Project 1: Determining Full

Ben Heard

March 7, 2021

Abstract

One of the more interesting problems that you'll need to solve for the first project is knowing when you've captured all of the operands for a computation. The operands A, B, C, and D are allowed to appear in any order so you will need to track when each has been captured to know when you may complete the computation, present the result, and assert valid.

In this lab you will design, implement, and test a portion of the project that will track inputs being captured.

1 Getting the Content

This has been covered a number of times in the previous labs. Please refer to them for reference. Once you pull the latest from your repository you will have a new folder called lab_005 and, in it, there will be a README file, modelsim.ini file, and this PDF.

2 Input/Output

NOTE: This won't be a direct drop in to your project. Although, you should be able to integrate it with you other implementation easily.

The above being said, the inputs should look familiar.

1. **rst_n**: Active low synchronous reset
2. **clock**: Freerunnning clock

3. **op**: A 2-bit indication as to which operand to capture
4. **capture**: Capture indication
5. **full**: Indication that all 4 operands have been captured

The four inputs behave the the same as in the project. The output, however, is a signal that may pass from the project datapath to the project controller to indicate that all four operands have been captured. For this lab, you're just to make that an output of the overall system.

3 Requirements

Your system shall conform to the following requirements.

- Reset to a known initial state on the active low synchronous global reset, **rst_n**.
- Track operands that are being presented to you system on **op** along with the control signal **capture** to know when each operand has been captured.
- Assert the **full** indication once each operand has been captured.
- Once **full** has been asserted for a clock cycle, the system should clear tracking and be ready to accept another set of operands.

4 Block Diagrams

4.1 Top Level Block Diagram

The following the top level block diagram for the system.

4.2 Second Level Block Diagram

The start of the second level block diagram can be seen below. This looks a little different than the ones that we've been putting together so far. The

Figure 1: Top Level Block Diagram

requirements are that the output show when the operands have all been captured. The requirements also say that your system must clear that indicator so that it is ready to accept a new set of operands.

Those two requirements together mean that the output for the system, generated by the datapath, is also needed by the controller to make decisions about what to do next. Therefore, it is both an output of the system and an internal signal that connects to the controller.

Finally, note that this is not a complete second level block diagram. There will be other signals between the datapath and controller. Specifically, signals that indicate to update that an operand has been captured and a signal that will clear that indication.

5 Approaches

There are at least three approaches to tracking when operands have been captured. (There are certainly many more but I will only cover three here). You may choose to design your system using any of these approaches or any other that you come up with. These are just some starting points.

5.1 Flag

One way to track operands is to implement a flag for each of the operands. This is done with a flip flop that, when not enabled, retains its current value. When enabled, it takes on the value of 1. Here, a mux feeding the input

