KERNEL MASTERS

# Assembly Language Programming (ALP)

## Material

Prepared By Kishore Kumar Boddu

# Assembly Language Programming

1. 8085/8086 Microprocessor
   - 1.1. 8085 microprocessor Specifications
   - 1.2. Signal Description of 8085 up
   - 1.3. Functional Block Diagram of 8085 up
   - 1.4. How Does the Microprocessor Works?
   - 1.5. Instruction set of 8085 up
   - 1.6. Drawbacks of 8085 up (Difference between 8085 & 8086)
   - 1.7. Functional Block Diagram of 8086 up
   - 1.8. Interfacing Peripherals

2. Memory Organization
   - 2.1. Memory Basics – Structure of a Memory
   - 2.2. Memory Mapping
   - 2.3. Memory Hierarchy.

3. I/O Organization
   - 3.1. Introduction to I/O organization
   - 3.2. Programmed Control I/O
   - 3.3. Interrupt Driven I/O
   - 3.4. Direct Memory Access

# 1. 8085/8086 Microprocessor

## 1.1. 8085 microprocessor Specifications

- It is enclosed with 40 pins DIP (dual in line package).
- It requires a signal +5v power supply and operates at 3.01 MHZ signal phase clock.
- It is manufactured with N-MOS technology.
- Data base bus is group of 8 lines $D_0$ -$D_7$. It is an bit microprocessor.
- It has 16-bit address bus and hence can address up to 216=65536 bytes (64kb memory location through $A_0$-$A_{15}$.
- The first 8 lines of address bus and 8 lines of data bus are multiplexed $AD_0$-$AD_7$.
- It is having on chip clock generation facility.
- Crystal frequency of 8085 microprocessor is 6.144 MHZ. it is always double to that of clock frequency.
- It support five hardware interrupts and eight software interrupts.
- Hardware interrupts: TRAP, RST 7.5, RST 6.58, RST 5.5 and INTR.
- TRAP is also called RST 4.5. TRAP is high priority interrupt and INTR is least priority interrupt.
- Software interrupts: RST n (RST: Restart) where n=0 to 7
- Lengths of 8085 instruction vary from one to three bytes.
- 8085 supports five status flags: Sign (S), Zero (Z), Auxiliary Cary (AC), Parity (P) and Carry (CY).
- 8085 microprocessor consists of two 16- bit address register: Program Counter (PC) and Stack Pointer (SP).
- Program Counter always holds address of next memory location to be accessed.
- Stack Pointer always holds address of the top of the stack.
- 8085 consists of six 8- bit general purpose registers which are accessible to the programmer: B, C, D, E, H and L.
- Based on requirement six bit processor registers can be used as three register pairs: BC, DE and HL.
- 8085 also contains an 8-bit processor register called Accumulator 'A'.
- Range of addresses generated by 8085 microprocessor: 0000H to FFFFH

## 1.2. Signal Description of 8085

All the available signals (40) of 8085 can be classified in to six groups:

a. Power supply & Frequency signals.
b. Serial IO Ports
c. Address Bus
d. Data Bus
e. Status, Control& Acknowledge signals.
f. Interrupts and externally initiated signals
g. Power Supply & Frequency signals:
   Vcc: +5 Volt supply
   Vss: Ground reference.
   X1, X2 (Input)

Crystal or R/C network connections to set the internal clock generator X1 can also be an external clock input instead of a crystal. The input frequency is divided by 2 to give the internal operating frequency.

**a. Serial IO Ports**

**SID (Input)**

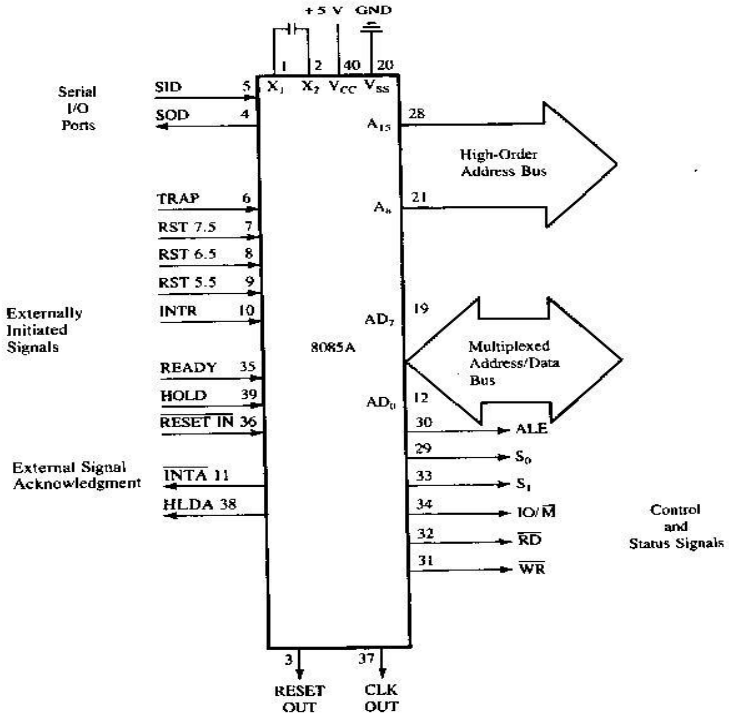
Serial input data line the data on this line is loaded into accumulator bit 7 whenever a RIM (Read Interrupt Mask) Instruction is executed.

**SOD (Output)**

Serial output data line. The output SOD is set or reset as specified by the SIM (Set Interrupt Mask) Instruction.

Pair of signal used for serial I/O communication: SID & SOD.

Pair of instruction used for serial I/O communication: SIM & RIM.




**b. Address bus**

**A8-A15 (Output**)

Address bus; the most significant 8 bits of the I/O address, 3 stated during hold and halt modes.

**c. Data Bus**

**AD0-AD7 (Input/output 3State)**

Multiplexed Address/Data Bus; Lower 8 bits of the memory address (or I/O addresses) appear on the bus during the first clock cycle of a machine state. 3 stated during Hold and Halt modes.

**d. Status, Control & Acknowledge signals**

- S1, S0 and IO/M signals are called status signals.
- RD and WR signals are control signals.
- HLDA and INTA are acknowledge signals.
- Hold and HLDA signals are used for DMA (Direct Memory Access) operation.
- ALE is a special signal to indicate the beginning of the operation.

**ALE (Output)**

Address Latch Enable: This is positive going pulse generated every time the 8085 begins an operation (machine cycle); it indicates that the bits on AD7-AD0 are address bits. This signal is used primarily to latch the low-order address from the multiplexed bus and generate a separate set of address lines, A7-A0.

**SO, S1 (Output)**

S0 and S1 signals to indicate the current of the processor.

| S1 | S0 | Status |
|---|---|---|
| 0 | 0 | Halt |
| 0 | 1 | Write |
| 1 | 0 | Read |
| 1 | 1 | Fetch |

**IO/M (Output)**

This is a status signal used to differentiate between I/O and memory operations. When it is high, it indicates an I/O operation; when it is low. It indicated a memory operation.

**RD (Output 3state)**

READ; indicates the selected memory or I/O device is to be read and that the Data Bus is available for the data transfer.

**WR (Output 3state)**
WRITE; indicates the data on the Data Bus is to be written into the selected memory or I/O location. Data is set up at the trailing edge of WR. 3 stated during Hold and Halt modes. By Combining the status signal IO/M with control signals RD and WR we can generate four different signals.

| IO/M | RD | WR | Operation |
|---|---|---|---|
| 0 | 0 | 1 | MEMR |
| 0 | 1 | 0 | MEMW |
| 1 | 0 | 1 | IOR |
| 1 | 1 | 0 | IOW |

**HLDA (Output)**
HOLD ACKNOWLEDGE; indicates that the CPU has received the Hold request and that it will relinquish the buses in the next clock cycle. HLDA goes low after the Hold request is removed. The CPU takes the buses one half clock cycles after HLDA goes low.

**RESET OUT (Output)**
Indicates CP1J is being reset. Can be used as a system RESET. The signal is synchronized to the processor clock.
RESETOUT signals is used to connect to RESET IN of other interfacing circuits used in microprocessor based systems.

**CLK (Output)**
Clock Output for use as a system clock when a crystal or R/C network is used as an input to the CPU. The period of CLK is twice the X1, X2 input period.
CLOCKOUT of 8085 will be connected to clock in of other interfacing circuits used in micro based systems to synchronize the operation with 8085.

## e. Interrupts and externally initiated signals

| Interrupts | Type | Trigger | Vector |
|---|---|---|---|
| TRAP | Nonmaskable | Level& Edge Sensitive | 0024H |
| RST 7.5 | Maskable | Edge Sensitive | 003C |
| RST 6.5 | Maskable | Level Sensitive | 0034 |
| RST 5.5 | Maskable | Level Sensitive | 002C |
| INTR | Maskable | Level Sensitive | 0000H to 0038H |

**INTR (Input)**
INTERRUPT REQUEST; is used as a general purpose interrupt. It is sampled only during the next to last clock cycle of the instruction. If it is active, the program Counter (PC) will be inhibited from incrementing and an INTA will be issued. During this cycle a RESTART or CALL instruction can be inserted to jump to the interrupt service routine. The INTR is enabled and disabled by software. It is disabled by reset and immediately after an interrupt is accepted.

**INTA (Output**) INTERRUPT ACKNOWLEDGE; is used instead of (and has the same timing as) RD during the instruction cycle after an INTR is accepted. IT can be used to activate the 8259 Interrupt chip or some other interrupt port.
RST 5.5
RST 6.5 – (Inputs)
RST 7.5
RESTART INTERRUPTS; These three inputs have the same timing as INTR except they cause an internal RESTART to be automatically inserted.
RST 7.5 – Highest Priority
RST 6.5
RST 5.5 $_{o}$ lower Priority

The priority of these interrupts is shown above. These interrupts have a higher priority than the INTR.

**TRAP (Input)**

Trap Interrupt is a non maskable restart interrupt Enable. It has the highest Priority of any interrupt.

**READY (Input)**

Ready signal is used by the microprocessor to communicate with slow operating peripherals. If Ready is high during a read to send or receive data. IF Ready is low, the CPU will wait for Ready to go high before completing the read or write cycle.

**HOLD (Input)**

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 buses as soon as the completion of the current machine cycle. Internal processing can continue. The processor can regain the buses only after the Hold is removed.

When the Hold is acknowledged, the Address, Data, RD, WR, and IO/M lines are 3Stated.

**RESET IN (Input)**

Reset sets the program Counter to zero and resets the interrupt Enable and HLDA flip-flops. None of the other flags or registers (except the instruction register) are affected The CPU is held in the reset condition as long as Reset is applied.

DMA is having highest priority over all the interrupts.

- ✓ SIM instruction is used for serial output data operation as well as to mask or unmask different makeable vectored interrupts.
- ✓ RIM instruction is used for serial input data operation as well as to read the status of different Makeable vectored interrupts.
- ✓ Vectored address corresponds to the S W interrupts

RST 0 ------ 0000H
RST 1 ------ 0008H
RST 2 ------ 0010H
RST 3 ------ 0018H
RST 4 ------ 0020H
RST 5 ------ 0028H
RST 6 ------ 0030H
RST 7 ------ 0038H

## 1.3. Functional Description of 8085.

It includes the following blocks

a. ALU
b. Timing and Control Unit
c. Instruction Register and Decoder
d. Register Array
e. Interrupt Control
f. Serial I/O Control.

**a. ALU**

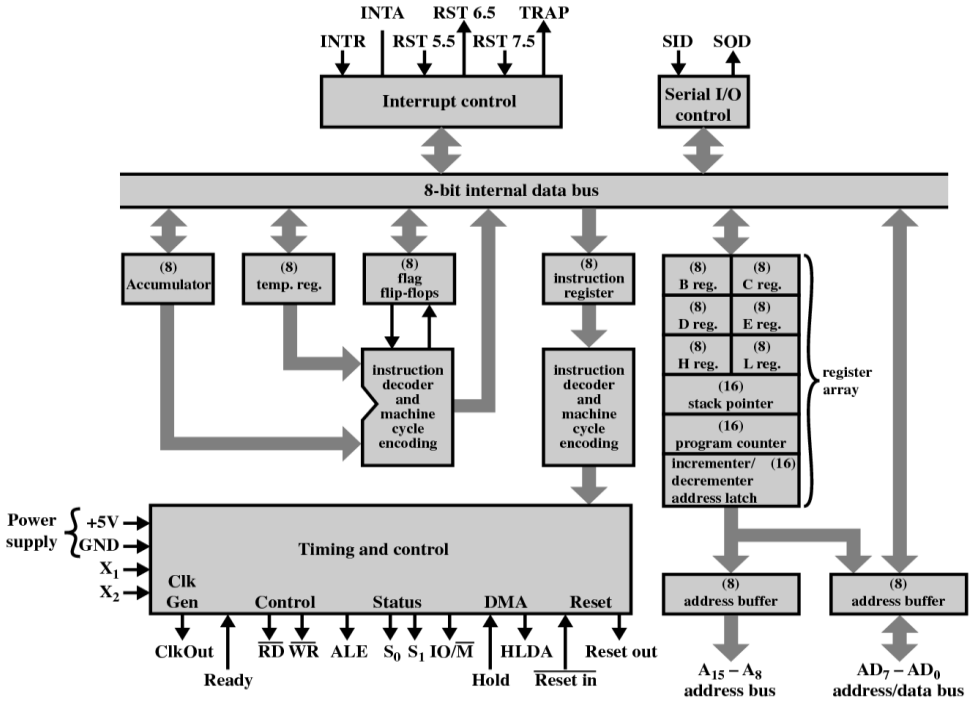
The arithmetic/logic unit performs the computing functions; it includes the accumulator, the temporary register, the arithmetic and logic circuits, and five flags. The temporary register is used to hold data during an arithmetic /logic operation.

