Charles Zhang
19 October 2021
COM SCI M152A, Lab 5

Lab 1: Sequencer Report

1 Introduction

UART communication is a process involving asynchronous communication between two UART devices. In this context, asynchronous implies that there is no clock signal used to synchronize the signal between transmission and reception. Instead, the transmitting UART sends data from its `Tx` pin, adding start and stop bits to the data packet, allowing the UART receiving that data through its `Rx` pin to detect the beginning and end of a transmission. This signal operates at a frequency known as the baud rate, a term used to measure the speed of bit transmission, measured in bits per second. For the purposes of this lab, we will use the UART on the Nexys 3 board to transmit data to the UART console on the lab machine.

In this lab, we focused on the running, understanding, and modification of a small-scale FPGA project, with which the user can encode `PUSH`, `ADD`, `MULT`, and `SEND` instructions to interface with a series of 32-bit registers using switches and buttons on the Nexys 3. These register files would need to be able to print to a UART console in order to display their contents. To begin, we will modify the existing testbench code in order to clean up the UART output, such that the hex-encoded contents of a register print on a single line rather than four lines. From there, we must write code that allows the testbench to read in instructions from a file instead of using hard-coded instructions. With those modifications complete, we can then write a file containing binary-encoded instructions such that the Fibonnaci sequence is printed out. We will then take a closer look at individual components of the code that implements this system, namely the clock divider, button debouncer, and register files, using simulations to analyze their behavior.

2 Design Description

The first task in modifying the testbench is to clean up the UART output in the simulation, so that a register's contents are displayed horizontally, instead of vertically. In the context of the code, this means that the file `model_uart.v` should output a single line per register file. In the original code, the `rxData` register would take in 8 bits and then print the ASCII translation of those 8 bits followed by a carriage return. Once the register contents had been printed, a newline followed. This resulted in the contents of a register being printed across 4 separate lines. **To clean this up so that each register would print on a single line, we introduced a new register `fullUART`. This register would store the contents of `rxData` over 4 iterations, essentially reading in the register file's contents and storing them. It would intentionally ignore the carriage returns that had broken up the old implementation's output. These contents could then be printed using `$display` upon receiving the newline character that indicated that the register contents had concluded.**

The other task involving the testbench was to alter the `tb.v` file so that instructions could be read in from a file, not just hard-coded in. **The original code in the file made use of the `tskRunPUSH`, `tskRunADD`, `tskRunMULT`, and `tskRunSEND` tasks provided in the module. Each of these tasks took in a set of integer inputs that represented switch values, placed them into registers as binary-encodings, and then called on the `tskRunInst` task, which would then use those registers to simulate switch values and actually perform the operations on the register files.** Since the goal was to use binary-encoded instructions from an external file, it would be possible to bypass the operation-specific tasks and just pass each byte to `tskRunInst`. **To accomplish this, the `$readmemb` built-in was used to read the contents of a file into an array of registers called instructions. This array could take 1024 entries, as defined by the spec, and each entry contained an 8-bit value, representing a single instruction. Our code would then read in the first line of the file by accessing the first element of this array in order to see how many instructions the file contained. It would then use this value to iterate through the array, calling `tskRunInst` on each line to run the encoded instruction.**

3 Simulation Documentation

Clock Divider

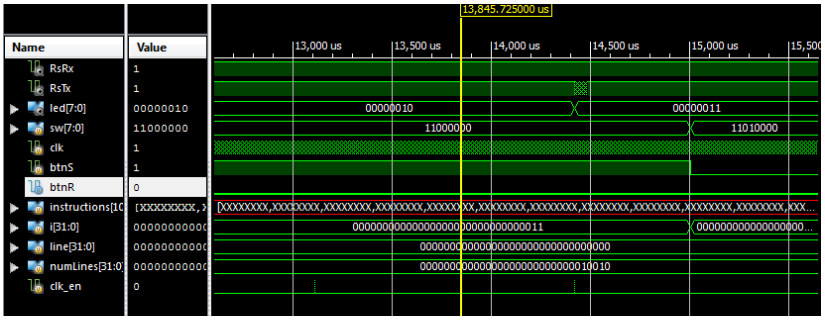

Figure 1: Waveform displaying 2 occurrences of `clk_en`, with the first posedge occurring at t_1 = 153,355,255ns, and the second occurring at t_2 = 154,665,975ns

