Compiler optimizations, Avoid load-use stalls

- Move instruction between
  "fill load-delay slot"

OK, if
- can find instruction w/o dependencies

```
lw  $1, offset($base)   <-- OR from before lw
add $3, $1, $2
and $4, $4, 0           E.G.
```

- Let hardware insert Nop

- Compiler fill w/ nop

  HW doesn't have load-use detection

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for A = B + E; C = B + F; ← reg assignments
  (3 1 2 5 1 4)

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2   <- stall
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4   <- stall
sw  $t5, 16($t0)
```

13 cycles

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)     <- OK
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

11 cycles

pipe-fill: 5 stages ⇒ 4 nops
execution: 7 instructions
load-use stalls: 2 bubbles
13 ticks

4
+ 7
11 ticks

what about pipe drain?

$S = 13/11 = 1\ 2/11$

≈ 15%

Control Hazards: BR

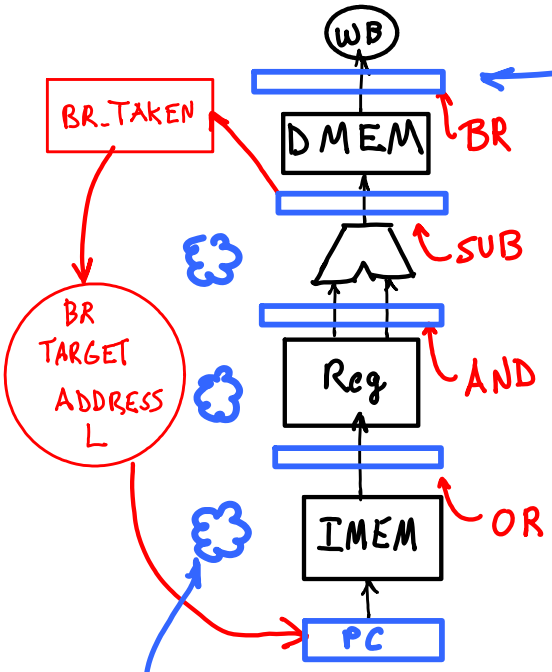
BR, MIPS: $1 - $2 is 0? → MEM
BR, LC4: R2 is 0? → ALU or REG?

On clock tick:

1. Instructions written to next pipe stage
2. BR target address written to PC

```
BR  L
SUB
AND
OR
L:
ADD
```

do not execute? stall?



Pipe stage Reg

Target feedback is forwarding to PC.

PC is 6th pipe register.

write NOPs To Pipe stages?
= "NULLIFY" prefetched instrs
→ 3 bubbles per BR?

Problems w/ NOPs?
No Reg or Mem writes: OK

loop
BR mostly taken in loops, but
done fetching as if not taken.

20% BR ⇒ CPI = (0.2)(1+3) + (0.8)1
= 0.8 + 0.8
⇒ slow down of 2!

Costly! missed prediction (predicted not-taken)

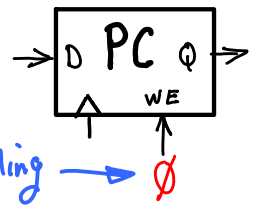
CHOICES

Nulls : worry about data forwarding?

1. **Stall on branches** EASY
   - Wait until you know the answer (branch CPI becomes 3)
   - Control inserted in ID stage of the pipeline

ALL 4

→ STALL FETCHES, INSERT NULLS at ID



2. **Predict not-taken** HARDER
   - Assume branch is not taken and continue with PC+4
   - If branch is actually not taken, all is good
   - Otherwise, nullify misfetched instructions and restart from PC +4+offset

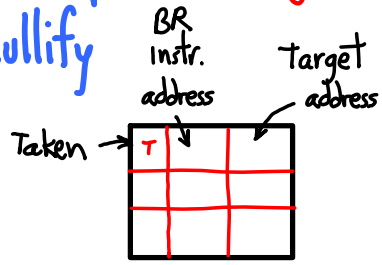
CPI = 4 if Taken, 1 if not taken

→ add Nullify HW To ID, REG, EX

3. **Predict taken** Can we do this? How?
   - Assume branch is taken
   - More complex, cause you also need to predict PC+4+offset
   - Still need to nullify instructions if assumption is wrong
- Most machines do some type of prediction (taken, not-taken)

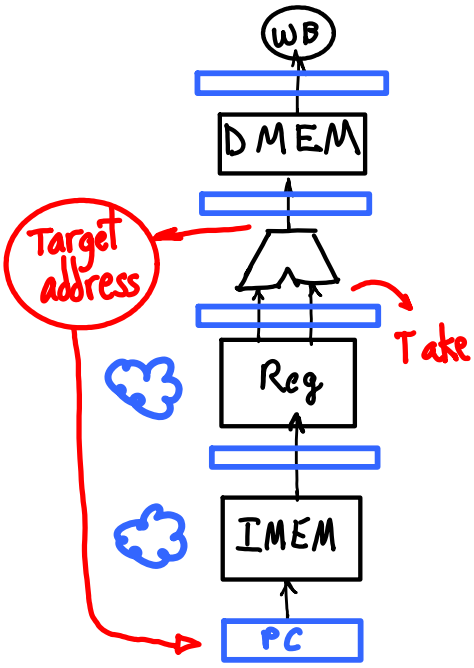
how?

