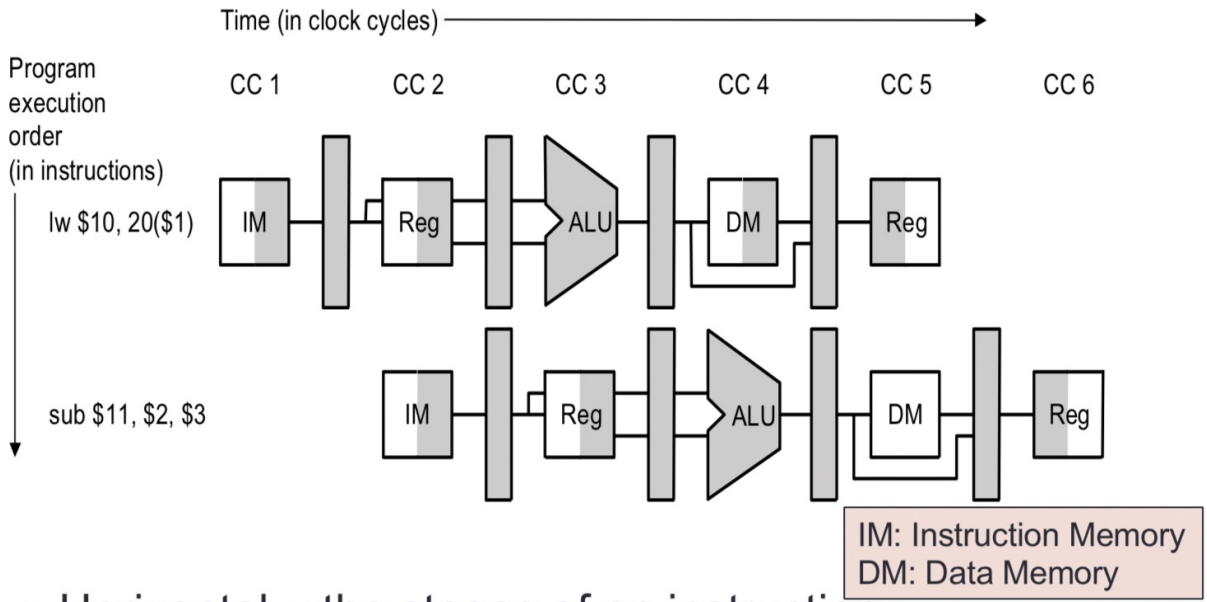
Pipelining Hazards.

- Speedup from pipeline implementation:
  - Based on the assumption that new instructions can be "pumped" into pipeline every cycle.
- However, there are pipeline hazards:
  - Problems that prevent next instruction from immediately following previous instruction.
  - Structural Hazards:
    - Simultaneous use of a hardware resource (RegFile, Memory, ALU)
  - Data Hazards:
    - Data dependencies between instructions. (eg. lw, sw)
  - Control Hazards:
    - Change in program flow. (eg. beq)

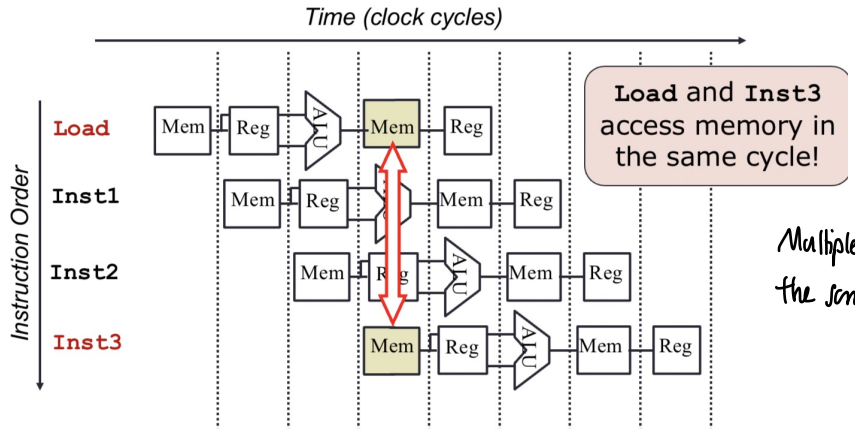
Graphical Notation for Pipeline:



