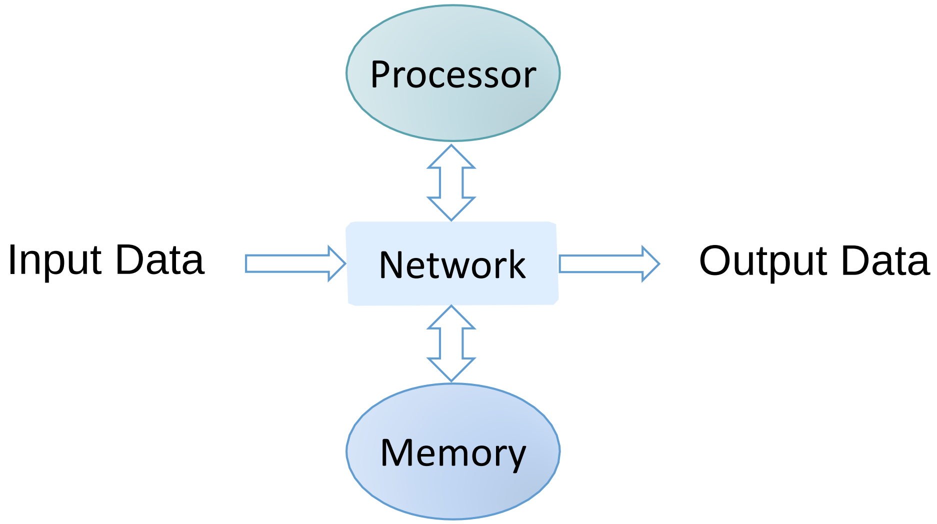
Building Blocks

Three Building Blocks



Computation

Communication

Storage

- Trends
  - Growing diversity in application requirements.
  - Energy and power constrain systems.
  - Multiple cores integrated onto a single chip.
  - Heterogeneous systems-on-chip.
  - New device technologies for three basic blocks.

First, a Simple Processor

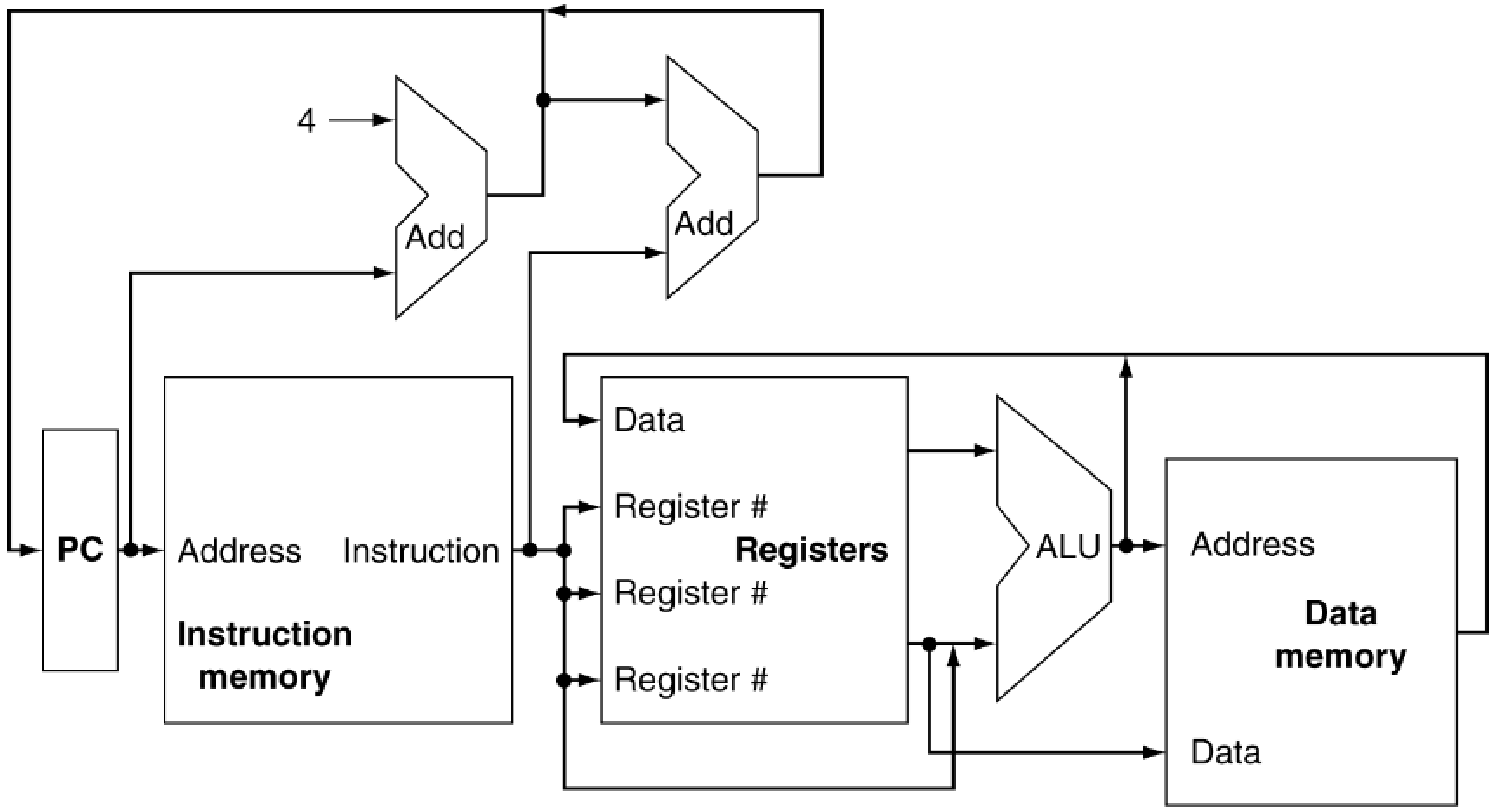
- CPU performance factors.
  - Instruction count
    - Determined by ISA and compiler
  - CPI and cycle time.
    - Determined by CPU hardware
- We will examine two implementations.
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects.
  - Memory reference: `LDUR`, `STUR`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`

R
I
J

Instruction Execution

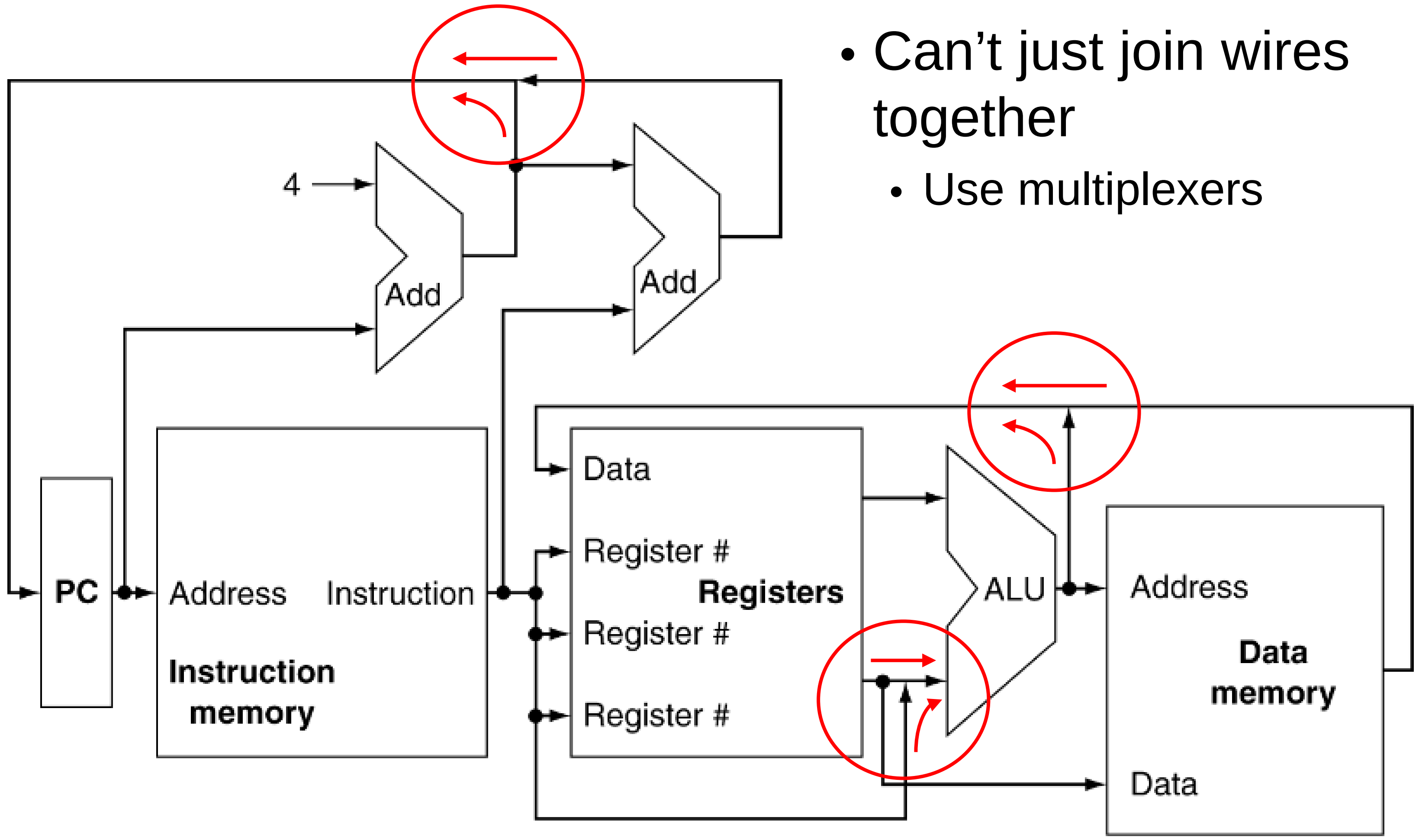
- PC $\rightarrow$ instruction memory, fetch instruction
- Register numbers $\rightarrow$ register file, read registers
- Depending on instruction class
  - Use ALU to calculate:
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store.
  - PC $\leftarrow$ target address or PC + 4.

CPU Overview



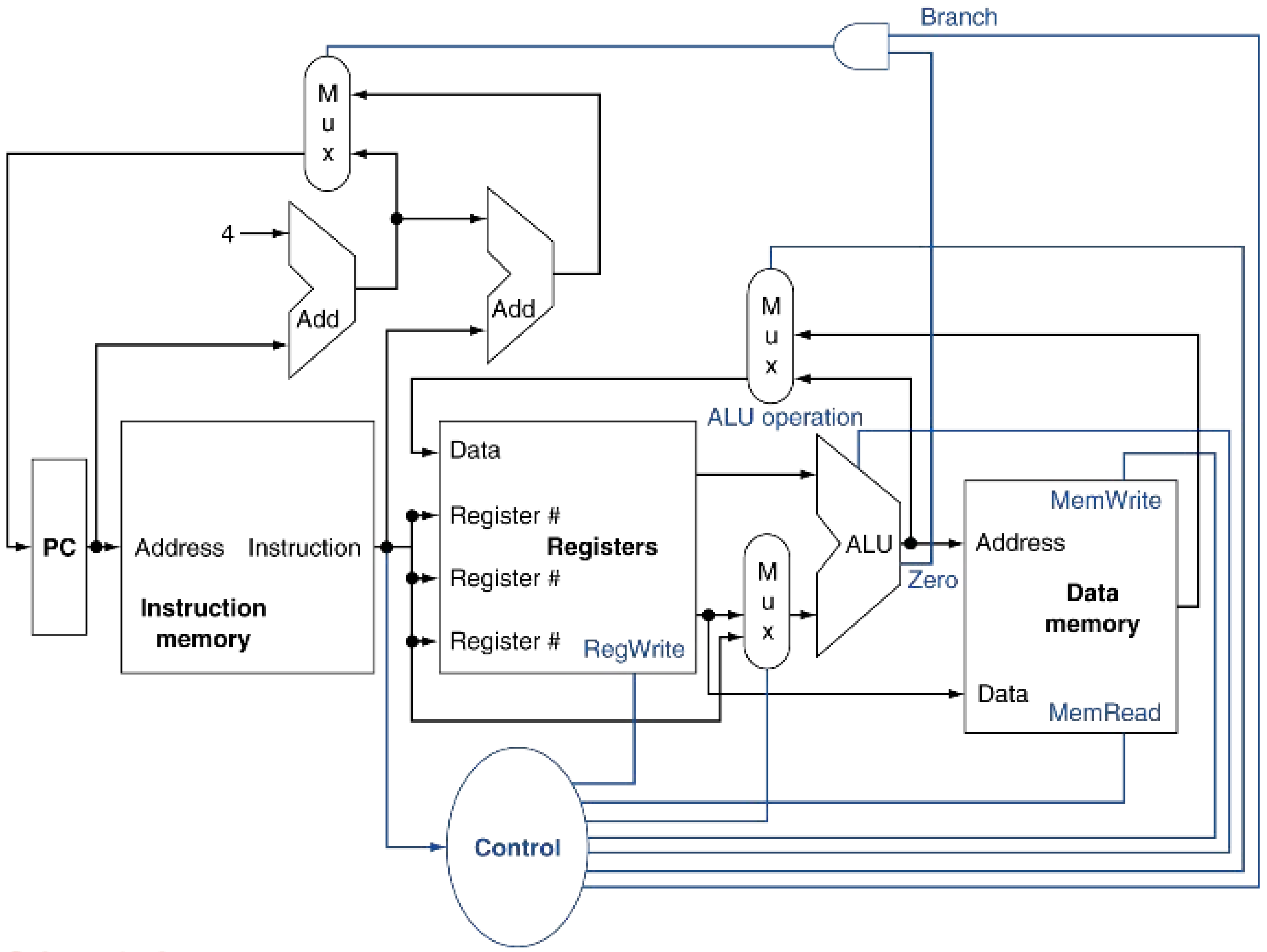
ENGINEERING@SYRACUSE

Multiplexers



- Can’t just join wires together
  - Use multiplexers

Control



Branch
M u x
4
Add
Add
M u x
ALU operation
Data
Register #
Registers
Register #
Register #
RegWrite
PC
Address
Instruction
Instruction memory
M u x
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
Control

ENGINEERING@SYRACUSE

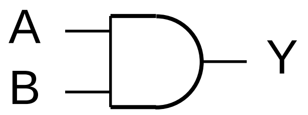
Logic Design Conventions

Logic Design Basics

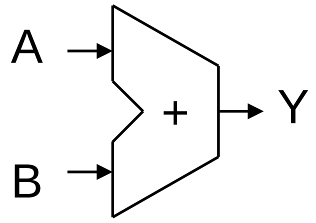
- Information encoded in binary
  - Low voltage = 0, high voltage = 1
  - One wire per bit
  - Multibit data encoded on multiwire buses
- Combinational element
  - Operate on data.
  - Output is a function of input.
- State (sequential) elements
  - Store information.

Combinational Elements

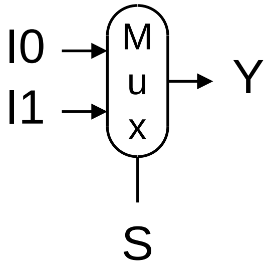
- AND-gate
  - Y = A & B



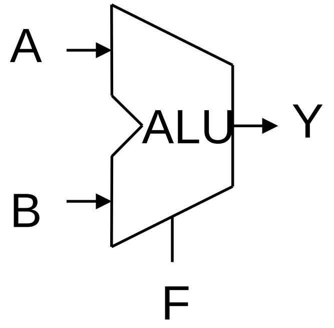
- Adder
  - Y = A + B



- Multiplexer
  - Y = S ? I1 : I0



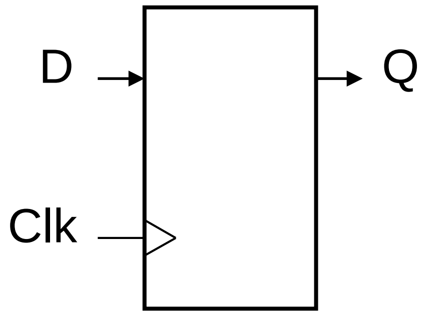
- Arithmetic/Logic Unit
  - Y = F(A, B)

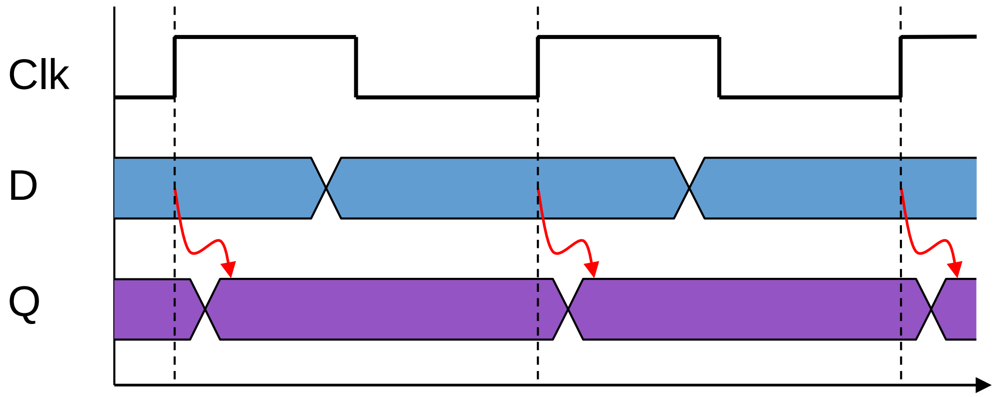


ENGINEERING@SYRACUSE

Sequential Elements

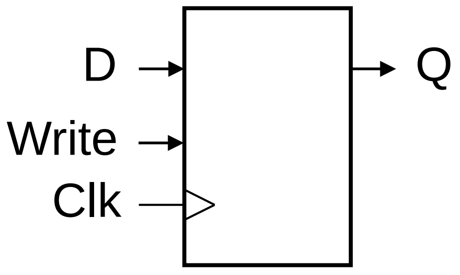
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: updates when Clk changes from 0 to 1

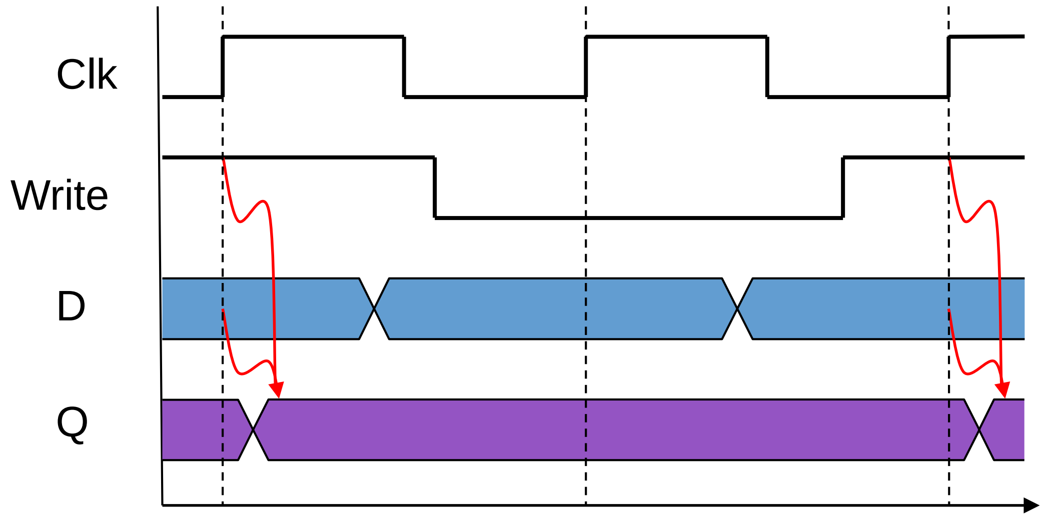




Sequential Elements (cont.)

