Lab 5: Arithmetic Logic Unit (ALU)

Purpose

In this lab, you will:
- Design an 8-bit ALU
- Use the Xilinx Core Generator to create pre-defined IP (intellectual property) blocks
- Use VHDL test benches to simulate your ALU before loading it onto the board
- Implement the ALU on an FPGA

This lab is different from the previous exercises in that it is a mini-design project. This gives you more freedom to come up with your own design strategy. The following write-up serves as a guideline to help you design your modules. Because your choice of design may vary dramatically from others’, make sure you explain your design clearly in the lab report.

You will have **two weeks** to complete this lab. In the first week, it is recommended that you finish all the individual functions of the ALU so that in the second week, you can put them together into one ALU module and add the components to test it on the board.

**You must demo your fully-functional ALU by the end of your second lab session.**

---

Background Information

An Arithmetic and Logic Unit (ALU) is a combinational circuit that performs logical and arithmetic operations on a pair of n-bit operands (in our case, A[7:0] and B[7:0]). Unless otherwise stated, you can assume that the inputs A and B are signed, two’s complement numbers when they are presented to the input of the ALU. The operations performed by an ALU are controlled by a set of operation-select inputs. In this lab you will design an 8-bit ALU with 4 operation-select inputs, S[3:0]. Logical operations take place on the bits that comprise a value (known as **bitwise** operations), while arithmetic operations treat inputs and outputs as two’s complement integers. Errors must be detected by the ALU, specifically division by zero; if any occur, enable the Error signal. If an addition results in overflow or a multiplication results in a value that cannot be shortened to 8 bits, enable the Overflow output. The 16 functions performed by the ALU are specified in Tables 1 and 2.
### Table 1: Logical Functions

<table>
<thead>
<tr>
<th>S3</th>
<th>S2</th>
<th>S1</th>
<th>S0</th>
<th>O[7:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A AND B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A OR B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A XOR B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>NOT (A)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>NOT (A AND B)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>NOT (A OR B)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>NOT (A XOR B)</td>
</tr>
</tbody>
</table>

### Table 2: Arithmetic Functions

<table>
<thead>
<tr>
<th>S3</th>
<th>S2</th>
<th>S1</th>
<th>S0</th>
<th>O[7:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A+1 (incrementer)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A-1 (decrementer)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A+B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A-B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A/B (unsigned)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A%B (remainder)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A*B (only lower 8-bits used)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>A≫B or A≪B (logical shift)</td>
</tr>
</tbody>
</table>

Again, the logical operations are bitwise, meaning that for $S = “0001”$,

$$O(1) = A(1) \& B(1)$$

This, of course, means that logical operations operate on the bits only, and ignore the value and sign of the inputs. For arithmetic operations, the ALU should truncate at 8 bits. You may ignore the carry-out in additions and subtractions. Overflow occurs when the ALU result cannot be represented with 8-bits, and serves as a warning to anyone using your ALU. This can occur only in addition/subtraction and multiplication. ERR is defined only for division by zero. The overflow and error signals must remain off for all other cases, including all other operations and correct addition/subtraction/multiplication/division computations.
The logical shift operates on the input value by shifting its bits left or right, filling zeroes in after. In your logical shifter, the A input is the value being shifted, while the B input is the amount to shift. A is treated as an unsigned number (since this is a logical, not arithmetic shifter), while B is treated as a signed number. B represents both how much to shift as well as in what direction to shift; a positive value refers to a right-shift, while a negative value refers to a left-shift. For example, if you have $A = "11011011"$ and $B = "11111111"$, the output should be “10110110”, as A is shifted right by -1 bits with a zero filled in on the right. For a right-shift, this differs from an arithmetic shift. An arithmetic shift retains the sign of the value by filling in zeroes for positive numbers and one’s for two’s complement negative numbers. There is no such distinction for a left-shift. You are asked to implement a logical shifter for the sake of simplicity. However, if you so desire, you may build an arithmetic shifter. If you do, please make a note of it in your lab report.

**Designing the Macros**

Since this is a mini-design lab, we will not require you to use schematics or VHDL explicitly. Instead, for the most part, you may choose either method depending on what you find easier. That said however, for most modules (as for many things in life), there is an easy way and a hard way. Through the course of the lab, we may provide hints as to which one to use, but you should think about what method to use for each module and why.

We do have some restrictions however, as to what you can and cannot use:

1. **You may** use schematic entry for any and all modules, with the following guidelines
   a. You may (and should) use Xilinx library modules such as adders and subtractors
   b. You may not use any ALUs or similar high-level Xilinx library modules
2. **You may** use VHDL for any and all modules, with the following guidelines
   a. You may use logical operators: `and, or, xor, not`
   b. You may not use arithmetic operators: `+, -, *, /`
      (you should Xilinx library modules for these operations)
      i. The only exception to this rule is for two’s complement. You may use the ‘+’ operator for computing the two’s complement of a number **ONLY**.
   c. You may use shift operators, but you may not use them for variable shifting: `≪, ≫`
3. In general, you may reuse modules from previous labs. However, note that your adder (lab 3) and multiplier (lab 4) will not work here as they are unsigned and only operate on four-bit inputs.

