Computer Organization and Architecture

★ Introduction :-

⇒★ Digital Computers :

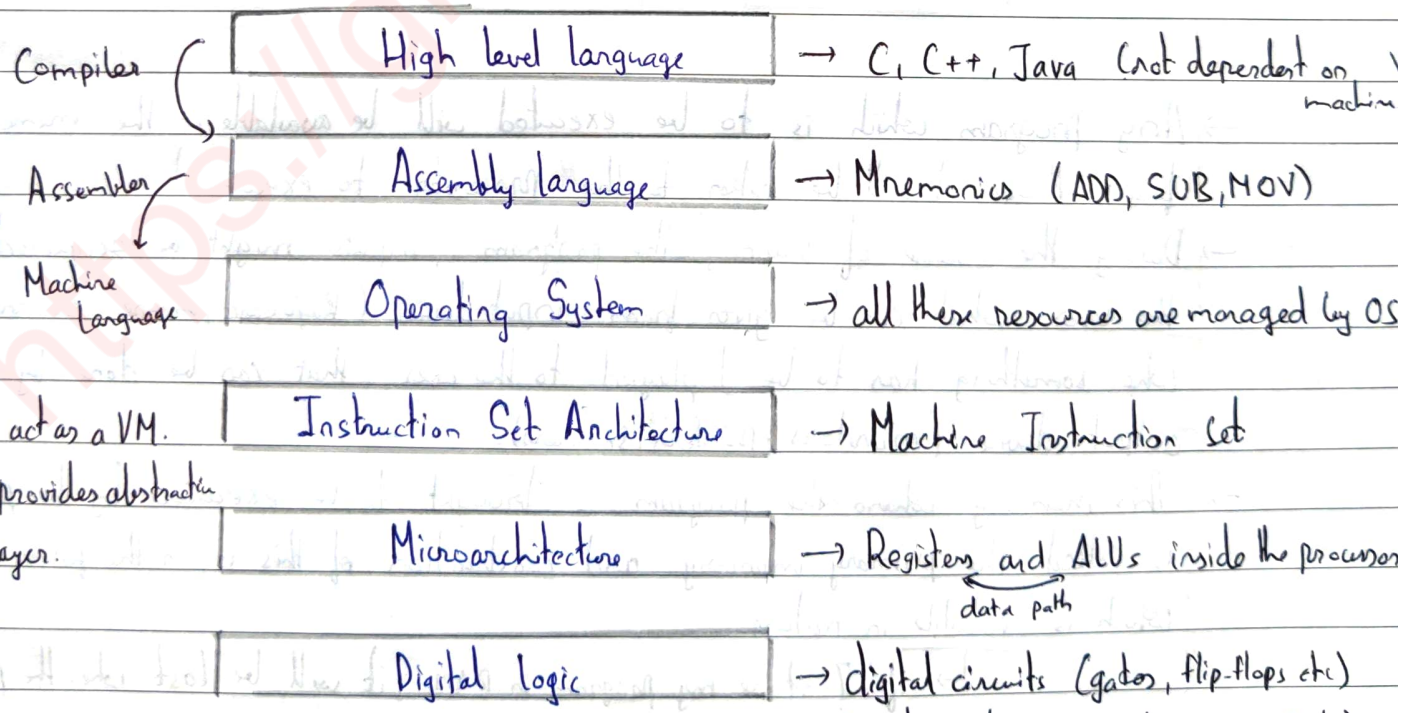
→ These are machines which can perform a set of instructions and they work on discrete data.

→ These consists of both Hardware (microprocessors, storage, input-output devices) and Software (compilers, Operating system, application programs) components.

⇒★ Types of Grereral purpose computers :

1) Personal Computers (PCs)
2) Workstations (more capability, high resolution graphics, high computing power)
3) Mainframes (large storage capacity, process bulk data)
4) Super computers (can perform large scale numerical calculation, have multiple functional units)

⇒ Layers, Languages, and VMs



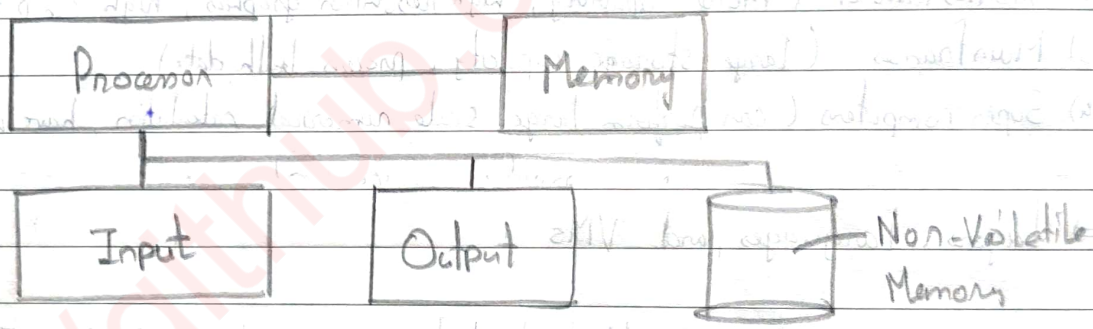
Each layer act as a VM.
Each layer provides abstraction to below layer.

Registers are a collection of flip-flops that can store data.
16 bit Register → stored 16 bit of data

⇒ Architecture Vs. Organisation

Architecture	Organisation
→ What a computer does!	→ How it does?
→ Functional behaviour of computer system	→ Structural relationship between parts of computer
→ Deals with high-level design.	→ Deals with low-level design.
→ Comprises of logical entities like instruction sets, registers, data types etc	→ Consists of physical units like circuit design, peripherals etc.
→ Architecture decided first.	→ Organization decided after architecture.

★ Basic components of a Computer:-



→ Any program which is to be executed will be available in the memory and from there it will be taken to the processor to execute it.

→ During the course of running the program, inputs might be required from the user which can be given from input device (Keyboard, mouse, scanner) and when something has to be displayed to the user, that can be done by the output device (monitors, printers, scanners).

→ This memory where the program is brought to be executed by the processor is called the primary memory. and construction of this is in the forms of RAM. which is volatile in nature.

↳ if I store my program in RAM, it will be lost when the power is switched off

→ So, We need a permanent storage for the programs, so we also use Non-Volatile

memory devices (Harddisks).

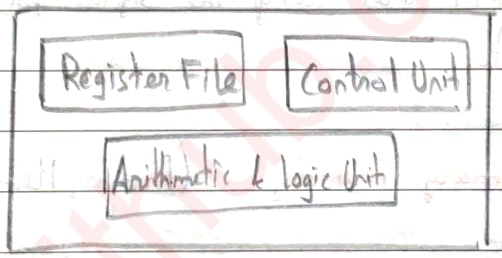
→ All the components in the fig are connected by means of a system bus.

→ This non-volatile memory stores the program and anytime when program is to be executed, it will be taken to the primary memory.

Q Why can't we directly taken the program from non-volatile memory to the processor? Because non-volatile memory are mechanical in nature and are very slow, so slow down the processor while primary memory is a fast memory.

→ This entire process is known as Von Neumann Architecture.

★ Inside a Processor :-



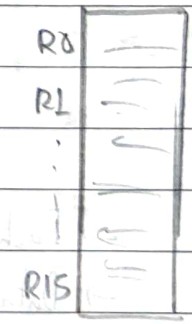
All of the component inside the processor are connected through wires.

⇒ Register File:

→ They are set of registers and are collectively known as Register File.

→ They are high speed elements, they provide local storage for the operands to be stored, data to be stored, any temporary result to be solved.

→ From here the data can be taken to the ALU and control unit & operation can be performed on them and result can be stored back into the Register File.



⇒ Arithmetic and Logic Unit (ALU):

→ Most computer operations are executed in ALU.

data on which a particular operation has to be performed

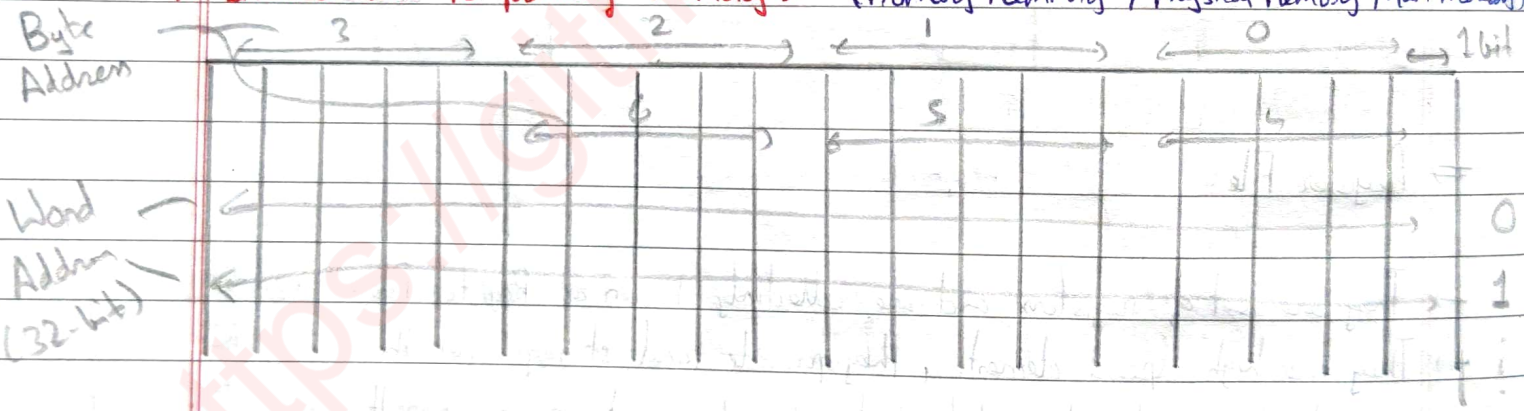
→ For any arithmetic on logical operation, operands are brought into the processor and operation is performed by ALU.

→ Operands are stored in storage called Registers.
→ Registers are high-speed elements and can store one word of data.

⇒ Control Unit:

→ Operations of ALU, memory, I/O devices are coordinated by the control unit.
→ I/O transfers, data transfers between processor and memory are controlled by control unit through timing signals.
→ A large set of control lines carry the signals for timing and synchronization of events in all units.

★ Introduction to primary memory :- (Primary Memory / Physical Memory / Main memory) (RAM)



→ Each cell of primary memory contains 1 bit. (either 0 or 1)
→ Data transfer between the memory and processor will not be as bits but it would be in bytes / words.

→ Word depends upon the word length.
32 bit machine → word length = 32 bits = 4 bytes
16 bit machine → word length = 16 bits = 2 bytes

→ Each location in the memory needs an address because if I have to transfer data between processor and the memory, the processor should know exactly which memory location it wants to get data from or which location it wants to store data.

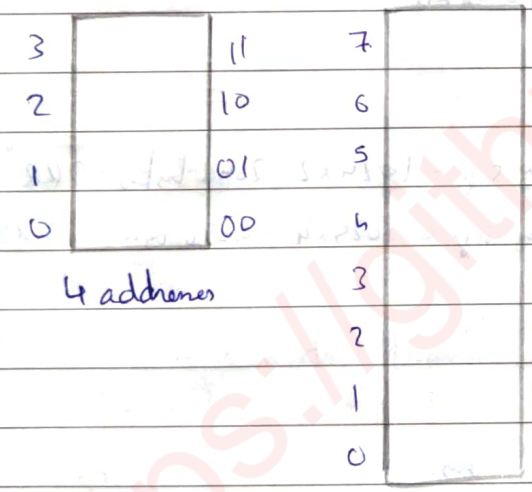
→ So, two operations are possible: 1) Read → getting data out of memory from a location.
2) Write → storing data at a memory address/location.

→ So, how to specify addresses in the memory, it depends whether the memory is byte addressable or it is word addressable.

byte addressable: each byte is going to have an address

word addressable: each word is going to have an address

⇒ Memory addresses:



n addresses → $\log_2 n$ bits required to specify the memory address

Q How many bits are required to specify addresses in a 16 GB memory?

KB – 2^{10} bytes
MB – 2^{20} bytes
GB – 2^{30} bytes
⋮

16 GB = 16×2^{30} bytes = 2^{34} bytes

Byte Addressable → No. of address = 2^{34}
No. of bits required = $\log_2 2^{34}$ = 34 bits

Word Addressable
Let Word length = 4 bytes (32 bit machine)
No. of Address = $2^{34}/4 = 2^{32}$ words
No. of bits required = $\log_2 2^{32}$ = 32 bits

⇒ Memory Size :

Ex : 1) No. of bits = 8 → $2^8 = 256$ addresses (0-255)

Maximum Memory Size : Byte Addressable → memory size = 256 bytes

Word Addressable → 256 words (memory size)
Word length = 16 bits = 2 bytes → 256 × 2 = 512 bytes
Word length = 32 bits = 4 bytes → memory size = 256 × 4 = 1024 bytes

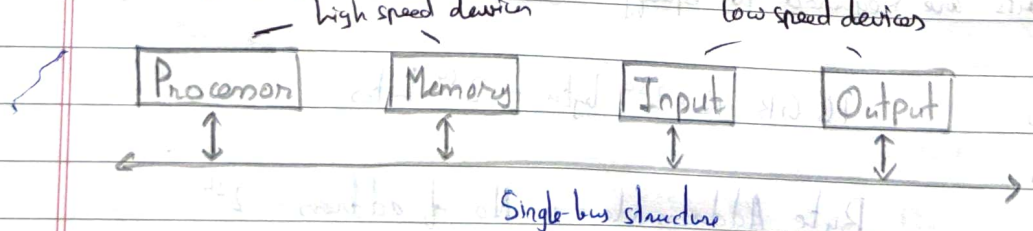
2) No. of bits = 10 → $2^{10} = 1024$ addresses (0-1023)

Byte Addressable → memory size = 1024 bytes = 1 KB

Word Addressable → 1024 words
Word length = 16 bits = 2 bytes → memory size = 1024 × 2 = 2048 bytes = 2 KB
Word length = 32 bits = 4 bytes → memory size = 1024 × 4 = 4096 bytes = 4 KB

★ Introduction to Bus :-

→ Group of wires connecting devices is called bus.
→ When a word is transferred between components, all bits are transferred in parallel simultaneously
→ One bit per line



Single Bus Structure:

→ At any given point of time, only two components will be able to trasfer data and will be able to use the bus.
→ This is a drawback of this structure that if suppose the processor and memory are using the bus, then input and output devices cannot use and vice versa.
→ It is less expensive.

Multi Bus Structure:

→ Using this structure, I can have a separate bus between processor and memory and other for input and output devices..
→ Though this would be more expensive but the concurrency of operations would increase.
→ The transfer of data on the bus should be such that the high speed devices should not be constrained by the low speed devices.

Processor, Memory → high speed devices
Input, Output, Not-Volatile memory → low speed devices

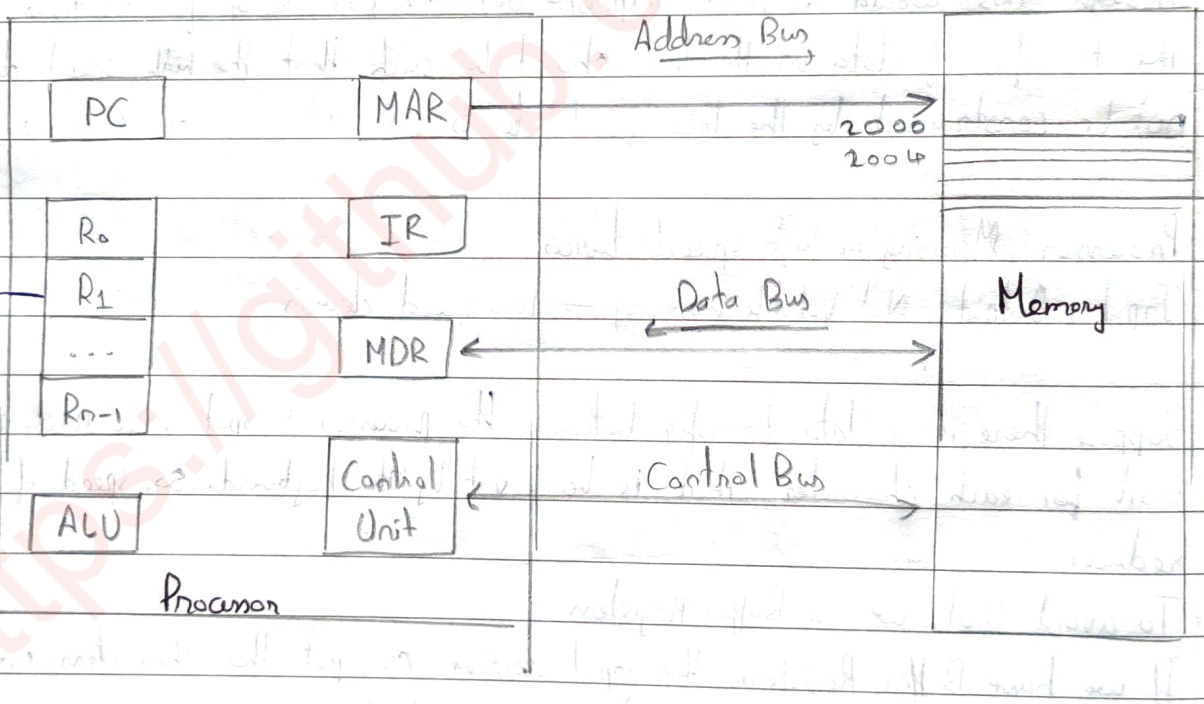
→ Suppose there is a data transfer between the processor & input, so processor will have to wait for each character that is being sent by the keyboard. So, speed of processor reduces.
→ To avoid that, we use Buffer Registers.
→ If we have Buffer Registers, the input devices can put the characters over it, once the Buffer Register gets full, it can be sent to the processor.

Similar situation between output device & processor.

★ Basic Operations :-

→ Computer accepts program and data through input units and stores in memory.
→ The program is the set of instructions which tells the system what task has to be performed and if the program requires permanent storage, it will be stored in secondary memory which is non-volatile and it will be brought into primary memory / RAM when it has to be executed.
→ Instructions and data are fetched into processor and executed.
→ Processed information is sent to the outside world through output devices.
→ All activities and data transfer are controlled by control Unit.

⇒ Processor and Memory:



→ PC (Program Counter) holds the address of the next instruction that has to be executed.

Assumptions: Word length = 32 bits = 4 bytes
Byte Addressable memory and each instruction (Machine language form) requires 1 word to store (4 bytes)

→ Suppose the next instruction that has to be executed is at address 2000. So, 2000 will be stored in PC.

→ MAR (Memory Address Register) holds the address of the memory location that has to be addressed / accessed. So, PC will pass this address 2000 to the memory address register. At the same PC will increament to next instruction location i.e. 2001.

→ Now, this MAR holds 2000, this will be passed on to the address bus which carries the address from the processor to the memory

→ Now, whatever data is stored at 2000 in the memory is transfer to MDR (Memory Data Register) via Data Bus.
or instruction

→ From MDR, it is brought to IR (Instruction Register) which decodes the instruction and operations are performed using ALU.

→ All these operations are controlled by the Control Unit and these control signals are sent via the control bus.

X ★ Number Representation :-

→ A number in a computer system is represented by a string of bits, called a binary number.

→ Consider an n-bit vector $B = b_{n-1} \ldots b_1 b_0$ (MSB ↑ … LSB ↑)

$$V(B) = b_{n-1} \times 2^{n-1} + \ldots + b_1 \times 2^1 + b_0 \times 2^0$$

Positive and Negative numbers can be represented using any of the 3 systems:

1) Sign and Magnitude : ~~positive~~ MSB = Sign Ex: 5 - 0101 positive: MSB=0
~~negative~~ Rest - number -5 - ~~1101~~ 1101 negative: MSB=1

2) 1's Complement : Ex: 5 = 0101 , -5 = 1010

✓ (Generally used) 2) 2's Complement : Ex: 5 = 0101 , -5 = 1's complement of 5+1 = 1010+1 = 1011

⇒ Addition of Unsigned Integers:

n bits → 0 to $2^n - 1$ numbers

```
  5          0101          8          1000
 +9    ⇒   + 1001        +9    ⇒   + 1001
----        ------       ----       ------
 14         1110          17        (1)0001   5 bits
                                       ↓
                                   don't ignore
```

⇒ Addition of Signed Integers:

Using 2's Complement: To add 2 numbers, add their n-bit representations, ignoring the carry-out bit from the most significant bit (MSB) position.

4 bit representation → −8 to +7
n bit representation → -2^{n-1} to $2^{n-1}-1$

```
Ex: +7      0111
    -5      1011
           ------
          (1)0010 = 2
            ↓
          ignore

   +7      0111
   +3      0011
          ------
Overflow ← (1010) × 10
            ⇓
           01010
```

Same for Subtraction

⇒ Overflow in Integer Arithmetic:

→ Using 2's complement representation, n bits can represent values in the range -2^{n-1} to $+2^{n-1}-1$.

→ When the actual result of an arithmetic operation is outside the representable range, an arithmetic overflow has occurred.

→ Clearly, overflow may occur only if both summands have the same sign.

(+8) + (+7) ✓

(+8) + (-7) ×

(-8) + (-7) ✓

→ When adding unsigned numbers, a carry-out of 1 from the MSB indicates that an overflow has occurred.

→ However, this is not always true when adding signed number.

Ex: Add +7 and +4 and check carry out

Add -4 and -6 and check carry out

Unsigned Integer : 4 bit → 0 to 15

$+9$ → 1001

$+8$ → 1000

17 → ① 0001

↓ overflow has occurred

Signed Inter : 4 bit → -8 to +7

$+7$ → 0111

$+4$ → 0100

17 → ① 1011

↓ overflow not occured

★ Instruction Set Architecture :-

Q What is a Instruction Set?

It is a collection of machine language instructions that a particular processor understands and executes.

It is the set of Assembly Language mnemonics that represents the machine code of a particular computer.

→ Instruction in a machine is dependent on computer i.e. different processors may have different instruction sets.
→ A newer processor that may belong to some family may have a compatible but extended instruction set of an old processor of that family.

→ Instruction Set Architecture is the set of processor design techniques used to implement the instruction work flow or hardware.
→ ISA tells that how your processor is going to process your program instructions.

⇒ RISC and CISC:

An ISA for a machine can be designed in two ways: 1) CISC 2) RISC

CISC (Complex Instruction Set Computing): → Intel

→ Complete a task in as few lines of assembly as possible.
→ Compiler has to do very little work to translate a high-level language statement into assembly.
→ Because the length of the code is relatively short, very little RAM is required to store instructions.
→ Build processor hardware that is capable of understanding and executing the operations.
→ Complex instructions, hence complex instruction decoding.
→ Instruction may take more than size of one word.
→ less number of general purpose registers required.

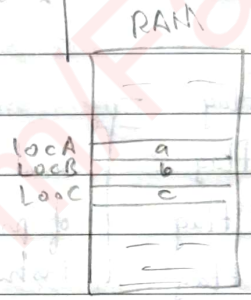
RISC (Reduced Instruction Set Computing): → SPARK

→ Compiler must perform more work to convert a high-level language statement into code of this form.

→ Simpler instructions, hence simple instruction decoding.
→ More instructions, so code is large.
→ Instructions come under size of one word.
→ Instructions takes single clock cycle to be executed.
→ LOAD/STORE architecture is used.
→ More number of general purpose registers is required.
→ Pipelining can be achieved.

RISC v/s CISC

C = a * b ; → HLL



CISC : MULT LocC, LOCA, LocB → performed by hardware (Complex Instruction)

RISC : LOAD R1, LocA
LOAD R2, LocB
MULT R3, R1, R2
STORE LocC, R3
} Performed by compiler (Simple Instructions)

⇒ Registers :

The registers that are available in the processor can be broadly classified into two:
1) General Purpose Registers 2) Special Purpose Registers

1) General Purpose Registers :

→ Hold the local variables, and intermediate results of calculations.
→ So, rather than storing these operands and intermediate results in memory while the instruction is being executed, they are stored in these locally available registers which provided high speed access to these operands.
→ These registers are interchangeable → ADD R1, R2, R3
ADD R7, R2, R3

2) Special Purpose Registers: PC, SP → visible at ISA level

→ They are Registers with a specific function like PC (points to the next instruction that has to be executed), Stack Pointer (always pointing to the top of the stack) and Program Status Word (PSW).

(If/else) ← Program Status Word Register →

↳ this bit represents zero flag

↓

- Zero Flag
- Sign Flag
- Parity Flag
- Carry Flag

— used to effect/show of previous instruction

Zero flag will go high wherever the result of previous operation resulted in a zero

→ Other special purpose register includes: MAR (address of the memory location that has to be accessed), MDR (holds the data that is been transfered between the processor and memory), IR (holds the instruction that is currently been executed).

MAR, MDR, IS → visible only at microarchiture level

⇒ Big Endian v/s Little Endian:

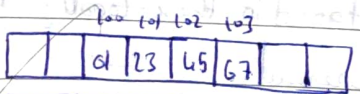
There are two ways in which memory address can be specified:
1) Big Endian 2) Little Endian

1) Big Endian

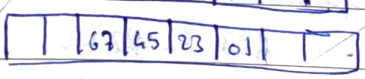
Lower byte address are used for the most significant bytes (the left-most bytes) of the word.

Extra:

Big Endian



Little Endia



MSB

32 bit word = 4 bytes

Byte Addr				
0	0	1	2	3
4	4	5	6	7
1020	1020	1021	1022	1023

Representing Data : Represent int 6 and string "RAM JAIN"

Requires 4 bytes in 32-bit machine

1 byte 0 | 1 | 2 | 3

00000000	00000000	00000000	00000110
R	A	M	
J	A	I	N

2) Little Endian :

Higher Byte addresses are used for the ~~most~~ least significant bytes (the ~~right~~ left most bytes) of the word

32 bit word = 4 bytes

1 byte

LSB

0	3	2	1	0
4	7	6	5	4
1020	1023	1022	1021	1020

Representing data : Represent int 6 and string = "RAM JAIN"

Requires 4 bytes in 32-bit machine

Addresses → 3 → 2 → 1 → 0

00000000	00000000	00000000	00000110
	M	A	R
N	I	A	J

15 14 13 12

Note : Never transfer data between Big and Little Endian (☺)

⇒ World Alignment : (Assume word length = 4 bytes , Byte Addressable)

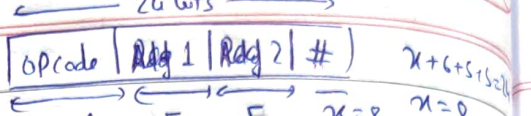
				0
				4
			'c'	8
				12

Non-Aligned Word ← no wastage of memory however accessing the memory by word addresses become difficult.

