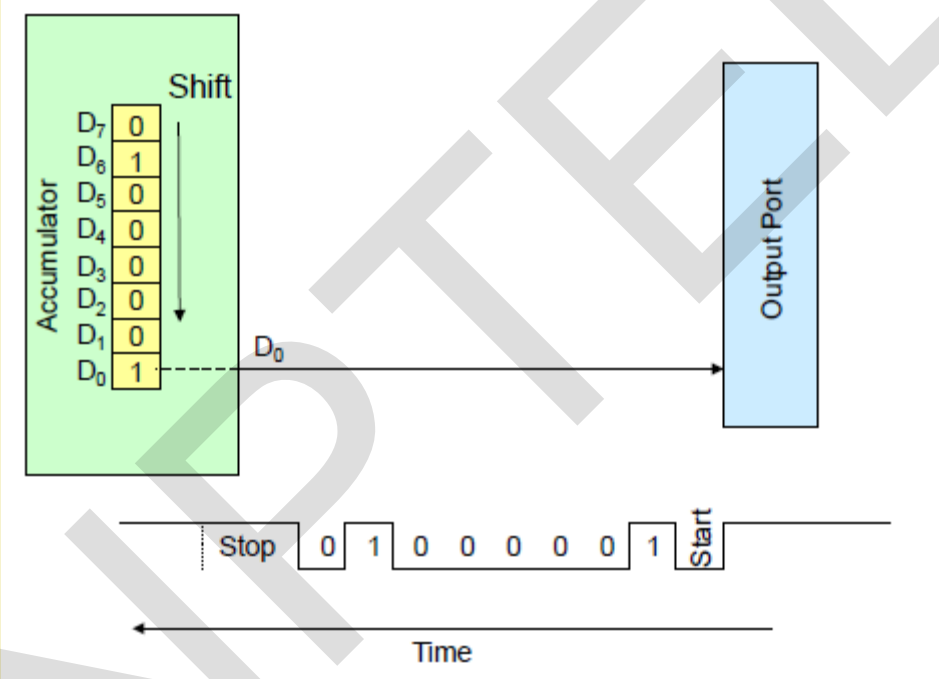
- The most common communication standard.
- Defined in the 1950's.
- It uses voltages between +15 and –15 V.
- Restricted to speeds less than 20 K baud.
- Restricted to distances of less than 50 feet (15 m).

The original standard uses 25 wires to connect the two devices. However, in reality only three of these wires are needed.

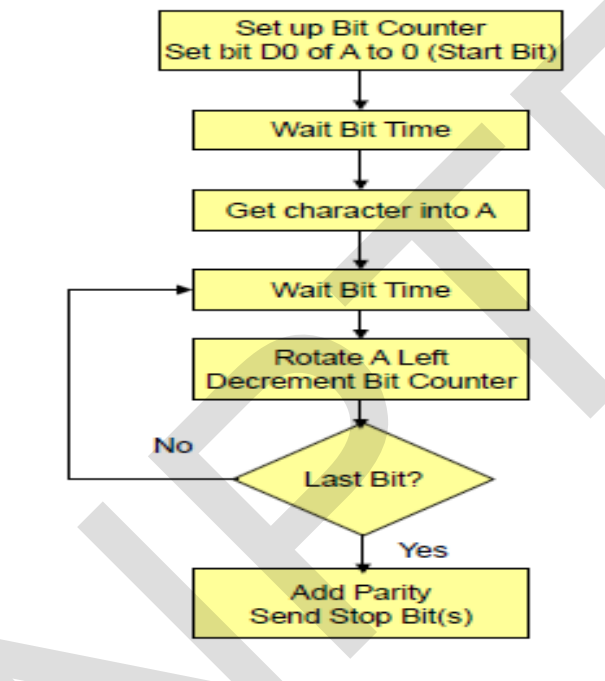
Software-Controlled SerialTransmission

- The main steps involved in serially transmitting a character are:
  - Transmission line is at logic 1 by default.
  - Transmit a start bit for one complete bit length.
  - Transmit the character as a stream of bits with appropriate delay.
  - Calculate parity and transmit it if needed.
  - Transmit the appropriate number of stop bits.
  - Transmission line returns to logic 1.