Accumulator

The accumulator is an 8- bit register that is a part of arithmetic /logic unit (ALU). This register is used to store 8-bit data and to perform arithmetic and logical operations. The result of an operation is stored in the accumulator. The accumulator is also identified as operation is stored in the accumulator register is also called processor register.

The result is stored in the accumulator, and the flags (flip-flops) are set or reset according to the result of operation. The ALU performs the actual numerical and logic such as 'add', 'subtract', 'AND', 'OR', etc.

Uses data from memory and from accumulator to perform arithmetic Always stores result of operation in Accumulator.




**Flags**

The ALU includes five flip-flops, which are set or reset after an operation according to data conditions of the result in the accumulator and other registers. They are called Zero (Z), Carry (CY). Sign (S), Parity (P), and Auxiliary Carry (AC) flags; they are listed in the Table and their bit positions in the flag register are shown in the Figure below. The most commonly used flags are Zero, Carry, and Sign. The microprocessor uses flags to test data conditions.

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| S | Z | | AC | | P | | CY |

**S-Sign flag:** After the execution of an arithmetic or logic operation, if bit D7 of the result is 1, the sign flag is set. This flag is set. This flag is set. This flag is used with signed numbers. In a given byte, if D7 is 1, the number will be viewed as a negative number; if it is 0, the number will be considered positive.

**Z-Zero flag:** The Zero flag is set if the ALU operation results in 0, and the flag is reset if the result is not 0. This flag is modified by the results in the accumulator as well as in the other registers.

AC-Auxiliary Carry flag: In an arithmetic operation, when a carry is generated by digit D3 and passed on digit D4, the AC flag is set. The flag is used only internally for BCD operations and it not available for the programmer to change the sequence of a program with a jump instruction.

**P-Parity flag:** After an arithmetic or logical operation, if the result has as even number of 1's, the flag is set. If it has an odd number of 1's, the flag is reset.

**CY-Carry flag:** If an arithmetic operation results in a carry, the flag is set; otherwise it is reset. The carry flag also serves as a borrow flag for subtraction.

However, it is not used as a register; five bit positions out of eight are used to store the out puts of the five flip-flops. The flags are stored in the 8-bit register so that the programmer can examine these flags (data conditions) by accessing the register through an instruction.

These flags have critical importance in the decision-making process of the microprocessor. The conditions (set or reset) of the flags are tested through the software instructions. For example, the instruction JC (Jump on

Carry) is implemented to change the sequence of a program when CY flag is set. The Thorough understanding of flag is essential in writing assembly language programs.

**b. Timing and Control Unit**

This unit synchronizes all the microprocessor operations with the clock and generates the control Signals necessary for communication between the microprocessor and peripherals. The control signals are similar to sync pulse in an oscilloscope. The RD and WR signals are sync pulses indicating the availability of data on the data bus.

**Control Unit**

Generates signals within up to carry out the instruction, which has been decoded. In reality causes certain connection between blocks of the microprocessor to be opened or closed, so that data goes where it is required, and so that ALU operations occur.

**c. Instruction Register and Decoder**

The instruction register and the decoder are part of the ALU. When an instruction is fetched from memory, it is loaded in the instruction register. The decoder decodes the instruction and establishes the Sequences of events to follow. The instruction register in not programmable and cannot be accessed through any Instruction.

**d. Register Array**

Two additional register called temporary register W and Z are included in the register array. These register are used to hold 8-bit data during the execution of some instruction .however, because they are used internally, they are not available to the programmer. The 8085/8080 A-programming model includes six register, one accumulator and one flag register, as shown in figure. In addition, it has two 16-bit register: the stack pointer and the program counter .they are described briefly as follow. the 8085/8080 A has six general -purpose register to store 8-bit data; these are identified as B, C, D, E, H, and L as shown in the figure .they can be combined as register pairs –BC, DE, and HL – to perform Some 16-bit operation .the programmer can use these register to store or copy data into the register by using data copy instruction.

**Program Counter (PC)**

This 16-bit register deals with sequencing the execution of instruction .this register is a memory pointer. Memory location have 16-bit address, and that is why this is a 16-bit register .the microprocessor uses this register to sequence the execution of the instructions. The function of the program counter is to point to the memory address from which the next byte is to be fetched .when a byte (machine code) is being fetched; the program counter is incremented by one to point to the next memory location

**Stack pointer (SP)**

The stack pointer is also a 16-bit register used as a memory pointer .it points stack is defined by loading 16-bit address in the stack pointer. The stack concept is explained in the chapter “stack and subroutines”. Holds address, received from PC, of next program instruction. Feeds the address bus with addresses of location of the program under execution.

**Control Generator**

Generates signals within up to carry out the instruction which has been decoded. In reality causes certain connections between blocks of the up to be opened Or closed, so that data goes where it is required, and so that ALU operations occur.

**Register selector**

This block controls the use of the register stack in the example. Just a logic circuit which switches between different registers in the set will receive instructions from Control Unit.

**General purpose Registers**

Microprocessor requires extra registers for versatility. Can be used to store additional data during a program. More complex processors may have a variety of differently named registers.

**Microprogramming**

How the up does knows what an instruction mean, especially when it is only a binary number? The micro program in a uP/uC is written by the chip designer and tells the uP/uC the meaning of each instruction uP/uC can then carry out operation.

**e. Interrupt Control**

Refer “Interrupts and externally initiated signals” section.

**f. Serial I/O Control**

**SID (Input)**

Serial input data line. The data on this line is loaded into accumulator bit -7 whenever a RIM (Read Interrupt

Mask) instruction is executed.
SOD (OUTPUT)
Serial output data line. The output SOD is set or reset as specified by the SIM (Set Interrupt Mask) instruction.
Pair of signals used for serial I/O communications: SID&SOD.
Pair of instruction used for serial I/O communications: SIM & RIM.

**Basic Operational Concepts of a Computer**
Most computer operations are executed in the ALU (arithmetic and logic unit) of processor.
Example: to add two numbers that are both located in memory.

- Each number is brought into the processor, and the actual addition is carried out by the ALU.
- The sum then may be stored in memory or retained in the processor for immediate use.

**Instructions**
Instructions for a processor are defined in the ISA (Instruction Set Architecture)
Typical instructions include:

– Mov BX, AX
  - Fetch the instruction
  - Fetch the contents of memory location AX
  - Store the contents in general purpose register BX

– Add AX,BX
  - Fetch the instruction
  - Add the contents of registers BX and AX
  - Place the sum in register AX

**How are instructions sent between memory and the processor?**

1. The program counter (PC) or instruction pointer (IP) contains the memory address of the next instruction to be fetched and executed.
2. Send the address of the memory location to be accessed to the memory unit and issue the appropriate control signals (memory read).
3. The instruction register (IR) holds the instruction that is currently being executed.
4. Timing is crucial and is handled by the control unit within the processor.

| Parameter | 8085 | 8086 |
|---|---|---|
| **Size** | 8085 is 8 bit microprocessor | 8086 is 16 bit microprocessor |
| **Address Bus** | 8085 has 16 bit address bus | 8086 has 20 bit address bus. |
| **Memory** | 8085 can access up to 2^16 = 64 Kb of memory | 8086 can access up to 2^20 = 1 MB of memory. |
| **Instruction Queue** | 8085 doesn't have an instruction queue | 8086 has instruction queue. |
| **Pipelining** | 8085 does not support pipelined architecture | 8086 supports pipelined architecture. |
| **Multiprocessing Support** | 8085 does not support multiprocessing | multiprocessing support |
| **I/O** | 8085 can address 2^8 = 256 I/O's | 8086 can access 2^16 = 65,536 I/O's |
| **Multiplication and Division** | 8085 doesn't support | whereas 8086 supports |
| **Operating Modes** | 8085 supports only single operating mode | 8086 operates in two modes. |
| **Memory Segmentation** | In 8085, memory space is not segmented | In 8086, memory space is segmented. |

**Programmable Interfacing IC:**
8259 - Programmable Interrupt Controller
8253 - Programmable Timer
8250 – UART (Universal Asynchronous Receiver and Transmitter)
8279 – Keyboard Controller
8255 – Programmable IO
8237/8257 – DMA (Direct Memory Access)

# 8085 Instruction set

## 1. DATA TRANSFER INSTRUCTIONS

| Opcode | Operand | Description |
|---|---|---|
| **Copy From Source to Destination** | | |
| MOV | Rd, Rs<br>M, Rs<br>Rd, M | This instruction copies the contents of the source Register into the destination register; the contents of The source register are not altered. If one of the operands is a memory location, its location is specified by the contents of the HL registers.<br>*Example: MOV B, C or MOV B, M* |
| **Move immediate 8-bit** | | |
| MVI | Rd, data<br>M,data | The 8-bit data is stored in the destination register or memory. If the operand is a memory location, its location is specified by the contents of the HL registers.<br>*Example: MVI B, 57 or MVI M, 57* |
| **Load accumulator** | | |
| LDA | 16-bit address | The contents of a memory location, specified by a 16-bit address in the operand, are copied to the accumulator. The contents of the source are not altered.<br>*Example: LDA 2034 or LDA XYZ* |
| **Load accumulator indirect** | | |
| LDAX | B/D Reg. pair | The contents of the designated register pair point to a memory location. This instruction copies the contents of that memory location into the accumulator. The contents of either the register pair or the memory location are not altered.<br>*Example: LDAX B* |
| **Load register pair immediate** | | |
| LXI | Reg. pair, 16-bit data | The instruction loads 16-bit data in the register pair designated in the operand.<br>*Example: LXI H, 2034* |
| **Load H and L registers direct** | | |
| LHLD | 16-bit address | The instruction copies the contents of the memory location pointed out by the 16-bit address into register L and copies the contents of the next memory location into register H. The contents of source memory locations are not altered.<br>*Example: LHLD 2040* |
| **Store accumulator direct** | | |
| STA | 16-bit address | The contents of the accumulator are copied into the memory location specified by the operand. This is a 3-byte instruction, the second byte specifies the low-order address and the third byte specifies the high-order address.<br>*Example: STA 4350 or STA XYZ* |
| **Store accumulator indirect** | | |
| STAX | Reg. Pair | The contents of the accumulator are copied into the memory location specified by the contents of the operand (register pair). The contents of the accumulator are not altered.<br>*Example: STAX B* |
| **Store H and L registers direct** | | |
| SHLD | 16-bit address | The contents of register L are stored into the memory location specified by the 16-bit address in the operand and the contents of H register are stored into the next memory location by incrementing the operand. The contents of registers HL are not altered. This is a 3-byte instruction, the second byte specifies the low-order address and the third byte specifies the high-order |

| | | |
|---|---|---|
| | | address.<br>*Example: SHLD 2470* |
| **Opcode** | **Operand** | **Description** |
| **Exchange H and L with D and E** | | |
| XCHG | None | The contents of register H are exchanged with the contents of register D, and the contents of register L are exchanged with the contents of register E.<br>*Example: XCHG* |
| **Copy H and L registers to the stack pointer** | | |
| SPHL | none | The instruction loads the contents of the H and L registers into the stack pointer register, the contents of the H register provide the high-order address and the contents of the L register provide the low-order address. The contents of the H and L registers are not altered.<br>*Example: SPHL* |
| **Exchange H and L with top of stack** | | |
| XTHL | None | The contents of the L register are exchanged with the stack location pointed out by the contents of the stack pointer register. The contents of the H register are exchanged with the next stack location (SP+1); however, the contents of the stack pointer register are not altered.<br>*Example: XTHL* |
| **Push register pair onto stack** | | |
| PUSH | Reg. pair | The contents of the register pair designated in the operand are copied onto the stack in the following sequence. The stack pointer register is decremented and the contents of the highorder register (B, D, H, A) are copied into that location. The stack pointer register is decremented again and the contents of the low-order egister (C, E, L, flags) are copied to that location.<br>*Example: PUSH B or PUSH A* |
| **Pop off stack to register pair** | | |
| POP | Reg. pair | The contents of the memory location pointed out by the stack pointer register are copied to the low-order register (C, E, L, status flags) of the operand. The stack pointer is incremented by 1 and the contents of that memory location are copied to the high-order register (B, D, H, A) of the operand. The stack pointer register is again incremented by 1.<br>*Example: POP H or POP A* |
| **Output data from accumulator to a port with 8-bit address** | | |
| OUT | 8-bit port address | The contents of the accumulator are copied into the I/O port specified by the operand.<br>*Example: OUT 87* |
| **Input data to accumulator from a port with 8-bit address** | | |
| IN | 8-bit port address | The contents of the input port designated in the operand are read and loaded into the accumulator.<br>*Example: IN 82* |