Insert : char 'c' → 1 byte required
int 9 → 4 bytes required

Aligned Word → Bytes wasted But addressing will be better.

Q A machine has 24-bit instruction format. It has 32 registers and each of which is 32 bit long. It needs to support 49 instructions. Each instruction has two register operands and one immediate operand. If the immediate operand is signed integer then the minimum value of immediate operand is -128.

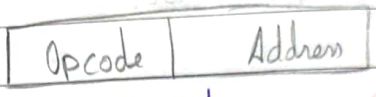


⇒ Instruction Formats:

Instruction consists of an opcode (operation code), usually along with information of where operands come from and where results go to.

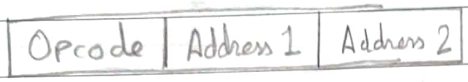
Zero-address

Ex: HALT → end of the program



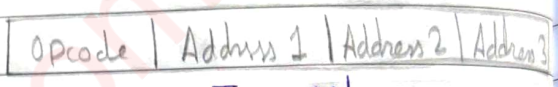
1 - Address

Ex: INC Reg, JUMP LOCA (memory)



Two-Address

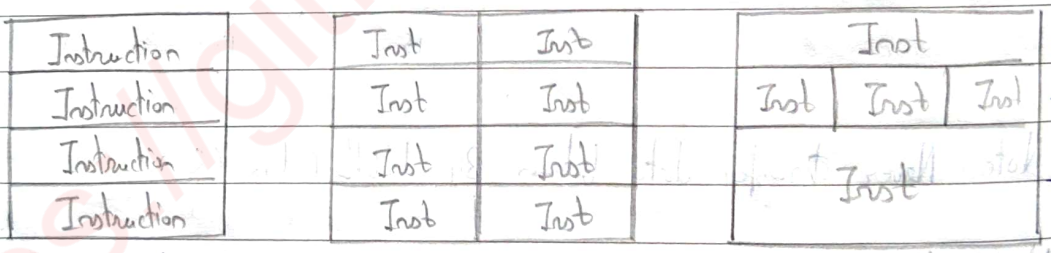
Ex: ADD Reg, Reg, LOAD R1, LOCA (destination, source)



Three-Address

Ex: ADD Reg, Reg, Reg

→ Instructions may have fixed or variable size



easy to access & decode

⇒ Instruction length:

n - 16 bit instructions → 16×n bits to store in memory → less space required but difficult

n - 32 bit instructions → 32×n bits to store in memory →

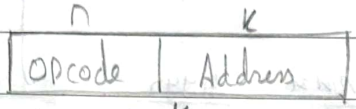
Shorter instructions are better than long ones because they have to be brought

Instruction length ↓ bits/s

from the memory into the processor and the Memory Bandwidth will increase & hence will be able to bring more no. of instructions.

⇒ Expanding Opcodes:

Consider an (n+k) bit instruction with a n-bit opcode and a single k-bit address.



→ 2^n machine operations possible
→ 2^k memory address possible

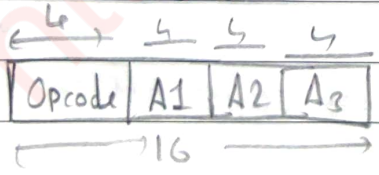
↓ ½

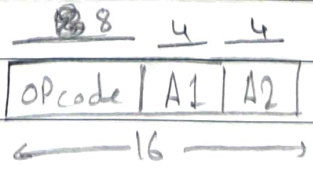
Opcode: n-1 & Address: k+1
→ 2^{n-1} machine operations possible
→ 2^{k+1} memory addresses possible
x2

So, by changing the no. of bits we are allocating to opcode and the address, we can change the no. of instructions machine can have and also change the addresses a machine can access.

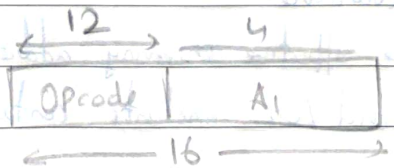
This kind of approach to specify codes for the instruction a machine can perform without changing the length of instruction is called Expanding Opcode.

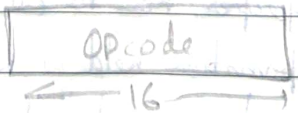
Q Consider a machine in which instructions are 16 bits long and addresses are 4 bits long. Required: 15 three-address instructions, 14 two-address instructions, 31 one-address instructions and 16 instructions with no address at all.





2^4 = 16 Registers R0 to R15





Combinations:

0000	xxxx	yyyy	zzzz	15	15×2^{12}
0001	xxxx	yyyy	zzzz		
⋮	⋮	⋮	⋮		
1110	xxxx	yyyy	zzzz		
1111	0000	xxxx	yyyy	14	14×2^{8}
1111	0001	xxxx	yyyy		
⋮	⋮	⋮	⋮		
1111	1101	xxxx	yyyy		
1111	1110	0000	xxxx	31	31×2^{4}
1111	1110	0001	xxxx	16	
⋮	⋮	⋮	⋮		
1111	1110	1111	xxxx		
1111	1111	0000	xxxx	15	
1111	1111	0001	xxxx		
⋮	⋮	⋮	⋮		
1111	1111	1110	xxxx		
1111	1111	1111	0000	16	16×2^{0}
1111	1111	1111	0001		
⋮	⋮	⋮	⋮		
1111	1111	1111	1111		

$+$

$\leq 2^{16}$ ← total combinations possible with 16 bits

⇒ Addressing Modes:

→ Addressing modes specifies where the operand is available.

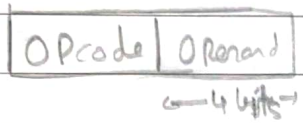
→ An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within an machine instruction or else where.

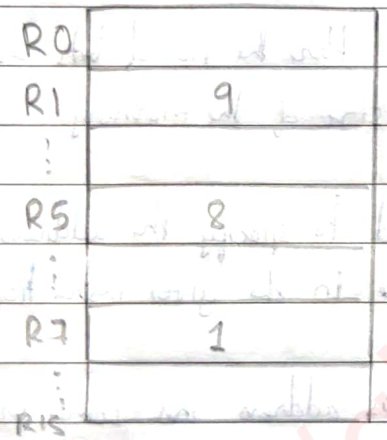
1) Register addressing mode:

→ Each instruction that we have could have an opcode (what operation is to be performed) and operands (where the data is available).

→ So, if this is the Register addressing mode, these operand will specify the name of the register where the data is kept.

Ex: Consider there 16 Registers so, 4 bits are required to represent them.

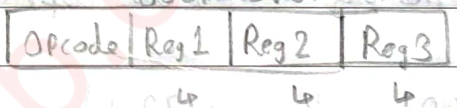




Destination

⇒ ADD R1, R5, R7

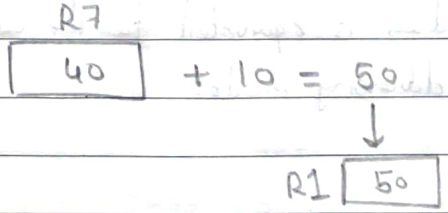
0001 0101 0111



2) Immediate addressing mode:

→ If we have to use constants in our ~~registers~~ instructions, we use the immediate addressing mode.

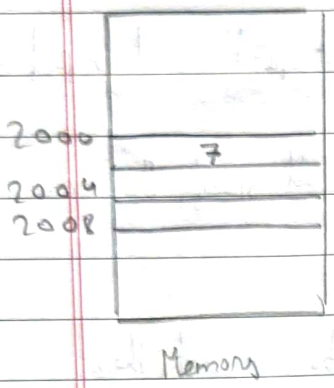
Ex: ADD R1, R7, #10



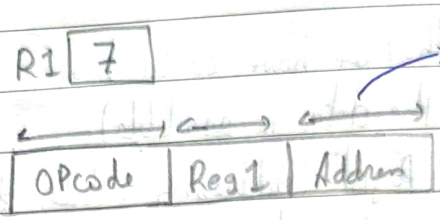
3) Direct / Absolute addressing mode:

→ If we have to transfer data between the memory and the processor, we need to know the memory address which has to be accessed and from where I need to read the data or write the data.

int x, y, z;
x = 7;



LOAD R1, 2000;



→ The no. of bits used by this memory address will be depend upon the size of the memory limited by the no. of bits that we have in this instruction.

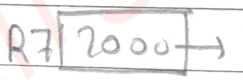
If 4GB memory → 32 bits required to specify the address

4) Indirect addressing mode:

→ In direct addressing mode: LOAD R1, 2000; . Here the no. of bits used by this memory address will depend upon the size of the memory.

If 4GB memory → 32 bits will be required to specify the address and this might not be possible in the given instruction length.

The other way of going about is to store this address ins some processor register.



LOAD R1, (R7)

→ This shows that R7 is being used as a pointer to address 2000 and the memory address is equivalent to the contents of Register R7. This is the Indirect addressing mode.

- Advantages:

1) No. of bits used to specify the address
2) If I want to access subsequent memory locations, I can do so by incrementing the register every time when I am going in loop. so, If in the 1st pass of the loop, R7 is containing 2000, I can increment this register R7 to point to the next address 2004 and so on.

5) Index addressing mode:

→ It is used when memory addresses are changing like an array

→ storing integers where each integer is taking 4 bytes / 1 Word if it is 32 bit machine.

If I want to bring this (2) data element into the processor, I will use an instruction like:

If direct addressing mode is used

1st element: LOAD R1, 2000
2nd element: LOAD R1, 2004

Better way would be to use index addressing mode

Access 1st element: LOAD R1, 2000(R5)

EA = 2000 + R5
= 2000

2000 → off-set (base)

R5 → index Register

R5 [0] → index

increment

OR

Access 2nd element: ADD R5, R5, #4

R5 [4]

LOAD R1, (R5,R6)

EA = [R5] + [R6]

R5 [0]

R6 [2000]

LOAD R1, 2000(R5)

EA = 2000 + R5
= 2004

So, this way in a loop, if I go, I can't access the successive elements of array by using this Index addressing mode.

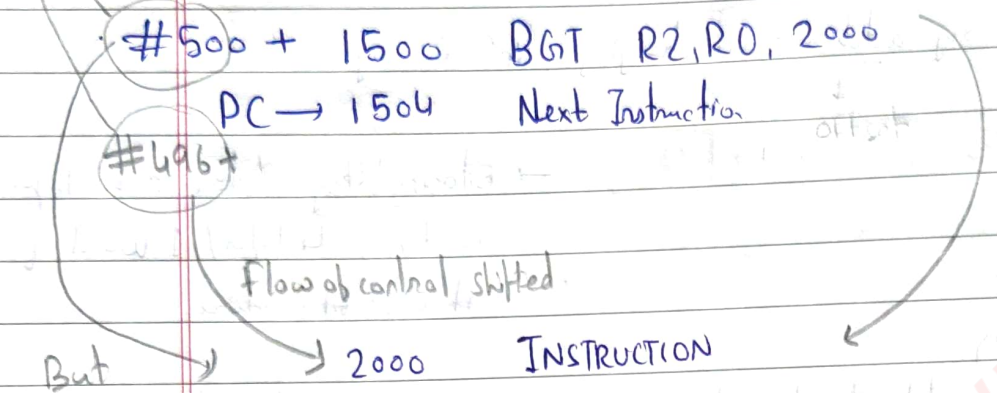
6) PC Relative addressing modes:

→ We know Program Counter contains the address of the next instruction that has to be executed.

Suppose We have instruction: BGT R2, R0, 2000

PC → next instruction

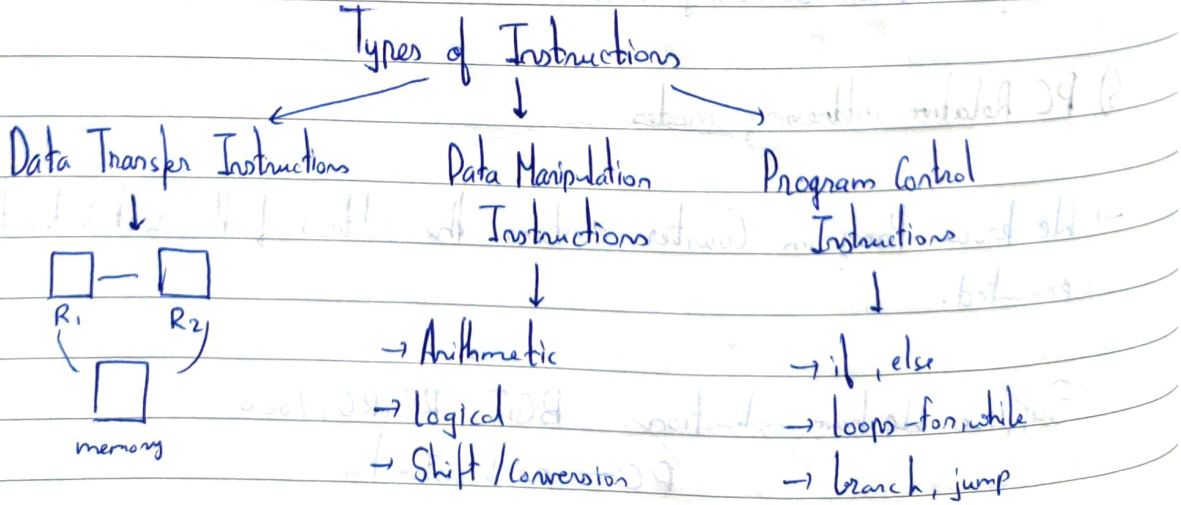
Suppose we have an instruction :



During the course of executing this instruction, if the condition is found to be true, I have to jump to address 2000, so there has to be change in flow of control.
The normal flow of control would be the next instruction but now I have to change the flow of control from this next instruction to the instruction at 2000.

So, this is the meaning of PC Relative addressing mode. Instead of specifying the memory address if there is a Branch instruction, we specify the offset. This offset has to be added to the value of PC, so we need to take care that the PC has already incremented and is pointing to the next instruction.

Types of Instructions in General Purpose Computer:



⇒ Data Transfer Instructions:

1) MOV Ex: MOV R1, R2

2) LOAD Ex: LD R1, 2000 → memory

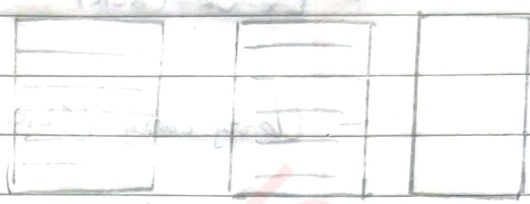
3) STORE Ex: STA 2000, R1 (2000 → memory)

3) EXCHANGE Ex: XCHG R1, R2 (Swap data of R1 & R2)

5) INPUT Ex: IN memory, input device

6) OUTPUT Ex: OP output device, memory

7) Push
8) Pop
→ when memory is implemented as Stack. (zero-addressing mode)



⇒ Data-Manipulation Instructions:

1) Arithmetic Instructions

Add, Sub, Mul, DIV, INC, DEC, add with carry, sub with borrow, negate

2) Logical Instructions (Operations on Binary)

- Complement (COM or NOT) ⇒ 0101 $\xrightarrow{COM}$ 1010 ⇒ XOR 0101 / 1111 / 1010

- Clear (CLR)

- Logical AND and OR ⇒ If in 1011, I want to change all bits to zero → AND to one → OR

⇒ If 1010 is even or odd (lsb=0)
AND 1010 / 0001 ⇒ [odd]

1011 AND 0000 / 0000

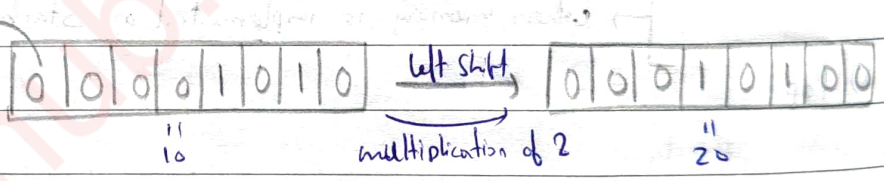
1011 OR 1111 / 1111

- Ex-OR (XOR) ⇒ Modulo 2 function 10 ⇒ 1+0=1/.2=1 11 ⇒ 1+1=2/.2=0
- Clearg carry (CLRC) ⇒ [X] → 0 (→ Flag bit)
- Set carry (STC) ⇒ [Ø] → 1
- Complement carry (CMC) ⇒ [Ø] → 1 or [X] → 0
- Enable Interrupt (EI) → enabling to stop a running program to perform an program. (flag)
- Disable Interrupt (DI)

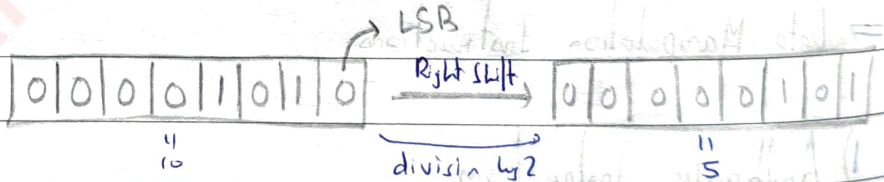
3) Shift Instructions: (Instruction Format: | AM | OPcode | Type of Shift | Operands (count) |)

Generally used when shifting the bits of operands stored in Registers.

- Logical Shift Left



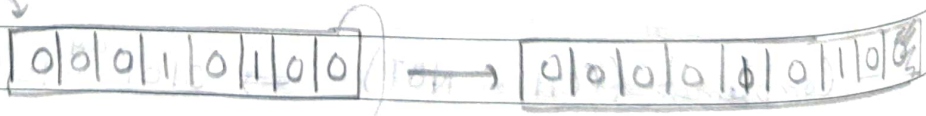
- Logical Shift Right



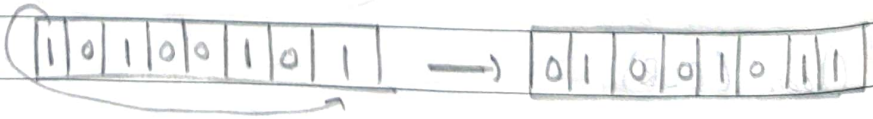
- Arithmetic Shift Right

- Arithmetic Shift left same as Logical Shift left

- Rotate Right



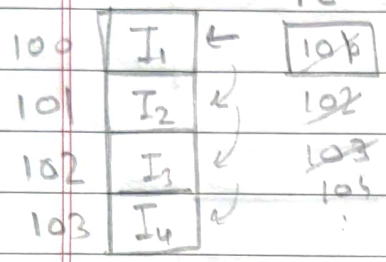
- Rotate left



- Rotate Right through carry

- Rotate left through carry

⇒ Program Control Instructions:



Branch Instructions

Sequential	Unconditional	Conditional
	Jmp 2000	BE R_1, R_2, 2000 If $R_1 = R_2$ then only
	B 3000	BNZ R_1, 2000
	Skp	
	Call & return → same as C function	

★ Types of CPU Organisations :-

⇒ Single Accumulator Organization :

1-address instruction → length of instruction increases (use minimum registers) (min cost) (slow speed)

Ex: X = A+B

```
LDA A    → stored A in accumulator
ADD B    → add A+B & stored it in accumulator
X ← AC   → X takes the value from AC
```

⇒ General Register Organization :

2 or 3-address instruction → length of instruction decreases, max cost (more registers used), fast

Ex: X = (A+B) * (C+D)

```
ADD R1, A, B    R1 ← (A) + (B)
ADD R2, C, D    R2 ← (C) + (D)
MUL X, R1, R2   (X) ← R1 * R2
```

⇒ Register Stack Organisation: (Zero-Addressing) (Expensive because of no. of Registers) but faster

Full	Empty
1	0
0	1

Push

SP ← SP+1

M[SP] ← DR

If (SP==0) then (Full ← 1)

Empty ← 0

POP

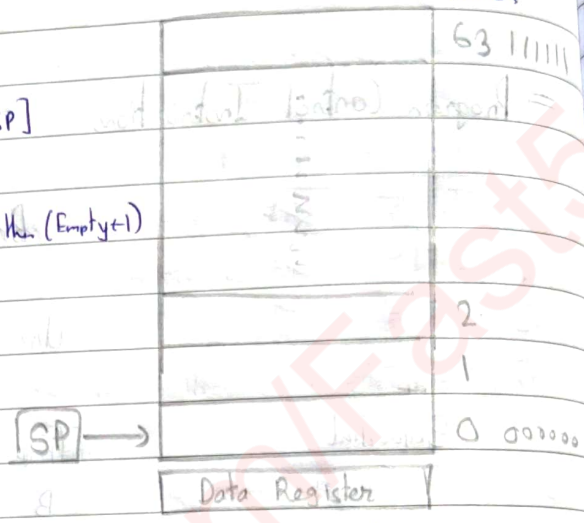
DR ← M[SP]

SP = SP-1

If (SP==0) then (Empty ← 1)

Full ← 0

Ex: (a+b) * (c+d)



Push a → address 1

Push b → address 2

add → Pop from add 1 & add 2, adds, push in add 1

Push c → add 2

Push d → add 3

add → pop from add 2 & add 3, add, push in add 2

MUL → pop from add 1 & add 2, mult, push in add 1

If I push When SP reaches to the address 63 and now if I push another element at the address 64 (1000000) and it pushes the element at address 0 (000000), thus the ignore stack becomes full

⇒ Memory Stack Organisation: (Slow) (Zero Addressing)

Push

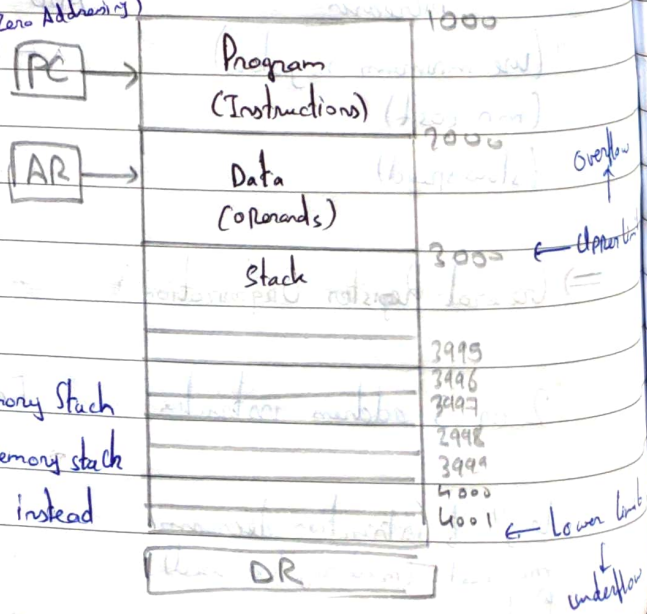
SP ← SP-1

M[SP] ← DR

POP

DR ← M[SP]

SP ← SP+1



Difference between Register Stack and Memory Stack

1) The operations in both are same but the memory stack is implemented using computer memory instead of the CPU register array.

Q A certain architecture supports indirect, direct and register addressing modes for use in identifying operands for arithmetic instructions. Which of the following cannot be achieved with a single instruction?

(A) Specifying a register no. in instruction such that register contain value of an operand that will be used by operation (Register AM)

(B) Specifying a register no. in instruction such that register will serve as destination of operation's output. (Register AM)

(C) Specifying an operand value in instruction such that value will be used by operation (Immediate AM)

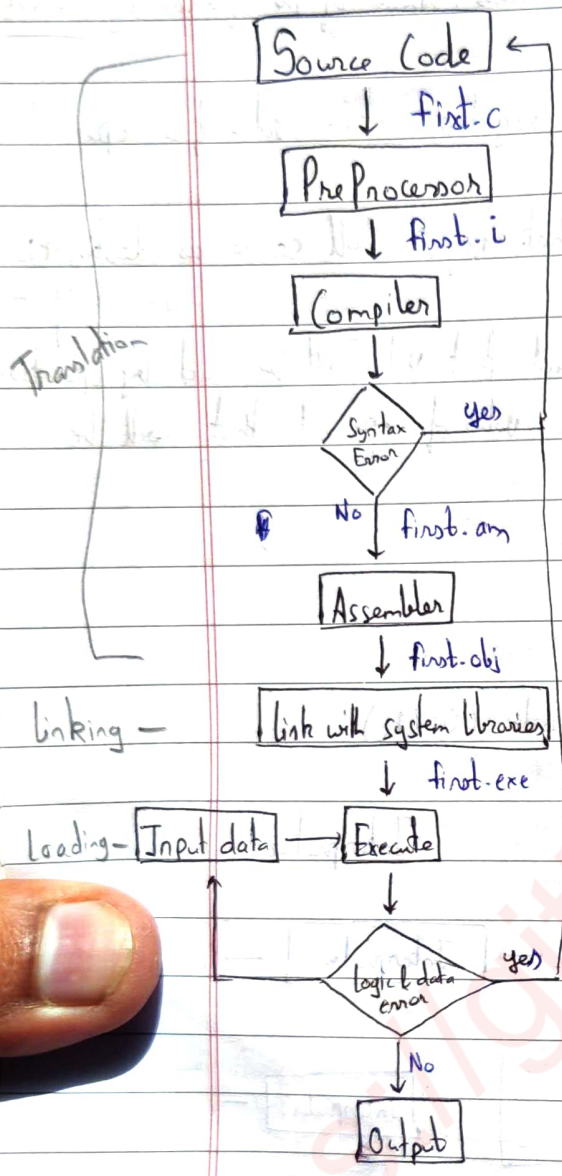
(D) Specifying a memory location such that it contains value of operand that will be used by operation. (Direct AM)

★ Translating Code :-

Assembly language → [assembler] → machine code ($0s$ & $1s$)

Compiler	Interpreter
HLL → [Compiler] → machine code	HLL → [Interpreter] → machine code
[file] → [Compiler] → object code	[file] → [Interpreter] → output
first.c → first.exe (output)	
→ Compiler only translates.	→ Interpreter translates as well as executes
→ Faster	→ Slower (because it performs both)
→ You will get all errors & warnings in the end of compilation.	→ Here you will get errors and warnings line by line as it runs line by line (stop at that line)
→ So, finding error is difficult.	→ So, finding error is easy.
→ Requires more memory	→ Requires less memory

★ Execution of a C program:-



1) Preprocessor: `#include <stdio.h>` → printf, scanf
`#include <math.h>`
`#define N 100`
Compiler cannot read instructions starting with '#'. so, what preprocessor do, it will replace these files with declaration of that header files.
(This process is called expanding source code)

2) Assembler: Converts assembly code to object code

3) Linker: It will link all object code into 1 file and link it with system libraries (definitions of predefined functions)

4) Loader: Now, the file is ready to execute but before executing, all the files/program has to be loaded into the main memory (from hard disc to RAM). Then only CPU can access that file from main memory and can execute.

