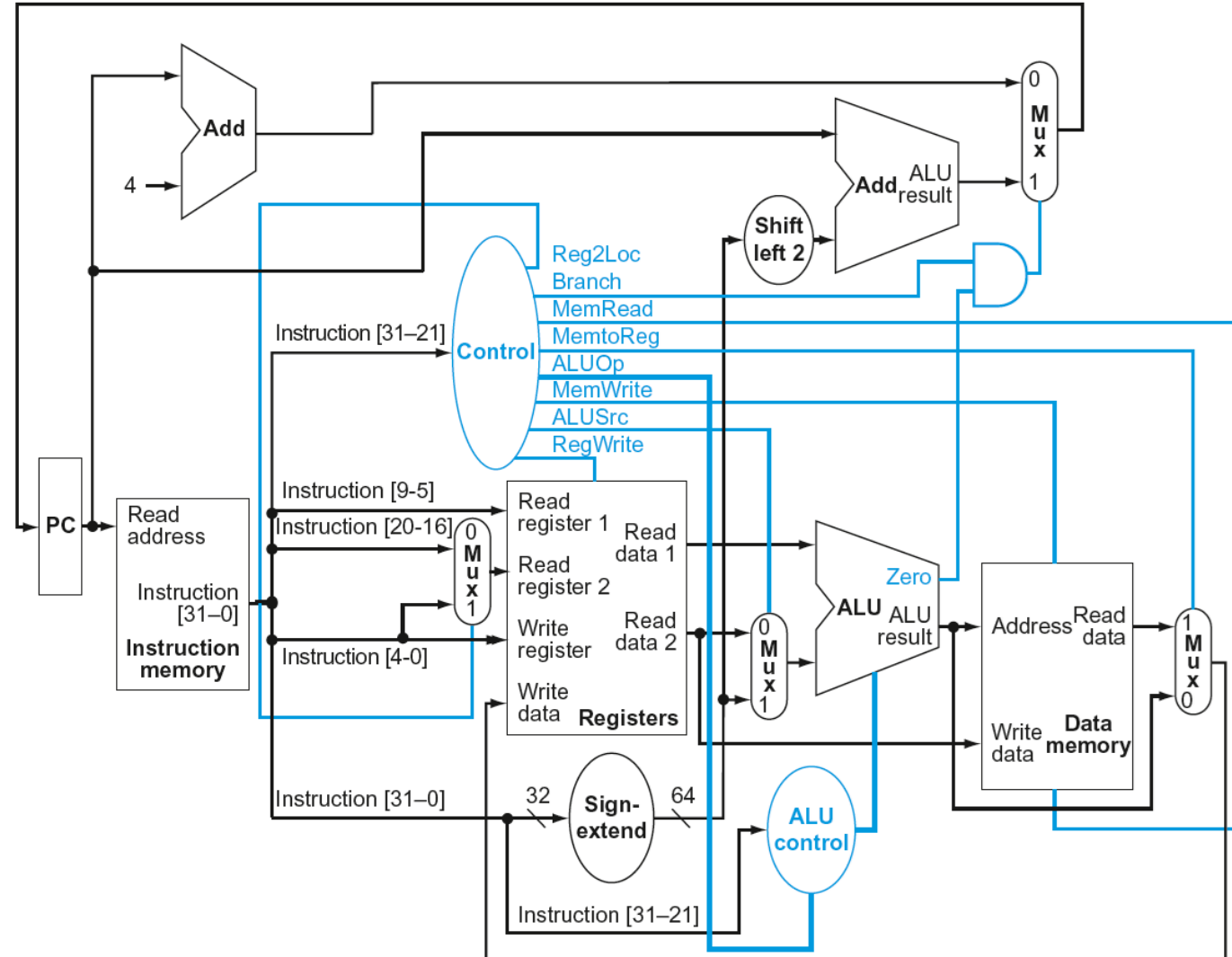


# Computer Organization and Architecture

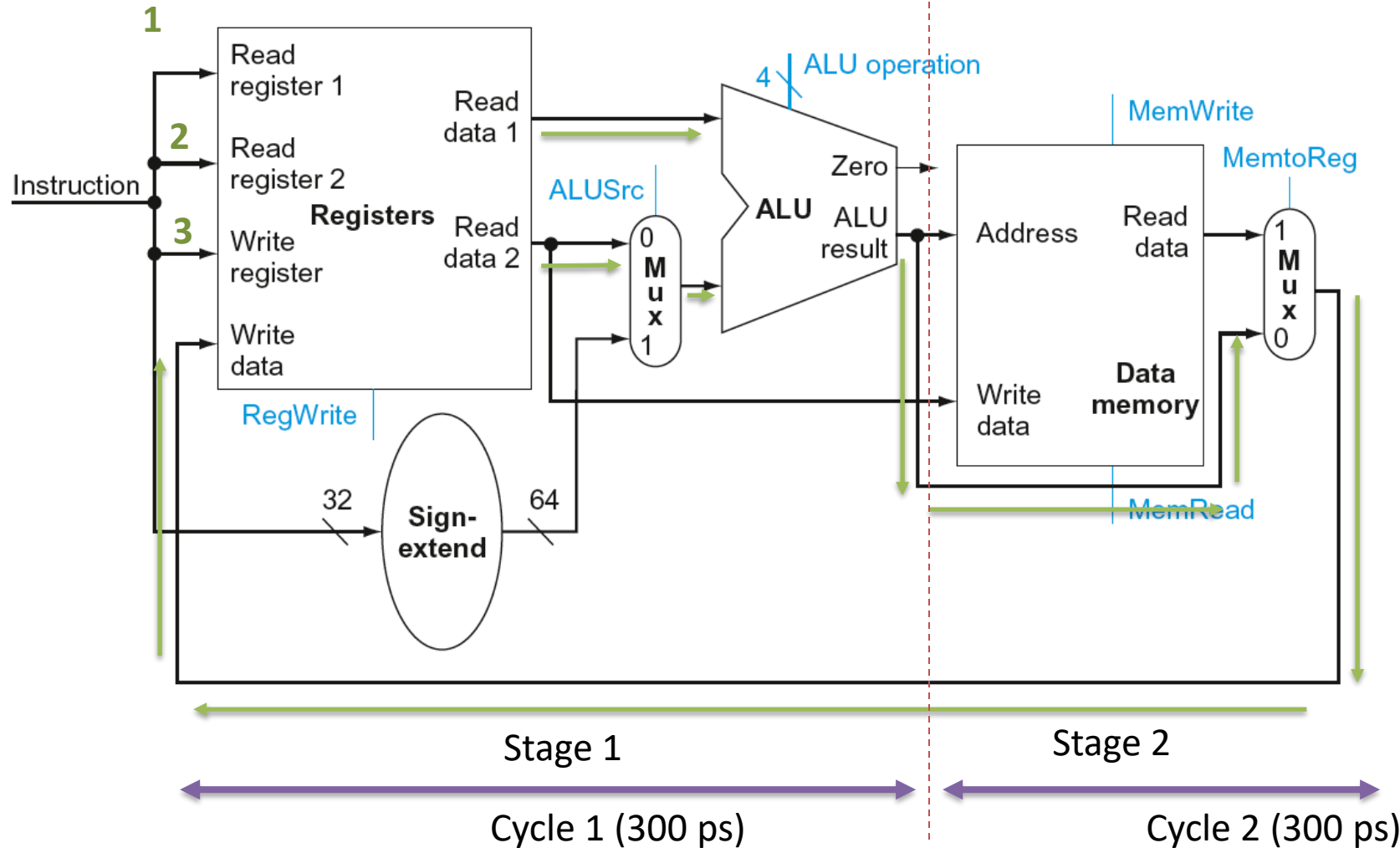
Lecture – 21

Oct 31<sup>st</sup>, 2022

# Datapath With Control



# Multi-Cycle Implementation



R-type  
ADD X3, X1, X2

Let's say ADD takes a total of 500 ps.  
If add is the instruction is broken into two stages  
Stage 1 : 300 ps  
Stage 2: 200 ps

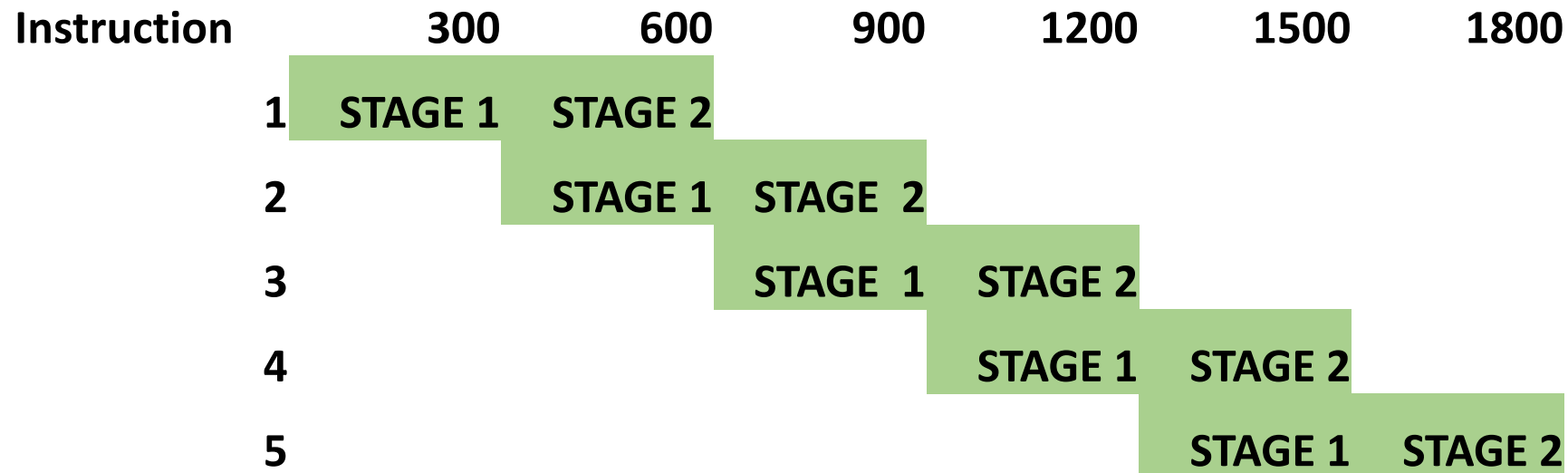
No other instruction takes longer than 300 ps  
What is the clock cycle time?  
300 ps.  
How long will it take to execute ADD?  
600 ps

# Impact on Performance.

- Executing in stage can reduce clock cycle time.
  - 500 ps → 300 ps
- But can affect or reduce overall performance.
  - Reduces throughput
    - Time to execute 5 Add instructions
      - Single cycle implementation =  $5 * 500 = 2500$  ps
      - Multi cycle implementation =  $5 * 600 = 3000$  ps

# Pipelining

- 5 Add instructions



Time to execute 5 Add instructions

Single cycle implementation =  $5 * 500 = 2500$  ps

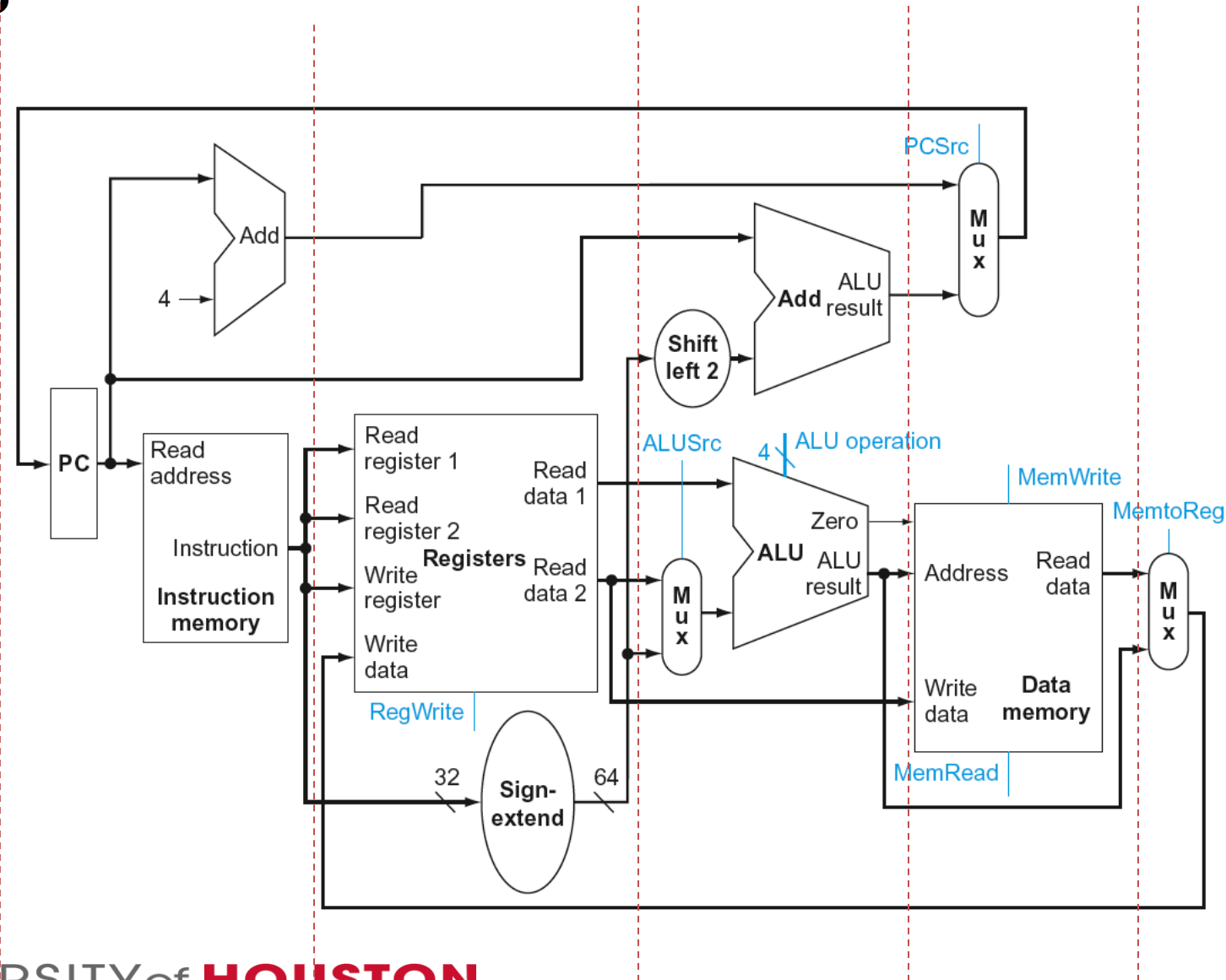
Multi cycle implementation =  $5 * 600 = 3000$  ps

**Pipelining = 1800 ps**

# Stages in LEGv8

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



# Can Pipeline cause Issues?

# Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - An instruction depends on completion of data access by a previous instruction
- Control hazard
  - Deciding on control action depends on previous instruction
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction

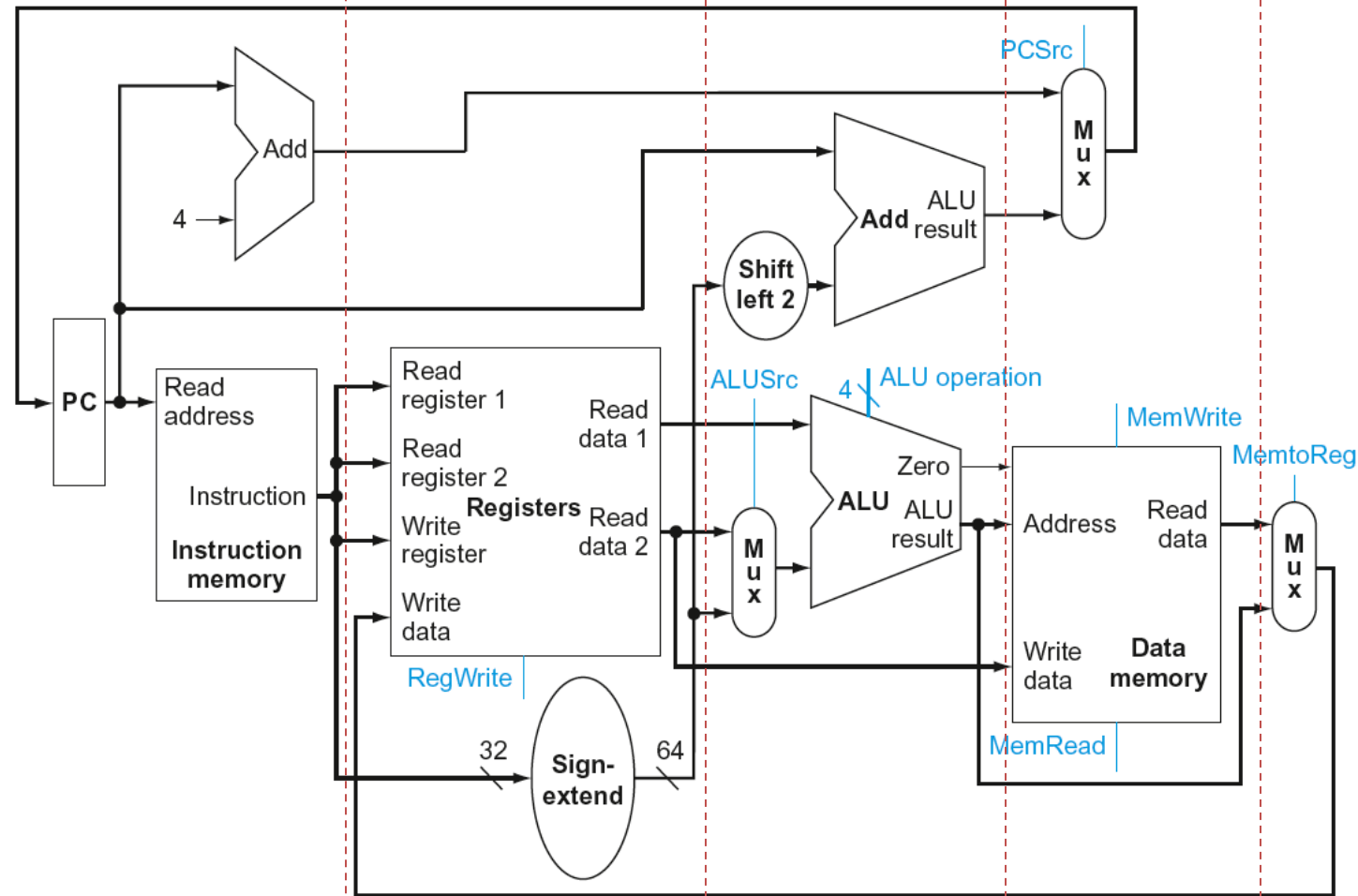


# Single Memory Example

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

Inst./ Cycle	1	2	3	4	5	6
LDUR	1	2	3	4	5	
ADD		1	2	3	5	
ADD			1	2	3	5
ADD				1	2	3

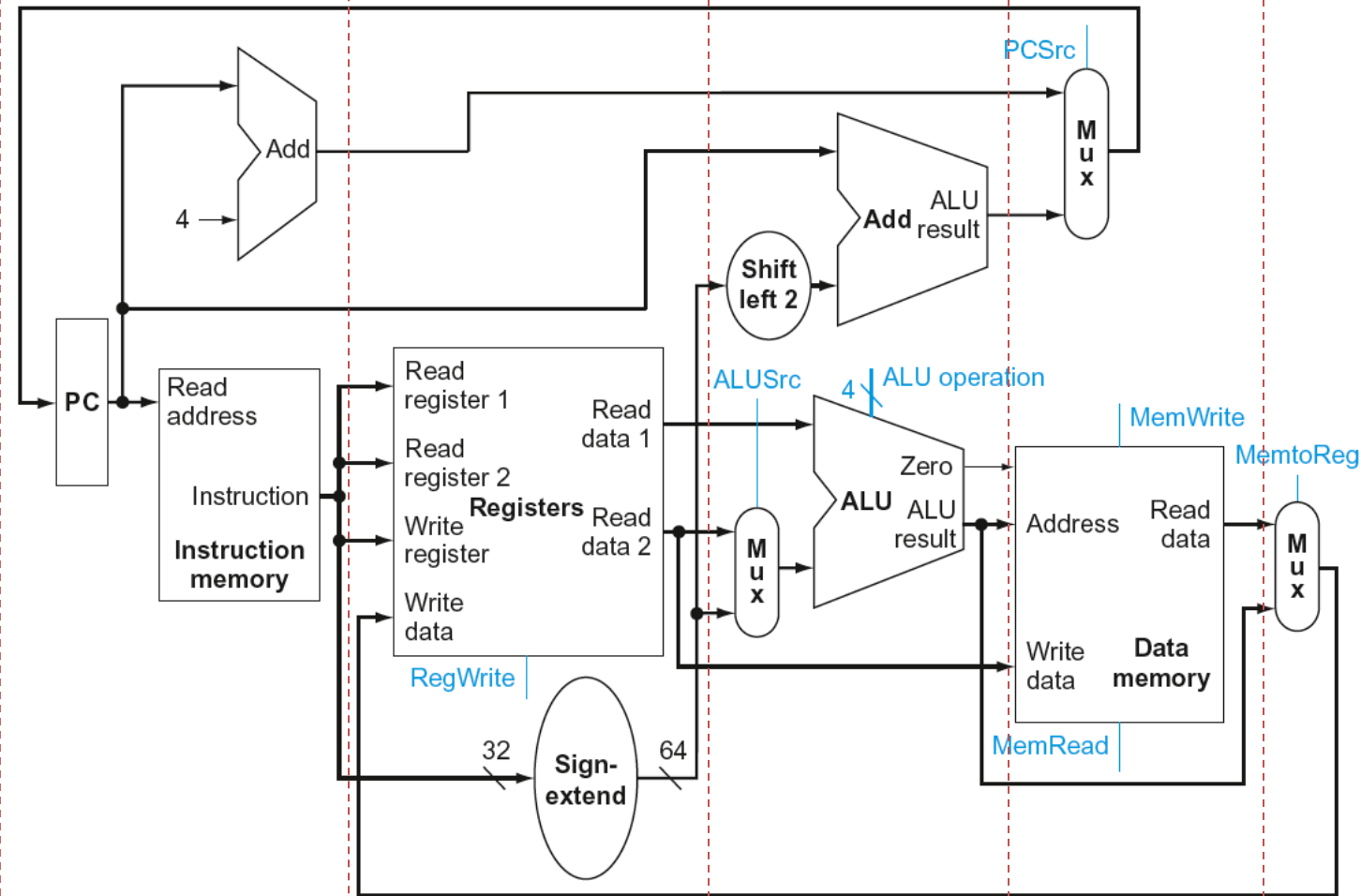


# Single Memory Example

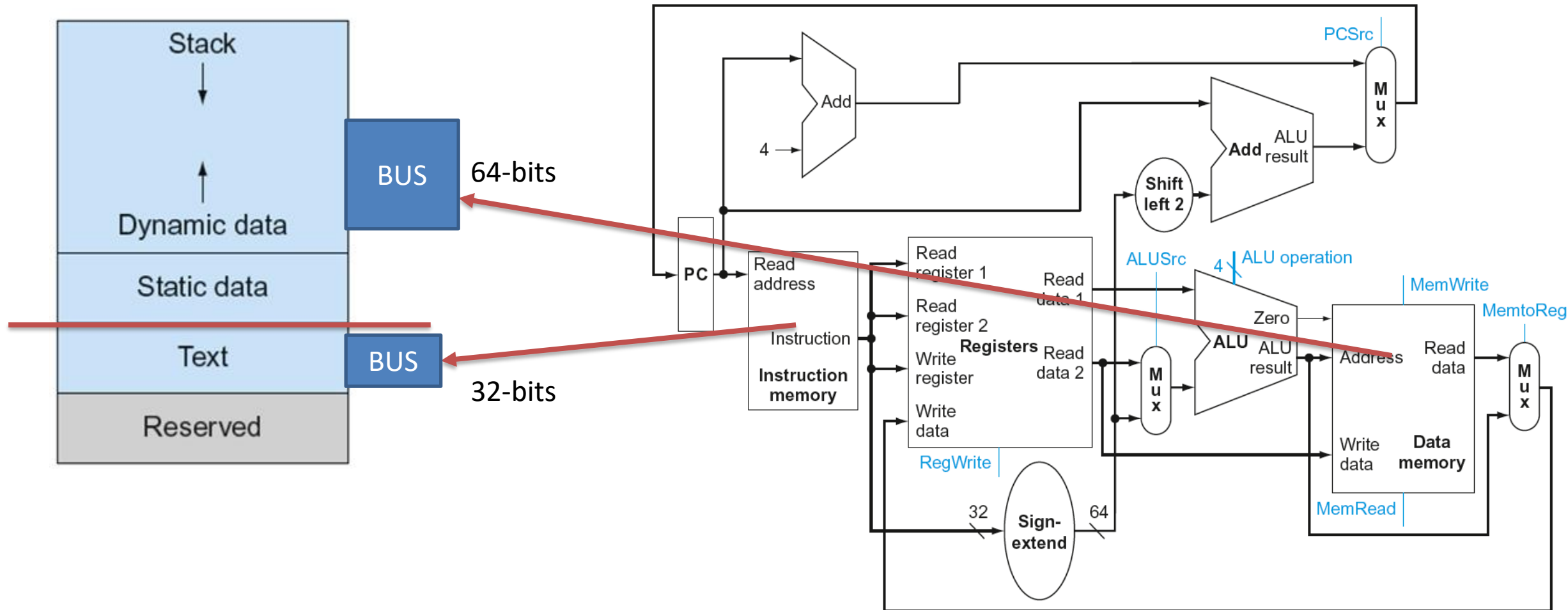
Five stages, one step per stage

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

Inst./ Cycle	1	2	3	4	5	6
LDUR	1	2	3	4	5	
ADD		1	2	3	5	
ADD			1	2	3	5
Bubble	----	--	--	--	--	--
ADD					1	2



# LEGv8 Memory



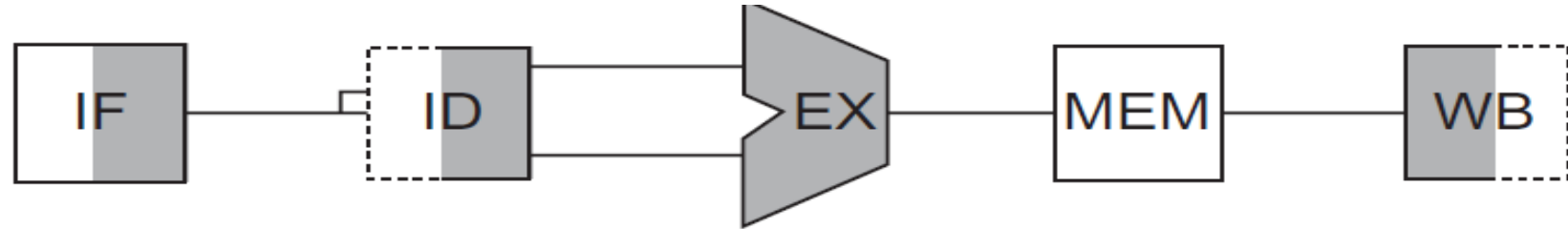
# Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - Not necessarily an issue in LEGv8
- Data hazard
  - An instruction depends on completion of data access by a previous instruction
- Control hazard
  - Deciding on control action depends on previous instruction
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction

# Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - ADD    **x19**, x0, x1
  - SUB    x2, **x19**, x3

# Example R-Type Instruction



Cycles	1	2	3	4	5
ADD X19, X0, X1	Fetch instruction from mem	Read data from registers 0, 1	Add values of registers 0, 1		
SUB X2, X19, X3		Fetch instruction from mem	Read data from registers 19, 3		

The correct value of reg 19 is not available until the write backstage.

