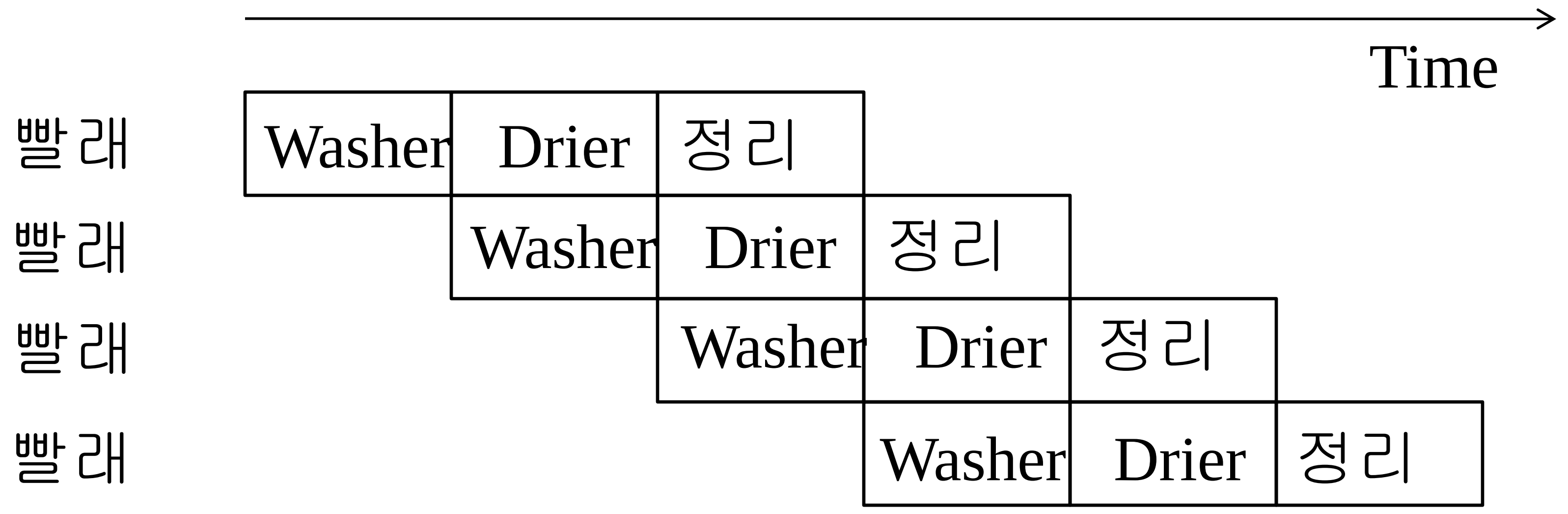
Chapter 4

Implementing ISA (Fetch, Decode, Execute)

Part 2: Pipelining

Pipelined Laundry (반복)

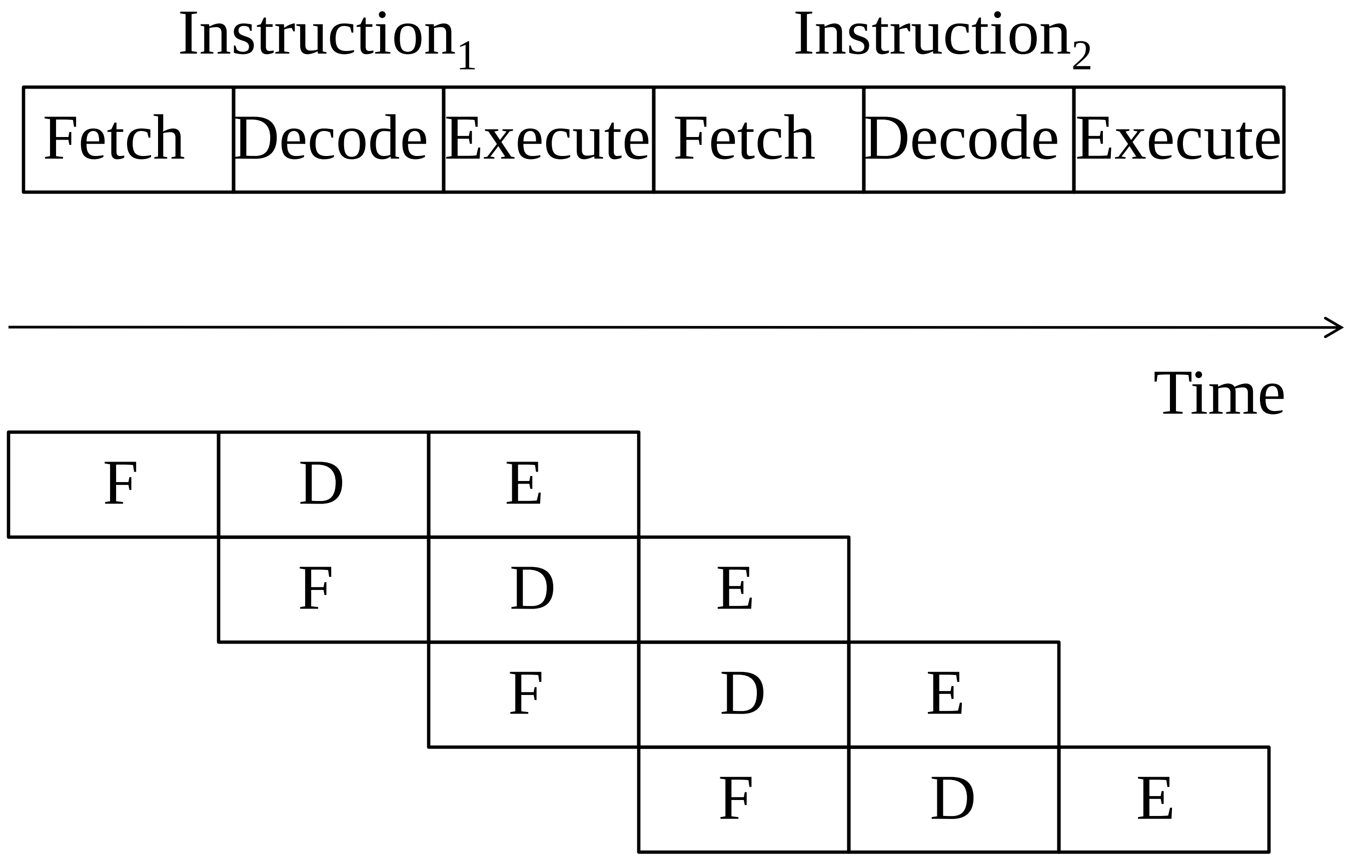
- 3-stage pipelining problem
  - Four loads: speedup = 12/6 = 2
  - Infinite loads: speedup = 3 = number of stages
- Need more hardware? What if we have more hardware?



† Pipelining: general speedup technique

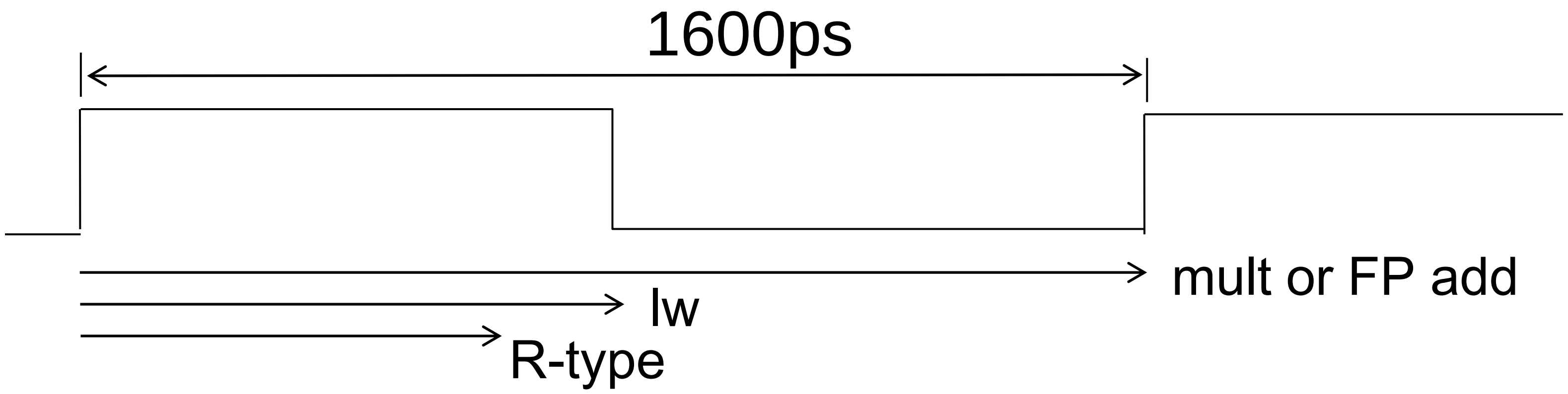
Pipelining (반복)

- 3-stage pipeline
  - Overlapped execution, instruction-level parallelism

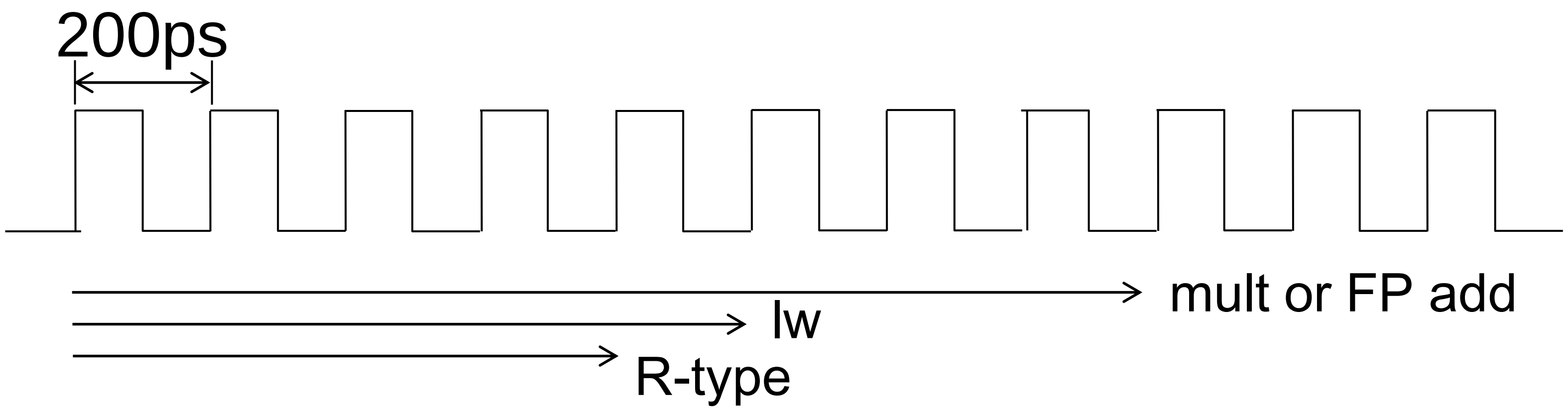


Performance: SingleC/MultiC/Pipelined (반복)

- Single-cycle: CPI = 1, clock cycle



- Multi-cycle: CPI ↑, clock cycle ↓, overall performance ↑



- Pipelined: CPI = 1, clock cycle ↓

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
  5. WB: Write result back to register

Instr	Instr fetch	Register read	ALU op	Memory access	Register 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

Pipelining vs. Multicycle (반복)

- Each stage run different instructions

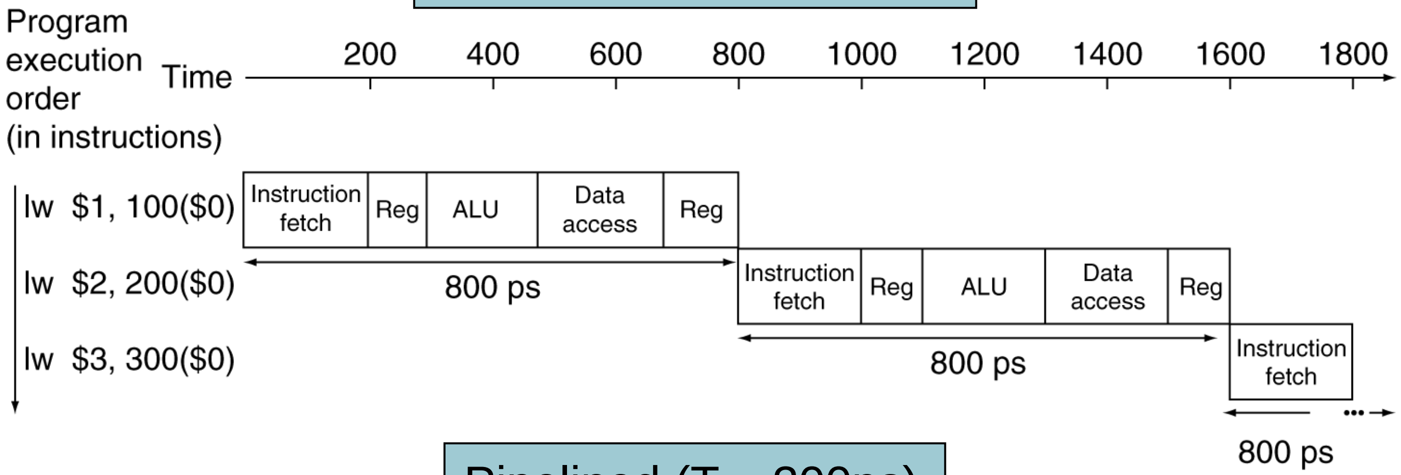
Time

	IF	ID	EX	DM	WB
CC1	I_0				
CC2	I_1	I_0			
CC3	I_2	I_1	I_0		
CC4	I_3	I_2	I_1	I_0	
CC5	I_4	I_3	I_2	I_1	I_0
CC6	I_5	I_4	I_3	I_2	I_1

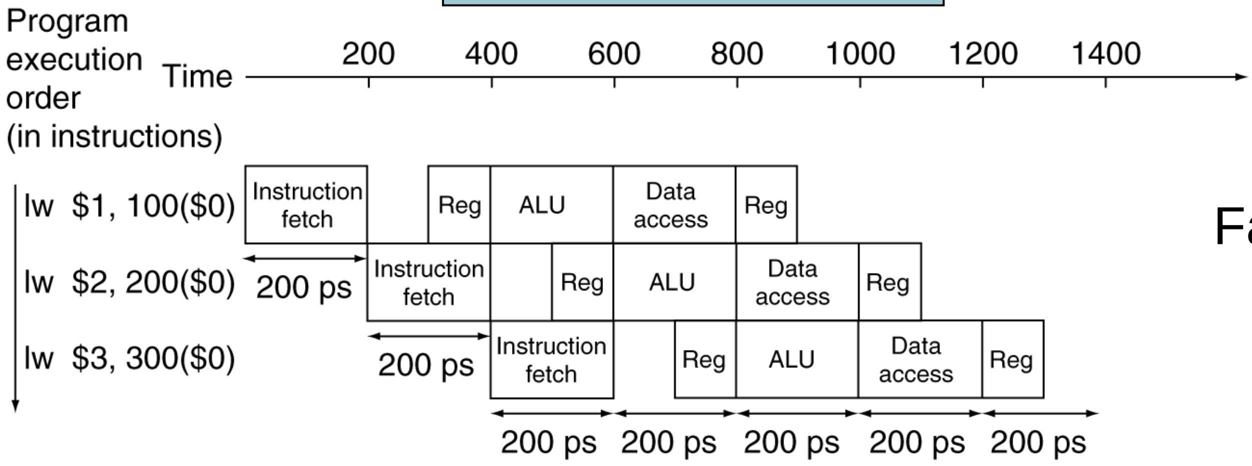
CC1	I_0				
CC2		I_0			
CC3			I_0		
CC4				I_0	
CC5					I_0
CC6	I_1				

Pipeline Performance

Single-cycle (T_c= 800ps)



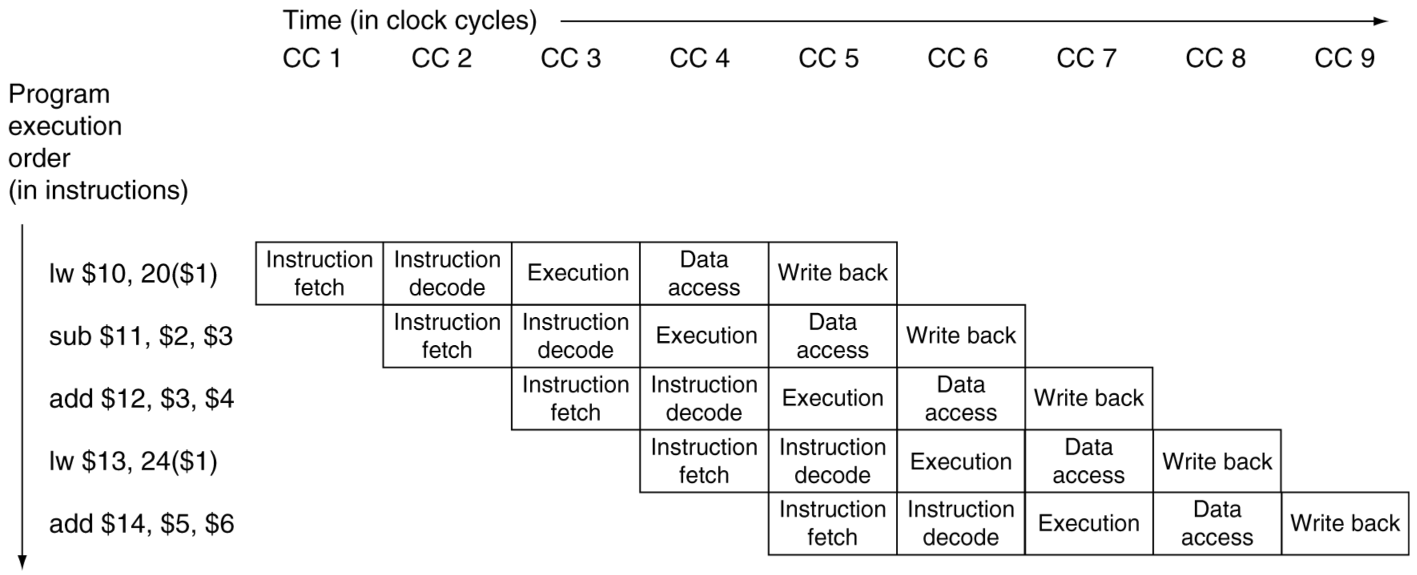
Pipelined (T_c= 200ps)



Faster clock, CPI = 1

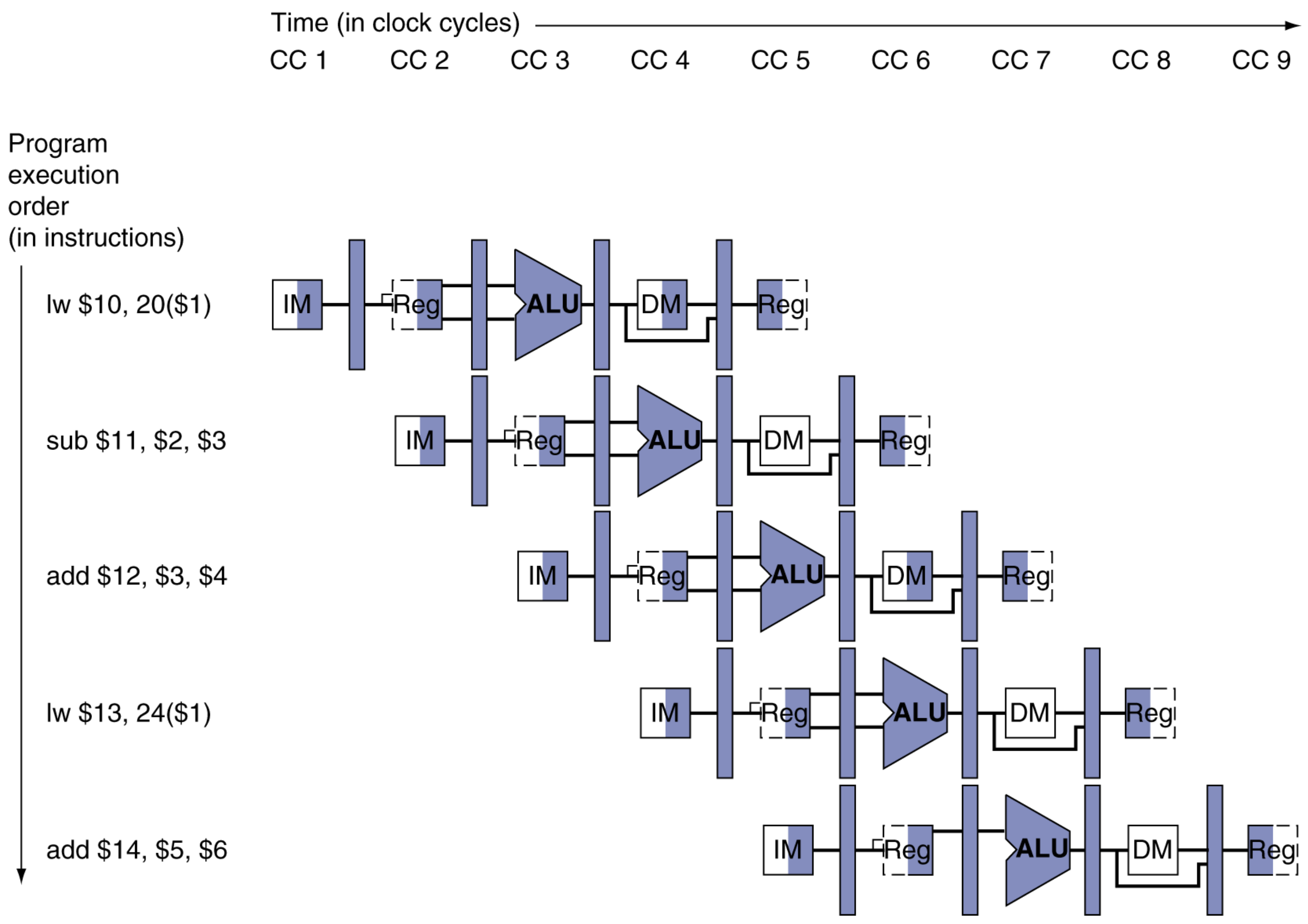
Pipeline Diagram

- Traditional form



Pipeline Diagram

- Form showing resource usage



Pipeline Diagram

- Each stage run different instructions
  - Only one set of hardware

	IF	ID	EX	DM	WB
CC1	I_0				
CC2	I_1	I_0			
CC3	I_2	I_1	I_0		
CC4	I_3	I_2	I_1	I_0	
CC5	I_4	I_3	I_2	I_1	I_0
CC6	I_5	I_4	I_3	I_2	I_1

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - $$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

RISC Strategy

- Clock cycle
- CPI
- Instruction count

Pipelining

- What makes it easy
  - All instructions are the same length
  - Just a few instruction formats
  - Memory operands appear only in loads and stores
- What makes it hard?
  - Structural hazards: suppose we had only one memory
  - Control hazards: need to worry about branch
  - Data hazards: instruction depends on previous one
- We’ll build a simple pipeline and look at these issues

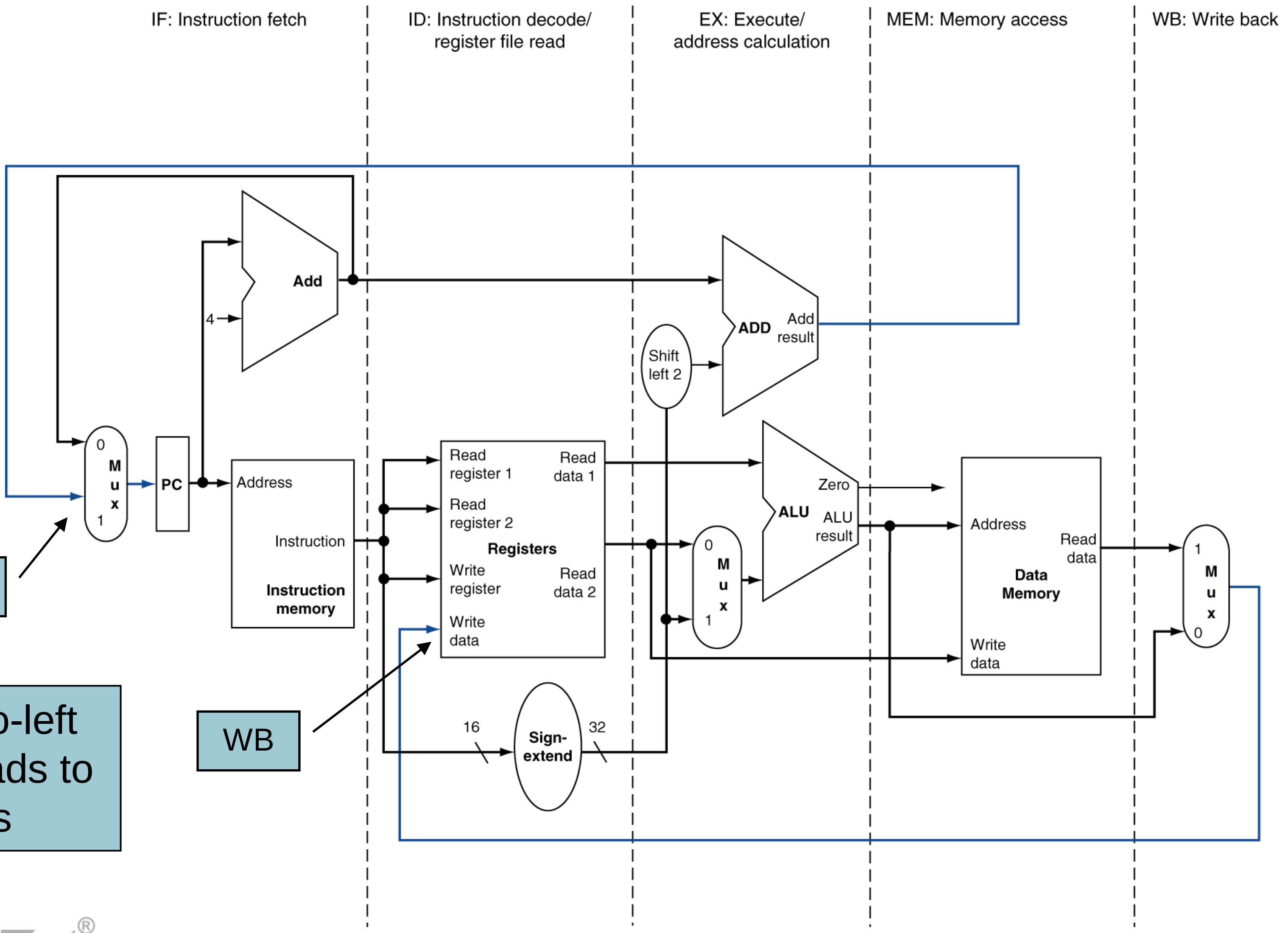
MIPS Pipeline (skip)

- Assume time for stages is
  - 100ps for register read or write (why?)
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register 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

Pipelined Datapath and Control

MIPS Pipelined Datapath



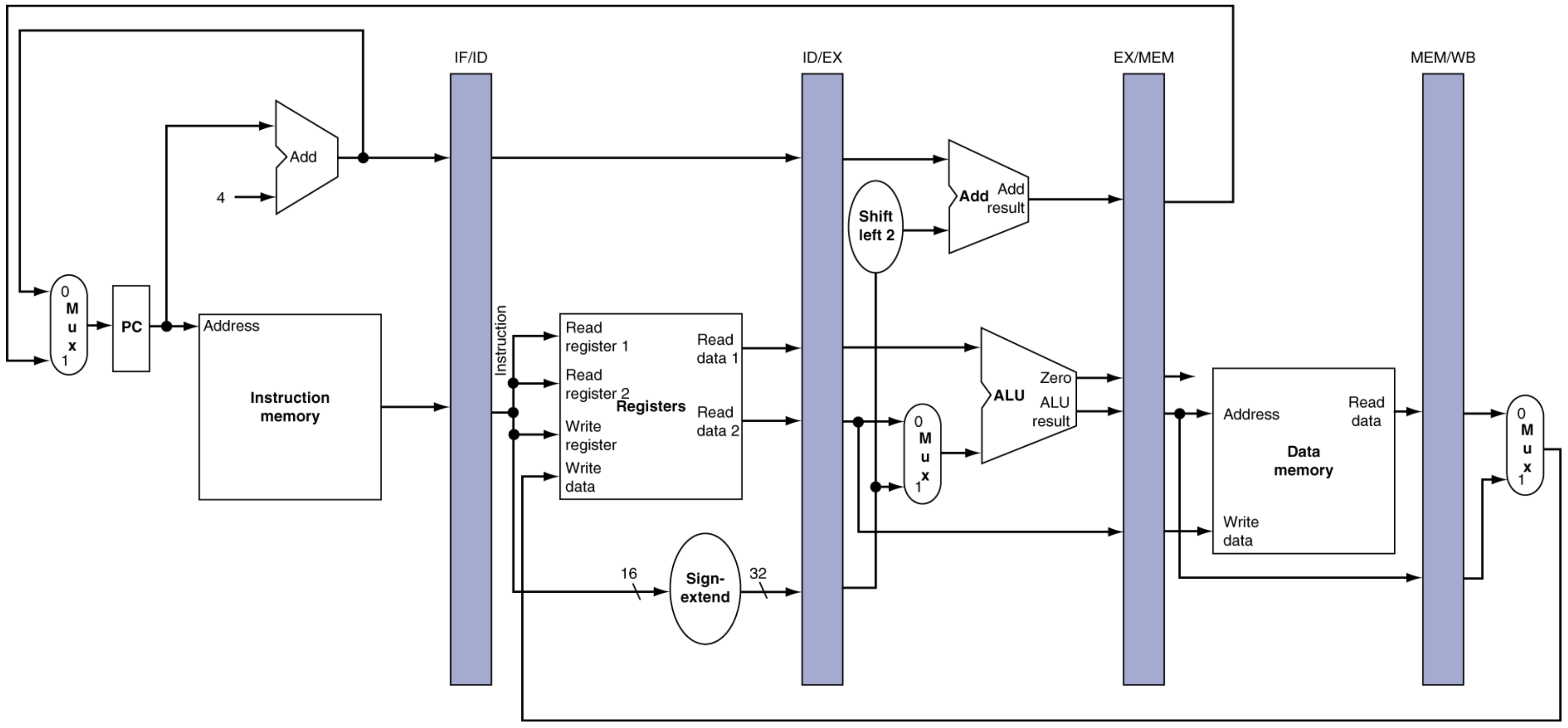
MEM

WB

Right-to-left flow leads to hazards

Pipeline registers

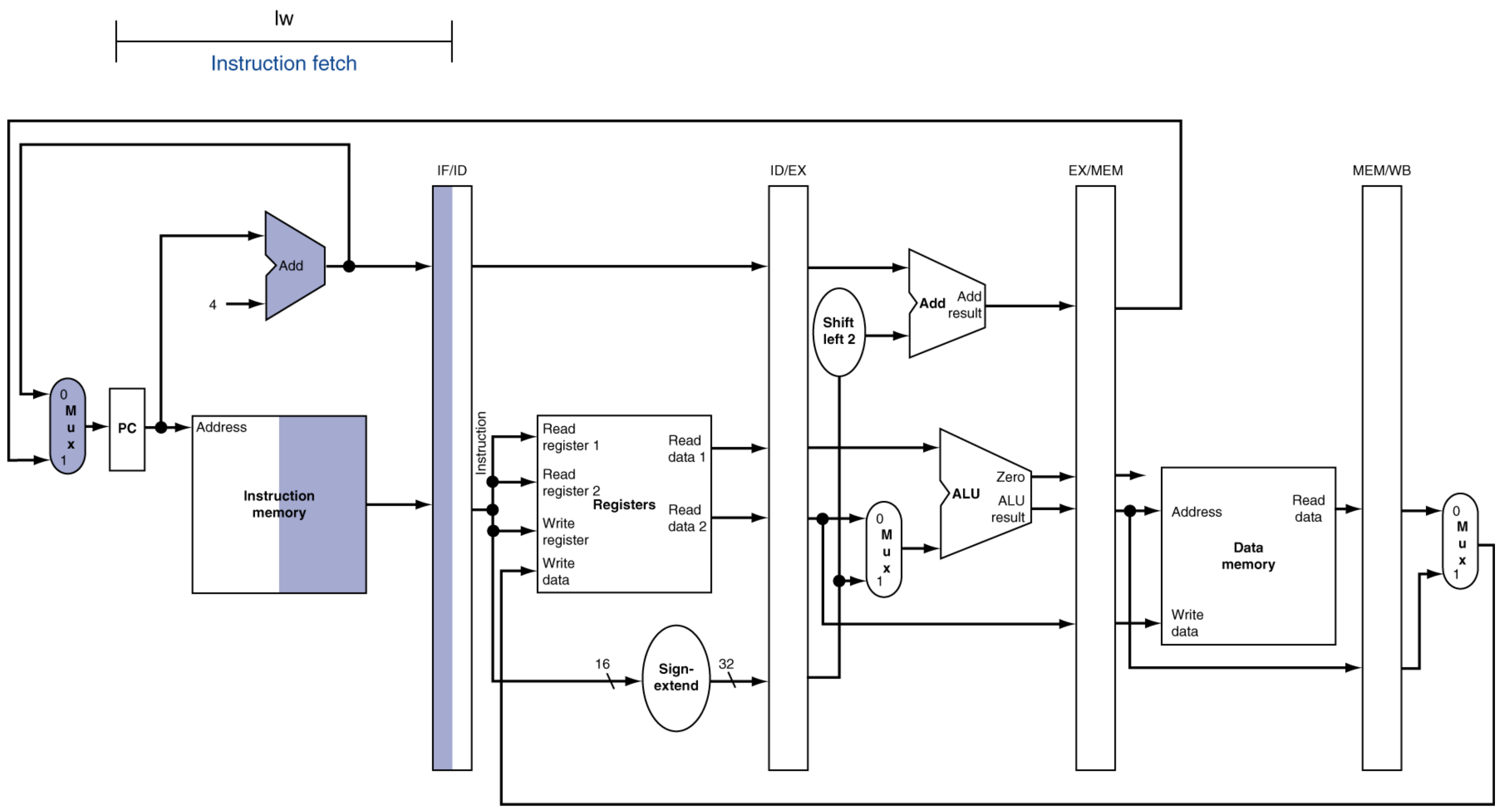
- Need registers between stages
  - To hold information produced in previous cycle



Pipeline Operation

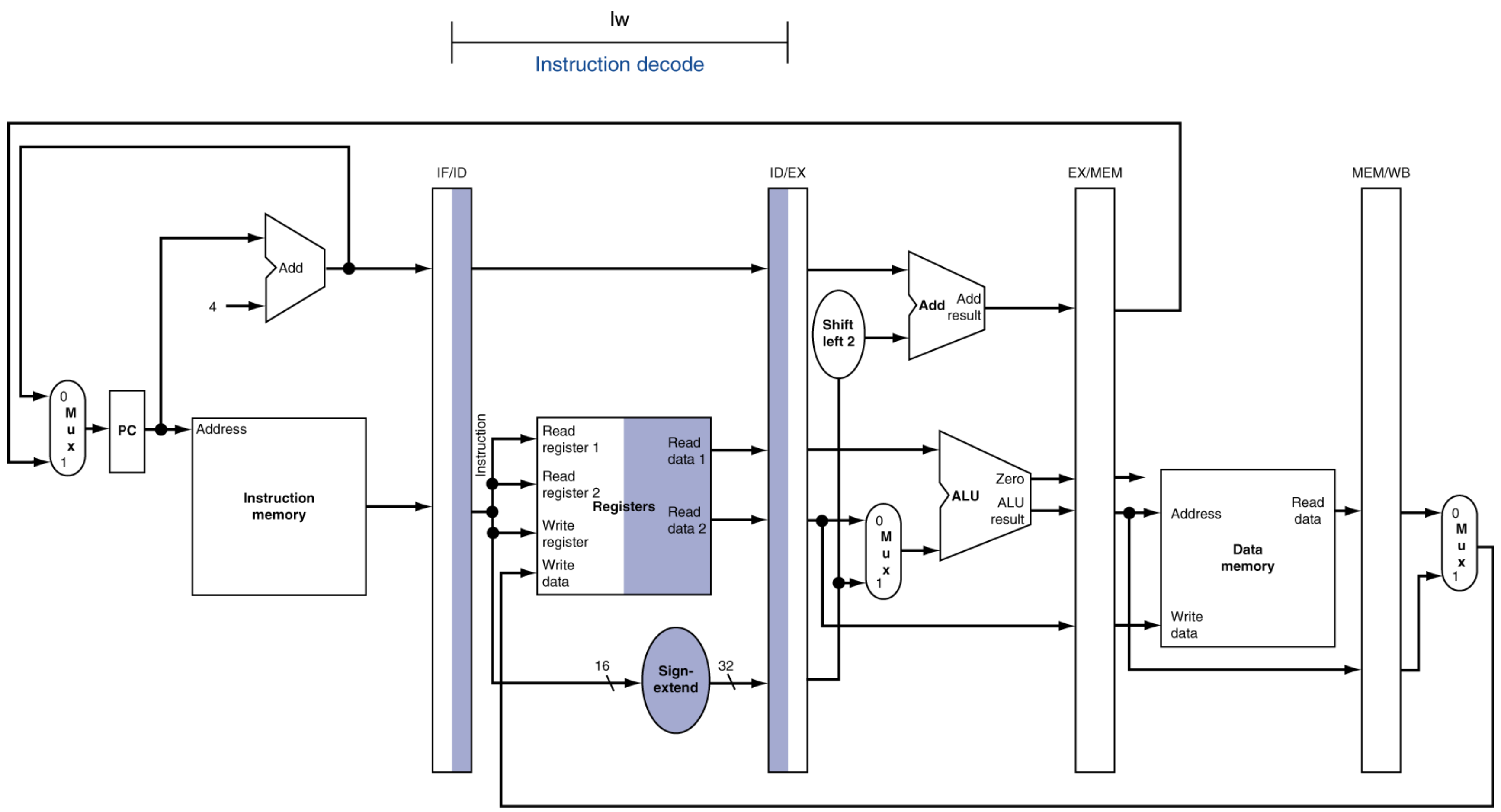
- Cycle-by-cycle flow of instructions through the pipelined datapath
  - We'll look at diagrams for load & store

IF for Load, Store, …



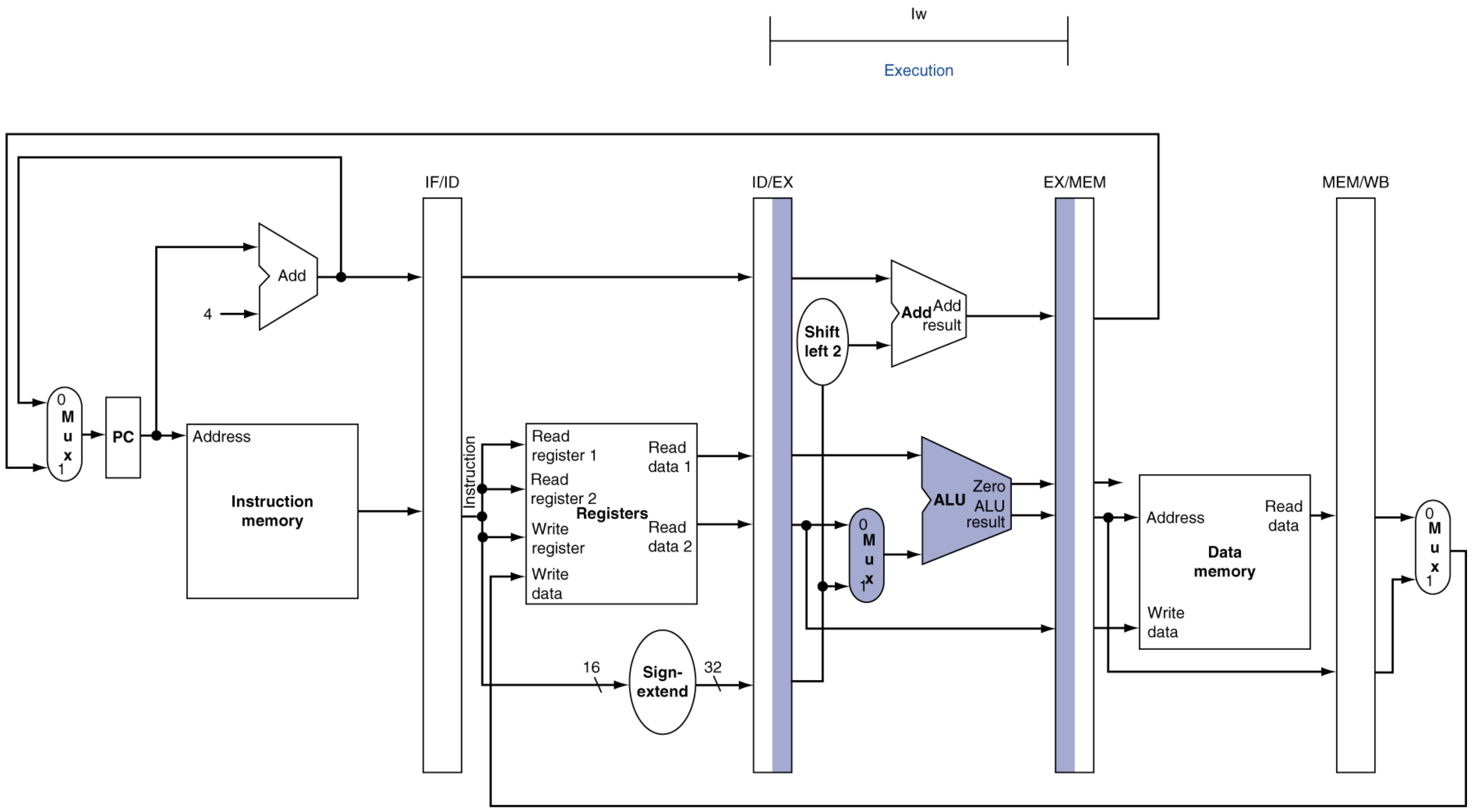
- What kinds of information do we carry with IF/ID latch?

ID for Load, Store, …



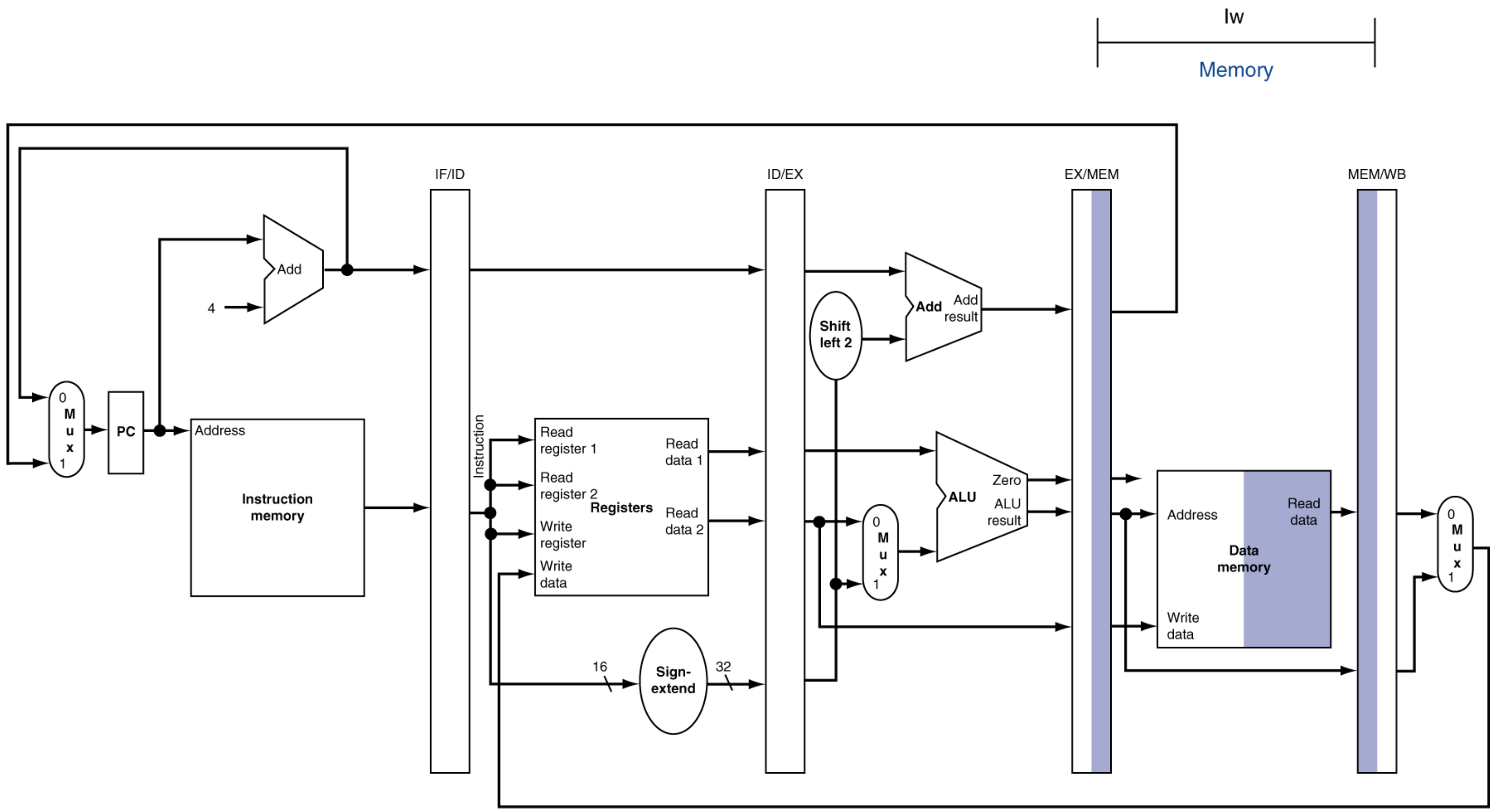
- What kinds of information do we carry with ID/EX latch?

EX for Load



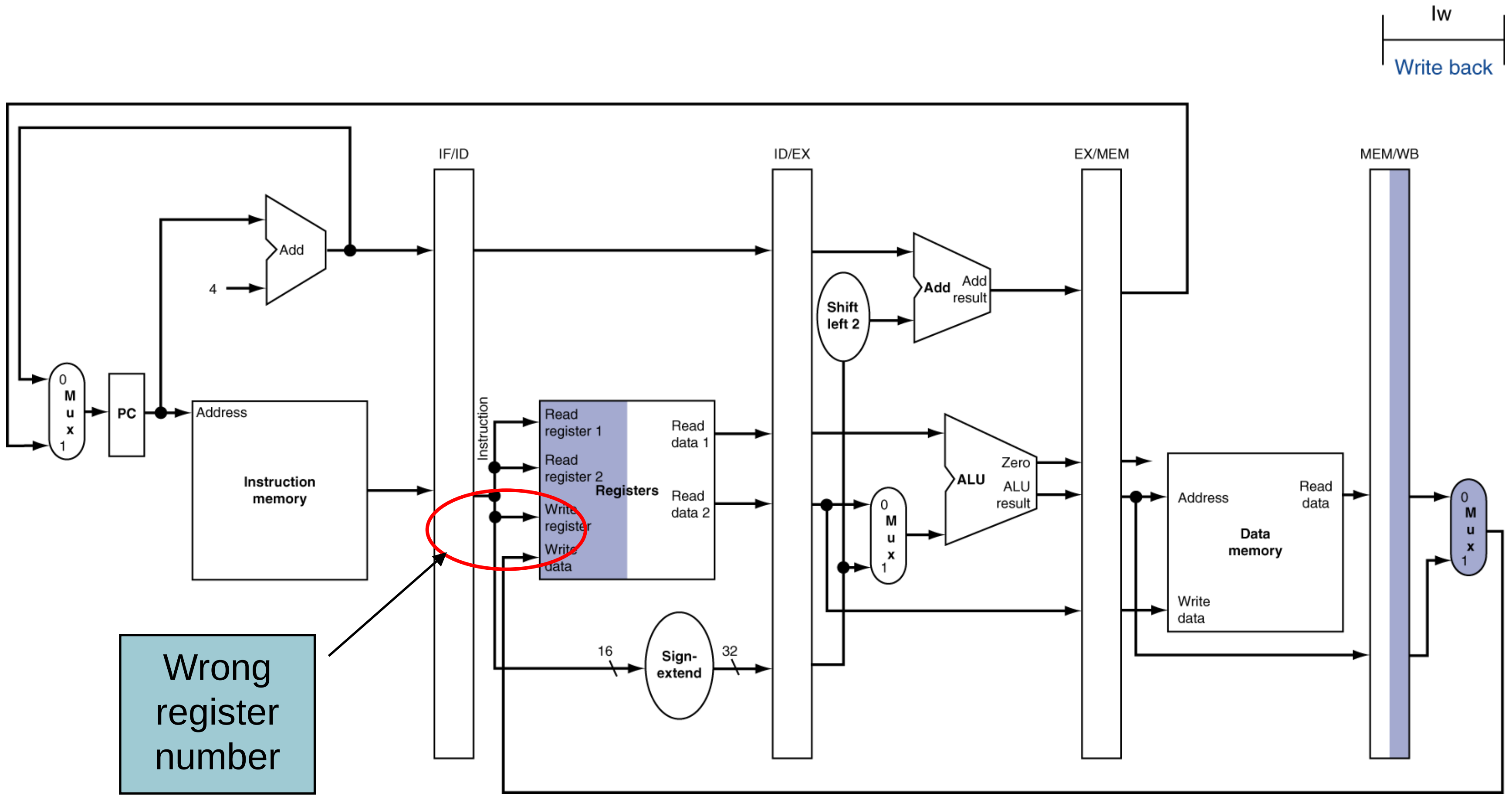
- What kinds of information do we carry with EX/MEM latch?

MEM for Load



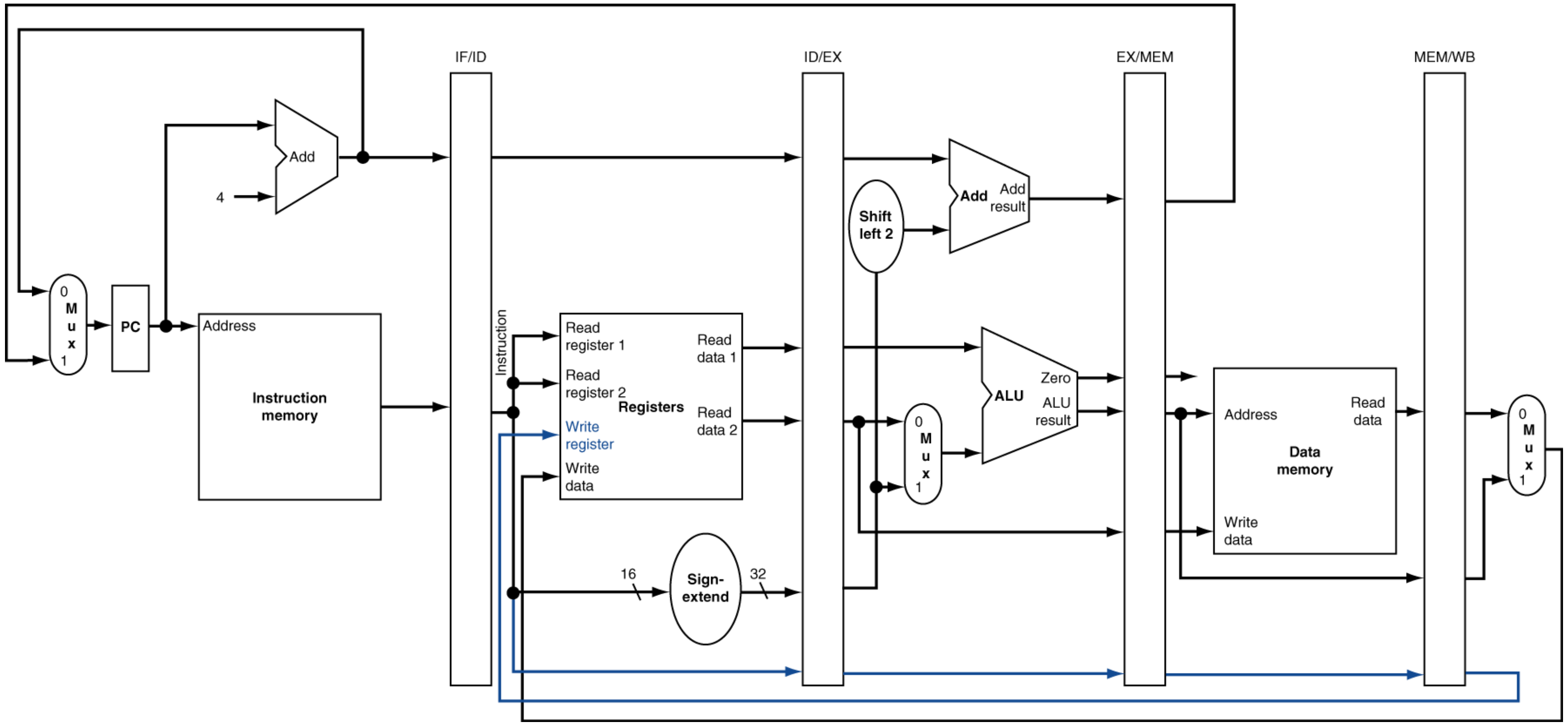
- What kinds of information do we carry with MEM/WB latch?

WB for Load

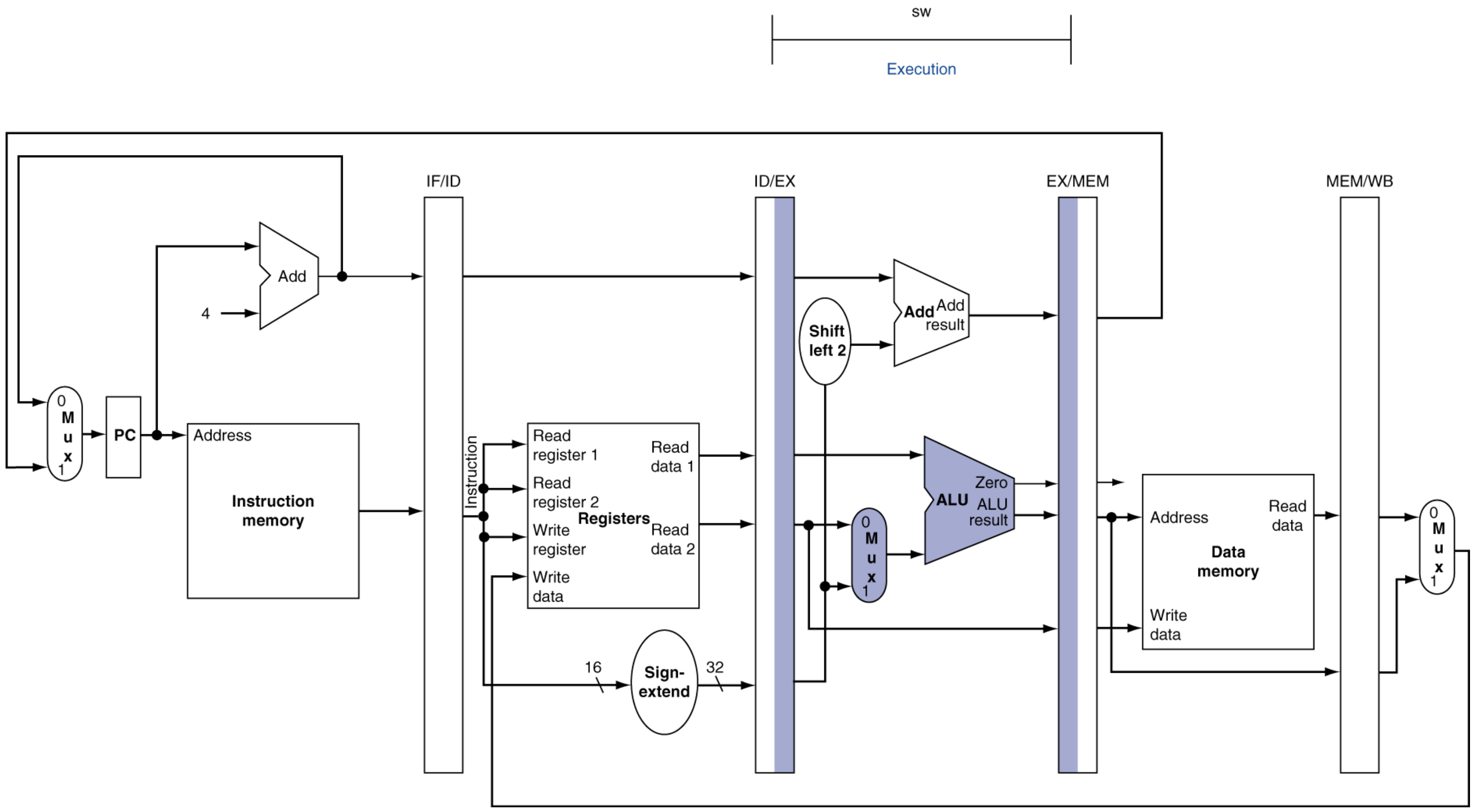


- What kinds of information do we carry with MEM/WB latch?

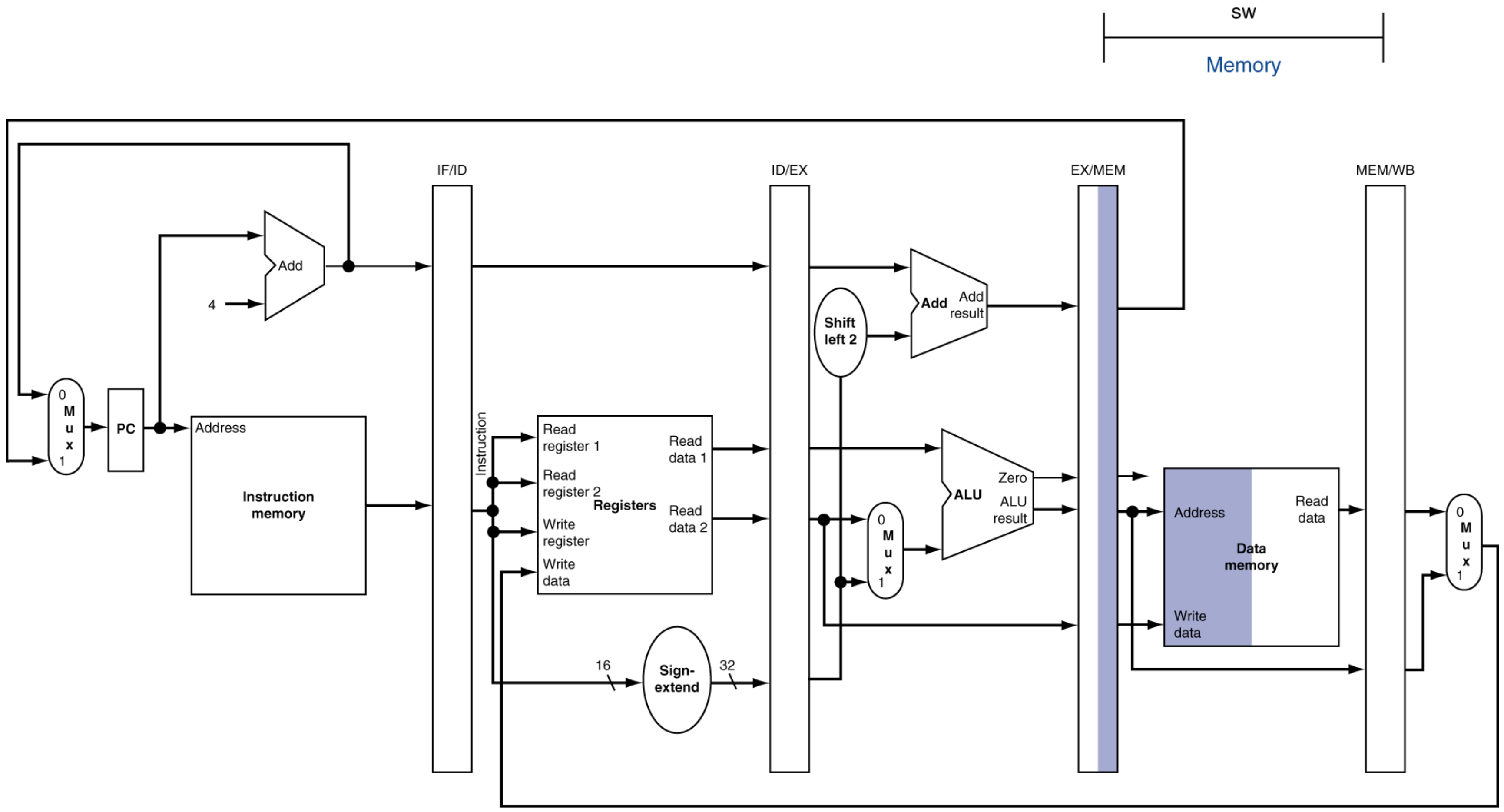
Corrected Datapath for Load



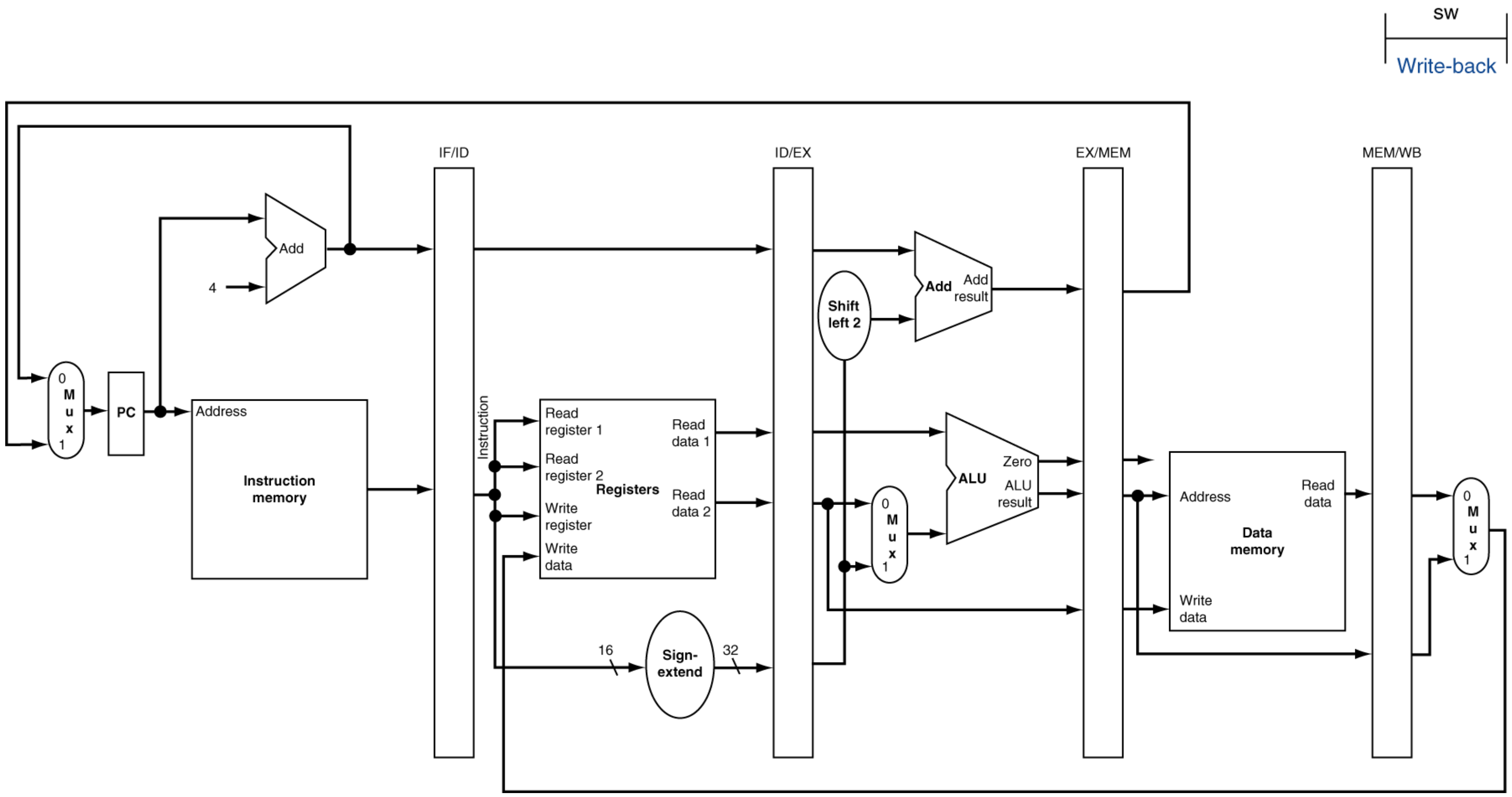
EX for Store


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

MEM for Store



WB for Store



- Doesn't matter whether there is meaningful operation
  - Important: at every cycle, start (or end) one instruction

ALU instruction and Branch

- Operation of R-type instructions (“add”)
  - No operation in MEM
    - Not affect overall performance
    - Important: at every cycle, start one instruction
- Operation of “addi”
- Operation of branch instructions (“beq”)
  - Finished in MEM, no operation in WB

Pipelining vs. Multicycle

- Each stage run different instructions

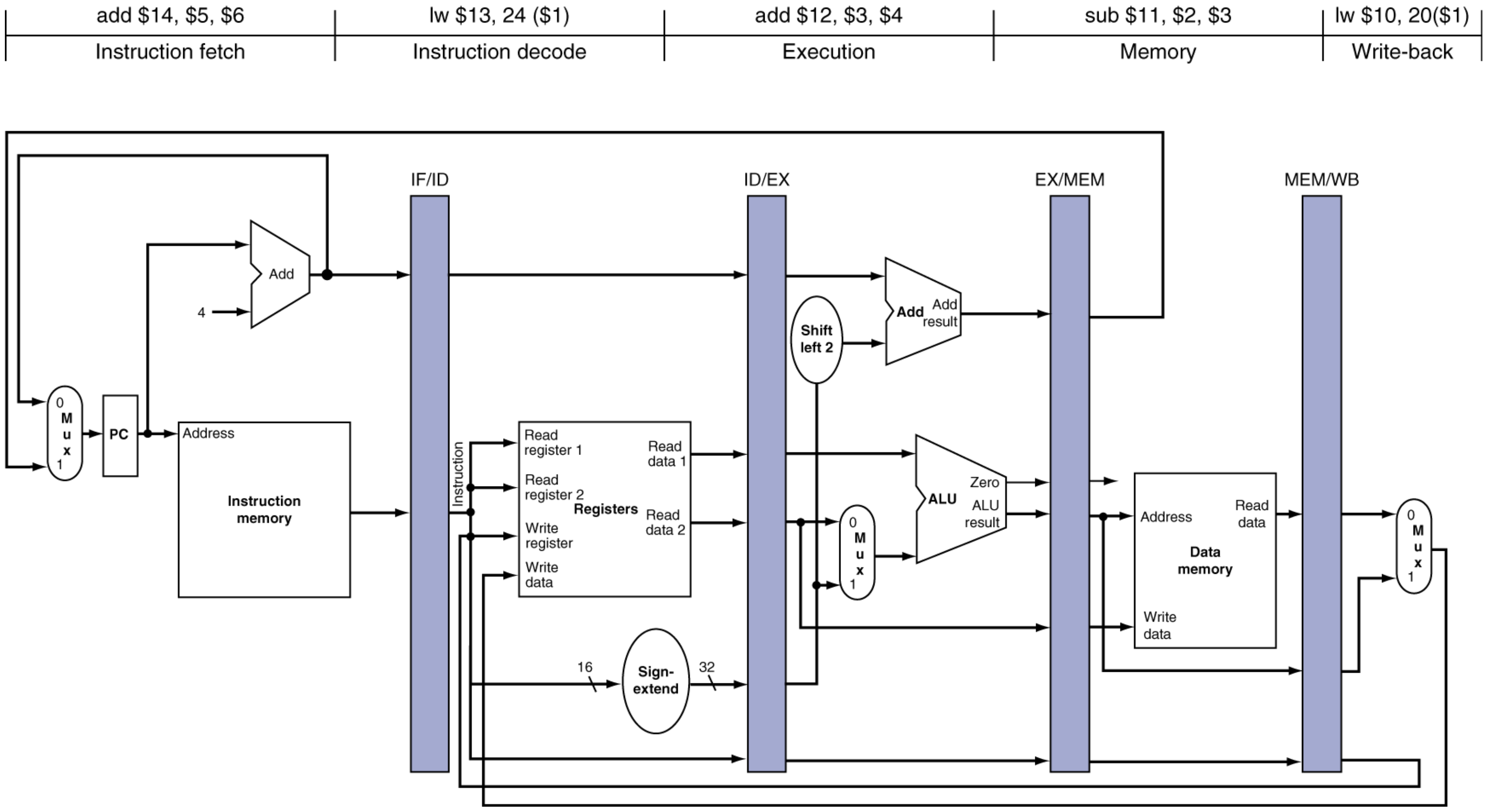
	IF	ID	EX	DM	WB
CC1	I_0				
CC2	I_1	I_0			
CC3	I_2	I_1	I_0		
CC4	I_3	I_2	I_1	I_0	
CC5	I_4	I_3	I_2	I_1	I_0
CC6	I_5	I_4	I_3	I_2	I_1

Time $\downarrow$

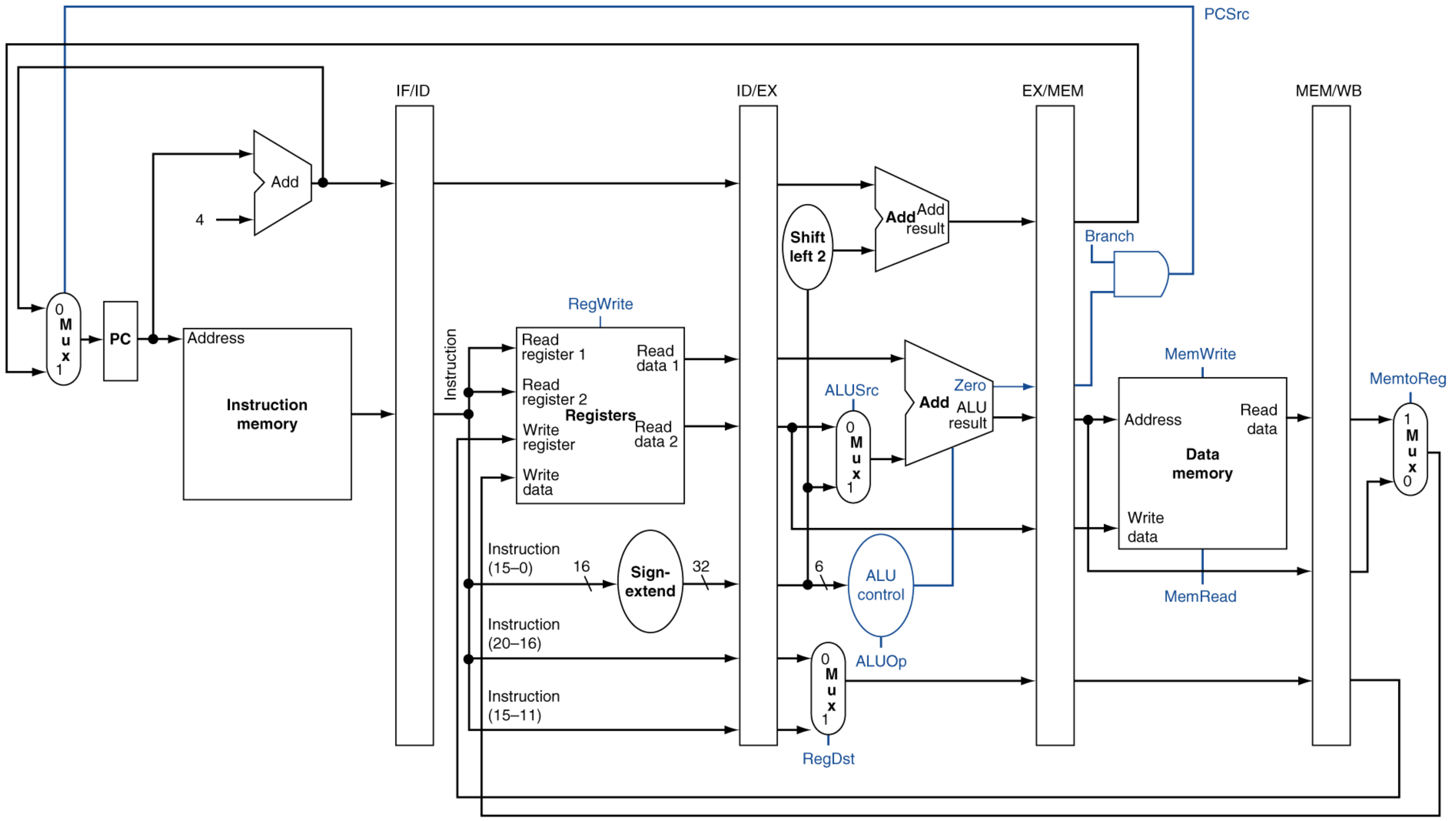
	IF	ID	EX	DM	WB
CC1	I_0				
CC2		I_0			
CC3			I_0		
CC4				I_0	
CC5					I_0
CC6	I_1				

Pipeline Diagram

- State of pipeline in a given cycle

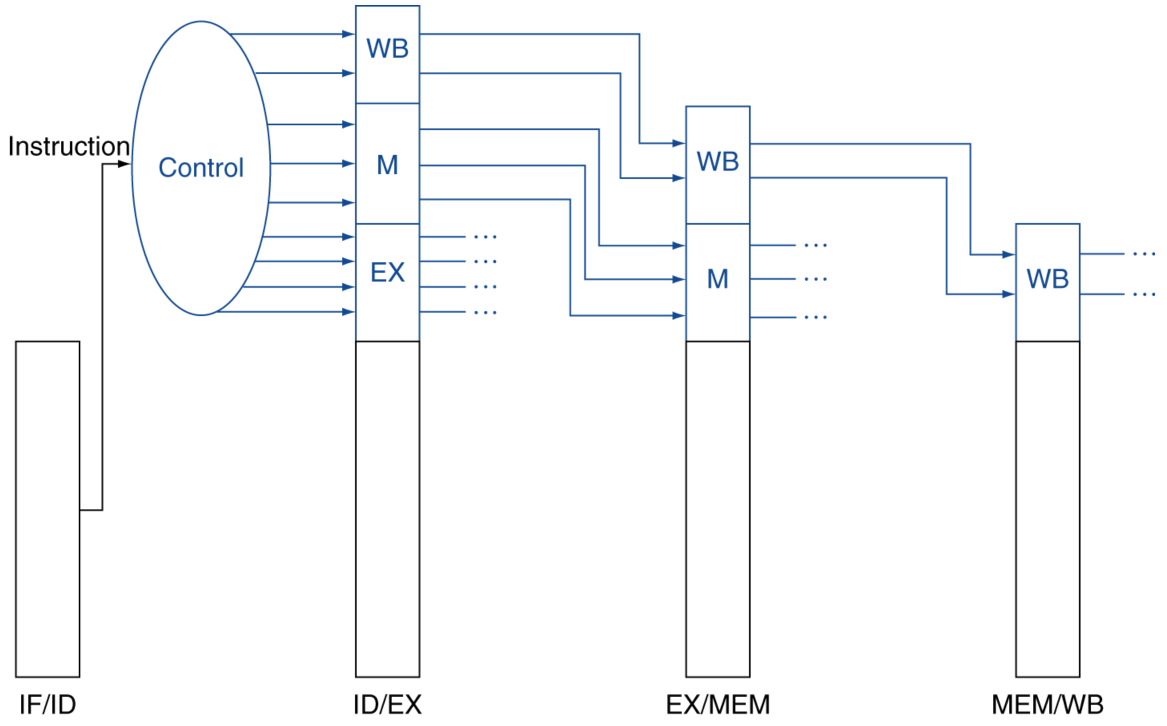


Pipelined Control (Simplified)

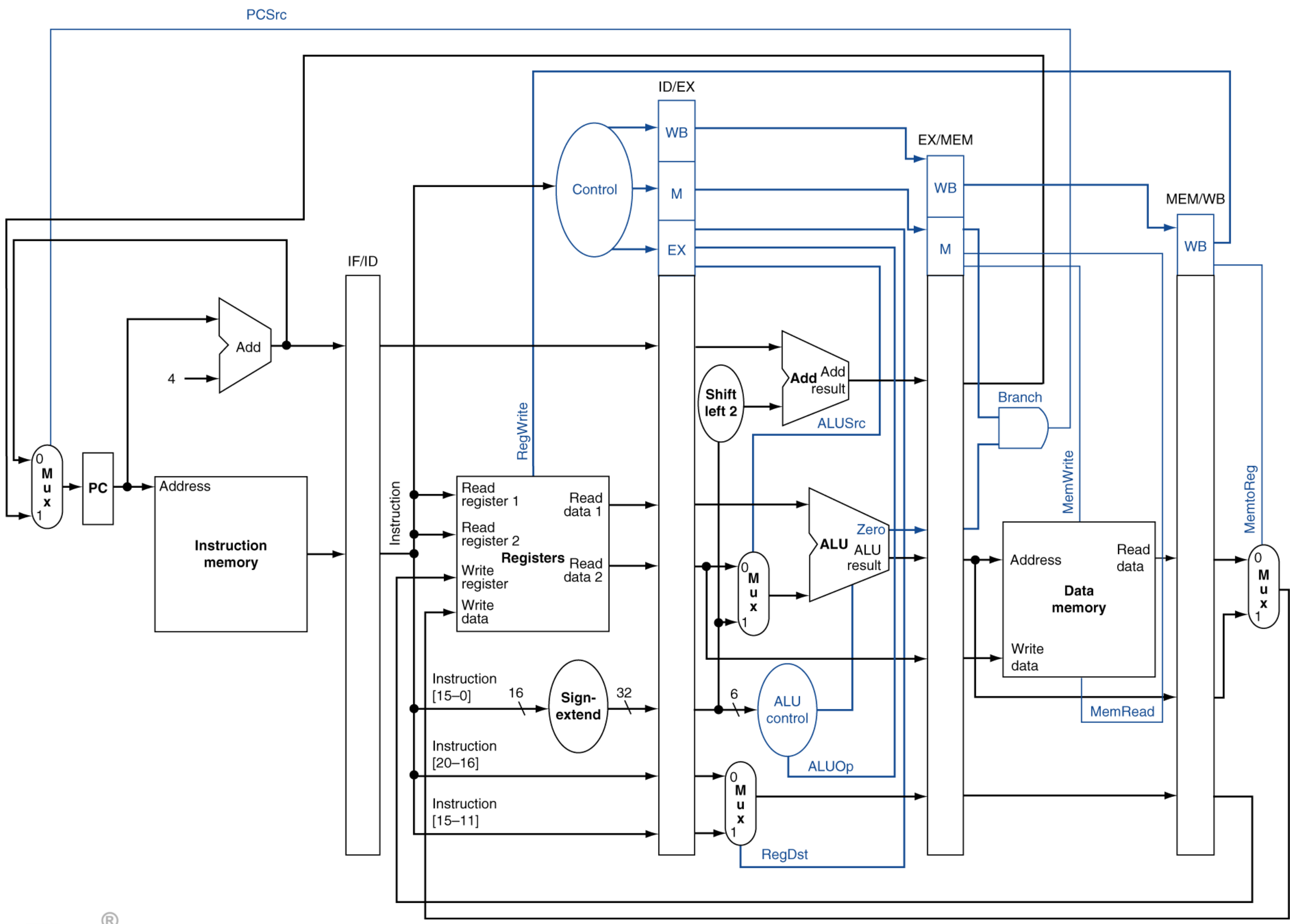


Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



Pipelined Control

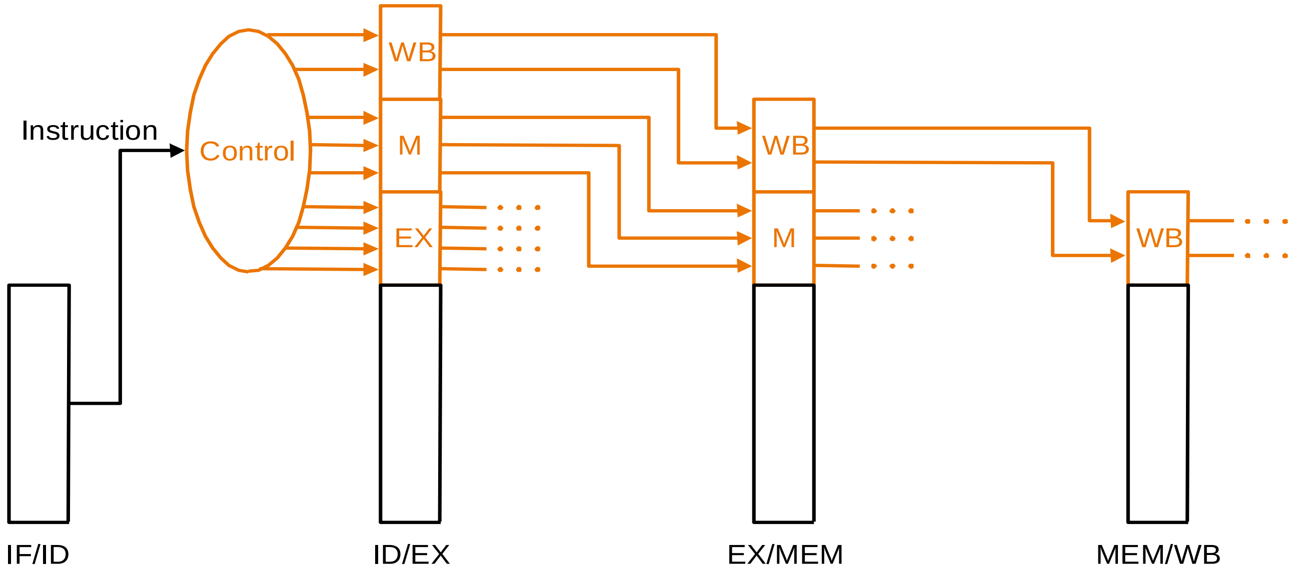


Pipeline Control

- Pass control signals along just like the data

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

† Note the register destination signal



Pipelining and ISA Design

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

Pipeline Hazards

Hazards

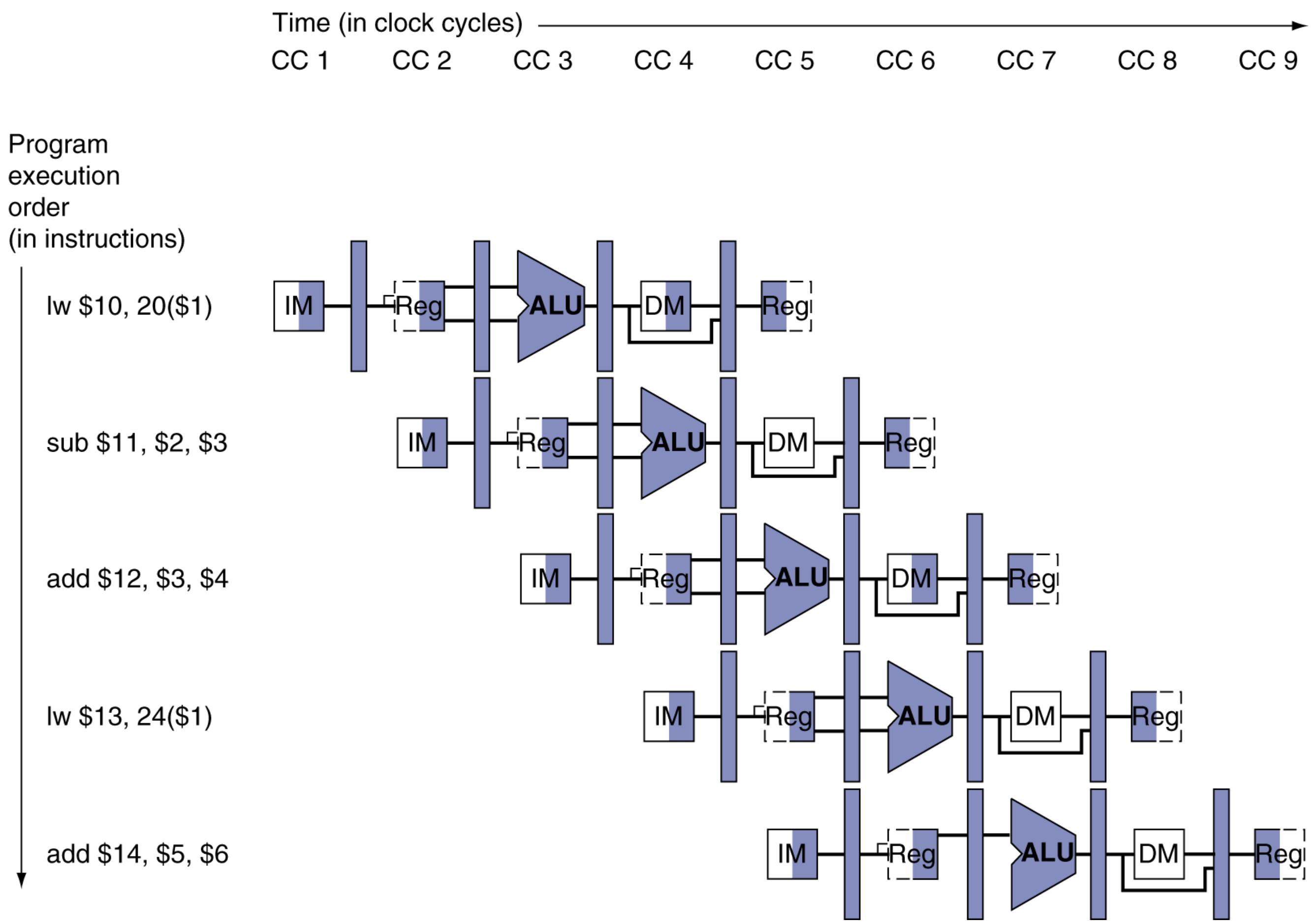
- Situations that prevent starting the next instruction in the next cycle
- 1) Structure hazards
  - A required resource is busy
- 2) Data hazard
  - Need to wait for previous instruction to complete its data read/write
- 3) Control hazard
  - Deciding on control action depends on previous instruction

Structural Hazards

- Resource conflicts
  - Different stages want to use same hardware resource
    - Can avoid with hardware duplication
- Our datapath has no structural hazards, but what if
  - Instruction and data memory not separated
    - fetch vs. DM of lw/sw
  - Not separate adder for PC + 4
  - Register access
    - ID vs. write back
- Sometimes, can be better not to pipeline
  - Return on investment (e.g., divider)

Structural Hazards

- Integrated IM and DM – impact on CPI?



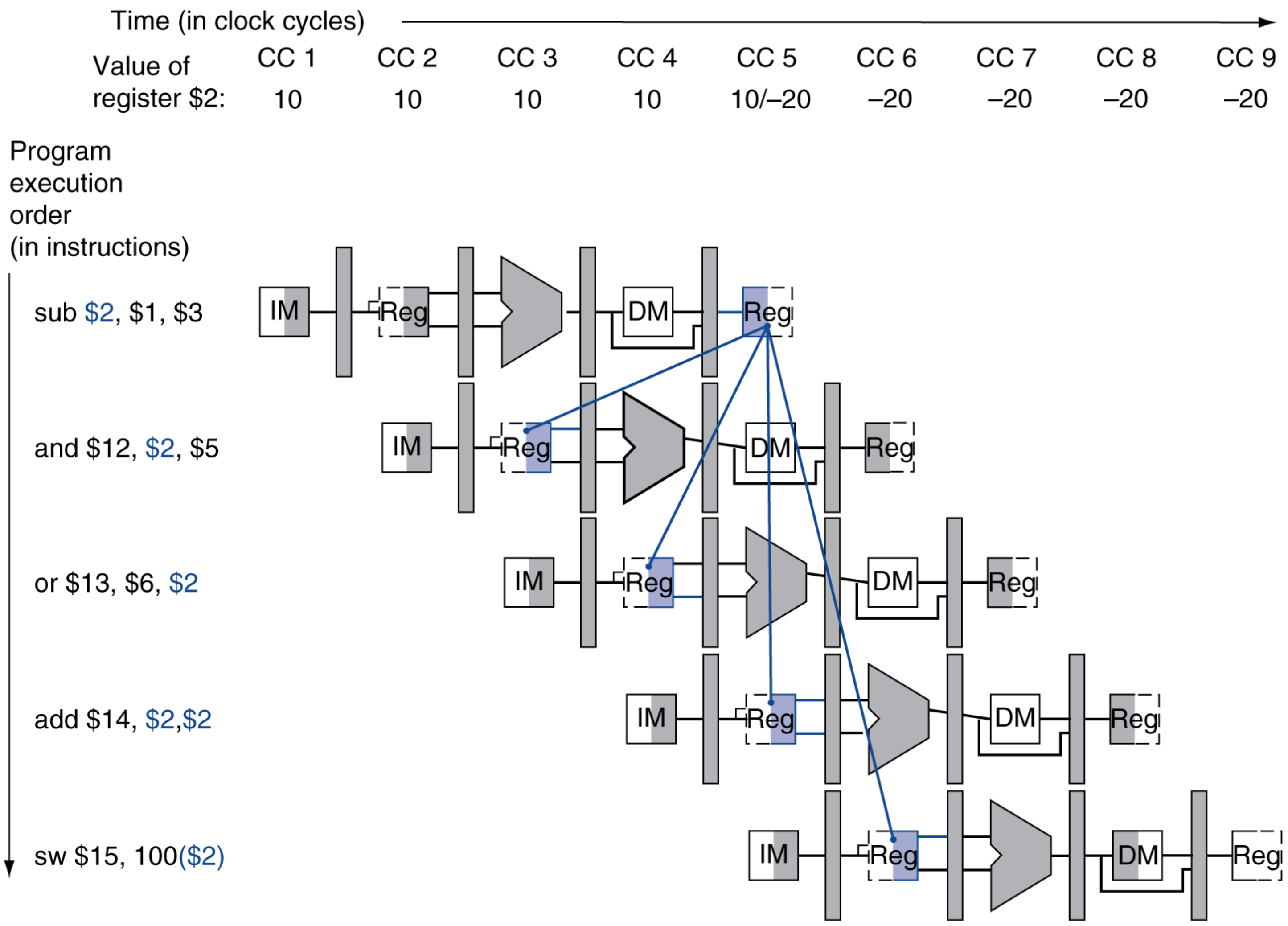
Data Hazards in ALU Instructions

- Consider this sequence:

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

- We can resolve hazards with forwarding
  - How do we detect when to forward?

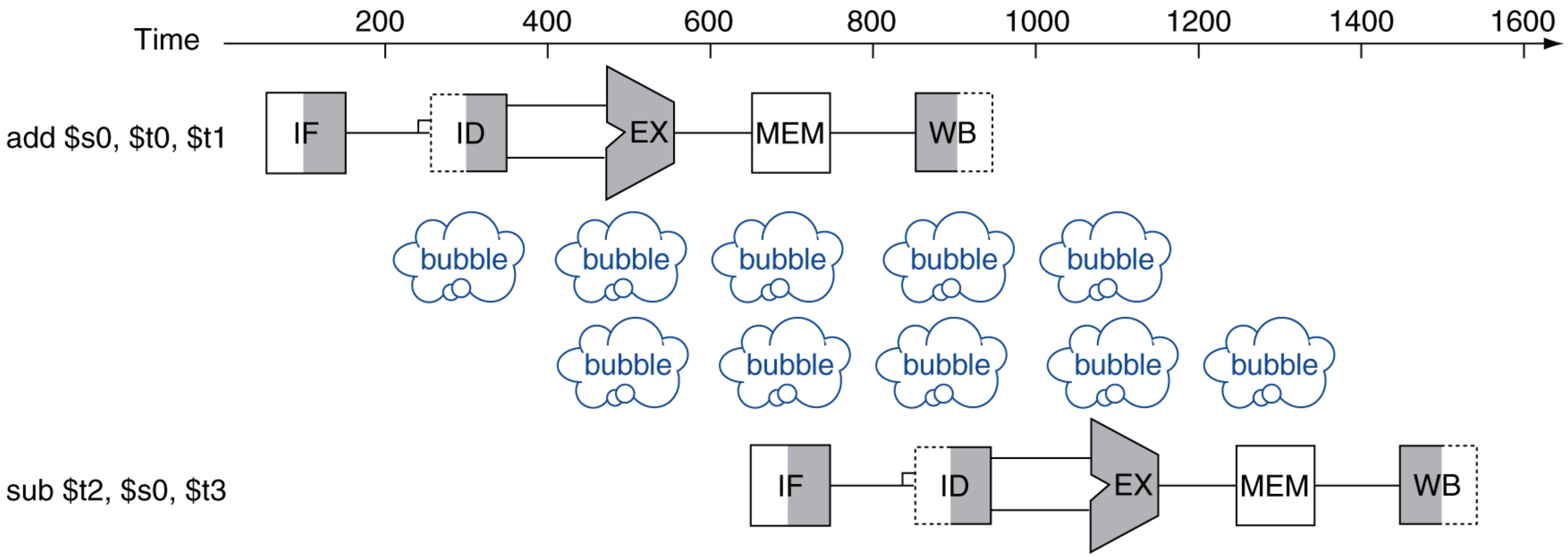
Dependencies & Forwarding



Data Hazards

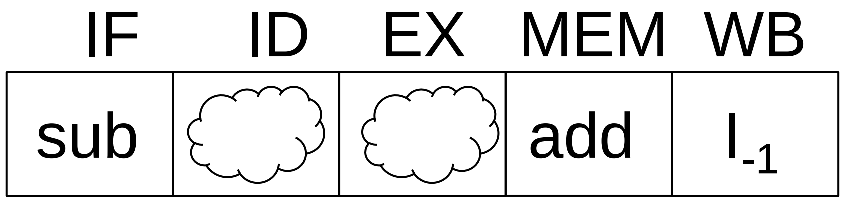
- An instruction depends on completion of data access by a previous instruction

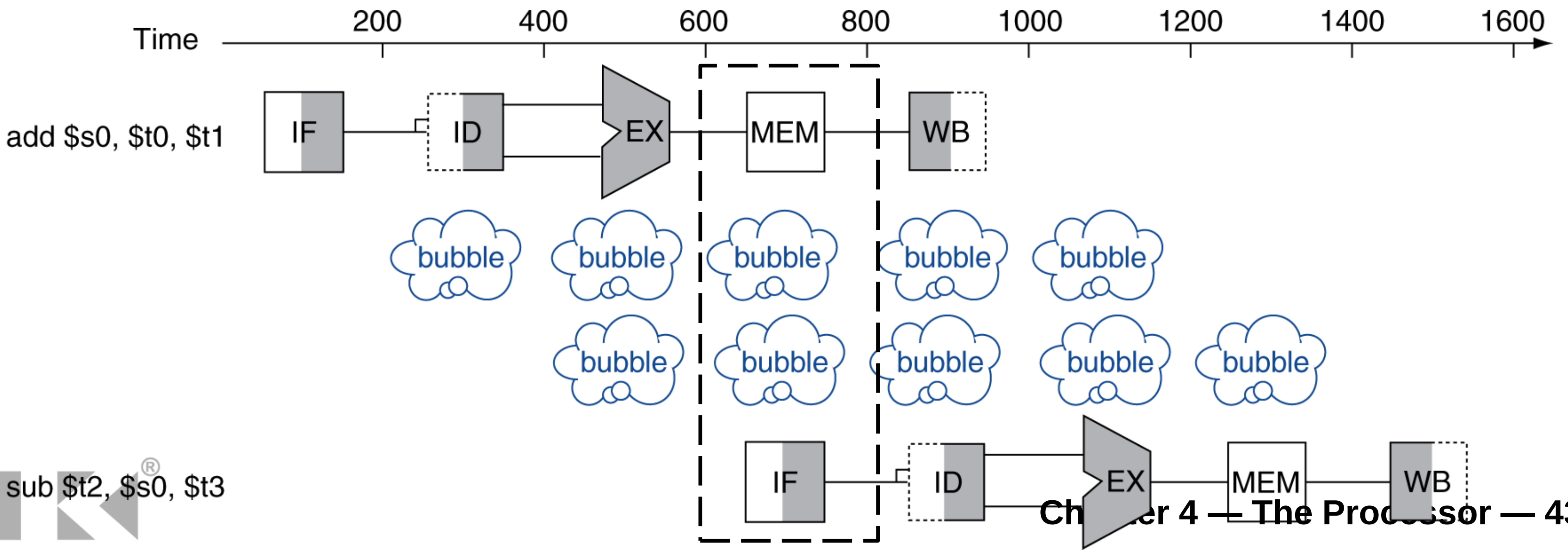
  -
    ```
    add  $s0, $t0, $t1
    sub  $t2, $s0, $t3
    ```



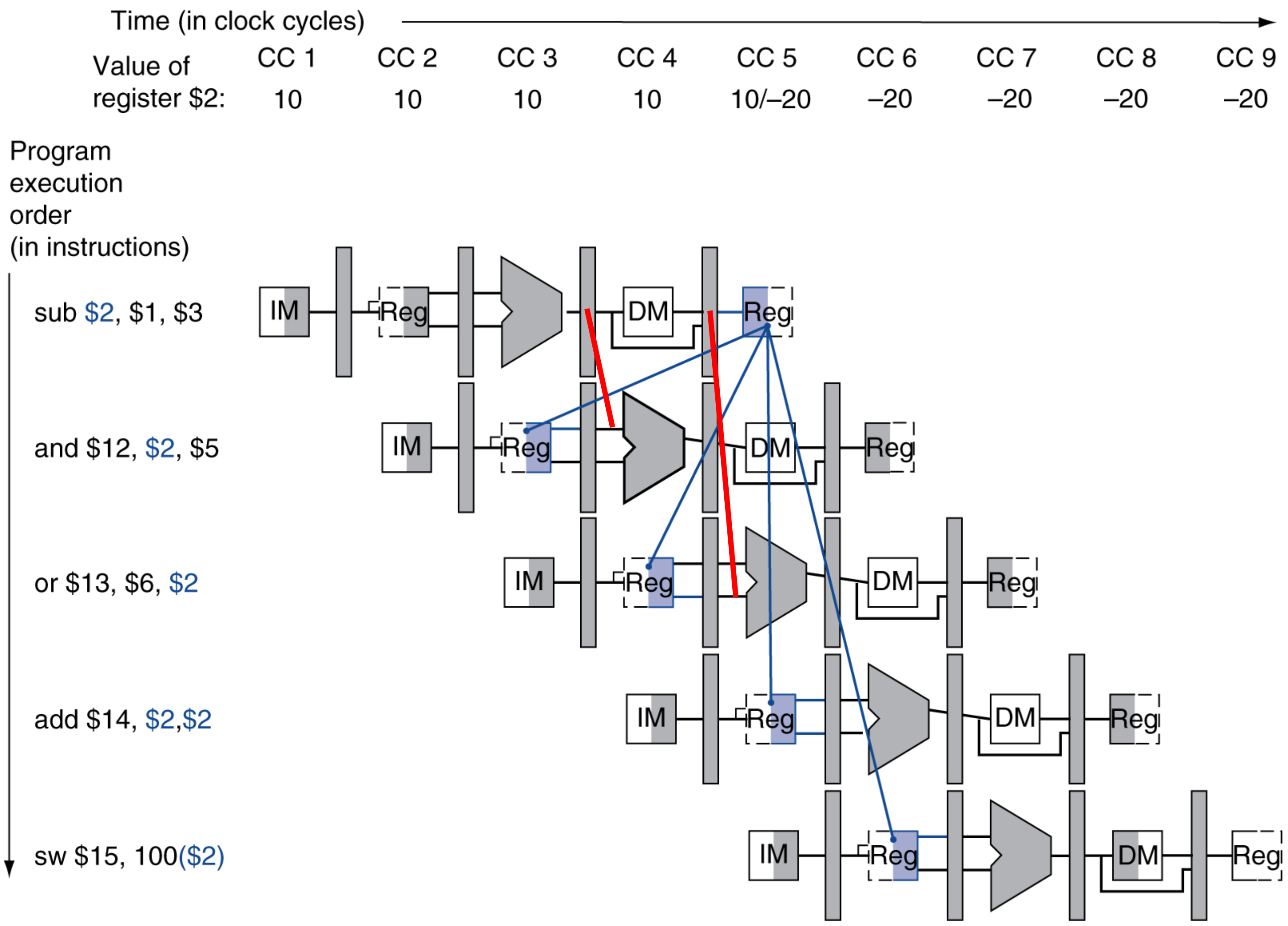
Pipeline Stall, Bubble

- Lose two cycles: increase in CPI
  - What is bubble: NOP instruction



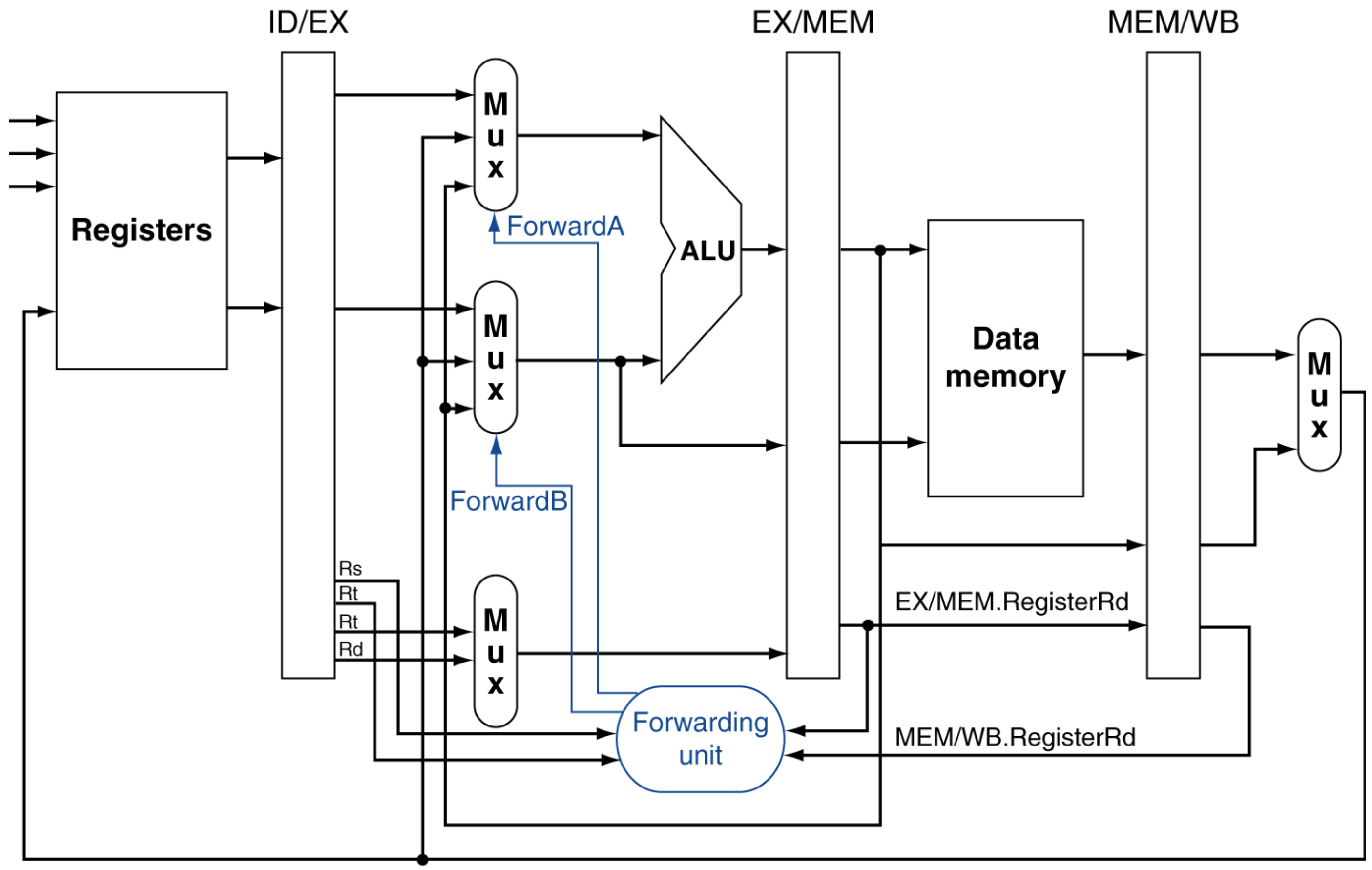


Dependencies & Forwarding



Forwarding Paths

- More datapath and control



b. With forwarding

Think about Forwarding

- Source: instructions that generate new value (producer)
  - New values generated by ALU or data memory read
  - Non-producer: **sw, beq**
- Data receiver: data consumer
  - All instructions
    - For ALU operations or write to data memory
- To sum up
  - Forwarding source:

    ALU result, delayed ALU results, result of **lw**
  - Destination of forwarding: ALU inputs, write data to DM

Think about Forwarding

- Forwarding to ALU inputs
  - From EX/MEM latch (I_{-1}) or MEM/WB latch (I_{-2})
  - Results of I_{-3} available from registers
- Will not cover forwarding to DM write data input
  - Your personal exercise

Detecting the Need to Forward

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

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg

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

Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

- 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

Forwarding Conditions

- EX 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 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

Think about Forwarding

- Difference between **add** and **addi** instructions
  - Two ALU inputs vs. one ALU input
  - Why do we ignore it?
    - ALU do not use that input anywat
      - Note that datapath for 16-bit immediate part not shown in figure

Double Data Hazard

- Consider the sequence:

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

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn’t true

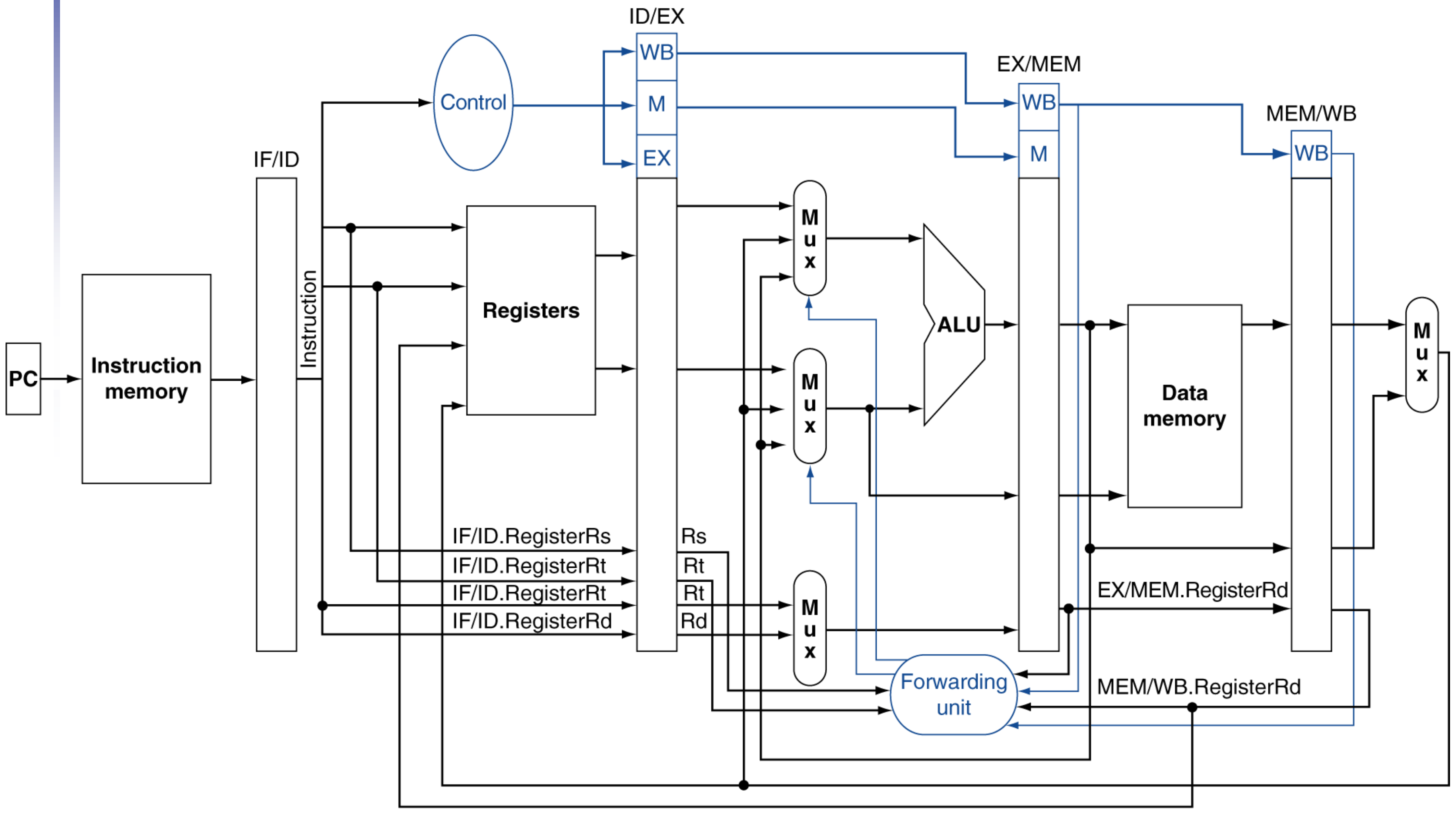
Revised Forwarding Condition

- 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
    ```

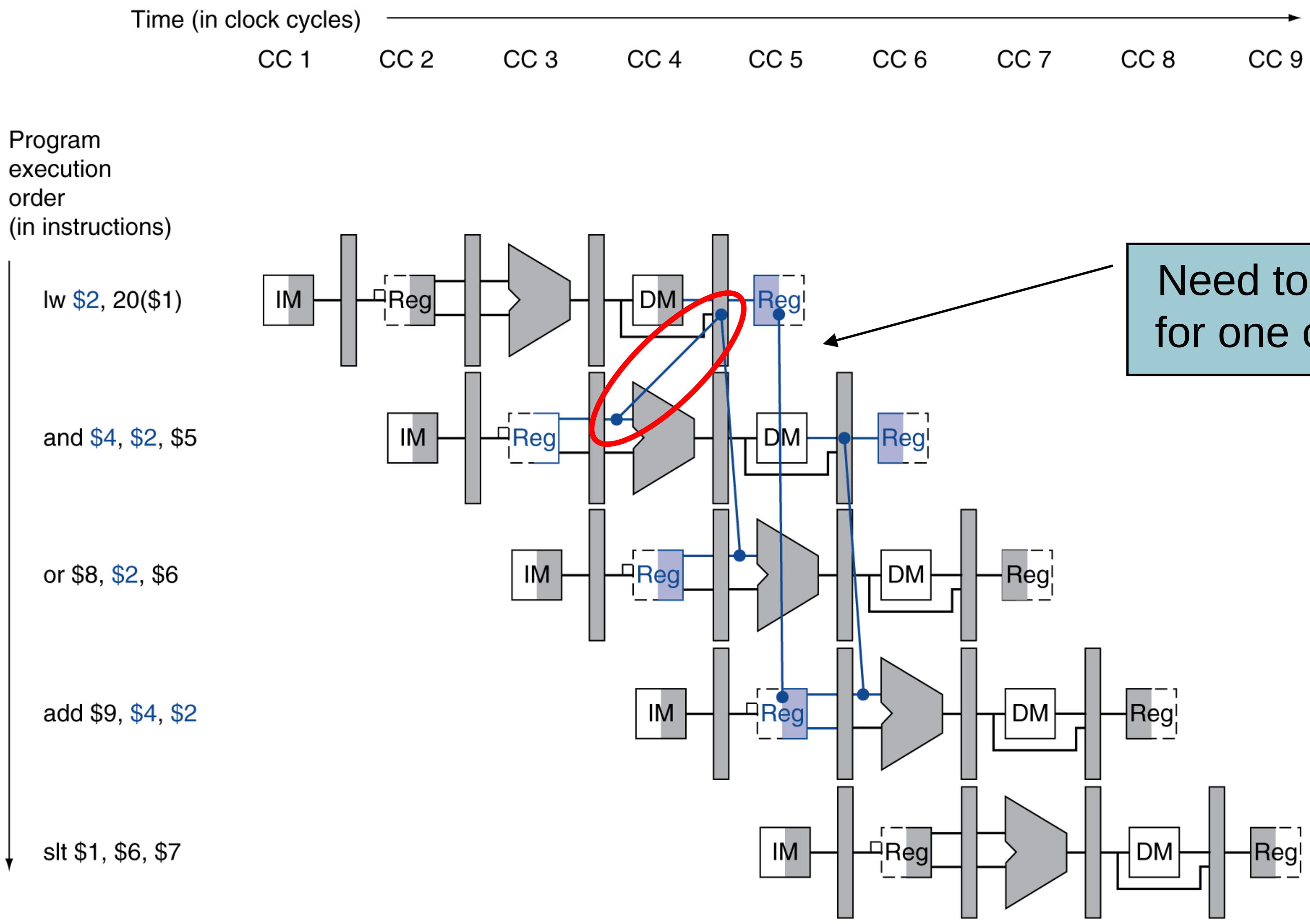
Datapath with Forwarding



Load-Use Data Hazard

- Only data hazard that require stall in our 5-stage design

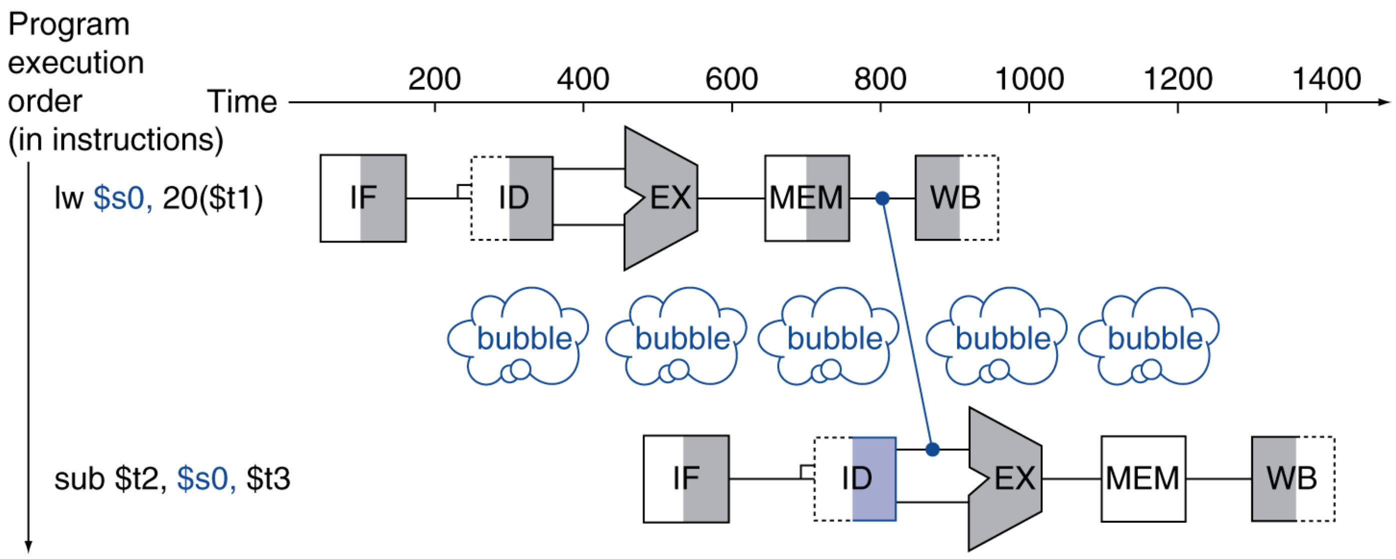
Load-Use Data Hazard



Need to stall for one cycle

Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



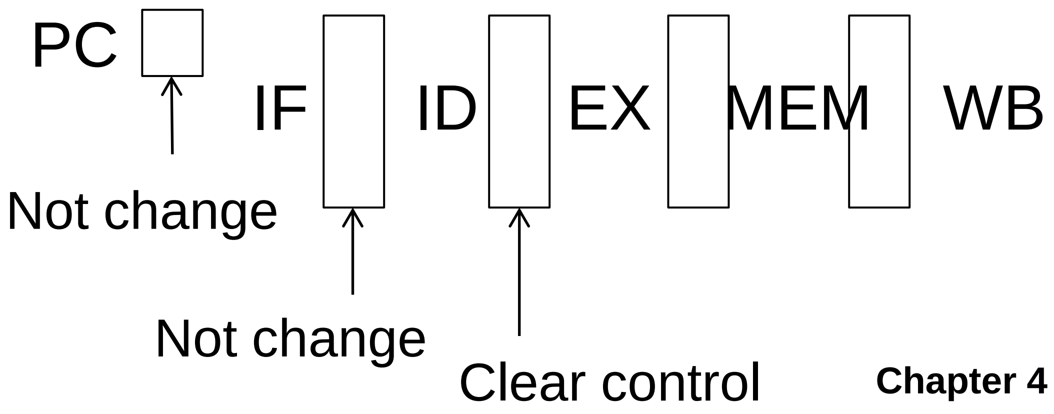
Load-Use Hazard Detection

- 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-use hazard 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

	IF	ID	EX	MEM	WB
CC1	or	and	lw	I_{-1}	I_{-2}
CC2	or	and	nop	lw	I_{-1}
CC3	add	or	and	nop	lw
CC4	slt	add	or	and	nop
CC5	I_{+1}	slt	add	or	and

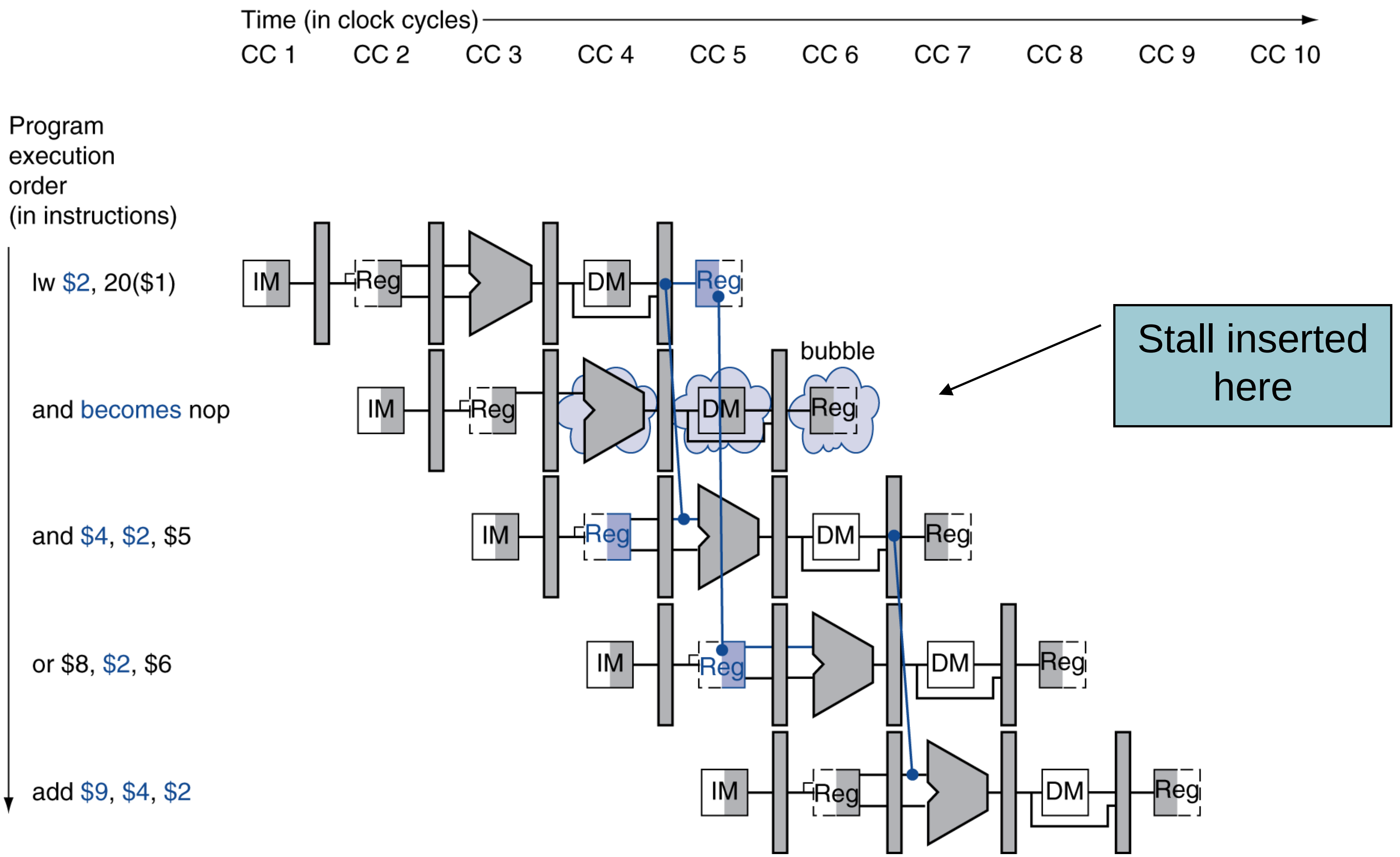
- To insert NOP (or bubble), at the end of clock, do:



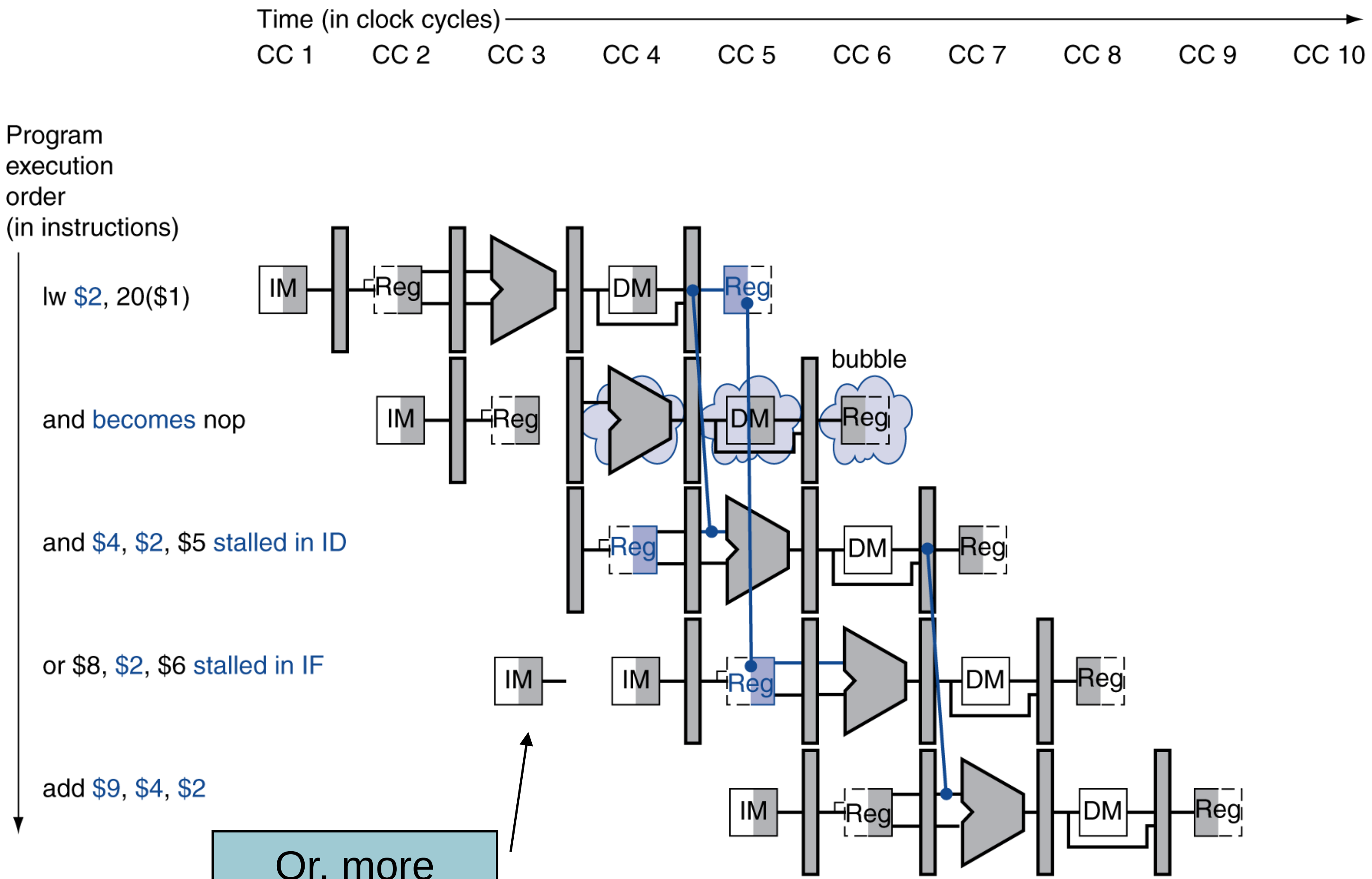
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX do "nop" (no-operation)
- Prevent update of PC and IF/ID register
  - ID stage: same instruction is decoded again
  - IF stage: same 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 (skip)

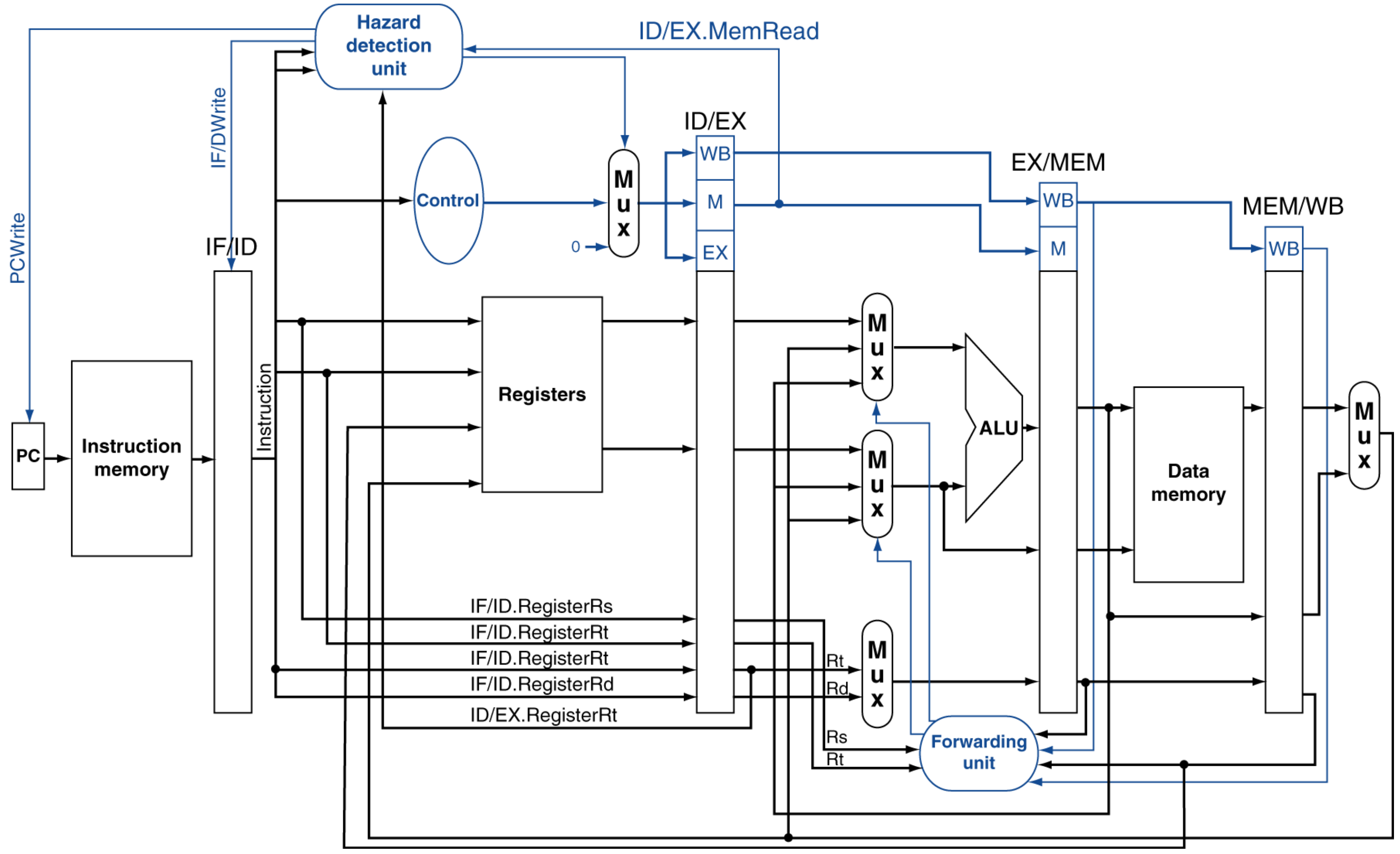


Stall/Bubble in the Pipeline (skip)



Or, more accurately…

Datapath with Hazard Detection



- More datapath and control

Software Solution

- Compiler can help

Software Solution

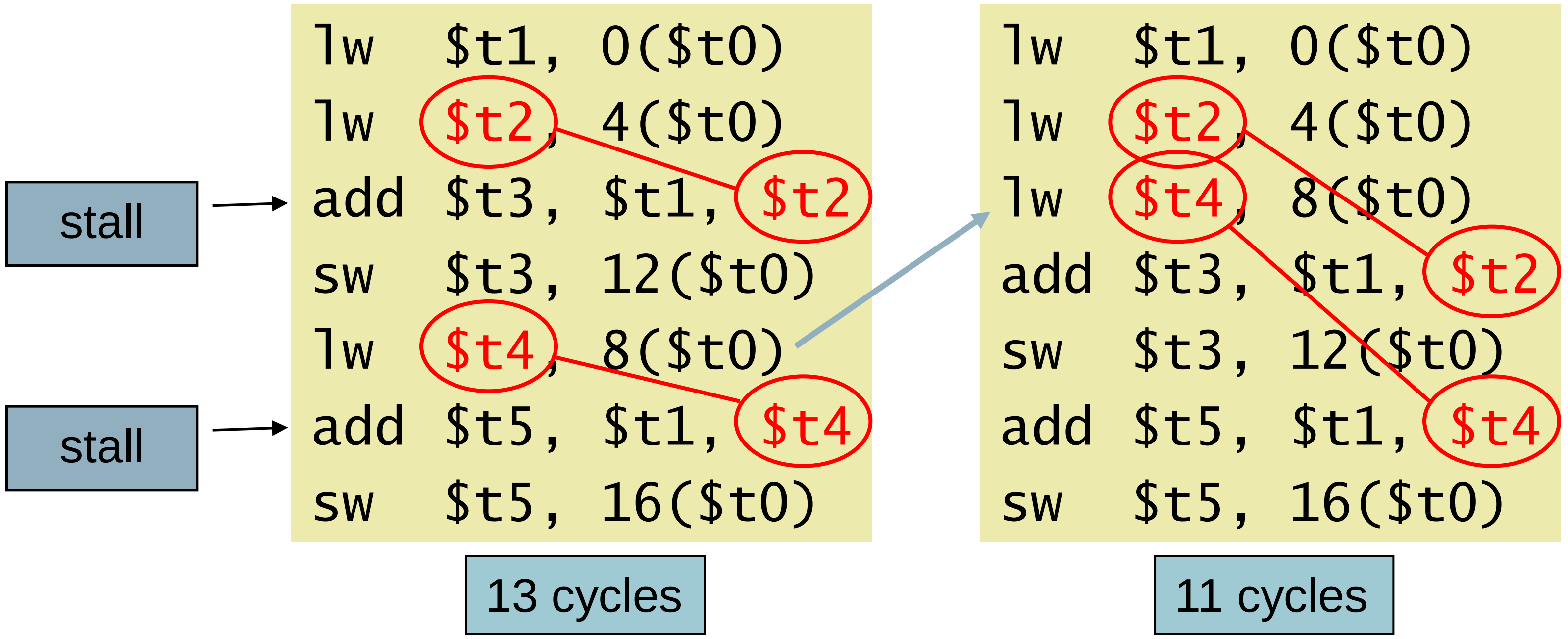
- Let compiler guarantee no hazards
  - Where do we insert the “nops” ?

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

- Problem: this really slows us down! Layer violation!
  - Use forwarding!

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;`



```
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)
```

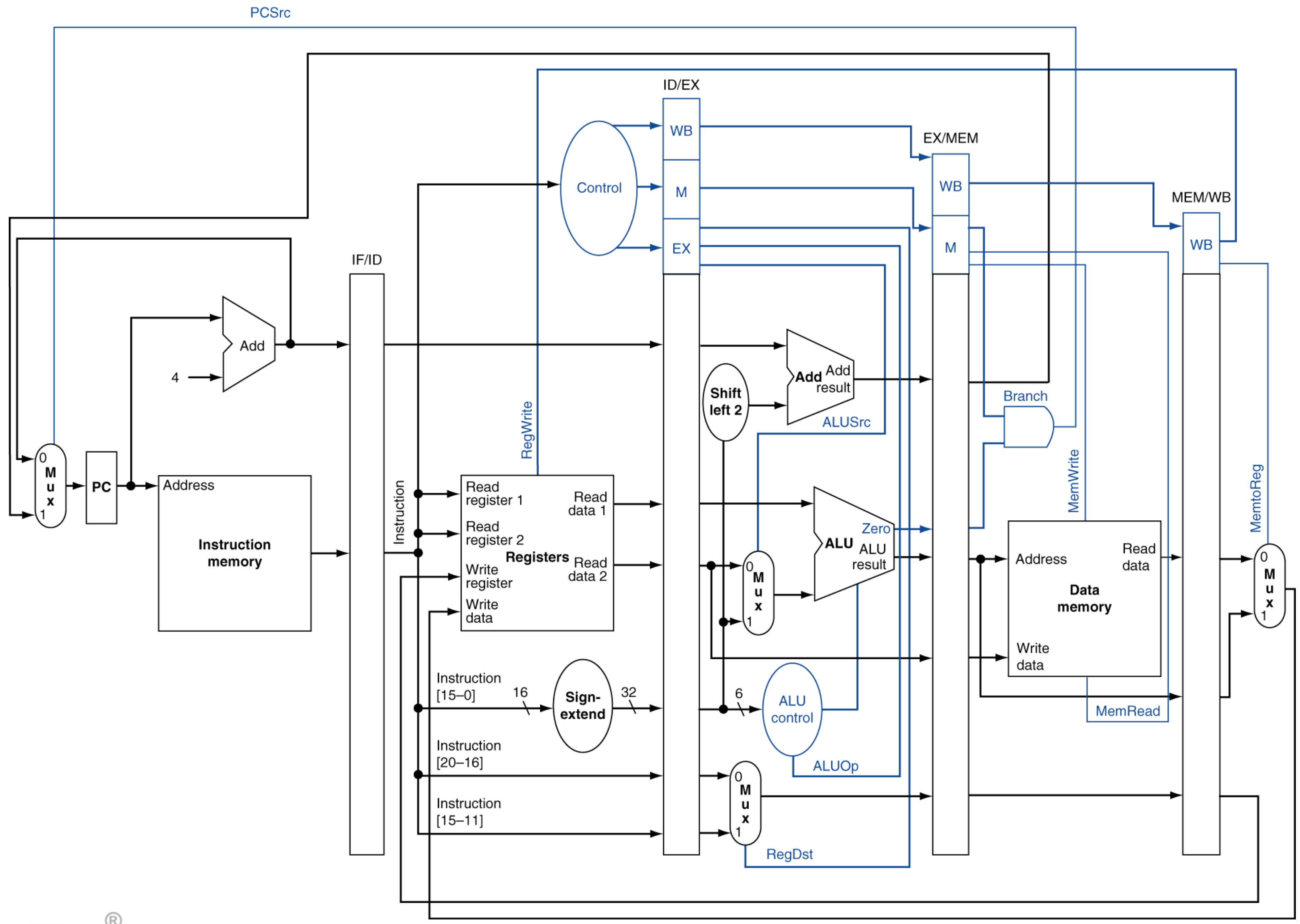
```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

Software Solution

- Compiler can arrange code to avoid load-use stalls
  - Requires knowledge of the pipeline structure
  - Layer violation and dependency
- Hardware can also do it without difficulty
  - Runtime (dynamic) out-of-order execution
    - No compiler dependency

Branch Hazards

Pipelined Datapath and Control

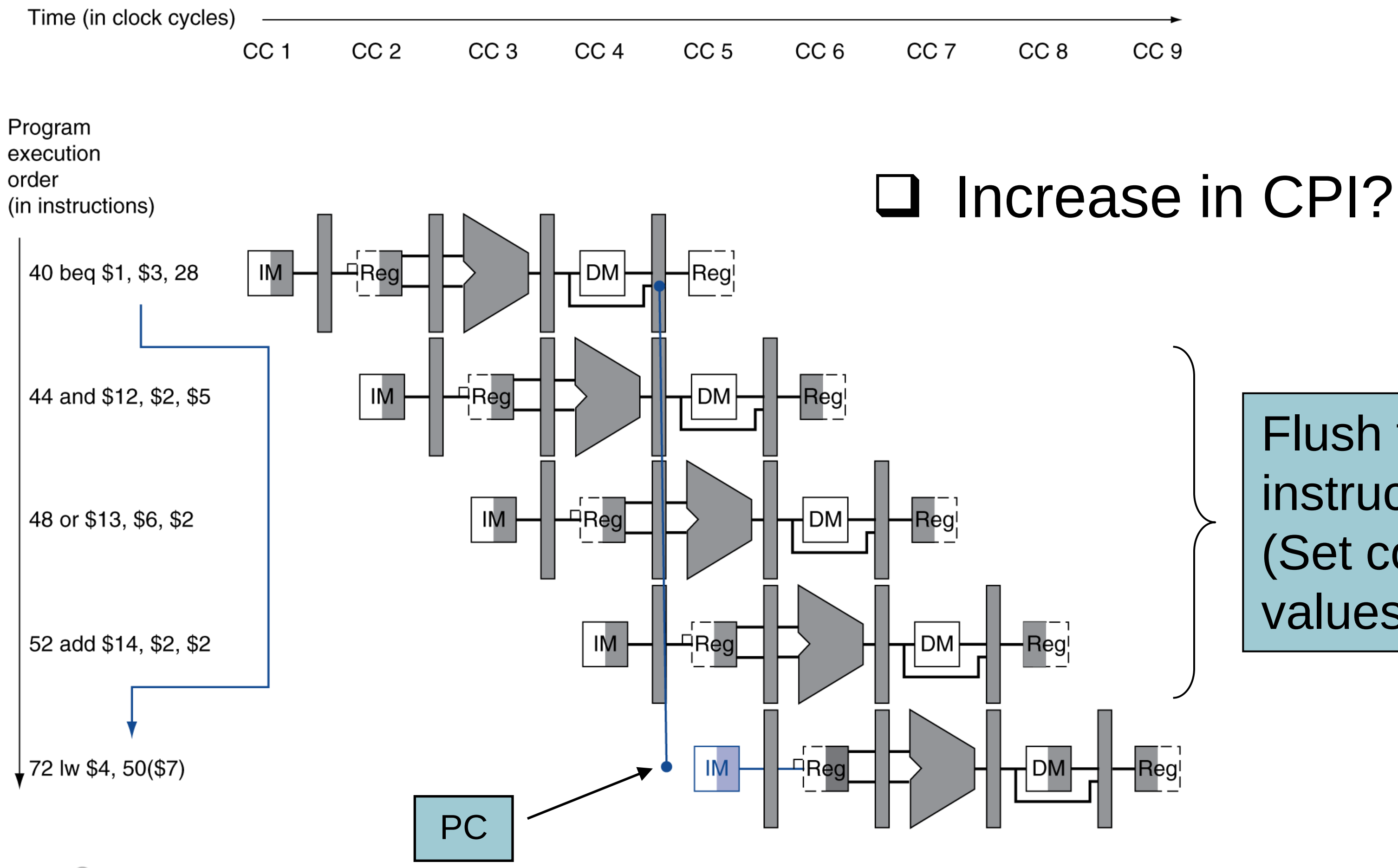


Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Branch Hazards

- If branch outcome determined in MEM



- ❑ Increase in CPI?

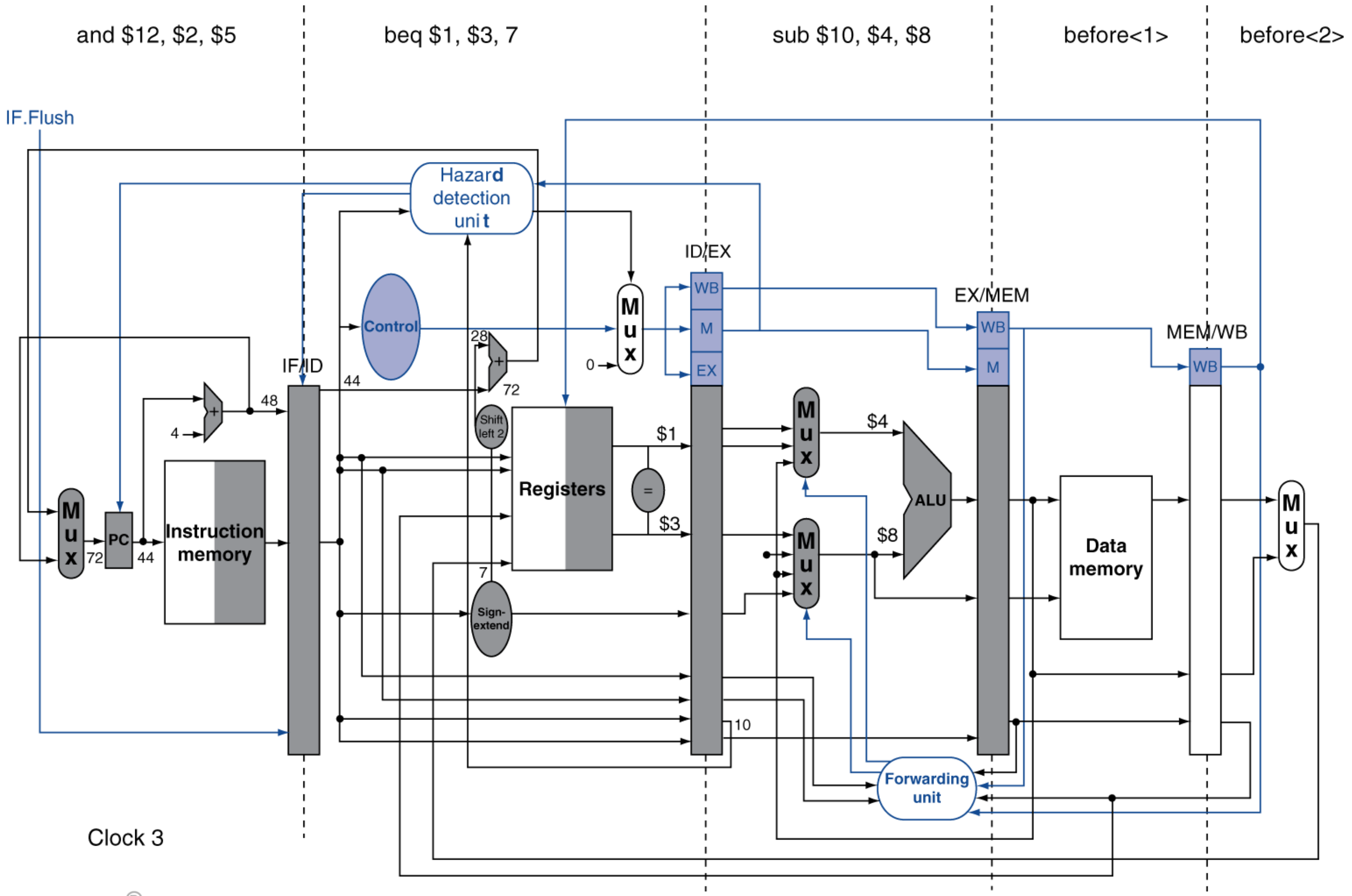
Flush these instructions (Set control values to 0)

Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

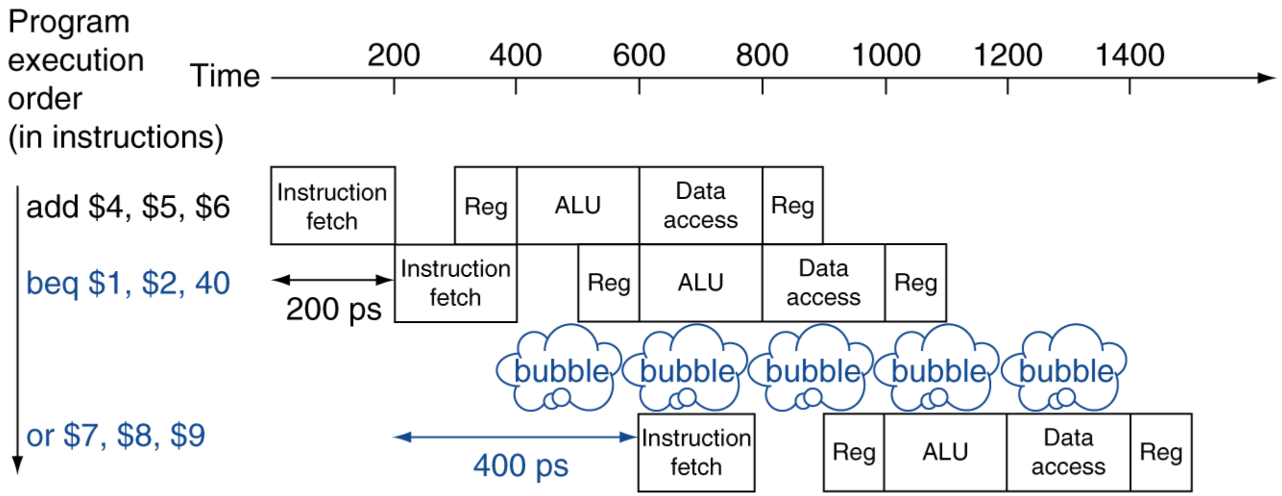
```
36:  sub  $10, $4, $8
40:  beq  $1,  $3, 7                // see next slide
44:  and  $12, $2, $5
48:  or   $13, $2, $6
52:  add  $14, $4, $2
56:  slt  $15, $6, $7
          ...
72:  lw   $4, 50($7)
```

Moving Branch to ID Stage



Stall on Branch

- Wait until branch outcome determined before fetching next instruction



- Increase in CPI?

Moving Branch to ID Stage

- Compute branch target address in ID
  - MIPS branch rely on simple test (equal, sign)
- Evaluate branch decision in ID
  - Additional forwarding
    - From ALU/MEM, MEM/WB latches
  - Additional stall
  - IF.Flush control signal

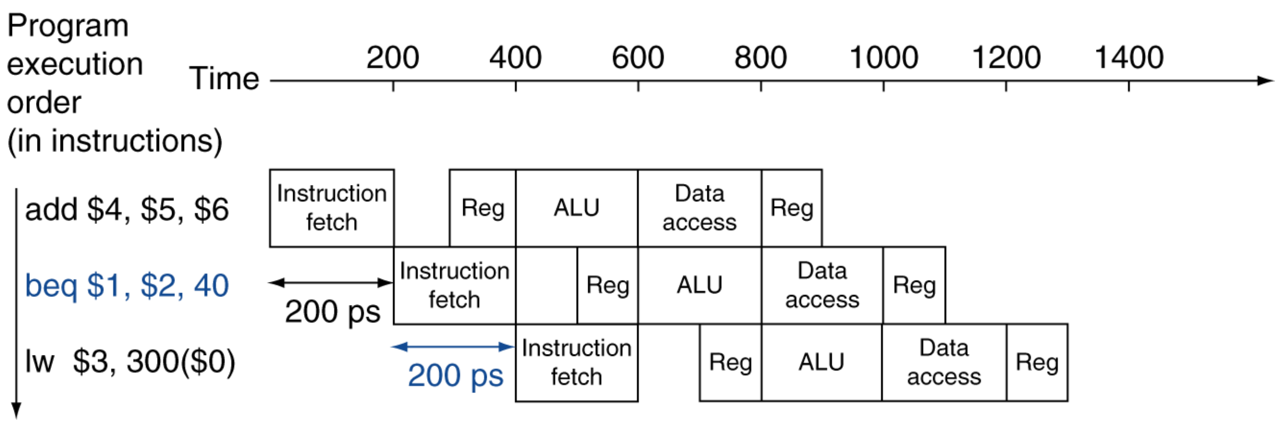
- Think about ARM conditional instruction (skip)

Branch Prediction

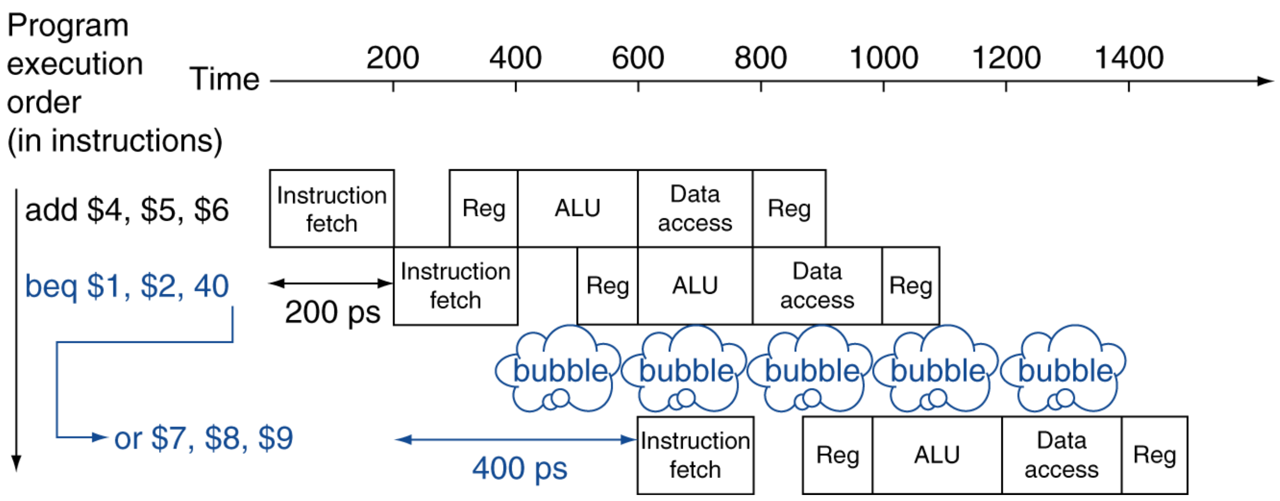
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with (Static) Predict Not Taken

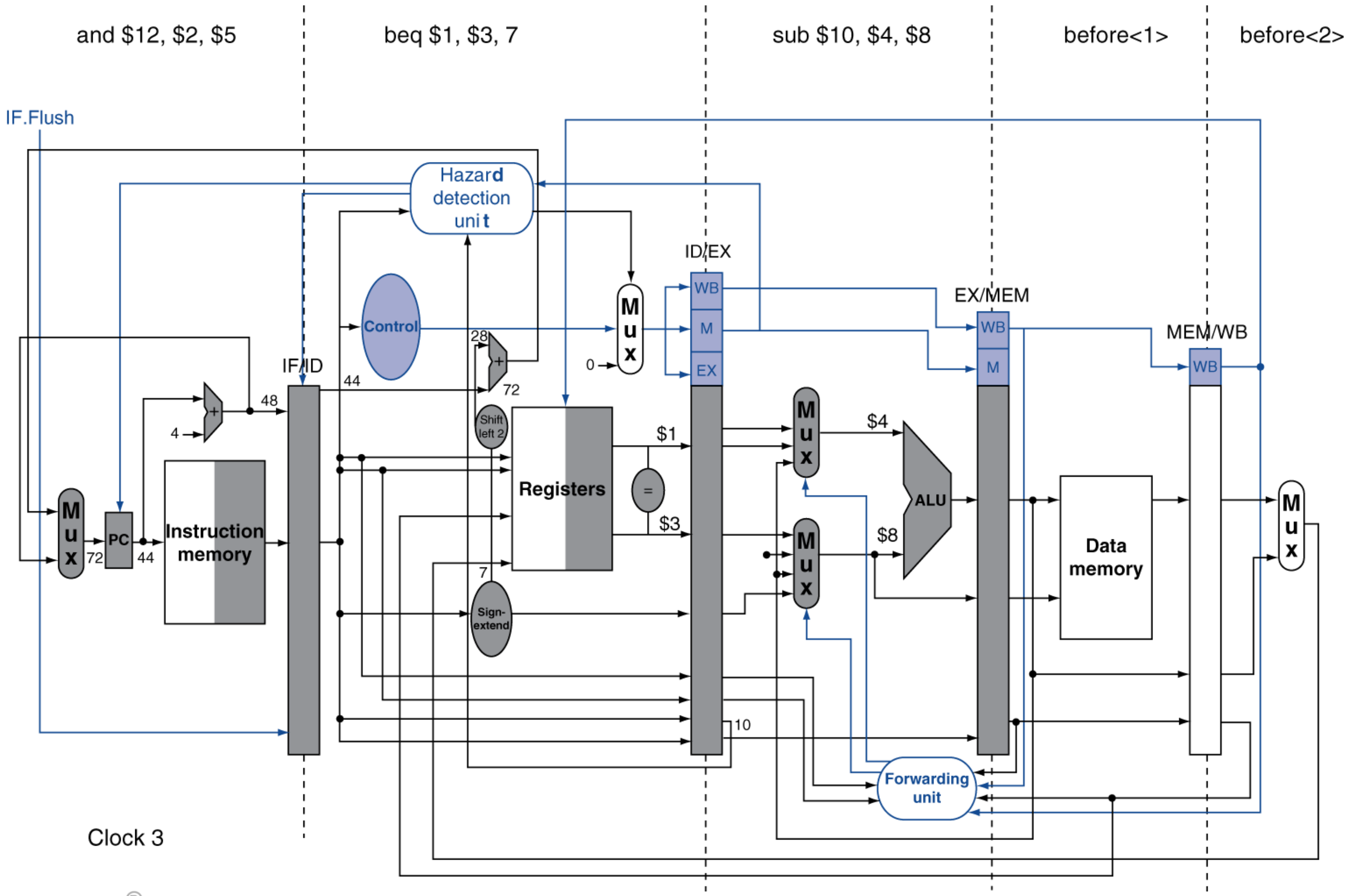
Prediction correct



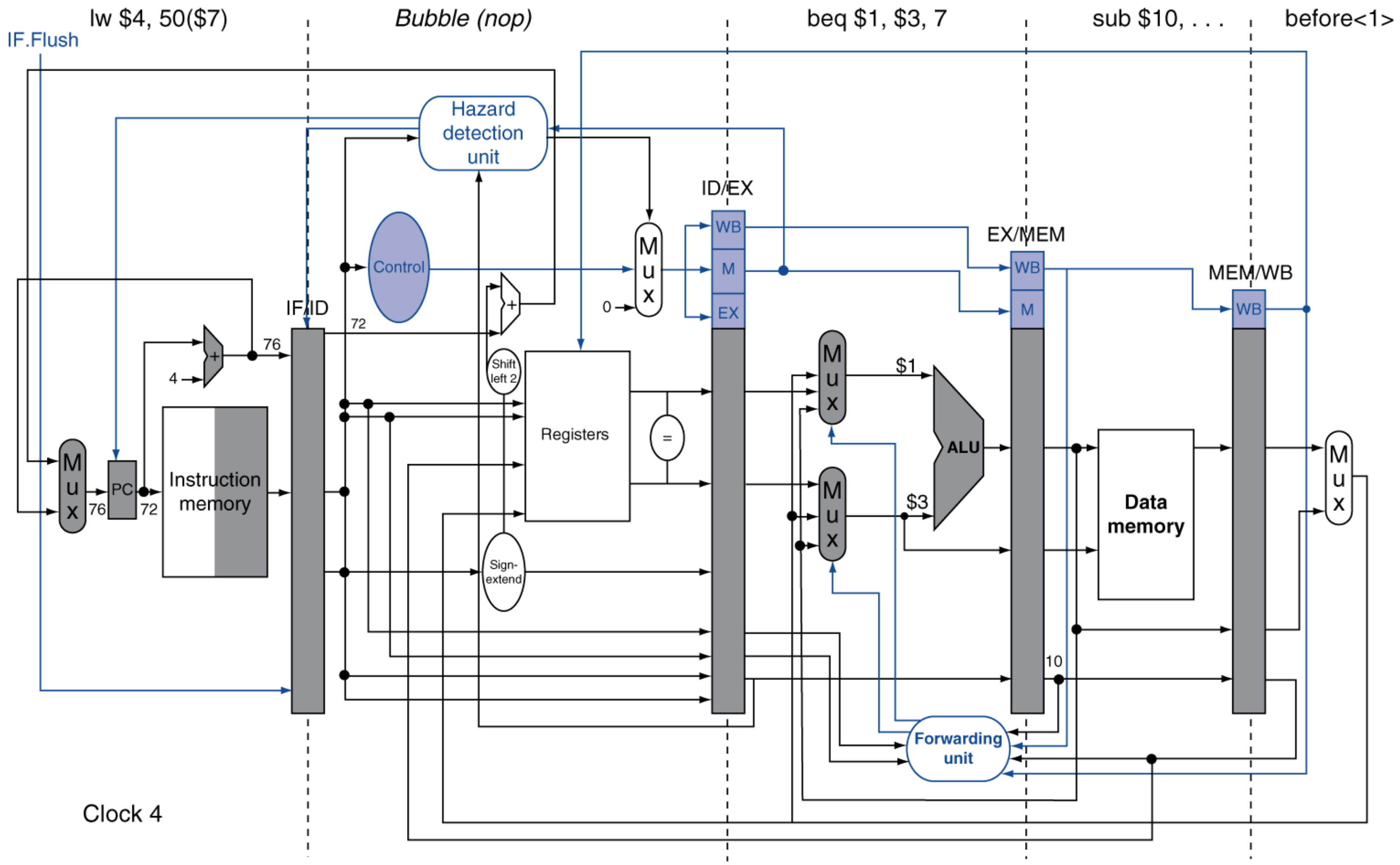
Prediction incorrect



Example: Branch Taken



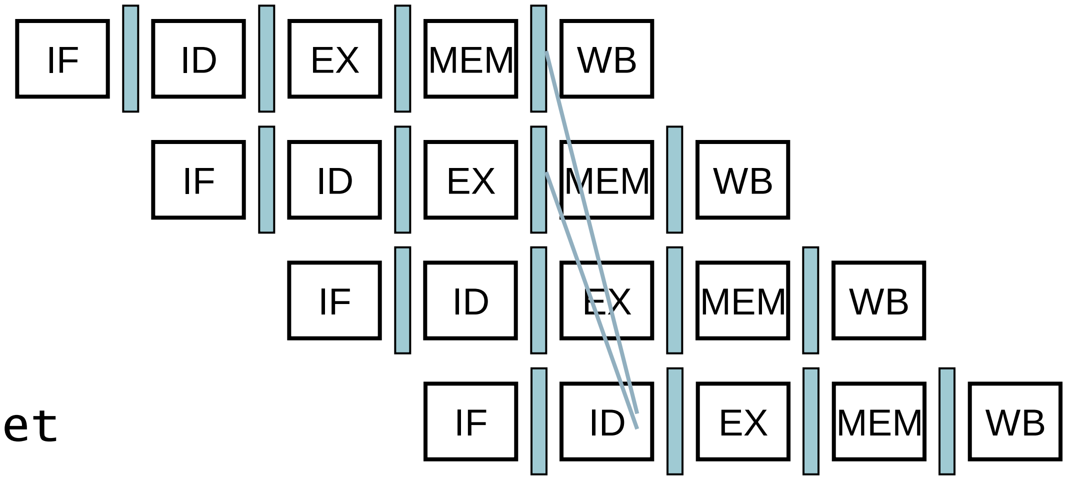
Example: Branch Taken



Data Hazards for Branches

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

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

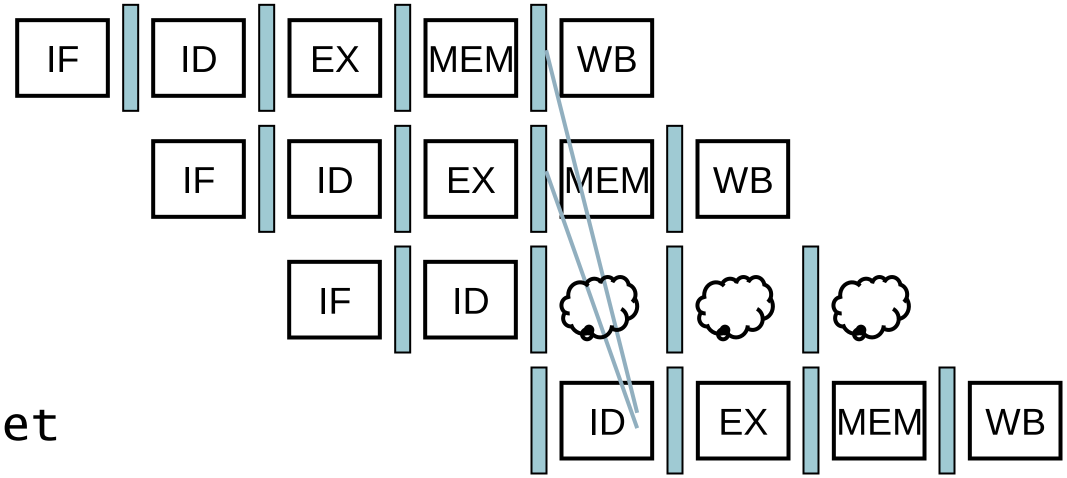


- Can resolve using forwarding

Data Hazards for Branches

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

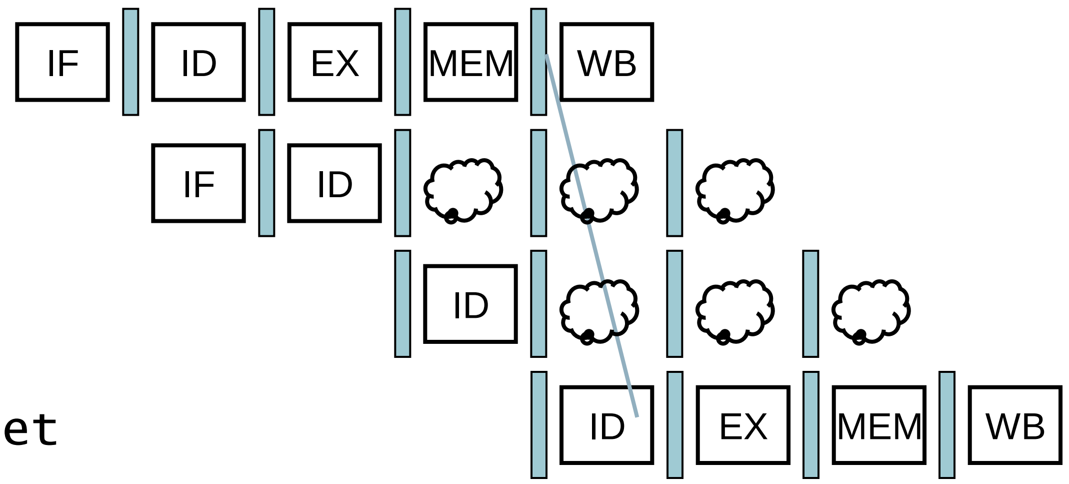
```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



Data Hazards for Branches

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

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



More–Realistic Branch prediction

- The remaining part is optional
- Branch prediction is important in deep pipeline

More-Realistic Branch Prediction

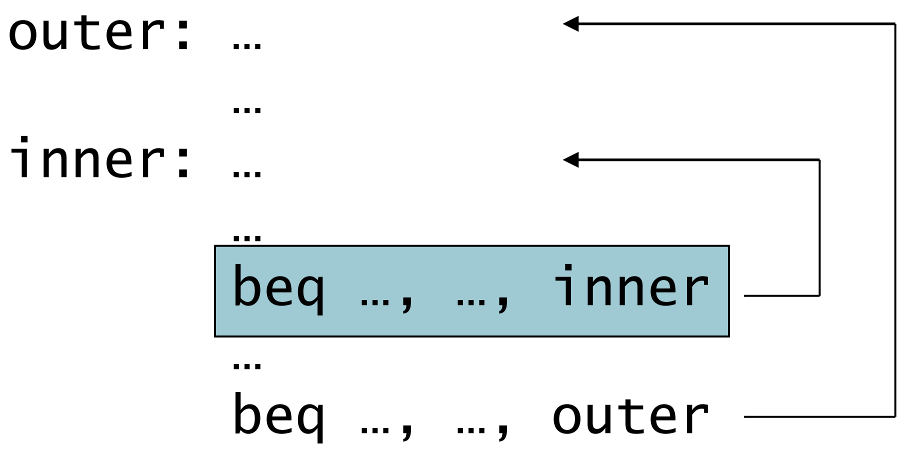
- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic 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, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!

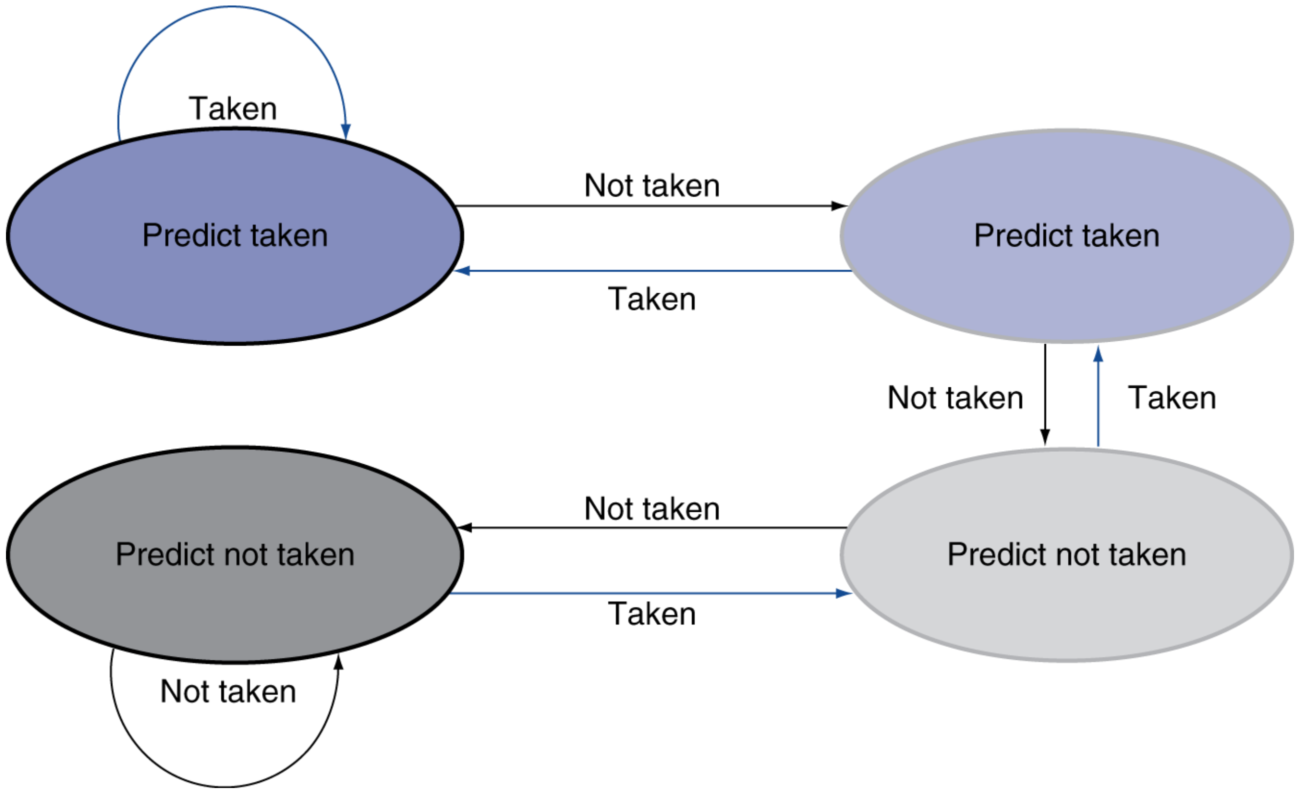


```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```

  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions



Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation