Computer Organization & Architecture

Computer Architecture : Deals with conceptual design & fundamental operational structure

Computer Organization: Deals with physical devices and their interconnection with a perspective of improving the performance.

Computer Architecture	Computer Organization
→ CPU Design → Instructions → Addressing modes → Data formats	→ I/O organization → Memory organization → pipelining

System Bus:

It has 3 things:

(i) Address bus (unidirectional)

(ii) Data bus (Bidirectional)

(iii) Control bus (every line is individually unidirectional)

Memory may also send controls to CPU.

Those signals are wait, ready.

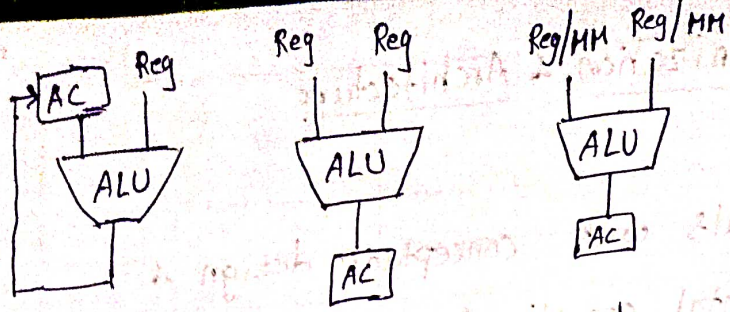
CPU Registers:

(i) General Purpose Registers (store any general content)

(ii) Special Purpose registers (stores special values)

Special Purpose registers

(i) **Accumulator**: It is used to store result of ALU operations. Sometime it may even store one i/p to ALU.



Accumulator based architecture

Register based architecture

complex architecture

→ If inputs to ALU are given from stack then it is called stack based architecture.

32-bit Architecture :

→ size of each i/p given to ALU is 32 bit.

→ word size = 32 bits

64-bit Architecture

→ size of each i/p given to ALU is 64 bit.

→ word size = 64 bits.

(iii) Program Counter :

→ It stores the address of next instruction to be executed.

(iv) Instruction Register :

→ It used to store current instruction which CPU is executing (At the time of instruction fetch, the instruction is directly fetched into IR)

(v) Stack Pointer :

→ It used to store the address of top of the stack.

(vi) Flag Register / Program Status Word (PSW) / Status Register

Each bit of flage register has a purpose

(or)

Flag register stores status of ALU result

Z (zero flag) ← 1 (ALU's result = 0)
0 (ALU's result = 1)

S (sign flag) ← 0 (+ve)
1 (-ve)

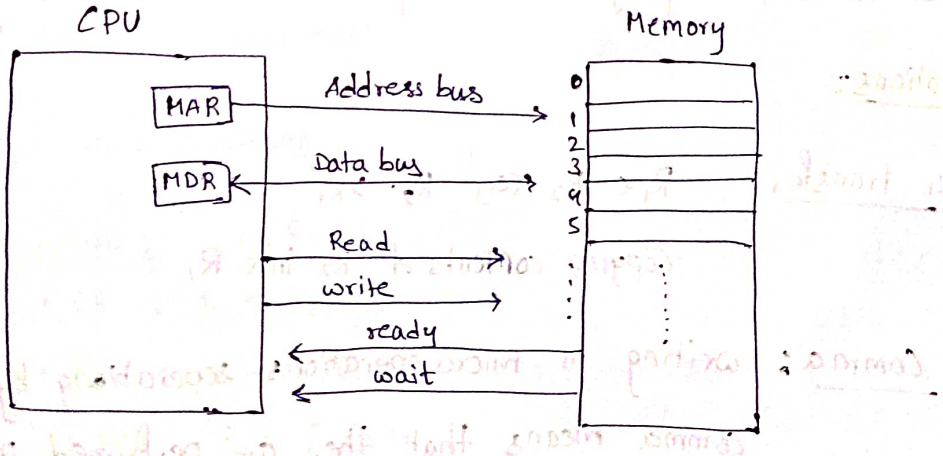
→ In some CPUs PSW is same as flag register
In some CPUs PSW is flag register + accumulator

(vi) Memory Address Register:

It is used to send address to memory.

(vii) Memory Data Register:

It is used to send data to memory (memory write) or to receive data from memory (memory read).



Note:

→ size of PC = size of MAR = Physical memory address length.
→ size of MBR = size of acc = length of each memory location.

If virtual memory is used size of PC ≥ $\log_2$ (VAS)
size of MA = PA

Memory Read:

(i) CPU sends address to memory using address bus.

(ii) CPU enables read control signal.

(iii) Memory performs read operation & sends data to CPU using data bus.

Memory write:

(i) CPU sends address to memory using address bus.

(ii) CPU sends data using data bus. (from MBR)

(iii) CPU sends enabled write signal.

(iv) Memory performs write operation.

Micro Operation:

→ The smallest indivisible operation performed by CPU is called micro-operation. Every μ-op is performed in 1 cycle.

→ These are the operations performed on values stored in registers.

→ Register Transfer Language is a symbolic notation of μ-ops.

RTL Notations:

Register transfer: $R_1 \leftarrow R_2$ (or) $R_2 \rightarrow R_1$

Copying contents of R_2 into R_1

Comma: writing to micro operations seperating by comma means that they are performed parallely.

Eg: $R_1 \leftarrow R_2 + R_3$, $PC \leftarrow PC + 1$

Memory transfer

- Memory Read: $R_1 \leftarrow M[address]$
  - two ways
    - $R_1 \leftarrow M[2000]$ → so Directly specify address
    - $R_1 \leftarrow M[MAR]$

Memory write:

$M[Address] \leftarrow R_1$

$M[2000] \leftarrow R_1$ $\quad$ $M[MAR] \leftarrow R_1$

Types of Micro operations:

3 types:

(i) Arithmetic micro operations

(ii) Logic Micro Operations

(iii) Shift Micro Operations

(i) Arithmetic micro operations:

- Addition, substraction,
- Complement : $R_1 \leftarrow \overline{R_2}$
- 2's Complement : $R_1 \leftarrow \overline{R_2} + 1$
- Adding with 2's complement : $R_1 \leftarrow R_2 + \overline{R_3} + 1 \cong R_1 \leftarrow R_2 - R_3$
- Increment, Decrement

(ii) Logic MicroOperations:

- AND : $R_1 \leftarrow R_2 \wedge R_3$
- OR : $R_1 \leftarrow R_2 \vee R_3$
- XOR : $R_1 \leftarrow R_2 \oplus R_3$
- X-NOR : $R_1 \leftarrow R_2 \odot R_3$

(iii) Shift Micro Operations:

3 types:

1. Logical shift — Left Shift / Right shift

Left shift : 1011 ⇒ 0110

right shift : 1011 ⇒ 0101

Every time a new bit is added, the new bit is '0'.

2. Circular shift : (rotation)

i) left circular shift :

1011 ⇒ 0111

ii) right circular shift

1011 ⇒ 1101

3. Arithmetic shift :

* → Arithemetic shift can be performed only on signed numbers.

* → After performing arithmetic shift the signed must be retained.

Right shift :

1011 → 1101

0011 → 0001

Left shift :

It is same as logical left shift but it is allowed by CPU only when sign is not going to change.

Eg : 1101 → 1010

Eg : 0101

Here apply left shift will result in change of sign

∴ This operation will not be allowed by CPU. CPU generates Arithmetic left shift overflow error.

nme a new bit
ded. the new bit

ly on signed

igned must be

it is allowed
to change.

esult in change

CPU.

Word addressable & Byte addressable memory

word addressable:

Every word has an address. Bytes within a word doesn't have any address.

Byte addressable:

Every byte has an address.

Instructions:

Instruction: | opcode | operands |

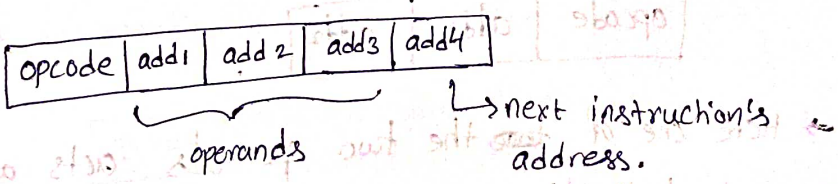
ISA (Instruction set Architecture):

It is collection of all the instructions supported by CPU

Types of instruction based on operands:

(i) **4-address instruction:**

within each instruction a maximum of 4 addresses can be specified



→ The CPUs supporting 4-address instructions don't have Program Counter.

→ Also here inst^n's need not to be store contiguously

Disadvantage:

→ Large size instructions

∴ large program

→ Instruction fetch takes more time.

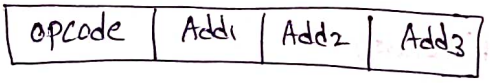
→ Relocation is not easy.

∵ we need to modify add4 of every instruction.

(ii) 3-Address Instruction:

→ Here we use PC instead of 4th address field.

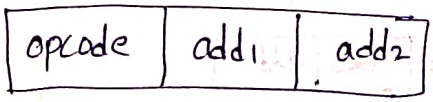
→ So a maximum of 3 operands can be given



→ All disadvantages of 4-address instruction are eliminated here.

(iii) 2- Address Instructions:

→ Maximum of 2-addresses can be specified within an instruction.



→ Here one of the two operands acts as both source and destination.

Disadvantage:

* → More number of instruction in program compared to 3-address format.
→ So it require more memory for program

Eg: Consider $x = (a+b) * (c+d)$

3-address:

$R_1 \leftarrow a+b$

$R_2 \leftarrow c+d$

$x \leftarrow R_1 * R_2$

2-address

$R_1 \leftarrow a$

$R_1 \leftarrow R_1 + b$

$R_2 \leftarrow c$

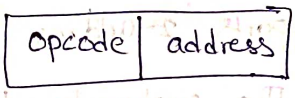
$R_2 \leftarrow R_2 + d$

$R_1 \leftarrow R_1 * R_2$

$x \leftarrow R_1$

(iv) 1-address instruction:

→ Maximum of 1 address is given in an instruction.



→ The 2nd operand is accumulator (implicitly)

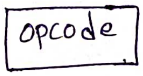
→ This accumulator based architecture.

Disadvantage:

→ More no of instructions in program

(v) 0-address instruction:

→ No address is mentioned in instruction



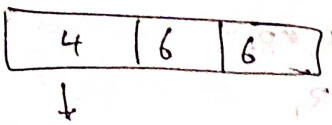
→ This is used in stack based architecture.

The operands are taken from the stack.

11/11/20

Q1) Consider a digital computer which supports only 2-address instructions each with 16-bits. If address length is 6-bits then maximum and minimum how many instructions the system can support.

Sol:



$\therefore max = 2^4 = 16$

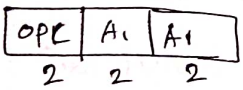
min = 1 instruction

Multiple instruction support:

Q2) Consider a system which supports 2-address instructions and 1-address instructions both. The system has both 1-address and 2-address instructions. The system has 8-bit instructions and 2-bit addresses. If there are three 2-address instructions in the system then maximum and minimum no of 1-address instructions supported is?

Sol:

2-add:



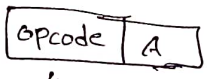
max: ∴ 4 2-add ins
(00, 01, 10, 11)

but given only 3
2-add instructions

assume 00, 01, 10 are used:

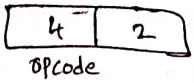
let 11 be unused for
2-address instruction

1-add:



max: ∴ 16 1-add ins

Now every instruction that starts with 11 is 1-address instruction.



1100
1101
1110
1111

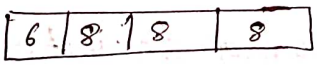
∴ The system can have a maximum of 4 1-address instructions.

Minimum 1-address instructions = 1

(83) Consider a system which supports 3-address and 2-address instructions. It has 30-bit instructions with 8-bit addresses. If there are 'x' 3-address instructions then find ^{max} no of 2-address instructions possible.

sol:

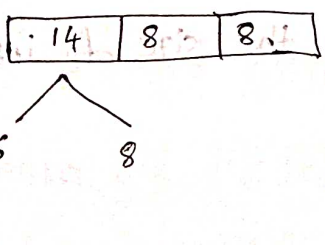
3-add:



↳ ∴ 2^6 3-address instructions are possible but only x are used

$2^6 - x$ are unused.

2-add:



∴ $(2^6 - x)\,2^8$ are max no of 2-address instructions possible.

(Q4) In the previous question, if there are 512 2-address instructions then find no of 3-address instructions.

Sol:

$$(2^6 - x)2^8 = 512$$

$$2^6 - x = \frac{512}{2^8}$$

$$2^6 - x = 2$$

$$x = 64 - 2 = 62$$

Addressing Modes

Effective address:

→ Address of operand in a computation type instruction or target address in a branch type instruction.

Instruction cycle:

There are several phases in execution of an instruction. These phases are called instruction cycle.

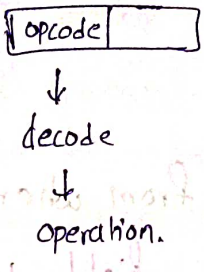
→ Generally there are 6 phases.

(1) Instruction fetch:

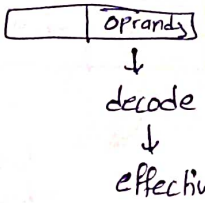
→ Instruction is brought to IR from memory.

→ Now PC is increamented by the size of instruction that is fetched

(ii) Instruction decode:



(iii) Effective address calculation:



(iv) Operand fetch

From the effective address computed operands are brought to the CPU.

(v) Execution:

The operation is performed and result is stored in the accumulator.

(vi) Copy back result:

Now the result is stored in memory.

This process is called instruction cycle.

* → These phases are divided into 2 cycles

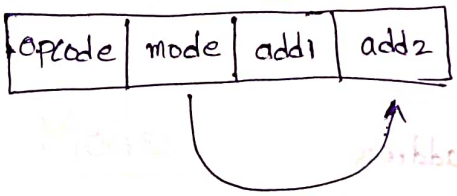
(i) fetch cycle: Instruction fetch

(ii) execution cycle: remaining 5 phases

→ Every instruction need not require all 6 phases of the instruction cycle.

Defn of addressing mode:

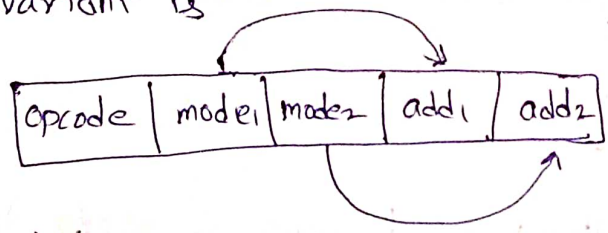
Addressing mode specifies how and from where the operands is obtained using address field value of instruction.



Here mode specifies how add_2 has to be interpreted.

add_1 is always taken from register.

Another variant is



Different addressing modes are

(i) Implied mode:

→ Here one of the operands is understood by the opcode itself

Eg: INCA

This instruction is increament accumulator

Here accumulator is implied from the opcode.

(ii) Immediate mode:

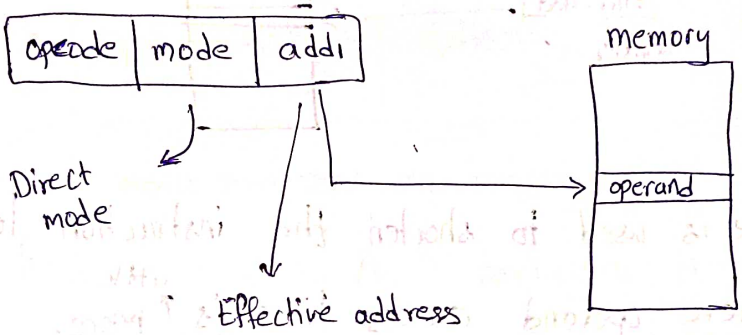
whatever is present in operand field that itself is considered operand.

generally operand range is small in immediate mode

→ It is used to initialize registers with constants.

(iii) Direct mode / Absolute mode:

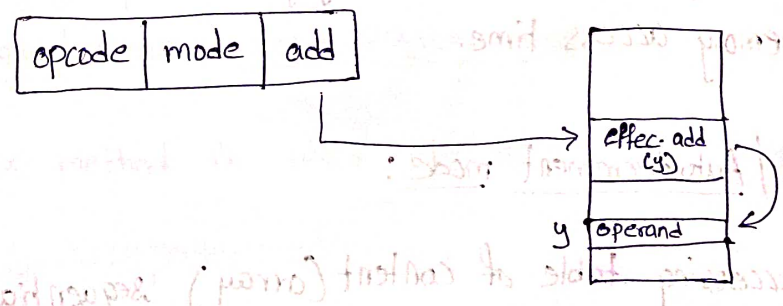
address field has address of operand.



→ Suitable for accessing static variables.

(iv) Indirect mode:

Address field has address of main ~~main~~ memory in which effective address is present.



Denoted as (loc)

Eg: Add R1, (1000)

This not much suitable for pipeline execution as it require 2 mem accesses for 1 operand fetch

So indirection is generally used through registers

→ This mode is used for implementing pointers.

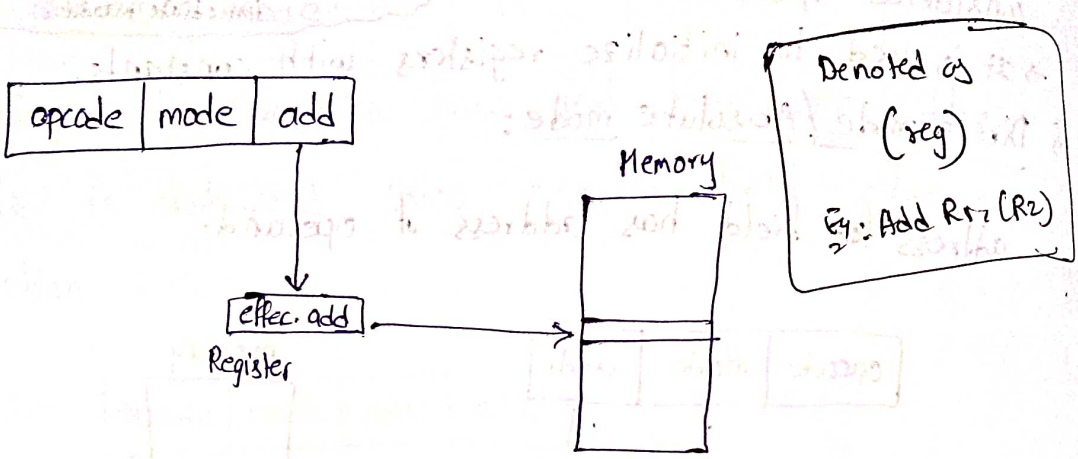
12/11/20 → Used in parameter passing.

(v) Register mode / Register direct mode:

Address part of instruction specifies a register which hold the operands.

(vi) Register indirect mode:

→ The address field specifies the address of register which contains effective ~~addr~~ address



→ This mode is used to shorten the instruction length.

→ However here operand access time is ↑ little more.

Direct mode → 1 memory access time.

Reg. indirect mode → 1 memory access time
+
1 register access time

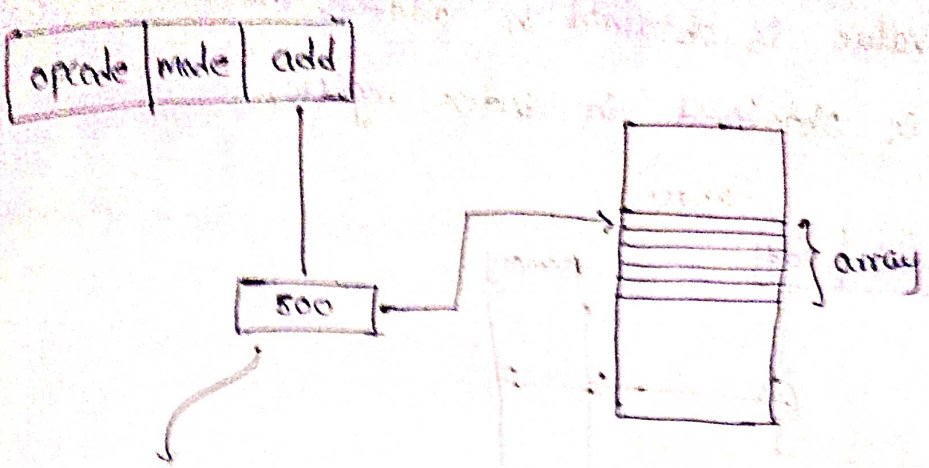
But register access time negligable compared to memory access time.

(vii) Auto increment / Auto decrement mode:

→ Used when accessing table of content (array) sequentially.

→ It is a variant of register indirect mode.

→ Content of register (effective address) is automatically increamented or decremented.



After 1 access, the content of register is increamented / decremented automatically.

* Auto increment mode ⟹ post increment

i.e., first memory access is performed then value is incremented. Denoted as (Ri)+

* Auto decrement mode ⟹ pre defement

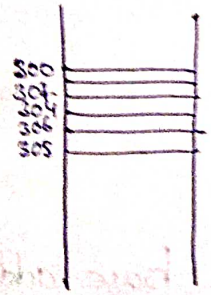
i.e., first decrement content of register then perform memory access.

Denoted as : -(Ri)

The amount to be inc/dec is implicitly determined by the size of operand accessed

(viii) Indexed mode / Index register mode:

→ This method is used to access an array element.



address of A[3] = 300 + 3*w

↳ index value

= 300 + 6

= 306

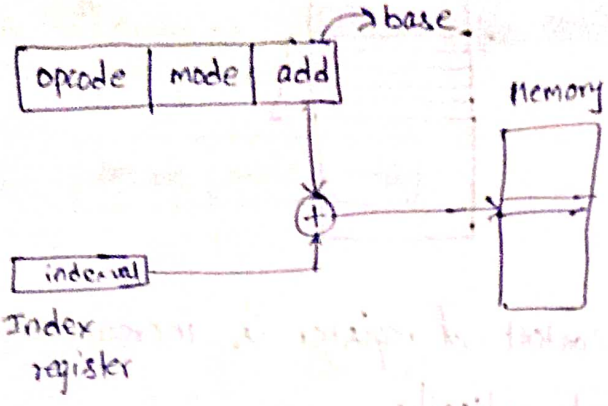
∴ effective address = base + index value

→ It is denoted as X(R)

Eg: Add X(R1), R2

→ The base value is obtained in address field

→ The index is obtained in index register.



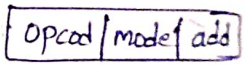
Generally there will be 2 instructions for accessing A[i]

1. Index register ← w * i
2. operand from base + index value.

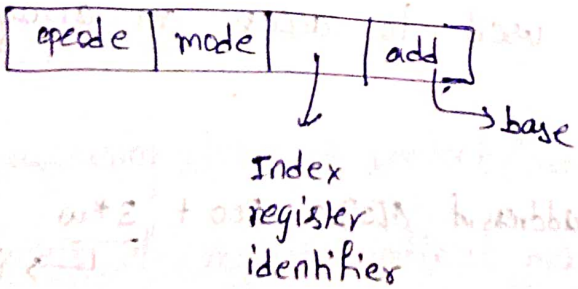
There are two ways for implementing this mode.

(i) Index register is special purpose register:

Here CPU knows where index register is. i.e., address of index register is implied.



(ii) Index register is not a special purpose register:

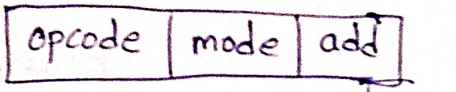


Disadvantage:

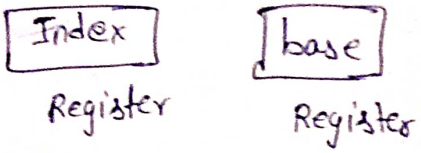
→ If array has to be relocated, then the base address which is stored in every instruction (array access instruction) has to be changed. This is a costly operation and hence this mode is not preferable. (Updation of instruction is costly)

(ix) Index, Base register mode:

This is same as Indexed mode but the difference is that the base value is also put in a register.



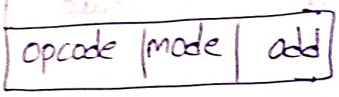
→ This field has no use



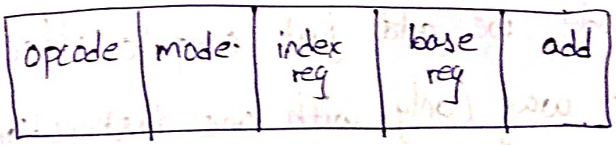
→ Here the disadvantage of indexed mode is overcome.

→ Here also 2 implementations are possible.

(i) Index, base registers are special purpose registers:



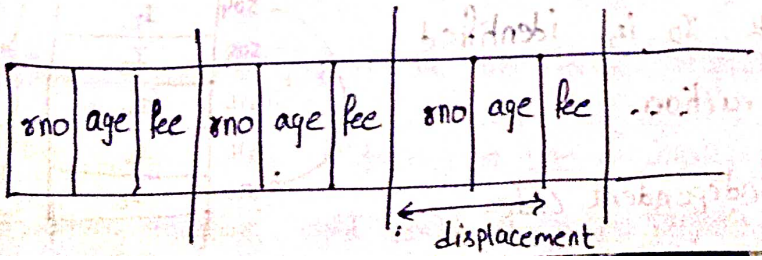
(ii) Index, base registers are special purpose registers



(x) Index, base register + Displacement mode:

→ This method is used for accessing members within ~~records~~ array of records (array of structures).

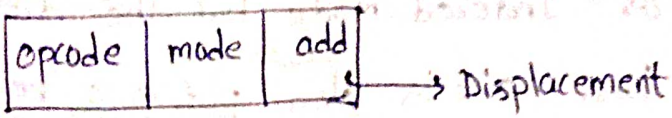
Eg: student(rno, age, fee)

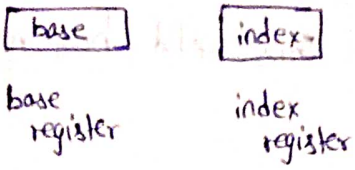


Assume accessing S[2].fee

base index displacement

∴ Effective address = base + index + displacement





(xi) Scaled mode:

→ It is upgraded version of indexed register mode.

→

$$\text{Effective address} = \text{address field value of instruction} + (\text{index register value} * \underbrace{\text{scaling factor}}_{\text{taken implicity.}})$$

→ In index register mode, we need to execute two instructions.

→ In scaled mode we ~~add~~ just compute effective address in a different way (only with one instruction) and thus it is fast.

