EEL 5721 – Reconfigurable Computing

# Lab 1 – Finite State Machines

---

By

Nathan Jessurun

September 16, 2020

# Lab 1 - Fibonacci Calculator

## Introduction

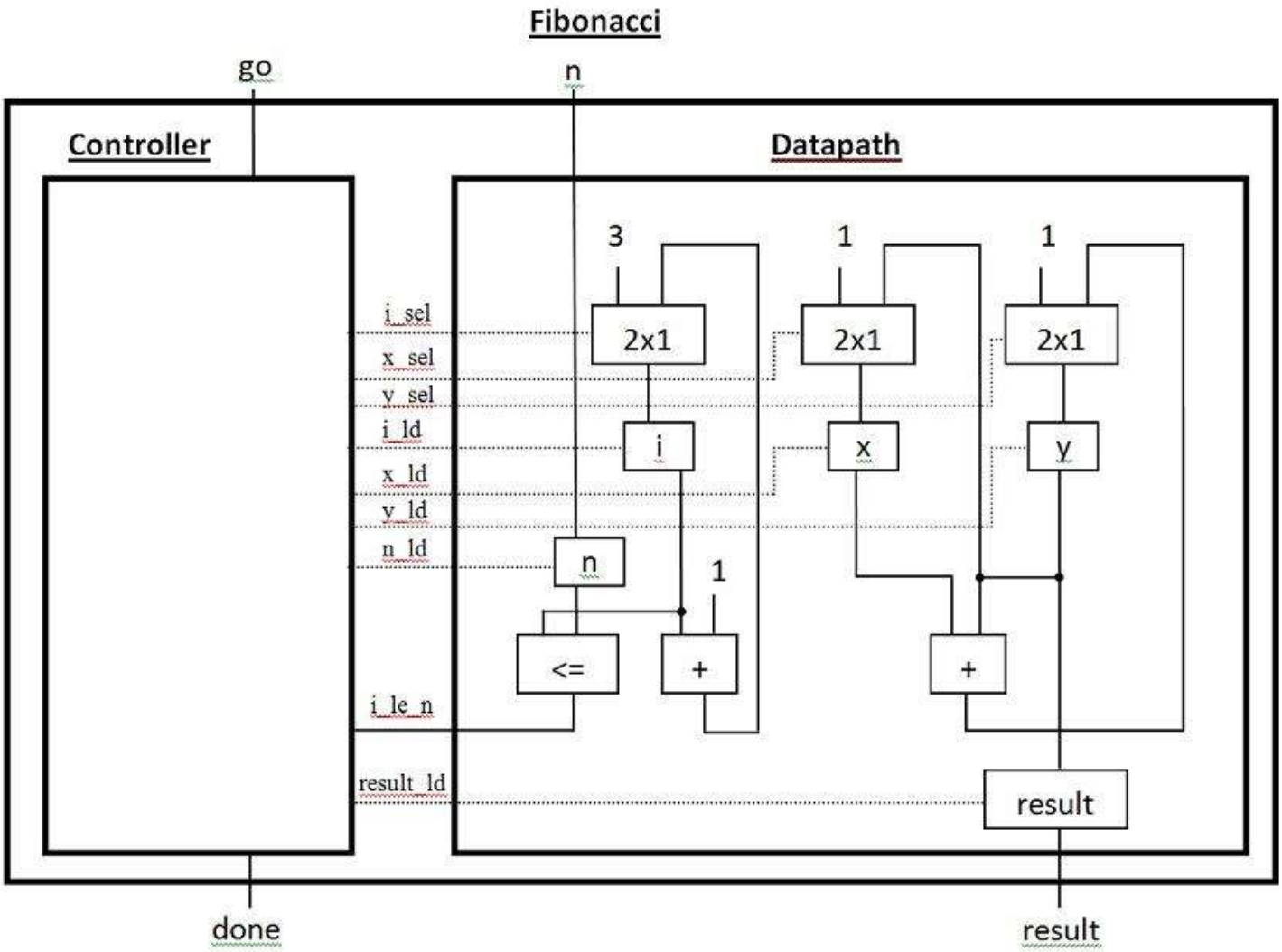
In this lab, you will be implementing a circuit in VHDL that calculates Fibonacci numbers. Here is the pseudocode that describes the behavior of the circuit:

```
Input: n (specifies the Fibonacci number to be calculated)
Output: result (the nth Fibonacci number)

// reset values (add any others that you might need)
result = 0;
done = 0;

while(1) {

  // wait for go to start circuit
  while (go == 0);
  done = 0;

  // store input in register
  regN = N;

  // do the Fib calculation
  i = 3;
  x = 1;
  y = 1;
  while (i <= regN)
  {
    temp = x+y;
    x = y;
    y = temp;
    i ++;
  }

  // output result and assert done
  result = y;
  done = 1;

  // wait for go to clear so circuit doesn't constantly repeat
  while (go == 1);
}
```

You will turnin in your project using e-learning. You should submit both the code and a screenshot of a working simulation. Please do not turn in ISE or Vivado files, only vhd files and screenshots are necessary.
SUBMISSION FORMAT: Your project should be submitted in a zipped folder with your UFID as the name. Inside that folder should be a separate directory for part 1 and 2. Include a readme file in the base directory with your name. If there is anything the grader should be aware of, include that in the readme file also.

