Hazards

Dependences in Programs

- **Data dependence**

  Example:

```
lw    $1, 200($2)
add   $3, $4, $1
```

  `add` can't do **ID** (i.e., read register $1) until `lw` updates `$1`

- **Control dependence**

  Example:

```
bne   $1, $2, target
add   $3, $4, $5
```

  next **IF** can't start until `bne` completes the comparison

- **These dependences may cause the pipeline not be fully filled**
  - Execution stops to wait for **data** or **control** to be produced
  - next instruction cannot be executed in next cycle

Pipeline Hazards

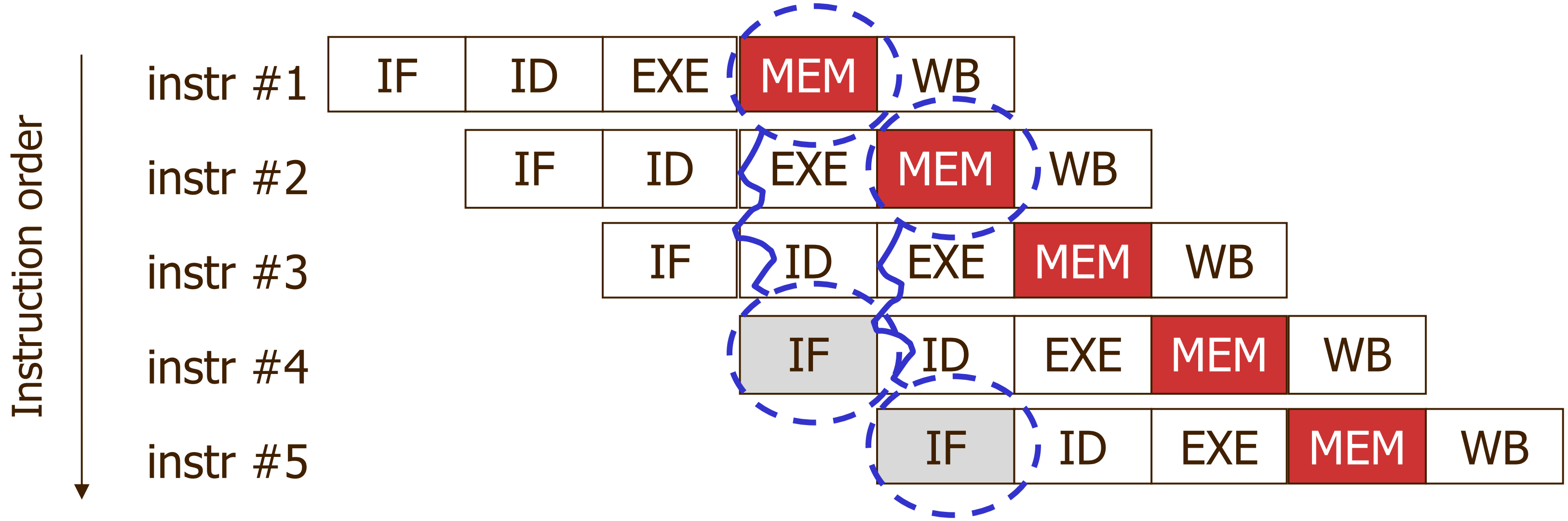
- **Hazards** are situations in pipelining when the next instruction cannot be executed in the following clock cycle.

- **Three types of pipelined hazards**
  - **Structural hazards:** hardware cannot support the combination of instructions to execute in the same clock cycle. Different instructions compete for the same hardware.
  - **Data hazards:** an instruction depends on the results of a previous instruction still in the pipeline.
  - **Control hazards:** which instruction to execute next depends on the results of a previous instruction still in the pipeline. Branch instruction must complete before we know the next instruction.

- **Hazards can always be resolved by waiting.** But this slows down the pipeline.

Structural Hazards (1): Memory

- If instructions #1 and #2 are load operations, instruction fetch (#4, #5) and data load (#1, #2) conflict for memory access

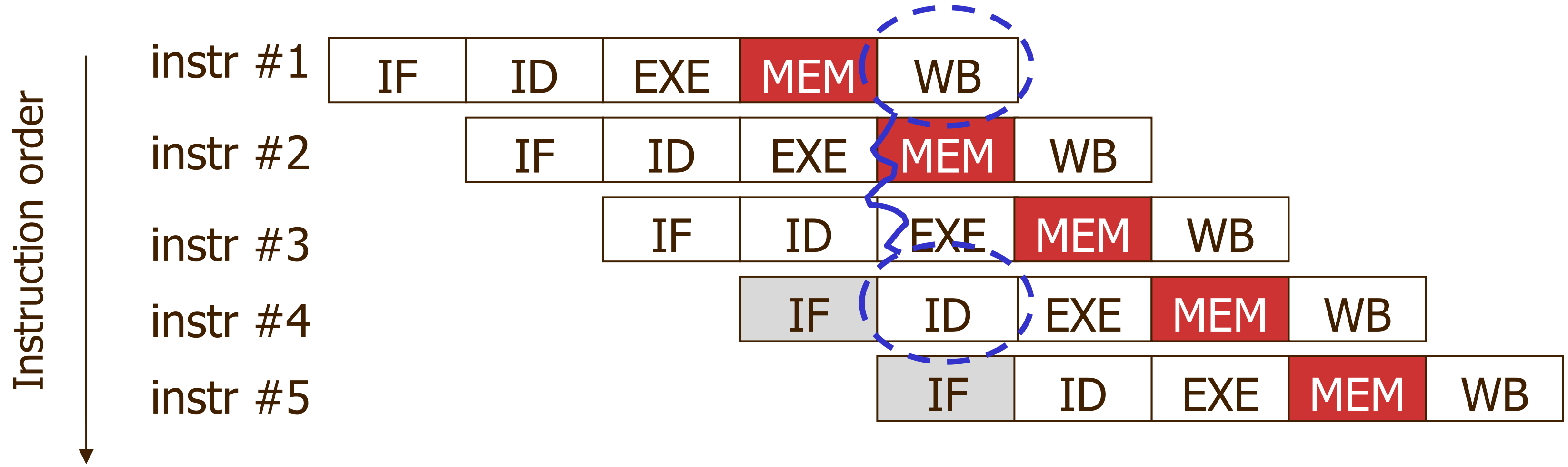


Read same memory twice in same clock cycle

- Solution:
  - Add memory ports to allow parallel accesses to independent addresses
  - Separate Instruction memory from data memory

Structural Hazards (2): Registers

- If instr. #1 is a load operation, it wants to write while instr. #4 wants to read the register file



Can't read and write to registers simultaneously

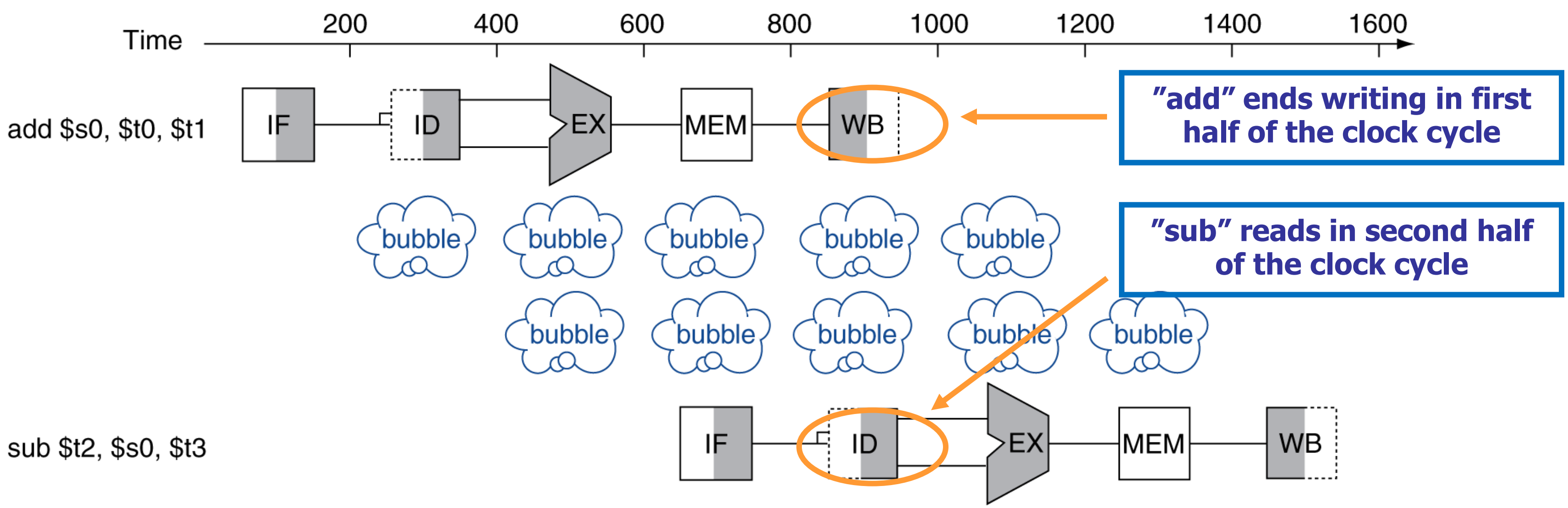
- Solution:

  Fact: Register access VERY fast. Takes half the time of ALU stage or less

  - always Write to registers during 1st half of each clock cycle
  - always Read from Registers during 2nd half of each clock cycle
  - Register file supports Write and Read during same clock cycle (in this order)

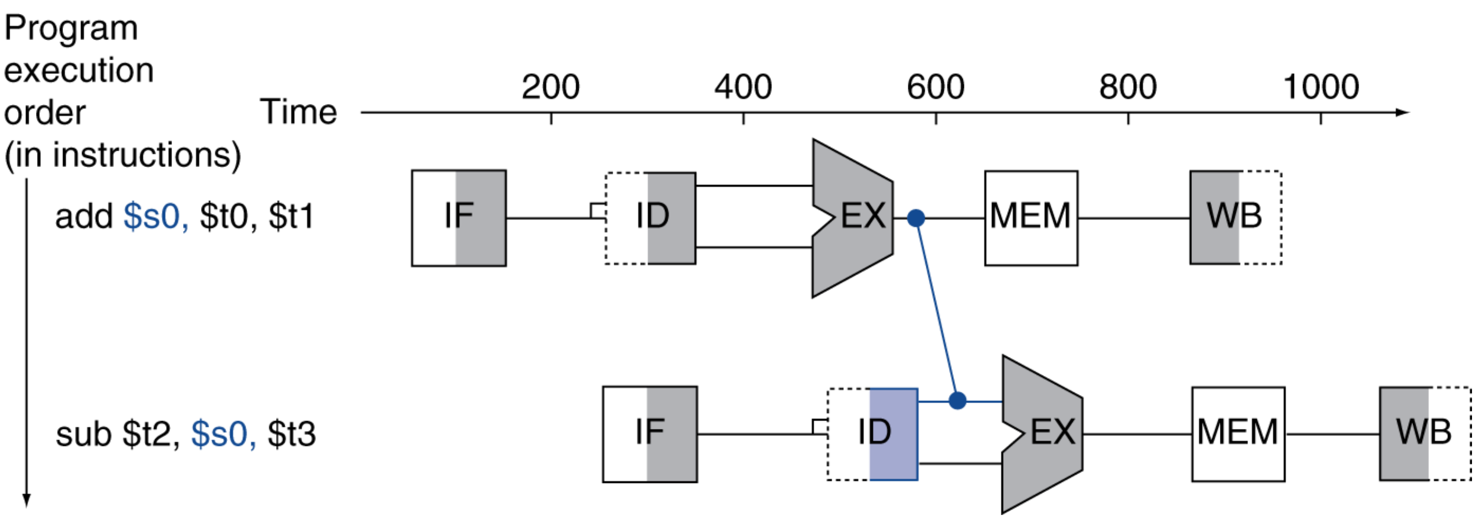
Data Hazard (1)

- Later instruction tries to read an operand before earlier instruction writes it
- Example: `add $s0, $t0, $t1`
  `sub $t2, $s0, $t3`
- The "sub" instruction needs to wait until the "add" instruction has finished writing $s0 before it reads from $s0.

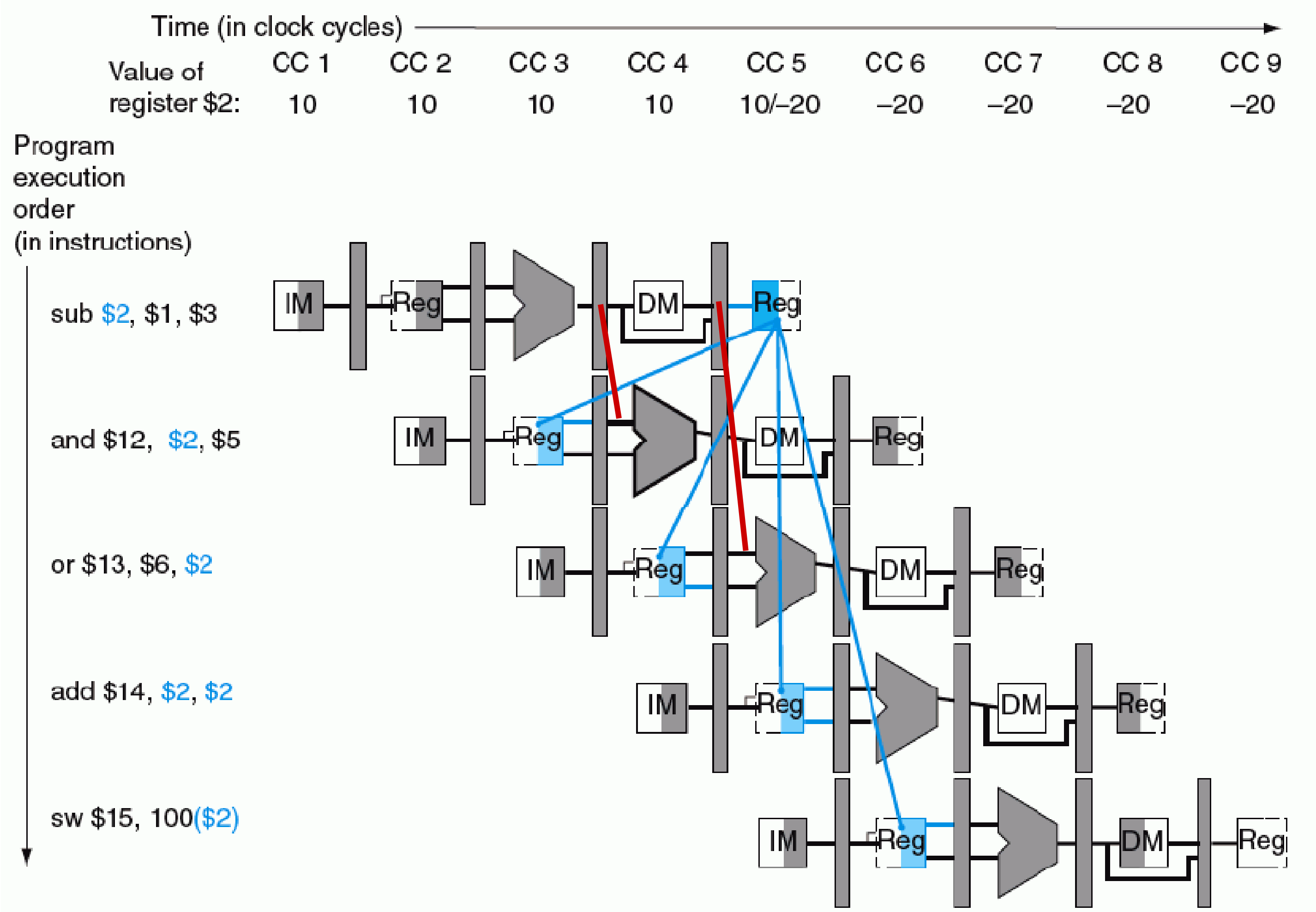


- a bubble or **pipeline stall** is a delay in execution of an instruction in an instruction **pipeline** in order to resolve a hazard.

- **Forwarding partially solves the data hazard problem**:
  - "add" has the result for \$s0 right after stage 3 (EX)
  - If we have a "wire" that "forwards" the value of \$s0 from the EX stage of "add" to "sub", then "sub" does not need to wait!