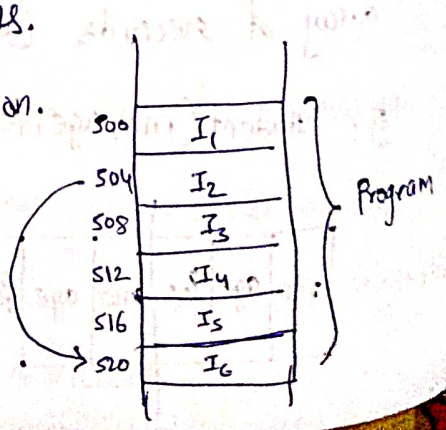
(xii) PC-Relative mode / Position independent mode:

→ This used for branch instructions.

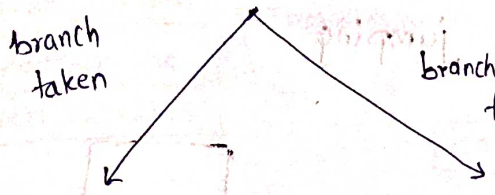
→ Assume, I_2 is currently under execution.
⇒ PC = 508

→ In decode phase I_2 is identified as branch instruction.

→ Use for position independent code.



now there will be 2 ~~possibilities~~ possibilities.



Assume target instruction is I_6

next instruction in the sequence will be executed

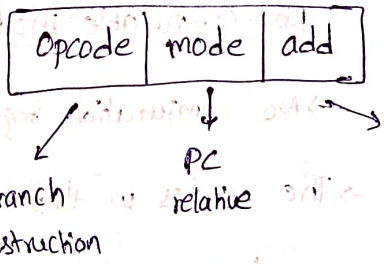
∴ no change in value of PC

target address
= PC + 12
= 508 + 12
= 520

This is obtained based on no of instruction that have to be skipped

Effective address of branch instruction is target address

EA = PC + no of locations to be skipped



EA = PC + add field
↳ offset

→ This mode is used for intra segment jump.

i.e., jump within same segment

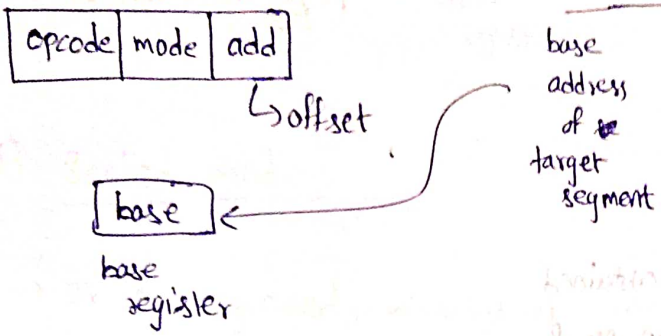
For inter segment jump we use base register mode.

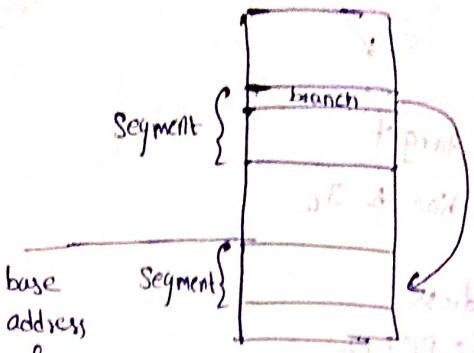
→ PC relative mode also reduces a size of instn because without address being relative, add field has to be larger.

(xiii) Base Register mode::

→ used for inter segment jumping

target address = base register value + offset





→ This mode allows relocation.

Classification of addressining modes:

Computable modes

→ Computation is req for obtaining EA

→ The modes under this category are:

(i) Auto inc / Auto dec.
(ii) Indexed mode
(iii) Index, base register mode
(iv) Index, base reg + displacement
(v) Scaled mode
(vi) PC-relative
(vii) base register mode } used for braching (to calculate target add in branching)

→ Remaining all are used for operands.

Non-Computable mode

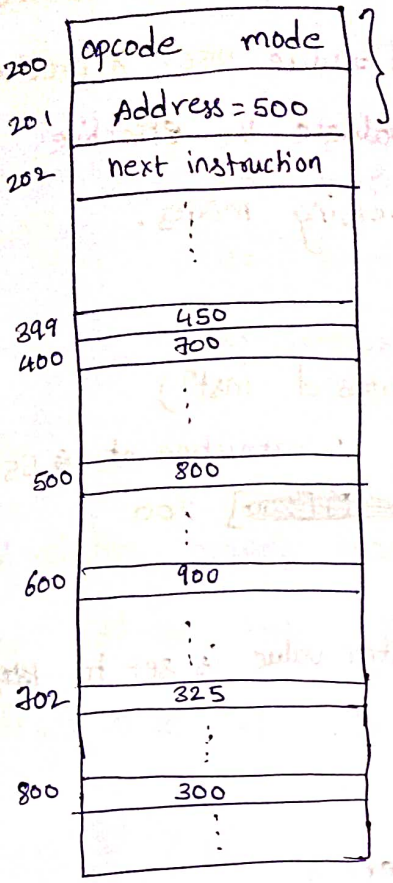
→ No computation required.

→ The modes in this category are:

(i) Implied
(ii) Immediate
(iii) Direct
(iv) Indirect
(v) Register direct
(vi) Register indirect

13/11/20

Eg:



200 PC → After fetch PC is set to 202

400 R_1

100 - index register

In immediate mode EA is address of instruction

Mode	Effective address	Operand
1. Immediate mode	201	500
2. Direct mode	500	800
3. Indirect mode	800	300
4. Register mode	—	400
5. Register indirect	400	700
6. Auto decrement	399	450
7. Indexed mode	600 (500 + 100)	900
8. PC relative	202 + 500 = 702	~~325~~

EA is given only MM references but not register references

(Assume 500 is refering register)

Predecrement

Assuming that 500 is no of locations to be skipped

target instruction

(Q5) An instruction is stored at location 300 with its address field at location 301. The address field has value 250. A processor register contains the number 200. Evaluate the effective address for each of the below addressing modes.

1. Direct ⟶ 250
2. Immediate ⟶ 301 (address of instn)
3. Relative ⟶ 302 + 250 ∴ instruction at 552
4. Register indirect ⟶ ~~(content of R1 = 200)~~ 200
5. Auto increament ⟶ 200 (EA)
   New register value is set to ~~200~~ 201

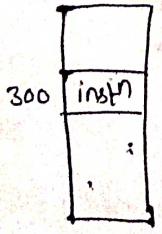
→ In pc-relative, jumping can be ~~either~~

(i) forward jumping : jumping to next address
i.e., offset is +ve

(ii) backward jumping : jumping to previous addresses
i.e., offset is -ve.

(Q6) A relative branch mode type instruction is stored in memory at address 300. The branch is made to an address 450.

a) What should be the value of relative address field of the instruction.



PC value during execution is 301

∴ 450 = PC + address field content
= 301 + address field
∴ 149

b) Determine
and af

(Q7) A relative
an addre
offset value

301 :

(Q8) 4-15 For comp
each add
following

S1:

S2:

b) Determine the value of PC before instⁿ fetch, after fetch and after the execution phase

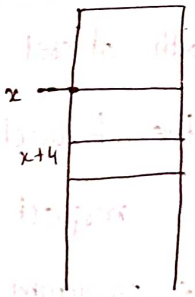
before fetch ⟶ 300

after fetch ⟶ 301

after execution phase ⟶ 450

Q7 A relative branch mode type instruction has taken branch to an address 750. The branch instruction is 4 bytes and it has offset value 400. This starting address of instⁿ in memory is —

Sol:



PC x+4 → during execution

PC + offset = 750

(x+4) + 400 = 750

∴ x = 346

∴ 346 is the starting address.

Q8 (4-15) For computers based on three-address instruction formats, each address field can be used to specify which of the following

S1: A memory operand

S2: A processor register

S3: An implied accumulator register.

a) either S1 or S2

b) either S2 or S3

c) only S2 and S3

d) All of S1, S2, S3

Sol:

An address field can be reference to memory or to a register but implied is not possible.

(References to implied are implied within operation)

∴ opt (a)

(Q9 / 4-17) Consider a RISC machine where each instruction is exactly 4 bytes long. Conditional and unconditional branch instructions use PC-relative addressing mode with offset specified in bytes to the target location of the branch instruction. Further the offset is always with respect to the address of next instruction in the program sequence. Consider the following instruction sequence

Instruction No:	Instruction
i:	add R_2, R_3, R_4
i+1:	sub R_5, R_6, R_7
i+2:	cmp R_1, R_9, R_{10}
i+3:	beq R_1, offset

If target of branch instruction is i, then the decimal value of the offset is _____

Sol:

while i+3 is under execution, PC will have i+4

∴ i = PC + offset

$i = (i+4) + offset$

$offset = -4$ (instructions)

$= -16$ (4*4)

CPU:

→ It is main component of computer system which performs operations and controls the system.

CPU Cycle:

It is amount of time required for a CPU to perform one micro operation.

It is time period of clock.

$$\text{clock rate} = \frac{1}{\text{CPU cycle time}} = \frac{1}{\text{clock time period}}$$

CPI (Cycles per instruction):

No of CPU cycles required to execute an instruction.

Execution time:

time for execution of 1 instruction

i.e., CPI_{avg} * cycle time

∴ execution time for n instructions = $n * CPI_{avg} *$ cycle time

→ CPI need not to be same for every instruction

MIPS : (Million Instructions Per Second)

CPU performance is given by MIPS

→ If CPU takes t-seconds for n instruction then

no of instruction executed in one second = $\frac{n}{t}$ MIPS

no of instructions executed in 1 sec = $\frac{\text{no of instruction}}{\text{Total execution time}}$

= ~~no of~~

$\therefore$ performance in MIPS = $\frac{\text{no of instruction}}{(\text{execution time}) * 10^6}$ MIPS

= $\frac{n}{(n * CPI_{avg} * \text{cycle time}) \times 10^6}$ MIPS

$$MIPS = \frac{1}{CPI_{avg} * \text{cycle time} * 10^6}$$

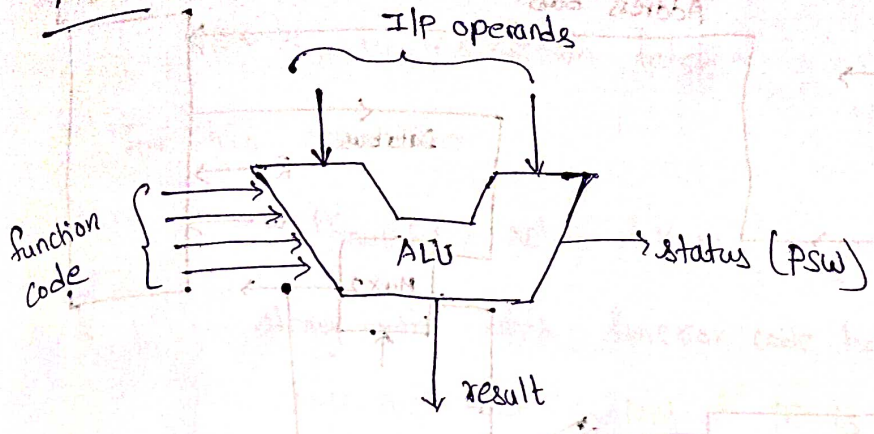
↓ in seconds

$$MIPS = \frac{\text{Clock rate (in Hz)}}{CPI_{avg} \times 10^6}$$

→ MIPS may provide false result for comparision of 2 processes. because it doesn't consider complexity of instructions.

→ To overcome this we use FLOPS (Floating point operations per second)

ALU (Arithmetic & Logical Unit):



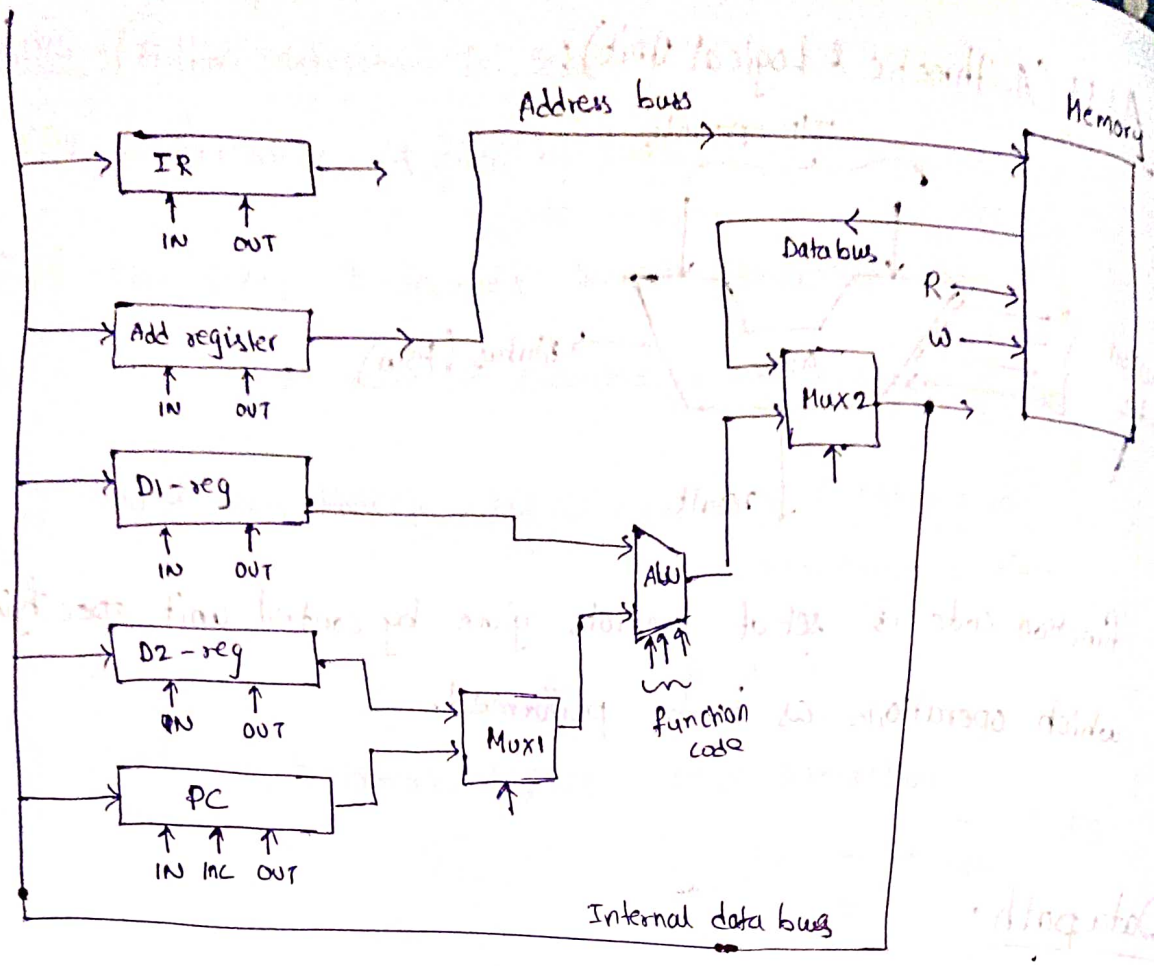
function code is set of controls given by control unit specifying which operations as to be performed.

Datapath:

→ Collection of functional units such as arithmetic logic units or multipliers.

→ Performs data processing operation.

→ For any data processing operations, data follows a specific path. This path is called datapath.



Instruction fetch:

- AR ← PC
- IR ← M[AR], PC ← PC+1

Control Unit:

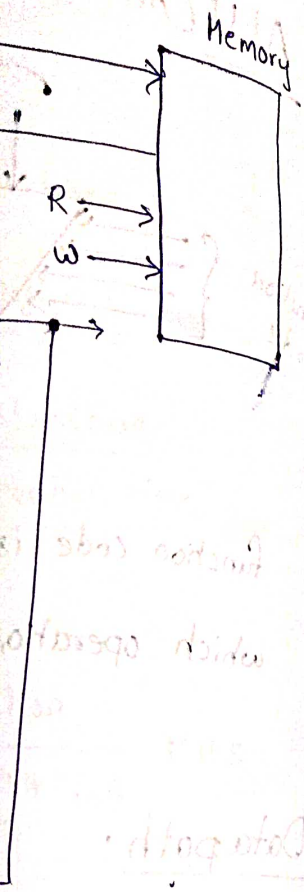
It generates control signals, sends those signal to different components and compononents work accordingly.

Control variable:

It is a name given to control signal

Control word:

Collection of all control signals. Each bit of control word corresponds to one signal. If the bit is 1 the corresponding control signal is sent.



Consider

$AR \leftarrow PC$ in instruction fetch.

For this operation

$PC_{out} = 1$; select of Mux1 = 1; select of Mux2 = 1; $AR_{in} = 1$;

along with this function code has to be given to ALU such that content of PC is o/p of ALU.

All these signals are specified in control word.

Consider

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

$IR_{in} = 1$; $AR_{out} = 1$; read = 1; $PC_{inc} = 1$;

select of MUX2 = 0;

14/11/20

Control unit Organization:

CU can be organized in 2 ways:

i) Hardwired CU.

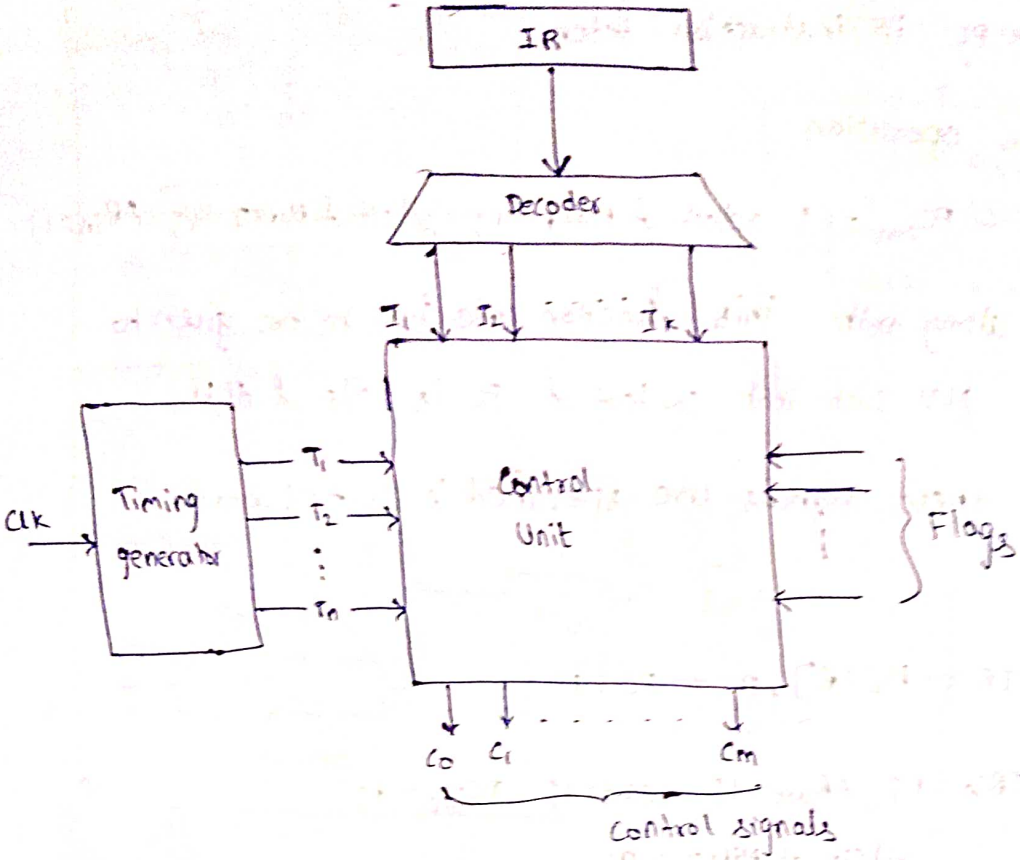
ii) Microprogrammed CU.

i) Hardwired CU:

Control logic is implemented with gates, FFs, decoders and other digital circuits.

Adv: Can be optimized to produce a faster mode of operation

Disadv: Updation is difficult cuz it requires rearranging wires.



we ~~can~~ create a table as shown below

inst^n time / type	I_1	I_2	I_3	I_4	. . .
T_1	C_1, C_2 C_5, C_9	. .	.		
T_2	C_5, K_6				
T_3	C_1, C_3				
⋮	⋮	⋮	⋮	⋮	⋮

no of rows
= no of cycle in that type of instruction

Based on this table we build the hardware by obtaining logical function for each control signal.

→ Timing generator is reset after execution of every instruction.

Q10 Solve gate 2005 & on Hardwired CU design

(ii) Microprogrammed Control Unit

Here all the possible control words are stored in memory.

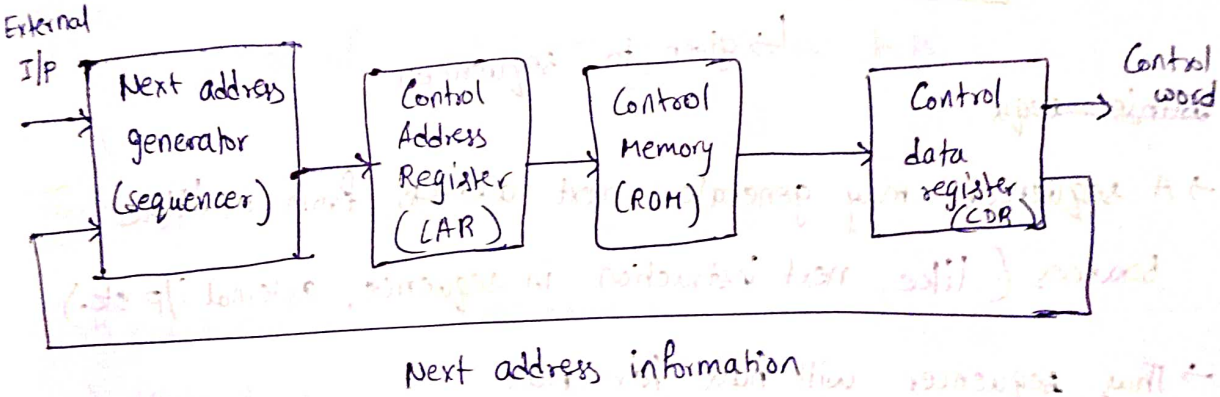
This memory is known as control memory.
Based on req. specified control word is fetched.
→ control logic is implemented with micro-programs

Adv: Updating the control logic is easy.

Disadv: Slower than hardwired CU.

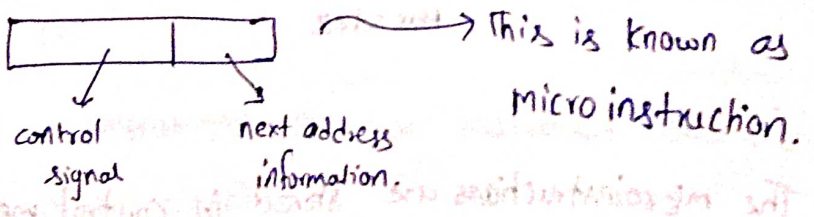
→ Microprogrammed CU is mainly used in complex highend computers.

Control word sequencing:

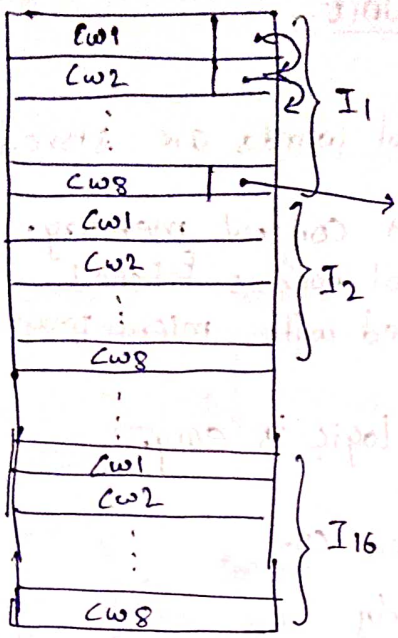


~~Each location~~

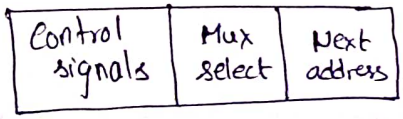
Each location of control memory is as shown below



→ Assume a CPU has 16 different instructions and each instruction has 8 μ-ops. The μ-ops are stored in is control memory as shown



Standard format of micro-instruction:

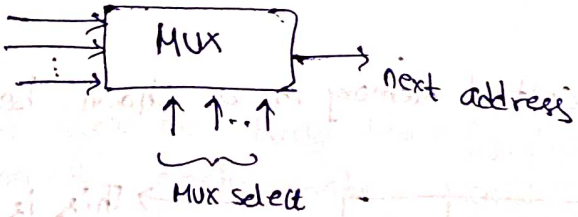


↳ given to sequencer.

~~within sequ~~

→ A sequencer may generate next address from multiple sources (like next instruction in sequence, external i/p etc.)

→ Thus sequencer will have have MUX



Q11 / a-04) The microinstructions are stored in control memory of a processor have a width - - - - - -

take address from
al_input.

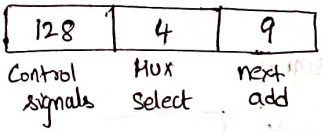
from multiple
e, external i/p etc.)

memory of a

(Q12) A CPU has 64 distinct instructions and each instruction takes 8 micro-operation. Every micro instruction has 3 fields control signals, mux select, address field. Let mux has 16 input. Assume there are 128 control signals what is the size of control memory required.

Sol:

no of ~~conti~~ microinstructions to be stored = $64 \times 8 = 2^9$



∴ size of each micro-inst^n = 141 bits

∴ size of control memory = $2^9 * 141$ bits.

Types of micro-programmed Control Unit:

(i) Horizantal micro programmed CU

(ii) Vertical micro programmed CU

Horizantal:

→ The CU that is discussed so far is called horizantal microprogramed CU.

→ In horizantal μ. programmed CU, we maintained one bit for each control signal.

→ If there a large no of control signals and if a μ. instn needs only a few signals then storing all the bits is waste of space.

