The Pipelined Processor

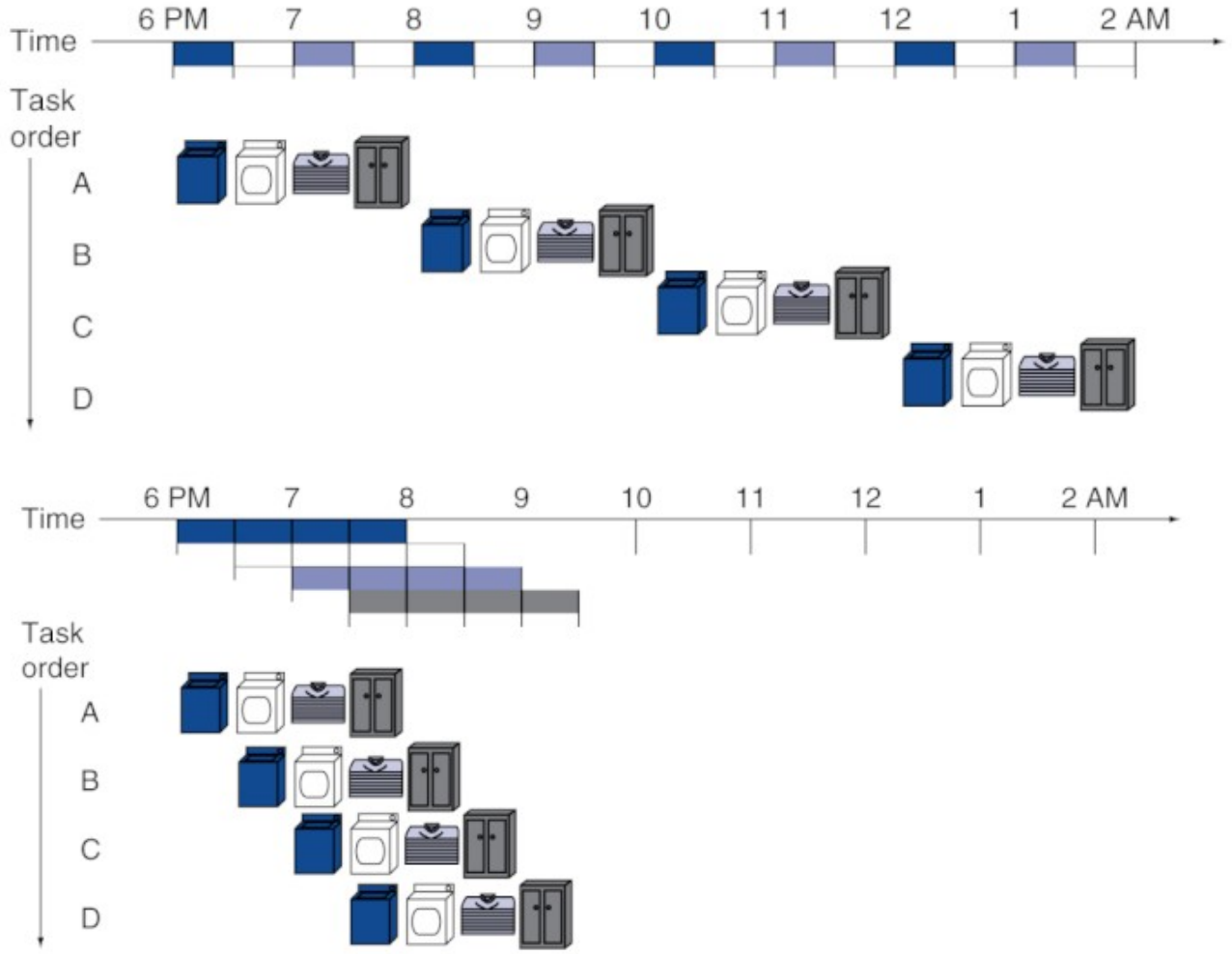
Sections 4.4, 4.5

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

- Four loads:
  - Sequential execution

- Non-stop:
  - Speedup achieved with pipelining



MIPS Pipeline

- Five stages, one step per stage
  1. **IF**: Instruction fetch from memory
  2. **ID**: Instruction decode & register read
  3. **EX**: Execute operation or calculate address
  4. **MEM**: Access memory operand [not always]
  5. **WB**: Write result back to register [not always]

Pipelining and MIPS ISA Design

- MIPS ISA designed for pipelining
  - One instruction size [32-bits]
    - Easier to fetch in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Simple Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands

Pipeline Performance

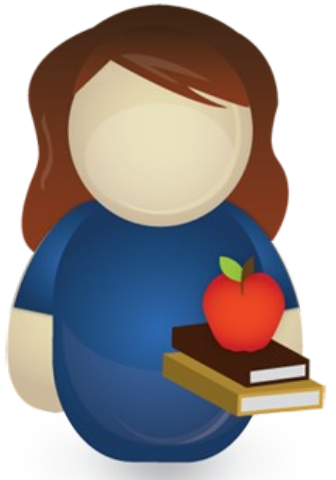
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Table below shows the **single-cycle datapath** instruction execution time

Instr	Instr fetch	Reg read	ALU op	Memory access	Reg write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

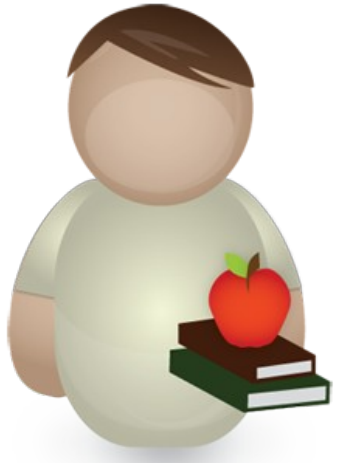
What is the clock period of this single-clock-cycle processor?

Critical path is the instruction that has the longest execution time --> clock cycle = 800ps
Load instruction in MIPS

Hi Jack, I missed the last lecture.
What did Ahmed teach?



Ahmed taught us how to do our laundry!



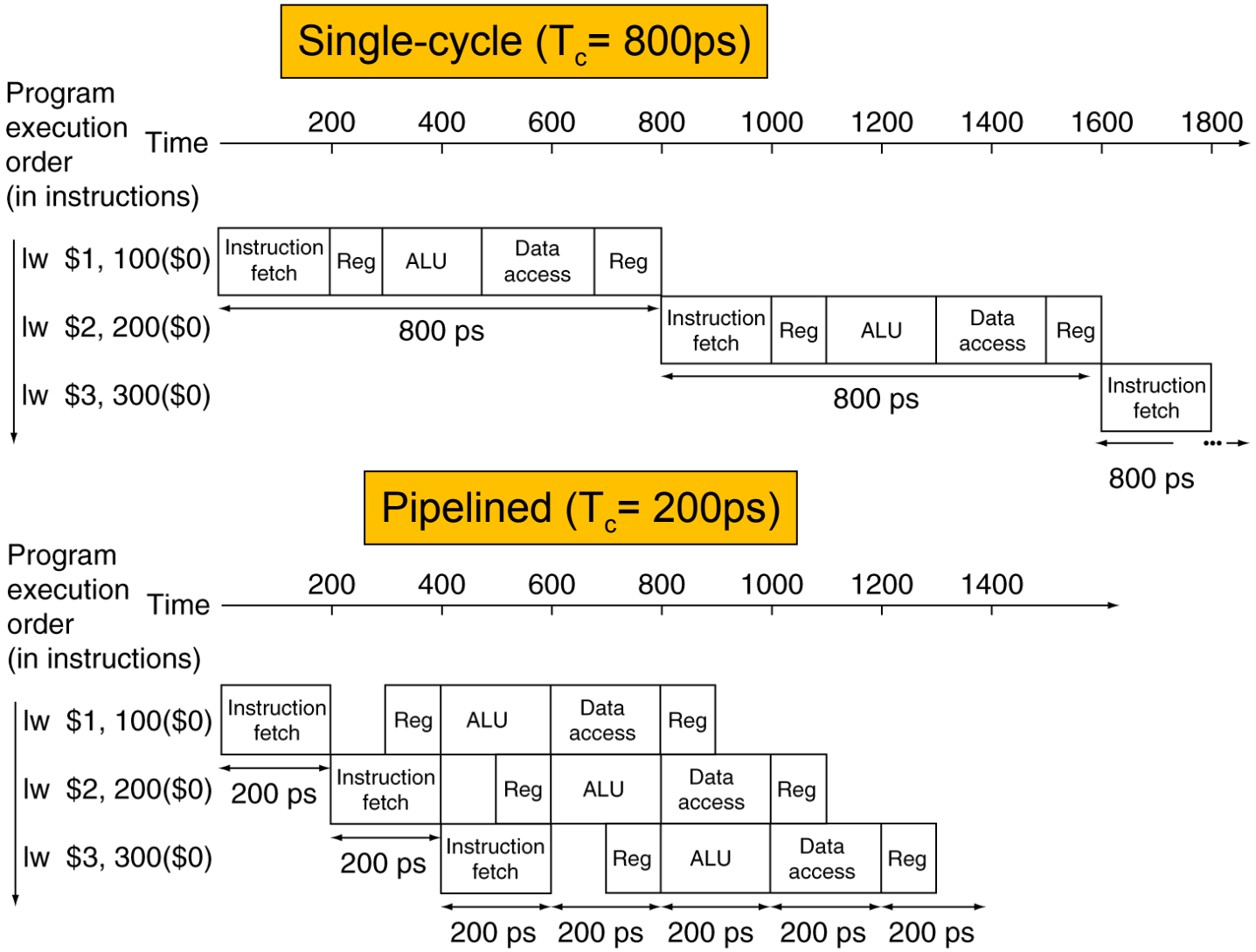
Question

Assume that individual stages of the MIPS processor datapath have the following latencies

IF	ID	EX	MEM	WB
250ps	350ps	150ps	300ps	200ps

(a) What is the clock cycle time in a pipelined and non-pipelined processor?
(b) What is the total latency of an LW instruction in a pipelined and non-pipelined processor?

Pipeline Performance Comparison



What is the clock cycle of this pipelined processor?

General rule: ***The clock cycle of pipelined architectures has to be larger than the time needed by the longest stage***

Pipeline Speedup

- If all stages are balanced
  - i.e., all **N** pipeline stages take the same time **T** and we have **I** instructions
  - Single clock cycle execution time = I * (NT)
  - Pipelined procesor execution time = NT + (I-1)T
  - speedup factor = I*N*T / [(N + I -1)T] ~N

- If not balanced, speedup is less
- Speedup due to increased throughput
  - Instruction Latency (time for each instruction) does not **decrease**

So, Pipelining is Good, right?

What if?

- The processor needs to fetch an instruction from the memory at stage 1 and you need to execute a memory load instruction?

- The Processor executing this instruction sequence

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

- You are executing a branch instruction and the branch is taken?

Pipeline Hazards

Situations that prevent starting the next instruction in the next cycle

1. ***Structure hazard***
   - Two instructions need to access the same resource at the same time, e.g., memory
2. ***Data hazard***
   - An instruction needs data that from a previous instruction in the pipeline, e.g., incomplete memory load
3. ***Control hazard***
   - *Changing instruction sequence (branch, call, ..) invalidates the pipeline fetch/decode*

Structure Hazards

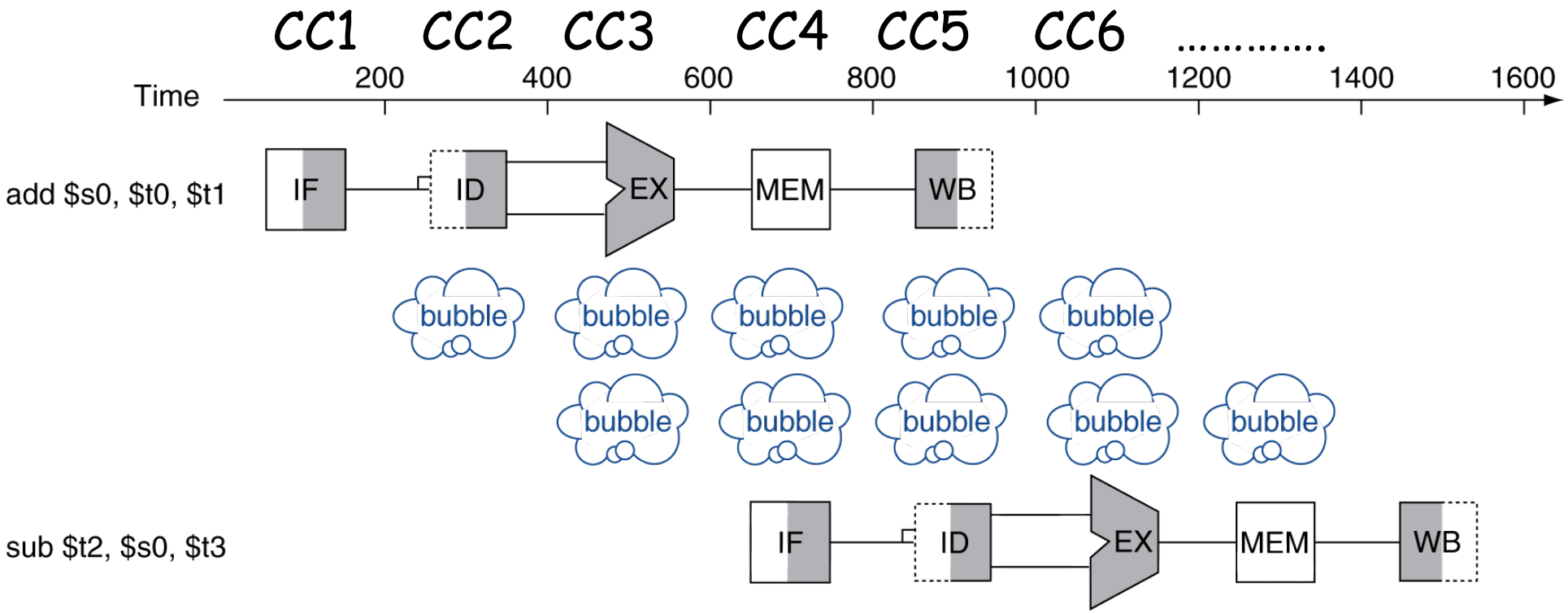
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"

- ***Solution:***
  - pipelined datapaths require separate instruction and data memories

