23/06/2017

Computer Organisation:

#RISC Pipeline: 5 stage pipeline → only index & register addressing mode

→ To analyse the implementation issues of a pipeline, let us consider RISC pipeline as a reference model.

→ RISC processor supports 5 stage pipeline, used to execute the instructions.

→ Different types of instructions possible in a RISC CPU is as follows:

① **Data Transfer instruction**: In the RISC CPU, load and store instructions are used as a data transfer instructions to carry the data between the memory and registers respectively.

→ Memory references implemented using index addressing mode.

Syntax:

Load	r0	3(r1)	⇒	r0 ← m[3 + [r1]]

Store	3(r1)	r0	⇒	M[3+[r1]] ← r0

② **Data Manipulation Instruction**: In the RISC CPU, ALU operations are performed on a registes data.

Syntax:

Add	r0	r1	r2	⇒	r0 ← r1 + r2

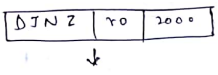
③ **Transfer of control instruction:**

Syntax: ⓐ Unconditional Jump

Jmp	2000	⇒	PC: ~~Sequential address~~ 2000 replaced with target address

ⓑ Conditional Jump:

Syntax:



Dec r0

NZ compare r0 (status of zero flag)

TRUE FALSE

PC: ~~sequent addr~~ 2000 PC: Seq. address

→ To execute the above set of instructions 5 stage instruction pipeline is used in the RISC CPU.

→ Different stages present in the pipeline is as follows:

* Stage 1: Instruction fetch (IF):
In this stage, CPU reads the instruction from the memory based on the PC simultaneously, PC will be incremented to next instruction address in a sequence.

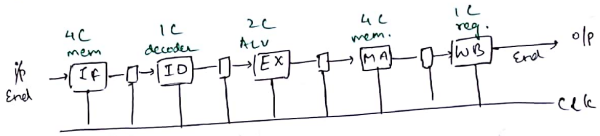
* Stage 2: Instruction Decode (ID):
In this stage, two operations are performed:
ⓐ Decode the instruction
ⓑ Access the register file. to read the data (operand fetch)
This stage also contain the comparator circuit to evaluate the branch condition

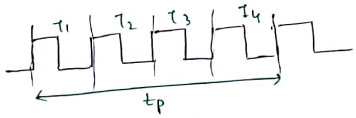
* Ⓔ Stage 3: Execute (EX):
In this stage, ALU operations are performed.

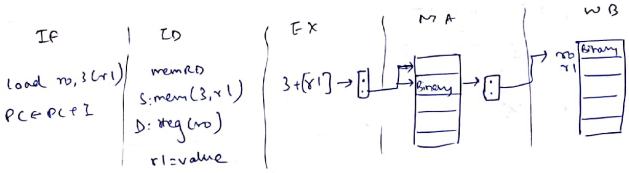
* Stage 4: Memory Access (MA):
In this stage, memory will be accessed to read or write the data.

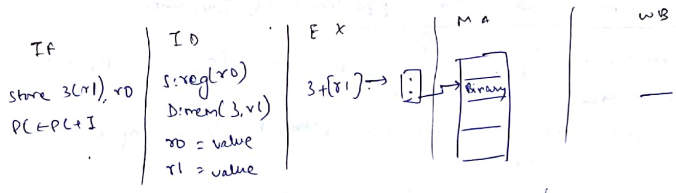
* Stage 5: Write Back (WB):
In this stage, register write operation is performed to store the result.

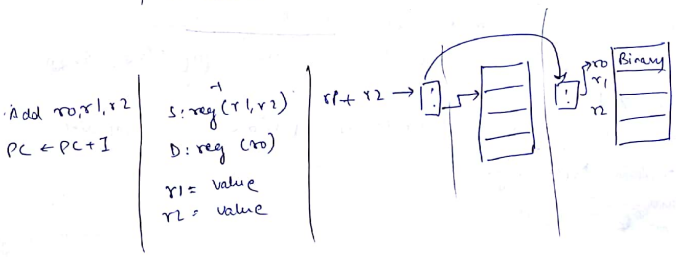
→ In the RISC CPU, above stages are connected in a pipeline sequence, becomes 5-stage instruction pipeline.











IF	ID	EX	MA	WB
Jmp 2000	Uncond. Toc	—	—	—
PC ← PC + 1	TA : 2000 ↓ PC: ~~seq~~ 2000			

IF	ID	EX	MA	WB
DJNZ R0, 2000	Cond Toc	—	—	—
PC : PC + 1	Cond : NZ			
	R0 = value ↓ Dec R0 ↓ NZ compare R0			
	true: PC: ~~seq~~ 2000	false: PC: seq.		

* # Dependencies in the pipeline: (*) Important.

→ Dependency is a major problem in the pipeline.
→ It causes extra cycles.
→ A cycle in the pipeline without new input is called as extra cycle.
→ It is also named as stall cycle.
→ When there is a stall in the pipeline, then CPI ≠ 1.
→ There are 3 kinds of dependencies possible in the pipeline.
① Structural Dependency.
② Data
③ Control

Structural Dependency:

→ Structural dependency will occur in the pipeline due to a resource conflict.
→ Resource may be a register or memory or function unit(ALU)

Diagram 1

	CC1	CC2	CC3	CC4
I_1	mem	ID	ALU	mem
I_2		mem	ID	ALU
I_3			mem	ID
I_4				mem

Conflict

→ In the above diagram, I_1 and I_4, both of the instructions are trying to access the same resource (memory) in the same cycle (CC4). This situation is called as Resource Conflict.
→ Conflict is a unsuccessful operation.
→ To make it successful, we need to keep the I_4 instruction into a waiting until the resource becomes available.
→ This waiting creates stalls in the pipeline, i.e., described below.

Diagram 2

	CC1	CC2	CC3	CC4	CC5	CC6	CC7
I_1	mem	ID	ALU	mem	WB		
I_2		mem	ID	ALU	mem	WB	
I_3			mem	ID	ALU	mem	WB
I_4				///	///	///	mem

7 cycles − 4 i/p = 3 stalls

→ To minimize the structural stalls hardware technique is used, i.e., renaming.
→ In diagram-1, I1 instruction accessing the memory in a 4th stage of pipeline to read or write the data. Simultaneously, I4 instruction accessing the m/m in the 1st stage of pipeline to read the instruction in the same cycle of CC4.

→ When the instruction and data, both are present in the same m/m, then, the above situation creates conflict.

→ Re-naming mechanism states that divide the m/m chip into a two- independent modules used to store instruction and data seperately called as code memory (CM) and data memory (DM) respectively..

→ Refer the code mem. in the first stage and refer the DM in the 4th stage of a pipeline. So, accessing of these two memories in a same cycle does not creates conflict, i.e, described below:

	cc1	cc2	cc3	cc4	cc5	cc6	cc7
I1	CM	ID	ALU	DM	WB		
I2		CM	ID	ALU	DM	WB	
I3			CM	ID	ALU	DM	WB
I4				CM	ID	ALU	DM
I5					CM	ID	ALU
I6						CM	ID
I7							CM

Diagram 3

7 cycles - 7 i/ps = 0 stalls

Data Dependency:

→ Consider the program segment, where instruction 'j' follows instruction 'i' in a program order.

Neg. I instruc. / J instruc.

R-B-W → Read before write → Data loss

→ Data dependency condition will occur when the instruction 'J' tries to read the data before instruction 'I' writes it.

Eg: I_1 : ADD r0, r1, r2 ⇒ r0 ← r1 + r2

I_2 : MUL r3, r0, r4 ⇒ r3 ← r0 * r4

→ When the above instructions are executed in a non-pipeline processor, then data dependency condition does not occur, because, I_2 is executing after I1, so, I2 tries to read the register ~~r0~~ r0 data after I1 writes it. (No data loss)

→ When the above instructions are executed in a pipelined processor, then, data dependency condition will be occcured. because I2 is executing alorgwith I1, so, I2 tries to read the register r0 data before I1 writes it. So, I2 incorrectly reads the old value from the r0 (Data loss).

	cc1	cc2	cc3	cc4
I_1	IF	ID	EX	
I_2		IF	ID * S: r0, r4 D: r3 r0: value r4: value	old value → Data loss (Read Before write)
I_3			IF	

→ To correct the above problem, we need to stop the accessing of a dependent data until the previous instruction, (dependent) is completed.

→ for this purpose, some logic will be maintained in the ID stage.

Structure of the logic:

ID stage: TOMASULO Algorithm:

S/No.	Function unit	Destination	Independent source 1	Inde. src 2	Dependent src 1	Dep. src 2	Status
I1	ADD	r0	r1	r2	—	—	0 (r1: value, r2: wt. → stage allocated) EX
I2	MUL	r3	—	r4	r0 (I1)	—	0 (r4: value; r0: stage not allocated)

→ By using the above logic, we can lock the dependent ins. (I2) in the ID stage until the data becomes available.

→ This process creates stalls in the pipeline, i.e., described below.

7 cycles − 4 I/Ps = 3 stalls (Data dependency stalls)

→ To minimize the data dependency stalls, hardware techniques used, i.e., operand forwarding also known as bypassing/short circuiting.

→ This technique states that use the buffer between the stages to hold the intermediate outputs. So, the dependent instruction will be accessing the new value from the interface register (buffer) before update the register file.

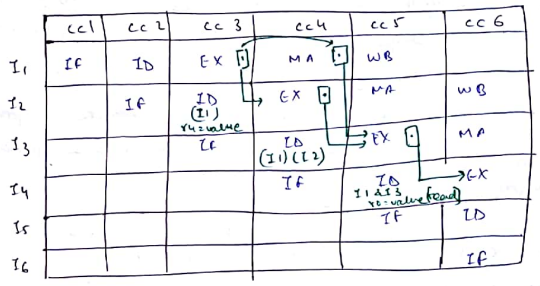
→ Consider the following program segment.

I1: ADD r0, r1, r2
I2: SUB r3, r0, r4
I3: MUL r5, r0, r3
I4: DIV r6, r0, r5

—— True Data dependency (Adjacent)

---- Data Dependency (Non-adjacent)

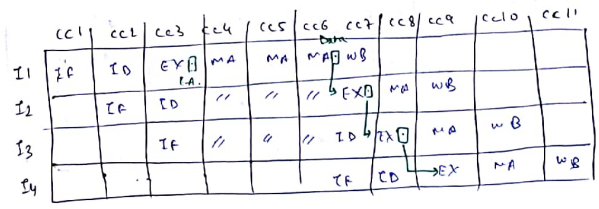
→ Execution sequence of a program with operand forwarding is as follows:



Q. Consider the following program code, executed on a RISC pipeline. All the instructions one spending one cycle on all the stages. but load instruction spending 3 cycles on 4th stage of pipeline. How many cycles are required to complete the program code.

I1: Load r0, 3(r1)
I2: Add r3, r0, r2
I3: MUL r4, r3, r2
I4: DIV r5, r4, r2

Solution: RISC pipeline → (k = 5 [IF, ID, EX, MA, WB]; → operand forwarding is default)



Total cycles = 11

Q. Consider 4 stage pipeline (IF, ID, EX, WB) used to execute the following program where all the instructions are spending 1 cycle on all the stages but DIV ins. spends 4 cycles on EX stage and MUL spends 3 cycles on EX stage. Operand forwarding is used in the pipeline.

(a) How many cycles are required to complete the program
(b) How many cycles are saved using the operand forwarding over w/o operand forwarding.

Code:

I1 : DIV R0, R1, R2
I2 : MUL R3, R4, R5
I3 : ADD R9, R3, R5
I4 : SUB R5, R6, R2

	IF	ID	EX	WB
I_1	1	1	4	1
I_2	1	1	3	1
I_3	1	1	1	1
I_4	1	1	1	1

(a)

	cc1	cc2	cc3	cc4	cc5	cc6	cc7	cc8	cc9	cc10
I1	IF	ID	EX	EX	EX	EX	WB			
I2		IF	ID	//	//	//	EX	EX	EX	WB
I3			IF	//	//	//	ID	//	//	EX
I4				IF	//	//	//	ID	EX	WB

Total cycles = 10

	cc1	cc2	cc3	cc4	cc5	cc6	cc7	cc8	cc9	cc10	cc11	cc12
I1	IF	ID	EX	EX	EX	EX	WB					
I2		IF	ID	//	//	//	EX	EX	EX	WB		
I3			IF	//	//	//	ID	//	//	EX	WB	
I4				IF	//	//	//	ID	//	//	EX	WB

(b) w/o operand forwarding

	cc1	cc2	cc3	cc4	cc5	cc6	cc7	cc8	cc9	cc10	cc11	cc12	cc13	cc14	cc15	cc16
I1	IF	ID	EX	EX	EX	EX	WB									
I2		IF	ID	//	//	//	EX	EX	EX	WB	EX	WB				
I3			IF	//	//	//	ID	//	//	//	//	EX	WB	EX	WB	
I4				IF	//	//	//	//	//	//	ID	//	ID read	EX	WB	

Cycles saved = 4

24/06/2017:

Control Dependency:

→ Control dependency will be occured in the pipeline due to a jump instructions.

→ Consider a program segment

1000 : I1
1001 : I2
1002 : I3 (Jmp 2000)
1003 : I4
⋮
2000 : BI1
2001 : BI2

(fall-through path / Taken path)

Expected o/p sequence

$I_1 - I_2 - I_3 - BI_1 - BI_2$

PC:4000	cc1	cc2	cc3	cc4	cc5	cc6	cc7
I1	IF PC:1001	ID PC:1001	EX	MA	WB		
I2		IF PC:1002	ID	IX	MA	WB	
I3			IF PC:1003	ID Undord PC:2000	EX	MA	
I4				IF PC:1004	ID	EX	MA
BI1					IF	ID	EX
BI2						IF	

unwanted

$I_1 - I_2 - I_3 - I_4 - BI_1 - BI_2$

Actual o/p sequence.

→ In the above execution sequence, unwanted instructions are overlapping with the Jump instruction; so, the program functionality is missing.

→ This kind of problem in the pipeline is called as control dependency.

→ To correct the above problem, there is a need of inserting the delay slot (NOP) instruction) after the jump instruction.

→ NOP insertion in the program is called as Flush/freeze.