- Horizontal = the stages of an instruction
- Vertical = the instructions in different pipeline stages

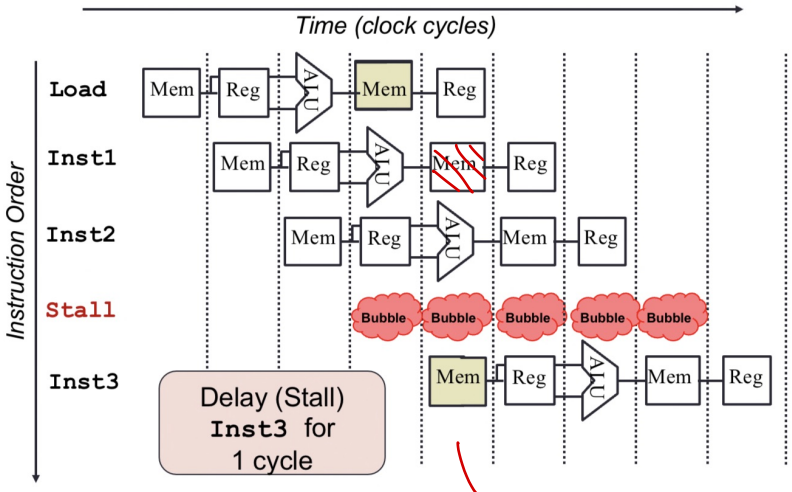
Structural Hazard

- If there is only a **single memory module:**



Multiple instructions are accessing the same resource.

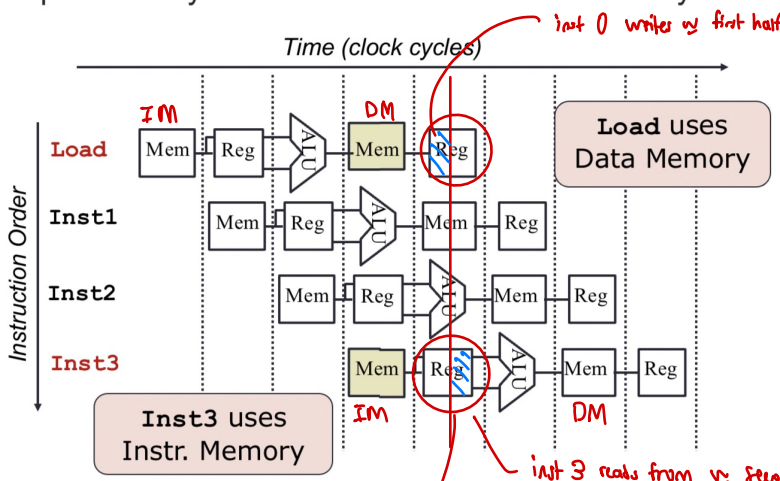
Solution 1: Stall the Pipeline



still got structural hazard ⇒ need to stall by 3 clock cycles.

2. Solution 2: Separate Memory

- Split memory into **Data** and **Instruction** memory



Inst 0 writes w first half.

Inst 3 reads from w second half.

Still got conflict here when Reg is used.

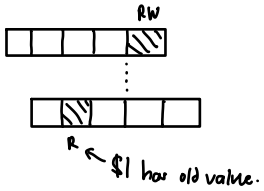
RegFile cannot be split!

⇒ Split cycle into half. First half for writing into a register, second half for reading from a register. (Reg are fast memory)

reverse does not work as instructions are executed from top to bottom. i.e. Instr.3 always depends on instruction above.