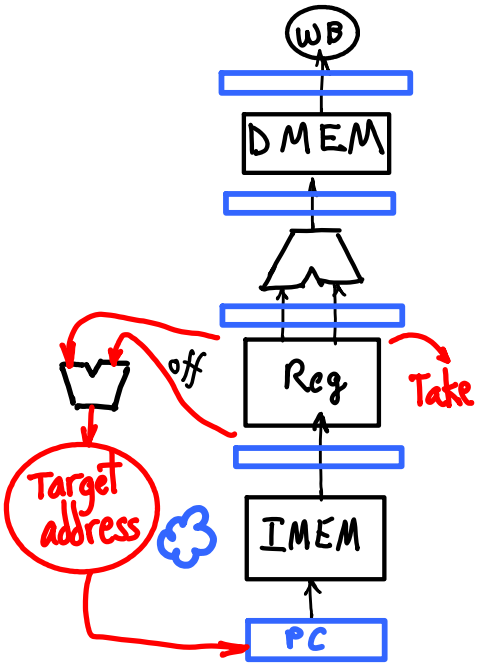
→ Pre-compute BR Target? + nullify



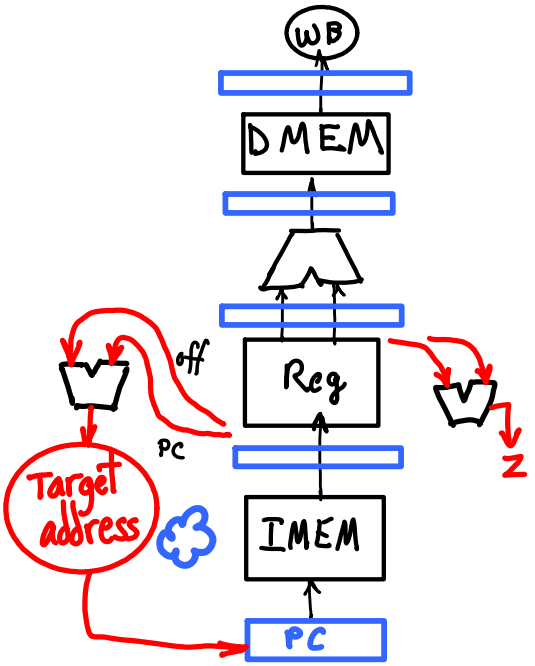
Reduce BR penalty

LC4 BR





MIPS BR



CPI = { 2, Taken ; 1, not Taken

Added HW, added delays:
0. BR hazard detection (similar to forwarding).
1. adder after sign-extension (reg fetch).
2. EQUALS test after register fetch.

Suppose BR = { 50% taken / 50% not taken

1. early BR + hazard detection — eh

$$CPI_{BR} = \begin{cases} 2, \text{ taken} \\ 1, \text{ not taken} \end{cases} \Rightarrow \overline{CPI_{BR}} = (0.5)2 + (0.5)1 = 1.5$$

2. early BR + Compiler NOPs — ec

$$CPI_{BR} = 2$$

3. late BR + hazard detection — lh

$$CPI_{BR} = \begin{cases} 4, \text{ taken} \\ 1, \text{ not taken} \end{cases} \Rightarrow \overline{CPI_{BR}} = (0.5 \cdot 4 + 0.5 \cdot 1) = 2.5$$

4. Late BR, compiler NOPs — lc

LC4: $CPI_{BR} = 3$ MIPS: $CPI = 4$ cheap HW!

Which is best balance? Power? Area? Yeild?

Suppose trace has 25% BR instructions. →

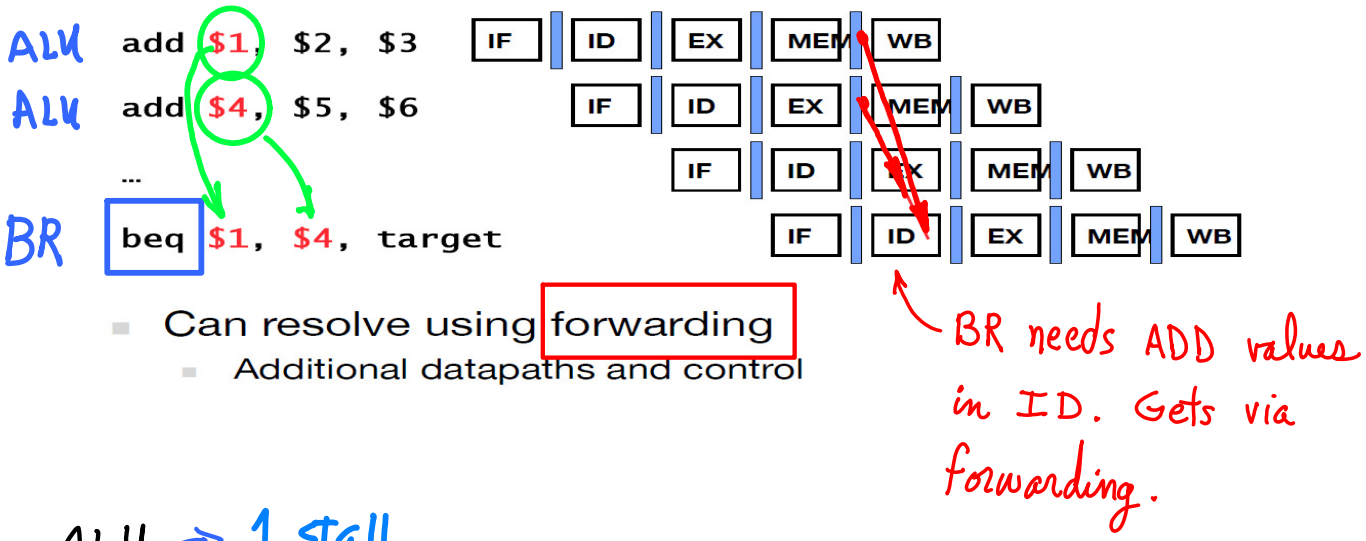
$S_{eh\text{-}ec} = 4/3 \sim 33\%$

$S_{eh\text{-}lh} = 5/3 \sim 67\%$

$S_{ec\text{-}lh} = 5/4 \sim 25\%$

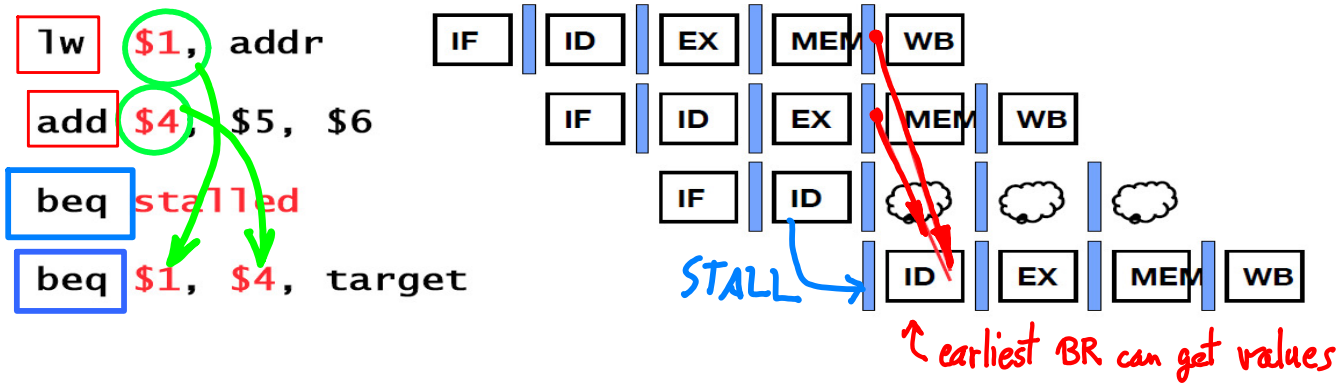
Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

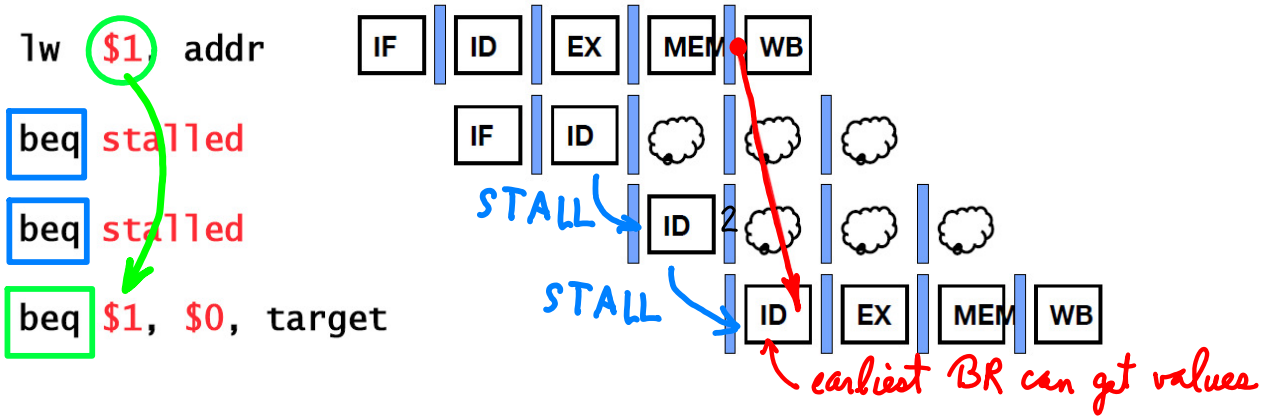


- Can resolve using forwarding
  - Additional datapaths and control

ALU → 1 stall



lw BR ⇒ 2 stalls



Delayed BR alternative

```
add          add
add          BR
BR           add
SUB          SUB
```

BR Taken (BR Taken)

"delay slot" is always executed, no bubble needed to avoid mis-predicted not-taken.

compiler fills when possible, else fill w/ NOP

⇒ 50% NOPs in delay slot.

Are branches really 3 instructions:

1) condition, 2) address, 3) take branch ?

