A Handbook on

Computer Science

3

Computer Organization and Architecture

CONTENTS

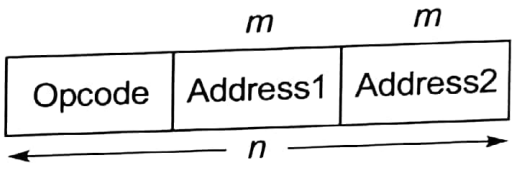
Machine Instructions and Addressing Modes

1

Machine Instructions

Types of instructions : Zero address instructions, one address instructions, and two address instructions.

- Zero address instructions : NOP
- One address instructions : PUSH, POP
- Two address instructions : ADD, MULT, SUB



$P \times 2^{2m} + Q \times 2^{2m} + R = 2^n$ (Total Instructions)

2-address instructions | 1-address instructions | Zero Address Instructions

Addressing Modes

- Addressing modes shows the way where the required object is present. The object may be an instruction or data.
- Addressing Modes are basically classified in two types:
  - (a) **Sequential Control Flow Addressing Modes:** When the program is stored in the sequential memory locations, the program counter itself points the next instruction address, therefore such Addressing Modes are focused on data.
  - (b) **Transfer of Control Flow Addressing Modes:** When the program is stored in the random memory locations (using structured programming) there is a need of special instructions and addressing modes in order to calculate the next instruction address.

Sequential Control Flow Addressing Modes

1. **Register Based Addressing Modes:**
   - (*i*) These Addressing Modes are used to access the data when it is available in the registers.
   - (*ii*) **Register/Register Direct Addressing Mode:** This Addressing Mode is used to access the local variables. In this mode the data is present

in the register, that register address is available in the address field of the instruction. Therefore, the effective address is equal to address field value. Data = [EA] = [Register Name]

Example: MOV R_1, R_2

2. **Memory Based Addressing Modes:** When the data is available in the memory, different memory based addressing modes are used to access the data. Under this, the EA is always the memory address.

(*i*) **Implied/Implicit Addressing Modes:** In this mode the data is available in the opcode itself. Therefore, there is no effective address.

Example: Compliment Accumulator (CMA) and all zero address instructions.

(*ii*) **Immediate Addressing Mode:**

- This mode is used to access the constants or to initialize registers to a constant value.
- Here the data is present in the address field of the instruction.
- The range of values initialized is limited by the size of the address field.
- If the address field size is *n*-bit, the possible range of immediate constants or data is: 0 to $2^n - 1$.

Example: MOV A, 10

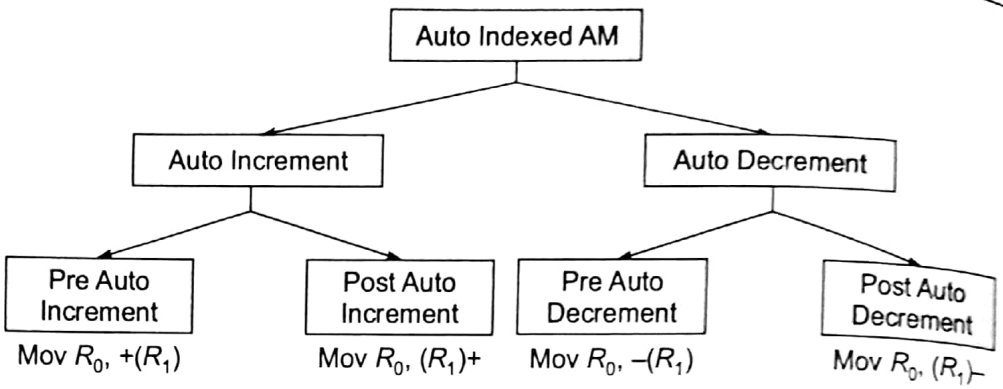
(*iii*) **Direct/Absolute Addressing Mode:**

- Used to access the static variables.
- The data is present in the memory, that memory cell address is present in the address field of the instruction. Data = [EA] = [Memory Address]
- One memory reference is required to read or write the data by using the direct addressing mode.

Example: MOV R_1, [1000]

(*iv*) **Auto Indexed Addressing Mode:**

- This mode is used to access the linear array elements. Thus, "base address" is required to access the data.
- The base address is maintained in the base register. Therefore the EA is Base Register ± Step Size.
- Step size is dependency on the amount of the data to be accessed from the memory. Data = [EA] = [[Base Register] ± Step size]



(*v*) Indirect Addressing Mode: (Array as parameter)

- Used to implement the pointers. The EA is available in either register or memory.
- This mode is divided into two types:
  - (a) **Register Indirect Addressing Mode:** Here, the EA is present in the address register, that register name is available in the address field of the instruction.
    EA = [Address Field Value] ⇒ [Register Name]
  - (b) **Memory Indirect Addressing Mode:** Here, the EA is present in the memory, that memory address is available in the address field of the instruction.
    EA = [Address field value] ⇒ [Memory address]
    Data = [EA] = [[Memory address]]
- Two memory references are required in memory indirect Addressing Mode.

(*vi*) Indexed Addressing Mode:

- Allows to implement array indexing.
  EA = Base Address + Index Value
- The register that holds the index value is "Register Indexed Addressing Mode", it is used to access the Random array element.
- In Indirect Index Addressing Mode the base address is present in the memory, that memory address is present in the address field of the instruction.

Transfer of Control Flow Addressing Mode

- During the execution of selection statements (if, then, else, goto, switch), iterative statements (For loop, while loop, do while loop) and subprogram concept, the control is transferred from one location to another location.
- Transfer-of-Control operations can be implemented by using three possible mnemonics: (*i*) Jump (*ii*) Branch and (*iii*) Skip.

- Transfer-of-Control instruction is divided into two types:
  - (*i*) **Unconditional Transfer-of-Control:** While execution of these instruction the program control is transferred to the target address without checking any condition. ***Example:*** HALT
  - (*ii*) **Conditional Transfer-of-Control:** While execution of these instruction the associated condition undergoes evaluation. If it evaluates to true, then the target address is loaded into Program Counter (PC) else no change in PC.

    The condition can be evaluated based on the status of the previous instruction.

    Example: They are used to implement 'if', 'switch', 'for', 'do while' and 'while' statements.
- Instruction set = opcode
- Vector interrupt – Interrupting source supplies branch info at processor through an interrupted vector.
- $\overline{INTA} \rightarrow INTR$

Computation of EA for the Next Instruction

Relative/PC Relative Addressing Mode

(Relocation data, Branch address at run time; Reduce instruction size)

- This Addressing Mode is used to access the instruction within the segment, therefore only the offset address is required.
- The offset address is available in the address field of the instruction.
- EA = Base Address + Offset Address
  ↓ ↓
  PC Address Field Value
- PC ← PC + IR [Address Field]
- Position dependent.
- Relocation at run time.

Base Register Addressing Mode (Position independent)

- This Addressing Mode is used to access the instruction between segments. Therefore base address as well as offset are required.
- Base address is maintained in the base register and offset address is maintained in the address field of the instruction.
- EA = [Base Register] + IR [Address Field]
- PC ← [Base Register] + IR [Address Field]

■■■

ALU and Data-path, CPU Control Design

2

Introduction

A CPU divided into two sections : Data section and Control section. Data section also called as "**data path**" and Control section is "**control unit**".

Data path contains registers and ALU, which performs certain operations on data items. Control unit issues the control signals to the data path.

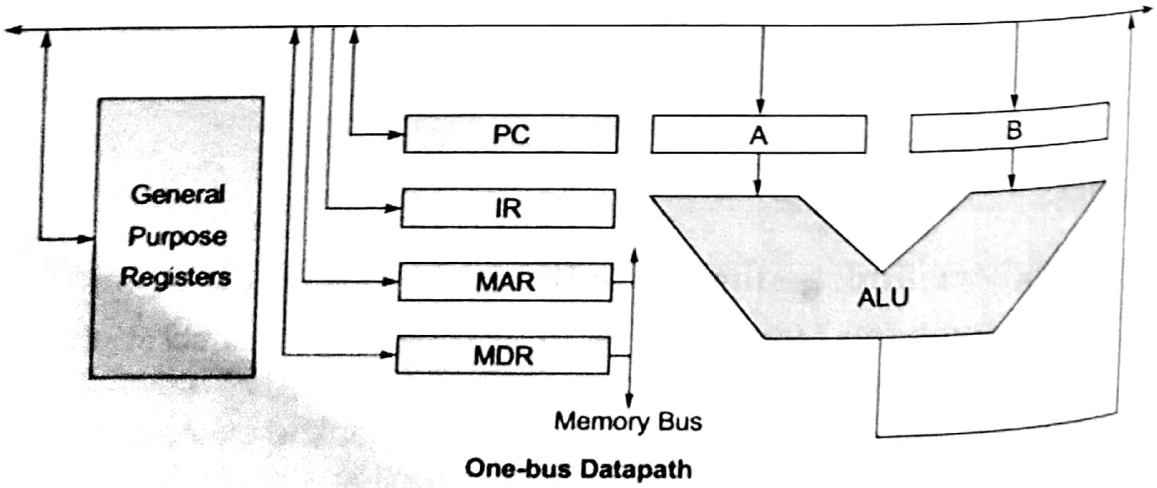
ALU and Data path

Types of data path : One bus data path, two bus data path and three bus data path.

