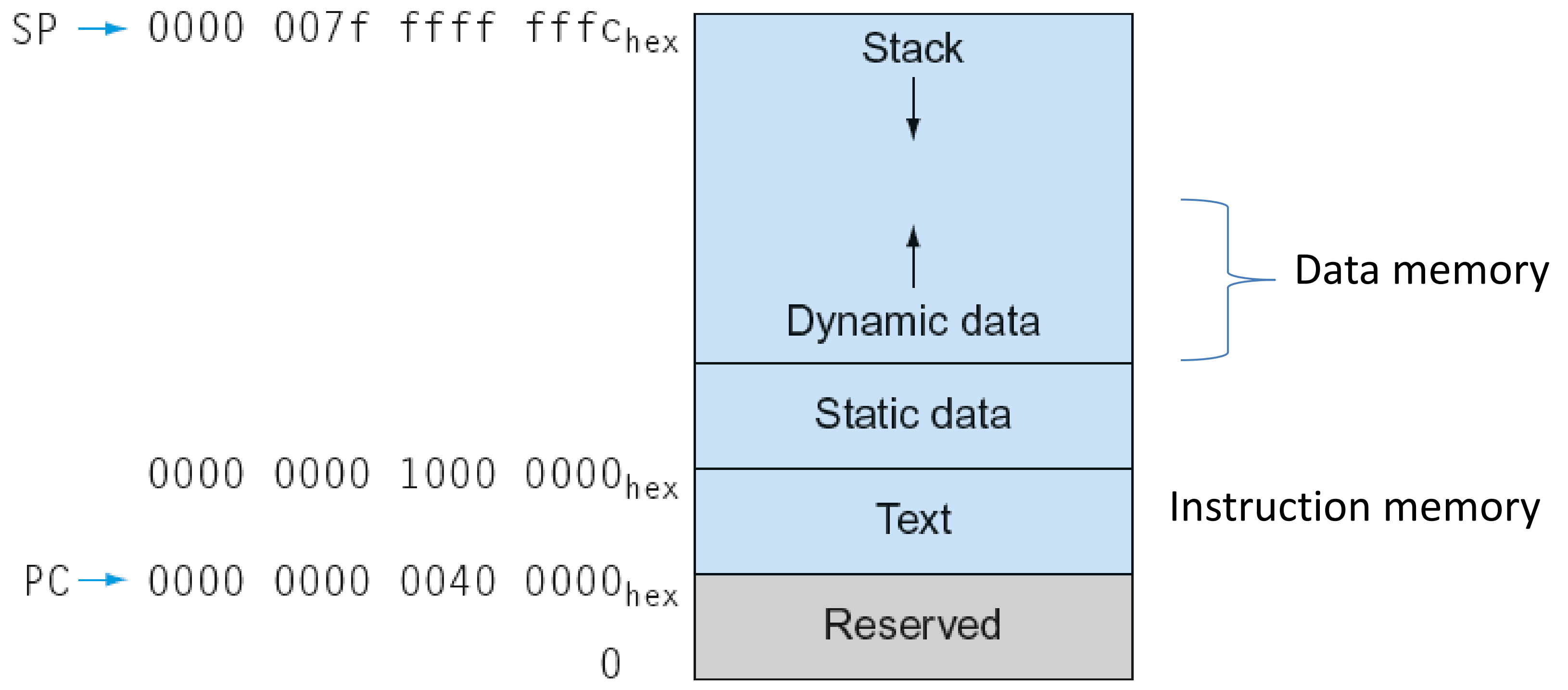
Computer Organization and Architecture

Lecture – 18

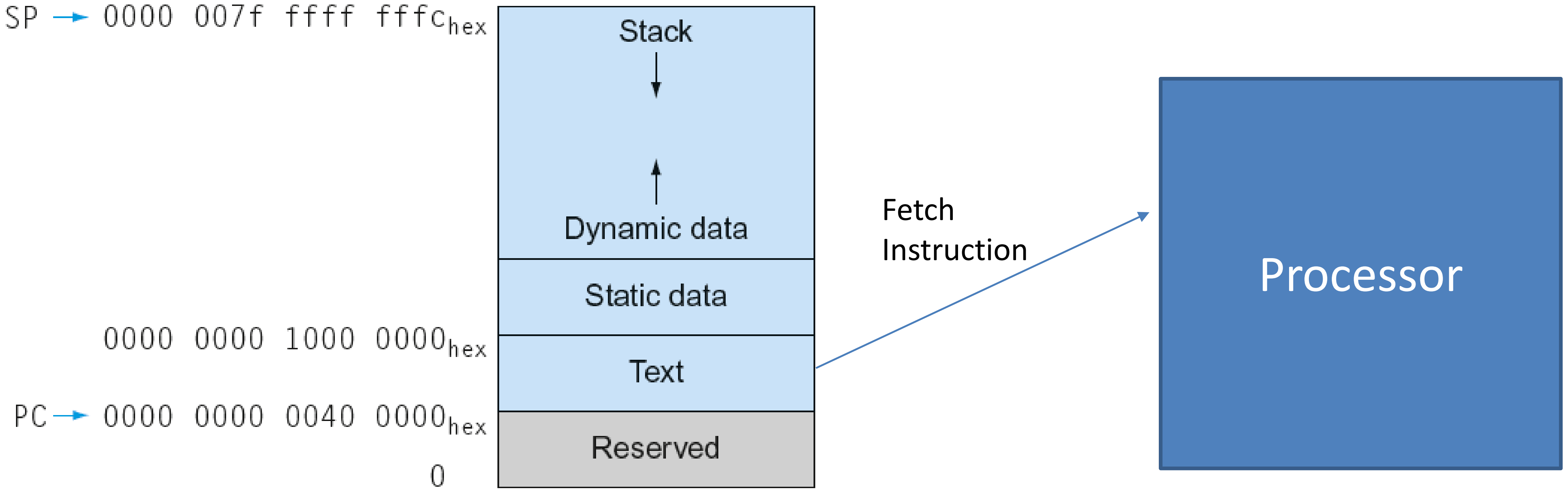
Oct 19th , 2022

Chapter – 4: The Processor

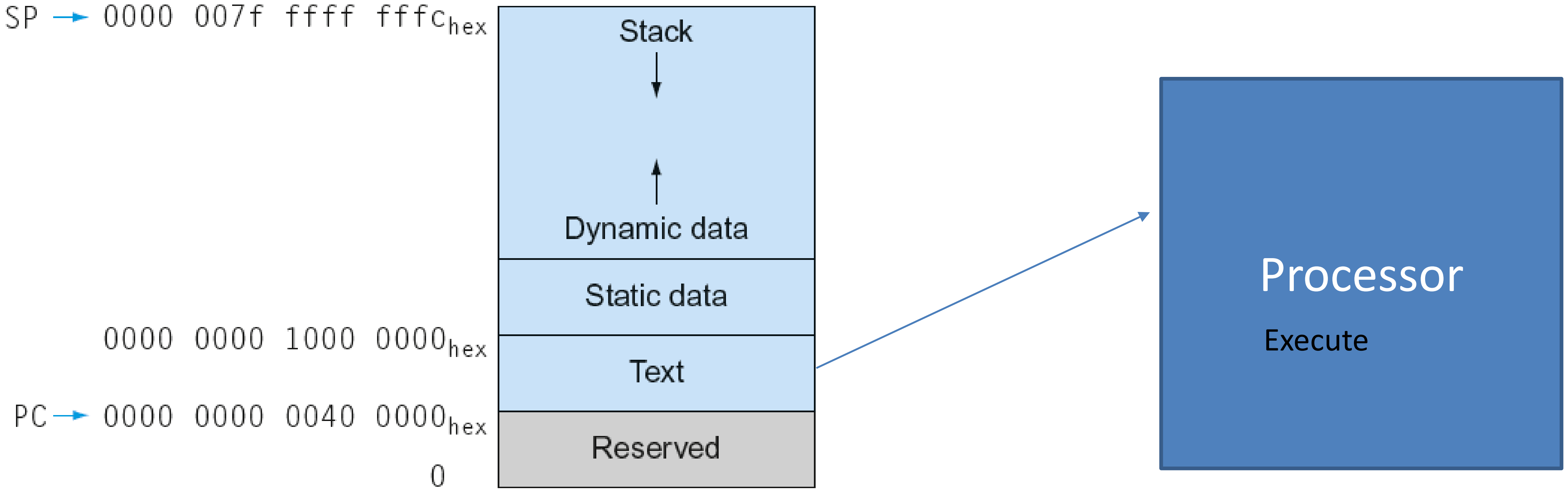
Memory Layout



Executing instructions


SP → 0000 007f ffff fffc$_{hex}$
Stack
Dynamic data
Static data
0000 0000 1000 0000$_{hex}$
Text
PC → 0000 0000 0040 0000$_{hex}$
Reserved
0
Fetch Instruction
Processor

Executing instructions



Introduction

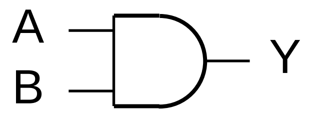
- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two LEGv8 implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: LDUR, STUR
  - Arithmetic/logical: ADD, SUB, AND, ORR
  - Control transfer: CBZ， B

Building a Datapath

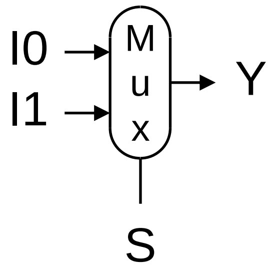
- Combine various circuits elements to create a processor
  - Combinational elements
  - State elements

Combinational Elements

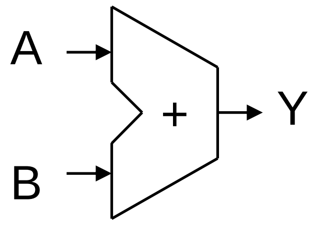
- AND-gate
  - Y = A & B



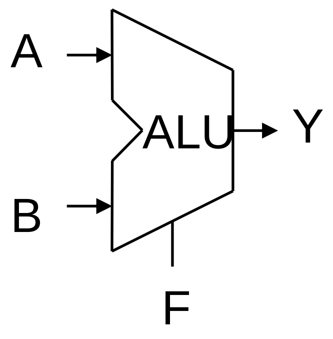
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

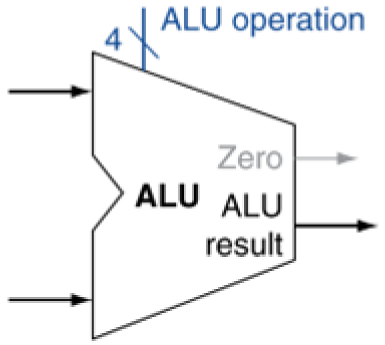


- Arithmetic/Logic Unit
  - Y = F(A, B)



ALU

- Combines adder and And/OR logic gate

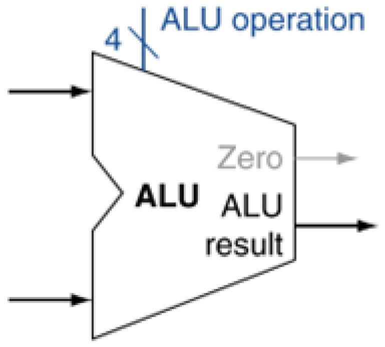


b. ALU

ALU

- Combines adder and And/OR logic gate

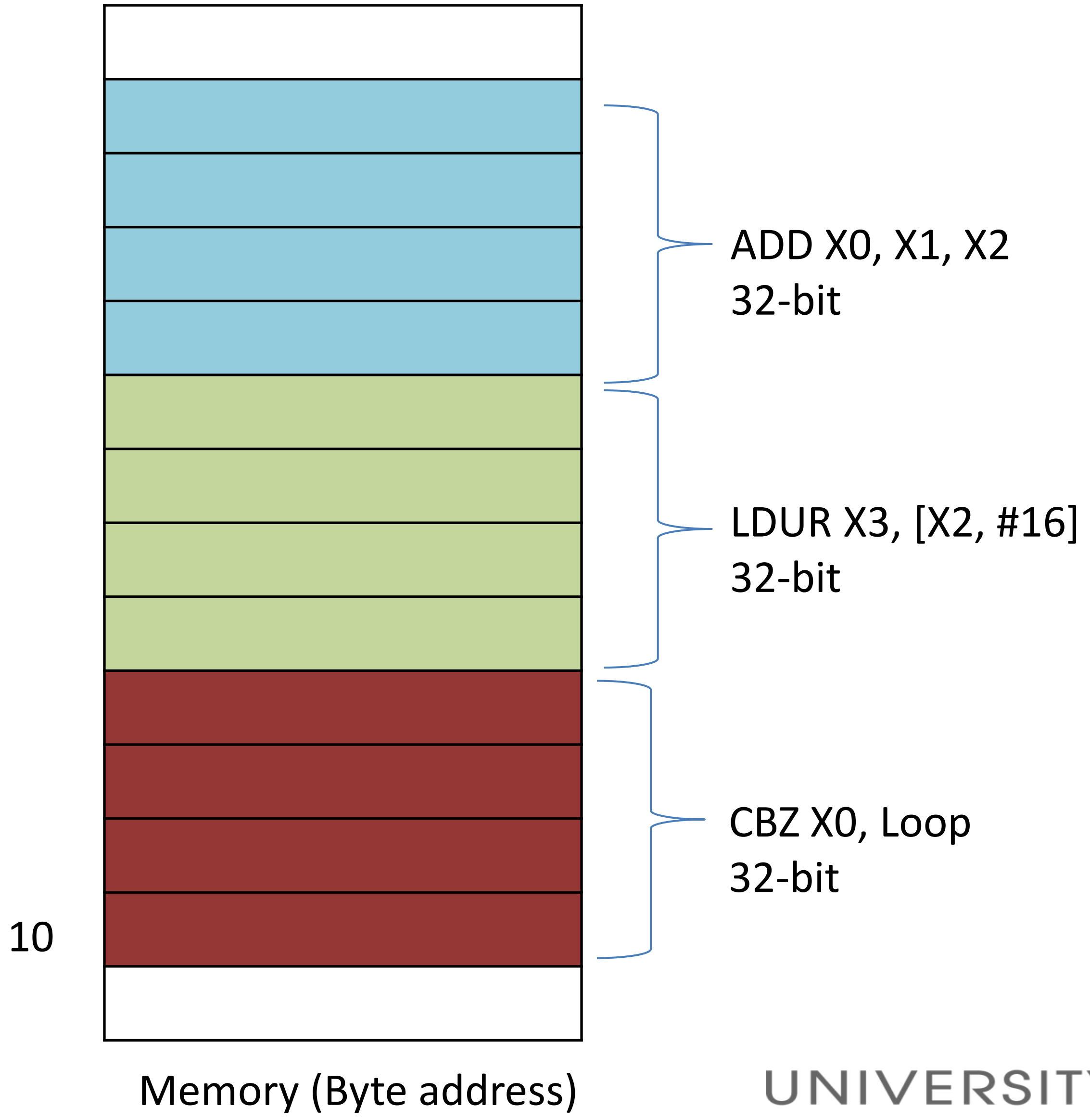
ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	pass input b
1100	NOR



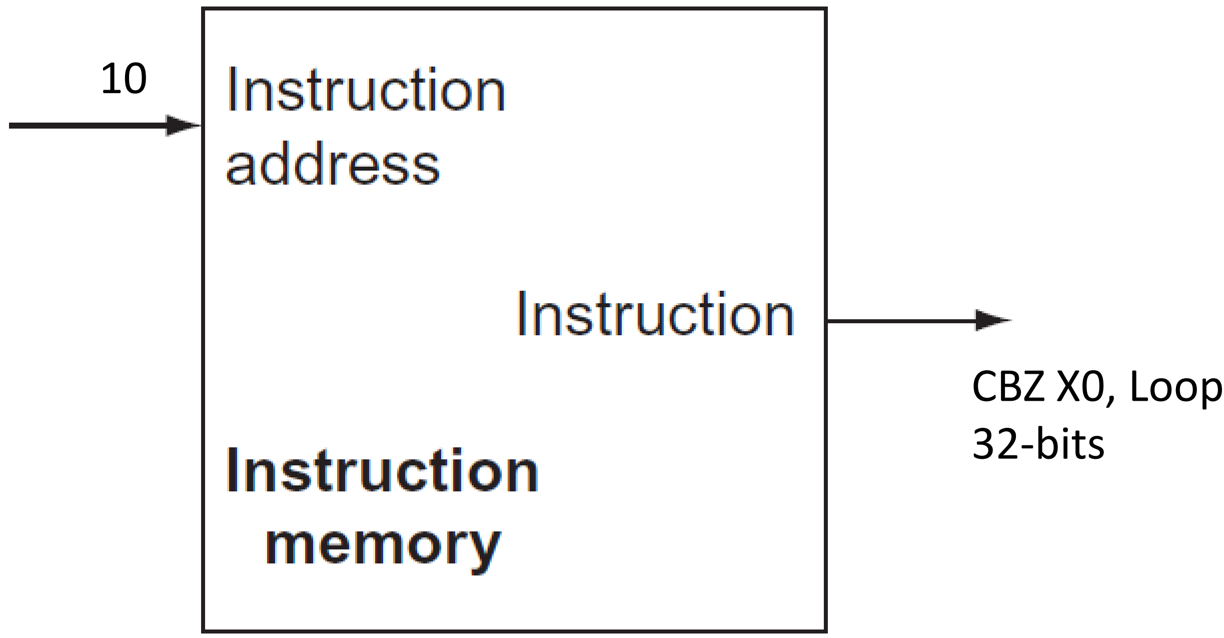
b. ALU

State Elements

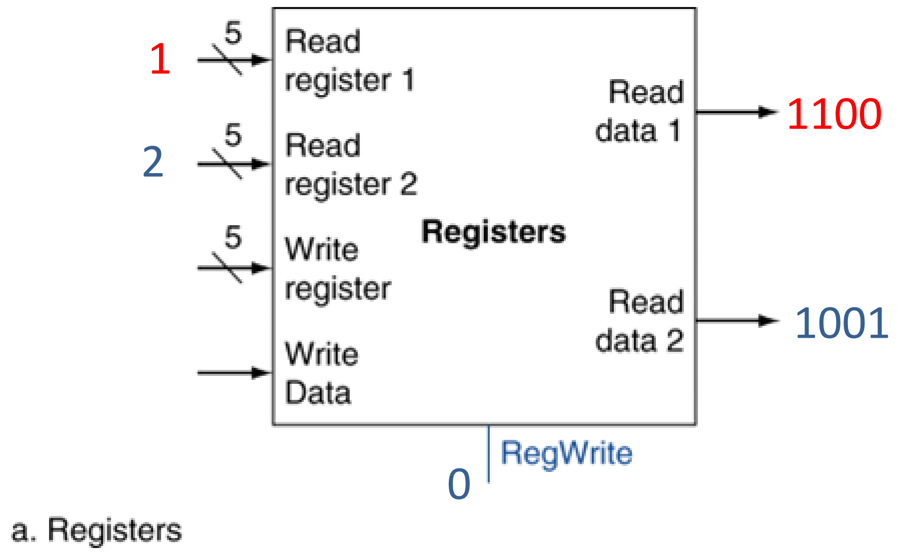
Instruction Memory



Instructions stored in memory



Register File



a. Registers

Register file:

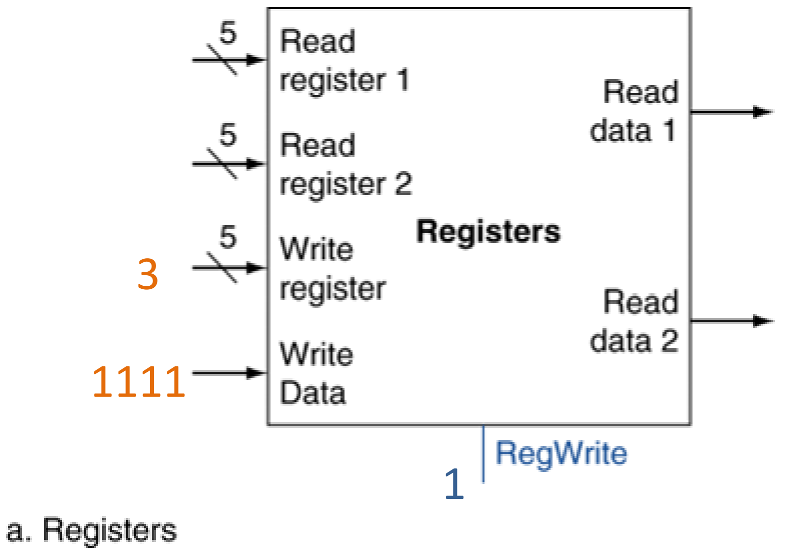
1. Read values from registers

2. Write values to registers

Register values

X0	1000
X1	1100
X2	1001
X3	
..	

Register File



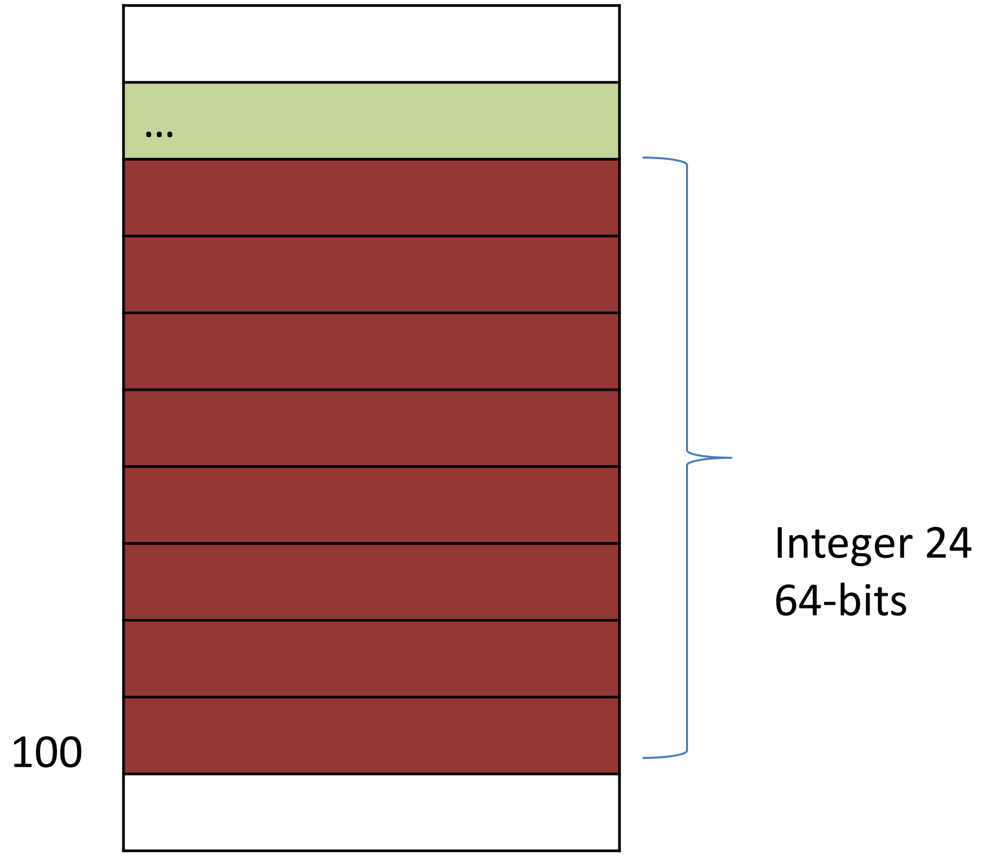
Register values

X0	1000
X1	1100
X2	1001
X3	1111
..	

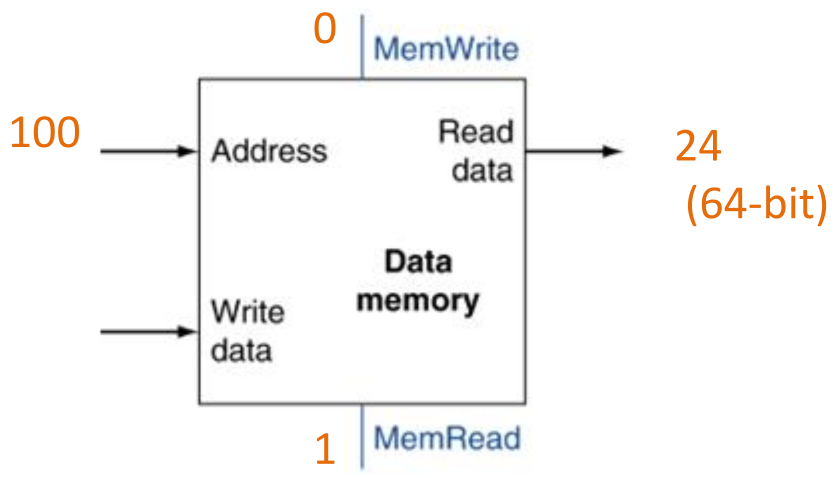
Register file:

1. Read values from registers
2. Write values to registers

Data Memory



Data stored in memory



a. Data memory unit

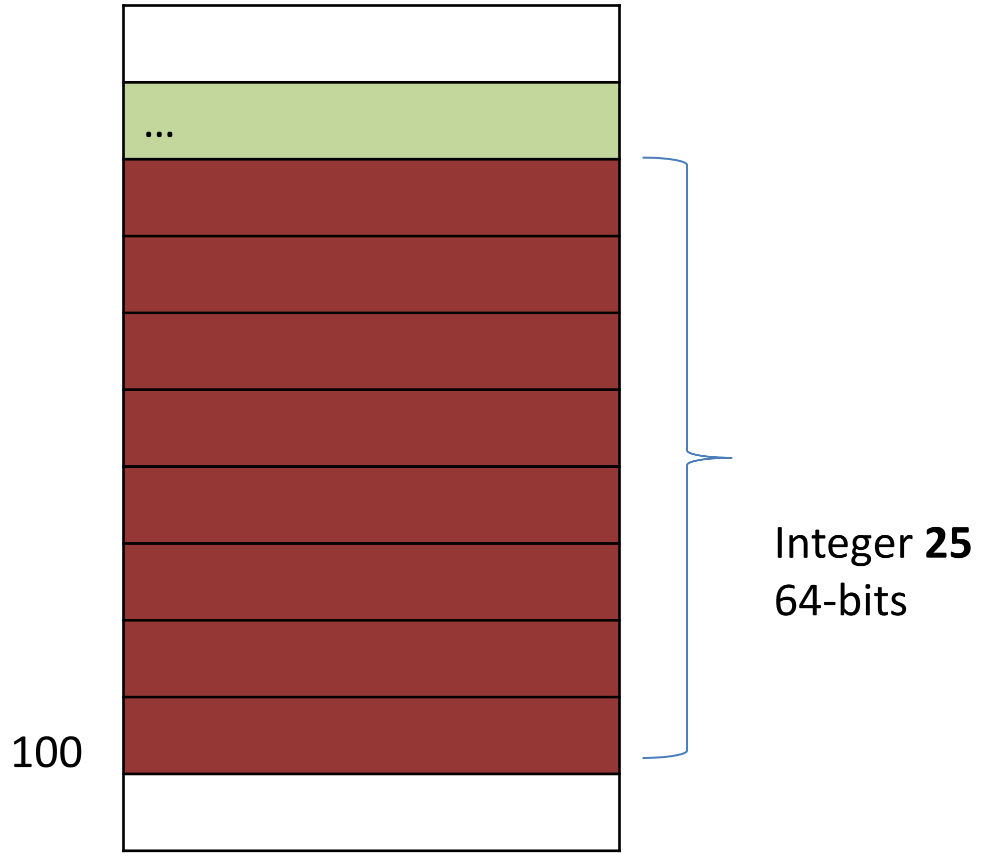
Register file:

1. Read values from memory

2. Write values to Memory

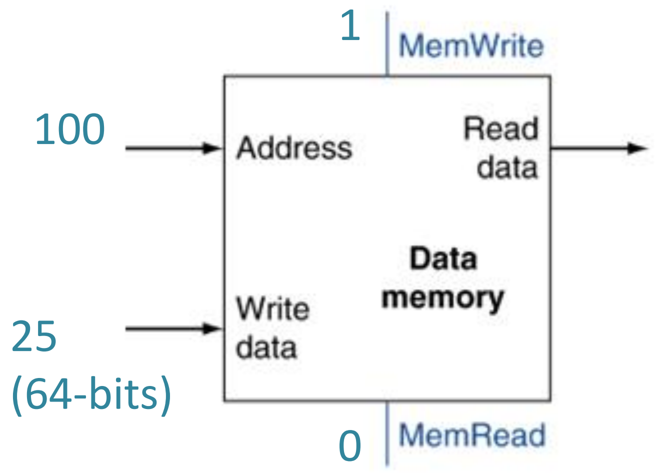
DataMemory (Byte address)

Data Memory



Register file:
1. Read values from memory
2. Write values to Memory

Data stored in memory



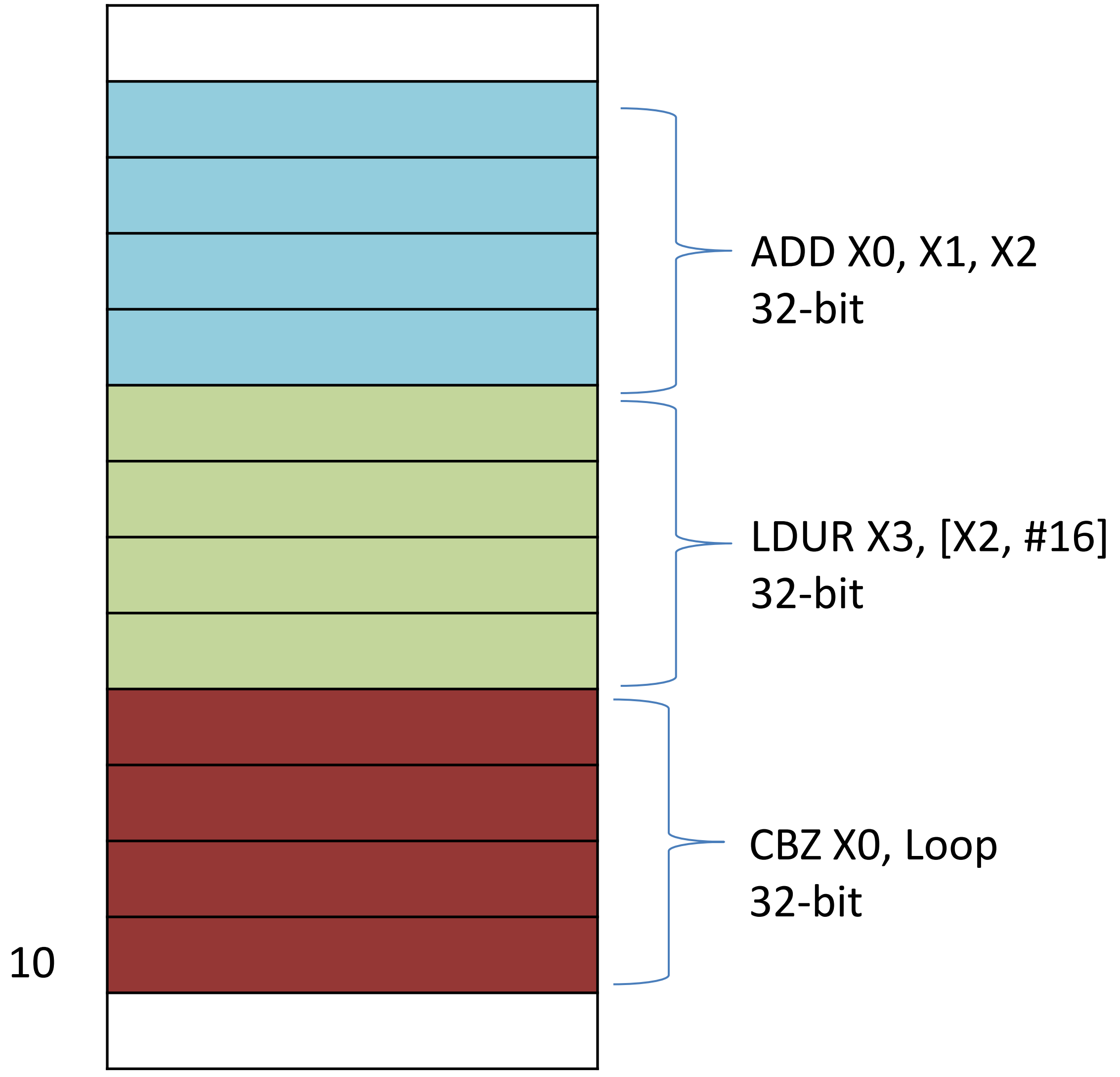
a. Data memory unit

DataMemory (Byte address)

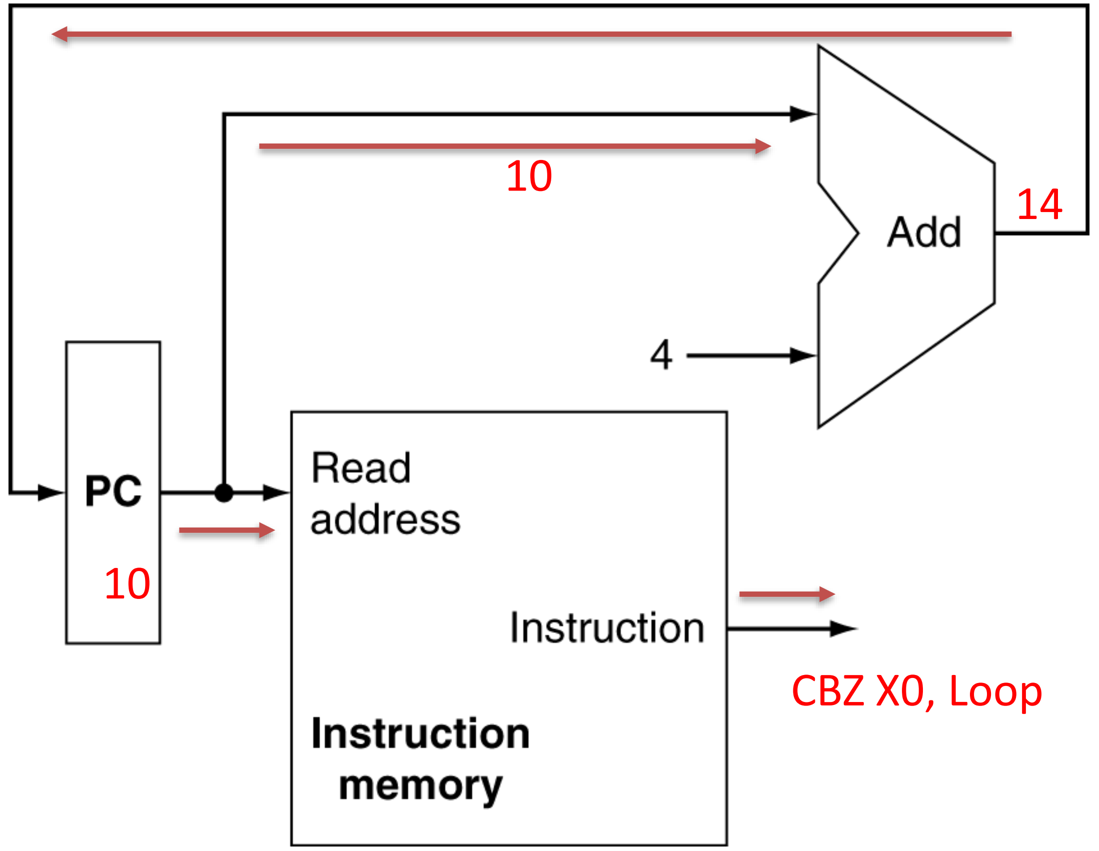
Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a LEGv8 datapath incrementally
  - **Fetch Instruction**
  - Execute Instruction
    - ADD, LDUR/STUR, CBZ

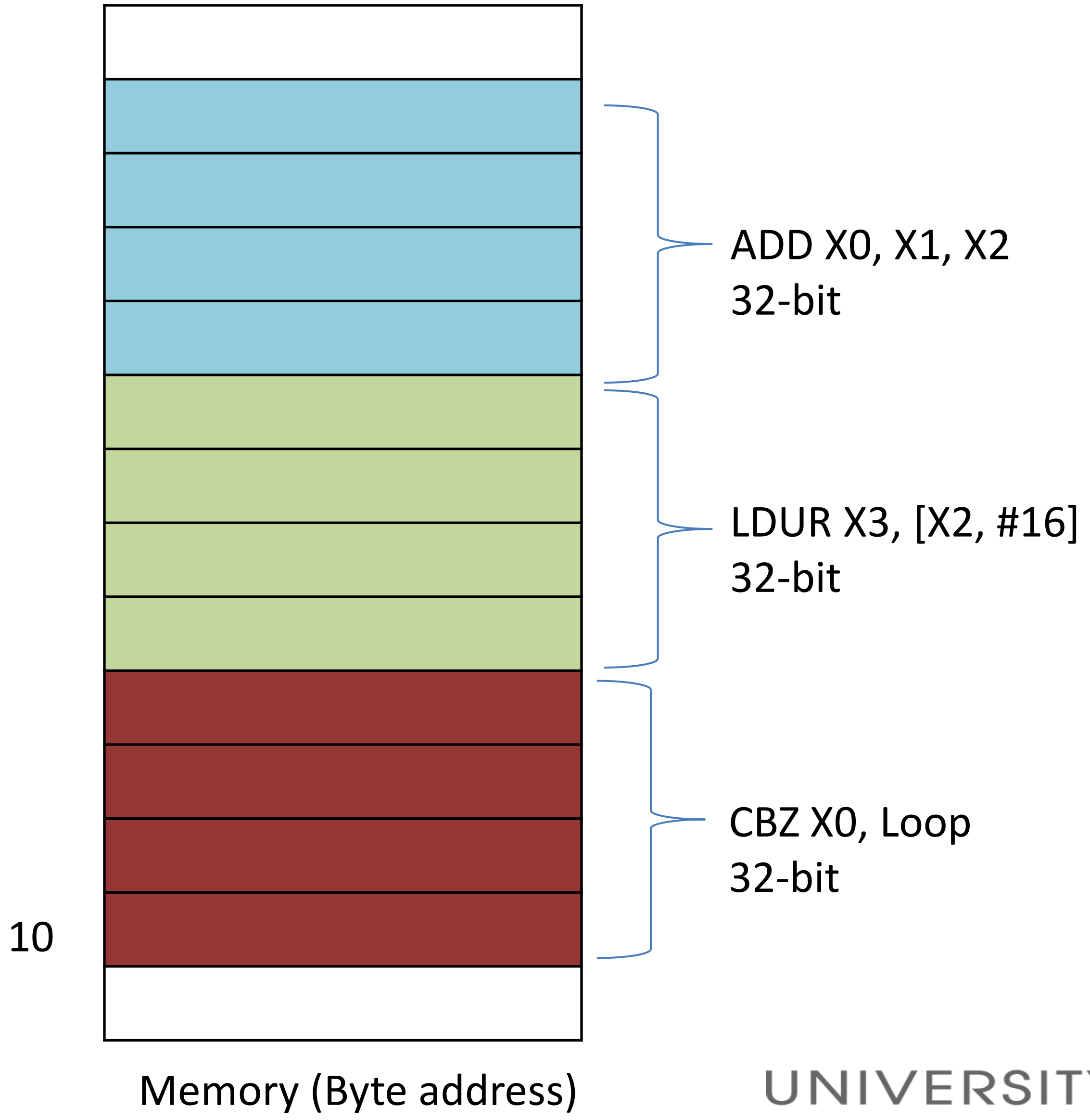
Fetch Instruction

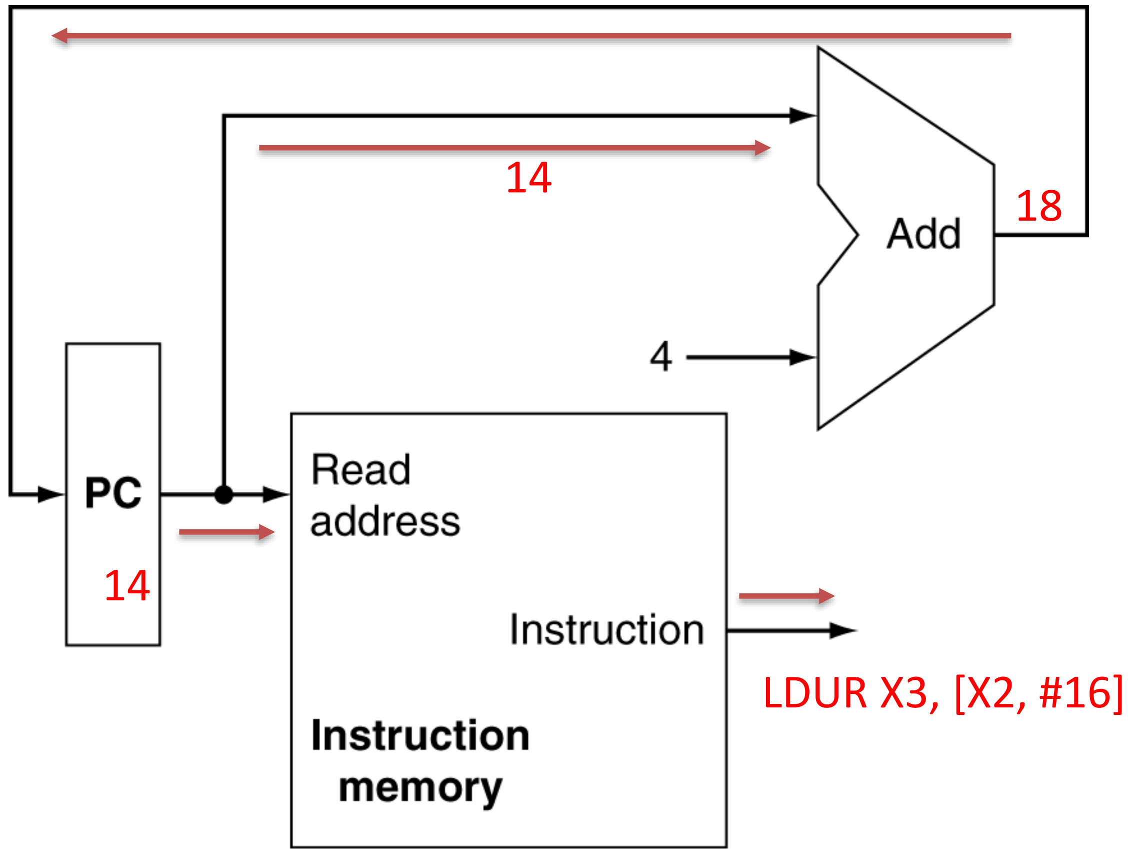


Memory (Byte address)



Fetch Instruction

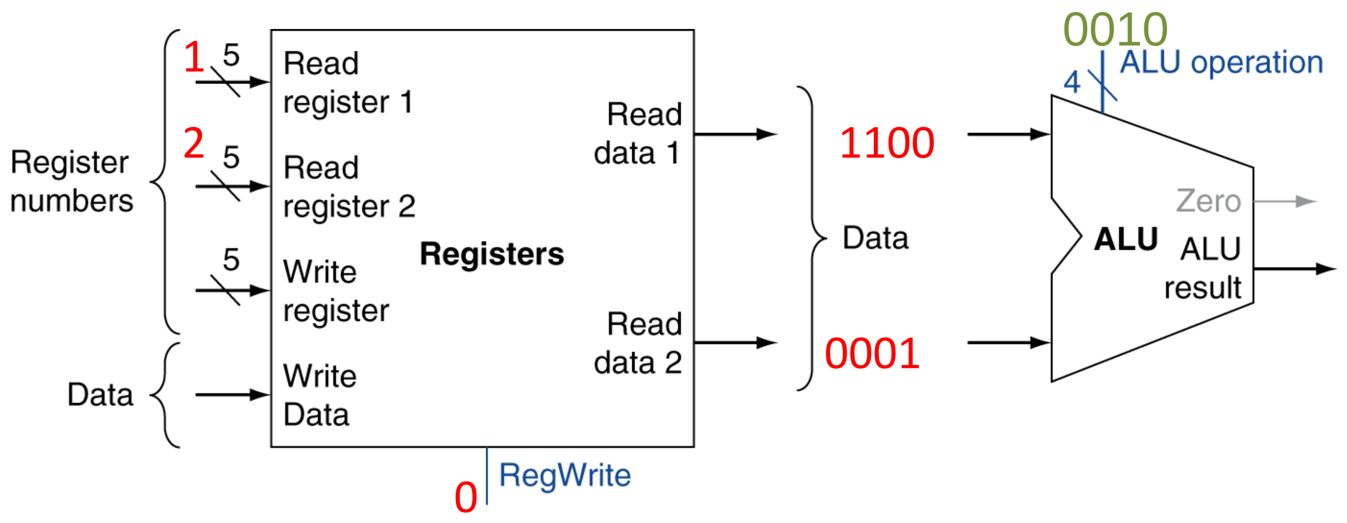




ADD (R-Type) Instruction

ADD (R-Type) Instruction

- EG. `ADD X3, X1, X2`
- Read two register operands
- Perform arithmetic/logical operation
- Write register result



a. Registers

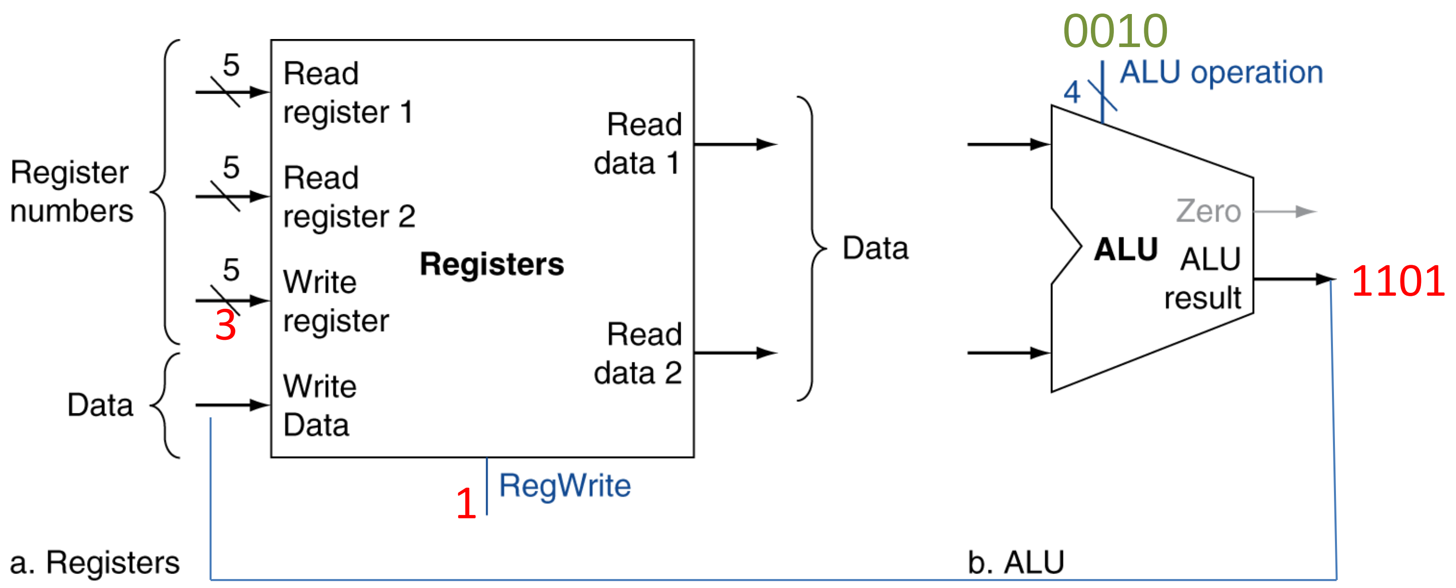
b. ALU

Register values

X0	1000
X1	1100
X2	0001
X3	
..	

ADD (R-Type) Instruction

- EG. `ADD X3, X1, X2`
- Read two register operands
- Perform arithmetic/logical operation
- Write register result

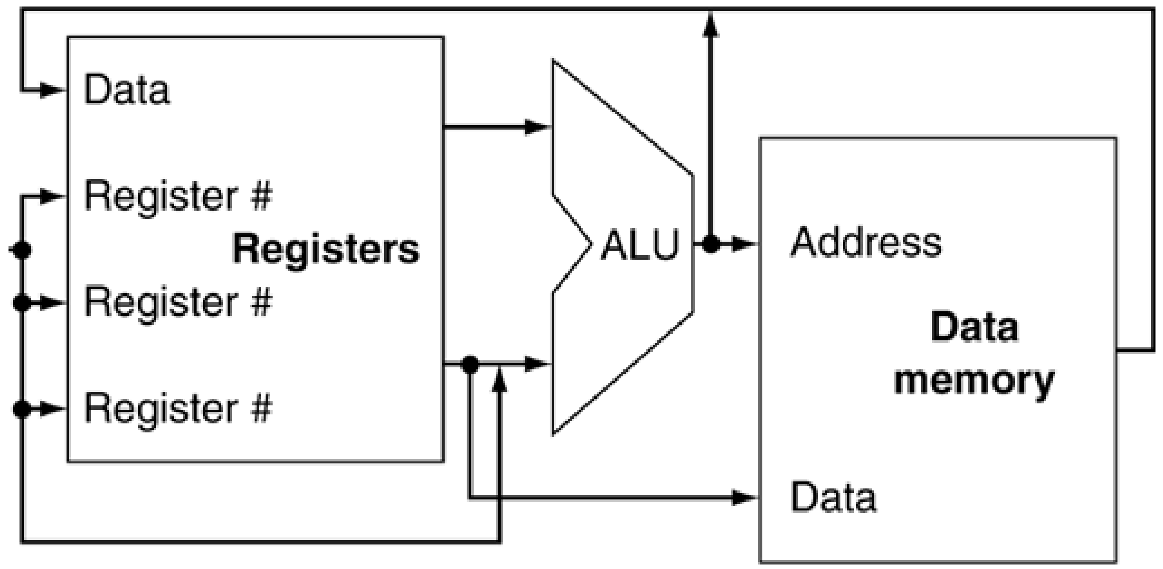


Register values

X0	1000
X1	1100
X2	0001
X3	1101
..	

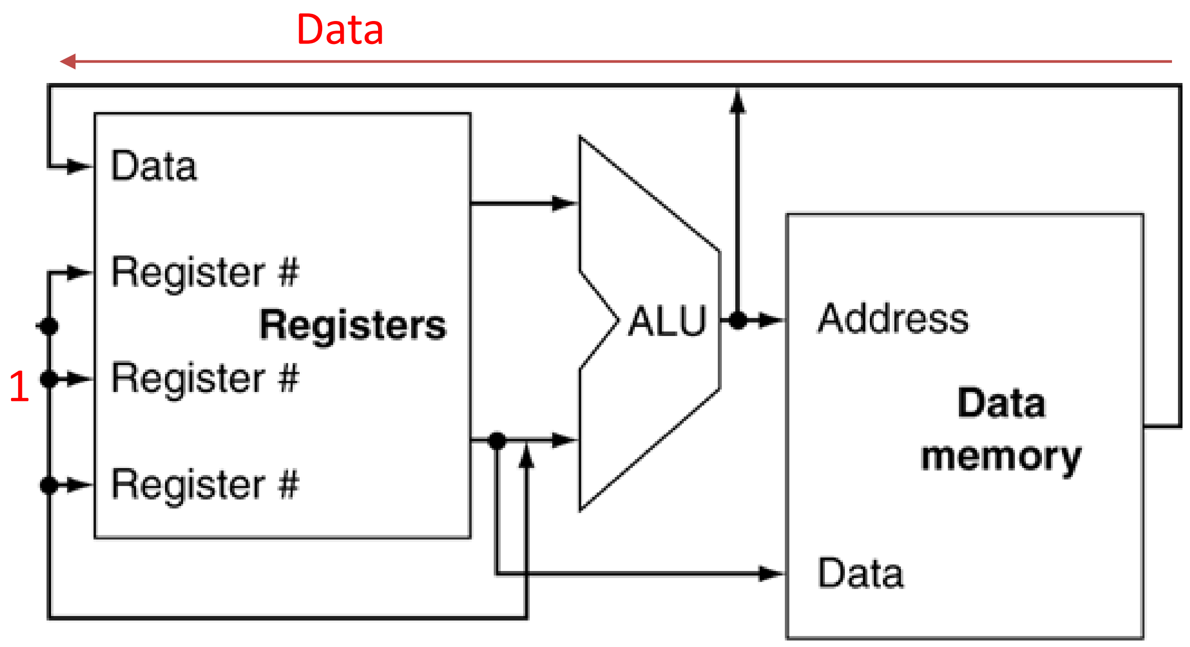
Load/Store Instruction (D-Type)

- `LDUR X1, [X2, #offset]`



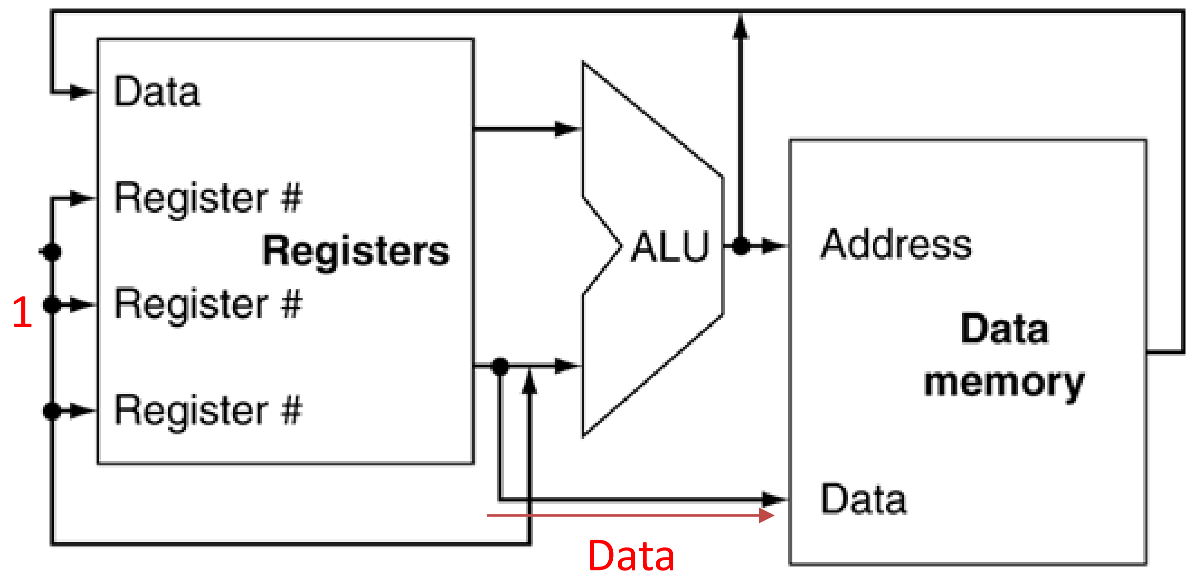
Load/Store Instruction (D-Type)

- `LDUR X1, [X2, #offset]`

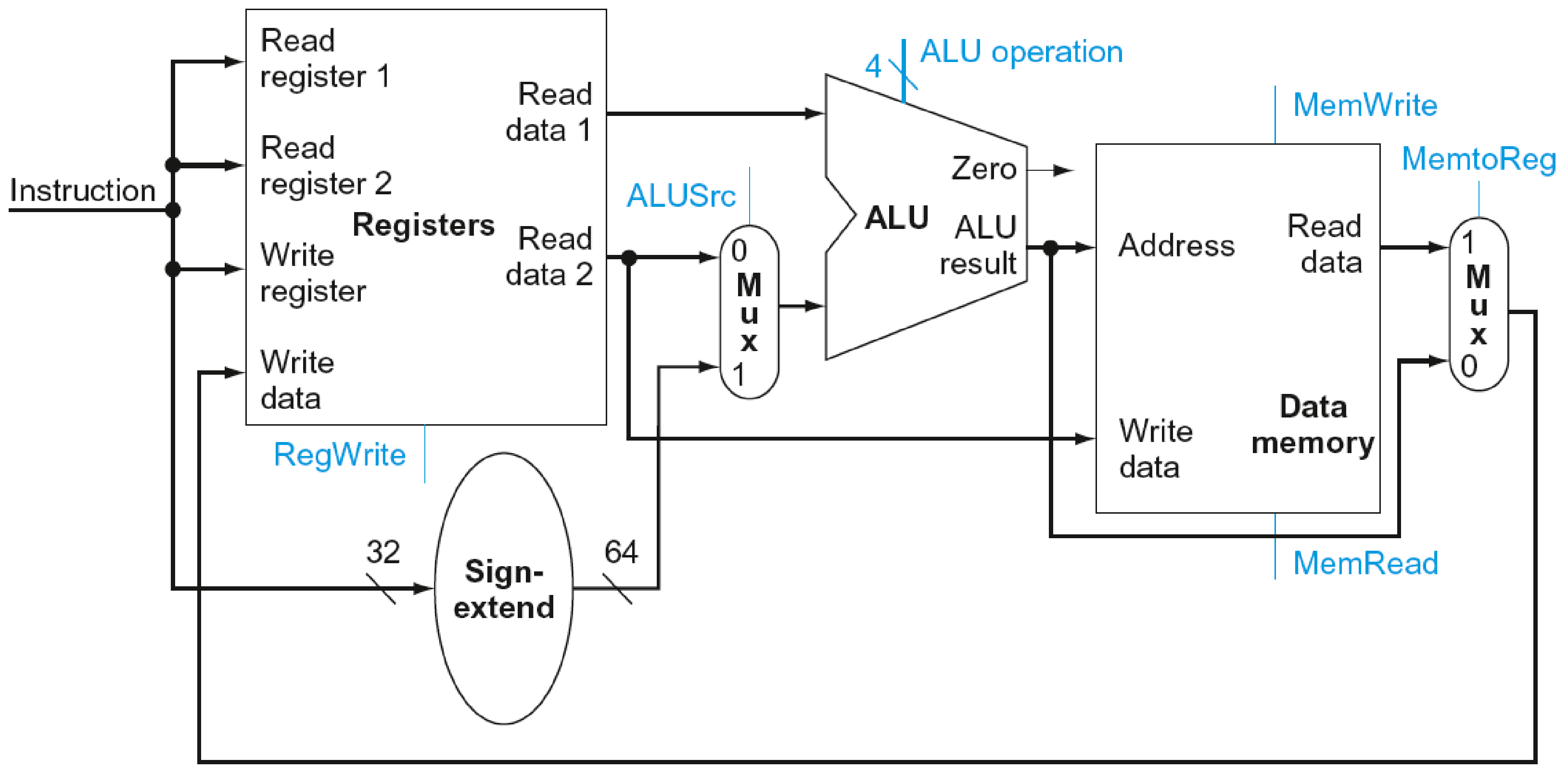


Load/Store Instruction (D-Type)