Data Hazards

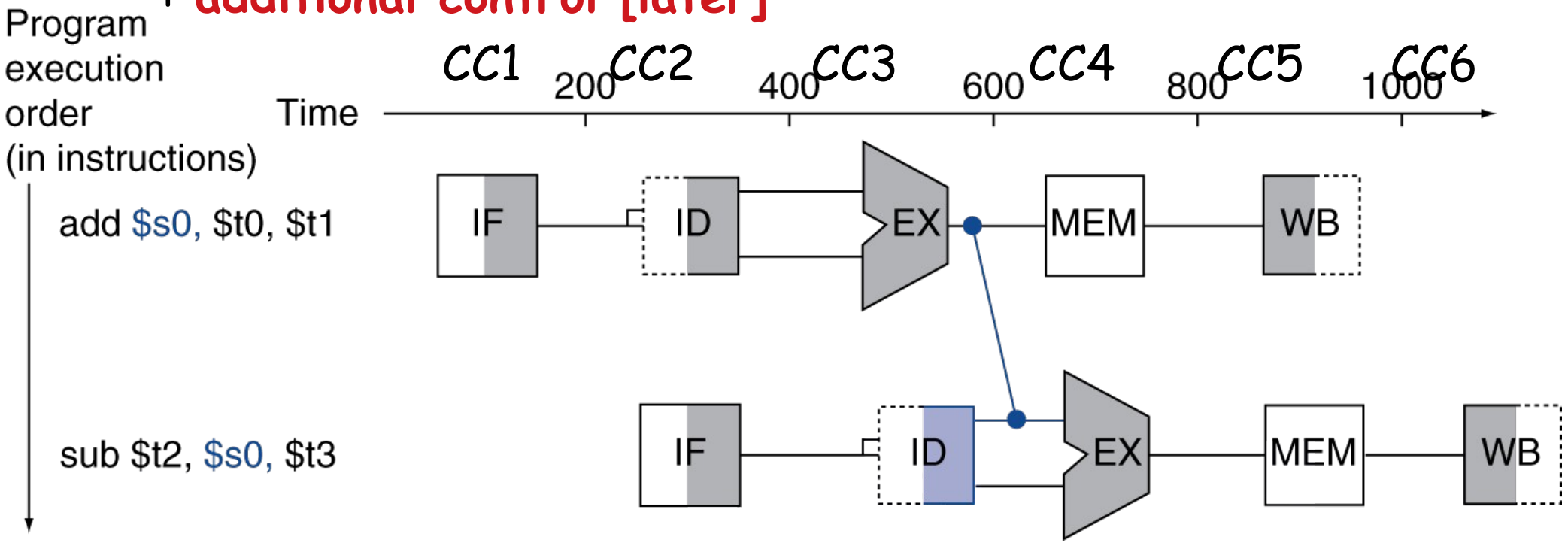
- An instruction depends on completion of data access by a previous instruction that is still in the pipeline

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



Solution 1: Forwarding (aka Bypassing)

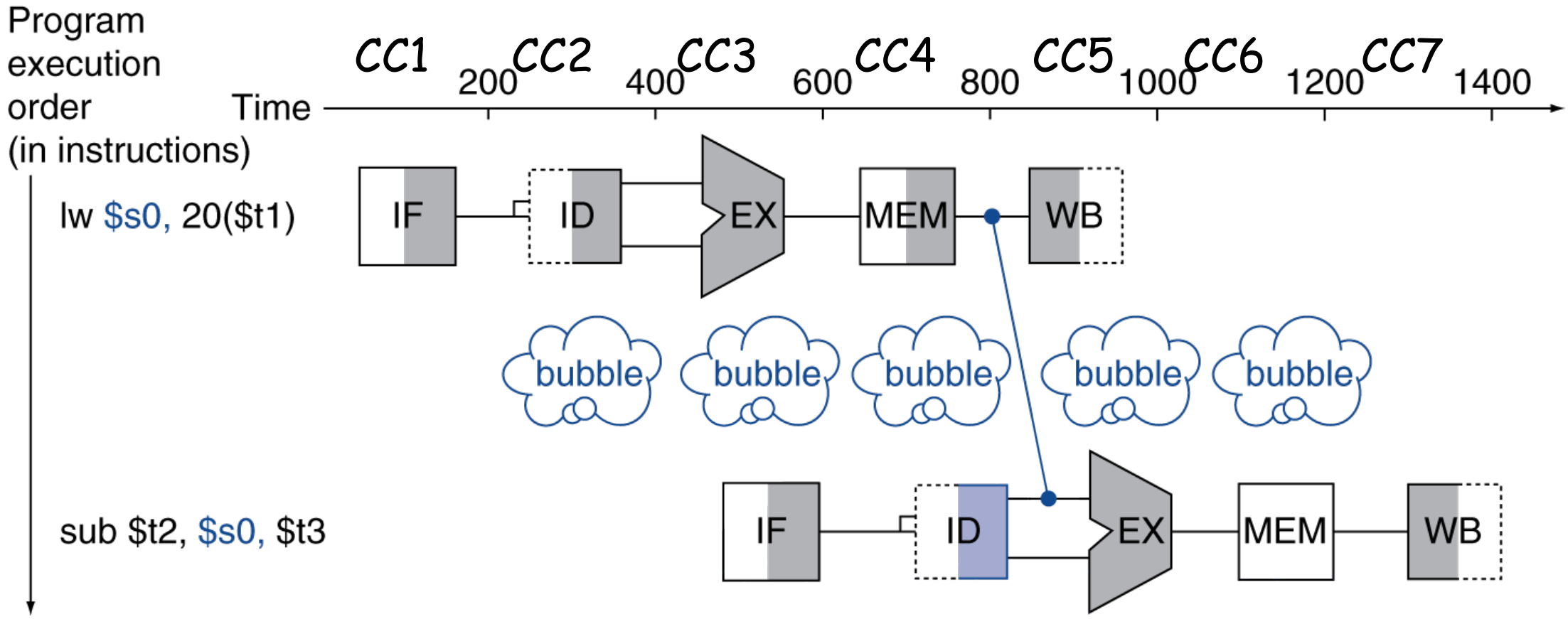
- *HW connections that allow using the data when it is available (after computing or memory load)*
  - Don't wait for it to be stored in a register
  - Requires ***extra connections*** in the datapath
    + **additional control [later]**



When does forwarding Fails?

Load-Use Data Hazard

- loaded data won't be available before 2 stages
  - Can't always avoid bubble/stalls by forwarding
  - Can't forward backward in time!



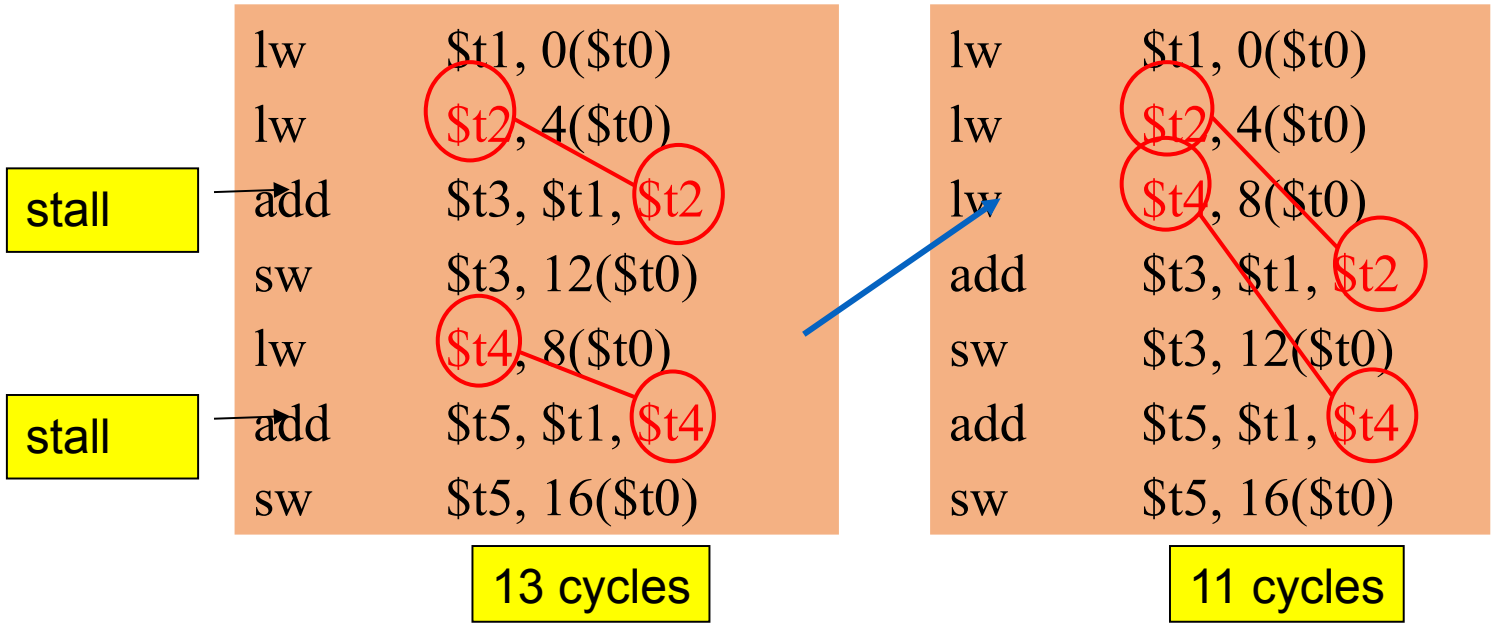
Code Scheduling

A possible solution to avoid load-use scenario

- ***Reorder instructions*** *to avoid use of load result in the next instruction*
- C code for A = B + E; C = B + F;

Software Solution

How many clock cycles are needed to exexute the code?



Discussion

- Explain two techniques to reduce the impact of data hazards in MIPS pipeline.
  - Forwarding (hardware)
  - Instruction scheduling (software)

- Explain what is meant by structure hazard. Provide an example of structure hazard.

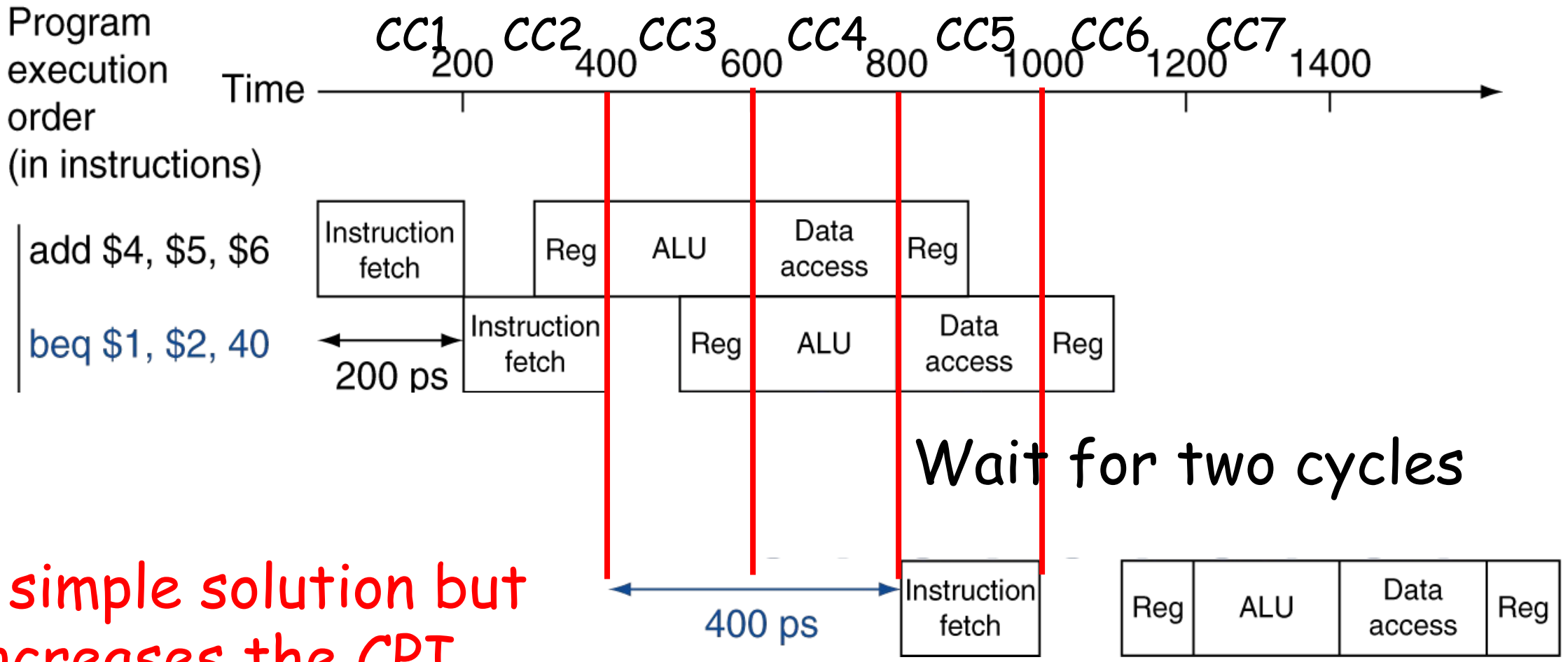
- Identify hazards in a given code

Handling control hazards

- **More challenging** than data and structure hazards
- We will explore some solutions
    - stall on branch! (trivial-do nothing)
    - Optimized waiting!
    - Prediction
        - *Static*
        - *Dynamic*

Stall on Branch

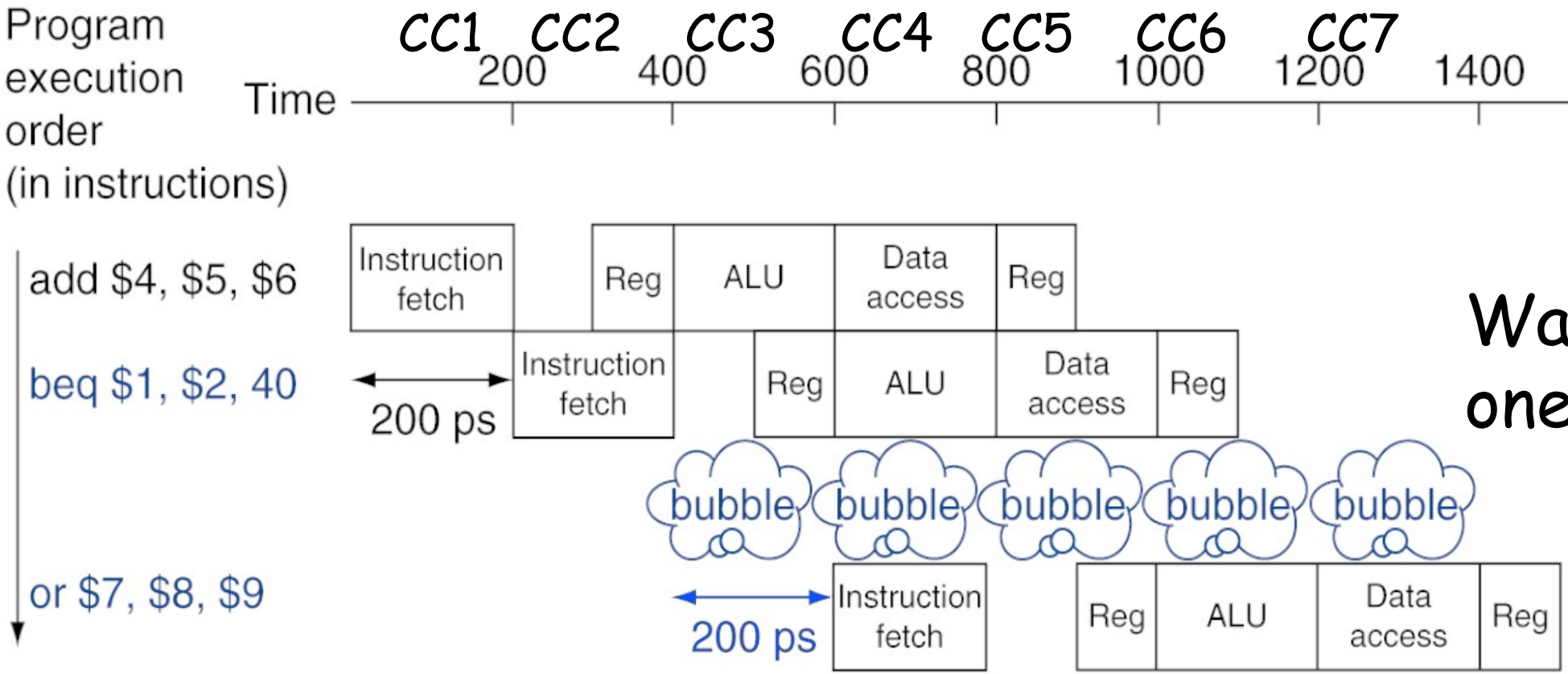
- Wait until branch outcome is determined before fetching next instruction