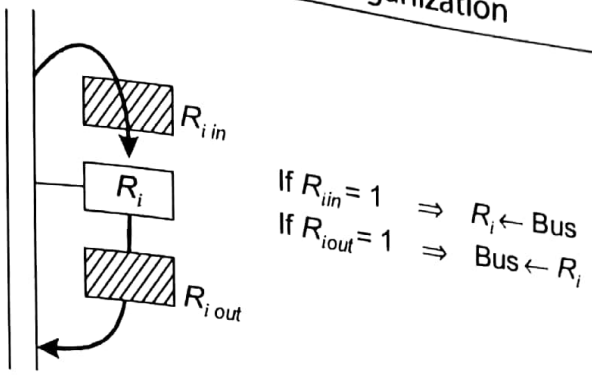
In one bus data path, CPU registers and ALU use same bus for all incoming and outgoing data.

In two bus data path, general purpose registers are connected to two buses. Data can be transferred from two different registers to the input point of the ALU at the same time.

In three bus data path, two buses are used as source (moves data out of registers) while the third is used as destination (Moves data into registers). Source bus is also called as in-bus and destination bus is also called as out-bus.

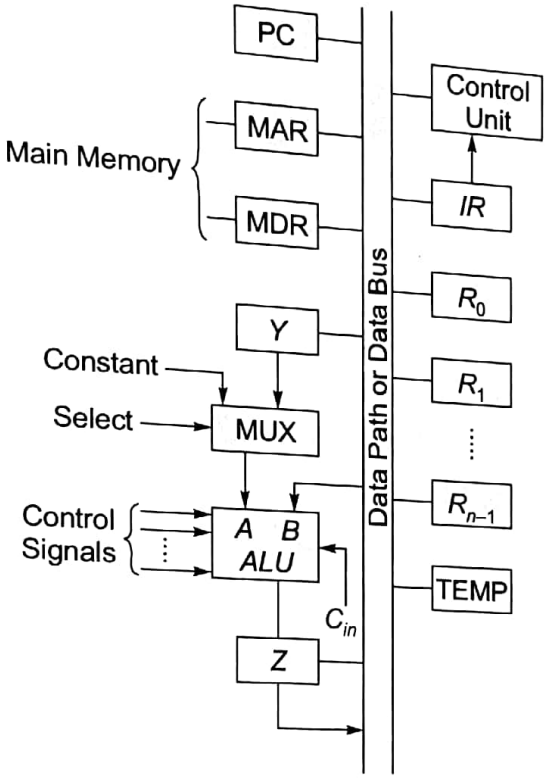


One-bus Datapath



Data flow Into register and Out from register

- Data bus = Bi-directional



Processor with One Bus

- The ALU, registers and interconnecting path is called "ALU data path".
- Constant is used to increment PC.
- If select = 0; output of MUX is constant
  = 1; output of MUX is *Y*.
- **CPU Registers:**
  - (*i*) **PC:** It is used to hold the starting instruction address and immediately point the next instruction's address.
  - (*ii*) **IR:** It is used to hold the currently fetched instruction to decode. As instruction format it predefined in this register therefore it is not a user accessible register.

(*iii*) **Accumulator:** It is used to hold the one of the ALU input and output.

(*iv*) **MAR (Memory Address Register):** It is used to carry the address. This register is directly connected to the address lines of the system bus.

(*v*) **MBR (Memory Buffer Register) or MDR (Memory Data Register):** It is used to carry the binary sequence. This register is directly connected to the data lines of the system bus.

- **Micro-operation:** It is a basic register to register transfer operation, also called as automatic operation/control mode/microinstruction.

  (*i*) All the micro-operations must complete the operation in one cycle.

  (*ii*) Control signals are required to execute the micro-operation.

  Example: $I_1 : R_1 \longrightarrow R_2$

  $R_1 \longrightarrow R_2$

  $R_{1\,out}$ $R_{2\,in}$

- **Microprogram:** The sequence of micro-operation is called as the microprogram.

  The fetch cycle microprogram is:

  $T_1 : PC \rightarrow MAR;\ PC_{out}, MAR_{in}$

  $T_2 : M[MAR] \rightarrow MBR;\ MAR_{out}, MBR_{in}$

  $T_3 : MBR \rightarrow IR;\ MBR_{out}, IR_{in}$

  $PC \leftarrow PC + \text{Step size};\ PC_{out}, PC_{in}$

- The basic operations performed are:

  (*i*) Register transfer ($R_2 \leftarrow R_1$):

  $R_{1\,out}, R_{2\,in}$ (only one clock cycle required)

  (*ii*) ALU operation ($R_3 \leftarrow R_1 + R_2$):

  $R_{1\,out}, Y_{in}$

  $R_{2\,out}$, Select = 1, Add

  $Z_{out}, R_{3\,in}$

  Minimum number of clocks = 3

  (*iii*) If two data paths are used:

  $R_{1\,out}, Y_{in}, R_{2\,out}$, Select = 1, Add

  $Z_{out}, R_{3\,in}$

  Minimum number of clocks = 2

  (*iv*) Memory Read:

  $R_2 \leftarrow M[(R_1)]$

  $R_{1\,out}, MAR_{in}$, read

Wait for memory function to complete (WMFC).

MDR_{out}, $R_{2\,in}$

Here number of clocks can be 2 or 3 depending on WMFC.

(v) **Memory Write:**

$M[(R_1)] \leftarrow R_2$

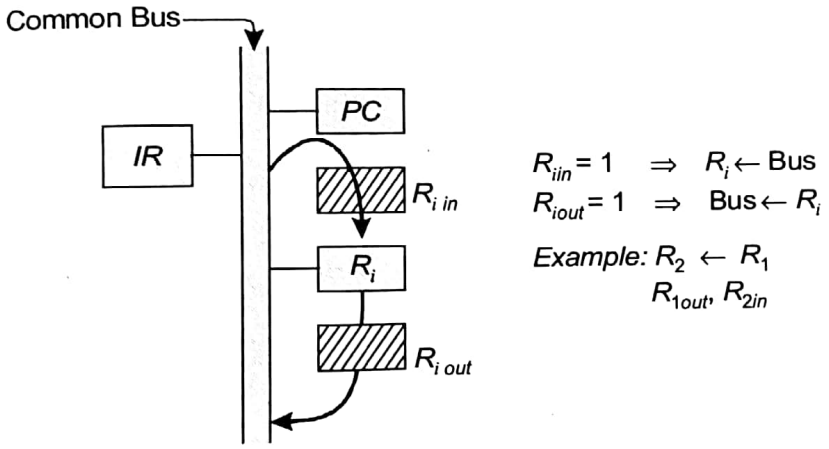
$R_{1\,out}$, MAR_{in}

$R_{2\,out}$, MDR_{in}, write

WMFC

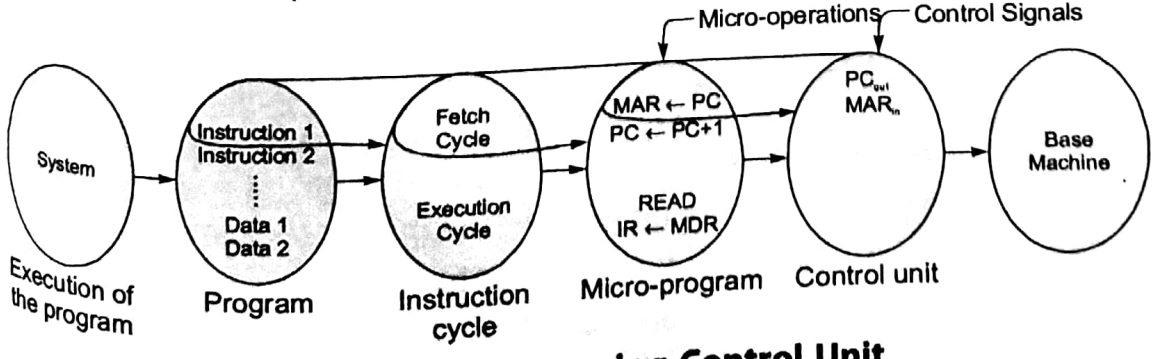
Minimum number of clocks = 3.

Control Unit



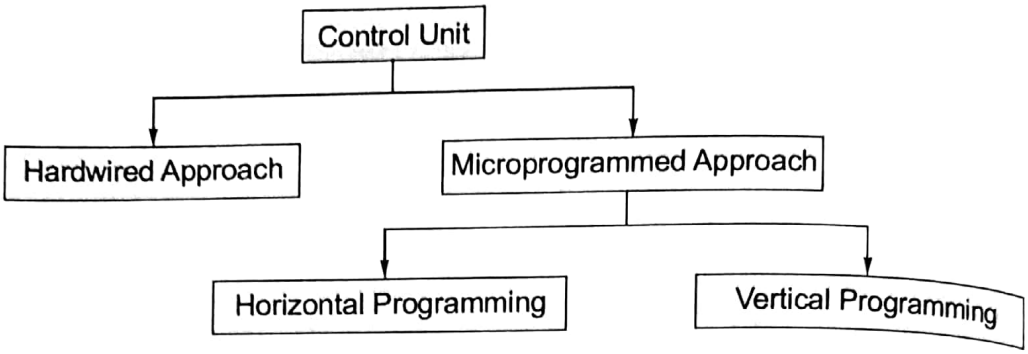
Data flow from one register to another register

- The purpose of control unit is to provide appropriate timing and control signals which are required to execute micro-operations.
- Control signals are directly executed on the base machine.
- Micro-program contains sequence of micro-operations.
- Subcycles of instruction cycle invokes their own micro-program.
- Instruction cycle is used to execute one instruction.
- System functionality is execution of the program.
- Program is a sequence of instruction along with data.



Program Execution using Control Unit

Control Unit Design



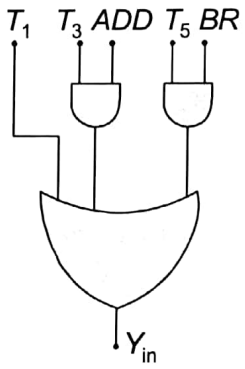
Hardwired Control Unit

1. The control signals are expressed as Sum-of-Product (SOP) expression and they are directly realized on the independent hardware.

   Example: $Y_{in} = T_1 + T_3 \cdot ADD + T_5 \cdot BR + \ldots$

   Here Y_{in} is enabled during T_1 for all instructions, during T_3 for *ADD* and so on...

   It can be realised as :



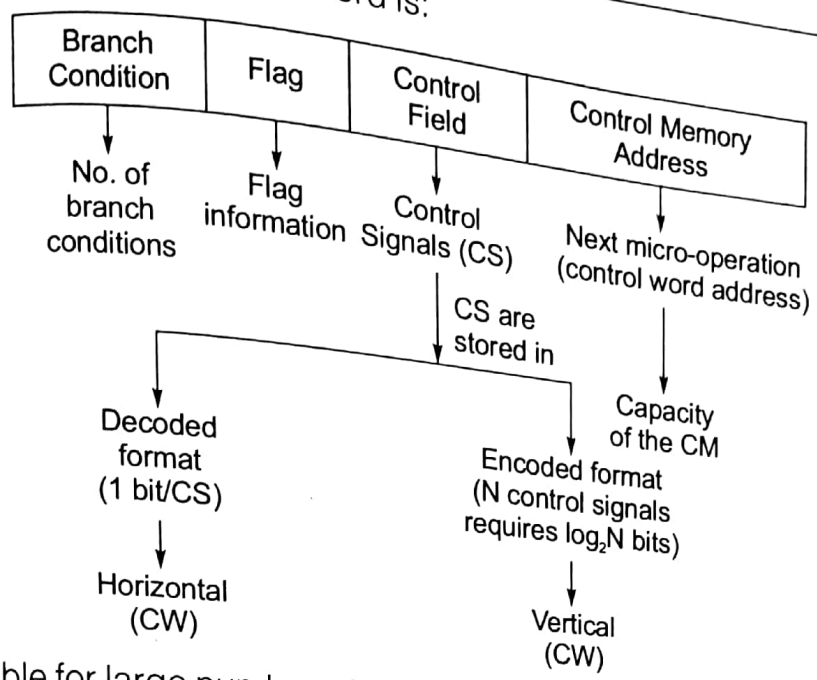
2. It is the fastest control unit design because of independent circuits.
3. For large number of instructions and control signals the complexity of hardware is more.
4. Relatively inflexible for any of the changes like modification, recorrection, thus it is not used in design and testing places.
5. Used in the Real Time Applications.

   Example: Air Craft simulation and weather forecasting etc.
6. Implemented in RISC processor.

Micro-programmed Control Unit

- Control Memory (CM) is used to store the microprogram.
- Control Memory (CM) is a permanent memory that is ROM.
- Microprogram contains sequence of microoperations (control word).

- The format of the control word is:



- Flexible for large number of instructions and CS.
- Control signals are generated slow in comparison to hardwired.
- Based on the type of Control Word (CW) stored in the Control Memory (CM), the control unit is classified into two types:
  (*i*) Horizontal Microprogrammed Control Unit (H μPCU)
  (*ii*) Vertical Microprogrammed Control Unit (V μPCU)

Horizontal μPCU

1. The control signals are represented in the decoded binary format that is 1 bit/CS.
   Example: If 68 CS are present in processor then 68 bits are required. More than one CS can be enabled at a time.
2. It supports longer Control Word (CW).
3. It is used in parallel processing applications.
4. It allows high degree of parallelism. If degree is *n*, *n* CS are enabled at a time.
5. Requires no additional hardware (decoders) ⇒ Faster than vertical μPCU.

Vertical μPCU

1. The control signals are represented in the encoded binary format.
   For *N* control signals $\log_2 N$ bits required.
   Example: If $N = 68$ then $\log_2 68 = 7$ bits required.
2. It supports shorter Control Word (CW).
3. It supports easy implementation of new control signals therefore it is more flexible.

4. It allows low degree of parallelism i.e. degree of parallelism is 1 or 0.
5. Requires an additional hardware (decoders) to generate control signals, it implies slower than horizontal μPCU.

Note:

- The ascending order of control unit w.r.t speed vertical < Horizontal < Hardwired (highest)
- The ascending order of control unit w.r.t flexibility (low flexibility) Hardwired < Horizontal < Vertical.
-

RISC	CISC
1. Reduced instruction set computer	1. Complex instruction set computer
2. Supports rich register set	2. Supports less number of registers
3. Supports less number of addressing modes	3. Supports more number of addressing modes
4. Supports fixed length instructions	4. Supports variable length instructions
5. One instruction/Cycle (CPI = 1)	5. (CPI ≠ 1)
6. Allows successful pipeline implementation (CPI = 1)	6. Supports unsuccessful pipeline (CPI ≠ 1)
7. *Example:* Motorola, Power PC, Advance Risk Machine	7. *Example:* Pentium

■■■

Memory and I/O Interfaces

3

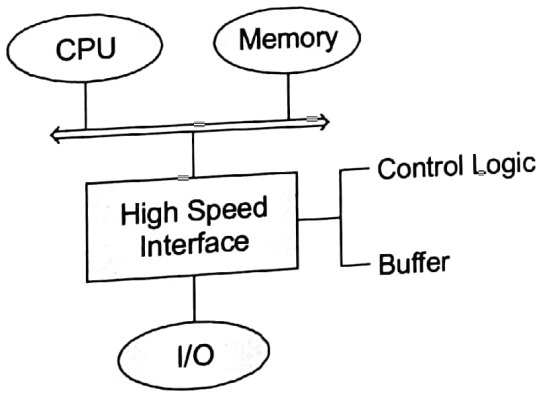
Introduction

In the isolated I/O CPU has distinct instructions for memory read/write and I/O read/write. The address what we use for accessing a device may be same as one of the address in the memory however separate lines are used to indicate I/O device address.

In memory mapped devices, their interface registers are memory mapped on to the address space of the memory.

Input Output Organisation

- Input output devices are very slow devices, therefore they are not directly connected with the system bus because Input output devices are electromagnetic devices and the CPU is the electronic device; there is difference in the operating mode, data transfer rate and word format.
- To synchronize the input output devices with the processor there is a need of high speed interface (I/O Module).