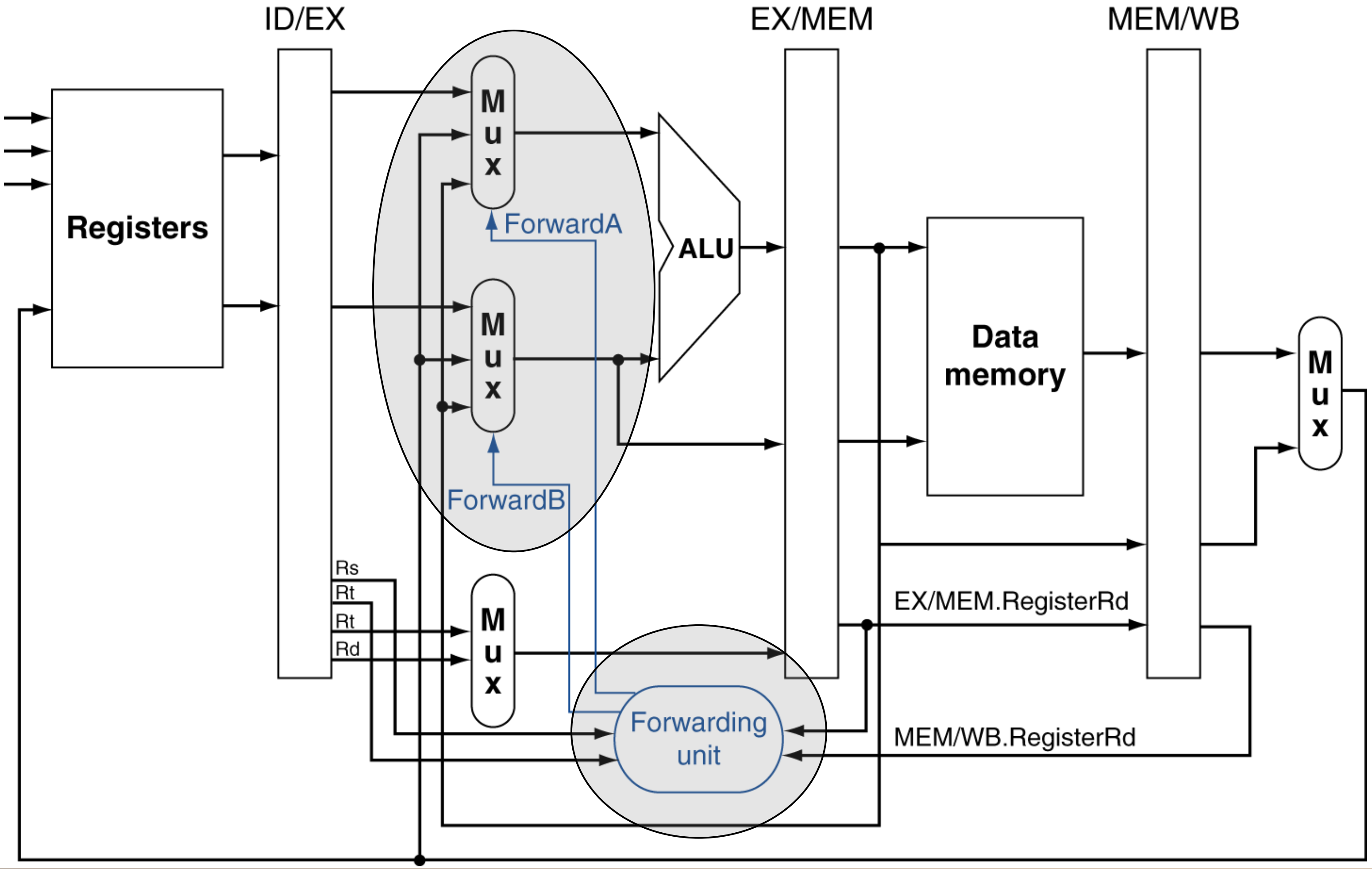


- From the figure the decision is simple (required "forwardings" are represented by the two red lines):

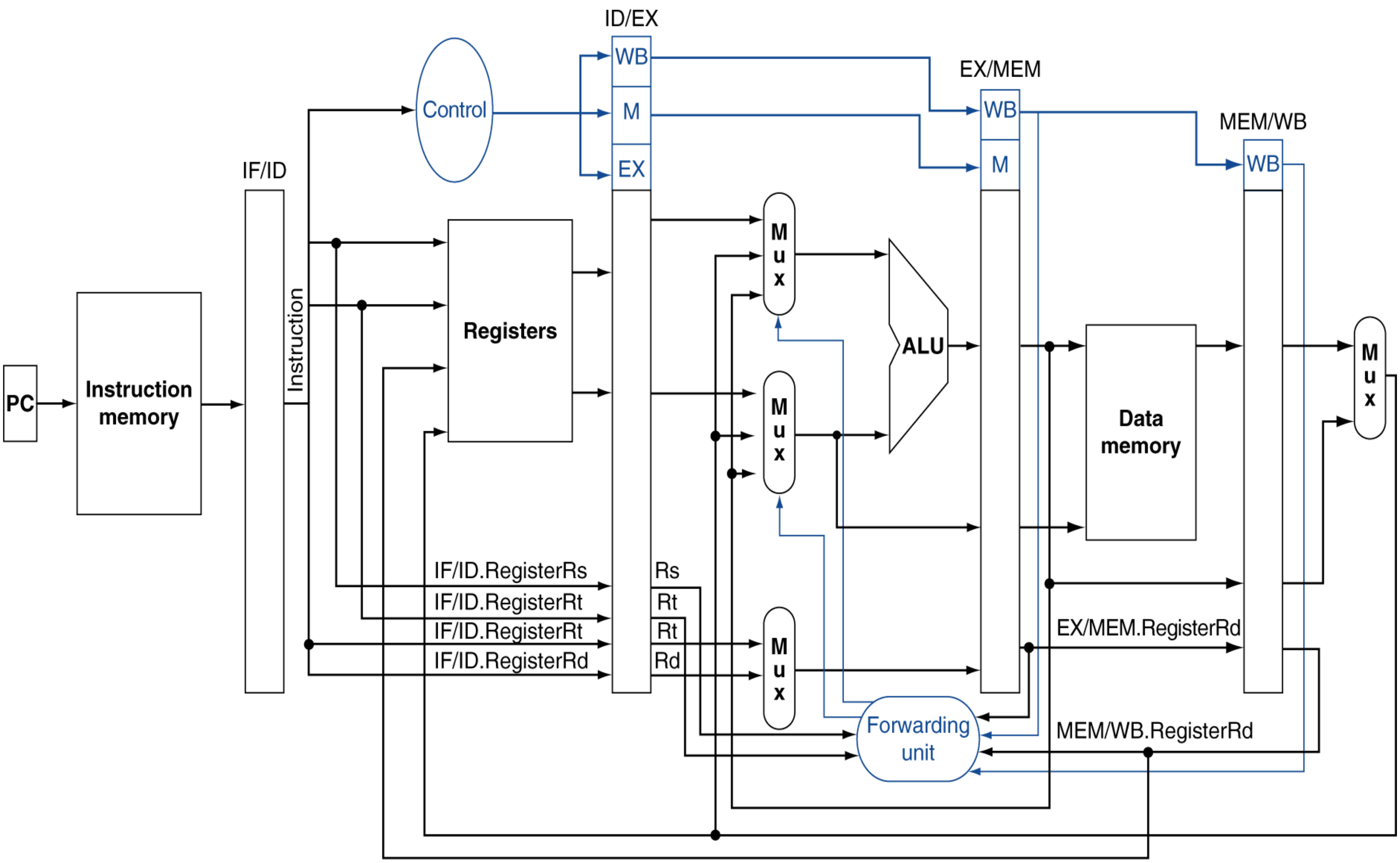


Forwarding circuits

- Forwarding always takes place to EX stage
  - Implementing these conditions in a **forwarding control unit**
  - Using **two multiplexers** to decide what is the input of operands A and B of the ALU