→ Flush operation creates the stalls in the pipeline, i.e., described below:

Modified Code:

```
1000 : I1
1001 : I2
1002 : I3 (JMP 2000)
1003 : NOP
1004 : I4
  :
2000 : BI1
2001 : BI2
  :
```

Expected o/p

$I_1 - I_2 - I_3 - BI_1 \& BI_2$

PC:1000	CC1	CC2	CC3	CC4	CC5	CC6	CC7
I_1	IF PC:1001	ID	EX	MA	WB		
I_2		IF PC:1002	ID	EX	MA	WB	
I_3			IF PC:1003	ID	EX	MA	
NOP				IF PC:1004	ID	EX	
BI_1					IF PC:2001	ID	
BI_2						IF PC:1002	

O/P → $I_1 - I_2 - I_3 - NOP - BI_1 - BI_2$

→ No. of stalls created in the pipeline due to a jump instruction is called as branch penalty.

→ It depends on the availability of a target address in the pipeline, i.e.,

* At what stage target address is available in the pipeline — 1

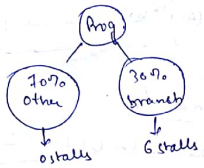
NOTE:

→ RISC pipeline branch penalty is always 1 because target address is available in the second stage of the pipeline.

Q. Consider 8-stage pipeline which overlaps all the instructions except the branch instruction. Means when the instruction is a branch instr. then processor stops the fetching of the following instructions until the target address is available. In this pipeline target address is available in the 7th stage. How many stalls are present in the pipeline. When program contains 30% branch instructions, then, how many stall cycles are created during the execution of a programs.

8-stage pipeline

- All other instruc. are optimized → No stalls
- Branch instructions are not optimized → Stalls: 7 - 1 = 6



Stalls/Branch = 0.3 * 6 = 1.8

→ No. of stalls created in the pipeline due to a branch instruction during a program execution is calculated as.

Branch frequency * Branch Penalty

(Branch frequency: % branch ins. in the prog.; Branch Penalty: stalls per branch.)

→ To handle the control dependency stalls hardware technique is used, i.e., branch prediction buffer. Also known as branch target buffer or loop buffer.

NOTE: → It is a high speed buffer maintained in the first stage of the pipeline, used to hold the predicted target addresses.
→ When the prediction is true, then stalls are not present, otherwise stalls are created.
→ It is not a optimised solution. So, not in the practical systems.

NOTE: → When the question contains pipeline with a branch prediction, then, assume that target address is present in the first stage, so, branch penalty becomes zero, otherwise, (w/o branch prediction), then, assume that target address is not present in the first stage. So, branch penalty is calculated based on the target address availability.

NOTE: → To handle the control dependency stalls, one of a software technique is used, i.e., delayed branch.

Delayed Branch:

→ It is a compiler technique, so, compiler will rearrange the program if possible or substitute the NOP instruction after the Jump instruction, if not possible to rearrange, to handle the control dependency problem.

Mem Prog:

1000 : I1
1001 : I2
1002 : I3 (JMP, 2000)
1003 : I4
:
2000: BI1
2001: BI2

Expected O/P:

I1 - I2 - I3 - BI1 - BI2

Actual:

I1 - I2 - I3 - I4 - BI1 - BI2

COMPILER "Re-arrangement":

Code:

Machine generated address:
I1
I3 (JMP BI1)
I2
~~I4~~
:
BI1
BI2

PCI1	cc1	cc2	cc3	cc4	cc5
I1	IF PC:I3	ID	EX	MA	WB
I3		IF PC:I2	ID Uncond JOC PC:I(BI1	EX	MA
I2			IF PC:I4	ID	EX
BI1				IF PC:BI2	ID
BI2					PC:BI3 IF

O/P:

I1 - I3 - I2 - BI1 - BI2

"NOP-Substitution":

Machine Generated Address:
I1
I2
I3 (JMP BI1)
NOP
I4
BI1
BI2

PC:I1	cc1	cc2	cc3	cc4	cc5	cc6
I1	IF PC:I2	ID	EX	MA	WB	
I2		IF PC:I3	ID	EX	MA	WB
I3			IF PC:NOP	ID Uncond JOC PC:I(BI1	EX	MA
NOP				IF PC:I4	ID	EX
BI1					IF PC:BI2	ID
BI2						IF PC:BI3

O/P:

I1 - I2 - I3 - NOP (Stall) - BI1 - BI2

Instruction Scheduling:

→ CPU always executes the program in a sequence called as in order execution.

→ In the in order execution sequence, if any instruction is dependent, then the remaining instructions are also sharing the stall cycles.

→ E.g.

I1 : ADD R0, R1, R2
I2 : SUB R3, R0, R4
I3 : MUL R4, R5, R6
I4 : DIV R5, R7, R8

In order execn seq.

I1 - I2 - I3 - I4
stalls

→ I2 is data dependent on I1, so, I2 will be waiting until the I1 execution is completed. This waiting creates stalls. These stalls are also shared by the I3 and I4 instructions, even they are independent.

→ Instruction scheduling concept is used to handle the above situation. It states that:

"Inserting the independent instructions first, to utilise the stalls."

→ It causes out of order (re-order) execution, i.e., I1 - I3 - I4 - I2

→ Out of order execution creates 2 more data dependencies in the pipeline:

① ANTI DATA DEPENDENCY ② OUTPUT DATA DEPENDENCY

*

Instn I		Instn J	
Read	before	write	[TRUE DATA LOSS]
write	before	Read	[ANTI DATA LOSS]
write	before	write	[O/P DATA LOSS]

before → After → Delay → Hazard

→ Anti data dependency will be occured in the pipeline, when the instruction J tries to write the data before instrn. I reads it.

E.g. I3 is executing before I2. So, I3 updates the register R4 before I2 reads it. Therefore, I2 incorrectly gets the new value from the R4 (Data loss).

→ O/P Data dependency, will be occured when the instruction J tries to write the data before instruction I writes it

E.g. I4 is executing before I2, so, I4 updates the register R3 before I2 writes it. Therefore, destination register R3 is incorrectly updated with a old value (DATA LOSS).

→ To handle the above dependency problems, h/w technique is used, i.e., Register Re-naming.

→ This technique states that:

* "Use the re-order buffer to store the o/p of a out of order instructions. Later, update the register file with a re-order buffer contents after the execution of a dependent instruction."

Register Re-Naming
re-order buffer
i.e., Execn seq.

I1 → R0
I3 → re-order buffer
I4 → re-order buffer
I2 → R3 (status = 1)

Re-order buffer → Reg. file (R3 & R4)

Hazards: → Hazard is a delay.

→ Delay is created in the pipeline due to a dependency problem. So, hazard is divided into 3 types:

1. Structural Hazard
2. Data Hazard
3. Control Hazard

→ Data Hazard is further divided into 3 types, based on the order of the read and write operations, named as:

1. RAW Hazard created because of RBW
2. WAR Hazard " " " WBR
3. WAW Hazard " " " WBW

→ RAW Hazard is created when the instruction 'J' tries to read the data before instruction I writes it. (TRUE data dependency)

→ WAR Hazard is created when the instruction 'J' tries to write the data before instruction I reads it (ANTI data dependency)

→ WAW Hazard is created when the instruction 'J' tries to write the data before instruction I writes it. (o/p data dependency)

Instruction J	Instruction I	
In-reg.	Out-Reg.	[RAW]
Out reg.	In-Reg	[WAR]
Out reg	Out. Reg	[WAW]

Note:
RAR [Read After Read] is not a hazard because RBR is not a data dependency.

Performance Evaluation with stalls:

CPI → Cycle per ins.

$$S = \frac{\text{Average instruction execution time}_{\text{non pipeline}}}{\text{Average instruction execution time}_{\text{pipeline}}}$$

$$S = \frac{CPI_{\text{non-pipe}} * \text{cycle time}_{\text{non-pipe}}}{CPI_{\text{pipe}} * \text{cycle time}_{\text{pipe}}}$$

$$S = \frac{n t_n}{(k+n-1).t_p} = \frac{t_n}{t_p}$$

$$\boxed{S = k}$$

→ Ideal CPI of the pipeline is always 1 (one). But, due to a dependency problem stall cycles are created in the pipeline. So,

$$S = \frac{CPI_{\text{non-pipe}} * \text{Cycletime}_{\text{non-pipe}}}{(1 + \text{No. of stalls/instr.}) * \text{Cycletime}_{\text{pipe}}}$$

→ When the pipeline stages are perfectly balanced, then, one task execution time in the non-pipeline is also equal to no. of stages in the pipeline, i.e.,

$[\because t_n = k \text{ cycles} = k \cdot t_p]$ So,

$$S = \frac{\text{No. of stages in pipeline} * \text{Cycletime}_{\text{pipe}}}{(1 + \text{no. of stage/instr.}) * \text{cycle time}_{\text{pipe}}}$$

$$\boxed{S = \frac{\text{No. of stages in pipeline [Pipeline depth]}}{[1 + \text{No. of stalls/instru.}]}}$$

When the system is operating with 100% efficiency then no stalls present, So,

$$\boxed{S = \text{Pipeline depth}}$$

Q. Consider 5 stage pipeline which allows overlapping all the instructions except the branch instructions. Processor stops the fetching of a following instructions after the branch instruction until the instruction is completed. In the program, 40% of instructions are branch instructions. Among them, 60% of instructions are conditional branch. In which, 30% instructions does not satisfies the condition. When the condition is ~~false~~ false, then the following instructions are overlapped: Pipeline is operated with a 4 ns clock.

(a) What is the average instruction execution time.

(b) What is the performance gain.

**Target address availability:

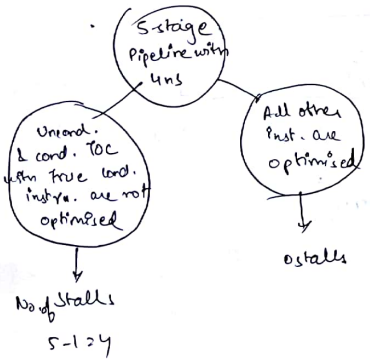
(1) Stage number ⇒ Branch Penalty = (Stage no. − 1)

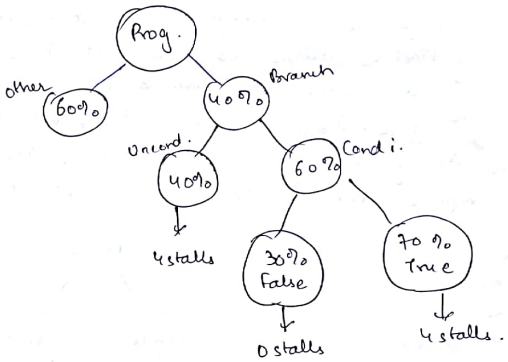
(2) Stage Name ⇒ BP = (Corresponding stage no. − 1)

(3) Until the Inst. is completed ⇒ BP = (last stage no. − 1)

(4) All the ins. are processed with all stage

⇒ BP = last stage number − 1





No. of stalls / instructions = (0.6 * 0)
+ (0.4 * 0.4 * 4)
+ (0.4 * 0.6 * 0.3 * 0)
+ (0.4 * 0.6 * 0.7 * 4)

= 1.312

(a) Average Ins. E.T. = (1 + No. of stalls/ins.) * cycle time

= (1 + 1.312) * 4 ns

= 9.248 ns

(b) Speed up (S) = Pipeline depth / (1 + No. of stalls/ins.)

S = 5 / (1 + 1.31)

S = 2.16

Q. Consider 6 stage pipeline which is operating with a 2ns clock. Pipeline allows all the instructions except branch ins. Target address of the branch ins. is available in the 4th stage. Program contains 40% branch instructions. Among them, 70% of instructions are optimised with a compiler. What is the avg. ins. exec. time?

No. of stalls = 3

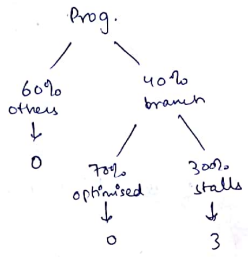
~~6 stage~~ stage with 2 ns

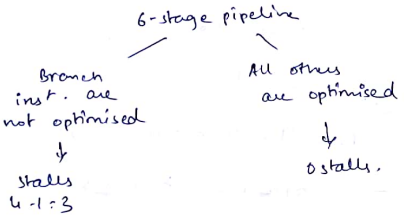
No. of stalls/ins. = 0.4 * 0.3 * 3

~~Avg time = 1+~~ = 0.36

Avg time = (1 + 0.36) * 2 ns

= 2.72 ns



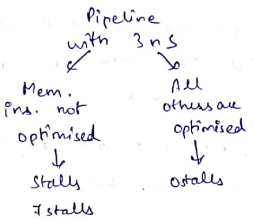


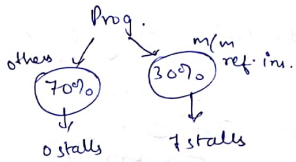
Q. Consider a hypothetical pipeline which allows all the instructions except memory instructions. Penalty associated with a m/m instruction is 7 cycles. ~~Total cost~~ Program contains 30% of memory instructions. Pipeline is operating with 3 ns clock. What is the average instruction execution time.

No. of stalls/ins. = 0.3 * 7 = 2.1

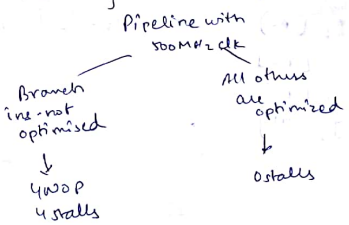
Avg. exec. time = (1 + 2.1) * 3

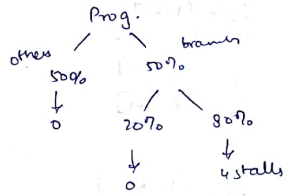
= 9.3 ns





Q. A hypothetical pipeline uses the delayed branch technique to handle the control dependency. Compiler substitutes the 4 NOP instructions after the branch instructions to manage the control dependency. Program contains 50% branch ins. Among them, compiler optimises 20% instructions. Pipeline contains 8 stages and operating with 500 MHz clock. What is the average instruc. exe. time.





No. of stalls/ins. = 0.5 * 0.8 * 4

= 1.6

~~Avg. exec.~~ Clock = $\frac{1}{5 \times 10^8}$ s

= 2 ns

Avg. exec. time = (1 + 1.6) * 2 ns

= 5.2 ns

Q. Consider the following program code executed on a hypothetical pipeline:

Concept → Only True data dependency is considered.

