Chapter 4

Building the Processor

Part A

Introduction

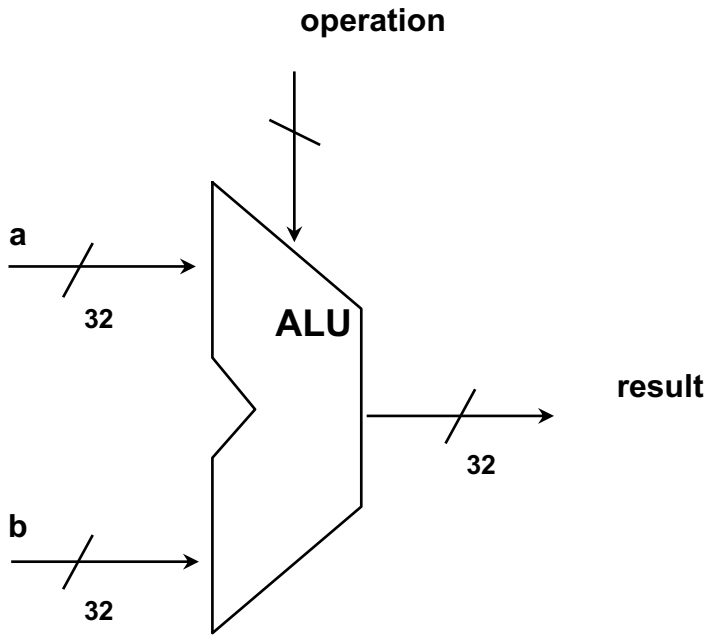
- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware

What will be covered ?

- Building the ALU
- Two MIPS implementations
  - A simplified version (datapath and control)
  - A more realistic pipelined version (datapath and control)
- Advanced Issues

Building the ALU (Chapter 4)

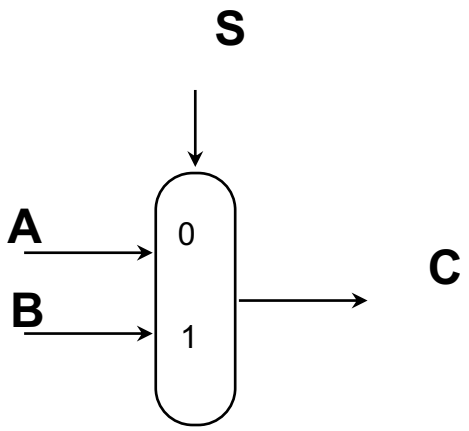
- Review Boolean Logic and build the ALU we'll need



- (Material from Appendix B - Reading Assignment)

Review: The Multiplexor

- Selects one of the inputs to be the output, based on a control input



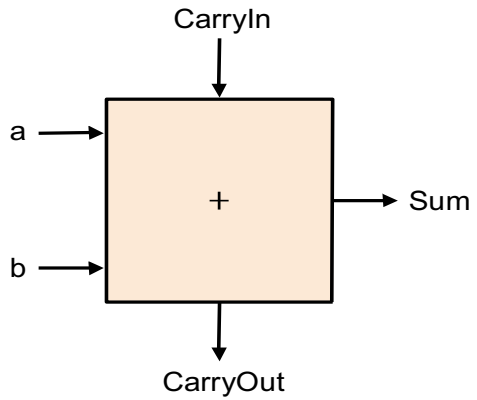
we call this a 2-input mux
even though it has 3 inputs!

- Lets build our ALU using a MUX:

Different Implementations

- Not easy to decide the "best" way to build something
  - *Don't want too many inputs to a single gate*
  - *Don't want to have to go through too many gates*

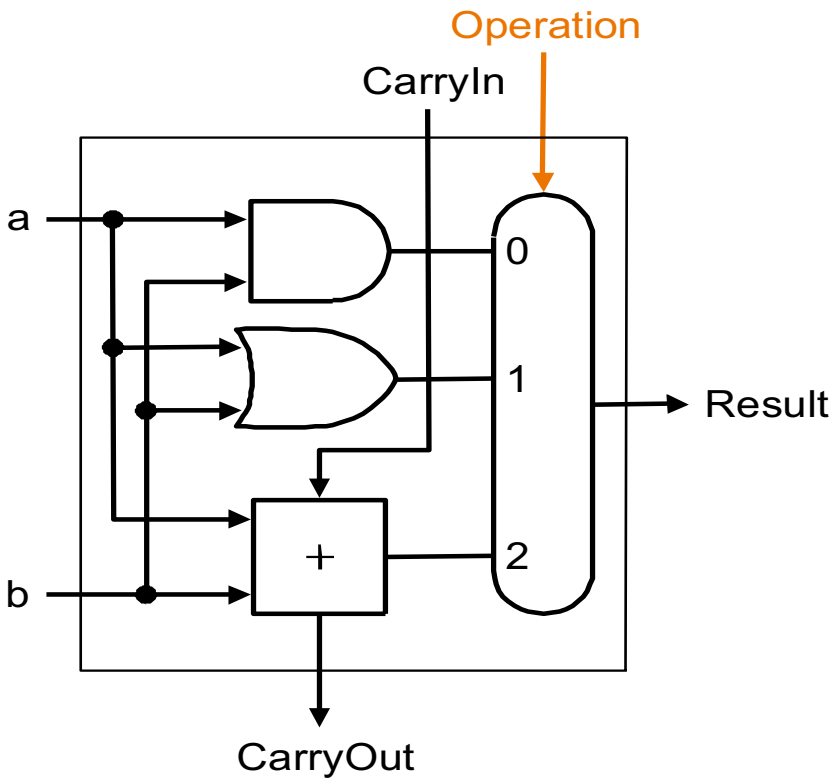
- Let's look at a 1-bit ALU for addition:

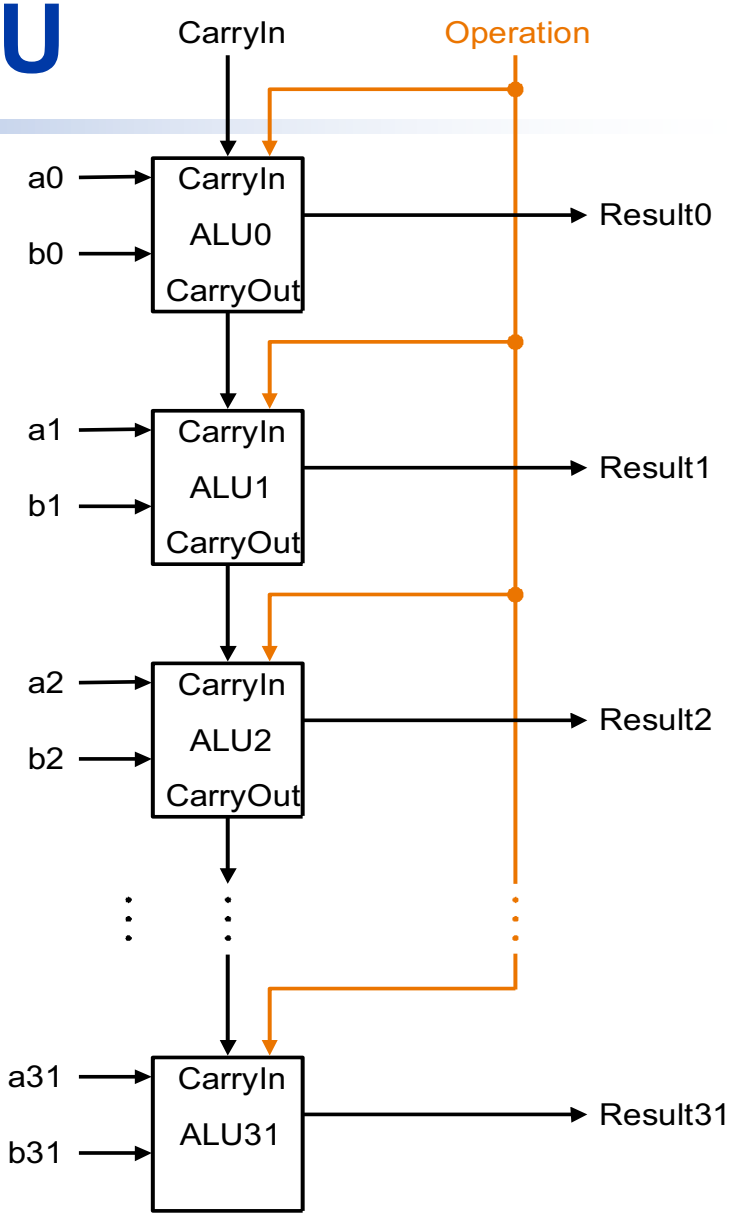


$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a\ xor\ b\ xor\ c_{in}$$

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

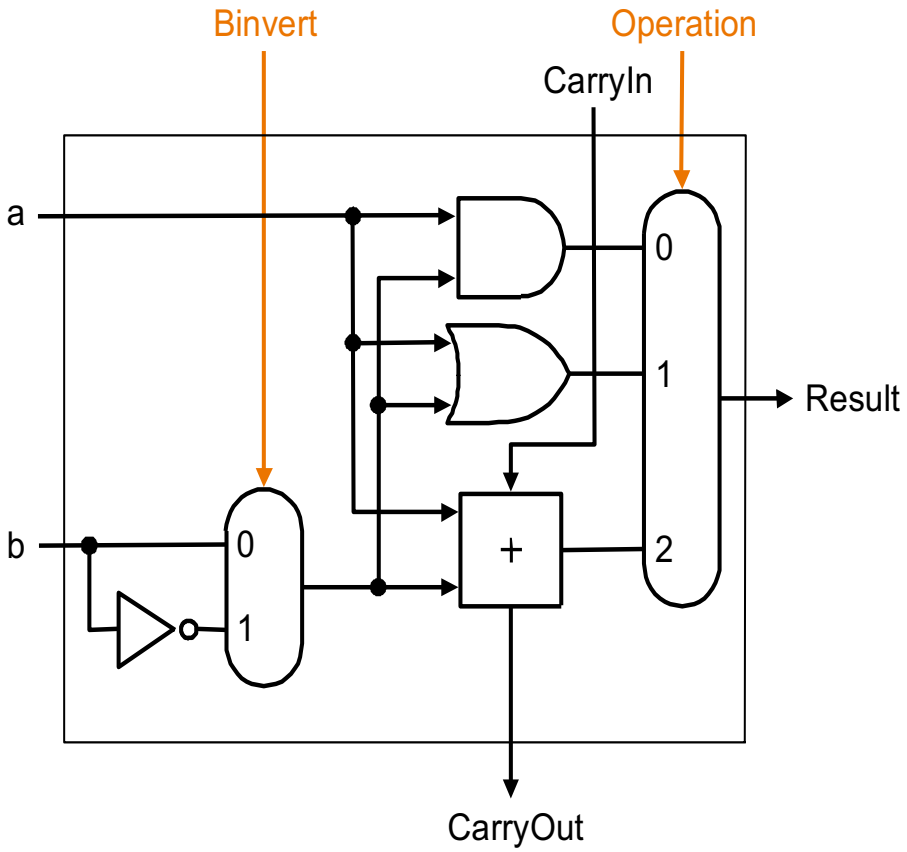
Building a 32 bit ALU





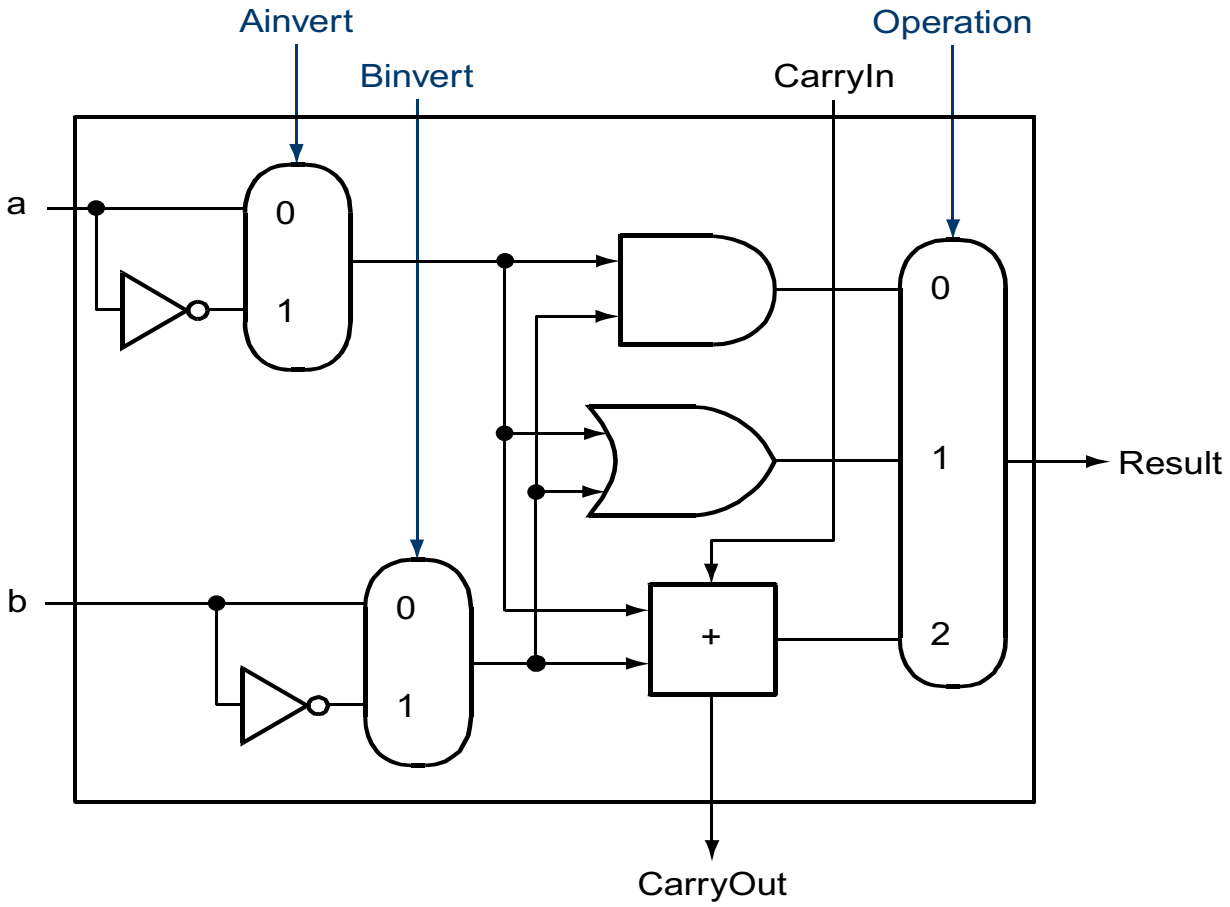
What about subtraction (a – b) ?

- Two's complement approach: just negate b and add.
- How do we negate?
- A very clever solution:



Adding a NOR function

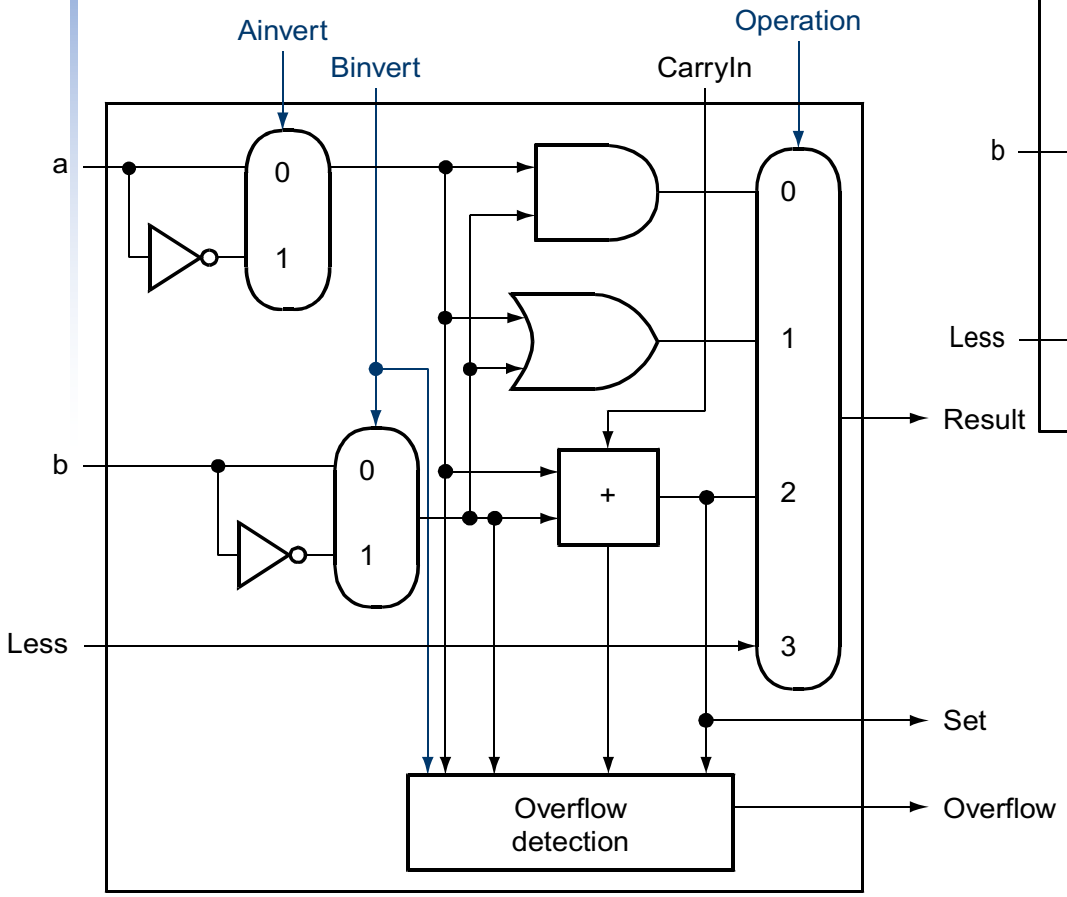
- Can also choose to invert a. How do we get “a NOR b” ?



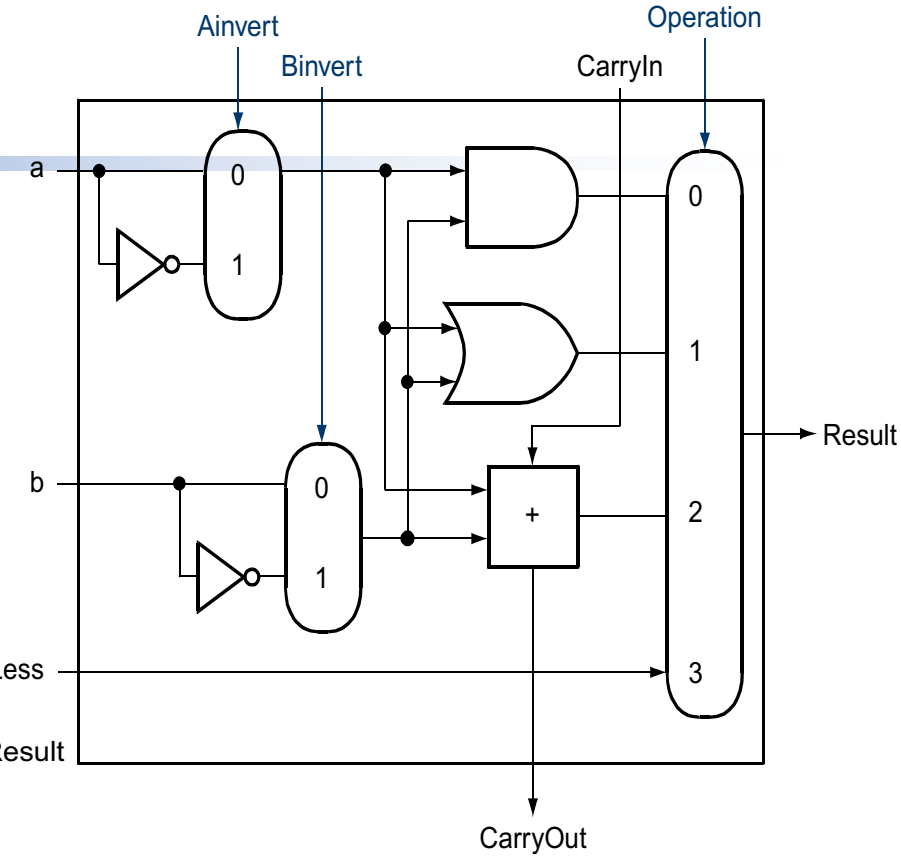
Tailoring the ALU to the MIPS

- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality (beq $t5, $t6, $t7)
  - use subtraction: (a-b) = 0 implies a = b

