for complete execution to make a program
1. Logical operation
2. Memory communication (Read + Write)
3. Change flow control [:

Instruction should define

1. Operation + what operator to use
   Op 1 (operand)
   Op 2
   Destination
   → Addresses can be explicit or implied
   ↳ Addresses mentioned explicitly
   ↑ pre-specified

32 bits needed to specify memory location.

$\lceil \log_2 (\text{Total No. of operations}) \rceil$ → to uniquely specify the operation

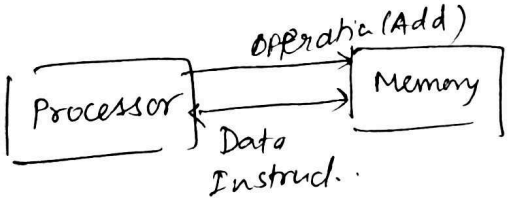
Constraints
↳ 1. Limited Memory → Minimize No. of bits per instruction

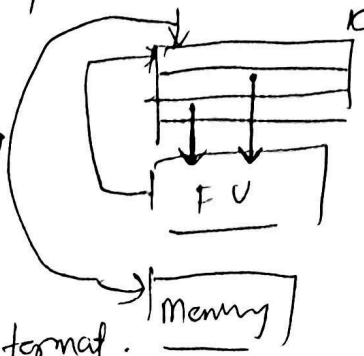
Program stored in magnetic strips.
when running pgm (> RAM), we bring parts of RAM & overwrite it again & again to run the program.

2. Slow Memory → (complex instruction) Read/Write operation slower than processor operation
3. Ease of programming (Machine level language used at first level)

Implicit operands → 0-address format → only specify operation
[No of bits per instn ↓]

follows LIFO
↓
take top 1/2 element
compute, write result on top





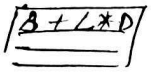
2 additional
PUSH →
POP
instruction
which need to
have 1 address format.
[39 bits] = 32+7

↑ filled by seperate instr which transfer data betw memory & stack.

Eg $A = B + C \times D$

① PUSH C 39
② PUSH D 39
③ MULTIPLY 07
④ PUSH B 39
⑤ ADD - 7
POP A 39





If stack is not sufficient, then pop to some empty temp. location T1.
compute the rest-
and then bring the T1 data back to temp. memory stack and compute.

Eg if only 2 in stack 
the $A = B + C * D + E / F$
1, 2, 3, 4, 5. POP T1, PUSH E. PUSH F, DIV., PUSH T1, ADD

* Write in postfix notation.
Scan → ① Encounter variable → push
② Encounter operator → perform operation

$A = B + C \times D$
$= BCD \times +$

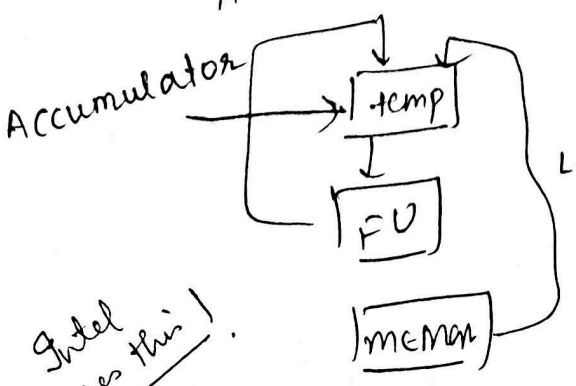
PUSH B
" C
" D
MULTIP.
ADD
POP A

All instructions of 39 bits

using 1 address format everywhere. → OPERATION OPERAND
One operand from temp,
" " " memory

Accumulator based architecture

A[1]



Intel uses this.

LOAD | STORE location
(-Instruction to transfer from memory)

A = B + C
LOAD B
ADD C

B X C + D X E

A = LOAD B
MUL C
STORE temp ← from main memory location

LOAD D
MUL E
ADD T1
STORE A

★ ACCUMULATOR BOTTLENECK

Stored in register.

OPERATION	OPERAND
7	2

= 10 instead of 40 bits

ADDRESSING MODE
Register or memory

OP	Add Bit	Operand
7	1	32

= 40

A = B + C
D = A + C
P = A * R

LOAD B → 40
ADD A → 40
STORE R0 → 10 [storing in register R0]
LOAD E → 40
ADD R0 → 10
STORE D.

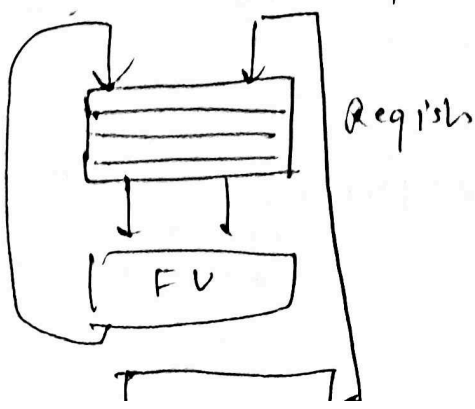
Able to target limited memory as well as slow memory

Cost of registers ↑↑

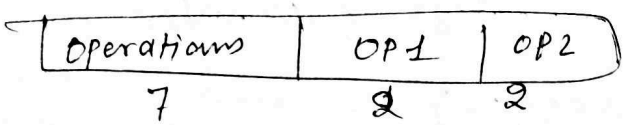
Programmer specifies and decides which variable used multiple times & stores it in the registers.

Using only registers!

↳ Need main memory to load / unload registers



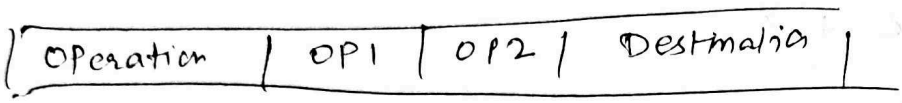
6 ~~ALU~~ Instructions (Arithmetic Logic).



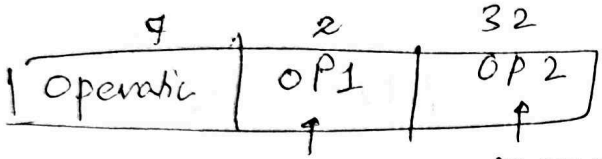
= 11

main memory can be accessed by load & store

go for 3 address format - bcoz cost only 2



LOAD & STORE FORMAT



= 41.

Ex $A = B + C$ (R0 ← B, R1 ← C, R2 ← A)

R4 ← $D = A + E$ (R3 ← E)

$P = A + R$

```
LOAD    R1, B
LOAD    R2, C
ADD     R1, R2, R0
LOAD    R3, E
ADD     R0, R3, R4
STORE
```

Registers only for SCALAR No VECTORS like array.

ONLY 4 registers!

CLEAR SPACE

LIVENESS RANGE: → the time where the var. is being used. [use registers for them until till that time]

32 regis can store so as of live variables

Once it overflows, then add 1 line variable to the main memory & bcoz it will be used

Before execution, everything in memory
while " , something in memory, something in register

$$A = B \times C + D$$

for the processor speed to be dominate, we need to bring entire thing in a lot not in chunks.

```
MUL R0, R1, R4.   (+1000+10)
ADD R2, R4, R3    (+1000+10 → 2020)
```

→

```
MAC R1, R0, R2, R3
```

1000 + 10 + 10 = 1020

$$C = \sum_{i=0}^{1000} A_i B_i$$

= CONV A, B, C

= 1000 + (10+10) * 1000

= 1000 + 20,000 + .

heavy computation → Faster processor significant decrease in computational time.

= 21,000

Array addition dealt by using pointers (dynamic addressing)

Specify addressing mode by spcl. field

- MEMORY INDIRECT → Direct memory Addressing
- AUTO INCREMENT / AUTO DECREMENT → Use directly content of register (content is operand)
- IMMEDIATE (VALUE) → Register Indirect (Address given in register) Content is address Operand is at that address.
  - BASE + OFFSET
  - BASE + INDEX

Addressing Mode.

operation	A/M	operand 1	op 2	Dest

Base + Offset: Base offset — to reach

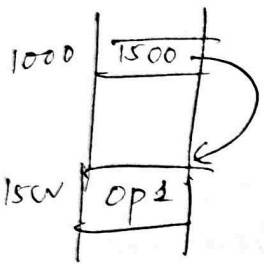
Base + Index: 2 registers

A[i] = Operand 1 → Base | Index

Base add of A; integer i

Size x index

Memory Indirect :→ Double pointers.



Register stores the address of memory which contains the address where operand is stored.

Auto Increment / Decre (kind of stack).

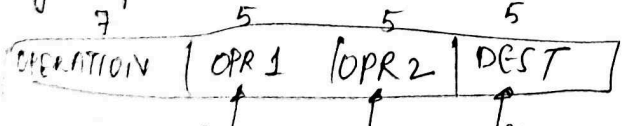
PUSH R1, R2

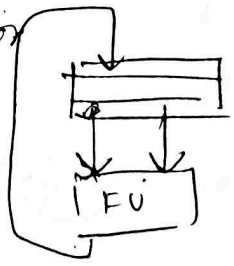
Automatically done in auto incre mode { M(R2) = R1
R2 = R2 + 1

IMMEDIATE (VALUE): value (constant) given as instruction

Alm for all operands & destinations

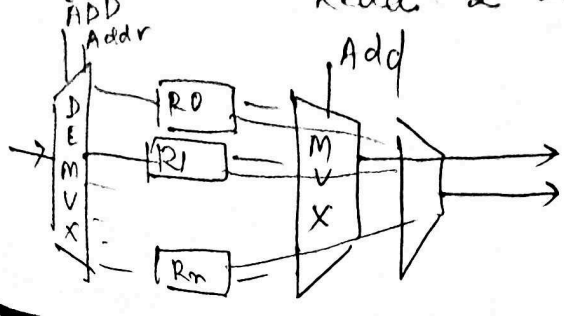
Using simple instructions:





very simple
So I know before how many operands

Hi is generated directly to FU.



Read 2 out of n registers & write to a third one.

Architecture called RISC → Reduced ISC (Simple Instruct

↳ Reduces memory communicati for memory (thus, faster as

x86 used by Intel use CISC.

CISC FIXED ENCODING ← can process in parallel.

Operation	A/M	Opr1	A/M	opr2	A/M	DESTNO
7	3	32 or 5	3	22 or 5		= 112 bits

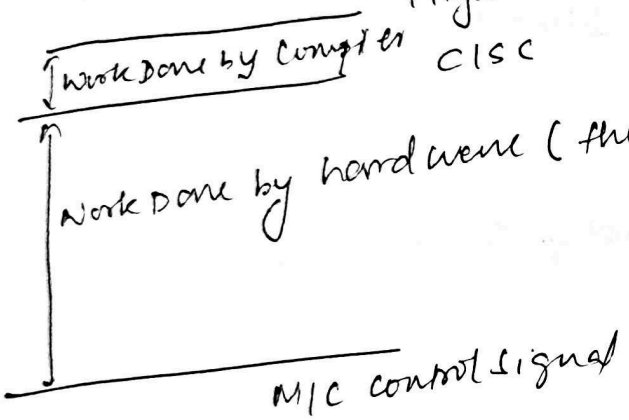
Register Addressing mode ↓ Use Max = 32

Using max is wastage of memory.

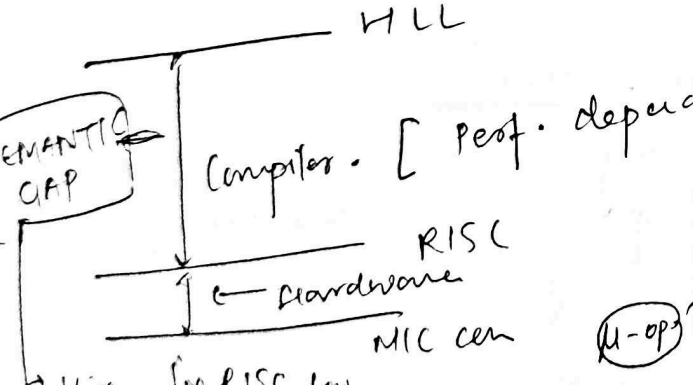
VARIABLE - LENGTH ENCODING ← Don't know len of instruction. So Not in parallel?

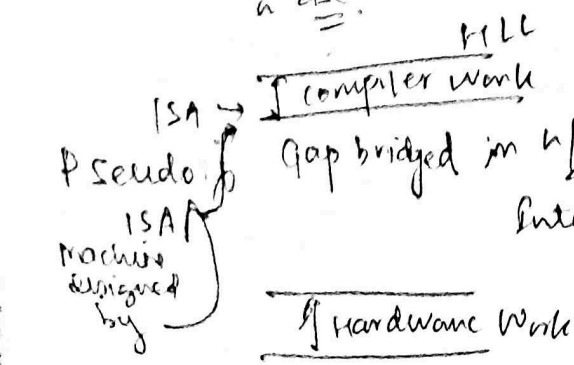
difficult to parallelize

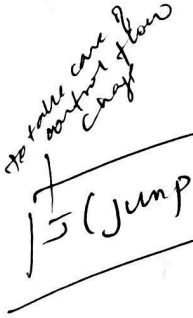
RISC → fixed CISC → variable length encoding

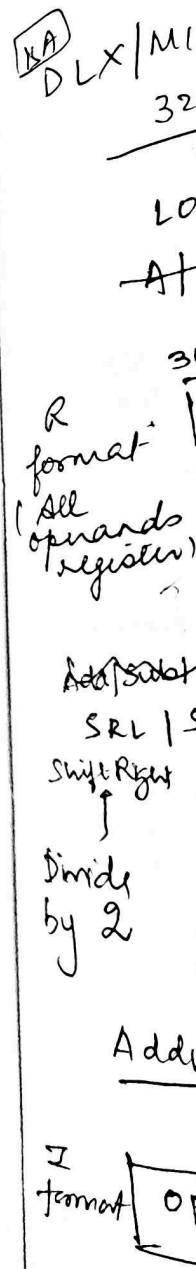


load/store access memory
Rest - ALU op. access only registers





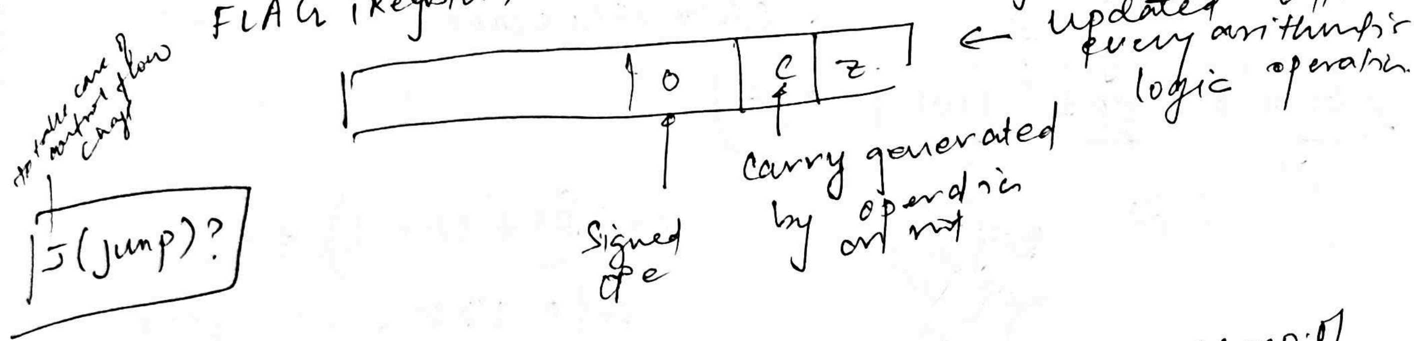




CONTROL FLOW CHANGE

Unconditional / Conditional - Data Dependant.

FLAG (Register) = Condition Code / Status register.

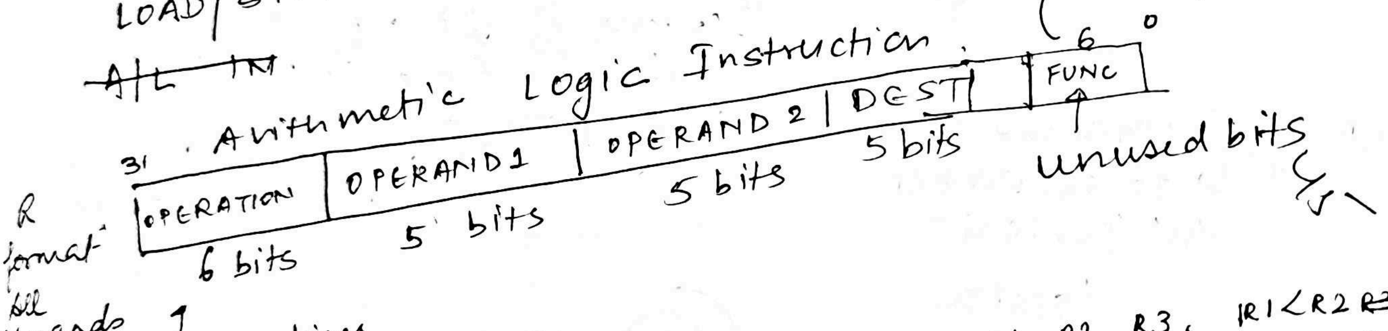


ISA
DLX / MIPS - (RISC-V)
32-bit ISA | 32 REGISTER R0 - R31 | uses small no. of addressing mode.

LOAD / STORE ARCHITECTURE.

~~A/L IN.~~

Arithmetic Logic Instruction.



64 operations encoded.

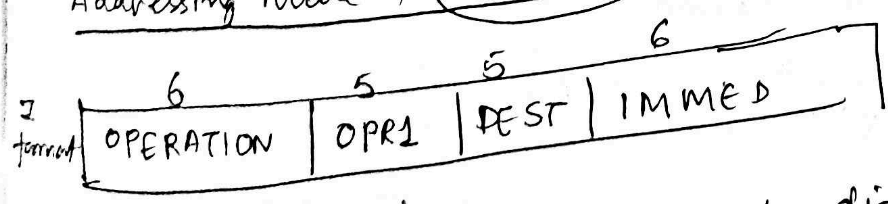
SLT R1 R2 R3. R1<R2 ~~R3~~ R3= 0 0
else R3= 0 1

~~Add / Sub / And / OR~~

SLT } less than
SLE } ?- equality
SGT } Greater than.

SRL | SLL
Shift Right → Divide by 2
Shift left operator → Multiply by 2

Addressing Mode → (Immediate) when the freq is high of 1 operand



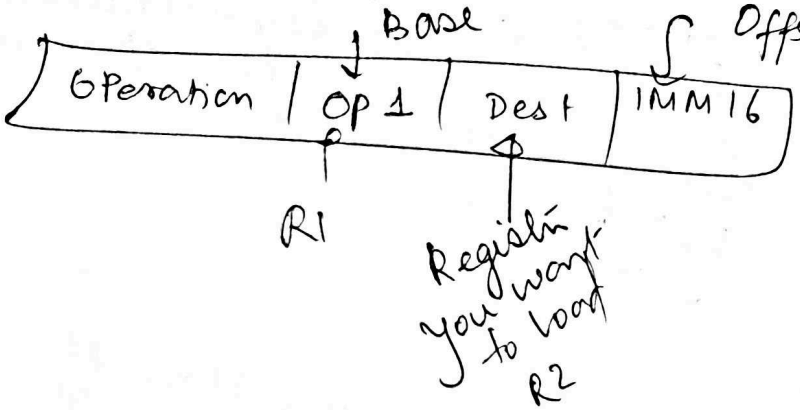
ADD ADDI → treated differently
↑ this one for immediate operations.

64 (000000) = 64 op.
f^n has 2^6.
will use other (63) combination
∴ 64 + 63 = 127 opn

B + D ← I-type format
Base Displacement

(2) B + I ← R-type format
Base Index

(provide 2 address R1, R2 & register)

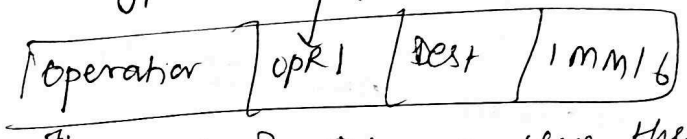


R2 ← M(R1 + IMM16) LOAD

M(R1 + IMM16) ← R2 STORE

BEQZ REG LOC ← I-type.

How far you need to go



If content of R1 = 0 then then it should IMM16 instructions away from here.

[4 * IMM16 from the current location]

Mostly Programmer Counter ↳ to remember the current location

IP = IP + IMM16 × 4

IP → Instruction Pointer

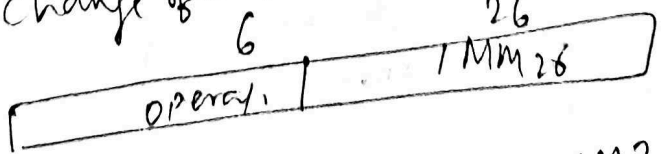
4 because every instruction is 4 byte

BVEQ REG, LOC [transfer → conditional like f^n call

for unconditional change of location,

J LOC

jmp



IP = IP + IMM26 * 4

example

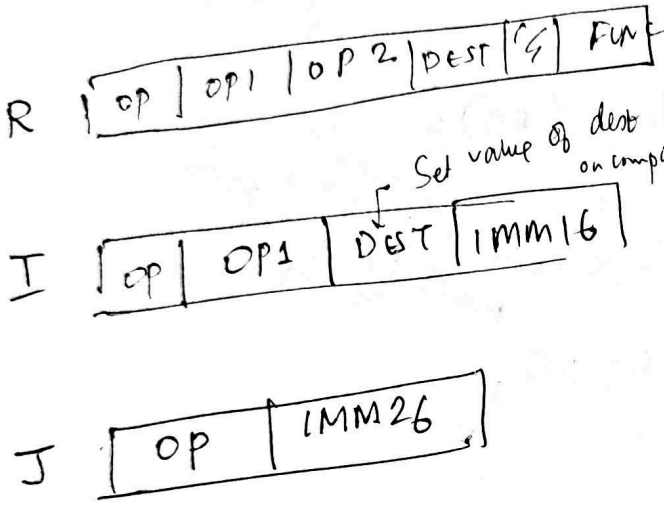
JAL LOC R31 jump

↑ loads the curr loc to R31 & then move to desired local.

original ↓

JR REG → Jump and Return (to the address in Register)

R31 usually stores but JR can use any register



BEQZ RX, LOC
/BNEQ ↑ in terms of no. of instructions [every inst. = 4 byte]

JAL. B

↳ Here in R31 (B+4) ← After function call, next instruction

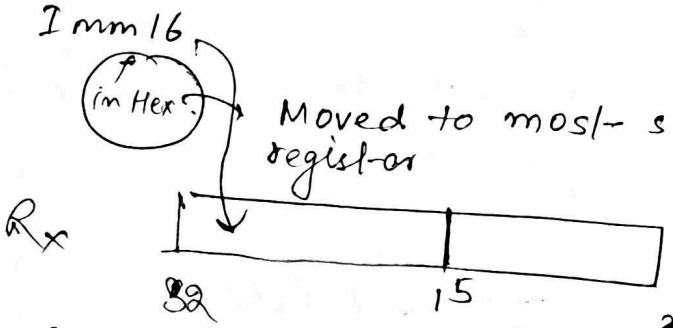
JAL C ↳ R31 overwritten by C+4 so, we lose B+4. so when JR R31 it will again execute C.

↑ Save in free register. ADD R5, R4, 0

↳ No free register, make a stack in memory. Use a stack pointer.

JR R31

LHI Rx, Imm16



Read only Register R_0 which stores only 0
Rest of register = Read & Write

ADD R5 R0, 1000H — location of stack

Acts as a stack pointer

to store

SWI R31 (R5) 0 ← offset-0

↓ Now increment because pointer must always point to top

ADD R5, R5 4

to return:

SUBTRACT R5, R5 4

SWI R31 (R5) 0

to calc sum of elements of array
Base: 1000H

A[100]

R1 → counter = 100
R2 → #ELEMENTS = 0 SUM
R3 → 0001000

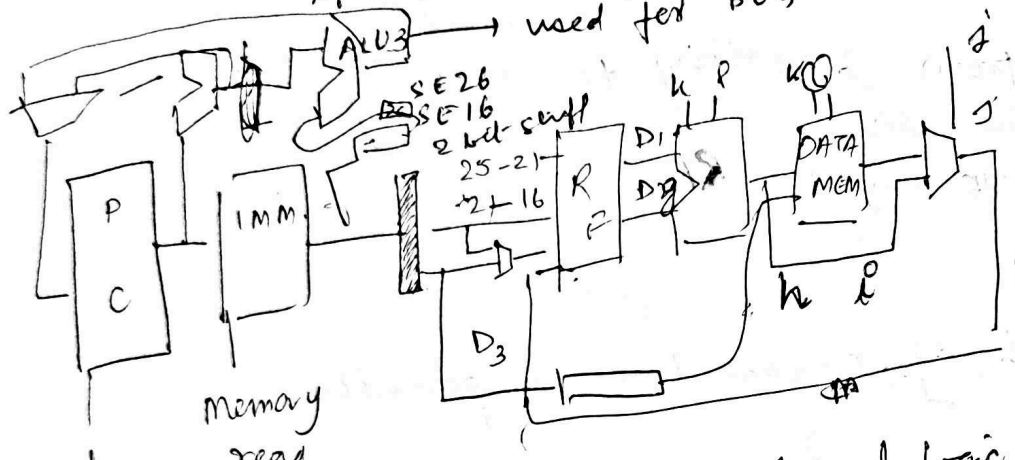
```
ADD   R1  R0  100
ADD   R2  R0  R0
ADD   R3  R0  1000H
LOOP: LOAD  R4  (R3)0
      ADD   R2  R2  R4
      ADD   (R3)0  R3  4
      SUB   R1  R1  1
      BNE0  R1  LOOP = (-5)
```

in terms of no. of instructions

STORE R2 (R3) 0

HLT

For function calls with parameters, store parameters in regists & push them into stack.
↳ push right to left. So that on popping, same order is kept.
once pushed, stack pointer — appropriate value of ACCUM stored.



combinational logic.

	a	b	c	d	e	f	g	h	i	j	k	l
R 000000	1	1	1	X	1	1	0	0	0	1	1	0
L 000100												
S 000010												
BEQ 001000												
J 010000												

Always

$CPI = 1$

time required

2 memory — 1 for program
1 for data

$\frac{\text{time}}{\text{prog}} = IC \times CPI \times \tau$

$\times 1 \times$

depends on the circuit.

Who decides the time, the longest path.

dynamic instruction count

$\text{freq} = \frac{1}{8 \times 10^9 \text{ sec}} = 125 \text{ mHz}$

All operations in sequence ← read from reg, write a reg

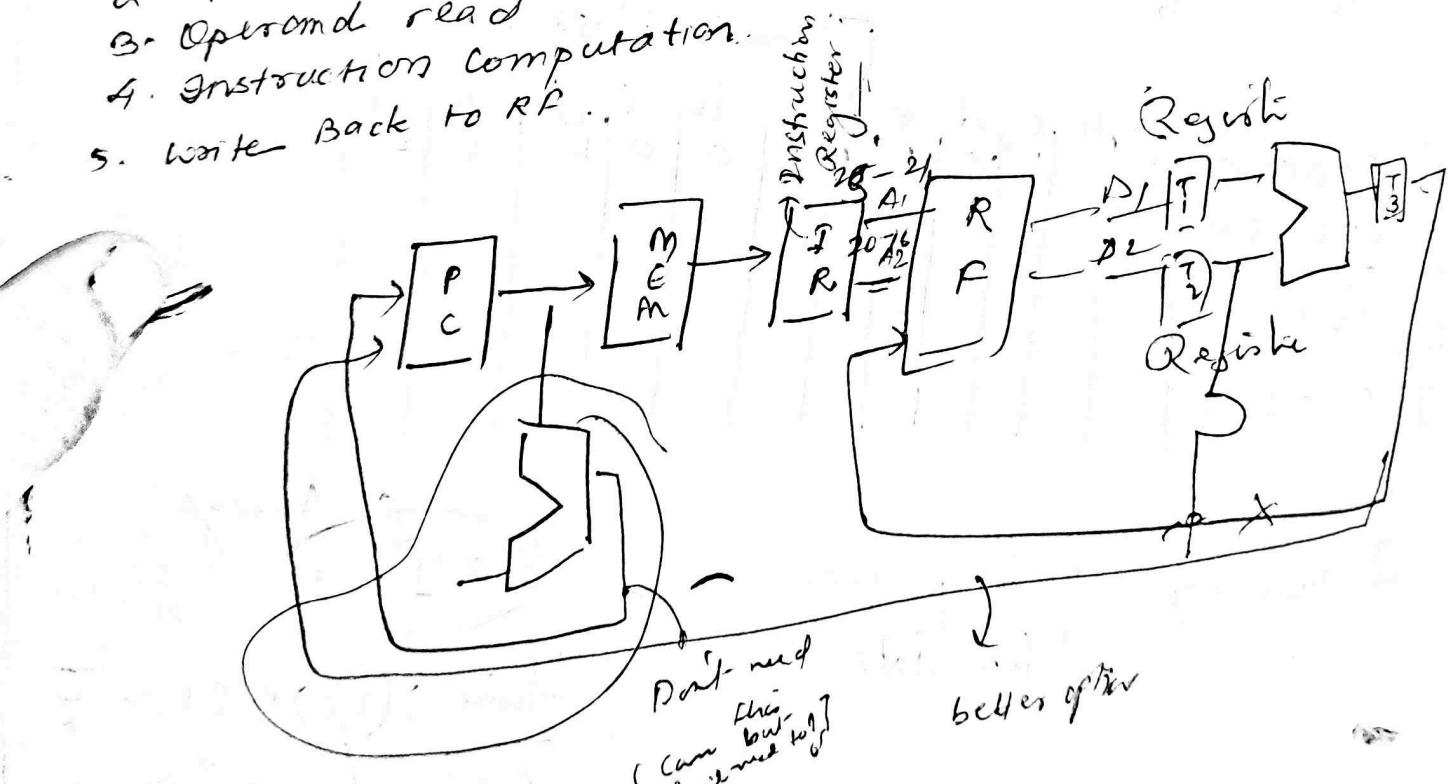
$S \to S'$
$S \to P \to P' \to P'' \to S'$ make a transition system.

SUB-TASKS

1. Fetching of instruction. (from the memory & update the program counter
2. Update Program counter.
3. OPERAND READ
4. Computation of Instruction or Computation of address of memory in case of load or store
3. Reading from & writing to memory.
6. Update the state (write to register RF.

A & L inst-

1. Inst. fetch.] → Can be done in parallel.
2. Update PC
3. Operand read
4. Instruction computation.
5. write Back to RF.



* BEQ R_{S1}, R_{S2}, Imm

OPERATION	OP1/DEST	IMM16

If $R_{S1} == R_{S2}$ then

$PC = PC + 4 + IMM16 \times 4$

ELSE

$PC = PC + 4$

1. fetch instruction.
2. Update PC ← PC+4
3. Read operands (R_{S1} & R_{S2})
4. Compare (Subtract)
5. If zero then
   $PC = PC + 4 + IMM16 \times 4$.

S13

PC → Mem A, alu A

+4 → alu B

alu = Mem D → PC, IR

S14

IR_{25-21} → RF − A1

IR_{20-4} → RF − A2

RF − D1 → T1

RF − D2 → T2

T1 → alu A

T2 → alu B

PC → alu2 A ← precompute here we have instruction & PC. thus can precompute

IR_{15-0} → SE16 → 25 → alu2 − B

if (zero == 1) then

alu2 · C → PC

Not efficient

PC → alu A

IR_{15-0} → SE_{16} → 25 → alu − B

alu − C → T3

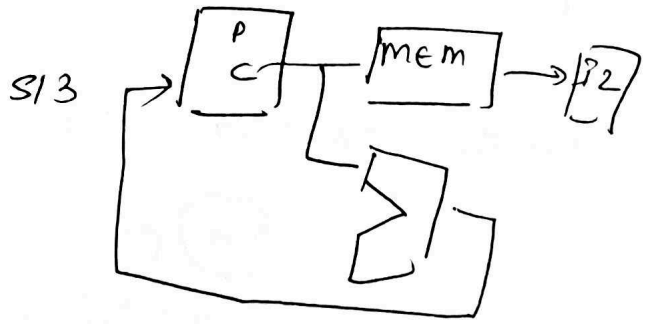
T1 → alu − A

T2 → alu − B

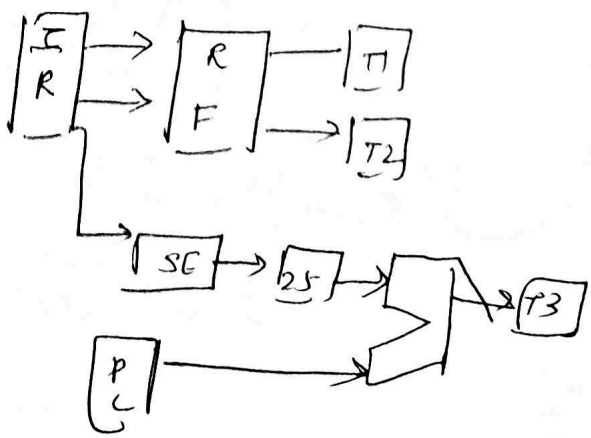
if (zero == 1) then S15

T3 → PC

S13



S14



J

OPERATION	IMM26

1. fetch instruction
2. Update PC ← PC + 4
3. Compute PC + 4 + Imm26*4
4. Update PC.