Data Dependency

- When 2 instructions access (read/write) the same register (register contention)
- RAW
  - "Read-After-Write"
    - occurs when a later instruction reads from the destination register written by an earlier instruction.
    - also "true data dependency"
    - eg. i1: add $1, $2, $3 # writes to $1
      i2: sub $4, $1, $5 # reads from $1

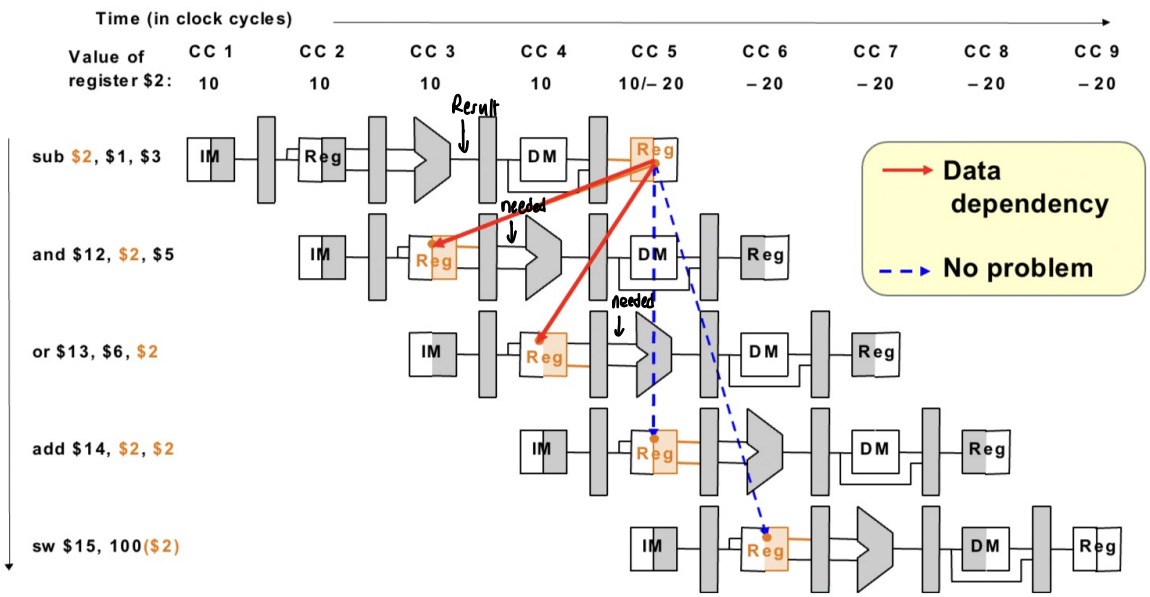


    - Effect of incorrect execution:
      - If i2 reads register $1 before i1 can write back the result, i2 will get a stale result (old result)
  - Also have:
    - WAR: "Write-after-Read"
    - WAW: "Write-after-Write"

} Do not cause any pipeline hazards; Affects processor only when instructions are executed out of program order.

MIPS is in sequential order

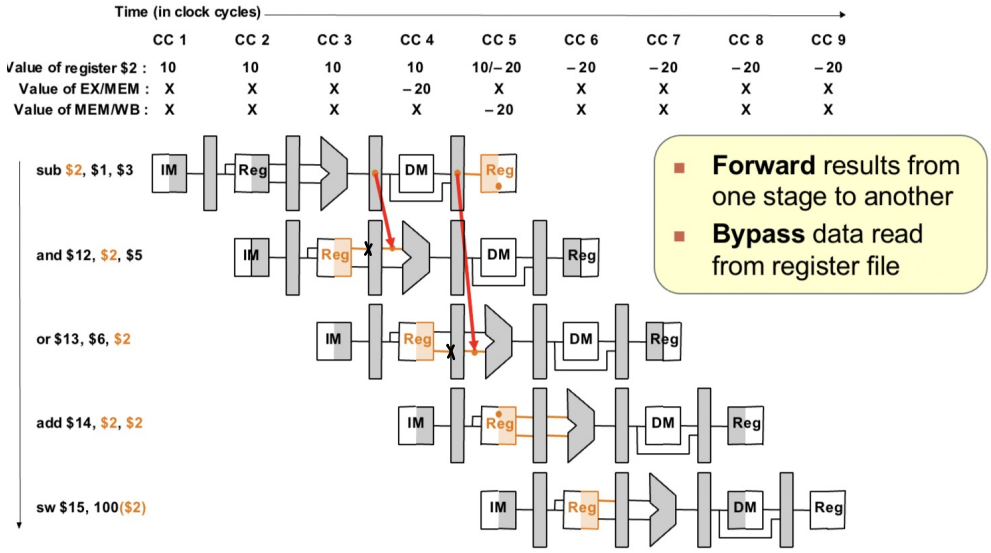
- Value from prior instruction is needed before write back



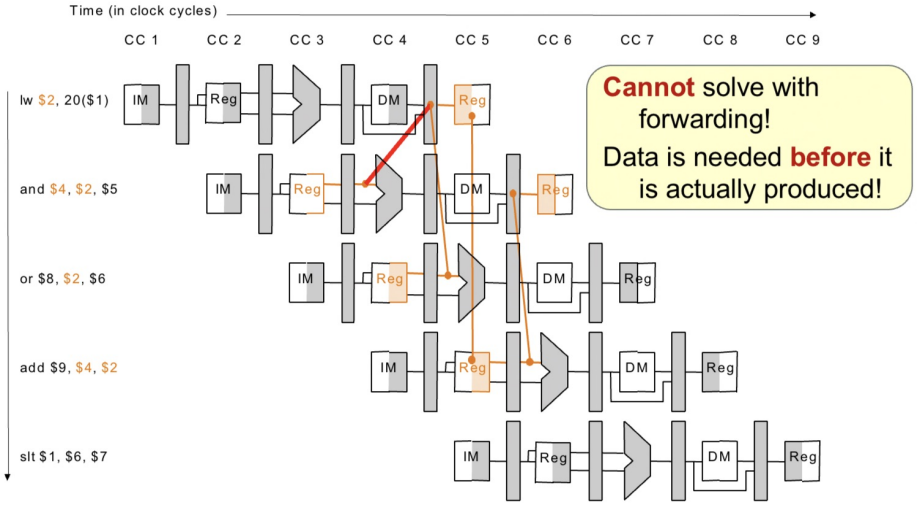
Solution: (Branch Forwarding)

- Forward the result to any trailing (later) instructions before it is reflected in the register file.

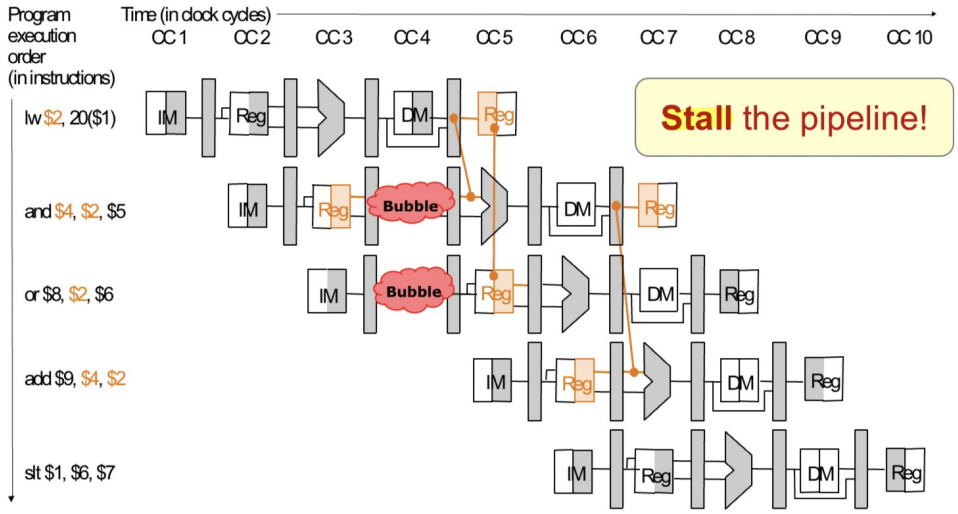
→ Bypass (replace) the data read from register file.



4.2 Data Hazards: **LOAD** Instruction



4.2 Data Hazards: **LOAD** Instruction Solution



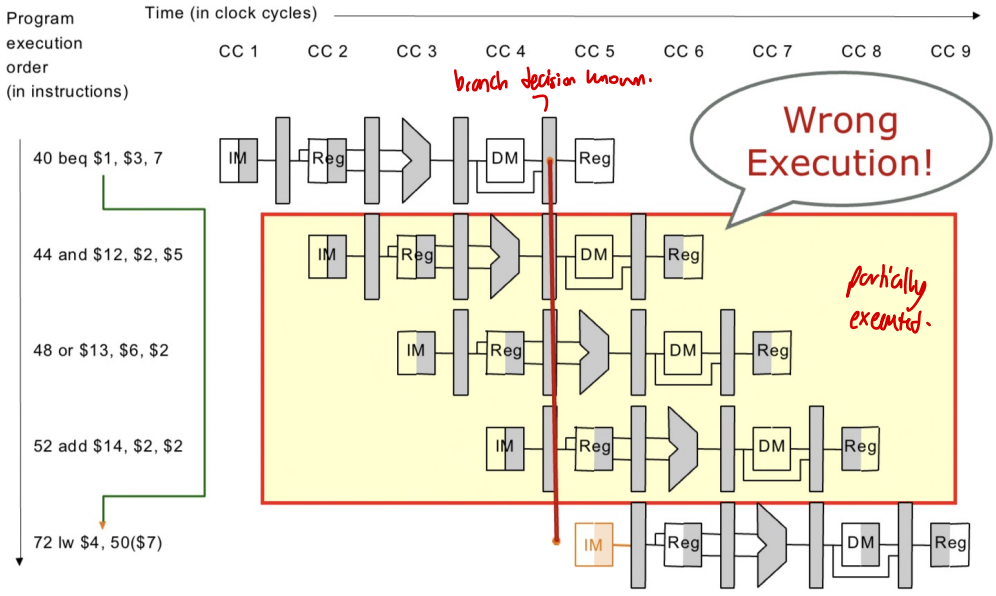
Control Dependency

An instruction j is control dependent on i if i controls whether or not j executes

- Typically i would be a branch instruction.

- eg. i1: beq $3, $5, label # branch
  i2: add $1, $2, $4. # depends on i1.

- Effect of incorrect execution:
  - If i2 is allowed to execute before i1 is determined, register $1 may be incorrectly changed.



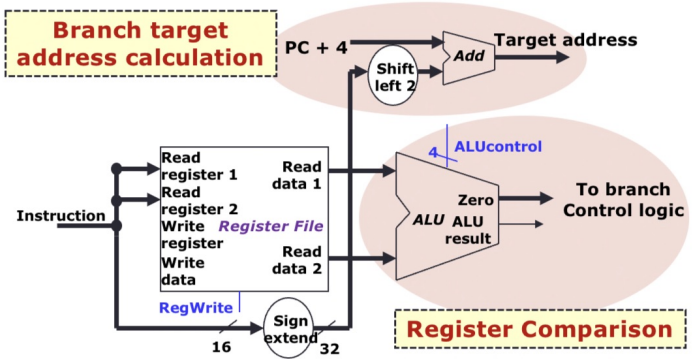
Solution:

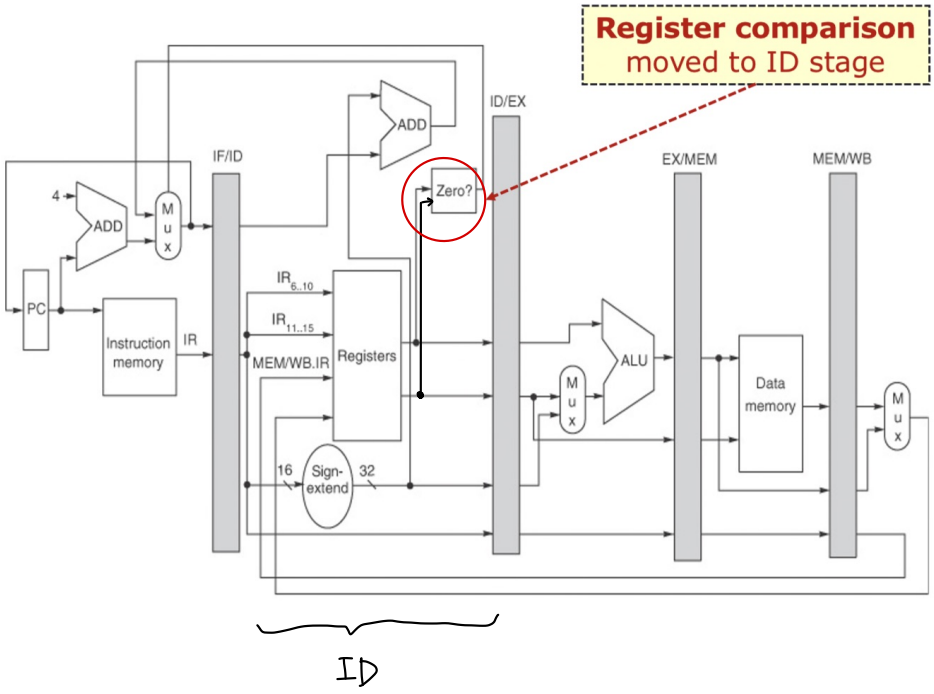
- Add delays until decision is known ⇒ wasting 3 clock cycles.
- 3 clock cycle penalty (reduce):
  - Early Branch Resolution – move branch decision calculation to earlier pipeline stage
  - Branch Prediction – guess the outcome before it is produced. (Correct if wrong)
  - Delayed Branching – do something useful while waiting for the outcome. (Calculate instr. not dependent on branch)