A simple solution but Increases the CPI

Optimized Stall on Branch

- Decide while decoding (additional HW)
  - NOT while executing



Wait for one cycle

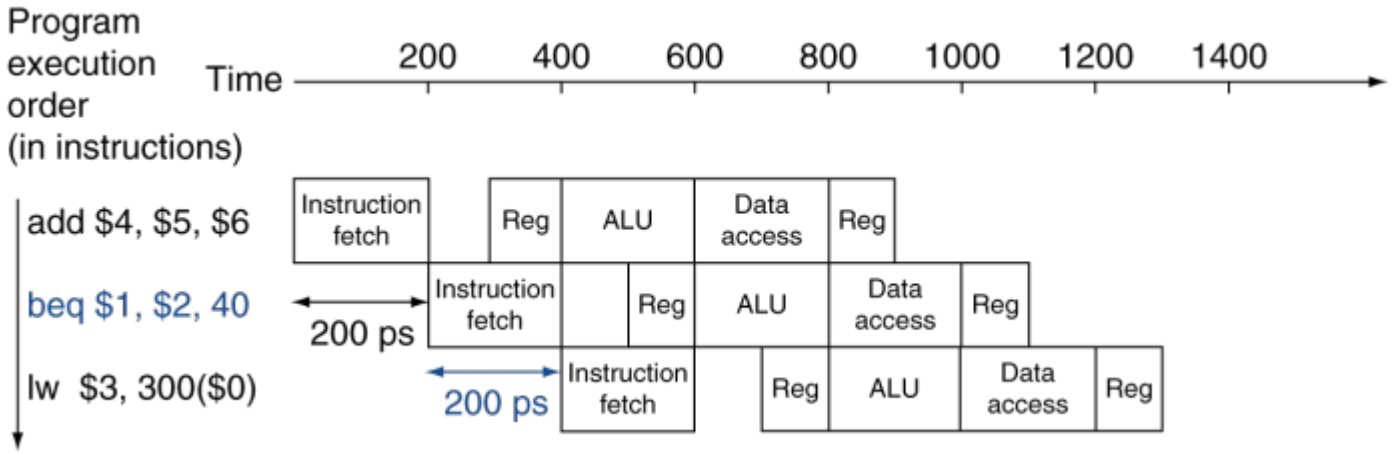
Branch Prediction

- Adopted by most architectures
  - Only stall if prediction is wrong
  - Accurate prediction →high performance

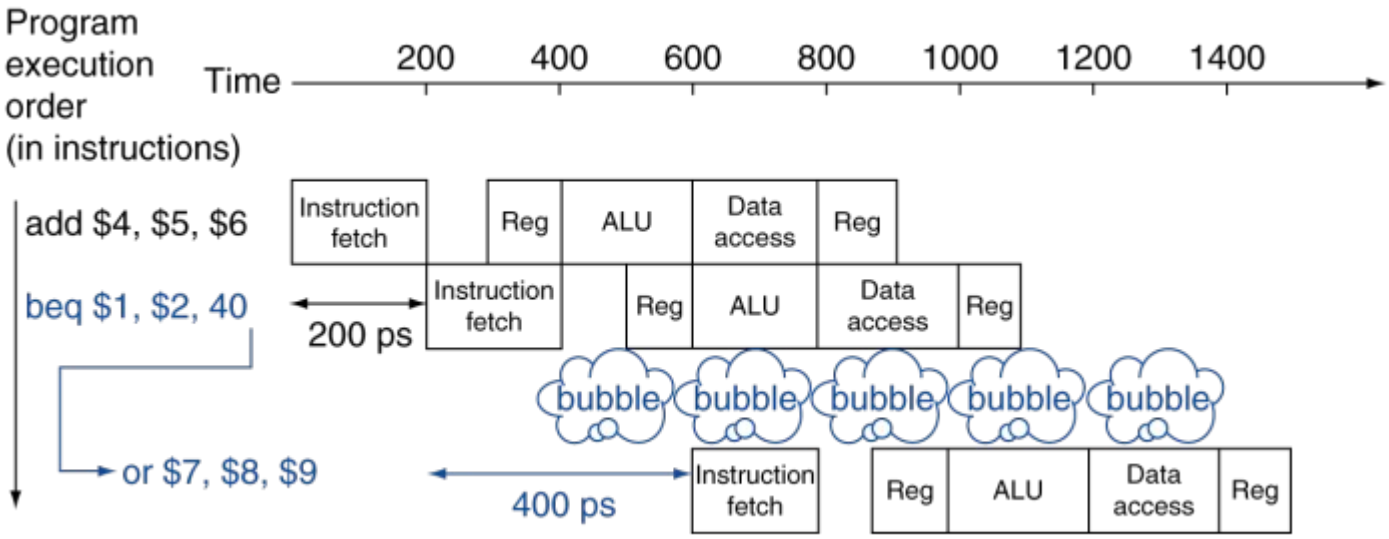
- Complex but necessary prediction policies are needed ***for Longer pipelines*** *(HIGH penalty)*

- In simpler pipelines (e.g., MIPS), simple policies are highly efficient
  - policies may be dynamic or static

Simple Policy: Predict Branch Not Taken



Prediction correct



Prediction incorrect

Optimized hardware –> only one bubble

More-Realistic Branch Prediction

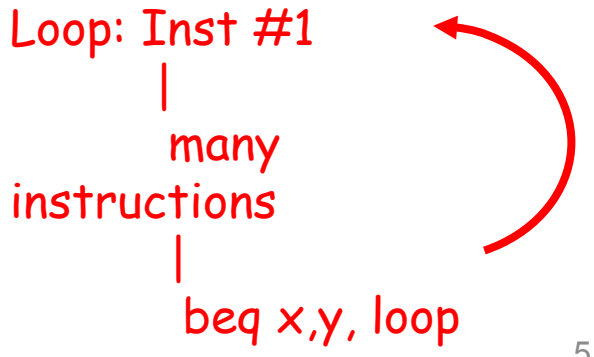
- **Static** branch prediction
  - Based on typical branch behaviour
    - Backward Taken Forward Not Taken (**BTFNT**)
    - Example: loop and if-statement branches

- **Dynamic** branch prediction
  - Hardware track branch behaviour (not/Taken)
  - Assume future behaviour will continue the trend

Dynamic Branch Prediction

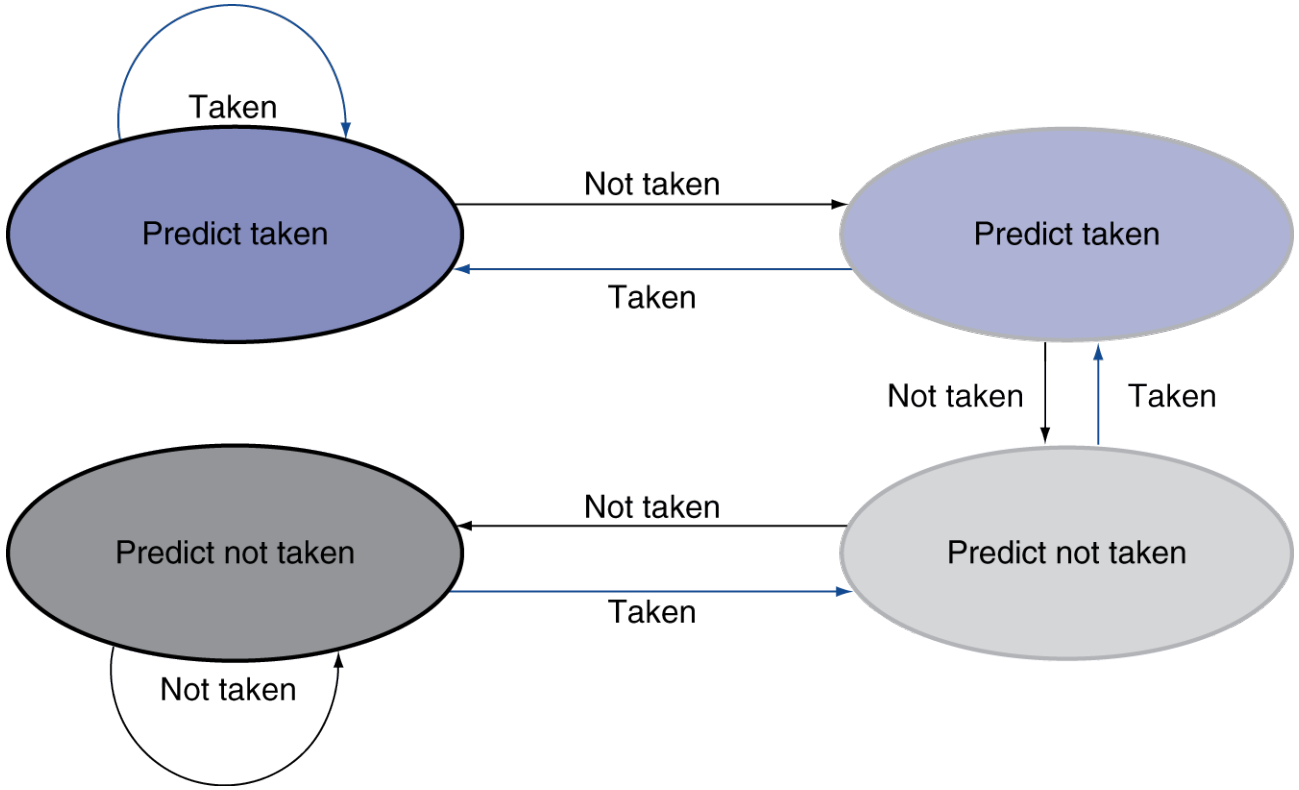
- ***Dynamic branch hazard prediction***
  - **Branch prediction buffer** (aka branch history table)
    - Indexed by recent branch instruction addresses
    - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, predict the outcome [**repeat the last action**]
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

What would happen for a loop?



2-Bit Predictor

- Only change prediction on two successive mispredictions



Pipeline Summary

- Pipelining improves performance by increasing instruction throughput by a factor of # pipeline stages

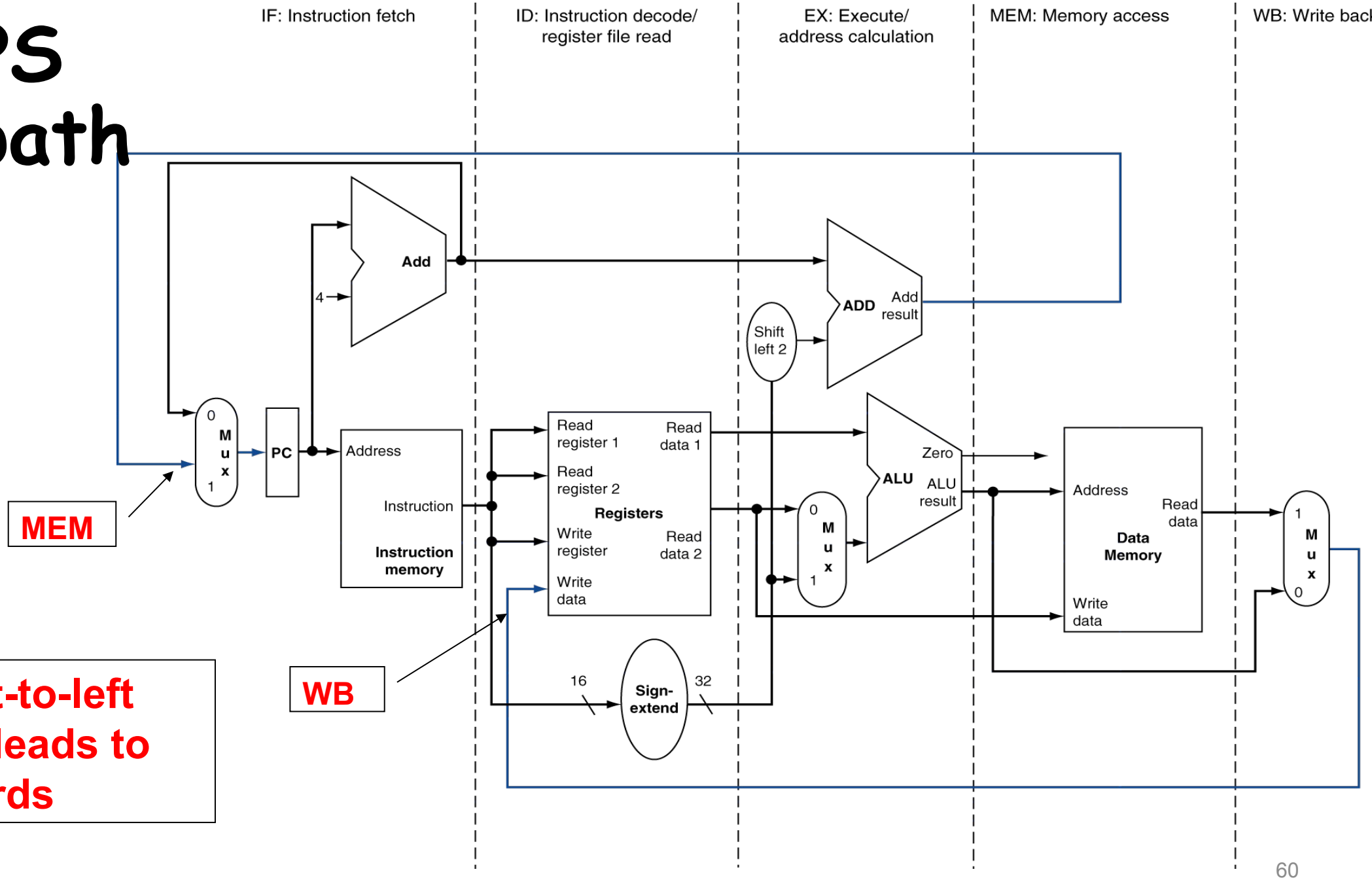
- Subject to hazards
  - Structure → adding HW
  - Data → forwarding and code scheduling
  - Control → wait, optimized wait, predict, delay slot

MIPS Pipelined Datapath Hardware

Remember: Five-Stage pipeline for MIPS

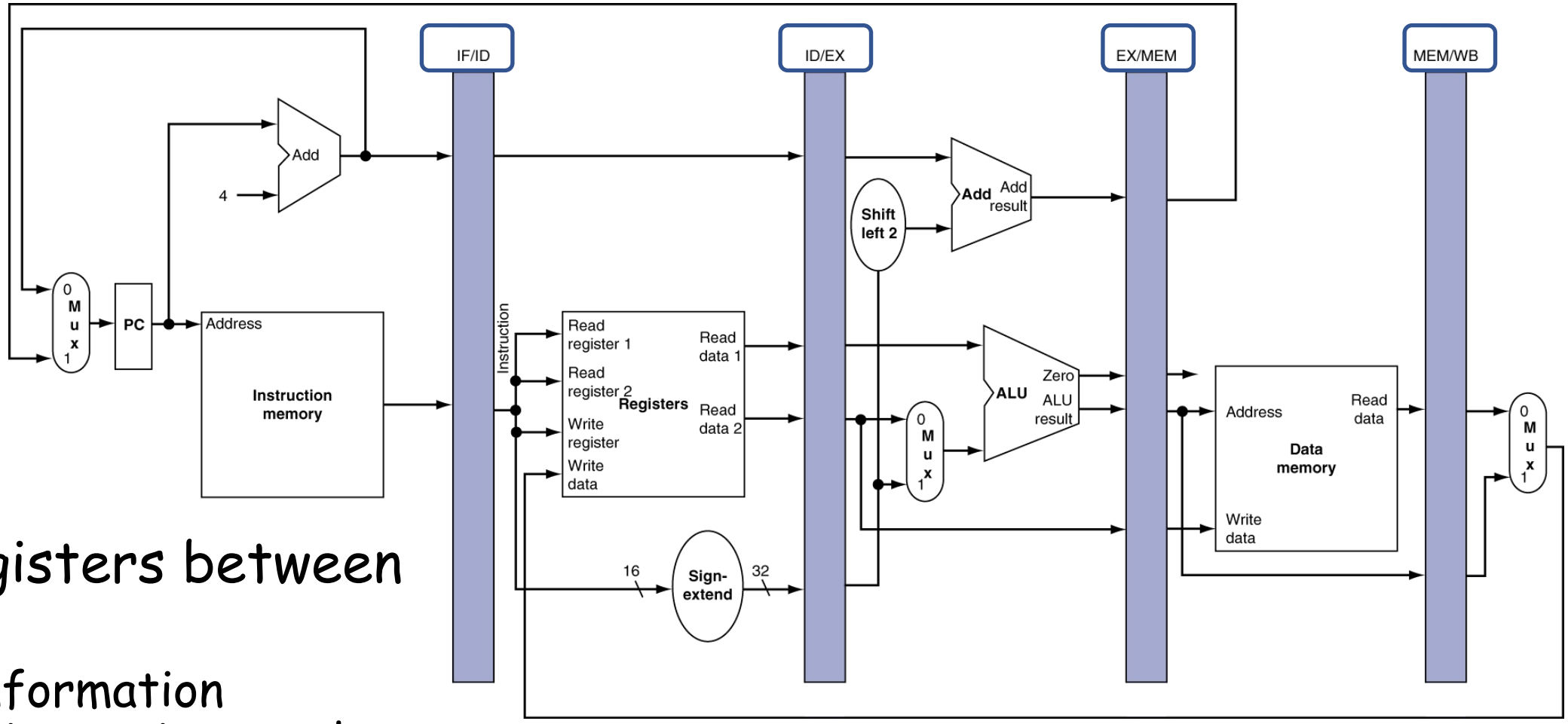
- Dividing instruction into a five-stage pipeline
  - Up to **five instructions** could be in execution during any single clock cycle
  - Instruction and data move from left to right through the stages
    - ***Two exceptions*** to this left-to-right flow of instructions
      - The write back stage
      - Branching

MIPS Datapath



Right-to-left flow leads to hazards

MIPS Pipelined Datapath

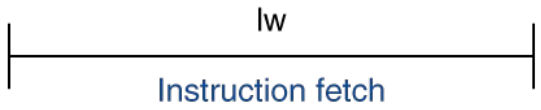


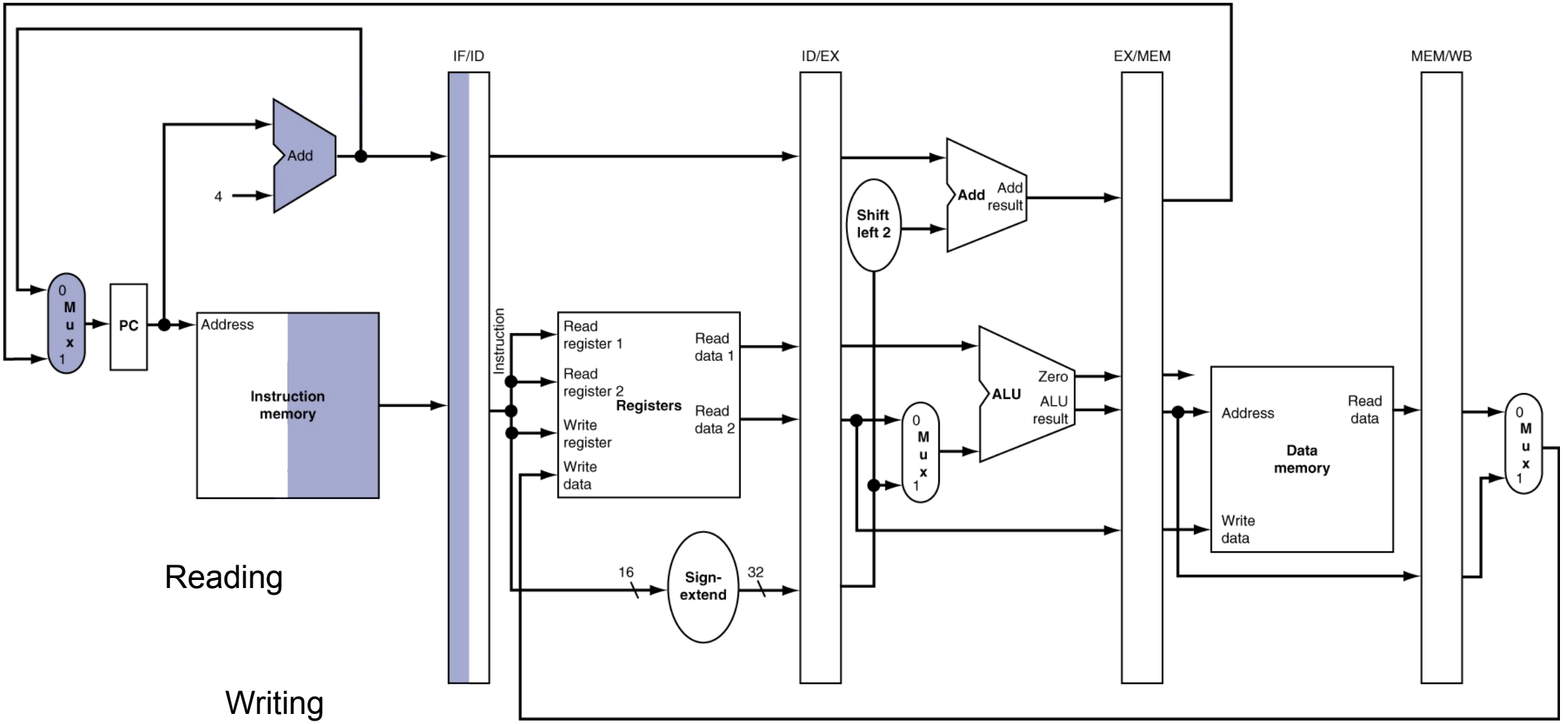
- Pipeline registers between stages
  - To hold information produced in previous cycle

Pipeline Operation Example

- We'll look at "single-clock-cycle" diagrams for load
  - Highlight the right half of register or memory when read operation
  - Highlight left half when they are being written