Static loading

Dynamic loading → load the files which are needed at that time

★ Assembly Language:-

⇒ Introduction:

→ Machine instructions are represented by patterns of 0s and 1s.
→ Such patters are difficult to deal with while preparing programs.
→ Therefore, we use symbolic names (Mnemonics) to represent the patterns

→ These symbolic names (Mnemonics + Rules) constitute to Assembly Language.

→ Each assembly language is specific to a processor.

→ Assembly Language Format:

Label	Opcode	Operands	Comments
↓ memory addrs of particular instruction	↓ describes operation to be performed	↓ source & destination data	↓ description of what the instruction is doing

Assembly language (Source Program) —Assembler→ Machine code (Object Program)

→ Assembler is a software / program written in machine language.

→ If this assembler has to convert assembly language program into machine language, again the assembler also has to be loaded into the RAM.

→ This assembler program is also a sequence of machine instructions in 0s and 1s.

→ This assembler program will read the user program one by one, it will analyse it and it will convert that assembly language ~~prog~~ to machine language program.

→ So, the user program which is written in assembly language is called the source program and the machine language program is called the object program.

⇒ Assembler Directives: (Pseudo Instruction):

→ If the programmer wants to convey any information to the assembler, it can do so by means of Assembler Directives.

→ So, these are instructions to the assembler while it is converting a source program to an object program.

Ex: ✓ EQU → assembler directive which assigns names to numerical values

X EQU 20 (not to be part of the final object program)

wherever in the program it will find an X it will replace it by 20

✓ ORIGIN → this tells the assembler that the next code segment which is coming after this should start from a particular memory address

ORIGIN 200 → this tells the assembler that next code segment will start at memory address 200

3) DATA WORD /DW → This will reserve space equivalent to a word length for the variable

MUTIPLIER DW 480 → This will tell the assembler that a space equivalent to a word length has to be reserved for this multiplier and it is to be initialised with a value 480

4) RESERVE → to reserve some space
5) DB → to reserve space equivalent to a byte
6) END → tells the assembler that this is the end of the program module & the assembler will ignore any instruction that is written after this directive
7) PUBLIC → tells this assembler that the variable which has been declared as public, this variable can be accessed from other modules also
8) EXTERN → tells the assembler this variable is available in other object modules and should be accessed from there.

⇒ Assembly Process:

→ The assembler translates source programs written in an assembly language into object programs that comprise machine instructions.

So,
→ It generates the binary encoding for the opcode and other instruction fields.
→ It also recognizes directives that specify numbers and characters and directives that allocate memory space for data areas.
→ It also replaces Address labels to assigned values based on their position relative to the beginning of an assembled program.
→ It also keeps track of all names and their corresponding values in a symbol table.

⇒ Two-pass Assembler:

→ A problem arises when a name appears as an operand before its value is defined.

Forward Referencing Problem: If a forward branch is required to an address label that appears later in the program.

Ex:

	Label	Opcode	Operands	Length	ILC (Instruction location counter)
	MAY:	MOV	R1, X	5	100
		MOV	R2, Y	5	105
	ROSE:	MOV	R3, Z	5	110
		CMP	R1, R0	2	112
not yet defined ←		BZ STEP	STEP	4	116
		SET MACRO			
		ADD	R1, R2	2	118
		ADD	R1, R3	2	120
		END M			
		MUL	R1, R2	4	124
		SET			
	STEP:	JMP MAY		~~8~~ 2	

→ A commonly-used solution is to have the assembler scan through the source program twice.

1) First Pass

→ Creates the symbol table (where it will add the symbols & values assigned to them)
→ Stores macros and expands them (stores either in the symbol table or in a separate macro table)
→ For address labels, determines value of each name from its position relative to the start of the source program.

2) Second Pass

→ Once all the values have been figured out and stored in a symbol table or a macro table, the second pass will start and each name that is encountered will be checked in the symbol table and replaced by its corresponding numerical value.

Ex:	Label	Opcode	Operands	Length	ILC (Instruction Location Counter)
	MAY:	MOV	R1,X	5	100
		MOV	R2,Y	5	105
	ROSE:	MOV	R3,Z	5	110
		CMP	R1,R0	2	112
		BZ	STEP	4	116
		SET	MACRO		
		ADD	R1,R2	2	118
		ADD	R1,R3	2	120
		ENDM			
		MUL	R1,R2	4	124
		SET			
	STEP:	JMP	MAY	52	130

ADD R1,R2 2 126
ADD R1,R3 2 128

First Pass: Symbol Table

MAY	100
ROSE	110
STEP	130

Macro Table

SET		→

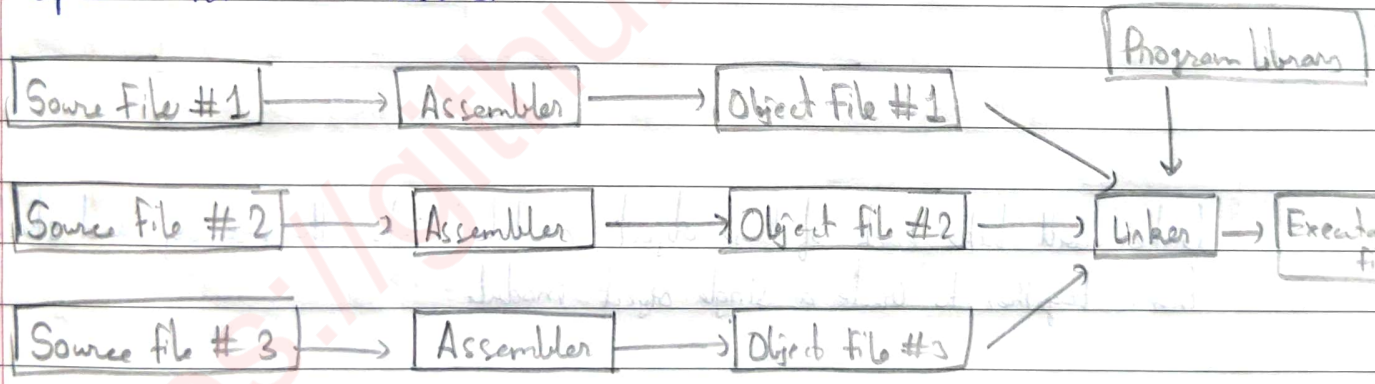
Second Pass : It will convert all instructions in binary format

When it reaches to BZ STEP, BZ → convert to its binary
STEP → convert to its binary equivalent of 130 − 118 = (12) → offset

⇒ Linker:

→ In many cases, a programmer may wish to call subroutines created by other programmers. So, the assembler will create an object file for each of the different source files.

→ And these each individual output file will not be an complete object program because each source file could have references to other source files and so these external references have to be resolved.

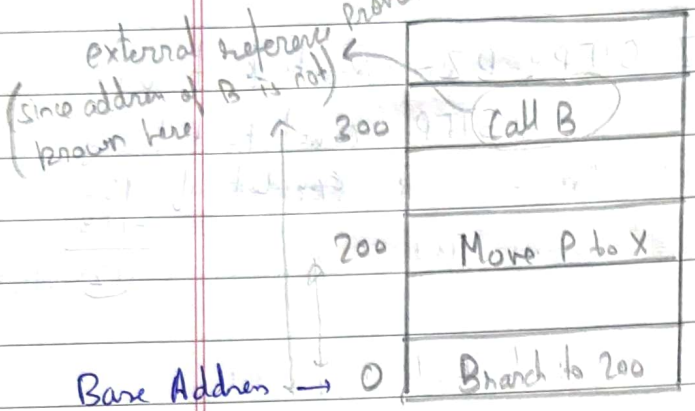


→ The task of the linker is to combine all these object files into 1 object program, so while doing this, it has to resolve external references, so it needs the relative position of the address labels and for references within the object module it needs the length of the object module also.

→ So, all of these information, assembler provides to the linker in the header of the object file.

PTO

Ex: Object Module A : length 400



Object Module B : length 600

500	CALL C
300	MOVE Q to X
0	BRANCH to 300

Object Module C : length 500

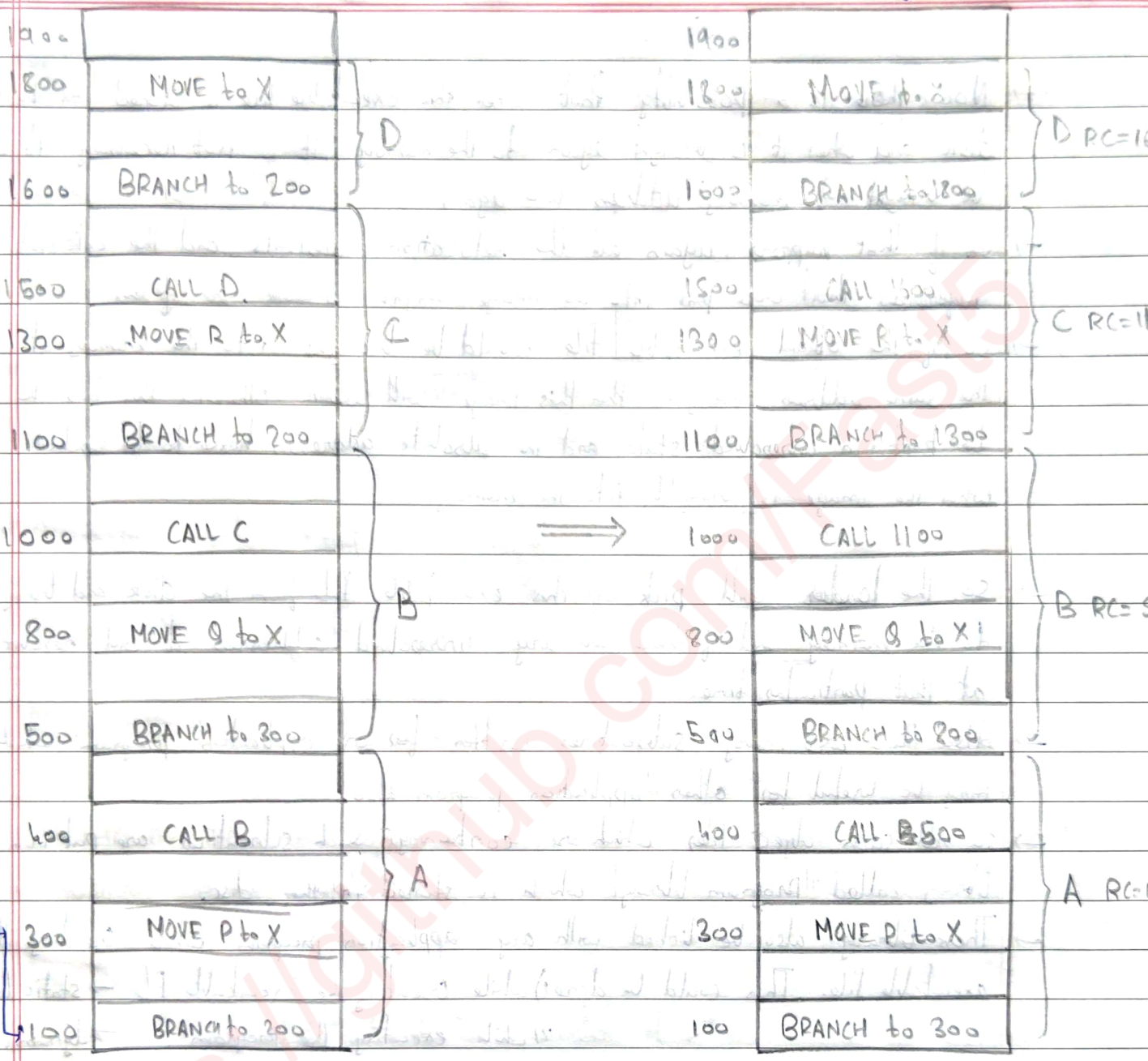
400	CALL D
200	MOVE R to X
0	BRANCH to 200

Object Module D : length 300

200	MOVE to X
0	BRANCH to 200

Linker will take each of these object modules into the memory and stack them together to create a single object module.

final image



→ Since base address of module A is changed from 0 to 100, the instruction has to be also changed i.e. Branch to (200+100). (Relocation)

→ So, In module A relocation constant = 100.

→ Similary In module B relocation constant = relocation constant of A + length of module A

= 100 + 400

= 500

→ Apart from taking care of internal references and internall offsets in the branch instructions, there are some external references which need to be resolved like CALL B → CALL 500.

→ Now, there is a possibility that when this executable file is stored on to the disk and when it is brought again to the memory, it is not necessary it will be brought to memory address too again.

→ So, if that happens, again all the relocation constants and the external references that were put into the image might not work. So, if the

→ So, if the ofjinal executable file would be brought into the memory at the same address everytime then this image will work otherwise this has to be kept in unresolved state and the absolute addresses have to be resolved when the image is brought into the memory.

→ So, the **loader** will pick up this executable file from the disk and bring it to the memory and if there are any unresolved addresses it will resolve at that particular time.

→ Also there are many subroutines written for one application program which may be useful for other application program also.

→ So, all such object files which are containing such subroutines are put in a library called program library which is stored in the disc.

→ These libraries also be linked with any application program while creating the executable file. This could be done 1) while creating the executable file → static linking
2) while executing the program → dynamic linking

★ Instruction Execution :-

1) Fetch the instruction from the memory

→ To fetch from the memory, the address from the PC will be taken & put on the address bus via the MAR (memory address register).

2) Increment the PC (program counter)

3) Decode the instruction to determine the operation to be performed.

→ The instruction from the memory will be brought through the data bus & placed into the IR (instruction register) where it will be decoded & the operation that is to be performed will be determined.

3) Read source operands

4) Perform operation / compute effective address
   - Arithmetic / logical
   - LOAD / STORE instruction

5) Access memory for read / write operation

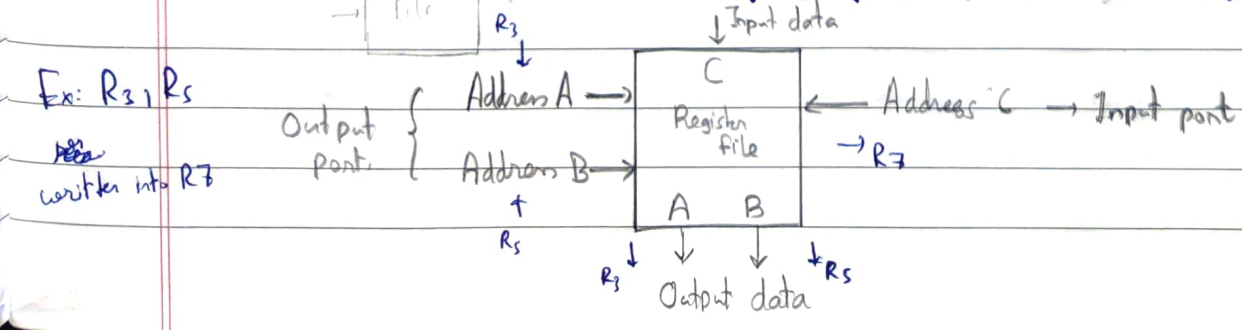
6) Write results in destination register.

Ex: LOAD Instruction : Load R5, X(R7) EA ← X + [R7]

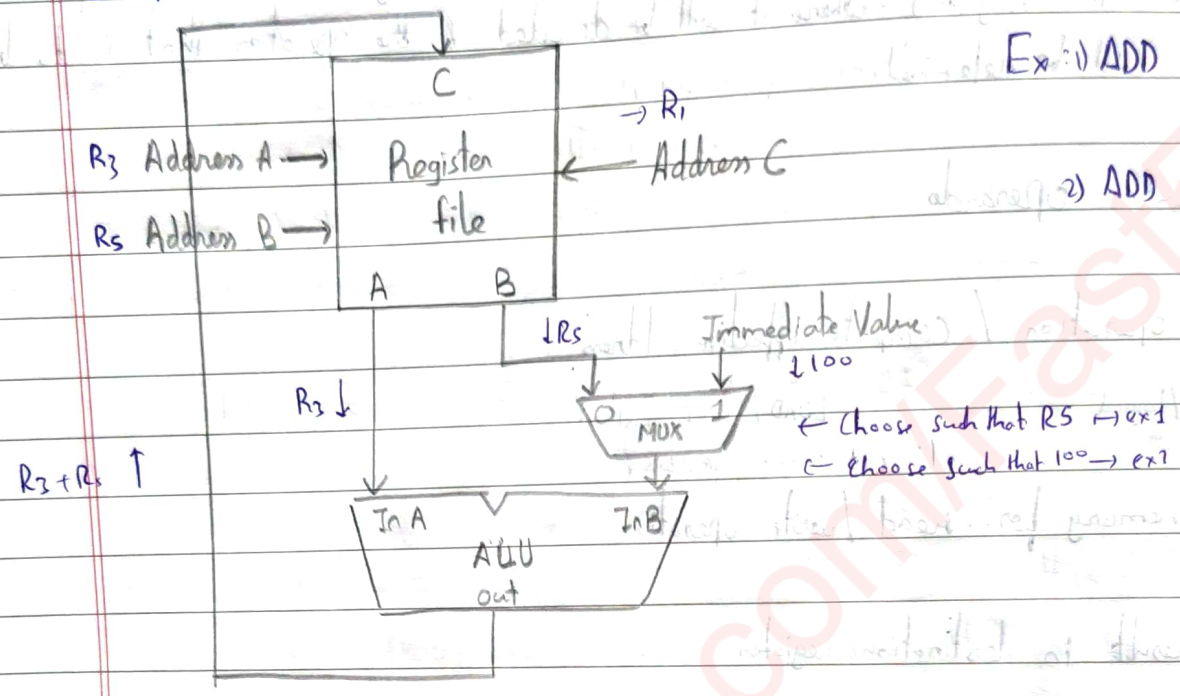
- Fetch the instruction from the memory
- Increment the program counter
- Decode the instruction to determine the operation to be performed.
- Read register R7
- Add the immediate value X to the contents of R7.
- Use the sum X + [R7] as the effective address of the source operand, and read the contents of that location in the memory.
- Load the data received from the memory into the destination register R5

⇒ Register File and ALU:

Register file refers to the collection of the general purpose registers of the processor.



Computation:



Ex: 1) ADD R1, R3, Rs

2) ADD R1, R3, #100

← Choose such that Rs → ex1

← Choose such that 100 → ex1

⇒ 5 stage organization:

→ Divide the fetch-decode-execute cycle in stages.

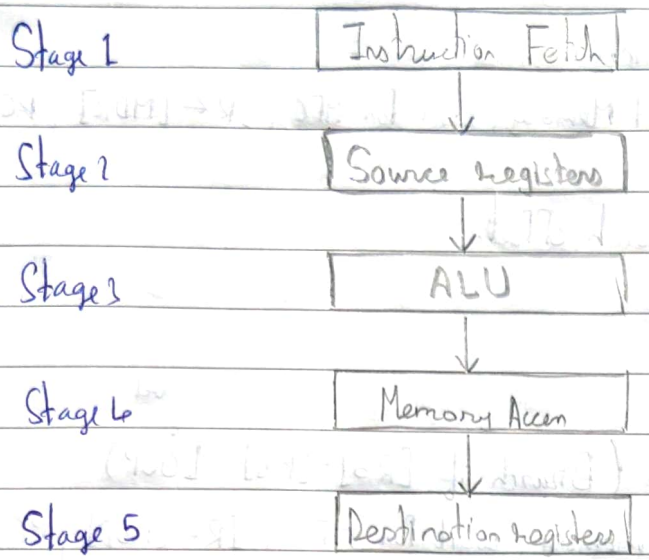
→ Assuming that the processor has five hardware steps, execution of each instruction is divided into five steps:

1) Fetch the instruction and increment the program counter.
2) Decode the instruction and read the source operands
3) Computation (ALU)
4) Access the memory
5) Write result in register file

→ The actions taken in each of the five stages are completed in one clock cycle.

→ It is necessary to insert registers between stages (inter-stage buffers)

MFC → Memory Function Completed



Ex: 1) Add R3, R4, R5 (assuming length of the current instr. is 4; to point to next instr.)

a) MAR ← [PC], Read memory, wait for MFC, IR ← [MDR], PC ← [PC] + 4

b) Decode instruction, RA ← [R4], RB ← [R5] (Inter Stage Register)

c) RZ ← [RA] + [RB]

d) RY ← [RZ] (Here no memory access is required, so simply the contents of the previous inter-stage register is pass to next inter-stage register.)

e) R3 ← [RY], END

2) LOAD R5, X(R7)

a) MAR ← [PC], Read memory, wait for MFC, IR ← [MDR], PC ← [PC] + 4

b) Decode instruction, RA ← [R7]

c) RZ ← [RA] + X (Effective address)

d) MAR ← [RZ], Read memory, wait for MFC, RY ← [MDR]

e) R5 ← [RY]

3) Store R6, X(R8)

a) MAR ← [PC], Read memory, wait for MFC, IR ← [MDR], PC ← [PC] + 4

b) Decode instruction, RA ← [R8], RB ← [R6]

c) RZ ← [RA] + X, RM ← [RB]

d) MAR ← [RZ], MDR ← [RM], write memory, wait for MFC

e) No action

4) Unconditional Branch

a) PCout, MARin, Read, Select4, Zin
b) Zout, PCin, Yin, WMFC
c) MDRout, IRin
e) Offset-field-of-IRout, Add, Zin
f) Zout, PCin, End

a) MAR ← [PC], Read Memory, wait for MFC, IR ← [MDR], PC ← [PC]+4
b) Decode instruction
c) PC ← [PC] + Branch offset
d) No action
e) No action

5) Conditional Branch (Branch if [R5]=[R6] LOOP)

label ↓

a) MAR ← [PC], Read memory, wait for MFC, IR ← [MDR], PC ← [PC]+4
b) Decode instruction, R_A ← [R5], R_B ← [R6]
c) Compare $[R_A]$ to $[R_B]$, if $[R_A]=[R_B]$, then PC ← [PC] + branch offset
d) No action
e) No action

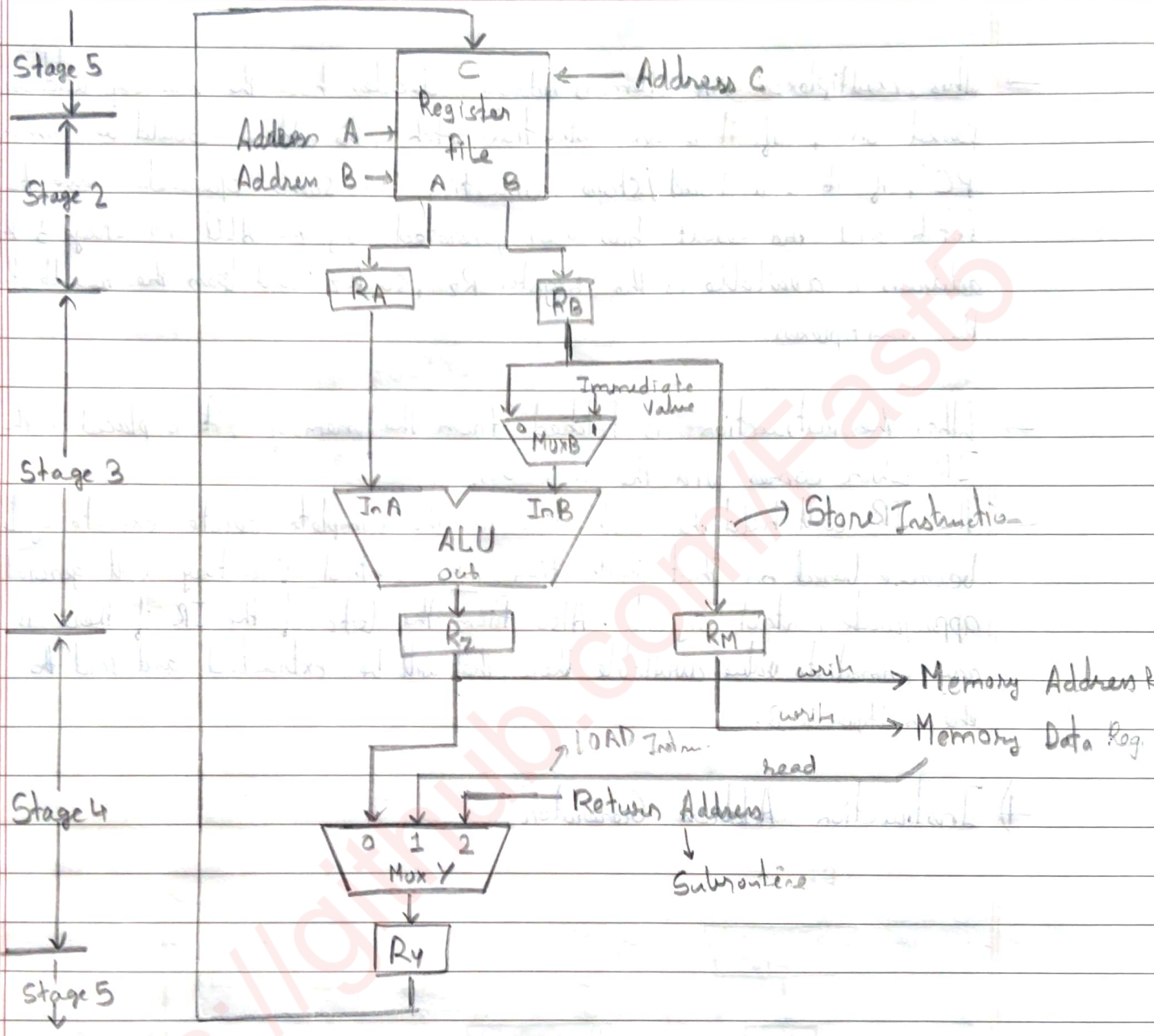
6) Subroutine call (Call-Register R9)

a) MAR ← [PC], Read memory, wait for MFC, IR ← [MDR], PC ← [PC]+4
b) Decode instruction, R_A ← [R9]
c) PC_TEMP ← [PC], PC ← $[R_A]$
d) R_Y ← [PC_TEMP]
e) Register LINK ← $[R_Y]$

