Chapter 4
The Processor

Introduction

* CPU performance factors → software for the hardware
  - → instruction count (ISA & compiler)
  - → CPI & cycle time (CPU hardware)

datapath
↳ how does an instruction affect parts of the hardware?

* simplified MIPS instructions (subset)
  - memory reference: lw, sw
  - arithmetic/logical: add, sub, and, ori, slt
  - control transfer: beq, j

→ pipelined
→ non-pipelined
↳ sequential (one after another)

* MIPS → any instruction boils down to 3 formats.

Instruction Formats

→ shift amount

R → op, rs, rt, rd, shamt, funct → arithmetic/logical op where operands are always registers

I → op, rs, rt, addr/~~const~~ immediate → transfer, immediate branch, memory

J → op, target addr

MIPS has only one type of branching
↳ equality
→ bne, beq

R
```
        rd  rs  rt
add     $t0, $t1, $t2
```
$t0 = $t1 + $t2

I
```
lw   $t0, 80($a0)
sw   $t0, 80($a0)
```
load
→ base address
→ offset/immediate
→ base } memory
→ offset
save

generally MIPS operations have destination on the left.
→ sw is an exception.

```
lw   $t0, 80($a0)
     rt   rs

sw   $t0, 80($a0)
     rs   rt
     rt   rs
```

ori $t0, $t1, 10
rt rs immediate

beq $t0, $t1, L10
rt rs

check equality using subtraction
↳ Z=1 → equal
Z=0 → not equal

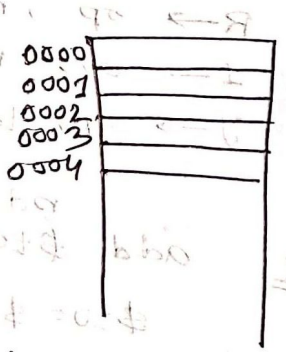
* Is the branch taken?
→ check if Z is 1 → need to check the instruction
* if the instruction is beq & Z=1 → branch is taken
other ways can also cause Z=1, even R-format ones.

means → jump → updates PC
↳ shift the immediate 2 bits.

→ Why 4 bits jump?
2 types of memory
→ flat memory → consecutive memory
→ band memory → blocks & breaks.

memory → each location has an address

→ only possible to take 4 byte from hardware memory → hardware limitation



* gets 4 location values at once.
→ can't get immediate ones.

get 0004 → returns

abstraction → 0001 00 → 1
0010 00 → 2
} cut off 2 bits in instruction

0004
0005
0006
0007
} datab

while operating in hardware → shift by 2.

paragraph → must start from start
↳ can't enter in the middle.

↳ index the paragraphs
↳ shift 2 bits to get the address.

J

address → 26 bit → shift by 2 → to get actual address.
↳ paragraph address

Combinational Elements

clock rate → ...GHz
↳ cycle length → ns
→ hardware delay → 1ns
} big problem.

2 types of ckt
→ combinational
→ sequential

difference?
sequential has states.
→ clock ?⊛

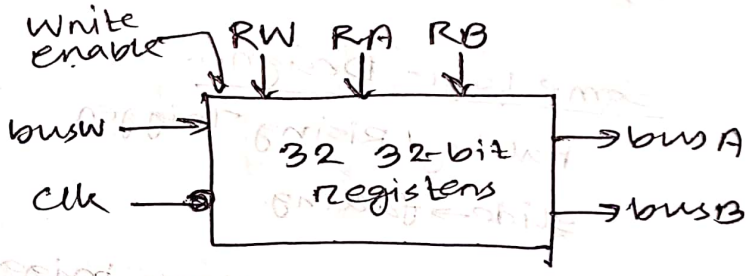
no clocks in combinational

necessity of the clock

Storage Element: Reg File

store 32 regs → register file ⊛ in MIPS terms
↳ distinguish from file

memory snap



⊛ read is a combinational operation → no clocks needed (although delay occurs)

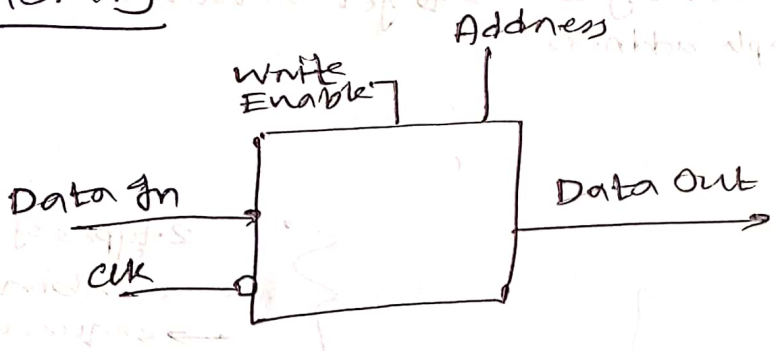
⊛ write is a sequential operation.

↳ can read 2 & write 1 in one cycle

RA, RB → read → put on busB
RW → write
busA, busB → output
busW → write input → always has some value.
write enable → controls write
↳ 1 → registers are valid → execute write.

put on busA

Memory



not possible ☒ to have r/w in same clock cycle

→ 1 address

R → read from 2, write to 1

memory → lw, sw → separate → not possible together

↳ supports one address at a time

same clock cycle → r/w → concurrency issues.

↳ shared memory access

→ all cores use same hard disk.

Some Logic Design...

Falling/Rising Triggen

slide → falling.

* clk-to-Q → clock triggen to valid output time

delay

combinational time → longest time of all parts in combinational part.

F/F → needs setup time

↳ data must be present before clk triggen.

after clk triggen → some delay → then ops.

quantz → clock skew → one physical clock

cycle time = clk-to-Q + Longest Delay + Setup + Clock Skew

different chips are at different distances from the clock.
↳ different clock delays
↳ wire resistance

consider all these

Datapath: IF Unit

tasks of processor
- fetch
- decode
- execute

fetch → bring instruction from memory using instruction address.

Instruction Memory & Data Memory

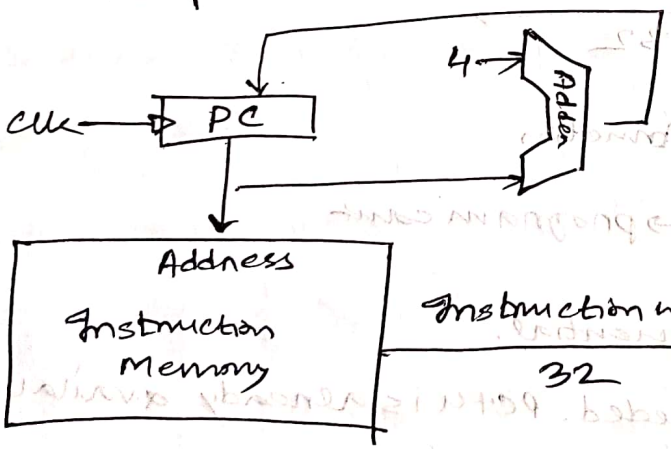
↓

Read only

⊛ if code writes in instruction memory → virus (must be)

address to memory → get instruction

↳ update PC.



⊛' shouldn't always update PC
↳ not always PC+4
↳ branch / jump may happen

one of these 3 addresses will be connect

04/11/2024

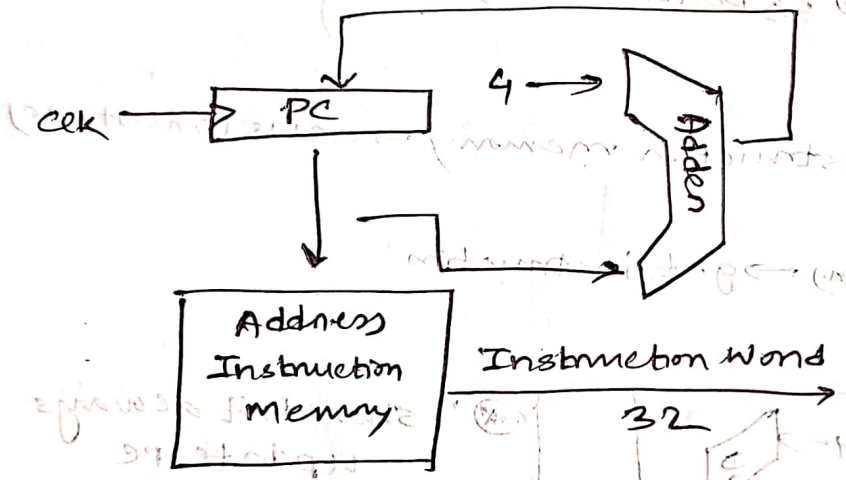
CSE-209

combinatorial vs sequential → clock

memory vs register
↳either r or w → 2r, 1w

cycle time = clk-to-q + ...

Datapath : Instruction Fetching (IF unit)



↳ o/p → the 32bit instruction

PC → dedicated register → program counter

* write requires clk → sequential.
* assume ~~to~~ clk isn't needed. PC is already available
should it be written? → NO

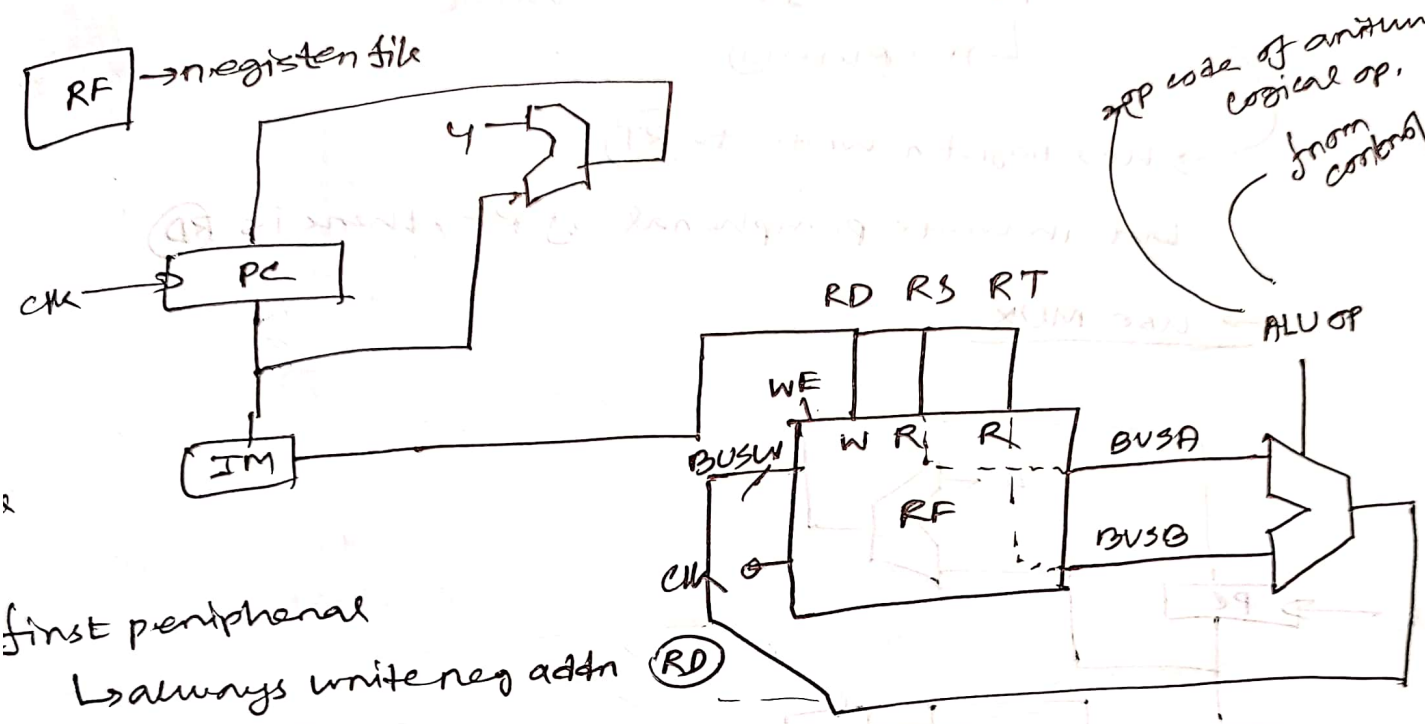
Add RTL

add rd, rs, rt

↳ R-Format
↳ arithmetic / logical op with both operands from reg

OP=0	rs	rt	rd	shamt	funct

rd rs rt
add $t0, $t1, $t2
$t0 = $t1 + $t2



first peripheral
↳ always write reg addr

next 2 → read reg → RS, RT

BUSA, BUSB → O/P
↓ ↓
RS RT
→ goes to ALU

data input → BUSW → from ALU O/P

WE → 1, clk → writes
↳ doesn't always write
→ ensures if the RD, BUSW → can/is writable/is in writable config
→ from control.

BUS → any data path one after another in a line.

I-type

memory transfer, immediate, branch

ORI $t1, $t2, 10

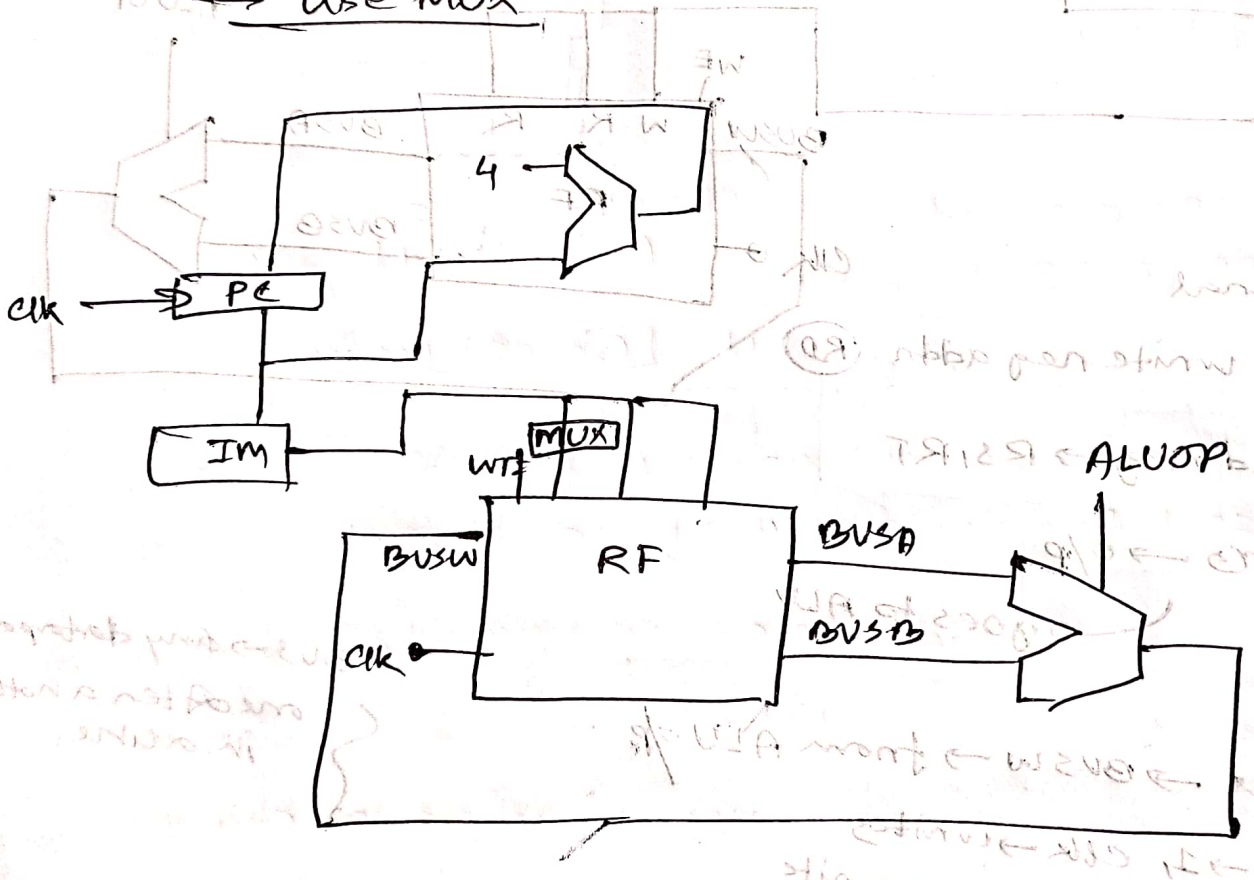
RT RS Immediate value

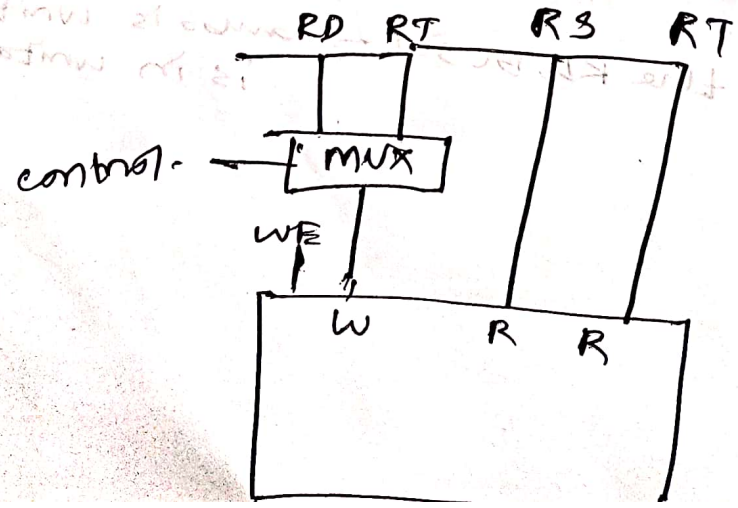
↳ no RD in (I)

↳ has register write to (RT)

but in write peripheral of RF, there is (RD)

→ use MUX





get immediate from IM → last 16 bit

ALU → 32 bit

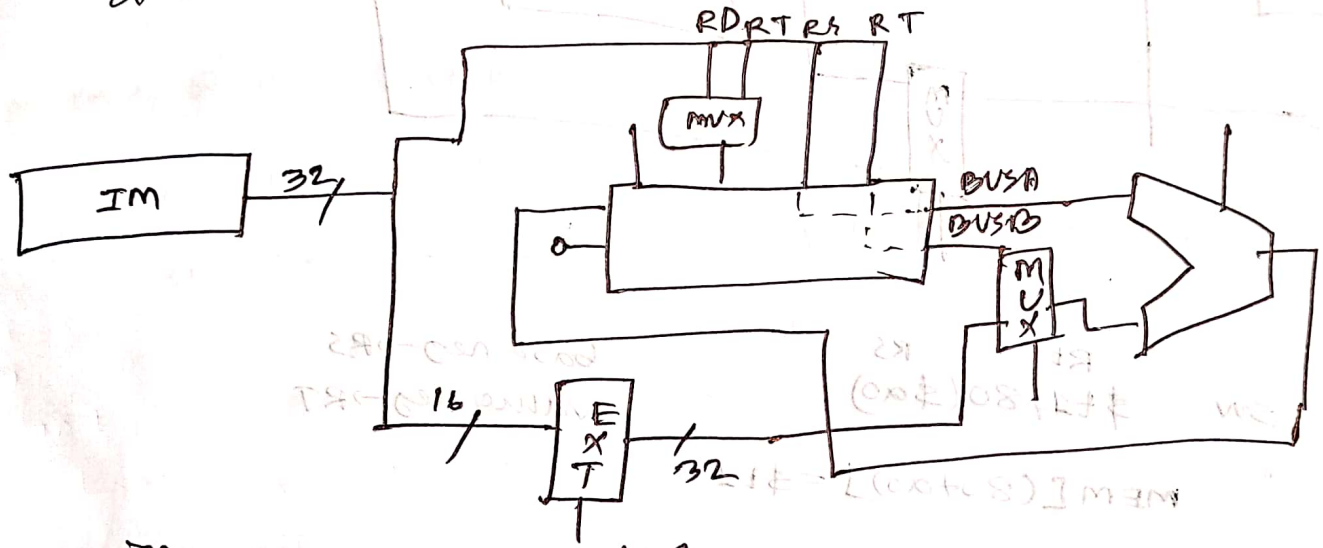
2 ways

fill with 16th bit → sign extension → sign 0 → first 16 → 0 else 1

→ 0 extension → first 16 → 0

logical → first 16 → meaningless → 0 extension

arithmetic → signed → signed extension.



extension module.

o/p of ALU goes to (RT)

Load Rt RS

lw $t1, 80($a0)

$t1 = MEM [80 + $a0]

has reg write to Rt already in write peripheral

Addition o/p not final o/p

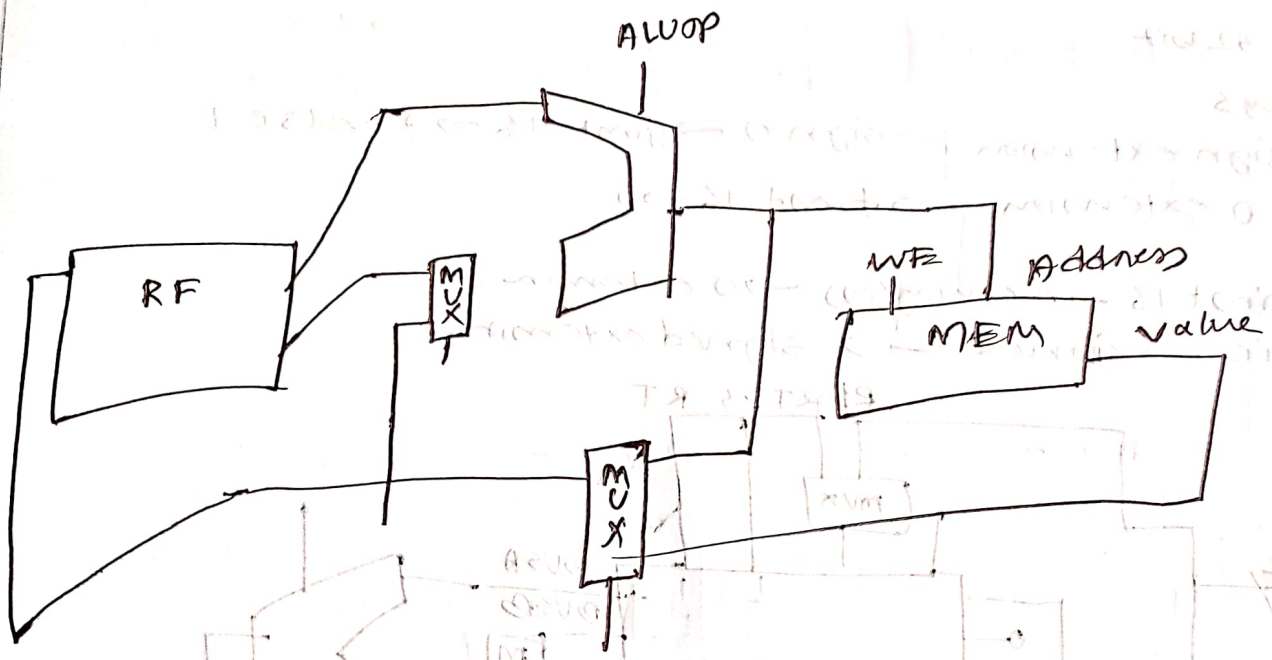
↳ addn → goes to memory

ALU → addition of immediate & RS

↳ already done

signed extension

Cohere.



RT RS

SW $t1, 80($a0)

base reg → RS

value reg → RT

MEM[(80+a0)] = $t1

no register write here → memory write

→ nothing to change in W peripheral of RF

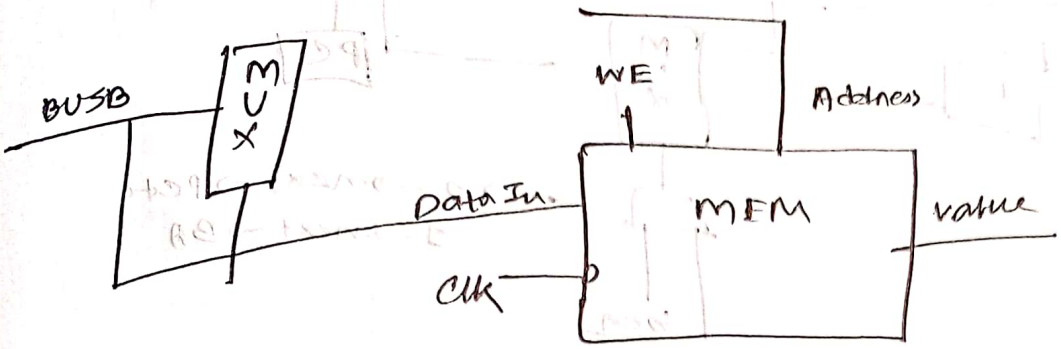
ALU → RS + Immediate → already there

↓

sign extended

→ o/p → goes to MEM address.

→ RT from BUS B **before MUX**



⊗ swap RS, RT → need 2 more MUX → instruction bits of RS/RT are fixed.

in BUSA, BUSB

beq $t1, $t2, 100 → according to instruction format

comparison → use sub

PC → PC+4

update

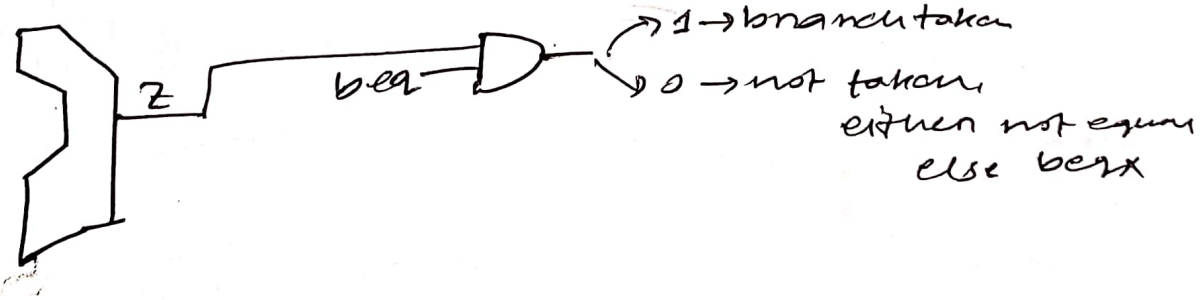
→ no reg write

→ ALU → sub RS ~~—~~ RT

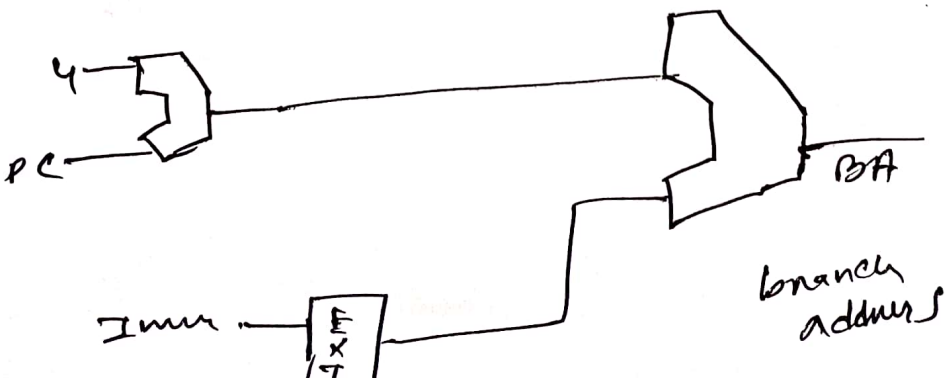
mux already

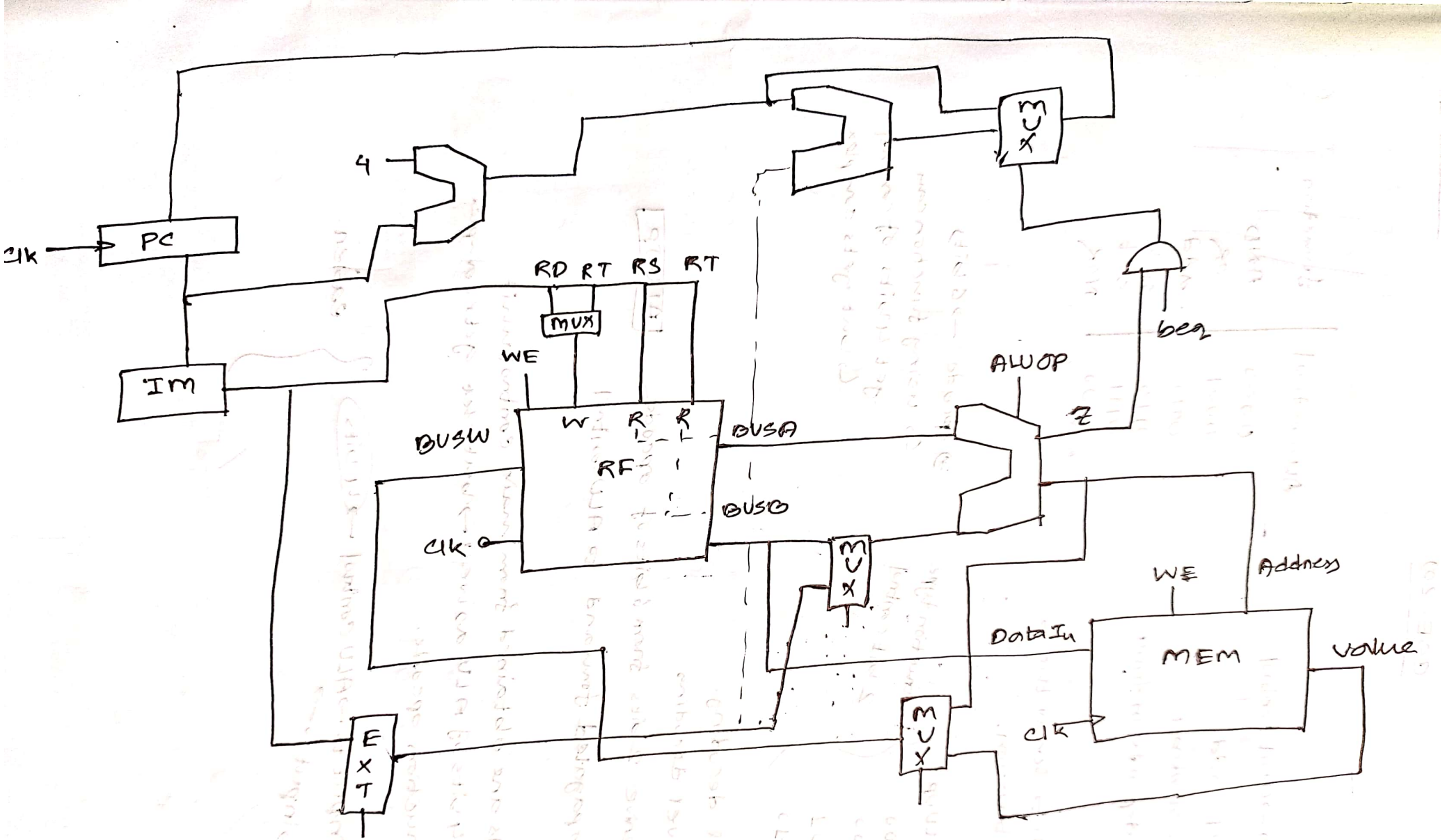
} ALU op not needed

→ check (Z) flag & check if operation was beq.



→ address PC+4, immediate




clk
PC
4
MUX
IM
RD RT RS RT
MUX
WE
W R R
RF
BUSW
BUSA
BUSB
clk
ALUOP
Z
beq
MUX
WE
Address
DataIn
MEM
Value
clk
MUX
EXT
MUX

Full Datapath

ALU Operation → 4 bit control

load/store : F = add
branch : F = subtract
R-type : F depends on funct

ALU control	function
0000	AND
0001	OR
0010	add
0110	sub
0111	slt
1100	NOR

→ ALU only needs to know the op not the instruction.

opcode	ALUOp
lw	00
sw	00
beq	01
R-type	10

} instruction type 2 bit control

* opcode → 6 bits
  ↳ using function can get 4 bits of control
  ↳ but gets complex

use 2-level decoding.
multilevel decoding
→ derive 2 bits from 6 bits of opcode — ALUOP
propagated forward to ALU control

→ all controls are obtained from main control unit.
→ can get 4 bits of ALU as well → but ckt gets complex
→ use instruction opcode.

(2 bits) ALUOp → input to → ALU control → (4 bits) } easier.

funct → input to →
(6 bits)

ALU control fixed → MIPS Hardware.

ALU OP	funct	ALU control
00	xxxxxx	0010
00	xxxxxx	0010
01	xxxxxx	0110
10		

(add) funct not needed

(sub)

(11) → not used

	Op1	Op2	F5	F4	F3	F2	F1	F0	Op
	0	0	———						0010
01 →	x	1	———						0110
10	1	x	x	x	0	0	0	0	0010
	1	x	x	x	0	0	1	0	0110
	1	x	x	x	0	1	0	0	0000
	1	x	x	x	0	1	0	1	0001
	1	x	x	x	1	0	1	0	0111

F0 not necessary for the subset of instn.

The Main Control Unit

→ R-type: opcode: 0
→ Load: " : 35
→ store: " : 43
→ branch: " : 4

Datapath with control

8 controls → ALU OP → 2 bits
total (9) bits

control unit → input → only opcode (31:26)

deassert 0/1 } or vice-versa
assert 1/0 }

disable

enable the control

enable high/low

PCSnc control can't be used directly. → AND with Zero

① RegDst → 0 → RD RT
   ↳ 1 → RD

② Branch → 0/1 → from opcode [if instn is beq]

③ MemRead → not needed

Memory Access → costly
↳ slower than register → close proximity
↳ high performance

read memory always → ops become slow.
→ read only when necessary → MemRead
assert → performance

④ MemtoReg → Mem o/p on ALU o/p

⑤ MemWrite → write enable MEM

⑥ ALUSnc → RT or Imm.

⑦ RegWrite → write enable RF

⑧ PCSnc → 0 → next addn; PC+4
1 → " ; BA

R-Type Instruction

Write reg → RD → RegDst : 1
Imm → not needed.
funct → ALU control
ALUOp → ALU op

ALU → i/p → RS
↳ RT → ALUSnc : 0

Z → not needed

MemtoReg → 0 → ALU Op → valid

Load

ReadReg 1 → RS
ReadReg 2 → X
WriteReg → RT
RegDst → 0

Imm → sign extend
↳ ALUSrc → MUX
funct → Don't care
control from ALU op.

ALU i/p → RS, Imm
MemRead → 1
MemtoReg → 1 → o/p → Mem.

Beq

RS, ~~RT~~ RR2, WR → X
Imm
WriteEnable → ~~deassert~~
ALUSrc → Imm
Imm → shift left 2
↳ ~~A~~ MUX → PC+4, Imm

(Fig)

Control Values

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp2
R-format	1	0	0	1	0	0	0	1	0
lw									
sw									
beq									

Implementing Jumps

2	addn.
31:26	25:0

Reg write → X
ALU op → X
ALU src → X
no memory func.

only change → instr fetching.

take last 26 bits → shift left 2 → second MUX b

append first 4 bits of original PC at front

concatenation unit

prev addn.

PC+4

MUX → i/p → Jump addn
b → BA/PC+4
→ sequential next instn.

10 controls additional for jump.

MIPS Pipeline

5 stages, one stepp per stage

IF → 200ps
ID → 100ps
EX → 200ps
MEM → 200ps
WB → 100ps

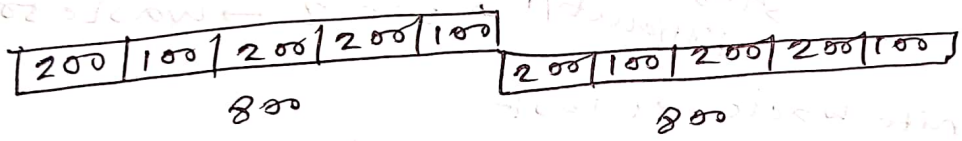
why non-even slice?
memory ops need more length.
why not slice memory op?
↳ atomic op.

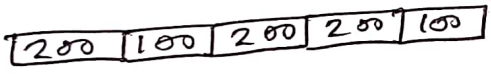
↳ if possible → 8 stages → speedup = 8

tradeoff { stage ↑
speedup ↑
problems ↑

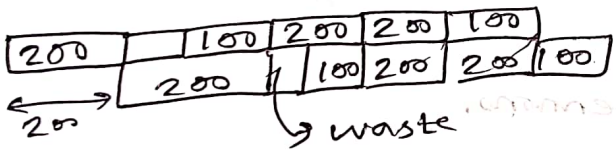
Pipeline performance

Total → 800ps





speedup = $\frac{0.8}{0.2} = 4$



Pipelining & ISA design

MIPS ISA → designed for pipelining.
↳ fixed instn size (4) byte
x86 → 1-15 byte variable size.

fewer addressing format:
MIPS → 3
x86 → 7 for memory access

Load / store addressing
MIPS → memory & op register op not possible

x86
family of
86 processor

8086
specific
processor

arithmetic on mem & reg not possible
→ load first

x86 → one of the operands can be memory
not both
↳ Memory needed both before & after ALU

memory is never needed before ALU op. → better

Alignment of memory operands.

MIPS → per cycle 32 bits.
8086 → 16 bit processor
↳ brings 16 bit data
↳ may arrive in 1/2 cycles. not good for pipelining.
not guaranteed

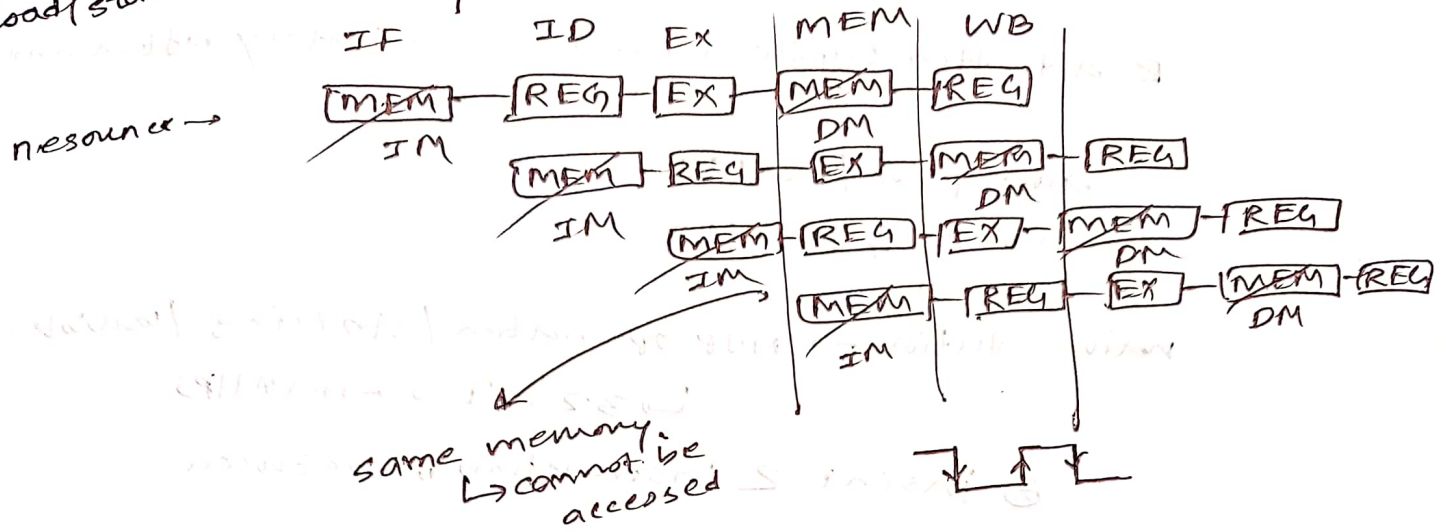
stage splitting difficulty ↑ → use 2 → waste 50%

* MIPS write back at last
→ no intermediate reg write.

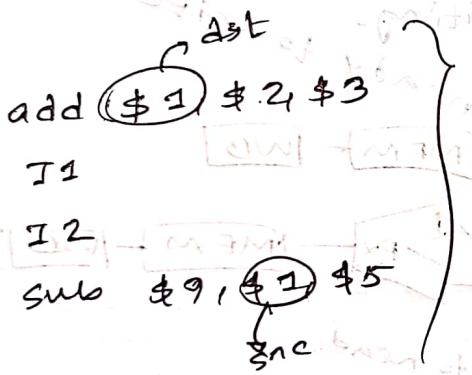
MIPS → 5 stages
↳ pipelining can run into errors.

Hazards
→ structural → when a single time slice / instance has the same resource requested by two instructions.
→ Data
→ control

instruction from memory
load/store to memory

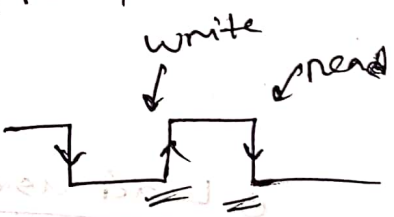


uses instruction memory & Data Memory
↳ solves structural hazard using resource splitting.



w/n @ same slice
→ 2 diff RF won't help.
→ read won't transfer to write.
Reg → must read what has been written

split cycle { WB → 100
clock cycle → 200 } ~~1/2 cycle → read~~ 1/2 cycle → write ready



write on rising edge
read on falling edge
↳ doesn't need edge
but delay the read so it may
the written value.

if I_1, I_2 were not present → RAW Hazard.

Read after Write → read immediately after write

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

naive solution → NOP operation / stalling / bubble...

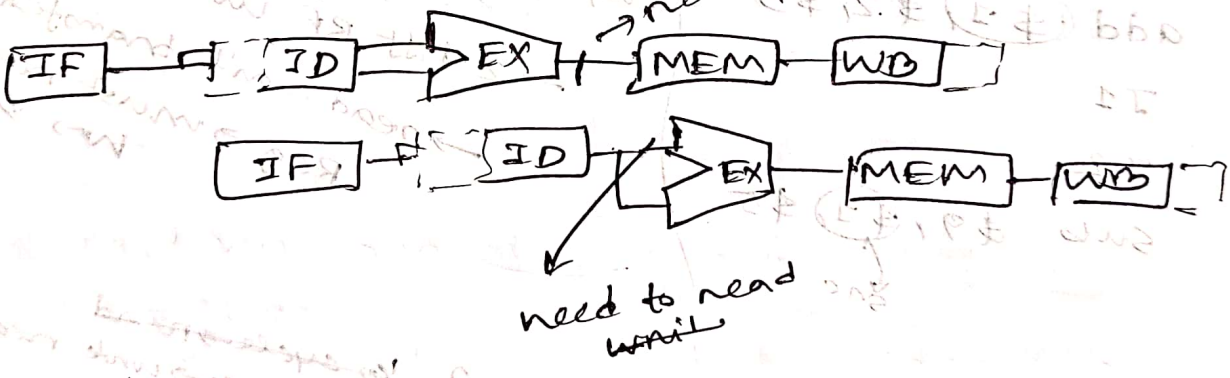
↳ 32 bit 0 → in MIPS

* insert 2 instructions in between

↳ trade off → more time

each RAW Hazard needs 2x waiting → penalty.

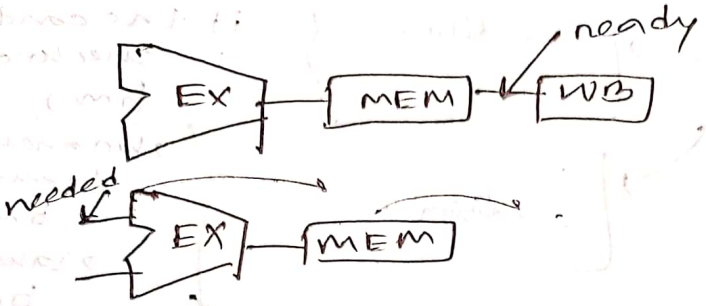
any hazard can be solved by waiting.



↳ forwarding / bypassing. → not always possible

② Load-use data hazard.

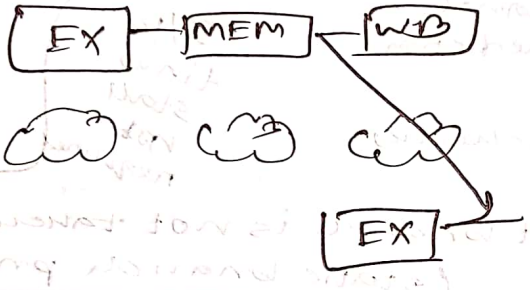
```
lw  $s0, 20($s1)
sub $t2, $s0, $t3
```



forwarding not possible.

backwarding not possible

→ use 1 NOP + forwarding



→ try to add some instruction in between for stalling.

↳ done by compiler.

freeze & flush

-O 1, 2, 3 inc

compiler optimizations are significant

→ checks if there is any independent instruction that can inserted in between.

Control Hazard

↳ Branching.

→ branch taken or not taken → decided after MEM

→ naive soln → stall

add $4, $5, $6

beq $1, $2, 40

still one instruction enters. as it is known that the op is a branch one only after ID.

3 stalls needed.

→ IF after MEM

top-tested loop → condition

bottom-tested loop → condition

if the condition is checked 10 times
↳ branch not taken 9 times
↳ taken 1 time

pipeline → 3 instr branch

9 will be correct, 1 time discard 3 instr
stall

solution → branch prediction
↳ diff. schemes:

9/10 times stall not required

→ predict branch is not taken.
(static branch prediction)
↳ 90%/99% time correct.

similar to bottom test?
→ dynamic branch prediction

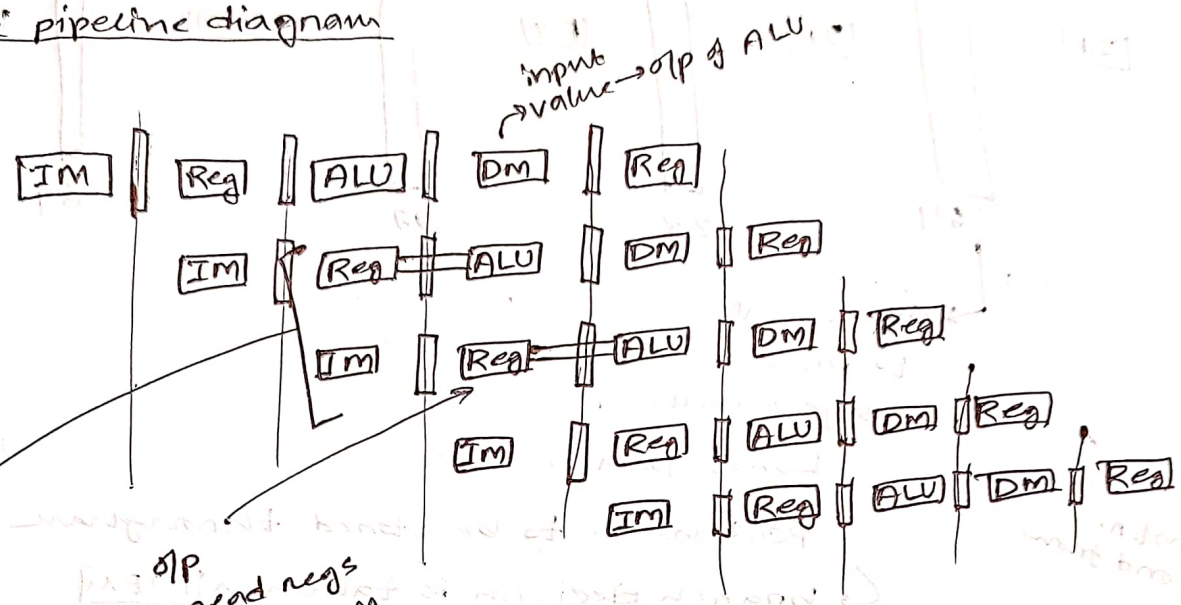
Delayed Branch

beq $1, $2, 40 → write
add $4, $5, $6

independent instruction

if none of the branch neg are needed, insert instruction in between
↳ upto 3 → then MEM has decision

Multi-cycle pipeline diagram



same clock cycle → ALU cannot have 2 sets of i/p prev → vanishes.

o/p → 2 read regs from IM

depends on corresponding IM

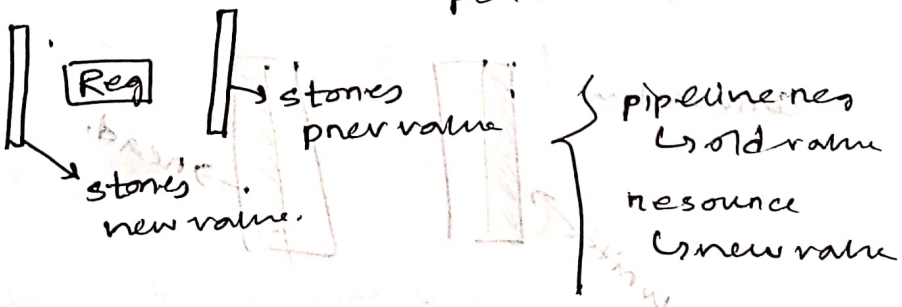
same issue for all stage i/p, o/p.

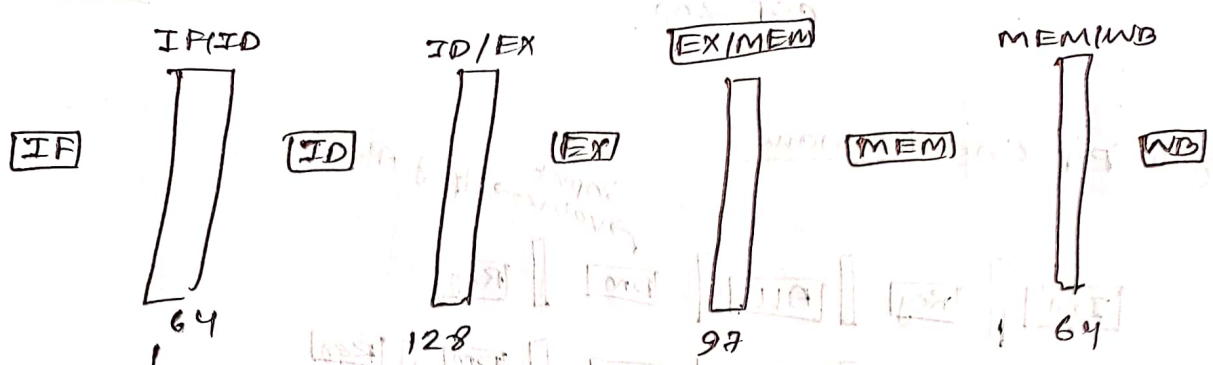
clock rise/fall would work if all worked in loops.

```
lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
..
```

needs → $2, $3 → use pipeline registers.

needs → $3, $4 → stores previous register value per clock-cycle





↳ size equal to prev stage output

↳ not fanout.

PC+4 needs to be stored throughout

↳ branch decision is taken in MEM

ID instruction decode.

get instr. word from.

IM

all stages update (PC+4)

but is needed for branch.

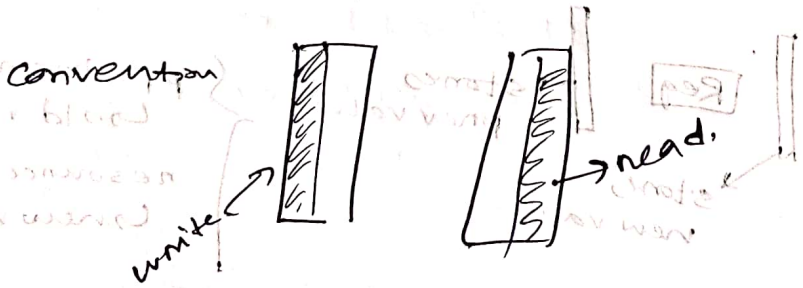
→ after 3 stages.

why not use PC+4-12 PC+16 ?

→ not guaranteed.

* → branching may happen midway. → won't have correct values.

MEM → needs RT for store.

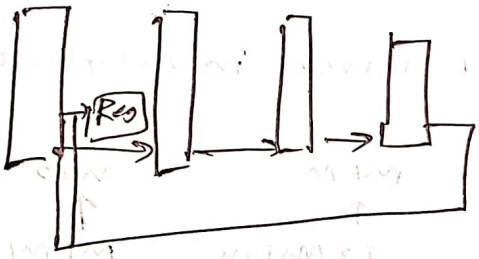


* long in (WB)

→ write reg value gets updated.

(*) 3 clock cycles back so not even in any pipelined regs.

Soln → store & carry the write reg.



Load, Store

pipelined control

Control → controls excluding jump.

IF → no control

ID → no control RegWrite → for WB

EX → ALUSrc, ALUOp, RegDst
1 2 1 → 4 bits

MEM → branch, MemRead, MemWrite

WB → MemtoReg, RegWrite

when is the control signal generated?

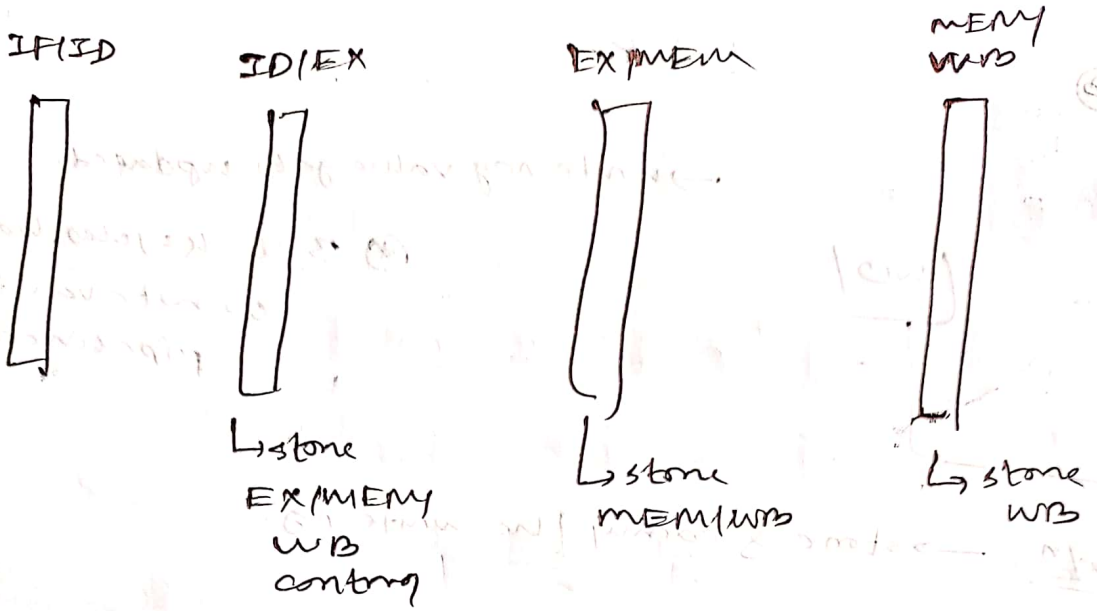
ID

↳ each clk has new instrs.

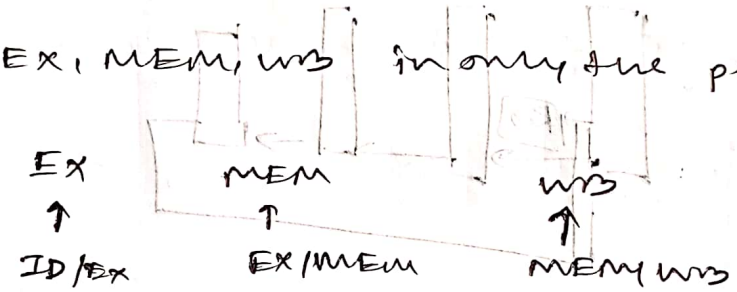
IF → generation not possible. → instn. not known

ID → instn. op code known → generate control
6 bits of opcode to g control

↳ store previous control signals.



Why not EX, MEM, WB in only the pipelines



→ gets overwritten

→ we carry forward, stores values valid for that point.

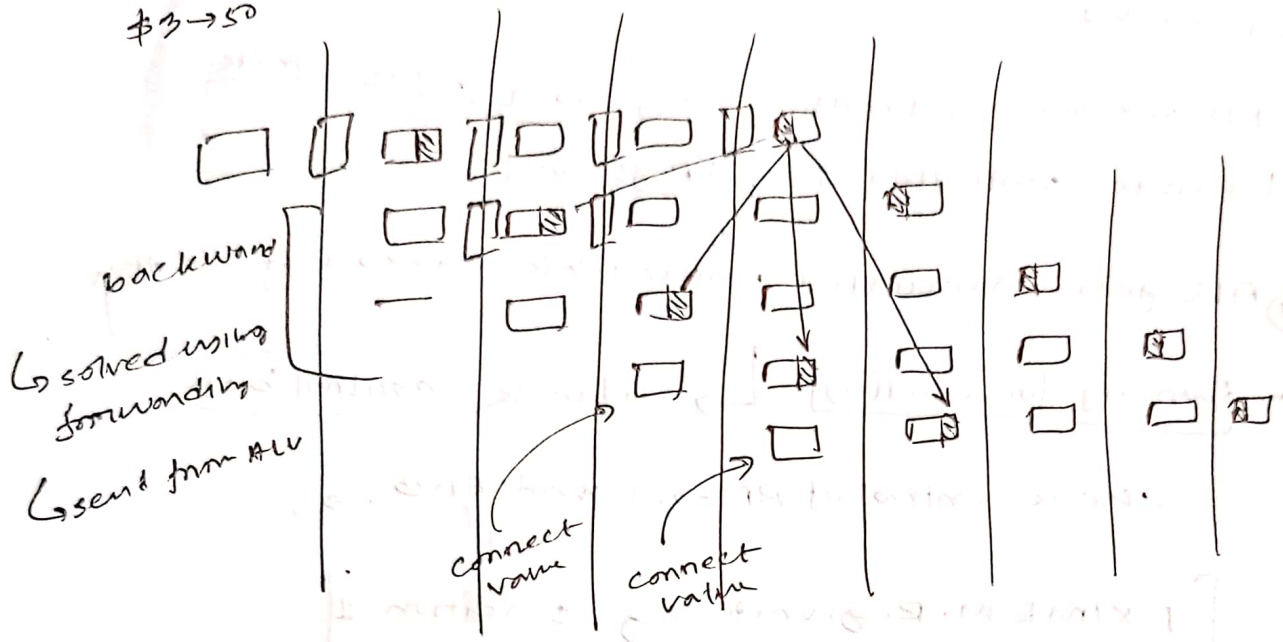
Data Hazards in ALU instructions

```
sub  $2, $1, $3
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)
```

first write to $2

four reads from $2

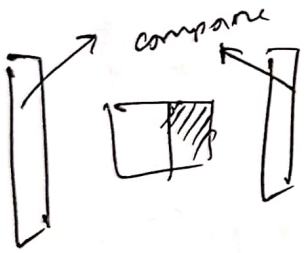
$1 → 30
$3 → 50



$2 written on rising edge.

→ how to forward?
→ how to tell if it is needed?

how to detect hazard?
↳ check RD of prev instn. (immediate)
↳ it matches RS/RT of current,
↳ hazard.



where to get this?
↳ check EX/MEM.RD

known

EX/MEM.RD == ~~EX/MEM~~.RS (IF/ID)
EX/MEM.RD == ~~EX/MEM~~.RT (IF/ID)
→ HAZARD + immediate instruction
1st type
RD is saved

also a variant of RAW

② how to detect the third one? → check MEM/WB

~~EX/MEM~~ ②
IF/ID.RS == MEM/WB.RD
MEM/WB.RD == IF/ID.RT
→ HAZARD
2nd type

detection condition.

previous 2 + ...

RD not always written. eg. beq → no write

↳ write controlled using RegWrite

↳ ALU gets calculated value which was not written.

→ wrong forwarding ↳ check control bits

check control of RF → if WriteEnb → ..

[EX/MEM.RegWrite & condition 1]

[MEM/WB.RegWrite & condition 2]

(*) what if $RD is $ZERO ?

→ can be used in instn

→ not allowed in hardware.

→ wasted instn.

add $0, $1, $2;

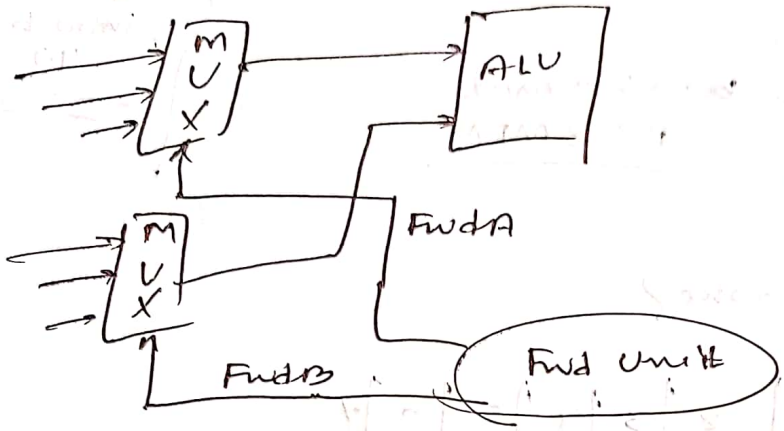
additional check:

- EX/MEM.RD ≠ 0
- MEM/WB.RD ≠ 0

— detected forward till now —

how to forward?

select one of 3 values.



00 → RS →
01 → MEM / WB
10 → EX / MEM

00 → RT / Imm
01 → MEM / WB
10 → EX / MEM

RS → detect → FwdA

RT → detect → FwdB

EX/MEM.RD == ID/EX.RS → FwdA → 10
FwdB → 00

MEM/WB.RD == ID/EX.RT → FwdA → 00
FwdB → 01

Double Data Hazard.

```
add $1,$2,$3
add $1,$1,$3
add $1,$1,$4
```

→ both hazards
read after write
read after after write

double data hazard.

16/09/2024

CSE-209

CT till 115

Ch-4 intro to 4.7

Double Data Hazard

```
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
```

both hazards EX & MEM

how & which value to take?

cc:	1	2	3	4	5	6	7
$1:	10	10	10	10	10 40	40 60	60 64

$2: 30
$3: 20
$4: 4

EX & MEM → both → fwd EX
when checking MEM/WB, check EX/MEM
↳ if true → don't fwd from MEM/WB.

MEM hazard

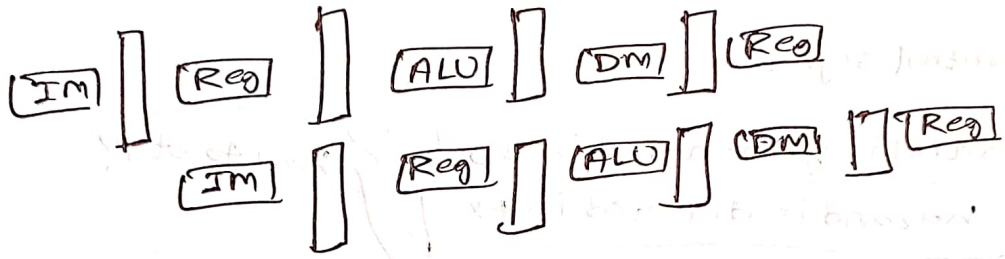
if (MEM/WB.RegWrite & (MEM/WB.RD ≠ 0)
& !((EX/MEM.RegWrite) & (EX/MEM.RD ≠ 0))

Load → Store (Mem to Mem Copy)

lw $2, 20($1)
sw $2, 15($3)

Reg to Reg → possible → same RF
Reg to MEM → no connection

no register to memory connection
↳ register same
↳ must go via ALU



→ use forwarding.

* → how to detect this & how to forward? (HW)
↳ abstract path using MUX, source, DM
→ not full datapath
↳ truth table for MUX

Load-Use data Hazard

↳ stall + forwarding on code reschedule
↳ not guaranteed
↳ must handle using hardware.

RAW / RAAW
↳ soln: forwarding

* How to detect?
* How to stall?

RAW hazard is detected in ~~ID~~ EX stage.

Stall → stop values of instructions pushed to pipeline
↳ use control value 0 → virtually no write
↳ all write signals are deasserted. → virtually garbage instruction

Steps:

① control signal = 0

control is generated in ID.
hazard is detected in EX
} How to stop?

↳ hazard must be detected in ID. ⊛

↳ how?

match with prev RT curr RS/RT.

[ID/EX.RT == IF/ID.RS or ID/EX.RT == IF/ID.RT]

↳ hazard

⊛ how to detect a load? → check MemRead (check control)

only asserted for load

ID/EX.MemRead and (ID/EX.RT == IF/ID.RS or ID/EX.RT == IF/ID.RT)

IF/ID → no controls. → Load-use hazard.

→ forward using additional value in MUX.

(*) the operations still take place but no write happens.

But a new instruction enters in the meantime & the current instruction needs to be reviewed.

→ other values are reserved. PC gets updated. ⇒ PC+4

→ new instruction loaded

→ how to resume? Next pipelines are already preserving values.

(*) ⇒ need to pass IFID to next IFID (1) & IM word to next IM (2)

⇒ set next → PC+8 → stop this.

→ PC+4 is written at the end of CC 3.

→ set PC write signal = 0.

next one also PC+4

→ stop IFID write. → also written @ end of CC 3.

↳ also set IFID write = 0

(*) set next controls = 0, PC write = 0, IFID write = 0

↳ one stall → no more hazard. → other probable ones solved using fwd.

Stall / NOP push → some activity