Computer Architecture

Chapter 4B. A Pipelined Processor

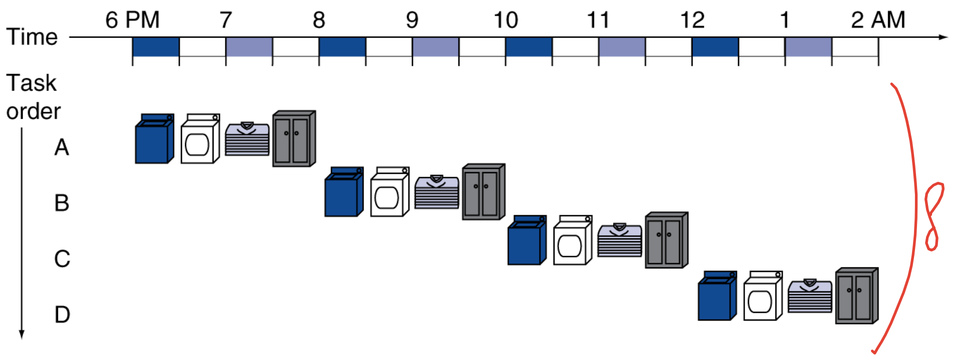
Hyuk-Jun Lee, PhD

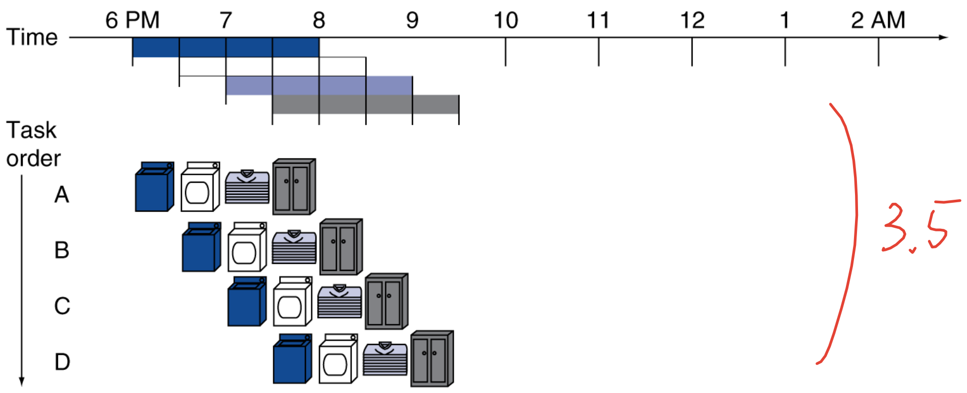
Dept. of Computer Science and Engineering
Sogang University
Seoul, Korea

Email: hyukjunl@sogang.ac.kr

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance





- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop: 새로운 작업을 시작하는 시간
  - Speedup = 2n/0.5n + 1.5 ≈ 4 = number of stages

MIPS Pipeline

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

Instruction 마다 거치는 단계가 다름

ex) load : 5단계 모두

ALU : 4번은 안하지

Pipeline Performance

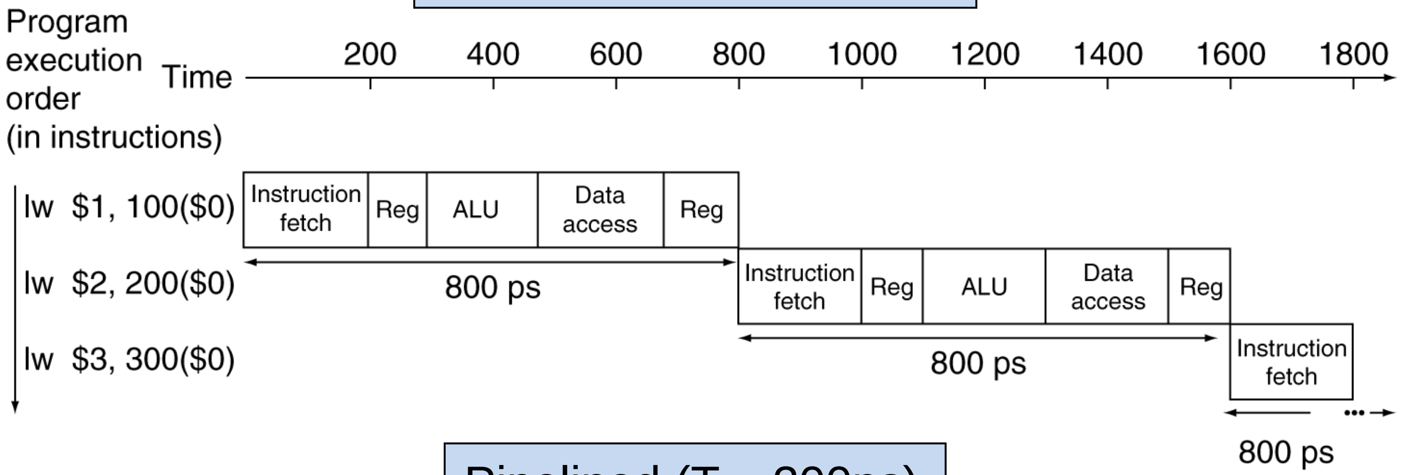
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

'instruction 에따른 시간

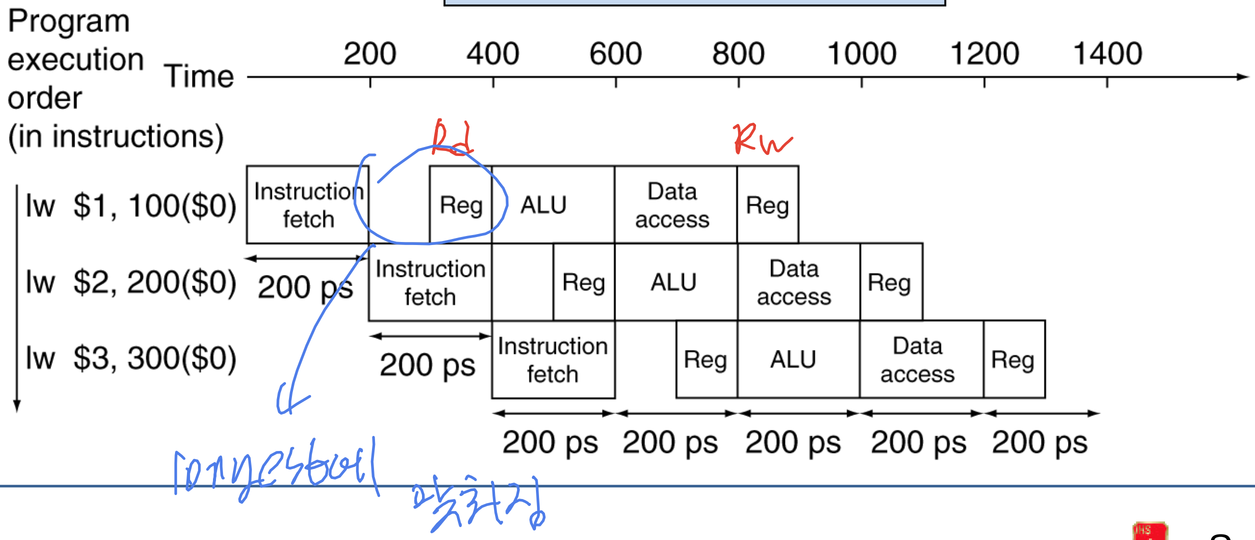
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{\text{pipelined}}$
    $$= \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

T_c = clock cycle time

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

하나 실행이 줄어드는건 아님

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 horrible
  - 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

Hazards

- Situations that prevent starting the next instruction in the next cycle 매 사이클마다 새 instruction을 실행 못할수도.
- Structure hazards
  - A required resource is busy
- Data hazard (data dependency에 의해 발생)
  - Need to wait for previous instruction to complete its data read/write
- Control hazard (control dependency에 의해 발생)
  - Deciding on control action depends on previous instruction

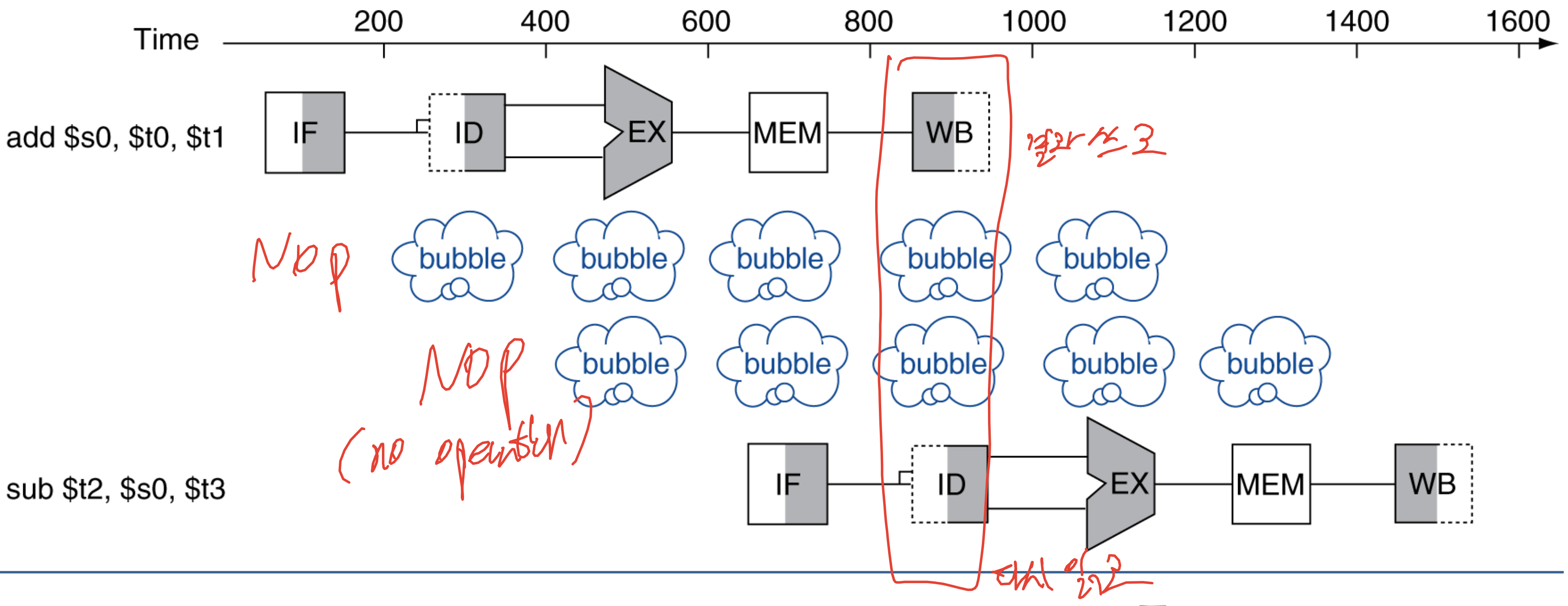
Structure Hazards

- 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"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add **$s0**, $t0, $t1
    sub $t2, **$s0**, $t3

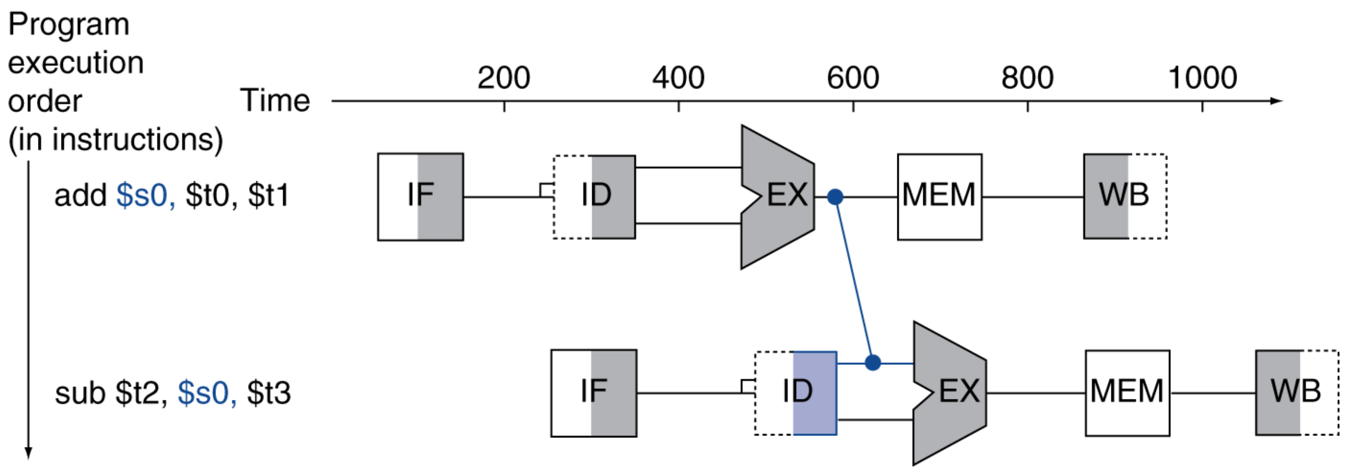
```
– add   $s0, $t0, $t1
  sub   $t2, $s0, $t3
```



Forwarding (aka Bypassing)

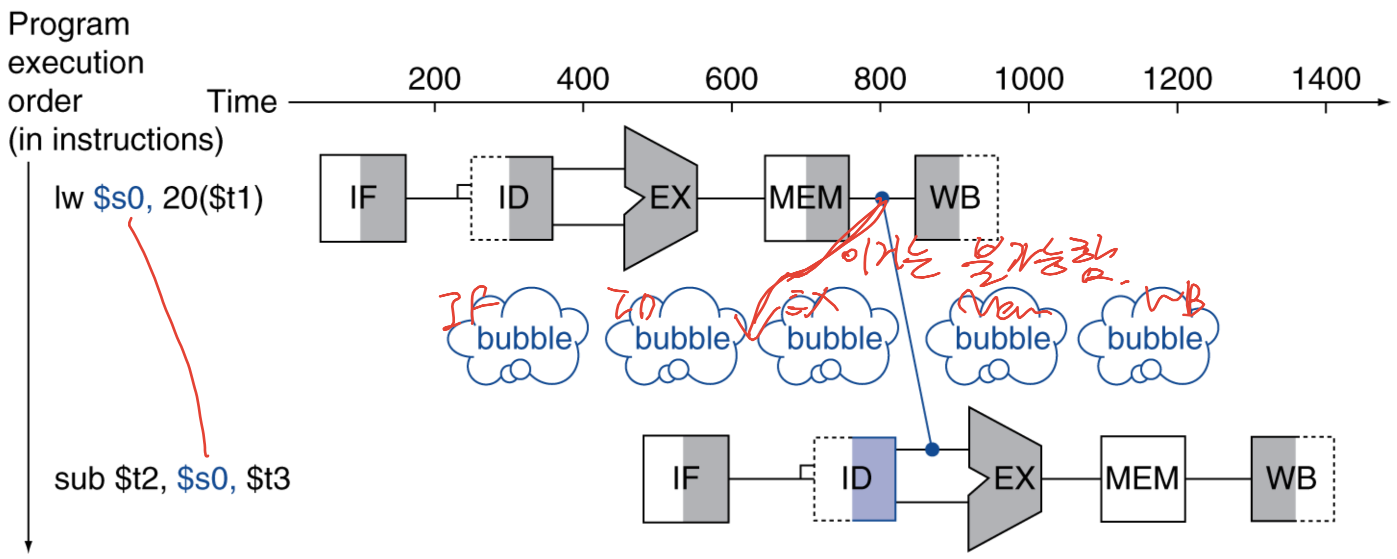
[illegible]

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



Load-Use Data Hazard

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



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;`

memory의 서는 한 번 찾아했고 득이가

```
B      lw   $t1, 0($t0)
E      lw   $t2, 4($t0)
stall  B+E → add  $t3, $t1, $t2
A      sw   $t3, 12($t0)
F      lw   $t4, 8($t0)
stall  B+F → add  $t5, $t1, $t4
C      sw   $t5, 16($t0)
```

13 cycles =7+4+2

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

11 cycles =7+4

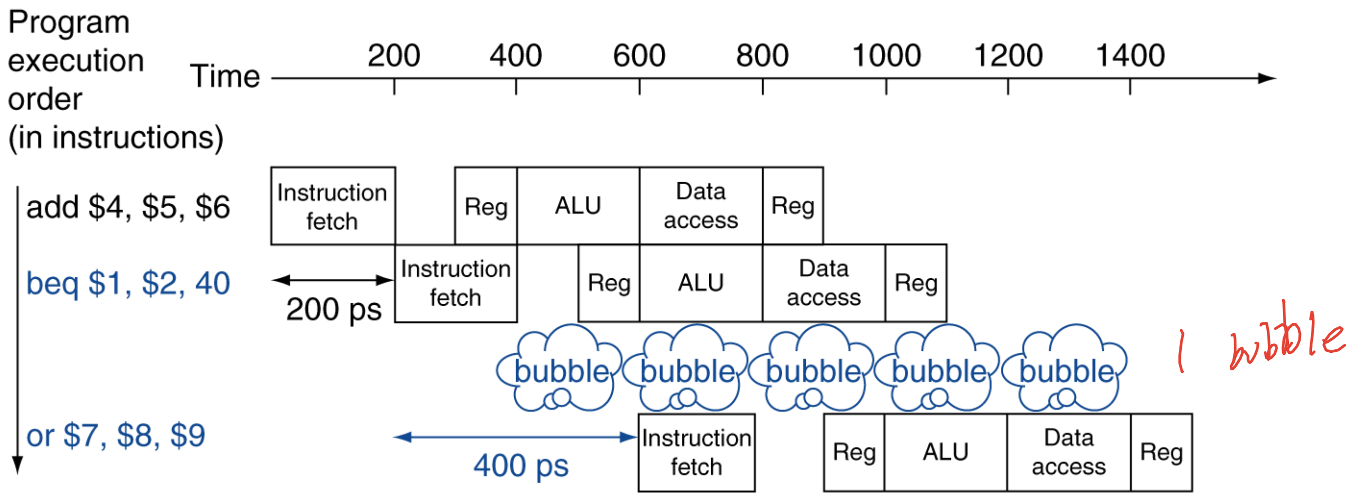
BEQ $1, $2, OFFSET
=> $1 - $2
=> PC+4+4*offset

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

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



Branch Prediction

예측, 예언

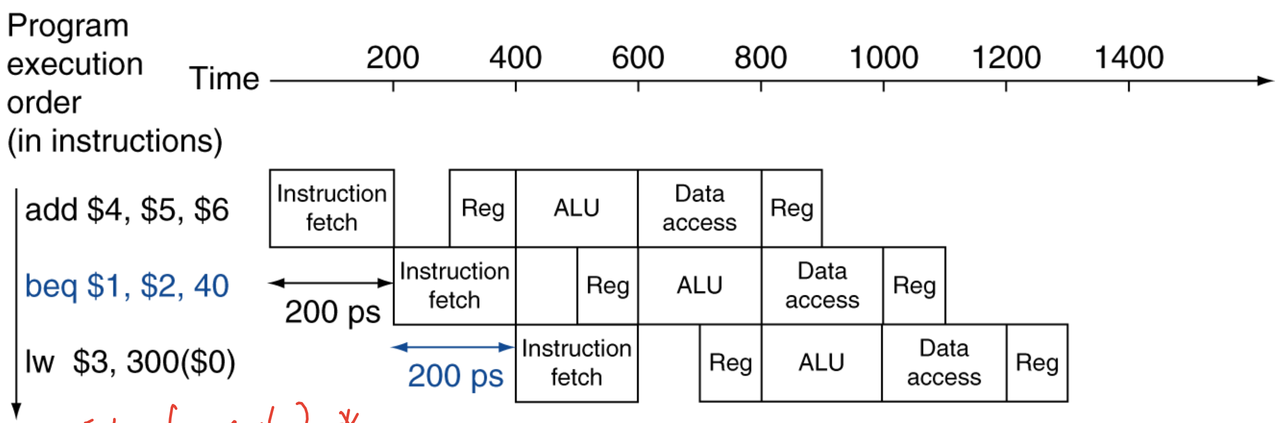
해결책

- 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

그냥 PC+4 라고 생각하다가 틀리면 처리를 해서 → 1 cycle 페널티 먹고 올바른 방향으로.

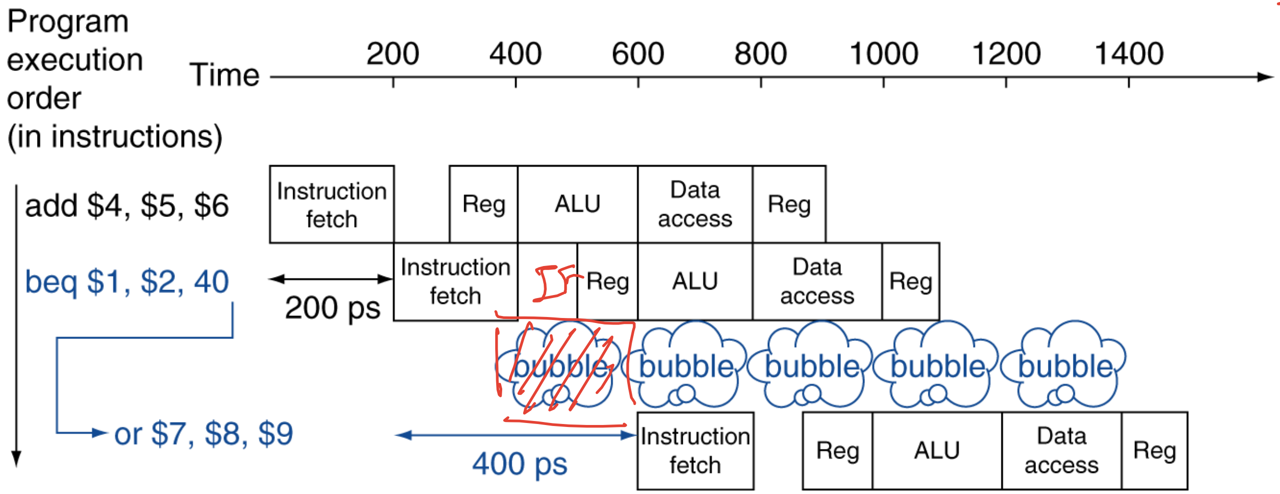
MIPS with Predict Not Taken

Prediction correct



CpI = 1 + 0.2 (20% branch) * 0.5 (50% of branch taken) * 1 = 1.1 (10% 증가)

Prediction incorrect

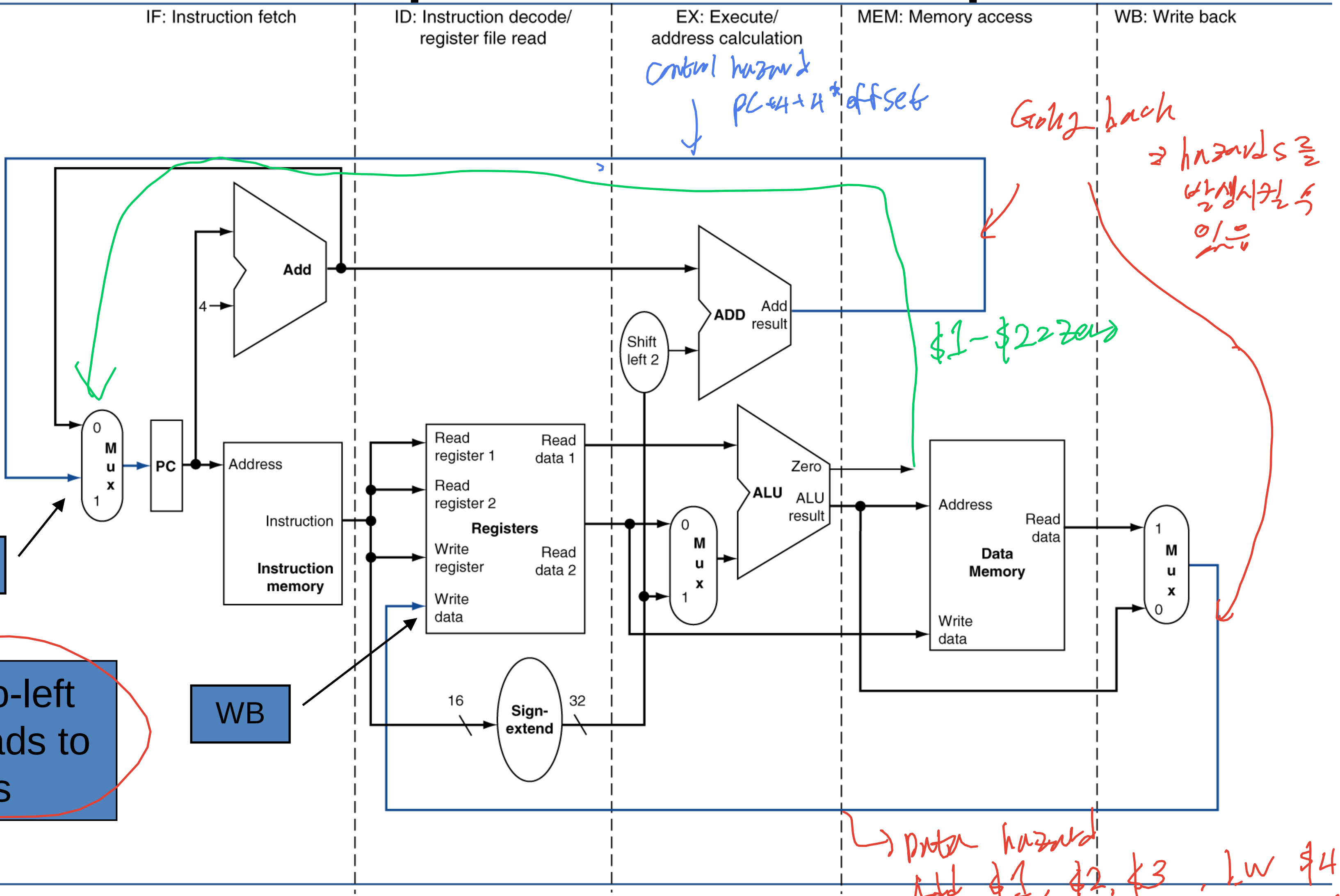


Pipeline Summary

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

MIPS Pipelined Datapath



control hazard

PC+4+4*offset

Going back

→ hazards를 발생시킬 수 있음

$1-$2 2zero?

→ Data hazard

add $1, $2, $3 , lw $4, 100($5)

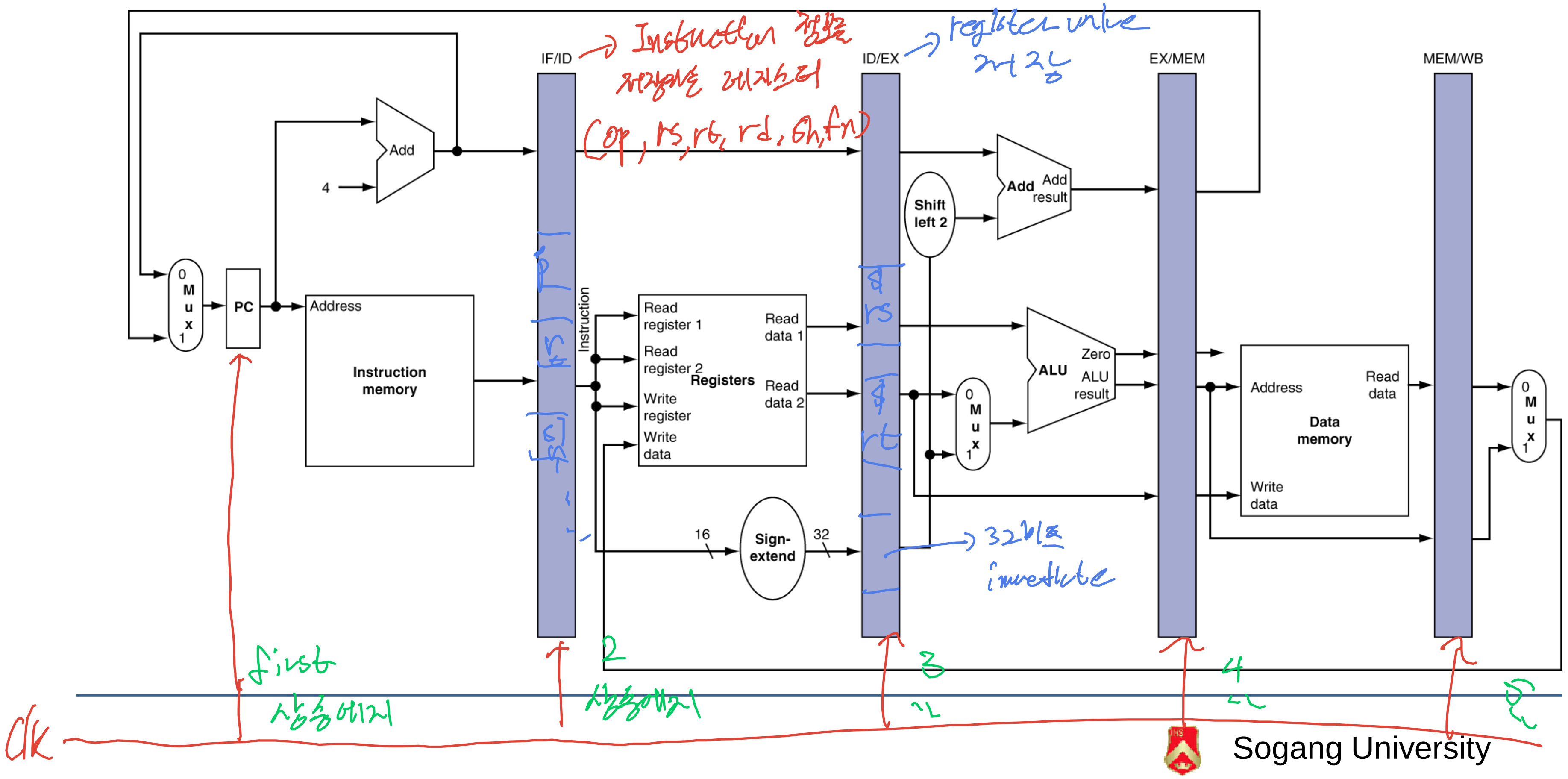
MEM

WB

Right-to-left flow leads to hazards

Pipeline registers

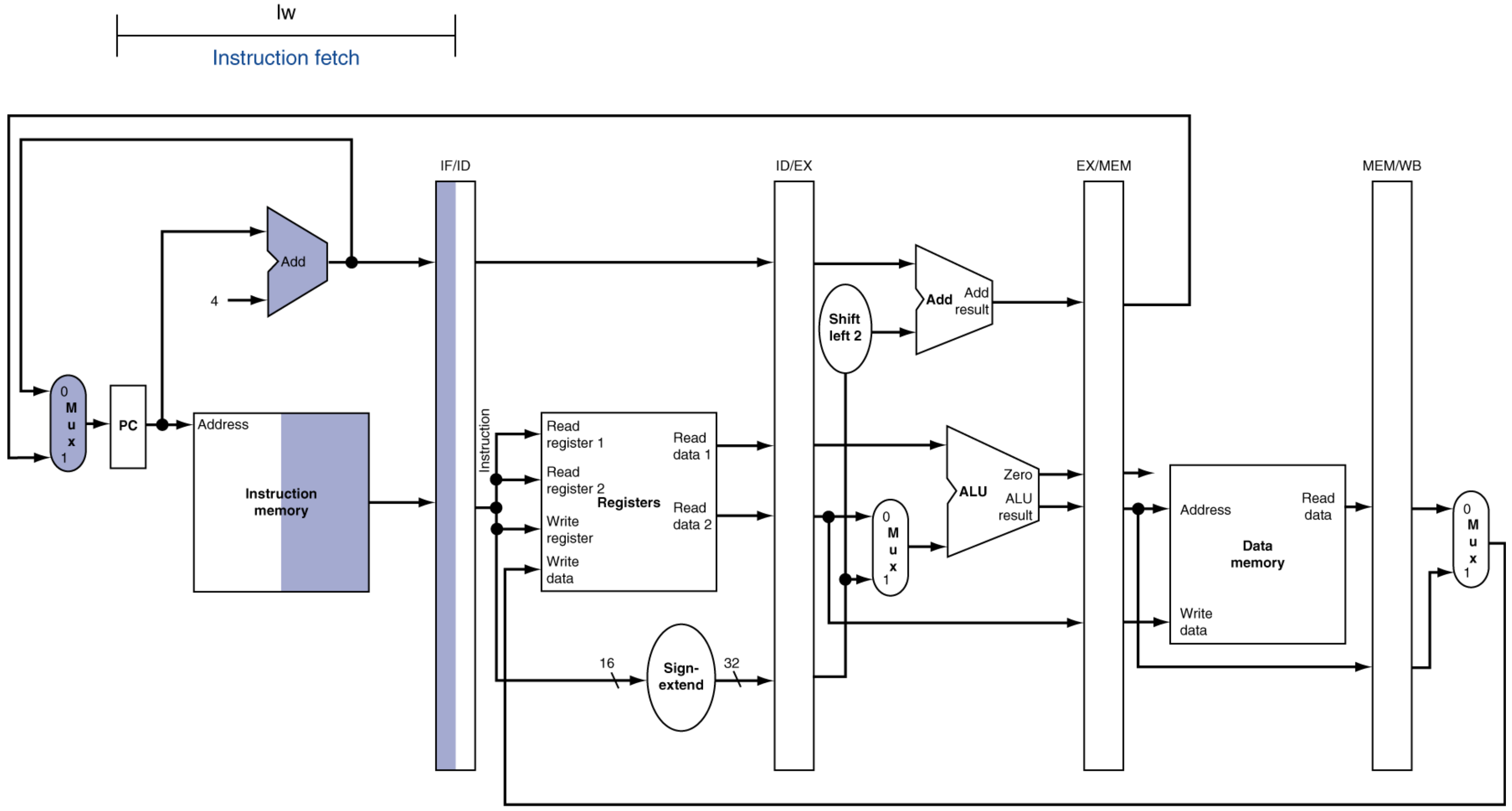
- Need registers between stages
  - To hold information produced in previous cycle



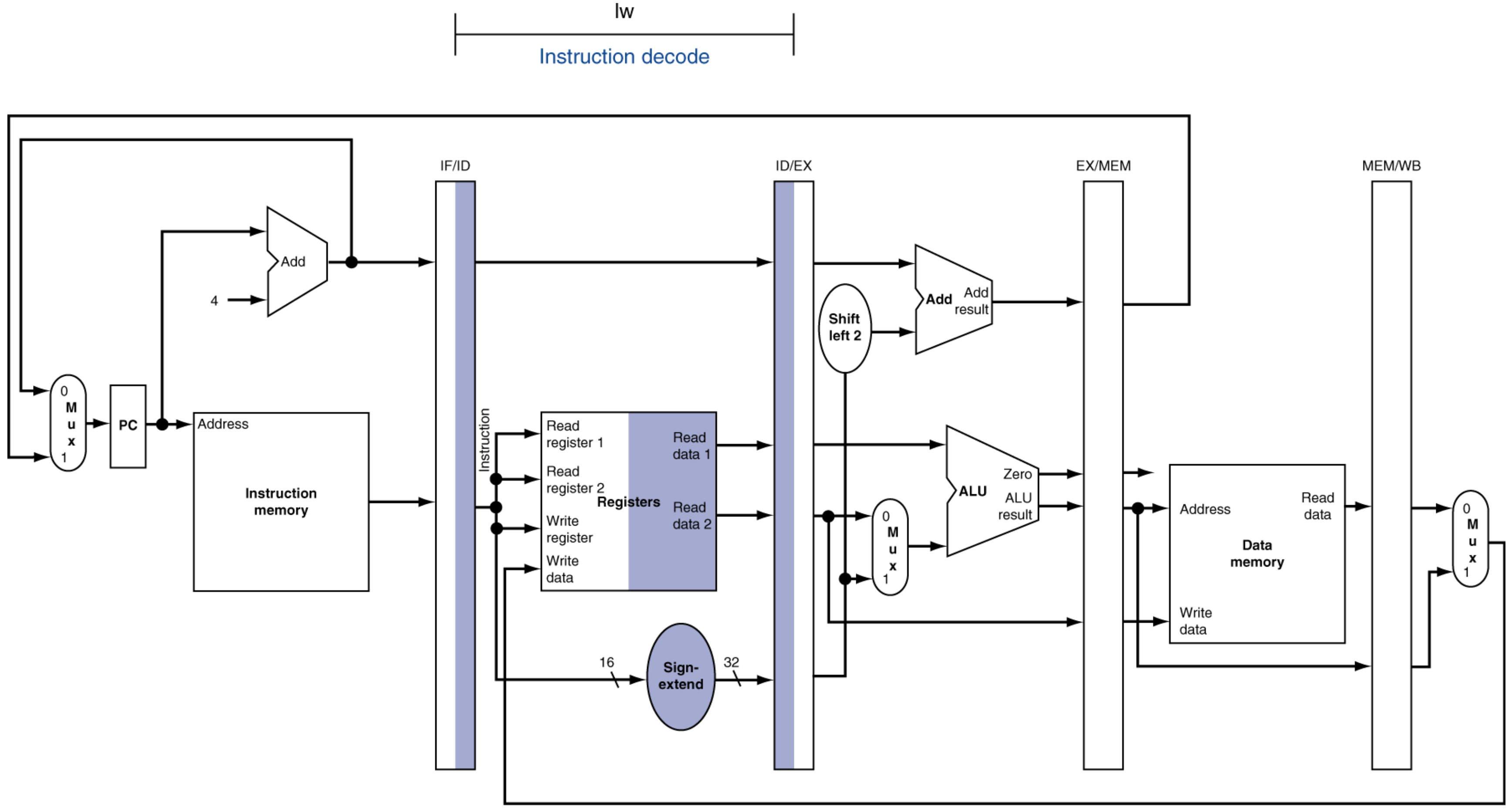
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store

IF for Load, Store, …

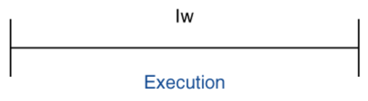


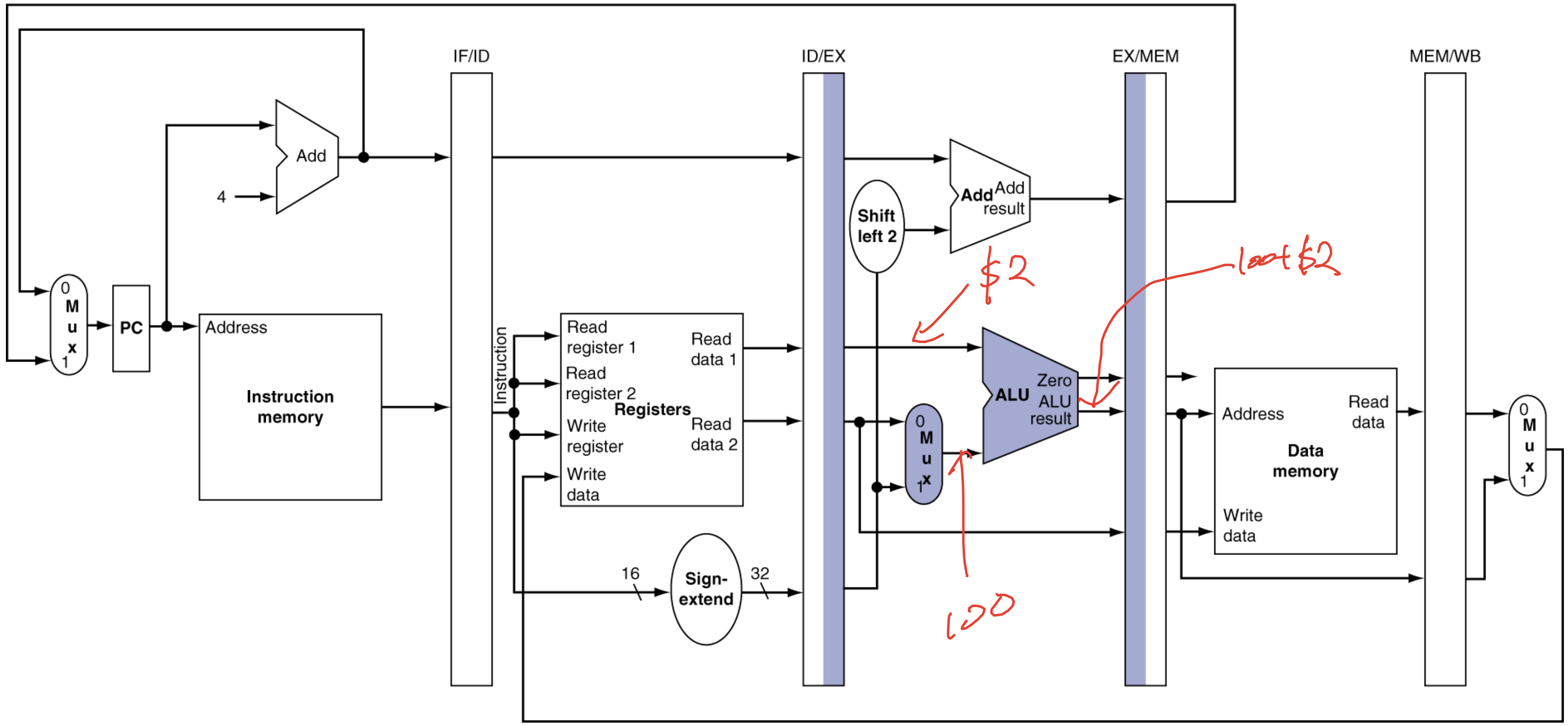
ID for Load, Store, …



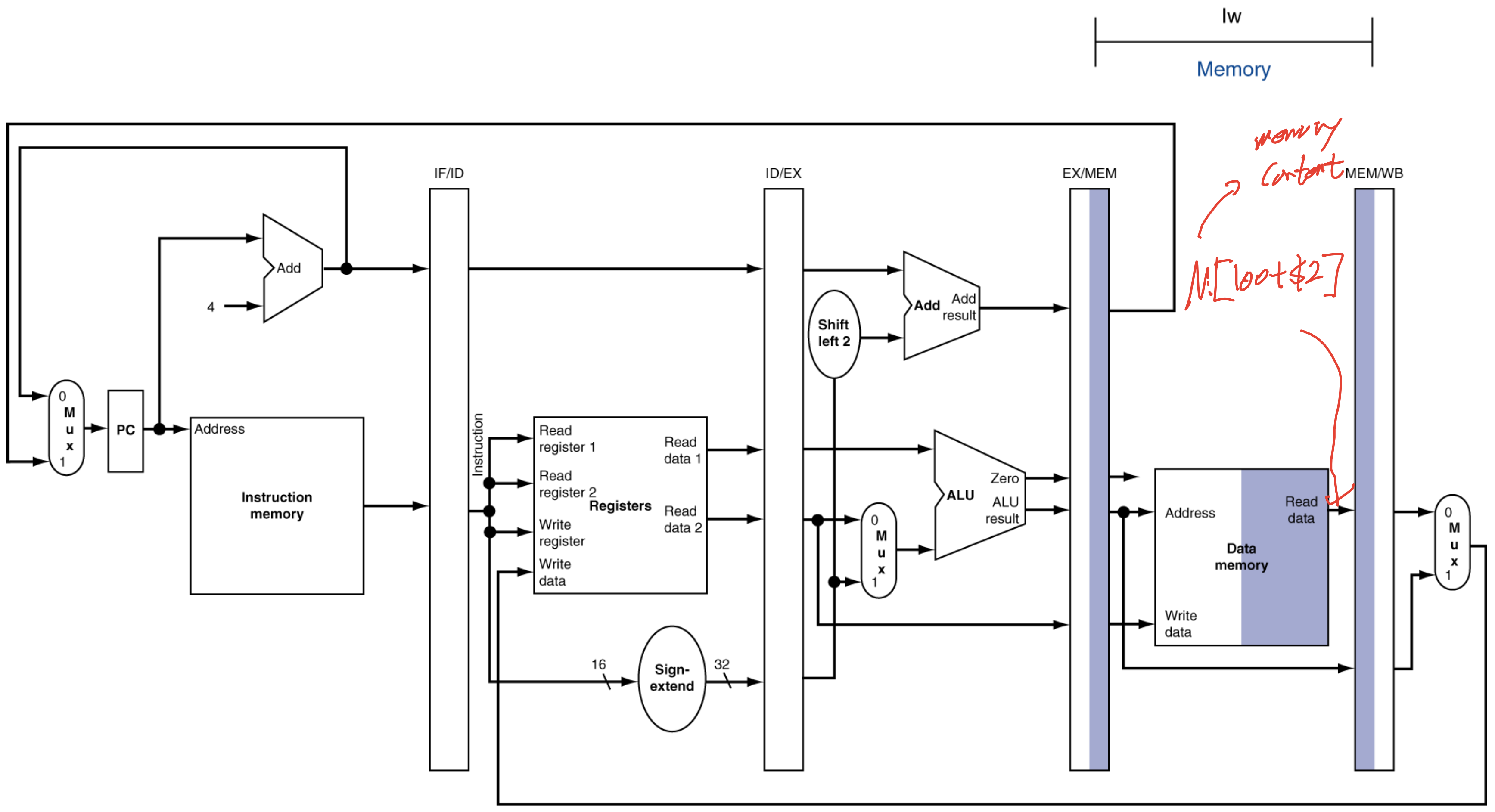
EX for Load

ex)
Lw $1, 100($2)



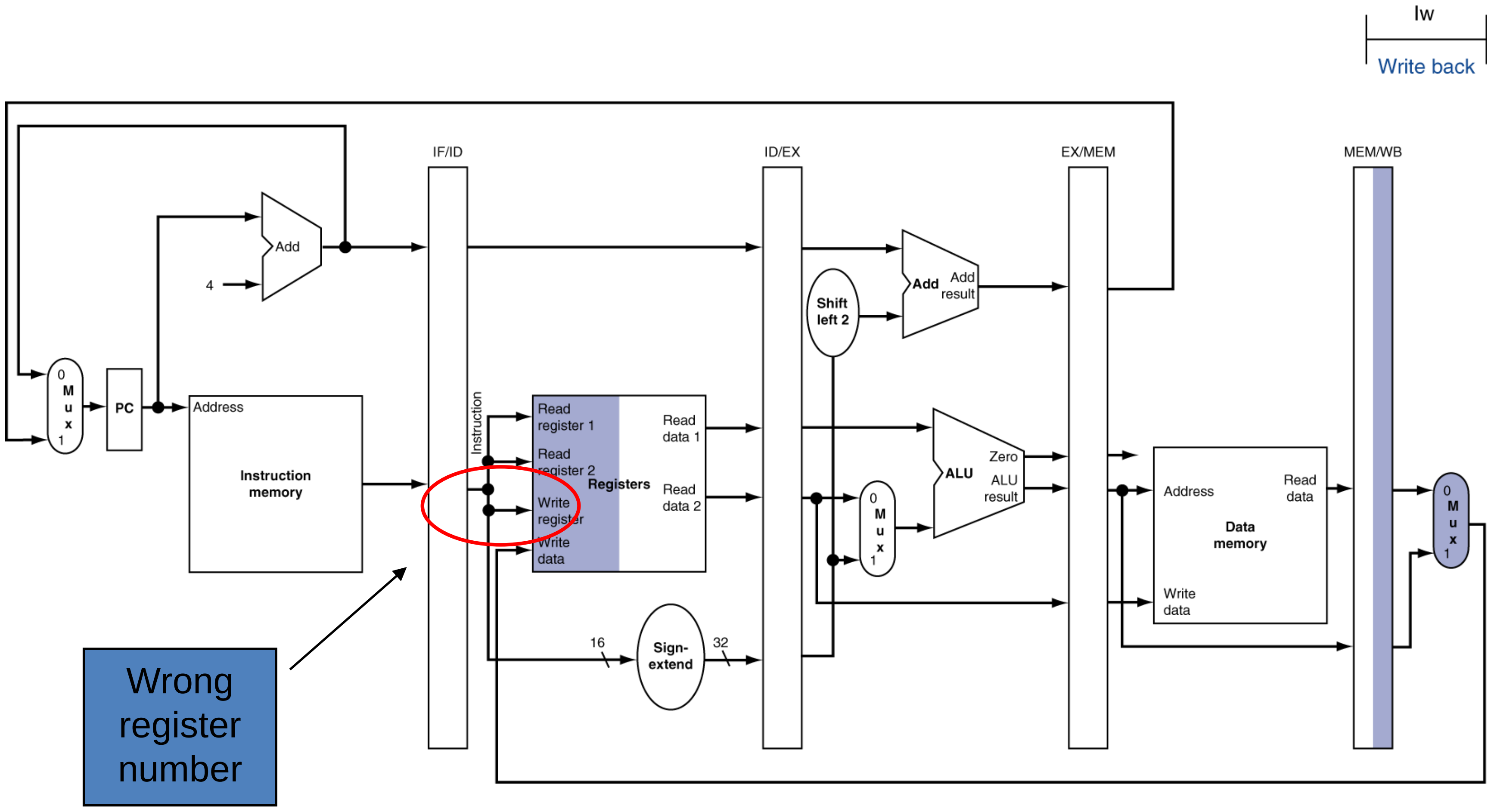


MEM for Load



Sogang University

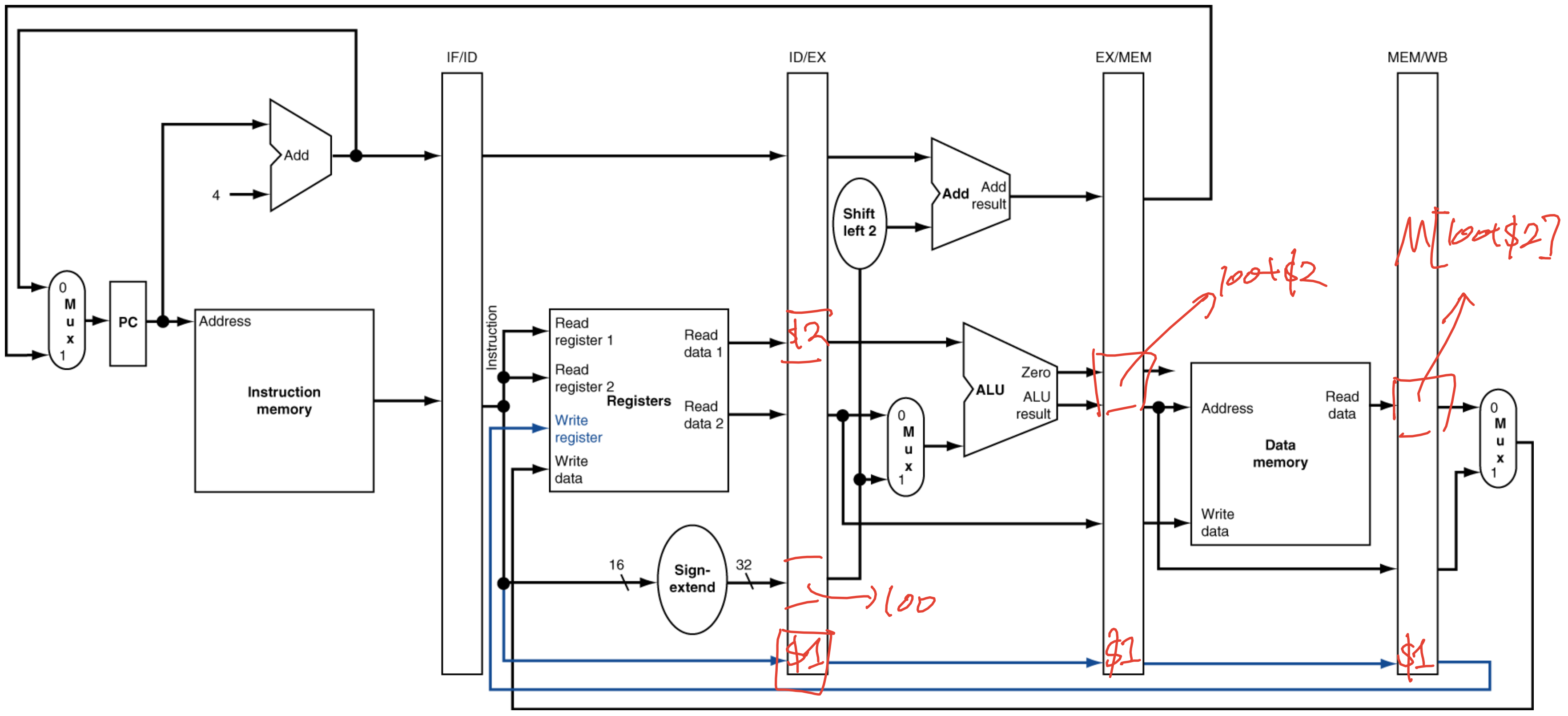
WB for Load



lw
Write back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
Add
Add result
Shift left 2
0
M
u
x
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
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
Wrong register number

lw $1, 100($2)

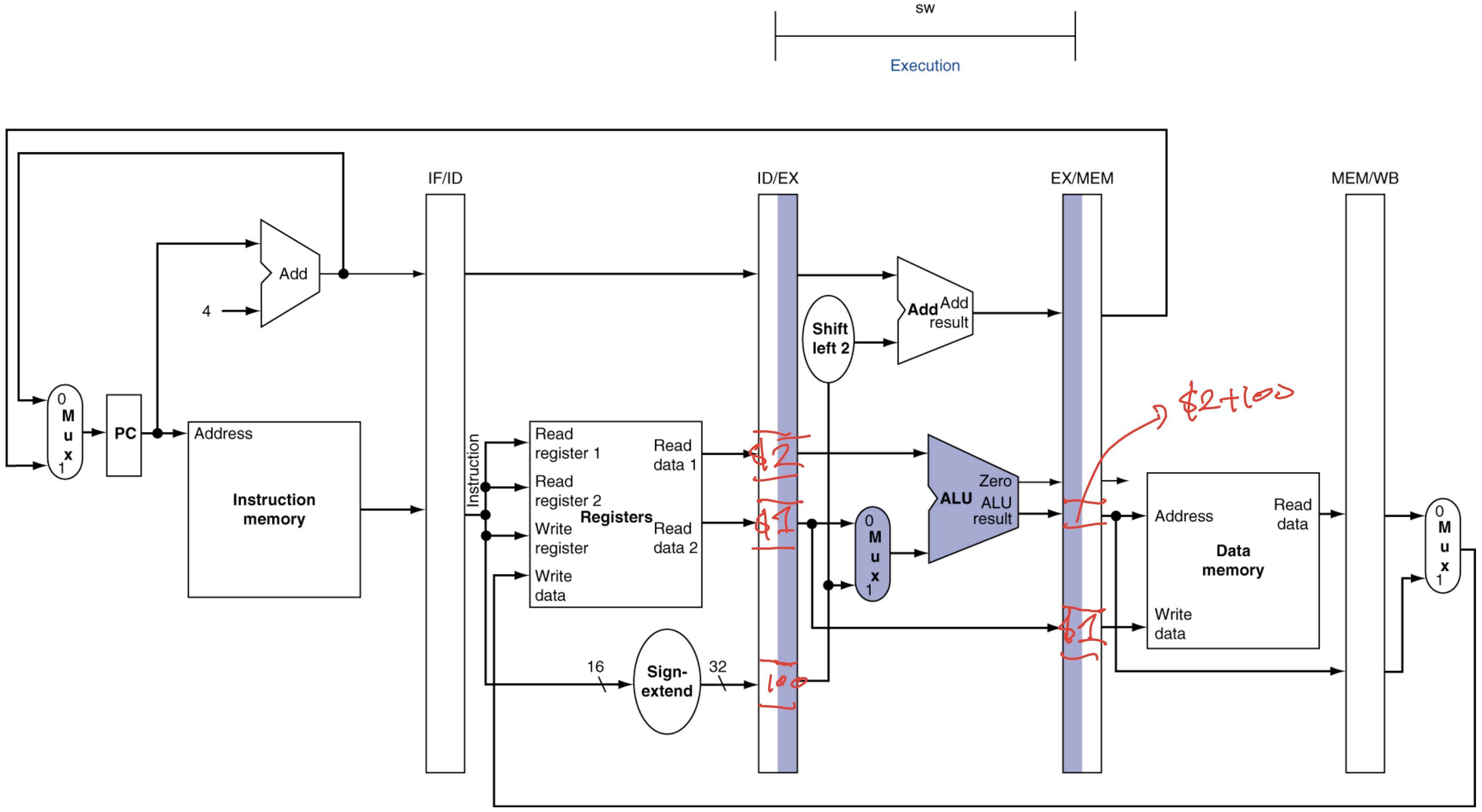
Corrected Datapath for Load



Sogang University

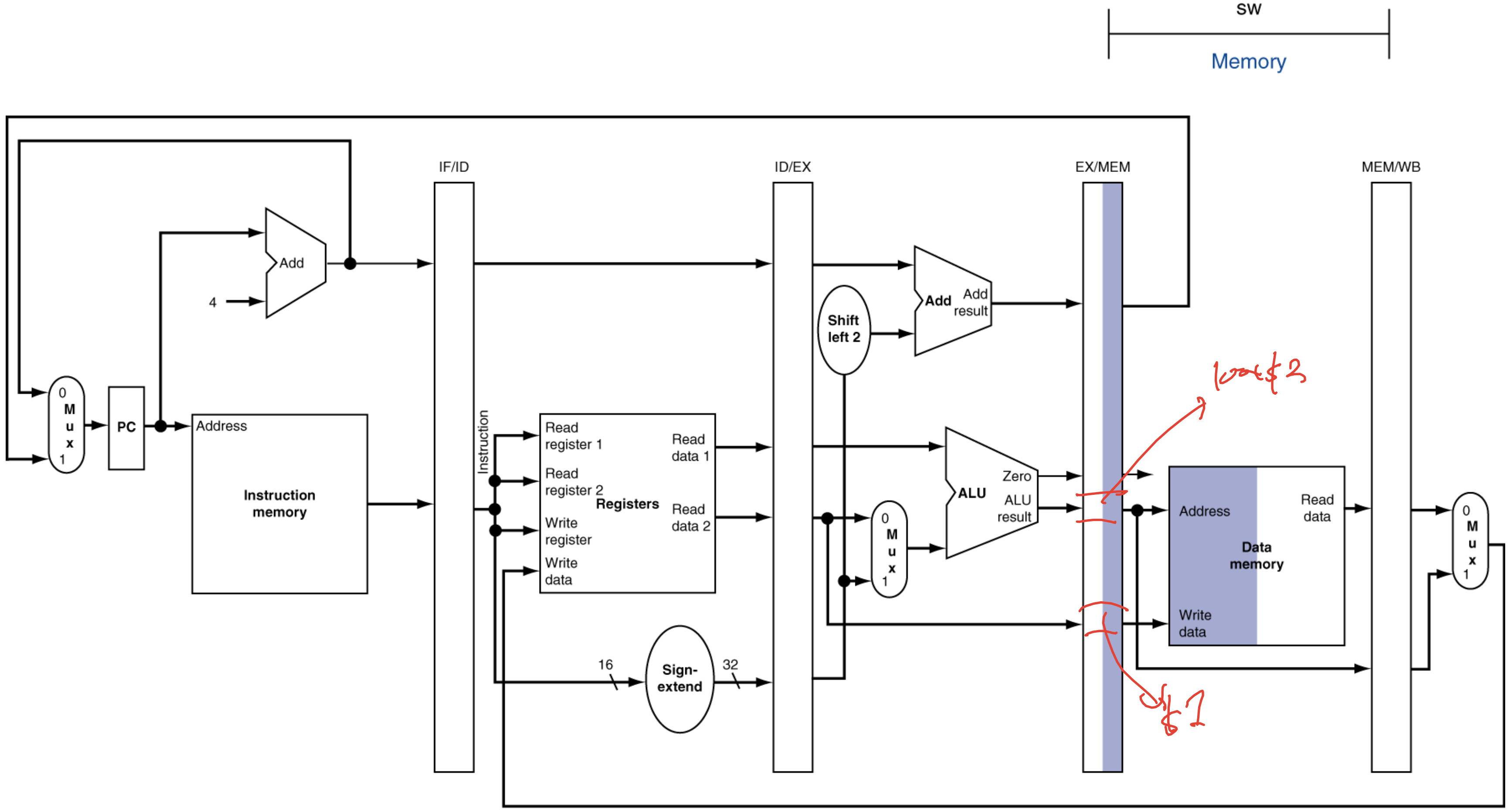
sw $1, 100($2)

EX for Store

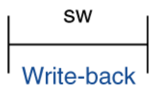


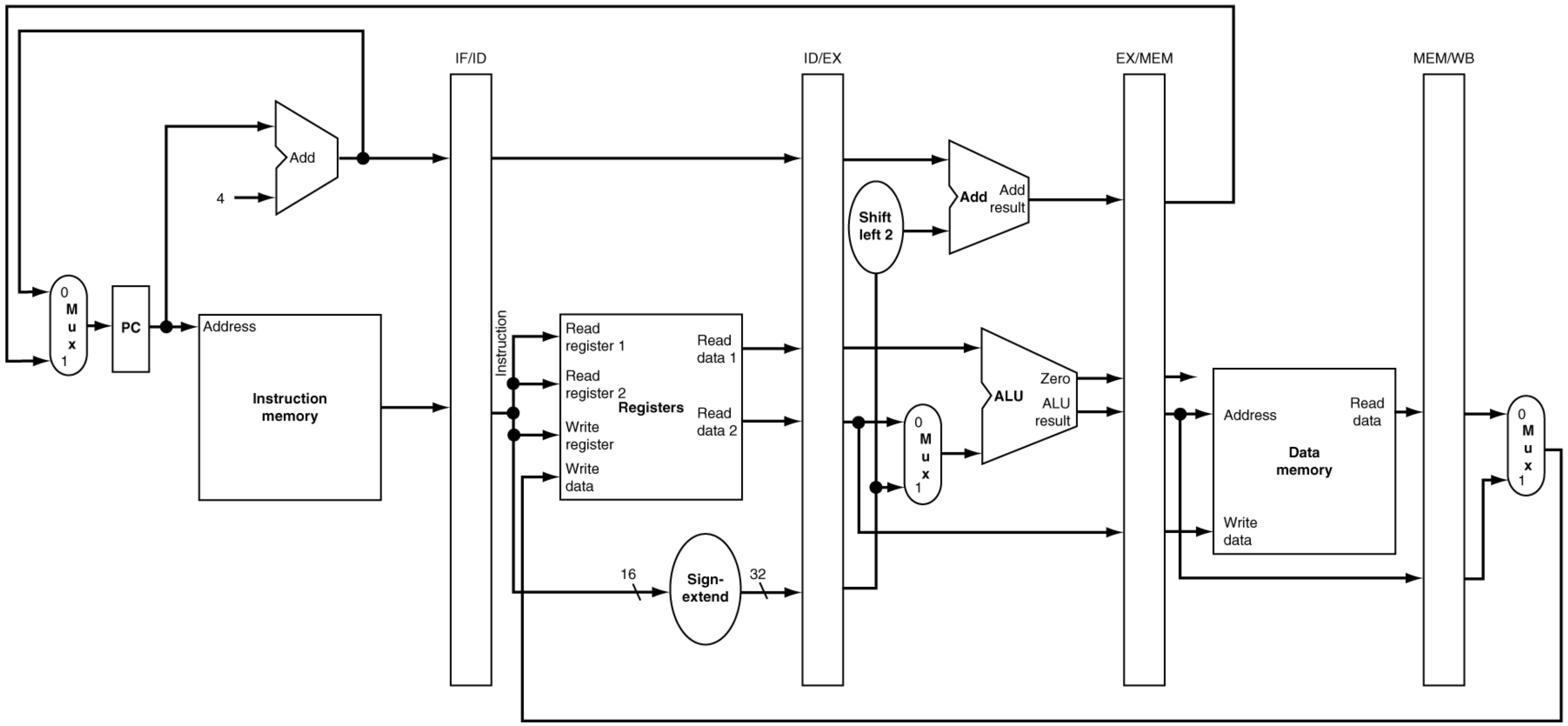
Sogang University

MEM for Store



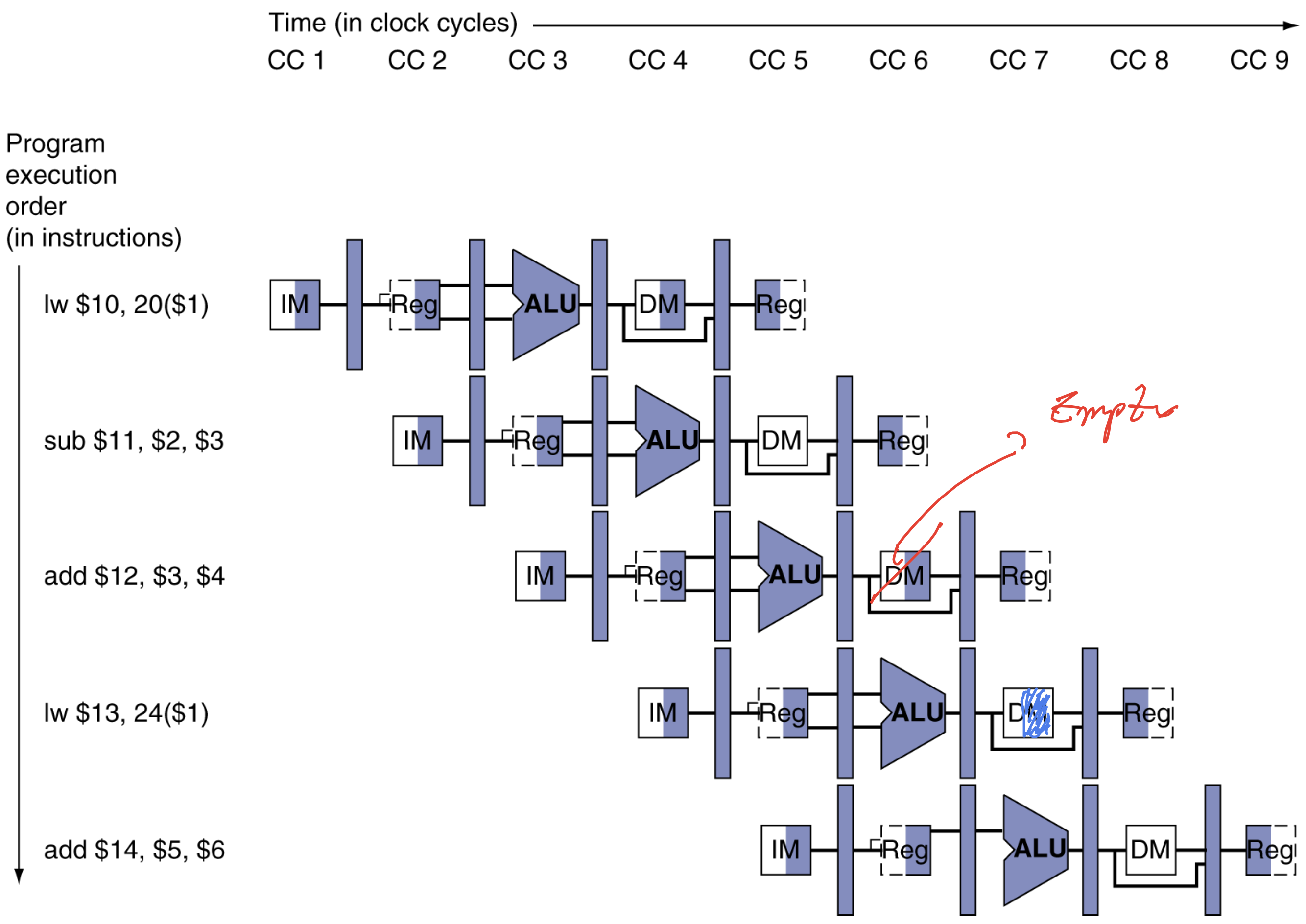
WB for Store


sw
Write-back


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
Registers
Write register
Write data
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
Read data
Data memory
Write data
0 M u x 1

Multi-Cycle Pipeline Diagram

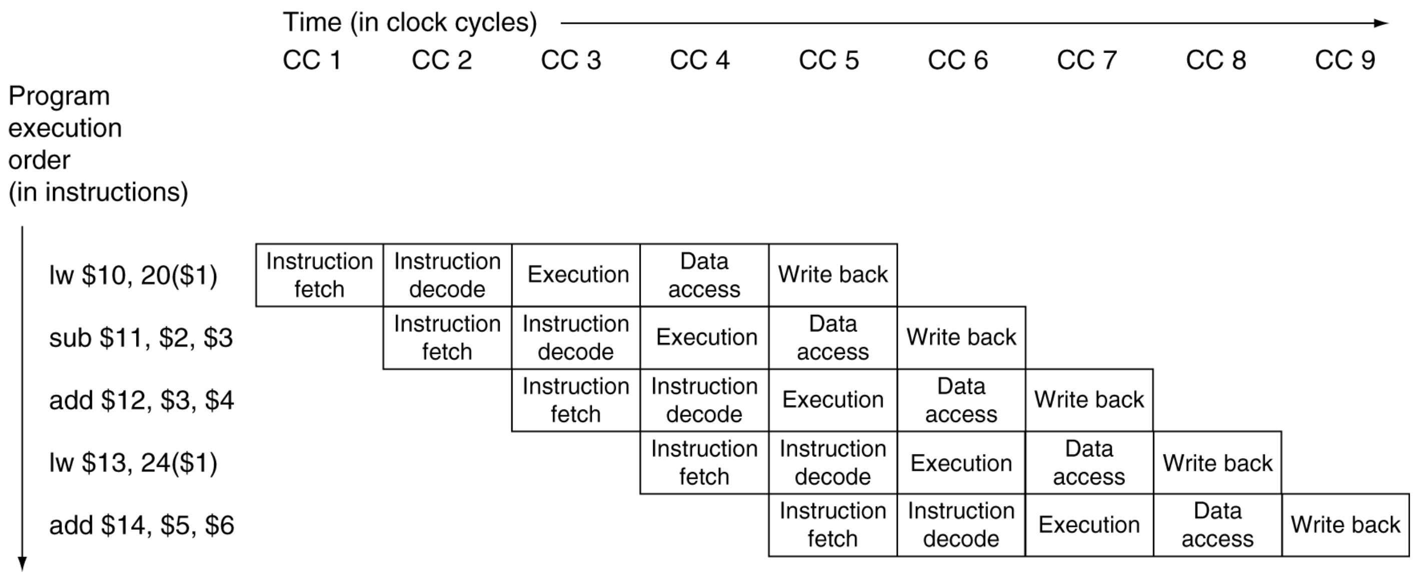
- Form showing resource usage



Sogang University

Multi-Cycle Pipeline Diagram

- Traditional form



Time (in clock cycles) →

Program execution order (in instructions) ↓

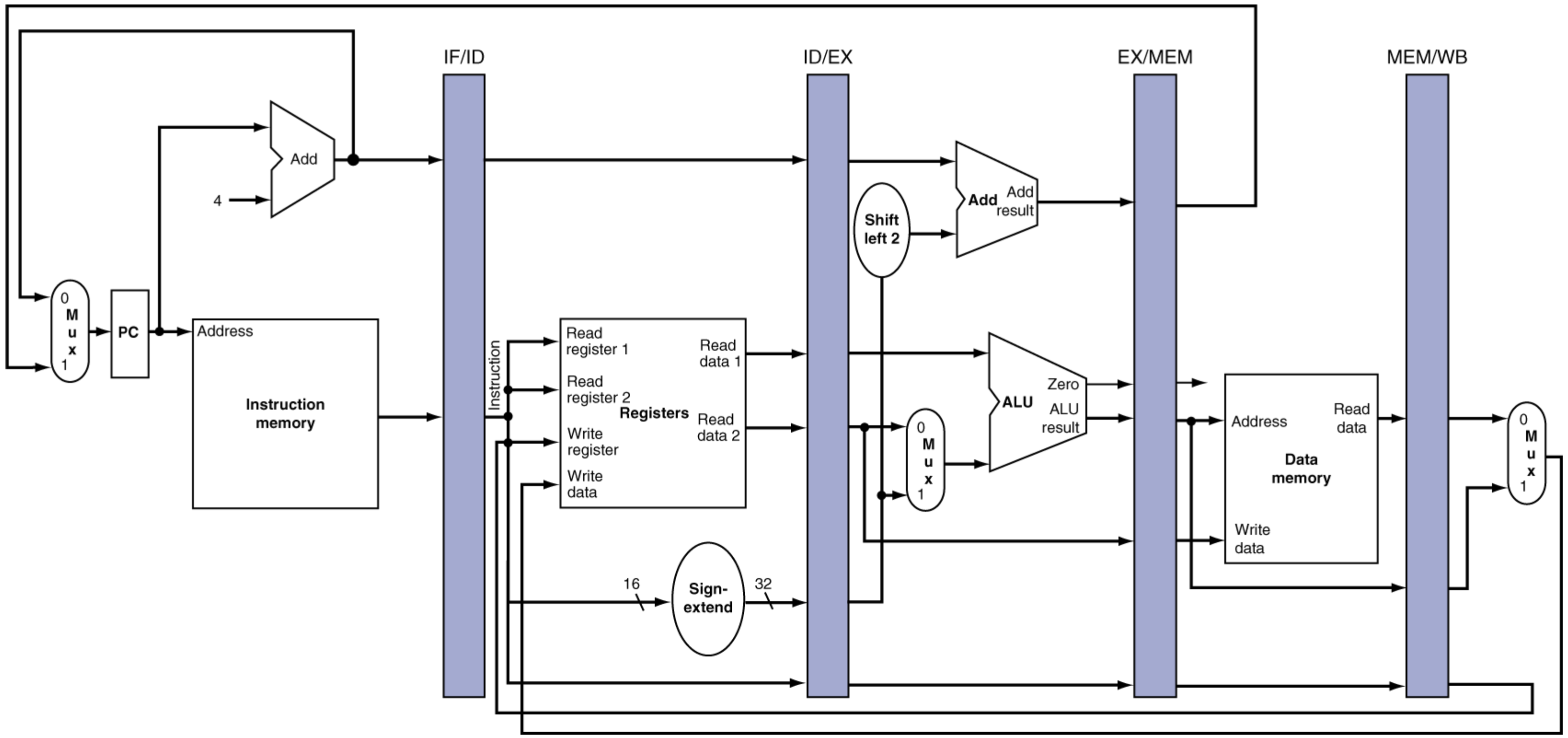
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write back

Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

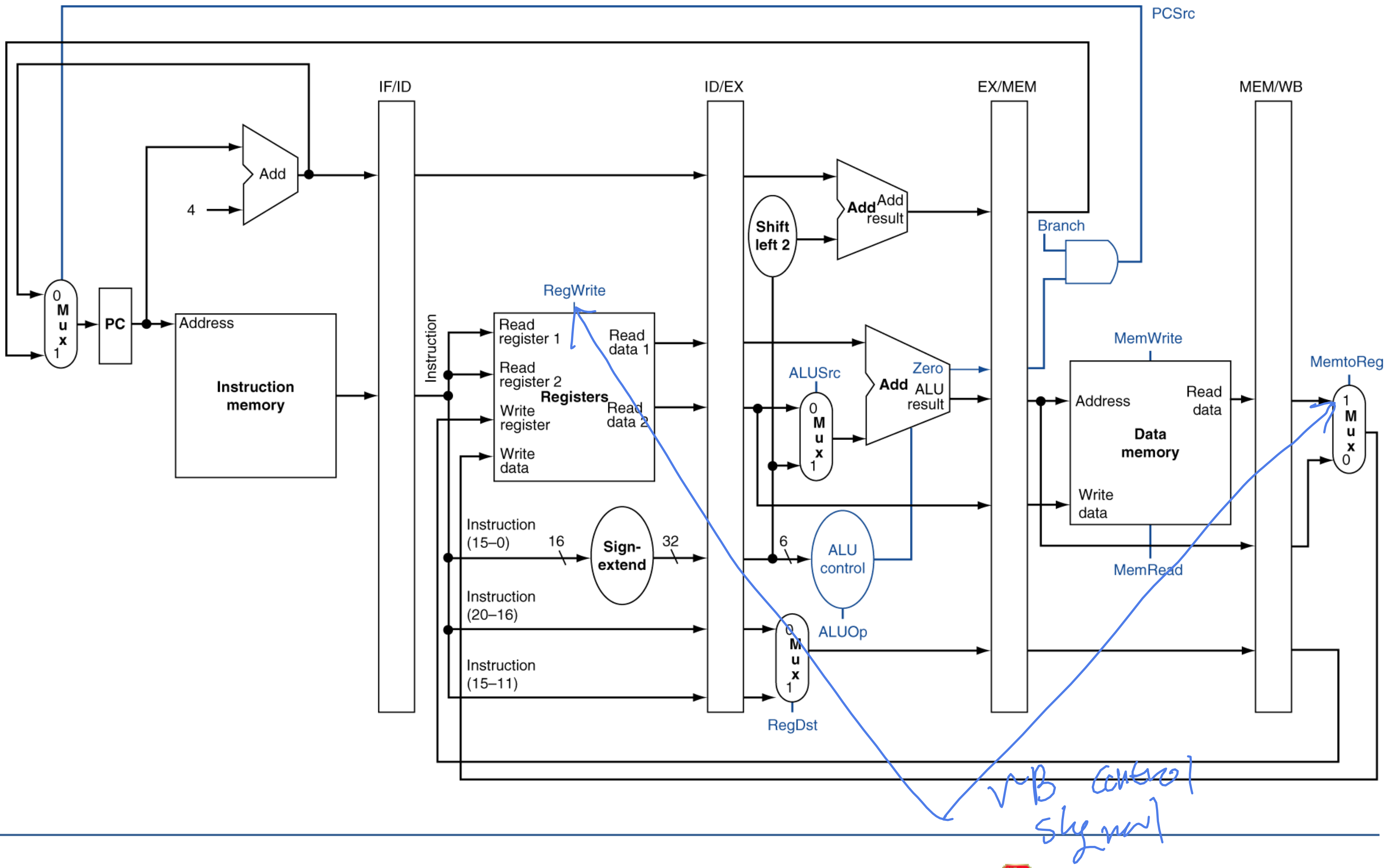
특정 cycle에서 뭘 하는지

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



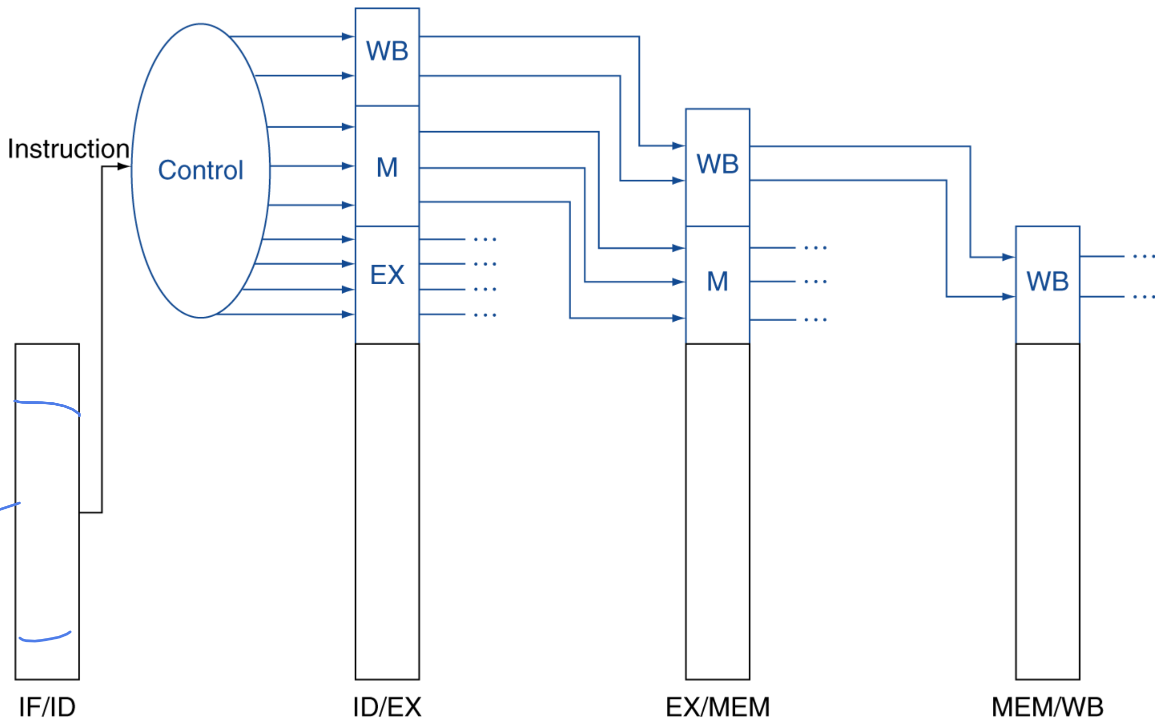
Sogang University

Pipelined Control (Simplified)



Pipelined Control

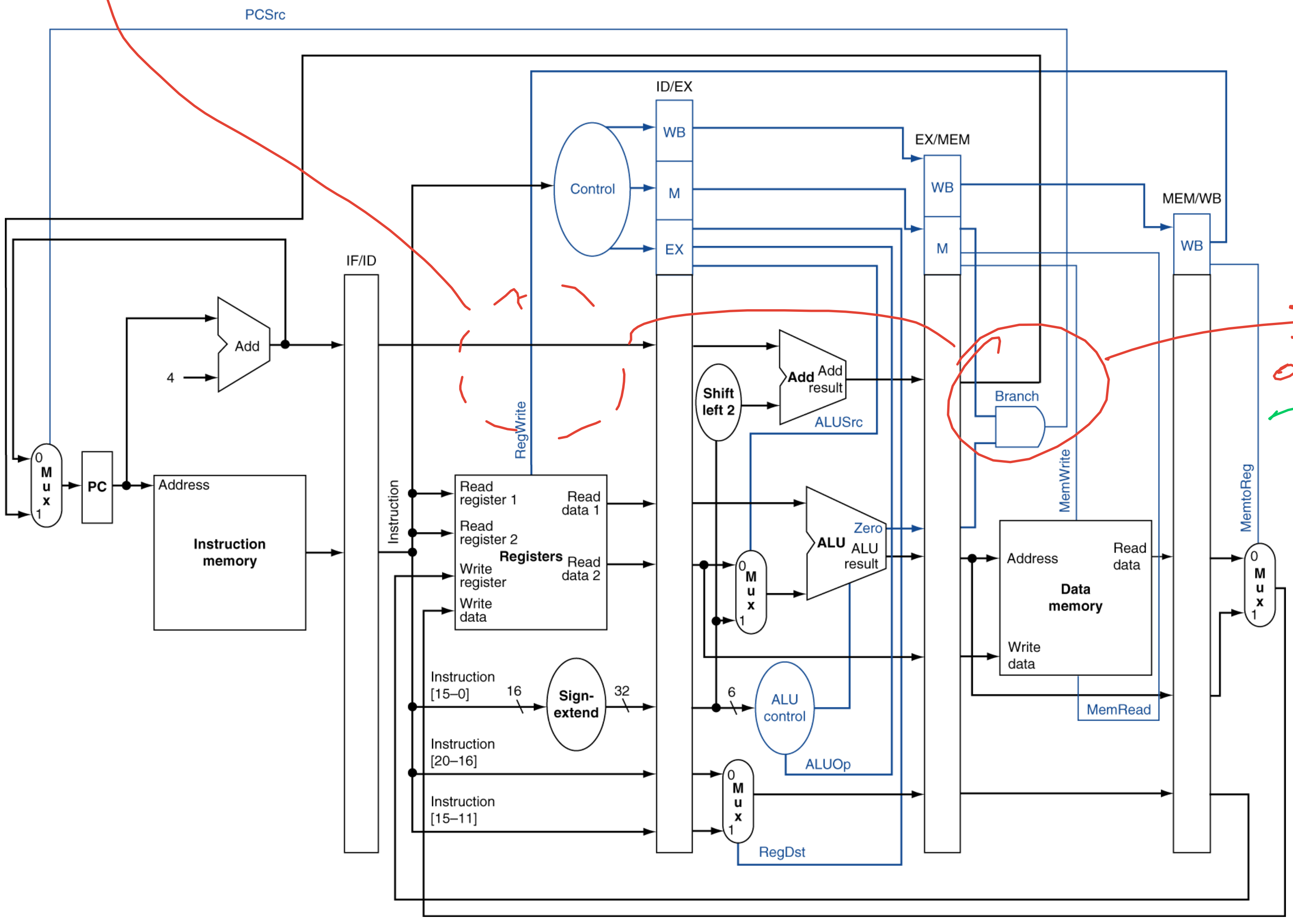
- Control signals derived from instruction
  - As in single-cycle implementation



32bits
instructions.

여기로 branch 옮기기 전

Pipelined Control



old design