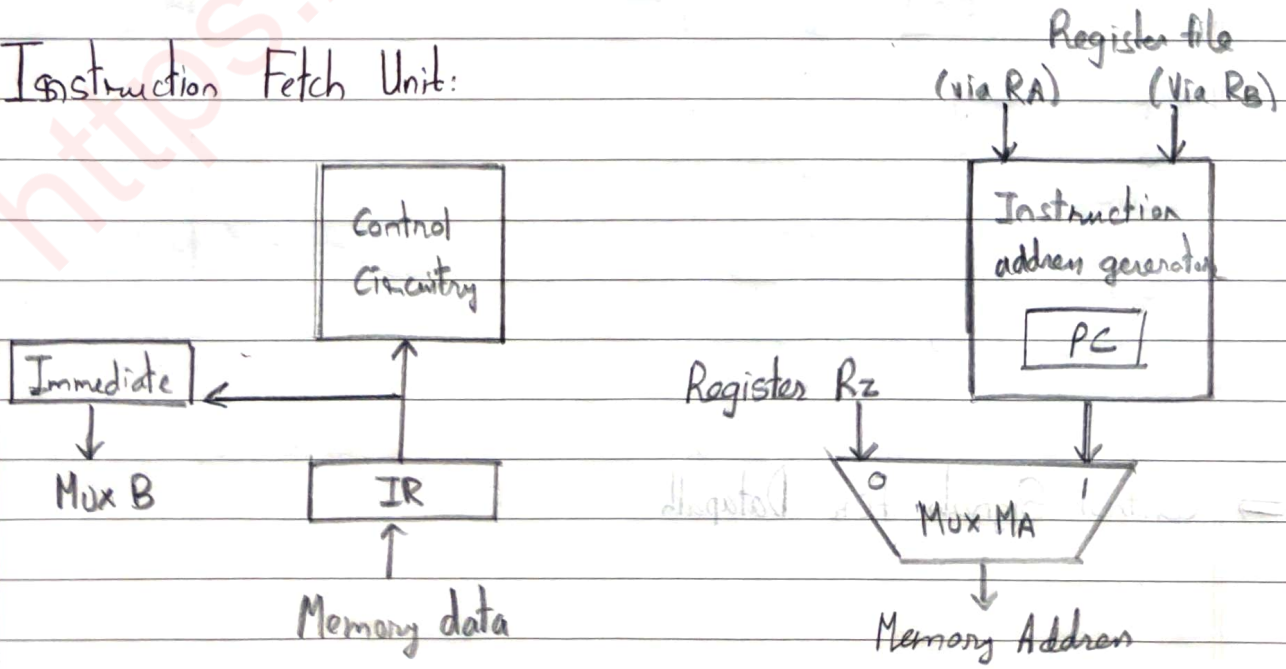
Datapath: OR Add (R3), R1

a) PCout, MARin, Read, Select4, Add, Zin
b) Zout, PCin, Yin, WMFC
c) MDRout, IRin
d) R3out, MARin, Read
e) R1out, Yin, WMFC
f) MDRout, Select Y, Add, Zin
g) Zout, R1in, End

⇒ Datapath:



⇒ Instruction Fetch Unit:

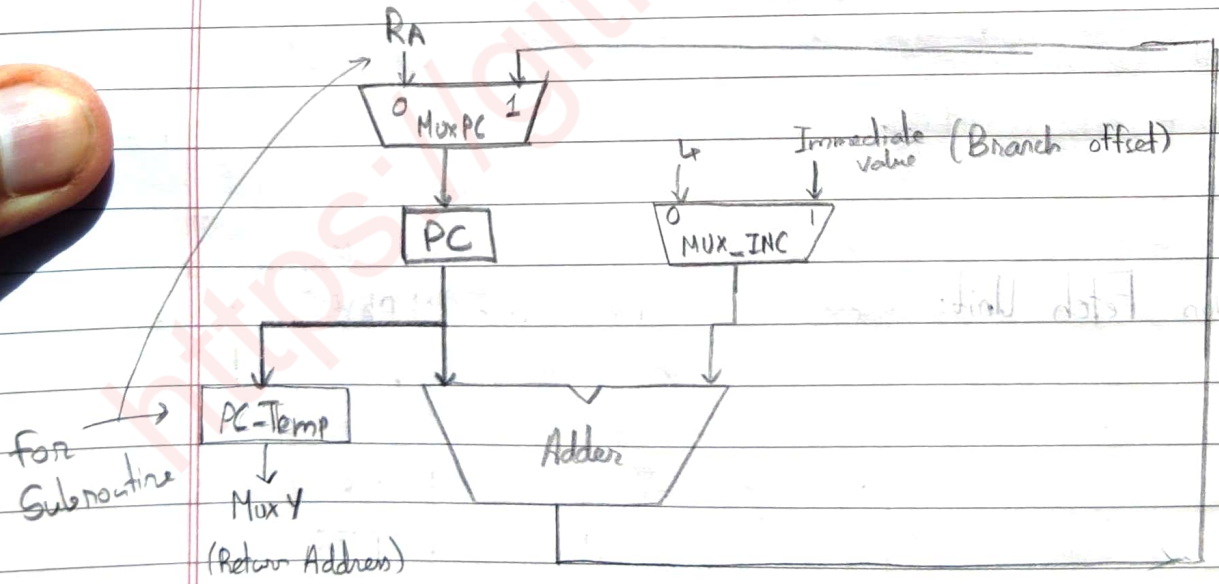


→ This multiplexer MA (Memory Address) decides how the memory address will be passed on, if it is an instruction fetch, the address would be coming from PC, if it is a load/store instruction or some operands have to be fetch and the result has been generated by the ALU in stage 3 that address is available in the register R_Z, so R_Z will pass the result through this multiplexer.

→ When the instruction is fetched from the memory, it is placed in the IR which comes via the data bus.

→ The IR will hold the instruction for the complete decode-execute cycle because based on that instruction, the control circuitory will generate appropriate control signals. Also from the bits of the IR if there is any immediate value available there, that will be extracted and send to the multiplexer B.

1) Instruction Address Generator



⇒ Control Signals for Datapath:

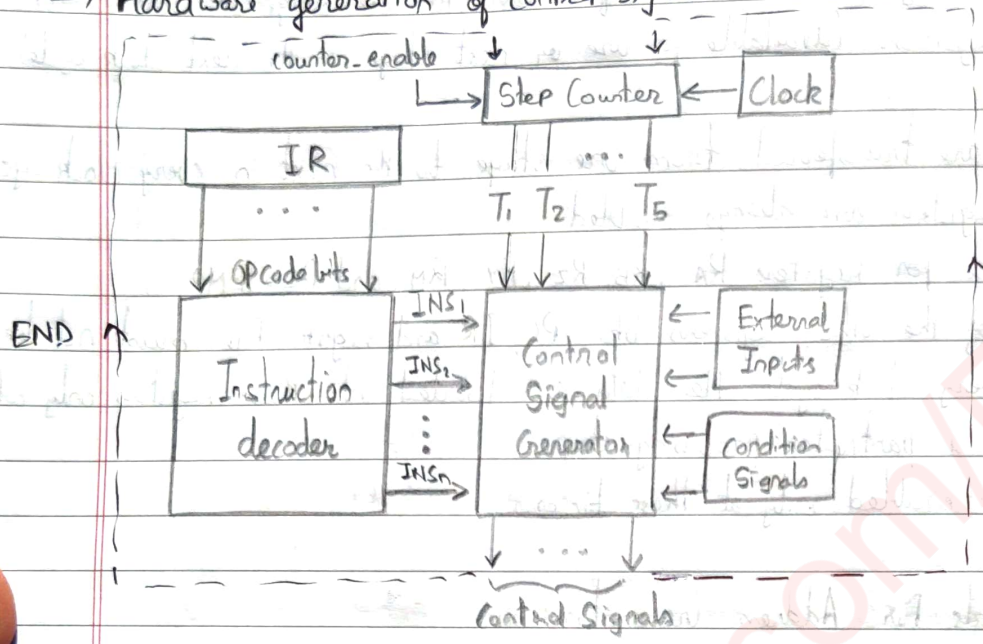
→ In each clock cycle, the results of actions in one stage are stored in inter-stage registers. (available for use by next stage in the next clock cycle).

→ Since data are transferred from one stage to the next in every clock cycle, inter-stage registers are always enabled.
→ This is the case for registers R_A, R_B, R_Z, R_Y, R_M and PC-TEMP.
→ The contents of the other registers like PC, IR and register file must not be changed in every clock cycle. New data are loaded into these registers only when called for in a particular processing step.
→ They must be enabled only at those times.

⇒ Control Signals For Address Generator:

⇒ Control Signals for Processor-memory interface:

⇒ Hardware generation of control signals:



→ As we have seen earlier that in each step of execution of an instruction, certain control signals are required. So, how are these signals generated? By means of hardware.

→ As you can see in the above figure, this control signal generator which is generating the control signals is taking inputs from multiple units like instruction decoder, step counter, external input and so on.

→ This step counter keeps track of the control steps, it is getting inputs from the clock, so the clock pulses are being sent as an input to the step counter and each state that is generated corresponds to one control step.

→ If there is an implementation which just requires 5 stage for each instruction then $T_1, T_2, \ldots, T_5$ are generated corresponding to each step.

→ This will be enabled when the counter_enable is set.

→ The instruction decoder generates an output signal depending upon the contents of the IR, so the opcode bits are read and the IR will set that instruction high.

→ The external inputs correspond to any interrupt request from the input-output

devices.

→ They also correspond to the wait for MFC signal that we receive from the memory in case there is memory read/write operation.

→ The condition signals represent the flags.

→ One of the signals generated by Control signal generator is RUN signal.

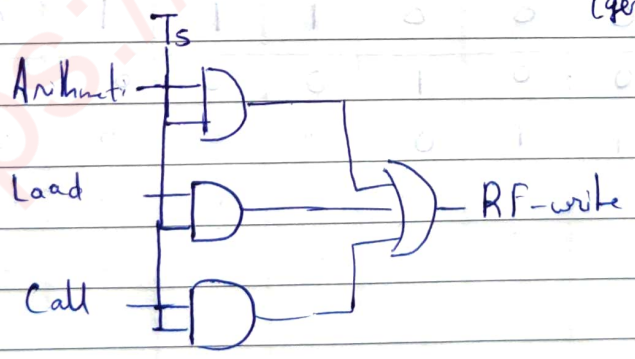
→ The RUN signal is sent as an input to the step counter and this causes the counter to be incremented by 1 at end of every clock cycle.

→ When RUN=0, counter will stop counting which is needed wherever there is wait for MFC after which only counter will be increment

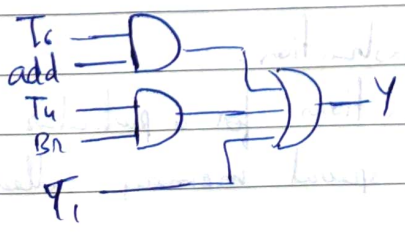
→ Another important signal is END signal which corresponds to end of execution of a particular instruction.

→ This starts a new instruction fetch cycle i.e. it will reset the step counter, so the step counter will again be pointing to T_1 and the next instruction fetch cycle will begin.

Ex: 1) Register File - write (Control signal) → require (generated) T_5 — Arithmetic, laad, call



2) $Y - T_1$ / $Y - T_4 - Bn$ / $Y - T_6 - Add$



i.e. by software

⇒ Microprogrammed Control (alternative to generating signals by hardware)

→ Desired setting of the control signals in each step determined by a program stored in a special memory.
→ This control program is called a microprogram.
→ Stored on the processor chip in a small and fast memory called the control store.

Control word: word whose individual bits represent control signals for each step in sequence.

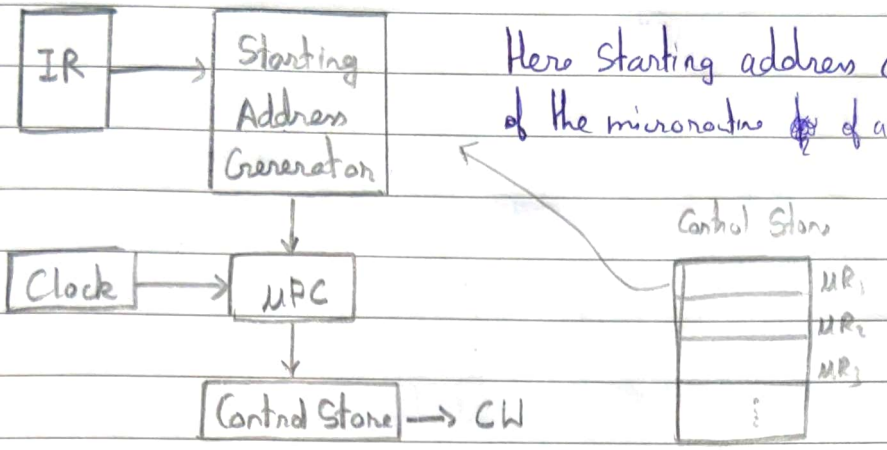
Microroutine: Sequence of control words corresponding to control sequence for a machine instruction.

→ Individual control words are referred to as micro instructions.
→ Microroutines for all machine instructions are stored in the control store.

Micro Instruction	PCin	PCout	MARin	Read	MDRout	IRin	Yin	Select	Add	Zin	Zout	R1out	R1in	R2out	WMFC	End
1	0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0
2	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
5	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
6																
7																

Control Word (Each bit of control word is representing 1 control signal)

Each control word is 1 micro instruction.
The whole of set of 7 micro instructions for a particular instruction is called microroutine. which is stored in a special memory called controlled store.

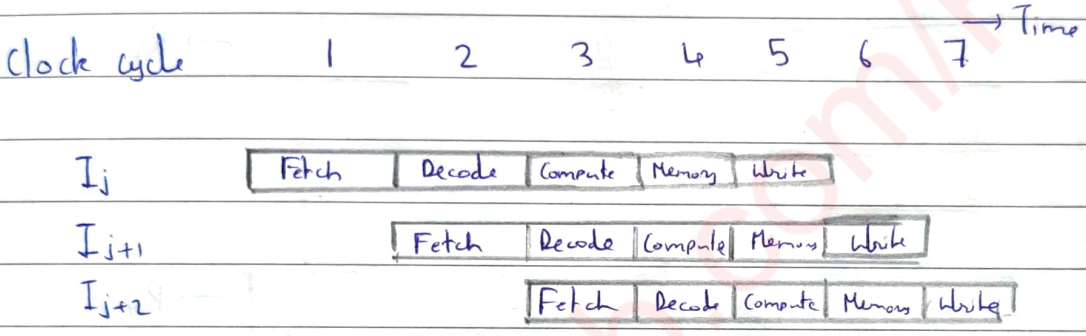


Here starting address corresponds to the addr of the microroutine of a particular instruction

☆ Introduction to pipelining :-

If we want to increase the speed of execution of a program, we can use faster circuit technology to implement the microprocessor & memory or we can arrange the hardware in such a manner so that more than one operation can be performed at the same time.

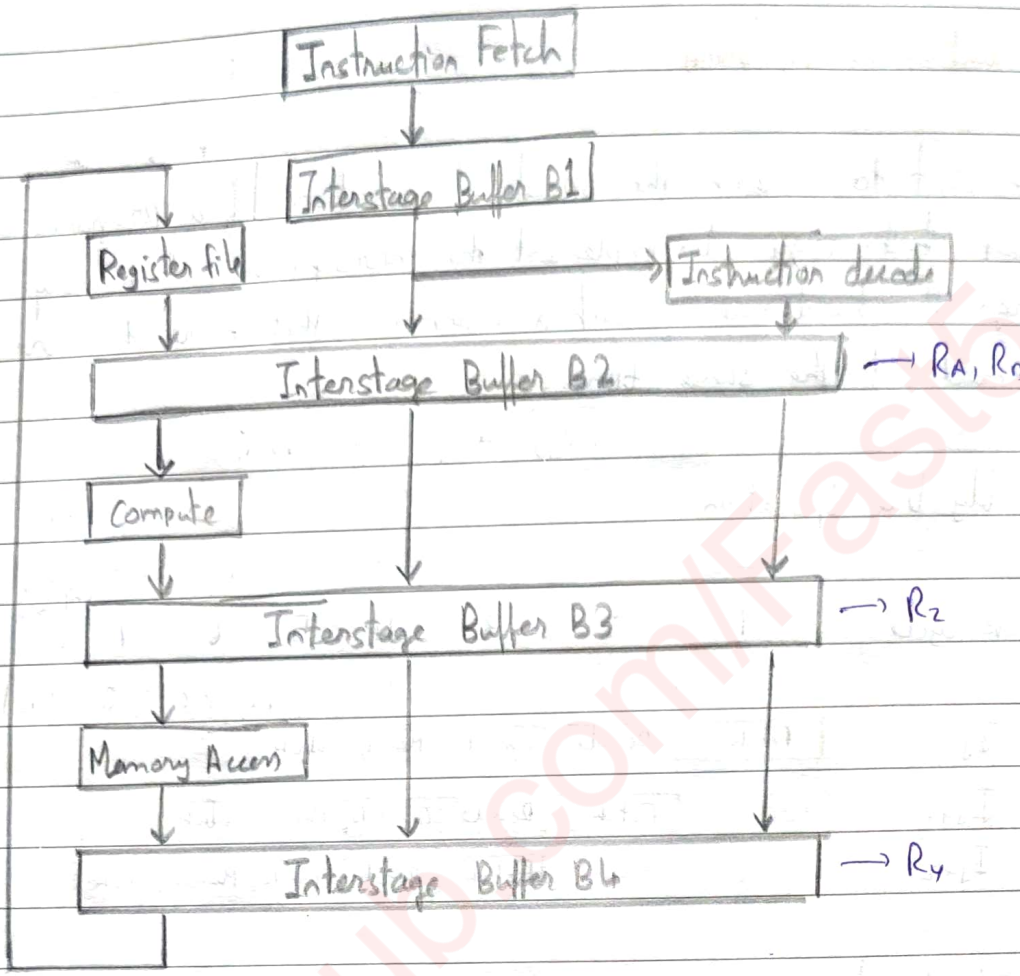
⇒ Assembly line Operation



→ This leads to concurrent execution of instructions. Each stage is available for another instruction and no stage is lying idle

→ It is also assumed that each stages over here, it completes 1 in 1 clock cycle. so, each instruction is coming out in every clock cycle.

→ Though the time that is required to execute one instruction remains the same, so if it was 5 clock cycles, each instruction is still taking 5 clock cycles but every clock cycle, we are having one instruction being completed.



→ Fetch unit will fetch the instruction and at the next clock cycle it will pass onto the interstage buffer B1 – will hold the instruction for the decode stage

⇒ Performance Evaluation of Pipeline:

1) For a non-pipelined processor, the execution time T of a program is given by

$$T = \frac{N \times S}{R}$$

N → dynamic instruction count (Actual no. of instructions)
S → average no of clock cycle it takes to fetch and execute one instruction
R → clock rate in cycle/second

2) Instruction throughput : number of instructions executed / second.

For a non-pipelined execution, the throughput P_{np} is given by

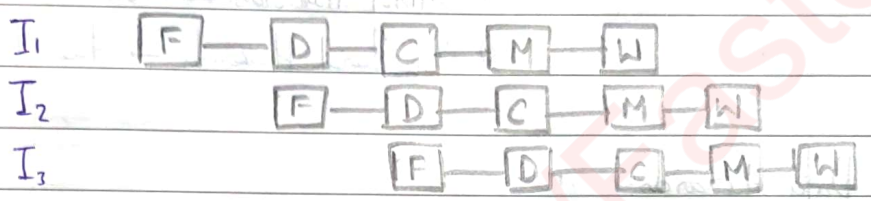
$$P_{np} = \frac{R}{S}$$

5 cycles → 1 instn

$\frac{5}{R}$ secs → 1 instn.

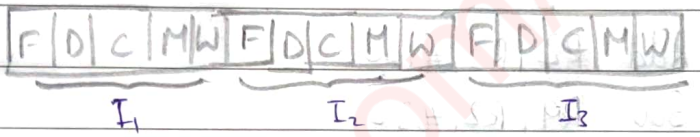
1 sec → $\frac{R}{5}$ instructions

Ex: Pipelined (no stalls)



$\Rightarrow S = 1$

Non-Pipelined:



$\Rightarrow S = 5$

Assume there are 100 instructions (Each 5-stage Instructions)

No. of cycles to execute these 100 instruction i) Non-Pipelined = $100 \times 5 = 500$ clock cycles

ii) Pipelined = $5 + 99 \times 1 \approx 100$ clock cycles

∴ Increase in Performance = $\frac{500}{100} = 5$ times

If n-stage pipeline → % increase = n%

→ n-stage pipeline may increase instruction throughput by a factor of n, so should we use a large number of stage?

→ As the number of pipeline stages increases:
- There are more instructions being executed concurrently.
- There are more potential dependencies between instruction, thus increasing stalls.
- Branch penalty may be larger than one cycle.

⇒ Pipeline Hazards:

The pipeline may have to be stalled due to reasons referred to as hazards.

Types of Hazards: 1) Data Hazards → source operands are not available
2) Control Hazards → due to branch instructions
3) Structural Hazards → due to resource limitations

1) Data Hazards:

Ex: Add R2, R3, #100
Sub R9, R2, #30

Clock cycle	1	2	3	4	5	6	7	8	9 → time
Add R2, R3, #100	F	D	C	M	W				
Sub R9, R2, #30	F	D		C	M	W			

No. of stalls = 5 - 2
= 3

a) Read after Write (RAW) : An instruction refers to a result that has not yet been calculated or retrieved.

Ex: I_1 : ADD R2, R3, R5 (Write)
I_2 : ADD R4, R3, R2 (Read)

b) Write after Read (WAR) : a problem with concurrent execution due to out of order execution.

Ex: I_1 : ADD R4, R1, R5 (Read)
I_2 : ADD R5, R1, R2 (Write) → If executed before I_1, then there is a problem because in I_1 R5 has to read the original value.

Write after Write (WAW): a problem with concurrent execution due to out of order execution

Ex: I_1: ADD R2, R4, R7 (write)
I_2: ADD R2, R1, R3 (write)

} The final value of R2 should be the one updated by I_2 but if I_1 takes more time than I_2, wrong value will be updated

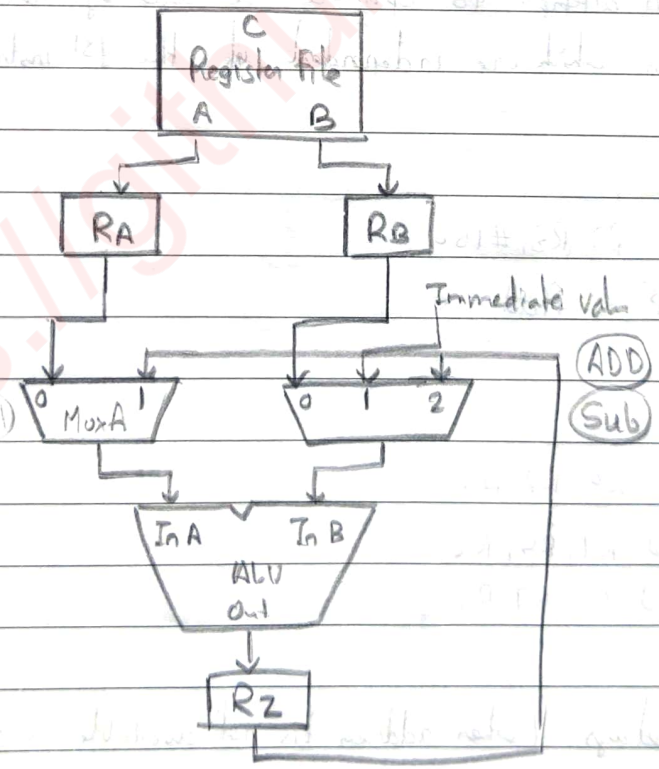
Operand Forwarding (to solve RAW)

Ex: ~~Add~~ R2, R3,

Clock cycle 1 2 3 4 5 6 7 8

R2 (Buffer)

Add R2, R3, #100 | F | D | C | M | W |
Sub R9, R2, #30 | F | D | C | M | W |



Ex: Clock cycle 1 2 3 4 5 6 7 → time

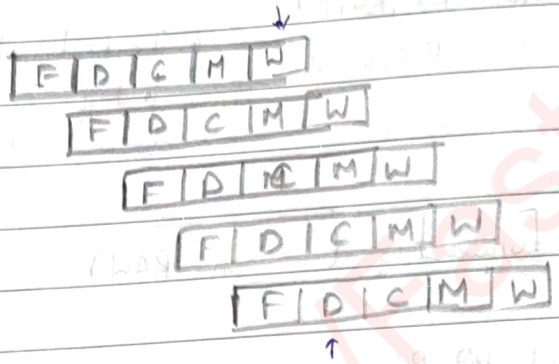
R2 (Buffer)

Add R2, R3, #100 | F | D | C | M | W |
OR R4, R5, R6 | F | D | C | M | W |
Sub R9, R2, #30 | F | D | C | M | W |

Handling data dependencies by compiler

Ex: clock cycle

Add R2, R3, #100
NOP
NOP
NOP
Sub R9, R2, #30



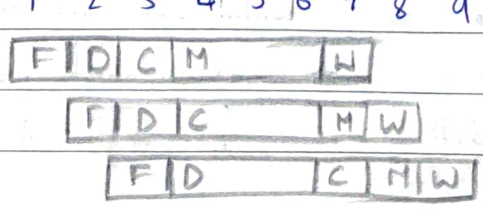
→ Using these NOP, simplifies the hardware implementation of the pipeline.
- However, the code size increases
- Execution time is not reduced as it would be with operand forwarding

→ Compiler can attempt to optimize the code by reordering instructions i.e. the instruction which are independent of the 1st instruction can be replaced with the NOP.

Ex: Add R2, R3, #100
~~NOP~~ ~~Add~~
~~NOP~~
NOP
Sub R9, R2, #30
Add R1, R4, R5
Add R8, R7, R3

Memory delays (when address is not available in cache & need to search in main memory (RAM))

Ex: clock cycle



Ex: Clock cycle 1 2 3 4 R_y 6 7 → time

LOAD R2, (R3) F D C M W

SUB R9, R2, #30 F D C M W

No memory delay is there still there is a stall of 1 cycle even if we are using operand forwarding.

So, compiler can put a independent useful instruction or a NOP inst.

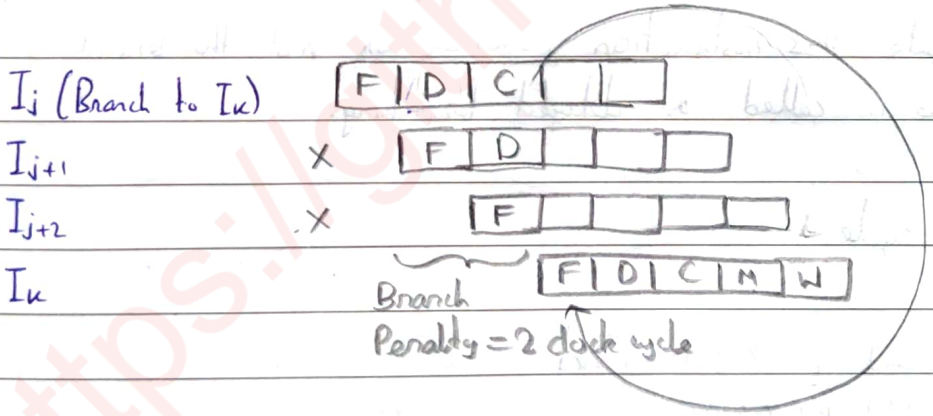
2) Control Hazards

→ Branch instructions can alter the sequence of execution but they must first be executed to determine whether and where to branch.

