The Processor

Adapted and Supplemented by,
Dr. R. Shathanaa

MIPS ISA

- Microprocessor without Interlocked Pipelined Stages

Arithmetic and Logical Instructions

Instruction	Operation
add $d, $s, $t	`$d = $s + $t`
addu $d, $s, $t	`$d = $s + $t`
addi $t, $s, i	`$t = $s + SE(i)`
addiu $t, $s, i	`$t = $s + SE(i)`
and $d, $s, $t	`$d = $s & $t`
andi $t, $s, i	`$t = $s & ZE(i)`
div $s, $t	`lo = $s / $t; hi = $s % $t`
divu $s, $t	`lo = $s / $t; hi = $s % $t`
mult $s, $t	`hi:lo = $s * $t`
multu $s, $t	`hi:lo = $s * $t`
nor $d, $s, $t	`$d = ~($s | $t)`
or $d, $s, $t	`$d = $s | $t`
ori $t, $s, i	`$t = $s | ZE(i)`
sll $d, $t, a	`$d = $t << a`
sllv $d, $t, $s	`$d = $t << $s`
sra $d, $t, a	`$d = $t >> a`
srav $d, $t, $s	`$d = $t >> $s`
srl $d, $t, a	`$d = $t >>> a`
srlv $d, $t, $s	`$d = $t >>> $s`

Comparison Instructions

Instruction	Operation
slt $d, $s, $t	`$d = ($s < $t)`
sltu $d, $s, $t	`$d = ($s < $t)`
slti $t, $s, i	`$t = ($s < SE(i))`
sltiu $t, $s, i	`$t = ($s < SE(i))`

Instruction	Operation
beq $s, $t, label	`if ($s == $t) pc += i << 2`
bgtz $s, label	`if ($s > 0) pc += i << 2`
blez $s, label	`if ($s <= 0) pc += i << 2`
bne $s, $t, label	`if ($s != $t) pc += i << 2`

Jump Instructions

Instruction	Operation
j label	`pc += i << 2`
jal label	`$31 = pc; pc += i << 2`
jalr $s	`$31 = pc; pc = $s`
jr $s	`pc = $s`

Load Instructions

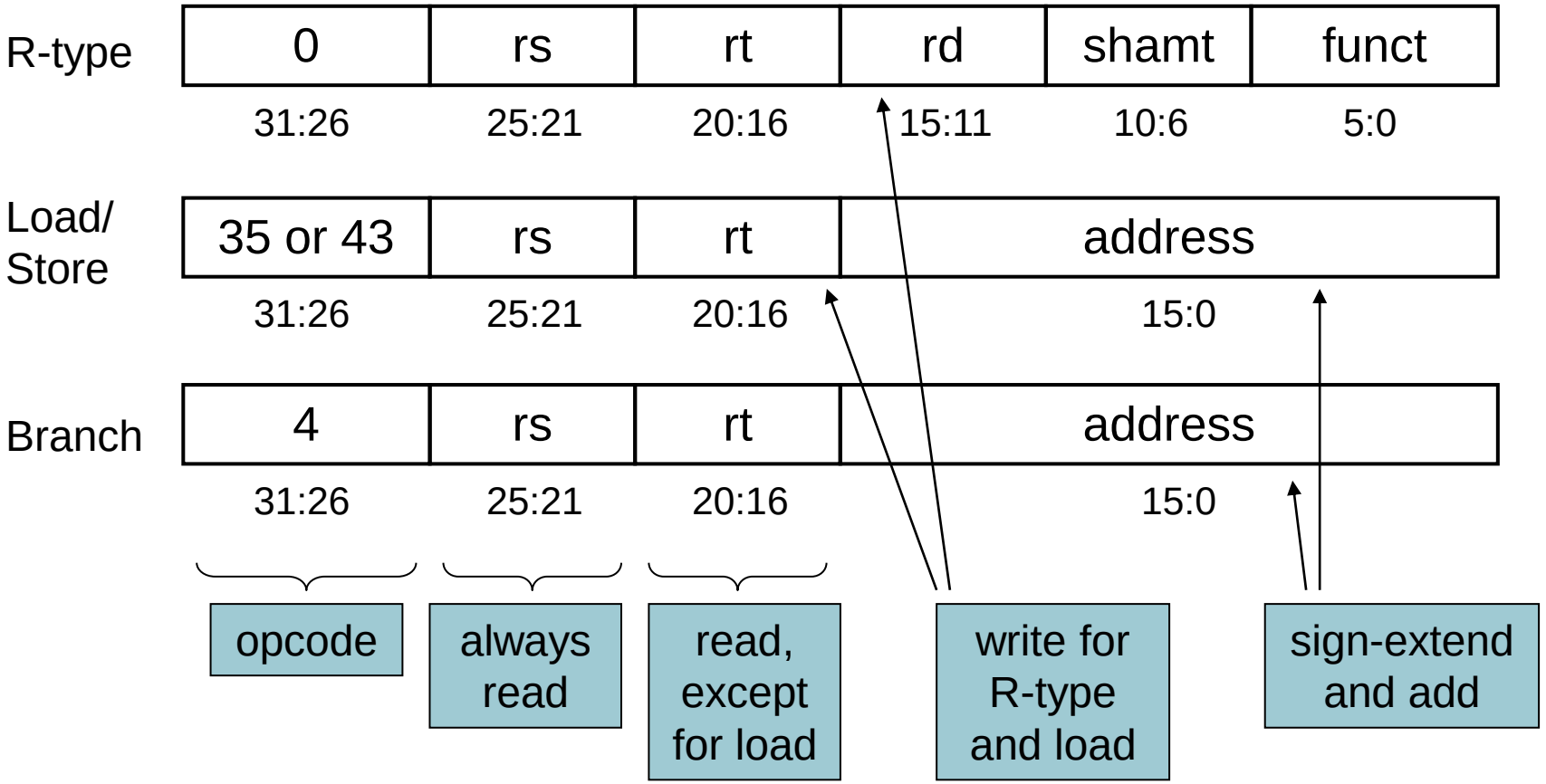
Instruction	Operation
lb $t, i($s)	`$t = SE (MEM [$s + i]:1)`
lbu $t, i($s)	`$t = ZE (MEM [$s + i]:1)`
lh $t, i($s)	`$t = SE (MEM [$s + i]:2)`
lhu $t, i($s)	`$t = ZE (MEM [$s + i]:2)`
lw $t, i($s)	`$t = MEM [$s + i]:4`

Store Instructions

Instruction	Operation
sb $t, i($s)	`MEM [$s + i]:1 = LB ($t)`
sh $t, i($s)	`MEM [$s + i]:2 = LH ($t)`
sw $t, i($s)	`MEM [$s + i]:4 = $t`

The Main Control Unit

- Control signals derived from instruction



Introduction

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

Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → 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 ← target address or PC + 4

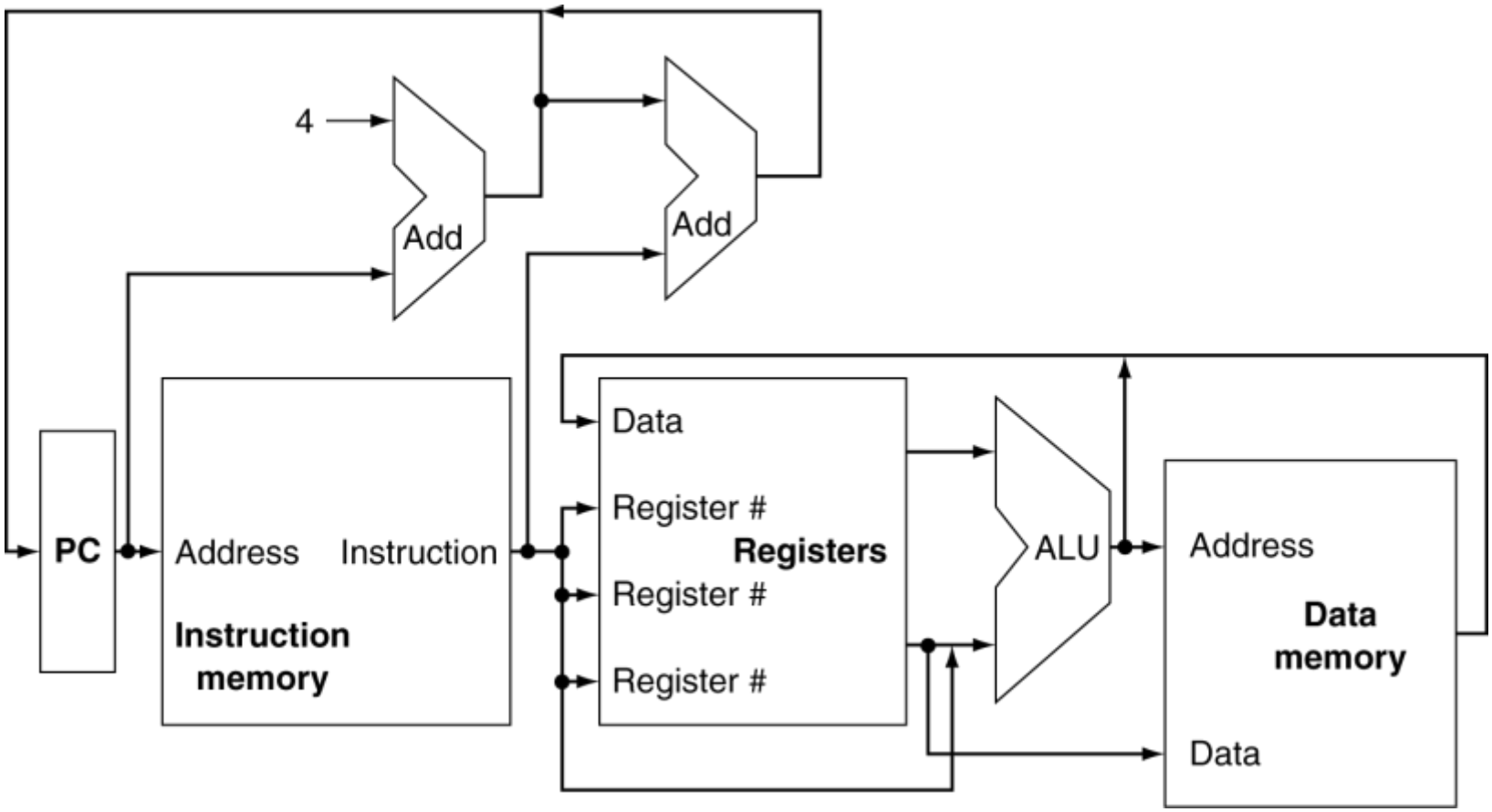
Instruction Execution

1. Read instruction from instruction memory
2. Decode instruction and read operands

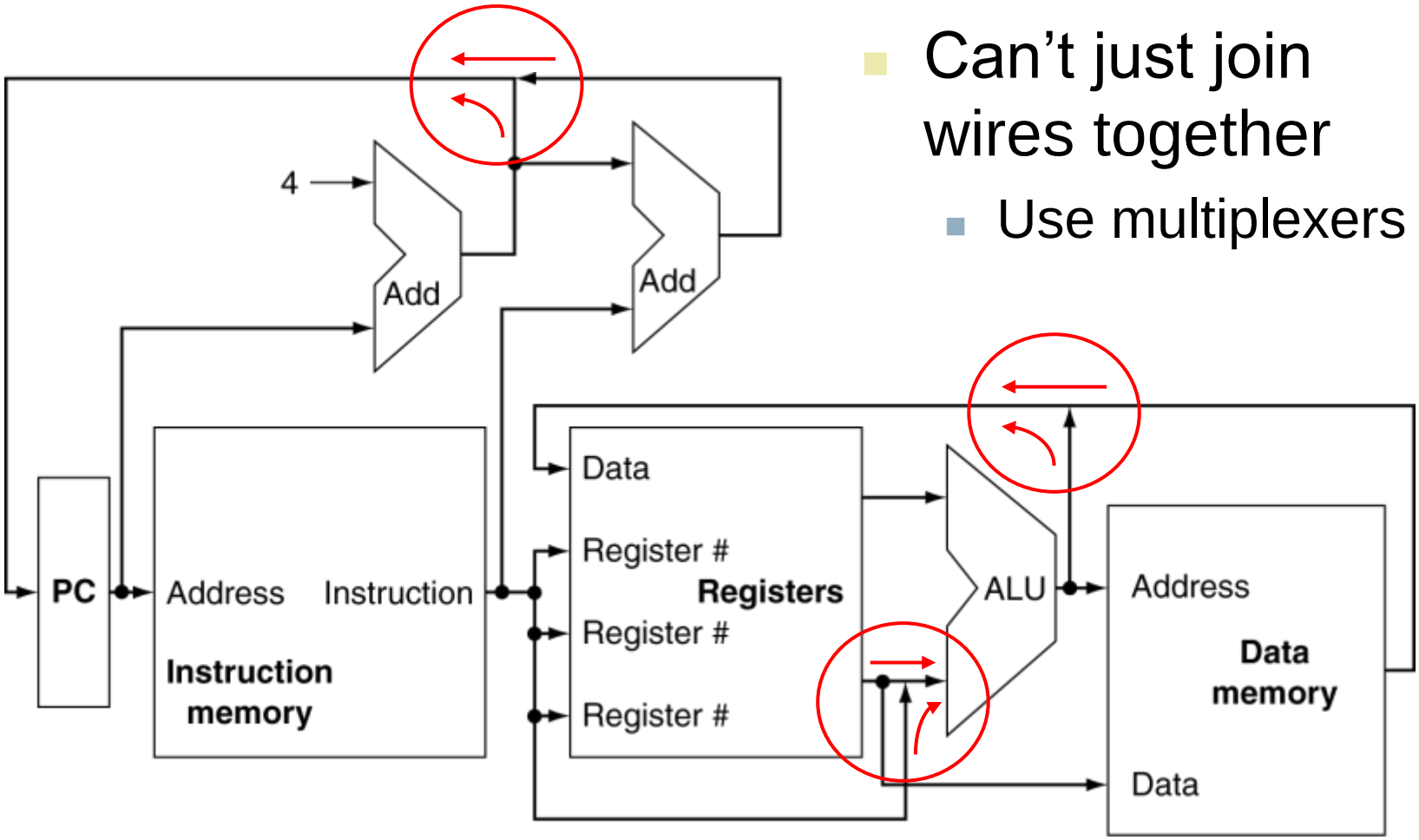
- Arithmetic (add $s1, $s2, $s3)
  3. Perform arithmetic operation (add, sub, etc.)
  4. Write the result to destination register
- Load / Store (lw $s1, 0($t1))
  3. Calculate Effective Address
  4. Read from memory (Write to memory incase of store)
  5. Write the read value to the destination register (For load alone)
- Branch (bne $t1,$t2, loop)
  3. Calculate branch outcome and branch target address

Final step: Update PC with PC+4 or branch target address.