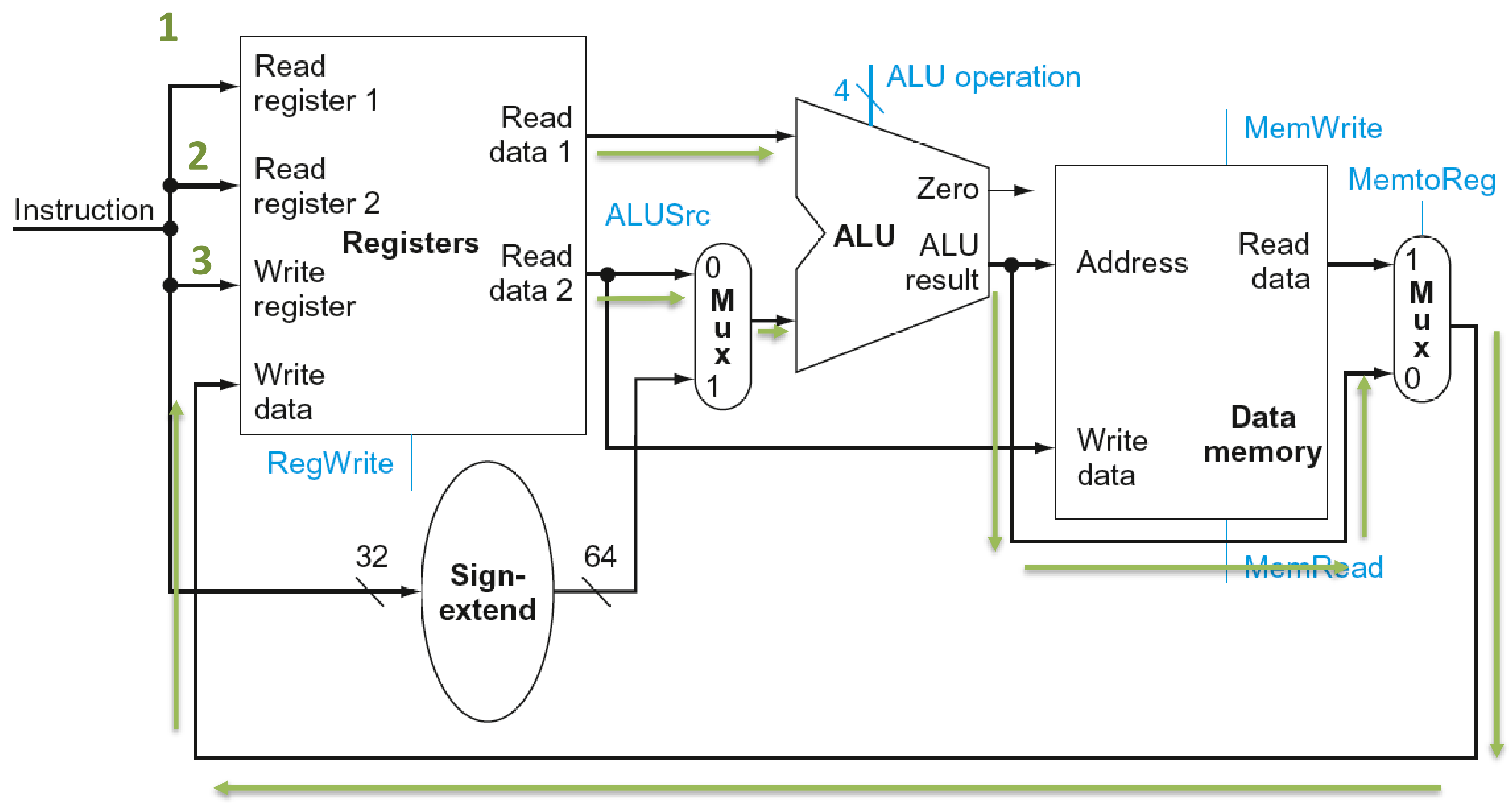
- `STUR X1, [X2, #offset]`



R-Type/Load/Store Datapath



R-Type/Load/Store Datapath



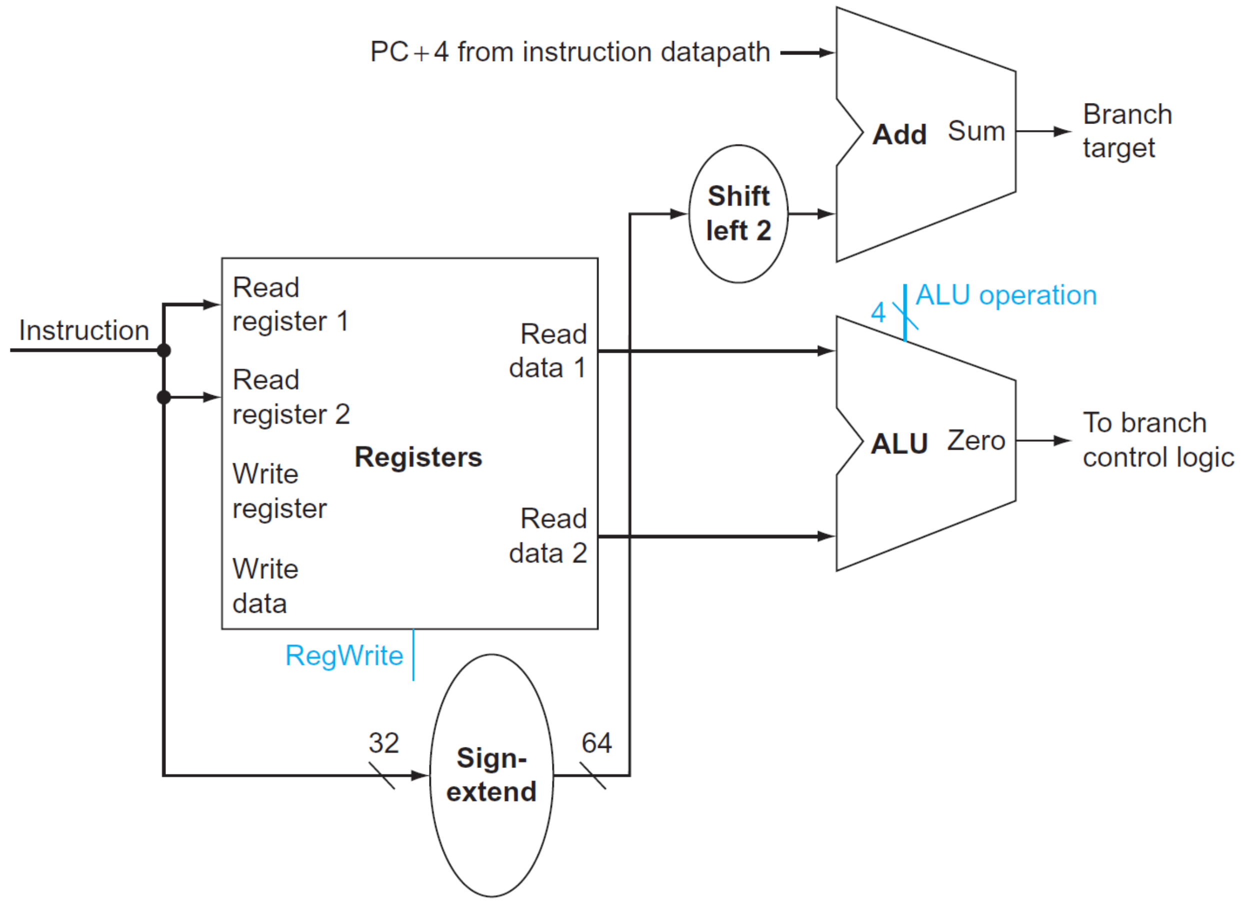
R-type
ADD X3, X1, X2

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

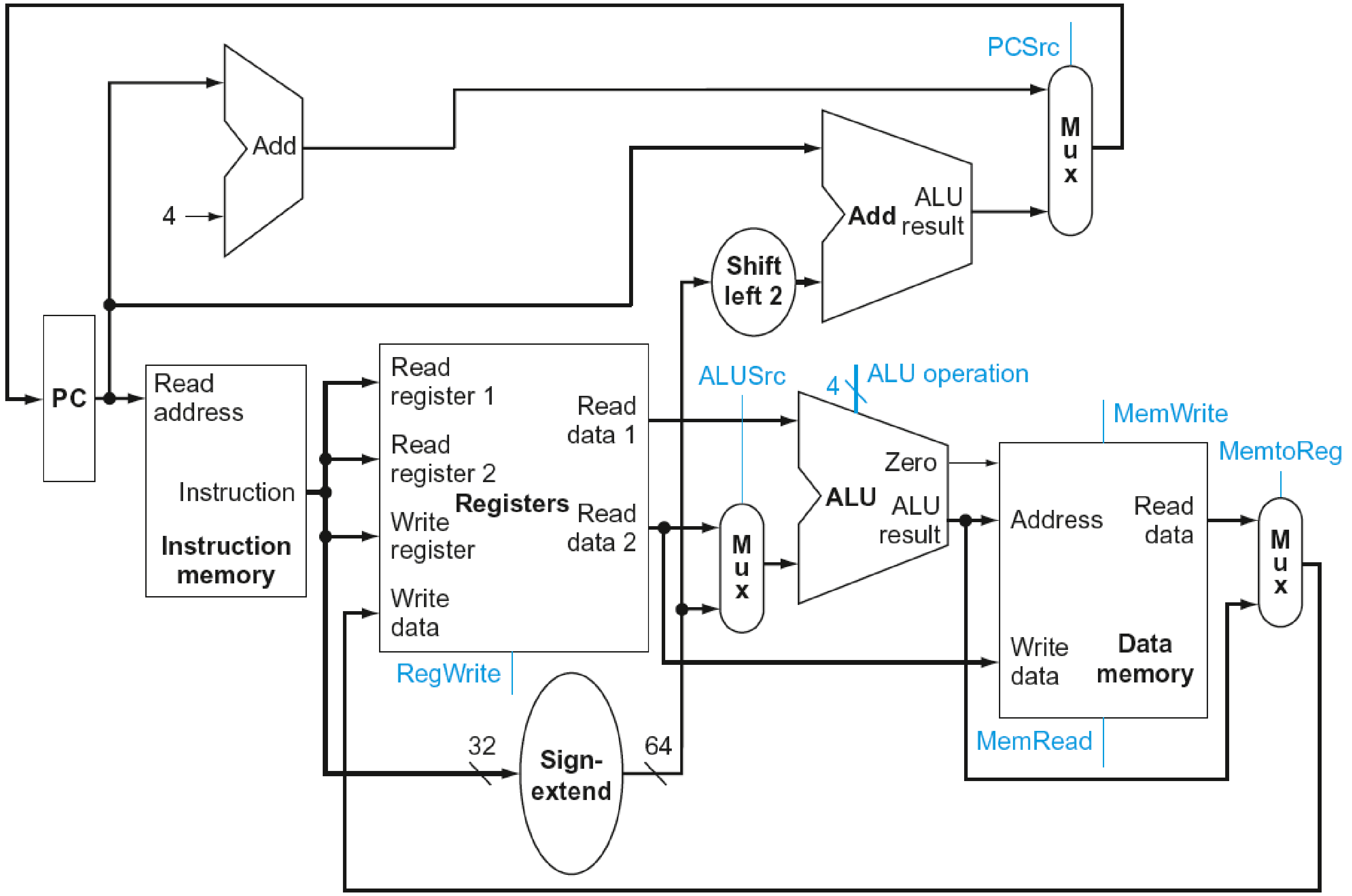
Later
RegWrite → 1

Branch Instructions

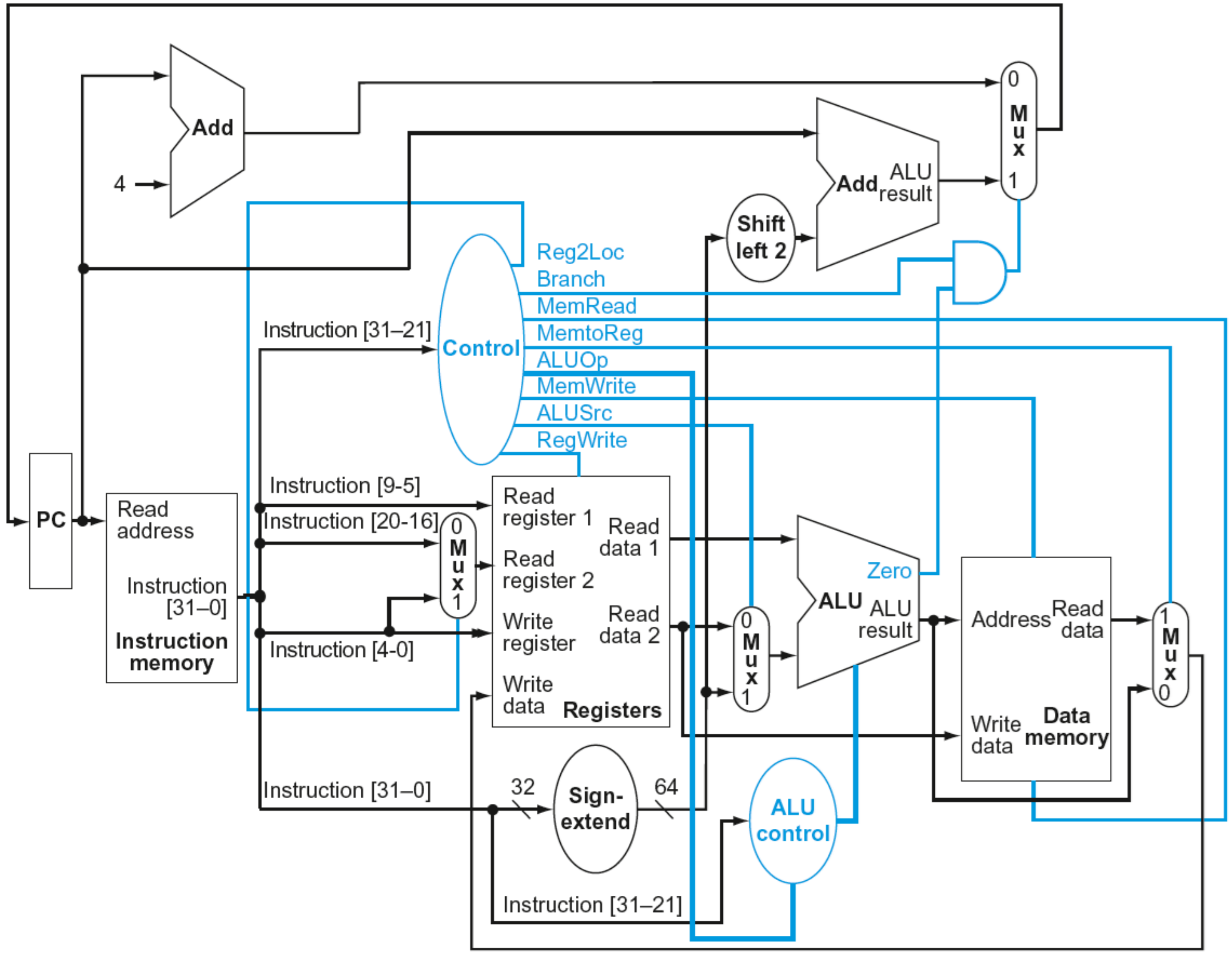
CBZ X0, address(offset)



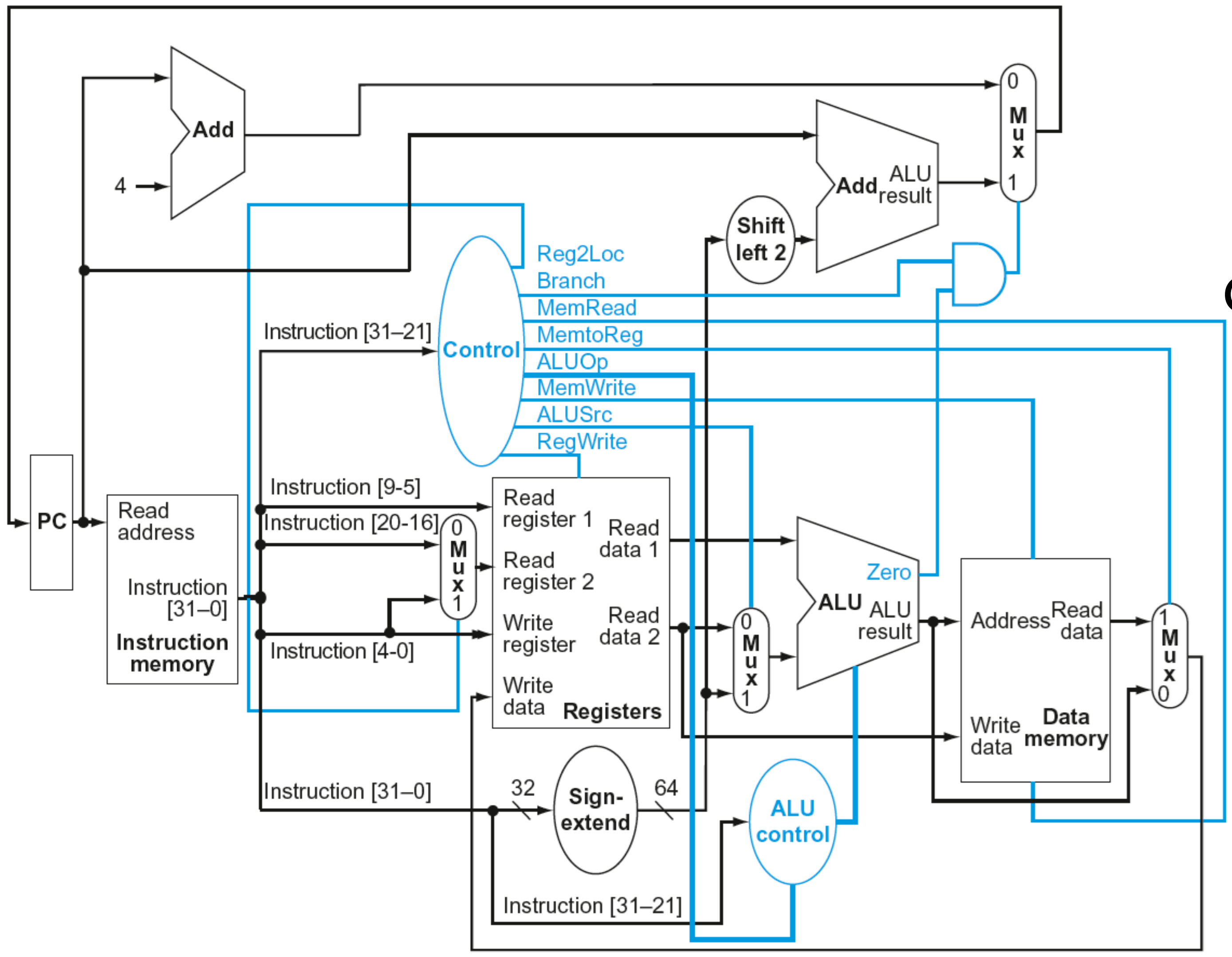
Full Datapath



Datapath With Control



Single Cycle Implementation



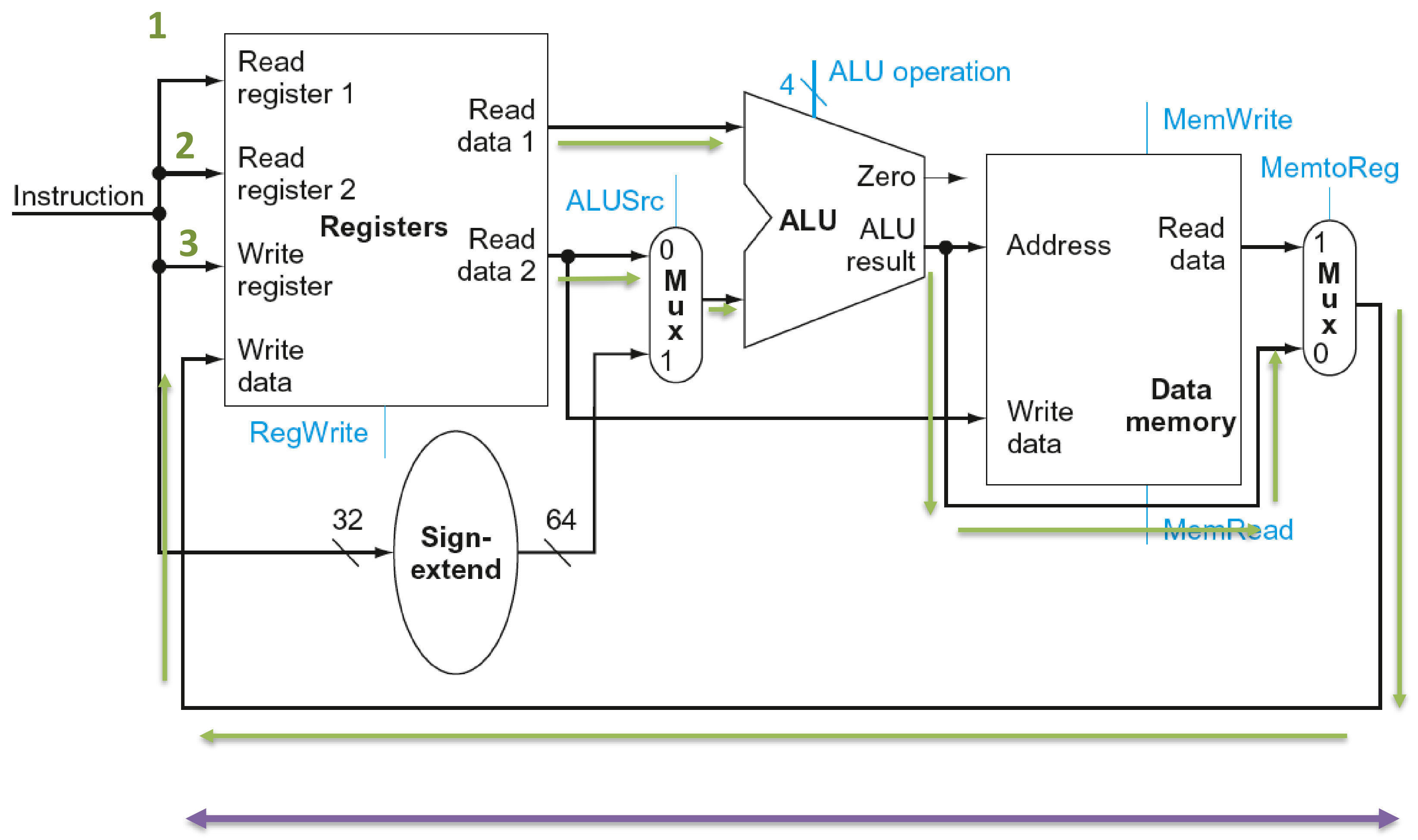
Cycle 1: ADD X9,X20,X21

10001011000$_{two}$	10101$_{two}$	000000$_{two}$	10100$_{two}$	01001$_{two}$

Cycle 2: LDUR X1,[X2, #offset]

