Lecture - 01

13 Jan, 2020

Computer Organization and Design
D. A. Patterson
5th edition, Morgan Kaufmann

- DLD
- Simplifications (not k-map; use Boolean Algebra)
- MUX (multiplexers design)
- ALU (arithmetic logic unit)
- Adder
- Registers / Shift registers
- Internal Structure of Register File

32-bit Architecture

$x = y + z$ (x → \$10, y → \$11, z → \$12)

Add \$10, \$11, \$12 — R
D S_1 S_2

R
I
J

OP	R_s	R_t	Rd	Shmt	fun
000000	01011	01100	01010	00000	******
6	5	5	5	5	6

feed this and this is what the processor will understand

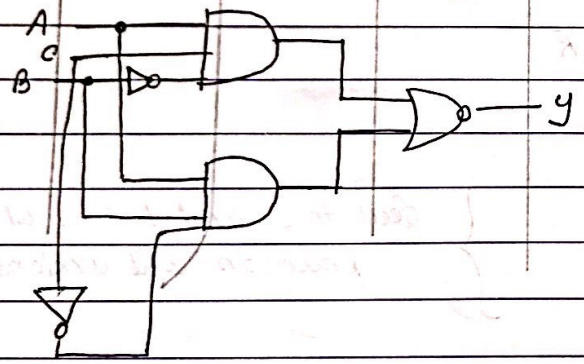
- convert into assembly instructions and identify types (R, I, J)
- each form has diff types of encoding
- all the instructions will be 32-bits.

Lecture - 02 15-01-2020

$y = \overline{A.B.C}\,(A.B + A.C)$

A	B	C	A.B.C	$\overline{A.B.C}$	A.B	A.C	A.B+A.C	y
0	0	0	0	1	0	0	0	0
0	0	1	0	1	0	0	0	0
0	1	0	0	1	0	0	0	0
0	1	1	0	1	0	0	0	0
1	0	0	0	1	0	0	0	0
1	0	1	0	1	0	1	1	1 → AB'C
1	1	0	0	1	1	0	1	1 → ABC'
1	1	1	1	0	1	1	1	0

SOP



$\bar{y}$
1 A'B'C'
1 A'B'C
1 A'BC'
1 A'BC
1 AB'C'
0
0
1 ABC

$\bar{y} = A'B'C' + A'B'C + ABC' + A'BC + AB'C' + ABC$

$\bar{\bar{y}} = y = \overline{A'B'C' + A'B'C + ABC' + A'BC + AB'C' + ABC}$

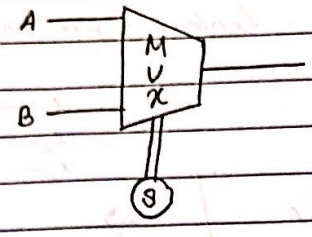
$\Rightarrow y = \overline{(A'B'C')}\ \overline{(A'B'C)}\ \overline{(ABC')}\ \overline{(A'BC)}\ \overline{(AB'C')}\ \overline{(ABC)}$

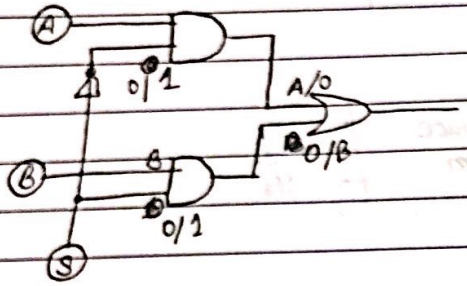
$= (A+B+C).(A+B+C').(A'+B'+C).(A+B'+C').(A'+B+C).(A'+B'+C')$

(product of sum)

POS

If S
Then y := B
ELSE y := A





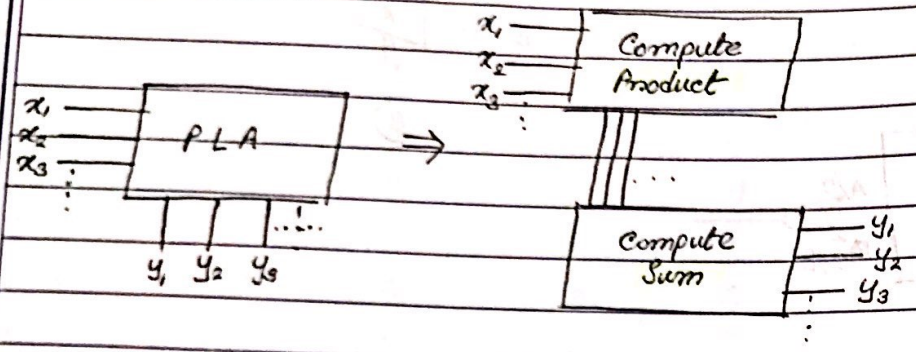
carry
& rippled
stage.

Lecture - 03

20-01-2020

PLA → Programmable Logic Array

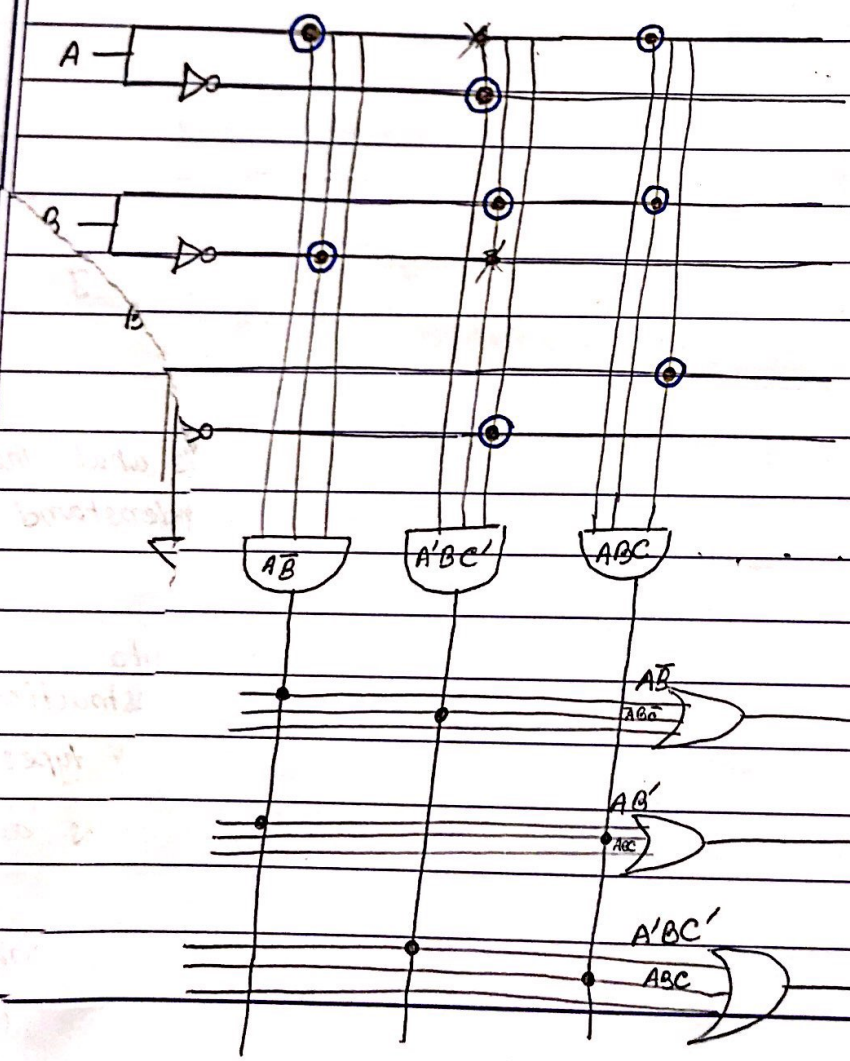
Functionality - boolean expressions



$y_1 = A\bar{B} + \bar{A}B\bar{C}$

$y_2 = A\bar{B} + ABC$

$y_3 = \bar{A}B\bar{C} + ABC$



Open Circuit

3 combinations in y_1, y_2, y_3:

$A\bar{B}$, $\bar{A}B\bar{C}$, and ABC

$y_1 = A\bar{B} + \bar{A}B\bar{C}$

$y_2 = A\bar{B} + ABC$

~~$y_3 = AB' + \bar{B}BC'$~~ A

$y_3 = A'BC' + ABC$

$y_1 = AB'C \oplus ABC$

$y_2 = ABC' \oplus A'BC$

$y_3 = A'B'C' \oplus AB'C$

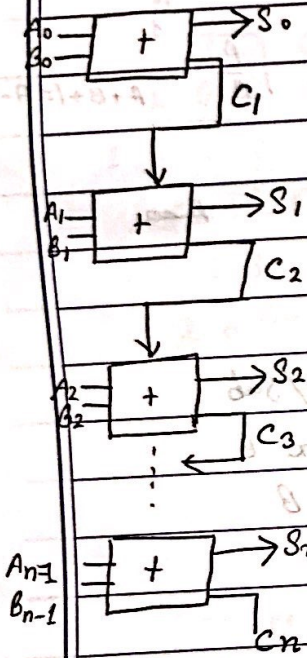
6 AND Gates
3 OR
⊕

PLA is always one circuit

✓ regardless of how many expressions, PLA is one circuit.

RCA

Ripple Adder (full adder)



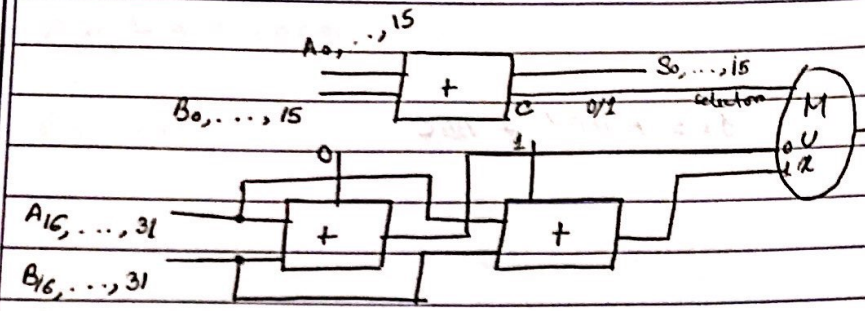
Ripple Carry Adder (RCA)

not efficient

It is called a ripple carry adder because each carry-bit gets rippled into the next stage.

The carry has to propagate from LSB to MSB.

Conditional Sum Adder / Fast Adder

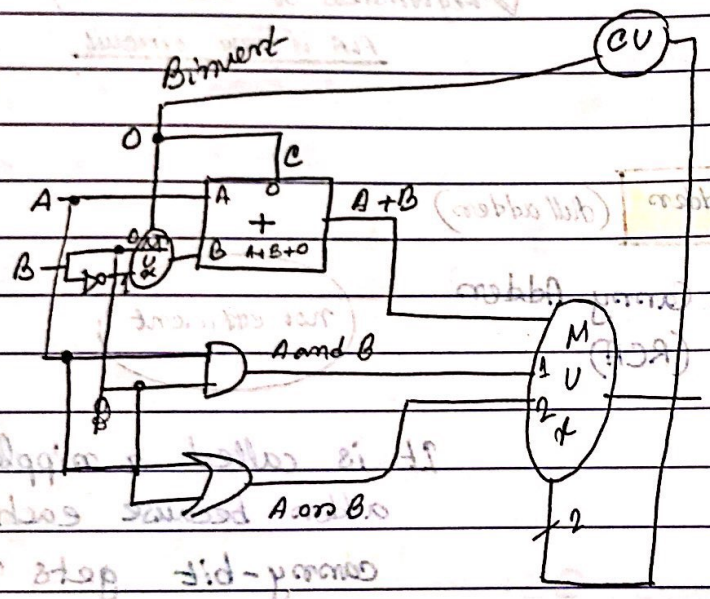


Fast Adder

CLA

$A - B = A + \bar{B} + 1$

$A + B$



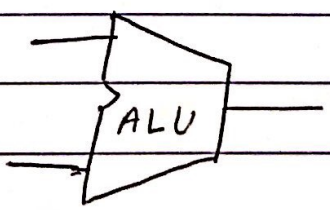
00: Add / Sub

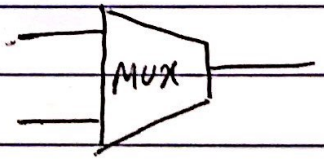
01: A and B

10: A or B

ALU - Arithmetic Logic Unit

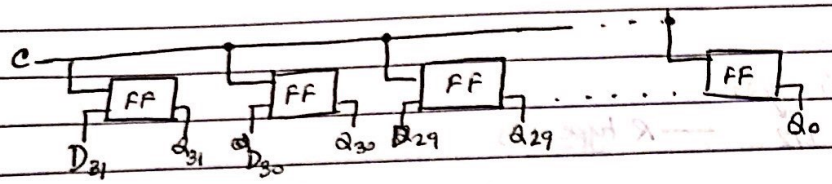
(2 mux)





To build the register, we need (flipflops.)

Register



32-bit register

- When we read, we've to read the entire register.

* exception: lb (allows to access first 8-bit of the register)
Byte load byte

- lb → load byte (allows to access the first 8 bits of the register)
otherwise, we've to take / read values for the entire register

1 0 0	1 0 0 0
1 0 0 0	0 1 0 0
1 0 0 0 0	0 0 1 0
left shift	right shift
L.S.	R.S.
$x * 2^i$	$x / 2^i$

i → no. of bit shift

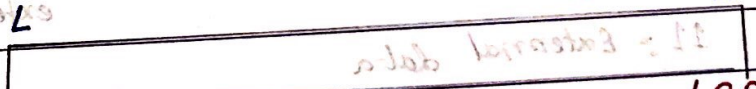
~~$x * 2^i$~~ L.S. $x * 2^i$ left shift operation (multp)

R.S. $x / 2^i$ right shift operation (div)

L ← R

31 30 29 . . . . . 2 1 0

L R



i starts at 0 and increases from R to L

MSB (bit 31) (32 bits wide) LSB (bit 0)

$x = 3y + 5z$

$y = 4x + 3z$

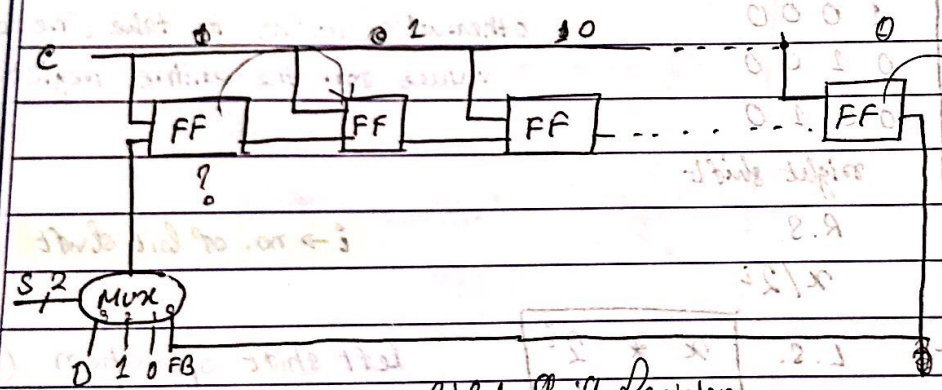
$z = 5x - 6y$

Add $10, $11, $12 — R type ins (D, S_1, S_2)

mult $11, $12 — R type ins

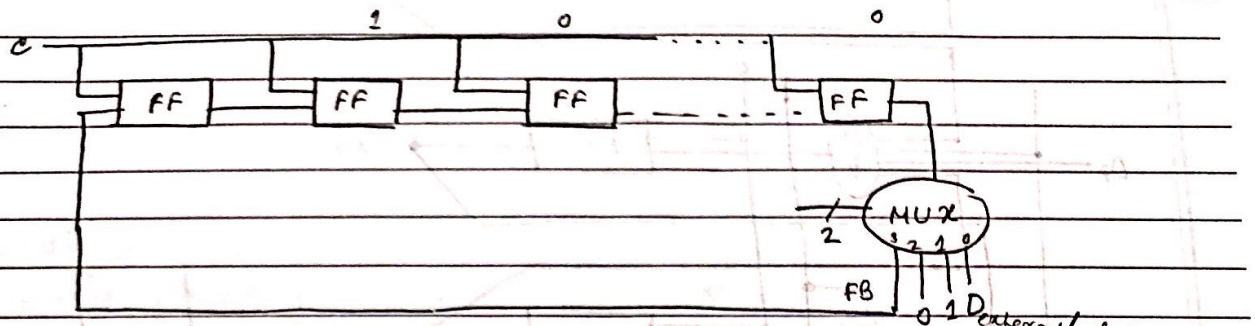
↳ Result Lo, Hi

Designing Right Shift Register.



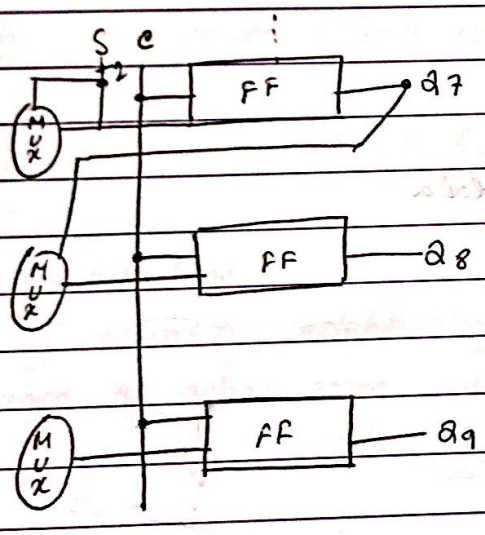
Right Shift Register

F	00 : Feedback	random no. coming from some external source
R	01 : Reset	
S	10 : Set	
E	11 : External data	



Left Shift Register

0	00 : External data
1	01 : Set
2	10 : Reset
3	11 : FB



00 : LS

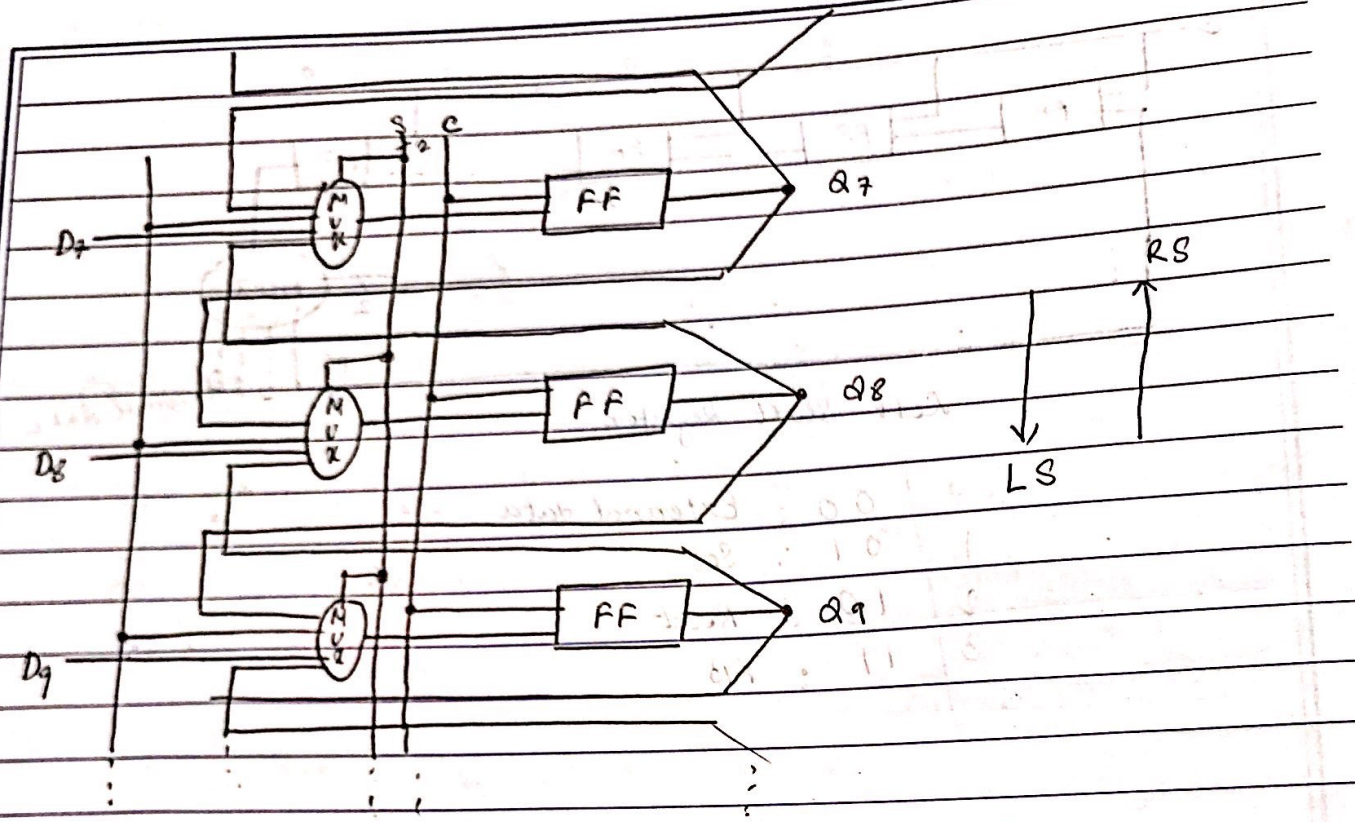
~~01 : RS~~

11 : RS

01 : Reset

10 : External data

See: next page



00 : LS

11 : RS

01 : Reset

10 : External data

* key: bucse340sp

Lecture-05

27th Jan, 2020

Chapter 1

- Moore's law
- API
- Interface
- Parallel Processing

high-level language —(Compiler)→ assembly instruction —(Assembler)→ machine instruction (this is what the machine understands)
(C or Java)

Application Software is written in high-level language.

Controller
Control Unit responsible for allowing CPU to carry out operations.

- datapath: components & path connecting CPU

Abstraction - hiding the detail from user

Computer Architecture
- designing, selecting, establishing interconnection
- interface to achieve some goals like functionalities.

set of instructions { Add, sub, bne, beq, $11, }

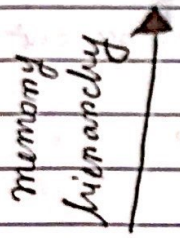
{ MIPS

x = y + z
↓ ↓ ↘
$10 $11 $12

Add $10, $11, $12
↓ ↘ ↘
Destination Register $1 $2

* definition's on the diagram.

• memory tends to get smaller higher-up. the memory hierarchy.



(and vice versa for down)

cost, speed, size

• multiprocessor shares a RAM / memory.

Role of the Architect

- backward (literature review, existing designs)
- forward
- look up
- look down (scale down the design for the current tech to accomodate the idea into reality) e.g. limitations with transistors

Lecture - 06

- 01 - 2020

We need trade-offs when considering performance.

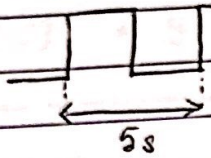
- Response time and throughput.
- Relative performance

$$\frac{P_x}{P_y} = \frac{E_y}{E_x} = n$$

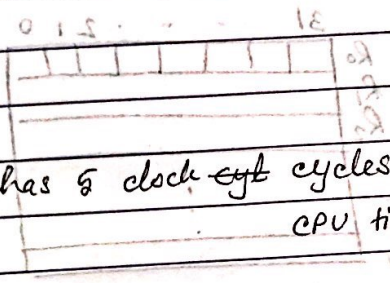
P → performance

E → execution time

→ x is n times faster than y



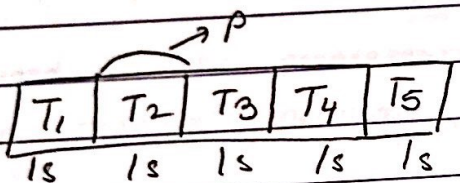
(one clock cycle)



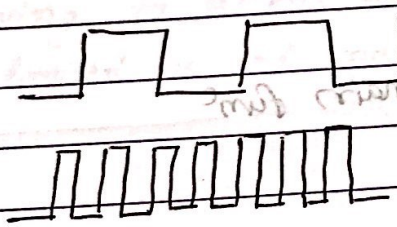
Q. If program has 5 clock cycles, then, CPU time?

Ans: CPU Time = 25 s

For comp B, we need higher clocks i.e. higher GHz.



R_E = 3+4×2 + other waiting



RISC vs. CISC

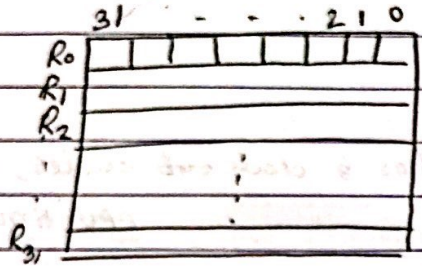
0302-2020

Lecture 07

Computer Language

Machine Language → Assembly Instruction

- identify the registers which store the values of variable



$t0 ~ $t9

$S0 ~ $S7

$$S \rightarrow f = \frac{(g+h)}{t} - \frac{(i+j)}{t}$$

* $0 / $zero

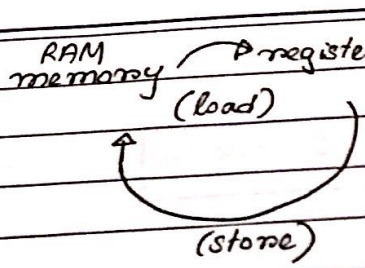
* $ra / $31 → give return address to main func

$sp → point to stack

$fp → frame-pointer points to stack

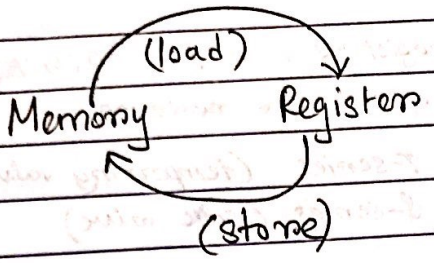
$a0 ~ $a3 → argument registers used in assembly instruction

$v0 ~ $v1 → return func



load (memory → register)

store (register → memory)



MR. load

store	load
A[10] = y + A[6]	

• memory is byte-addressed

Processor operations mostly involve processing data.

To speed up the processor operations, the processor includes some internal memory storage locations, called registers.

The registers store data elements for processing without having to access the memory.

A limited no. of registers are built into the processor chip.

Registers vs. Memory

- load and store operations are **Memory Operations.**

32 registers with each 32-bit.
Some of these are reserved.

T-series (temporary value) } available to us
S-series (save value) }

- spill
- optimization

$10 $11 $12
x = y + 2;

load, store, immediate → I-type Instruction

x = y + 10;

Add $10, $11, $12 → R-type ✓

Addi $10, $11, 10 → I-type ✓

x = y - 10;

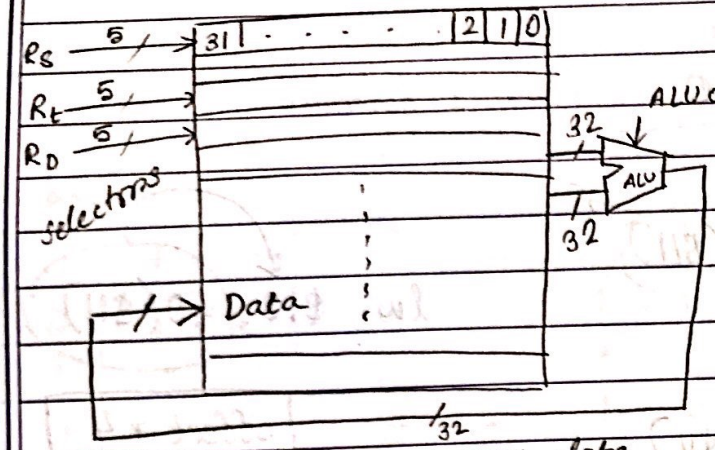
✗ Subi $10, $11, 10 (doesn't exist; No Subtract immediate Instruction; Just use a negative constant)

Please Note:

Addi $10, $11, -10 ✓✓

Registers are processor components that hold data and address.

Fig: Register File

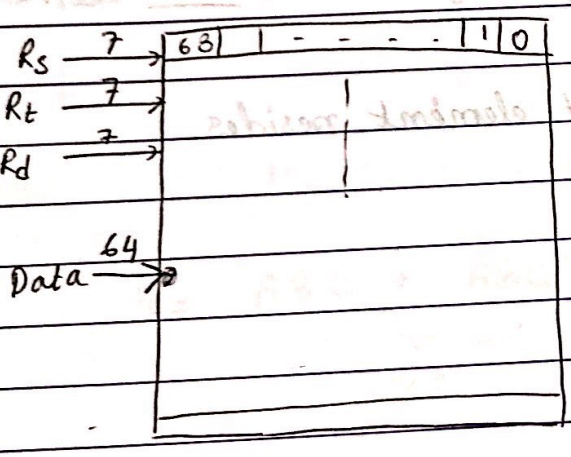


Add $10, $11, $12

D Rs Rt

Data-line to carry data

- Register's size one/ae operation kroy

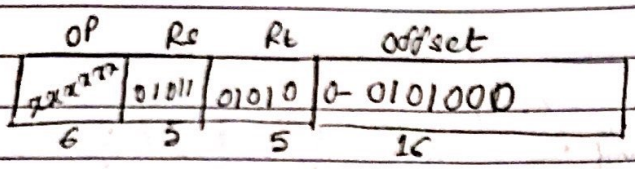


(no. of register changed, bit-length changed)

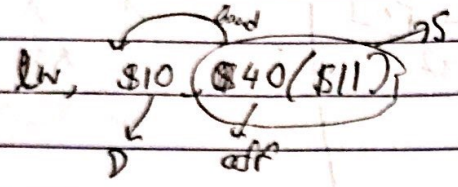
$2^5 = 32$

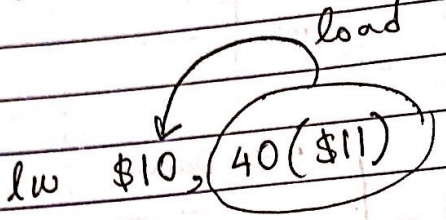
$2^7 = 128$

* • Better to encode numbers rather than names.

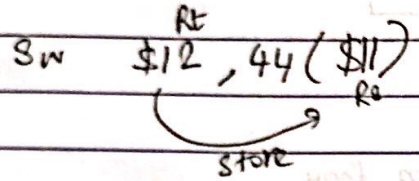


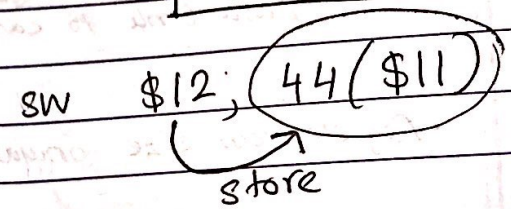
I-format





offset * 4





Address where the first element of array stays is the base element.

Base Address

Address where the array's first element resides.

Practice Qus Quiz-01

0-F
1-T

$y = \overline{A.B.C}\,(A.B + A.C)$

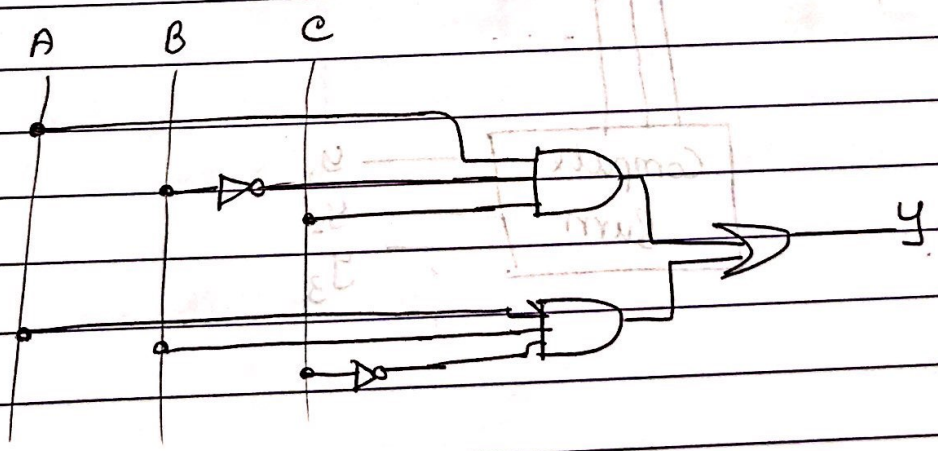
A	B	C	A.B.C	$\overline{A.B.C}$	A.B	A.C	A.B + A.C	y
0	0	0	0	1	0	0	0	0
0	0	1	0	1	0	0	0	0
0	1	0	0	1	0	0	0	0
0	1	1	0	1	0	0	0	0
1	0	0	0	1	0	0	0	0
1	0	1	0	1	0	1	1	1
1	1	0	0	1	1	0	1	1
1	1	1	1	0	1	1	1	0

$y = AB'C + ABC'$

SOP

OR (choice) T F

AND (no choice) T F



Programmable Logic Array – PLA

PLA → You need SOP expressions only.
The only limitation is the number of AND gates that are available.

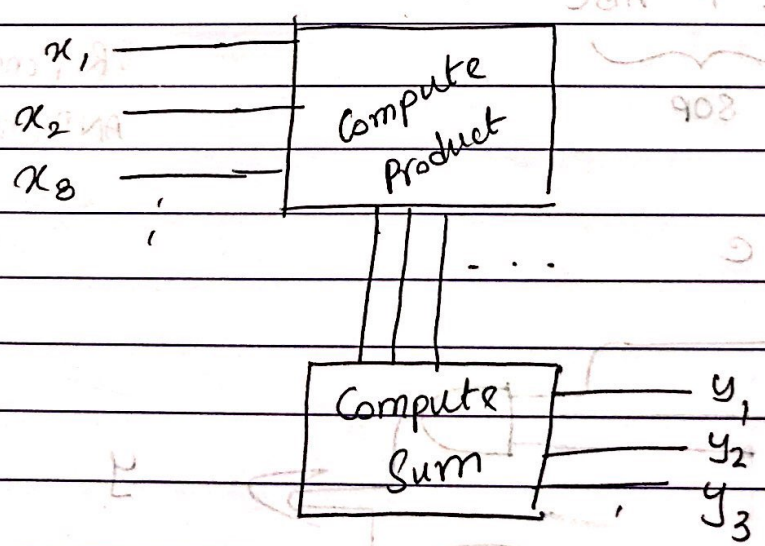
It is used to build reconfigurable digital circuit.
It is a combination of memory and logic.

PLA has programmable AND Gate array and programmable OR Gate array.

adv: flexibility
disadv: not easy to use

(All the minterms are not realized, but only reqd minterms are implemented)

PLA is used as a decoder, counter, used to provide control over datapath.

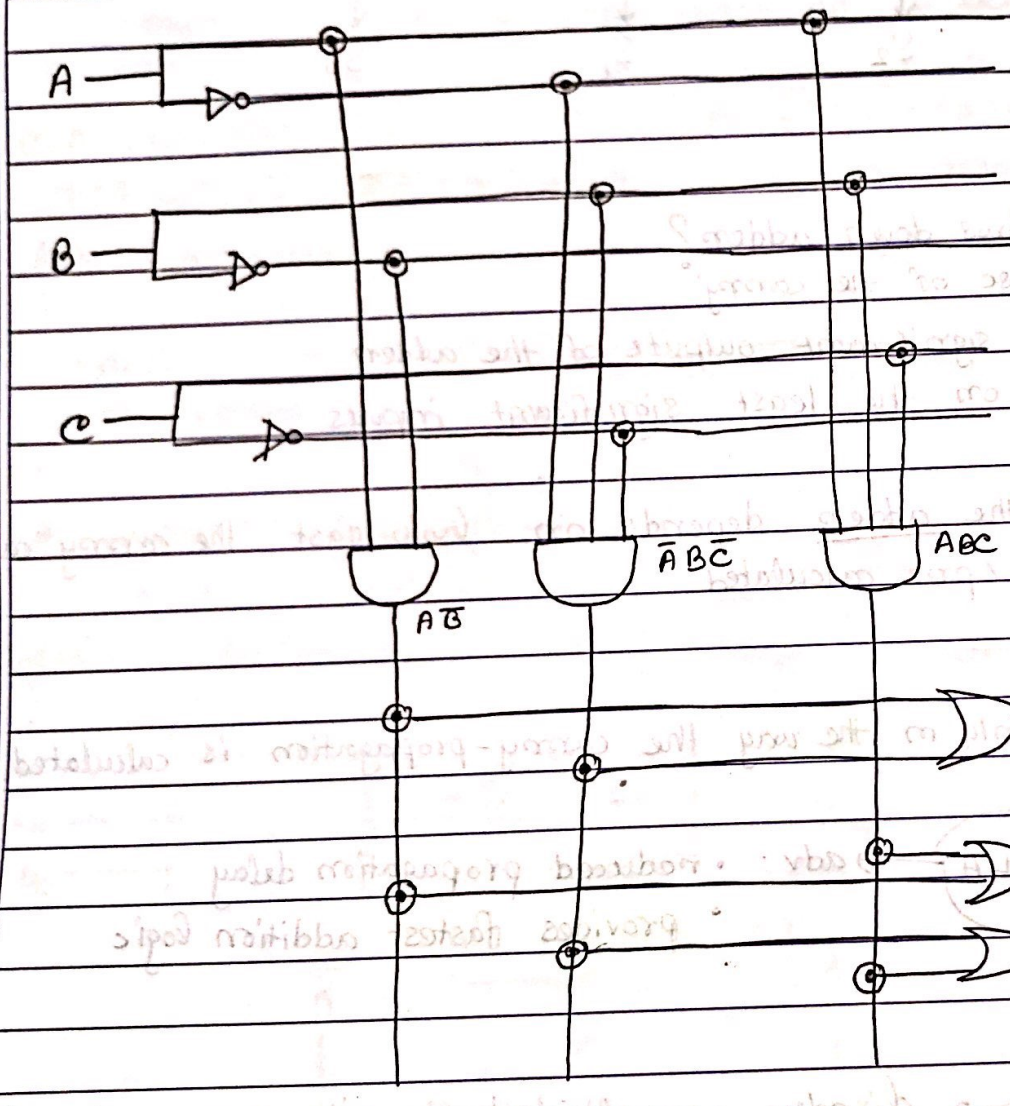


Q. Design a PLA for the expression below.

$y_1 = A\bar{B} + \bar{A}B\bar{C}$

$y_2 = A\bar{B} + ABC$

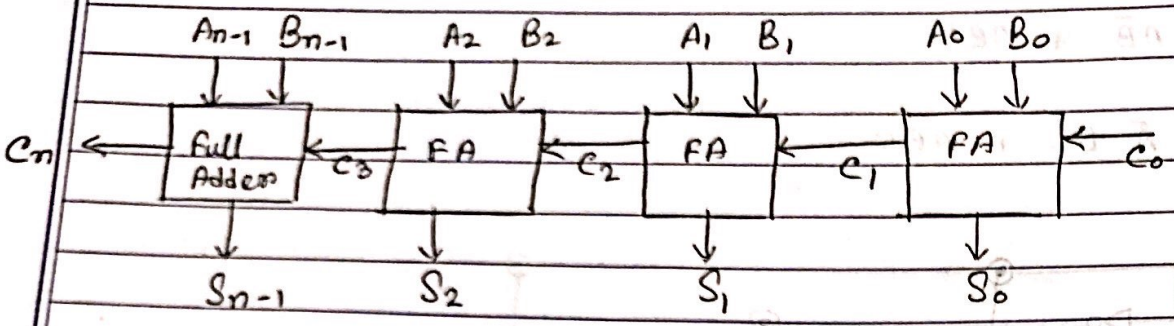
$y_3 = \bar{A}B\bar{C} + ABC$



limitation: no. of AND gates available

Q. Design a ripple adder and explain its limitations. Design another adder that overcomes the limitations of a ripple adder.

Ripple Carry Adder (RCA)



Q. What slows down adder?

A: The 'curse of the carry'.

The most significant outputs of the adder depends on the least significant inputs.

The speed of the adder depends on how fast the carry can be propagated / pre calculated.

Adders differ only in the way the carry-propagation is calculated.

CLA → adv:
- reduced propagation delay
- provides fastest addition logic

disadv:
- complicated circuit as no. of variables increase
- costliers circuit as more numbers of hardware involved.

Fast (Parallel) Adder

Conditional Sum Adder / CLA / Fast Adder

A carry look-ahead adder (CLA) or Fast Adder improves speed by reducing the amount of time reqd to determine carry bits.