→ They are referred to as control hazards.

a) Due to Unconditional Branch

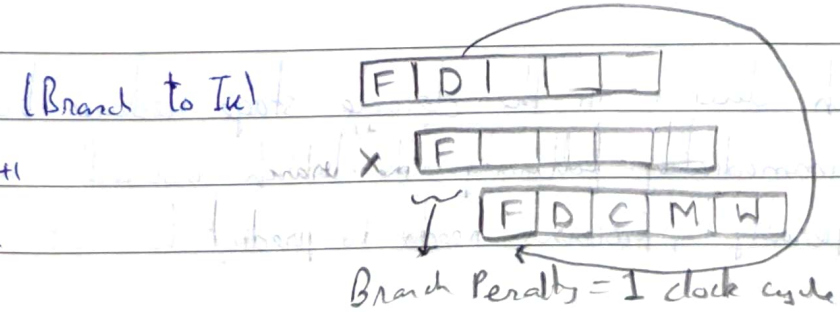
Ex: Clock cycle 1 2 3 4 5 6 7 8 → time



⇒ Compute Stage : PC + offset

This branch penalty can be reduced if we put an offset added at the decode stage only

Ex: clock cycle 1 2 3 4 5 6 7



b) Due to conditional branch

Similar to the previous example where we put an adder into decoder, we can also put a comparator in decoder to reduce the branch penalty.

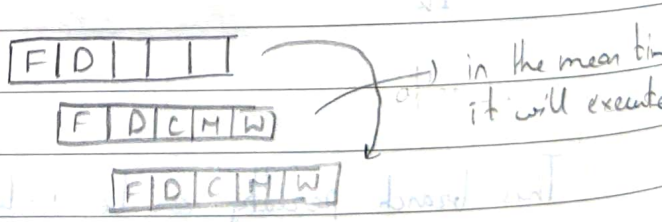
Ex: clock cycle 1 2 3 4 5 6 7 8

I_0: Add R7, R8, R9
I_j (Branch if [R3]=0 to I_k)
I_{j+1}
I_k

→ So, the instruction immediately following the branch instruction is always fetched.
→ The location of the instruction that follows a branch instruction is called the branch delay slot.
→ So, Rather than conditionally discard the instruction in the delay slot can we put an instruction that has to always execute and have the pipeline to always execute this instruction, whether or not the branch is taken.
→ This technique is called as delayed branching.

Ex: clock cycle

I_j (Brand if [R3]=0 to I_k)
I_0 : Add R7, R8, R9
I_k :



Static branch prediction

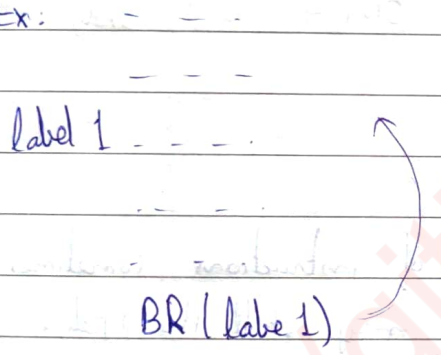
→ Making the branch decision in the decode stage reduces branch penalty but the instruction immediately following the branch instruction is still fetched.
→ To eliminate this penalty, Processor needs to predict to determine which instruction should be fetched.

→ In static branch prediction, use same choice (assume not-taken) everytime a conditional branch is encountered.
→ Considering branch outcomes to be random, half of all conditional branches would be taken resulting in prediction accuracy of 50%.

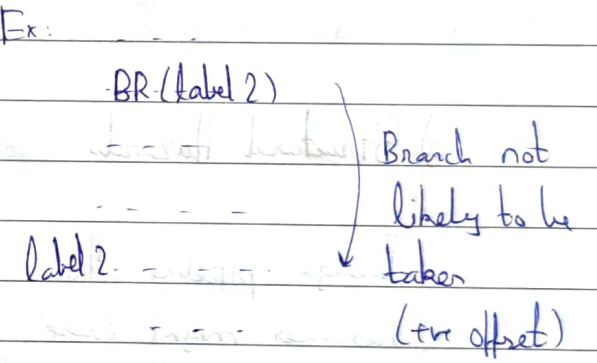
→ A backward branch at the end of a loop is taken most of the time.
→ For a forward branch, branch (not-taken) prediction leads to good prediction accuracy.

→ The processor can determine static prediction by checking the sign of the branch offset.

Ex:



Ex:

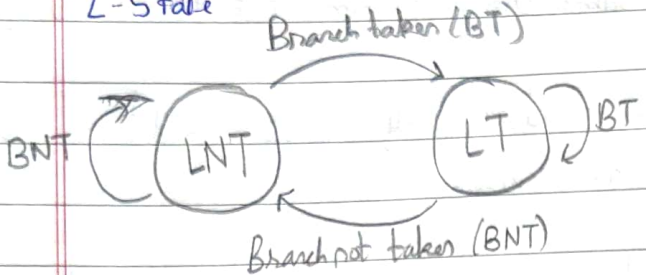


Dynamic Branch Prediction

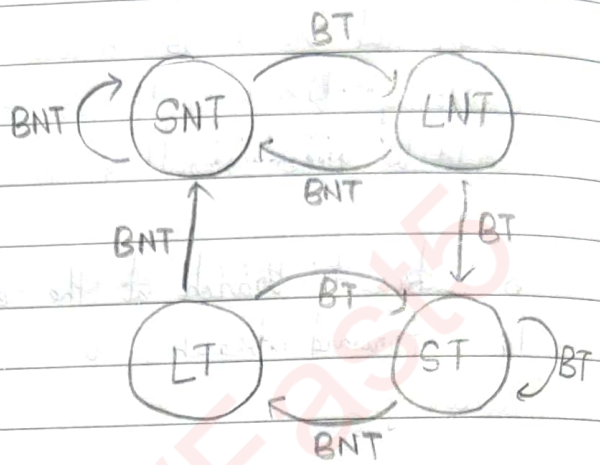
→ Use the actual branch behaviour to predict whether the branch will be taken or not next time. (Unlike Static Branch Prediction where the same prediction is used everytime).
→ The processor hardware keeps track of branch decisions and everytime a branch instruction is executed, it takes every decisions into considerations.
→ It uses the result of the most recent execution of a branch instruction.

Assumption: Next time the instruction is executed, the branch decision is likely to be the same as last time.

2-State



4-State

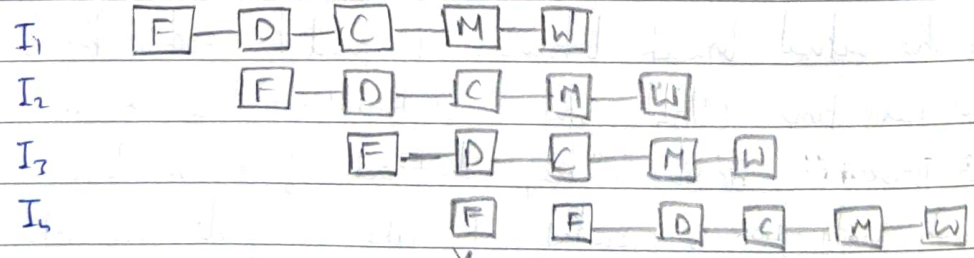


ST → Strongly likely to be taken
LT → Likely to be taken
LNT → Likely not to be taken
SNT → Strongly likely not to be taken

3) Structural Hazards

Though pipeline allows concurrent execution of instructions, sometimes limited resources might cause some hazards which may stall the pipeline, such hazards are called as Structural Hazards. This happens when two instructions which are running concurrently might want to access the same resource at the same time.

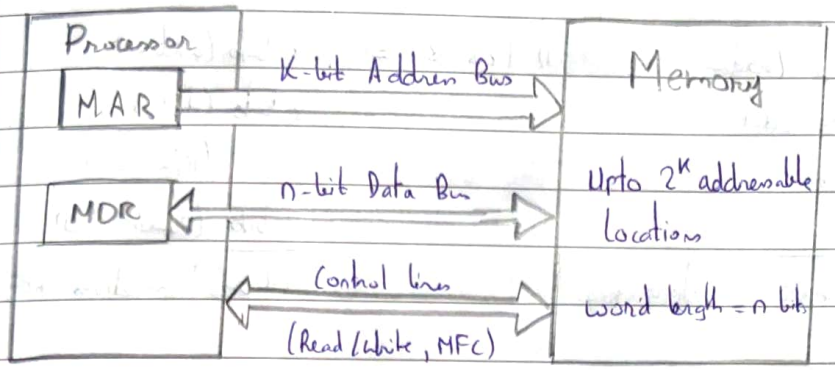
Ex: clock cycle



→ can be resolved if we had a seperate Data cache and instruction cache

↑ I1 ↑ I4

✱ Memory :-



Byte addressable memory (32-bit word)

3	2	1	0	Word 0
7	6	5	4	Word 1
11	10	9	8	Word 2
15	14	13	12	Word 3
.	.	.	.	.
.	.	.	.	.
.	.	.	.	Word $2^{30}-1$

Since each word is byte addressable

∴ Total Bytes in each word = $\frac{32}{8}$

= 4 bytes

∴ Total addressable location = 2^{32} (total bytes)

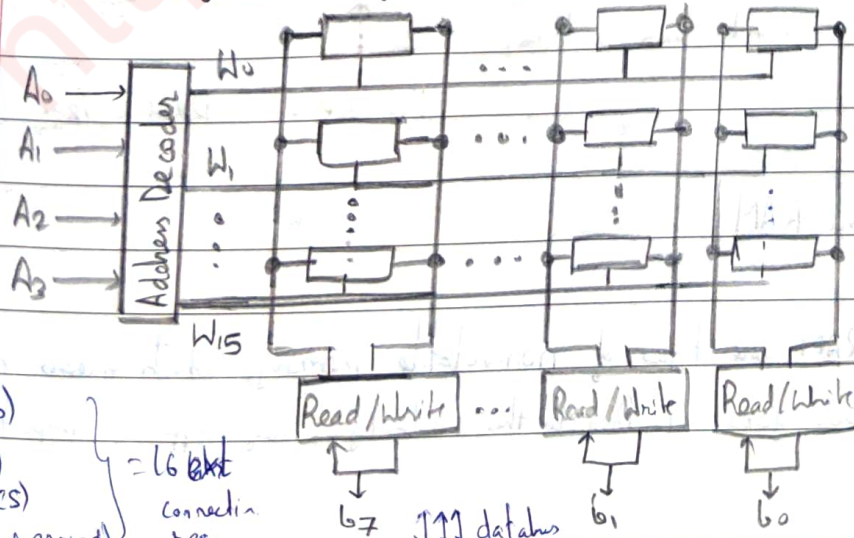
∴ Total addressable words = $\frac{2^{32}}{2^2} = 2^{30}$

32 bit → 00000000 00000000 00000000 00001110

30-bit → 3rd word

2 bit → 2nd byte location

⇒ Internal Organization of Memory



16×8 → memory cell/org.

= 128 bits

Each cell can store 1 bit of information

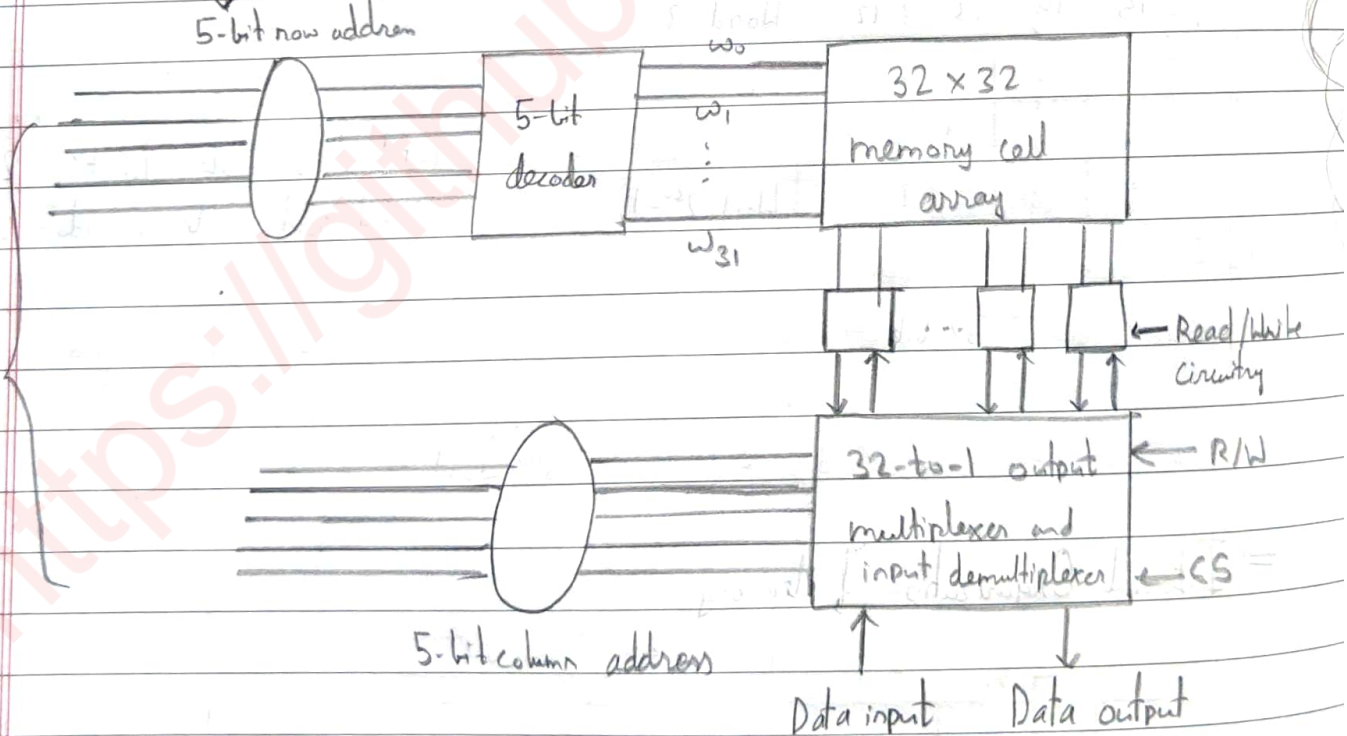
4 bits (address)
8 bits (data)
2 bits (R/W + CS)
2 bits (Power + ground)
= 16 connection req

If memory organization was of size = 1024 bits

i) 128×8 ⇒ 7 bits (addrs)
8 bits (data)
2 bits (R/W + CS)
2 bits (Power + gnd)
19 ext connections req

ii) 1024×1 ⇒ 10 bits (addrs)
1 bit (data)
2 bit
2 bit
15 ext connections required

iii) 32×32 ⇒ 5 bits (addrs)
5 bits (data)
2 bit
2 bits
14 ext connections req.



⇒ Static RAM and Dynamic RAM:

→ Both SRAM and DRAM are types of non-volatile memory, which means they lose their data if the power goes out. Despite this similarity, they differ in important ways

→ SRAM uses a flip-flop circuit to store each data bit.

→ The circuit delivers two stable states, which are read as 1 or 0. To support these states, the circuit requires six transistors, four to store the bit and two to control access to the cell.

→ Because of all these transistors, a SRAM chip has a much lower capacity than a DRAM chip of comparable size.

→ SRAM's are generally used for a processor's cache.

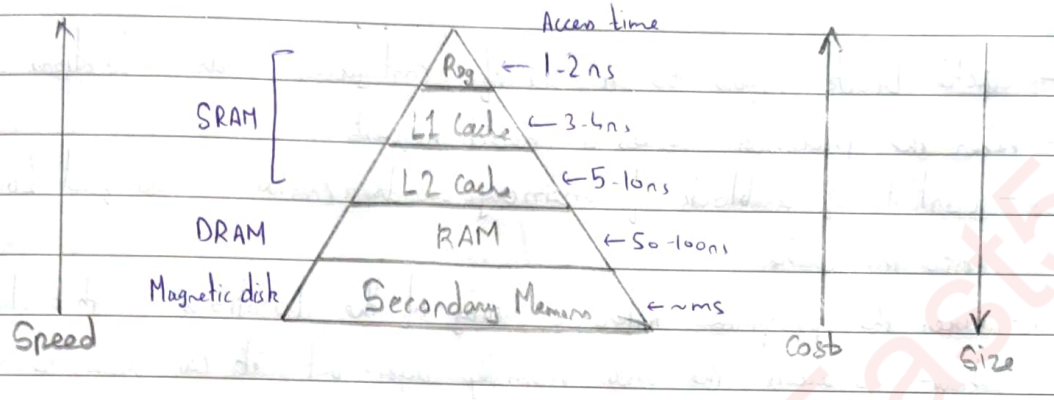
→ DRAM requires only one transistor and one capacitor to store a bit.

→ The capacitor holds the electrons that determine whether the bit is a 0 or 1. Unfortunately, DRAM capacitors have a tendency to leak electrons and lose their charge, so they must be refreshed periodically to retain their data, which can affect access speeds and increase power usage.

→ The transistor acts as a switch for reading and charging the capacitor's state.

→ Because of the different architectures, SRAM tends to perform better and require less power, especially when it sits idle. However, it cannot store as much data as DRAM, and it is more expensive.

⇒ Memory Hierarchy:



★ Introduction to Cache:-

→ Cache memories are small, very fast and are implemented using SRAM.
→ Accessing the main memory requires a lot of time, so these cache memories are imposed between the processor and the main memory.
→ These memories are based on a property called the locality of reference.
- Temporal (time): a recently executed instruction is likely to be executed very soon (Ex: loop)
- Spatial (Space → close): instructions close to a recently executed instruction are also likely to be executed soon (Ex: sequential flow of instruction)

⇒ Caches on processor chip:

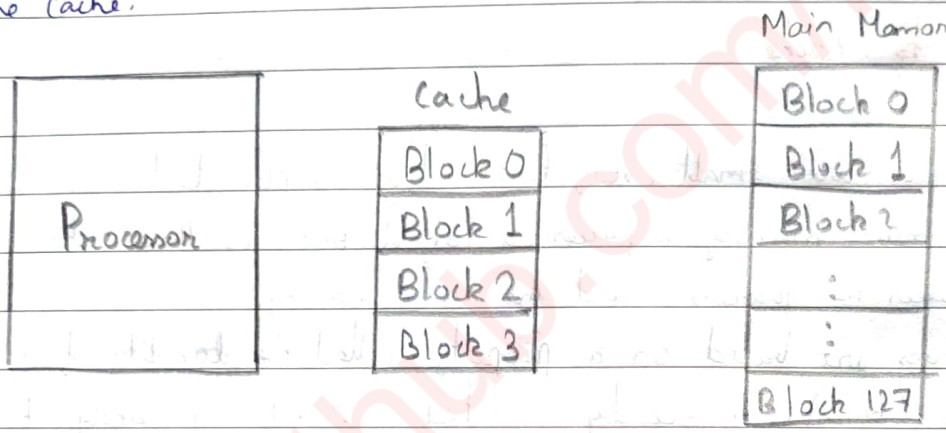
→ Most processor chips include at least one L1 cache (often two separate L1 caches, one for instructions and another for data).
→ In high performance processors, two levels of caches are normally used:
- Separate L1 caches for instructions and data and a larger L2 cache.
- These caches are often implemented on the processor chip.

Size: L1 cache → tens of KB
L2 cache → hundreds of KB or even MBs

⇒ Cache Block:

→ Cache block refers to a set of contiguous address locations of some size.

→ When the processor issues a Read request:
- Contents of a block of memory words containing specified location are transferred into the cache.
- When the program references any of the locations in that block, it will not have to access the main memory again but it can now read directly from the cache.



→ Suppose each block is of 8 words ⇒ Total words in cache = 8×4=32 words
Total words in RAM = 128×8 = ... words

→ If the processor issues out an address either to fetch an instruction or to read/write operation in the memory, so whatever address is sent out it is first checked in the cache, if it is not found in cache then it is checked in the main memory.

→ Suppose this address belongs to block b of the main memory, then this block b will be brought into the cache and where it will be placed that will depend upon some mapping that will take place between cache & MM.

→ What if the cache is full, then one of the block in it have to be replaced and that will happen with the use of some replacement algorithm.

Read Request:

1) Miss

→ When the processor sends a Read request and if it was not found in the cache then the processor will access the main memory and the block wherever the address is available, it will brought into the cache and then it will be sent to the processor.

→ This requires a lot of time because first the block will be sent to the cache then the address will be sent to the processor so there is another alternative that while the block is being transfered to the cache, that address that was requested by the processor can be sent directly to the processor parallely. This is called a <u>Load through</u> or <u>early start.</u>

2) Hit

→ When the processor sends a Read request and if it was found in the cache then the processor can directly read from the cache and the main memory will not be accessed at all.

Write Request:

1) Miss

→ When the processor sends a Write request and if it was not found in the cache:

a) Write through Protocol

→ Here the write operation will happen directly in the main memory & the cache memory will not be accessed at all.

b) Write back Protcol

→ Here first the block will be brought into the cache, then it will be written and later it will be replaced (changed) in the main memory.

2) Hit

→ When the processor sends a write request and if the address was found in cache then the processor updates the value in the cache while main memory does not contain the update data.

a) Write through protocol

→ Whenever a change is done in the cache at the same time the change will be done in the main memory also.

→ The drawback of this is that suppose the same address was updated many times by the processor, that many times the main memory must also updated.

b) Write Back Protocol

→ In this implementation all the changes will be made in cache only.

→ The main memory will be updated only when that block will be replaced by some other block.

→ So, this requires some maintainance of some kind of data to know whether this block was modified or not.

→ If this block was modified then only that complete block should be overwritten in the main memory.

→ If this block was not modified then that block can be directly replaced by a new block (no need to overwrite in MM).

* → Dirty bit / Modified bit is used to know whether a block is modified or not.

0 ✗ not modified
1 ✓ modified

→ Drawback of this is even if one address in a block was modified that whole block will be written back which will require so many memory accesses.

→ Advantage of this is that everytime a change is made in cache, the main memory is not required to be accessed.

⇒ Valid bit:

→ Valid bit is a control bit for each cache block which is used to indicate whether data in that block is valid or not.
→ Valid bits of all cache blocks are set to 0 when power is initially applied.
→ Valid bit may also be set to 0 when new programs or data are loaded from the disk into the main memory.
→ Valid bit of a given cache block is set to 1 when a memory block is loaded into that location.
→ The processor fetches data from a cache block only if its valid bit is equal to 1.
→ This ensures that the processor will not fetch stale data from the cache.

⇒ Hit rate and miss penalty:

→ Succesful access to data in a cache is called a hit.

$$\text{Hit rate} = \frac{\text{Number of hits}}{\text{attempted accesses}} \qquad \text{miss rate} = \frac{\text{number of misses}}{\text{attempted accesses}}$$

h $\quad$ $1-h$

→ Performance will be adversely affected when a miss occurs.
→ Performance penalty incurred because of extra time needed to bring a block of data from main memory to cache memory.
→ When the data is also not available in MM then it will be transferred from HD to MM and then to cache, during this period, the processor is stalled.
→ Total access time seen by the processor when a miss occurs is referred to as the miss penalty.

Consider a system with only one level of cache.

Let h be the hit rate, M is the miss penalty, and C is the time to access information in the cache.

Here Miss penalty includes time for
- initial access to the cache to check if address is available
- Then access to the main memory
- Transfer of word to the processor after the block is loaded into the cache (assuming no load-through)

Average access time experienced by the processor is $t_{avg} = hC + (1-h)M$

Q Consider a system with following parameters:
- Access times to cache and main memory are t and 10t, respectively
- When a cache miss occurs, a block of 16 words is transferred from main memory to cache.
- It takes 10t to transfer the first word of the block, and the remaining words are transferred at the rate of one word every t seconds.

Calculate the miss penalty.

$M = t + 10t + 15t + t = 27t$

- t: to access cache
- $10t$: to access first word of main memory
- $15t$: to access remaining word
- t: to again access cache

Q Assume 30% of the instructions in a program perform a Read or a Write operation. There will be 130 memory accesses for every 100 instructions executed. Hit rate in the cache is 0.95 for instructions. Hit rate in the cache is 0.9 for data. Miss penalty is same for data both read and write accesses. Calculate time required by the processor to execute those operations using i) Cache ii) without cache (Use above data)

→ 100 instructions were fetched from memory + 30 additional other memory accesses for read/write (load/store)

Time without cache $T_1 = 130 \times 10t = 1300t$

130 = 100 (Inst) + 30 (Data)
100 → load/store
30 → Read/Write

Time with cache $T_2 = (hC + (1-h)M)_{Inst} \times 100 + (hC + (1-h)M)_{Data} \times 30$

$= (0.95 \times t + (1-0.95) \times 27t) \times 100 + (0.9 \times t + (1-0.9) \times 27t) \times 30$

$= 338t$

$\therefore \frac{T_1}{T_2} = \frac{1300t}{338t} = 3.85$

∴ Cache makes the memory almost 4 times faster than it really is.

Compare this cache with an ideal cache where the hit rate is 100%.
Also, with ideal cache behaviour, all memory references take one t

Time for real cache, $T_2 = 338t$
Time for ideal cache, $T_3 = 130t$

$T_2/T_3 = 2.6$

∴ 100% hit rate in the cache would make the memory appear more than twice as fast as other realistic cache is used.

Assume : → A split L1 cache and one L2 cache
→ Average access time of the L2 cache is the miss penalty of either of the L1 caches.
→ Hit rates are the same for instruction & data.

Average access time experienced by the processor is : $t_{avg} = h_1 C_1 + (1-h_1) M_1$

& $M_1 = h_2 C_2 + (1-h_2) M_2$

$\therefore t_{avg} = h_1 C_1 + (1-h_1)[h_2 C_2 + (1-h_2) M_2]$

h_1 → hit rate in L1 cache
h_2 → hit rate in L2 cache
C_1 → time to access information in L1 cache
C_2 → time to access information in L2 cache
M_1 → miss penalty for L1 cache
M_2 → miss penalty for L2 cache

If both h_1 and h_2 are 90%, number of misses in L2 cache will be less than 1% of all memory accesses. : $(1-h_1)(1-h_2)$

0.01

Q Assume two L1 caches, one for instructions and one for data and an L2 cache. Let t be the access time for the two L1 caches. Miss peralties are : i) 15t for transferring a block from L2 2) 100t for transferring a block from main memory. Hit rates are same for instructions & data. Hit rates in L1 and L2 caches are 0.96 and 0.80 respectively. What is the average access time as seen by the processor?

$$t_{avg} = 0.96\,t + (1-0.96)[0.80 \times 15t + (1-0.80) \times 100t]$$

⇒ Replacement Algorithms:

→ Wherever a main memory block has to be brought into the cache, so a mapping function decides where in the cache that block can be placed, so there are only a few places available where that block can be placed and if those positions are occupied then some replacement algorithm needs to exist to decide which block has to be overwritten.

