Lab 3: Integer Division

ECE 212: Digital Circuits II – Fall 2024

Abstract. Integer division is implemented in hardware using an algorithm that iteratively checks if the divisor can fit inside the current remainder. If true, the divisor is subtracted from the remainder and the quotient bit is set to 1. In lecture, this algorithm was presented in pseudocode and via a hardware schematic. In this lab, you will use both behavioral and structural modeling in SystemVerilog to implement division in hardware.

Objectives.

- Implement a hardware-based division algorithm
- Test the operation of a circuit using a testbench

Materials and Equipment.

- Xilinx Vivado

Deliverables.

- In-person lab demonstration (per team)
- Code submission via GitHub repository

Part 0: Set Up a Vivado Project and a Shared GitHub Repository

To begin the lab, set up a new Vivado project for Lab 3 and import the SystemVerilog files provided with this assignment on Moodle. You can reference Lab 0, Part 1 for a more detailed description of these steps.

1. Create a new folder named “Lab03” inside of your Git repository for this assignment. You should save all of your source files inside of this folder and/or the entire Vivado project.

2. Create a Xilinx Vivado project inside of the Lab01 folder made in the previous step. Import the provided constraints file (`lab03_constraintsxdc`) and SystemVerilog/VHDL modules (`*.sv`, `*.vhdl`).

Part 1: Implement a Ripple-Carry Adder

Integer division can be implemented using a *subtract-and-shift approach*. The divisor, **`B`**, is subtracted from current version of the remainder, **`R`**. If the result is positive, then the corresponding quotient bit is set to 1 and the reminder is updated to **`R-B`**. Otherwise, the corresponding quotient bit is set to 0. Afterwards, the next bit of the dividend, **`A`**, is shifted into the remainder before the next round. This process is repeated until all quotient bits are determined. The circuit for 4-bit addition is shown in Figure 1. Each box encloses a single round of division, which produces a quotient bit and a remainder

for use in the next round. The shift of the remainder is achieved via the diagonal connections between each round. The pseudocode for a single round of division is given in Figure 2.

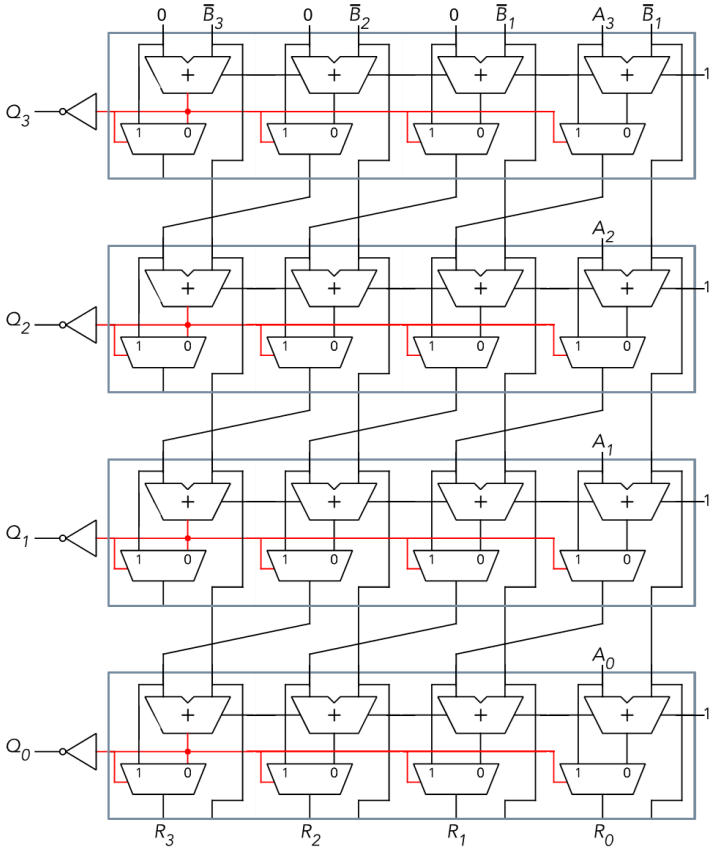

Figure 1. 4-bit Divider Circuit Schematic

```
Given the current remainder, R, and the divisor, B:
     Compute D = R - B

     If D[3] == 1:
        The result is negative; The divisor does not "fit"
        into the current remainder
        Output R for the new remainder
        Output 0 for the quotient bit

     Otherwise, if D[3] == 0: output R - B

        The result is positive; The divisor can "fit"
        into the current remainder
        Output D for the new remainder
        Output 1 for the quotient bit
```