Test → Adjacent data dependency is also considered.

I_1 : R0 ← R1 * R2
I_2 : R1 ← R0 + R3
I_3 : R0 ← ~~R3 * R4~~ R1 / R0
I_4 : R2 ← ~~I1~~ ~~I4~~ R3 * R4
I_5 : R1 ← R2 − R4
I_6 : M[x] ← R1

How many RAW, WAR and WAW hazards are possible in the above code.

RAW
In-our
$I_2 - I_1$ [R0]
$I_3 - I_2$ [R1]
$I_4 - I_4$ [R2]
$I_6 - I_5$ [R1]

WAR:
Out-in:
$I_2 - I_4$ [R1]
$I_3 - I_2$ [R0]
$I_4 - I_1$ [R2]
$I_5 - I_1$ [R1]
$I_5 - I_3$ [R1]

WAW,
out-out;
$I_3 - I_1$ [R0]
$I_5 - I_2$ [R1]

Ans. (4, 5, 2)

NON-LINEAR PIPELINE:

→ This pipeline contains forward and backward connections. So, reservation table is used to process the entries.

→ Let us consider, a sample reservation table used to execute the I/ps in a hypothetical non-linear pipeline.

	1	2	3	4	5
S1	X				X
S2		X		X	
S3			X		

→ Multiple checkmarks in the row indicates repeated use of same stage in different cycles.

→ Continuous checkmarks in the row indicates extended use of same stage upto some number of cycles.

→ Multiple checkmarks in the column indicates parallel use of different stages in a particular cycle.

NOTE:
→ In the linear pipeline, latency is always 1.
→ In the non-linear pipeline, latency will be different (varies), calculated based on the reservation table.

Latency Analysis: → Latency means time difference between the two successive initiations in the pipeline.

→ It is a positive integer.
→ In the non-linear pipeline, some of the latencies do cause conflicts called as forbidden latency (Non-permissible) and some of the latencies do not cause the conflict called as non-forbidden latency (permissible).
→ To detect the forbidden latency, we need to take ~~any~~ the difference between any two checkmarks in the same row.

E.g: row 1 ⇒ [5 − 1] = 4
row 2 ⇒ [4 − 2] = 2

→ Based on the forbidden latency, collision vector will be generated.

Collision vector:

i.e., $[C_i : C_n \ldots C_3 C_2 C_1]$

$C[$ 0 → Permissible, 1 → Non-permissible $]$

Collision Vector → $\begin{bmatrix} C_5 & C_4 & C_3 & C_2 & C_1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$

*Collision vector only gives the initial latency.

→ Latency sequence is identified based on the collision vector to specify the correct flow of initiation, i.e.,

ⓐ Latency sequence w.r.t C_1:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S1	1	2			1	2	3	4			3	4	5	
S2		1	2	1	2			3	4	3	4			5
S3			1	2					3	4				

Latency cycle:

(2-1) = 1

(7-2) = 5

(8-7) = 1

(13-8) = 5

(1,5),(1,5), . . .

Average latency = $\frac{1+5}{2} = 3$

ⓑ Latency sequence w.r.t. C_3:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S1	1			2	1		3	2		4	3		5	4
S2		1		1	2		2	3		3	4		4	5
S3			1			2			3			4		

Latency cycle:

(4-1) = 3

(7-4) = 3

(10-7) = 3

3, 3, 3, . . .

Average latency = 3

ⓒ Latency sequence w.r.t. C_5:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
S1	1				1	2	3			2	3	4	5			4	5
S2		1		1			2	3	2	3			4	5	4	5	
S3			1					2	3					4	5		

Latency cycle:

(6-1) = 5

(7-6) = 1

(12-7) = 5

(13-12) = 1

(5,1),(5,1) . . .

Avg latency = $\frac{5+1}{2} = 3$

Q. What is MAL? Minimum Avg. latency.

Soln = 3.

MEMORY ORGANISATION:

Types of Memory Organisation:

→ Based on the style of accessing the system supported memories, memory organisation is divided into two types:

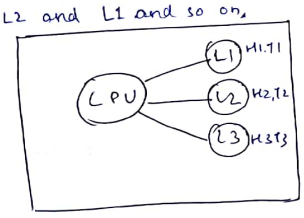
① Simultaneous Access memory organisation
② Hierarchical Access

Simultaneous Access:

→ In this organisation, CPU directly communicating with all the levels of memories to access the data in a sequence, i.e., CPU request is initially forwarded Level 1 memory.

→ When there is a miss in level 1 (L1) m/m, then, CPU directly accesses the data from the L2 memory, without copying into L1.

→ When there is a miss in L2 memory, then, CPU directly accesses the data from L3 memory, w/o copying it into ~~to~~ L2 and L1 and so on,



Here,

$H_1, H_2, \ldots H_n$ → Hit ratios of the respective m/ms

$$\text{Hit ratio} = \frac{\text{No. of Hits}}{\text{Total no. of accesses}}$$

$T_1, T_2 \ldots T_n$ → Access time of a respective m/m.

→ Amount of time required to access one word data from the memory is called as average access time (T_{avg}).

$$T_{avg} = H_1T_1 + (1-H_1).H_2T_2 + (1-H_1)(1-H_2).H_3T_3 + \ldots + (1-H_1)(1-H_2)\ldots(1-H_{n-1})H_nT_n$$

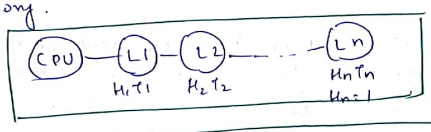
→ 1 word ⇌ T_{avg}
No. of words ⇌ 1 sec

$$\eta_{mem} = \frac{1}{T_{avg}} \text{ words/sec}$$ (Throughput)

Hierarchical Access:

→ In this organisation, CPU always accessing the data only from the L1 memory. If data is not in L1 (miss), then, the respective data will be transfered from higher levels to L1 memory.

→ Later, CPU will be accessing the data only from the L1 memory.



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

Prog:
I_1: Load (Data in L2 m/m)
I_2
I_3: Load [I]
I_4
I_5: Load [I]
I_6
I_7: Load [I]
I_8
I_9: Load [I]
I_{10}

reuse the same data

$P \propto \frac{1}{RT}$

Simultaneous
I_1 : M_{L1}, T_2
I_3 : M_{L1}, T_2
I_5 : M_{L1}, T_2
I_7 : M_{L1}, T_2
I_9 : M_{L1}, T_2
5 times L2 m/m is accessed

Hierarchical
I_1 : $M_{L1}, (T_2+T_1)$
I_3 : T_1
I_5 : T_1
I_7 : T_1
I_9 : T_1
Temporal locality
$5 \times T_1 + T_2$

NOTE:

→ In the Hierarchical memory organisation, CPU will be accessing the higher levels of memory data from the L1 memory.

→ When the application is developed with a reuseability concept, then, in this organisation, higher levels memory data is available in the first level itself. This principle is named as Locality of reference.

→ It is of two types:

① Temporal locality ② Spatial Locality

→ Temporal means same word of data will be referred by the CPU again and again in the near future.

→ Spatial means adjacent words in the block are referred by the CPU in a sequence.

→ In the computer system design, hierarchical memory org. is used to implement the memory system (simultaneous access memory org. is not in practice).

NOTE:

→ When the question contains hierarchy word or hierarchy meaning or cache memory word, then use hierarchical m/m organisation.

→ Otherwise, use simultaneous m/m organisation.

Q. In a two-level m/m design, Level 1 m/m is 8 times faster than L2 memory and its access time is 30ns less than the average access time. Let L1 access time is 20ns. What is the hit ratio.

$H_2 = 1$ → since, data is definitely present in L2.

$T_{avg} = 50ns$

$T_{avg} = H_1 T_1 + (1-H_1) H_2 T_2$

$50 = H_1 \cdot 20 + (1-H_1) H_2 \cdot 160$

$50 = H_1 \cdot 20 + (1-H_1) \cdot 160$

$50 = 20H_1 + 160 - 160H_1$

$-110 = -140H_1 \Rightarrow H_1 = \frac{110}{140} = 0.785$

Q. 3-level m/m org. has the following specifications.

Level	Access time per word	Block size in words	Hit ratio	Access time per block
1	20ns	—	0.7	$T_1 = 20ns$
2	100ns	2	0.9	$T_2 = 200ns$
3	200ns	4	1	$T_3 = 800ns$

If the refered block is not in L1, then transfer it from L2 to L1. If not in L2, then transfer it from L3 to L2 and L2 to L1. How long will it take to access the data block?

Soln:

$T_{avg} = H_1 T_1 + (1-H_1) H_2 (T_2 + T_1) + (1-H_1)(1-H_2) H_3 (T_3 + T_2 + T_1)$

$= 0.7 * 20 + (0.3).0.9.(220) + (0.3).(0.1).1.(1020)$

$= 14 + 59.4 + 30.6$

$= 104 ns$

Q. In a 2-level hierarchical m/m design, L1 m/m access time is 40ns and it is referenced by the CPU 80% of time. L2 m/m accessing time is 200ns, what is the avg. access time of m/m.

$T_{avg} = H_1 T_1 + (1-H_1) H_2 (T_2 + T_1)$

$= 0.8.40 + (0.2) 1.(240)$

$= 32 + 48$

$= 80 ns$

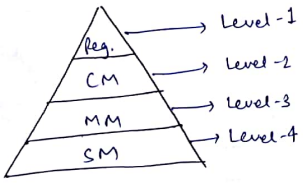
Memory Hierarchy Design:

→ In the computer design memory system is organised using the hierarchical accessing principle. So, system supported memories are organised into 4 levels.

→ According to a hierarchy design, system supported m/m standards is as follows:

Level : 1	2 (CM)	3 (MM)	4 (SM)
Name : Register	Cache memory	Main memory	Secondary m/m
Typical Size : < 1kB	< 16MB	< 16GB	> 1004B
Implementation : Customised Multiports	SRAM (flip-flop)	DRAM (Capacitor)	Magnetic
Access time : (0.25 - 0.5)	(0.5 - 25)	(80 - 250)	50,00,000
Bandwidth (MB/sec) : (20,000 - 1,00,000)	(5000 - 10000)	(1000 - 5000)	(20 - 150)
Managed by : Compiler	H/w	O.S.	O.S
Backed by : CM	MM	SM	Compact Disk (CD)

Memory Hierarchy Design:

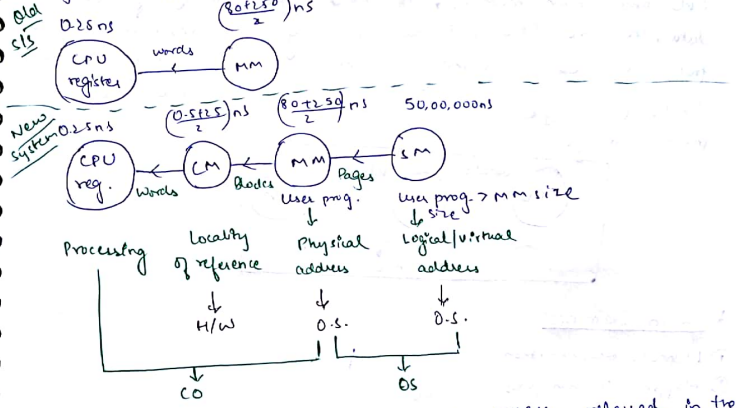


Bottom-Top Approach:

① Capacity is ↓
② Access time is ↓
③ Performance is ↑
④ Cost/bit is ↑
⑤ Expensive

25/06/2017:

→ Accessing sequence of system supported memories according to a hierarchy design is as follows.



→ CPU generated memory request is initially referred in the cache memory. When the data is present in cache, then, operation becomes hit. So, hit data will be transfered to CPU in the form of words. Otherwise, operation becomes miss.

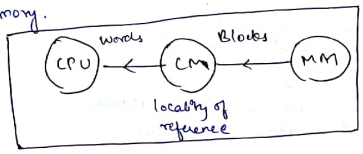
→ So, the reference is forwarded to main memory.

→ When the operation is hit in main memory, then the data will be transfered to cache in a form of blocks. Later, cache to CPU in the form of words. Otherwise, reference will be forwarded to secondary memory.

→ Secondary memory is a final memory in the computer, so, operation is always hit in secondary memory, therefore, hit data is transfered to main memory in the form of pages, main to cache in the form of blocks and cache to CPU in the form of words.

Cache Memory: (Imp.) ***

→ Cache memory is used as a intermediate memory between the CPU and main memory, to hold the image of a main m/m data, therefore, CPU will be accessing the main memory data from the cache memory within a less amount of time. Hence, speed gap is balanced between the CPU and main memory.



* Design Elements:

1. Memory Organisation
2. Mapping Techniques
3. Replacement Algorithms
4. Updating Techniques
5. Multilevel Cache Design

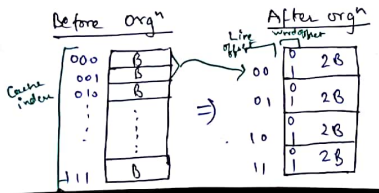
Memory Organisation:

→ In the memory hierarchy design, data will be transferred between the main memory and cache memory in a blockwise manner. So, both of the memories are organised into blocks based on the block size. So,:

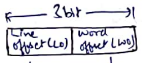
$$\text{No. of blocks in m.m.} = \frac{\text{MM size}}{\text{Block size}}$$

$$\text{No. of blocks (lines) in CM} = \frac{\text{CM Size}}{\text{Block size}}$$

Eg. Consider 8 B cache with 2 B block



Cache m/m addr. format



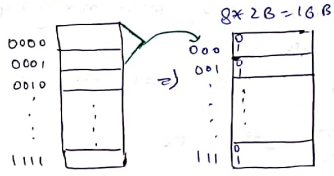
Line no. in CM → $\log_2$ #lines → $\log_2 4$ → 2 bit

word addr. in the blk → $\log_2$ #Blocks size → $\log_2 2$ → 1 bit

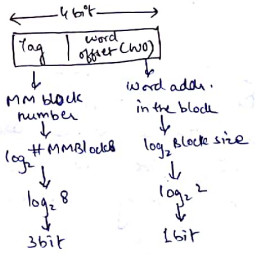
Eg. Consider 16 B MM with 2 B Block.

