BCS361: Computer Architecture

The Processor: Datapath and Control

Computer Science

Logic Blocks

- A logic block has a number of binary inputs and produces a number of binary outputs
- A logic block is termed *combinational* if the output is only a function of the inputs
- A logic block is termed *sequential* if the block has some internal memory (state) that also influences the output
- A basic logic block is termed a *gate* (AND, OR, NOT, etc.)

Truth Table

- A truth table defines the outputs of a logic block for each set of inputs

- Consider a block with 3 inputs A, B, C and an output E that is true only if *exactly* 2 inputs are true

A	B	C	E

Truth Table

- A truth table defines the outputs of a logic block for each set of inputs

- Consider a block with 3 inputs A, B, C and an output E that is true only if *exactly* 2 inputs are true

A	B	C	E
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Boolean Algebra

- Equations involving two values and three primary operators:
  - OR : symbol + , X = A + B ➔ X is true if at least one of A or B is true
  - AND : symbol . , X = A . B ➔ X is true if both A and B are true
  - NOT : symbol ¯

    $X = \bar{A}$ ➔ X is the inverted value of A

Boolean Algebra Rules

- Identity law : $A + 0 = A \quad ; \quad A \cdot 1 = A$
- Zero and One laws : $A + 1 = 1 \; ; \; A \cdot 0 = 0$
- Inverse laws : $A \cdot \overline{A} = 0 \; ; \; A + \overline{A} = 1$
- Commutative laws : $A + B = B + A \quad ; \quad A \cdot B = B \cdot A$
- Associative laws : $A + (B + C) = (A + B) + C$

  $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Distributive laws : $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$

  $A + (B \cdot C) = (A + B) \cdot (A + C)$

DeMorgan's Laws

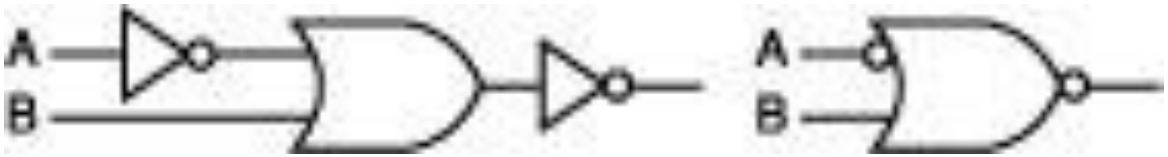
- $\overline{A + B} = \overline{A} \cdot \overline{B}$

- $\overline{A \cdot B} = \overline{A} + \overline{B}$

Pictorial Representations

AND OR NOT

What logic function is this?



Boolean Equation

- Consider the logic block that has an output E that is true only if exactly two of the three inputs A, B, C are true

Boolean Equation

- Consider the logic block that has an output E that is true only if exactly two of the three inputs A, B, C are true

Multiple correct equations:

Two must be true, but all three cannot be true:

$E = ((A \cdot B) + (B \cdot C) + (A \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$

Identify the three cases where it is true:

$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (C \cdot B \cdot \overline{A})$

Sum of Products

- Can represent any logic block with the AND, OR, NOT operators
  - Draw the truth table
  - For each true output, represent the corresponding inputs as a product
  - The final equation is a sum of these products

A	B	C	E
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

$(A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (C \cdot B \cdot \overline{A})$

- Can also use “product of sums”
- Any equation can be implemented with an array of ANDs, followed by an array of ORs

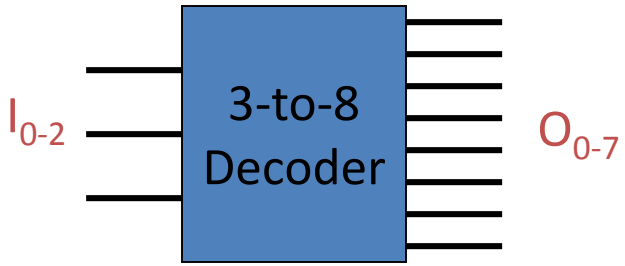
NAND and NOR

- NAND : NOT of AND : A nand B = $\overline{A \cdot B}$
- NOR : NOT of OR : A nor B = $\overline{A + B}$
- NAND and NOR are *universal gates*, i.e., they can be used to construct any complex logical function

Common Logic Blocks – Decoder

Takes in N inputs and activates one of 2^N outputs

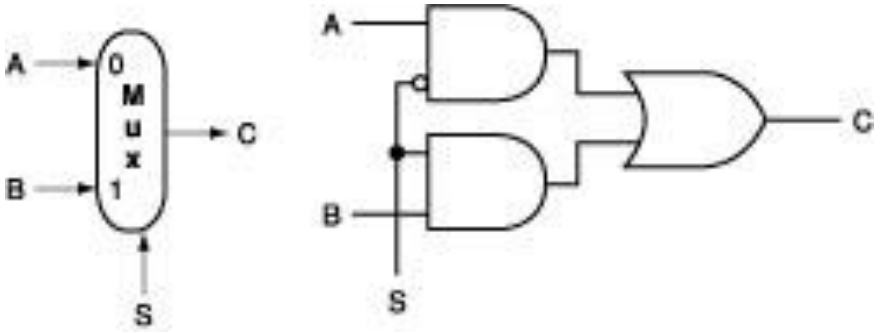
I_0	I_1	I_2	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



Common Logic Blocks – Multiplexor

- Multiplexor or selector: one of N inputs is reflected on the output depending on the value of the $\log_2 N$ selector bits

2-input mux



Adder Algorithm

	1	0	0	1
	0	1	0	1
Sum	1	1	1	0
Carry	0	0	0	1

Truth Table for the above operations:

A	B	Cin	Sum	Cout
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Adder Algorithm

	1	0	0	1
	0	1	0	1
Sum	1	1	1	0
Carry	0	0	0	1

Truth Table for the above operations:

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Equations:

$$\text{Sum} = Cin \cdot \overline{A} \cdot \overline{B} + B \cdot \overline{Cin} \cdot \overline{A} + A \cdot \overline{Cin} \cdot \overline{B} + A \cdot B \cdot Cin$$

$$\text{Cout} = A \cdot B \cdot Cin + A \cdot B \cdot \overline{Cin} + A \cdot Cin \cdot \overline{B} + B \cdot Cin \cdot \overline{A}$$

$$= A \cdot B + A \cdot Cin + B \cdot Cin$$

Carry Out Logic

Equations:

CarryOut = A . B + A . Cin + B . Cin

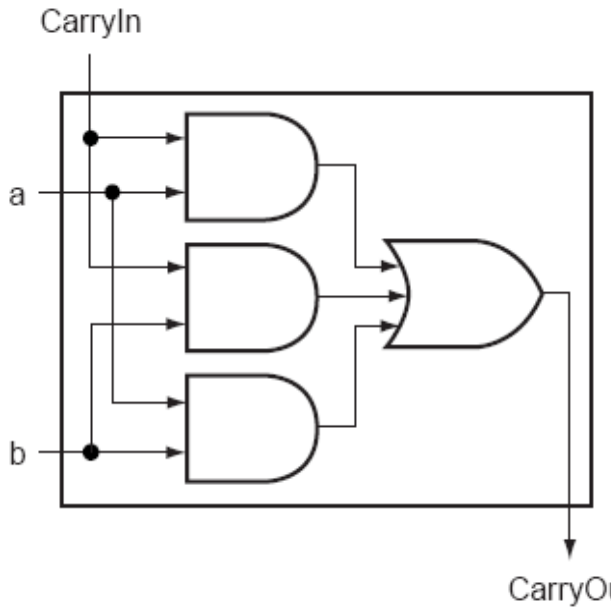
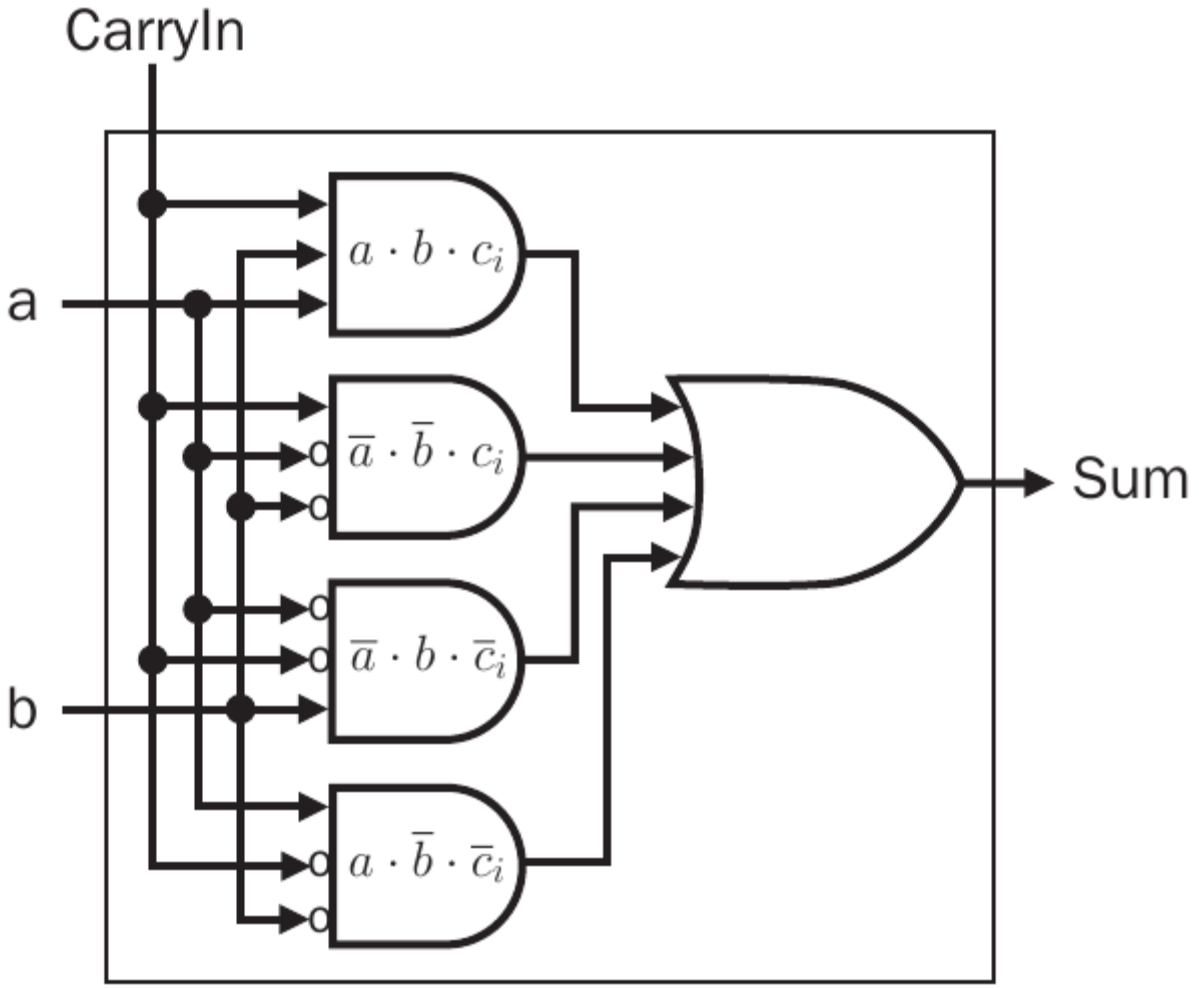