1986	**64**	0	22	9
11 bits	**9 bits**	2 bits	5 bits	5 bits

Single Cycle Implementation



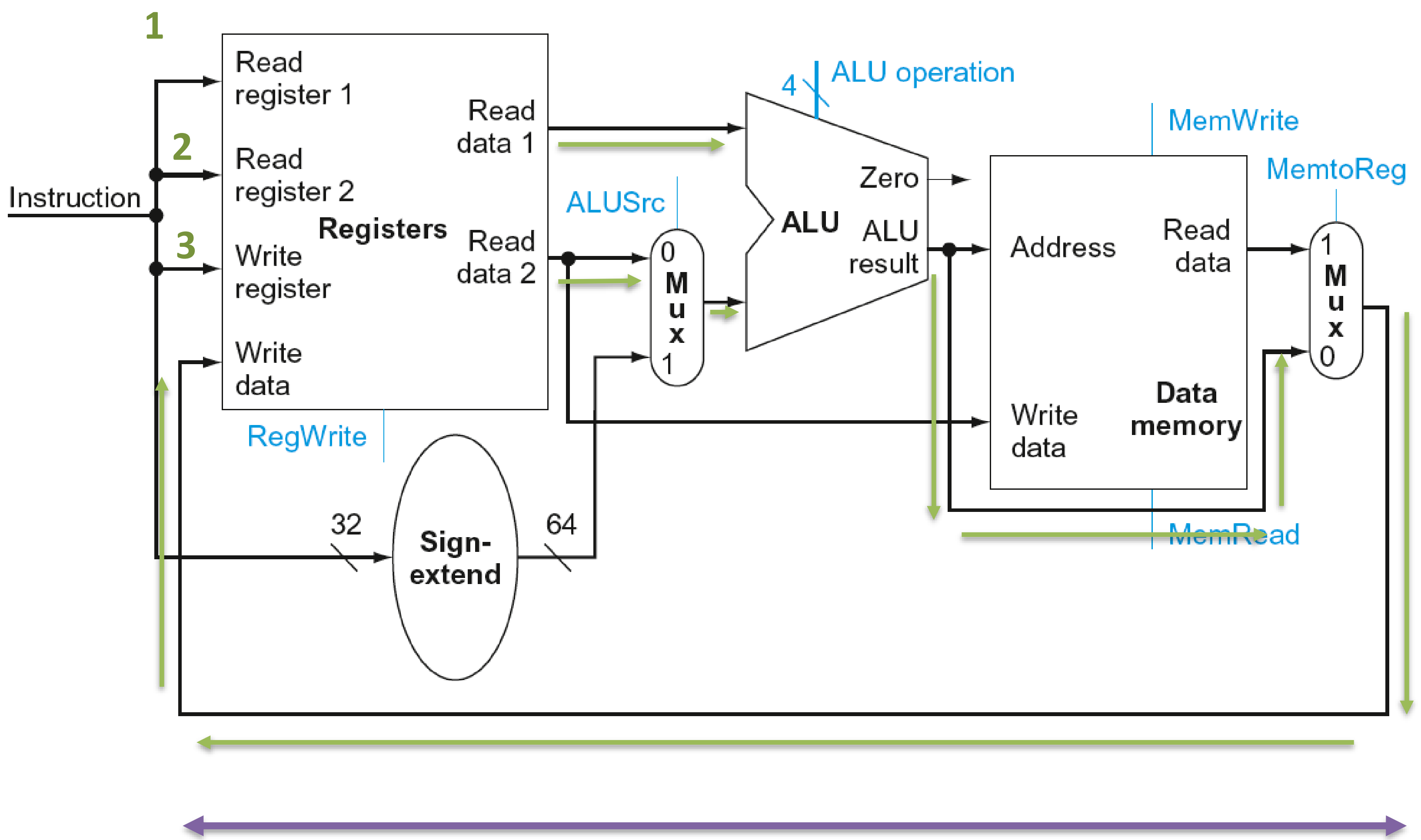
Clock Cycle

R-type
ADD X3, X1, X2

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

Later
RegWrite ➔ 1

Single Cycle Implementation

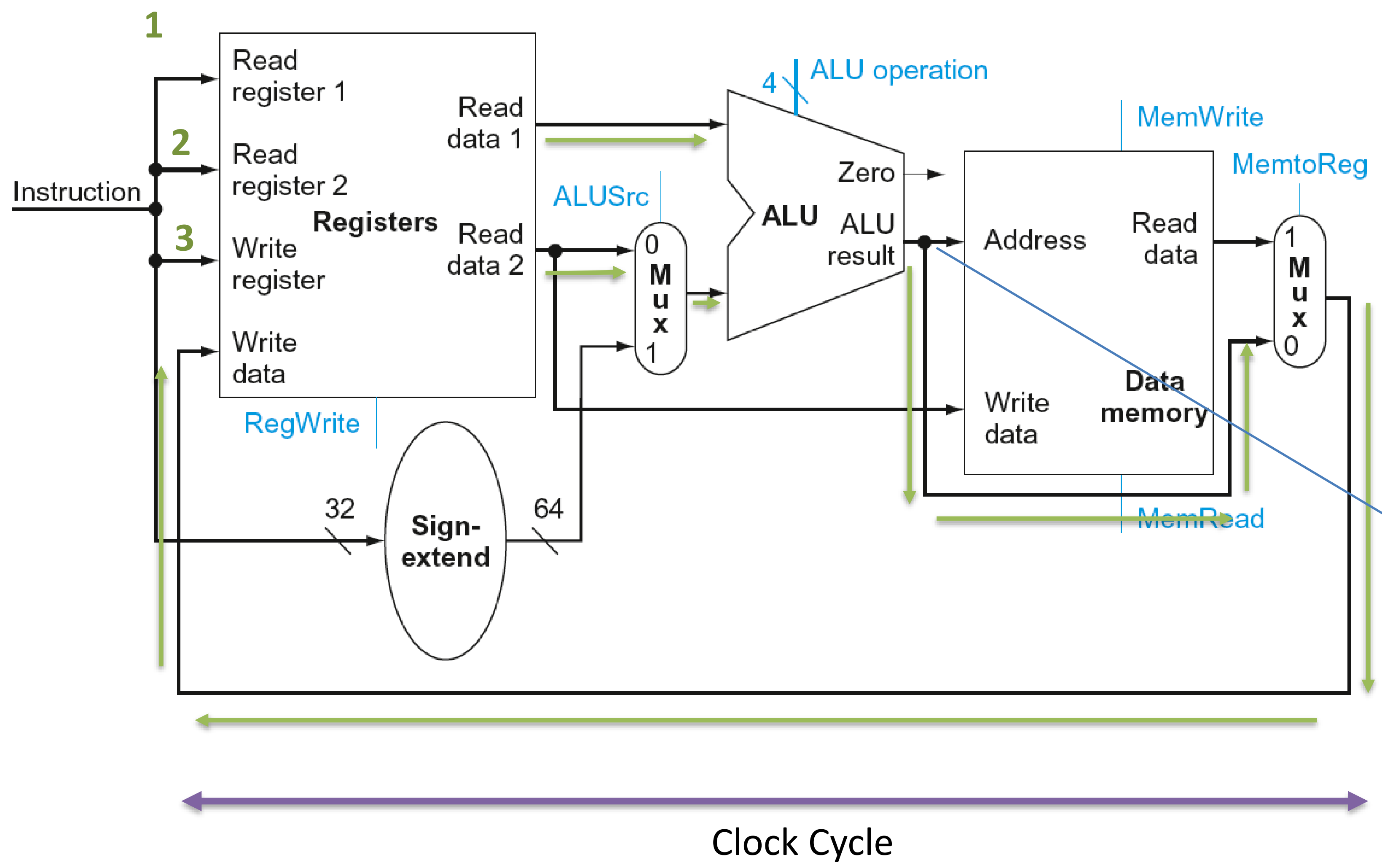


Clock Cycle

R-type
ADD X3, X1, X2

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

Single Cycle Implementation

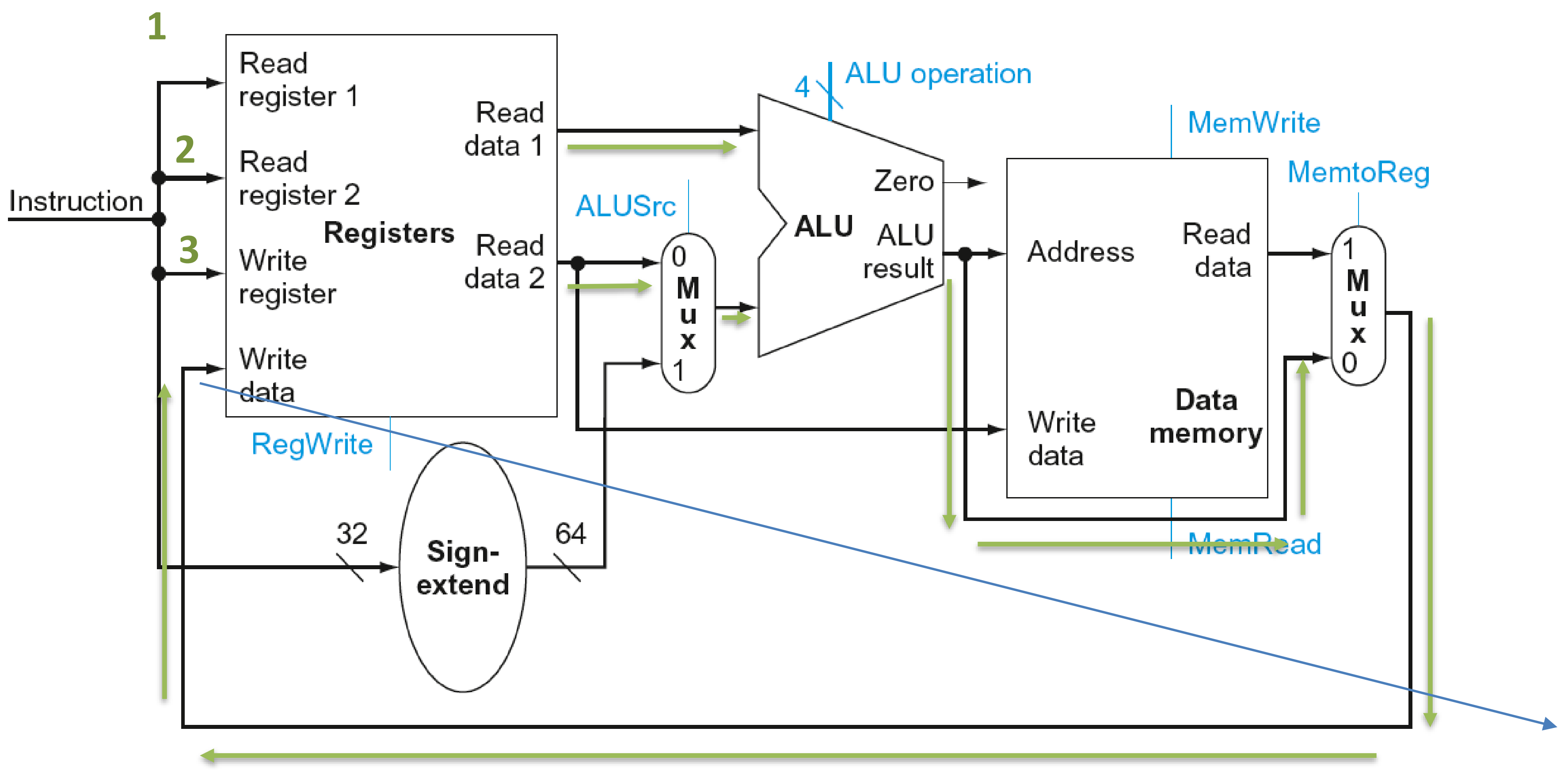


R-type
ADD X3, X1, X2

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

1. Data is not available until after ALU

Single Cycle Implementation

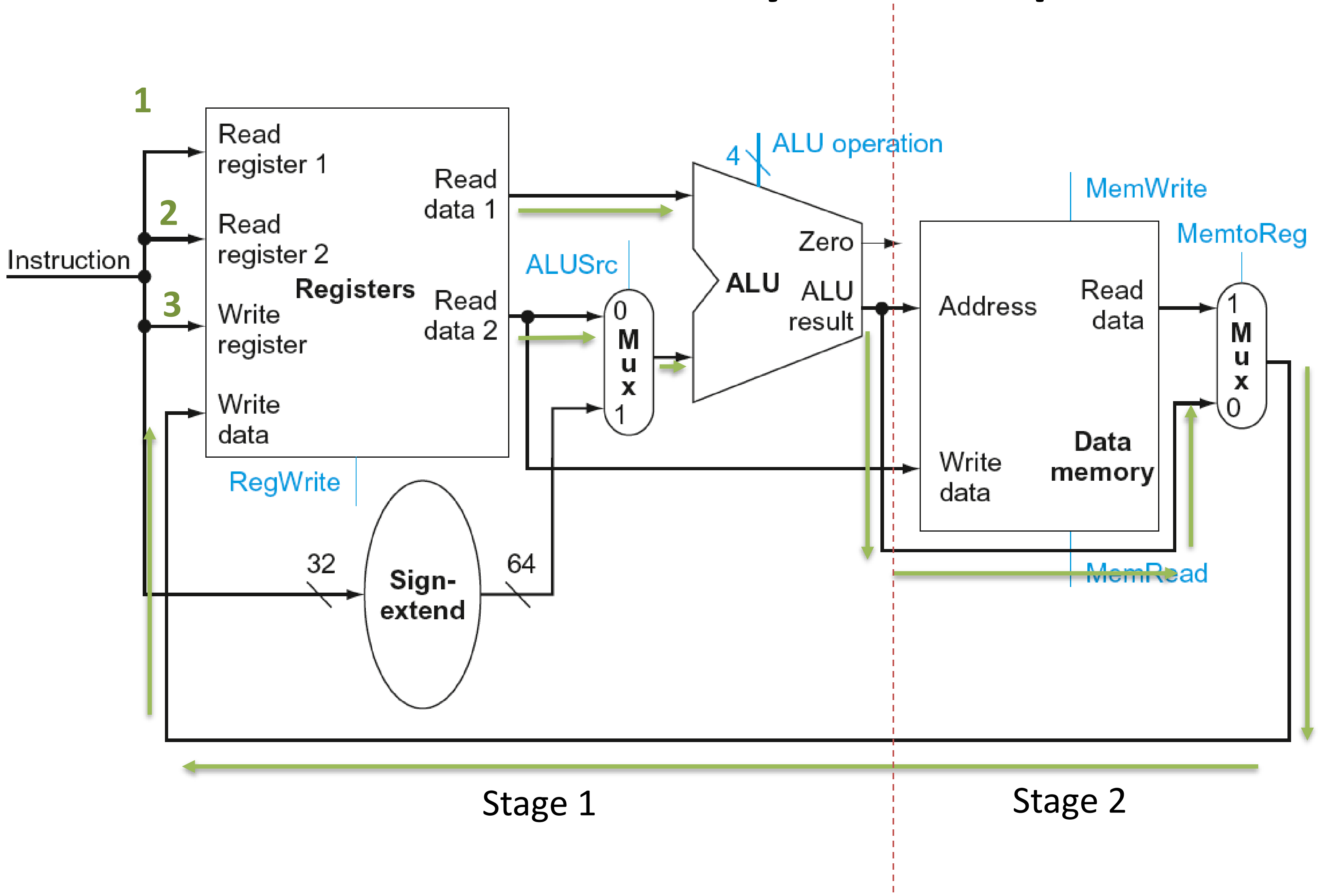


R-type
ADD X3, X1, X2

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

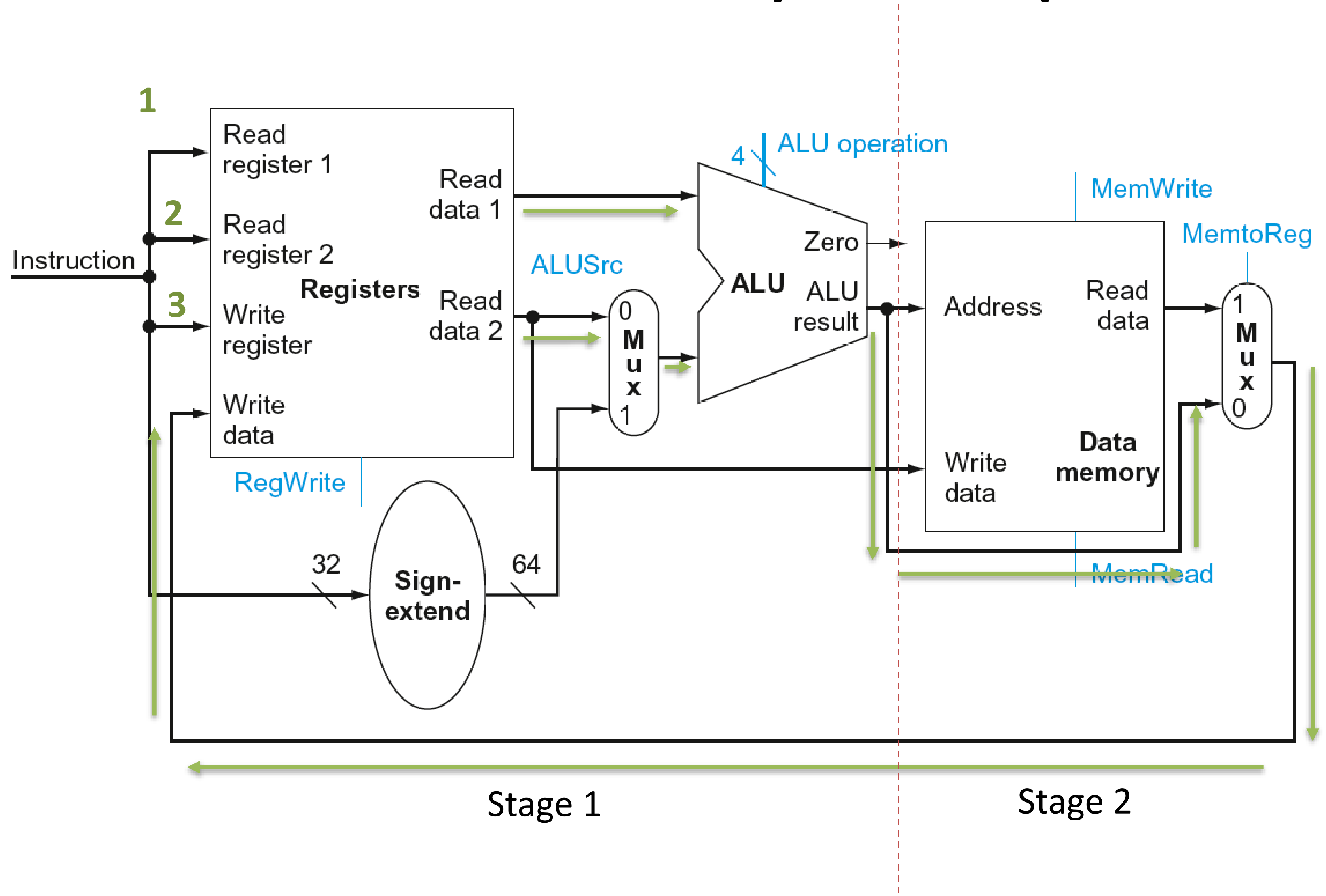
Multi-Cycle Implementation



R-type
ADD X3, X1, X2

Break into **stages**

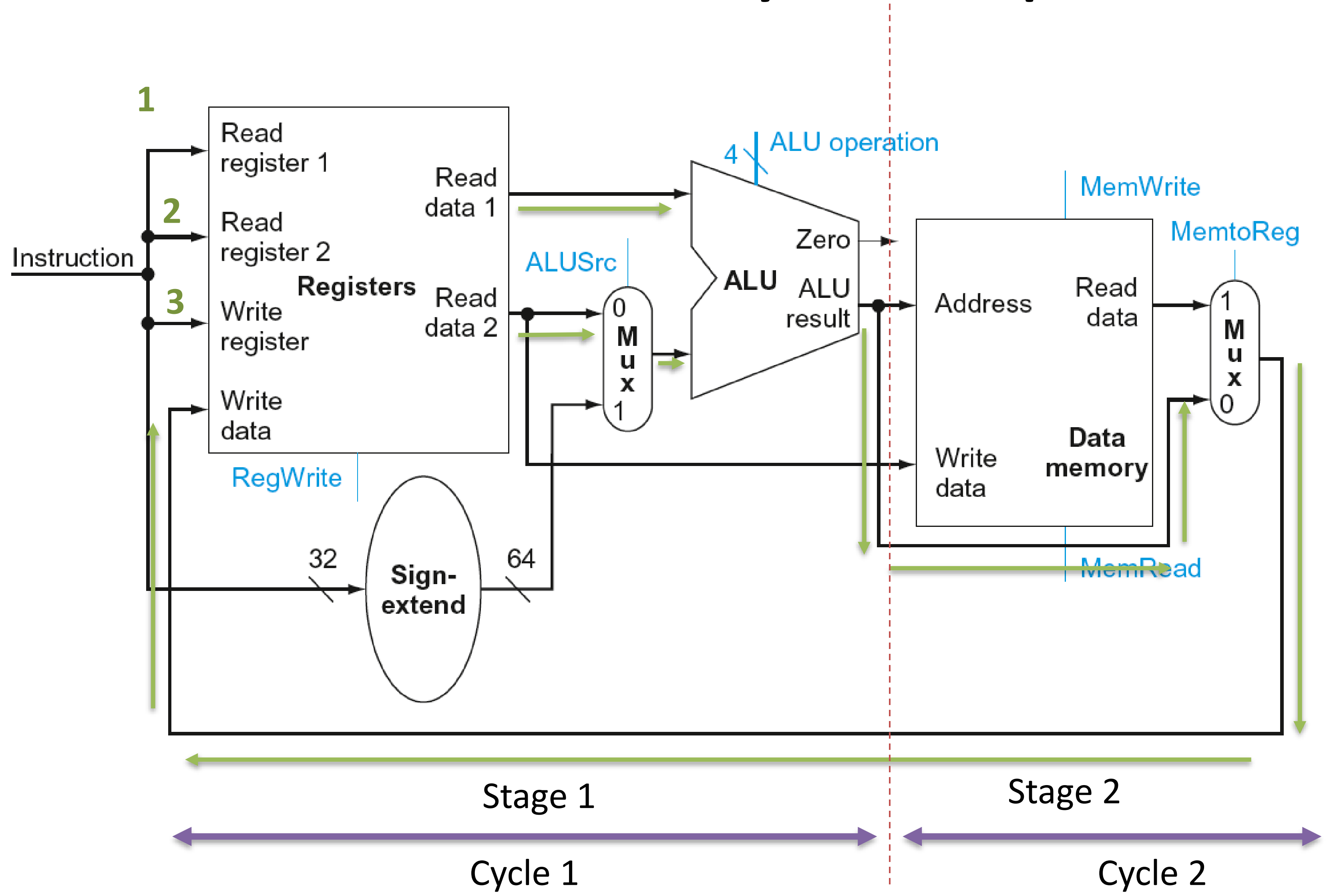
Multi-Cycle Implementation



R-type
ADD X3, X1, X2

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

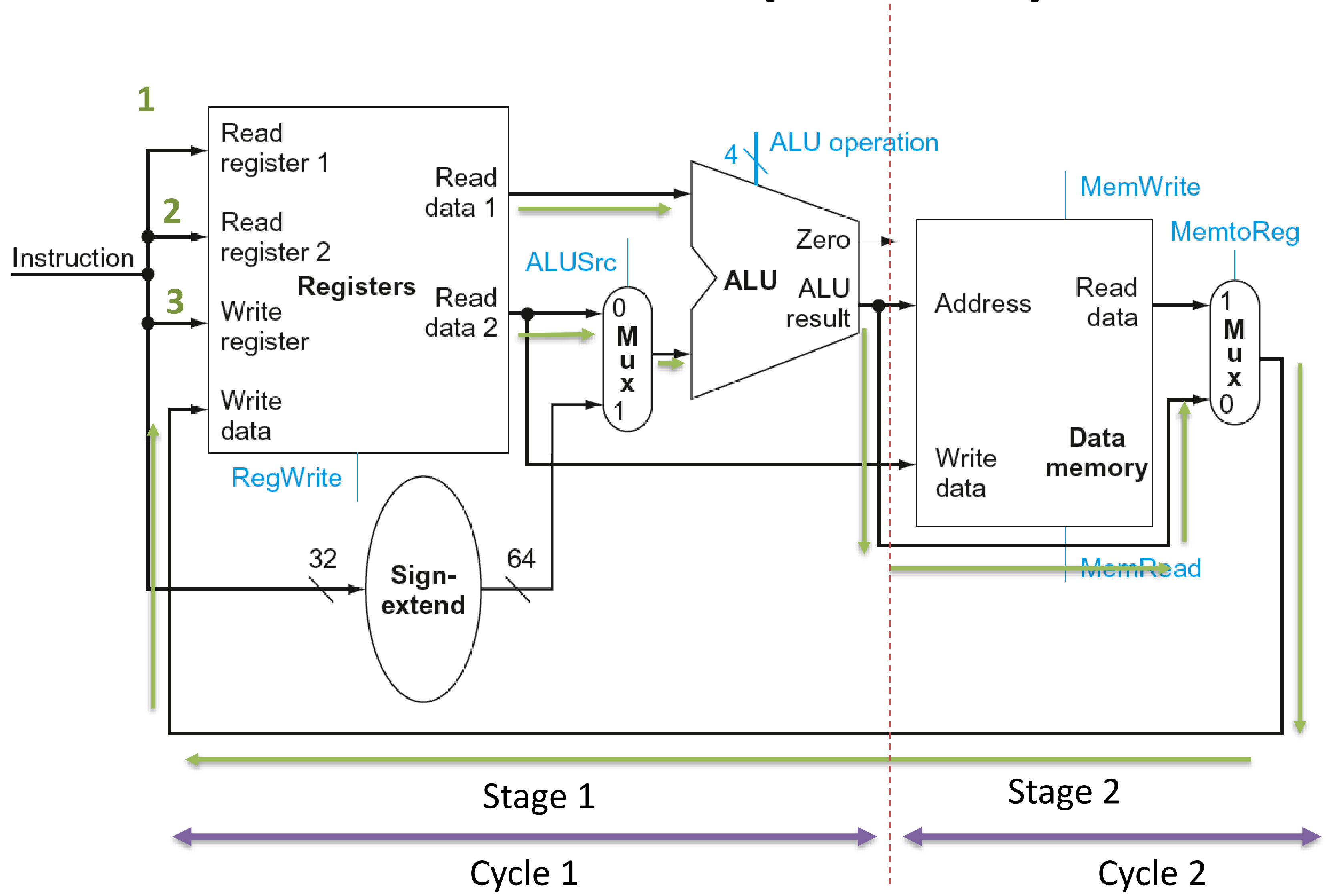
Multi-cycle Implementation



R-type
ADD X3, X1, X2

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

Multi-Cycle Implementation

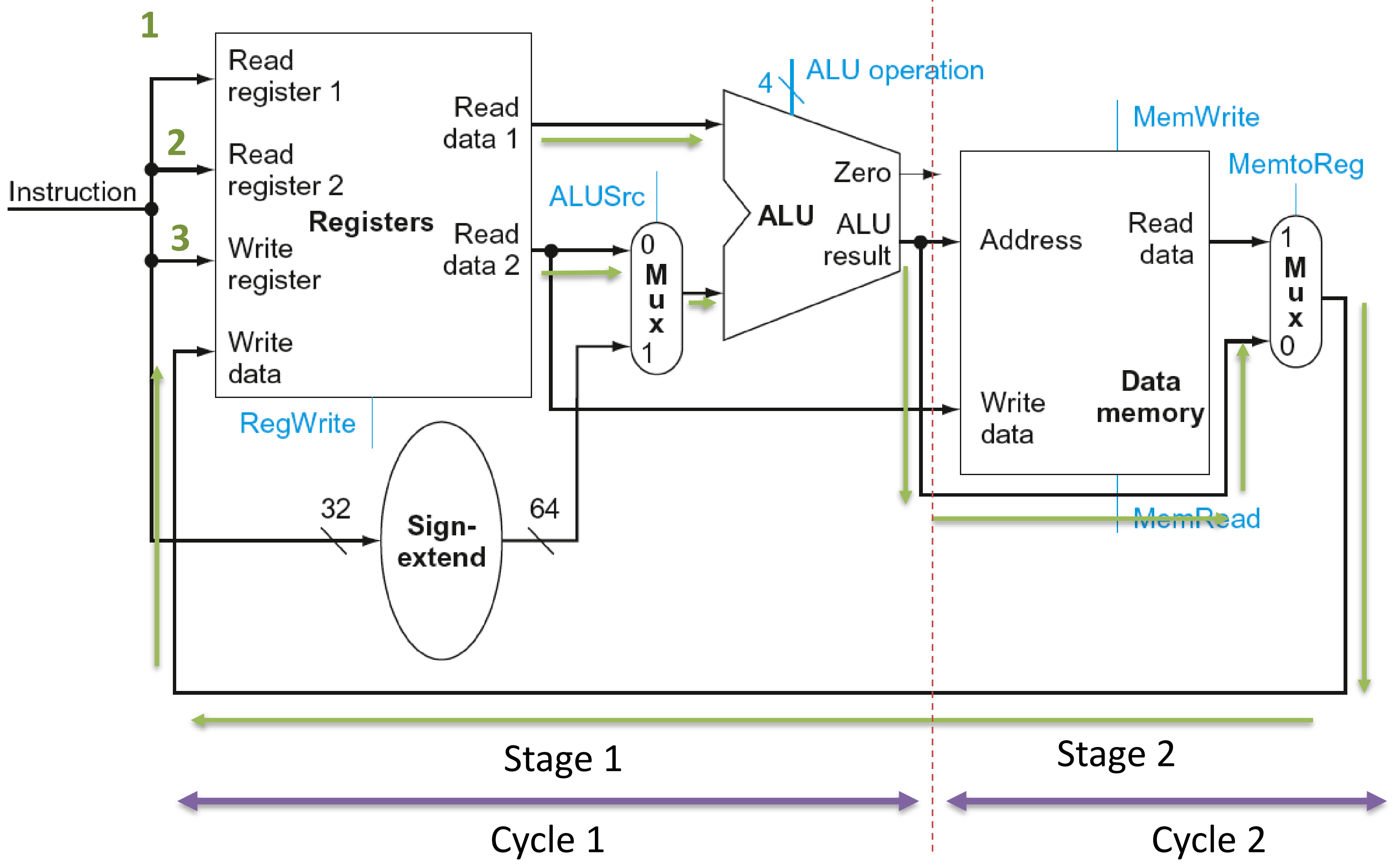


R-type
ADD X3, X1, X2

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

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

Multi-Cycle Implementation



R-type
ADD X3, X1, X2

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

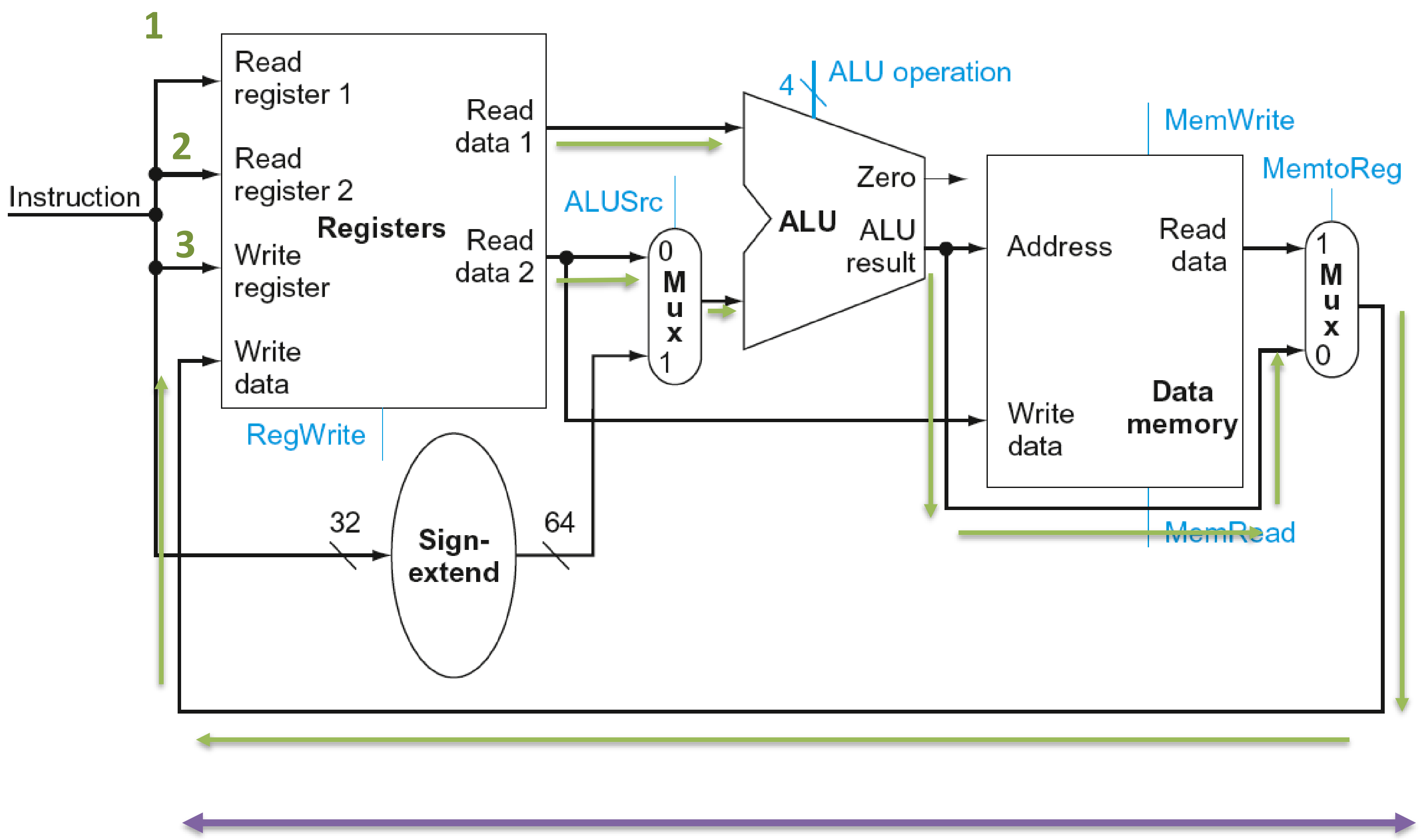
Stage 1 (cycle 1):
RegWrite ➔ 0
ALUSrc ➔ 0
ALU operation ➔ 0010

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

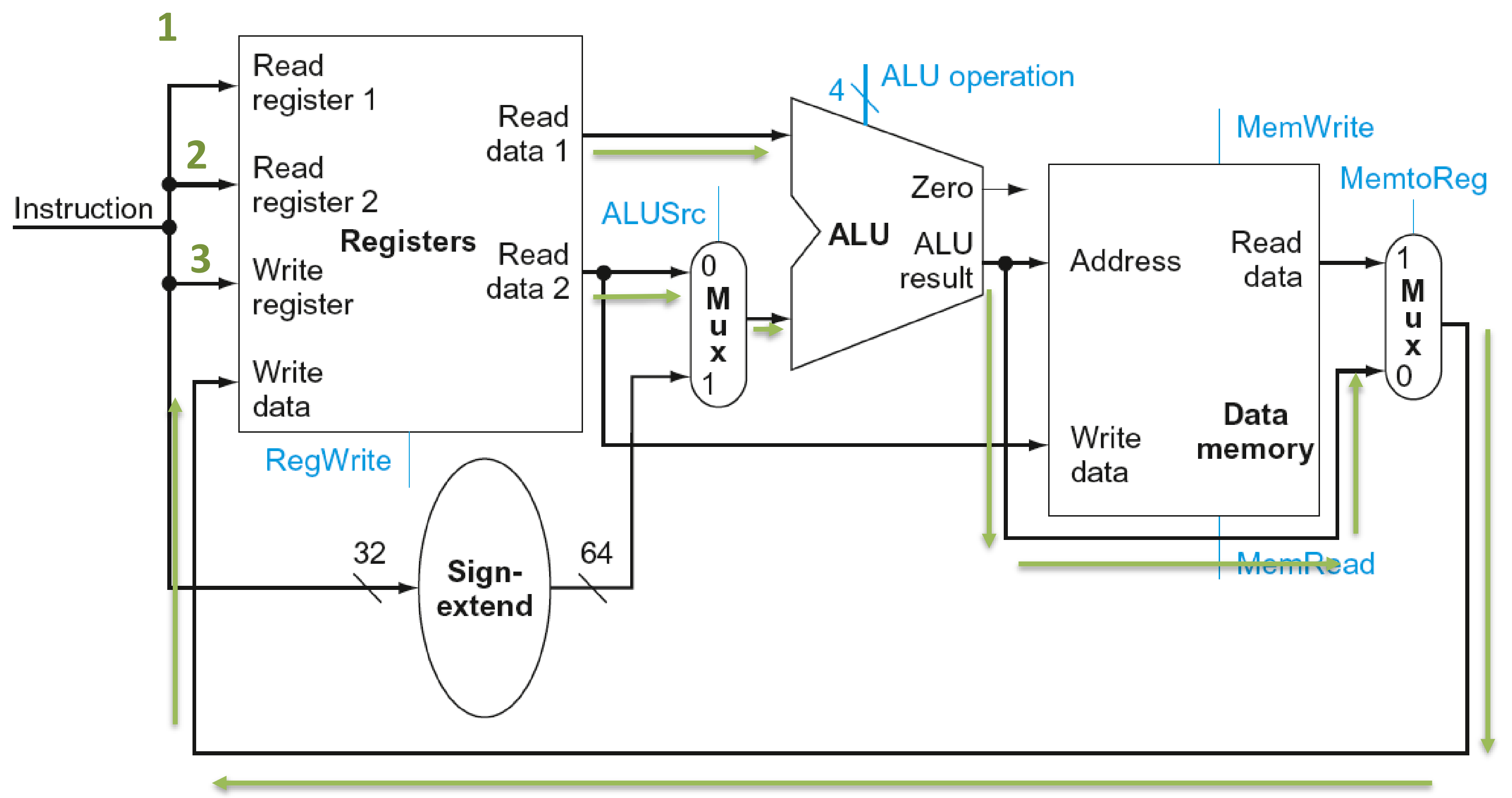


Cycle time (500 ps)

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

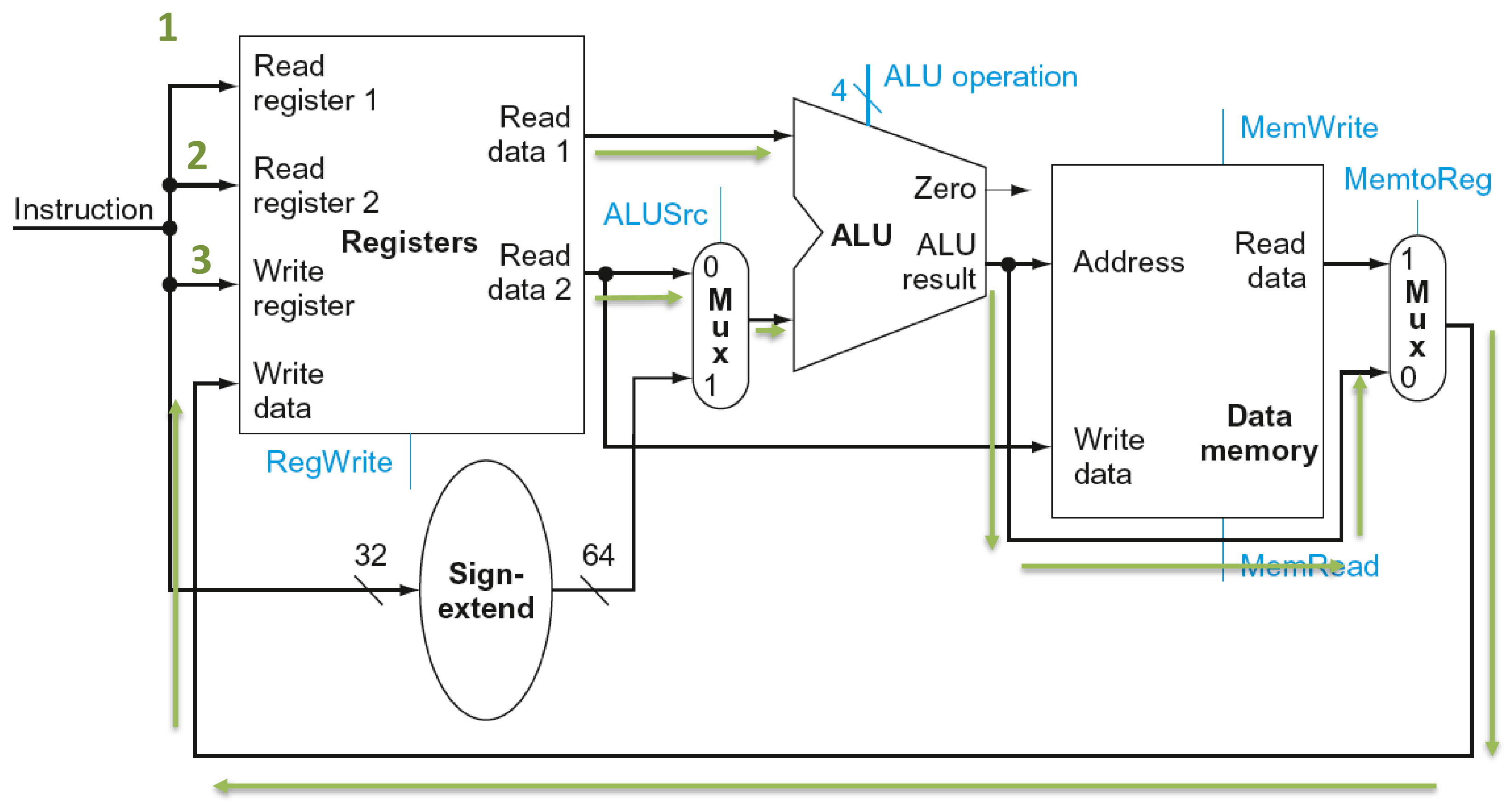


R-type
ADD X3, X1, X2

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 clock cycle time?

Multi-Cycle Implementation

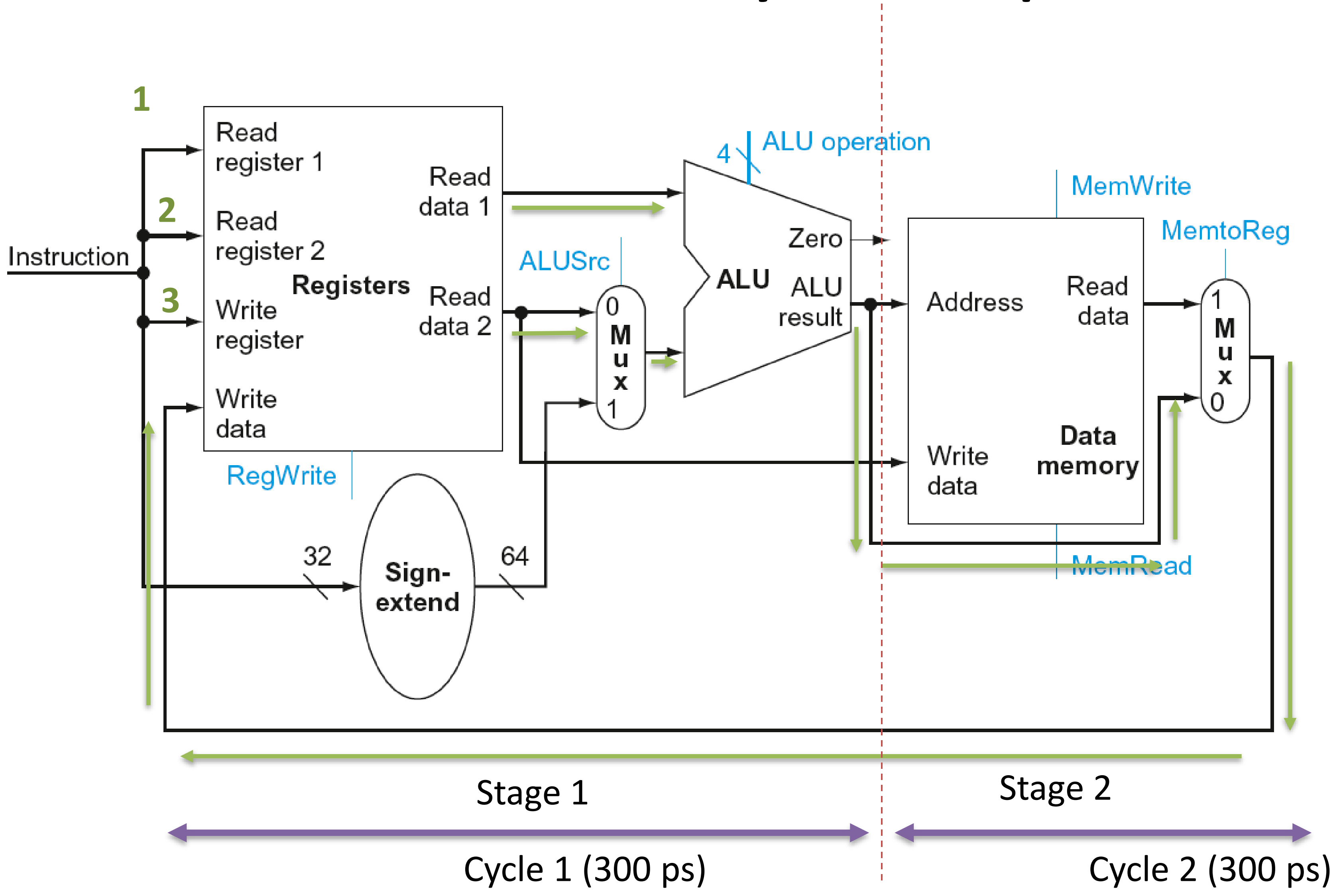


R-type
ADD X3, X1, X2

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?