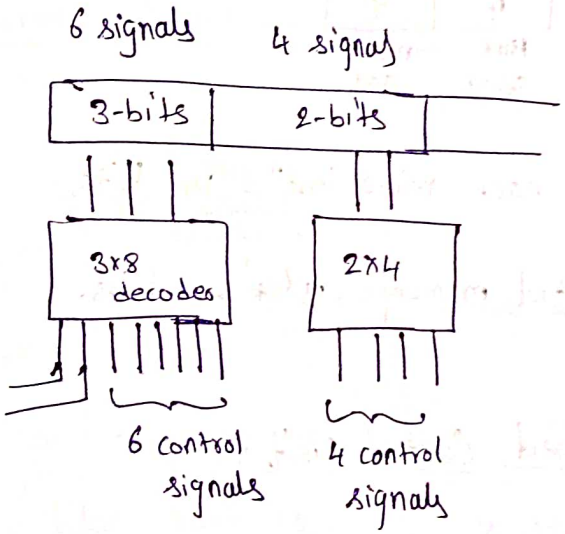
Vertical μ-programmed CU:

→ Here we divide control signals into set of groups. The groups are formed such that, for any micro-instn a requires only one control signal from ~~that~~ group enabled

→ Group sizes may be unequal.

→ These bits of each group are encoded and thus size of control unit is reduced.

→ Consider



There are 2 types of vertical μ-programmed CU

(By default we consider this)

→ Always exactly 1 signal is enable from each group

→ Here for a 4 signal group we encode in 2 bits

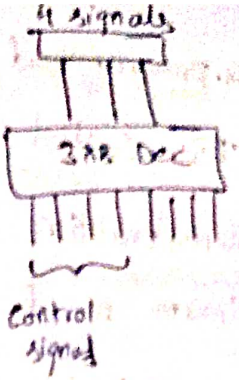
→ Atmost 1 signal is enabled from each group

→ Here for a 4 signal group we encode into 3 bits

∵ we have 5 combinations

groups.
micro-instn
group enabled

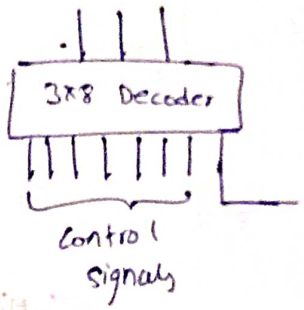
thus size



for 100 to 111 all control signals will be disabled.

→ If we have 7 signals then we need to represent 8 combinations

∴ 3 bits required



for 111 all control signals are disabled.

→ Vertical μ-programmed CU is slower (due to decoders) compared to horizantal μ-programmed CU.

Notes:

→ Sometimes there may be some signals which cannot be grouped. So these signals will be stored in horizantal manner.

1 signal
d from each
oup

a 4 signal
encode into

5 combinations)

(Q12) A μ-programmed CU is required to generate a total of 25 control signals. Assume that during any microinstruction atmost 2 control signals are active. Minimum no of bits required in the control word to generate the required control signals will be?

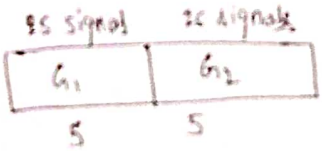
Sol:

Here we don't know how to group.

So we create 2 groups each of size 25 signals

no of combination for each group = 25 + 1 = 26

∴ it requires 5 bits



∴ Minimum bits req = 10

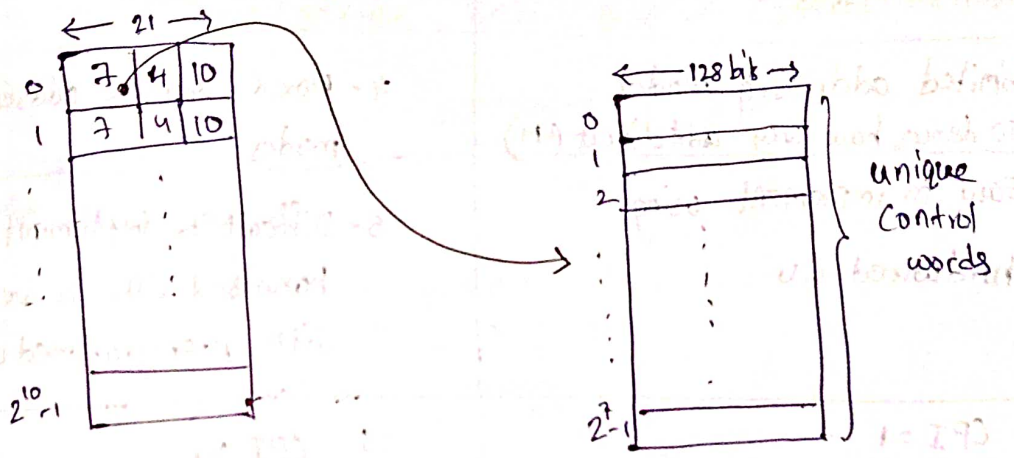
Nano-programmed CU:

→ Some instructions may have common μ-insth. However we are still store them separately in μ-programme control unit.

→ In nano programmed CU we store all unique micro instⁿ separately in memory.

→ Assume we have 128 unique micro inst^n and 128 control signals. Also assume we have 16 inst^n where each inst^n has 64 µ-op. Then organization of nanoprogrammed CU is as shown below:

no of micro instructions = $16 \times 64 = 2^{10}$



$\therefore$ size: $2^7 \times 2^7 = 2^{14}$ bits

Thus in this a way memory required is less and flexibility is also good.

(Q13) Consider a µ-programmed CU which has to support 32 no of inst^{n}s. For each inst^n execution, CU generates a sequence of 64 control words. Each microinstruction contains 3 fields: 118 control signals to support horizontal CU, a MUX select field to select one of 8 inputs and a next address field. The size of control memory used is?

Sol:

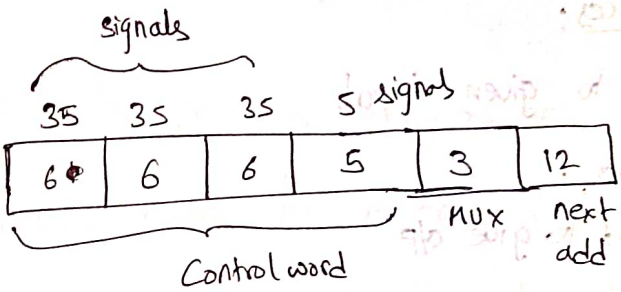
$2^5 * 2^6 (118 + 3 + 11) = 2^{11} * (132)$ bits

RISC vs CISC

RISC	CISC
1. Less no of instⁿs	1. More no of instⁿs
2. Fixed length instructions	2. Variable length Instructions
3. Simple instⁿs	3. Complex instⁿs
4. Limited addressing modes (IO devices have only ~~add~~ direct AM)	4. More complex addressing modes
5. Easy to implement using hardwired CU	5. Diffcult to implement with harwired CU. So we implement with μ-programmed CU
6. CPI = 1	6. CPI > 1
7. Register to register arithmetic operations only	7. reg to mem or mem to reg arithmetic operations possible
8. More no of registers are available	8. Less no of registers are available.
9. ∴ Provides register window (Efficient parameter passing)	9. No register window
10. Complier design is easy Programmer overhead is more	10. Compiler design is complex Programmer overhead is less
11. More compatible for instruction pipeline	11. Less compatible for instruction pipeline.

(Q14) Design of a vertical microprogrammed CU requires to generate 40 signals. Out of first 35, only 3 signals can be active at a time and for remaining 5 anyone can be active any time. The micro instruction of the CU stores control signal information along with 3-bit mux select and 12bit address field. size of control memory required is _____

Sol:



∴ size of each micro instn = 38 bits

no of micro instns = 2^{12}

∴ size of control memory = $2^{12} * 38$ bits

15/11/20

~~Peripheral Devices~~

IO Organization

Peripheral Devices:

→ Devices that are connected to the processor externally are known as peripherals (apart from main memory)

There are 3 types peripheral devices

(i) Input devices:

used to given input

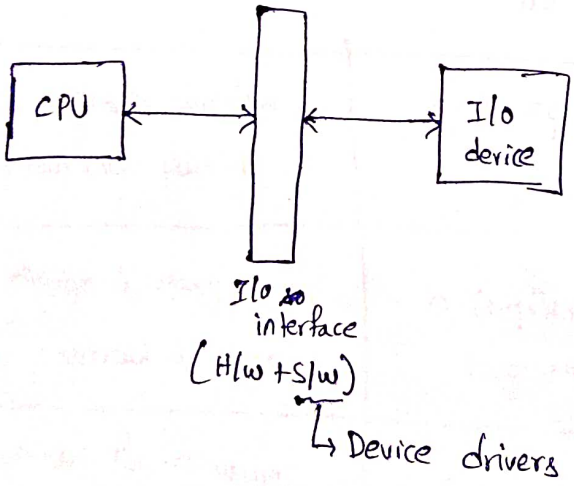
(ii) O/P devices:

used to give o/p

(iii) Storage devices:

used for storage purposes

→ CPU cannot access IO directly. It connected through I/O interface.



→ Interface is needed for (i) signal conversion.

(ii) providing synchronization b/w CPU & I/O

(iii) Data format conversion

(iv) Management & control (so that a peripheral doesn't disturb operation of others)

IO vs Memory Buses

→ Every IO device has an address, so that CPU uniquely accesses an IO device.

∴ We ~~need address bu~~

for IO bus also we have address bus, data bus, control bus.

~~→ IO & memory has seperate buses.~~

~~But they have common address~~

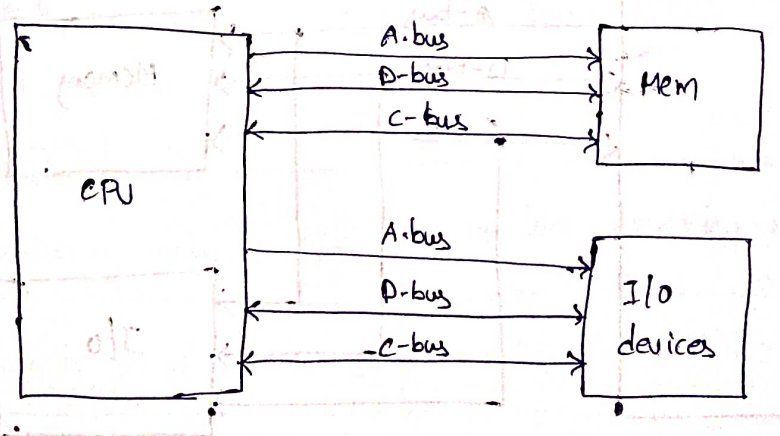
There are 3 types of bus configurations

(i) Seperate buses for memory & IO

(ii) common address and data ~~bus~~ bus but different control buses.

(iii) All are common buses

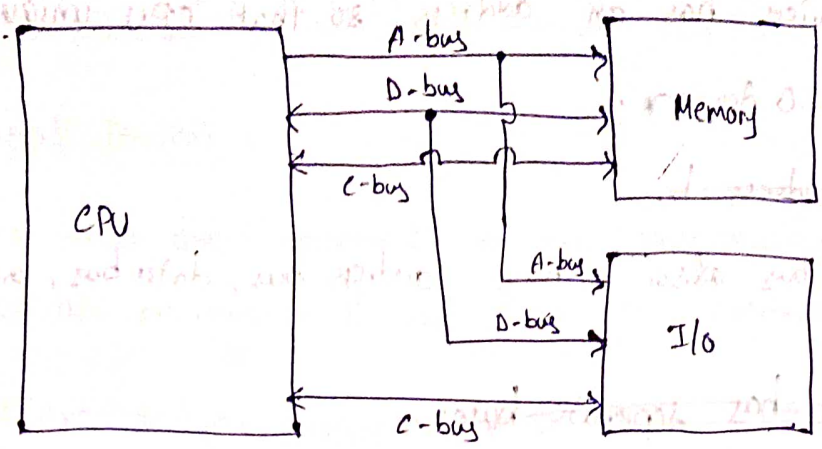
(i) Seperate buses for memory & IO:



→ This organization is costly

→ This is generally used in computers that provide separate I/o processor. Here memory communicates with both CPU and IOP using memory bus.

(ii) Common address & Data buses:

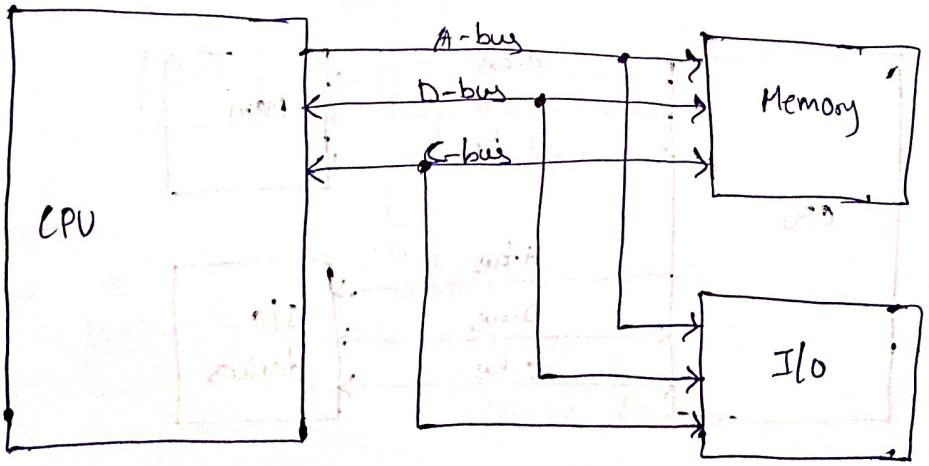


→ Since Address and data buses are same, to resolve the conflict regarding which device CPU wants access, the control signals (read & write) are used.

→ This configuration is called I/O mapped I/O (or) port-mapped I/O. / Isolated IO

→ This is mostly used

(iii) Common address, data, control buses:



~~→ Assume a~~

→ In this configuration IO ~~addr~~ devices a given addresses from memory.

→ Here computers have only one set of read & write signals.

Assume a system with memory 8 bytes & 2 I/O devices d_1 & d_2

Assume I/O devices are ~~allocated~~ given address as

$d_1 \rightarrow 010$

$d_2 \rightarrow 101$

(Here interface register are mapped to memory)

This configuration is called memory mapped IO

→ Here interface register data is stored in memory. So load & store can be used to access ~~data~~. I/O devices also.

Memory mapped I/O	IO mapped IO
(i) IO devices don't have seperate address space	(i) I/O have own address space
(ii) Some memory is wasted	(ii) No wastage
(iii) All the memory access inst^{n}s can be used to access I/O devices also	(iii) I/O : access inst^{n}s and memory access inst^{n}s are different.
(iv) More inst^n for IO access (∵ it has all possible memory access inst^{n}s)	(iv) less inst^n for I/O
(v) ∴ more addressing modes for I/O	(v) less addressing modes for I/O
(vi) Can have more no of I/O devices	(vi) It has less no. of I/O devices

Asynchronous Data Transfer (serial):

→ If two devices involved in data transfer operate under same clock then such data transfer is called synchronous data transfer.

→ In synchronous data transfer, the clock rate is set based on slower devices and due to this performance falls.

→ Thus we use asynchronous data transfer in which we have seperate clocks for devices.

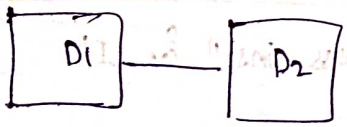
→ Since there is no synchronization provided by clock, we provide external synchronization.

This synchronization is provided by I/O interface.

→ There are two types of data transfers

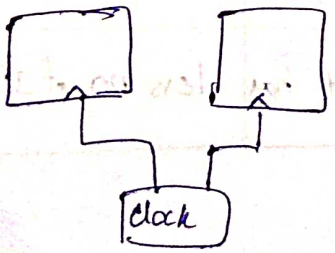
(i) Serial: single line for data transfer.

(ii) Parallel: multiple lines are used

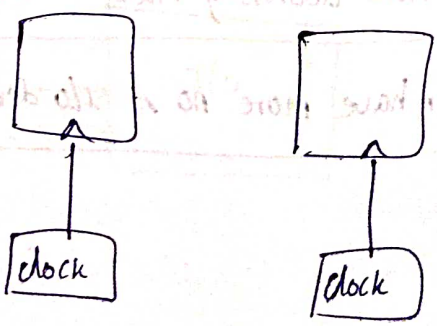


Serial
(less cost)

Parallel
(costly)



Synchronous



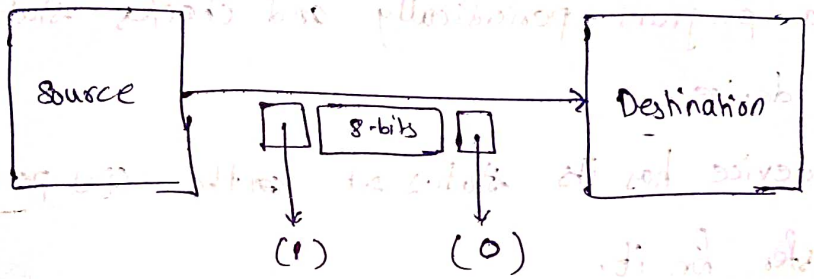
Asynchronous.

→ Let us discuss asynchronous serial data transfer

note:

* whenever data is sent on a line; the last bit of data sent will remain on communication line until new data is sent.

→ In this transfer ~~we provide synchronization by~~ adding a start bit & a stop bit for an 8-bit data.



Here as long as 1 is present on data line it means no data is being sent.

Once a zero is seen, destination understands that data is being set.

→ Here for every 8-bits, we send 10 bits

Thus effective data transfer will be reduced.

Q15 (Grey) How many 8-bit characters

Mode of transfer:

→ It tells about the way in which data can be transfered from an I/O device.

(i) Programmed I/O:

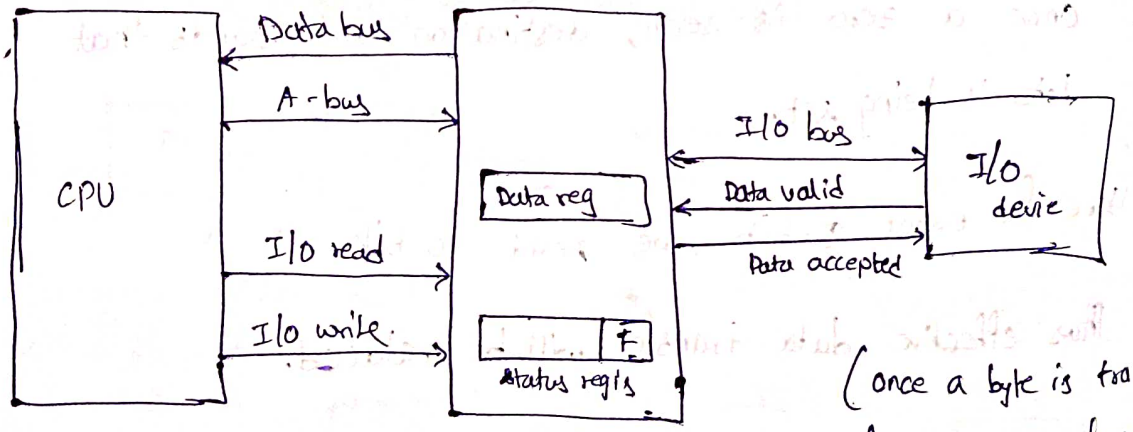
→ There is no provision through which IO can inform CPU about data transfer.

→ IO sets its status & waits.

→ CPU runs a program periodically and checks status of each device.

→ If any device has its status set the CPU performs data transfer for it.

→ This works on the apprach of polling.



Interface

F → flag bit

F=1 → I/O needed

(once a byte is transferred from I/O to Interface, data accepted is enabled by Interface. This is not disabled until the byte is read by CPU)

→ I/O device writes data into data register of the interface and sets its flag saying it is ready for I/O

** → time req in programmed I/O

**

= time req to check status + Data transfer time

→ CPU reads flag reg from I/O (depends on speed of I/O device speed) (time to transfer 1 byte from device) (here 1 byte is size of flag register.)

→ Check flag bit → takes negligible time compared to step 1.

(ii) Interrupt Initiated IO

→ I/O device has provision to interrupt CPU abt communication.

→ when CPU receive interrupt:

(i) It completes execution of current instn

(ii) Saves the status (PC, PSW etc) of current process onto the stack

(iii) Branches to service the interrupt

(iv) Resumes the previous process by taking out the values from stack.

→ Memory never generates interrupt.

ISR:

→ ISR is set instns executed to service an ~~xxx~~ interrupt.

→ All the ISRs are part of device drivers.

Vectored vs Non-Vectored interrupt:

→ In vectored interrupt, the IO device send interrupt signal and vector address of the corresponding ISR.

Interrupt Vector ← → vector add. need not to be mem add: It is index or anything that help obtain address of the ISR.

→ In non-vetored interrupt (scalar interrupt), the device sends only interrupt signal.

Here CPU runs a default service routine through which it obtains the location of the corresponding ISR.

Maskable interrupt vs Non-Maskable interrupt:

→ Maskable interrupt : interrupt that can be rejected if required.

→ Non-maskable interrupt : interrupt that must be serviced.

→ when an interrupt is masked, CPU may completly reject it or can keep it pending.

Internal interrupt vs External interrupt:

External interrupt : If any device generates intr. then it is called external interrupt.
(↓ Hlw intr)

Internal interrupt: Sometimes CPU may encounter an unexpected error while execution. This is known as internal interrupt. Eg: page fault, ~~exceptions~~

(~~Slw intr~~)

Slw intr: Systemcalls

During this ~~kind~~ kind of interrupt, CPU stops execution of current inst^n, solves the problem & restarts the inst^n.

Simultaneous Interrupts:

→ when CPU gets interrupts simultaneously, higher priority device will be service first.

i.e., priority based interrupt handling

(i) S/w soln (Polling)

→ A s/w runs first before than ISR or default service routine. Using this CPU finds the higher priority device.

→ This is based on polling.

CPU checks higher priority device first if it has generated the interrupt. Later it checks next higher priority device and so on...

→ Disadv: If these are more devices, time req to poll could exceed the time available to service I/O device.

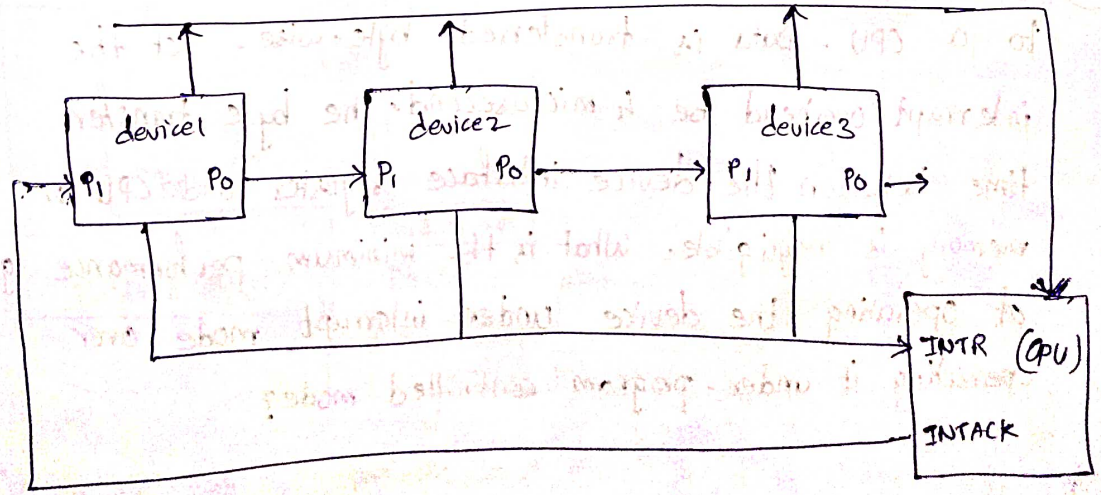
(ii) H/w solns:

- Serial soln (Daisy Chaining)
- parallel soln (not)
  - ↳ not these in syllabus

High priority are assigned to interrupts which, if not serviced would lead to serious consequences.

Also higher priority is given to faster data transfer (Eg: disk) and low priority for slow devices (Eg: keyboard)

Daisy Chaining:



→ CPU receives intr and understands interrupt occurs.

→ Now it sent intack to device1 first.

Device1 will accept ack if it has interrupted and it sends vector address. and it sends '0' to device 2.

Otherwise Device1 will send 1 to device 2;

and this process continues ~~until device is found~~.

i.e., until INTR is set to 0.

After this cpu will disable INTACK.

~~→ Here device1 is selected such that~~

→ In daisy chaining the device that is electrically nearest gets highest priority.

→ Time required in interrupt IO:

interrupt overhead + service routine

→ execution of current instn
→ pushing data onto stack
→ default service routine etc.

Data transfer rate refers to internal speed of device. The rate at which data is produced or processed

(Q16) A device with data transfer rate 10 kB/sec is connected to a CPU. Data is transferred byte-wise. Let the interrupt overhead be 4 microsecond. The byte transfer time between the device interface register and CPU or memory is negligible. What is the minimum performance gain of operating the device under interrupt mode over operating it under program controlled mode?

Sol:

time req in programmed I/O is

= 1 byte data read + transfer time

= 100 μs + 0

= 100 μs

time req in interrupt I/O

= overhead + service time

= 4 + 0

= 4

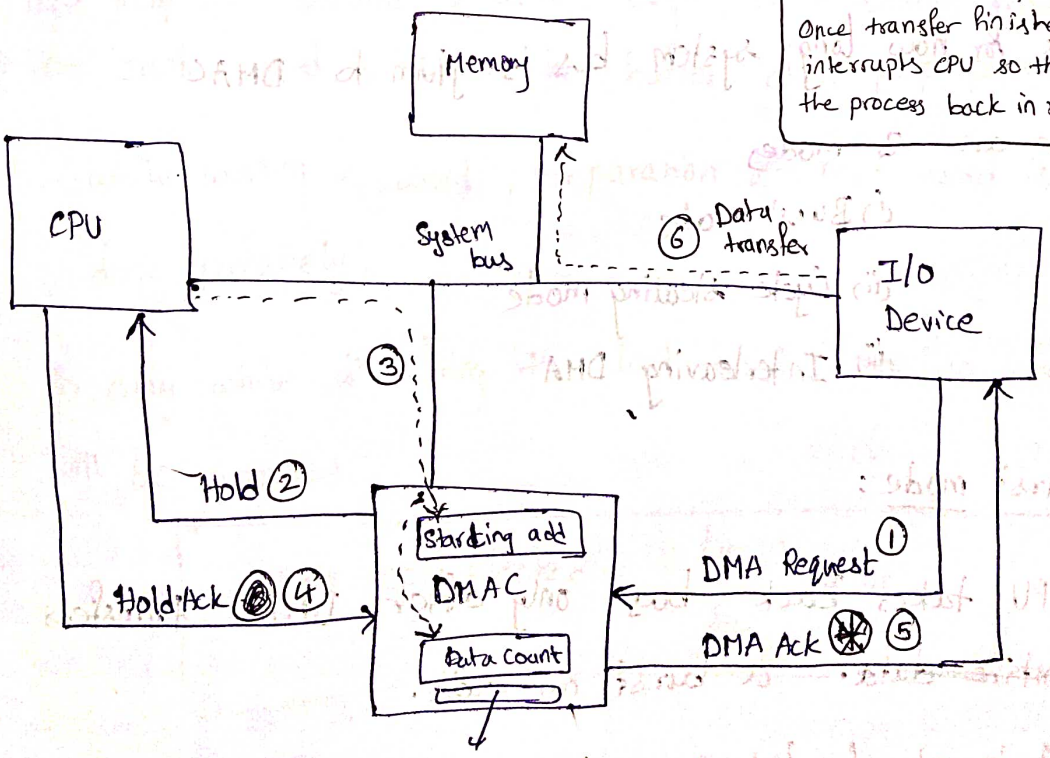
∴ performance gain = $\frac{100}{4}$ = 25

Direct Memory Access

→ DMA is a technique that enables data transfer blw I/O and memory without CPU intervention.