## 2. ARITHMETIC INSTRUCTIONS

| Opcode | Operand | Description |
|---|---|---|
| **Add register or memory to accumulator** | | |
| ADD | R<br>M | The contents of the operand (register or memory) are added to the contents of the accumulator and the result is stored in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the addition.<br>*Example: ADD B or ADD M* |
| **Add register to accumulator with carry** | | |
| ADC | R<br>M | The contents of the operand (register or memory) and the Carry flag are added to the contents of the accumulator and the result is stored in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the addition.<br>*Example: ADC B or ADC M* |
| **Add immediate to accumulator** | | |
| ADI | 8-bit data | The 8-bit data (operand) is added to the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the addition.<br>*Example: ADI 45* |
| **Add immediate to accumulator with carry** | | |
| ACI | 8-bit data | The 8-bit data (operand) and the Carry flag are added to the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the addition.<br>*Example: ACI 45* |
| **Add register pair to H and L registers** | | |
| DAD | Reg. pair | The 16-bit contents of the specified register pair are added to the contents of the HL register and the sum is stored in the HL register. The contents of the source register pair are not altered. If the result is larger than 16 bits, the CY flag is set. No other flags are affected.<br>*Example: DAD H* |
| **Subtract register or memory from accumulator** | | |
| SUB | R<br>M | The contents of the operand (register or memory) are subtracted from the contents of the accumulator, and the result is stored in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the subtraction.<br>*Example: SUB B or SUB M* |
| **Subtract source and borrow from accumulator** | | |
| SBB | R<br>M | The contents of the operand (register or memory) and the Borrow flag are subtracted from the contents of the accumulator and the result is placed in the accumulator. If the operand is a memory location, its location is specified by the contents of the HL registers. All flags are modified to reflect the result of the subtraction.<br>*Example: SBB B or SBB M* |
| **Subtract immediate from accumulator** | | |
| SUI | 8-bit data | The 8-bit data (operand) is subtracted from the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the subtraction.<br>*Example: SUI 45* |

| Opcode | Operand | Description |
|---|---|---|
| **Subtract immediate from accumulator with borrow** | | |
| SBI | 8-bit data | The 8-bit data (operand) and the Borrow flag are subtracted from the contents of the accumulator and the result is stored in the accumulator. All flags are modified to reflect the result of the subtraction.<br>*Example: SBI 45* |
| **Increment register or memory by 1** | | |
| INR | R<br>M | The contents of the designated register or memory) are incremented by 1 and the result is stored in the same place. If the operand is a memory location, its location is specified by the contents of the HL registers.<br>*Example: INR B or INR M* |
| **Increment register pair by 1** | | |
| INX | R | The contents of the designated register pair are incremented by 1 and the result is stored in the same place.<br>*Example: INX H* |
| **Decrement register or memory by 1** | | |
| DCR | R<br>M | The contents of the designated register or memory are decremented by 1 and the result is stored in the same place. If the operand is a memory location, its location is specified by the contents of the HL registers.<br>*Example: DCR B or DCR M* |
| **Decrement register pair by 1** | | |
| DCX | R | The contents of the designated register pair are decremented by 1 and the result is stored in the same place.<br>Example: DCX H |
| **Decimal adjust accumulator** | | |
| DAA | none | The contents of the accumulator are changed from a binary value to two 4-bit binary coded decimal (BCD) digits. This is the only instruction that uses the auxiliary flag to perform the binary to BCD conversion, and the conversion procedure is described below. S, Z, AC, P, CY flags are altered to reflect the results of the operation.<br>If the value of the low-order 4-bits in the accumulator is greater than 9 or if AC flag is set, the instruction adds 6 to the low-order four bits.<br>If the value of the high-order 4-bits in the accumulator is greater than 9 or if the Carry flag is set, the instruction adds 6 to the high-order four bits.<br>Example: DAA |

## 3. BRANCH INSTRUCTIONS

| Opcode | Operand | Description |
|---|---|---|
| **Jump unconditionally** | | |
| JMP | 16-bit address | The program sequence is transferred to the memory location specified by the 16-bit address given in the operand.<br>*Example: JMP 2034 or JMP XYZ* |
| **Jump conditionally** | | |
| Operand: 16-bit address | | The program sequence is transferred to the memory location specified by the 16-bit address given in the operand based on the specified flag of the PSW as described below.<br>*Example: JZ 2034 or JZ XYZ* |

| Opcode | Description | Flag Status |
|---|---|---|
| JC | Jump on Carry | CY = 1 |
| JNC | Jump on no Carry | CY = 0 |
| JP | Jump on positive | S=0 |
| JM | Jump on minus | S=1 |
| JZ | Jump on zero | Z=1 |
| JNZ | Jump on no zero | Z=0 |
| JPE | Jump on parity even | P=0 |
| JPO | Jump on parity odd | P=1 |

| Opcode | Operand | Description |
|---|---|---|
| **Unconditional subroutine call** | | |
| CALL | 16-bit address | The program sequence is transferred to the memory location specified by the 16-bit address given in the operand. Before the transfer, the address of the next instruction after CALL (the contents of the program counter) is pushed onto the stack.<br>*Example: CALL 2034 or CALL XYZ* |
| **CALL conditionally** | | |
| Operand: 16-bit address | | The program sequence is transferred to the memory location specified by the 16-bit address given in the operand based on the specified flag of the PSW as described below. Before the transfer, the address of the next instruction after the call (the contents of the program counter) is pushed onto the stack.<br>*Example: CZ 2034 or CZ XYZ* |

| Opcode | Description | Flag Status |
|---|---|---|
| CC | Call on Carry | CY = 1 |
| CNC | Call on no Carry | CY = 0 |
| CP | Call on positive | S=0 |
| CM | Call on minus | S=1 |
| CZ | Call on zero | Z=1 |
| CNZ | Call on no zero | Z=0 |
| CPE | Call on parity even | P=0 |
| CPO | Call on parity odd | P=1 |

| Opcode | Operand | Description |
|---|---|---|
| **Return from subroutine unconditionally** | | |
| RET | none | The program sequence is transferred from the subroutine to the calling program. The two bytes from the top of the stack are copied into the program counter, and program execution begins at the new address.<br>**Example: RET** |
| **Return from subroutine conditionally** | | |
| Operand: none | | The program sequence is transferred from the subroutine to the calling program based on the specified flag of the PSW as described below. The two bytes from the top of the stack are copied into the program counter, and program execution begins at the new address.<br>*Example: RZ* |

| Opcode | Description | Flag Status |
|---|---|---|
| RC | Return on Carry | CY = 1 |
| RNC | Return on no Carry | CY = 0 |
| RP | Return on positive | S=0 |
| RM | Return on minus | S=1 |
| RZ | Return on zero | Z=1 |
| RNZ | Return on no zero | Z=0 |
| RPE | Return on parity even | P=0 |
| RPO | Return on parity odd | P=1 |

## 4. LOGICAL INSTRUCTIONS

| Opcode | Operand | Description |
|---|---|---|
| **Compare register or memory with accumulator** | | |
| CMP | R<br>M | The contents of the operand (register or memory) are compared with the contents of the accumulator. Both contents are preserved . The result of the comparison is shown by setting the flags of the PSW as follows:<br>if (A) < (reg/mem): carry flag is set, s=1<br>if (A) = (reg/mem): zero flag is set, s=0<br>if (A) > (reg/mem): carry and zero flags are reset, s=0<br>*Example: CMP B or CMP M* |
| **Compare immediate with accumulator** | | |
| CPI | 8-bit data | The second byte (8-bit data) is compared with the contents of the accumulator. The values being compared remain unchanged. The result of the comparison is shown by setting the flags of the PSW as follows:<br>if (A) < data: carry flag is set, s=1<br>if (A) = data: zero flag is set, s=0<br>if (A) > data: carry and zero flags are reset, s=0<br>*Example: CPI 89* |
| **Logical AND register or memory with accumulator** | | |
| ANA | R<br>M | The contents of the accumulator are logically ANDed with the contents of the operand (register or memory), and the result is placed in the accumulator. If the operand is a memory location, its address is specified by the contents of HL registers. S, Z, P are modified to reflect the result of the operation. CY is reset. AC is set.<br>*Example: ANA B or ANA M* |

| Opcode | Operand | Description |
|---|---|---|
| **Logical AND immediate with accumulator** | | |
| ANI | 8-bit data | The contents of the accumulator are logically ANDed with the 8-bit data (operand) and the result is placed in the accumulator. S, Z, P are modified to reflect the result of the operation. CY is reset. AC is set.<br>Example: ANI 86 |
| **Exclusive OR register or memory with accumulator** | | |
| | | |
| XRA | R<br>M | The contents of the accumulator are Exclusive ORed with M the contents of the operand (register or memory), and the result is placed in the accumulator. If the operand is a memory location, its address is specified by the contents of HL registers. S, Z, P are modified to reflect the result of the operation. CY and AC are reset.<br>*Example: XRA B or XRA M* |
| **Exclusive OR immediate with accumulator** | | |
| XRI | 8-bit data | The contents of the accumulator are Exclusive ORed with the 8-bit data (operand) and the result is placed in the accumulator. S, Z, P are modified to reflect the result of the operation. CY and AC are reset.<br>*Example: XRI 86* |
| **Logical OR register or memory with accumulator** | | |
| ORA | R<br>M | The contents of the accumulator are logically ORed with M the contents of the operand (register or memory), and the result is placed in the accumulator. If the operand is a memory location, its address is specified by the contents of HL registers. S, Z, P are modified to reflect the result of the operation. CY and AC are reset.<br>*Example: ORA B or ORA M* |
| **Logical OR immediate with accumulator** | | |
| ORI | 8-bit data | The contents of the accumulator are logically ORed with the 8-bit data (operand) and the result is placed in the accumulator. S, Z, P are modified to reflect the result of the operation. CY and AC are reset.<br>*Example: ORI 86* |
| **Rotate accumulator left** | | |
| RLC | none | Each binary bit of the accumulator is rotated left by one position. Bit D7 is placed in the position of D0 as well as in the Carry flag. CY is modified according to bit D7. S, Z, P, AC are not affected.<br>*Example: RLC* |
| **Rotate accumulator right** | | |
| RRC | none | Each binary bit of the accumulator is rotated right by one position. Bit D0 is placed in the position of D7 as well as in the Carry flag. CY is modified according to bit D0. S, Z, P, AC are not affected.<br>*Example: RRC* |
| **Rotate accumulator left through carry** | | |
| RAL | None | Each binary bit of the accumulator is rotated left by one position through the Carry flag. Bit D7 is placed in the Carry flag, and the Carry flag is placed in the least significant position D0. CY is modified according to bit D7. S, Z, P, AC are not affected.<br>*Example: RAL* |
| **Rotate accumulator right through carry** | | |
| RAR | none | RAR none Each binary bit of the accumulator is rotated right by one position through the Carry flag. Bit D0 is placed in the Carry flag, and the Carry flag is placed in the most significant position D7. CY is modified according to bit D0. S, Z, P, AC are not affected.<br>*Example: RAR* |

| Opcode | Operand | Description |
|---|---|---|
| **Complement accumulator** | | |
| CMA | none | The contents of the accumulator are complemented. No flags are affected.<br>*Example: CMA* |
| **Complement carry** | | |
| CMC | none | The Carry flag is complemented. No other flags are affected.<br>*Example: CMC* |
| **Set Carry** | | |
| STC | none | The Carry flag is set to 1. No other flags are affected.<br>*Example: STC* |

## 5. CONTROL INSTRUCTIONS

| Opcode | Operand | Description |
|---|---|---|
| **No operation** | | |
| NOP | none | No operation is performed. The instruction is fetched and decoded. However no operation is executed.<br>*Example: NOP* |
| **Halt and enter wait state** | | |
| HLT | None | The CPU finishes executing the current instruction and halts any further execution. An interrupt or reset is necessary to exit from the halt state.<br>*Example: HLT* |
| **Disable interrupts** | | |
| DI | none | The interrupt enable flip-flop is reset and all the interrupts except the TRAP are disabled. No flags are affected.<br>*Example: DI* |
| **Enable interrupts** | | |
| EI | none | The interrupt enable flip-flop is set and all interrupts are enabled. No flags are affected. After a system reset or the acknowledgement of an interrupt, the interrupt enable flip-flop is reset, thus disabling the interrupts. This instruction is necessary to reenable the interrupts (except TRAP).<br>*Example: EI* |