FIGURE B.5.5 Adder hardware for the carry out signal. The rest of the adder hardware is the logic for the Sum output given in the equation on page B-28.

The Sum Logic



$$s = a \cdot \overline{b} \cdot \overline{c}_i + \overline{a} \cdot b \cdot \overline{c}_i + \overline{a} \cdot \overline{b} \cdot c_i + a \cdot b \cdot c_i$$

1-Bit ALU with Add, Or, And

- Multiplexor selects between Add, Or, And operations

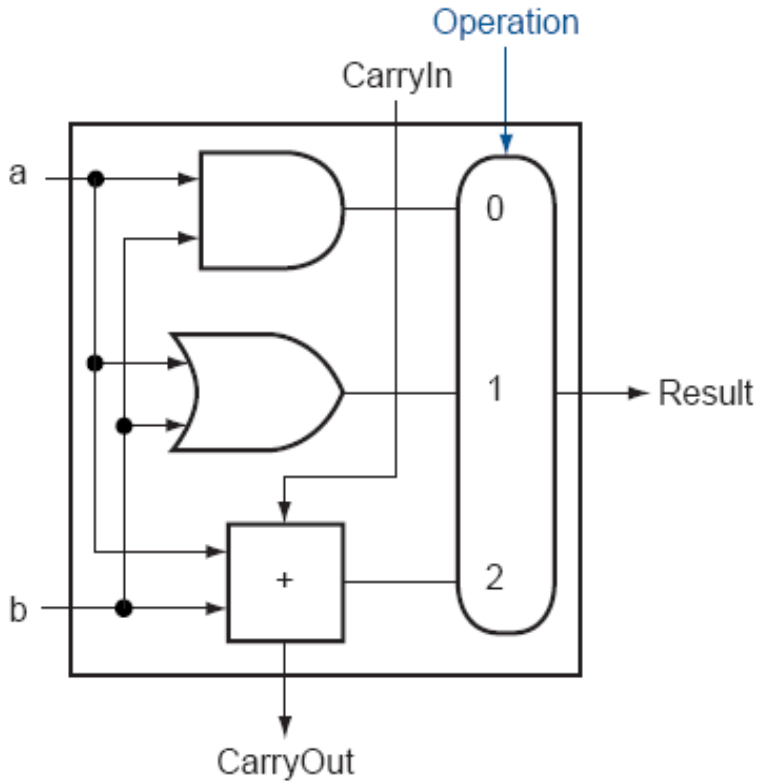


FIGURE B.5.6 A 1-bit ALU that performs AND, OR, and addition (see Figure B.5.5).

32-bit Ripple Carry Adder

1-bit ALUs are connected "in series" with the carry-out of 1 box going into the carry-in of the next box

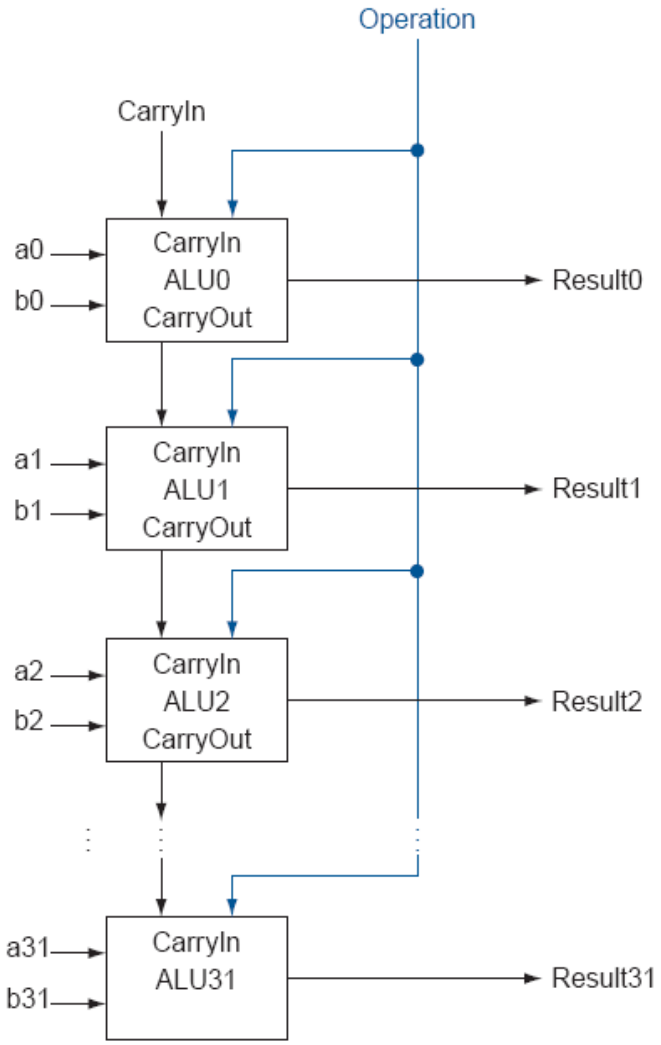


FIGURE B.5.7 A 32-bit ALU constructed from 32 1-bit ALUs. CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry.

Incorporating Subtraction

Incorporating Subtraction

Must invert bits of B and add a 1

- Include an inverter
- CarryIn for the first bit is 1

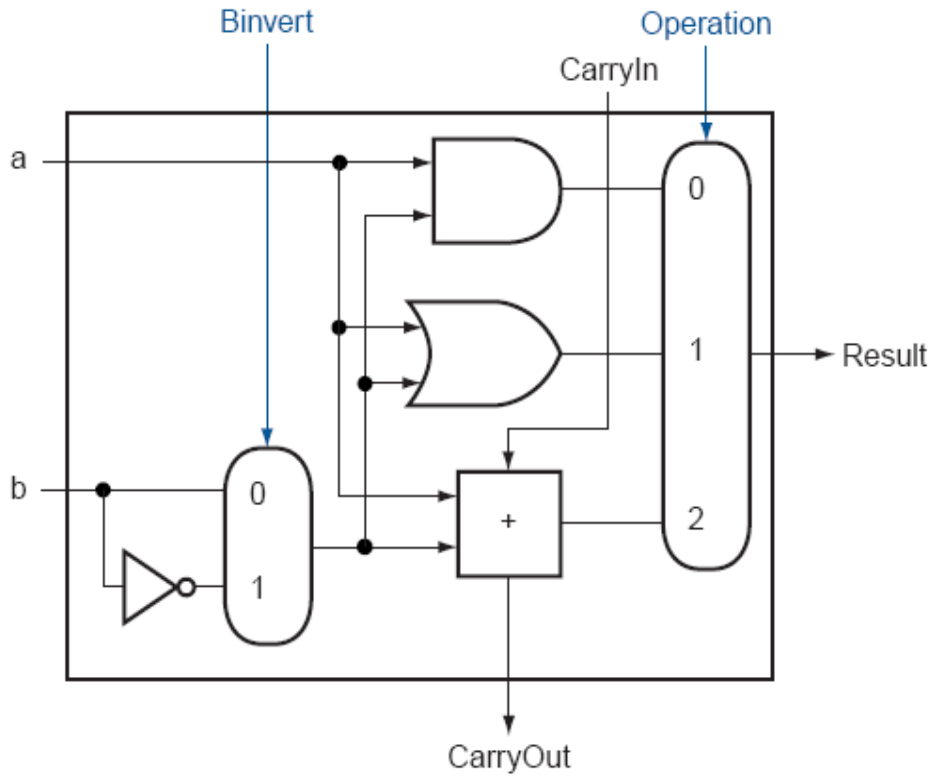


FIGURE B.5.8 A 1-bit ALU that performs AND, OR, and addition on a and b or a and $\overline{b}$. By selecting $\overline{b}$ (Binvert = 1) and setting CarryIn to 1 in the least significant bit of the ALU, we get two's complement subtraction of b from a instead of addition of b to a.