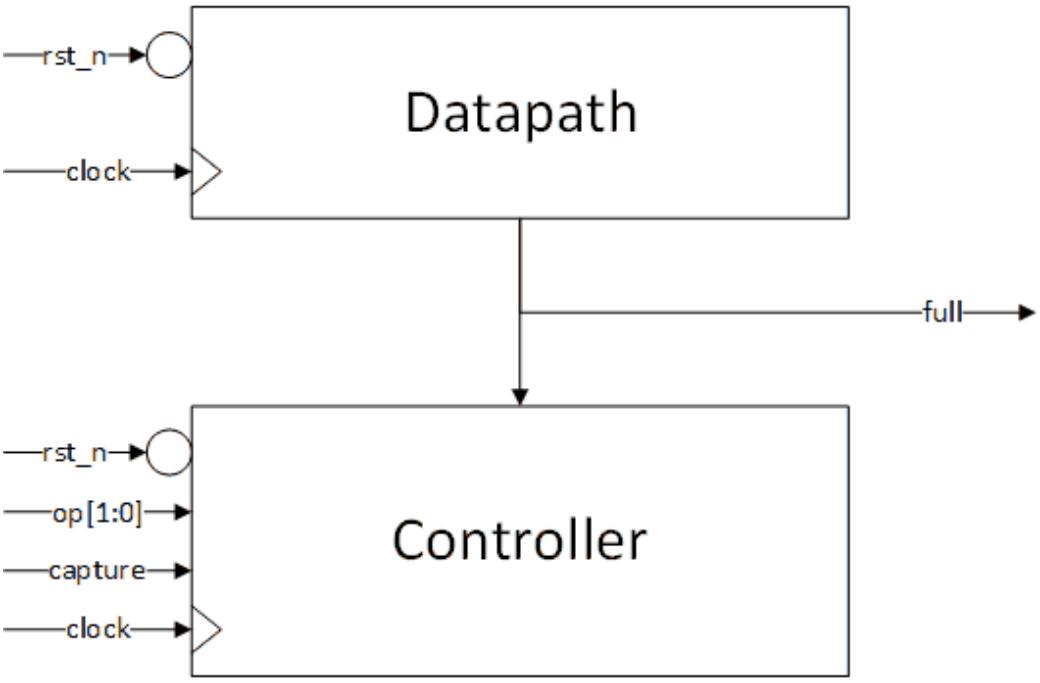

Figure 2: Second Level Block Diagram

logic of the flip flop selects between the existing value and a hard-coded 1'b1. You will need to get the enable logic correct and implement additional functionality to clear the flag.

5.2 Counter

Another way to track is to simply count each time that an operand is captured. This works especially well in this application because I guarantee that I won't send an operand more than once. If it's possible to send an operand more than once then you would need to qualify whether you can already seen an operand before incrementing the counter. That would bring you back to the flag requirement.

5.3 Encode in Controller

This implementation doesn't require a datapath since you encode the captures in the controller itself by moving states when you see a capture. Since

the operands may appear in any order you need a large number of states to encode the captures. The state diagram below is the just the start of the controller.

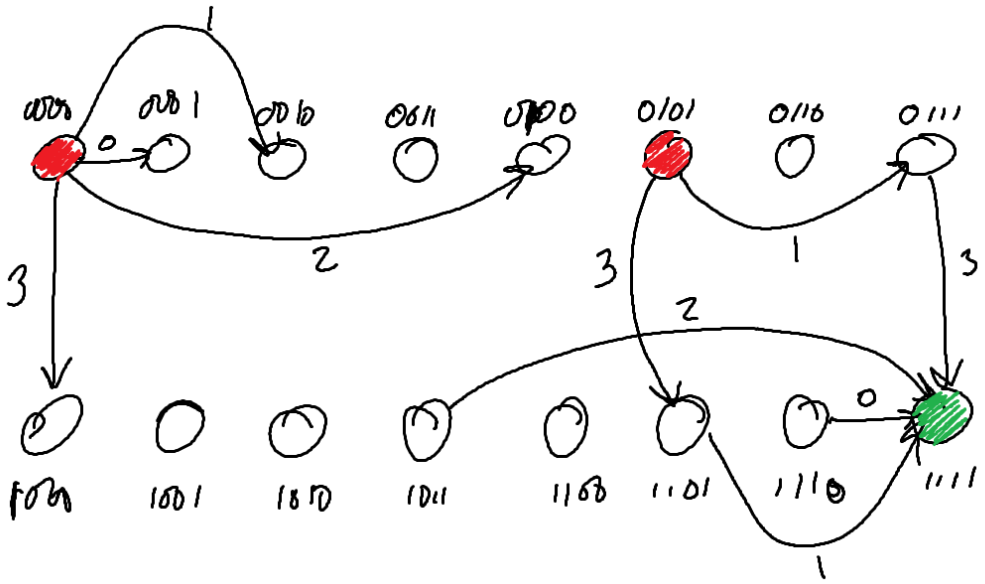

Figure 3: Encoded in State Diagram

You can see the transitions out of the red colored states. For example, if you are in state 4'b0000 that means that you haven't seen any operands. Depending on which you see with capture, you move to one of 4 other states. Not show is the arc where if you weren't capturing anything, you would state in that state. If you are in 4'b0101 there are only two next states, 4'b1101 or 4'b0111. Otherwise you would state in that state.

You can also see transitions into the final state, 4'b1111, from the possible states that had 3 of their operands set. To finish the design you would need to work out all of the other arcs in the system.

One thing to note about this implementation its similarity to the flag implementation. The flag implementation uses 4 flip flops, one for each operand, to track what has been captured. This implementation requires 16 states to encode the possible combinations. Those 16 states also require 4 flip flops.

6 What to Do

Design, implement in Verilog, and test using a Verilog testbench a system that tracks four operands being captured. Once the operands have been captured, assert a full indication and then clear the tracking to be able to track a new set of operands.

Note that you are not actually storing any data (no operands for A, B, C, and D), just some indication that they have been captured. Commit your source to your repo and push. Record the SHA for the commit in the Lab 5 assignment.