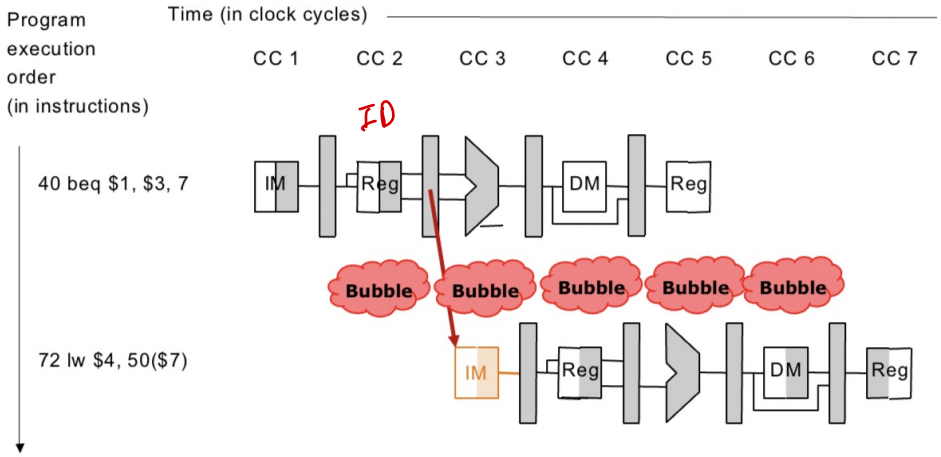
Early Branch Resolution

Make decision in **ID stage** instead of **MEM** — branch decision is known at the end of ID stage.

- Move branch target address calculation
- Move register comparison ➔ cannot use ALU for register comparison any more





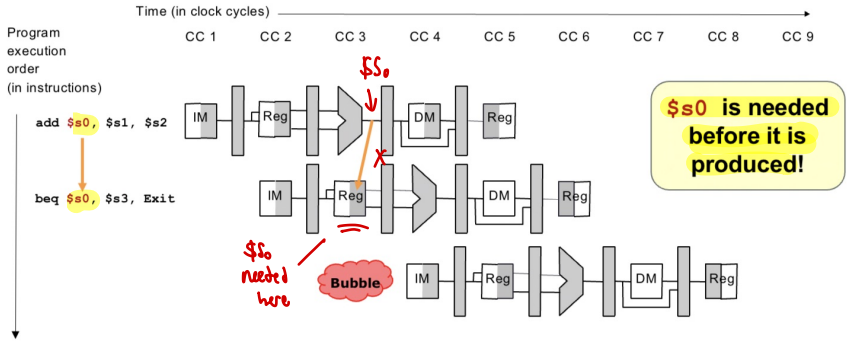


- Wait until the branch decision is known:
  - Then fetch the correct instruction
- Reduced from 3 to **1 clock cycle delay**

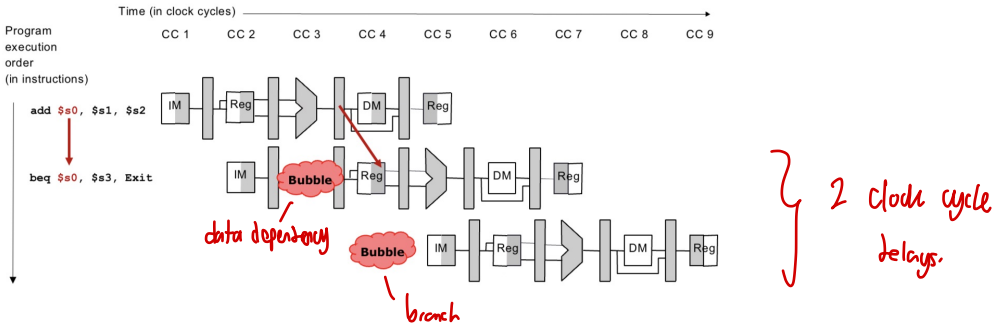
Problems:

- Decision making process is brought from MEM stage to ID stage.

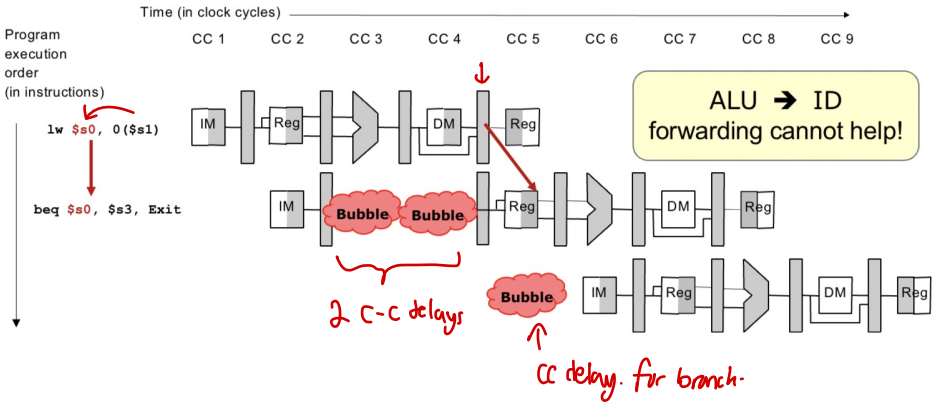
- However, if the register(s) involved in the comparison is produced by preceding instruction:
  - Further stall is still needed!



- **Solution:**
  - Add forwarding path from ALU to ID stage
  - **One clock cycle delay** is still needed



  - Problem is worse with **load** followed by **branch**
  - **Solution:**
    - MEM to ID forwarding and 2 more stall cycles!
    - In this case, we ended up with 3 total stall cycles
      ➔ no improvement!



Branch Prediction

- Simple prediction
  - All branches are assumed to be not taken.

→ Fetch the successor instruction and start pumping it through the pipeline stages.

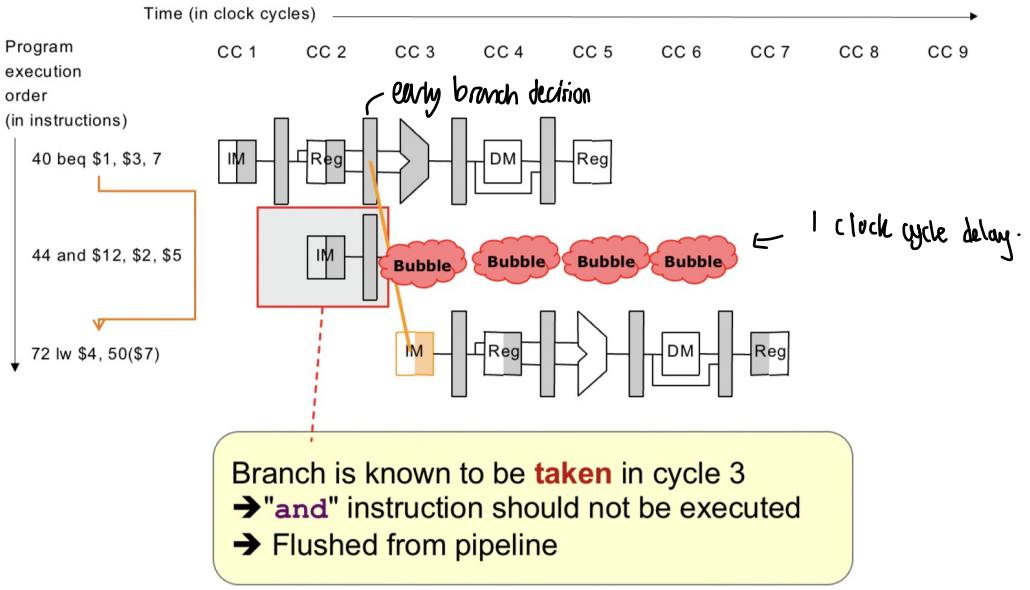
  - When the actual branch outcome is known:

Not Taken – Guessed Correctly → no pipeline stall.

Taken – Guessed wrongly → wrong instructions in the pipeline

→ Flush successor instruction from the pipeline.

→ Stalls.



Delayed Branch

- Branch outcome takes X number of cycles to be known
  → X cycles stall.
- Idea:
  - Move non-control dependent instructions into the X slots following a branch.
    - Known as the branch-delay slot.
  - These instructions are executed regardless of the branch outcome
- In MIPS processor:
  - Branch-Delay slot = 1 (with the early branch)
- Eg.

Non-delayed branch

```
or  $8, $9, $10
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
xor $10, $1, $11

Exit:
```

Delayed branch

```
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
or  $8, $9, $10
xor $10, $1, $11

Exit:
```

- The "or" instruction is moved into the delayed slot:
  - Get executed regardless of the branch outcome

  ➔ Same behavior as the original code! → Correctness should be preserved!

- Done by compiler.

- **Best case scenario**
  - There is an instruction **preceding the branch** which **can be moved** into the delayed slot
    - Program correctness must be preserved!
- **Worst case scenario**
  - Such instruction cannot be found

  ➔ Add a no-op (**nop**) instruction in the branch-delay slot
- Re-ordering instructions is a common method of program optimization
  - Compiler must be smart enough to do this
  - Usually can find such an instruction at least 50% of the time

Summary

0. Bare instr. : no. of stages + no. of instructions − 1

1. Without forwarding : all RAW instructions +2

note: RAW for BEQ

```
sll  $t3, $a0, 2
slt  $t4, $t0, $t3   +2
beq  $t4, $0, end    +2
```

"early branching"

2. branch at MEM : +3 for next instr.

3. branch at ID : +1 for next instr.

no +1 if correct

4. branch prediction : +1 if incorrect (this instr.)

5. forwarding lw instruction : +1 for next instr.

Can be +1 from being further away:

```
add  $t9, $s1, $0
add  $s1, $s2, $0
add  $s2, $t9, $0   +1
sw   $s1, 0($t1)    +1
sw   $s2, 0($t2)    +1
```

CONTROL DEPENDENCY

1. **w/o early branching & no branch prediction** +3

2. **with early branching & no branch prediction** +1

3. **with early branching & branch prediction**
- correct prediction: +0
- wrong prediction: +1

4. **w/o early branching + branch prediction** :
- correct prediction: +0
- wrong prediction: +3

5. **j instructions** : +1 at next instr.

```
sw   $t0, 0($t0)
lw   $t8, 4($t0)    address not reg.
lw   $t9, 0($t2)
add  $t8, $t8, $t9
sw   $t8, 4($t0)    address not reg.
addi $t0, $t0, 8
```

The jump (j) instruction incurs one stall cycle, if computation of the next PC value is done at ID stage (stage 2).

DATA DEPENDENCY

1. **without data forwarding**: +2

2. **with data forwarding**: +0

EXCEPT TWO SPECIAL CASE

i) when the current instruction is bne/beq and one of the operand is being written to in the last instruction (see slide 2) : +1

```
add  $s2, $zero, $zero   #inst B
bne  $s2, $zero, done    #inst C
```

(with correct branch prediction?) → additional +1 on beq.

ii) when the previous instruction is a lw/sw and I need to use the written register in the current instruction : +1

```
lw   $s4, 0($t4)    # I11
sub  $s3, $s3, $s4  # I12
```