Incorporating NOR

Incorporating NOR

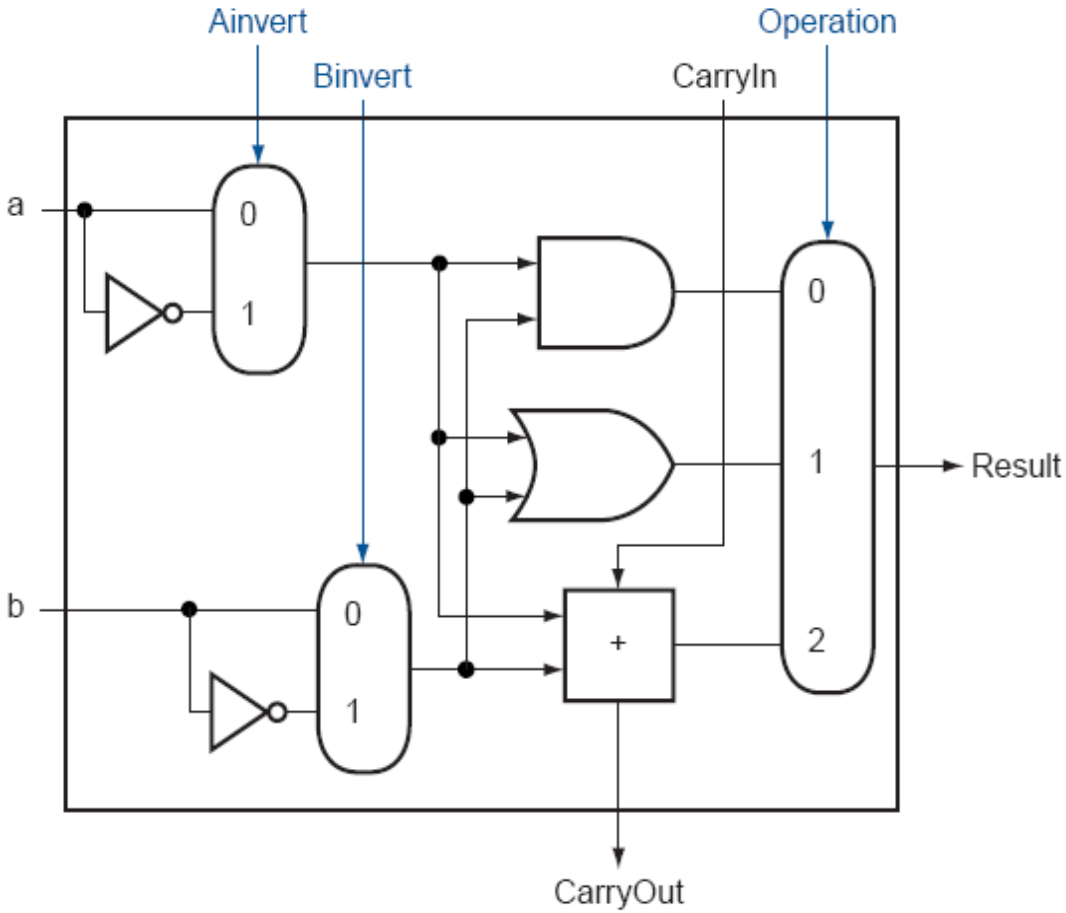
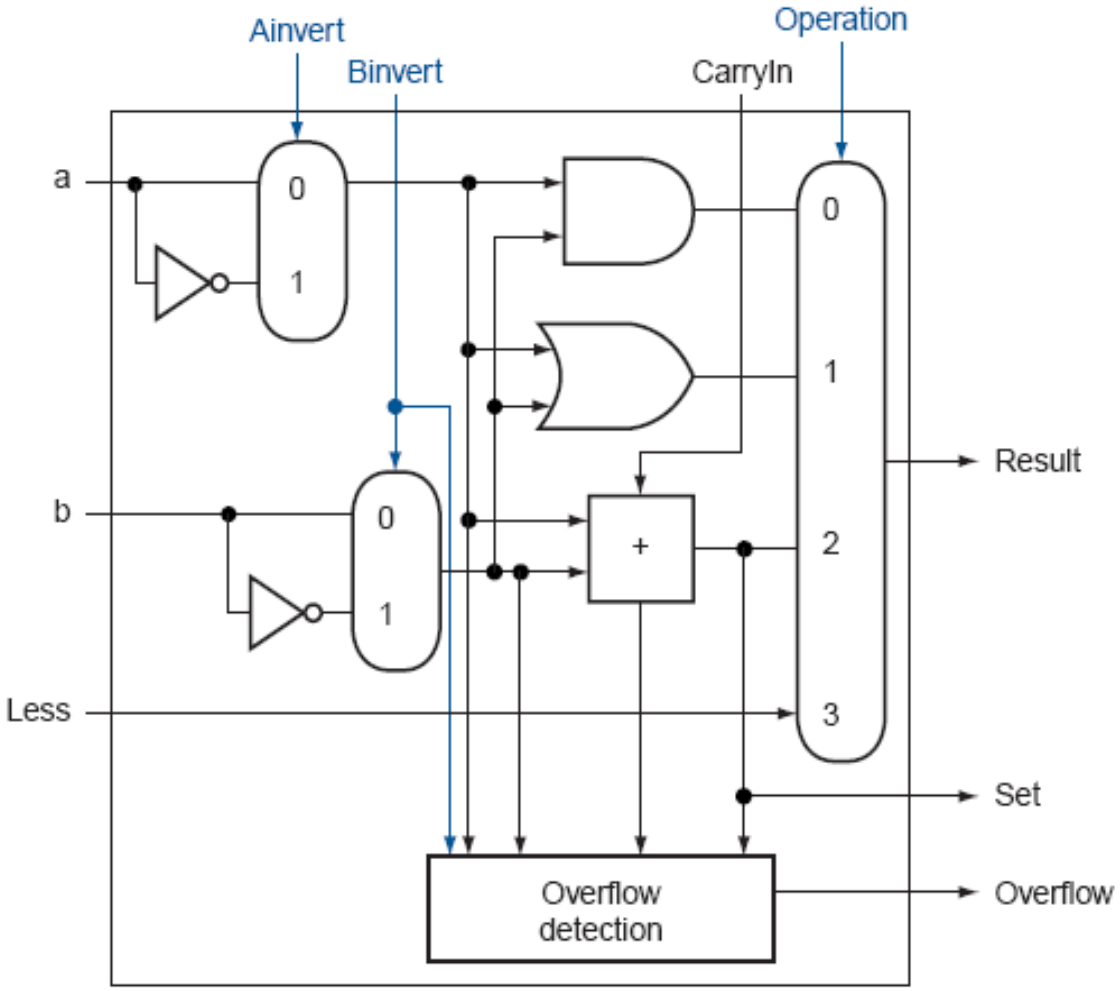


FIGURE B.5.9 A 1-bit ALU that performs AND, OR, and addition on a and b or $\overline{a}$ and $\overline{b}$. By selecting $\overline{a}$ (Ainvert = 1) and $\overline{b}$ (Binvert = 1), we get a NOR b instead of a AND b.