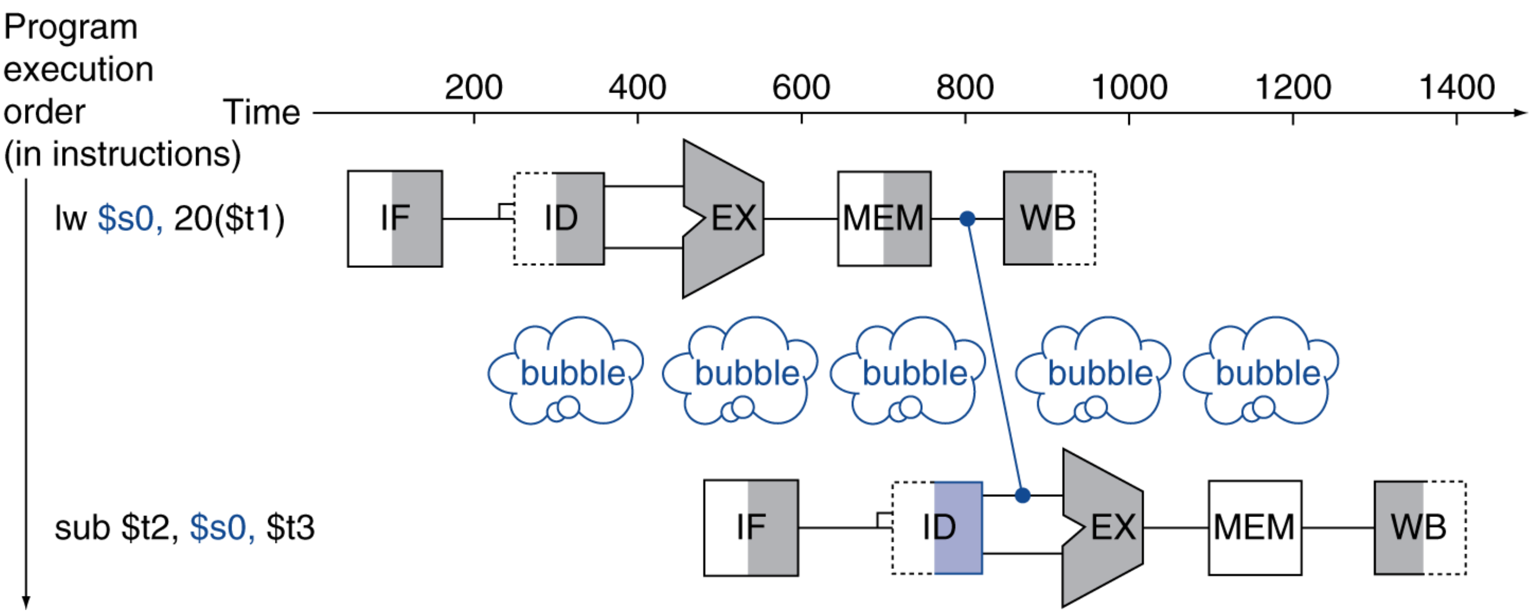


The complete datapath with forwarding

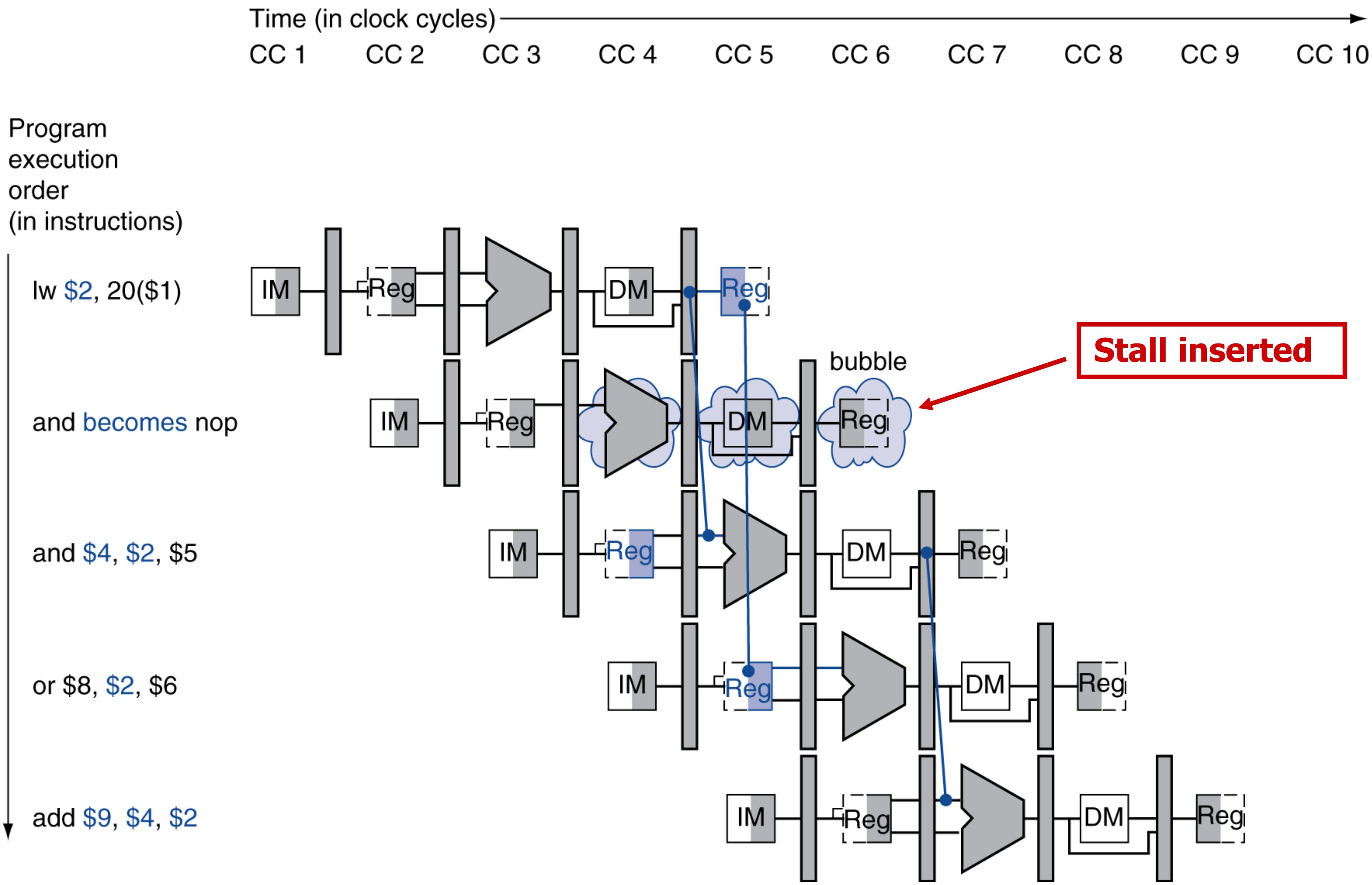


Data Hazard (2): Load-Use Hazard

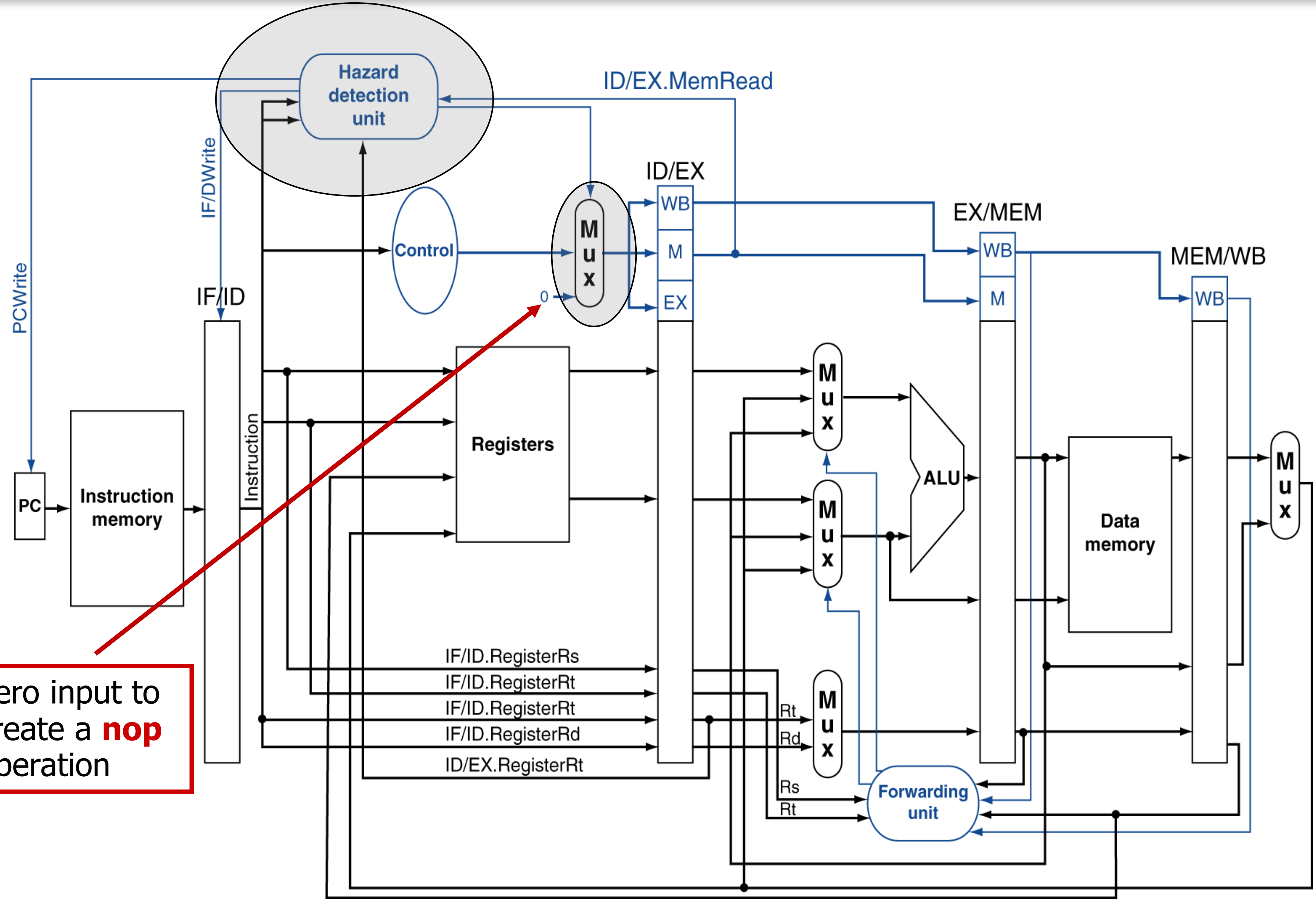
- **Even with forwarding we can not always solve the problem** (i.e. avoid stalls)
- Example: `lw $s0, 20($t1)`
  `sub $t2, $s0, $t3`
  - The "lw" instruction produces value for $s0 in stage 4 (MEM),
  - The "sub" needs $s0 before its stage 3 (EX),
  - We can't forward back in time!



A stall in the pipeline



The final datapath with forwarding and hazard detection



Zero input to create a **nop** operation

Solving Data Hazards with Code rearrangement

- Consider this code sequence

```
a = b + c;
d = b + e;
```

Assume **a** to **e** are stored in memory address `0($t0)`, `4($t0)`, `8($t0)`, `12($t0)` and `16($t0)` respectively. Assume **forwarding is used**.

stall (before `add $t3, $t1, $t2`)
stall (before `add $t5, $t1, $t4`)

