دانشگاه کردستان
University of Kurdistan
زانکۆی کوردستان

Department of Computer Engineering
University of Kurdistan

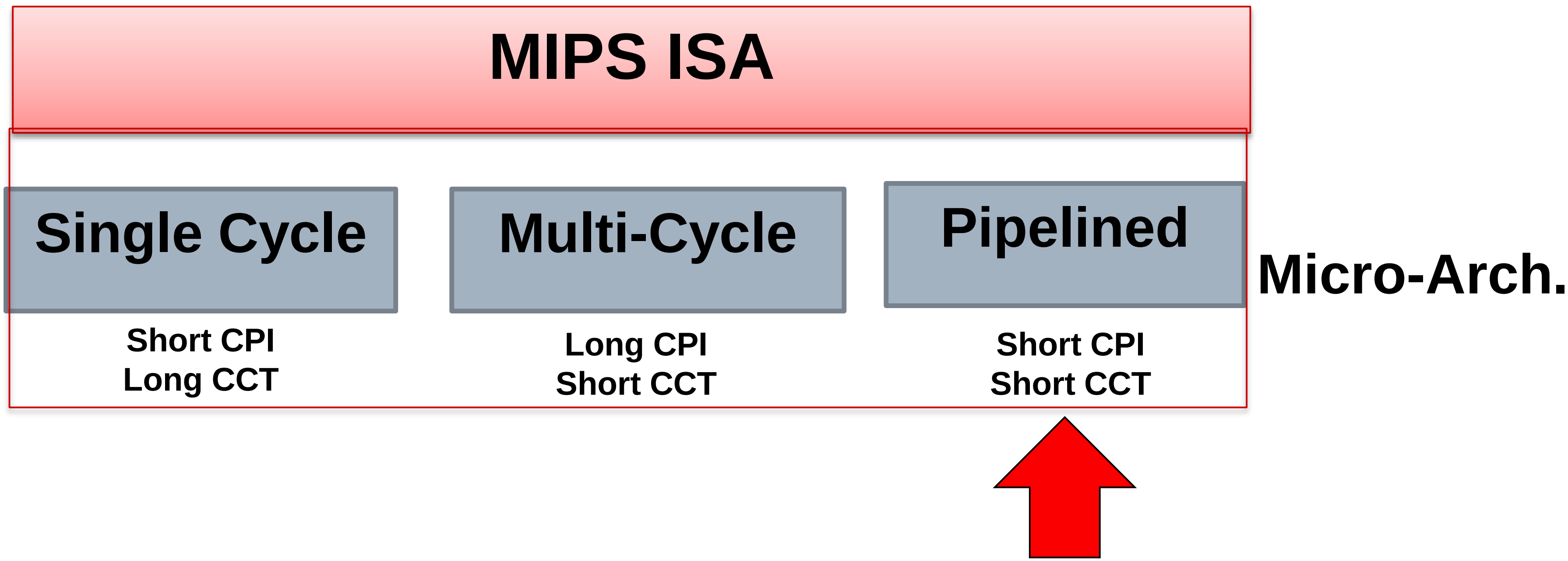
Computer Architecture

Pipelining

By: Dr. Alireza Abdollahpouri

Pipelined MIPS processor

Any instruction set can be implemented in many different ways



Getting the Best of Both Datapaths

Pipelined:
Clock rate = 500 MHz
CPI ≅ 1

Single-cycle:
Clock rate = 125 MHz
CPI = 1

Multicycle:
Clock rate = 500 MHz
CPI ≅ 4

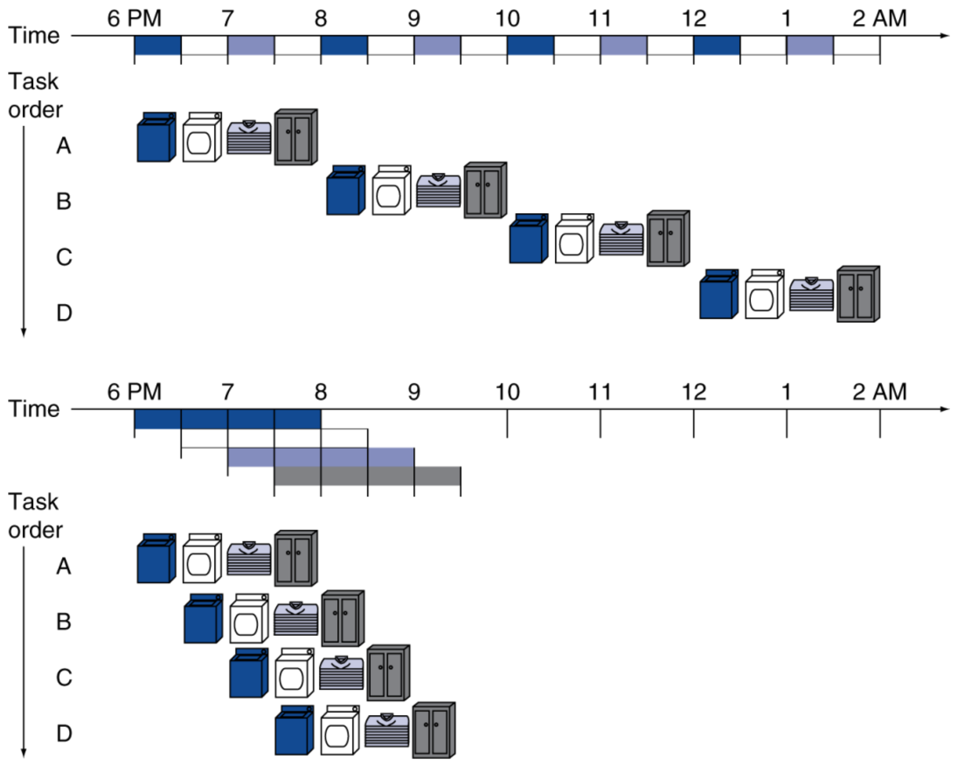

Fetch
Reg
Read
ALU
Data
Memory
Reg
Write

Pipelining Analogy

- Car assembly

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop loads:
  - Speedup = number of stages
  - = 4

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

Pipeline Performance

Assume time for stages is

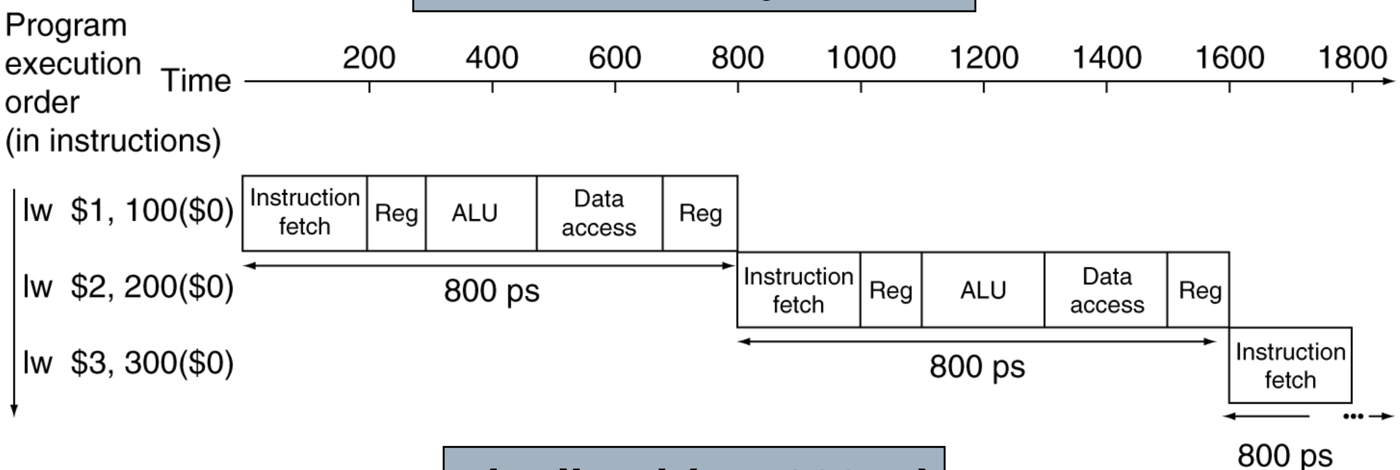
- 100ps for register read or write
- 200ps for other stages

Compare pipelined datapath with single-cycle datapath

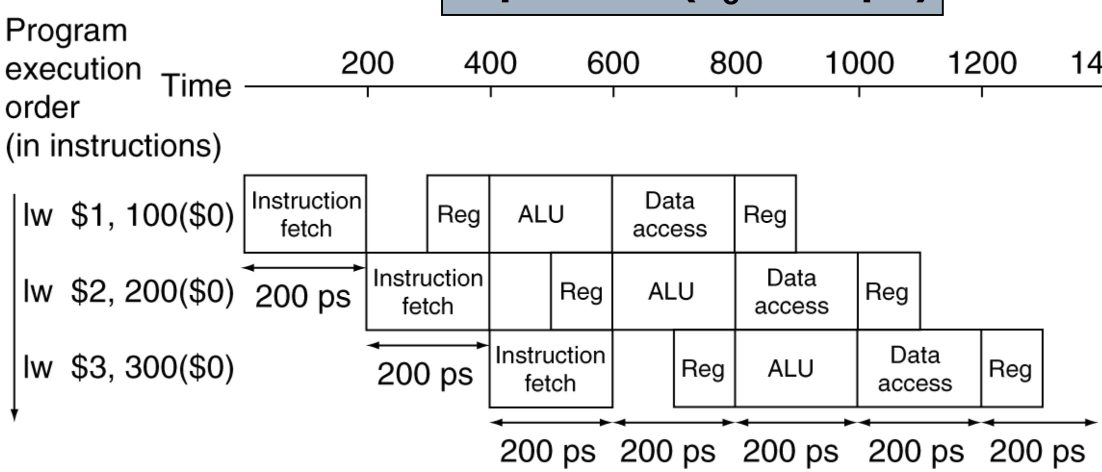
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

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

$$\text{Time between instructions}_{pipelined} = \frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$$

If not balanced, speedup is less

Speedup due to increased throughput

- Latency (time for each instruction) does not decrease

Pipelining and ISA Design

MIPS stands for: **M**icroprocessor without **I**nterlocked **P**ipelined **S**tages

- 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

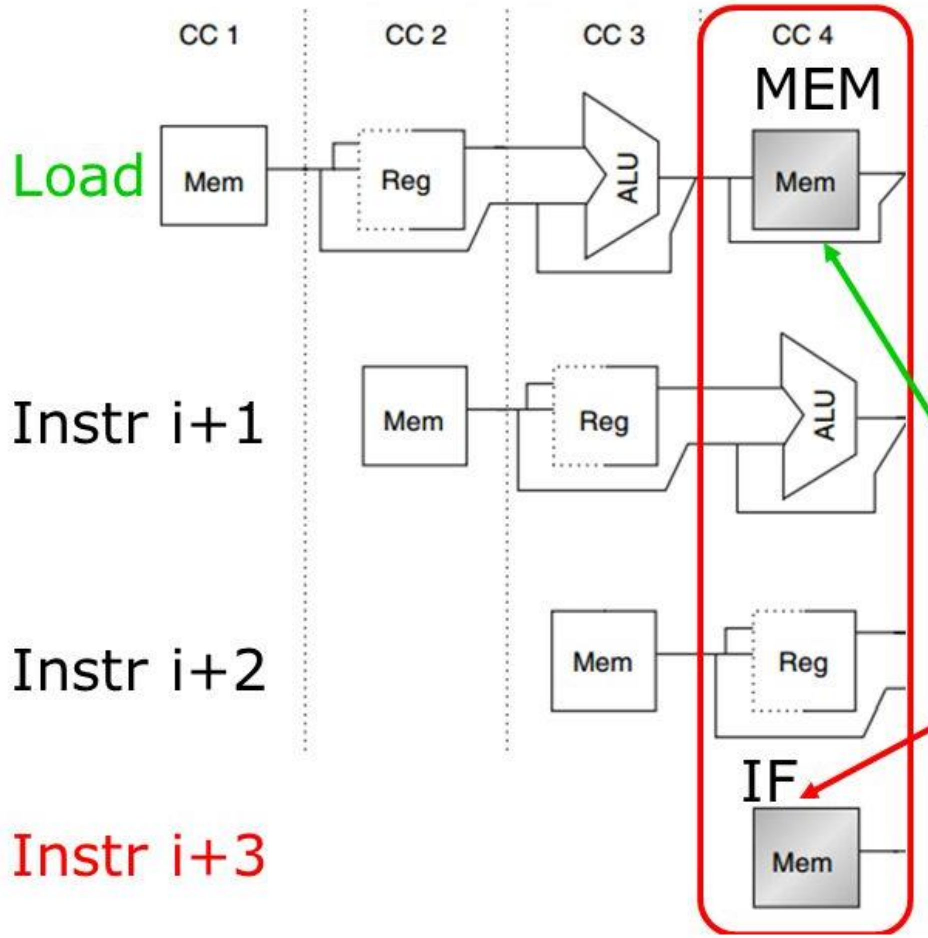
Hazards

- Situations that prevent starting the next instruction in the next cycle
- **Structure hazards**
  - A required resource is busy
- **Data hazard**
  - Need to wait for previous instruction to complete its data read/write
- **Control hazard**
  - 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

Structural Hazards



If same memory is used for Instruction and Data

data access

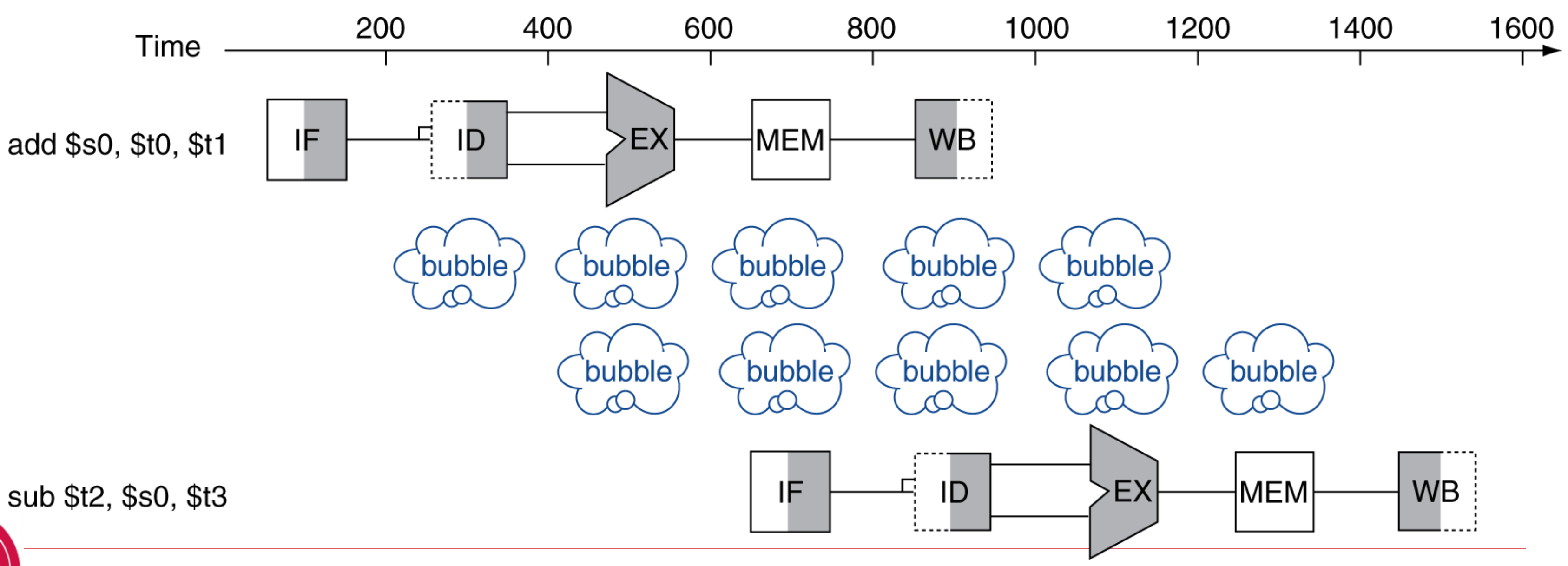
vs

instr fetch

Data Hazards

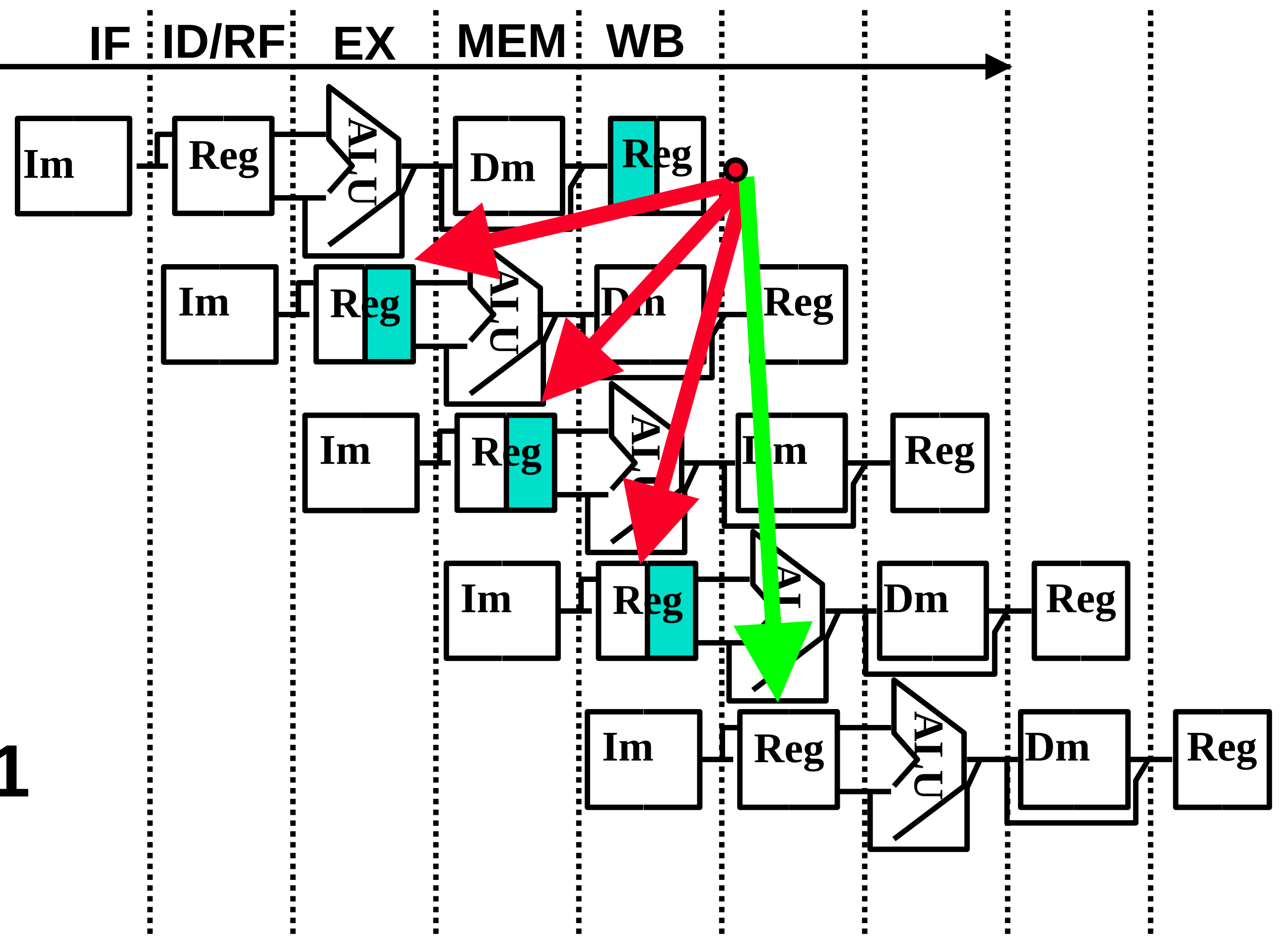
An instruction depends on completion of data access by a previous instruction

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



Backward dependencies in time

Time (clock cycles)