CPU Overview

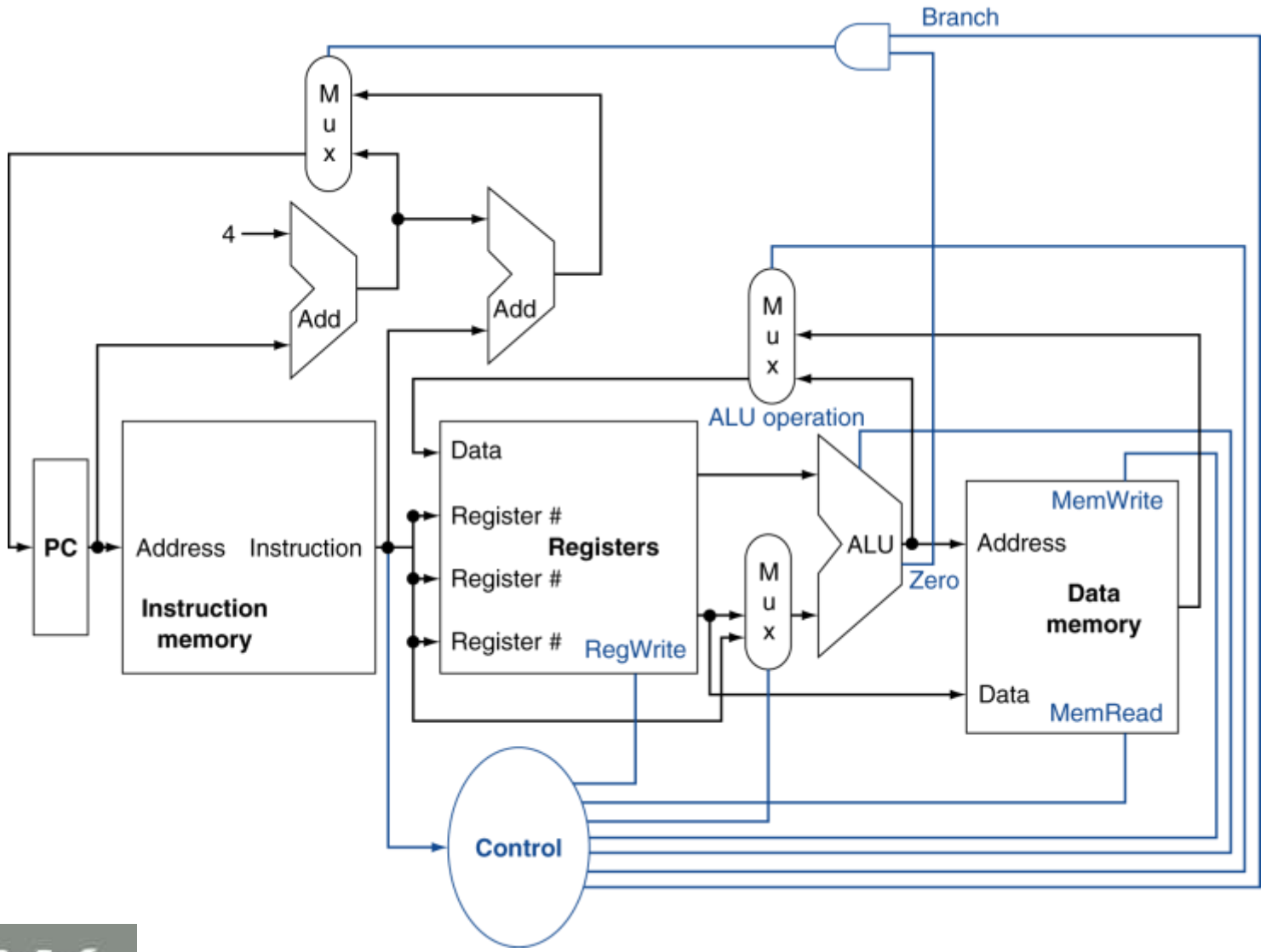


Multiplexers



- Can't just join wires together
  - Use multiplexers

Control

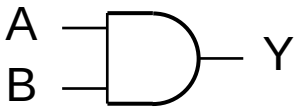

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Logic Design Basics

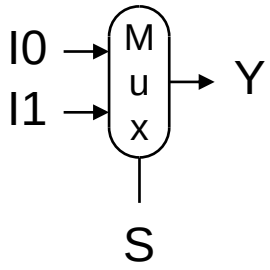
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

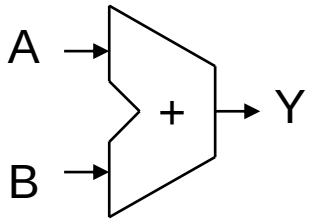
- AND-gate
  - Y = A & B



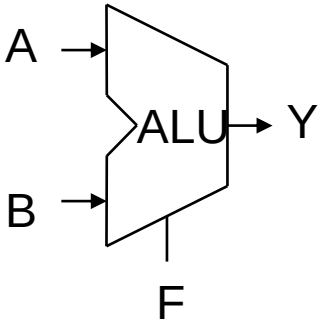
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B