Before orgn: 16 B of MM

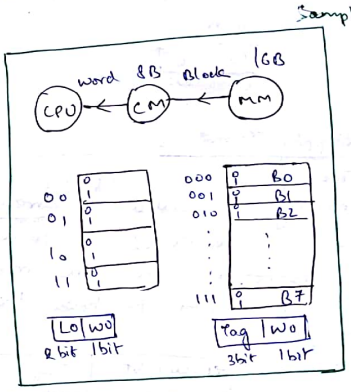
After orgn: #Blocks = $\frac{16}{2} = 8$; $8 \times 2\,B = 16\,B$



Physical Addr. Format:



Sample



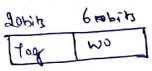
Q. A hypothetical computer supports 64 kB cache memory and 26 bit physical address space. Memory is organised into a 64 byte blocks. How many no. of blocks are present in the cache m/m and in MM respectively. Also show addr. formats.

No. of lines in cache = $\frac{2^{16}}{2^6} = 2^{10}$ blocks

No. of blocks in MM = $\frac{2^{26}}{2^6} = 2^{20}$ blocks

Cache m/m address format:

MM address format:



Q. A hypothetical computer supports 14 bit cache index and 24 bit physical address space. 4 bit word offset is used to access the data from the block. How many no. of blocks are present in the cache memory and main memory respectively. Also show the addr. formats.

No. of Blocks/lines in Cache = $\frac{2^{14}}{2^4} = 2^{10}$ blocks

No. of blocks in MM = $\frac{2^{24}}{2^4} = 2^{20}$ blocks

Cache: | 10 | 4 | (L0, W0)

MM: | 20 | 4 | (Tag, W0)

Mapping Techniques:

→ The process of transfering the data from main mem. to cache memory is called as mapping. During the mapping data block is transfered to cache alongwith a physical address. So, in the cache memory every line contains 3-fields of information to hold the main memory data, i.e.,

| Valid bit | Tag | Data |] line

- Valid bit: Status of a block
  - 0: Data block is not reused/ refered by CPU
  - 1: Data block is reused/ refered by CPU
- Tag: Holds the mm addr. info. ↓ Depends on the type of mapping
- Data: Holds the data block ↓ Depends on the block size

Total cache = Tag mem. size + Data mem. size

Tag mem. size = No. of lines in CM * No. of tag bits in the line

Data mem. size = No. of lines in CM * Block size

→ Cache mem. is designed with 3 kinds of mapping fun.n:

- Direct Mapping
- Associative Mapping
- Set-associative Mapping

Direct Mapping:

→ In this technique, mod function is used to map the data

i mod N = j

- i: MM block number
- N: No. of blocks in CM
- j: CM line number

0 mod 4 = 0
1 mod 4 = 1
2 mod 4 = 2
3 mod 4 = 3
4 mod 4 = 0
5 mod 4 = 1
6 mod 4 = 2
7 mod 4 = 3

→ Mapping function shows the relationship in the main memory block and the cache memory line.

CM lines:

0	0, m, 2m ...
1	1, m+1, 2m+1, ...
m-1	m-1, 2m-1, 3m-1 ...

Address Binding:

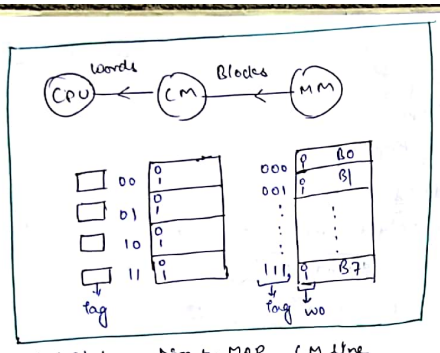
MM: | tag | w0 | (3bit, 1bit)

Direct CM: | tag | L0 | w0 | (1bit, 2bit, 1bit)

No. of MM blocks possible in the line, not at the same time i.e. $2^{tag} = 2^1 = 2$

MM ⇒ Direct CM

MM tag (3bit)	Direct CM (1bit, 2bit)
B0[000], B4[100]	0/1 | 00 → Line '0'
B1[001], B5[101]	0/1 | 01 → Line 1
B2[010], B6[110]	0/1 | 10 → Line 2



MM Block	Direct MAP	CM line
B0 [000]	K mod N = i; 0 mod 4 = 0	line '0'
B7 [111]	7 mod 4 = 3	line '3'
B4 [100]	4 mod 4 = 0	line '0'

→ In the mapping process, some of the tag bits are connected to the address logic of cache memory and rest of the bits are stored in a cache controller as a tag. Therefore,

Tag mem. size = No. of lines in CM * No. of tag bits in the line

= 4 * 1 bit
= 4 bits

→ Explicit replacement techniques are not required in the direct cache design to replace the data because every main memory block is having fixed location in the cache memory.

Q. Consider 32 KB direct mapped cache organised into 32 bytes blocks. Cache memory data is a subset of the 2^{32} bytes memory space. In the cache controller each tag is comprising of valid bit and 1 modified bit.

(a) No. of lines in cache.
(b) No. of blocks in main memory.
(c) Address binding
(d) No. of tag bits in the line
(e) Tag mem. size in the cache controller.

(a) No. of lines = $\frac{2^{15}}{2^5} = 2^{10}$

(b) Blocks in MM = $\frac{2^{32}}{2^5} = 2^{27}$

(c) Address binding:

CM Tag	LO	WO
17	10	5

MM	tag	WO
	27	5

(d) 17 bits

(e) $2^{10} * (17 + 1 \text{ valid} + 1 \text{ modified})$
$= 2^{10} * 19$ bits
$= 19$ K bits → Tag directory size

Machine ins. MOV R0, [1001]

↓

CPU generates the mem. request

1001	RD

↓

Direct CM

↓

Address format

Tag	LO	WO
1	2	1

1	00	1

Add. decoder of CM

Compare

1

↓

Hit

1	00	B4 → R0
	01	
	10	
	11	B7

Limitations:

① In the direct cache design, each line is capable to hold only one tag (block) at a time, so, number of conflict misses will be increased.

② During the program execution, when the CPU frequently refers the multiple blocks, which are mapped into a same cache line, then, the cache blocks are continuously swapping. So, HT ratio will be falling down. This phenomenon is called as Thrashing, described below.

CM is initially empty

MM Block ref : B0, B4, B0, B4, B0, B4, - - -

B0 – Miss (compulsary miss) : 0 mod 4 = 0
B4 – Miss (comp. miss) : 4 mod 4 = 0
B0 – Miss (conflict miss) : 0 mod 4 = 0
B4 – Miss (conflict miss) : 4 mod 4 = 0

→ To handle the above problem, alternative cache is required, i.e., associative cache.

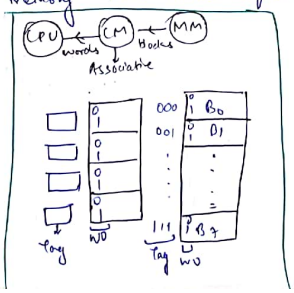
→ Associative cache is designed w/o address called as content-addressable memory. In this, mapping technique, sequence function is used to map the data, so that any main memory block can be mapped into any cache memory line in a sequence.

→ Address binding is:

MM : | Tag (3bit) | W0 (1bit) |

⇓

Associative CM : | Tag (3bit) | W0 (1bit) |



MM Block	Associative MAP	CM Line
B0 [000]	Sequence ⟹	Any line
B7 [111]	Sequence ⟹	Any line
B4 [100]	Sequence ⟹	Any line

→ During the mapping process, complete tag is stored in the cache controller because cache memory does not contain the address. Therefore, tag memory size is equal to:

Tag m/m size = No. of lines in CM * No. of tag bits in the line

⇒ 4 * 3 bits = 12 bits

→ Explicit replacement techniques are used in the associate cache design to replace the data when the cache is full.

Q. Consider a fully associative cache of 64kB, organised into a 32 bytes block. CPU generates 24 bit address to access the data. In the cache controller each tag is comprising of 1 valid bit, 1 modified bit and 1 replacement bit.

ⓐ No. of lines in cache.
ⓑ No. of blocks in MM.
ⓒ Address binding
ⓓ No. of tag bits in the
ⓔ Tag mem. size

ⓐ No. of lines = $\frac{2^{16}}{2^5} = 2^{11}$

ⓑ No. of blocks = $\frac{2^{24}}{2^5} = 2^{19}$

ⓓ Tag bits = 19

ⓔ 22 * 2^{11}

= 44kb bits

ⓒ MM:

Tag	W0
19	5

CM:

Tag	W0
19	5

Machine Inst. MOV r0, [1001]

↓

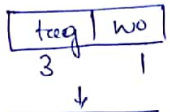
CPU generates the mem. ref request.

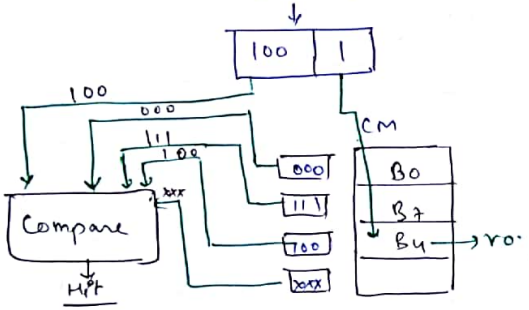
↓

Associative CM

↓

Addr. format





#Limitations:

① → In the associative cache m/w, all the existing tags are compared with a CPU generated tag in parallel, to decide operation as hit or miss. Implementation of a parallel comparison logic circuit becomes very complex and expensive, so, alternative is required, i.e., set associative cache.

#Set-Associative Cache Design:

→ Set-associative cache is used to compromise the disadvantages in the direct and associative cache designs. In this org^n, lines are grouped into a sets to accomodate more than one blocks in the set.

→ $\text{No. of sets } (S) = \frac{N}{P\text{-way}}$

N ⇒ No. of lines in CM

P ⇒ No of lines in set

→ Consider 2-way set-associative cache of 8-byte with a 2 byte blocks.

Before dsgn

8B ⇒ CM

After organised into 4 lines

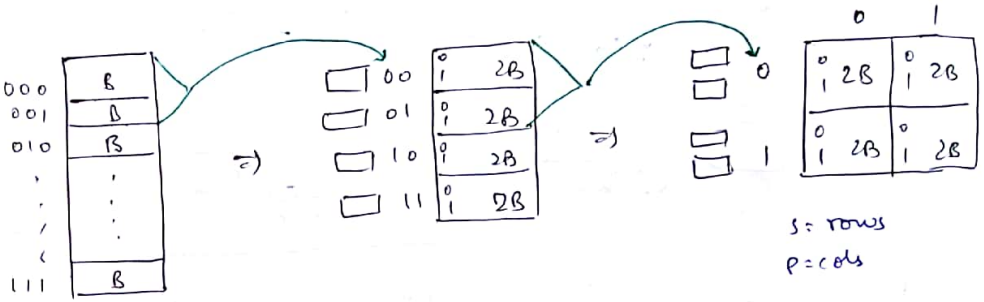
No. of lines (N) = 8/2 = 4

4 * 2B = 8B

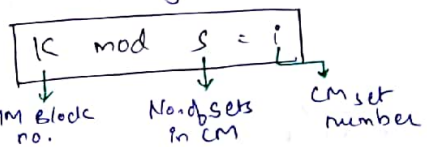
After organised into sets.

No. of sets = 4/2 ⇒ 2

2 * 2 * 2B = 8B



→ In this cache design, mod func.^n is used to map the data.

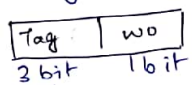


2-way set associative:

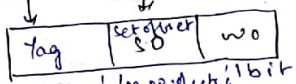
0 mod 2 = 0
1 mod 2 = 1
2 mod 2 = 0
3 mod 2 = 1
4 mod 2 = 0
5 mod 2 = 1
6 mod 2 = 0
7 mod 2 = 1

→ Mapping function shows the relationship b/w the main m/m block and cache m/m set, so, address binding is:

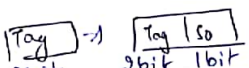
MM:



2 way set-ass^n:

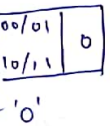


MM



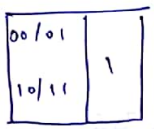
2way set ass^n.

B0 [000]
B2 [010]
B4 [100]
B6 [110]

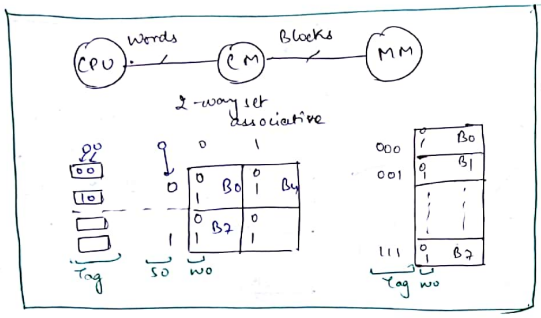


Set '0'

B1 [001]
B3 [011]
B5 [101]
B7 [111]



Set '1'



MM Blocks	Set Assoc. MAP	CM Line
B0 [000]	0 mod 2 = 0	Set '0'
B7 [111]	7 mod 2 = 1	Set '1'
B4 [100]	4 mod 2 = 0	Set '0'

→ During the mapping, some of the tag bits are connected to the address logic of the cache mem. and rest of the bits are stored in the cache controller as a tag. So,

Tag mem. size = No. of sets in CM (S) * No. of lines in the set (P) * No. of tag bits in the line

= S * P * No. of tag bits

= 2 * 2 * 2 bit

= 8 bits.

→ This cache is designed with explicit replacement techniques used to replace the data, when the set is full.

Q. Consider 16-way set associative cache of 64KB organised into a 32 byte block. Main mem. capacity is 32 4GB. In the cache controller, each tag contains 1 update bit and 2 replacement bits.

(a) No. of sets in CM
(b) No. of blocks in MM
(c) Address binding
(d) No. of tag bits in line
(e) Tag mem. size

(a) No. of lines = $\frac{2^{16}}{2^5} = 2^{11}$

No. of sets = $\frac{2^{11}}{2^4} = 2^7$

(b) No. of blocks in MM = $\frac{2^{32}}{2^5} = 2^{27}$

(c) MM

27	5

CM

tag	SO	WO
20	7	5

(d) Tag bits = 20

(e) Tag mem. size = ~~$2^7 * 16 * 2^4$~~
= ~~2^{22} bits~~ $2^7 * 16 * 23$
= $23 * 2^{11}$
= ~~23 Kbits~~
= 46 Kbits

→ Machine inst. : MOV R0, [1001]

↓

CPU generates the mem. req.

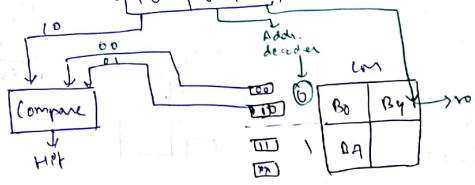
1001	RD

2 way set associative CM

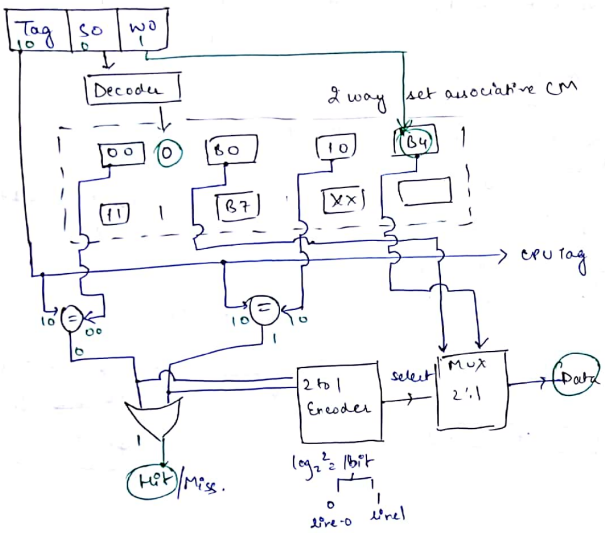
Address format

Tag	SO	WO
2	1	1

10	0	1



Address Format:



NOTE:

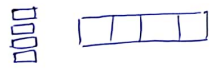
→ When the no. of sets are approached to 1, then, the cache becomes fully associative.

E.g: 8B CM, 2B blocks
4-way set associative

$N = \frac{8}{2} = 4$

$S = \frac{4}{4} = 1$

P ≅ N
S ≅ 1



→ When the no. of sets are approached to N, then, the cache becomes direct.

E.g- 8B CM, 2B Block
1-way set asso.

$N = \frac{8}{2} = 4$

$S = \frac{4}{1} = 4$

P ≅ 1
S ≅ N

Replacement Algorithms:

→ Replacement algorithms are used in the cache design to replace the data when the cache is full. Three types of replacement techniques implemented in the cache design:

① Random Replacement
② FIFO
③ LRU

→ In Random Replacement, replace the blocks in the cache, randomly. So, performance is affected.

→ In FIFO, replace the block which is having the longest timestamp.

→ In LRU, replace the block which is present in the cache, for the longest time without any reference.

Q: Consider 4-block cache, initially empty with the following block references. 4, 5, 7, 12, 4, 5, 13, 4, 5, 7

Identify the hit ratio, using:

ⓐ FIFO
ⓑ LRU
ⓒ Direct mapped cache
ⓓ 2-way set associative with LRU

→ ⓐ CM

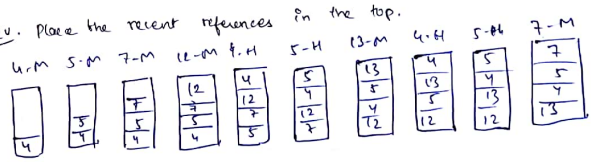
FIFO

4 13
8 4
7 5
12 7

4-M
5-M
7-M
12-M
4-H
5-H
13-M
4-M
5-M
7-M

Hit ratio = $\frac{2}{10} = 0.2$

LRU. Place the recent references in the top.



Hit ratio = $\frac{4}{10} = 0.4$

① Direct Map [K mod N = i]

0	4 ~~12~~
1	~~13~~ 5
2	
3	7

4-M 4 mod 4 = 0
5-M 5 mod 4 = 1
7-M 7 mod 4 = 3
12-M 12 mod 4 = 0
4-M 4 mod 4 = 0
5-H ~~5 mod 4~~
13-M 13 mod 4 = 1
4-Hit ~~4 mod 4~~
5-M 5 mod 4 = 1
7-Hit

Hit ratio = $\frac{3}{10}$ = 0.3

② 2-way set-associative with LRU :

No. of sets = $\frac{4}{2}$ = 2

0	4 ~~12~~	[k mod 2 = i, LRU]
1	5 ~~13~~	7

4-M ; 4 mod 2 = 0
5-M ; 5 mod 2 = 1
7-M ; 7 mod 2 = 1
12-M ; 12 mod 2 = 0
4-Hit
5-Hit
13-M 13 mod 2 = 1
4-Hit
5-Hit
7-Miss 7 mod 2 = 1

Hit ratio = $\frac{4}{10}$ = 0.4

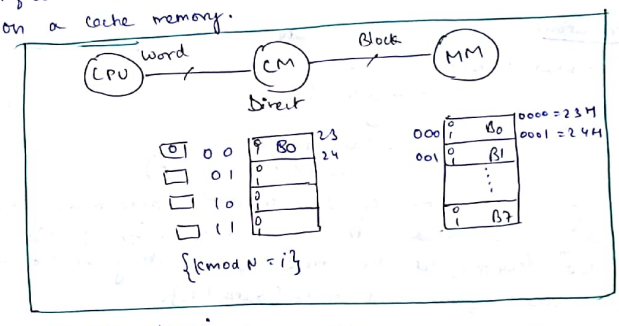
Updating Techniques:

→ In a memory hierarchy structure CPU, always performs the read and write operations only on a cache memory. In this regard, CPU checks the availability of a data block in the cache memory.

→ When the data is present in cache, then the operation becomes read hit or write hit, otherwise operation becomes read miss or write miss.

Due to a miss operation, the respective block will be transferred from main m/m to cache mem. called as read allocate or write allocate.

→ After the allocation, CPU performs read or write operation only on a cache memory.



B0 => K mod N = i
[000] 0 mod 4 = 0 => Line '0'

Prog.

I1 : Mov r0, [0001] ; B0 – 1st byte → r0
I2 : Add r0, #23 ; r0 + 23 → r0
I3 : Mov [0000], r0 ; r0 → B0 – 0th Byte
B0

Exec^n :

I1 = Read Hit ; r0 = 24H
I2 ; r0 + 23 ; r0 = 47H
I3 : write Hit ; CM : [0000] = ~~23~~ 47

MM [0000] = 23
CM [0000] = 47
Cache coherence

→ In the above code, during the execution of a I3 instruction, CPU performs the write operation only on a cache memory. The corresponding block in the main memory is not updated. So, the same address contains different values at different places. This kind of data inconsistency problem in the memory is called as cache coherence.

→ Coherence causes the data loss, i.e., described below:

I4: MOV r0, [1000] (B4) ; B4 - 0th Byte → r0
I5
I6
I7
I8: MOV r0, [1000] (B0) : B0 - 0th Byte → r0

Execution:

I4: Read miss : Read - allocate
k mod N = i
4 mod 4 = 0
⇓
CM: B0 B4
↳ [updated data lost]

→ During the execution of I4 instruction, cache memory data block is replaced with new block. So, updated data is lost. Therefore, at the time of executing I8 instruction old data is transferred to r0 [data loss].

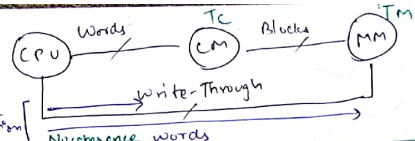
→ To handle the above problem, updating techniques are used in the cache design. These are of two types:

① Write-Through ② Write-Back

Write-through: In this technique, simultaneous write operations are performed in both cache memory and main m/m. So, coherence is not present. Therefore, lower level m/m data always becomes the subset of a higher level. This principle is called as inclusion.

→ To hold the inclusion b/w two levels of the memories, we need to design the lower level memory as a write-through memory.

→



Simultaneous Write Time (Tw) = Max [word updation time in CM, word updation time in MM] → In write through

→ Amount of time required to read one word data from the m/m is called as Read cycle time [Tavgread].

$$T_{avg\,read} = H_1 T_c + (1 - H_1)(T_m + T_c)$$

(H_1: Read Hit, T_c: Read Data, $(1-H_1)$: Read Miss, T_m: Read allocate, T_c: Read Data)

→ Amount of the time required to write one word data into the memory is called as write cycle time (Tavgwrite)

$$T_{avg\,write} = H_w T_w + (1 - H_w)(T_m + T_w)$$

(H_w: Write Hit, T_w: Simultaneous write, $(1-H_w)$: Write Miss, T_m: write allocate, T_w: simultaneous write)

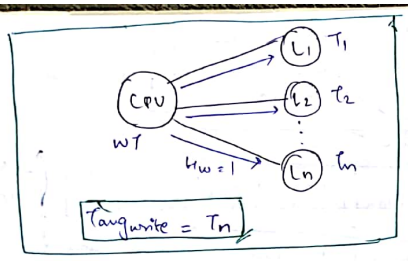
→ Total average access time, when considering both, read and write operations is, calculated as follows:

$$T_{avg\,write\,through} = (f_1 * T_{avg\,read}) + (f_w * T_{avg\,write})$$

(f_1: No. of read opt., f_w: No. of write operations)

$$\eta_{WT} = \frac{1}{T_{avg\,WT}} \text{ words/sec}$$

Note:
→ When the write-through technique is implemented in a simultaneous access mem. organisation, then, hit ratio of write operation always becomes 1, because data block is need not be required in the level 1 memory.



NOTE:

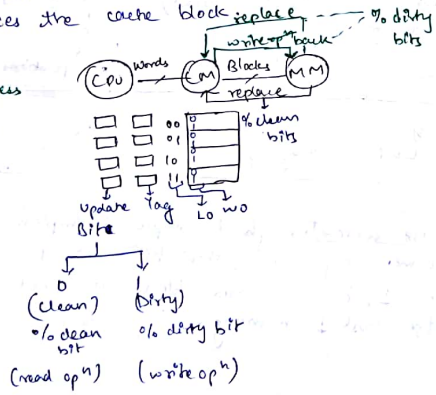
→ When the question contains cache memory word, write-through and $H_w = 1$, then use simultaneous memory organisation. Otherwise, $[H_w \neq 1]$, use hierarchical organisation.

Write-Back: In this technique, CPU performs the write operation only in the cache memory. So, coherence is present. This coherence does not cause the data loss when the CPU reads the status of a cache block before replacement.

→ In a write-back cache design, each line contains one extra bit space called as update bit. This bit is set when the cache block is updated otherwise reset.

→ CPU reads the status of a update bit before replacement. When it is 1, then, write-back the block into main m/m and replace the data in the cache memory, otherwise simply replaces the cache block.

Only applicable for Hierarchical Access



→ Read cycle time is $T_{avg\,read}$:

$$T_{avg\,read} = H_r T_c + (1 - H_r)\left[\%\,\text{dirty bits}\,(T_m + T_m + T_c) + \%\,\text{clean bits}\,(T_m + T_c)\right]$$

($H_r T_c$: Read Hit data; $(1-H_r)$: Read Miss; dirty: Write back, Read allocate data, Read data; clean: Read allocate data, Read data)

→ Write cycle time is:

$$T_{avg\,write} = H_w T_c + (1 - H_w)\left[\%\,\text{dirty bits}\,(T_m + T_m + T_c) + \%\,\text{clean bits}\,(T_m + T_c)\right]$$

($H_w T_c$: Write Hit, Write data; $(1-H_w)$: Write Miss; dirty: Write back, Write allocate data, Write data; clean: Write allocate data, Write data)

→ Average access time of the memory, when considering both, read and write operation is:

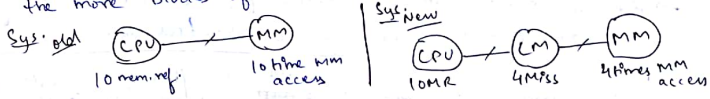
$$T_{avg_{WB}} = (f_r * T_{avg\,read}) + (f_w * T_{avg\,write})$$

Multilevel Cache Design:

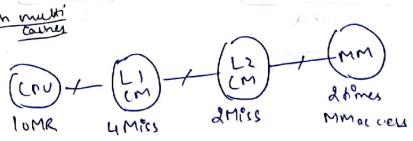
→ Multilevel cache organisation is used in the computer system to reduce the miss penalty.

→ Amount of the time required to access the data from the higher levels due to a miss operation is called as miss penalty.

→ In this kind of organisation, cache memory size is increased logically by replacing the multi levels of caches to hold the more blocks of information.



Sys. with multi caches



→ In the multilevel cache design, L2 cache size is greater than L1 cache, And L1 mem. associativity is greater than level2 (L2) memory.

→ In this organisation, we can calculate two kinds of miss rates;
① Global Miss Rate (GMR) ② Local Miss Rate (LMR)

GMR = No. of Misses in cache / Total no. of CPU generated references

LMR = No. of misses in cache / Total no. of accesses to the cache

$GMR_{L1} = \frac{4}{10} = 0.4$ | $GMR_{L2} = \frac{2}{10} = 0.2$

$LMR_{L1} = \frac{4}{10} = 0.4$ | $LMR_{L2} = \frac{2}{4} = 0.5$

GMR and LMR of L1 is always same.

→ Average access time of a memory system calculated in terms of hit time, miss rate and miss penalty is as follows:

$T_{avg} = Hit\ time_{L1} + (miss\ rate_{L1} * miss\ penalty_{L1})$

$Miss\ penalty_{L1} = Hit\ time_{L2} + (Miss\ rate_{L2}{}_{Local} * Miss\ Penalty_{L2})$

$Miss\ penalty_{L2}$ = MM access time

→ Average memory stalls created per instruction during the execution of a program is calculated as:

Avg. mem. stalls/ins. = (No. of misses$_{L1}$/ins. (Global) * Hit time$_{L2}$) + (No. of misses$_{L2}$/ins. (Global) * MM access time)

Types of Cache Misses:

① Compulsary Miss (Cold start miss/first reference miss):
→ This miss will occur when the very first reference block is a miss. Compulsary misses can be reduced by increasing the block size.