Serial Transmission



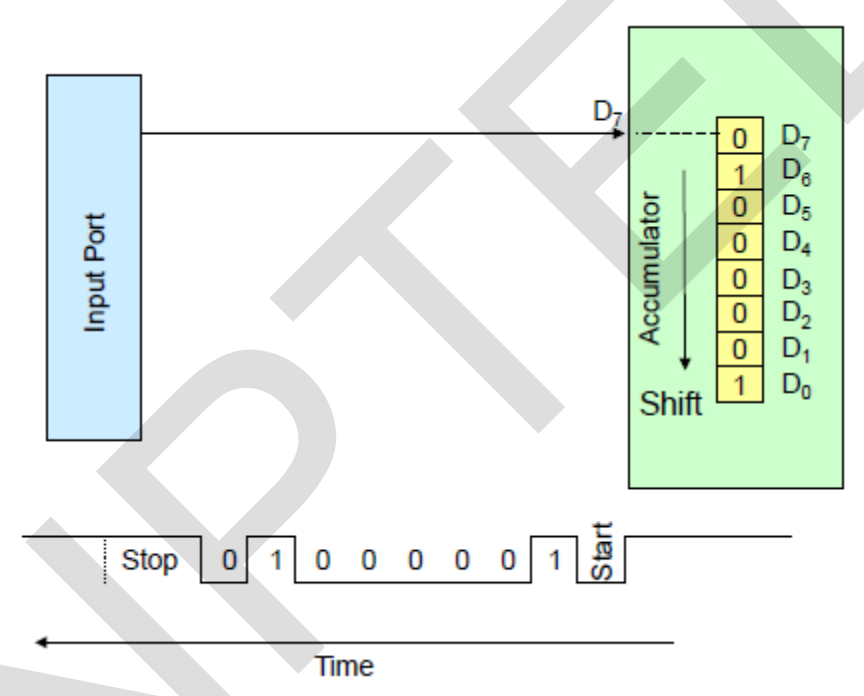
Flowchart of Serial Transmission



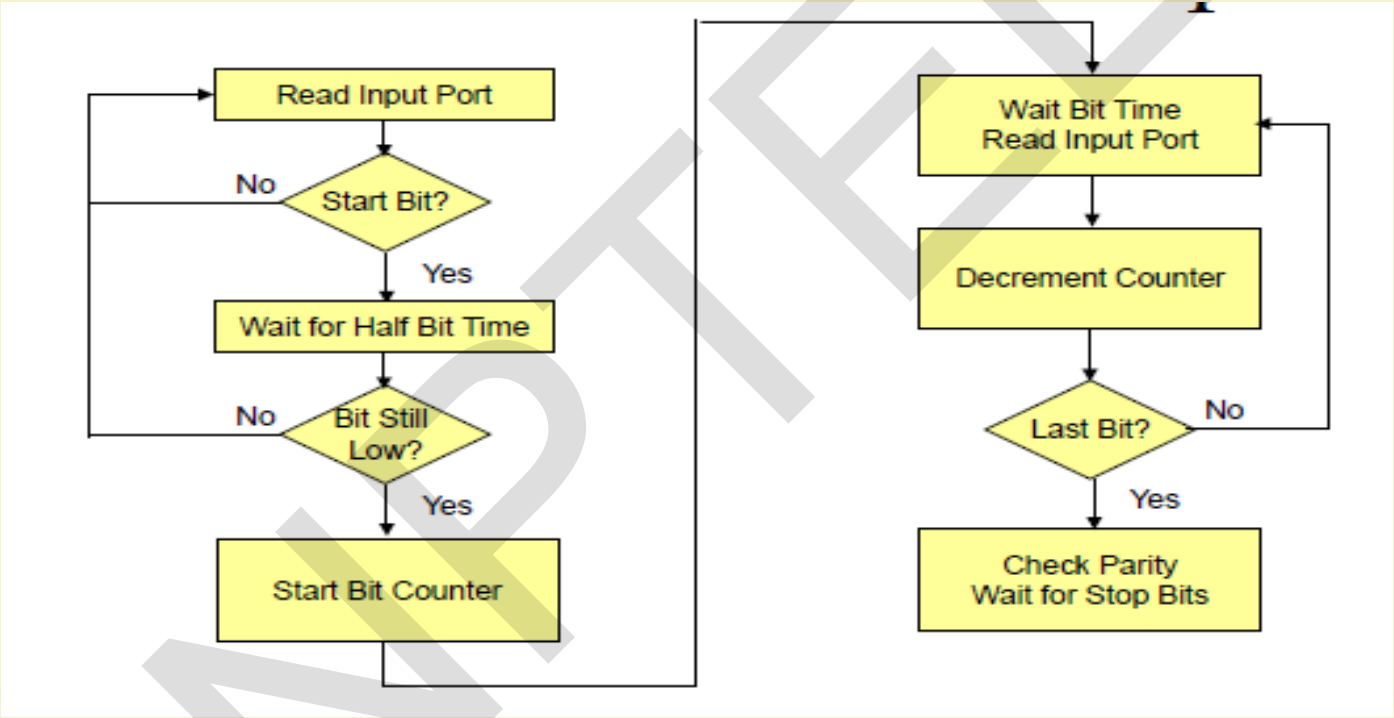
Software-Controlled Serial Reception

- The main steps involved in serial reception are:
  - Wait for a low to appear on the transmission line.
    - Start bit
  - Read the value of the line over the next 8 bit lengths.
    - The 8 bits of the character.
  - Calculate parity and compare it to bit 8 of the character.
    - Only if parity checking is being used.
  - Verify the reception of the appropriate number of stop bits.

Serial Reception



Flowchart of Serial Reception

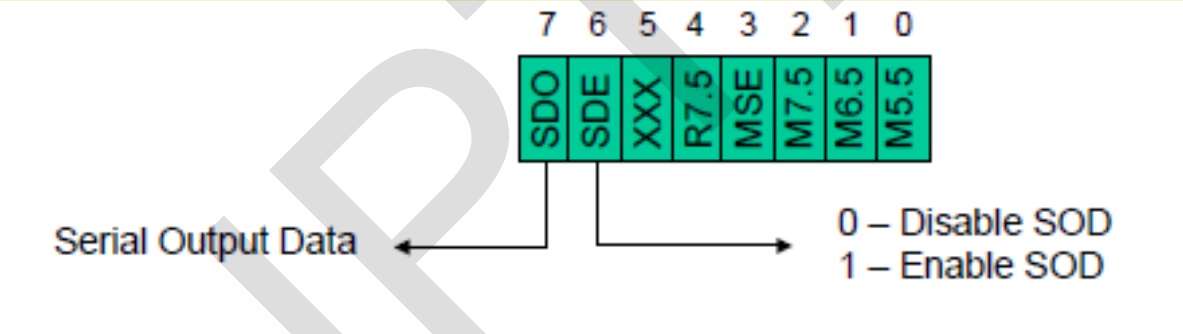


The 8085 Serial I/O Lines

- The 8085 Microprocessor has two serial I/O pins:
  - SOD – Serial Output Data
  - SID – Serial Input Data
- Serial input and output is controlled using the RIM and SIM instructions respectively.

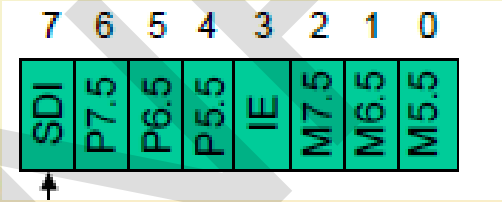
SIM and Serial Output

- The figure below shows how SIM uses the accumulator for Serial Output.



RIM and Serial Input

- Again, the RIM instruction has dual use
  - Reading the current settings of the Interrupt Masks
  - Serial Data Input
- The figure below shows how the RIM instruction uses the Accumulator for Serial Input



Ports?

- Using the SOD and SID pins, the user would not need to bother with setting up input and output ports.
  - The two pins themselves can be considered as the ports.
  - The instructions SIM and RIM are similar to the OUT and IN instructions except that they only deal with the 1-bit SOD and SID ports.

