Rodrigo Becerril Ferreyra
CECS 361 Section 01
Lab 4
31 October 2020

1 Introduction

The purpose of this lab assignment is to practice implementing finite-state machines (FSMs) and modularization of a design. Specifically, the objective of this lab assignment is to implement a Euclidean division algorithm.

In short, for $A, B \in \mathbb{Z}$ and $B \neq 0$, there exists two unique integers Q and R such that

$$A = BQ + R \tag{1}$$

and

$$0 \leq R < |B|. \tag{2}$$

In this lab, we are given an algorithm and digital implementation to find Q and R (called the quotient and remainder, respectively); our job is to construct this implementation from a given digital circuit:

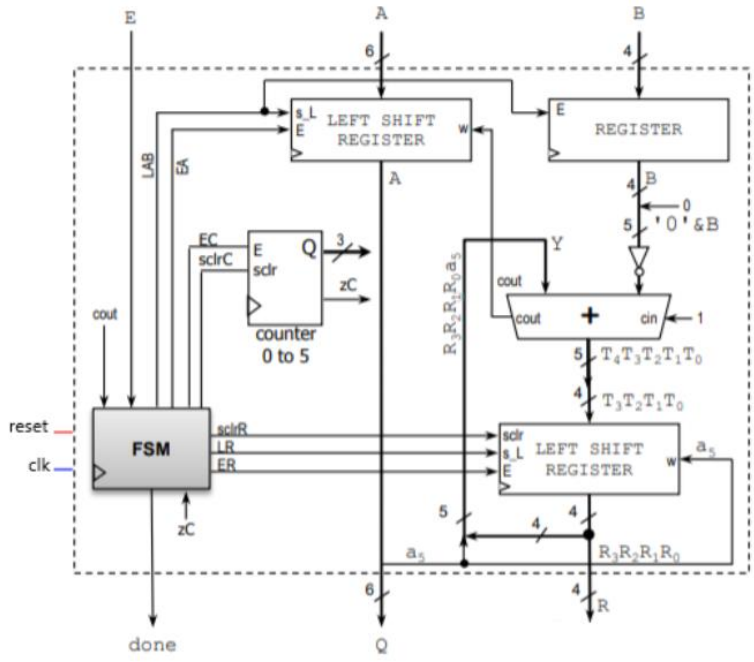

Figure 1: Architectural schematic. (c) Oakland University, ECE department, used without permission.

This circuit incorporates various smaller digital circuits, such as various types of registers, counters, and a combinatorial addition circuit. The circuit takes in `A` and `B`, and outputs `Q` and `R`, along with a `done` bit which is high when the algorithm is finished.

2 Implementation

To start, we were given two modules to use: one six-bit left-shift loadable register, and one four-bit left-shift loadable register with synchronous clear. We were required to create the following circuits in order for the implementation to work:

- a four-bit loadable register.
- a modulo-6 counter.
- a combinatorial five-bit full adder.
- a controller modified by a finite state machine (FSM).

After all the modules were constructed, we were tasked with connecting them in accordance to Figure 1. All sequential circuits share `clk` (clock) and `rst` (reset) values unless otherwise stated.

2.1 Loadable, left-shift registers

These circuits were given in the project description. Their functionality is as follows: when their enable input `E` is high, the value held by the register is shifted over one place to the left, and the value at `w` is written to the LSB, all on the rising edge of the clock; when `E` is low, the register retains its value. The four-bit register also has a synchronous clear input `s_L` (as opposed to the asynchronous clear input `rst`) which clears the contents of the register on the rising edge of the clock.

2.2 Four-bit loadable register

This module was the easiest of all to construct. It is simply a four-bit register with an "enable" or "write" input (named `E`), and its functionality is as follows: if `E` is low, the register holds its current value; if `E` is high, the value on its input `D` gets copied to its output `Q` on the next rising edge of the clock.

2.3 Modulo-6 counter

The functionality of this circuit is as follows: internally, the circuit consists of a two-bit register that holds the current count. When the enable input `E` is high, the value of this register will increment on the rising edge of the clock; if `E` is low, it will keep its value. The register has a `sclr` (synchronous clear) which clears the value of the register on the rising edge of the clock.

