Class Topics (클래스 홈페이지 참조)

- Part 1: Fundamental concepts and principles
- Part 2: 빠른 컴퓨터를 위한 ISA design
  - Topic 1 Computer performance and ISA design (Ch. 1)
  - Topic 2 RISC (MIPS) instruction set (Chapter 2)
  - Topic 3 Computer arithmetic and ALU (Chapter 3)
- Part 3: ISA 의 효율적인 구현 (pipelining, cache memory)
  - Topic 4 Processor design (Chapter 4)
    - Single cycle implementation
    - Pipelined implementation
  - Topic 5 Memory system design (Chapter 5)

Chapter 4

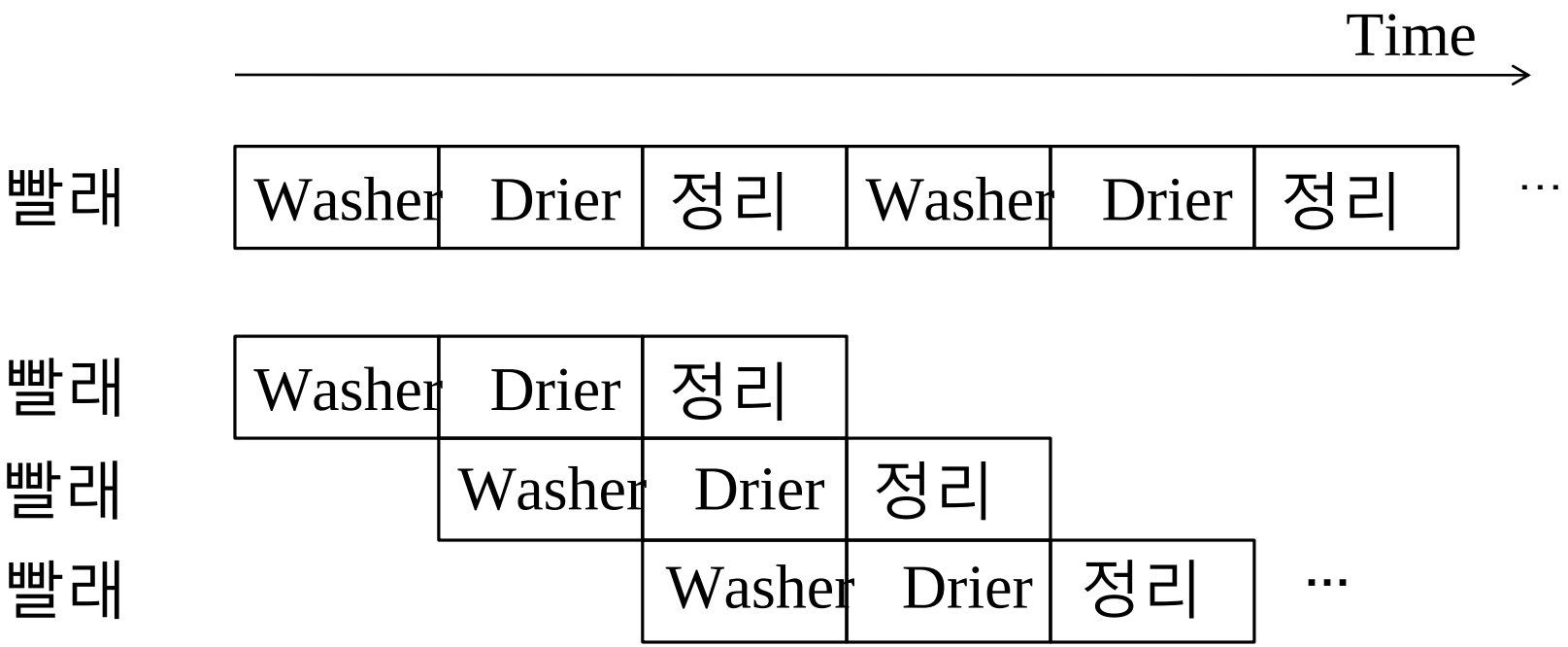
The Processor (Implementing ISA)

Part 2: Pipelining

Some of authors' slides are modified

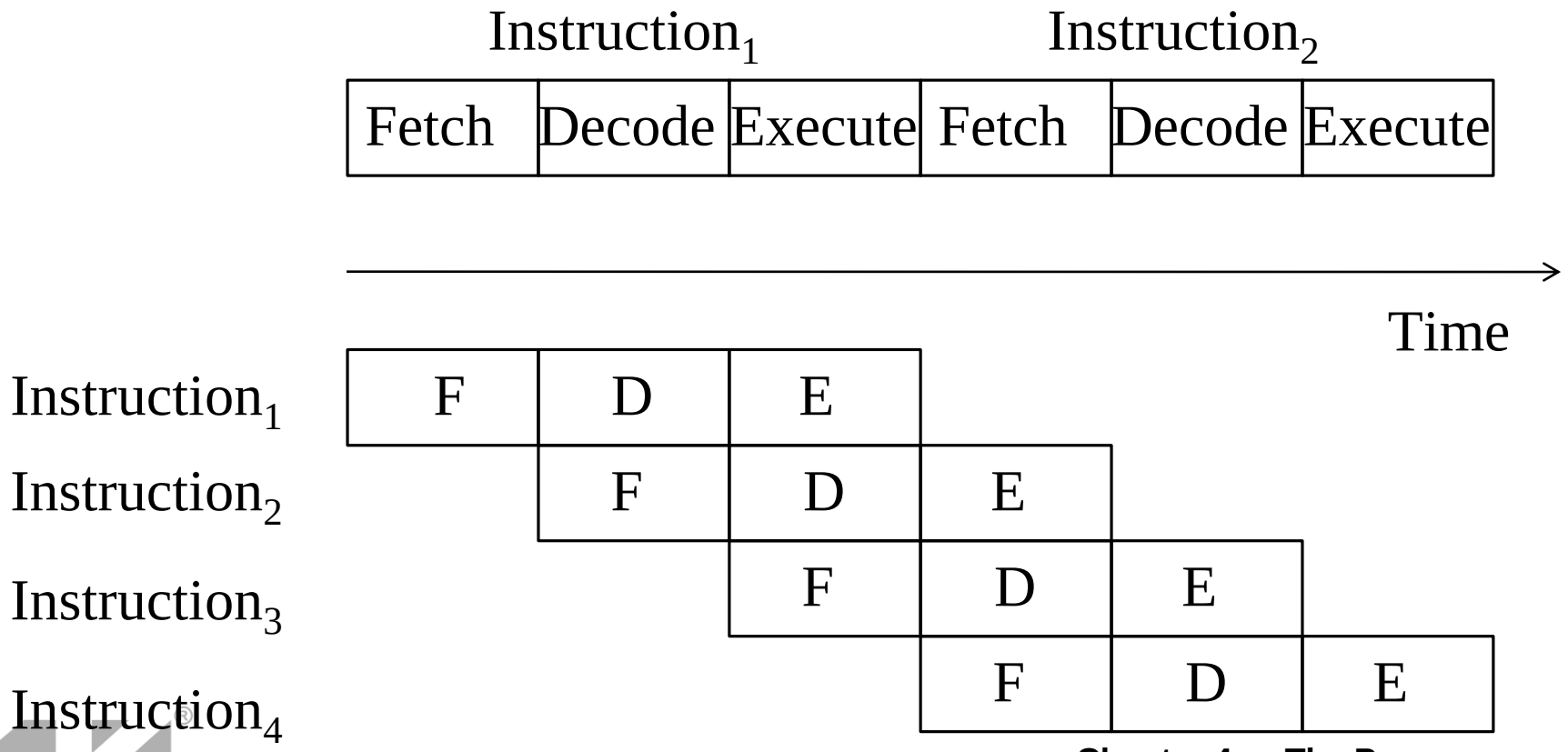
Pipelined Laundry (복습)

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



Pipelined instruction Execution(복습)

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

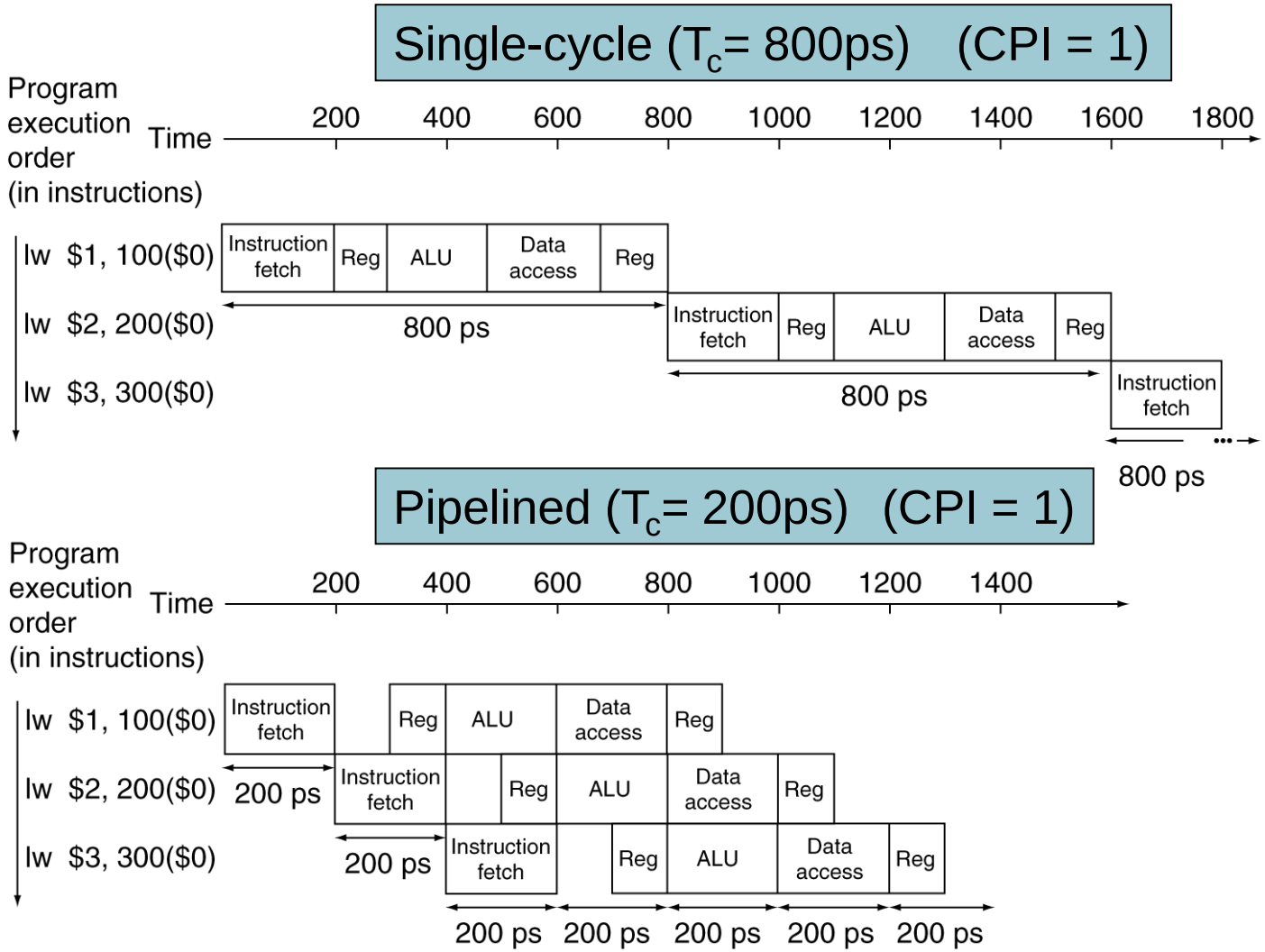


MIPS Pipeline (반복)

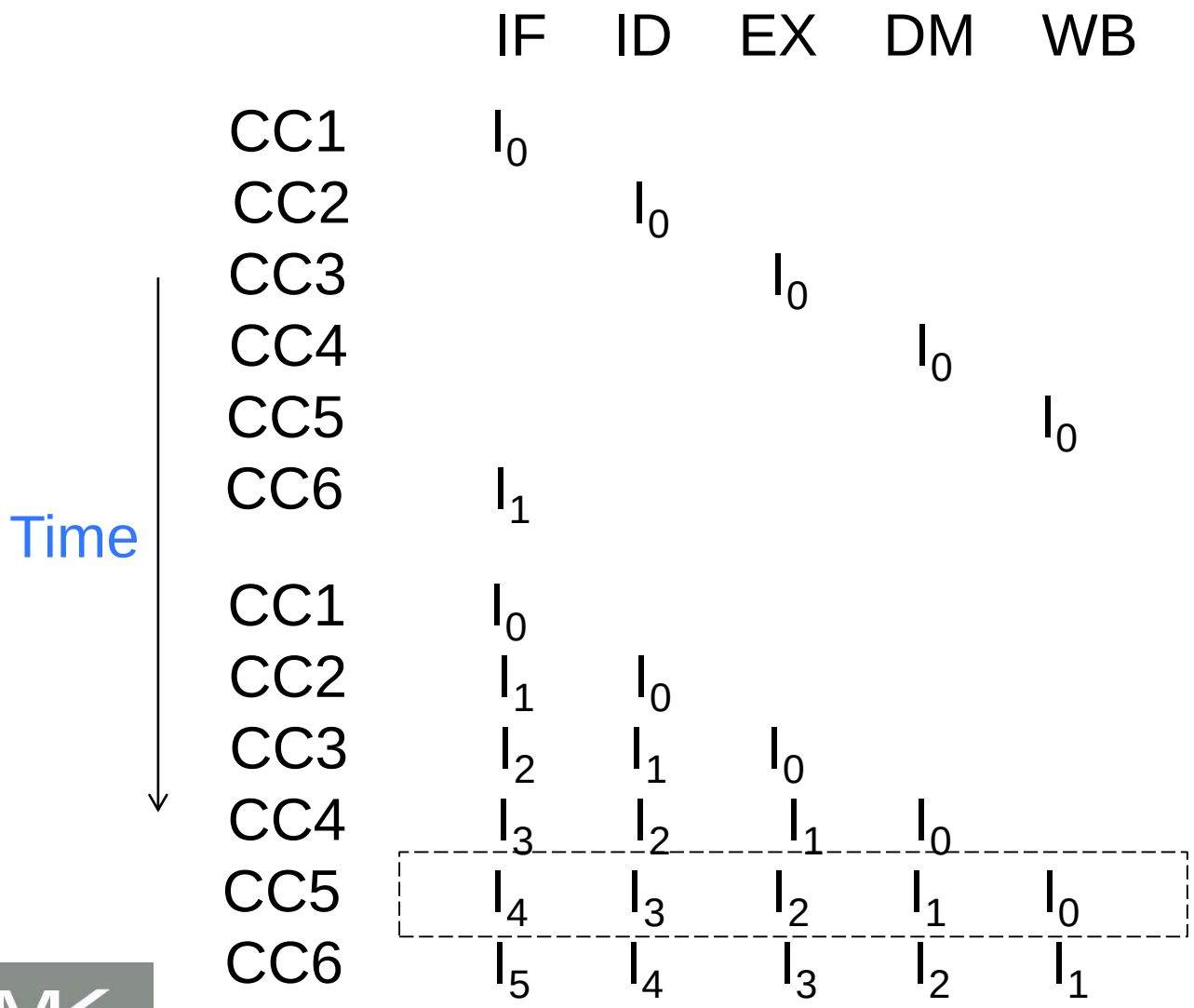
- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

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

Pipeline Performance



Multicycle vs. Pipelining (반복)



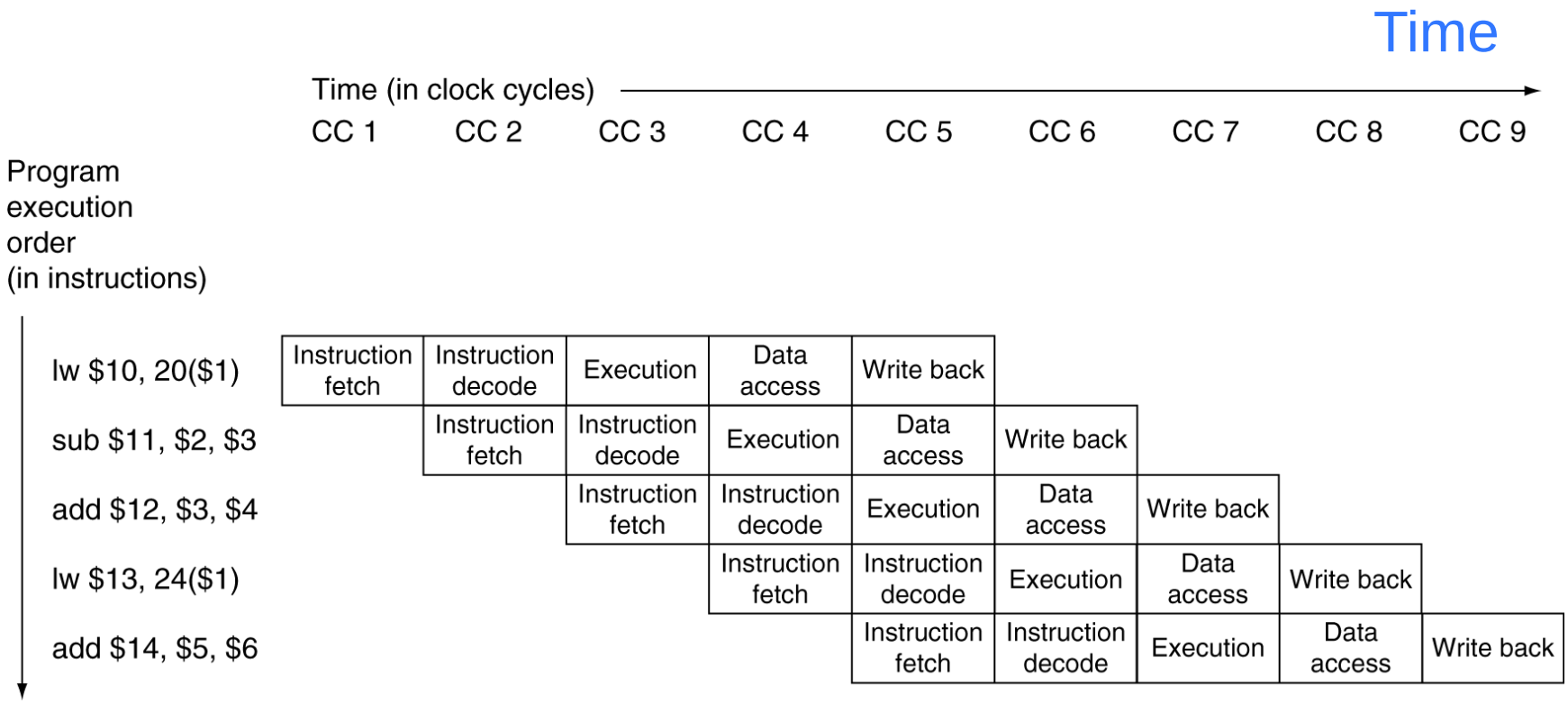
(cct = 200ps)

Multicycle
(CPI = 5)

Pipeline
5-stage
(CPI = 1)

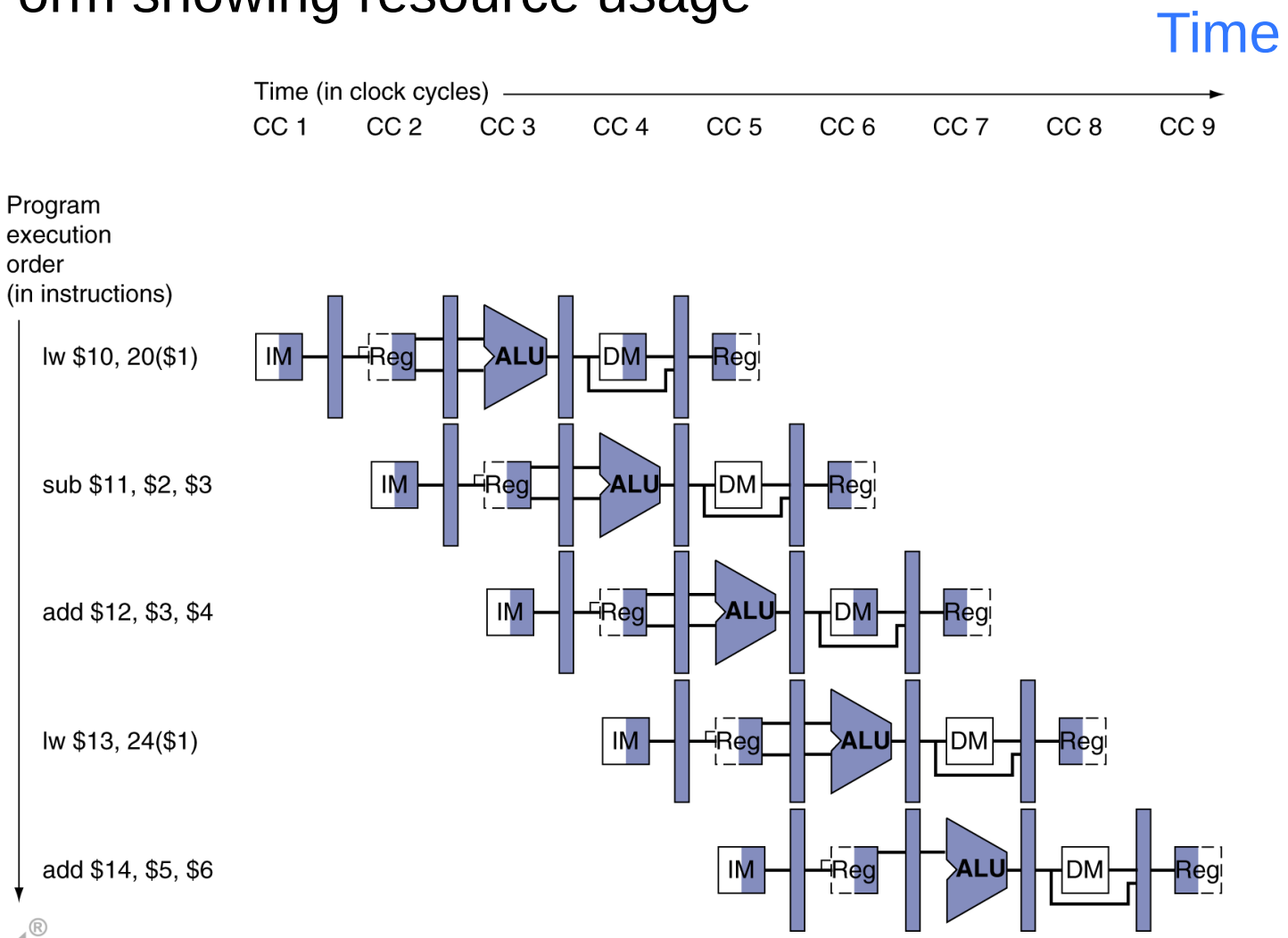
Multi-Cycle Pipeline Diagram

- Traditional form



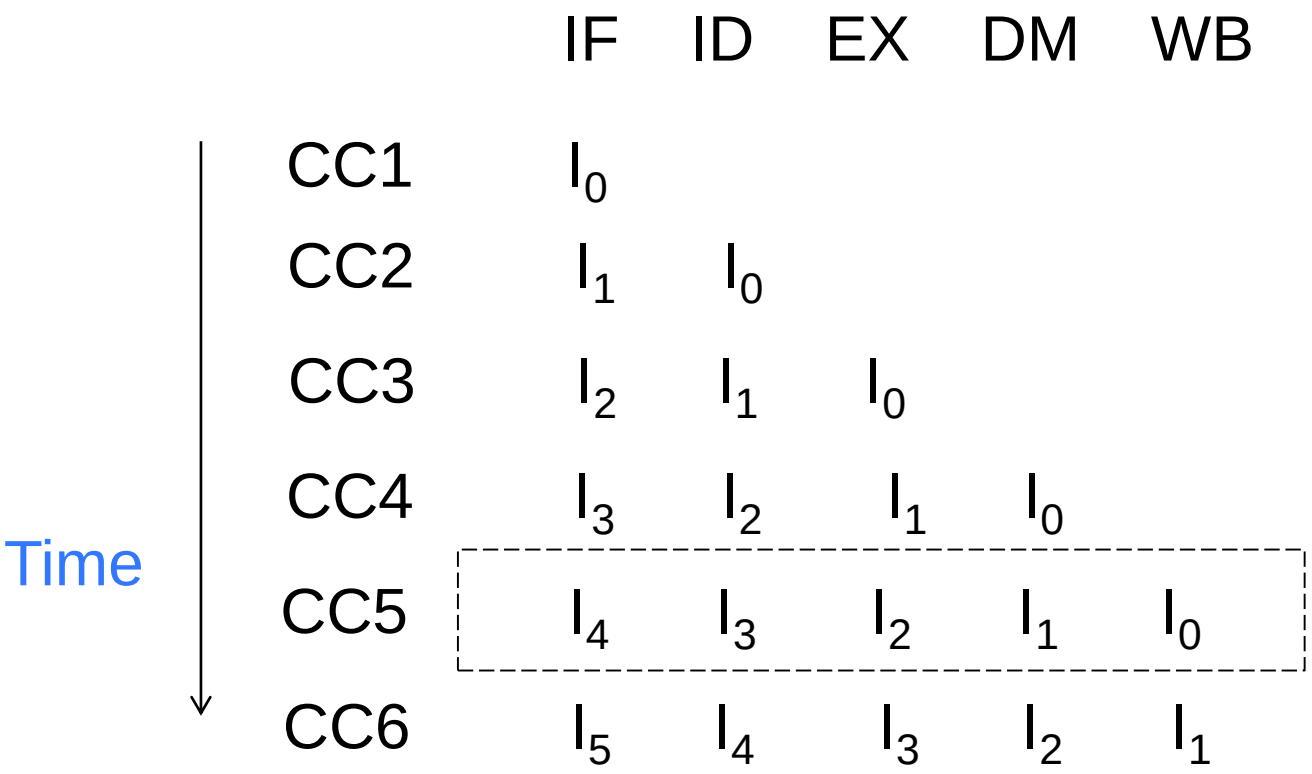
Pipeline Diagram

- Form showing resource usage



Pipeline Diagram

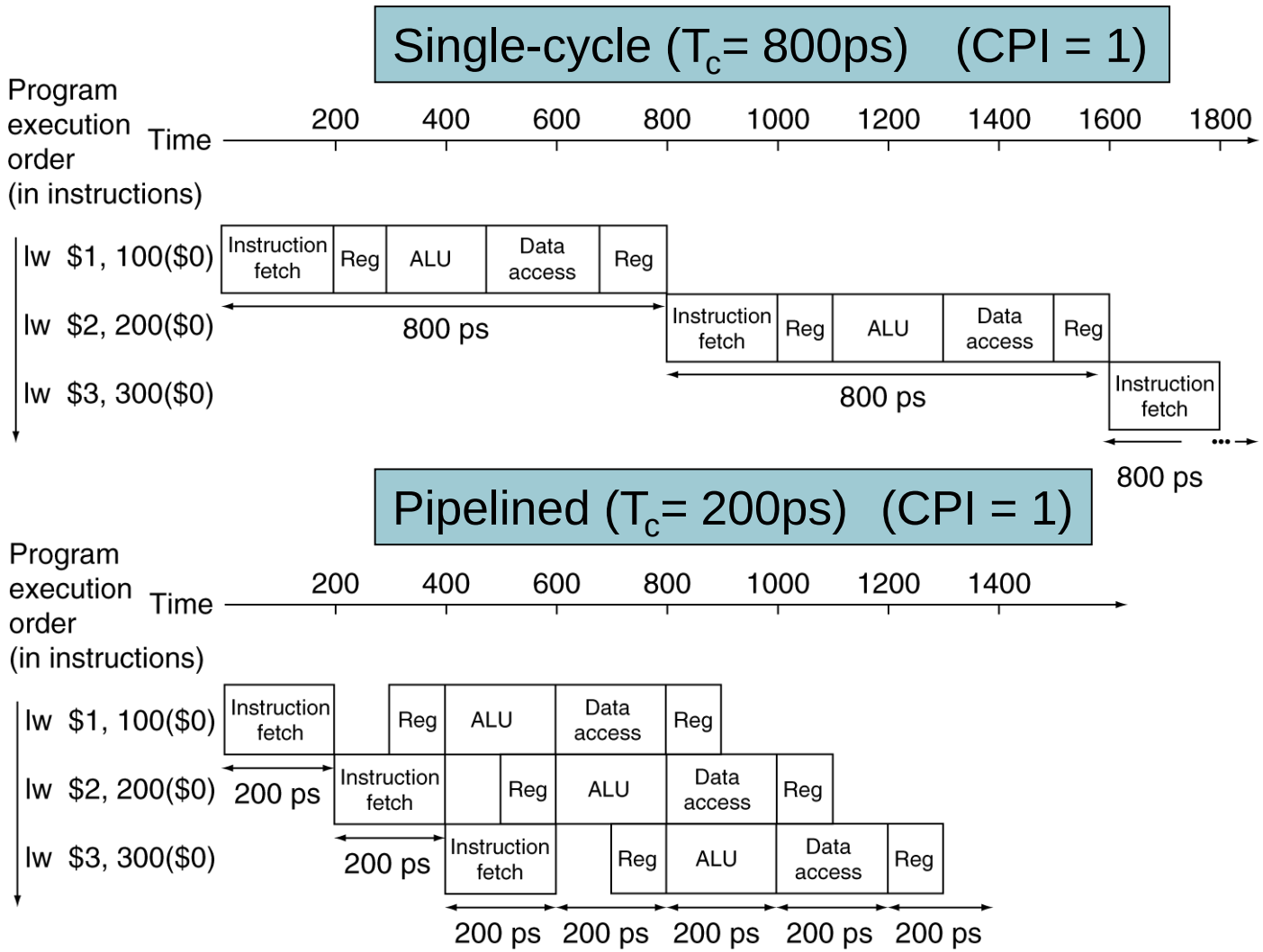
- Each stage run different instructions
  - Only one set of hardware



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Ideal speedup = number of stages
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipeline Performance (반복)



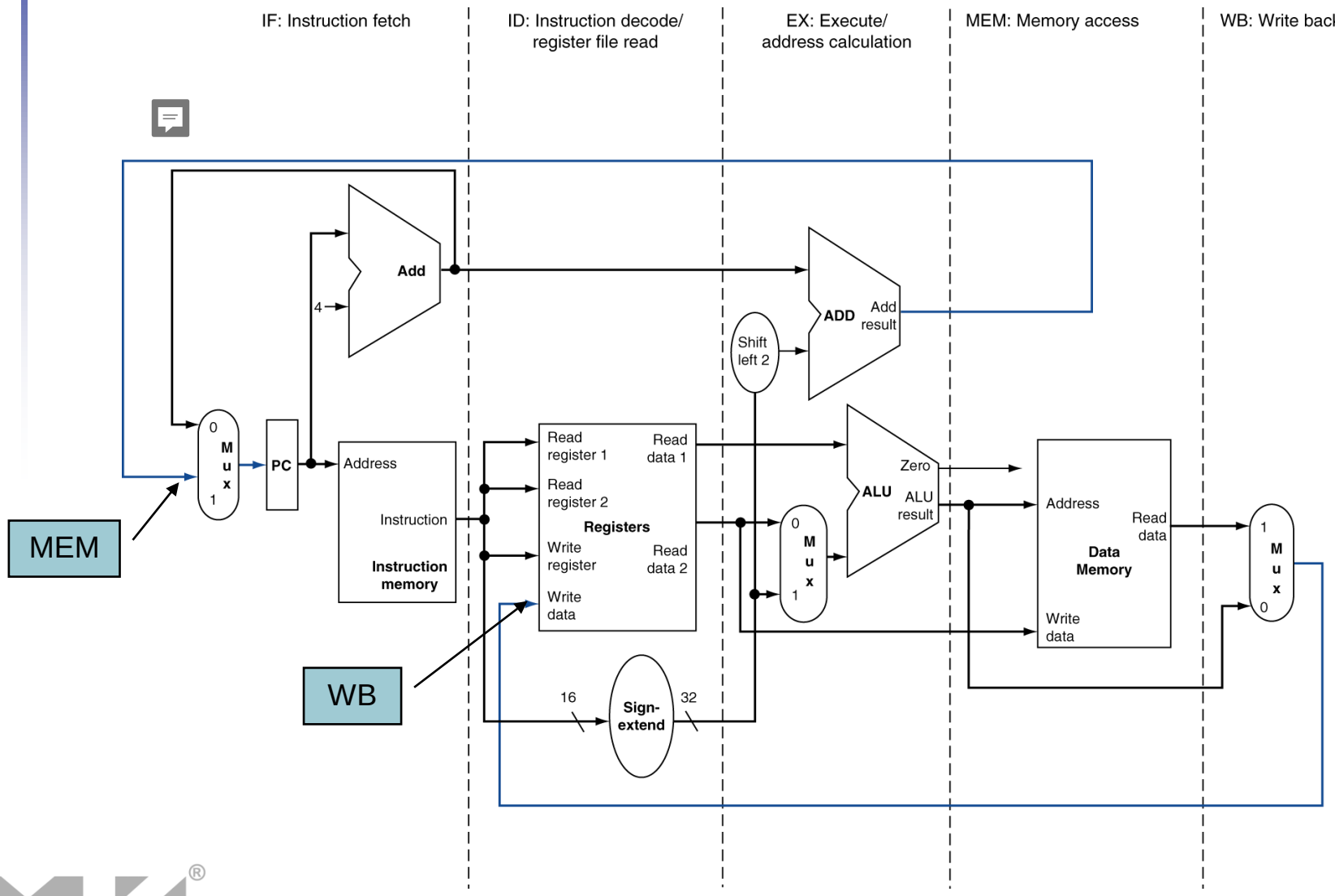
Pipelining (미리보기)

- 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
  - Data hazards: instruction depends on previous one
  - Control hazards: need to worry about branch
- We'll build a simple pipeline and look at these issues

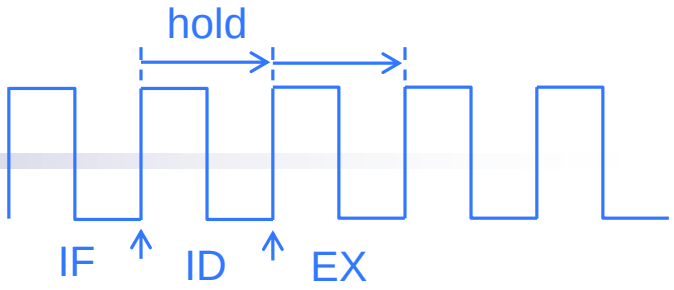
Pipelined Datapath and Control

(Implementation for Pipelining)

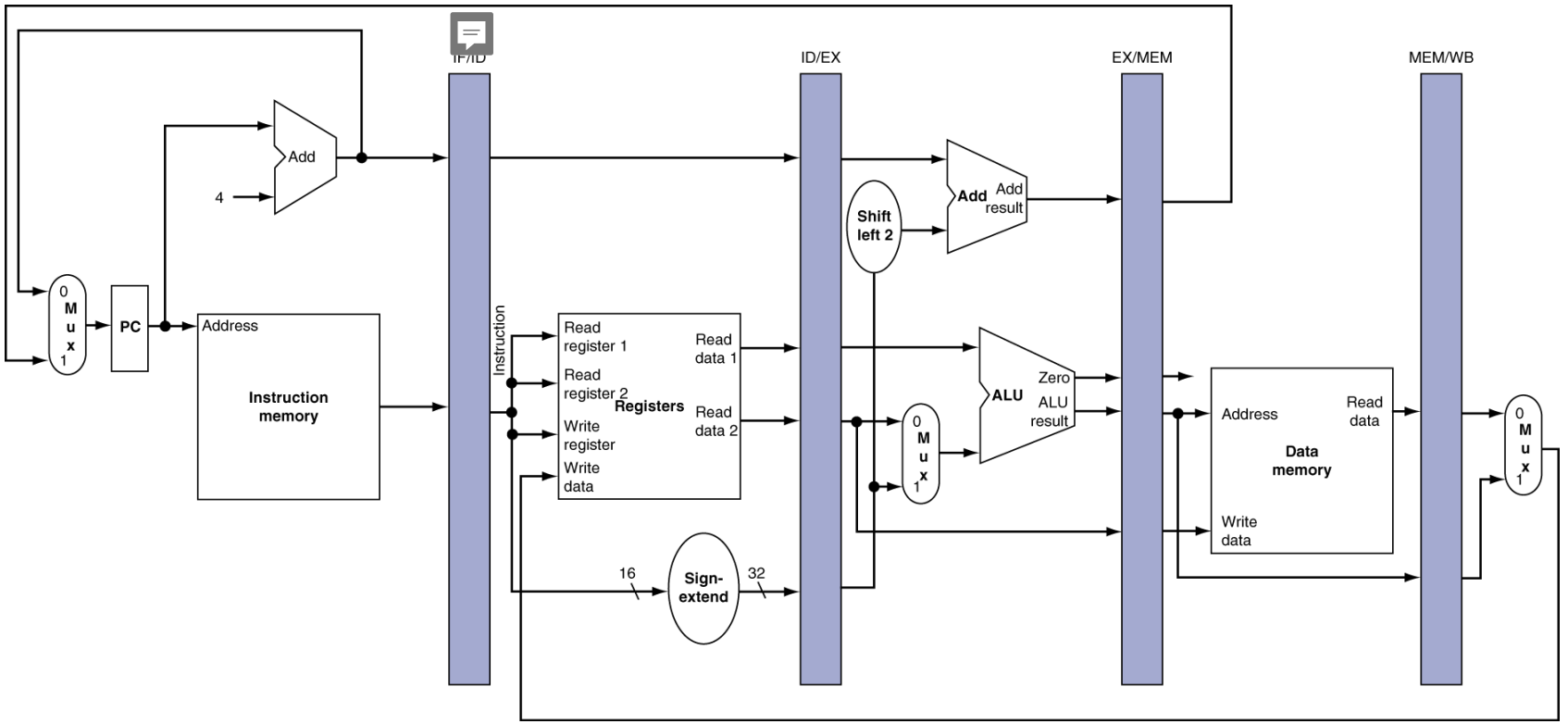
MIPS Pipelined Datapath



Pipeline registers



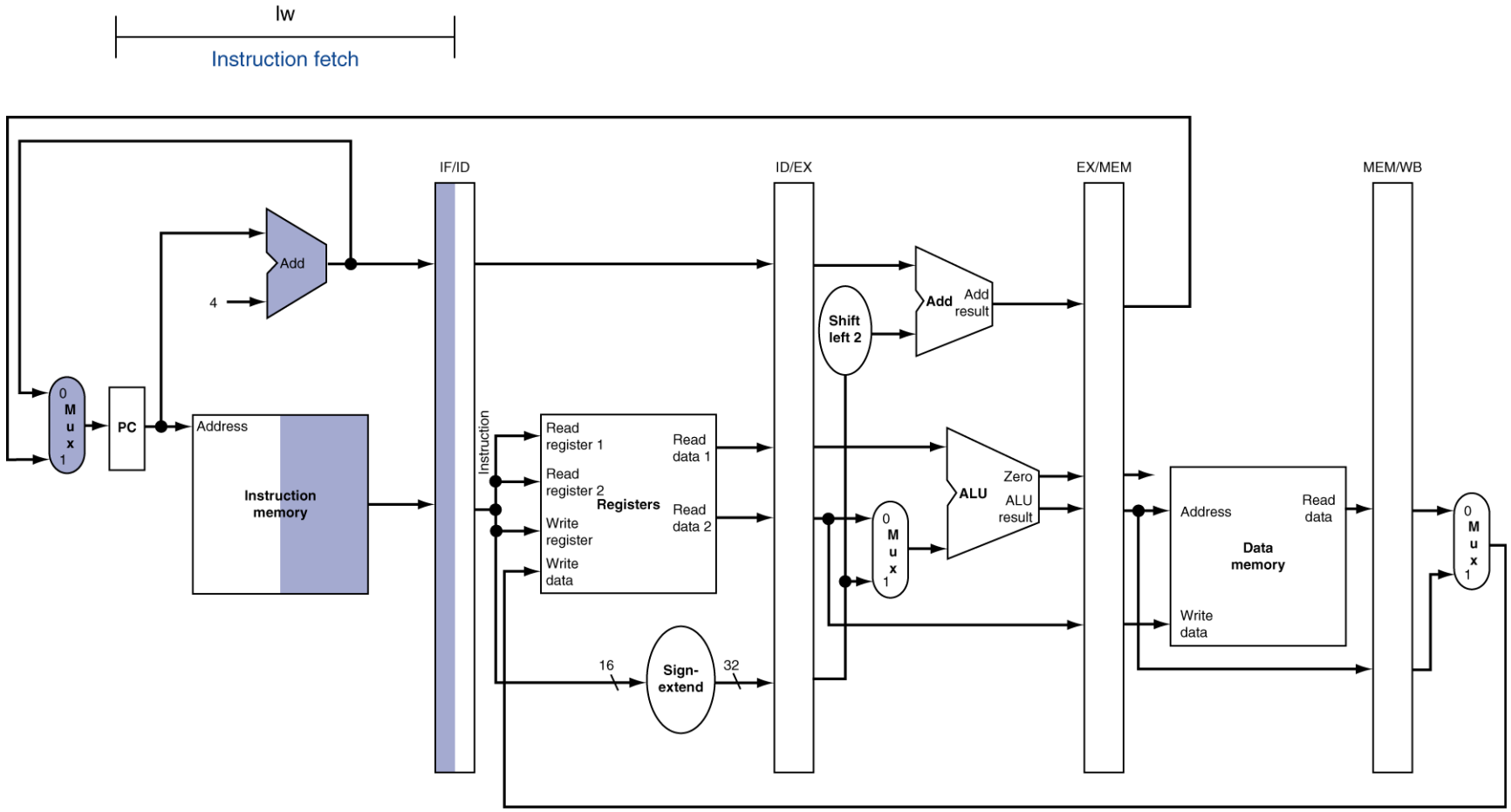
- Need registers between stages
  - To hold information produced in previous cycle



Pipeline Operation

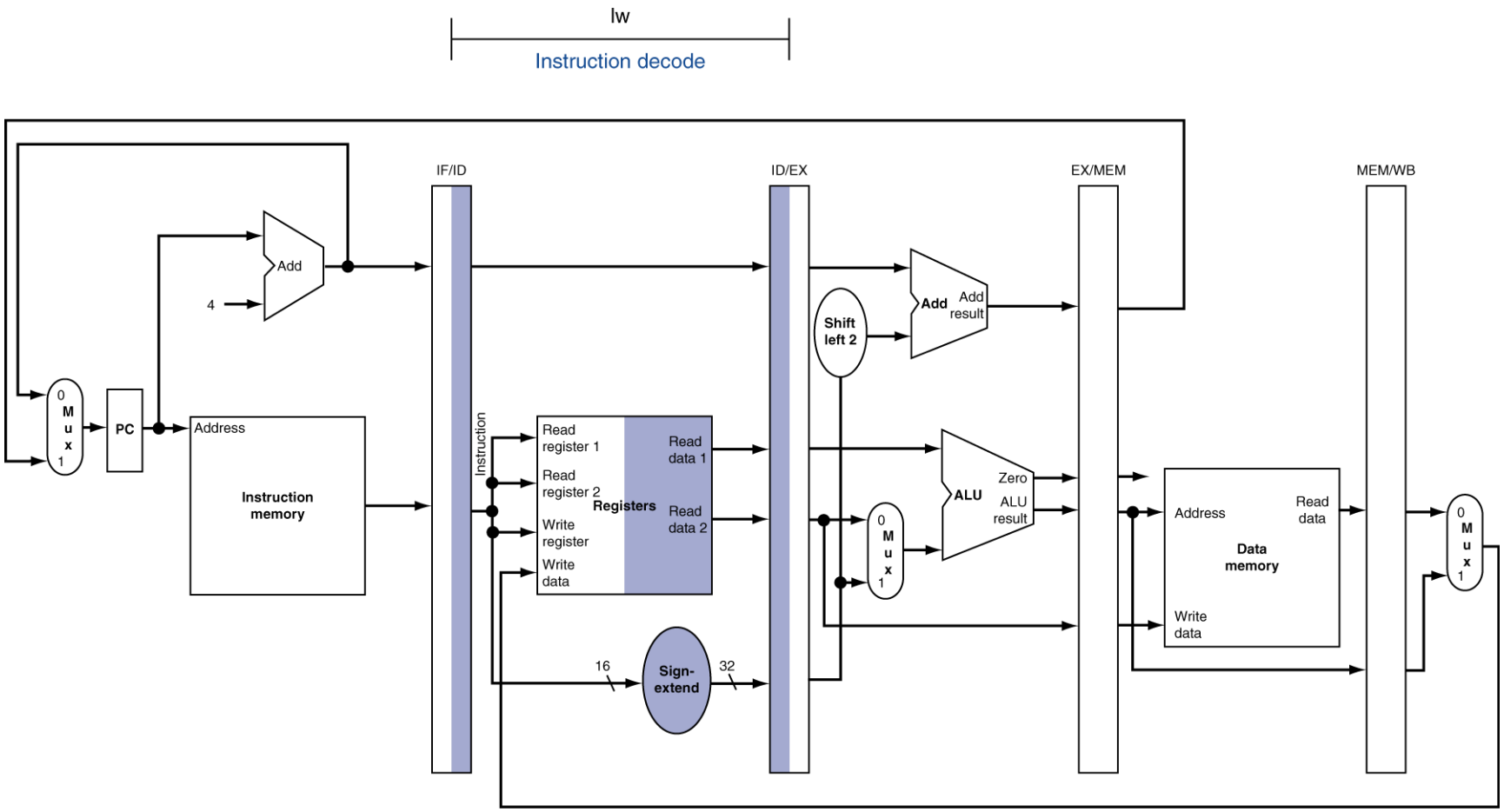
- Cycle-by-cycle flow of instructions through the pipelined datapath
  - We'll look at diagrams for load & store

IF for Load, Store, …



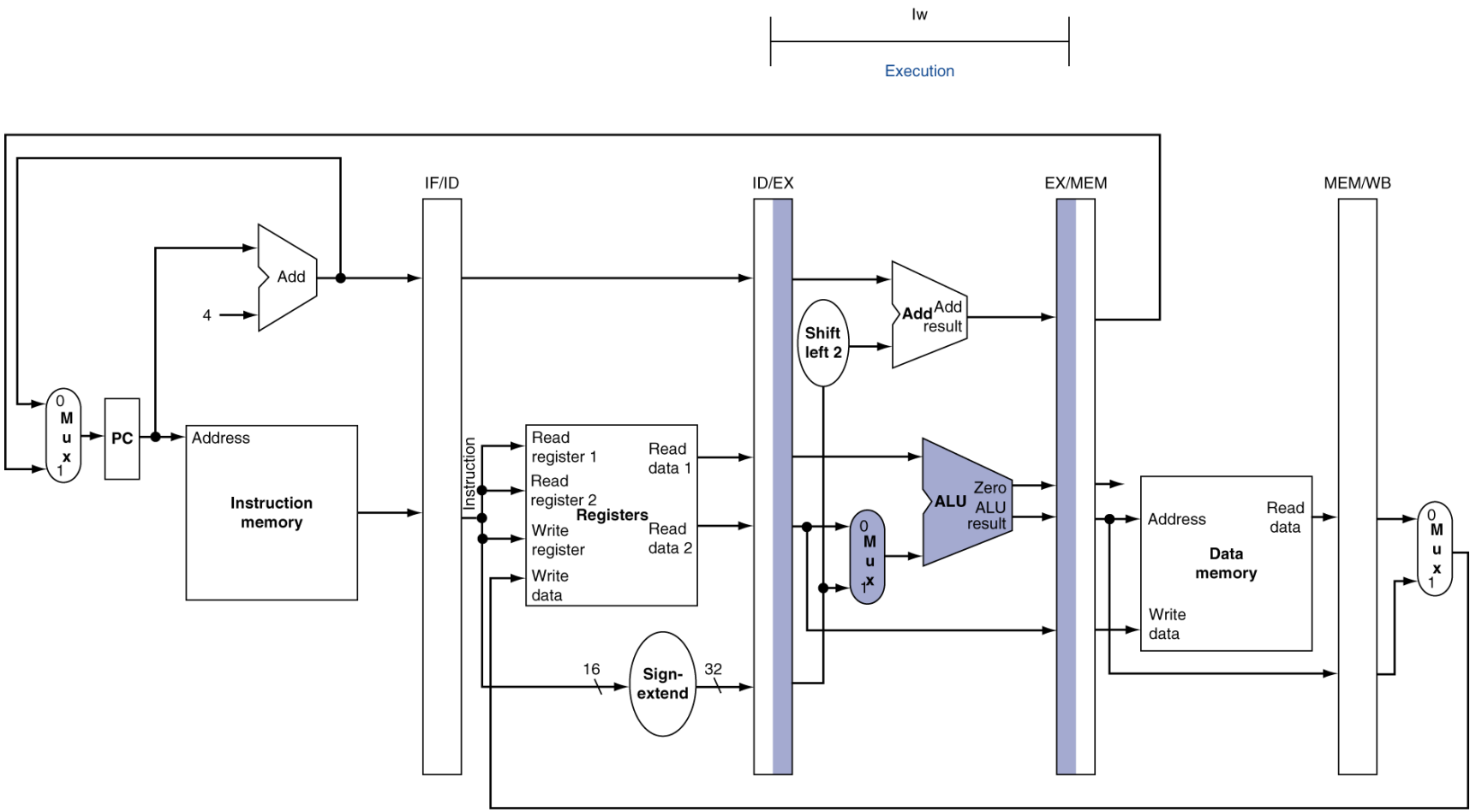
- Information in IF/ID register: instruction, PC+4 (공통)

ID for Load, Store, …



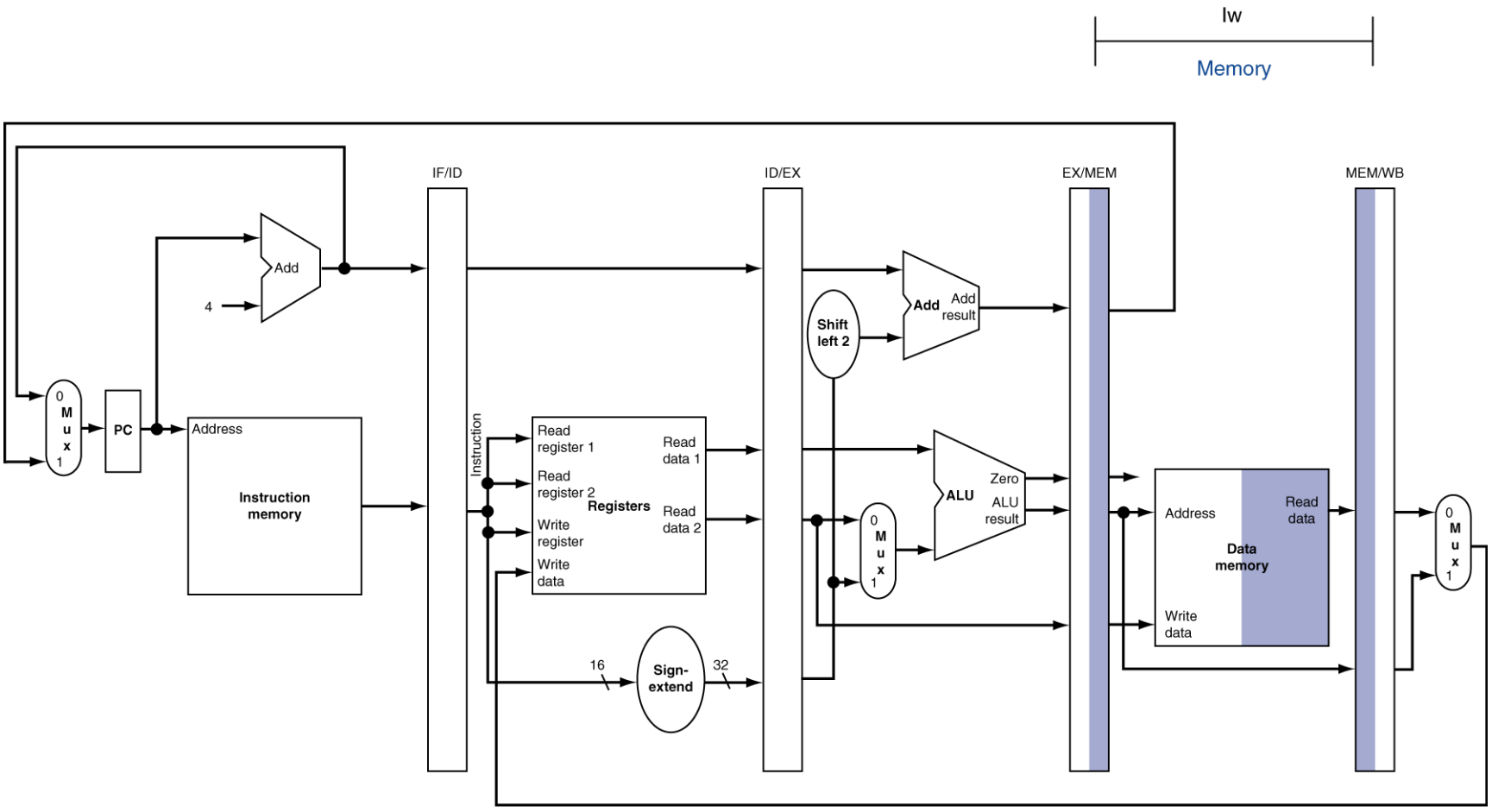
- ID/EX: PC+4, registers, immediate, control signals (공통)

EX for Load



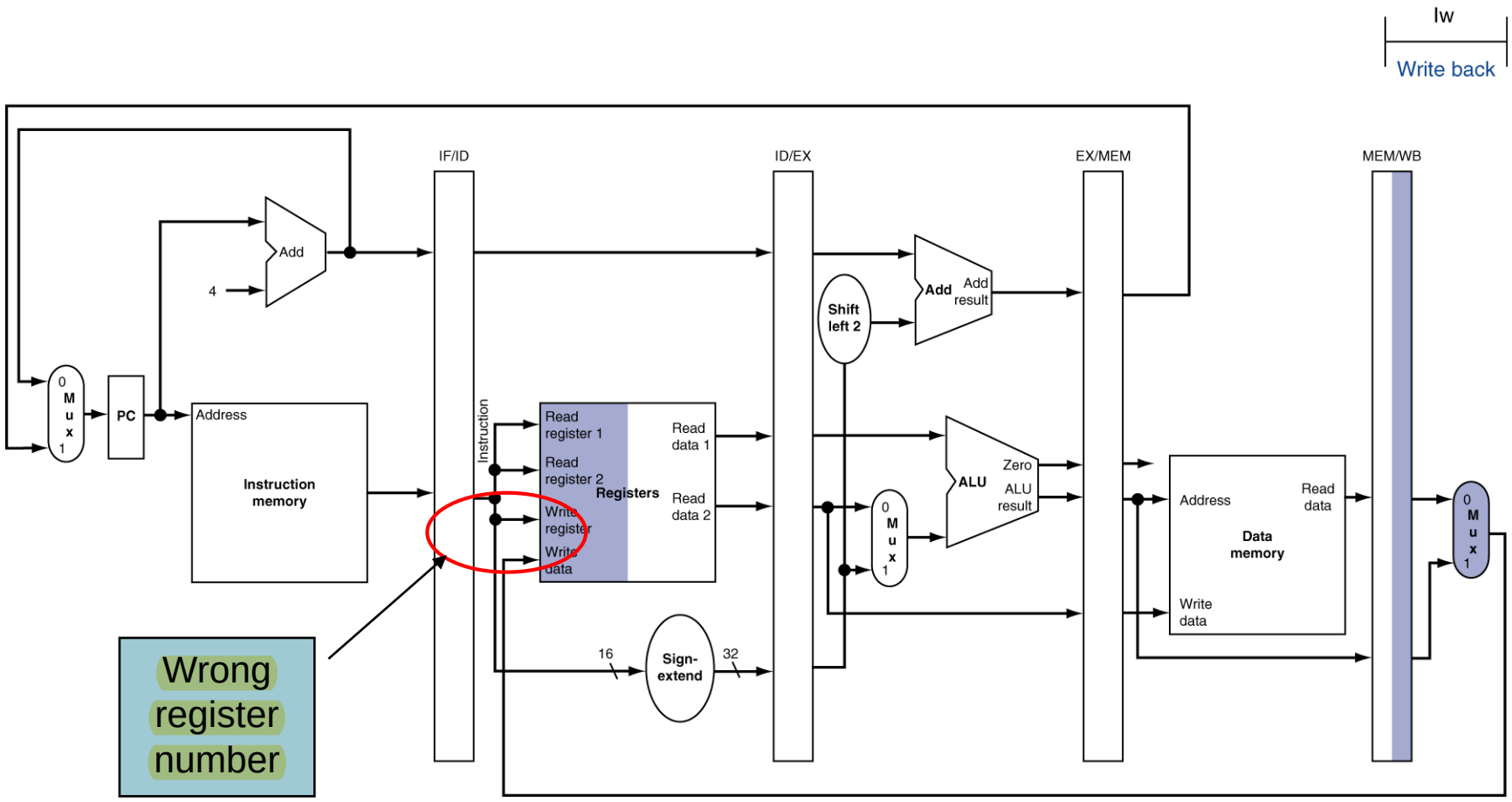
- EX/MEM register: memory address, control signals

MEM for Load



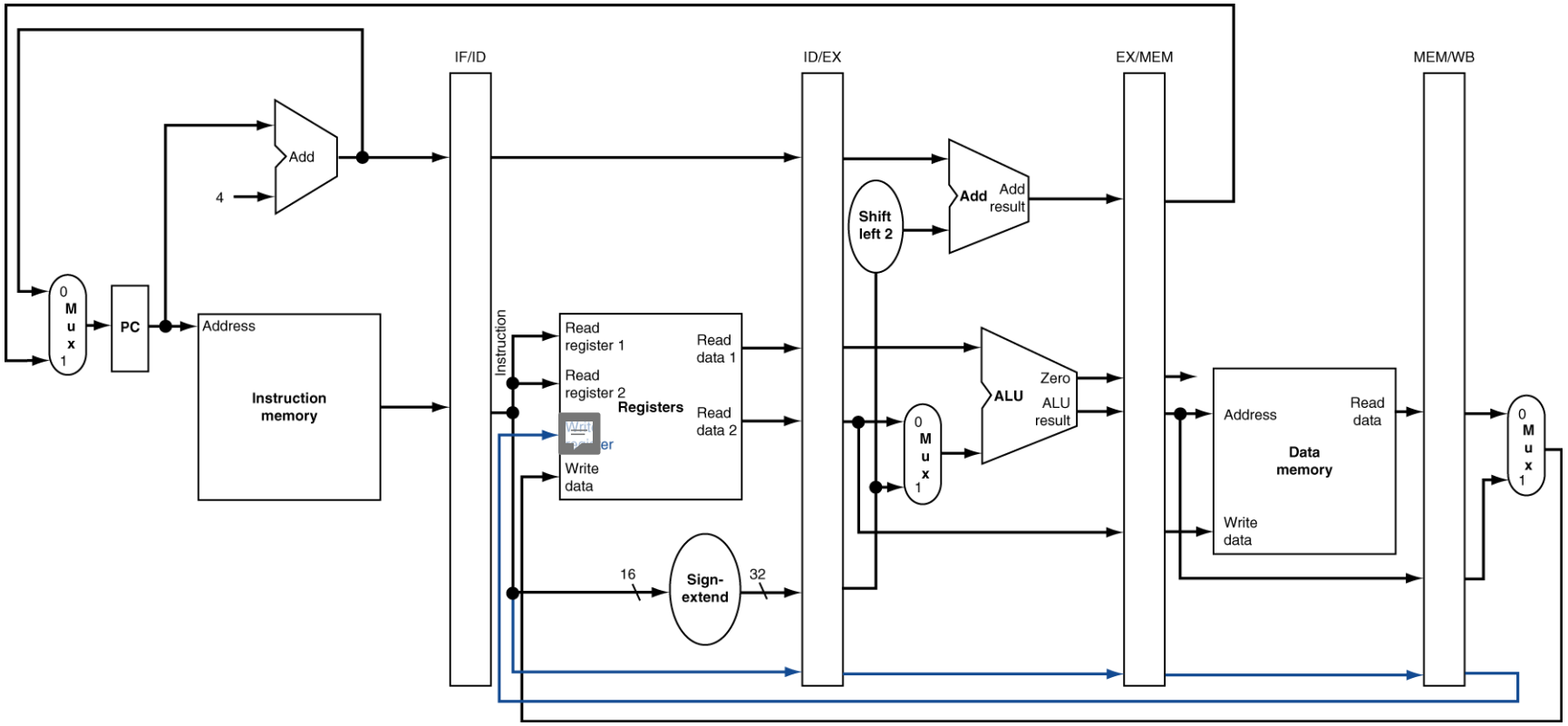
- MEM/WB register: data read, control signals

WB for Load



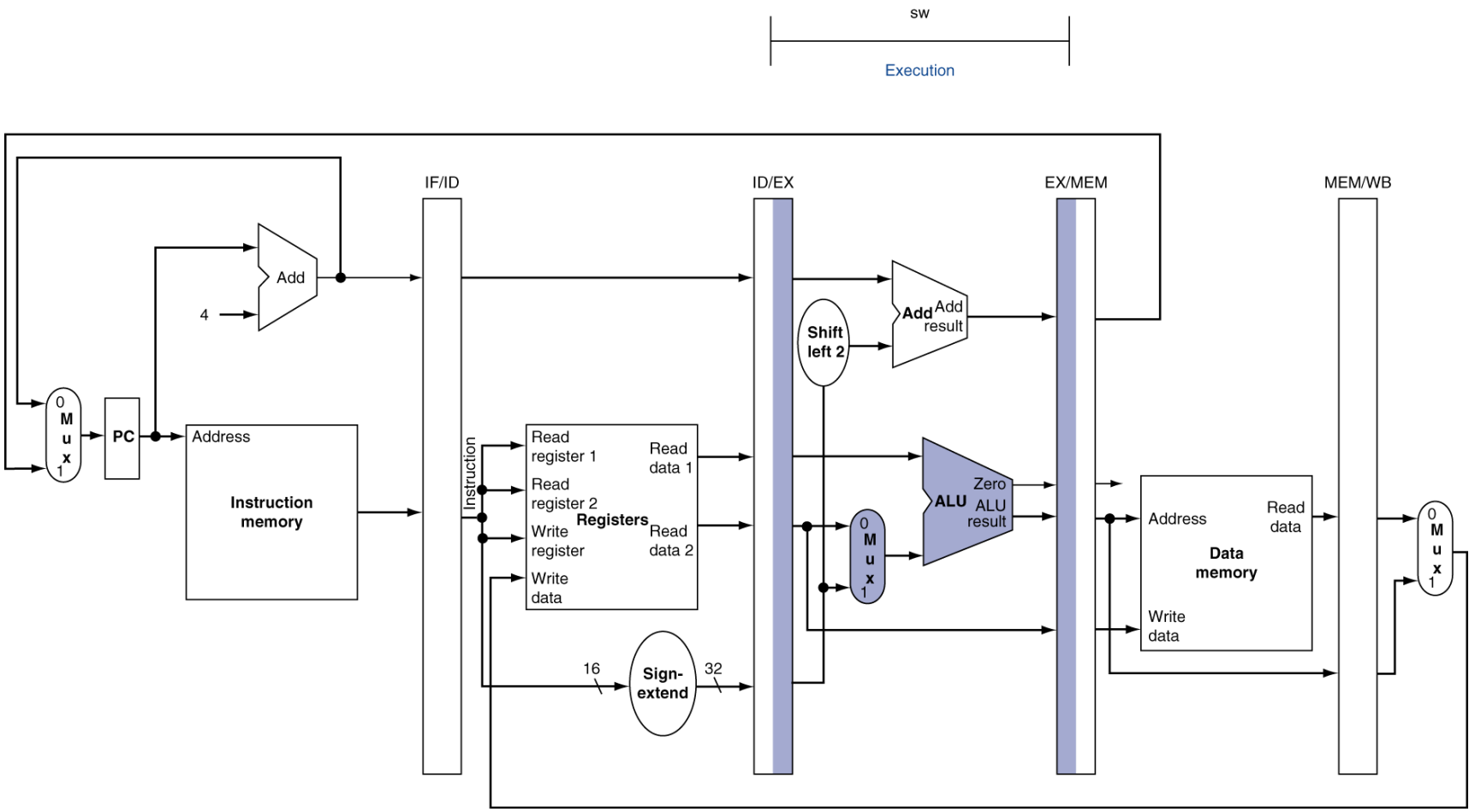
- Execution finished in WB; register write by WB

Corrected Datapath for Load



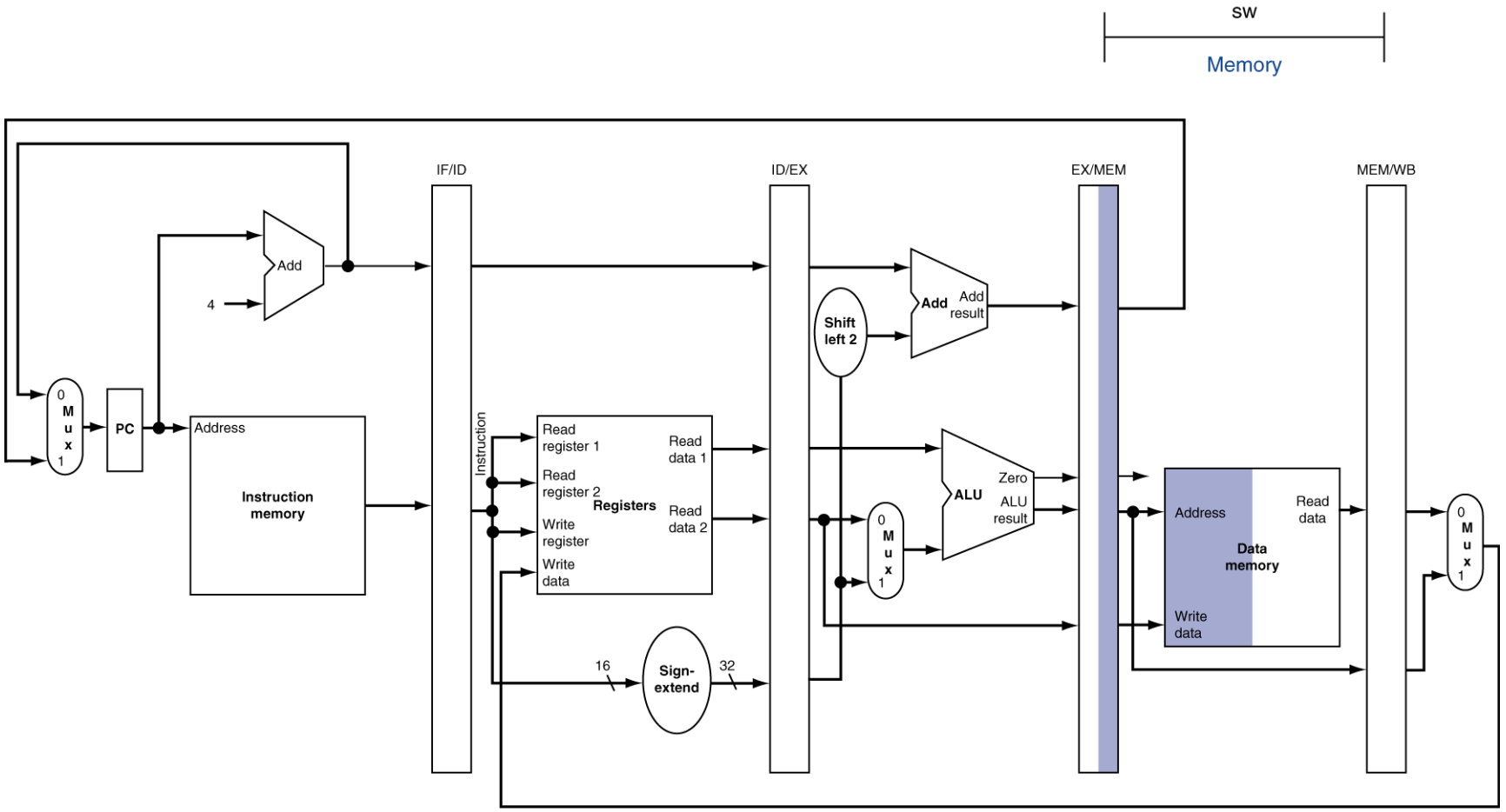
- Carry “write register no.” information all the way through WB

EX for Store



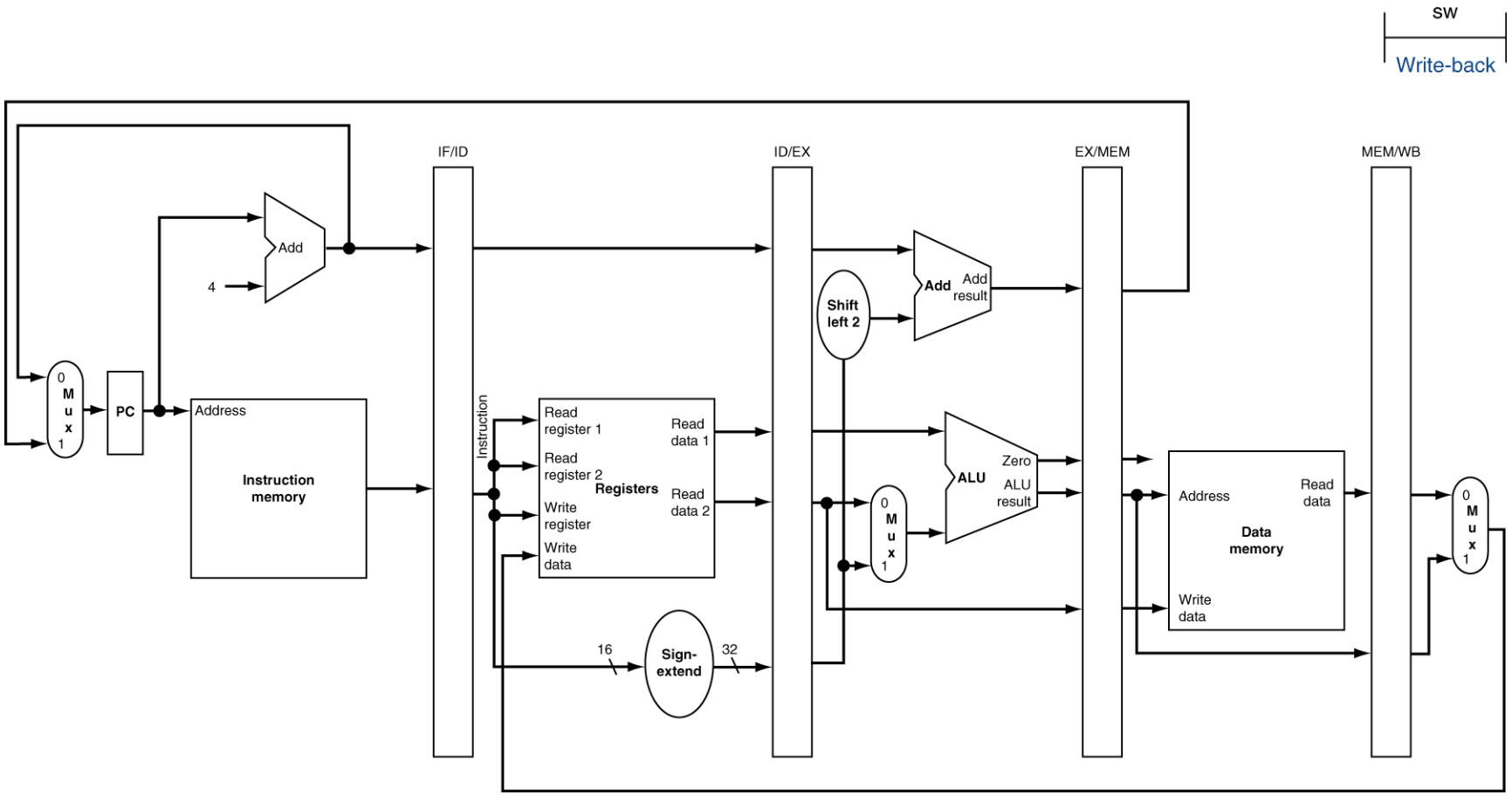
- EX/MEM: memory address, data to write, control signals

MEM for Store



- MEM/WB register: none (store is finished)

WB for Store



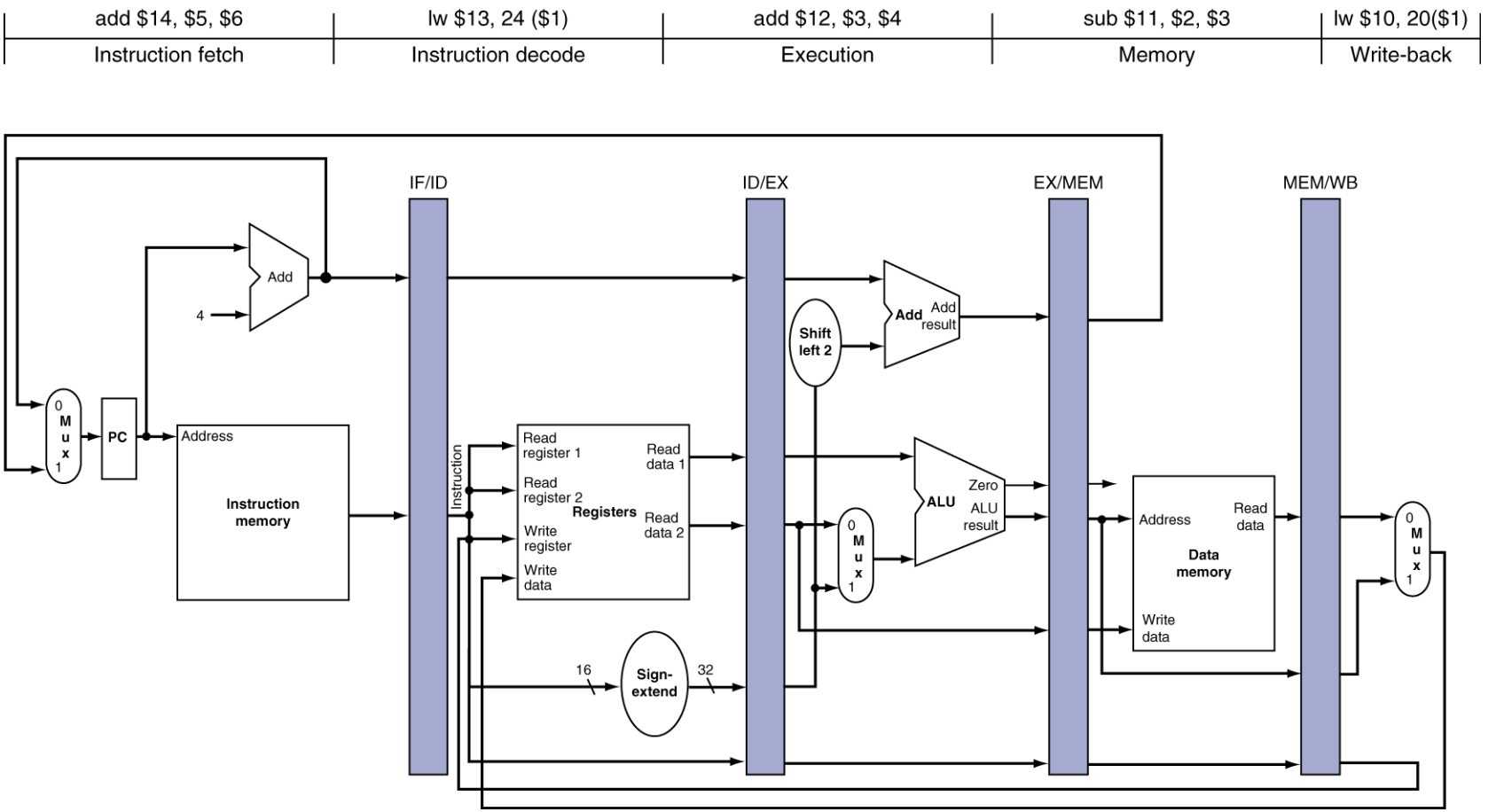
- Nothing to do; doesn't matter
  - Important: at every cycle, start (or finish) one instruction

ALU instructions and Branch

- ALU instructions (“add $3, $1, $2”)
  - EX/MEM: ALU result, write register no., control signals
    - NO operation in MEM stage
  - MEM/WB: ALU result, write register no., control signals
  - Execution finished in WB stage
- Branch instructions (“beq $1, $2, 8”
  - EX/MEM: branch target, zero or not, control signals
  - Execution finished in MEM stage
    - Update PC if branch is taken (do nothing if untaken)

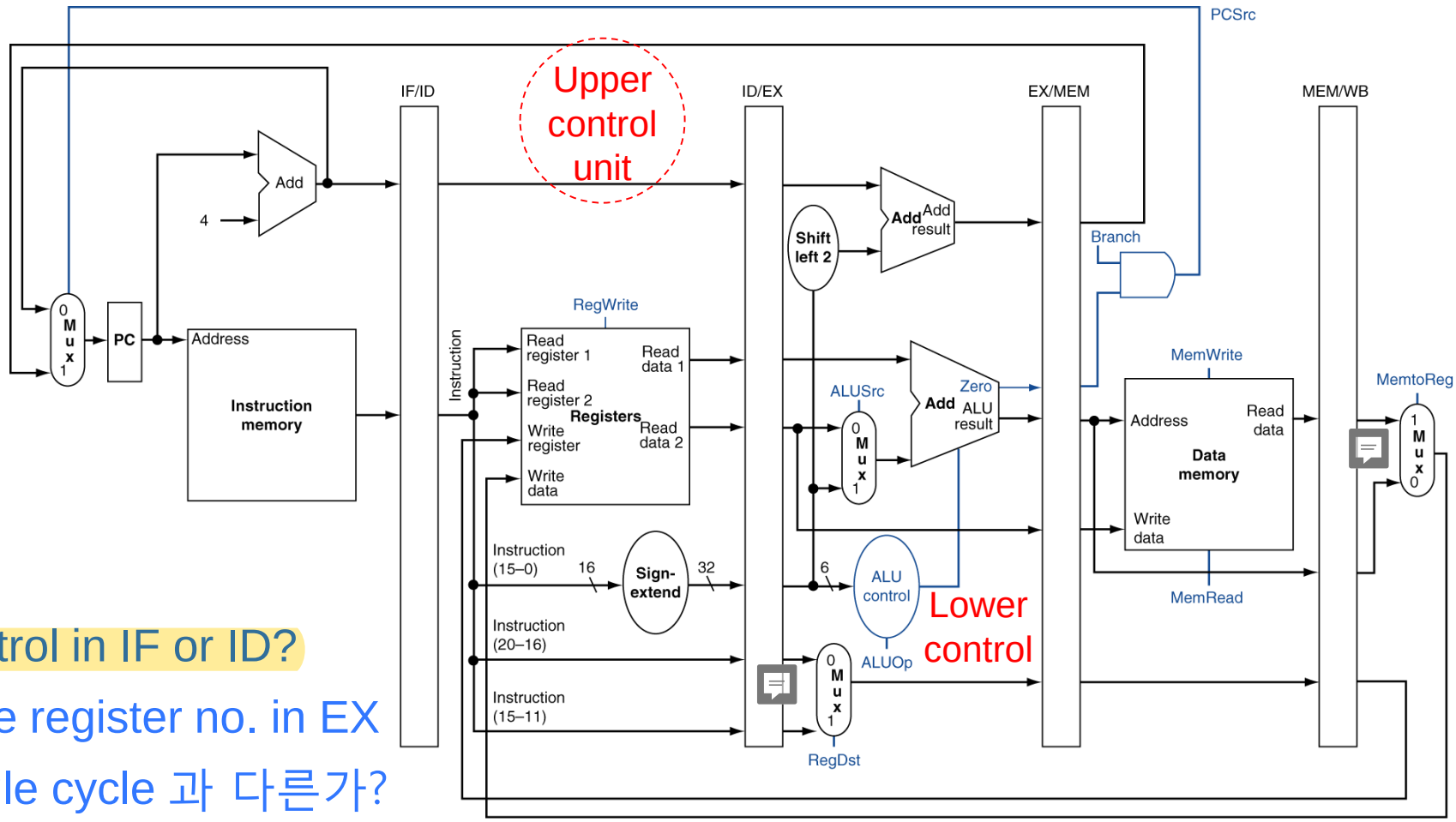
Pipeline in Action

- Each stage run a different instruction



Pipelined Control

Pipelined Control (Simplified)

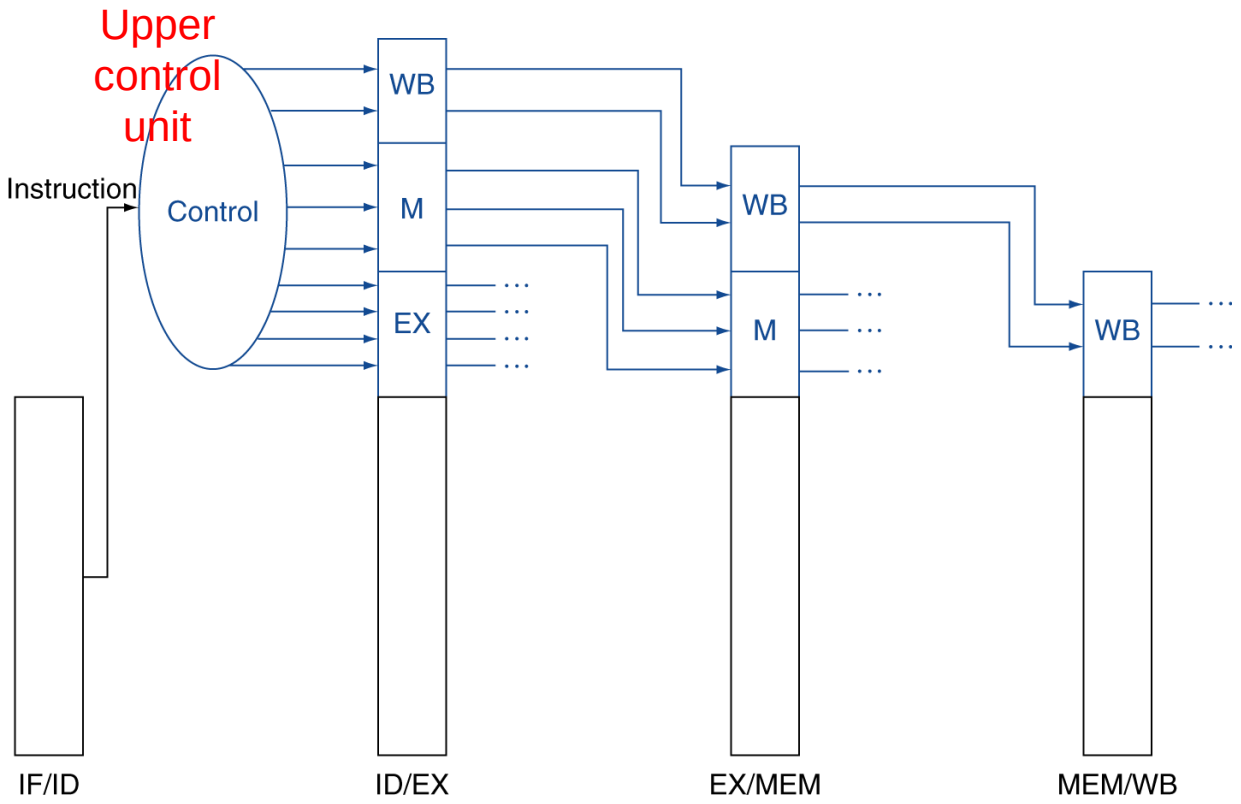


1) Control in IF or ID?
2) Write register no. in EX
3) Single cycle 과 다른가?
   - Stage 별로 구분되었을 뿐

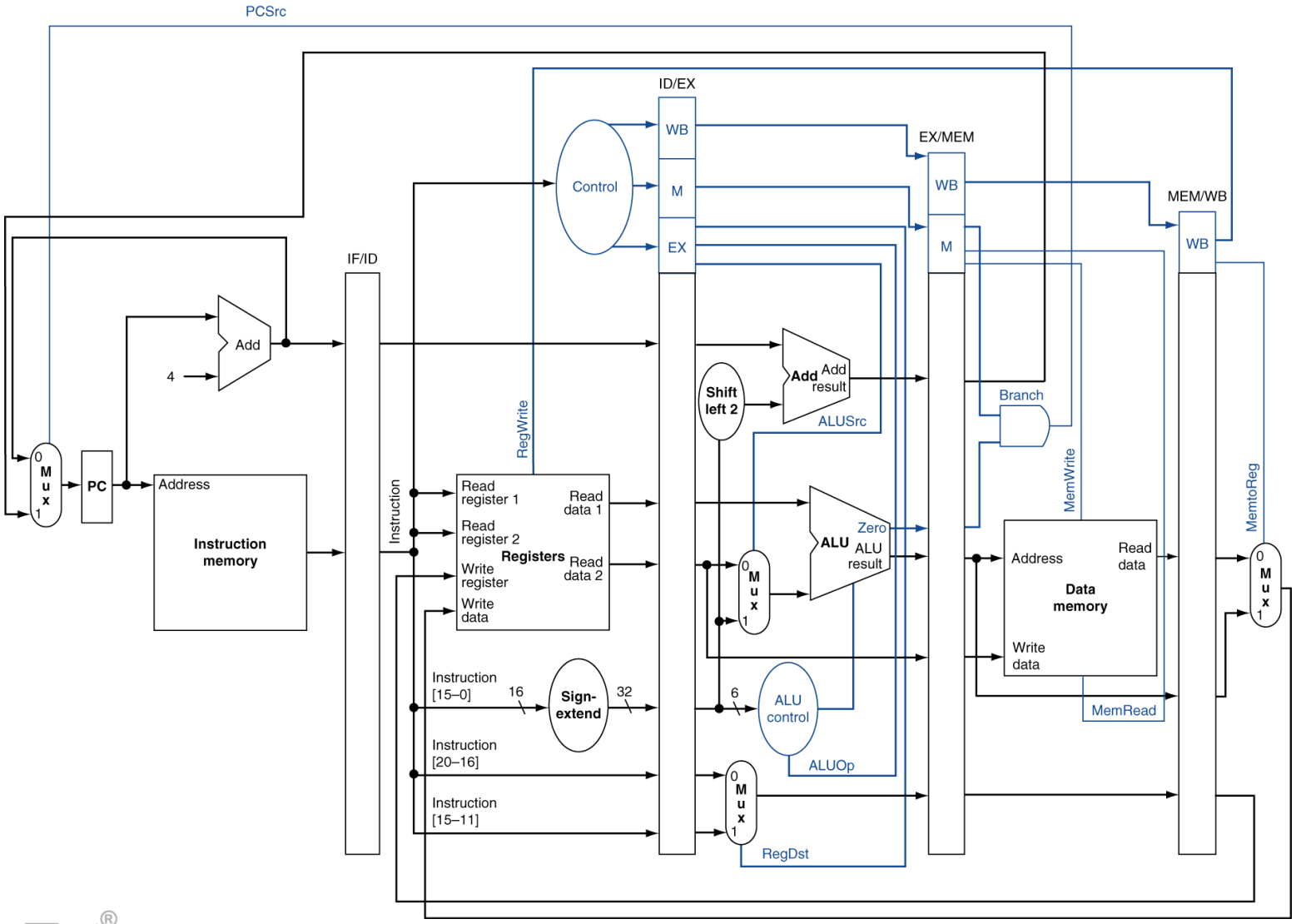
Pipelined Control

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

Single cycle 과 동일한 control unit



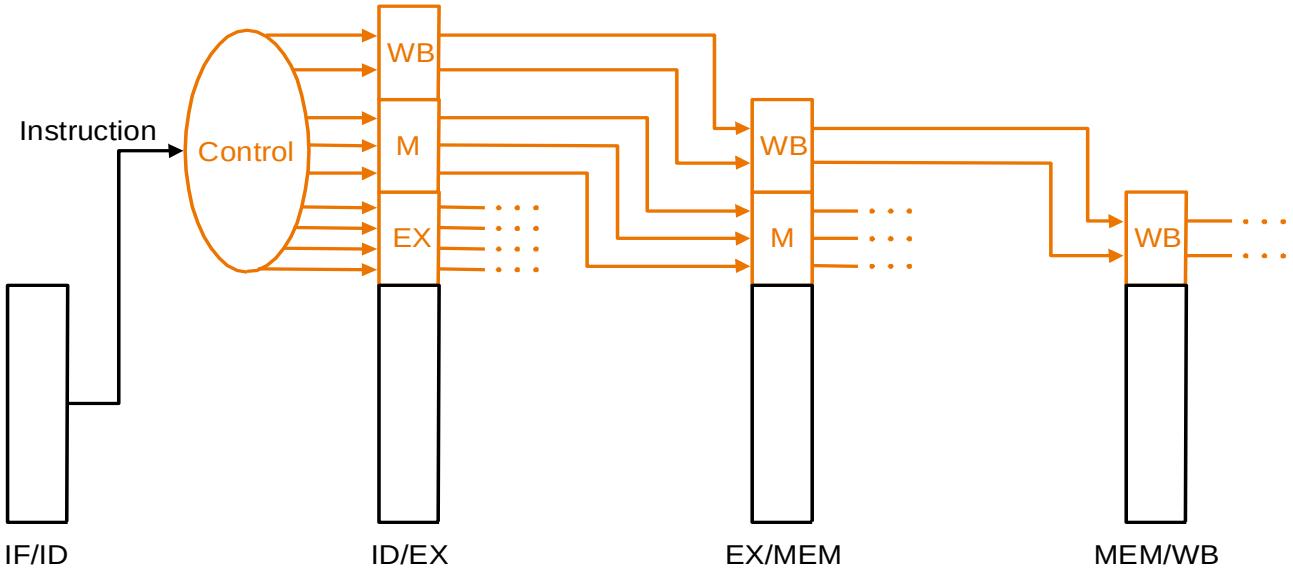
Pipelined Control



Pipeline Control

- Same truth table we saw in single cycle (stage 별로 구분)

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



Pipelining and ISA Design (부연)

- 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
    - Calculate address in 3rd, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Chapter 4

Pipeline Hazards

(Problem-solving for powerful machines)

Hazards (미리보기)

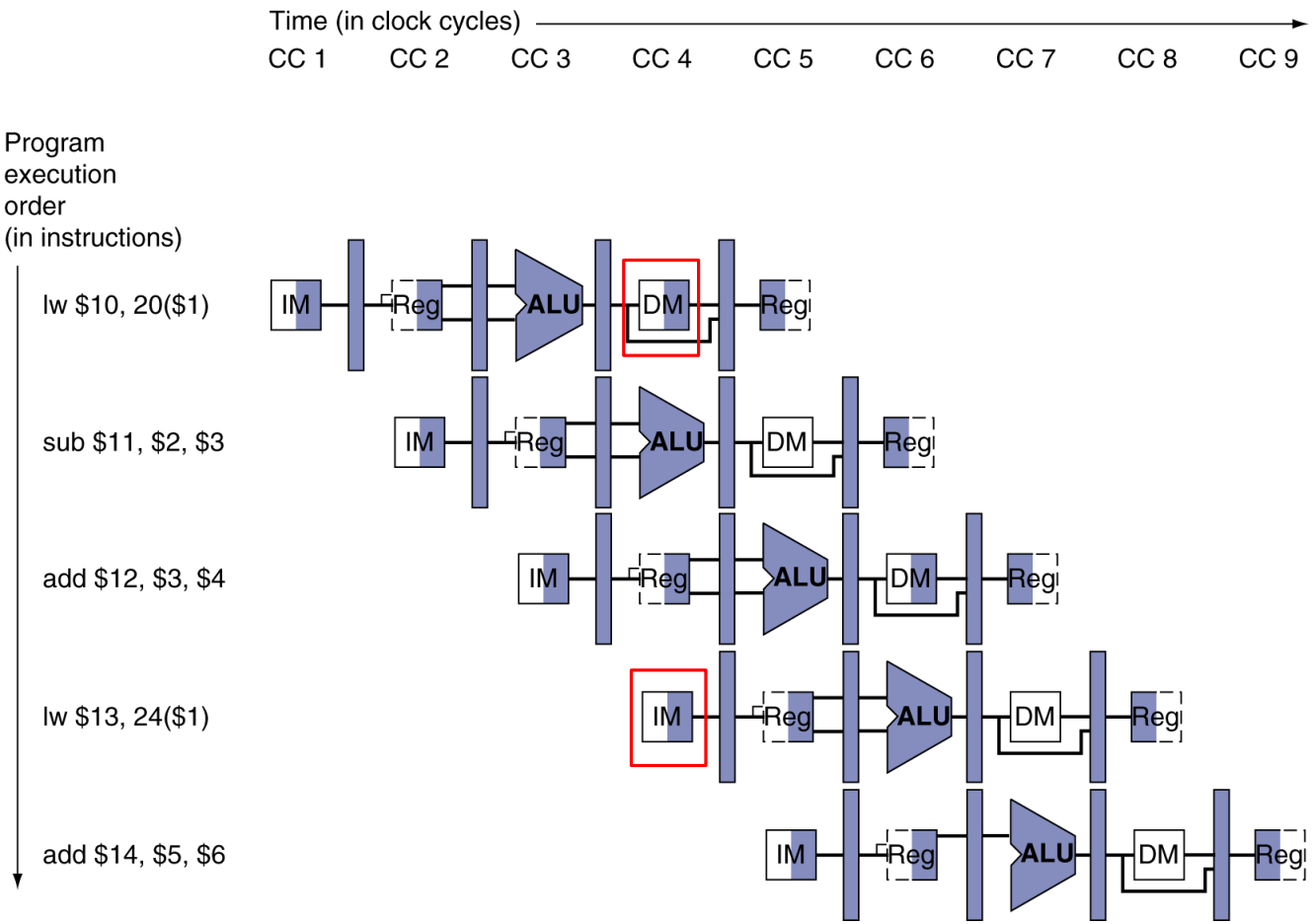
- Situations that prevent starting the next instruction in the next cycle
  1) Structural hazards
     - A required resource is busy
  2) Data hazards
     - Need to wait for previous instruction to complete its data read/write
  3) Control (or branch) hazards
     - Deciding on control action depends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

Structure Hazards

- Conflict for use of a resource
  - Different stages want to use same functional unit
- 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”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

Structural Hazards

- Integrated IM and DM – impact on CPI? (1 → 1.2)

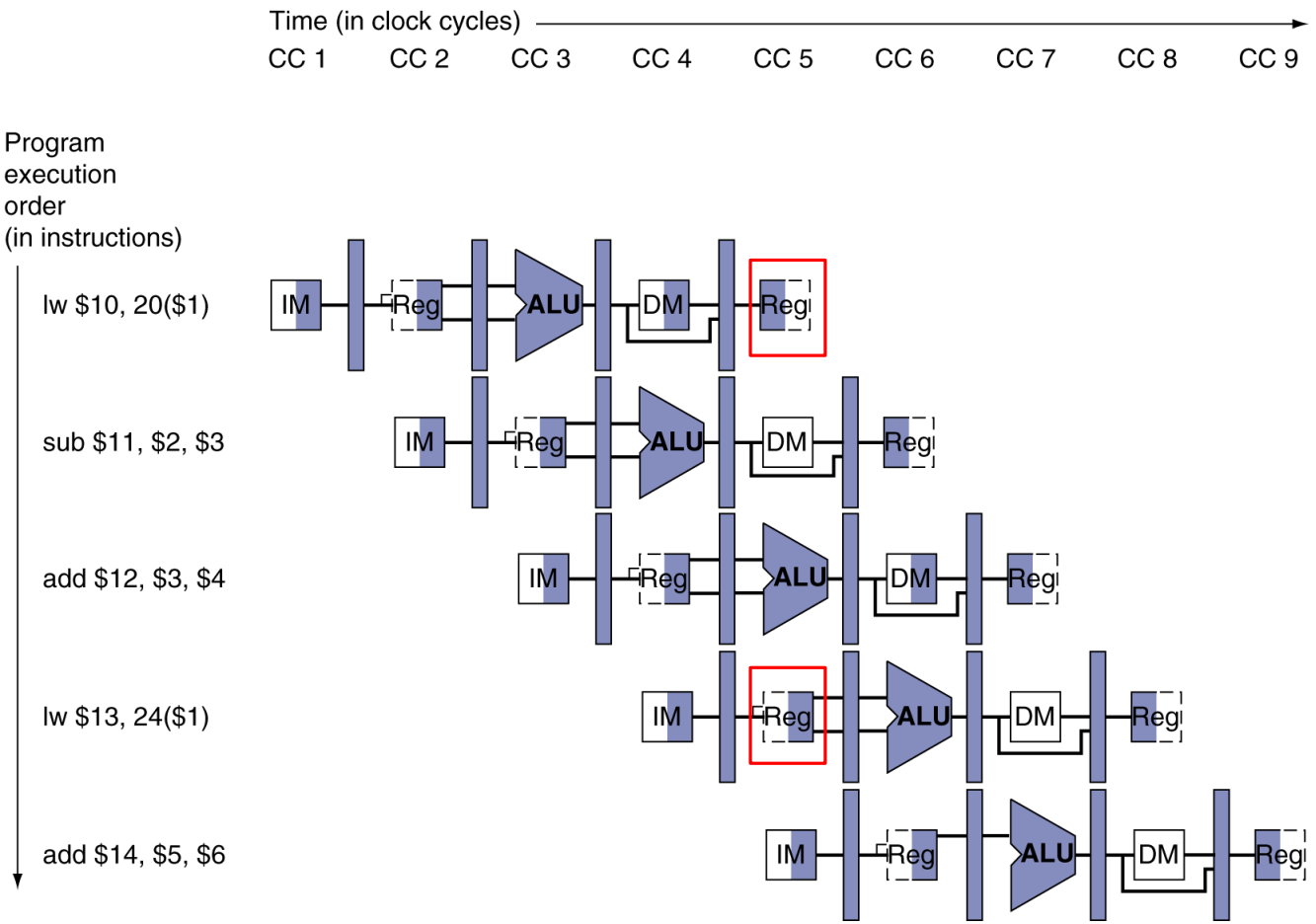


Structural Hazards

- Different stages want to use same functional unit
  - Can avoid with harwdare duplication
- But what about register access?
  - ID vs. write back
  - Divide a clock period into two halves
    - Register access: 100 ps, other operations: 200 ps
- Our datapath has no structural hazards
- Why the name “structural” hazards?

Structural Hazards

- Register access: divide a clock period into two halves

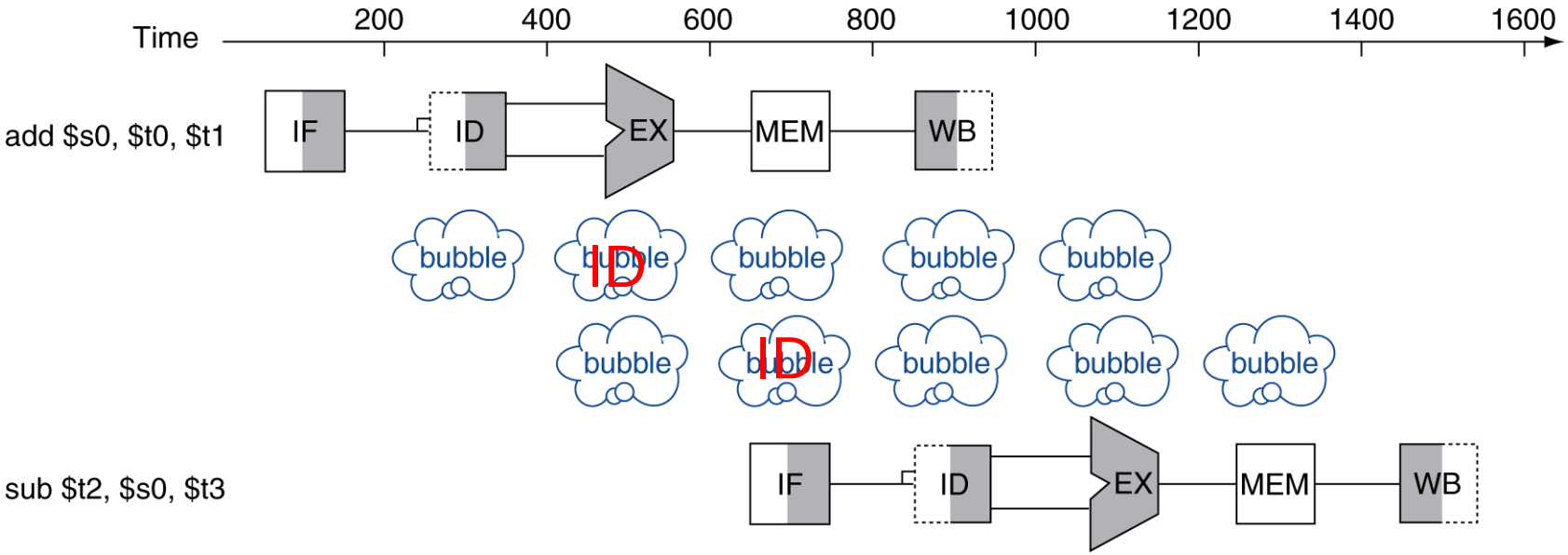


Data Hazards

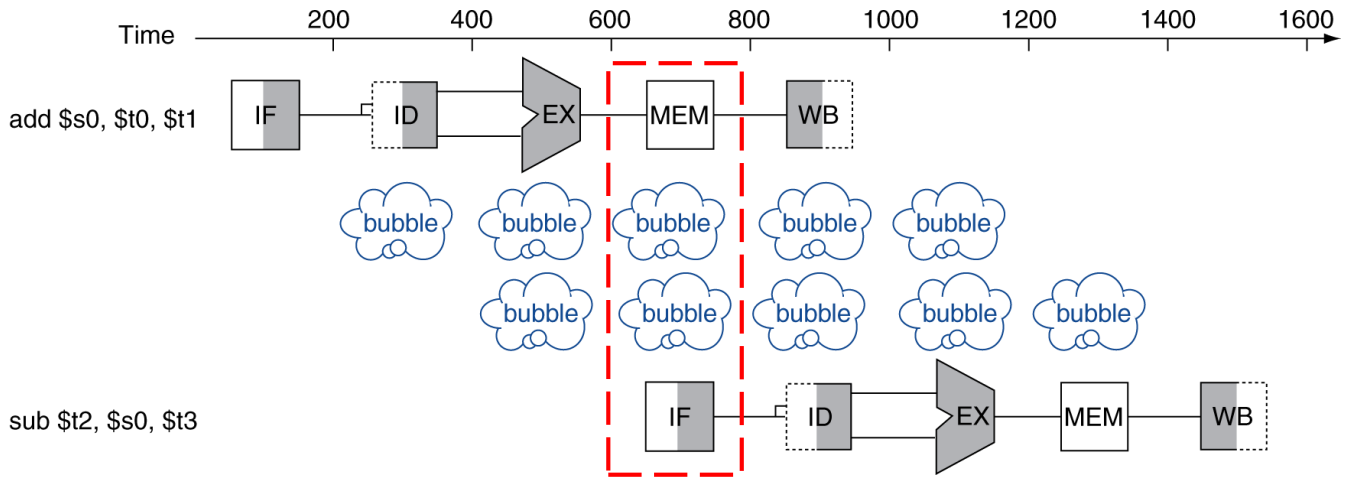
Data Hazards

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

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



Pipeline Stall, Bubble

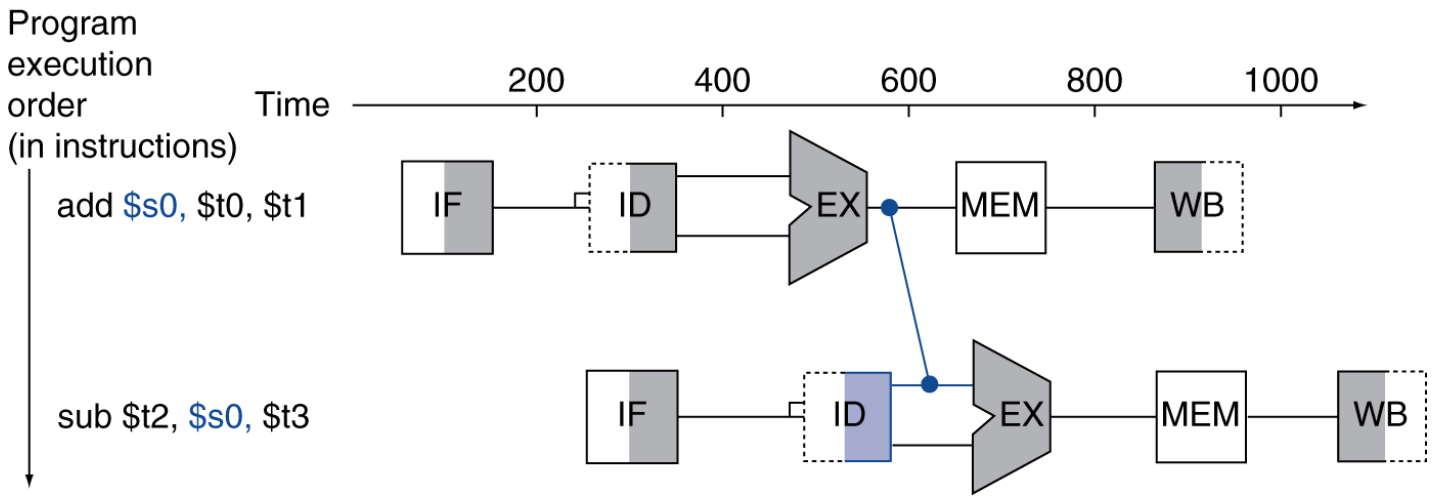


- Lose two cycles: increase in CPI
  - What is bubble: **nop** instruction (e.g., **sll** $0, $0, 0)
    - All it does is to increment PC by 4

IF	ID	EX	MEM	WB
sub	nop	nop	add	I_{-1}

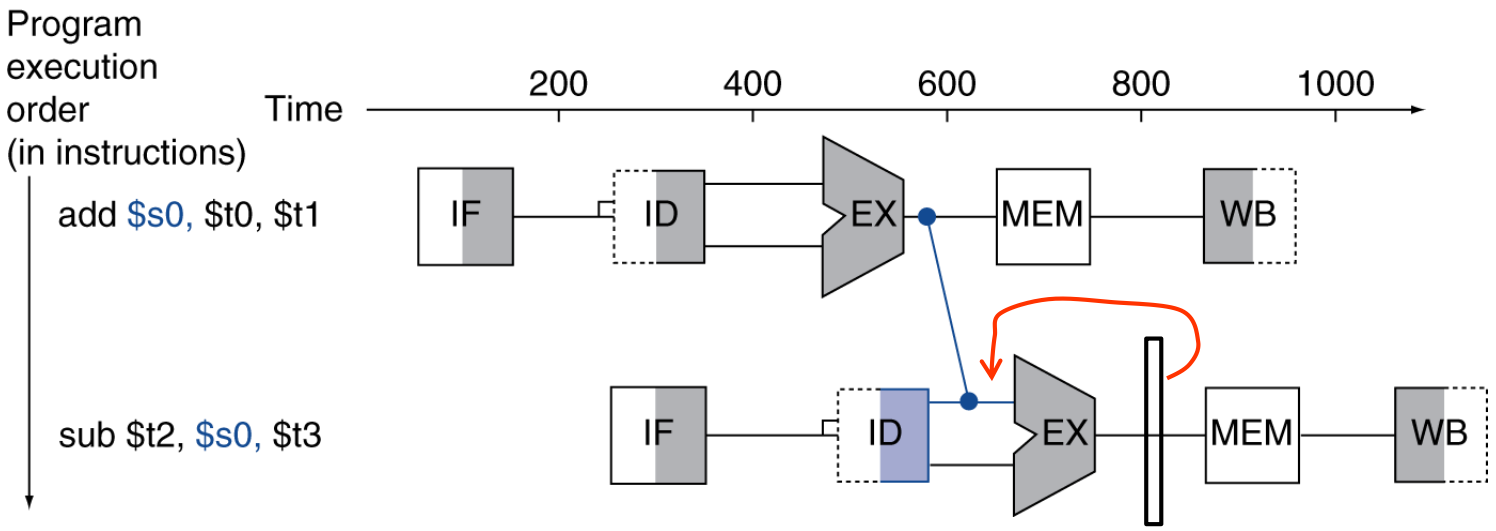
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath



Forwarding (aka Bypassing)

- Add feedback datapath from EX/MEM pipeline register



Data Hazards and Forwarding (1)

Data Hazards in ALU Instructions

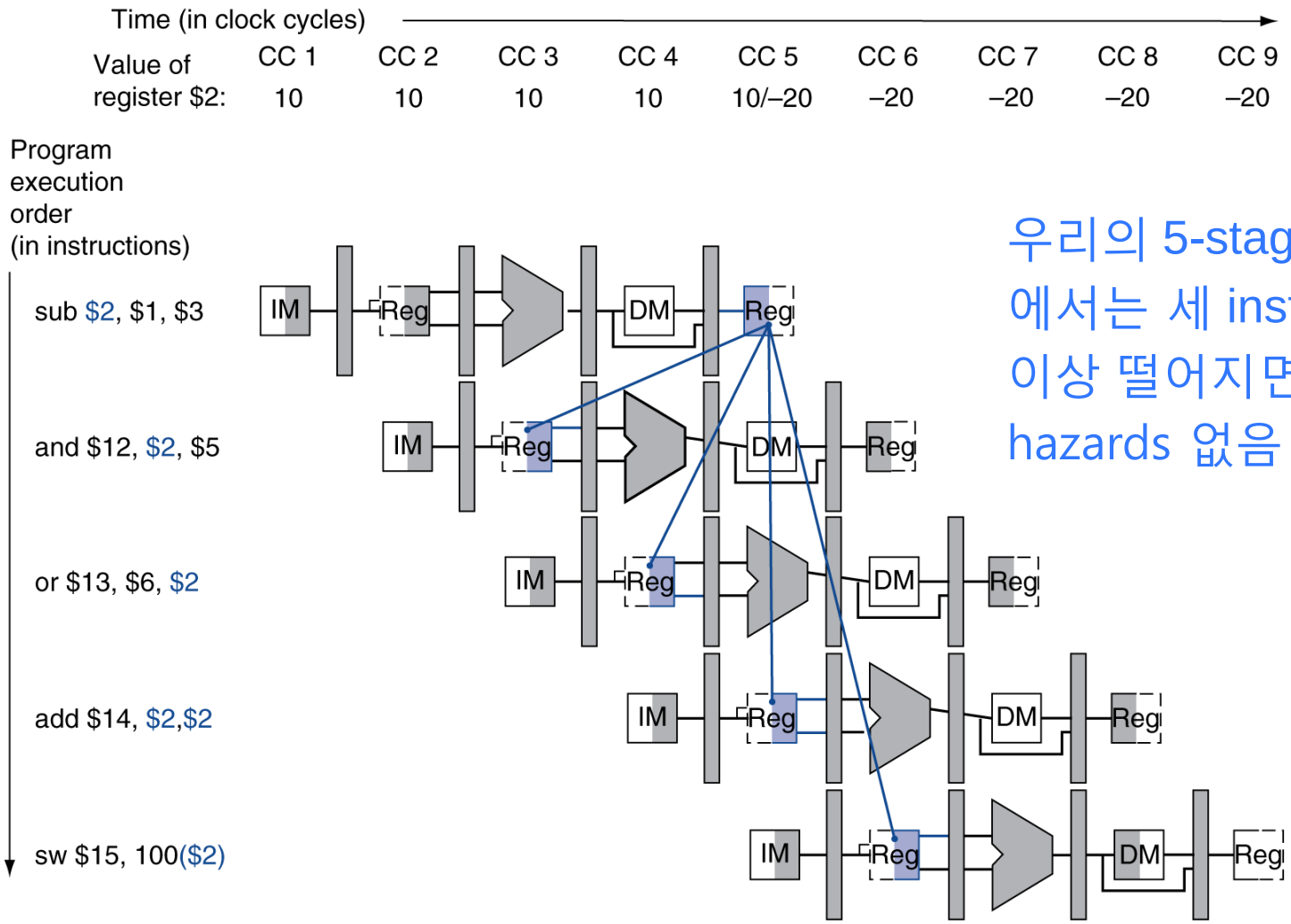
- Consider this sequence:

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

Data dependence
vs. parallelism

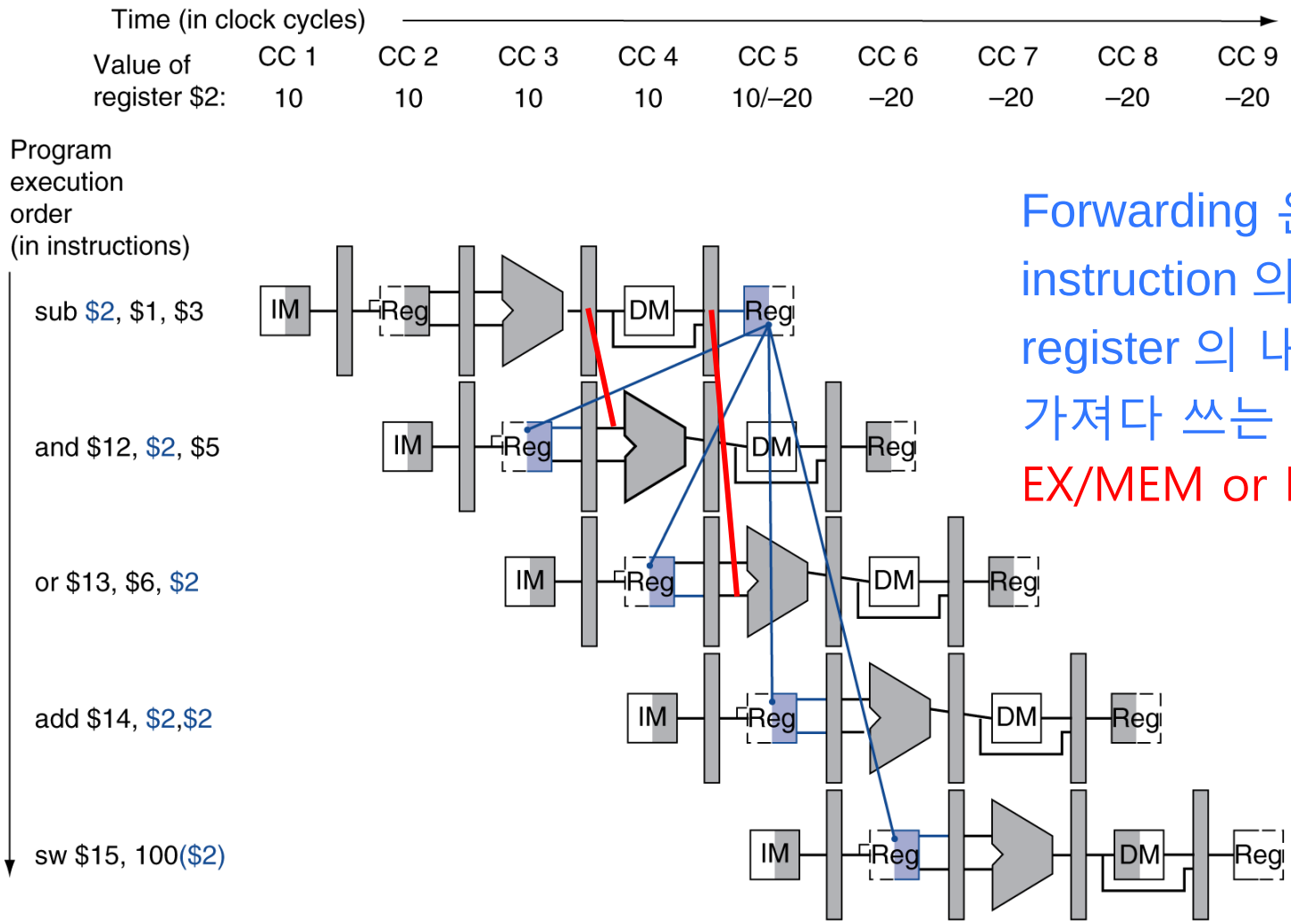
- RAW (read after write) data dependence (프로그램 특성)
- We can resolve hazards with forwarding
  - How do we detect when to forward?

RAW Data Dependence



우리의 5-stage pipeline 에서는 세 instruction 이상 떨어지면 data hazards 없음

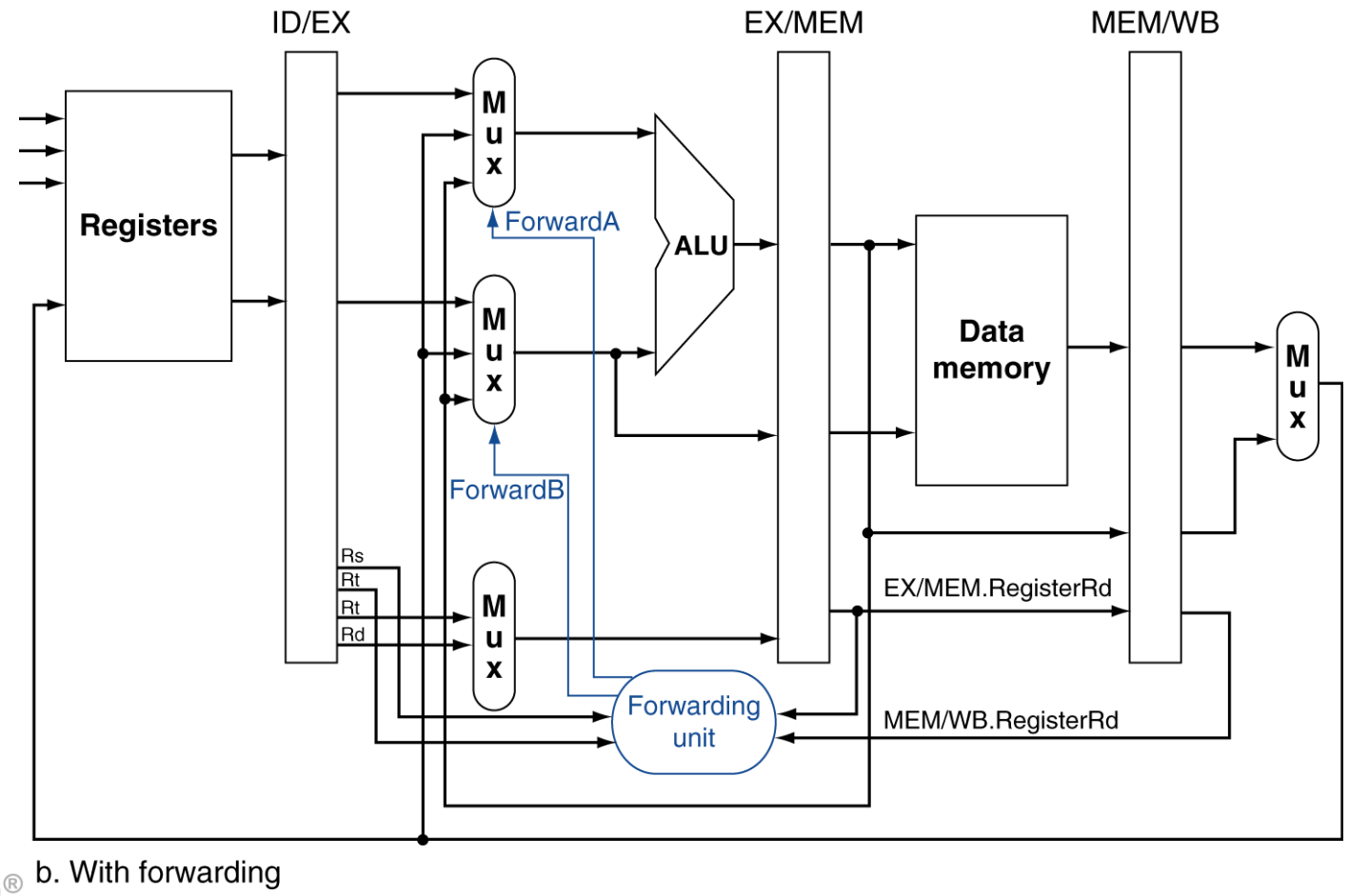
Dependence and Forwarding



Forwarding 은 앞의 두 instruction 의 pipeline register 의 내용을 가져다 쓰는 것임;
EX/MEM or MEM/WB

Datapath for Forwarding

- More datapath and control

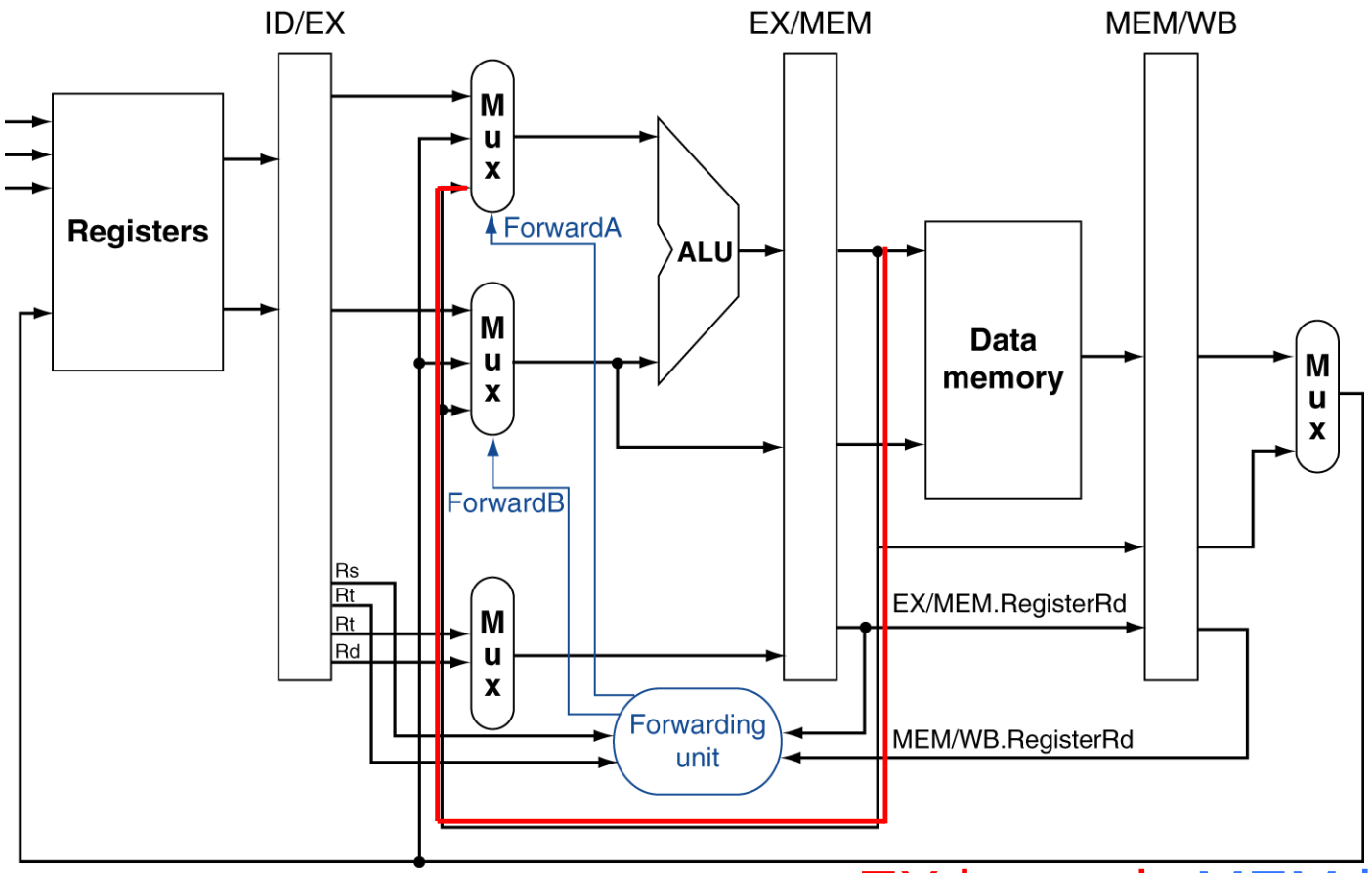


b. With forwarding

What does the forwarding unit do?

Datapath for Forwarding

- Case study: forwarding from EX/MEM to ALU upper input



b. With forwarding

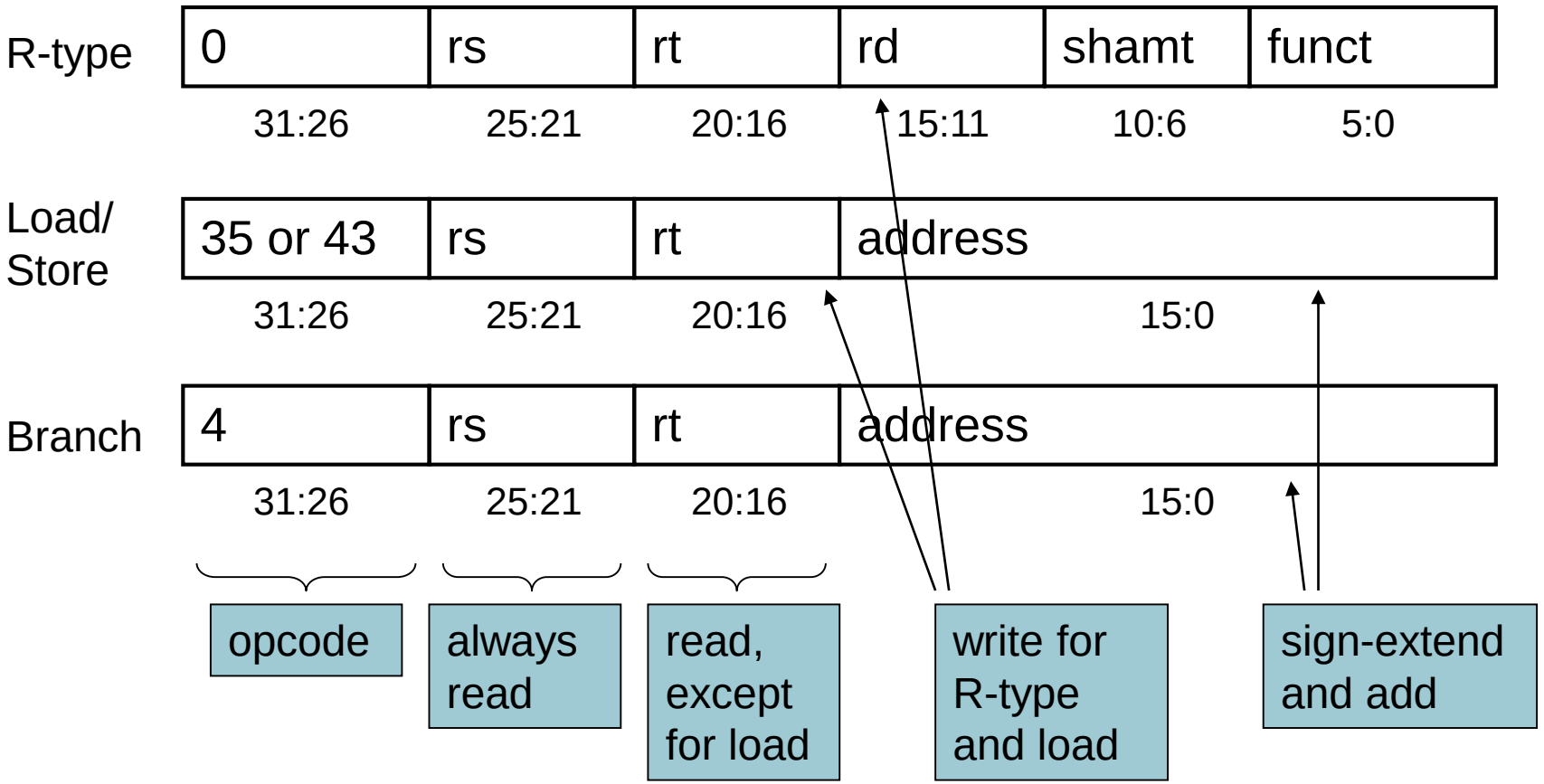
EX hazard MEM hazard

Detecting EX Hazard (condition 1)

- Register number for ALU upper operand
  - ID/EX.RegisterRs
- 바로 앞선 instruction 이 결과를 쓰는 register
  - EX/MEM.RegisterRd
- Data hazards when
  - EX/MEM.RegisterRd = ID/EX.RegisterRs

Fwd from EX/MEM pipeline reg

MIPS Instruction Format (반복)



Detecting EX Hazard (condition 2)

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite = 1
    - ALU or lw instructions
- Some instructions don't forward data to others
  - They don't write data to registers

  (or they don't compute new register values)

```
beq  $t1,  $t2,  4
sw   $t1,  8($gp)
```

Detecting EX Hazard (condition 3)

- And only if Rd for forwarding instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0
- 컴파일러는 필요시 destination 이 $zero 인 instruction 만듬 (예: sll $0, $0, 0)
  - 이런 instruction 으로 부터는 forwarding 받으면 안 됨

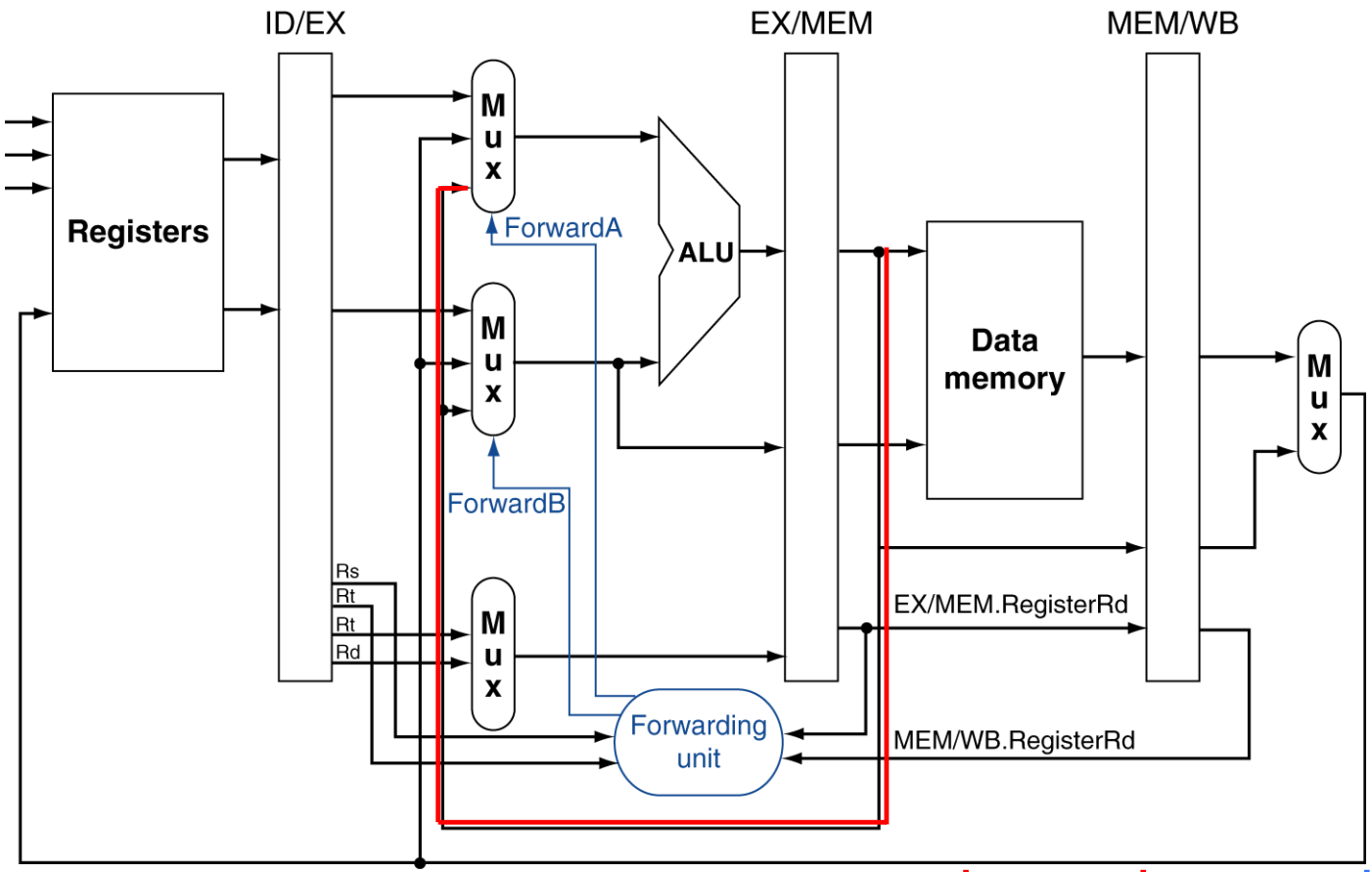
Forwarding Conditions

- Forwarding from EX/MEM to ALU upper input
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))

    ForwardA = 10

Datapath for Forwarding (반복)

- Case study: forwarding from EX/MEM to ALU upper input



b. With forwarding

EX hazard MEM hazard

Data Hazards and Forwarding (2)

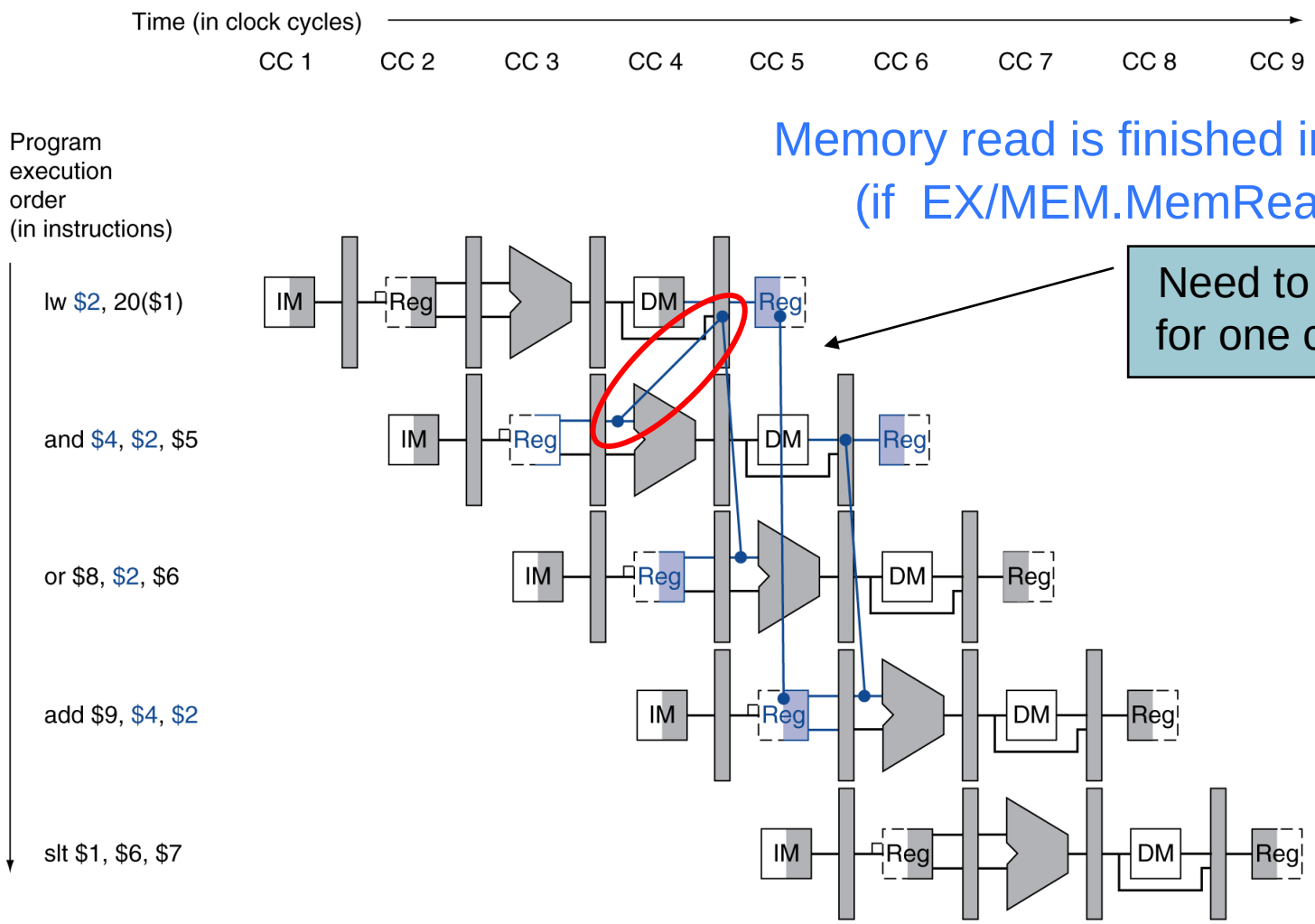
Forwarding Conditions

- Forwarding from EX/MEM to ALU upper input (반복)
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))

    ForwardA = 10

- What is missing?
  - If “load” instruction in MEM stage, cannot forward
    - if EX/MEM.MemRead = 1

Load-Use Data Hazard (미리보기)



Control for Forwarding

- EX hazard (if EX/MEM.MemRead = 0)
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10

We did it!

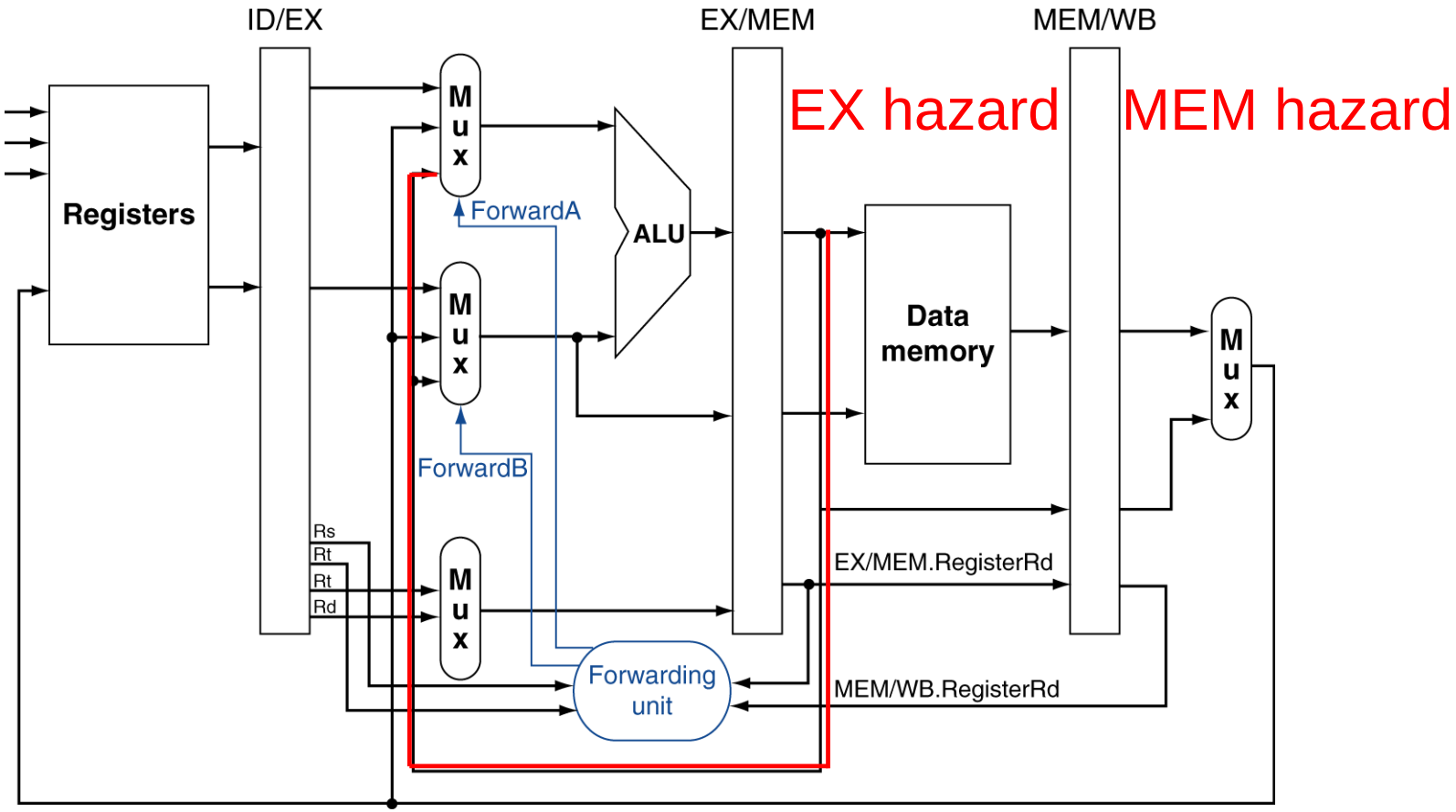
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10

ALU lower input

- 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

Forwarding Paths (반복)

- Designed forwarding from EX/MEM to ALU upper input



b. With forwarding

Thinking about Forwarding

- Source of forwarding: EX/MEM or MEM/WB
  - Instructions that produce new register values
    - Producer: **ALU** instructions, **lw**
    - Non-producer: **sw, beq**
- Destination of forwarding: ALU inputs, write data to DM
  - Where data is consumed
  - Any instruction can receive forwarding
    - Forwarding to DM write data – exercise!

Double Data Hazard

- Consider the sequence:

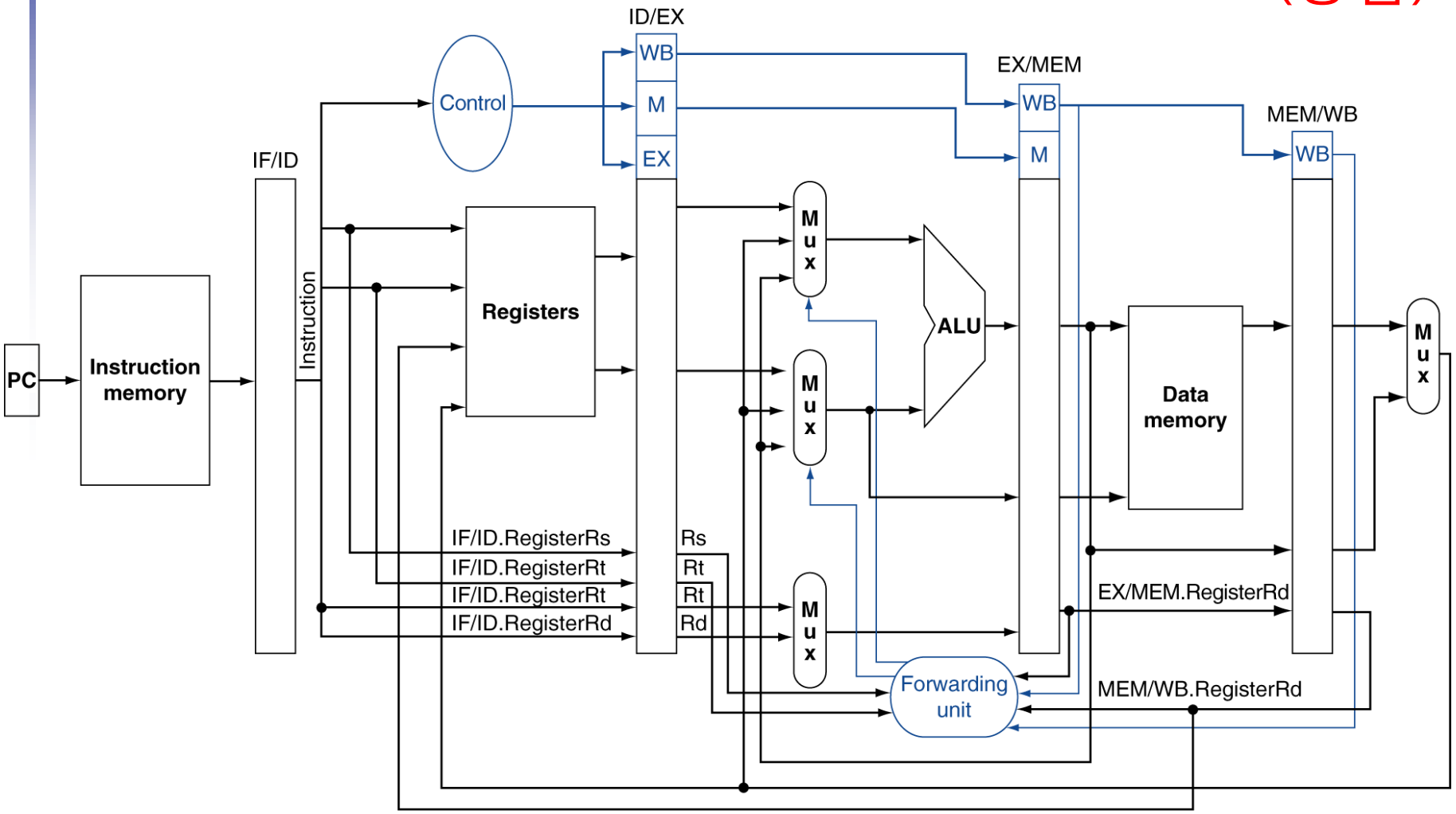
```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

- 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(참고)

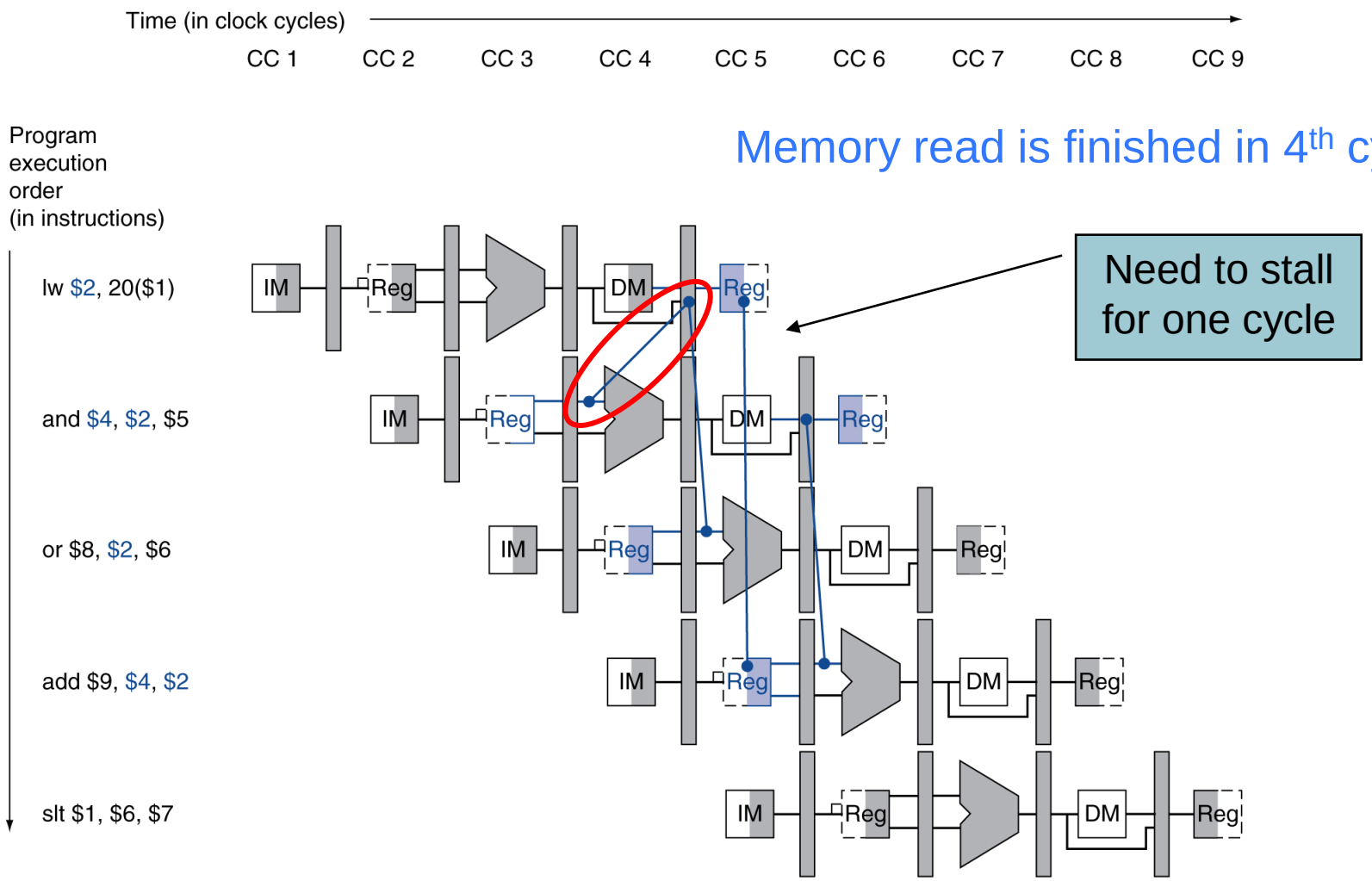
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite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

Datapath and Control with Forwarding (종합)



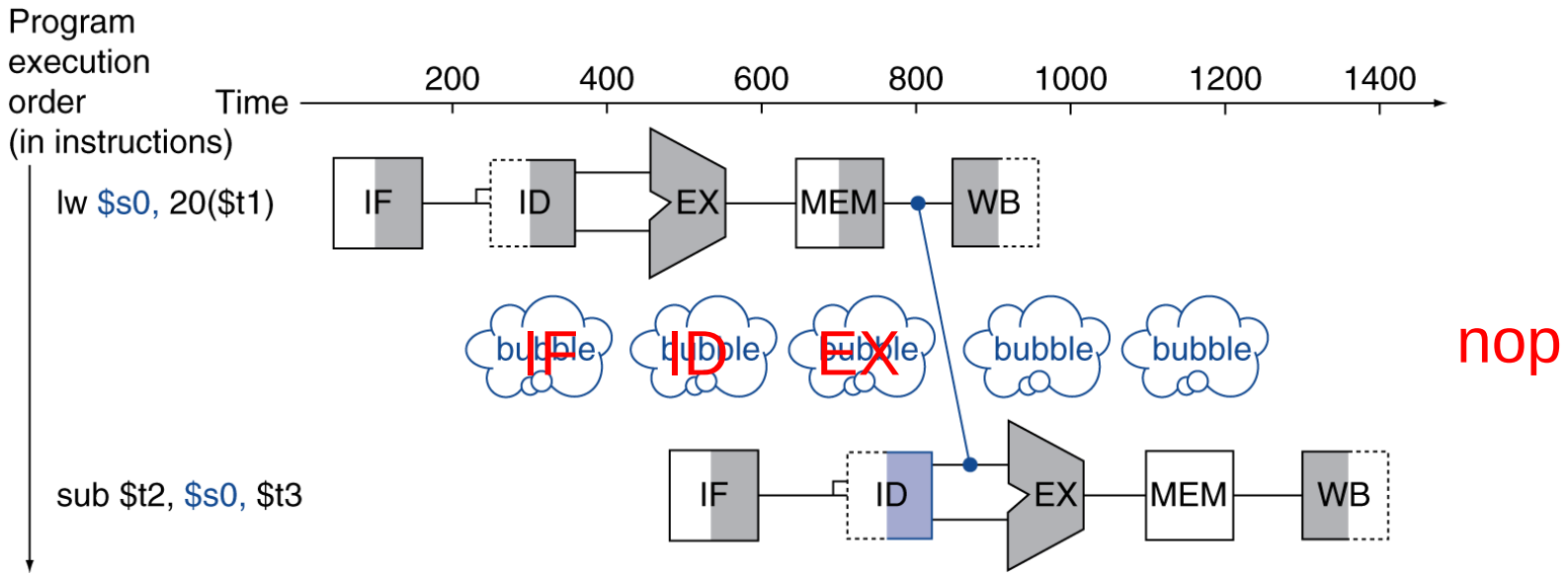
Load-Use Data Hazard

Load-Use Data Hazard (반복)



Load-Use Data Hazard

- Can't always avoid stalls by forwarding
- Only data hazard that require stall in our 5-stage design
  - CPI: 1 → 1.05



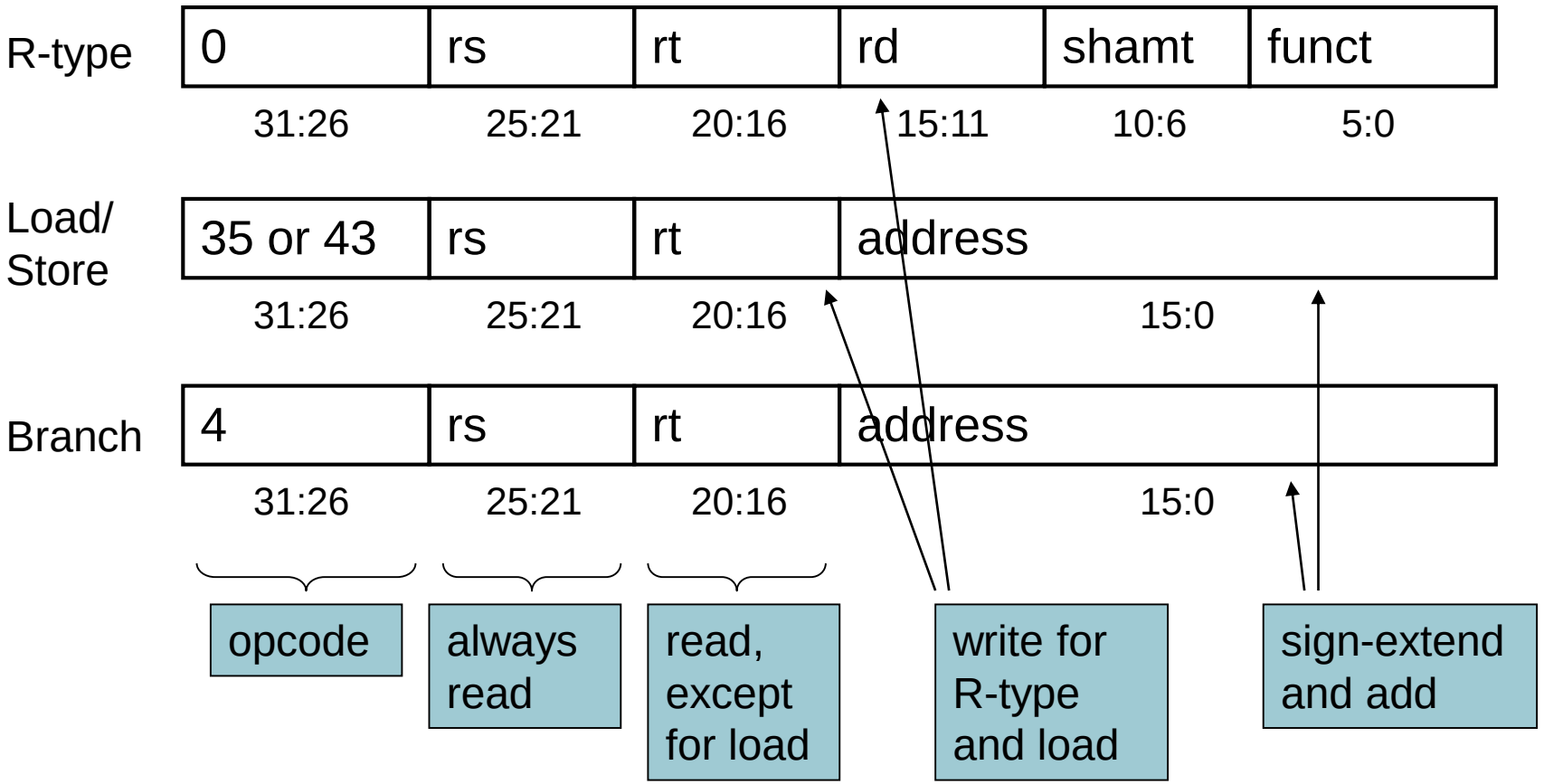
Load-Use Hazard Detection

- 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 (i.e., insert **nop** instruction or bubble)

MIPS Instruction Format (반복)



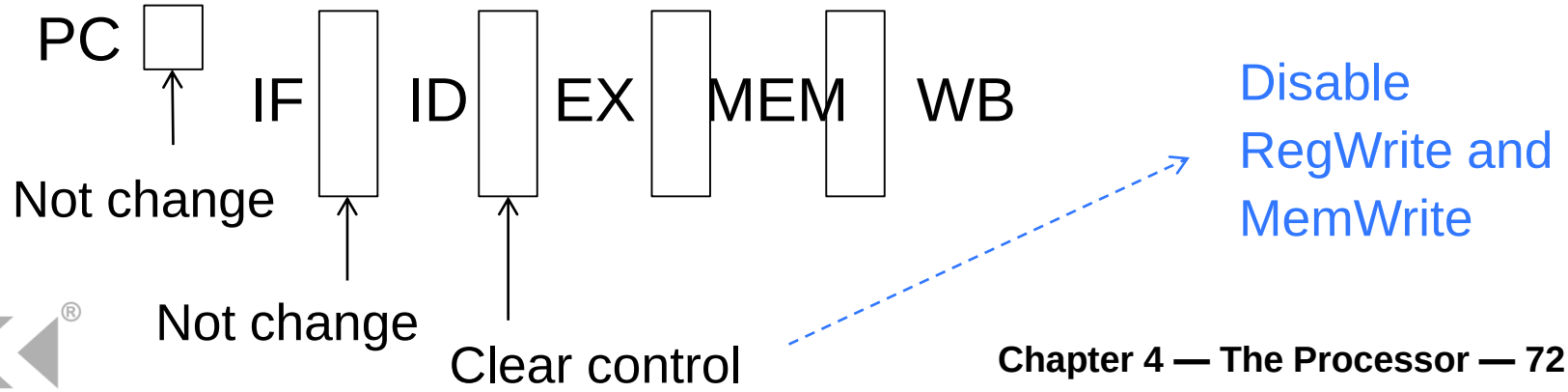
How to Stall the Pipeline

	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

lw r1, 4(r2)

and r4, r1, r3

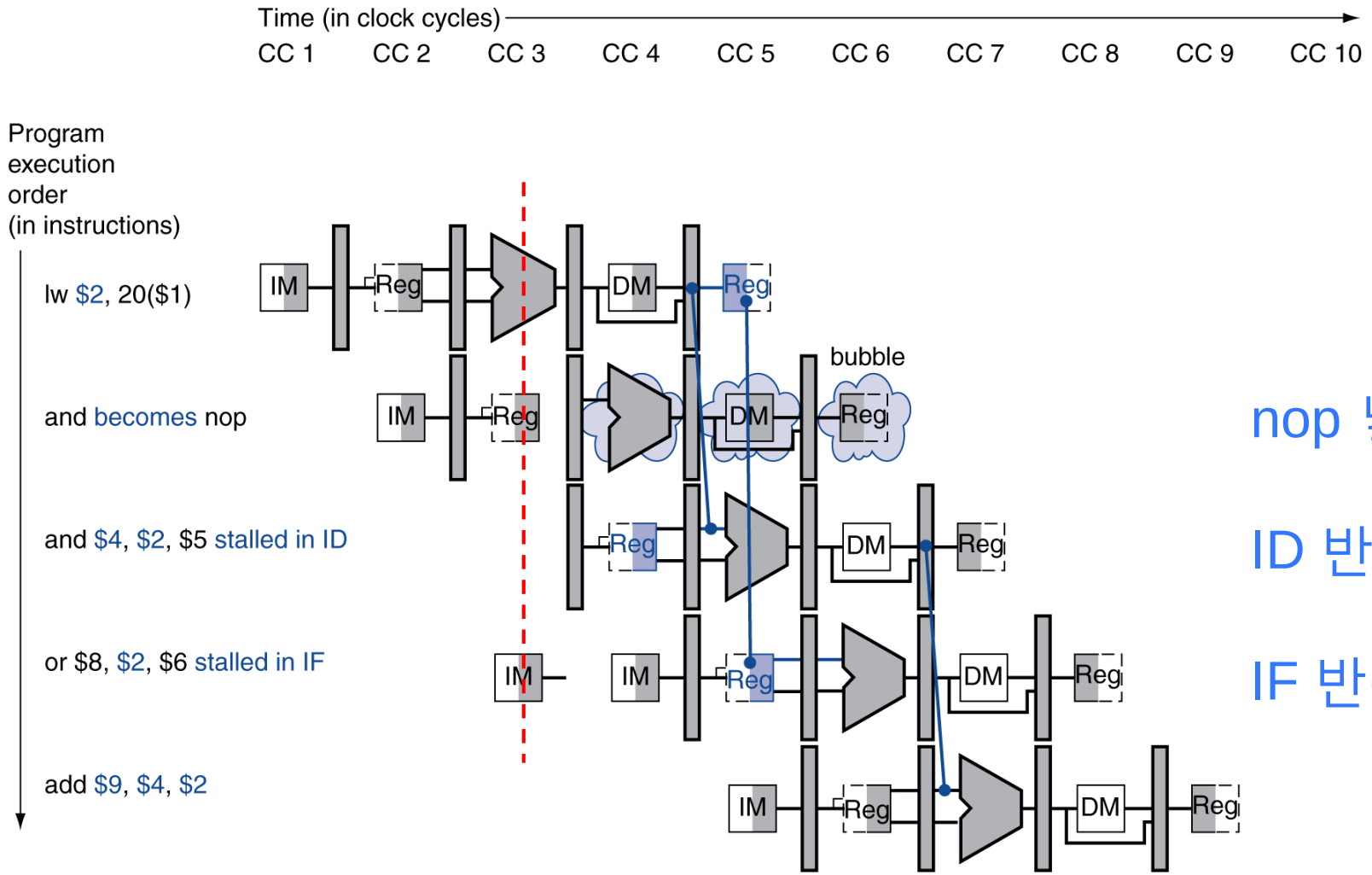
- To insert **nop** (or bubble), at the end of clock, do:



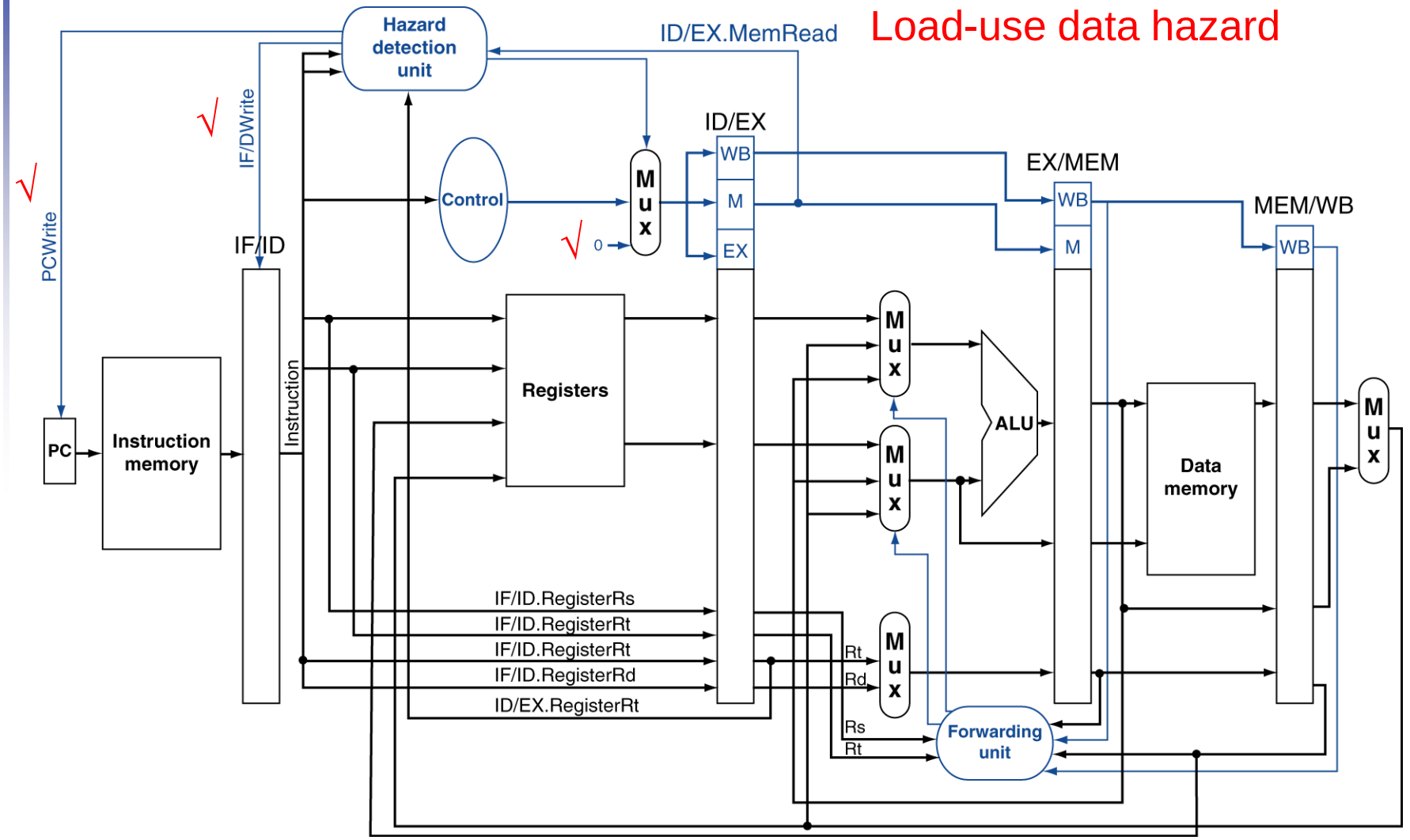
How to Stall the Pipeline (부연)

- Force control values in ID/EX register to 0
  - Disable “RegWrite” and “MemoryWrite”
    - EX executes “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



Datapath with Hazard Detection



- More datapath and control

Data Hazards and Compilers

Software Solution

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls

Software Solution

- Let compiler guarantee no hazards
  - Where do we insert **nop**'s ?

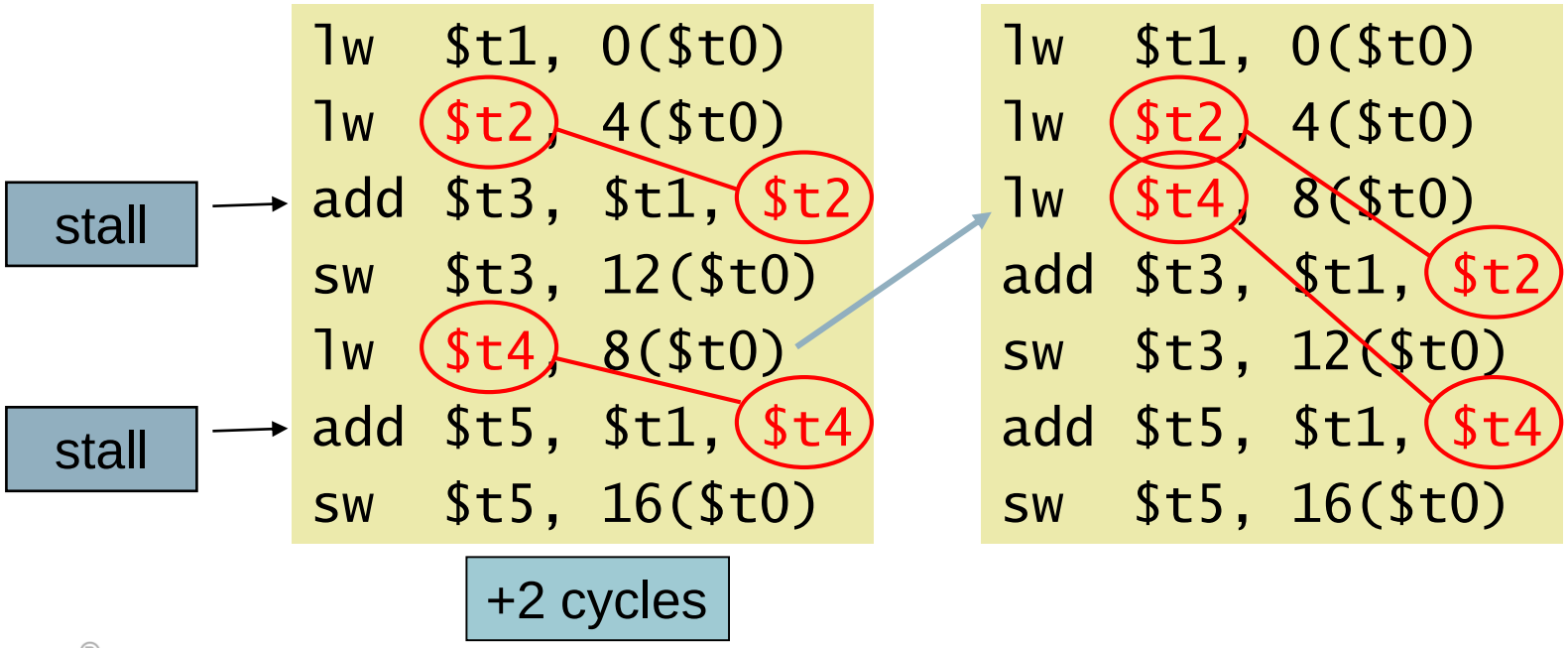
```
sub  $2, $1, $3
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)
```

(No HW forwarding)

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

Code Scheduling to Avoid Stalls

- Reorder code to avoid load-use data hazard
- C code for `A = B + E; C = B + F;`
  - Assume forwarding



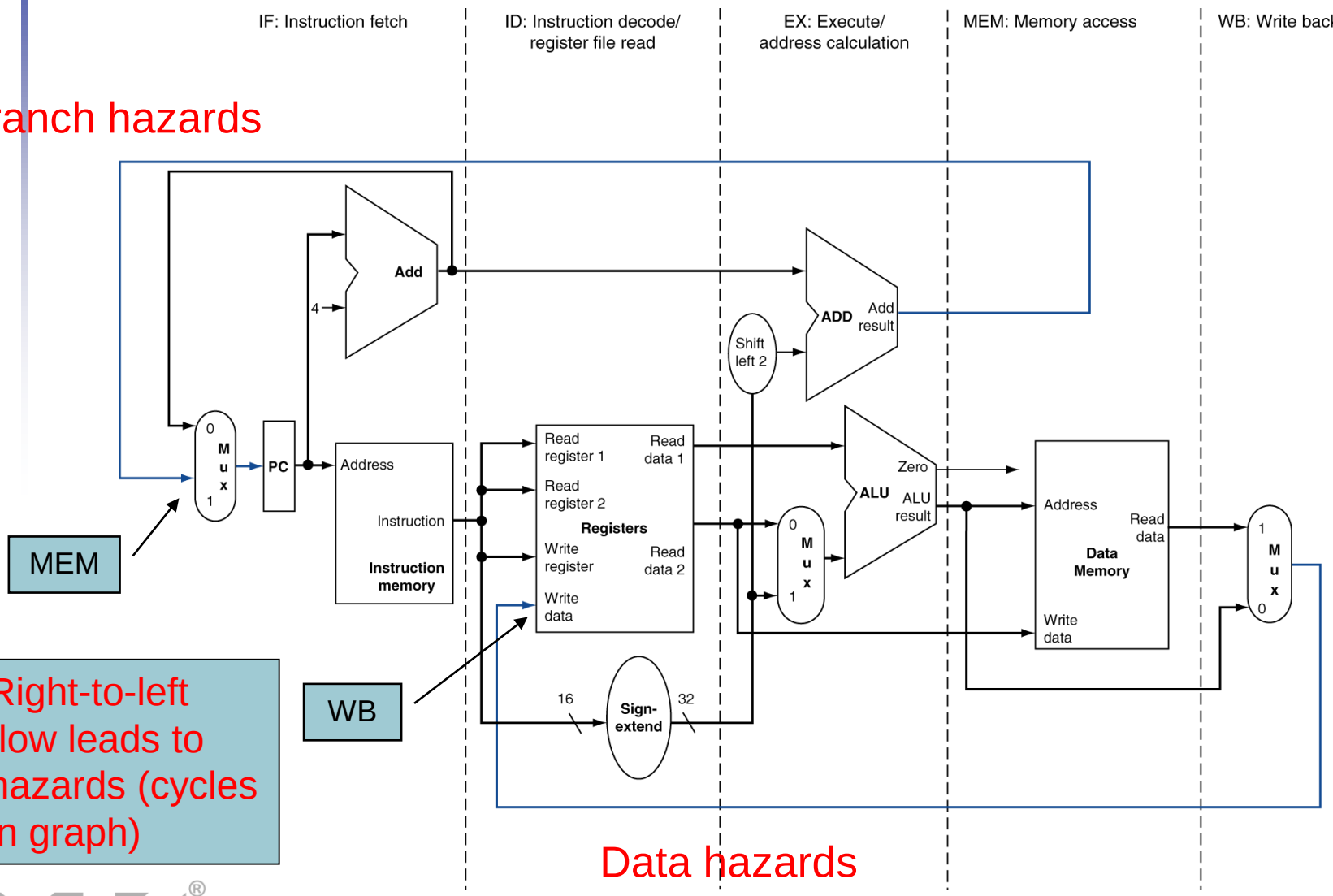
Software Solution

- Compiler can arrange code to avoid load-use stalls
  - Requires knowledge of the pipeline structure
    - What if the implementation changes?
  - Layer violation and compiler dependence
- Hardware can also do it without difficulty
  - Runtime (dynamic) out-of-order execution and in-order completion (Topic 4-3 참고)
  - No compiler dependence

Control (or Branch) Hazards

MIPS Pipelined Datapath (반복)

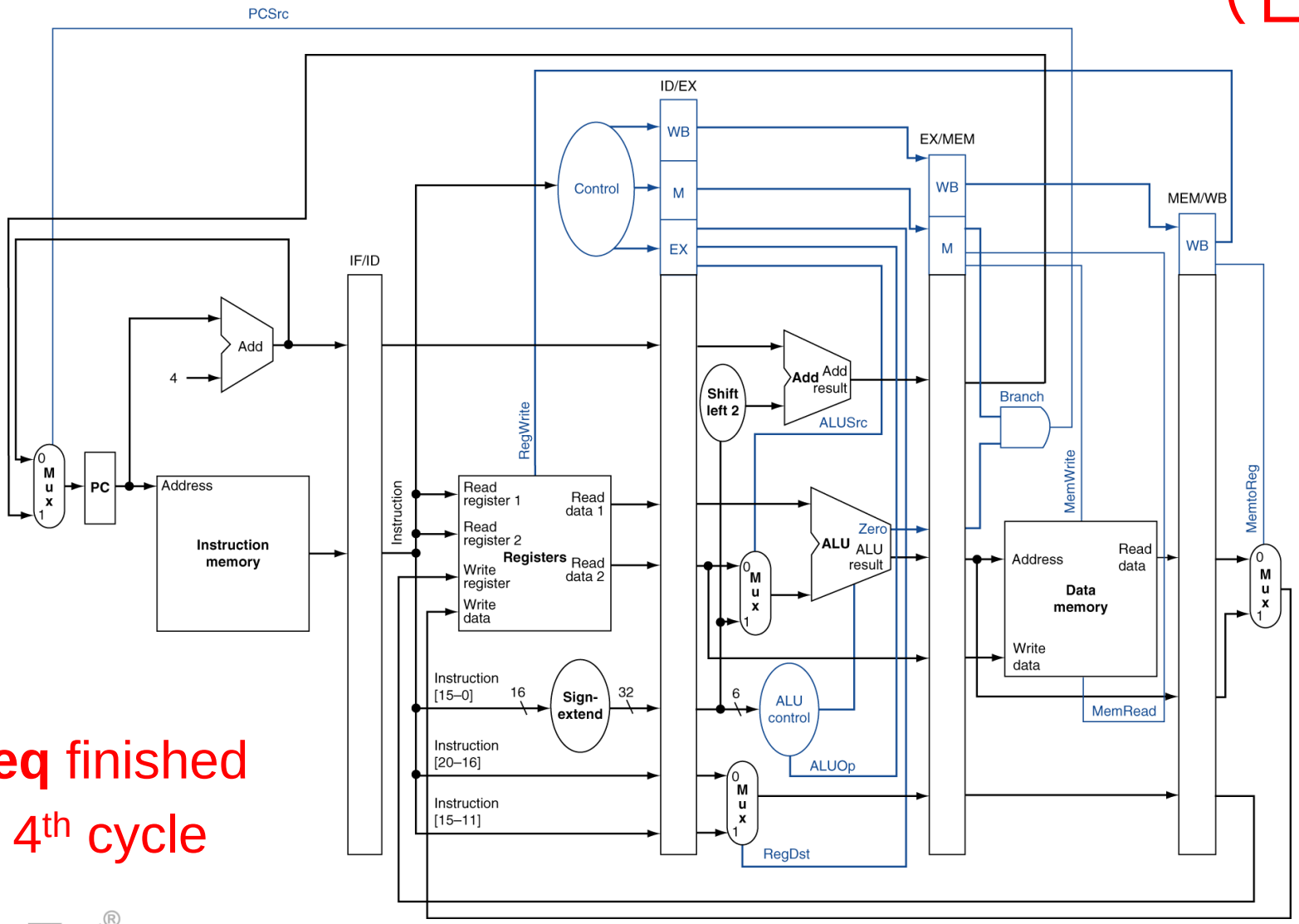
Branch hazards



Right-to-left flow leads to hazards (cycles in graph)

Data hazards

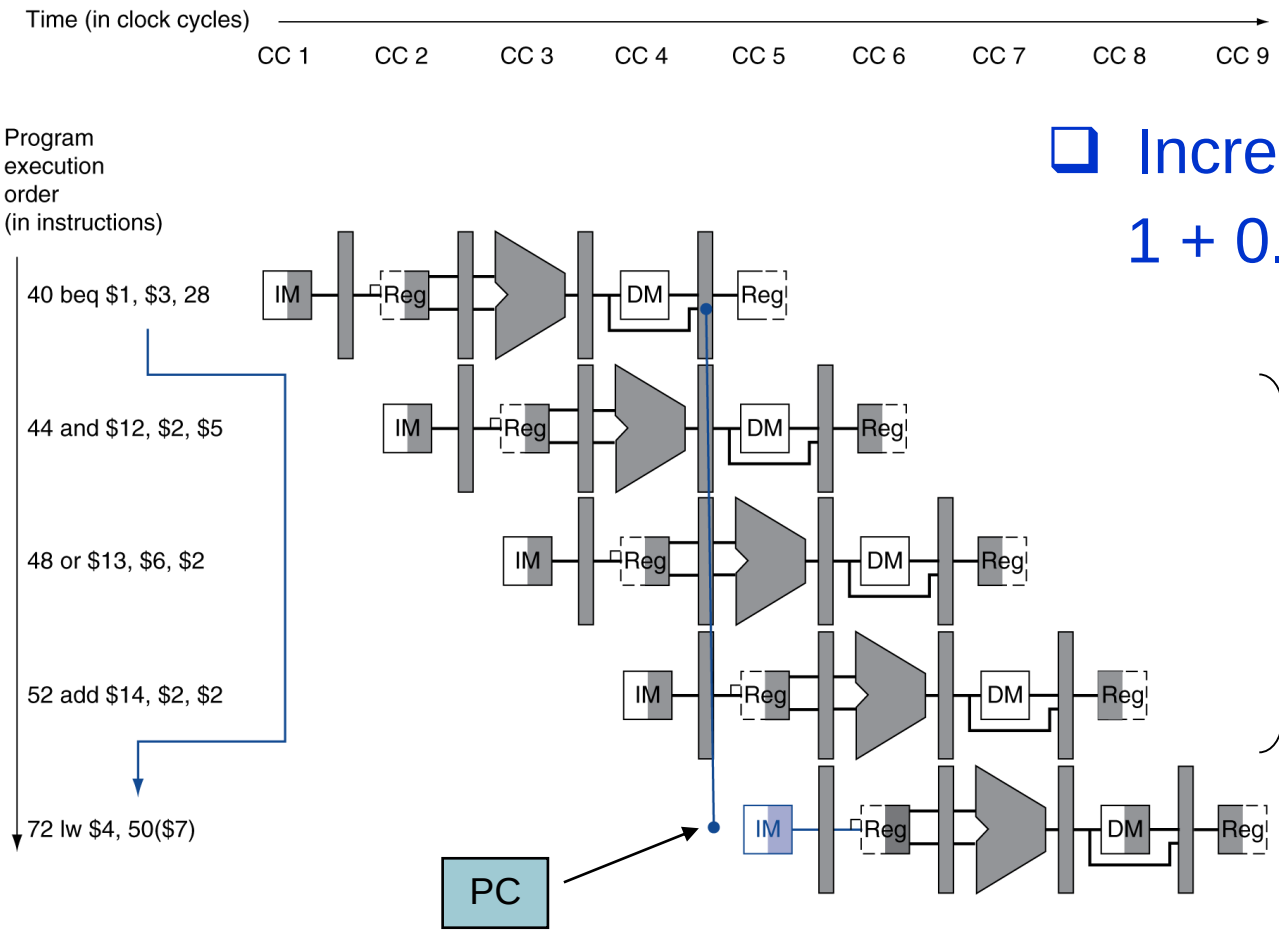
Pipelined Datapath and Control (반복)



beq finished in 4th cycle

Branch Hazards

- If branch outcome determined in MEM



- Increase in CPI?

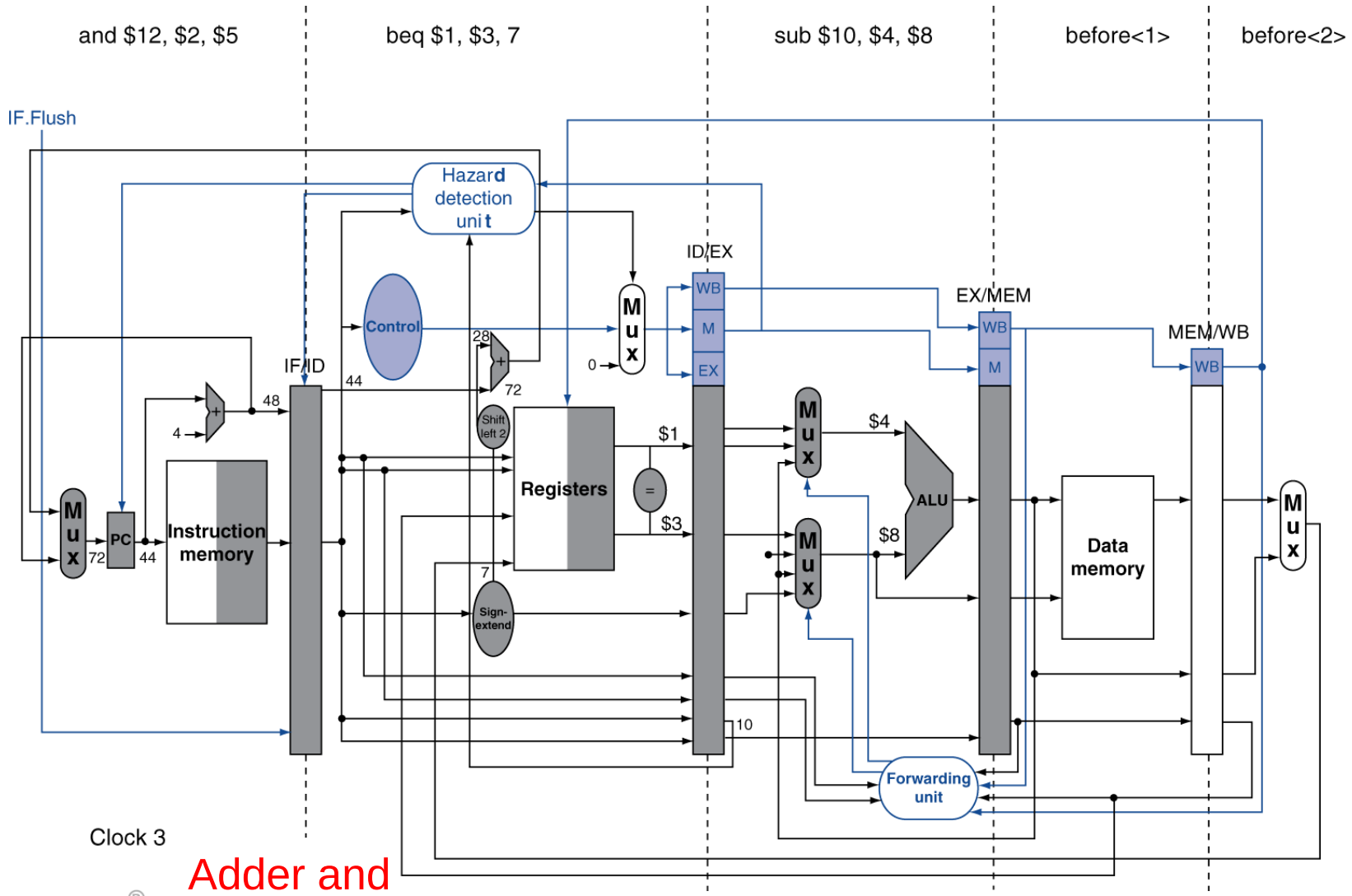
  1 + 0.15 *3 = 1.45

Flush these instructions (Set control values to 0)

Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
- Branch delay: 3 cycles
  - Change datapath to finish **beq/bne** in ID stage
    - Branch delay: 1 cycle

Moving Branch to ID Stage



Adder and comparator

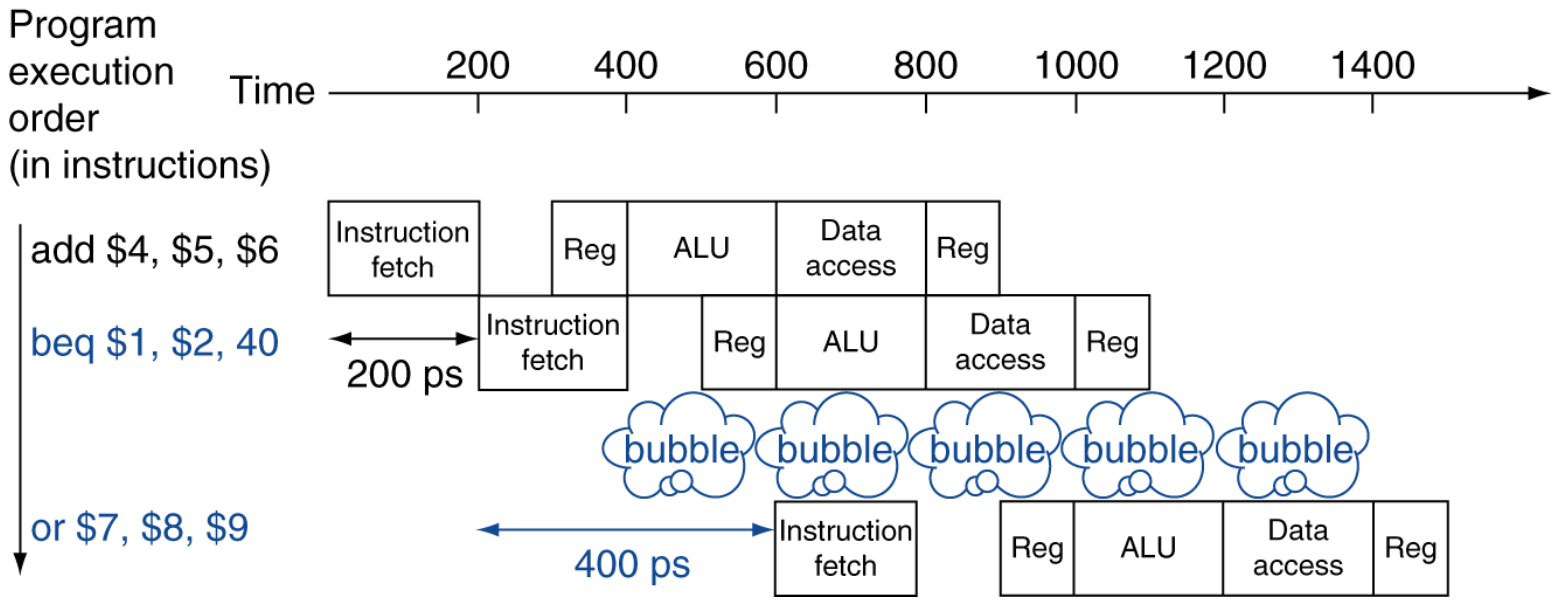
Reducing “Branch Delay”

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
    - Equality check (**beq/bne**): bit by bit comparison
      - † Faster than subtract (**blt** → **slt** and **beq**)
- ISA design with pipelining in mind

Stall on Branch

- Wait until branch outcome determined before fetching next instruction
  - Increase in CPI?

    1 + 0.15 * 1 = 1.15

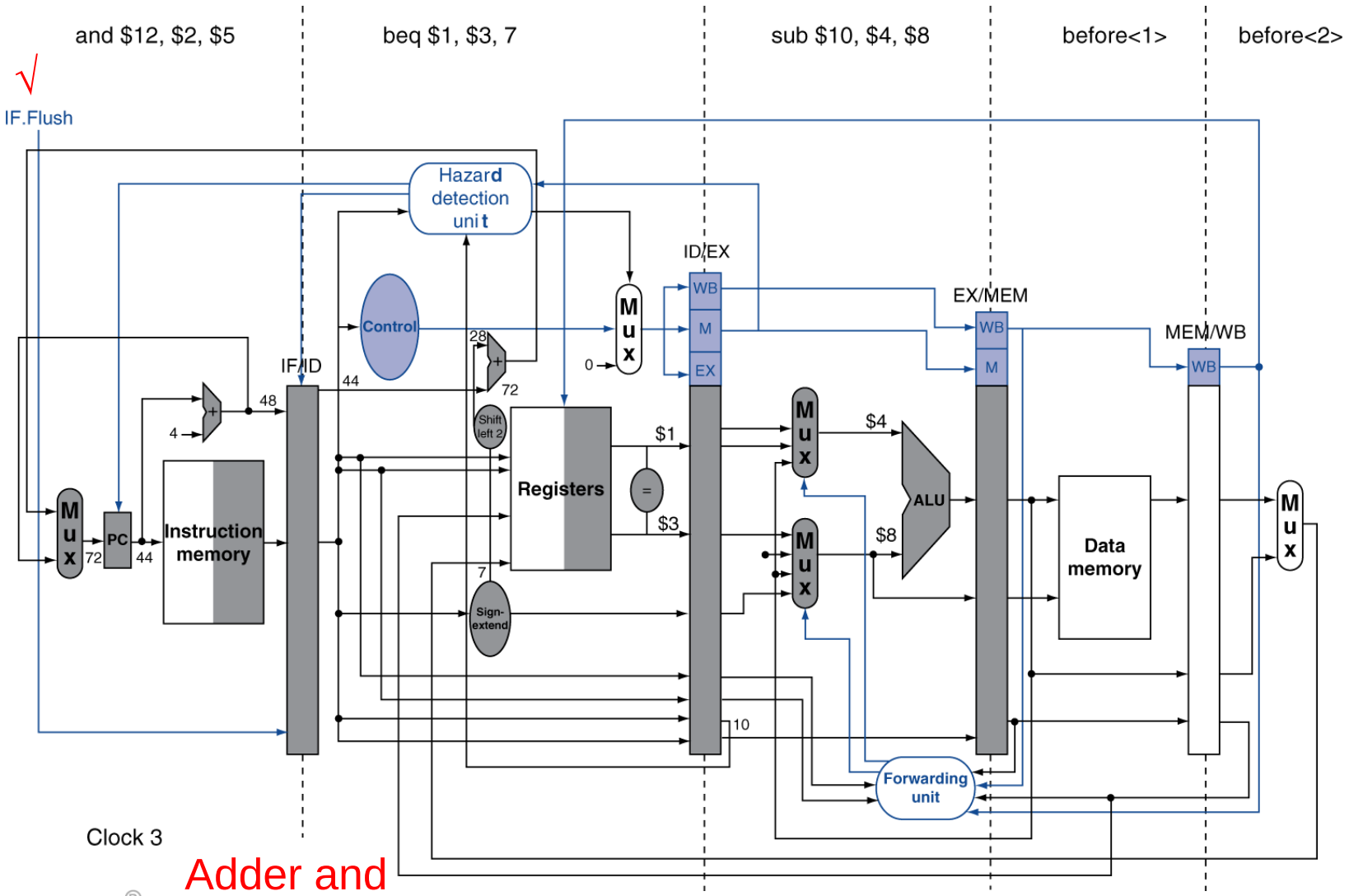


Moving Branch to ID Stage

- Evaluate branch decision in ID
  - Use IF.Flush control signal to insert NOP
    - **sll** $0, $0, 0 (all 0s; opcode & function field are 0s)
- Side effect: more forwarding and data hazards
  - Additional forwarding to register comparator
    - From EX/MEM, MEM/WB registers
  - Additional stalls

- Think about ARM conditional instruction (skip)

Moving Branch to ID Stage (반복)

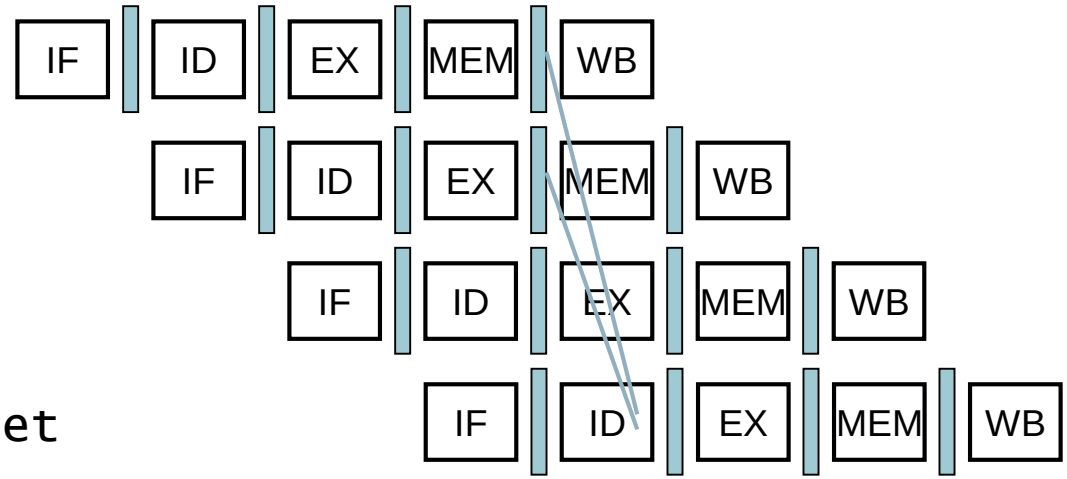


Adder and comparator

Additional Data Hazards

- If a comparison register is a destination of immediately preceding ALU instruction
  - Need 1 stall cycle

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

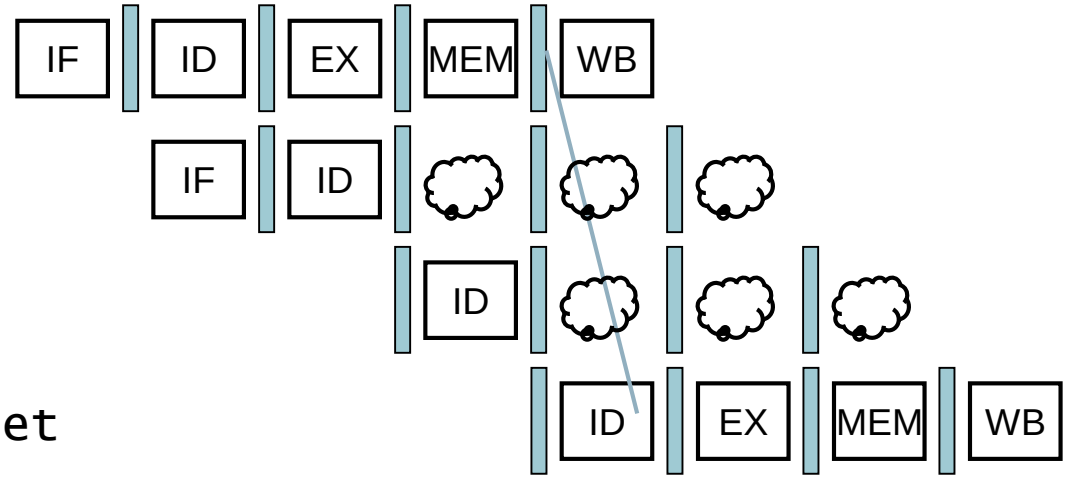


Additional Data Hazards

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

그러나 control hazard 를 줄이는 이득이 더 큼

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



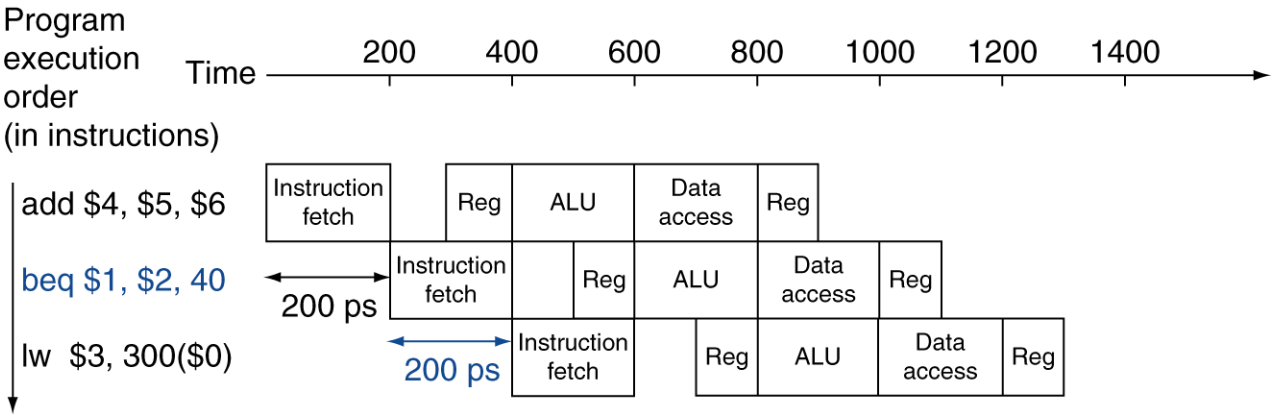
Branch Prediction

Branch Prediction

- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken (predict not taken)
  - Fetch instruction after branch, with no delay
    - Prediction accuracy: 40%
  - Cannot use “predict taken” in our design

MIPS with Predict Not Taken

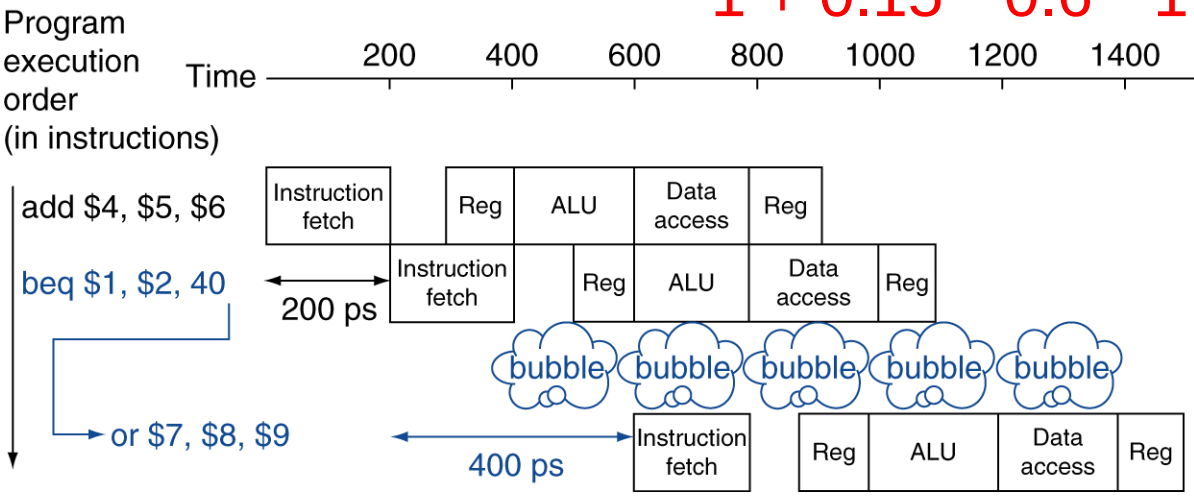
Prediction correct



- Increase in CPI?

$1 + 0.15 * 0.6 * 1 = 1.09$

Prediction incorrect



More–Realistic Branch prediction

- ❖ Advanced pipelining 에 대해 미리 공부 (Topic 4-3 참고)
  - 상세한 방법보다는 개념만 이해하면 됨

- ❑ Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- ❑ Branch prediction is important in deep pipeline

More-Realistic Branch Prediction

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

Dynamic Branch Prediction

- Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
- Branch target buffer
  - Cache of target addresses
    - If hit and instruction is branch predicted taken, can fetch target immediately

Dynamic Branch Prediction

- To execute a branch (speculative execution)
  - Check table, expect the same outcome
  - Start fetching from fall-through or target
  - If wrong, flush pipeline and flip prediction

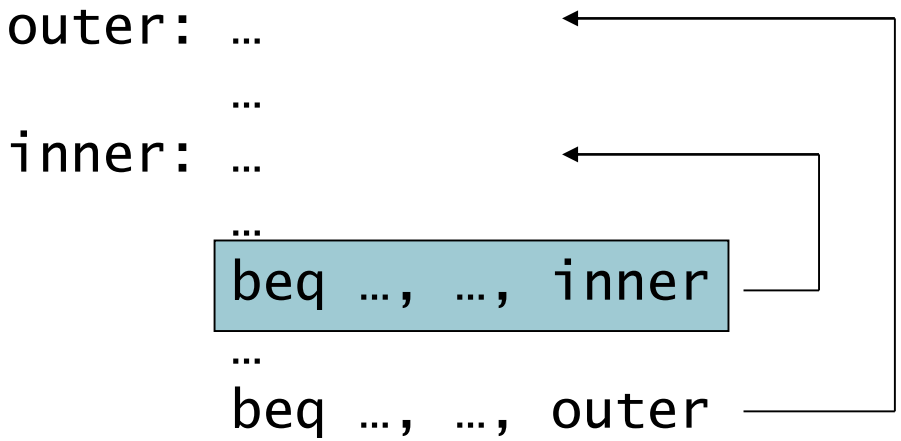
Hardware data structure

Addresses of branch instructions	Branch taken or untaken	Branch target

- Hardware overhead vs. performance
  - Prediction accuracy over 90% in desktop processors

1-Bit Predictor: Shortcoming

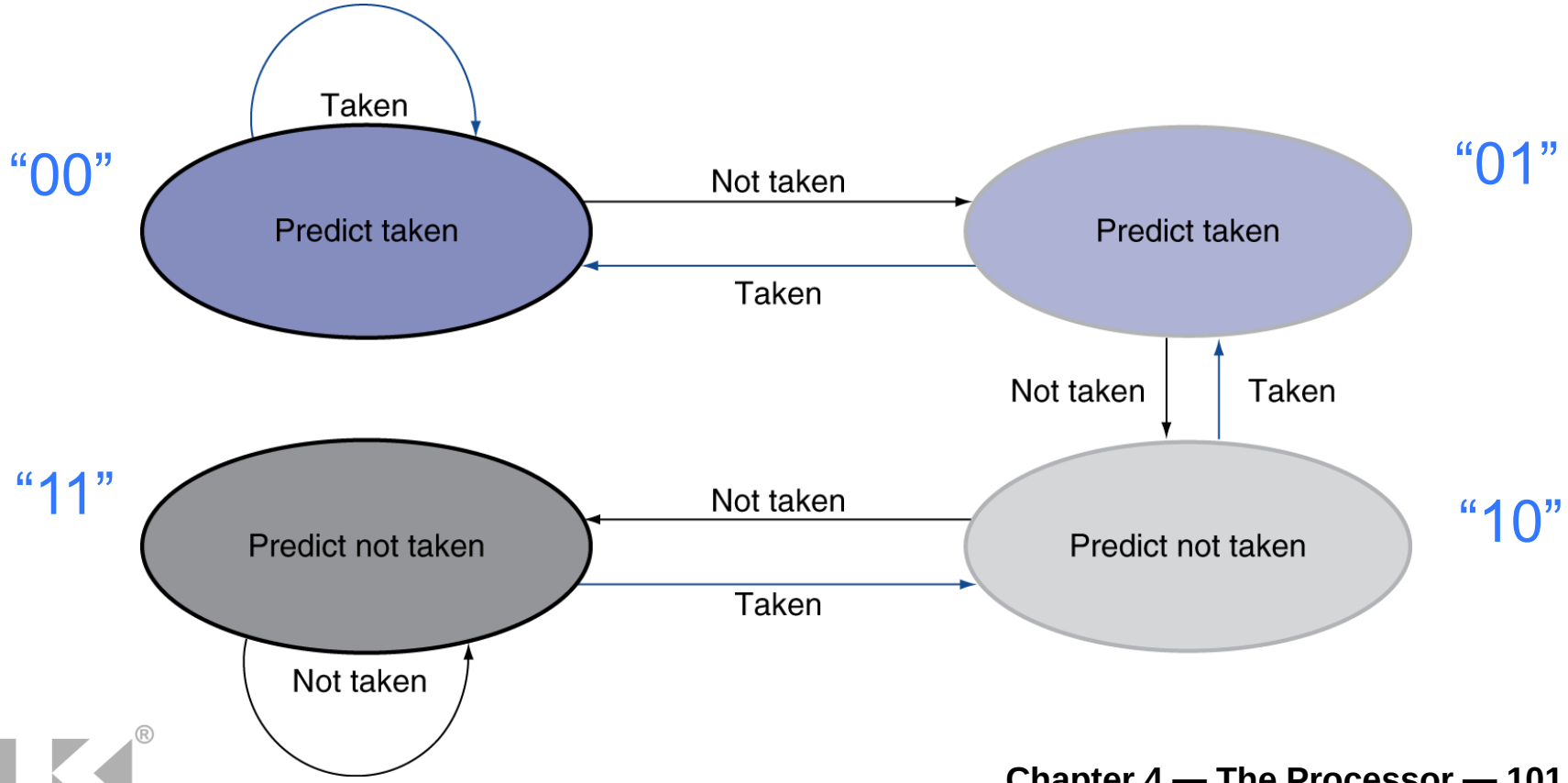
- Inner loop branches mispredicted twice!



- 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
  - Inner loop branches mispredicted only once!



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 (RISC vs. CISC)

Homework #12 (see Class Homepage)

1) Write a report summarizing the materials discussed in Topic 4-2 (이번 주 수업 내용)

** 문장으로 써도 좋고 파워포인트 형태의 개조식 정리도 좋음

2) Solve Chapter 4 exercises 4.8, 4.12, 4.13.2, 4.13.4

- Due: see Blackboard
  - Submit electronically to Blackboard

Section 4.10

Introduction to Advanced Pipelining

(From 5-Stage MIPS to Powerful MIPS for PC)