8085
Instruction Set with Machine Cycles and Flag Status (see notes at end of table)

| Instruction | | | Code[1] | B/M/T[2] | Machine[3] Cycles | Flags[4] S $D_7$ | Z $D_6$ | AC $D_4$ | P $D_2$ | CY $D_0$ |
|---|---|---|---|---|---|---|---|---|---|---|
| ACI | DATA | : Add 8-bit and CY to A | CE data | 2/2/7 | F R | ✓ | ✓ | ✓ | ✓ | ✓ |
| ADC | REG | : Add Reg. and CY to A | 1000 1SSS | 1/1/4 | F | ✓ | ✓ | ✓ | ✓ | ✓ |
| ADC | M | : Add Mem. and CY to A | 8E | 1/2/7 | F R | ✓ | ✓ | ✓ | ✓ | ✓ |
| ADD | REG | : Add Reg. to A | 1000 0SSS | 1/1/4 | F | ✓ | ✓ | ✓ | ✓ | ✓ |
| ADD | M | : Add Mem. to A | 86 | 1/2/7 | F R | ✓ | ✓ | ✓ | ✓ | ✓ |
| ADI | DATA | : ADD 8-BIT TO A | C6 DATA | 2/2/7 | F R | ✓ | ✓ | ✓ | ✓ | ✓ |
| ANA | REG | : AND Reg. with A | 1010 0SSS | 1/1/4 | F | ✓ | ✓ | 1 | ✓ | 0 |
| ANA | M | : AND Mem. with A | A6 | 1/2/7 | F R | ✓ | ✓ | 1 | ✓ | 0 |
| ANI | DATA | : AND 8-bit with A | E6 data | 2/2/7 | F R | ✓ | ✓ | 1 | ✓ | 0 |
| CALL | ADDR | : Call Unconditional | CD addr | 3/5/18 | S R R W W | | | | | |
| CC | ADDR | : Call On CY | DC addr | 3/5/9–18 | S R R W W | | | | | |
| CM | ADDR | : Call On Minus | FC addr | 3/5/9–18 | S R R W W | | | | | |
| CMA | | : Complement A | 2F | 1/1/4 | F | | | | | |
| CMC | | : Complement CY | 3F | 1/1/4 | F | | | | | ✓ |
| CMP | REG | : Compare Reg. with A | 1011 1SSS | 1/1/4 | F | ✓ | ✓ | ✓ | ✓ | ✓ |
| CMP | M | : Compare Mem. with A | BE | 1/2/7 | F R | ✓ | ✓ | ✓ | ✓ | ✓ |
| CNC | ADDR | : Call On No CY | D4 addr | 3/5/9–18 | S R R W W | | | | | |
| CNZ | ADDR | : Call On No Zero | C4 addr | 3/5/9–18 | S R R W W | | | | | |
| CP | ADDR | : Call On Positive | F4 addr | 3/5/9–18 | S R R W W | | | | | |
| CPE | ADDR | : Call On Parity Even | EC addr | 3/5/9–18 | S R R W W | | | | | |
| CPI | DATA | : Compare 8-bit with A | FE data | 2/2/7 | F R | ✓ | ✓ | ✓ | ✓ | ✓ |
| CPO | ADDR | : Call On Parity Odd | E4 addr | 3/5/9–18 | S R R W W | | | | | |
| CZ | ADDR | : Call On Zero | CC addr | 3/5/9–18 | S R R W W | | | | | |
| DAA | | : Decimal-Adjust A | 27 | 1/1/4 | F | ✓ | ✓ | ✓ | ✓ | ✓ |
| DAD | Rp | : Add Reg. Pair to HL | 00Rp 1001 | 1/3/10 | F B B | | | | | |
| DCR | REG | : Decrement Reg. | 00SS S101 | 1/1/4 | F | ✓ | ✓ | ✓ | ✓ | ✓ |
| DCR | M | : Decrement Mem. Contents | 35 | 1/3/10 | F R W | ✓ | ✓ | ✓ | ✓ | |
| DCX | Rp | : Decrement Reg. Pair | 00Rp 1011 | 1/1/6 | S | | | | | |
| DI | | : Disable Interrupt | F3 | 1/1/4 | F | | | | | |
| EI | | : Enable Interrupt | FB | 1/1/4 | F | | | | | |
| HLT | | : Halt | 76 | 1/2/5 | F B | | | | | |
| IN | PORT | : Input from 8-bit Port | DB data | 2/3/10 | F R I | | | | | |
| INR | REG | : Increment Reg. | 00SS S100 | 1/1/4 | F | ✓ | ✓ | ✓ | ✓ | |
| INR | M | : Increment Mem. Contents | 34 | 1/3/10 | F R W | ✓ | ✓ | ✓ | ✓ | |
| INX | Rp | : Increment Reg. Pair | 00Rp 0011 | 1/1/6 | S | | | | | |
| JC | ADDR | : Jump On Carry | DA addr | 3/3/7–10 | F R R | | | | | |
| JM | ADDR | : Jump On Minus | FA addr | 3/3/7–10 | F R R | | | | | |
| JMP | ADDR | : Unconditional Jump | C3 addr | 3/3/10 | F R R | | | | | |
| JNC | ADDR | : Jump On No Carry | D2 addr | 3/3/7–10 | F R R | | | | | |
| JNZ | ADDR | ; Jump On No Zero | C2 addr | 3/3/7–10 | F R R | | | | | |
| JP | ADDR | : Jump On Positive | F2 addr | 3/3/7–10 | F R R | | | | | |
| JPE | ADDR | : Jump On Parity Even | EA addr | 3/3/7–10 | F R R | | | | | |
| JPO | ADDR | : Jump On Parity Odd | E2 addr | 3/3/7–10 | F R R | | | | | |
| JZ | ADDR | : Jump On Zero | CA addr | 3/3/7–10 | F R R | | | | | |
| LDA | ADDR | : Load A Direct | 3A addr | 3/4/13 | F R R R | | | | | |
| LDAX | Rp | : Load A from M; memory address is in BC/DE | 000X 1010 | 1/2/7 | F R | | | | | |

**Codes[1]**
DDD = Binary digits identifying a destination register
SSS = Binary digits identifying a source register
B = 000, C = 001, D = 010, Memory = 110
E = 001, H = 100, L = 101, A = 111
Rp = Register Pair: BC = 00, HL = 10, DE = 01, SP = 11

**B/M/T[2]**
B = Bytes
M = Machine cycles
T = T-states

**Machine Cycles[3]**
F = Fetch with 4 T-states
S = Fetch with 6 T-states
R = Memory Read
I = I/O Read
W = Memory Write
O = I/O Write
B = Bus Idle

S = Sign
Z = Zero
AC = Auxiliary Carry
P = Parity
CY = Carry

**Flags[4]**
✓ = Flag is modified according to result
0 = Flag is cleared
1 = Flag is set
Blank = No change in flag, remains in previous state

8085

Instruction Set with Machine Cycles and Flag Status (see notes at end of table)

| Instruction | | | Code[1] | B/M/T[2] | Machine[3] Cycles | Flags[4] S $D_7$ | Z $D_6$ | AC $D_4$ | P $D_2$ | CY $D_0$ |
|---|---|---|---|---|---|---|---|---|---|---|
| LHLD | ADDR | : Load HL Direct | 2A addr | 3/5/16 | F R R R R | | | | | |
| LXI | Rp, 16-bit | : Load 16-bit in Reg. Pair | 00Rp 0001 16-bit | 3/3/10 | F R R | | | | | |
| MOV | Rd,Rs | : Move from Reg. $R_s$ to Reg. $R_d$ | 01DD DSSS | 1/1/4 | F | | | | | |
| MOV | M,R | : Move from Reg. to Mem. | 0111 0SSS | 1/2/7 | F W | | | | | |
| MOV | R,M | : Move from Mem. to Reg. | 01DD D110 | 1/2/7 | F R | | | | | |
| MVI | R,DATA | : Load 8-bit in Reg. | 00DD D110 data | 2/2/7 | F R | | | | | |
| MVI | M,DATA | : Load 8-bit in Mem. | 36 data | 2/3/10 | F R W | | | | | |
| NOP | | : No Operation | 00 | 1/1/4 | F | | | | | |
| ORA | R | : OR Reg. with A | 1011 0SSS | 1/1/4 | F | ✓ | ✓ | 0 | ✓ | 0 |
| ORA | M | : OR Mem. Contents with A | B6 | 1/2/7 | F R | ✓ | ✓ | 0 | ✓ | 0 |
| ORI | DATA | : OR 8-bit with A | F6 data | 2/2/7 | F R | ✓ | ✓ | 0 | ✓ | 0 |
| OUT | PORT | : Output to 8-bit Port | D3 data | 2/3/10 | F R O | | | | | |
| PCHL | | : Move HL to Program Counter | E9 | 1/1/6 | S | | | | | |
| POP | Rp | : Pop Reg. Pair | 11Rp 0001 | 1/3/10 | F R R | | | | | |
| PUSH | Rp | : Push Reg. Pair | 11Rp 0101 | 1/3/12 | S W W | | | | | |
| RAL | | : Rotate A Left through CY | 17 | 1/1/4 | F | | | | | ✓ |
| RAR | | : Rotate A Right through CY | 1F | 1/1/4 | F | | | | | ✓ |
| RC | | : Return On Carry | D8 | 1/3/6–12 | S R R | | | | | |
| RET | | : Return | C9 | 1/3/10 | F R R | | | | | |
| RIM | | : Read Interrupt Mask | 20 | 1/1/4 | F | | | | | |
| RLC | | : Rotate A Left | 07 | 1/1/4 | F | | | | | ✓ |
| RM | | : Return On Minus | F8 | 1/3/6–12 | S R R | | | | | |
| RNC | | : Return On No Carry | D0 | 1/3/6–12 | S R R | | | | | |
| RNZ | | : Return On No Zero | C0 | 1/3/6–12 | S R R | | | | | |
| RP | | : Return On Positive | F0 | 1/3/6–12 | S R R | | | | | |
| RPE | | : Return On Parity Even | E8 | 1/3/6–12 | S R R | | | | | |
| RPO | | : Return On Parity Odd | E0 | 1/3/6–12 | S R R | | | | | |
| RRC | | : Rotate A to Right | 0F | 1/1/4 | F | | | | | ✓ |
| RST | N | : Restart | 11XX X111 | 1/3/12 | S W W | | | | | |
| RZ | | : Return On Zero | C8 | 1/3/6–12 | S R R | | | | | |
| SBB | R | : Subtract Reg. from A with Borrow | 1001 1SSS | 1/1/4 | F | ✓ | ✓ | ✓ | ✓ | ✓ |
| SBB | M | : Subtract Mem. Contents from A with Borrow | 9E | 1/2/7 | F R | ✓ | ✓ | ✓ | ✓ | ✓ |
| SBI | DATA | : Subtract 8-bit from A | DE data | 2/2/7 | F R | ✓ | ✓ | ✓ | ✓ | ✓ |
| SHLD | ADDR | : Store HL Direct | 22 addr | 3/5/16 | F R R W W | | | | | |
| SIM | | : Set Interrupt Mask | 30 | 1/1/4 | F | | | | | |
| SPHL | | : Move HL to Stack Pointer | F9 | 1/1/6 | S | | | | | |
| STA | ADDR | : Store A Direct | 32 addr | 3/4/13 | F R R W | | | | | |
| STAX | Rp | : Store A in M, memory address is in BC/DE | 000X 0010 | 1/2/7 | F W | | | | | |
| STC | | : Set Carry | 37 | 1/1/4 | F | | | | | 1 |
| SUB | R | : Subtract Reg. from A | 1001 0SSS | 1/1/4 | F | ✓ | ✓ | ✓ | ✓ | ✓ |
| SUB | M | : Subtract Mem. from A | 96 | 1/2/7 | F R | ✓ | ✓ | ✓ | ✓ | ✓ |
| SUI | DATA | : Subtract 8-bit from A | D6 data | 2/2/7 | F R | ✓ | ✓ | ✓ | ✓ | ✓ |
| XCHG | | : Exchange DE with HL | EB | 1/1/4 | F | | | | | |
| XRA | R | : Exclusive OR Reg. with A | 1010 1SSS | 1/1/4 | F | ✓ | ✓ | 0 | ✓ | 0 |
| XRA | M | : Exclusive OR Mem. with A | AE | 1/2/7 | F R | ✓ | ✓ | 0 | ✓ | 0 |
| XRI | DATA | : Exclusive OR 8-bit with A | EE data | 2/2/7 | F R | ✓ | ✓ | 0 | ✓ | 0 |
| XTHL | | : Exchange Stack with HL | E3 | 1/4/16 | F R R W W | | | | | |

**Codes[1]**
DDD = Binary digits identifying a destination register
SSS = Binary digits identifying a source register
B = 000, C = 001, D = 010, Memory = 110
E = 001, H = 100, L = 101, A = 111
Rp = Register Pair BC = 00, HL = 10 DE = 01, SP = 11

**B/M/T[2]**
B = Bytes
M = Machine cycles
T = T-states

**Machine Cycles[3]**
F = Fetch with 4 T-states
S = Fetch with 6 T-states
R = Memory Read
I = I/O Read
W = Memory Write
O = I/O Write
B = Bus Idle

S = Sign
Z = Zero
AC = Auxiliary Carry
P = Parity
CY = Carry

**Flags[4]**
✓ = Flag is modified according to result
0 = Flag is cleared
1 = Flag is set
Blank = No change in flag, remains in previous state