Supporting slt

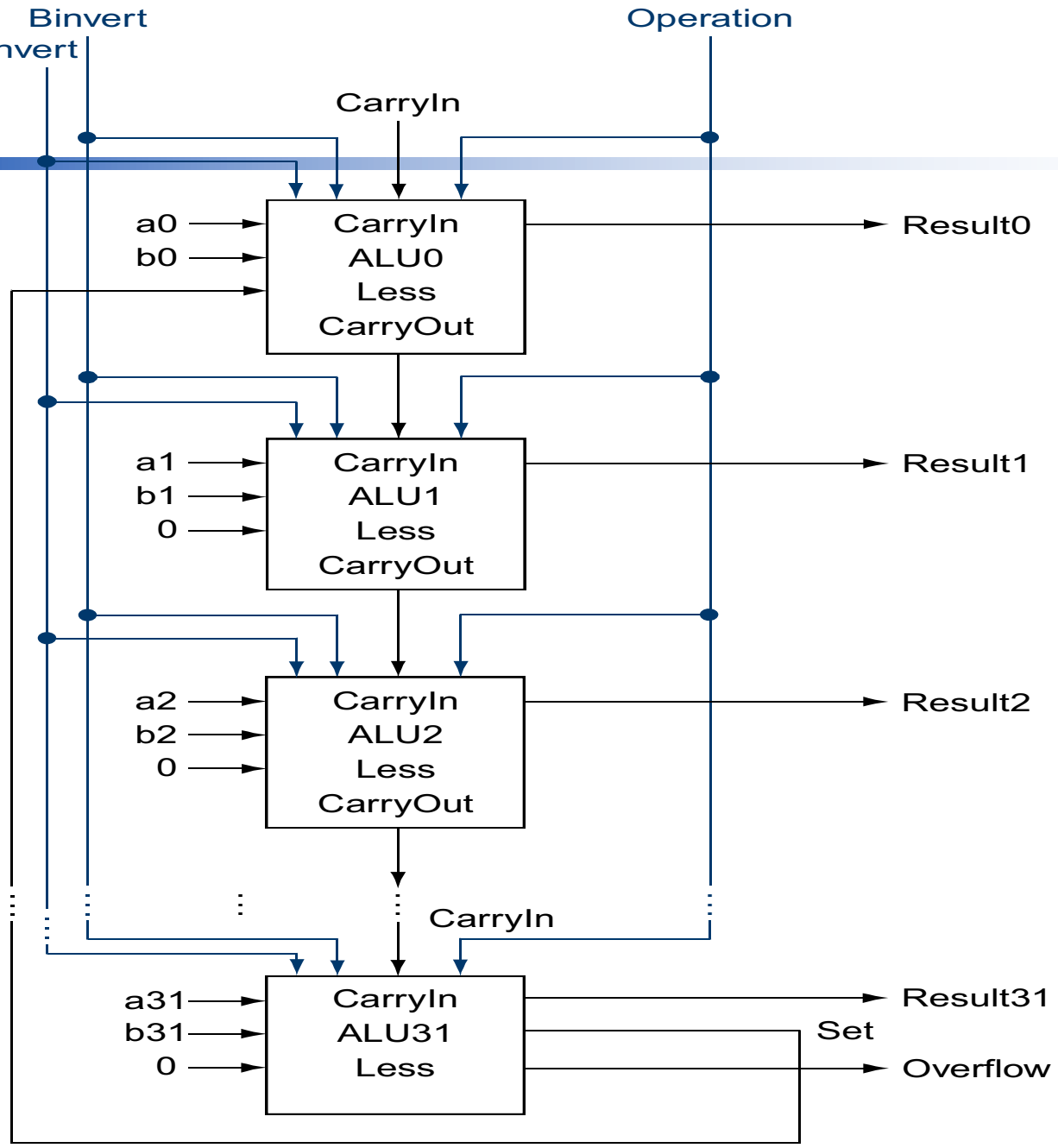


Use this ALU for most significant bit



Use this ALU for all other bits

slt

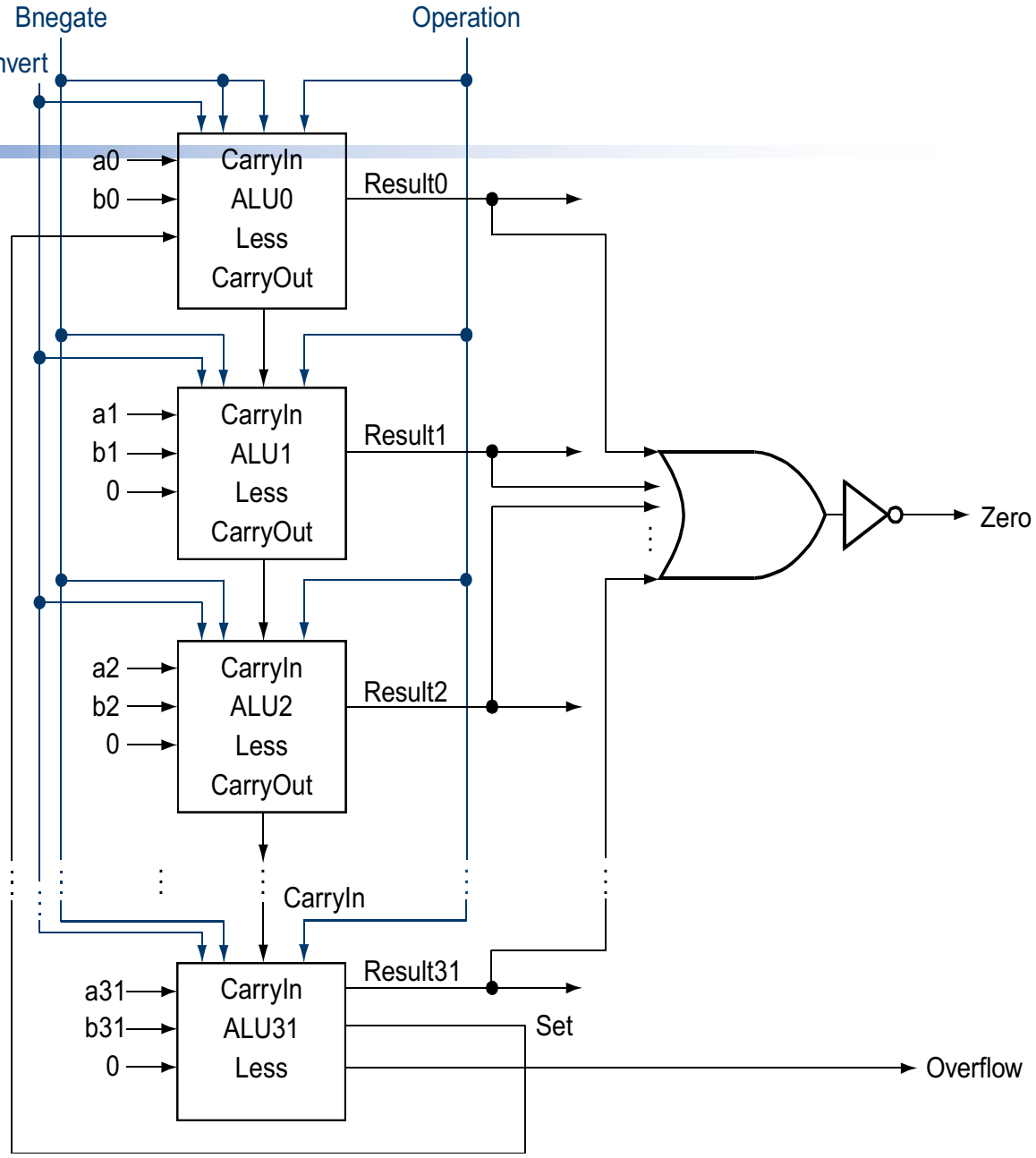

Binvert
Ainvert
Operation
CarryIn
a0
b0
CarryIn
ALU0
Less
CarryOut
Result0
a1
b1
0
CarryIn
ALU1
Less
CarryOut
Result1
a2
b2
0
CarryIn
ALU2
Less
CarryOut
Result2
CarryIn
a31
b31
0
CarryIn
ALU31
Less
Result31
Set
Overflow

Test for equality

- Notice control lines:

```
0000 = and
0001 = or
0010 = add
0110 = subtract
0111 = slt
1100 = NOR
```

- ***Note: zero is a 1 when the result is zero!***



Review on ALU Design

- **We can build an ALU to support the MIPS instruction set**
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- **Important points about hardware**
  - all of the gates are always working
  - the speed of a gate is affected by # of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")
- **Our primary focus:** comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)

The Processor: Datapath & Control

- We're ready to look at an implementation of the MIPS

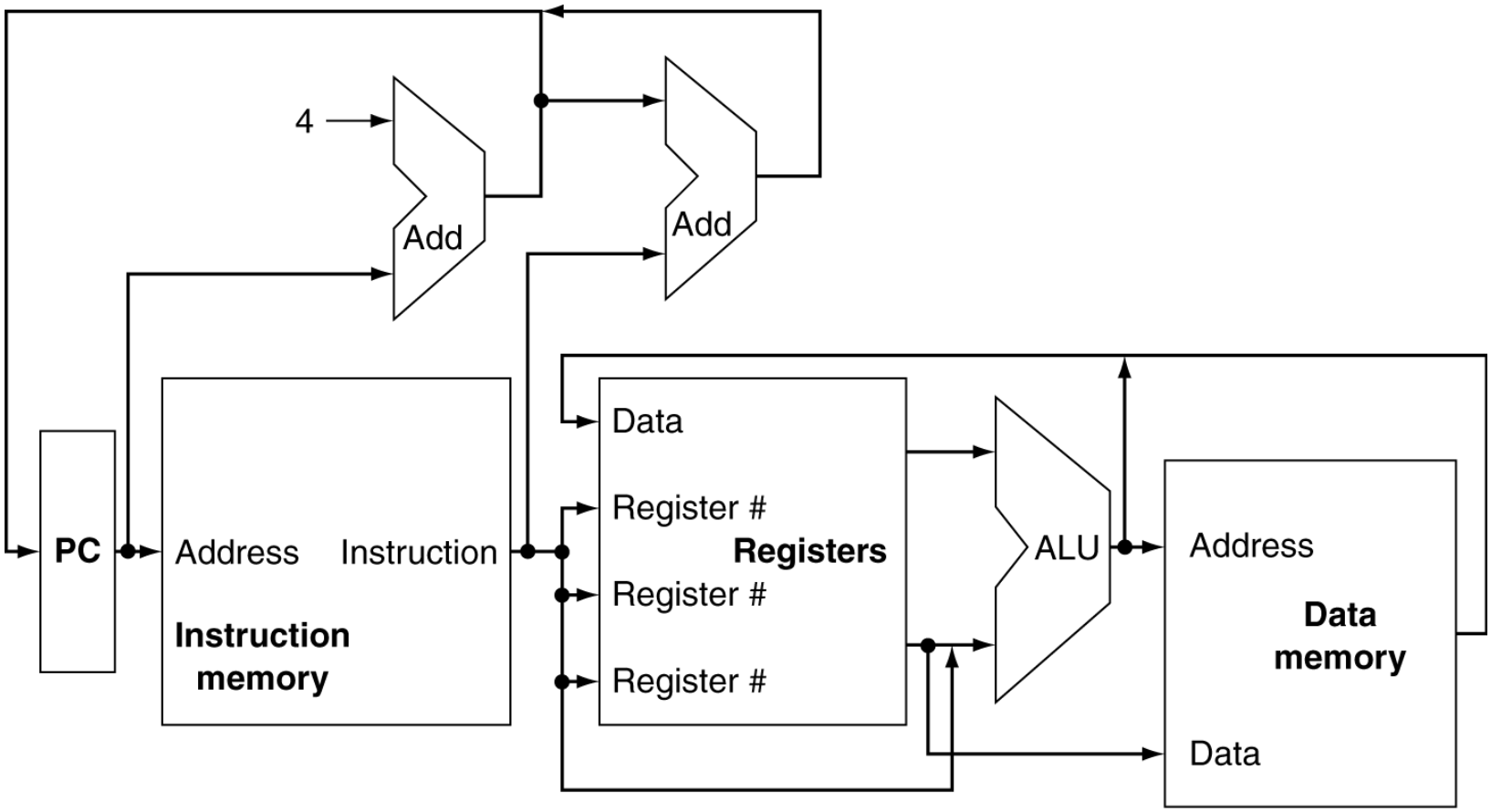
- Simplified to contain only:
  - memory-reference instructions: **`lw, sw`**
  - arithmetic-logical instructions: **`add,sub,and,or,slt`**
  - control flow instructions: **`beq, j`**

- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version

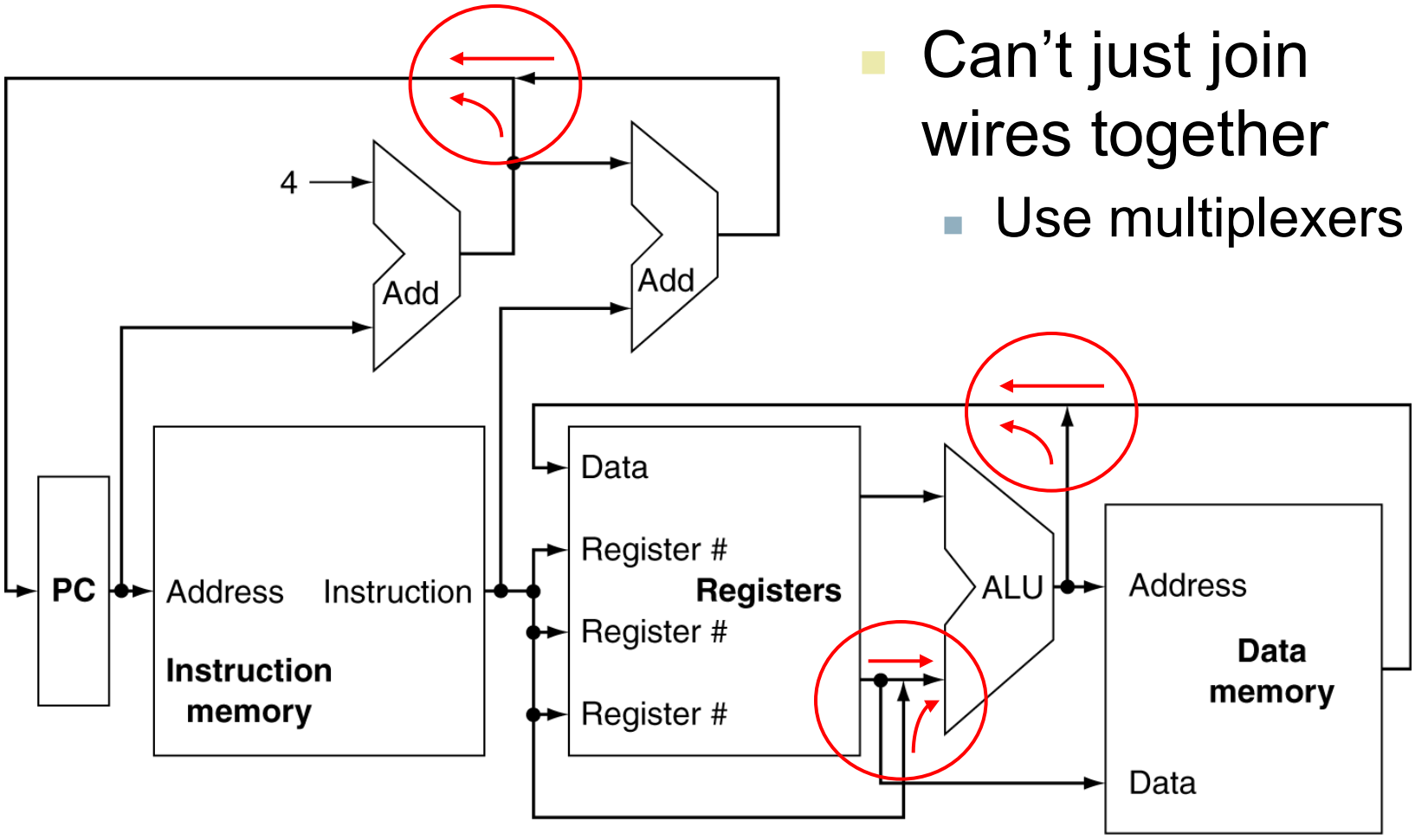
Generic Implementation

- use PC to supply instruction address
- get the instruction from memory
- read registers
- use the instruction to decide exactly what to do
- Depending on instruction class, all instructions use the ALU after reading the registers
  - Arithmetic result
  - Memory address for load/store
  - Branch (comparison)
- Access data memory for load/store
- Arithmetic-logical: write data from ALU to register
- PC ← target address **or** PC + 4

CPU Overview

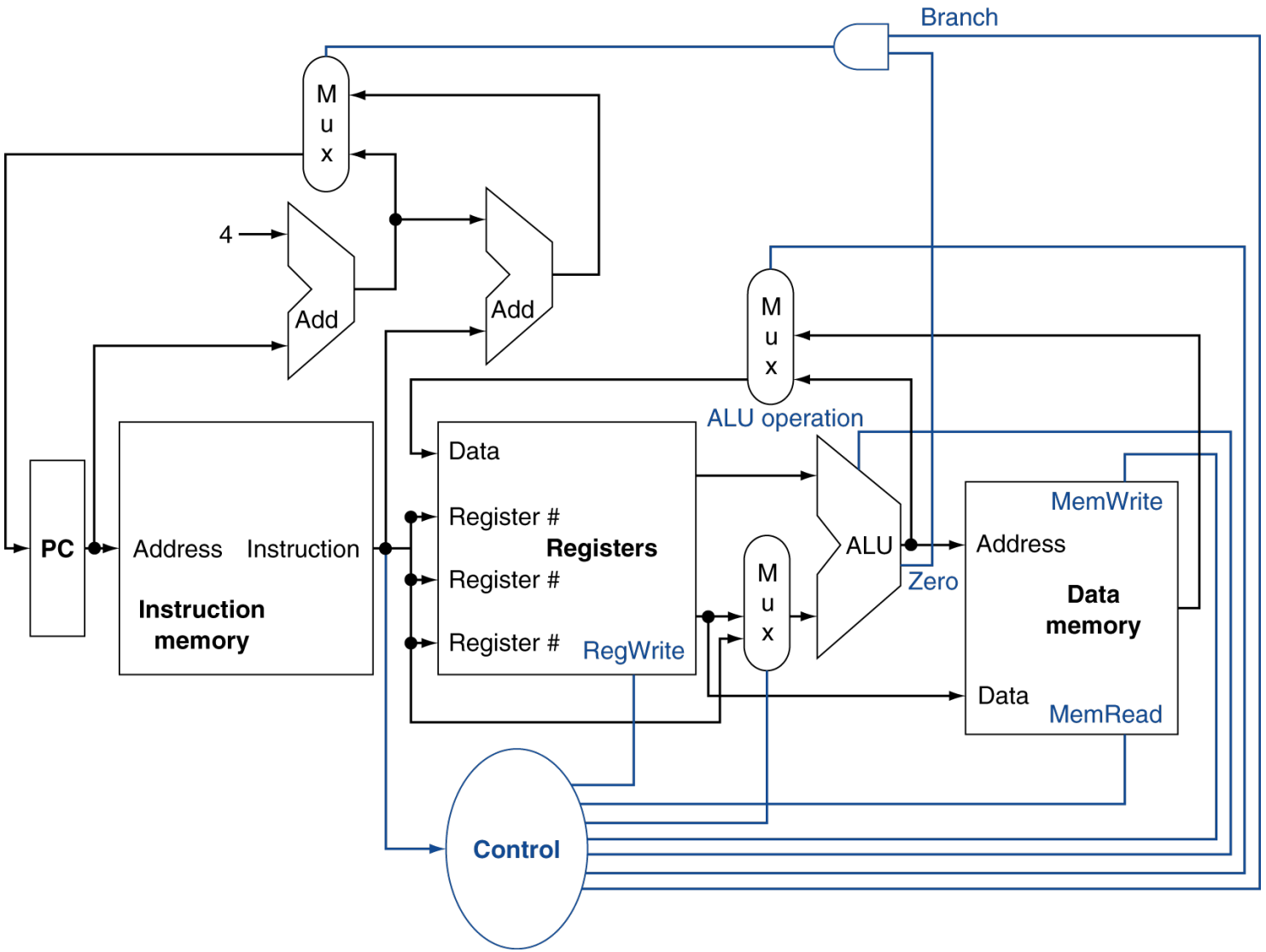


Multiplexers



- Can't just join wires together
  - Use multiplexers

Control

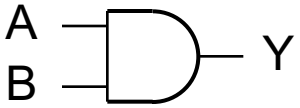

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Logic Design Basics

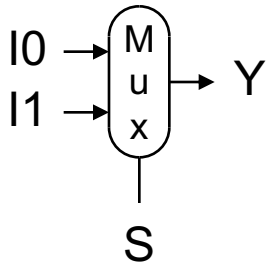
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

