Computer Architecture Lab 05

Arithmetic Logic Unit

Introduction

This lab focuses on designing and understanding the Arithmetic Logic Unit (ALU) — the core component responsible for performing computations inside a processor. Every instruction that involves arithmetic (like addition or subtraction) or logical operations (like AND, OR, or XOR) ultimately passes through the ALU.

By implementing an ALU in Verilog, you will gain insight into how a processor executes operations at the hardware level, how control signals determine the operation performed, and how status flags (like zero, carry, or overflow) are generated as a result.

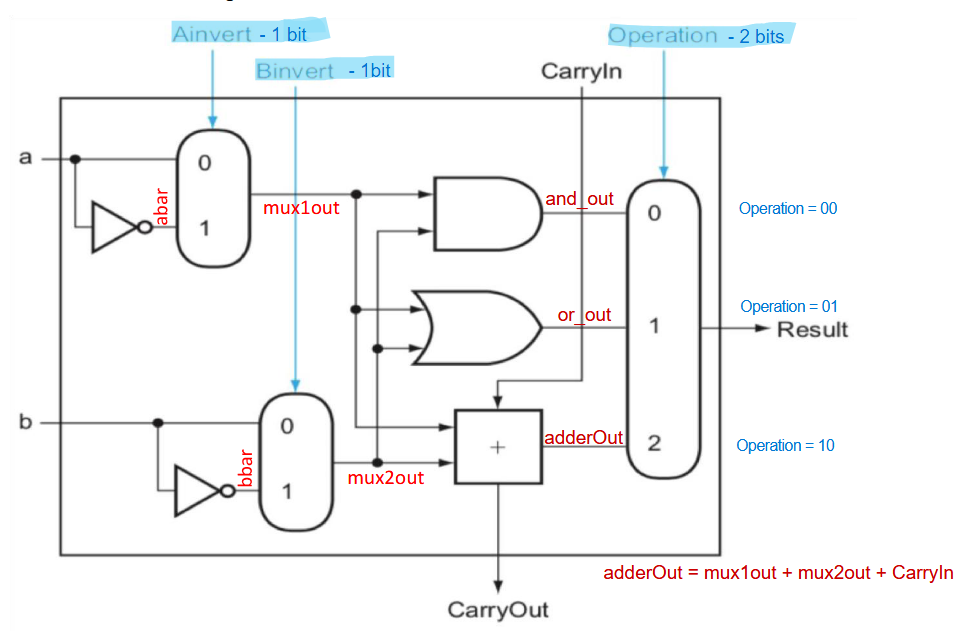
This lab also serves as a foundation for later topics such as instruction execution and datapath design. You’ll use combinational logic to perform different operations based on an operation select input, and later integrate your ALU with other components like registers and control units.

Objectives

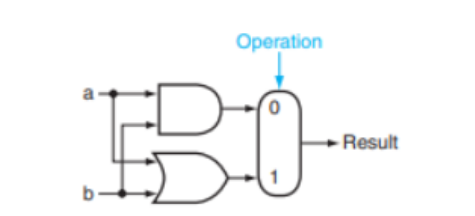
* Understand the role and structure of an ALU in a processor datapath.
* Implement arithmetic and logical operations using Verilog.
* Learn to design hardware modules that use control signals to select between multiple operations.
* Observe how status flags are generated and interpreted.

Task 1: Single Bit ALU

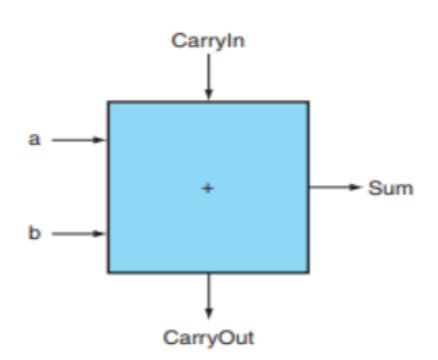
We start by constructing a 1-bit ALU



The 1-bit logical unit for AND and OR looks like this. The multiplexor on the right then selects a AND b or a OR b, depending on whether the value of Operation is 0 or 1. The line that controls the multiplexor is shown in colour to distinguish it from the lines containing data. Notice that we have renamed the control and output lines of the multiplexor to give them names that reflect the function of the ALU.



The next function to include is addition. An adder must have two inputs for the operands and a single-bit output for the sum. There must be a second output to pass on the carry, called CarryOut. Since the CarryOut from the neighbor adder must be included as an input, we need a third input. This input is called CarryIn. The inputs and the outputs of a 1-bit adder look like this:



The ALU decides the operation to be performed through a 4-bit signal called operation, which becomes the control signal for all multiplexers. Fill the table below to gain clarity on the operations to be performed:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operation | A-Invert (operation[3]) | B-Invert (operation[2]) | operation[1] | operation[0] |
| and | 0 | 0 |  |  |
| or | 0 | 0 |  |  |
| add | 0 | 0 |  |  |
| subtract |  |  | 1 | 0 |
| nor |  |  | 0 | o |
| nand |  |  | 0 | 1 |

Provide your module here:

|  |
| --- |
|  |

Task 2: N-Bit ALU

After designing and understanding the 1-bit ALU, we can now extend this concept to create an N-bit ALU, a unit capable of operating on multi-bit inputs (such as 8-bit, 16-bit, or 32-bit numbers).