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						
3						
4						
5						

Pipelining

- 5 Add instructions

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

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						
5						

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

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**

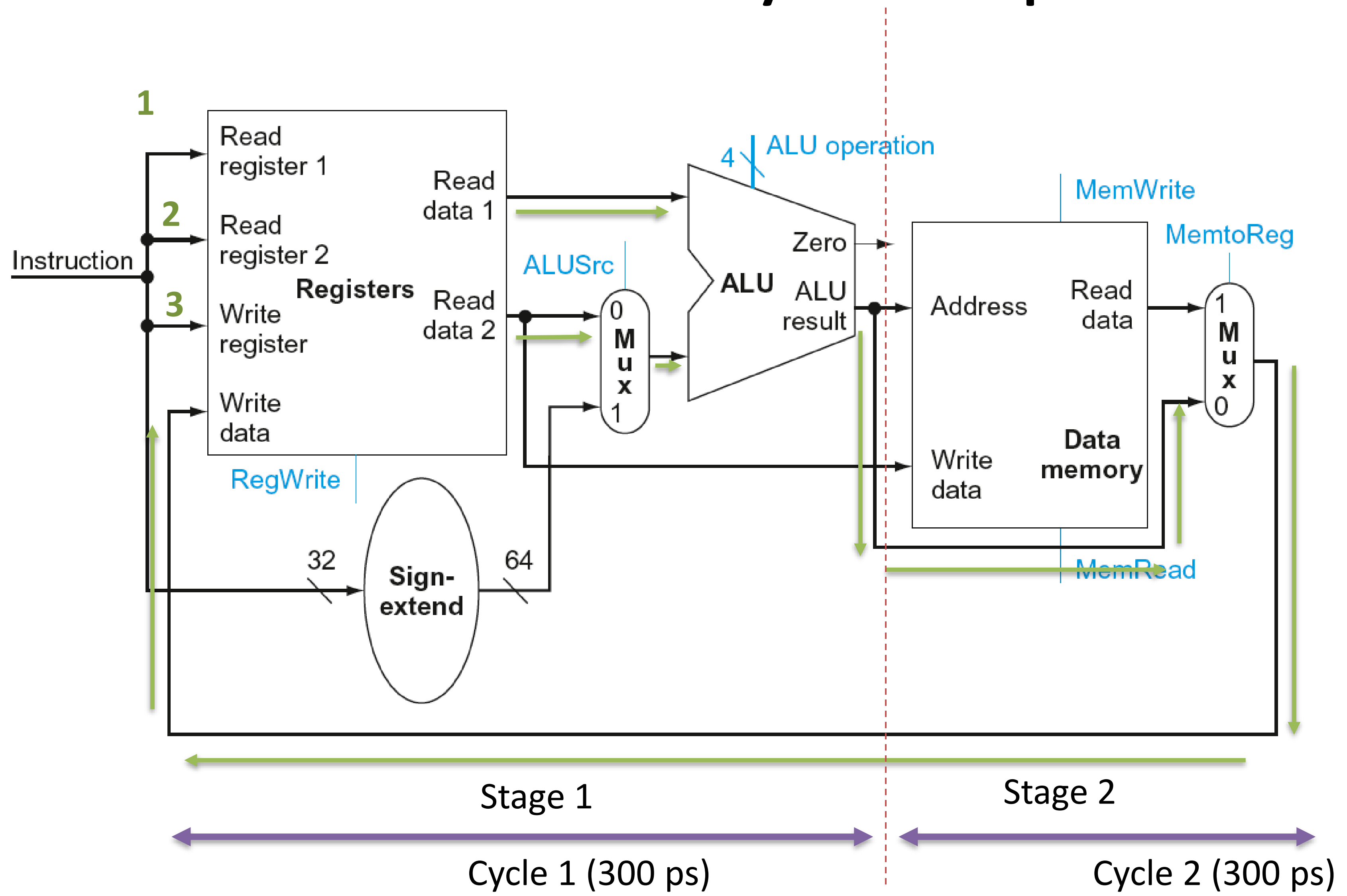
Time to execute 5 Add instructions

Single cycle implementation = 5 * 500 = 2500 ps

Multi cycle implementation = 5 * 600 = 3000 ps

Pipelining = 1800 ps

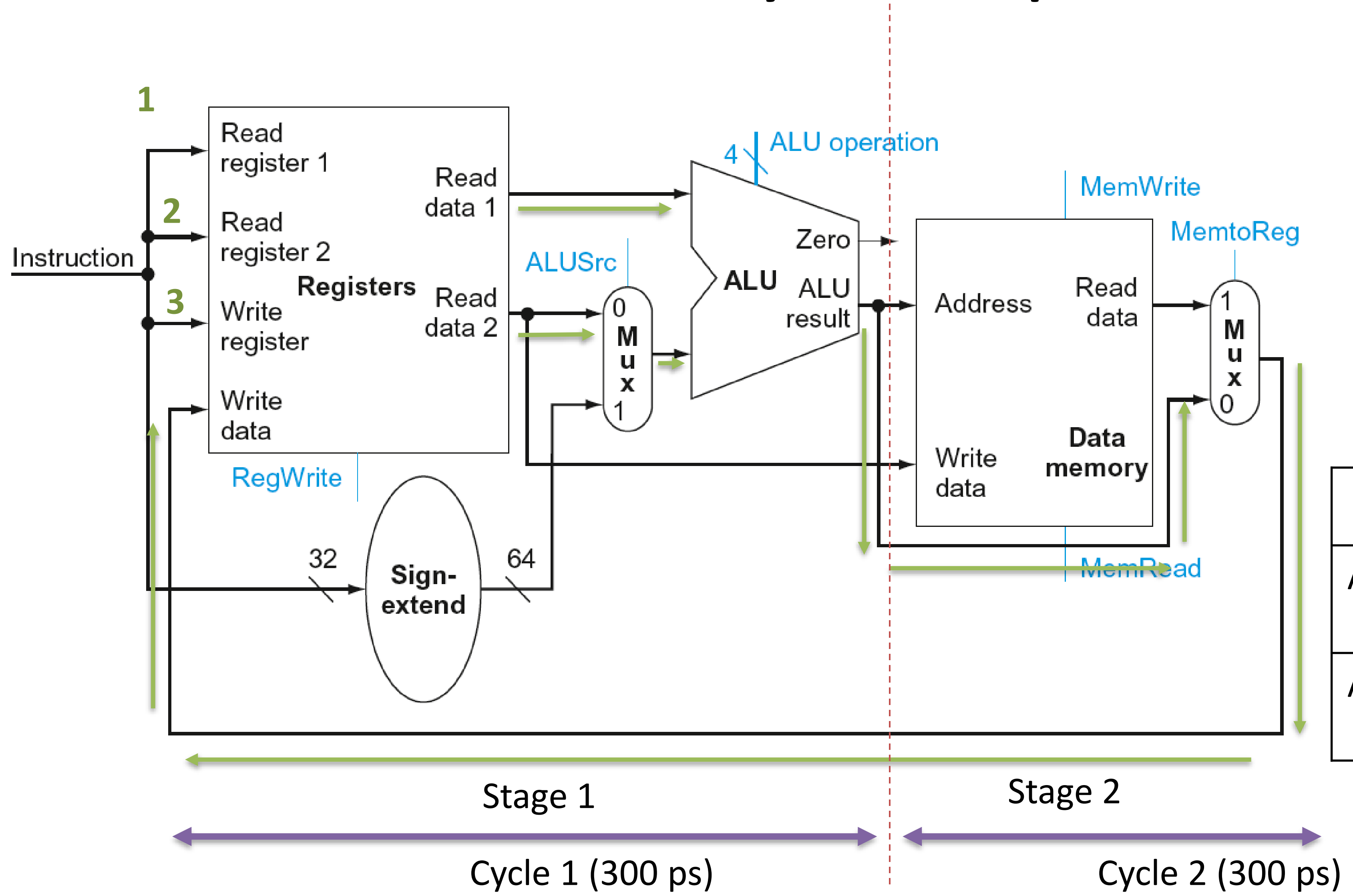
Multi-Cycle Implementation



Not possible to implement with ADD

ADD **X3**, X1, X2
ADD **X4**, X3, X5

Multi-Cycle Implementation



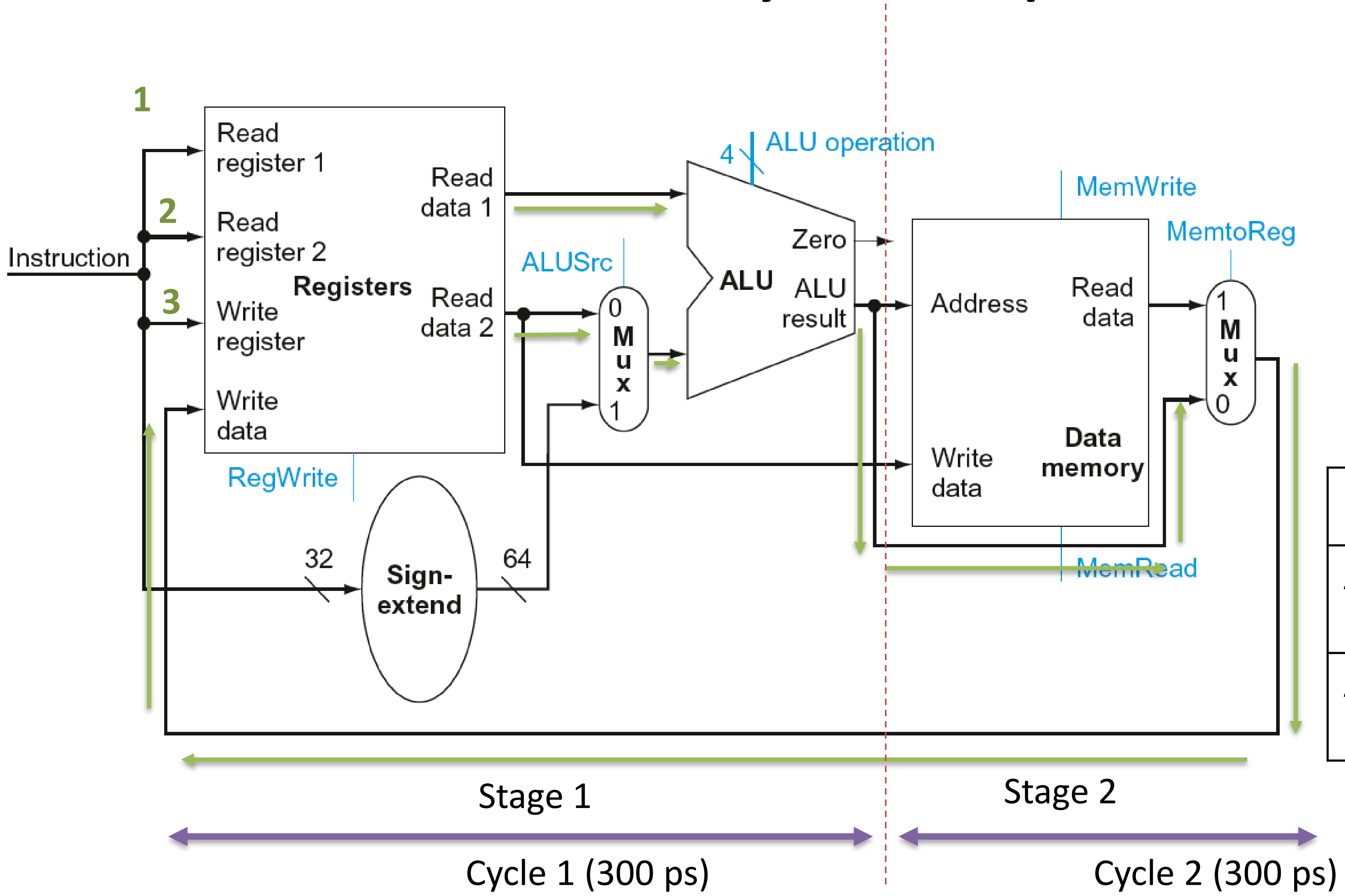
R-type
ADD X3, X1, X2

Not possible to implement with ADD

Value of write register

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

Multi-Cycle Implementation



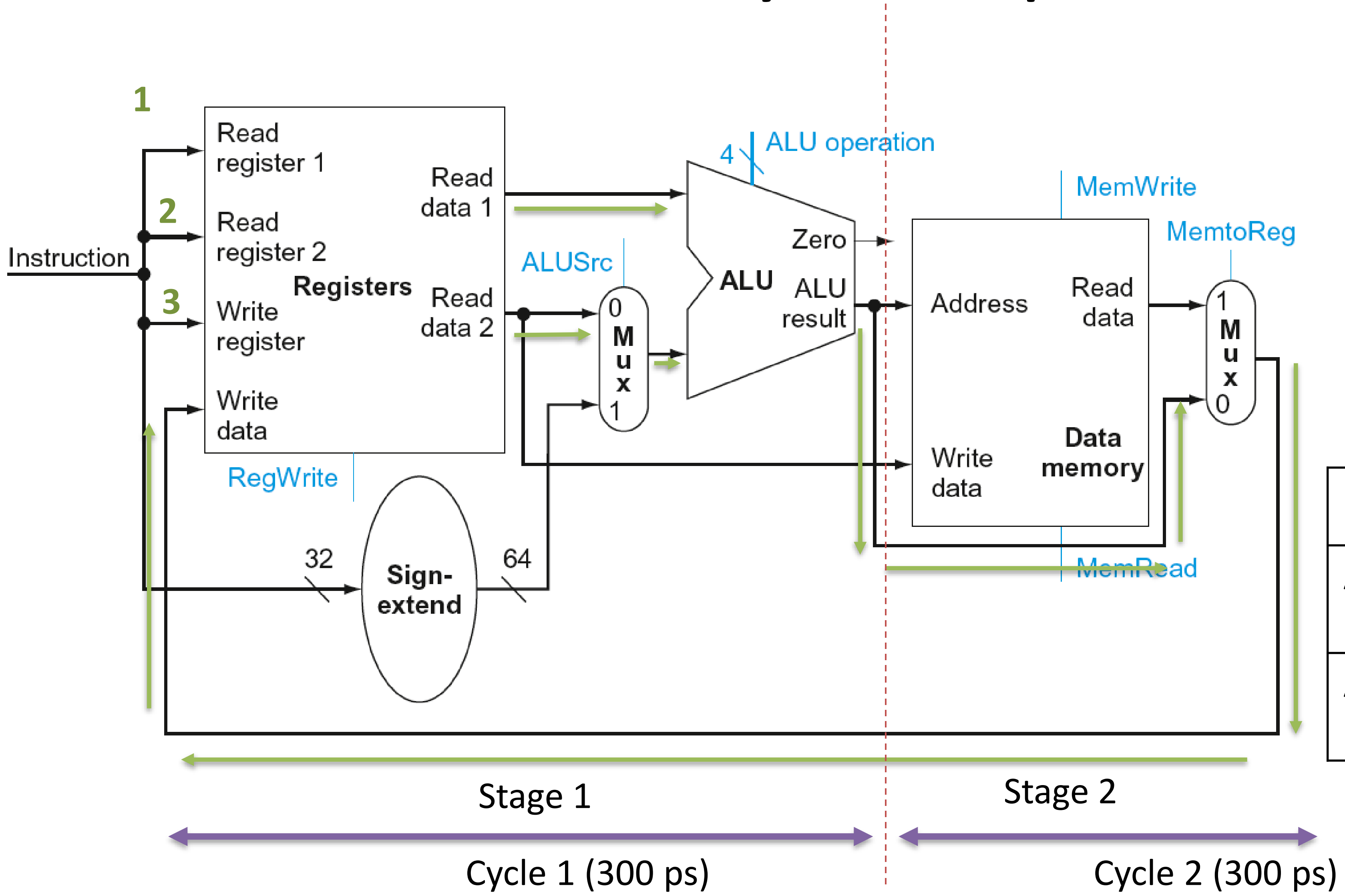
R-type
ADD X3, X1, X2

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	

Multi-Cycle Implementation



R-type
ADD X3, X1, X2

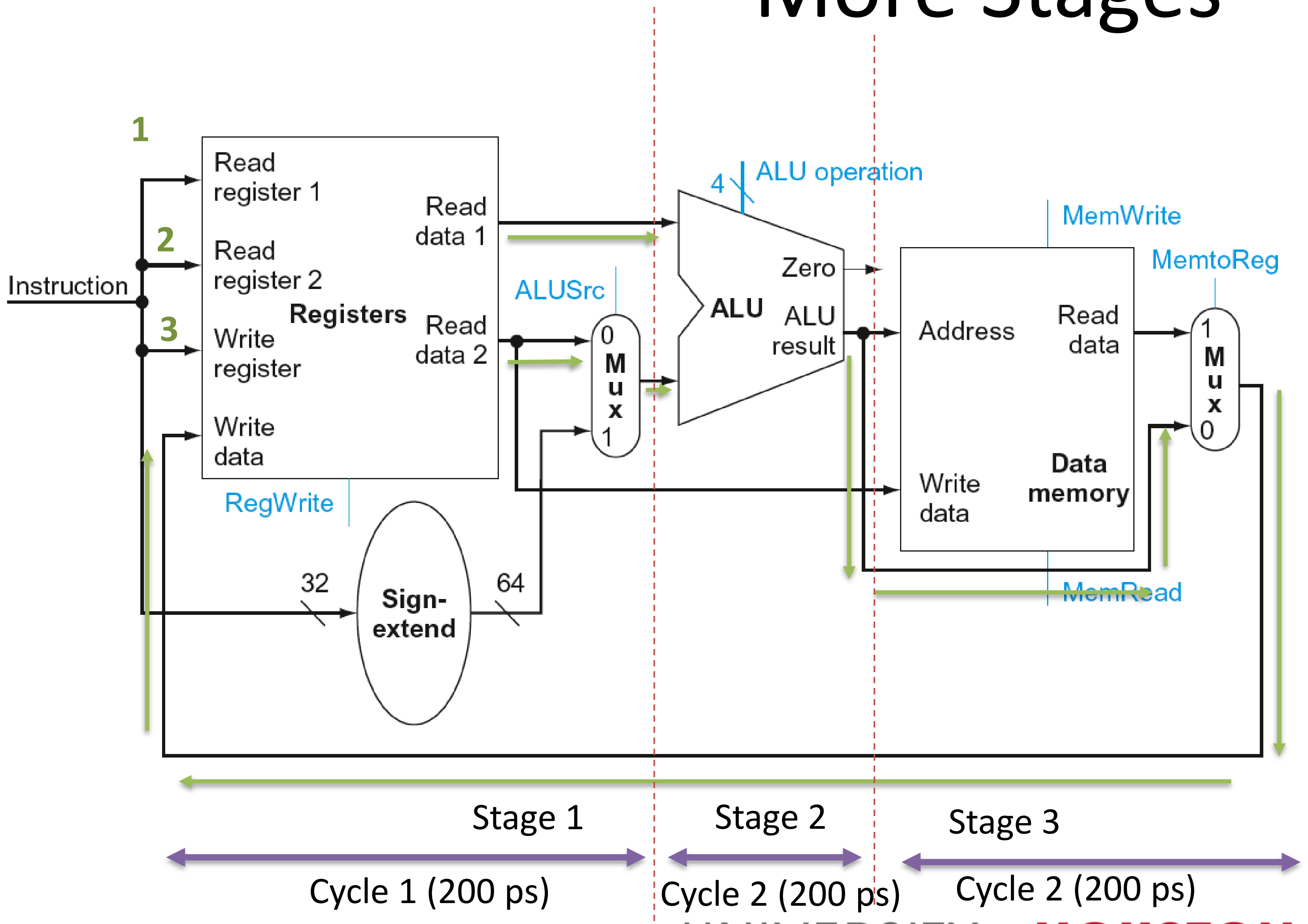
Not possible to implement with ADD

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**

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

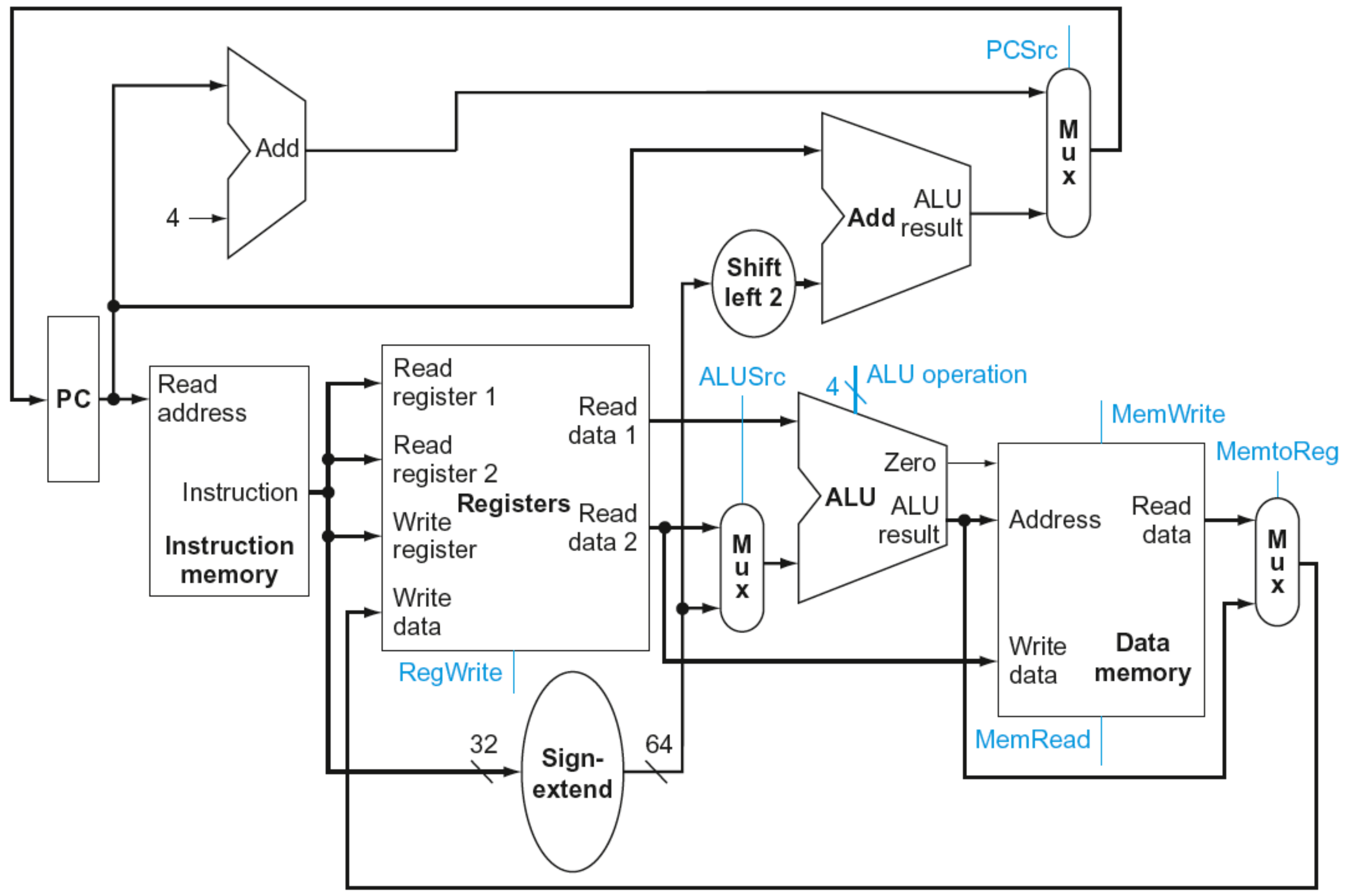
Multi cycle implementation = 5 * 600 = 3000 ps

Pipelining (2 stages) = 1800 ps

Pipelining (3 stages) = 1400 ps

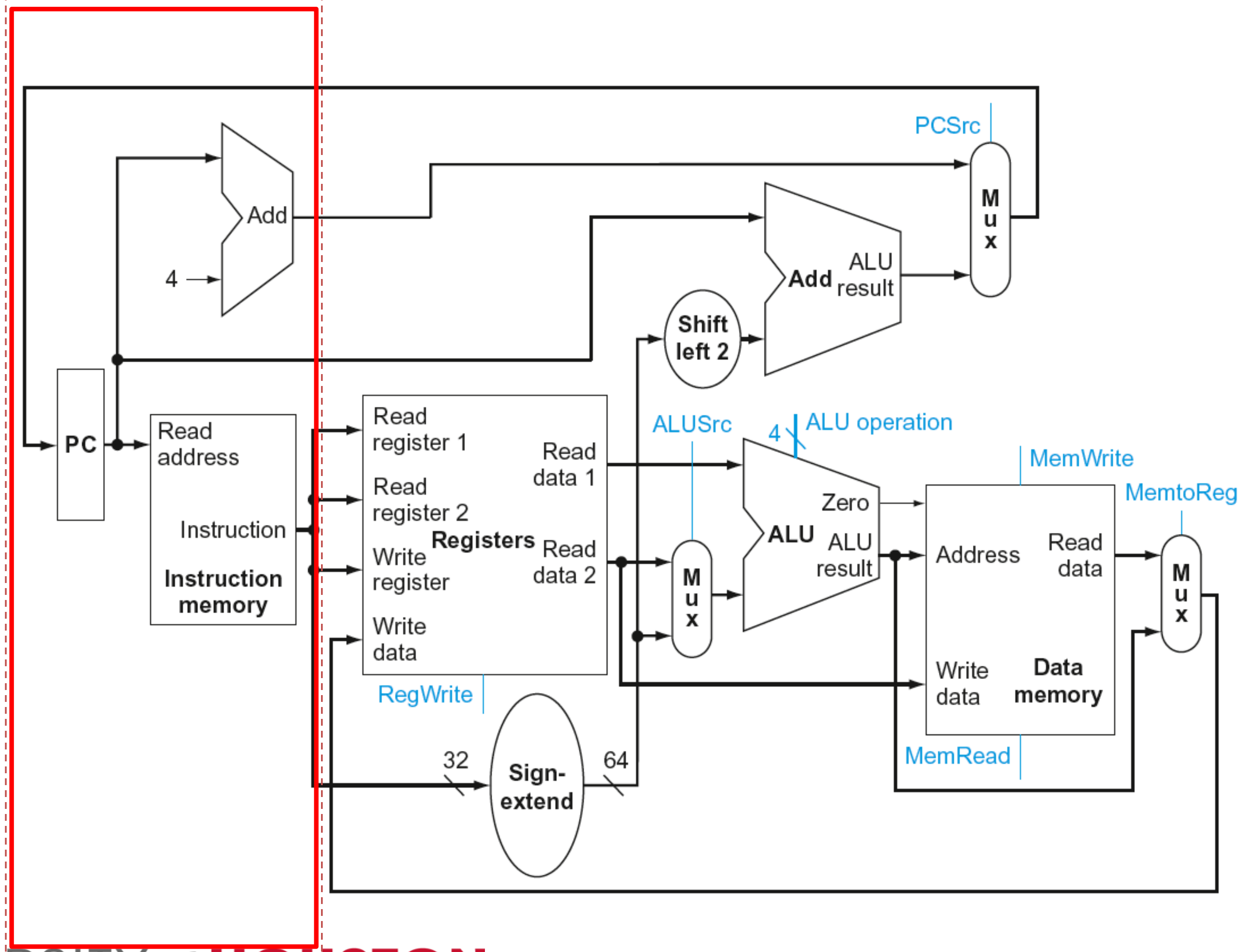
Speed up is approximately equal to the number of stages.

Stages in LEGv8



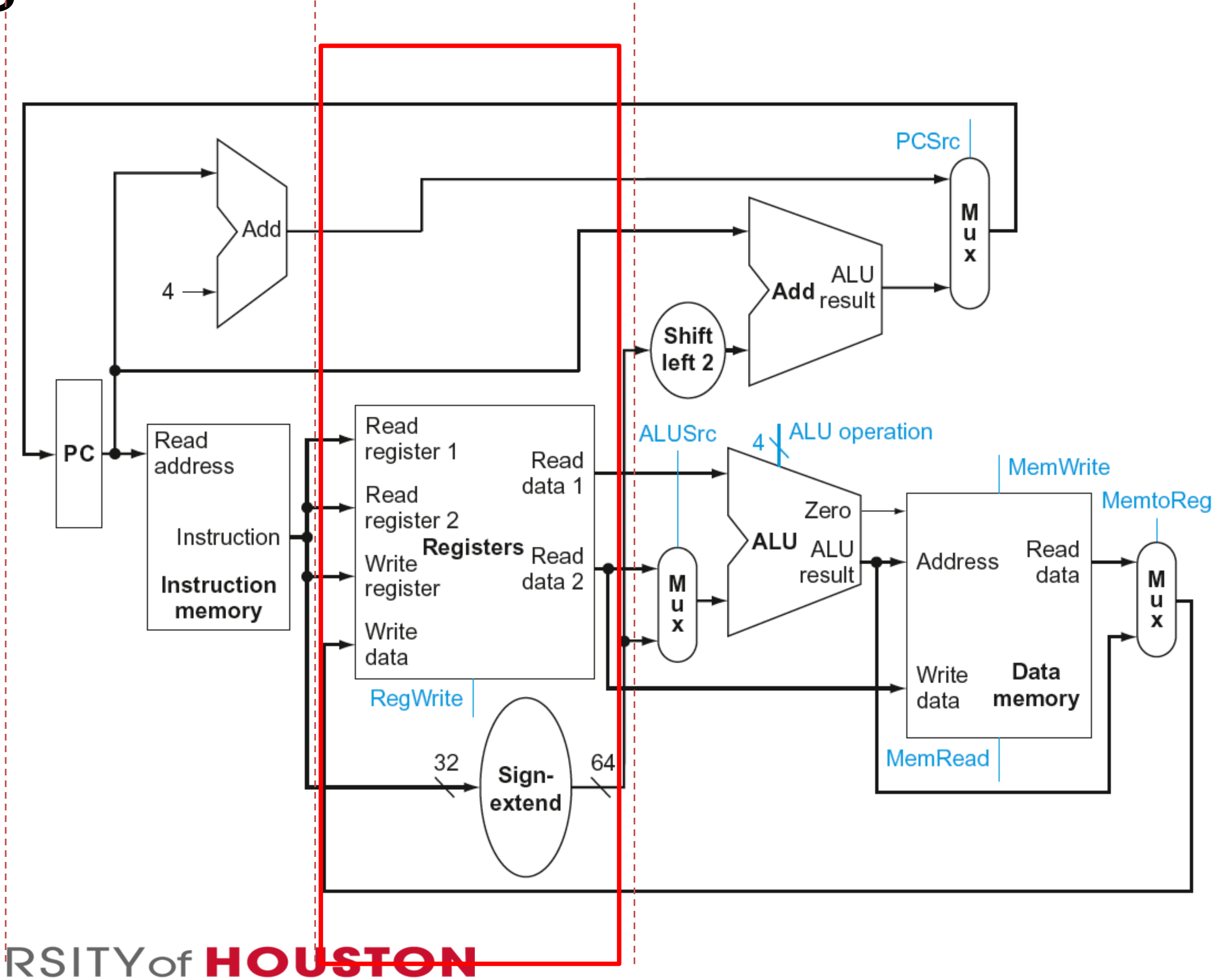
Stages in LEGv8

1. IF: Instruction fetch from memory
    1. Use PC to fetch instruction
    2. Update PC



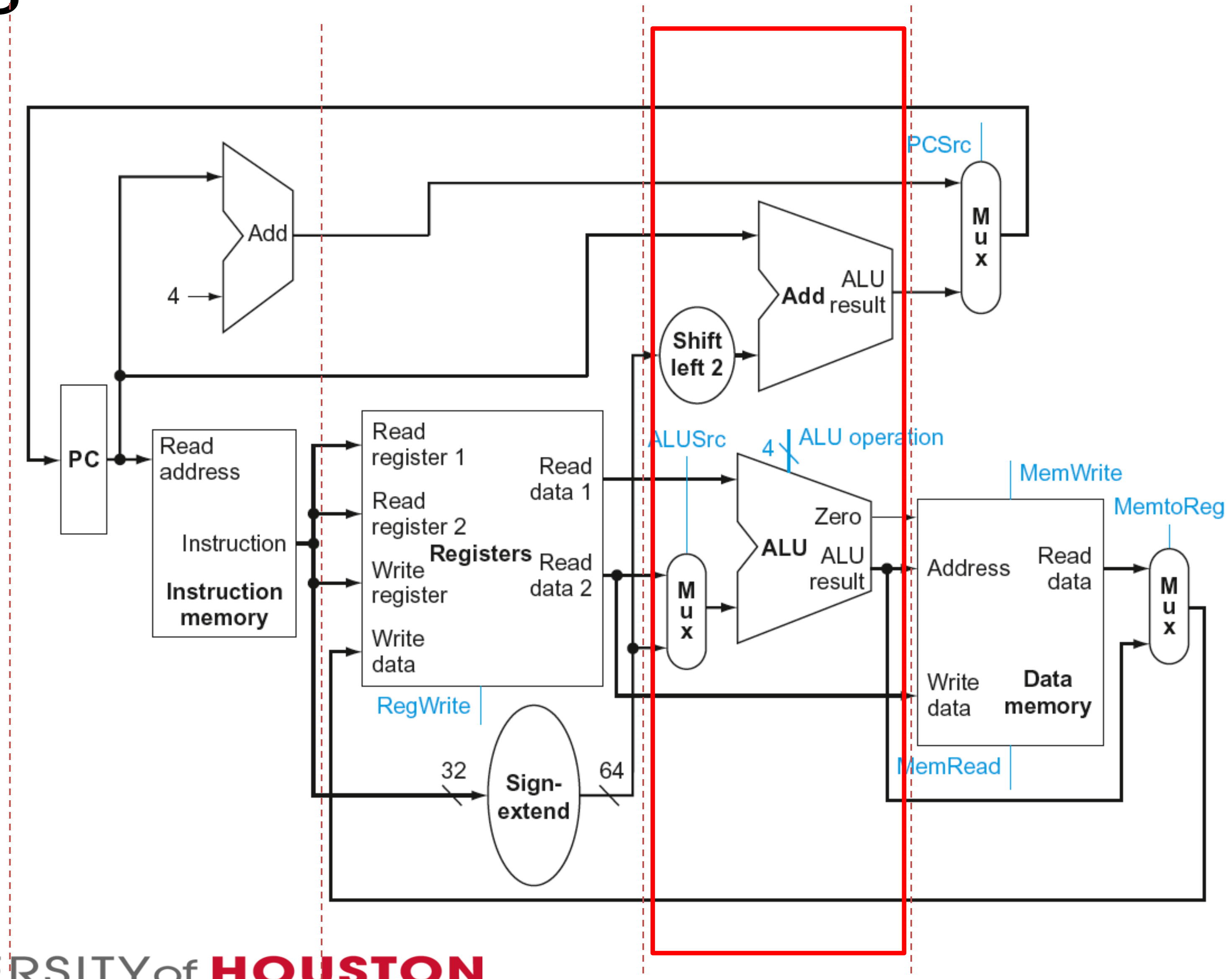
Stages in LEGv8

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
   1. Pass opcode to control
   2. Read registers