Using the posedges in the waveform diagram above, we can calculate the period as follows:

$$P = t_2 - t_1$$

$$P = 154,665,975\text{ns} - 153,355,255\text{ns} = 1,310,720\text{ns}$$

Therefore, we know **the period of `clk_en` is 1,310,720 nanoseconds**. This period can also be derived from the code, where a 17-bit counter is used as a clock divider. This implies that the period of `clk_en` is 2^{17}, or 131,072, clock ticks. Since each clock tick is defined as 10 nanoseconds, we arrive at the same 1,310,720 nanosecond period.

Within each period, the `clk_en` signal is high for exactly 10 nanoseconds, or a single clock tick. Taking this interval along with the period, we can calculate the duty cycle of `clk_en` as follows:

$$D = \frac{T}{P} \times 100\%$$

$$D = \frac{10\text{ns}}{1,310,720\text{ns}} \times 100\%$$

$$D = 7.63 \times 10^{-4}\%$$

Here, we can see **the duty cycle of `clk_en` is 7.63×10^{-4}%. During the intervals where `clk_en`'s signal is high, we can see that `clk_dv`'s value is 0**. A schematic of the interaction between `clk_en`, `clk_dv`, and `clk_en_d` can be created as follows:

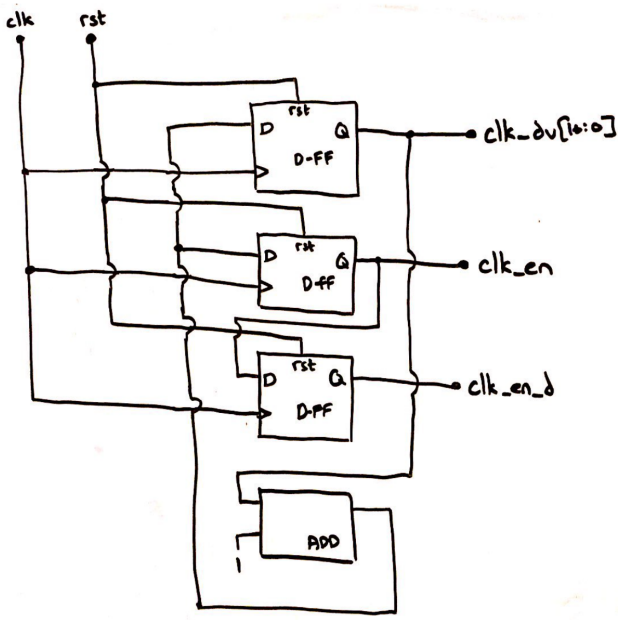

Figure 2: Schematic showing the relationship between `clk_en`, `clk_en_d`, and `clk_dv`

Debouncing

As calculated previously, `clk_en`'s signal is high for 10 nanoseconds out of its 1,310,720 nanosecond period. During this clock tick, incoming instructions are allowed to be saved to `inst_wd`. This is the program's way of ensuring a clean signal is read in. However, since `clk_en` is only high for a single clock tick, and combinational logic ensures that we must read the contents of `inst_wd` the clock tick *after* the instruction is saved to it, we need to use a different signal. To accomplish this, we use `clk_en_d`, which will be high exactly one clock tick after `clk_en` is. **By offsetting `clk_en_d` by one clock tick, it allows us to read in the instruction while `clk_en` is high, and then check if it's valid in the following clock tick, while `clk_en_d` is high**.

