NTNU

Det skapende universitet

TDT4255 Computer Design

Lecture 3: Pipelining

Magnus Jahre

Outline

- Appendix C.1 to C.3

Appendix C

Pipelining: Basic and Intermediate Concepts

Acknowledgement: Slides are adapted from Morgan Kaufmann companion material

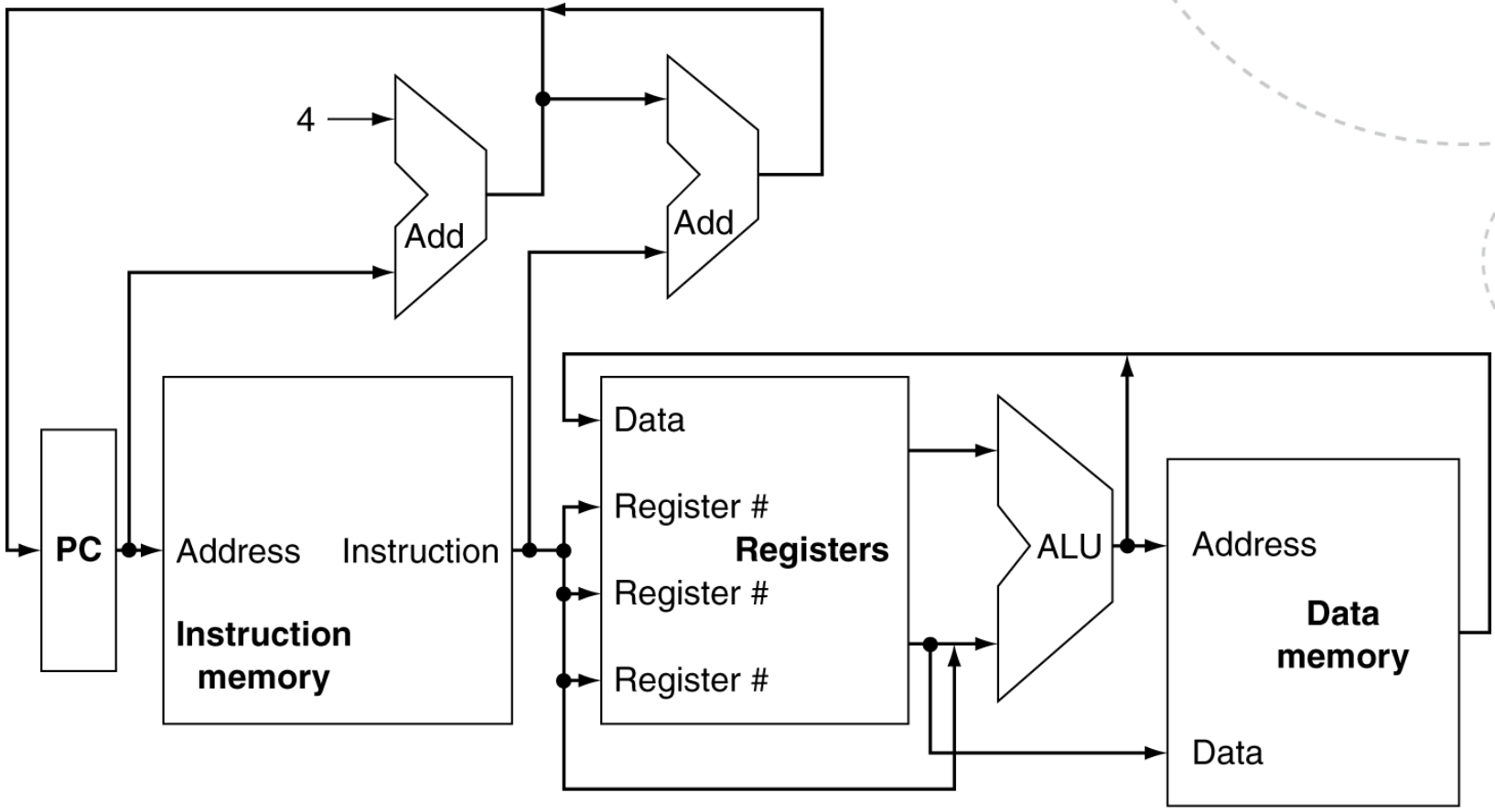
Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine three implementations
  - Single cycle
  - In-order pipelined
  - Out-of-order superscalar (and pipelined)

Instruction Execution

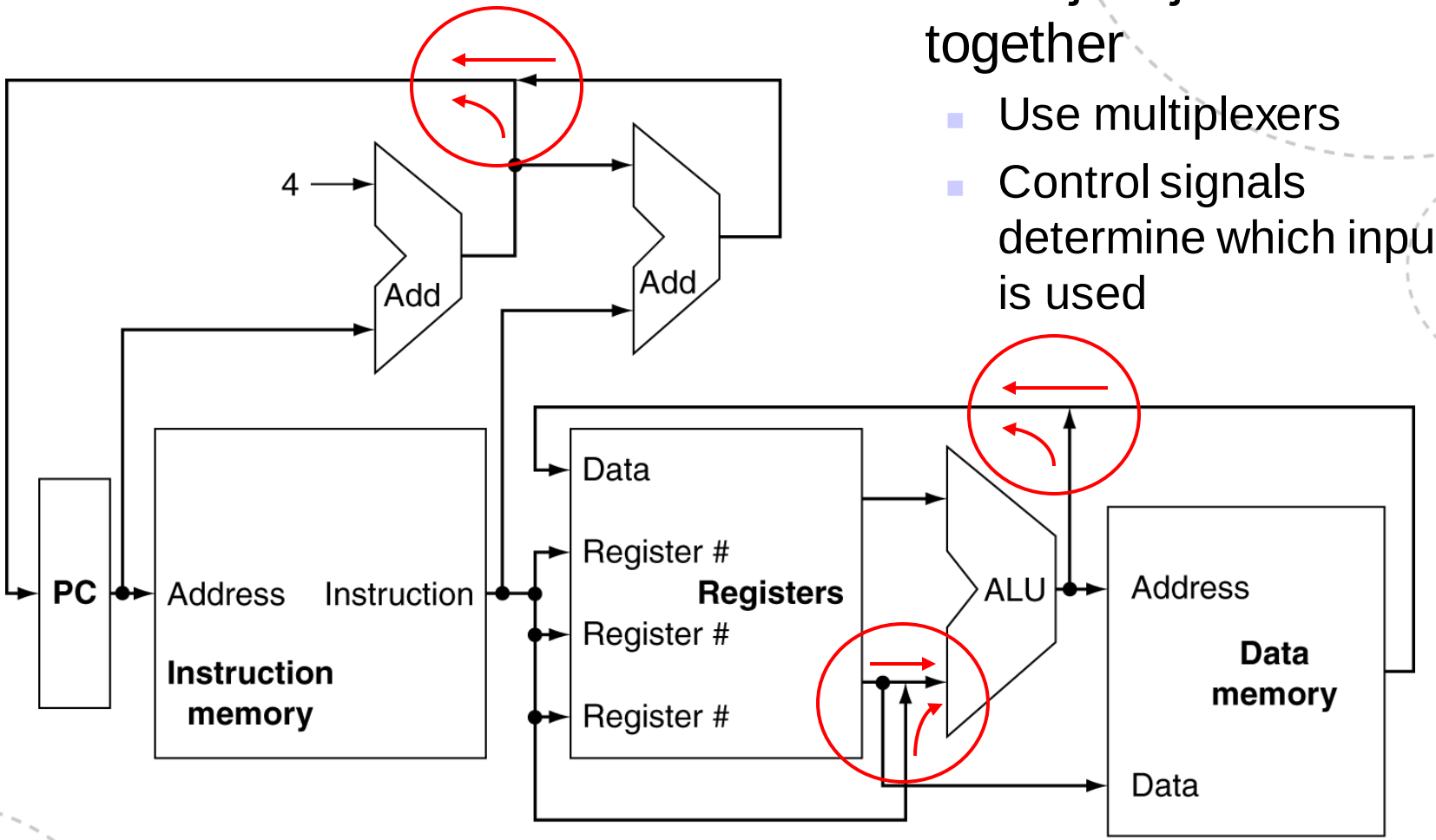
1. PC → instruction memory, fetch instruction
   - Compute the next PC ← target address or PC + 4
2. Register numbers → register file, read registers
3. Depending on instruction class
   - Use ALU to calculate
     - Arithmetic result
     - Memory address for load/store
     - Branch target address
4. Access memory (if necessary)
5. Write result to register file (if necessary)

CPU Overview



Multiplexers

- Can't just join wires together
  - Use multiplexers
  - Control signals determine which input is used



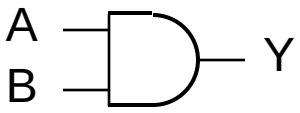
LOGIC DESIGN CONVENTIONS

Logic Design Repetition

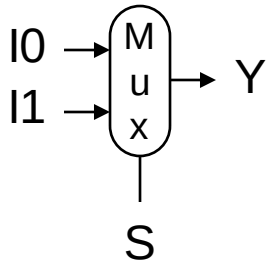
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

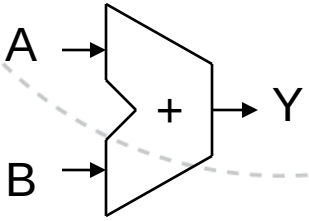
- AND-gate
  - Y = A & B



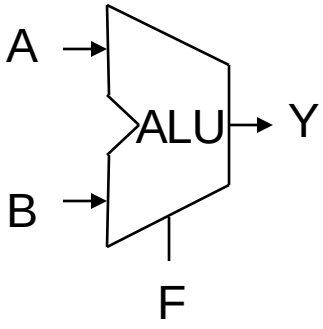
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

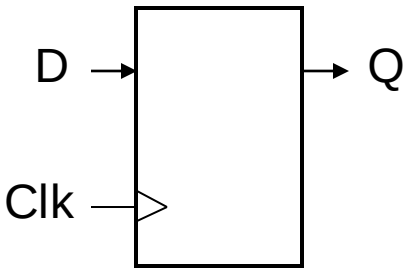


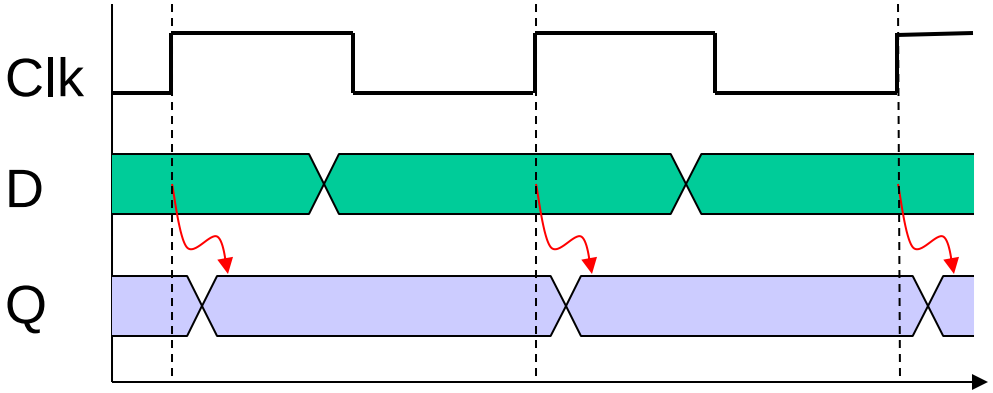
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

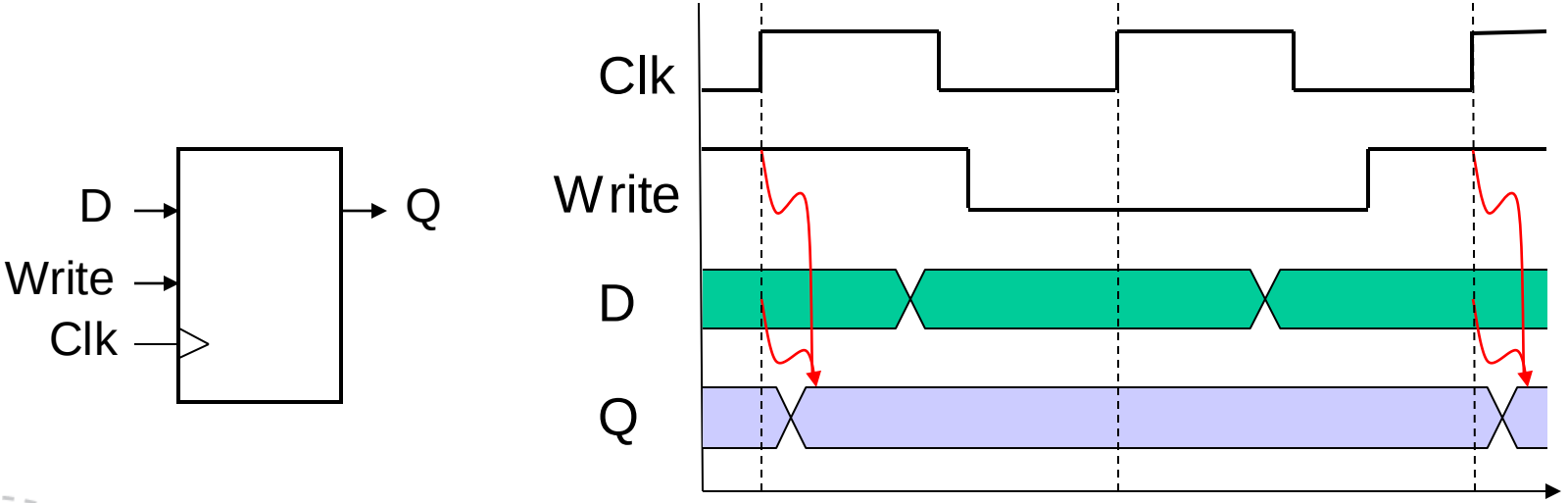
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1





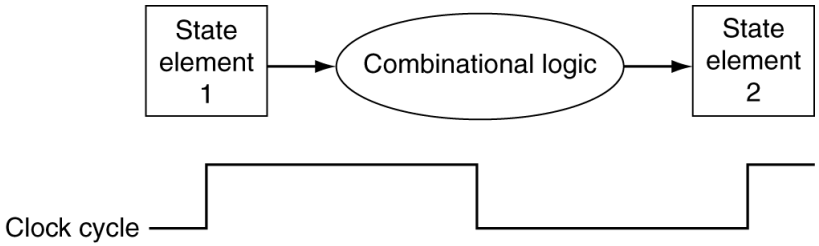
Sequential Elements

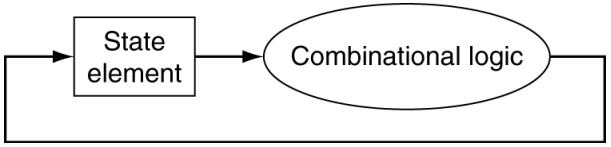
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



Clocking Methodology

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





BUILDING A DATAPATH

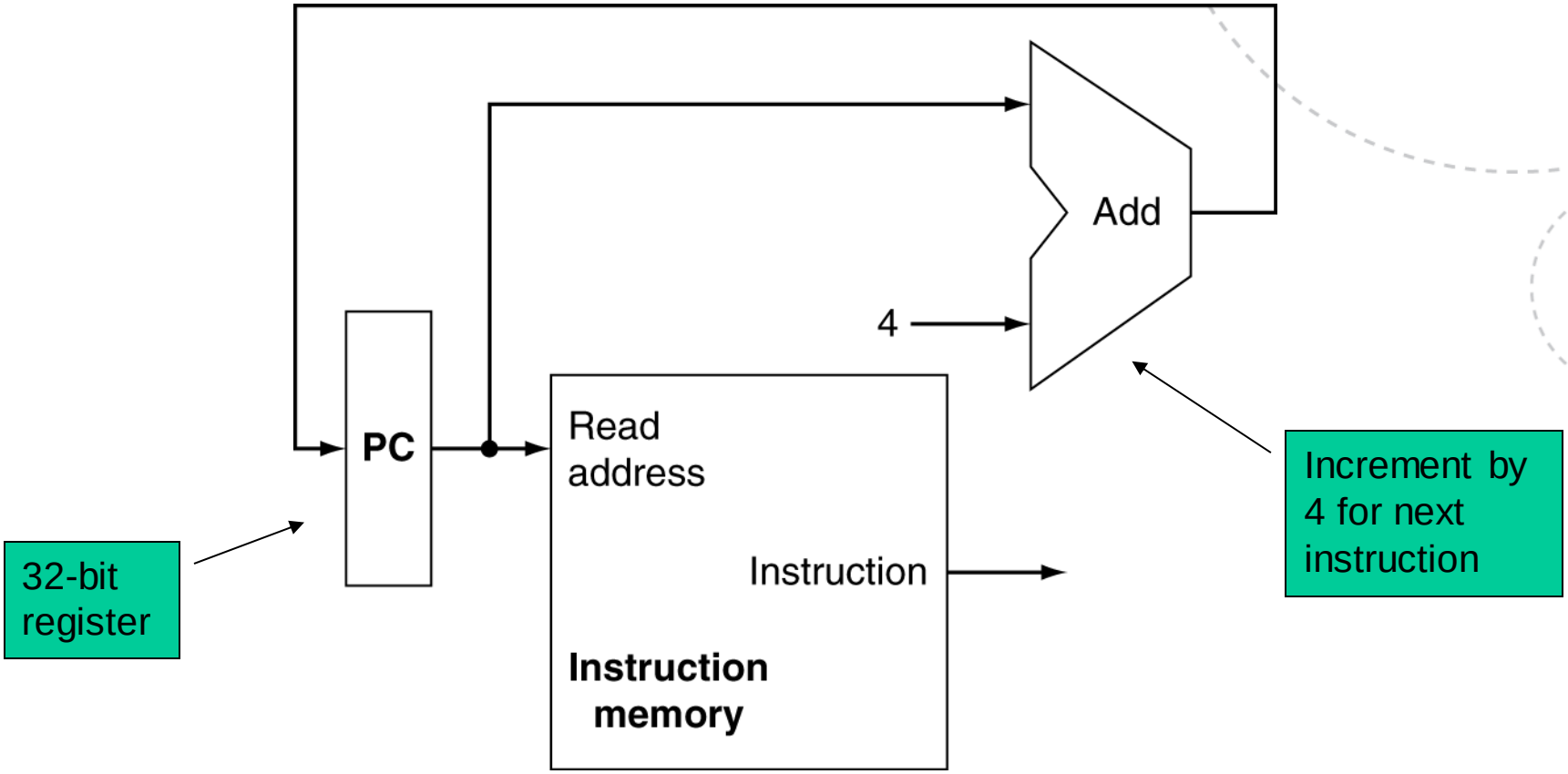
Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - You will implement a RISC-V datapath in the exercises
- Design goal: Efficiently implement the three instruction formats:

R	Opcode (6)	Rs (5)	Rt (5)	Rd (5)	Shamt (5)	Func (6)
I	Opcode (6)	Rs (5)	Rt (5)	Constant or address (16)		
J	Opcode (6)	Address (26)				

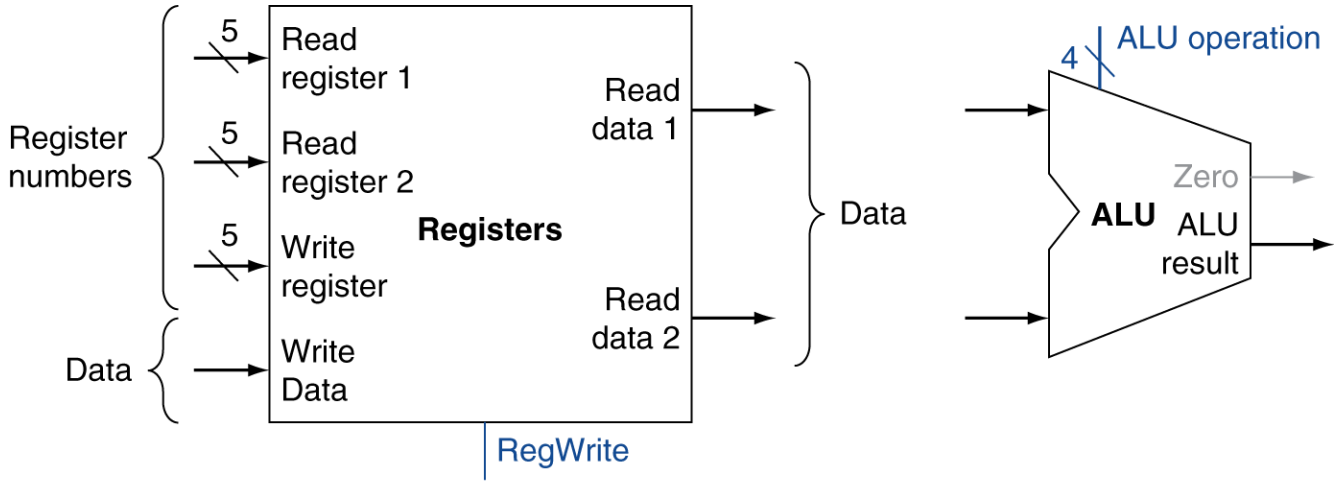
We'll start with these

Instruction Fetch



ALU Instructions

1. Read two register operands
2. Perform arithmetic/logical operation
3. Write register result

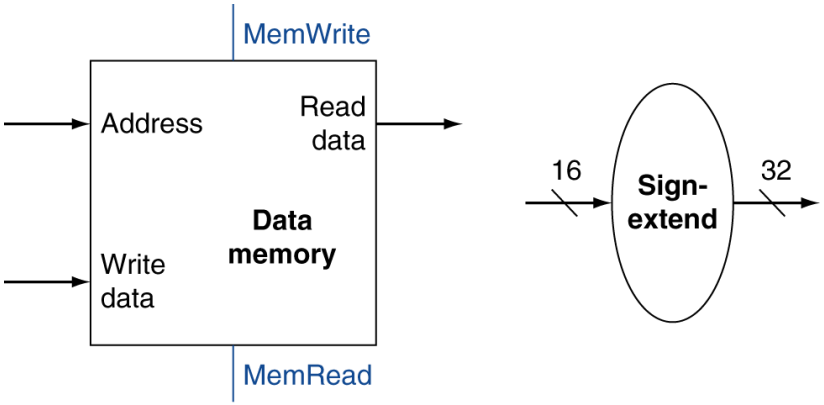


a. Registers

b. ALU

Load/Store Instructions

1. Read register operands
2. Calculate address
   - Use ALU, but sign-extend offset
3. Finish instruction:
   - Load: Read memory and update register
   - Store: Write register value to memory



a. Data memory unit

b. Sign extension unit

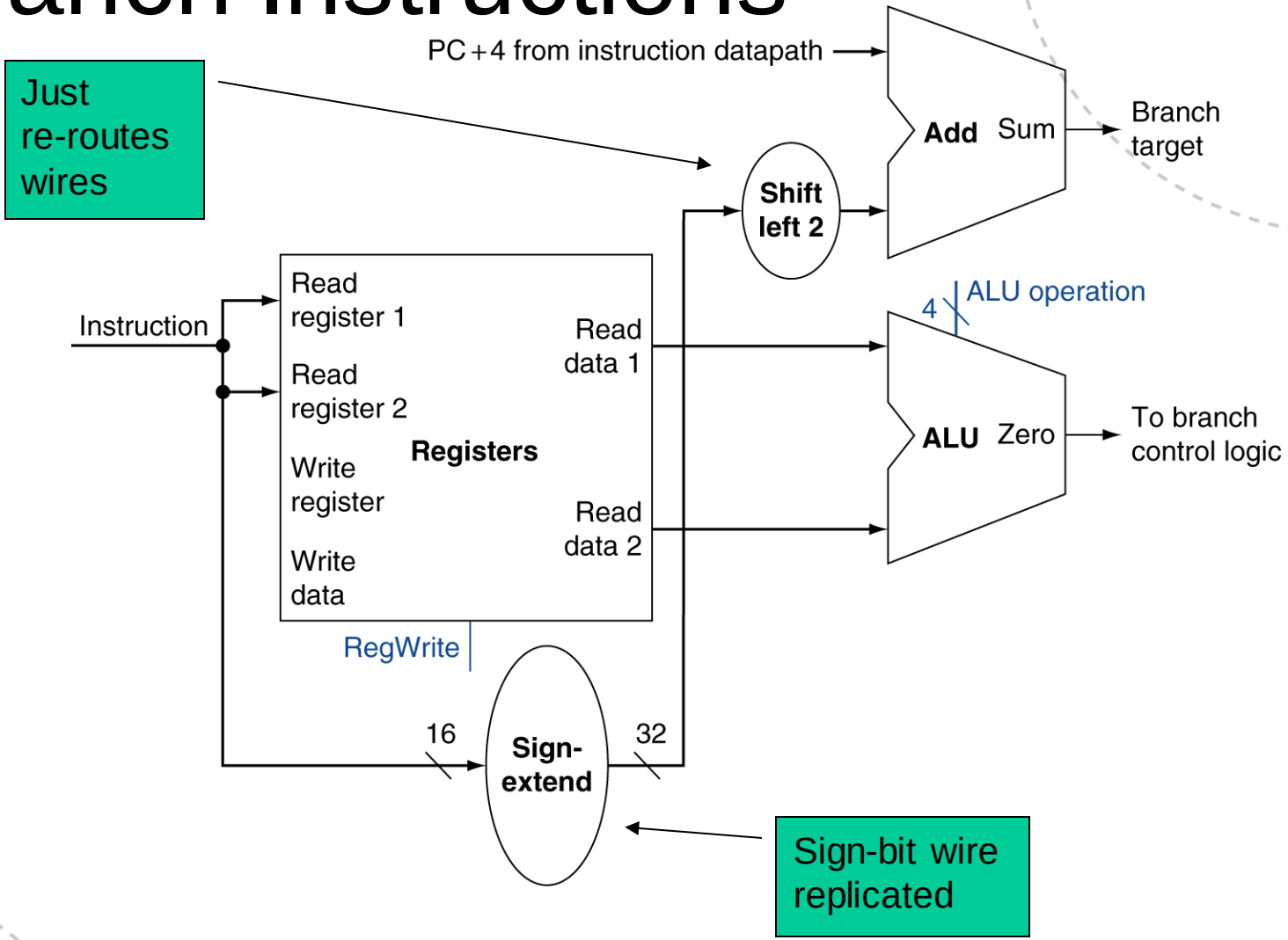
Branch Instructions

1. Read register operands

2. Compare operands
   - Use ALU, subtract and check Zero output

3. Calculate target address
   - Sign-extend displacement
   - Shift left 2 places (word displacement)
   - Add to PC + 4
     - Already calculated by instruction fetch

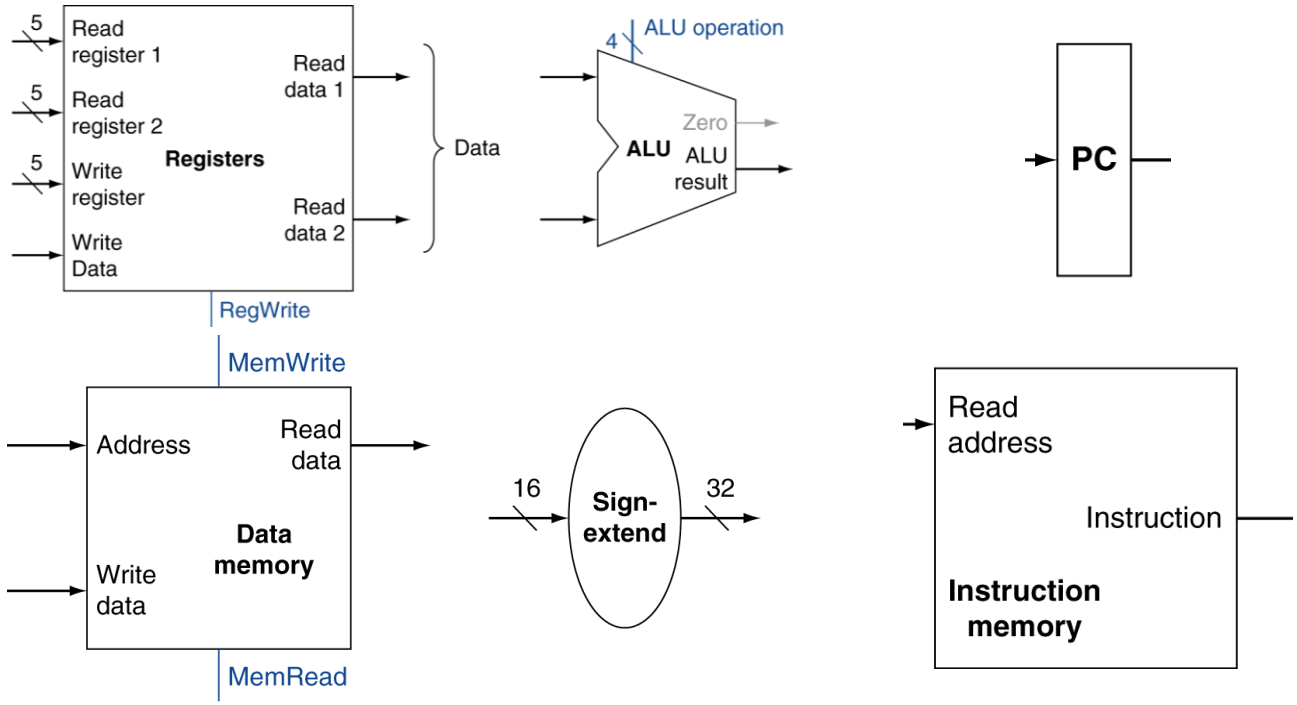
Branch Instructions



Composing the Elements

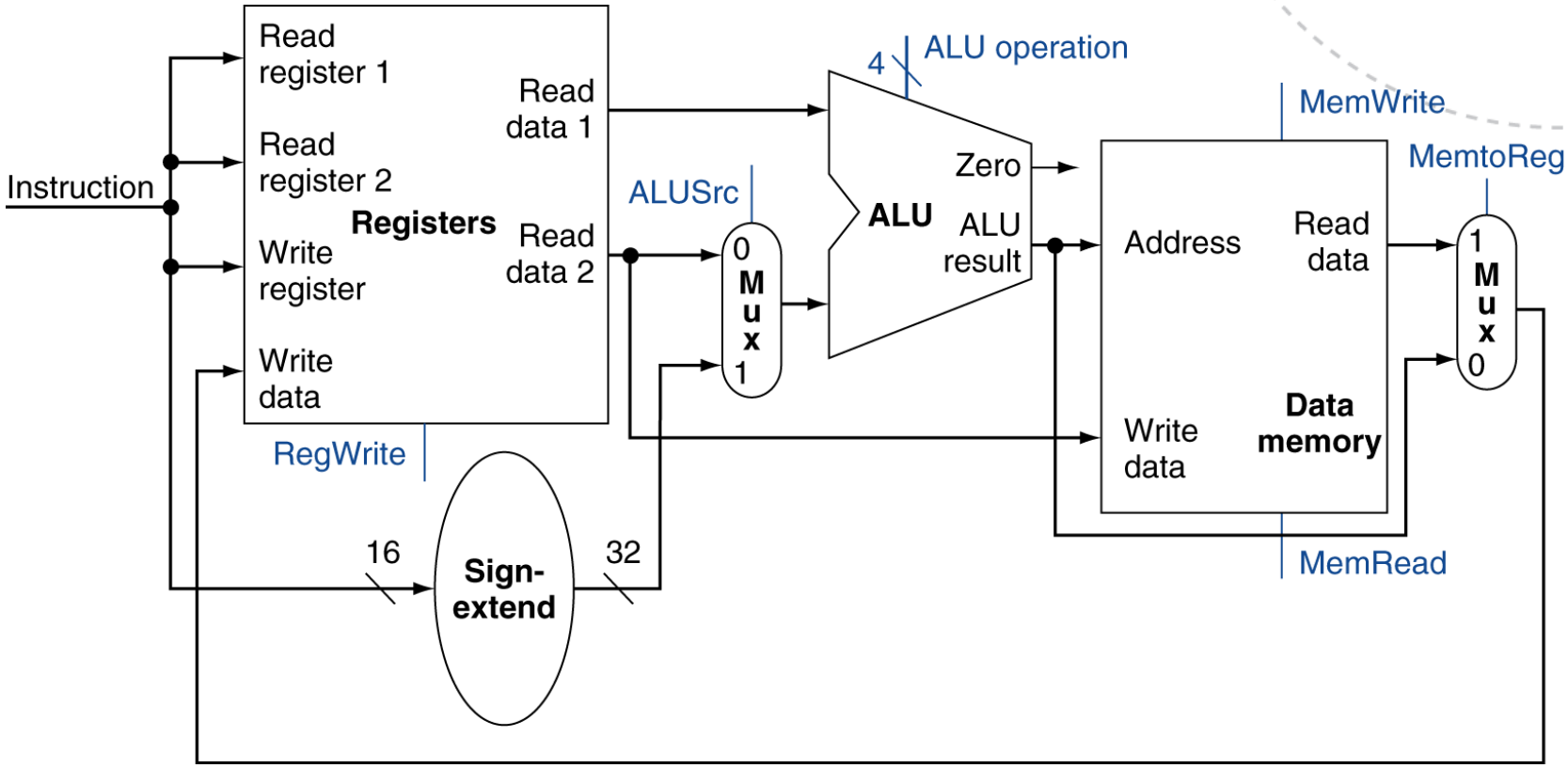
- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

In-class assignment: Draw your own datapath



R	Opcode (6)	Rs (5)	Rt (5)	Rd (5)	Shamt (5)	Func (6)
I	Opcode (6)	Rs (5)	Rt (5)	Constant or address (16)		
J	Opcode (6)	Address (26)				

A Possible Datapath



A SINGLE CYCLE MIPS PROCESSOR

ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

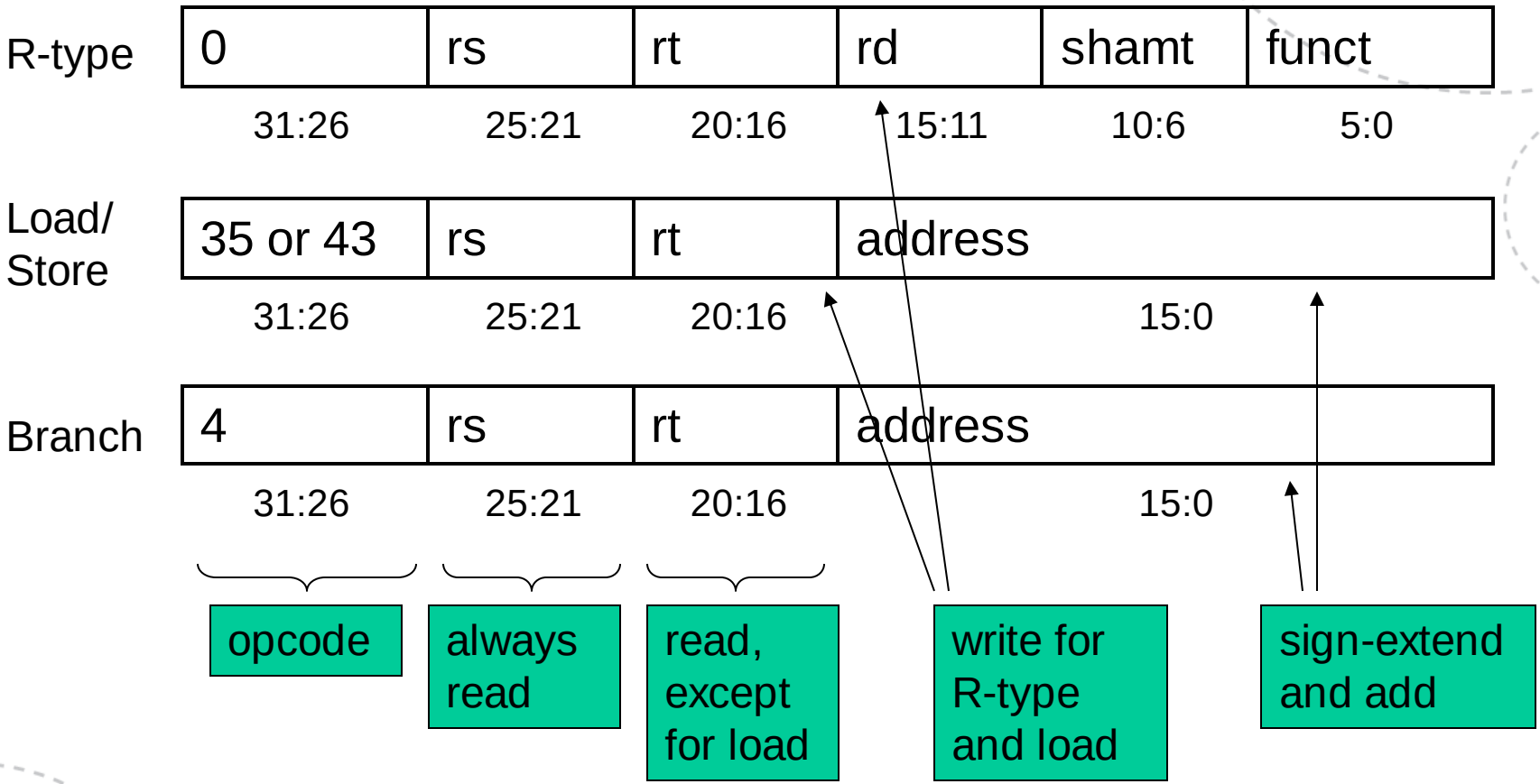
ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

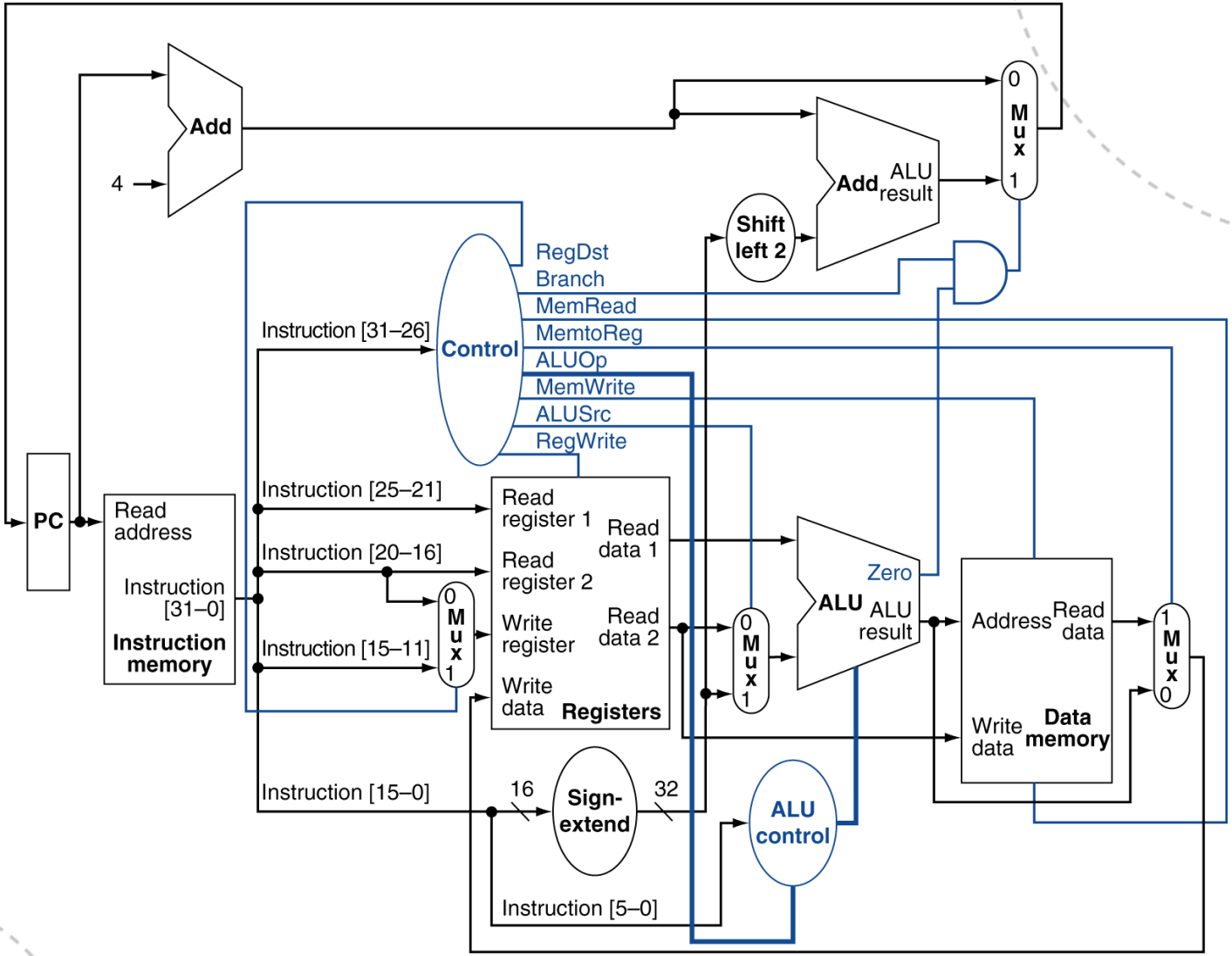
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The Main Control Unit

- Control signals derived from instruction



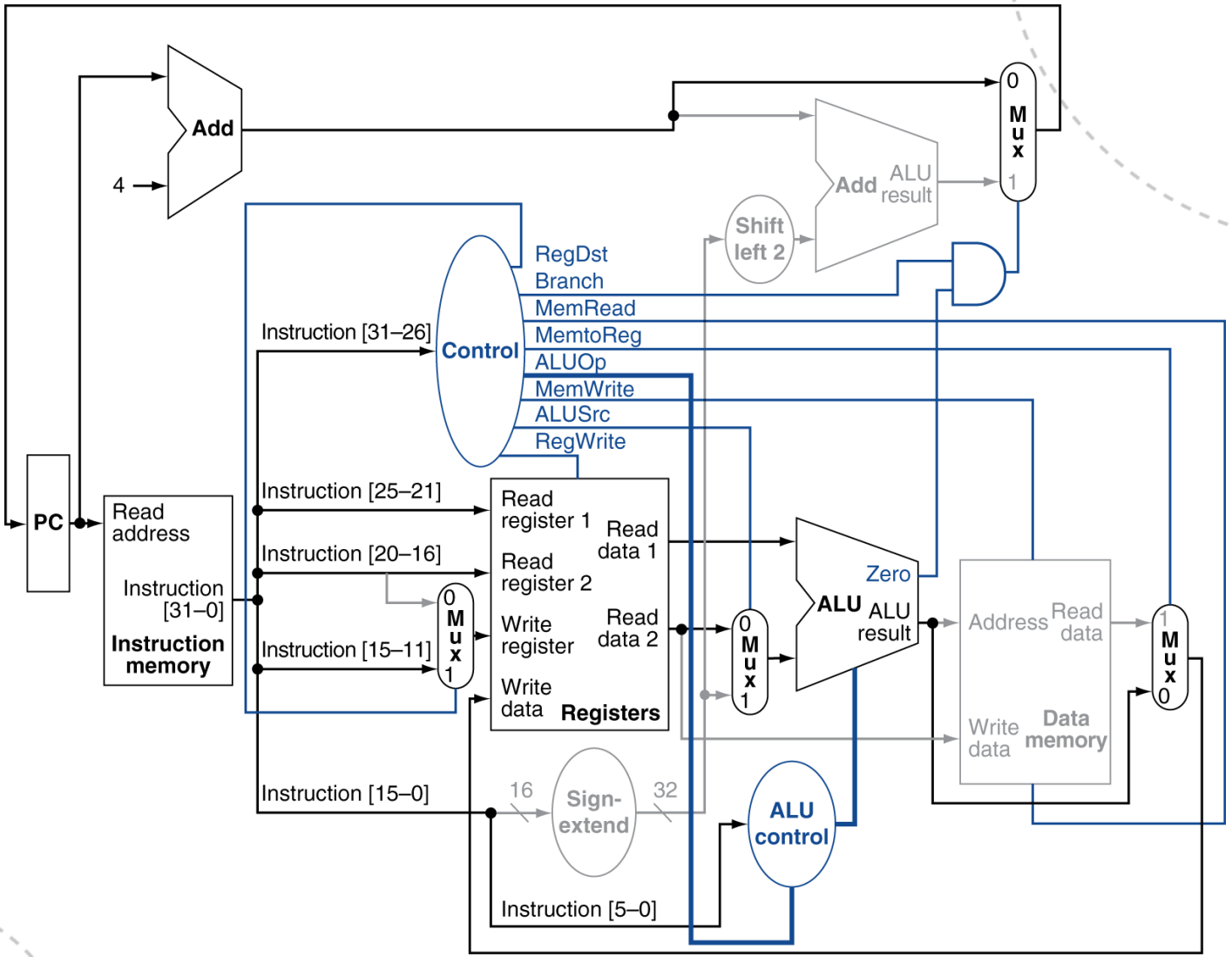
Datapath With Control



R-Type Instruction

Example

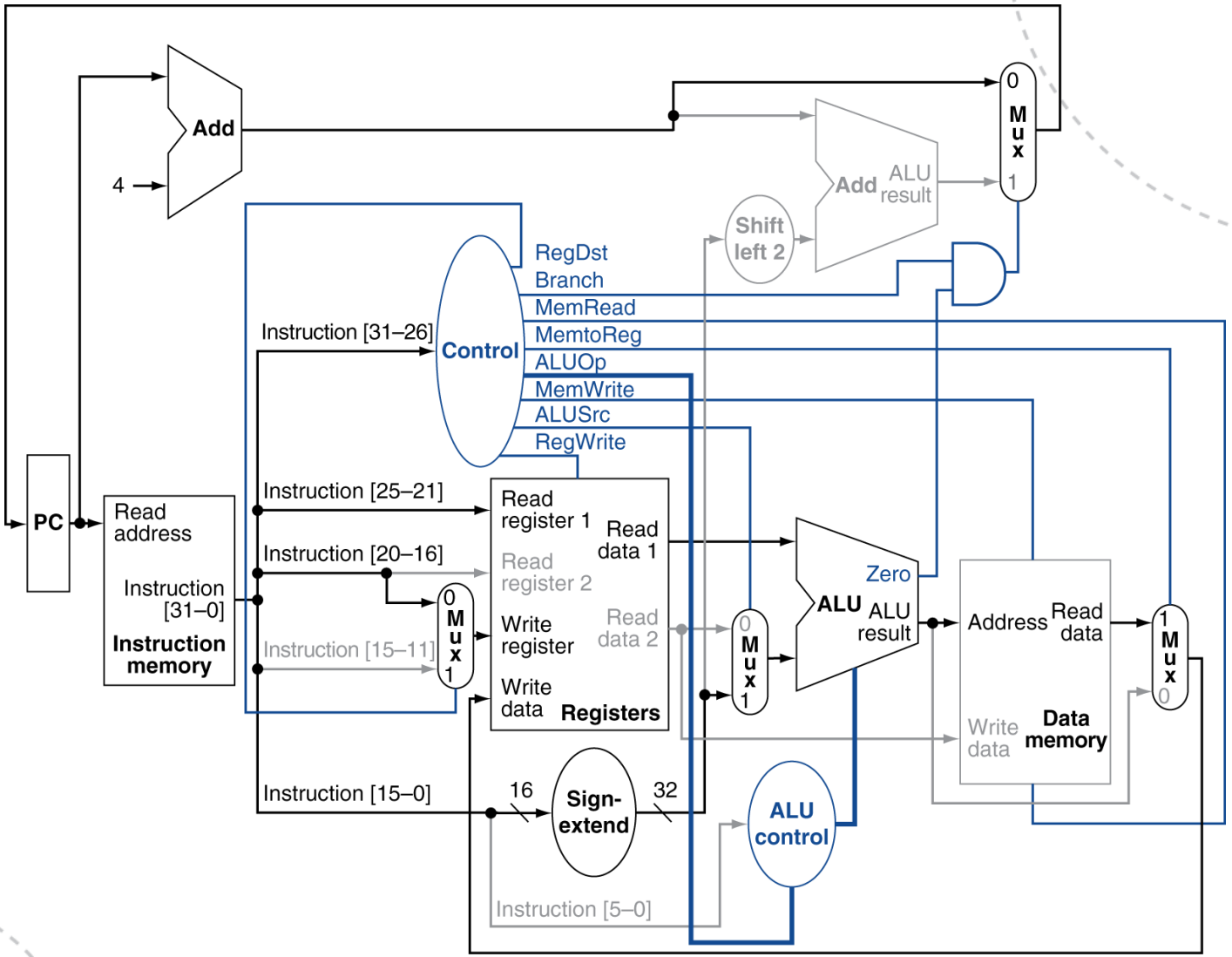
```
add $t0, $s1, $s2
```



Load Instruction

Example

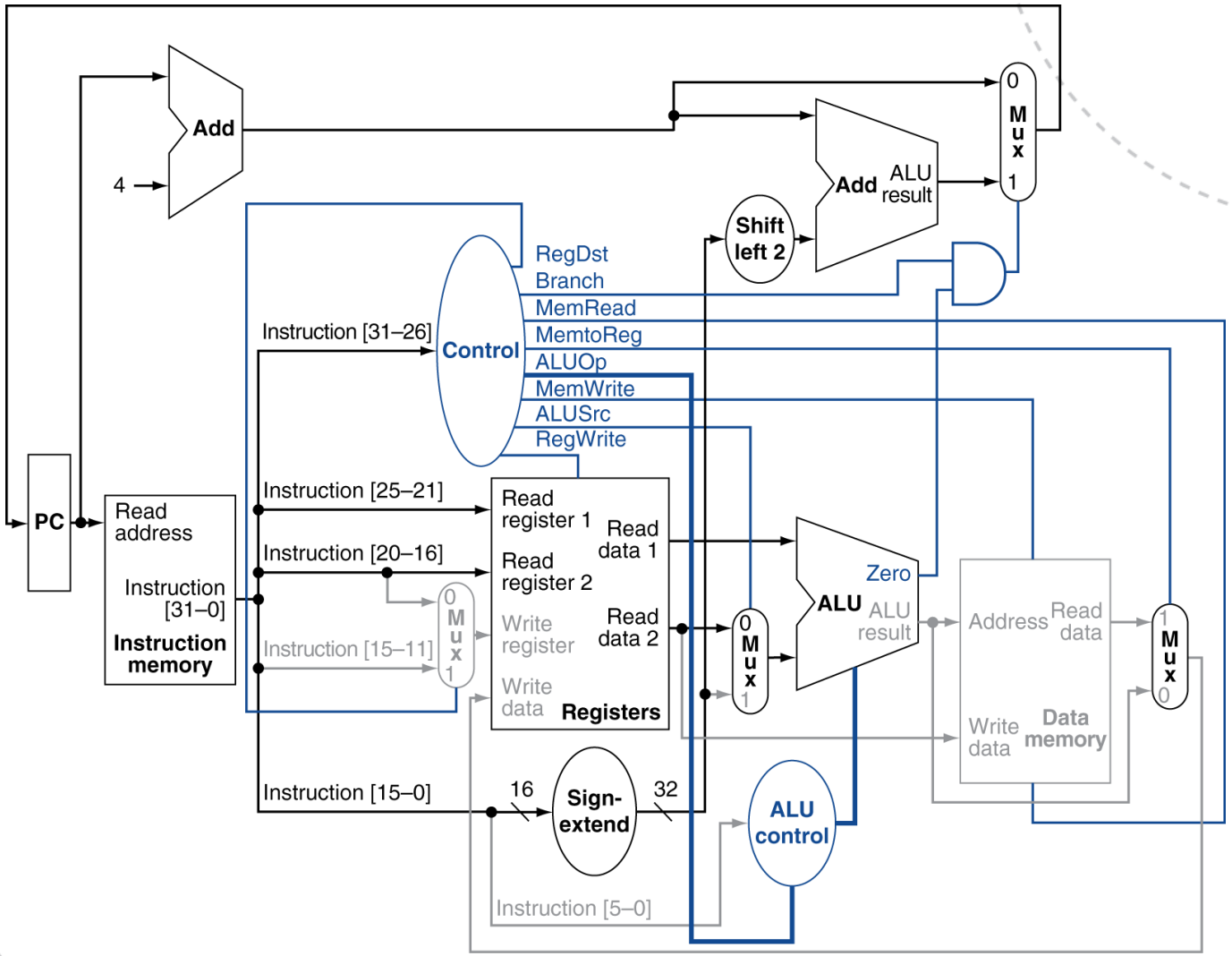
```
lw  $t0, 32($s3)
```



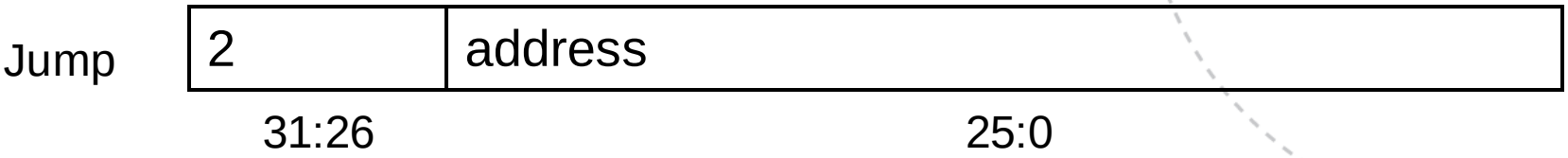
Branch-on-Equal Instruction

Example

```
beq $t0, $t1, label
```

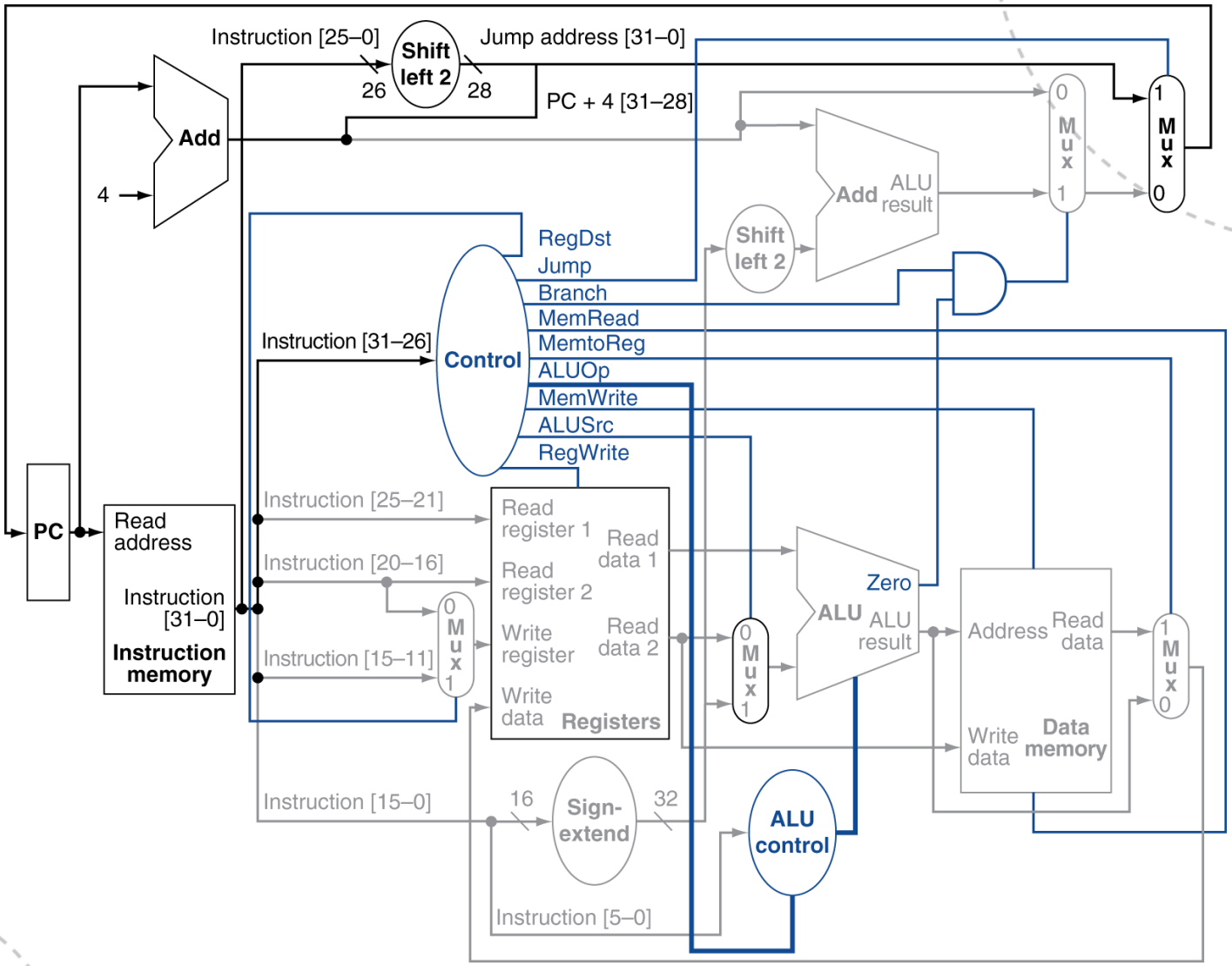


Implementing Jumps



- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

Datapath With Jumps Added



Single Cycle Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
  - Violates the design principle “Making the common case fast”
- Possible improvements:
  - A multi-cycle implementation (not so common)
  - Pipelining (very common)

Control Units

- Combinatorial control units
  - Lacks state
  - The single cycle processor has a combinatorial control unit: All state is determined by the current instruction
- Sequential control units
  - Larger and more complex than combinatorial control units
  - Can use smaller combinatorial units internally
- Finite state machines vs. Microprograms

PIPELINING BASICS

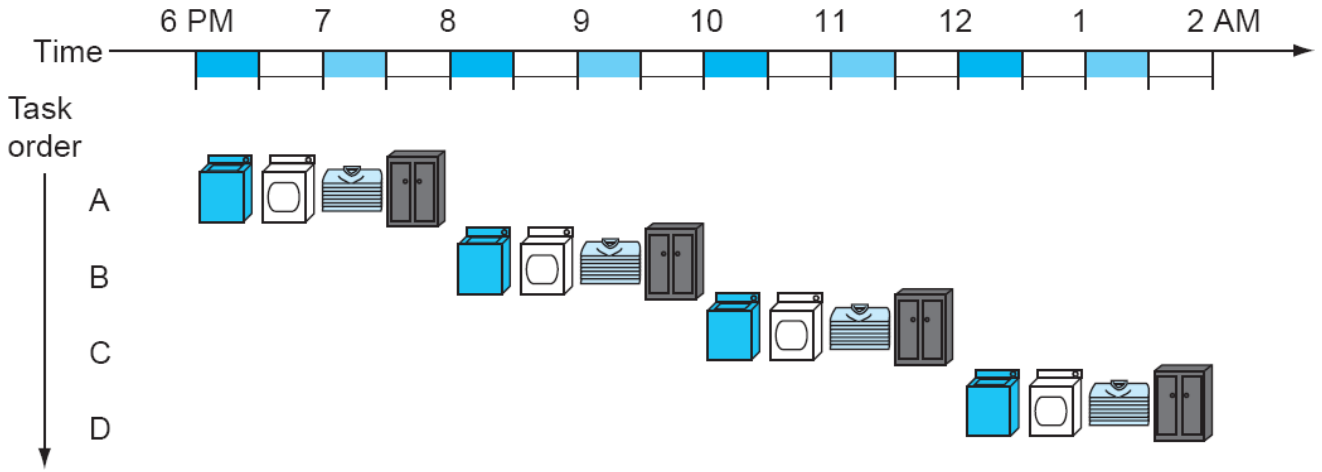
Pipelining

- Pipelining: An implementation technique in which multiple instructions are overlapped in execution.

- Lets illustrate it with something we all can relate to
  - The laundry analogy from the book
    - Wash dirty clothes in washer
    - Dry the clean wet load in the dryer
    - Fold the dry clothes
    - Ask your roommate to put the clothes away

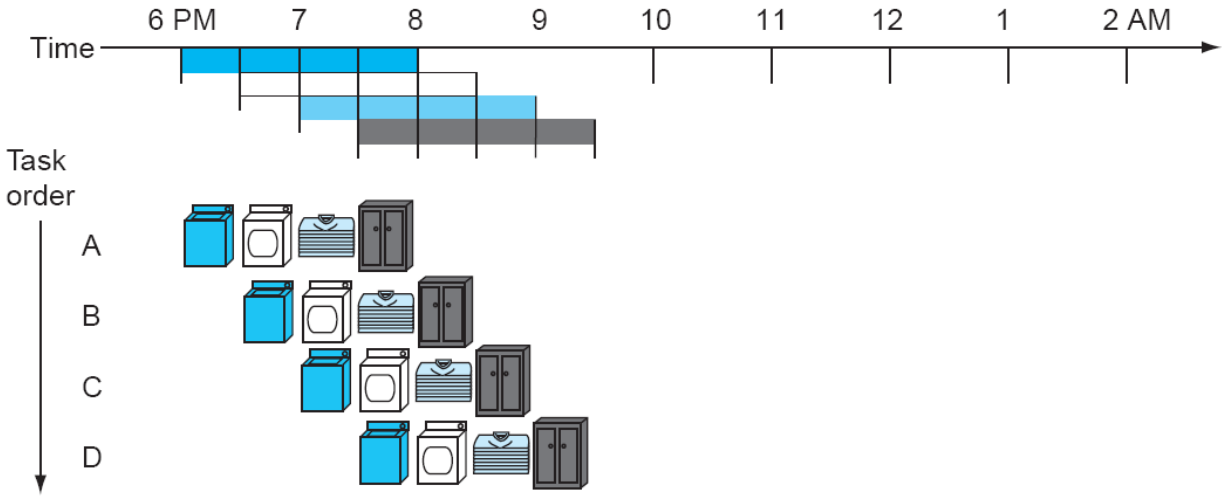
The laundry analogy

- Lets assume 30 minutes for each operation
- Doing four peoples laundry in sequence will take 8 hours
- We do not utilize all resources efficiently
  - The washer is used only for half an hour at 6, 8, 10 and 12
  - The dryer likewise the second half of these hours
  - We will be watching the washer and dryer and our roommate putting the cloths away, only when folding the cloths are we active



The laundry analogy - pipelined

- Still 30 minutes for each operation
- We now finish the washing in 3.5 hours
- The resources are used in parallel
  - We start the next wash when we have finished the first
  - From half past 7 the we are busy, the roommate is busy and the washer and dryer is both running
- A full wash still takes 2 hours
  - Throughput is increased.



The laundry analogy - some conclusions

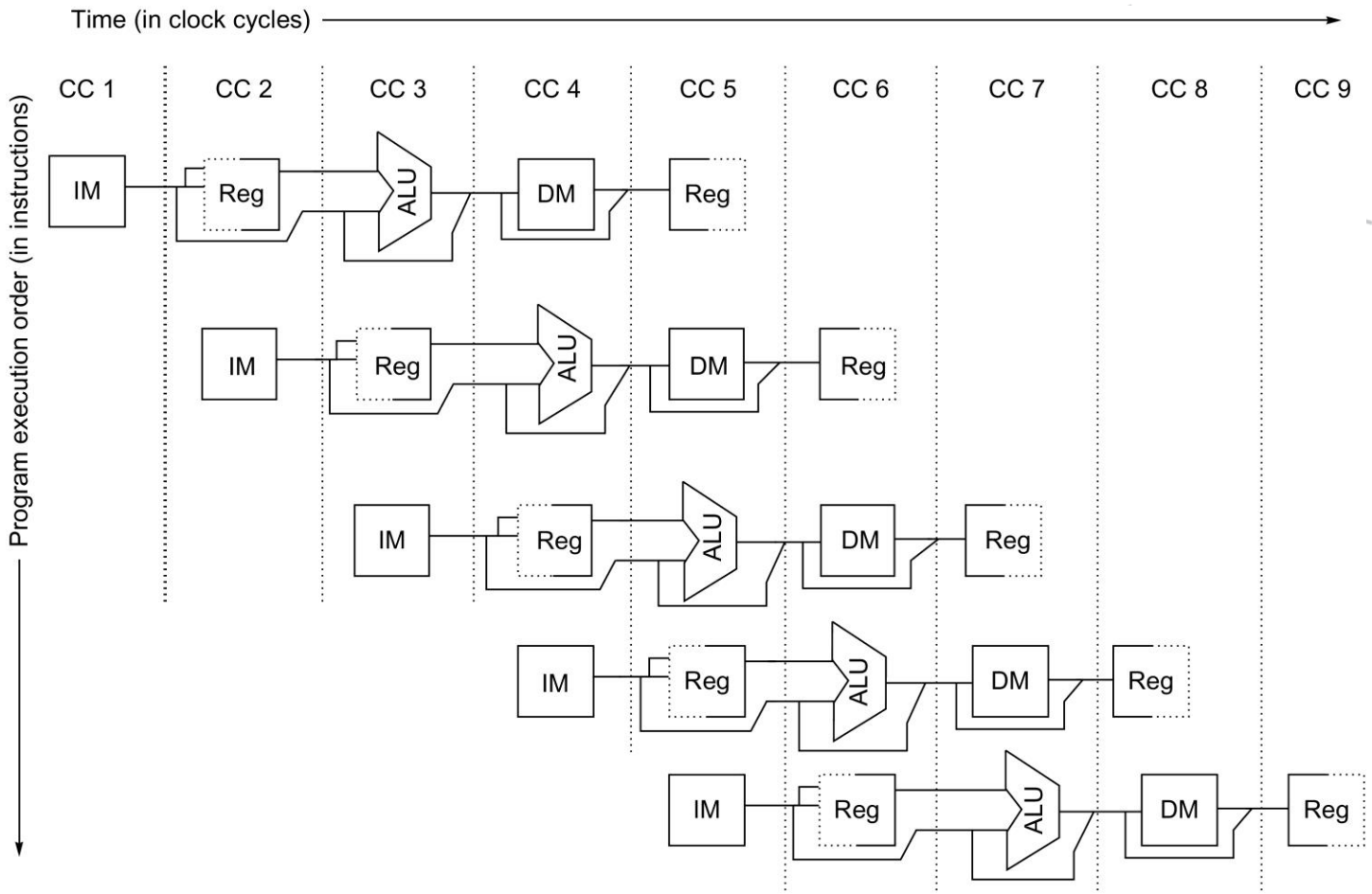
- With four loads the speedup is 8/3.5 = 2.3 times
- There is a start-up/wind-down cost of 3 half-hours

- Potentially the laundry can take one fourth of the time
- The speedup moves towards 4 times as the number of washes increase
- Hundred washes:
  - 200 hours sequential
  - 103 half-hours -> 51.5 hours pipelined
  - ~3.9 times as fast

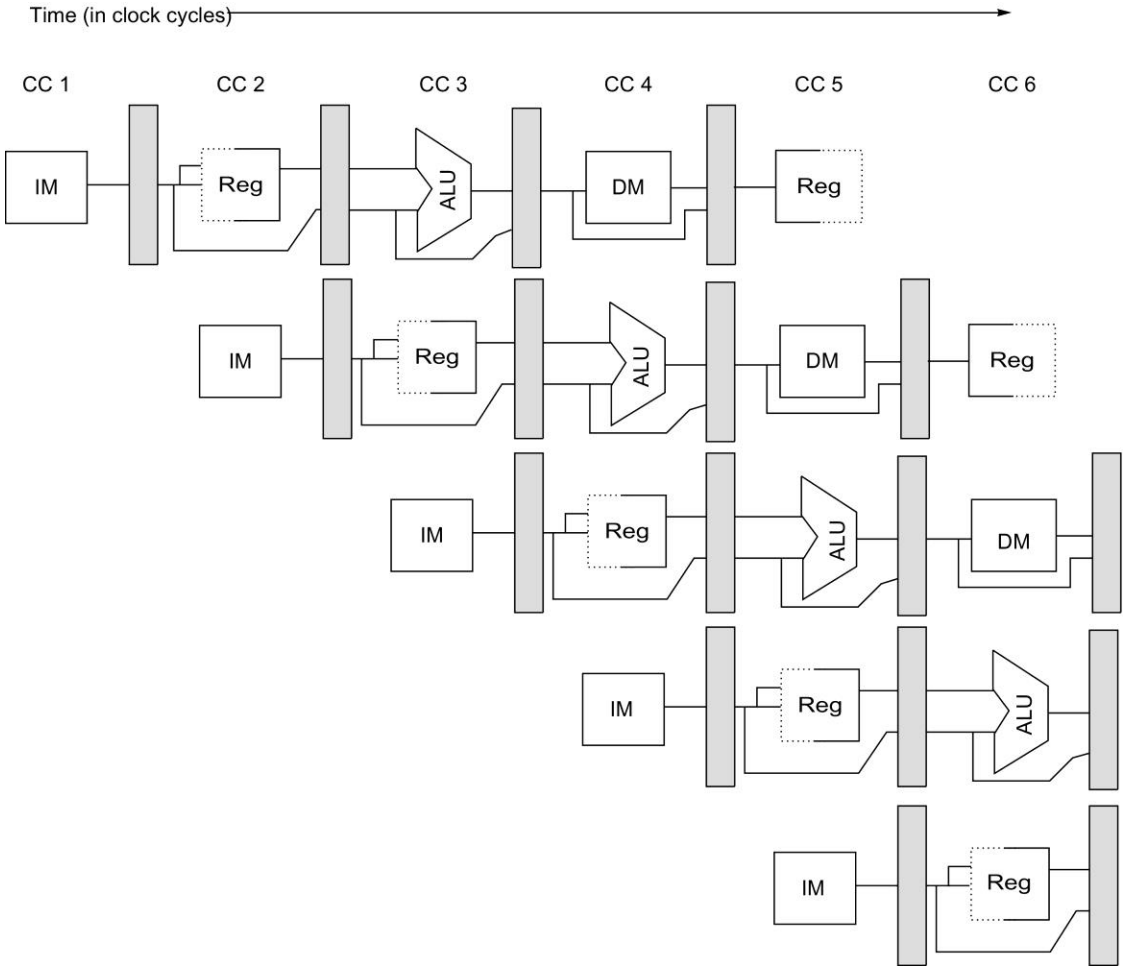
Pipelining: Big Picture

- Can you think of more examples of real-world pipelining?
- Does pipelining impact latency? (latency is the time to complete a single instruction)
- Does pipelining impact throughput? (throughput is the rate at which instructions are completed)

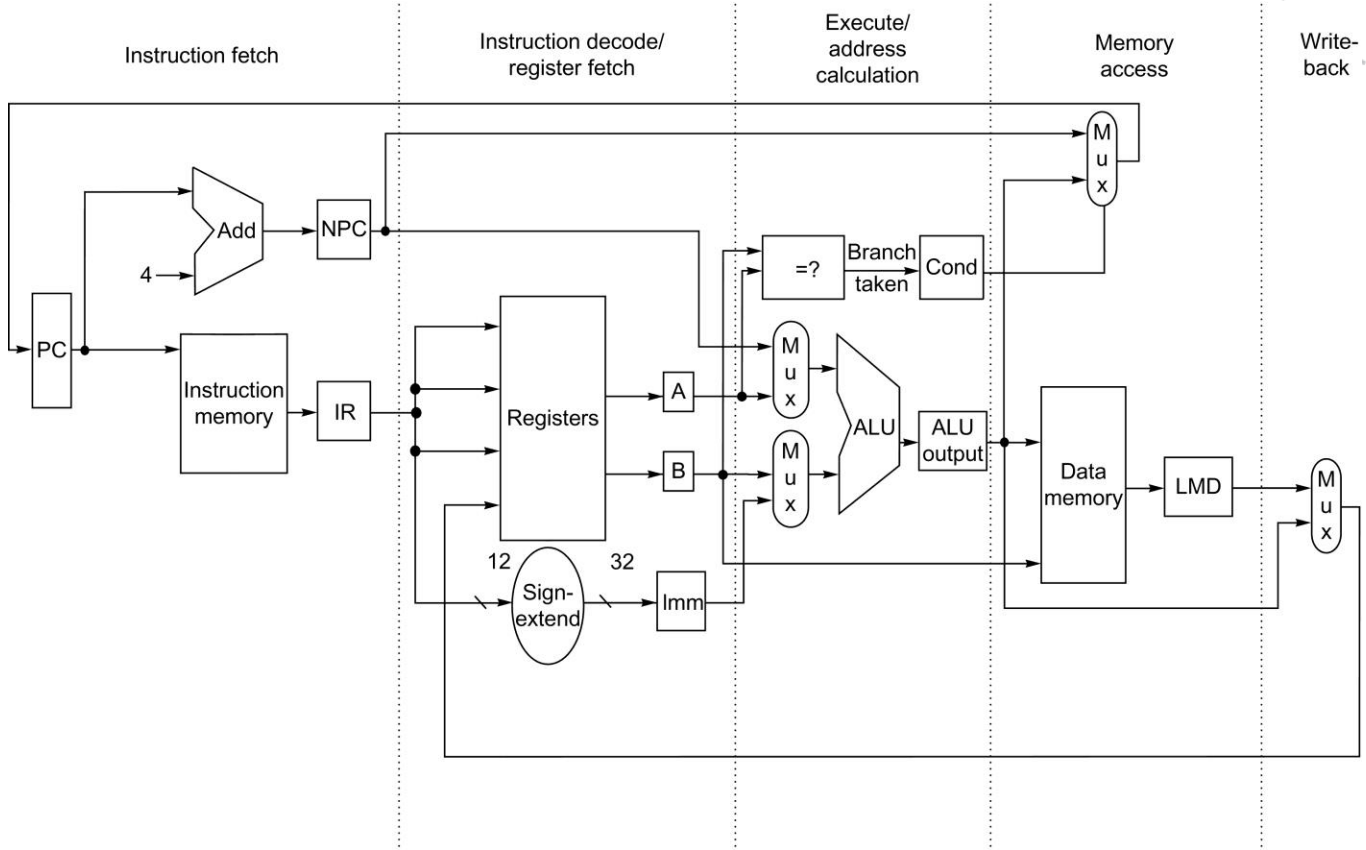
Pipelining the datapath



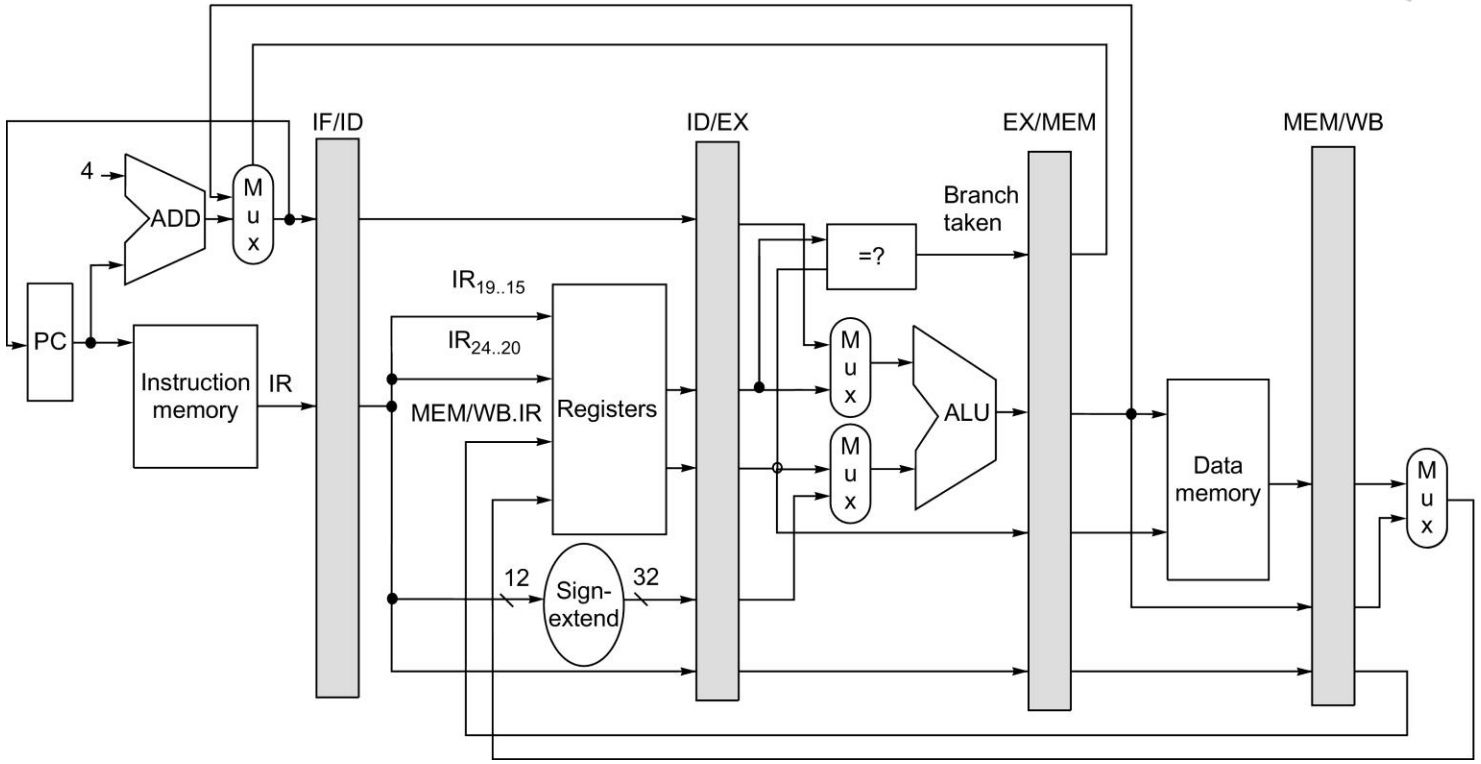
Pipeline Registers



RISC-V Datapath



RISC-V Datapath With Pipeline Registers



RISC-V Pipeline Events

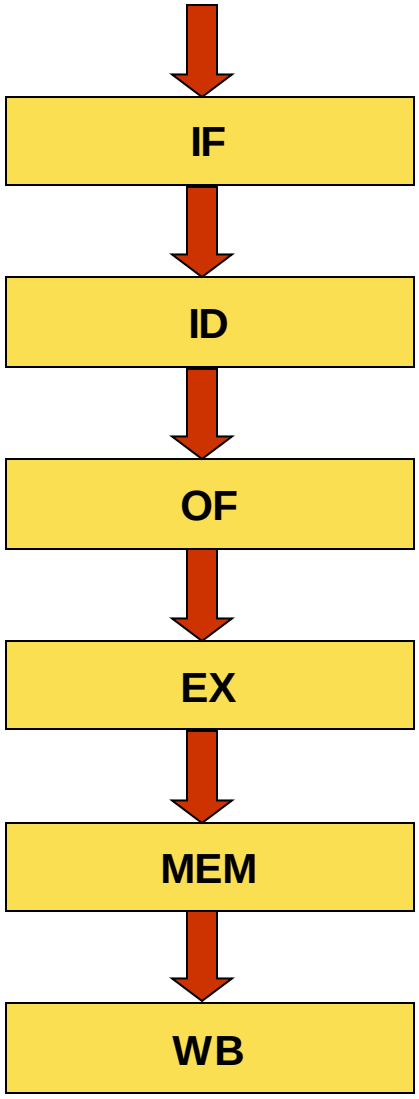
Stage	Any instruction		
IF	IF/ID.IR ← Mem[PC] IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM.ALUOutput} else {PC+4});		
ID	ID/EX.A ← Regs[IF/ID.IR[rs1]]; ID/EX.B ← Regs[IF/ID.IR[rs2]]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]);		
	ALU instruction	**Load instruction**	**Branch instruction**
EX	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B; or EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm;	EX/MEM.IR to ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A+ID/EX.Imm; EX/MEM.B ← ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm << 2); EX/MEM.cond ← (ID/EX.A == ID/EX.B);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] ← EX/MEM.B;	
WB	Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rd]] ← MEM/WB.LMD;	

PIPELINING HAZARDS

Slides in this section are by Lieven Eeckhout, Ghent University. Reused with permission.

Instruction Pipeline

Note:
In the RISC-V pipeline there is only a single stage for ID and OF.



Instruction Fetch

Instruction Decode

Operand Fetch

Execute

Memory Read/Write

Write Back

Why don’t we have perfect pipeline behavior?

Program dependences

- Three types
  - Data dependence through memory
  - Data dependence through registers
  - Control dependence

- A hazard is a result of not respecting a program dependence
- A hazard should therefore NOT happen!

Data Dependences

- Real dependence= read-after-write (RAW)

```
V3 ← V1 op V2
V4 ← V3 op V5
```

- Anti dependence = write-after-read (WAR)

```
V3 ← V1 op V2
V1 ← V4 op V5
```

- Output dependence = write-after-write (WAW)

```
V3 ← V1 op V2
V3 ← V4 op V5
```

Hazards due to memory dependences?

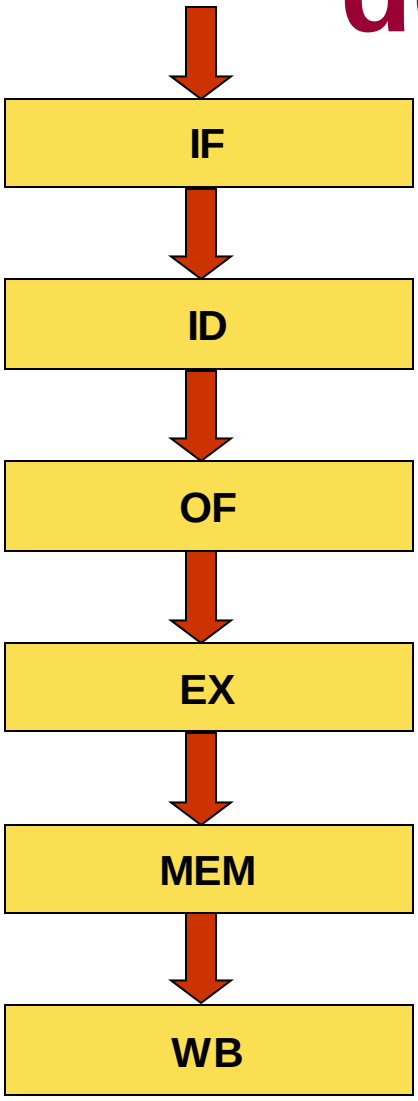
- Only the MEM stage reads/writes to/from memory
- All accessess to memory execute sequentially and in program order

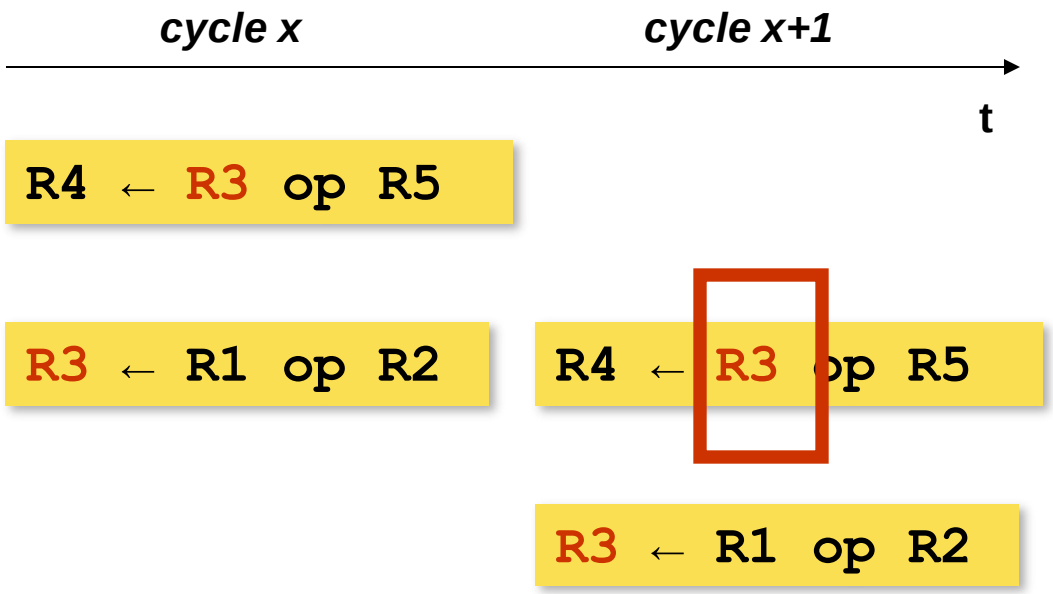
- Answer: NO

Hazards due to register data dependences?

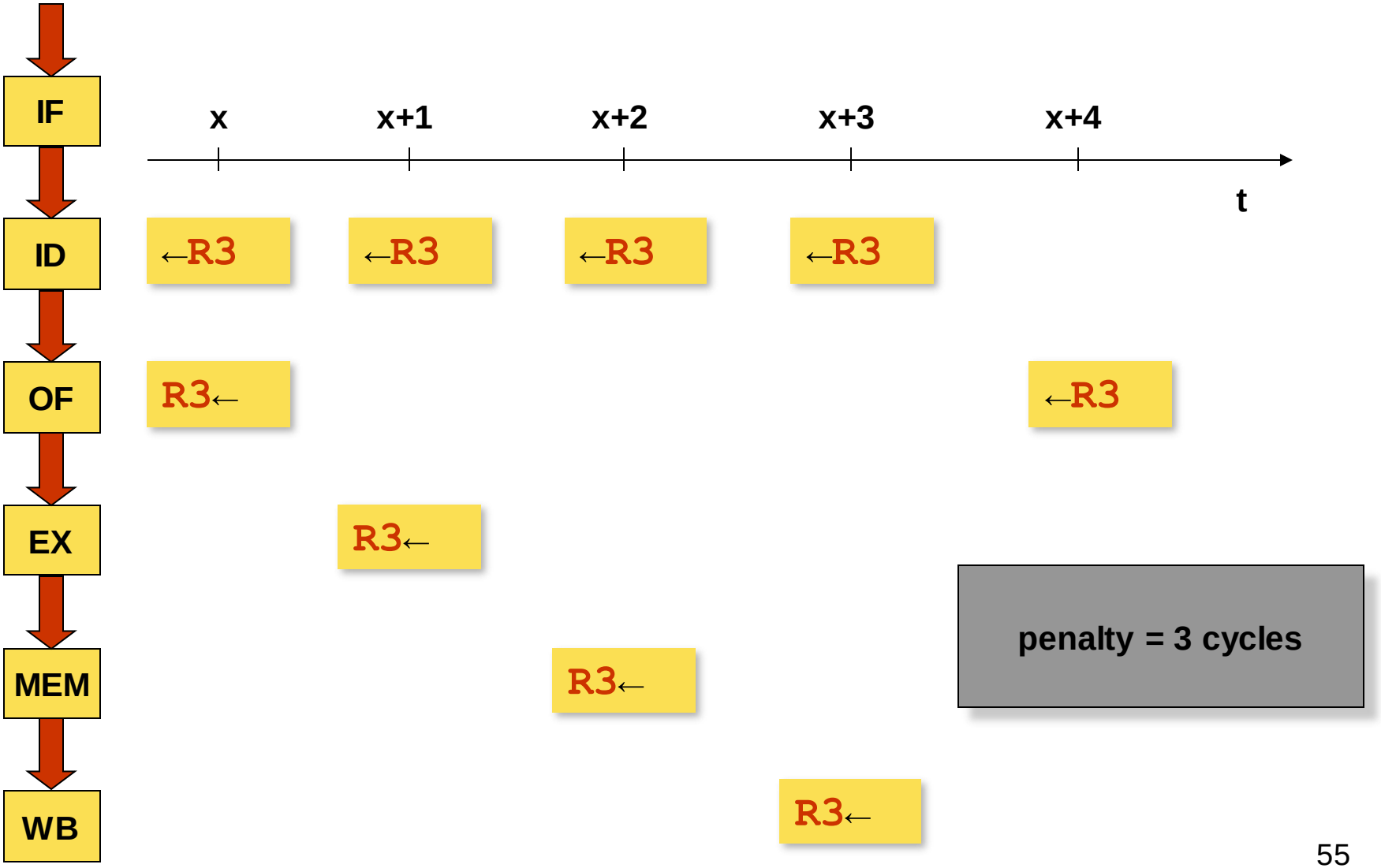
- Hazard due to WAW register dependence?
  - NO: only WB stage writes, and does so sequentially and in program order
- Hazard due to WAR register dependence?
  - NO: reading is done in OF stage; writing is done in WB stage
- Hazard due to RAW register dependence?
  - YES: may happen if an instruction reads an old value from the register file

Hazard due to RAW register dependence

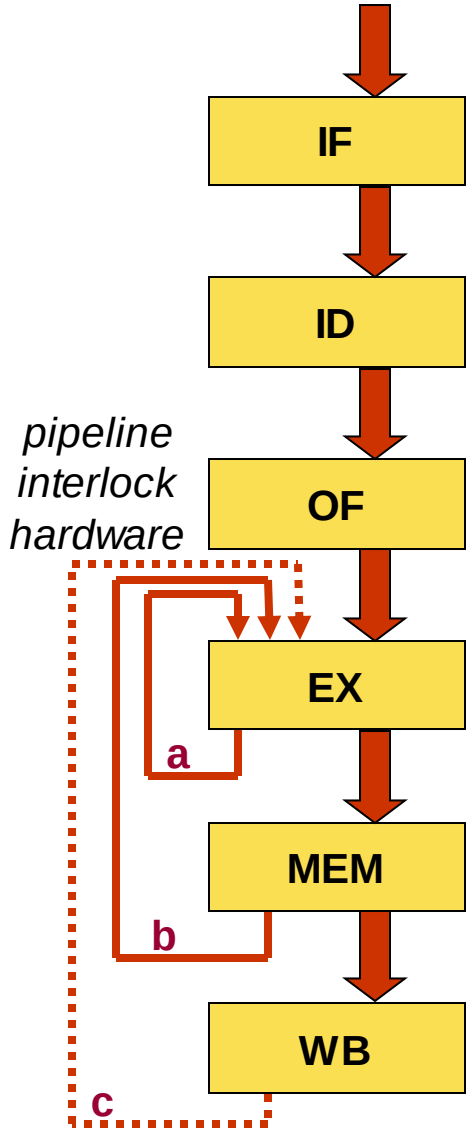




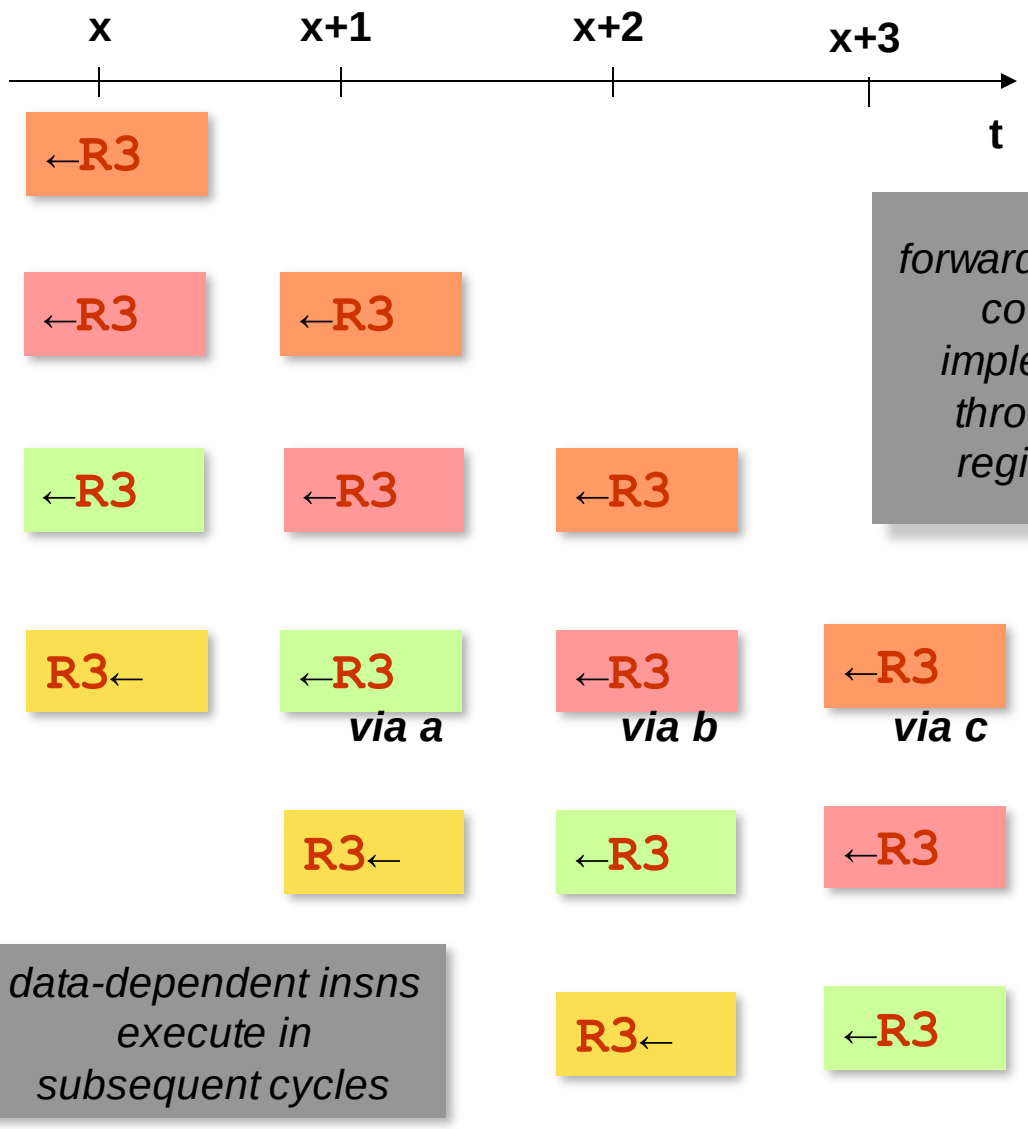
Naive solution: Pipeline Stall



Better solution: Forwarding



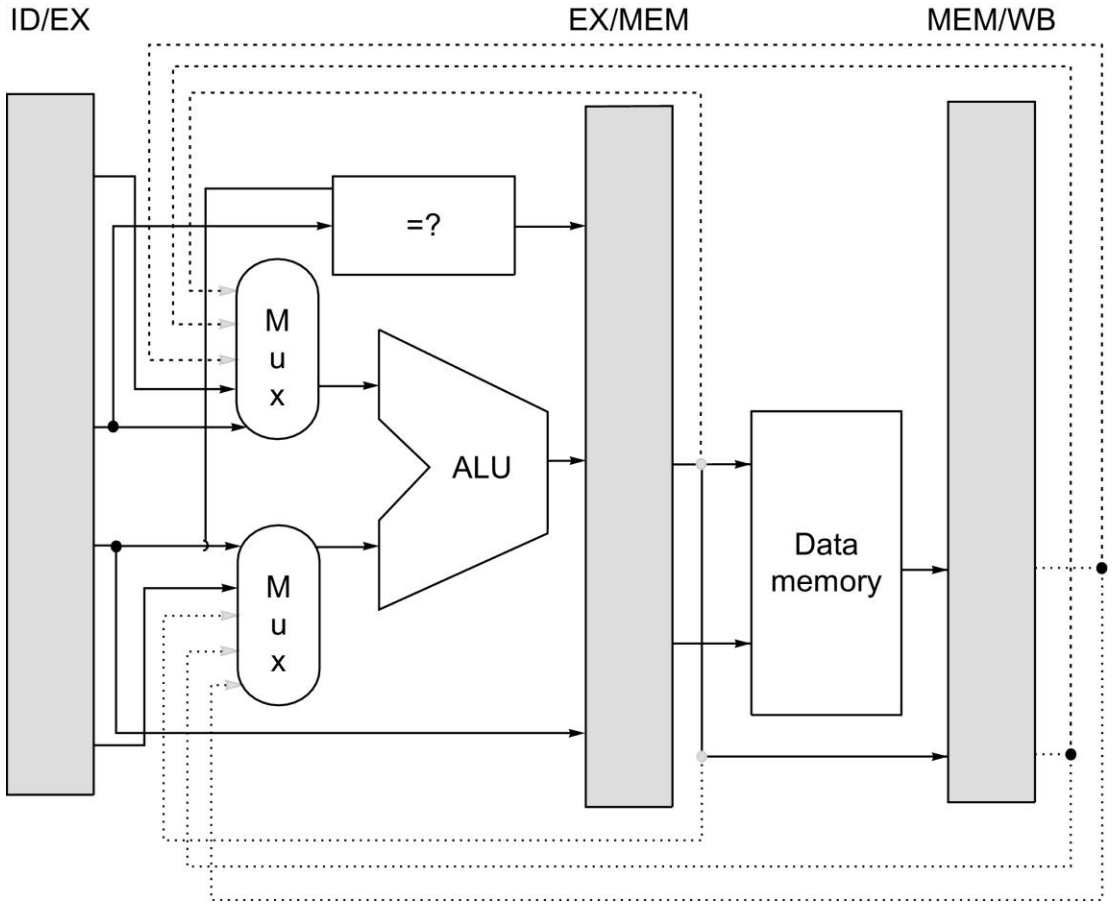
pipeline interlock hardware



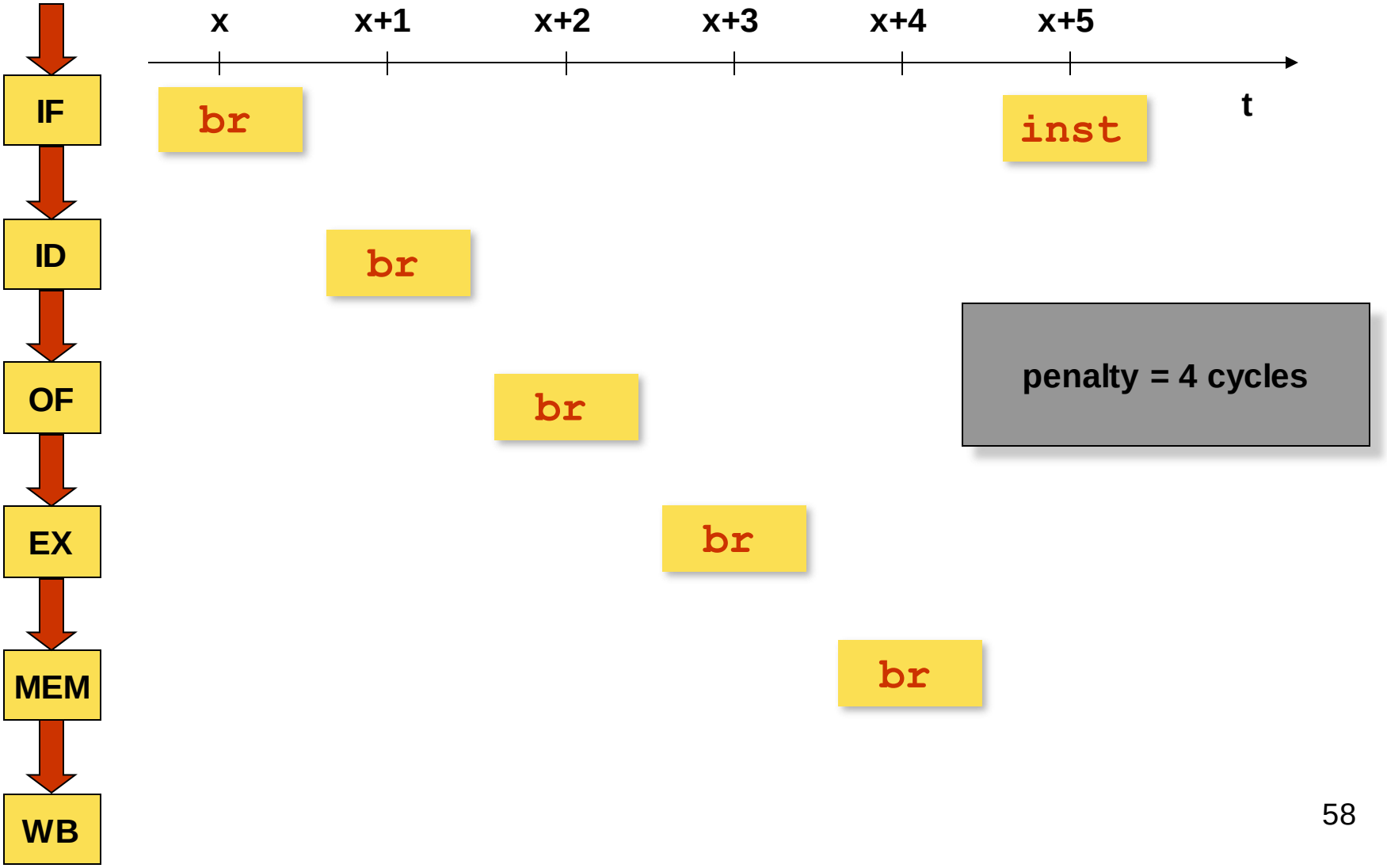
forwarding path c could be implemented through the register file

data-dependent insns execute in subsequent cycles

RISC-V Forwarding Example

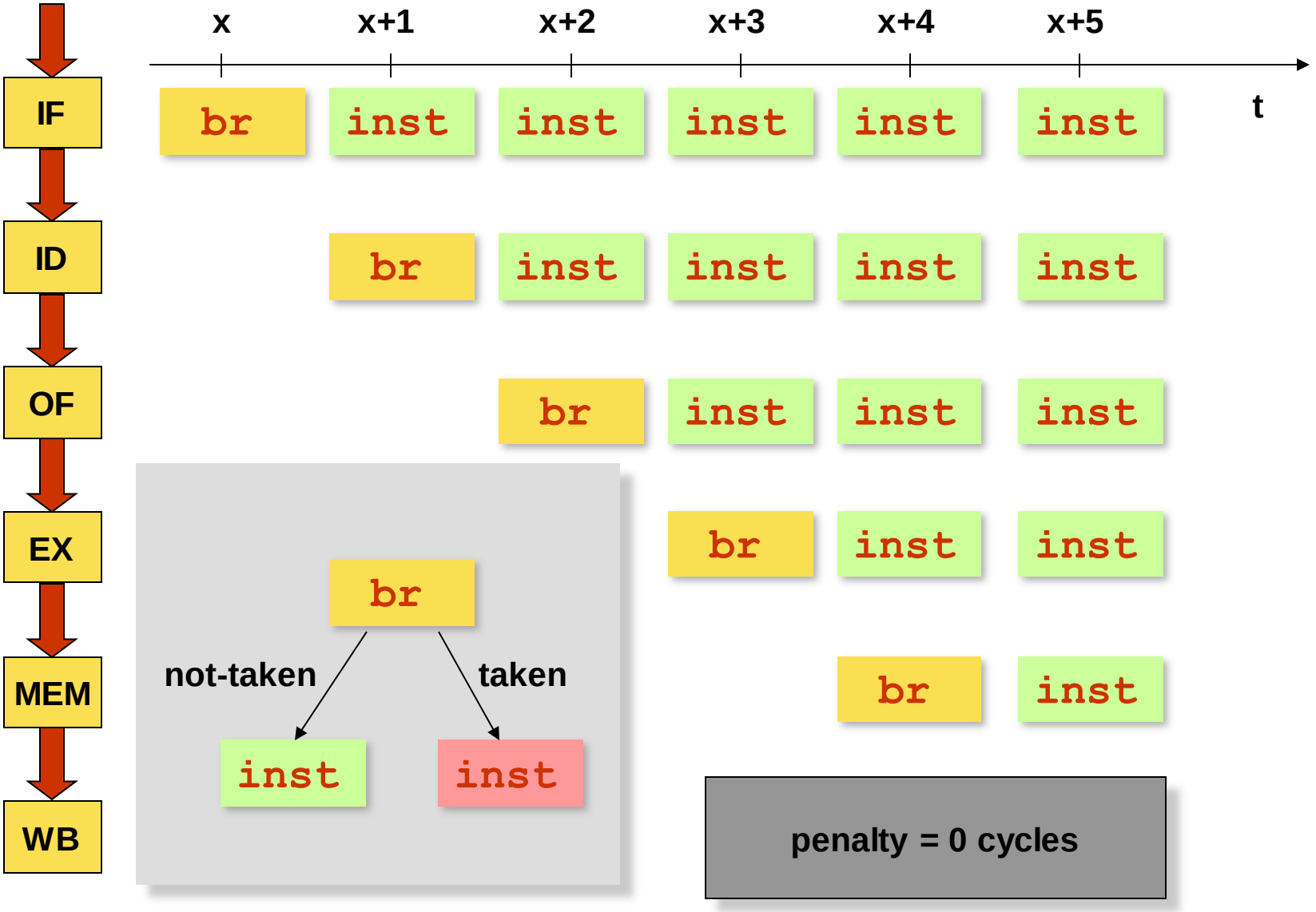


Hazard due to control dependence

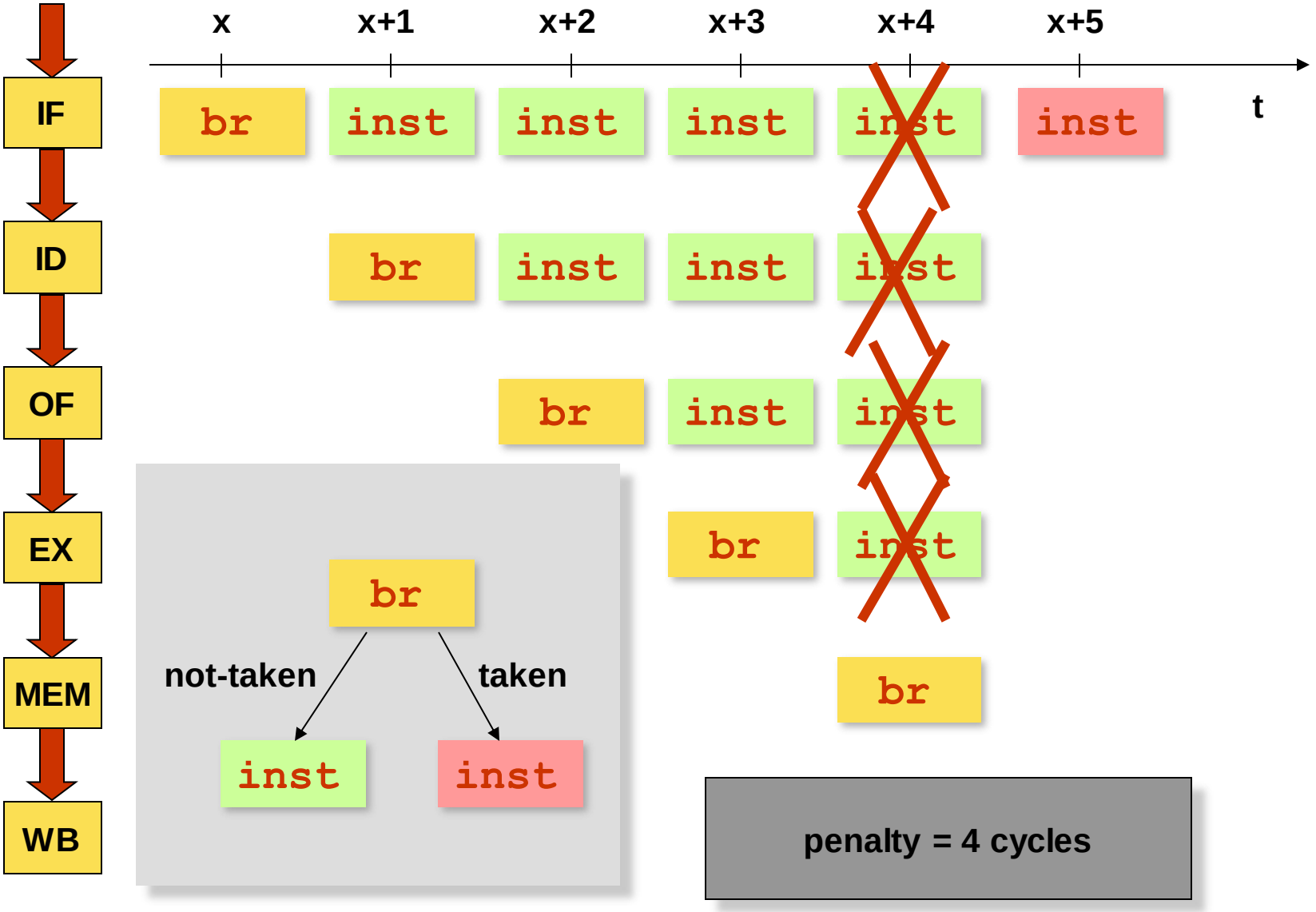


Solution: Branch Prediction & Speculative Execution

Upon a correct prediction



Upon a misprediction



Optimal pipeline depth

Deeper pipelines:

- higher clock frequency f
- higher cost due to mispredictions → higher CPI

Hence: there is an optimum

Also, dynamic power consumption increases with clock frequency! ($P \sim f^3$)

Question

- How important is branch prediction in a 5-stage pipeline?

SUMMARY

Summary

- Single cycle processor
  - Simplest possible implementation
- Pipelined processor
  - “Minimal” increment on single cycle processor
  - Can improves performance significantly by exploiting parallelism across independent instructions (in time)
  - Dependent instructions cause hazards which needs to be handled to ensure that the programs execute correctly.
  - Stalling on all hazards needlessly waste performance
  - Key techniques: Stalling, forwarding and branch prediction.