Lab 1: New Beginnings

Abstract

In this lab, you will be thrown feet-first into the ~~wonderful~~ world of logic design. Our goal for this lab is for you to familiarize yourself with your FPGA, SystemVerilog, and writing testbenches. Your task is to design a basic ALU, which you will then validate in simulation.

Introduction

Welcome to DAV! Hopefully, you'll have attended or watched lecture 1, where we presented an overview of the DAV project and some elements of digital logic. If you are reading this after week 4, you'll also have attended worksession 1, where we helped set up Questa, demonstrated how to create a Quartus project, and walked through how to write a module. Now, it's your turn: you will implement a rudimentary calculator that interacts with the world through your FPGA. We often refer to these circuits as "arithmetic logic units", or ALUs for short; ALUs serve as a fundamental building block for modern CPUs.

As you work your way through this first lab spec, you may encounter difficulties or points of confusion. Please consult the resources linked in the section below, and don't hesitate to reach out to your leads (Bryan and Adrian) for help!

Resources

links to lecture 1, workshop 1

Pin Sheet

Discord: drone? // bryab_

(please spend some time consulting the internet first!)

1. The top module

This module acts as the interface between your FPGA and the rest of your calculator. More functionality will be added to this module as you progress through this lab. In this section, your task is to control the LEDs above each switch: each LED should individually turn on when the switch below it is "on", and vice versa.

1.1 A new file is born

Firstly, create a project in Quartus. If you need help, refer to the slides from worksession 1. Then, create a new file for your top module: a good naming convention to follow would be `"project_name"_top`. In this case, we suggest something along the lines of `miniALU_top.sv`.

Inside the module declaration, define the input and output ports. We want the FPGA to process the **switches as inputs** and produce **outputs on the LEDs**. There are ten switches and ten LEDs, each representing a single binary digit (up/down, on/off). Defining 20 ports won't be necessary! Think about how we may combine the ten input bits into a single port, and how the same may be done with the ten output bits. If you need help with syntax, refer to the worksession slides or the Verilog docs linked in the resources section.

Finally, with your ports defined, we need to assign (wink wink) the output LEDs to our switch inputs. There is a single Verilog statement that may be used to connect the wires. If you can't figure it out, reread the first sentence of this paragraph, or refer to the Verilog docs!

1.2 A bench for testing

We can use a testbench to simulate our module and confirm that it functions as intended. We hope that you'll understand the importance of writing testbenches to validate your designs sooner rather than later, as it'll prove to be a powerful tool in helping debug your capstone project. For now, let's walk through a simple testbench for the switch-to-LED connections we just created.

Start off by creating a new file for your testbench. Testbenches follow the same `module`/`endmodule` declaration, but require an additional directive to specify the time units for your simulation. You may choose to copy/paste the following code to the first line of your testbench file, or read more about the `timescale` directive here.

```
`timescale 1ns/1ns // time units of 1ns with 1ns precision
```

Declare a module (we named ours `miniALU_tb`) that takes **no input ports**, and **output the LEDs**. Additionally, you'll need to create a signal (using `logic`) for your simulated switch inputs in the module. Instantiate your top module (`miniALU_top`, if you've been following our naming convention) as your Design Under Test (DUT), and pass in the switches and LED signals accordingly. We highly recommend connecting ports by name; refer to the Verilog docs if you want to learn more.

Once you've written your test cases, **set your testbench as the top module** in the file hierarchy. You'll first need to run the **Analysis & Synthesis** task for Quartus to synthesize your design. After that, navigate to **Tools > Run Simulation Tool > RTL Simulation** in the Quartus menu to launch your simulation in Questa.

In essence, a testbench is closer to a script than a digital circuit. Inside the testbench, Verilog code is executed sequentially inside an `initial begin` block, with various timings dictated by the use of delays. Keep in mind that these delays are only recognized by Questa, our simulation software, and have no physical meaning in a digital circuit! Worksession 1 covers these ideas in greater detail, so refer to those slides or the Verilog docs for more information.