## Part 1 - Modeling a controller and datapath (FSM+D)

In this part of the lab, you will implement a controller and datapath in VHDL to implement the functionality of the code shown above. A block diagram of the circuit is shown below:




The circuit has 4 inputs: *go*, *n (8-bit)*, and *clock/reset* (not shown). There are also 2 outputs: *done* and *result (8-bit)*. The circuit should initially wait until go is asserted (active high), and then read in input *n*, control the datapath as needed to generate the nth fibonacci number, and then store the result in the result register that is connected to the output *result*. After a result has been generated, *done* should be asserted (active high) until *go* is set back to 0 and then reasserted (i.e., until the circuit starts again). The circuit should not start calculating another number until go is reset to 0 and then set to 1.

Inside of the <= entity, use the UNSIGNED type (do not use it on the port). See the previous tutorials for examples of how to use UNSIGNED. For all other signals, use the STD_LOGIC or STD_LOGIC_VECTOR type.

The dotted lines between the controller and datapath represent control signals need to control the functionality of the datapath. The control lines for the muxes handle selecting the appropriate input. The control lines for the register specify when values should be loaded. The solid line *i_le_n* is an output from the datapath that the controller uses to determine if the code has completed.

Your project should have the following entities, implemented in the specified way:

1. 2x1 mux - behavioral description
2. 8-bit register with load input - behavioral description (used for n, y, x, and y in the datapath)
3. 8-bit adder - behavioral description
4. 8-bit less-than-or-equal comparator - behavioral description
5. Datapath - structural description using the above entities with the connections shown in the figure (clock and reset are not shown)
6. Controller - behavioral description of an FSM that controls the datapath. Note that you will have to determine what the FSM should be. (clock and reset are not shown)
7. Fib - structural description that connects the controller and datapath.
8. Testbench - A testbench that tests at least 3 different input values for the top-level Fib entity. Although this is the only testbench that is required, you might want to create other testbenches for the other entities.

Note: You can ignore invalid values of n, such as values < 1 or values that result in overflow from the 8-bit registers.

**Part 2 - FSMD**

In this part of the lab, you will implement another circuit for the same pseudocode, but this time using an FSMD as explained in class and in my VHDL tutorial. The main difference here is that you are not explicity specifying the datapath, and instead are specifying operations that are executed during each stage of the FSM.

For part 2, you will only need 2 entities:

1. Fib - behavioral FSMD model of pseudocode. The inputs and outputs are the same as in part 1 (go, n, clk, rst, done, and result). The entity should behave the same way as in part one, simply using an FSMD instead of an FSM+D.
2. Testbench - a testbench for the Fib entity that tests at least 3 inputs.

### Extra Credit

Synthesize the circuits from part 1 and part2 using Vivado (or ISE), and compare their areas. Use any device you want as a target, and compare the area using slices or LUTs. Explain the difference in area, if any. This comparison does not have to be detailed. It is mainly intended to get familiar with the synthesis toolflow.

### Common Questions

- When doing the FSM+D, you will likely see a warning during simulation that looks similar to this:

```
# ** Warning: NUMERIC_STD."<": metavalue detected, returning
FALSE
#    Time: 0 ps  Iteration: 0
```

  As long as this warning occurs at time 0, you can safely ignore it. This warning occurs when a "metavalue" (e.g., 'U', 'X', '-', anything other than '0' and '1') is used as input to an operation. For example, adding two uninitialized vectors is undefined. In these cases, the operation returns a default value (e.g., sometimes false, sometimes X, etc.).

  In general, you should avoid these warnings. However, with a structural architecture it isn't always possible (or at least easy) to avoid the warnings at time 0, because at the beginning of the simulation there might temporarily be an uninitilize value used as input that immediately gets updated when the simulation proceeds.

  So, the general rule is that you can ignore these at time 0. There are other rare situations where you can ignore these warnings, but you need to be 100% sure that they are harmless.

- You do not need to worry about overflow on the output. Only inputs that will not cause overflow will be tested. Make the output width the same as the input width.

# 1 Working Simulations

`true_fib` represents the output of a fibonacci function on input $n$ to assist in verifying the FSM output.

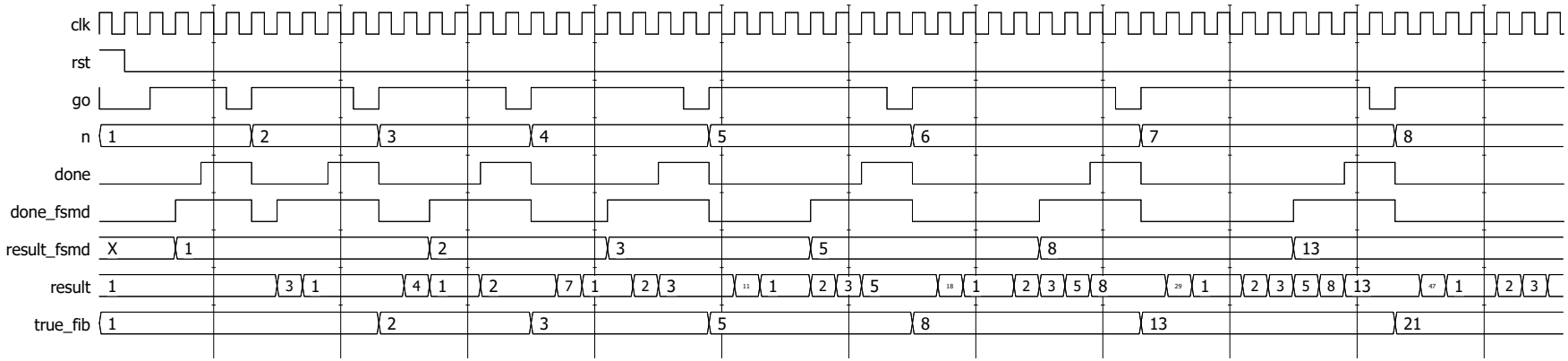
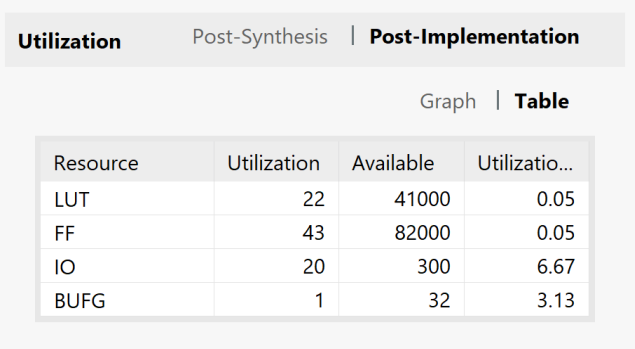



Figure 1: 'true_fib' represents the fibonacci result for input $n$ to assist with verifying the FSM result

## 2 Design Size Comparison

The number of LUTs required for FSM+D and FSMD designs are shown in Figure 2. Both implementations are similar in their underlying logic, so it makes sense their LUT sizes should be close to each other (22 vs. 32). However, the FSMD path used registers more liberally than the alternative approach. Rather than specifying muxes and register loads to determine behavioral logic, I chose to make $x$, $y$, and $i$ all come with additional `next` counterparts to greatly simplify the amount of vhdl code. Hence, it makes sense this design would feature a larger footprint.

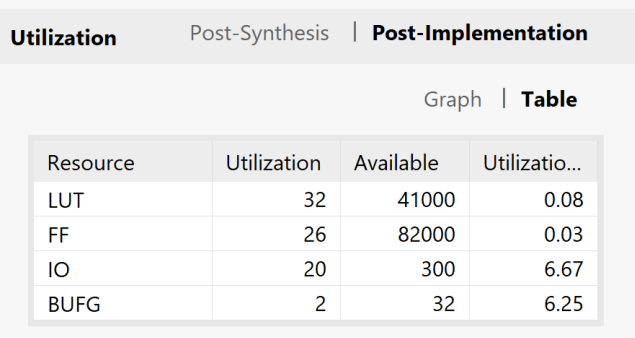


**Utilization** Post-Synthesis | **Post-Implementation**

Graph | **Table**

| Resource | Utilization | Available | Utilizatio... |
|---|---|---|---|
| LUT | 22 | 41000 | 0.05 |
| FF | 43 | 82000 | 0.05 |
| IO | 20 | 300 | 6.67 |
| BUFG | 1 | 32 | 3.13 |

(a)



**Utilization** Post-Synthesis | **Post-Implementation**

Graph | **Table**

| Resource | Utilization | Available | Utilizatio... |
|---|---|---|---|
| LUT | 32 | 41000 | 0.08 |
| FF | 26 | 82000 | 0.03 |
| IO | 20 | 300 | 6.67 |
| BUFG | 2 | 32 | 6.25 |

(b)

Figure 2: Comparison of LUT sizes for FSM+D (left) and FSMD (right) implementations of a Fibonacci calculator.

## 3 Code Listings

Per instructions, all code files are zipped along with the project report.