Chapter 4-1

The Simple Processor

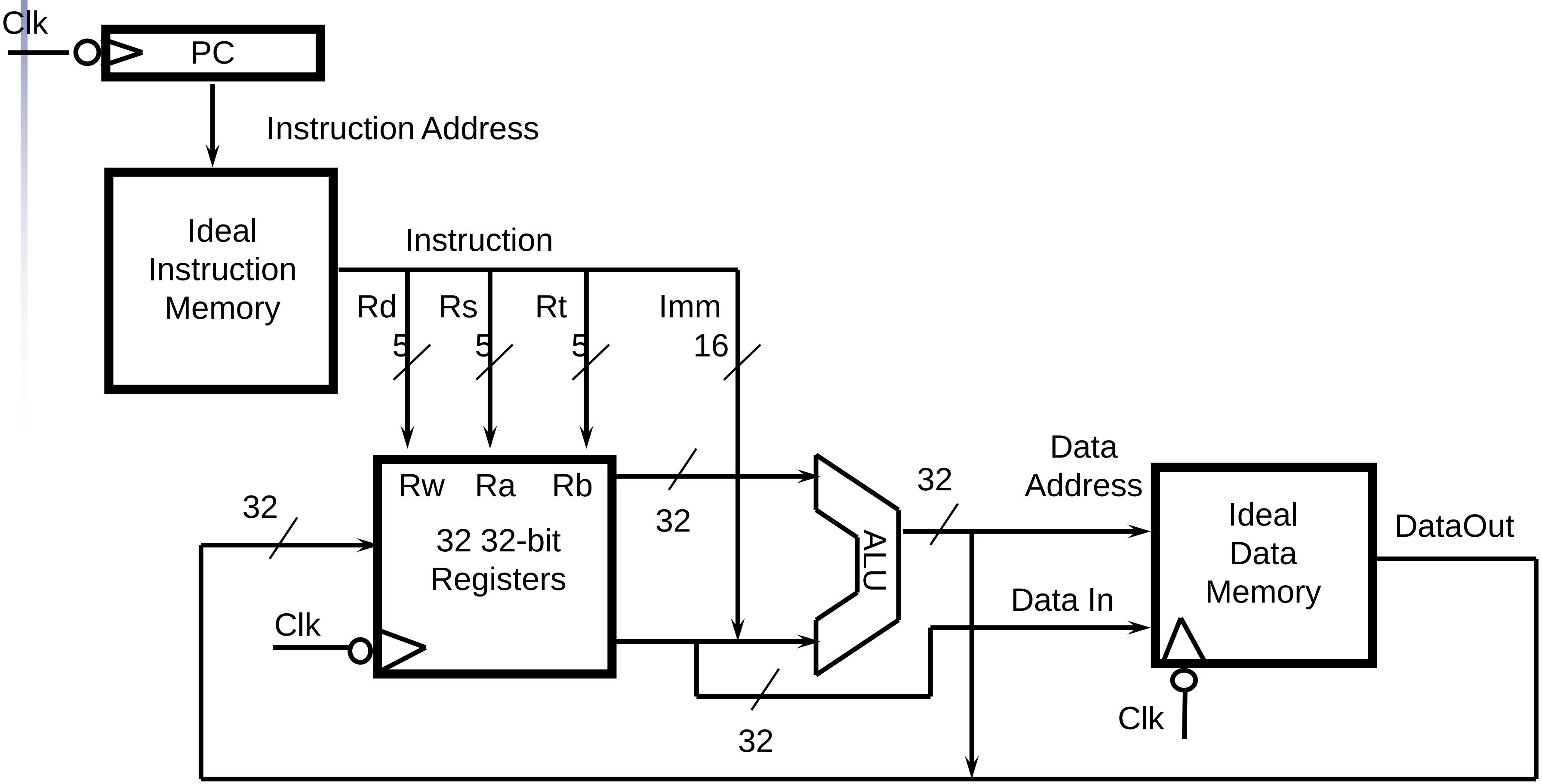
Introduction

- 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`

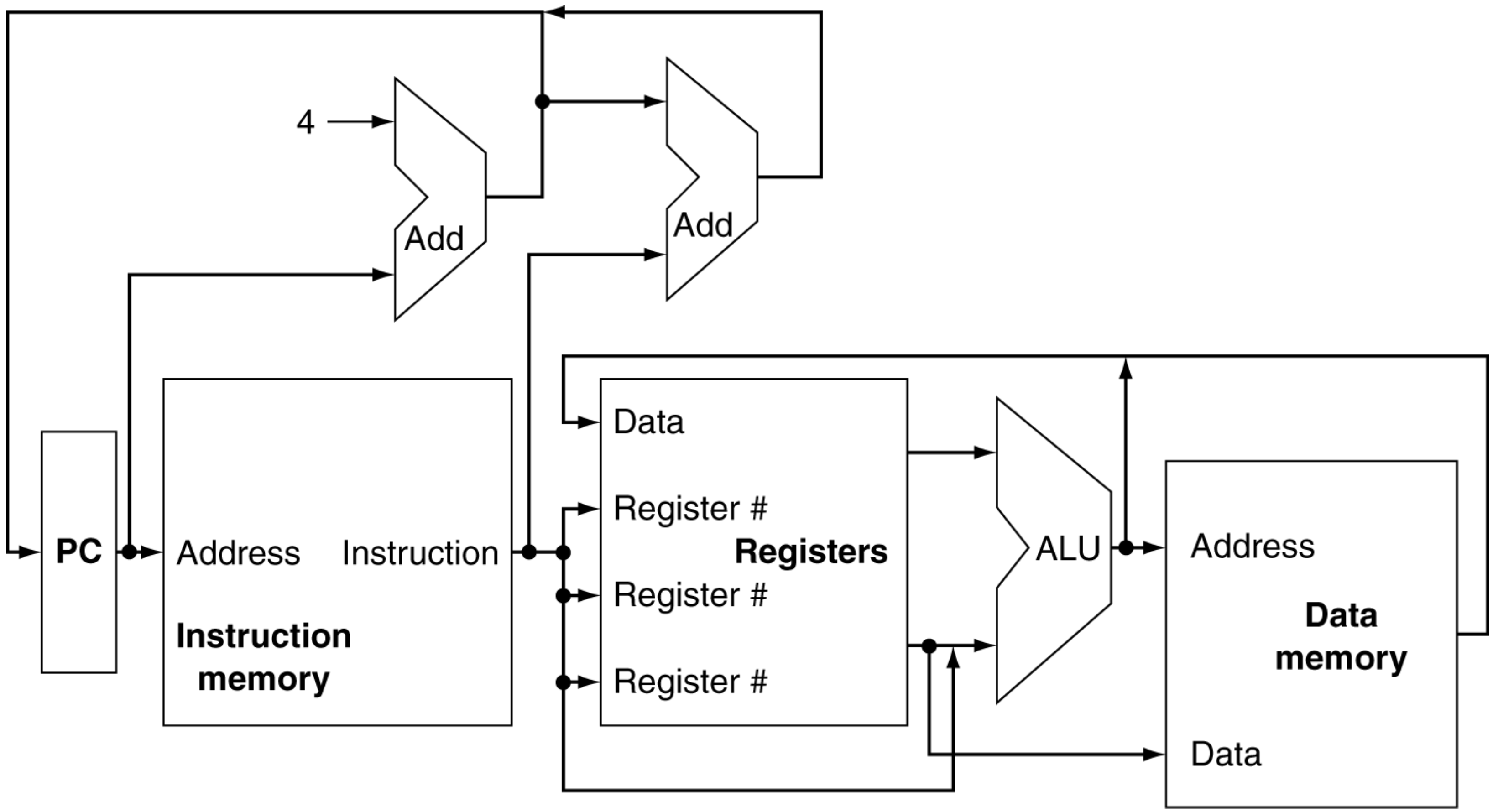
Instruction Execution

- 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

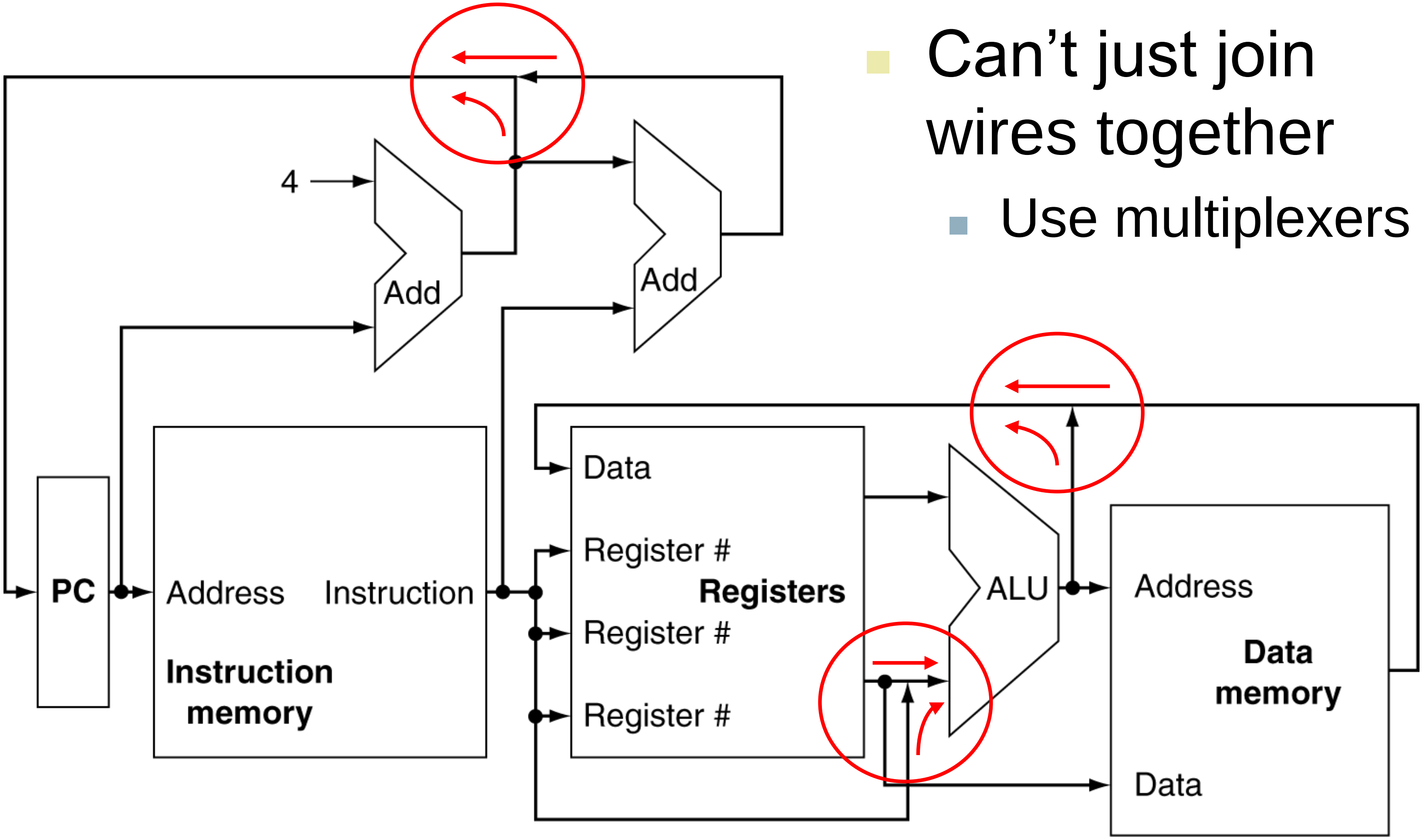
An Abstract View of the Implementation



CPU Overview

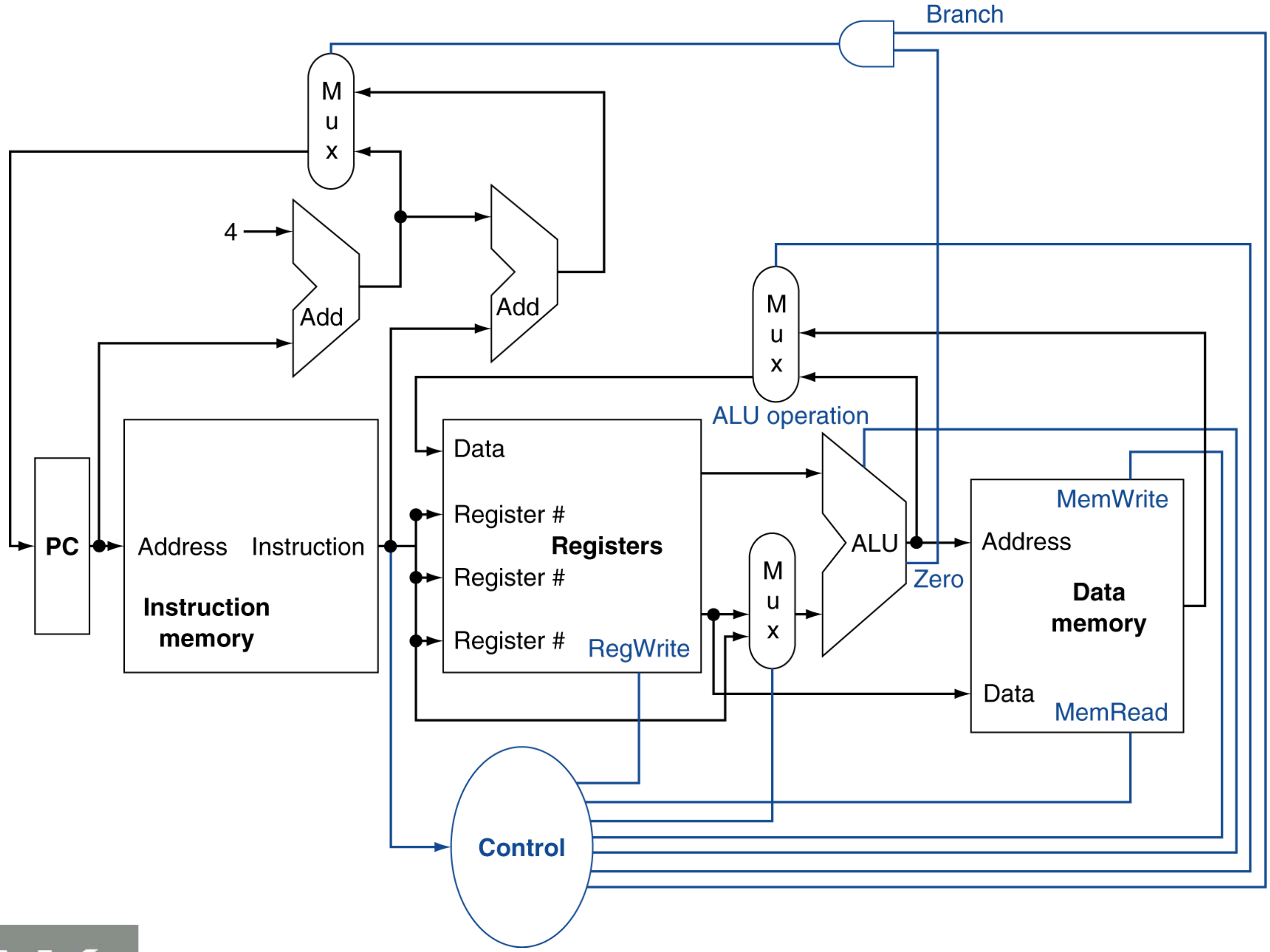


Multiplexers



- Can't just join wires together
  - Use multiplexers

Control

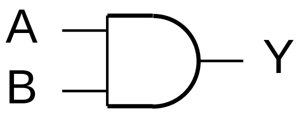

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
Control

Logic Design Basics

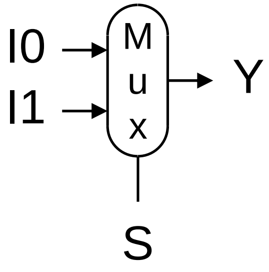
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

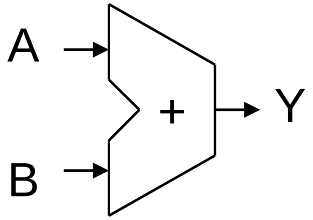
- AND-gate
  - Y = A & B



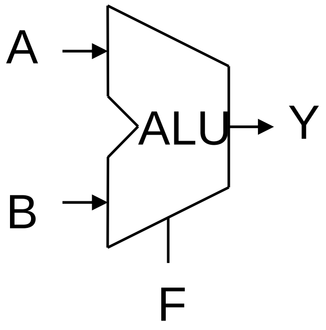
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

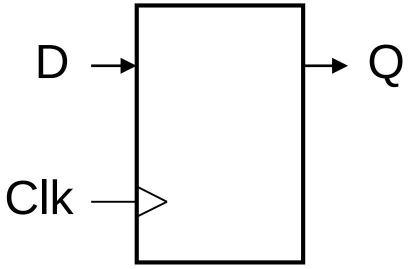


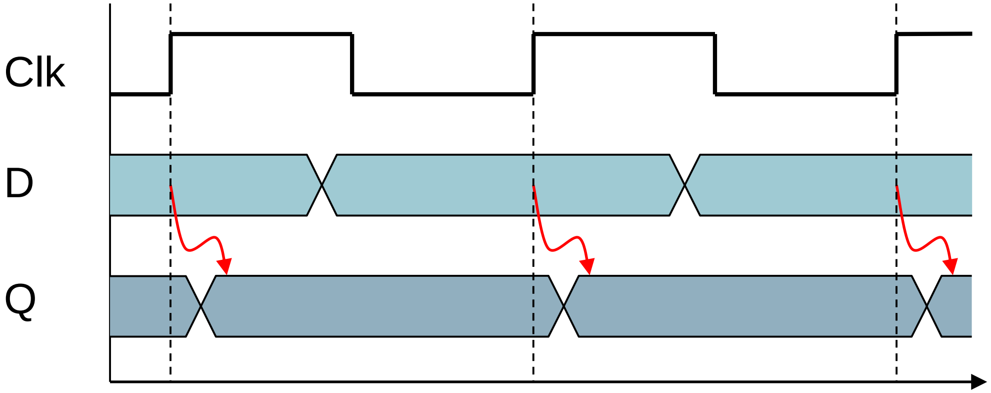
- Arith./Logic Unit (ALU)
  - Y = F(A, B)



Sequential Elements

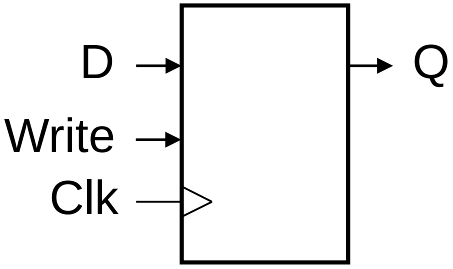
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

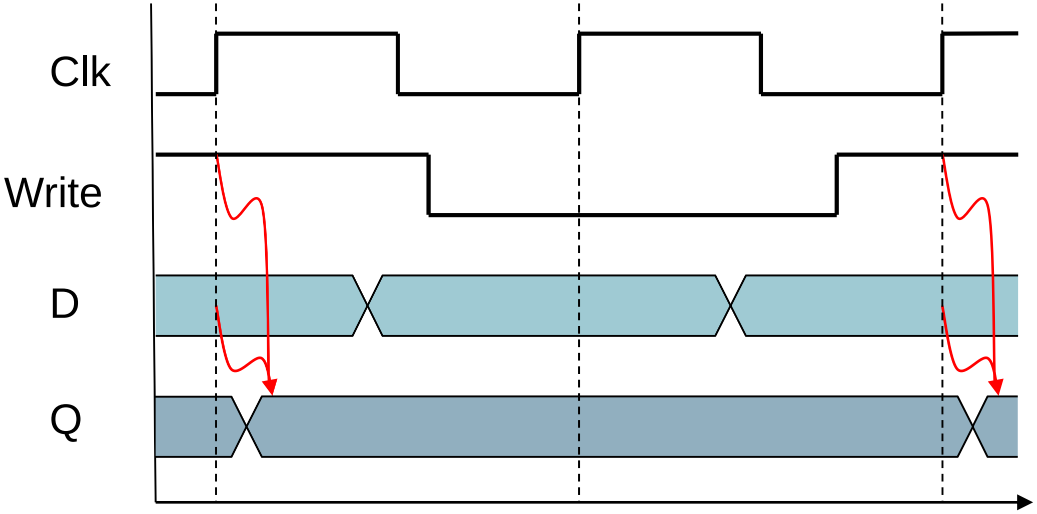




Sequential Elements

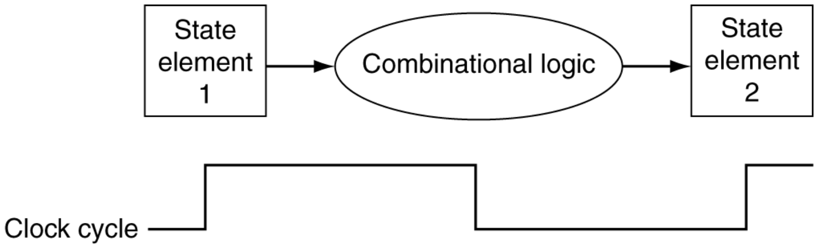
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

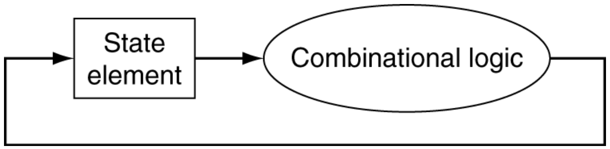




Clocking Methodology

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





Steps of Designing a Processor

- Instruction Set Architecture =>

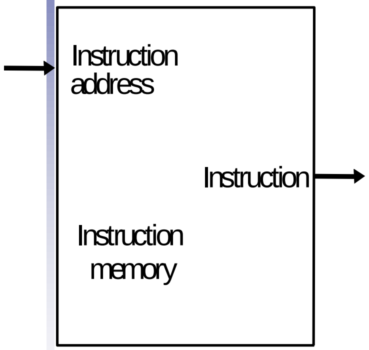
  Register Transfer Language (RTL)
- Register Transfer Language =>
  - Datapath components
  - Datapath interconnect
- Datapath components => Control signals
- Control signals => Control logic

Building a Datapath

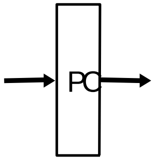
§4.3 Building a Datapath

- 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

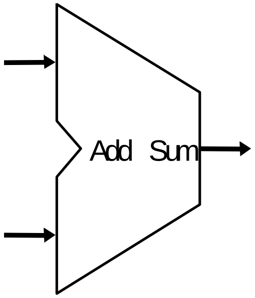
Simple Implementation



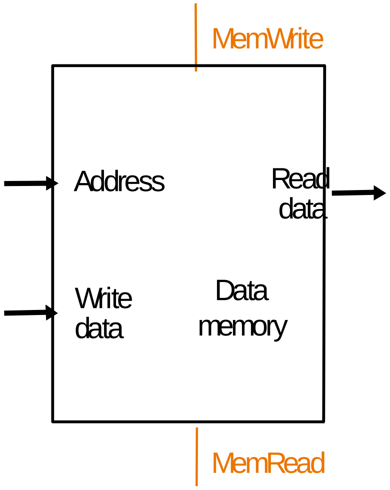
a. Instruction memory



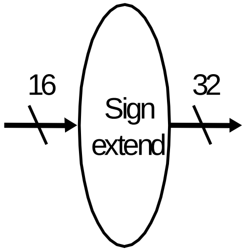
b. Program counter



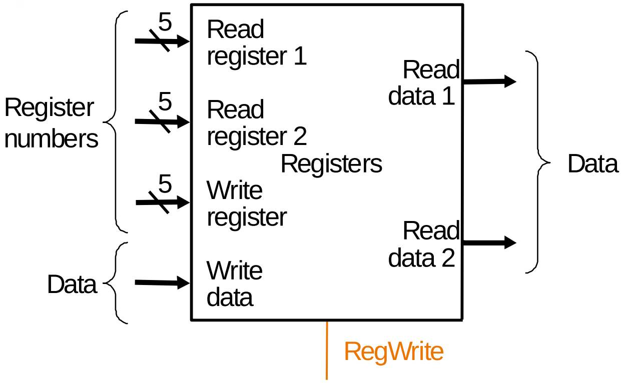
c. Adder



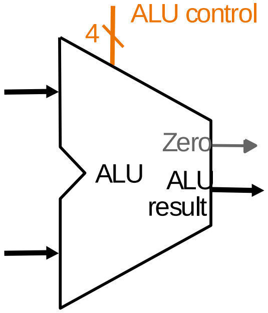
a. Data memory unit



b. Sign-extension unit



a. Registers

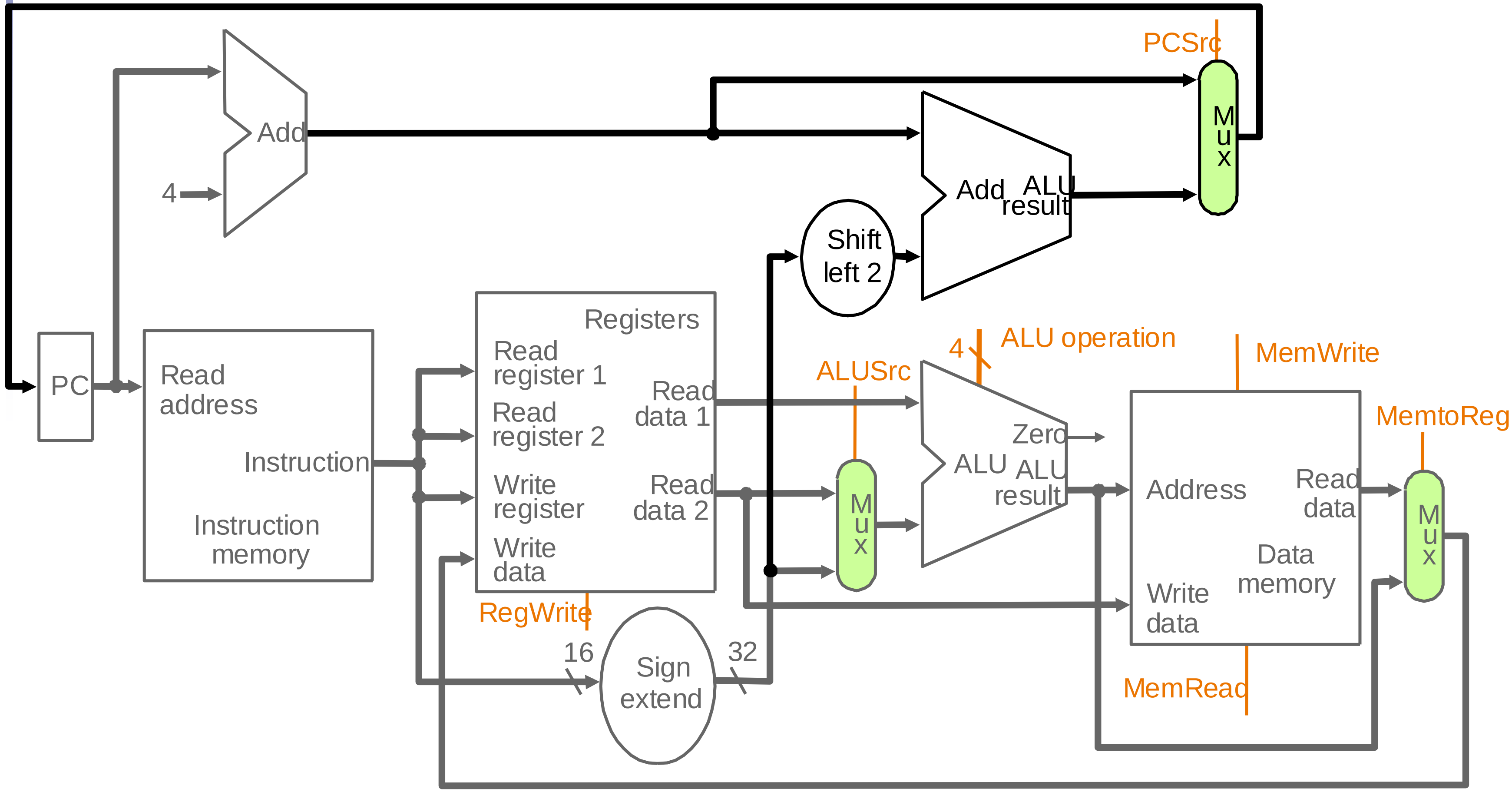


b. ALU

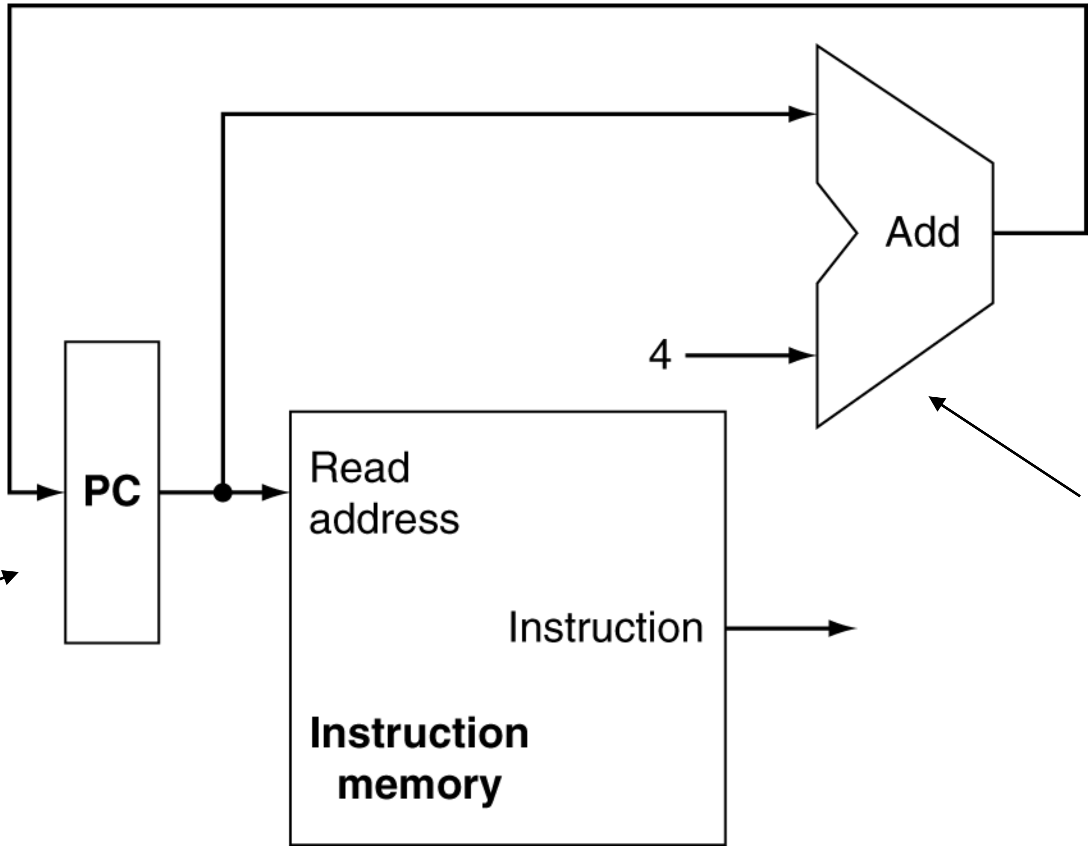
Why do we need this stuff?

Building the Datapath

- Use multiplexors to stitch them together



Instruction Fetch

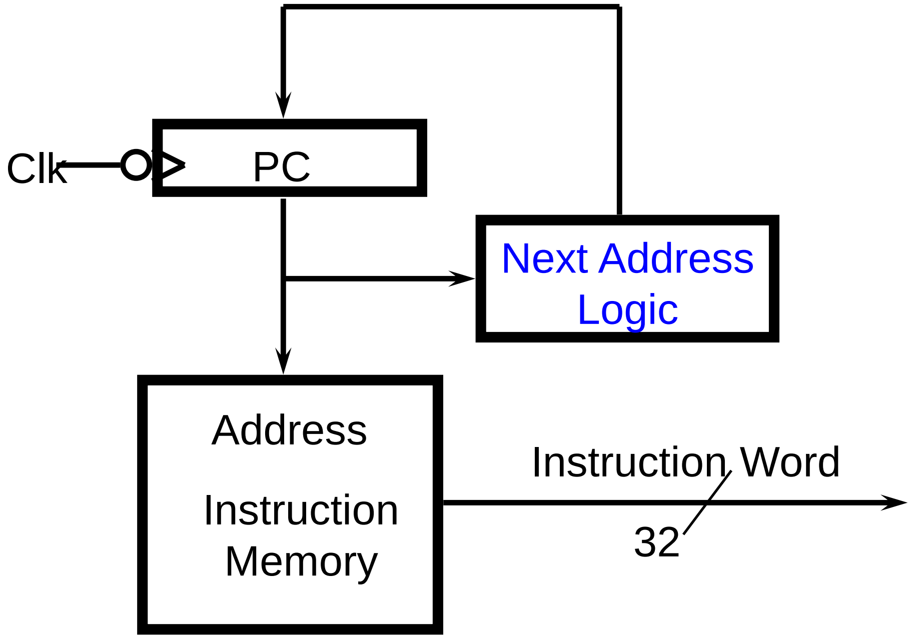


32-bit register

Increment by 4 for next instruction

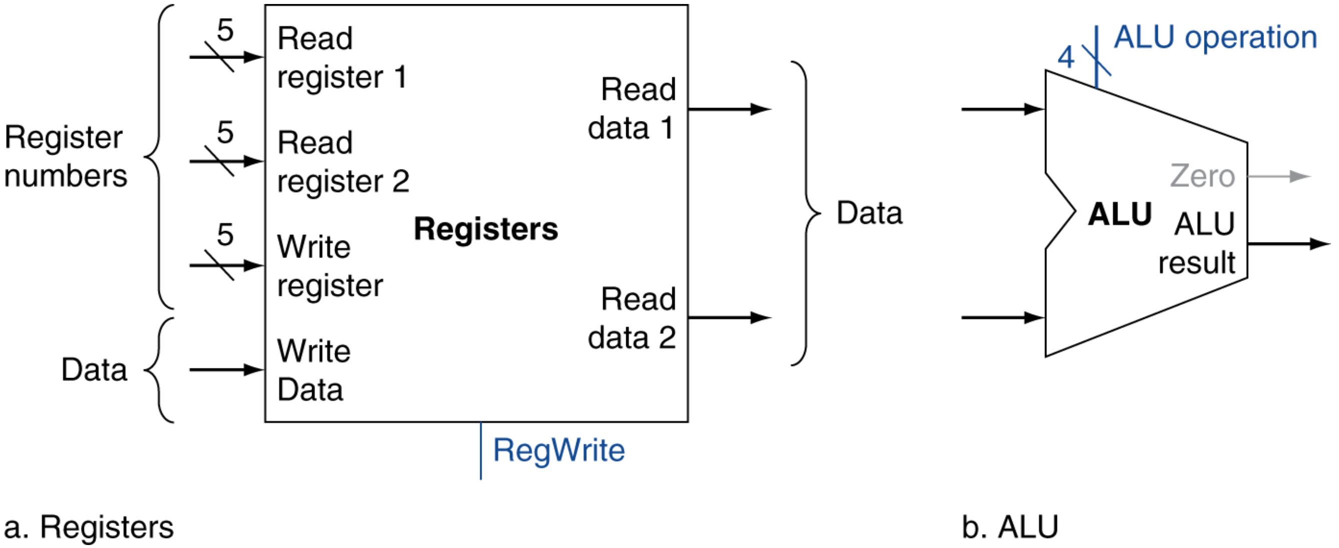
Overview of Instruction Fetch Unit

- **The common RTL operations**
  - **Fetch the Instruction**: mem[PC]
  - **Update the program counter**:
    - Sequential Code: PC <- PC + 4
    - Branch and Jump   PC <- “something else”



R-Format Instructions

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



RTL: ADD (Subtract) Instruction

31 26	21	16	11	6	0
op	rs	**rt**	**rd**	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

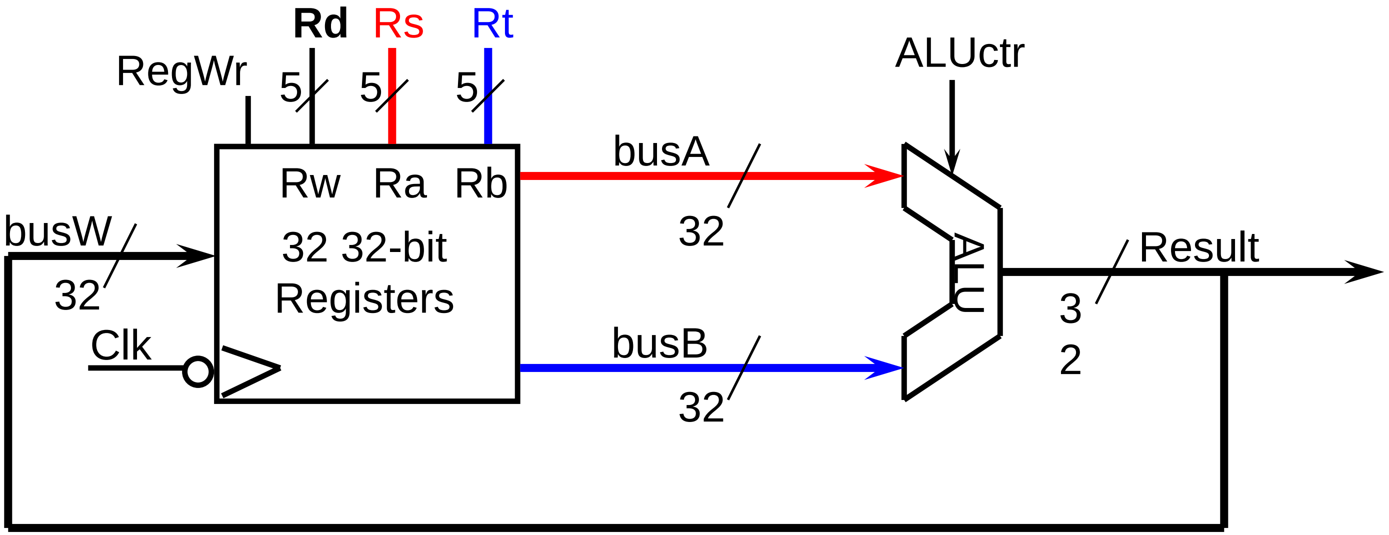
- **add rd, rs, rt**
  - mem[PC] ;Fetch the instruction from memory
  - R[rd] <- R[rs] ± R[rt] ;The actual operation
  - PC <- PC + 4 ;Calculate the next instr’s address

Datapath for Reg-Reg Operations

- **R[rd] <- R[rs] op R[rt]** Example: **add rd, rs, rt**
  - Ra, Rb, and Rw comes from instruction’s rs, rt, and rd fields
  - ALUctr and RegWr: control logic after decoding the instruction

31 26 21 16 11 6 0

op	rs	rt	**rd**	sham	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits



RTL: OR Immediate Instruction

31 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- **ori rt, rs, imm16**
  - mem[PC] ;Fetch the instruction from memory
  - R[rt] <- R[rs] or ZeroExt(imm16)
    ;The OR operation
  - PC <- PC + 4 ;Calculate the next instruction’s address

31 16	15 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	immediate
16 bits	16 bits

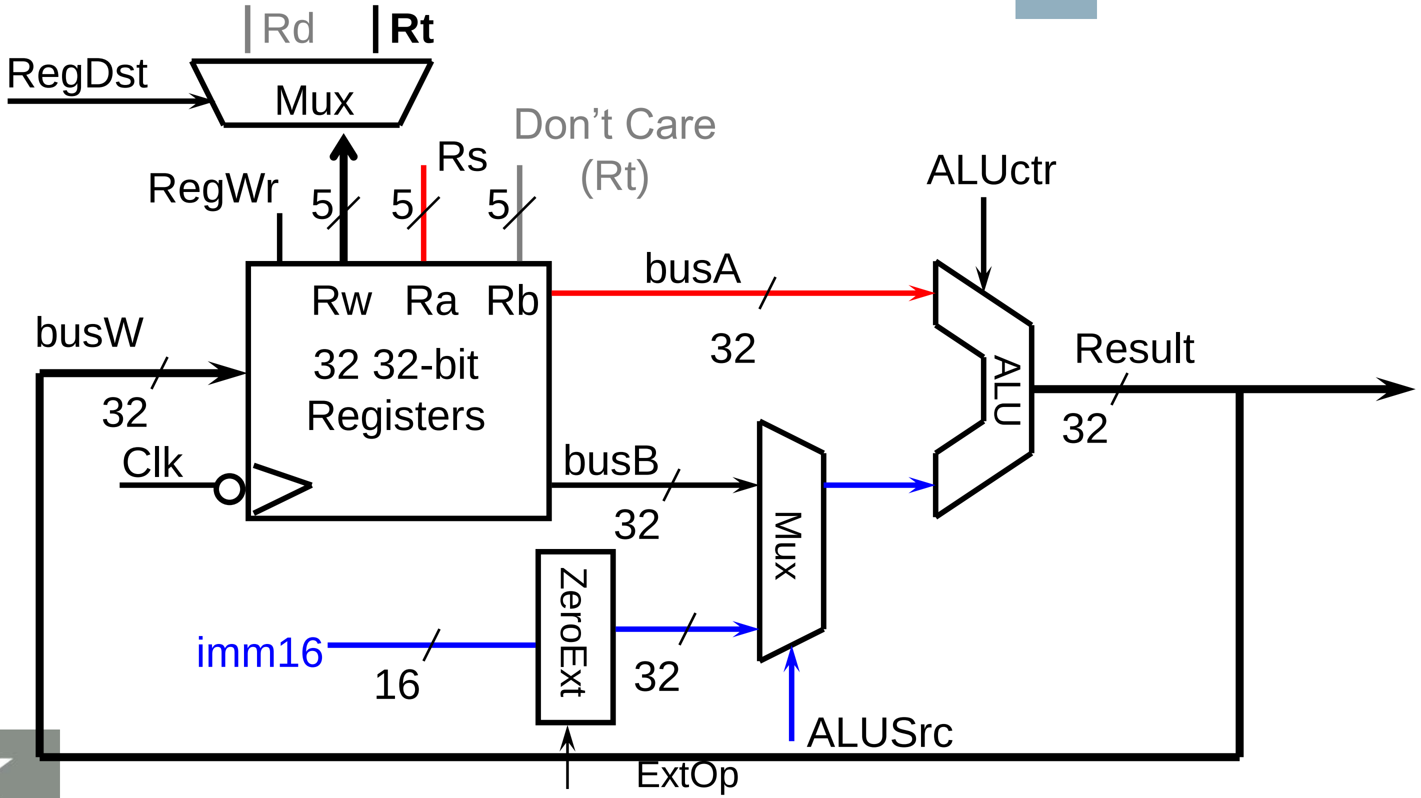
Datapath for Logical Operations with Immediate

- R[rt] <- R[rs] op ZeroExt(imm16)   **Example:** ori rt, rs, imm16

31 26 21 16 11 0

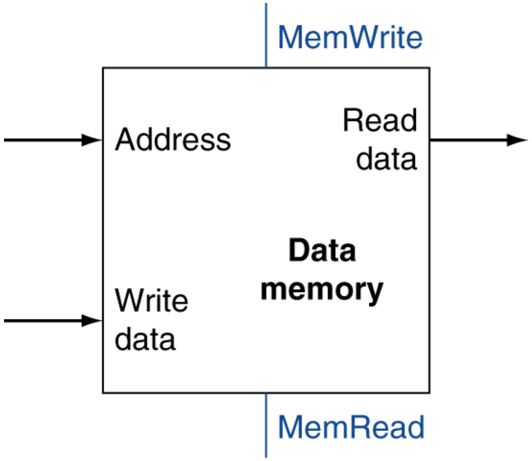
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

rd

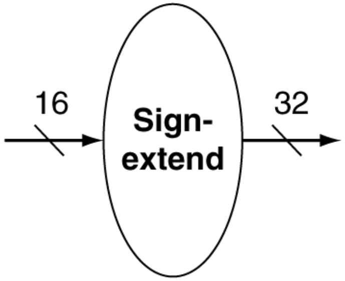


Load/Store Instructions

- 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

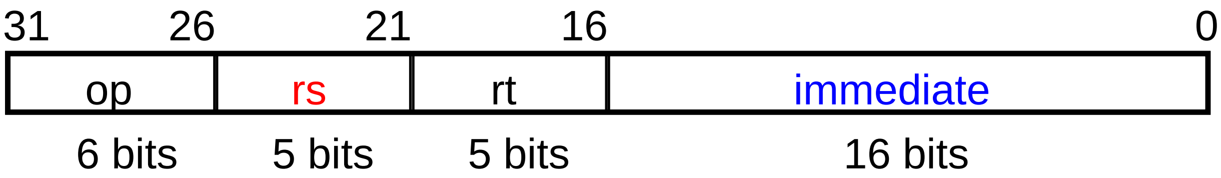


a. Data memory unit

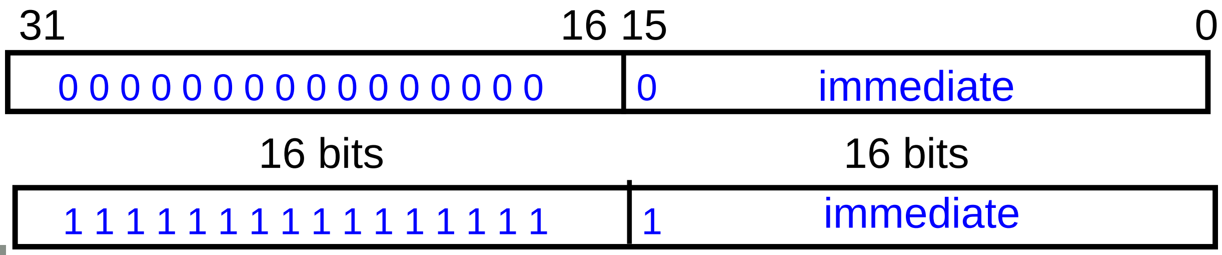


b. Sign extension unit

RTL: Load Instruction



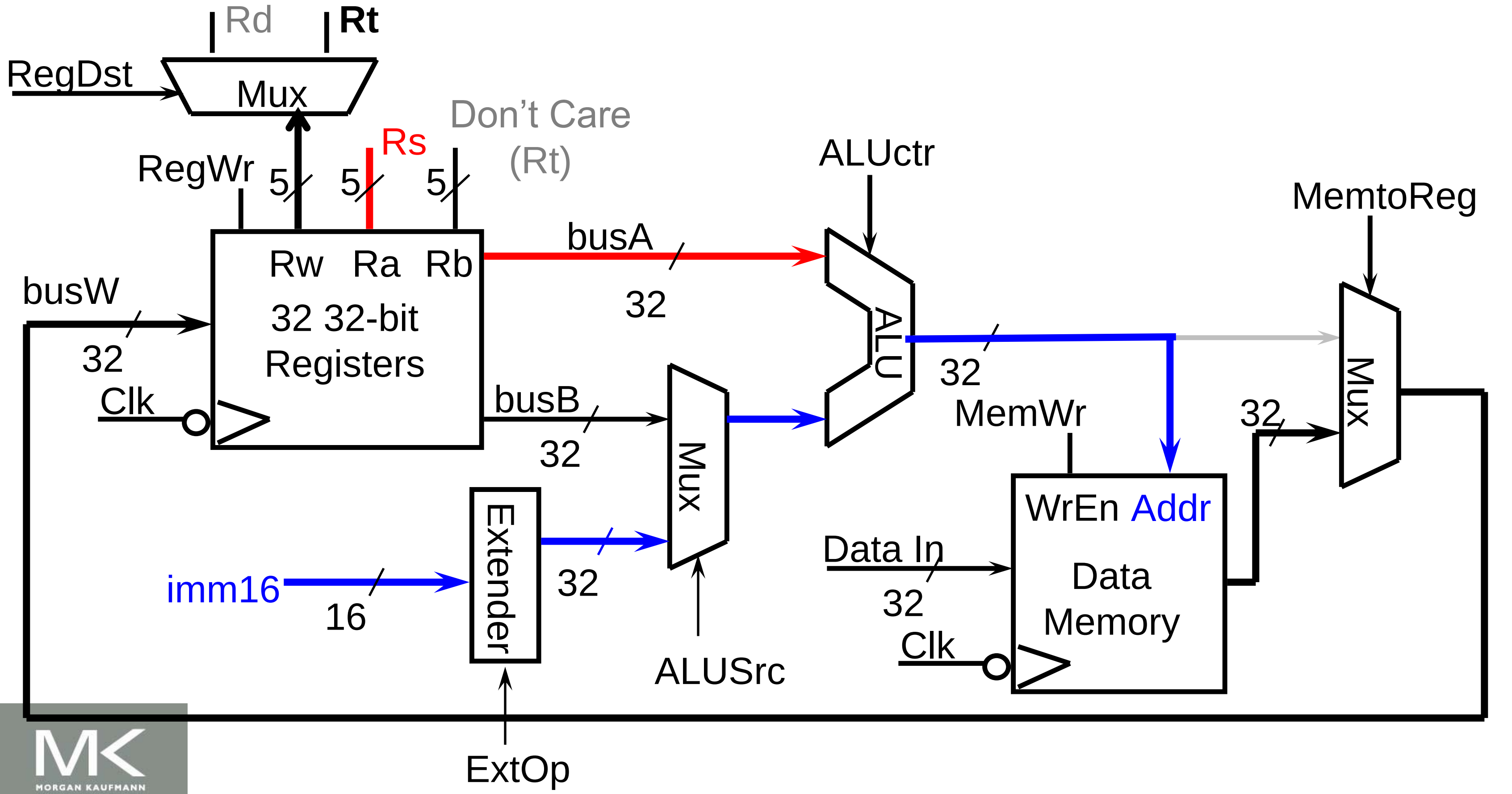
- **lw rt, rs, imm16**
  - mem[PC] ;Fetch the instruction from memory
  - Addr <- R[rs] + SignExt(imm16) ;Calculate the memory address
  - R[rt] <- Mem[Addr] ; Load the data into register
  - PC <- PC + 4 ;Calculate the next instr's address



Datapath for Load Operations

- **R[rt] <- Mem[R[rs] + SignExt(imm16)]** Example: lw rt, rs, imm16

31 26	21	16	11 0
op	rs	rt	immediate
6 bits	5 bits	5 bits	rd 16 bits



RTL: The Store Instruction

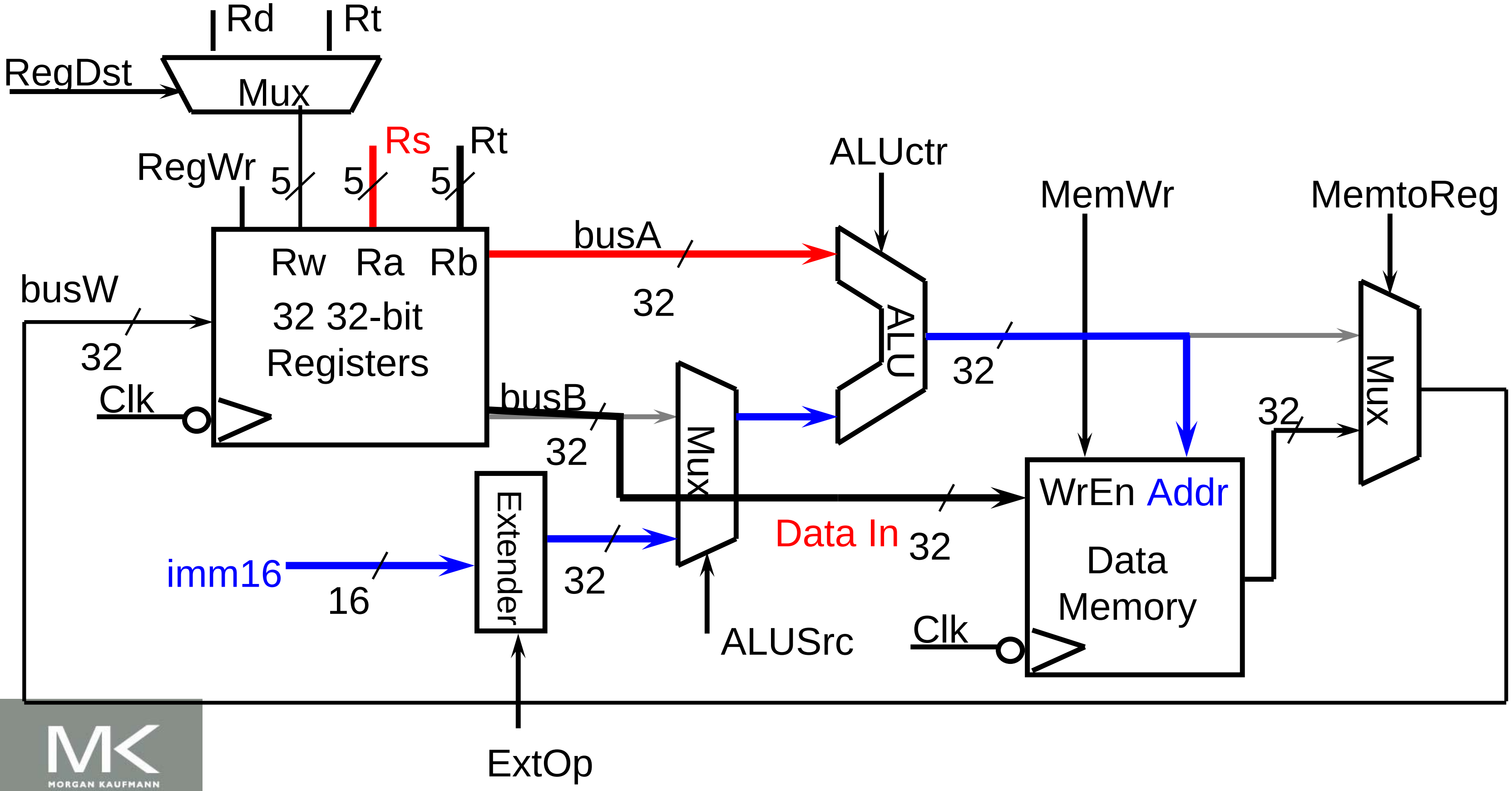
31	26	21	16	0
op	rs	rt	immediate	
6 bits	5 bits	5 bits	16 bits	

- sw rt, rs, imm16
  - mem[PC] ;Fetch the instruction from memory
  - Addr <- R[rs] + SignExt(imm16) ;Calculate the memory address
  - Mem[Addr] <- R[rt] ;Store the register into memory
  - PC <- PC + 4 ;Calculate the next instr’s address

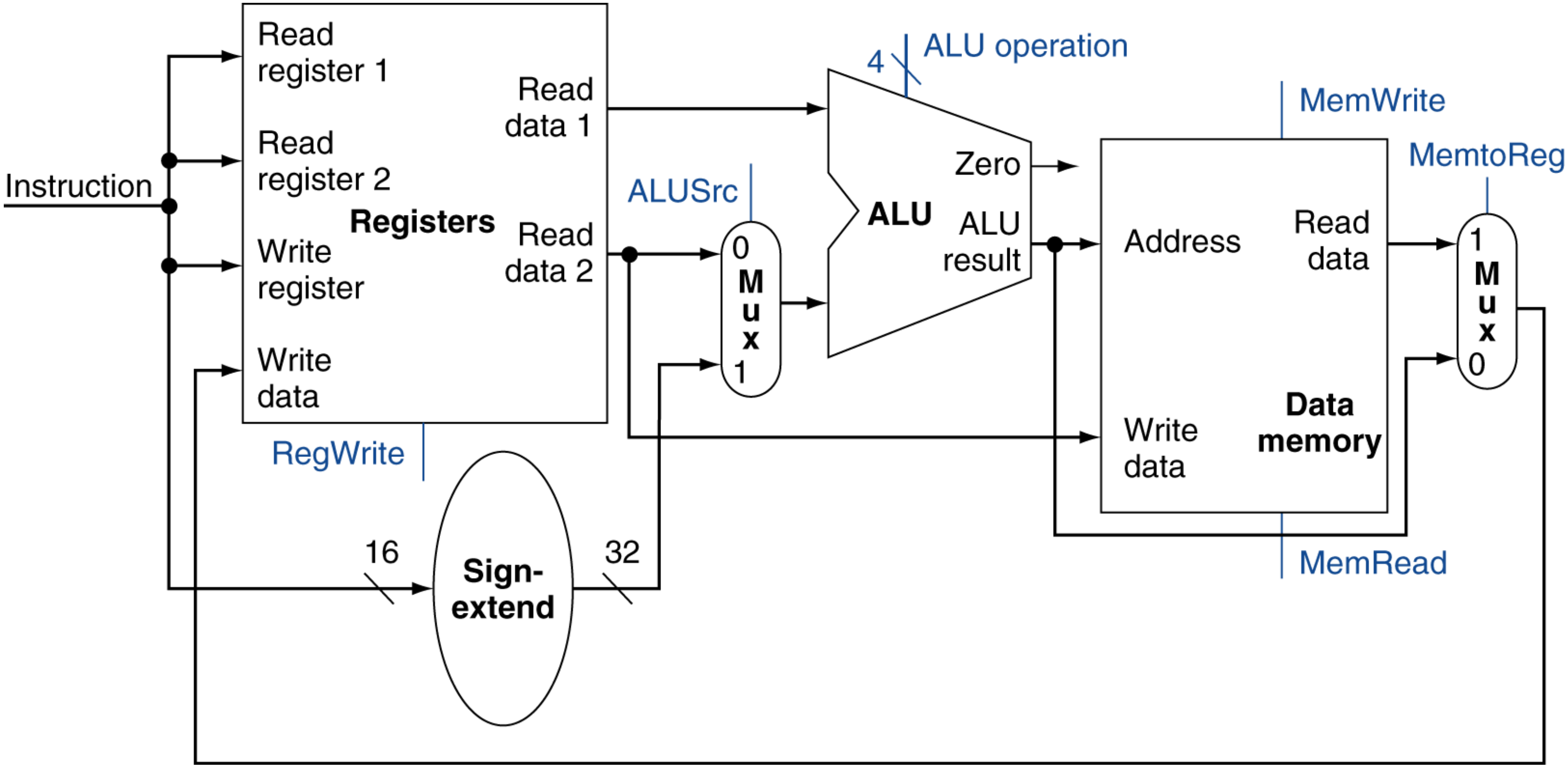
Datapath for Store Operations

- **Mem[R[rs] + SignExt(imm16)] <- R[rt]** Example: sw rt, rs, imm16

31	26	21	16	0
op	rs	**rt**	immediate	
6 bits	5 bits	5 bits	16 bits	



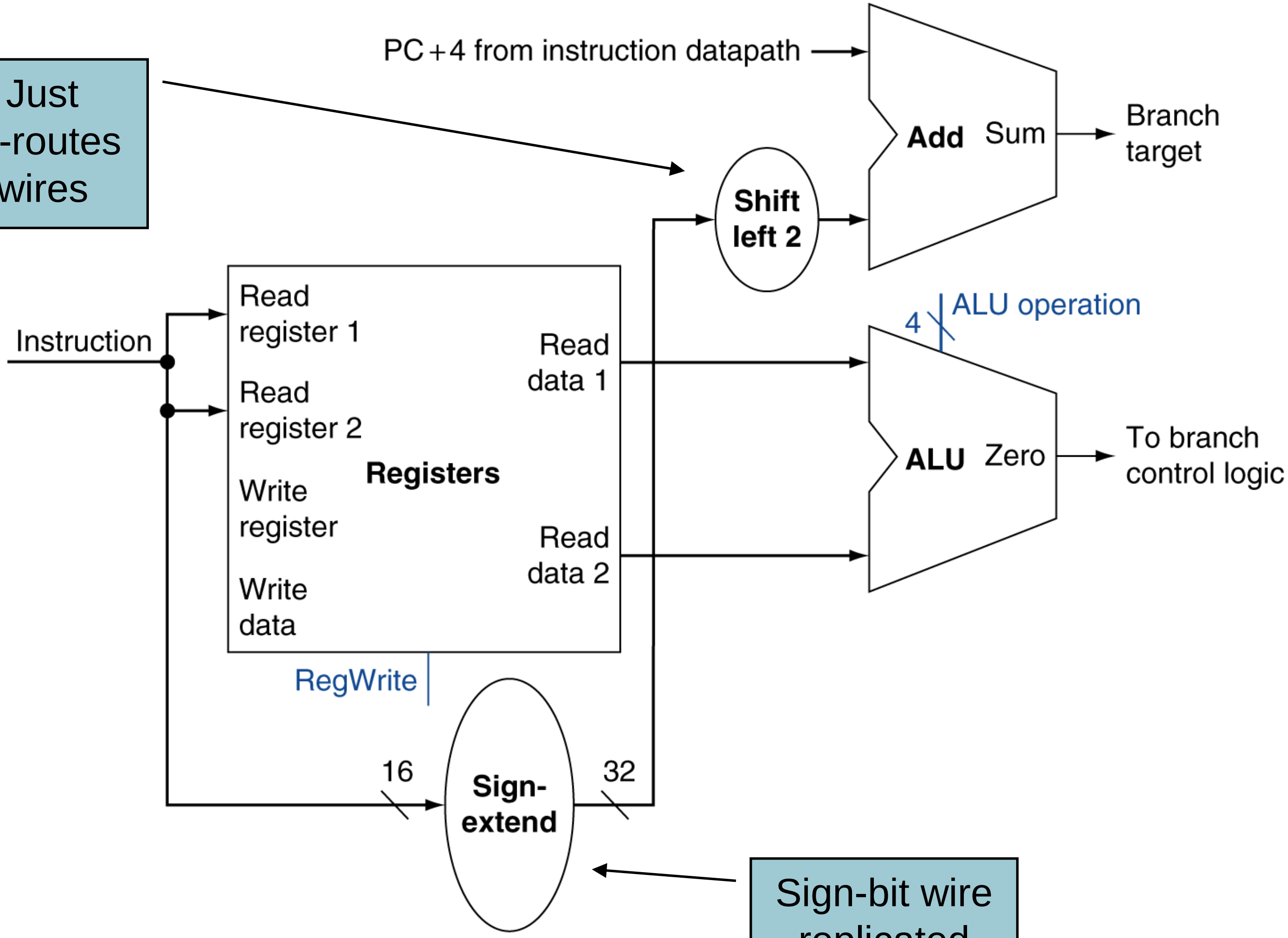
R-Type/Load/Store Datapath



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

Branch Instructions



Just re-routes wires

Sign-bit wire replicated

RTL: The Branch Instruction

31	26	21	16	0
op	rs	rt	immediate	
6 bits	5 bits	5 bits	16 bits	

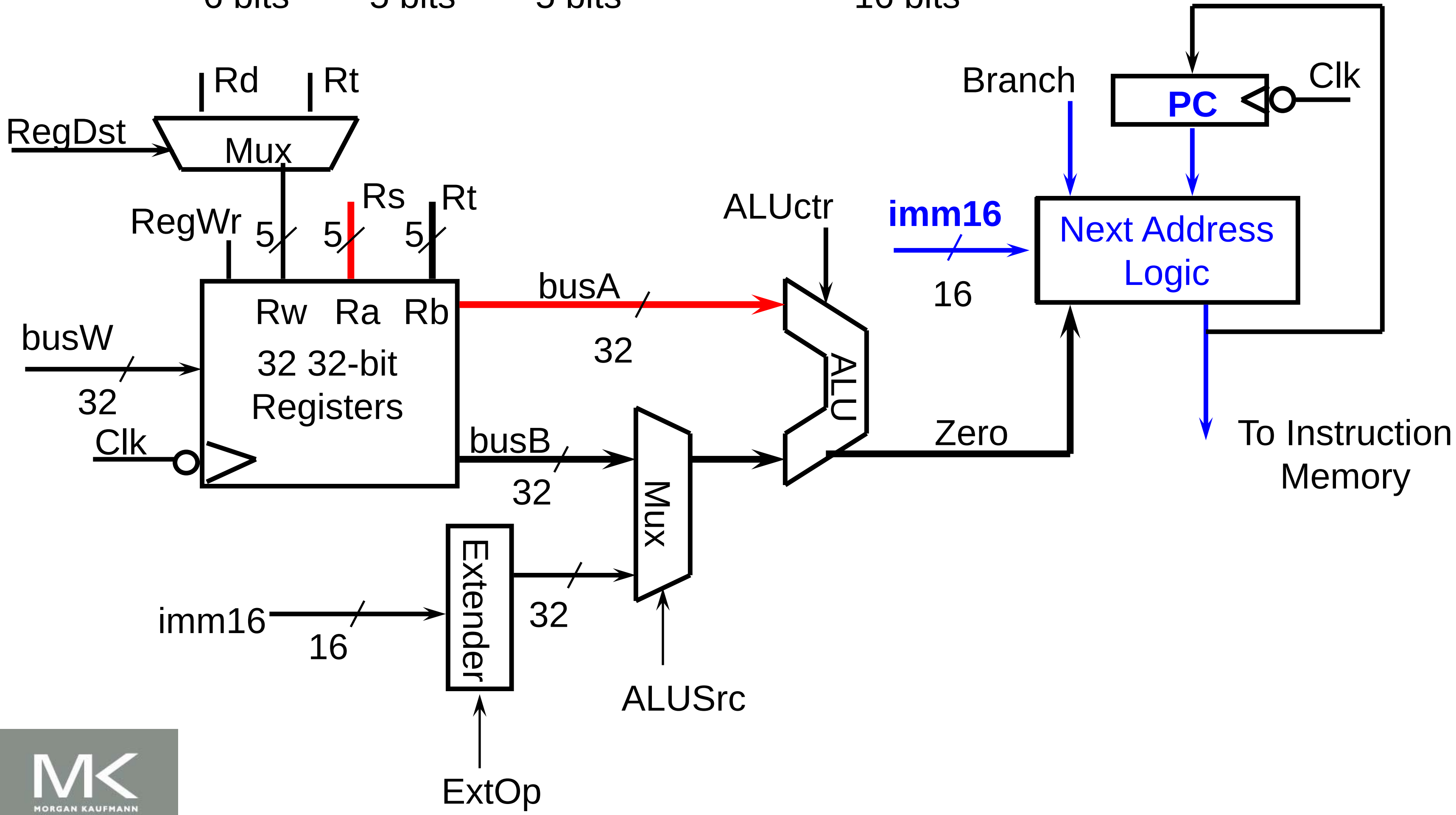
- beq rs, rt, imm16
  - mem[PC] ;Fetch the instruction from memory
  - COND <- R[rs] - R[rt] ;Calculate the branch condition
  - if (COND eq 0) ;Calculate the next instr's address

```
    PC  <-  PC + 4 + ( SignExt(imm16) x 4 )
else
    PC  <-  PC + 4
```

Datapath for Branch Operations

- **beq rs, rt, imm16** We need to compare Rs and Rt!

31 26	21	16	0
op	rs	rt	**immediate**
6 bits	5 bits	5 bits	16 bits

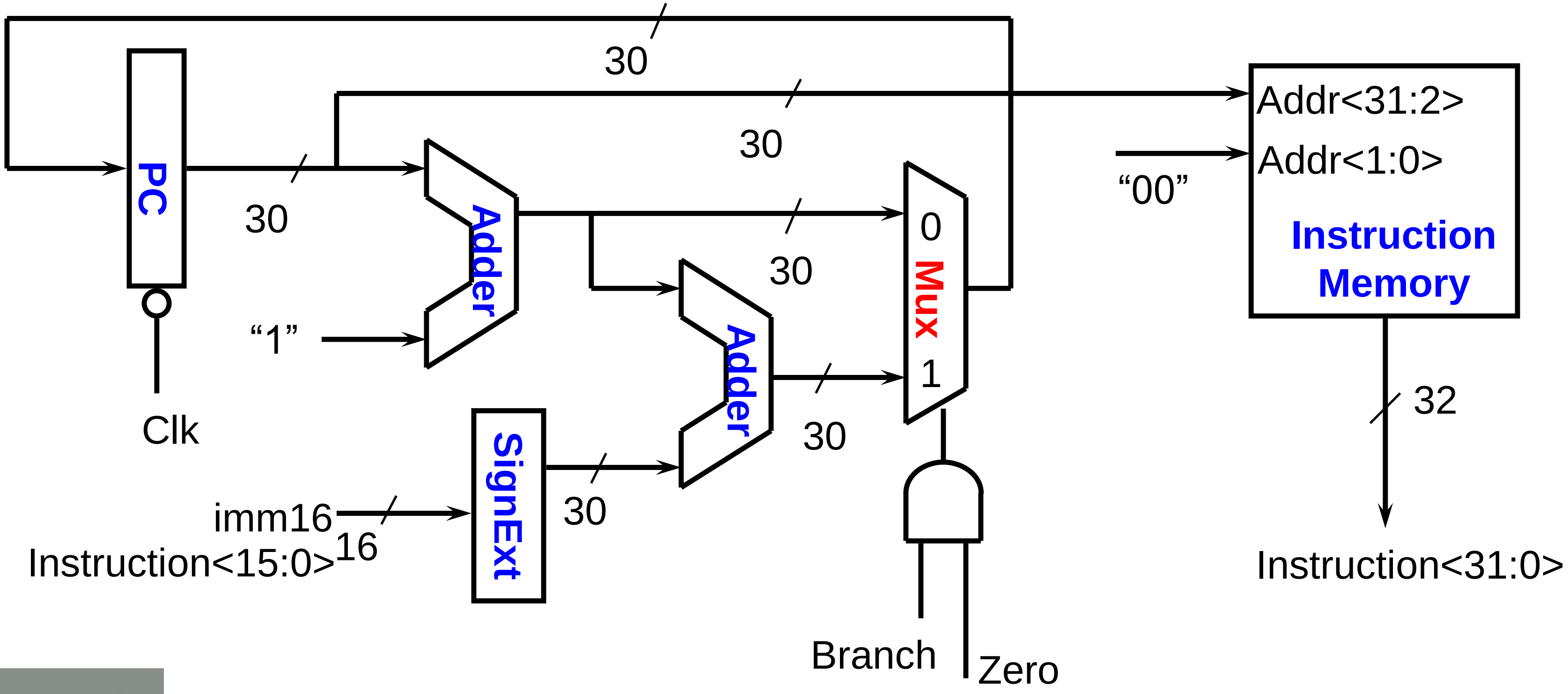


Arithmetic for the Next Address

- In theory, the PC is a 32-bit byte address into the instruction memory:
  - Sequential operation: PC<31:0> = PC<31:0> + 4
  - Branch operation: PC<31:0> = PC<31:0> + 4 + SignExt(imm16) * 4
- The magic number "4" always comes up because:
  - The 32-bit PC is a byte address
  - And all our instructions are 4 bytes (32 bits) long
- In other words:
  - The 2 LSBs of the 32-bit PC are always zeros
  - There is no reason to have hardware to keep the 2 LSBs
- In practice, we can simplify the hardware by using a 30-bit PC<31:2>:
  - Sequential operation: PC<31:2> = PC<31:2> + 1
  - Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt(imm16)
  - In either case: Instruction Memory Address = PC<31:2> concat "00"

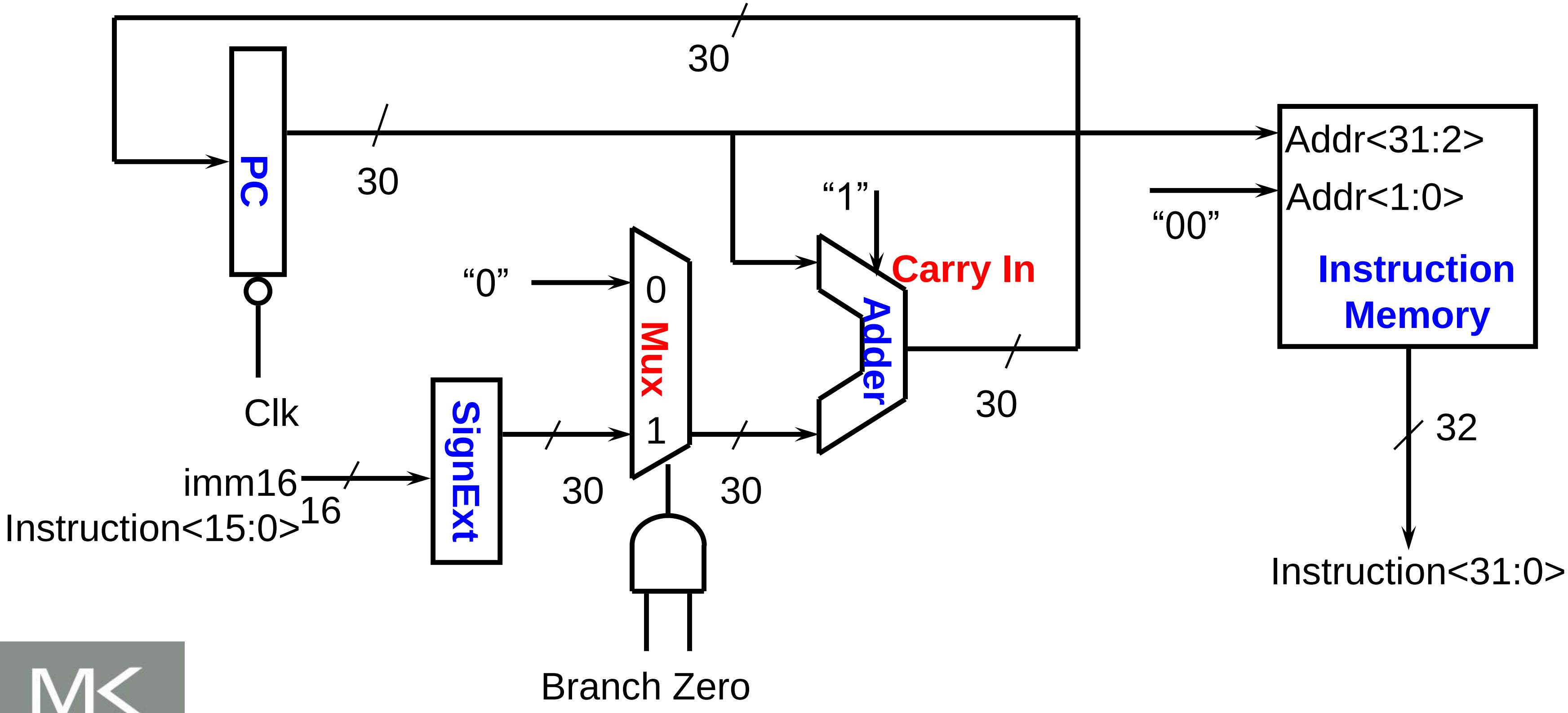
Next Address Logic: Expensive & Fast Solution

- Using a 30-bit PC:
  - Sequential operation: PC<31:2> = PC<31:2> + 1
  - Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt(imm16)
  - In either case: Instruction Memory Address = PC<31:2> concat “00”

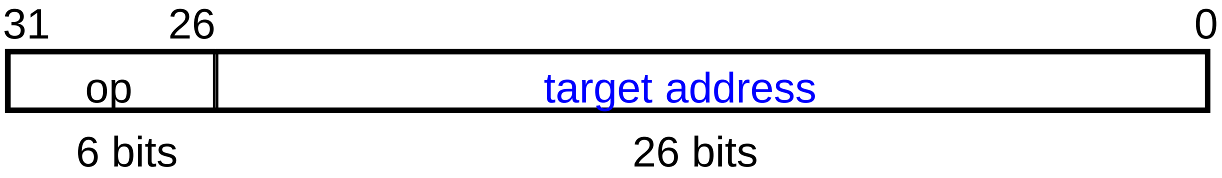


Next Address Logic: Cheap and Slow Solution

- Why is this slow?
  - Cannot start the address add until Zero (output of ALU) is valid
- Does it matter, slow in the overall scheme of things?
  - Probably not here. Critical path is the load operation.



RTL: The Jump Instruction



31 26 0

op	target address
6 bits	26 bits

- j target
  - mem[PC] ;Fetch the instruction from memory
  - PC<31:2> <- PC<31:28> concat target<25:0>

    ;Calculate the next instr’s address

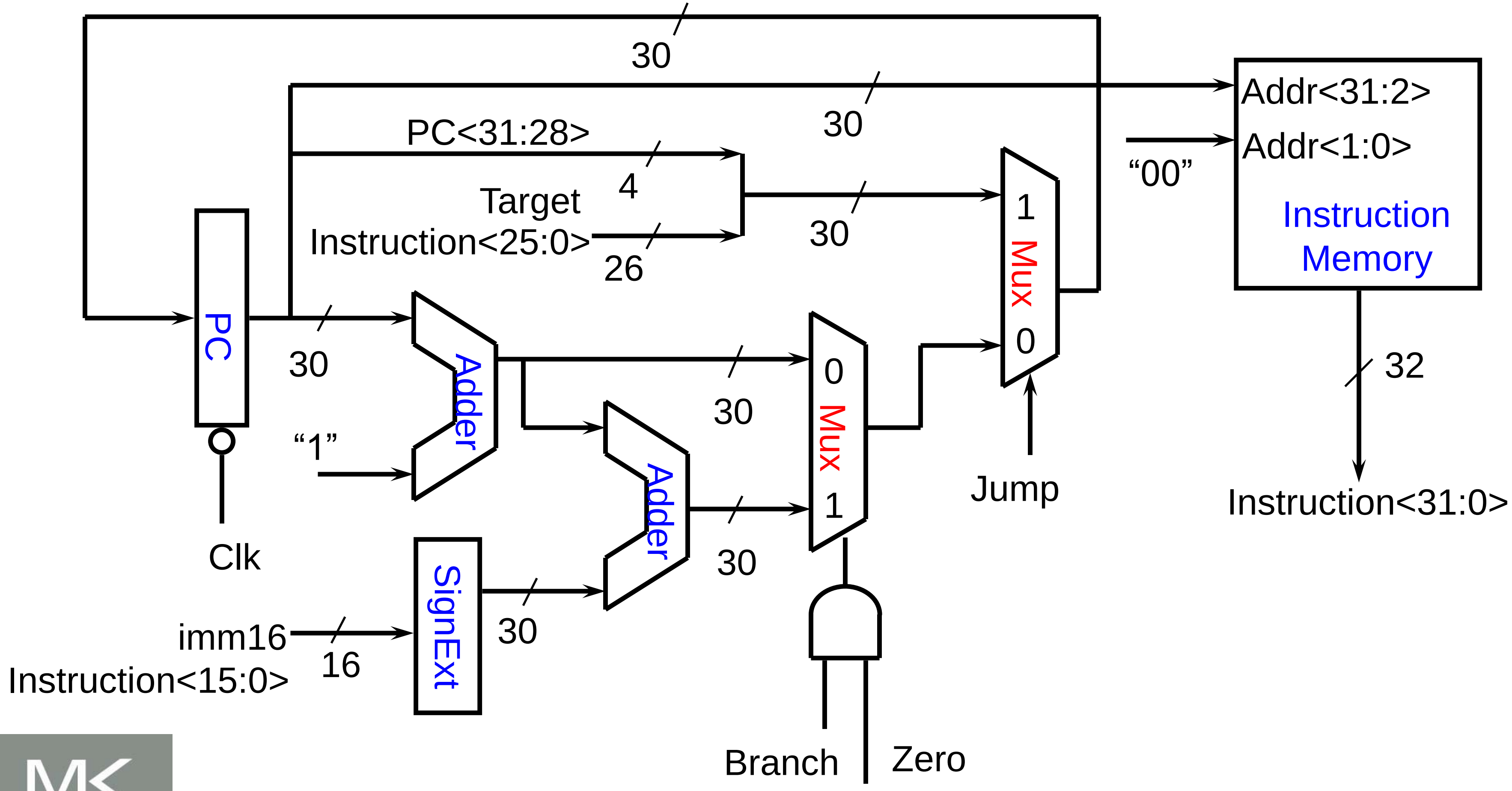
Implementing Jumps

Jump	2	address
	31:26	25:0

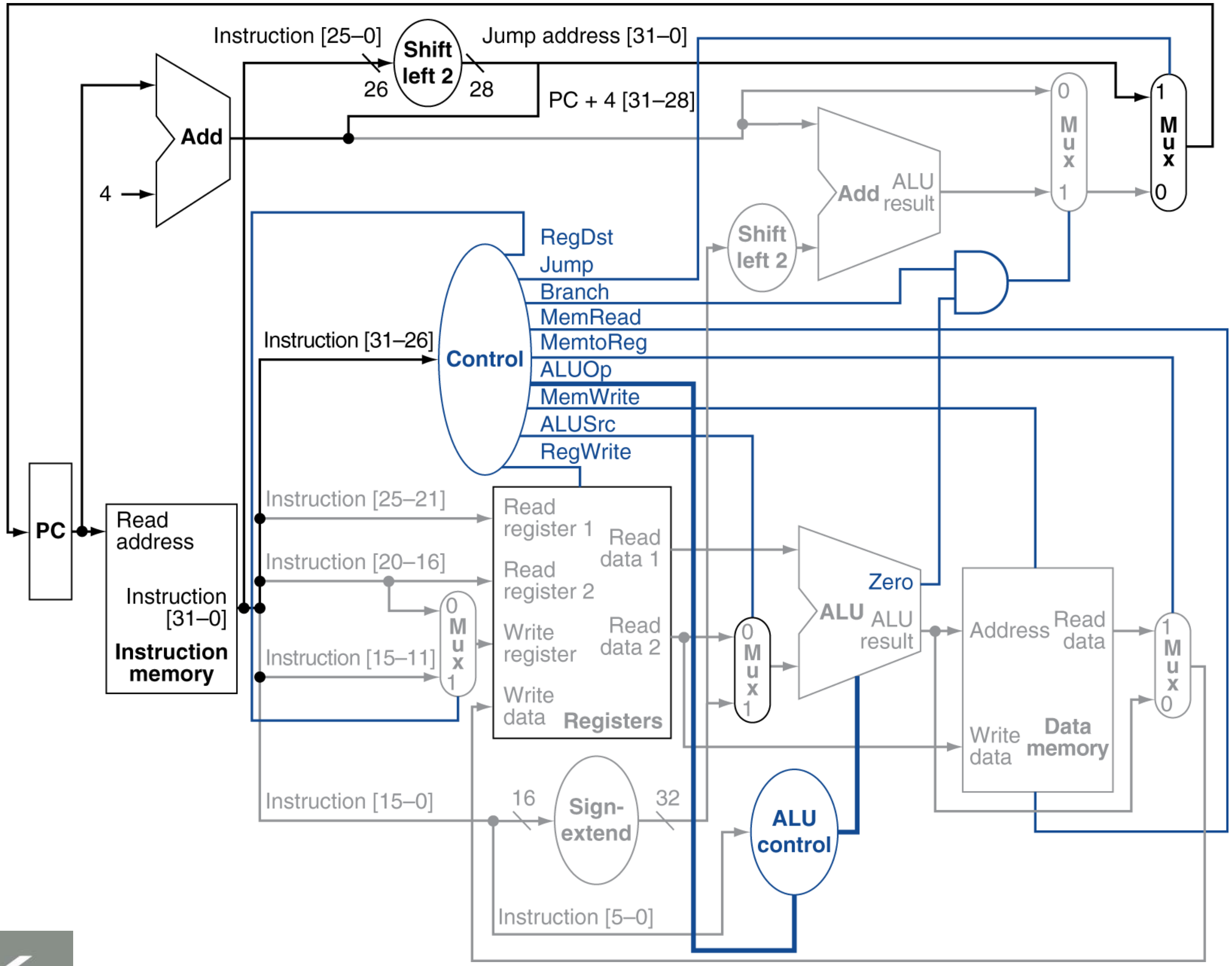
- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

Instruction Fetch Unit

- **j target**
  - PC<31:2> <- PC<31:28> concat target<25:0>

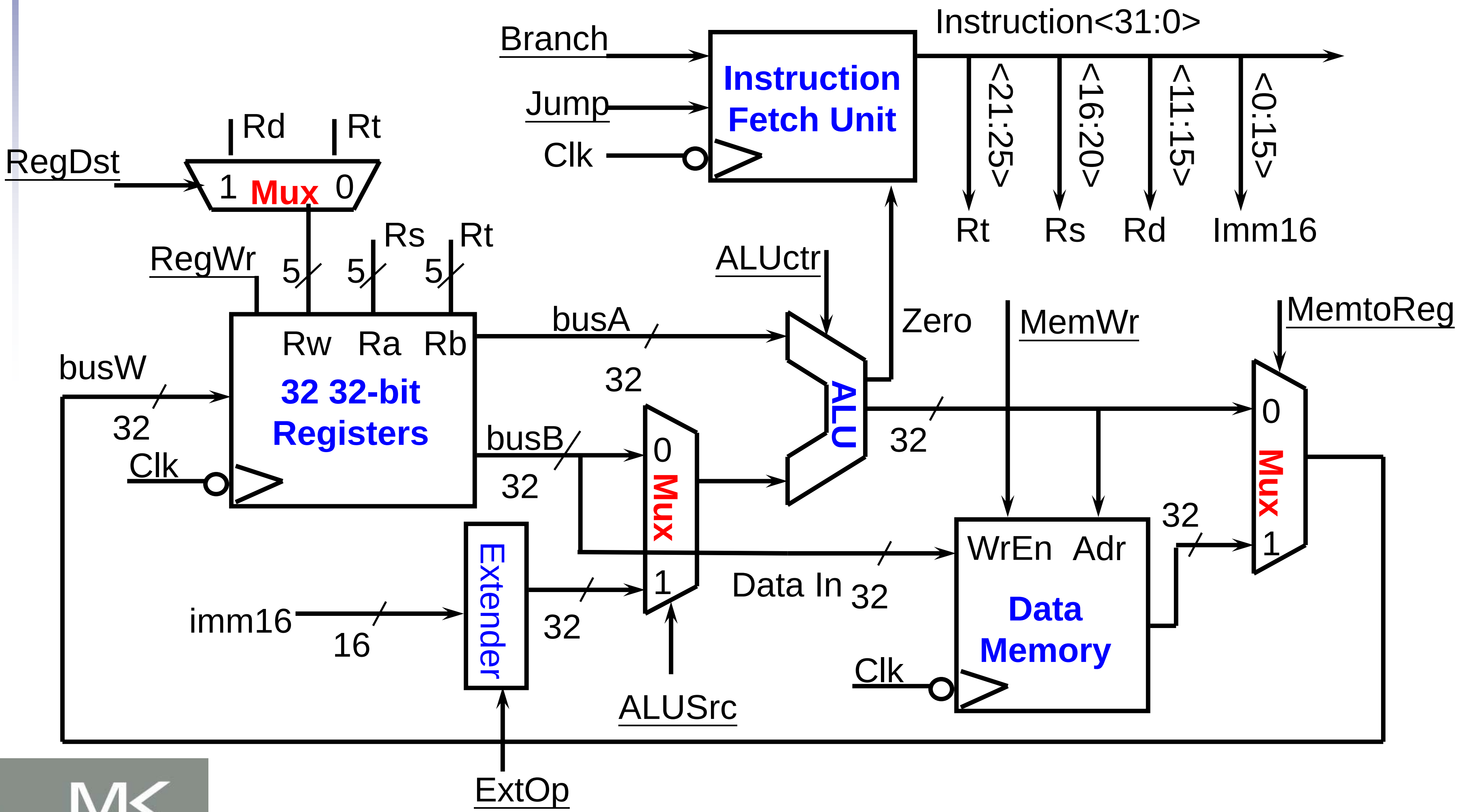


Datapath With Jumps Added



Putting it All Together: A Single Cycle Datapath

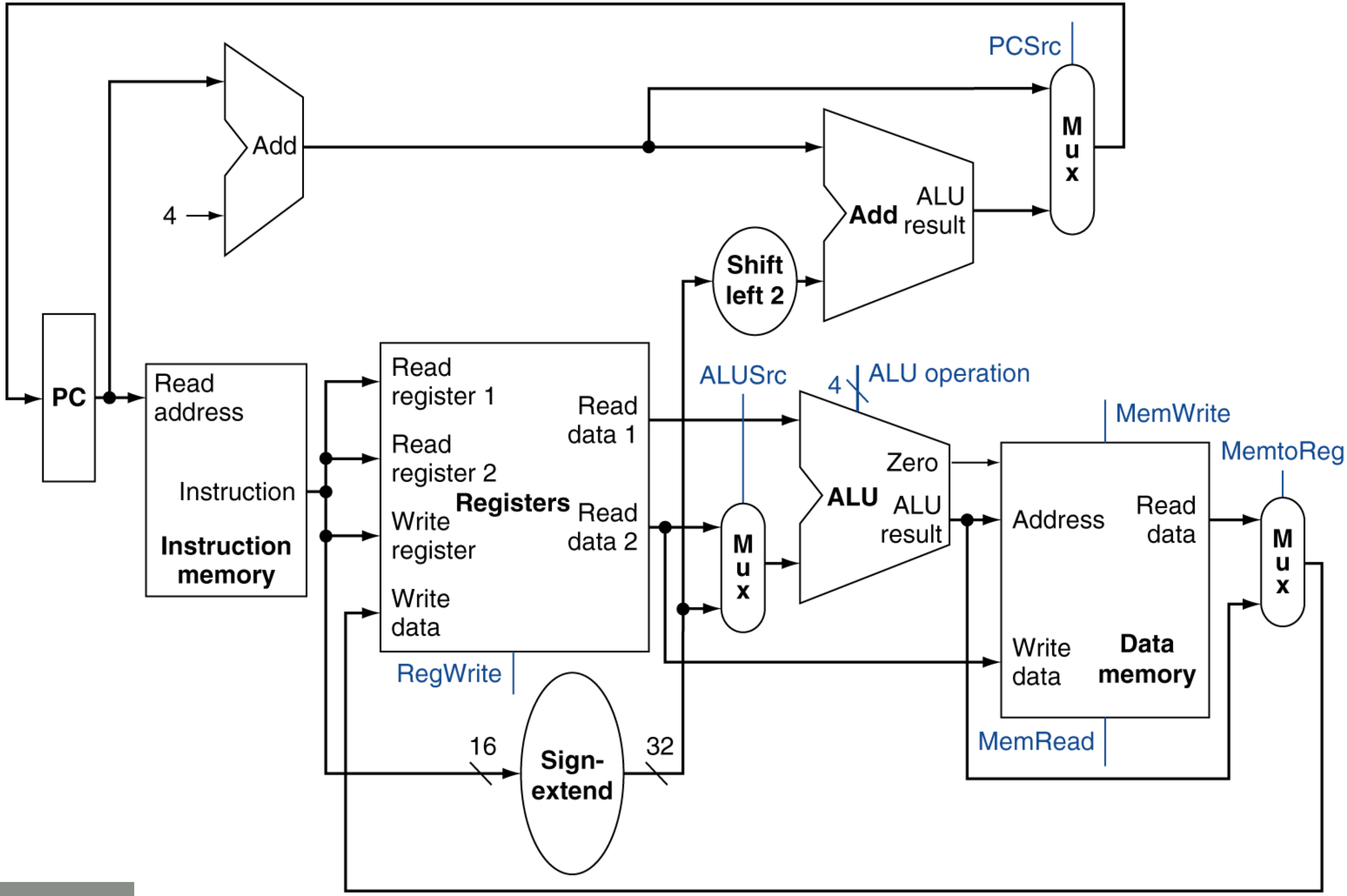
- We have everything except control signals (underline)



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

Full Datapath



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

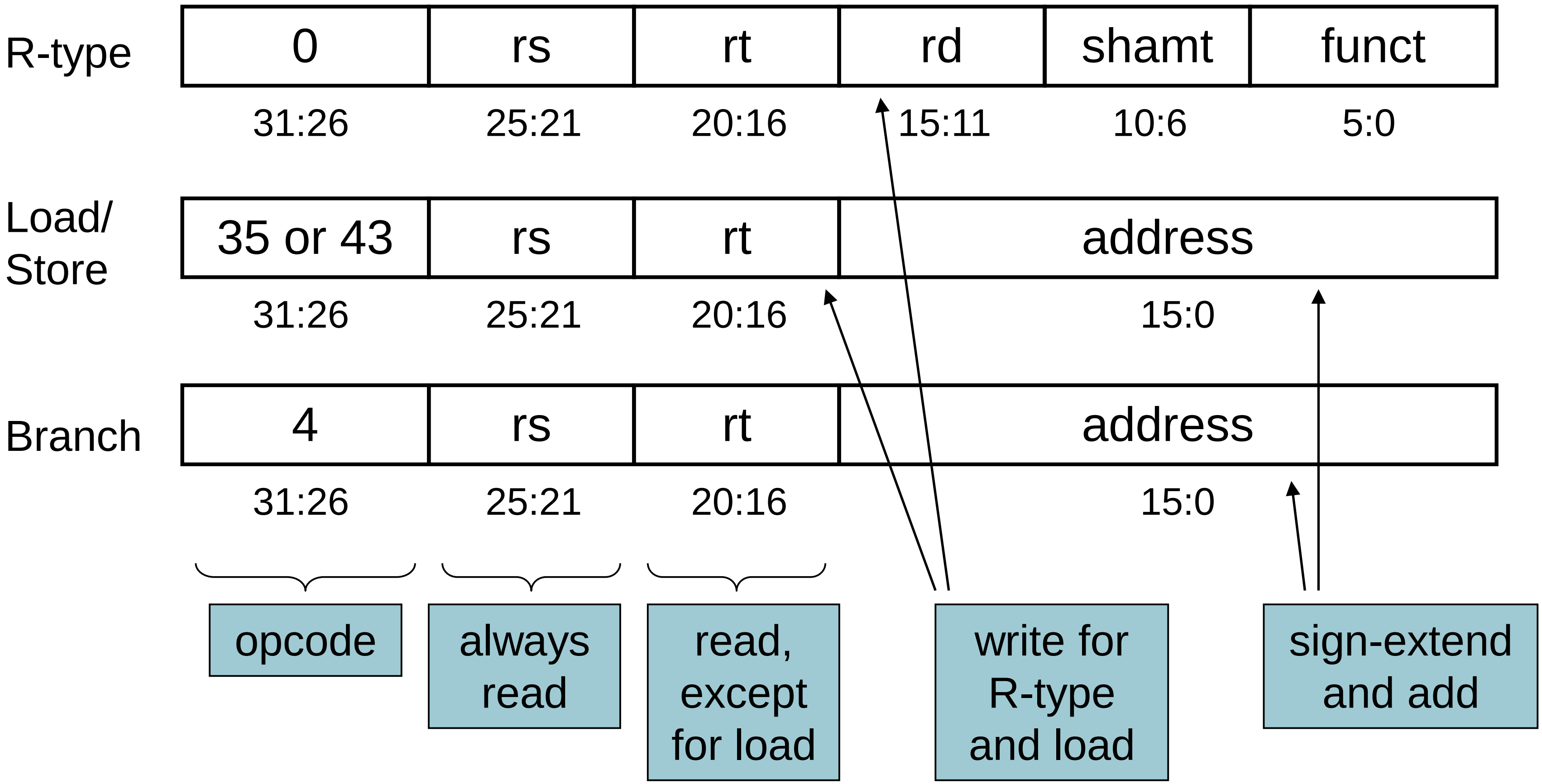
ALU Control

- 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



R-type	0	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0
Load/ Store	35 or 43	rs	rt	address		
	31:26	25:21	20:16	15:0		
Branch	4	rs	rt	address		
	31:26	25:21	20:16	15:0		

opcode

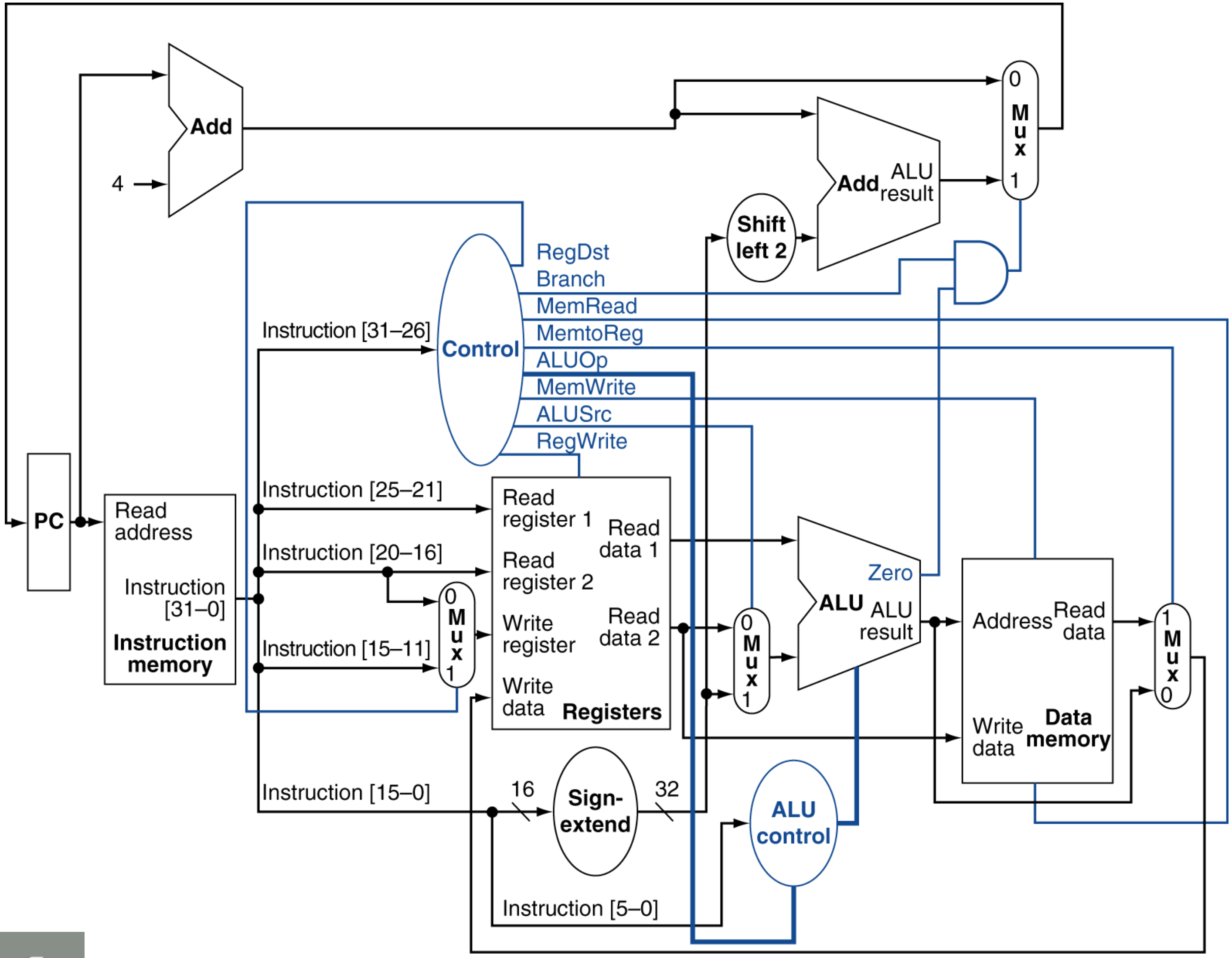
always read

read, except for load

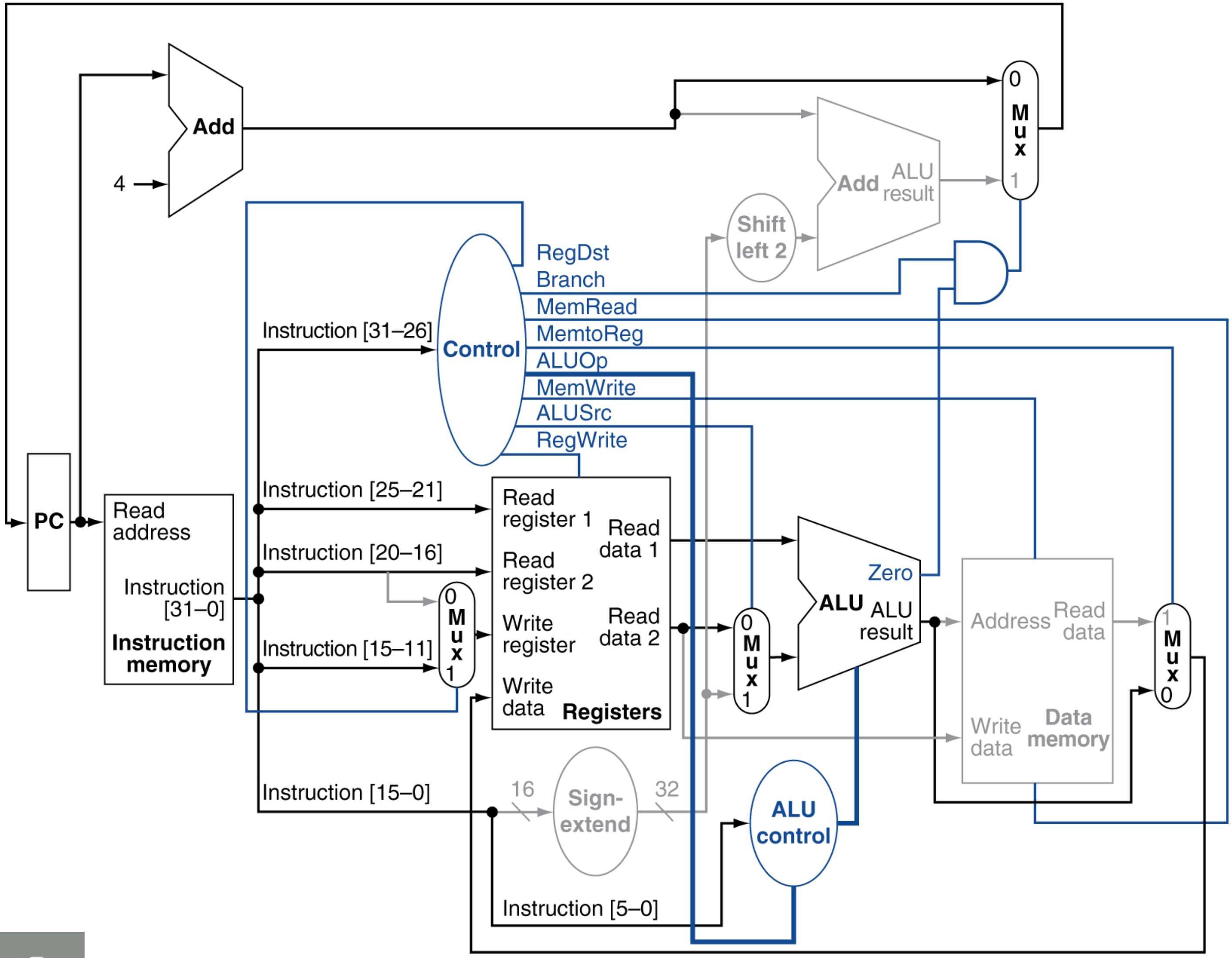
write for R-type and load

sign-extend and add

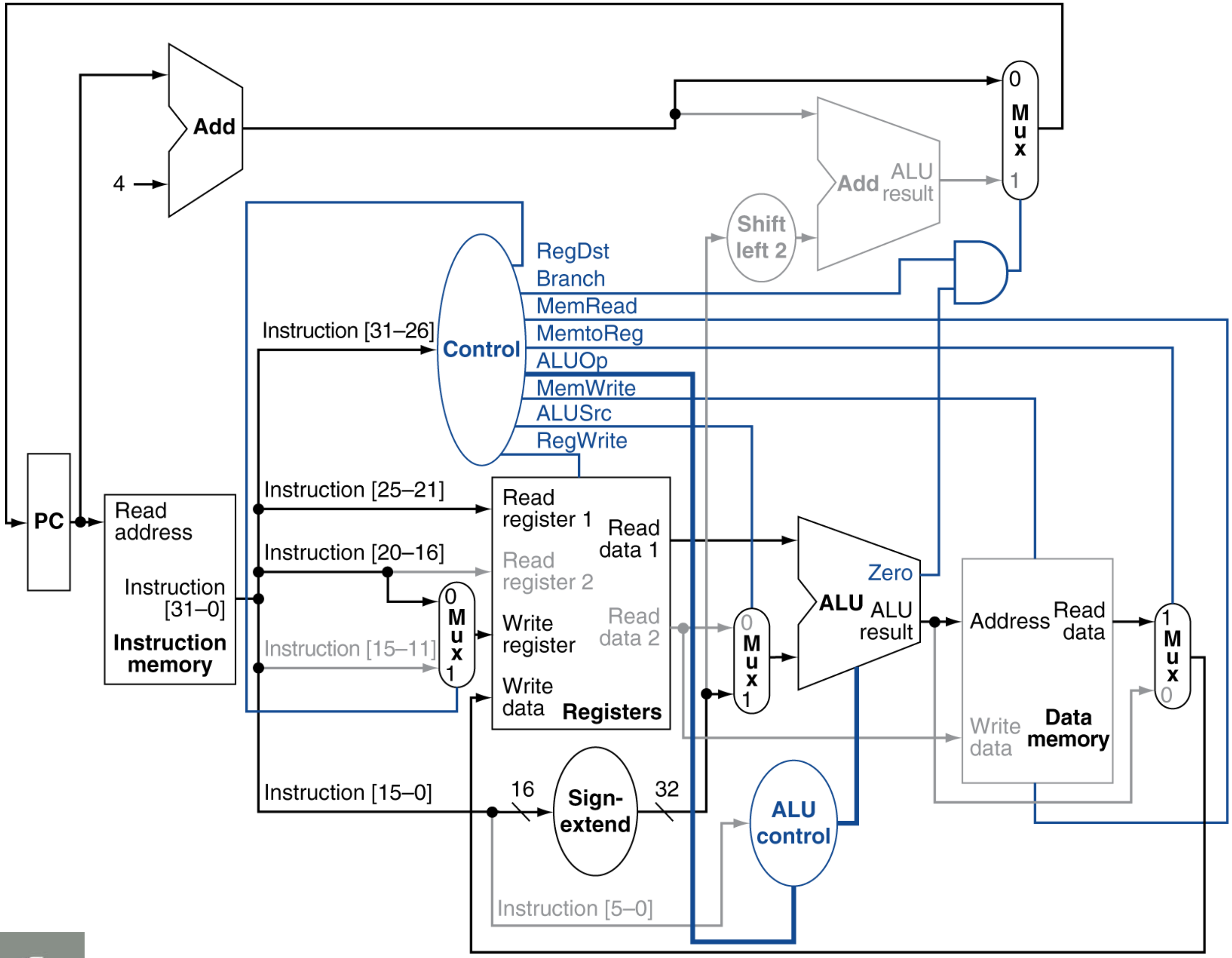
Datapath With Control



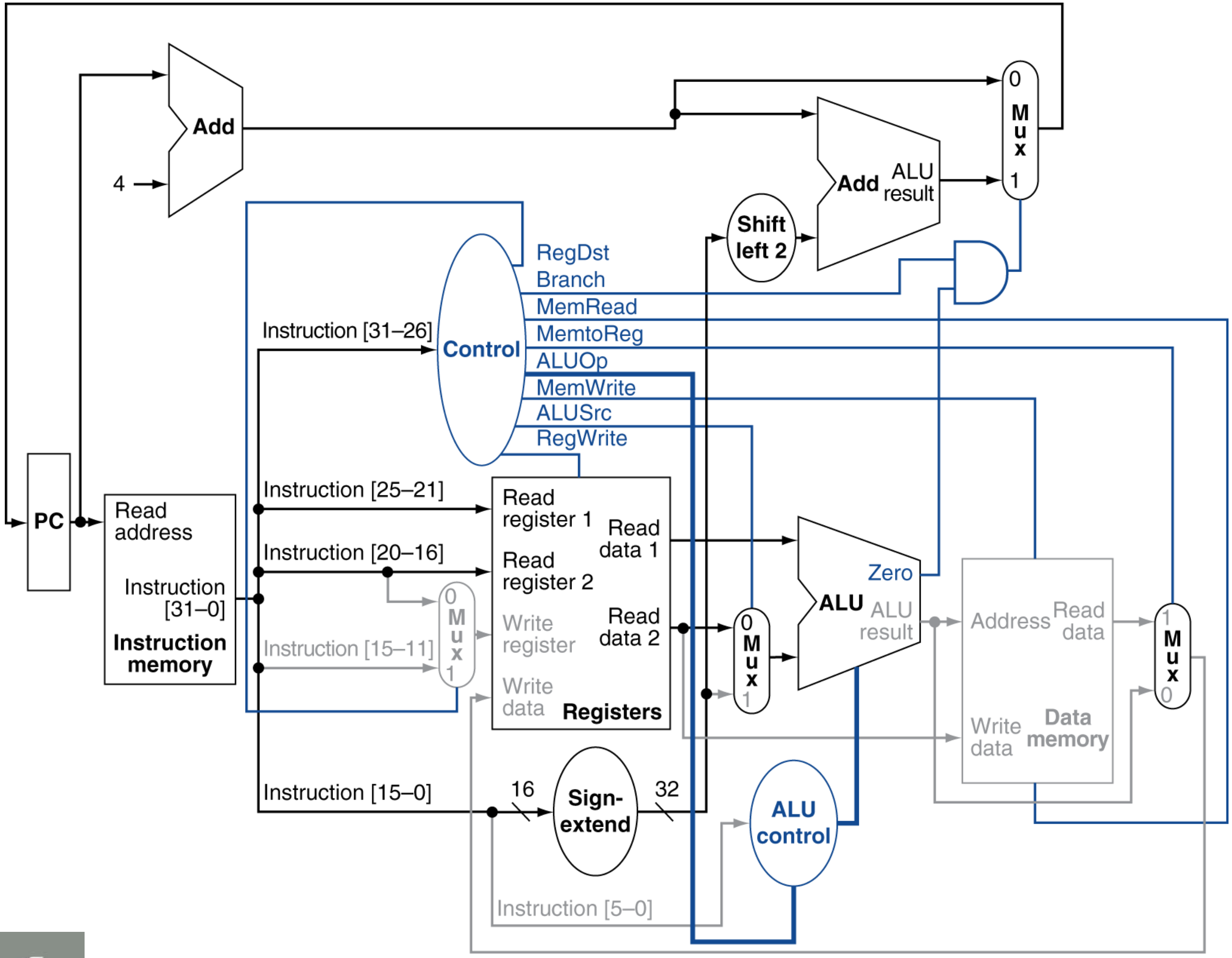
R-Type Instruction



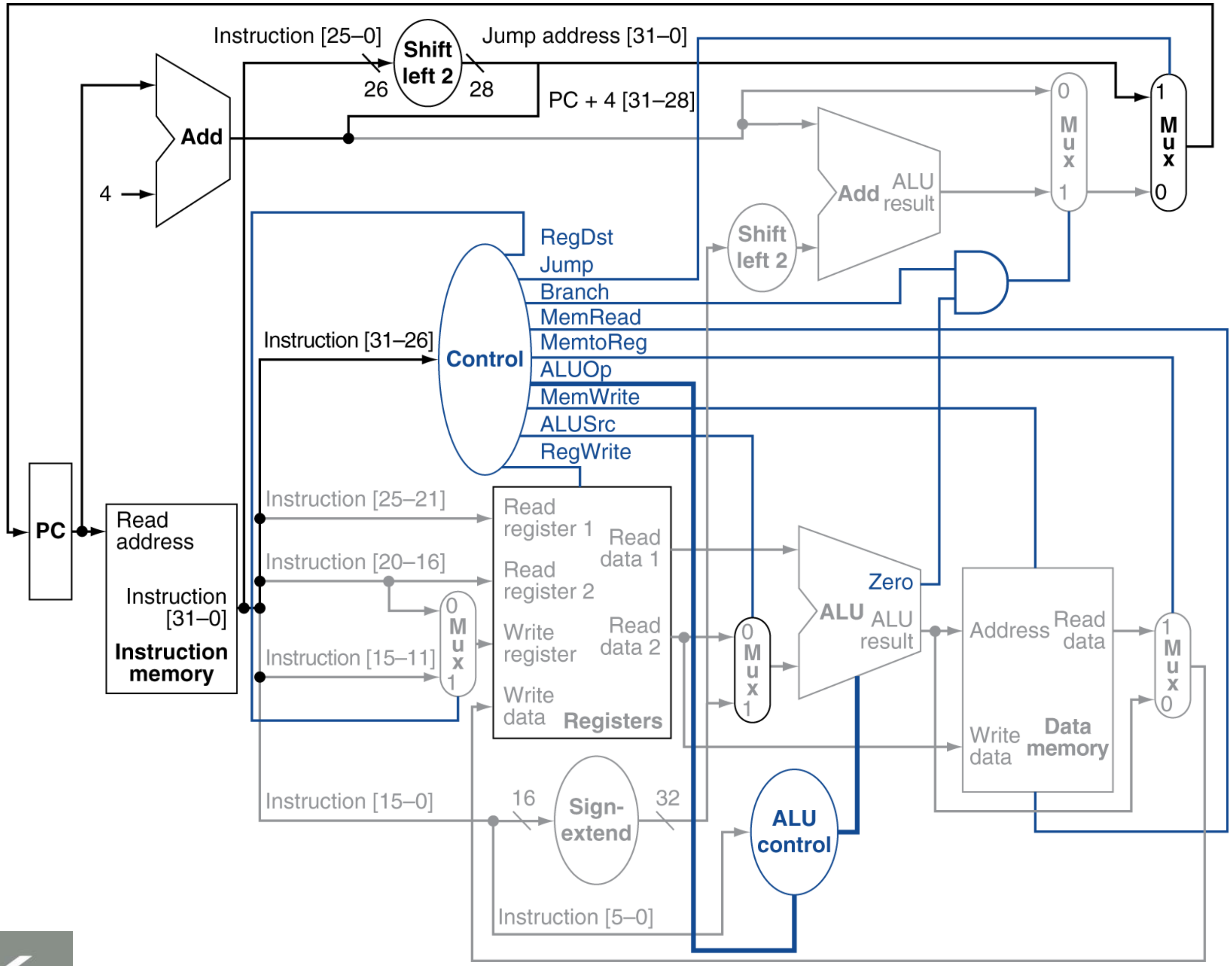
Load Instruction



Branch-on-Equal Instruction



Datapath With Jumps Added



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