→ This problem does not occur in direct mapping because in direct mapping there is only one specific block where a main memory block can be placed and then that block will be overwritten. But in associative or set associative there are multiple positions available where the main memory block can be placed.

→ Program execution usually stays in localized areas for reasonable period of time.
→ There is a high probability that blocks referenced recently will be referenced again soon.
→ so, it is sensible to overwrite one that has not been referenced for a long time.
→ This technique is called as the Least Recently Used (LRU) replacement algorithm.

Ex: Consider cache: Fully associative, 4-blocks in cache

~~Reference~~ String: 2 4 7 1 2 1 3 8 4 1 8 8

2	Block 0	0	0	1	1	1						
4	Block 1	0	1	0	1	1						
7	Block 2	0	1	0	0	1						
1	Block 3	0	1	0	1	0						

[ ] → highest count

H

Hit, Hit, Miss (Replace with Block 1), Miss, Hit, Hit, Hit

Reference String: 2 4 1 7 2 1 3 8 4 1 8 8

~~2~~ 4	Block 0	0	←0	1	2	3	←0	1	2	3	0	1	2	2
~~4~~ 3	Block 1	0	1	←0	1	2	3	→3	0	1	2	3	3	3
1	Block 2	0	1	2	←0	1	2	←0	1	2	3	0	1	1
~~7~~ 8	Block 3	0	1	2	3	←0	1	2	3	0	1	2	0	0

⇒ Memory-to-Cache Mapping: Wherever a block from the main memory has to be brought into the cache, there should be some mapping function which will help decide where that main memory block has to be placed into the cache.

1) Direct-Mapping

→ Here block j of main memory will map to (block j) % (no. of blocks in cache).
→ Assume there are 128 blocks in cache.
→ Main memory blocks 0, 128, 256, ... will be loaded one by one into cache block 0.
→ Main memory blocks 1, 129, 257, ... will be loaded one by one into cache block 1.
→ Problem may arise for a block even when the cache is not full as Assume that main memory block 0 is loaded into cache block 0, and now the process wants to access an address which is main memory block 128 which will also map to cache block 0, so block 0 has to be replaced with block 128 even though space was available.

Contention Problem

Ex: Cache: 128 blocks of 16 words each
Main Memory: addressable by 16-bit address, word addressable

∴ Total no. of words in Cache = 128 × 16
∴ No. of (word) address = 2^{16}

16 words → 1 block in cache
2^{16} words → $\frac{2^{16}}{2^4}$ blocks in RAM (assuming 1 block in RAM also has 16 words)
$= 2^{12}$ blocks in RAM

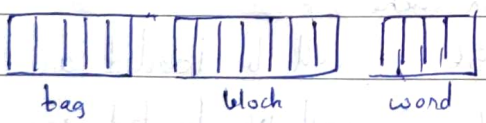
Cache

Tag	Block 0
Tag	Block 1
Tag	Block 2
⋮	⋮
⋮	⋮
Tag	Block 127

RAM

Block 0	0 – 15
Block 1	16 – 31
Block 2	32 – 47
Block 3	48 – ⋮
Block 4	
⋮	
⋮	
Block 4095	

Main Memory address bits can be divided into 4 bits for word + 7 bits for block
(16 words in 1 block) (Total 128 blocks)
+ 5 bits for Tag



What is the use of tag bits?
Ex: Block 1 of the RAM will be mapped to 1 % 128 = Block 1 in cache
Also Block 129 of the RAM will be mapped to 129 % 128 = Block 1 in cache
⋮
Block 3969 of the RAM will be mapped to 3969 % 128 = Block 1 in cache
Now, when the processor sends out an address, how does it know whether the Block 1 of RAM is present or block 129 is present or ...
∴ Total tag bits = 5 bits = 2^5 = 32 → 1, 129, ..., 3969

✓ correct assumption

→ Now checking all the tag bits will require some time. So, to reduce this search time, all the searches are done in parallel and this is called associative search. Thus, the complexity is more.

Q Identify in which cache block memory address 26 will map to in the above cache & RAM diagram.

$\left\lfloor \frac{26}{16} \right\rfloor = 1$ ⇒ Block 1 of RAM → map 1 % 128 → Block 1 of cache

⇓

16 bit address ⇒ 00000 | 0000001 | 1010 ← (26) binary

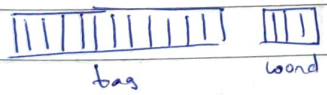
Tag 0 | Block 1 in cache | 10th word

2) Associative Mapping:

→ Here main memory block can be placed in any cache block position.
→ Complete freedom in choosing the cache location, resulting in a more efficient use of the space in the cache.
→ When new block is brought into cache, it replaces an existing block only if the cache is full.
→ A replacement algorithm selects the block to be replaced.
→ Complexity of an associative cache is higher compared to direct mapping.

→ Same Example

Main Memory address bits can be divided into 4 bits for word + 12 bits for tag (16 bits in 1 block)



26 address → 0000 0000 0001 | 1010

tag | 10th word

→ Whenever the processor sends out a new address it will have to check the tag bits of all the cache blocks to determine whether this particular block which this particular address is there is available in the cache or not. Only if the tag bits are available that means this address is present in the cache otherwise it will be a miss.

3) Set-associative mapping:

→ Blocks of cache are grouped into sets.
→ Block of main memory can reside any block of a specific set.
→ Contention problem is eased by having some few choices for block placement.
→ Hardware cost is reduced by decreasing the size of associative search.
→ Here (main memory block j) will map to (block j) % (no. of sets in cache).

Ex: For a cache with 128 blocks and two blocks per set → (K=2)
→ Memory blocks 0, 64, 128, ..., 4032 map into cache set 0
→ Can occupy either of the two block positions within this set.

→ Cache that has K blocks per set is referred to as a K-way set associative cache.

In the previous example if K=128 → Associative
K = 1 → Direct

Ex: Cache: 128 blocks of 16 words each, 2 blocks per set (2-way)
Previous example ✓ Main memory: addressable by a 16-bit address, word addressable

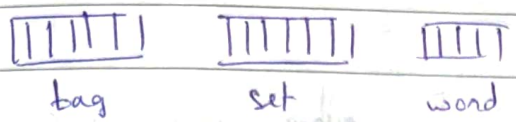
Cache

Set 0	Tag	Block 0
	Tag	Block 1
Set 1	Tag	Block 2
	Tag	Block 3
		⋮
		⋮
Set 63	Tag	Block 127

RAM

Block 0
Block 1
Block 2
Block 3
Block 4
⋮
⋮
⋮
Block 4095

Main Memory address bits can be divided into - 4 bits for word + 6 bits for set + 6 bits for Tag
(16 words in a block) (64 sets)



26 → 000000 000001 1010
address
↓ tag 0 ↓ Set 1 ↓ 10th Block

$\left\lfloor \frac{26}{16} \right\rfloor = 1 \Rightarrow$ Block 1 of RAM → map → 1 % 64 = 1 → Block 1 of cache
Block 2/Block 3

$\left\lfloor \frac{2070}{16} \right\rfloor = 129 \Rightarrow$ Block 129 of RAM → map → 129 % 64 = 1 → Block 1 of cache

129 → 000010 000001 0110
tag 2 | Block 1 | 6th word

So it didn't replace Block 1 with Direct mapping

★ Introduction to Virtual memory :-

→ Main memory may not be as large as the address space of the processor.
(Ex: 32-bit address passed by processor → 2^{32} = 4 GB (Address space) but memory = 1GB/2GB/3GB).

→ A program may not completely fit into the main memory. So, only those parts of the program which are being executed they will be brought into the memory & the parts which are not currently being executed are stored on the secondary storage.

→ So, whatever parts are needed for execution, they must first be brought into the main memory
- So, the parts which are not in the main memory will be automatically brought into MM when they are to be executed by the Operating System.
- Application programmers need not be aware of their limitations.

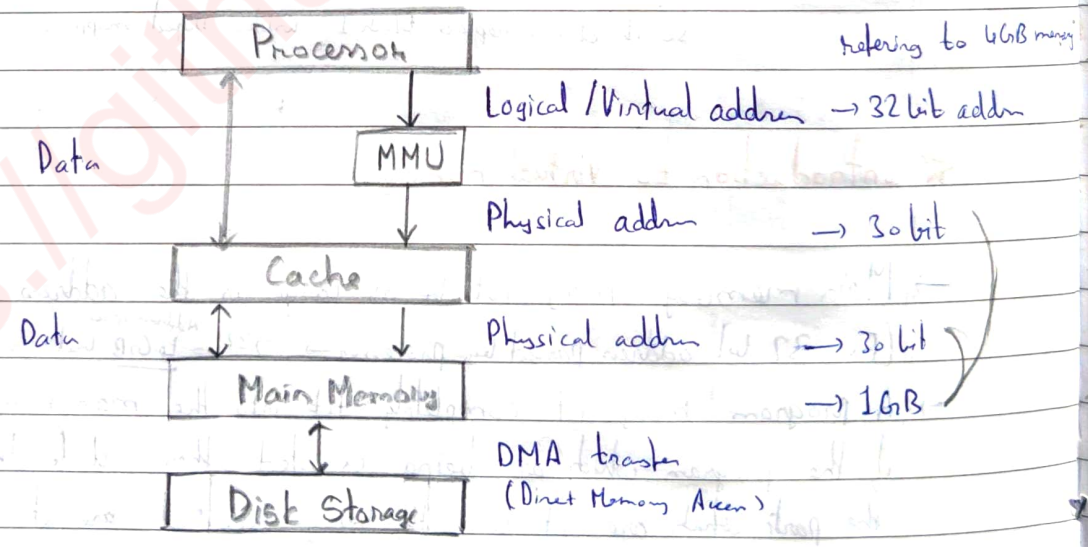
Why can't we use mapping in HD and RAM?
Because of Size of HD is not finite/fixed.

→ So, this approach of bringing parts which are required into the main memory & showing an address space equivalent to the address space of processor, this approach is called Virtual Memory.

→ Binary address issued by processor for either instructions or data are called virtual / logical addresses.
→ They are later translated to physical address
refers to the address in main memory

⇒ Memory Management Unit (MMU):

→ A special hardware unit.
→ Translates the virtual address into the corresponding physical address.
→ Keeps track of which parts of the virtual address space are in the physical / main memory.



⇒ Logical to physical address translation:

→ All programs and data are composed of fixed-length units called pages.
- A block of words that occupy contiguous locations in main memory.
- Constitute basic unit of information that is transferred between the main memory and disk.

→ All these virtual pages which are being referred by the processor are kept in the secondary storage.

→ Pages should not be too small : Takes a considerable amount of time to locate the data on the disk

→ Pages should not be too large also: If page size is too large then we are bringing a large amount of information to main memory and possibly many parts of that page in that many addresses may not be used and they would be occupying valuable memory space.

→ An area in the main memory that can hold one page is called a page frame

→ Each virtual address generated by the processor consists of:
a) Virtual page number (high-order bits) = 22 bits
b) An offset (low-order bits) = 10 bits
↓ what is the distance of that address from the starting address of page

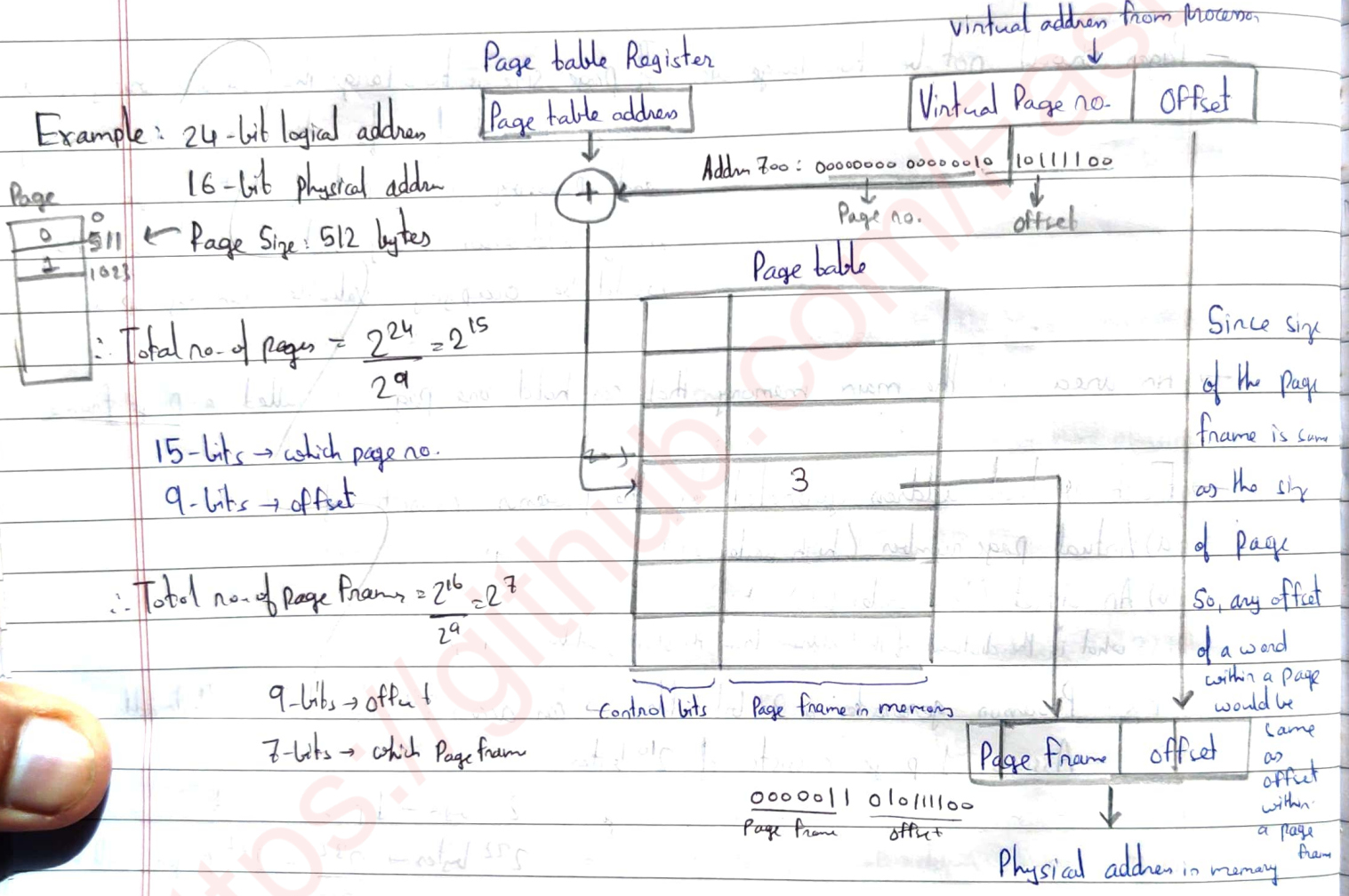
Ex: Processor generates a 32 bit address → can access 2^{32} address locations (byte addressable) = 2^{32} bytes
Assume 1 page consists of 2^{10} bytes

2^{10} bytes → 1 page
2^{32} bytes → $\frac{2^{32}}{2^{10}} = 2^{22}$ pages will be present

Now, within a page where is the particular bytes we are referring to? That will be determined by offset

→ When a page is brought from the HD to MM, this information needs to be also stored somewhere. So, the information about main memory location of each page is kept in a page table. (located in the RAM)
→ Starting address of page table is kept in a page table base register. to locate where it is present in the main memory.

→ Each entry in the page table includes some control bits:
- One bit indicates validity of the page (valid → if present or not)
- Another bit indicates whether the page has been modified (Dirty bit → 1 if modified, 0 if not → Replace / overwrite)
- Other control bits indicate various restrictions that may be imposed on accessing the page.



Example: 24-bit logical address
16-bit physical addr
Page Size: 512 bytes ← 0, 511, 1023

∴ Total no. of pages = $\frac{2^{24}}{2^9} = 2^{15}$

15-bits → which page no.
9-bits → offset

∴ Total no. of Page Frames = $\frac{2^{16}}{2^9} = 2^7$

9-bits → offset
7-bits → which Page frame

⇒ Address Translation:

→ Page table information is used by the MMU for every read and write access.
→ So, if the processor generates any virtual address, the MMU will need to access the page table and thus so many main memory access would be required and to reduce this time, we can keep the page table inside MMU but it is not possible to include the whole page table as it is large.
→ So, only a small portion of the complete table will be copied within the MMU.

→ And within this table only those entries/pages which have been accessed recently are kept.

→ So, this small table is called Translation Lookaside Buffer (TLB).

→ This TLB is a kind of acting like a cache for the page table available in the MM.

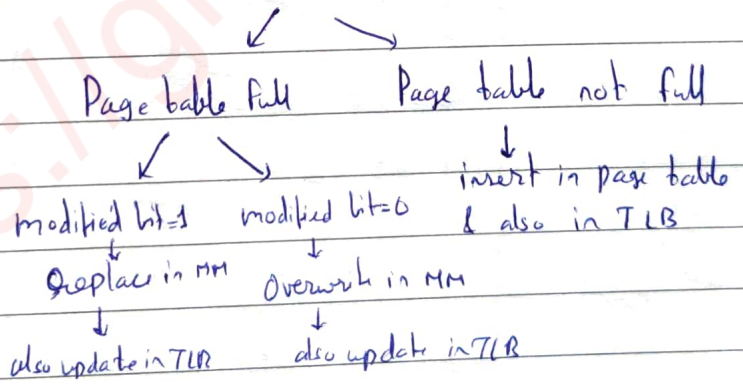
→ Each entry in TLB includes a copy of the information in the corresponding entry of the page table.

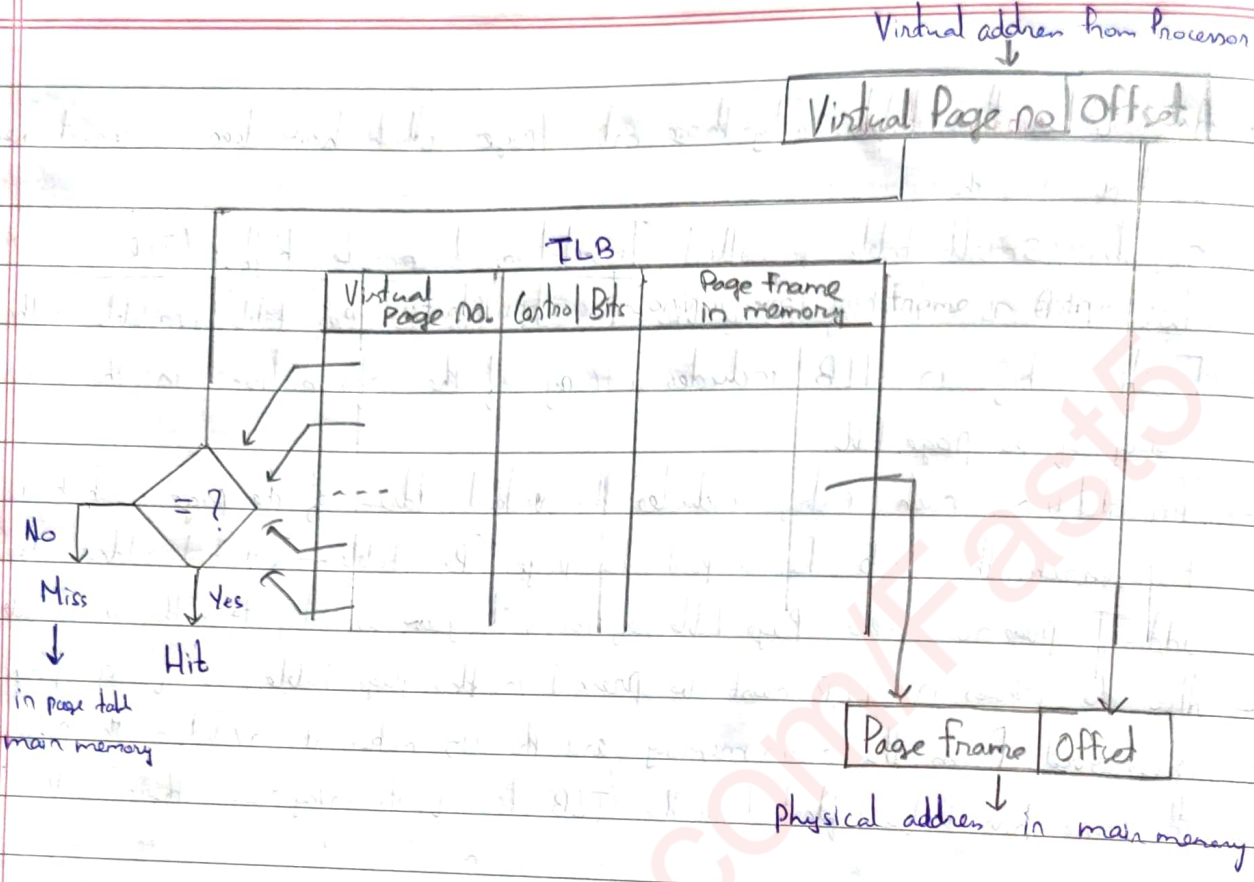
→ In addition, each entry includes the virtual address of the page, which is needed to search the TLB for a particular page. (Page table → x virtual address because adding page no. with Page table register, it gave us entry from the page table)

→ Also the entries in TLB must be present in the page table, so if a page is removed from the main memory and the OS makes it invalid in the page table, the same has to be reflected in the TLB also if its entry was there in the TLB.

→ When a program requests a page that is not in main memory, a page fault is said to have occured.

→ If a processor generates an address which is not available in TLB, it then checks in the MM (page table) if there also not available then it is brought from the HD.

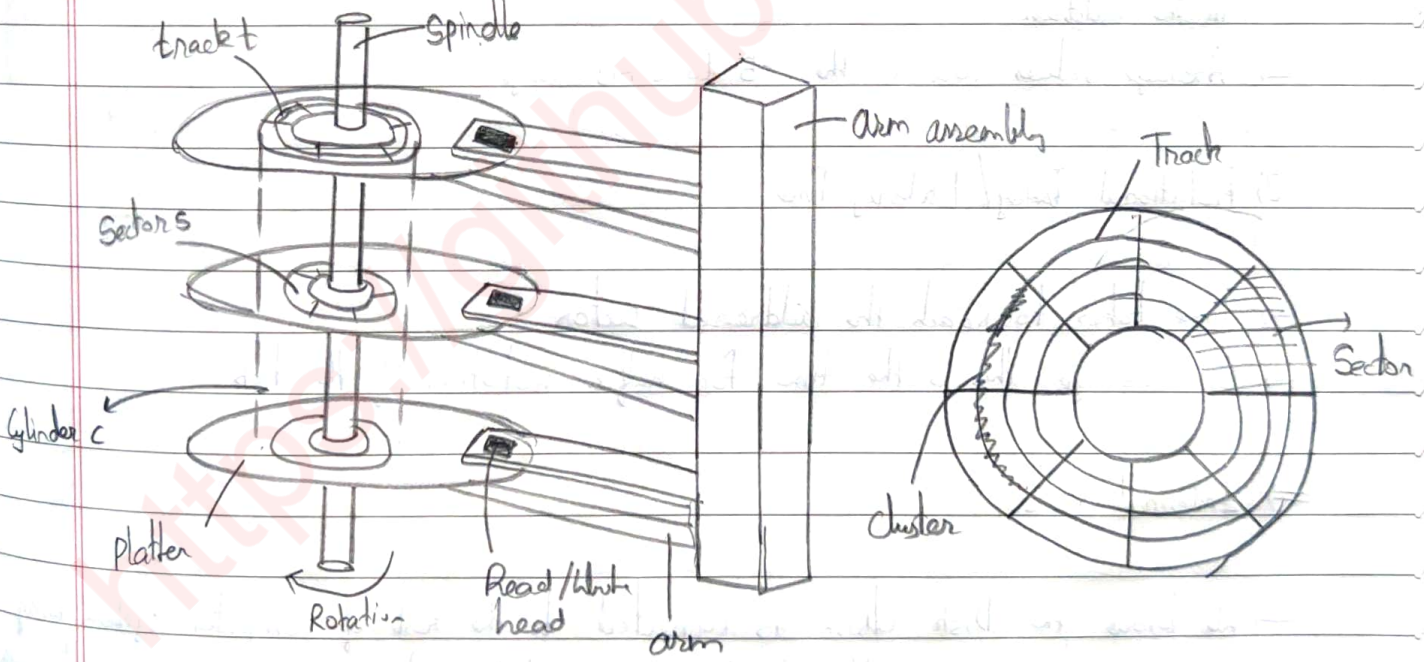


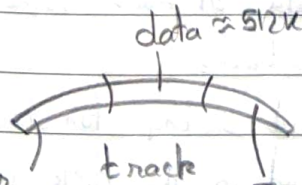


8

★ Magnetic hard disk :-

→ The secondary storage in the system is usually implemented as Magnetic Hard Disk.
→ One or more disk platters are mounted on a common spindle.
→ On each platter, there is a magnetic film, usually on both sides which are used to store & retrieve information. This magnetic surfaces come in close proximity to read / write heads.
→ Some assembly is placed in a drive that causes it to rotate (clockwise or anticlockwise) at a constant speed.
→ This Read/Write heads move radially (in & out) to access different tracks as data is stored on concentric tracks.
→ Digital information can be stored on the magnetic ~~field~~ film by applying current pulses of suitable polarity to the magnetizing coil.





(Has information to identify this sector)

(which maintains the integrity of data to see whether the data is correct)

Each sector is divided (seperated) from the other sector by an intersector gap.

Disk system consists of mainly 3 parts:

1) Assembly of disk platters, referred to as the disk.
2) Electromechanical mechanism that spins the disk and moves the read/write heads, called the disk drive.
3) Electronic circuitry that controls the operation of the system, is known as the disk controller.

⇒ Access Time:

1) Seek Time:

→ Time required to move the read/write head to the proper track.
→ Depends on the initial position of the head relative to the track specified in the address.
→ Average values are in the 5 to 8 ms range.

2) Rotational ~~Delay~~/ Latency Time:

→ Time taken to reach the addressed sector.
→ On average, this is the time for half a rotation of the disk.

⇒ Internal Parts:

→ We know the Disk drive is connected to the rest of computer system using the Bus which is capable of transferring data at much higher rates than the rate at which data can be read from disk tracks.
→ Since we are using a single bus, so to avoid waiting for storing/retreived data from disk we use data buffer which is present in the disk controller.