② Capacity Miss:
→ This miss will occur when the capacity is full. These misses can be minimised by increasing the cache size.

③ Conflict Miss (Collision / Interference Miss):
→ This miss will occur when too many blocks are mapped into a same line or same set. These misses can be reduced by doubling the associativity.

27/06/2017:

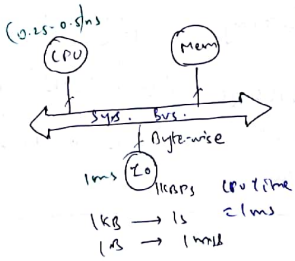
I/O Organisation:

→ CPU is a electronic component and I/O devices are the electromagnetic components. So, ~~there~~ there is a difference existing in terms of operating modes, word formats and data transfer rate.

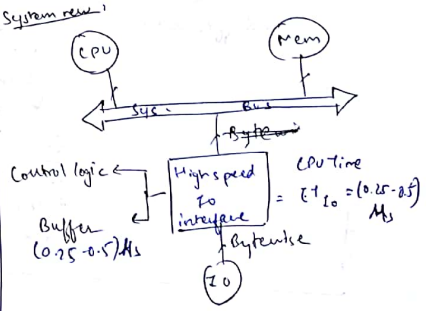
→ To synchronise the I/O speed with CPU, high-speed interface logic is used, named as I/O interface or IO module.

→ I/O interface is responsible to perform the I/O operations. So, I/O devices are connected to a system bus via I/O interface

System old:

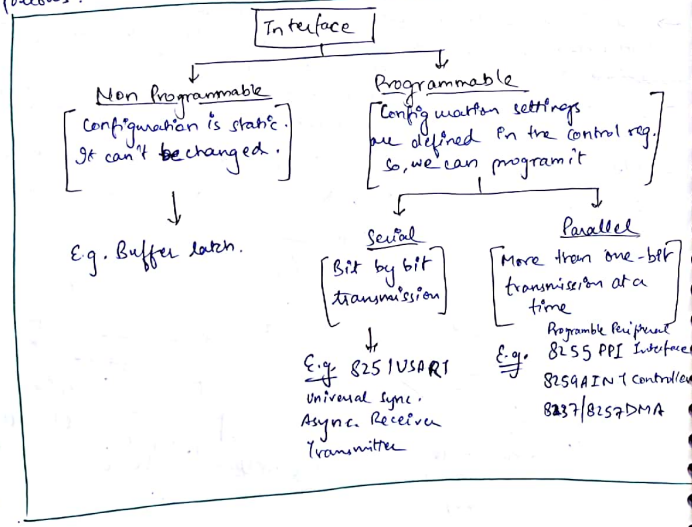


System new:



Accessing Sequence:

→ CPU initialises the I/O interface alongwith a I/O command. Later, busy with other useful tasks.
→ I/O interface control logic interprets the I/O command and enables the operation.
→ Based on the speed of a device, consume the time in completing the operation. Later, transfers the operation status into a interface buffer.
→ When the buffer contains the status, then, I/O interface generate the interrupt signal to CPU and waiting for acknowledgement signal.
→ After receiving the ACK signal, I/O interface transfers the data to CPU with high speed. Therefore, speed gap is synchronised between the CPU and I/O.
→ Different interfaces used in the computer design are as follows:

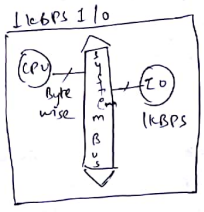


I/O-Modes:

→ IO modes are used to transfer the data between the IO and other components of a computer. (CPU, memory).
→ 3 different IO modes are used in the system design, named as:
① Programmed I/O
② Interrupt driven I/O
③ Direct Memory Access (DMA)

Programmed I/O:

→ In this mode, I/O devices are directly connected to system bus, w/o using a interface. So, CPU takes the responsibility of a I/O operation.
→ In this mode, CPU will be blocked during the I/O operation.
→ Therefore, CPU utilization is inefficient.
→ In this mode, CPU time depends on the speed of the I/O device and the amount of data to be transferred.
→ E.g. Consider a system, where 1KBPS I/O device is interfaced to CPU in a programmed I/O mode.

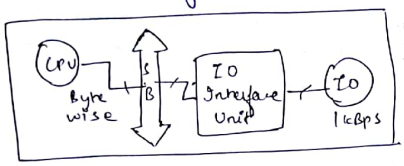


1KB → 1sec
1B → $\frac{1}{1\times10^3}$ ⇒ 10^{-3} s.
= 1ms.

Interrupt Driven I/O:

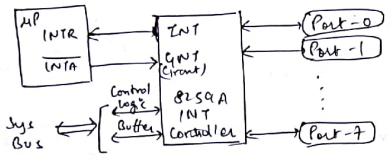
→ In this mode, I/O operations are controlled based on the interrupt signals. So, all the I/O devices are interfaced to CPU via I/O interface chip.
→ In this mode, CPU utilization is efficient because CPU does not undergoes idle state during the I/O operation.
→ In this mode CPU time depends on the latency of the interface, rather than the speed of the I/O device.

E.g. Consider a system, where 1kBps I/O device is interfaced to a CPU using interrupt I/O mode.



$ET_{INT\ IO} = 0.254s$

Implementation:



→ 8259A interface is operating in two modes:
① Single buffer mode
② Cascading mode.

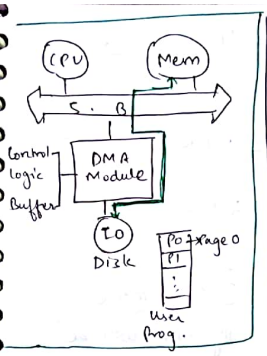
→ When the interface is programmed in a single buffer mode, then maximum 8 I/O devices are managed. When it is programmed in cascading mode, then, maximum 64 I/O devices can be managed using master and slave approach.

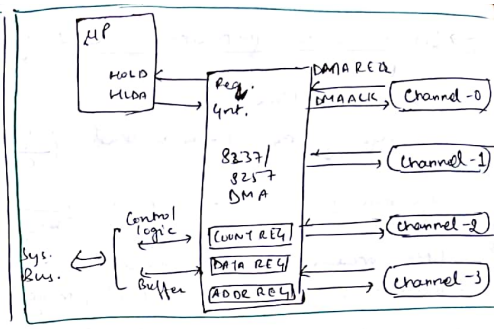
DMA:

→ In this mode, bulk amount of the data will be transfered from I/O to main memory w/o involvement of CPU.

→ When the user program size is greater than the main m/m size, then, virtual memory concept is used to increase the address space. This concept states that use the secondary storage components to store the user program.

→ Secondary storage is a electromagnetic storage ~~st~~ so, it is classified under I/O category. These components are interfaced to CPU via DMA module, therefore, during the program execution bulk amount of data will be transferred from I/O to main memory via DMA w/o involvement of the CPU.





Accessing Sequence:

→ In the DMA operations, CPU initialises the DMA module alongwith a I/O command. Later, busy with other useful tasks.

→ I/O command contains the information about port address, m/m address, control signals and the count value.

→ DMA control logic interprets the command and enables the operation.

→ Based on the speed of a disk (I/O) consumes the time to ~~comp~~ prepare the data. Later, enables the DMAREQ signal.

→ When the DMA module receives this signal from the I/O, then it enables HOLD signal to CPU, to gain the control of the system bus and waiting for HLDA signal.

→ After receiving the HLDA signal, DMA module enables the DMA ACK signal to the I/O.

→ After receiving this signal, I/O device tranfers the data to main memory via DMA until count becomes zero. After the DMA operation, bus connection is re established to CPU.

NOTE:

→ In the DMA operation, CPU is present in 2 states:
① Busy state ② Blocked state (HOLD state)

→ CPU is in busy state until prepare the data. It depends on the I/O speed.

→ CPU is in blocked state until transferring the data, it depends on the m/m speed.

→ Let 'x' is the preparation time; & y is transfer time.

x → Preparation Time

y → Transfer Time

Percentage of time CPU is blocked = $\left(\frac{y}{x+y}\right) * 100$

Percentage of time CPU is busy = $\left(\frac{x}{x+y}\right) * 100$

→ DMA module is programmed in 3 modes:

① Burst Mode ② Cycle stealing mode ③ Block mode

→ In burst mode of DMA, bulk amount of data will be transferred to main mem, based on the HOLD and HLDA signals.

→ In cycle stealing mode of DMA, small amount of data will be transferred to main memory by stealing the control of a system bus from the CPU hand, when it is in idle state.

→ In Hock mode of DMA, data will be transferred to main m/m in the form of blocks based on the hold and HLDA signals.

Q. Consider a 32-bit hypothetical processor which is interfaced to a 1MBPs I/O device in a programmed I/O mode. Take a transmission is in the form of word-wise. I/O interface interrupt overhead is 1μs. How much performance will be gained by operating the device in a interrupt mode over the programmed I/O mode.

Programmed I/O: (Depends on IO speed)

1MBPS → 1s

1B → 10^{-6} s = 1μs

For 1word = 4μs (32-bit)

Interrupt driven: (Depends on IO Interface latency)

1μs

gain = 4

Q. Consider 1MBPs I/O device interfaced to a CPU in a cycle stealing mode of DMA. Whenever, 8 bytes data is available in the buffer then, it is transferred to CPU. Mem cycle time is 2μs. How much %age of CPU time is consumed in the DMA operation.

(x) Preparation time (Depends on I/O) = 1MBPS x 8 = 8μs

(y) Transfer time = Mm cycle time = 2μs

% time CPU blocked = $\frac{2}{10} \times 100 = 20\%$

Q. Consider 10MBPs device interfaced to a 32 bit CPU in a cycle stealing mode whenever 4-word data is available in the buffer, then it is transferred to main mem. M/c cycle time is 0.5μs. How much %age of time CPU blocked?

x = 1.6μs

y = 0.5 x 4 = 2μs

$\frac{2}{3.6} \times 100$

= 55.56%

M/c cycle time = 0.5μs

(No. of cycles reqd for one mem. opn)

1 word → 1 m/c cycle

4 words → 4 m/c cycles.

Q. Consider 10MBPs I/O device which is interfaced to a 1MHz clock freq. CPU in a cycle stealing mode of DMA. Whenever, 32 byte data block is available in the buffer, then, it is transferred to MM. DMA initialisation and transfer takes 30 and 60 cycles respectively. How much %age of CPU time is blocked in the DMA operation.

10MBPS

1B → 0.1μs

x = 32 x 0.1 = 3.2μs

1MHz → $\frac{1}{10^6}$

$= 10^{-6}$ s → 1μs m/c cycle

y = 30 cycles + 60 cycles

= 90 cycles = 90μs

CPU blocked = $\frac{90}{93.2} \times 100$

= 96.57%

Q. A 32-bit hypothetical processor is interfaced with a DMA module to transfer the data. DMA contains 8-bit count register and 32 bit data register. How many bytes of data is transmitted to main memory in a DMA cycle.

1 word = 4 bytes

DMA cycles is controlled by the count register.

Count register is 8 bit long, So it maintains 256 counts.

In each count 1 word data is transferred to Mem. So, total data transmission = 256 words
= 256 × 4 = 1KB

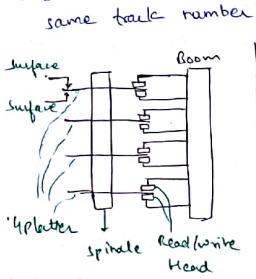
SECONDARY STORAGE COMPONENTS:

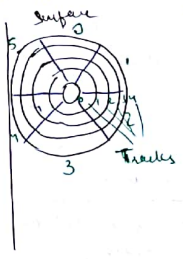
→ Classification of secondary storage components is as follows

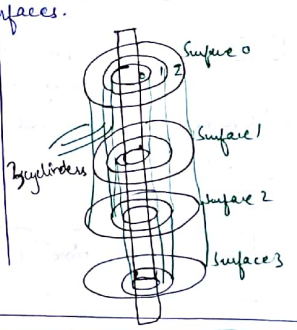
- Secondary Storage
  - → Direct Access Storage Devices
    - → Magnetic Disks
      - → Floppy Disks [Low Capacity, Slow-Cheap]
      - → Hard Disks [High capacity, low cost/bit]
    - → Optical Disks
      - → CDROM [Compact Disc-Read only mem]
  - → Serial Access Storage Devices
    - → Magnetic Tape [High Speed Sequential Access

Hard Disk Design:

→ Hard Disk is a direct access electromagnetic storage component. It contains a bunch of magnetic coded platters used to store the data.
→ Each platter contains 2 surfaces.
→ Every surface has its read-write head used to access the data.
→ Surface contains set of concentric circles called as tracks.
→ Track is divided into a set of sectors. Sector holds the data.
→ In the hard disk design, surfaces, tracks and sectors are the addressable units.
→ In the hard disk, cylinder is formed by connecting the same track number in all the surfaces.







① No. of cylinders in the disk = No. of tracks in the surface
② Track capacity = No. of sectors/track × Bytes/sectors
③ Cylinder capacity = No. of surfaces in disk × track capacity
④ Disk Capacity = No. of cylinders in the disk × cylinder capacity

Alternate:

Disk size = No. of surfaces in the disk × No. of track/surface × No of sectors/track × Bytes/sect

→ Different adjustments are required in the disk operation:

① Seek Time: Amount of the time required to move the head point from the initial position to ~~desirable~~ desired track in the surface.

② Rotational Latency: Amount of time required to adjust the head point at the beginning of a desired sector in the track.

(a) Best case: In this view, rotational latency is zero because head point is adjusted during the seek time itself.

(b) Worst case: In this view, one complete rotation is required to adjust the head point.

(c) Average case: In this view, half of a revolution is required to adjust the head point.

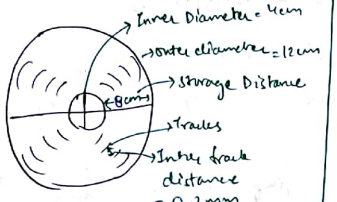
③ Transfer Time: Amount of time required to access the data from the disk.

④ Overhead: Additional delays in the disk operation.

⑤ ~~Total t~~ → Total time required to access the data from the hard disk is calculated as:

$$T_{avg} = \text{seek time} + \text{average rotational latency} + \text{transfer time} + \text{overhead}$$

→ In the hard disk design, storage distance is restricted by the inner and outer diameters.



