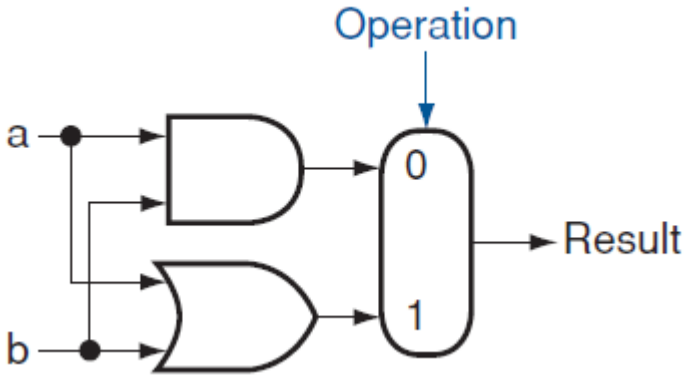
Arithmetic Logic Unit

- The ALU is the brawn of the computer
- Performs integer arithmetic operations
  - Addition and subtraction
  - Multiplication and division
- Performs logical operations
  - AND, OR, XOR
- Acts on commands from control unit

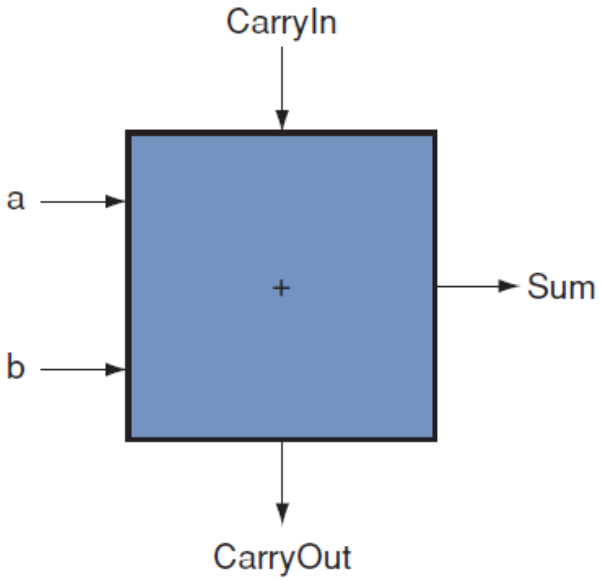
Building an ALU

- Multiple 1-bit ALUs can be used to build a 32-bit ALU
- This is called a bit sliced design
- The AND and OR operations map directly to gates



Building an ALU

- From module 3 we know how to build a full adder

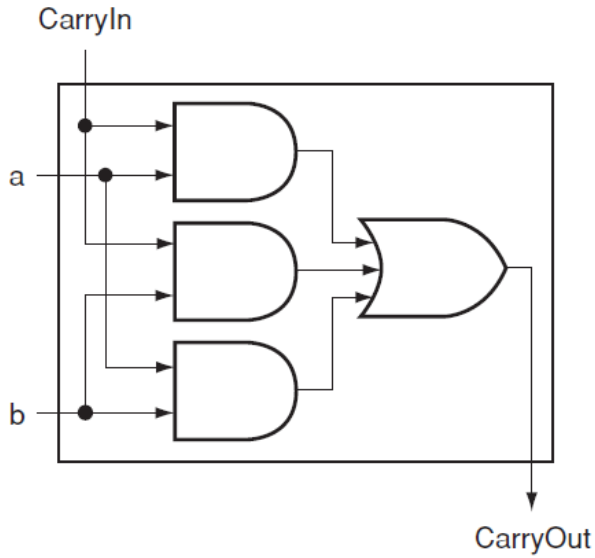


- The single bit inputs (a and b) together with the Carryin are added to produce the Sum and Carryout

Building an ALU

- Truth table and possible circuit to generate CarryOut

Inputs		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

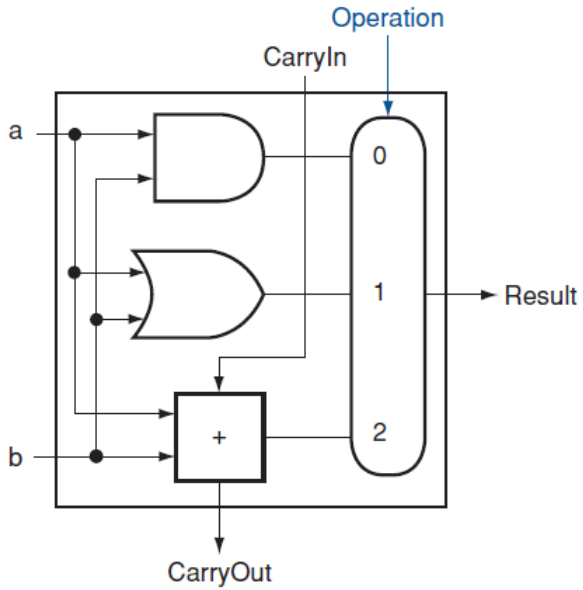


CarryOut = (a • CarryIn) + (a • b) + (b • CarryIn)

= (a • b) + (a + b) • CarryIn

Building an ALU

- 1-Bit ALU containing AND gate, OR gate and Full adder



- Full 32-Bit ALU can be builtup from multiple copies

Building an ALU

Subtraction can be included by noting that:

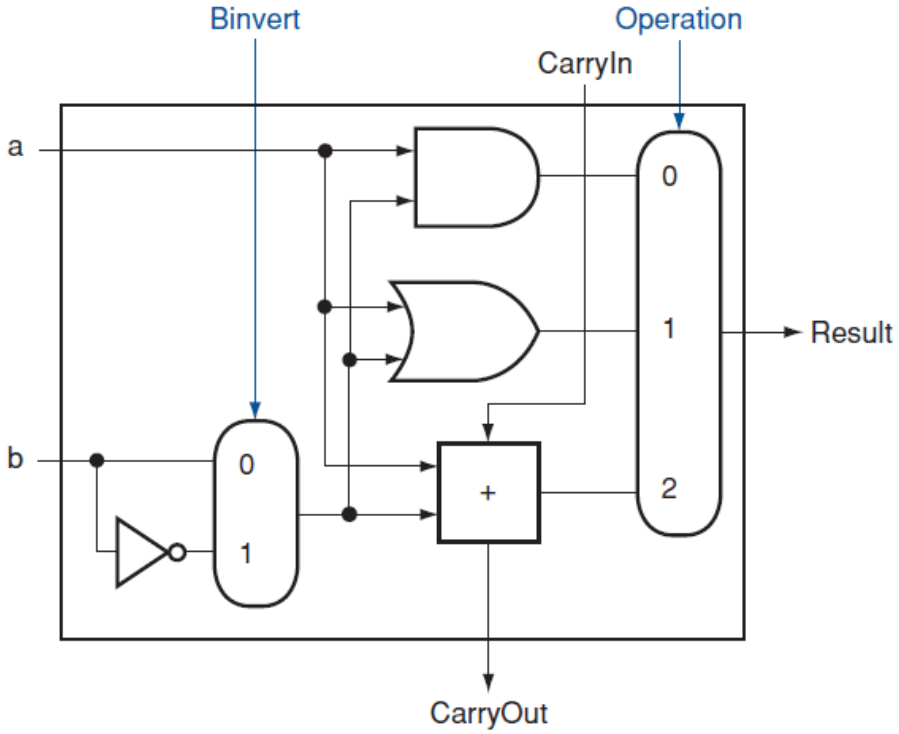
$$a + \bar{b} + 1 = a + (\bar{b} + 1) = a + (-b) = a - b$$

$\bar{b} + 1$ is the negative of b

Subtract b from a by adding the negative of b to a

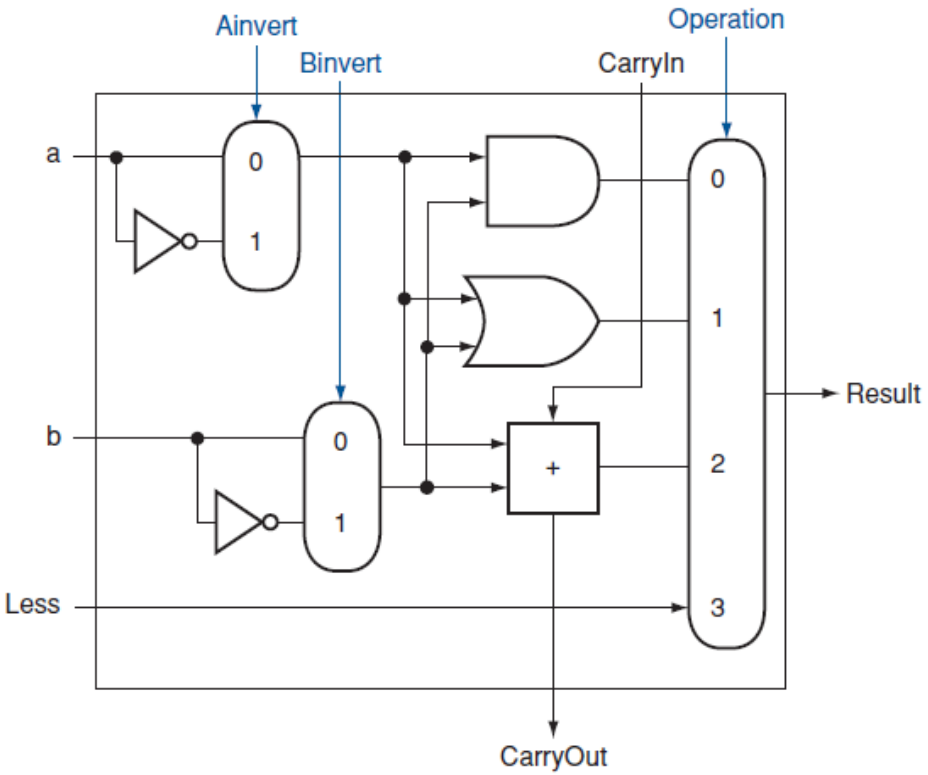
Uses two's complement of b

Building an ALU



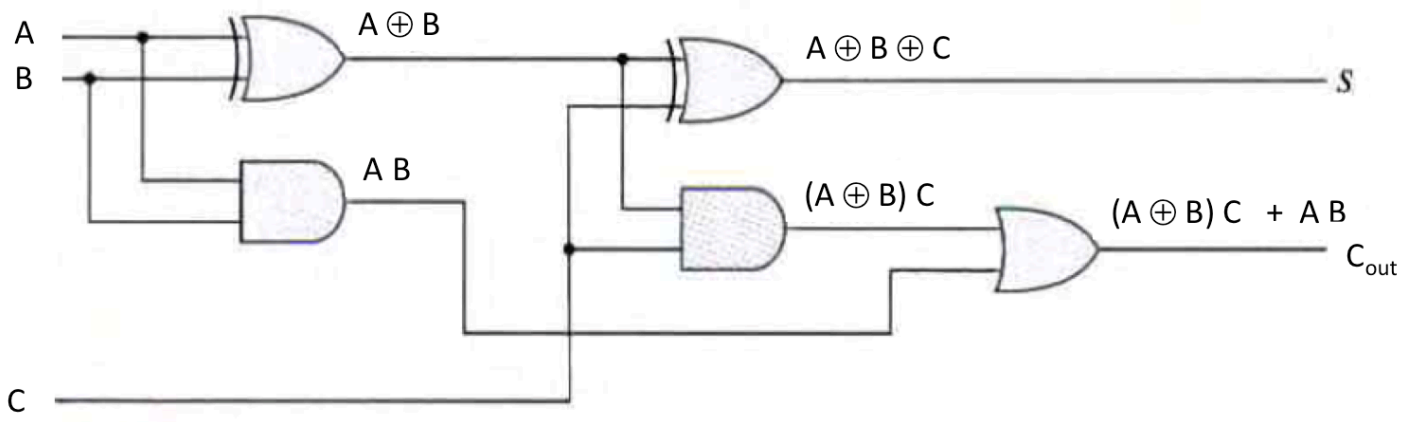
1-Bit ALU that subtracts, adds and performs AND and OR

Building an ALU



Ainvert allows the computation of (b – a) and other functions

Full Adder



A ⊕ B and A•B can be generated in parallel for each position

S is then generated as A ⊕ B ⊕ C (where C is the carry-in)

Hence S_0, the LSB of the sum requires 2 gate delays

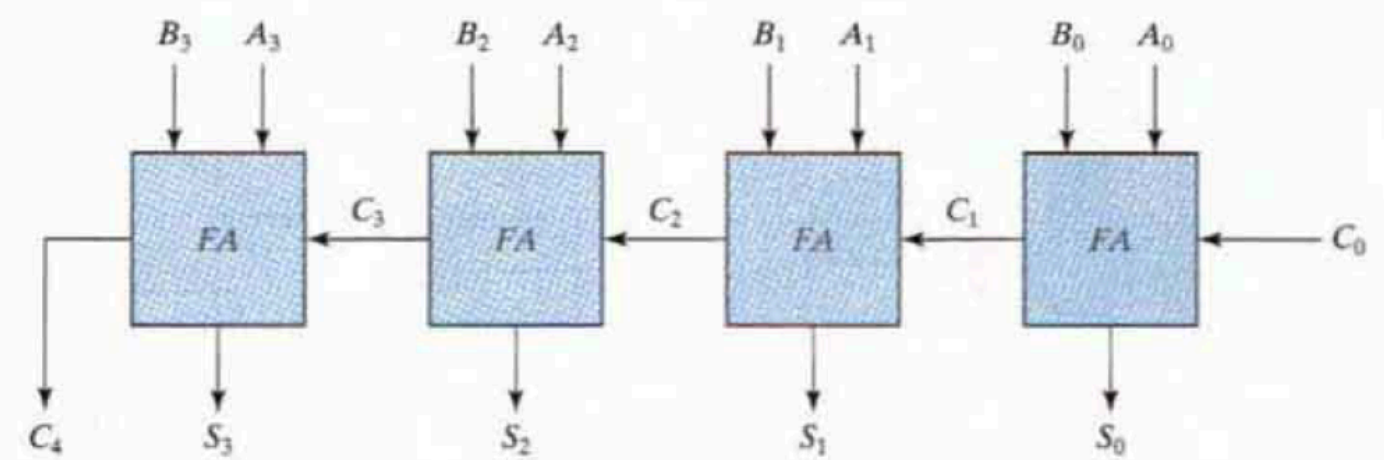
C_{OUT} is available after 3 gate delays

Other sum bits need the carry from the bit position on the right

S = A ⊕ B ⊕ C, so another gate delay is needed once C is available

C_{out} = (A ⊕ B) C + A B , and requires 2 more gate delays

Four-bit Ripple Carry Adder



S_0 is available after 2 delays
C_1 is available after 3 delays
S_1 is available after 1 + 3 = 4 delays
C_2 is available after 2 + 3 = 5 delays
S_2 is available after 1 + 5 = 6 delays
C_3 is available after 2 + 5 = 7 delays
S_3 is available after 1 + 7 = 8 delays
C_4 is available after 2 + 7 = 9 delays

Faster Addition

Recall that for the full adder:

CarryOut = (a • b) + (a ⊕ b) • CarryIn

If a and b are both 1, they generate a CarryOut when added

If either a or b is 1, Carryin is propagated to CarryOut

In general: $c_{i+1} = (a_i \bullet b_i) + (a_i + b_i) \bullet c_i = g_i + p_i \bullet c_i$

$g_i = a_i \bullet b_i$ and $p_i = a_i \oplus b_i$ the subscript indicates the bit position

C_0 is the input carry for bit 0, the LSB

$C_0 = 0$ for addition

$C_0 = 1$ for subtraction

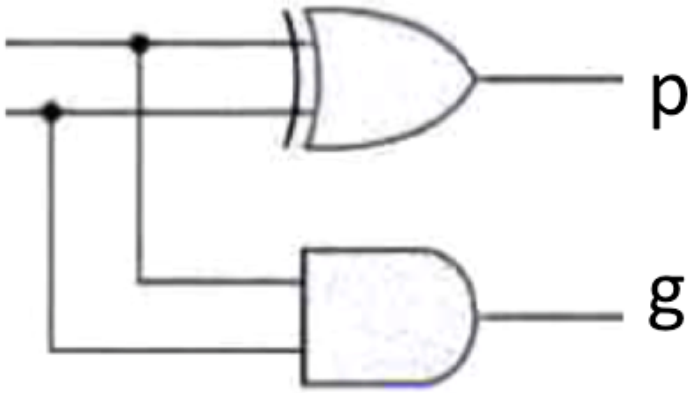
Faster Addition

recurrence relation for all of the output carries:

$c1 = g0 + p0 \bullet c0$

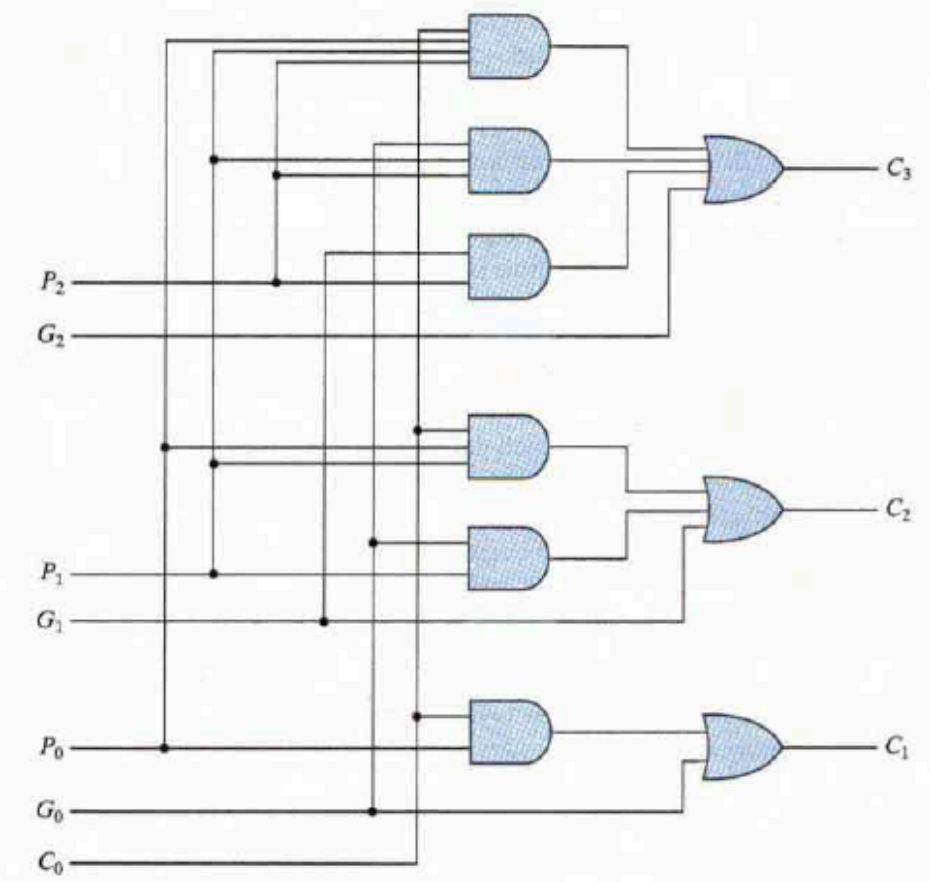
$c2 = g1 + p1 \bullet c1$

$c_i = g_{i-1} + p_{i-1} \bullet c_{i-1}$ (for i>0)



Inputs are the data bits for each position

Faster Addition

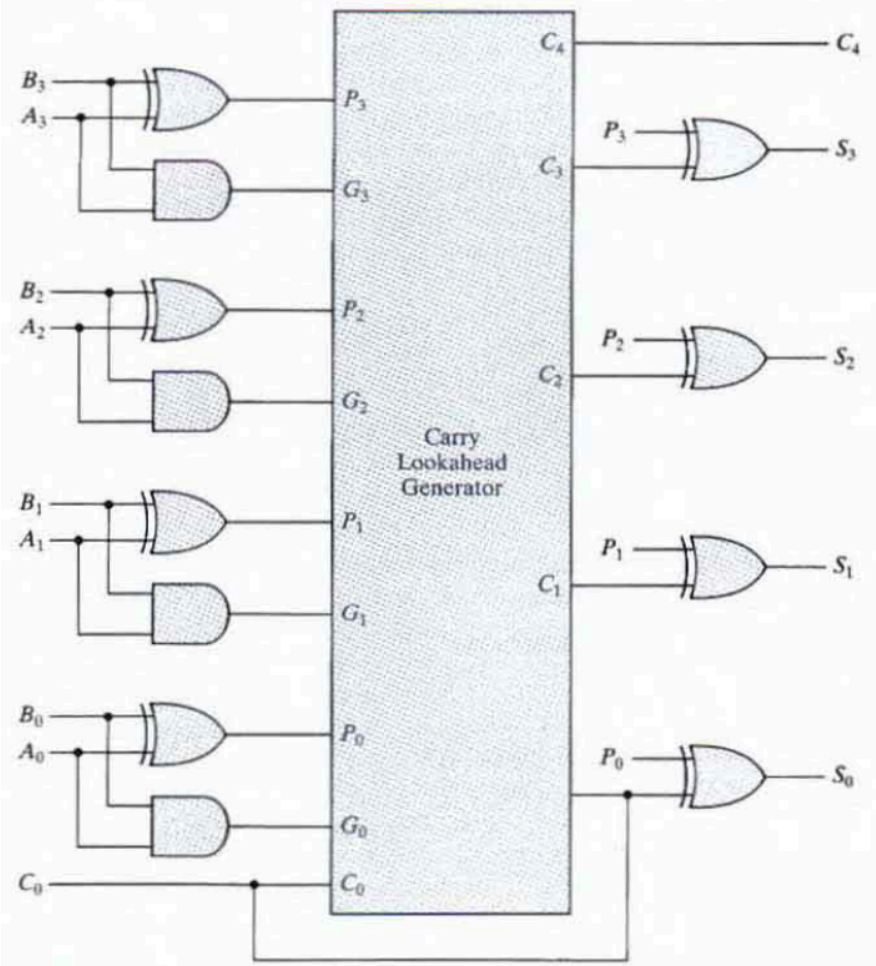


Propagate & generate bits are used to produce carry bits

Carries require 2 gate delays and are produced in parallel

Lookahead carry generator circuit

Faster Addition



Bits in sum are available after
1+2+1 = 4 gate delays

4-bit adder with lookahead carry

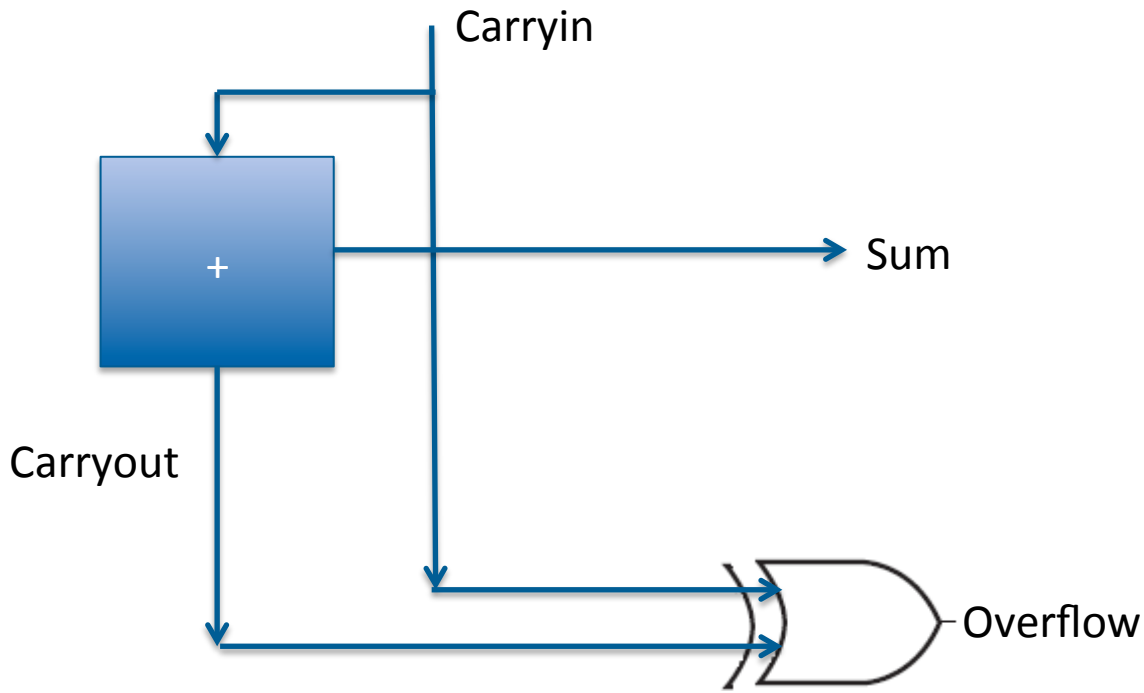
Look Ahead Carry Adder

- c0 and all of the a_i and b_i bits are known up front
- All of the p_i and g_i can be generated in parallel (1 delay)
- All carry bits c_i are generated in parallel (2 more delays)
- All sum bits are produced in parallel (1 more delay)
- This is faster than the ripple carry adder
- Ripple carry adder computes each sum sequentially from LSB to MSB

Overflow Detection

- Overflow exists if the result is larger than the register
- Mismatching carry-in and carry-out of the MSB indicates overflow
- Carry out alone signals overflow only for unsigned numbers
- Signed arithmetic requires a separate overflow indicator bit
- Negative results have MSB = 1, if no overflow occurs
- Positive results have MSB = 0, if no overflow occurs

Overflow Detection



For MSB, compare carryin with carryout

Mismatch indicates signed overflow

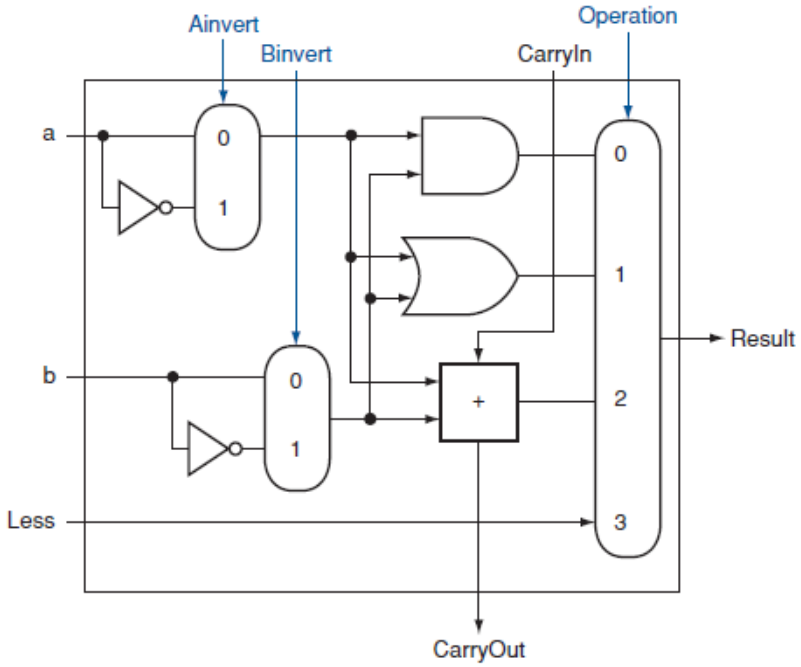
ALU for MIPS Core

- The ALU must also support slt (set on less than) instruction
- The slt instruction generates 1 in the result if a < b
- Otherwise, it generates 0
- If a-b < 0, then a < b (indicated by sign bit in result)
- Assumes no overflow occur

ALU for MIPS Core

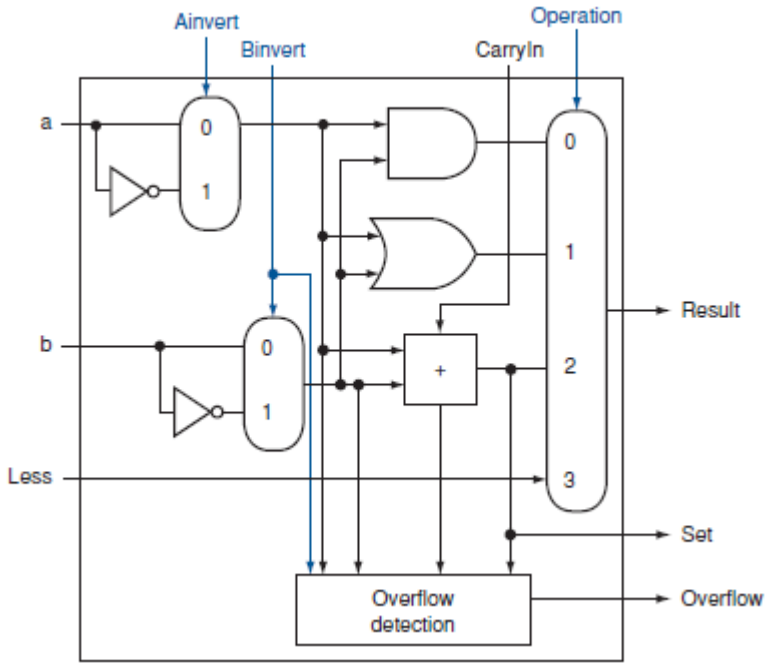
- A new input, Less, is included for ALU
- Less = 0 for the high order 31 bit positions
- LSB of result = 1 if a < b, otherwise LSB = 0
- Logic is only needed in the ALU for the MSB to generate both Overflow and Set (replaces LSB of result)

ALU for MIPS Core



ALU for all but the MSB position

ALU for MIPS Core



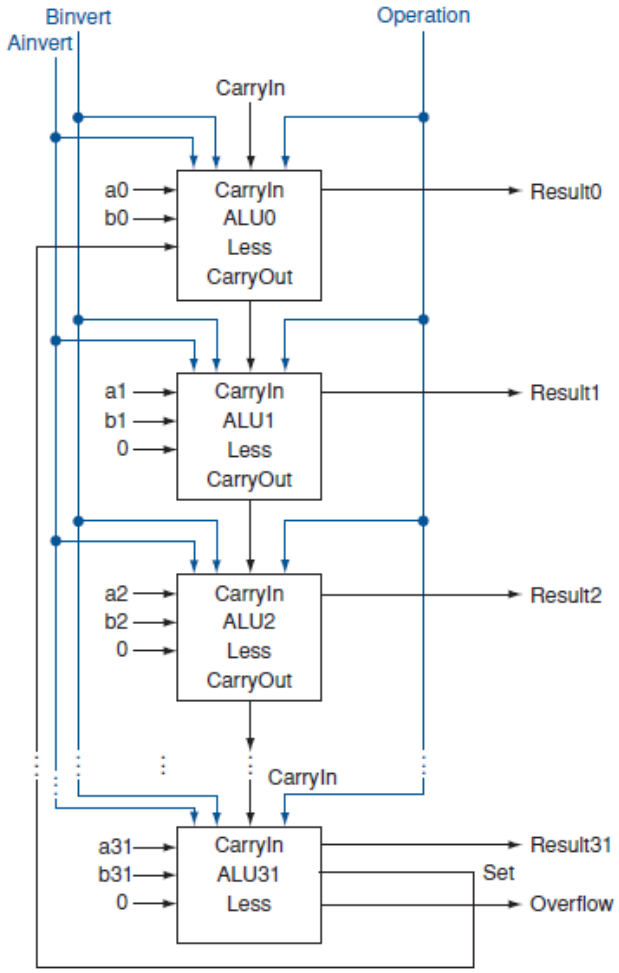
ALU for the MSB position

Includes logic to generate Set bit for slt instruction
Also generate overflow flag

ALU for MIPS Core

Includes support for slt

Set from MSB routed back to LSB of result



ALU for MIPS Core

- Branch on equal (beq) requires zero flag output from ALU
- Branch is taken if zero flag = 1 (PC is loaded with target address)
- Otherwise PC+4 is used as address of next instruction
- If a-b = 0, a and b are equal, so zero flag is set to 1
- NOR of all result bits yields zero flag

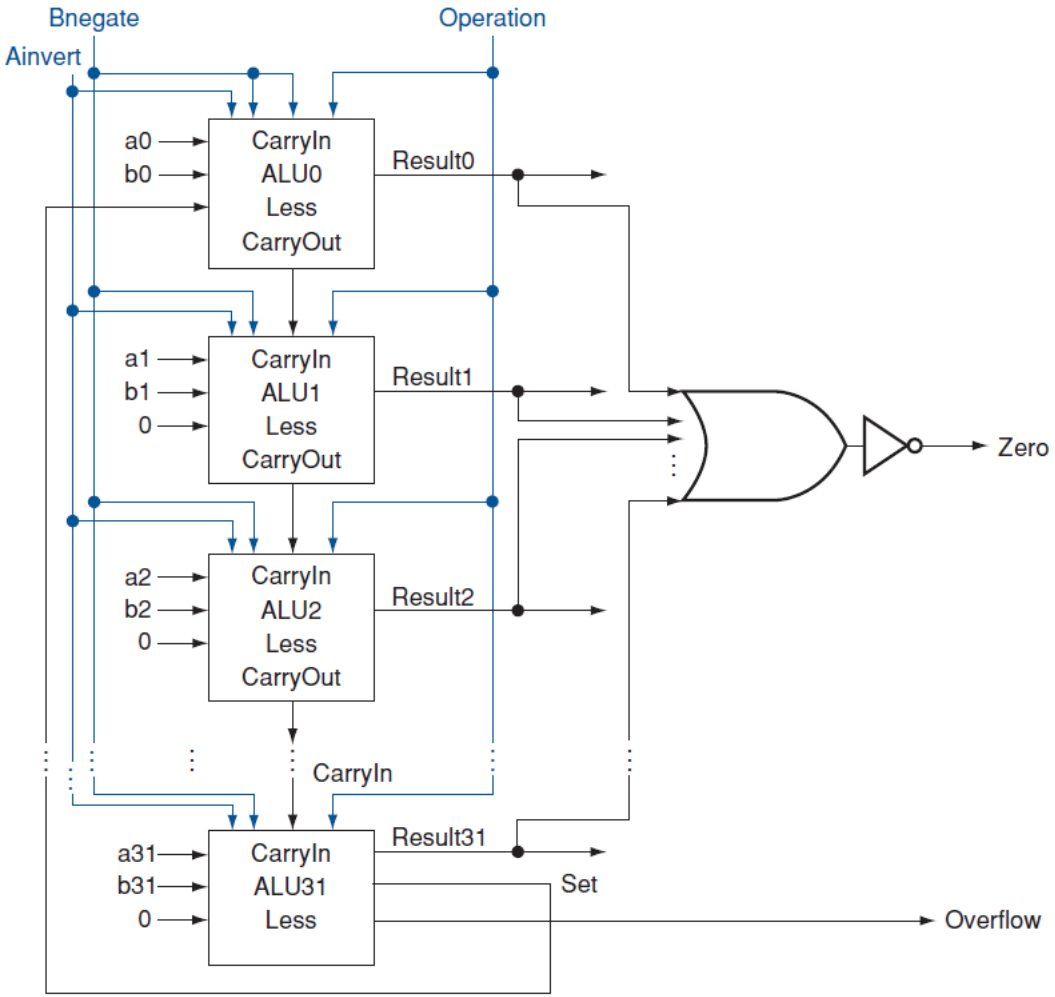
$$\text{Zero} = \overline{(\text{Result31} + \text{Result30} + \ldots + \text{Result2} + \text{Result1} + \text{Result0})}$$

ALU for MIPS Core

Final 32-bit ALU

Bnegate negates b (Carryin for LSB = 1 and all b bits are inverted)

So a-b is computed



Supports all operations needed for the core MIPS instruction subset

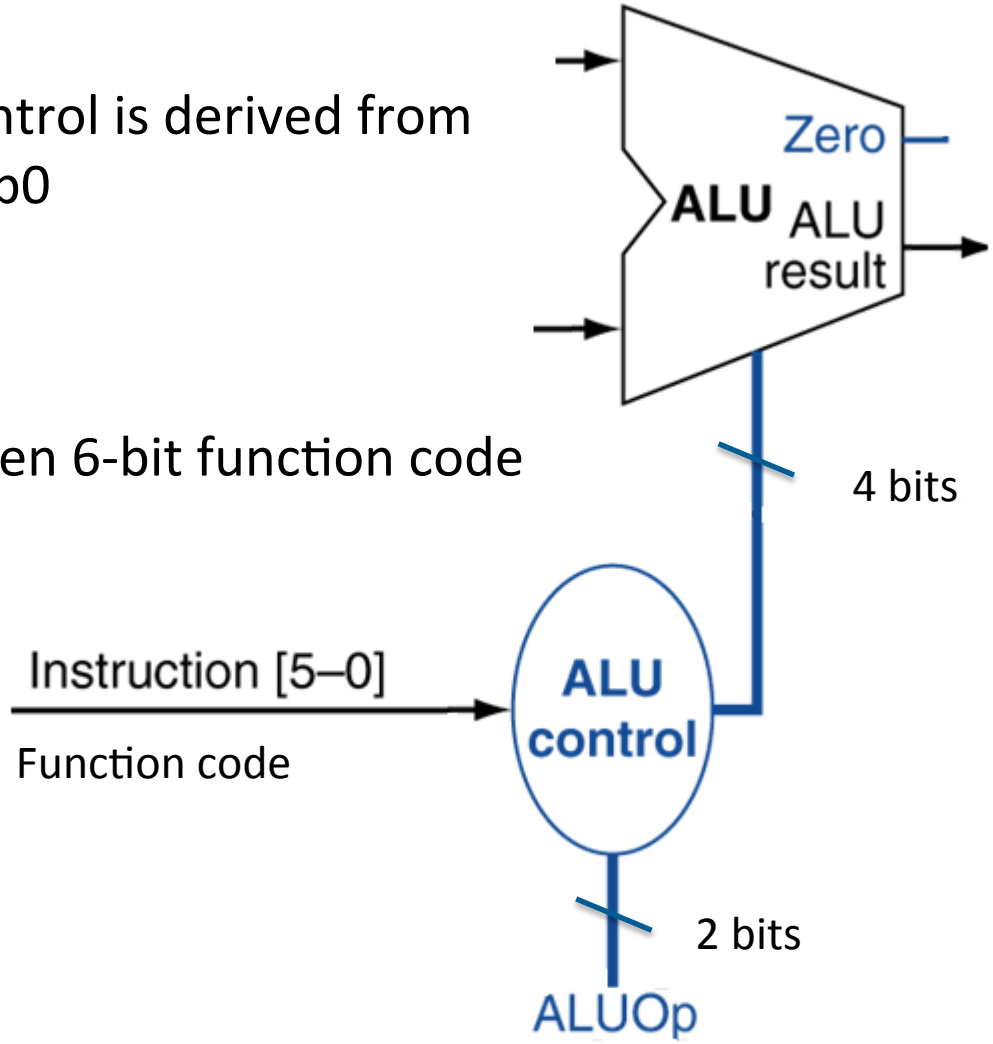
ALU Control

- The ALU takes two input operands
- Generates result as directed by 4-bit control signal
- Sets zero flag if result is 0
- 4-bit control signal derived from opcode & function code

ALU Control

The 4-bit ALU control is derived from ALUOp1 & ALUOp0

If ALUOp = 10, then 6-bit function code is also used



ALU Control

- Instruction type determines ALU operation
  - Load/Store: operation = add
  - Branch: operation = subtract (compare operands)
  - R-type: operation depends on funct field

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

ALU Control

- Combinational logic derives the 2 ALUOp bits from opcode

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

ALU Control

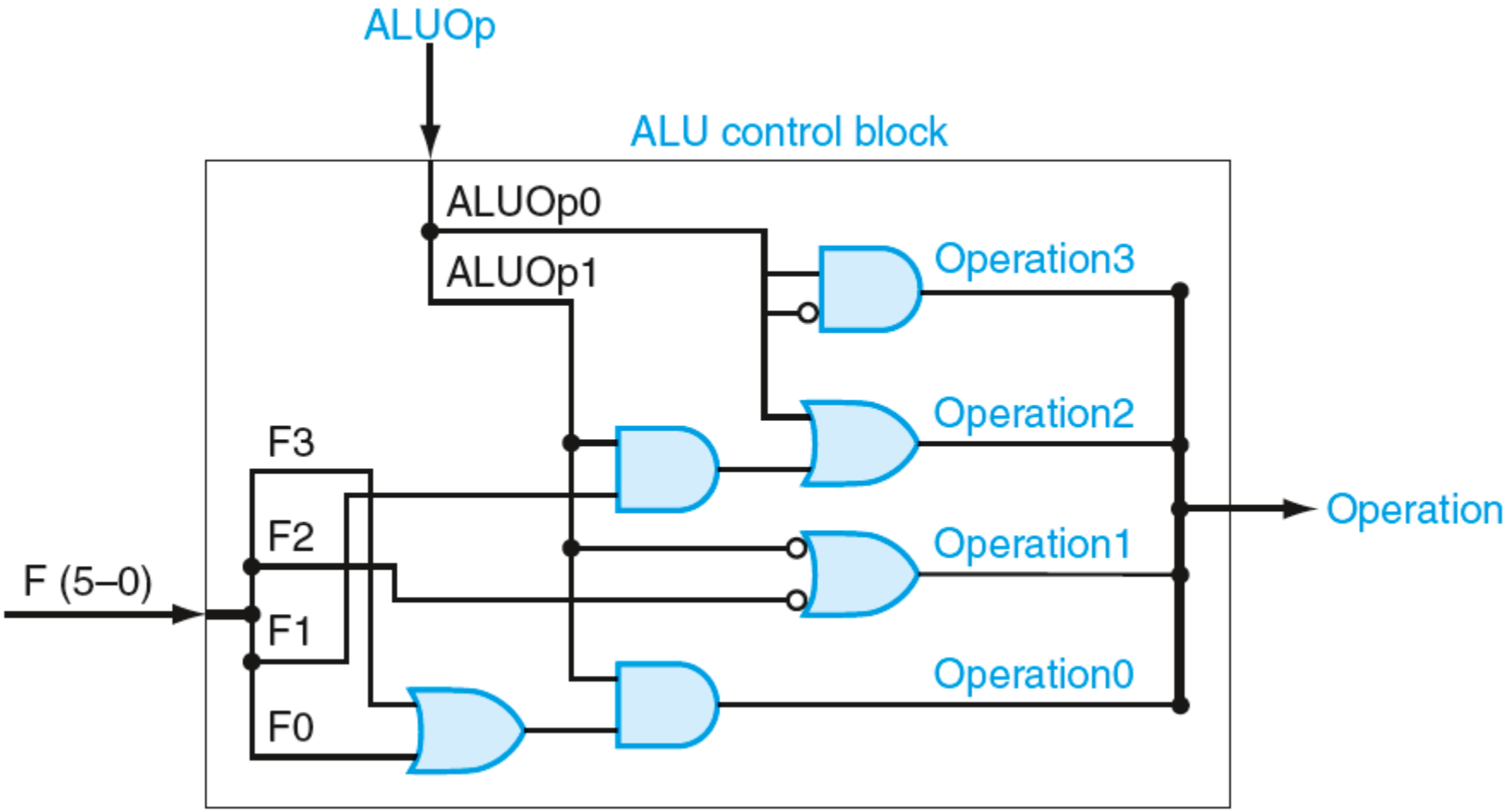
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

- Truth table for the 4 ALU control bits (called Operation)

- Depends on ALUOp and instruction function code field

(X's represent "don't cares")

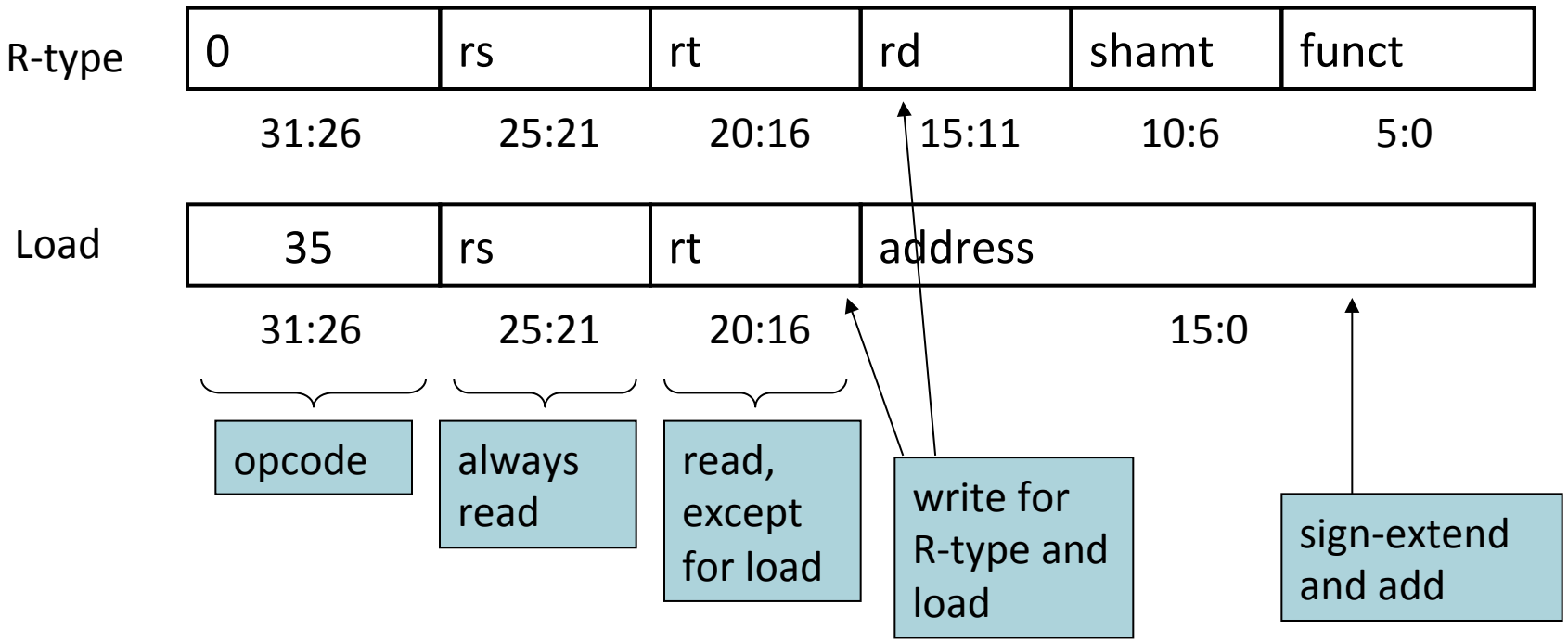
ALU Control



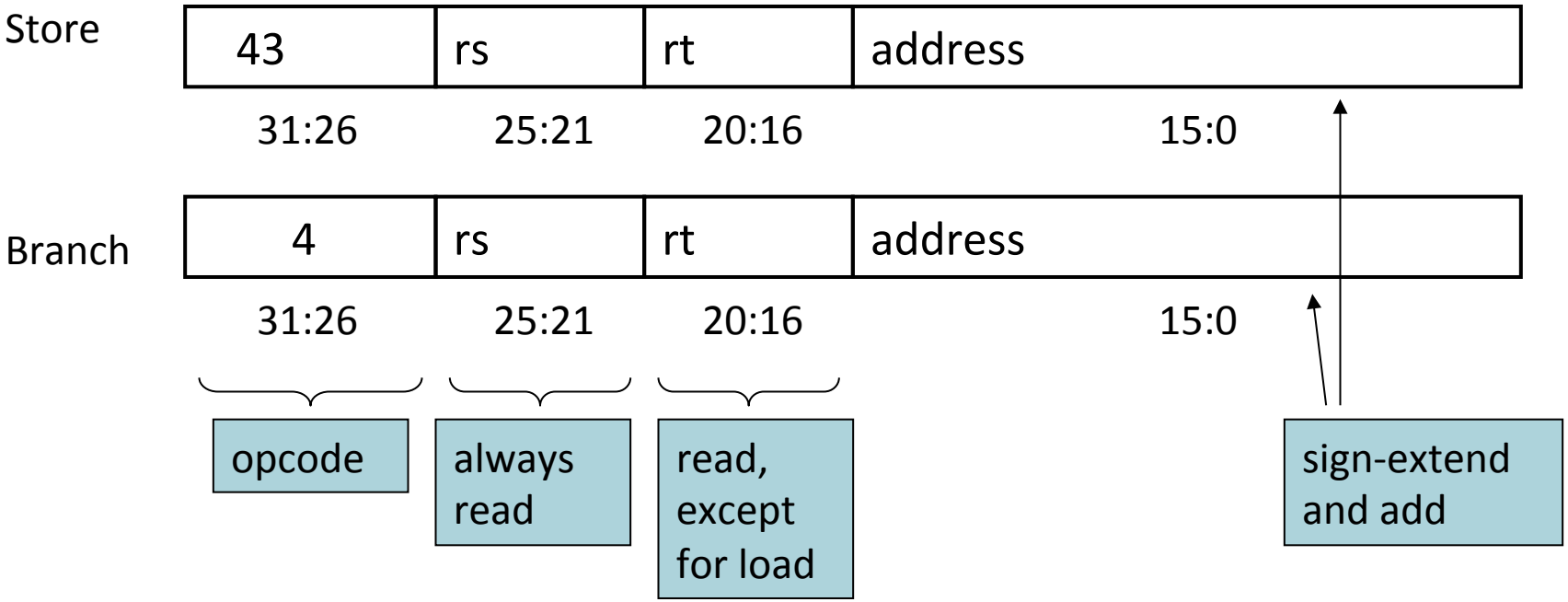
Combinational circuit to generate the 4-bit ALU control signal

Main Control Unit

- Control signals are derived from the instruction



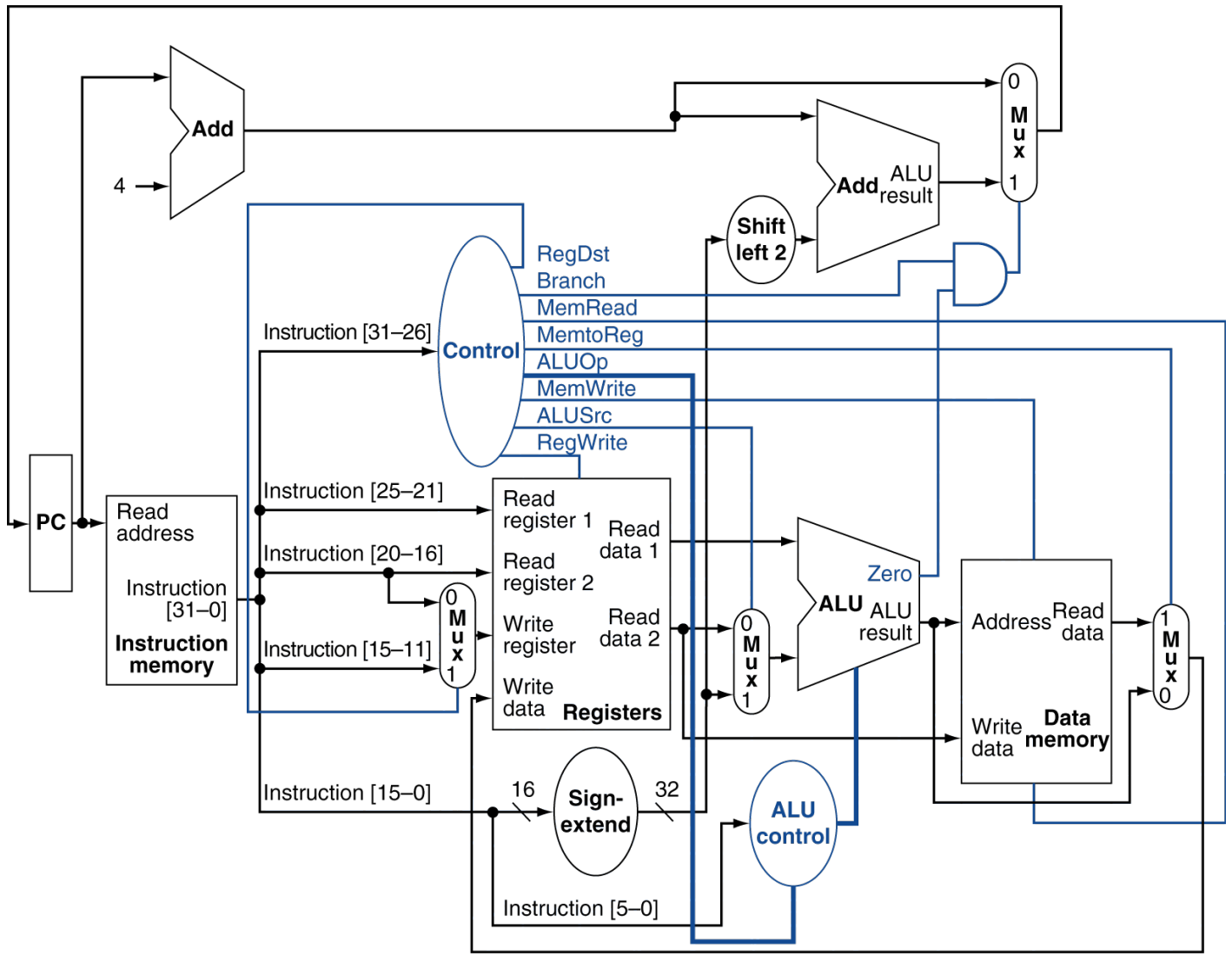
Main Control Unit



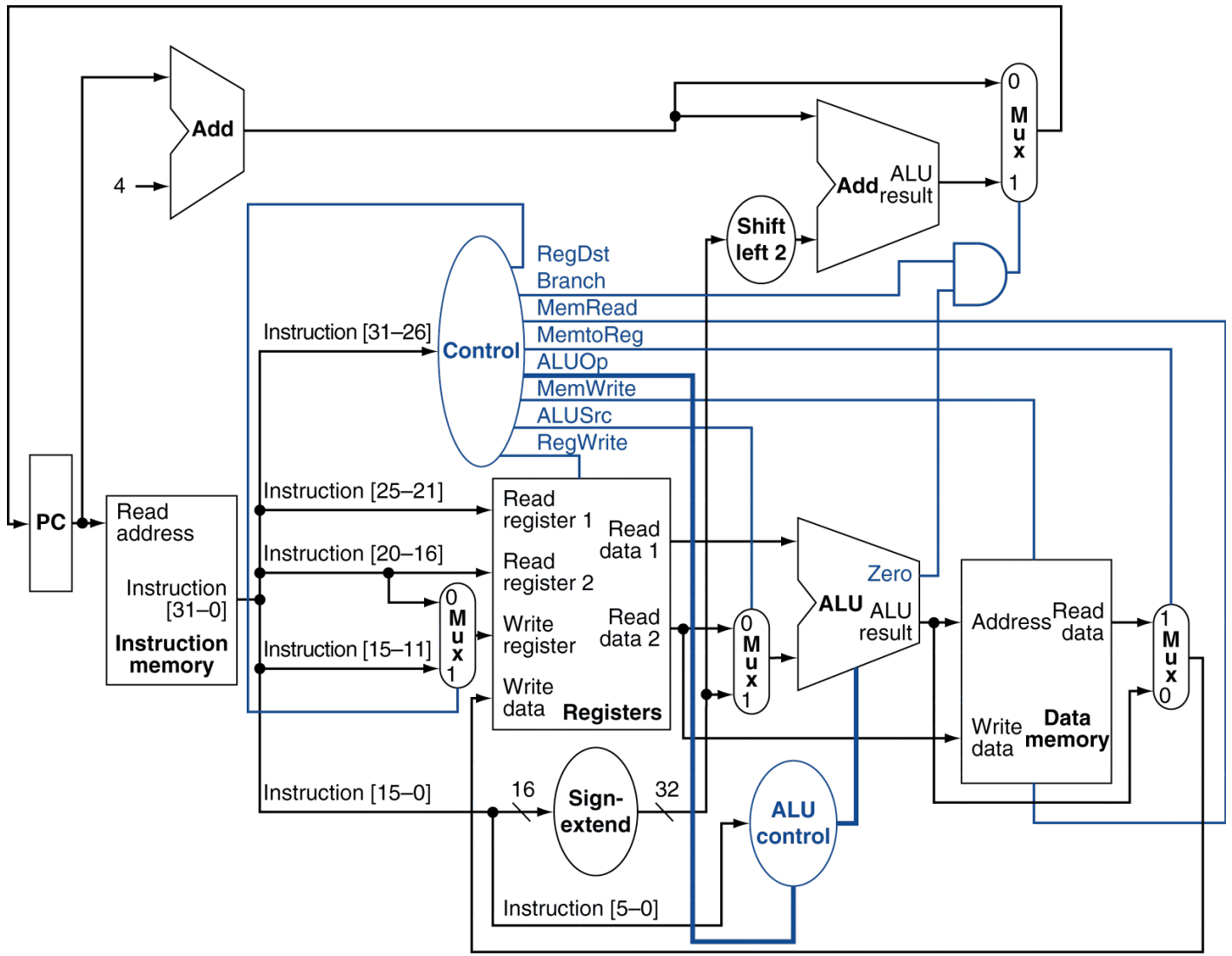
Store copies content of rt register into memory, rt not changed

Branch subtracts rs from rt to generate zero flag, rs and rt not changed

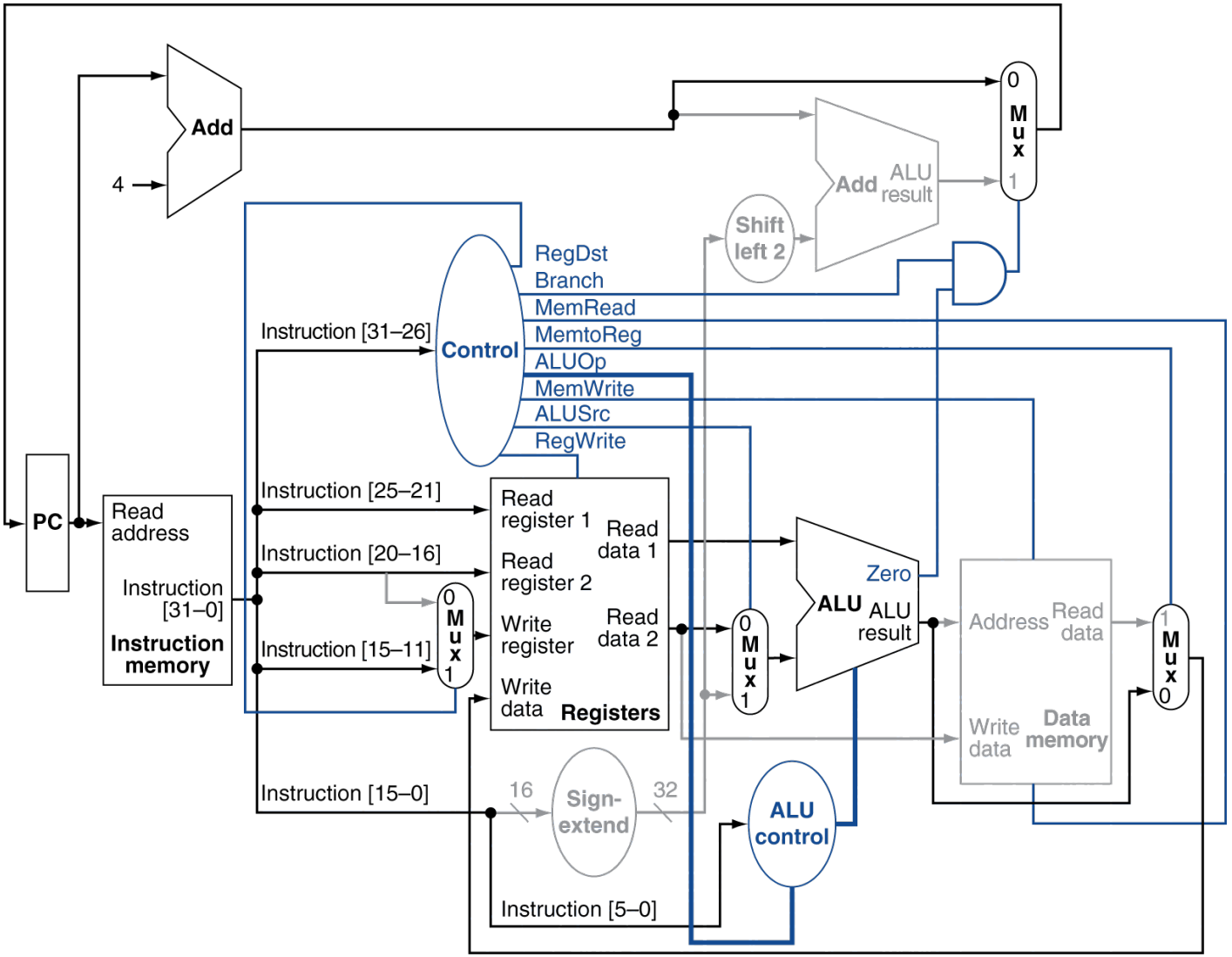
Datapath with Control



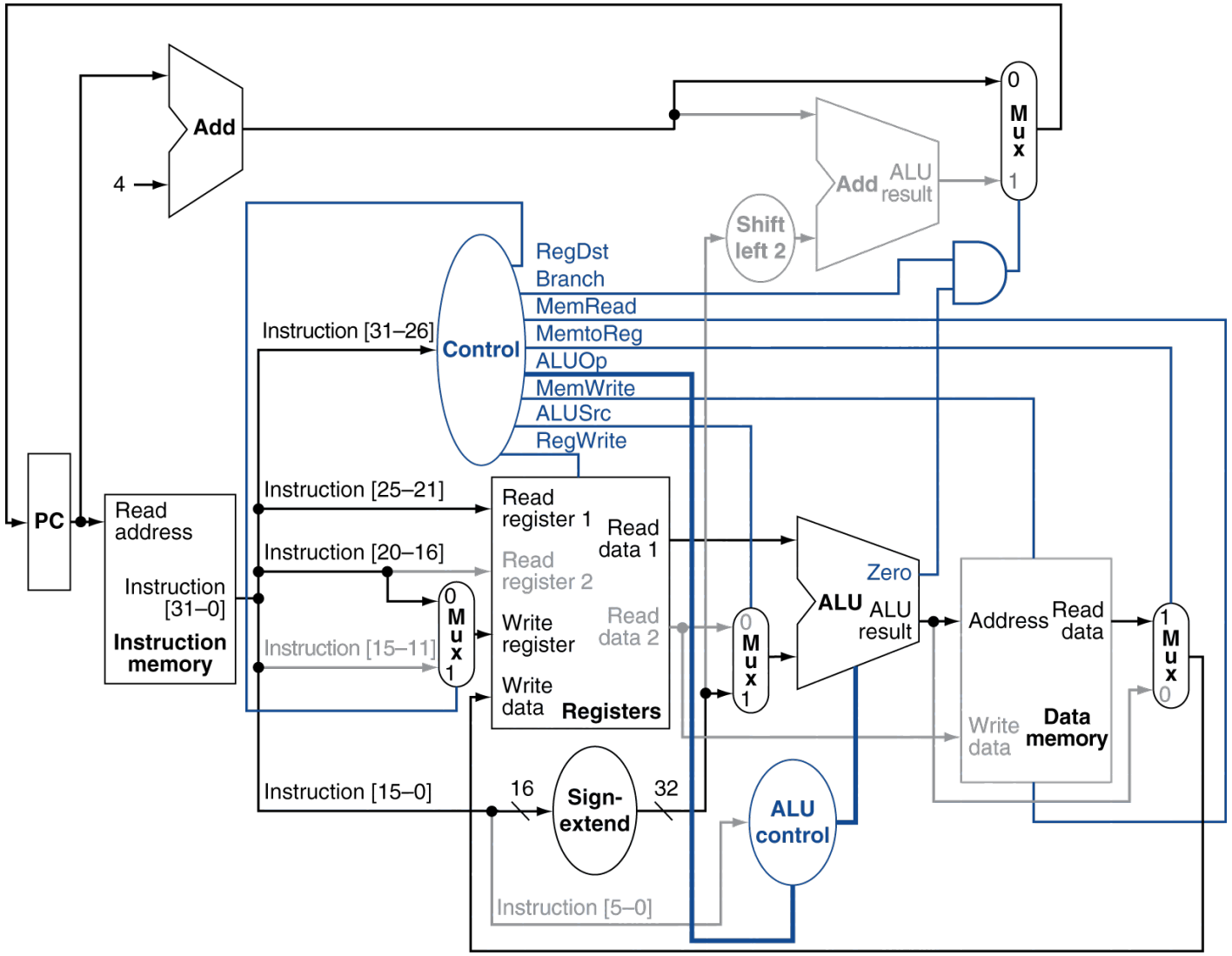
Datapath with Control



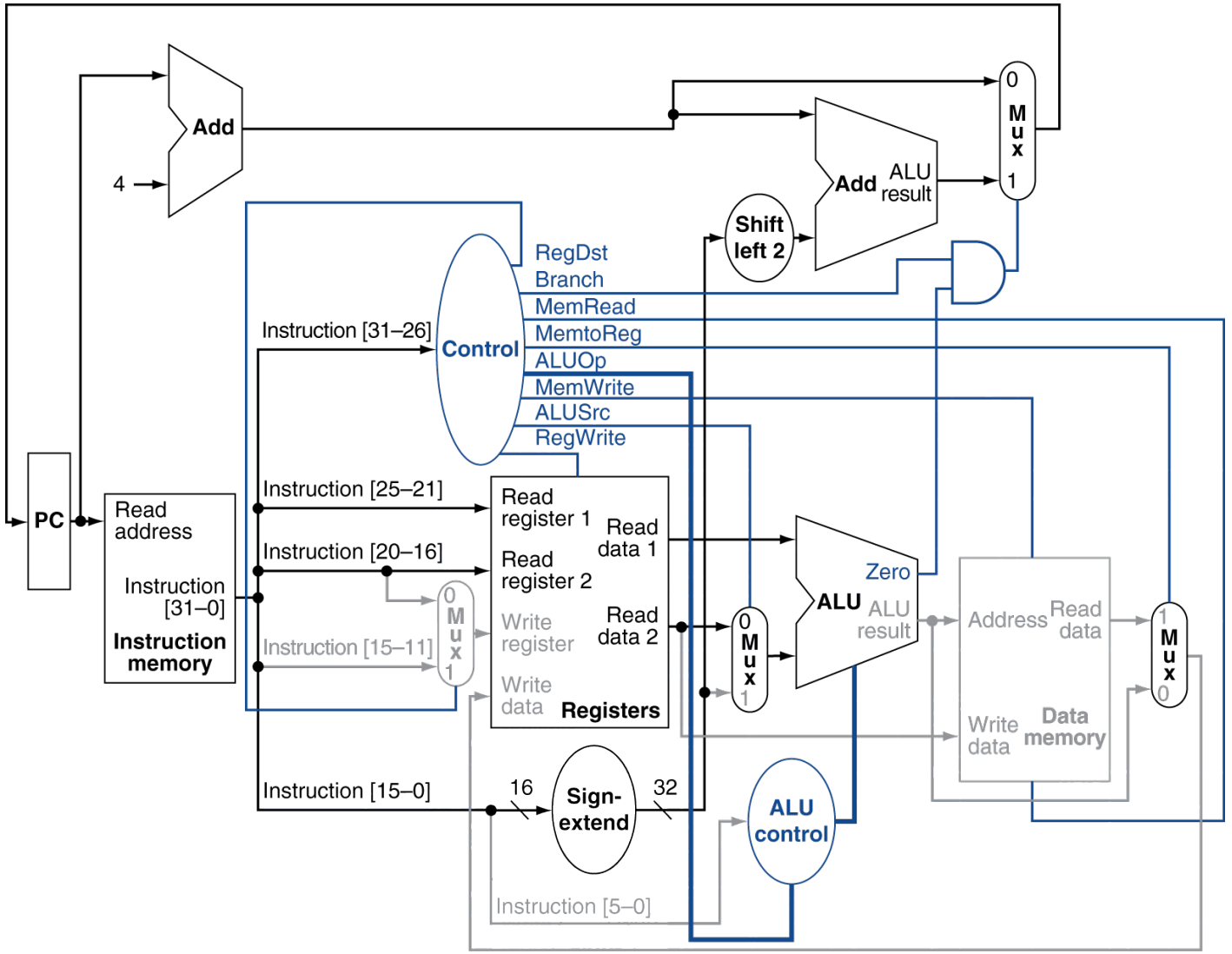
R-type Control



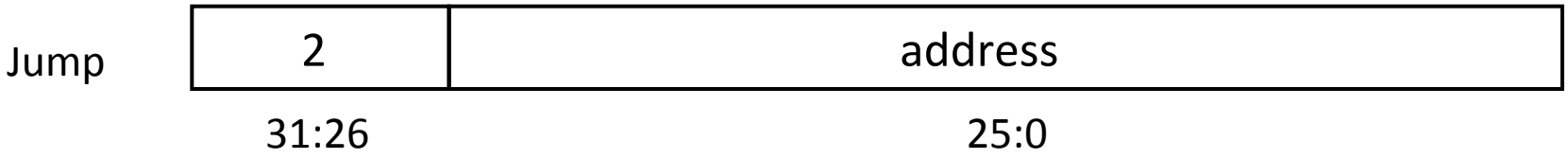
Load Instruction



Beq Instruction

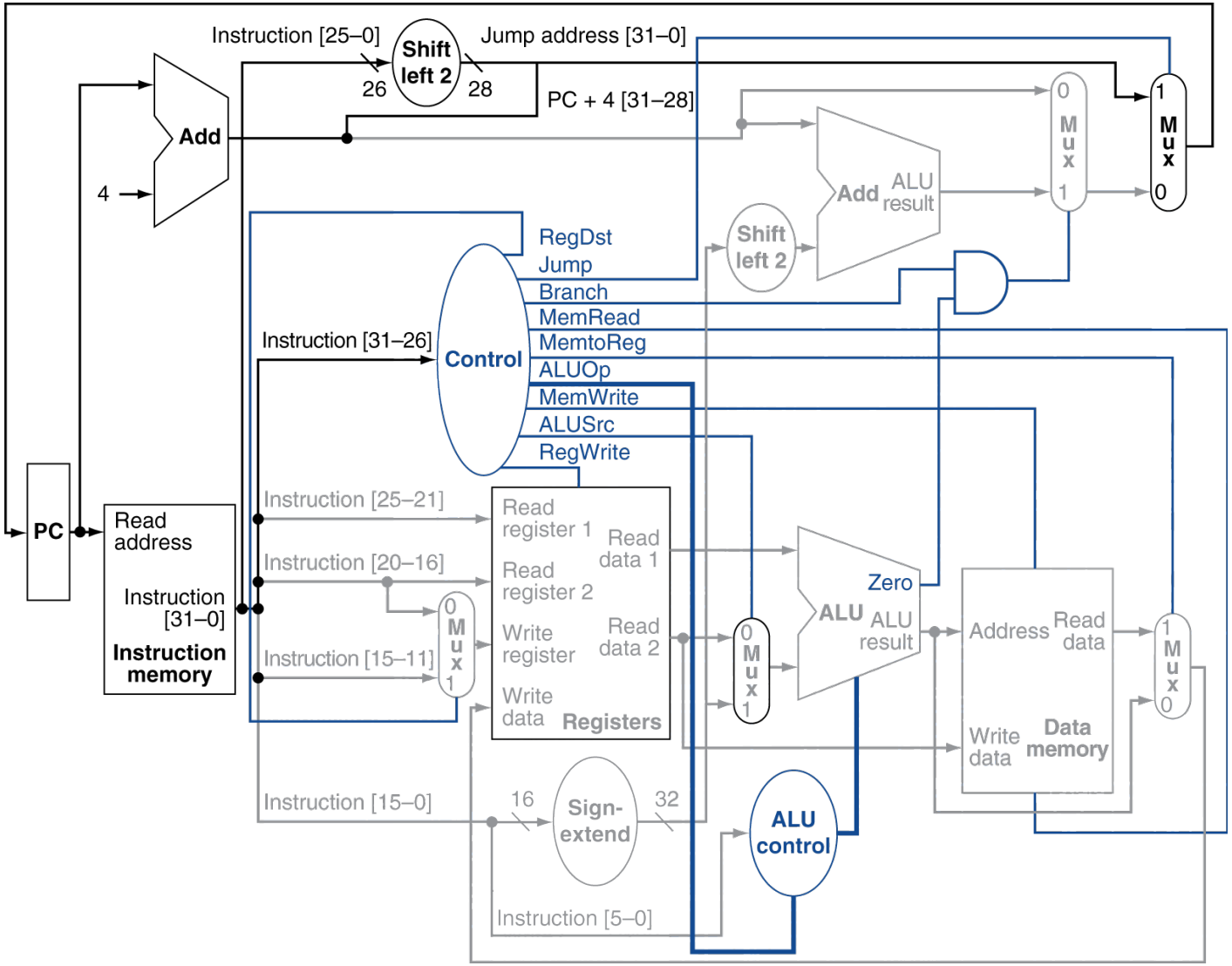


Jump Instruction



- Jump uses word address (instruction boundary)
- Sets PC = (PC & 0xF0000000) + 4*(26-bit jump address)
- Needs an extra control signal decoded from opcode

Datapath with Jump Added



Mapping Control to Gates

- Use of instruction subset simplifies control
- Only 9 control signals have to be generated

Signal name
RegDst
ALUSrc
MemtoReg
RegWrite
MemRead
MemWrite
Branch
ALUOp1
ALUOp0

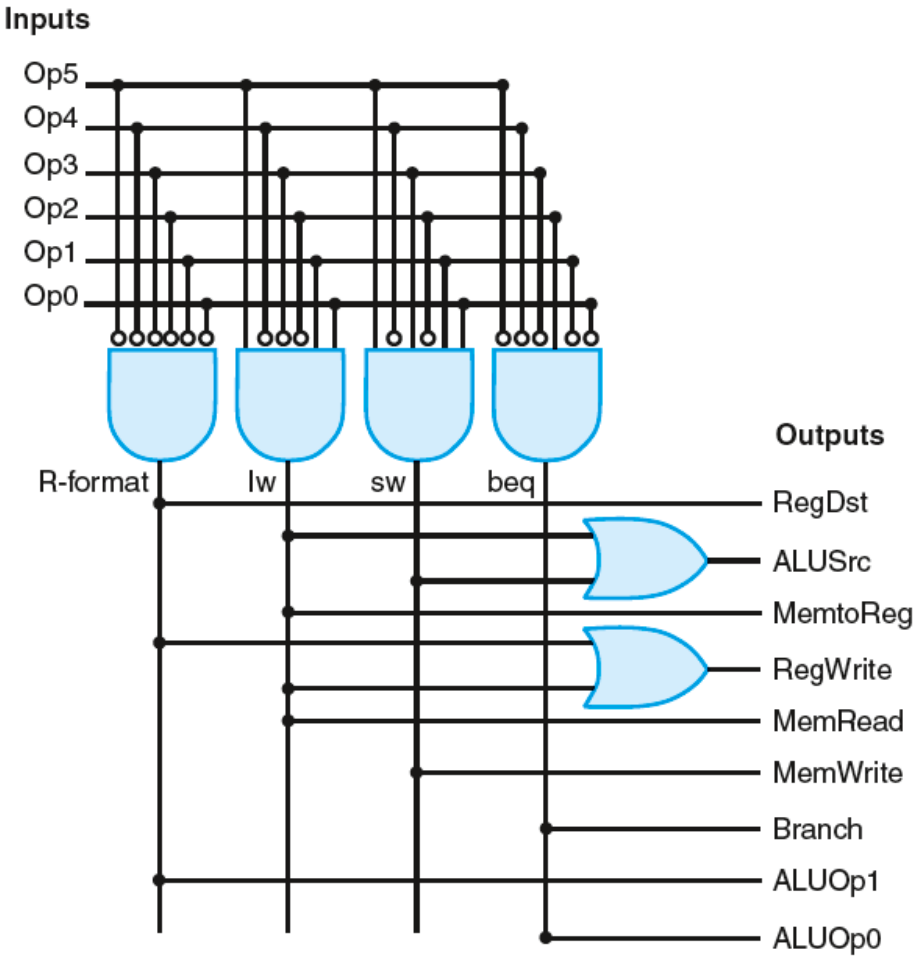
Mapping Control to Gates

- A truth table defines the outputs as function of opcode

Control	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Mapping Control to Gates

- Simple logic circuit derived from truth table



Performance Issues

- With this option, instruction executes in a single clock cycle
- Longest instruction 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

Datapath

- This module describes how instructions are executed using the MIPS datapath
- The datapath includes the ALU, control unit, register file and the pathways that connect the various components
- The ALU and control unit are implemented using the logic circuits described in the previous module

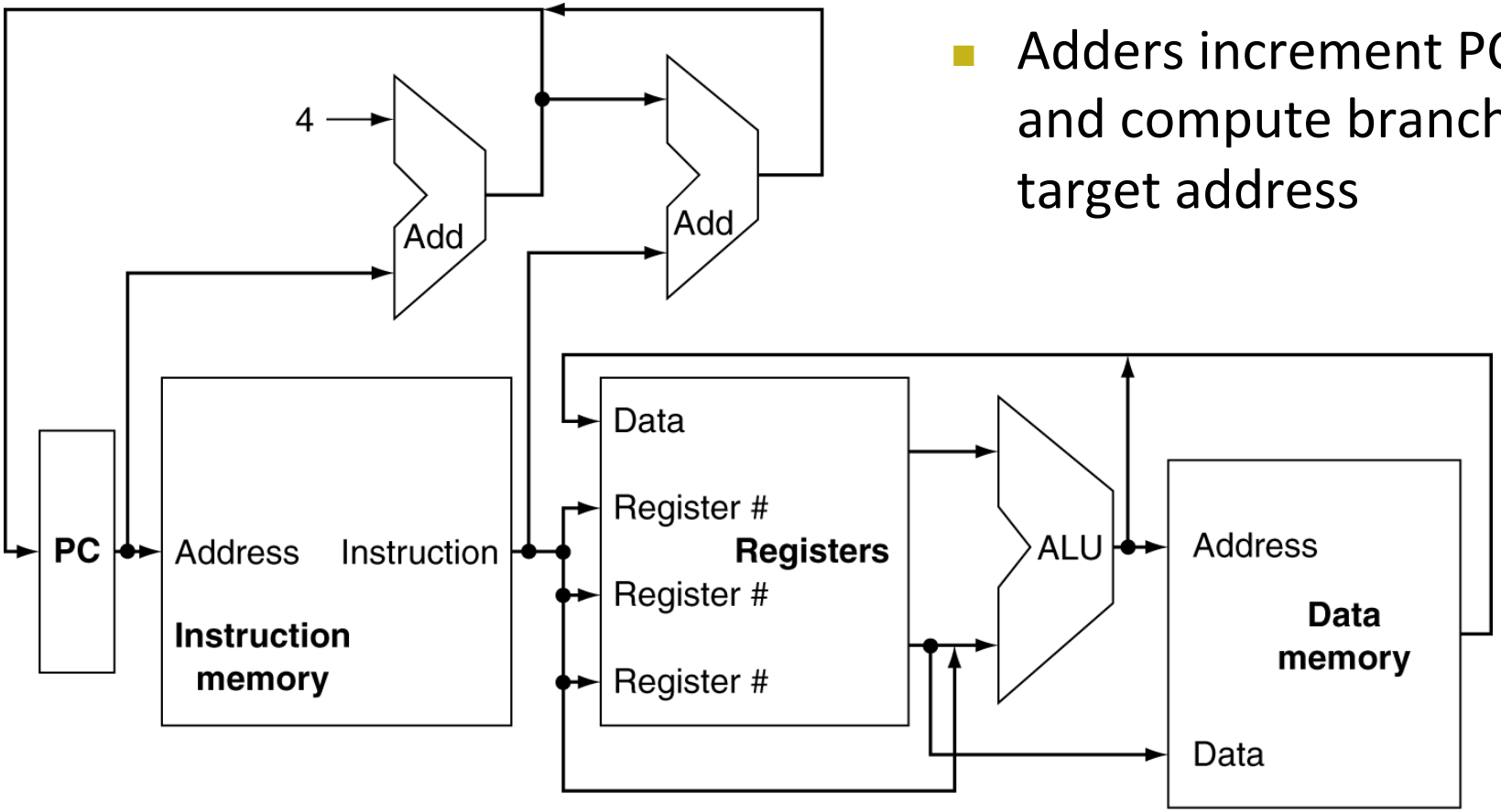
Datapath

- 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 single-cycle version
  - And later, a more realistic pipelined version
- A simple instruction subset will be used
  - 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 result
    - Compute memory address for load/store
    - Evaluate branch condition
  - Access data memory for load/store
  - PC ← target address or PC + 4

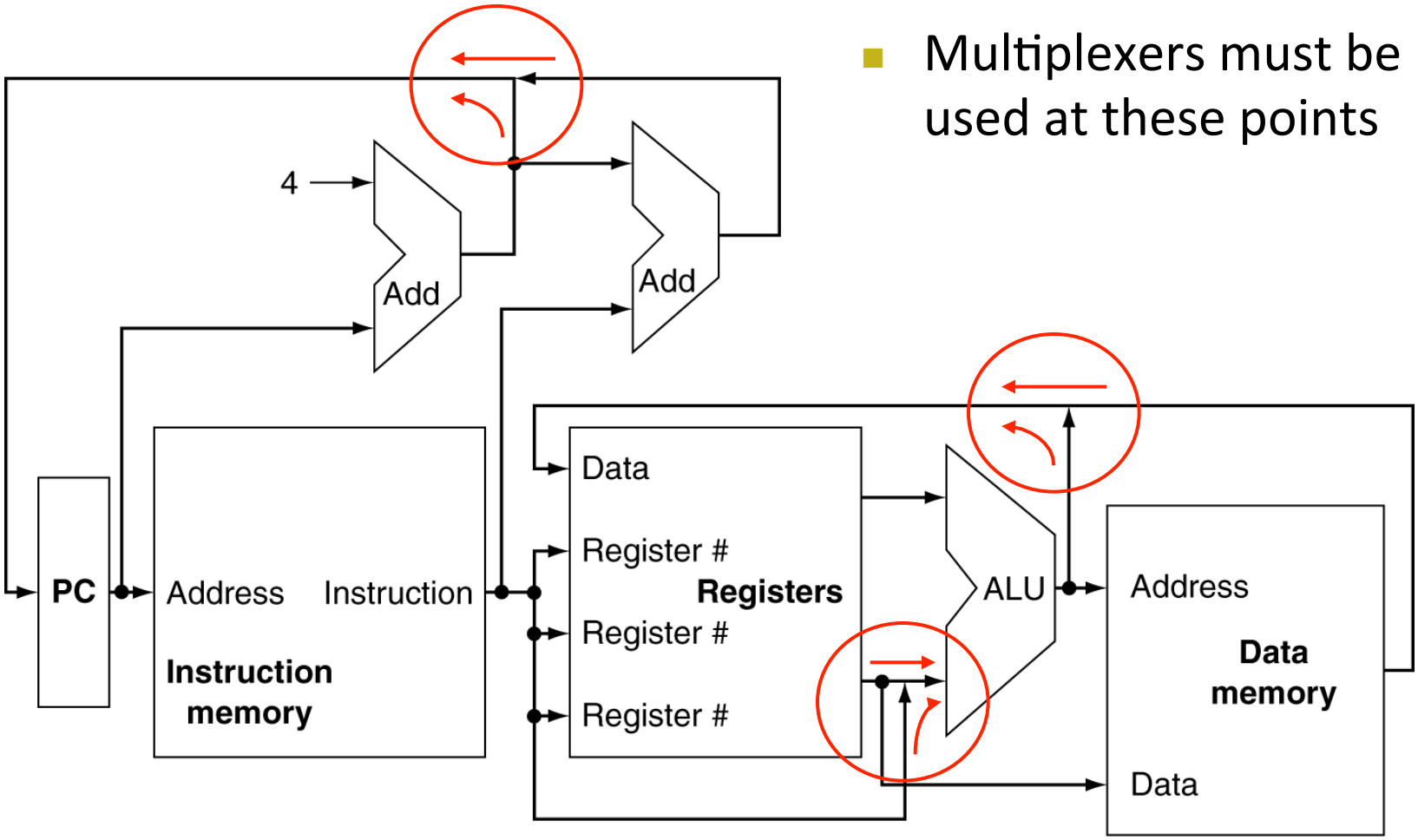
Datapath Overview



- Adders increment PC and compute branch target address

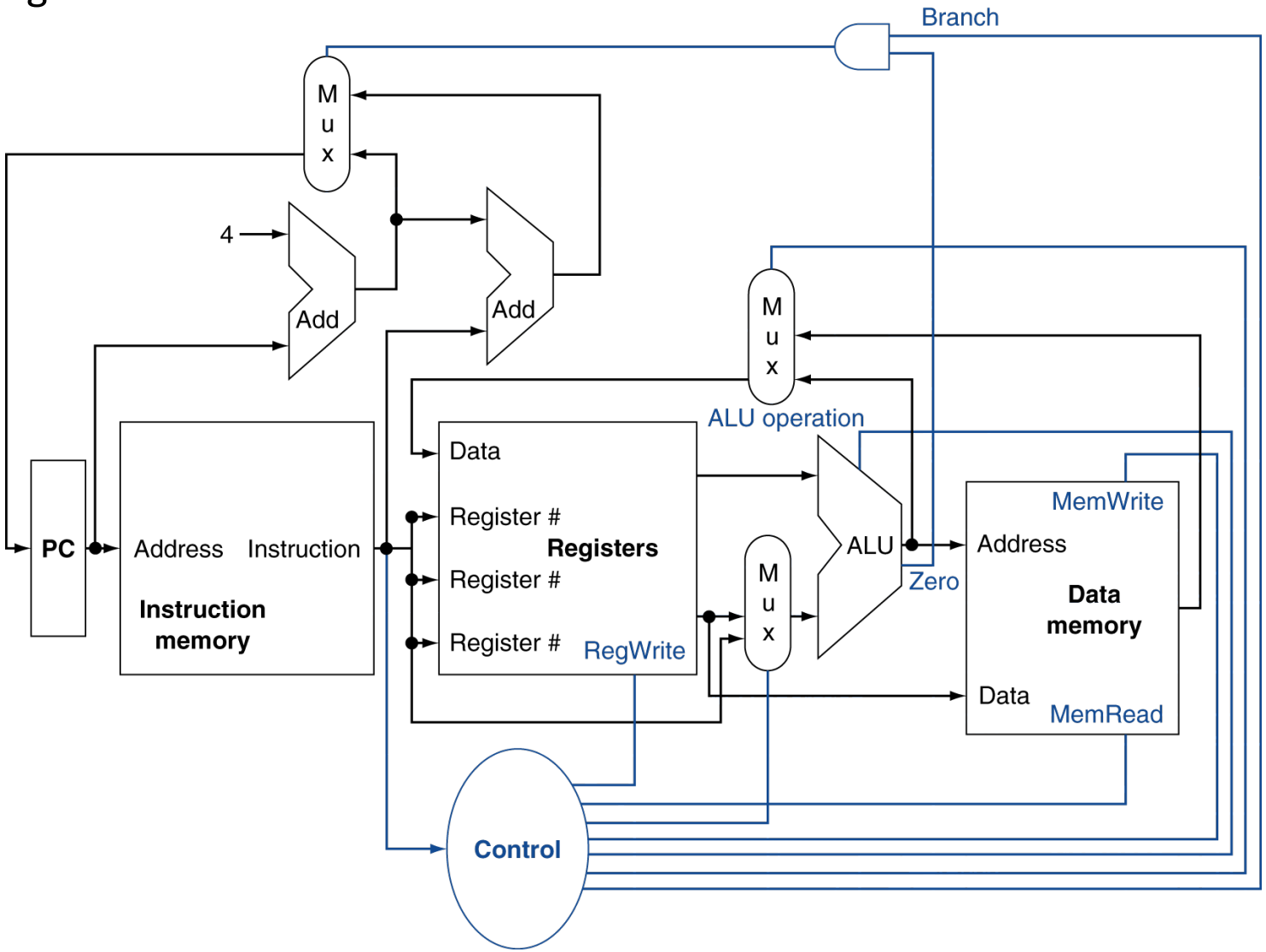
Datapath Overview

- Multiplexers must be used at these points



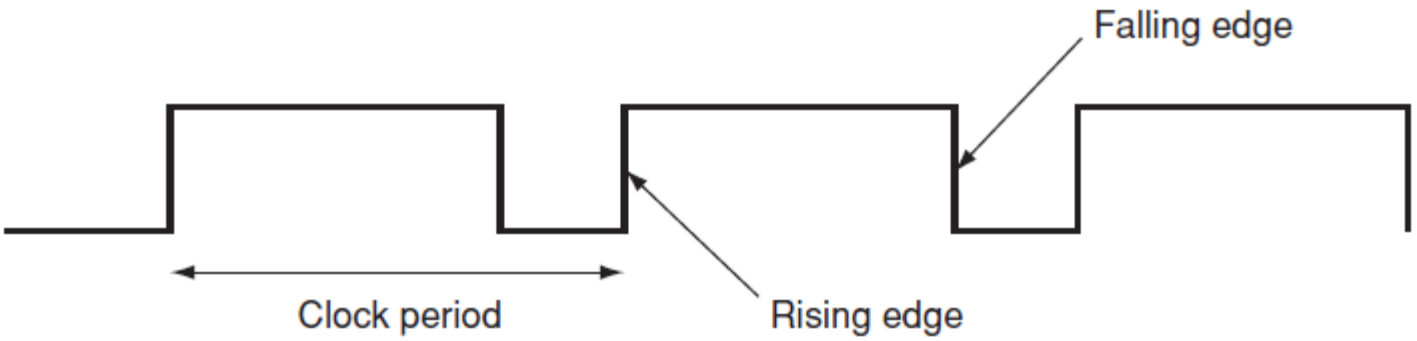
Datapath Control

Control signals determine what actions are taken



Datapath Timing

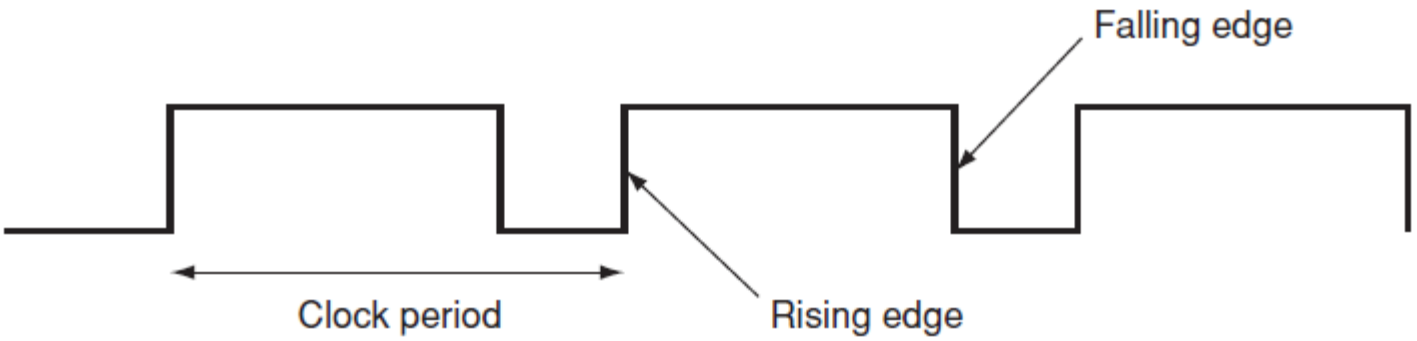
- Operations are synchronized to a clock
  - For example, when a register is written
  - Instructions complete at clock edges



- Clock signal oscillates between high and low values
- Clock period is one full clock cycle
- State changes only on clock edge (either rising of falling edge)

Datapath Timing

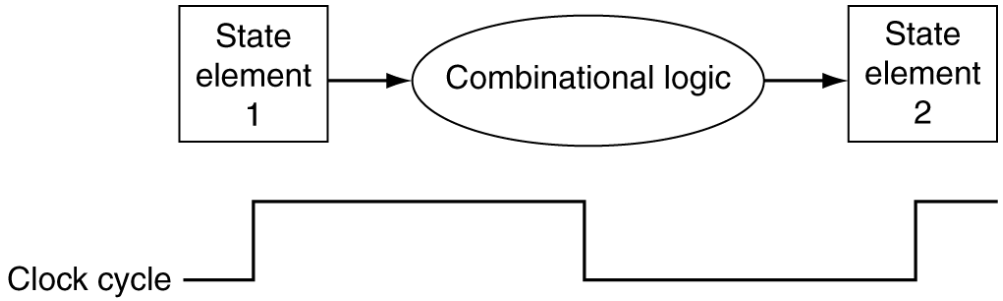
- Operations are synchronized to a clock
  - For example, when a register is written
  - Instructions complete at clock edges

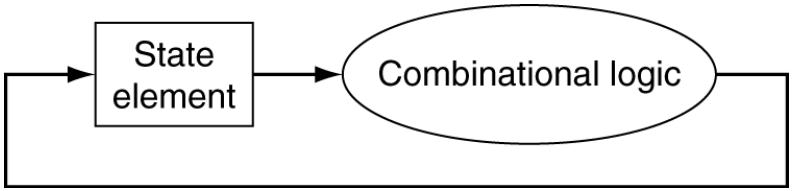


- Clock signal oscillates between high and low values
- Clock period is one full clock cycle
- State changes only on clock edge (either rising of falling edge)

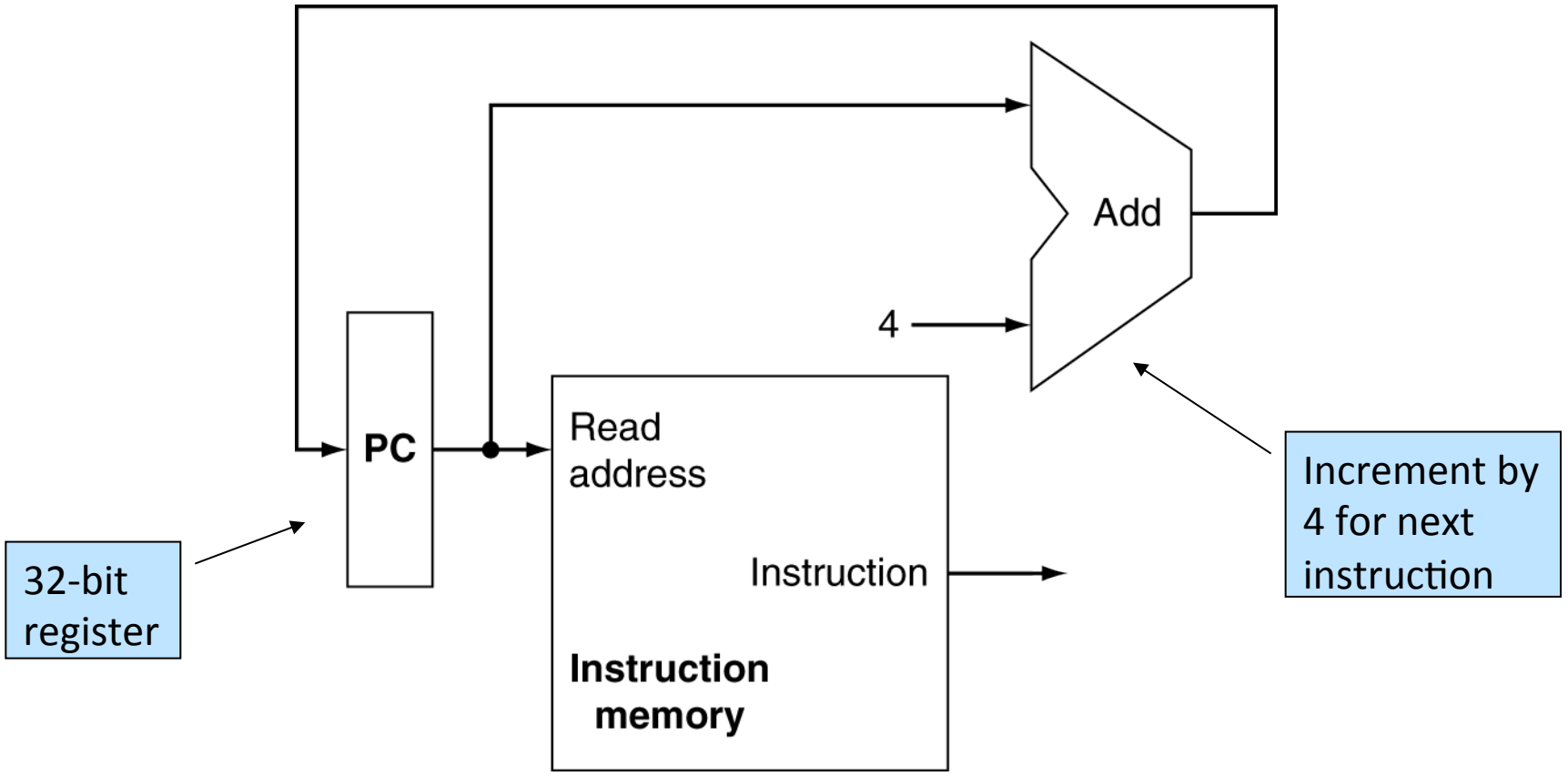
Datapath Timing

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Inputs come from state elements
  - Outputs go to state elements
  - Longest delay determines minimum clock period





Building a Datapath

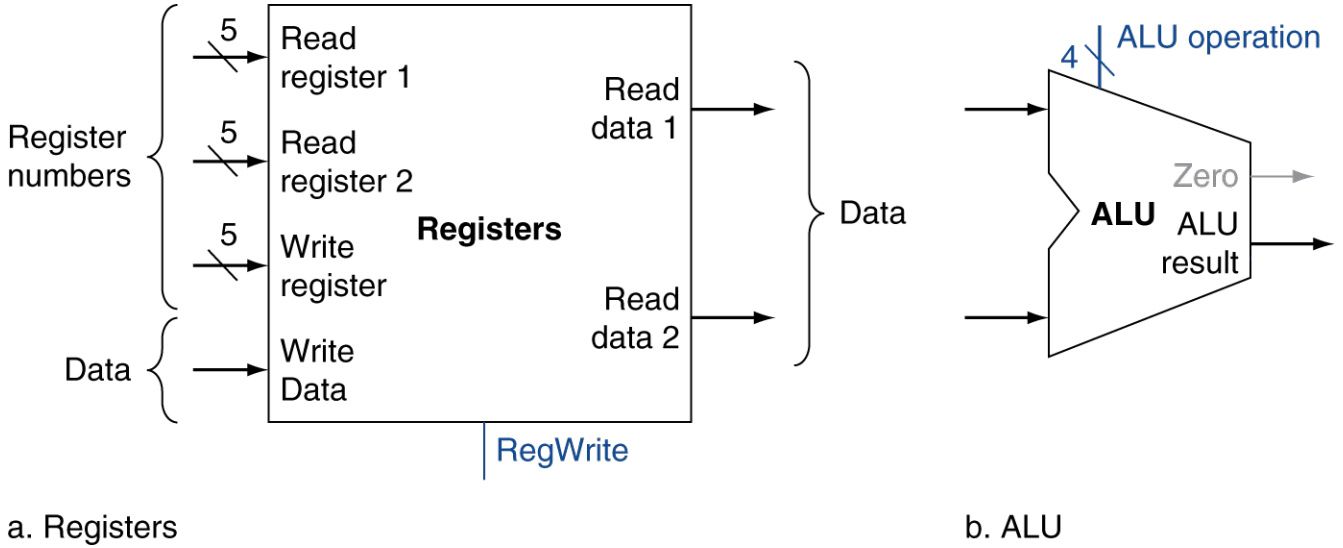


Performs Instruction Fetch

Building a Datapath

R-Format Instructions

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



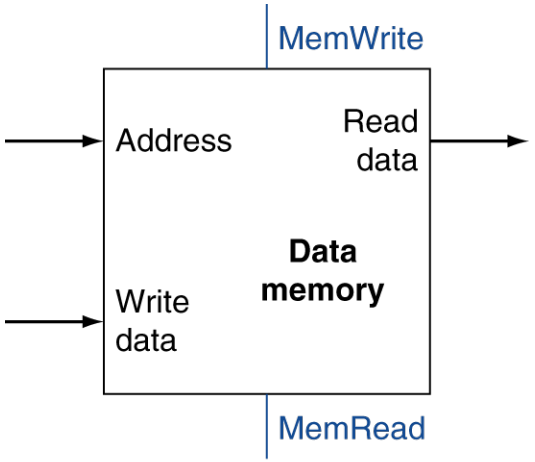
a. Registers

b. ALU

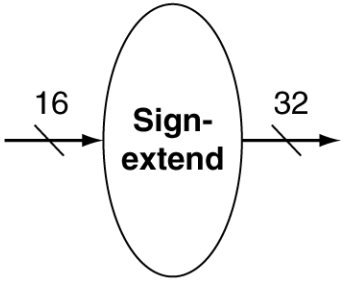
Building a Datapath

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

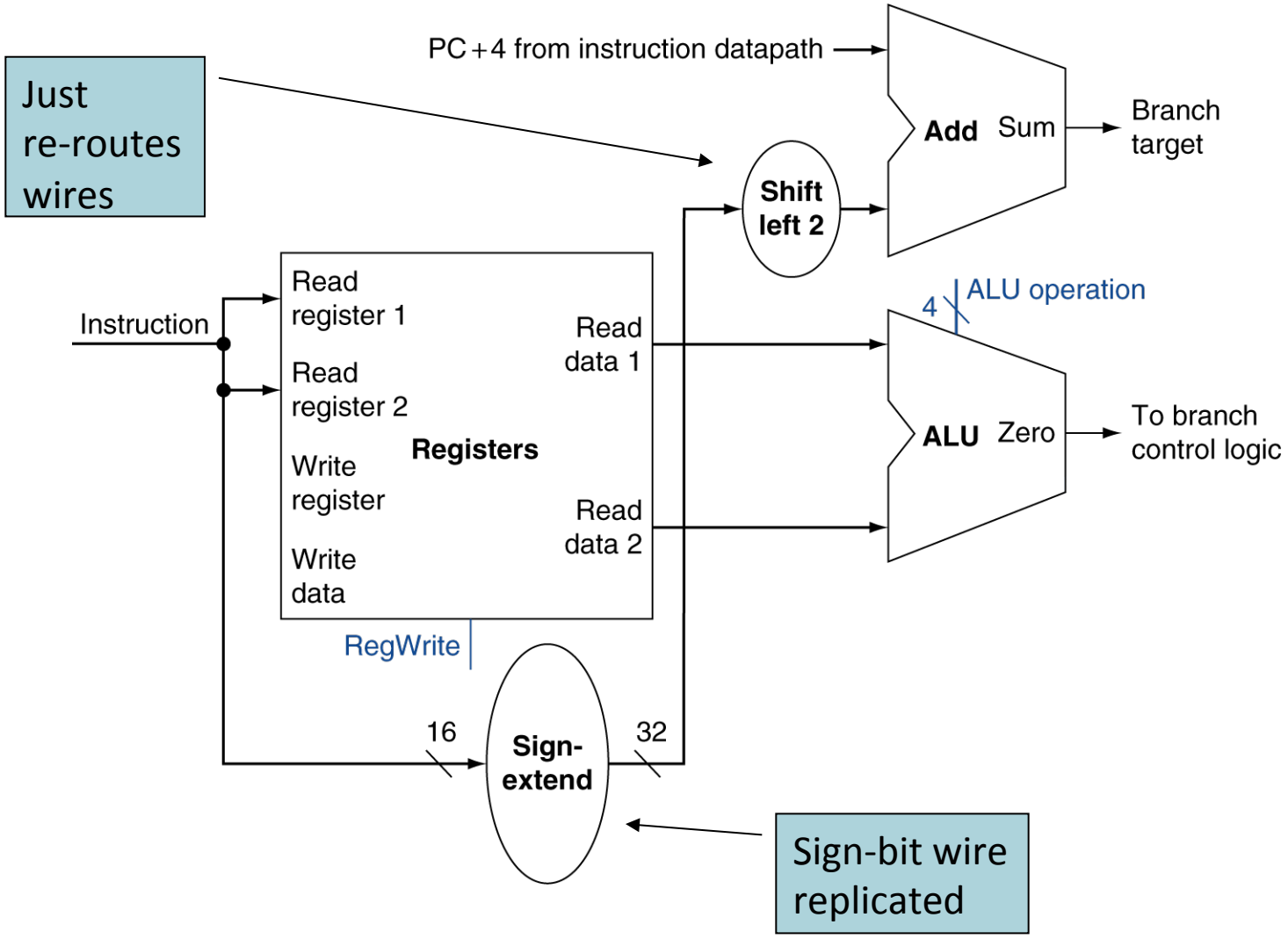


b. Sign extension unit

Branch Instructions

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

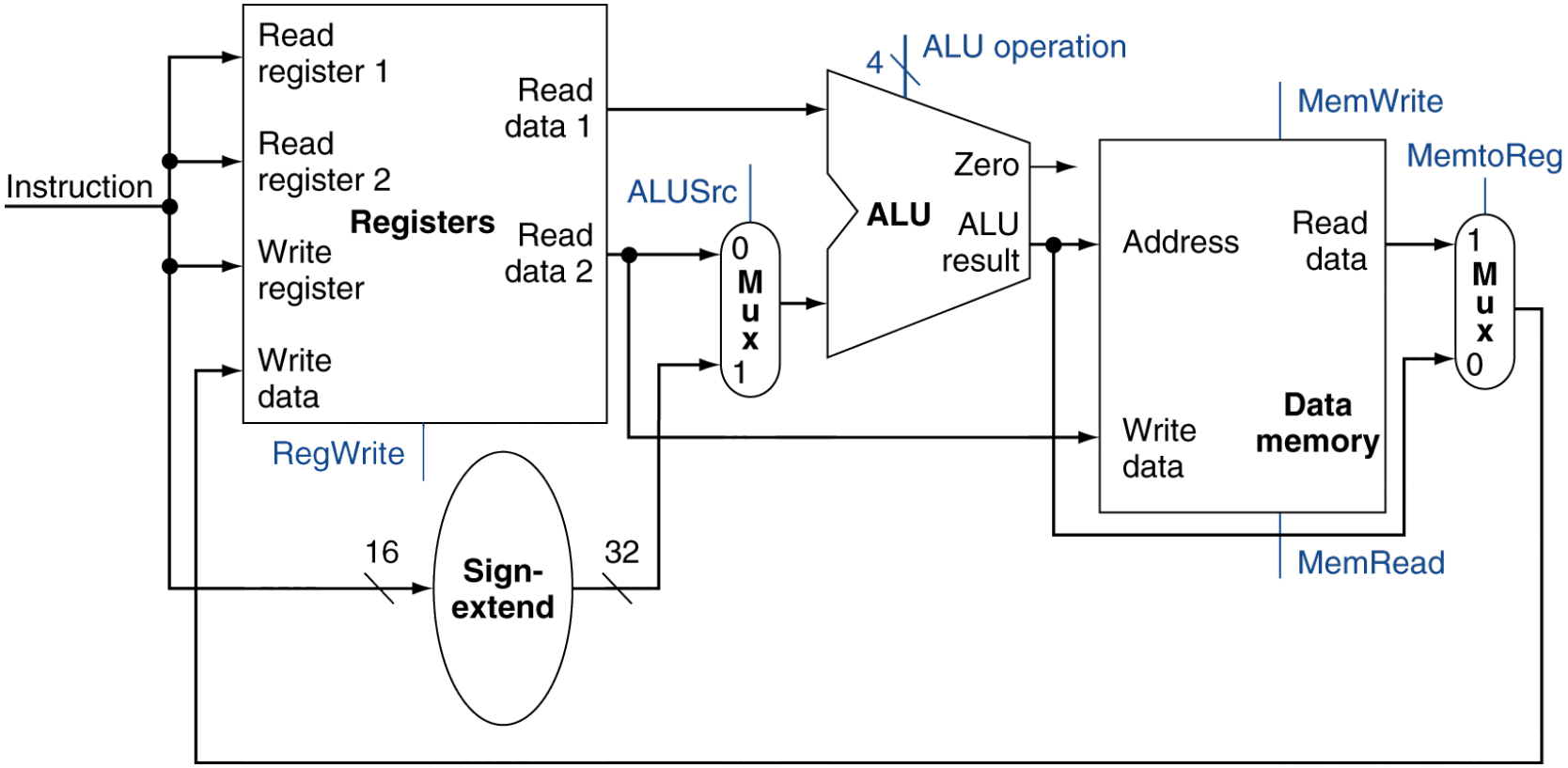
Building a Datapath



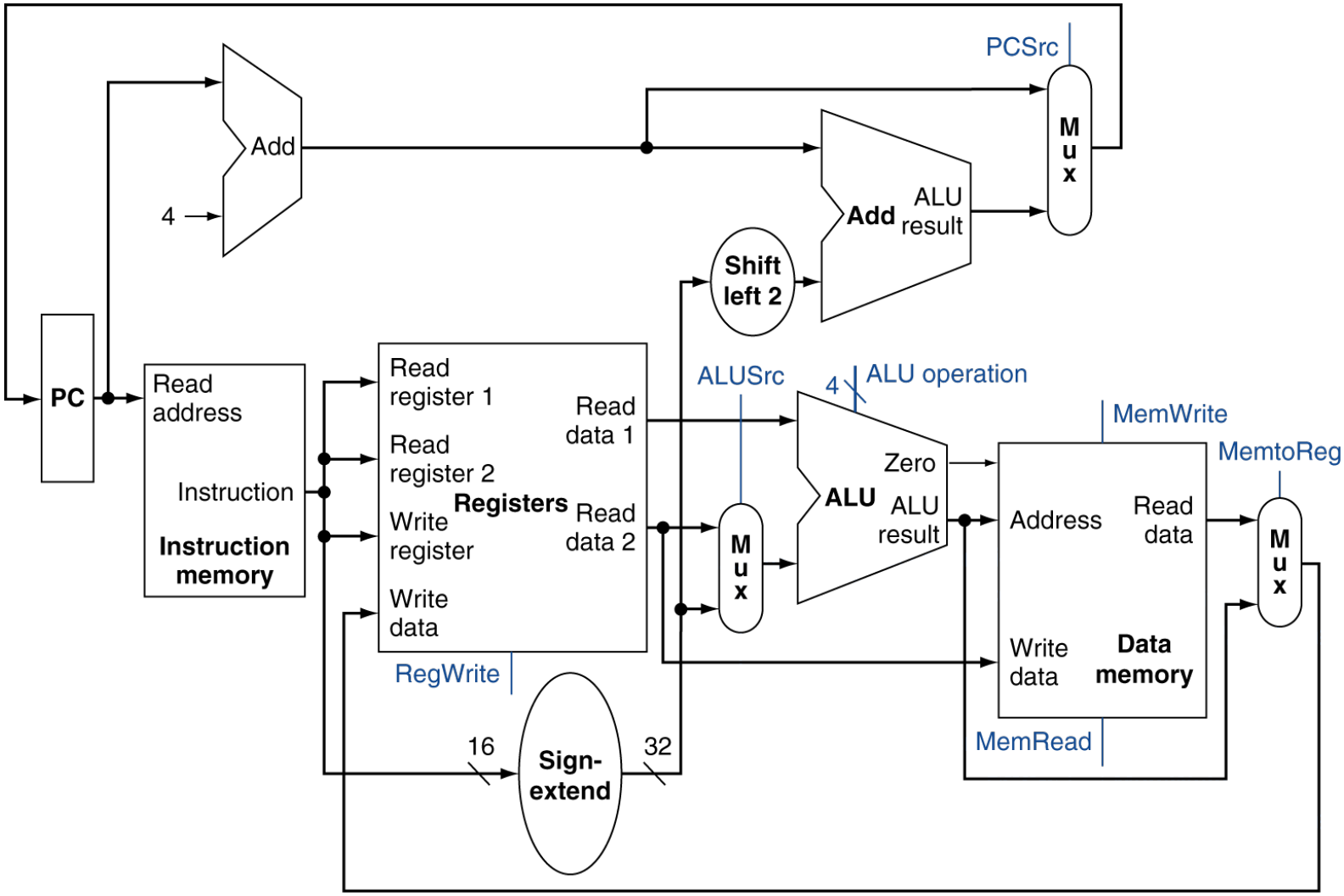
Single Cycle Execution

- One option is for the datapath to execute 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 (Harvard Architecture)
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



Full Datapath



Conclusion

- The datapath components have now been described
- Next, the ALU will be examined in more detail
- We will also see how the control unit produces the required signals