Incorporating slt

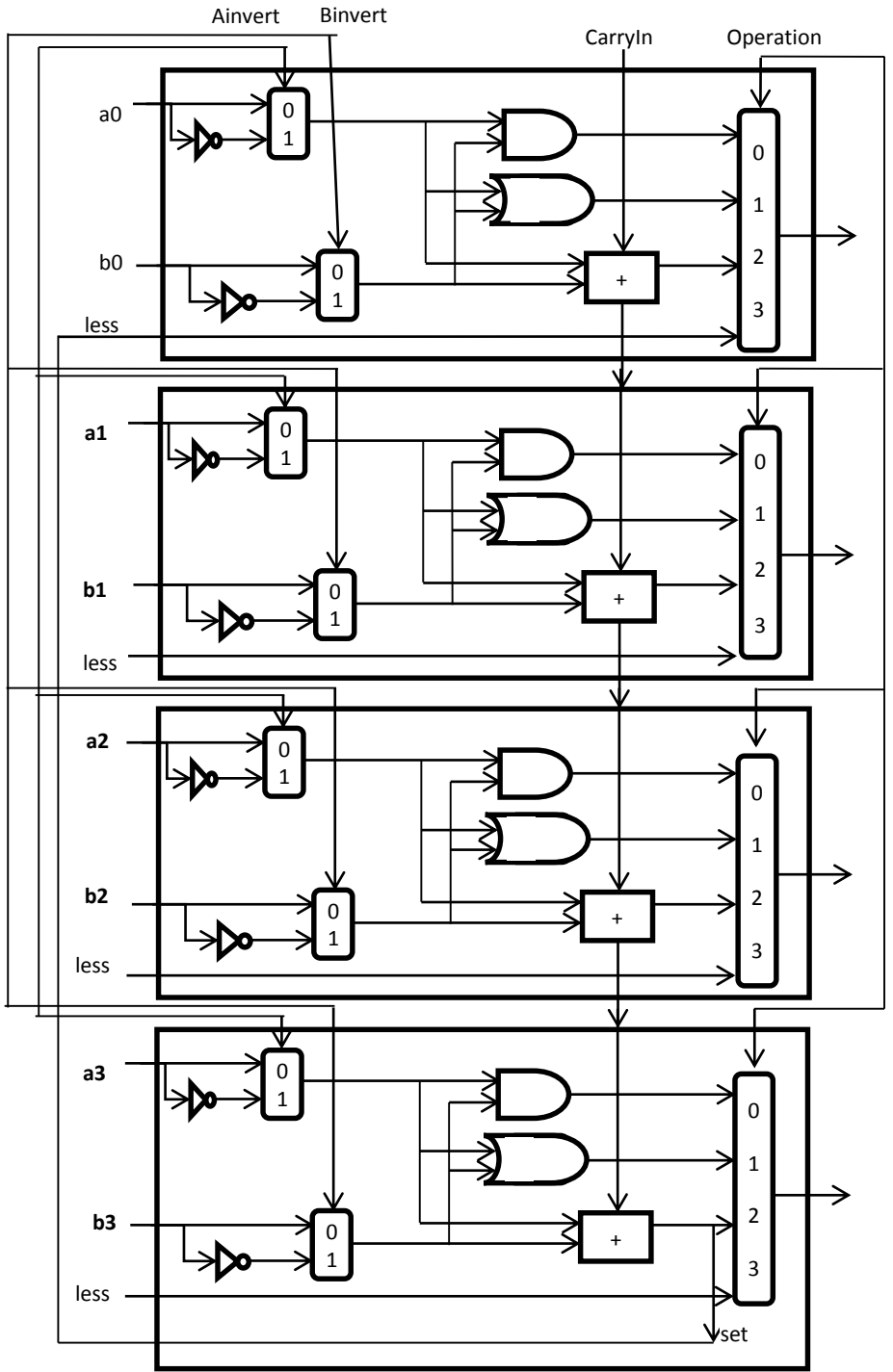
Incorporating slt

- Perform a – b and check the sign
- The 31st box has a unit to detect overflow and sign
- If 31st bit is 1, then a – b < 0; Set is true and is fed to Less in bit 0.
- For all bits other than bit 0, Less must be 0.



Incorporating slt

- Perform a – b and check the sign
- The 31st box has a unit to detect overflow and sign
- If 31st bit is 1, then a – b < 0; Set is true and is fed to Less in bit 0.
- For all bits other than bit 0, Less must be 0.



Incorporating beq

- Perform a – b and confirm that the result is all zero's

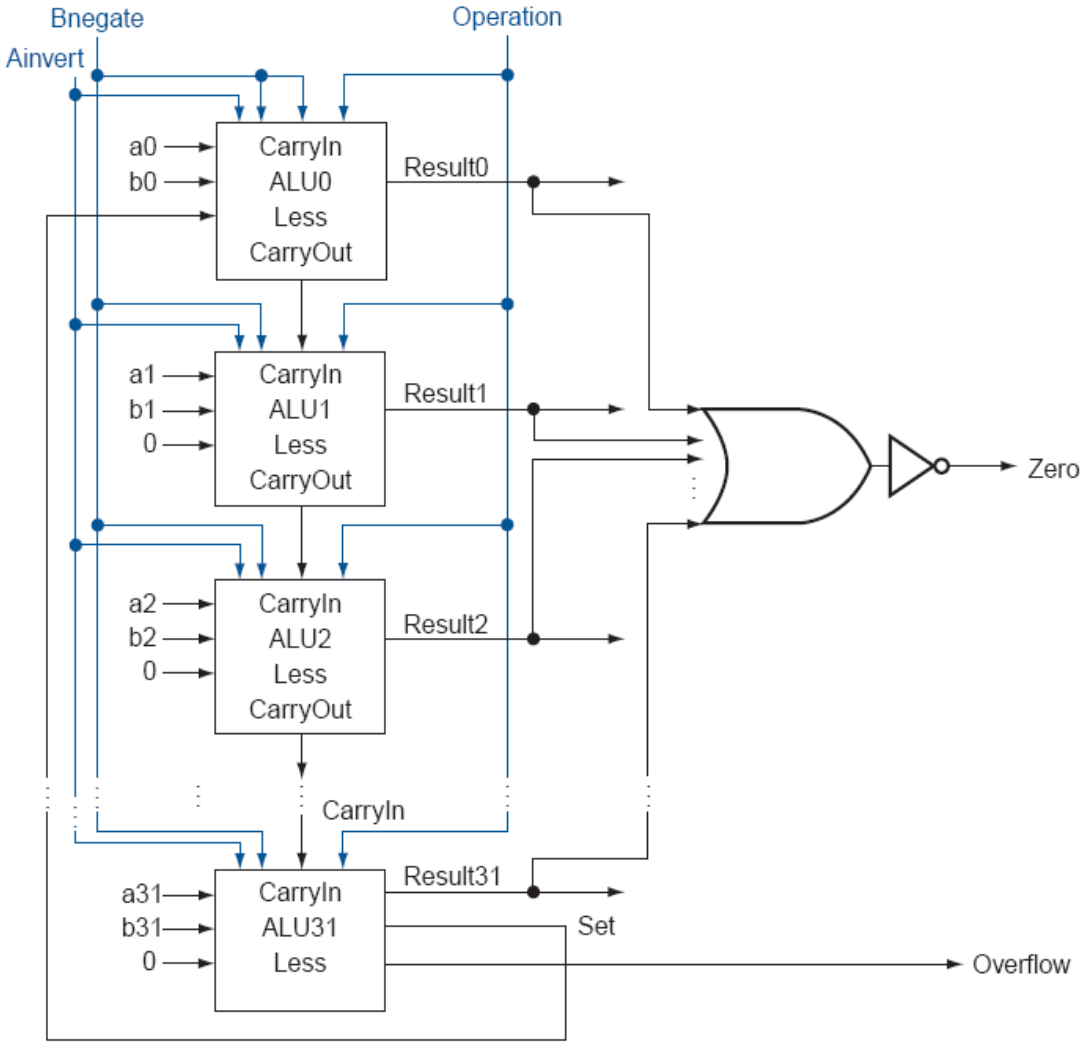


FIGURE B.5.12 The final 32-bit ALU. This adds a Zero detector to Figure B.5.11.

Control Lines

What are the values of the control lines and what operations do they correspond to?

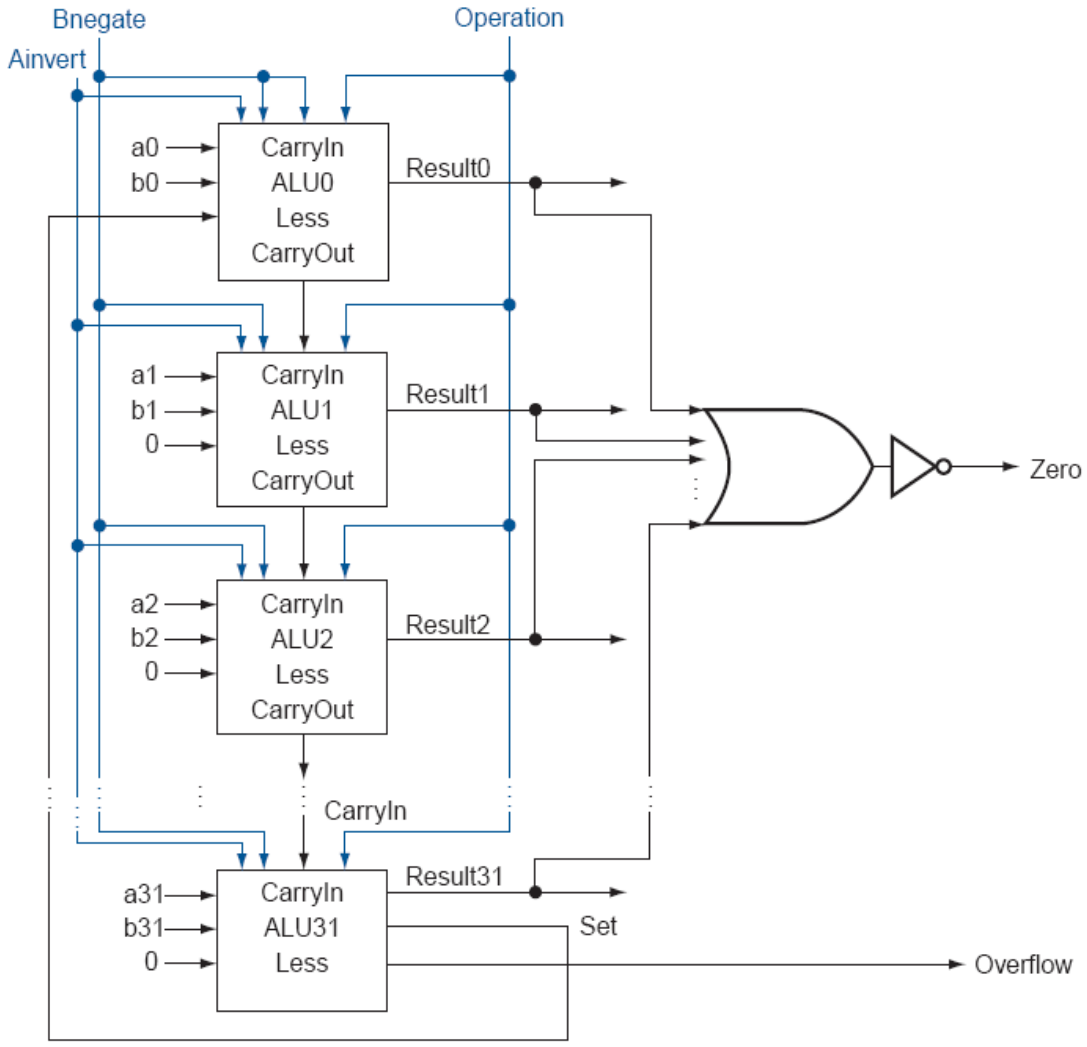
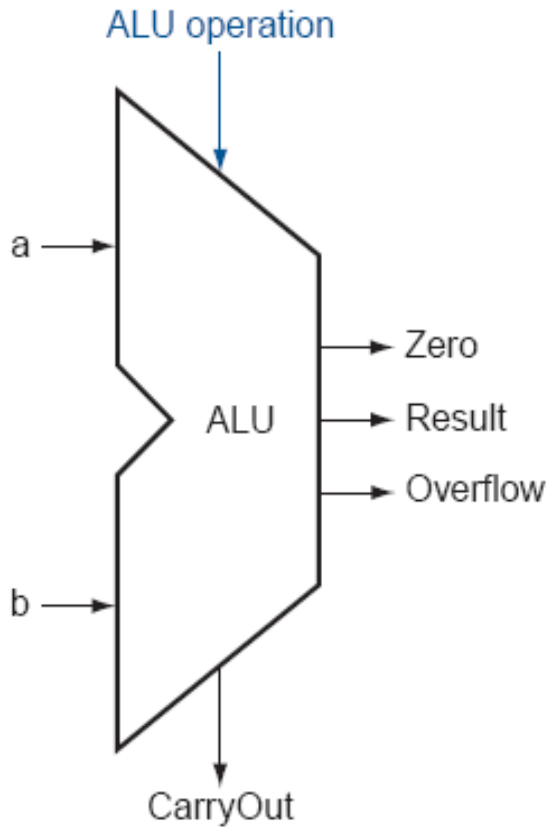


FIGURE B.5.12 The final 32-bit ALU. This adds a Zero detector to Figure B.5.11.

Control Lines

What are the values of the control lines and what operations do they correspond to?

	Ai	Bn	Op
AND	0	0	00
OR	0	0	01
Add	0	0	10
Sub	0	1	10
SLT	0	1	11
NOR	1	1	00



Basic MIPS Architecture

- We'll design a simple CPU that executes:
  - basic math (add, sub, and, or, slt)
  - memory access (lw and sw)
  - branch and jump instructions (beq and j)

Implementation Overview

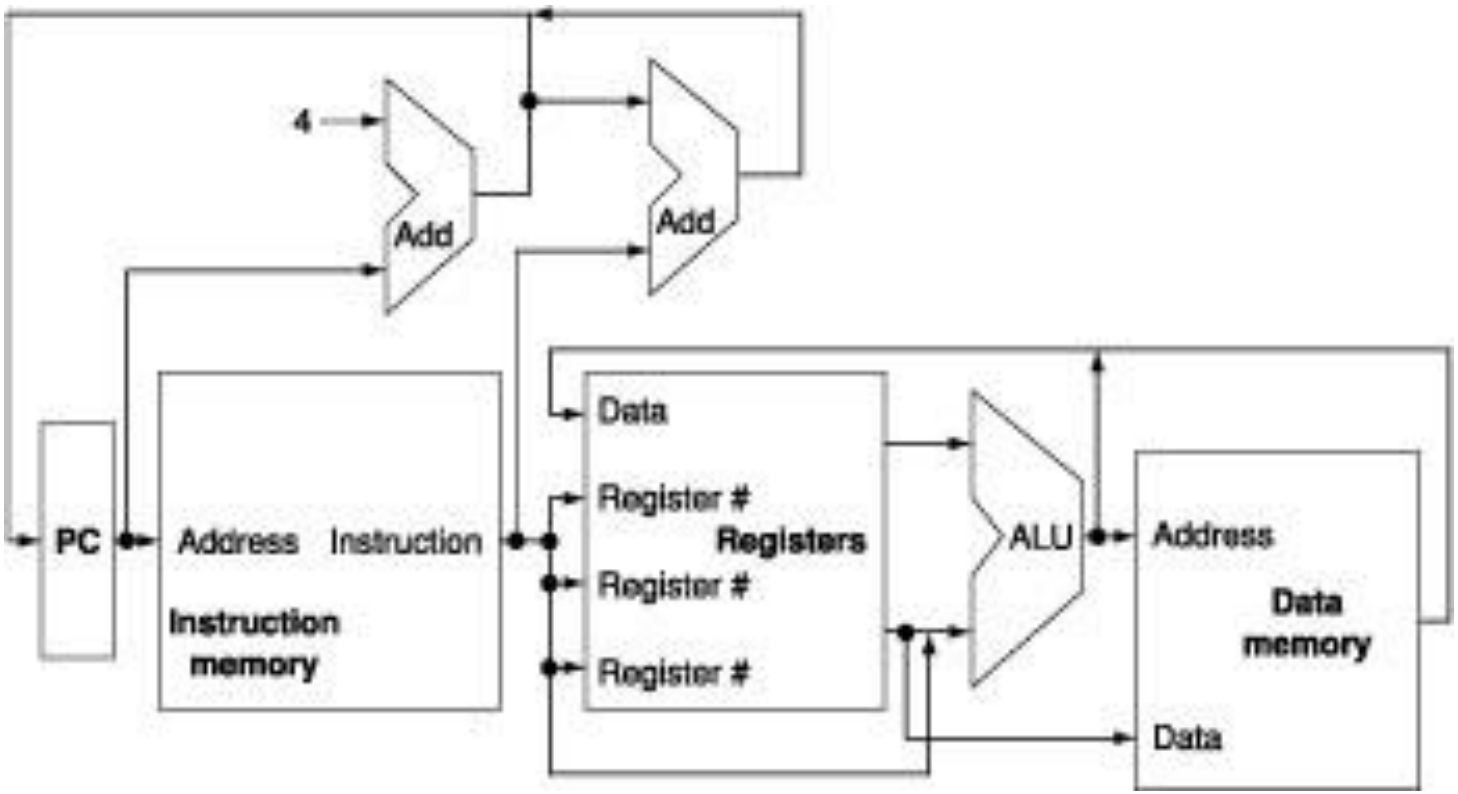
- We need memory
  - to store instructions
  - to store data
  - for now, let's make them separate units
- We need registers, ALU, and a whole lot of control logic
- CPU operations common to all instructions:
  - use the program counter (PC) to pull instruction out of instruction memory
  - read register values

Datapath

- Datapath Element: A functional unit used to operate or hold data within the processor. Examples in MIPS implementation are memory, register file, ALU and adders.

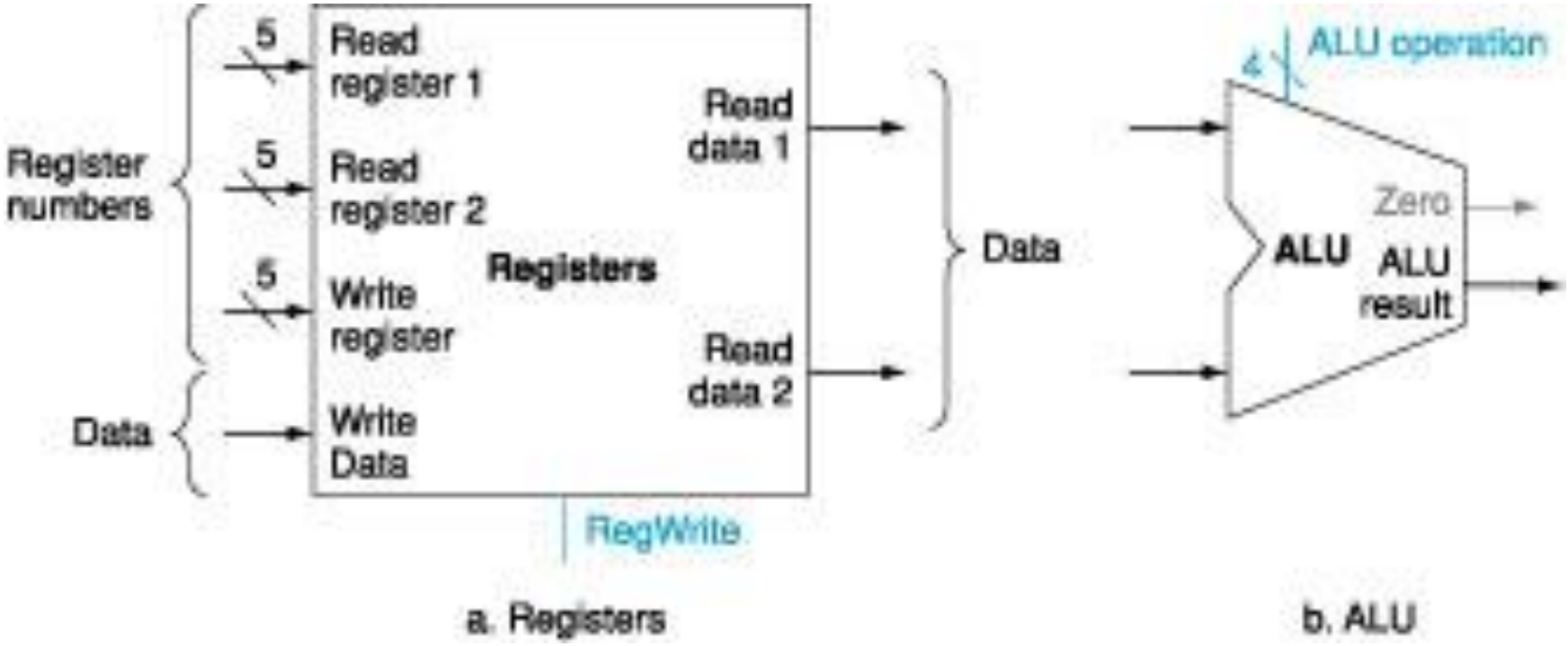
- Datapath: Collection of all datapath elements