# Three Generic Data Hazards

- True Data Dependency: also called Read-After-Write (RAW)

Instr<sub>j</sub> tries to read operand before Instr<sub>i</sub> writes it

 I: ADD x1, x2, x3  
J: SUB x4, x1, x3

- Caused by a “Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.

Slide based on a lecture by David Culler,  
University of California, Berkeley  
<http://www.eecs.berkeley.edu/~culler/courses/cs252-s05>

# Three Generic Data Hazards

- Anti-dependency: also called Write-After-Read (WAR)

Instr<sub>j</sub> writes operand before Instr<sub>i</sub> reads it



I: SUB X4, X1, X3  
J: ADD X1, X2, X3

- Results from reuse of the name “X1”.
- Can’t happen in 5 stage pipeline as long as order of instructions is maintained
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5



# Three Generic Data Hazards

- Output dependency: also called Write-After-Write (WAW)

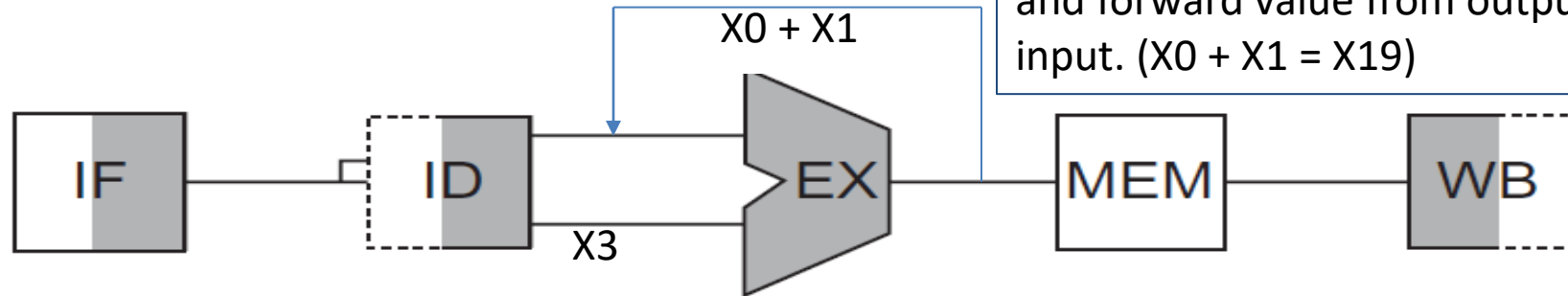
Instr<sub>j</sub> writes operand before Instr<sub>i</sub> writes it.



I: SUB x1, x4, x3  
J: ADD x1, x2, x3

- Results from the reuse of name “x1”.
- Can’t happen in 5 stage pipeline as long as order of instructions is maintained
  - All instructions take 5 stages, and
  - Writes are always in stage 5

# Example R-Type Instruction



Cycle 4:

Inst 1 → No Mem stage

Inst 2 → ignore the loaded value, and forward value from output to input. ( $X0 + X1 = X19$ )

Cycles	1	2	3	4	5
ADD <b>x19</b> , X0, X1	Fetch instruction from mem	Read data from registers 0, 1	Add values of registers 0, 1 (value to write in reg 19 is computed)		
SUB X2, <b>x19</b> , X3		Fetch instruction from mem	Read data from registers <b>19</b> , 3	Subtract X0 from <b>X19</b>	Forward value Bypass

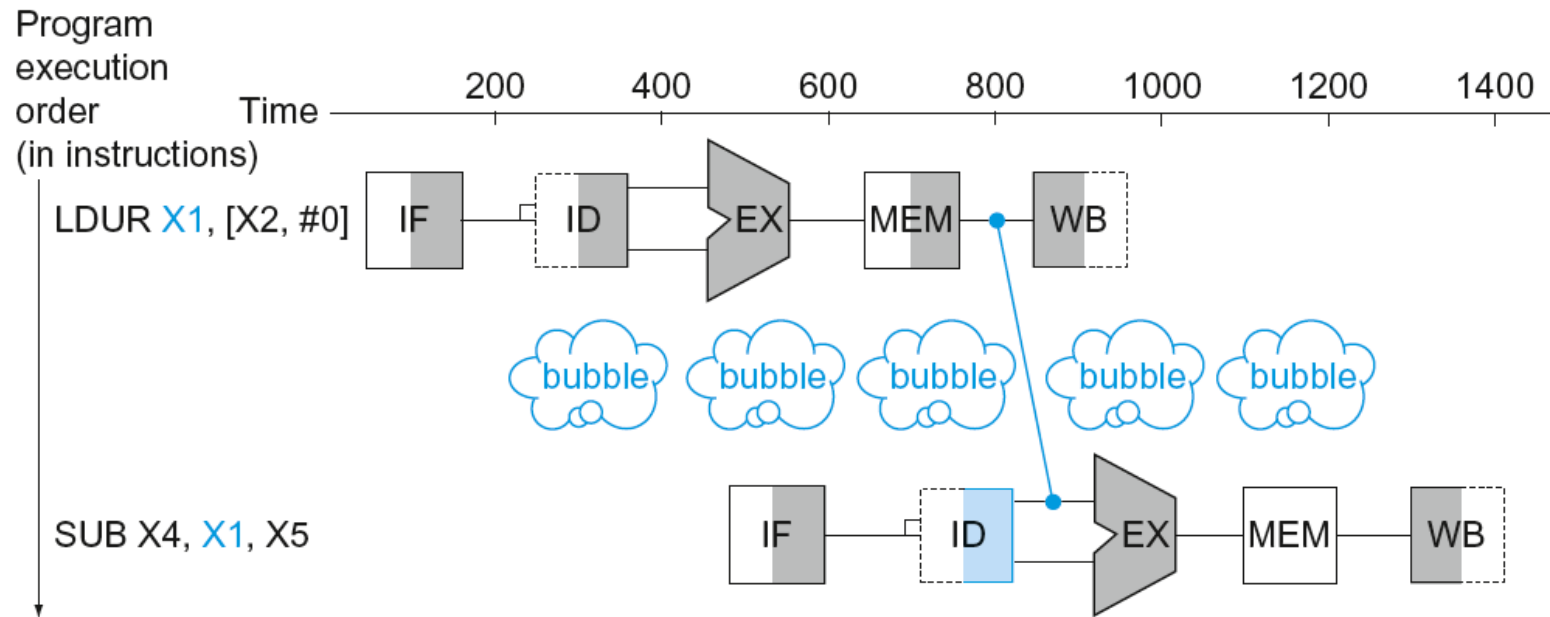
# Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - Not necessarily an issue in LEGv8
- Data hazard
  - Forwarding or Bypassing
- Control hazard
  - Deciding on control action depends on previous instruction
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction

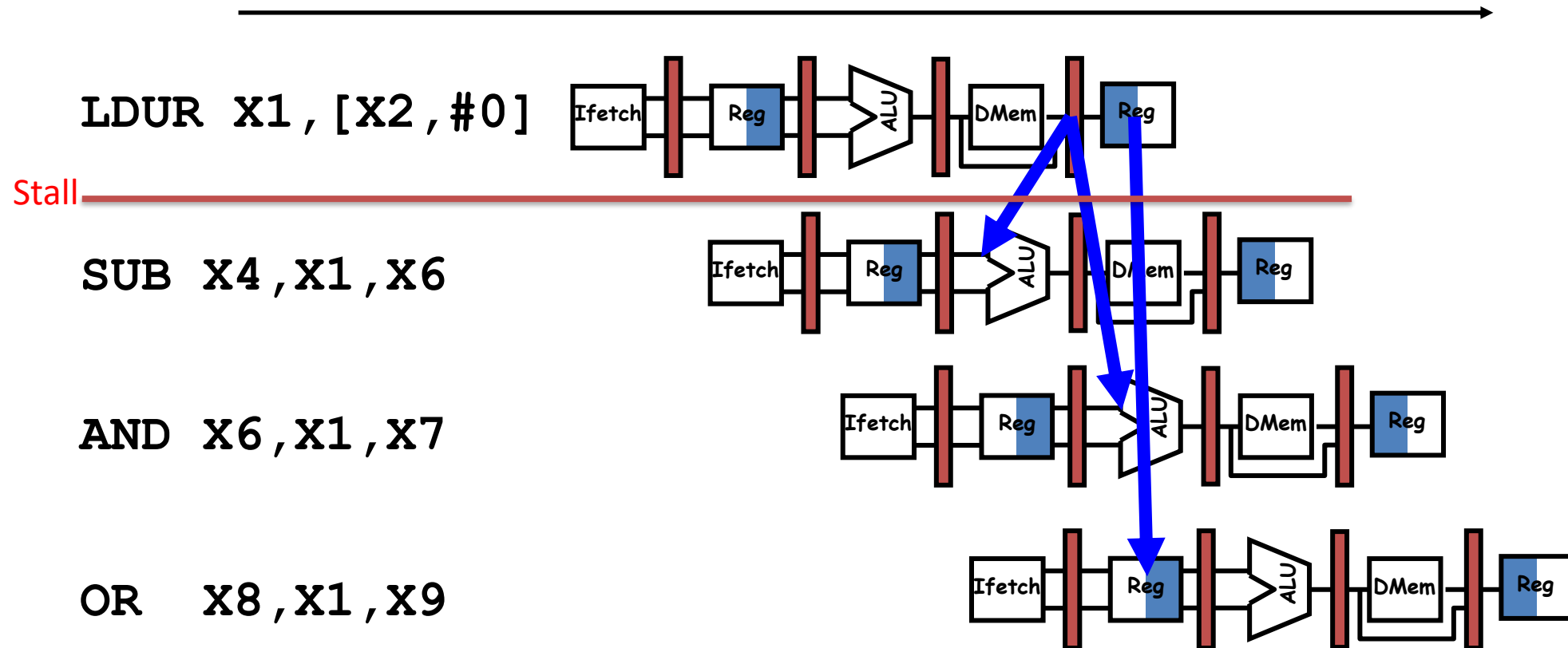
# Data Hazards Even with Forwarding

LDUR X1 [ x2, #0]

SUB X4, X1, X5



# Data Hazard even with Forwarding



# Code Scheduling to Avoid Stalls

- C code for

```
a[3]=a[0]+a[1];
```

```
a[4]=a[0]+a[2];
```

Base address of  
a in x0

# Code Scheduling to Avoid Stalls

- C code for  
a[3]=a[0]+a[1];  
a[4]=a[0]+a[2];  
Base address of  
a in x0

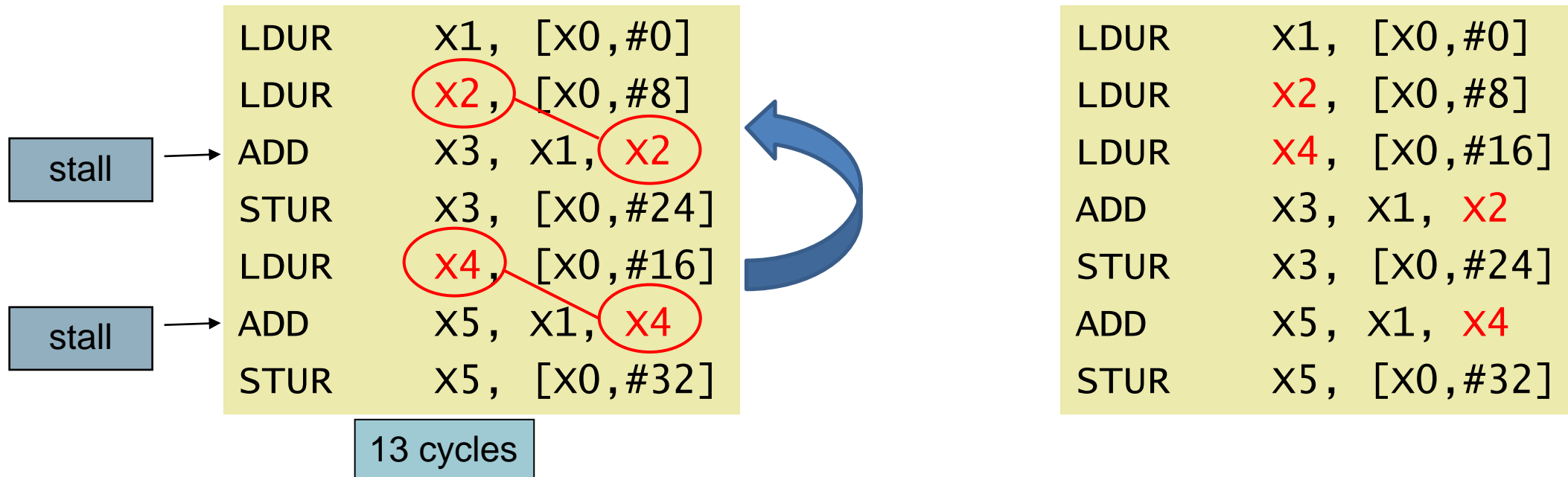
LDUR	x1, [x0, #0]
LDUR	x2, [x0, #8]
ADD	x3, x1, x2
STUR	x3, [x0, #24]
LDUR	x4, [x0, #16]
ADD	x5, x1, x4
STUR	x5, [x0, #32]

Clock Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13
LDUR X1, [X0,#0]	IF	ID	EXE	MEM	WB								
LDUR X2, [X0,#8]		IF	ID	EXE	MEM	WB							
<b>Stall</b>													
ADD X3, X1, X2				IF	ID	EXE	MEM	WB					
STUR X3, [X0,#24]					IF	ID	EXE	MEM	WB				
LDUR X4, [X0,#16]						IF	ID	EXE	MEM	WB			
<b>Stall</b>													
ADD X5, X1, X4								IF	ID	EXE	MEM	WB	
STUR X5, [X0,#32]									IF	ID	EXE	MEM	WB

**Total 13 Clock Cycles**



# Code Scheduling to Avoid Stalls



# Code Scheduling to Avoid Stalls

Clock Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13
LDUR X1, [X0,#0]	IF	ID	EXE	MEM	WB								
LDUR X2, [X0,#8]		IF	ID	EXE	MEM	WB							
LDUR X4, [X0,#16]			IF	ID	EXE	MEM	WB						
ADD X3, X1, X2				IF	ID	EXE	MEM	WB					
STUR X3, [X0,#24]					IF	ID	EXE	MEM	WB				
ADD X5, X1, X4						IF	ID	EXE	MEM	WB			
STUR X5, [X0,#32]							IF	ID	EXE	MEM	WB		

**Total 11 Clock Cycles**

# Hazards

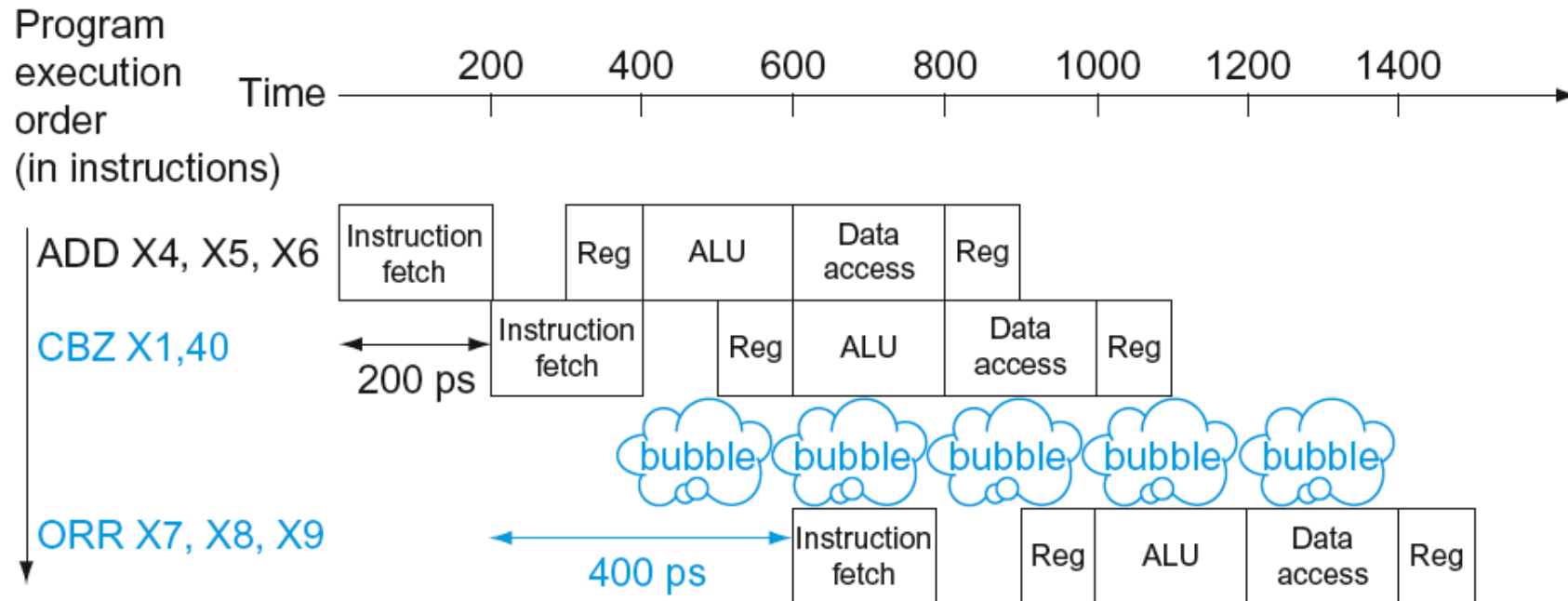
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - Not necessarily an issue in LEGv8
- Data hazard
  - Forwarding or Bypassing (Can still stall)
  - Code scheduling
- Control hazard
  - Deciding on control action depends on previous instruction
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction

# 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 LEGv8 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



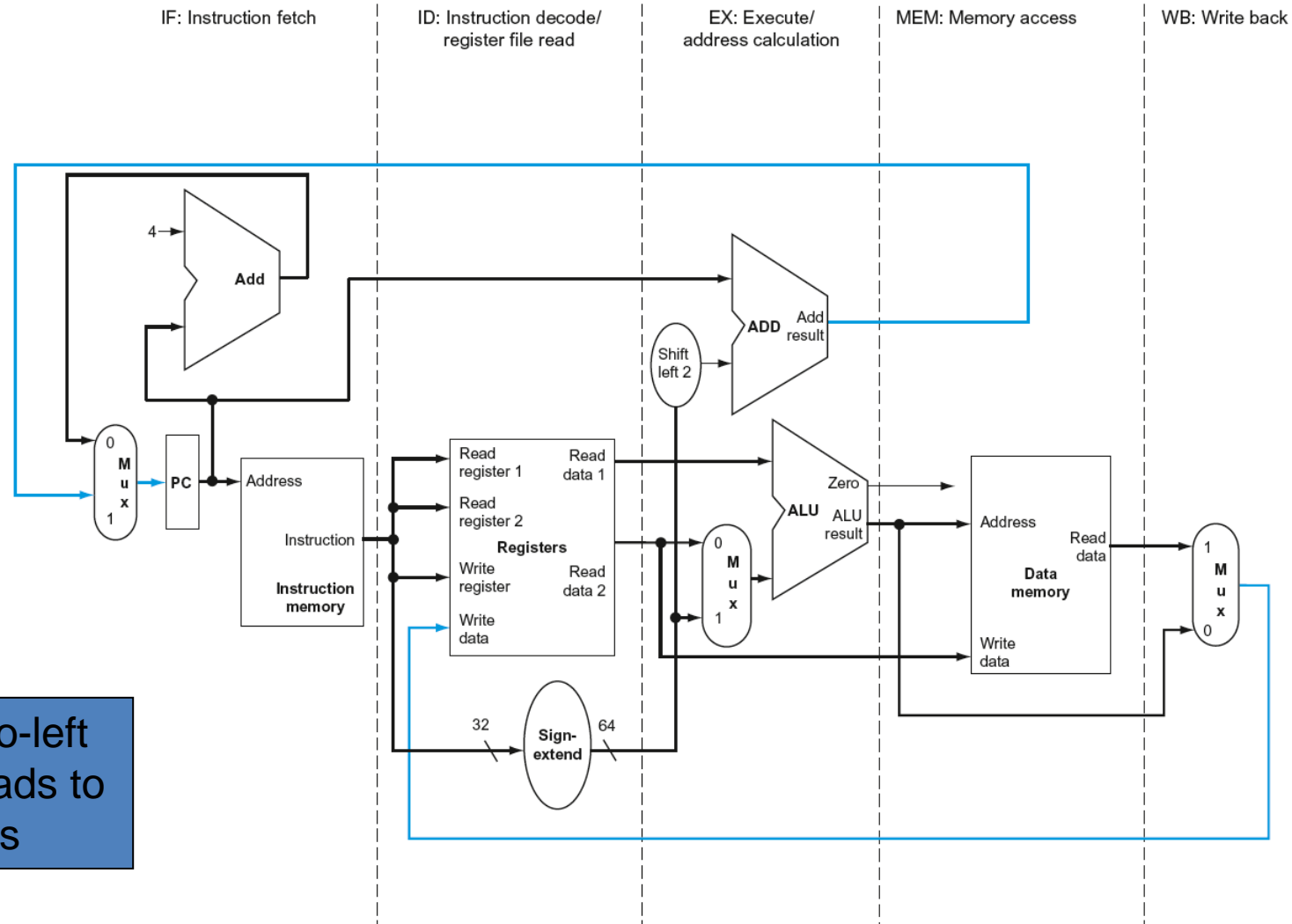
# Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In LEGv8 pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

# Hardware to realize

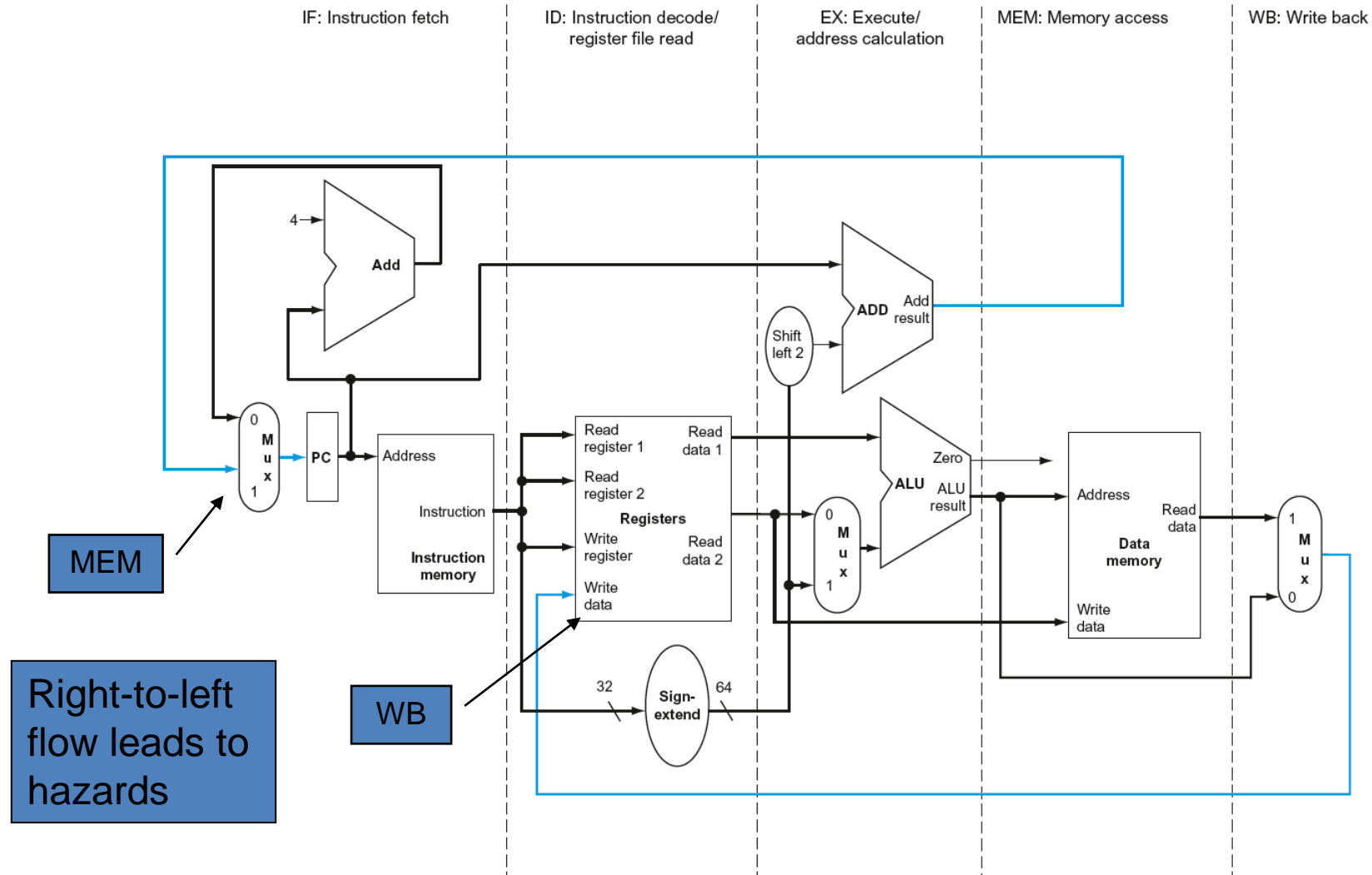
1. Pipelining
2. Data Hazards
  1. Forwarding
  2. Stalling
3. Control Hazards
  1. Branch prediction