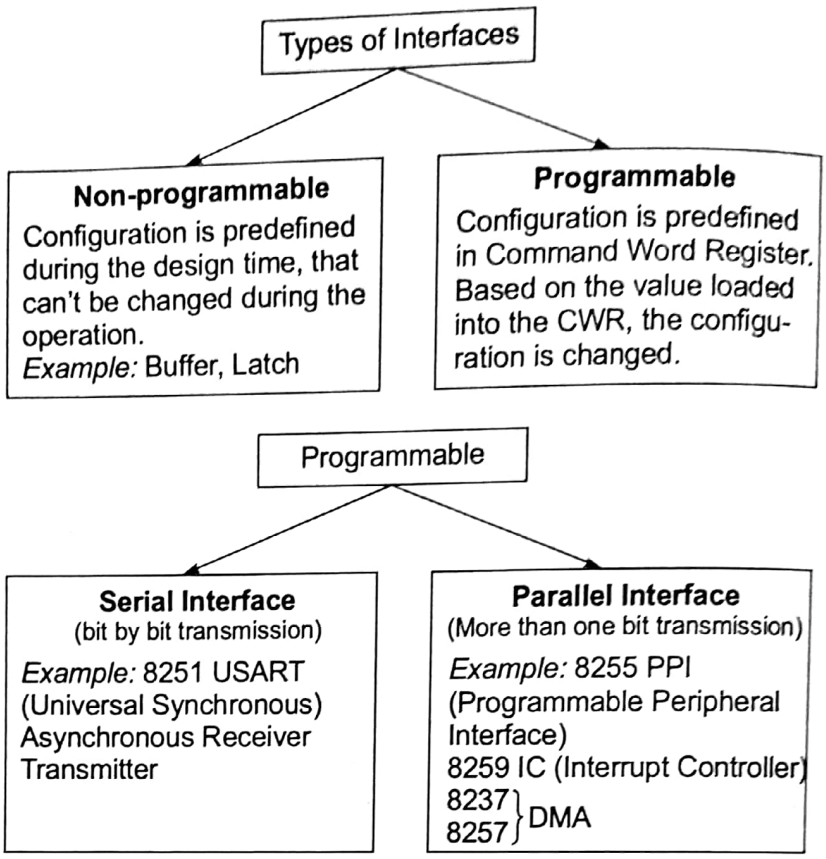


I/O Interface

The high speed interface enables the corresponding device for the operation. After preparing the data, the I/O device transfers it to the buffer.

- When the data is present in the buffer than the high speed interface generates the signal to the CPU and waiting for the acknowledgment. After receiving the acknowledgment it transfers the data to the CPU with high rate. Thus, the speed gap is minimized.

- Types of Interfaces:

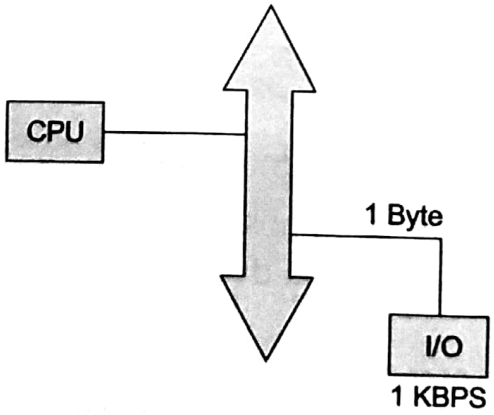


I/O Transfer Modes

Three different modes available to transfer data from I/O to CPU/memory.

1. Programmed I/O

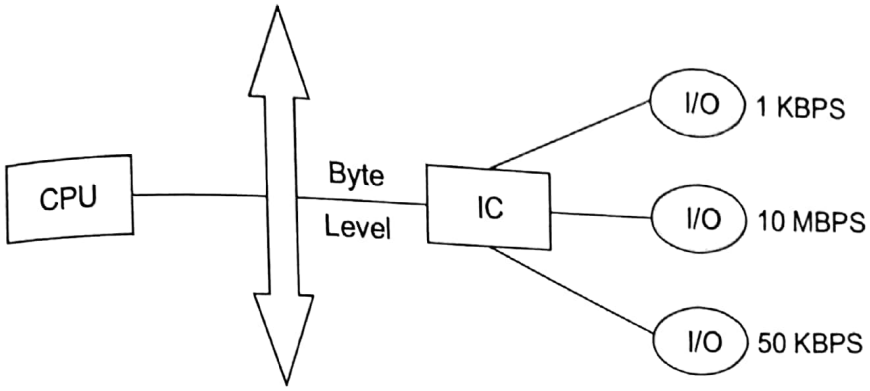
- Here the CPU directly communicates with the I/O device. Hence, the processor utilization is inefficient.
- The processor undergoes waiting until completion of the I/O operation. This waiting depends on the speed of the I/O device.



2. Interrupt Driven I/O

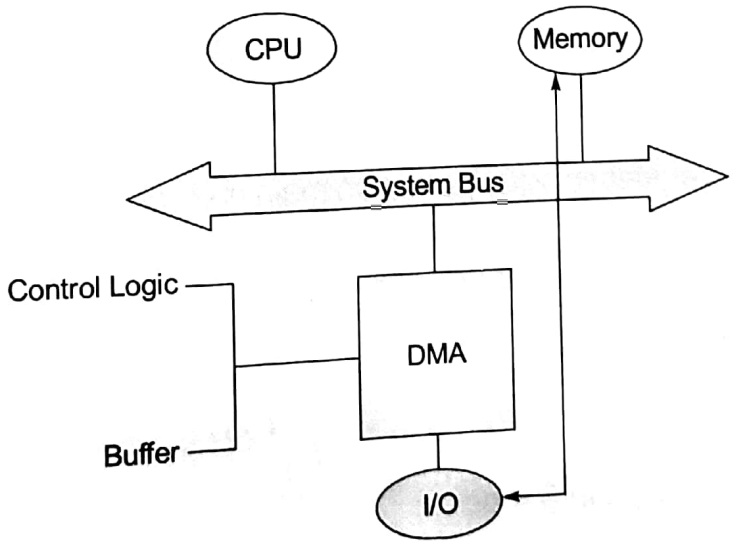
- Interrupt controller (IC) is used as the high speed interface between the basic I/O devices to CPU. Hence, the processor's utilization is good (efficient).

- Here, the CPU is communicating with IC only. Thus, the execution/transfer time is not depending on the speed of I/O device, rather it depends on the latency of IC.



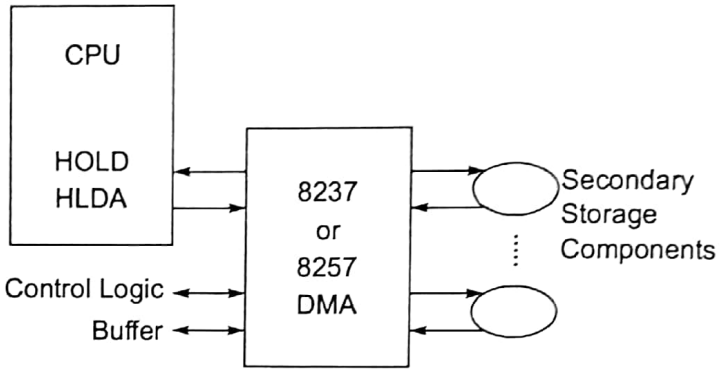
3. Direct Memory Access (DMA)

- The bulk amount of data is transferred from the I/O devices to the main memory without involvement of the CPU.
- The secondary storage devices are connected to the system bus through DMA, while execution of user program from the auxiliary memory to main memory the program is transferred page by page through the DMA.



- The CPU initializes the DMA along with source and destination address, control signals and count value. Later it is busy with some other execution.
- Based on priority - Daisy chain - non uniform priority to each device - polling.
- RFE is top privileged instruction.

- The DMA control logic interprets the request and enables the corresponding device for the operation. After preparing the data device transfers it to the buffer.
- When the data is available in the buffer then DMA enables the HOLD signal, to gain the control of the bus and waiting for the acknowledgment. After receiving the HLDA signal, the DMA transfers the data from I/O device to main memory until the count becomes zero. After transfer operation, it establish connection to CPU.



- **During the DMA operation, the CPU is in two states :**
  (*i*) Busy State
  (*ii*) Block /Hold State
- Until preparing the data, the CPU is busy with other executions. While transferring the data, the CPU is in blocked state.
- Let '*X*' = Preparation time (Depends on I/O speed),
  '*Y*' = Transfer time (Depends on main memory speed).

Then, % time CPU is blocked = $\left(\frac{Y}{X+Y}\right)\times 100$

% time CPU is busy = $\left(\frac{X}{X+Y}\right)\times 100$

- **DMA is operated in 3 modes:**
  (*i*) **Burst Mode:** After receiving the HLDA signal, bulk amount of data is transferred to main memory.
  (*ii*) **Cycle Stealing Mode:** Before receiving the HLDA signal, it forcefully suspend the CPU operation and transfers very important data to the main memory.
  (*iii*) **Block Mode:** After receiving the HLDA signal, the data is transferred to the main memory in block wise.

■■■

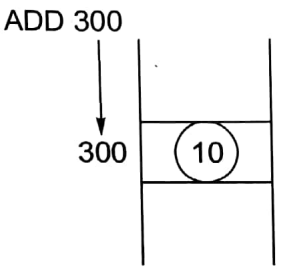
Instruction Pipelining

4

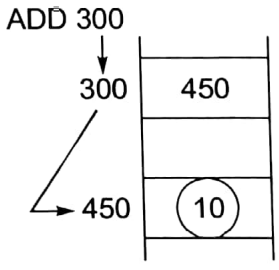
Introduction

The set of phases for an instruction execution cycle:

1. **Instruction Fetch (IF):**
   - (*i*) MAR ← PC [MAR ← 2000]
   - (*ii*) PC ← PC + 1 [Next instruction: 2004]
   - (*iii*) READ CONTROL Signal is enabled
   - (*iv*) IR ← MDR
2. **Instruction Decode (ID):** Two know the purpose of instruction.
3. **Operand Fetch (OF):**
   - (*i*) Memory Read: MAR ← LOC X
   - (*ii*) Write into any general purpose register: R_i ← MDR
4. **Execute and Store (EX):**
   - (*i*) MAR ← LOC Z
   - (*ii*) MDR ← R_i
   - (*iii*) Write into memory.
5. **Indirect Phase:**



Here *EA* = 300 (Direct Address)



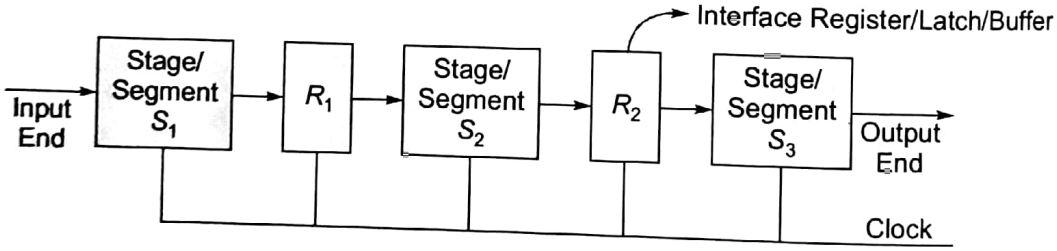
Here *EA* = 450 (Indirect Address)
(2 memory cycles required)

6. **Interrupt Phase:**
   - (*i*) Each instruction execution involves a sequence of microoperations to be implemented by the processor.
   - (*ii*) A microoperation is a register transfer operation, completes in 1 clock cycle.
   - (*iii*) If T is the total time for instruction cycle that T will be divided into T_1, T_2, T_3 ..... among different phases.

- The CPU is operated in 2 modes:
  (*i*) User mode or Non-privileged mode executes user program.
  (*ii*) System mode/Supervisor/Kernel/Privileged mode executes operating system to obtain system services.
- Types of interrupts:
  (*i*) **External or Hardware interrupt:** Araised due to timing and I/O devices.
  (*ii*) **Software Interrupt:** Araised due to switching from user to system mode or vice versa.
  (*iii*) **Internal Interrupt:** Araised due to incorrect use of instructions and data.
  Example: Division by zero register overflow invalid opcode.

Pipelining

- Accepting new input at one end before previously accepted input appears as an output at the other end is pipelining.
- Pipelining allows overlapping execution.
- The successful characteristic of the pipeline is for every new cycle, new input must be inserted into the pipeline.
  ∴ CPI = 1



Types of Pipelines

1. **Linear Pipeline:** This pipeline is used to perform only one specific function.
2. **Non-linear pipeline:** This pipeline is used to perform the multiple functionalities. It uses feed forward and feed backward connections.
3. **Synchronous Pipeline:** On a common load/clock all the registers transfer data to the next stages simultaneously.
4. **Asynchronous Pipeline:** The data flow along the pipeline stages is controlled using handshake protocol.

Performance Evaluation of Pipeline Processor

- Consider '*K*' segment pipeline with clock cycle time t_p used to execute *n*-tasks. The first task is executed in the non-overlapping time span, so it requires *K* cycles to complete the operation.

- The remaining $(n-1)$ tasks emerge from the pipe at the rate of one cycle per task, so $(n-1)$ tasks requires $(n-1)$ cycles to complete the operation. Thus, execution time of the K segment pipeline

$$ET_{pipe} = K + (n-1) \text{ cycles}$$

$$\boxed{ET_{pipe} = (K + (n-1)) \cdot t_p}$$

- The performance gain of the pipeline processor over the non-pipeline is:

$$\text{Speed up(S)} = \frac{\text{Performance}_{pipe}}{\text{Performance}_{nonpipe}} = \frac{\frac{1}{ET_{pipe}}}{\frac{1}{ET_{nonpipe}}} = \frac{ET_{nonpipe}}{ET_{pipe}}$$

$$\boxed{S = \frac{nt_n}{(K + (n-1))t_p}}$$

- For large number of tasks or instructions $K + (n-1)$ approaches to n

$$\boxed{S = \frac{nt_n}{nt_p}} \Rightarrow \boxed{S = \frac{t_n}{t_p}}$$

- When all the instructions are taking the same number of cycles then one instruction execution time is equal to the number of stages in the pipeline (K) i.e.,

$$\boxed{t_n = K_{t_p}} \Rightarrow \boxed{S = \frac{K \cdot t_p}{t_p}} \Rightarrow \boxed{S = K}$$

Instruction Pipeline

The instruction pipeline operates on a stream of instructions by overlapping the phases of instruction cycle like Instruction Fetch (IF), Instruction Decode (ID), Operand Fetch (OF), Execute (EX), Memory Access (MA), Write Back (WB) and so on.....

Stall Cycle

- A stall cycle is one during which no meaningful operation performed for an instruction.
- It arises due to:
  - (i) Uneven clock cycles needed by each stage for instructions.
  - (ii) Increased buffer overhead

(*iii*) Memory operands

(*iv*) Pipelined dependencies

- $$S_{eff} = \frac{S_{ideal}}{(1 + \text{Stall frequency} \times \text{Stall cycles})} = \frac{K(\text{number of stage})}{(1 + \text{Stall frequency} \times \text{Stall cycles})}$$

- The maximum speedup that can be achieved by using a pipelined processor = number of stages.

$$S_{max} = S_{ideal} = K$$

- $$\text{Efficiency } E_K = \frac{S}{K} = \frac{n}{K + (n-1)}$$

- $$\text{Throughput} = \frac{\text{Number of tasks processed}}{\text{Total time required to process the tasks}}$$

$$\text{Throughput} = \frac{n}{(K + (n-1)) \cdot t_p}$$

Advance Pipeline (RISC Pipeline)

- In the RISC processor, instruction pipeline is implemented to execute the instruction.
- RISC processor supports three categories of instructions.

(*i*) **Data Transfer:** Data transfer instruction can be implemented by using load and store

Load r_0, 2(r_1); $r_0 \leftarrow M[2 + [r_1]]$

Store 3(r_2), r_1; $M[3 + [r_2]] \leftarrow r_1$

(*ii*) **Data Manipulation (ALU Operation):** ALU operation are directly performed on the registers.

Add r_0, r_1, r_2; $r_0 \leftarrow r_1 + r_2$

(*iii*) **Branch Operation:** The unconditional branch operation syntax is

Jmp 1000; PC ← target address.

Conditional Branch Operation

JNZ r_0, 2000; [True ; PC ← target address
False ; P ← sequential address

- In RISC pipeline the branch instruction execution is completed at the end of the second stage. So branch penalty [2 – 1] = 1.

- Register renaming can eliminate all WAR hazard and WAW.
- Bypass cannot handle all RAW hazard (Memory load instruction).
- Control hazard penalties can be eliminated by dnomic branch prediction.
-

Dependencies in the Pipeline

- Dependency causes stalls in the pipeline. Stall is a extra cycle created in the pipeline with NOP.
- There are three kinds of dependencies possible in the pipeline

(*i*) Structural dependency:

- This dependency is present because of resource conflict. The resource may be a memory or register or functional unit.
- Conflict is an unsuccessful operation, so to make it successful make the instruction wait until the resource become available. This waiting creates stalls in the pipeline.
- To minimize the number of stalls in the pipeline due to the structural dependency, "Renaming" concept is used.
- RAW = Register rename decrease.
- WAR = All decrease (removed).

(*ii*) Data dependency:

- Consider the program segment

  i : instruction

  j : instruction

  Data dependency exists between i and j when the "instruction j tries to read the data before instruction i writes it".

(*iii*) Control dependency:

- While execution of transfer of control operation the program control is transferred from the current location to the target location.
- If the current instruction is decoded as data transfer or data manipulation operation than the sequential instruction is a wanted instruction.
- The process of removing unwanted instruction from the pipeline is called as flush or freeze.

  Flush operation creates stalls in the pipeline.
- The number of stall cycles created in the pipeline due to branch operation is called as Branch Penalty.

Branch Penalty = 'At what stage the target address is available – 1

- To reduce the number of stalls in the pipeline due to the branch operation, branch prediction buffer or loop buffer or branch target buffer is used.

  Branch target buffer: It is the high speed buffer maintained at the instruction fetch stage to store the predicted target addresses.

- The out-of-order execution creates two more dependencies in the pipeline
  1. **Anti Dependency:** It exists when instruction *j* tries to write the register before instruction *i* reads it.
  2. **Output Dependency:** It exists when instruction *j* tries to write the destination before instruction *i* writes it.
- To handle the Anti and Output dependencies "Register Renaming" concept is used.

  Register Renaming means use the temporary storage to hold the out-of-order execution output. After receiving the exception from the dependent instruction update the register file with temporary storage content.

Hazards

- Hazard is a delay. Delay creates extra cycles, these extra cycles without operation is called as stalls.
- **Hazards are classified as:**
  (*i*) Read-After-Write (RAW)
  (*ii*) Write-After-Read (WAR)
  (*iii*) Write-After-Write (WAW)
- Consider a program segment where the instruction *j* follows instruction *i* in the program order.
  (*i*) RAW: This hazard is created when instruction *j* tries to read the data before instruction *i* write it (TRUE DATA DEPENDENCY).
  (*ii*) WAR: This hazard is created when instruction *j* writes the data before instruction *i* reads it (ANTI DEPENDENCY).
  (*iii*) WAW: Instruction *j* writes the data before instruction *i* writes it (OUTPUT DEPENDENCY).

Performance Evaluation of the Pipeline with Stalls

$$S = \frac{CPI_{Nonpipe} \times \text{Cycle Time}_{Nonpipe}}{CPI_{pipe} \times \text{Cycle Time}_{pipe}} \quad \text{for ideal } CPI_{pipe} = 1$$

$$S = \frac{CPI_{Nonpipe} \times Cycle\ Time_{Nonpipe}}{\left(Ideal\ CPI + \frac{No.\ of\ stall\ cycles}{Instruction}\right) \times Cycle\ Time_{pipe}}$$

- When there is no setup time overhead, then both the cycle times are equal.

$$S = \frac{CPI_{Nonpipe}}{1 + \left(\frac{No.\ of\ stall\ cycles}{Instruction}\right)}$$

- When all instructions are taking the same number of cycles then one instruction requires cycle equal to number of stages in pipeline to complete the execution.

$$S = \frac{Pipeline\ Depth}{1 + \frac{No.\ of\ stall\ cycles}{Instruction}}$$

- When efficiency is 100%, no stalls

$$S = Pipeline\ depth$$

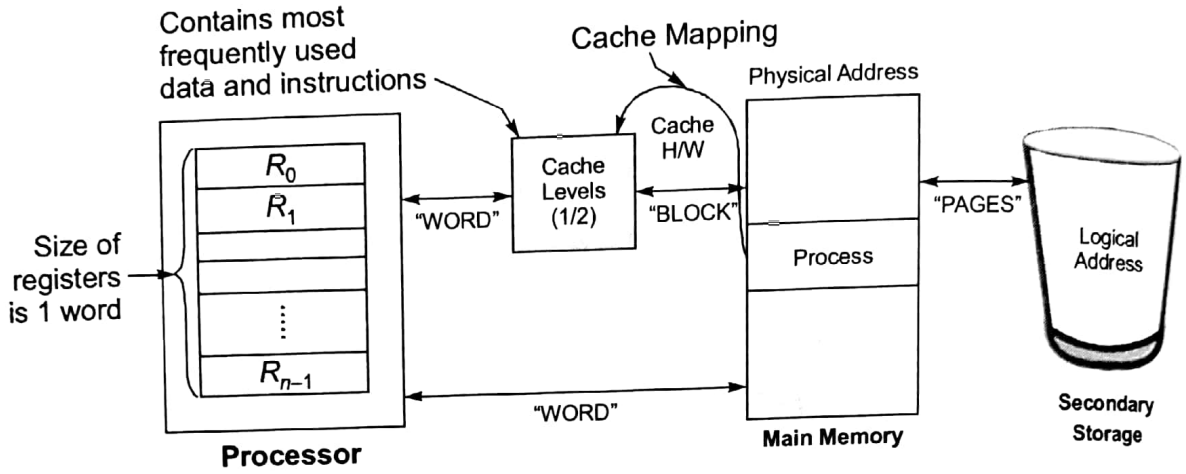
- Instruction pipeline can create one or more stall, since register rename can eliminate it also. So always get false.

■■■

Cache and Main Memory, Secondary Storage

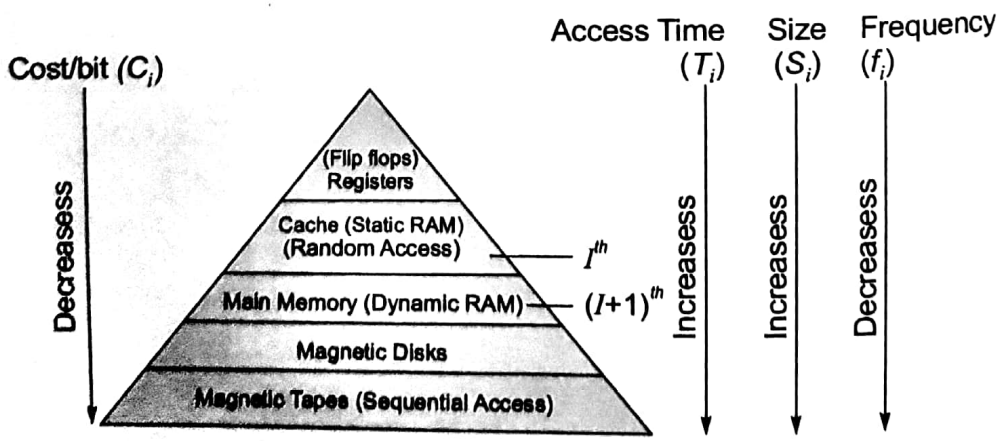
5

Memory Organisation



Memory Hierarchy

- The memory hierarchy shows the accessing order of the existing memory system.
- Average Memory Access Time = $\frac{\text{All Instruction Fetch Time + Write Time}}{\text{Total instruction}}$
- Smaller cache block → ↓ Cache miss penalties
  ↑ Larger TAG → ↑ Cache TAG overhead
  ↑ Spatial locality
- The purpose is to bridge the speed mismatch between fastest processor to the slowest memories at reasonable cost.



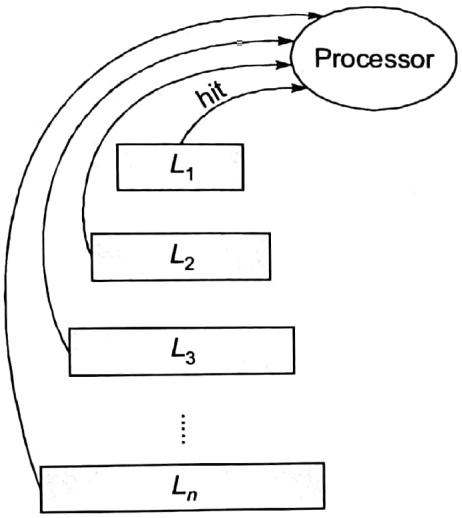
- When the processor refers to I^{th} level memory if it is found then "hit" else "miss".

Types of Memory Organisation

Based on the order of accessing the memory system, the memory organisation is divided into two types:

Simultaneous Access

- In this the CPU directly communicates with all the levels of memories.
- Consider a 'n' level memory system.



Let $H_1, H_2, \ldots, H_n$ be the hit ratios upto n level and $T_1, T_2, \ldots T_n$ be the access time upto n level memory system.

(*i*) The average access time of the memory is;

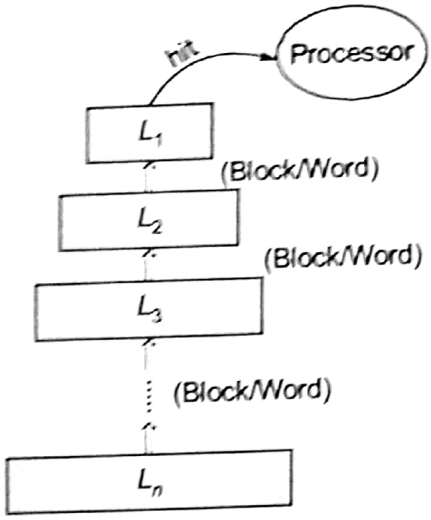
$$T_{avg} = H_1 T_1 + (1 - H_1) H_2 T_2 + \ldots + (1 - H_1)(1 - H_2)\ldots(1 - H_{n-1}) H_n T_n$$

(*ii*) Throughput of the memory = $\frac{1}{T_{avg}}$ word/sec

(*iii*) $C_{avg}/\text{bit} = \frac{C_1 S_1 + C_2 S_2 + \ldots + C_n S_n}{S_1 + S_2 + \ldots + S_n}$

Hierarchial Access

- In this the data from 2nd level is transferred to 1st level and CPU accesses the data from 1st level.
- Consider a 'n' level memory system.