Example

- Transmit an ASCII character stored in register B using the SOD line.

```
SODDATA     MVI     C, 0BH   ; Set up counter for 11 bits
            XRA     A        ; Clear the Carry flag
NXTBIT      MVI     A, 80H   ; Set D7 =1
            RAR              ; Bring Carry into D7 and set D6 to 1
            SIM              ; Output D7 (Start bit)
            CALL    BITTIME
            STC              ; Set Carry to 1
            MOV     A, B     ; Place character in A
            RAR              ; Shift D0 of the character to the carry
                               Shift 1 into bit D7
            MOV     B, A     ; Save the interim result
            DCR     C        ; decrement bit counter
            JNZ     NXTBIT
```

Count number of 1's in the contents of D register and store the count in the B register

```
        MVI B, 00H
        MVI C, 08H
        MOV A, D
BACK:   RAR             ; D0 to CY, CY to D7
        JNC SKIP
        INR B
SKIP:   DCR C
        JNZ BACK
        HLT
```

Sort given 10 numbers from memory location 2200H in the ascending order

```
        MVI B, 09      ;Initialize counter 1
START : LXI H, 2200H ;Initialize memory pointer
        MVI C, 09H     ;Initialize counter 2
BACK:   MOV A, M       ;Get the number
        INX H          ;Increment memory pointer
        CMP M          ;Compare number with next number
        JC SKIP        ;If less, don't interchange
        JZ SKIP        ;If equal, don't interchange
        MOV D, M       ;Interchange two numbers
        MOV M, A
```

Sort given 10 numbers from memory location 2200H in the ascending order (Contd.)

```
        DCX H
        MOV M, D
        INX H
SKIP:   DCR C       ;Decrement counter 2
        JNZ BACK    ;If not zero, repeat
        DCR B       ;Decrement counter 1
        JNZ START
        HLT         ;Terminate program execution
```

Calculate the sum of series of even numbers from the list of numbers. The length of the list is in memory location 2200H and the series itself begins from memory location 2201H. Assume the sum to be 8 bit number so you can ignore carries and store the sum at memory location 2210H.

```
        LDA 2200H
        MOV C, A        ;Initialize counter
        MVI B, 00H      ;sum = 0
        LXI H, 2201H    ;Initialize pointer
BACK:   MOV A, M        ;Get the number
        ANI 01H         ;Mask Bit 1 to Bit7
        JNZ SKIP        ;Don't add if number is ODD
        MOV A, B        ;Get the sum
        ADD M           ;SUM = SUM + data
        MOV B, A        ;Store result in B register
SKIP:   INX H           ;increment pointer
        DCR C           ;Decrement counter
        JNZ BACK        ;if counter 0 repeat
        STA 2210H       ;store sum
        HLT             ;Terminate program execution
```

Unpack the packed BCD number

```
        LDA 3000H    ;Get the packed BCD number from the memory, let it be 98H
        MOV B,A
        MVI C,04
        ANI F0       ;A = 90H
L1:     RRC          ;Need to be rotated right for 4 times to get A = 09H
        DCR C
        JNZ L1
        STA 3001
        MOV A,B
        ANI 0F       ;A = 08H
        STA 3002
        HLT
```

Other Important Instructions

- ADC <reg>: Add register to accumulator with carry
- CMC: Complement carry
- DAA: Decimal adjust accumulator. Acc. content changed from 8-bit binary to two 4-bit BCD digits. If (D3-D0) > 9, add 6.
- DAD <reg_pair>: Add register pair to HL
- IN <port>: Get data from port
- LHLD <addr> : L ← Mem[addr], H ← Mem[addr + 1]
- SHLD <addr>
- PCHL : PC ← HL
- SPHL : SP ← HL
- XCHG : Exchange HL with DE
- XTHL: Exchange HL with top of stack

8086 Microprocessor

Santanu Chattopadhyay
Electronics and Electrical Communication Engineering

Overview

First 16-bit processor released by INTEL in the year 1978

Originally HMOS, now manufactured using HMOS III technique

Approximately 29, 000 transistors, 40 pin DIP, 5V supply

Does not have internal clock; external asymmetric clock source with 33% duty cycle

20-bit address to access memory ⇒ can address up to 2^{20} = 1 megabytes of memory space.

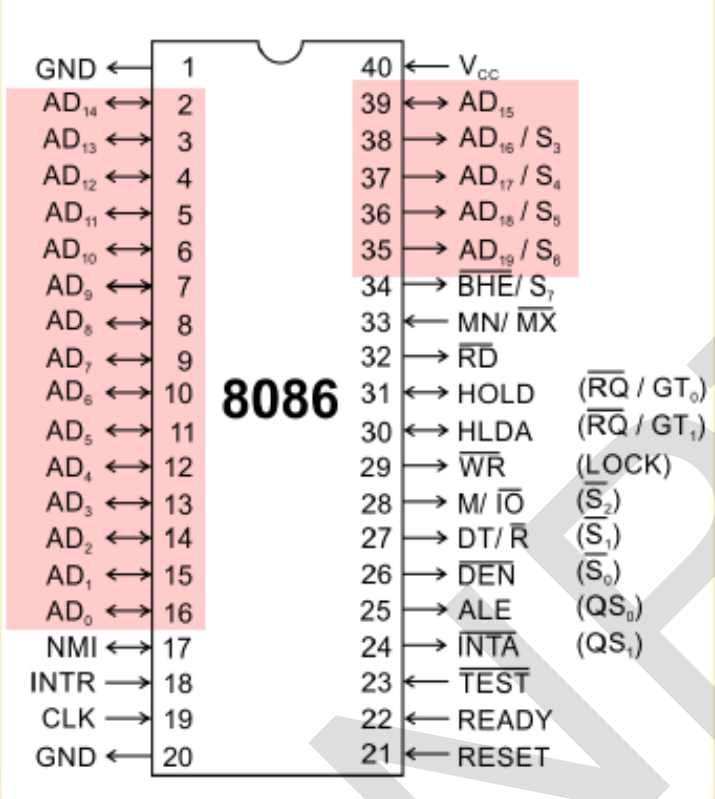
Addressable memory space is organized in to two banks of 512 kb each; Even (or lower) bank and Odd (or higher) bank. Address line A_0 is used to select even bank and control signal $\overline{BHE}$ is used to access odd bank

Uses a separate 16 bit address for I/O mapped devices ⇒ can generate 2^{16} = 64 k addresses.

Operates in two modes: minimum mode and maximum mode, decided by the signal at MN and $\overline{MX}$ pins.