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

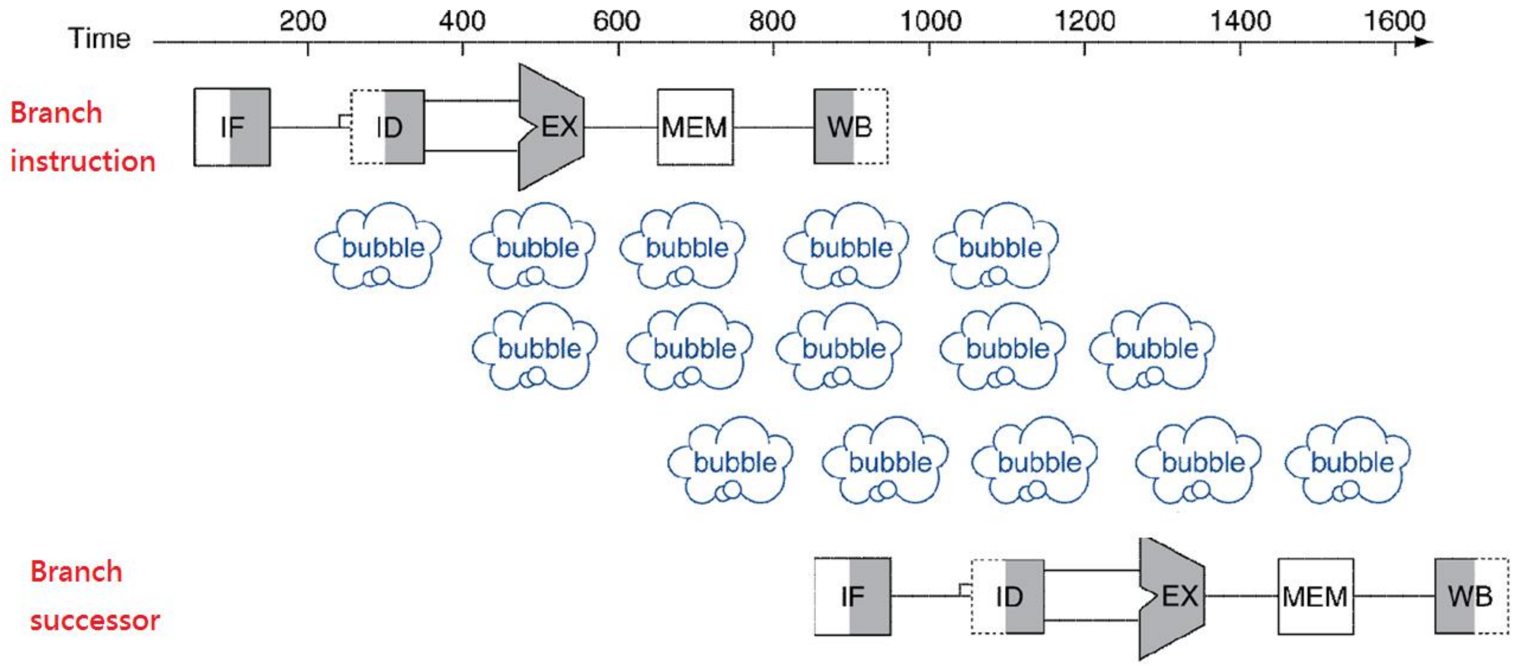
15 cycles

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

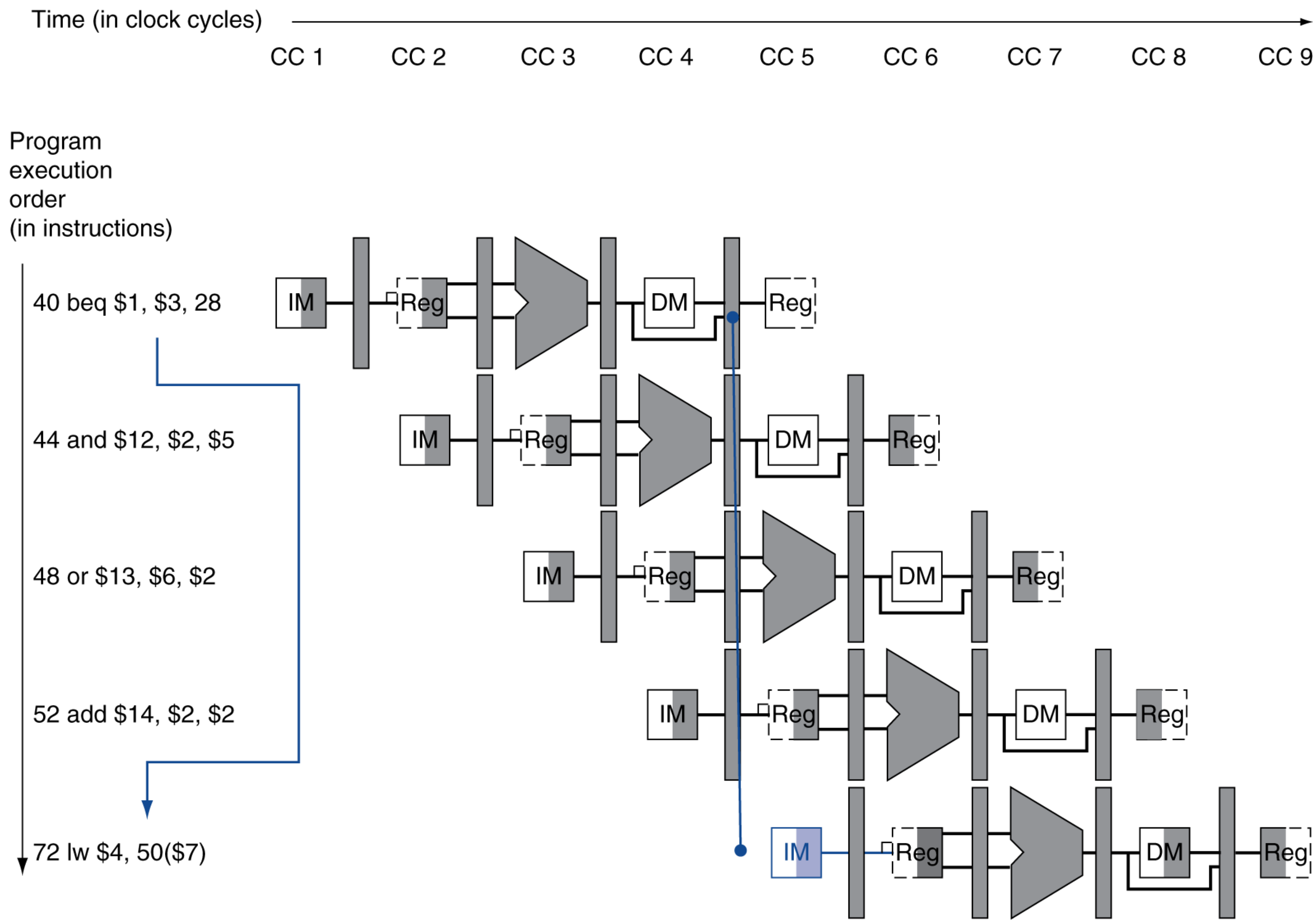
11 cycles

Control Hazards

- **Control Hazards** arise from the pipelining of branches and other instructions that change the Program Counter.
  - E.g. branch instruction needs **three cycles of stalls** before fetching the next instruction



Impact of the branch instruction on the pipeline



Possible solutions to the control hazard

- **Wait until the branch outcome has been determined**
  - Fetch instruction after the branch outcome has been clear
  - This solution always solves the problem (i.e. program runs correctly), but it imposes performance penalty (3 cycles of delay)