- Let $H_1, H_2, ..., H_n$ be the hit ratios upto n level memory system and $T_1, T_2, ... T_n$ be the access time upto n level memory system.

$$T_{avg} = H_1T_1 + (1 - H_1)\, H_2\, (T_1 + T_2) + (1 - H_1)\, (1 - H_2)\, H_3\, (T_3 + T_2 + T_1) + \\ ... + ((1 - H_1)(1 - H_2)...(1 - H_{n-1})H_n\, (T_n + T_{n-1} + ... + T_2 + T_1))$$

Cache Memory

- Cache is used as the intermediate memory between CPU and main memory.
- It is small and fast memory.
- By placing most frequently used data and instructions in cache, the average access time can be minimized.
- When miss occurs the processor directly obtains data from main memory and a copy of it is send to cache for future references.
- The cache memory works on the principle of Locality of Reference (LOR). LOR states that "the references to memory at any given interval of time tends to be confined to within localized areas of memory".
- **The performance of cache depends on:**
  - (*i*) Cache size (small)
  - (*ii*) Cache block size
  - (*iii*) Number of levels of cache
  - (*iv*) Cache mapping technique
  - (*v*) Cache replacement policy
  - (*vi*) Cache updation policy
- Cache is divided into equal parts called as cache blocks/cache lines.
- Data is transferred from main memory to cache in form of blocks, thus both are divided into blocks of equal size.

- Cache line size = Main memory block size

  Number of cache lines = $\frac{\text{Cache size}}{\text{Block size}}$

Mapping Techniques

The process of transferring the data from the main memory to cache memory is called as mapping. There are three different mapping techniques:

1. Direct mapping
2. Associative mapping
3. Set associative mapping

Direct Mapping

- In this technique, mapping function is used to transfer the data from main memory to cache memory.

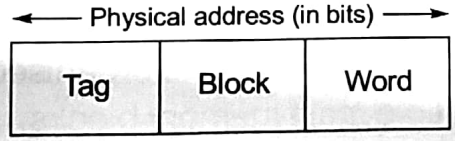
  The mapping function is K Mod $N = i$

  where K = Main memory block number

  N = Number of cache lines

  i = Cache memory line number

- The address representation is:



- Accessing same block from different pages is always a miss. The replacement is done when "conflict miss" occurs.
- Hit ratio is less.
- The higher order bits of the address is compared simultaneously with each cache tag. If a match is not found it is miss. The delay of tag comparator is called hit latency or hit delay. So, the reference is forward to the main memory.
- The main memory control logic interprets the request into its known format.

Tag	Word

- The tag field is directly connected with the address logic of the main memory. Therefore, the corresponding block is enabled.
- By using the mapping function the main memory block is transferred to corresponding cache line along with the complete tag. Later the CPU accesses the data from the cache.

- The number of bits in the tag is tag size or tag length.
  The maximum number of tag comparisons = 1.

Associative Mapping

- Any main memory block can be mapped to any location in cache. Thus, the cache address is not in use.
- The data is transferred from main memory to cache along with the complete tag. The address representation is

← Physical address (in bits) →

Tag	Word

- The replacement is done only when the cache is full or when "capacity miss" occurs.
- The block number is updated in the tag field. The higher order bits of the address compared sequentially with each cache tag, if a match is not found then it is "miss".
- The maximum number of tag comparisons = N (Number of cache blocks)
- The complexity of tag comparator is more.
- Cache size = Tag memory size + Data memory size
  Data memory size = Number of cache lines × Number of bits in the line

Set Associative Mapping

- In this mapping the cache memory is organised into sets, each set is capable to hold multiple main memory blocks.

$$\text{Number of sets} = \frac{N}{\text{P-way}}$$

where N = Number of cache lines
P = Number of MM blocks possible on each cache line

- The address interpretation is:

← Physical address (in bits) →

Tag	Set offset	Word offset

- The mapping function used is

$K \bmod S = i$

where, K = Main memory block number
S = Number of cache sets
i = Cache memory set number

- The replacement is done when the set is full.

- Hit ratio and complexity of tag comparator is optimal.
- Maximum number of tag comparisons = P.
- When P =1 set associative is direct mapping.
- There is a need of multiplexers to compare the existing tags in the set one by one with respect to CPU generated tag.
- Tag memory size = Number of sets in the cache × Number of blocks in the set × Number of tag bits in each block

Compulsory, Capacity and Conflict Misses

- Compulsory misses are caused by the first reference to a location in memory. Cache size and associatively makes no difference to the number of compulsory misses. Pre-fetching can help here, as can larger cache block sizes (which are a form of pre-fetching).
- **Capacity misses** are those misses that occur regardless of associativity or block size, solely due to the finite size of the cache.
- **Conflict misses** are those misses that could have been avoided, had the cache not evicted an entry earlier.
- **Double the associativity :**
  - (*i*) Capacity and line size constant.
  - (*ii*) Halves the number of sets.
  - (*iii*) Decrease conflict misses, no effect on compulsory and capacity misses.
  - (*iv*) Typically higher associativity reduces conflict misses because there are more places to put the same element.
- **Halving the line size:**
  - (*i*) Associativity and number of sets constant.
  - (*ii*) Halves the capacity.
  - (*iii*) Increases compulsory and capacity misses but no effect on conflict misses.
  - (*iv*) Shorter lines mean less "pre-fetching" for shorter lines. It reduces the cache's ability to exploit spatial locality. Capacity has been cut in half.
- **Doubling the number of sets:**
  - (*i*) Capacity and line size constant.
  - (*ii*) Halves the associativity.
  - (*iii*) Increases conflict misses but no effect on compulsory and capacity misses.
  - (*iv*) Associativity is less.

Replacement Algorithms

When the cache is full, the existing block is replaced with new block. There are two replacement algorithms used

(a) **FIFO (First-In-First-Out):** In replaces the cache block with new block which is having longest time stamp.

(b) **LRU (Least Recently Used):** It replaces the cache block with new block which is having less number of references with longest time stamp.

Updating Techniques

Coherence: The same address contains different values at different places (cache is updated while main memory is not updated). This property is called "coherence" which creates loss of data.

There are two updating techniques used: Write through and write back.

Write Through

In this technique, the CPU performs the simultaneous updation in the cache memory and main memory. Thus, there is no coherence.

- Word updation (T_W) Time = $\max\begin{pmatrix}\text{Word updation} \\ \text{time in cache}\end{pmatrix}\begin{vmatrix}\text{Word updation time} \\ \text{in main memory}\end{vmatrix}$

- $T_{avg_{read}} = \underset{\text{read hit}}{H_r}\ \underset{\text{read data}}{T_c} + \underset{\text{read miss}}{(1-H_r)} + \underset{\text{read allocate}}{(T_m} + \underset{\text{read data}}{T_c)}$

- $T_{avg_{write}} = \underset{\text{write hit}}{H_w}\ \underset{\text{word updation time}}{T_w} + \underset{\text{write miss}}{(1-H_w)} + \underset{\text{write allocate}}{(T_m} + \underset{\text{word updation time}}{T_w)}$

- Total access time of memory, when both read and write cycle is considered is:

  $T_{avg} = f_{read} \times T_{avg_{read}} + f_{write} \times T_{avg_{write}}$

- Throughput $\eta_{WT_{cache}} = \dfrac{1}{T_{avg_{WT}}}$ word/sec

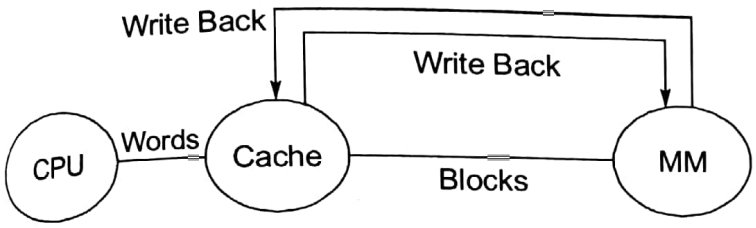
- The hit ratio for the write operation is always 1,

- $T_{avg_{write}} = H_w T_m$; where T_m = Main memory access time

Write Back

The CPU performs the write operation only on the cache, still coherence is present but that doesn't create problem because before replacing the cache block with new block the updation are taking back into the main memory.

- Each cache line contains one extra bit, called modified bit (update bit), when the block is updated the corresponding modified bit is set.

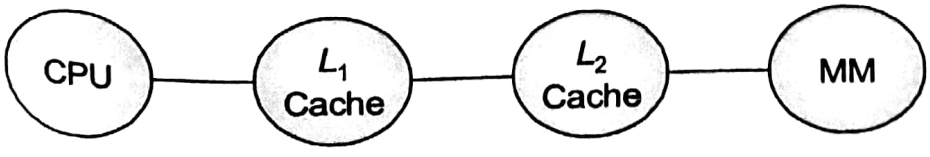


Modified bit
- 1 (Dirty bit) [Hierarchical: $(T_{read} = T_{write})$] $[T_{read} = H.\ T_c + (1 - H)\ (T_{cache} + T_{mem\text{-}block} + \%(\text{dirty}) .\ T_{mem\text{-}block})]$
- 0 (Clean bit) [Simultaneous: $T_{read} = H.\ T_c + (1 - H)\ (T_m + \%(\text{dirty}) - T_{mem\text{-}block})]$

- $T_{avg_{read}} = H_r\ T_c + (1 - H_r)$ [% dirty bits $(T_m + T_m + T_c)$ + % clean bits $(T_m + T_c)$]
  (read hit, read data, read miss; write back, read allocate, read data; read allocate, read data)
- $T_{avg_{write}} = H_w\ T_c + (1 - H_w)$ + [% dirty bits $(T_m + T_m + T_c)$ + % clean bits $(T_m + T_c)$]
  (write back, write allocate, write data; write allocate, write data)
- $T_{avg_{write\ back}} = f_{read} \times T_{avg_{read}} + f_{write} \times T_{avg_{write}}$
- Throughput $\eta_{write\ back} = \dfrac{1}{T_{avg_{write\ back}}}$ word/sec

Multilevel Caches

- To reduce the miss penalty, multilevel caches are used in the system design.