---

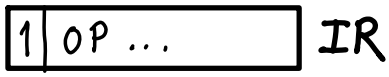
evaluate condition
evaluate target address
execute branch (load PC)

Q. Delayed BR has same performance as (1) above?

Long pipelines?

⇒ Predicated instructions

Predicate register = 1



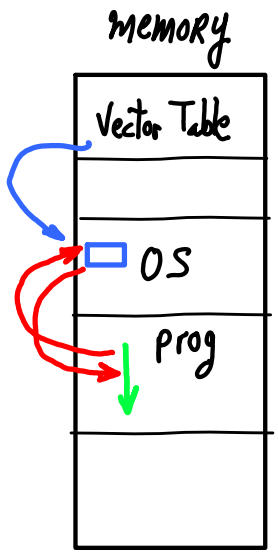
execute if Predicate = 1 else NOP

execute if Predicate = 0 else NOP

```
evalCond Pred, $1, $2
evalAddr $3, PC, offset
loadPC $3        ← loads PC if Pred = 1
NAND             ← execute if Pred = 0
```

fetched into pipe, but No need for nullify HW

Control Hazards: Exceptions, Traps, Interrupts



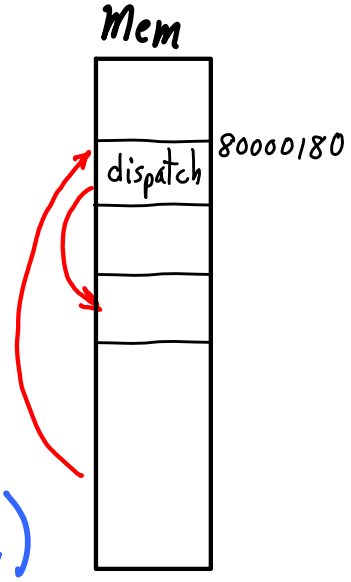
– Something happens

- I/O device sends signal: INTERRUPT
- CPU detects execution error: EXCEPTION
- Execution of a sys call: TRAP

– Do something about it

- Talk to device, get data, send data → jump back to prog.
- Send error message, terminate program
- Jump to OS routine, do service → jump back to prog.

OPTIONS FOR Control Transfer



– Hardwired: always go to 8000 0180

- figure out what routine to jump to ("cause" Reg.)
- Maybe a few Targets hardwired:

80000080 INT
80000180 EXC
80000280 TRAP

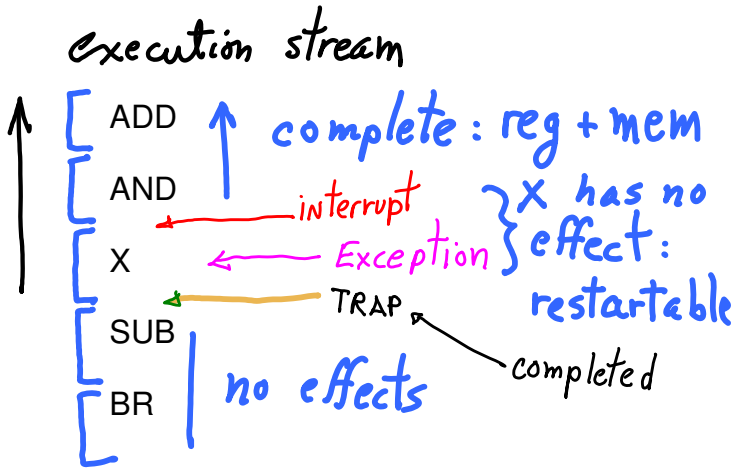
– Hardwired jump via jump Table (Vector Table)

– Combination of these (dispatcher per vector)

Jump ≈ BR hazard

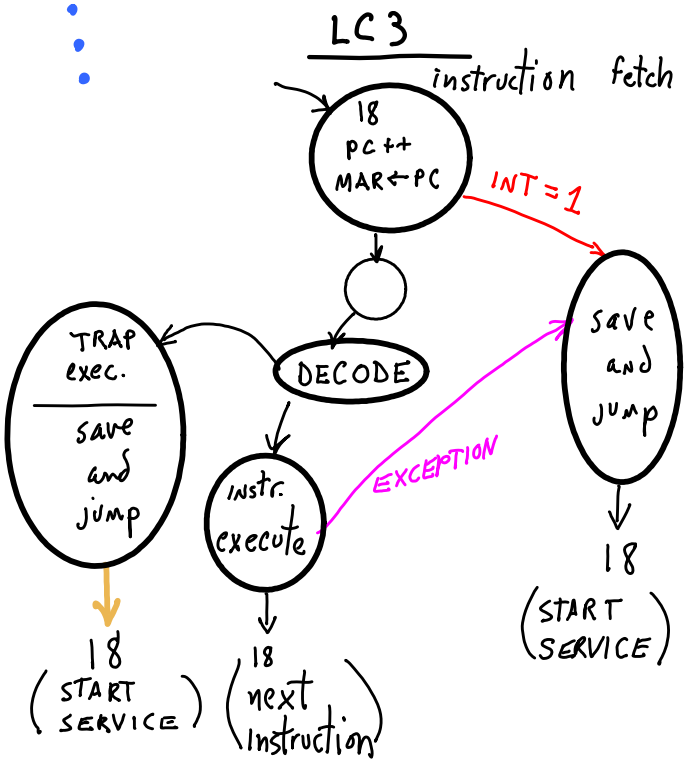
Precise Exceptions

- Definition: precise exceptions, as if
  - All previous instructions had completed
  - The faulting instruction was not started
  - None of the next instructions were started
    - No changes to the architecture state (registers, memory)
- Why are precise exceptions desirable by OS developers?