Pins and signals

Pins and Signals



AD_0-AD_{15} (Bidirectional)

Address/Data bus

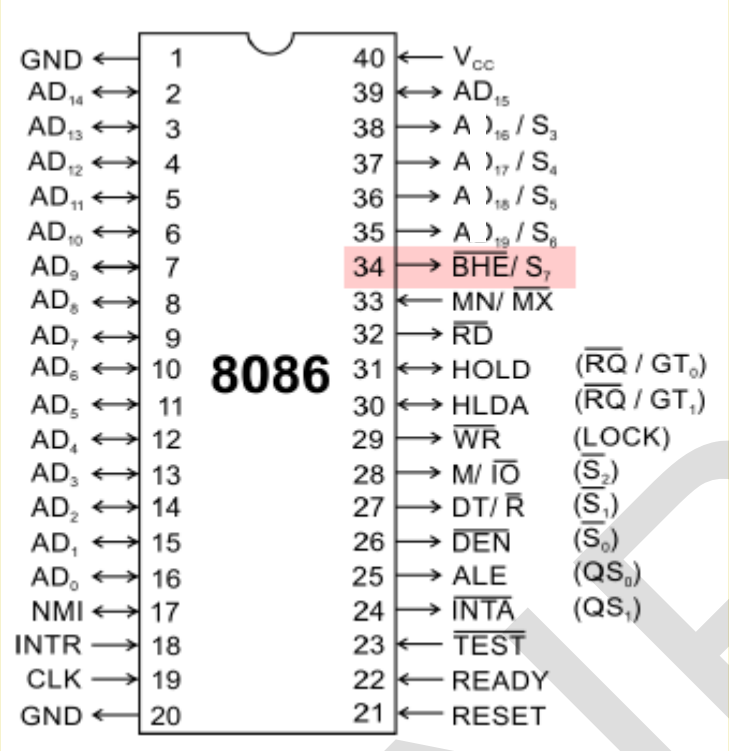
Low order address bus; these are multiplexed with data.

When AD lines are used to transmit memory address the symbol A is used instead of AD, for example A_0-A_{15}.

When data are transmitted over AD lines the symbol D is used in place of AD, for example D_0-D_7, D_8-D_{15} or D_0-D_{15}.

A_{16}/S_3, A_{17}/S_4, A_{18}/S_5, A_{19}/S_6

High order address bus. These are multiplexed with status signals



BHE (Active Low)/S_7 (Output)

Bus High Enable/Status

It is used to enable data onto the most significant half of data bus, D_8-D_{15}. 8-bit device connected to upper half of the data bus use BHE (Active Low) signal. It is multiplexed with status signal S_7.

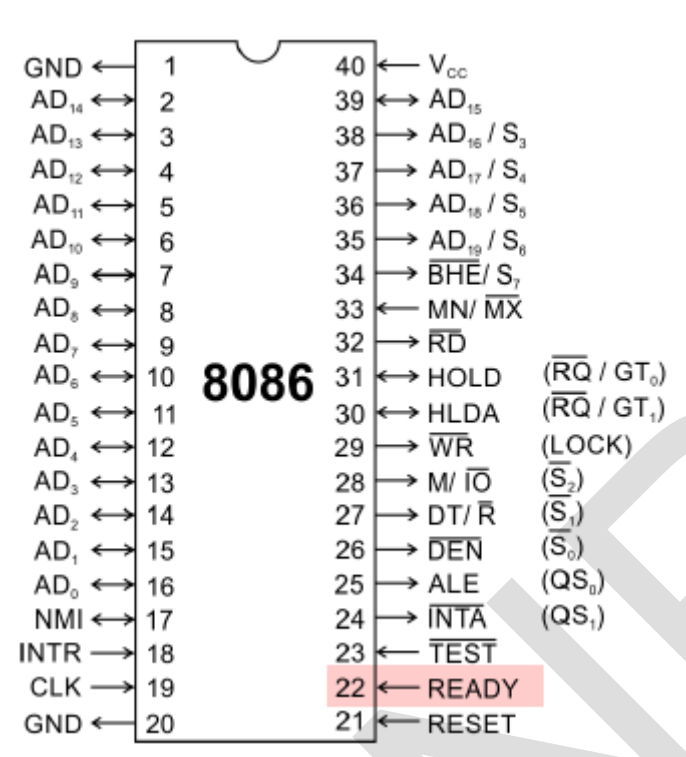
MN/ MX

MINIMUM / MAXIMUM

This pin signal indicates what mode the processor is to operate in.

RD (Read) (Active Low)

The signal is used for read operation.
It is an output signal.
It is active when low.



TEST

$\overline{\text{TEST}}$ input is tested by the 'WAIT' instruction.

8086 will enter a wait state after execution of the WAIT instruction and will resume execution only when the $\overline{\text{TEST}}$ is made low by an active hardware.

READY

This is used to synchronize an external activity to the processor internal operation.

This is the acknowledgement from the slow device or memory that they have completed the data transfer.

RESET (Input)

Causes the processor to immediately terminate its present activity.
The signal must be active HIGH for at least four clock cycles.

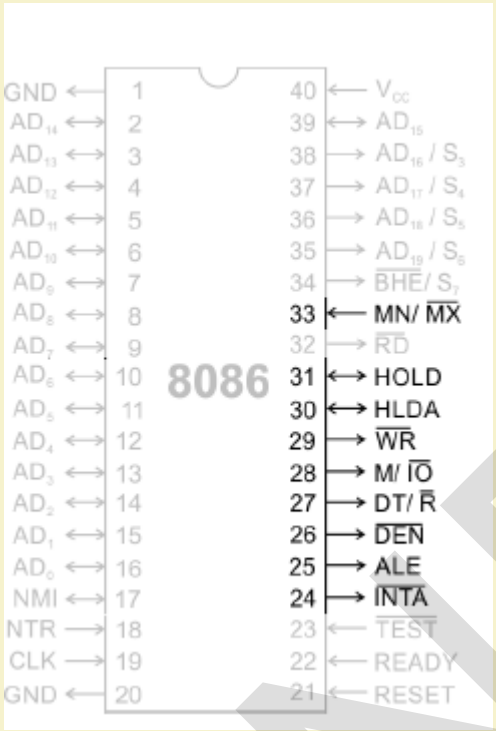
CLK

The clock input provides the basic timing for processor operation and bus control activity. Its an asymmetric square wave with 33% duty cycle.

INTR Interrupt Request

This is a triggered input. This is sampled during the last clock cycles of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle.

Min/ Max Pin



The 8086 microprocessor can work in two modes of operations : Minimum mode and Maximum mode.

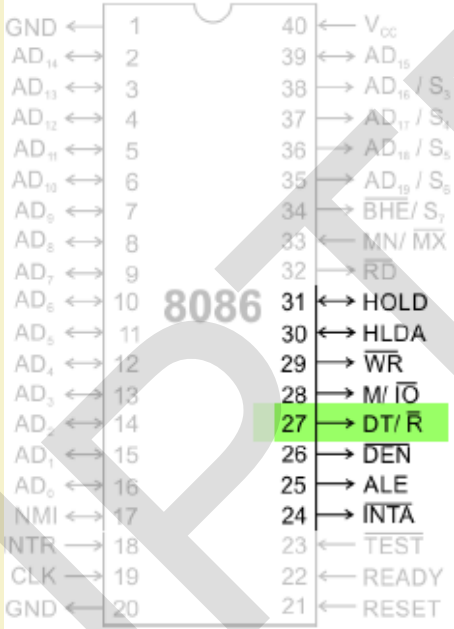
In the minimum mode of operation the microprocessor do not associate with any co-processors and can not be used for multiprocessor systems.