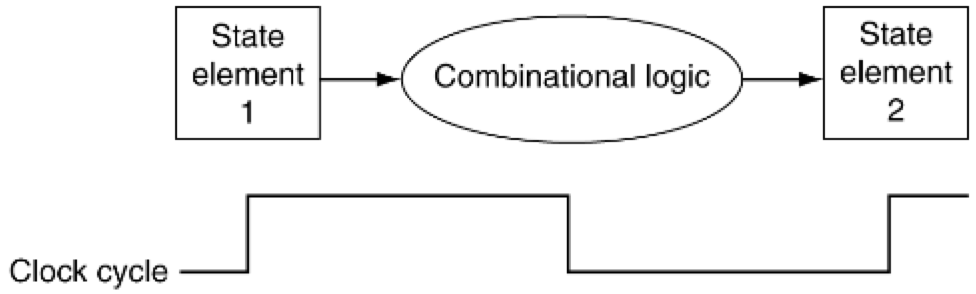
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

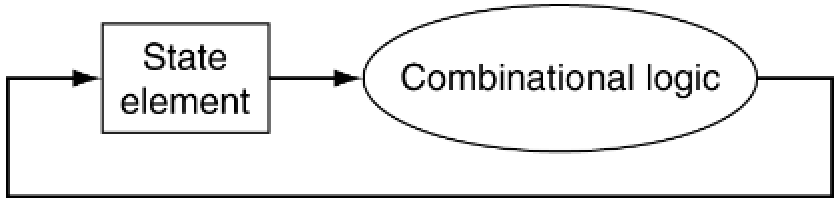




Clocking Methodology

- Combinational logic transforms data during clock cycles.
  - Between clock edges.
  - Input from state elements, output to state element.
  - Longest delay determines clock period.





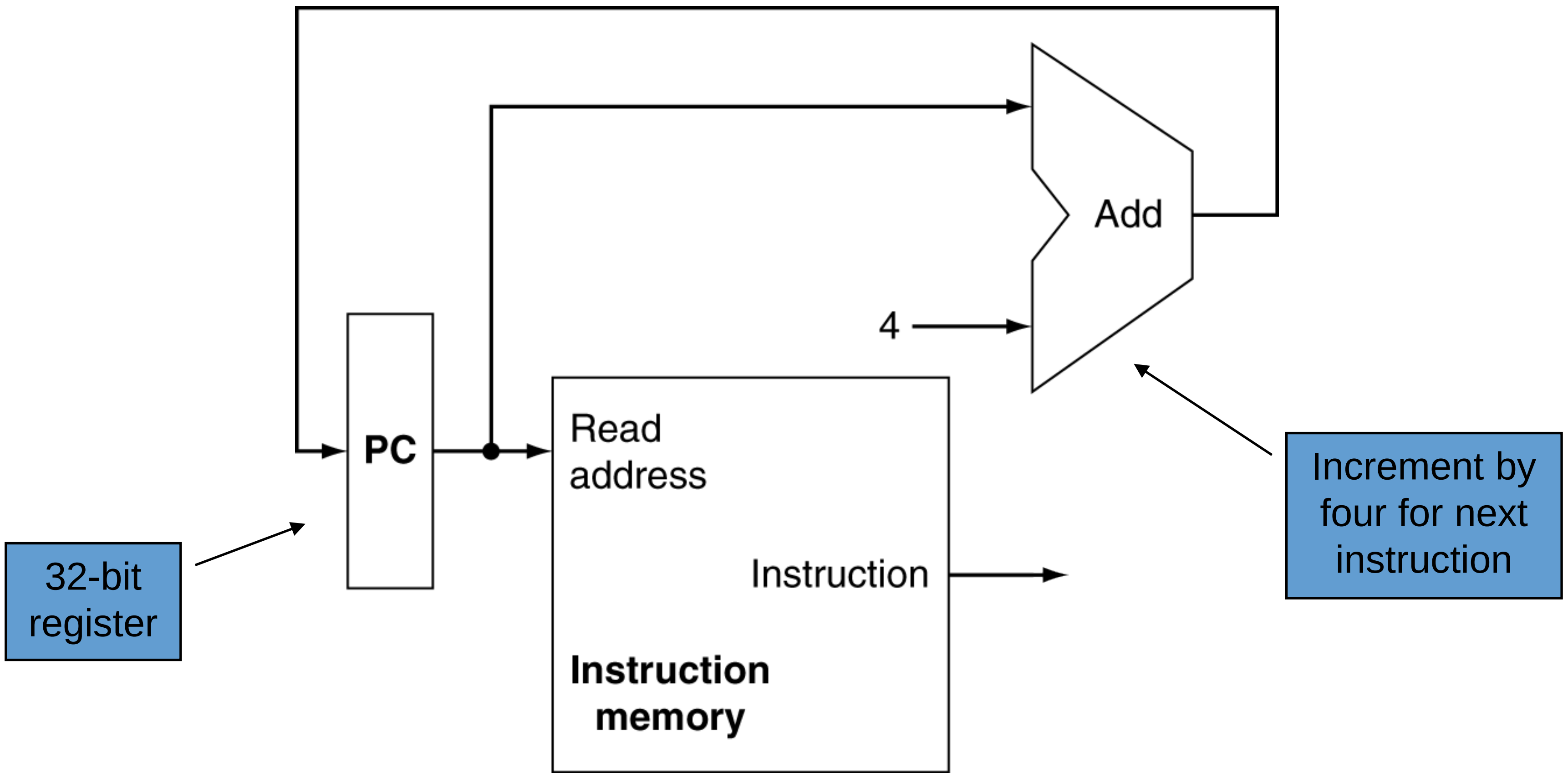
ENGINEERING@SYRACUSE

Building a Datapath

Building a Datapath

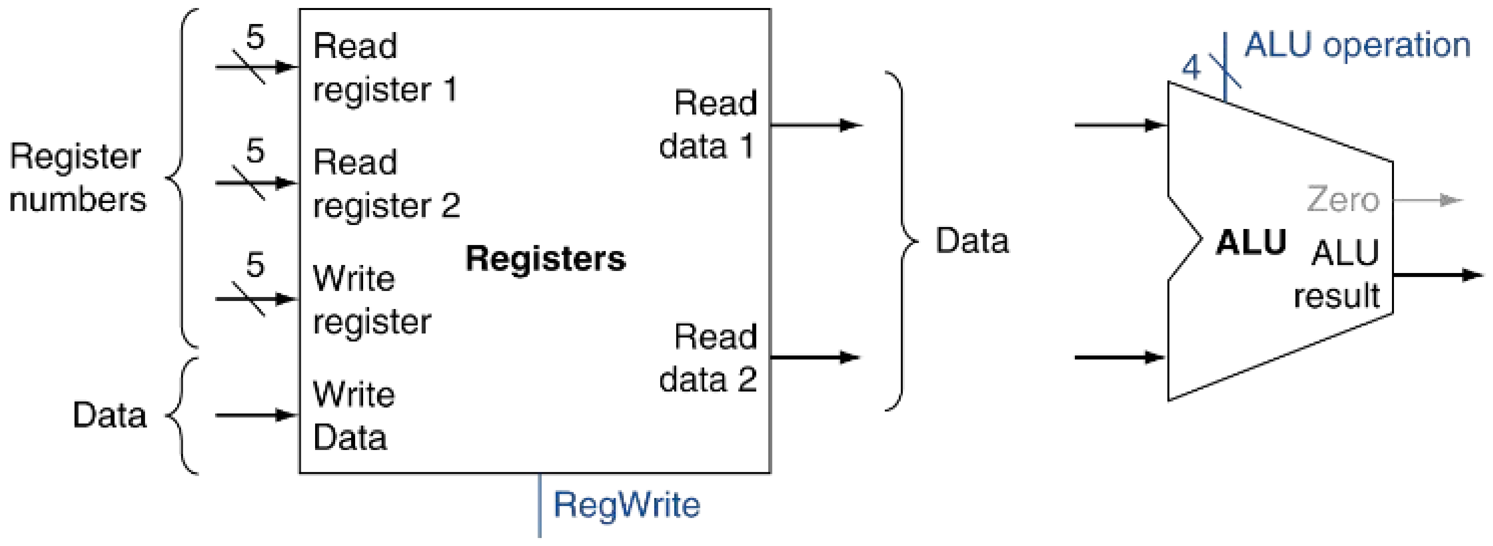
- Datapath.
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, muxs, memories, …
- We will build a Av8 datapath incrementally.
  - Refining the overview design

Instruction Fetch



R-Format Instructions

- Read two register operands.
- Perform arithmetic/logical operation.
- Write register result.



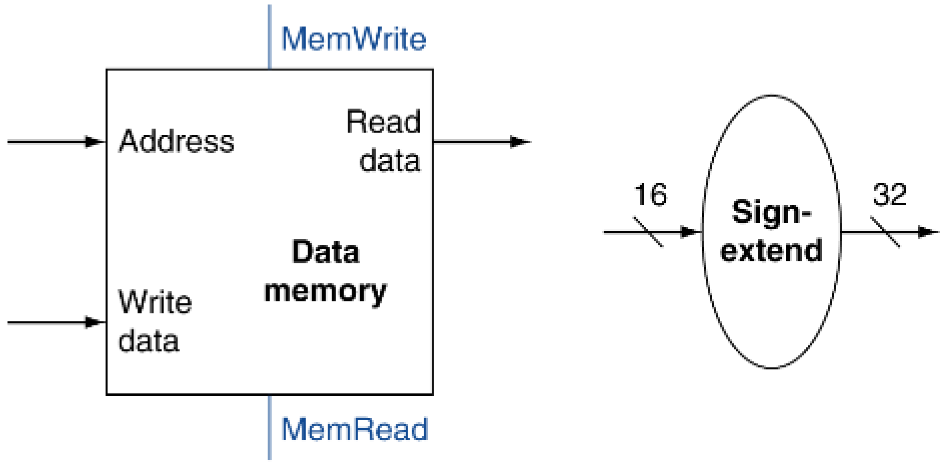
a. Registers

b. ALU

ENGINEERING@SYRACUSE

Load/Store Instructions

- Read register operands.
- Calculate address using 16-bit offset.
  - Use ALU, but sign-extend offset.
- Load: Read memory, and update register.
- Store: Write register value to memory.



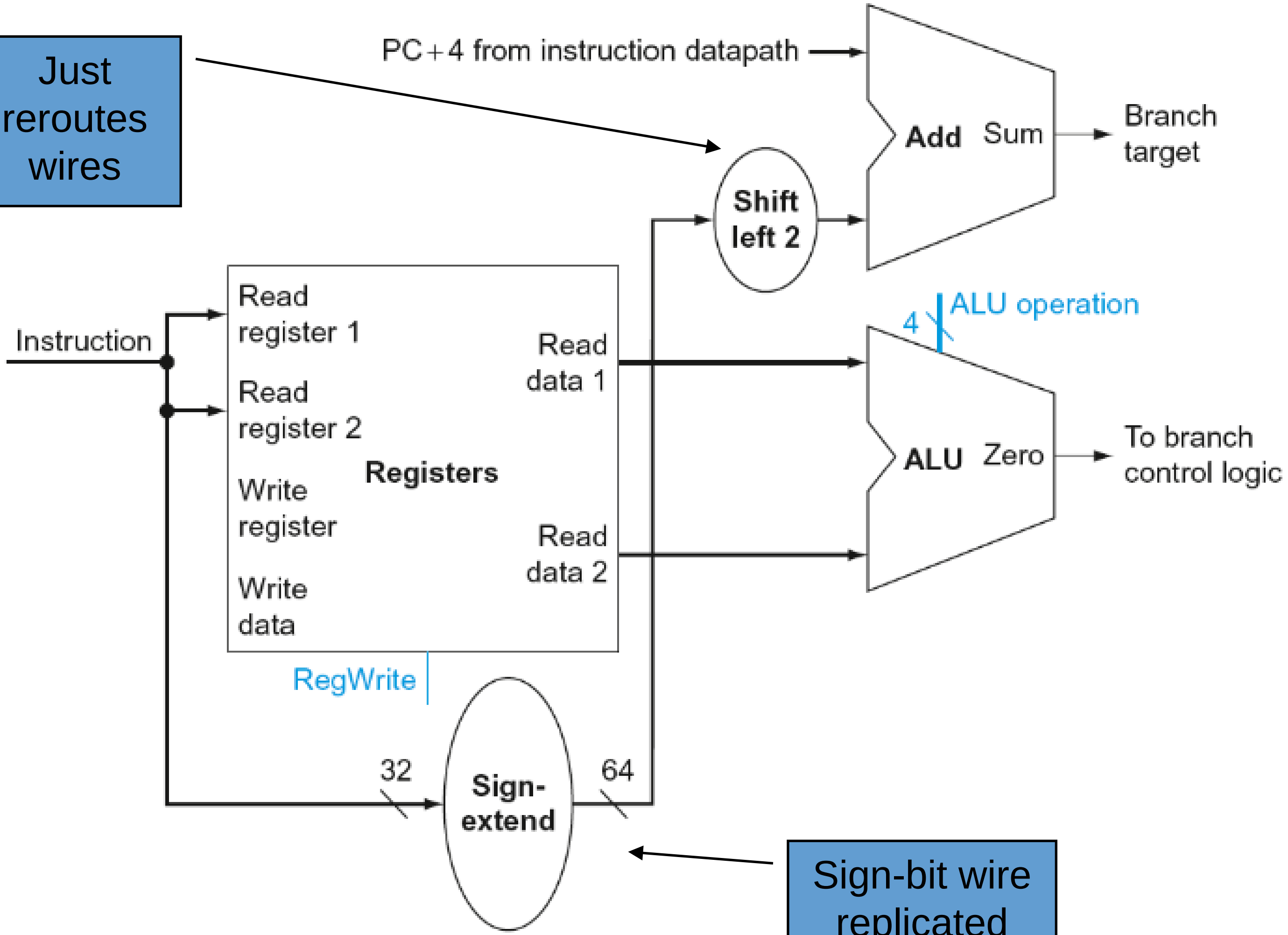
a. Data memory unit b. Sign extension unit

Branch Instructions

- Read register operands.
- Compare operands.
  - Use ALU, subtract and check Zero output.
- Calculate target address.
  - Sign-extend displacement.
  - Shift left two places (word displacement).
  - Add to PC + 4.
    - Already calculated by instruction fetch