Figure 2. Divider Row Algorithm

In this part of the lab, you will follow two distinct programming approaches to implement the divider. First, you will use a strictly structural approach to implement a division round shown in the Figure 1 schematic. Then, you will use a behavioral approach and implement the division round algorithm shown in Figure 2 with high-level constructs like if-statements and arithmetic operators (*e.g.,* +, -). Writing the module each way will help you draw connections between their operation.

1. Create a new SystemVerilog module, **`div_rnd_structural`**, that takes a 4-bit remainder and a 4-bit divisor, and outputs a new 4-bit remainder and 1-bit quotient according to the schematic in Figure 1. Your module should use *only structural modeling* in SystemVerilog.
    a. Structural modeling is the lowest level of abstraction in SystemVerilog, where you are restricted to specifying connections between gates and modules. Your implementation should only consist of full-adder modules, 2-input MUX modules, and NOT gates (using either a gate instantiation or the **`!`** operator).

2. Create a new SystemVerilog module, **`div_rnd_behavioral`**, that takes a 4-bit remainder and a 4-bit divisor, and outputs a new 4-bit remainder and 1-bit quotient according to the algorithm in Figure 2. Your module should use *behavioral modeling* in SystemVerilog.
    a. Behavioral modeling captures the functionality of a circuit rather than connections between gates. With behavioral modeling, you can use if-statements in an always_comb block and arithmetic operators to implement your circuit.

3. Create a new SystemVerilog module, **`div_4b`**, that takes a 4-bit dividend and a 4-bit divisor, and outputs a 4-bit quotient and a 4-bit remainder. Your implementation should use two instantiations of **`div_rnd_structural`** and **`div_rnd_behavioral`**, each and connect them following the structure shown in Figure 1.

4. Create a testbench to test that your 4-bit divider functions correctly. Record your test cases to include in the results section of your lab report.

5. Run your testbench and ensure that the module's output correctly meets the specification. For more information about creating testbenches, refer to Lab 0.

Part 2: Synthesize Your Circuit for the Nexys A7 FPGA

In Part 2 of this lab, you will connect your module to the switches and seven-segment display on the Nexys A7 FPGA, according to the layout shown in Figure 3. Specifically, your circuit will use **`SW[7:4]`** as the dividend for the division operation and **`SW[3:0]`** as the divisor. The provided module **`result_display`** is used to show the quotient and remainder on the seven-segment display.

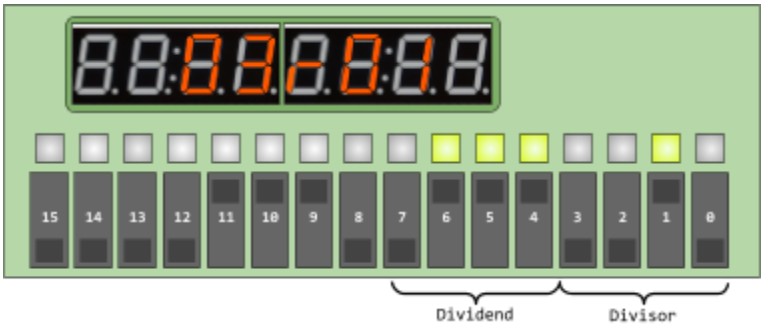

Figure 3. Nexys A7-100T Input/Output Configuration for Lab 3

1. In the top-level module, `lab03_top.sv`, instantiate your 4-bit divider module and connect it to the corresponding switch inputs. The final results should be connected to the provided **`result_display`** module.

2. Generate a bitstream and program the FPGA. More information on how to connect and program the FPGA board can be found in Lab 0.

3. Check that the circuit meets the specification. Demonstrate your working implementation to the instructor.

Part 3: Push your Code to GitHub

Push your changes to your new Git repository.

1. Add your new (or modified) files to your Git repository using the "`git add`" command.

2. Commit your changes using the "`git commit`" command and then push your changes to the cloud using the "`git push`" command. For more information about using Git, please refer to Lab 1.