**2. Memory Organization**

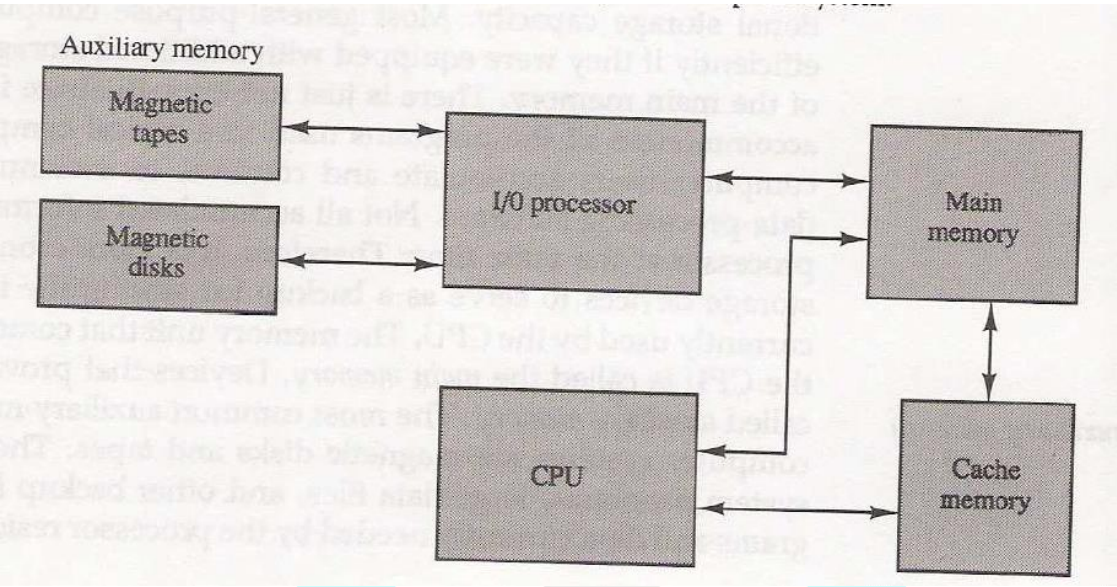
**2.1. Memory Hierarchy**

**2.2. Memory Management**

## 2.1. Memory Hierarchy

Memory is used to store the information in digital form. The memory hierarchy is given by:

- Register
- Cache Memory
- Main Memory (ROM,RAM)
- Auxiliary Memory (Magnetic Disk)
- Removable media (Magnetic tape)




The main goal of using memory hierarchy is to obtain the highest possible average access speed while minimizing the total cost of the entire memory system.

### 2.1.1 Main Memory (or) Primary Memory:

The memory unit that communicate directly with the CPU is called "Main Memory". The main memory of a computer is semiconductor memory. The main memory unit of computer is basically consists of two kinds of memory:
**RAM:** Random access memory; which is volatile in nature.
**ROM:** Read only memory; which is non-volatile.

The permanent information are kept in ROM and the user space is basically in RAM. The smallest unit of information is known as bit (binary digit), and in one memory cell we can store one bit of information. 8 bit together is termed as a byte.
**Internal Organization of memory chips:** A memory cell is capable of storing 1-bit of information. A number of memory cells are organized in the form of a matrix to form the memory chip. One such organization is shown in the Figure 2.1.

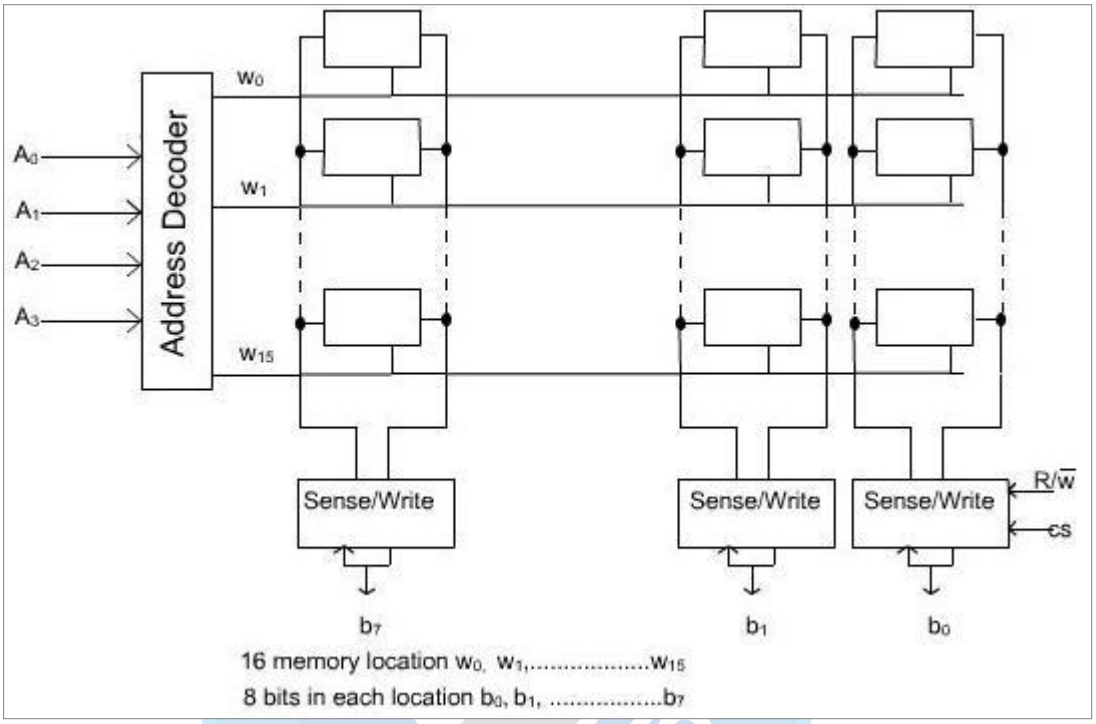
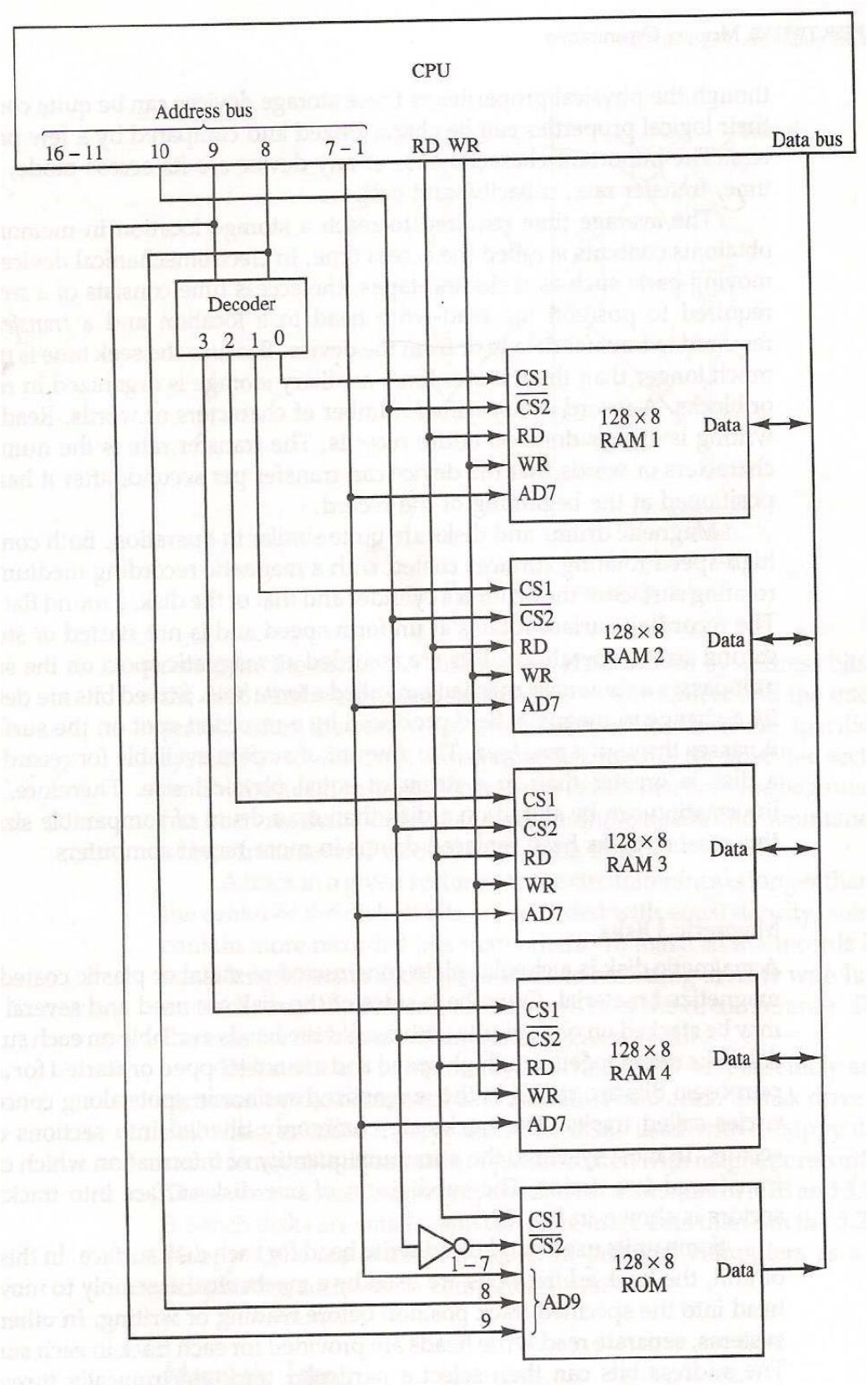



**Fig 2.1: Internal organization of Memory.**

The maximum size of main memory that can be used in any computer is determined by the addressing scheme. A computer that generates 16-bit address is capable of addressing up to $2^{16}$ which is equal to 64K memory location. Similarly, for 32 bit addresses, the total capacity will be $2^{32}$ which is equal to 4G memory location.

A computer employs RAM chips of 128x8 and ROM chips of 512x8. The computer system needs 512 bytes of RAM, 512 bytes of ROM.

| Component | Hexadecimal address | Address bus 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| RAM 1 | 0000–007F | 0 | 0 | 0 | x | x | x | x | x | x | x |
| RAM 2 | 0080–00FF | 0 | 0 | 1 | x | x | x | x | x | x | x |
| RAM 3 | 0100–017F | 0 | 1 | 0 | x | x | x | x | x | x | x |
| RAM 4 | 0180–01FF | 0 | 1 | 1 | x | x | x | x | x | x | x |
| ROM | 0200–03FF | 1 | x | x | x | x | x | x | x | x | x |




**2.1.2. Auxiliary Memory (or) Secondary Memory:** Device that provides backup storage are called "Auxiliary Memory". Example: Magnetic disks, Hard Disks, CD ROM etc.

**3. I/O Organization:**

- **3.1. Introduction to I/O organization**
- **3.2. Programmed Control I/O**
- **3.3. Interrupt Driven I/O**
- **3.4. Direct Memory Access**

## 3.1. Introduction to I/O organization

**Input/Output Organization:**

- The computer system's input/output (I/O) architecture is its interface to the outside world.
- Till now we have discussed the two important modules of the computer system -
  - The processor and
  - The memory module.
- The third key component of a computer system is a set of I/O modules.
- Each I/O module interfaces to the system bus and controls one or more peripheral devices.

There are several reasons why an I/O device or peripheral device is not directly connected to the system bus. Some of them are as follows -

- There are a wide variety of peripherals with various methods of operation. It would be impractical to include the necessary logic within the processor to control several devices.
- The data transfer rate of peripherals is often much slower than that of the memory or processor. Thus, it is impractical to use the high-speed system bus to communicate directly with a peripheral.
- Peripherals often use different data formats and word lengths than the computer to which they are attached.

Thus, an **I/O module** (or) **Device controller** (or) **I/O controller** is required.

**Input/Output Modules:**

The major functions of an I/O module are categorized as follows –

- Control and timing
- Processor Communication
- Device Communication
- Data Buffering
- Error Detection

During any period of time, the processor may communicate with one or more external devices in unpredictable manner, depending on the program's need for I/O.

The internal resources, such as main memory and the system bus, must be shared among a number of activities, including data I/O.

**Control & timings:**

The I/O function includes a control and timing requirement to co-ordinate the flow of traffic between internal resources and external devices.

For example, the control of the transfer of data from an external device to the processor might involve the following sequence of steps –

1. The processor interacts with the I/O module to check the status of the attached device.
2. The I/O module returns the device status.
3. If the device is operational and ready to transmit, the processor requests the transfer of data, by means of a command to the I/O module.
4. The I/O module obtains a unit of data from external device.
5. The data are transferred from the I/O module to the processor.

If the system employs a bus, then each of the interactions between the processor and the I/O module involves one or more bus arbitrations.

**Processor & Device Communication**

During the I/O operation, the I/O module must communicate with the processor and with the external device.

Processor communication involves the following -

1. **Command decoding:** I/O module accepts command from the processor, typically sent as signals on control bus.
2. **Data:** Data are exchanged between the processor and the I/O module over the data bus.
3. **Status Reporting:** peripherals are so slow, it is important to know the status of the I/O module. For example, if an I/O module is asked to send data to the processor (read), it may not be ready to do so because it is still working on the previous I/O command. This fact can be reported with a status signal. Common status signals are **BUSY** and **READY**.
4. **Address Recognition:** Just as each word of memory has an address, so thus each of the I/O devices. Thus an I/O module must recognize one unique address for each peripheral it controls.

On the other hand, the I/O must be able to perform device communication. This communication involves command, status information and data.

