Department of Electrical & Electronic Engineering
Imperial College London

2nd Year Laboratory

Experiment VERI: FPGA Design with Verilog (Part 2)

(webpage: www.ee.ic.ac.uk/pcheung/teaching/E2_Experiment /)

PART 2 – Counters and FSMs

1.0 Learning Outcomes

Part 2 of VERI teaches you:

- how to design different types of counters and timers;
- how to use the Modelsim simulator to verify the correct function of your design and the use of testbenches;
- how to predict the maximum operating clock frequency of your circuit sequential circuits;
- how to design some useful timing and counting components for later part of Experiment VERI.

1.1 Experiment 5: Designing a Counter

Step 1: Create the project for an 8-bit counter

- Create in your directory a folder named part_2.
- Click **file>New Project Wizard**, and create project **ex5** and top level file **ex5_top**. Then click **Finish**.
- Create the Verilog file: "**counter_8.v**" which contains your design in Verilog. I suggest you use convention of using **"_n"** to indicate the number of bits in a module.
- Click **File > New …** and select Verilog as the new file. An edit window will appear.

Step 2: Enter the Verilog specification of the 8-bit binary counter

- Enter the Verilog module as shown below (next page). Although you can miss out the comments, I recommend that you to retain them because the code is deliberately verbose in order to explain the meaning of the Verilog language.
- The line **`timescale 1ns / 100ps** tells the system to use 1 ns as the unit time step with a time resolution of 100ps.
- Make sure that you fully understand this Verilog code before proceeding to the next step. Save the file as **counter_8.v**. (I recommend that you use module name as the file name to avoid confusion.)

Step 3: Enter the Verilog specification of the 8-bit binary counter

- While is opened in the Editor window, click **Project > Add Current File to** Project, then click **Project > Set as Top Level Entity**. This command tells Quartus that this module is the top-level of your design.

> Normally we use …**_top.v** as the top-level module, which connects to physical pins of the FPGA. However, for this experiment, the counter module is verified through simulation. So we don't need to create pin connects. The "Set as Top Level Entity" is very useful if you want to use the simulator to verify different modules in a large design. You can move up and down the module hierarchy and verify them from the lowest level up.

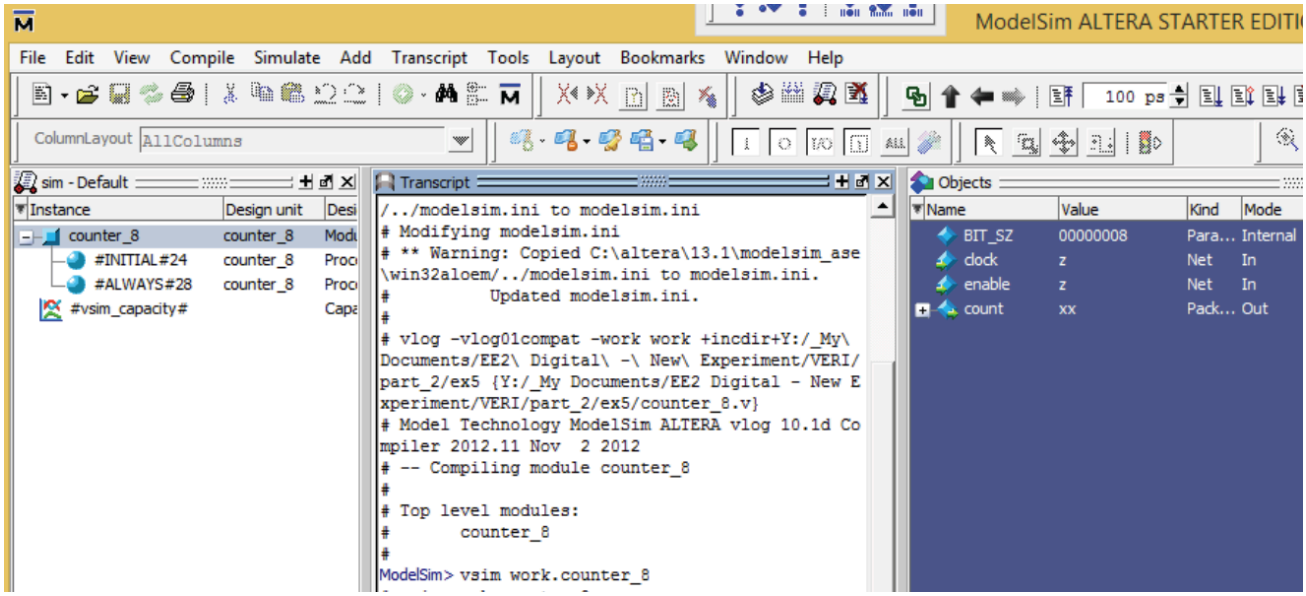
- Click **Processing > Analyze Current File**. This is the fastest way to check if this .v file has any syntax error.
- Then Click **Processing > Start > Start Analysis and Synthesis**. This takes the current Verilog module (and all other modules that it uses if any), and produce a register-level model of your design ready for register-transfer level (RTL) simulation. Unlike full compilation, this step does not require pin assignment and other device specific steps, but is sufficient for you to simulate the circuit as specified in Verilog.

Verilog code: 8-bit counter
(Note that the first character on line 1 before 'timescale' is a backquote ` - not easy to find on many keyboards!)

```
`timescale 1ns / 100ps     // unit time is 1ns, resolution 100ps
//-------------------------------------
// Design Name: counter_8
// Function : an 8-bit synchronous counter with enable input
//-------------------------------------

module counter_8 (
  clock,        // clock input
  enable,       // high enable counting
  count         // count value
);

//-------- Declare ports ---------

  parameter  BIT_SZ = 8;
  input  clock;
  input  enable;
  output [BIT_SZ-1:0]  count;

// count needs to be declared as reg
  reg [BIT_SZ-1:0]  count;

//---- always initialise storage elements such as D-FF
  initial count = 0;

//---- Main body of the module --------

  always @ (posedge clock)
      if (enable == 1'b1)
         count <= count + 1'b1;

endmodule   // end of module
```

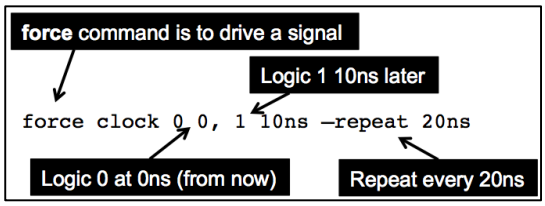
Step 4: Simulate the binary counter

- Click **Tools > Run Simulation Tools > RTL Simulation**. This command starts up Modelsim simulator programme as a separate process. Now you have entered the Modelsim environment.
- Click **Simulate > Start Simulation** …. Then select **work -> counter_8** from the popup window. This tells Modelsim to simulate this module.
- Note that Modelsim provides several windowpanes. The most important is the Transcript pane – this is where you enter commands[1] to drive the simulator. The wave pane is where results are displayed as waveforms. You are recommended to un-dock this pane as shown below so that it is in a separate window and spans the whole width of your monitor. Finally, there is the object pane, which shows all the signals (objects) of your design.

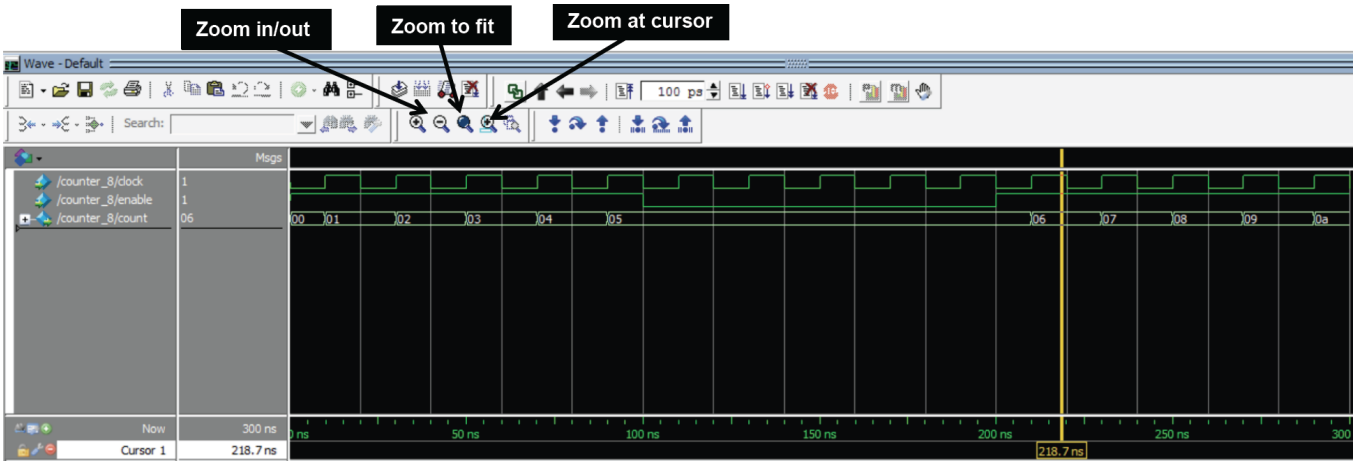


[1] Modelsim uses a scripting language known as **Tcl** to control how it is driven. You only need to learn Tcl if you want to do advance stuff with Modelsim for your personal interest.

Step 5: Add waveforms to the Wave window and drive signals



- In the transcript window, enter two commands: "**add wave clock enable**" and "**add wave –hexadecimal count**". This will add these signals as waveforms in the wave pane and show count values as hexadecimal.
- Now we want to drive **clock** with a 50MHz symmetrical signal. To do this, enter:
- Enter: "force enable 1" to enable the counter.
- Enter: "run 100ns" to run the simulator for 5 clock cycles (5 x 20ns = 100ns).
- You will see the waveform pane showing the counter counting from 0 to 5. Now force enable low and run for another 100ns. Then high again and run for 100ns.



- Click on the waveform put a cursor at a specific time for inspecting the signal values. The icons above the waveforms (as labeled) allow you to zoom in and out of the waveform. Try this yourself.

Step 6: Create a Testbench as a DO-file

- Interactively specifying the driving signals is very tedious and prone to error. Therefore the preferred method is to create a **"do"** file which is a text file containing a sequence of commands (as you have previously entered in the transcript window).
- Click **File > new > source** and select new "**do**" file. Then enter the command lines as shown on the right. Then save this as "**tb_counter.do**".

ment/VERI/part_2/ex5/simulation/modelsim/tb_counter.do - Default

```
Ln#
 1  add wave clock enable
 2  add wave -hexadecimal count
 3  force clock 0 0, 1 10ns -repeat 20ns
 4  force enable 1
 5  run 100ns
 6  force enable 0
 7  run 100ns
 8  force enable 1
 9  run 1000ns
10
```

- Delete all signals from the wave window, and enter command

  vsim> restart
  vsim > do ./tb_counter.do

- This should provide exactly the same waveform results as in step 5. However, the **.do** file can be reused and modified far easier than typing them into the transcript window. It acts as a simple form of a **test-harness** (or **testbench**) for your design. Generally speaking, you must produce testbenches for all your designs instead of using interactive means to test your circuit. Not only because this saves time, it also allows you to change the code and verify its correctness in the same way for each version of your design.

Step 7: Single stepping

- Modelsim is very powerful. You can use it to debug your Verilog design almost like software. However, do remember that we are dealing with a hardware description that operates in parallel. In contrast, software codes are generally procedural and operate sequentially.
- Try the **vsim> step** command or click on the step-command pane to watch how you can step through your Verilog code. Signal values in the object and the wave windows are updated accordingly.
- Modelsim has many useful features to help you debug your design. Details of all the commands can be found in the Modelsim Reference Manual. This is easily available under **Help > PDF Documentations > Reference Manual**. Beware that this manual is very thick! DO NOT print this out.