1.3 A plan for your pins

After verifying your design in simulation, it's time to test it in the real world! Set your top module (`miniALU_top`) as the top module in the file hierarchy, and run Analysis & Synthesis again.

Once that's complete, open the Pin Planner. This interface will list the input and output ports of your top level underneath a large diamond. Ignore the diamond, and refer to the pin sheet to map your ports to the corresponding pins.

Finally, run the **Fitter** and **Assembler** tasks, then use **Program Device** (found in the same menu) to upload the programming file to your FPGA. If you've done everything correctly, flipping on each switch should turn on the LED right above it!

1.4 Checkoff

No checkoff will be needed for this section. :)

2. The ALU

This module will contain the core functionality of your calculator: the **arithmetic logic unit**. We've released a file named `miniALU.sv` that contains some skeleton code to aid you through this section of the lab. For your checkoff, you will be pushing your repository branch to our autograder on Github; we will require that you use the specific ports mentioned in this spec and in the skeleton.

2.1 ALU logic

You will be designing an ALU that can perform two kinds of operations: **addition/subtraction**, and **arithmetic left/right shift**.

As inputs, your ALU will take:

- A **4-bit signal**, `op1`, for your first operand
- A **4-bit signal**, `op2`, for your second operand
- A **1-bit signal**, `operation`, to select between the two types of operation.
    - When `operation` is LOW, the ALU performs addition or subtraction (depending on the value of `sign`).
    - When `operation` is HIGH, the ALU performs an arithmetic left or right shift (depending on the value of `sign`).
- A **1-bit signal**, `sign`, to toggle the sign of the second operand.
    - When `sign` is LOW, perform the positive operation: either addition or a left shift.
    - When `sign` is HIGH, perform the negative operation: either subtraction or a right shift.

A total of ten input bits will correspond to the ten switches on your FPGA; see the figure below for how the switches should be assigned.

As output, your ALU will produce:

- A **20-bit signal** for the result of the operation performed, `result`.

This output should be derived from a block of **combinational** logic. In `miniALU.sv`, we've provided an `always_comb` construct, in which you will describe a combinational circuit. There's a commonly used hardware block (which we covered in lecture) to select between two or more values based on a boolean condition, which you'll need to properly assign your output.

2.2 ALU testing

With your module finished, it's time to write a testbench! You can modify the testbench we created earlier to test the `miniALU` module. Keep in mind that because this module takes in a different set of inputs as your `miniALU_top` module, you'll need to create some more signals.

Because your ALU is a bit more complex than the top module, you'll want to write more than just a couple test cases. While you may choose to program each test case by hand, Verilog provides a handy `for` loop to help iterate through some potential cases. Consider using the template below to simulate a larger number of test cases!

```
for (integer i = 0; i < 15; i = i + 1) begin
     #5; // simulation delay
     /* YOUR CODE HERE */
end
```

Verilog also provides the ability to implement a layer of abstraction in a testbench. For example, let's say we didn't want to manually set the bits for `operation` and `sign` to LOW when we wanted to switch our ALU's behavior to addition. We can define a `task` as follows:

```
task addition_operation()
     #5 operation = 0;
     #5 sign = 0;
endtask
```

Now, every time we call `addition_operation()` inside the `initial begin` block, the `operation` and `sign` bits will be set automatically!

2.3 Checkoff

Push your code to your team's branch on Github! In your repository, you should have a file `.github/workflows/test-config.txt`. In this file, make sure that `miniALU` appears on line 1, and nothing else. Once you've confirmed this, add/commit your changes, then push them to your branch. You should then see a job run under the Actions tab on Github. If the job finishes successfully, you're all good to go! If you need help, please refer to lab 0, or if you're really stuck, feel free to reach out to us.