**Data Buffering:**

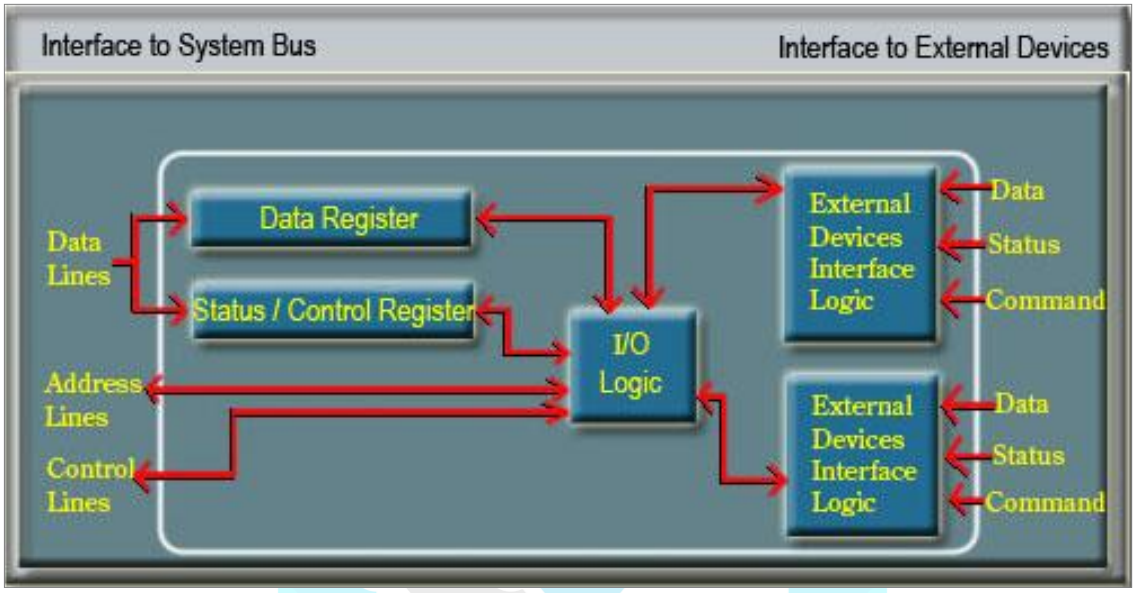
- An essential task of an I/O module is data buffering. The data buffering is required due to the mismatch of the speed of CPU, memory and other peripheral devices. In general, the speed of CPU is higher than the speed of the other peripheral devices. So, the I/O modules store the data in a data buffer and regulate the transfer of data as per the speed of the devices.

- In the opposite direction, data are buffered so as not to tie up the memory in a slow transfer operation. Thus the I/O module must be able to operate at both device and

memory speed.

**Error Detection:**

Another task of I/O module is error detection and for subsequently reporting error to the processor. One class or error includes mechanical and electrical malfunctions reported by the device (e.g. paper jam). Another class consists of unintentional changes to the bit pattern as it is transmitted from devices to the I/O module.




Block diagram of I/O Module is shown in the Figure.

**I/O System:**

- There will be many I/O devices connected through I/O modules to the system. Each device will be identified by a unique address.
- When the processor issues an I/O command, the command contains the address of the device that is used by the command. The I/O module must interpret the addres lines to check if the command is for itself.
- Generally in most of the processors, the processor, main memory and I/O share a common bus (data address and control bus).

Two types of addressing are possible -

1. Memory-mapped I/O
2. Isolated or I/O mapped I/O

**Memory-mapped I/O:**

- There is a single address space for memory locations and I/O devices.
- The processor treats the status and address register of the I/O modules as memory location.
- For example, if the size of address bus of a processor is 16, then there are 216 combinations and all together 216 address locations can be addressed with

these 16 address lines.

- Out of these 216 address locations, some address locations can be used to address I/O devices and other locations are used to address memory locations.
- Since I/O devices are included in the same memory address space, so the status and address registers of I/O modules are treated as memory location by the processor. Therefore, the same machine instructions are used to access both memory and I/O devices

**Isolated or I/O -mapped I/O:**

In this scheme, the full range of addresses may be available for both.
The address refers to a memory location or an I/O device is specified with the help of a command line.

In general $IO/\overline{M}$ command line is used to identify a memory location or an I/O device.

if $IO/\overline{M}$ =1, it indicates that the address present in address bus is the address of an I/O device.

if $IO/\overline{M}$ =0, it indicates that the address present in address bus is the address of a memory location.

Since full range of address is available for both memory and I/O devices, so, with 16 address lines, the system may now support both 2 16 memory locations and 2 16 I/O addresses.

| **Characteristics** | **Memory-Mapped I/O** | **I/O Mapped I/O** |
|---|---|---|
| Control signals | MEMR,MEMW | IOR,IOW |
| Instructions available | Memory related Instructions like MOV M,R | IN,OUT instructions |
| Data transfer | Between any register and I/O | Only between I/O and the accumulator. |
| Hardware requirements | More hardware needed to decode 16-bit address | Less Hardware is needed to decode 8-bit address. |

**Input / Output Subsystem**

There are three basic forms of input and output systems –

1. Programmed I/O
2. Interrupt driven I/O
3. Direct Memory Access (DMA).

- With programmed I/O, the processor executes a program that gives its direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data.
- With interrupt driven I/O, the processor issues an I/O command, continues to execute other instructions, and is interrupted by the I/O module when the I/O module completes its work.
- In Direct Memory Access (DMA), the I/O module and main memory exchange data

directly without processor involvement.

### 3.2. Programmed Control I/O

With both programmed I/O and Interrupt driven I/O, the processor is responsible for extracting data from main memory for output operation and storing data in main memory for input operation.

To send data to an output device, the CPU simply moves that data to a special memory location in the I/O address space if I/O mapped input/output is used or to an address in the memory address space if memory mapped I/O is used.

| | | |
|---|---|---|
| Data | I/O Address Space (in memory) | if I/O mapped input/output is used |
| | memory address space | if memory mapped I/O is used |

To read data from an input device, the CPU simply moves data from the address (I/O or memory) of that device into the CPU.

**Input/Output Operation:** The input and output operation looks very similar to a memory read or write operation except it usually takes more time since peripheral devices are slow in speed than main memory modules.

The working principle of the three methods for input of a Block of Data is shown in the Figure 3.1.

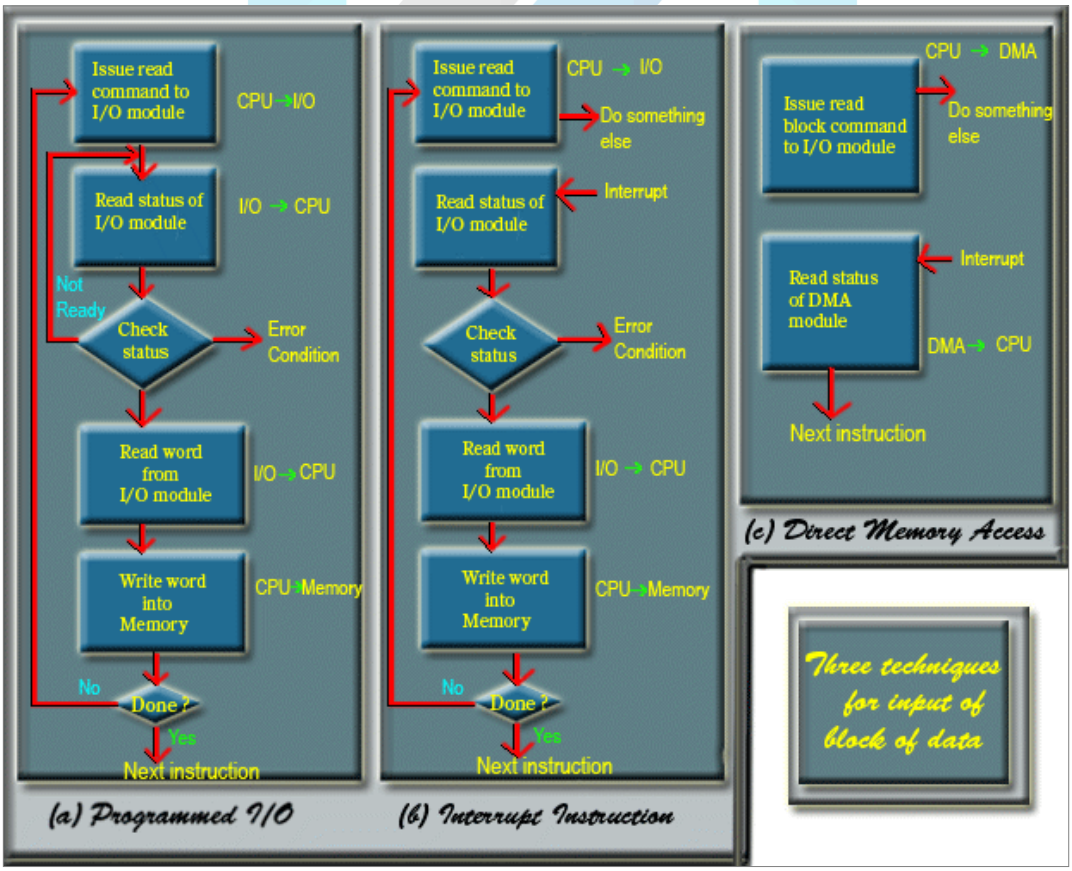



Fig 3.1 Working of three techniques for input of block of data

| | | Number of cycles |
|---|---|---|
| BACK: | MOV AL,[SI] | 10 |
| | OUT PORT,AL | 10 |
| | INC SI | 02 |
| | LOOP BACK | 17 |
| | Total | 39 |

**Input/Output Port**

- An I/O port is a device that looks like a memory cell to the computer but contains connection to the outside world.
- An I/O port typically uses a latch. When the CPU writes to the address associated with the latch, the latch device captures the data and makes it available on a set of wires external to the CPU and memory system.
- The I/O ports can be read-only, write-only, or read/write. The write-only port is shown in the Figure3.2.

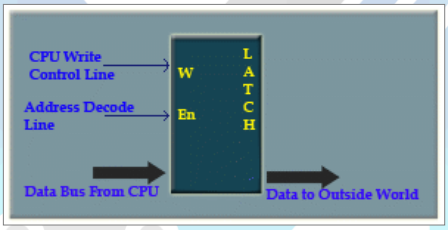



Fig 3.2: The write only port

- First, the CPU will place the address of the device on the I/O address bus and with the help of address decoder a signal is generated which will enable the latch.
- Next, the CPU will indicate the operation is a write operation by putting the appropriate signal in CPU write control line.
- Then the data to be transferred will be placed in the CPU bus, which will be stored in the latch for the onward transmission to the device.
- Both the address decode and write control lines must be active for the latch to operate.
- The read/write or input/output port is shown in the Figure3.3.

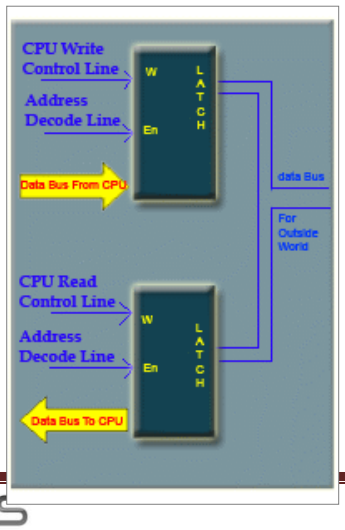


Fig3.3:Read/Write Port

- The device is identified by putting the appropriate address in the I/O address lines. The address decoder will generate the signal for the address decode lines. According to the operation, read or write, it will select either of the latch.
- If it is a write operation, then data will be placed in the latch from CPU for onward transmission to the output device.
- If it is in a read operation, the data that are already stored in the latch will be transferred to the CPU.
  **A read only (input) port is simply the lower half of the Figure 3.3.**
- In case of I/O mapped I/O, a different address space is used for I/O devices. The address space for memory is different. In case of memory mapped I/O, same address space is used for both memory and I/O devices. Some of the memory address space is kept reserved for I/O devices.
- To the programmer, the difference between I/O-mapped and memory-mapped input/output operation is the instruction to be used.
  **For memory-mapped I/O, any instruction that accessed memory can access a memory-mapped I/O port.**
  **I/O-mapped input/output uses special instruction to access I/O port.**
- Generally, a given peripheral device will use more than a single I/O port. A typical *PC* parallel printer interface, for example, uses three ports, a read/write port, and input port and an output port.
- The read/write port is the data port ( it is read/write to allow the CPU to read the last ASCII character it wrote to the printer port ).
  The input port returns control signals from the printer.
  - These signals indicate whether the printer is ready to accept another character, is off-line, is out of paper, etc.
  The output port transmits control information to the printer such as
  - whether data is available to print.

Memory-mapped I/O subsystems and I/O-mapped subsystems both require the CPU to move data between the peripheral device and main memory.
For example, to input a sequence of 20 bytes from an input port and store these bytes into memory, the CPU must send each value and store it into memory.

**Programmed I/O:**

In programmed I/O, the data transfer between CPU and I/O device is carried out with the help of a software routine.
When a processor is executing a program and encounters an instruction relating to I/O, it executes that I/O instruction by issuing a command to the appropriate I/O module.
The I/O module will perform the requested action and then set the appropriate bits in the I/O status register.
The I/O module takes no further action to alert the processor.
It is the responsibility of the processor to check periodically the status of the I/O module until it finds that the operation is complete.
In programmed I/O, when the processor issues a command to a I/O module, it must wait

until the I/O operation is complete.
Generally, the I/O devices are slower than the processor, so in this scheme CPU time is wasted. CPU is checking the status of the I/O module periodically without doing any other work.

**I/O Commands**

To execute an I/O-related instruction, the processor issues an address, specifying the particular I/O module and external device, and an I/O command. There are four types of I/O commands that an I/O module will receive when it is addressed by a processor –

- **Control :** Used to activate a peripheral device and instruct it what to do. For example, a magnetic tape unit may be instructed to rewind or to move forward one record. These commands are specific to a particular type of peripheral device.
- **Test :** Used to test various status conditions associated with an I/O module and its peripherals. The processor will want to know if the most recent I/O operation is completed or any error has occurred.
- **Read :** Causes the I/O module to obtain an item of data from the peripheral and place it in the internal buffer.
- **Write :** Causes the I/O module to take an item of data ( byte or word ) from the data bus and subsequently transmit the data item to the peripheral.

**3.3. Interrupt Control I/O**

**Interrupt driven I/O**

- The problem with programmed I/O is that the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data. The processor, while waiting, must repeatedly interrogate the status of the I/O module.
- This type of I/O operation, where the CPU constantly tests a part to see if data is available, is polling, that is, the CPU Polls (asks) the port if it has data available or if it is capable of accepting data. Polled I/O is inherently inefficient.
- The solution to this problem is to provide an interrupt mechanism. In this approach the processor issues an I/O command to a module and then go on to do some other useful work. The I/O module then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer. Once the data transfer is over, the processor then resumes its former processing.

**Let us consider how it works**

**A. From the point of view of the I/O module:**

- For input, the I/O module services a READ command from the processor.
- The I/O module then proceeds to read data from an associated peripheral device.
- Once the data are in the modules data register, the module issues an interrupt

to the processor over a control line.
- The module then waits until its data are requested by the processor.
- When the request is made, the module places its data on the data bus and is then ready for another I/O operation.

**B. From the processor point of view; the action for an input is as follows:**

- The processor issues a READ command.
- It then does something else (e.g. the processor may be working on several different programs at the same time).
- At the end of each instruction cycle, the processor checks for interrupts
- When the interrupt from an I/O module occurs, the processor saves the context (e.g. program counter & processor registers) of the current program and processes the interrupt.
- In this case, the processor reads the word of data from the I/O module and stores it in memory.
- It then restores the context of the program it was working on and resumes execution

**Interrupt Processing**

The occurrence of an interrupt triggers a number of events, both in the processor hardware and in software.
When an I/O device completes an I/O operation, the following sequences of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt.
3. The processor tests for the interrupt; if there is one interrupt pending, then the processor sends an acknowledgement signal to the device which issued the interrupt. After getting acknowledgement, the device removes its interrupt signals.
4. The processor now needs to prepare to transfer control to the interrupt routine. It needs to save the information needed to resume the current program at the point of interrupt. The minimum information required to save is the processor status word (PSW) and the location of the next instruction to be executed which is nothing but the contents of program counter. These can be pushed into the system control stack.
5. The processor now loads the program counter with the entry location of the interrupt handling program that will respond to the interrupt.

**Interrupt Processing:**

1. An interrupt occurs when the processor is executing the instruction of location *N*.
2. At that point, the value of program counter is *N+1* .
3. Processor services the interrupt after completion of current instruction execution.
4. First, it moves the content of general registers to system stack.
5. Then it moves the program counter value to the system stack.
6. Top of the system stack is maintained by stack pointer.
7. The value of stack pointer is modified to point to the top of the stack.
8. If M elements are moved to the system stack, the value of stack pointer is changed

from *T* to *T-M*.

9. Next, the program counter is loaded with the starting address of the interrupt service routine.
10. Processor starts executing the interrupt service routine.

The data changes of memory and registers during interrupt service is shown in the Figure3.4.

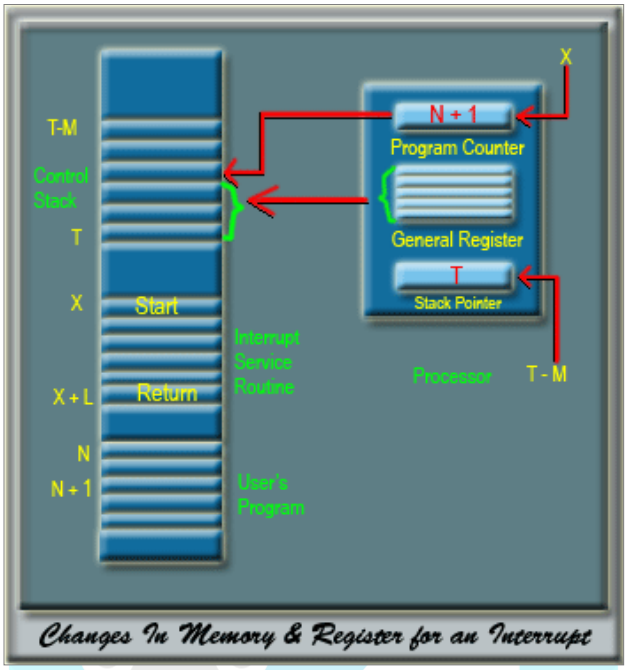



Fig3.4: Changes of memory and register for an interrupt

**Return from Interrupt:**

1. Interrupt service routine starts at location *X* and the return instruction is in location *X + L*.
2. After fetching the return instruction, the value of program counter becomes *X + L + 1*.
3. While returning to user's program, processor must restore the earlier values.
4. From control stack, it restores the value of program counter and the general registers.
5. Accordingly it sets the value of the top of the stack and accordingly stack pointer is updated.
6. Now the processor starts execution of the user's program (interrupted program) from memory location *N + 1*.

The data changes of memory and registers during return from and interrupt is shown in the Figure 3.5.

Once the program counter has been loaded, the processor proceeds to the next instruction cycle, which begins with an interrupt fetch. The control will transfer to interrupt handler routine for the current interrupt.

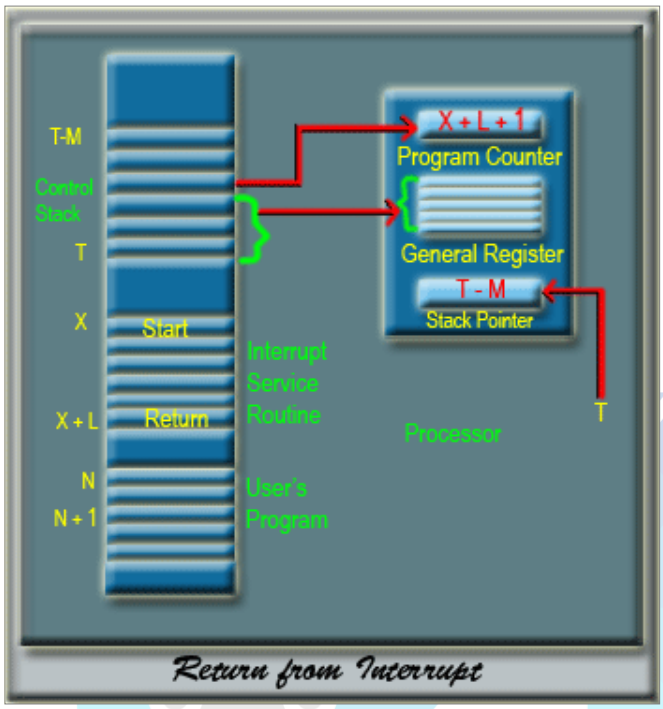



Fig 3.5: Return from interrupt

The following operations are performed at this point.

6. At the point, the program counter and PSW relating to the interrupted program have been saved on the system stack. In addition to that some more information must be saved related to the current processor state which includes the control of the processor registers, because these registers may be used by the interrupt handler. Typically, the interrupt handler will begin by saving the contents of all registers on stack.
7. The interrupt handles next processes the interrupt. This includes an examination of status information relating to the I/O operation or, other event that caused an interrupt.
8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers.
9. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

**Design Issues for Interrupt**

Two design issues arise in implementing interrupt I/O.

- There will almost invariably be multiple I/O modules, how does the processor determine which device issued the interrupt?
- If multiple interrupts have occurred how the processor does decide which one to process?

**Device Identification**

Four general categories of techniques are in common use:

1. **Multiple interrupt lines**
2. **Software poll**
3. **Daisy chain (hardware poll, vectored)**
4. **Bus arbitration ( vectored)**

**Multiple Interrupts Lines:**

- The most straight forward approach is to provide multiple interrupt lines between the processor and the I/O modules.
- It is impractical to dedicate more than a few bus lines or processor pins to interrupt lines.
- Thus, though multiple interrupt lines are used, it is most likely that each line will have multiple I/O modules attached to it. Thus one of the other three techniques must be used on each line.

**Software Poll :**

- When the processor detects an interrupt, it branches to an interrupt service routine whose job is to poll each I/O module to determine which module caused the interrupt.
- The poll could be implemented with the help of a separate command line (e.g. TEST I/O). In this case, the processor raises TEST I/O and place the address of a particular I/O module on the address lines. The I/O module responds positively if it set the interrupt.
- Alternatively, each I/O module could contain an addressable status register. The processor then reads the status register of each I/O module to identify the interrupting module.
- Once the correct module is identified, the processor branches to a device service routine specific to that device.
- The main disadvantage of software poll is that it is time consuming. Processor has to check the status of each I/O module and in the worst case it is equal to the number of I/O modules.

**Daisy Chain :**

- In this method for interrupts all I/O modules share a common interrupt request lines. However the interrupt acknowledge line is connected in a daisy chain fashion. When the processor senses an interrupt, it sends out an interrupt acknowledgement.
- The interrupt acknowledge signal propagates through a series of I/O module until it gets to a requesting module.

- The requesting module typically responds by placing a word on the data lines. This word is referred to as a vector and is either the address of the I/O module or some other unique identification.
- In either case, the processor uses the vector as a pointer to the appropriate device service routine. This avoids the need to execute a general interrupt service routine first. This technique is referred to as a vectored interrupt. The daisy chain arrangement is shown in the Figure 3.6.

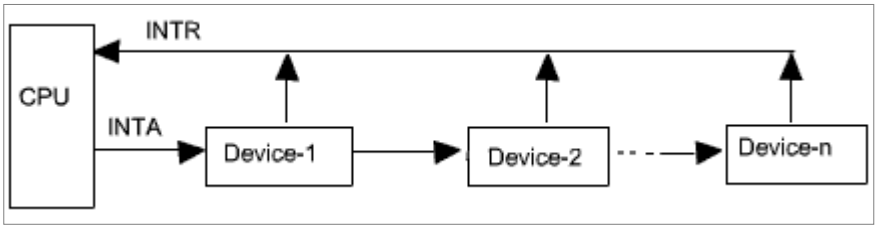



Fig 3.6: Daisy chain arrangement

**Bus Arbitration:**

In bus arbitration method, an I/O module must first gain control of the bus before it can raise the interrupt request line. Thus, only one module can raise the interrupt line at a time. When the processor detects the interrupt, it responds on the interrupt acknowledge line. The requesting module then places it vector on the data line.

**Handling multiple interrupts**

- There are several techniques to identify the requesting I/O module. These techniques also provide a way of assigning priorities when more than one device is requesting interrupt service.
- *With multiple lines*, the processor just picks the interrupt line with highest priority. During the processor design phase itself priorities may be assigned to each interrupt lines.
- *With software polling*, the order in which modules are polled determines their priority.
- *In case of daisy chain configuration*, the priority of a module is determined by the position of the module in the daisy chain. The module nearer to the processor in the chain has got higher priority, because this is the first module to receive the acknowledge signal that is generated by the processor.
- *In case of bus arbitration method*, more than one module may need control of the bus. Since only one module at a time can successfully transmit over the bus, some method of arbitration is needed. The various methods can be classified into two groups – centralized and distributed.

**In a centralized scheme,** a single hardware device, referred to as a bus controller or arbiter is responsible for allocating time on the bus. The device may be a separate module or part of the processor.

**In distributed scheme,** there is no central controller. Rather, each module contains access control logic and the modules act together to share the bus.

It is also possible to combine different device identification techniques to identify the

devices and to set the priorities of the devices. As for example multiple interrupt lines and daisy chain technologies can be combined together to give access for more devices.
In one interrupt line, more than one device can be connected in daisy chain fashion. The High priorities devices should be connected to the interrupt lines that have got higher priority.

A possible arrangement is shown in the Figure 3.7.

