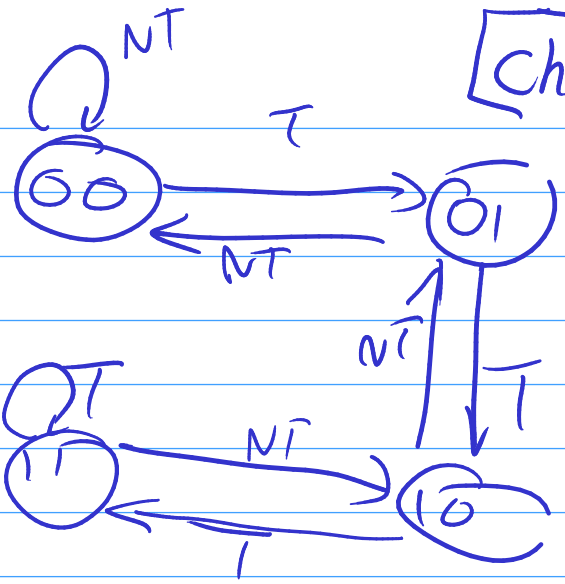


Changes state, if it predict wrong twice

← bit predictor

2-bit predictor as good as k bit

```
for (i=0; i<10; i++)
{ if(i%2!=0)
  {
  }
}
```

1-bit predictor 93%

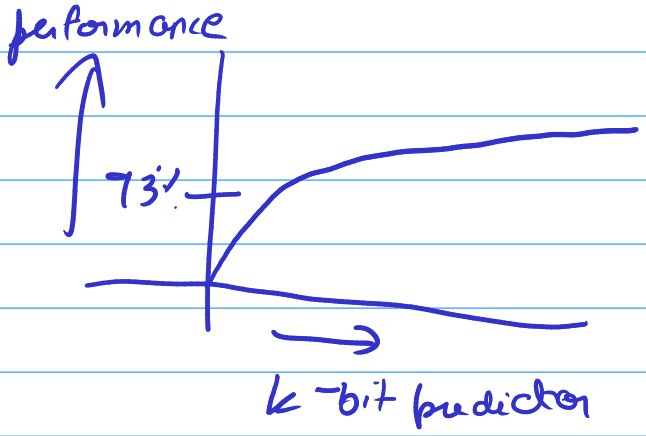
2-bit predictor 96%

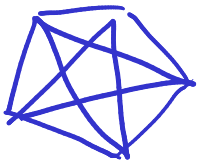
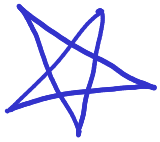
Outcome	T	NT	T	} for 1 bit predictor
Action	NT	T	NT	

100% failure

Outcome	NT	NT	T
Action	T	NT	NT

i%3, i%4 lower



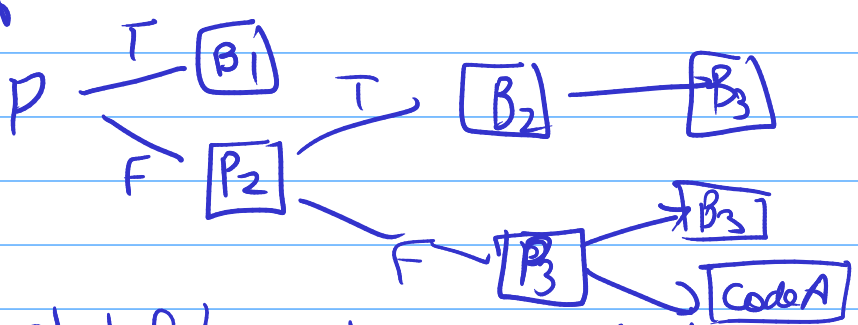



* Global predictors / Correlatory predictors
* local predictors predict branches based on the history of same branch

1024 → beqz
1036 → bnez
↙ predict this

[Global predictors] based on other branches eg @1024 or 1048 (typically recently executed)

P1 if (x==1) {jump;} B1
P2 if (x%4==0) {jump;} B2
P3 if (x%2==0) {jump} B3
CodeA



These are global/correlating predictors

Global predictors — (m,n) predictors
↓
m-bits of correlation
n- bit for branch

1-bit local predictor → (0, 1) global predictor

2-bit local " → (0, 2) global

(2, 2) prediction ⇒ 2^2 global (predictors) correlation

2 local prediction

1012 → B_1
1016 → B_2
→ 1020 → B_3
1024 → B_4
1028 → B_5
1032 → B_6

} For this, select 4 recent branches

(1, 1) predictors

↓ ↘

2¹ global | local

↳ we know this

B_1, B_2 } Check there

B_3 } decide this

x = 11

!(x > 4) B_1 (NT), B_2 (T) }

!(x > 2) B_3 (NT) change this? → T

x = 8

111 ly

$X = 6$

B_1 – NT

B_2 – NT

B_3 – T

To decide this

Mk table

change outcome

B_1 Pred	B_2 Pred	B_3 Pred	B_1 out	B_2 out	B_3 out	(m,n) prediction
NT	T	NT	NT	NT	T	

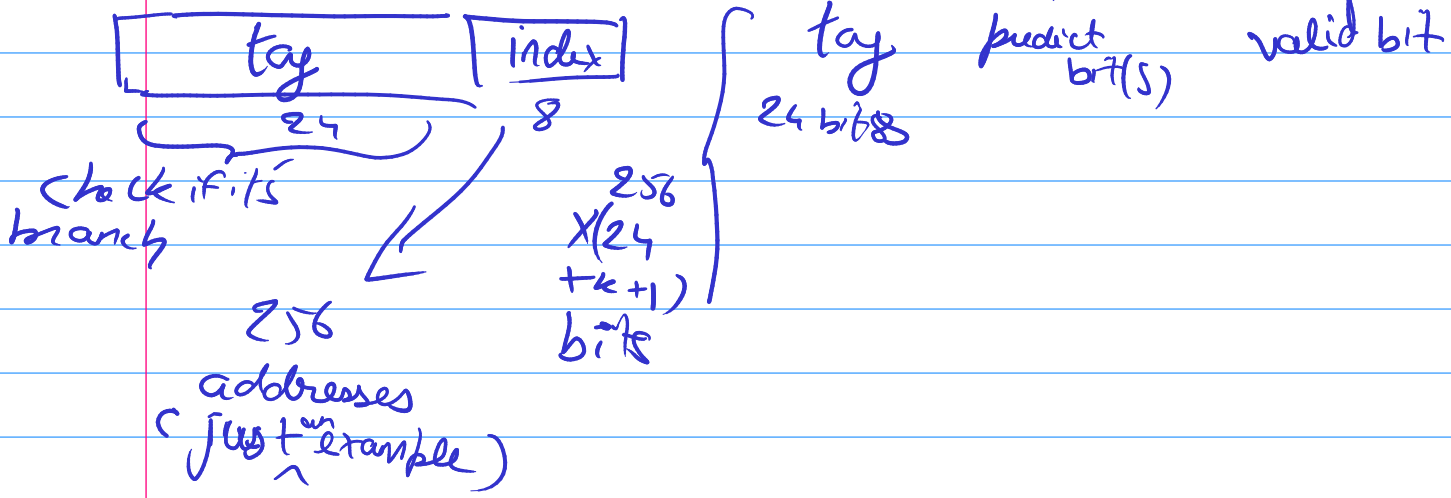
8 Combinations

---

h/w ⇒ Branch target buffer

Local prediction ⇒ How to build a BTB?

PC – 32 bits



2000 0000 | 00 | 00 32 bits

tag indexing

200000 | 2 bits | 0/1

Problem

(1) Same index, different tag

(2) ⇒ finish executing a program, replace it with new program.

the instructions will be wrongly predicted as branch

based on tag

to fix that we use valid bit

Whenever we reinitialize memory, we reset valid bits.

we Load the BTB when we load the first program ever, then we populate the content addressable memory & BTB.

doesn't fix collision however!

so it is not good to place jumps exactly 256 bits apart!!

Compiler helps here!

(Doubt, what is CAM (content addressable memory) though)

(very small unit)
BTB is very power hungry BTW
⇒ Flushing it takes a lot of power
⇒ use valid bit.

Practice problems on this!!

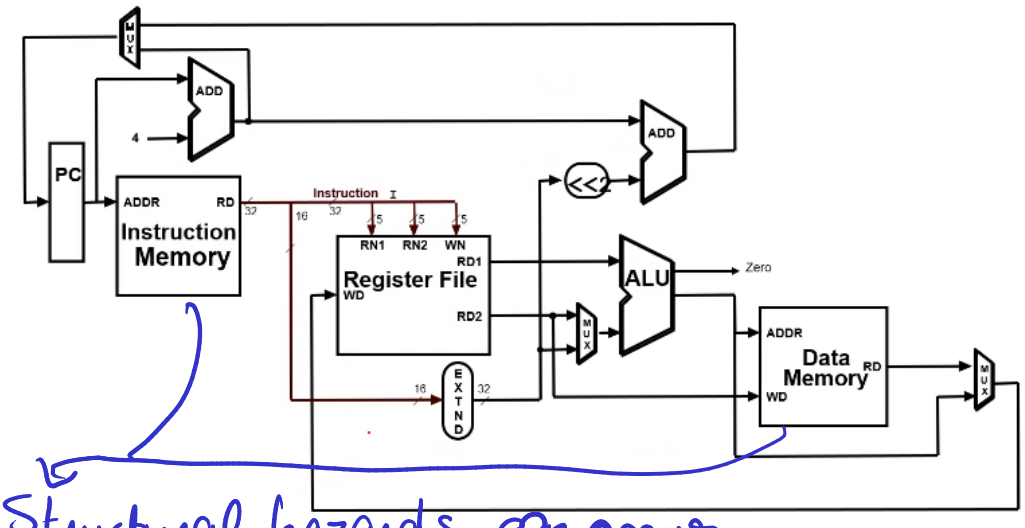
Pipelined Datapath

- We now move to actually building a pipelined datapath
- First recall the 5 steps in instruction execution
  - Instruction Fetch & PC Increment (IF)
  - Instruction Decode and Register Read (ID)
  - Execution or calculate address (EX)
  - Memory access (MEM)
  - Write result into register (WB)
- Review: single-cycle processor
  - all 5 steps done in a single clock cycle
  - dedicated hardware required for each step

Lw - MEM stage
Something else - IF stage
Structural hazard
(both need access)

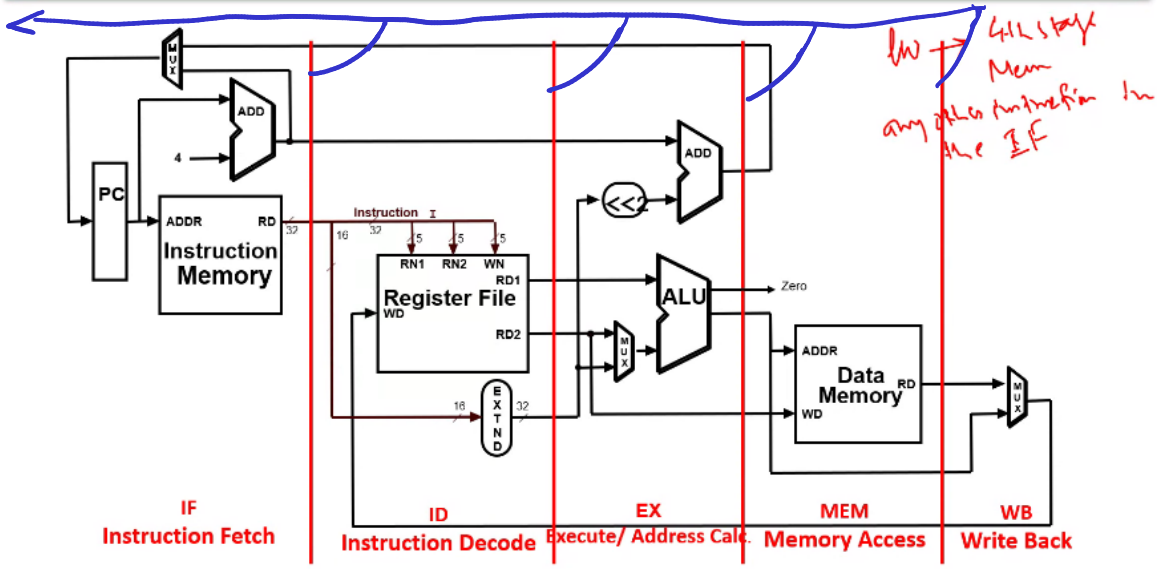
- *What happens if we break the execution into multiple cycles, but keep the extra hardware?* Why?

Review - Single-Cycle Datapath "Steps"



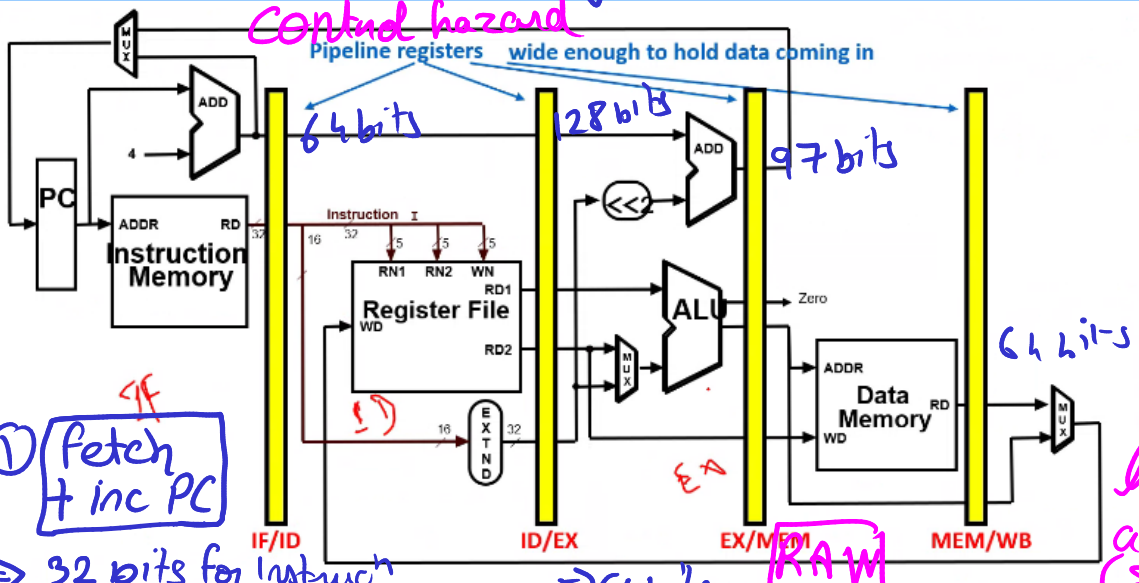
Structural hazards can occur

Review - Single-Cycle Datapath "Steps"



We may be able to start executing a new instruction every clock cycle

Registers between every clock cycle



① Fetch + inc PC

Size of regs ⇒ 32 bits for Instruction
32 bits for new value of PC ⇒ 64 bits

② Operands 32+32
Sign extended value 32
Passing PC+4 (we see later) 32

③ final PC 32
after shift 97
Zero 1bit
result new 32
Data 32

In addition to RAW even when there is no RAW and regs are available.

(Hazard right there)

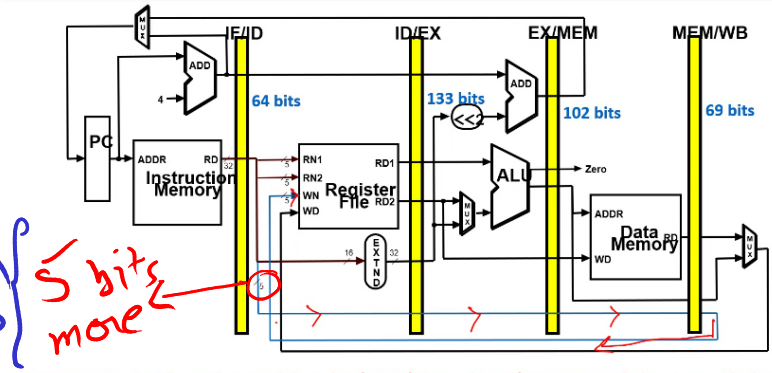
[Hazards can't always be removed, they're inherent]

There's a bypass

lw $1, 4($2)
add $2, $3, $4

This happens

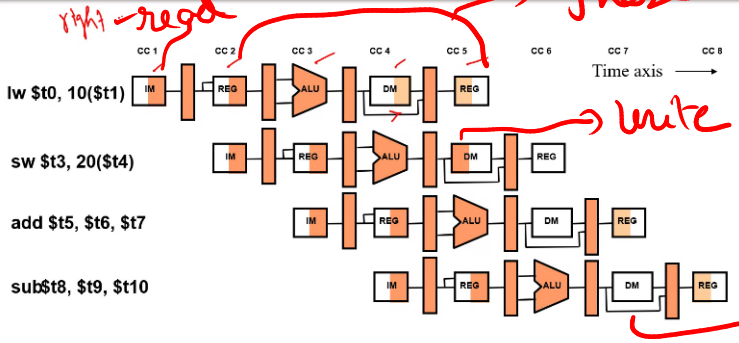
Corrected Datapath



ID/EX
EX/MEM
MEM/WB
5 bits more

Destination register number is also passed through ID/EX, EX/MEM and MEM/WB registers, which are now wider by 5 bits

Alternative View – Multiple-Clock-Cycle Diagram



right - regd

These are same

write

operand forwarding

(not the reservation station kind, but by definition)