Instr. Order

add $1,$3,$0

sub $4,$1,$3

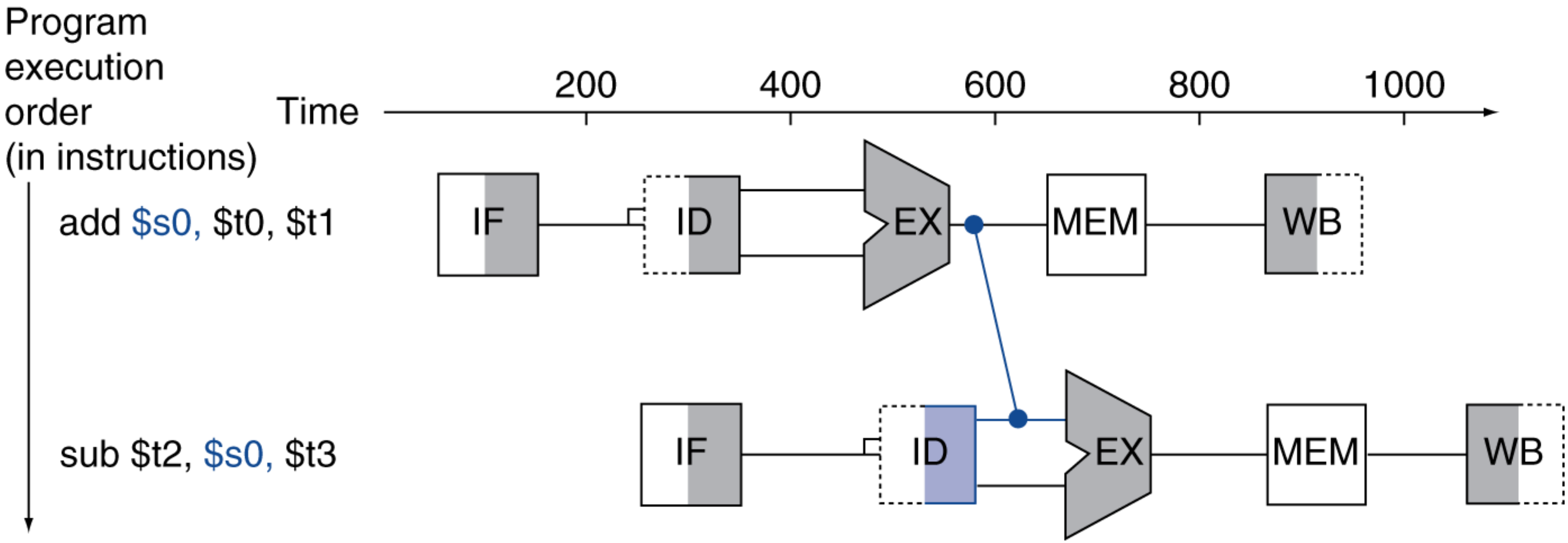
and $6,$1,$7

or r$,$1,$9

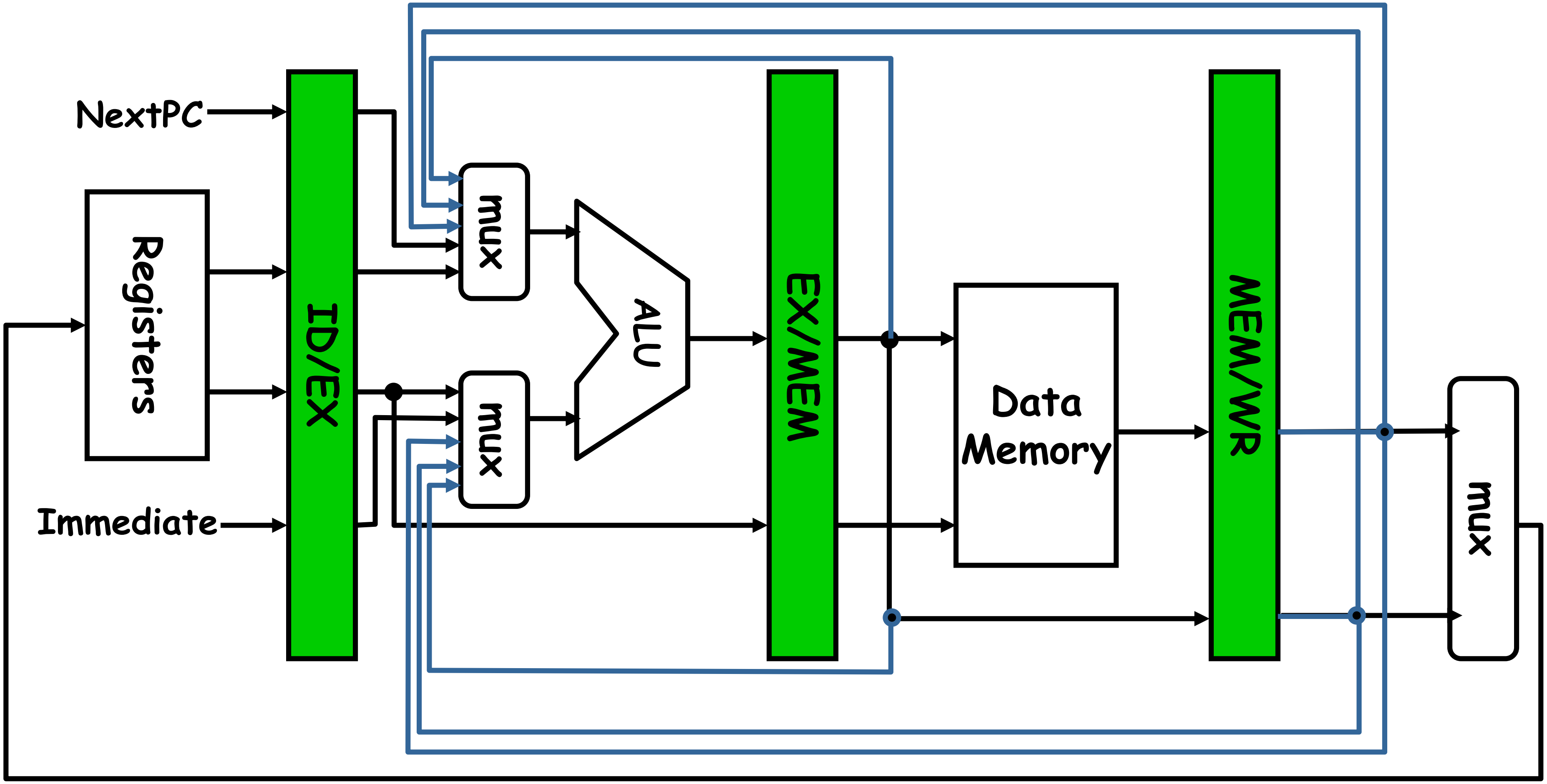
xor $10,$1,$11

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



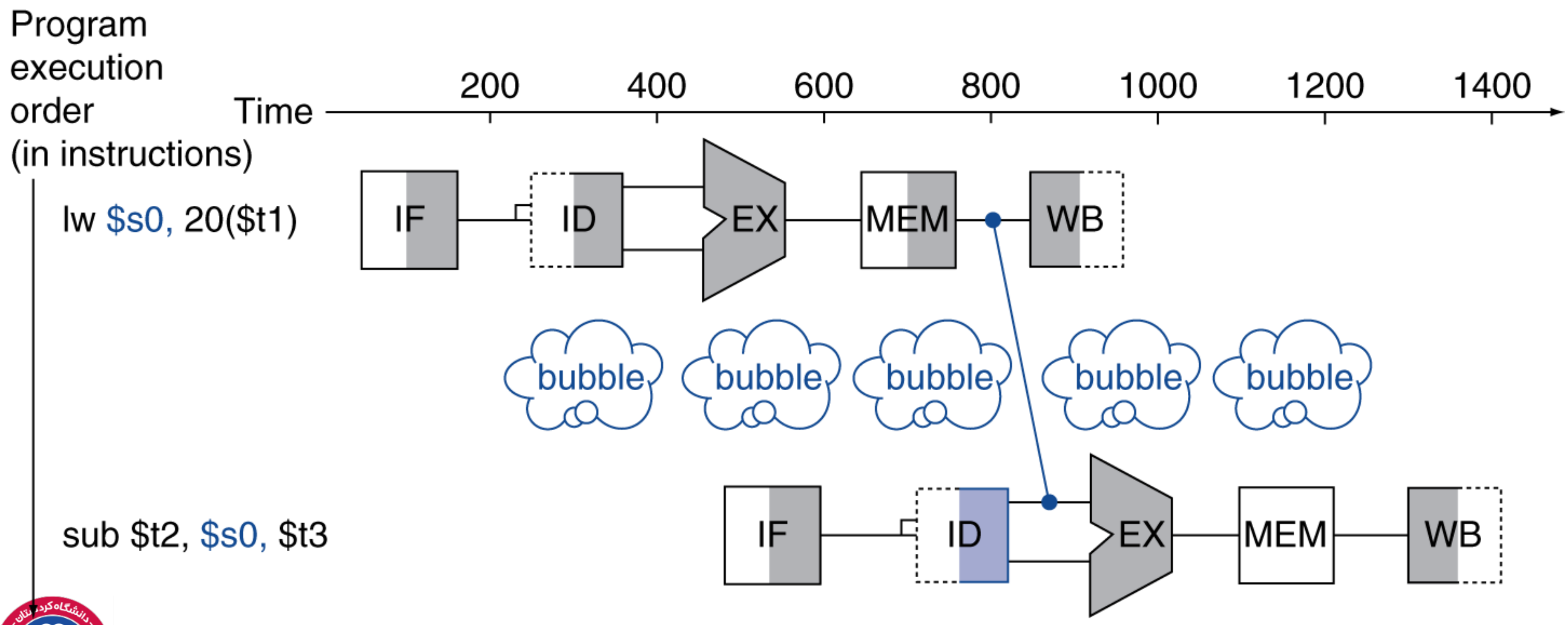
New Paths to support Forwarding



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;`

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

13 cycles

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

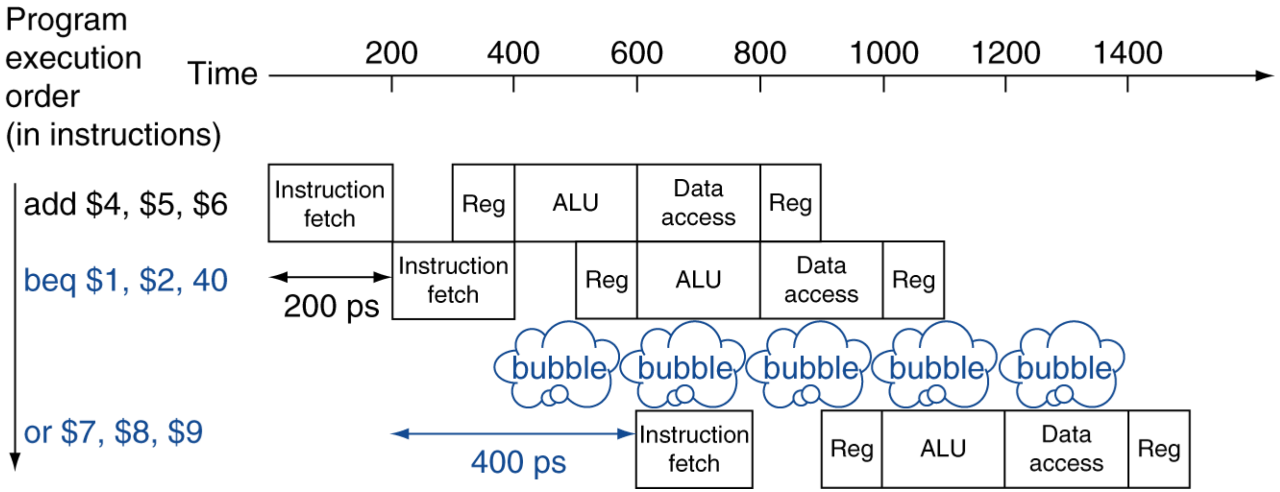
11 cycles

Control Hazards

- 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

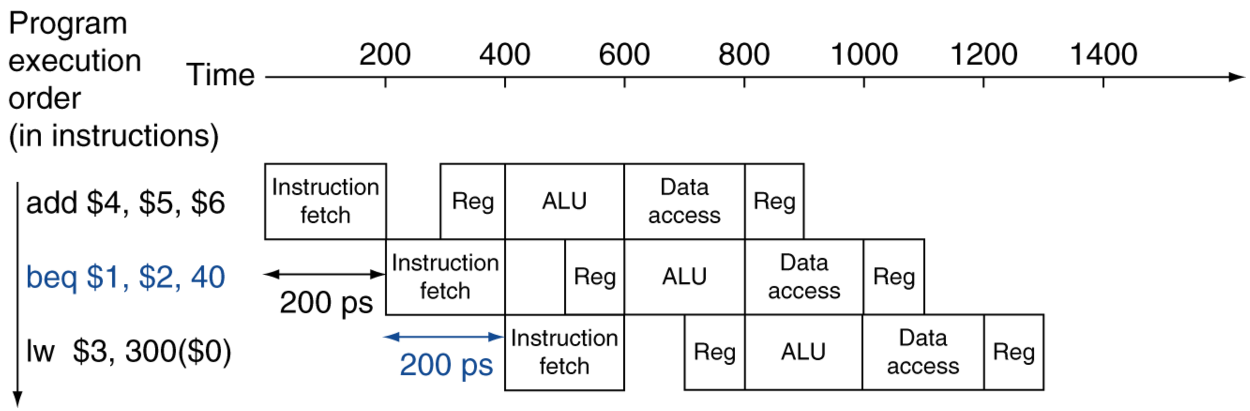


Branch Prediction

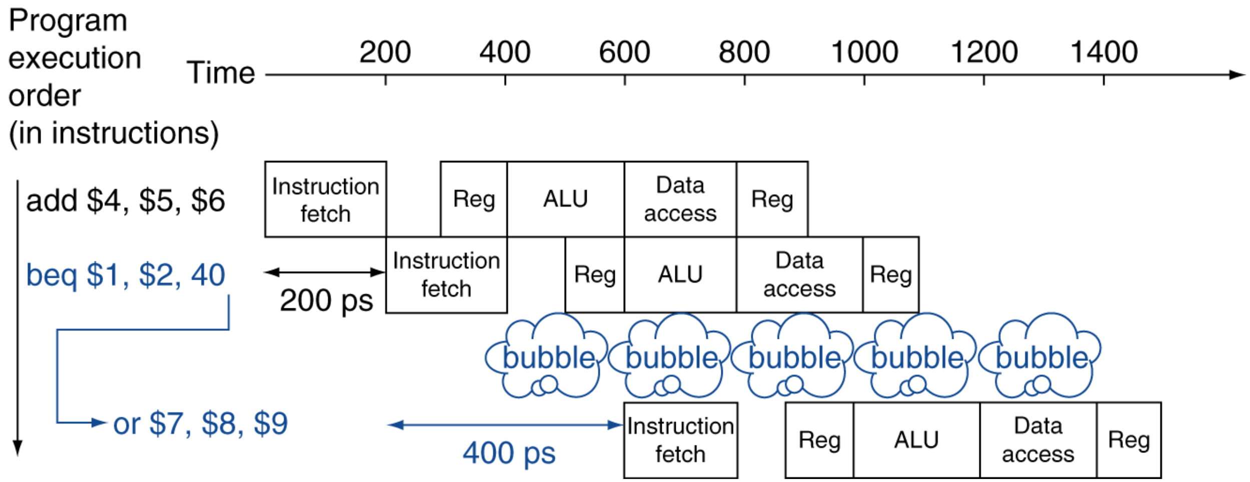
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



More-Realistic Branch Prediction

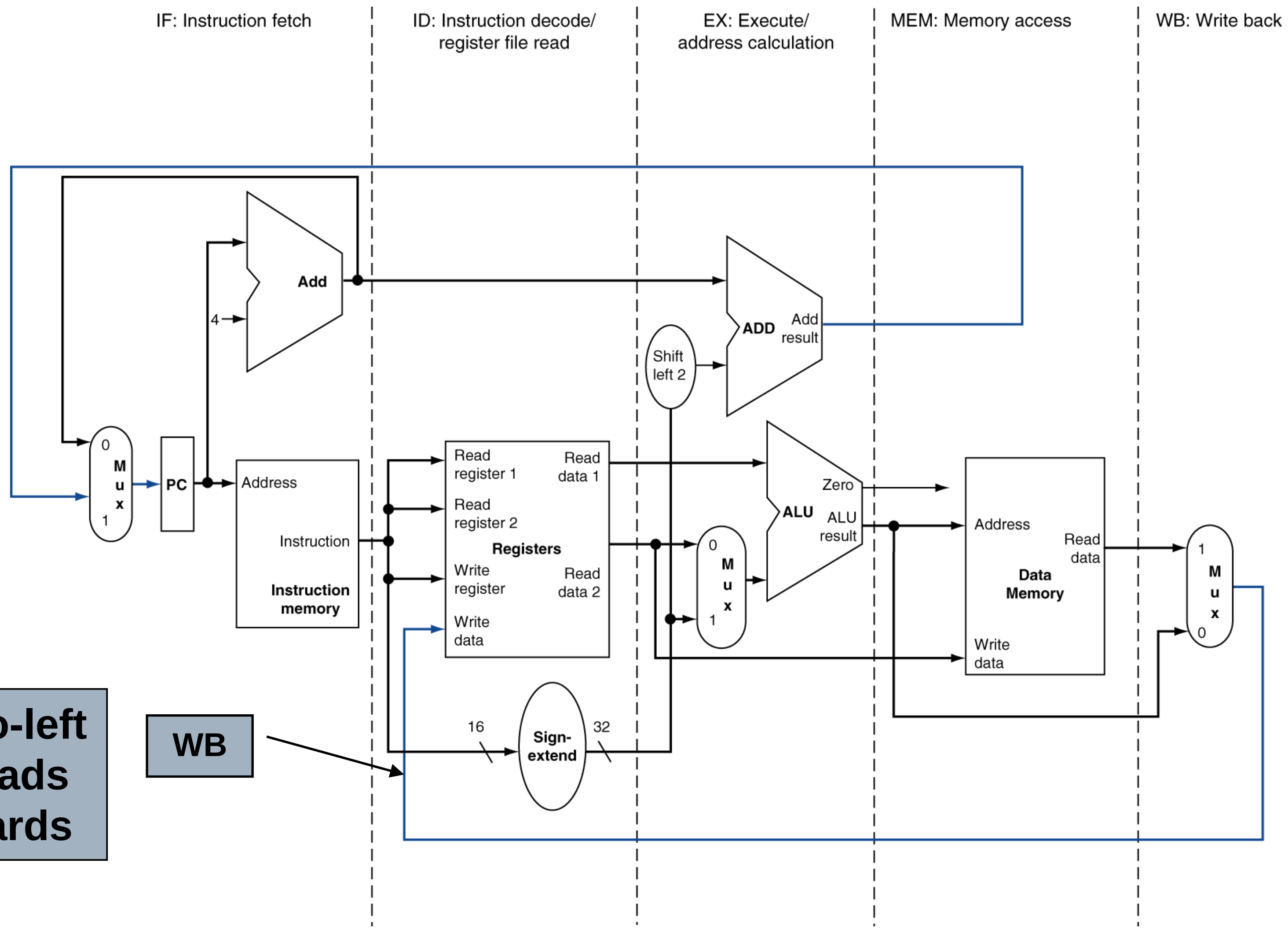
- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each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

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

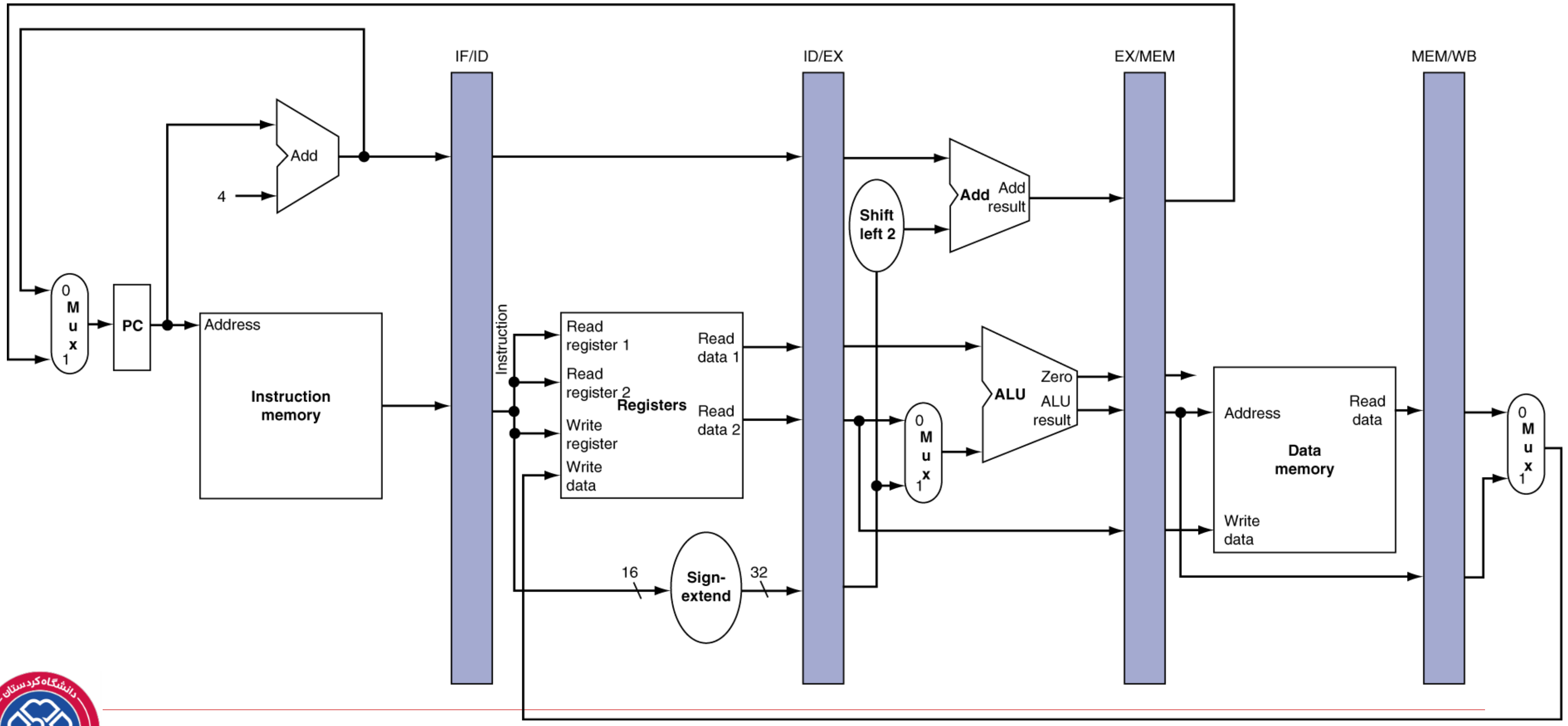


Right-to-left flow leads to hazards

WB

Pipeline registers

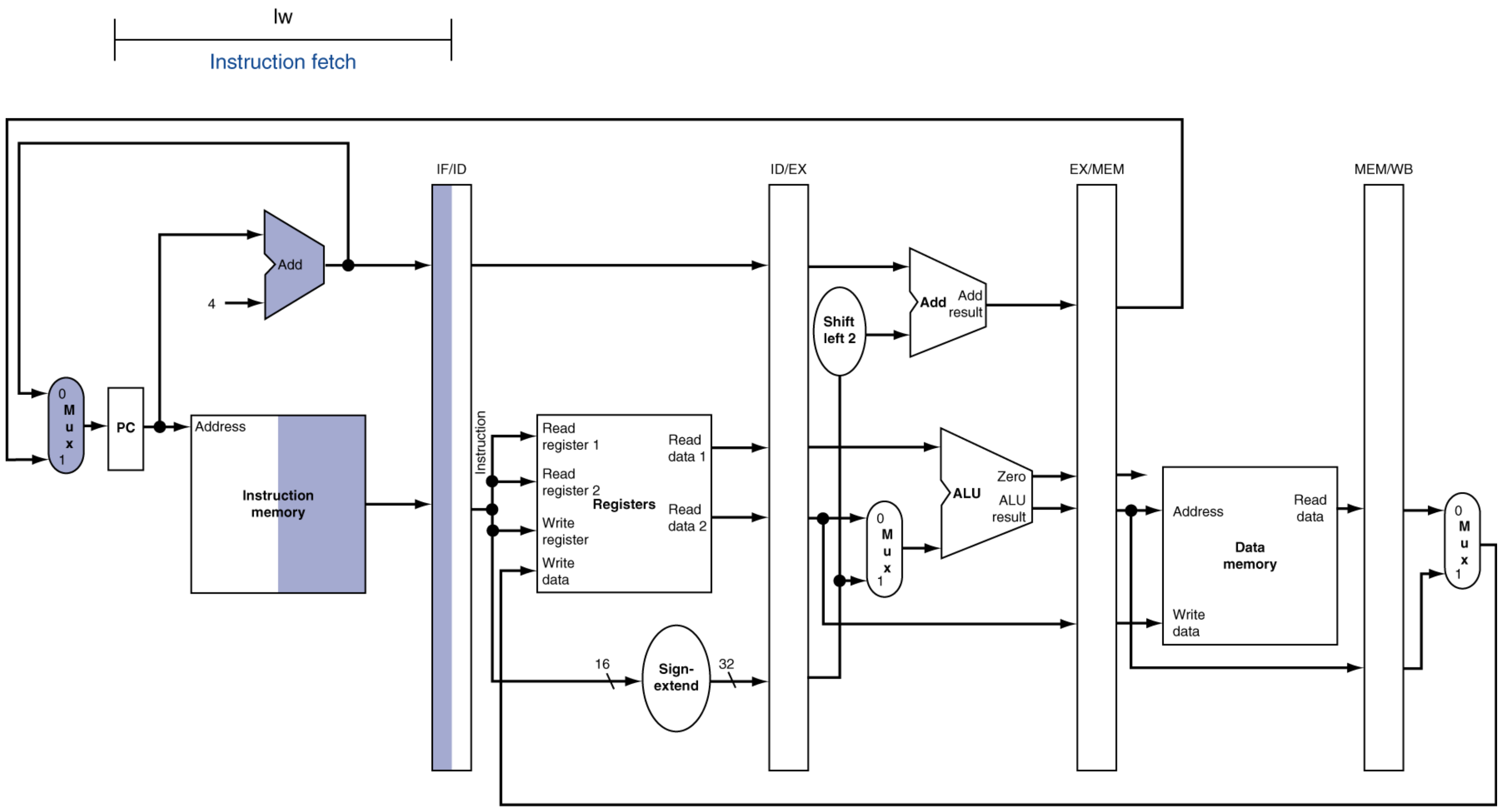
- Need registers between stages
  - To hold information produced in previous cycle



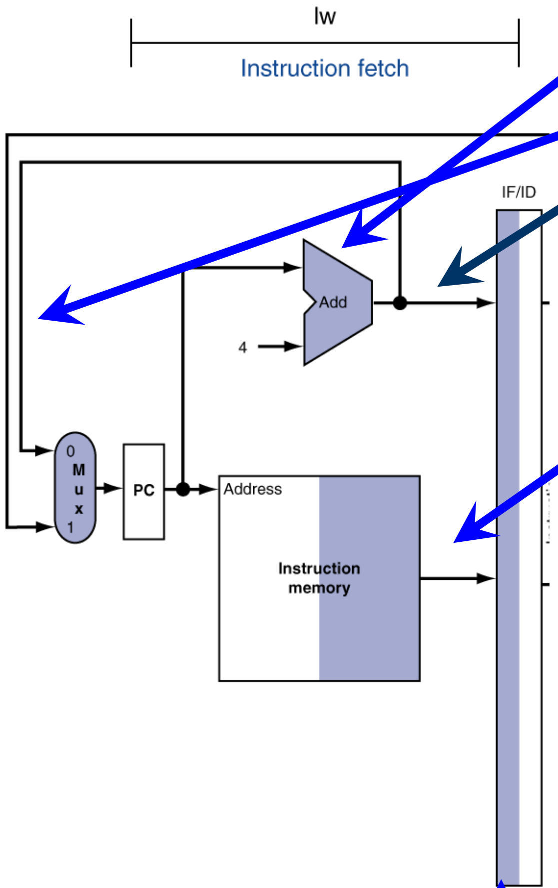
Pipeline Operation

- 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, …



IF for Load, Store, …



PC+4 is computed,

stored back into the PC,

stored in the IF/ID buffer although it <u>will not</u> be needed in a later stage for LW or SW

Instruction word is fetched from memory,

and stored in the IF/ID buffer because it <u>will</u> be needed in the next stage.

Write into the buffer

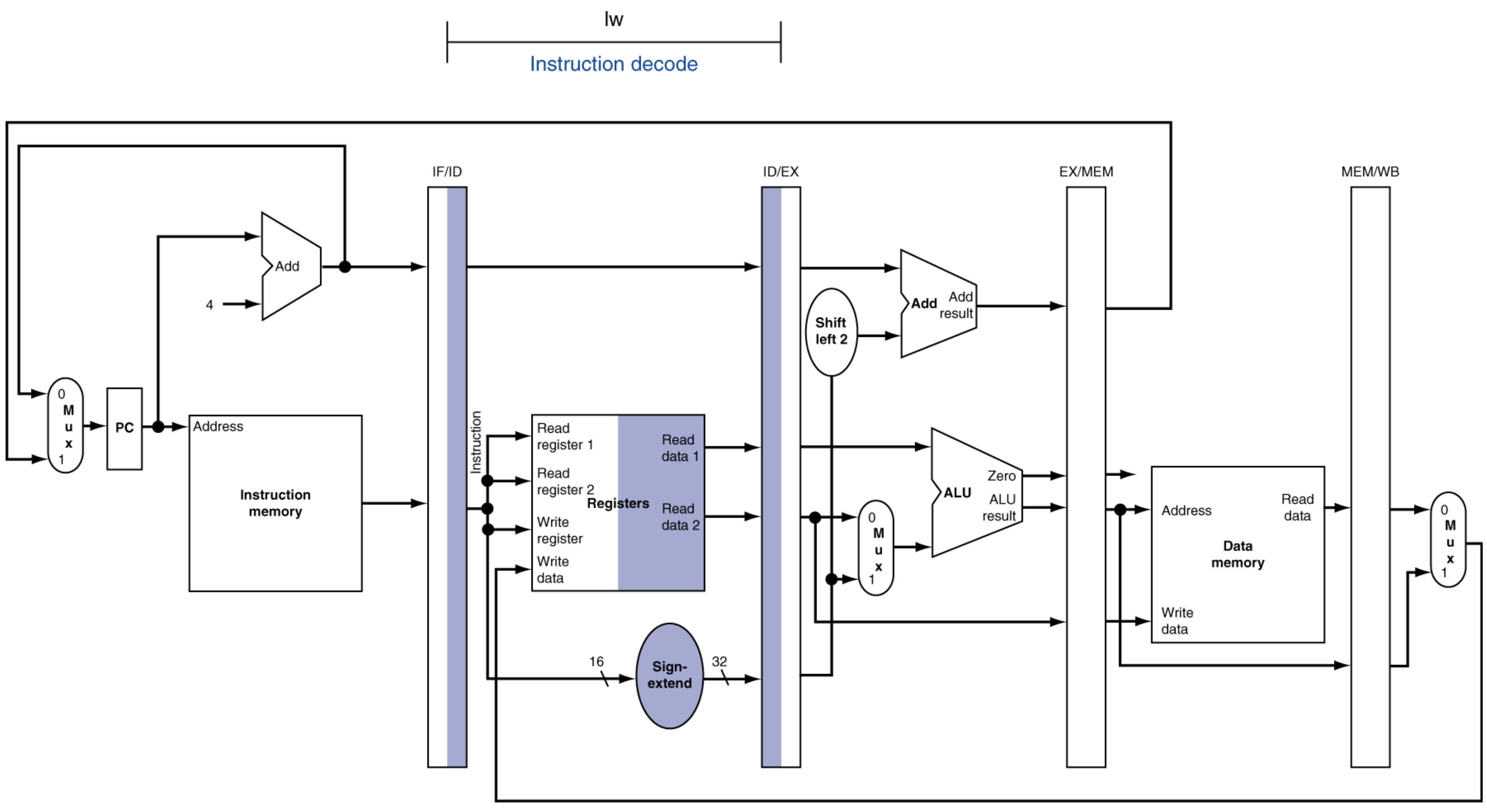
IF for Load, Store, …

- Instruction is read from memory using the address in PC and is placed in the IF/ID pipeline register

- PC address is incremented by 4 and then written back into PC to be ready for the next clock cycle

- This incremented address is also saved in IF/ID pipeline register in case it is needed later for an instruction

ID for Load, Store, …

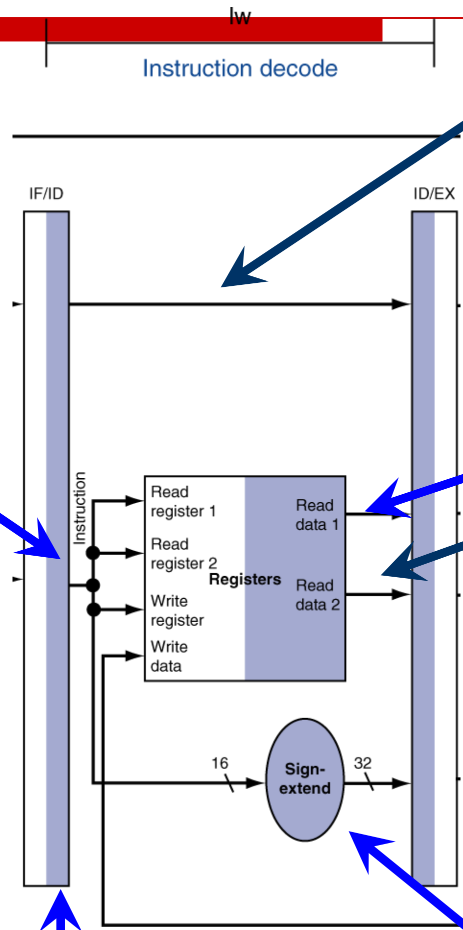


ID for Load

Bits of load instruction are taken from IF/ID buffer, while

new instruction is being fetched back in stage 1.

PC+4 is passed forward to ID/EX buffer...



Read register #1 and #2 contents are fetched and stored in ID/EX buffer until needed in next stage… #2 won't be needed.

Read from the buffer

16-bit field is fetched from IF/ID buffer, then sign-extended, then stored in the ID/EX buffer for use in a later stage.

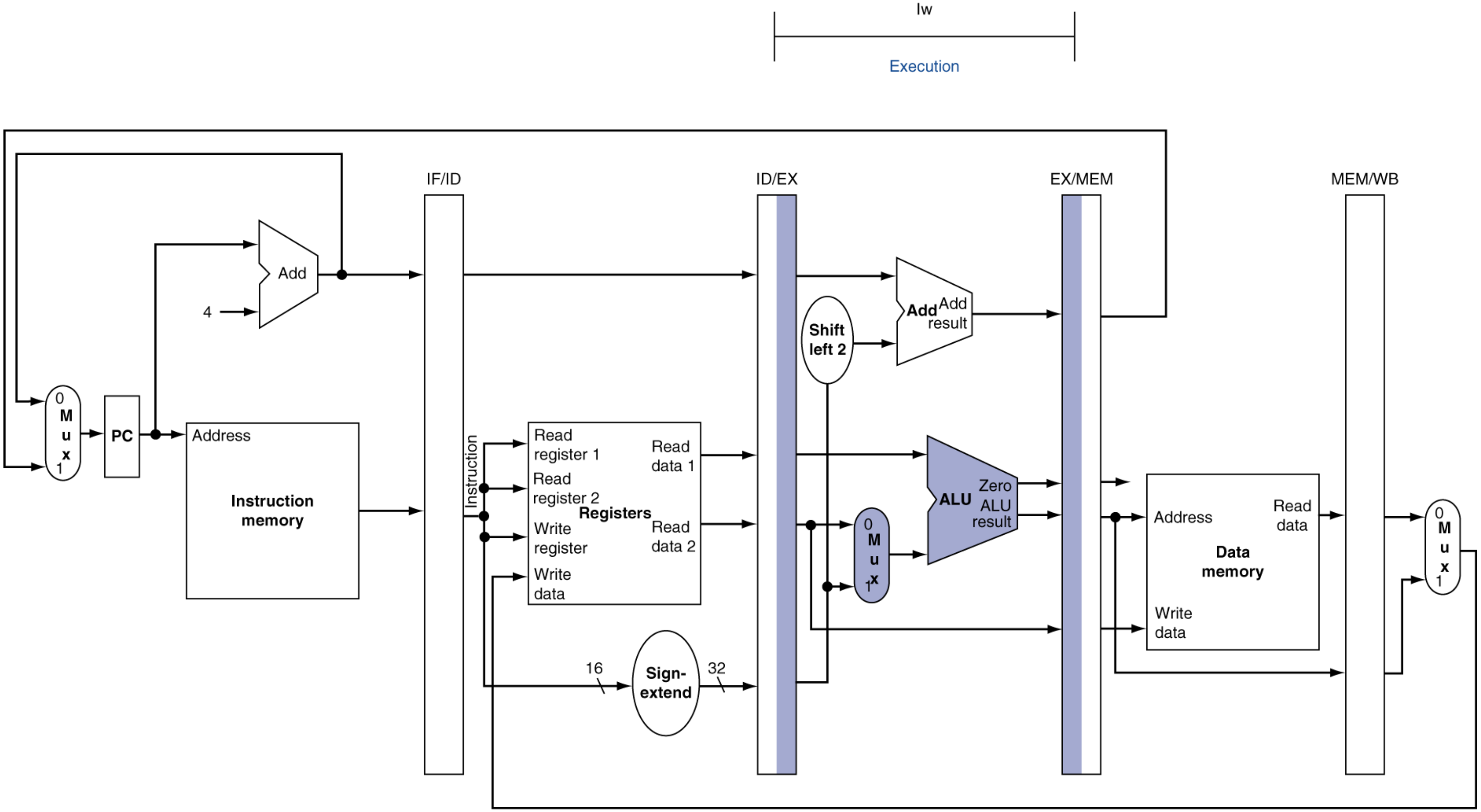
ID for Load, Store, …

- Instruction portion of IF/ID pipeline register supplying 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers

- All three values are stored in the ID/EX pipeline register, along with incremented PC address

- Everything might be needed by any instruction during a later clock cycle is transferred

EX for Load


lw
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
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
Write data
Read data

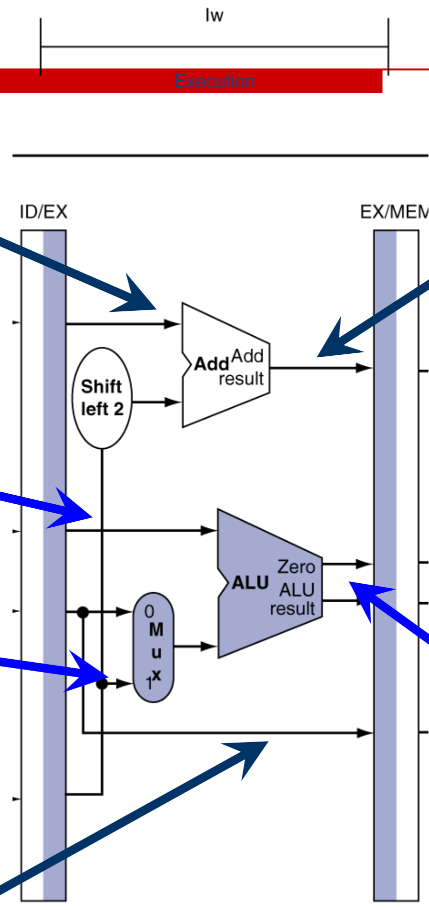
EX for Load

PC+4 is taken from ID/EX buffer and added to branch offset…

Read register #1 contents are taken from ID/EX buffer and provided to ALU.

16-bit literal is provided to ALU as second operand

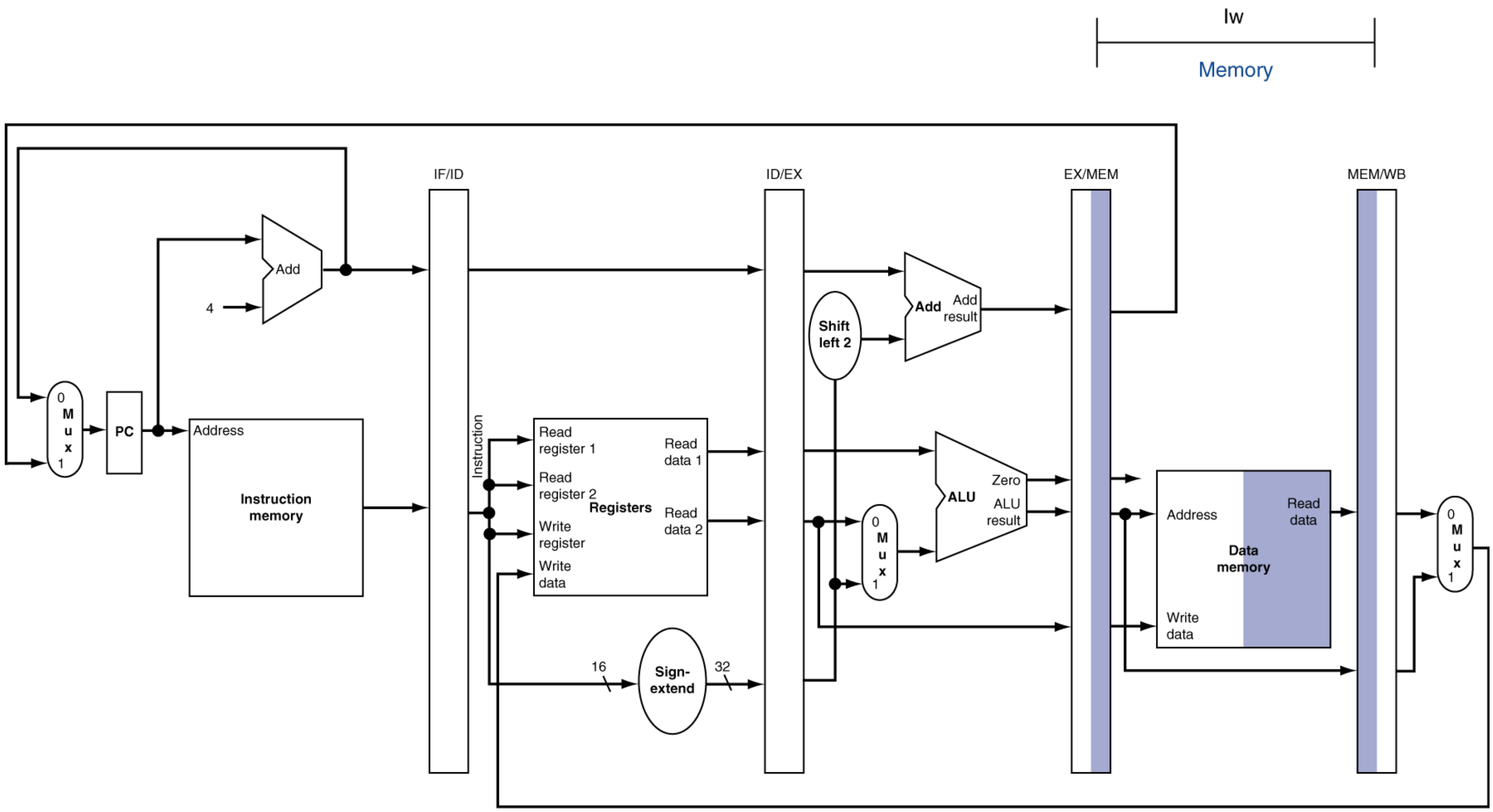
Read register #2 is passed forward to EX/MEM buffer, for possible use in later stage… but won't be needed.



Computed branch target address is stored in EX/MEM buffer to await decision in next stage... but won't be needed.

ALU result and Zero line are stored in EX/MEM buffer for use as memory address in next stage.

MEM for Load



lw
Memory
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
Data memory
Write data
Read data
0
M
u
x
1

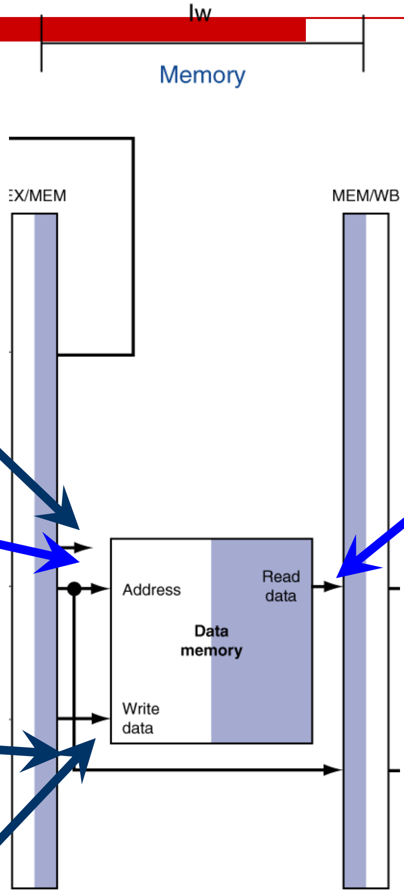
MEM for Load

Zero line taken from EX/MEM buffer for branch control logic in this stage…

ALU result is taken from EX/MEM buffer and passed to Address port of data memory.

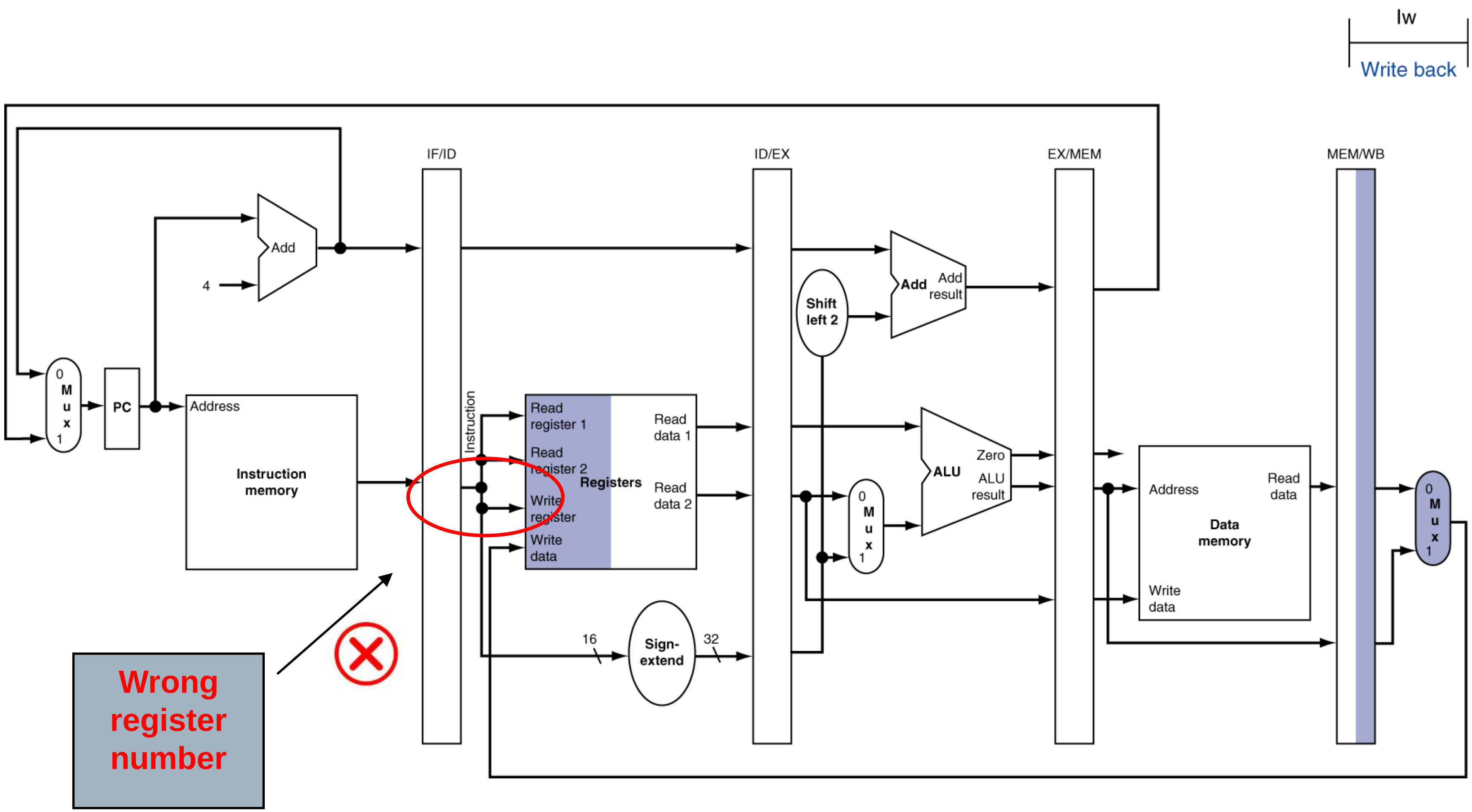
ALU result also stored in MEM/WB buffer for possible use in last stage…

Read register #2 contents taken from EX/MEM buffer and passed to Write data port of data memory.



Value on Read data port of data memory is stored in MEM/WB buffer, awaiting decision in last stage..

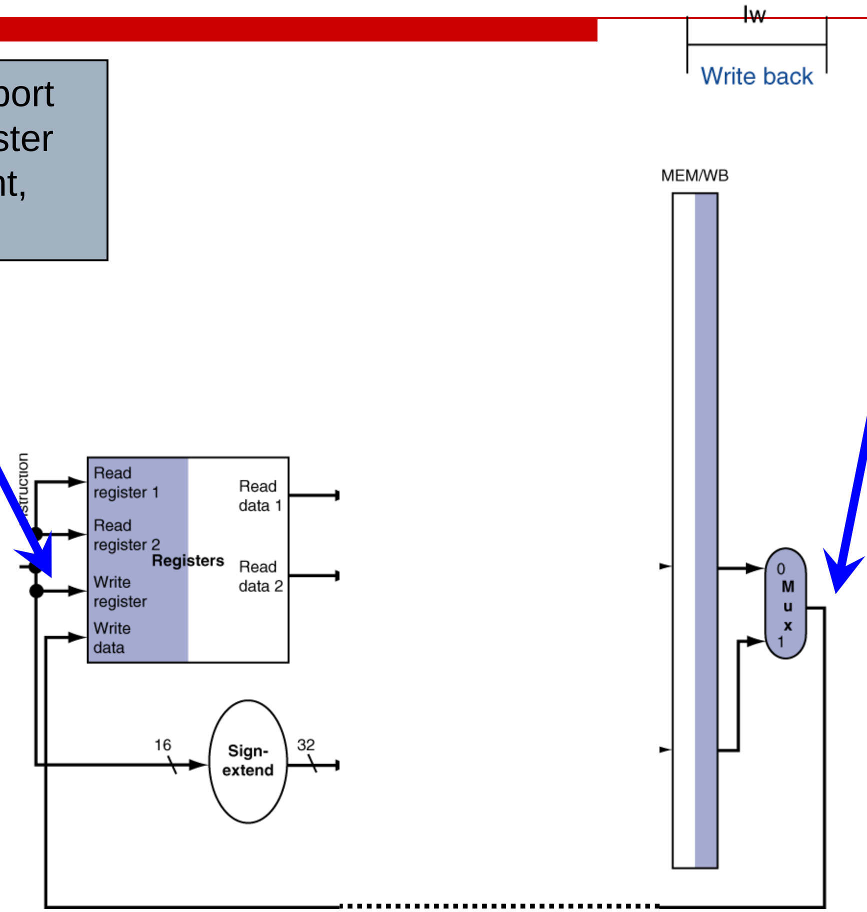
WB for Load



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

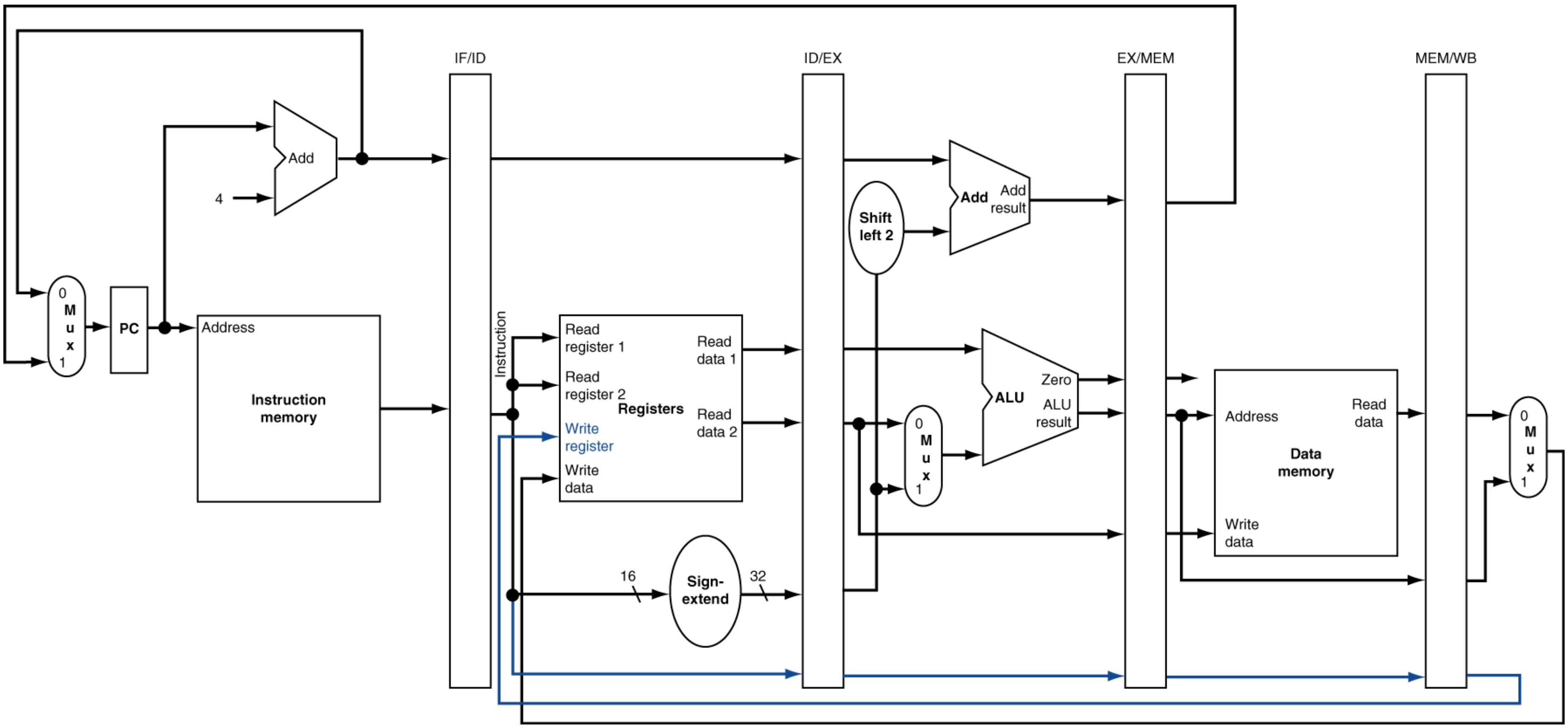
WB for Load

But the Write register port is now seeing the register number from a different, later instruction.



Since load instruction, value from data memory is selected and passed back to register file.

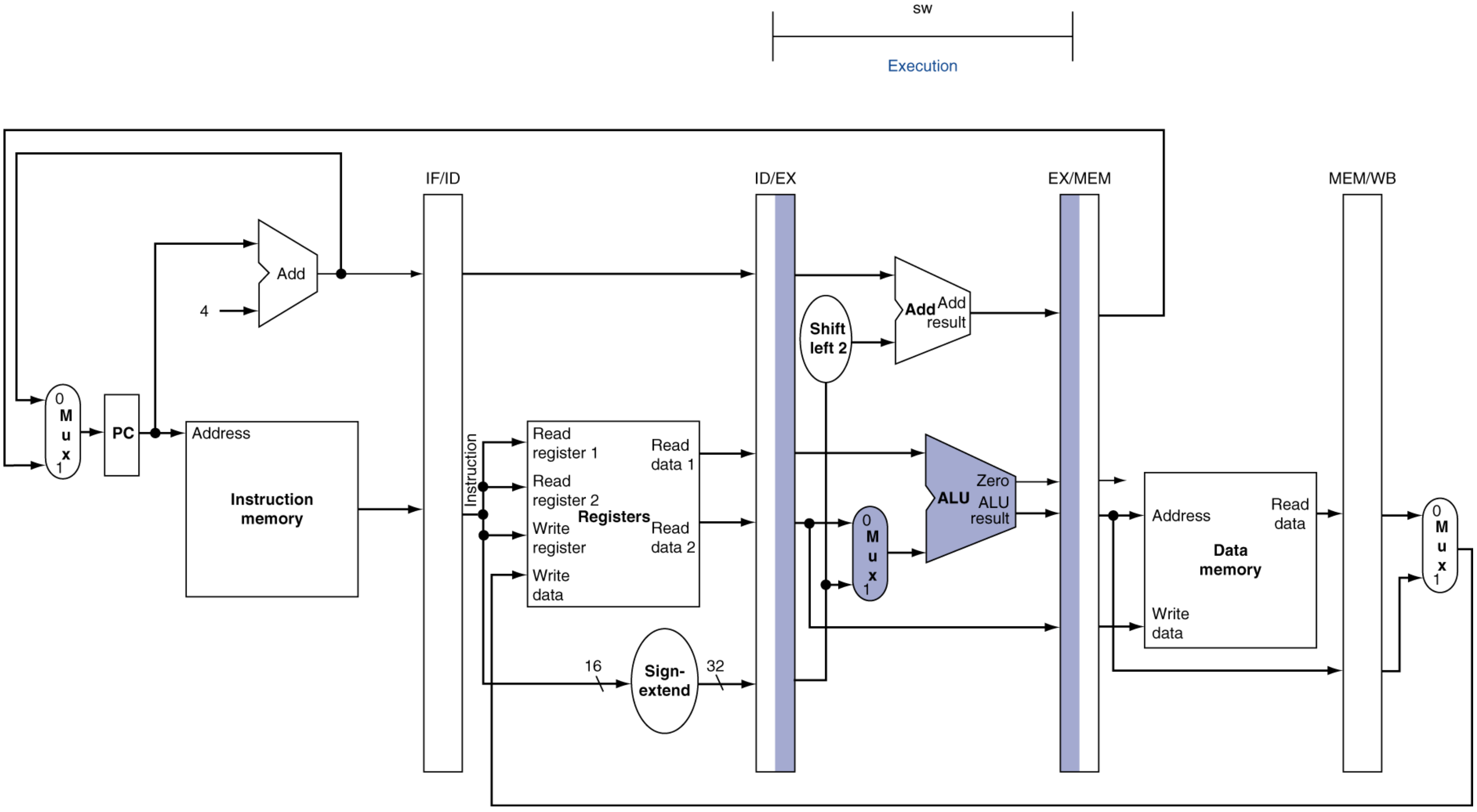
Corrected Datapath for Load



So we fix the register number problem by passing the Write register # from the load instruction through the various inter-stage buffers…

…and then back, on the correct clock cycle.

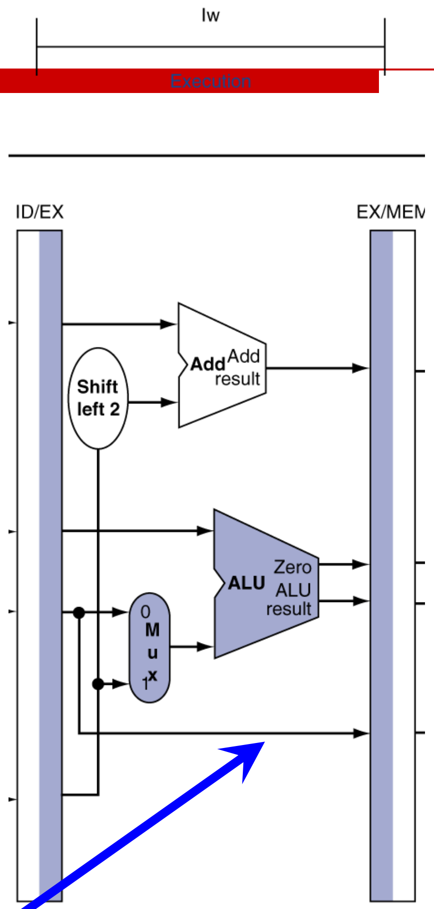
EX for Store



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
Zero
ALU
ALU result
Address
Data memory
Write data
Read data

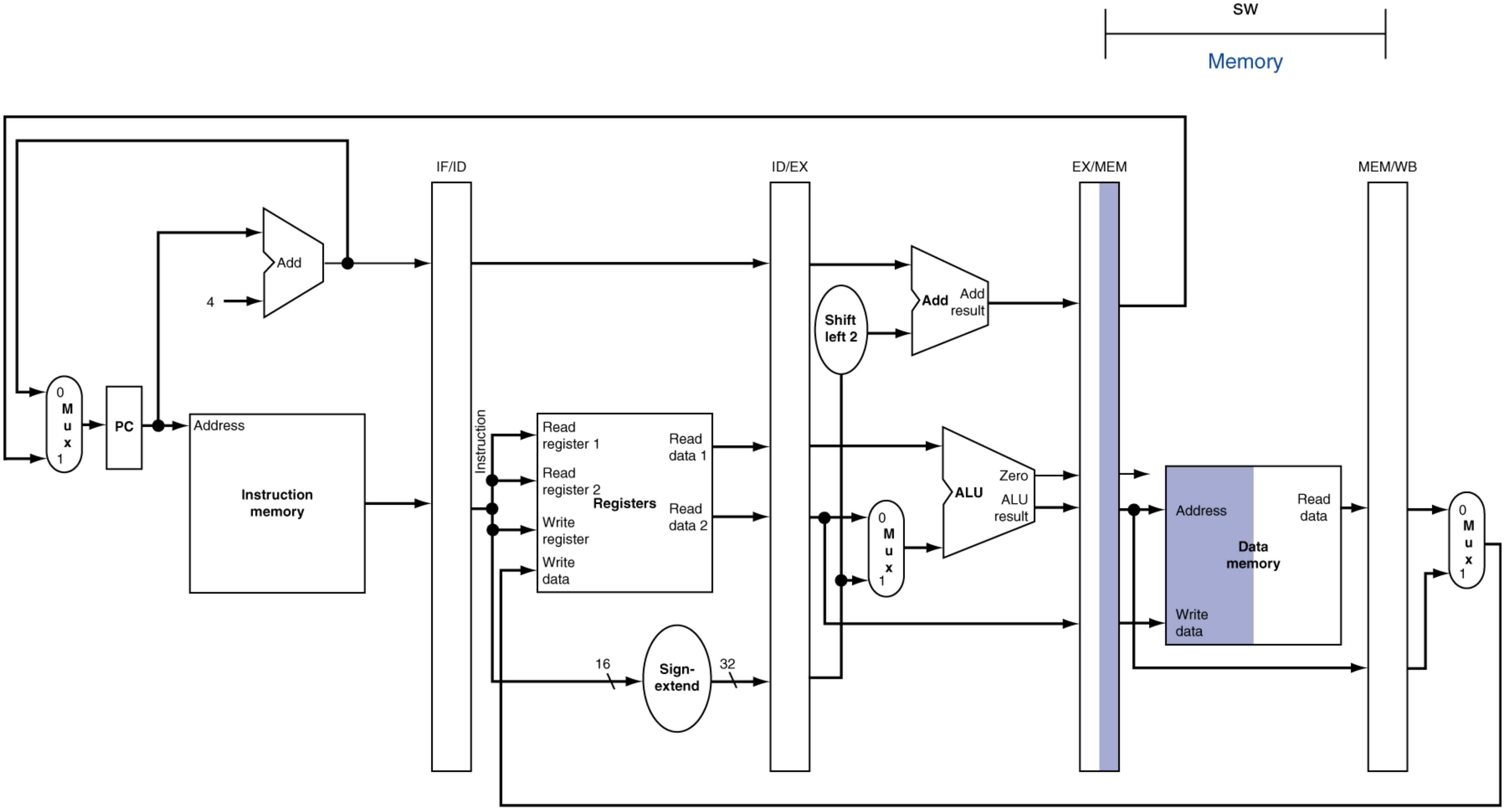
EX for Store

Almost the same as for LW…



Read register #2 is passed forward to EX/MEM buffer, for use in later stage… for SW this will be needed.

MEM for Store



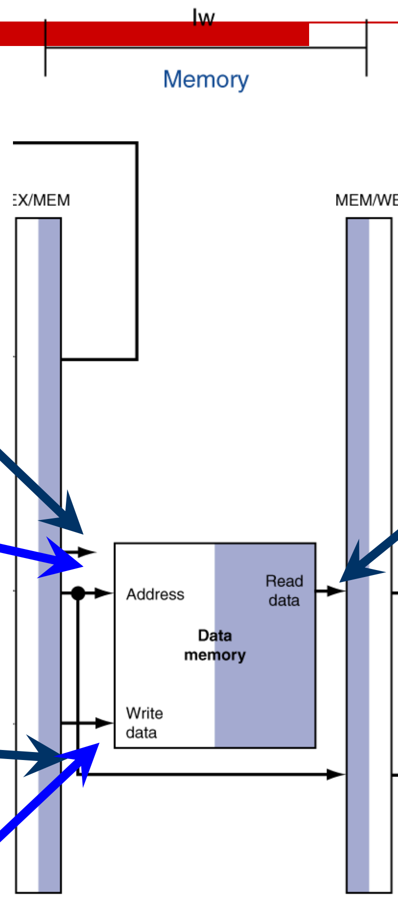
MEM for Store

Zero line taken from EX/MEM buffer for branch control logic in this stage…

ALU result is taken from EX/MEM buffer and passed to Address port of data memory.

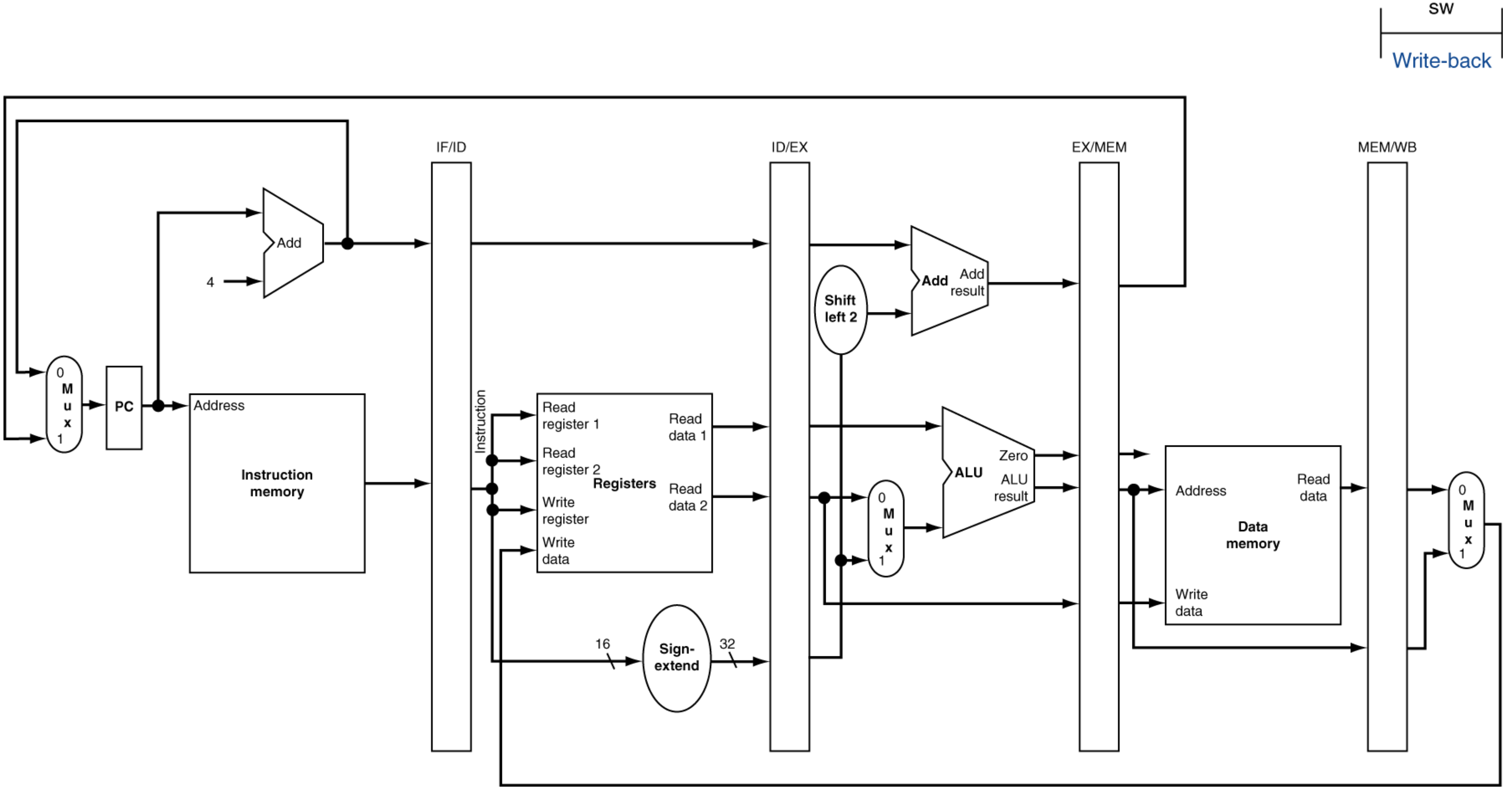
ALU result also stored in MEM/WB buffer for possible use in last stage…

Read register #2 contents taken from EX/MEM buffer and passed to Write data port of data memory.

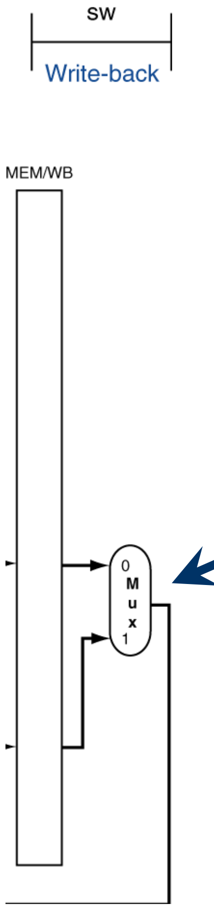


Value on Read data port of data memory is stored in MEM/WB buffer, awaiting decision in last stage..

WB for Store


SW
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
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32

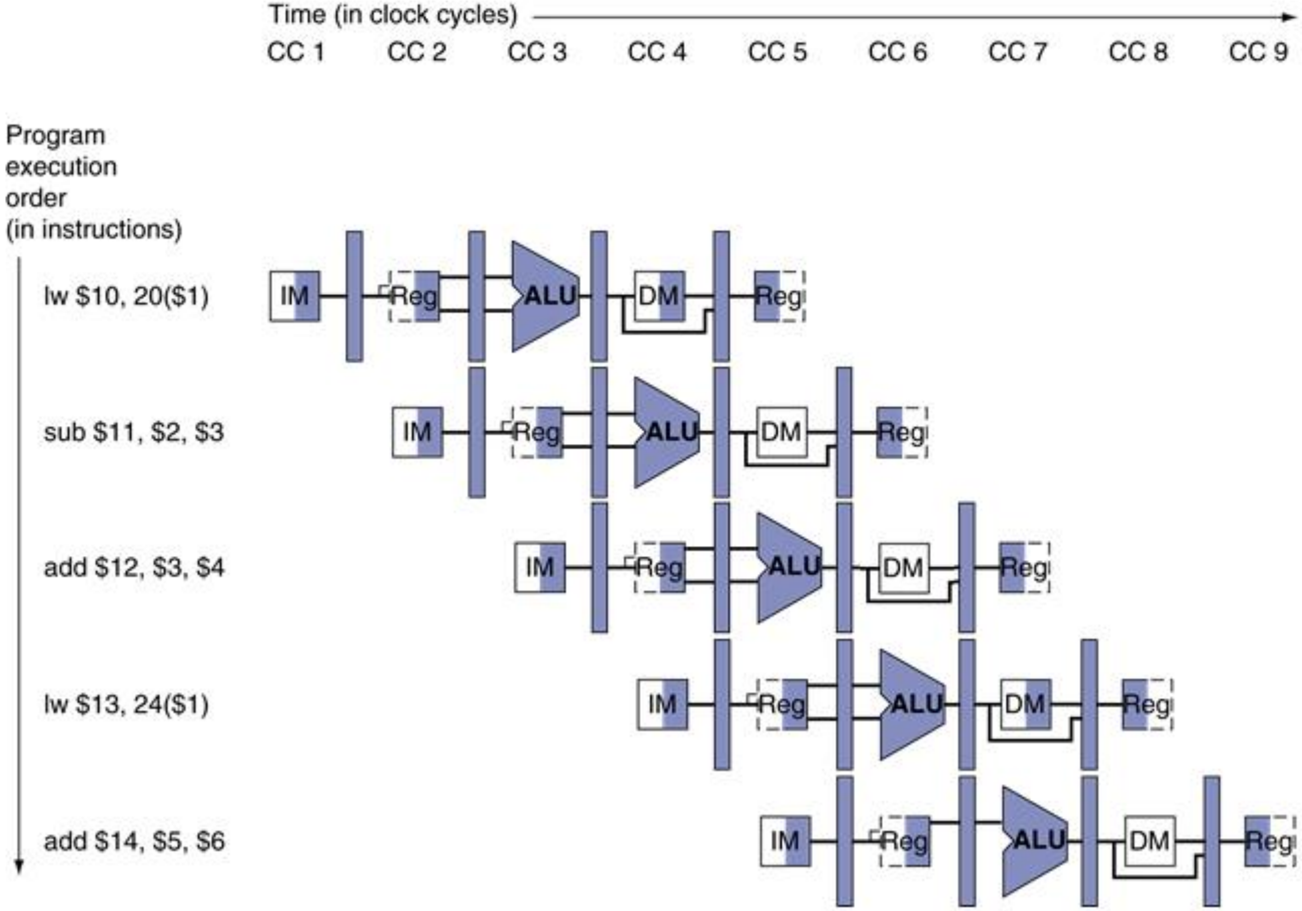
WB for Store



Since SW instruction, neither value will be written to the register file… doesn't really matter which value we send back…

Multi-Cycle Pipeline Diagram

- Form showing resource usage



Multi-Cycle Pipeline Diagram

- Traditional form

Time (in clock cycles) →

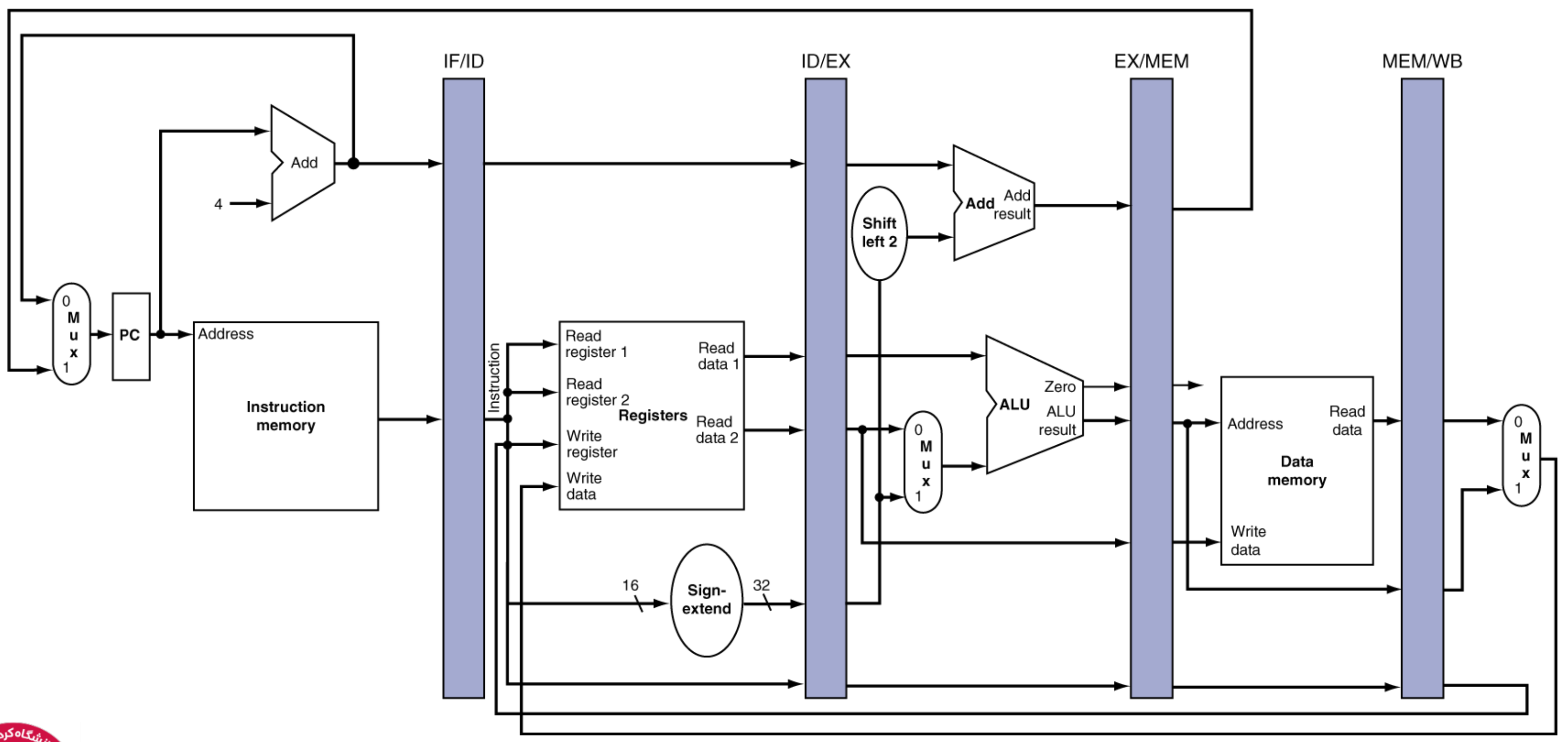
Program execution order (in instructions) ↓

	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

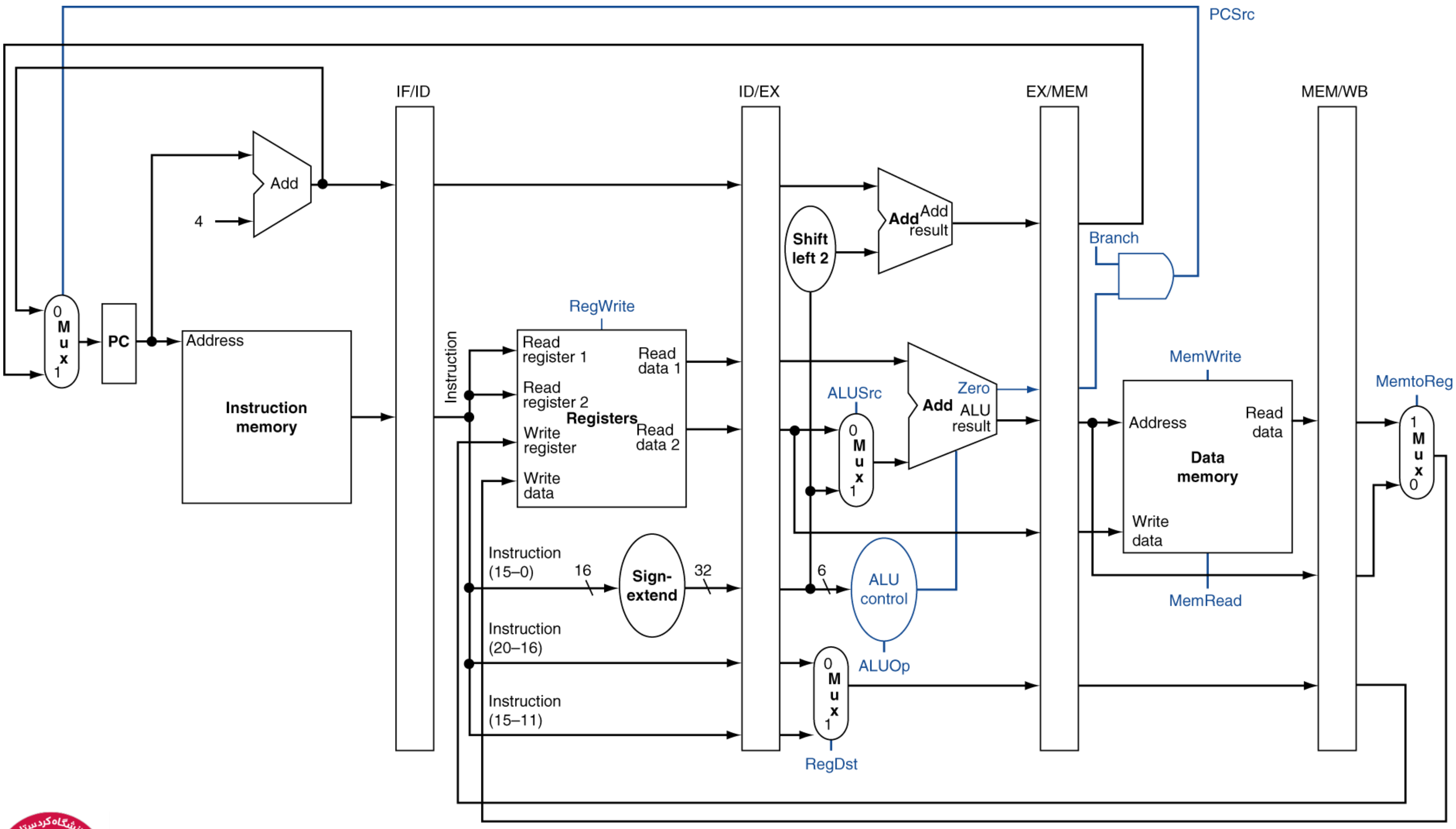
Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

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

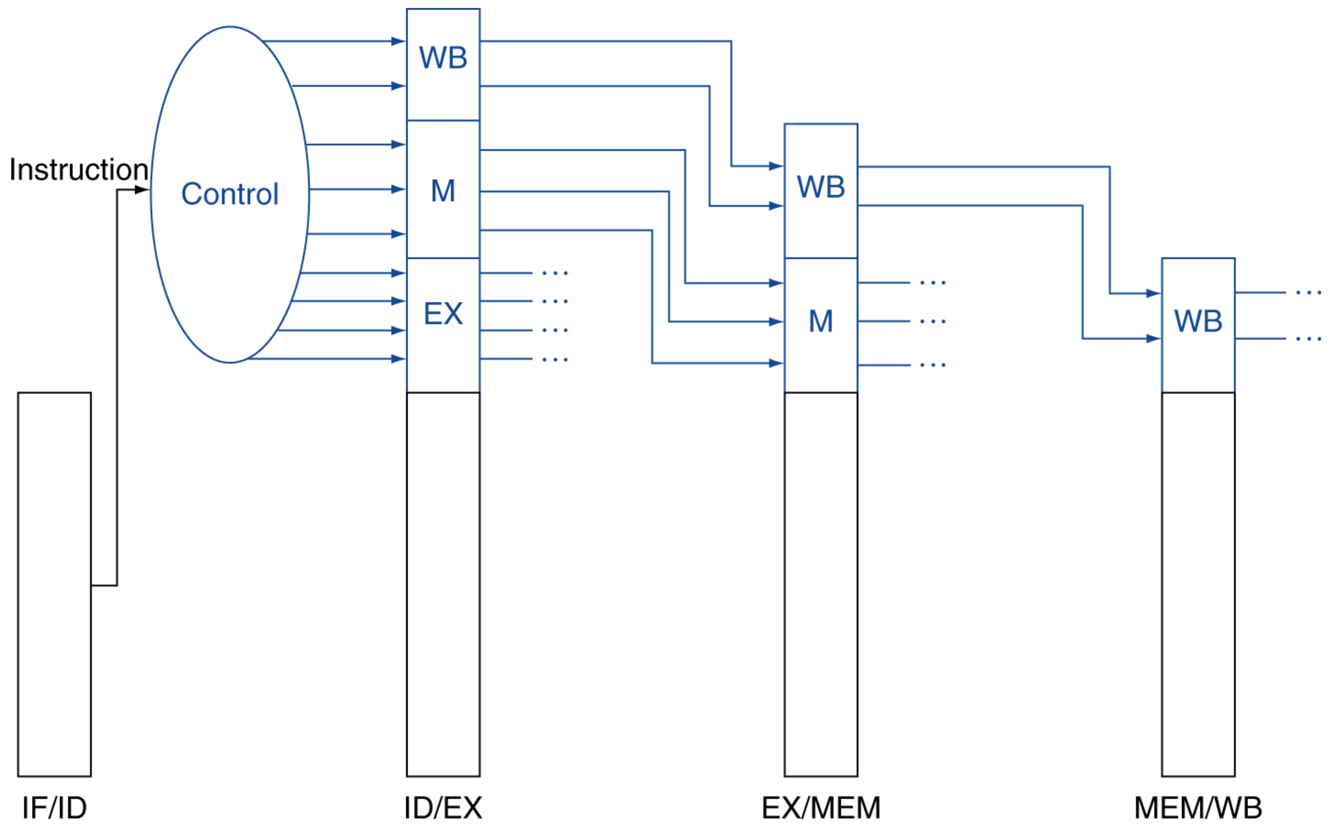


Pipelined Control (Simplified)

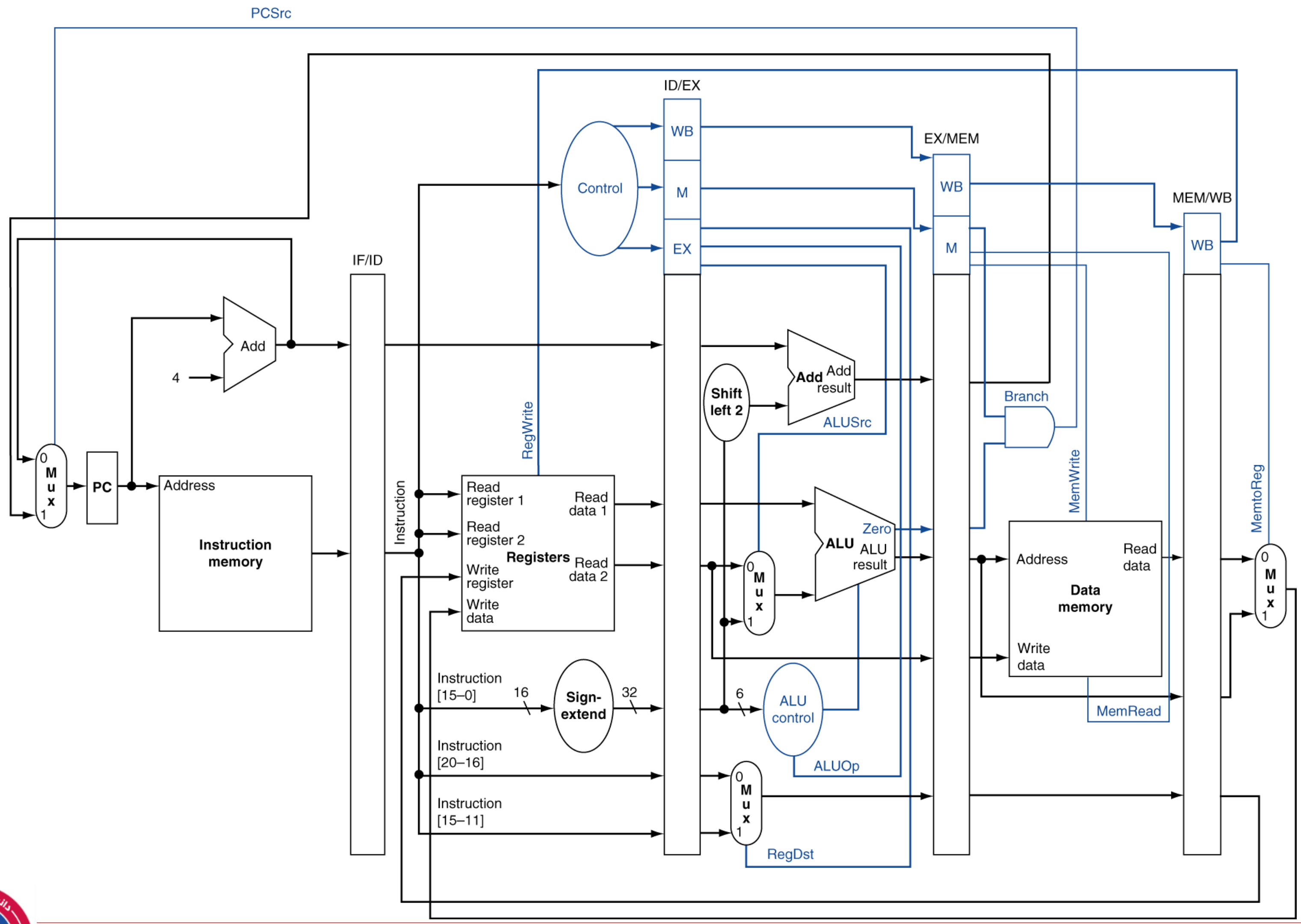


Pipelined Control

- Control signals derived from instruction (as in single-cycle implementation)



Pipelined Control



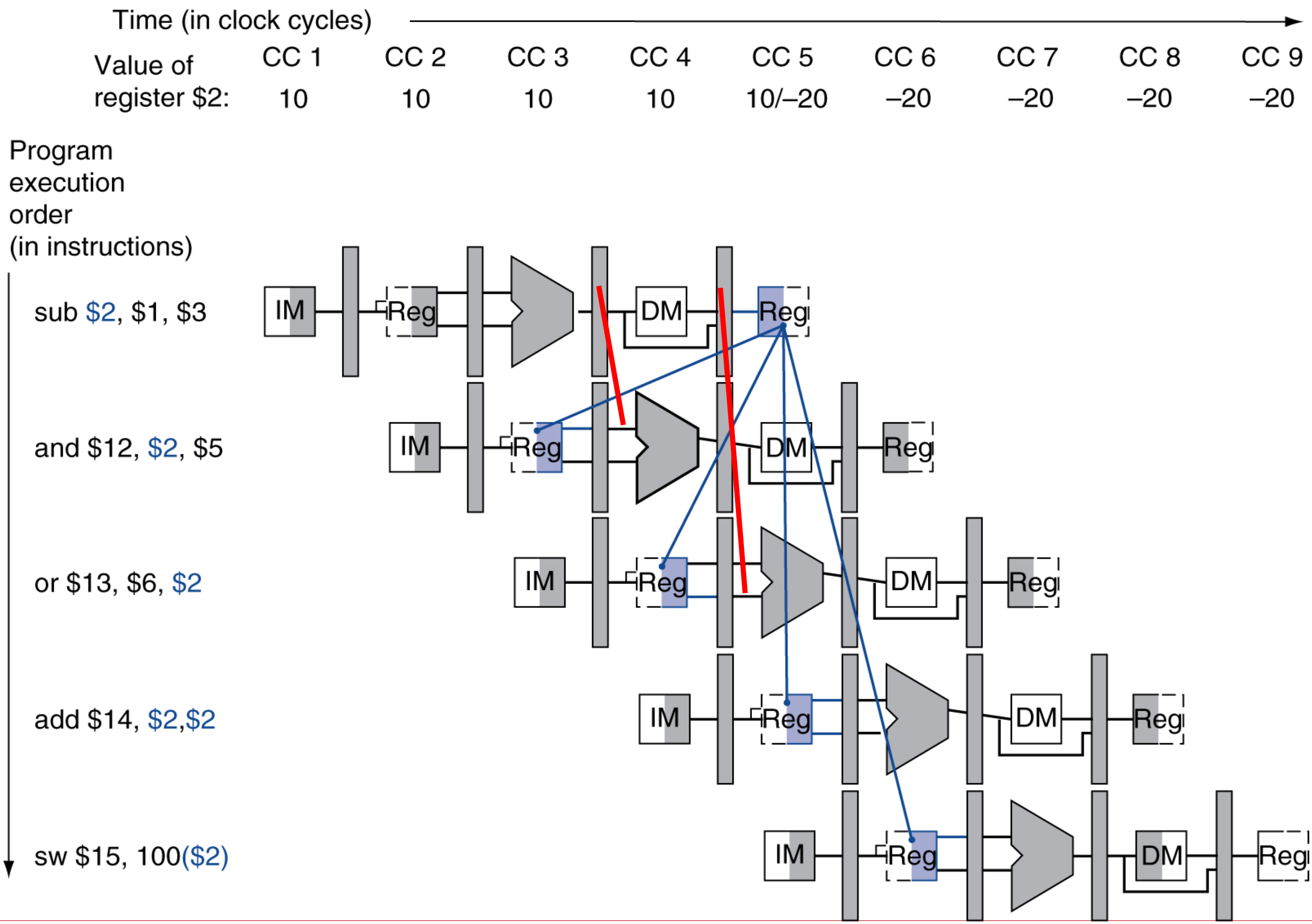
Data Hazards in ALU Instructions

- 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



Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., 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
- 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

First hazard between sub $2, $1, $3 and and $12, $2, $5 is detected when “and” is in EX and “sub” is in MEM because

EX/MEM.RegisterRd = ID/EX.RegisterRs = $2 (1a)

Similar to above this time dependency between “sub” and “or” can be detected as

MEM/WB.RegisterRd = ID/EX.RegisterRt = $2 (2b)

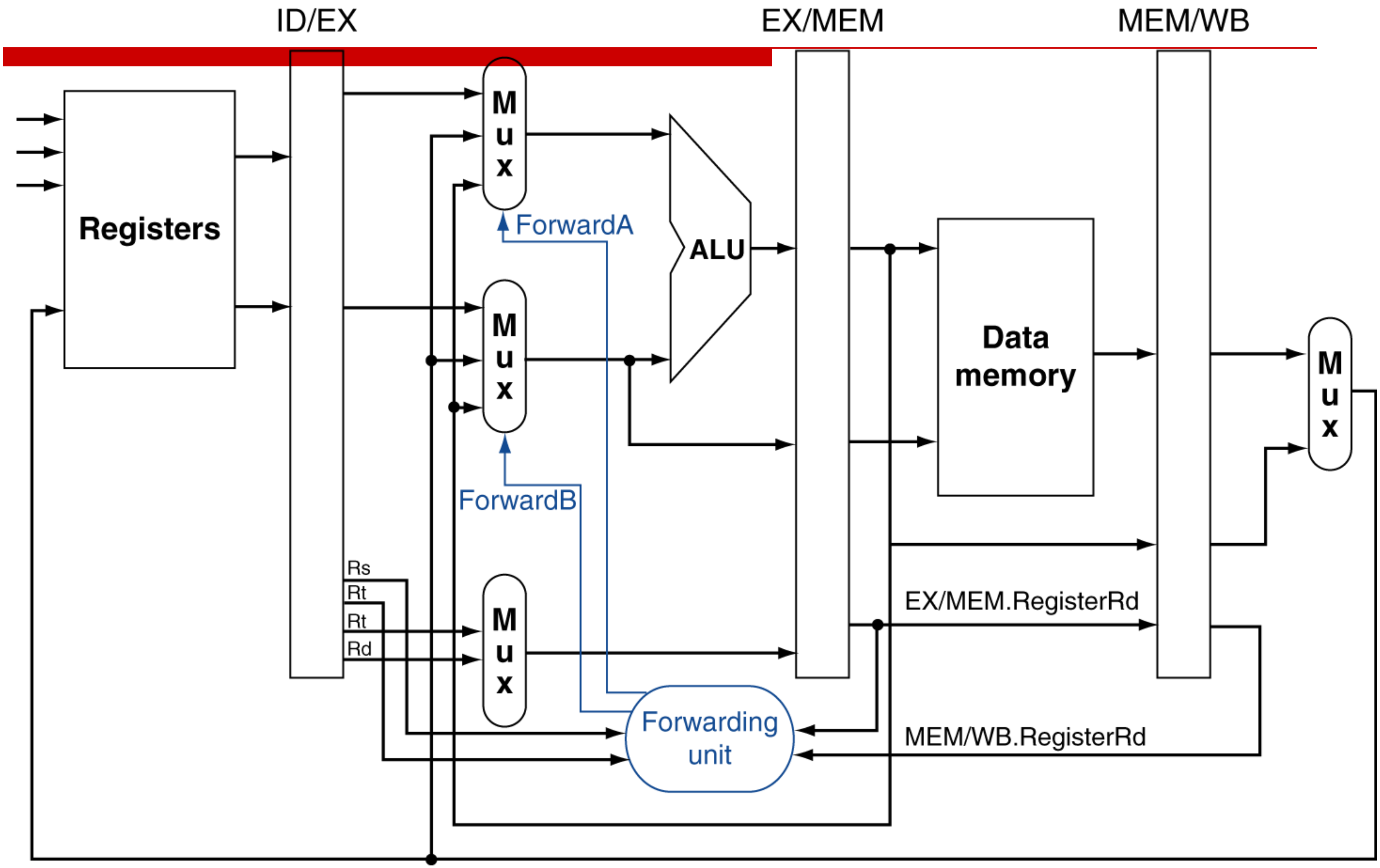
Two dependencies between “sub” and ”add” are not hazard Another form of forwarding but it occurs within reg file

There is no hazard between “sub” and “sw”

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 Paths



b. With forwarding

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

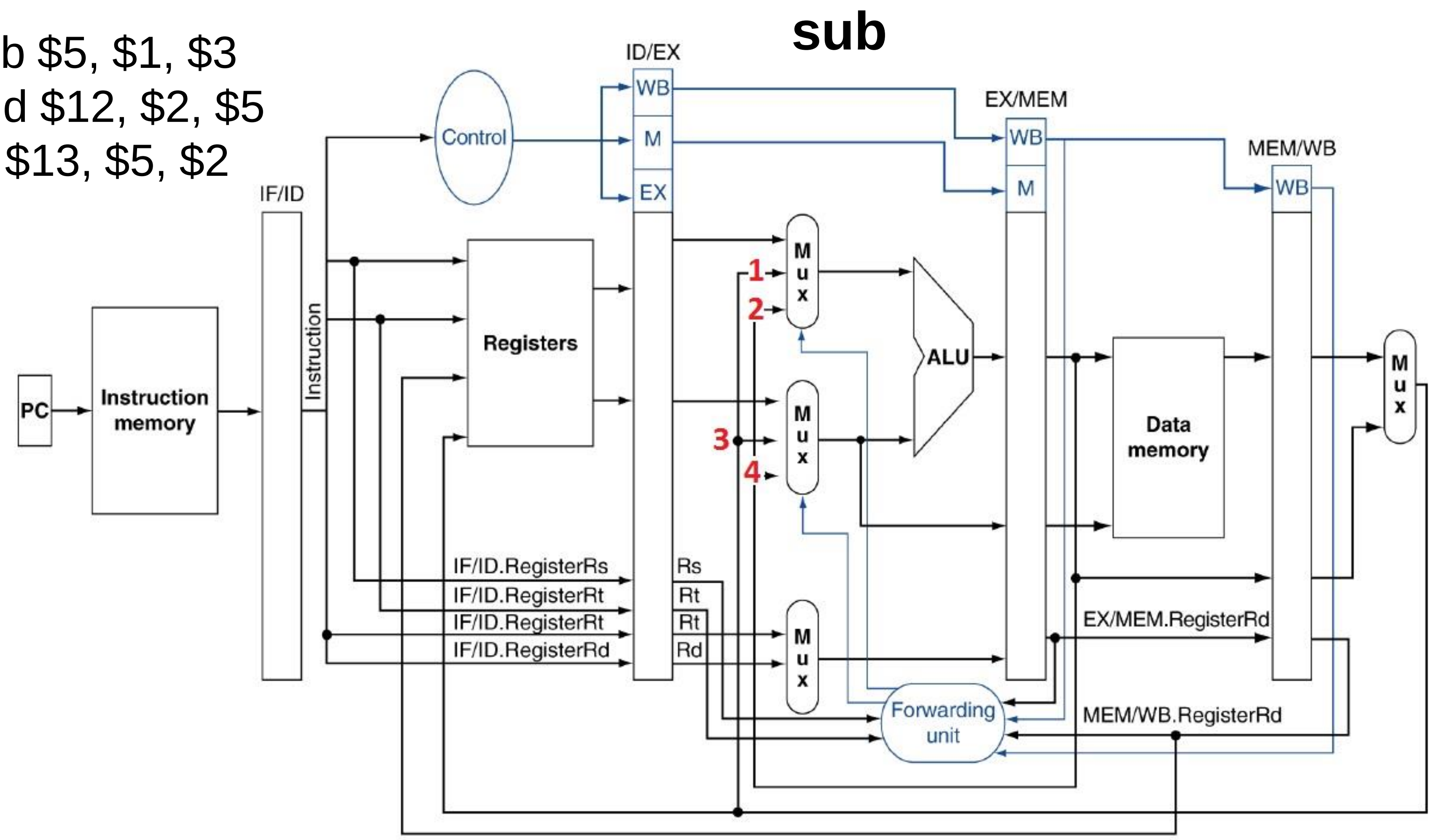
Forwards the result from the previous instr. to either input of the ALU

- 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

Forwards the result from the second previous instr. to either input of the ALU

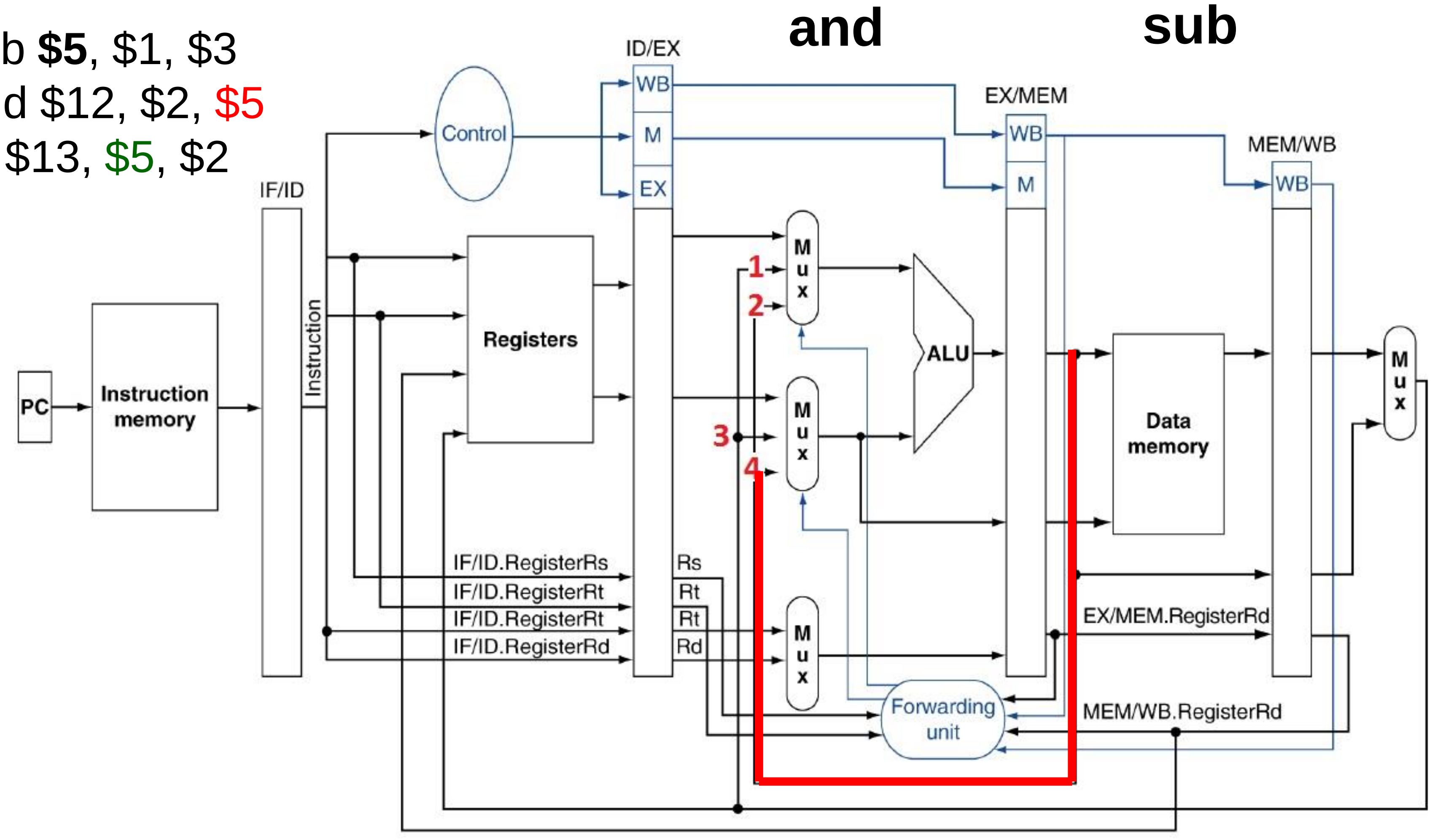
Forwarding Example

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



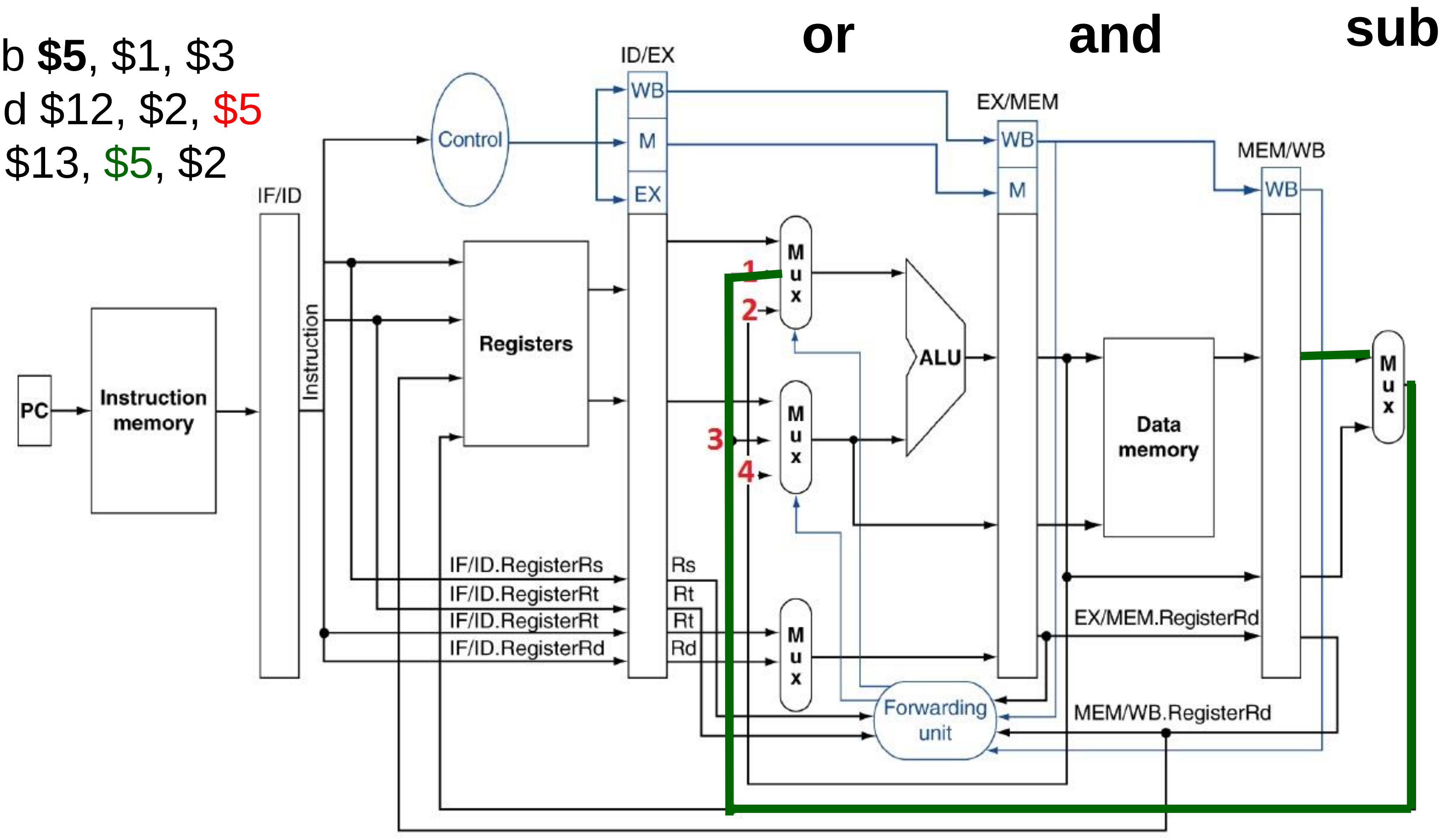
Forwarding Example

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



Forwarding Example

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



Double Data Hazard

- Consider the sequence:

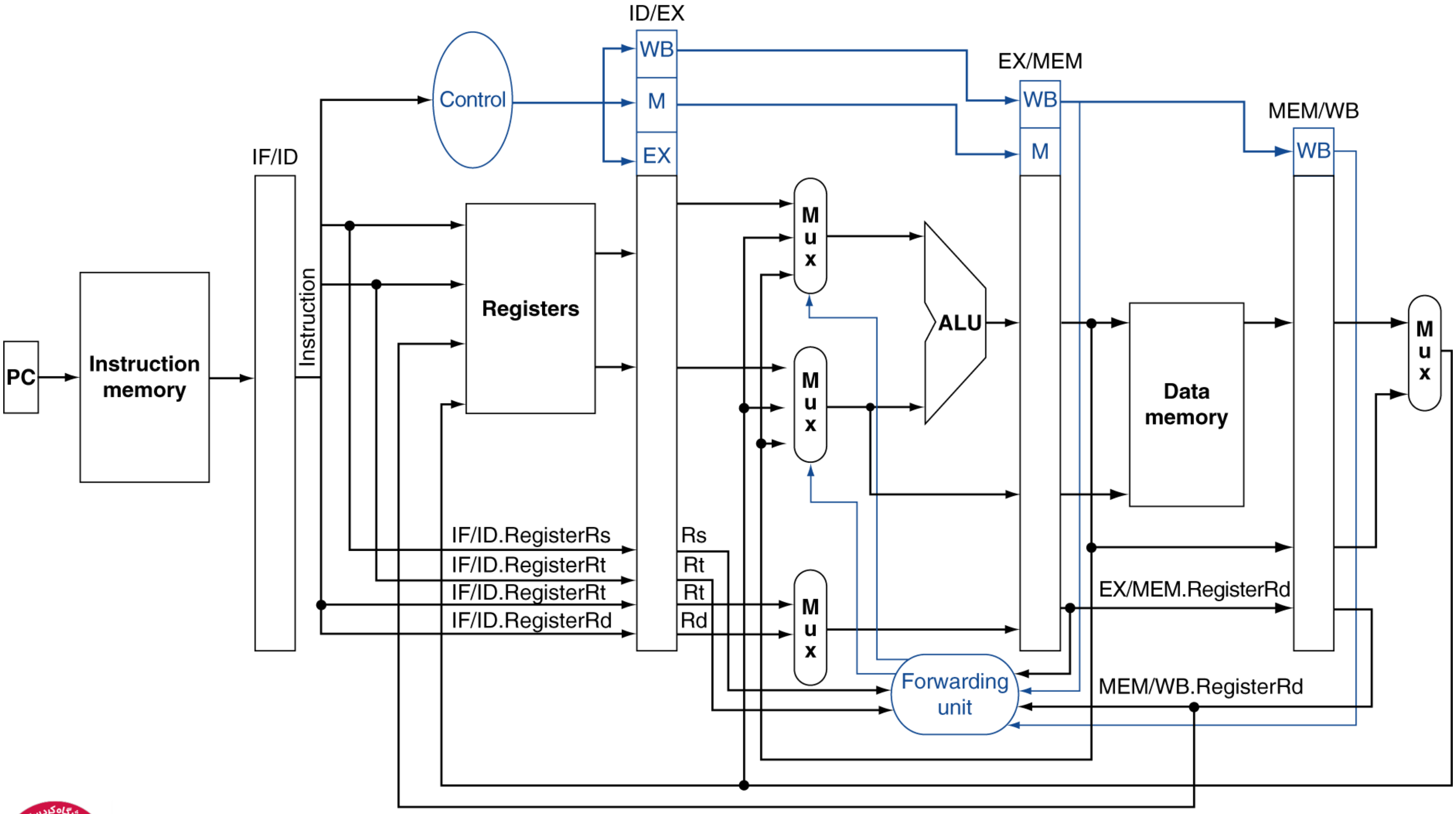
```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

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

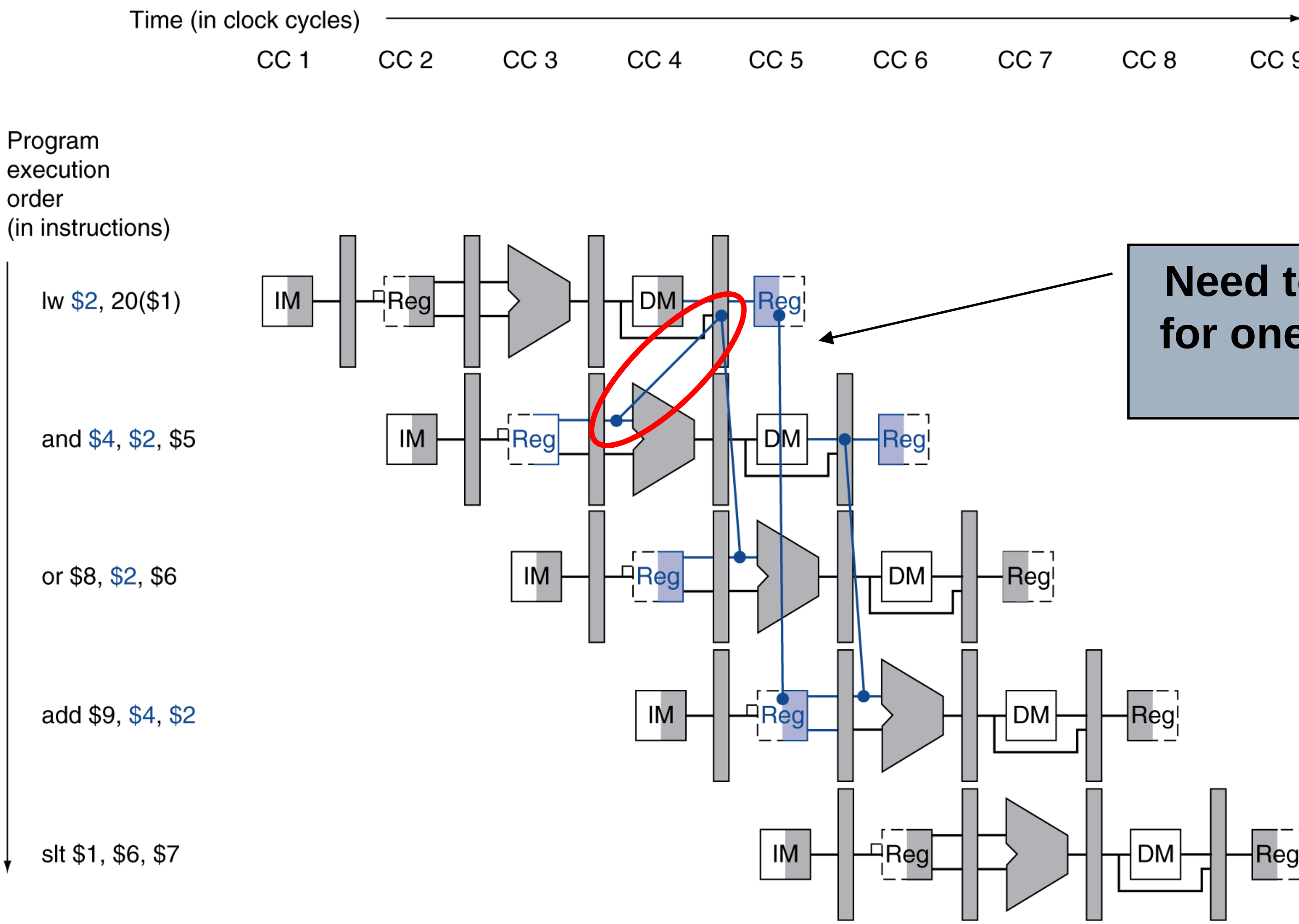
Revised Forwarding Condition

- 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



Need to stall for one cycle

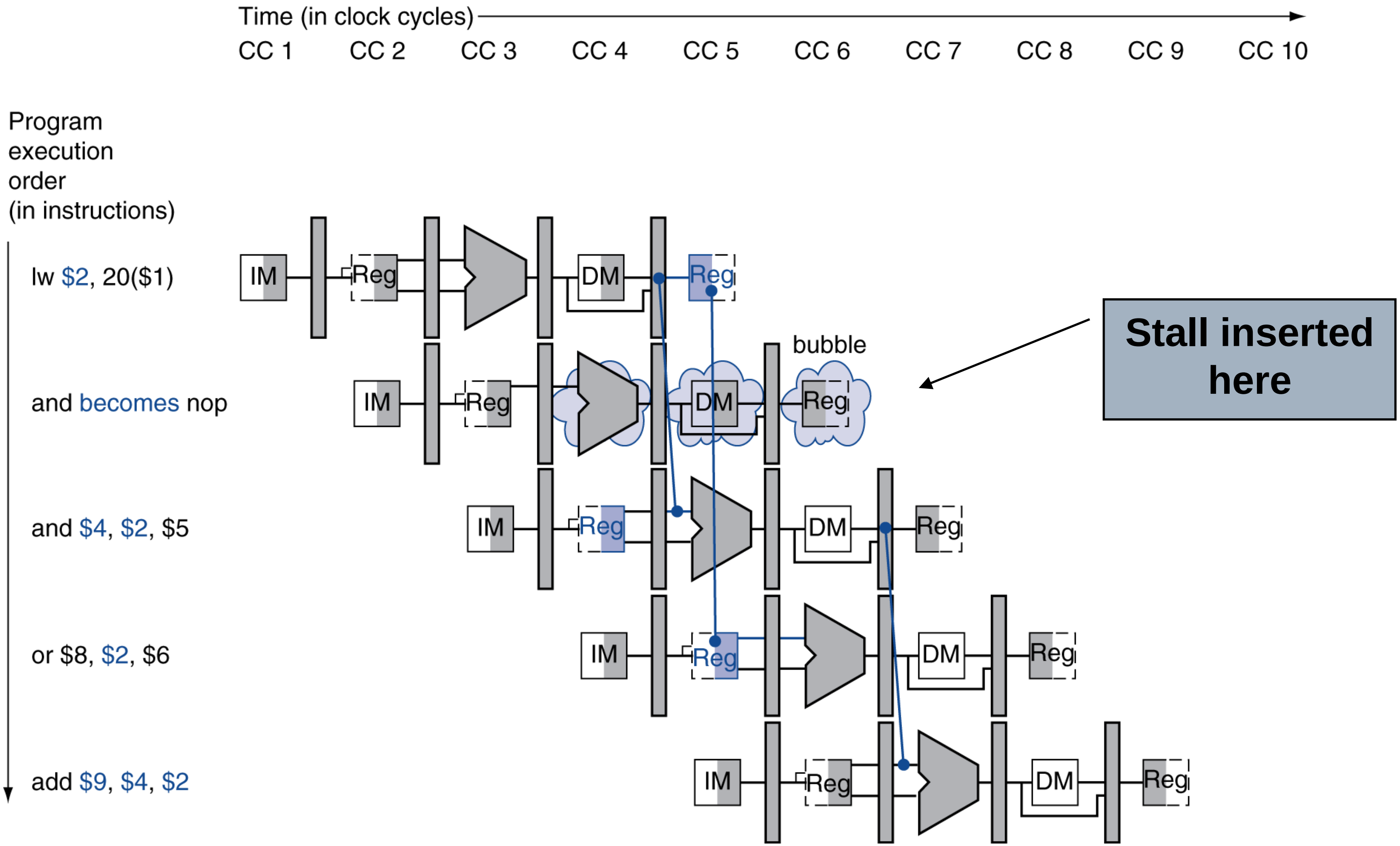
Load-Use Hazard Detection

- 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

- 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
  - Using (current) 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

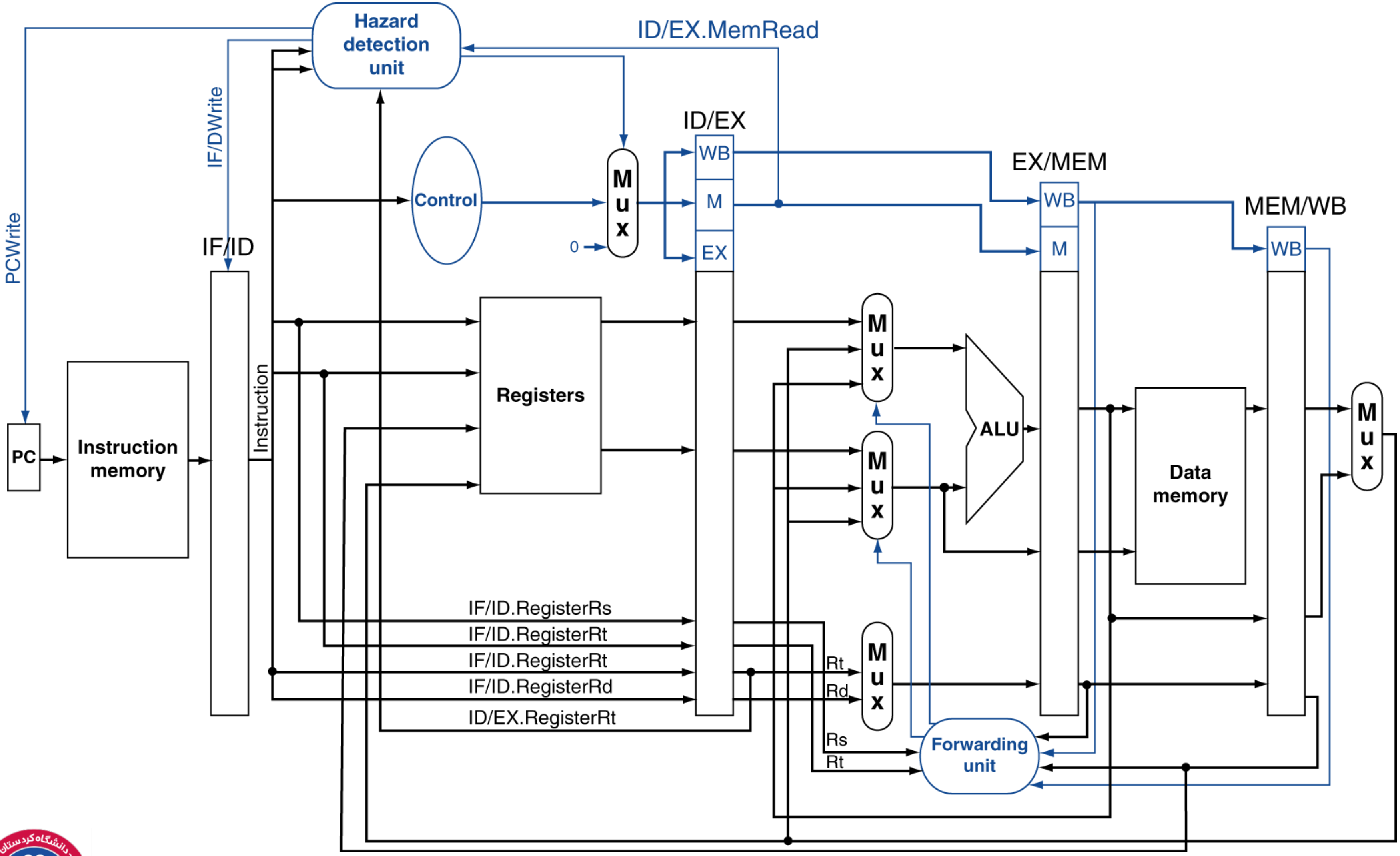
Stall/Bubble in the Pipeline



Stall Hardware

- Along with the Hazard Unit, we have to implement the stall
- Prevent the instructions in the IF and ID stages from progressing down the pipeline – done by preventing the PC register and the IF/ID pipeline register from changing
  - Hazard detection Unit controls the writing of the PC (`PC.write`) and IF/ID (`IF/ID.write`) registers
- Insert a "bubble" between the `lw` instruction (in the EX stage) and the load-use instruction (in the ID stage) (i.e., insert a `noop` in the execution stream)
  - Set the control bits in the EX, MEM, and WB control fields of the ID/EX pipeline register to 0 (`nop`). The Hazard Unit controls the mux that chooses between the real control values and the 0's.
- Let the `lw` instruction and the instructions after it in the pipeline (before it in the code) proceed normally down the pipeline

Datapath with Hazard Detection



Pipeline with and without forwarding

Instructions	1	2	3	4	5	6	7	8	9	0	1	2
sub \$2, \$3, \$1	F	D	X	M	W							
lw \$5, 0(\$2)		F	d*	d*	D	X	M	W				
addi \$4, \$5, 1					F	d*	d*	D	X	M	W	
add \$5, \$3, \$1								F	D	X	M	W

Now show what would happen if the pipeline had full bypassing:

Instructions	1	2	3	4	5	6	7	8	9	0	1	2
sub \$2, \$3, \$1	F	D	X	M	W							
lw \$5, 0(\$2)		F	D	X	M	W						
addi \$4, \$5, 1			F	d*	D	X	M	W				
add \$5, \$3, \$1					F	D	X	M	W			

Stalls and Performance

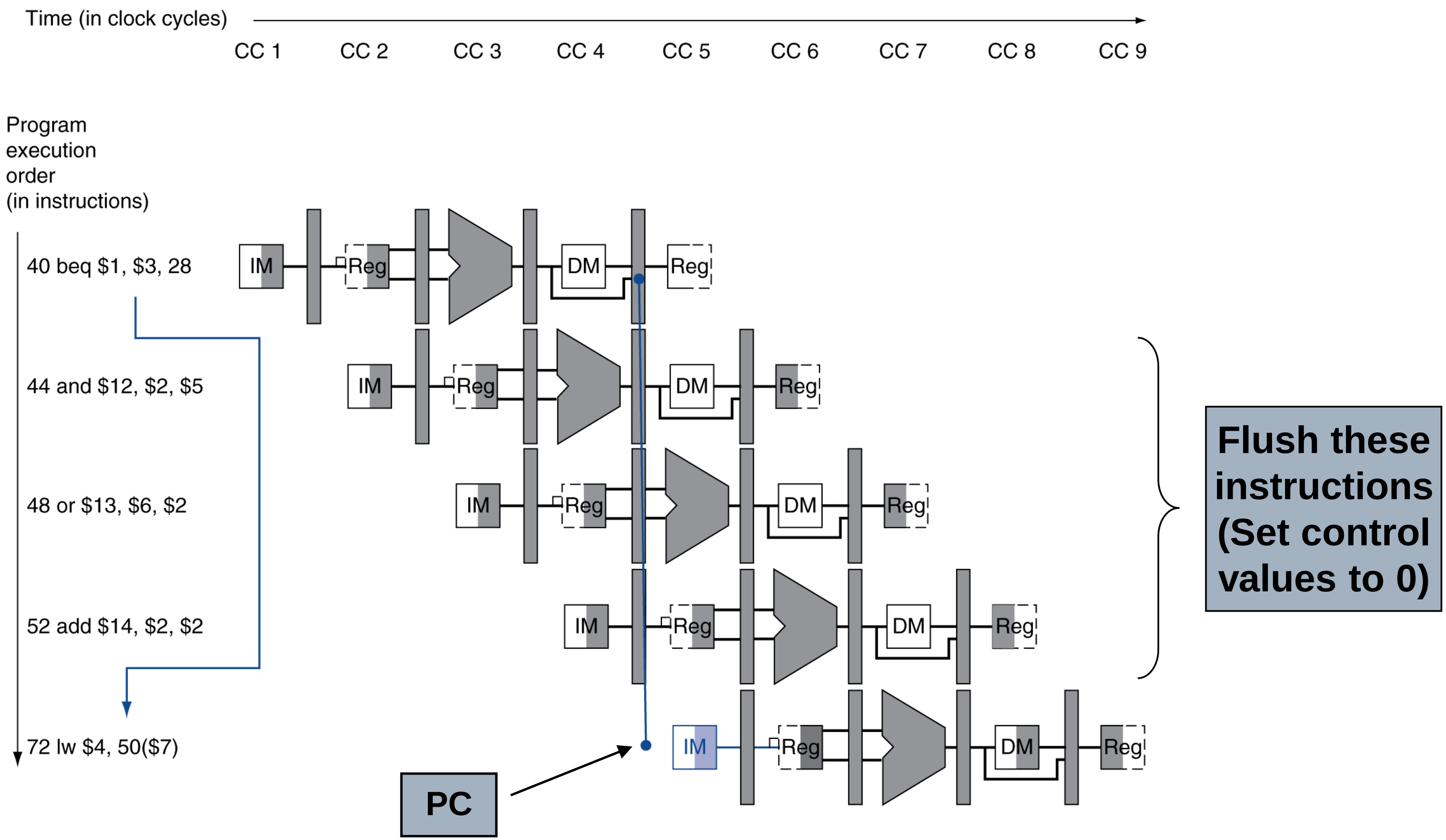
- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Control Hazards

- When the flow of instruction addresses is not sequential (i.e., PC = PC + 4); incurred by change of flow instructions
  - Conditional branches (`beq`, `bne`)
  - Unconditional branches (`j, jal, jr`)
  - Exceptions
- Possible approaches
  - Stall (impacts CPI)
  - Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  - Delay decision (requires compiler support)
  - Predict and hope for the best !
- Control hazards occur less frequently than data hazards, but there is *nothing* as effective against control hazards as forwarding is for data hazards

Branch Hazards

- If branch outcome determined in MEM

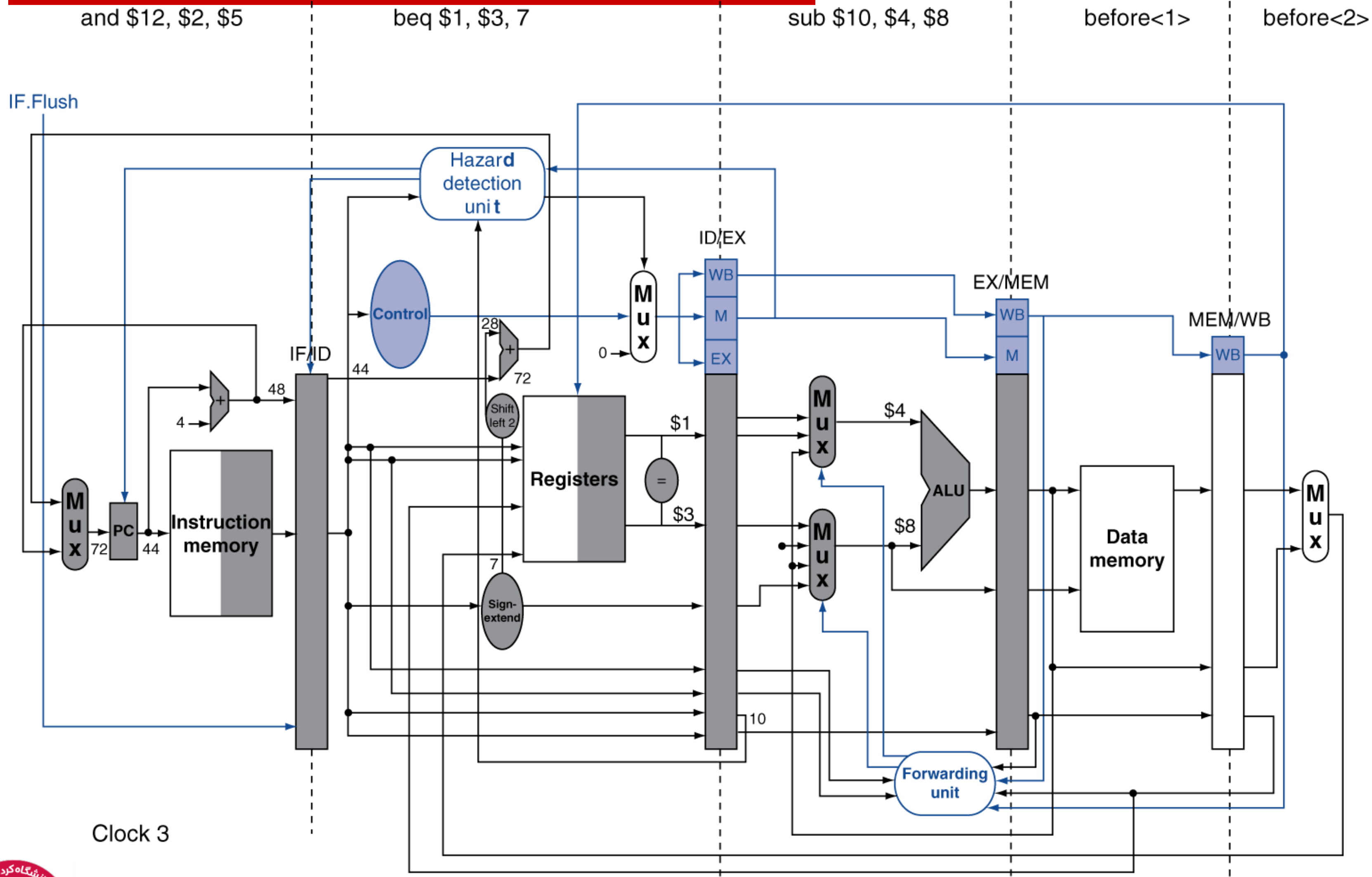


Reducing Branch Delay

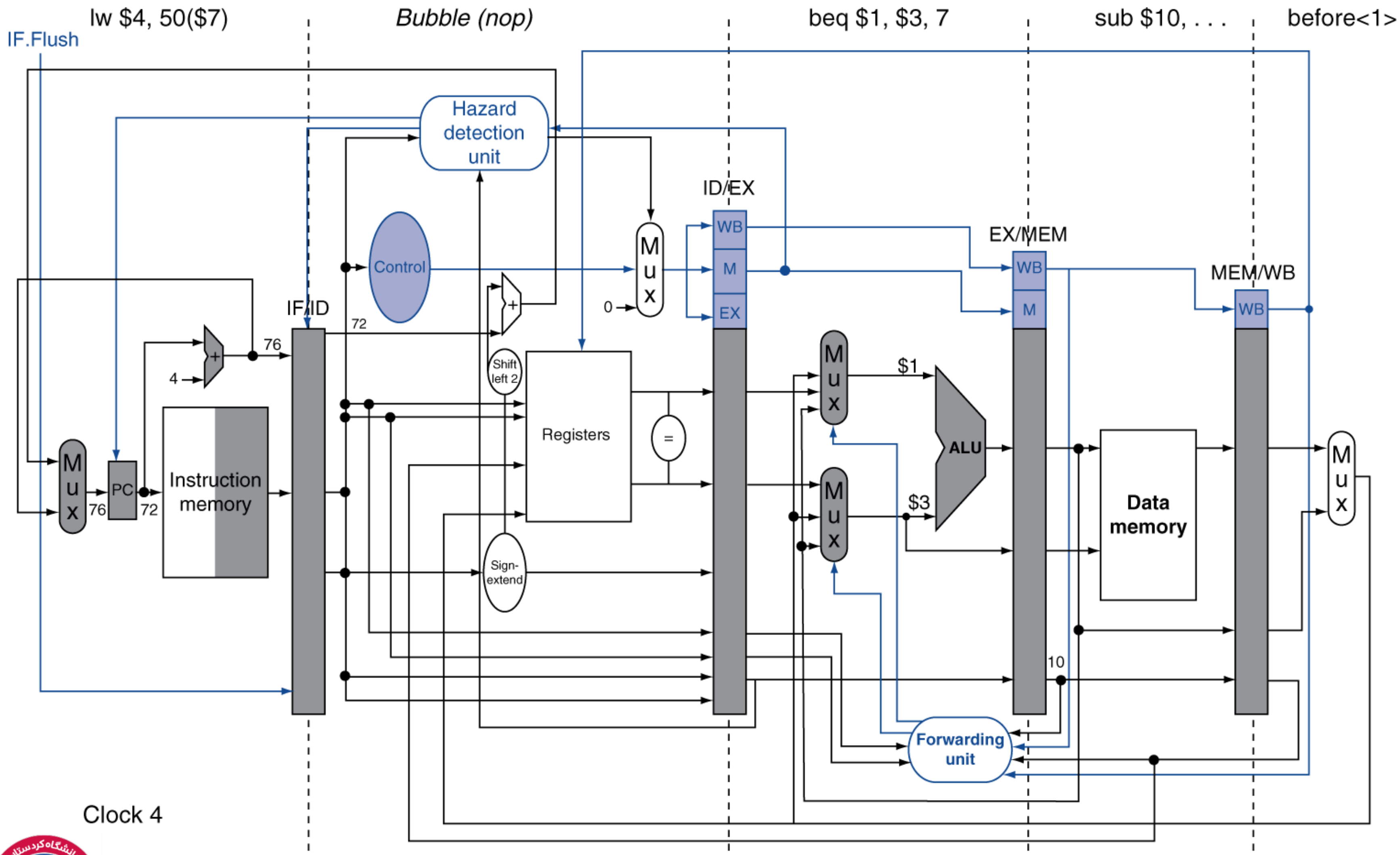
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:  sub  $10, $4, $8
40:  beq  $1,  $3, 7
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)   #44+7x4=72 (PC+4 + Imm*4)
```

Example: Branch Taken



Example: Branch Taken



Data Hazards for Branches

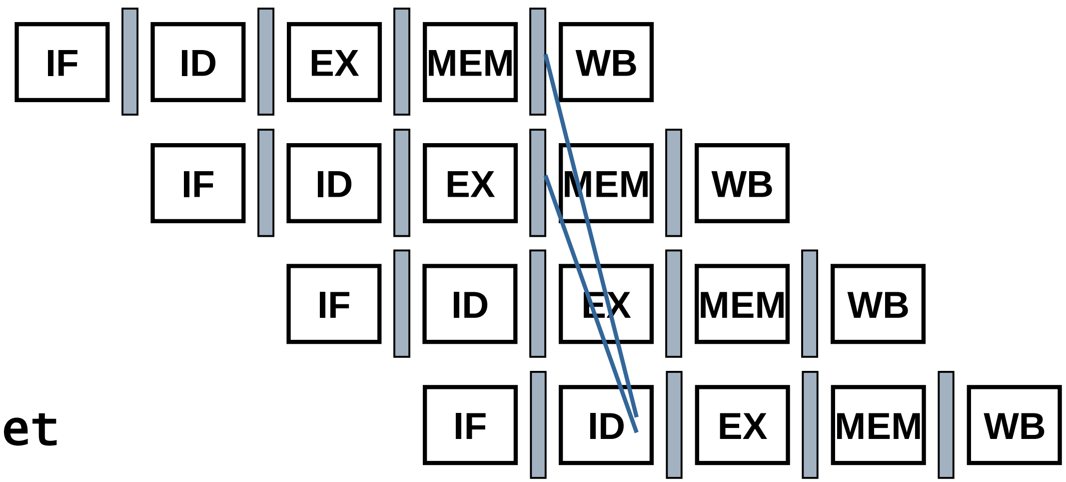
- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

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

add $4, $5, $6

…

beq $1, $4, target
```

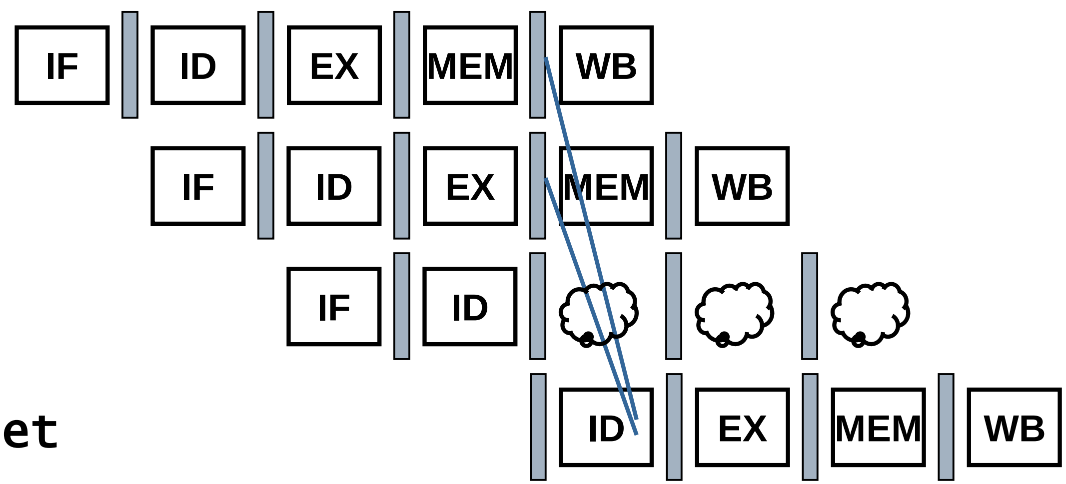


- Can resolve using forwarding

Data Hazards for Branches

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

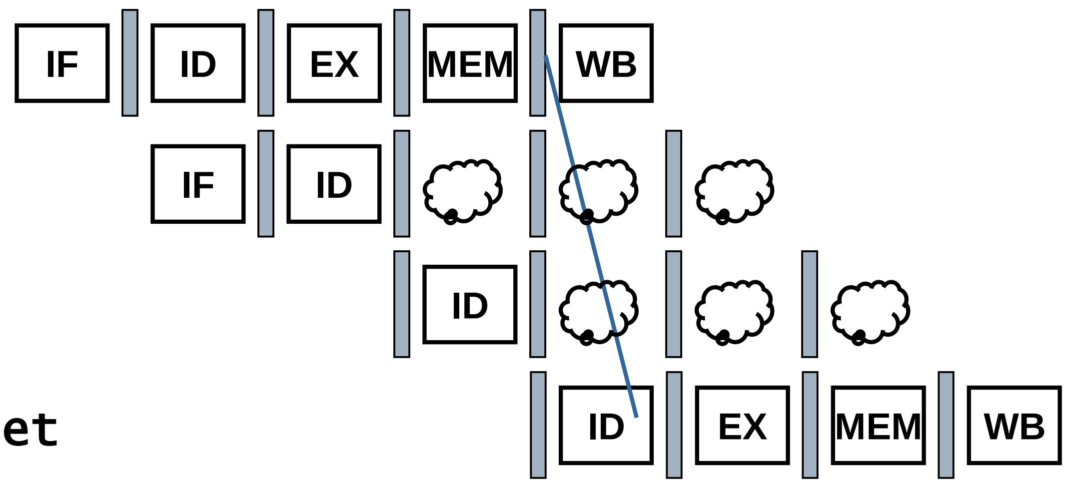
```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



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
```



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!

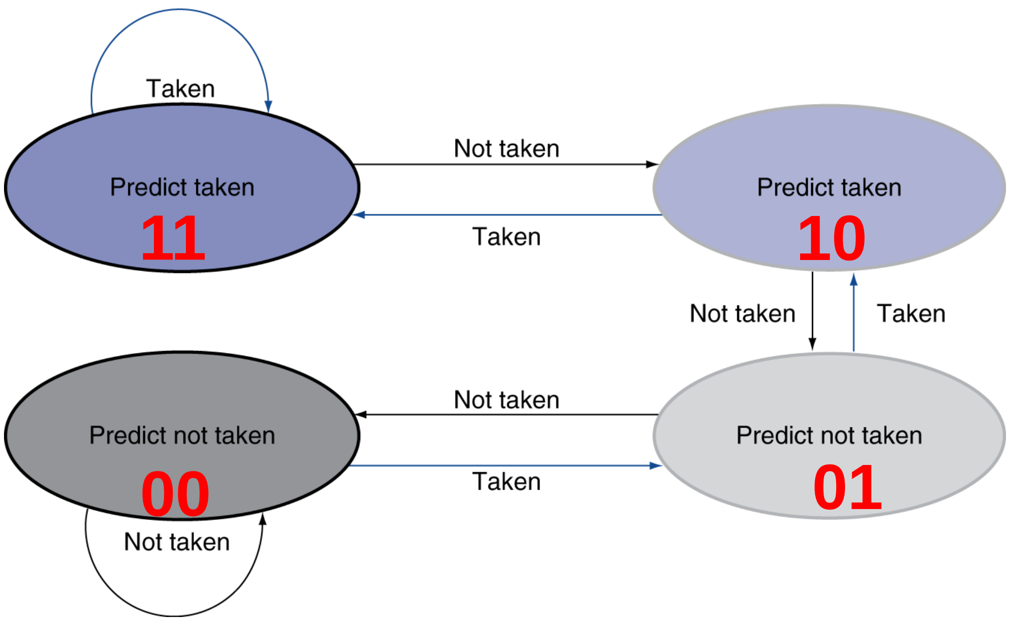
```
outer: …
       …
inner: …
       …
beq …, …, inner
       …
beq …, …, outer
```

- 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

Calculate the correct prediction rate for the below branch. Assume starting from state:**00**



Actual N N N T T T T T T N N T T T T N N T T T T T T T T

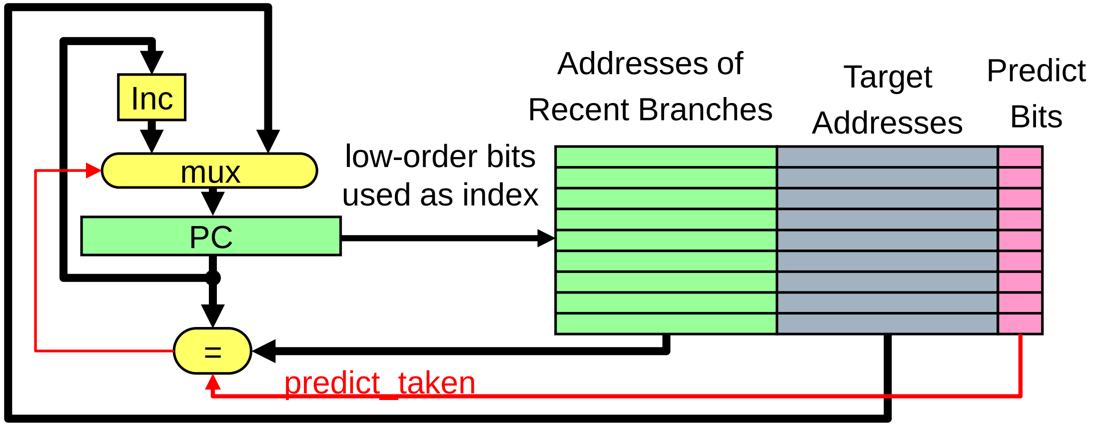
Next state 00 00 00 01 10 11 11 11 11 10 01 10 11 11 11 10 01 10 11 11 11 11 11 11

Prediction N N N N N T T T T T T N T T T T T N T T T T T T T

✓ ✓ ✓ ✗ ✗ ✓ ✓ ✓ ✓ ✗ ✗ ✗ ✓ ✓ ✓ ✗ ✗ ✗ ✓ ✓ ✓ ✓ ✓ ✓ ✓

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



Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

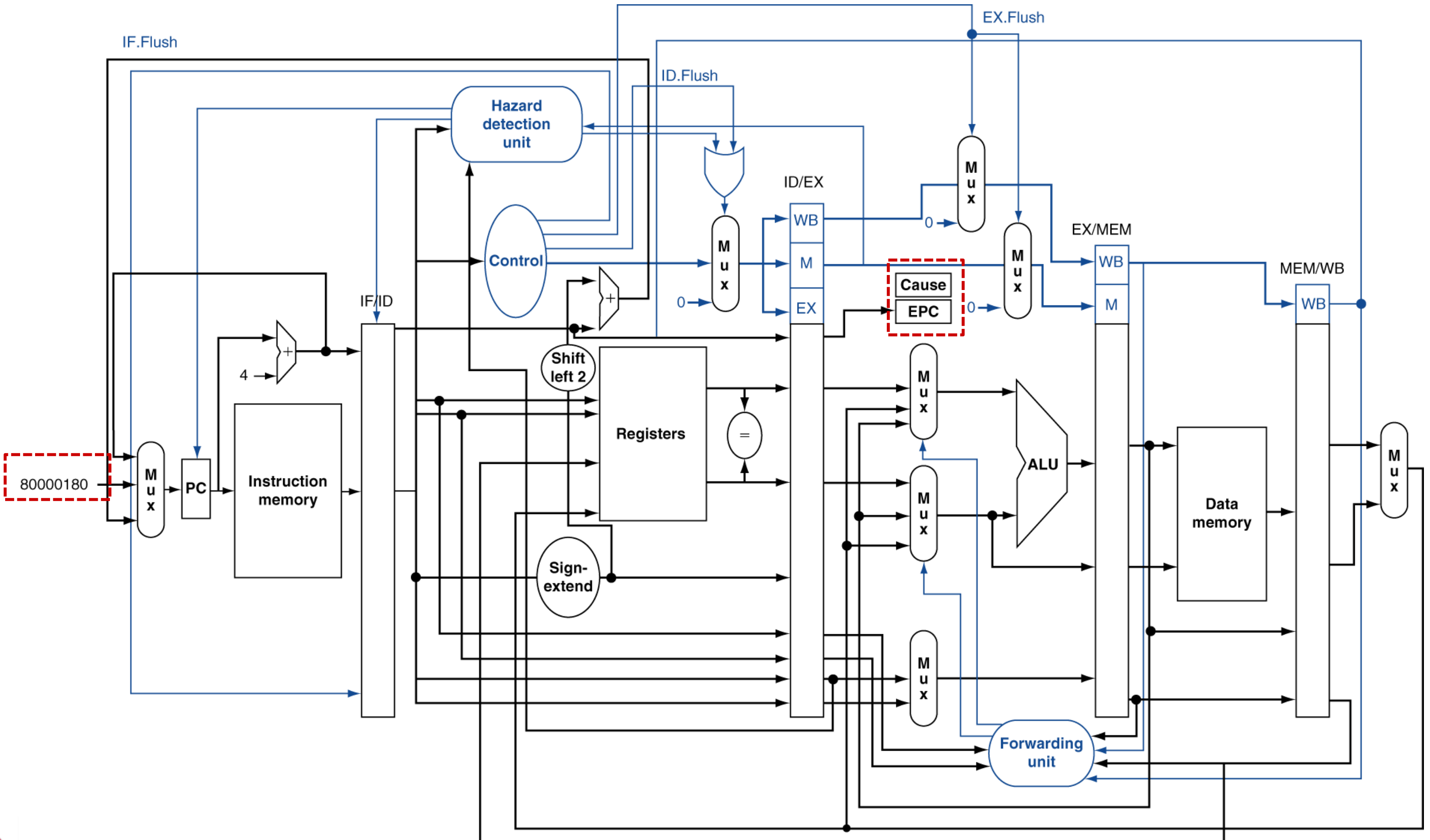
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage

  `add $1, $2, $1`
  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions

- New input value for PC holds the initial address to fetch instruction from in the event of an exception.
- A Cause register to record the cause of the exception.
- An EPC register to save the address of the instruction to which we should return.



Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - **Deeper pipeline**
    - Less work per stage $\Rightarrow$ shorter clock cycle
  - **Multiple issue**
    - Replicate pipeline stages $\Rightarrow$ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - $\Rightarrow$ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

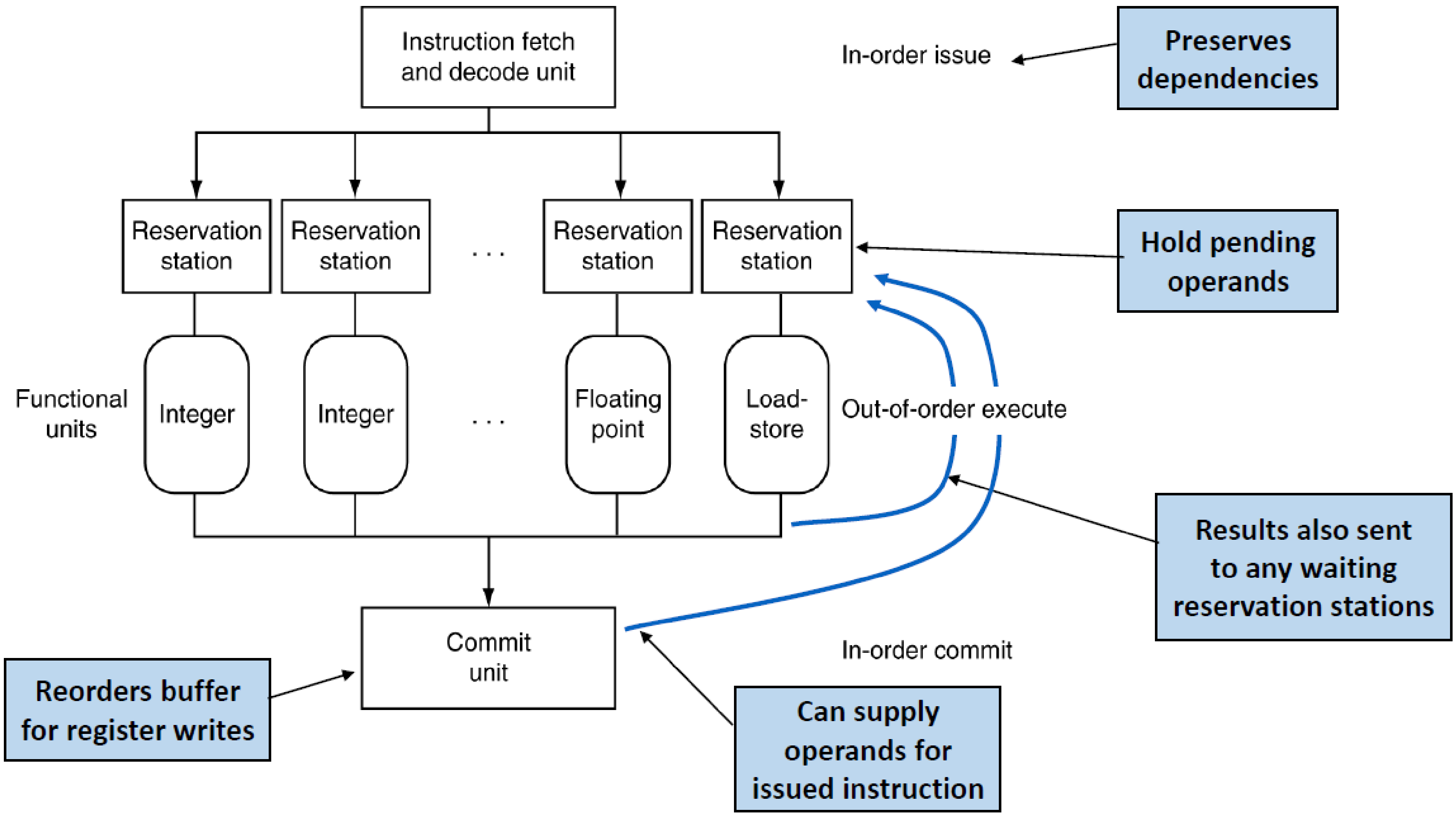
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

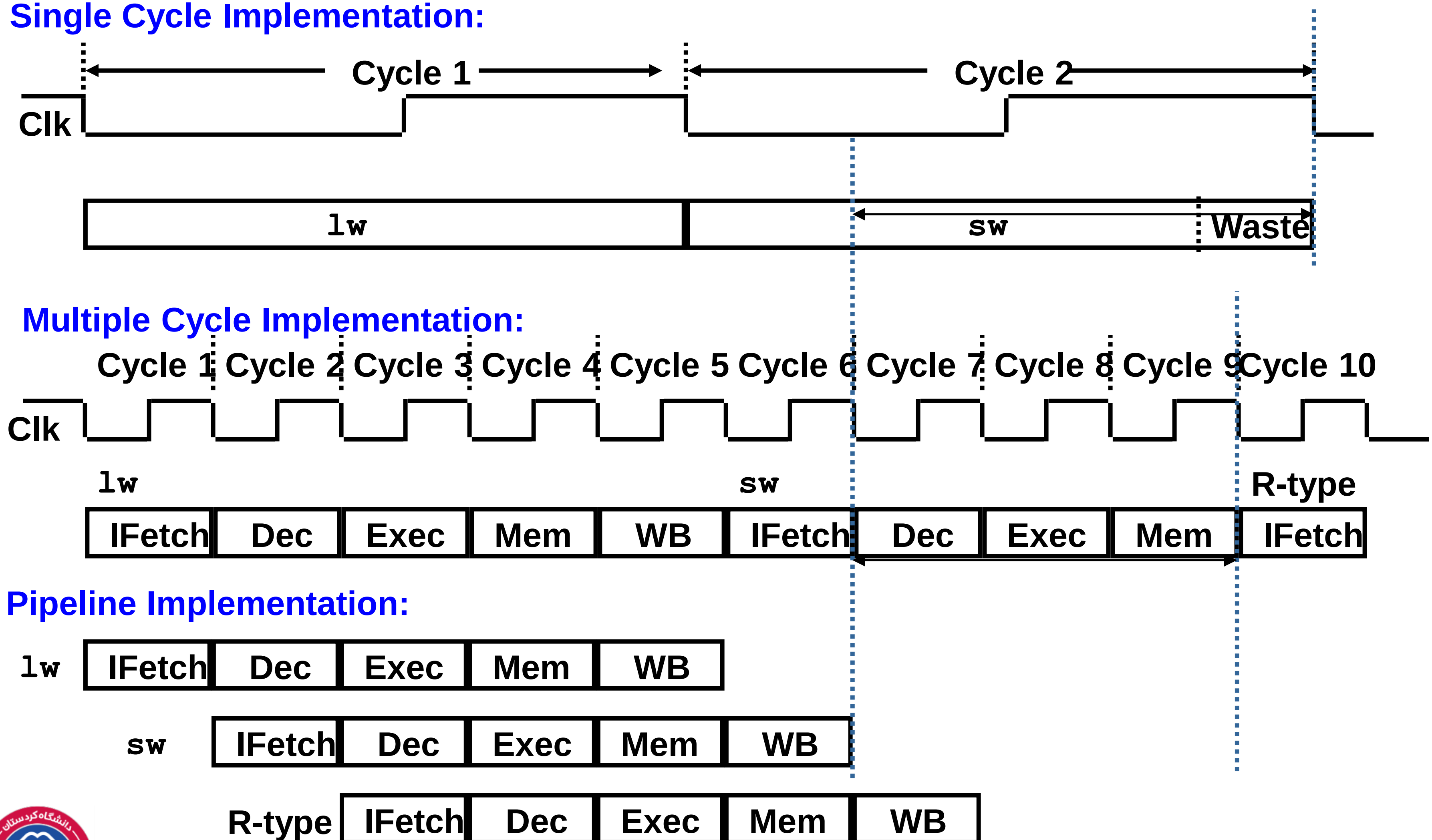
Dynamic Multiple issue (OoO execution)



Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

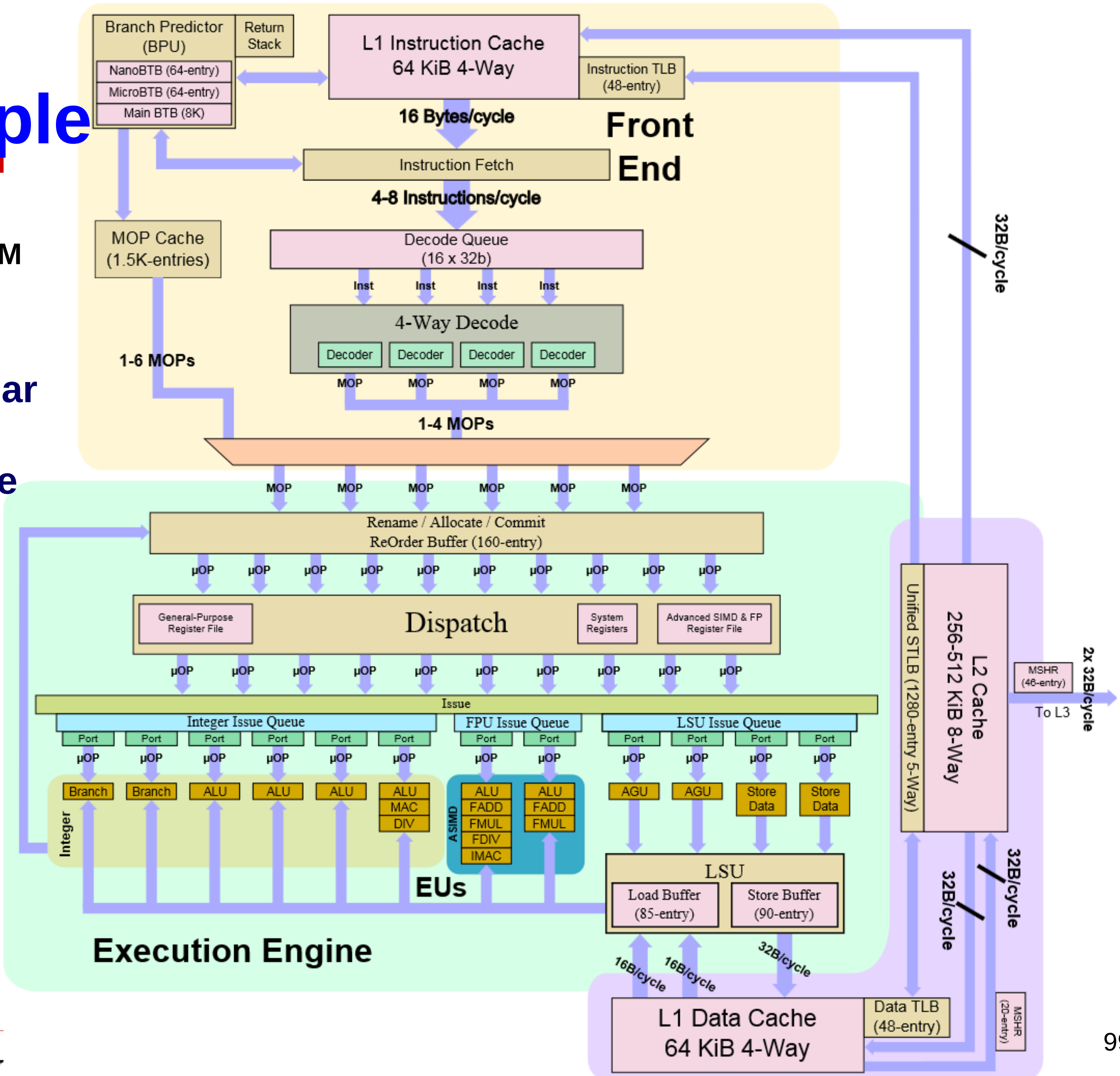
Single Cycle, Mult-Cycle, vs. Pipeline



Real Example

Cortex-A77
Microarchitectures - ARM

- **6-way superscalar**
- **out-of-order**
- **13-stage pipeline**



University of K...


Questions