pc → Mem A, alu A
+4 → alu - B
alu C → PC
Mem D → IR

S16

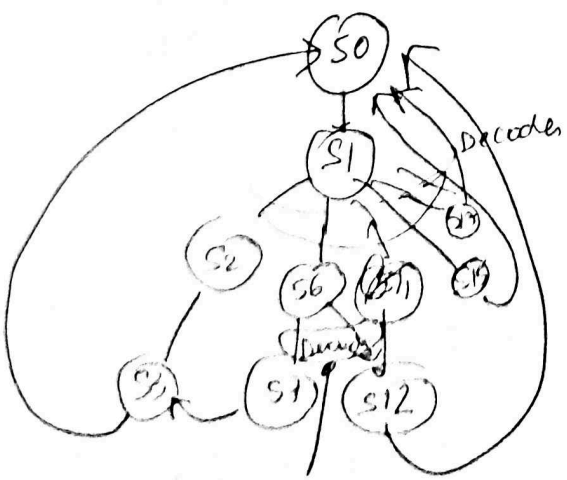
↓

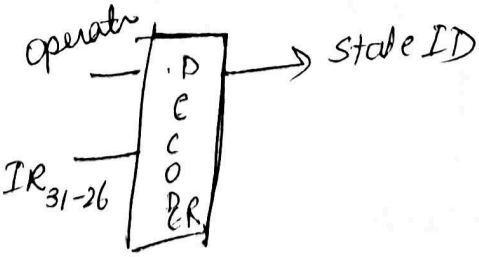
pc → alu A
IR_{25-0} → $SE_{26 \to 25}$ → alu-B
alu-C → PC
~~pc → alu A, mem A~~

S17

Operation was S0 = S4 = S9 = S13 = S18 ← different in the sense of next state evaluation

S1 = S5 = S10 = S14 = S7





1 bit that says store or load.

↳ Advantage

Share resources. (No need of multiple alus & memories

clock-period → ~~Max Delay~~ Max Delay at any of the stages

ALU – 2ns
Mem – 2ns
RF – 1ns

Operation at 2ns

R → 4 cycle
LW → 5 cycle
SW → 4 cycle
BEQ → 3 cycle
J → 3 cycle

R	S2
LW	S6
~~SW~~	S6
BEQ	S15
J	S17.

$$\text{ClockPeriod} = IC \times \underset{\uparrow}{CPI} \times T$$

Avg. (Arithmetic Mean) over all possible instructions ↑ only if instructions are uniformly distributed.

Now performance dependant on distribution of instructions.

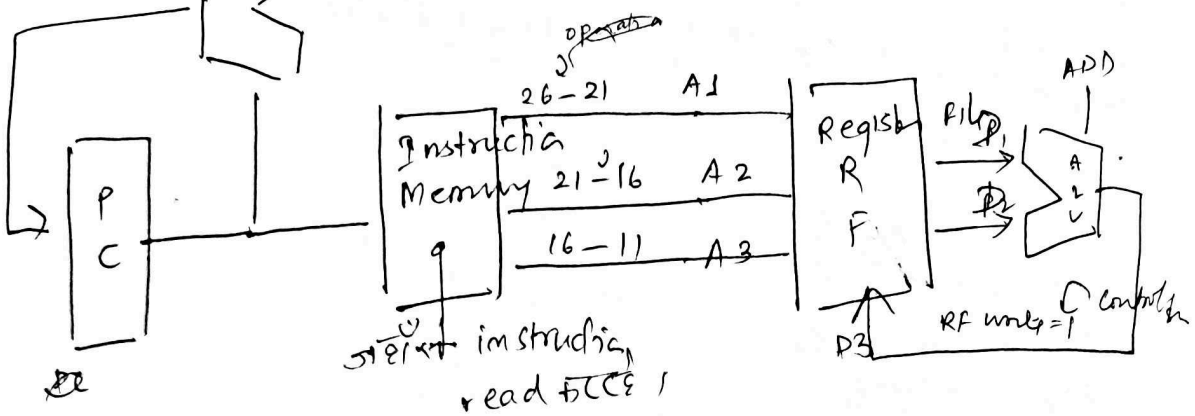
Weighted mean dist.

this ALU updates the value of PC

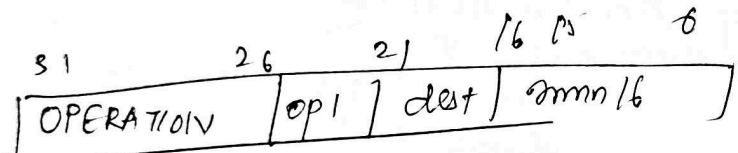
+4

Instruction fetch, read & execute system

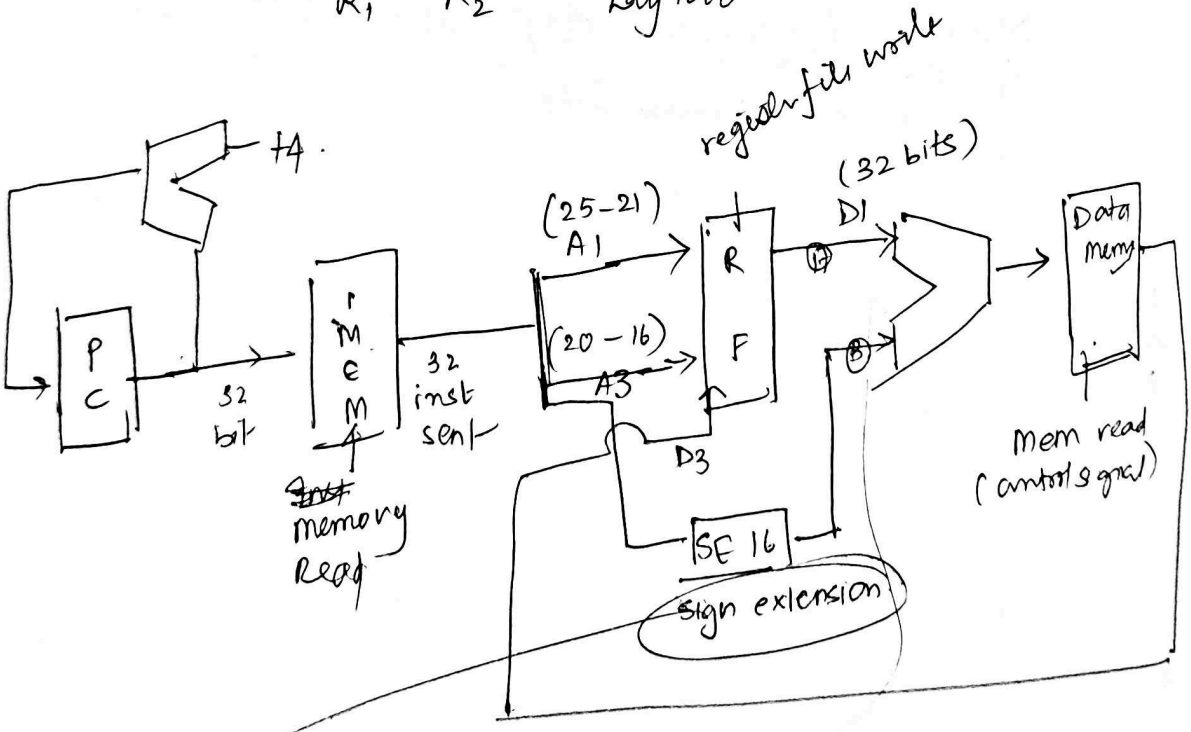
A1, A2, A3 operand 1, 2 & destination



LOAD INSTRUCTION



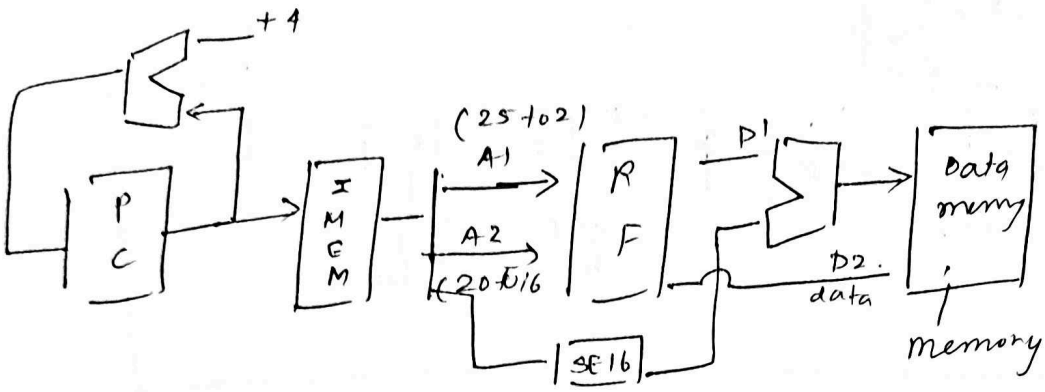
R_1 R_2 Lay 1000



ALU need 32 bits input but required imm 16 only 16 therefore sign-extension used for padding ones at the end.

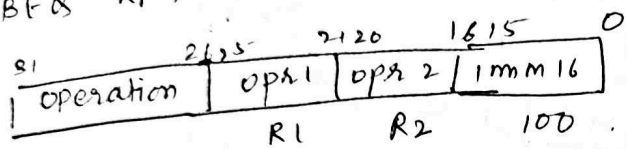
imm 16 need to be padded to be added with 32 bits D1.

STORE INSTRUCTION



Control Flow

BEQ R1, R2, 100



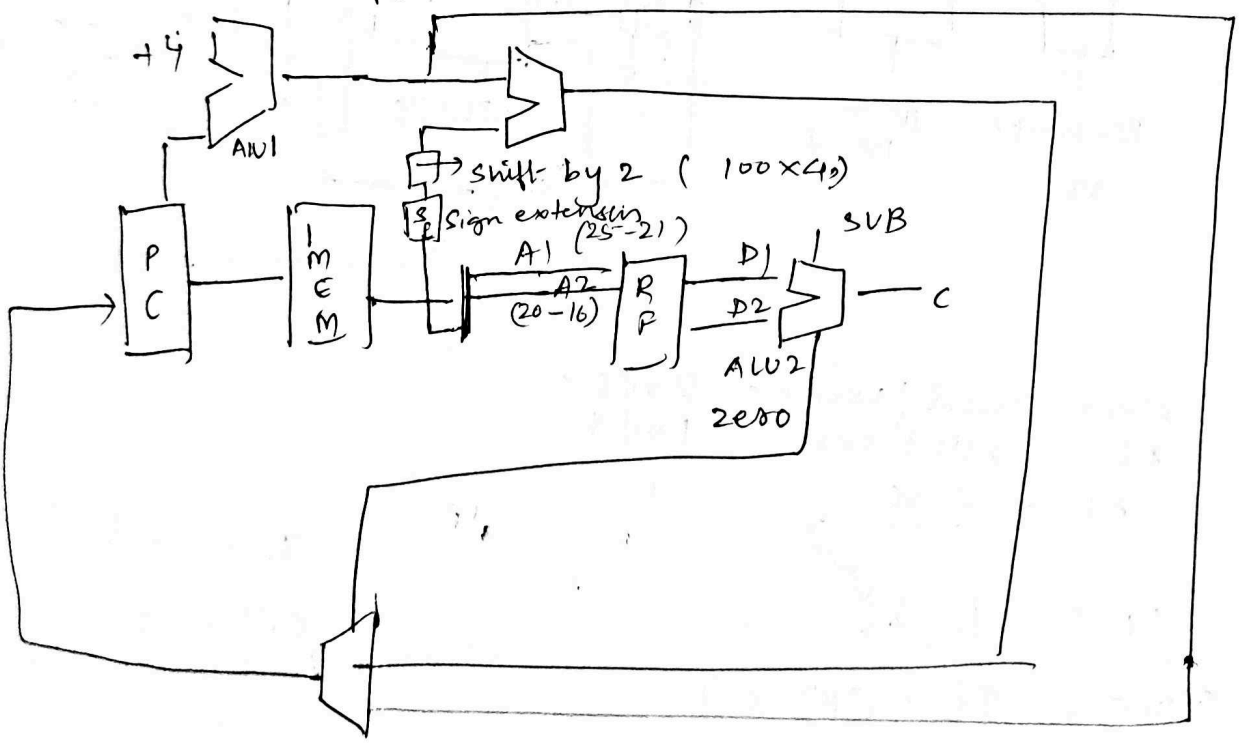
If R1 = R2.

PC = PC + 4 + 100 × 4 ← × 4 because PC changes only in multiples of 4.

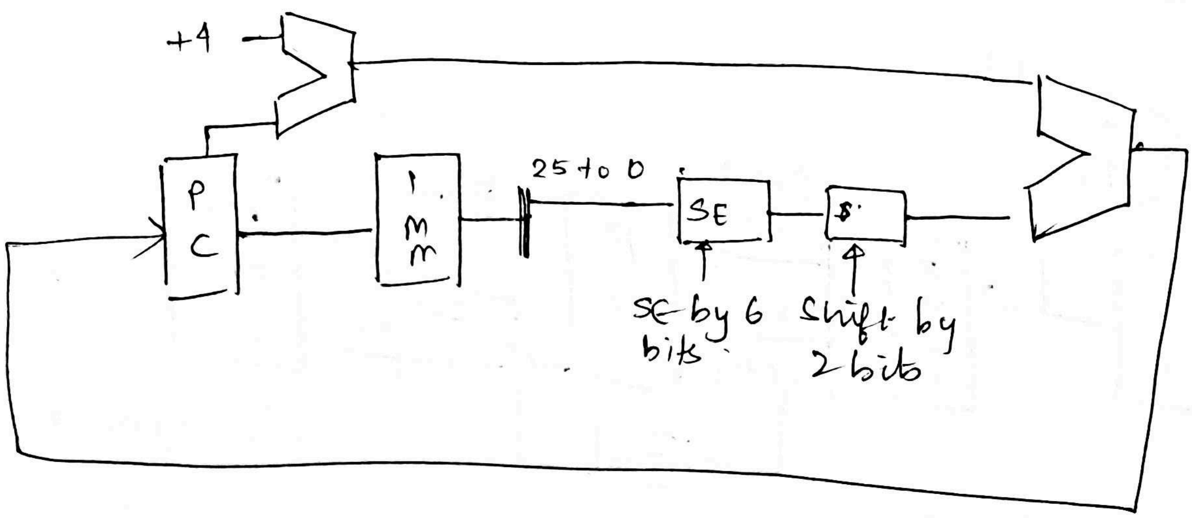
Imm

else

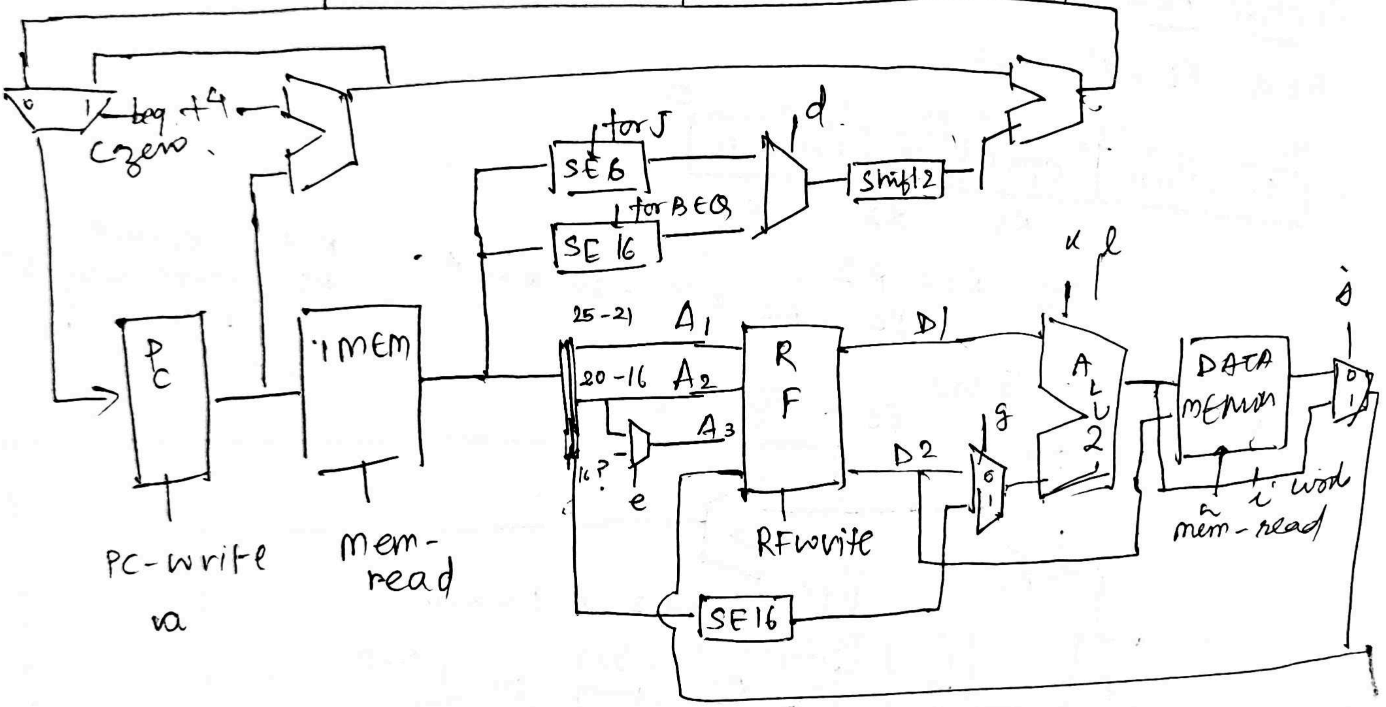
PC = PC + 4



J → PC = PC + 4 + IMM26 × 4



1 মে এটি jump & BEQ implement করা হল



mem read | write = 2 n/s
RF write | read = 1 n/s
ALU = 2 n/s

IC → Inst. count
CPI → Cycle per Inst
T → Clock Cycle

CPI = 1

$\frac{\text{Time}}{\text{Prog}}$ = IC × CPI × T (IC: dynamic)

= 2 + 1 + 2 + 2 + 1 ← Max.

= 8 n/s

Longest-path when we take load instruction.

MIPS-125

for load →

2 ns + 1 ns + 2 ns + 2 + 1 ns = 8 ns

2 ns: 1 memory read
1 ns: R. file
2 ns: ALS
2: data memory
1 ns: write into RF.

frequency = $\frac{1}{8}$ ns = 125 MHz

Since 1 & CPI = 1 is fixed, time depends on IC.

$$\frac{\text{time}}{\text{Prog}} = \boxed{IC} \times C$$

IC: dynamic

dynamic/static instruction

To reduce IC,

$s \rightarrow s'$ replaced by

$s \rightarrow p \rightarrow p' \rightarrow p'' \rightarrow s'$

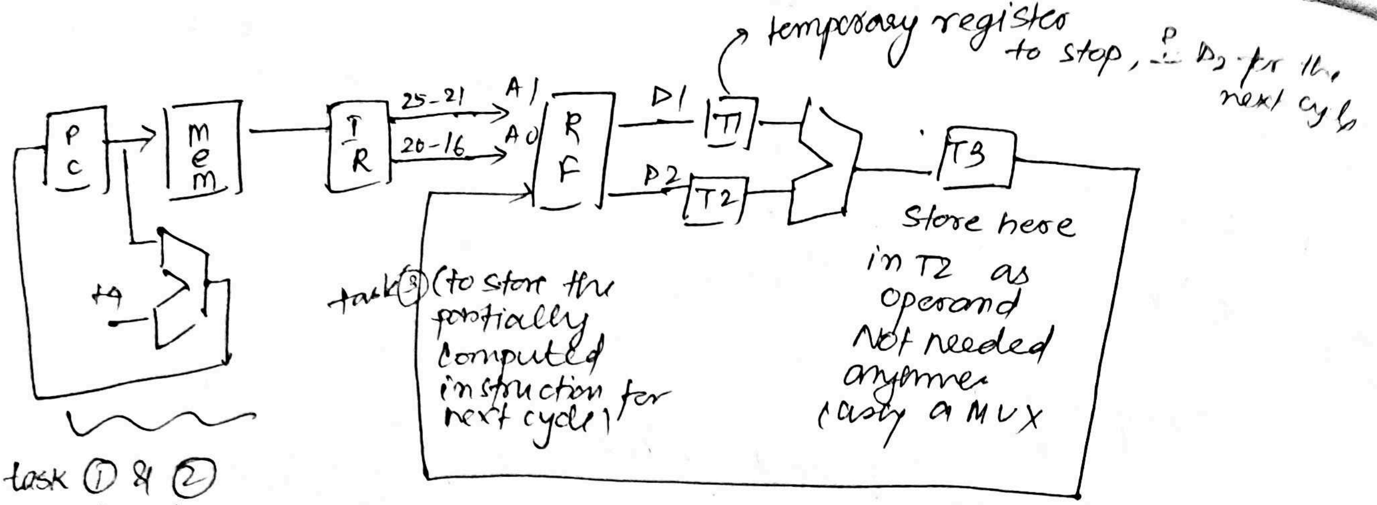
pipeline

Subtasks.

1. Fetching the instruction } housekeeping task. (can be done concurrently, or rest in sequence).
2. Update PC
3. operand read
4. Instruction computation or computation of address of memory (in case of load / store).
5. Read / write memory.
6. Update the state (write to RF)

I & L insl

1. Insl. fetch, Update PC } can be done in parallel
2. Operand
3. Instruction Computation (execute)
4. Write Back to RF.



task ① & ② can be done in cycle

↓

Instead of using the ALU, ALU (right wala) can be used using MUX as that ALU is free in the clock cycle.

दोनों अलग - अलग clock-cycle में use हो रहा है तो करके। PC wala काम और दूसरे भी operation करो।

CPI changes but extra register needed

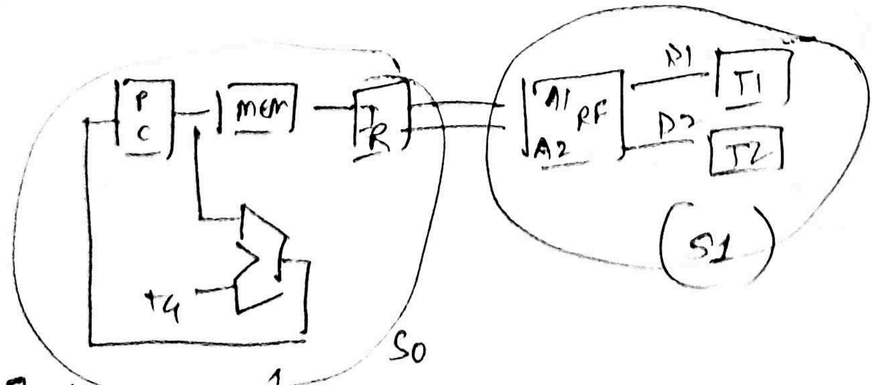
AL → 6 ns
ST → 7 ns
Branch → 5 ns
J → 4 ns.

Resource L ALU
RF
, PC

AW
States.

S_0

PC → memory address, alu A	ADD
Memory D → IR	S0
+4 → alu B	S1
alu C → PC	S2



Instruction inside IR

S1.

$IR_{25\to21} \longrightarrow RF\ A1$

$IR_{20\to16} \longrightarrow RF\ A2$

$RF_{D1} \to T1$

$RF_{D2} \to T2$

} (Temporary register)

(IR cannot be reused currently storing what operation to do)

S2

$T1 \to$ alu A

$T2 \to$ alu B

alu C $\to T3$

can use T1, T2 again

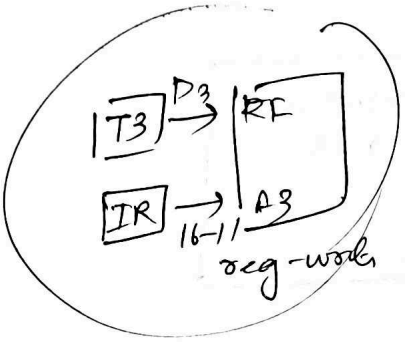
S2

same ALU which was used in S0

S3

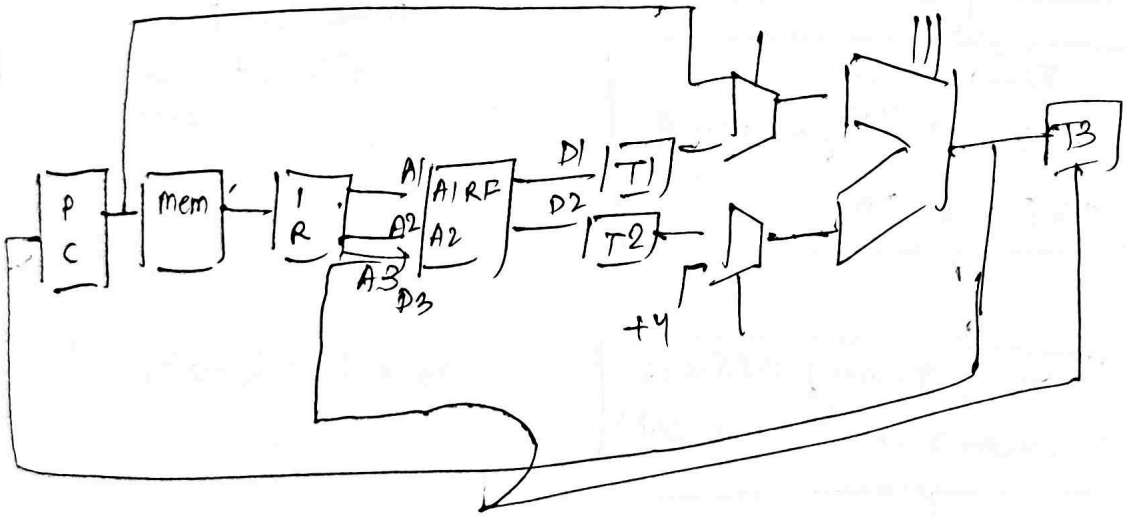
$T3 \to RF\ D3$

$IR_{16-11} \to RF\ A3$



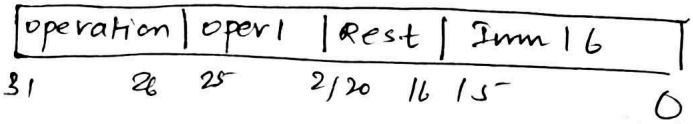
S3

$S_0 \to S_1 \to S_2 \to S_3$

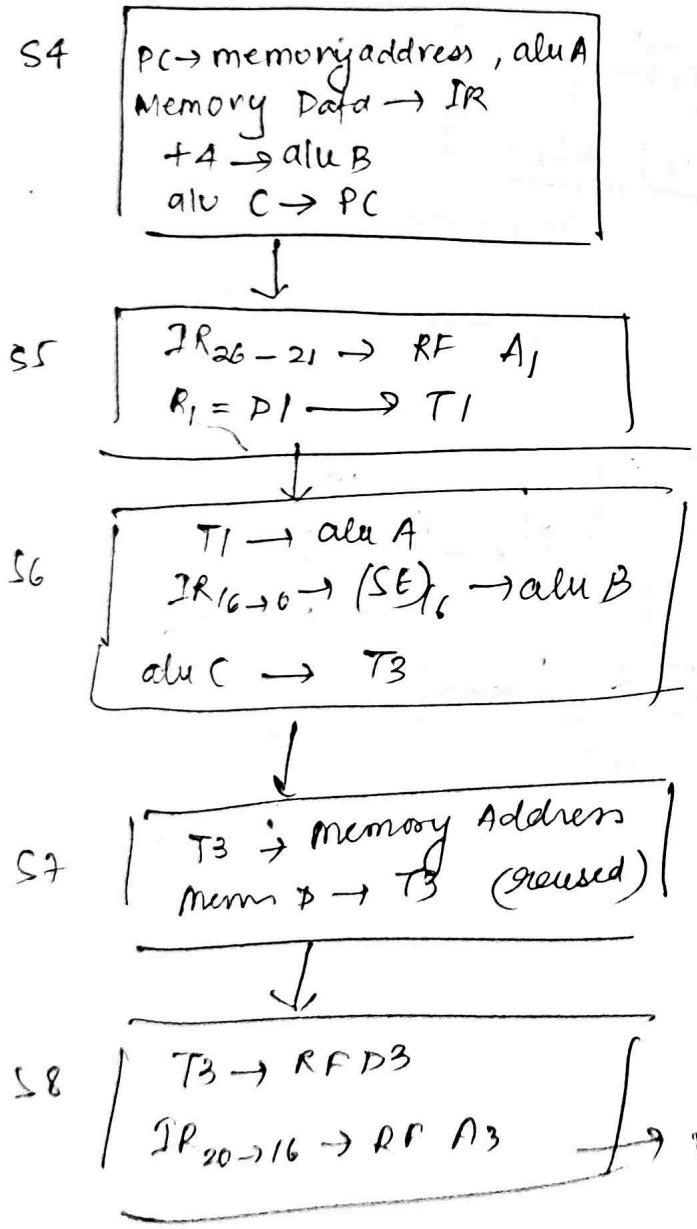


Load instructions

LW R1 (R2) 100



Subtasks : (1) Fetch Instruction
(2) Update PC
(3) Read operand
(4) Compute address of memory
(5) Read memory
(6) Write the result in register



In any clock cycle, read operation at the beginning & write at end

$S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \rightarrow S_8$

STORE INSTRUCTIONS

SW R1 (R2) 100

OPERATION	OPR 1	DEST.	mm16

Subtask

1. Fetch
2. Update
3. Read oper
4. Compute Add
5. Write to memory.

S9: PC → memory address, alu A
Memory D → IR
+4 → alu B
alu C → PC

S10: IR_{25-21} → RF A1
IR_{20-15} → RF A2
RFD1 → T1
RFD2 → T2

S11: T1 → alu A
IR_{16-0} → Set6 → aluB
aluC → T3

★ Doubtful.

S12: T3 → memory Address
T2 → memory Data

S9 – S10 – S11 – S12.

BEQ R_{S1}, R_{S2}, Imm

OPERATION	OPR 1	DEST	Imm 16
31 – 26	25 – 21	20 – 16	15 – 0

If ($R_{S1} == R_{S2}$) then

PC = PC + 4 + Imm16 × 4.

else

PC = PC + 4

Subtasks

1. Fetch Info
2. update PC (PC+4) } housekeeping
3. read operands (R_{S1} & R_{S2})
4. Compare (subtract).
5. If zero, PC = PC + Imm16 × 4

S13

- PC → mem addr., alu A
- +4 → alu B
- alu C → PC.
- Mem D → IR

↓

S14

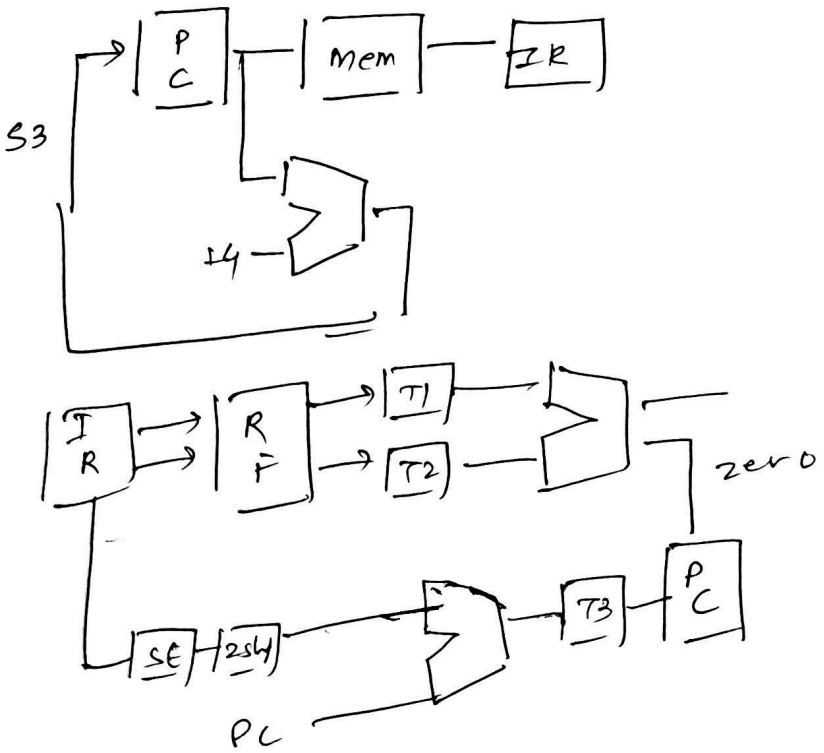
- ~~IR~~ IR_{25-21} → RF A1
- IR_{20-16} → RF A2
- RF D1 → T1
- RF D2 → T2
- ~~alu C → T3~~

↓

S15

- T1 → alu A
- T2 → alu B
- PC → alu 2. A
- IR_{15-0} → SE_{16} → 2-shift → alu2 B
- If (zero == 1) then
  - (alu 2) C → PC
  - change this to T3

CPI = 3 but another ALU needed to shift. Move these 2 instructions to S14 & use same ALU


P
C
mem
IR
S3
+4
I
R
R
F
T1
T2
zero
SE
2s4
T3
P
C
PC