Branch Instructions

Just reroutes wires



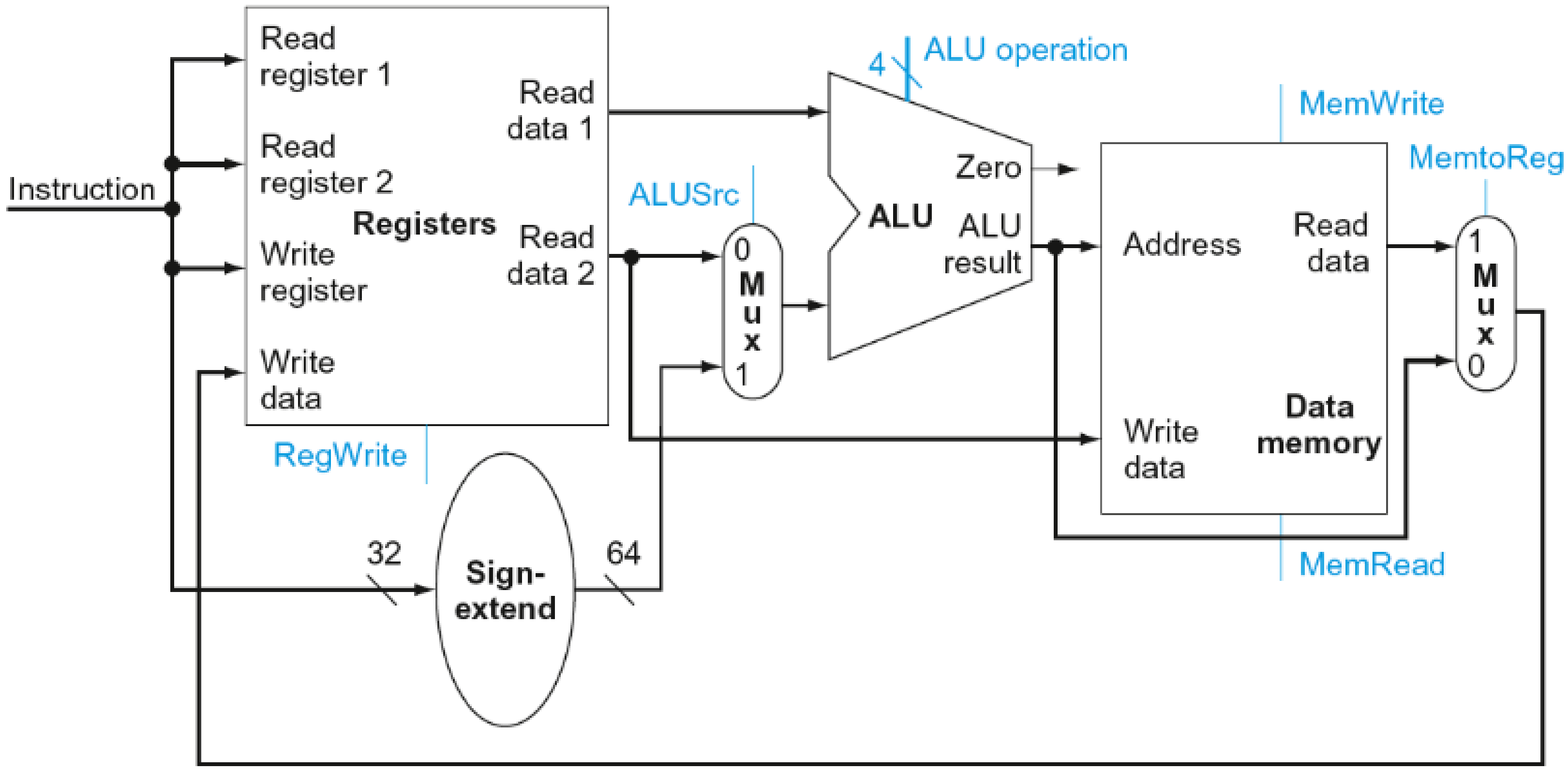
Sign-bit wire replicated

ENGINEERING@SYRACUSE

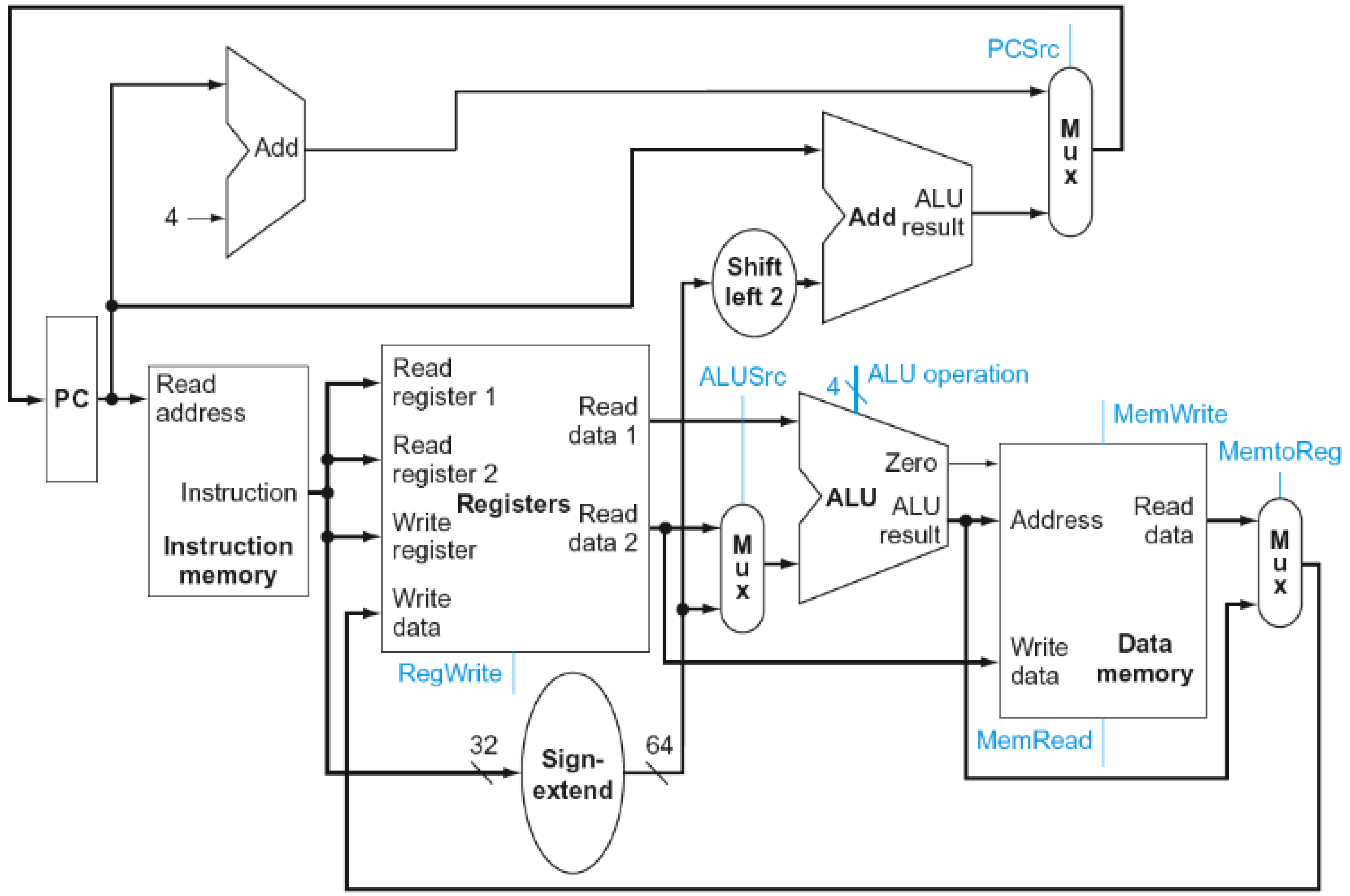
Composing the Elements

- First-cut datapath does an instruction in one clock cycle.
  - Each datapath element can do only one function at a time.
  - Hence, we need separate instruction and data memories.
- Use multiplexers where alternate data sources are used for different instructions.

R-Type/Load/Store Datapath



Full Datapath



ENGINEERING@SYRACUSE

ENGINEERING@SYRACUSE

Simple Implementation

ALU Control

- ALU used for
  - Load/store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode

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

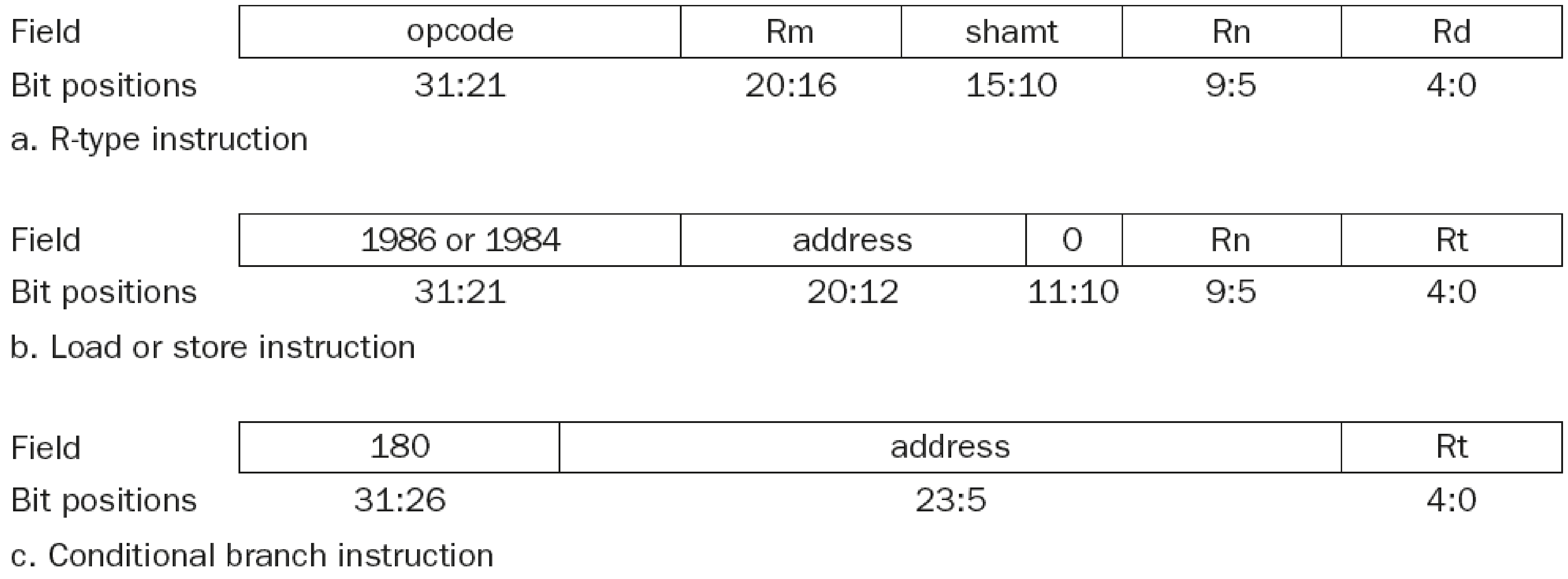
ALU Control (cont.)

- Assume two-bit ALUOp derived from opcode.
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
LDUR	00	load register	XXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXX	pass input b	0111
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		ORR	100101	OR	0001

The Main Control Unit

- Control signals derived from instruction



Field	opcode	Rm	shamt	Rn	Rd
Bit positions	31:21	20:16	15:10	9:5	4:0

a. R-type instruction

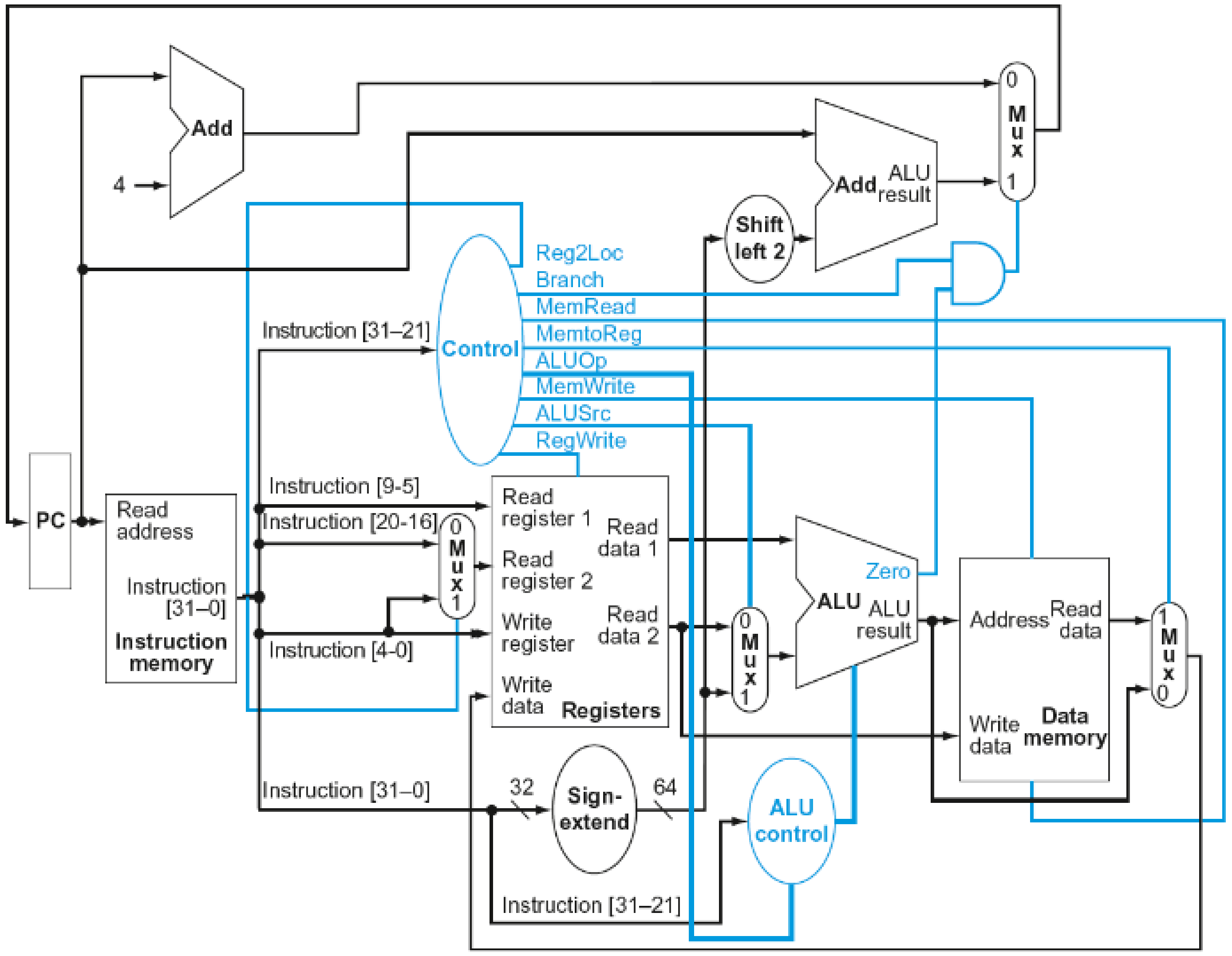
Field	1986 or 1984	address	0	Rn	Rt
Bit positions	31:21	20:12	11:10	9:5	4:0

b. Load or store instruction

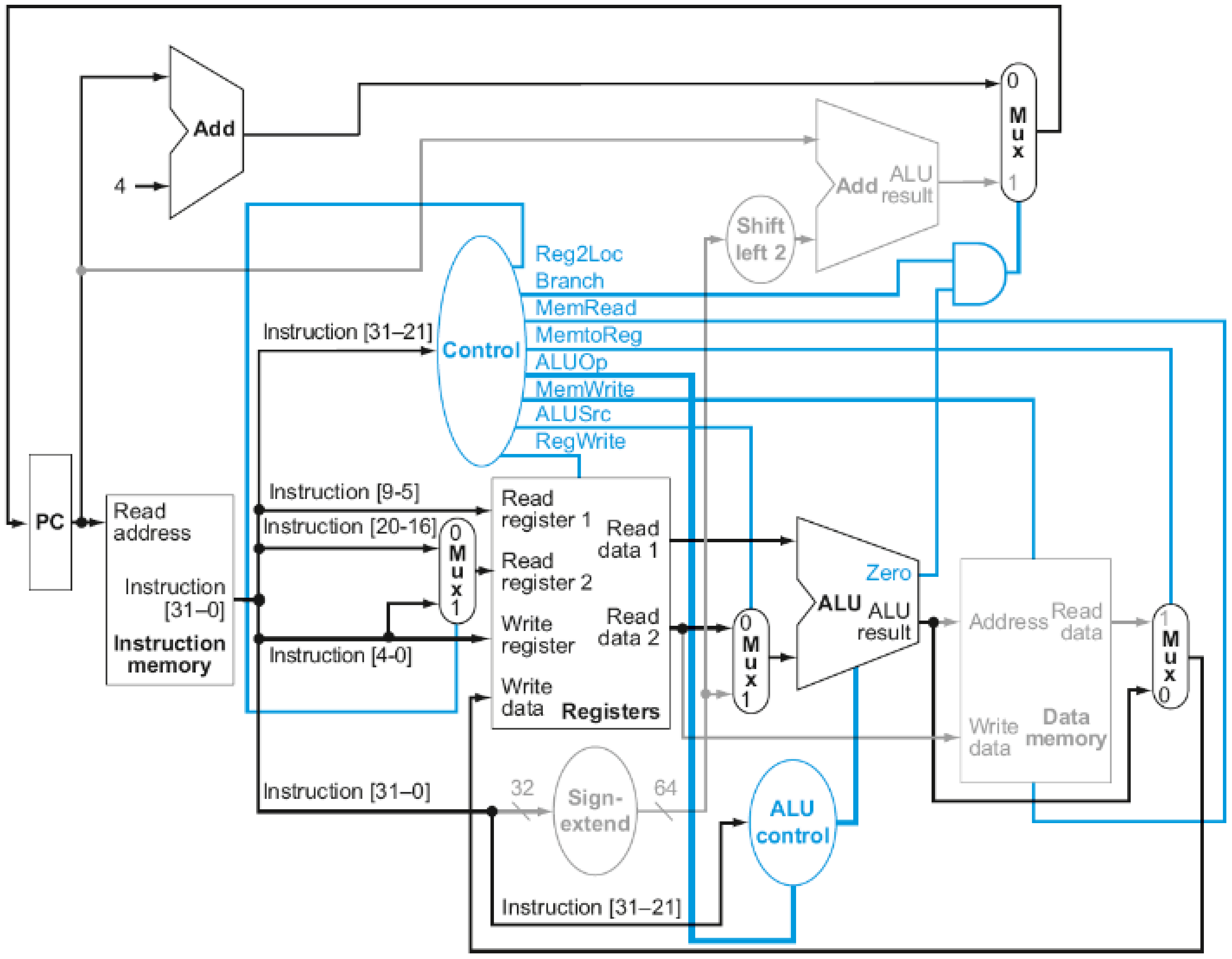
Field	180	address	Rt
Bit positions	31:26	23:5	4:0

c. Conditional branch instruction

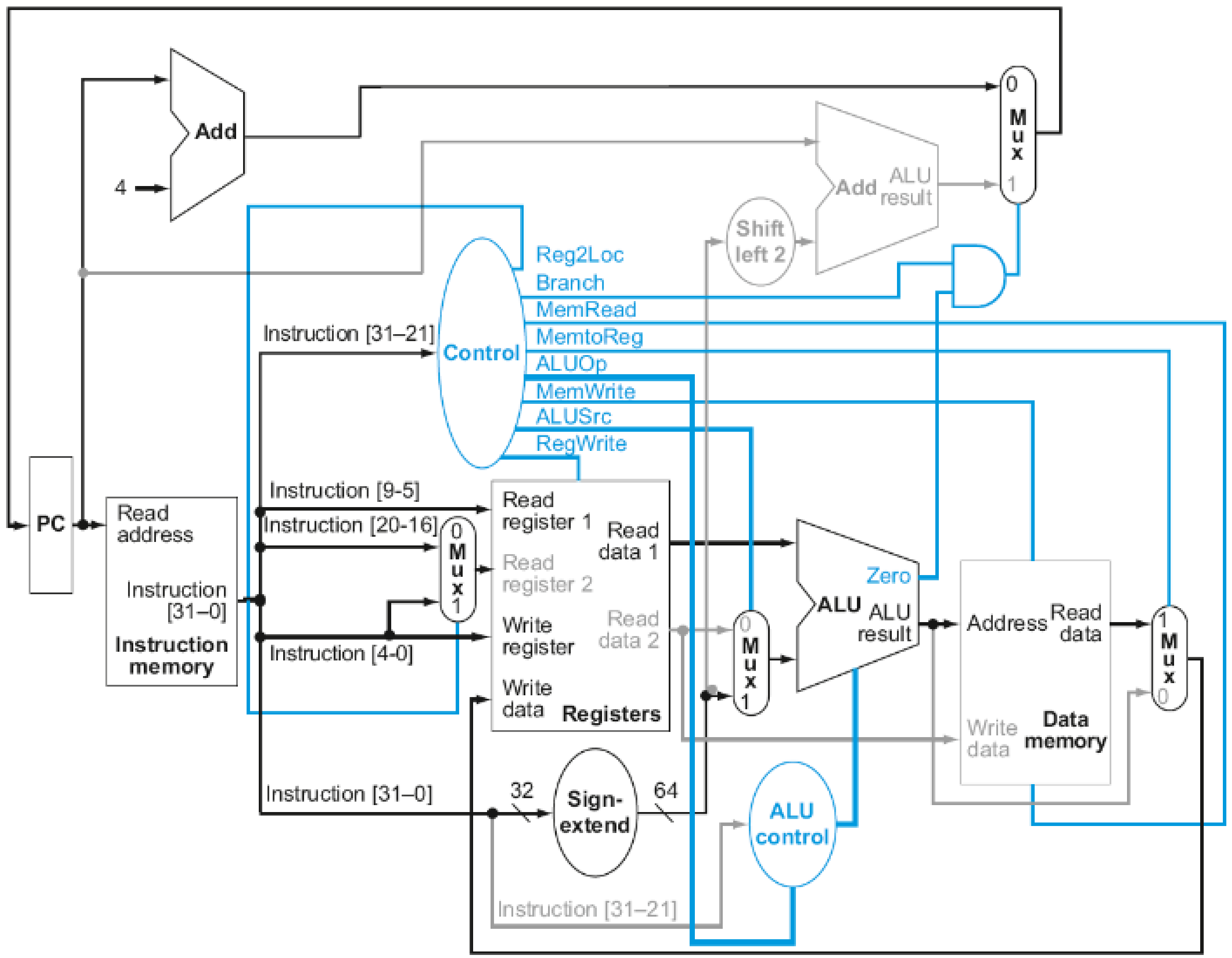
Datapath with Control



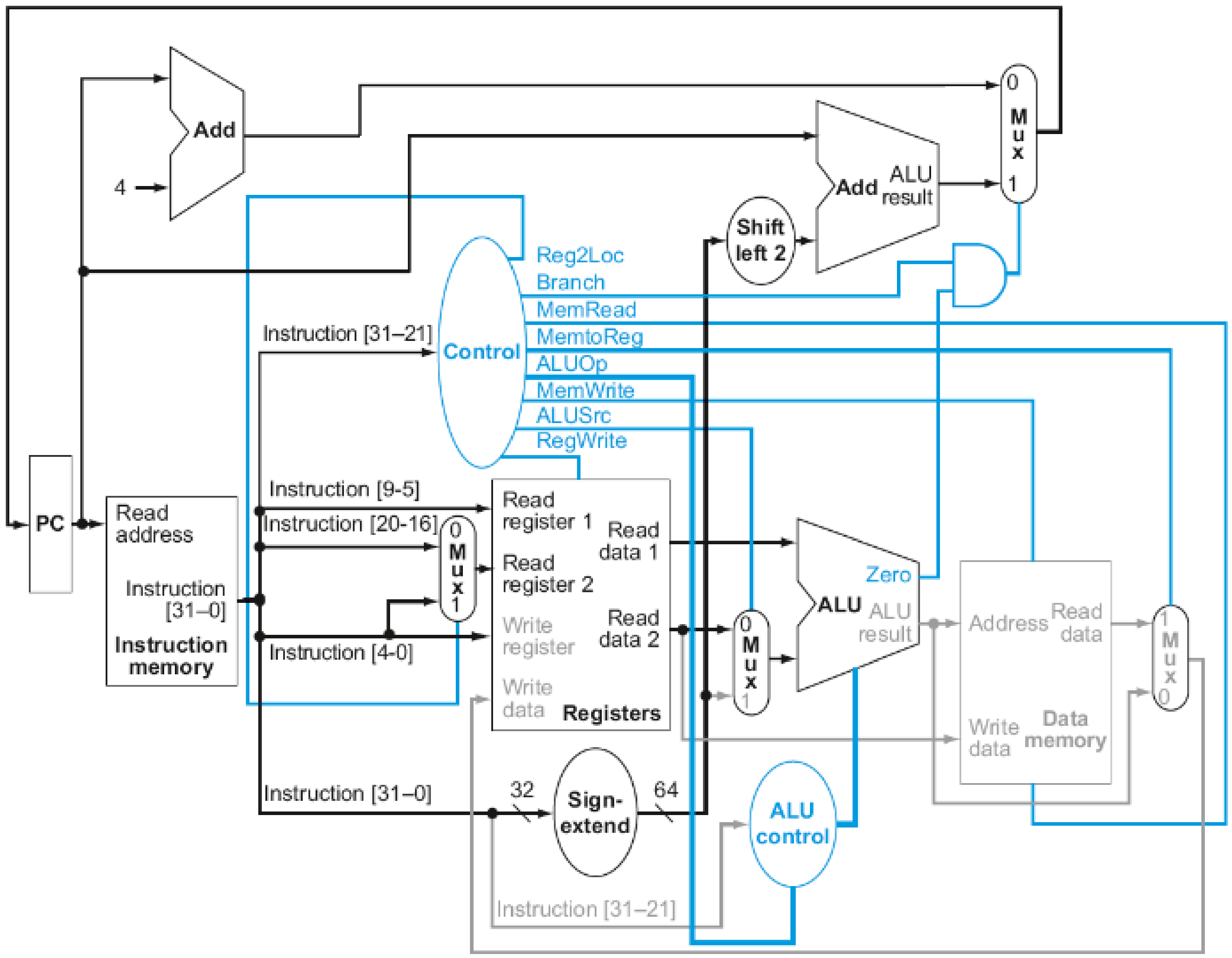
R-Type Instruction



Load Instruction



CBZ Instruction



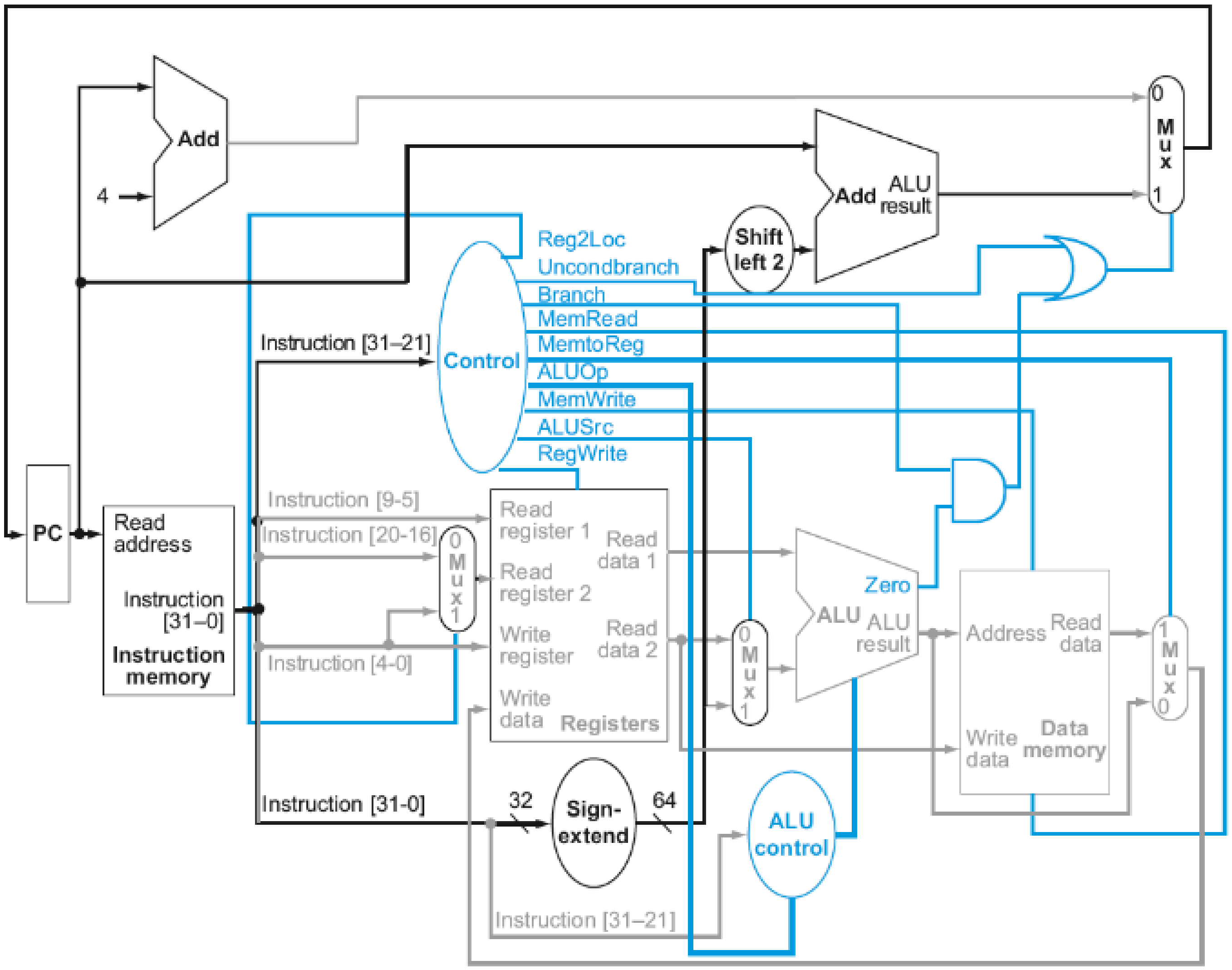
ENGINEERING@SYRACUSE

Implementing Unconditional Branch

Jump	2	address
	31:26	25:0

- Jump uses word address.
- Update PC with concatenation of:
  - Top four bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode.

Datapath with B Added



ENGINEERING@SYRACUSE

Performance Issues

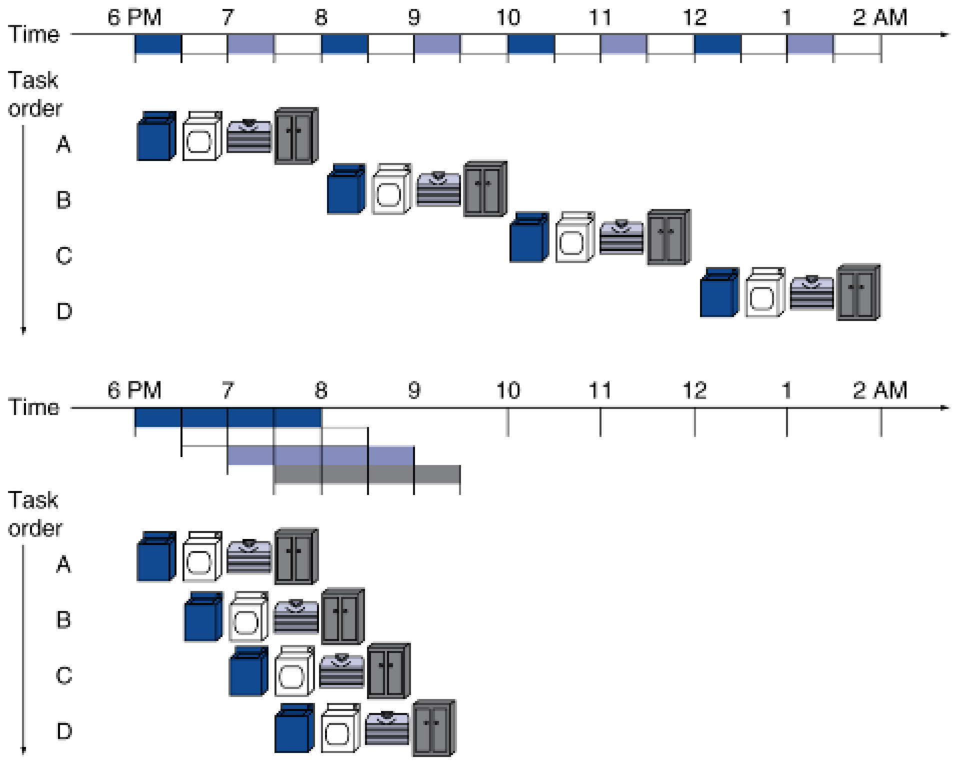
- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory $\rightarrow$ register file $\rightarrow$ ALU $\rightarrow$ data memory $\rightarrow$ register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining
  - BTW: $CPUtime = IC.CPI.CCT$ $\quad MIPS = \frac{IC}{CPU\ time\ in\ seconds} \times \frac{1}{10^6}$

ENGINEERING@SYRACUSE

Pipelining

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance.



- Four loads:
  - Speed up = 8/3.5 = 2.3
- Nonstop:
  - Speed up = 2n/0.5n + 1.5 ≈ 4 = number of stages

Av8 Pipeline

Five stages, one step per stage

1. IF: Instruction fetch from memory
2. ID: Instruction decode and register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

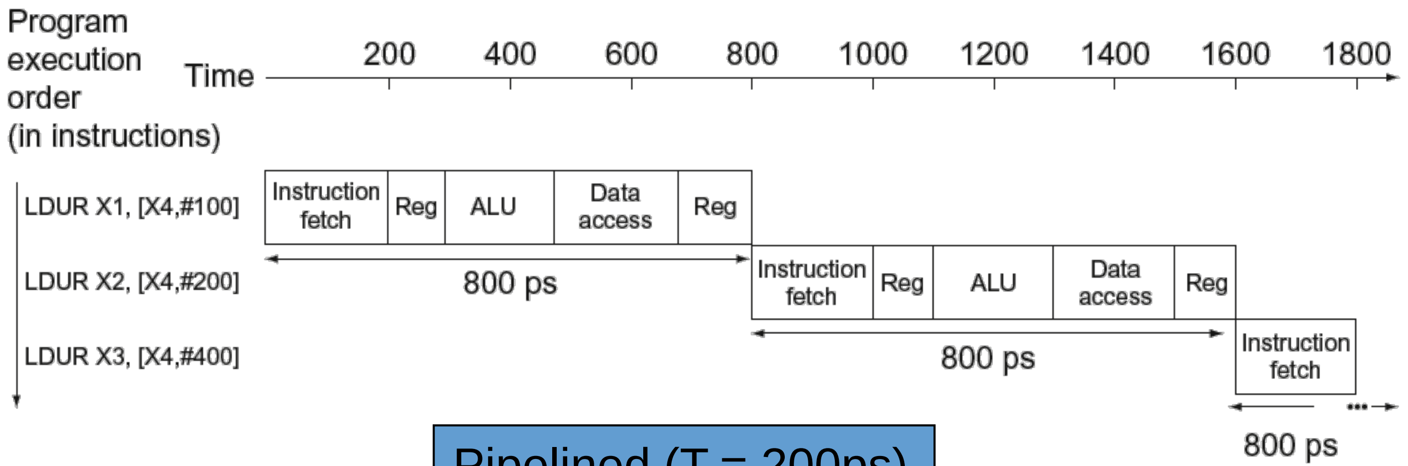
Pipeline Performance

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

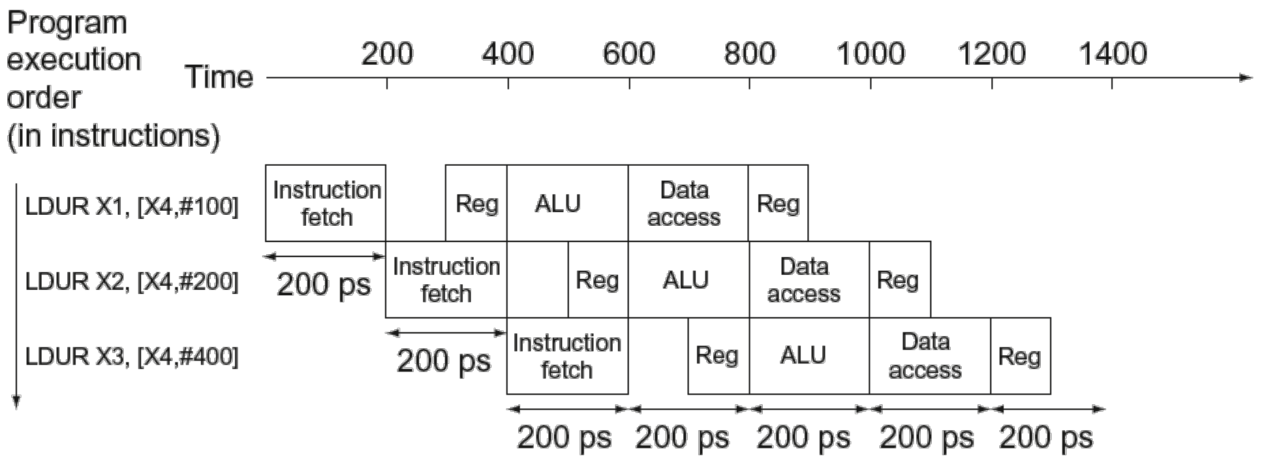
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
LDUR	200ps	100 ps	200ps	200ps	100 ps	800ps
STUR	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
CBZ	200ps	100 ps	200ps			500ps

Pipeline Performance (cont.)

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



ENGINEERING@SYRACUSE

Pipeline Speedup

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

Pipelining and ISA Design

- Av8 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 third stage, access memory in fourth stage
  - Alignment of memory operands.
    - Memory access takes only one cycle.

ENGINEERING@SYRACUSE

Pipelining Hazards

Hazards

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

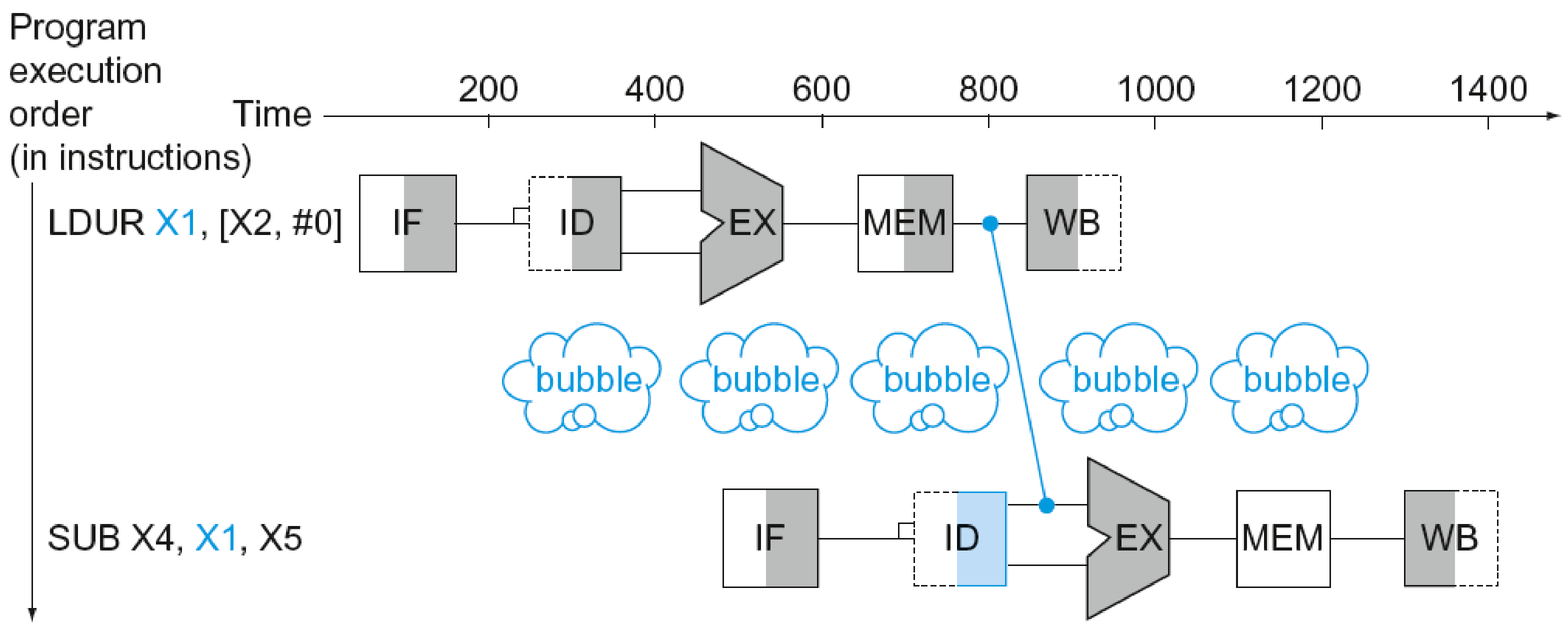
Structure Hazards

- Conflict for use of a resource.
- In Av8 pipeline with a single memory.
  - Load/store requires data access.
  - Instruction fetch would have to *stall* for that cycle.
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories.
  - Or separate instruction/data caches

Data Hazards

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

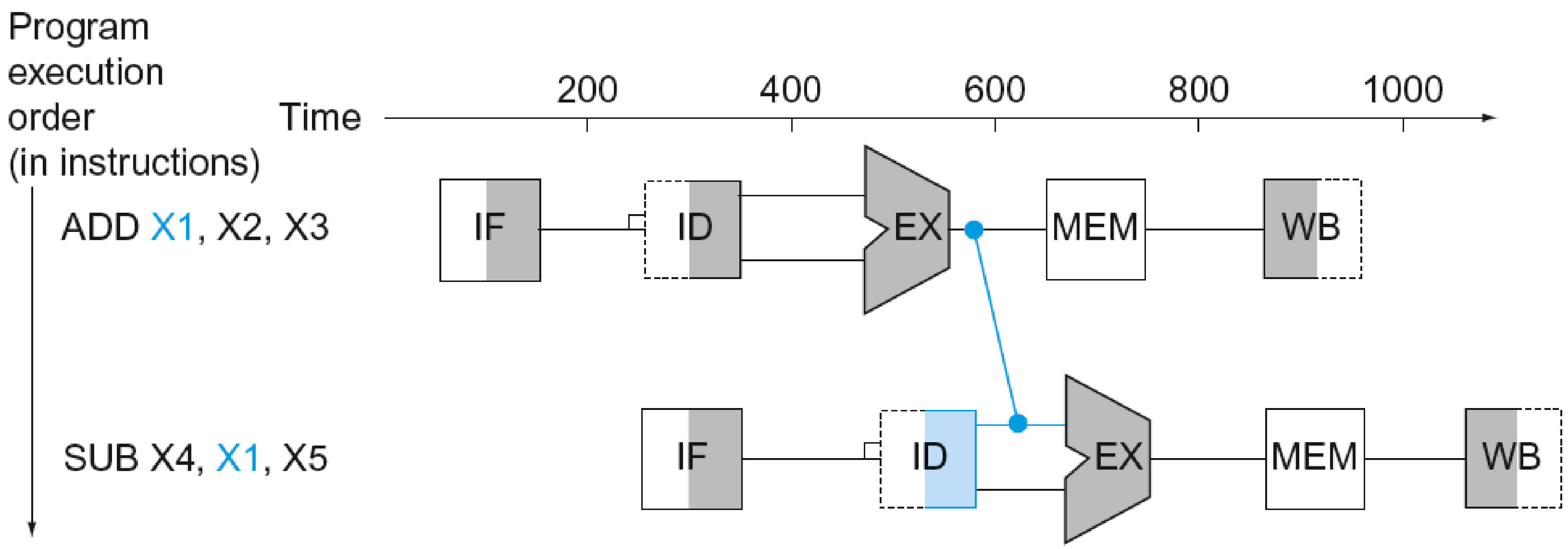
  -
    ```
    ADD     X19, X0, X1
    SUB     X2, X19, X3
    ```



ENGINEERING@SYRACUSE

Forwarding (aka Bypassing)

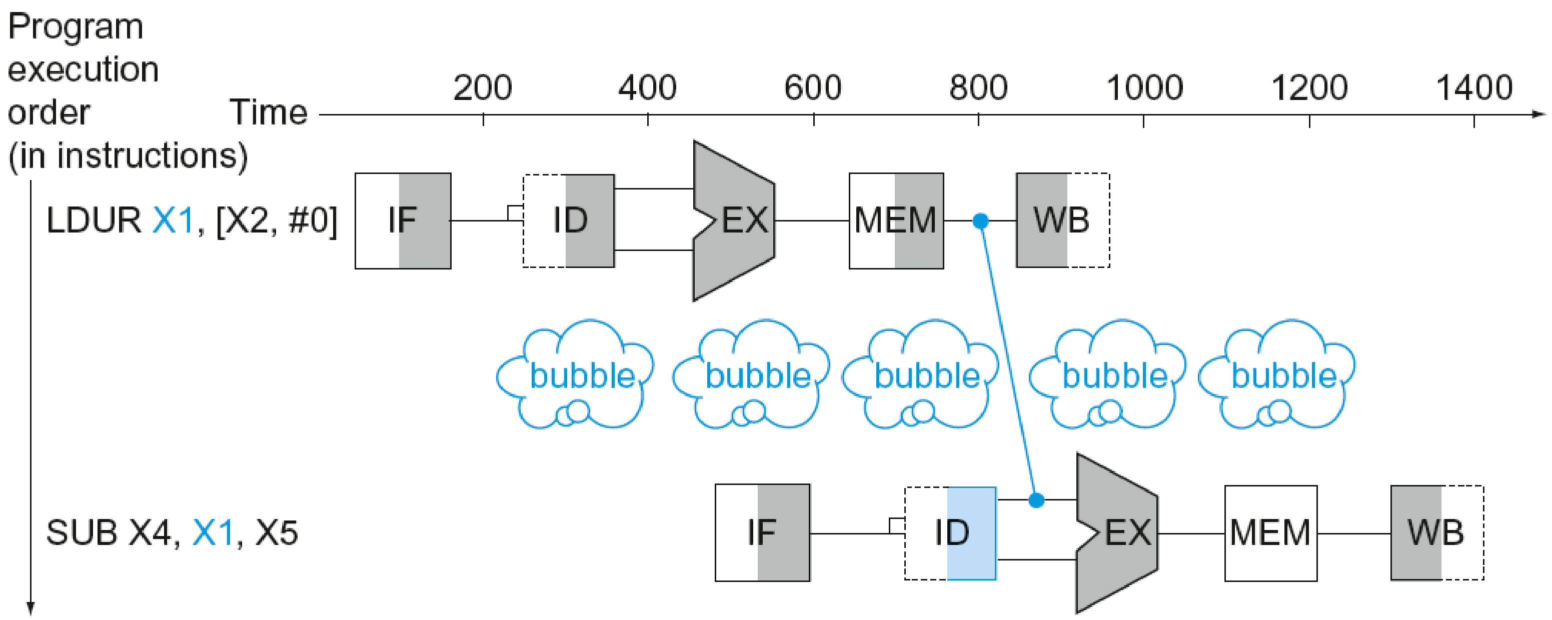
- Use result when it is computed.
  - Don’t wait for it to be stored in a register.
  - Requires extra connections in the datapath.



ENGINEERING@SYRACUSE

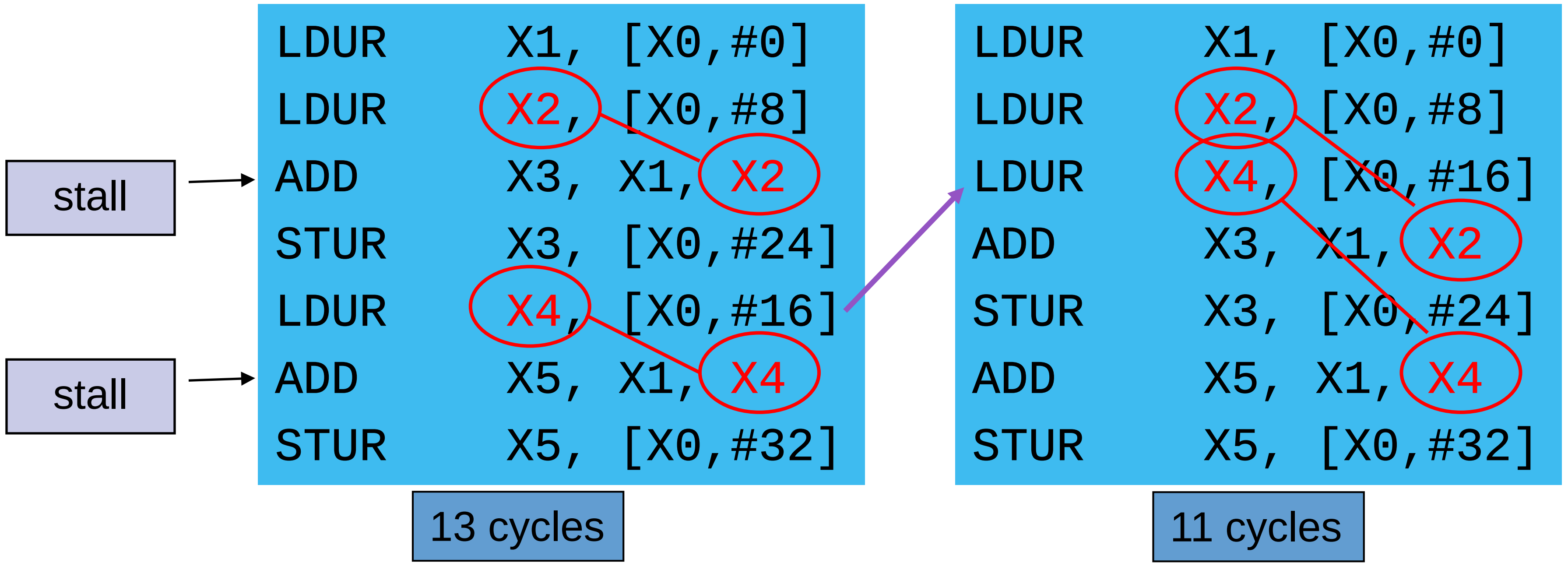
Load-Use Data Hazard

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



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction.
- C code for A = B + E; C = B + F;



```
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]
```

```
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]
```

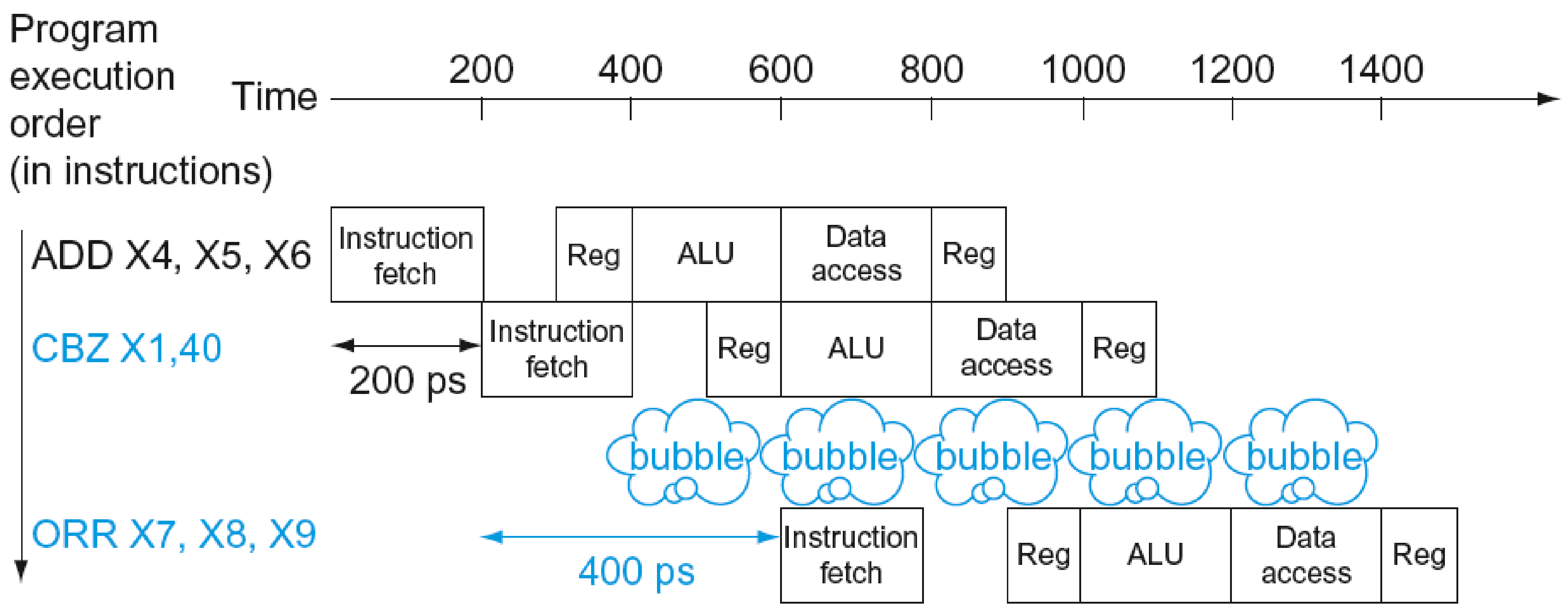
ENGINEERING@SYRACUSE

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 Av8 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 Av8 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 refetching, and update history.

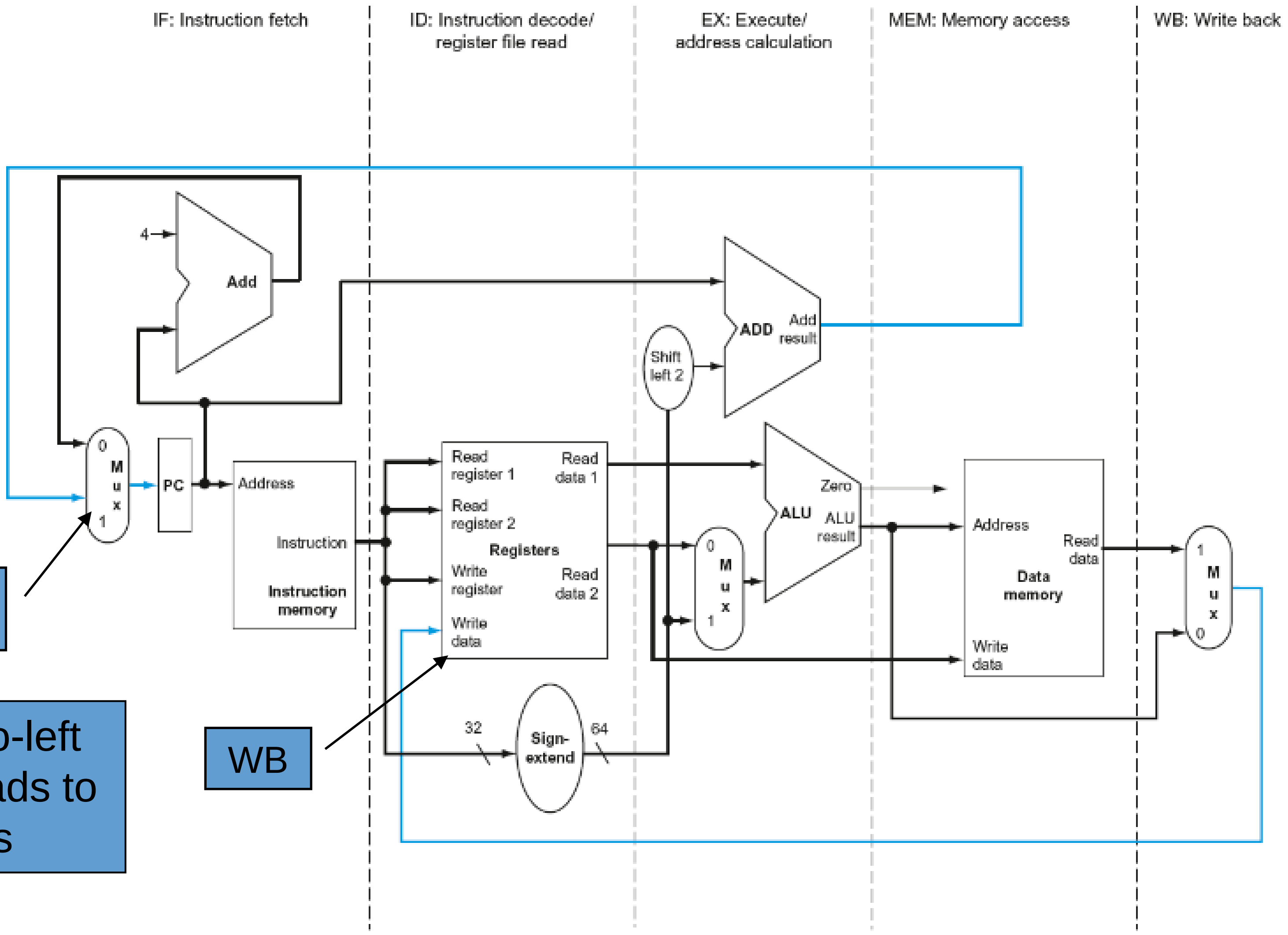
Pipeline Summary

- 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.

ENGINEERING@SYRACUSE

Pipelined Datapath

Av8 Pipelined Datapath



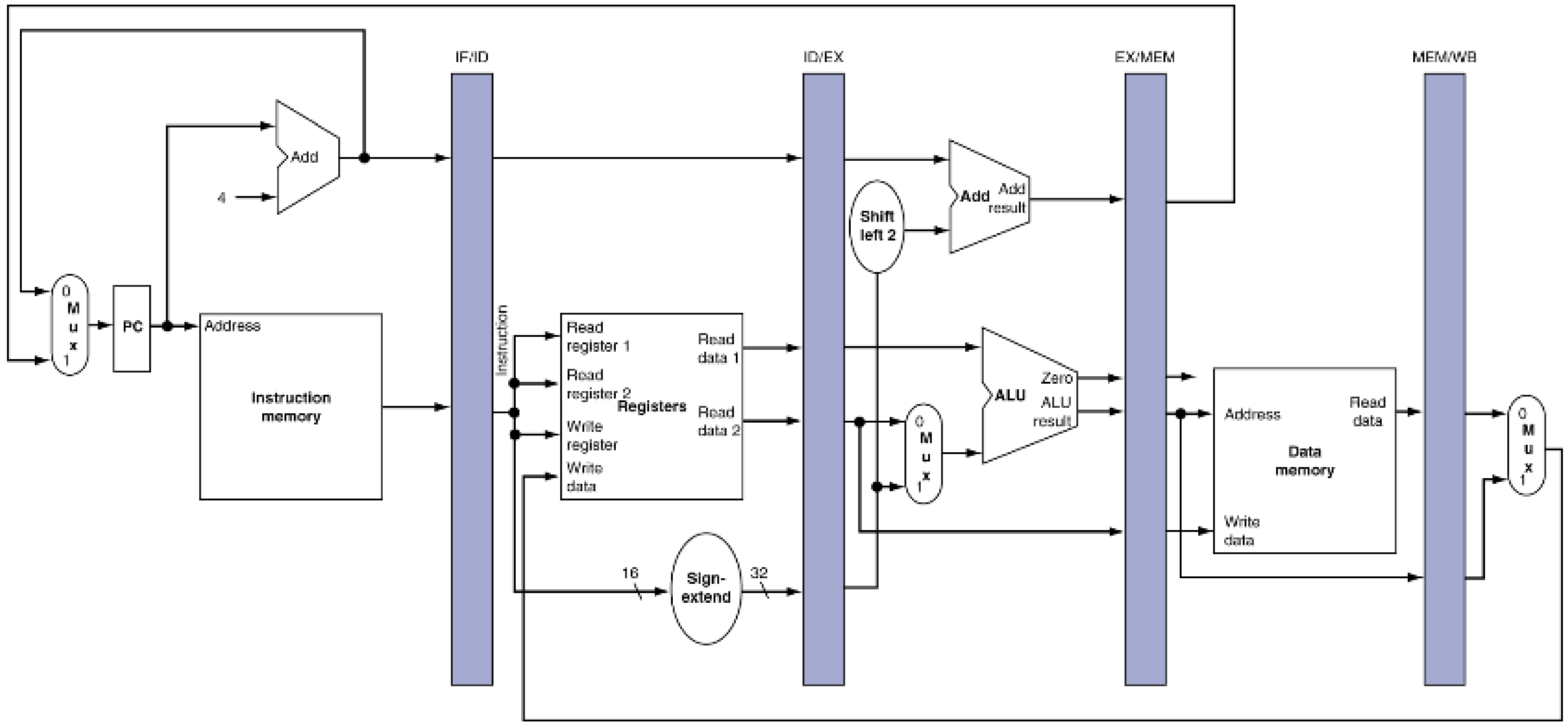
MEM

WB

Right-to-left flow leads to hazards

Pipeline Registers

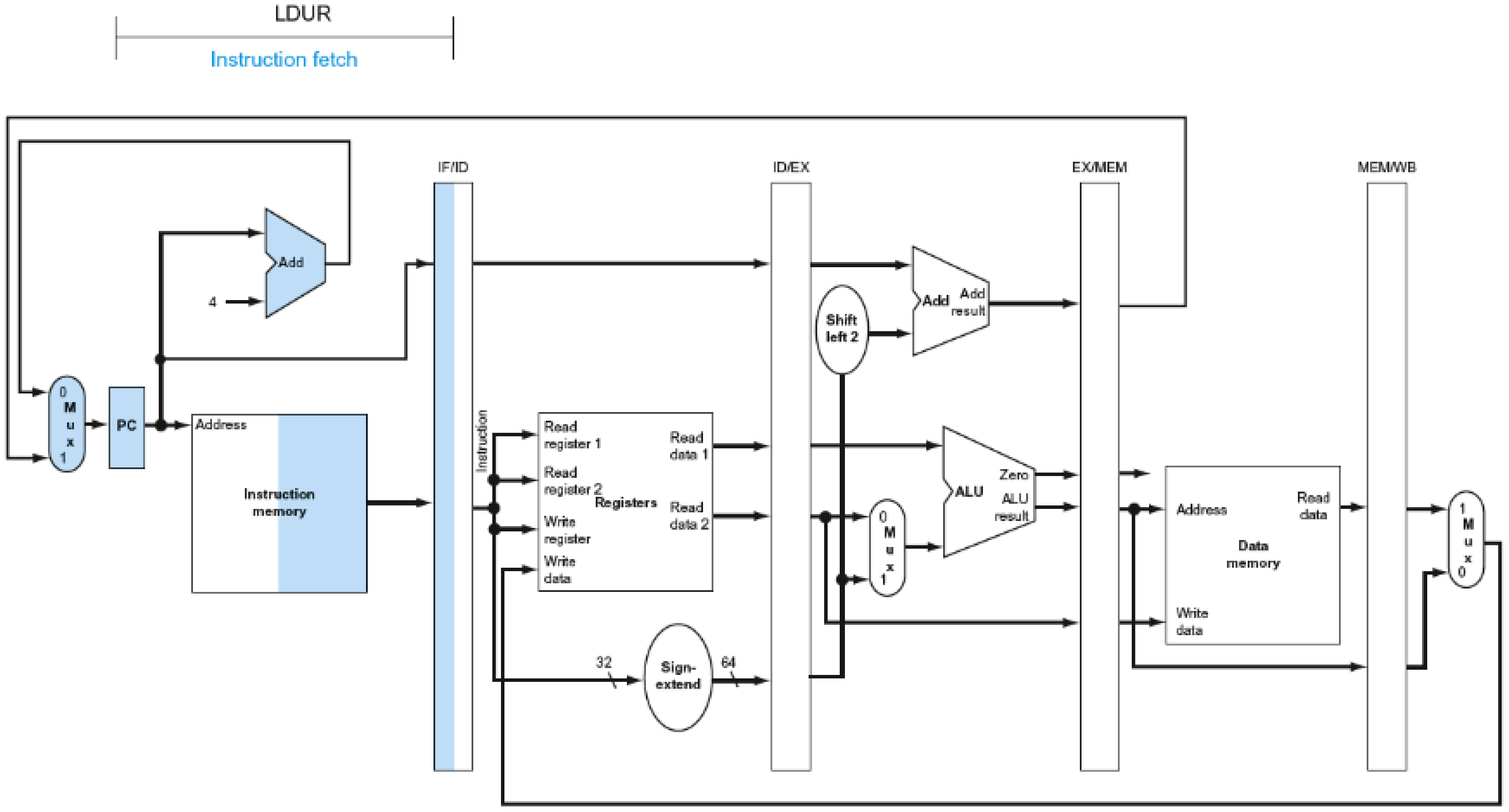
- Need registers between stages
  - To hold information produced in previous cycle



Pipeline Operation

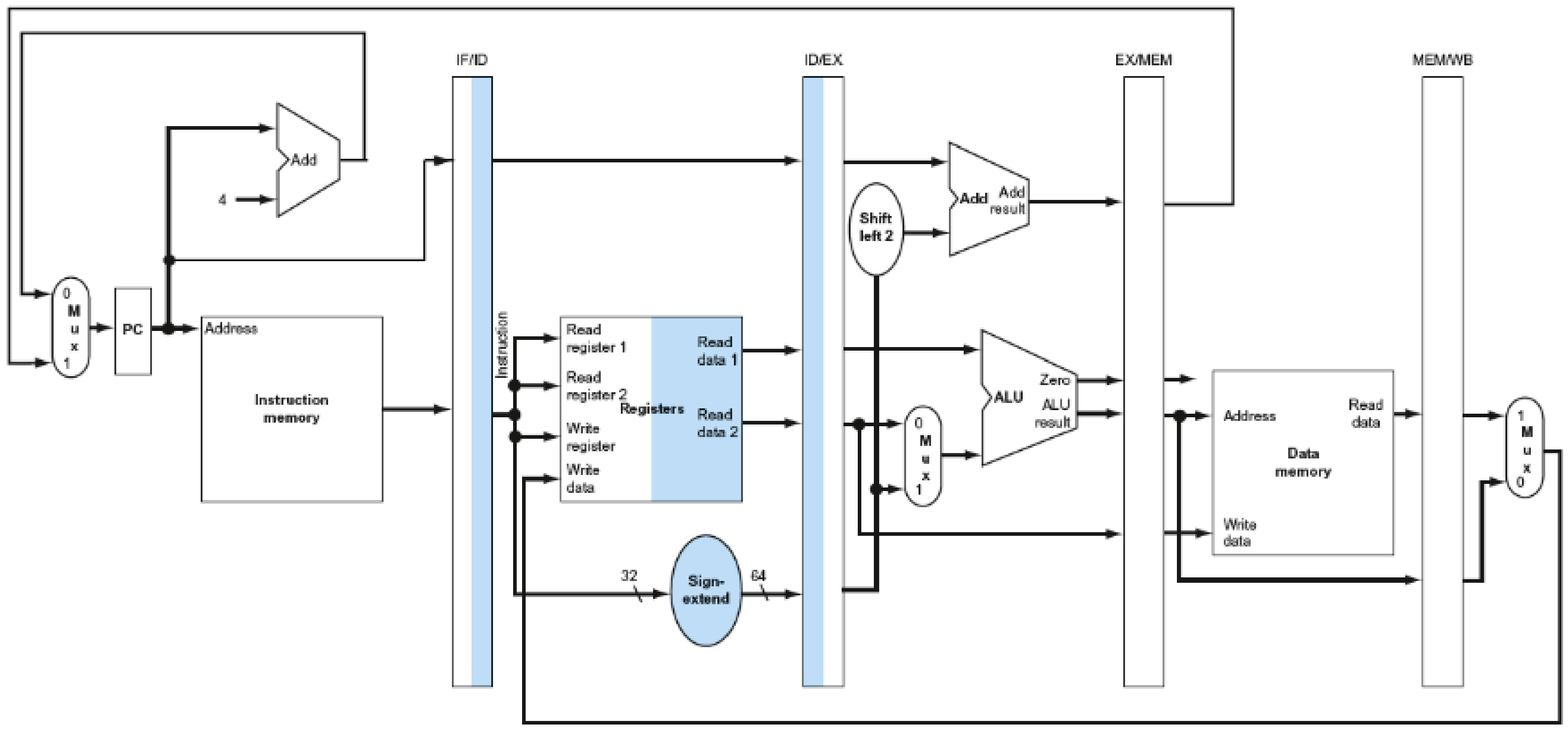
- Cycle-by-cycle flow of instructions through the pipelined datapath.
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - C.f. “multi-clock-cycle” diagram
    - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams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

IF for Load, Store, …

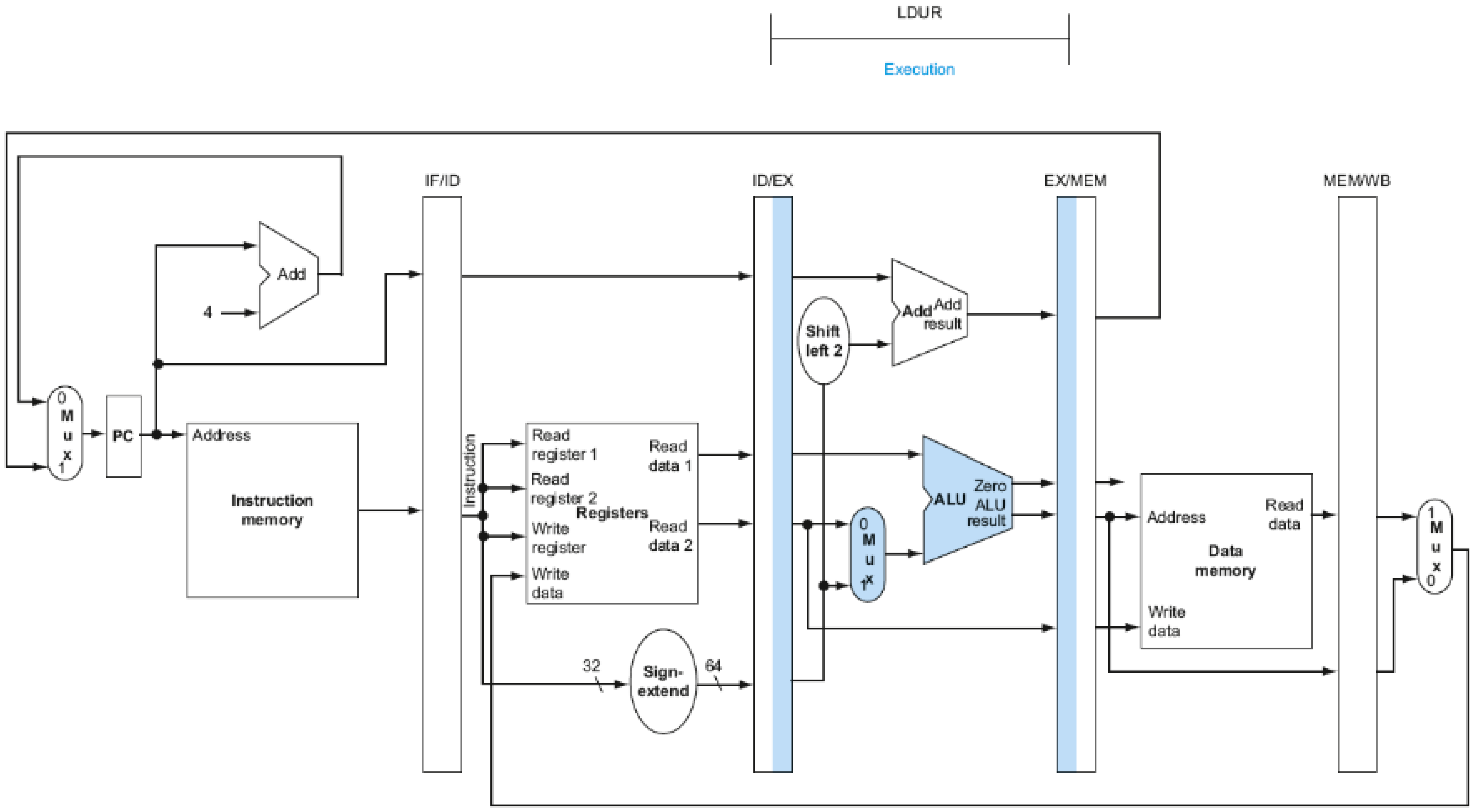


ID for Load, Store, …

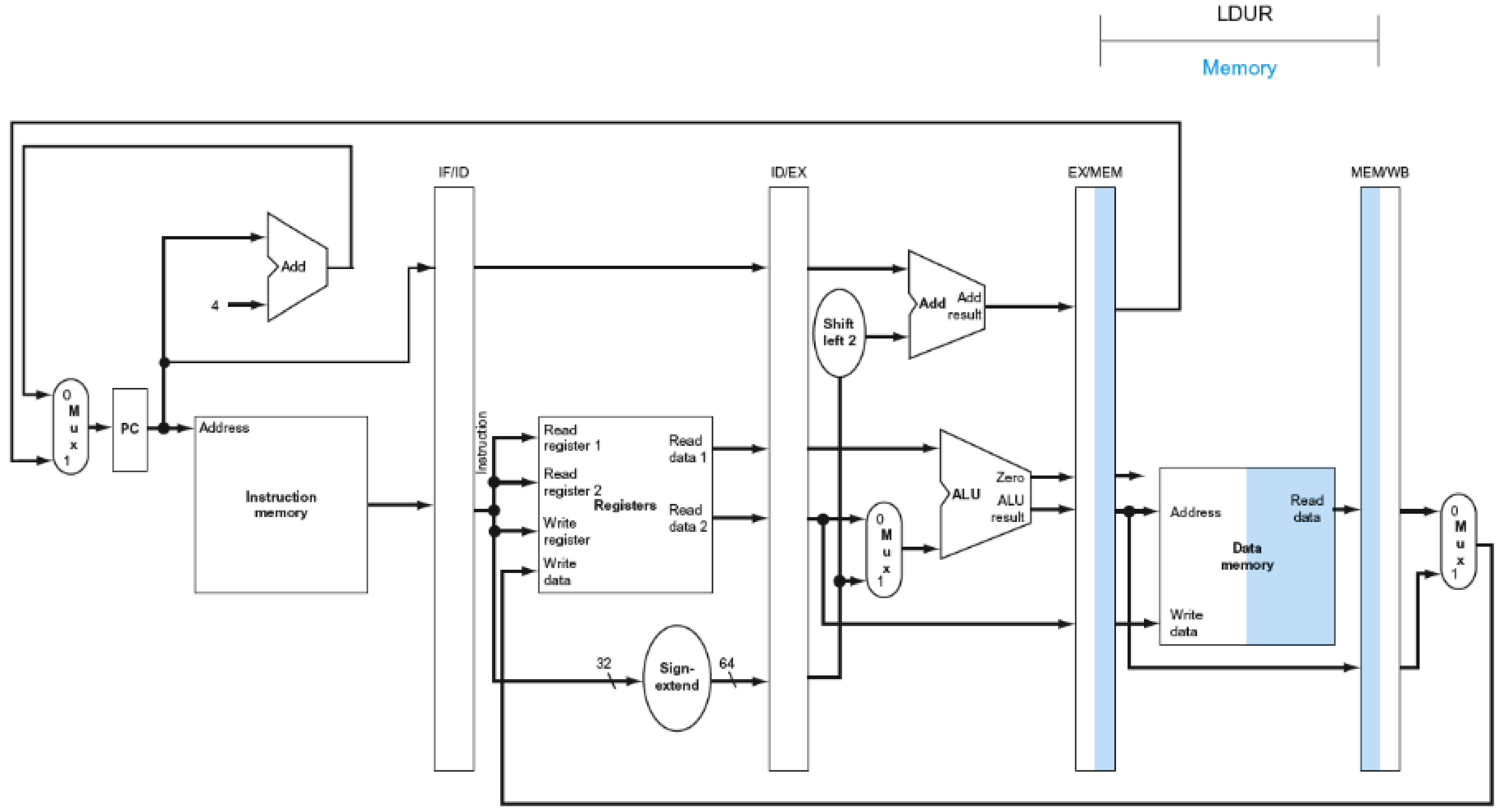
LDUR
Instruction decode



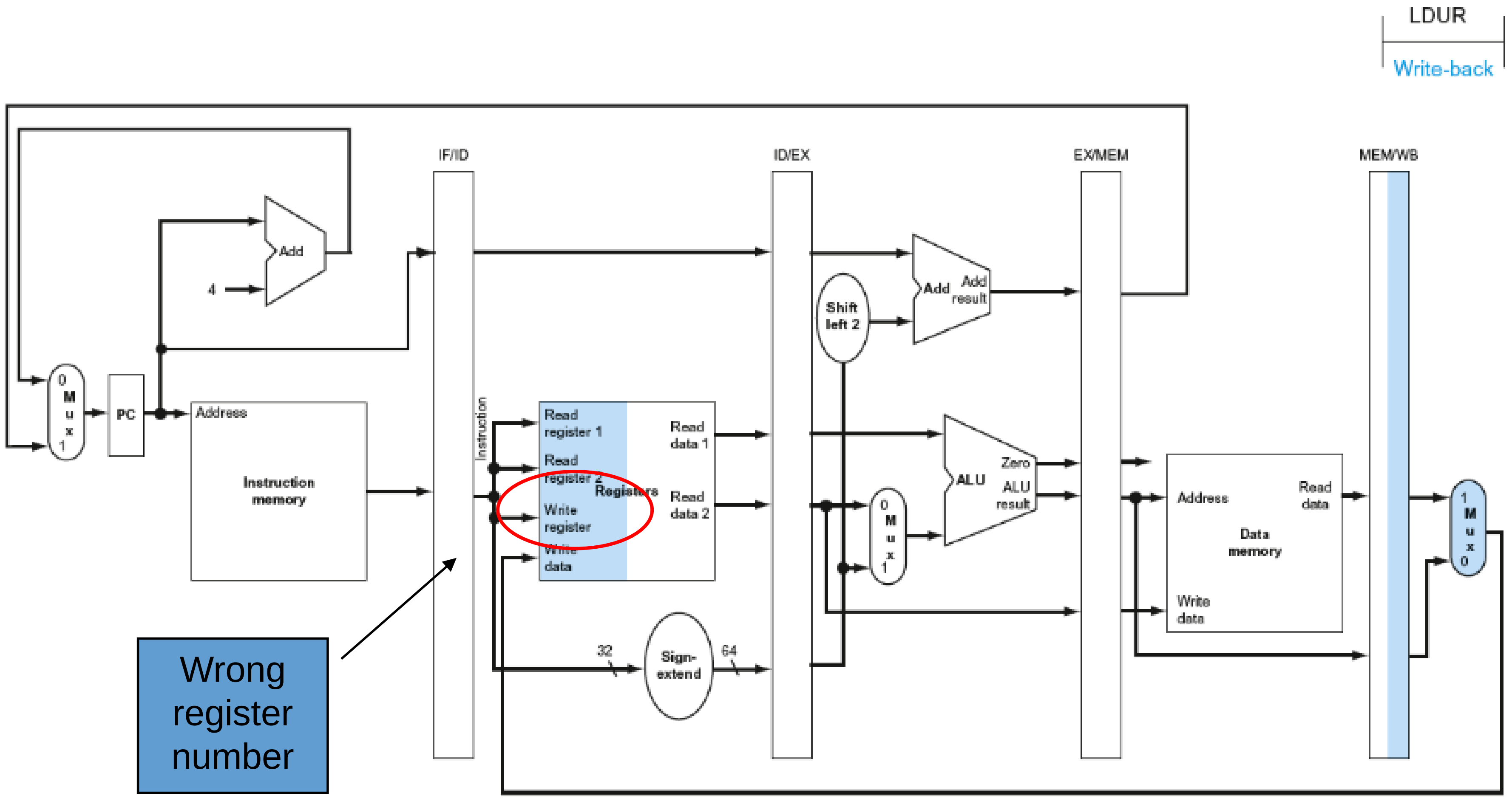
EX for Load


LDUR
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0 M u x 1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
32
Sign-extend
64
Shift left 2
Add
Add result
0 M u x 1
ALU
Zero
ALU result
Address
Data memory
Write data
Read data
1 M u x 0

MEM for Load

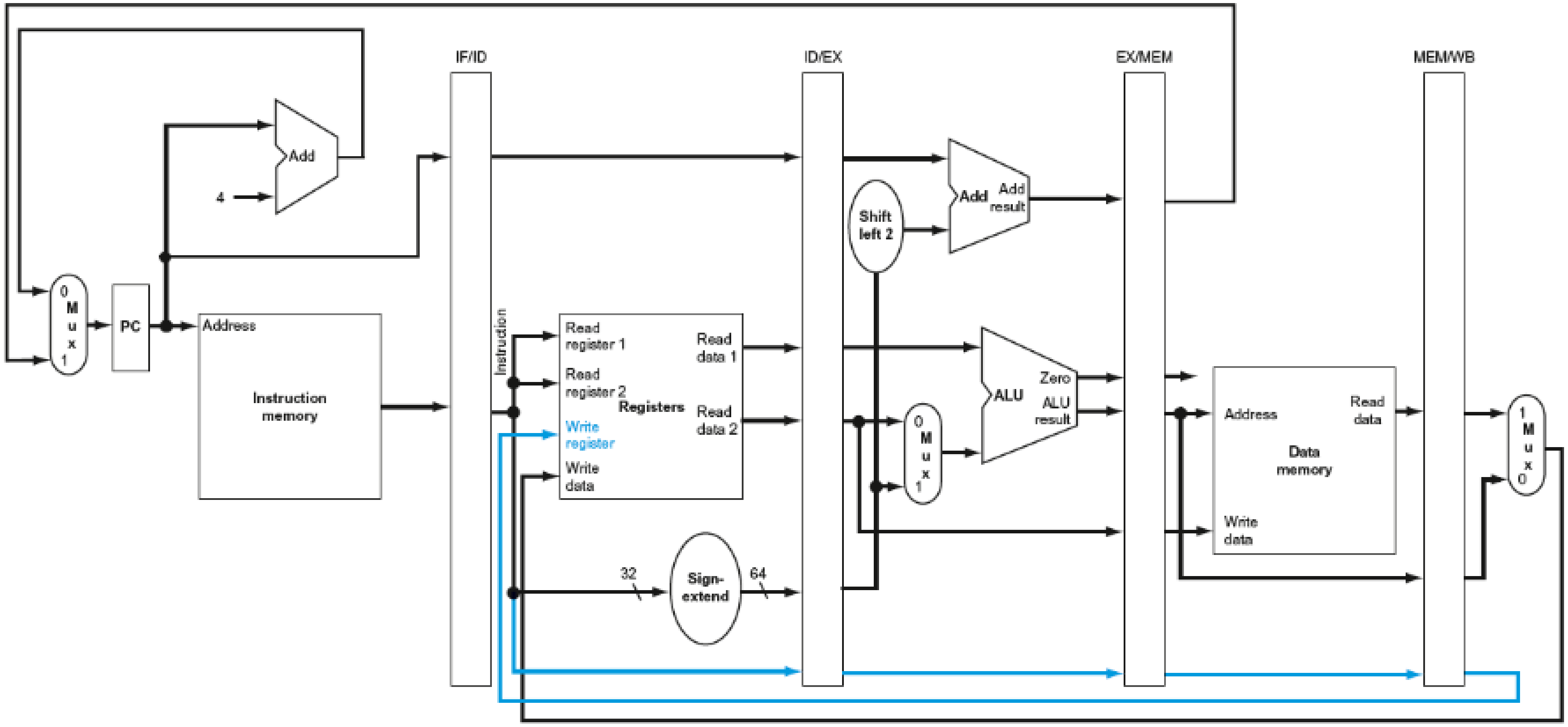

LDUR
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
32
Sign-extend
64
Address
Read data
Data memory
Write data

WB for Load



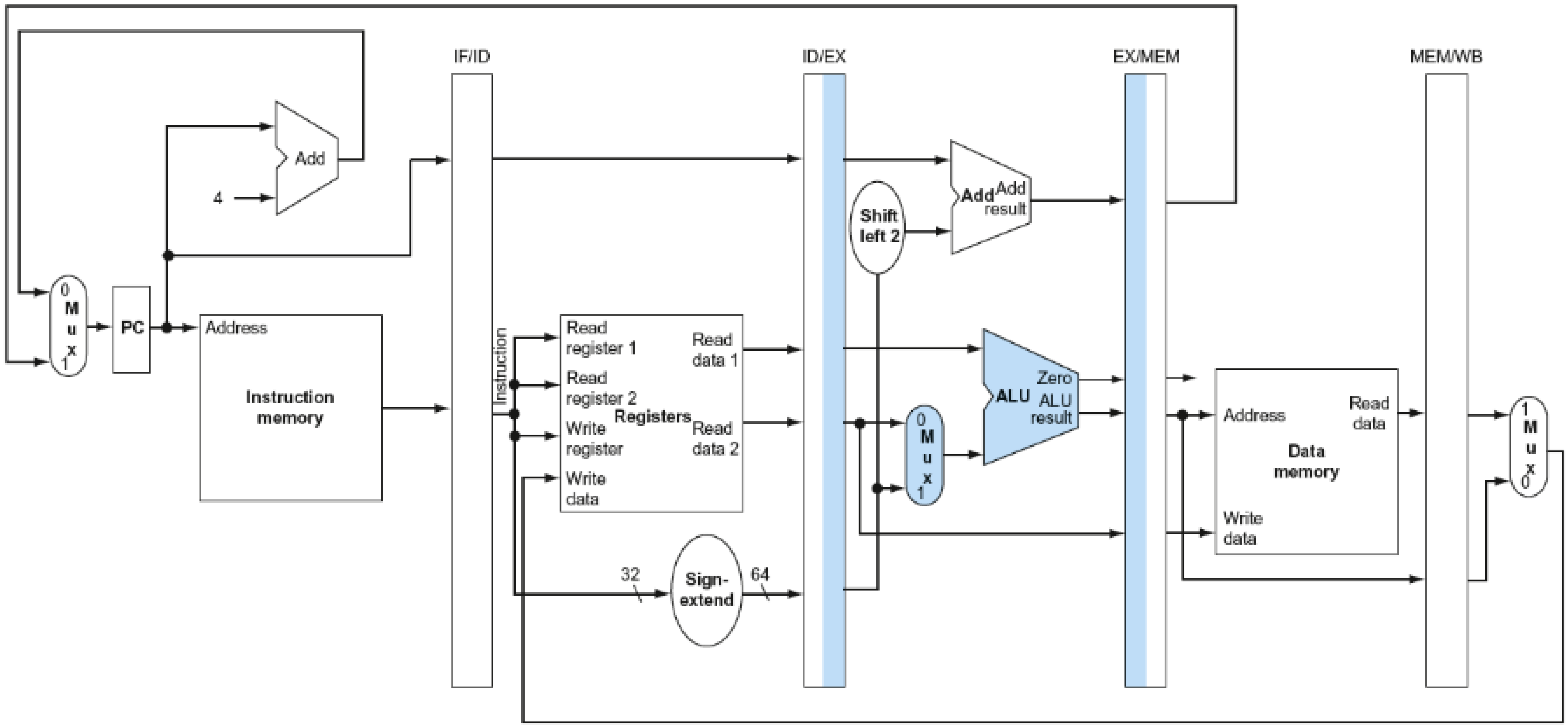
Wrong register number

Corrected Datapath for Load



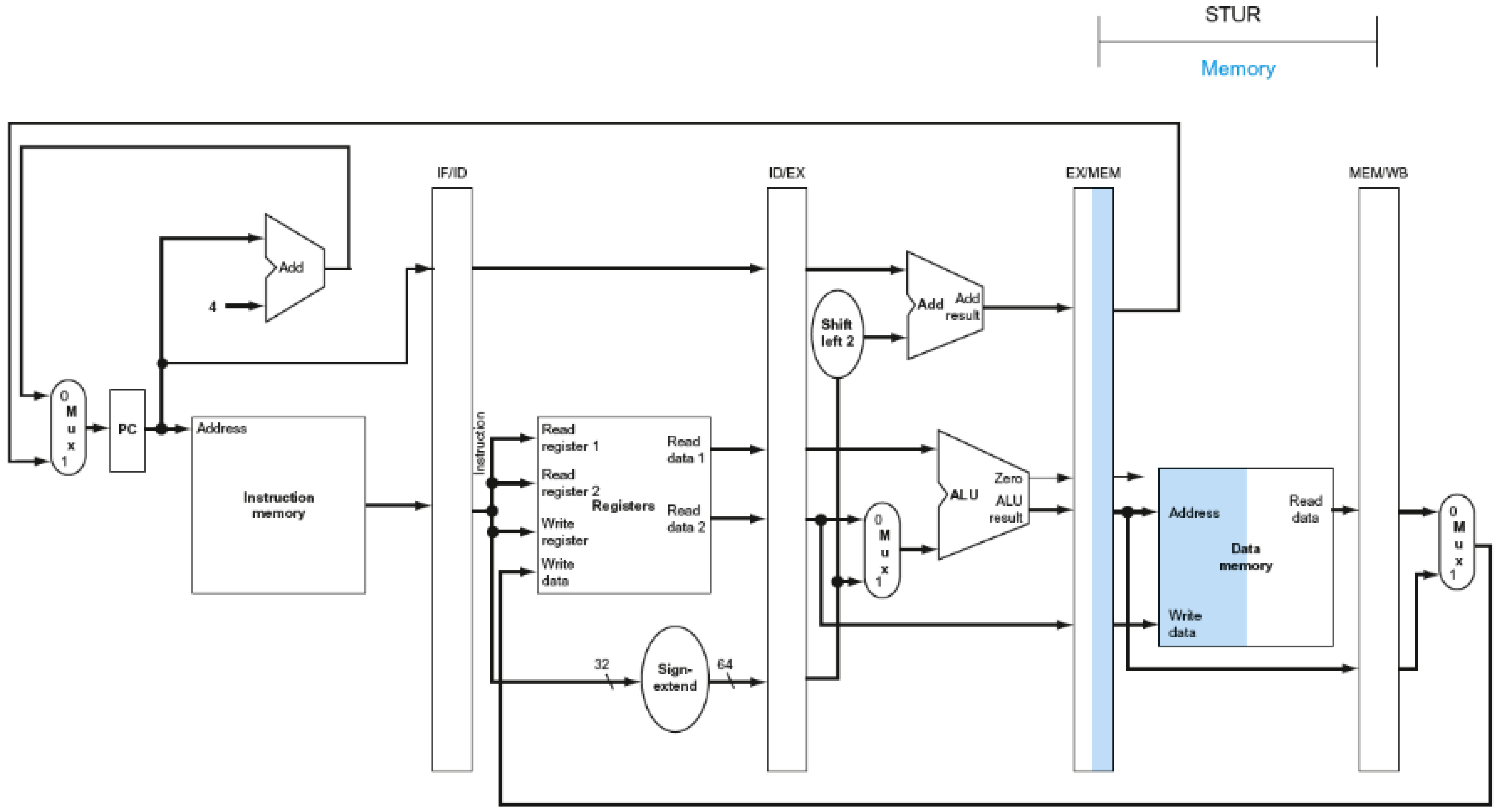
EX for Store

STUR
Execution

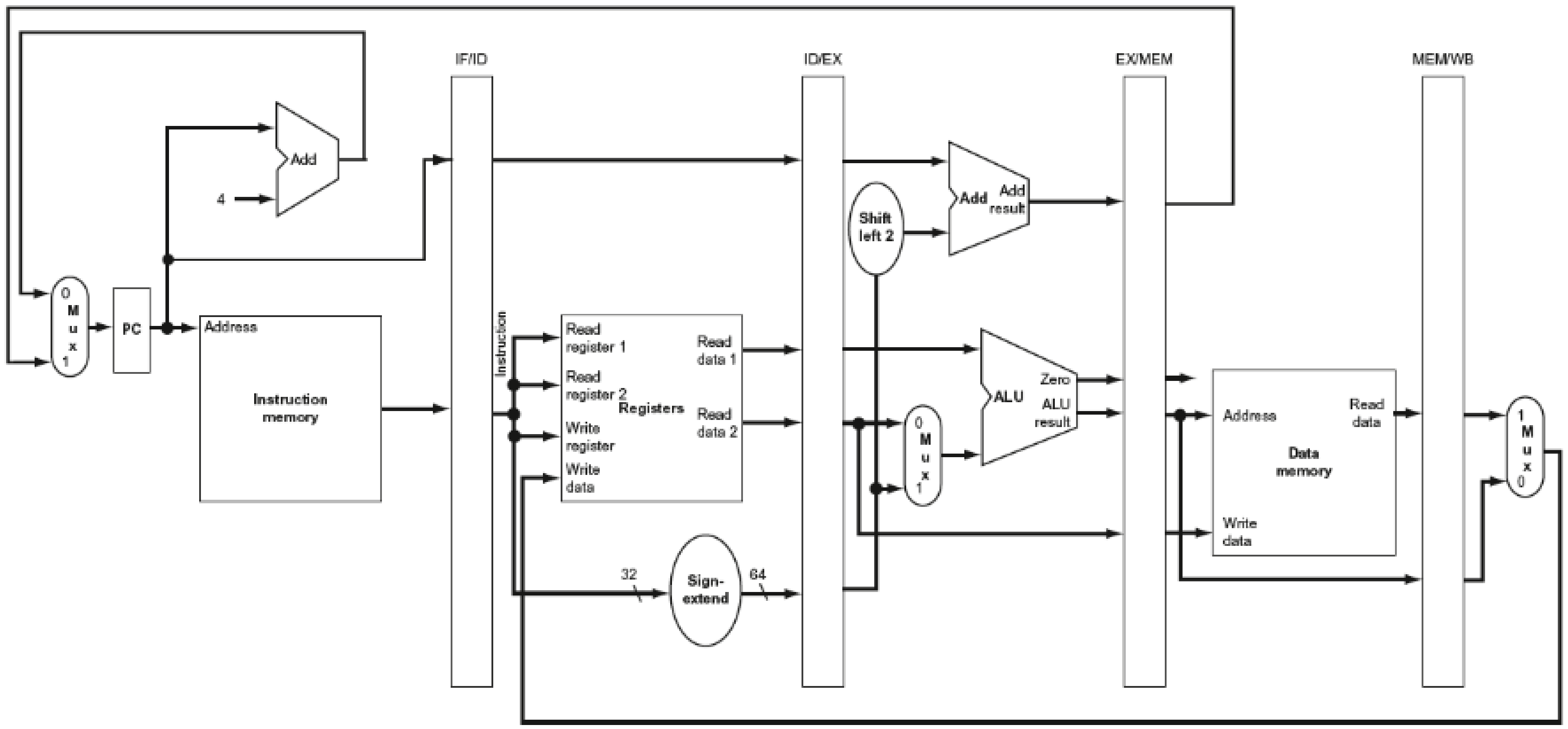


ENGINEERING@SYRACUSE

MEM for Store



WB for Store



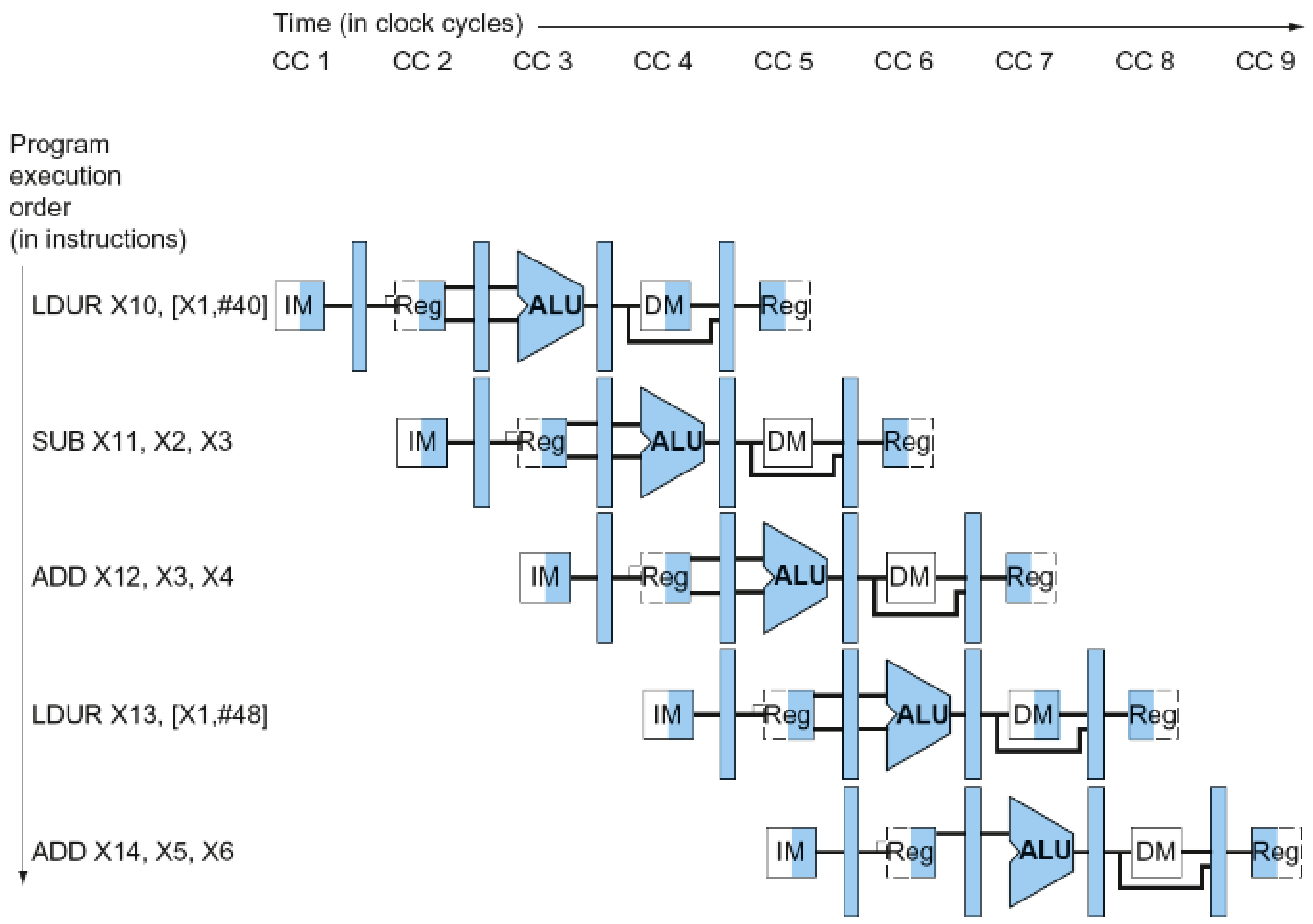
ENGINEERING@SYRACUSE

ENGINEERING@SYRACUSE

Pipelined Control

Multicycle Pipeline Diagram

- Form showing resource usage



Multicycle Pipeline Diagram (cont.)

- Traditional form

Time (in clock cycles) →

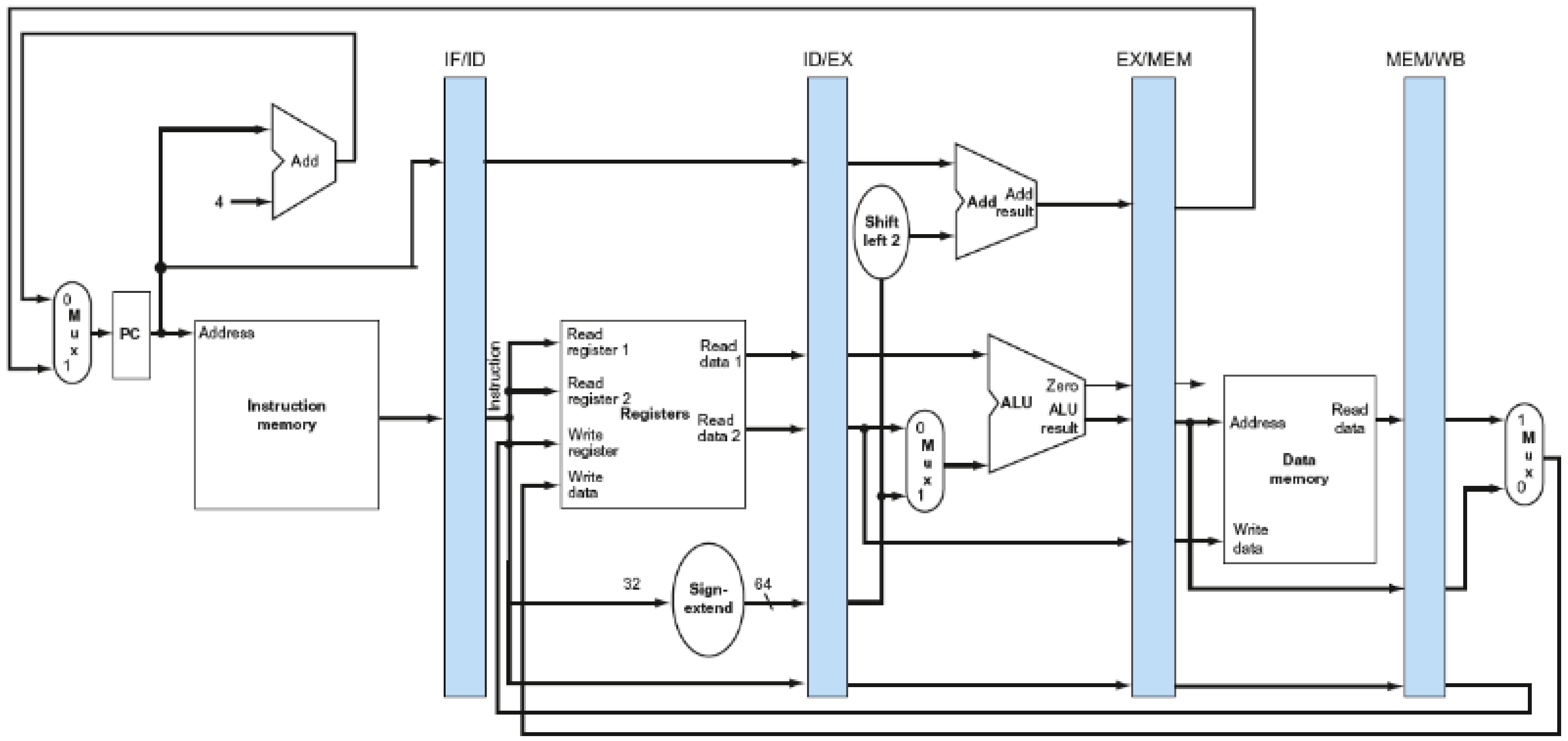
Program execution order (in instructions) ↓

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
LDUR X10, [X1,#40]	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