IF for Load, Store, …

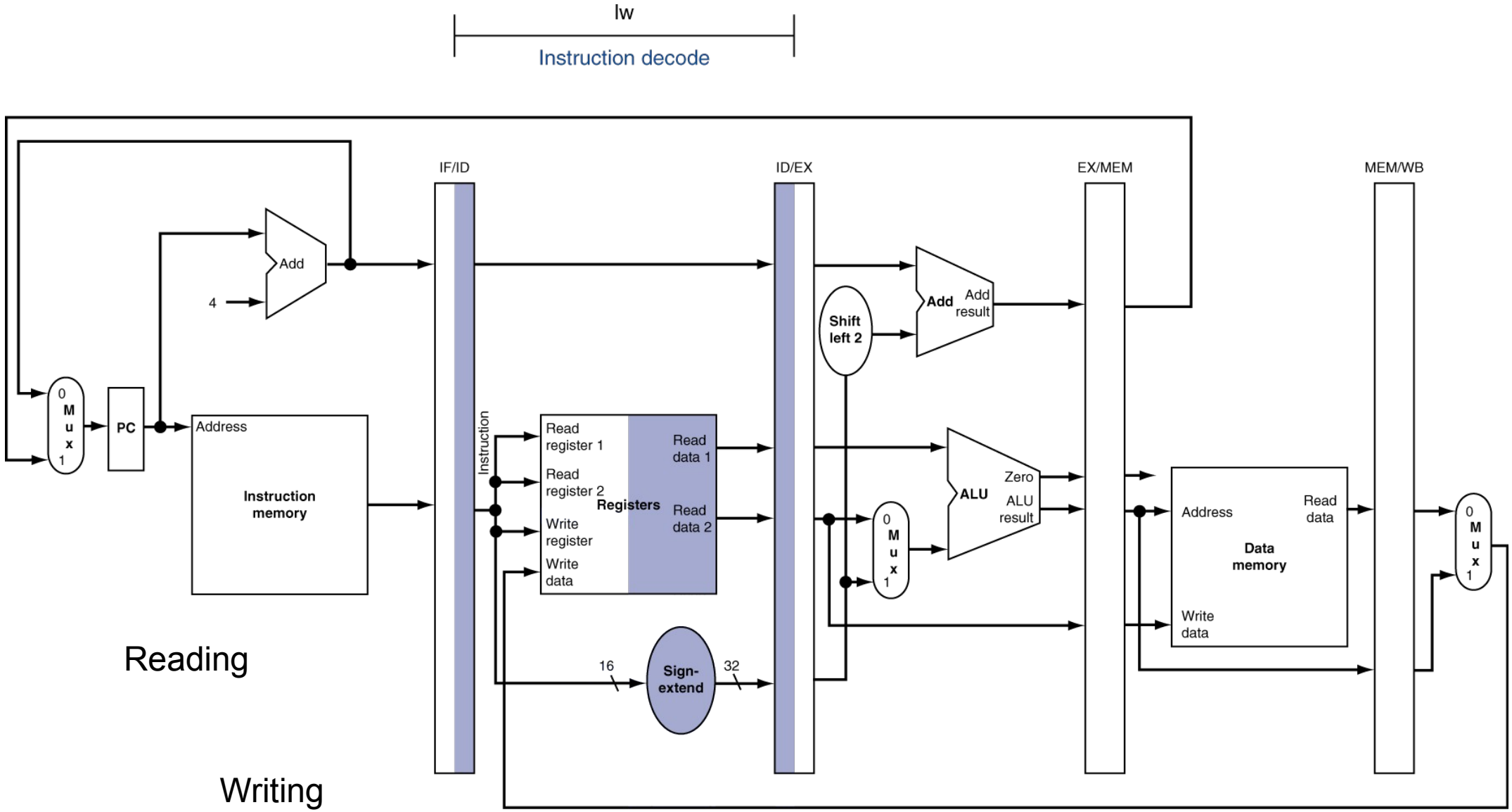




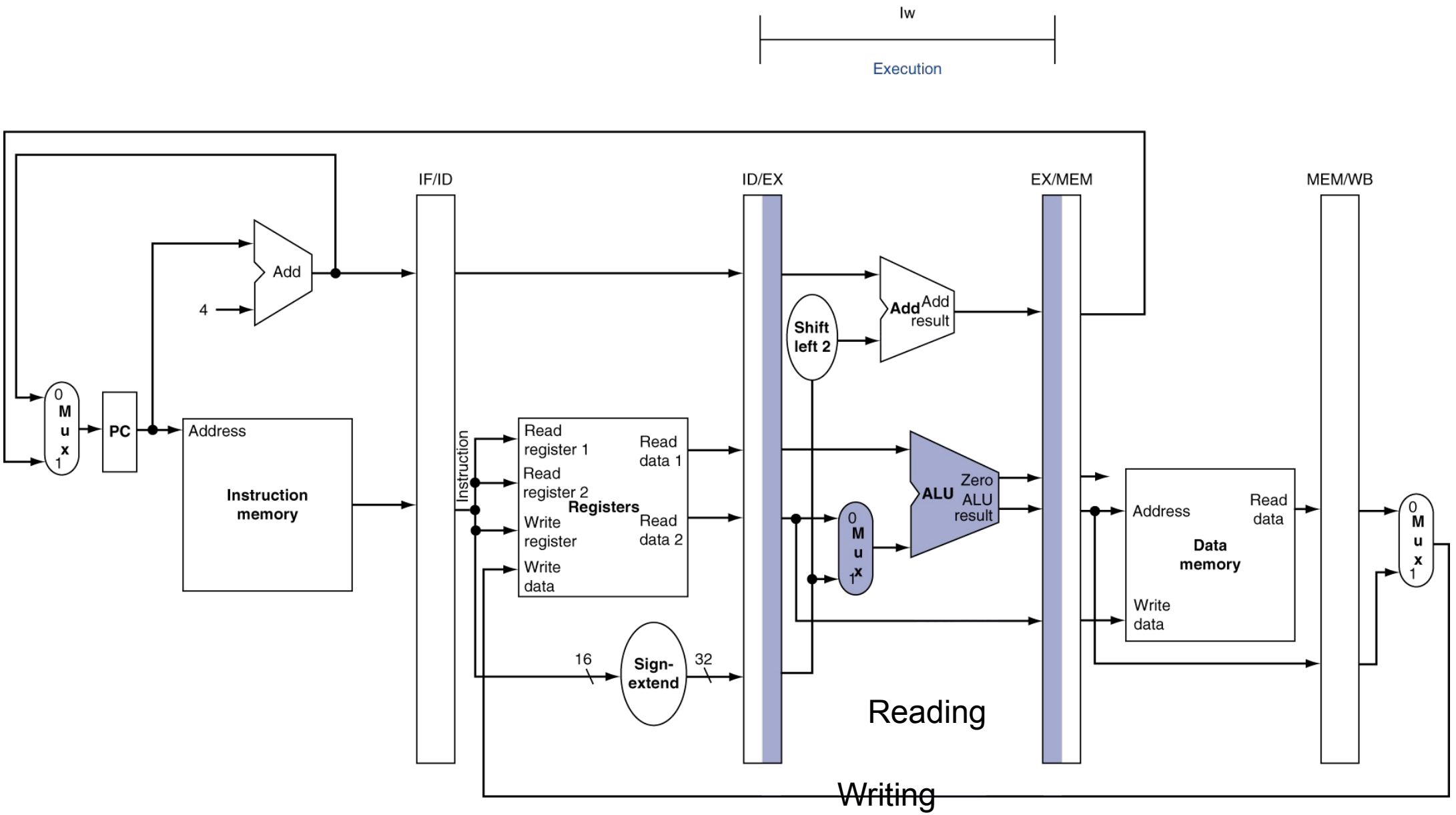
Reading

Writing

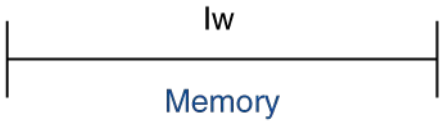
ID for Load, Store, …

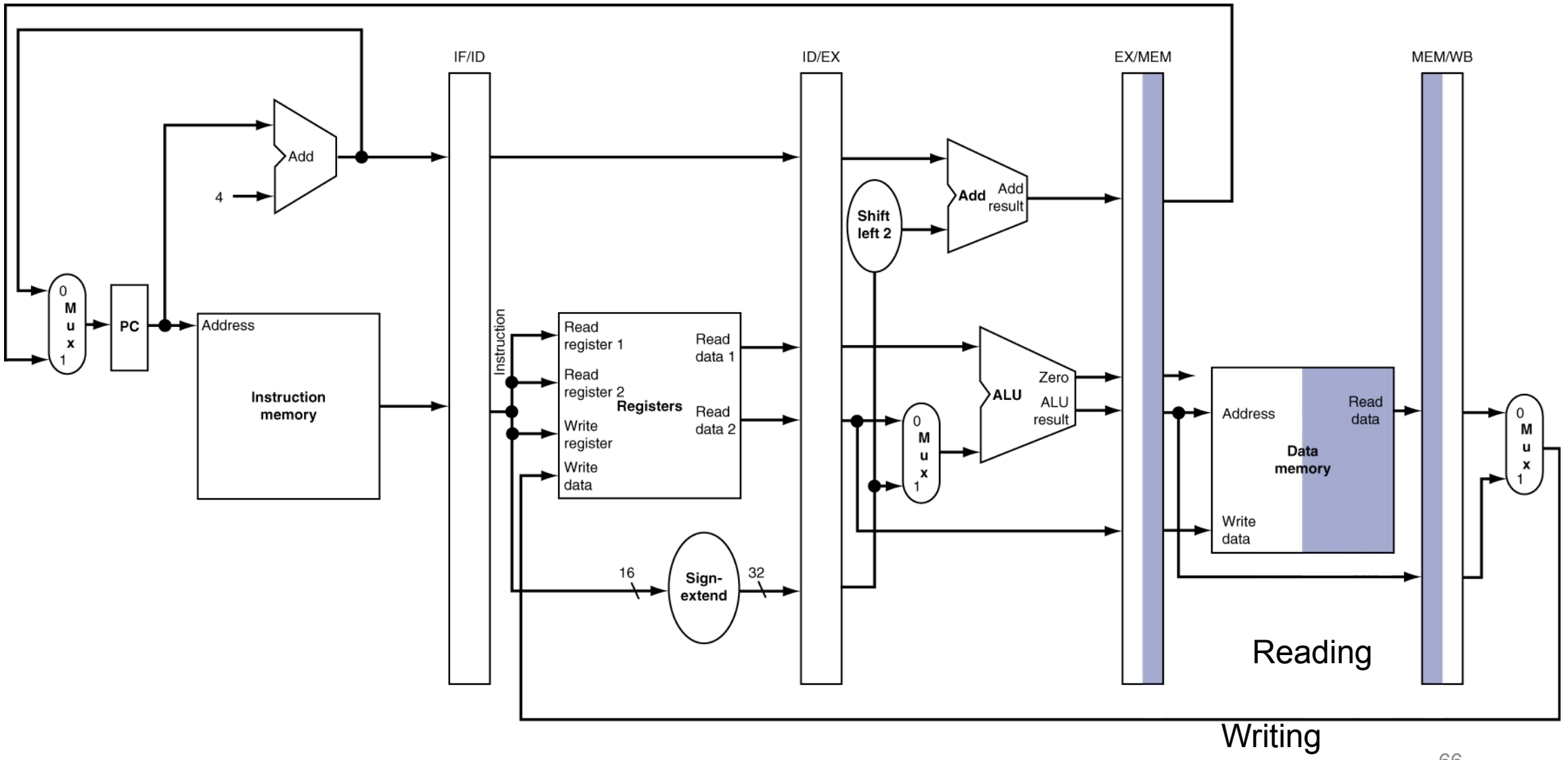


EX for Load


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32
Reading
Writing

MEM for Load

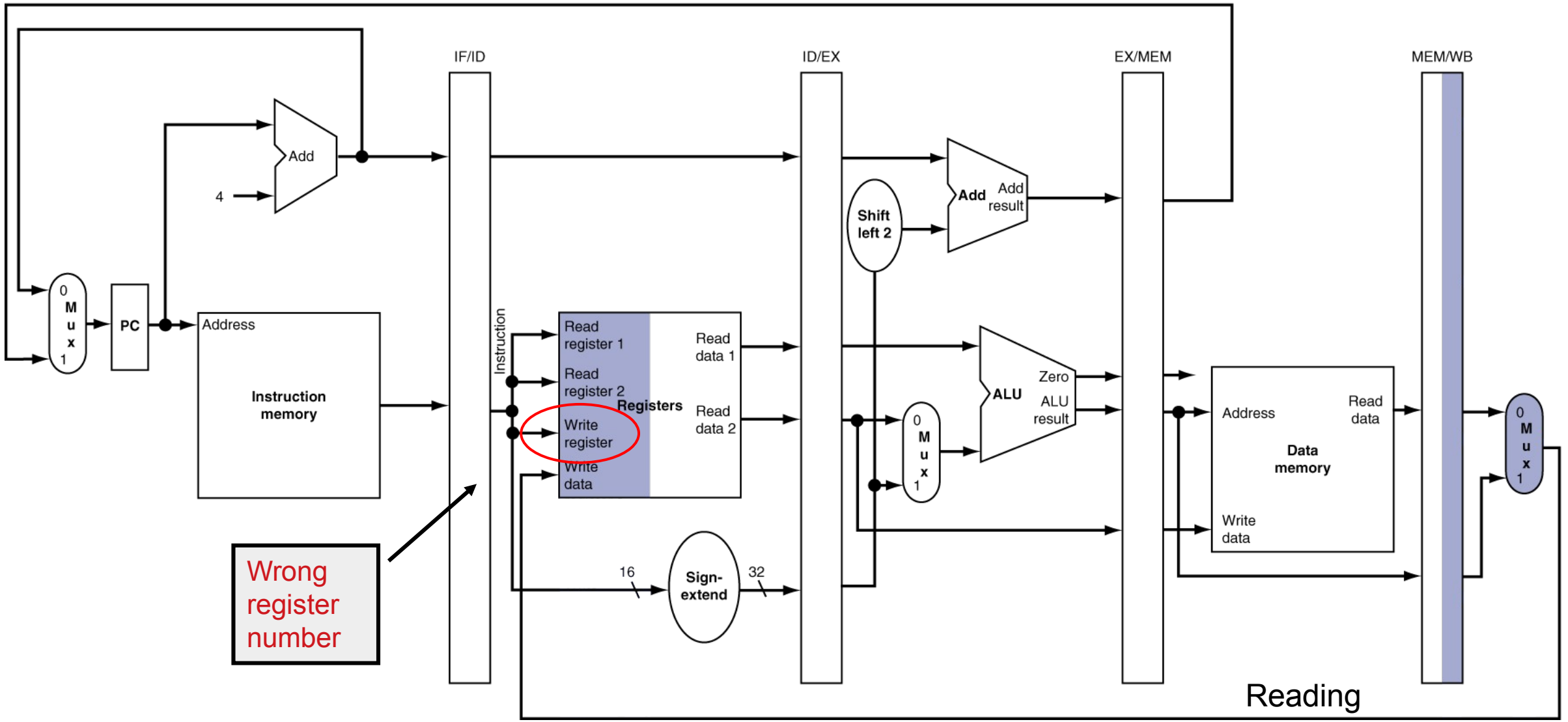

lw
Memory


IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
Reading
Writing

WB for Load

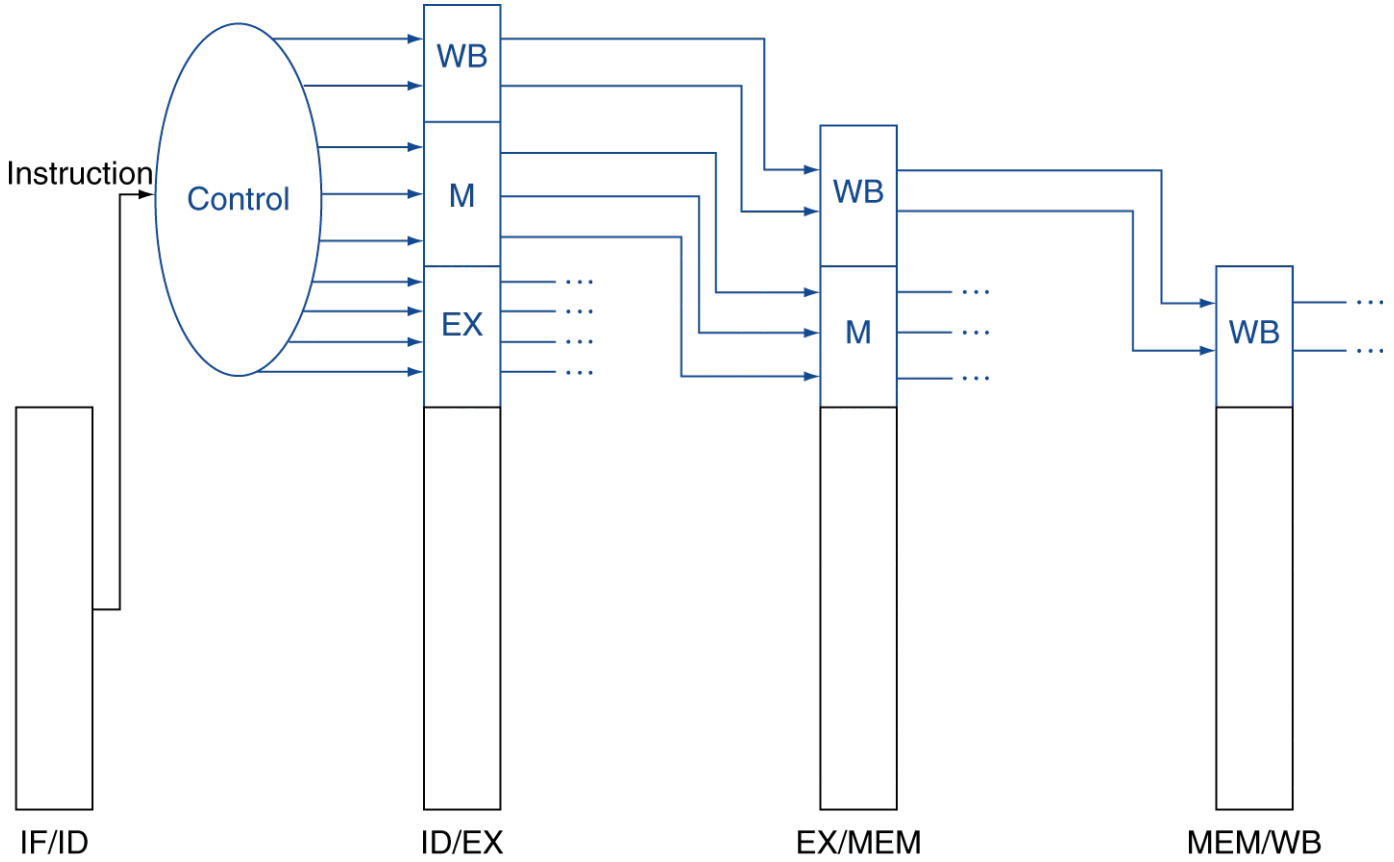
lw

Write back



Pipelined Control

- Control signals derived from instruction
- Control lines start with execution stage
- ***Relevant control should propagate with the instruction → Extended stage registers***



