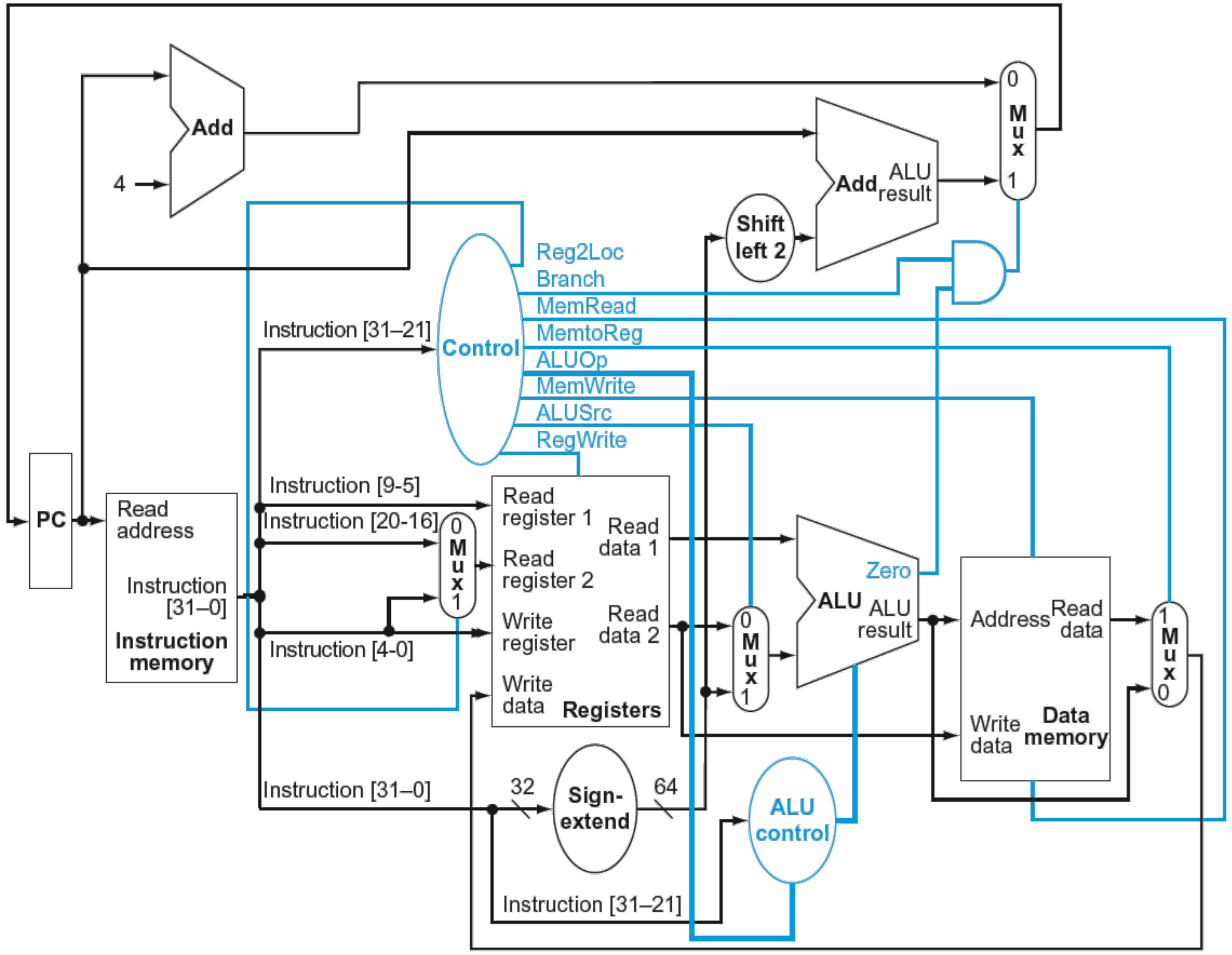
Computer Organization and Architecture

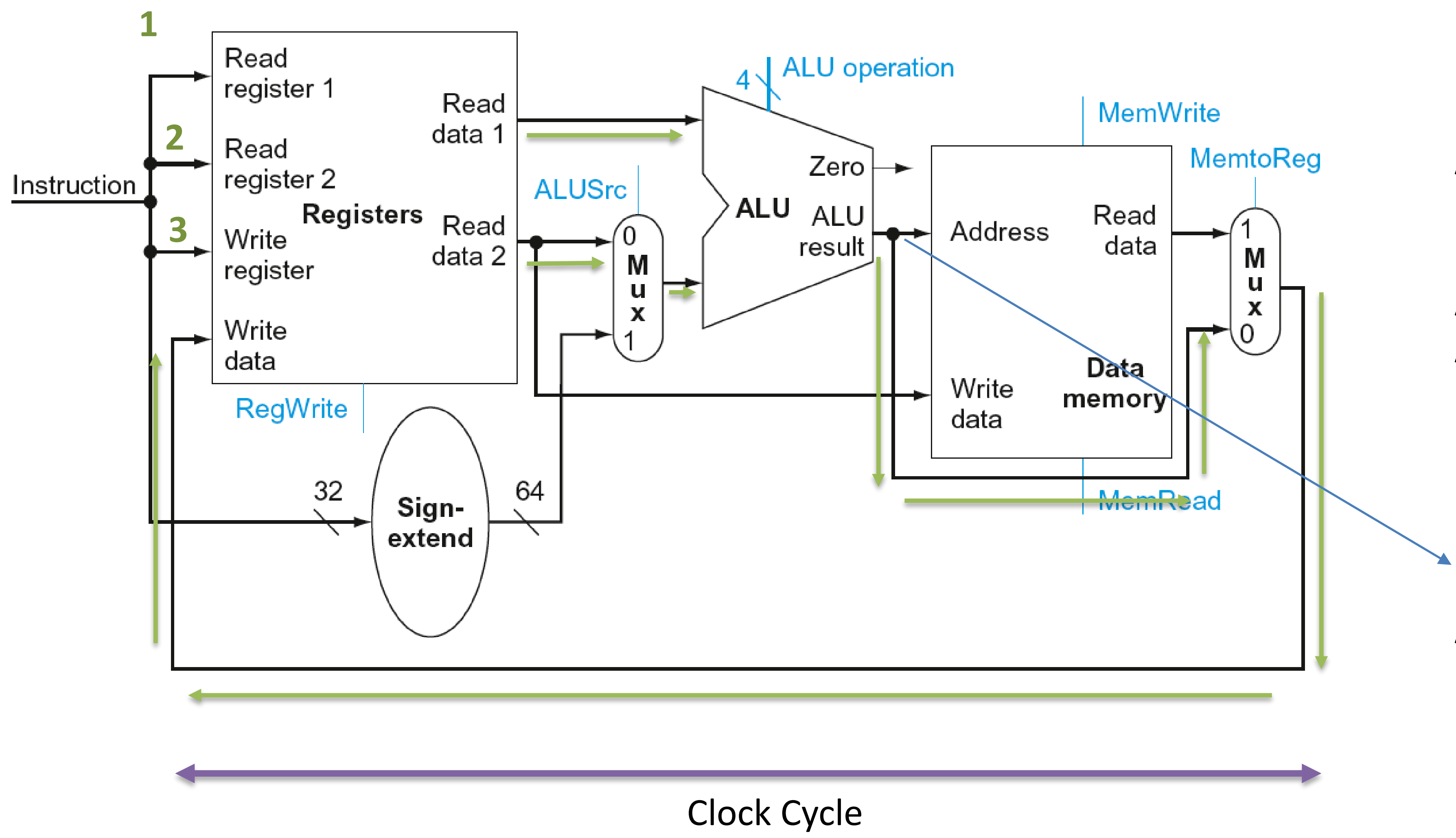
Lecture – 19

Oct 24th, 2022

Datapath With Control



Single Cycle Implementation

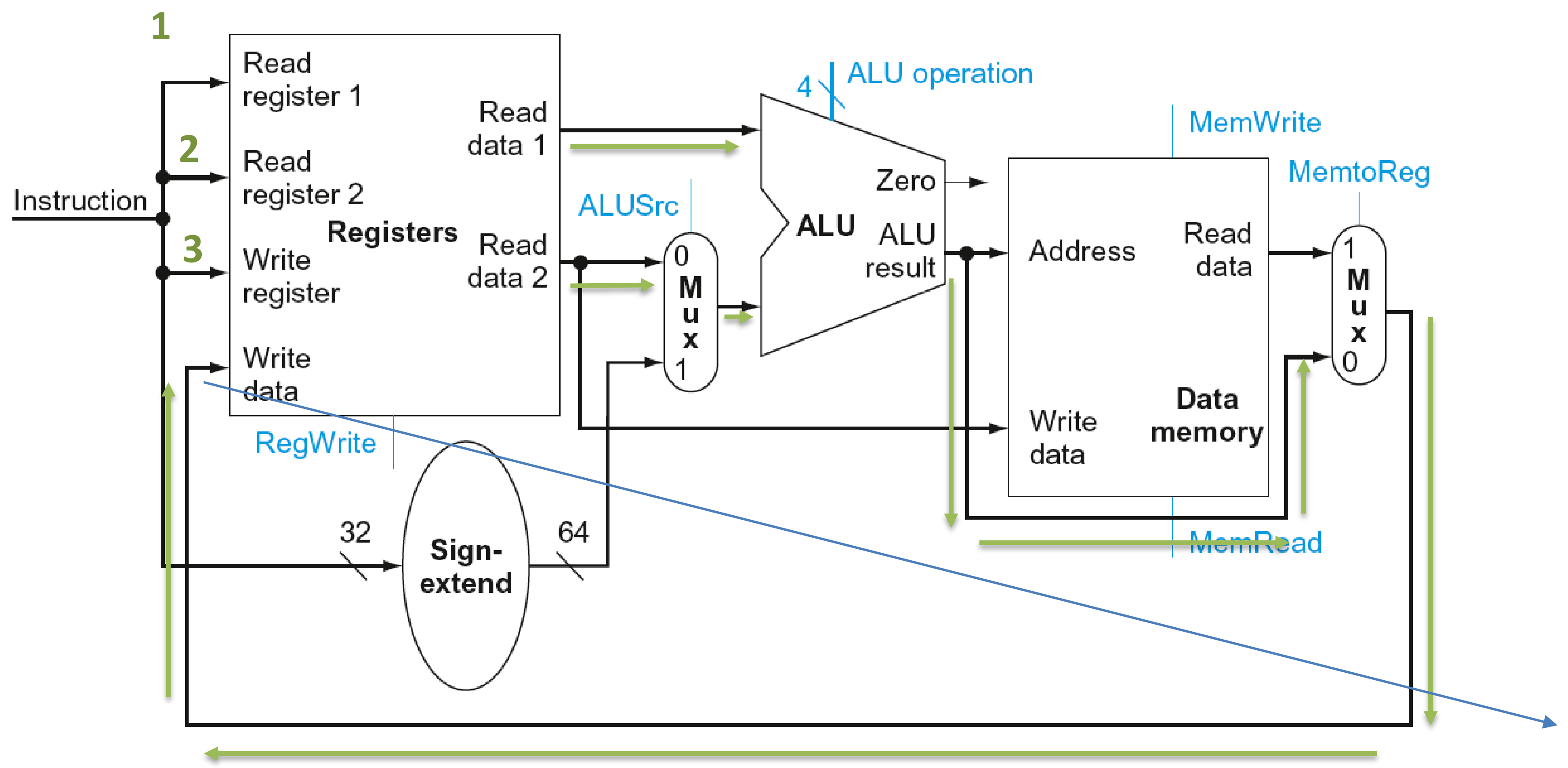


R-type
ADD X3, X1, X2

RegWrite ➔ 1
ALUSrc ➔ 0
ALU operation ➔ 0010
MemWrite ➔ 0
MemRead ➔ 0
MemtoReg ➔ 0

1. Data is not available until after ALU

Single Cycle Implementation

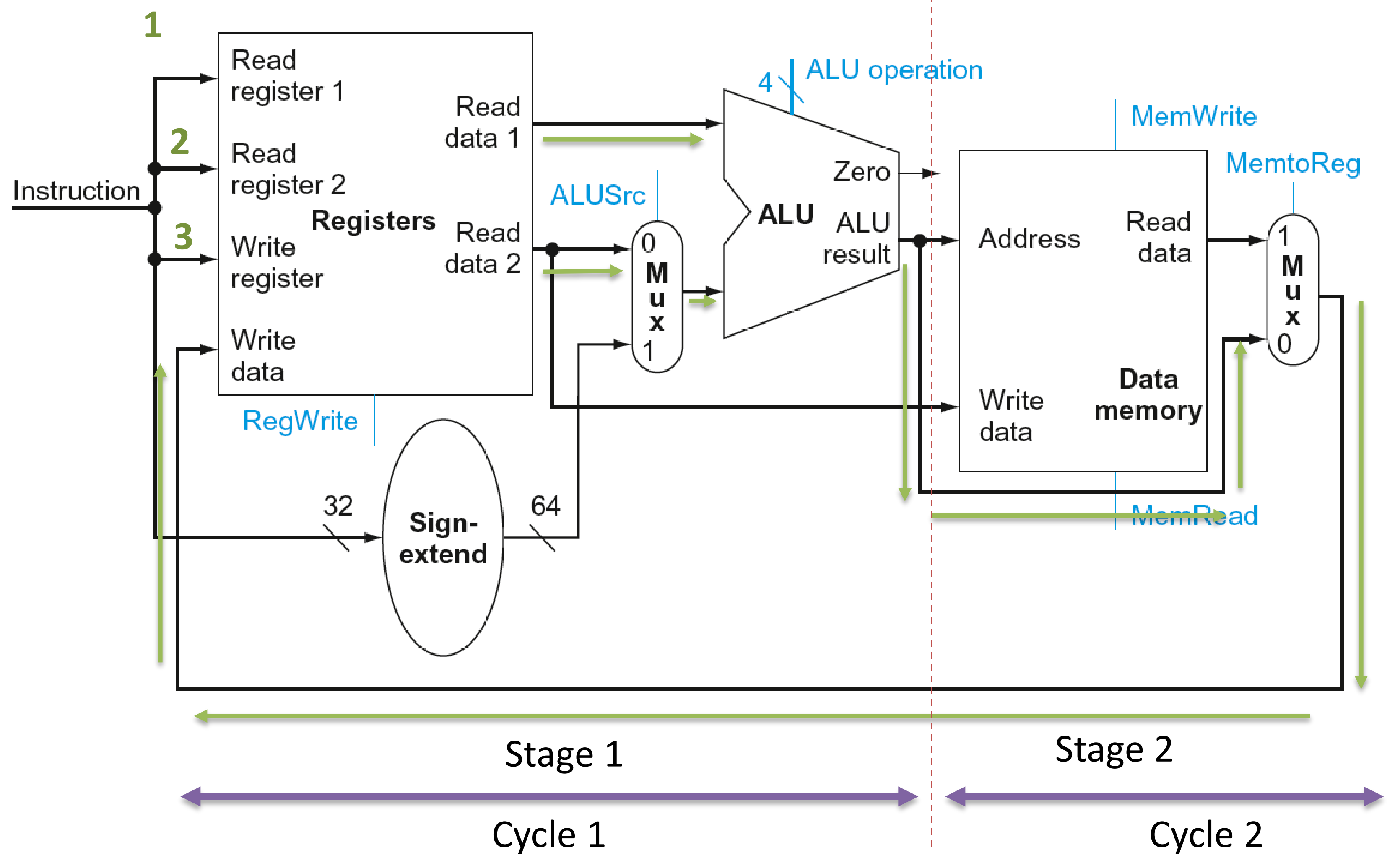


R-type
ADD X3, X1, X2

RegWrite → 1
ALUSrc → 0
ALU operation → 0010
MemWrite → 0
MemRead → 0
MemtoReg → 0

1. Data is not available until after ALU
2. Old values are written

Multi-Cycle Implementation



R-type
ADD X3, X1, X2

Break into **stages**
Execute in two clock cycles.