- **Reduce branch delay via Hardware**:
  - Compare the registers and compute target earlier in the pipeline
  - Add hardware to do it in the **ID** stage
- **Speculate on (predict) the branch decisions**:
  - **Static branch prediction**
  - **Dynamic branch prediction**
- **Delayed branch**
  - Reduces the branch penalty
  - Schedule independent instruction to fill the branch delay slots of the branch instruction

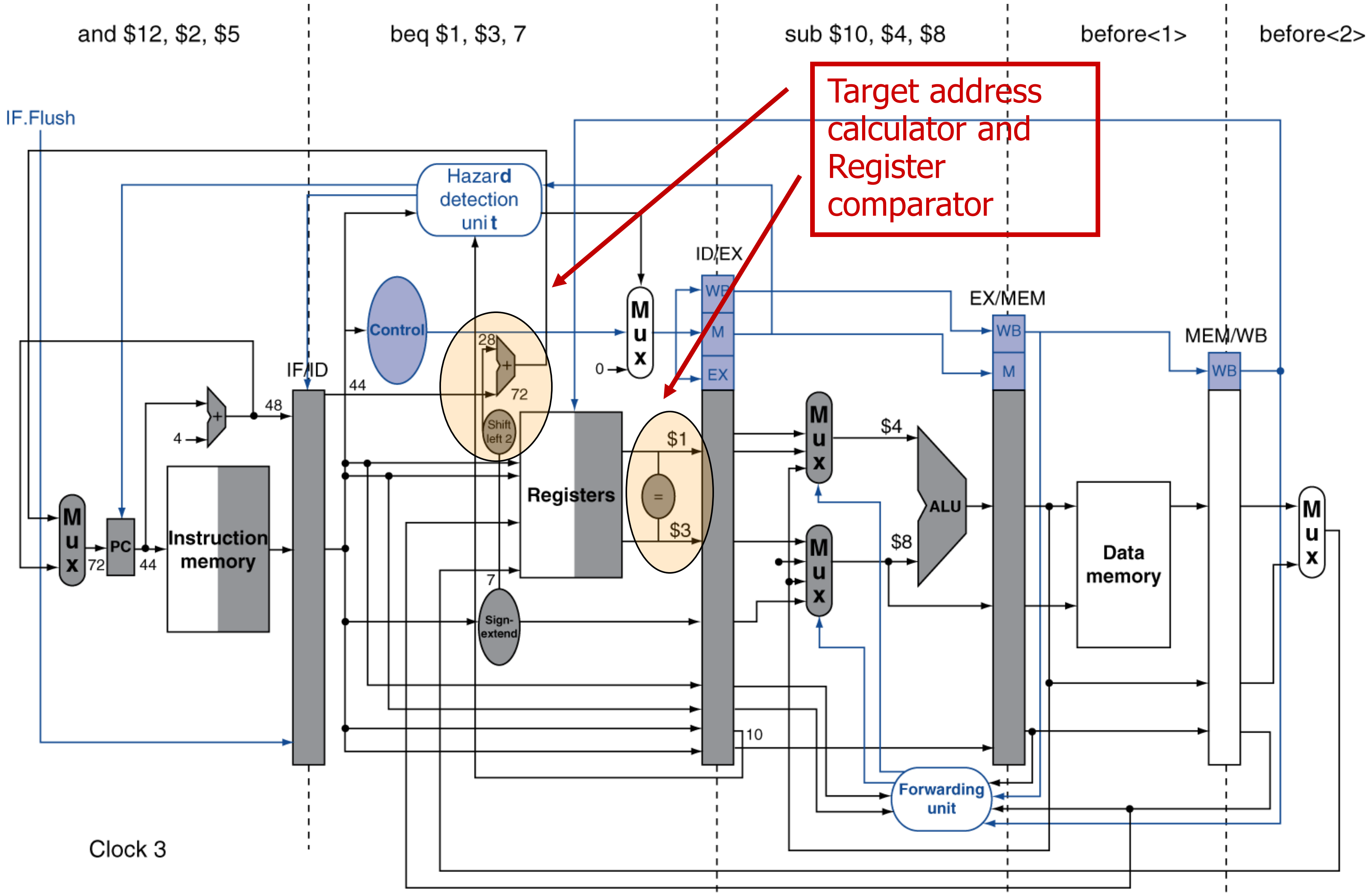
Reducing Branch Delay via Hardware

- Add hardware to the MIPS pipeline to determine the branch result in the **ID stage**
  - Target address calculation requires an **adder**
  - Register comparator