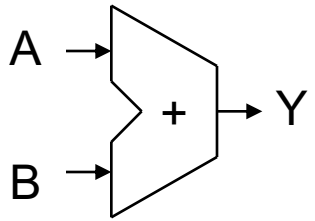
- AND-gate
  - Y = A & B



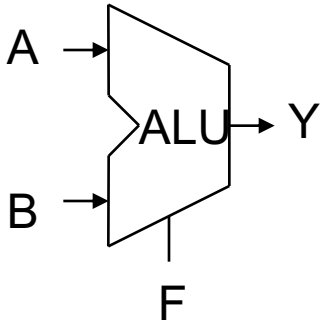
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

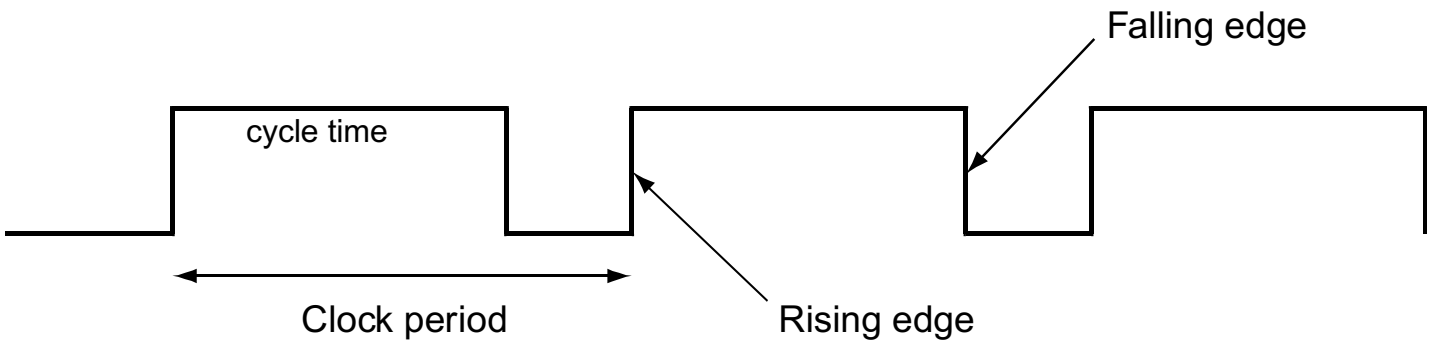


- Arithmetic/Logic Unit
  - Y = F(A, B)



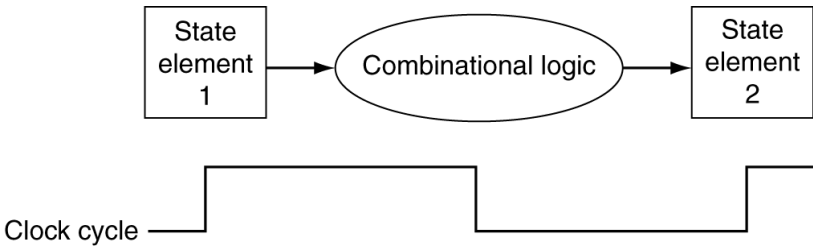
Sequential Elements

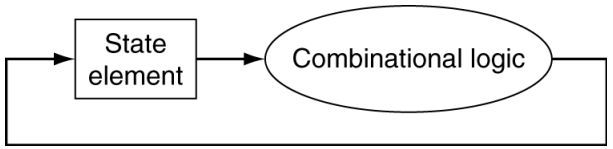
- Unclocked vs. Clocked
- Clocks used in synchronous logic
  - when should an element that contains state be updated?



Clocking Methodology

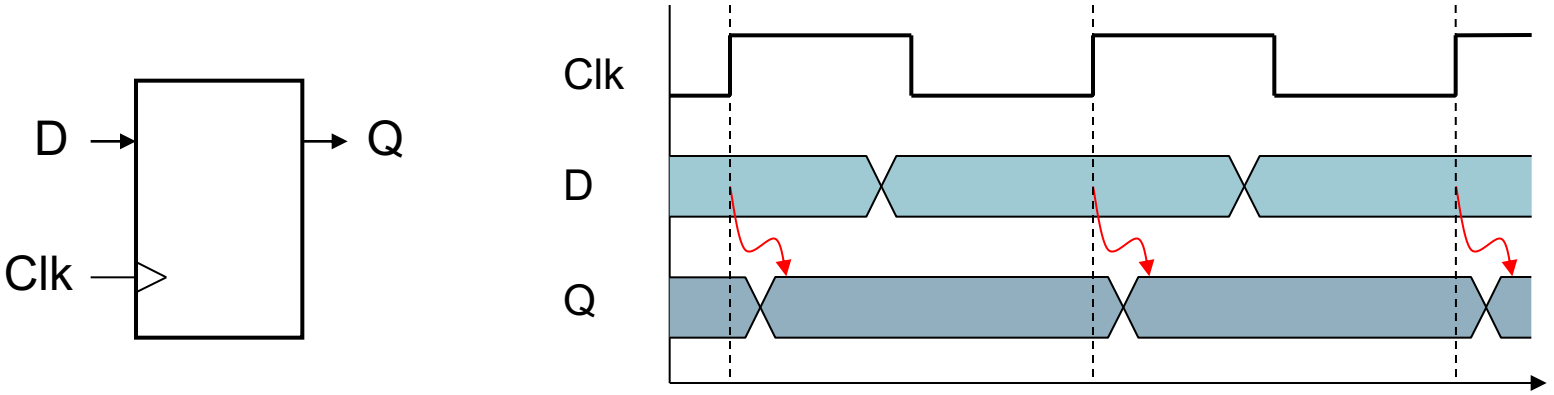
- A clocking methodology defines when signals can be read and written
- Edge-triggered clocking = all state changes occur on a clock edge
  - Positive edge-triggered (change on rising clock edge)
- All signals propagate from “State Element 1" through “combinational logic” and to “State Element 2” in the time of one clock cycle
- An edge-triggered methodology allows us to *read* the content of a register, *send* the value through some combinational logic, and *write* that register in the same clock cycle.





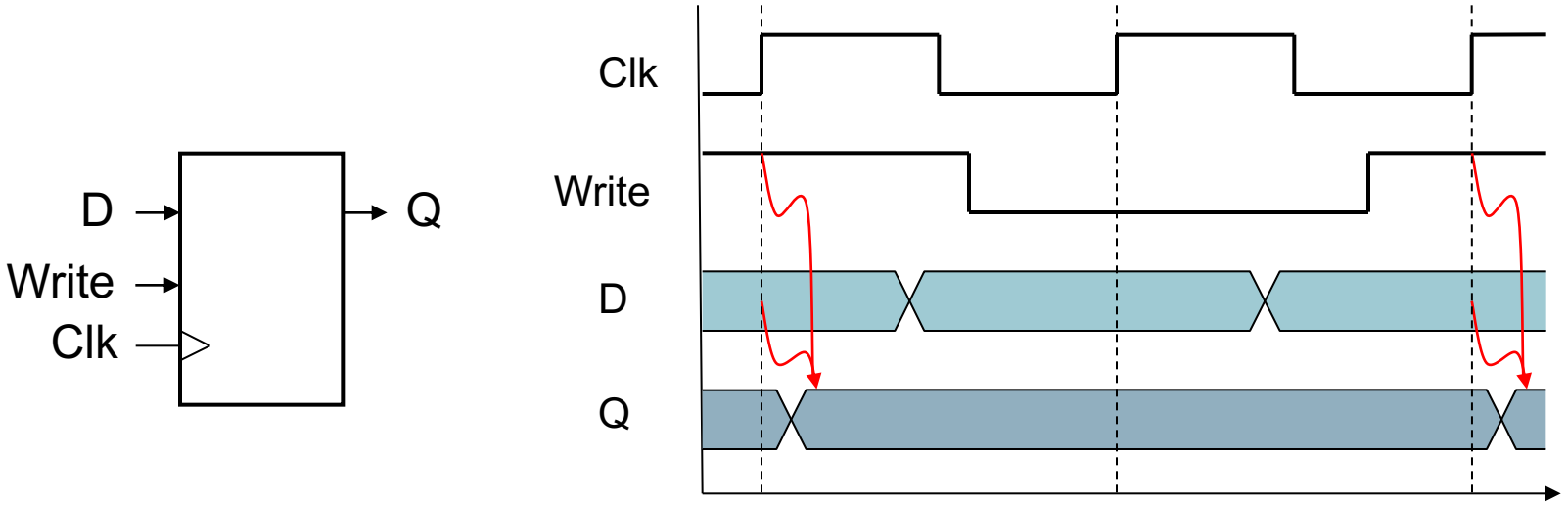
Sequential Elements

- 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



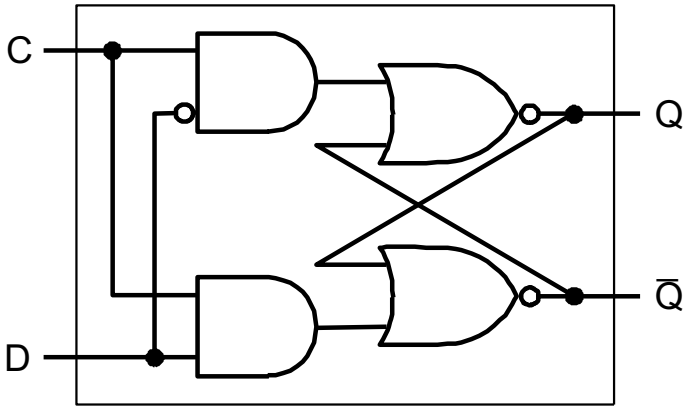
Latches and Flip-flops

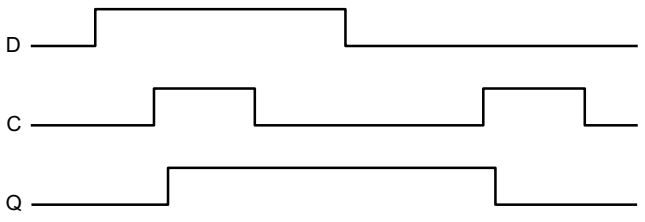
- Output is equal to the stored value inside the element
- Change of state (value) is based on the clock
  - Latches: whenever the inputs change, and the clock is asserted (assert = signal is logically high or true)
  - Flip-flop: state changes only on a clock edge (edge-triggered methodology)

- Do not show a write control signal when a state element is written on every clock edge.
  - If it is not updated on every clock, than a separate control signal

D-latch

- Two inputs:
  - the data value to be stored (D)
  - the clock signal (C) indicating when to read & store D
- Two outputs:
  - the value of the internal state (Q) and it's complement

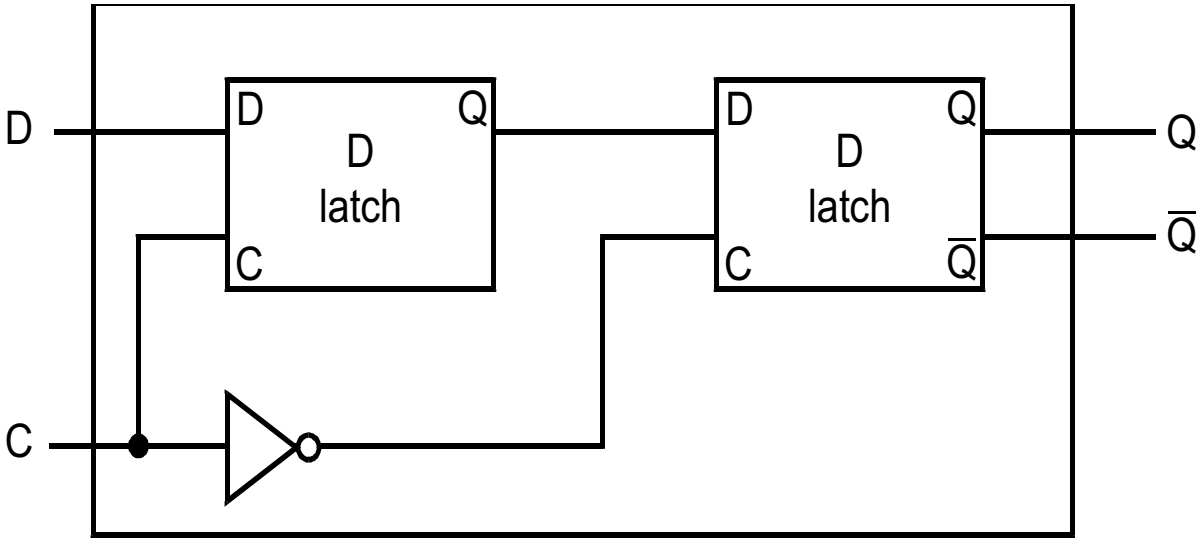


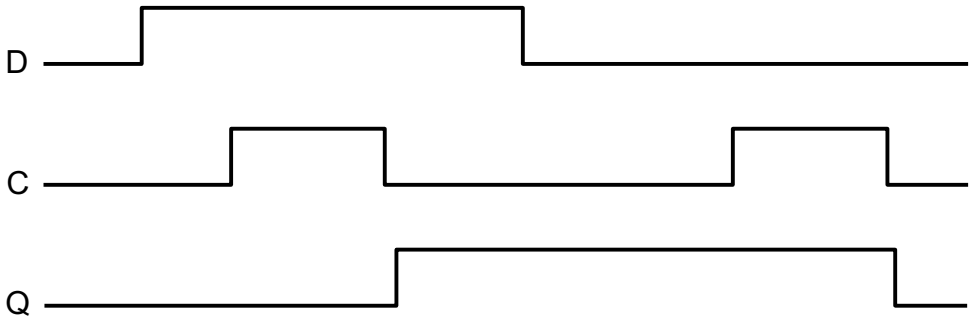


Assume that output Q is initially false and D changes first.
When C is asserted, latch is open
➔ Q assumes value of D input.