In the maximum mode the 8086 can work in multi-processor or co-processor configuration.

Minimum or maximum mode operations are decided by the pin MN/ MX(Active low).

When this pin is high 8086 operates in minimum mode otherwise it operates in Maximum mode.

Minimum mode signals



Pins 24 -31

For minimum mode operation, the MN/ $\overline{MX}$ is tied to VCC (logic high)

8086 itself generates all the bus control signals

DT/$\overline{R}$ — (Data Transmit/ Receive) Output signal from the processor to control the direction of data flow through the data transceivers

$\overline{DEN}$ — (Data Enable) Output signal from the processor used as out put enable for the transceivers

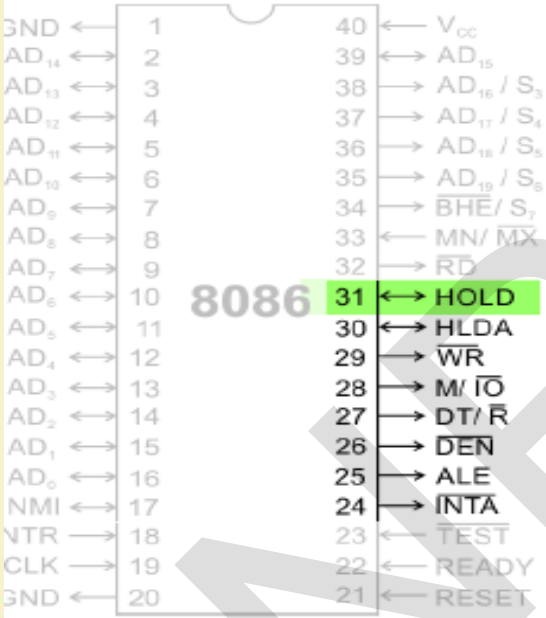
ALE — (Address Latch Enable) Used to demultiplex the address and data lines using external latches

M/$\overline{IO}$ — Used to differentiate memory access and I/O access. For memory reference instructions, it is high. For IN and OUT instructions, it is low.

$\overline{WR}$ — Write control signal; asserted low Whenever processor writes data to memory or I/O port

$\overline{INTA}$ — (Interrupt Acknowledge) When the interrupt request is accepted by the processor, the output is low on this line.

Minimum mode signals



Pins 24 -31

For minimum mode operation, the MN/ $\overline{MX}$ is tied to VCC (logic high)

8086 itself generates all the bus control signals

HOLD — Input signal to the processor form the bus masters as a request to grant the control of the bus.

Usually used by the DMA controller to get the control of the bus.

HLDA — (Hold Acknowledge) Acknowledge signal by the processor to the bus master requesting the control of the bus through HOLD.

The acknowledge is asserted high, when the processor accepts HOLD.

Maximum mode signals

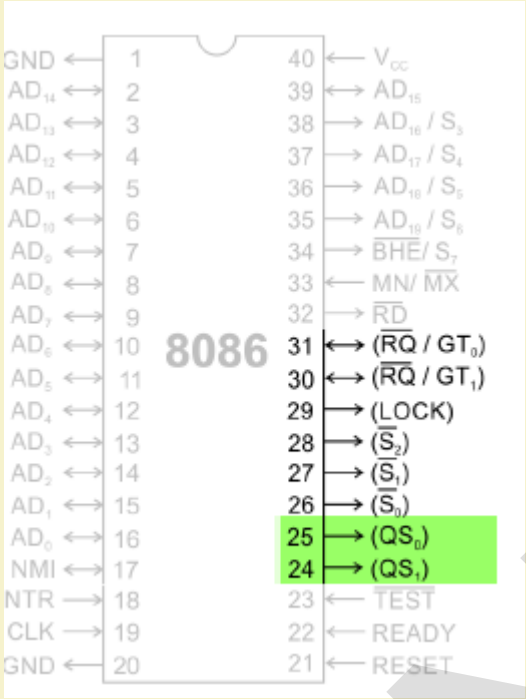
During maximum mode operation, the MN/ $\overline{MX}$ is grounded (logic low)

$\overline{S_0}, \overline{S_1}, \overline{S_2}$	**Status signals; used by the 8086 bus controller to generate bus timing and control signals. These are decoded as shown.**

Pins

Status Signal			Machine Cycle
$\overline{S}_2$	$\overline{S}_1$	$\overline{S}_0$	
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive/Inactive

Maximum mode signals



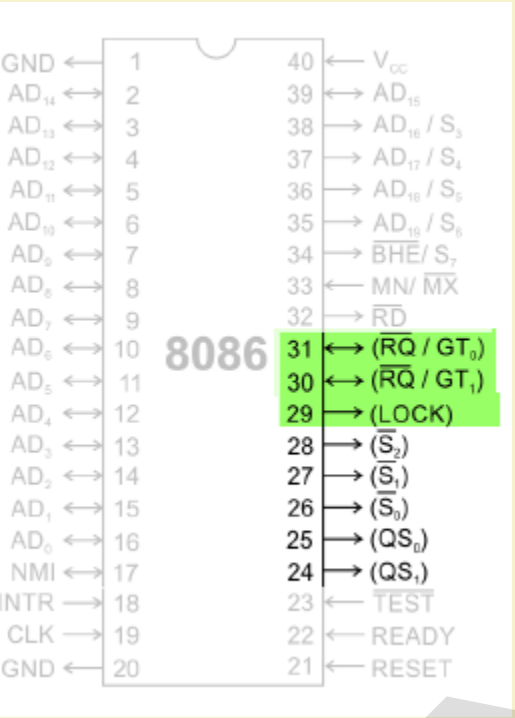
$\overline{QS_0}$, $\overline{QS_1}$ (**Queue Status**) **The processor provides the status of queue in these lines.**

The queue status can be used by external device to track the internal status of the queue in 8086.

The output on QS_0 and QS_1 can be interpreted as shown in the table.

Queue status		Queue operation
QS_1	QS_0	
0	0	No operation
0	1	First byte of an opcode from queue
1	0	Empty the queue
1	1	Subsequent byte from queue

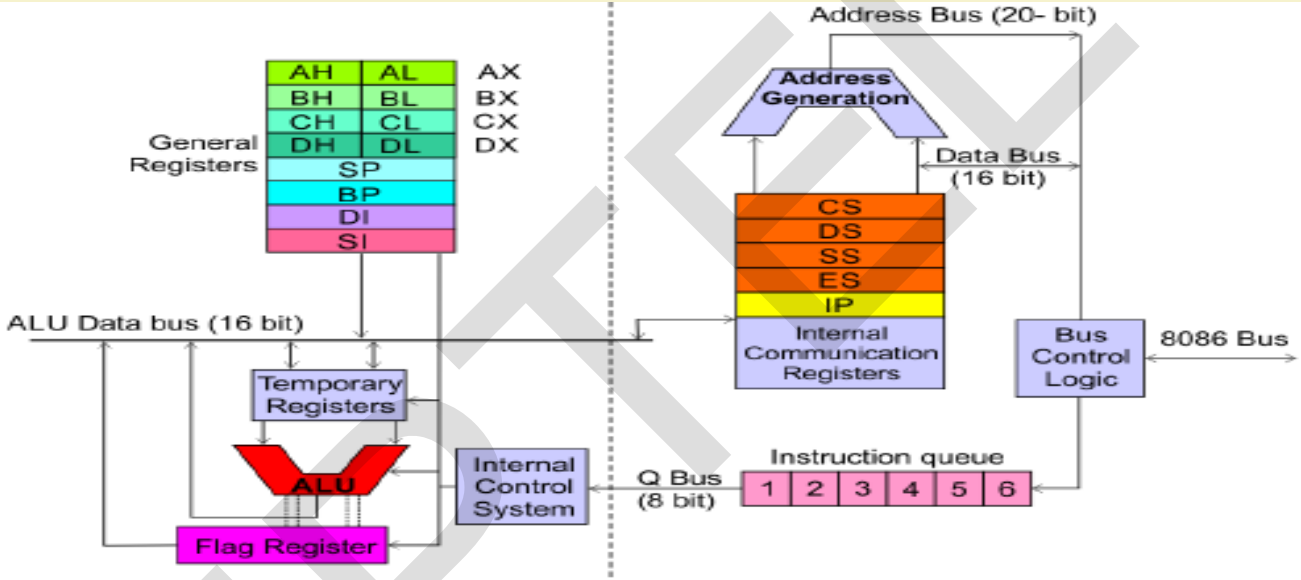
Maximum mode signals



$\overline{RQ}/\ \overline{GT_0}$, $\overline{RQ}/\ \overline{GT_1}$	(Bus Request/ Bus Grant) These requests are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. These pins are bidirectional. The request on $\overline{GT_0}$ will have higher priority than $\overline{GT_1}$
$\overline{LOCK}$	An output signal activated by the LOCK prefix instruction. Remains active until the completion of the instruction prefixed by LOCK. The 8086 output low on the $\overline{LOCK}$ pin while executing an instruction prefixed by LOCK to prevent other bus masters from gaining control of the system bus.

Architecture

Architecture



Execution Unit (EU)

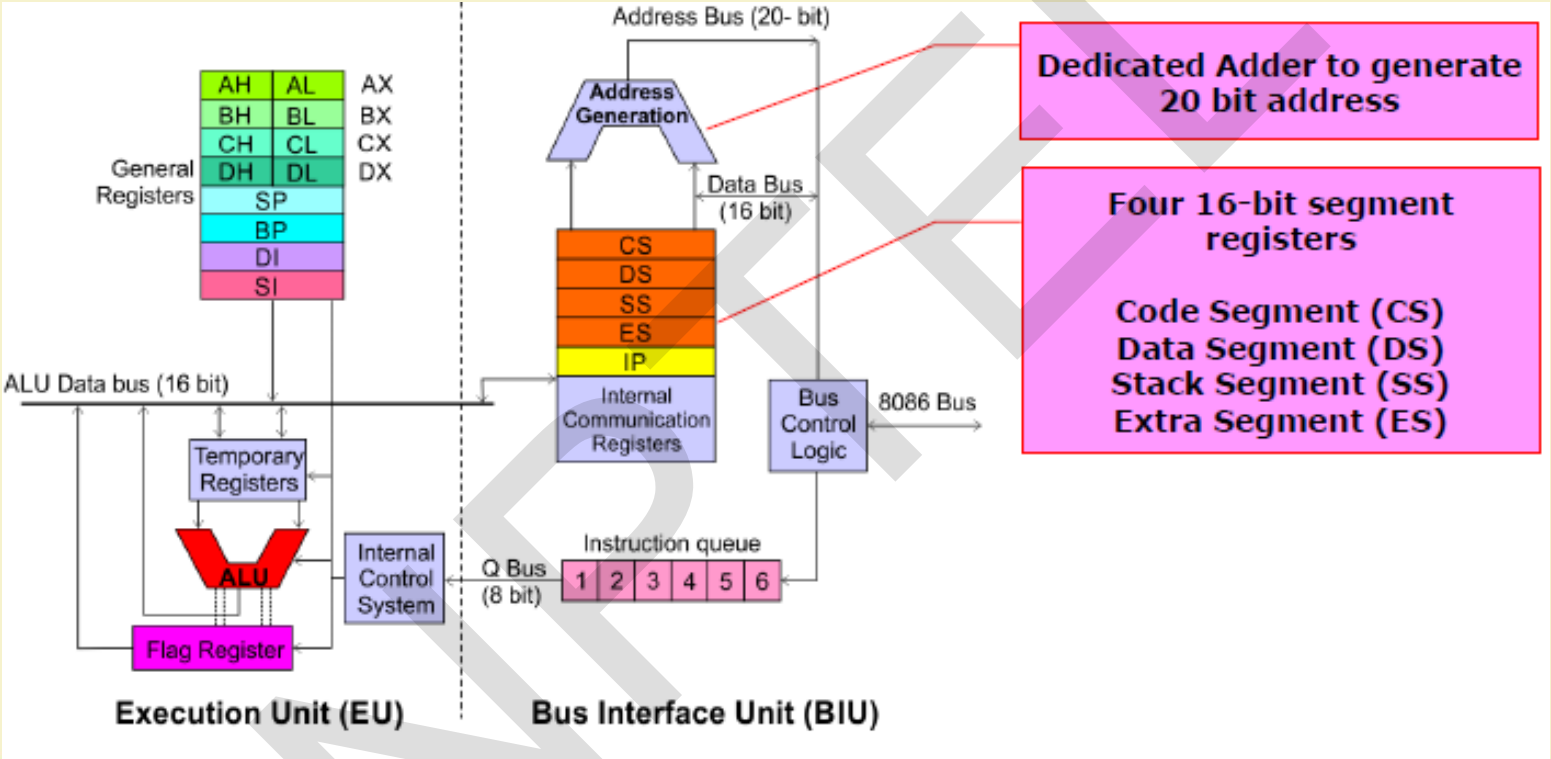
EU executes instructions that have already been fetched by the BIU.