The N-bit ALU is built by cascading multiple 1-bit ALU units together. Each bit position in the operands is handled by one instance of the 1-bit ALU. The CarryOut of each lower bit becomes the CarryIn of the next higher bit, allowing arithmetic operations (like addition and subtraction) to propagate carries correctly across the entire word.

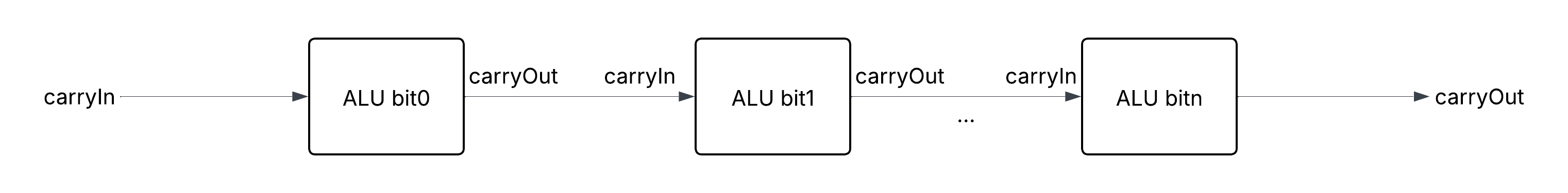
The control signals (such as the operation select lines) are shared across all 1-bit ALUs, ensuring that each bit performs the same operation in parallel.

This modular design approach demonstrates one of the most powerful ideas in hardware design: scalability. By constructing complex systems from smaller building blocks, we can handle larger data widths without fundamentally changing our logic.

Your N-bit ALU module should:

* Take two N-bit inputs (a and b).
* Include a single-bit CarryIn input and a CarryOut output.
* Use a 4 bit control input (operation) to select which function to perform (AND, OR, ADD, etc.).
* Output an N-bit result.

Your chain of ALUs may look something like this:



Think carefully about what the initial carryIn should be (hint: it has to do with the operations) and state your observations:

|  |
| --- |
|  |

Provide your module, and a suitable test bench testing each operation here:

|  |  |
| --- | --- |
| Code: | Testbench: |
| Waveform: | |

Task 3: Devising ALU control

So far, you have built an ALU capable of performing a variety of arithmetic and logical operations depending on a 4-bit control signal (operation).  
But where does this 4-bit control signal come from?

In an actual processor, the ALU’s behaviour isn’t hardcoded — it’s determined by the instruction being executed.  
For example:

* A load instruction needs the ALU to perform addition (to compute memory addresses).
* A branch instruction (like beq) needs the ALU to subtract the operands and check if the result is zero.
* An R-type instruction (like add, sub, and, or) specifies its exact operation using a function field (funct) in the instruction.

To handle all this, we design a separate component called the ALU Control Unit.

|  |
| --- |
| Understanding the ALU Control Unit  The main control unit (from the processor’s control logic) sends a 2-bit ALUOp signal to this ALU control block.  The ALU control unit then uses ALUOp and the instruction’s function field (funct) to determine the 4-bit operation signal that goes into your ALU. |

Fill the table below to determine the logic of this module

|  |  |  |
| --- | --- | --- |
| ALUOp | funct | Operation |
| 00 | 0010 |  |
| 10 | 0000 |  |
| 10 | 1000 |  |
| 10 | 0111 |  |
| 10 | 0110 |  |
| 10 | 0001 |  |

Provide your module, with a suitable testbench here:

|  |  |
| --- | --- |
| Code: | Testbench: |
| Waveform: | |

Task 4: Handling branch (BEQ/BNEQ/BLT) instructions:

In real processors, the ALU not only performs computations but also provides status outputs that help determine control flow. For example, whether two values are equal or whether one is less than another.

When executing branch instructions, these status outputs are crucial:

* For BEQ (Branch if Equal), the processor checks whether the result of (A - B) is zero.
* For BLT (Branch if Less Than), it checks if A < B.

These checks are performed using special signals that your ALU should generate alongside the normal result output.

Modify your ALU to produce two status signals:

* ZERO: Indicates whether the output of the ALU is zero.
* lessThan — Indicates whether the first operand (A) is less than the second operand (B).

Also modify your ALUControl to handle the case where the ALUOp reflects a branch instruction. Think about what the operation should be in this case, and state your observations:

|  |
| --- |
|  |

Provide your modules, with a suitable testbench here:

|  |  |
| --- | --- |
| Code: | Testbench: |
| Waveform: | |