D flip-flop

- Output changes only on the clock edge

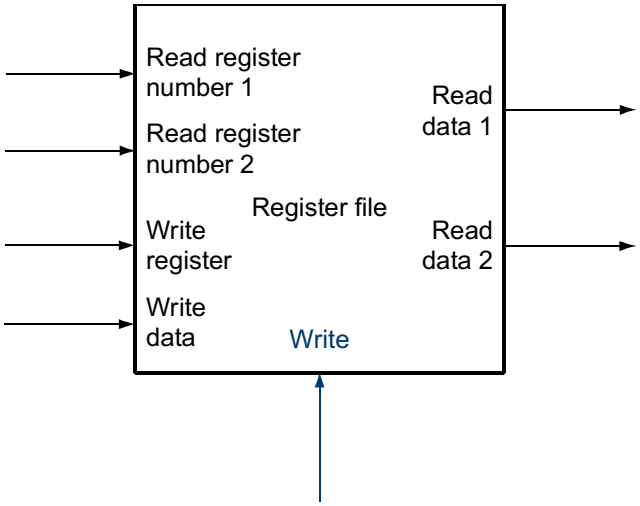


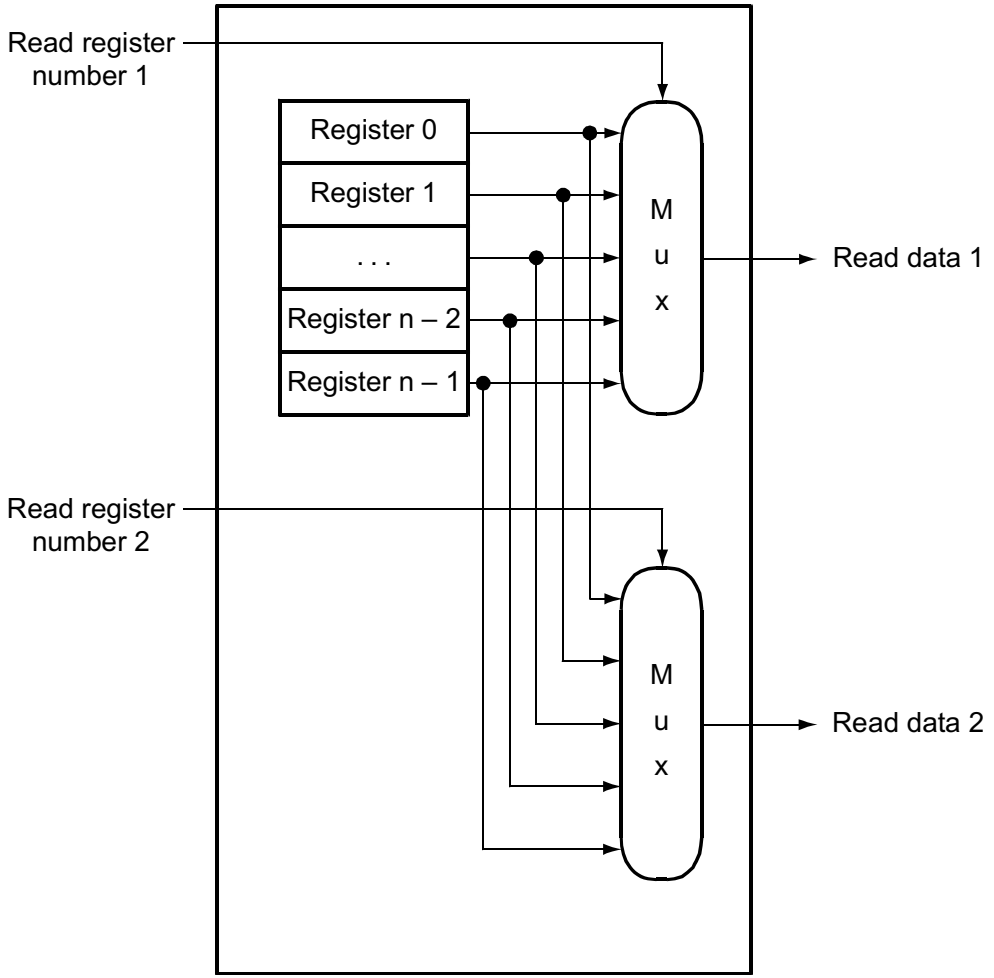


** **Falling-edge trigger**
When C changes from asserted to deasserted, Q stores the value of D.

Register File

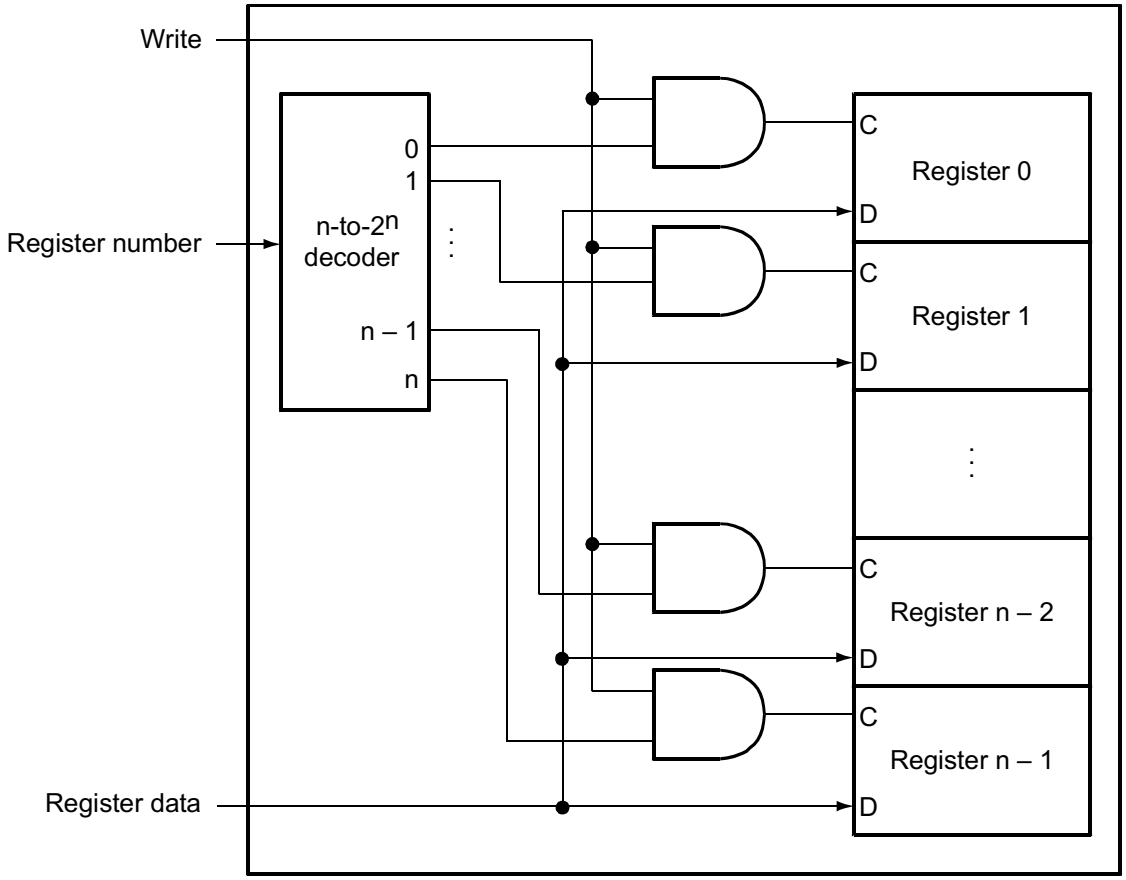
- Built using D flip-flops





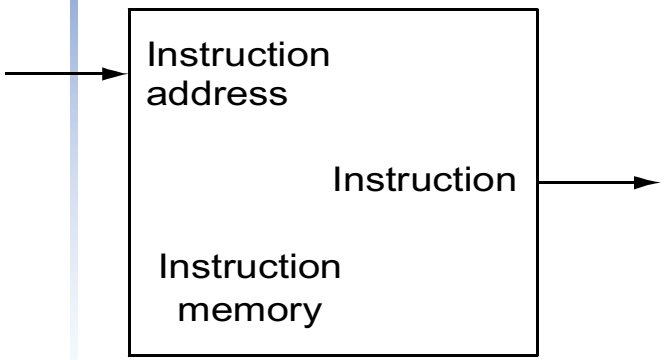
Register File

- Note: we still use the real clock to determine when to write

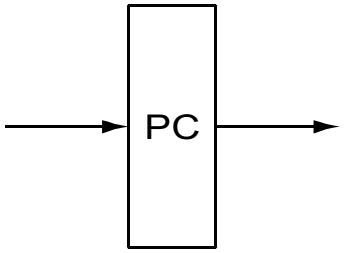


Simple Implementation

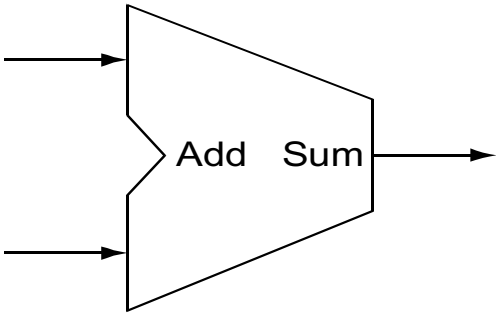
- Include the functional units we need for each instruction



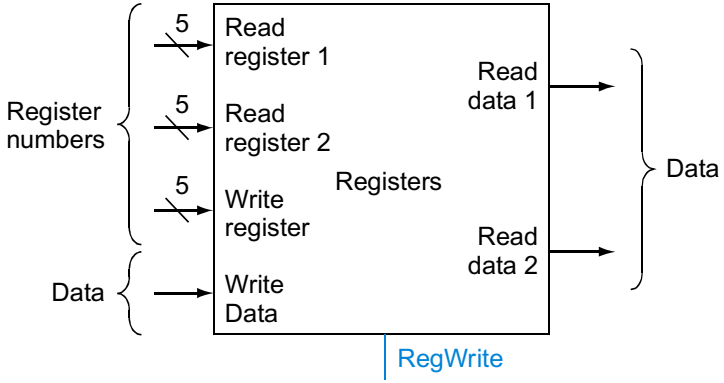
a. Instruction memory



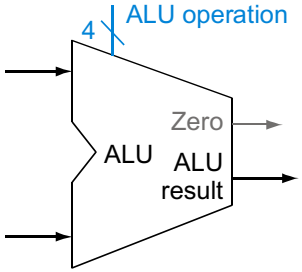
b. Program counter



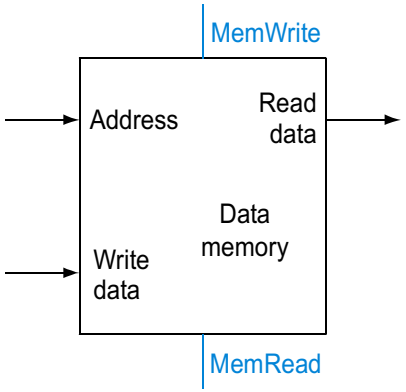
c. Adder



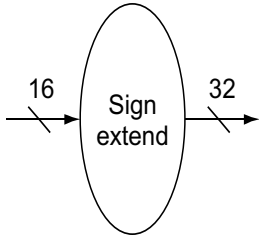
a. Registers



b. ALU



a. Data memory unit

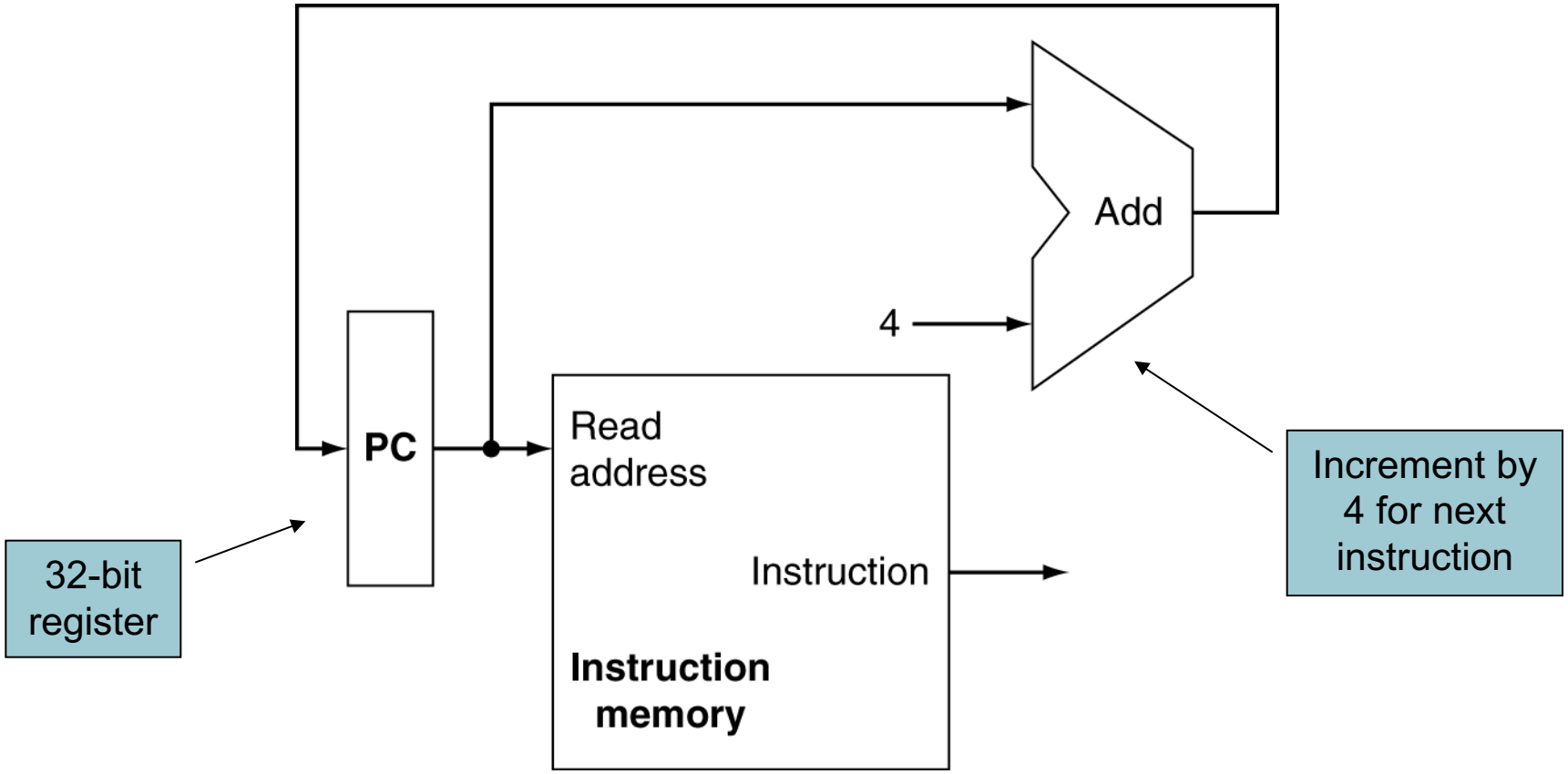


b. Sign-extension unit

Building a Datapath

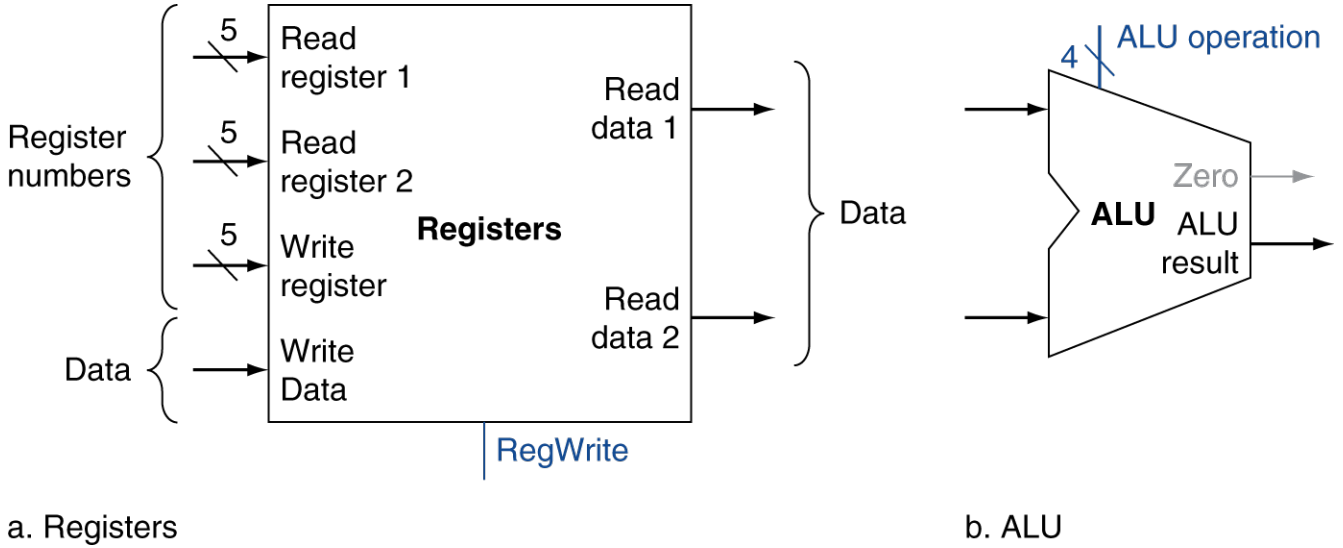
- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