When this register reaches the value of 5, the output `zC` is set, and the next value the register takes on is 0. Any other value of the register clears `zC` (the value of the register is also an output (named `Q`), but is not used in this design).

2.4 Combinatorial five-bit full adder

This full adder takes in two five-bit inputs `A` and `B` and a one-bit input `cin`, and adds them together in the output vector `{cout, Sum}`, where `Sum` is five bits long and `cout` is a scalar. This design allows the `cout` to be differentiated from the five-bit result of the summation, allowing it to be separated and sent different directions.

2.5 Controller

Last but not least, the controller, modeled by a FSM, was the most complex part of this lab. The controller can have three states (if modeled by a Mealy machine) or six states (if modeled by a Moore machine). Below is a comparison of the state diagram of the FSM as modeled by different types of machines.

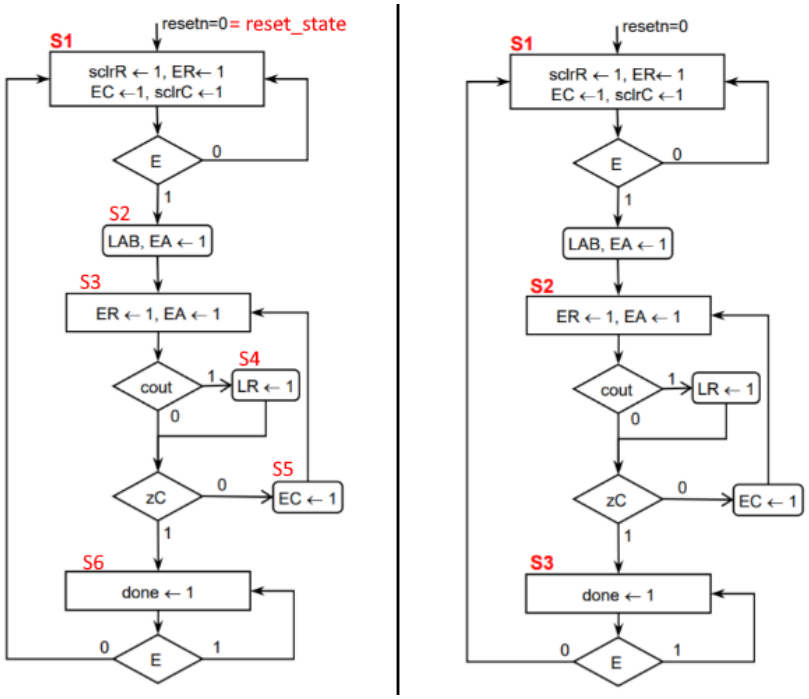

Figure 2: A Moore machine (left) vs a Mealy machine (right). (c) José Aceves & Amin Rezaei.

Its outputs drive the inputs to almost all of the other circuits (excluding the adder). Its inputs are driven by a combination of user inputs and internal wires from the outputs of other circuits.

At first, to simplify the design process, I designed the controller using an Moore machine model, as I find them easier to implement. It was very helpful in choosing the format to describe the controller in the Verilog language. However, I ran into trouble due to the fact that a Moore machine requires double the amount of states with respect to a Mealy machine, and therefore you need double the amount of bits to encode all states. My trouble stemmed from the fact that I was using two bits to encode my states instead of the required three. Once that was fixed, the controller ran as planned. Below are the block diagrams of the defective Moore machine, working Moore machine, and Mealy machine implementations of the controller.

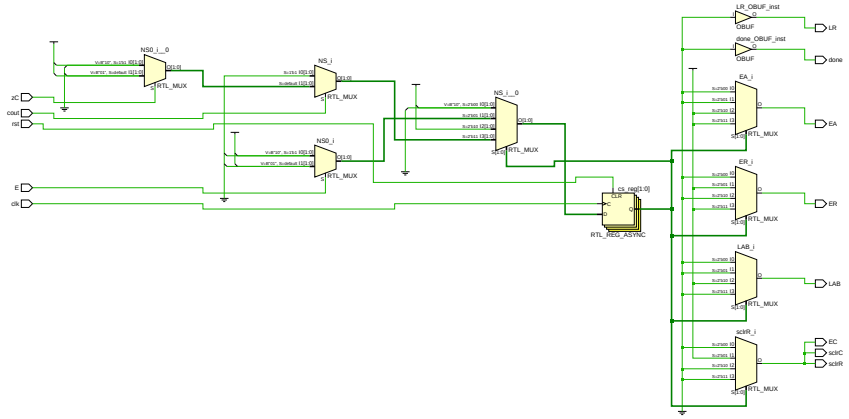

Figure 3: Defective Moore machine implementation.

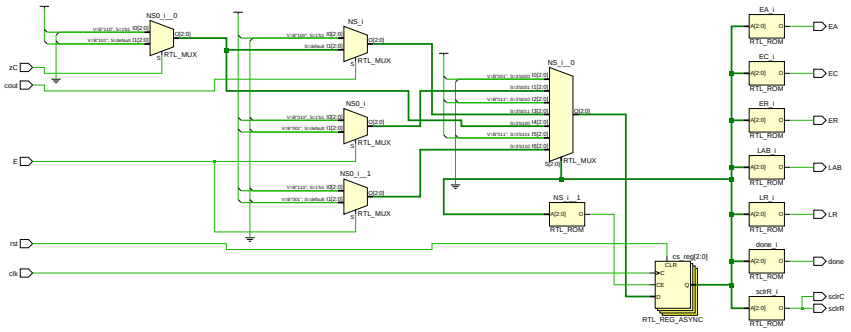

Figure 4: Working Moore machine implementation.

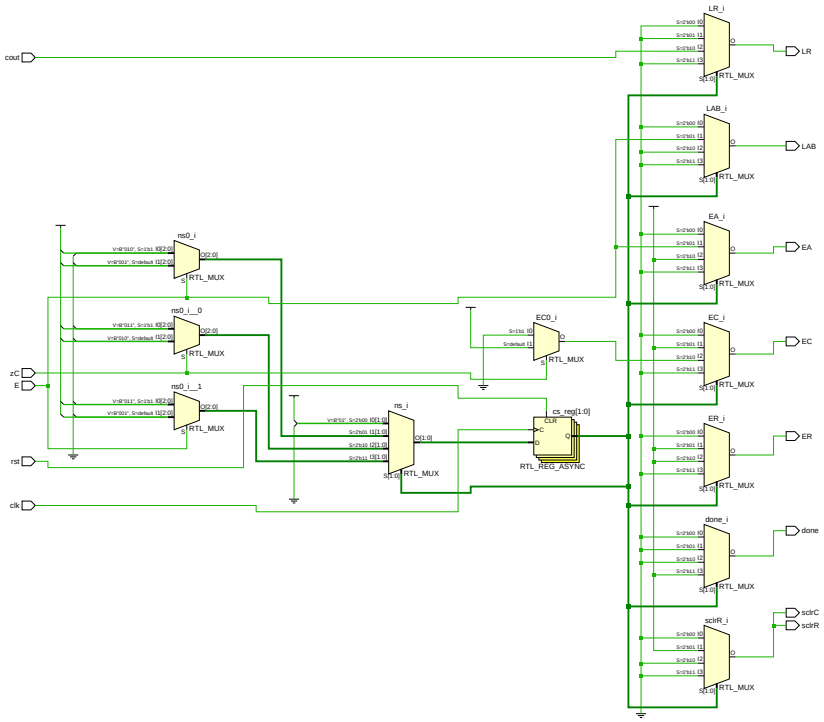

Figure 5: Mealy machine implementation.

(The above diagrams are vector images, so feel free to zoom in if viewing on a computer.) Note that Figure 3 has various outputs stuck at zero (connected to ground); this is a clear indication that certain states cannot be reached (namely those with state encodings greater than 3). The working Moore machine in

Figure 4 fixes this issue, but introduces `RTL_ROM` blocks. Figure 5 gets rid of these blocks and has an easy-to-follow flow. In the end, I decided to use the Mealy design, due to the fact that it did not include any `RTL_ROM` blocks.

2.6 Top-level module

The top-level design was as simple as following the design layed out in Figure 1; in the case of implied connections such as `cout`, like-named wires are connected. Below is a diagram of the final top-level circuit; feel free to compare it to the circuit in Figure 1.

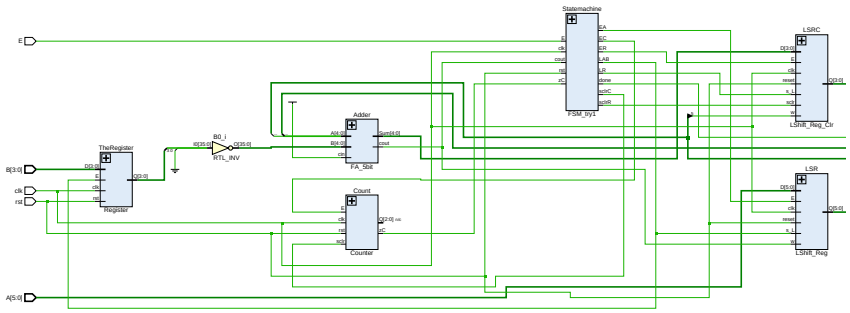

Figure 6: Top-level diagram.

3 Testing and Verification

Testing was performed at each step of the designing process: after a circuit was completed, it was verified for correctness. Below is a compilation of waveform outputs of all the circuits that were created. Note that the given circuits were not tested and taken to be correct.

3.1 Four-bit loadable register

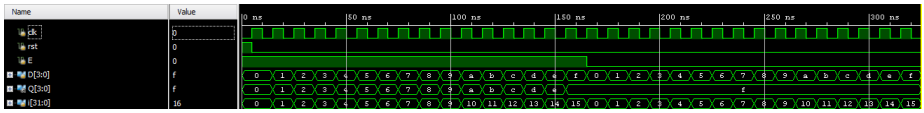

Figure 7: Loadable register testing.

In this test, the enable input is set for 16 clock cycles, and as one can see, the value gets copied from `D` to `Q` on the rising edge of the clock, as a register should. However, when `E` is cleared, the register retains its value. This is the desired behavior.

3.2 Modulo-6 counter

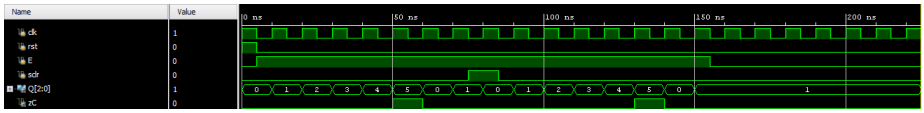

Figure 8: Modulo-6 testing.

In this test, the enable input `E` is turned on, allowing the counter to count. As specified, when the value of the counter `Q` reaches 5, `zC` is set, and the next value of `Q` is 0. At about 75 ns, `sclr` is asserted, clearing the contents of the register on the rising edge of the clock. Later in the test, `E` is cleared, stopping the counter and keeping the value constant. This is the desired behavior.

3.3 Combinatorial five-bit full adder

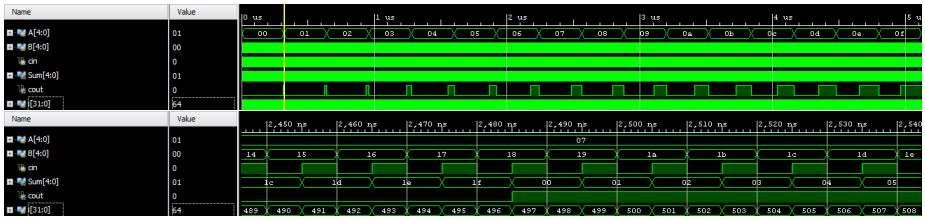

Figure 9: Five-bit adder testing.

The first image in Figure 9 shows the full waveform in a compressed format, while the second image shows a portion of this waveform for clarity.

In this test, the values of `A` and `B` were looped through iteratively, which is guaranteed to test all possible values for these two inputs. The output is then verified with the expected result. This is the desired behavior.