→ When a Read request arrives at the disk, the controller checks to see if the required data is already available in the buffer.
→ If not available then it fetches from the disk along with other sequential datas. (Locality of Reference)

→ Operation of a disk drive is controlled by a disk controller circuit.
- communicates directly with the processor
- contains a number of registers that can be read/written by the OS.
- These registers are required during DMA to transfer data and OS initiates the transfers (Register contains starting address of memory from which data has to be transferred, no. of words to transfer, read/write operation).

Major Functions of disk controller circuit

1) Seek - to find the particular track and sector in the disk

2) Read - initiates operation, starting at address in disk address register
- data read serially from the disk are assembled into words and placed into the data buffer.

3) Write - transfers data to the disk

4) Error Check - compute the ECC i.e. the ECC computed by the controller is matched with the ECC which is stored on the sector

↙ If match correct ↘ If not correction needed

→ In modern disk units, disks and the read/write heads are placed in a sealed, air-filtered enclosure which is known as Winchester technology.
→ So, the storage medium is not exposed to contaminating elements like dust and read/write heads can operate closed to the magnetized track surfaces.
→ The closer the heads are to track surface, the more densely the data can be packed along the track and more closer the tracks can be to each other.
→ So, larger capacity for a given physical size compared to unsealed units.

Cache Q Assume that for a certain processor, a read request takes 50ns on a cache miss and 5ns on a cache hit. Suppose while running a program, it was observed that 80% of the processor's read requests results in a cache hit. The average read access time in ns is ____.

$t_{avg} = hC + (1-h)M$

↓ Miss Penalty (access to cache, access to MM, load to processor after loading into cache)

$= 0.8 \times 5 + (1-0.8) \times 50$

$= 14 ns$

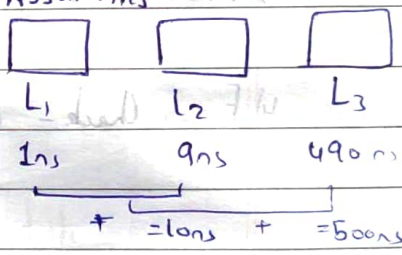
Cache Q Consider a system with 2 level caches. Access times of level 1 cache, level 2 cache, and main memory are 1ns, 10ns and 500ns, respectively. The hit rates of level 1 and level 2 caches are 0.8 and 0.9 respectively. What is the average access time of the system ignoring the search time within the cache?

$t_{avg} = h_1C_1 + (1-h_1)[h_2C_2 + (1-h_2)M]$

$= 0.8 \times 1 + (1-0.8)[0.9 \times 10 + (1-0.9)500]$

$= 12.6 ns$

Assuming



Cache Q A hierarchical memory system that uses cache memory has cache access time of 50ns, main memory access time of 300 ns, 75% of memory requests are for read, hit ratio of 0.8 for read access and the write-through scheme is used. What will be the average access time of the system both for read and write requests? (Assume cache access time is not included in main memory access time).

$t_{avg} = hC + (1-h)M$

↳ for read: $t_1 = 0.8 \times 50 + (1-0.8)(300+50) = 110 ns$

↳ for write: $t_2 = \max(50, 300) = 300 ns$

∴ Average Memory Access Time $= \frac{75}{100} \times 110 + \frac{25}{100} \times 300 = 157.5 ns$

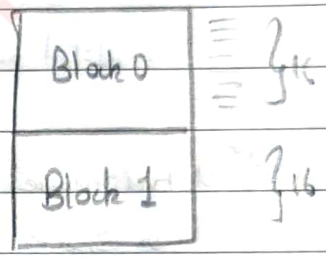
Cache Q Consider an array A[999] and each element occupies 4 word. A 32-word cache is used and divided into 16 word blocks. What is the miss ratio for the following statement. Assume one block is read into cache in case of a miss.

```
for (int i=0 , i<1000 , i++)
    A[i] = A[i] +99
```

→ first it is read then it is updated & then written back

Cache Size = 32 words
Size of each block = 16 word
∴ Total blocks in Cache = $\frac{32}{16} = 2$

Each element size = 4 words
∴ No. of elements in each block = 4 elements



On every cache miss, 4 elements are brought in cache block
so, after one miss, next 3 elements will be hit

Each element requires one read and one write (2 references)
So, for each block, there will be 8 reference — 1 miss and 7 hit

∴ Miss Ratio = $\frac{1}{8} = 0.125$

Cache Q A direct mapped cache memory of 1MB has a block size of 256 bytes. The cache has an access time of 3 ns and a hit rate of 94%. During a cache miss, it takes 20ns to bring the first word of a block from the main memory, while each subsequent word takes 5ns. The word size is 64 bits. The average memory access time in ns is ____. (Round off to 1 dp)

Cache Size = 1 MB = 2^{20} bytes
Block Size = 256 bytes
∴ Total Blocks in Cache = $\frac{2^{20}}{2^8} = 2^{12}$ blocks

Word Size = 8 bytes
∴ Total words in each block = $\frac{2^8}{2^3}$ = 32 words

M = 20 + 5×31 = 175 ns

$t_{avg} = 0.94 \times 3 + (1-0.94) \times (3+175)$
= 13.5 ns

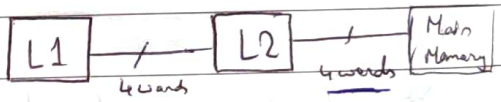
Cache Q A cache line is 64 bytes. The main memory has latency 32 ns and bandwidth 1 GB/s. The time required to fetch the entire cache line from the main memory is ____

Cache line (Block) size = 64 bytes
Time required to access memory = 32 ns
Bandwidth = 1 GB/s = 10^9 bytes/s

∴ Time required to transfer data = $\frac{64}{10^9}$ = 64 ns

∴ Total time = 32 + 64 = 96 ns

Cache Q A computer system has an L1 cache, an L2 cache, and a main memory unit connected as shown. The block size in L1 cache is 4 words. The block size in L2 cache is 16 words. The memory access times are 2 ns, 20 ns and 200 ns for L1 cache, L2 cache and main memory unit respectively. When there is a miss in L1 cache and a hit in L2 cache, a block is transferred from L2 cache to L1 cache. What is the time taken for this transfer? When there is a miss in both L1 cache and L2 cache, first a block is transferred from main memory to L2 cache, and then a block is transferred from L2 cache to L1 cache. What is total time taken for these transfers?



a) Time taken to transfer 4 words from L2 to L1 = 2 + 20 = 22 ns

b) Time taken to transfer 16 words from MM to L2 and = (200 + 20) × 4 + 20 + 2 = 902 ns
Then 4 words from L2 to L1

Cache Q A CPU has 32KB direct mapped cache with 128 byte block size. Suppose A is a 2D array of size 512 × 512 with elements that occupy 8 bytes each. Consider the code segment:

```
for(i=0, i<512, i++){
    for(j=0, j<512, j++){
        x = x + A[i][j];
    }
}
```

Assuming array is stored in order A[0][0], A[0][1], A[0,2], ...
Number of cache misses is ____

Block Size = 128 bytes

Size of each element = 8 byte

∴ No. of elements in each block = $\frac{2^7}{2^3}$ = 16 elements ⇒ 1st element miss then next 15 elements hit

16 elements → 1 miss

512 elements → $\frac{512}{16}$ = 32 miss ⇒ In each row (i) (512 columns are present)

Cache
A[0][0] A[0][1] ... A[0][15]
A[0][16] ... A[0][31]
.

∴ Total no. of misses = 32 × 512 = 16384

for (i=0 to 511)

mapping + LRU

Q Consider a 16-way set associative cache (initially empty) with total 16 cache blocks. The main memory consists of 256 blocks and the request for memory blocks is in the following order: 0, 255, 1, 4, 3, 8, 133, 159, 216, 129, 63, 8, 48, 32, 73, 92, 155. Which one of the following memory block will not be in cache if LRU replacement policy is used?

(A) 3 (B) 8 (C) 129 (D) 216

No. of cache blocks = 16

∴ No. of sets in cache = 16/4 = 4

No. of main memory blocks = 256

Block in Cache = Block in MM % 4

Memory block	0	255	1	4	3	8	133	159	216	129	63	8	48	32	73	92	155
Cache set	0	3	1	0	3	0	1	3	0	1	3	0	0	0	1	0	3
	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✗	✗
													miss Replaced with 0	miss Replaced with 4		miss Replaced with 216	miss Replaced with 255

Cache

Set 0	~~48~~ 32
	8
	~~216~~ 92
Set 1	1
	133
	129
	73
Set 2	
Set 3	255 155
	3
	159
	63

mapping

Q A computer has a 256 Kbyte, 4-way set associative, write back data cache with block size of 32 Bytes. The processor sends 32 bit addresses to the cache controller. Each cache directory entry contains, in addition to address tag, 2 valid bits, 1 modified bit and 1 replacement bit. The number of bits in the tag field of an address is ____. The size of the cache tag directory is ____. The size of the cache directory is ____

Cache Size = 256×2^{10} bytes

Block Size = 32 bytes

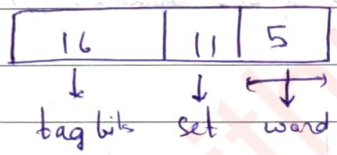
$\therefore$ No. of blocks in cache = $\frac{2^8 \times 2^{10}}{2^5} = 2^{13}$

No. of sets in cache = $\frac{2^{13}}{4} = 2^{11}$

$2^{11} \rightarrow 2^{13}$

$1 \rightarrow \frac{2^{13}}{2^{11}} = 2^2$

Total address bits given by processor = 32



$\therefore$ Each tag entry = 16 + 2 + 1 + 1 = 20 bits

$\therefore$ Cache tag directory Size = 20×2^{13} = 160 KBits

mapping

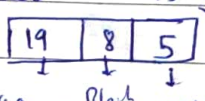
Q An 8 KB direct-mapped write-back cache is organized as multiple blocks, each of size 32 bytes. The processor generates 32-bit addresses. The cache controller maintains tag information for each cache block comprising of the following: 1 valid bit and 1 modified bit. As many bits as minimum needed to identify the memory block mapped in cache. What is total size of memory needed at cache controller to store meta-data (tags) for cache? (Each word has 1 byte size)

Cache Size = 8 KB = $2^3 \times 2^{10} = 2^{13}$ Bytes

Block Size = 32 Bytes

$\therefore$ No. of Blocks in cache = $\frac{2^{13}}{2^5} = 2^8$

Total address bits given by processor = 32



$\therefore$ Each tag entry = 19 + 1 + 1 = 21 bits

$\therefore$ Cache tag directory size = 21×2^8 = 5376 bits

mapping Q Consider a direct mapped cache with 64 blocks and a block size of 16 bytes. To what block number does the byte address 1206 map to ____. (1 word = 1 byte)

Block Size = 16 bytes

Block with contains address 1206 in main memory $= \left\lfloor \frac{1206}{16} \right\rfloor = 75$

For direct mapping : cache block = 75 % 64 = 11

∴ Addrn 1206 map to cache block 11

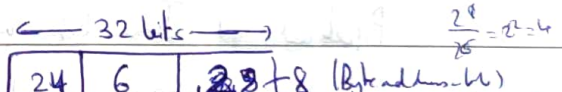
mapping Q A computer system with a word length of 32 bits has a 16 MB byte addressable main memory and a 64 KB, 4-way set associative cache memory with a block size of 256 bytes. Consider the following four physical addresses represented in hexadecimal notation. A1 = 0x62C8A4, A2 = 0x546888, A3 = 0x6A289C, A4 = 0x5E4880. Which addresses are mapped to the same cache set ?

Cache Size = 64KB = $2^6 \times 2^{10} = 2^{16}$ bytes

Block Size = 256 bytes = 2^8 bytes

No. of blocks in Cache = $\frac{2^{16}}{2^8} = 2^8$ blocks

No. of sets in Cache = $\frac{2^8}{4} = 2^6$ sets



$\frac{2^8}{2^6} = 2^2 = 4$

$\frac{2^8}{2^8} = 2^0$

$\frac{2^{16}}{2^{12}}$

$2^8 \times 2^2 = 2^{10}$

A1 = 0x62C8A4 = 00 001000 | 10100100

A2 = 0x546888 = 01 010100 | 01001000

A3 = 0x6A289C = 00 101000 | 10011100

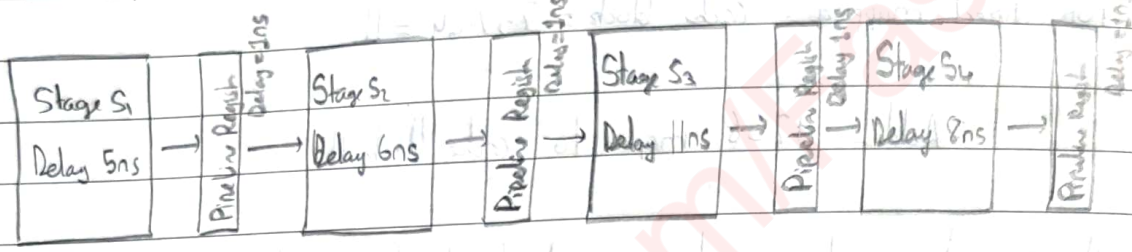
A4 = 0x5E4880 = 01 001000 | 10000000

→ These two are mapped to same cache set with bits 101000

→ These two are mapped to same cache set with bits 001000 (6 bits)

Pipelining

Q Consider an instruction pipeline with four stages (S1, S2, S3 and S4) each with combinational circuit only. The pipeline registers are required between each stage and at the end of the last stage. Delays for the stages and for the pipeline registers are as given in the figure. What is the approximate speed up of the pipeline in steady state under ideal conditions when compared to the corresponding non-pipeline implementation?



Non-Pipelined → no interstage pipeline buffer

Total time = 5+6+11+8 = 30ns
taken by each instruction

Pipelined → Time taken by each stage = max(5, 6, 11, 8) = 11+1 = 12ns
(its interstage buf)

→ Ignoring the latency of first instruction, each instruction takes 12ns
(1 clock cycle to complete)

∴ Speed up = $\frac{30}{12}$ = 2.5

Pipelining

Q Consider a 6-stage instruction pipeline, where all stages are perfectly balanced. Assume that there is no cycle-time overhead of pipelining. When an application is executing on this 6-stage pipeline, the speed-up achieved with respect to non-pipelined execution if 25% of the instructions incur 2 pipeline stall cycles is ____.

Non-pipelined : Total time = 6 cycles (Each stage takes 1 cycle)
taken by each instruction

Pipelined : Time for 75% instructions = 1 cycle (without stall)
Time for 25% instructions = 3 cycles (with stall)
∴ Total time = 75% + 3×25%

$\therefore$ Speed-up $= \frac{6}{1.5} = 4$

Q Consider the sequence of machine instructions given below:

```
MUL R5, R0, R1
DIV R6, R2, R3
ADD R7, R5, R6
SUB R8, R7, R4
```

R0 to R8 are general purpose registers

First register stores result of operation performed on 2nd & 3rd registers.

This sequence is to be executed in a pipelined instruction processor with 4 stages: 1) Instruction Fetch and Decode (IF), 2) Operand Fetch (OF), 3) Perform operation (PO) and 4) Write back the result (WB). IF, OF and WB stages take 1 clock cycle each for any instruction. PO stage takes 1 clock cycle for ADD or SUB instruction, 3 clock cycle for MUL instruction and 5 clock cycles for DIV instruction. Processor uses operand forwarding from PO stage to OF stage. Number of clock cycles taken for execution of instruction is ___.

	1	2	3	4	5	6	7	8	9	10	11	12	13
MUL R5, R0, R1	IF	OF	PO	PO	PO	WB							
DIV R6, R2, R3		IF	OF	PO	PO	PO	PO	PO	PO	PO	WB		
ADD R7, R5, R6			IF	OF						OF	PO	WB	
SUB R8, R7, R5				IF	OF						OF	PO	WB

Forward operand

$\therefore$ 13 clock cycles for all instructions

Q Consider an instruction pipeline with five stages without any branch prediction: Fetch Instruction (FI), Decode Instruction (DI), Fetch Operand (FO), Execute Instruction (EI) and Write Operand (WO). The stage delays for FI, DI, FO, EI and WO are 5ns, 7ns, 10ns, 8ns & 6ns respe. There are intermediate storage buffers after each stage and the delay of each buffer is 1ns. A program consisting of 12 instructions I1, I2, I3, ... I12 is executed in this pipelined processor. Instruction I4 is the only branch instruction and its branch target is I9. If the branch is taken during the execution of this program, the time (in ns) needed to complete the program is ___.

Clock cycle time for pipeline = max(5,7,10,8,6) + 1 = 11ns

Required clock cycles: I1 → 5

I2, I3, I4 → 3

Branch Penalty = 3

I9, I10, I11, I12 → 4

∴ Total clock cycles = 15

∴ Total time = 15 × 11 = 165 ns

	1	2	3	4	5	6
I4	FI	DI	FO	EI	WO	
I5		FI	DI			
I6	3 penalty		FI	DI		
I7				FI	DI	
I8						
I9						FI

OR

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I1	FI	DI	FO	EI	WO										
I2		FI	DI	FO	EI	WO									
I3			FI	DI	FO	EI	WO								
I4				FI	DI	FO	EI	WO							
I5		Stall	—	FI	DI										
I6		Stall	—	—	FI	DI									
I7		Stall	—	—	—	FI									
I9							FI	DI	FO	EI	WO				
I10								FI	DI	FO	EI	WO			
I11									FI	DI	FO	EI	WO		
I12										FI	DI	FO	EI	WO	

Q In a computer system, four files of size 11050 bytes, 4990 bytes, 5170 bytes and 12640 bytes needs to be stored. For storing these files on disk, we can use 100 byte disc blocks or 200 byte disc blocks (but can't mix block sizes). For each block used, 4 bytes of bookkeeping information needs to be stored. Thus, total space used to store a file is sum of space taken to store file and space to store bookkeeping information for eathe blocks allocated for storing the file. A disk block can store either bookkeeping information for a file or data from a file, but not both. What is the total space required for storing files

using 100 byte disk blocks and 200 byte disk blocks respectively.

1) 100 byte disk blocks :

file 1: 11050 bytes ⇒ Blocks for data = $\left\lceil \frac{11050}{100} \right\rceil = 111$

Blocks for bookkeeping = $\left\lceil \frac{111 \times 4}{100} \right\rceil = 5$

file 2: 4990 bytes ⇒ Blocks for data = $\left\lceil \frac{4990}{100} \right\rceil = 50$

Blocks for bookkeeping = $\left\lceil \frac{50 \times 4}{100} \right\rceil = 2$

file 3: 5170 bytes ⇒ Blocks for data = $\left\lceil \frac{5170}{100} \right\rceil = 52$

Blocks for bookkeeping = $\left\lceil \frac{52 \times 4}{100} \right\rceil = 3$

file 4: 12640 bytes ⇒ Blocks for data = $\left\lceil \frac{12640}{100} \right\rceil = 127$

Blocks for bookkeepis = $\left\lceil \frac{127 \times 4}{100} \right\rceil = 6$

∴ Total Blocks of 100 bytes required = (111+5) + (50+2) + (52+3) + (127+6)

= 356

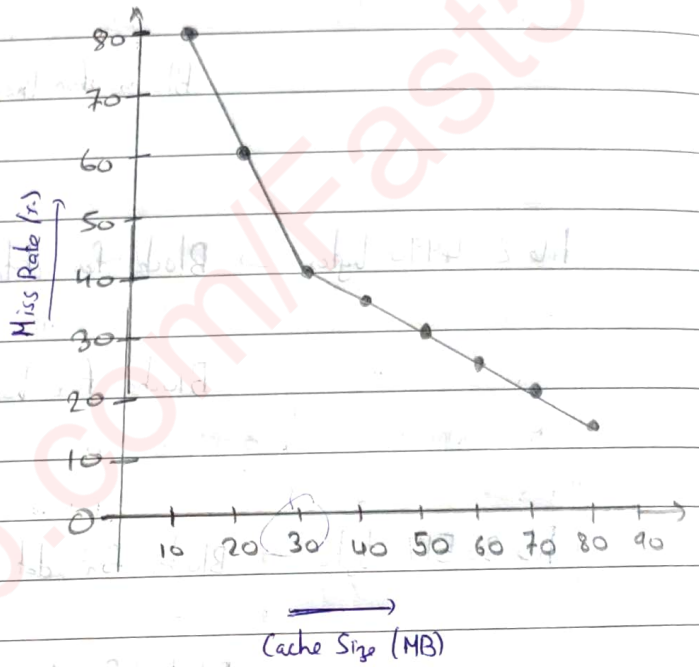
∴ Total Space required = 356 × 100 = 35600 bytes

2) Similar Procedure ....

∴ Total Blocks of 200 bytes required = 177

∴ Total space required = 177 × 200 = 35400 bytes

Cache Q A file system uses an in-memory cache to cache disk blocks. The miss rate of the cache is shown in the figure. The latency to read a block from the cache is 1 ms and to read a block from the disk is 10ms. Assume that the cost of checking whether a block exists in the cache is negligible. Available cache sizes are in multiples of 10MB. The smallest cache size required to ensure an average read latency of less than 6ms is ____ MB.



Assume miss rate to be m.
So, hit rate will be (1-m).

For a hit → Read latency from cache = 1ms
For a miss → Read latency from disk = 10ms

For required latency of 6ms:

$t_{avg} \leq 6$

$(1-m) \times 1 + m \times 10 \leq 6$

$m \leq 0.556$

Miss rate ≤ 55%.

From figure : Cache size = 30 MB

Processor with less participation of

Direct memory access helps I/O devices to directly access memory

disk Q The size of the data count register of a DMA controller is 16 bits. The processor needs to transfer a file of 29,154 KB from disk to main memory. The memory is byte addressable. The minimum number of times the DMA controller needs to get the control of the system bus from the processor to transfer the file from the disk to main memory is ____.

Data count register provides the number of words DMA can transfer in 1 cycle.
If memory is byte addressable ⇒ 1 word = 1 byte

∴ Size of the file = 29154 KB = 29154×2^{10} bytes

Size of data counter register of DMA controller = 16 bits

$\therefore 2^{16}$ words can be transferred in 1 cycle

$\therefore 2^{16}$ bytes can be transferred in 1 cycle (1 word = 1 byte)

$\therefore$ Minimum number of times DMA controller needs bus $= \left\lceil \frac{291654 \times 2^{10}}{2^{16}} \right\rceil = 256$

Q How many 32K×1 RAM chips are needed to provide a memory capacity of 256 K-bytes?

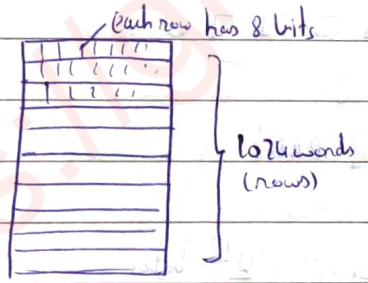
Memory required = 256 KB = $256 \times 2^{10} \times 8$ bits

RAM chip available = $32 \times 2^{10} \times 1$ bits

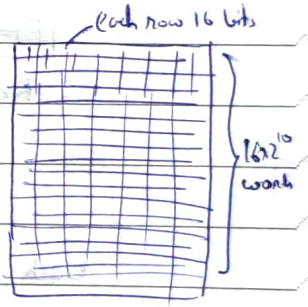
Number of chips required = $\frac{2^{21}}{2^{15}} = 2^6 = 64$

Q A RAM chip has a capacity of 1024 words of 8 bits each (1K×8). The number of 2×4 decoders with enable line needed to construct a 16K×16 RAM from 1K×8 RAM is?

1 RAM chip : 1K×8 :

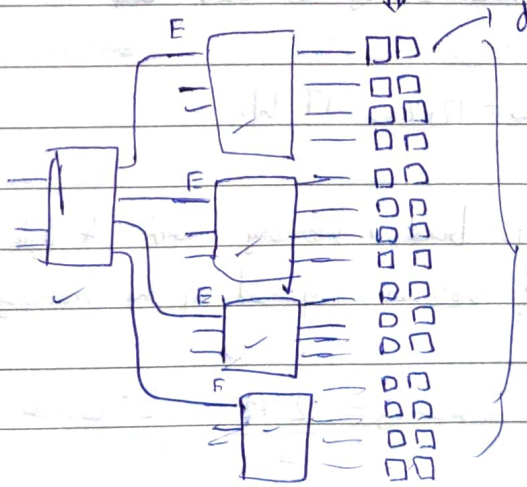


Required RAM : 16K×16



Construct how?

We can join two 1K×8 RAMs to make 16 bits



5 → 2×4 decoders required

decoder needed : 4×16

But we can only use 2×4

16×2^{10} words

RAM

Q A dynamic RAM has a memory cycle time of 64 ns. It has to be refreshed 100 times per ms and each refresh takes 100ns. What percentage of the memory cycle time is used for refreshing?

Memory cycle time = 64ns = 64×10^{-6} ms

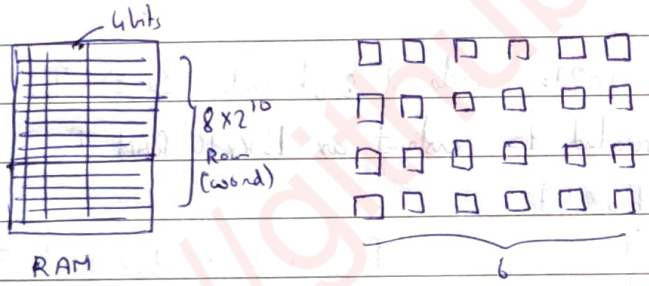
No. of refreshes per memory cycle = $100 \times 64 \times 10^{-6}$.

∴ Total refresh time per memory cycle = $64 \times 10^{-6} \times 100 \times 100$ ns = 64×10^{-2} ns

% memory cycle time for refreshing = $\frac{64 \times 10^{-2}}{64} \times 100 = 1\%$.

RAM

Q If each address space represents one byte of storage space, how many address lines are needed to access RAM chips arranged in a 4×6 array, where each chip is 8K×4 bits?



→ To select a row = 2 bits

To select a column = 6 bits $\lceil \log_2 6 \rceil$ = 3 bits

Each RAM chip = $8 \times 2^{10} \times 4$ bits = $\frac{32 \times 2^{10}}{8} = 2^{12}$ bytes

∴ Bits required to select one byte in each chip = 12 bits

∴ No. of address lines = 12+2+3 = 17 bits

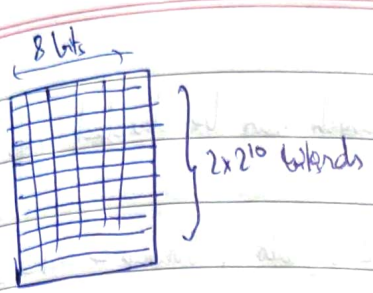
RAM

Q Suppose you want to build a memory with 4 byte words with a capacity of 2^{21} bits. What is the type of decoder required if the memory is built using 2K×8 RAM chips?