SUB X11, X2, X3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
ADD X12, X3, X4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
LDUR X13, [X1,#48]				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
ADD X14, X5, X6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

Single-Cycle Pipeline Diagram

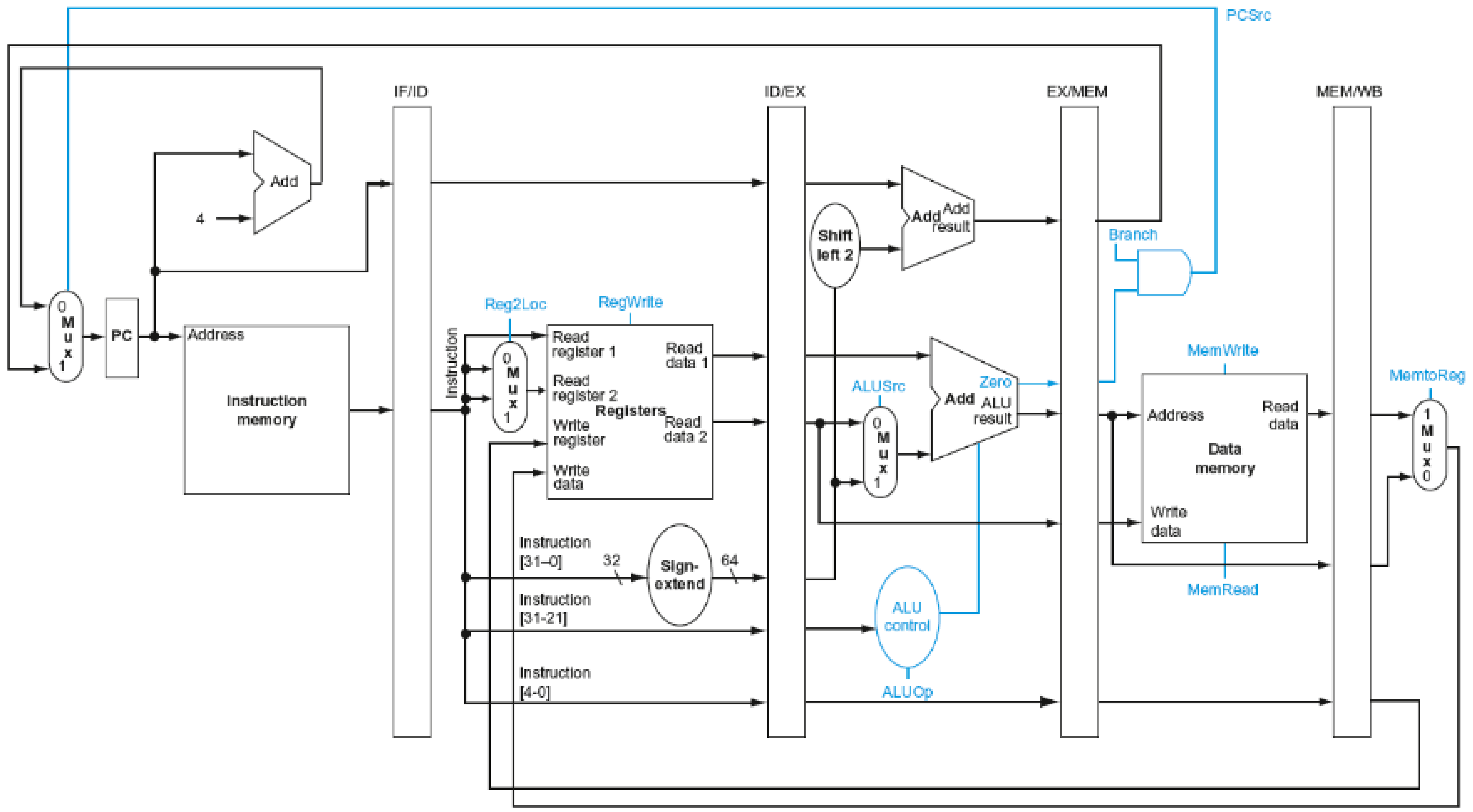
- State of pipeline in a given cycle

ADD X14, X5, X6	LDUR X13, [X1,48]	ADD X12, X3, X4	SUB X11, X2, X3	LDUR X10, [X1,40]
Instruction fetch	Instruction decode	Execution	Memory	Write-back



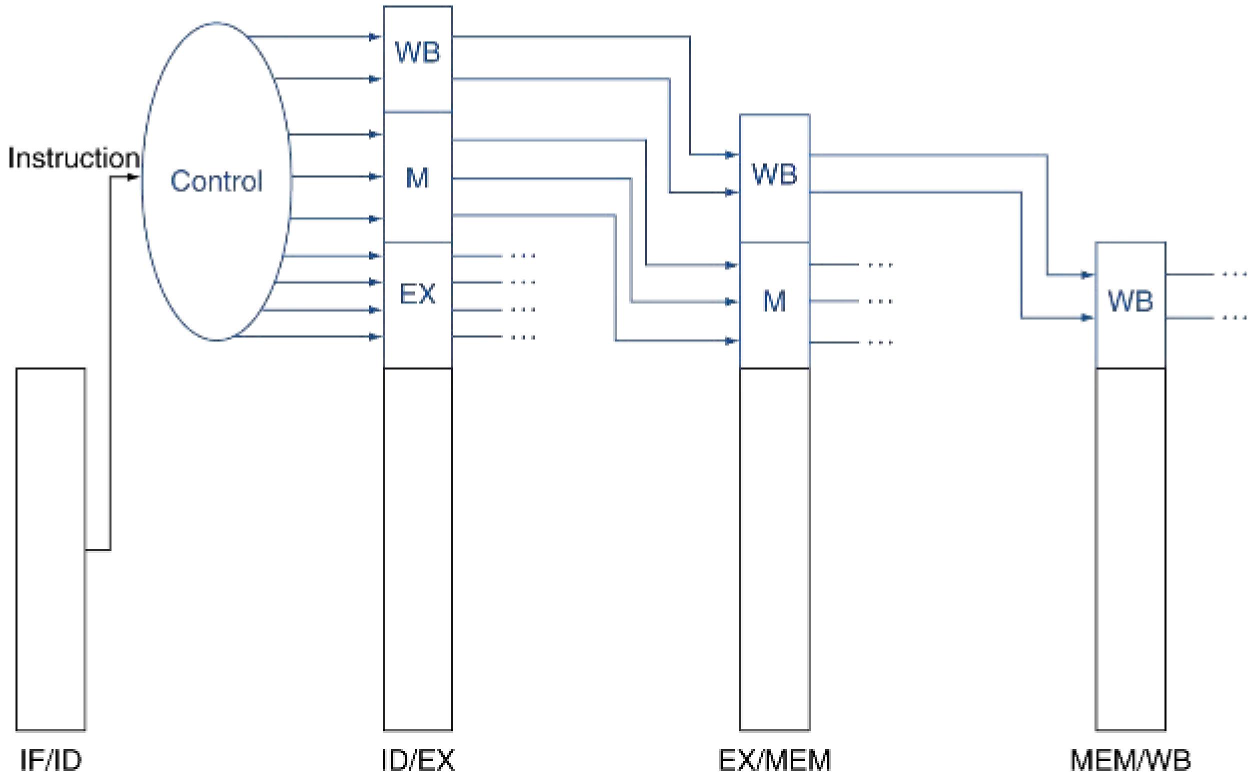
ENGINEERING@SYRACUSE

Pipelined Control (Simplified)



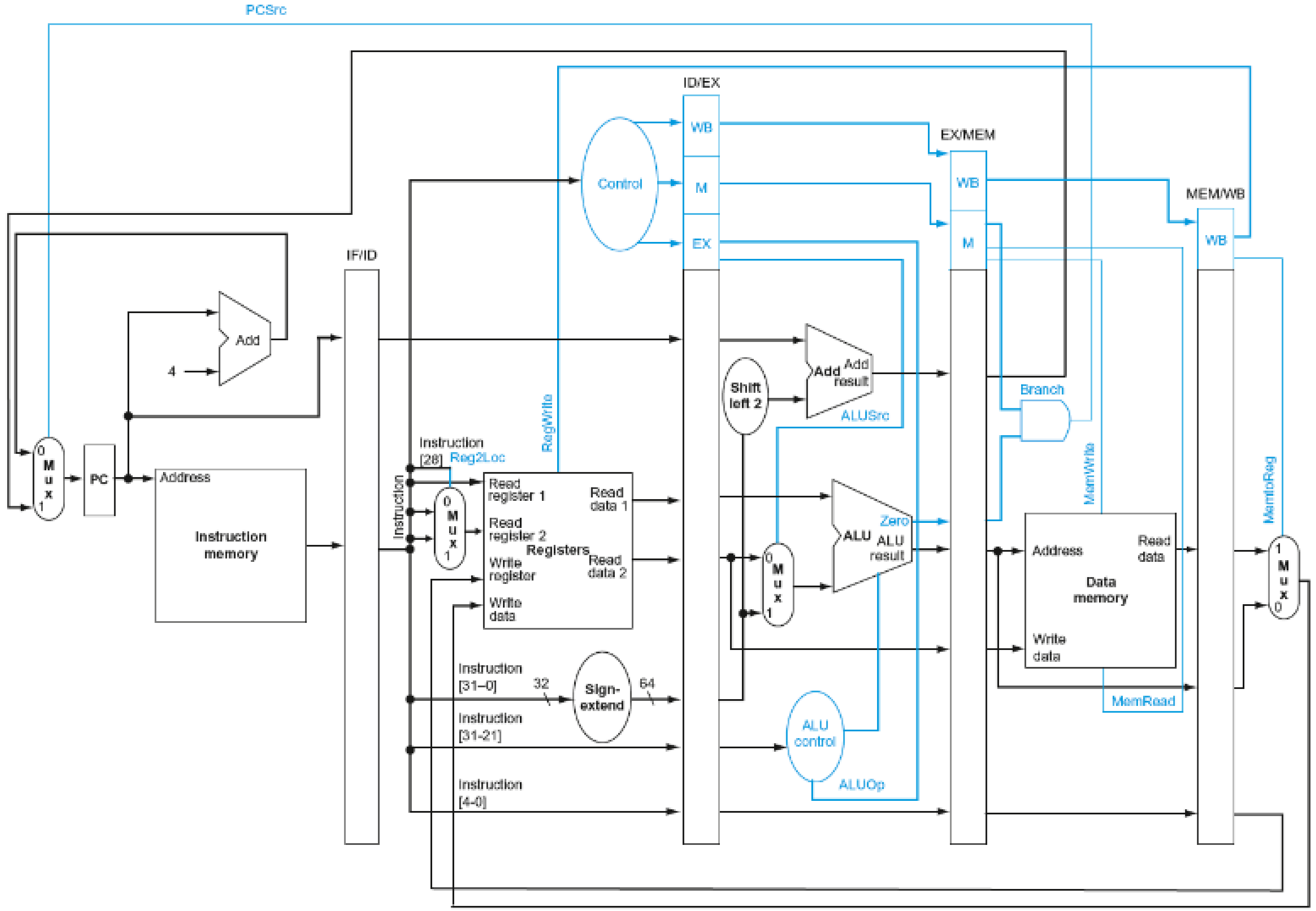
Pipelined Control

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



ENGINEERING@SYRACUSE

Pipelined Control (cont.)



ENGINEERING@SYRACUSE

Conclusions

In Conclusion

- Building blocks
  - Computation, communication, storage
- Logic design
  - Gates, flip-flops, latches, clock
- Datapath
  - Register, memory, branch operations
- Implementation
  - Execution cycle: Fetch, Decode, Operands, Execute, Store, Next
- An overview of pipelining
  - Overlapping instructions to improve throughput
  - Structural, data, control hazards

ENGINEERING@SYRACUSE