View from 30,000 Feet



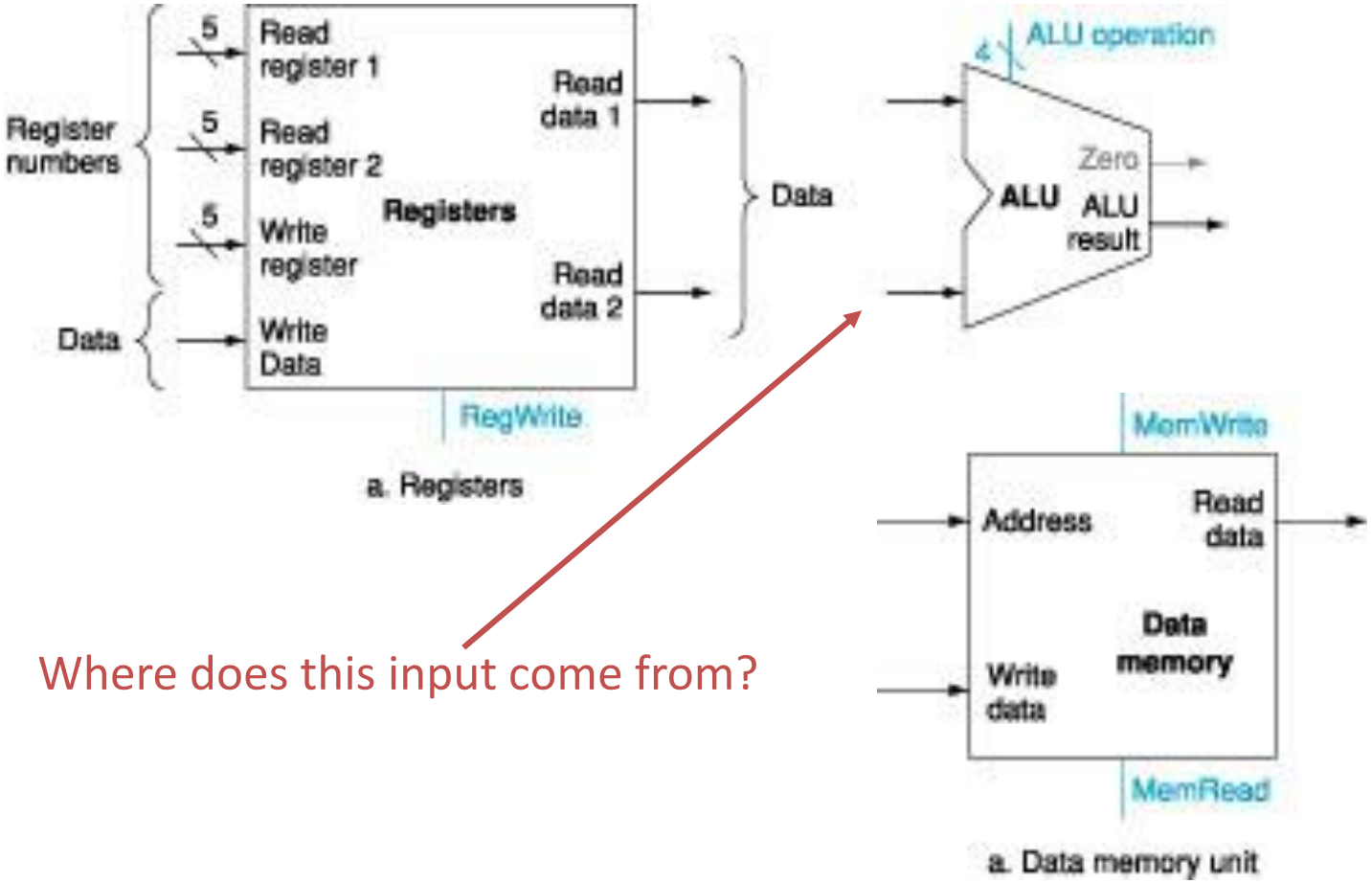
Implementing R-type Instructions

- Instructions of the form add $t1, $t2, $t3



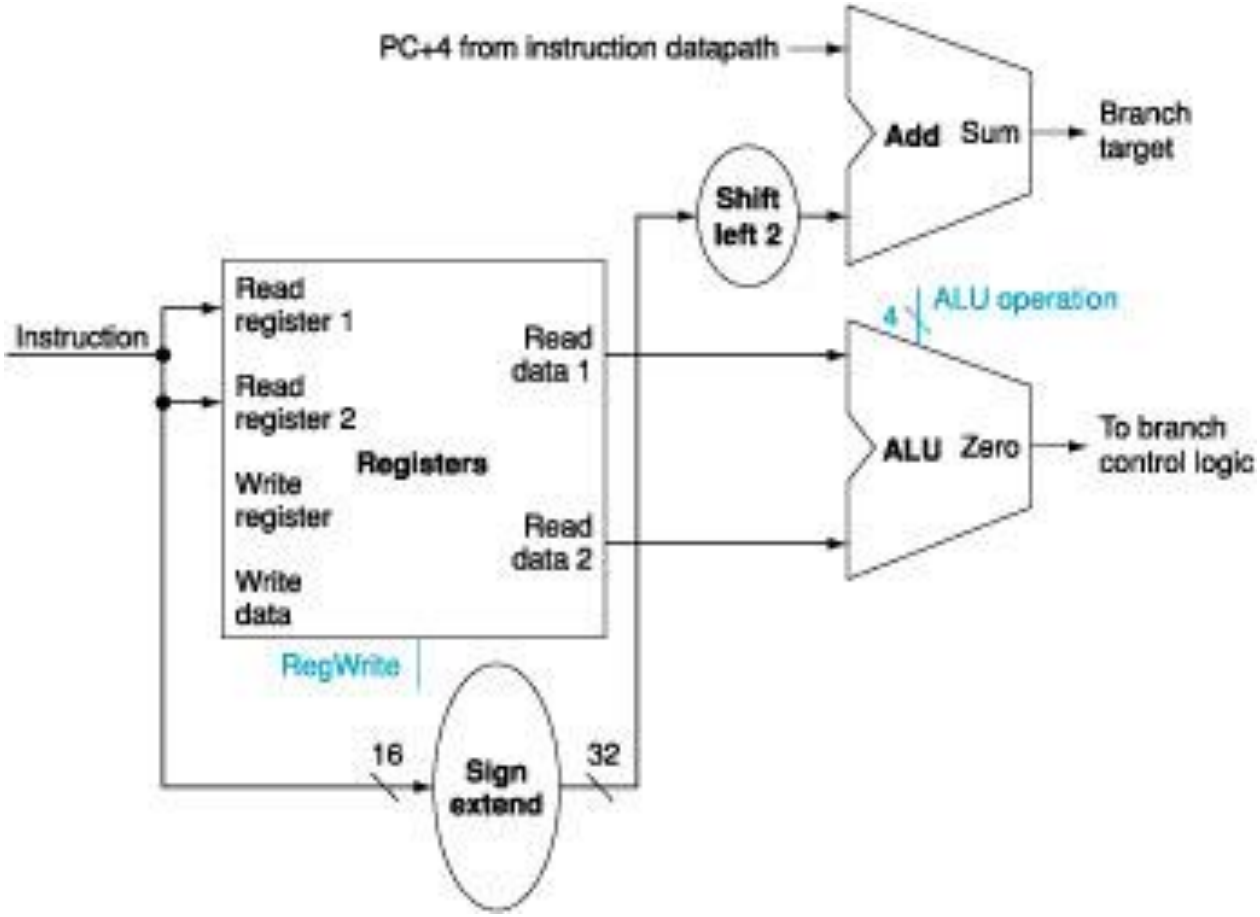
Implementing Loads/Stores

- Instructions of the form lw $t1, 8($t2) and sw $t1, 8($t2)