Total no. of Words in memory = $\frac{2^{21}/2^3}{4} = 2^{16}$ words

1 word → 4 bytes

2^{16} words →



RAM chip

Since each words consists of 4 bytes i.e. 32 bits, we will have 4 RAM chips in same row.

Required Memory = $2^{16} \times 32$

RAM chip = $2^{14} \times 8$

$\therefore$ No. of RAM chips required = $\frac{2^{16} \times 32}{2^{11} \times 8} = 32 \times 4$ (↑ rows, ↑ columns)

$\therefore$ Decoder required to select rows = 5×32

(5×2^5 decoder)

Bus

Q A processor can support a maximum memory of 4 GB, where the memory is word-addressable (a word consist of two bytes). The size of the address bus of the processor is atleast __ bits.

Maximum memory = 4 GB = 4×2^{30} Bytes

$\therefore$ No. of words (address) = $\frac{4 \times 2^{30}}{2} = 2^{31}$ words

$\therefore$ No. of address bits required = 31 bits

Hardwired control

Q A CPU has only three instructions I_1, I_2 and I_3, which use the following signals in time steps T1-T5:

	I_1	I_2	I_3
T1	Ain, Bout, Cin	Cin, Bout, Din	Din, Aout
T2	PCout, Bin	Aout, Bin	Ain, Bout
T3	Zout, Ain	Zout, Ain	Zout, Ain
T4	Bin, Cout	Bin, Cout	Dout, Ain
T5	End	End	End

Which logic function will generate the hardwired control for the signal Ain? $T3 + T1 \cdot I_1 + T2 \cdot I_3 + T4 \cdot I_3$

Ain

Microprogrammed Control

Q An instruction set of a processor has 125 signals, which can be divided into 5 groups of mutually exclusive signals as follows:
Group 1 → 20 signals, Group 2 → 70 signals, Group 3 → 2 signals, Group 4 → 10 signals, Group 5 → 23 signals
How many bits of the control words can be saved by using vertical microprogramming over horizontal microprogramming?

Horizontal Microprogramming:  ⇒ 125 bits

Vertical Microprogramming:

	G1	G2	G3	G4	G5
Signals	20	70	2	10	23
bits	5	7	1	4	5

∴ Total bits = 22

∴ Bits saved = 125 − 22 = 103 bits

Microprogram Control

Q A micro-instruction format has micro-ops field which is divided into three subfields F1, F2, F3 each having seven distinct micro-operations, condition field CD for four status bits, branch field BR having four options used in conjunction with address field ADF. The address space is of 128 memory locations. The size of micro-instruction is ____

	F1	F2	F3	CD	BR	ADF
Options	7	7	7	4	4	128
Bits	3	3	3	2	2	7

⇒ 20 bits

[Logic]

PSW

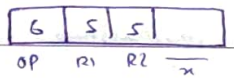
Q A processor that has carry, overflow, and sign flag bits as a part of its program status word (PSW) performs addition of the following two 2's complement numbers 0100 1101 and 1110 1001. After the execution of this addition operation, the status of the carry, overflow and sign flags, respectively will be:

Note: In 2's complement representation, n bits can represent values in the range -2^{n-1} to $2^{n-1}-1$.
Overflow can occurs when both summands have the same sign → if +ve +ve = -ve, -ve -ve = +ve then overflow has occurs

```
  (0)100 1101   → +ve
+ (1)110 1001   → -ve   ] → so overflow cannot happen
  (1)011 0110
```

Carry: 1
Overflow: 0
Sign: 0
Zero: 0

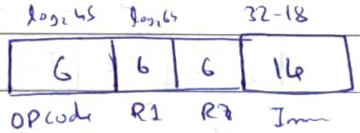
ISA Q A processor has 40 distinct instructions and 24 general purpose registers. A 32-bit instruction word has an opcode, two register operands and an immediate operand. The number of bits available for the immediate operand field is —



32 - 11 = 21

40 distinct instructions → $\lceil \log_2 40 \rceil = 6$ bits required for opcode
24 general purpose registers → $\lceil \log_2 24 \rceil = 5$ bits required for register operand
∴ Immediate Operand field = 32 - 6 - 5 - 5 = 16 bits

ISA Q A machine has a 32-bit architecture, with 1-word long instructions. It has 64 registers, each of which is 32 bit long. It needs to support 45 instructions, which have an immediate operand in addition to two register operands. Assuming that the immediate operand is an unsigned integer, the maximum value of the immediate operand is:



32-bit architecture → 1 word contains 32 bits
∴ Instruction length = 1 word = 32 bits

Unsigned-integer → n bits → range of value: 0 to $2^n - 1$
16 bits → range of values: 0 to $2^{16}-1$
max value = $2^{16}-1 = 16383$

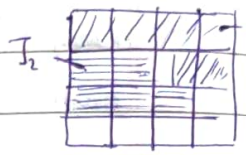
ISA Q Consider a processor with 64 registers and an instruction set of size twelve. Each instruction has five distinct fields, namely opcode, two source register identifiers, one destination register identifier, and a twelve bit immediate value. Each instruction must be stored in memory in a byte-aligned fashion. If a program has 100 instructions, the amount of memory (in bytes) consumed by the program is ____.

Instruct Set = 12 instructions ⇒ Opcode = $\lceil \log_2 12 \rceil = 4$

Opcode	R_1	R_2	R_3	imme
4	6	6	6	12

∴ Instruction length = 4+6+6+6+12 = 34 bits

∴ Each Instruction length = $\frac{34}{8}$ = 4.25 bytes = 5 bytes (byte aligned fashion)



∴ Size of memory = 100 × 5 = 500 bytes

ISA Q A processor has 16 integer registers (R0, R1, .., R15) and 64 floating point registers (F0, F1, .., F63). It uses a 2-byte instruction format. There are four categories of instructions: Type-1, Type-2, Type-3 and Type-4. Type-1 category consists of four instructions, each with 3 integer register operands. Type-2 category consists of eight instructions, each with 2 floating point register operands. Type-3 category consists of fourteen instructions, each with one integer register operand and one floating point register operand. Type-4 category consists of N instructions, each with a floating point register operand. The maximum value of N is:

Instruction length = 2 Bytes = 16-bit ∴ Total Instruction = 2^{16}

Type 1: (4) | Opcode | IR | IR | IR | ⇒ $4 \times 2^4 \times 2^4 \times 2^4$

Type 2: (8) | Opcode | FR | FR | ⇒ $8 \times 2^6 \times 2^6$

Type 3: (14) | Opcode | IR | FR | ⇒ $14 \times 2^4 \times 2^6$

Type 4: (N) | Opcode | FR | ⇒ $N \times 2^6$

∴ $4 \times 2^4 \times 2^4 \times 2^4 + 8 \times 2^6 \times 2^6 + 14 \times 2^4 \times 2^6 + N \times 2^6 \le 2^{16}$

∴ $N_{max} = 32$

OR ④ → 16 Type 1 ✓ Type 2 ✓

④ → ⑫ → ② → 4 Type 3 ✓

⑥ → 2 + ④ → 16 Type 4 ⇒ 16×2 = 32

Q The program below uses six temporary variables a, b, c, d, e, f. Assuming that all operations take their operands from registers, what is the minimum number of registers needed to execute this program without spilling?

```
a=1
b=10
c=20
d=a+b
e=c+d
f=c+e
b=c+e
e=b+f
d=5+e
return d+f
```

	R_1	R_2	R_3
	a	b	c
d=a+b	d=a+b		
e=c+d	e=c+d		
f=c+e		f=c+e	
b=c+e			b=c+e
e=b+f	e=b+f		
d=5+e	d=5+e		

Magnetic Disc Q What is the average access time for transferring 512 bytes of data with the following specification:

Avg Seek time = 5ms

Disk Rotation = 6000 rpm

Data Rate = 40KB/s

Controller overhead = 0.1ms

Time taken for one full rotation = $\frac{60}{6000}$ = 10 ms

6000 R → 60

1

Average Rotational Delay = $\frac{1}{2}$ × (Time taken for one full rotation) = 5ms

Transfer Time = $40 \times 2^{10} \rightarrow 1s$ = 12.5ms

$512 \rightarrow \frac{512}{40 \times 2^{10}}$

∴ Average Access Time = Avg Seek time + Average Rotational delay + Transfer time + Controller overhead + Queuing delay

= 5 + 5 + 12.5 + 0.1 + 0

= 22.6 ms

Magnetic Disc

Q A certain moving arm disk storage with one head has the following specifications: Calculate the Average latency & Data transfer rate.
Number of tracks per surface = 200
Disk rotation speed = 2400 rpm
Track storage capacity = 62500 bits

Time taken for one full rotation = $\frac{60}{2400}$ = 25 ms

2400 → 60
1 →

Average latency = $\frac{1}{2}$ x (Time taken for one Full rotation) = 12.5 ms

Data transfer rate = Number of heads x Capacity of one track x Number of rotations in 1 sec
= $1 \times 62500 \times \left(\frac{2400}{60}\right)$

= 2.5×10^6 bits/sec

ROM

Q The address bus with a ROM of size 1K×8 is 10.

1024 words and Each word has 8 bits
↓ ↓
$2^{(10)}$ → address bus — data bus width

Cache

Q A hierarchical memory system has the following specification, 20 MB main storage with access time of 300 ns, 256 bytes cache with access time of 50 ns, word size 4 bytes, page size 8 words. What will be the hit ratio if the page address trace of a program has the pattern 0,1,2,3, 0,1,2,4 following LRU page replacement technique?

Since Block size (line size) = Page size = 8 words = 32 bytes

∴ No. of blocks (lines) in Cache = $\frac{256}{32}$ = 8 blocks

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

∴ Hit Rate = $\frac{3}{8}$

Expanding Opcode

Q A certain machine has 16-bit instructions and 6-bit addresses. Some instructions have one address and others have two. If there are n two-address instructions, what is the maximum number of one-address instruction?

Total no. of Instruction = 2^{16}

n-two address instruction = $n \times 2^6 \times 2^6$

maximum no. of one address instruction = N

$\therefore (N - n) \times 2^6 + n \times 2^6 \times 2^6 \leq 2^{16}$ ⇒ Expanding Opcode se aaya

$\therefore N - n + 64n \leq 2^{10}$

$\therefore N - n \leq 2^{10} - 64n$

$\therefore (N-n)_{max} = (16-n)64$

Expanding Opcode

Q A processor has 64 registers and uses 16-bit instruction format. It has two types of instructions: I-type and R-type. Each I-type instructions contains an opcode, a register name, and a 16-bit immediate value. Each R-type instruction contains an opcode and two register names. If there are 8 distinct I-type opcodes, then the maximum no. of distinct R-type opcode is ____

Borrow register from opcode

I-type | Opcode | Reg | Imm |
(8) | 6 | 6 | 16 |

R-type | Opcode | Reg | Reg |
(x) | 4 | 6 | 6 |

$(2^4 - x) \times 2^2 = 8$

$8 \times 2^6 \times 2^6 + N \times 2^6 \times 2^6 \leq 2^{16}$

$8 + N \times 2^2 \leq 2^4$

$2 + N \leq 2^4$

$N \leq 14$

expanding opcode

Q Using an expanding opcode encoding for instructions, is it possible to encode all of the following in an instruction format shown in the below figure.

14 - two address instructions → | 4 | 6 | 6 |

127 - one address instructions → | OP | R1 | R2 | 10 | 6 |

60 - zero address instructions → | 16 |

$[(2^6 - 14) \times 2^6 - 127] \times 2^6 = 64$ zero address instructions possible

RISC

Q A RISC processor that uses the five step sequence: F, D, E, M, W is driven by a 1-GHz clock. Instruction statistics in a large program are as follows:

Branch 20%

Load 20%

Store 10%

Computational Instructions 50%

Estimate the rate of instruction execution in each of the following cases:

a) Access to the memory is always completed in 1 clock cycle.

b) 90% of instruction fetch operations are completed in one clock cycle and 10% are completed in 4 clock cycles. On average, access to the data operands of a load or store instruction is completed in 3 clock cycles.

a) Memory access (for F, M) is completed in one clock cycle. Here, each of the above four classes of instructions would be executed in exactly five steps each taking 1 clock cycle.

1 clock cycle = $\frac{1}{1\text{GHz}}$ = 1ns

Suppose there are 100 instructions. 20 Branch Inst → 20×5×1 = 100 clock cycle = 100ns

20 Load Inst → 20×5×1 = 100 clock cycle = 100ns

10 Store Inst → 10×5×1 = 50 clock cycle = 50ns

50 ALU Inst → 50×5×1 = 250 clock cycle = 250ns

∴ Therefore, in total all this 100 instructions would take = 500ns

∴ Rate of Instruction Execution $= \frac{100}{500 \times 10^{-9}} = 200 \times 10^6/\text{s} = 200$ MIPS

b) From part (a), the execution time for the F stage of 10% instructions would increase by 3 clock cycles because previously it took 1 clock cycle, so now it would take = 3+1 = 4cc

Similarly, execution time for M stage of all load and store instructions would increase by 2 cc because previously it took 1 cc, so now it would take = 2+1 = 3cc

Therefore, in total all the 100 instructions would take = 500 + 10×3cc + (10+20)×2cc
= 500 + 30 + 60
= 590 ns

∴ Rate of Instruction Execution = $\frac{10^9}{590 \times 10^{-9}} = \frac{10^9}{590} \times 10^9 = 170$ MIPS

Machine

Q Compare zero-, one-, two-, three-address and the load-store machines by writing programs to compute : $X = C*B - B*C*D$, $Y = (X + A*D)/(B-X)$, for each of the 5 machines. The instructions available for use are as follows. Each program should contain code for both of the above assignment statement and should not write out the values in A, B, C, or D.

3 Address
MOVE (X ← Y)
ADD (X ← Y+Z)
SUB (X ← Y−Z)
MUL (X ← Y*Z)
DIV (X ← Y/Z)

2 Address
MOVE (X ← Y)
ADD (X ← X+Y)
SUB (X ← X−Y)
MUL (X ← X*Y)
DIV (X ← X/Y)

1 Address (Accumulator)
LOAD M
STORE M
ADD M
SUB M
MUL M
DIV M

SUB M : AC = AC − M
DIV M : AC = AC / M

0 Address (Stack)
PUSH M
POP M
ADD
SUB
MUL
DIV

SUB : POP T
POP T2
T3 = T2 − T
PUSH T3

DIV : POP T
POP T2
T3 = T2 / T
PUSH T3

Assume 8-bit opcodes, 32-bit absolute addressing, 5-bit register numbers (used only for load-store machine), and 32-bit (data) operands. Compute the number of bits needed during program execution for each programs.

$$X = C * B - B * C * D$$
$$Y = (X + A * D) / (B - X)$$

3 Address Machine:

	ML Instruction Size (Read)	Data Transferred during execution
MUL X, C, B	8+32+32+32 = 104	32+32+32 = 96
MUL T, B, C	104	96
MUL T, T, D	104	96
SUB X, X, T	104	96
MUL T, A, D	104	96
ADD Y, X, T	104	96
SUB T, B, X	104	96
DIV Y, Y, T	104	96
	8×104 = 832 bits	8×96 = 768 bits

∴ Total Bits = 832 + 768 = 1600 bits

2 Address Machine:

	ML Instruction Size	Data transferred during execution
MOV X, C	8+32+32 = 72	32+32 = 64
MUL X, B	72	32+32+32 = 96
MOV T, B	72	64
MUL T, C	72	96
MUL T, D	72	96
SUB X, T	72	96
MOV Y, A	72	64
MUL Y, D	72	96
ADD Y, X	72	96
MOV T, B	72	64
SUB T, X	72	96
DIV Y, T	72	96
	12×72 = 864 bits	4×64 + 8×96 = 1024 bits

∴ Total Bits = 864 + 1024 = 1888 bits

1 Address Machine:

	ML Instruction Size	Data transferred during execution
LOAD B	8+32=40	32
MUL C	40	32
MUL D	40	32
STORE X	40	32
LOAD C	40	32
MUL B	40	32
SUB X	40	32
STORE X	40	32
LOAD B	40	32
SUB X	40	32
STORE Y	40	32
LOAD A	40	32
MUL D	40	32
ADD X	40	32
DIV Y	40	32
STORE Y	40	32
	16x40=640 bits	16x32=512 bits

∴ Total bits = 640+512 = 1152 bits

X=C*B−B*C*D → Inorder

(CB*)−(BC*)*D

(CB*)−(BC*D*)

CB*BC*D*− → Postorder
Left → right

(*CB)−B*(*CD)

(*CB)−*B*CD

−*CB*B*CD → Preorder
Right → left

0 - Address Machine:

	MI Instruction Size	Data transferred during execution
PUSH C	8+32=40	32
PUSH B	40	32
MUL	8	0
PUSH B	40	32
PUSH C	40	32
MUL	8	0
PUSH D	40	32
MUL	8	0
SUB	8	0
POP X	40	32
PUSH X	40	32
PUSH A	40	32
PUSH D	40	32
MUL	8	0
ADD	8	0
PUSH B	40	32
PUSH X	40	32
SUB	8	0
DIV	8	0
POP Y	40	32
	12x40+8x8=544 bits	12x32 = 384 bits

∴ Total bits = 544+384 = 928 bits

LOAD STORE Machine:

	ML Instruction Size	Data transferred during execution
LOAD R1, C	8+5+32 = 45	32
LOAD R2, B	45	32
MUL R6, R1, R2	8+5+5+5 = 23	0
LOAD R3, D	45	32
MUL R5, R2, R1	23	0
MUL R5, R5, R3	23	0
SUB R6, R6, R5	23	0
STORE R6, X	45	32
LOAD R4, A	45	32
MUL R5, R4, R3	23	0
ADD R5, R6, R5	23	0
SUB R2, R2, R6	23	0
DIV R1, R5, R2	23	0
STORE R1, Y	45	32
	6×45 + 8×23 = 454 bits	6×32 = 192 bits

∴ Total bits = 454 + 192 = 646 bits

Cache

Q. A computer system has a three level memory hierarchy, with access time and hit ratios as shown below:

Level 1 (Cache Memory) → Access Time = 50ns/byte

Size	Hit ratio
8M bytes	0.80
16M bytes	0.90
64M bytes	0.95

Level 2 (Main memory) → Access Time = 200ns/byte

Size	Hit ratio
4M bytes	0.98
16M bytes	0.99
64M bytes	0.995

Level 3 → Access Time = 5μs/byte

Size	Hit ratio
260M bytes	1.0

a) What should be the minimum sizes of level 1 and level 2 memories to achieve an average access of time of less than 100ns?

b) What is the average access time achieved using the chosen sizes of level 1 and level 2 memories?

a) We know that average memory access time for 2-level memory is given by $t_{avg} = h_1 t_1 + (1-h_1)$

Similarly, average access time for 3-level memory is given by $t_{avg} = t_1 + (1-h_1)t_2 + (1-h_1)(1-h_2)$

$t_1 \rightarrow$ access time for L_1

$h_1 \rightarrow$ hit rate for L_1

$h_2 \rightarrow$ hit rate for L_2

Here $t_1 = 50ns$, $t_2 = 200ns$, $t_3 = 5000ns$ & $t_{avg} < 100$

Case 1: $L_1 = 8M$ bytes, $L_2 = 4M$ bytes, $h_1 = 0.8$, $h_2 = 0.98$

$\therefore t_{avg} = 50 + (1-0.8)\,200 + (1-0.8)(1-0.98) \times 5000$

$= 110ns \times (>100)$

Case 2: $t_1 = 8M$ bytes, $L_2 = 16M$ bytes, $h_1 = 0.8$, $h_2 = 0.99$

$\therefore t_{avg} =$

$= 100ns \times$

Case 3: $L_1 = 8M$ bytes, $L_2 = 64M$ bytes, $h_1 = 0.8$, $h_2 = 0.995$

$\therefore t_{avg} =$

$= 95ns$ ✓ can be but we need to check other case, as we want memory sizes to be minimum

Case 4: $L_1 = 16M$ bytes, $L_2 = 4M$ bytes, $h_1 = 0.9$, $h_2 = 0.98$

$\therefore t_{avg} =$

$= 80ns$ ✓

$\therefore L_1 = 16MB$, $L_2 = 4MB$ will be the minimum possible sizes of both level-1 & level-2 memory

b) 80ns

Q An instruction pipeline has five stages where each stage takes 2ns and all instructions use all five stages. Branch instructions are not overlapped i.e. the instruction after the branch is not fetched till the branch instruction is completed.
Under ideal conditions,

a) Calculate the average instruction execution time assuming that 20% of all instruction executed are branch instructions. Ignore the fact that some branch instructions may be conditional.

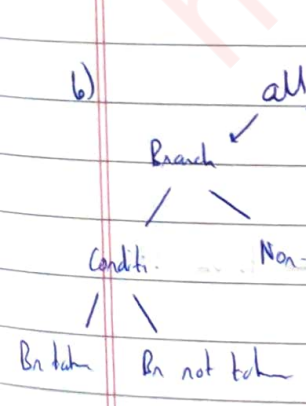
b) If a branch instruction is a conditional branch instruction, the branch need not be taken. If the branch is not taken, the following instructions can be overlapped. When 80% of all branch instructions are conditional branch instructions, and 50% of the conditional branch instructions are such that the branch is taken, calculate the average instruction execution time.

Each stage is 2ns. So, after 5 time units each of 2ns, the first instruction finishes (i.e. after 10ns), in every 2ns after that a new instruction gets finished. This is assuming no branch instructions. ~~Now, once the pipeline is full, we can~~

If an instruction branches then it takes 2ns × 5 = 10ns, coz if branch is taken then the instruction after that branch instruction is not fetched until entire current branch instruction is completed, this means it will go through all stages.
If an instruction is not branch then, it takes 2ns to get completed

a) Avg time taken = $\frac{80}{100} \times 2ns + \frac{20}{100} \times 10ns = 3.6ns$

b)



Avg time taken = $\frac{80}{100} \times 2ns + \frac{20}{100}\left(\frac{80}{100}\left(\frac{50}{100} \times 10ns + \frac{50}{100} \times 2ns\right) + \frac{20}{100} \times 10ns\right)$

$= 2.96ns$

I/O

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

The minimum performance gain for interrupt mode happens for the smallest unit of data transfer - which here is 1 byte.

Time to transfer 1 byte of data in programmed I/O mode $= \frac{1}{10KBps} = 100\,\mu s$

In interrupt mode, to transfer 1 byte of data, overhead is 4μs

$\therefore$ Performance Gain $= \frac{100}{4} = 25$

Q Figure gives part of μinstructions sequence corresponding to one of the machine instructions of a microprogrammed computer. Micro instruction B is followed by C, E, F or I, depending on bits b_6 and b_5 of the machine instruction register. Compare three possible implementations described below:

a) Micro instruction sequencing is accomplished by means of a μPC. Branching is achieved by microinstructions of the form

If b_6b_5 branch to X

(condition, branch address)

b) Same as Part 'a' except that the branch instruction has the form

Branch to X, OR

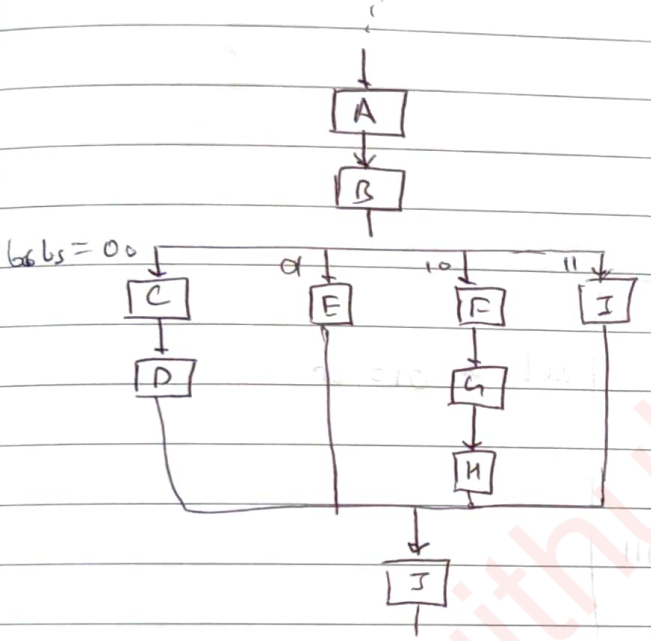
(bare branch address)

The branch address is modified by bit-ORing of bits b_5 and b_6 with the appropriate bits within X.

c) A field in each micro instruction specifies the address of the next μ instruction, which has bit-ORing capability

For example: in Part a, you could choose addresses as follows: (arbitrary)

Addrs	Micro instruction
0010	A
0011	B
0100	If bbs=00 branch to xx
...	...
xx	C



a)

Addrs	Micro instruction
0000	A
0001	B
0010	if (bbs=00) then μBranch 0111
0011	if (bbs=01) then μBran 1010
0100	if (bbs=10) then μBranch 1100
0101	I
0110	μBranch 1111
0111	C
1000	D
1001	μBranch 1111
1010	E
1011	μBranch 1111
1100	F
1101	G
1110	H
1111	J

b) Assume the bits bb-s of IR are ORed into bit μPC_{3-2}

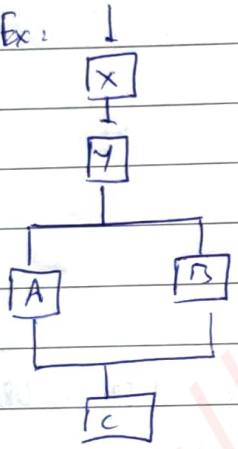
Addrs	μInst
0000	A
0001	B $\mu PC_{3-2} \leftarrow bb_{-s}$
0010	C
0011	D
0100	μBran 1111
0101	E
0110	μBra 1111
1010	F
1011	G
1100	H
1101	μBran 1111
1110	I
1111	J

μInst

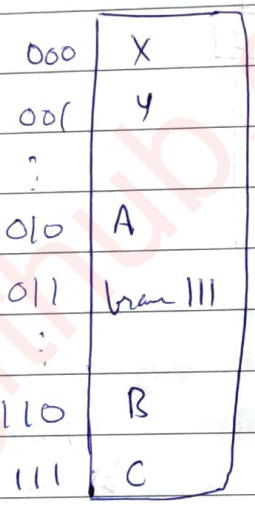
c) Addr | Next addr | Fn

Addr	Next addr	Fn
0000	0001	A
0001	0010	B μPC(3-2) ← b6-5
0010	0011	C
0011	1111	D
0110	1111	E
1010	1011	F
1011	1100	G
1100	1111	H
1110	1111	I
1111	—	J

Ex:



⇒



μBranch 010, or

Control Store

000 – 002

003 μBranch (μPC ← 010)

octal 010 if (IR10-8 = 000) then μBranch {μPC ← 101} → dr

: