II RISC DESIGN

* Application specific instruction set processor :-
  - Strikes a balance between flexibility of general µP and speed of a circuit.
  - Application-specific :- eg- Fast implementation of Fourier Transform for image processing.

→ RISC
- Small no. of instructions
- Only few addressing modes
- Simple design
- All ~~arithmetic~~ instructions other than register arithmetic, logical and LOAD/STORE instructions must use direct addressing ~~or immediate~~.
- Only Load and Store instructions can access memory
- We can afford to have fixed length encoding.
  eg- If there are 32 registers (5 bits),

OP	OPR1	OPR2	DEST
7	5	5	5

- Arithmetic & logical instructions can also use immediate addressing
  Load/Store can use any addressing mode.

• If 32-bit ~~dr~~ instructions are decided,

Register direct (R)

OP	OPR1	OPR2	DEST	Wasted
6	5	5	5.	

Immediate (I)

OP	OPR1	DEST	16-bit number
6	5	5	

No memory direct in RISC because too many bits required for memory address

Say all registers are ~~32-bit~~ n-bit. n could be 32

- Load/store supports Base + Index and Base + Displacement.

LOAD Rd, (RS1) RS2
≡ Rd ← [RS1 + RS2]
'R-type instruction'

LOAD Rd, (RS1) Imm
≡ Rd ← [RS1 + Imm]
'I-type instruction'
Imm is 16-bit. Max displacement = 65,536

→ 3-types of load/store instructions :- LW :- Load word (32-bits)
LH :- half-word (16-bits)
LB :- byte (8-bits)

→ Change in control flow instructions

We have studied branching on flag values
Other possibility :- No flags at all. #different flavour
Condition is computed by same instruction and decision is taken accordingly

eg Branch if not equal :-

```
bneq RS1, RS2, offset
≡ if (RS1 == RS2) {
        PC = PC+4 }
  else {
        PC = ~~PC + offset*4~~ PC + 4 + offset*4 }
```

one instruction is 4 bytes

Jump 'offset' no. of instructions ahead.

offset is corresponding to next instruction, not current instruction

- Similarly ~~the~~ Branch if equal beq

- Jump
  - J loc ← 26 bit

Design new type of instruction :- J-type

OP	Imm (26-bit)
6	

PC = PC+4+loc*4

- Register direct jumping is also possible :- JR R1 ---- R-type instruction
  PC ← R1

- Calling a subroutine :- 'Jump and link' :- JAL loc
  - Uses J-type instruction.

→ R31 ← PC+4 :- Store location of next instruction in R31. Define R31 as return register.

PC = PC + 4 + loc*4

- All jumping, offsetting is done wrt next instruction.

- Jump and link to register :- JALR R1
  ≡ R31 ← PC + 4
  PC ← R1

→ SLE :- Set if less than equal to

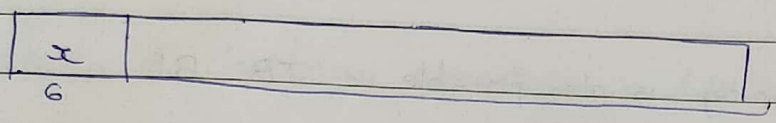
SLE R1 R2 R3
if (R1 <= R2) : R3 = 01
else R3 = 00

SLEI R1 R2 I
if R1 ≤ I ----

Similarly we have SLT, SLTI, SGET, SGETI, SGT, SGTI

- These instructions compensate for lack of flags

→ Structured Instruction format.



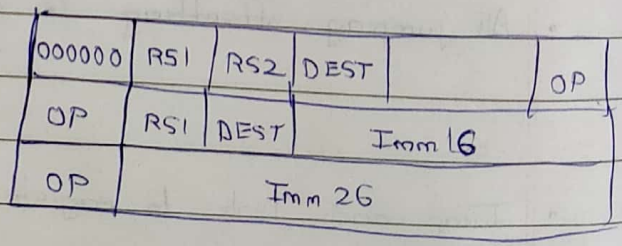
- If x = 000000 → R-type instruction.
  OP is stored as last 6 bits (bits 27 to 32)
  ∴ 64 R-type instructions possible.

~~eg =~~

- If x ≠ 000000 → I or J type instruction
  Total 63 I or J type instructions possible.

4/10

- Add/Sub/NAND/shift :- R-type
  LW, SW, BEQ :- I-type
  J :- J-type

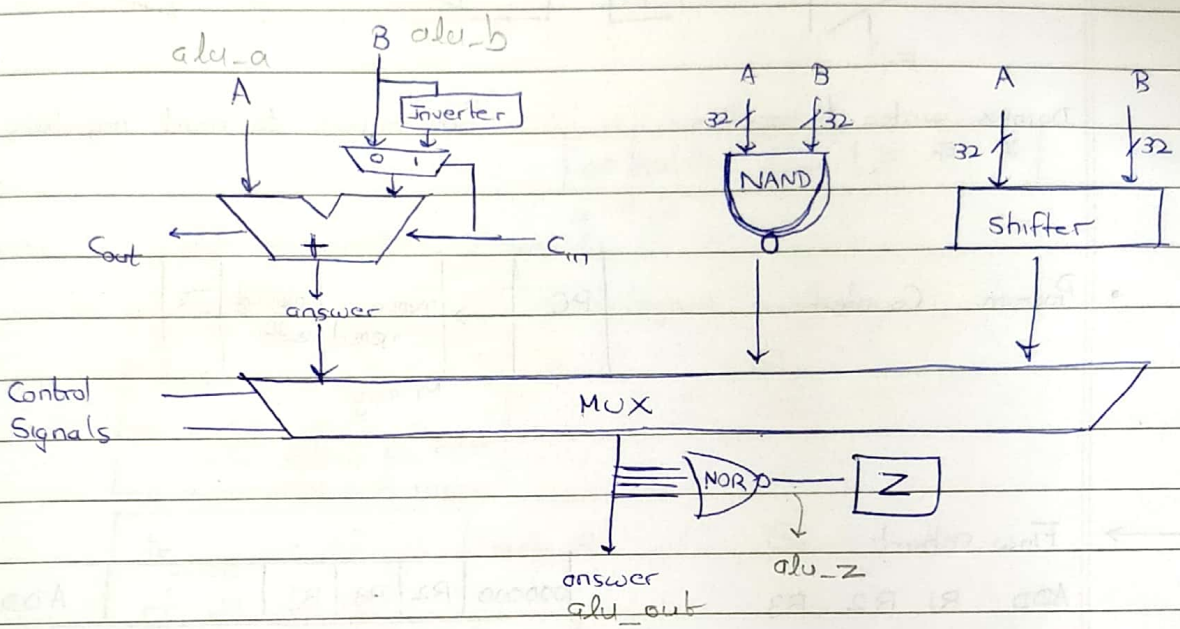


- In RISC, there is almost no possibility of merging HKT with operational tasks, because all instructions do similar stuff and resources are already utilized.
  - Either of HKT or OT can be performed first.
  - We will do HKT first, unlike what we did in CISC
    #different flavour

→ Implementation ← HFC — ISA

→ Minimal Datapath
- 32 registers (programmer-accessible)
- Steering Logic can be point-to-point or broadcast
  - Point-to-point transport # different flavour

• ALU



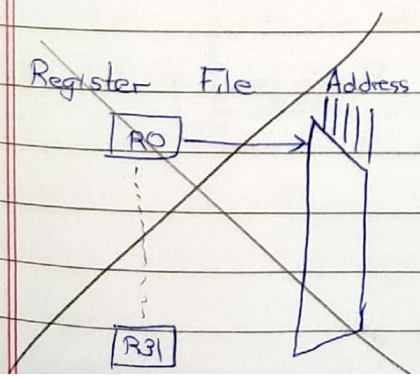
- Add / Subtract :- $C_{in} = 0$:- Add
  $C_{in} = 1$:- Subtract (by 2's complement)

* Add with carry is not supported

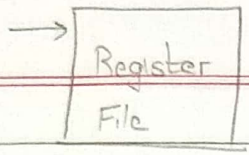
- Shifter :- 5 LSBs of B will tell how much to shift A by.

- NOR :- Checks if output is 0 (⇒ NOR = 1)
  Z bit is NOT visible to programmer

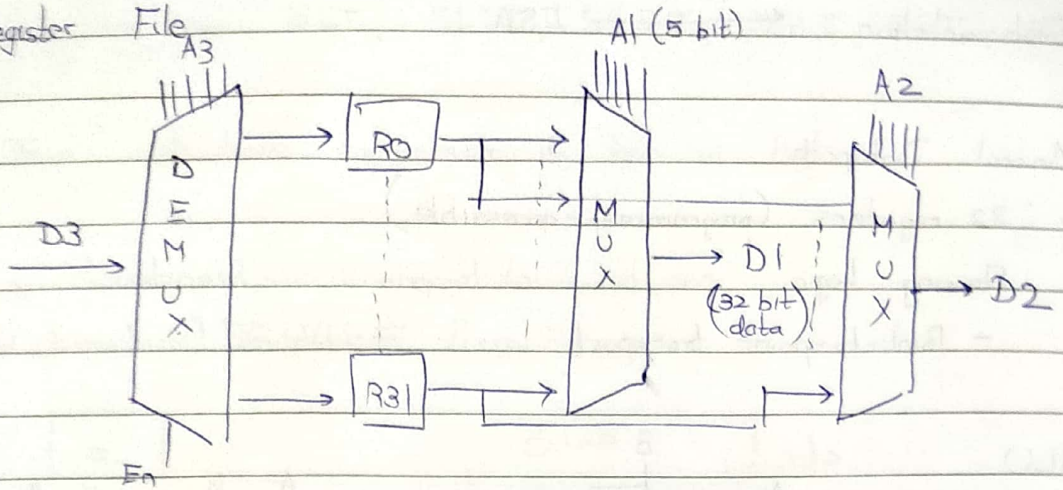
• Register File Address



All 'writing to registers' happens at clock edge



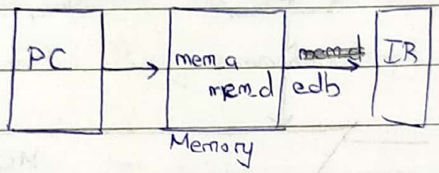
Register File



Demux writes to registers if En = 1

Two muxes to read registers

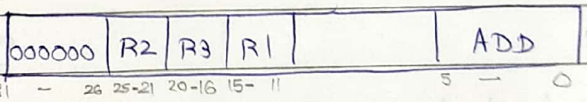
Program Counter



mem_a = a0
mem_d = edb

Flow chart

ADD R1 R2 R3



PC → mem_a/a0/~~edb~~, alu_a
mem_d/edb → ir
+4 → alu_b
alu_out → pc

S1

RD, Add, S1, S2

alu_out is not just connected to T1

In one state,
one use of ALU
or
one use of memory
or
one use of register file
in series

ir_{21-25} → rf_a1
ir_{16-20} → rf_a2
rf_d1 → t1
rf_d2 → t2

S2 Need temporary registers t1 & t2, because we cannot serially use ALU in the same state we read from register file in S2,

t1 → alu_a
t2 → alu_b
alu_out → t1

S3

S3

Parallel use of ALU, RF, memory is allowed.

t1 → rf_d3
ir_{11-15} → rf_a3

S4

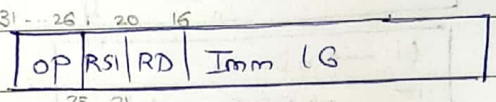
S4

8/10

- Time-consuming tasks (only one possible (serially) in one state) :-
  ALU use, Register File RD/WR, Memory RD/WR.

- Cycle time = ~~Clock~~ max (time for individual states)
  - Hence we do not want to perform time consuming tasks serially in one state.

→ LOAD



B+D :- RD ← [RS1 + Imm16]

Signed number

HKT

PC → mem_a, alu_a mem_d → ir +4 → alu_b alu_out → PC	S5	S1

OT

ir_{21-25} → rf_a1 rf_d1 → t1	Fetch RS	S6	S2

Now, RS is 32-bit ∴ Use sign extender to extend Imm16 to 32 bits ⇒ If MSB is 0, ~~one~~ pre-append sixteen '0's / '1's

t1 → alu_a ir_{0-15} → SE16 → alu_b alu_out → t1	Find RS1 + Imm16	S7	S5
t1 → mem_a mem_d → t1	Fetch operand	S8	S6
t1 → rf_d3 ir_{16-20} → rf_a3	Write to RD	S9	S4

→ STORE

HKT

S10 | S1

PC → mem_a, alu_a
+4 → alu_b
mem_d → ir
alu_out → PC

S2

OT

S11 | Store RS in t1 and RD in t2

ir_{21-25} → rf_a1
ir_{20-16} → rf_a2
rf_d1 → t1
rf_d2 → t2

S12 | S5

t1 → alu_a
ir_{0-15} → SE16 → alu_b
alu_out → t1

S13 | S7

t1 → mem_a
t2 → mem_d

→ BEQ R1, R2, Imm16

if (R1 == R2) { PC = PC + 4 + 4 * Imm16 }
else { PC = PC + 4 }

HKT

S14 | Stores PC+4 in PC | S1

OT

S15 | S2

ir_{21-25} → rf_a1
ir_{20-16} → rf_a2
rf_d1 → t1
rf_d2 → t2

S16 | S8

t1 → alu_a
t2 → alu_b
alu_z → Z

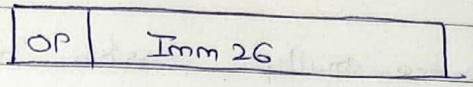
S17 | S9

PC → alu_a
ir_{0-15} → SE16 → 25 → alu_b
If (z=1) { alu_out → PC }

→ Shift left twice to multiply by 4

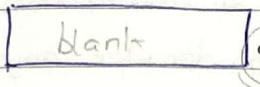
Use some concoction of MUX to execute if statements

→ JUMP

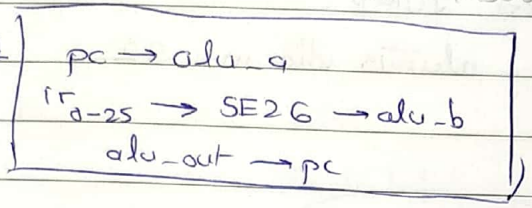


HKT

OT



One state is blank for time to decode the instruction. In other instructions, decoding was done in parallel



9/10

→ Level 2 flowchart :- Memory Access, ALU operation, current state, next state.

- Merge ~~tasks~~ states doing similar tasks

Note that second state of LOAD, is a subset of second state of ADD and JUMP which is same as second states of STORE and BEQ. ~~and JUMP~~

Merge all these (S2, S6 .....)

Even for LOAD, use same second state as ADD

- States 1 and 2 have become common to all instructions
- The last state of each instruction has S1 as next state, which has S2 as next state.

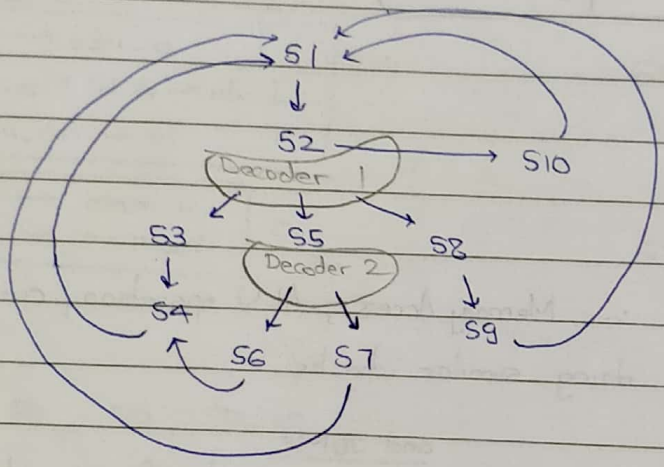
After S2, next state is different.

- We can merge S4 and S9, and choose between ir_{15-11} and ir_{16-20} using a MUX, at the cost of an additional control signal.
- Merge S12 in STORE and S7 in LOAD.

- Wherever there are multiple possible 'next states' (S2, S5, etc), we require a decoder that decodes current instruction

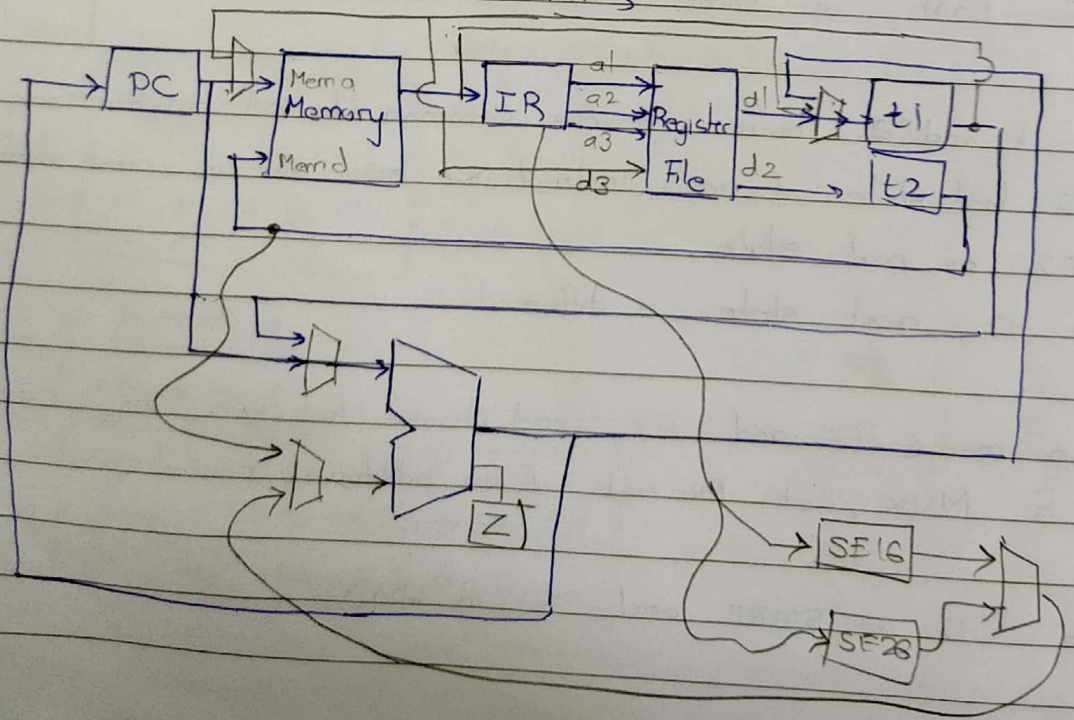
- We can even merge S10 and S2, with a modification that pc will be updated only if the instruction decoder says that current instruction is indeed JUMP.
This can be done because alu is idle in S2

S3 - R type
S5 - I type
S8 - J type



We will use a non-microcode style FSM #different flavour

→ Point to point communication links



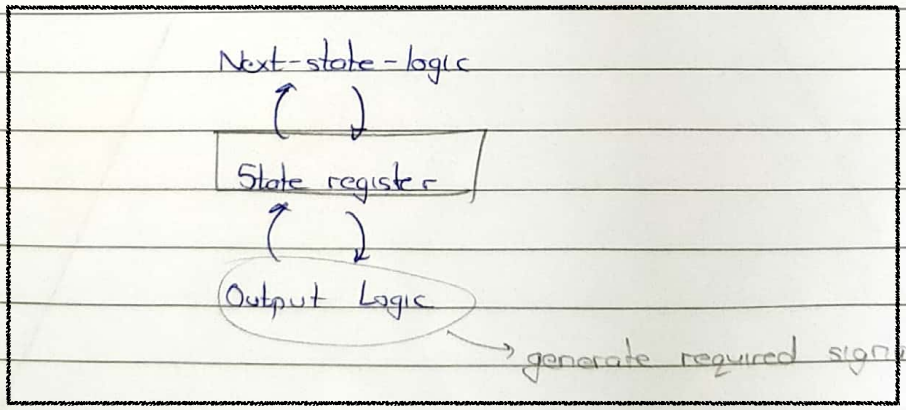
Basically → Go over all states. Draw the links that you need. Use MUXes at whichever inputs you need.

11/10

→ Control signals

1. Memory :- $\overline{RD}$, $\overline{WR}$ (Active low)
2. Register File :- WR
3. IR :- Enable bit
4. MUX at memory input :- 1 bit
5. " " T1 " :- 2 bits
6. " " ALU input 1 :- 1 bit
7. 2 :- 1 bit
8. SE output :- 1 bit
9. Choose ALU operation :- 2 bits
10. PC :- Enable bit ?
11. Updating Z flag :- 1 bit

- 10 states encoded in 4 bits



→ Next state logic

```
process (current state, decoder 1, decoder 2)
    when (S#1)
        next state = S2
    when (S2)
        next state = decoder 1
        :
        :
    when (others)        to ensure no sequential circuits. (tutorial)
        next state = S1)
```

→ Output Logic

```
process (                    )
    when (S#1)
        // Specify ALL control signals.
    :
    :
```

→ 3rd process :- To instantiate flip flops.

15/10

* Quiz

- Average CPI = Weighted mean of CPIs of different type of instructions

  where weight = frequency of occurrence of each type of instructions

* Say, memory R/W takes 3 ns
ALU operation 2.5 ns
RF= R/W 2 ns
Then, clock cycle chosen should be at least 3 ns.

→ Single cycle implementation
- Wasn't possible in CISC (complex & variable-length instructions)
- Possible in RISC, because state transitions are simple and similar.

$S \rightarrow T \rightarrow T' \rightarrow T'' \rightarrow S' \implies S \rightarrow S'$

• Single cycle implementation :- CPI = 1

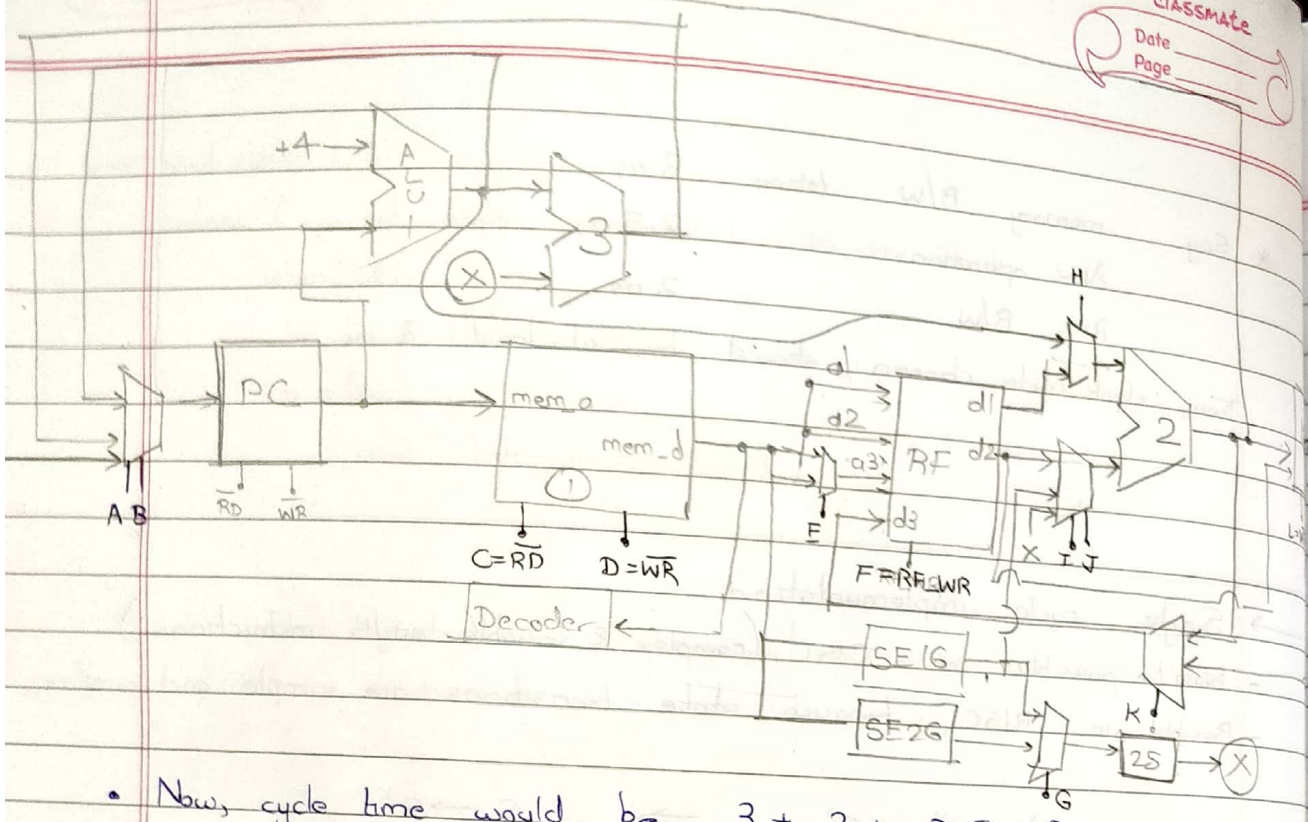
eg Arithmetic/Logical instructions

pc → alu1_a, mem_a
+4 → alu1_b
alu1_out → pc
mem_d_{21-25} → rf_a1
mem_d_{16-20} → rf_a2
rf_d1 → alu2_a
rf_d2 → alu2_b
alu2_out → rf_d3
mem_d_{11-15} → rf_a3

When we do entire instruction in single cycle, there cannot be storage in between because writing happens at clock edge

• Controller is no longer an FSM. It is a combinational logic that generates appropriate control signals based on value of opcode.

a3 inputs :- 11-15 OR 16-20



- Now, cycle time would be $3 + 2 + 2.5 + 2 = 9.5$ ns

Mem + alu read → RF read → alu_2 → RF write

- All signals will stabilize by 9.5 ns
  - PC and RF will be updated at clock edge at the end of 9.5 ns.

eg LOAD

pc → alu1_a, mem1_a
+4 → alu1_b
alu1_out → pc
mem1_d_{21-25} → rf_a1
mem1_d_{0-15} → SE16 → alu2_b
rf_d1 → alu2_a
alu2_out → mem2_a
mem2_d → rf_d3
mem1_d_{16-20} → rf_a3

Mem1 for code
Mem2 for data

ALU3 is adder only

eg STORE

pc → alu1_a, mem1_a

+4 → alu1_b

alu1_out → pc

mem1_d_{21-25} → rf_a1

mem1_d_{0-15} → SE16 → alu2_b

rf_d1 → alu2_a

alu2_out → mem2_a

mem1_d_{16-20} → rf_a2

rf_d2 → mem2_d

~~* Control~~

eg BEQ R1, R2, Imm16

pc → mem1_a, alu2_a

+4 → alu1_b

mem1_d_{20-25} → rf_a1

mem1_d_{16-20} → rf_a2

rf_d1 → alu2_a

rf_d2 → alu2_b

alu1_out → alu3_~~out~~ a

mem1_d_{0-15} → SE16 → ~~alu3_b~~ 2S → alu3_b

if(alu2_z) { alu3_out → pc }

else { alu1_out → pc }

P.T.O

eg JUMP

pc → meml_a, alu2_a

+4 → alu1_b

alu1_out → alu2_a.

meml_d_{0-25} → SE 26 → 25 → alu2_b

alu2_out → pc.

→ Control Signals.

	A	B	C	D	E	F	G	H	I	J	K	L	M
000000 (R type)	0	0	0	1	1		0	1					
⋮													

Simplify ~~to find~~

eg D is always 1

→ Cycle Time

- LOAD takes most time (costliest path)

3 + 2 + 2.5 + 3 + 2 ≈ 12.5 ns

mem1 rf alu mem2 rf

→ Performance = # of instructions × $\underbrace{CPI}_{=1}$ × Cycle time

- In single cycle implementation, performance does not depend on fractions of different type of instructions.

- Performance :- Arithmetic / Logical :- 50%
  Load :- 20%
  Store :- 10%
  ~~Branch~~ BEq :- 15
  Jump :- 5%

  For multicycle, Performance = N × ~~4.45~~ 4.15 × 3ns
  = 12.45
  single cycle = 12.5

- Depending on fractions of instruction types, either of single cycle or multicycle implementation.
- In spite of fewer registers, single-cycle is more expensive overall in terms of hardware.
- Single cycle :- Lowest possible CPI (1)
  Multi cycle cycle time
  - Possible to strike a balance :- eg- 2 cycle implementation.

→ Pipelining.

- Begin next instruction before previous is over, using the idle resources.
- In RISC, subtasks in every instruction are very similar - fetch instruction, fetch operands, alu operation, etc.

- It is possible to get least possible cycle time (like multicycle)
  - If we assume all instructions to be independent, then we can do pipelining and then we will get cycle time equal to max{sub-task time}

→ Subtasks

1. R-type :- Instruction fetch, Instruction Decode / Operand Read, Execute, Write Back
2. LOAD :- IF, ID/OR, EX, Memory Read, WB
3. Store :- IF, ID/OR, EX, Mem
4. BEQ :- IF, ID/OR, EX
5. JUMP :- IF, ID

exhaustive set of sub-tasks

- First 5 (or 4) cycles, no instruction will be completed.
  After that, one instruction will be completed every cycle.
  (for independent instructions and if instructions are consecutive from top to bottom (no control flow changes in between))

→ Performance = $N \times \underbrace{1}_{CPI} \times \underbrace{3ns}_{\text{Cycle time}}$