- In multilevel caches, two kinds of miss rates will be calculated

  (*i*) Local miss rate:

$$\text{Local miss rate} = \frac{\text{No. of misses in the cache}}{\text{Total No. of references to that cache}}$$

  (*ii*) **Global miss rate:**

$$\text{Global miss rate} = \frac{\text{No. of misses in the cache}}{\text{Total No. of CPU generated references}}$$

- Average access time can be calculated as

$$T_{avg} = \text{Hit time}_{L_1} + \text{Miss rate}_{L_1}(\text{Local miss rate}) \times \text{Miss penalty}_{L_1}$$

$$\text{Miss penalty}_{L_1} = \text{Hit time}_{L_2} + \text{Miss rate}_{L_2} \times \text{Miss penalty}_{L_2}$$

$$\text{Miss penalty}_{L_2} = \text{MM Access time}$$

- Average memory stall cycles = (No. of miss_{L1}/Instruction × Hit time_{L2})+ (No. of misses_{L2}/Instruction × Miss penalty_{L2})

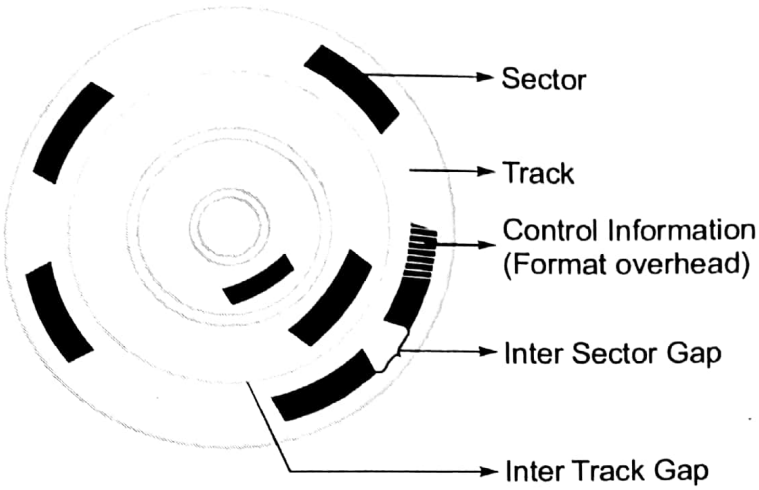
Note:

- There are three types of caches based on whether the index or tag correspond to physical or virtual addresses.
- **Physically indexed, physically tagged:** Caches use the physical address for both the index and the tag.
- **Virtually indexed, physically tagged:** Caches use the virtual address for the index and the physical address in the tag.
- **Virtually indexed, virtually tagged:** Caches use the virtual address for both the index and the tag.

Secondary Memory

- It is often required to have larger programs which exceeds the size of main memory.

  Example: Operating System.
- A magnetic disk is a thin circular metal plate, the data on the disk surface is organised as



- A set of concentric circles is called "tracks".
- Each track holds same number of manageable units called sectors.
  The universal size of a sector is 512 Bytes.
  The basic unit of transfer from the disk is a "sector".
- The disk space without control information is called formatted disk space.
- The disk is a semi random access memory and data is distributed across circular tracks and sectors. Two types of disk construction:
  (*i*) **Constant Track Capacity Disk:** In this case variable recording density is used across all the tracks and the disk has to move with angular velocity.
  (*ii*) **Variable Track Capacity Disk:** In this case constant recording density is used.
- Each rotation of the disk covers 1 track.
- The data transfer rate depends on rpm.
- **Seek Time (t_s):** The time to move Read/Write (R/W) head to the desired track.
- **Rotational Latency (t_r):** The time to move Read/Write (R/W) head to the desired sector beginning.

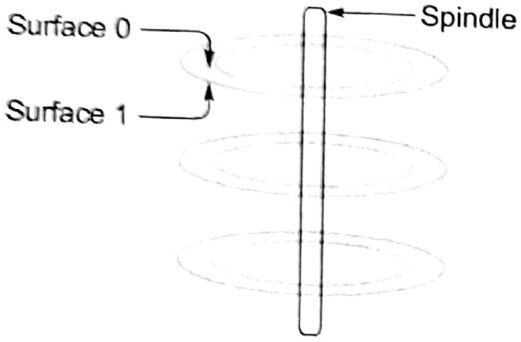
By default $t_r = \frac{1}{2} \times$ rotation time

t = access time of the disc = $t_s + t_r$

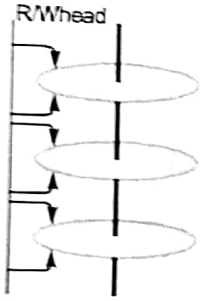
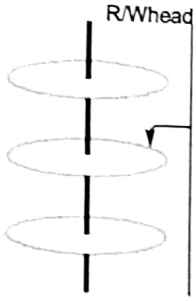
$$t_{avg} = t_s + t_r + t_{data\ transfer} + t_{overhead}$$

- The storage density **ρ = Number of Bytes/cm**
  ρ is maximum at innermost track.

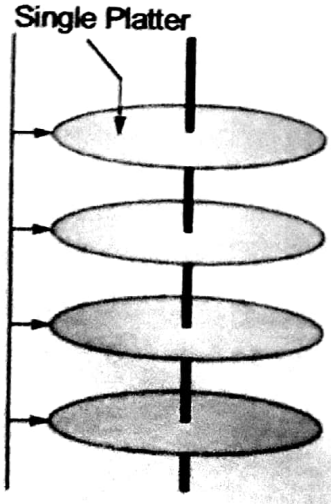
- The data transfer rate of the disc
  D = Number of Bytes/sec
- Multiple disks can be organised as one above another.

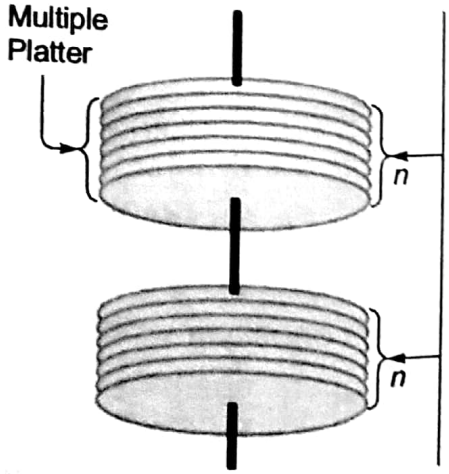


- **Characteristics of disks:**
  (*i*) **Single sided/Double sided disks:** By default double sided disks are used, in which recording is done on both the surfaces.
  (*ii*) Fixed Read/Write (R/W) head or Movable R/W head

Fixed R/W Head	Movable R/W Head
 (a) One R/W head per surface (b) Each rotation covers 'n' tracks n = no. of surfaces (c) Hardware increases and hence cost is more	 (a) One R/W head for all surface (b) Each rotation covers 1 track (c) Less hardware required but access time increases

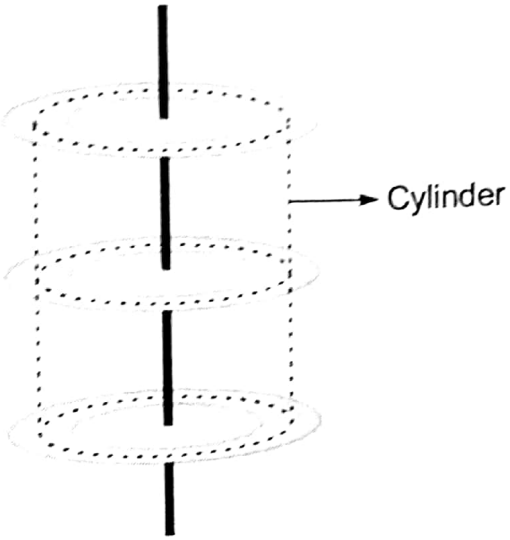
(*iii*) Single platter (or) Multiple platter disks





- A vertical set of all the tracks from the same position in a disk pack forms a cylinder.

Number of cylinders = Number of tracks.



Note:

- Capacity = Track × Sectors/Track × Bytes/Sector.
- $\text{Capacity}_{\text{formatted}}$ = Tracks × Sectors/Track × (Bytes/Sector – Format overhead).

■■■