# LEGv8 Pipelined Datapath

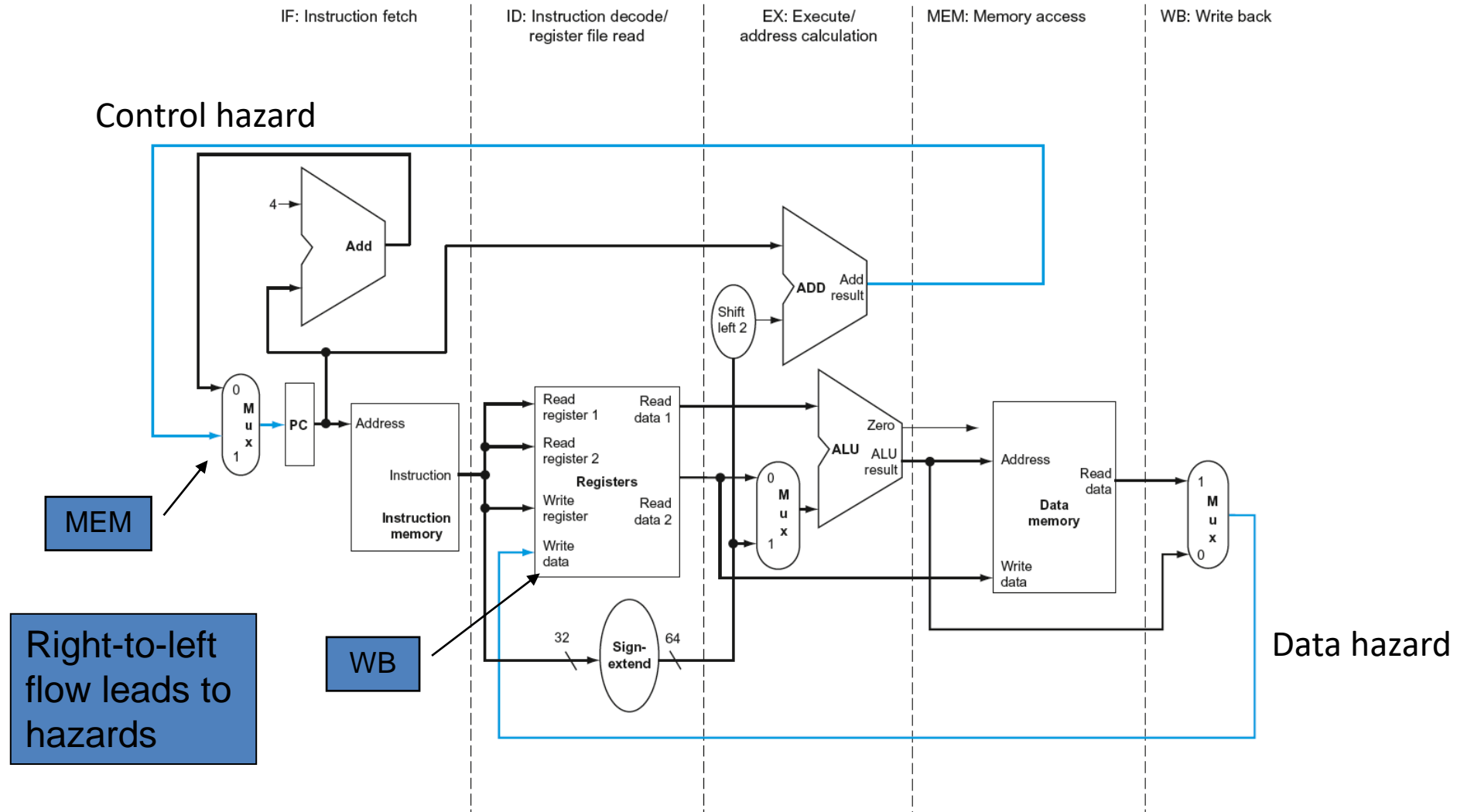




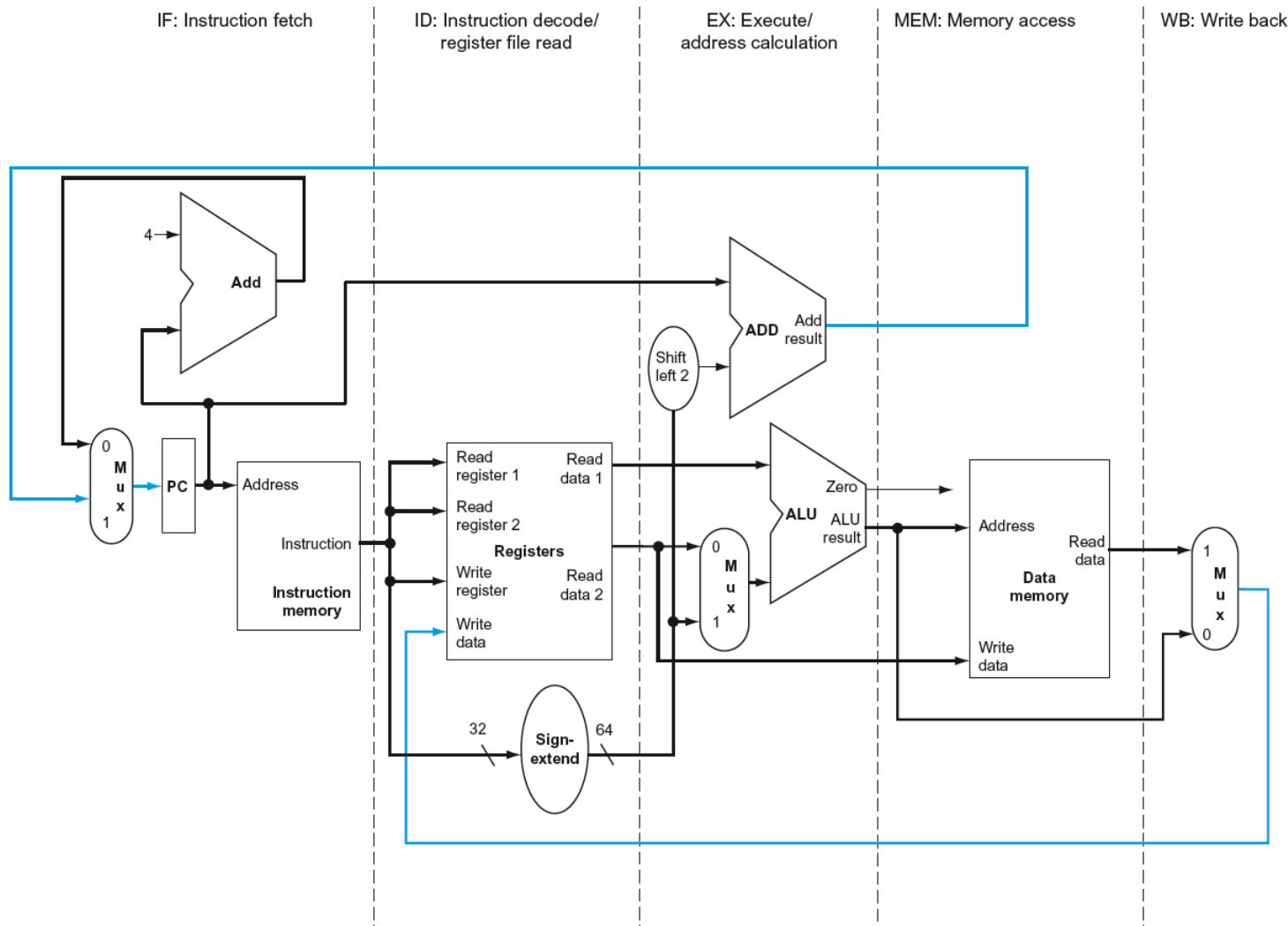
# LEGv8 Pipelined Datapath



# LEGv8 Pipelined Datapath



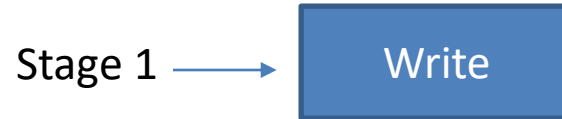
# LEGv8 Pipelined Datapath



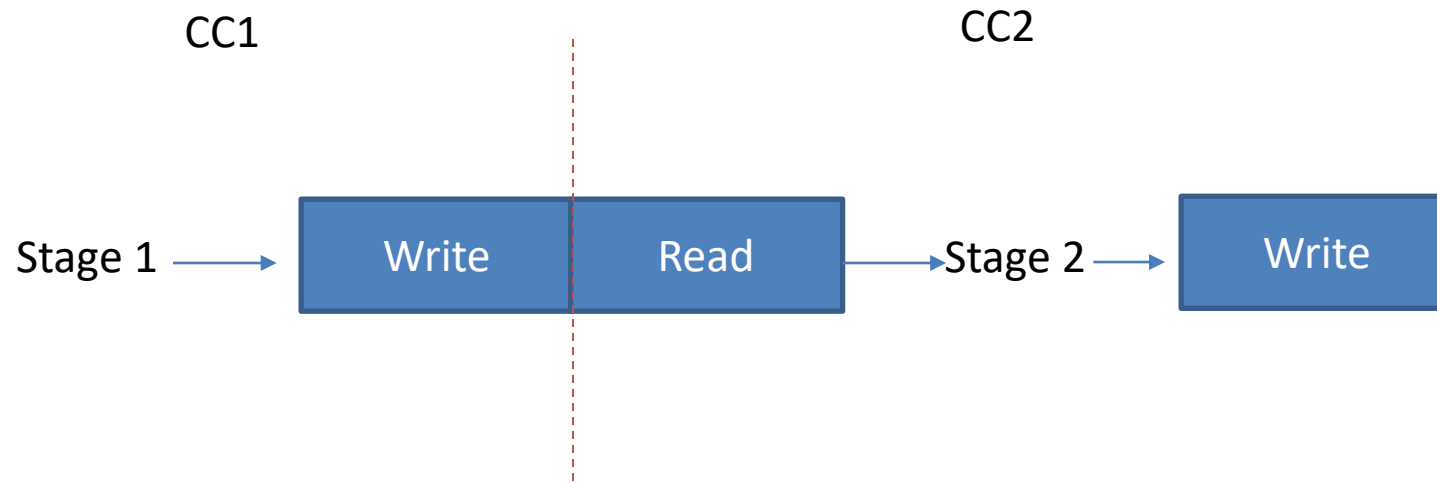
1. Resource only used in one of the 5 stages.
2. Instruction memory used in stage 1
3. Instruction and results of the first stage should be passed to the next stage.
4. Use registers in between to save results and synchronize flow of information

# LEGv8 Pipelined Datapath

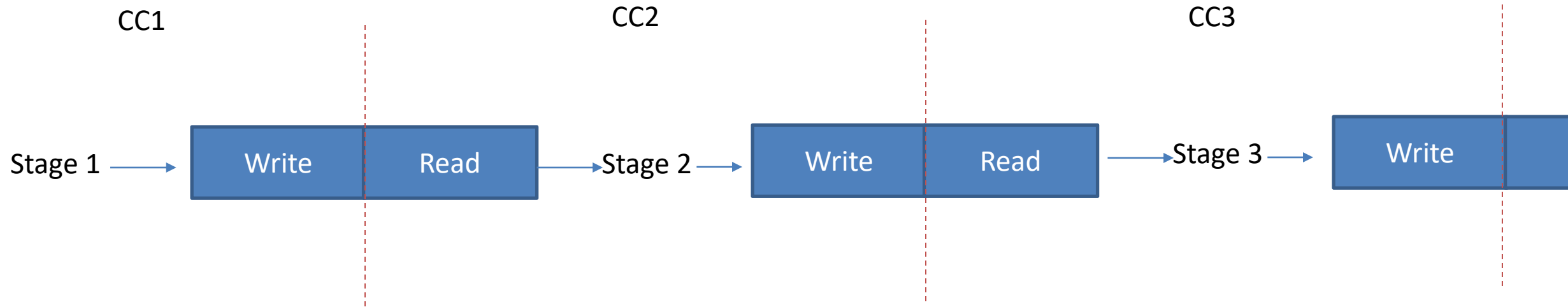
CC1



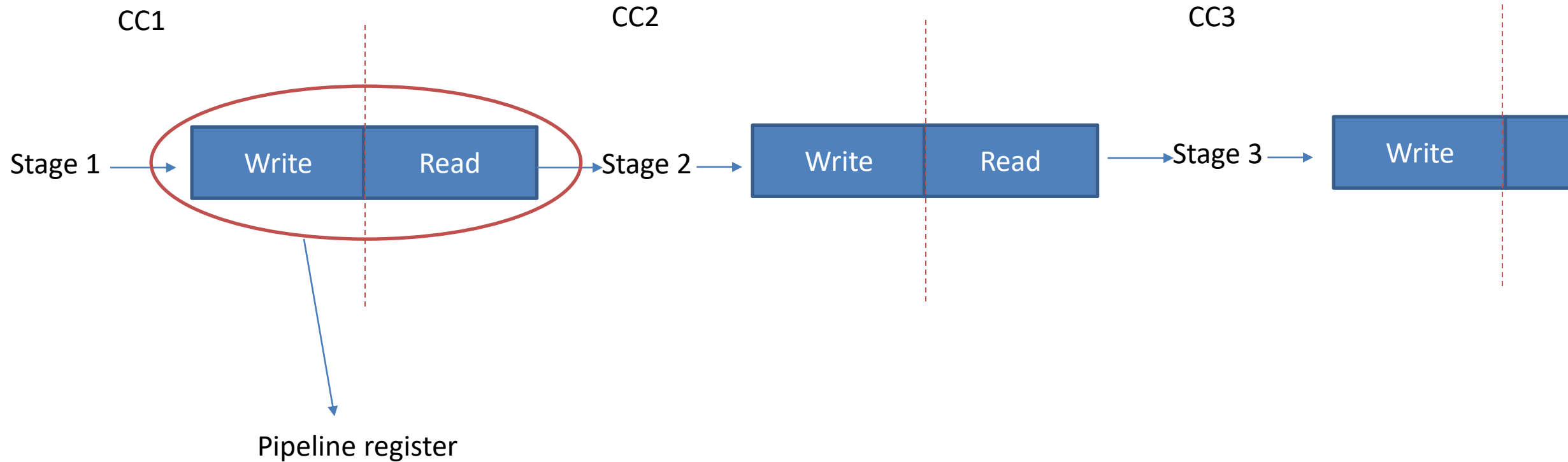
# LEGv8 Pipelined Datapath



# LEGv8 Pipelined Datapath

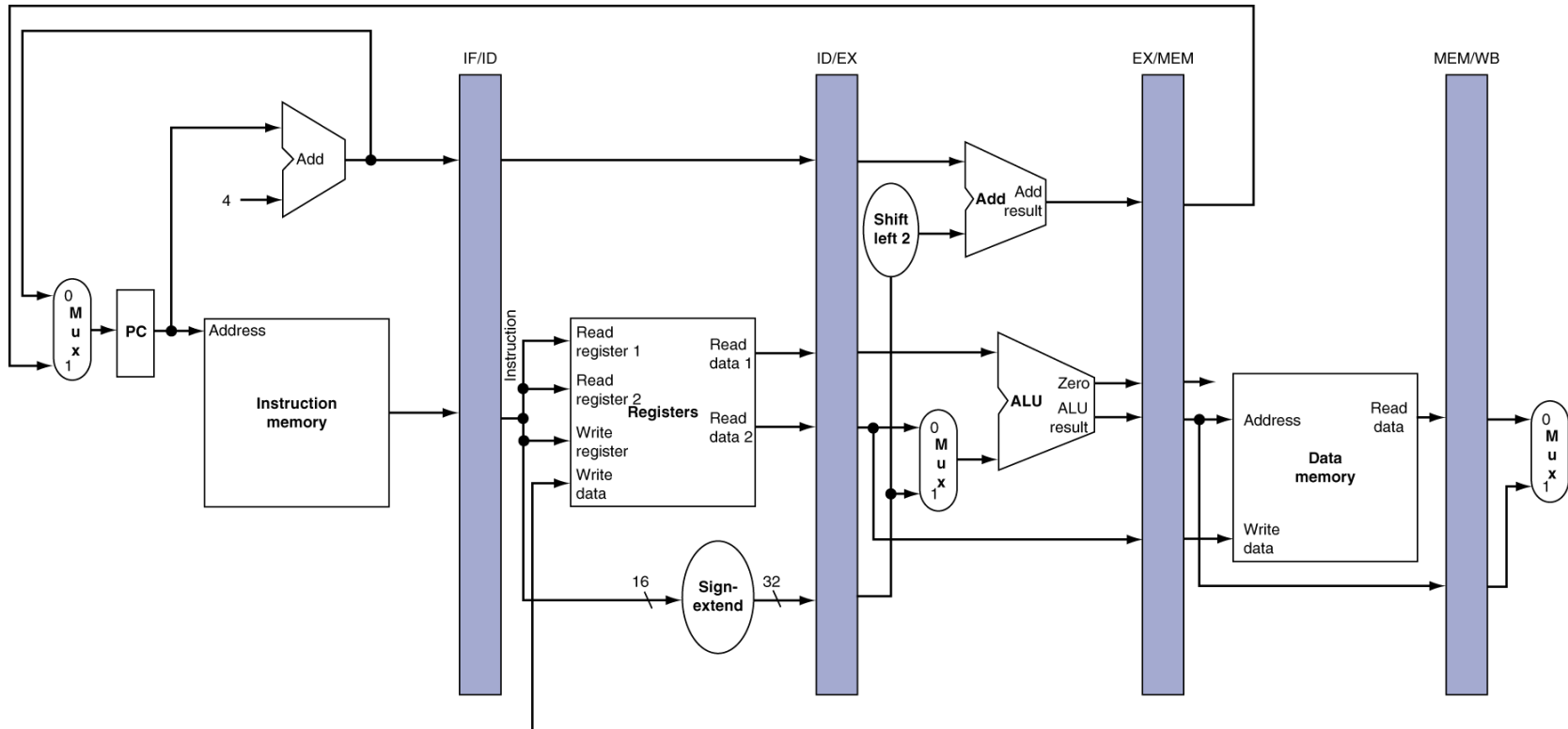


# LEGv8 Pipelined Datapath



# Pipeline registers

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



Pipeline registers

1. IF/ID
2. ID/EX
3. EX/MEM
4. MEM/WB

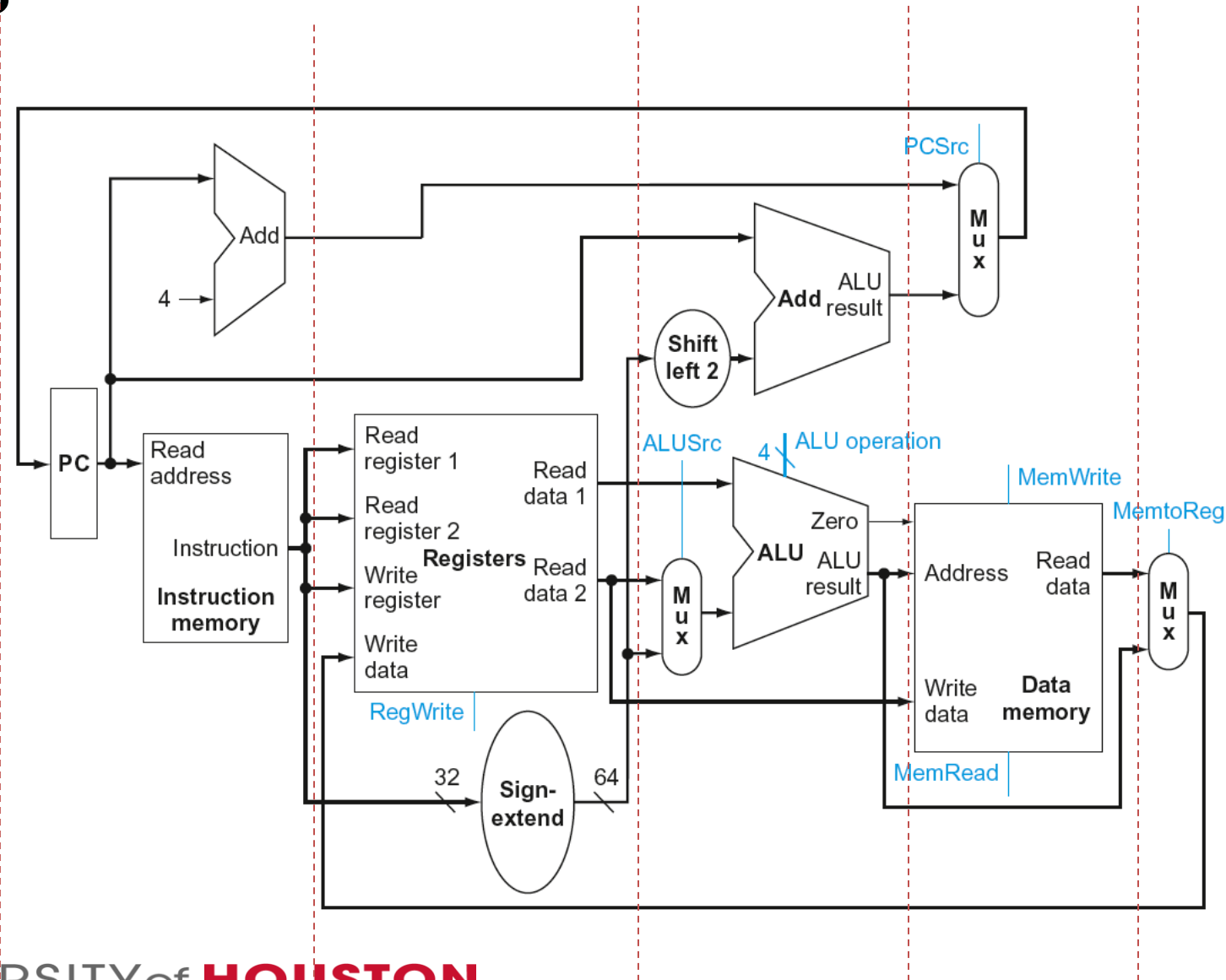


# Stages in LEGv8

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

Inst./Stage	1	2	3	4	5
ADD(R format)					
STUR					
<b>LDUR</b>					
CBZ					



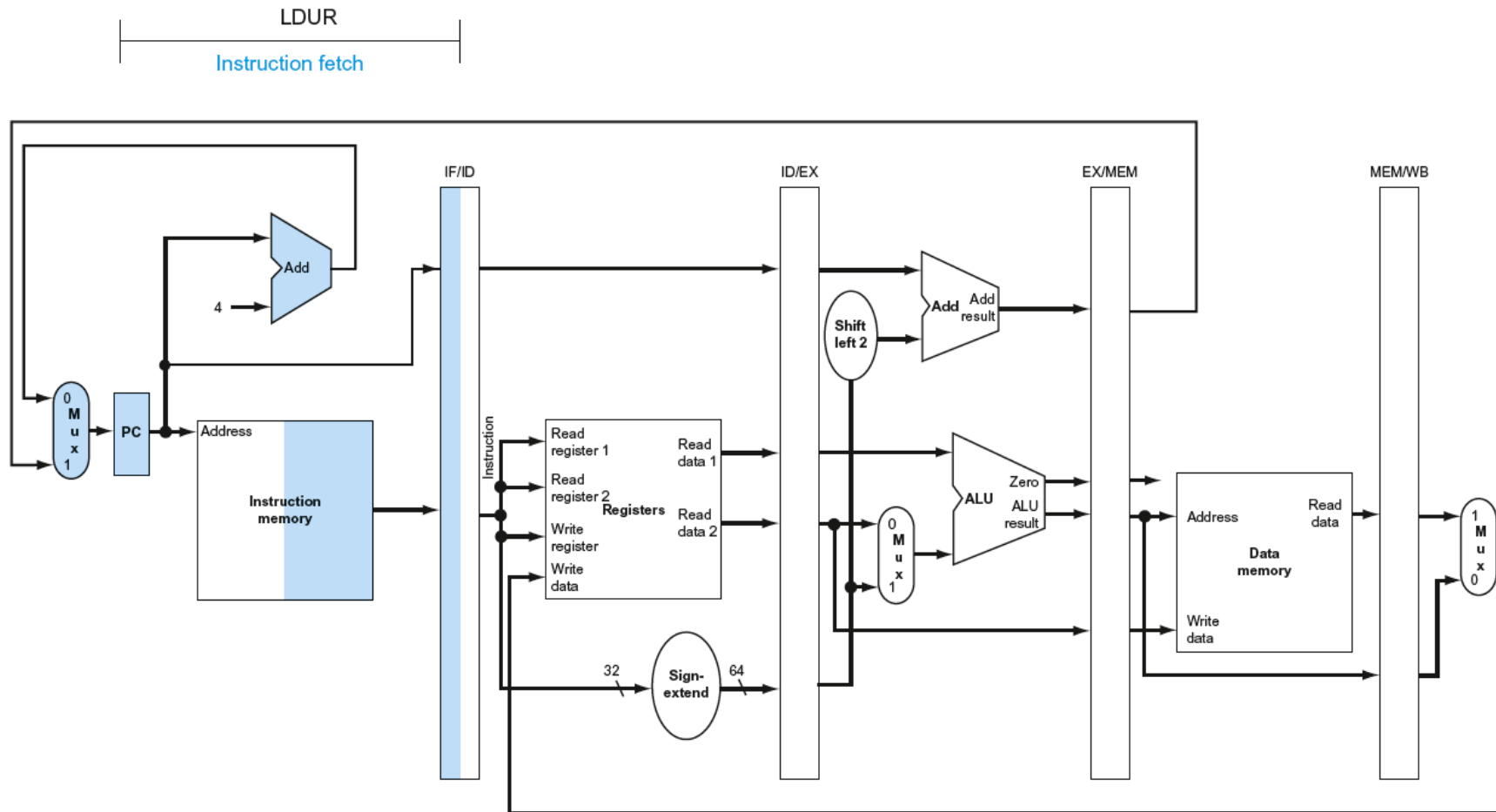
Memory, Register file

Colored left half indicates write.

In any clock cycle write is done in the first half followed by read.



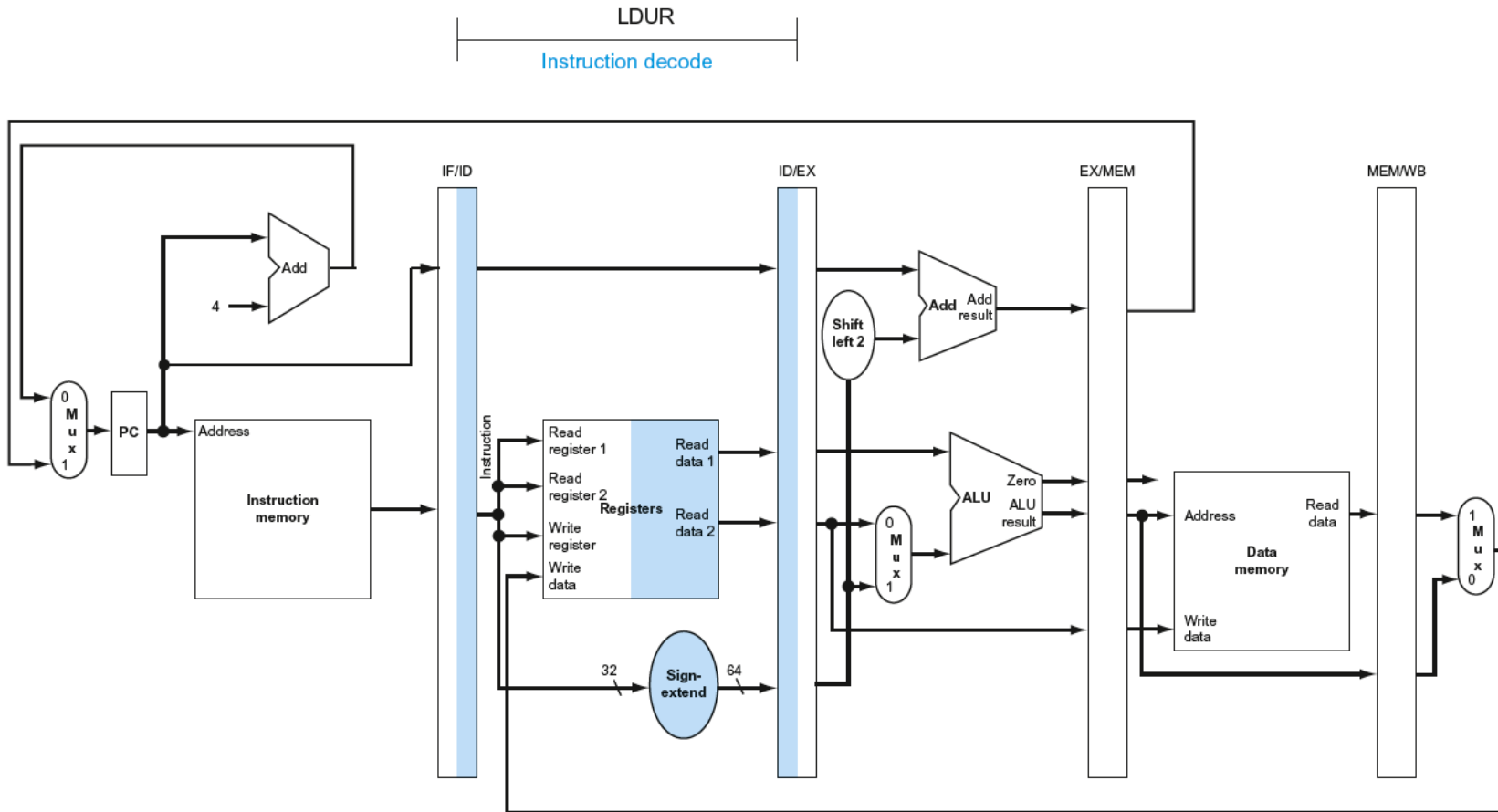
# IF for Load, Store, ...



1. Read instruction from mem
2. Place in IF/ID pipeline register
3. Increment PC by 4 and write back to PC and save it to IF/ID register (CBZ)
4. Does not know which type of instruction is read (yet!)
5. Everything needed for next CC

# ID for Load, Store, ...

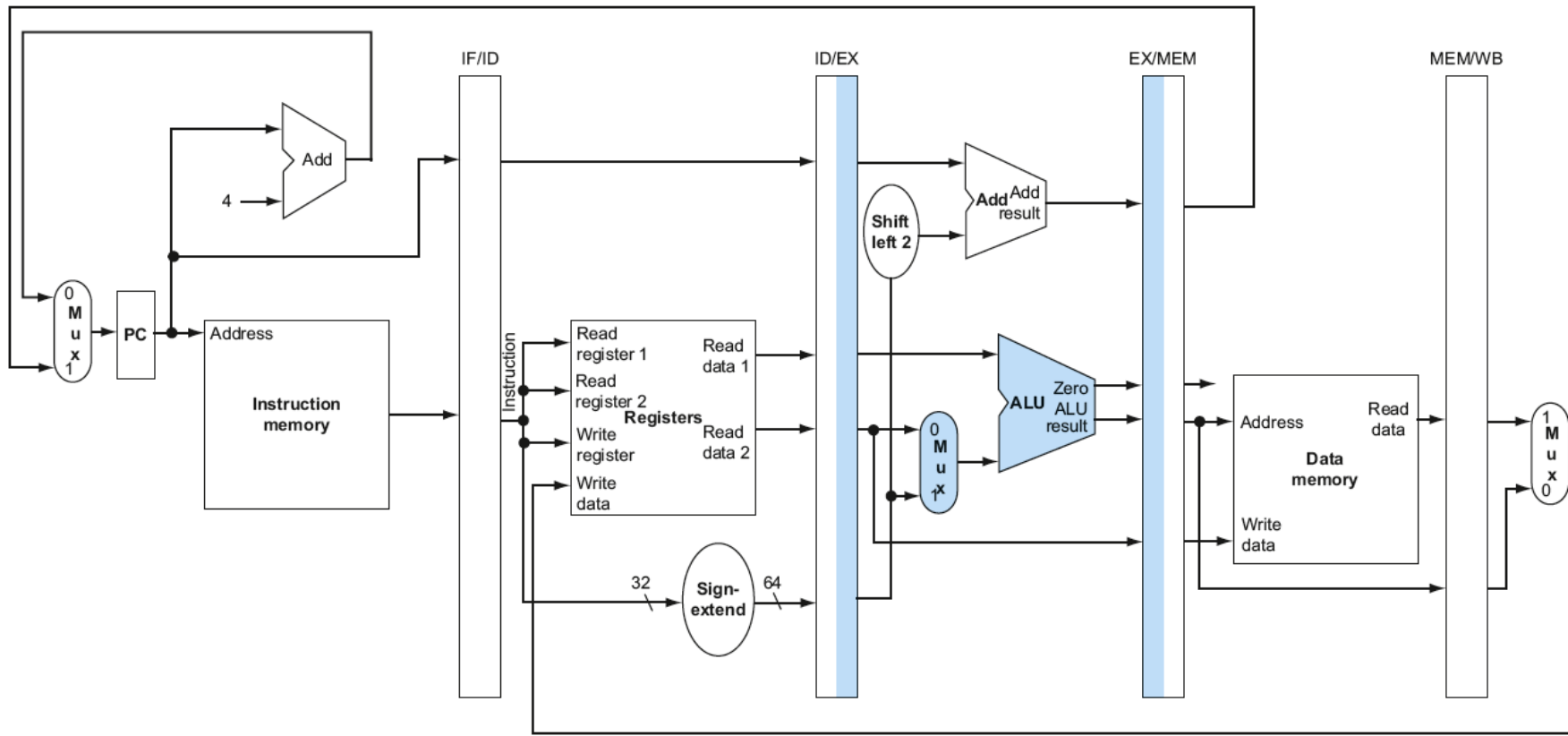
1. Read register values (base address) are written to ID/EX register
2. Sign extend for the offset
3. Incremented PC is also forwarded to ID/EX
4. Everything needed for next CC



# EX for Load

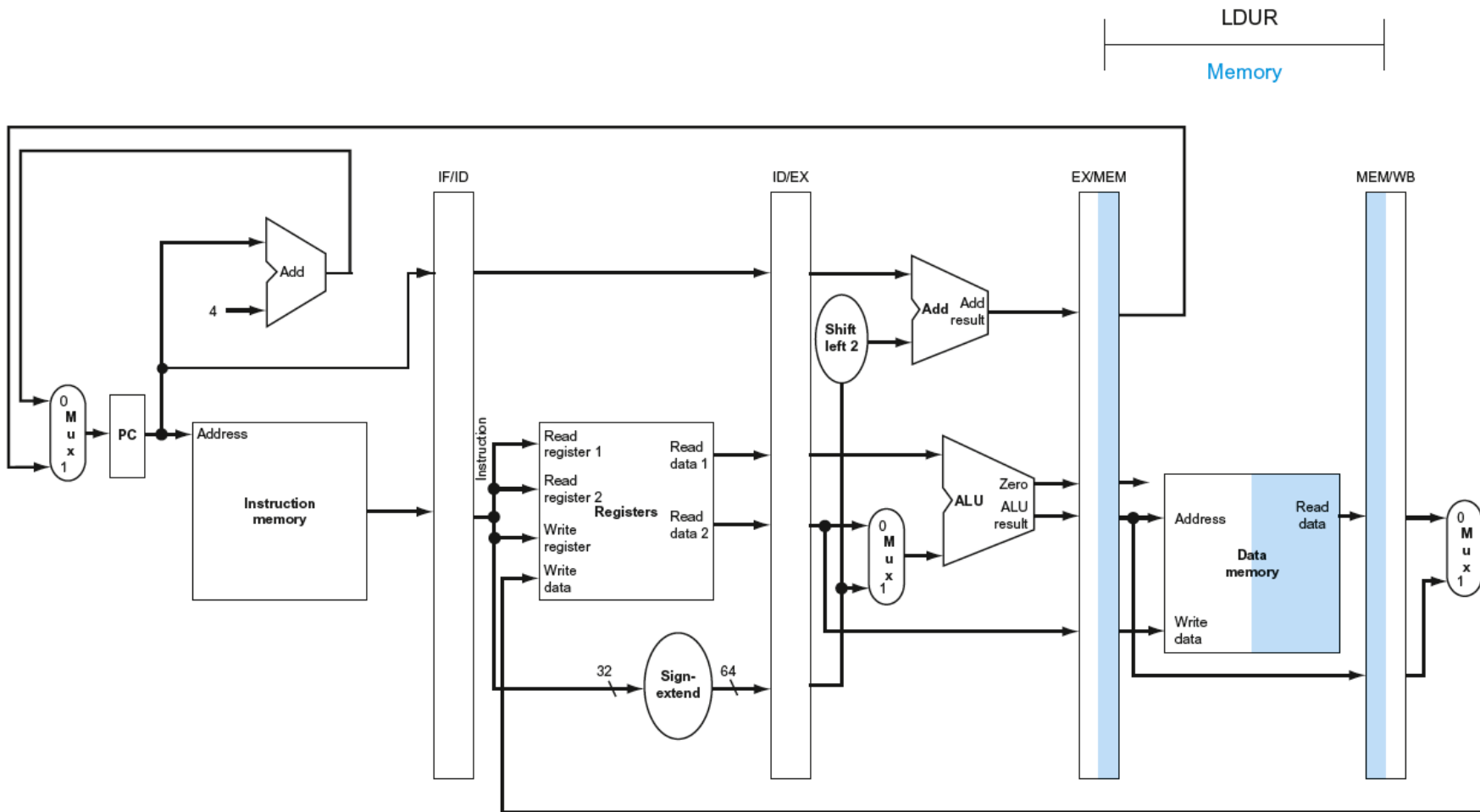


1. Add base address and offset and write to EX/MEM register



# MEM for Load

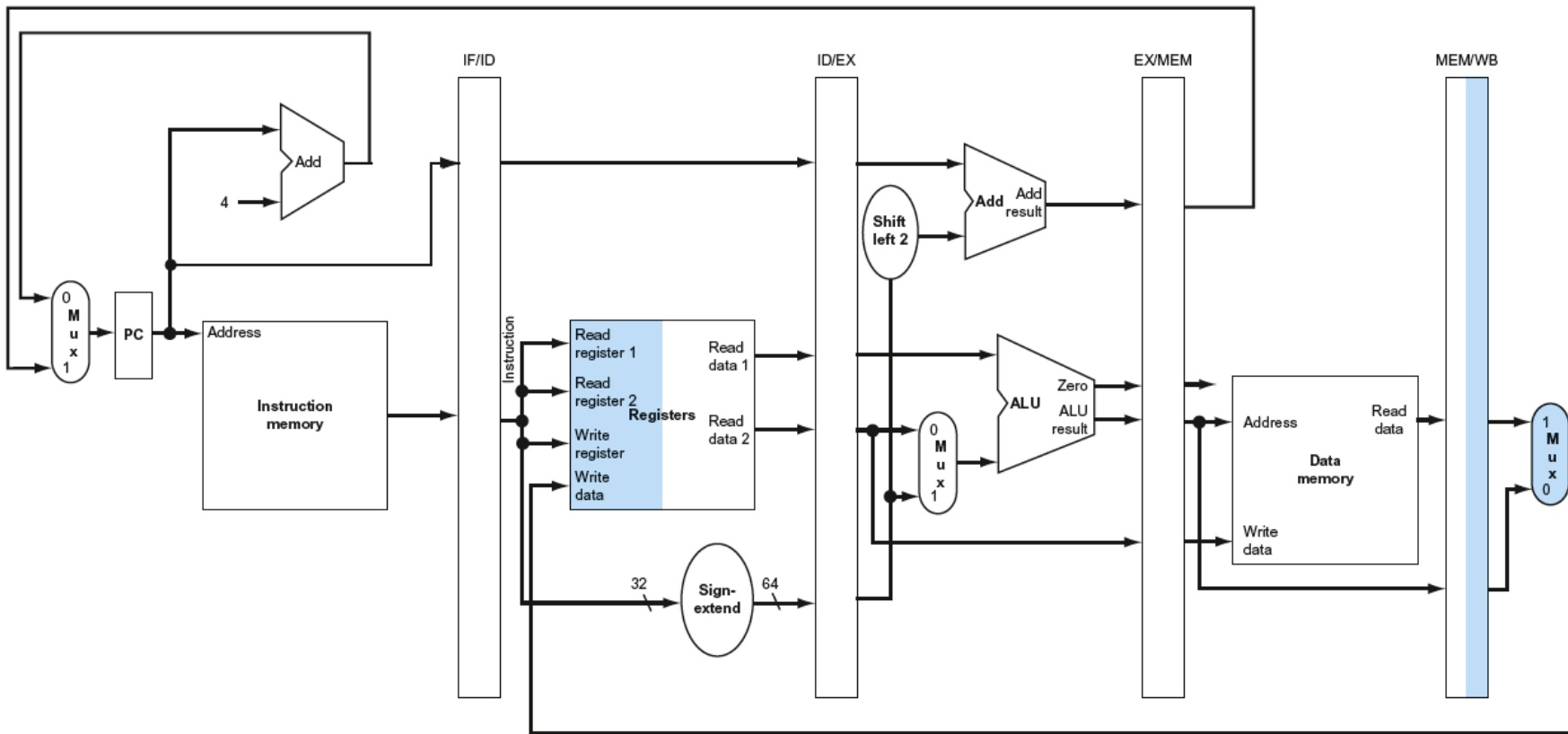
1. Read data and write to MEM/WB register



# WB for Load

LDUR  
Write-back

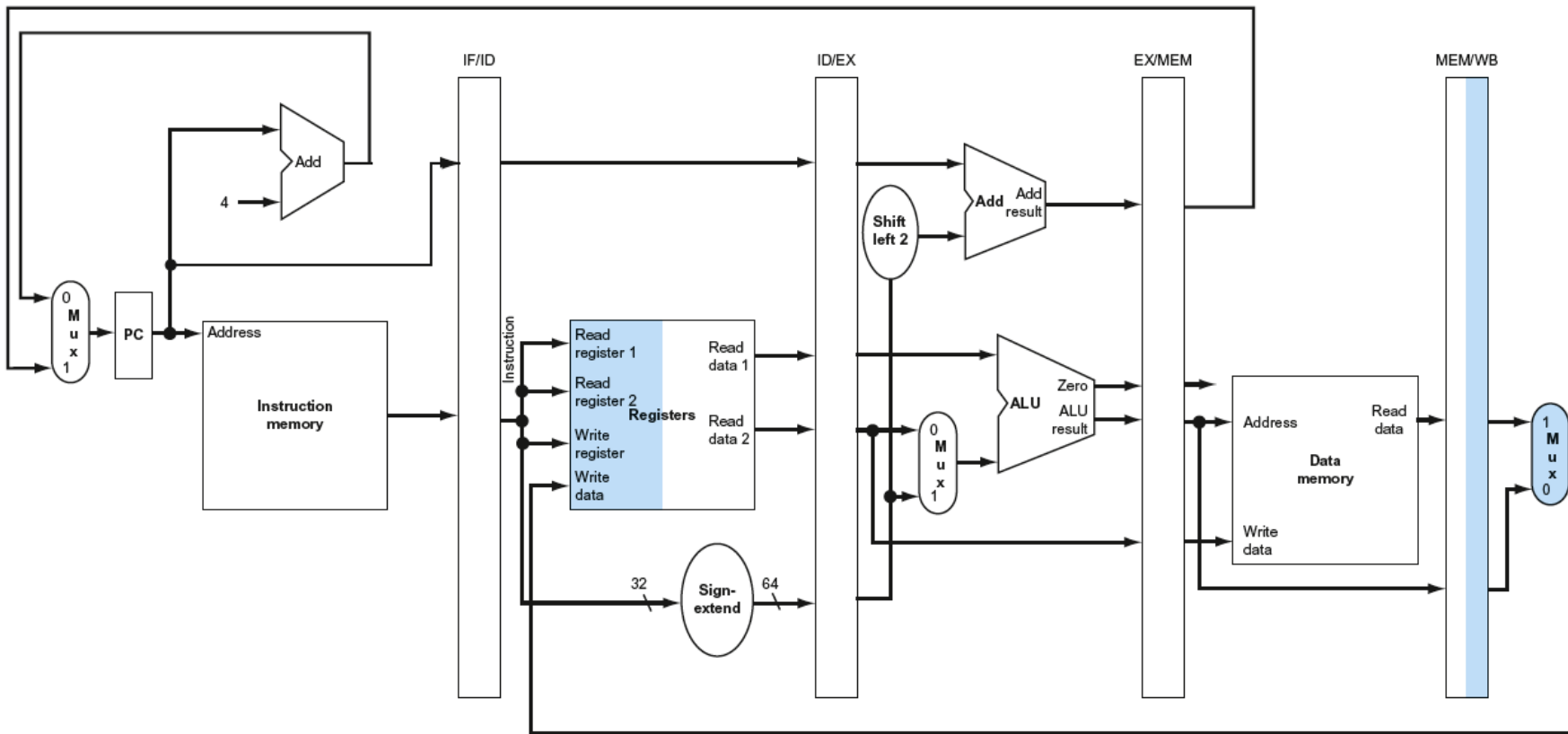
1. Read data from Mem/WB register and write to register



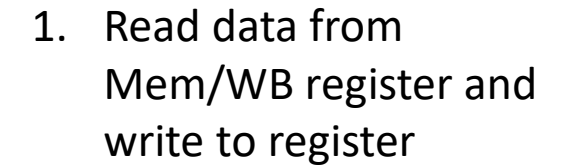
# WB for Load

LDUR  
Write-back

1. Read data from Mem/WB register and write to register



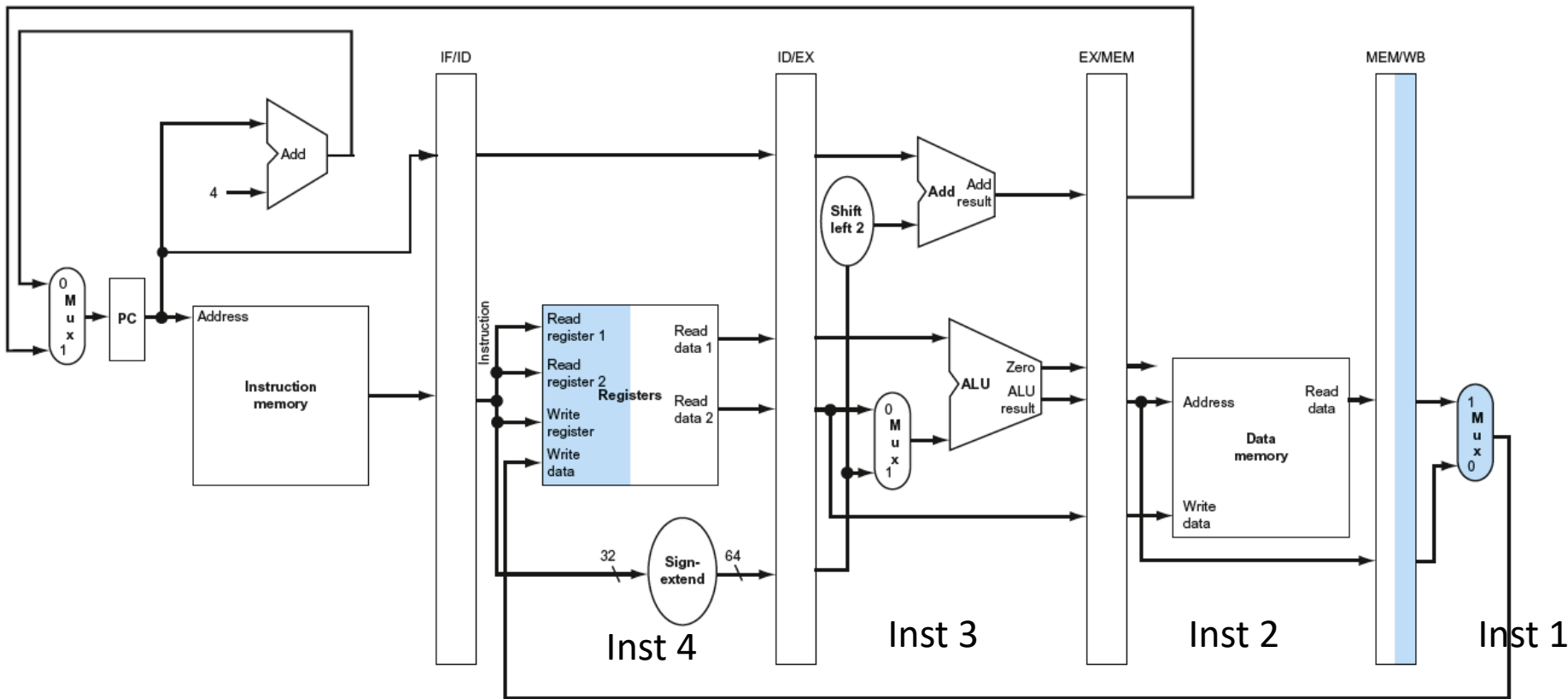




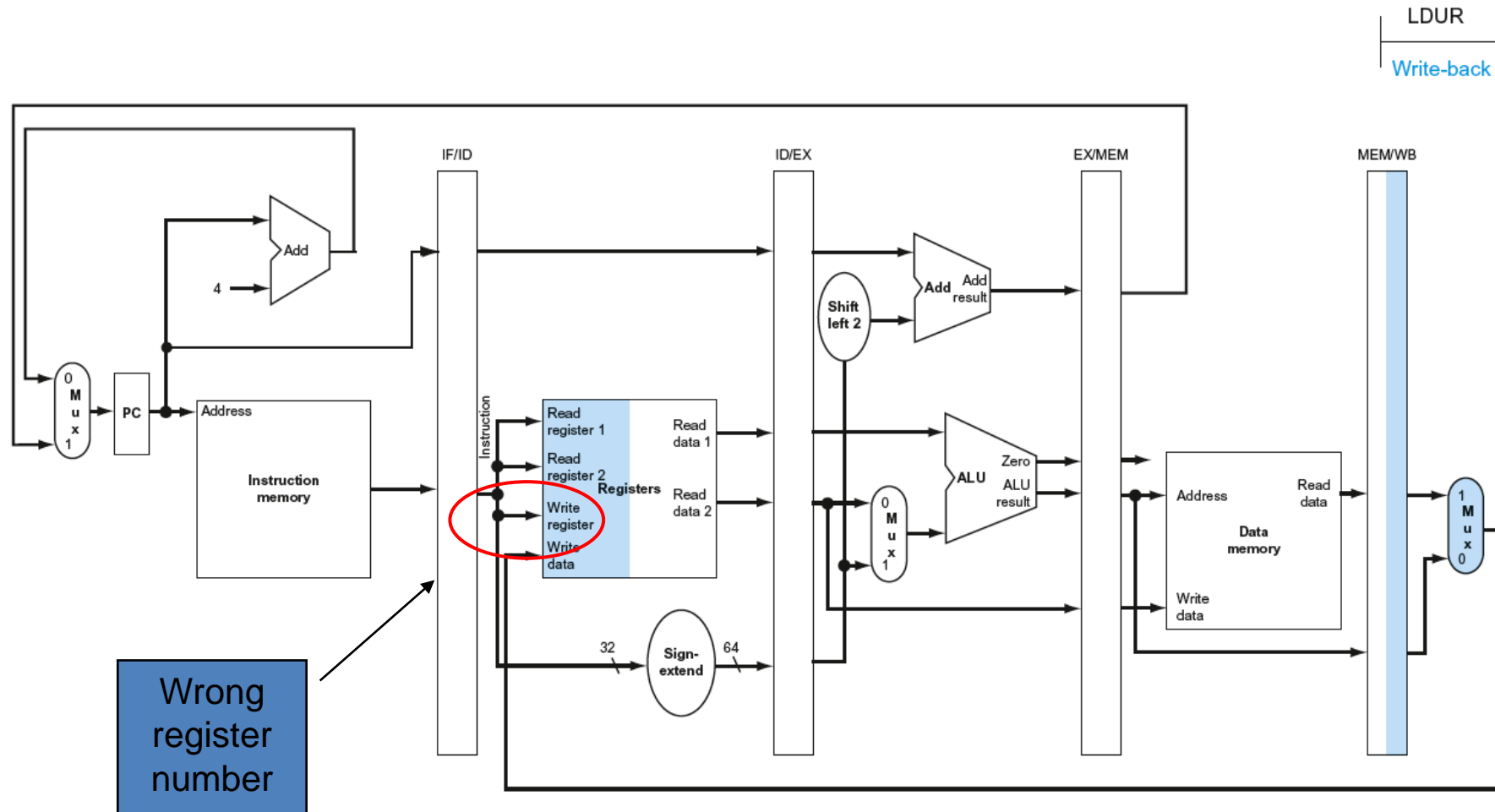
# WB for Load

LDUR  
Write-back

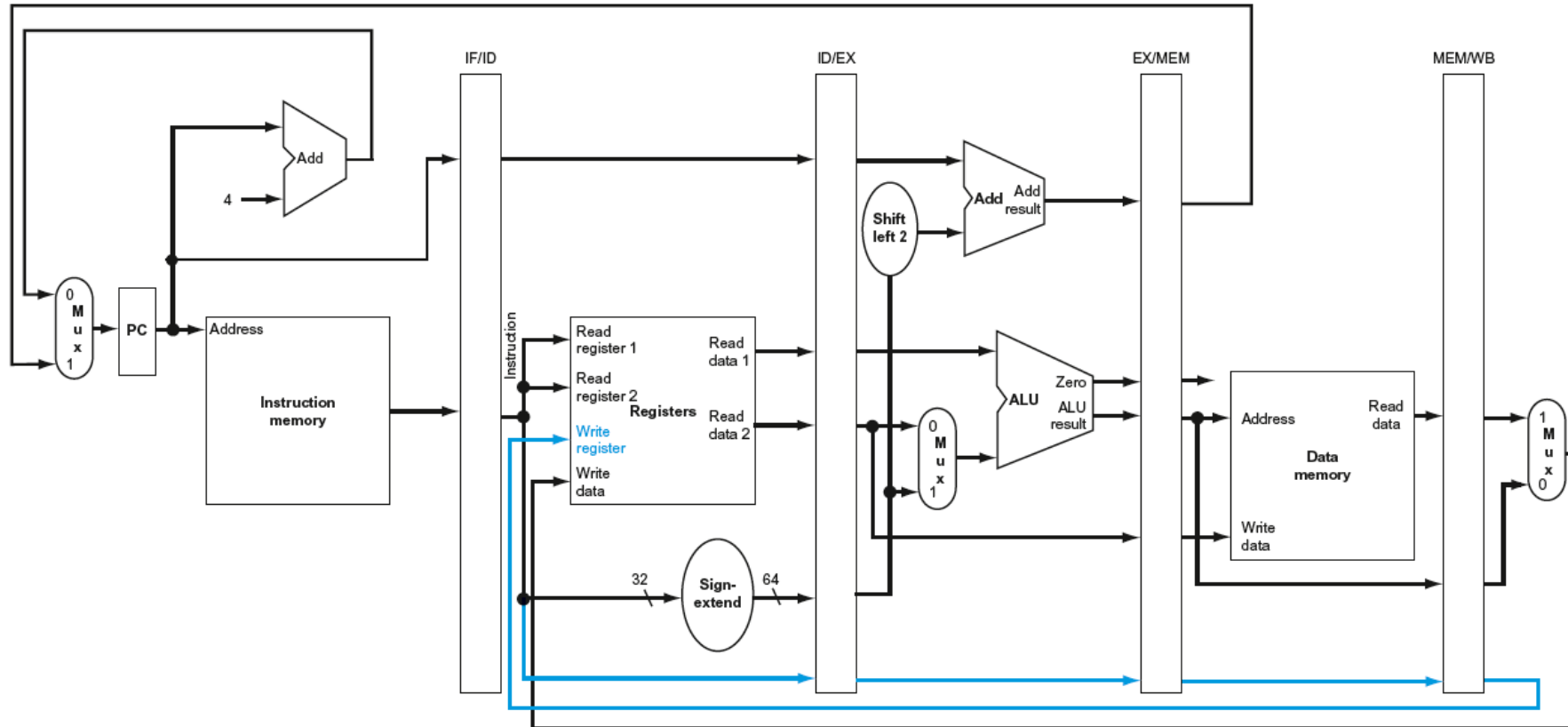
1. Read data from Mem/WB register and write to register



# WB for Load



# Corrected Datapath for Load

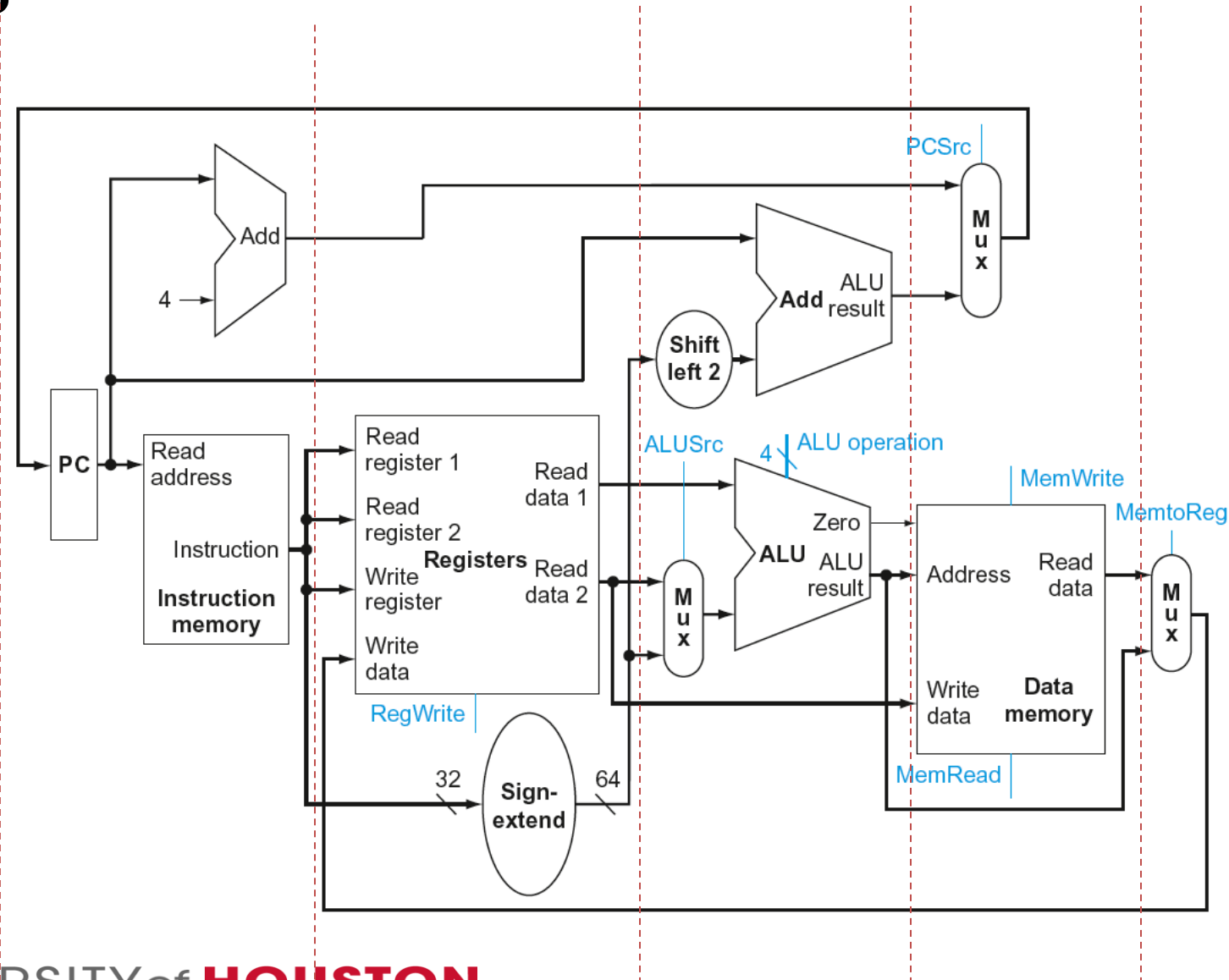


# Stages in LEGv8

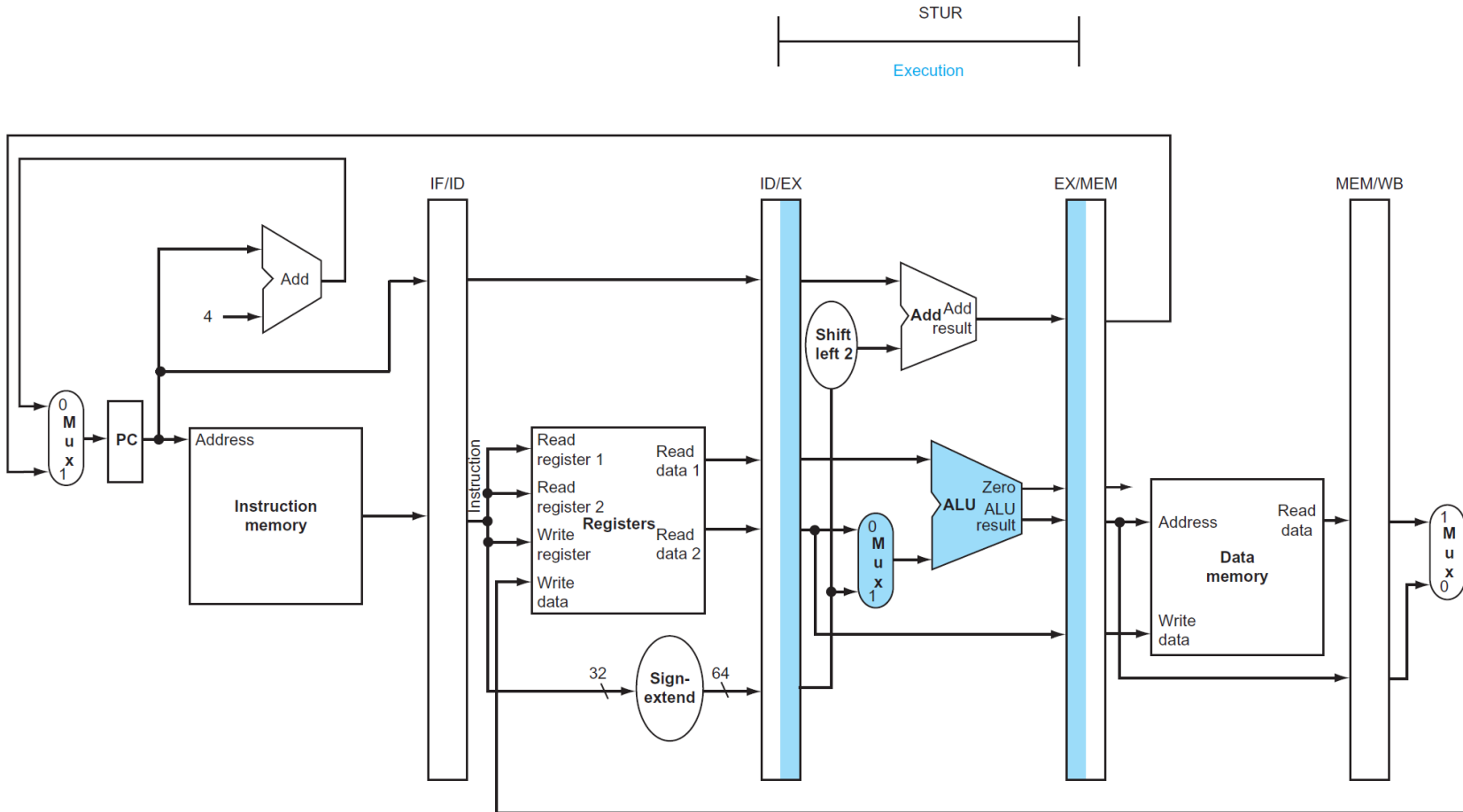
Five stages, one step per stage

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

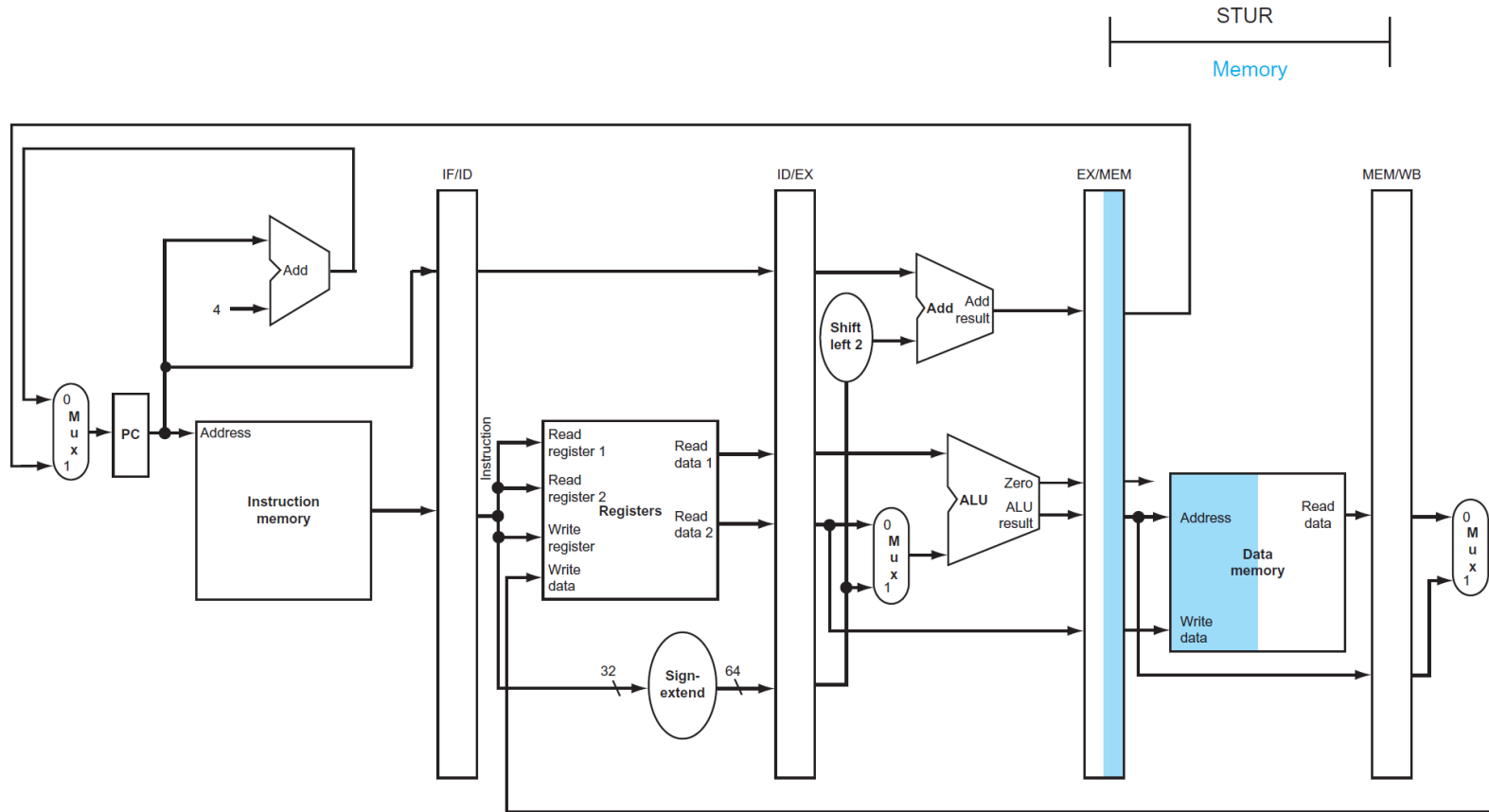
Inst./Stage	1	2	3	4	5
ADD(R format)					
<b>STUR</b>					
LDUR					
CBZ					



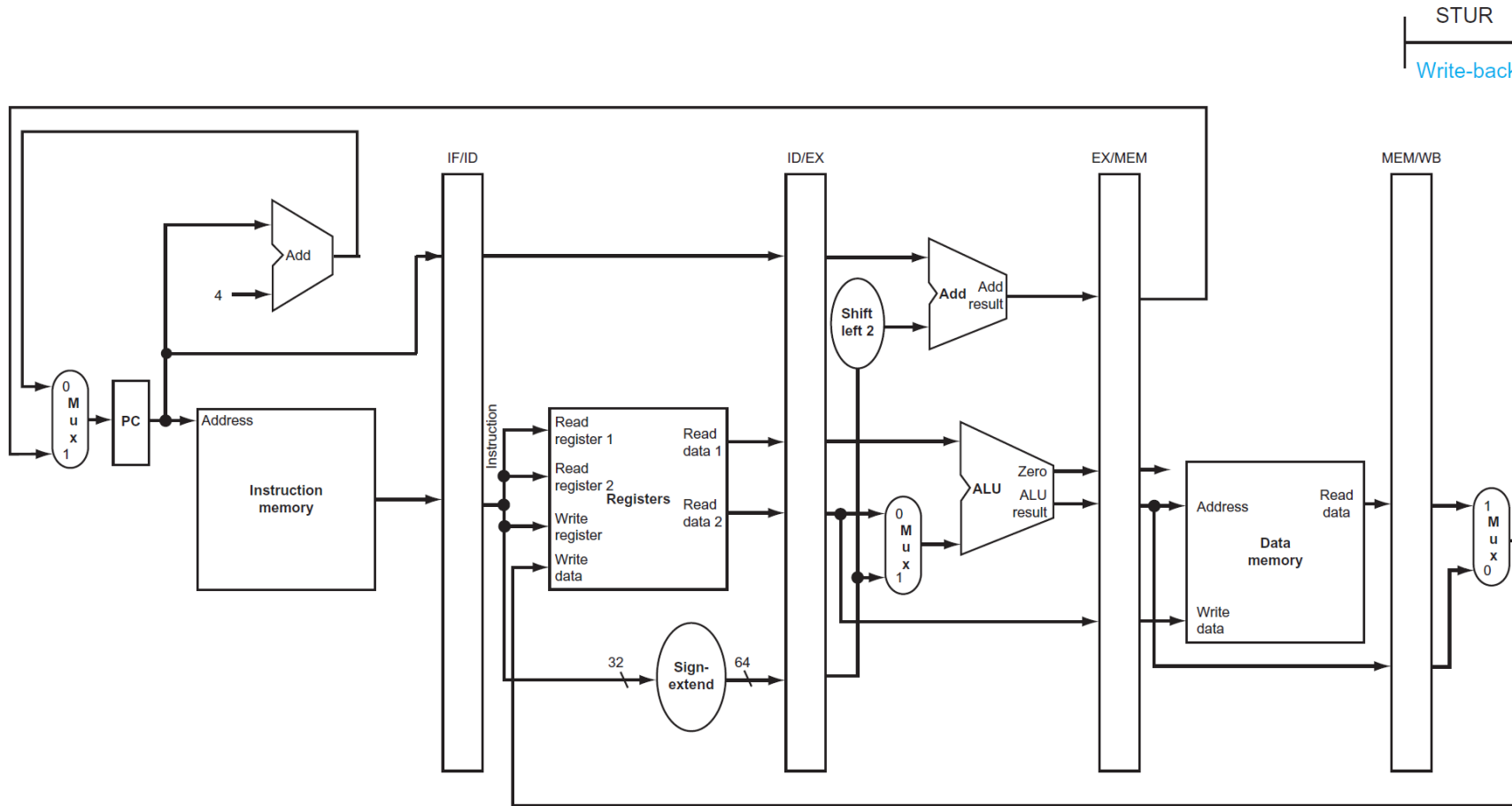
# Store instruction (Ex)



# Store instruction (MEM)

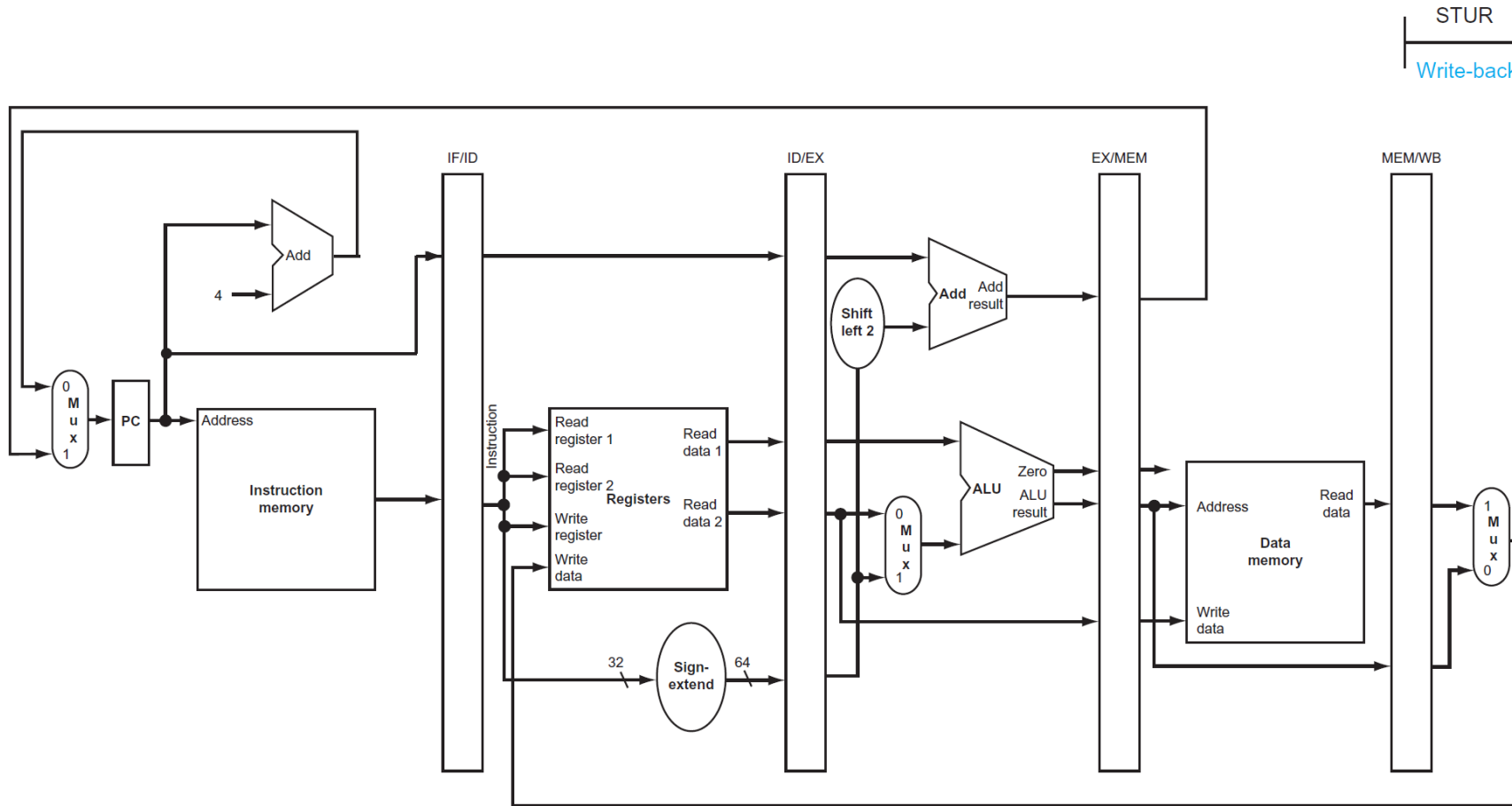


# Store instruction (WB)





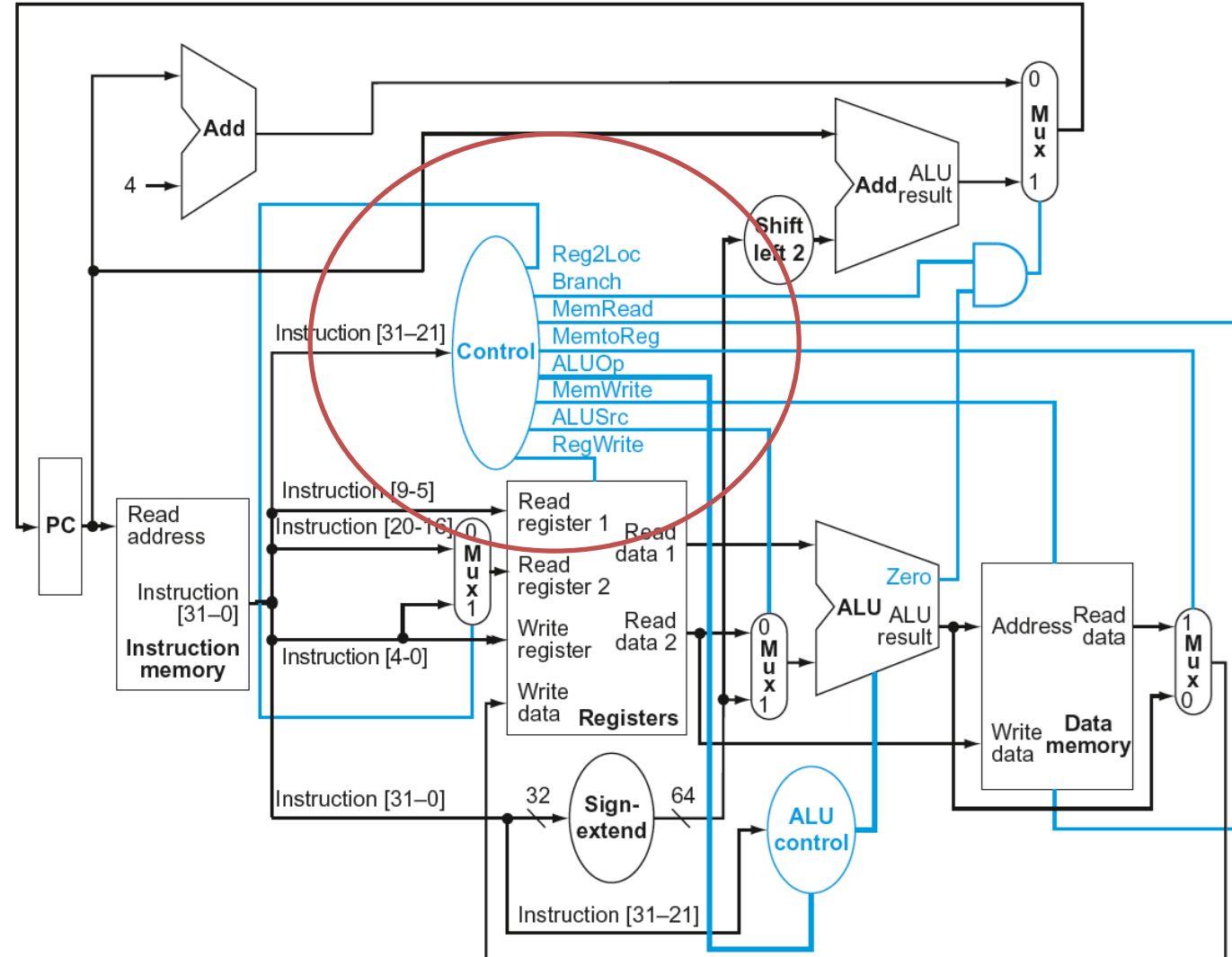
# Store instruction (WB)



**Single cycle pipeline diagram**  
Show state in each clock cycle

# Datapath With Control

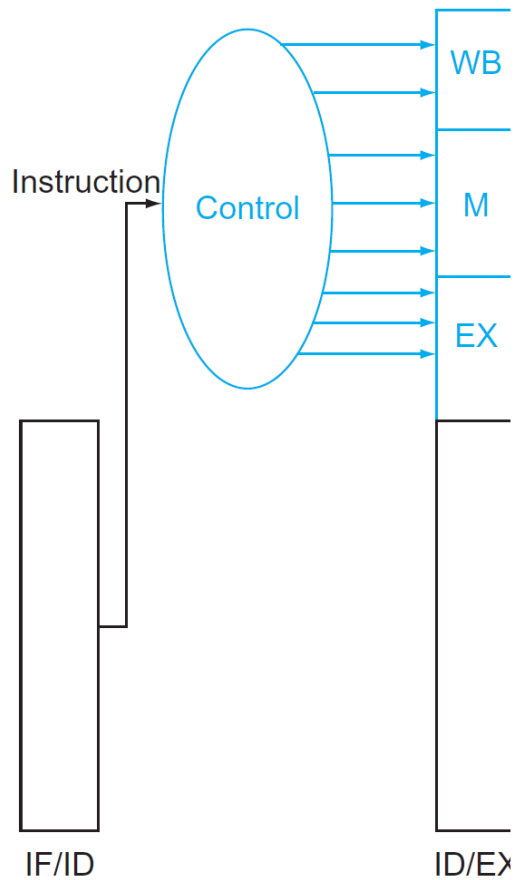
Control also needs to be pipelined.  
Only a subset of control are consumed at each stage  
Rest need to be forwarded



# Pipelined Controls

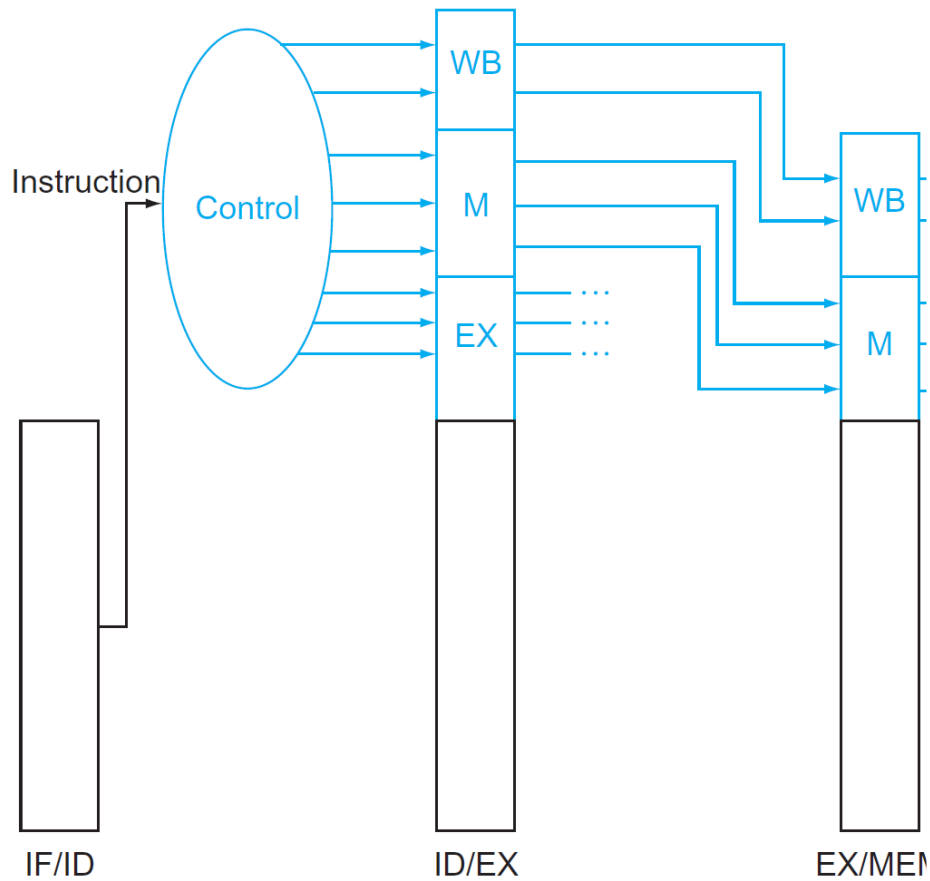
Set required controls for the current stage and forward the rest.

# Pipelined Controls



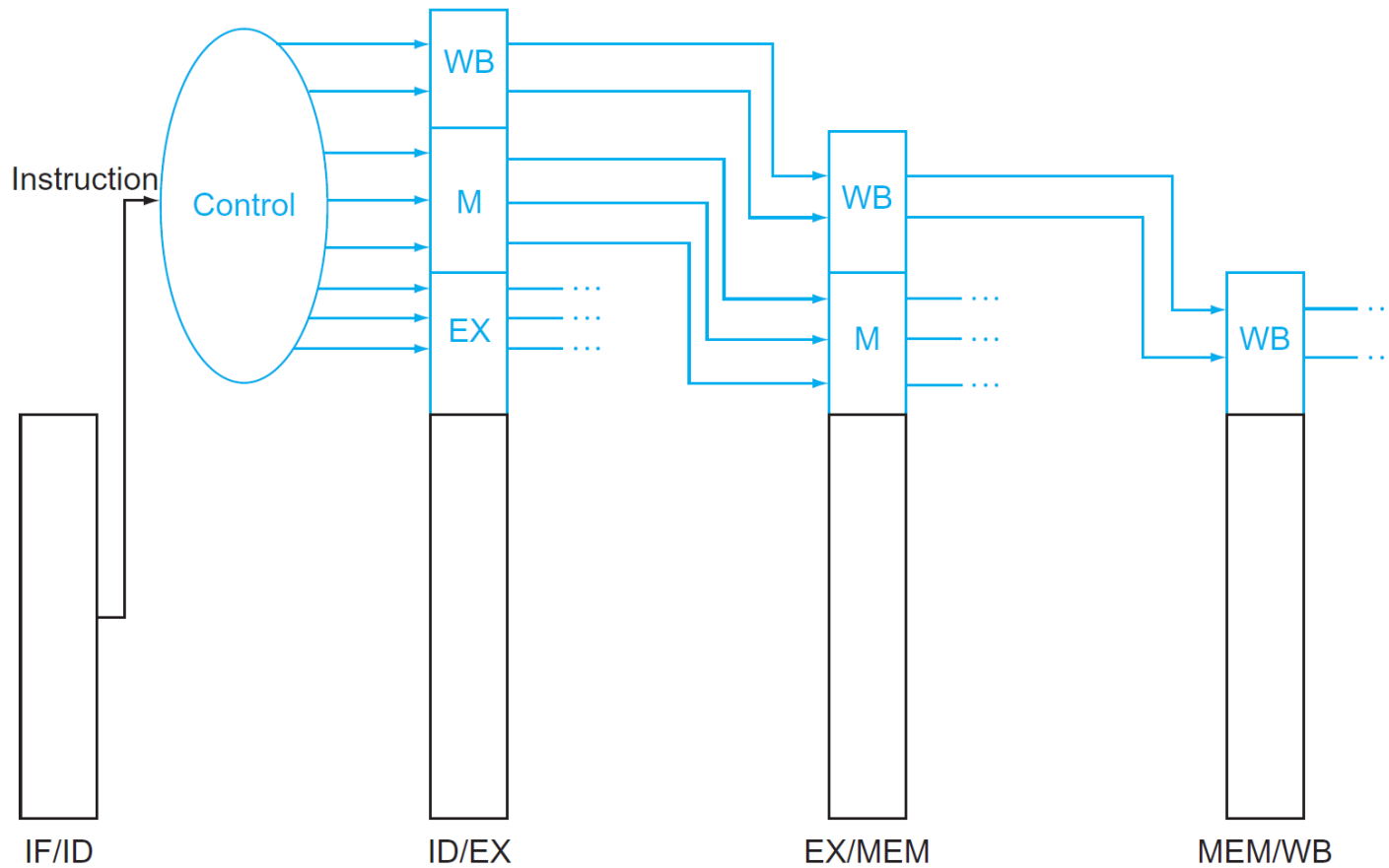
Set required controls for the current stage and forward the rest.

# Pipelined Controls



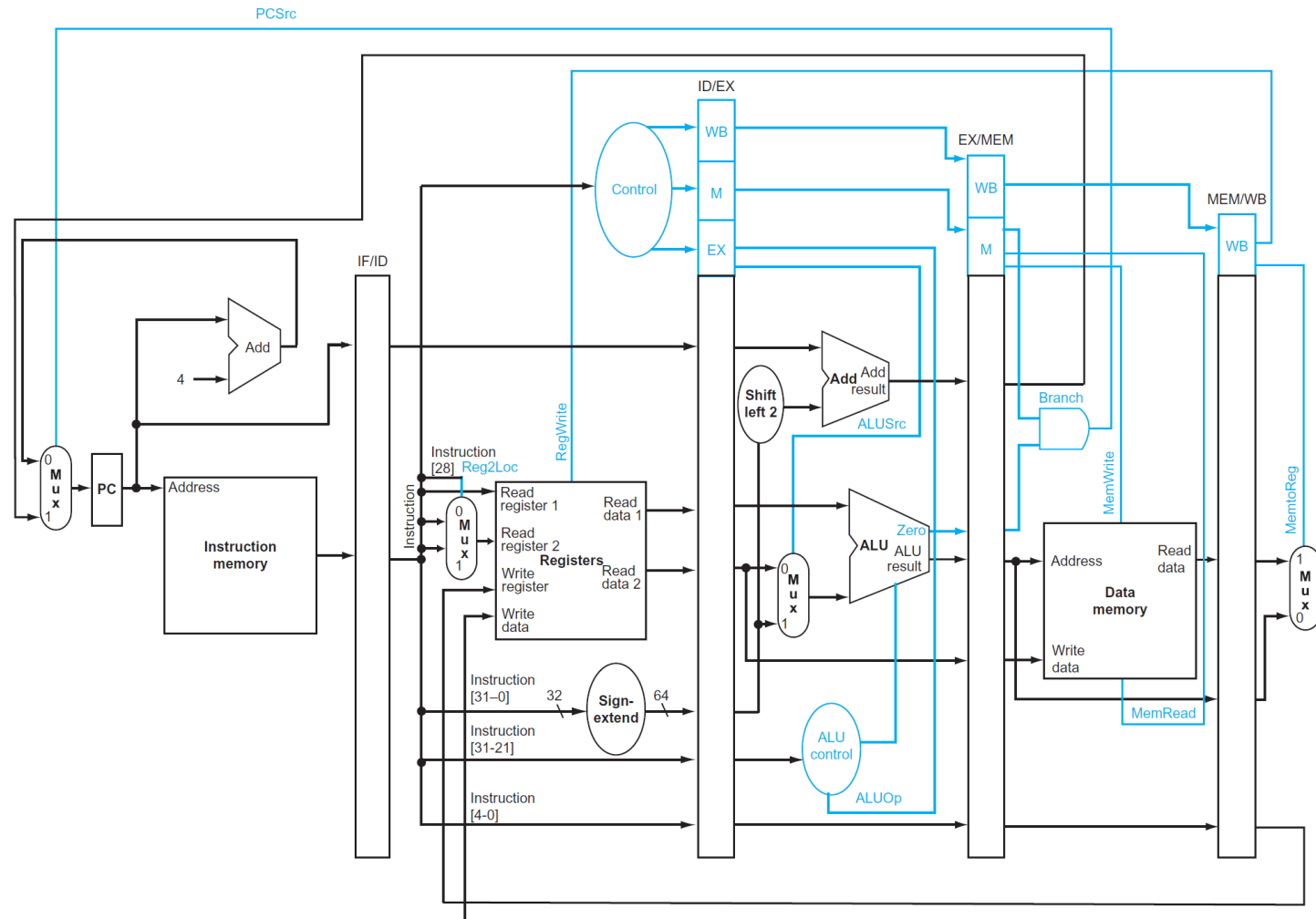
Set required controls for the current stage and forward the rest.

# Pipelined Controls



Set required controls for the current stage and forward the rest.

# Pipelined Datapath with Control Signals



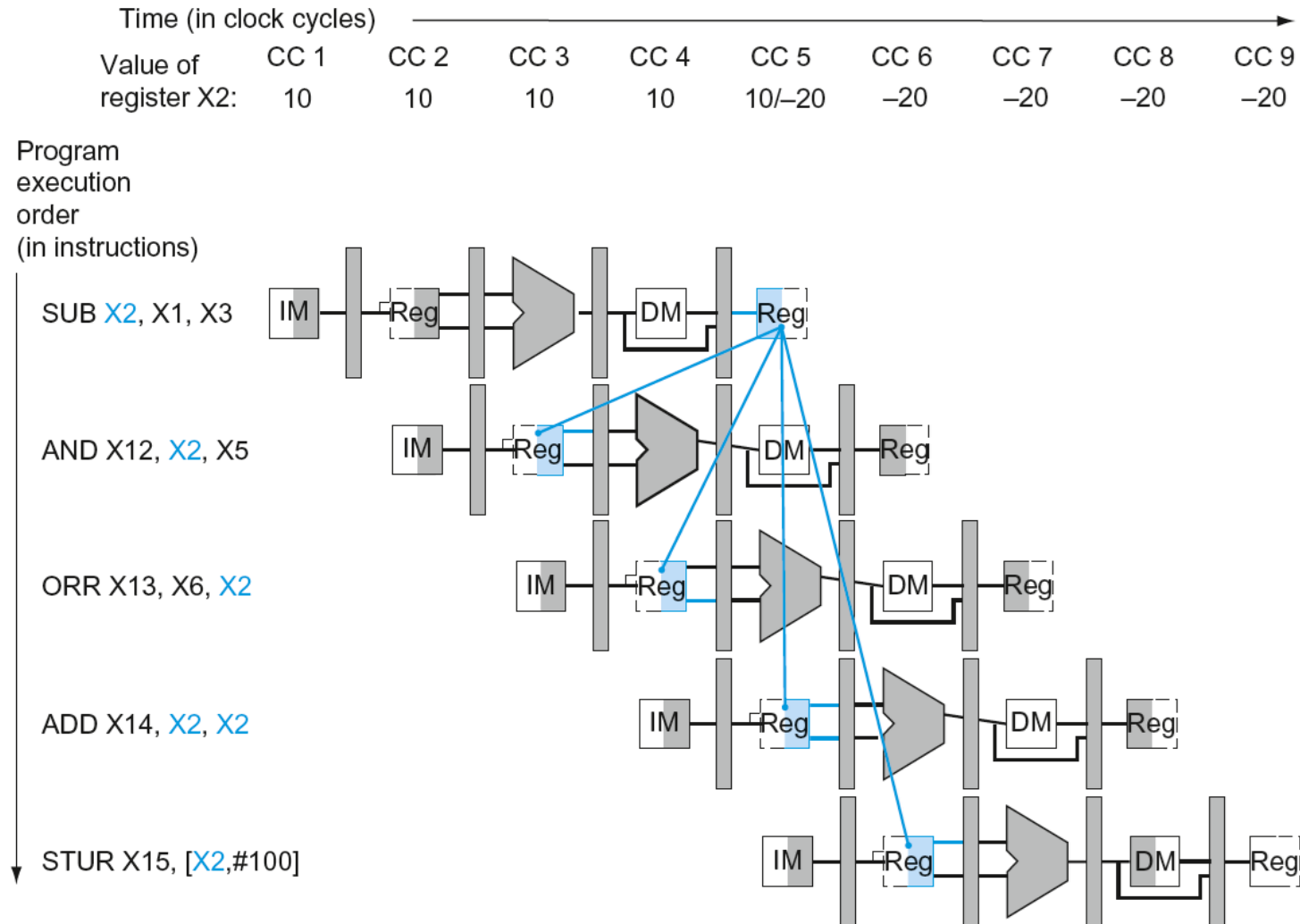
# Data Hazards in ALU Instructions

- Consider this sequence:

```
SUB  x2, x1, x3
AND  x12, x2, x5
OR   x13, x6, x2
ADD  x14, x2, x2
STUR x15, [x2, #100]
```



# Dependencies



# Data Hazards in ALU Instructions

- Consider this sequence:

```
SUB  x2, x1, x3
AND  x12, x2, x5
OR   x13, x6, x2
ADD  x14, x2, x2
STUR x15, [x2, #100]
```

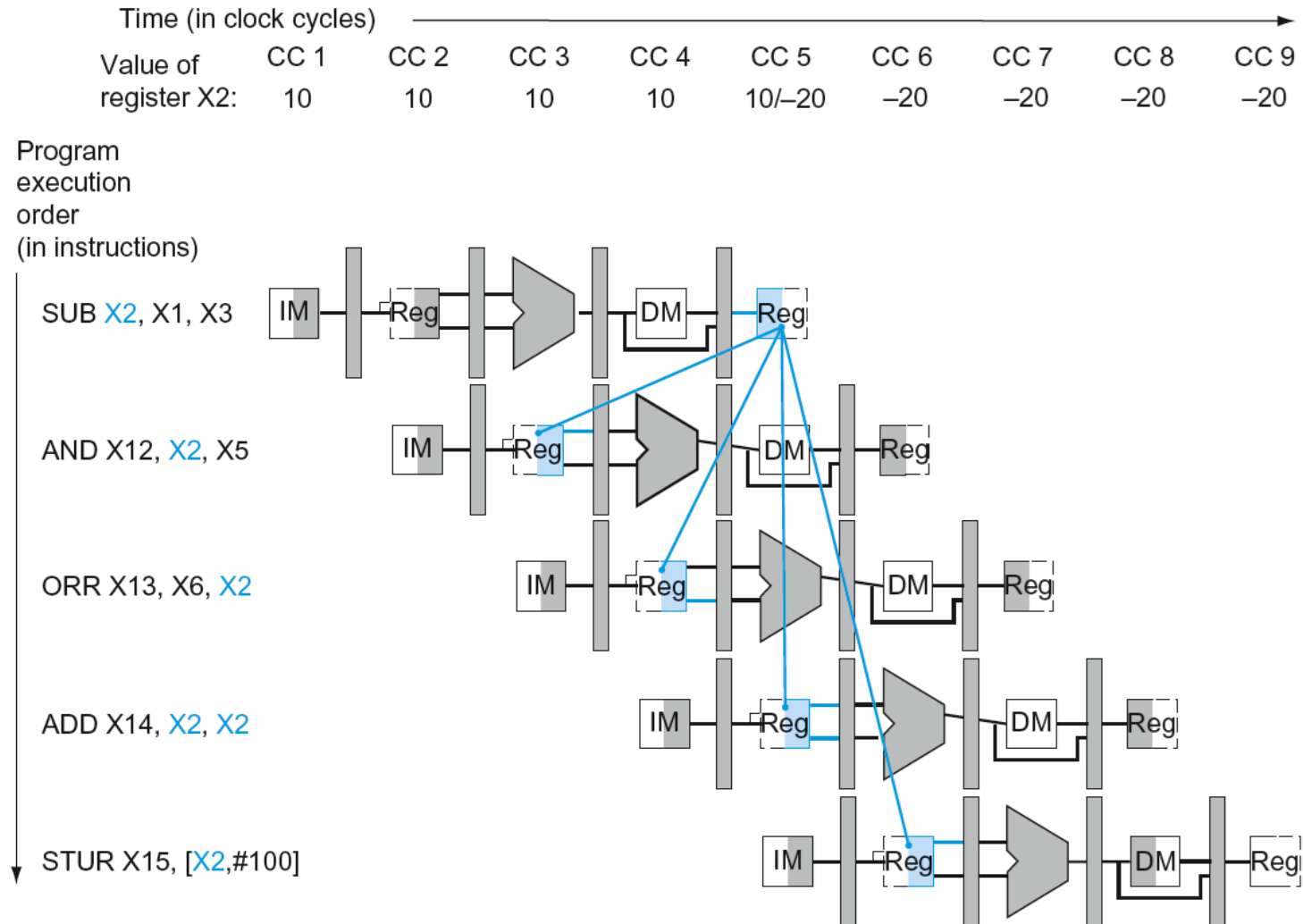
- We can resolve hazards with forwarding
  - How do we detect when to forward?

# Data Hazard

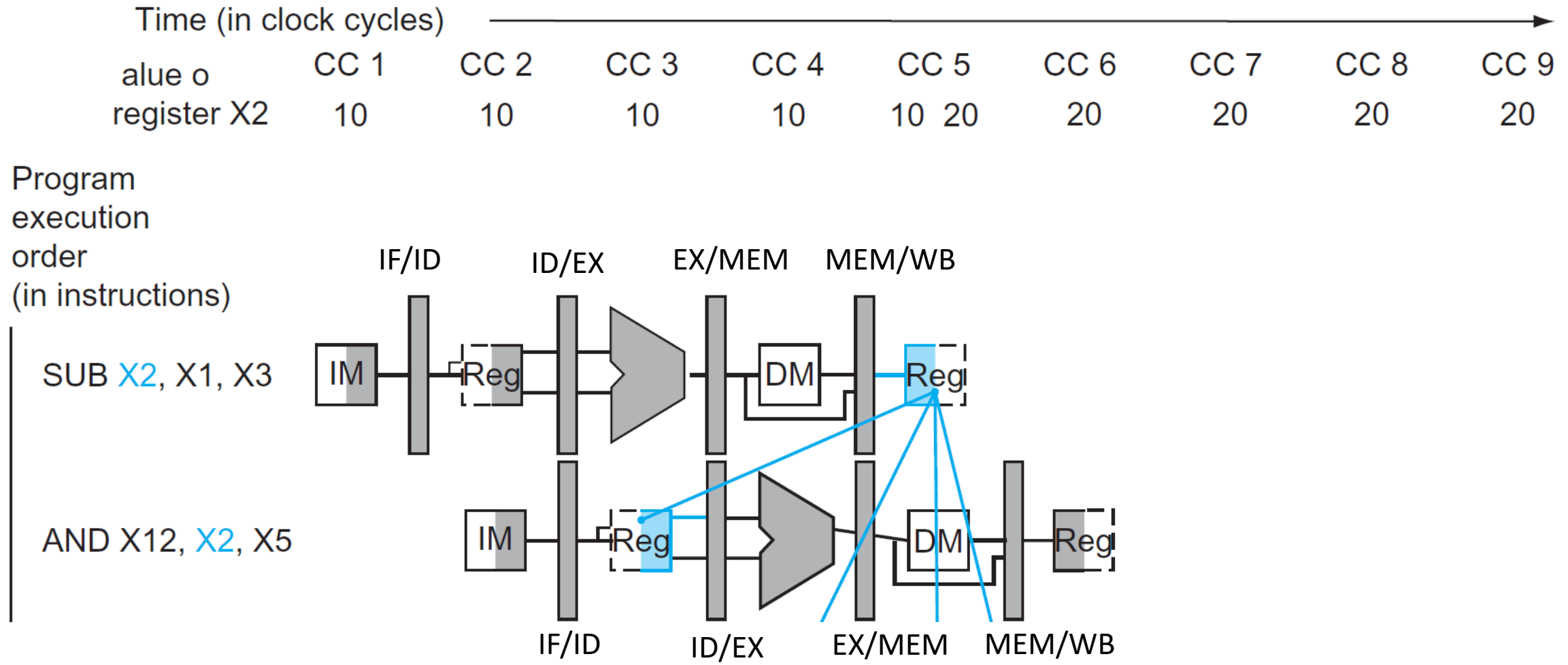
Hardware to

1. Identify Hazards
2. Forwarding
3. Stalls

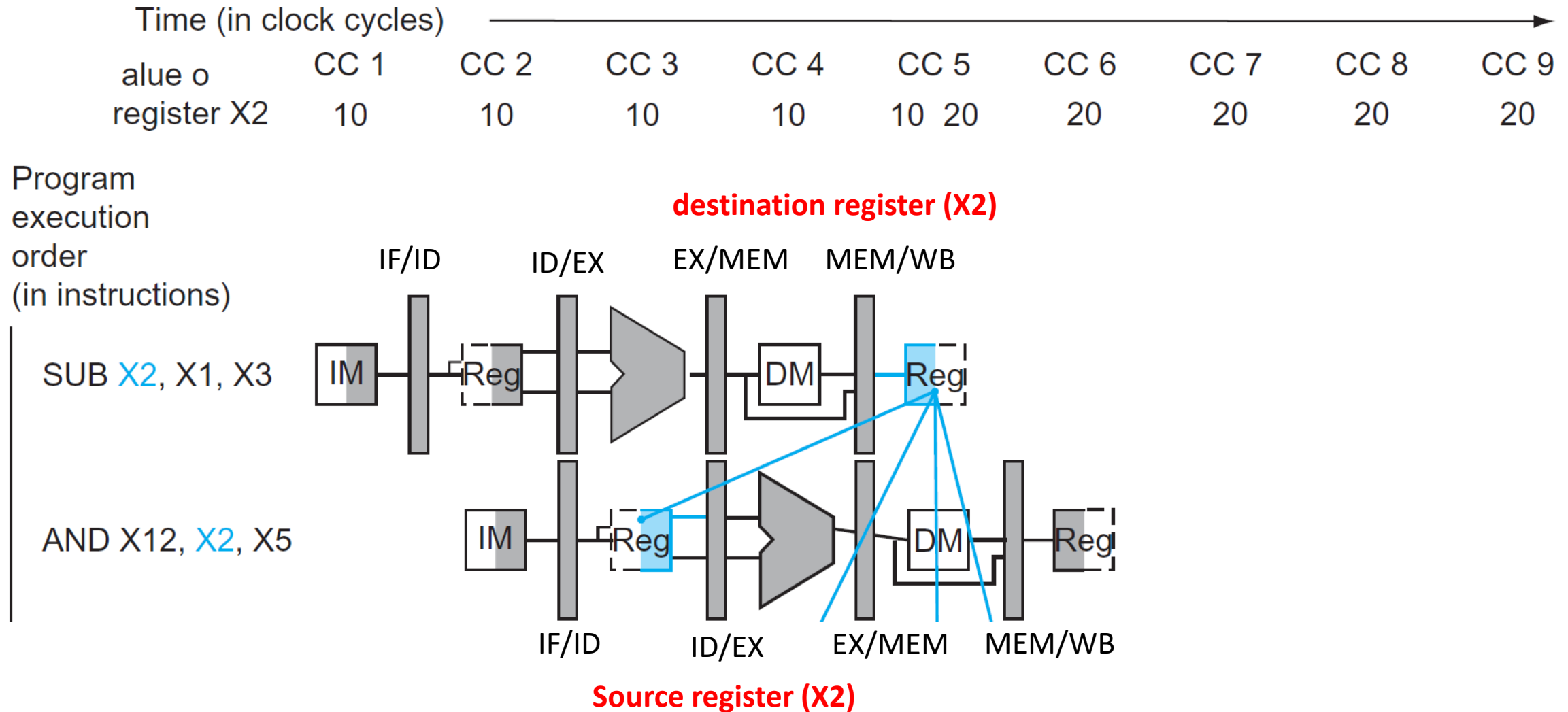
# Identify Data hazard



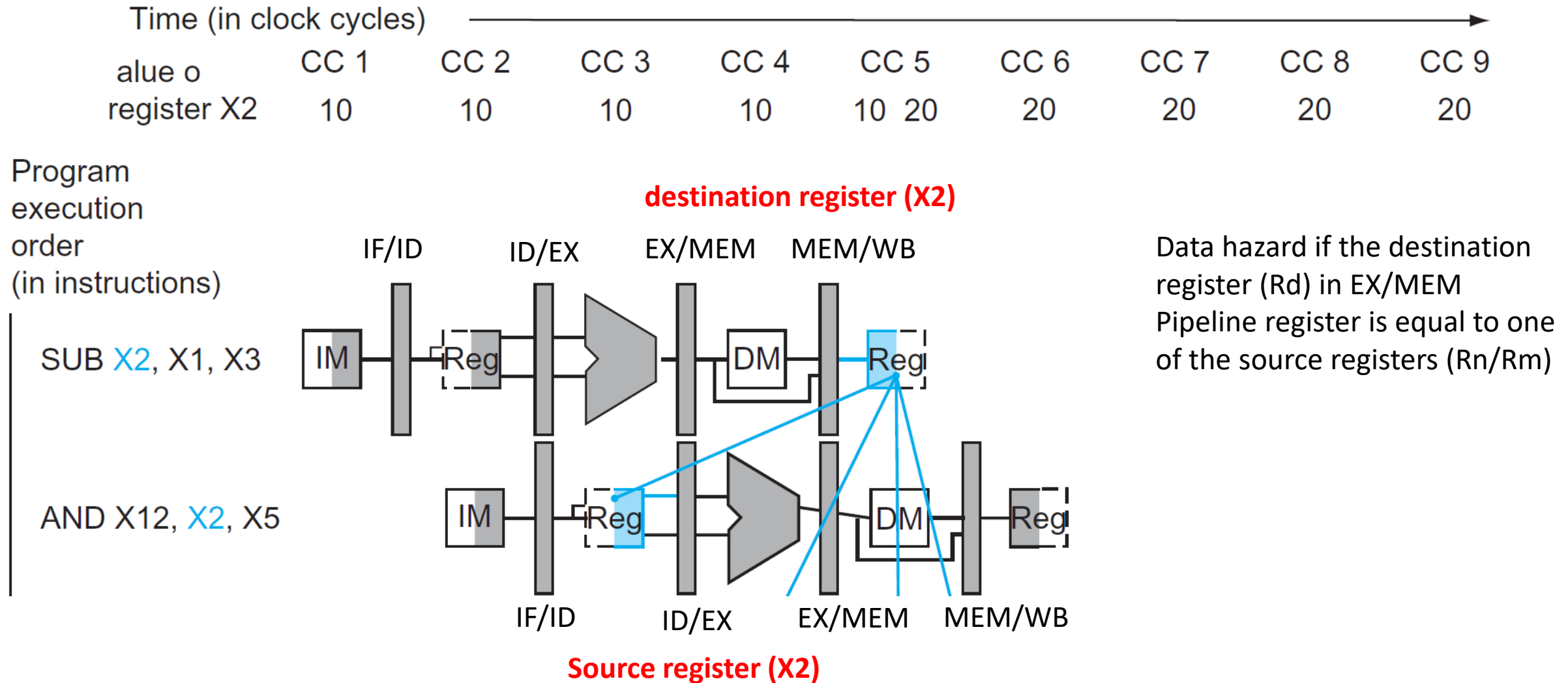
# Identify Data hazard



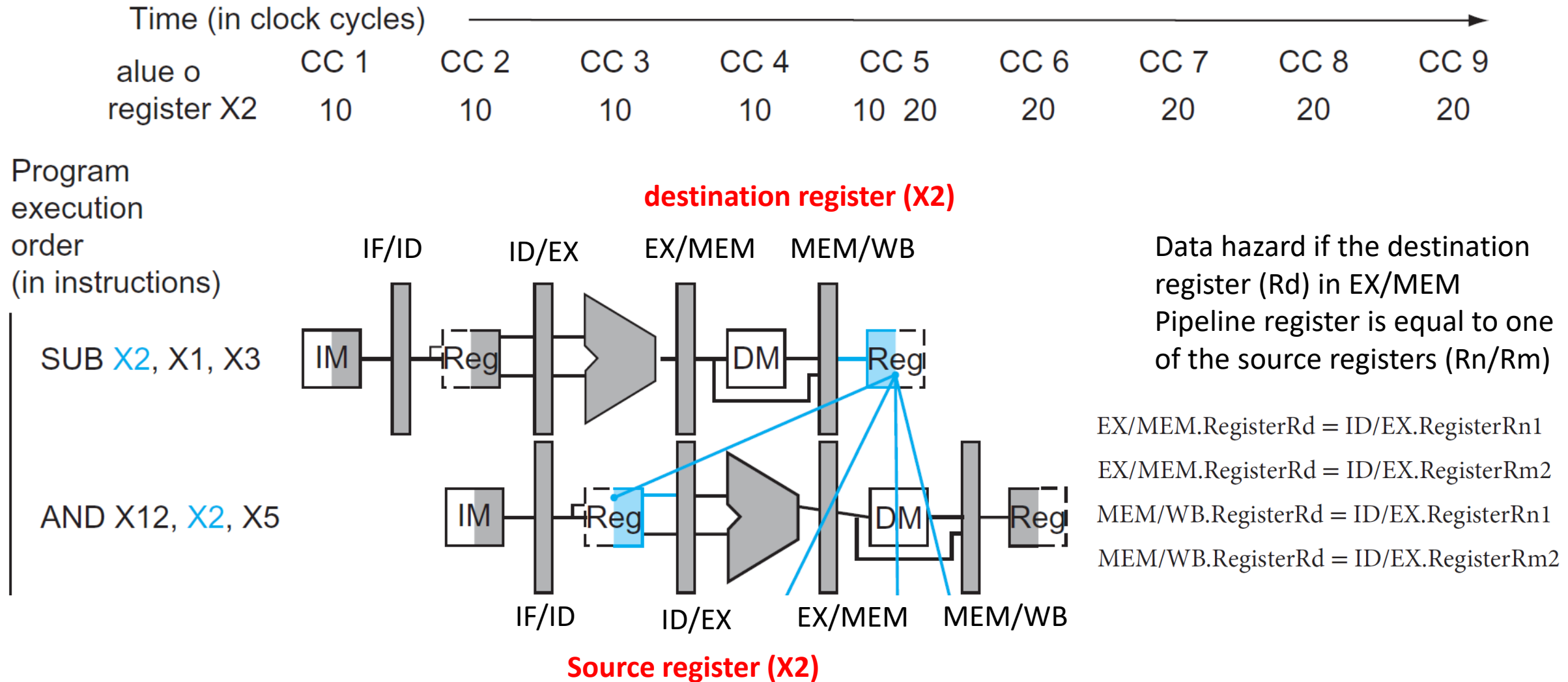
# Identify Data hazard



# Identify Data hazard

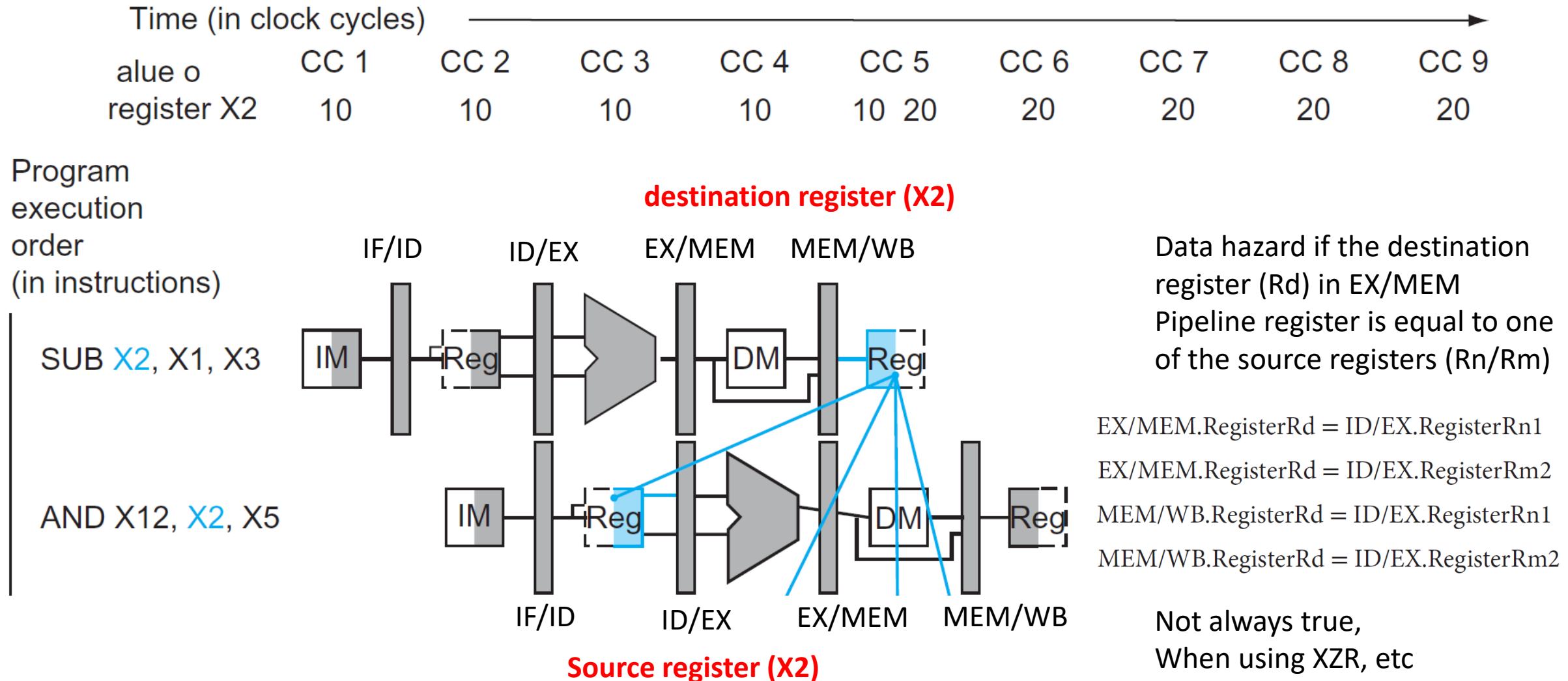


# Identify Data hazard

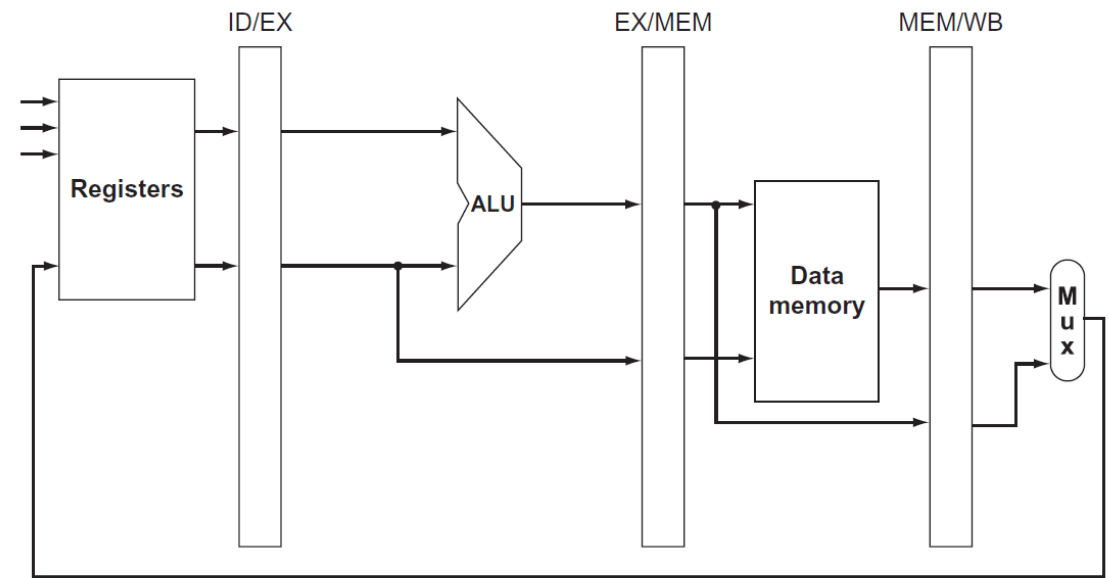




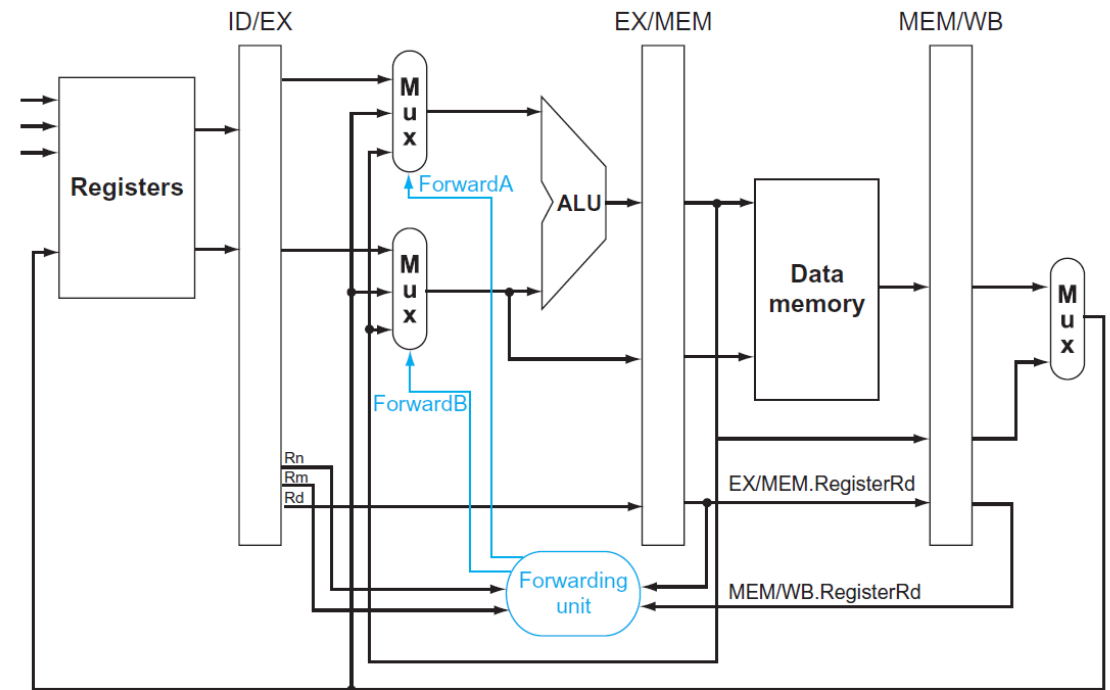
# Identify Data hazard



# Forwarding



a. No forwarding

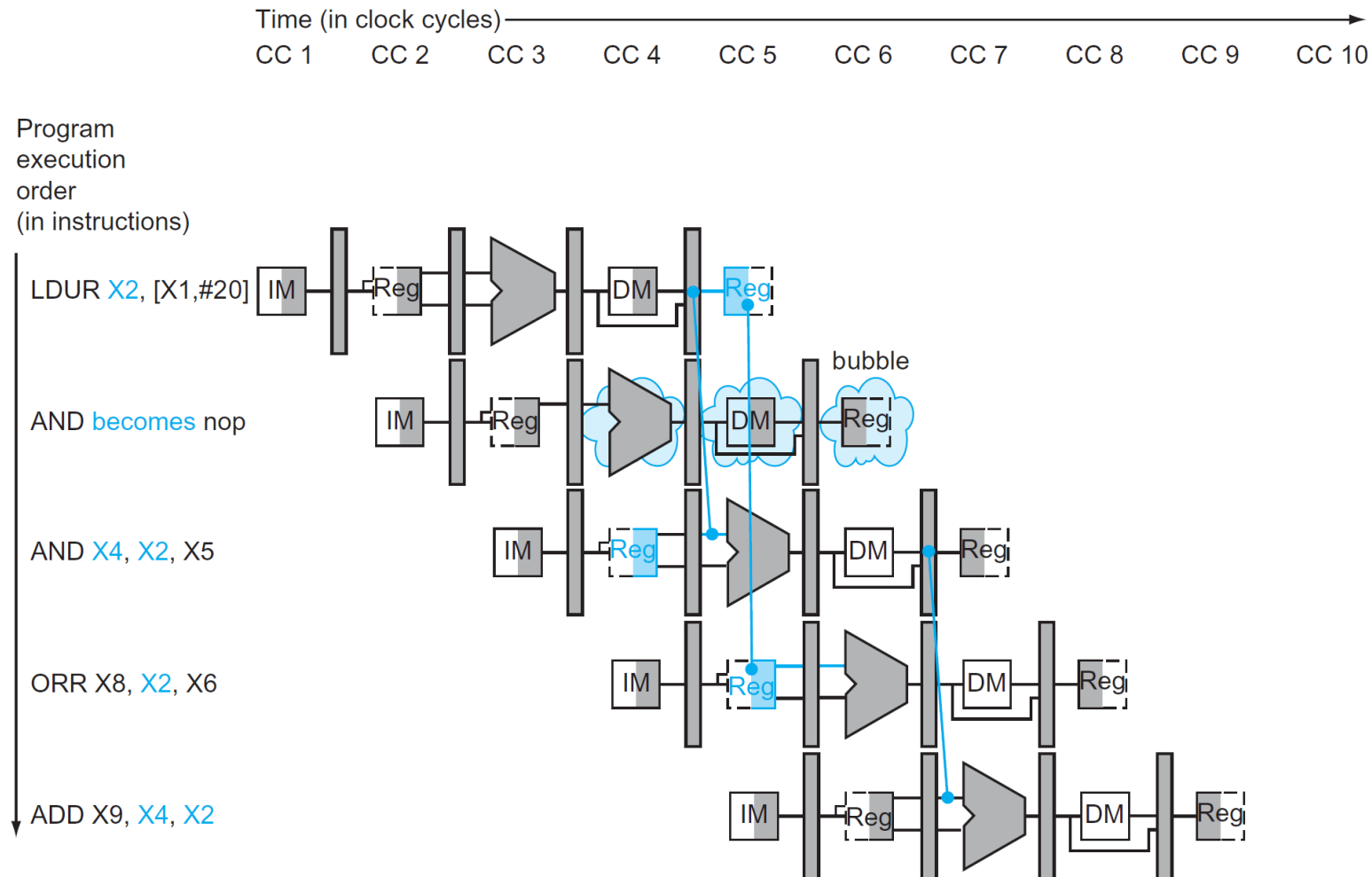


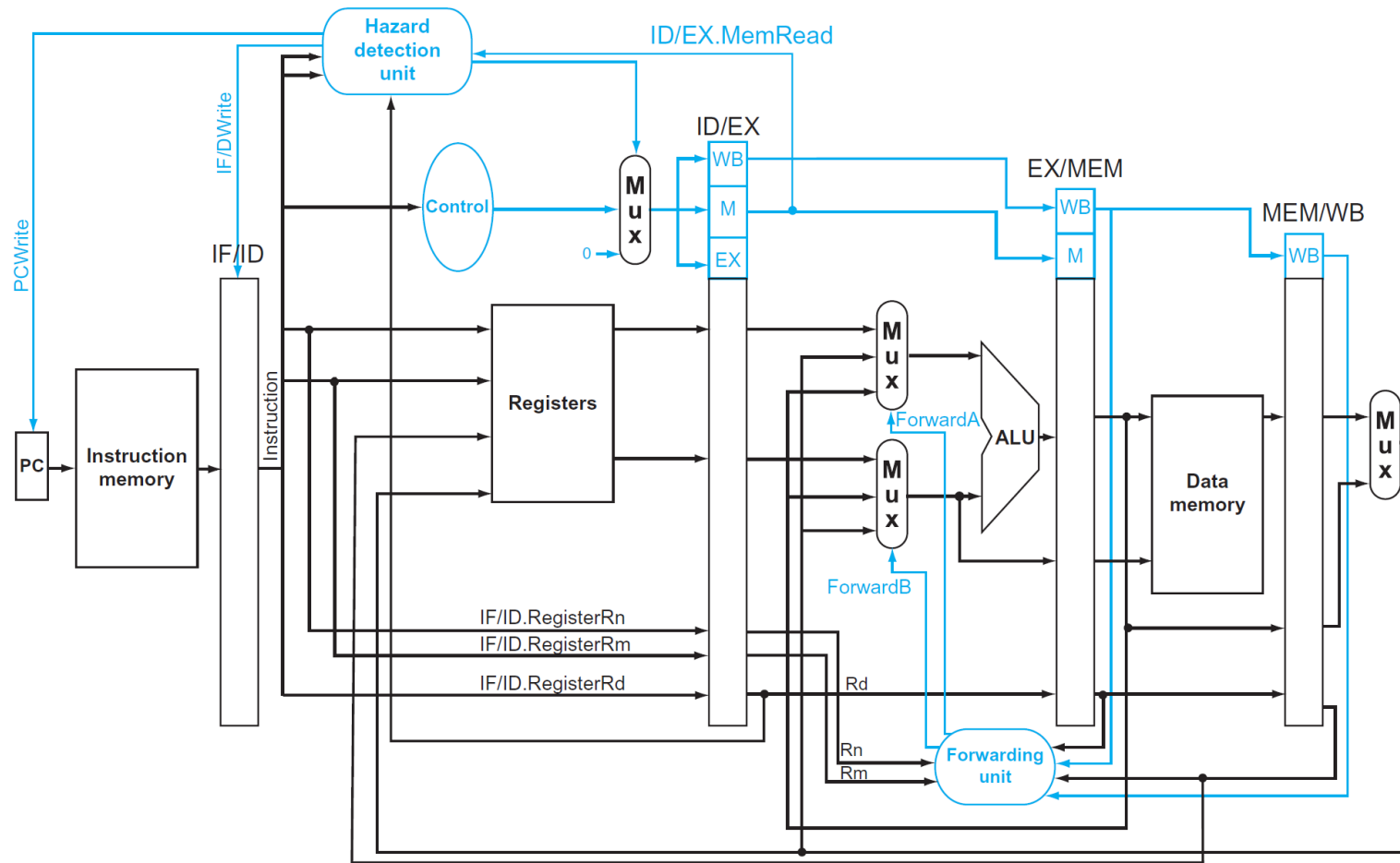
b. With forwarding

# 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 instruction is decoded again
  - Following instruction is fetched again

# Stalling/ Inserting NOP





Hazard detection,  
Set controls for  
EX, MEM, WB to 0 in the ID/EX  
register

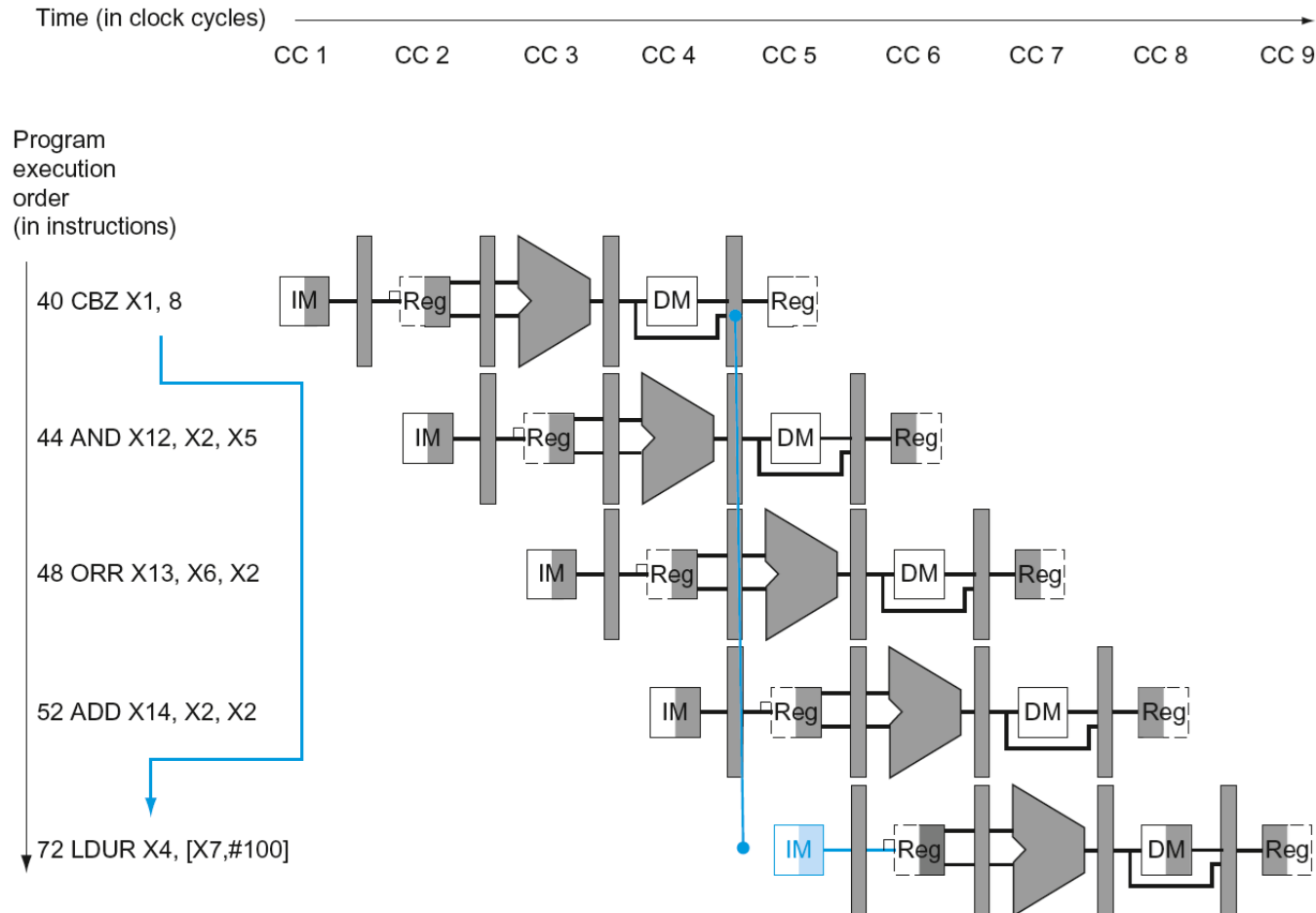
# Stalls and Performance

## The BIG Picture

- 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

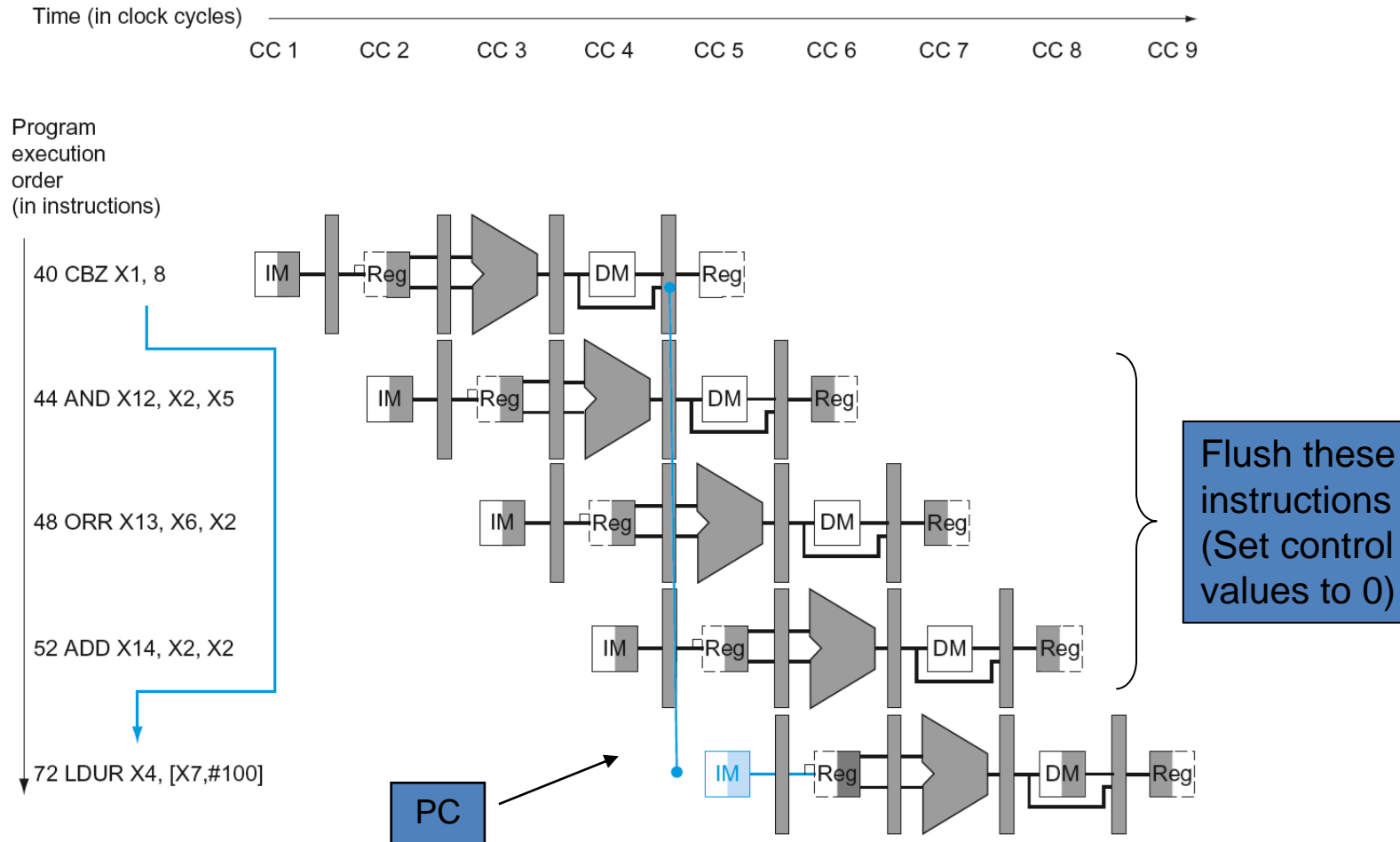
# Branch Hazards

- If branch outcome determined in MEM



# Branch Hazards

- If branch outcome determined in MEM



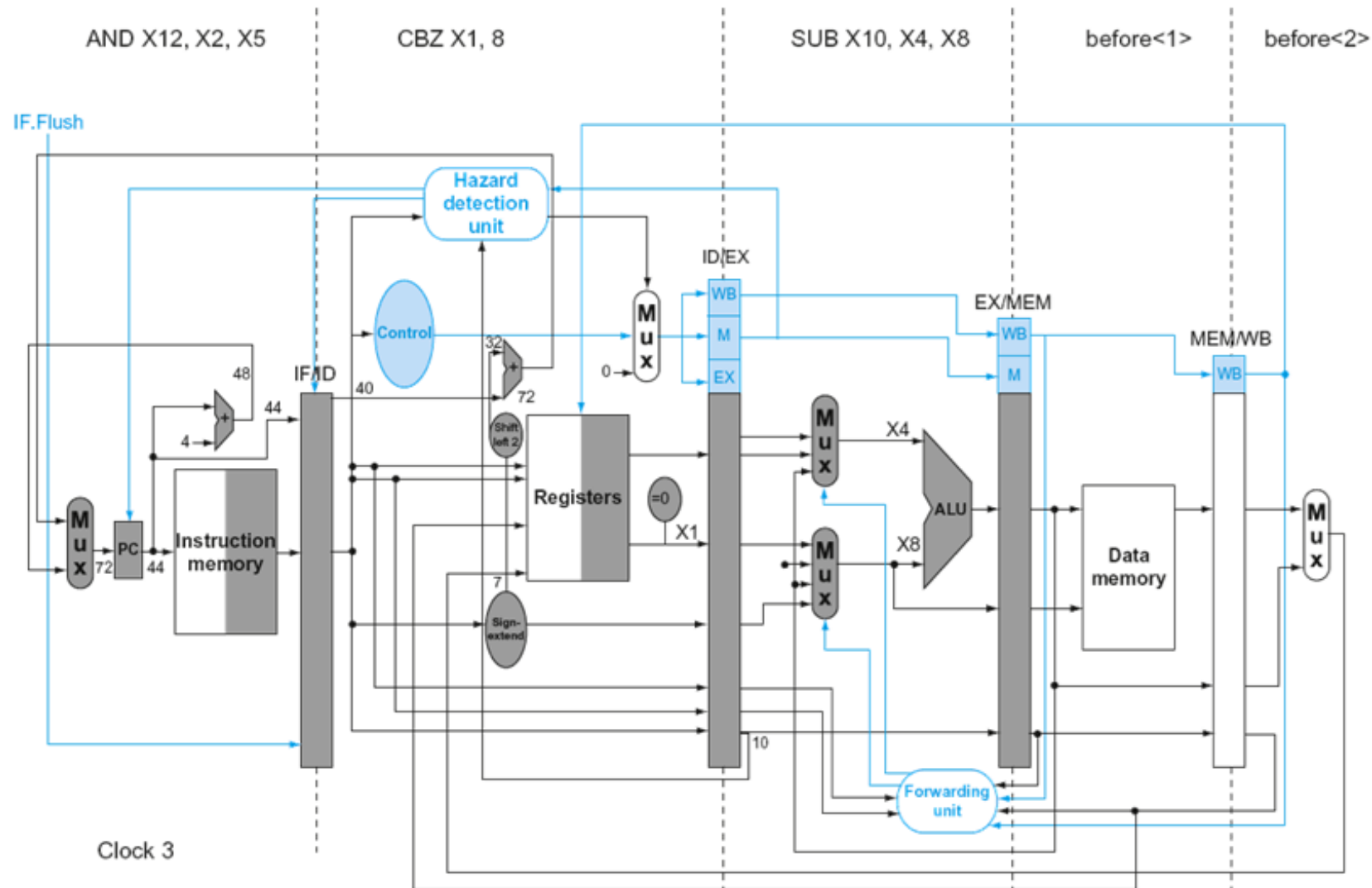


# Reducing Branch Delay

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

```
36:  SUB  X10, X4, X8
40:  CBZ  X1,  X3, 8
44:  AND  X12, X2, X5
48:  ORR  X13, X2, X6
52:  ADD  X14, X4, X2
56:  SUB  X15, X6, X7
...
72:  LDUR X4, [X7,#50]
```

# Reducing Branch Delay



# Branch Prediction

- Predict the outcome of the branch
  - Predict Not taken
  - 1-Bit predictor
  - 2-bit predictor

# Predict not taken

```
for (i =0; i< 100, i++){  
    a += 1  
}
```

i in register X0

a in register X1

ADD X0, XZR, XZR

Loop: ADDI X1, X1, #1

ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Next Instruction

# Predict not taken

```
for (i =0; i< 100, i++){  
    a += 1  
}
```

i in register X0

a in register X1

ADD X0, XZR, XZR

Loop: ADDI X1, X1, #1

ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Next Instruction

iteration	Prediction (T/NT)	Actual(T/NT)
i = 0	NT	

# Predict not taken

```
for (i =0; i< 100, i++){
    a += 1
}
```

i in register X0

a in register X1

```
    ADD X0, XZR, XZR
```

```
Loop: ADDI X1, X1, #1
```

```
    ADDI X0, X0, #1
```

```
    SUBI X2, X0, #100
```

```
    CBNZ X2, Loop
```

Next Instruction    In IF stage,  
                          So Flush  
                          Penalty 1 clock  
                          cycle

iteration	Prediction (T/NT)	Actual(T/NT)
i = 0	NT	T

# Predict not taken

```
for (i =0; i< 100, i++){
    a += 1
}
```

i in register X0

a in register X1

ADD X0, XZR, XZR

Loop: ADDI X1, X1, #1

ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Next Instruction    In IF stage,  
                               So Flush  
                               Penalty 1 clock  
                               cycle

iteration	Prediction (T/NT)	Actual(T/NT)
I = 0	NT	T
I = 1	NT	T

# Predict not taken

```
for (i =0; i< 100, i++){
    a += 1
}
```

i in register X0

a in register X1

```
    ADD X0, XZR, XZR
```

```
Loop: ADDI X1, X1, #1
```

```
    ADDI X0, X0, #1
```

```
    SUBI X2, X0, #100
```

```
    CBNZ X2, Loop
```

Next Instruction    In IF stage,  
                          So Flush  
                          Penalty 1 clock  
                          cycle

iteration	Prediction (T/NT)	Actual(T/NT)
I = 0	NT	T
I = 1	NT	T
...		
I = 100	NT	NT



# Predict not taken

```
for (i =0; i< 100, i++){
    a += 1
}
```

i in register X0

a in register X1

```
ADD X0, XZR, XZR
```

```
Loop: ADDI X1, X1, #1
```

```
ADDI X0, X0, #1
```

```
SUBI X2, X0, #100
```

```
CBNZ X2, Loop
```

Next Instruction    In IF stage,  
                          So Flush  
                          Penalty 1 clock  
                          cycle

iteration	Prediction (T/NT)	Actual(T/NT)
I = 0	NT	T
I = 1	NT	T
...		
I = 100	NT	NT

# Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction

# Dynamic branch prediction

- Algorithms using the previous execution of a branch to predict the outcome of the next execution
- Several techniques for dynamic branch prediction
  - 1bit branch prediction buffer
  - 2bit branch prediction buffer
  - Correlating Branch Prediction Buffer
  - Tournament predictors
- We only deal with 1 bit and 2bit branch predictors here

# 1-bit predictor

- Predict the outcome of the next branch instruction to be the same as the previous outcome.
  - If previous outcome is taken, predict next as taken
  - If previous outcome is not taken then predict next as not taken.
- This can be achieved using 1 memory bit.
  - predictor value 0: predicts branch not taken
  - predictor value 1: predicts branch taken
- Flip the bit on incorrect prediction

# 1-bit predictor example

predictor value 0: predicts branch not taken

predictor value 1: predicts branch taken

ADD X0, XZR, XZR

Loop: ADDI X1, X1, #1

ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	0			
1				
2				
3				

# 1-bit predictor example

predictor value 0: predicts branch not taken

predictor value 1: predicts branch taken

initial value in  
the branch  
predictor given  
(can be either  
0 or 1)

ADD X0, XZR, XZR

Loop: ADDI X1, X1, #1

ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	0			
1				
2				
3				

# 1-bit predictor example

predictor value 0: predicts branch not taken

predictor value 1: predicts branch taken

ADD X0, XZR, XZR

Loop: ADDI X1, X1, #1

ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	0	Not taken		
1				
2				
3				

# 1-bit predictor example

predictor value 0: predicts branch not taken

predictor value 1: predicts branch taken

ADD X0, XZR, XZR

Loop: ADDI X1, X1, #1

ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	0	Not taken	1 (Taken)	yes
1				
2				
3				



# 1-bit predictor example

predictor value 0: predicts branch not taken

predictor value 1: predicts branch taken

initial value in  
the branch  
predictor given  
(can be either  
0 or 1)

Actual outcome of the  
branch used to update  
the predictor

ADD X0, XZR, XZR

Loop: ADDI X1, X1, #1

ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	0	Not taken	1 (Taken)	yes
1	1			
2				
3				

# 1-bit predictor example

predictor value 0: predicts branch not taken

predictor value 1: predicts branch taken

initial value in  
the branch  
predictor given  
(can be either  
0 or 1)

Actual outcome of the  
branch used to update  
the predictor

ADD X0, XZR, XZR

Loop: ADDI X1, X1, #1

ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	0	Not taken	1 (Taken)	yes
1	1			
2				
3				

# 1-bit predictor example

ADD X0, XZR, XZR


Loop: ADDI X1, X1, #1

ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	0	Not taken	1 (Taken)	yes
1	1	Taken	1 (Taken)	No
2	1			
3				



# 1-bit predictor example

ADD X0, XZR, XZR


Loop: ADDI X1, X1, #1

ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	0	Not taken	1 (Taken)	yes
1	1	Taken	1 (Taken)	No
2	1	Taken	1 (Taken)	No
3	1			



# 1-bit predictor example

ADD X0, XZR, XZR

Loop: ADDI X1, X1, #1

ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	0	Not taken	1 (Taken)	yes
1	1	Taken	1 (Taken)	No
2	1	Taken	1 (Taken)	No
...				
100	1	Taken	0 ( Not Taken)	Yes
	0			

Penalty 2  
cycles

# 1bit Branch prediction buffer (I)

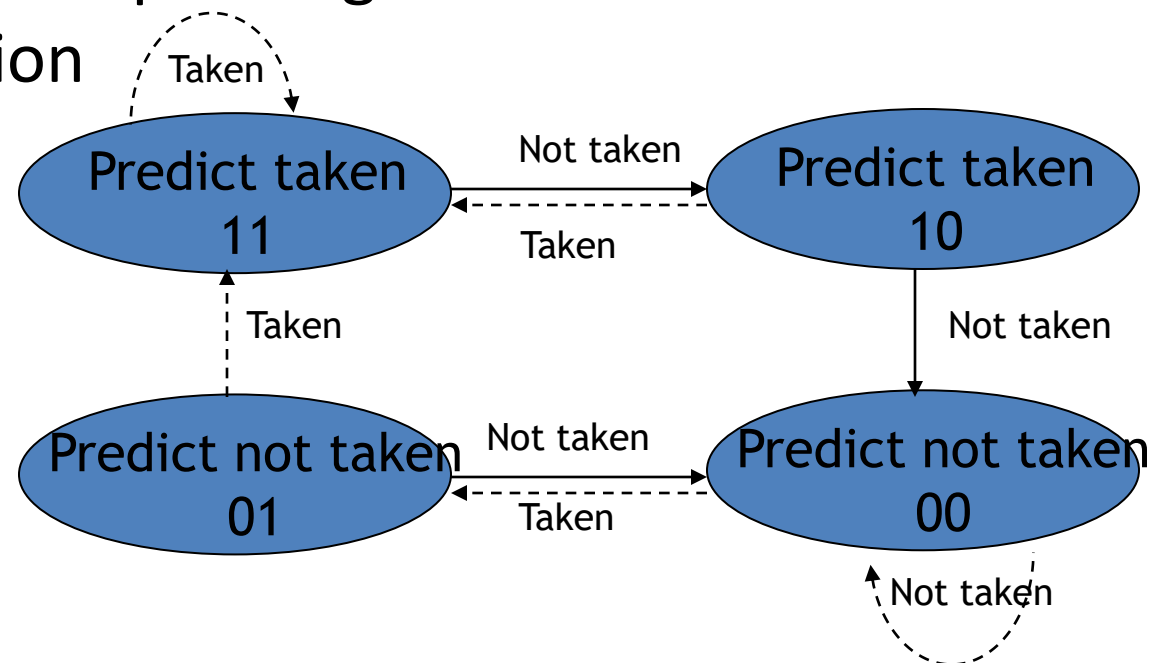
- Branch prediction buffer:
  - Small memory area indexed by the lower portion of the address of the branch instruction
  - Records whether the branch was taken the last time or not (1 bit is sufficient)

# 1bit Branch Prediction Buffer

- Limitations
  - Even for a regular loop the 1bit Branch Prediction Buffer will mispredict typically the first and the last iteration
    - 1<sup>st</sup> iteration: the bit has been set by the last iteration of the same loop to ‘not-taken’, but the branch will be taken
    - Last iteration: the bit says ‘taken’, but the branch won’t be taken
- Since you are using only a small part of the instruction address for identifying an entry in the branch prediction buffer, there is always the possibility that two branch instruction map to the same entry

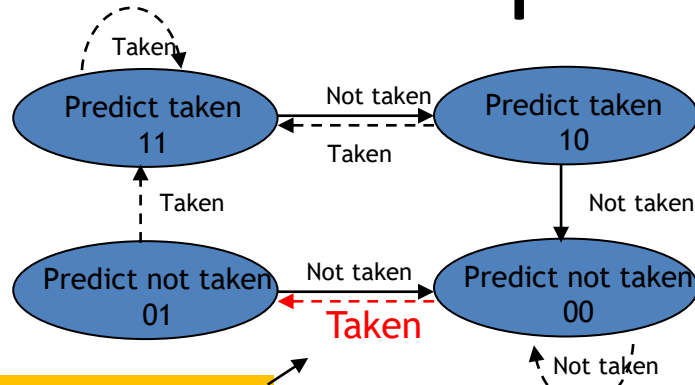
# 2bit Branch Prediction Buffer

- A prediction must miss twice before the prediction is changed
- Follow the State-Transition Diagram to update the predictor depending on the outcome of the branch instruction





# 2-bit predictor example



Update uses  
the State-  
Transition  
diagram

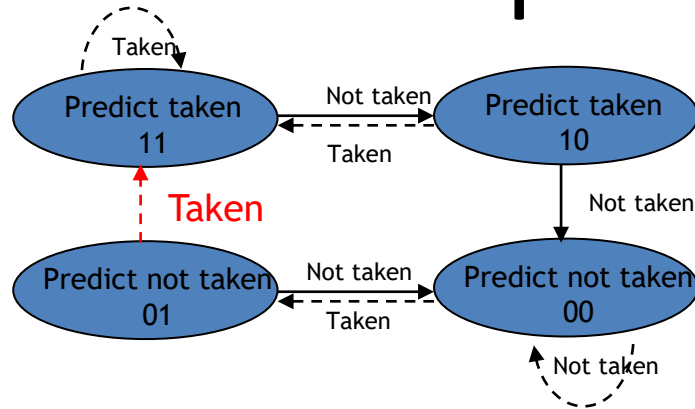
Actual outcome of the  
branch used to update  
the predictor

```

ADD X0, XZR, XZR
Loop: ADDI X1, X1, #1
      ADDI X0, X0, #1
      SUBI X2, X0, #100
      CBNZ X2, Loop
  
```

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	00	Not taken	1 (Taken)	yes
1	01			
2				
3				

# 2-bit predictor example



ADD X0, XZR, XZR

Loop: ADDI X1, X1, #1

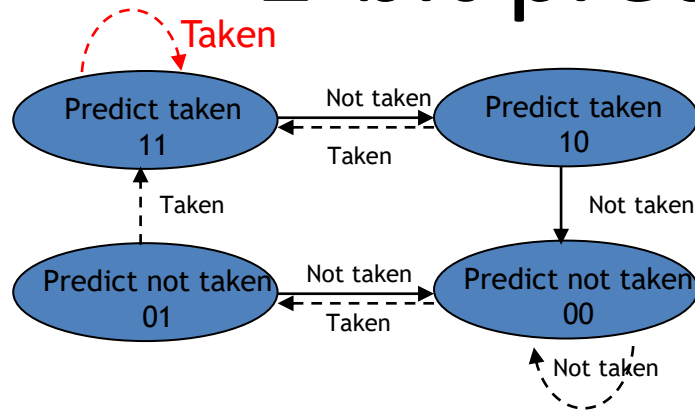
ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	00	Not taken	1 (Taken)	yes
1	01	Not taken	1 (Taken)	Yes
2	11			
3				

# 2-bit predictor example

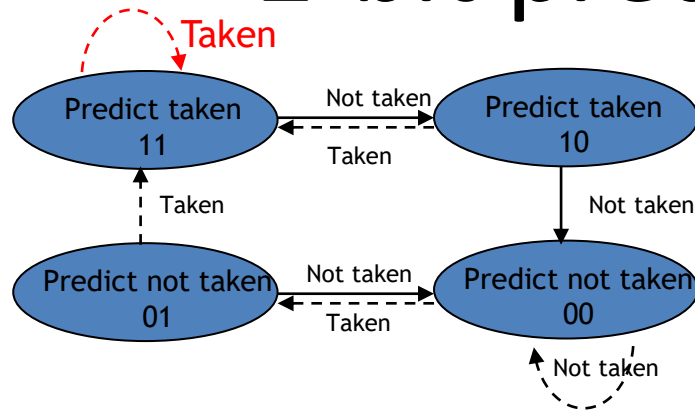


```

ADD X0, XZR, XZR
Loop: ADDI X1, X1, #1
      ADDI X0, X0, #1
      SUBI X2, X0, #100
      CBNZ X2, Loop
  
```

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	00	Not taken	1 (Taken)	yes
1	01	Not taken	1 (Taken)	Yes
2	11	Taken	1 (Taken)	No
3	11			

# 2-bit predictor example

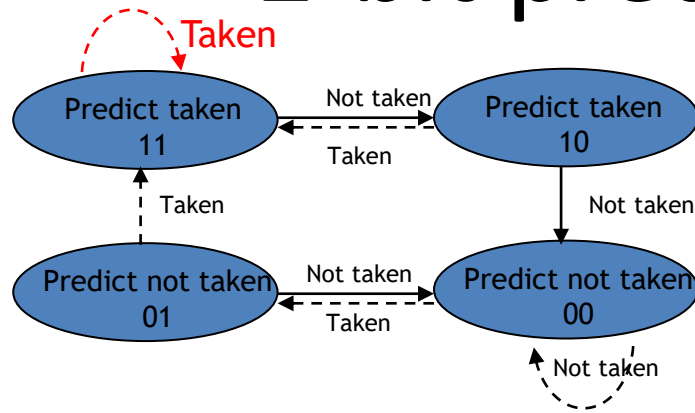


```

ADD X0, XZR, XZR
Loop: ADDI X1, X1, #1
      ADDI X0, X0, #1
      SUBI X2, X0, #100
      CBNZ X2, Loop
  
```

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	00	Not taken	1 (Taken)	yes
1	01	Not taken	1 (Taken)	Yes
2	11	Taken	1 (Taken)	No
3	11			
...				
100	11	Taken	0 (Not Taken)	yes

# 2-bit predictor example



ADD X0, XZR, XZR

Loop: ADDI X1, X1, #1

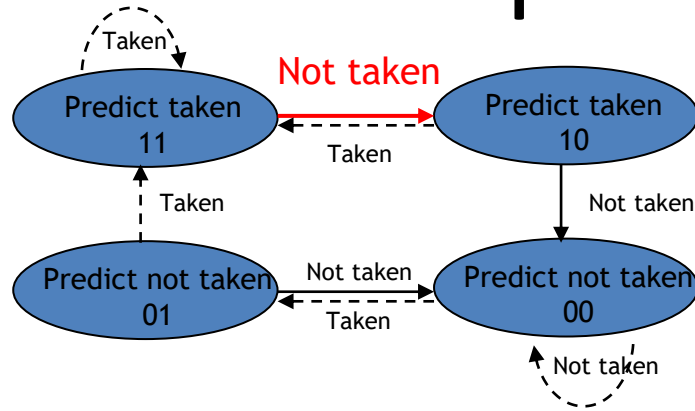
ADDI X0, X0, #1

SUBI X2, X0, #100

CBNZ X2, Loop

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	00	Not taken	1 (Taken)	yes
1	01	Not taken	1 (Taken)	Yes
2	11	Taken	1 (Taken)	No
3	11	Taken	1 (Taken)	No
...				
100	11	Taken	0 (Not Taken)	yes

# 2-bit predictor example



```

ADD X0, XZR, XZR
Loop: ADDI X1, X1, #1
      ADDI X0, X0, #1
      SUBI X2, X0, #100
      CBNZ X2, Loop
  
```

Value of i or X0	Branch predictor for Branch b1	Prediction (Taken/Not taken)	Actual outcome of branch b1	Misprediction? (Yes/No)
0	00	Not taken	1 (Taken)	yes
1	01	Not taken	1 (Taken)	Yes
2	11	Taken	1 (Taken)	No
3	11	Taken	1 (Taken)	No
...				
100	11	Taken	0 (Not Taken)	yes
	10	Taken		

# 1-bit vs. 2-bit predictor

- 2 bit predictor requires two mis-predictions to change the actual prediction
- 2-bit predictor looks worse in the trivial example here than the 1-bit predictor
- Has significant advantages for nested loops and many realistic scenarios