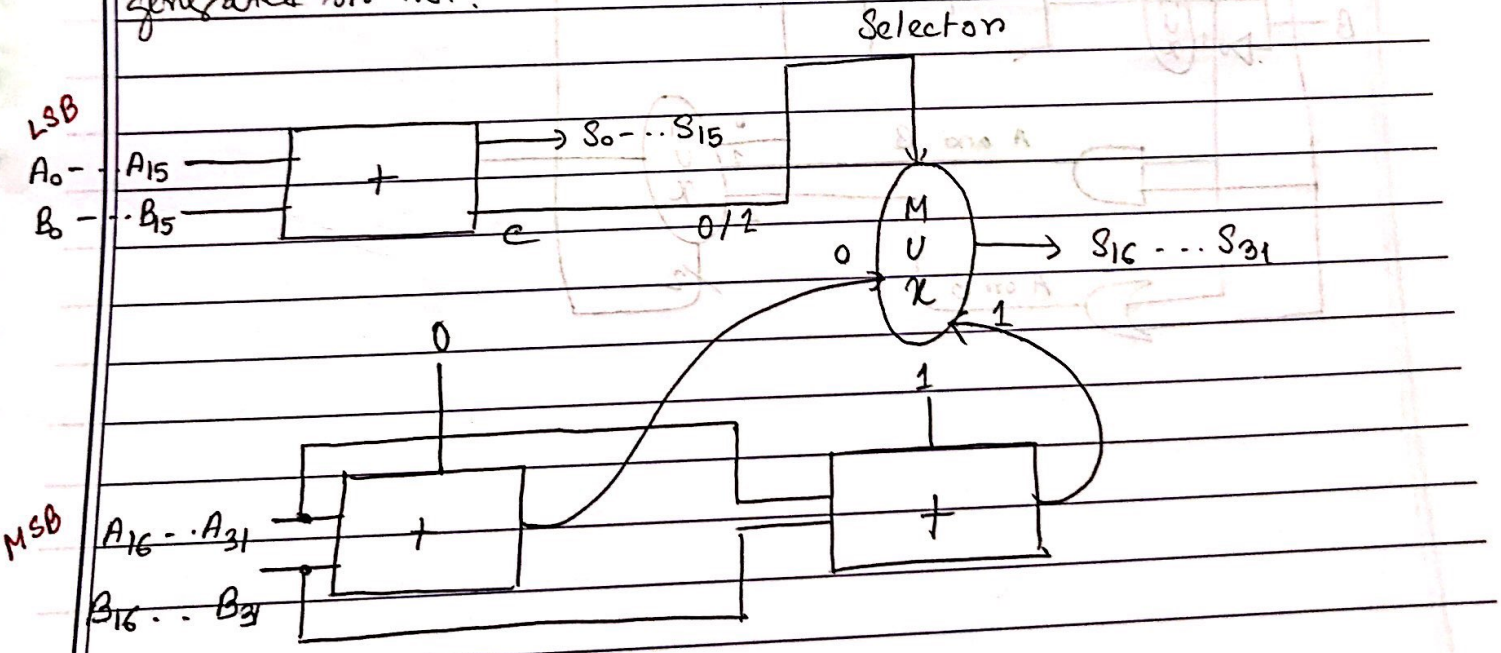
RCA is slower comparatively.

↳ Here, carry bit cal^c-ed alongside sum bit, and each stage must wait until the prev. carry bit has been cal^{c}ed.

CLA calculates one or more carry bits before the sum, which reduces the wait time to cal^c the result of the larger value bits of the adder.

- reduces propagation delay by more complex hardware, so costlier.

This method makes use of logic gates so as to look at the lower order bits of augend & addend to see whether higher order carry is to be generated or not.



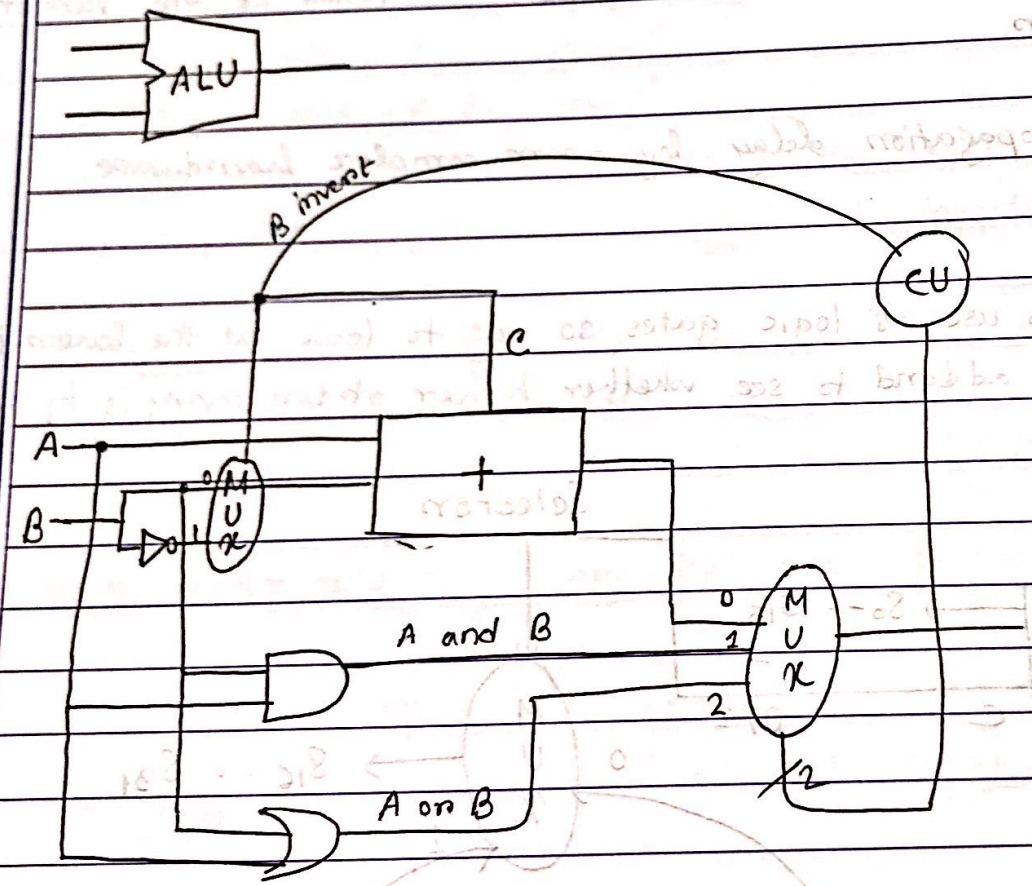
Q. Design an ALU with logical gates

ALU – Arithmetic Logic Unit

ALU is a part of CPU that carries out arithmetic and logic operations on the operands in computer.

* ALU is the building block of CPU.

$A - B = A + \overline{B} + 1$ (2's complement)



00 : Add/Sub
01 : A and B
10 : A or B

MUX is a combinational circuit that is mostly used to select path bet^n multiple sources and a single destination.

Sequential Circuit consists of a feed back path and employs some memory elements.

S.C. = C.C + Memory elements

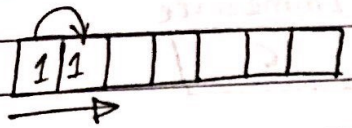
Flip-flops: Bistable logic devices
(2 stable states)

Sequential Circuit
- Synchronous (output change only at specific time i.e. with clock input)
- Asynchronous (outputs change at any time i.e. without clock input)

* Flip-flops are memory elements that change (state) on clock signals.

Memory element: a device which can remember value indefinitely, or change value on command from its inputs.

Srl (Shift right logical)
Sra (Shift right arithmetic)



sign of original sign not reserved by Srl.

Q. $11, $10, $11, 3

- identify type and then encode it.

sll $10, $11, 3

R-type op.code: 0

↳ because shamt

8 4 2 1
1 0 1 1

1011

op	RS	Rt	Rd	Shamt	Func
00000	01011	xxxxx	01010	00011	xxxxxx
6	5	5	5	5	6

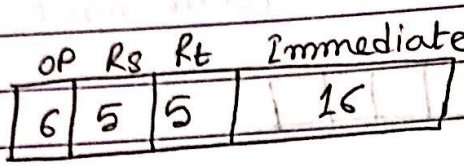
don't put anything here, because not a valid format $11, $10, $11, $3

shamt
↳ shiftamnt = 3
= 00011

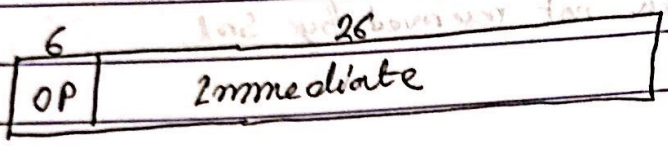
* beq, bne — I-type register

beq $10, $11, lb1

bne $10, $11, lb2



J lb1 (unconditional)



PC → if(i==j){

PC ← PC+4 → x = y+z;

PC ← PC+4 z = x-y;

y = x+z;

:

:

{

} else {

{

Target address

PC+4

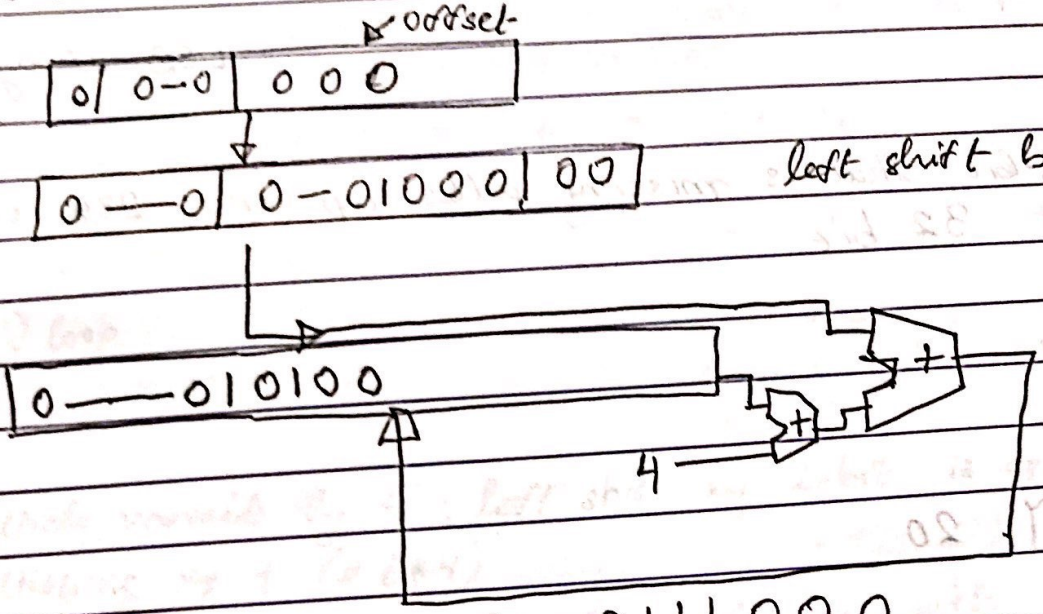
+

Sign extension of offset l.s. by 2

PC : 20

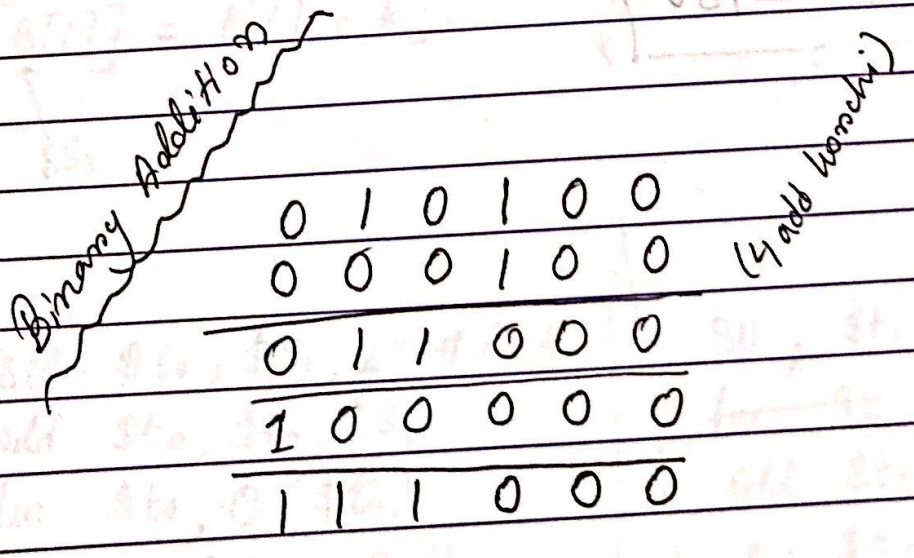
Offset : 8

What will be the memory address of L statement?



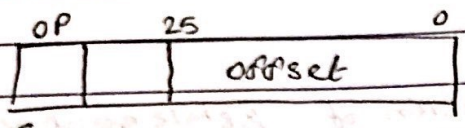
32168

Target address : 56



Lecture

J-type instruction



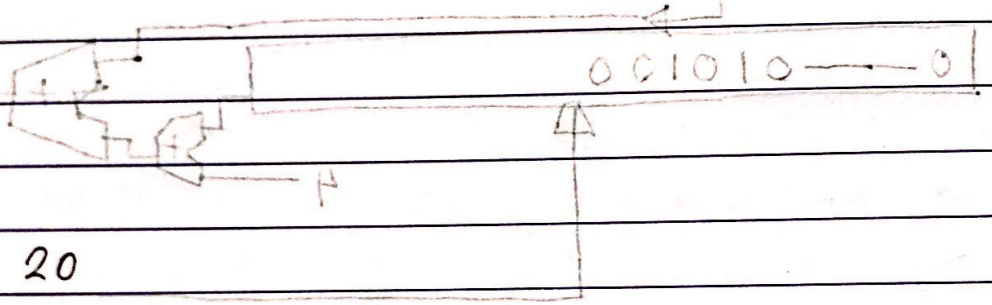
J exit

0 0	0 0
0 1	0 0
1 0	0 0
1 1	0 0

LSB always zero

msb 4-bit that is missing adds up with 28 and makes it 32 bit

Concating



J 20

PC : 1100 - - - - 100

$20

0-010100

1100 - - - - 100

0 0 1 0 1 0

0 0 1 0 0 0

0 0 0 1 1 0

0 0 0 0 0 1

0 0 0 1 1 1

while (save[5] == k) i += 1;

($Sc, $Ss, $Si)

MIPS code:

we add (loop) so we can branch back to that instruction at the end of the loop.

```
loop:
    lw $to, 20($Sc)   # $to ← save[5]
    bne $to, $Ss, Exit
    Add i $Si, $Si, 1
    J loop
Exit:
```

(multiply variable by 4 ; left-shift by 2-bit is equivalent to multiplying by 4. (# i*4)

To go to exact address, add base address with i+4

× lw $to, 4*i($Ss)

($S0, $S1, $S2)

while (save[i] == k)

sw

A[B[i]] = B[i] + k;

A[$t1]

load করলে যে content তা pabo, content'tai aman memory address.

($S4, $S3)

```
loop:
    sll $to, $S1, 2   # i*4        | sll $t1, $S4, 2
save[i] Add $to, $to, $S0          | ~~lw $t~~
    lw  $to, 0($to)                | Add $t1, $t1, $S3
    bne $to, $S2, Exit             | lw $t1, 0($t1)
    sll $to, $S1, 2                | sll $t1, $t1, 2
B[i] Add $to, $to, $S3             | Add $t1, $t1, $S4
    lw  $to, 0($to)                | lw $t1, 0($t1)
    Add $to, $to, $S2              | sw $to, 0($t1)
                                   | J loop
                                   | Exit:
```

Basic Blocks
(no branch or jumping)

$11 : 4	$11 : 5
$12 : 5	$12 : 5
$10 : 1	$10 : 0

Slt $10, $11, $12 — R

stays one till this value remains lower than this

bne, beq

==, ≠

Slti $10, $11, 100 — I

<, ≤, >, ≥ → expensive to implement in hardware

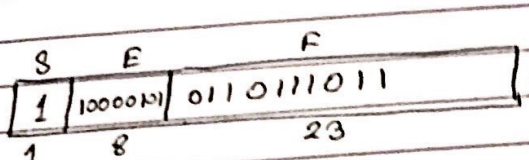
→ intensive operation

We're working on embedded economical architecture.

Adel $10, $11, $12

Addu $10, $11, $12

IEEE 754 Floating Point Representation



$-91.6875 = -1011011.1011$

Step1 Binary representation of 91

$= 1011011$

0.6875 (multiply by 2)

$= 0.1011$ ✓

Step2 Convert into normalized form

$= -1.0110111011 \times 2^6$

$Q = (127)_{bias} + 6 = 133 = 10000101$ ✓

$0.101 = 1.01 \times 2^{-1}$

$0.010 = 1.0 \times 2^{-}$

$2^{E-1} - 1$ (E = 8)

$= 128$

Q. Convert it into HEX value?

Binary to equivalent decimal value

3	C	C	8	0	——	0
0011	1100	1100	1000	0	——	0

$01111001 = 121$

$E = 121 - 127 = -6$

$1 \cdot 1001 \times 2^{-6}$

$= 0.\overset{-1}{0}\overset{-2}{0}\overset{-3}{0}\overset{-4}{0}\overset{-5}{0}\overset{-6}{1}\overset{-7}{1}\overset{-8}{0}\overset{-9}{0}\overset{-10}{1}$

$= 2^{-6} + 2^{-7} + 2^{-10} \approx 0.0244$

(Not for mid)

Lecture

24-2-20

main () {
. . . . . . PC
. . . . . . PC + 4

JAL → Test (): ← PC 0x1234
← PC + 4 0x1238 → $ra

}

→ Test () {
. . . . .
. . . .
. . . .
.
return (): ← PC 0x1444
PC + 4 0x1448

}

Jr $ra (Jump return to address in register ra)

JAL
Jump and Link

$ra/$31 → func caller's khetre return address ta store kore rakha

$ra register will hold that value.

* starting address after a func call, otherwise, infinite loop.

* Stack to handle more than one func

* Recursive/Cascading func call

Procedure

↙ leaf ↘ non-leaf

main fun^c calls a supporting fun^c and this returns a value to main fun^c; no cascading here

A0 A3

↳ to carry/pass argument values.

preserve the value of $s0 for each program; save the prev. state

P_1

P_2

P_3

:

$P_i \Rightarrow \boxed{\$0 = 5}$ → Stack

Stack → memory location in the RAM

$v0 → into return register

P_{i+1} $s0 ?

:

P_n

Non-leaf Procedure

↳ calls a cascading/recursive fun°/procedure

for every step, maintain a stack.

MSB 16-bit | LSB 16-bit

X = 1100...0 | 00...110 — I-type: 2

$t0

cant store it at one go, so we use

lui $t0, 110...0 (MSB 16-bit)

load upper immediate

$t0 : 110...0 | 0...0

MSB 16-bit | LSB 16-bit

Oni $t0, $t0, 00...110 (LSB 16-bit)

↳ bitwise or operation

110...0 | 0...0

00...0 | 00-110

$t0 ← 110...000...110

Computer Abstractions and Technology

Lecture 01

Computer makes applications feasible.
- Computers in automobile, cellphones, search engines, Human genome project, World Wide Web.

* Computers are (pervasive) → super-common.

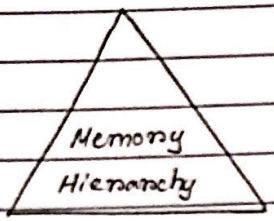
Computers are used in 3 different classes of applications. (Classes of Computers)

(i) Desktop Computers	(ii) Servers	(iii) Embedded Computers
• general purpose	• network-based • greater computing, storage, and I/O capacity. • Supercomputers are a class of computers with the highest performance and cost; they're configured as servers.	• hidden as components of systems (largest class of computers) • microprocessors in car, computers in a TV

An embedded computer is a computer inside another device used for running one predetermined app./collection of software.

- Both software and hardware affect program performance

Multicore microprocessor – A microprocessor containing multiple cores (processors) in a single integrated circuit.



- speed
- cost
- size

↑ (fastest) ↑ speed (slowest)
(expensive) ↑ cost (cheap)
(smallest) ↓ size (largest)

Shape indicates speed, cost and size.

Computers need to be fast and dependable.
Since any physical device can fail, we make systems dependable by including redundant components that can take over when a failure occurs and to help detect failures.

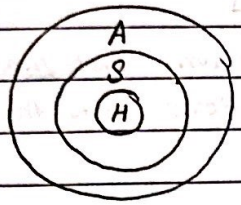
Understanding Performance
(How this affects performance)

- Algorithm determines the no. of operations executed.
- Programming language, Compiler, and architecture determines the no. of machine instructions executed per operation
- Processor and memory system determines how fast instructions are executed.
- I/O system (hardware and OS) determines how fast I/O operations are executed

The performance of a program depends on a combination of the effectiveness of the algorithms used in the program, the software systems used to create and translate the program into machine instructions, and the effectiveness of the computer in executing those instructions, which may include I/O operations.

Below your Program

Layers of Software → A, S, H



1. Application Software
(written in high-level language)

2. System Software → Compiler (translates HLL code to machine code)
→ OS (service code) → An OS is a system software that acts as an interface between a user and a hardware, and provides a variety of services and supervisory functions.

Software that provides services that are useful, including OS, compilers, loaders, and assemblers.

* imp fun's of OS *
- Handling input/output operations
- Allocating storage & memory
- Scheduling tasks, and sharing resources.

examples of OS: Linux, iOS, Windows.

3. Hardware
Processor (CPU), memory (RAM), I/O controllers

→ (translates HLL into instructions)

Q. What is a compiler? HLL → Java, C, C++

Ans: A compiler is a program that translates high-level language code into assembly language statements/code.
(into instructions that the hardware can execute)

Stored-program concept : the idea that data and instructions can be stored in memory as numbers

Binary digit / Bit : (0 or 1) that are the components of information

Instructions: A command that computer hardware understands and obeys.

We use numbers for instructions and data. } foundation of computing

Assembler: a program that translates a symbolic version of instructions into the binary version.

(one line for every instruction)

Assembly language (Symbolic language) – A symbolic representation of machine instructions.

Machine Language (Binary Language) – A binary representation of machine instructions.
↳ that the machine understands

High-level Language program (in C) → Compiler → HLL program compiled into assembly language and then assembled into machine language.

H.L.L are portable across multiple systems.

Assembly Language (for MIPS)

Assembly language is a low-level programming language specific to a particular comp. architecture.

Assembler → Binary Machine Language (for MIPS)

HLL → natural language
→ productivity improved
→ conciseness (fewer lines to express idea)
→ allow programs to be independent of the computer on which they were developed.

Basic functions of the underlying hardware
inputting data, outputting data, processing data, and storing data.

5 computer components

1. Input (keyboard, microphone)
2. Output (display, speaker)
3. Memory
4. Datapath } Processor / CPU
5. Control }

- input writes data to memory
- output reads data from memory
- Processor gets instructions and data from Memory.
- Control sends the signals that determine the operations of the datapath, memory, input, and output.

- datapath is the component of processor that performs arithmetic operations.

Cache memory: small, fast, SRAM memory for immediate access to data.

Cache memory acts as a buffer for a slower, larger memory.

SRAM is faster, expensive but less dense than DRAM
(Static) (dynamic)

- ISA (instruction set architecture) — software/hardware interface contains info to run ML correctly
- ABI (app. binary interface) — ISA + system software interface
- implementation — hardware that obeys the architecture abstraction

response time / execution time: Total time reqd for the computer to complete a task.

throughput / bandwidth : no. of tasks completed per unit time.
(total amount of work done in a given time)

Decreasing response time almost always improves throughput.

$$\text{Performance}_x = \frac{1}{\text{Execution time}_x}$$ (reciprocals

$$\frac{P_x}{P_y} = \frac{E_y}{E_x} = n$$

If X is n times as fast as Y, then the execution time on Y is n times as long as it is on X

improve performance → ↑ increase performance
improve execution time → ↓ decrease execution time

elapsed time
→ system performance

CPU time
→ CPU performance

CPU Time = no. of CPU clock cycles * clock period (clock cycle time)

CPU Time = $\frac{\text{no. of CPU clock cyles}}{\text{clock rate}}$

clock period $\propto \frac{1}{\text{clock rate}}$

clock period - time for a complete clock cycle

clock ~~cyt~~ cycle - time for one clock period (runs at a constant rate)

clock rate — cycles per unit second
→ unit Hz

no. of CPU clock cycles = I × CPI

no of CPU clock cycles = Instructions for a program (I) × Average clock cycles per instruction (CPI)

CPI → determined by CPU hardware

Clock Cycles per Instruction

- Average no. of clock cycles per instruction for a program.

Each computer executes the same number of instructions for the program.

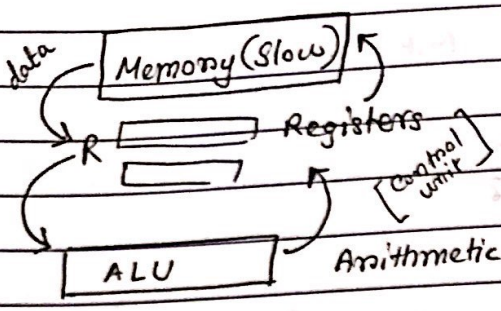
Instruction count: the no. of instructions executed by the program.

(determined by program, ISA and compilers)

CPU time = Instruction Count × CPI × Clock cycle time

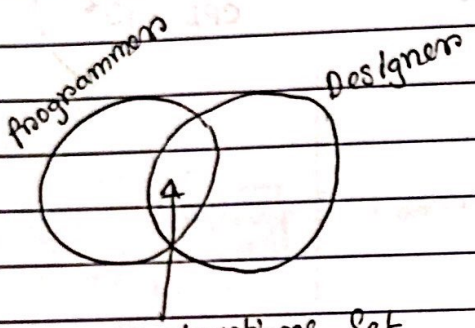
$$\text{CPU time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock rate}}$$

Performance → if Registers no. increased, then fast hobe.
Improved

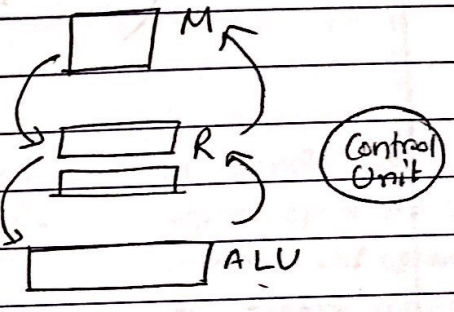


২টা বেশি register, তাই fast computer; cost o beshi

Arithmetic Logic Unit



Instructions Set
OR
Machine Instruction



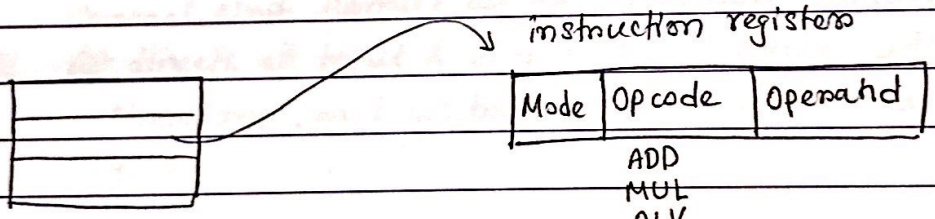


Fig: Instruction Format

ADD
MUL
DIV
INCA
↳ increment accumulator

- we take address from memory, not directly from memory.

2 + 3
↳ opcode
↳ operand taken from memory

MIPS (million instructions per second)

MIPS is an instruction execution rate.

- Faster computers have a higher MIPS rating.

$$\text{MIPS} = \frac{\text{Instruction Count}}{\text{Execution time} \times 10^6}$$

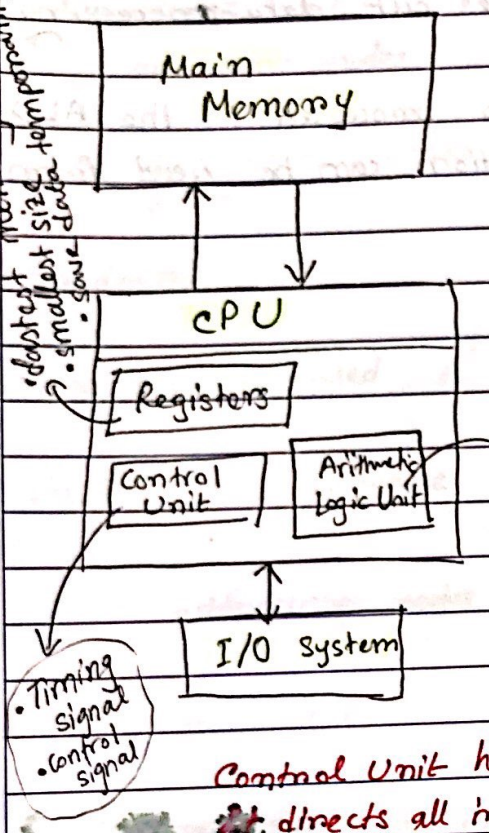
$$\left(= \frac{\text{I. count}}{\frac{\text{I. count} \times \text{CPI}}{\text{clock rate}} \times 10^6} = \frac{\text{Clock Rate}}{\text{CPI} \times 10^6} \right)$$

#

Von Neumann's Architecture (Stored Program Memory Concept)

Memory → we keep → (Data & Instructions) → Program set

we store in main memory



[connect them with (bus), which is implemented by multiplexers.] (registers)

Arithmetic Logic Unit → adder

+ , − , / , AND , OR , shift → perform arithmetic operations

ALU performs logical operations, arithmetical operations, Bit shifting operations

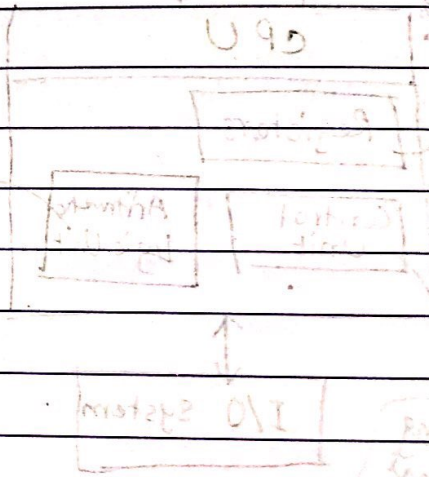
Control Unit handles all the processor control signals. It directs all input & output flow, fetches code for instructions, and controlling how data moves around the system.

Q.) Why Registers, and not MM? ⇒ * speed *

→ (increase throughput)

Parallel Processing : a class of techniques which enables the system to achieve simultaneous data-processing tasks to increase the computational speed of a computer system.

A parallel processing system carries out data-processing to achieve faster execution time.
For e.g. while an instruction is being processed in the ALU component of the CPU, the next instruction can be read from memory.



Scanned with CamScanner

Reduced Instruction Set Computer

Category of Processor (ISA)

Complex Instruction Set Computer

	RISC (register to register) (Single-clock)	CISC (memory to memory) (multi-clock)
1.	RISC processors have a smaller set of instructions with few addressing modes.	CISC processors have a larger set of instructions with many addressing modes.
2.	Fixed instruction format (Uniform)	2. Variable instruction format.
3.	Pipelining	3. No pipelining (Sequential)
4.	More registers used	4. Less registers
5.	Emphasis on software	5. Emphasis on hardware
6.	Simple addressing modes	6. Complex addressing modes
7.	Hardwired control	7. Microcode control
8.	Complex compiler design	8. Simple compiler design
9.	Calculations are faster and precise	9. Calc^s are slow and precise
10.	Execution time is very less	10. Execution time is very high.
11.	No memory unit	11. Has memory unit to implement complex instructions
Uses	image processing, telecommunication (high-end apps)	(low-end apps) home automation, security systems.

Q. Compare betn RISC and CISC architecture

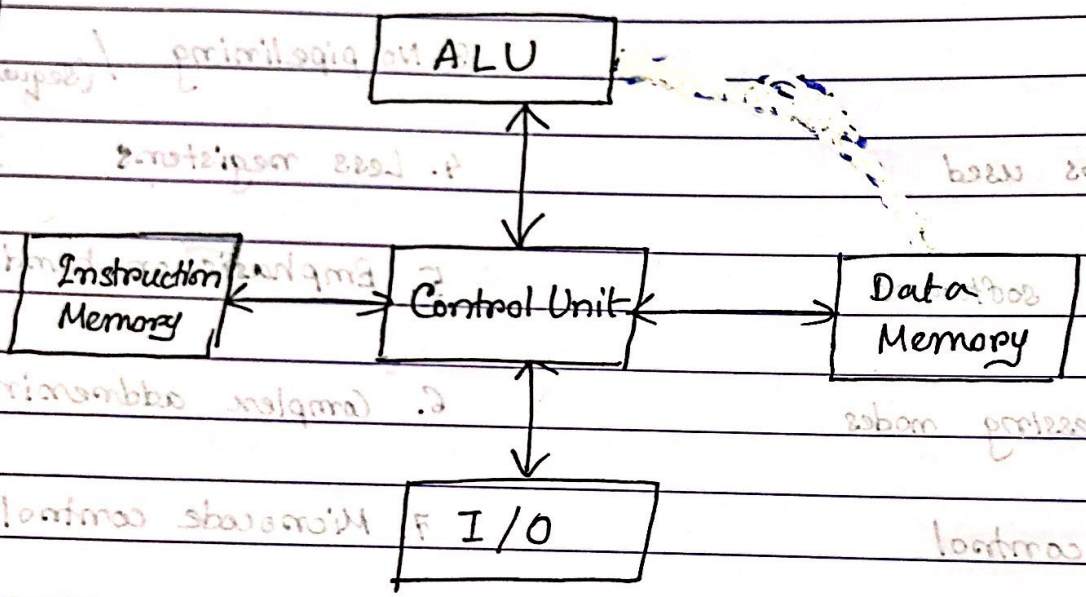
Q. Draw the diagram of Harvard model of computer

Harvard Architecture

A computer architecture with physically separate storage and signal pathways for instructions and data.

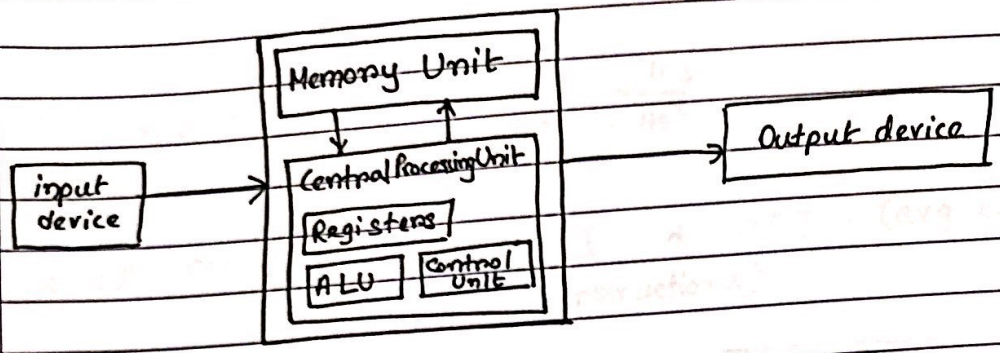
It constrasts with the von Neumann (share a common bus) architecture, where program instructions and data share the same memory and pathways.

Fig: A harvard architecture has a separate bus for signals and storage



Q. Draw the diagram of von Neumann model.

Fig: A von Neumann Architecture.



Q. Compare between von Neumann and Harvard architecture. (3 marks)

	V.N. Architecture	Harvard Architecture
1.	It uses same physical memory address for data and instructions.	It uses separate memory addresses for data and instructions
2.	Processor needs two clock cycles to execute an instruction.	Processor needs one cycle to complete an instruction
3.	Simpler CU (Control Unit) design ; cheaper and faster development of CU	CU for two buses is more complicate costlier development of CU.
4.	Data ~~fetches~~ transfers & instruction fetches cannot be performed simultaneously	Data transfers and instruction fetches can be performed concurrently
5.	Used in Laptops, PCs	Used in microcontrollers and signal processing.

$P \propto \frac{1}{E.t.}$

$c.period \propto \frac{1}{c.rate}$

1. CPU Time = no. of CPU clock cycles * Clock Period (CCT)
   (Et time)
   ↳ Clock Cycle Time

2. $\text{CPU Time} = \frac{\text{no. of CPU clock cycles}}{\text{clock rate (in Hz)}}$

3. no. of CPU clock cycles = I * CPI (avg. c cycles per I)
   (instructions)

4. CPU Time = I * CPI * CCT

5. $\text{CPU Time} = \frac{I \times CPI}{c.rate}$

6. $MIPS = \frac{I}{E.t \times 10^6}$

$= \frac{I}{\frac{I * CPI}{c.rate} \times 10^6}$

$= \frac{c.rate}{CPI \times 10^6}$

Q. A processor P has an 8 GHz clock rate and a CPI of 1.0
If P executes a program within 20s, then find (i) no. of cycles
(ii) no. of instructions

(i) CPU Time = 20s
(Et time)

Clock rate = 8×10^9 Hz

CPI = 1

~~no. of CPU cycles = I~~

$$\text{CPU Time} = \frac{\text{no. of cycles}}{\text{clock rate}}$$

$$\Rightarrow \text{no. of cycles} = 20 \times 8 \times 10^9$$
$$= 1.6 \times 10^{11}$$

(ii) no. of cycles = I × CPI

$\Rightarrow 1.6 \times 10^{11} = I \times 1$

$\Rightarrow I = 1.6 \times 10^{11}$

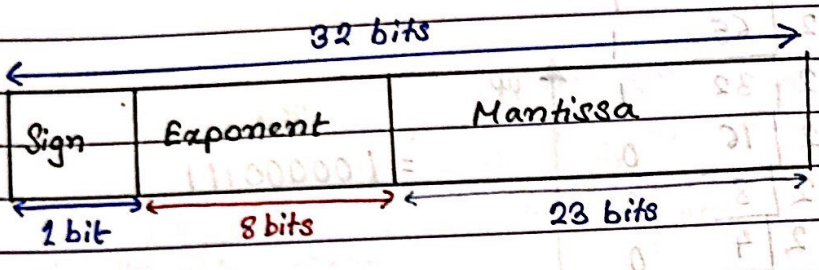
IEEE Standard 754 Floating Point Numbers

(most common representation for real numbers on computers)

- Single precision (32 bits)
- Double precision (64 bits)

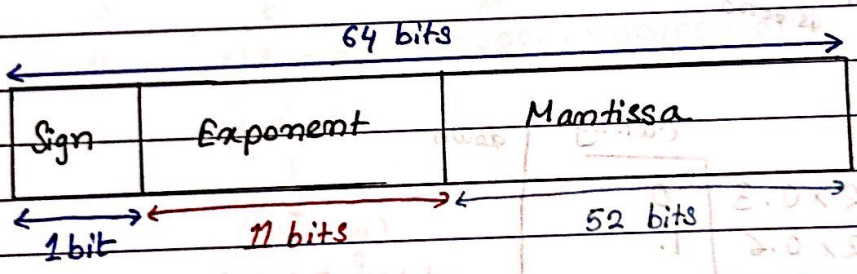
F.P. is a representation for non-integral numbers (including very small and very large numbers)

Single Precision



Bias
127

Double Precision



Bias
1023

Sign of Mantissa	Biased exponent	Normalised Mantissa
0 → +ve number 1 → -ve number	Exponent field represents both +ve and -ve exponents. A bias is added to the actual exponent in order to get the stored exponent	• consists of 1 significant digits

Decimal to IEEE 754 Floating Point Representation

- repeated division by 2 (whole no.)
- repeated multiplication by 2 (for fractions)

Example 1

263.3

(263) ⇒

		rem
2	263	
2	131	1
2	65	1
2	32	1
2	16	0
2	8	0
2	4	0
2	2	0
2	1	0
	0	1

* Rem

up

repeated divide by 2

* until Quotient is zero

263 = 100000111

(0.3) ⇒

	Carry
2 × 0.3	0
2 × 0.6	1
0.2 × 2	0
0.4 × 2	0
0.8 × 2	1
0.6 × 2	1
0.2 × 2	0
0.4 × 2	0
0.8 × 2	1
0.6 × 2	1
	0
	0
	1
	1
	0
	0
	1
	1

Carry down

0.3 = 0100110011 0011001 1 . .

Binary
$\pm 1.xxxxx + 2^{yyyy}$

263.3

Step 1 1.00000111 . 0100110011001100110011 . . .

shift left 8 times, (multiply by 2, ~~so 8×2 = 16~~)

Step 2

$1.00000111010011001100110011 \ldots \times 2^{8}$

(scientific form)

original expt +ve, so add to bias

Step 3 (IEEE format)

mantissa: bits that follow the dp in scientific notation

S	E	M
1bit	8bits	23 bits

Since 263.3 is a +ve no,

sign bit ⇒ 0

exp. bias : 127

S	E	M
0	127 + 8 = 135 (decimal)	00000111010011001100110

(binary) 10000111

-ve no ⇒ subtract to bias

+ve no. ⇒ add to bias

IEEE representation

0100 0011 1000 0011 1010 0110 0110 0110

10100001

Example 2

-91.6875

91 = 1011011

whole

2	91	rem	up
2	45	1	↑
2	22	1	
2	11	0	
2	5	1	
2	2	1	
2	1	0	
	0	1	

0.6875 = 1011

fraction

	Carry Down
0.6875×2	1
0.375×2	0
0.75×2	1
0.5×2	1

-91.6875

bias 127

Step 1 / BINARY

$= 1011011.1011$

STEP 2 / Scientific notation base 2

$= 1.0110111011 \times 2^{6}$

+ve original ~~bias~~ = 6 exponent

so ~~add to bias~~

1	8	23
S	Expt	Mantissa
	6+127	01101110 11
1	=133	

Sign ⇒ 1 since -ve original value

~~01111011~~

10000101

Step 3

Step3 IEEE format

1100 0010 1011 0111 011

C 2 B 7 3

Convert it into hex value

C2B73

Hexadecimal numbers in computing is used for abbreviating lengthy binary representations.

Example 3

Convert −29.632 into IEEE 754 floating point representation.
Show the hexadecimal representation of your conversion.

29 = 11101 ✓
whole

0.632
fraction

		rem-up
2	29	
2	14	1
2	7	0
2	3	1
2	1	1
	0	1

	Carry down
2 × 0.632	1
2 × 0.264	0
2 × 0.528	1
0.056 × 2	0
0.112 × 2	0
0.224 × 2	0
0.448 × 2	0
0.896 × 2	1
0.792 × 2	1
0.584 × 2	1
0.168 × 2	0
0.336 × 2	0
0.672 × 2	0
0.344 × 2	1
0.688 × 2	0
0.376 × 2	1
0.752 × 2	0
0.504 × 2	1
0.008 × 2	1
0.016 × 2	0
0.032 × 2	0
0.064 × 2	0
	0

Step1

11101.10100 00111000101011

Step2

1.11011010 00011100 10101011 $\times 2^{4}$

positive exponent so add to bias 127

S.n w base2

S	E	M
1	(127 + 4) = 131 ⇓ 10000011	11011010 00011100 10101011

Step3

FP Bin

1100 0001 1110 1101 0000 1110 0010 1011

Hex

C 1 E D 0 E 2 B

C1ED0E2B

Example 4

Convert −0.75

0 whole

0.75 fraction

	carry down
0.75 × 2	1
0.5 × 2	1
0 0 ?	

Binary: 00000000

Step 1 : 00000000.11

Step 2: 1.1×2^{-1}

Step 3

S	E	M
1	(−1+127)	10000000000000000000000
	126	
	↓	
	01111110	

1011 1111 0100 0000 0000 0000 0000 0000

IEEE 754 Floating Point to its Decimal

Scientific notation

$1.\underbrace{xxxxx}_{m} \times 2^{e}$

0 1 00 0011 0101 0100 0000 0000 0000 0000

exponent bits

+ve sign bit

So +ve decimal

10000110 (binary)

↓ decimal

134

bias + exponent = 127

127 + expt = 134

⇒ exponent = 7

e = 7

$1.10101000000000000000 \times 2^{7}$

$= 11010100.00000000000000$

my way: 212.0

$m = 1\times2^{-1} + 1\times2^{-3} + 1\times2^{-5}$

$= 0.65625$

$= (-1)^{s} \times (1+m) \times 2^{e}$

$= (-1)^{0} \times (1+0.65625) \times 2^{7}$

$= 1.65625 \times 2^{7}$

$= 212$

M. Sintaha

Example 5

-47.523

Whole: 47

Fraction: 0.523

$= 101111$ ✓

		rem-up
2	47	
2	23	1
2	11	1
2	5	1
2	2	1
2	1	0
	0	1

	Carry-Down
2×0.523	1
0.046×2	0
0.092×2	0
0.184×2	0
0.368×2	0
0.736×2	1
0.472×2	0
0.944×2	1
0.888×2	1
0.776×2	1
0.552×2	1
0.104×2	0
0.208×2	0
0.416×2	0
0.832×2	1
0.664×2	1
0.328×2	0
0.656×2	1
0.312×2	0
0.624×2	1
0.248×2	

8 4 2 1

Step 1

101111.10000101111000111010

Step 2

base 2 S.n.

$1.01111100001011110001101 \times 2^5$

Step 3

S	E	M
1	(5+127) = 132 ⇓ 10000100	01111100001011110001101

1100 0010 0011 1110 0001 0111 1000 1101

C 2 3 E 1 6 8 D

Hexadecimal: C23E168D

Chapter 2 Instructions: Language of the Computer

Every computer must be able to perform arithmetic.

Each MIPS arithmetic instruction performs only one operation and must always have exactly 3 variables.
(Each line contains at most one instruction unlike other programming languages.)

Fast locations for data. In MIPS, data must be in registers to perform arithmetic.

register $zero always equals to 0.
$at register reserved by assembler to handle large constants.

Memory holds arrays, data structures, and spilled registers.
MIPS uses byte addresses, so sequential word addresses differ by 4.

from → [MR. load] → to

MIPS Assembly Language

Category	Instruction	Example	Meaning	Comments
Arithmetic	(R) add	add $s1, $s2, $s3	$s1 = $s2 + $s3	Three register operands
	(R) subtract	sub $s1, $s2, $s3	$s1 = $s2 − $s3	Three register operands
	(I)* add immediate	addi $s1, $s2, 20	$s1 = $s2 + 20	Used to add constants
Data Transfer	(I) load word	lw $s1, 20($s2)	$s1 = Memory [$s2 + 20]	Word from memory to register
	(I) store word	sw $s1, 20($s2)	Memory [$s2 + 20] = $s1	Word from register to memory
	load byte	lb $s1, 20($s2)	$s1 = Memory [$s2 + 20]	Byte from memory to register
Logical	and	and $s1, $s2, $s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1, $s2, $s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	and immediate	andi $s1, $s2, 20	$s1 = $s2 & 20	Bit by Bit AND reg with constant
	or immediate	ori $s1, $s2, 20	$s1 = $s2 | 20	" " OR reg with constant
	(R) shift left logical	sll $s1, $s2, 10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1, $s2, 10	$s1 = $s2 >> 10	
Conditional Branch	branch on equal (I)	beq $s1, $s2, 25	if ($s1 == $s2) go to PC + 4 + 100	Equal test
	branch on not equal (I)	bne $s1, $s2, 25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch; procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 1000	For procedure call

* There is no subi

- Design Principle 1: Simplicity favors regularity

⇒ Hardware for a variable no. of operands is more complicated than hardware for a fixed number.

- Translation from C to MIPS assembly language instructions is performed by the compiler. MIPS code produced by a compiler.

Example 1

C
```
a = b + c;
d = a - e;
```

MIPS
```
add  a, b, c
sub  d, a, e
```
two source operands

one destination operand

Example 2

5 variables: f, g, h, i, j

C
```
f = (g + h) - (i + j)
```

MIPS
```
add  t0, g, h
add  t1, i, j
sub  f, t0, t1
```
→ The compiler creates a temporary variable.

* Only one operation per MIPS instruction. *

The size of a register in the MIPS architecture is 32 bits.

Arithmetic operations occur only on registers in MIPS instruction; so MIPS include data transfer instructions that moves data betn memory and registers

The operands of Arithmetic Instructions are restricted; so they must be from a limited number of special locations built directly in hardware called registers.

↳ bricks of computer construction

- Design Principle 2: Smaller is faster

A very large no. of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.

↳ reason for the limit of 32 registers.

Effective use of registers is critical to program performance.

32 general-purpose registers
- $0, $1, $2, ... $31

$zero → hardwired to 0
$t0 - $t9 → temp
~~$s0 - s7~~
$s0 - s7 → save
$ra → return address

The processor can keep only a small amount of data in registers, but computer memory contains billions of data elements.
Hence, data structures (arrays & structures) are kept in memory.

MIPS Memory

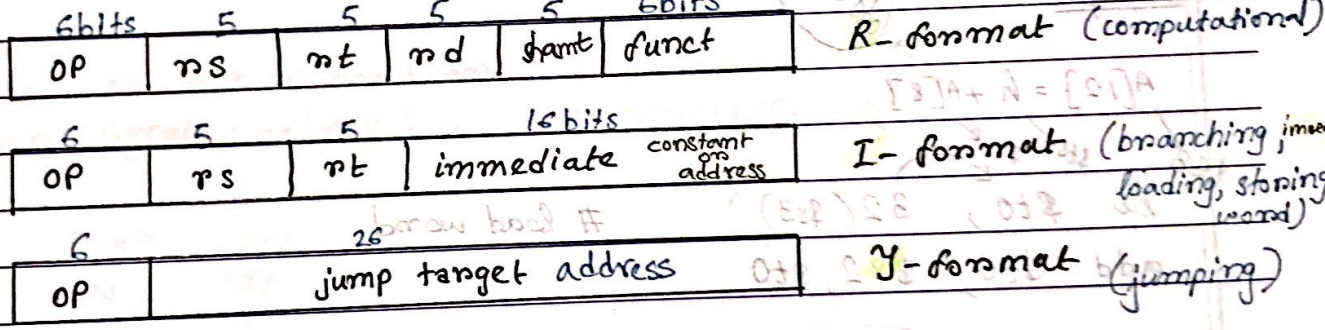
MIPS uses byte-addressing, with each word representing four-bytes.

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

- registers hold 32 bits of data

For MIPS, a word is 32-bits or 4-bytes.
every instruction increments by 4-bytes.

3 instruction formats (all 32-bits wides)



op - opcode → The field that denotes the operation and format of an instruction

Load word and Store word are instructions that copy words bet^n memory and registers in the MIPS architecture.

load word (lw): data transfer instruction that copies data from memory to a register.

store word (sw): copies data from a register to a memory

Example

$g = A[5] + i$

Array [something]

lw $t0, something * 4 ($s3)

lw $t0, 20 ($s3)

5×4

Example

offset * 4

A[12] = h + A[8]

$s5 $t0 $s3

lw $t0, 32($s3) # load word

add $t0, $s2, $t0

sw $t0, 48($s3) # store word

Data accesses are faster if data is in registers instead of memory.

Memory is slower than registers.

Registers take less time to access and have higher throughput than memory, making data in registers both faster to access and simpler to use i.e. data is more useful when in a register.

Accessing registers also uses less energy than accessing memory.

Constant / Immediate Operands

We've to load a constant from memory to use one.

An alternative to this is the arithmetic instruction called add immediate
add i

Constant operands occur frequently, so carrying out addi is much faster and uses less energy than if constants were loaded from memory. (Design Principle 3: common case fast)

* # MIPS support negative constants, there is no need for subi in MIPS (subtract immediate)

Numbers are kept in hardware as a series of high and low electronic signals; → base 2 numbers / binary number.

High	Low
ON	Off
1	0
True	False

Two's Complement Representation

→ Convention to represent Signed Binary Numbers

Step 1 : invert (0 → 1 and 1 → 0)

Step 2 : increment by 1

MIPS Register File

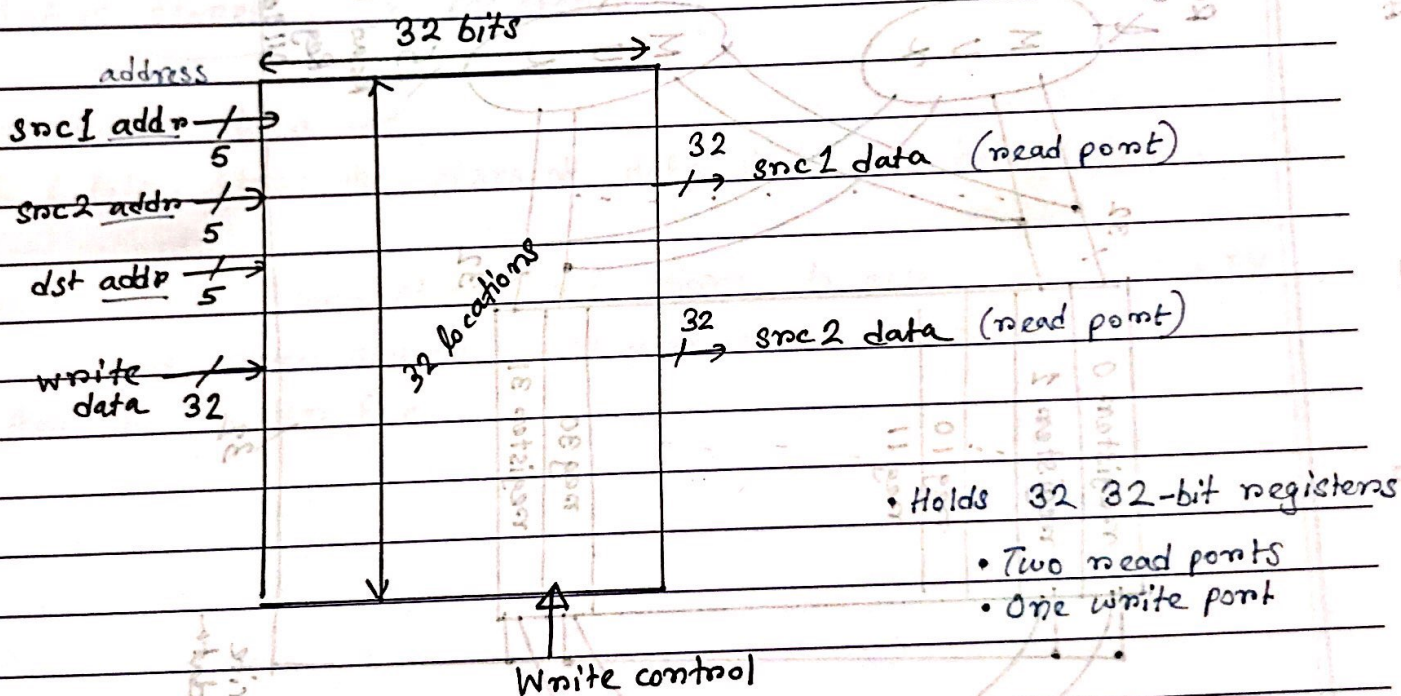
Q. Describe the internal structure of Register File with diagram. (7 marks)

MIPS register-file includes 32 32-bit general purpose registers.

* A register-file is an array of processor registers in a CPU.
A register-file is a small set of high-speed storage cells inside CPU.

MIPS is a load-store architecture, which means that only load and store instructions can access memory.
All other instructions (add, sub, mul, and, or, etc.) must get their operands from registers and store their results in a register.

Fig: Register file



- Holds 32 32-bit registers
- Two read ports
- One write port

Fig: Internal Structure of Register File

Add $10, $11, $12
dst src1 src2

16	8	4	2	1	
0	1	0	1	0	= 10
0	1	0	1	1	= 11
0	1	1	0	0	= 12

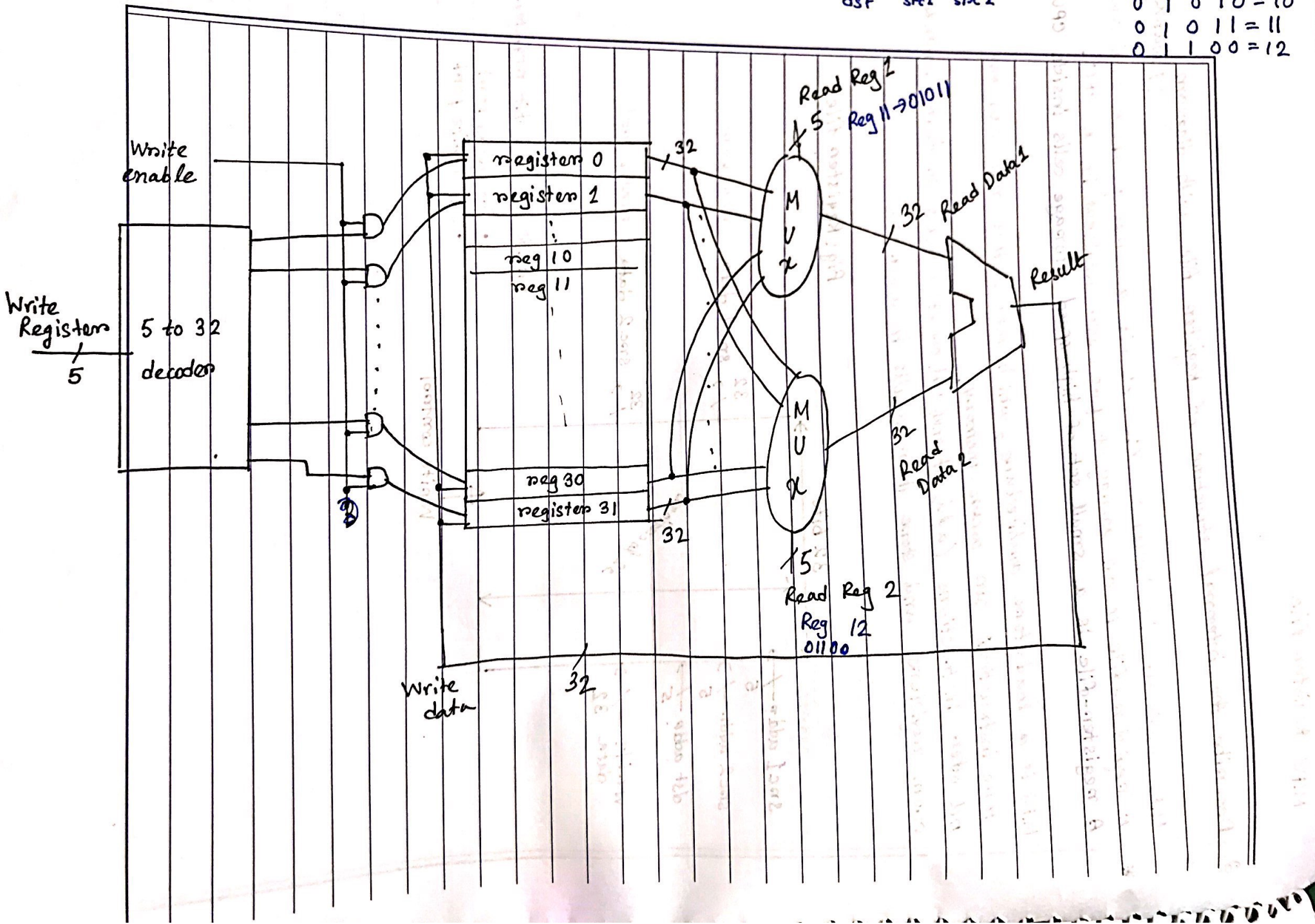
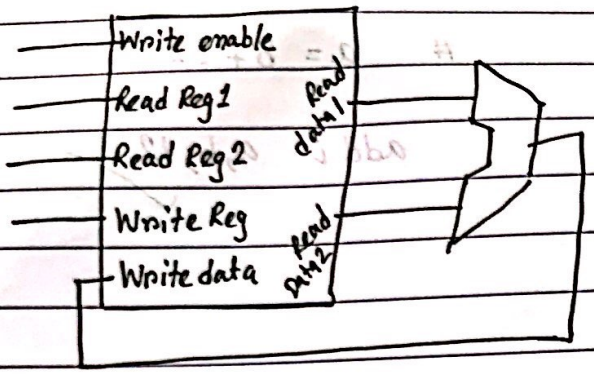


Fig: Block Diagram



Q. What is stored in the Register File?

Ans: The current data ✓✓

The R. File store the pieces of data that are being used by the instructions.

We must use load/store instructions to access the memory, and it is much slower and uses much more energy than the Register File.

slr → div
sll → mul

MIPS Instructions

MIPS format

op dest, src1, src2

$t0

\# x = y + z

$s0 $s1 $s2

add $t0, $s0, $s1, $s2

\# a = b + 12

add i a, b, 12

\# x = y − z

sub $s0, $s1, $s2

$s0 $s1

\# x[11] = x[10] + y

offset * 4

lw $t0, 40($s0)

add $t1, $t0, $s1

sw $t1, 44($s0)

load

load word temp, 4*offset (array variable register)

store word temp, 4*offset (array v. reg)

store

$s0 $s1 $s6 $s7

nested # f = g − A[B[4]]

*array thaakle load into a temp register.

lw $t0, 16($s7)

load shift add * sll $t0, $t0, 2 (power)

$2^{②}$

nest:
lw t, offset×4 (B$s7)
sll t, t, power
add t, t, baireer ghor
load contents t, 0(t)

* add $t0, $t0, $s6

lw $t0, 0($t0) // load contents for A(B(4))

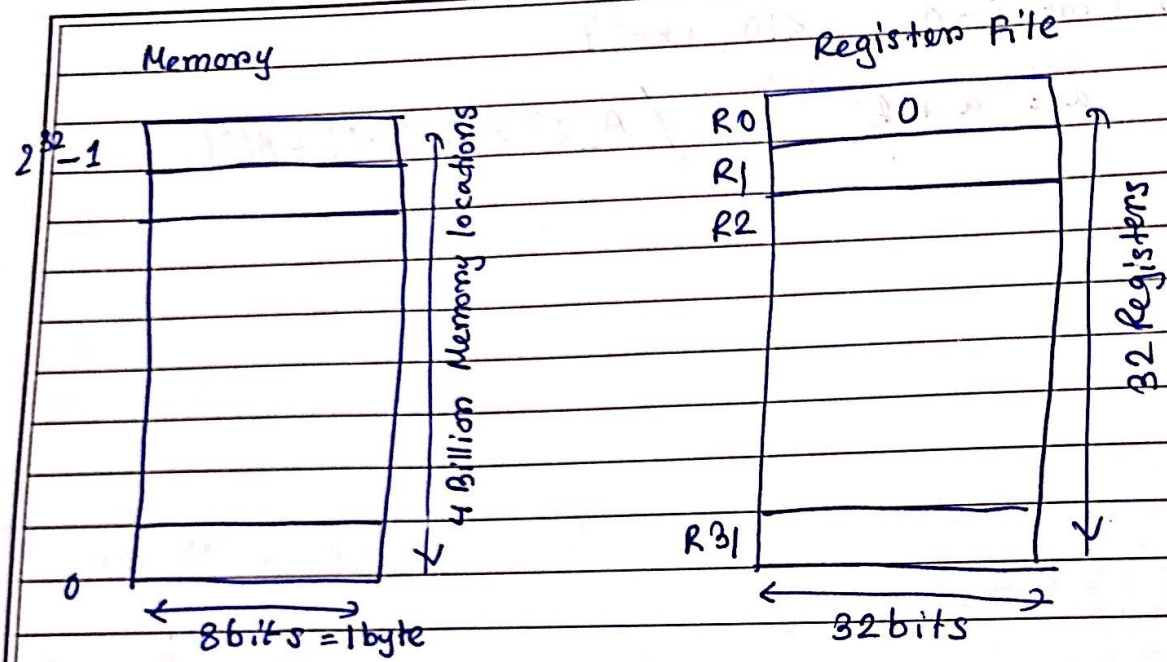
sub $s0, $s1, $t0

Scanned with CamScanner

$t0

```
for (int i=0 ; i<10 ; i++){
    a = a + b;        // A[i] = A[i-1] + B[i];
}
```

$s0 → a
$s1 → b



Slow
Huge

Fast
Small

each register is 1 word.

- A word is the size of a register (32 bits or 4 bytes)
- Registers hold 32 bits of data (1 word)

* Last 2 bits of a word address is 00.

Shifting left by i bits gives the same result as multiplying by 2^i

If statement with a goto (conditional branches)

~~go to statement~~
~~bne r1, r~~

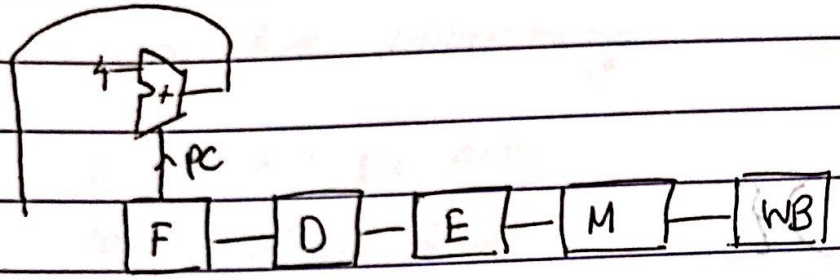
beq r1, r2, L1
(branch if equal)

(branch if not equal)
bne r1, r2, L1

Lecture

Branch Addressing

PC + 4 + (sign ext.n and l. shift by 2) offset



false

Jump Addressing

Target Addressing

Instruction in Int. Instruction Memory.

↳ RAM's theatre

bne $t0, $s5, Exit

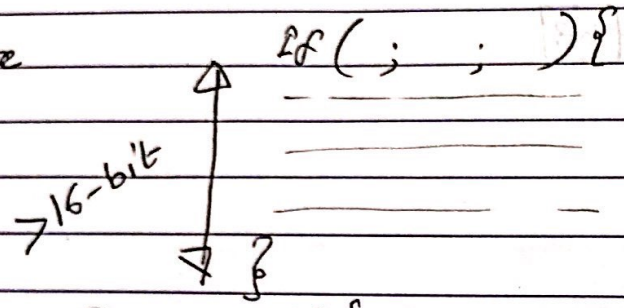
↳ underlying arithmetic op: Subtraction

Branching far away

bne $10, $11, ELSE

Q. How many bits of data this offset can take?

Ans: 16-bit offset



How does assembler handle this?

→ Assembler inverses the instruction

(A) # (5) Addressing Modes in MIPS

Give diagram w appropriate example

1. Immediate Addressing

Addi $10, $11, 20

2. Register Addressing

Add $10, $11, $12

3. Base Addressing

lw $10, 40($11)

sw $10, 44($11)

4. PC-relative addressing

bne $10, $11, ELSE

beq $10, $11, ELSE

5. Pseudo-direct Addressing

J exit

ARM & MIPS

→ add. modes → 32 registers

→ 15 registers

Slide Lecture (3)

$x = 3.538$

$y = 5.319$

Find $x+y = ?$

Using IEEE FP Repⁿ

x

$3 = 0011$

$0.538 =$

0.538×2	1
0.076×2	0
0.152×2	0
0.304×2	0
0.608×2	1
0.216×2	0
0.432×2	0
0.864×2	1
0.728×2	1
0.456×2	0

$x = 0011.1000100110$

$\boxed{x = 1.1100011 \times 2^1}$

y

$5 = 0101$

0.319

0.319×2	0
0.638×2	1
	0
	1
	0

$\boxed{y = 1.0101010 \times 2^2}$

* make both expt equal

$x = 1.1110001 \times 2^{1} = 0.1111000 1 \times 2^{2}$

$y = 1.0101010 \times 2^{2}$

$x + y = (0.1111000 1 + 1.0101010) \times 2^{2}$

$= 10.0011011 \times 2^{2}$

$= 1000.11011 \times 2^{0}$

$= 8 + 2^{-1} + 2^{-2} + 2^{-4} + 2^{-5}$

$\simeq 8.84375$

$$\begin{array}{r} 1.0101010 \\ 0.1111000 \\ \hline 10.0011011 \end{array}$$

$x = 3\ 5\ A\ 2\ 0 \text{---} 0$

$y = 5\ 8\ BA\ 0 \text{---} 0$

$x + y = ?$

$x = 00110101\ 0001\ 0010\ 0 \text{---} 0$

$x = 0011\ 0101\ 1010\ 0010\ 0 \text{---} 0$

$a = 01101011 = 107$

$E = 107 - 127 = -20$

$x = 1.010001 \times 2^{-20}$

$y = 0101\ 0011\ 1011\ 1010\ 0 \text{---} 0$

$a = 10100111 = 167$

$E = 167 - 127 = 40$

$y = 1.011101 \times 2^{40}$

$x = 1.010001 \times 2^{-20}$

$= 0.\ 0 \text{---} 0\ 1010001 \times 2^{40}$

59

$$x + y = (1.0111010\overbrace{\quad\quad\quad}^{53}01010001) \times 2^{40}$$

$$= 10111010\overbrace{0\cdots0}^{34} \cdot \overbrace{0\cdots0}^{19}1010001$$

$$= 2^{40} + 2^{38} + 2^{37} + 2^{36} + 2^{34} + 2^{-20} + 2^{-22} + 2^{-26}$$

$$\approx 1.597 \times 10^{12}$$

FP Adder Hardware * (imp)

- shifting opt. done on lower exponent.

$x = 4.517$
$y = 5.547$

$x * y = ?$

Step 1: Find out the normalized forms of x and y

x

$4.517 = 100.100001$
$= 1.00100001 \times 2^2$

y

$5.547 = 101.100011$
$= 1.01100011 \times 2^2$

(both the exponents are same)

* In multiplication, don't worry about Exponent because we add them.
* Take care of exponents in Add^n, Sub^n.

$x * y = ?$

```
          1.00100001
          1.01100011
          ----------
          100100001
         100100001
        000000000
       000000000
      000000000
     100100001
    100100001
   000000000
  100100001
  ---------------------
  1100100001100011 × 2^4
```

$= 11001.00001100001 1 \times 2^0$

$16 + 8 + 1 + 2^{-5} + 2^{-6} + 2^{-11} + 2^{-12}$

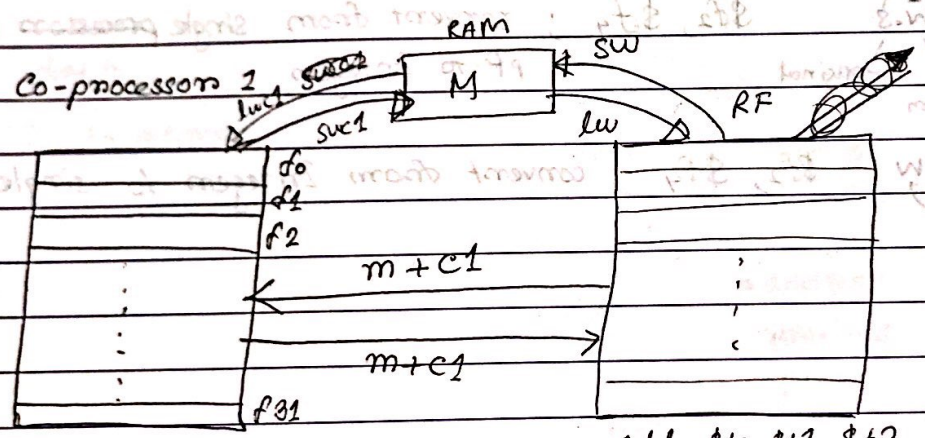
$\simeq 25.047$

\# $x = 5A2C0-0$

$y = 3B540-0$

$x * y = ?$

* Register-file is designed to hold integers values



Add.S $f_0, $f_1, $f_2
(Single point operation)

Add.d $f_0, $f_2, $f_4

Add $t0, $t1, $t2

Add.S $f_0, $t_0, $f_2 ✗ (Wrong)

floating point & integer ekatutha hobena, you must convert

$f_2 = 5.04$
↓
5 (integer)

$t0 = 5$
↳ 5.00

* Always convert Integer to F. Point.

Memory Transfer Inst"

l.s $f0, 100($t2); load word into $f0 from $t2 + 100
s.s $f0, 100($t2); store word from $f0 into $t2 + 100

@Data movement bet" Registers

mov.s $f0, $f2 ; move bet" F.P registers
mfc1 $t1, $f2 ; move from FP register (no conversion)
mtc1 $t1, $f2 ; move to FP register (no conversion)

Data Conversion

cvt.w.s $f2, $f4 ; convert from single precision FP to Integer
(original / after conversion)
cvt.s.w $f2, $f4 ; convert from Integer to single precision FP

wf integer

$t0 $s0 $t2

```
(float) x = (int) y + (float) z;
(int) y = (float) x + (float) z;
(int) z = (float) x + (float) y;
```

Datapath → functionalities & connectivity
(all those instructions executed over the datapath)
(R,I,J)

- fetch
- decode
- execute

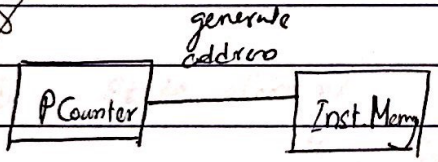
} 3 stages in the lifecycle of an instruction

data → Data memory

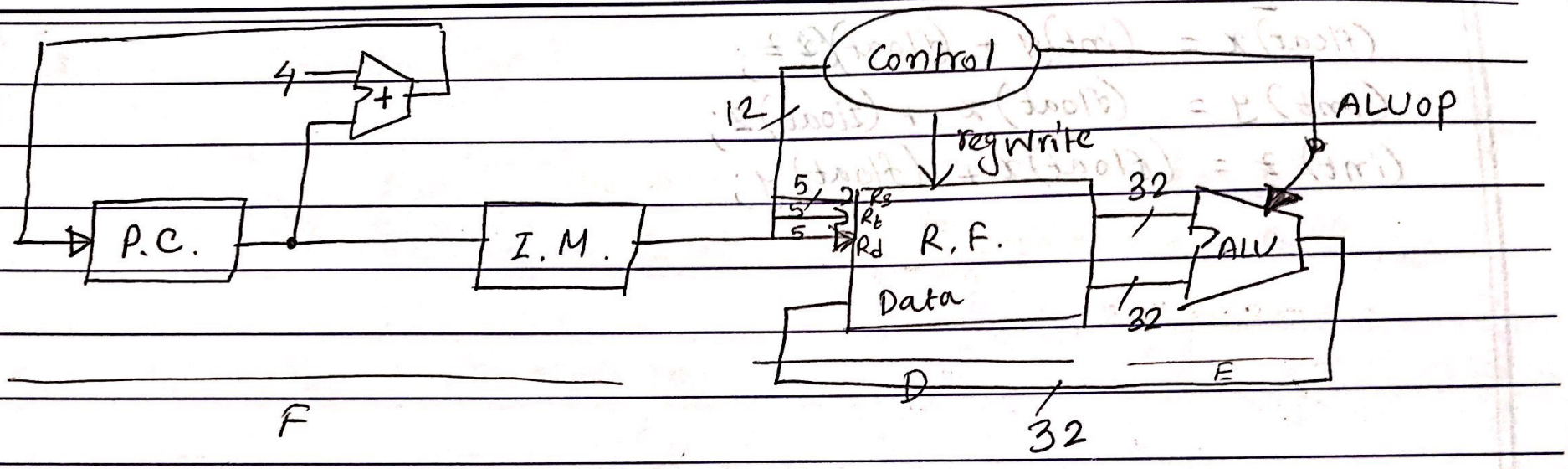
~~instruc~~ instruction → Instruction memory

- address provided by program counter inside the memory

Fig: Datapath



PCounter is the first in Datapath



Add $10, $11, $12

AND $10, $11, $12

Note: except shifting operation

Lecture

11/03/2020

Chapter 4
The Process

Datapath is inside the processor

Program Counter need to fetch instruction from Instruction Memory.
↳ (provides the address of IM)

* ALU is not needed for J-type instructions.

• Can't just join wires together. Use multiplexers

~~Add $5, $~~

R → Add $5, $6, $7

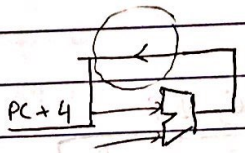
Addi $5, $6, 20

lw $5, 40($6)

R → sub $5, $6, $7

• two inputs, one of them selected based on operation

R → AND $5, $6, $7



(Slide)

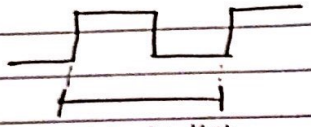
For these 3 R-type operations, they do perform diff. operations.

State element
↳ (Flipflop, registers)
যেগুলো পরে, state preserve kore ratihte pare unless asked to change.

(ALU is combinational so cannot hold data)

- we set clock cycle by the load req^d.

Single cycle Datapath



all instructions within this range

- longest delay determines clock period

*[load takes the longest instruction for finishing execution

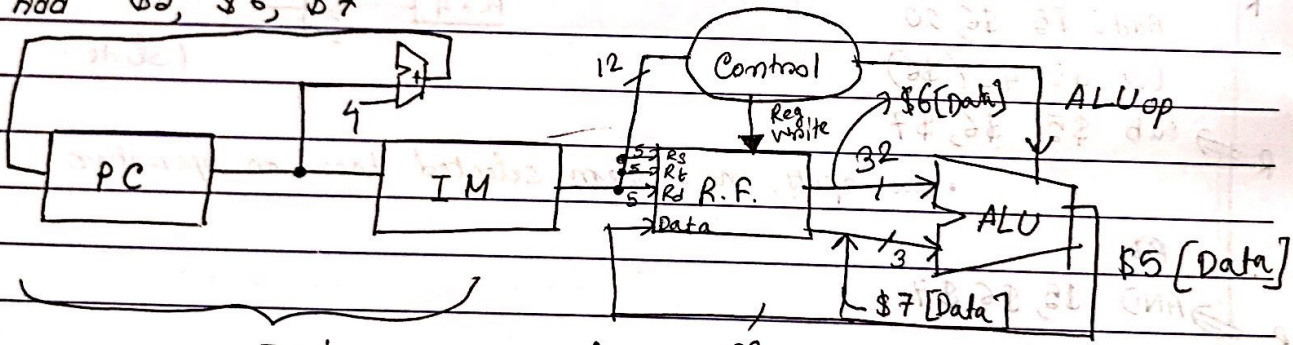
Building a datapath

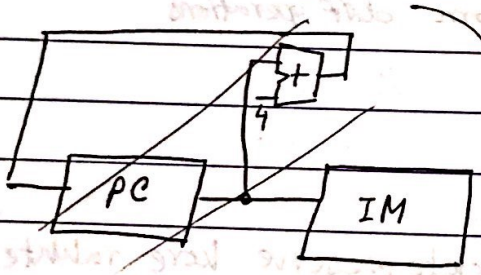
Datapath: Elements that process data and addresses in the CPU

First component needed for instruction fetch: Program Counter

- during decode, we understand the type & what the instruction is.

Add $5, $6, $7





Control identifies the instruction and its type (6 bit opcode, 6 bit funct)

PC generates address of the instruction (32 bit of binary combination)

Single cycle datapath

Read op ⇒ no need of signal

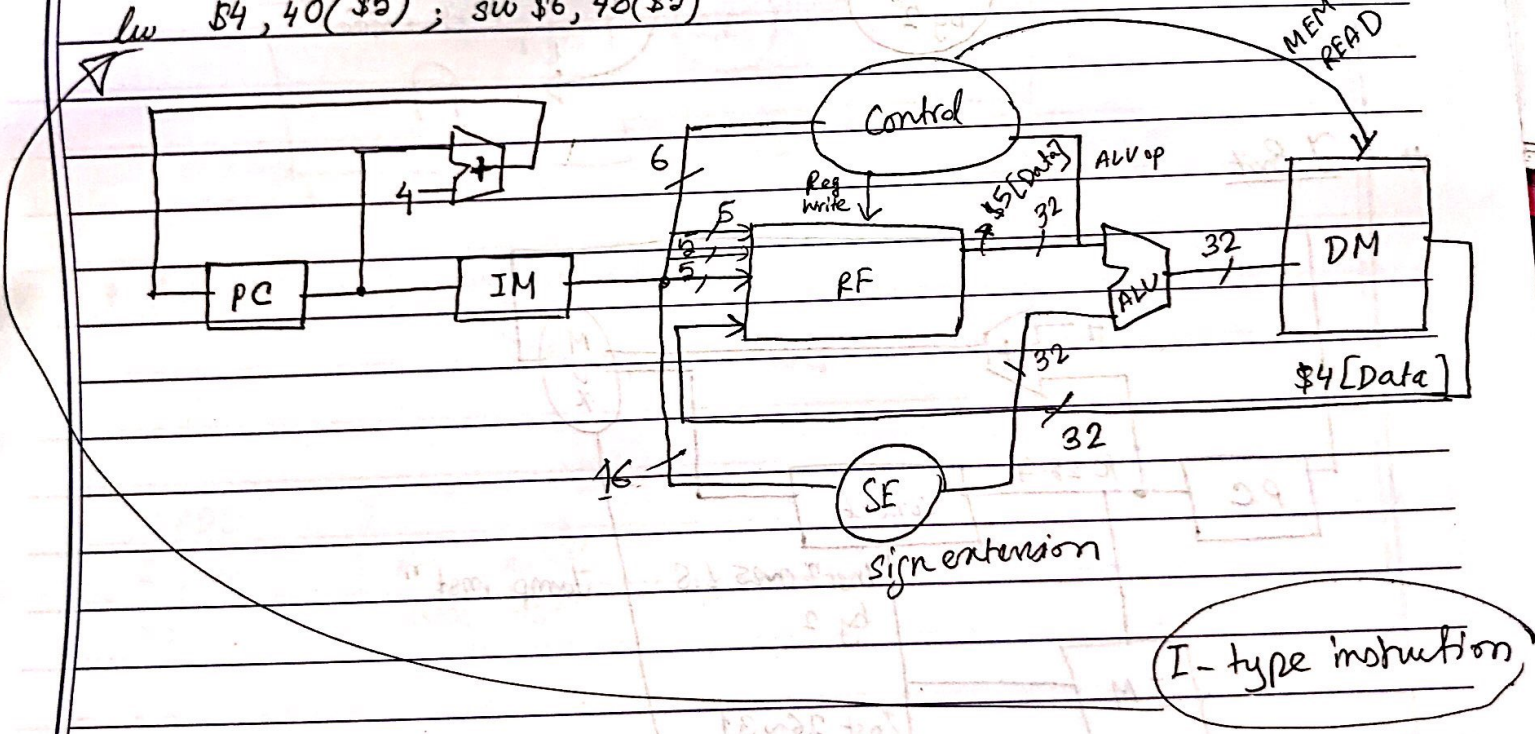
Write op ⇒ we need write op signal

ALU needs signal to know to perform what operation.

This datapath can execute almost all types of R-instructions, but not sll. (more extended)

Sub $5, $6, $7
Add $5, $6, $7
AND $5, $6, $7

lw $4, 40($5) ; sw $6, 48($5)



I-type instruction

lw → read
sw → write op

* (back link and Reg Write theke bena, see pic) *

MEM write, not MEM Read

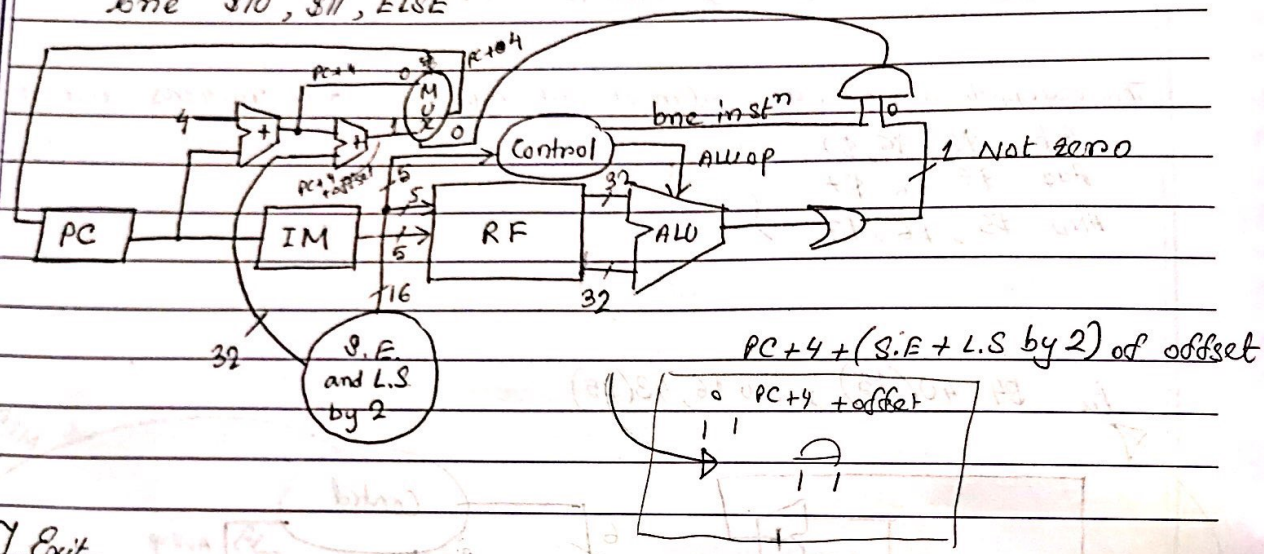
Lecture

16th March, 2020

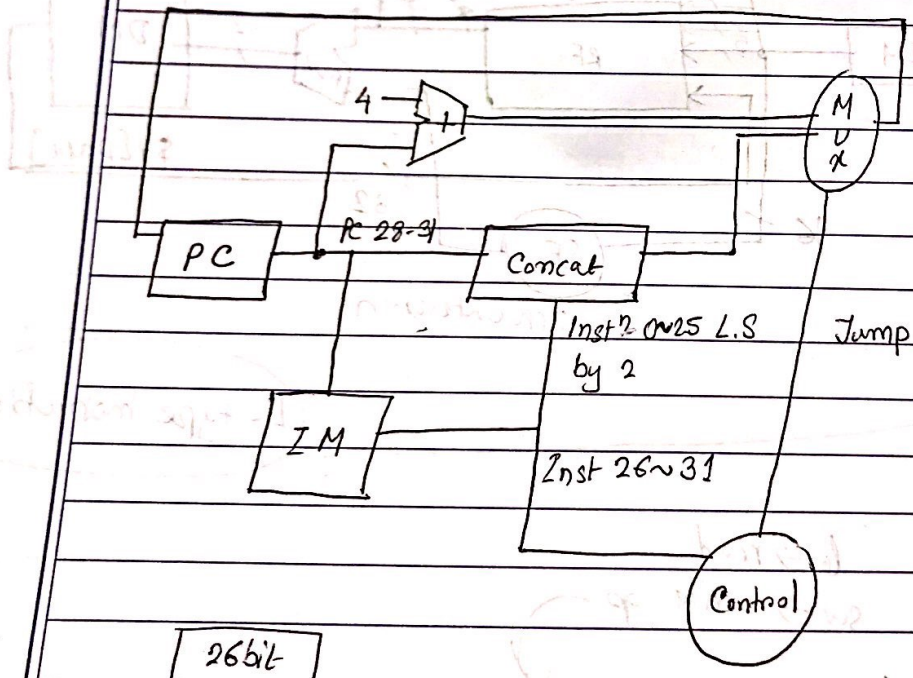
For shift, we need a diff. datapath.

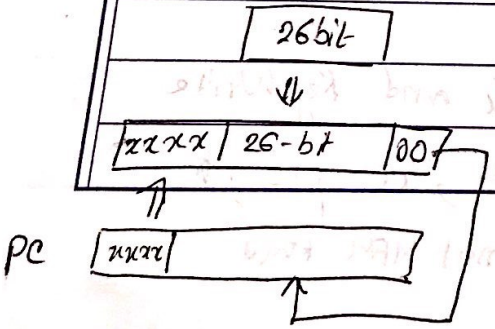
Designing a datapath for executing

bne $10, $11, ELSE



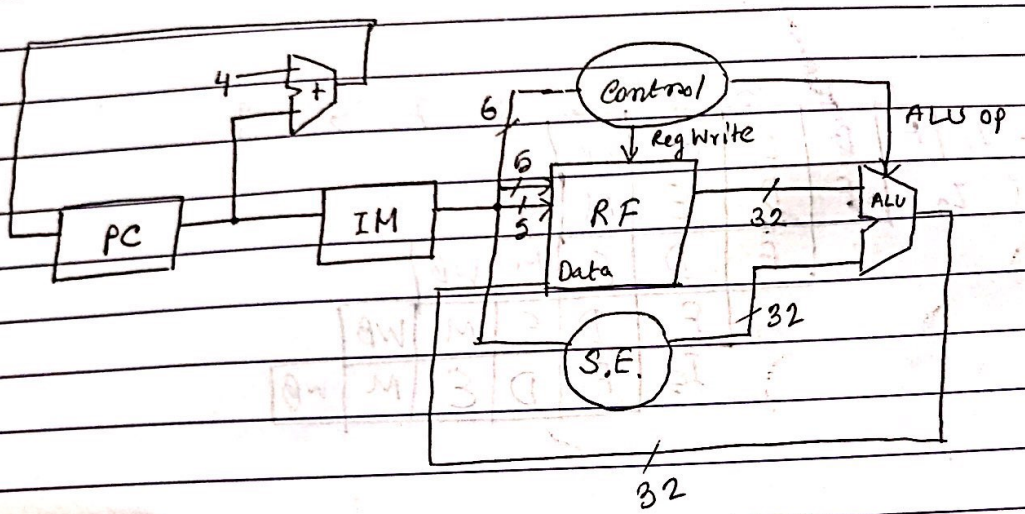
J Exit





Design datapath for

Add i $10, $20, 100



* datapath with control (Slide)

→ imp for finals

1. ADD
2. Sub
3. lw

} R.F. used by all these instn

one clock cycle → how much time load needs

	1	2	3	4	5	6	7	clock cycle
I_1	F	D	E	M	WB			
I_2		F	D	E	M	WB		

Instruction 1

2

.

.

.

	1	2	3	4	5	6	7	8	9
I_1	F	D	E	M	WB				
I_2		F	D	E	M	WB			
I_3			F	D	E	M	WB		
I_4				F	D	E	M	WB	
I_5					F	D	E	M	WB

Pipeline improves the performance of system and throughput

A[offset]

$10 $10

A[5] = A[6] + A[B[4]] + 5;

B[i] = A[5] - 6;

A = $10
B = $11
i = $8

- lw $t0, 24($10) // A[6]
- lw $t1, 16($11) // B[4]
  sll $t1, $t1, 2 // shift 2 bit-
  add $t1, $t1, $10
  lw $t1, 0($t1)
- add $t2, $t0, $t1 // A(6) + A(B(4))
- addi $t2, $t2, 5 // A(6) + A(B(4)) + 5
- sw $t2, 20($s10) // A[5] . . . .

~~addi $t~~

- lw $t0, 20($s10)
  addi $t0, $t0, -6

Process Synchronization

entry section → sets lock
→ trap

critical section → manipulating the shared variable

mutual exclusion (only one process comes at a time)

Exit section → lock release

Remainder Section

Semaphore (Synchronization tool)
[variable]

Binary
0 or 1
(Mutex)

Counting
(finite

Bounded Buffer ~~Soln~~ Prob / Producer Consumer Problem

Producer (increment) Consumer (remove ; decrement)

We use Semaphore to solve this

① Mutual Exclusion

S : 0 or 1 (mutex / Binary) [C.S]
E : size of the buffer [how many empty slots]
F : 0 [no. of filled slots]

Quiz 2 - 23rd
Threading, CPU scheduling
Quiz 03 Deadlock, Semaphore

create a class
that will extend thread

myclass
thread
(override)

concurrently

myclass.run(); // myclass.start();

Thread.currentThread.getId();
↳ wants from thread.

- sleep → Synchronize
(ekta time diye dicha)
200ms
↳ try catchen
Bhitor na dile
error throw
korbe

2. Thread T1, T2

method { runmode(); → yahoo, I'm in r.mode/sleep(1000)
wakeup(); → Emm got up from sleep

Public
static
synchronized

output :

Lecture - 01

13 Jan, 2020

Computer Organization and Design
D. A. Patterson
5th edition, Morgan Kaufmann

- DLD
- Simplifications (not k-map; use Boolean Algebra)
- MUX (multiplexers design)
- ALU (arithmetic logic unit)
- Adder
- Registers / Shift registers
- Internal Structure of Register File

32-bit Architecture

$x = y + z$ (x → \$10, y → \$11, z → \$12)

Add \$10, \$11, \$12 — R
D S_1 S_2

R
I
J

OP	R_s	R_t	Rd	Shmt	fun
000000	01011	01100	01010	00000	******
6	5	5	5	5	6

feed this and this is what the processor will understand

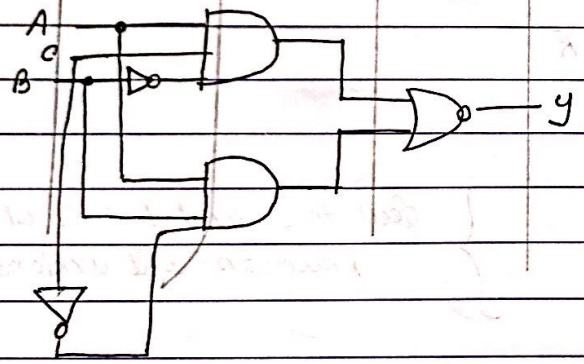
- convert into assembly instructions and identify types (R, I, J)
- each form has diff types of encoding
- all the instructions will be 32-bits.

Lecture - 02 15-01-2020

$y = \overline{A.B.C}\,(A.B + A.C)$

A B C	A.B.C	$\overline{A.B.C}$	A.B	A.C	A.B+A.C	y
0 0 0	0	1	0	0	0	0
0 0 1	0	1	0	0	0	0
0 1 0	0	1	0	0	0	0
0 1 1	0	1	0	0	0	0
1 0 0	0	1	0	0	0	0
1 0 1	0	1	0	1	1	1 → AB'C
1 1 0	0	1	1	0	1	1 → ABC'
1 1 1	1	0	1	1	1	0

SOP



$\bar{y}$
1 A'B'C'
1 A'B'C
1 A'BC'
1 A'BC
1 AB'C'
0
0
1 ABC

$\bar{y} = A'B'C' + A'B'C + ABC' + A'BC + AB'C' + ABC$

$\bar{y} = y = \overline{A'B'C' + A'B'C + ABC' + A'BC + AB'C' + ABC}$

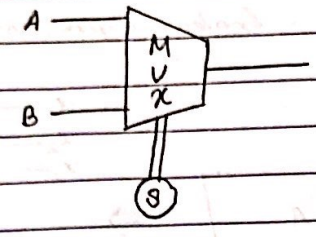
$\Rightarrow y = \overline{(A'B'C')}\ \overline{(A'B'C)}\ \overline{(ABC')}\ \overline{(A'BC)}\ \overline{(AB'C')}\ \overline{(ABC)}$

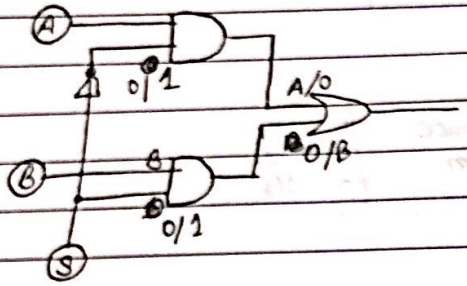
$= (A+B+C).(A+B+C').(A'+B'+C).(A+B'+C').(A'+B+C).(A'+B'+C')$

(product of sum)

POS

If S
Then y := B
ELSE y := A





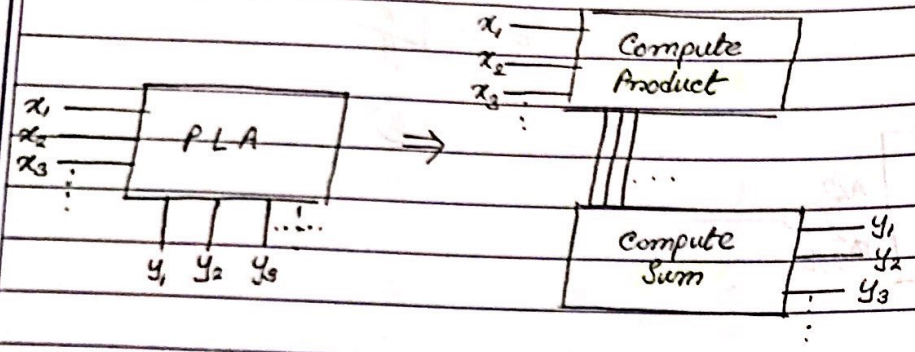
carry
& rippled
stage.

Lecture - 03

20-01-2020

PLA → Programmable Logic Array

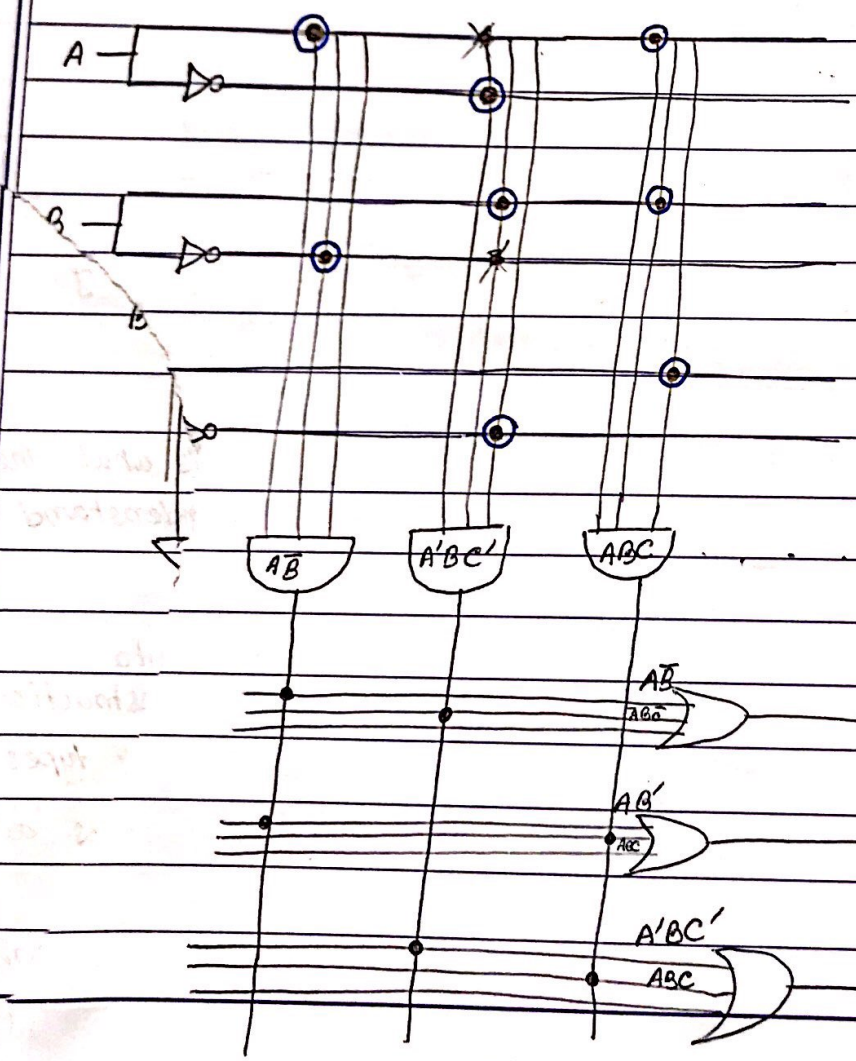
Functionality - boolean expressions



$y_1 = A\bar{B} + \bar{A}B\bar{C}$

$y_2 = A\bar{B} + ABC$

$y_3 = \bar{A}B\bar{C} + ABC$



Open Circuit

3 combinations in y_1, y_2, y_3:

$A\bar{B}$, $\bar{A}B\bar{C}$, and ABC

$y_1 = A\bar{B} + \bar{A}B\bar{C}$

$y_2 = A\bar{B} + ABC$

~~$y_3 = AB' + \bar{B}BC'$~~ A

$y_3 = A'BC' + ABC$

$y_1 = AB'C \oplus ABC$

$y_2 = ABC' \oplus A'BC$

$y_3 = A'B'C' \oplus AB'C$

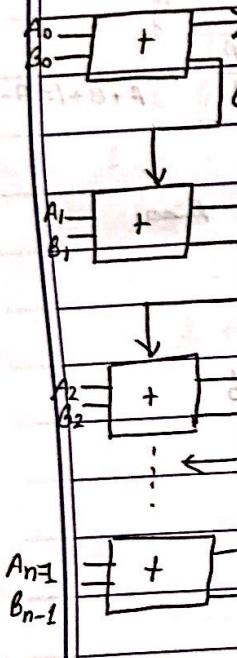
6 AND Gates
3 OR ⊕

PLA is always one circuit

✓ regardless of how many expressions, PLA is one circuit.

RCA

Ripple Adder (full adder)



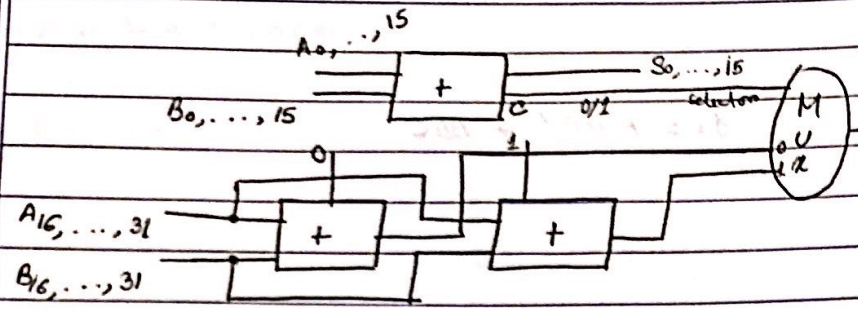
Ripple Carry Adder (RCA)

not efficient

It is called a ripple carry adder because each carry-bit gets rippled into the next stage.

