Simulation

Rev. 1: Fixed name of output signal (to "detected") in list of test vectors.

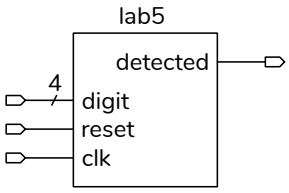
Introduction

In this lab you will design and simulate a sequence detector.

As discussed in the lectures, a sequence detector can be implemented in different ways. One way is to use a shift register to store previous inputs. Another is to count the number of consecutive correct digits. You may use any implementation that you wish.

Requirements

The sequence detector has one-bit **`reset`** and **`clk`** clock inputs, a 4-bit **`digit`** input, and a one-bit **`detected`** output:



`detected` should be asserted when a the last four digits of your BCIT ID are present on **`digit`** on successive rising edges of **`clk`**. **`detected`** should remain asserted for one clock cycle. Your sequence detector should reset itself if **`reset`** is asserted on the rising edge of **`clk`**.

Your test vector file should include 15 lines with one test vector per line. Each test vector should have three values: the input values of **`reset`** and **`digit`**, and the expected output value of **`detected`**. Your testbench should generate **`clk`**. Use the following test vectors:

1. A test vector with **`reset`** asserted.
2. The sequence of digits 1, 2, 3, 4. In this case **`detected`** should not be asserted since no student's ID ends in 1234.
3. The correct sequence of digits for your BCIT ID with an additional (incorrect) digit inserted before the last digit. For example if you ID ends in 4321 you might use the sequence 43221 in your test vector.
4. The correct sequence (the last four digits of your ID). For example, 4321.
5. An additional test vector with **`detected`** set to 1. This should cause an error message to be printed.

The sample simulation code applies the inputs from each test vector and waits for the next falling edge (i.e. after a rising edge) before testing the DUT output. Thus the expected output in each test vector should correspond to the expected output for that input.

Procedure

Write a module that implements the sequence detector module described above and a testbench that applies test vectors read from a **`.csv`** file. You should be able to use the example testbench in the lecture notes with a few appropriate changes. Remember to follow the course coding guidelines, including adding the appropriate comment at the beginning of each file.

Your testbench should print:

- Each test vector as it is read from the file.
- The state of your sequence detector module (even though it is *not* an output of your module). This state will depend on your design and might be a count value or the values in a shift register.
- An error message if there are mismatches between the expected and actual outputs. See the example testbench for an example.

You can create the **`.csv`** file containing the test vectors using a text editor such as Notepad. You can also use a spreadsheet and export the test vectors to a **`.csv`** file[1].

[1] But don't use a UTF-8 (Unicode) output encoding.

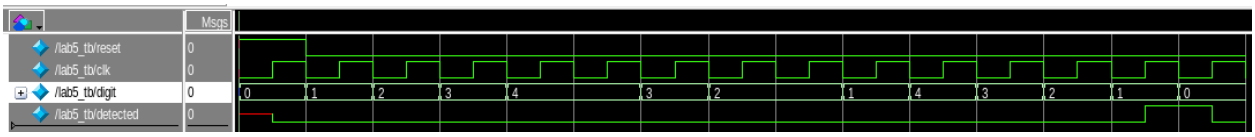


Figure 1: Simulation waveforms (Modelsim screen capture).

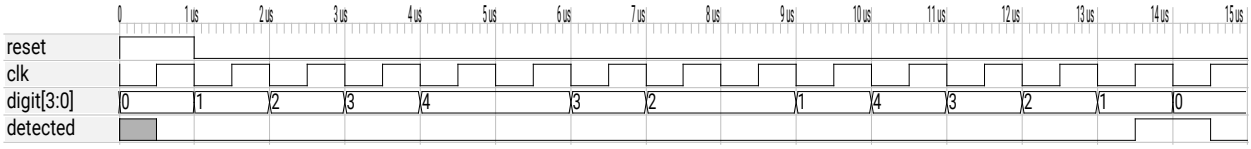


Figure 2: Simulation waveforms (GTKWave).

Follow the procedure in the *Software Installation and Use* document and the video on the course web site to create a simulation project, add the file(s)[2] with your modules to the project and compile them. Copy the test vector file to the project folder. Add the **`reset`**, **`clk`**, and **`detect`** signals to the Wave window. Run the simulation. The Transcript window should show any messages generated by the testbench and the Wave window should show the signal waveforms.

Report

Submit a PDF file to the appropriate Assignment folder that includes the following:

- Listing(s) of your DUT and testbench modules.
- A screen capture of the simulation waveforms similar to that in Figure 1 or Figure 2. Set the display radix for the **`digit`** waveform to decimal[3].
- A screen capture of the Transcript window showing the messages generated by running the simulation. For example:

```
run -all
# test vector=1, 0,0
# DUT state = ffff
# test vector=0, 1,0
# DUT state = fff1
# test vector=0, 2,0
# DUT state = ff12
# test vector=0, 3,0
# DUT state = f123
# test vector=0, 4,0
# DUT state = 1234
# test vector=0, 4,0
# DUT state = 2344
# test vector=0, 3,0
# DUT state = 3443
# test vector=0, 2,0
# DUT state = 4432
# test vector=0, 2,0
# DUT state = 4322
# test vector=0, 1,0
# DUT state = 3221
# test vector=0, 4,0
# DUT state = 2214
# test vector=0, 3,0
# DUT state = 2143
# test vector=0, 2,0
# DUT state = 1432
# test vector=0, 1,1
# DUT state = 4321
# test vector=0, 0,1
# DUT state = 3210
# ***Error:  0, 0, 1
# test vector=0, 0,1
# ** Note: $stop    : /home/edc/202410/lab5.sv(99)
#    Time: 15 us  Iteration: 2  Instance: /lab5_tb
# Break in Module lab5_tb at /home/user/202410/lab5.sv line 99
```

Appendix - Simulation Not Terminating?

The documentation for the system task **`$fscanf`** is the System Verilog standard which is available on the course website under Content / Resources / Verilog / IEEE-1800-2012. It's a huge document so you'll want to search for the string **`$fscanf`**. The return value is:

> *The number of successfully matched and assigned input items is returned in code; this number can be 0 in the event of an early matching failure between an input character and the control string. If the input ends*

[2] You may put the DUT and testbench modules in different files or in the same file.

[3] Right-click on the waveform name and select Radix / Decimal

before the first matching failure or conversion, EOF (-1) is returned.

The value assigned to `n` in the always block is monitored by the `wait` in the initial block. When the end of the file is reached, `n` is set to -1 and the simulation `$stops`.

If the simulation is not terminating, then it's likely that `$fscanf` is returning 0. Check this by `$display`'ing `n` each time through the loop.

If this is the case, then it's likely due to a mismatch between the format ("control") string supplied to `$fscanf` and what appears in your `.csv` file. A common reason is that your format string includes commas and your file does not, or vice-versa. Another possibility is that your file includes characters that cause the pattern match to fail. An example is a Unicode Byte Order Mark (BOM) at the start of a file that makes the file look OK in a text editor but cannot be read by (Unicode-unaware) `$fscanf`. To resolve such problems, try creating your `.csv` file "by hand" using a text editor such as Notepad[++].