- With a single cycle machine, precise exceptions are easy
  - Why?



We (OS) need To Know

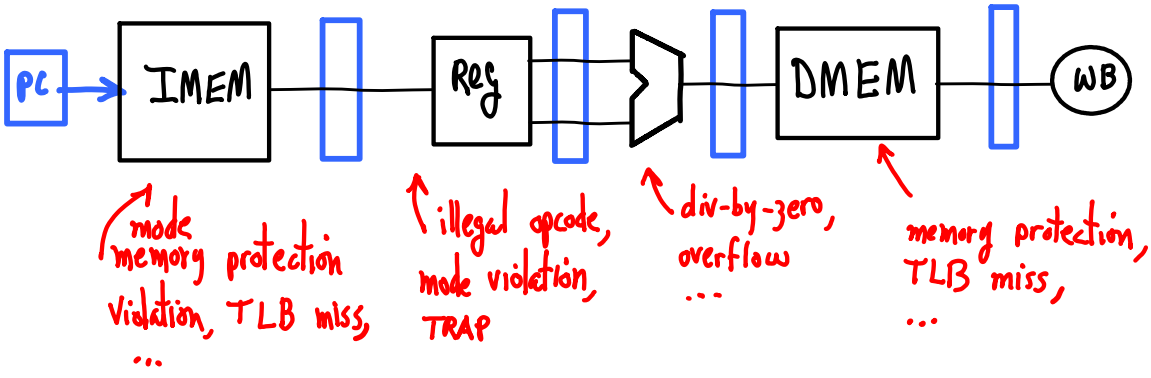
- What happened (INT, EXC, TRAP)
- How to restart (PC, ...)
- which instruction caused problem
- what data caused problem
- which device needs help



MIPS

Use EPC - record PC of current instruction

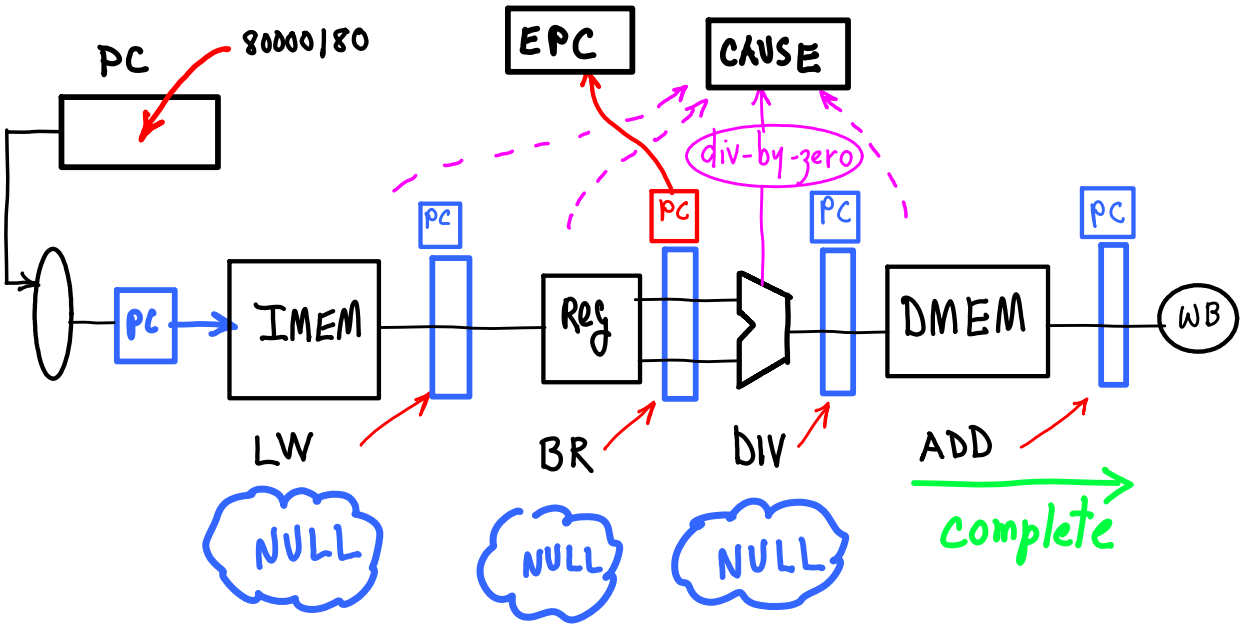
CAUSE-REG - record what happened



CAUSE REG:
which (stage, exception) for all stages

Bit vector = OR of stage-causes

EPC:
PCs calculated from EPC.



1. **Previous instructions complete,**
   NULL following instructions

2. **Save PC into EPC**

3. **Save exception code into EPC**

4. **NULL offending instruction (saves state = MEM+REG)**

5. **Jump to OS at 80000180**
   (or, freeze and start co-processor)

???---multiple exceptions?

overflow

- Exception on add in

```
40    sub  $11, $2, $4
44    and  $12, $2, $5
48    or   $13, $2, $6
4C    add  $1,  $2, $1
50    slt  $15, $6, $7
54    lw   $16, 50($7)
…
```

address

- Handler

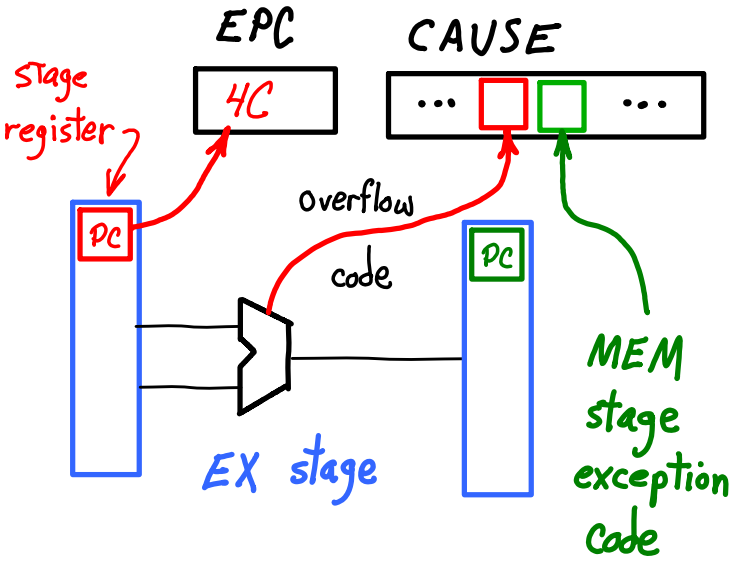
```
80000180   sw   $26, 1000($0)
80000184   sw   $27, 1004($0)
  …
```

OS code: save registers, …

access cause register:

mfc0 $18, $13

co-processor register $13 = cause register



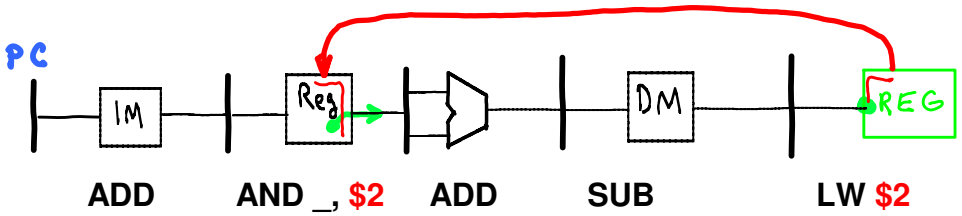
- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once — overflow, illegal opcode, seg fault, TLB miss, …
- Simple approach: deal with exception from earliest instruction → following instructions might not be restarted, e.g. abort
  - Flush subsequent instructions
  - Necessary for "precise" exceptions
- In more complex pipelines — might not cause exception often restart
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

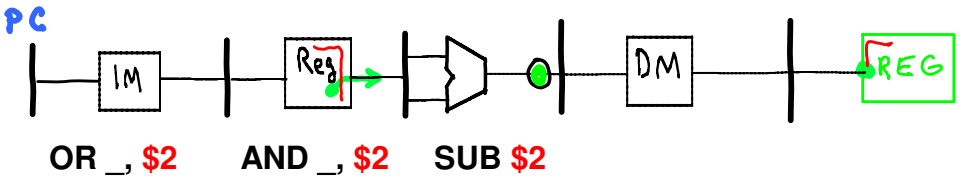
— **stop pipe**: save state, let software figure it out, as possible

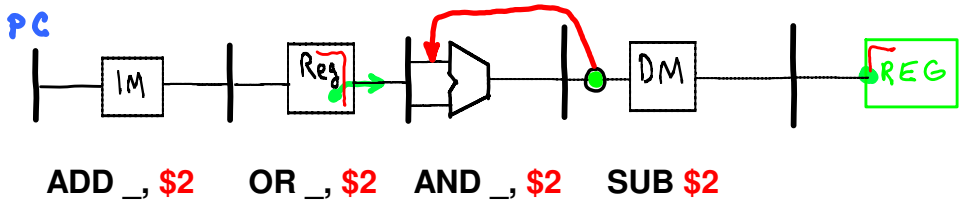
Pipe Summary

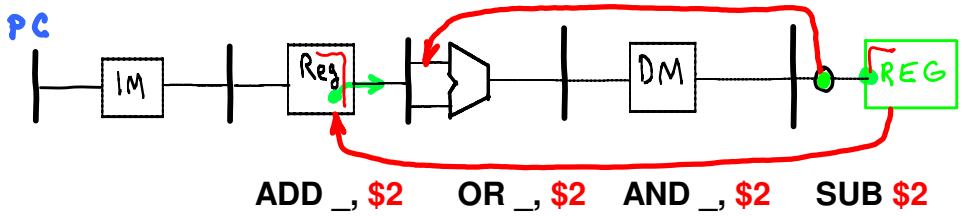
Shorten feedback through register file using neg. edge triggered FFs.