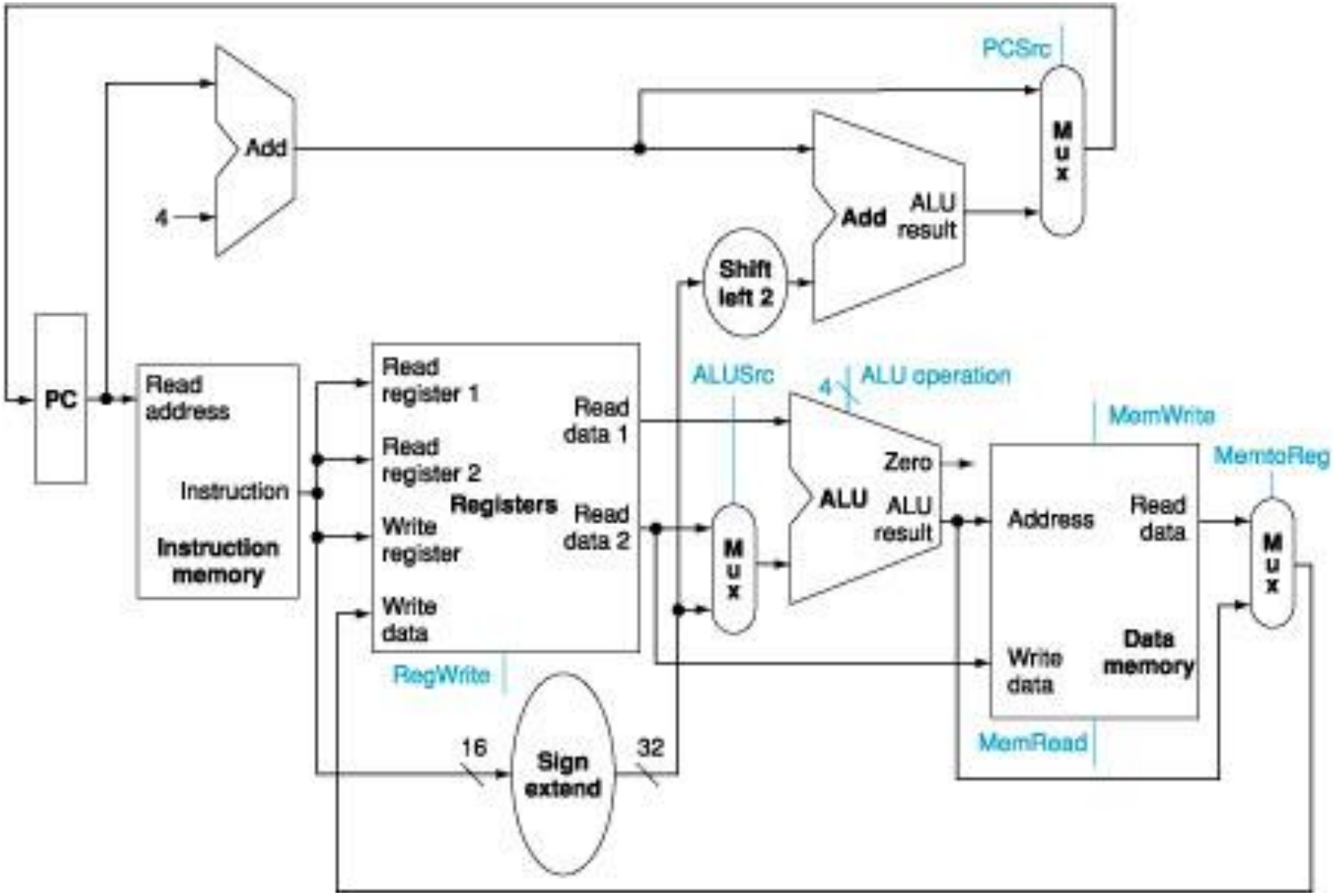


Implementing J-type Instructions

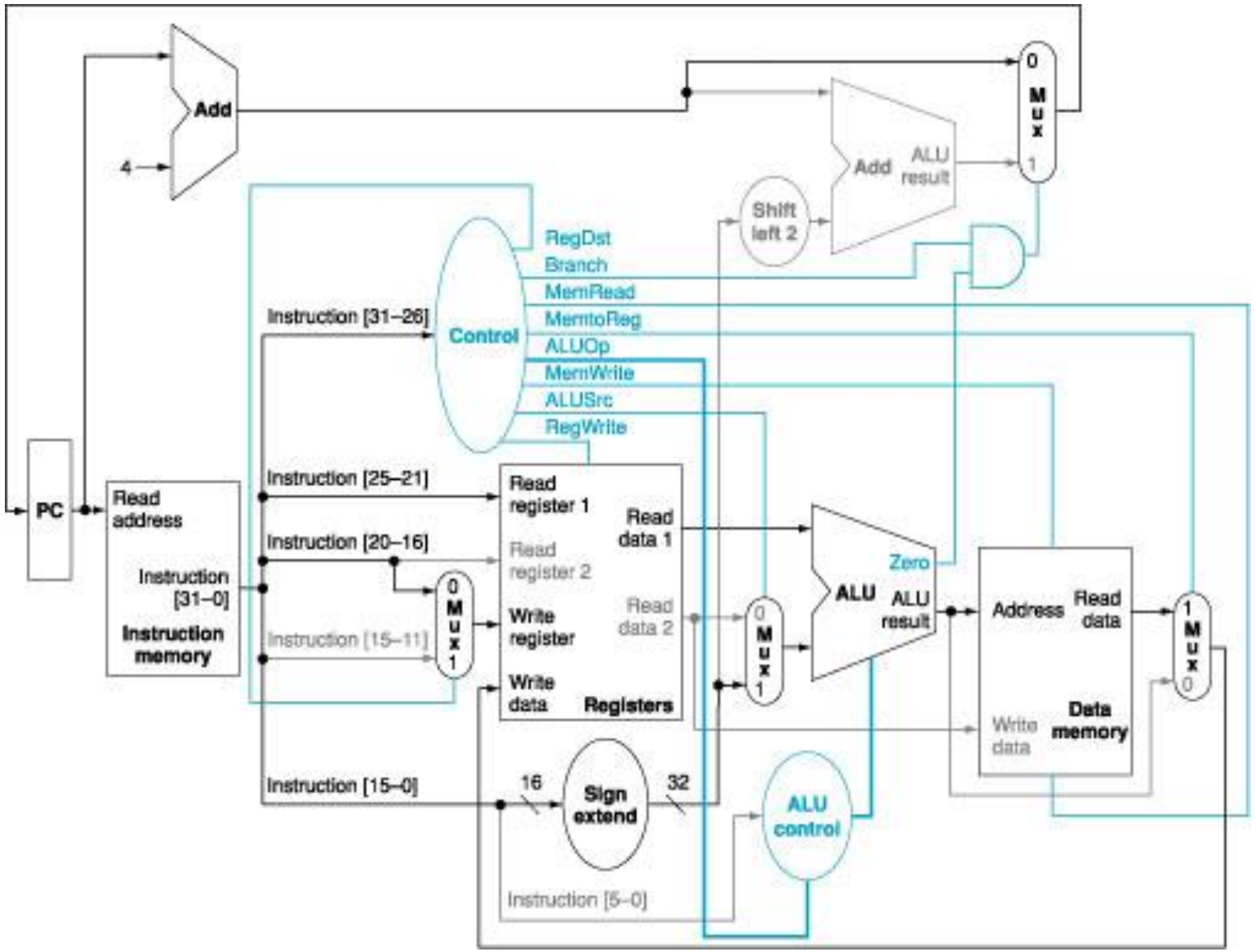
- Instructions of the form beq $t1, $t2, offset



View from 10,000 Feet



View from 5,000 Feet



Reviewing the R Type and I Type Format

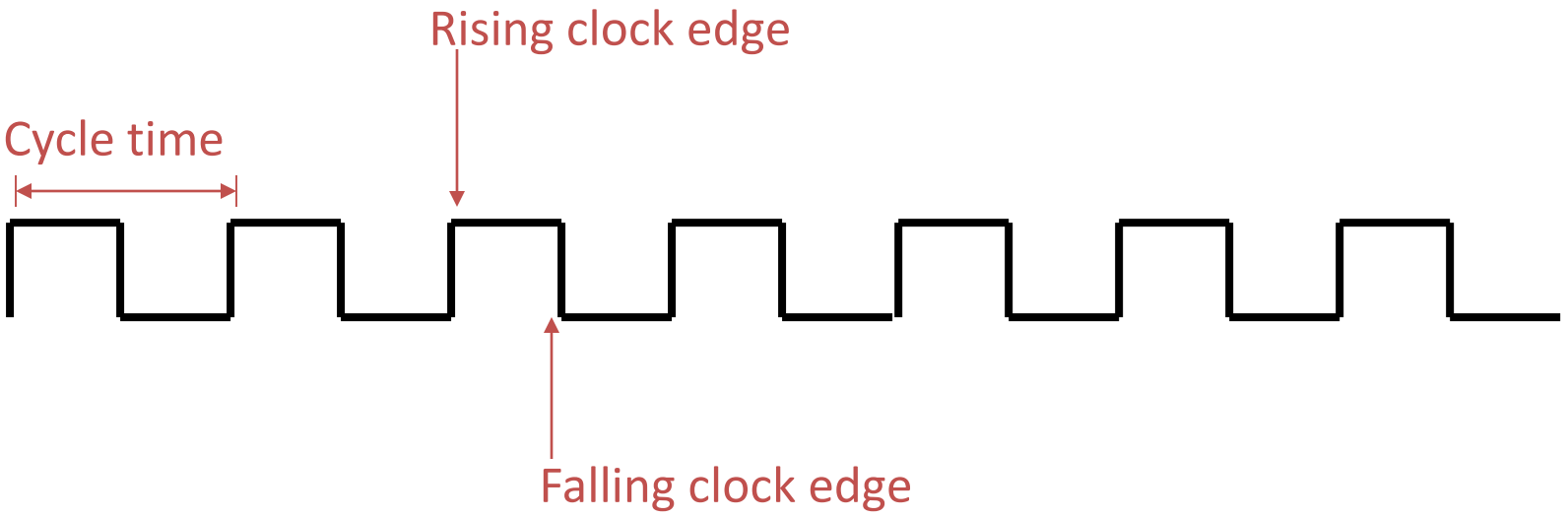
R-type instruction add $t0, $s1, $s2

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct
opcode	source	source	dest	shift amt	function

I-type instruction lw $t0, 32($s3)