→ H/w required for DMA is DMAC.

When DMA transfer is initiated OS puts the corresponding process in block state and may schedule another process. Once transfer finishes, DMAC interrupts CPU so that OS put the process back in ready state



→ Starting address is address from which memory access has to be done

→ Data count is no of memory locations (words/bytes) to be accessed from starting address.

→ These two information is sent by CPU to DMAC

→ After transfer of each byte/word data count value is reduced by 1. and starting address is increamented by 1.

→ The data transfer stops once data count reaches 0.

→ When system bus is with DMAC, CPU performs only those operations that doesn't require system bus.

This means mostly CPU will be blocked.
(cuz even instn fetch requires memory)

Modes of DMA Transfer:

→ Tells for how long system bus is given to DMAC

→ There are 3 modes

(i) Burst mode
(ii) Cycle stealing mode
(iii) Interleaving DHA

i) Burst mode:

→ CPU takes back bus only after DMAC transfers ~~entire data~~. a burst on data.

→ A burst of data may be entire data or burst of blocks

Cycle Stealing mode:

→ Preferable when IO-device is slow in preparing data

→ So here the idea is to give system bus to DMAC for short time when data is ready with IO device.

→ Here system bus is given to IO for DMA of 1 cycle in which the prepared word is transfered to memory.

→ During preparation of word CPU is not blocked.

→ CPU is blocked during data transfer only.

→ Let time required to prepare data = t_x

time required to transfer data = t_y

∴ % time CPU blocked = $\frac{t_y}{t_x + t_y} * 100$ (Burst mode)

(ii) Cycle Stealing Mode

→ There may be second possible case (Cycle stealing mode)

→ Here each word is stored in buffer after preparation.

∴ while writing a word, preparation of next word is done parallely.

So only while writing the last word, no word will be prepared.

% time CPU blocked = $\frac{t_y}{t_x} * 100$

In cycle stealing mode, transfer time overlaps with preparation time

(iii) Interleaving DMA:

Whenever CPU does not require system bus (doing internal work) then only control of the buses will be given to DMAC.

→ CPU will not be blocked in this.

Note:

→ Among all modes, data transfer will be fast in burst mode, second fastest is cycle stealing and finally interleaving DMA

→ During DMA transfer, DMAC sends read write signals to memory

→ DMAC is a special purpose processor which controls data transfer b/w memory and I/o.

→ For DMA HLDA may be enabled even between the execution of instn.

∴ DMA service can be done during execution of instn.
Interrupt service is done only after end of instn.

→ DMA & CPU can't work parallely. then how DMA improves performance over interrupt I/o.

In interrupt IO CPU state has to be switched.
In DMA no need to switch the state.

∴ DMA given better performance for data transfer b/w I/o & memory.

doing internal
e given to DMAC.

burst mode,
interleaving DMA
e signals to
controls
even between
tion of inst^n.
f inst^n.
s DMA
switched.
transfer blw

→ However for short data transfer blw CPU & I/O interrupt is better.

→ Also we use DMA for large data transfers and interrupt I/O for short data transfers.

16/11/20

Q17) Consider a device operating on 2MBPS speed and transferring the data to memory using cycle stealing mode of DMA. If it takes 2 μs to transfer 16 bytes data to memory when it is ready/prepared. Then % time CPU blocked due to DMA is ____

Sol:

2MBPS → 2×10^6 byts/sec

16 byts → $\frac{1}{2 \times 10^6} \times 16$ sec

= 8 μsec

Here transfer time overlaps with preparation time

∴ % time CPU blocked = $\frac{2}{8} \times 100 = 25\%$

Q18 / 4-16) The size of data count register of a DMA controller is 16 bits. The processor needs to transfer a file of 29,154 kB from disk to MM. The memory is byte addressable. The min no of times DMAC needs to get control of system bus from processor for this transfer

Sol:

DC → 16 bits

Memory is byte addressable.

So max of $2^{16}-1$ bytes can be transfered at once

* * * *

$$\therefore \text{no of times control required} = \frac{29154 \text{ kB}}{2^{16}-1} = 455.55$$

$\therefore$ 456 times

Additional Info:

In asynchronous data transfer, synchronization can be provided in 3 ways

(i) strobe signal (generally used b/w CPU & I/O)

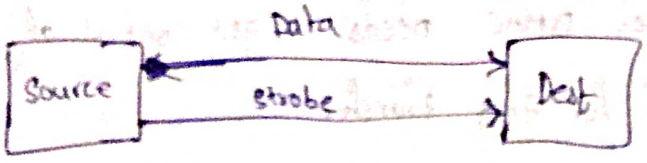
(ii) Handshaking signal (Generally used b/w I/O & interface unit)

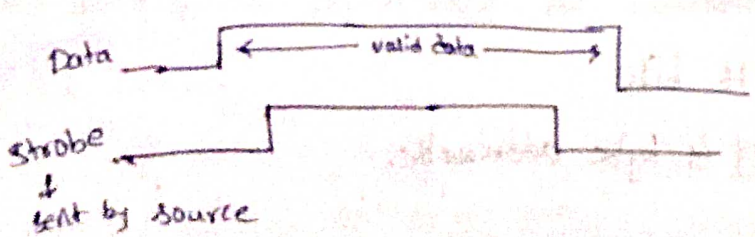
(iii) asynchronous serial data transfer

(i) Strobe signal:

→ Here we maintain a control line to time each transfer through which send strobe signal.

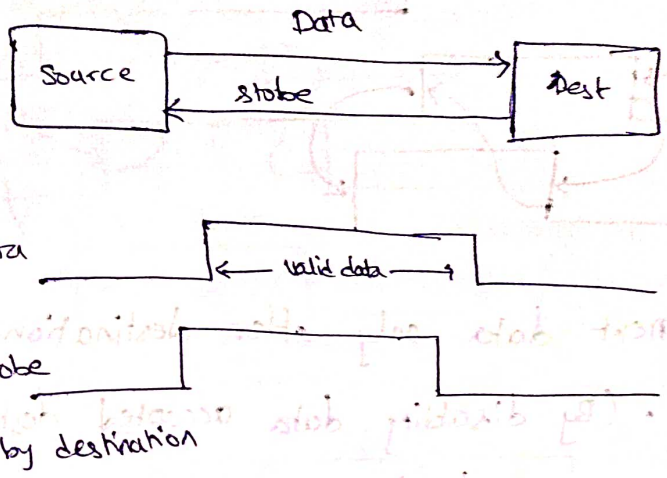
Source initiated transfer.





Here source place strobe signal after placing data.

Destination initiated transfer:



In both the cases, destination often uses (not necessary) falling edge of strobe to trigger destination registers.

→ whenever strobe signal is disabled implies that data on the bus is not valid.

(ii) Handshaking:

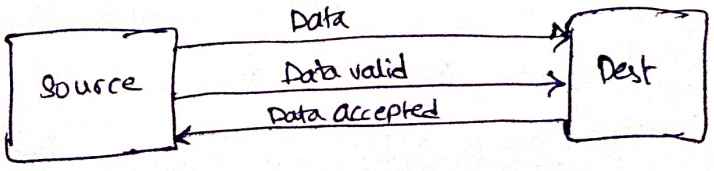
→ The problem with strobe signal is that- source nevers knows whether destination has received the data or not. There is no acknowledgement.

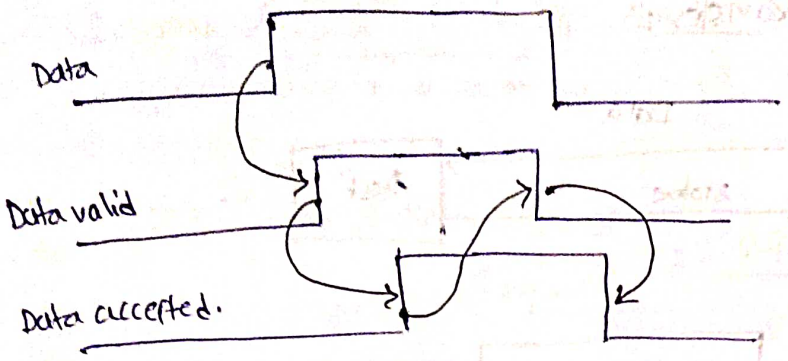
→ Here we use two control signals

Data valid: say data on bus is valid (sent by source)

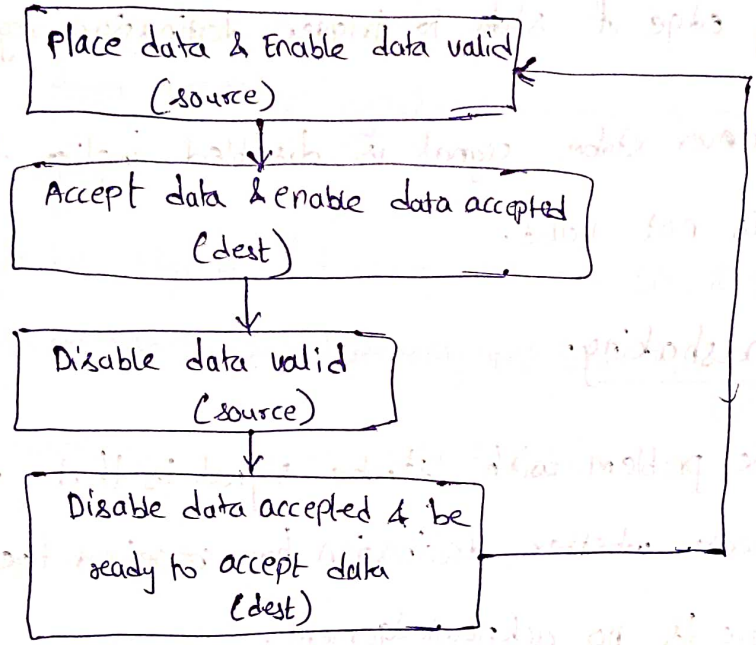
Data accepted: says data is received (sent by destination) and turned off only after it is ready for a next data

Source initiated transfer:





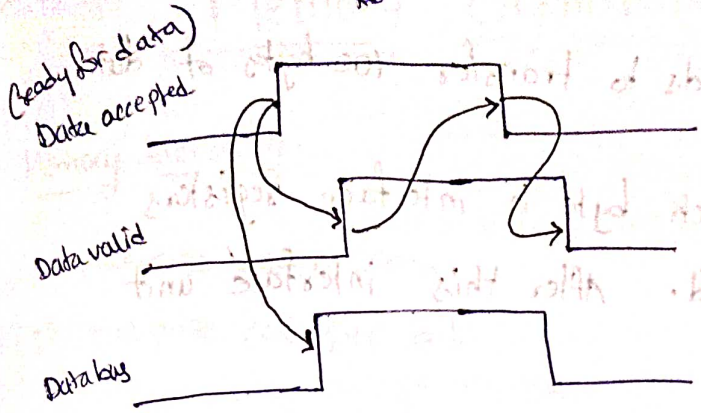
→ Source sends next data only after destination disables data accepted. (By disabling data accepted destination says it is willing to received data).

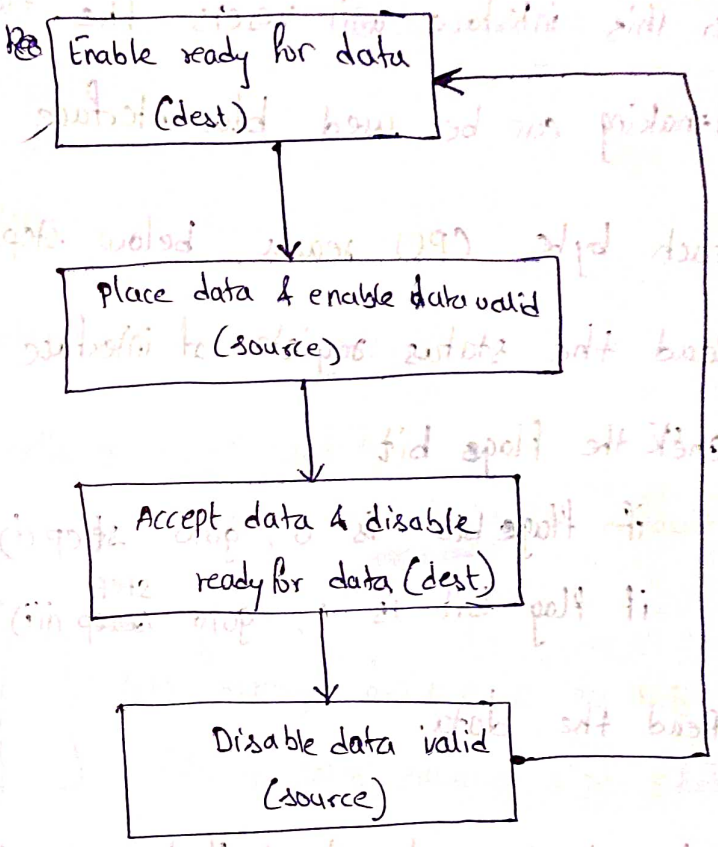


Destination initiated transfer:

→ Here we see data accepted line as ready for data.

→ So destination enables ready for data (data accepted) to start the data transfer.





→ If one of the two devices is faulty, data transfer will not be completed. This kind of error can be detected by means of a timeout mechanism.

Programmed I/O

→ Assume an I/O device needs to transfer 100 bytes of data.

~~For each~~

→ The I/O device puts each byte in interface registers as soon as they are prepared. After this interface unit sets its flag.

→ The CPU sees the flag and reads the data and after this interface unit resets the flag.

→ Handshaking can be used blw interface & I/O device.

For each byte CPU reads, below steps are performed

(i) Read the status register of interface unit

(ii) chek the flage bit

if flage bit is 0, goto step (i)

if flag bit is 1, goto ~~setep~~ step (iii)

(iii) Read the data.

→ Here it is clearly understood that, CPU loops as long a data is being pe-prepared and made available in interface registers.

→ So the time to depends on data transfer rate of device.

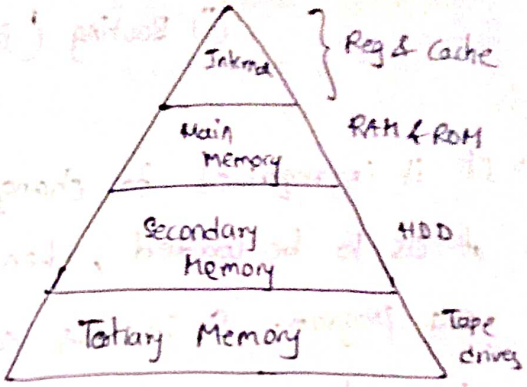
If device needs 't' time to produce data and make it ~~avala~~ available in registers, then CPU would also take around 't' time to finish the IO operation.

17/11/20

Memory Organization

Memory hierarchy:

→ maximize access speed
→ minimize cost per bit.

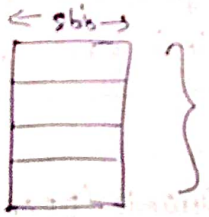


Memory Representation:

→ Memory is represented by

no of cells × 1 cell capacity
↳ in bits (by default)

Eg.



4×8^{bits} memory (or) 4 × 1 B (or) 4 B
i.e., 4 cells each of size 8 bits

~~If it is given like 4GB memory~~ and

Main Memory:

Defn: It is memory that is used to store currently running programs & their data

types:

RAM ROM

(ROM: Non-volatile
RAM: Volatile)

ROM:

→ Initially when CPU is turned on it gets initial program from ROM. This program does 2 things:

(i) Hardware check (POST (Power on Self-Test))

(ii) Booting (Bootstrap program)

→ If it is required to change initial set of programs of OS to be loaded, then we need to change bootstrap program. So in this case we store bootstrap program on disk and store bootstrap loader in the ROM.

RAM:

→ stores current running program & their data along with the OS.

types of RAM:

(i) Static (SRAM)

(ii) Dynamic (DRAM)

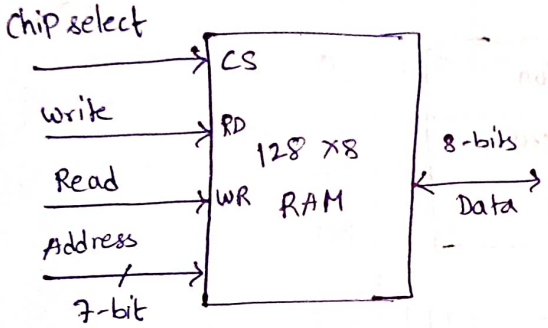
Memory Latency:
It is time blw initiating a request for a byte or a word in memory until it is retrieved by processor

Static (SRAM)	Dynamic (DRAM)
→ made up of Flip-Flops	→ made up of capacitors. Data is stored in the form of electric charges (Charges tend to discharge and data may louse)
→ No refresh is required	→ Periodic refresh is required
→ faster & costlier	→ slower & less costlier
→ Eg: Cache memory	Eg: Main memory

Static	Dynamic
→ idle power consumption is very less → Operational is power consumption is high comparitively	→ idle is power consumption is more → operational power consumption is less

18/11/20

RAM Chip:



Every computer has atleast 2 chips
i) RAM ii) ROM

CS is used to select that chip that we need.

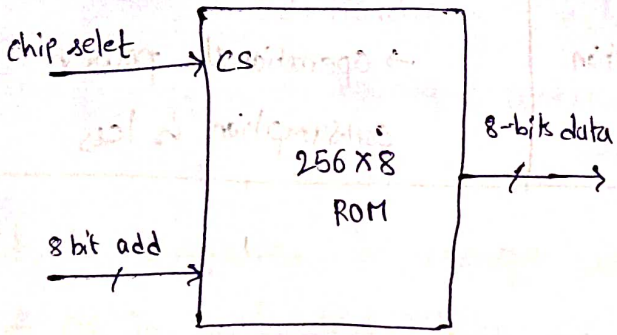
CS	Read	Write	operation
0	x	x	No operation
1	0	0	No operation
1	0	1	write
1	1	x	Read

→ no of pins needed for this RAM chip

$$7 + 8 + 1 + 1 + 1 = 18$$

(add) (data) (CS) (RD) (WR)

ROM Chip:



The only operation performed on ROM is read. So we don't have RD & WR signals.

$\overline{CS}$	operation
0	no operation
1	read

→ No of pines = 8 + 8 + 1 = 17
(add) (data) (CS)

(Q19) A ROM chip of 1024 x 8 bits has four select inputs and operates from a 5-volt power supply. How many pins are needed for the IC package

Sol:

Add → 10
data → 8
CS → 4
power → 1

wherever there is power we must consider a chip for ground also

∴ ground → 1

⇒ total pins = 24

Multiple Chip Support:

total memory capacity = no of chips * capacity of each chip

→ Assume we have two 128 x 8 bit chips;

from this we can build 256 x 8 bit ~~chip~~ RAM
or 128 x 16 bits RAM

add size of each chip = 7 bits

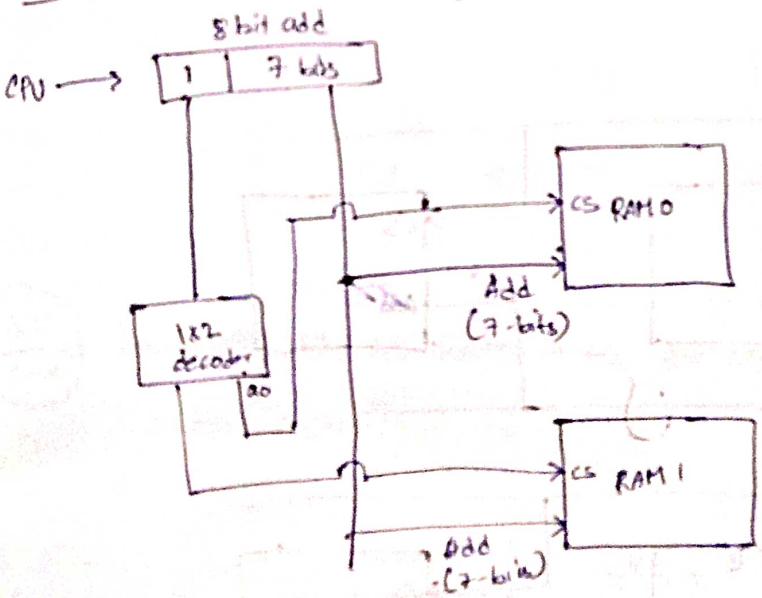
but CPU sees MM of capacity 256 x 8 bits

∴ cpu generates 8 bit address

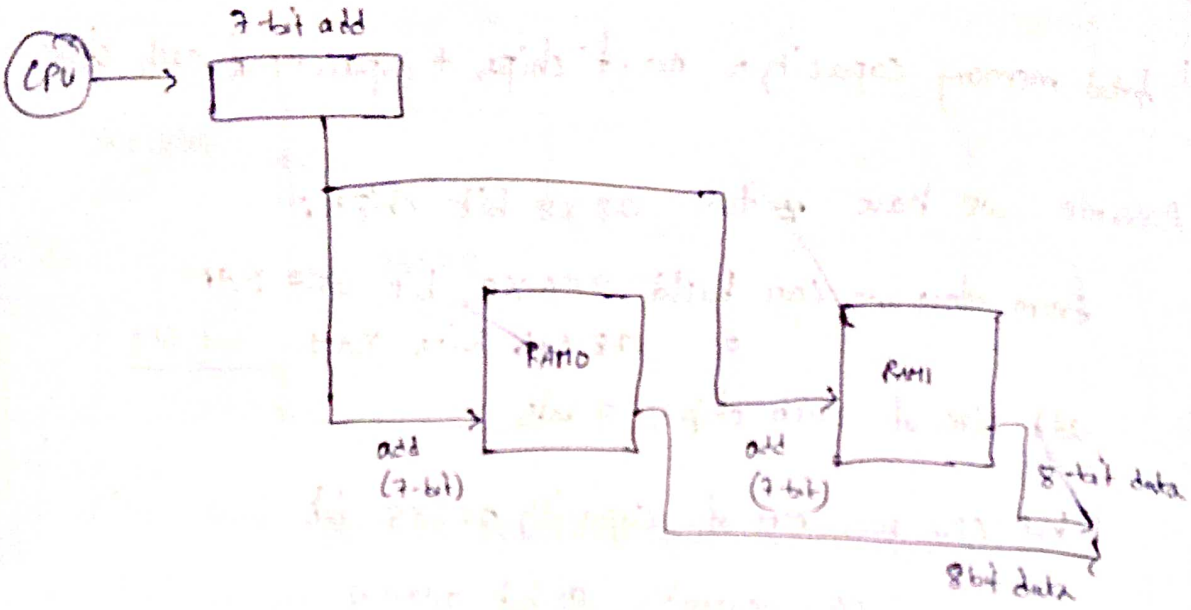
0	0000 0000	~~chip~~ RAM 0
127	0111 1111	
128	1000 0000	~~chip~~ RAM 1
255	1111 1111	

Deciding which RAM to access is done using MSB bit.

256 x 8 : (vertical arrangement)



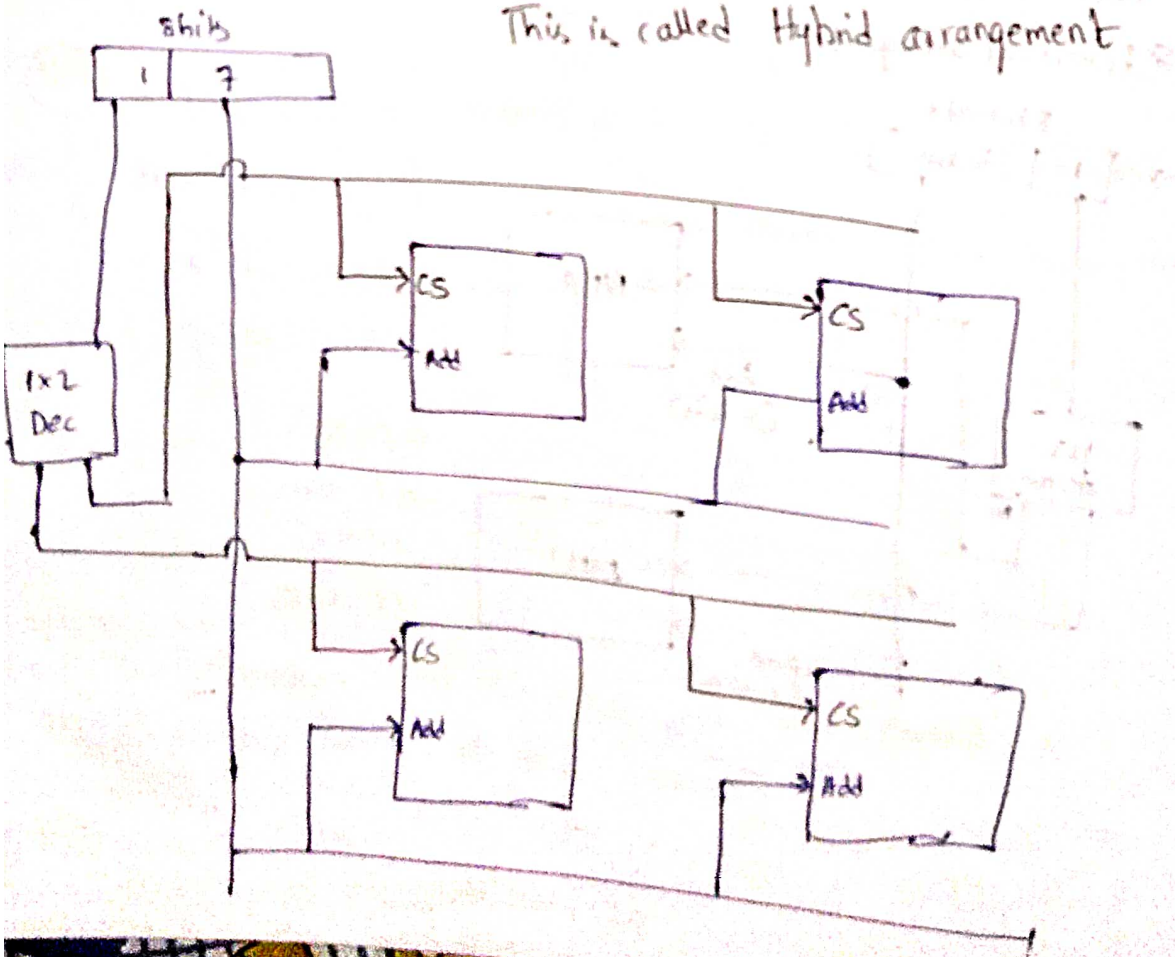
128 x 16 : (Horizantal arrangement)



→ If required addresses are more, then we use vertical arrangement.