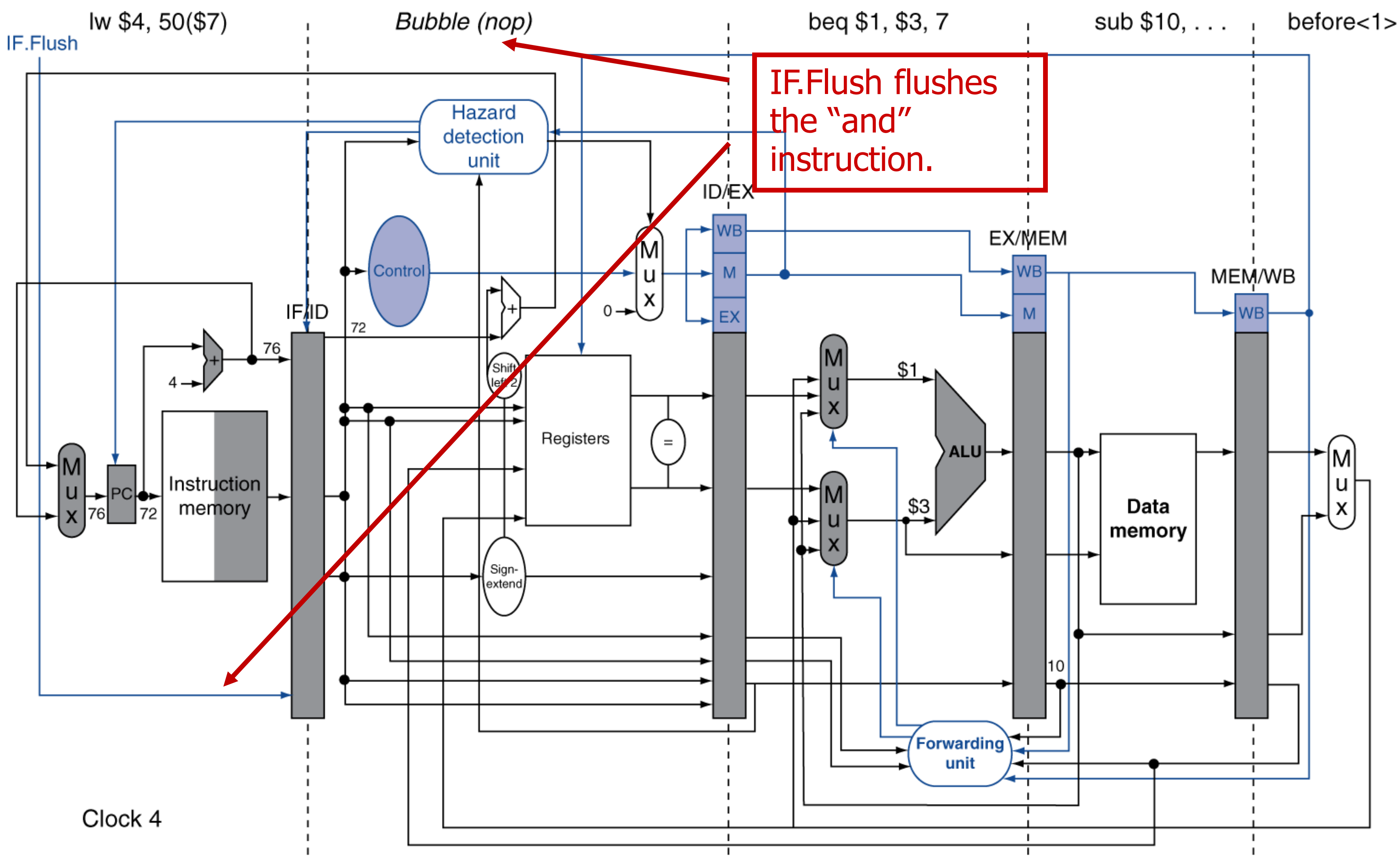
- An example (assume branch taken)

```
36: sub $10,$4,$8
40: beq $1,$3,7  #PC relative branch to 40+4+4*7=72
44: and $12,$2,$5
  :       :
72: lw $4,50($7)
```

An example (branch taken)



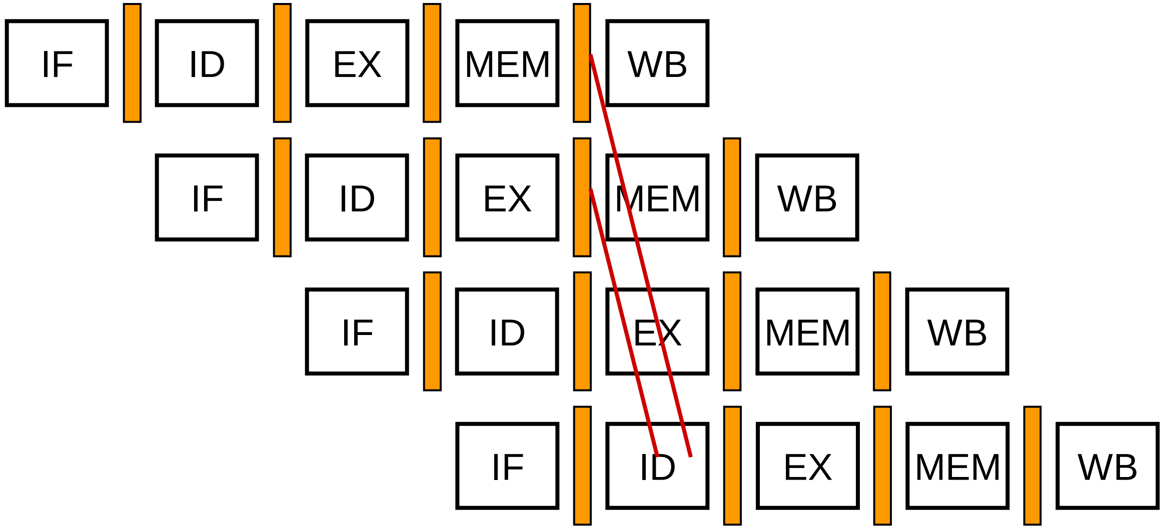
An example (branch taken)



Data hazards for branches (example 1)

- Branch instruction depends on data value (in register) to make decision, therefore it is prone to data hazards.

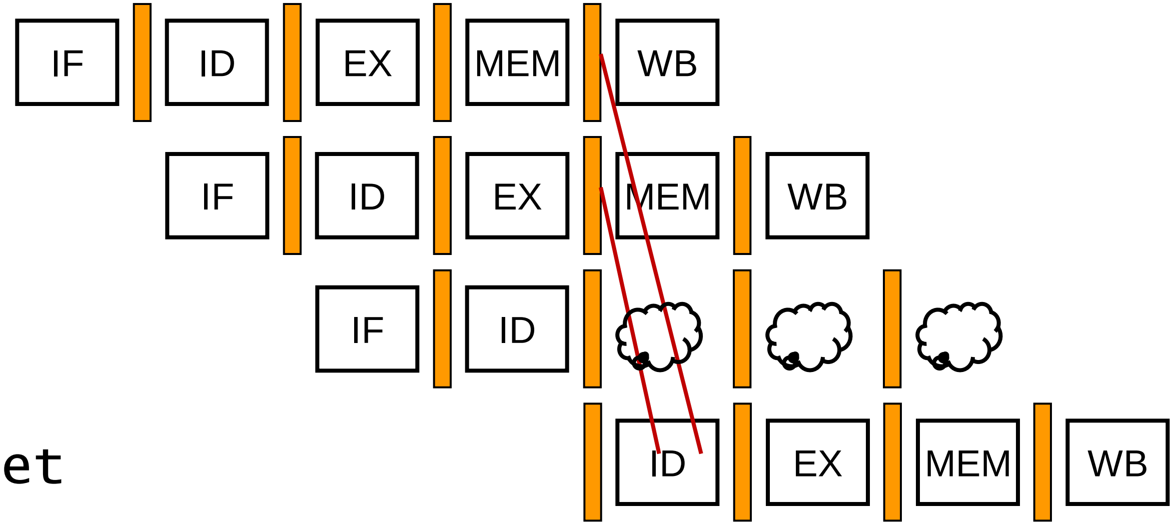
```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```



- If the comparison registers are to be written by the 2nd or by the 3rd preceding instructions, forwarding can pass the values to the branch instruction in time (i.e. the branch instruction don't need to stall)

Data hazards for branches (example 2)

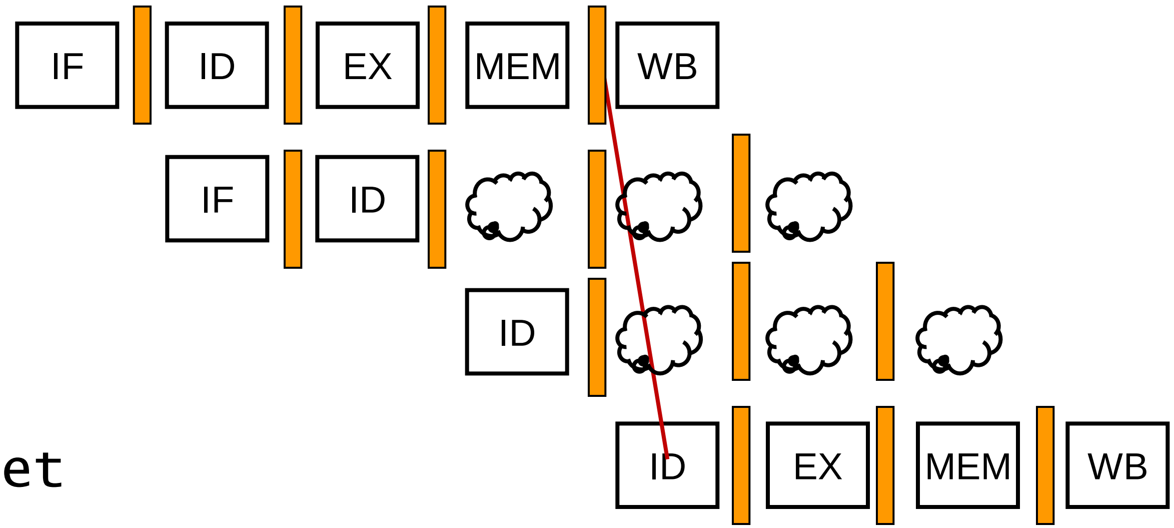
```
add $1, $2, $3
add $4, $5, $6
beq stalled
beq $1, $4, target
```



- If the comparison registers are to be written by the immediate preceding instruction or by the 2rd preceding instruction, forwarding can NOT pass the values to the branch instruction in time (i.e. the branch instruction need to stall)

Data hazards for branches (example 3)