We will provide you with a signed multiplier and an unsigned divider. While the multiplier is straightforward, you are not responsible for knowing how the divider works. However, you must test the divider to ensure proper functionality.
For the shifter, a possible design method is to use a *barrel shifter.* A barrel shifter builds a large shifter using smaller shifters. For example, to build a left-shifter that can shift an input up to eight positions, you can use three shifters: \( \ll 1 \), \( \ll 2 \), and \( \ll 4 \). Then, based on the amount you need to shift by (i.e. the B input to your ALU), you can choose to use or not use each of the three sub-shifters. You would need to build both a left-barrel-shifter and a right-barrel-shifter. You then would have to choose the correct shifter based on the input amount/sign (i.e. B). Of course, this is just a suggestion; you may implement the shift operation however you want. Again, as stated above, you may use, for example “A \( \ll 4 \)” but may not use “A \( \ll B \)”.

**Using Your ALU**

Once you build your ALU, you will have to implement an interface for both inputting numbers as well as viewing the output. You will display the results on the four-digit seven-segment display and the LEDs. Specific button/switch mapping is given in the procedure section as always.

1. Take as input two 4-bit 2's complement values from the board's switches. Obviously, the switches will not be able to test all possible situations, but it should be enough to check functionality.  *You will need to sign-extend the 4-bit input values for your 8-bit ALU.*

2. Use the push buttons to select the ALU operation

3. Display the 8-bit result on the right-most three digits of the 7-segment display. Use the left-most digit to display a negative sign for negative results. For positive results, this left-most digit should be off.

4. Also display the 8-bit ALU output using the eight peripheral-board LEDs

5. Use two of the four main-board LEDs for the Overflow and Error signals.
   *NOTE: The main-board LEDs are active-low.*
6. A top-level block diagram is shown below:

![Block Diagram](image)

Figure 1: Overall system, including the ALU and display units.
Pre-Lab

(Done individually)

NOTE: We will be checking the prelabs at the beginning of lab. Any prelab not ready by the beginning of your lab section will not receive credit. All prelabs must be done individually, not in groups.

1. Read the background information above. **It will be covered on the prelab quiz.**

2. Prepare a 1-page block diagram of your ALU. This block diagram should show how you plan to design your ALU. You may use simple blocks to represent any sub-modules (e.g. your shifter module) instead of actual logic. Think carefully about how you should design your ALU. Remember, the better the design, the easier it will be to implement it in lab.

3. Let’s say your boss gave you 48 additional operations to add to your ALU for a total of 64 ALU operations. What part of the ALU would you have to change to accommodate these additional operations?
IMPORTANT: READ BEFORE YOU BEGIN

- Use C:\user (appears as My Documents) as your working directory
  - Do NOT use your ENIAC (S:) drive or a flash drive
  - Do NOT use C:\users (files will be deleted at logoff)
- Save regularly
- At the end of a work session, archive your project from Xilinx
  (Project → Archive) and save this .zip archive to YOUR ENIAC (S:) drive or a flash drive.

You will have two weeks to finish the lab – this week and next week. **You must demo by the end of the second lab session.** If you feel that you cannot finish the lab in the second lab session, you should come in to another lab session or on your own time to work on the lab.

There is no immediate lab report for this lab. Instead, you may have some discussion questions to turn in next week. **You will write about this lab in Report 3, due after you build the RPN calculator in Lab 7.**

As stated earlier, you should finish building all the sub-modules that implement the sixteen operations in the first week. **You should also know how you are going to combine those sub-modules into the main ALU.**
In-Lab Assignment

Your task is to design and implement the ALU using the Xilinx ISE tools and the prototyping board. You will create a project with a top-level schematic implementing the block diagram as shown in the background section. This top-level module will have several sub-modules that you can create using any method you prefer, including schematic entry and VHDL (hint: VHDL is usually simpler). Remember, you must follow the guidelines and restrictions as described in the background section.

As this project is more complicated than earlier projects, it will be important that you be very systematic during the design. Each module should be simulated completely with all errors fixed before proceeding up the hierarchy of your design. Failing to do so will make it difficult for you to debug the system. It is also important that you plan your work for each week to ensure you will finish in time.

Here are some guidelines to help you in your design and implementation process:

- Design the components that implement each of the logical and arithmetic functions of the ALU. Try to reuse the same components for multiple functions, if possible.
- Simulate each component and verify that it works correctly
- Download the multiplier and divider (also available on Blackboard)
- You must simulate the divider module (you do not need to test the multiplier)

(You are expected to complete at least the above tasks in the first week.)

- Combine the macros into an ALU to be placed in the top level schematic
- Make the decoder and the display switch circuit, or reuse from a previous lab
- Make the sign-extenders that convert the switch inputs to the two 8-bit values for A[7:0] and B[7:0]
- Simulate ALL the various modules using testing procedures learned in previous labs. You do not have to simulate the top-level schematic, seven-segment decoders, or any module that you are using from previous labs that you have not changed. You should test the functionality of any Xilinx library module that you are using to make sure that it is working as expected.
- Implement the ALU and test it on the board
- Give a demo to the TA