Main control signals

Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Pipelined Control Design

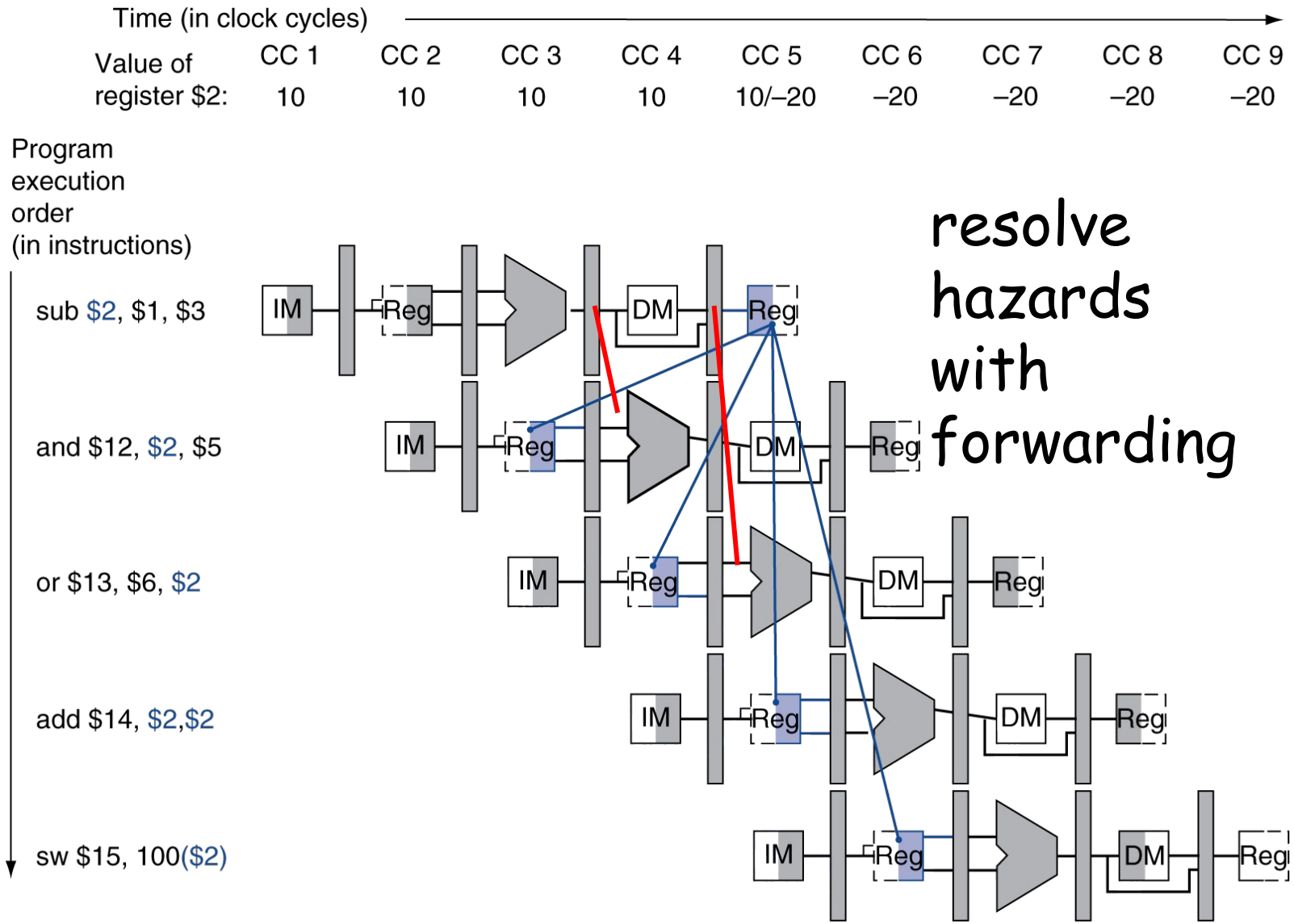
Sections 4.6

Data Hazards in ALU Instructions

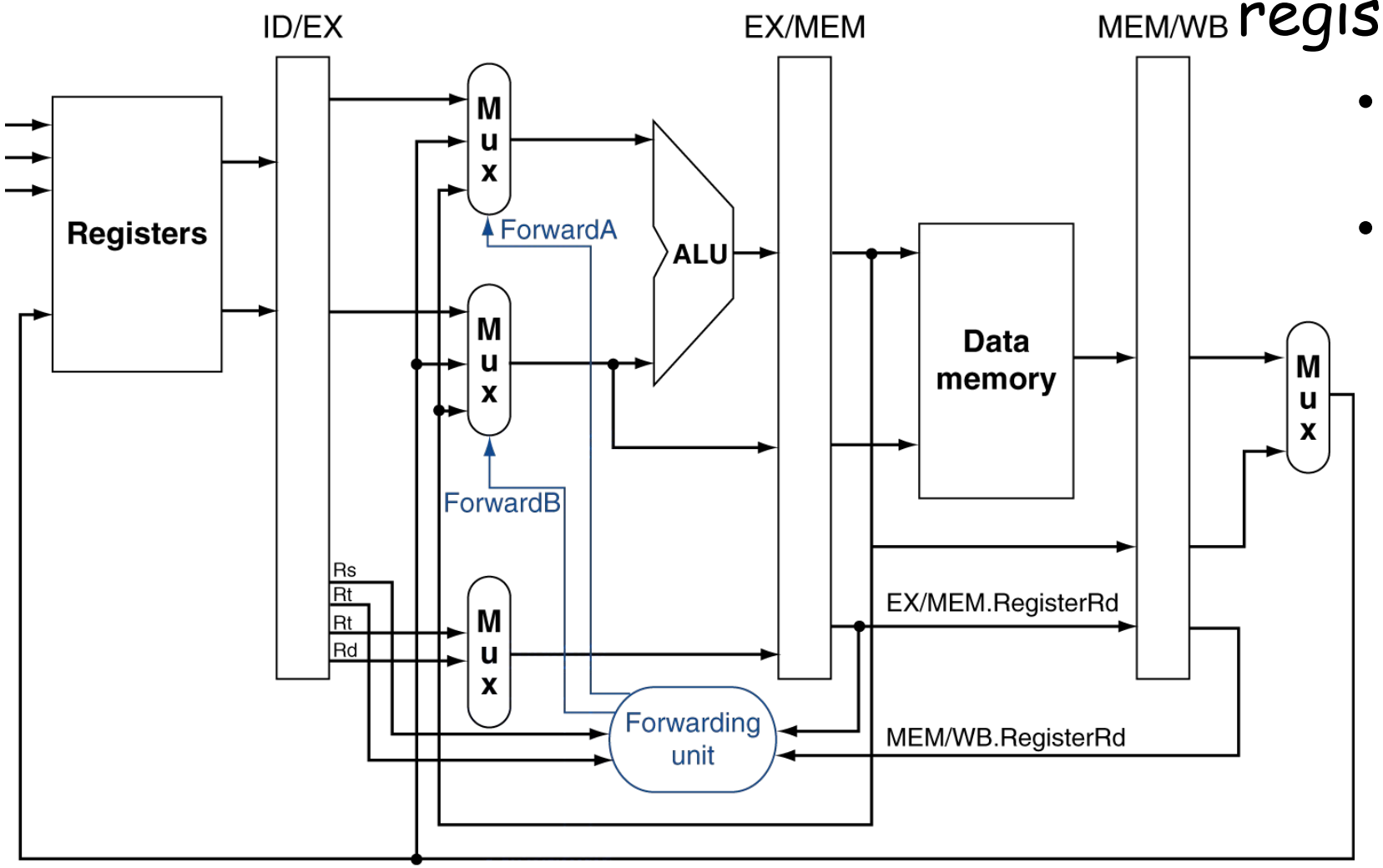
- Consider this sequence with many dependencies:

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

Assuming $2 has the value 10 before the subtraction -20 after the subtraction



Forwarding Hardware



b. With forwarding

- Forward implies reading ALU operands from a source other than the register-file registers
  - ***Multiplexers*** are needed at the ALU I/P
  - A ***forwarding unit*** determines the multiplexor control signal

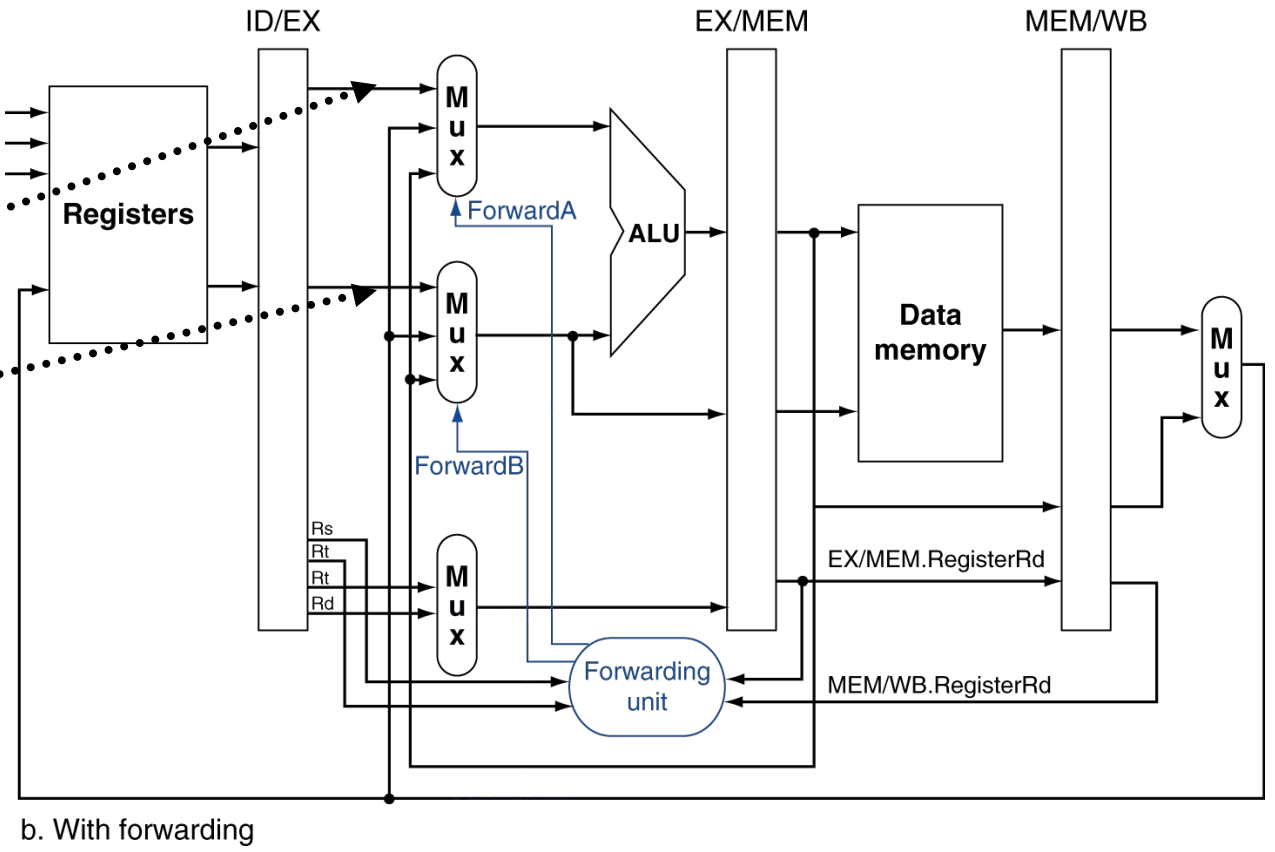
Forwarding Unit Output

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Forwarding HW Design

Register Naming convention: ID/EX.RegisterRs → register number for Rs sitting in ID/EX pipeline register

ALU operand register numbers in EX stage are given by ID/EX.RegisterRs, ID/EX.RegisterRt



b. With forwarding

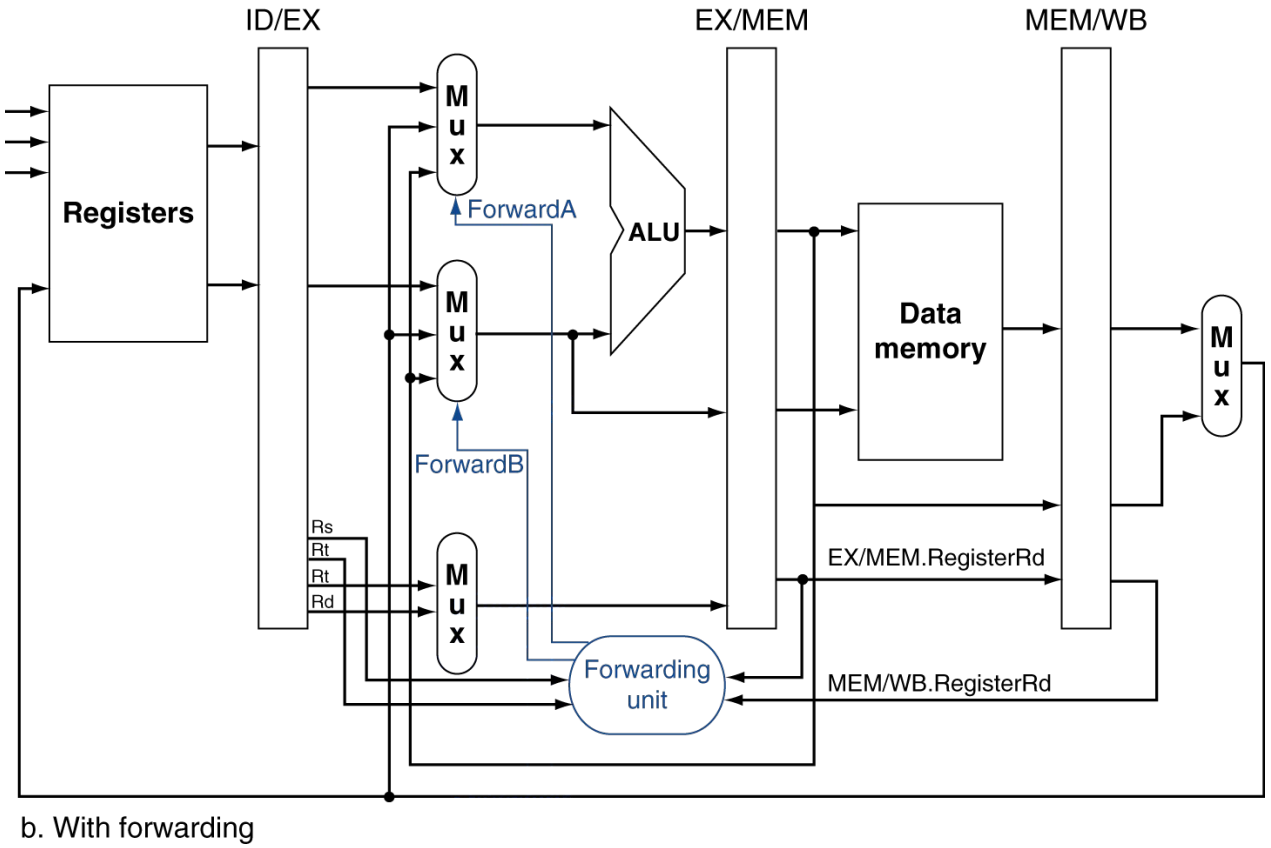
When to Forward? (1/2)

- **Data hazards** when
  - 1a. ID/EX.RegisterRs = EX/MEM.RegisterRd
  - 1b. ID/EX.RegisterRt = EX/MEM.RegisterRd

Fwd from EX/MEM pipeline reg

  - 2a. ID/EX.RegisterRs = MEM/WB.RegisterRd
  - 2b. ID/EX.RegisterRt = MEM/WB.RegisterRd

Fwd from MEM/WB pipeline reg



b. With forwarding

sub **$2**, $1,$3
and $12,$2,$5
or $13,$6,$2

When to Forward? (2/2)

- But only if forwarding instruction will write to a register!
  - ***EX/MEM.RegWrite, MEM/WB.RegWrite***
- And only if Rd for that instruction is not $zero
  - ***EX/MEM.RegisterRd ≠ 0,***
    MEM/WB.RegisterRd ≠ 0
- Recall in MIPS every use of $zero ($0) as operand yields an operand value of 0
  - Example: sll $0, $0, 0 (NOP)

Forwarding Conditions

- **EX/MEM hazard**
  - If (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - If (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- **MEM/WB hazard**
  - If (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - If (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Double Data Hazard

- Consider the sequence:

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

Double Data Hazard
Both hazards occur

- In this case, the processor should use the most recent result at MEM stage
  - Revise MEM hazard condition to Only fwd if EX hazard condition isn't true

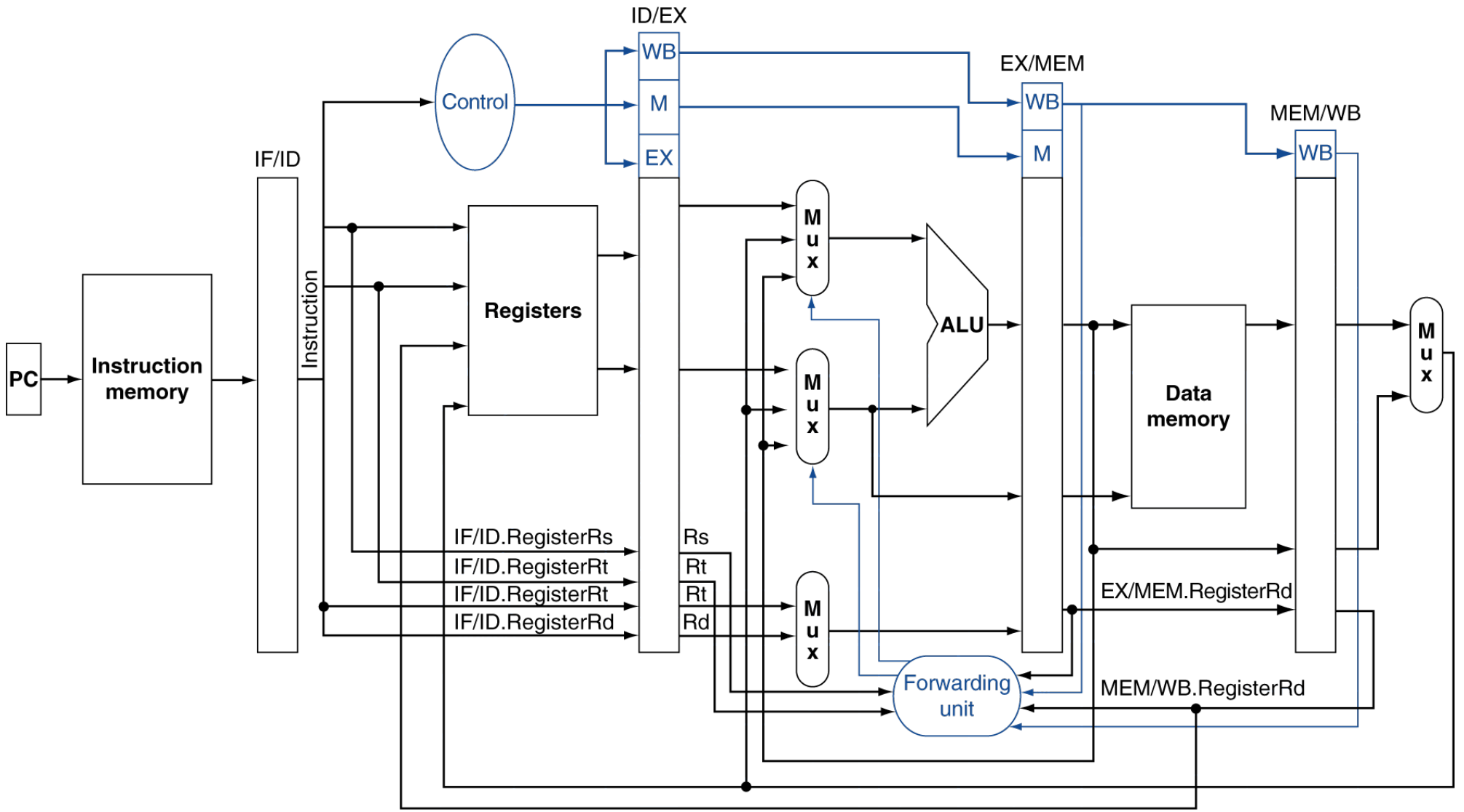
Is double data Hazard common?