→ If data required at each memory location is more then we use horizantal arrangement.

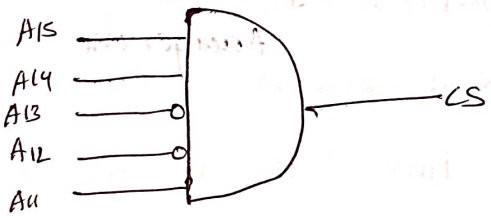
→ Now consider building 256 x 16 from 128 x 8 chips.
This is called Hybrid arrangement.



Q.20 / 4-14 The chip select logic for a certain DRAM chip in a memory system design is shown below. Assum that the memory system has 16 address lines denoted by A15 to A0 what is range of address (in hexadecimal) of the memory system that can get enabled by the chip select (CS) signal

a) e800 to CFFF b) CA00 to CAFF

c) C800 to C8FF d) DA00 to DFFF



Sol:

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}
1	1	0	0	1

∴ Address range is 1100 1000 0000 0000

to

1100 1111 1111 1111

i.e., C800 to CFFF

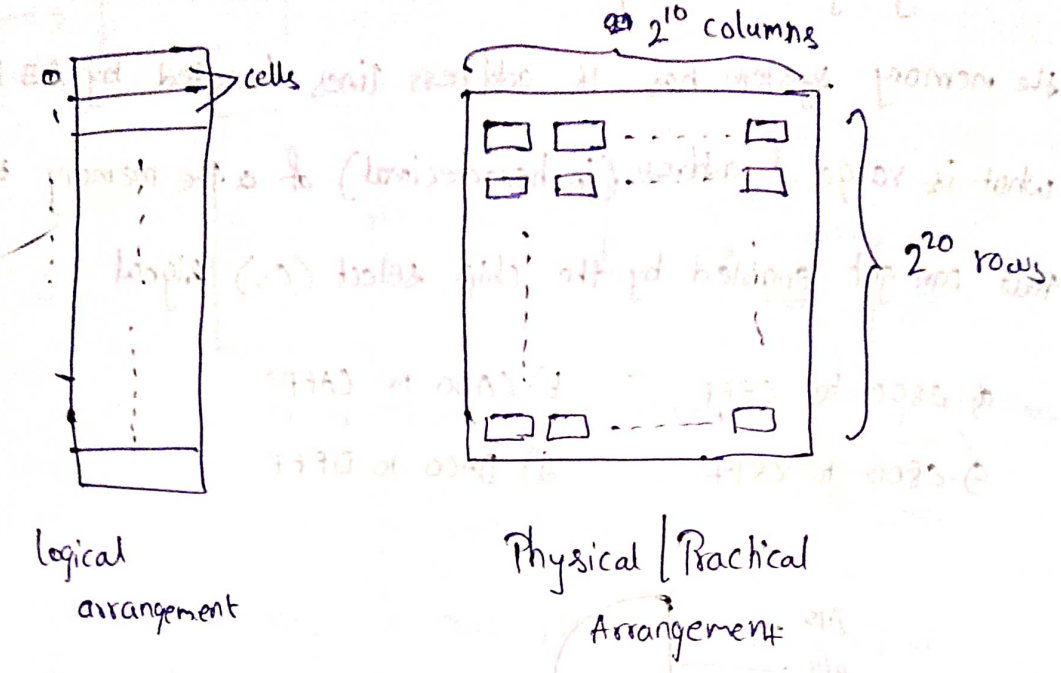
Q.21 / 4-09 How many 32k x 1 RAM chips are needed to provide memory capacity of 256 KBytes.

Sol:

$$\frac{256K \times 8}{32K \times 1} = \frac{2^{18} \times 8}{2^{15} \times 1} = 2^3 \times 8 = 64$$

DRAM chip Cell Arrangement

Consider 1G×1B memory ⇒ 2^{30} cells



logical arrangement

Physical / Practical Arrangement

DRAM Chip Refresh

* In one refresh operation, 1 row of cells can be refreshed.

∴ The above chip needs 2^{20} refreshes.

∴ total chip refresh time = (no of rows in the chip) * (1 refresh operation time)

→ If a system has n chips in memory

total refresh time of the memory system (n-chip refresh time) = 1 chip refresh time

* ∴ Each chip can be refreshed parallely

Q22) A DRAM chip of 512k x8 bits has 256 rows of cells with 2k cells in each row. If DRAM takes 20ns for 1 refresh then total refresh time of DRAM is ____

Sol:

we need 256 refreshes

$\therefore 256 \times 20\,ns$

$= 5120\,ns$

Q23 / G-10) A main memory unit with a capacity of 4 MB is built using 1M x 1-bit DRAM chips. Each DRAM chip has 1k rows of cells with 1k cells in each row. The time taken for a single refresh operation is 100 ns. The time required to perform one refresh operation on all the cells in the memory unit is

a) 100 ns
b) 100×2^{10} ns
c) 100×2^{20} ns
d) 3200×2^{20} ns

Sol:

time req of entire mem refresh = refresh time of single chip

i.e., no of rows x 1 refresh time

$2^{10} \times 100\,ns$

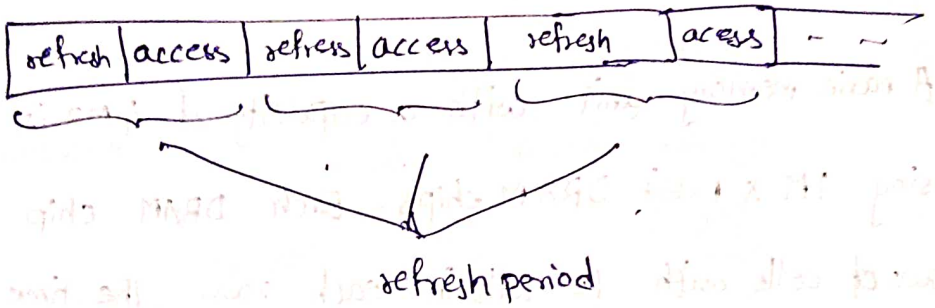
$\therefore$ opt (b)

Refresh period

→ Refresh period is amount of time for which refresh is performed once.

→ Refresh is performed regularly.

→ After performing refresh some part time is used for memory access



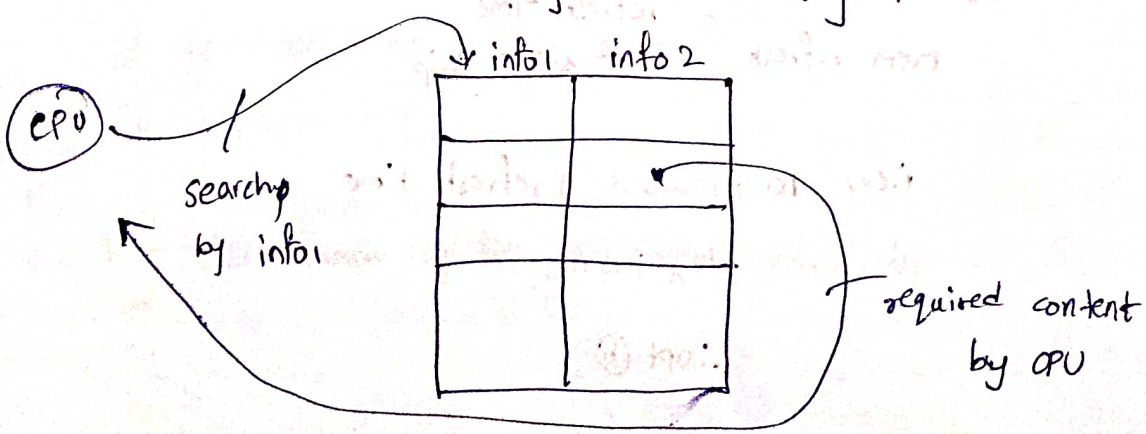
Associative Memory:

→ This is also known as content addressable memory

→ The memory cells of associative memory don't have addresses

→ Every cell contains 2 information. out which one is already known by CPU and other is the one required by CPU

→ so here searching is done using 1st information



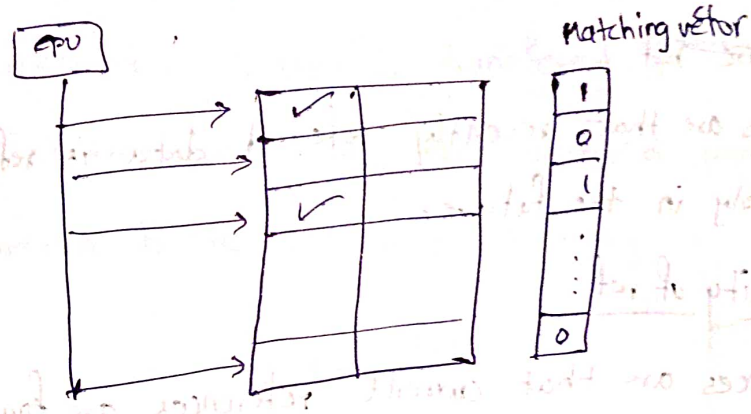
→ Comparision of each cell is done parallely using separate matching logic for each cell.

∴ searching is very fast. (faster than SRAM also)

Also cost is very expensive

→ In associative memory, we maintain a matching vector which contain one bit corresponding to each cell.

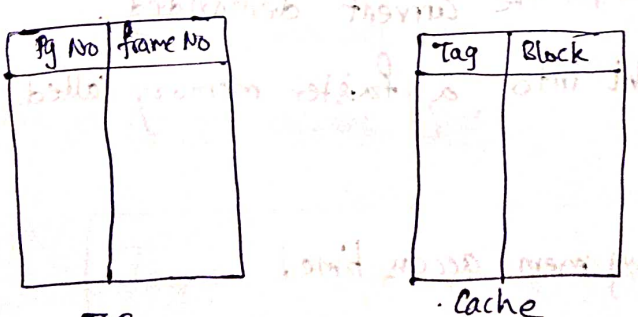
If match occurs the bit is set.



Applications:

is for cache & TLB implementation

↳ use when fully associative mapping is required.



TLB
(search by Pg.No)

Cache
(search by tag information)

Locality of Reference:

If CPU has requested one address for memory access, then that particular address or nearby address will be accessed soon.

types:

(i) Temporal locality of ref.
(ii) Spatial locality of ref.

(i) Temporal locality of ref

~~The future~~ ref

- Chances are that recently refered data is refered most probably in the future.

(ii) Spatial locality of ref

Chances are that current references are found around the previous references.

Cache Memory:

→ Based on locality of ref, the current demanded localities are brought into a faster memory called cache.

→ Using cache reduces avg mem. access time.

Cache hit: If CPU's demanded content is present in cache then we say it is cache hit

Cache miss: If CPU's demanded content is not present in cache, then we say it is cache miss.

→ Performance of cache is given by cache hit ratio

$$\text{hit ratio} = \frac{\text{no of hits}}{\text{total no of references}}$$

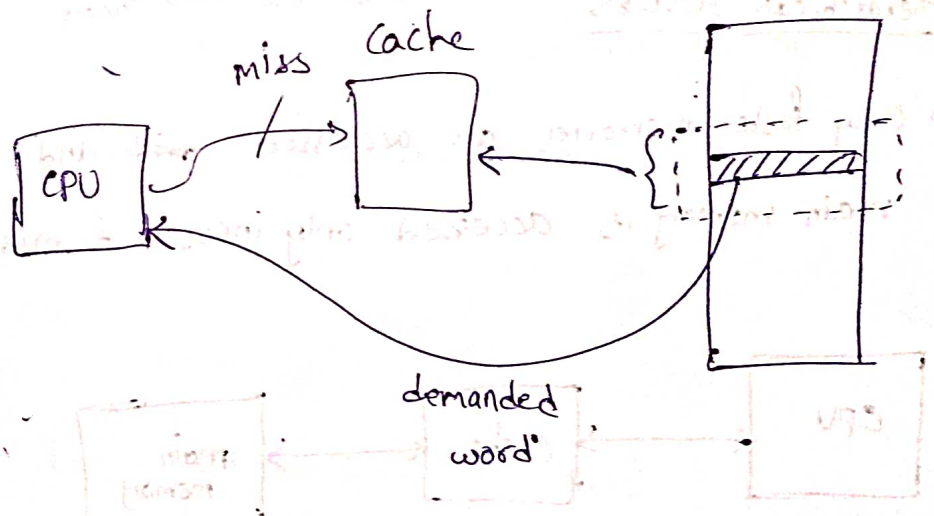
(H)

=) miss ratio = 1 − H

~~hit ratio = 1~~

hit rate = (hit ratio) * 100

→ when a cache miss occurs the block in which required word is present is transfered to the cache and at the same time the requested word is directly transfered to the CPU from memory



Average Memory access time:

$$T_{avg} = h * (\text{hit time}) + (1-h) * \text{miss time}$$

where h → hit ratio

Types of cache accesses:

1. Simultaneous Access:

Here request for cache & main memory are generated simultaneously.

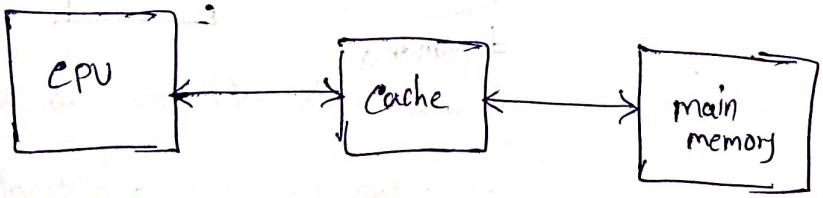
$$T_{avg} = (h * t_{cache}) + (1-h)\, t_{mm}$$

hit time $\rightarrow t_{cm}$

miss time $\rightarrow t_{mm}$

(ii) Hierarchical Access

→ Only faster memories are accessed ~~write~~ and ~~in case~~ main memory is accessed only incase of miss.



$$T_{avg} = (h * t_{cm}) + (1-h)(t_{cm} + t_{mm})$$

$$T_{avg} = t_{cm} + (1-h)\, t_{mm}$$

hit time $\rightarrow t_{cm}$

miss time $\rightarrow t_{cm} + t_{mm}$

Note:

→ If words like level or hierarchy are used the we go for hierarchical access..

Otherwise simultaneous access.

→ Also use simultaneous access if stmt like "ignore initial time req to check if miss or hit occured"

19/11/20

Tavg when locality of reference is ~~added~~, included:

Simultaneous:

$$T_{avg} = h * t_{cm} + (1-h)\, t_{block}$$

t_{block} → block access time for main memory

t_{block} = block size $* t_{mm}$

Hierarchical:

$$T_{avg} = t_{cm} + (1-h)(t_{block})$$

Tavg when block transfer included:

Simultaneous:

$$T_{avg} = H * t_{cm} + (1-H)(t_{block} + t_{cm})$$

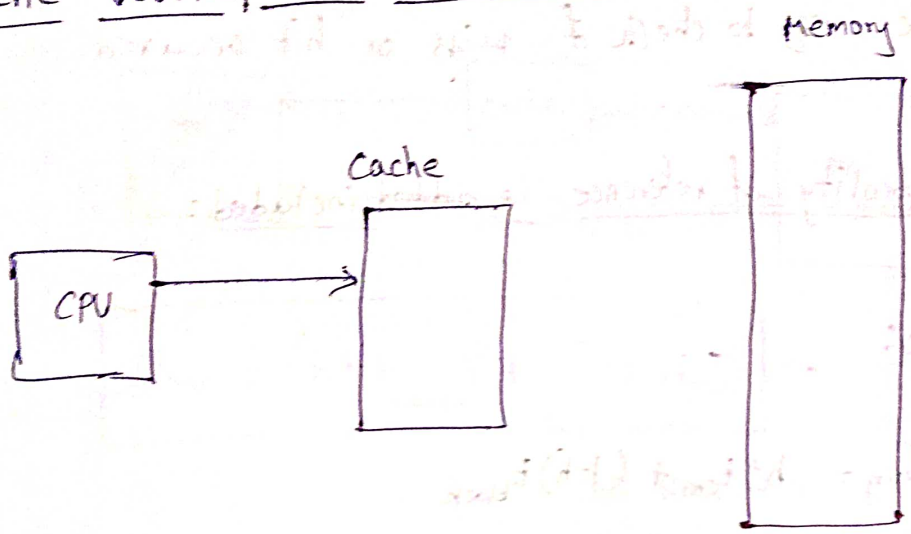
↳ To copy the block into CM:

Note: In one cache access 1 cache block can be accessed

Hierarchical:

$$T_{avg} = t_{cm} + (1-H)\left(t_{block} + t_{cm}\right)$$

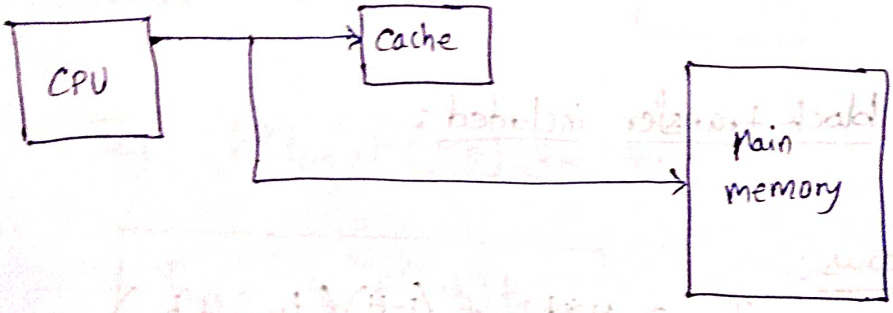
Cache Write/Write Propagation:



→ If CPU update cache memory content, then the original content in memory should also be updated.

This is done in two ways:

(i) Write through:



As soon as CPU updates cache memory contents, simultaneously main memory content is also updated.

i.e., irrespective of hit or miss, CPU will access MM.

→ Tavg for read remains same.

→ $(T_{avg})_{write} = \max\{t_{cm}, t_{mm}\} = t_{mm}$

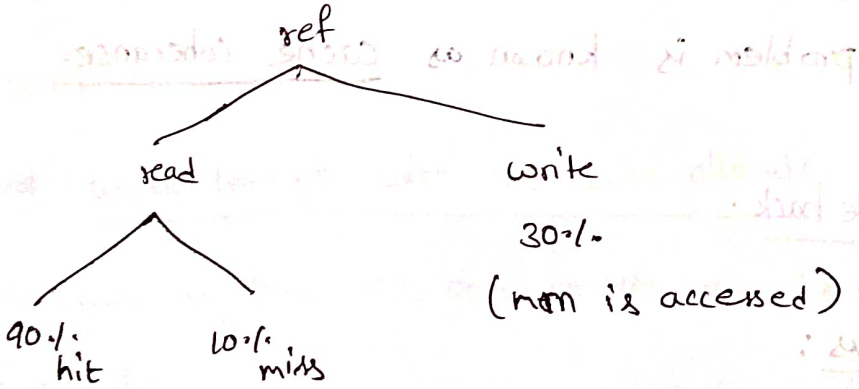
Effective hit ratio:

$$h_{eff} = \frac{\text{no of ref in which only cache is accessed}}{\text{total no of ref (both read \& write)}}$$

(Q24) Consider a write through cache with 90% hit rate.

If, 70% of references are for read and 30% are for write, then find effective hit ratio.

Sol:



∴ effective hit ratio = $0.7 * 0.9 = 0.63$

ii) Write Back:

→ CPU updates only in cache & when a block is replaced it is copied into MM.

Adv of write through:

→ MM always contains correct information (consistency)

Disadv of write through:

→ Takes more time.

Adv of write back:

→ Takes less time

Disadv of write back:

→ Inconsistency blw values in cache & main memory

→ Using DMA, IO devices may access wrong data.

This problem is known as cache coherance.

Tavg in write back:

Simultaneous:

Same Tavg for both read & write:

$$(T_{avg})_{read} = (T_{avg})_{write} = h * t_{cm} + (1-h)(t_{block} + t_{write\_back})$$

Hierarchical:

$$(T_{avg})_{read} = (T_{avg})_{write} = h * t_{cm} + (1-h)(t_{cm} + t_{block} + t_{write\_back})$$

$t_{wrik\_block} = x * t_{block}$

$x \rightarrow$ fraction of dirty block

20/11/20

* Based on the action we perform for write miss, we have 2 policies

(i) Write Allocate:

The block in mm is loaded into cache on write misse, followed by write-hit action

(ii) No write allocate:

The block is modified in the MM itself and is not loaded into the cache.

Note:

→ for better performance,

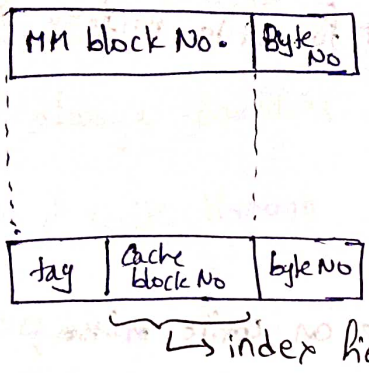
* we used write through with no write allocate.

It is because any how cpu goes for MM even in the case of hit or not.

* we used write back with write allocate.

Cache Mapping

Direct Mapping:



(tag information is also called metadata)

Cache block Num = (MM block no.) % (no of cache blocks)

no of bits in byte offset = $\log_2$(Block size)

no of bits in cache block no = $\log_2$(no of cache blocks)

no of tag bits = $\log_2\left(\frac{\text{Size of MM}}{\text{size of cache}}\right) = \log_2\left(\frac{\text{no of MM blocks}}{\text{no of cache blocks}}\right)$

Tag directory size = (no of blocks) * (no of tag bits) bits

(when no info abt V/I bit and other is given)

Cache initialization

→ when system is turned on, the content of tag is invalid.

To maitain information abot validity of content of a cache block we maintain a valid bit per each block.

→ when system is turned on, all valid bits are set to 0.

→ So whenever we index into a cache block, we also check if the valid bit is set or not.

Modified bit / Dirty bit

→ In write back cache we don't need to perform write back if the cache ~~is not~~ block content is not modified.

→ So we maintain this information using dirty bit

dirty bit = 1 ⇒ dirty block (modified)

dirty bit = 0 ⇒ clean block (unmodified)

Note:

→ Cache Controller is a hardware that ~~~~ controls cache accessing and has memory to store meta data (tags, valid bit, modified bit etc.)

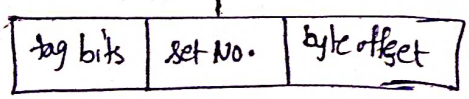
~~22/11/20~~
22/11/20

Set Associative Mapping

$$\text{set No.} = (\text{MM block No.}) \% (\text{no of sets})$$

→ no of sets in cache = $\frac{\text{no of blocks in cache}}{\text{associativity}}$

→ used for cache indexing.



→ $\text{no of tag bits} = \log_2\left(\frac{\text{no of MM blocks}}{\text{no of sets}}\right) = \log_2\left(\frac{\text{MM size}}{\text{cache size}}\right) + \log_2(\text{associativity})$

Eg: MM add : 32 bits

Cache size : 64 kB

block size : 32 B

4-way set associative.

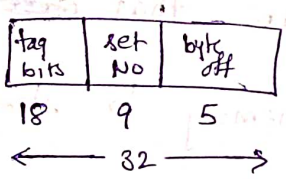
→ no of blocks = $\frac{2^{16}}{2^5} = 2^{11}$

no of sets = $\frac{2^{11}}{4} = 2^9$

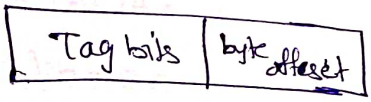
no of tag bits = $\log\left(\frac{\text{MM size}}{\text{cache size}}\right) + \log_2(\text{associativity})$

$= \log_2\left(2^{32-16}\right) + \log_2(4)$

$= 16 + 2 = 18$



Fully Associative Mapping



→ no of tag bits = $\log_2$(no. of MM blocks)

Hardware implementation of Mapping

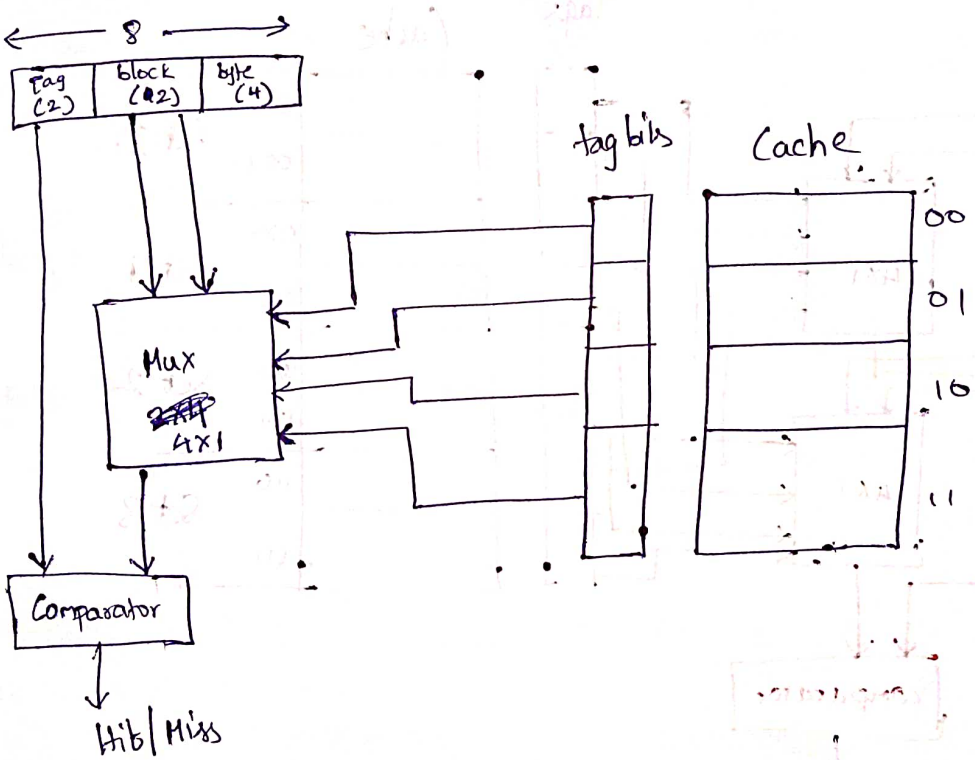
Direct Mapping:

Consider MM size : 2^8 B

Cache size : 2^6 B

Block size : 2^4 B

=> no of cache blocks = $2^6/2^4 = 2^2 = 4$ blocks



→ No of Muxes need = no of tag bits

→ size of Mux = $2^x \times 1$

~~no of bits in tag~~ 2^x → no of blocks

→ n-bit comparator

n → no of tag bits.