BIU and EU functions separately.

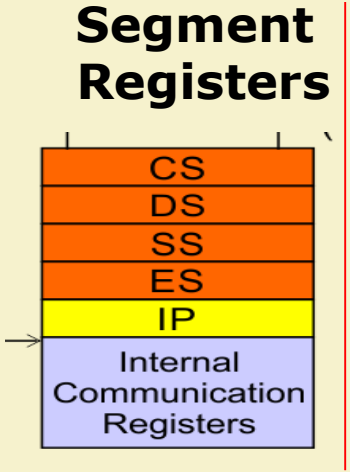
Bus Interface Unit (BIU)

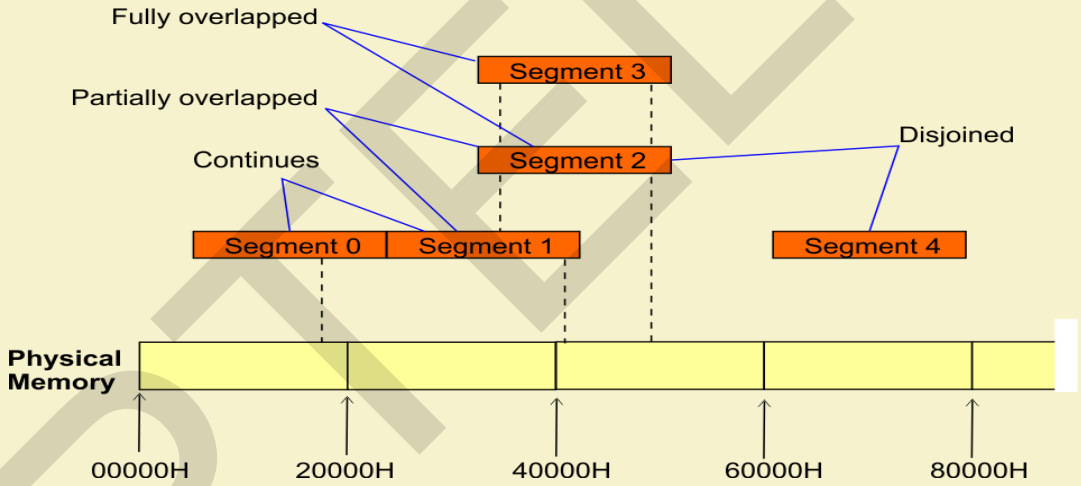
BIU fetches instructions, reads data from memory and I/O ports, writes data to memory and I/ O ports.

Bus Interface Unit (BIU)



Bus Interface Unit (BIU)

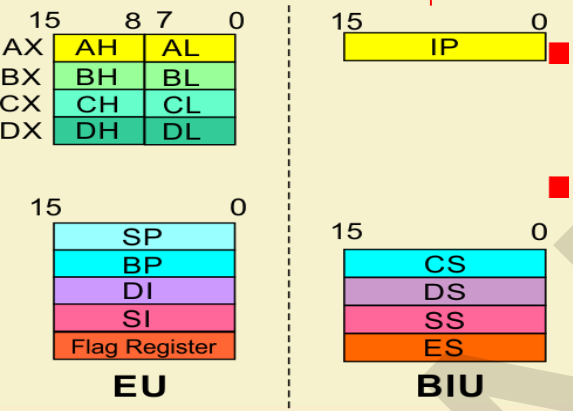




- 8086's 1-megabyte memory is divided into segments of up to 64K bytes each.
- The 8086 can directly address four segments (256 K bytes within the 1 M byte of memory) at a particular time.
- Programs obtain access to code and data in the segments by changing the segment register content to point to the desired segments.

Bus Interface Unit (BIU)

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.

Bus Interface Unit (BIU)

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.

Bus Interface Unit (BIU)

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.

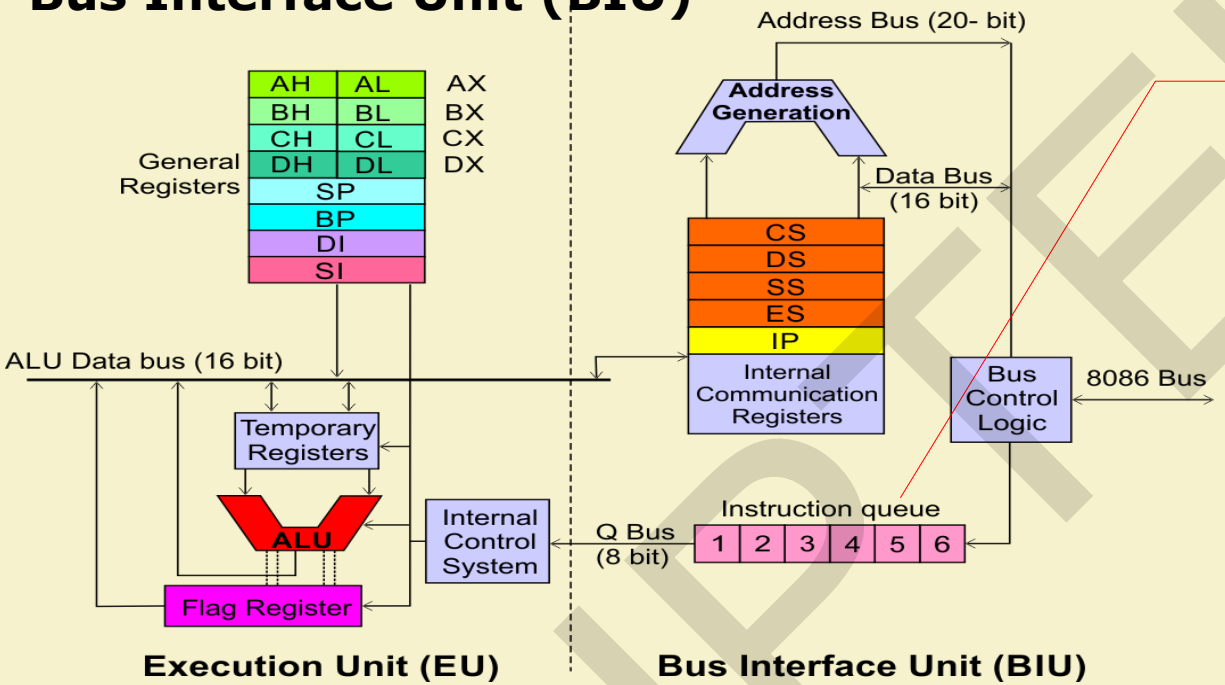
Bus Interface Unit (BIU)

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.