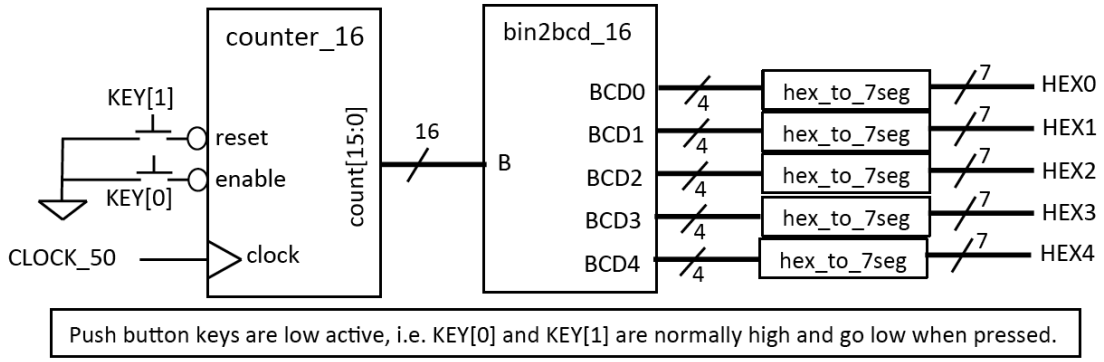
2.0 Experiment 6: Implementing a 16-bit counter on DE1

In this part of the experiment, you will test your counter design on the DE1 board. You will also learn how to find the maximum clock frequency that your design will work correctly.

Step 1: Create a new project ex6, and copy to this directly your files **counter_8.v**. Modify **counter_8.v** to **counter_16.v** and make it a 16-bit counter. Furthermore, add a reset input to reset the count value to zero synchronously to the clock. Download from the experiment webpage the component bin2bcd_16.v, a module I have designed to convert a 16-bit binary number to 5 BCD digits. You will also need the add3_ge5.v module. Put these module in the ../mylib folder, which should also contained the hex_to_7seg.v you designed in Part 1.

Step 2: Create a top-level module ex6_top.v in Verilog to specify the circuit shown below. Make sure that you have added all the relevant Verilog modules to the project using **Project**

> Add/Remove Files in Project: **counter_16.v**, **ex6_top.v** and finally add **hex_to_7seg.v**, **add3_ge5.v** and **bin2bcd_16.v** from your library folder **../mylib/**. Go to the **ex6_top.v** window and set this file as your top-level module.



Step 3: Use **Processing > Analyze Current File** check your newly create Verilog files. This is the quickest way to find the basic syntax errors in your Verilog code. Once all the simple errors are fixed, use **Processing > Start Analysis and Elaboration** to perform fuller check of the "**ex6_top.v"** to make sure that files are consistent and correct. There is no need to simulate this circuit.

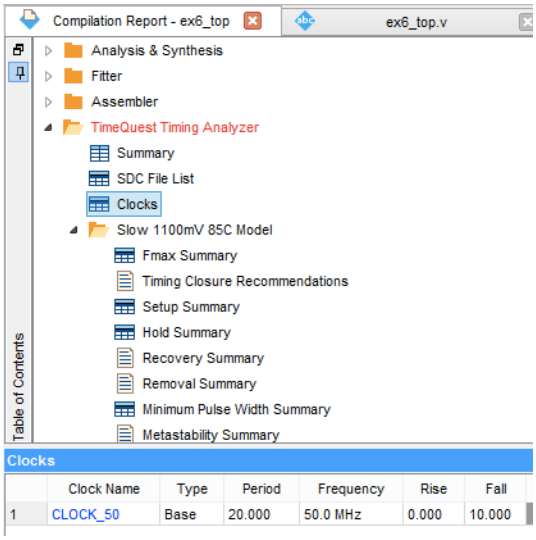
Step 4: Selecting the FPGA Device – Click **Assignments > Device**…. and select the correct Cyclone V FPGA: 5CSEMA5F31C6.

Step 5: Pin Assignment – Open the **ex6_top.qsf** file. Examine its content. You will find that no pins are being assigned yet. Insert into this file all the pin assignments. The easiest way to do this is click on: **Edit > Insert file ..** then select **../pin_assignment.txt** (you should have downloaded this file from the Experiment webpage). Note that you are currently not using all the pins assigned in the **pin_assignment.txt** file. Don't worry. This will only produce a few more warning messages. Full compilation can still go ahead without errors.