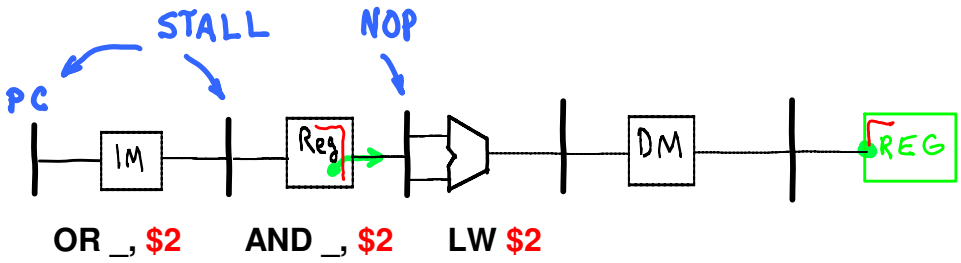
Data hazard detection can forward data without bubbles for operate instructions.

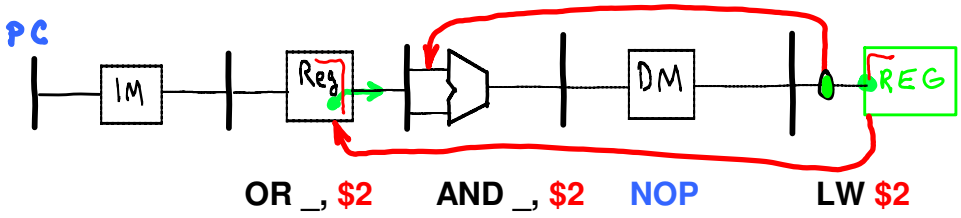




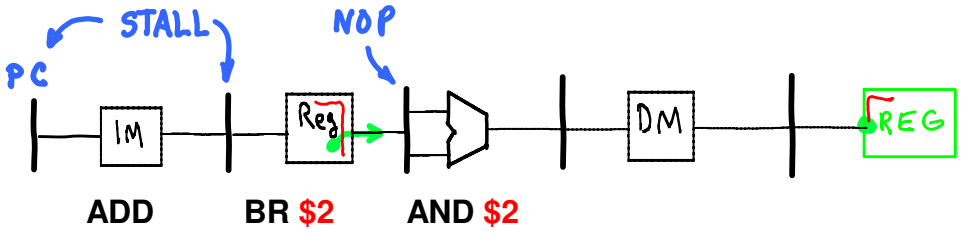


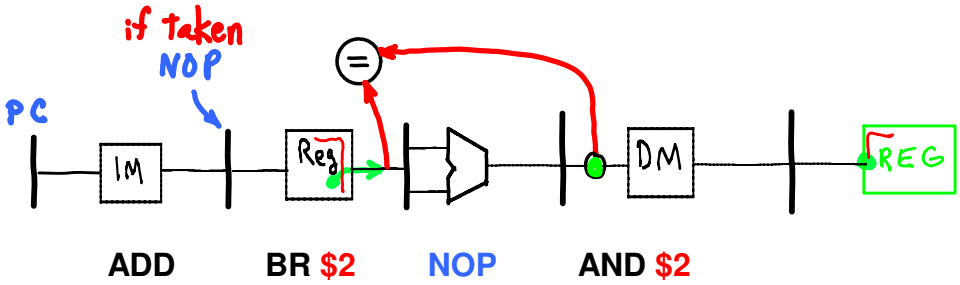
Load-use delay causes a bubble (unless compiler fills slot), then forwarding used.



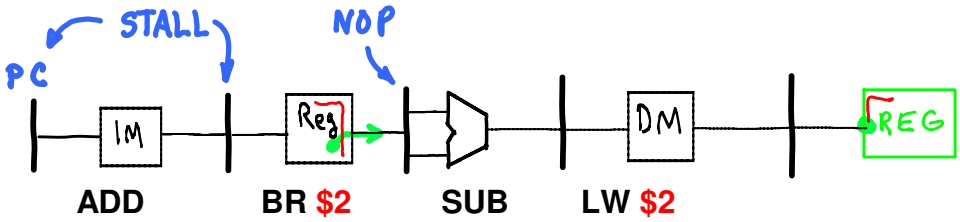


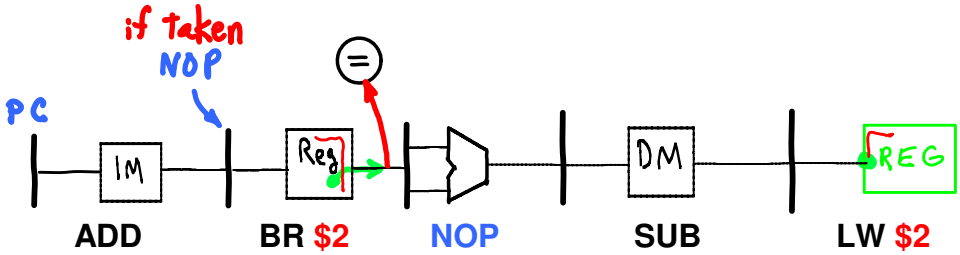
Branch data hazard from operate instruction cause stall and one bubble, then uses forwarding. Almost the same as load-use delay. Inserts NOP if branch taken.



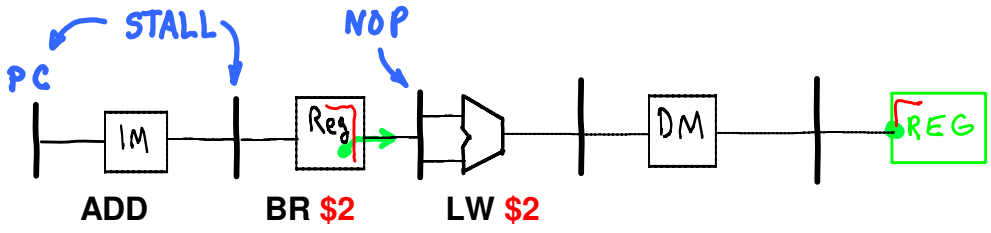


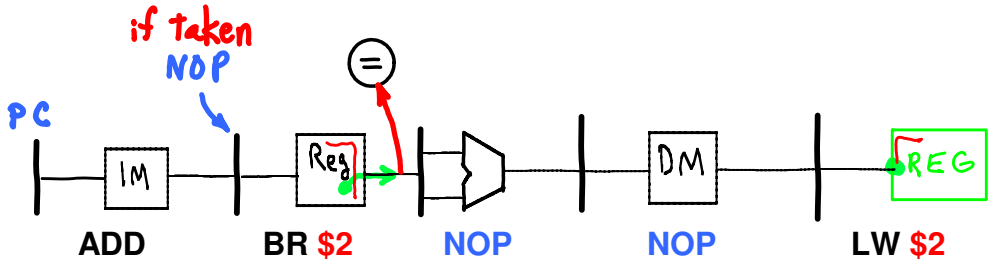
Branch data hazard from LW instruction in DMEM causes stall and one bubble, then uses forwarding. Same as load-use data hazard.



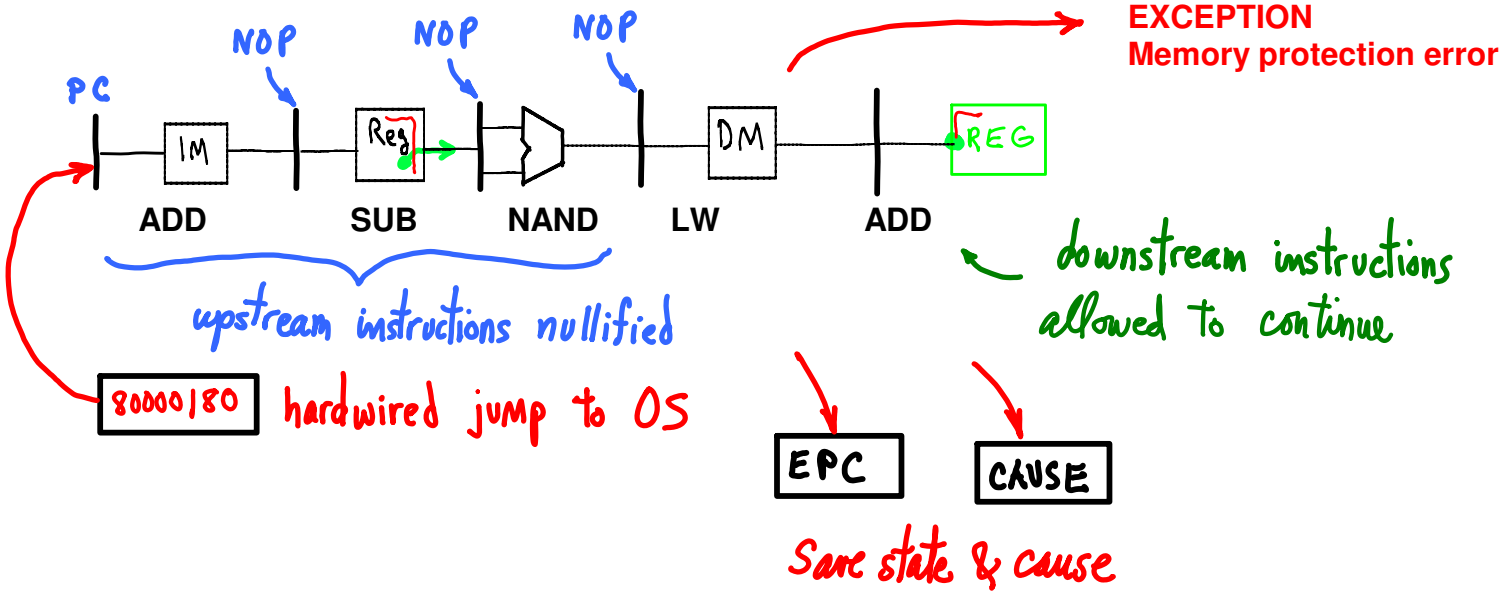


Branch data dependency with LW in EX causes two bubbles, then forwarding.





Exceptions, traps, and interrupts can cause many bubbles.



Questions

1. What to do w/ multiple exceptions during same clock cycle?

2. What to do w/ exceptions for completing instructions?

3. How to know what happened?

4. What about nested exceptions; i.e., exceptions occuring during exception handling?