Instruction Fetch



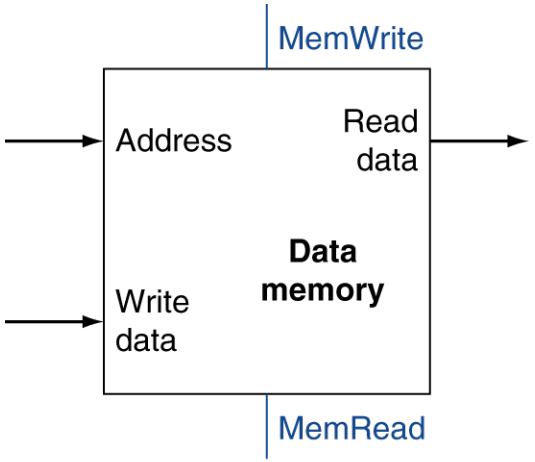
R-Format Instructions

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

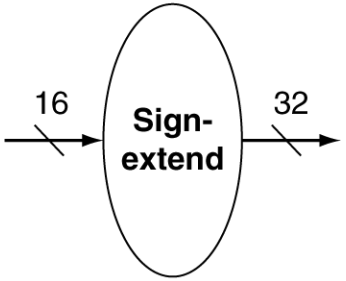


Load/Store Instructions

- 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

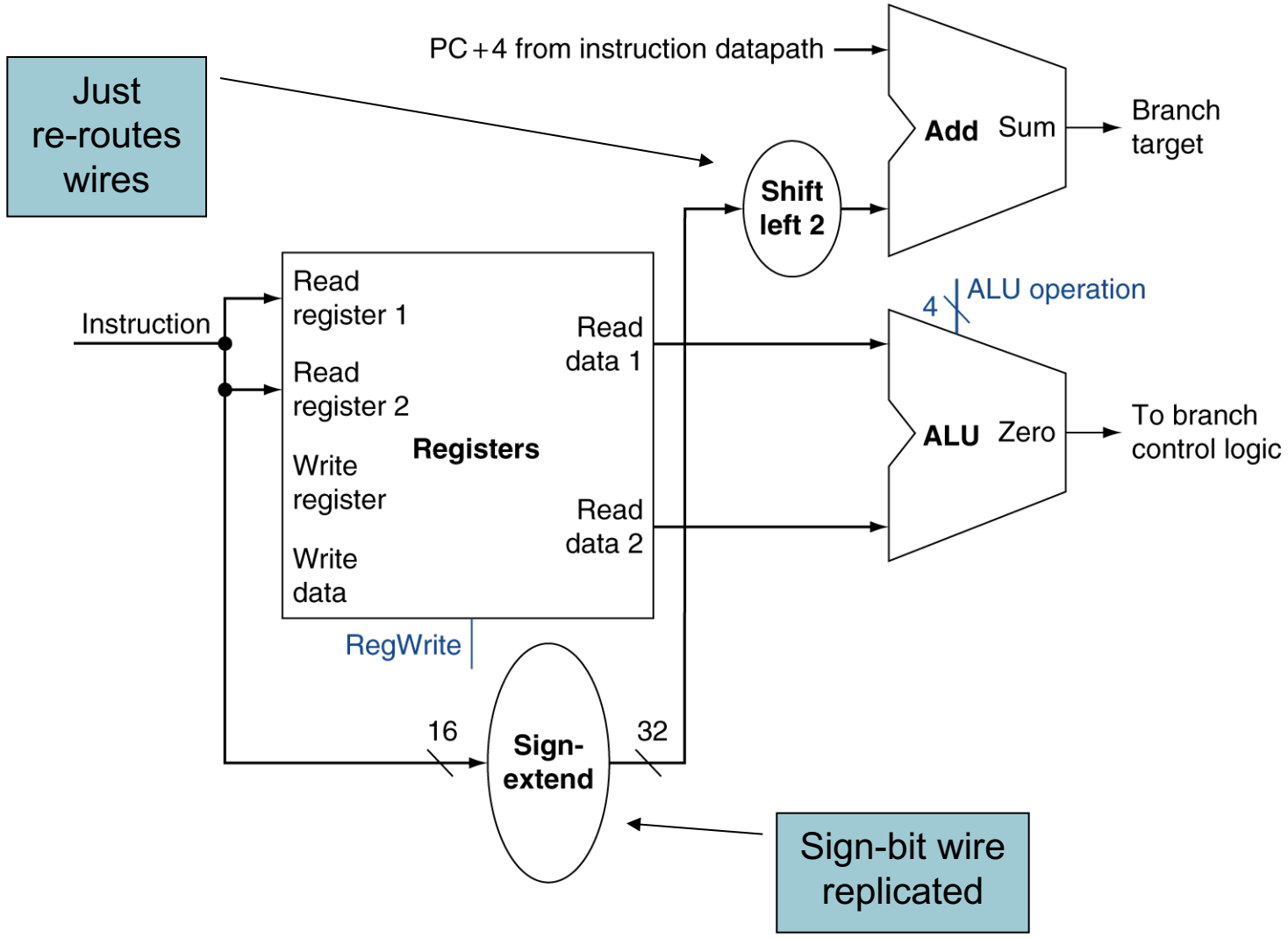


b. Sign extension unit

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 to PC + 4
    - Already calculated by instruction fetch

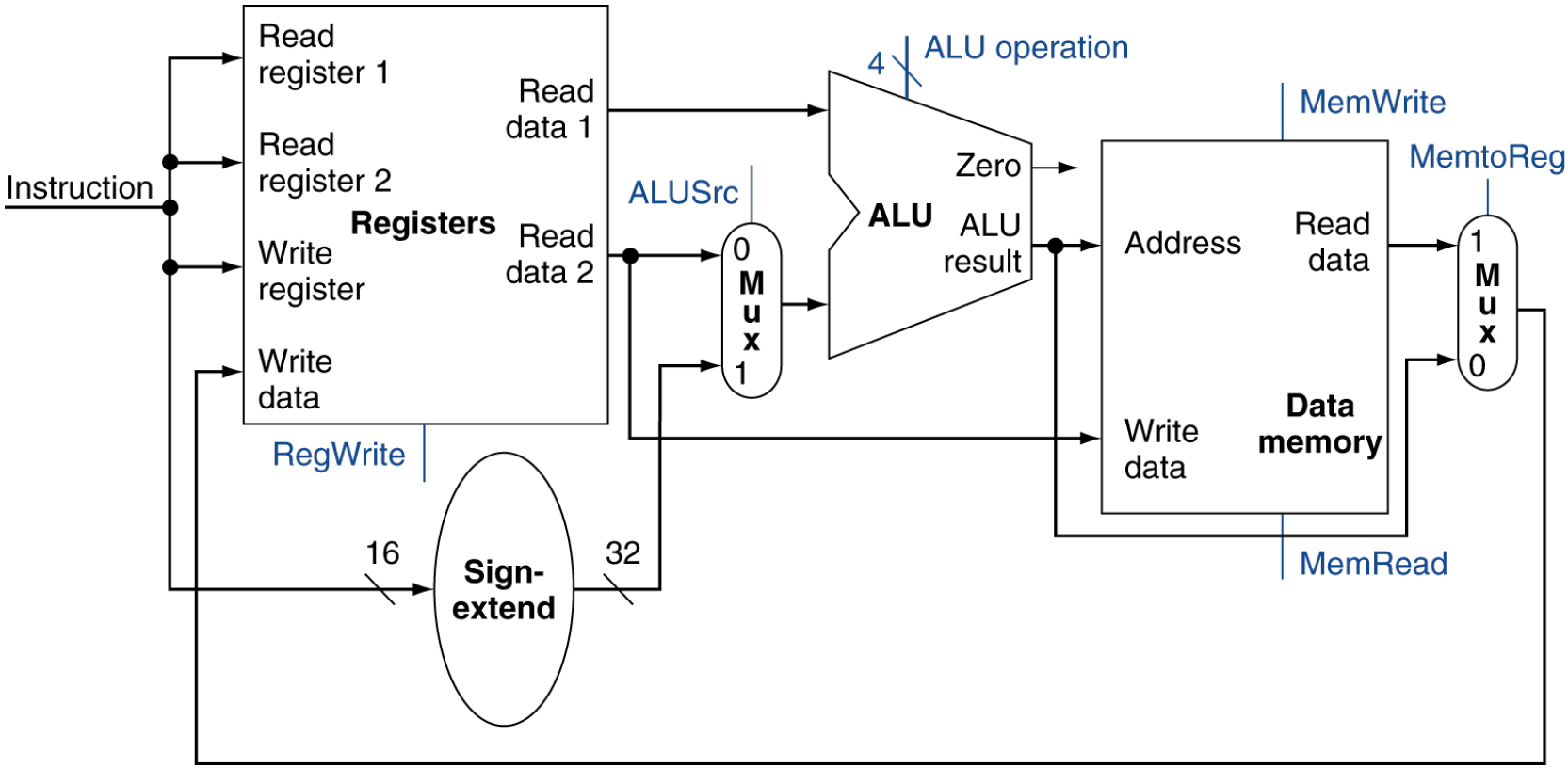
Branch Instructions



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

R-Type/Load/Store Datapath



Full Datapath