The carry has to propagate from LSB to MSB.

Conditional Sum Adder / Fast Adder

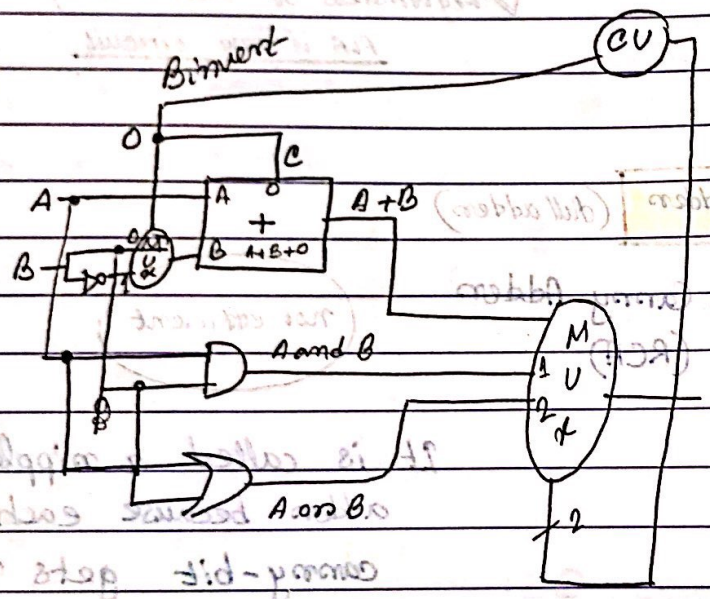


Fast Adder

CLA

$A - B = A + \bar{B} + 1$

$A + B$



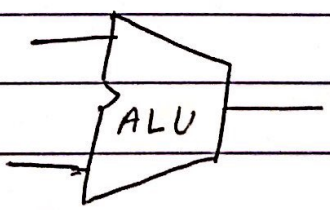
00: Add / Sub

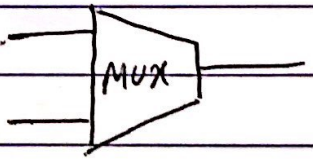
01: A and B

10: A or B

ALU - Arithmetic Logic Unit

(2 mux)



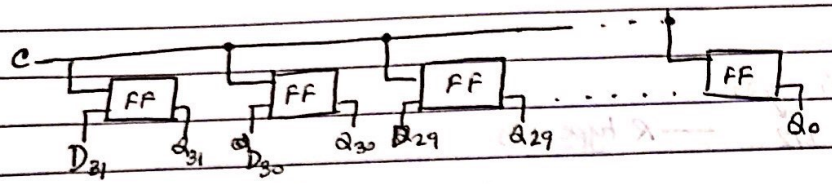


Lecture - 04

01-2020

To build the register, we need (flipflops.)

Register



32-bit register

- When we read, we've to read the entire register.

* exception: lb (allows to access first 8-bit of the register)
Byte load byte

- lb → load byte (allows to access the first 8 bits of the register)
otherwise, we've to take / read values for the entire register

1 0 0	1 0 0 0
1 0 0 0	0 1 0 0
1 0 0 0 0	0 0 1 0
left shift	right shift
L.S.	R.S.
$x * 2^i$	$x / 2^i$

i → no. of bit shift

~~$x * 2^i$~~ L.S. $x * 2^i$ left shift operation (multp)

R.S. $x / 2^i$ right shift operation (div)

L ← R

31 30 29 . . . . . 2 1 0

L R

i starts at 0 and increases from R to L

MSB (bit 31) (32 bits wide) LSB (bit 0)

$x = 3y + 5z$

$y = 4x + 3z$

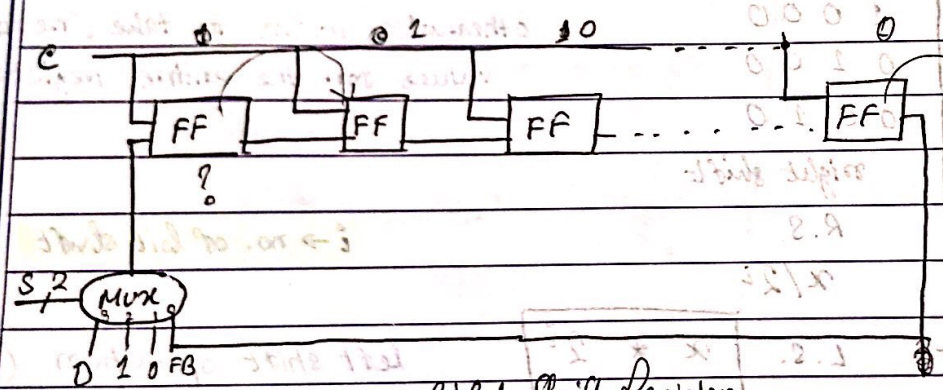
$z = 5x - 6y$

Add $10, $11, $12 — R type ins (D, S_1, S_2)

mult $11, $12 — R type ins

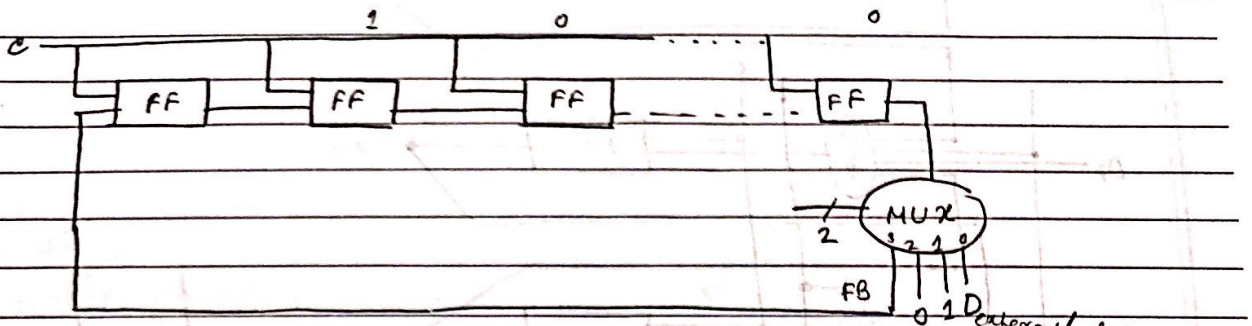
↳ Result [Lo, Hi]

Designing Right Shift Register.



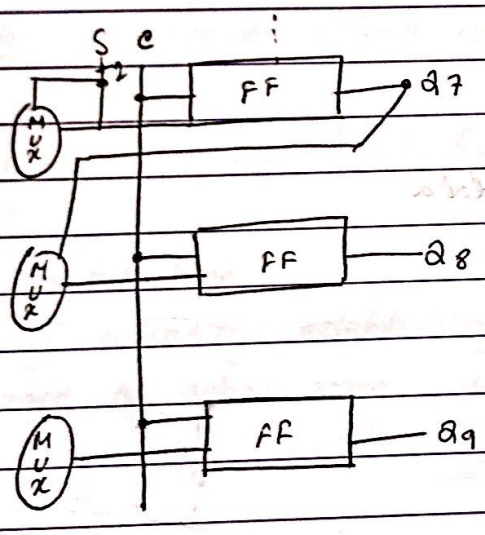
Right Shift Register

F	00 : Feedback	random no. coming from some external source
R	01 : Reset	
S	10 : Set	
E	11 : External data	



Left Shift Register

0	00 : External data
1	01 : Set
2	10 : Reset
3	11 : FB



00 : LS

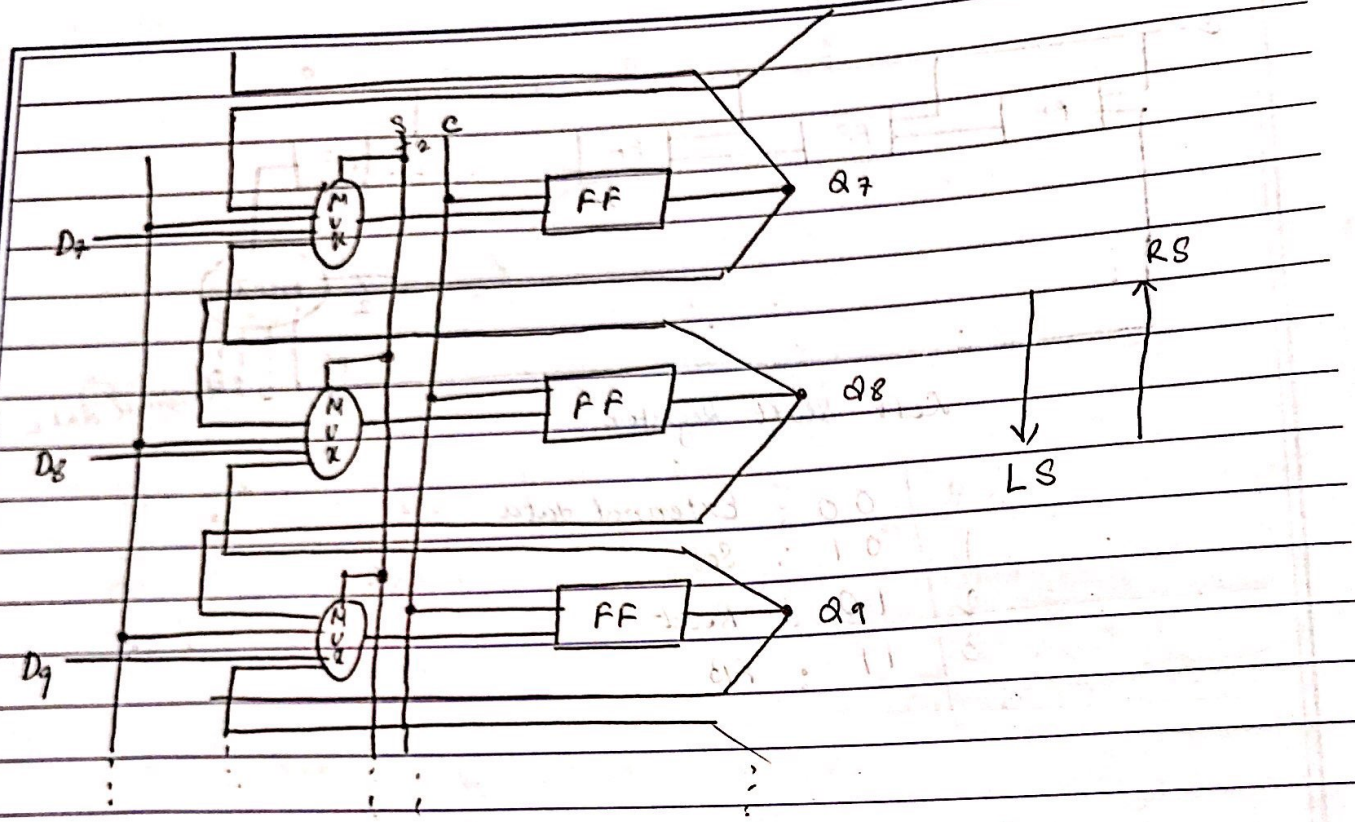
~~01 : RS~~

11 : RS

01 : Reset

10 : External data

See: next page



00 : LS

11 : RS

01 : Reset

10 : External data

* key: bucse340sp

Lecture-05

27th Jan, 2020

Chapter 1

- Moore's law
- API
- Interface
- Parallel Processing

high-level language —(Compiler)→ assembly instruction —(Assembler)→ machine instruction (this is what the machine understands)
(C or Java)

Application Software is written in high-level language.

Controller
Control Unit responsible for allowing CPU to carry out operations.

- **datapath**: components & path connecting CPU

Abstraction - hiding the detail from user

Computer Architecture
- designing, selecting, establishing interconnection
- interface to achieve some goals like functionalities

set of instructions { Add, sub, bne, beq, $11,

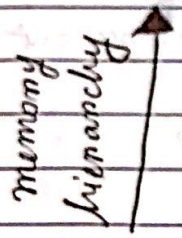
{ MIPS

x = y + z
$10 $11 $12

Add $10, $11, $12
$s1 $s2
Destination Register

* definition's on the diagram.

• memory tends to get smaller higher-up. the memory hierarchy.



(and vice versa for down)

cost, speed, size

• multiprocessor shares a RAM / memory.

Role of the Architect

- backward (literature review, existing designs)
- forward
- look up
- look down (scale down the design for the current tech to accomodate the idea into reality) e.g. limitations with transistors

Lecture - 06

- 01 - 2020

We need trade-offs when considering performance.

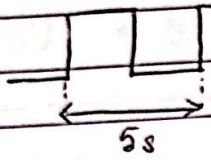
- Response time and throughput.
- Relative performance

$$\frac{P_x}{P_y} = \frac{E_y}{E_x} = n$$

P → performance

E → execution time

→ x is n times faster than y

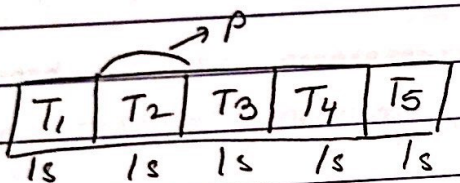


(one clock cycle)

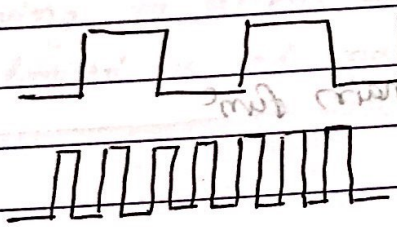
Q. If program has 5 clock cycles, then, CPU time?

Ans: CPU Time = 25 s

For comp B, we need higher clocks i.e. higher GHz.



R_E = 3 + 4 × 2 + others waiting



RISC vs. CISC

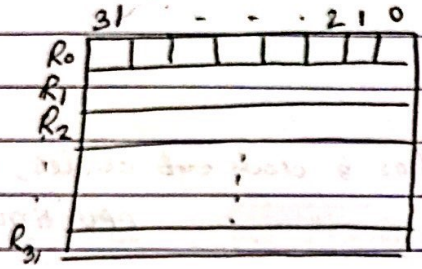
0302-2020

Lecture 07

Computer Language

Machine Language → Assembly Instruction

- identify the registers which store the values of variable



$t0 ~ $t9

$S0 ~ $S7

$$S \rightarrow f = \frac{(g+h)}{t} - \frac{(i+j)}{t}$$

* $0 / $zero

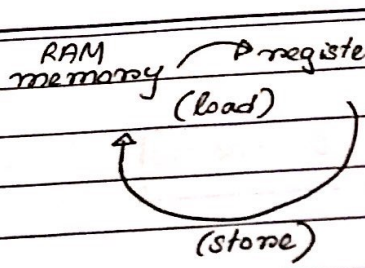
* $ra / $31 → give return address to main func

$sp → point to stack

$fp → frame-pointer points to stack

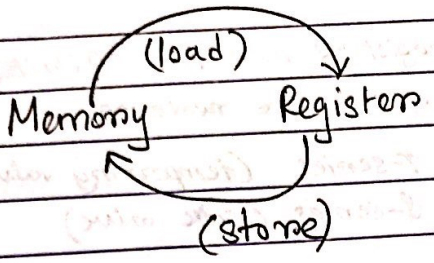
$a0 ~ $a3 → argument registers used in assembly instruction

$v0 ~ $v1 → return func



load (memory → register)

store (register → memory)



MR. load

store	load
A[10] = y + A[6]	

• memory is byte-addressed

Processor operations mostly involve processing data.

To speed up the processor operations, the processor includes some internal memory storage locations, called registers.

The registers store data elements for processing without having to access the memory.

A limited no. of registers are built into the processor chip.

Registers vs. Memory

- load and store operations are **Memory Operations.**

32 registers with each 32-bit.
Some of these are reserved.

T-series (temporary value) } available to us
S-series (save value) }

- spill
- optimization

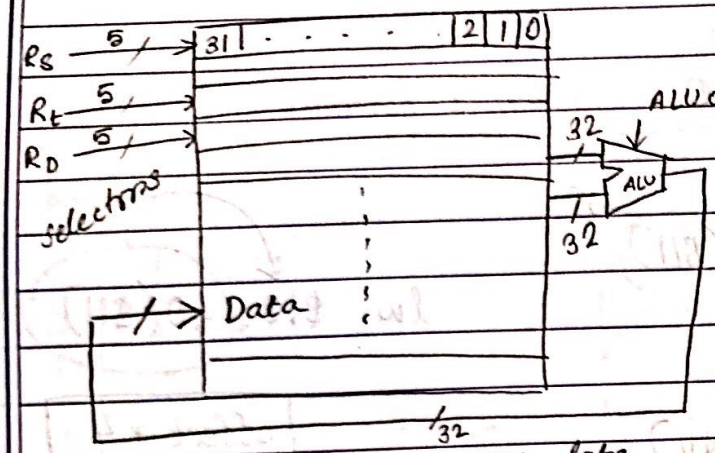
load, store, immediate → I-type Instruction

```
$10   $11   $12
x = y + 2;

x = y + 10;

Add  $10, $11, $12 → R-type ✓
Add i $10, $11, (10) → I-type ✓

x = y - 10;

✗ Subi $10, $11, 10   (doesn't exist; No Subtract immediate Instruction; Just use a negative constant)

Addi $10, $11, -10
```

Please Note:

Registers are processor components that hold data and address.

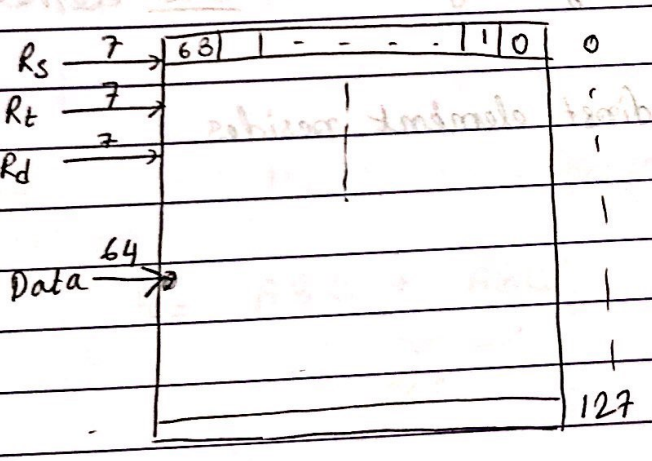
Fig: Register File



Data-line to carry data

$2^5 = 32$

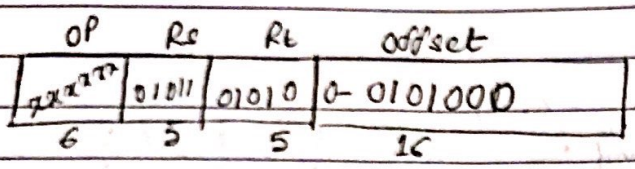
- Register's size anyone operation long



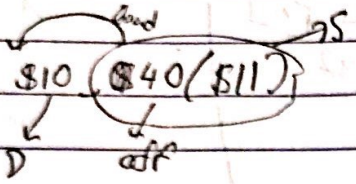
(no. of register changed, bit-length changed)

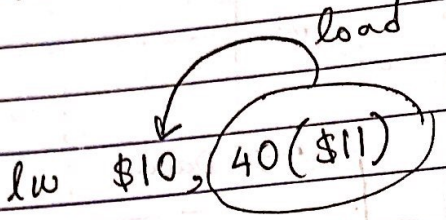
$2^7 = 128$

* • Better to encode numbers rather than names.

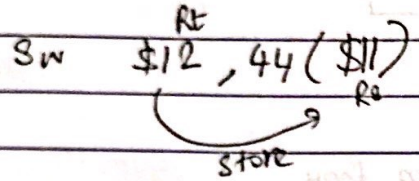


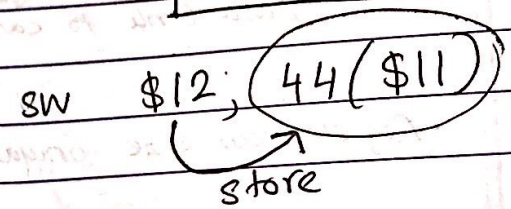
I-format





offset * 4





Address where the first element of array stays is the base element.

Base Address

Address where the array's first element resides.