a=a+b+c+d+...+z

Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

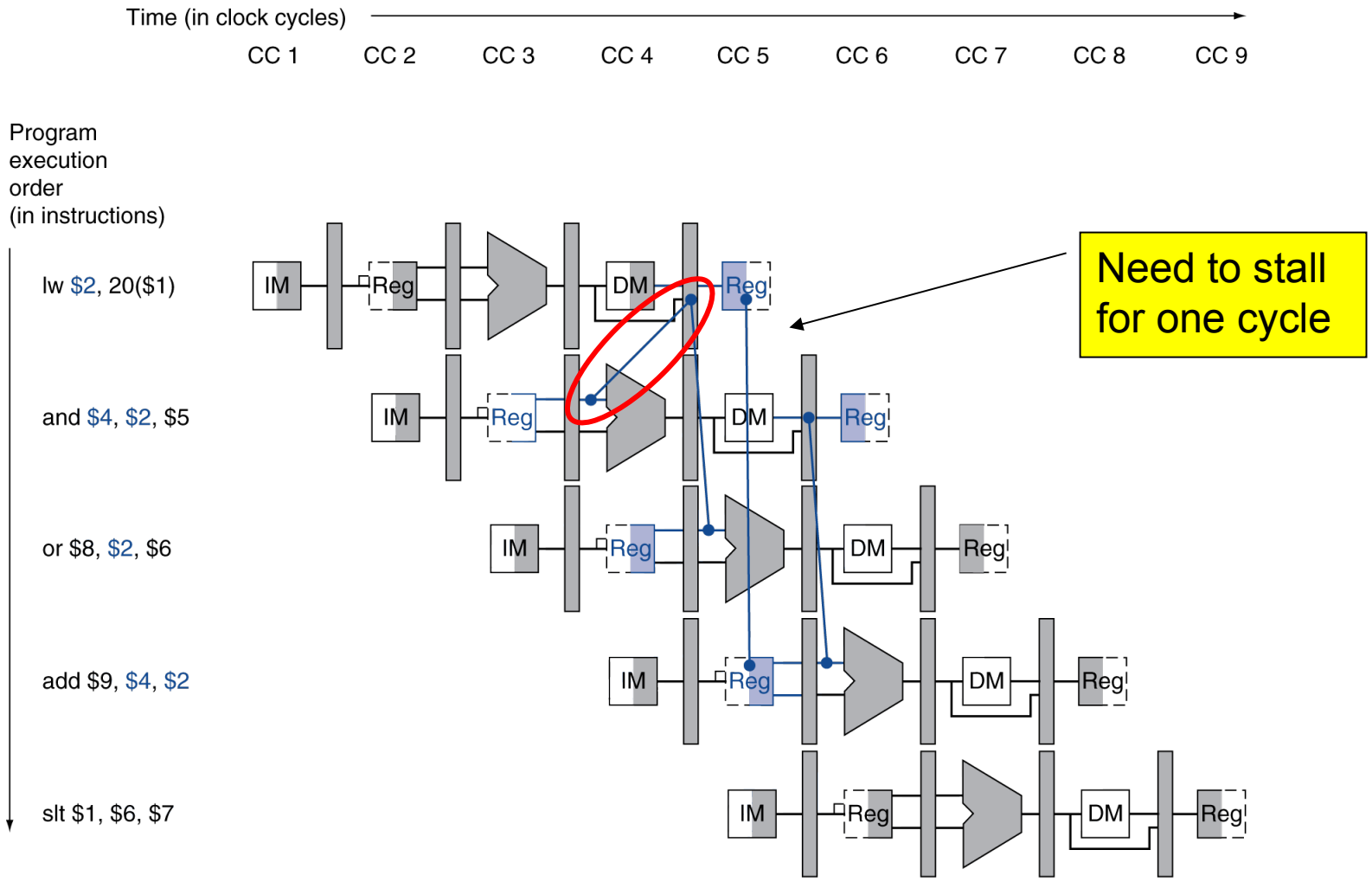
Datapath with Forwarding and Control



Load-Use Data Hazard

- Data forwarding does not work when
  - Instruction tries to read a register following a load instruction that writes the same register.
- Something must stall in the pipeline for a combination of load followed by an instruction that reads its result.
- Therefore, we need ***a hazard detection unit*** in the pipeline to implement a stall.

Pipelined Load Data Hazard



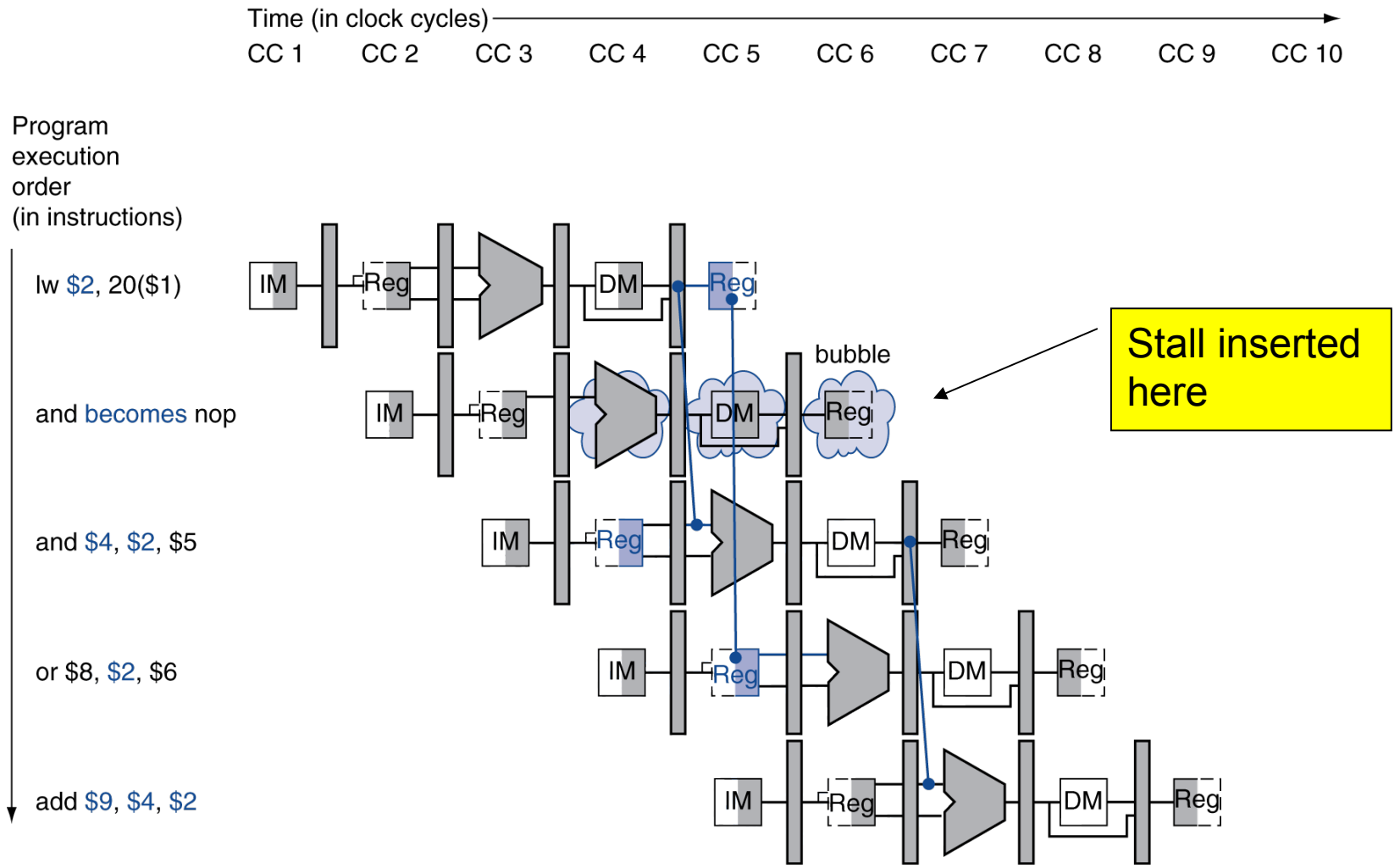
Memory Load Hazard Detection

- Check when using instruction is decoded ***in ID stage***
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load hazard occurs when
  - ID/EX.MemRead **and**
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) **or**
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble

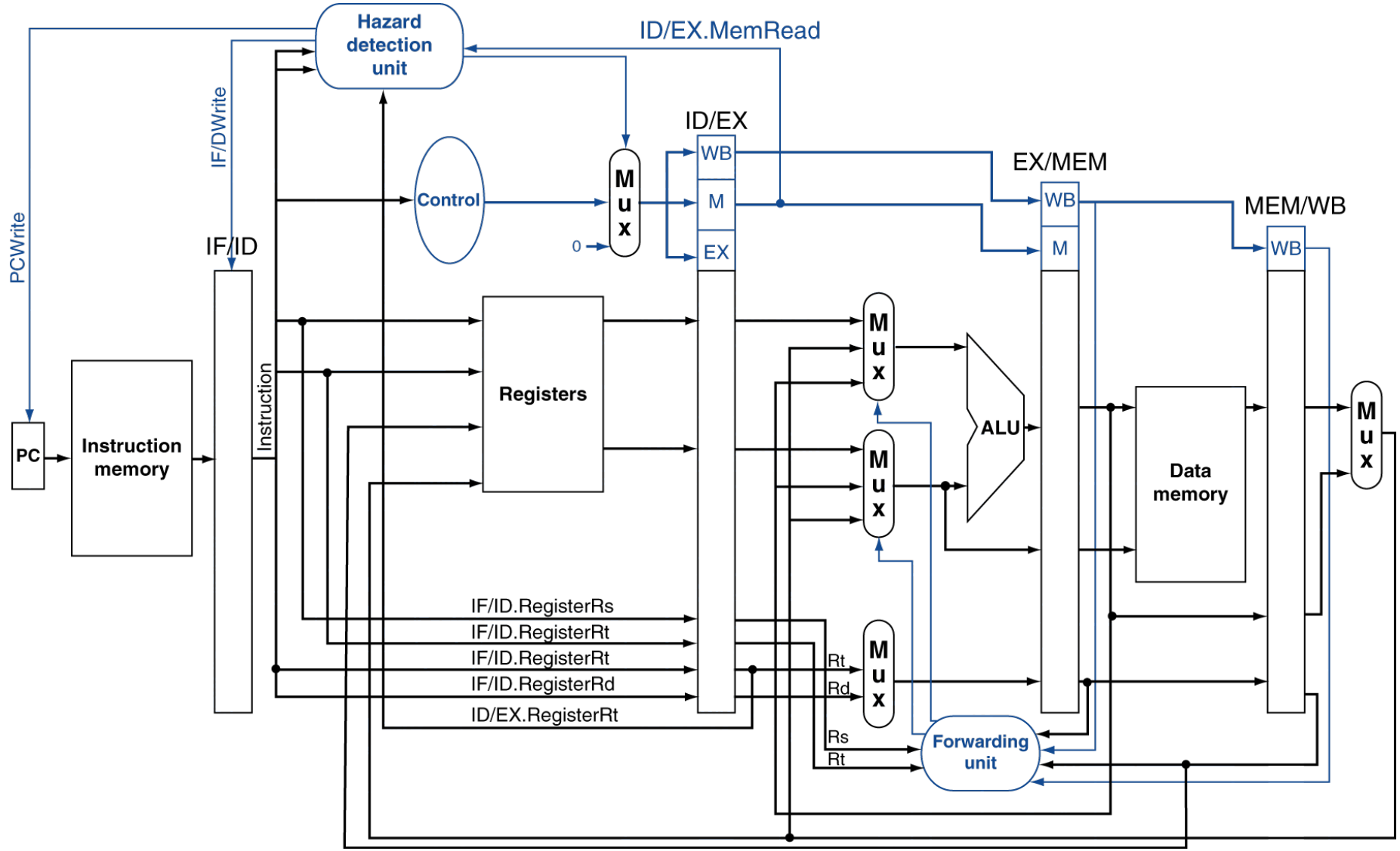
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do `nop` (no-operation)
- Prevent update of PC and IF/ID register
  - instruction is decoded again
  - Following instruction is fetched **again**
  - 1-cycle stall allows MEM to read data for `lw`
    - Can subsequently forward to EX stage

Stall/Bubble in the Pipeline



Datapath with Hazard Detection



- Explored hardware implementation of bypassing (datapath and control)

NEXT: Branch Hazards

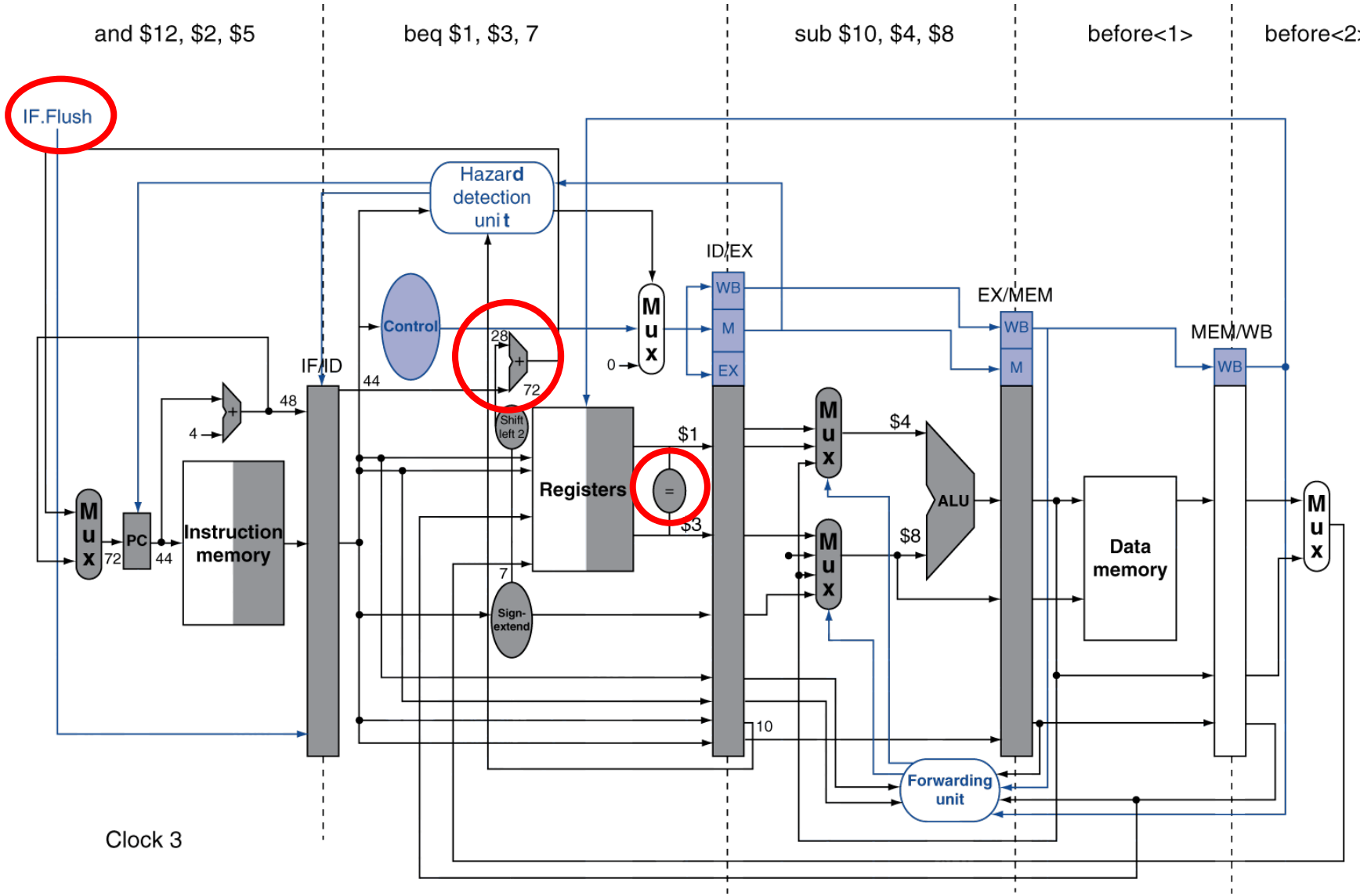
- Speeding using hardware optimization
- Data hazards for branch operands
- Dynamic branch prediction

Reducing Branch Delay

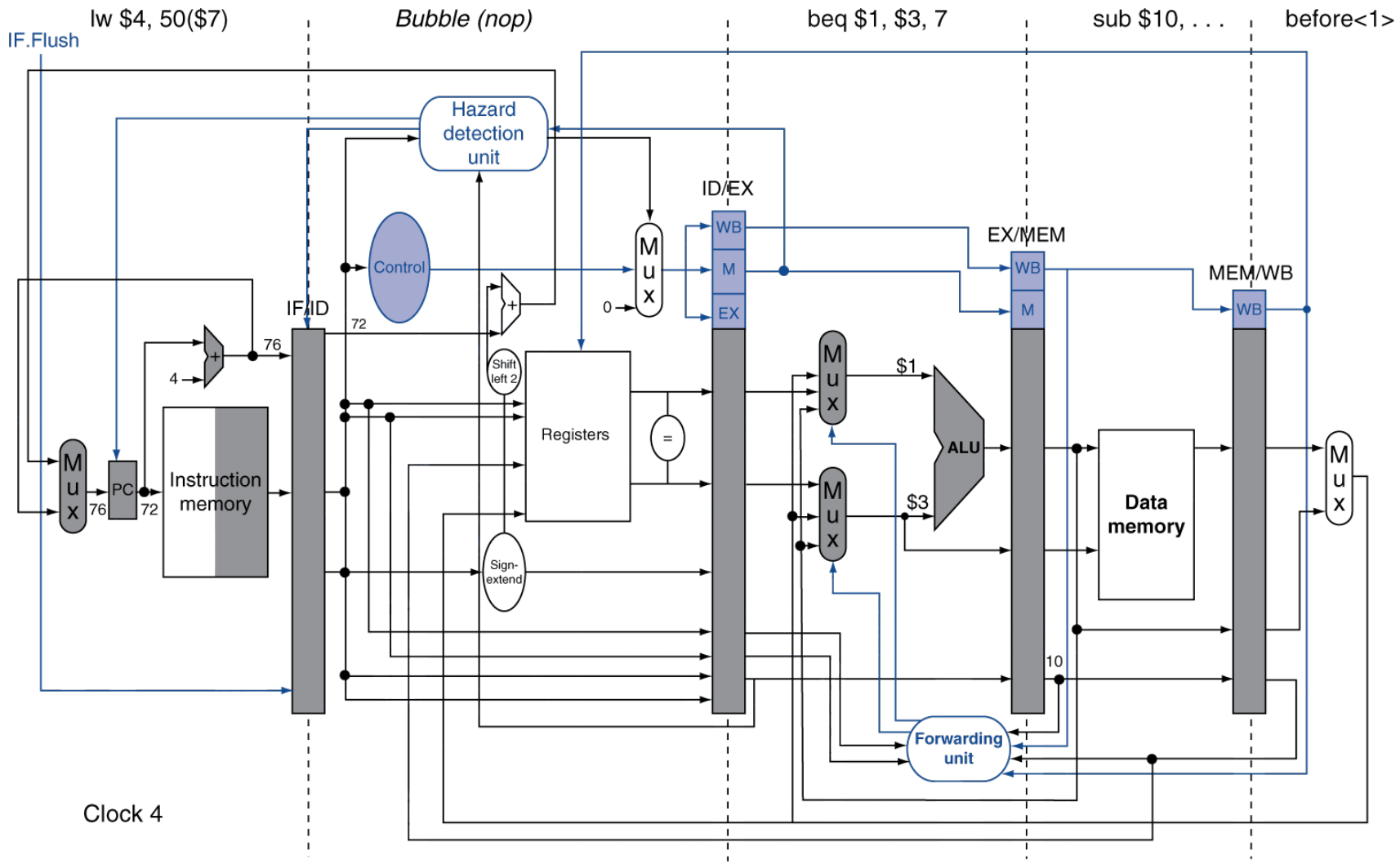
- Determine branching decision at the ID stage using additional hardware [hazard detection]
  - Target address adder
  - Register comparator (**XOR then ORing bits for equality**)
- Early branch detection reduces the penalty to one cycle
  - IF.flush to flush the fetched instruction
- Example: branch taken

```
36:  sub  $10, $4, $8
40:  beq  $1,  $3, 7
44:  and  $12, $2, $5
48:  or   $13, $2, $6
52:  add  $14, $4, $2
56:  slt  $15, $6, $7
     ...
72:  lw   $4, 50($7)
```

Branch Taken Implementation (1/2)

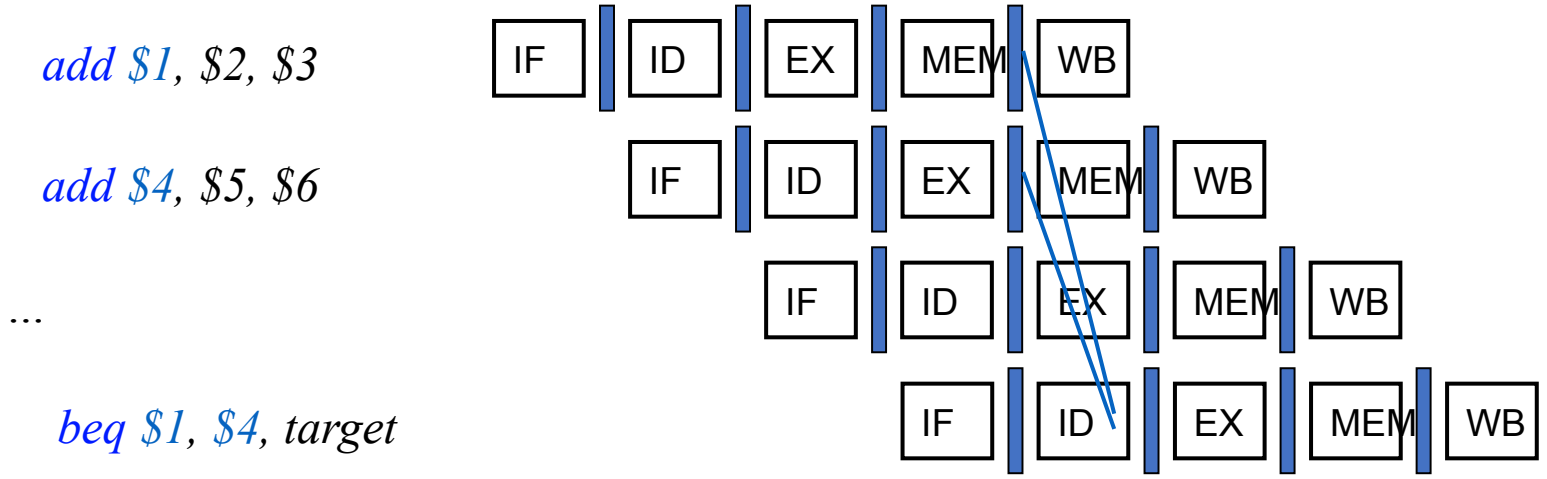


Branch Taken Implementation (2/2)



Data Hazards for Branches

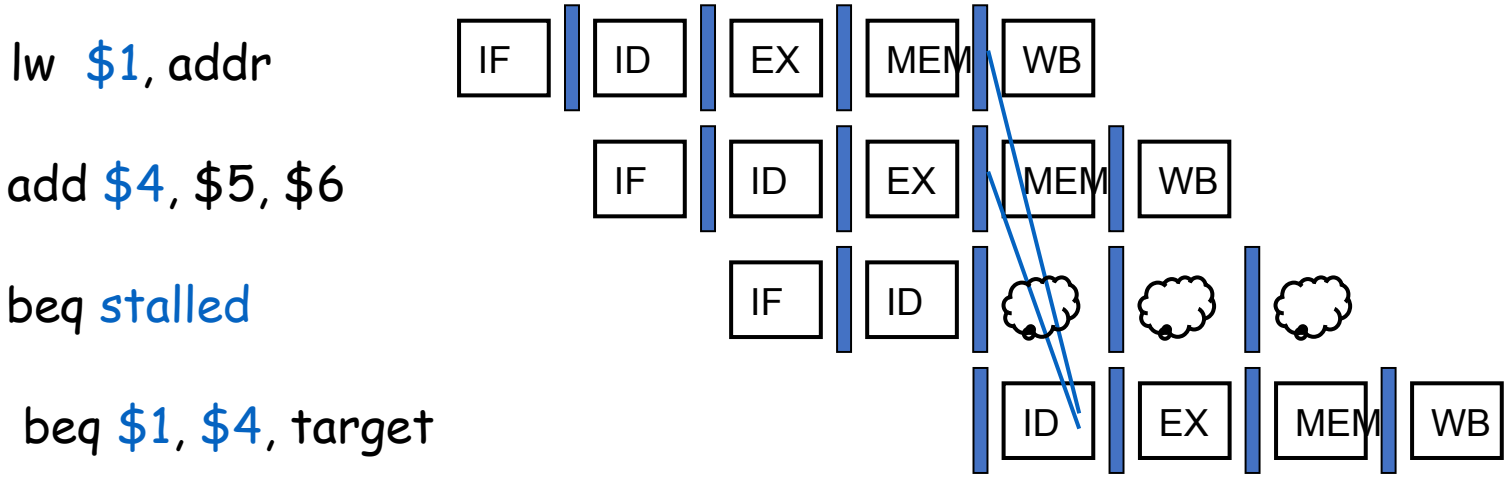
- If a comparison register is a destination of **2nd** or **3rd** preceding ALU instruction



- Can be resolved using forwarding

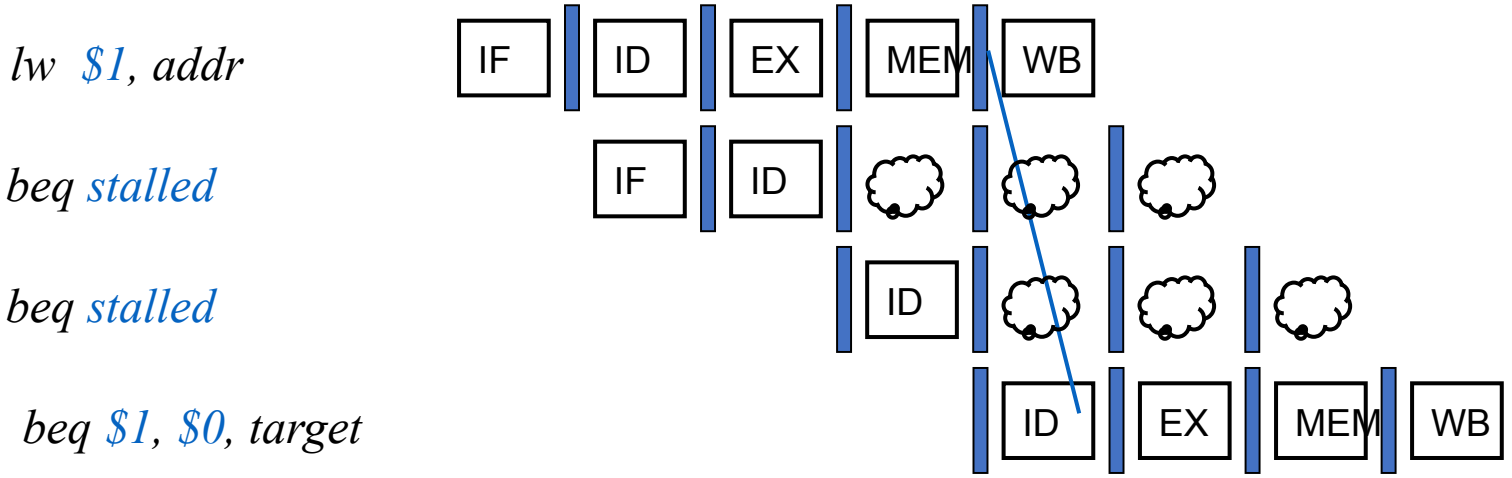
Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles



Calculating the Branch Target

- Dynamic prediction mechanisms need
  - **Branch prediction buffer** to store branch prediction data (status, not/Taken)
  - **Branch target buffer** to store the target address
    - Indexed by PC when instruction fetched
      - If hit and instruction is branch predicted taken, can fetch target immediately

Where are we?

We have explored a single clock cycle processor design [datapath and control]

We have introduced pipelining and established its performance benefits and operational challenges

We have explored pipelined processor implementation [datapath and control]

Detect the hazards

In funA, identify two pipelining hazards of different types. Explain if these hazards can be avoided or not. (Exam question)

```
28  funA:
29          lw $v0, 0($a0)
30          addi $t1, $0, 0
31          loopA:      addi $t1, $t1, 1
32                      beq $t1, $a1, endA
33                      addu $a0,$a0,4
34                      lw $t2,  0($a0)
35                      blt $v0, $t2, loopA
36                      add $v0, $zero, $t2
37                      j loopA
38          endA: jr $ra
```