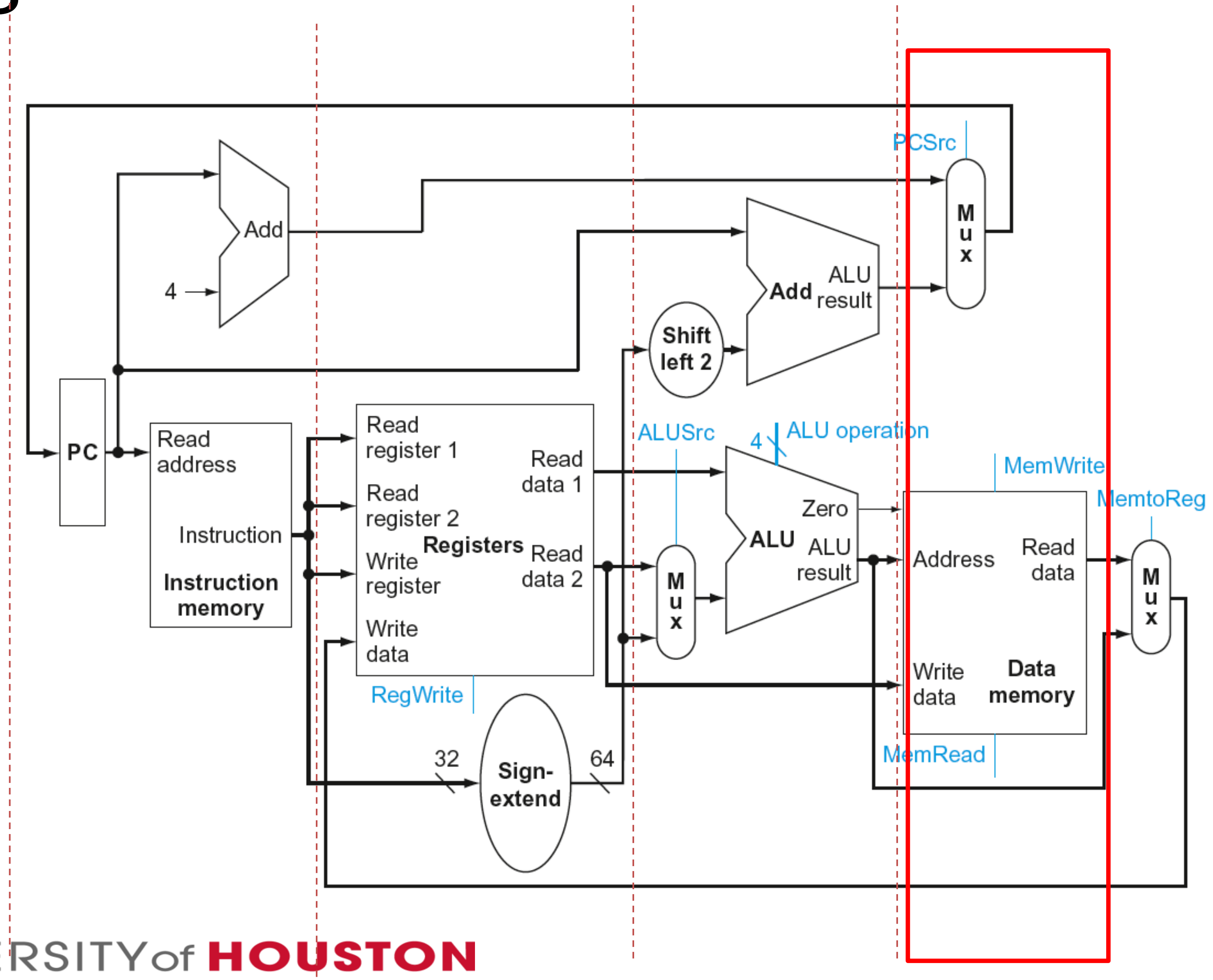
Stages in LEGv8

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
   1. Add operands (ADD)
   2. Calculate address (base + offset, LDUR, STUR)



Stages in LEGv8

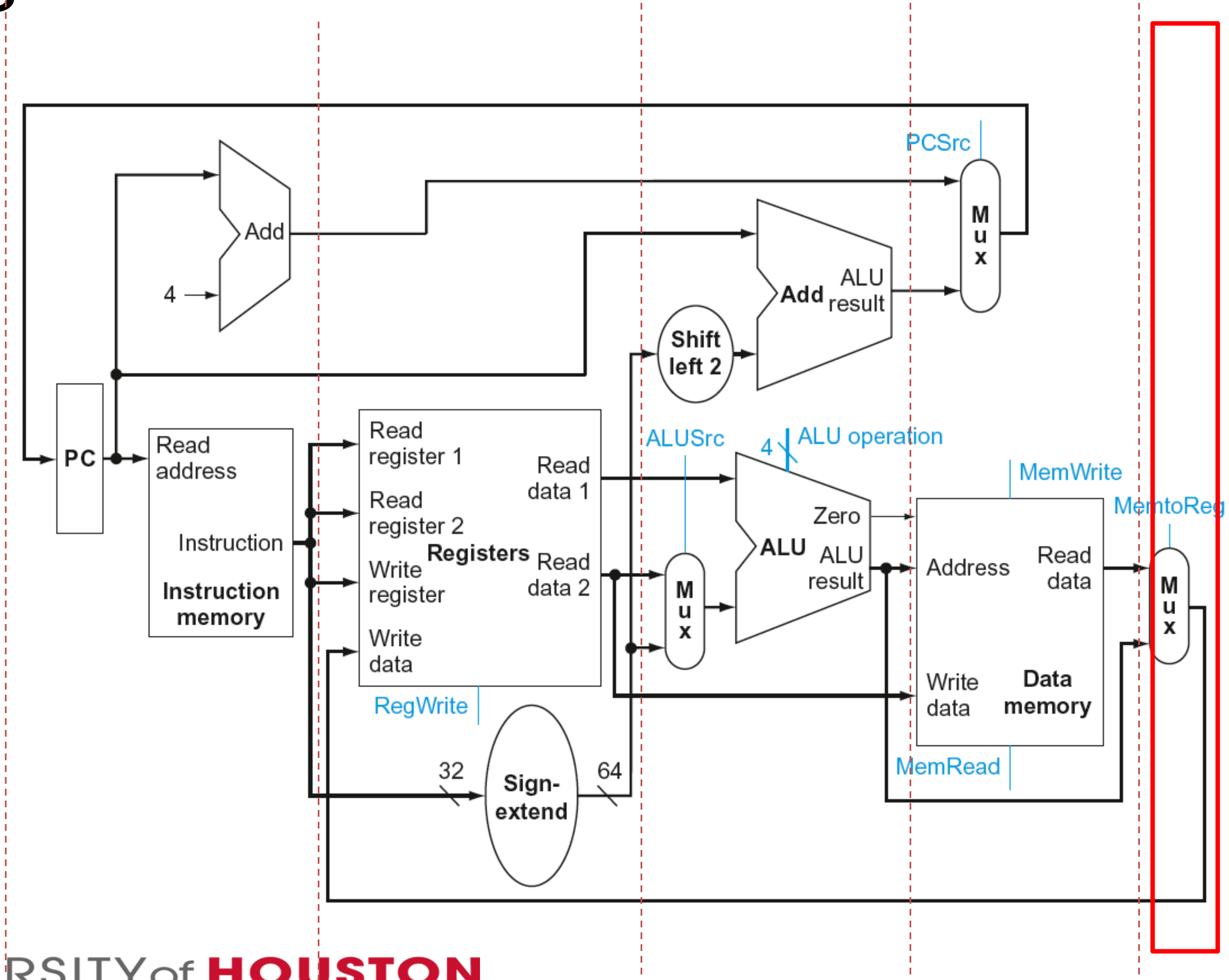
1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
    1. Read data from memory (LDUR)
    2. Store data in memory (STUR)



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
    1. Write result to destination register (add)
    2. Load value to register (LDUR)

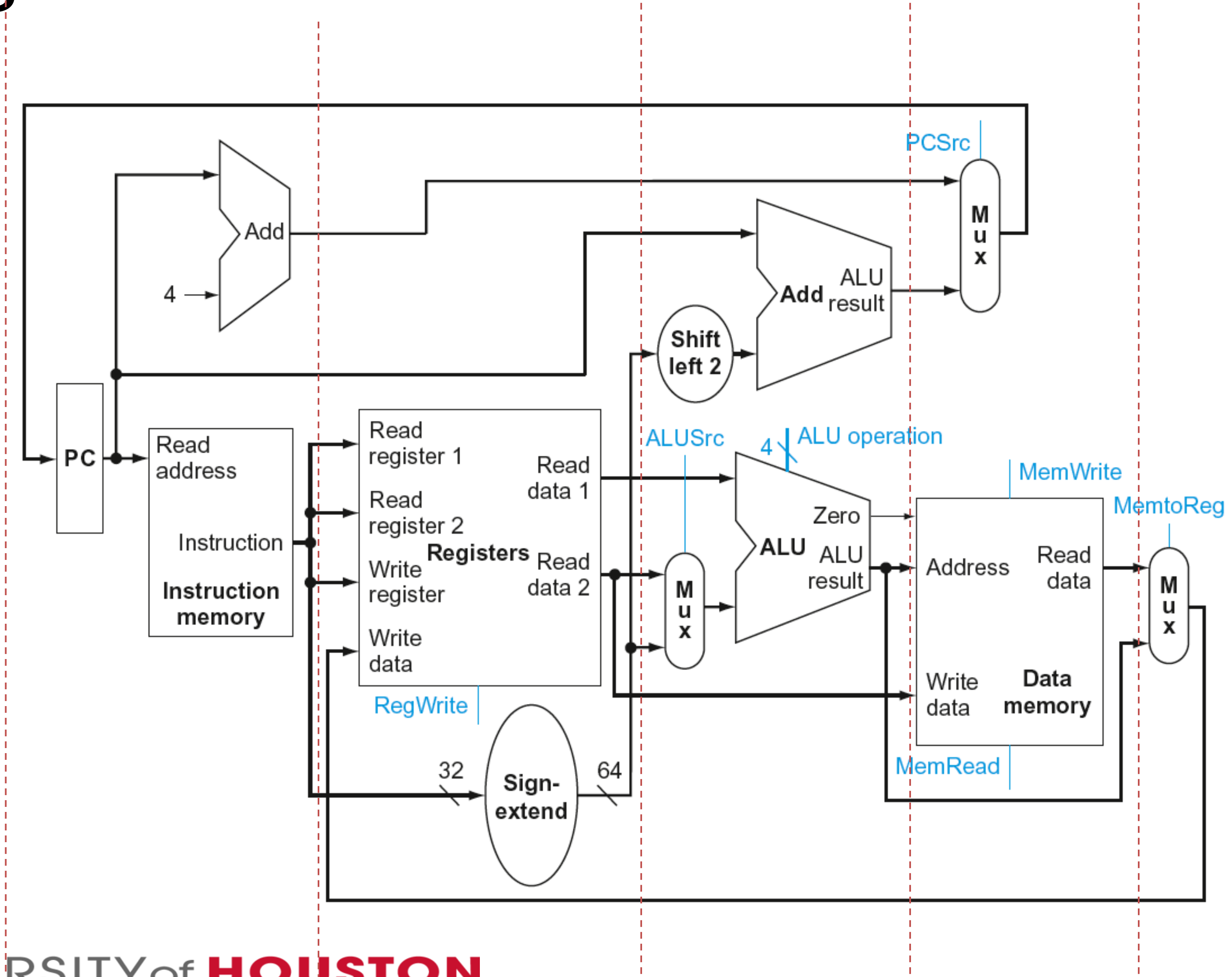


Stages in LEGv8

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					

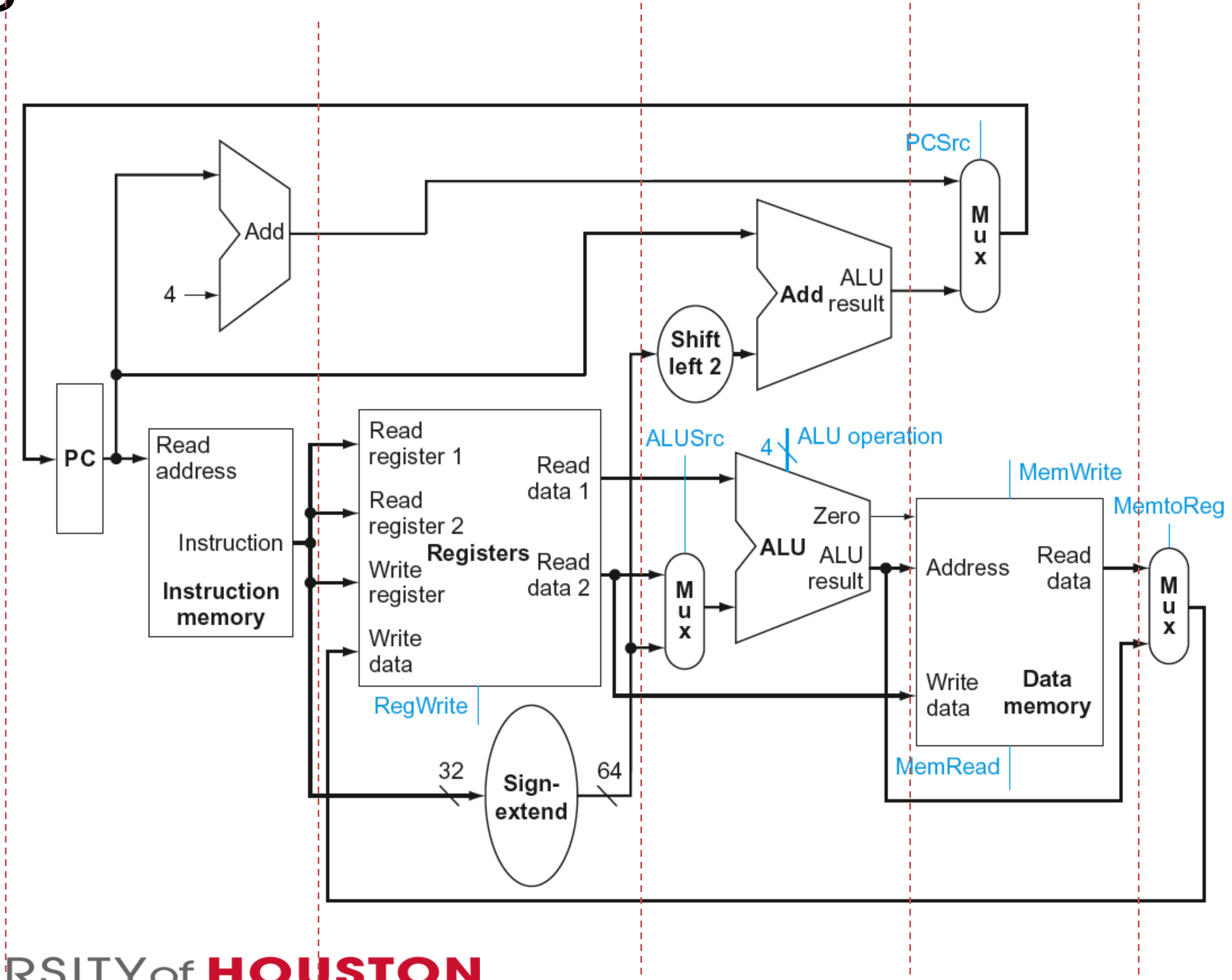


Stages in LEGv8

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					

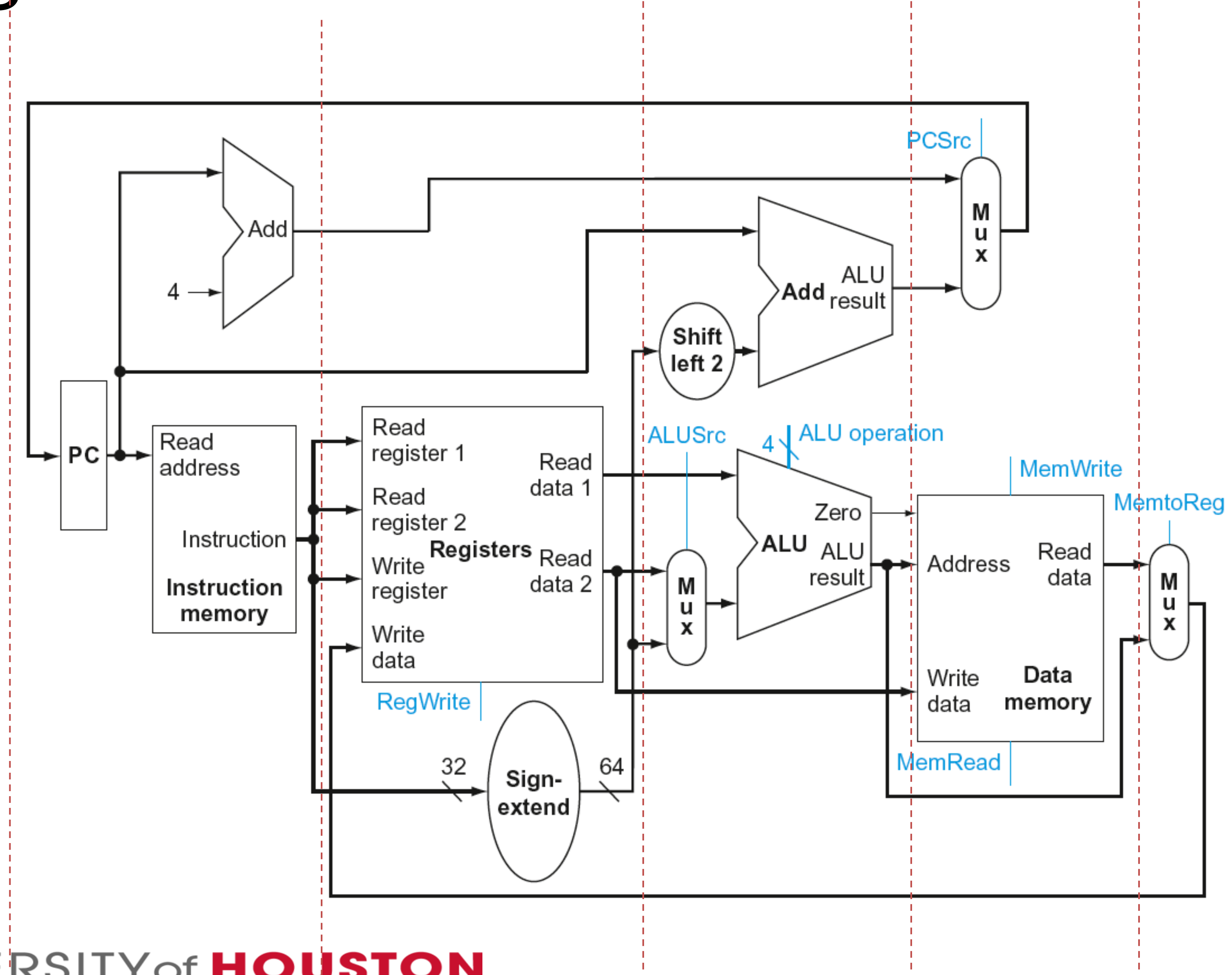


Stages in LEGv8

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					

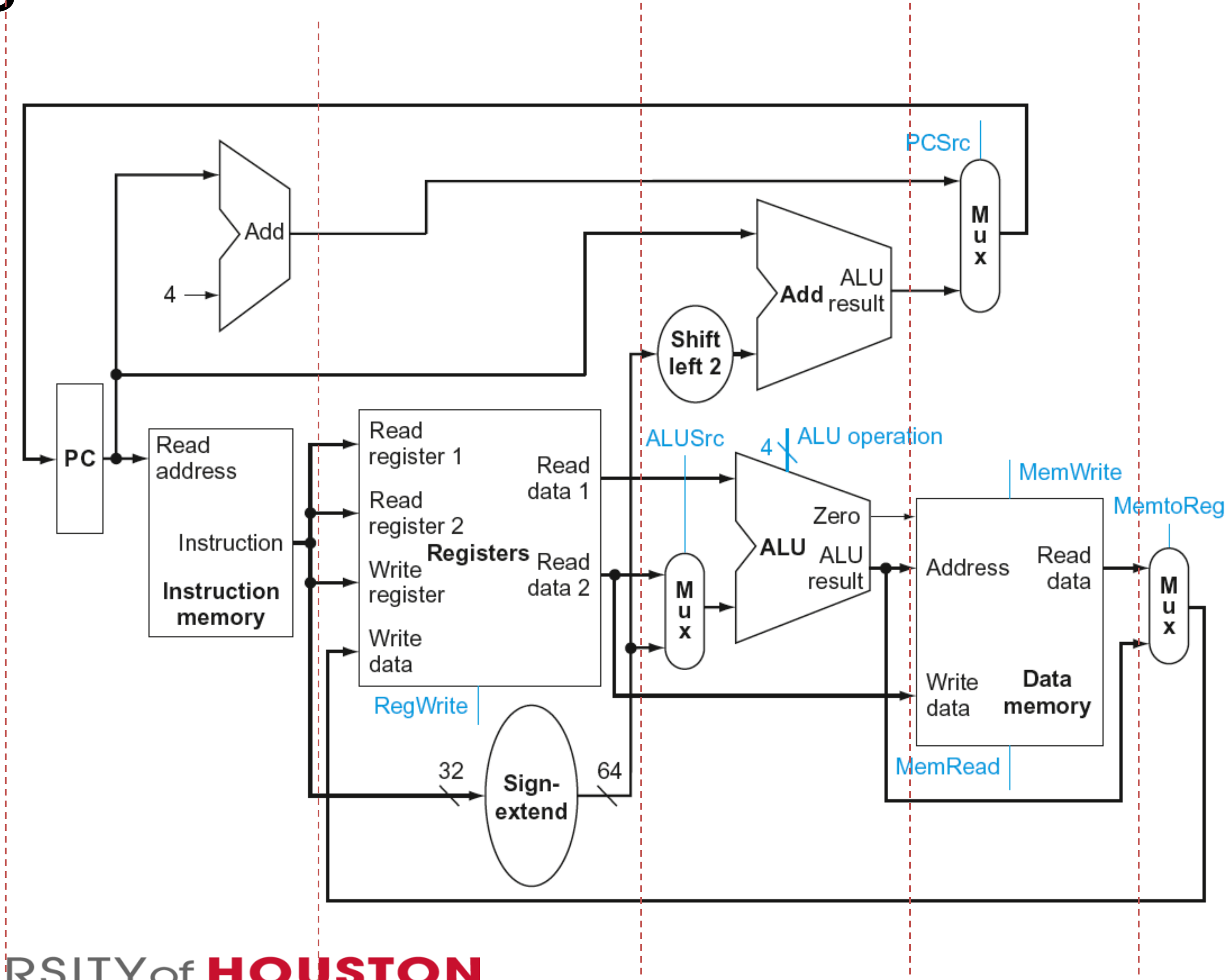


Stages in LEGv8

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					

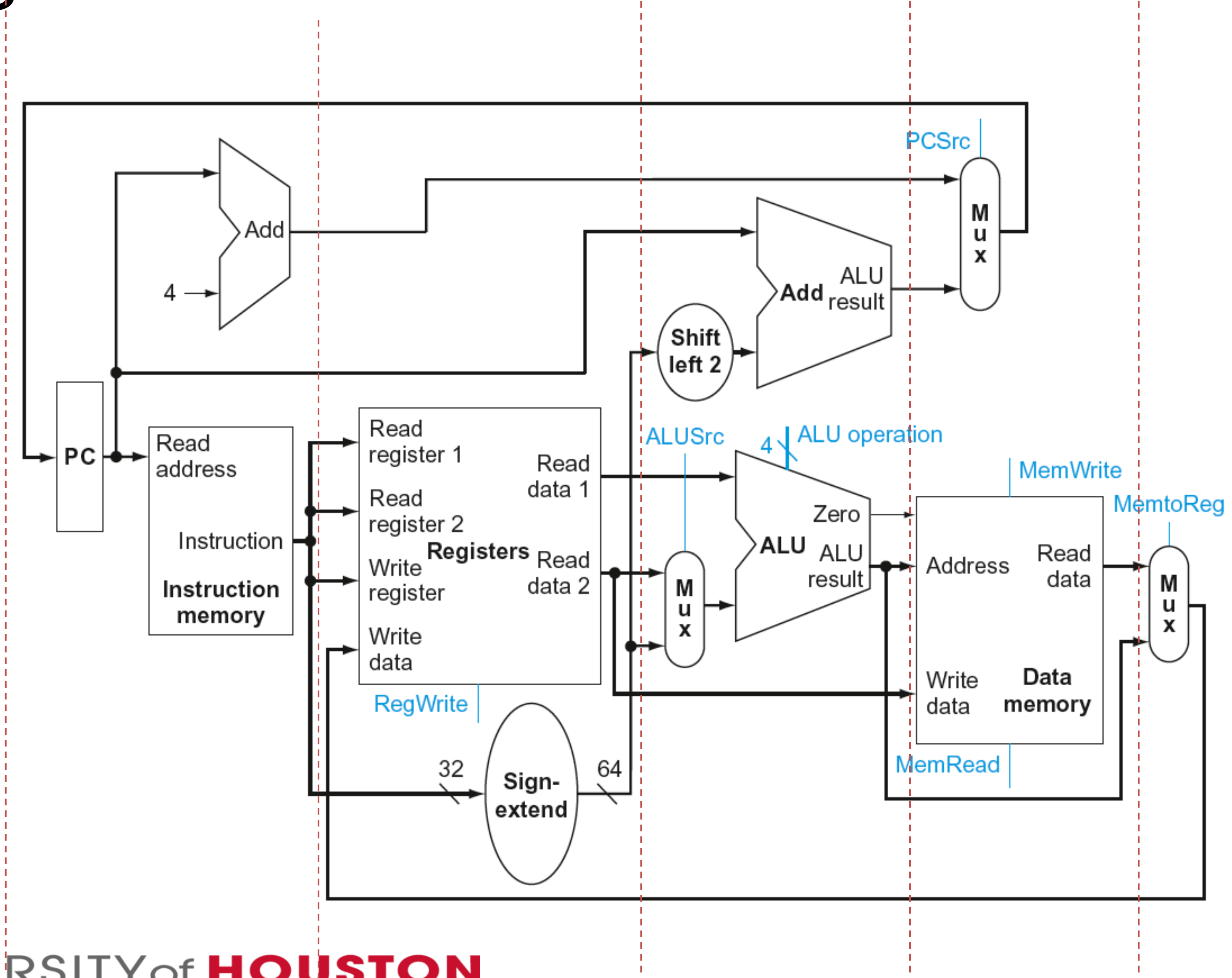


Stages in LEGv8

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	🟩	🟩	🟩		



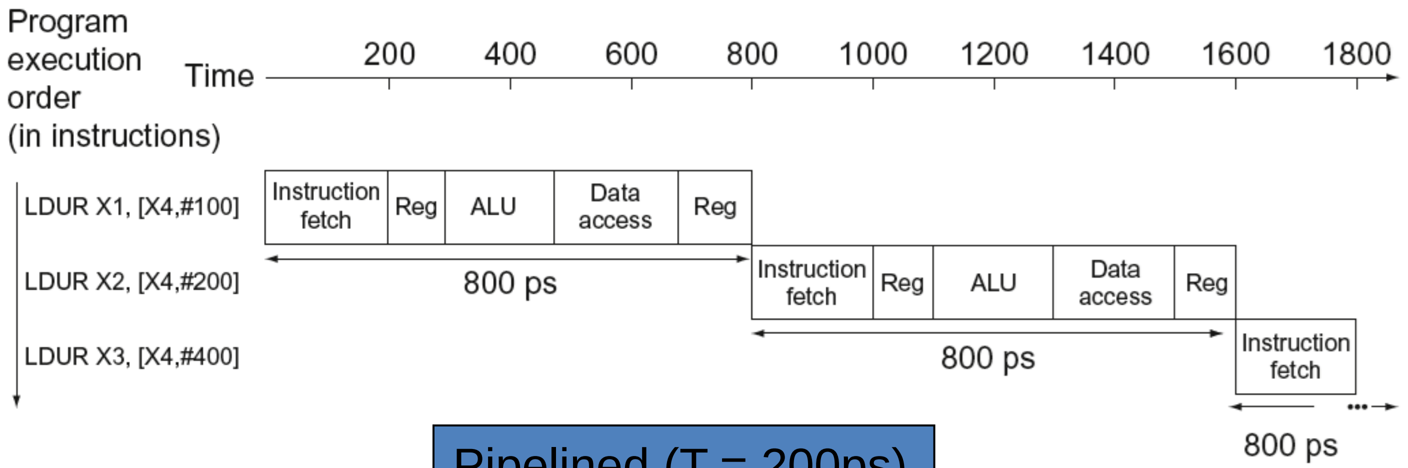
Pipeline Performance

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

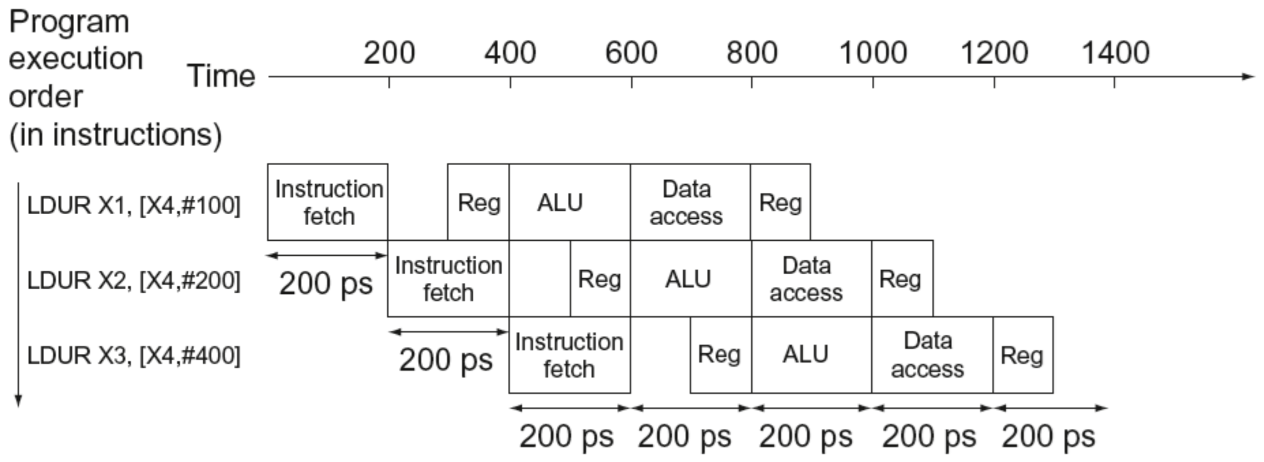
Instr	Instr fetch (IF)	Register read (ID)	ALU op (EX)	Memory access (MEM)	Register write (WB)	Total time
LDUR	200ps	100 ps	200ps	200ps	100 ps	800ps
STUR	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
CBZ	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)

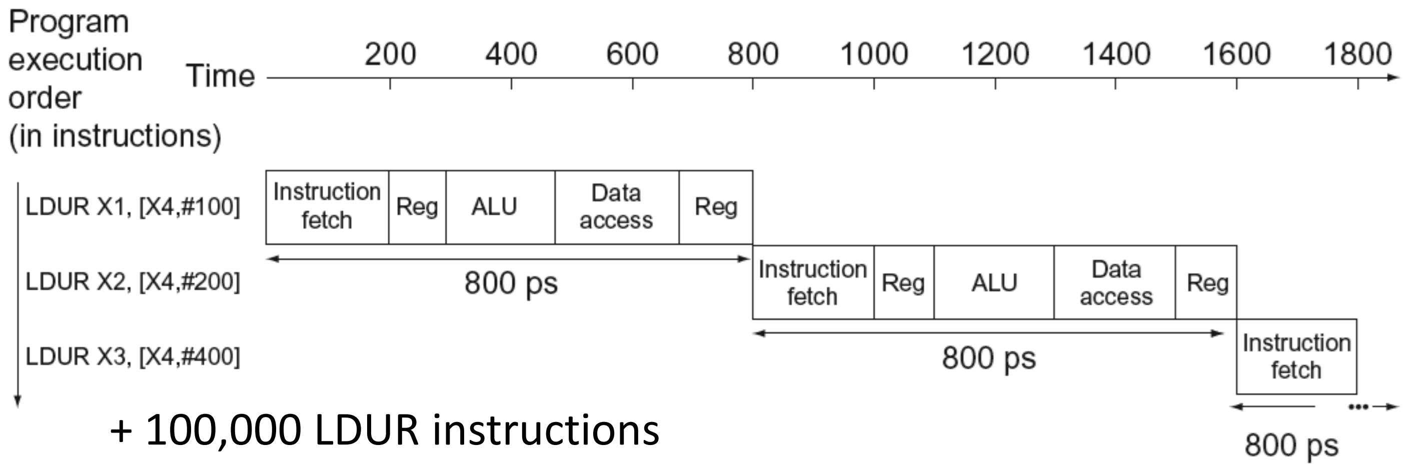


Pipelined (T_c= 200ps)

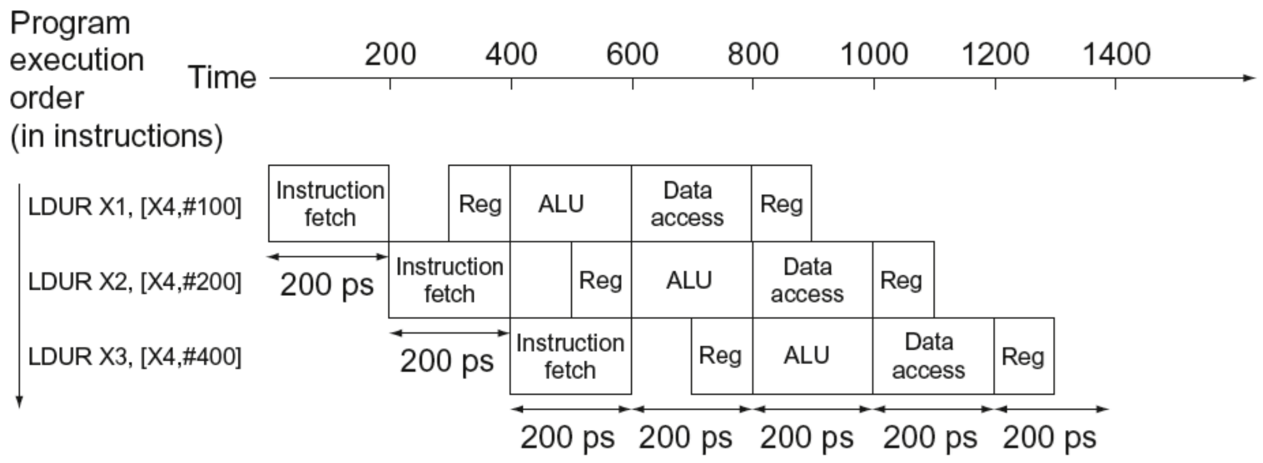


Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - $\text{Time between instructions}_{\text{pipelined}} = \dfrac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease



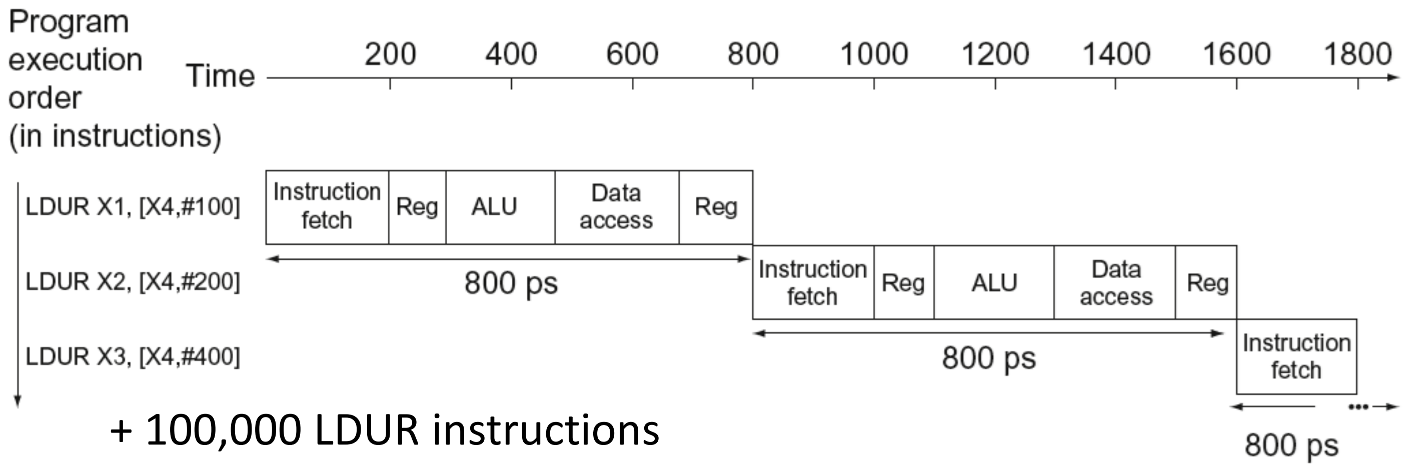
+ 100,000 LDUR instructions



+ 100,000 LDUR instructions

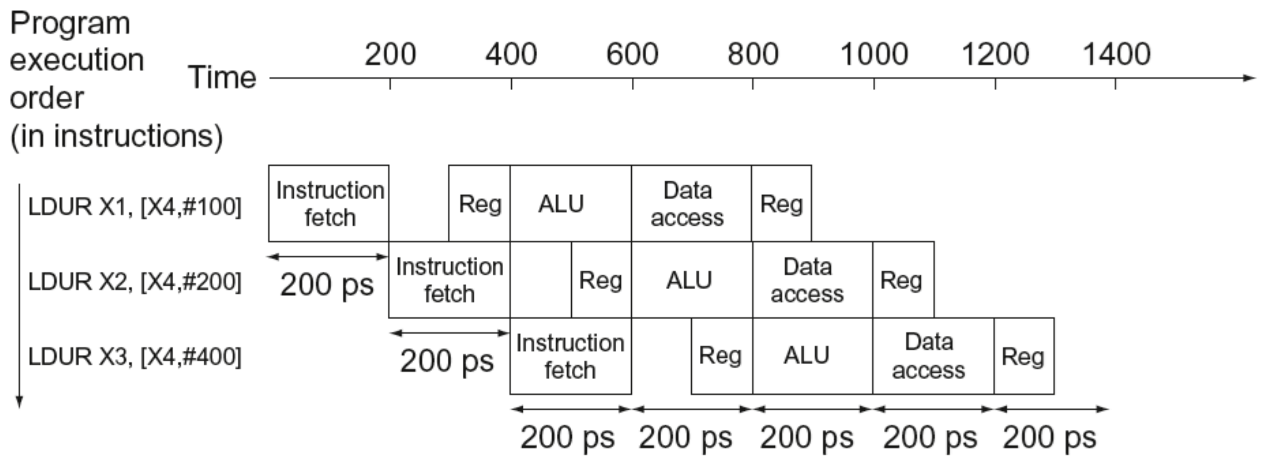
$$\left.\begin{matrix}\\ \\ \\ \end{matrix}\right\}$$ Time

$$\left.\begin{matrix}\\ \\ \\ \end{matrix}\right\}$$

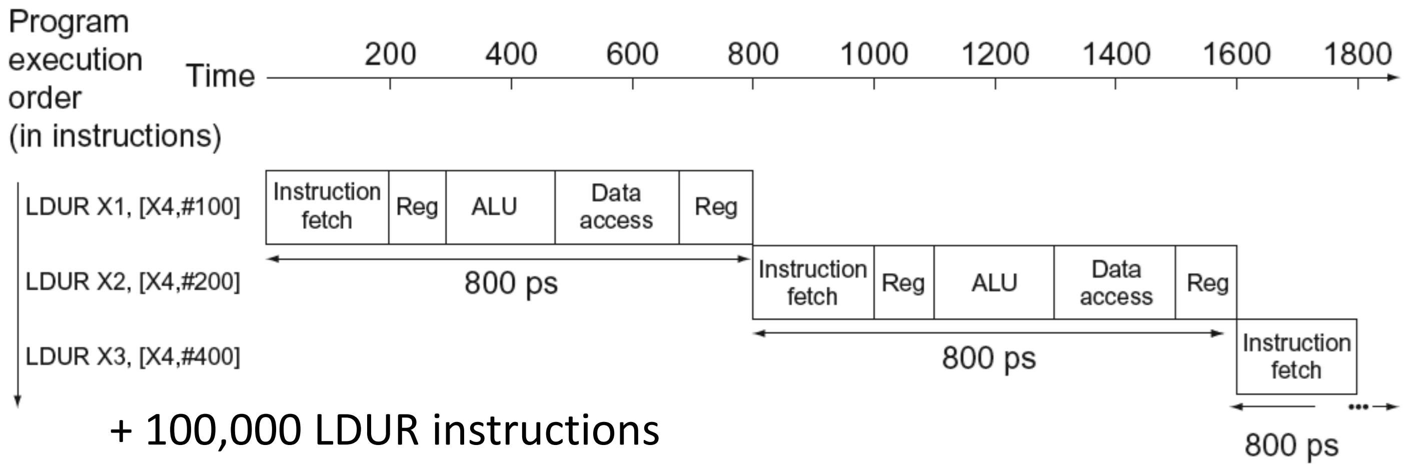


+ 100,000 LDUR instructions

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

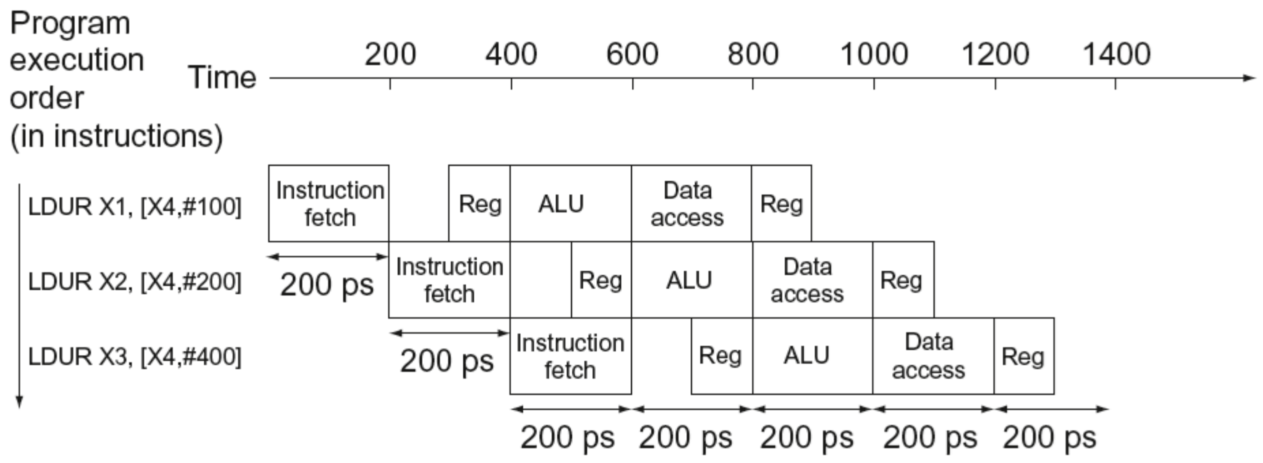


+ 100,000 LDUR instructions



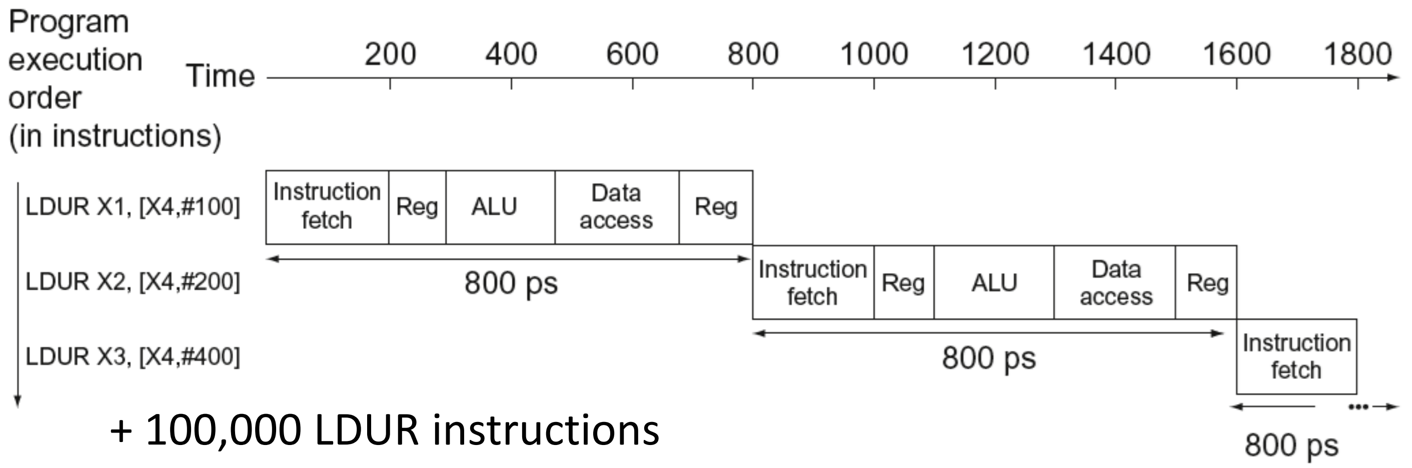
+ 100,000 LDUR instructions

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



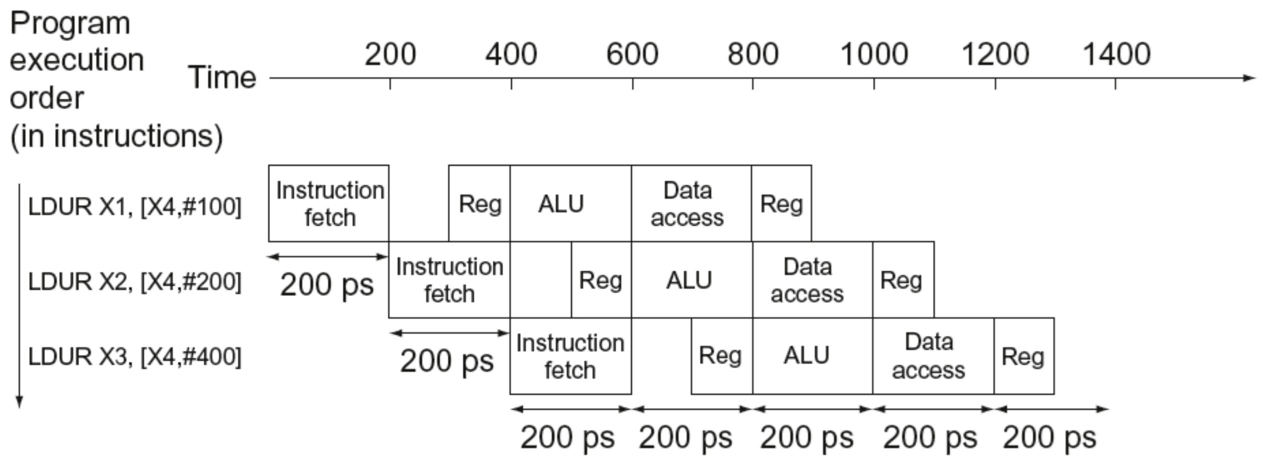
+ 100,000 LDUR instructions

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



+ 100,000 LDUR instructions

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



+ 100,000 LDUR instructions

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

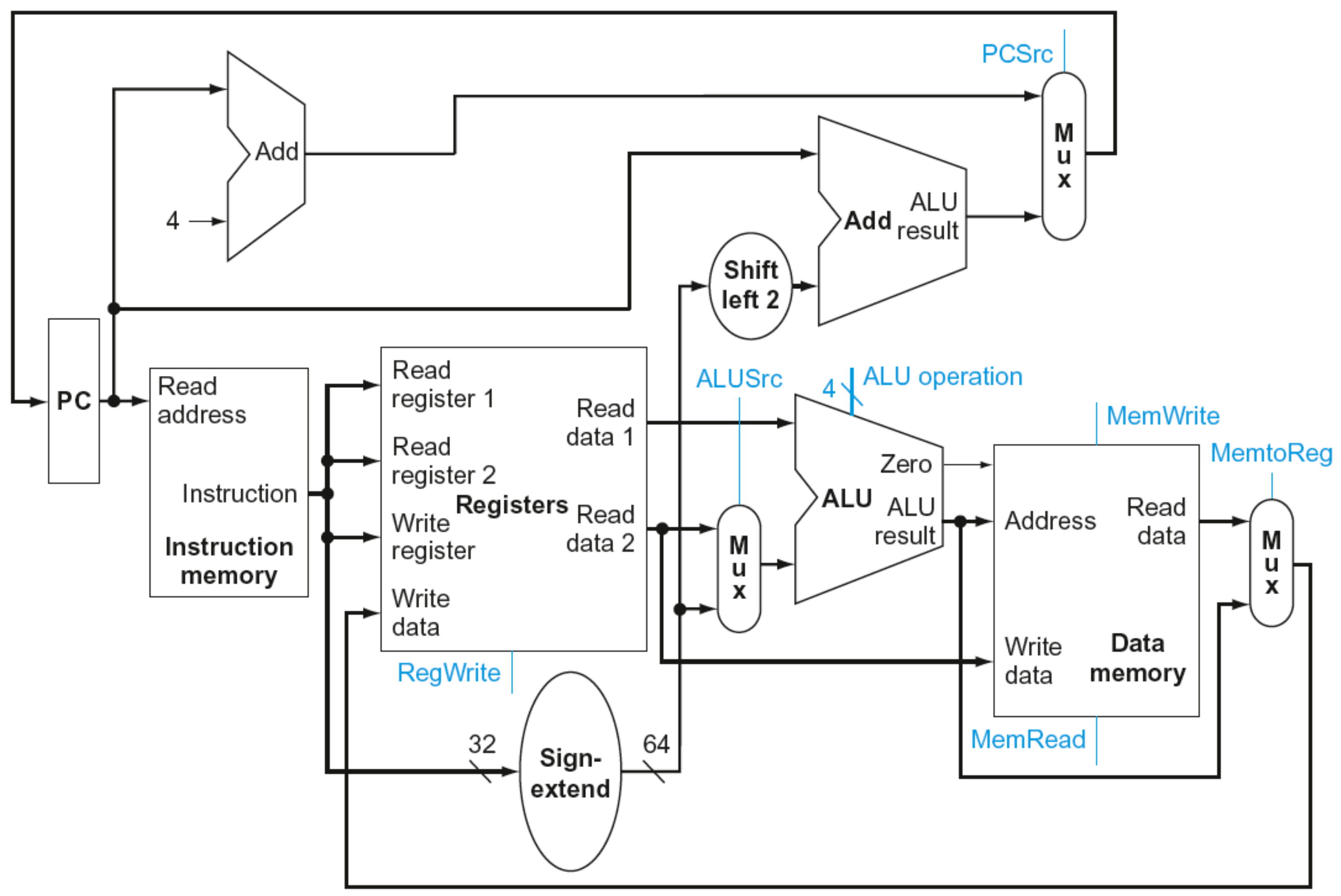
$$\text{Speedup} = \frac{800{,}002{,}400}{200{,}001{,}400} \cong 4$$

Pipelining and ISA Design

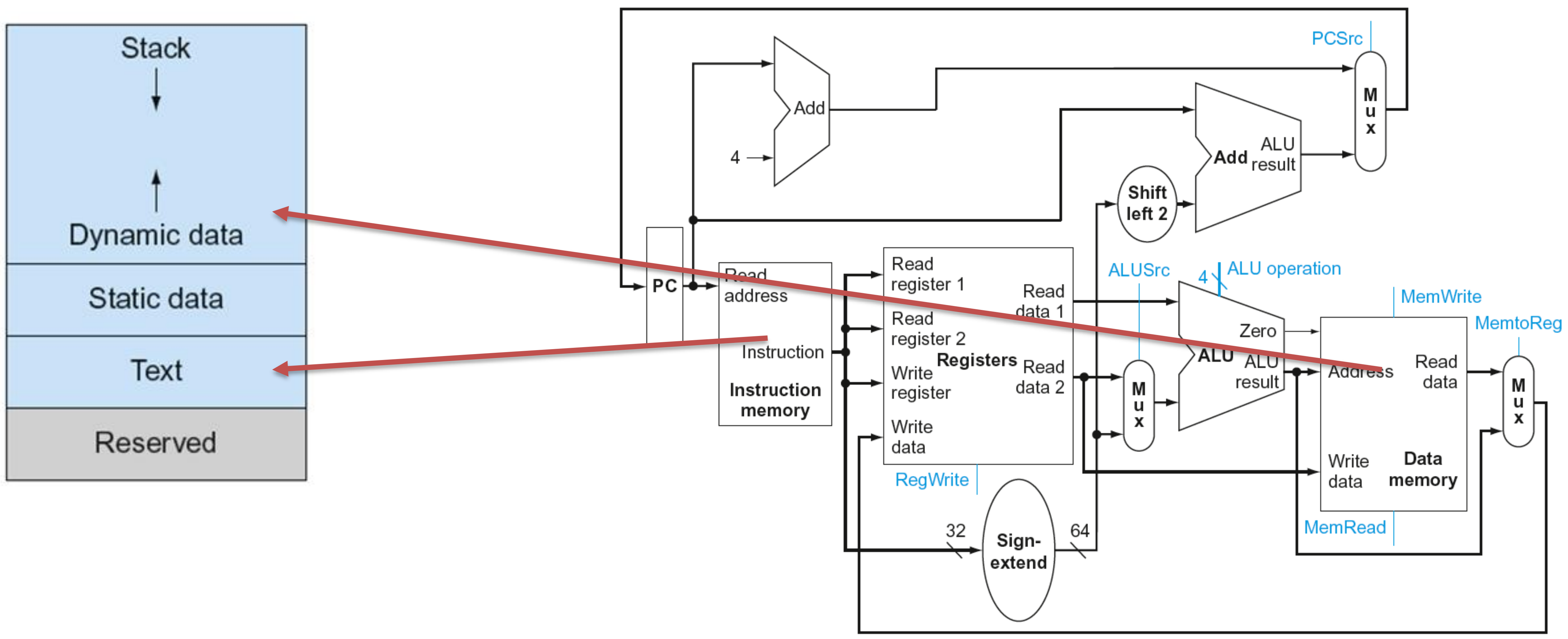
- LEGv8 ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Can Pipeline cause Issues?

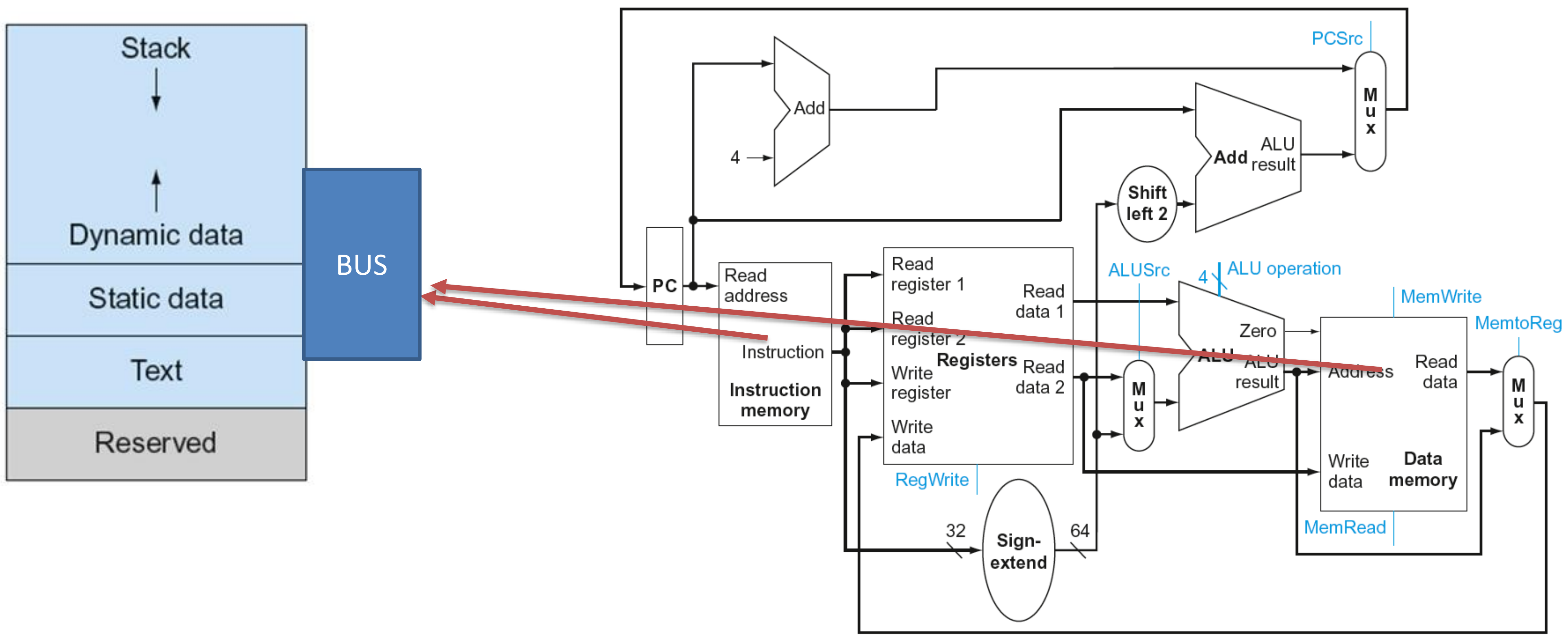
Resources Sharing



Resources Sharing



Single Memory Example

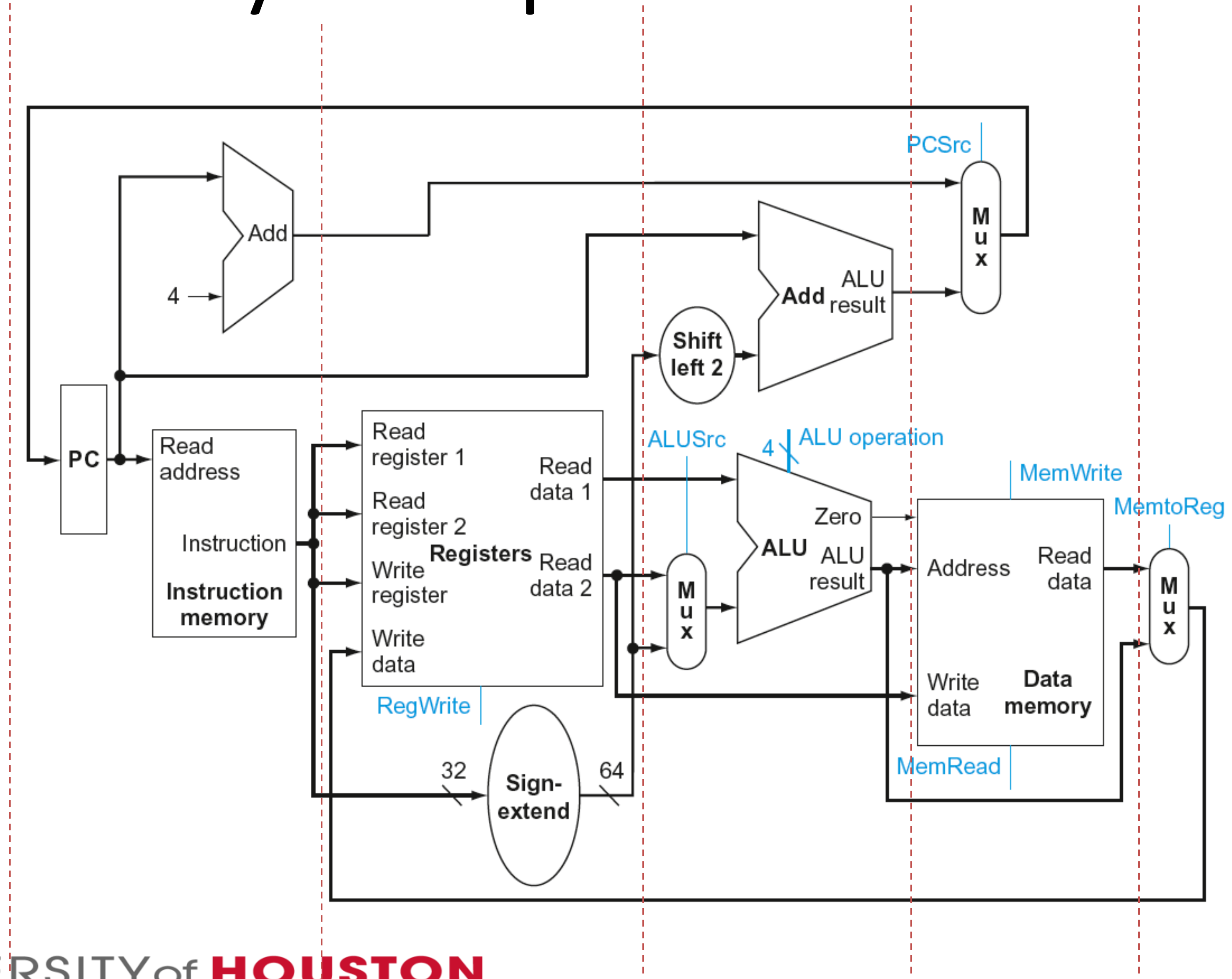


Single Memory Example

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	🟩	🟩	🟩	🟩	🟩

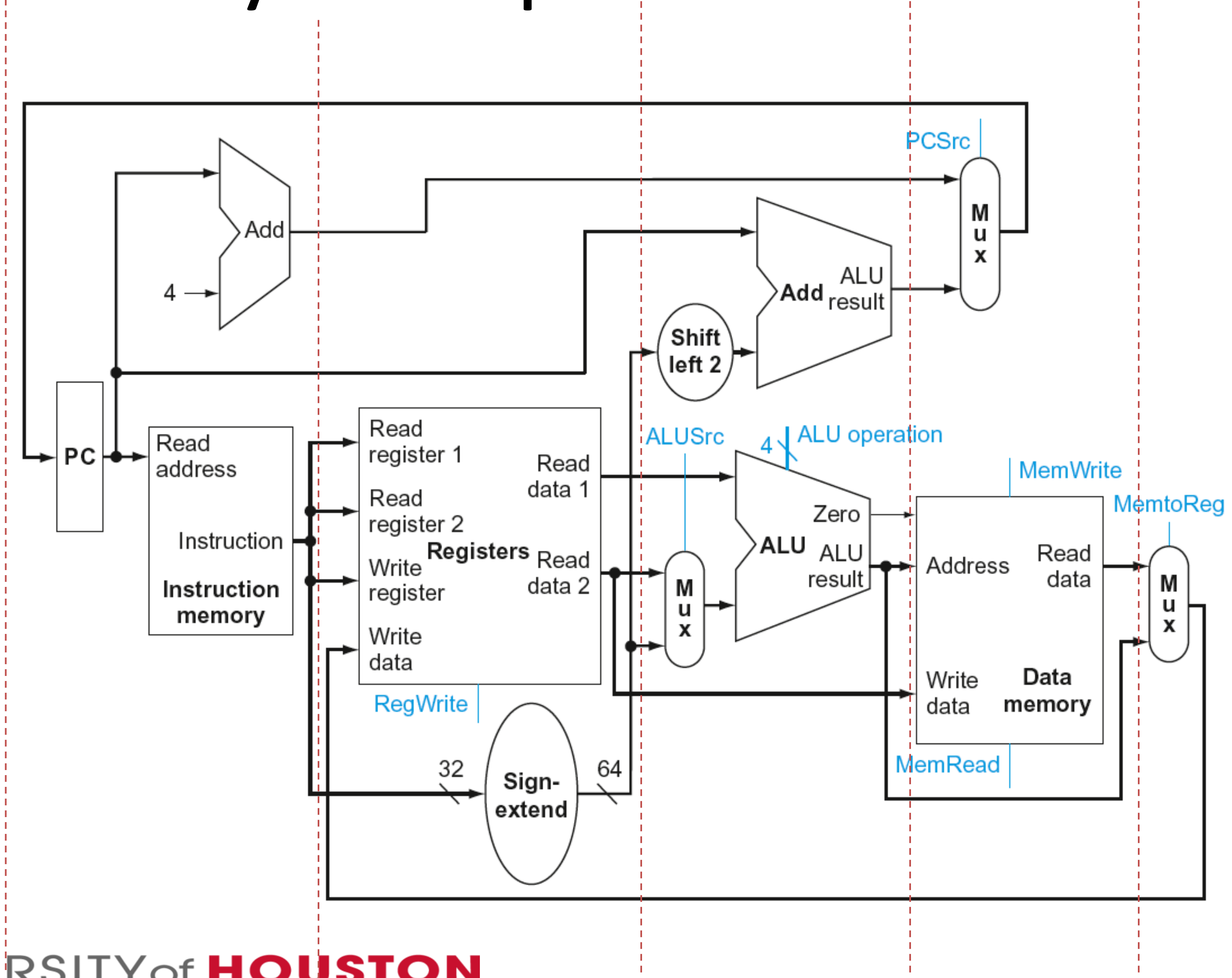


Single Memory Example

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	🟩	🟩	🟩	🟩	🟩



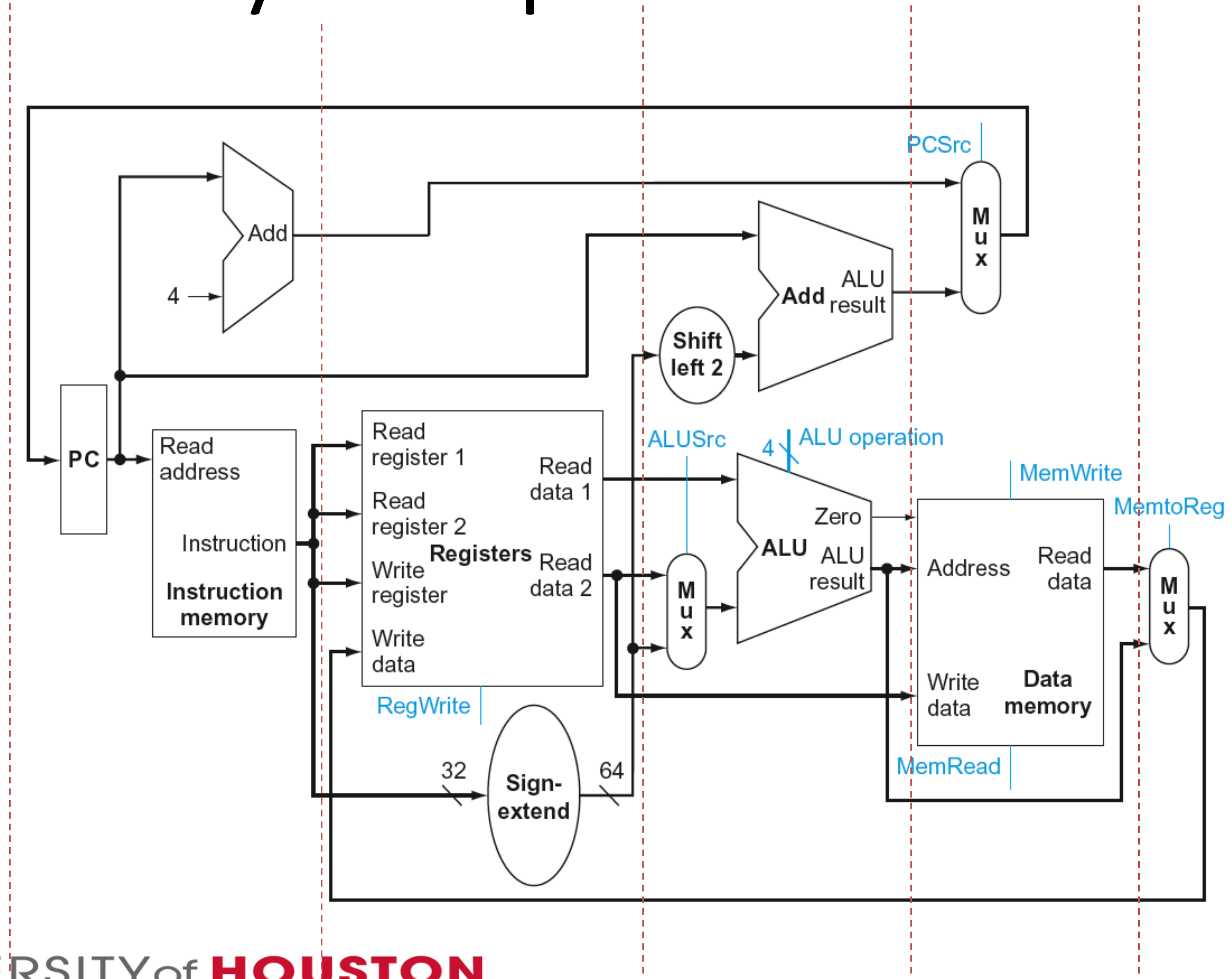
Single Memory Example

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)	🟩	🟩	🟩		🟩

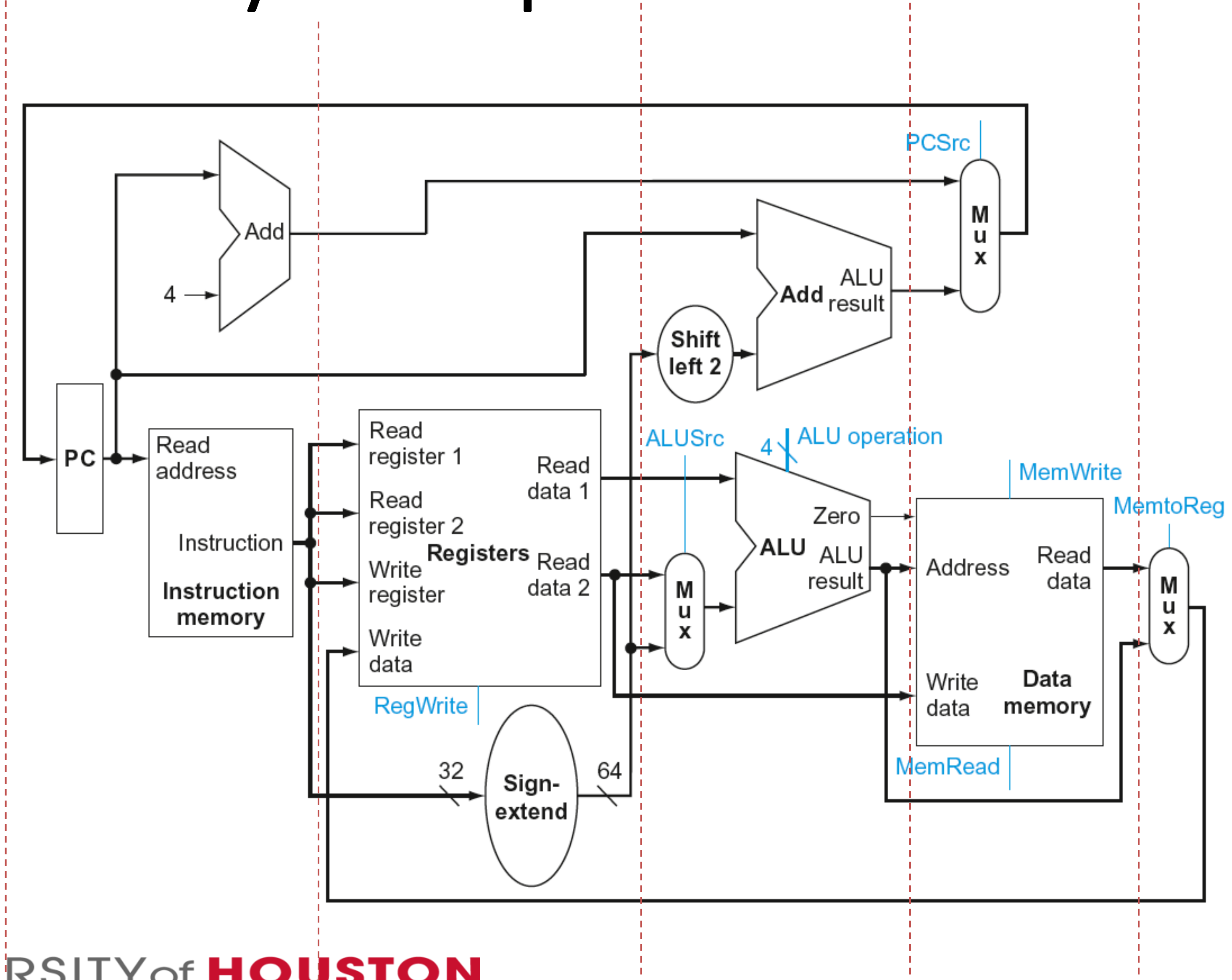


Single Memory Example

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					
ADD						
ADD						
ADD						

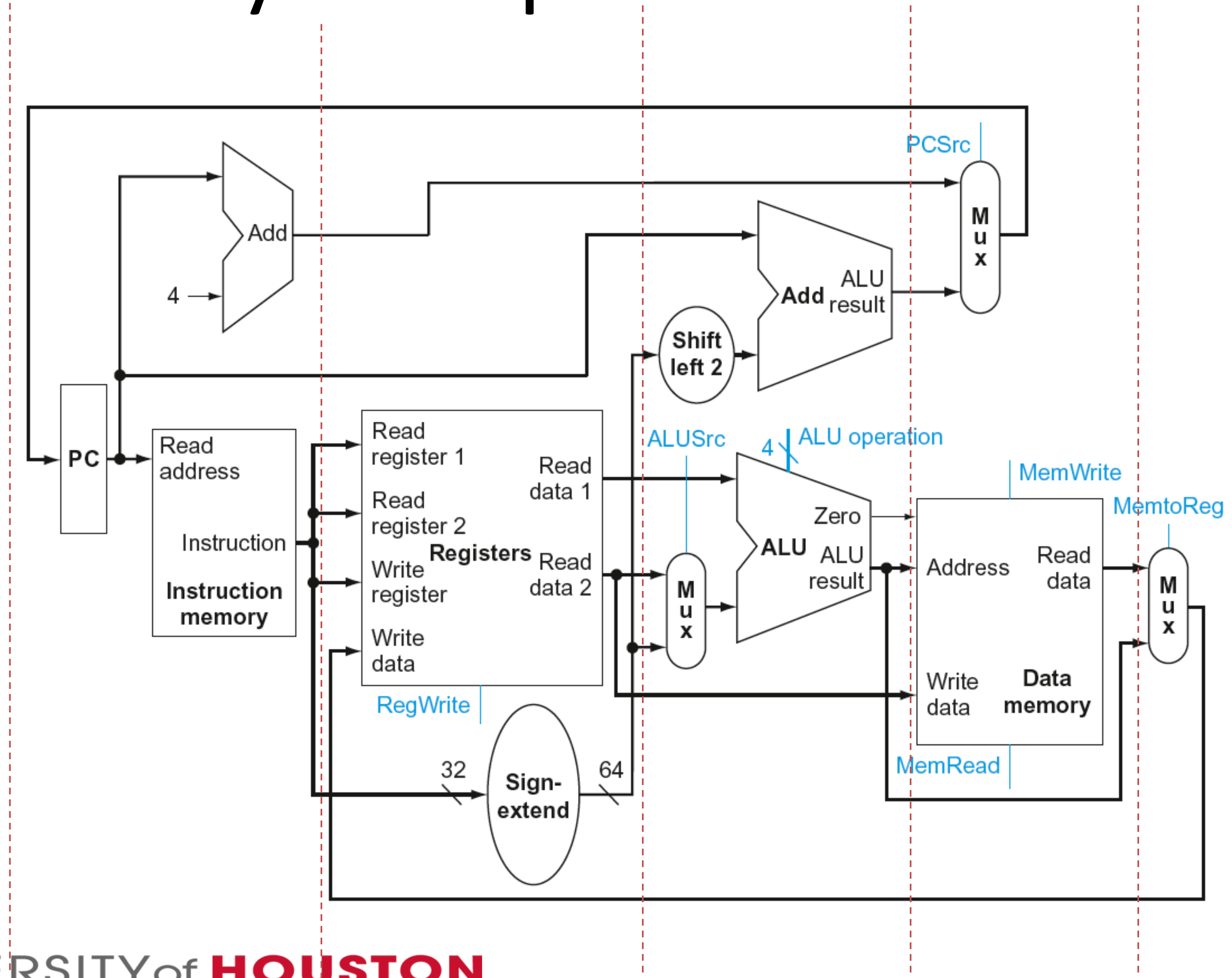


Single Memory Example

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				
ADD		1				
ADD						
ADD						

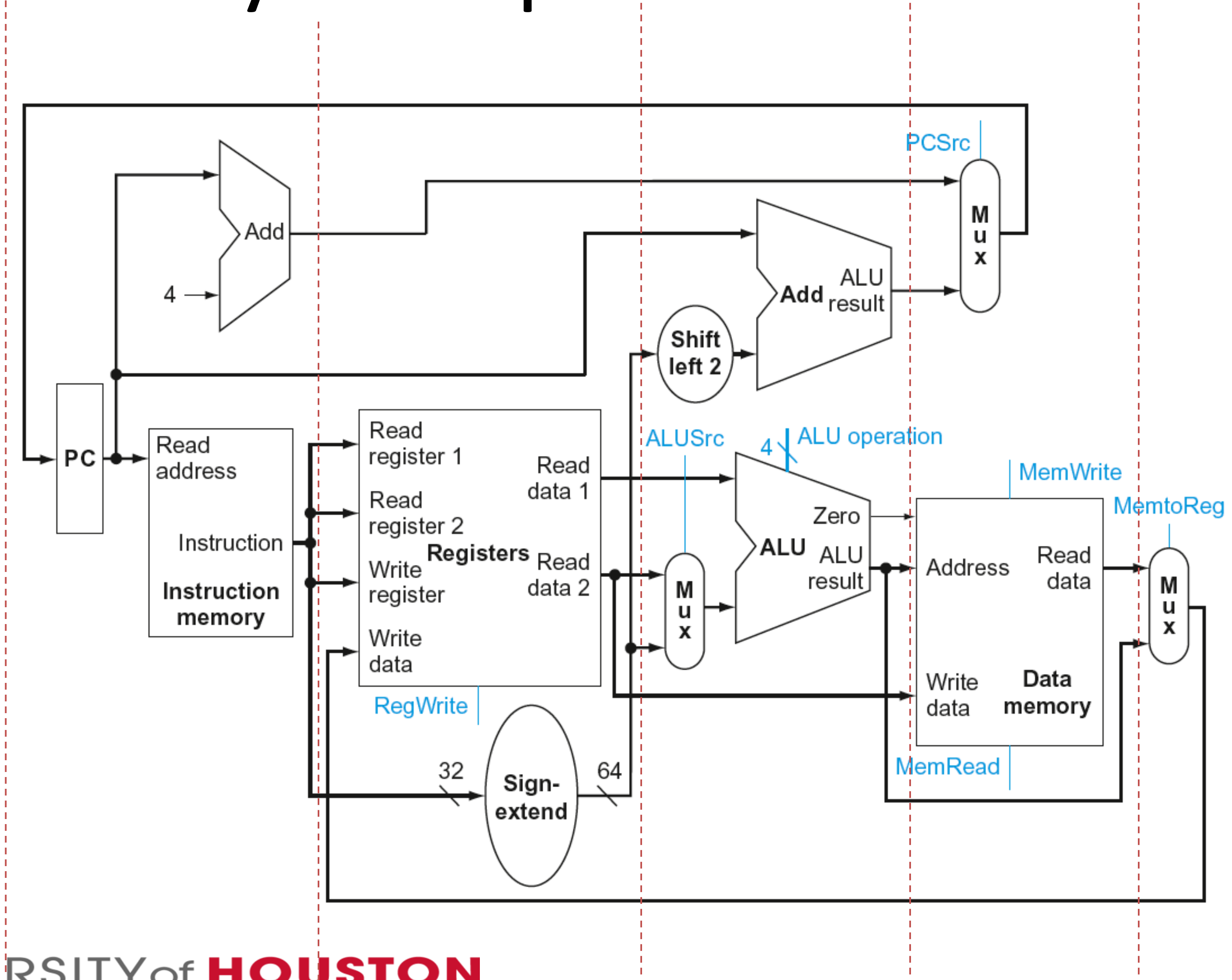


Single Memory Example

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

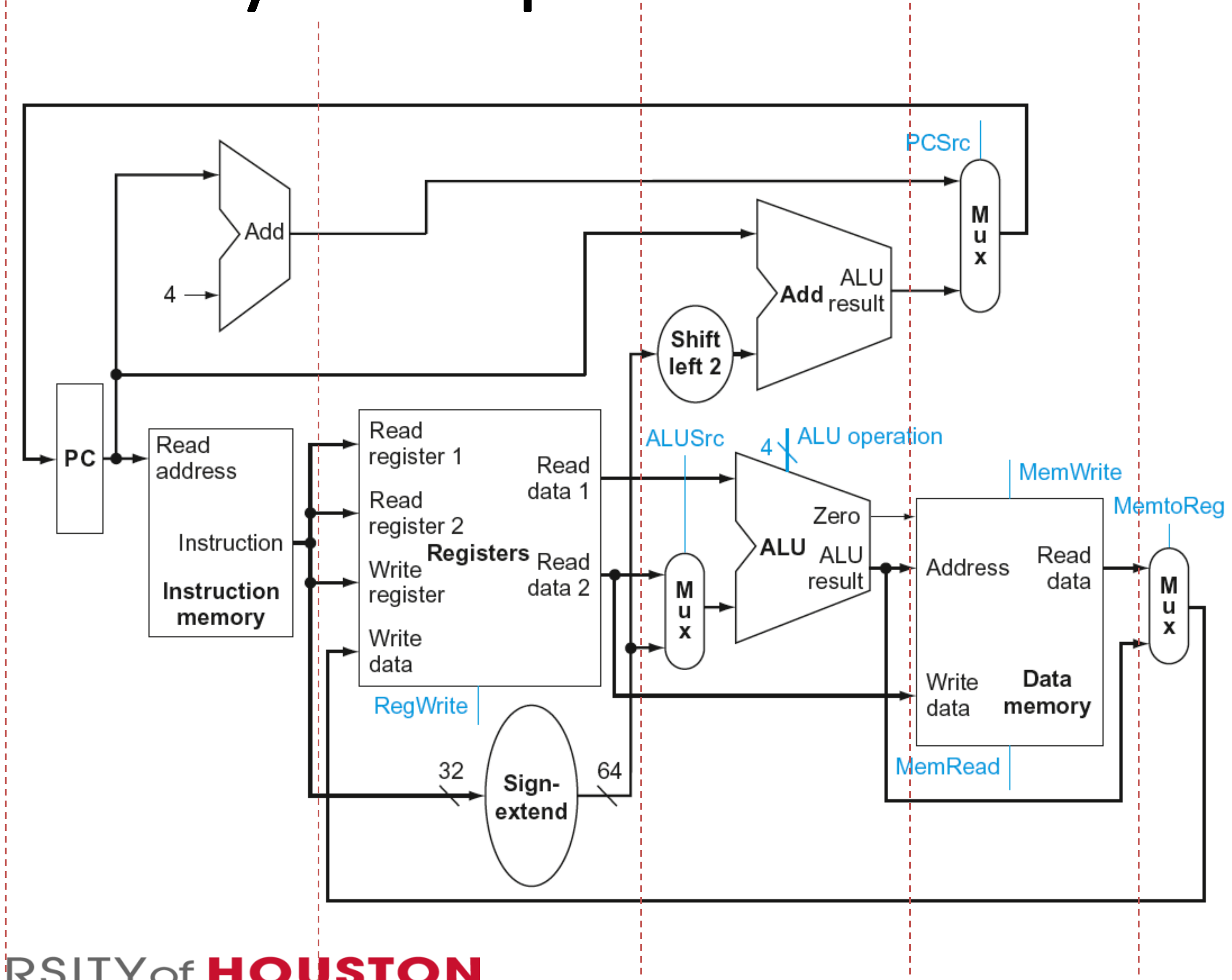


Single Memory Example

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	3



Hazards

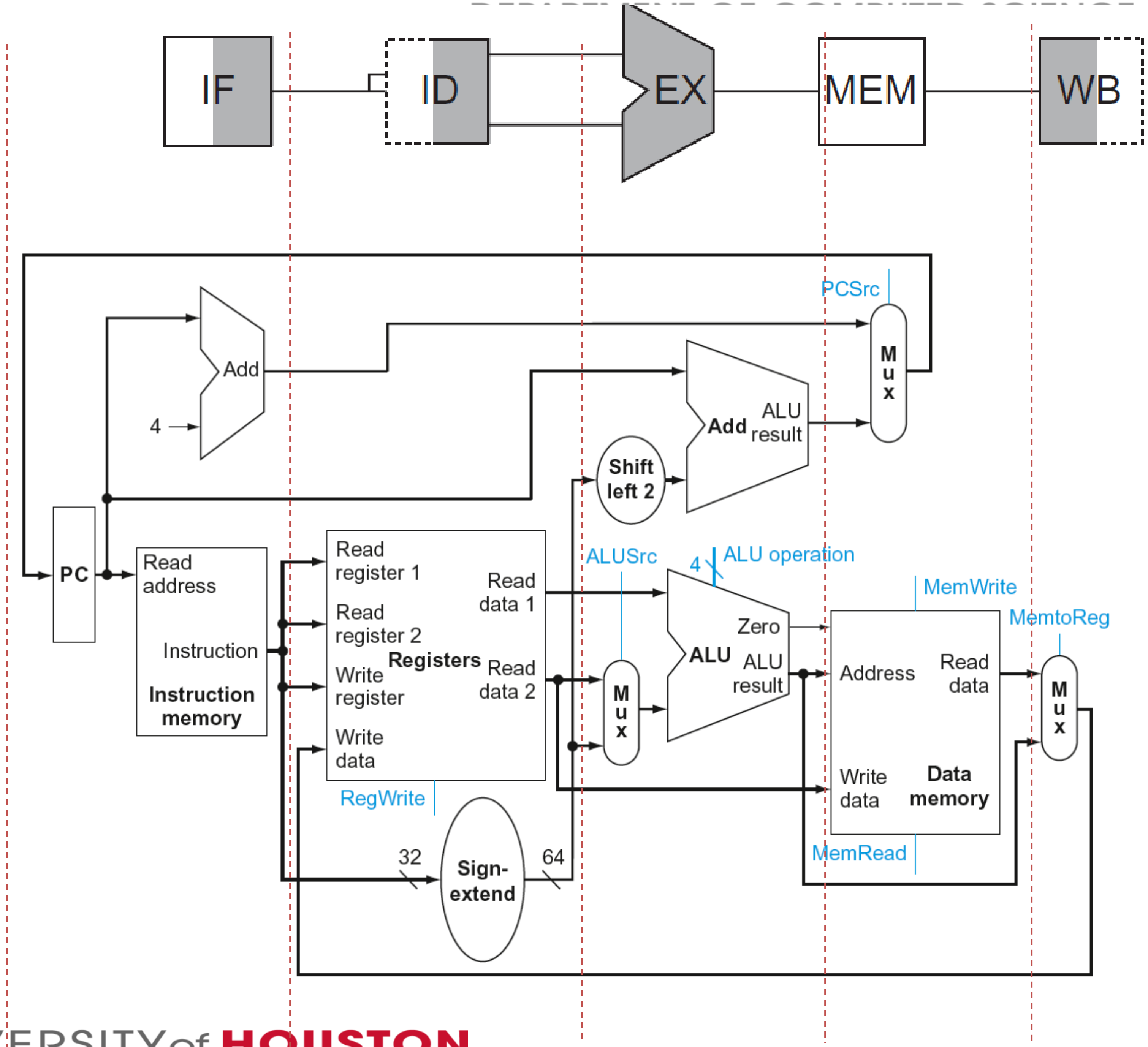
- Situations that prevent starting the next instruction in the next cycle

Hazards

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

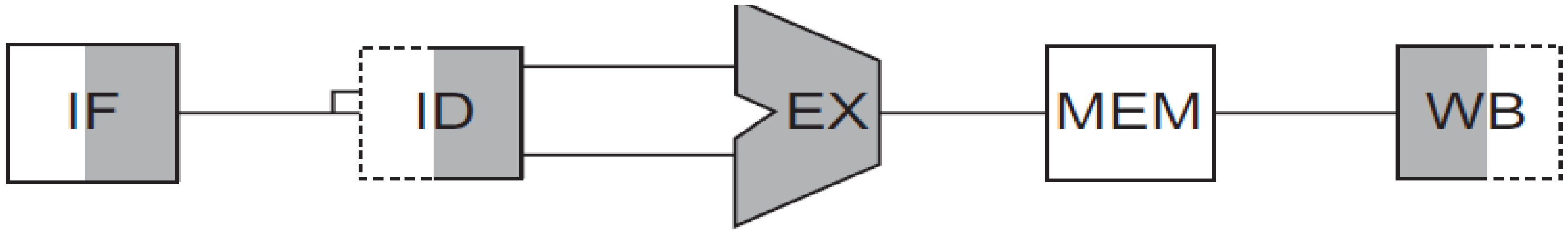
Five stages, one step per stage

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 Instruction

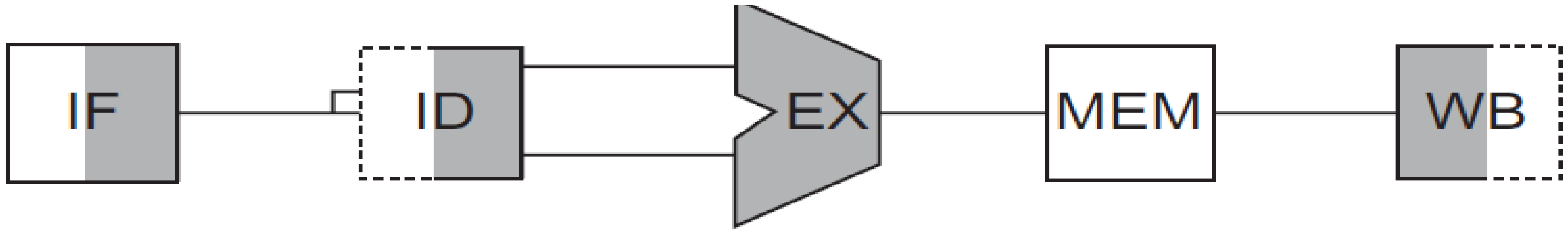
•



	IF	ID	EX	Mem	WB
ADD `X19, X0, X1`	Fetch instruction from mem	Read data from registers 0, 1	Add values of registers 0, 1	--	Write results to X19

Example R-Type Instruction

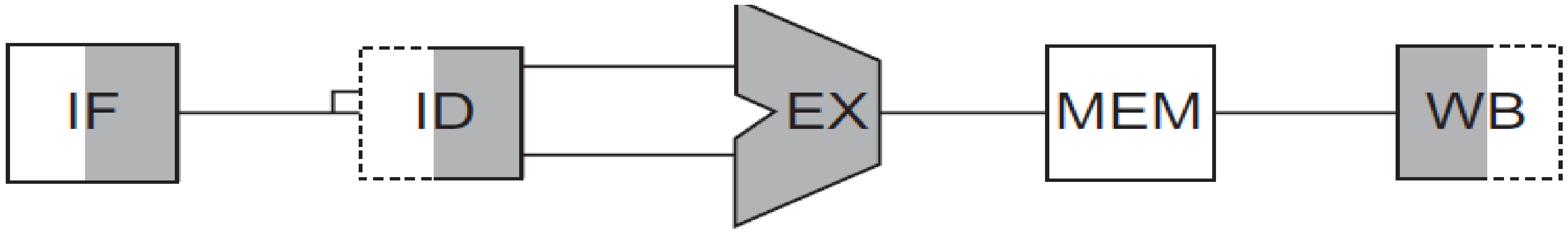
•



Cycles	1	2	3	4
`ADD X19, X0, X1`	Fetch instruction from mem			
`SUB X2, X19, X3`				

Example R-Type Instrucstion

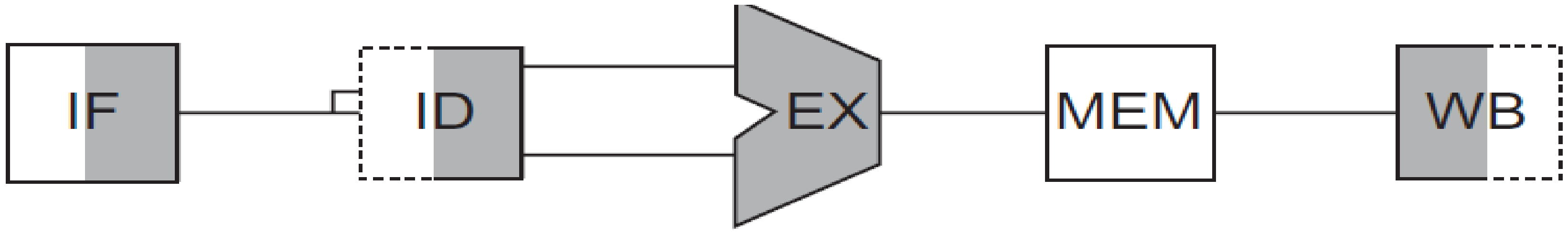
•



Cycles	1	2	3	4
`ADD X19, X0, X1`	Fetch instruction from mem	Read data from registers 0, 1		
`SUB X2, X19, X3`		Fetch instruction from mem		

Example R-Type Instrucstion

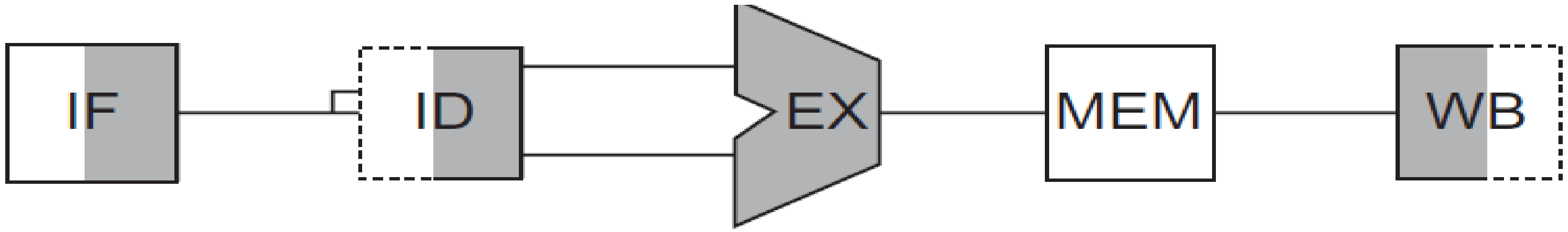
•



Cycles	1	2	3	4
`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	

Example R-Type Instrucstion

•



Cycles	1	2	3	4
`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