Storage distance = outer diameter − inner diameter

=

No. of tracks = $\frac{\text{Storage dist.}}{\text{Inner track dist.}}$

$= \frac{4\text{cm}}{0.2} = \frac{40}{0.2} = 200$ tracks

→ When the sector address is expressed in a 3-dimensional format, then sector no. in the hard disk is calculated using the following formula:

Sector number = (i, j, k)

i = Cylinder no.
j = Surface no.
k = Sector no.

$$\text{Sector no.} = k + s(j + i*t)$$

s = No. of sector / track
t = # track / cylinder
(No. of surfaces in the disk)

① Consider a hard disk with a rotational rate of 9800 rpm. Average seek time of the disk is 8.4ms. Disk contains 16 platters. Surface contains 128 tracks. Track contains 256 sectors. Sector holds 512 bytes of data. In the application program record size is 4kB. Calculate the following.

(a) Disk capacity = ~~2^{29} Bytes~~ ~~$= 2^{30}$ Bytes~~ 2^{29} Bytes = 512 MB

(b) Time required to access the record.

(c) Amount of time required to access 100 random sectors data.

(d) Calculate the disk transfer rate.

(b) Seek time = 8.4ms

Rotational latency: Depends on rotational speed

9800 revs → 60s

1 rev → $\frac{60}{9800} = 0.006125\text{s} = 6.12\text{ms}$

Avg rotational latency = $\frac{1}{2}$ × revolution time = $\frac{1}{2} \times 6.12$

= 3.06 ms

Transfer time: Depends on rotational speed

1 revolution time = 1 track data (256 sectors)

? → 1 record (8 sectors)

$= \frac{6.12}{256} \times 8 = 0.191\text{ms}$

T_{avg} = seek time + avg rot. latency + transfer time + overhead

= 8.4 + 3.06 + 0.191 + 0 = 11.651 ms

(c) Random sector accessing requires adjustment for every sector.

1 revolution — 1 track (256 sectors)

time — 1 sector

$= \frac{6.12 \times 1}{256} = 0.0239$ ms

Tavg. = [8.4 + 3.06 + 0.0239 + 0] for 1 sector

for 100 sectors:

$= 100 \times [8.4 + 3.06 + 0.0239]$

$= 1148.39$ ms

(d) Transfer rate of disk:

1 revolution time — 1 track (256 × 512 Bytes data)

1 s — ? Bytes

Data access transfer rate / surface $\leftarrow \frac{256 \times 512}{6.12 \times 10^{-3}} = 21416993.464$

Disk rate = # Surfaces × Data transfer rate / surface

$= 685,343790.849$

$= 653.59$ MB

28/06/2017

Module - III . #fined Point vs. Floating Point Data formats:

Multiplication:

→ Multiplication process is easy controlled by the multiplier. In the manual multiplication, 2 actions are performed:

① Generation of a partial products.

② Summation of a partial products.

→ Partial products are generated based on the multiplier bits, i.e., when the multiplier bit is 1, the partial product is multiplicand, otherwise partial product is zero (0).

→ After the generation of a partial products, summation is provided to produce the final product.

Eg. Multiplicand 1111 × multiplier 1111

```
 1111
 1111
 1111
 1111
--------
11100001
```

Consider the following multiplication and identify the values of w, y and z variables.

$(10w1z)_2 \times (15)_{10} = (y01011001)_2$

$(1111)_{10}$

z = 1
w = 1
y = 1

```
   10 0 1
 1 0 1 0 1 0w1z
 1   1 0w1z
   1 0w1z0
   1 0w1z0
-----------
 1 0 1 0 1 1 0 0 1
```

= y01011001

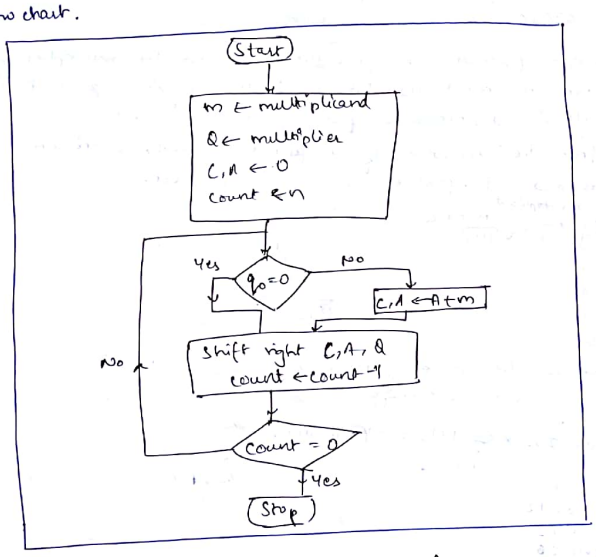
NOTE:

→ Disadvantage in the manual multiplication is, requires more no. of registers to hold the partial products and summation becomes complex in the hardware. So, optimization is required, i.e., accumulated addition.

Unsigned Multiplication:

→ When the multiplication takes place between 2 n-bit numbers, then, result will be expressed in a 2-n-bit size, i.e., n-bit * n-bit = 2n-bit.

→ Optimized multiplication process is described in the following flow chart.



Eg: (13) 1011 * 1101

M = 1011 ; count = 4

C	A	Q	
0	0000	1101 (q_0)	; initial
0 → 1	1011 → 1011	1101 → 1110 (last)	; A ← A+M, shift, count = 3 (0000 + 1011 = 1011)
0 → 0	0101 → 0101	1110 → 1111 (last)	; shift, count = 2
0	0010	1111	; shift, count = 1
0 → 0	1011 → 0110	1111 → 1111 (last)	; C, A ← A+M, shift, count = 0 (0010 + 1011 = 1101)

Product is in AQ register, i.e., 1000 1111 [143]

Multiplier Bit	Operation
0	Shift
1	Add & Shift

Signed Multiplication [Booth's / Bit Pair Multiplication]:

→ Signed data multiplication is controlled by the bit pairs. Bit pair is formed based on the multiplier. In this process, one assumption is required to get the initial bit pair, i.e., $q_{-1} = 0$.

E.g. Multiplier: $q_3 q_2 q_1 q_0$

Bit Pairs

Multiplier	Pair with
q_0	q_{-1}
q_1	q_0
q_2	q_1
q_3	q_2

→ Multiplication is controlled based on the bit pair status, i.e.,

Bit Pair	Operation	Reading
00	ASR	0
01	+A & ASR	+1
10	-A & ASR	-1
11	ASR	0

ASR → Arithmetic Shift Right

Q. Consider the following Booth's Multiplication.

Multiplicand : 0110 0001 1000

Multiplier : 1010 0101 1010

(a) How many arithmetic operations are required in the multiplication.

(b) What is the recoded multiplier.

Multiplier (q_0)	Pair with (q_{-1})	
0	0 → 0	LSB.
1	0 → −1	
0	1 → +1	
1	0 → −1	
1	1 → 0	
0	1 → +1	
1	0 → −1	
0	1 → +1	
0	0 → 0	
1	0 → −1	
0	1 → +1	
1	0 → −1	MSB

ⓐ → 9

ⓑ →

−1+1−1 0+1−1+10

−1+1−10

Q. Consider the following 8-bit multiplication: what is the recoded multiplier.

(+112) * (−121)

0 111 0000 * 1000 0111

0 1111001

1 0000 110

+1

1000 0111

Multiplier	Pair with	
1	0 → −1	(LSB)
1	1 → 0	
1	1 → 0	
1	1 → +1	
0	0 → 0	
0	0 → 0	
0	0 → 0	
1	1 → −1	(msb)

Recoded Multiplier.

= ~~1~~ −1000+100−1

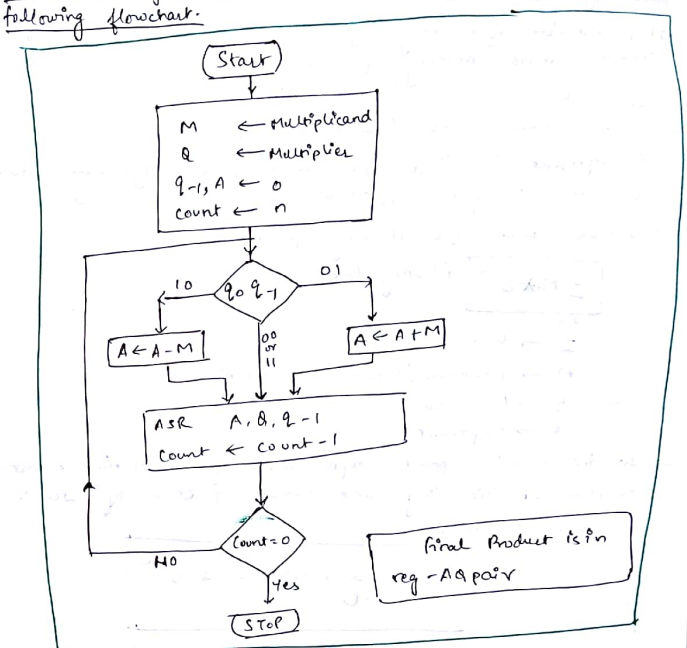
Q. Consider the following bit pair combination used in the Booth's multiplication.

Bit Pair: Fill the data.

DS j

A x
A C
B D
E E
F F
G G
H H
I I
J J
K K

Q# Optimized signed multiplication is described in the following flowchart.



Eg (−7) × (−3) = +21

1001 1101

Bit Pairs:

Multiplier	Pair with	
q_0 1	q_{-1} 0	→ −1
0	1	→ +1
1	0	→ −1
1	1	→ 0

Recoding Multiplier = 0 −1 +1 −1

m = 1001 count = 4

A	Q	q_0	q_{-1}	Initial	
0000	1101	1	0		2222 / 7777
0111	1101	1	0	last; A ← A−M	0000 / 1001 / 0111
0011	1110	1		ASR, count = 3	
1100	1110	1	last	A ← A+M	0011 / 1001 / 1100
1110	0111	0		ASR, count = 2	
1010	0111	0	last	A ← A−M	1110 / 1001 / 0101
0010	1011	1	last	ASR, count = 1	
0001	0101			; ASR, count = 0	

(01) → Product = 0 0 0 1 0 1 0 1 = +21

#Floating Point Data format: *

→ Representation of a very large and very small fraction of the data consumes more memory space.

E.g. +998000000000 (very large) (≅ ∞)
+0.000000000978 (very small) (≅ 0)

→ To represent the large and small fraction of a data within a less amount of memory, floating point rep. is used.

→ general form of a floating point format is:

$\pm M * B^{\pm e}$ Floating Point Data.

± → sign
M → Mantissa / significant (fraction)
B → Base / radix
e → exponent

→ In the floating point representation, normalisation format is used to represent the large and small fraction of data in a common format, i.e.,

$\pm$ valid digit . fraction $* B^{\pm e}$

→ When the base of the data is greater than 2, then the valid digit varies, so, extra space is required to store the valid digit.

E.g.

Base	Valid Digit
B = 3	{1, 2}
B = 4	{1, 2, 3}
B = 5	{1, 2, 3, 4}

→ To handle the above overhead, base of the data is always restricted to 2, so, valid digit becomes 1, as implicit. Therefore no need to store this digit in the memory.

→ general form of normalisation is:

Implicit. ← fraction

$\pm$ (1) . bbb . . . $*$ (2) $^{\pm e}$ → b = 0/1, B = 2 (Default)

→ Mantissa alignment process is used to adjust the decimal point in order to get the data in a normal form. In this process, right alignment increments the exponent and left alignment decrements the exponent.

E.g. +110110101
↓
(1.bbb . . .)
↓
Mantissa align right 8 times
↓
$1.10110101 * 2^{+8}$

E.g 0.000000101011101
↓
(1.bbb . . .)
↓
Mantissa align left 7 times
↓
$+1.0101101 * 2^{-7}$

→ When the data is in a normal form, then it is stored into a memory using the floating point format. Let us consider a 20-bit hypothetical format used to represent the data, i.e.,

Sign (S)	Biased Exponent (BE)	Mantissa (M)
1-bit	6-bit	13-bit

→ Floating point data is stored in the memory with 3 fields of information:

① Sign field: This field is used to represent the sign of a data, i.e., S → 0 (+ve), 1 (−ve)] Mantissa sign

② Biased Exponent field (BE): This field is used to store the exponent. In the memory, exponent is always stored in a biased exponent format, i.e., BE = Actual Exponent + Bias (AE)

→ Bias is a maximum possible exponent, depends on the format, BE field, ie,,

BE field size	Range of exponent	Bias
6-bit	$(-2^{(6-1)})$ to $+(2^{(6-1)}-1)$ -32 to $+31$	$+31$
9-bit	-256 to 255	$+255$
n-bit	$-2^{(n-1)}$ to $+2^{n-1}-1$	$(+2^{n-1}-1)$

→ This concept is used to preserve the sign of an exponent, i.e., when (BE > Bias) then AE is +ve.
when (BE < Bias) then AE is -ve.

NOTE:
→ To retrieve the exponent from the memory, we need to subtract the bias from the biased exponent, i.e., AE = BE - bias.

E.g. Format [S | BE | M] (1, 6, 13)

Bias = $+(2^{n-1}-1)$
$= +31$

Data = $-1.10101 \times 2^{+8}$

AE = +8

BE = 8 + 31 = 39

$001000 + 011111 = 100111$

BE : 100111

Retrieve (Read)

AE : BE - Bias

BE : 100111
Bias : 011111
AE : 001000 : 8

NOTE:
→ When the format is designed with an Excess Bias form, then bias becomes the center of a exponent range. In this format, actual exponent is retrieved from the memory by taking the compliment of a MSB of Biased exponent rather than the subtraction operation.

E.g: Format: [S | BE | M]
1 bit 6bit 13bit
↳ Excess-Bias

Data: $-1.10101 \times 2^{+8}$

Store:
BE = AE + Bias
$= \left(\frac{2^n}{2}\right) = \frac{2^6}{2}$
$= 32$

AE : 001000
Bias : 100000
BE : 101000

Retrieve:
AE = BE - Bias
BE : 101000
Bias : 100000
AE : 001000
[+8]

Alternate:
AE = Complement of MSB of BE
BE = 101000
↓ Complement of MSB
AE : 001000

(2) Mantissa field: → This field is used to store the fraction part of a data.