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.

Execution Unit (EU)

EU decodes and executes instructions.

A decoder in the EU control system translates instructions.

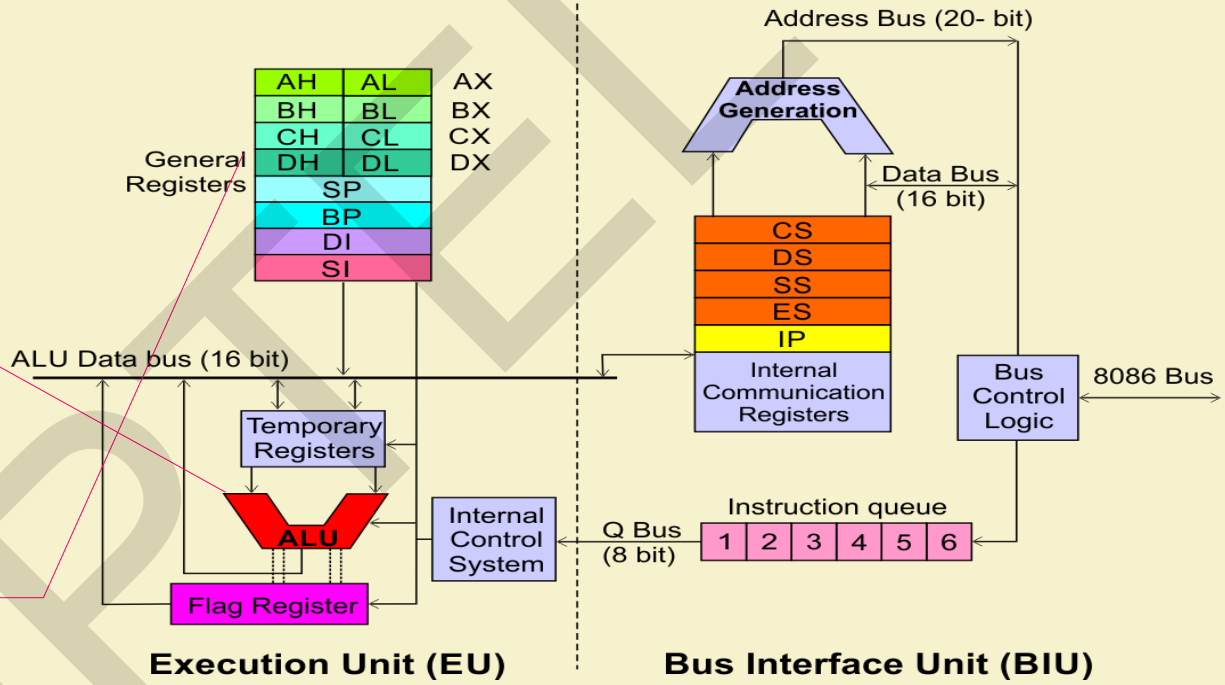
16-bit ALU for performing arithmetic and logic operation

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, BX, CX, DX

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.

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.

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.

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.**

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.

Tarp 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 registers categorized into 4 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

Registers and Special Functions

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

- **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

Register Addressing Modes

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)

Immediate 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) $\leftarrow$ 08_H

MOV AX, 0A9FH

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

(AX) $\leftarrow$ $0A9F_H$

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

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

Direct 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.

Register Indirect 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]

Operations:

EA = (BX)

BA = (DS) x 16_{10}

MA = BA + EA

(CX) ← (MA) or,

(CL) ← (MA)

(CH) ← (MA +1)

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

Based 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)**

$EA = (BX) + 0008_H$

$BA = (DS) \times 16_{10}$

$MA = BA + EA$

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

$(AL) \leftarrow (MA)$, $(AH) \leftarrow (MA + 1)$

Indexed 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,**

$(CL) \leftarrow (MA)$

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

Based Indexed 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,**

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

String Addressing

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

Calculation of destination memory location:

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

$(MA_E) \leftarrow (MA)$

If DF = 1, then (SI) ← (SI) – 1 and (DI) = (DI) - 1

If DF = 0, then (SI) ← (SI) +1 and (DI) = (DI) + 1

I/O Port 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) ← (PORT) Content of port with address 09_H is moved to AL register

In indirect port addressing mode, the instruction will specify the name of the register which holds the port address. In 8086, the 16-bit port address is stored in the DX register.

Example: OUT [DX], AX
Operations: $PORT_{addr}$ = (DX)
(PORT) ← (AX)

Content of AX is moved to port whose address is specified by DX register.

Relative 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) x 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.

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**

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.

Data Transfer Instructions

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

MOV reg2/ mem, reg1/ mem	
MOV reg2, reg1	(reg2) ← (reg1)
MOV mem, reg1	(mem) ← (reg1)
MOV reg2, mem	(reg2) ← (mem)
MOV reg/ mem, data	
MOV reg, data	(reg) ← data
MOV mem, data	(mem) ← data
XCHG reg2/ mem, reg1	
XCHG reg2, reg1	(reg2) ↔ (reg1)
XCHG mem, reg1	(mem) ↔ (reg1)

Data Transfer Instructions

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

PUSH reg16/ mem	
PUSH reg16	$(SP) \leftarrow (SP) - 2$ $MA_S = (SS) \times 16_{10} + SP$ $(MA_S ; MA_S + 1) \leftarrow (reg16)$
PUSH mem	$(SP) \leftarrow (SP) - 2$ $MA_S = (SS) \times 16_{10} + SP$ $(MA_S ; MA_S + 1) \leftarrow (mem)$
POP reg16/ mem	
POP reg16	$MA_S = (SS) \times 16_{10} + SP$ $(reg16) \leftarrow (MA_S ; MA_S + 1)$ $(SP) \leftarrow (SP) + 2$
POP mem	$MA_S = (SS) \times 16_{10} + SP$ $(mem) \leftarrow (MA_S ; MA_S + 1)$ $(SP) \leftarrow (SP) + 2$

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)

Arithmetic Instructions

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

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

Arithmetic Instructions

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

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

Arithmetic Instructions

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

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

Arithmetic Instructions

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

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

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

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)

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**

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

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

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

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**

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

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

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.

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

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

String Manipulation Instructions

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

Compare two string byte or string word

CMPS **CMPSB** **CMPSW**	**MA = (DS) x 16_{10} + (SI)** **MA_E = (ES) x 16_{10} + (DI)** **Modify flags ← (MA) - (MA_E)** **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**

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
	If DF = 1, then (DI) ← (DI) – 2

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

String Manipulation Instructions

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

Store byte from AL or word from AX in to string

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

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

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

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

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

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