We cannot choose to make `clk_en`'s duty cycle 50%. Doing so would mean that, for half of the period of `clk_en`, its signal would be high. **This would allow instructions to be read in and executed for that entire period of time. Since the clock works on the order of**

nanoseconds, this would make it impossible to detect individual button presses, leaving the user unable to submit their intended instructions.

Shown below are the waveforms depicting the relationship between various signals involved in the button debouncing process:

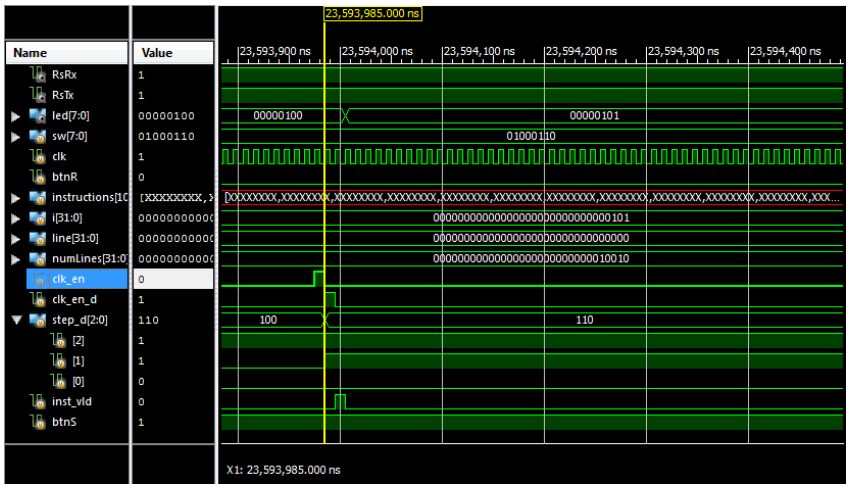

Figure 3: Waveform displaying the interaction between `clk_en`, `step_d[1]`, `step_d[0]`, `btnS`, `clk_en_d`, and `inst_vld`

Drawn below is a schematic, visualizing the logical relationship between the signals detailed above:

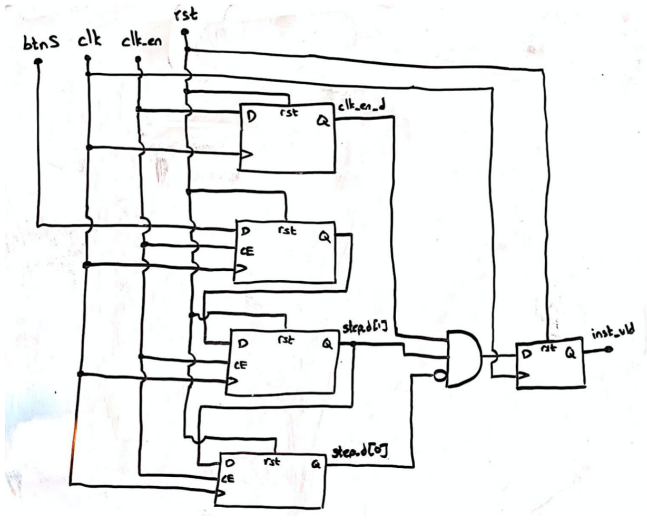

Figure 4: Schematic showing the relationship between `clk_en`, `step_d[1]`, `step_d[0]`, `btnS`, `clk_en_d`, and `inst_vld`

Register File

The line of code in which register files receive a non-zero value is `rf[i_wsel] <= i_wdata`. **Since this code both stores a value for later use, and is protected by an always block that triggers on the clock's posedge, we know this is sequential logic**.

The lines of code in which register values are read out from the register file are of the form `assign o_data_a = rf[i_sel_a]`. **The lack of reliance on a clock signal and the use of wires and the `assign` keyword tell us that this is done through combinational logic**. In order to implement this, we know we need to be able to select a specific output from a set of inputs, telling us we would need to use a multiplexer. Since the array `rf` has four entries and we're selecting one of them, **we would need to make use of a 4 x 1 multiplexer**.

Drawn below is a schematic of the register file:

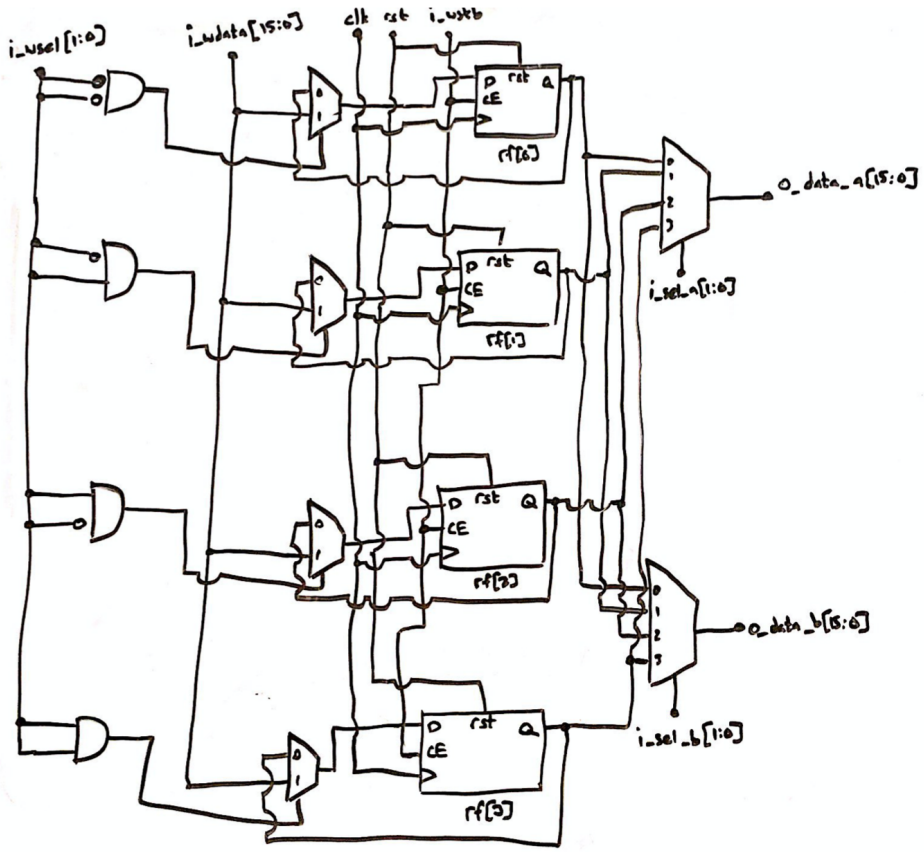

Figure 5: Schematic showing the relationship between inputs and output in the register file

Shown below is the waveform capturing the first time `rf[3]` is set with a non-zero value:

Figure 6: Waveform displaying the first instance where `rf[3]` is non-zero

4 Conclusion

In this lab, we experimented with and analyzed a simple sequencer program. In doing so, we learned the basic mechanics of clock dividers, button debouncing, register files, and UART protocol. We also worked with simple testbench code, requiring us to learn about simulation output and file I/O in Verilog. We did so by cleaning up console output by parsing through UART transmissions and making use of the `$readmemb` built-in to read data from a file.

My biggest problems concerned familiarizing myself with Verilog data types. My unfamiliarity with arrays resulted in my file I/O code being overly complicated and buggy. Ultimately, we were able to resolve these issues with the assistance of online documentation. In addition to this, attempting to visualize Verilog modules as a whole to draw schematics proved to be quite challenging. I ended up having to break down each module piece by piece to gain an understanding of the appropriate logic elements to use, and was eventually able to come up with a design.