3.4 Controller

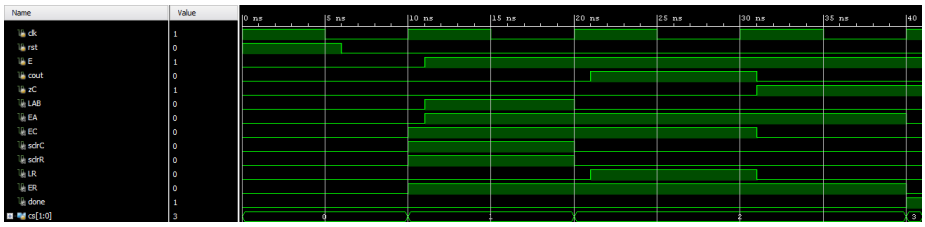

Figure 10: Controller testing.

In this test, all distinct combinations of possible outputs were tested; note that the bottommost signal in this waveform represents the internal state of the controller, is neither an input nor an output, and is only included for clarity.

The test follows the Mealy model FSM as described in Figure 2, and has been verified to be consistent with all outputs.

3.5 Top-level module

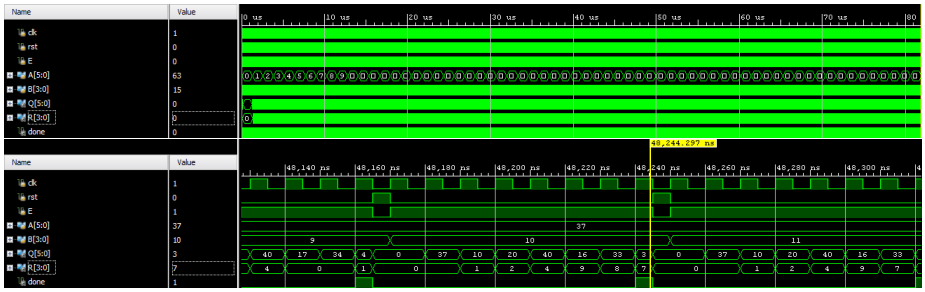

Figure 11: Top-level module testing.

The first image in Figure 11 shows the full waveform in a compressed format, while the second image shows a portion of this waveform for clarity. Note that results are only valid when `done` is set.

Extensive self-checking testing was used for this test. This means that each result is checked with its expected value; therefore, if any calculated value is

different from the expected value, the test will print a warning to the console. No negative warning was printed to the console during the testing of this module.

The way the testing was performed is as follows: the inputs `A` and `B` are assigned iteratively, making sure that no value is not assigned. Next, the test waits until the `done` bit is raised, because as stated earlier, the result is not valid unless `done` is high. At this point, the result is checked with the expected result: the desired quotient is calculated using `A/B`, and the desired remainder is calculated using `A%B`. The test then asserts a reset and moves on to the next test. In Figure 11, the result shown is

$$37 = 10 \times 3 + 7,$$

a true statement.

It is interesting to note what happens when the divisor is zero; in this case, Euclidean division is not defined. What we get is the following:

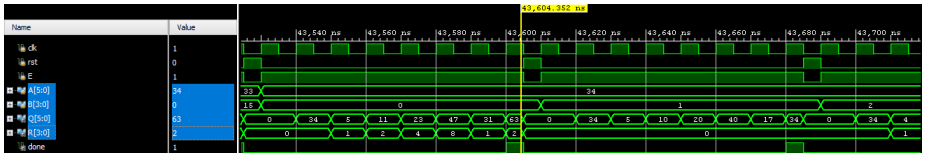

Figure 12: Division by zero example.

As we can see, the result we are getting is

$$34 = 0 \times 63 + 2$$

, which is not a true statement.

4 Conclusion

This lab has taught me that, unlike implementing a multiplication circuit, implementing a division circuit is more complex, as it needs to be sequential instead of combinatorial, and it requires more internal parts to do its work. The circuit also needs to take more time to calculate its result (several clock cycles worth of time, depending on the values being calculated), instead happening almost instantly as is with the multiplier circuit.

5 Media

Below are images of the board running the test depicting

$$37 = 10 \times 3 + 7.$$

The two images are the same, but the LEDs are visible in the first image, and the switches are visible in the second image.

Figure 13: Real-time board test images.