6 bits	5 bits	5 bits	16 bits
opcode	rs	rd	constant

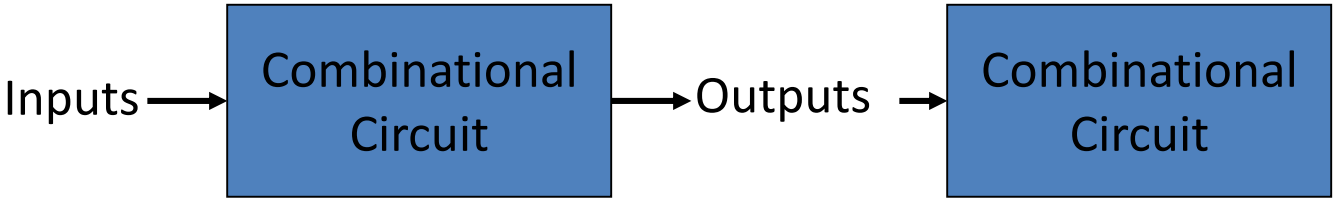
Clock Terminology



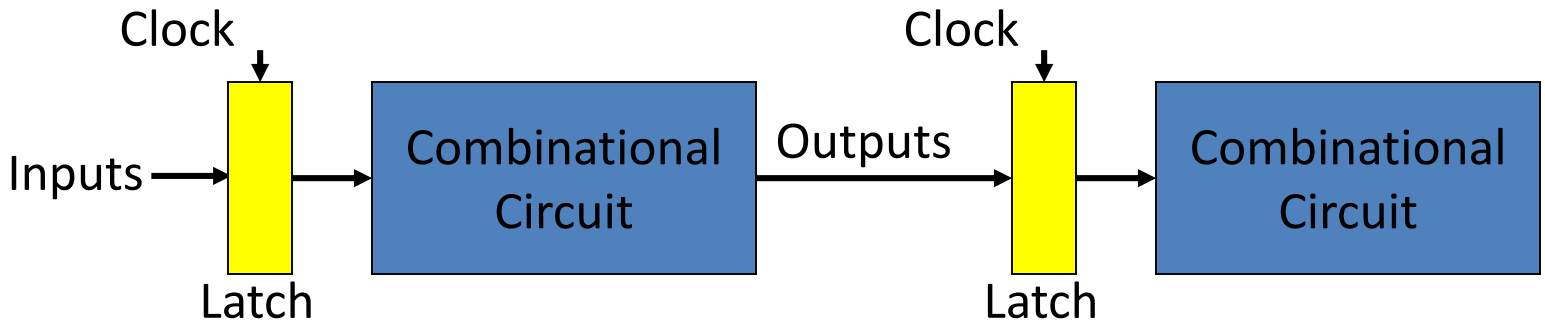
$$4\ \text{GHz} = \text{clock speed} = \frac{1}{\text{cycle time}} = \frac{1}{250\ \text{ps}}.$$

Sequential Circuits

- Until now, circuits were combinational – when inputs change, the outputs change after a while (time = logic delay through circuit)



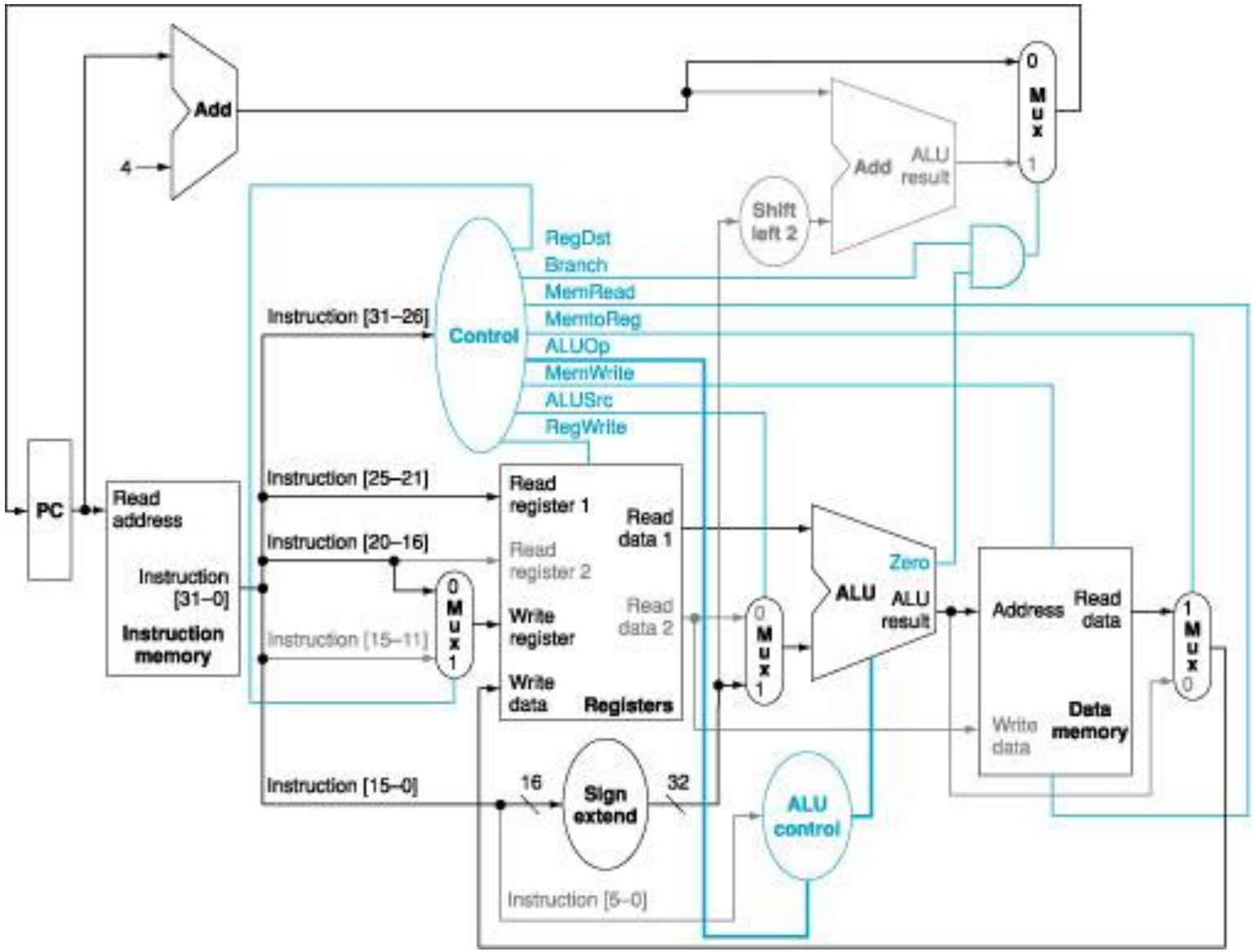
- We want the clock to act like a start and stop signal – a "latch" is a storage device that stores its inputs at a rising clock edge and this storage will not change until the next rising clock edge



Sequential Circuits

- Sequential circuit: consists of combinational circuit and a storage element
- At the start of the clock cycle, the rising edge causes the “state” storage to store some input values
- This state will not change for an entire cycle (until next rising edge)
- The combinational circuit has some time to accept the value of “state” and “inputs” and produce “outputs”
- Some of the outputs (for example, the value of next “state”) may feed back (but through the latch so they’re only seen in the next cycle).

View from 5,000 Feet



Example: MIPS Clock Rate

- Determine the clock rate for the MIPS architecture, assuming the following:
  - The MIPS is a Single Cycle Machine
    - 1 clock cycle per instruction
    - CPI = 1
  - Access time for memory units = 200 ps
  - Operation time for ALU and adders = 100 ps
  - Access time for register file = 50 ps

Example: MIPS Clock Rate

Instruction Class	Functional Units used by the Instruction Class				
ALU Instruction	Inst. Fetch	Register	ALU	Register	
Load Word	Inst. Fetch	Register	ALU	Memory	Register
Store Word	Inst. Fetch	Register	ALU	Memory	
Branch	Inst. Fetch	Register	ALU		
Jump	Inst. Fetch				

Example: MIPS Clock Rate

Instruction Class	Instr Memory	Register read	ALU operation	Data Memory	Register write	Total
ALU Instruction	200	50	100	0	50	400 ps
Load Word	200	50	100	200	50	600 ps
Store Word	200	50	100	200	0	550 ps
Branch	200	50	100	0	0	350 ps
Jump	200	0	0	0	0	200 ps

Example: MIPS Clock Rate

- The clock cycle time for a machine with a single clock cycle per instruction will be determined by the longest instruction.
  - In this example, the load word instruction requires 600 ps.
- The clock rate is then

  Clock rate = 1 / Clock Cycle Time

  Clock rate = 1 / 600 ps = 1.67 GHz

Performance Issues

- Longest delay determines clock period
  - Critical path: load word (lw) instruction
  - Instruction memory → register file → ALU → data memory → register file
- Improve performance by pipelining

1. How can we design future multi and many-core architectures to have better performance, spend less power and be easier to program—all at a lower cost?
2. How do we restructure the memory hierarchy to meet the demands of “big data” applications such as dynamically changing graphs, machine intelligence, and search?
3. What should be the architectures of future GPUs?
4. How do we help programmers efficiently create and debug their software for parallel architectures?
5. How to utilize characteristics of the emerging applications to specialize current and design next generation computing systems?
6. How do we quickly simulate future computer architectures with current computers?
7. What are the advantages and disadvantages of having a large number of pipeline stages in a processor? What is the future of Instruction Level Parallelism?
8. What is the support provided for Thread Level Parallelism in current architectures? What is the future of Thread Level Parallelism
9. What are the different Single Instruction Multiple Data (SIMD) architectures available

- Cloud:
  - Running Scientific High Performance Computing Applications on the Cloud

- University of Cambridge:
  1. Languages and Compilers for multi-core architectures
  2. Flexible support for speculation, synchronization and coherency
  3. Fine-grain parallel communication-centric architectures
  4. Techniques for improving Cache utilization

Princeton: The Liberty Research Group

- The Parallelization Project
- The Fault Tolerance Project
- The Compiler Foundations Project
- The Security Project