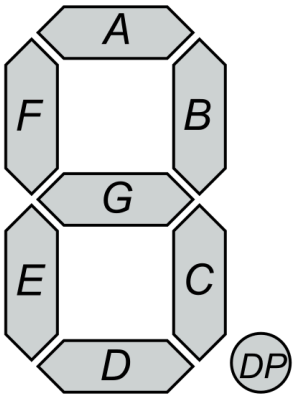
3. CD (calculator display)

These modules will contain the logic that converts the result of your ALU to a human-readable format. Specifically, you'll be implementing an encoder for the six seven-segment displays on your FPGA. This time, we've released two files with skeleton code: `displayEncoder.sv` and `sevenSegDigit.sv`.

3.1 DD (~~designated~~ display driver)

Each seven-segment display digit on the FPGA is driven by 8 bits. You will write a module to **drive these 8 bits**.

The bit that drives each segment in each digit is shown in the visual to the right. The bits used to drive the display are listed in alphabetical order: the LSB (least significant bit), or bit 0, corresponds to segment A; bit 1 corresponds to segment B, and so on. The eighth bit corresponds to the decimal point (DP) segment. Keep in mind that these segments are **active low**: a value of 0 turns each segment on, while a value of 1 turns it off.



As inputs, each display digit will take:

- A **4-bit signal**, `digit`, representing which numeral (0-9) should be displayed.

As output, each display digit will produce:

- A 8-bit signal, `displayBits`, representing which segments on the seven-segment display should be on (remember the display is active LOW!).

The logic of this module should be fairly straightforward: there are ten possible `digits` as input, each mapped to a unique string of `displayBits` as output. In such situations, the `case` construct (similar to a switch-case statement in many coding languages) is useful: you will compare the input to each of the 10 digits and produce **the correct `displayBits`** if they match, or a **required "default" output** if none match.

3.2 ED (encoding display)

The seven-segment display driver you wrote takes in a 4-bit input to determine which digit should be displayed. You will now write a module to **compute the correct 4-bit numbers and connect them to your display driver** from the 20-bit result of your `miniALU`.

You'll notice that we are asking you to split a 20-bit signal into six 4-bit signals. Think about why that is - note that we are converting from a binary (base 2) representation to a decimal (base 10) system.

As inputs, the decoder will take:

- A 20-bit signal, `result`, representing the number computed by the `miniALU`.

As outputs, the decoder will produce:

- An array of six 8-bit signals, `displayBits [0:5]`, that drives the seven-segment displays on the FPGA.

Logic-wise, think about what operators are available in Verilog; you'll need to perform some operation to extract each 4-bit `digit` from your 20-bit result. These `digits` will then need to be passed into six unique instances of `sevenSegDigit`, which will help you generate the array of 8-bit signals needed to drive the displays.

3.3 Testing the display

As always, we recommend that you write a testbench to confirm that your design functions as intended. This time, we won't be walking you through the process.

3.4 Checkoff

We trust that you'll write a testbench as we told you to, so there'll be no checkoff for this section.

4. Putting everything together

With your modules complete, it's now time to connect them in the top module.
In `miniALU_top`, instantiate your `miniALU` and `displayEncoder` modules. Create signals for the inputs and outputs of each module, and connect them accordingly:

- The `op1` input to `miniALU` should be assigned to the first four (9-6) switches.
- The `op2` input to `miniALU` should be assigned to the next four (5-2) switches.
- The `operation` input to `miniALU` should be assigned to the next (1) switch.
- The `sign` input to `miniALU` should be assigned to the last (0) switch.
- The `result` output from `miniALU` should be connected to the `result` input to `displayEncoder`.
- The `displayBits[0:5]` output from displayEncoder should be connected to a corresponding output of the top module.
    - If you've been following this spec closely, you'll notice that such an output doesn't exist yet. Add one to the declaration of your `miniALU_top` module now. :)

4.1 Checkoff

After following these steps, everything should be in place! Run the **Assembler** task in the left sidebar, then use the **Program Device** menu to program your FPGA. For your checkoff, come find your leads during one of our lab hours to show us your working mini ALU. Be prepared to answer some questions about the design choices you made (we'll be asking these questions from the lecture material and lab spec)!