→ Data is always stored in the memory in a normalised mantissa format, i.e., [1.bbb....] b: 0/1.
Implicit | Fraction

→ In the above structure only the fraction part is stored into a mantissa field of the memory because valid digit becomes implicit.

NOTE:
→ To retrieve the data from the memory, we need to align the mantissa to right direction upto 1 time to get the implicit digit as a valid fraction. This adjustment is called as Denormalisation (Subnormal).

Fetch the data from the memory
$(\pm .bbb \ldots \times 2^{\pm e})$
↓
mantissa align to right upto 1 time to get the implicit digit
↓
$\pm 0.1bbb \ldots \times 2^{(\pm e)+1}$
$\pm 0.1bbb \ldots \times 2^{\pm e'}$ e' → adjusted exponent

Q. Consider the following hypothetical format used to represent the data.

Format : | S | BE | M |
1 bit 7 bit 12 bit

Data : $-7.25 \times 2^{+11}$

What is its hexadecimal equivalent when the data is stored in the memory:

(a) w/o normalisation

(1) Sign = negative = 1

(2) BE = AE + BIAS $\times 2^{(7-1)} - 1$
= +63

AE = 0001011
Bias = 0111111
BE = 1001010

(3) Mantissa
$(7.25)_{10} = (x)_2$
111 . 01
x = 111.01

(b) with normalisation,

(a) Data : $111.01 \times 2^{1001010}$
↓
w/o normal form
↓
Store in mem.
↓
format Reqd.
↓

S	BE	M
1	1001010	0100000000
	C A	4 0 0

(b) Data with normal form :
$-111.01 \times 2^{1001010}$
↓
$(1.bbb - y)$
↓
Mantissa aling to right 2 times
↓
$1.1101 \times 2^{1001010+2}$
$1.1101 \times 2^{1001100}$
↓

1	1001100	1101	0000000000
1 bit	7 bit	4	12 bit
C	C	D	0 0

Q. Consider the following hypothetical format used to represent the data.

Format : | S | BE | M |
1 bit 7 bit 12 bit
↳ Excess Bias

Data : 0X 3CE 00

What is the actual data associated with the above hexadecimal equivalent in the memory. Report the decimal value.

3 C E 00
0011 1100, 1110 0000 0000
S BE M

0 1111100 BE
1111100 AE
2's complement
00000 11
+ 1
00000100 (-4)

(3) Mantissa :
M : 11100
M : 111

Data : $+0.111 \times 2^{-4}$

$\Rightarrow +0.1111 \times 2^{-4+1} = +0.1111 \times 2^{-3}$ → To report the decimal value, exponent will be making '0' by aligning the mantissa to right upto 3 times.

$= 0.0001111$ → forreport

$= 1 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6} + 1 \times 2^{-7}$

$= \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128}$

$= 0.1171875$

RANGE OF FLO

RANGE OF FLOATING POINT DATA :

→ Floating point data range is always dependent on the floating point format. Let us consider, 20-bit hypothetical format to analyse the ~~data~~ data, i.e.,

S	BE	M
1 bit	6 bit	13 bit

① Sign(S) → 0 (+ve)
→ 1 (-ve)

② Exponent Range : Depends on BE field

BE = AE + Bias.
Bias = $+(2^{n-1}-1)$
= +31

ⓐ Min BE = All 0's in BE
= 000000 = 0
AE = BE - Bias = 0 - (+31)
= -31

ⓑ Max BE = All 1's in BE
= 111111 = 63
AE = BE - Bias = 63 - (+31)
= +32

Exponent Range = {-31 to +32}

③ Mantissa Range : Depends on Normal Mantissa, i.e., 1.M

ⓐ Min M = All 0s in M
= 000... (13) 0s
= 0
Normal M = 1.M = 1.0
= 1

ⓑ Max M = All 1's in M
= 1111111111111
∴ Normal M = 1.M
= 1.1111111111111

→ Mantissa align to left upto 13 times to convert the fraction as a integer

↓

1111 1111 1111 11 × 2^{-13}

↓

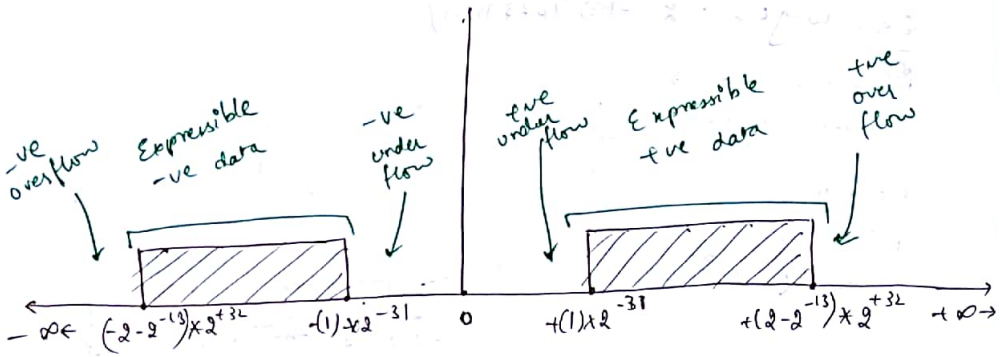
$(2^{14}-1) \times 2^{-13}$

↓

$(2-2^{-13})$ → $[2-2^{-n}]$ (M)

Mantissa Range = [1 to $2-2^{-13}$]

④ Range of a 20-bit floating point data : $[\pm M * B^{\pm e}]$



IEEE formats:

① Single Precision (32 bit)

S	BE	M
1	8 bit	23 bit

① Signs → 0 (+ve)
→ 1 (-ve)

② Exponent range : Bias = $2^7 - 1$
= 127

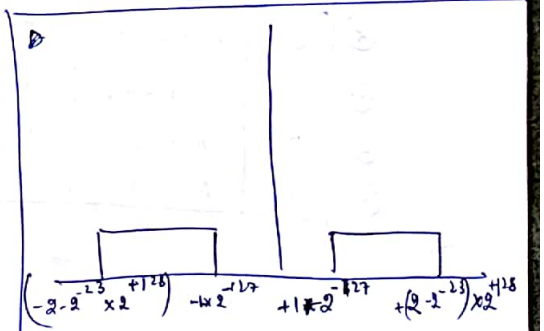
Min. BE = 0
AE = 0 - 127 = -127
Max BE = 255
AE = 255 - 127 = 128
E.R → {-127 to +128}

③ Mantissa Range:
Min M = 1.0000
= 1
ⓑ Max M = All 1's
= 1111...
Normal = 1.1111...

② Double Precision (64 bit)

S	BE	M
1 bit	11 bit	52 bit

11111111111 × $+2^{-23}$
$(2^{24}-1) \times 2^{-23}$
↓
$2-2^{-23}$
Mantissa Range = [1 to $2-2^{-23}$]



Double Precision : 1 11 52

Signum → 0 / 1

Expo. Range :- $2^{10} - 1 \Rightarrow 1023$ (Bias)

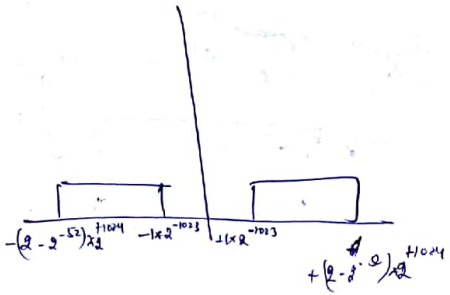
Min AE = −1023

Max AE = +1024

Mantissa :

Min. = 1.0

Max = $2 - 2^{-52}$



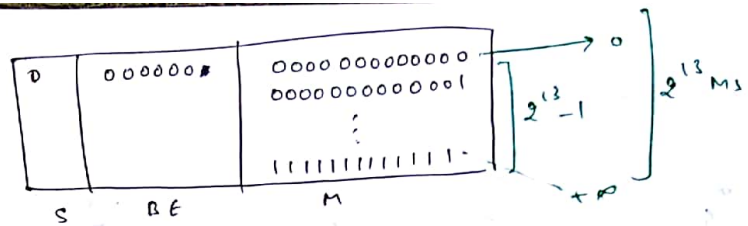
(5) # Special values in the floating point format is :

	S	BE	M	Value
@	0/1	All 0's	All 0's	0
$	0	All 1's	All 0's	$+\infty$
$	1	All 1's	All 0's	$-\infty$
$	0/1	All 1's	$\neq 0$	NaN

(6) Representable Data values :

	S	BE	M	Value
✱	0	$0 < e < 63$	f	$+(1.f) \times 2^{e-31}$
✱	1	$0 < e < 63$	f	$-(1.f) \times 2^{e-31}$
@	0	All 0's	$\neq 0$ (f)	$+(1.f) \times 2^{0-31}$
@	1	All 1's	$\neq 0$ (f)	$-(1.f) \times 2^{0-31}$

E.g.



0 | 0000000 | 00000000000000

↓ ↓ ↓

\+ 0 0

AE = BE − Bias → Normal Ms (1.M)

= 0 − (+31) ↓

= −31 1.0

Data : $+(1.0) * 2^{-31}$

↓

De-Normal Data

(0.1M)

↓

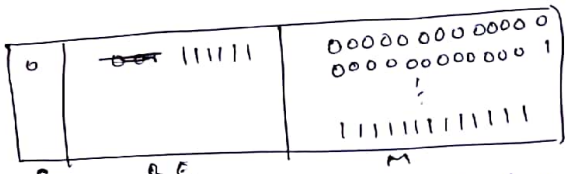
Mantissa align to right upto 1 time

↓

$+(0.1) * 2^{-31+1}$

$+(0.1) * 2^{-30}$ [0]

E.g.



0 | 111111 | 000 ... (13) 0's

↓ ↓

\+ 63 0

AE = BE − Bias → Normal (1.M)

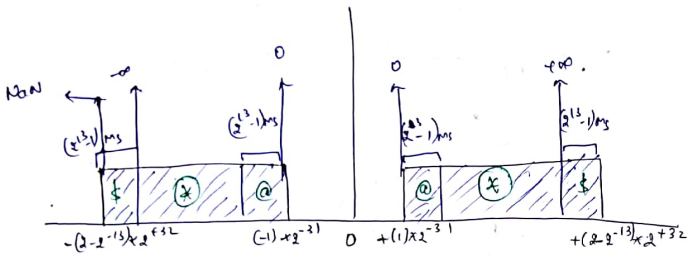
= +32 = 1.0

Data : $+(1.0) \times 2^{+32}$ → De normal Data → Mantissa align to right upto 1 time

(0.1M)

$= +(0.1) \times 2^{+32+1}$ → Max exponent is +32

$= +(0.1) \times 2^{+33}$ (+∞) 33 is > 32.



@ indicates the min^m exponent data with all possible M's.

$ indicates the max exponent " " " " "

* " " data which is existed b/w the min & max exponents with all possible M's.

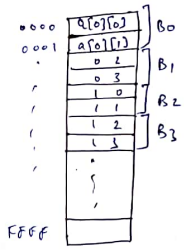
Spatial locality in the array accessing from the memory.

→ Array is a ordered set of the homogeneous data elements.

→ Array elements are always stored in the memory in a sequential cells row by row (row-wise).

E.g. $\begin{bmatrix} a[0][0] & a[0][1] & a[0][2] & a[0][3] \\ a[1][0] & . & . & . \\ a[2][0] & - & . & - \\ a[3][0] & - & - & - \end{bmatrix}_{4\times4}$

Mem storage (row-wise)



Assume Block size = 2 B

→ Array elements are accessed from the memory in 2 ways:

① Row major indexing:

a[0][0] - Miss ; a[0][0], a[0][1] → CM

a[0][1] - Hit

a[0][2] - Miss

a[0][3] - Hit

② Column major Indexing:

a[0][0] → Miss

a[1][0] → Miss

a[2][0] → Miss

NOTE:

Storage Seq.	Access Seq.	Locality of reference
Row	Row	Yes
Row	Col.	No
Col.	Col.	Yes
Col.	Row	No

→ When the main mem. block references are sequential, then arrangement in the cache is also sequential, so, mapping function is not required to map the data.

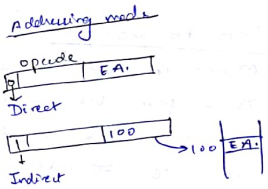
Eg.

0	B_0 B_8	B_4 B_{12}
1	B_1 B_9	B_5 B_{13}
2	B_2 B_{10}	B_6 B_{14}
3	B_3 B_{11}	B_7 B_{15}

Instruction Cycle - Pipelining:

- Fetch → M/m to register (T_0)
- Decode
- Compute Effective address
  - Direct → nothing
  - Indirect → memory
- Execute (T_7)

Addressing mode



Fetch Phase:

→ Getting instruction from main memory.

T_0: $AR \leftarrow PC$

T_1: $IR \leftarrow M[AR]$, $PC \leftarrow PC+1$

Decode Phase:

→ T_2: $D_7 \ldots D_1 D_0 \leftarrow$ Decode $IR(14,13,12)$; $AR \leftarrow IR(0 \ldots 11)$

$I \leftarrow IR(15)$

Compute Effective address:

→ Not required for direct addressing.

→ T_3: for indirect addressing.

$D_7 . I . T_3$: $AR \leftarrow M[AR]$

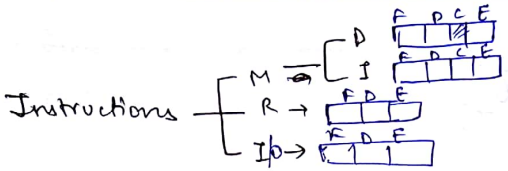
Execute

$D_7 I T_3$ → Execute I/o ins. → F D E | T_0 T_1 | T_2 | T_3

$D_7 \bar{I} T_3$: Execute Reg. → F D E | T_0 T_1 | T_2 | T_3

$\bar{D}_7 . \bar{I} . T_3$: Nothing

F D C E | T_0 T_1 | T_2 | | T_4 | T_5



* Data forwarding:

→ means if cpu writes to a memory location and subsequently reads from the same memory location, the second instruction can fetch the value directly from the register used to do the write rather than waiting for the memory.

→ So, this increases the performance.