```
lw  $1, addr
beq stalled
beq stalled
beq $1, $0, target
```



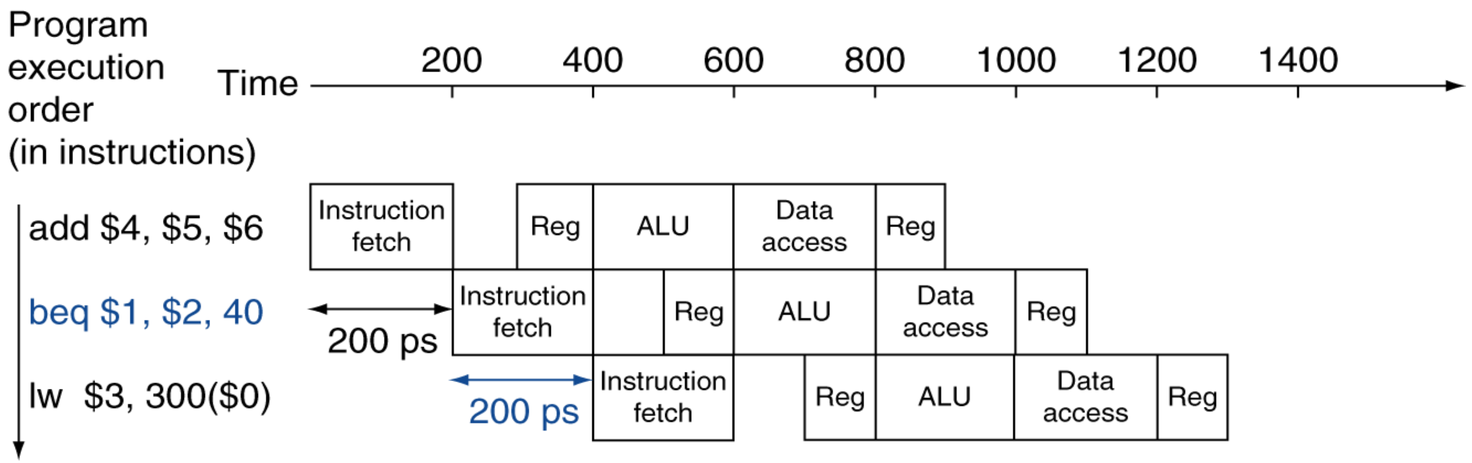
- If the comparison registers are to be written by preceding load instructions, forwarding may NOT be able to pass the values to the branch instruction in time (i.e. the branch instruction may need to stall)

- **Predict the outcome of a branch in a static manner**
  - either predict every branch is always taken,
  - or predict every branch is always not taken.

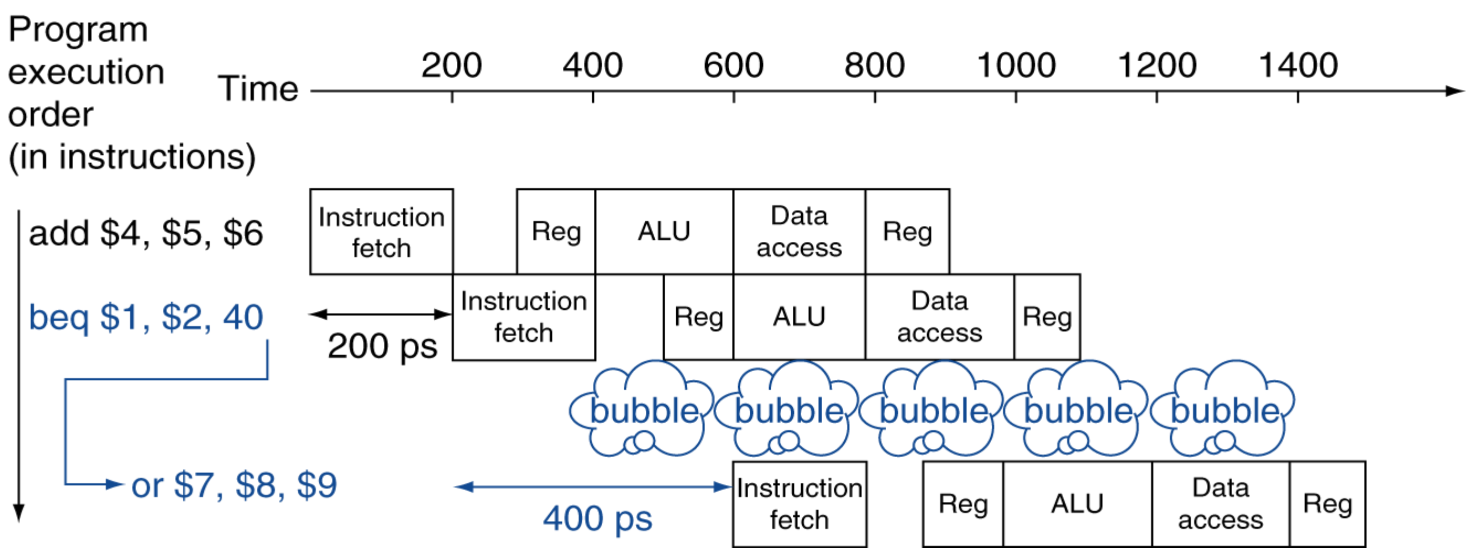
- In MIPS
  - Always predict branch not taken (why?)
    - With hardware improved pipeline, branch target is not available until **ID** whereas PC+4 is already available in **IF**
  - Fetch instruction right after branch, no delay if prediction is correct

MIPS with static Branch prediction (NOT TAKEN)

Prediction correct



Prediction incorrect, need to fetch the branch target and flush the wrong “lw”



Dynamic Branch Prediction

- **Dynamic branch prediction (the idea)**
  - Look up the address of the branch instruction to see if a branch was taken/not taken last time (assume the current decision will be highly correlated with the decision of last time),
  - Fetch new instruction from the same place as last time.

- **Dynamic branch prediction (implementation)**
  - Uses a small **Branch prediction buffer** (aka **branch history table**), to store recent branch outcomes (taken/not taken),
  - The branch prediction buffer is **indexed by** lower portion of **recent branch instruction addresses** .

Key Concepts to Remember

- **Pipelining improves the throughput** by allowing reuse of functional units by different instructions
- Pipelining allows an **instruction to complete in each clock cycle**, but it requires a very careful design and additional registers to store intermediate results between pipeline stages
- **Pipelined Control** is implemented like single cycle control with needed control signals are **forwarded down the pipeline**
- Concurrence between instructions in the pipeline may cause
  - **Data Hazard:** data is needed by an instruction before it is produced by a previous one
  - **Structural Hazard:** a hardware unit is needed by an instruction while another is still using it
  - **Control Hazard:** the next instruction cannot be determined in the next clock cycle
- **Hazards can always be solved by delaying (inserting bubbles)**

Key Concepts to Remember

- **Structural hazard is solved by:**
  - Separating the instruction memory from the data memory
  - Writing to the register file in the first half of the clock cycle and reading from it in the second half
- **Data hazard is solved by:**
  - **Forwarding/Bypassing**
  - **Inserting bubbles**
- **Control hazards are solved by:**
  - **Hardware:** add comparator to complete the comparison earlier
  - **Speculation:** guess if the branch is taken or not
  - **Delay the branch:** fill the bubbles with useful work that is independent of the branch