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



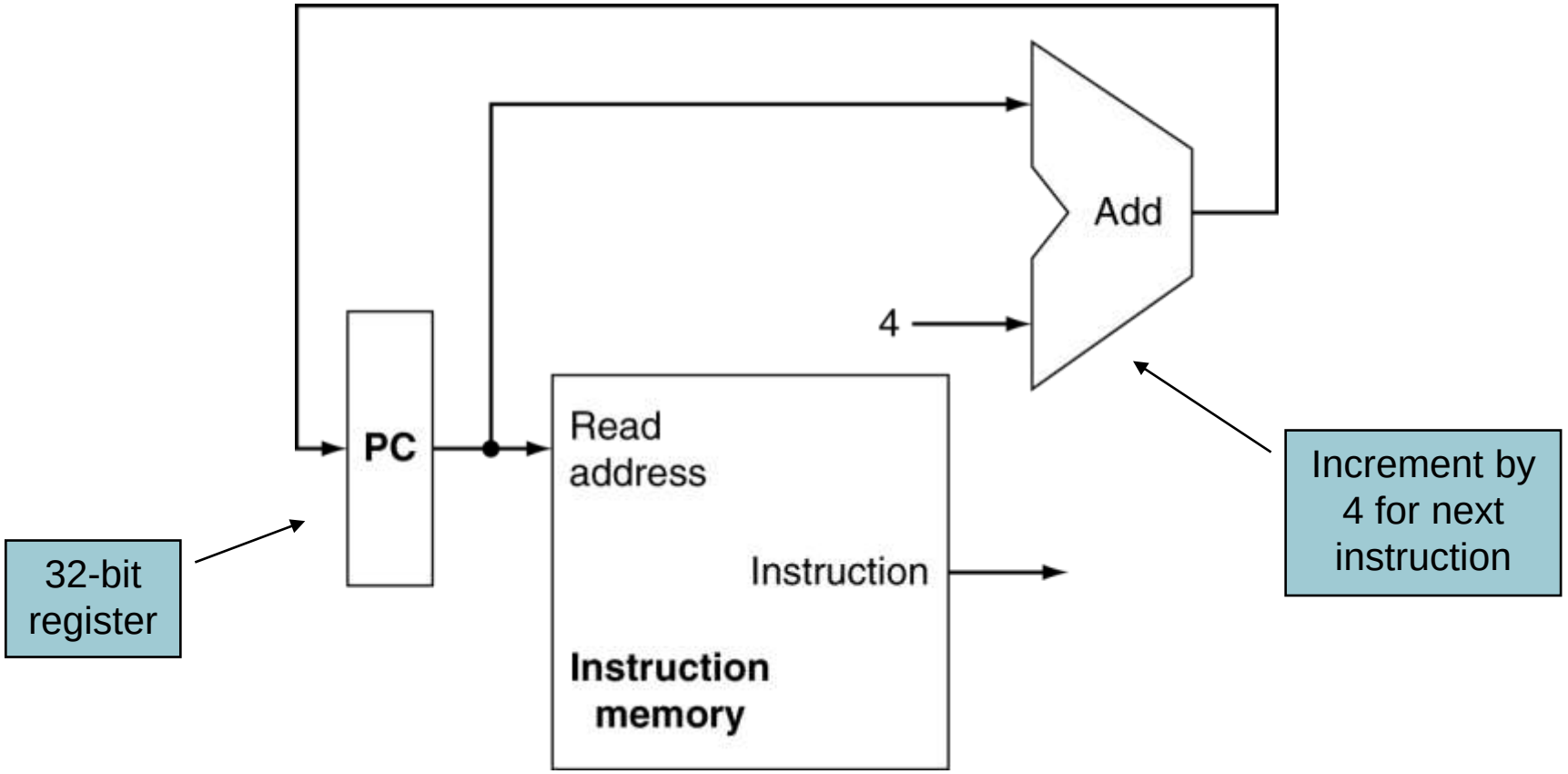
Sequential Elements

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

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
  - Refining the overview design

Instruction Fetch



R-Format Instructions

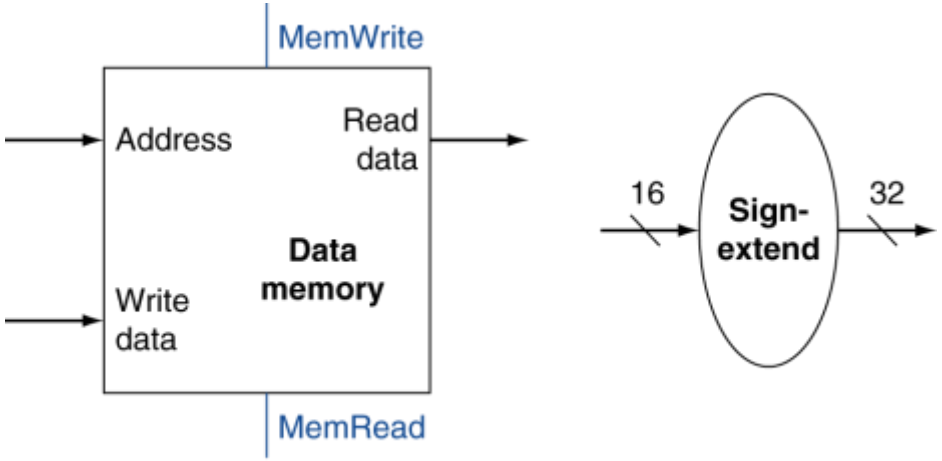
- Read two register operands ADD Perform arithmetic/logical operation
- Write register result

ADD $S1,$S2,$S3

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

LD $T1, 8($S1)



a. Data memory unit

b. Sign extension unit

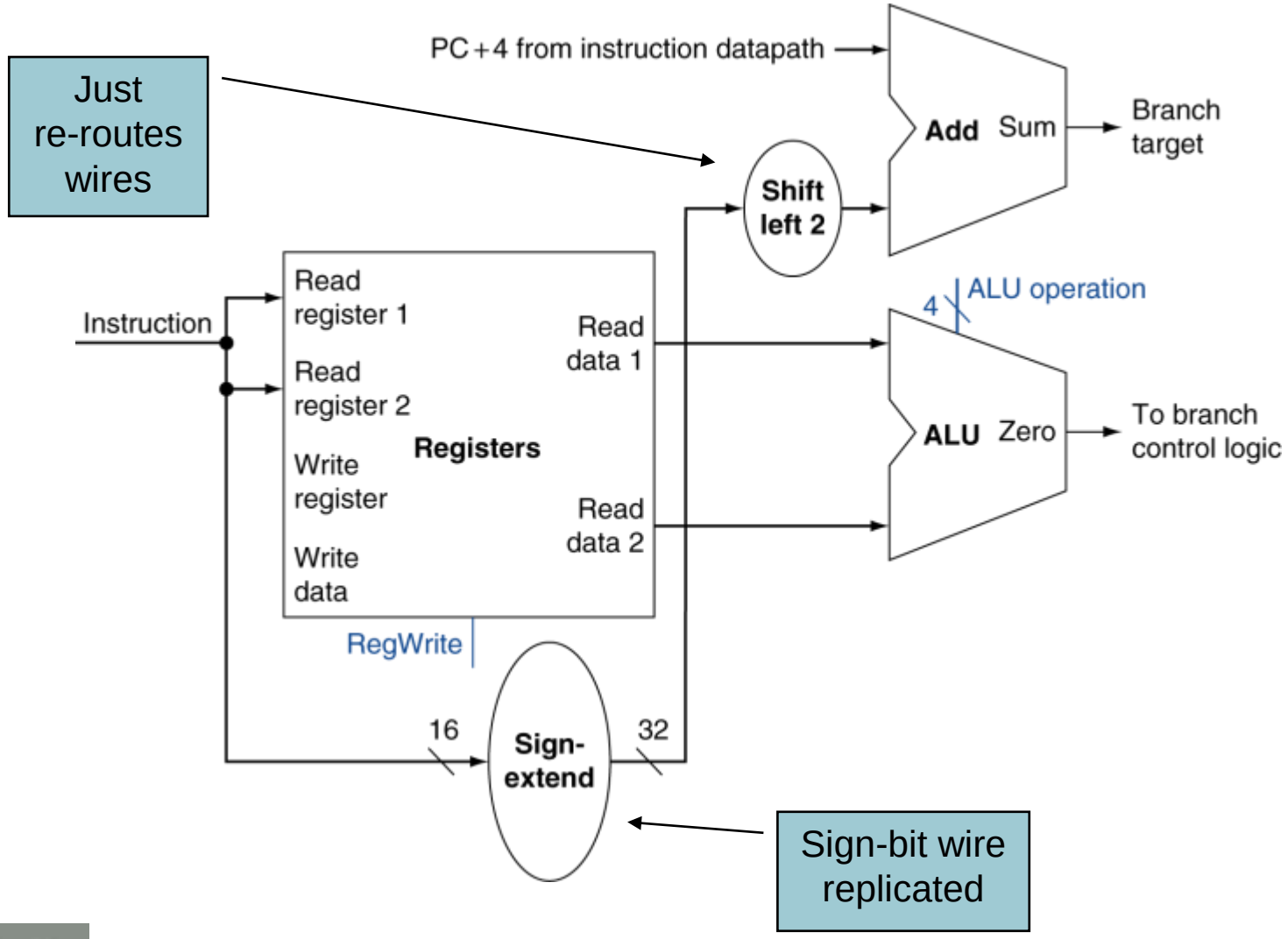
Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

BEQ $T1, $T2, LOOP

Branch Instructions

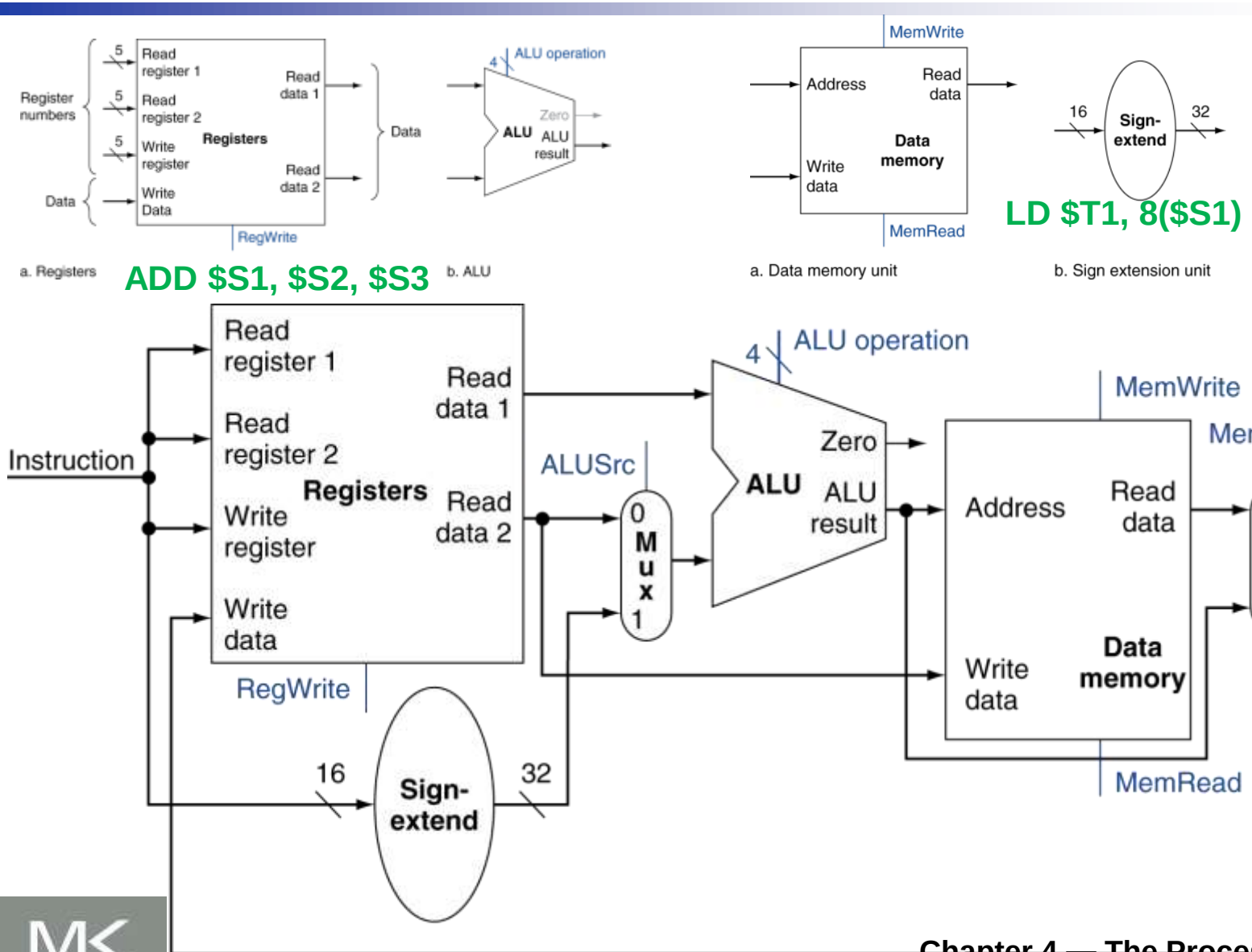
BEQ $T1, $T2, LOOP



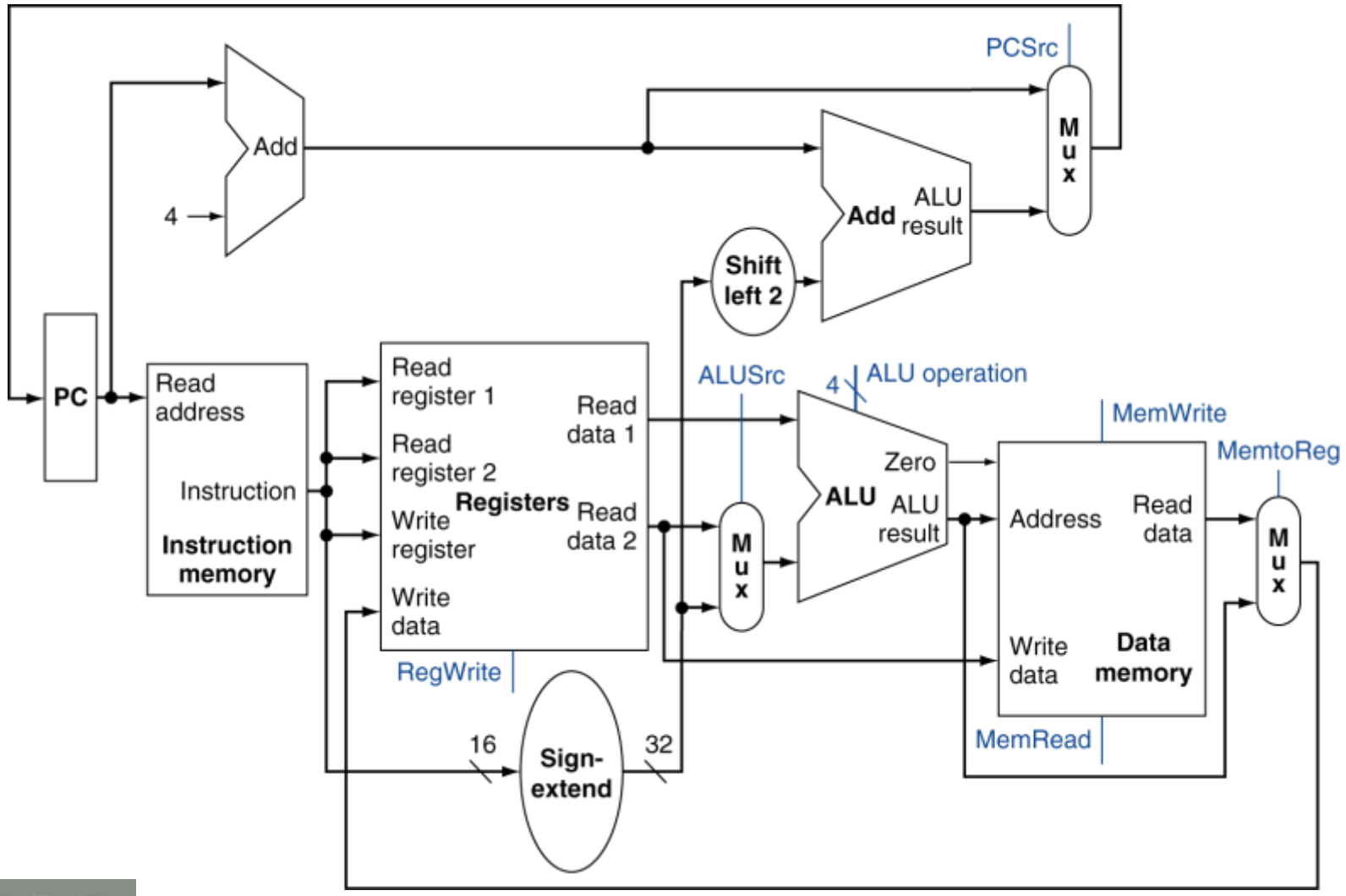
Composing the Elements

- 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
- Reuse units and use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



Full Datapath



ALU Control

Memory reference: `lw`, `sw`
Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
Control transfer: `beq`, `j`

- 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

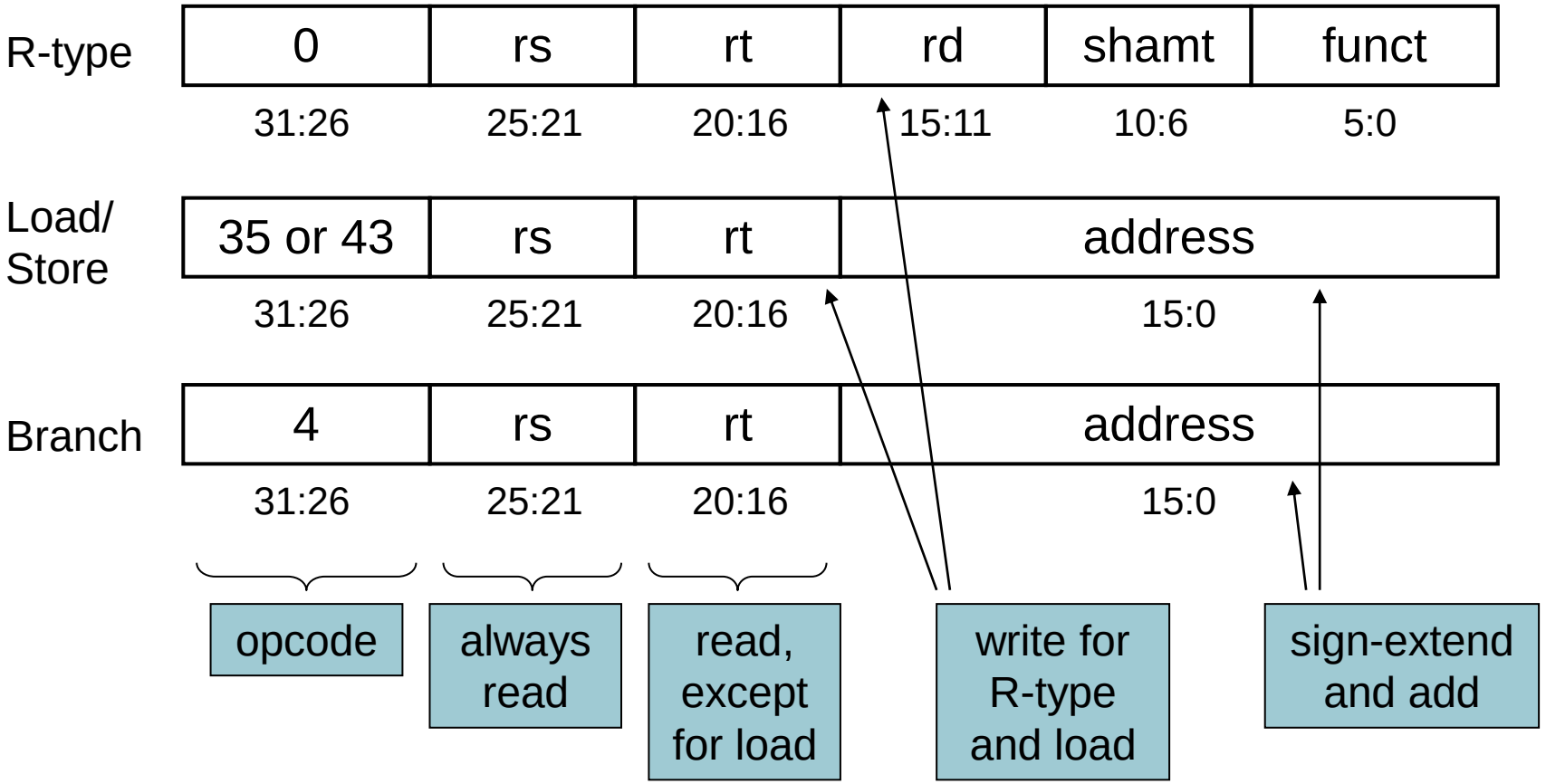
ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control
  - Multiple levels of control – reduces size of control

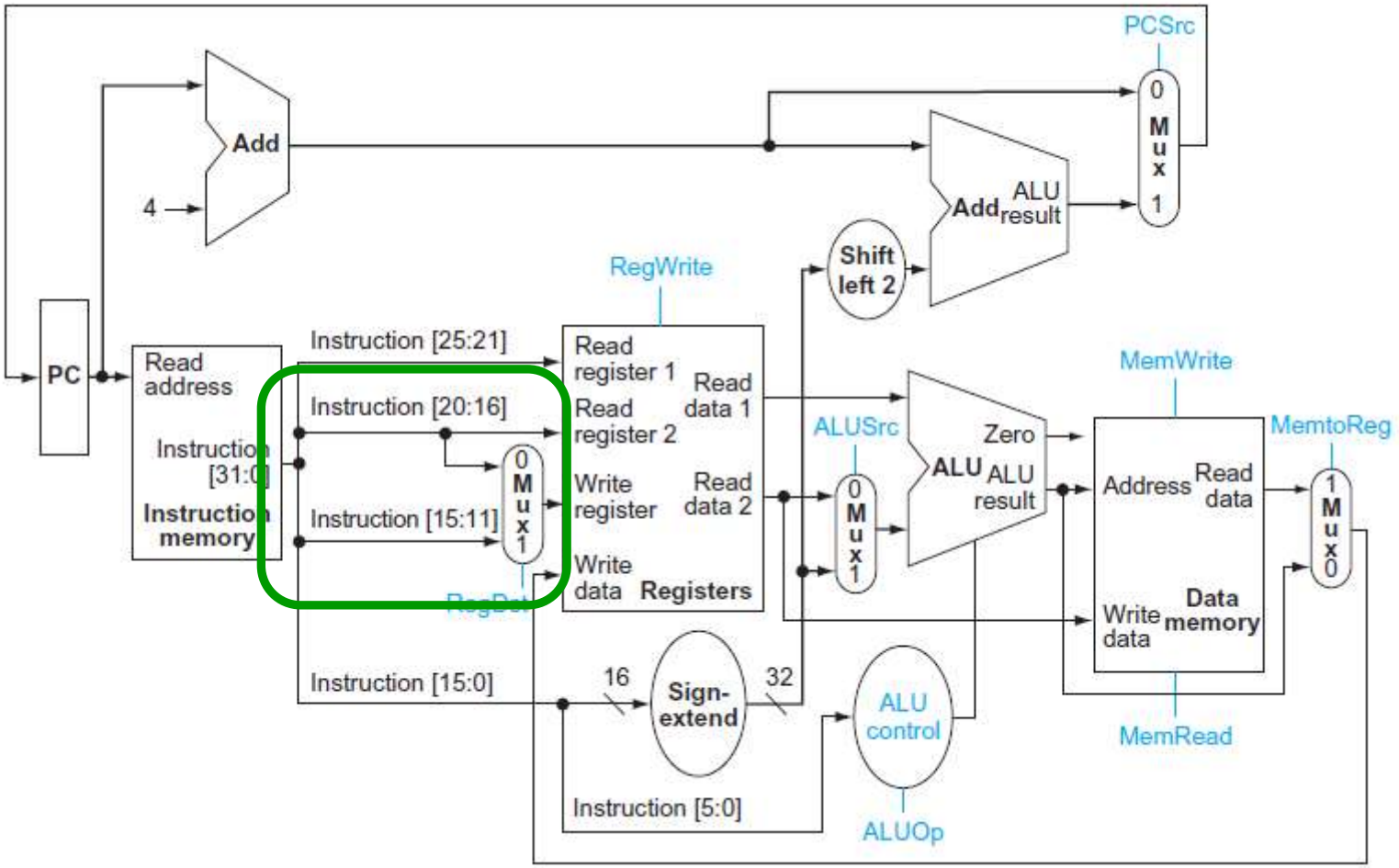
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than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

The Main Control Unit

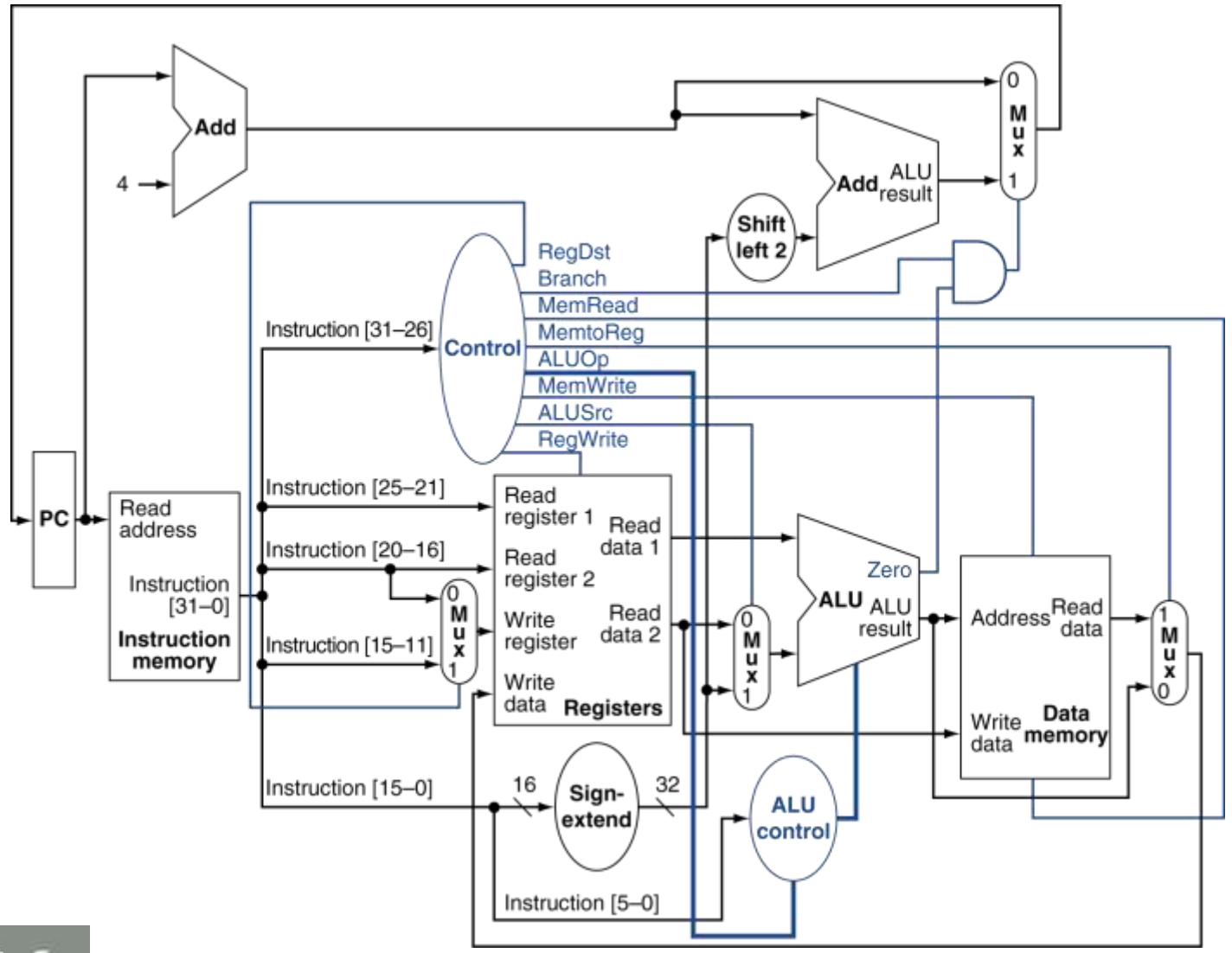
- Control signals derived from instruction



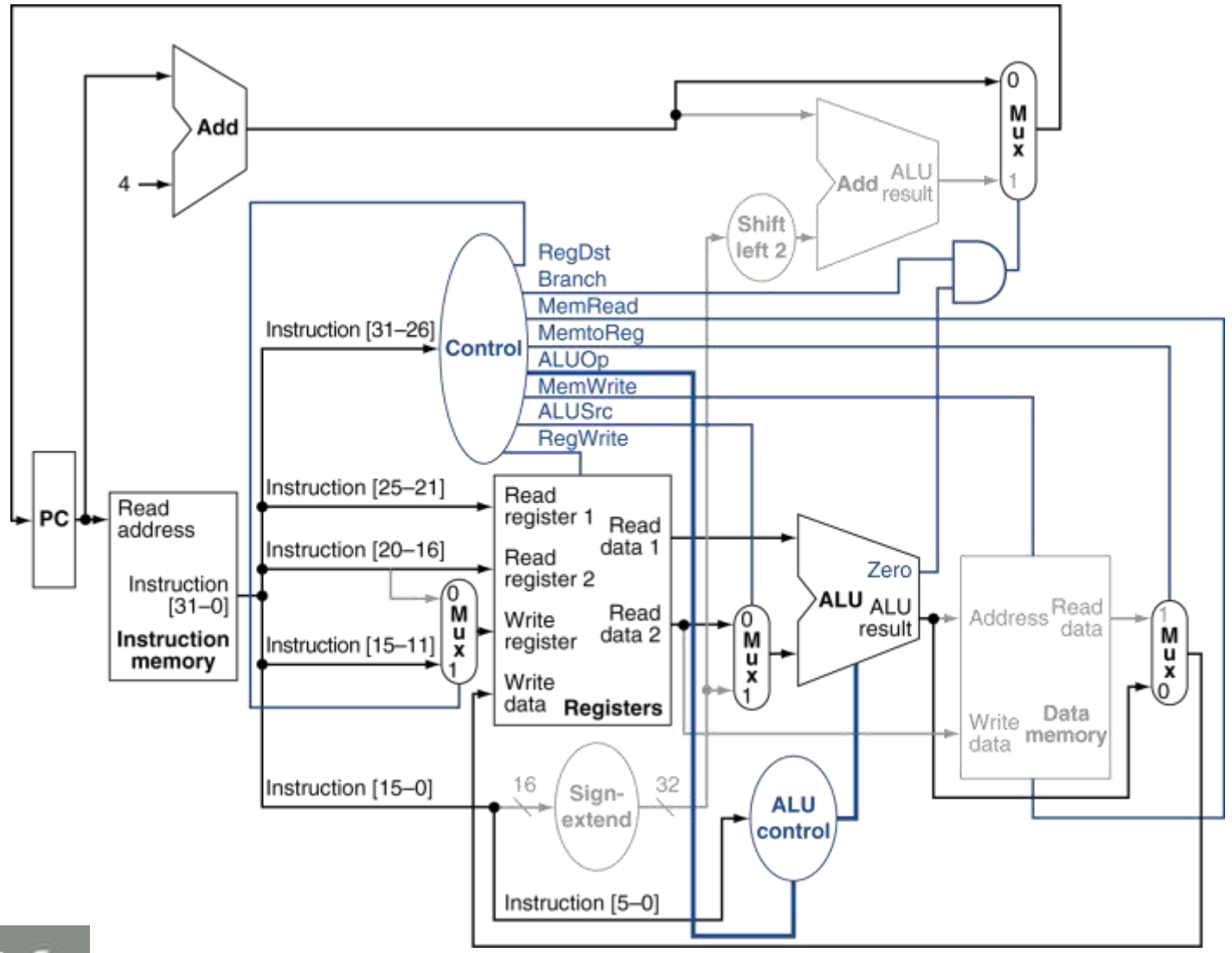
Datapath with Instruction Fields



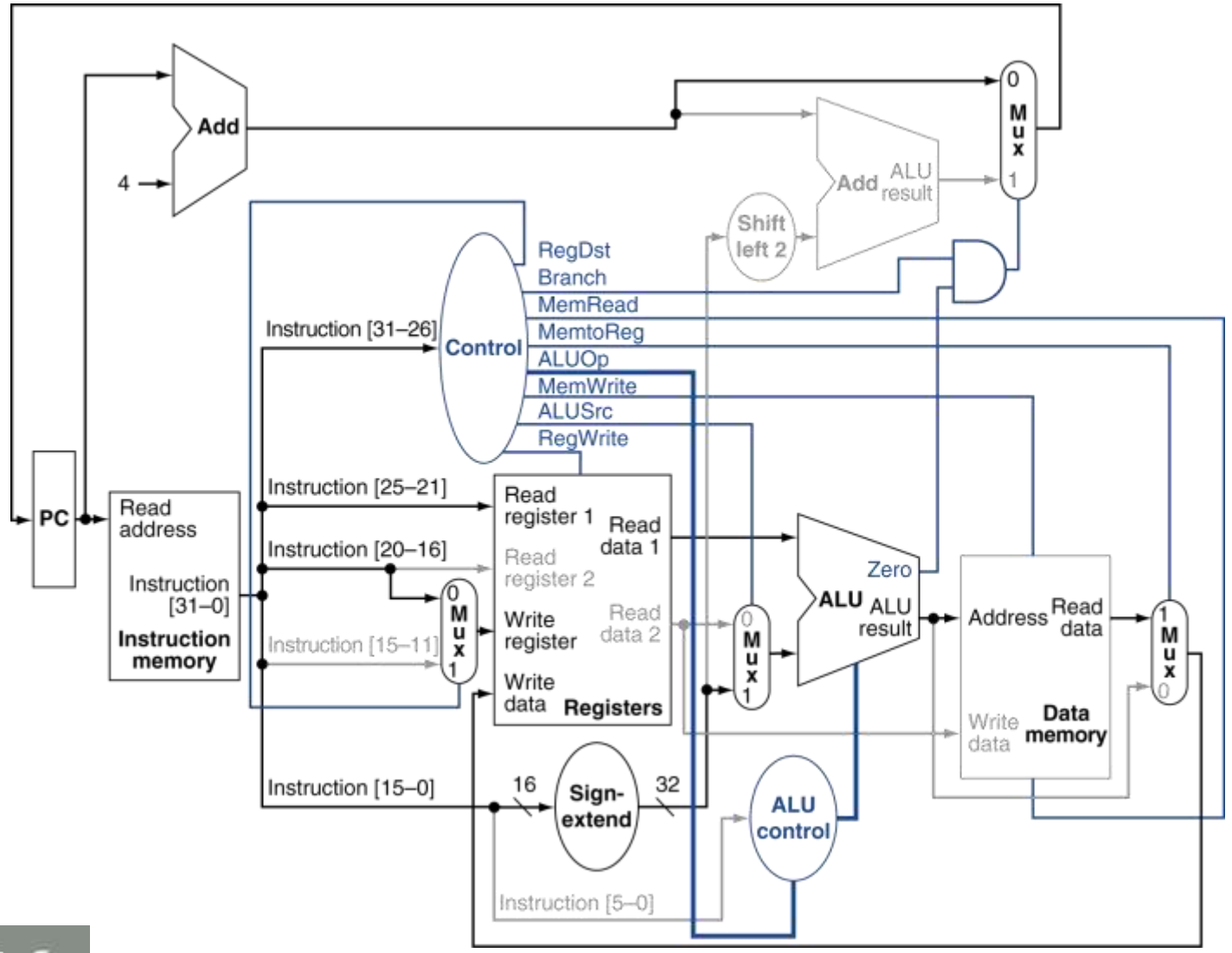
Datapath With Control



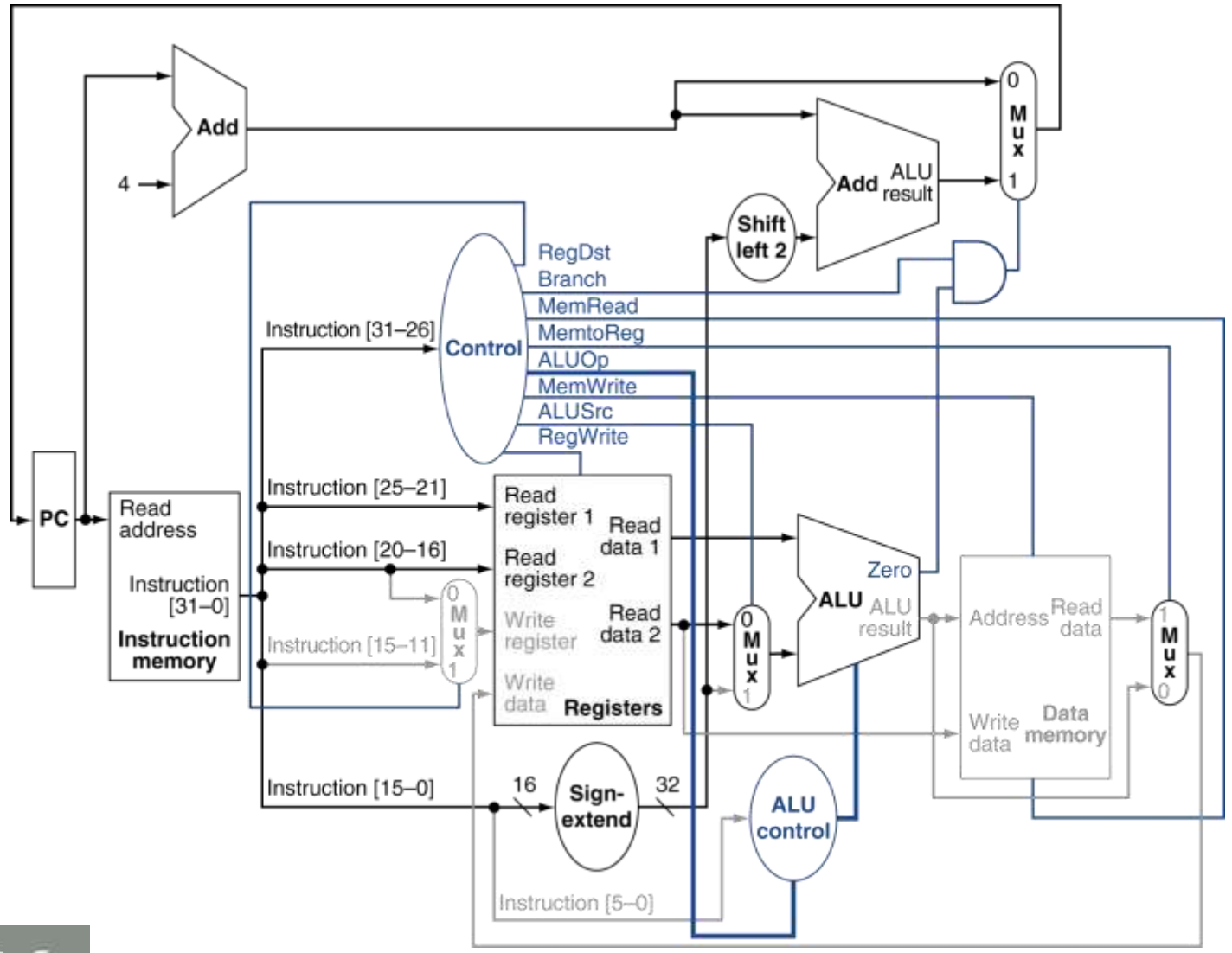
R-Type Instruction



Load Instruction



Branch-on-Equal Instruction



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 design principle
  - Making the common case fast
- We will improve performance by pipelining

Next Topic …

- Pipelining