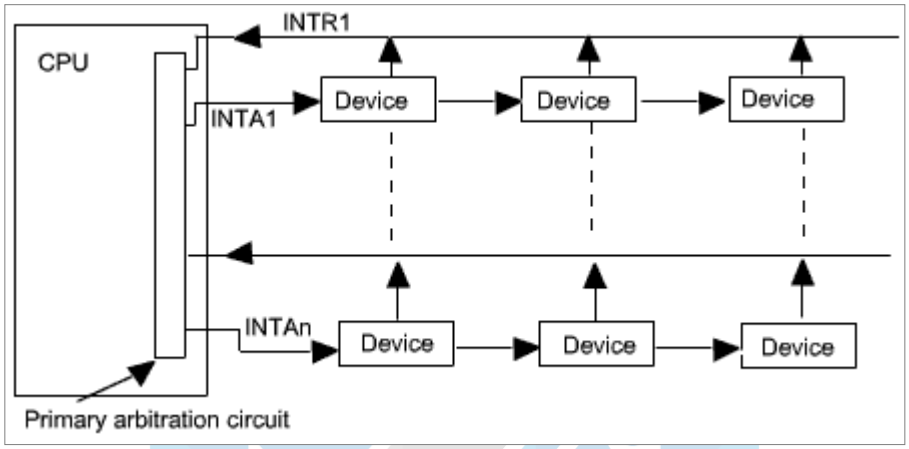



Fig 3.7: Possible arrangement to handle multiple interrupt

**Interrupt Nesting**

- The arrival of an interrupt request from an external device causes the processor to suspend the execution of one program and starts the execution of another. The execution of this another program is nothing but the interrupt service routine for that specified device.
- Interrupt may arrive at any time. So during the execution of an interrupt service routine, another interrupt may arrive. This kind of interrupts are known as nesting of interrupt.
- Whether interrupt nesting is allowed or not? This is a design issue. Generally nesting of interrupt is allowed, but with some restrictions. The common notion is that a high priority device may interrupt a low priority device, but not the vice-versa.
- To accomodate such type of restrictions, all computer provide the programmer with the ability to enable and disable such interruptions at various time during program execution. The processor provides some instructions to enable the interrupt and disable the interrupt. If interrupt is disabled, the CPU will not respond to any interrupt signal.
- On the other hand, when multiple lines are used for interrupt and priorities are assigned to these lines, then the interrupt received in a low priority line will not be served if an interrupt routine is in execution for a high priority device. After completion of the interrupt service routine of high priority devices, processor will respond to the interrupt request of low priority devices.

### 3.4. DMA -Direct Memory Access

We have discussed the data transfer between the processor and I/O devices. We have discussed two different approaches namely programmed I/O and Interrupt-driven I/O.

Both the methods require the active intervention of the processor to transfer data between memory and the I/O module, and any data transfer must transverse a path through the processor. Thus both these forms of I/O suffer from two inherent drawbacks.

- The I/O transfer rate is limited by the speed with which the processor can test and service a device.
- The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

To transfer large block of data at high speed, a special control unit may be provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor. This approach is called **direct memory access** or **DMA**.

DMA transfers are performed by a control circuit associated with the I/O device and this circuit is referred as DMA controller. The DMA controller allows direct data transfer between the device and the main memory without involving the processor.

To transfer data between memory and I/O devices, DMA controller takes over the control of the system from the processor and transfer of data take place over the system bus. For this purpose, the DMA controller must use the bus only when the processor does not need it, or it must force the processor to suspend operation temporarily. The later technique is more common and is referred to as cycle stealing, because the DMA module in effect steals a bus cycle.

The typical block diagram of a DMA controller is shown in the Figure 3.8.

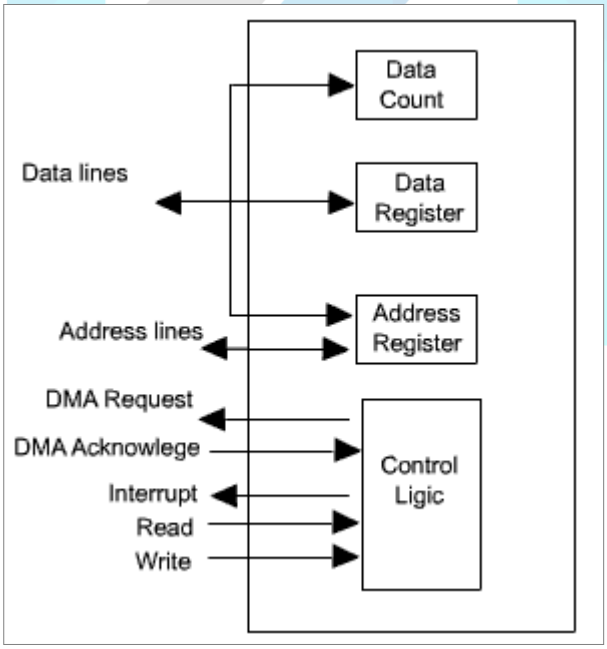



Fig 3.8: Typical DMA block diagram

When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information.

- Whether a read or write is requested, using the read or write control line between the processor and the DMA module.
- The address of the I/O devise involved, communicated on the data lines.
- The starting location in the memory to read from or write to, communicated

on data lines and stored by the DMA module in its address register.

- The number of words to be read or written again communicated via the data lines and stored in the data count register.

The processor then continues with other works. It has delegated this I/O operation to the DMA module.

- The DMA module checks the status of the I/O devise whose address is communicated to DMA controller by the processor. If the specified I/O devise is ready for data transfer, then DMA module generates the DMA request to the processor. Then the processor indicates the release of the system bus through DMA acknowledge.
- The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor.
- When the transfer is completed, the DMA module sends an interrupt signal to the processor. After receiving the interrupt signal, processor takes over the system bus.
- Thus the processor is involved only at the beginning and end of the transfer. During that time the processor is suspended.
- It is not required to complete the current instruction to suspend the processor. The processor may be suspended just after the completion of the current bus cycle. On the other hand, the processor can be suspended just before the need of the system bus by the processor, because DMA controller is going to use the system bus, it will not use the processor.

The point where in the instruction cycle the processor may be suspended shown in the Figure 3.9.

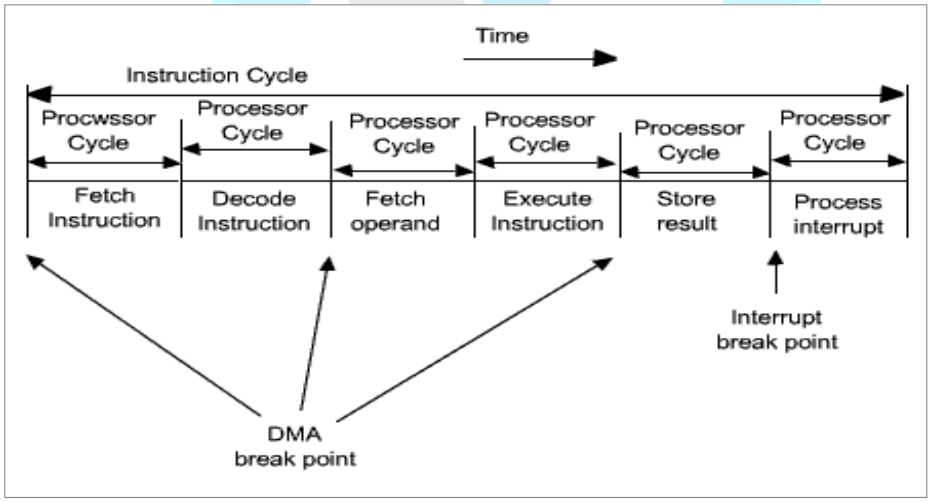



Fig 3.9: DMA break point

- When the processor is suspended, then the DMA module transfer one word and return control to the processor.
- Note that, this is not an interrupt, the processor does not save a context and do something else. Rather, the processor pauses for one bus cycle.
- During that time processor may perform some other task which does not involve the system bus. In the worst situation processor will wait for some time, till the DMA releases the bus.
- The net effect is that the processor will go slow. But the net effect is the

enhancement of performance, because for a multiple word I/O transfer, DMA is far more efficient than interrupt driven or programmed I/O.

The DMA mechanism can be configured in different ways. The most common amongst them are:

1. **Single bus, detached DMA - I/O configuration.**
2. **Single bus, Integrated DMA - I/O configuration.**
3. **Using separate I/O bus.**

**Single bus, detached DMA - I/O configuration**

- In this organization all modules share the same system bus. The DMA module here acts as a surrogate processor. This method uses programmed I/O to exchange data between memory and an I/O module through the DMA module.
- For each transfer it uses the bus twice. The first one is when transferring the data between I/O and DMA and the second one is when transferring the data between DMA and memory. Since the bus is used twice while transferring data, so the bus will be suspended twice. The transfer consumes two bus cycles.

The interconnection organization is shown in the Figure 3.10.

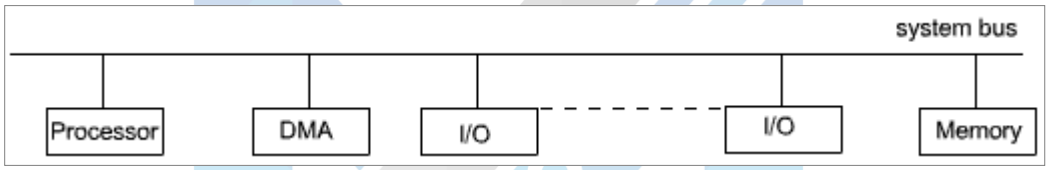
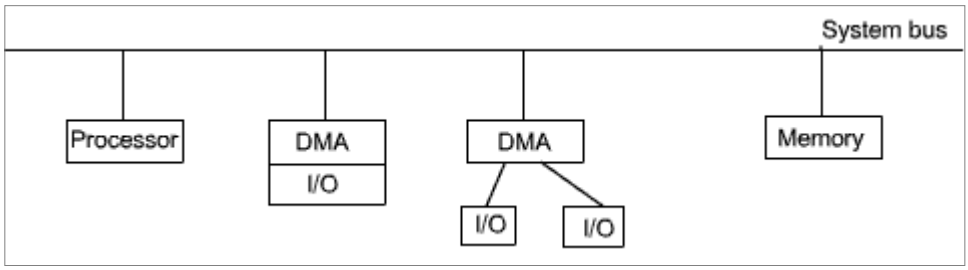



Fig 3.10: Single bus arrangement for DMA transfer

**Single bus, Integrated DMA - I/O configuration**

- By integrating the DMA and I/O function the number of required bus cycle can be reduced. In this configuration, the DMA module and one or more I/O modules are integrated together in such a way that the system bus is not involved. In this case DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules.
- The DMA module, processor and the memory module are connected through the system bus. In this configuration each transfer will use the system bus only once and so the processor is suspended only once.
- The system bus is not involved when transferring data between DMA and I/O device, so processor is not suspended. Processor is suspended when data is transferred between DMA and memory.

The configuration is shown in the Figure 3.11.




Fig

ure 3.11: Single bus integrated DMA transfer

**Using separate I/O bus**

- In this configuration the I/O modules are connected to the DMA through another I/O bus. In this case the DMA module is reduced to one. Transfer of data between I/O module and DMA module is carried out through this I/O bus. In this transfer, system bus is not in use and so it is not needed to suspend the processor.
- There is another transfer phase between DMA module and memory. In this time system bus is needed for transfer and processor will be suspended for one bus cycle. The configuration is shown in the Figure 3.12.

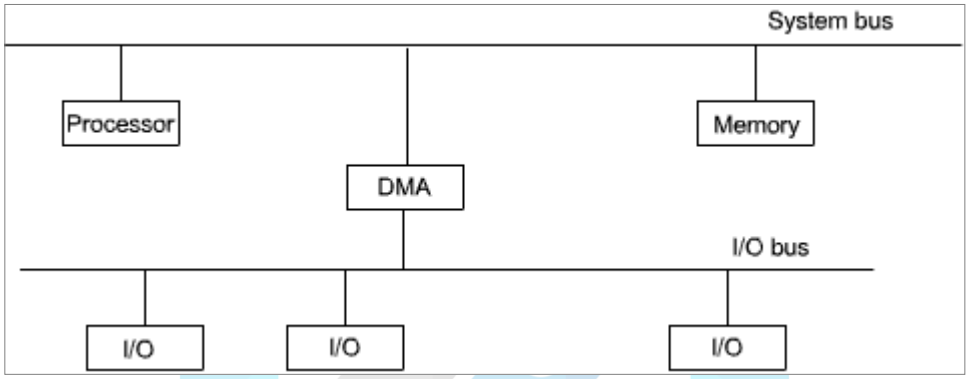



Figure 3.12: Separate I/O bus for DMA transfer

**Assignments:**

1. a. How many 128x8 RAM chips are needed to provide a memory capacity of 2048 bytes?
   b. How many lines of the address bus must be used to access 2048 bytes of memory? How many of these lines will be common to all chips?
   c. How many lines must be decoded for chip select? Specify the size of the decoders?
2. A computer uses RAM chips of 1024x1 capacity.
   a. How many chips are needed, and how should their address lines be connected to provide a memory capacity of 1024 bytes?
   b. How many chips are needed to provide a memory capacity of 16K bytes? Explain in words how the chips are to be connected to the address bus.
3. A computer employs RAM chips of 256x8 and ROM chips of 1024x8. The computer system needs 2K bytes of RAM,4K bytes of ROM, and four interface units, each with four registers. A memory-mapped I/O Configuration is used. The two highest order bits of the address bus are assigned 00 for RAM, 01 for ROM and 10 for interface registers.

a. How many ROM and RAM chips are needed?
b. Draw a memory-address map of the system.
c. Give the address range in hexadecimal for RAM, ROM, and Interface.