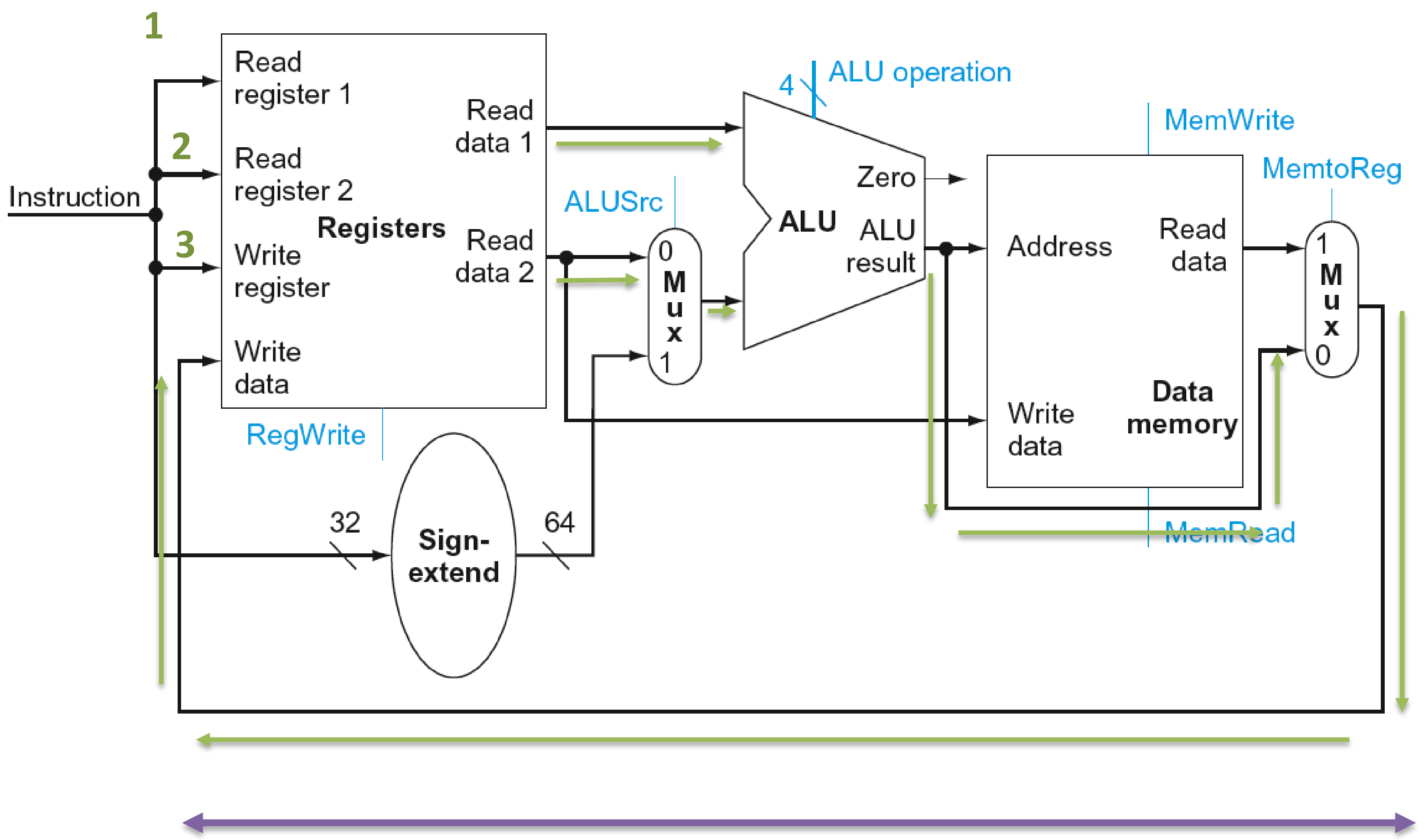
Stage 1 (cycle 1):
RegWrite ➔ 0
ALUSrc ➔ 0
ALU operation ➔ 0010

Stage 2 (cycle 2):
RegWrite ➔ 1
MemWrite ➔ 0
MemRead ➔ 0
MemtoReg ➔ 0

Impact on Performance.

- Executing in stage can reduce clock cycle time.
- But can affect or reduce overall performance.

Single Cycle Implementation

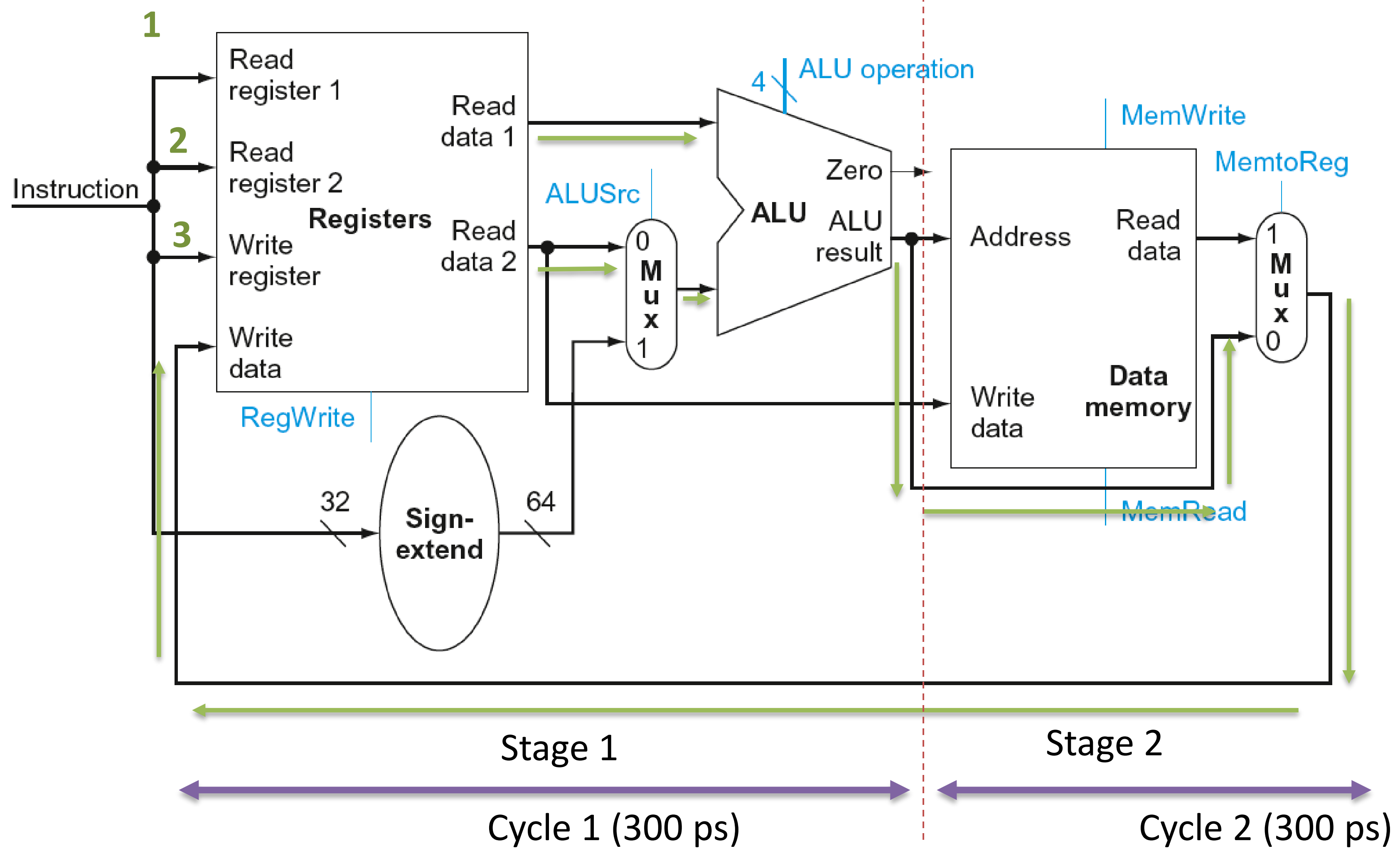


Cycle time (500 ps)

R-type
ADD X3, X1, X2

Let's say ADD takes a total of 500 ps.
If add is the instruction that takes the longest time.
Then my cycle time is 500ps

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 clocl 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

Instruction	300	600	900	1200	1500	1800
1	STAGE 1	STAGE 2				
2		STAGE 1	STAGE 2			
3			STAGE 1	STAGE 2		
4				STAGE 1	STAGE 2	
5					STAGE 1	STAGE 2

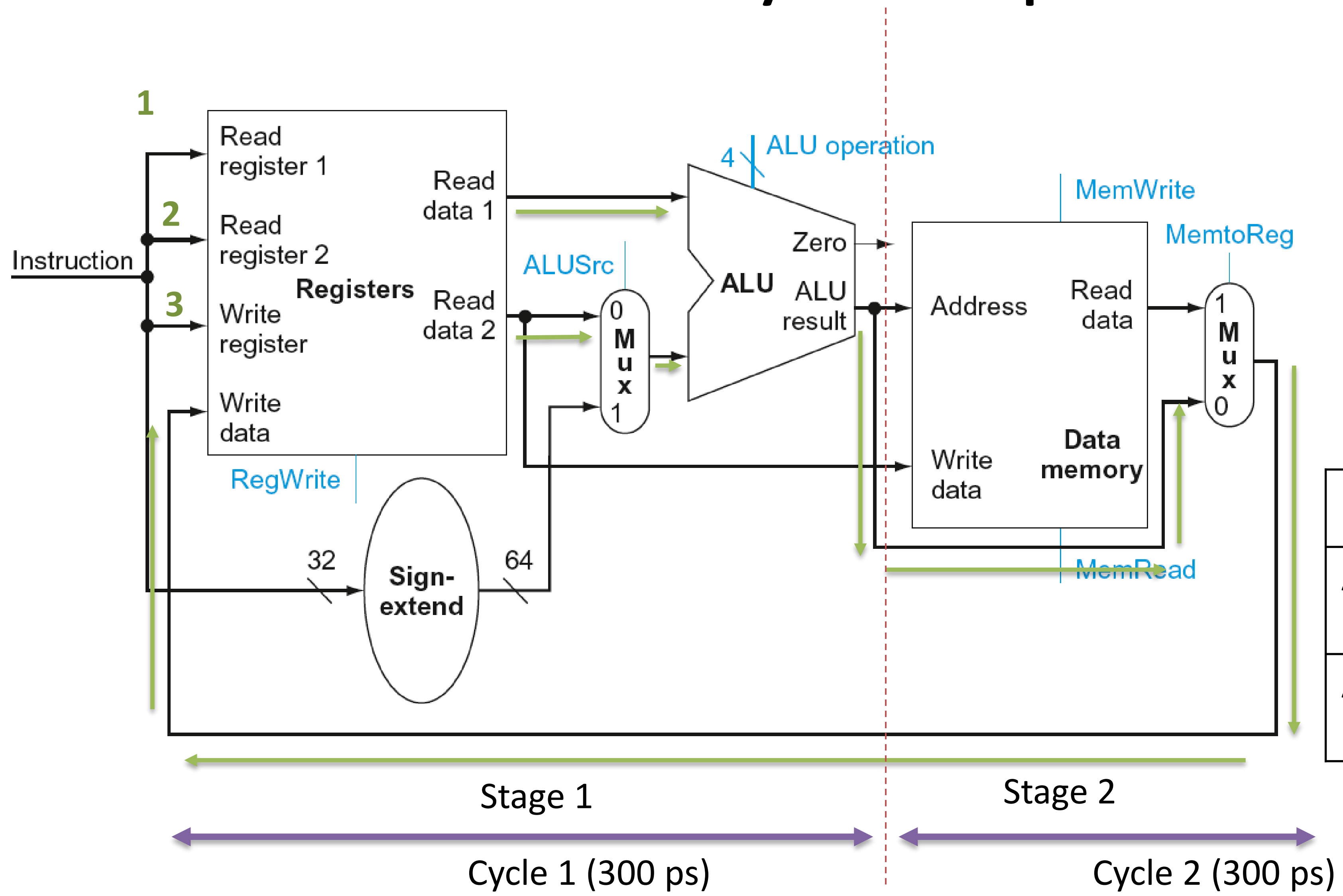
Time to execute 5 Add instructions

Single cycle implementation = 5 * 500 = 2500 ps

Multi cycle implementation = 5 * 600 = 3000 ps

Pipelining = 1800 ps

Multi-Cycle Implementation



R-type
ADD X3, X1, X2

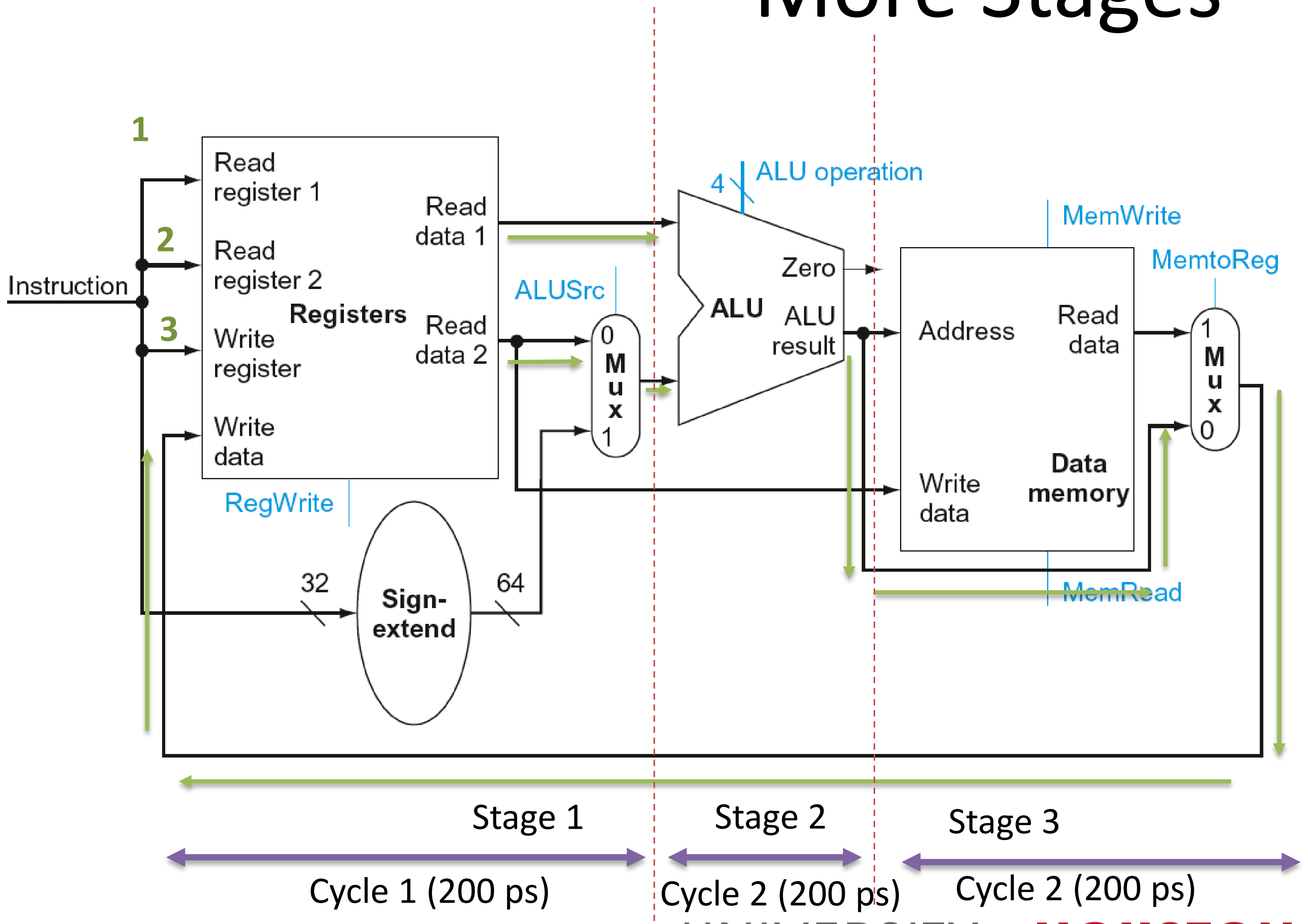
Not possible to implement with ADD

Value of write register

	1	2	3
ADD **X3**, X1, X2	S1: 3	S2: ?	
ADD **X4**, X3, X5		S1: 4	S2:

Need to update hardware.
More on this later

More Stages



R-type
ADD X3, X1, X2

Total 500 ps
Stage 1: 100 ps
Stage 2: 200 ps
Stage 3: 200 ps

More stages

Instruction	200	400	600	800	1000	1200	1400
1	Stage 1	Stage 2	Stage 3				
2		Stage 1	Stage 2	Stage 3			
3			Stage 1	Stage 2	Stage 3		
4				Stage 1	Stage 2	Stage 3	
5					Stage 1	Stage 2	Stage 3

Time to execute 5 Add instructions

Single cycle implementation = 5 * 500 = 2500 ps

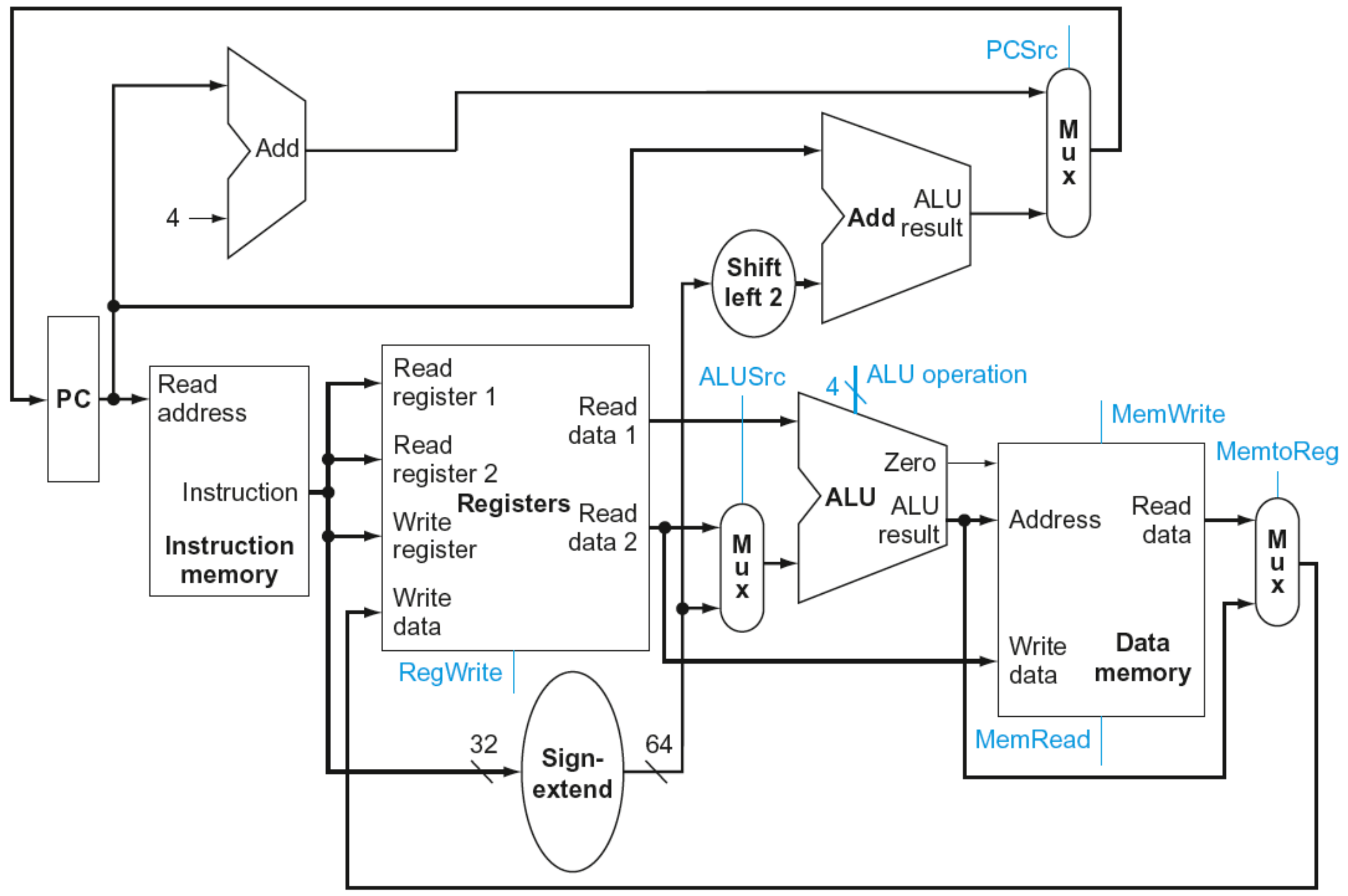
Multi cycle implementation = 5 * 600 = 3000 ps

Pipelining (2 stages) = 1800 ps

Pipelining (3 stages) = 1400 ps

Speed up is approximately equal to the number of stages.

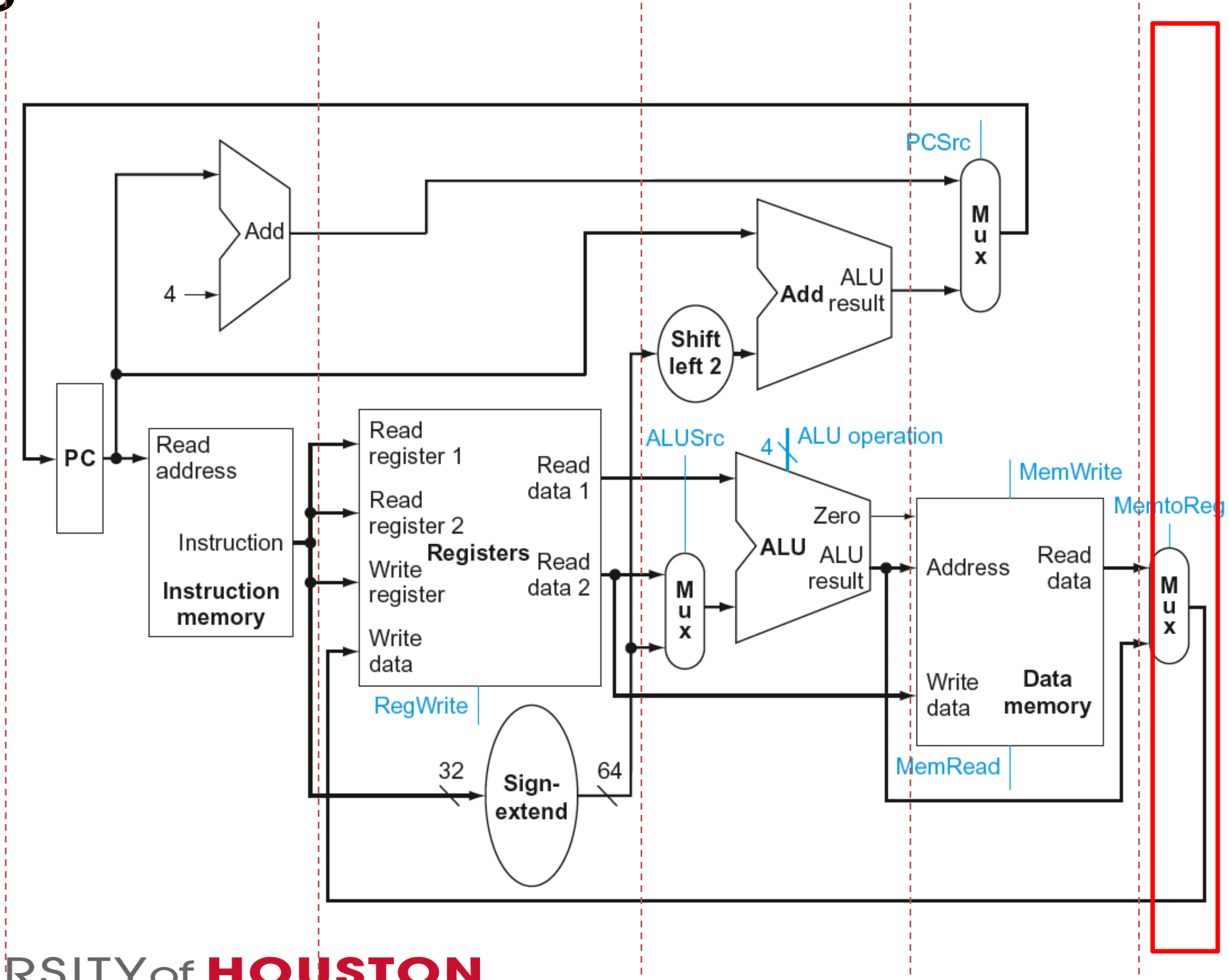
Stages in LEGv8



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

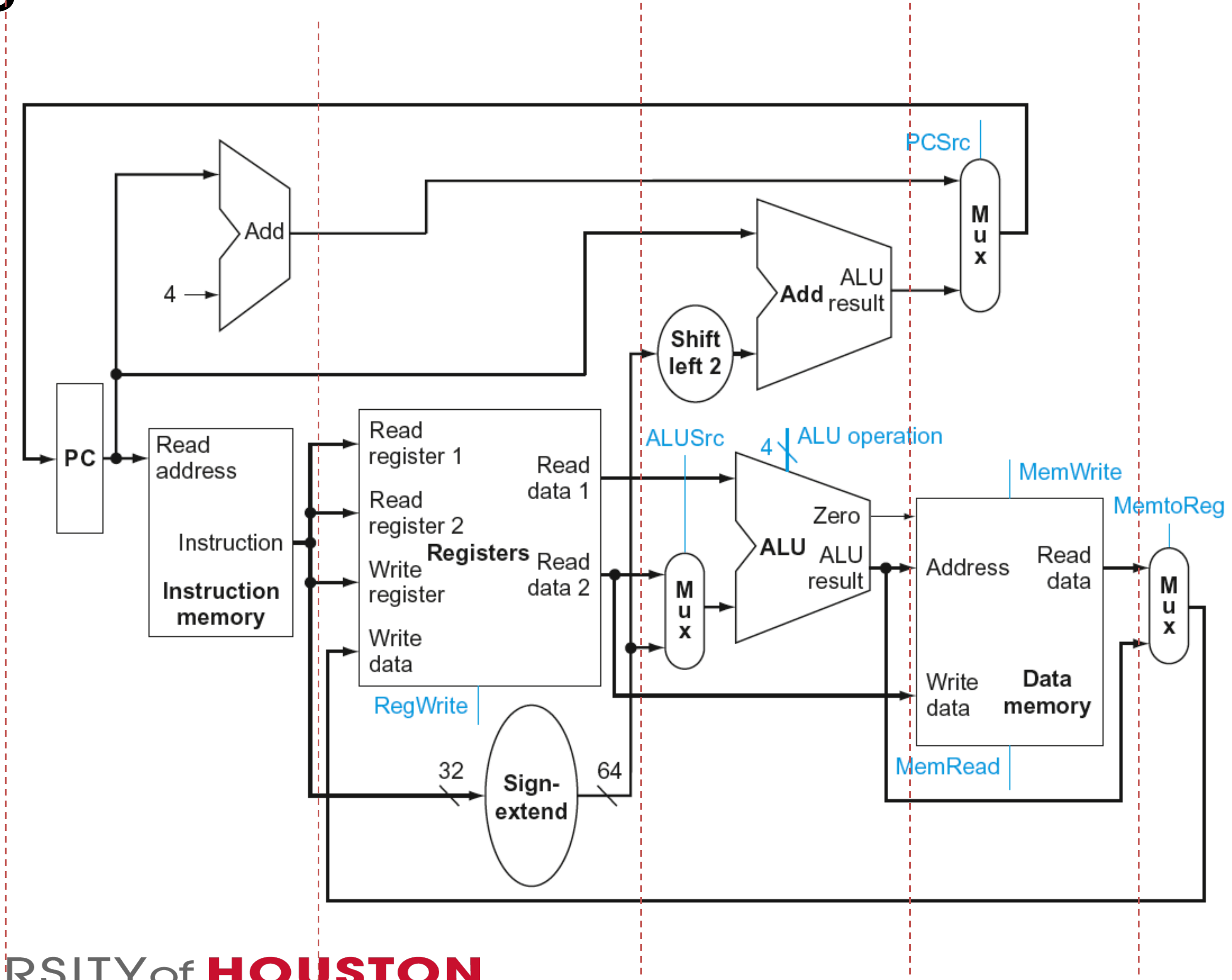


Stages in LEGv8

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					
LDUR					
CBZ					

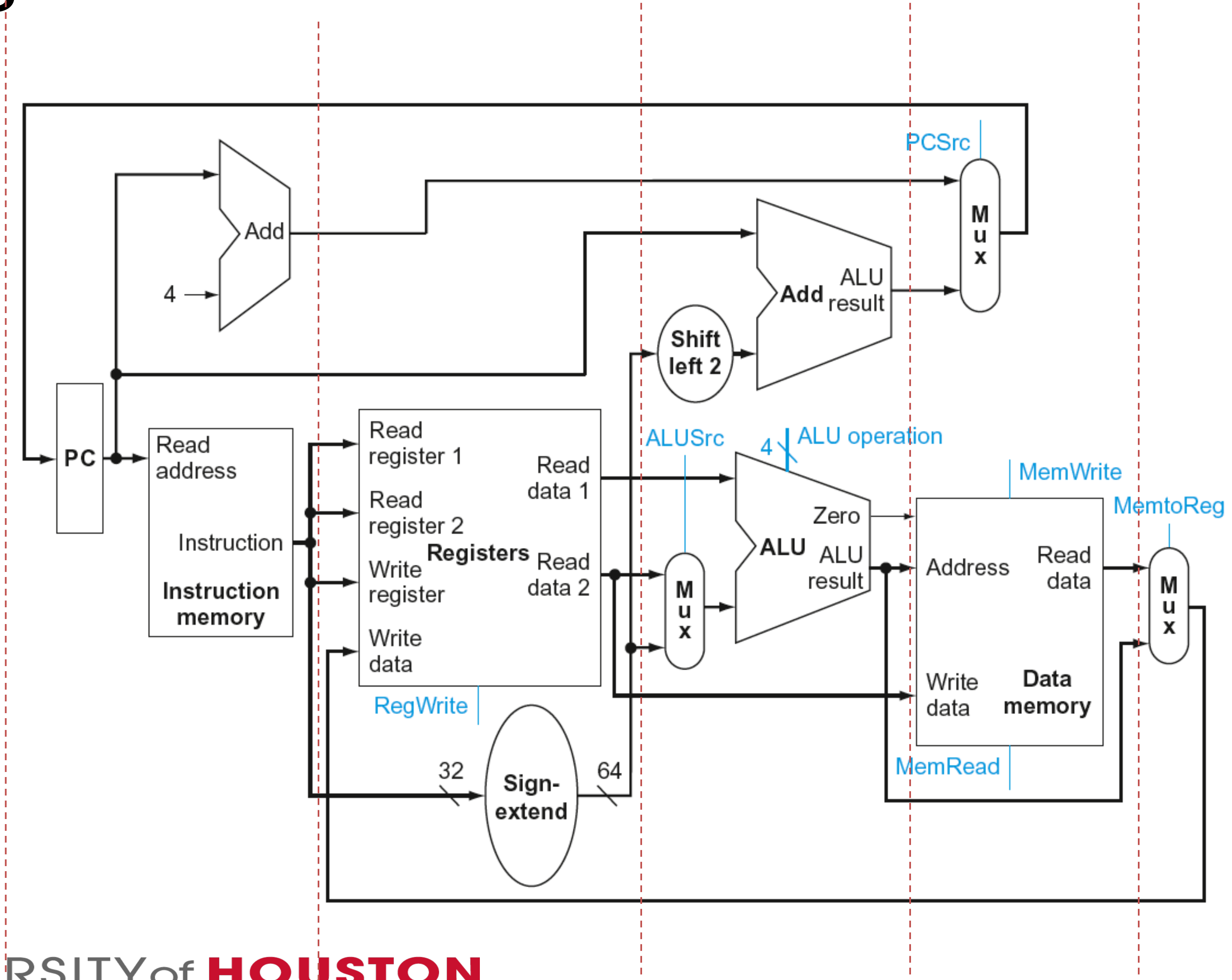


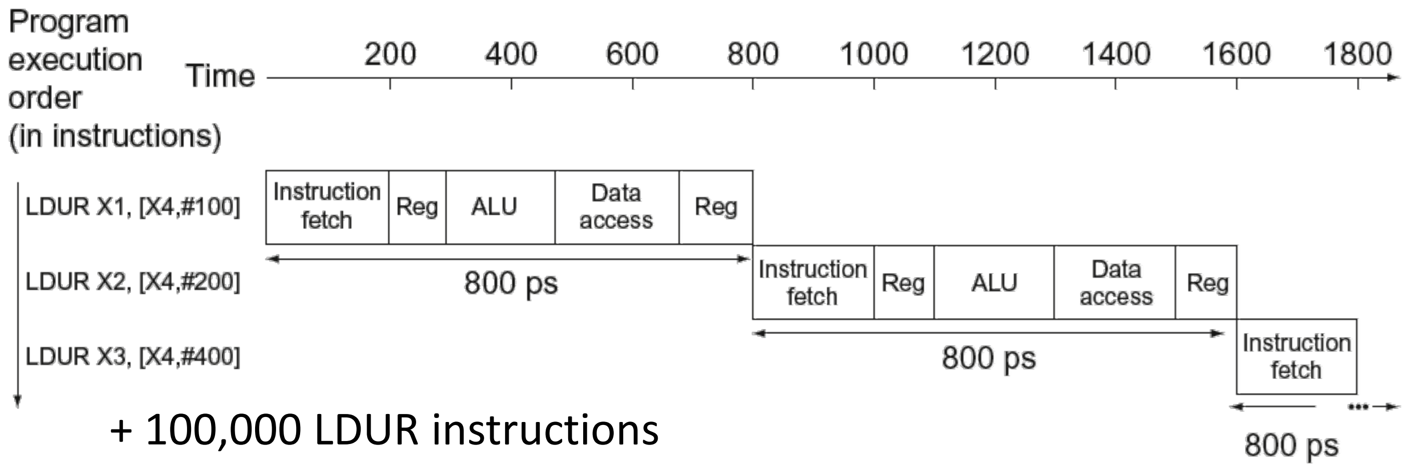
Stages in LEGv8

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

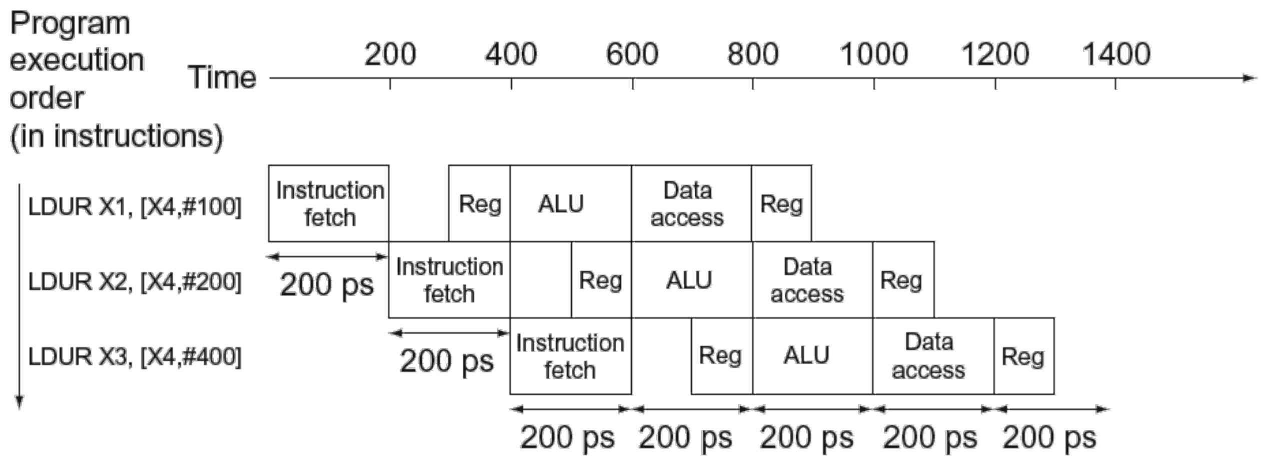
Inst./Stage	1	2	3	4	5
ADD(R format)	🟩	🟩	🟩		🟩
STUR	🟩	🟩	🟩	🟩	
LDUR	🟩	🟩	🟩	🟩	🟩
CBZ	🟩	🟩	🟩		





+ 100,000 LDUR instructions

Time = 100,003 * 800 =
800,002,400



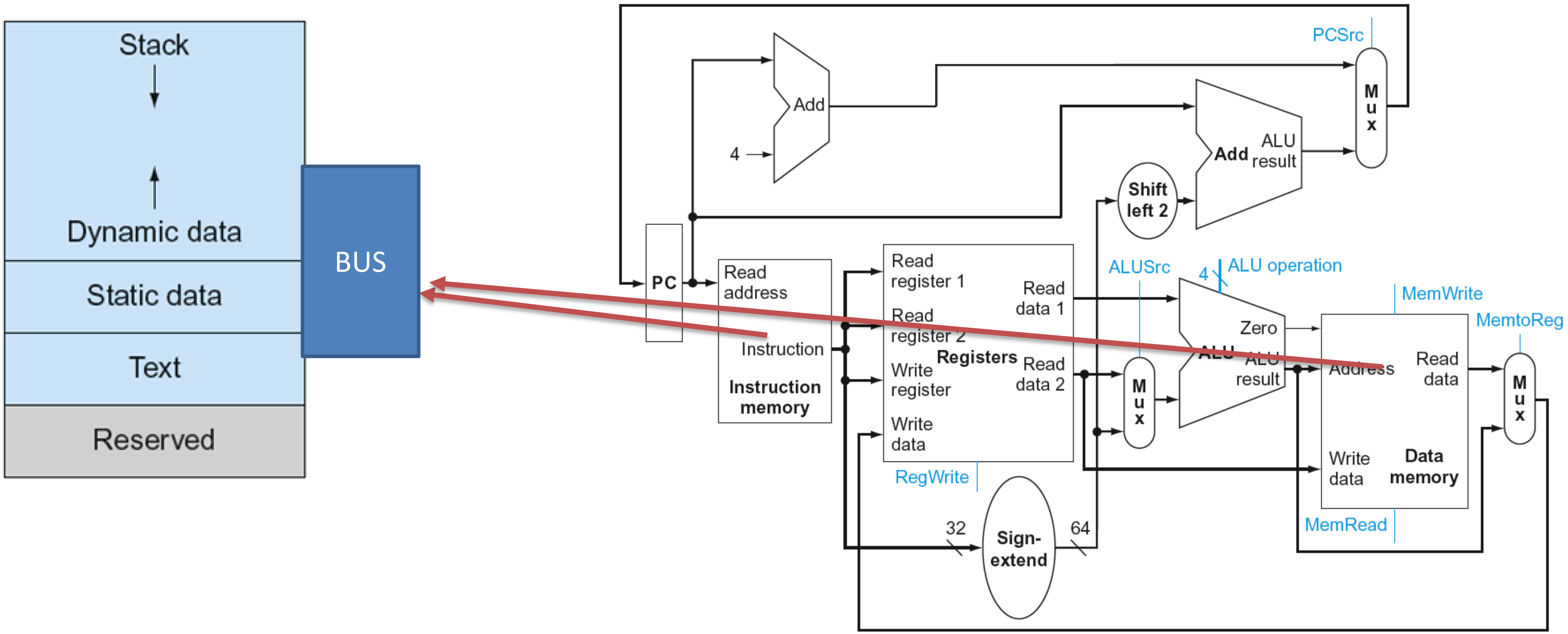
+ 100,000 LDUR instructions

Time = 1400 + 100,000 * 200
200,001,400

Speedup = $\frac{800{,}002{,}400}{200{,}001{,}400} \cong 4$

Can Pipeline cause Issues?

Single Memory Example



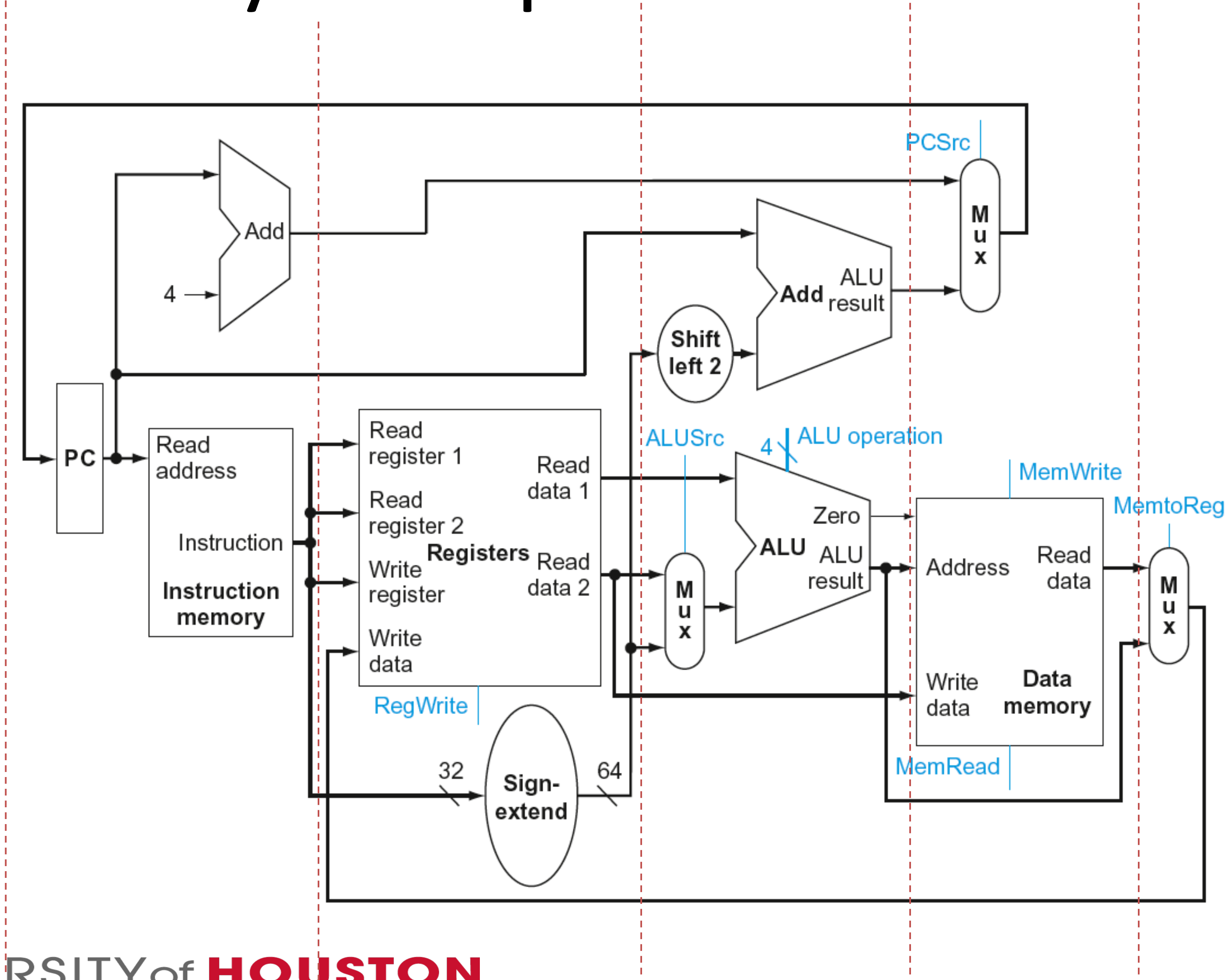
Single Memory Example

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./stages	1	2	3	4	5
LDUR	🟩	🟩	🟩	🟩	🟩

Inst./Stage	1	2	3	4	5
ADD(R format)	🟩	🟩	🟩		🟩

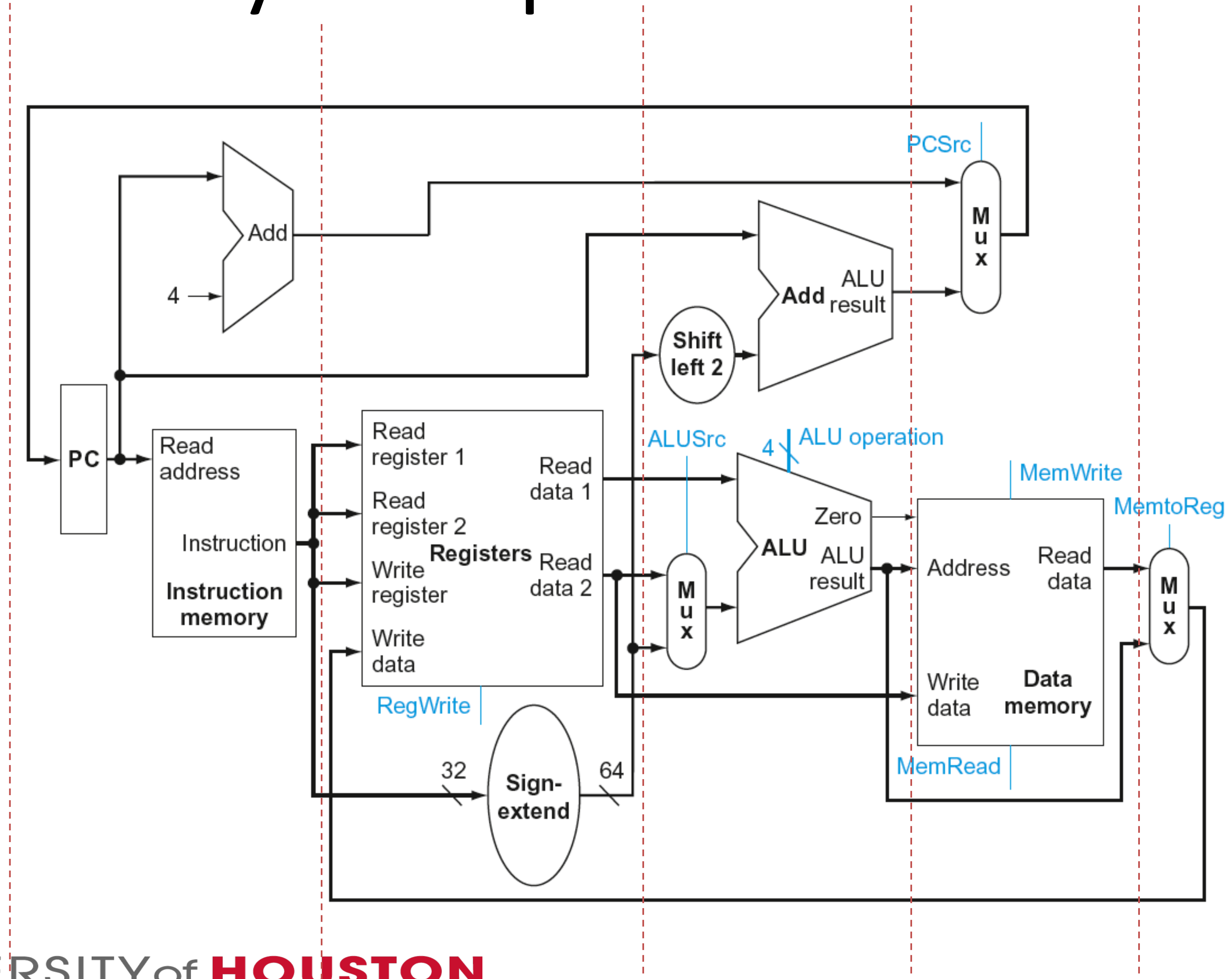


Single Memory Example

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

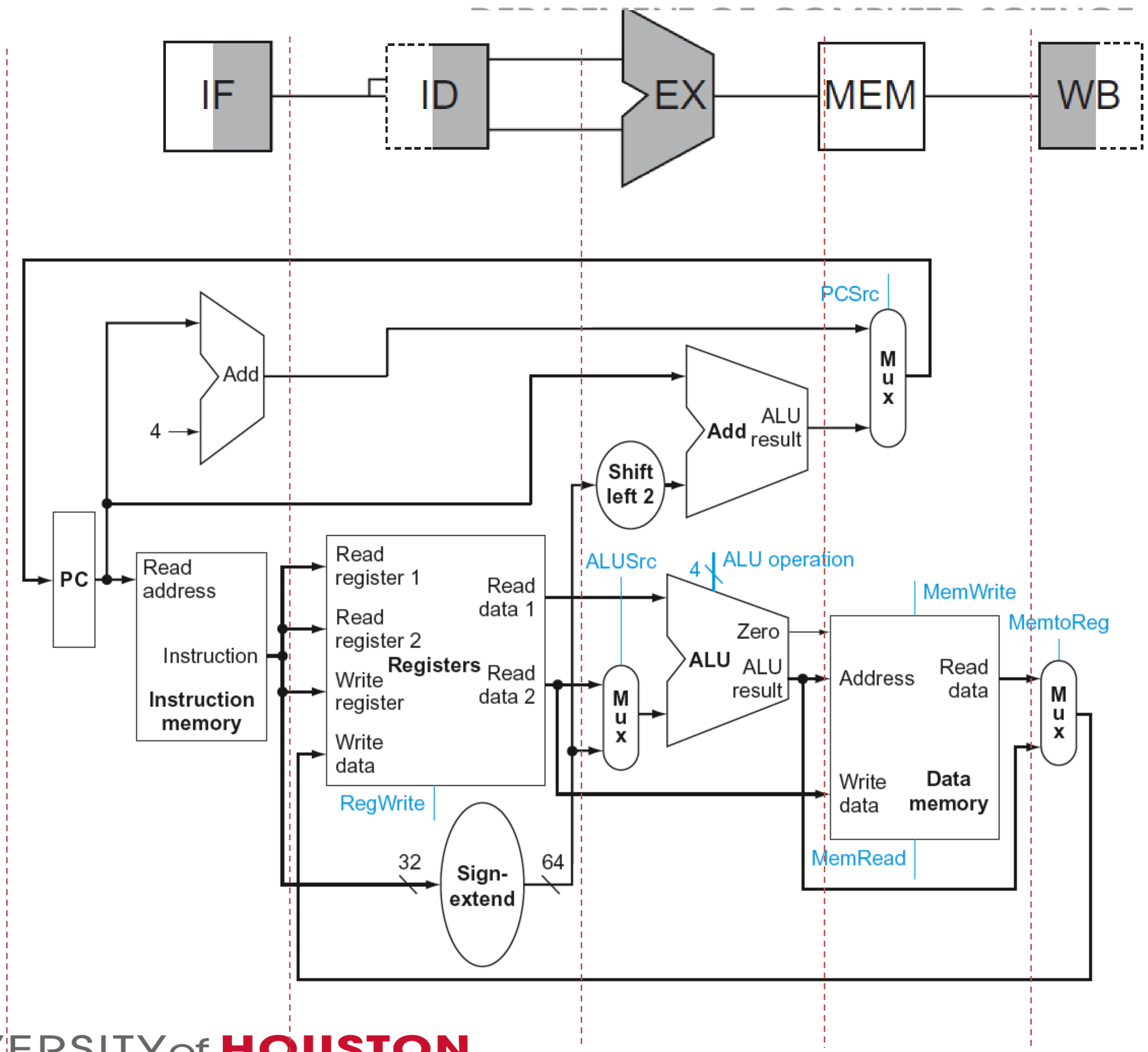


Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy

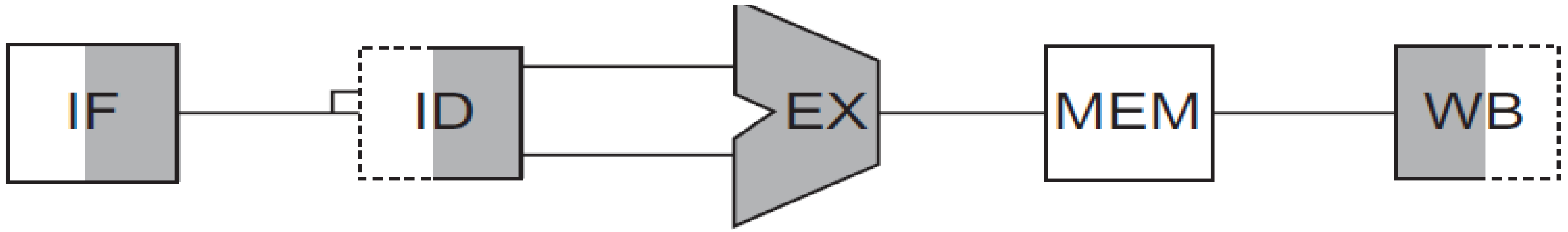
Five stages, one step per stage

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



Example R-Type Instrucstion

•



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.

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

Branches

CBZ X9, L2

Checks to see if the
value in register is 0
If yes, branches to L2

Register X9

L1: *ADD X10, X11, X12*

L2: *SUB X10, X11, X12*

Which instruction to fetch in the next cycle?

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

Structure Hazards

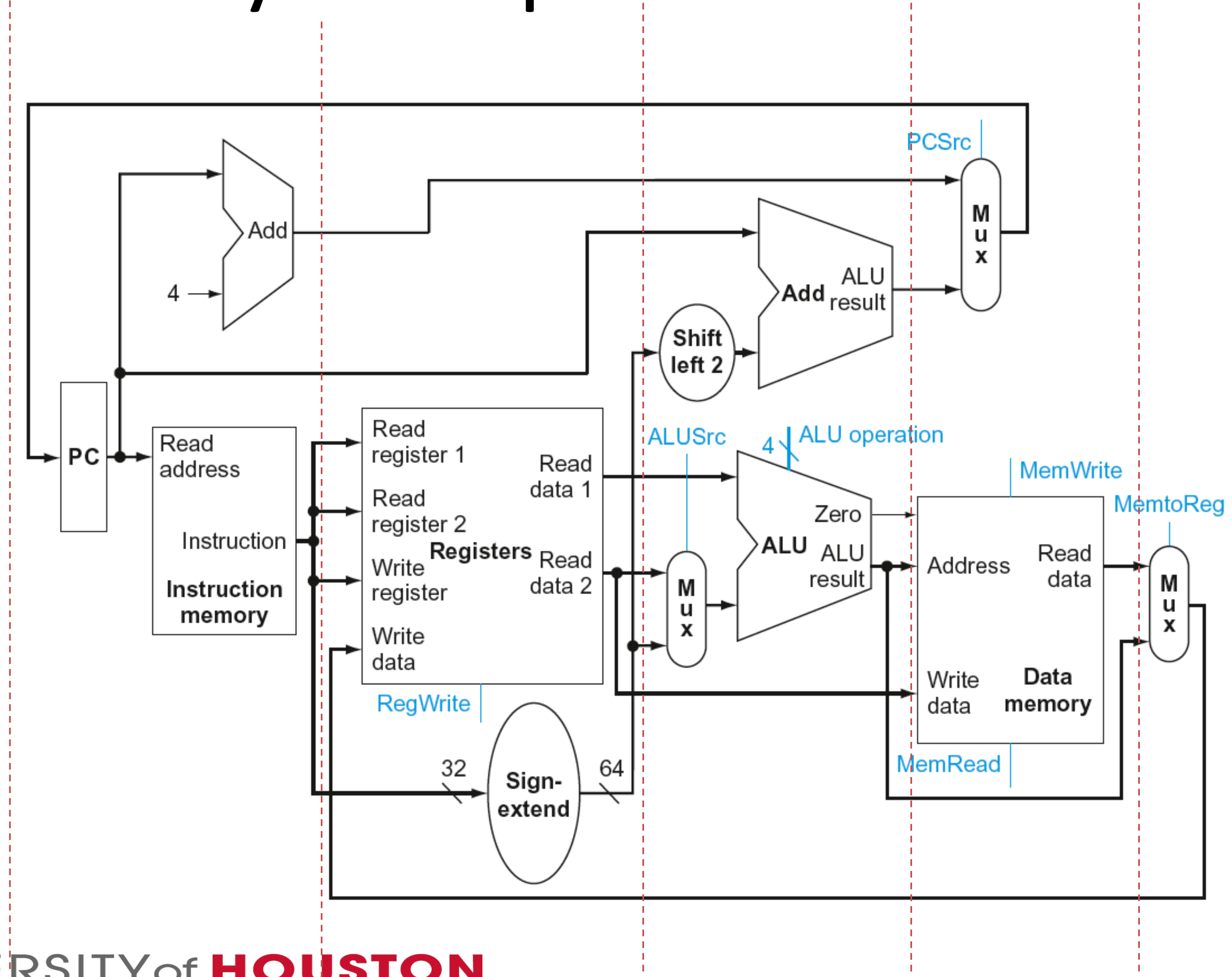
- Conflict for use of a resource
- In LEGv8 pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

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

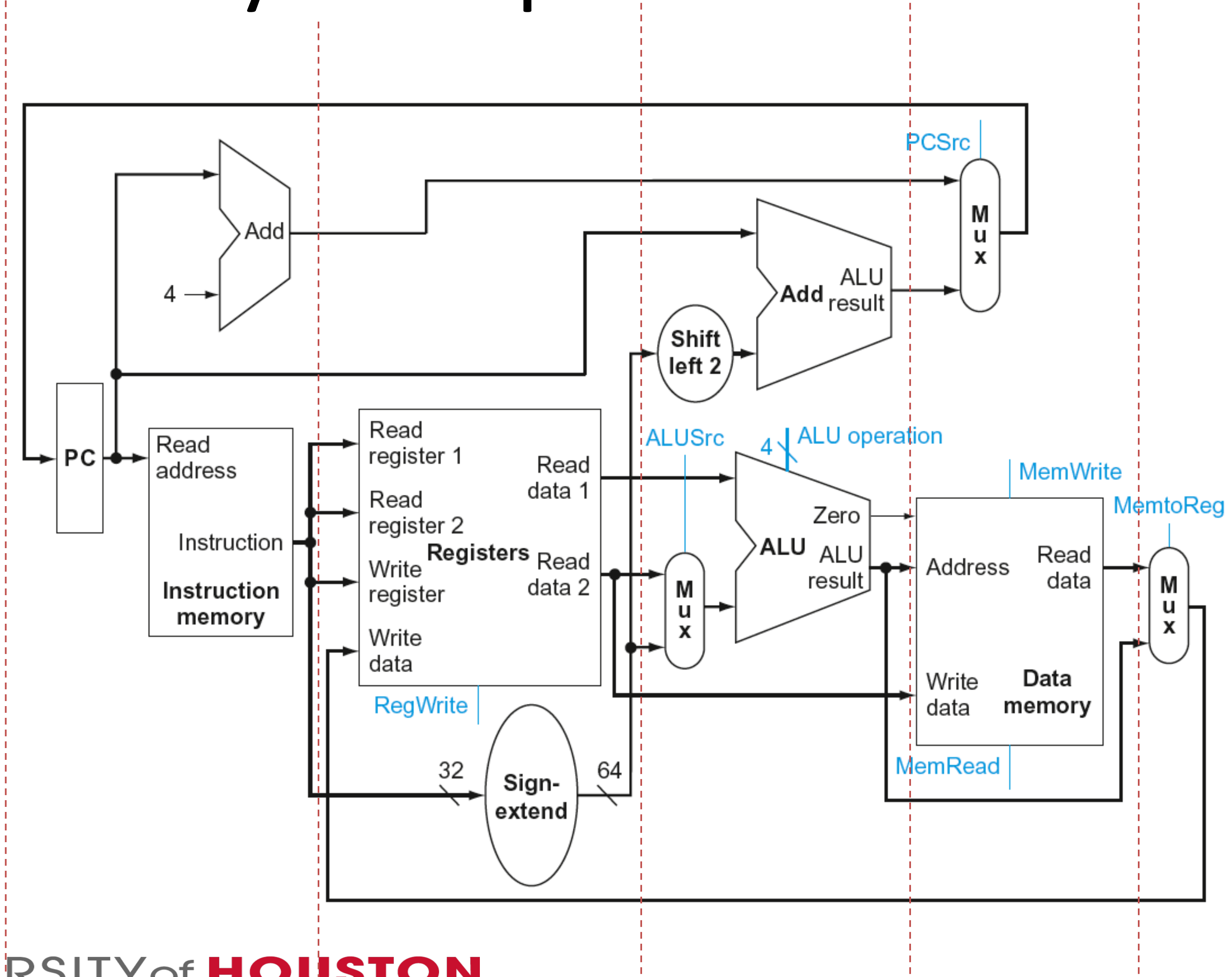


Single Memory Example

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
----	--	--	--	--	--	--
ADD					1	2

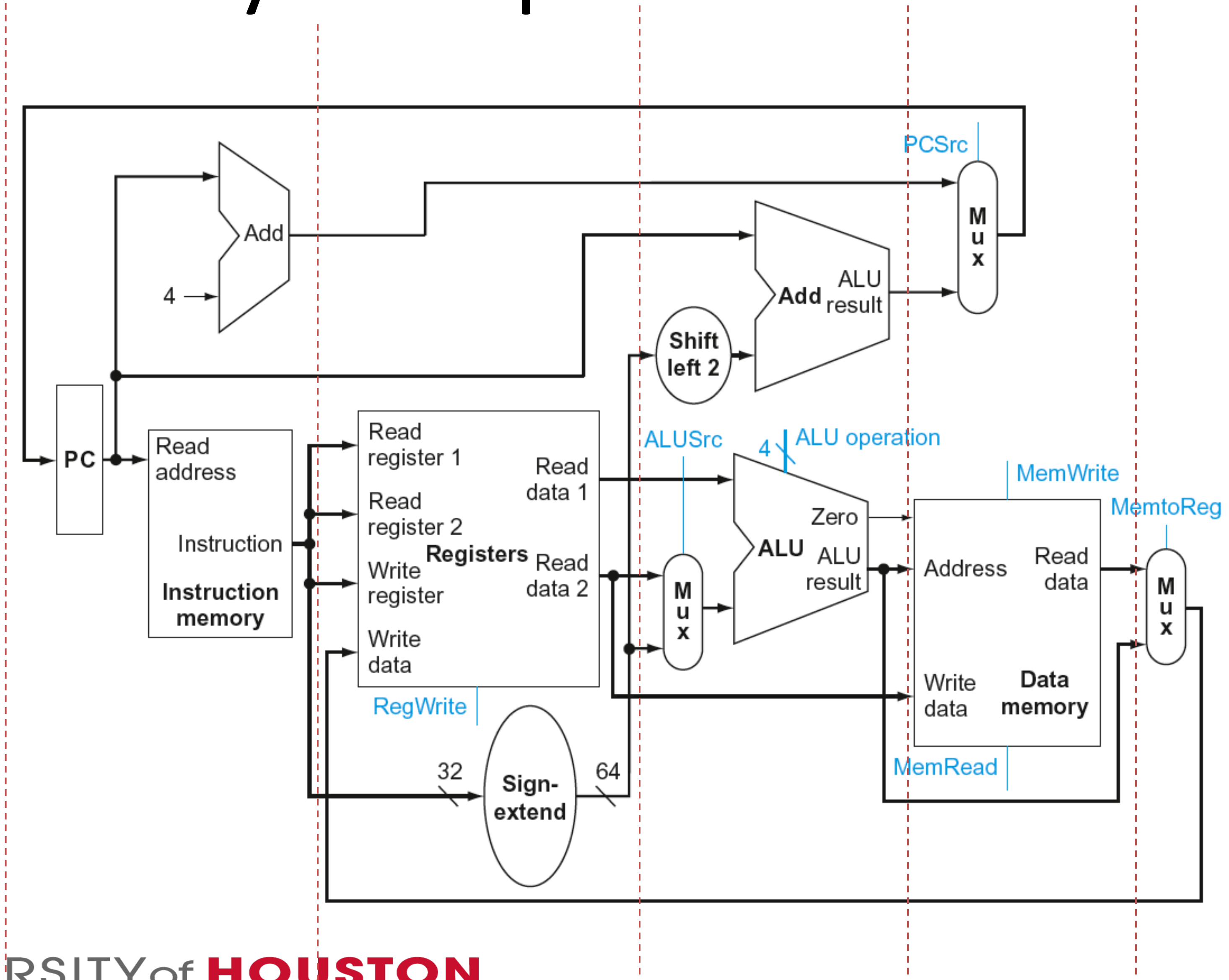


Single Memory Example

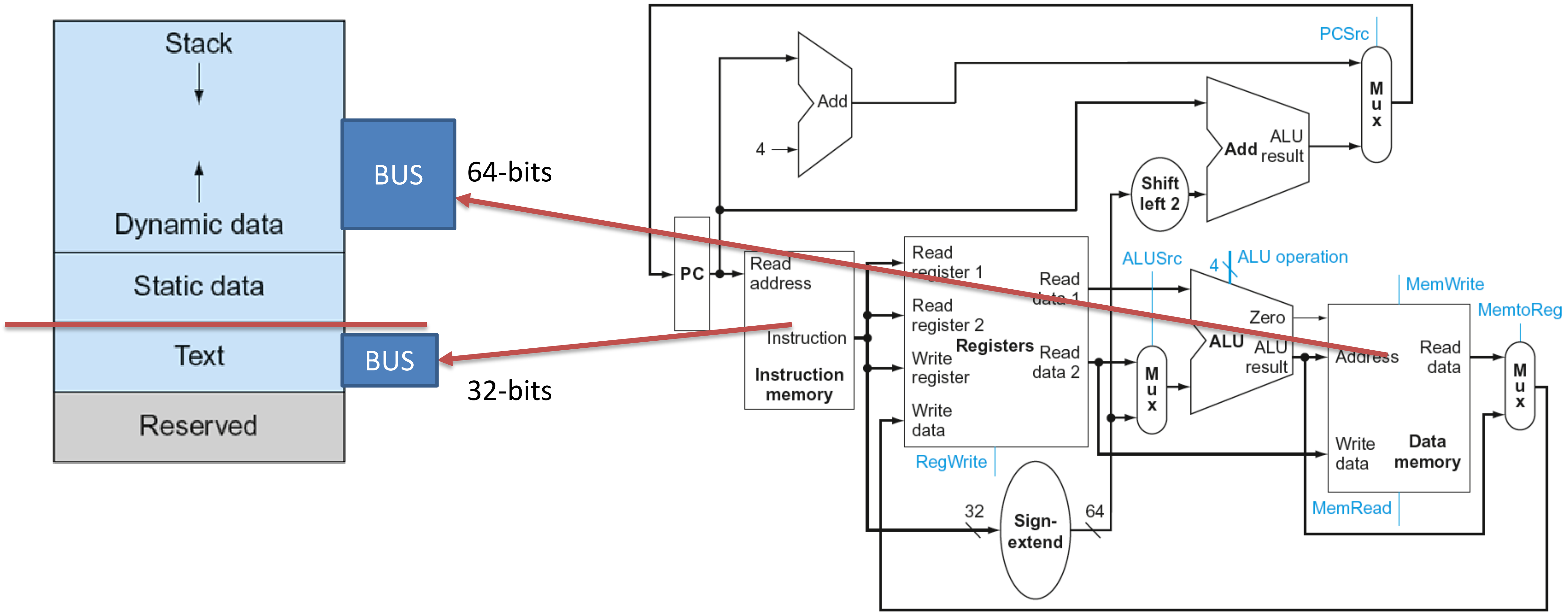
Five stages, one step per stage

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

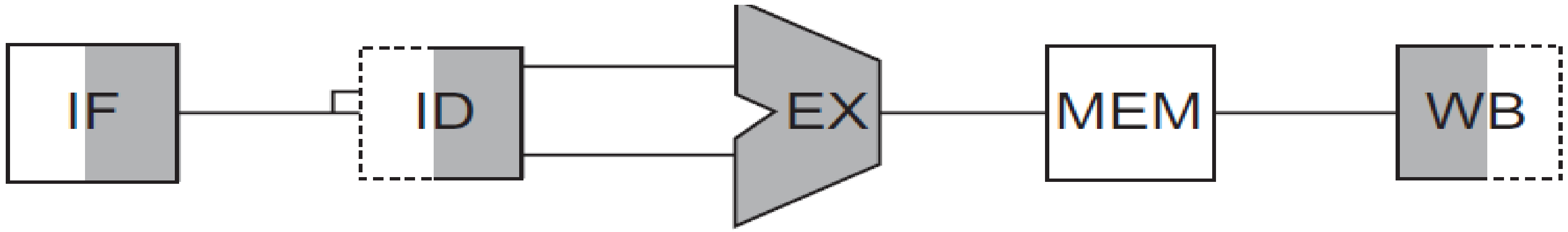
Data Hazards

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

  -
    ```
    ADD   X19, X0, X1
    SUB   X2, X19, X3
    ```

Example R-Type Instrucstion

•



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.

Data Hazards

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

  -
    ```
    ADD    X19, X0, X1     Write X19
    SUB    X2, X19, X3     Read X19
    ```

Three Generic Data Hazards

- True Data Dependency: also called Read-After-Write (RAW)
  Instr_J tries to read operand before Instr_I 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, Berkley
http://www.eecs.berkeley.edu/~culler/courses/cs252-s05

Three Generic Data Hazards

- Anti-dependency: also called Write-After-Read (WAR)
  Instr$_J$ writes operand ***before*** Instr$_I$ reads it

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

- Results from reuse of the name “X1”.

Three Generic Data Hazards

- Anti-dependency: also called Write-After-Read (WAR)
  Instr_J writes operand *before* Instr_I reads it

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

- Results from reuse of the name "X1".

Cycles	1	2	3	4	5	6
SUB X4,X1,X3	IF	ID: Read data from registers 1 and 3	EXE	Mem	WB	
ADD X1,X2,X3		IF	ID	EXE	MEM	WB: Value of X1 is written

Three Generic Data Hazards

- Anti-dependency: also called Write-After-Read (WAR)
  Instr$_J$ writes operand *before* Instr$_I$ 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$_J$ writes operand *before* Instr$_I$ writes it.

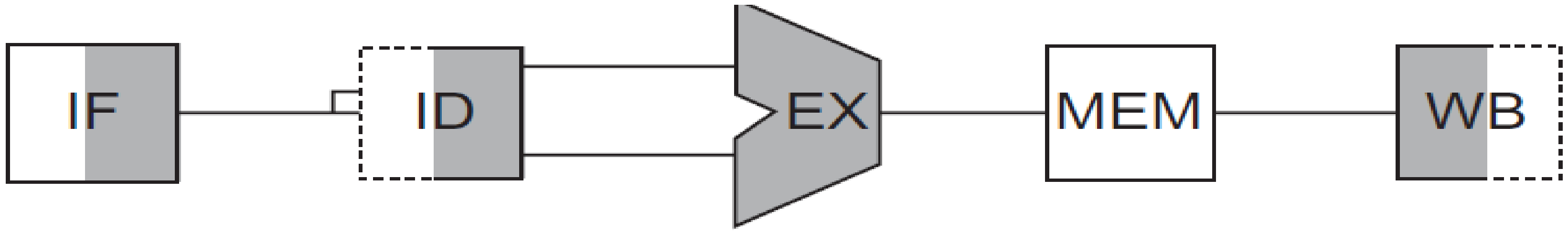
```
I:  SUB X1,X4,X3
J:  ADD X1,X2,X3
```

- 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 Instrucstion

•

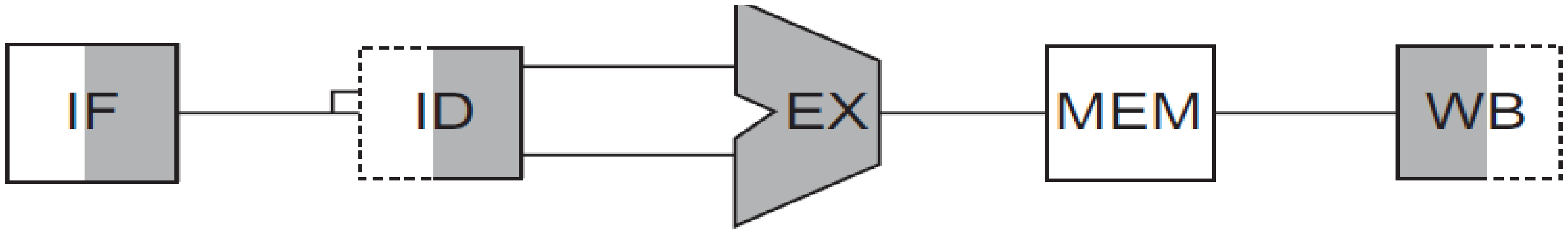


Cycles	1	2	3	4	5
ADD X19, X0, X1	Fetch instruction from mem	Read data from registers 0, 1	Add values of registers 0, 1		🟩
SUB X2, X19, X3		Fetch instruction from mem	Read data from registers 19, 3		

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

Example R-Type Instrucstion

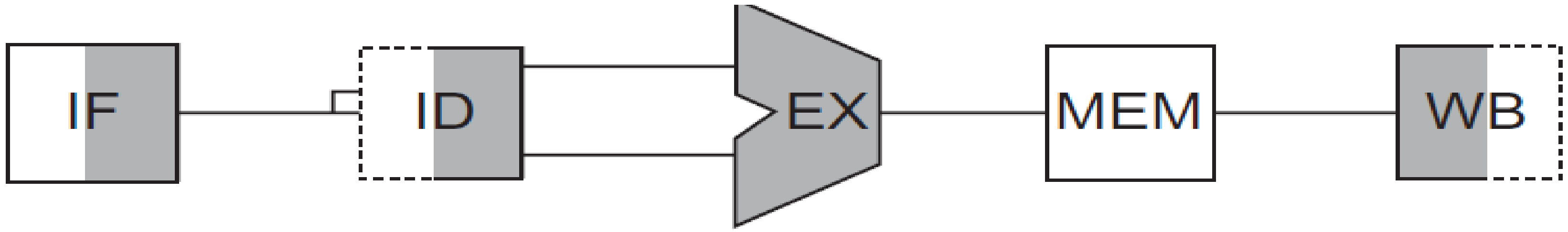
•



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 (value to write in reg 19 is computed)		🟩
`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.

Example R-Type Instruction

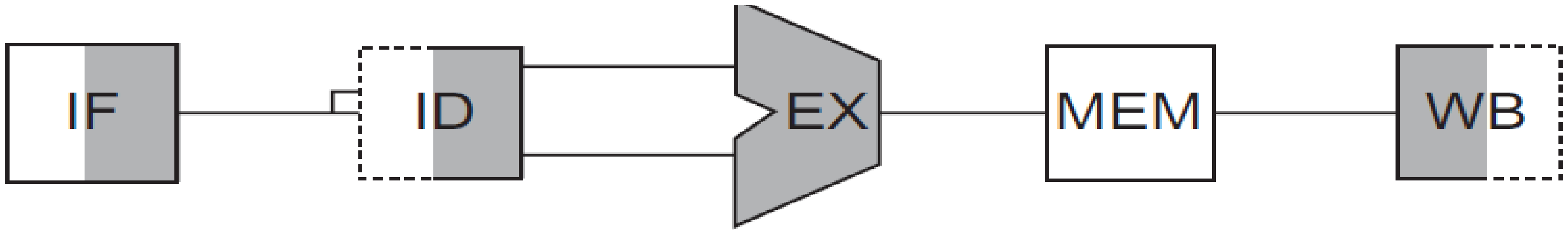


Cycles	1	2	3	4	5
`ADD x19, x0, x1`	Fetch instruction from mem	Read data from registers 0, 1	Add values of registers 0, 1 (value to write in reg 19 is computed)		🟩
`SUB x2, x19, x3`		Fetch instruction from mem	Read data from registers 19, 3	Forward value Bypass	

Example R-Type Instruction

Cycle 3:
Inst 1 ➔ Sum Is compute (X0+X1)
Inst 2 ➔ values for register 19 and 3 are read (X19 is not valid)

•

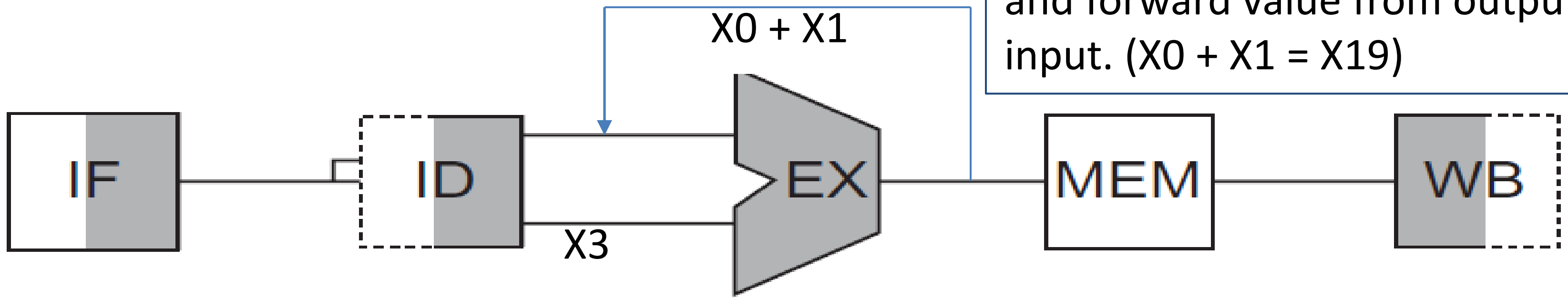


Cycles	1	2	3	4	5
ADD X19, X0, X1	Fetch instruction from mem	Read data from registers 0, 1	Add values of registers 0, 1 (value to write in reg 19 is computed)		🟩
SUB X2, X19, X3		Fetch instruction from mem	Read data from registers 19, 3		

Forward value
Bypass

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)

- 

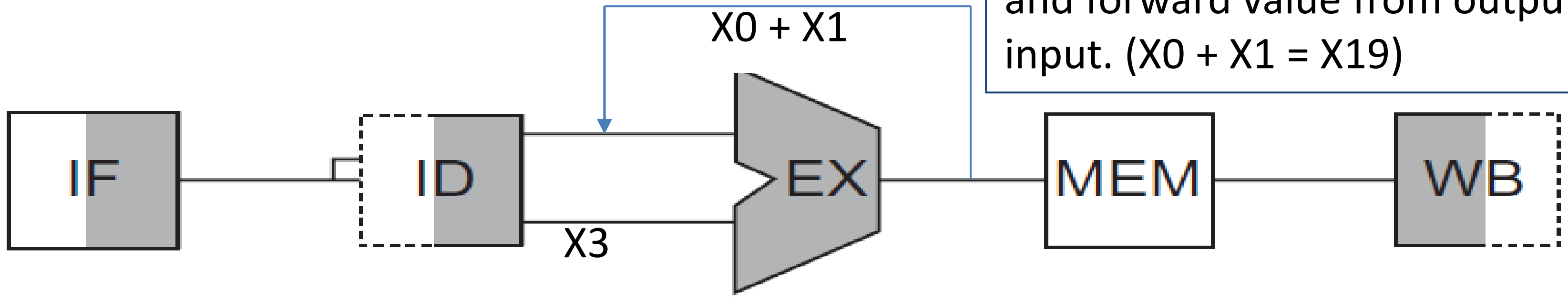
Cycles	1	2	3	4	5
ADD X19, X0, X1	Fetch instruction from mem	Read data from registers 0, 1	Add values of registers 0, 1 (value to write in reg 19 is computed)		🟩
SUB X2, X19, X3		Fetch instruction from mem	Read data from registers 19, 3	Subtract X0 from X19	

Forward value Bypass

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 X19, 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, X19, X3		Fetch instruction from mem	Read data from registers 19, 3	Subtract X0 from X19	

Forward value Bypass

No Need to Stall!!

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

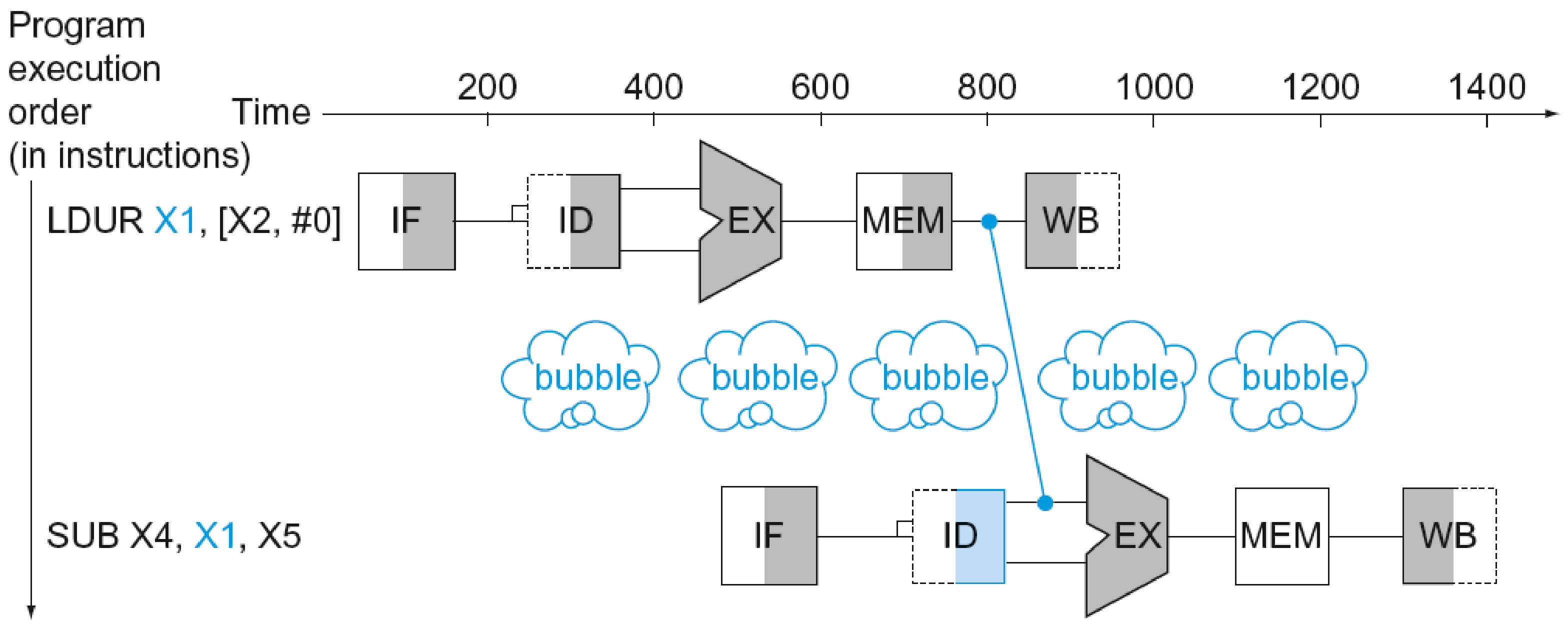
Data Hazards Even with Forwarding

```
LDUR X1 [ X2, #0]
SUB X4, X1, X5
```

Data Hazards Even with Forwarding

LDUR X1 [x2, #0]

SUB X4, X1, X5

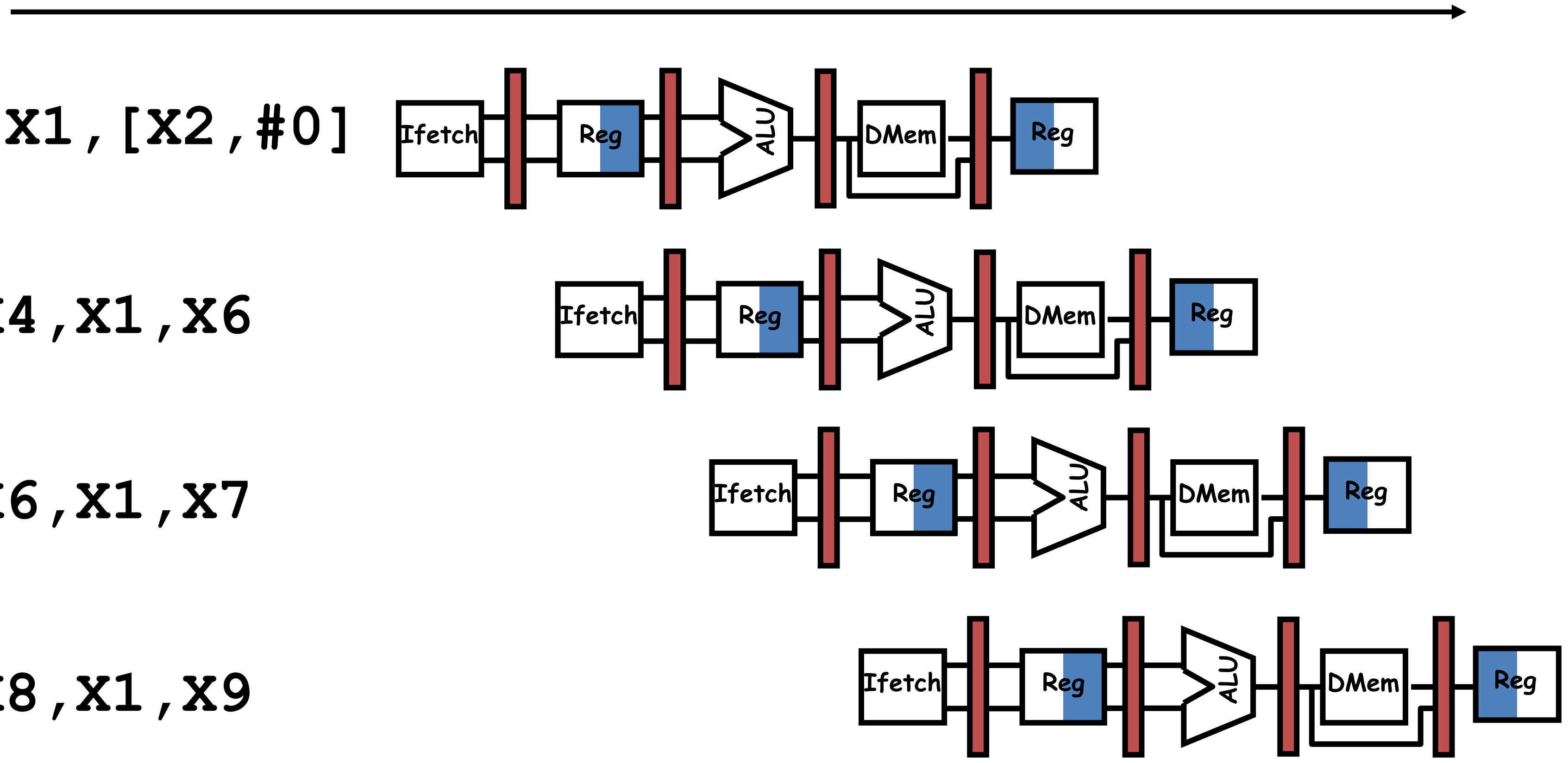


Data Hazard even with Forwarding

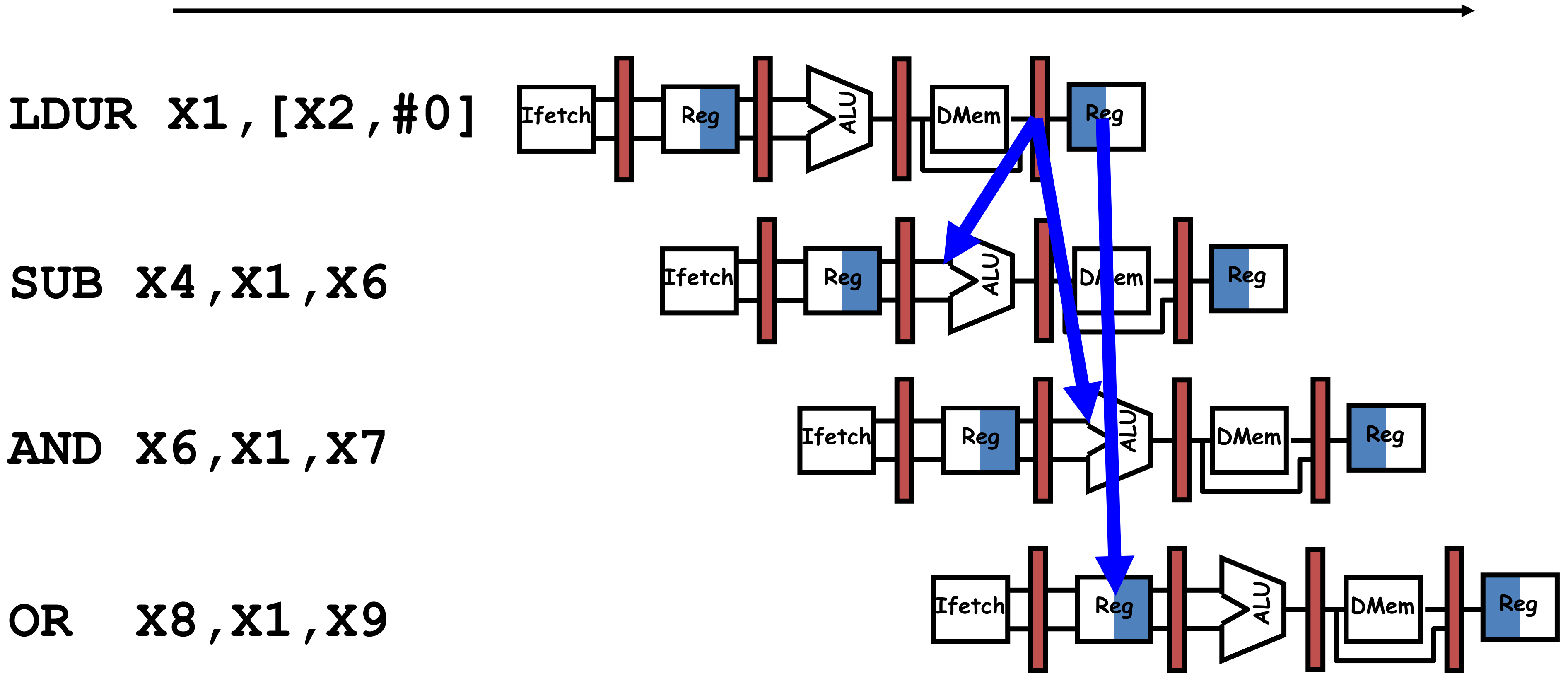
```
LDUR X1,[X2,#0]

SUB X4,X1,X6

AND X6,X1,X7

OR  X8,X1,X9
```

Data Hazard even with Forwarding

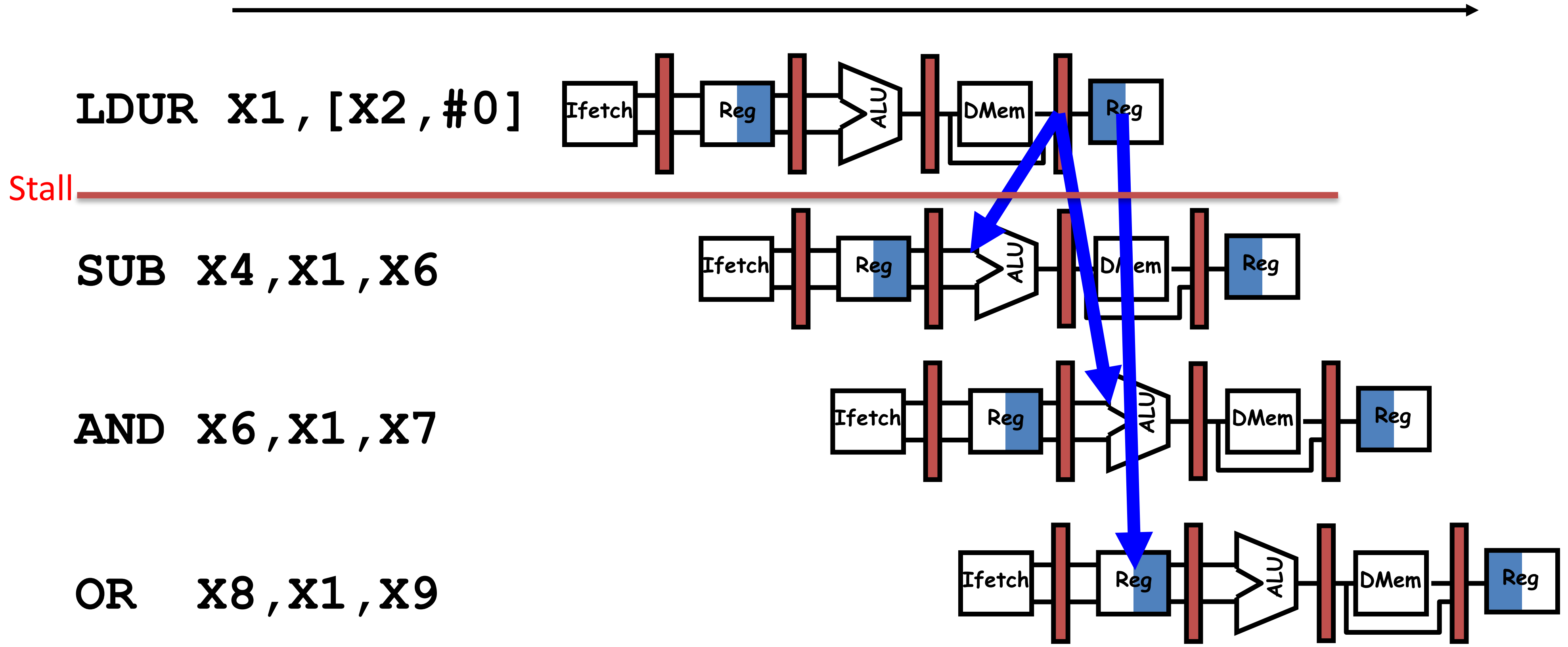
```
LDUR X1,[X2,#0]

SUB  X4,X1,X6

AND  X6,X1,X7

OR   X8,X1,X9
```



Data Hazard even with Forwarding



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]
```

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]
```

Code Scheduling to Avoid Stalls

Clock Cycles	1	2	3	4	5	6	7	8	9	10	11
LDUR X1, [X0,#0]	IF										
LDUR X2, [X0,#8]											
ADD X3, X1, X2											

Code Scheduling to Avoid Stalls

Clock Cycles	1	2	3	4	5	6	7	8	9	10	11
LDUR X1, [X0,#0]	IF	ID									
LDUR X2, [X0,#8]		IF									
ADD X3, X1, X2											

Code Scheduling to Avoid Stalls

Clock Cycles	1	2	3	4	5	6	7	8	9	10	11
LDUR X1, [X0,#0]	IF	ID	EXE	MEM	WB						
LDUR X2, [X0,#8]		IF	ID	EXE	MEM						
ADD X3, X1, X2			IF	ID	EXE						

X1 is available
X2?

Code Scheduling to Avoid Stalls

Clock Cycles	1	2	3	4	5	6	7	8	9	10	11
LDUR X1, [X0,#0]	IF	ID	EXE	MEM	WB						
LDUR X2, [X0,#8]		IF	ID	EXE	MEM						
ADD X3, X1, X2			IF	ID	EXE						

X1 is available
X2?
Even with forwarding not available until after CC 5

Code Scheduling to Avoid Stalls

Clock Cycles	1	2	3	4	5	6	7	8	9	10	11
LDUR X1, [X0,#0]	IF	ID	EXE	MEM	WB						
LDUR X2, [X0,#8]		IF	ID	EXE	MEM						
Stall											
ADD X3, X1, X2				IF	ID	EXE					

X1 is available
X2?
Even with forwarding not available until after CC 5

Code Scheduling to Avoid Stalls

	1	2	3	4	5	6	7	8	9	10	11	12
LDUR X1, [X0,#0]	IF	ID	EXE	MEM	WB							
LDUR X2, [X0,#8]		IF	ID	EXE	MEM	WB						
Stall												
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		
ADD X5, X1, X4							IF	ID	EXE	MEM	WB	

X1 is available
X4?
Even with forwarding not available until after CC 9

Code Scheduling to Avoid Stalls

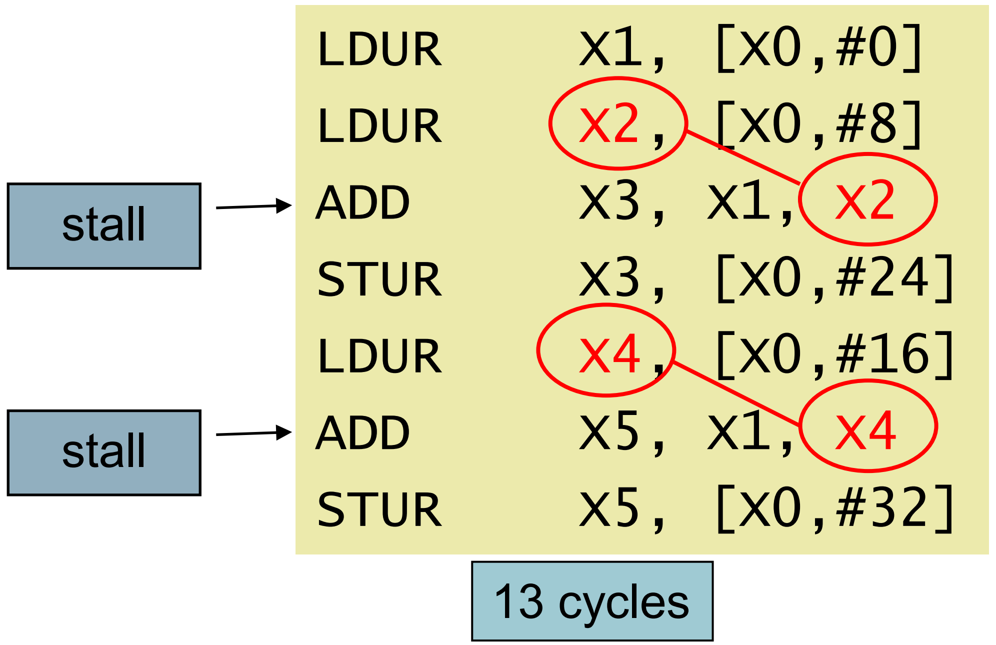
	1	2	3	4	5	6	7	8	9	10	11	12
LDUR X1, [X0,#0]	IF	ID	EXE	MEM	WB							
LDUR X2, [X0,#8]		IF	ID	EXE	MEM	WB						
Stall												
ADD X3, X1, X2				IF	ID	EXE	MEM	WB				
STUR X3, [X0,#24]					IF	ID	EXE	MEM	WB			
LDUR X4, [X0,#16]						IF	ID	EXE	MEM	WB		
Stall												
ADD X5, X1, X4								IF	ID	EXE	MEM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

X1 is available
X4?
Even with forwarding not available until after CC 9

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							
Stall													
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			
Stall													
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



```
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]
```

Code Scheduling to Avoid Stalls

```
        LDUR    X1, [X0,#0]
        LDUR    X2, [X0,#8]
stall → ADD     X3, X1, X2
        STUR    X3, [X0,#24]
        LDUR    X4, [X0,#16]
stall → ADD     X5, X1, X4
        STUR    X5, [X0,#32]
```

13 cycles

```
LDUR    X1, [X0,#0]
LDUR    X2, [X0,#8]
LDUR    X4, [X0,#16]
ADD     X3, X1, X2
STUR    X3, [X0,#24]
ADD     X5, X1, X4
STUR    X5, [X0,#32]
```

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					

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					

X1 Available
X2 Available from Forwarding
No need to stall

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			

X1 Available
X4 Available
No need to stall

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

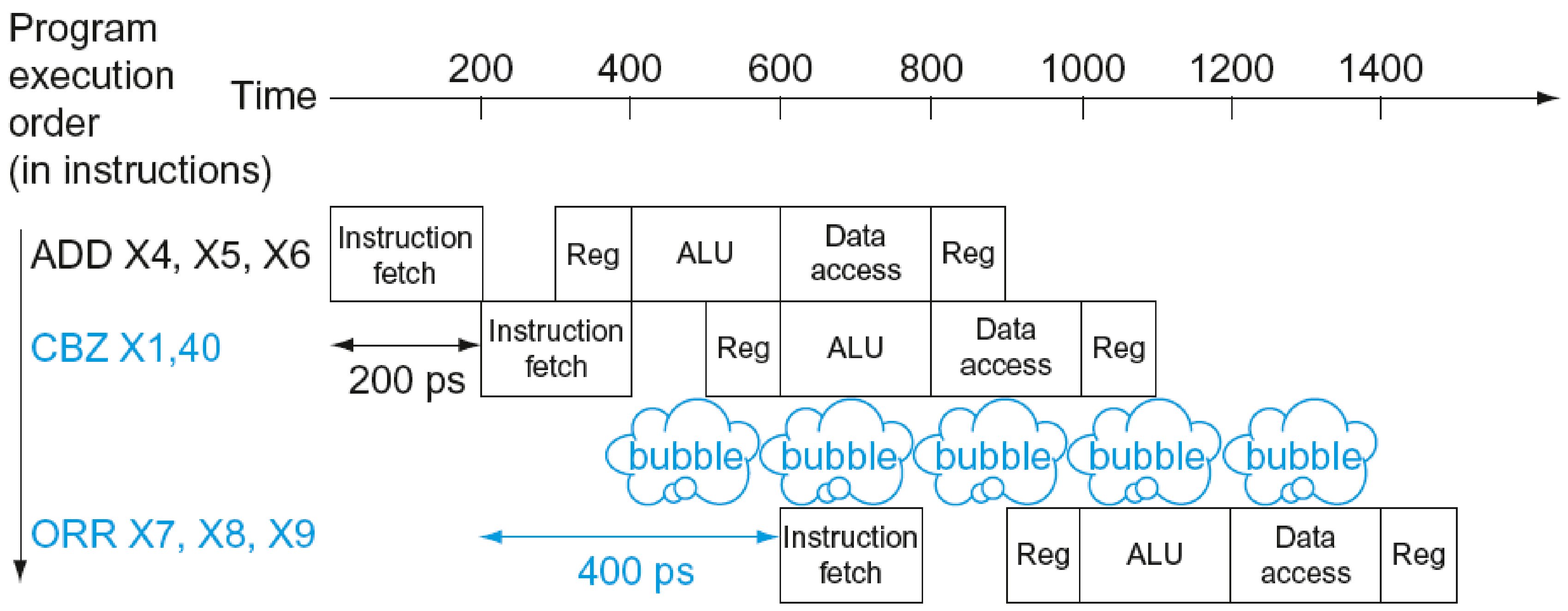
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - Not necessarily an issue in LEGv8
- Data hazard
  - Forwarding or Bypassing (Can still stall)
  - Code scheduling
- Control hazard
  - Deciding on control action depends on previous instruction
  - Fetching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

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