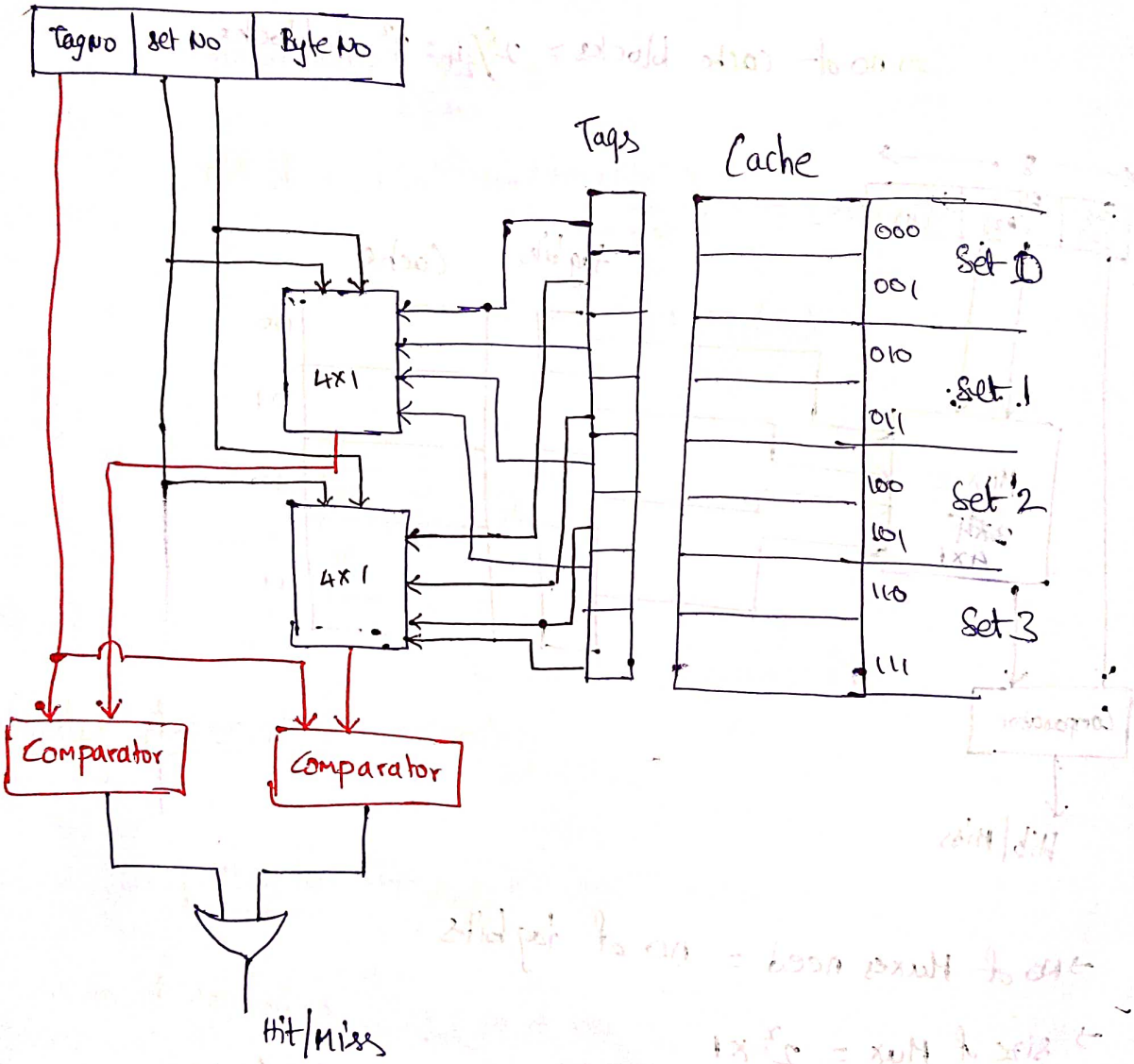
Hit latency:

Amount of time required for cache controller to determine whether hit or miss occured.

→ In direct mapping all the Muxes operate in parallel.

∴ Hit latency in direct mapping = Mux delay + Comparator.

Set Associative Mapping:



→ Size of Mux = $2^x \times 1$

↳ no of sets

→ no of Mux req = (no of tag bits) * (associativity)

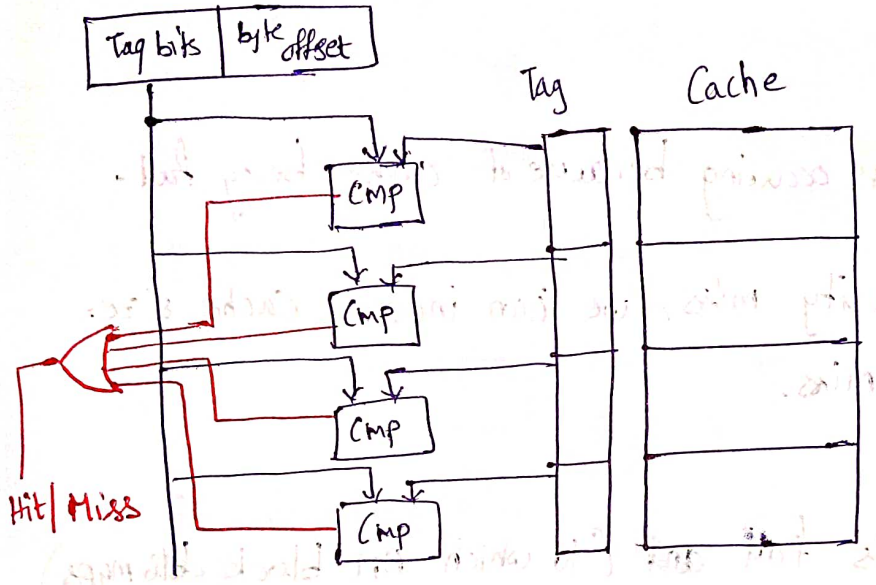
→ n-bit comparator ; n → no of tag bits

→ no of comparators = Associativity

→ k-i/p OR gate ; k → Associativity

Hit latency in set associative mapping = Mux delay + Comparator delay + OR gate delay

Fully Associative Mapping:



→ size of comparator = no of tag bits = $\log_2$ (no of MM blocks)

→ no of comparators req = no of cache blocks

→ size of OR gate = no of comparators = no of cache blocks

Block Replacement:

→ Direct Mapping doesn't req any replacement policy.

→ For set associative & fully associative we use below replacement policies

(i) FIFO

(ii) Optimal

(iii) LRU

Types of Cache misses:

(i) Cold / Compulsory miss:

This is miss due to first time reference of a cache block.

→ Compulsory miss can be reduced by increasing block size.

(ii) Capacity Miss:

→ It is cache miss occuring because of cache being full.

→ To reduce capacity miss, we can increase cache size.

→ It is not cold miss.

(iii) Conflict Miss:

If cache set is full (to which MM block maps) and hence miss occurs due to tag mismatch.

→ Conflict miss is not cold miss & not capacity miss.

→ Conflict miss can be reduced by increasing associativity.

Note:

→ When asked to check what kind of miss a given miss is, check only in the below order.

i) Cold miss

ii) Capacity miss

iii) Conflict miss

Note: Conflict miss is never possible in fully associative mapping.

a cache block.
lock size.

being full.

cache size.

block maps)

ty miss.

g associativity.

iven miss is,

mapping.

Eg: Consider

no of blocks in cache = 4

let cache be 2-way set associative

and let ~~part~~ policy be LRU.

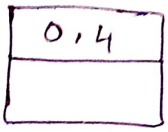
Consider ref string: 0, 4, 0, 8, 0, 4, 1, 3, 1, 5, 1, 3

0: ~~Conflict~~

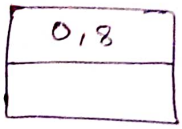
Compulsory miss

4: Compulsory miss

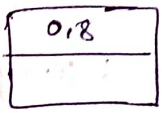
0: Hit



8: Compulsory Miss



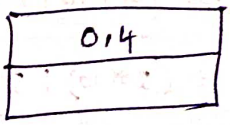
0: Hit



* 4: Here cache is not full.

but set to which 4 map is full.

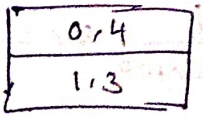
∴ Conflict miss



1: Cold miss

3: Cold miss

1: hit



5 : Compulsory miss

0,4
1,5

1 : hit

* 3 : Here cache is full & set is also full
still we say it is capacity miss

∴ Capacity miss

0,4
1,3

Cache miss penalty

It is time required to bring a missed block from main memory to cache.

Assume

- Cycles required to send address to MM : 1 cycle
Cycles req to access 1 MM cell (word) : 10 cycles
Cycles req to transfer 1 cell data to cache : 1 cycle

Cache block size	MM cell size	Miss penalty
4 bytes	1 byte	(1) + (4 * 10) + 4 = 45 cycles
4 bytes	2 bytes	1 + (2 * 10) + 2 = 23 cycles
4 bytes	4 bytes	1 + (1 * 10) + 1 = 12 cycles

→ Address is sent only once. From then accessing of entire block is done.

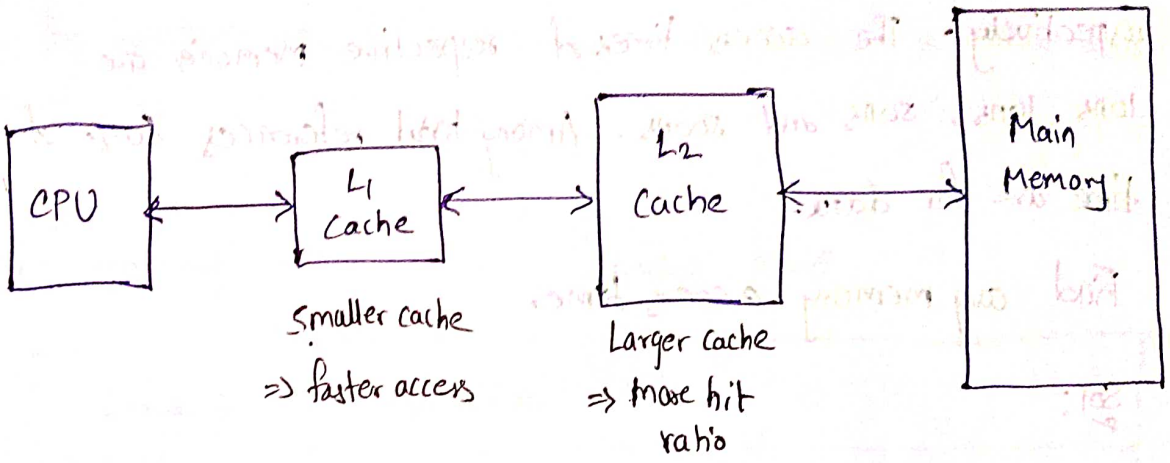
→ If CPU runs on 500 MHz clock then

$$45 \text{ cycles} = 45 \times \frac{1}{500 \times 10^6} \text{ sec} = 9 \times 10^{-8} = 90 \text{ ns}$$

Goals of using cache memory.

→ Minimize access time. ⟶ (smaller cache) } Multilevel Cache
→ Maximize hit rate. ⟶ (Larger cache) }
→ Minimize miss penalty.

Multilevel Cache:



Tavg in Multilevel cache:

(i) Simultaneous access:

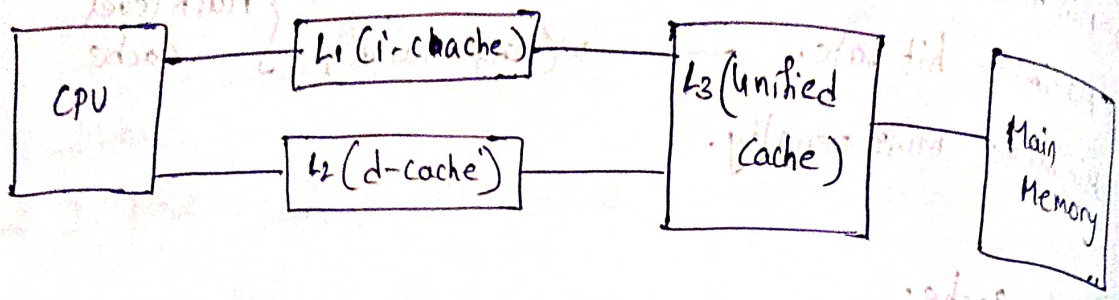
$$T_{avg} = h_1 * t_1 + (1-h_1)\left[h_2 * t_2 + (1-h_2)\, t_{mm}\right]$$

(ii) Hierarchical access:

$$T_{avg} = h_1 * t_1 + (1-h_1)\left[h_2 * (t_1+t_2) + (1-h_2)(t_1+t_2+t_3)\right]$$

$$\cong t_1 + (1-h_1)\, t_2 + (1-h_1)(1-h_2)\, t_3$$

Eg: The multilevel memory hierarchy is given:



The hits ratios of L_1, L_2, L_3 and MM are 0.8, 0.9, 0.95 and 1.0 respectively. The access times of respective memories are 10ns, 50ns, 500ns and 5000ns. Among total references 60% of time are for data.

Find avg memory-access time.

Sol:

avg access time for instruction

$$= t_1 + (1-h_1)\, t_2 + (1-h_1)(1-h_2)\, t_3$$

$$= 10 + (1-0.8)(50) + (1-0.8)(1-0.95)(500)$$

$$= 10 + 10 + (0.2)(0.05)(500)$$

$$= 10 + 10 + 5$$

$$= 25$$

avg access time for data

$$= 10 + (1-0.9)(50) + (1-0.9)(1-0.95)(500)$$

$$= 10 + 5 + 2.5$$

$$= 17.5$$

avg memory access time $= 0.6\,(17.5) + 0.4\,(25)$

Cache Inclusion Policies:

There are two types of policy

(i) inclusion policy

(ii) Exclusion policy

(i) Inclusion policy

The content of L_1 cache is present in L_2 cache also.

i.e., L_2 is inclusive of L_1.

For read operations, consider below table

Scenario	Operation
Hit in L_1	CPU gets content from L_1
Miss in L_1 & Hit in L_2	Copy a block from L_2 to L_1 & replaced block is copied only to memory and no participation of L_2
Miss in L_1 & Miss in L_2 & Hit in MM	Copy the block from MM to L_1 & L_2 both → If a block is replaced from L_2 then send back invalidation to L_1 (i.e., reset valid bit in L_1)

(ii) Exclusion Policy:

Content of L_1 cache need not to be present in L_2 cache.

→ For read operations, consider below table.

Scenario	Operation
Hit in L_1	CPU gets contents from L_1
Miss in L_1 & Hit in L_2	Move a block from L_2 to L_1, the replaced block of L_1 is moved to L_2.
Miss in L_1 & Miss in L_2	Copy the block from MM to L_1 only. The replaced block (if any) is moved to L_2 cache.

→ In exclusion policy, L_2 is called victim cache, since it is being filled by victim blocks (replaced blocks) of L_1.

(iii) Value inclusion policy:

→ In inclusion policy, all the blocks in L_1 are present in L_2 also, but it is not needed that values are same in L_1 & L_2.

→ Values inclusion policy ensures that value of blocks in L_1 is same as value of blocks in L_2.

24/11/20

(Q25) what hit rate is required to reduce the effective memory access time from 150 ns to 42 ns if cache access time is 30 ns.

Sol:

There is no word like level or hierarchy.

So go for simultaneous access

$$42 = h(30) + (1-h)(150)$$

$$\Rightarrow h = 0.9$$

hit rate = 90%

(Q26) A computer system contains a cache. Uncached memory access takes a times longer than access to cache. If cache has hit ratio 0.8, The ratio cache memory access time to uncached memory access time is ____

Sol:

let uncached access time = m

i.e mem access time

~~avg access time~~ ⇒ cache access time = $\frac{m}{9}$

$$\therefore \frac{0.8\left(\frac{m}{9}\right) + 0.2(m)}{m} = \frac{\frac{8}{9} + 2}{10} = \frac{26}{90} = \frac{13}{45}$$

(Q27) In a two-level hierarchy, the top level has an access time of 20ns and the bottom level has an access time of 80ns, the hit rate on the top level is 90%. If the block size of cache is 8 bytes then avg mem access time is _____

(Note: Consider the system uses locality of ref)

Sol:

Here we need to use hierarchical access

$T_{avg} = 0.9(20) + 0.1(20 + \underbrace{8*80}_{\text{block access time}})$

$= 18 + 0.1(660)$

$= 18 + 66 = 84$ ns

(Q28) 2-level hierarchy; Cache access time = 15ns; MM access = 110ns

hit rate is 90%. Block size = 16 bytes;

Find avg mem access time including miss penalty

Sol:

Miss penalty is time to bring block from MM to cache

$T_{avg} = 0.9(15) + 0.1(15 + \underbrace{110*16}_{\text{block access}} + \underbrace{15}_{\text{transfering block to cache}})$

$= 13.5 + 179 = 192.5$ ns

(Q29) Consider a memory hierarchy with a write back cache. The cache has an access time of 25 ns. 90% of all the memory accesses are found in the cache itself. The main memory access time is 300 ns. The 20% of the cache blocks are dirty. The cache block size is 4 bytes. Find avg mem access time.

Sol:

$$t_{avg} = h * t_{cm} + (1-h)(t_{cm} + t_{block} + t_{write\_back})$$

$$= 0.9(25) + 0.1(25 + 4*300 + 0.2*4*300)$$

$$= 22.5 + 0.1(25 + 1200 + 240)$$

$$= 169 \text{ ns}$$

(Q30) Consider direct mapped cache with

32-bit architecture

MM size = 4GB

Cache size = 512kB

block size = 16 words

Find

address division.

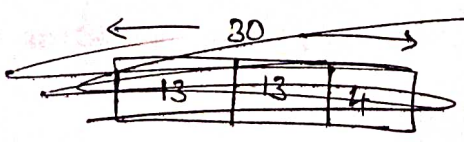
Sol:

~~word size = 4 words~~

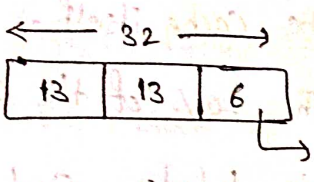
~~MM size = 2^{32} bytes = 2^{30} words~~

~~Cache size = 2^{19} bytes = 2^{17} words~~

~~no. of MM blocks = 2^{26} / no. of cache blocks = 2^{13}~~



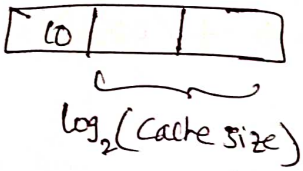
Since no information is given we consider @ memory is byte addressable.



block size = $16 \times 4 = 2^6$

(Q31) Consider a direct mapped cache of size 256 MB.
No of tag bits = 10. Find max size of main memory supported in the system is ____

Sol:



$= 28$ bits

$\therefore$ size of PA = 38 bits

$\therefore$ size of MM = 256 GB

(Q32) MM size : 2^{36} bytes

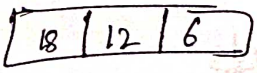
Cache size : 2^{18} bytes

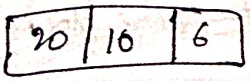
block size : 2^6 bytes

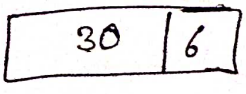
~~Find no of tag bits~~ a

Find add. division in direct, 4-way set ass, full associative

Sol:







27/11/20

Q33) Consider, direct mapped cache.

32 bit address of main memory.

Block size: 32 bytes.

no of cache blocks: 256

In which block of cache should we look for each of the following addresses.

1. 1A28C012
2. FFFF00FF
3. 12345678
4. C109D532

Sol:

Ans: 1 - 0
2 - 7
3 - 179
4 - 169

Note:

CPU require data bus only when it needs to access main memory. To access cache memory CPU uses internal bus.

** Q34) * Cache hit ratio = 0.9

block size: 4 words & use locality of reference.

CPU sends reference to the cache at a rate of 10^8 words per second.

20% of the above ref are write.

The bus reads or writes a single word at a time.

Assume at any one time, 20% of the block frames in the cache have been modified.

The cache uses write allocate on a write miss.

The percentage of the bus bandwidth used on average in

1. write through cache: _____

2. Write back cache: _____

Sol:

1. write through cache:

Operation	Bus bandwidth
Read hit	0
Read miss	$0.1 * 0.8 * 10^8 * 4$
write hit	$0.9 * 0.2 * 10^8$
write miss	$0.1 * 0.2 * 10^8 * (1+4)$

total: $(0.32 + 0.18 + 0.1) 10^8$

$= 0.51 \times 10^8$

∴ band width used = 51%

2- write back Cache:

Read hit.	0
Read miss	$0.1 * 0.8 * 10^8 * (4 + 0.2 * 4)$
write hit	0
write miss	$0.1 * 0.2 * 10^8 * (4 + 0.2 * 4)$

total: $(0.384 + 0.096) * 10^8$

frames in the

ics.

n average in

$= 0.48 \times 10^8$

$\therefore$ band width used = 48%.

Q35 Consider a cache with 2^{13} blocks of size 32 bytes each. The CPU generates 32-bit address. Cache controller has 1 valid bit, 1 modified bit and tag bits. Size of cache controller memory is 18k bytes. Cache is organized as k-way set associative. Find max value of k to utilize the controller memory maximum is _____?

Ans: 8

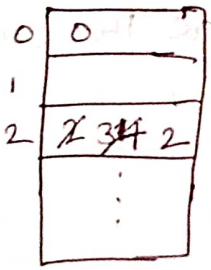
Q36 A computer system use 16 bit memory addresses. It has 2k bytes cache. organized in direct mapped manner. Block size: 64 bytes. CPU generates below references (decimal values)

28, 144, 2176, 2180, 128, 2176.

Assuming cache is initially empty find no of misses.

a) 3 b) 4 c) 5 d) 9

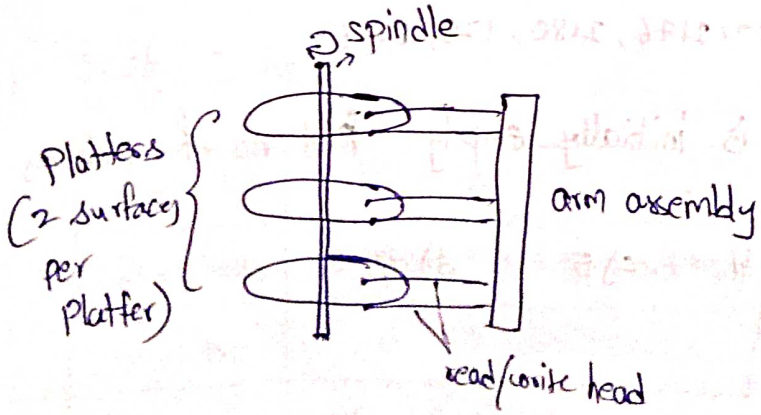
MM address	MM block (-/64)	Cache block (-/32)	Hit / Miss
28	0	0	Miss
144	2	2	Miss
2176	34	2	Miss
2180	34	2	Hit
128	2	2	Miss
2176	34	2	Miss

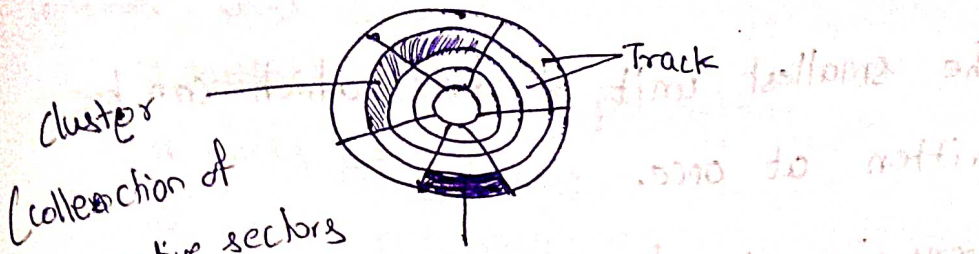


∴ 5 Misses

28/11/20

Magnetic Disk





Disk Capacity

Data is stored on disk in 2 ways:

i) Constant sector capacity
ii) Variable sector capacity

Constant Sector Capacity: (Default consideration)

→ All sectors store same amount of data.

→ Thus density of storage is lesser as we move to the outer tracks.

Disk capacity = 2 * no of platters * no of tracks * no of sectors per track * sector capacity

→ Here angular velocity is constant.

Variable sector capacity:

→ Storage density is constant in all the tracks.

→ Thus sector capacity varies from track to track.

Disk capacity = 2 * no of platters * surface capacity.

(Here linear velocity is constant)

Note:

→ Sector is the smallest unit of disk which can be read or written at once.

i.e., for every access to disk, an entire sector is accessed.

→ Sector is the least addressable unit of disk.

Disk Access Time:

→ Disk access time is time required for accessing 1 sector.

> Disk access time = seek time + Rotational latency + transfer time (for 1 sector) + additional delay given in question.

Seek time: time req to position the arm over the desired track

Rotational Latency: time req to rotate desired sector under R/w head

Transfer time: time req to read or write 1 sector.

Note:

→ In one rotation time, 1 track can be transfered.

$$\therefore \text{transfer time of 1 sector} = \frac{\text{1 rotation time}}{\text{no of sectors per track}}$$

Eg: Consider disk with

16 platters; 2 surfaces per platters;

1k tracks per surface;

2k sectors per track;

2048 bytes per sector;

Disk rotates with 3000 rpm

Seek time : 10 ms

a) Find capacity of disk

$$2 * 16 * \cancel{1000}^{2^{10}} * \cancel{2000}^{2^{11}} * 2048\ B$$

$$= 2^{37}\ Bytes$$

$$= 128\ GB$$

b) no of bits required for addressing the disk?

$$= \log_2 (\text{no of sectors})$$

$$= \log_2 \left(\frac{2^{37}}{2^{11}}\right)$$

$$= 26 \text{ bits}$$

c) Find disk access time.

seek time + rot. latency + transfer time

60 sec → 3000 rot

1 rot → $\frac{60}{3000}$ sec $= \frac{1}{50}$ sec $= 20$ ms

transfer time of 1 sector $= \frac{1}{60} \times \frac{1}{2^{11}} = \frac{20}{2^{11}}$ ms

$= 10 + \frac{1}{120} + 1$

$= 10 + \frac{20}{2} + \frac{20}{2^{11}}$

$= 10 + 10 + 0.01$

$= 20.01$ ms

d) Find disk transfer rate.

in 1 rotation, 1 track is transfered

20 ms $\rightarrow 2^{11} * 2^{11}$ bytes

20 ms $\rightarrow 2^{22}$ bytes

1 sec $\rightarrow \frac{2^{22}}{20 \times 10^{-3}}$

$= \frac{2^{22} \times 10^2}{2}$

$= 2^{21} \times 10^2$

$= 200$ MBPS

$= 200$ Mega Bytes Per Second.

e) If the disk is used in cycle stealing mode of DMA such that whenever 64-bit word is available, it will be transfered in 16 ns. Find % of time CPU is blocked?

% of time CPU blocke = $\frac{\text{transfer time}}{\text{preparation time}}$

1 sector prep time = 0.01 ms

1 byte = $\frac{0.01}{2048}$ ms = $\frac{10^4}{2\times10^3}$ ns = 5 ns

8 bytes = 40 ns

% time CPU blocked = $\frac{16}{40} \times 100 = 40\%$

Multiple Sectors Access time:

(i) Sequential:

Here sectors are stored on the same track consecutively.

Transfer time of n sectors = seek time + Rotational latency + n* transfer time of 1 sector

(ii) Random:

Here each sector is stored in a random location.

Transfer time of n sectors = n* (seek time + rot. latency + transfer time (for 1 sector))

Cyclinder: Collection of tracks of same radius from all surfaces

→ Data is stored cyclinder wise.

→ ~~this~~

→ ~~Cylin~~ → Using the concept of cylinder we reduce seek time and thus improve speed

→ No of ~~cyl~~ cylinders = no of tracks per surface.

Note:

A sector is addressed as <c, h, s>

c → cylinder number

h → surface number

s → sector number in track.

⇒ sector number = (c * no of surfaces * sectors/track)

+ (h * sectors/track) + s

(Assuming starting address is <0,0,0>)

→ Silly for given sector number 'n' we can find corresponding <c, h, s>

c = n // (sectors/cylinder) → integer division

h = [n % (sector/cylinder)] / (sec/track)

~~s = {[n % (sectors/cylinder)] / (tracks/surface)} %~~

s = [n % (sector/cylinder)] % (sec/track)

Parallel Processing:

→ It is simultaneous processing of data.

3 types of parallel processing

(i) vector processing
(ii) Array processing } not there in the syllabus.
(iii) Pipelining

Flynn's Classification of Computers:

(i) SISD (Single Instruction stream, Single data stream):

At a time one instⁿ is fetched and executed.

Eg: Von neuman computers

(ii) SIMD (Single Instⁿ stream, Multiple data stream):

→ One instⁿ is fetched at a time, but multiple instⁿs are executed at a time.

Eg: Pipelined processor.

(iii) MISD (Multiple Instⁿ stream, Single Data stream)

More instⁿs are fetched at a time but only one instⁿ is executed at a time.

→ This is not practical as fetch multiple instⁿs but executing only one at a time is of no use.

(iv) MIMD (Multiple Insth stream, Multiple data stream)

→ Here more than one insth is fetched at a time & more than one insth is executed at a time.

→ This type systems have multiple pipelines

Eg: Superscalar Computers.

→ This systems have Insth level parallelism (ILP)

Pipelining:

→ Pipelining is useful when same processing is applied over multiple inputs.

→ Pipelining decomposes a sequential process into sub-operations.

→ Suboperations are performed in stage (segment)

→ Each segment works parallely

Eg:

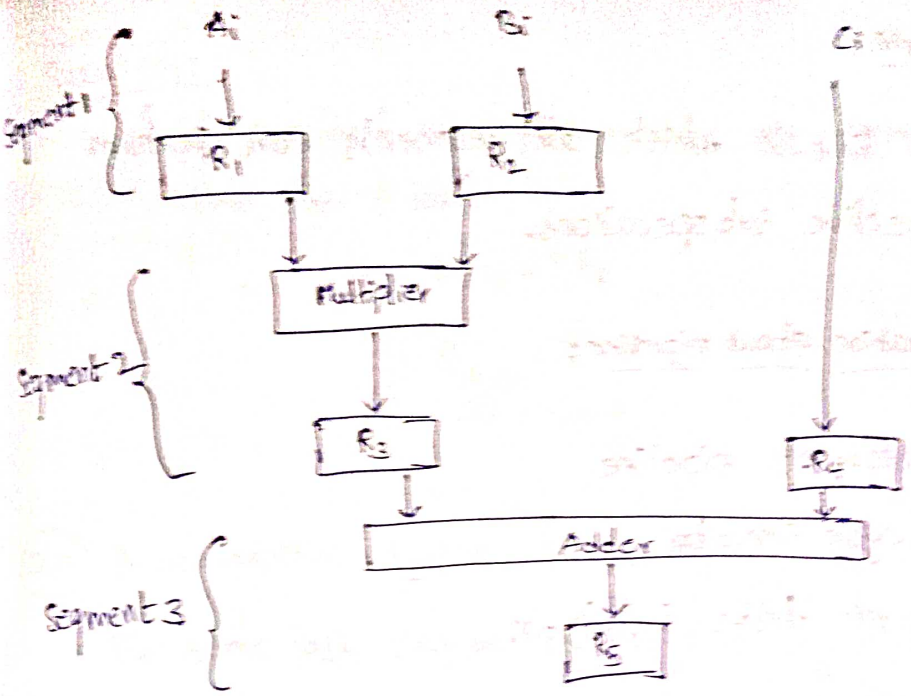
Consider $A_i * B_i + C_i$ for $i = 1$ to 5

Sub operations:

Seg1: $R_1 \leftarrow A_i$, $R_2 \leftarrow B_i$

seg2: $R_3 \leftarrow R_1 * R_2$, $R_4 \leftarrow C_i$

Seg3: $R_5 \leftarrow R_3 + R_4$



Clock Pulse	Segment 1		Segment 2		Segment 3
	R_1	R_2	R_3	R_4	R_5
1	A_1	B_1			
2	A_2	B_2	$A_1 * B_1$	C_1	
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6			$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7					$A_5 * B_5 + C_5$

no of cycles req with pipelining = 7

no of cycles req without pipeling (sequential) = $5 * 3 = 15$

Pipeline Cycle time:

It is minimum time in which all segments can perform all their respective suboperations.

General consideration about pipeline:

Consider k segment pipeline

clock cycle time = t_p

let no of tasks (instrns) = n

time req to perform 1 task = $k * t_p$

(i.e., o/p of 1st task)

time req to perform next n-1 tasks = $(n-1)\, t_p$

$\therefore$ total time req for n tasks = $(k + n-1)\, t_p$

~~total time~~

Note:

performance of pipeline is given by speed up ratio

$$\text{speed up ratio} = \frac{\text{non-pipeline time}}{\text{pipeline time}}$$

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

$t_n \rightarrow$ time req for each task in sequential processing.

if ~~n~~ n is large

$$S_{ideal} = \frac{t_n}{t_p}$$

→ This is ideal case.

Special case:

If pipeline & non-~~pipe~~pipeline req same amount of time for 1 task

i.e., $t_n = k * t_p$

$$S_{ideal} = k$$

(B37) A non-pipeline system takes 50 ns to process a task. The same task can be processed in a 6-segment pipeline with a clock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the max speed up that can be achieved?

Sol:

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

$$= \frac{100 * 50}{(6+99) * 10} = 4.76$$

$$S_{max} = \frac{t_n}{t_p} = \frac{50}{10} = 5$$

29/11/20

29/11/20

Synchronous Pipeline:

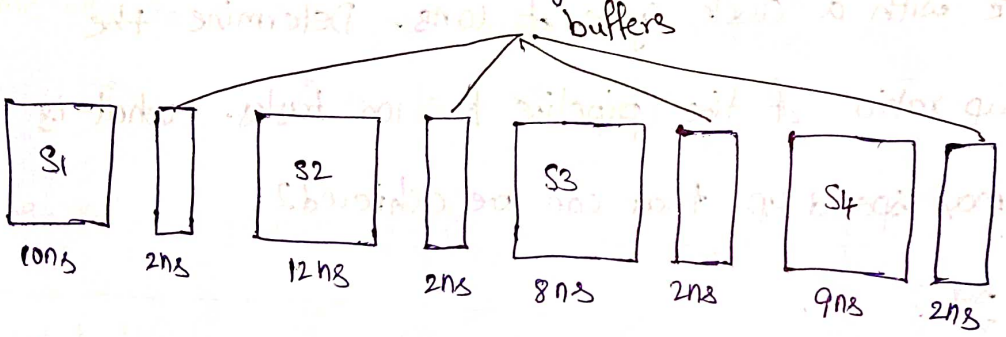
→ Each segment / stage may have different delays.

So we need to synchronize them.

Cycle time is choosen as max(stage delay)

→ o/p of one stage is i/p to next stage.

Since delays are different, we need intermediate buffers b/w stage to store o/p of a stage.



Clock period = max(seg. delay) + reg. delay

= 12 + 2 = 14 ns

But in a ~~non-pipe~~ non-pipeline system,

time for 1 task = 10 + 12 + 8 + 9 (seq. processing doesn't need reg delays)

= 39 ns

Latency:

It ~~time~~ amount of time after which machine takes next input.

In above example,

latency in non-pipeline = t_n = 39 ns

latency in pipeline = Cycle time, t_p = 14 ns

Throughput:

It is no of i/ps processed per unit time.

In pipelined system,

time for n tasks = $(k+n-1)t_p$

$\Rightarrow$ Throughtput = $\frac{n}{(k+n-1)t_p}$

for ideal case

throughput = $\frac{n}{n*t_p} = \frac{1}{t_p}$

Q38) 5-stage pipeline; cycle time = 15 ns; find processing time for 500 tasks.

Sol:

time = $(5+(500-1))15 = 504*15 = 7560$ ns

Q39) 6-segment pipeline;

stage delays → 20, 26, 21, 21, 24, 28 (in ns)

find processing time for 1000 tasks & speed up

Sol:

Cycle time = max(delays) = 28

$\Rightarrow$ Processing time = $(6+1000-1)*28$

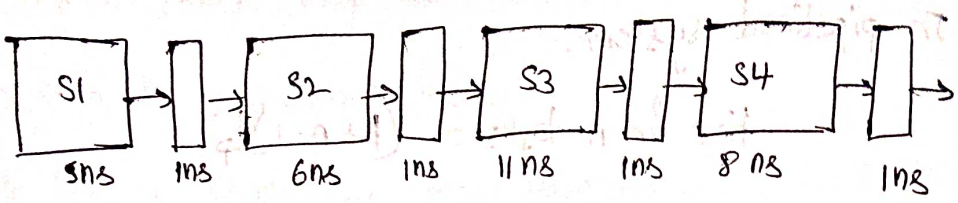
$= 1005*28$

$= 28,140$

speed up = $\frac{(20+26+21+21+24+28)(1000)}{28140} = \frac{149000}{28140} = 4.97$

$$\text{max speed up} = \frac{t_n}{t_p} = \frac{20+26+21+21+24+28}{28} = \frac{140}{28} = 5$$

Q40 / 4-11



Find approximate speed up under ideal conditions.

Sol:

pipeline cycle time = max(5, 6, 11, 8) + 1

$t_p = 12$

execution time of 1 task in non-pipeline system = 5 + 6 + 11 + 8 = 30

$$\text{ideal speed up} = \frac{t_n}{t_p} = \frac{30}{12} = 2.5$$

Note:

For n operations,

time under pipeline system = $(k+n-1)t_p$

$\Rightarrow$ avg time for 1 inst^n = $\frac{(k+n-1)t_p}{n}$

$\Rightarrow CPI = \frac{k+n-1}{n}$

under ideal conditions, $CPI_{ideal} = 1$

time per operation = t_p

(Sol) stage delays of 4-stage pipeline are 800, 500, 400 and 300.

If it is upgraded to 5-stage pipeline by replacing 1st stage by 2 stages of delays 600 and 350 then the throughput increase of the pipeline is _____ percent.

Sol:

4-stage:

$t_p = 800$

time for 1 instn = 800 ps $\Rightarrow$ throughput = $\frac{1}{800}$

5-stage:

$t_p = 600$

time for 1 instn = 600 ps $\Rightarrow$ throughput = $\frac{1}{600}$

$\Rightarrow$ % increase = $\frac{800-600}{600} \times 100 = 33.33\%$

throughput increase = $\frac{\frac{1}{600} - \frac{1}{800}}{\frac{1}{800}} \times 100$

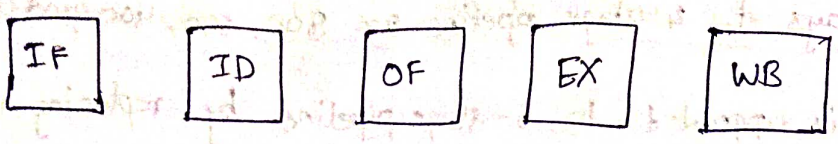
$= \frac{\frac{800-600}{800*600}}{\frac{1}{800}} \times 100 = \frac{200}{600} \times 100 = 33.33\%$

Instruction Pipeline:

→ It is pipeline implemented on instruction cycle.

Assume we have 5 stages

- Instn fetch (IF)
- Instn decode & Address Calculation (ID)
- Operand fetch (OF)
- Execution (EX)
- Write back. (WB)



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I_1	IF	ID	OF	EX	WB										
I_2		IF	ID	OF	EX	WB									
I_3			IF	ID	OF	EX	WB								
I_4				IF	ID	OF	EX	WB							
I_5					IF	ID	OF	EX	WB						
I_6						IF	ID	OF	EX	WB					
I_7							IF	-	-						
I_8															
I_9										IF	ID	OF	EX	WB	
I_{10}											IF	ID	OF	EX	WB

stall cycles/bubbles

Assume in cycle 7,

I_6 is decoded and found to be branch inst^n.

→ whenever branch inst^n is found, ~~no execution~~ pipeline is emplied. i.e., no other inst^{n}s next to branch are executed.

~~Assum~~ I_6

Note:

→ The target inst^n of branch is found after execution stage.

Here in this example, target inst^n is found at the end of 9th cycle.

Let target inst^n be I_9. so IF of I_9 is done in 10th cycle.

Note:

→ In the previous example, even if branch condition of I_6 evaluates to false, we still need to fetch I_7 again in 10th cycle.

→ In this example, no of inst^n's executed = 8

no of cycle req = 15

But no of cycles that actually needed = 5+8-1 = 12

Here we req 3 extra cycles
↳ (15-12)
i.e., 3 stall cycles

→ Here result of target inst^n is found at the end of 4th stage

∴ no of stall cycles = 4-1 = 3

i.e., If branch target is evaluated in i^{th} stage, then

no of stall cycles = i-1

Q42

Q42
4-13

Consider an inst^n pipeline ...

Sol:

t_p = 10+1 = 11 ns

for each branch, no of stall cycles = 4-1 = 3

no of inst^n executed = 12-4 = 8

no of cycle = 5+8-1 = 12

1 branch instn ⇒ 3 stall cycles

∴ total cycles = 12+3 = 15

∴ time = 15 × 11 = 165 ns

Pipeline Hazards:

→ Hazard is situation that prevents the next instn from ~~be exe~~ being executed in the exeepected cycle.

∴ Hazards lead to stall cycles.

3 types of hazards:

(i) Structural Hazard / Resource Conflict

(ii) Data Hazard / Data Dependency

(iii) Control Hazard / Branch Difficulty

(i) Structural Hazard / Resource Conflict:

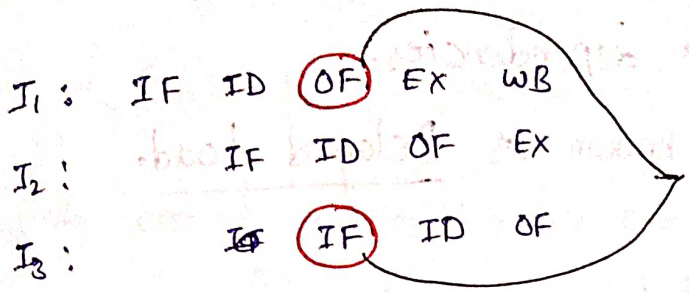
→ This is situation when 2 different inputs (instns) ~~segments~~ try to use same resource at same time.

Eg: Assume MUL req 3 cycle for execution and ADD req 2 cycles for execution

MUL :	IF	ID	OF	EX	EX	EX	WB	
ADD :		IF	ID	OF	-	-	EX	WB

(Assuming that MUL & ADD using same operands)

→ Also consider below case



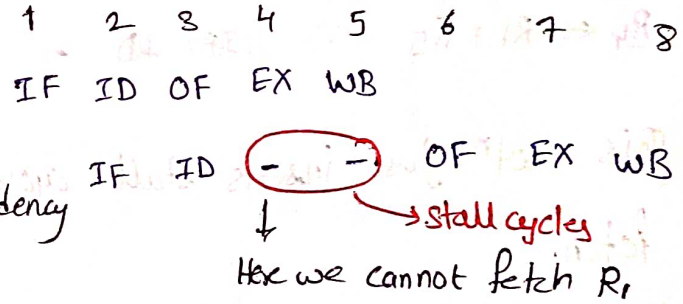
So in this case we use two cache memories
i.e., instⁿ cache & data cache
and thus avoid the conflict.

(i) Data Hazard | Data Dependency

This is situation where result of an instⁿ is used as input by next instⁿ.

Eg: Consider

i: $R_1 \leftarrow R_2 * R_3$
i+1: $R_4 \leftarrow R_1 + R_5$ — Data dependency



→ General pipeline hardware cannot detect data dependency

Soln for data dependency

- S/w soln
  - → This soln is given by compiler
- H/w soln
  - H/w interlock
  - Operand Forwarding

→ In slw soln, compiler generates insth such that there is no data dependencies.

→ This soln is known as Delayed Load.

Delayed Load:

$R_1 \leftarrow R_2 * R_3$

$R_5 \leftarrow R_1 + R_4$

Here we have seen that there are 2 stall cycles.

So compiler inserts 2 insths blw the two insths.

→ So we insert either independent insths or NOP insths.
(This insertion is done by compiler)

H/w Interlock:

$R_1 \leftarrow R_2 * R_3$	IF	ID	OF	EX	WB	
$R_4 \leftarrow R_1 + R_5$		IF	ID	-	-	OF EX WB

→ This soln just inserts stall cycles by locking operand fetch.

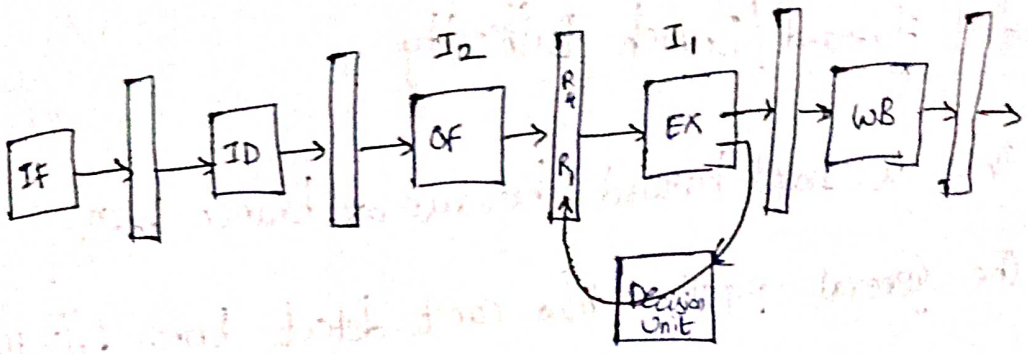
Operand Forwarding / Bypassing:

	1	2	3	4	5	6	
$R_1 \leftarrow R_2 + R_3$	IF	ID	OF	EX	WB		→ operand forwarding
$R_5 \leftarrow R_1 * R_4$		IF	ID	OF	EX	WB	

At end of 3rd cycle

I_2 fetch R_1 & R_4 (However R_1's value is wrong)

But at the end of 4th cycle, I_1's execution phase finishes and it will forward the operand to execution phase of I_1.



→ However this soln is expensive

→ Decision unit performs operand forwarding only if there is data dependency.

Note:

→ Operanding Forwarding can be used only if there is ALU to ALU data dependency.

i.e. ~~for ex~~

→ For other dependencies operand forwarding doesn't work

i.e., For example

$R_1 \leftarrow M[add]$
$R_4 \leftarrow R_1 * R_5$

IF ID OF EX WB
IF ID - OF EX WB

Here dependency is not from ALU to ALU

∴ we cannot use operand forwarding.

↓ possible when loading can be done in execution phase

Similarly for below ~~seen~~ scenario also operand forwarding doesn't work.

$R_4 \leftarrow R_1 * R_5$ IF ID OF EX WB

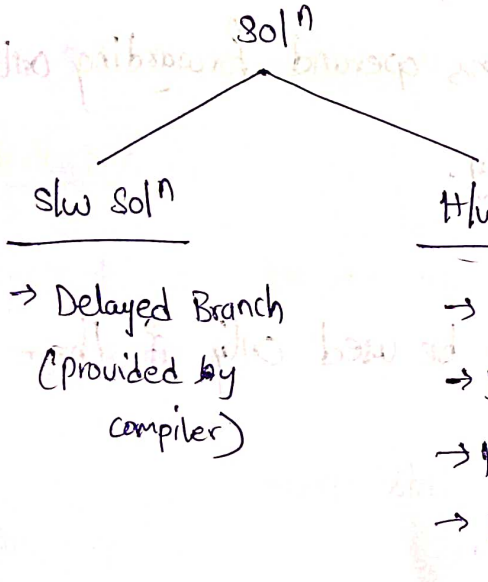
$M[add] \leftarrow R_4$ IF ID - - OF EX WB

So in these cases stall cycles must exist.

Control Hazard / Branch Difficulty

→ This is ~~hard~~ hazard because of branch instn.

→ The general pipeline H/w can't detect branch difficulty.



S/w Soln

→ Delayed Branch (provided by compiler)

H/w Soln

→ Prefetch target instn
→ Branch Prediction
→ Loop Buffer
→ Branch Target Buffer.

Delayed Branch:

→ ~~Based on no of stall cycles required compiler inserts those many no of NOP instns.~~

→ Some independent instn is put in delayed slot by compiler, if no such instn is found NOP is placed.

Prefetch Target Instruction

→ Here when branch insth is detected, two pipelines will be initiated.

One pipeline is started assuming branch not taken

Other pipeline is started assuming branch is taken

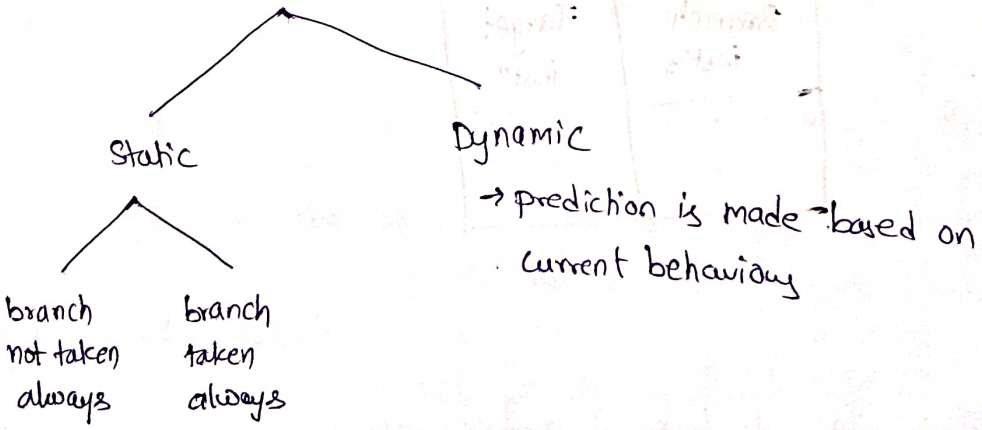
→ After finding the correct branch target, other pipeline will be discarded.

Branch Prediction:

→ Here HW makes prediction whether branch occurs or doesn't and thus starts appropriate insth.

→ If the prediction is found to be wrong, then the insth is rolled back.

2 types of prediction



Loop Buffer:

Consider a loop with 2 instⁿs that executes 100 times

```
loop:
{
    I1;
    I2;
}
```

H/w detects that loop needs to execute 100 times. After this the two instⁿs are stored in loop buffer and they are executed 100 times.

Branch Target Buffer:

→ This buffer is used with branch prediction H/w.

→ This buffer contains all branch instⁿs and their corresponding previous target instⁿs.

→ Based the content of this buffer prediction is made.

Branch instⁿs	Target instⁿ
⋮	⋮

Data Hazard Classification

→ Assume there are 2 instⁿ i & j such that i executes before j.

RAW (Read After Write)

→ j reads source value before i writes it. So j gets incorrect (old) value

```
i: R1 ← R2 + R3   } Data dependency
j: R4 ← R1 * R5   }
```

Here j reads R_1 before i saves result in R_1.

sol^n: solutions discussed under data hazard.

WAW (Write After Write)

→ j tries to write an operand before it is written by i.

```
i: R1 ← R2 * R3   } write dependency
j: R1 ← R4 + R5   }
```

If j writes R_1 first then final value in R_1 will be the value written by i.

WAR (Write after Read)

→ j writes to a destination before i read it. So i incorrectly gets new value.

```
i: R1 ← R2 + R3   } Anti-Dependency
j: R2 ← R4 * R5   }
```

This problem occurs if j writes R_2 before i reads it

→ write dependency & anti-dependency are false dependencies.

The sol^n for these two dependencies is register renaming.

Register renaming:

→ Here we use another register inplace of common register that is causing the problem.

i.e., i: $R_1 \leftarrow R_2 * R_3$
j: $R_6 \leftarrow R_4 + R_5$ (solving write dependency)

i: $R_1 \leftarrow R_2 + R_3$
j: $R_6 \leftarrow R_4 * R_5$ (solving anti-dependency)

→ Register Reraming is H/w sol^n.

i.e., H/w has to detect these dependencies and has to rename the registers.

(Q43 / G-08) Consider a pipelined processor with following four stage....

Sol:

	1	2	3	4	5	6	7	8
ADD	IF	ID	EX	WB				
MUL		IF	ID	EX	EX	EX	WB	
SUB			IF	ID	–	–	EX	WB

∴ 8 cycles

shortcut:

IF	ID	EX	WB
1	1	|	1
		|	
		|	

		Extra(stall)
ADD	1	- 0
SUB	1	- 0
MUL	3	- 2

no of cycles req (without stalls) = 4 + 3 - 1 = 6

no of mul operations = 1

=> no of stalls = 1 * 2 = 2

∴ total cycles req = 6 + 2 = 8

(This shortcut works only if one phase takes extra cycles)

Pipeline Efficiency:

→ Because (stall ~~stop~~ cycles) efficiency of pipeline reduces.

$$\text{efficiency of pipeline} = \frac{S'}{S} \times 100$$

S' → speed up with hazards

S → speed up without hazards

(Q44) Consider a 5-stage pipeline;
Consider execution of 1000 inst^{n}s of which 200 inst^{n}s cause 2 stall cycles each.

1. Calculate CPI of pipeline.

Sol:

no of cycles for execution

$= (k+n-1) + \text{extra cycles}$

$= (5+1000-1) + (200 \times 2)$

$= 1404$ cycles

$\Rightarrow CPI = \frac{1404}{1000} = 1.404$

2. If pipline cycle time is 3ns, what is avg inst^{n} execution time.

Sol:

$CPI * t_p = 1.404 * 3 = 4.212$

3. Find CPI under ideal conditions.

~~$CPI_{ideal} = 1$ (for any pipelined system)~~

$CPI_{ideal} = (n + \text{extra cycles}) / 1000$

$= (1000 + 200*2)/1000$

$= \frac{1400}{1000}$

$= 1.4$

Note:

$$CPI_{ideal} = 1 + (\text{stall freq}) * (\text{no of stall cycles})$$

Q45 / 4-14

Consider a 6-stage pipeline ....

Sol:

$$CPI_{(pipeline)} = 1 + (0.25)(2) = 1.5$$

1 instⁿ execution time = 1.5 t_p

1 instⁿ execution time in non-pipeline = $6 * t_p$

$$\Rightarrow \text{speed up} = \frac{6\,t_p}{1.5\,t_p} = 4$$

$$\text{Efficiency of pipeline} = \frac{S'}{S_{max}} \times 100 = \frac{4}{6} \times 100 = 66.66\,\%$$

Q46 Consider below stage delay for instⁿ I_1, I_2, I_3, I_4 for 4-stage pipline.

Find no of cycle for below program

```
for(i=1; i<=2; i++)
    (I1, I2, I3, I4);
```

	S_1	S_2	S_3	S_4
I_1	2	1	1	2
I_2	1	3	1	3
I_3	2	2	2	1
I_4	1	3	1	2

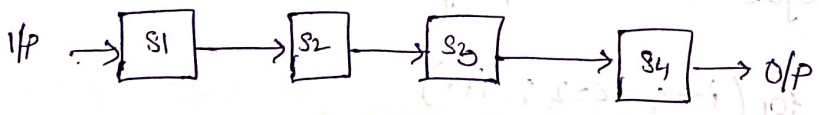
Sol:

	S1	S2	S3	S4
I_1	2	3	4	6
I_2	3	6	7	10
I_3	5	8	10	11
I_4	6	11	12	14
I_1	8	12	13	16
I_2	9	15	16	19
I_3	11	17	19	20
I_4	12	20	21	23

30/11/20

Non-Linear Pipeline

Linear pipeline:

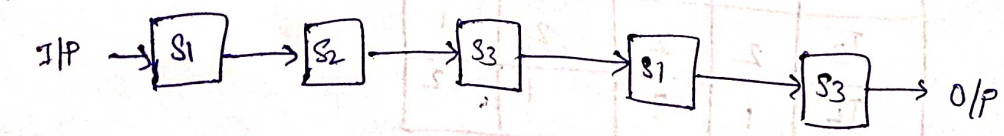


A task goes through each segment, in order, exactly once.

Non-linear pipeline:

A segment may be used multiple times during execution of one task.

Eg:



Non-linear pipeline is represented using reservation table

This table shows which segment is used at which cycle.

Consider 3-stage non-linear pipeline

Stage \ Clock cycle	1	2	3	4	5	6	7	8
S1	X					X		X
S2		X		X				
S3			X		X		X	

Latency: no of time units (clock cycles) blw two initiations of a pipeline is called latency. Latency is always non-negative.
(providing i/p in pipeline.)

→ Collision: without any hazard, latency in linear pipeline is 1 cycle.

Collision: When two or more initiations are done at same pipeline stage at same time, then we say a collision has occured. This is resource conflict.

Permissible Latency: Latency which does not cause a collision

Forbidden Latency: Latency which causes collision.

Eg: for above reservation table, consider

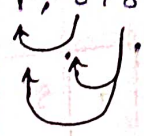
	1	2	3	4	5	6	7	8	9	10	11	12
I_1:	S1	S2	S3	S2	S3	S1	S3	S1				
(latency=5) I_2:						S1						
(latency=2) I_3:							S1	S2	S3	S2	S3	S1
(latency=7) I_4:												S1

So here for S1,

Forbidden latencies are 2, 5, 7

an easy way for determining these latencies is

S1 is used in 1, 6, 8 cycles



∴ Forbidden latencies w.r.t S1 are 6-1=5
8-6=2
8-1=7

Similarly we calculate for other stages also

For S2, 4-2=2

for S3, 5-3=2
7-5=2
7-3=4

∴ Forbidden latencies are 2, 4, 5, 7

⇒ permissable latencies are 1, 3, 6, 8

Collision Vector:

Collision vector is an n-bit vector

$C_n \ldots C_2 C_1$ such that

$C_i = 0$, if i is permissible latency

$C_i = 1$, if i is forbidden latency

⇒ Collision vector for this example is

8	7	6	5	4	3	2	1
0	1	0	1	1	0	1	0

State Diagram:

→ Take a state S.

→ For every 0 at position i

* shift collision vector of state S 'i' to right by i positions
* Take bitwise OR with original collision vector

Eg: state 01011010:

(i) 0 at position 1:

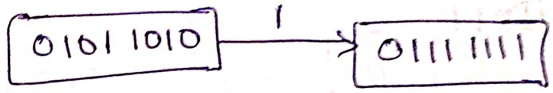
∴ right shift by 1

⇒ 00101101

bitwise OR

```
  01011010
  00101101
  --------
  01111111
```

↳ This is a new state.

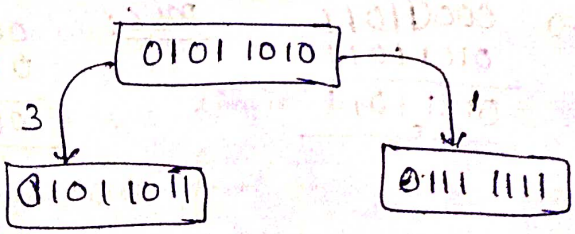
⇒ 

(ii) 0 at position 3:

∴ right shift by 3

```
  00001011
  01011010
  --------
  01011011
```

↳ this new state

⇒ 

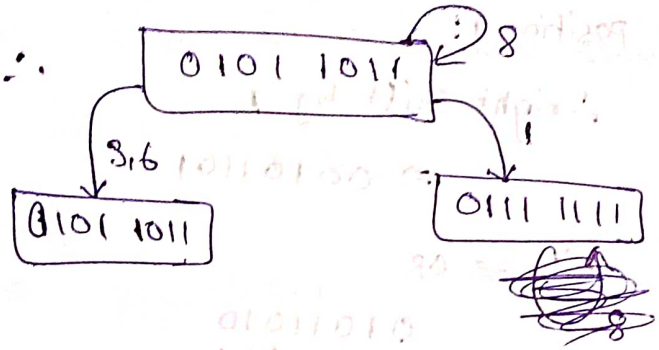
(iii) 0 at position 6:

R·shift ⇒ 0000 0001
0101 1010
0101 1011

↳ ~~new state~~ This is same state reached by B.

(iv) 0 at position 8:

R·shift ⇒ 0000 0000
0101 1011
0101 1011



~~State 0101 1011~~:

0 at pos 3:

0001 0011

State 0111 1111:

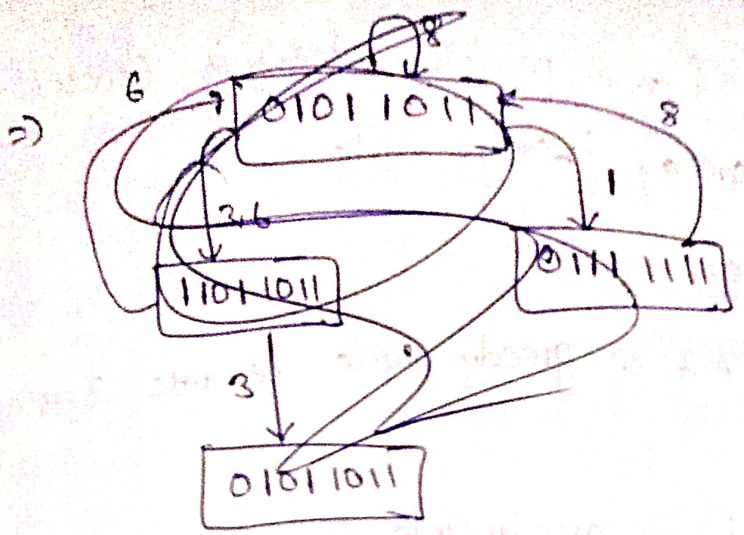
R·shift ⇒ 0000 0000
0101 1011
0101 1011

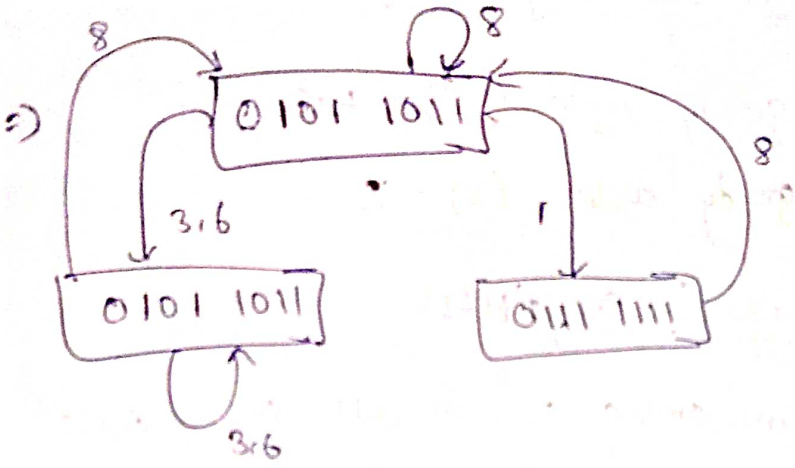
→ Here we must take original collision vector.

State 0101 1011:

pos 3: R·shift ⇒ 0001 0111
0101 1011
0101 1011

pos 6:
00000001
01011011
01011011

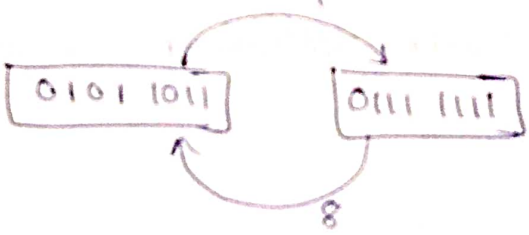




Simple Cycle:

A simple cycle is a latency cycle in which each state appears only once.

i.e.,



∴ (1,8) is a simple cycle.

silly (8) is a simple cycle

silly (3,8) (6,8), (3), (6) are also simple cycle.

Greedy Cycle:

A greedy cycle is a simple cycle whose edges are all made with minimum latencies from their respective starting states.

i.e., At each state if min possible latency is chosen then it is called greedy, ~~latency~~ cycle

=> (1,8) is a greedy cycle.

But (3,8) is not a greedy cycle because 3 is not minimum.

Silly (3) is also a greedy cycle.

avg of greedy cycle (1,8) = 4.5

avg of greedy cycle (3) = 3

Minimum Average Latency (MAL)

It is minimum among avgs of alls greedy cycles.

i.e., Here min avg latency = 3

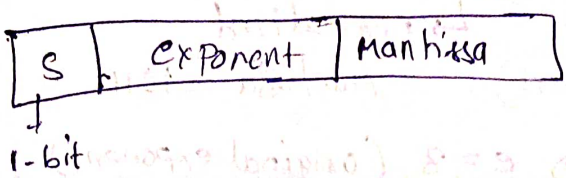
Note:

The cycle (1,8) means provide 1st i/p with latency 1, then with latency 8 then with latency 1 and so on...

Floating point Representation

→ Floating point representation can represent a larger range of numbers compared to fixed point number repre. given same no of bit.

→ General representation of floating point representation is



- Mantissa is signed normalized (implicit/explicit) fraction number.
- Exponent is stored in biased form.

Biased Exponent:

→ Biased exponent says that exponent should be only unsigned

Eg: consider 4-bit exponent

range with 4 bits: -8 to 7

now we need to change it to positive range

⇒ req. range: 0 to 15.

i.e., we add a bias 8

⇒ $E = e + 8$ → bias

- E: biased exp (or) stored exp
- e: original exp

→ If exponent is of k-bits then bias = 2^{k-1}

i.e., exponent is stored as excess 2^{k-1} code.

→ Mantissa is number after decimal point.

Normalized Mantissa:

Explicit Normalization (Default consideration)

- In explicit normalization a number is represented with 0.____ and the digit the immediatly follows '.' is 1.

$(101.11)_2 \xrightarrow{\text{explicit Norm.}} 0.10111 \times 2^{(3)}$ → original exponent.

↳ normalized mantissa = 10111

⇒ e = 3 (original exponent)

biased exponent, E = 3 + bias

Implicit normalization / Implicit preceding:

Here a number is represented as 1.____ and immediate digit following '.' can be any.

$(101.11)_2 \xrightarrow{\text{implicit norm}} 1.0111 \times 2^{2}$

⇒ M = 0111

e = 2

E = 2 + bias.

Value formula:

S	E	M

Explicit normalization:

$$\text{value} = (-1)^S \times 0.M \times 2^{E-\text{bias}}$$

Implict normalization

$$\text{value} = (-1)^S \times 1.M \times 2^{E-\text{bias}}$$

(Q50) Consider a 16-bit register used to store floating point numbers. The mantissa is normalized signed fraction number. Exponent is represented in excess-32 form. what is 16-bit value for $(13.5)_{10}$ in this register.

Sol:

bias = 32

⇒ 6 bits of exponent

$(13.5)_{10} = (1101.1)_2$

explicit normalization: 0.11011×2^4

$E = e + 32 = 36$

$E = 100100$

$S = 0$

1	6	9
S	E	M

⇒

0	100100	1101 1000 0

(Q51) Consider

S	E	M
1	6	9

i) Find max value that can be represented by above register.

Sol:

$$(-1)^S \times 0.M \times 2^{E-bias}$$

for ∘ val to be max

$M = 1111\ 1111\ 1$

$E = 1111\ 11$∘

$$(-1)^0 \times 0.1111\ 1111\ 1 \times 2^{63-32}$$

$$0.1111\ 1111\ 1 \times 2^{31} = 1111\ 1111\ 1 \times 2^{22}$$

$= 511 \times 2^{22}$

$= (2^9 - 1) \times 2^{22}$

$= 2^{31} - 2^{22}$

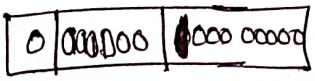
(ii) Find min value?

Sol:

Here every thing remain same. except sign

$\therefore -(2^{31}-2^{22})$

(iii) Find smallest positive value?



$val = (-1)^0 \times 2^{-32} \times 0.1000\ 0000\ 0$

$= 0.1 \times 2^{-32}$

$= 2^{-33}$

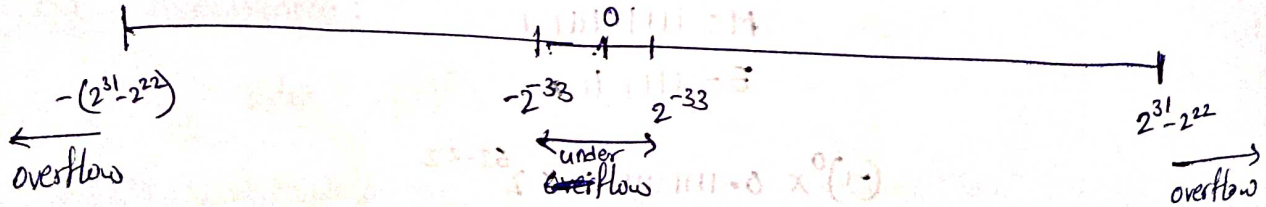
In explicit normalized form, mantissa can never be all zeroes since after decimal point there must be one.

(iv) Find largest negative value

i.e., -2^{-33}

Note:

In above question, the range of number than can be represented by above format is



→ If fixed point, with 16 bits we, max value is $(2^{15}-1)$

but here max value is $2^{31}-2^{22}$

i.e., range of number with floating point repretn is more.

Note:

→ If more number of bits are used in mantissa, then precision (accuracy) is more.

→ If more number of bits are used in exponent, range is more.

Disadvantages of conventional floating point representation:

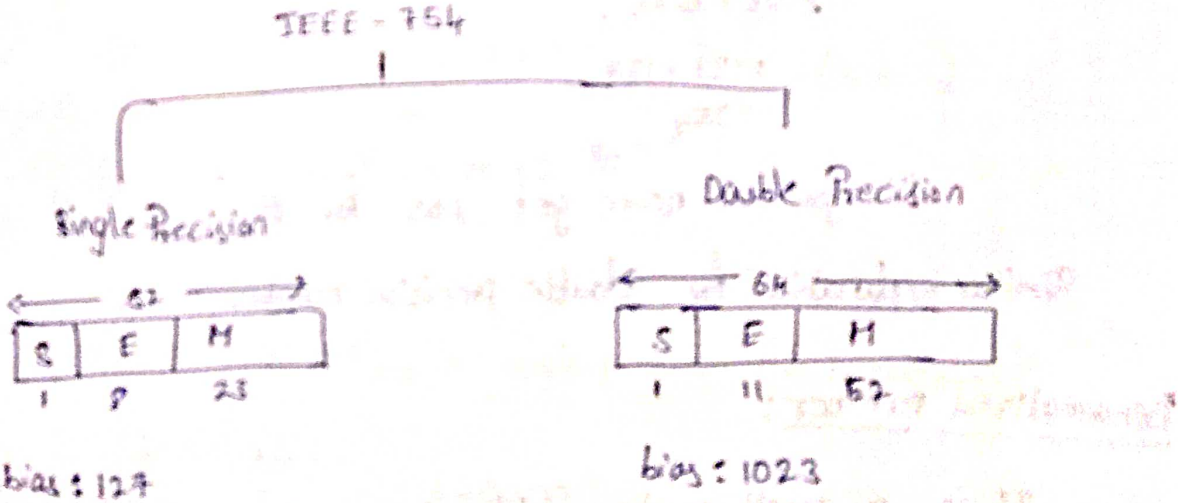
→ It cannot represent 0.

→ It cannot represent infinity.

→ It cannot store a number which cannot be normalized.

IEEE-754 Floating point Representation:

This overcomes the disadvantages of conventional representation.



→ ~~The bias is different~~

E = all 0's or E = all 1's } Special Numbers

S	E	M	number
0/1	00...0	0...00	±0
0/1	11....1	11...11	±∞
0/1	00....0	M≠0	Denormalized (or) fraction
0/1	11....1	M≠0	NAN(Not a number)
0/1	rest	rest	Implicit normalized

NAN:

Consider single precision,

if e=127 (i.e., max possible exponent)

E = e + bias.

= 127 + 127

= 254

i.e., you can never get 255 for E

Similar explanation for double precision number.

Denormalized number:

It is a small number which cannot be normalized.

Eg: Consider a below example for single precision

0.00000....11

↓ implicit normalization

1.1×2^{-130}

i.e., M=1 ; e=-130; E = -130+127

= -3

but stored exponent (biased) can have only non-negative values.

In conventional representation, storing this kind of numbers wouldn't be possible

→ But in IEEE-754,

we first normalize the number upto 23 bits

Consider we get 0.011×2^{-126}

In this case we store as below

E = 0000 0000

M = 011

Value formula:

implicit normalization:

value = $(-1)^s \times 1.M \times 2^{E-bias}$ bias = 127 or 1023

Denormalized:

value = $(-1)^s \times 0.M \times 2^{-126}$ (or) $(-1)^s \times 0.M \times 2^{-1022}$ $\Big\}$ $-(bias-1)$

Q51 Represent $(-14.25)_{10}$ in single precision (in hexadecimal)

Ans: C1640000

Q52 Find value of below single precision representation

01000001111000...00

Ans: 28

Q54 Find value of below single precision representation

00000000011100...0

Ans: E → all 0's

val = $(-1)^0 \times 0.11 \times 2^{-126} = 3 \times 2^{-128}$

Q54 Represent $(1.0)_{10}$ in single precision

Ans:

0	0111 1111	000...00

Q55 Find max vallue possible in IEE-754 (single precision)

Ans:

S	E	M
0	1111 1110	111...1

$$\Rightarrow \text{value} = 1.111\ldots1 \times 2^{254-127}$$

$$= \underbrace{1111\ldots1}_{24} \times 2^{104}$$

$$= (2^{24}-1)(2^{104}) = (2^{128}-2^{104})$$

Q56 Fin min value possible in single precision

Ans: $-(2^{128}-2^{104})$

Q57 Find min positive val for IEEE-754 single precision.

Sol:

This is possible under denormalized form

0	0000 0000	000...001

$$\text{value} = (-1)^0 \times 0.000\ldots001 \times 2^{-126}$$

$$= 1.0 \times 2^{-149}$$

$$= 2^{-149}$$

Practice Questions:

(Q58) Consider a digital computer which supports 32 3-address instⁿs. Size of each instⁿ is 38 bits. The system supports a word addressable memory with word size of 32 bits. Then maximum allowable size of memory is ____ kB?

Ans: 8

For same question if memory is byte addressable then

Ans: 2

(Q59) Consider a digital computer which supports max of 32kB word addressable with word size 32-bits. The computer supports 40 distinct instⁿs each of which has 4 fields

(i) Opcode

(ii) Mode field to specify one of 7 add. modes

(iii) Reg field to specify one of 24 processor reg.

(iv) Mem add field.

The size of instⁿ is ____ bits

Ans: 27

(Q60) Consider a hypothetical processor with largest instⁿ length being 32 bits and total registers are 16 R_0-R_5. Processor supports following instructions:

ADD R_i, R_j

SUB R_i, R_j

AND R_i, R_j

NOT R_i

MOV Ri, Rj

LOAD Address

Store Address

JUMP address

JZ Address

The max no of of address lines on this processor is ____

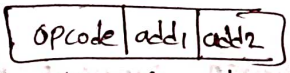
Sol:

If insth size is fixed ⇒ opcode length varies.

If insth size is variable ⇒ opcode size is fixed

Here we have variable length insth's

for insth 1, 2, 3, 5 the format is

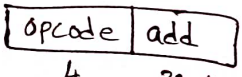


insth size = 12

for insth 4 the format is

insth size = 8

for insth's 6, 7, 8, 9



∴ Max no of address lines = 2^{28}

(Q61) Insth = 26 bits; add = 12 bits;

1-add & 0-add insth's are supported.

Find range of 1-add & 0-add insth's possible

Sol.

1-add

q 17

↓

we can't use all 512 combinations but 0-add are also supported

∴ Range : 1 to 511

0-add

Max 0-add will be possible if only 1 1-add instⁿ is present.

⇒ max 0-add instⁿs = 511×2^{19}

∴ Range : 1 to 511×2^{19}

whenever it is given that a certain k-add instⁿs are supported then there must be atleast 1 instⁿ of that type.

862

2-add, 1-add, 0-add, instⁿs are supported.

- instⁿ – 20 bits; add – 7 bits

find max no of 0-add instⁿs possible.

Sol

Ans : $8063 * 2^7$

863

instⁿ – i bits; add – a bits

0-add, 1-add, 2-add instⁿs are supported.

If no of 2-add instⁿs = x

" " 1-add " = y

" " 0-add = ?

Sol:

total no of combinations = 2^i

$x \times 2^{2a} + y \times 2^a + z = 2^i$

$z = 2^i - x \cdot 2^{2a} - y \cdot 2^a$

2-add: | i-2a | a | a |

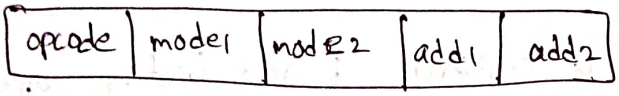
1-add: | i-a | a |

0-add: | i |

Q64) Consider a PC-relative mode type branch instⁿ which takes branch on address 740 in memory. The instⁿ has offset value 160. what is the address of this instⁿ in memory if instⁿ is stored on 4 memory locations.

Ans: 576

Q65) Consider a 6-word instⁿ



The first operand (destination) uses register indirect mode and second operand uses indirect mode. Assume each operand is of size 2 words, each add is of 2 words and main memory takes 50 ns for 1 word.

Total memory access time required in

1. Fetch cycle of instⁿ
2. Execution cycle of instⁿ
3. Instⁿ cycle of instⁿ

Sol:

1. $6 * 50 = 300$ ns
2. Reading: $2 \times 50 + 4 \times 50$
   Writing: 2×50
   400
3. total: 700ns