Step 6: Set clock frequency – Create a new file **"ex6_top.sdc"**[2] which should contain one single line:

```
create_clock -name "CLOCK_50" -period 20.000ns [get_ports {CLOCK_50}]
```

With this, Quartus will know that the signal CLOCK_50 is a 50 MHz clock.



Step 7: Full Compilation – Click: **Processing > Start Compilation**. This will go through the entire compilation process. Examine the Tasks window on the left and see all the steps being taken in order to generate the final bit-stream.

Step 8: Maximum clock frequency – As part of the compilation process, **TimeQuest** timing analyzer is used to predict various timing information. In the "Compilation Report" window, you should see a list of reports resulting from the compilation. Double-click **TimeQuest Timing Analyzer** entry, and you should see a list similar to the one

[2] Synopsis Delay Constraint (.sdc) files are standard formatted files introduced by Synopsis, a well-known company specializing on IC design CAD tools. With this, a designer can specify various timing constraints for the CAD tools the check against. Here we are only using this to define clock frequency.

shown here. Clicking on various entries under this will show the various timing specifications. Answer the following questions:

What are the predicted maximum frequencies for this circuit under the highest and lowest temperatures? What are the other interesting timing data that you can discover with these reports? Why is the TimeQuest entry red, indicating that there may be a problem?

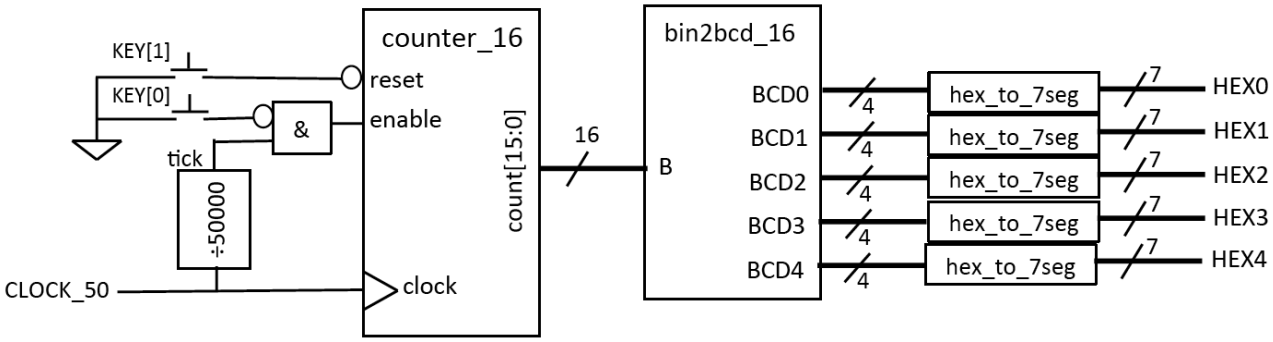
Step 9: Test your design on DE1 – program the DE1 and check that your design works.

Step 10: Examine the amount of FPGA resources being used by this 16-bit counter. Explain the results.

Test-yourself Task (compulsory) **– Cascade counter**

You are now required to create something yourself. In the previous exercise, the 16-bit counter is counting a 20MHz clock. This is much too fast for us to see the counter changing. This part of the experiment requires you use the counter to count the number of millisecond elapsed. You would need to do this by having two counters cascaded (i.e. connected in series) with each other. The overall block diagram is shown below.

The divide-by-50000 circuit generates a 1 cycle high pulse every 50,000 clock cycles. Therefore the output signal tick provides one enable pulse every millisecond. (See notes.)



Modify your circuit to implement this and test the new circuit on the DE1 board.

3.0 Experiment 7: Linear Feedback Shift Register (LFSR) and PRBS

You encountered a 4-bit LFSR in Lecture 5 slide 17, which implements the primitive polynomial: $1 + X^3 + X^4$. You are now required to implement a 7-bit LFSR implementing the polynomial: $1 + X + X^7$. Assuming that you initialize the shift register to 7'd1, work out manually the first 10 sequence values of the output sequence. (The output sequence should be 127 long without repetition, is known as a **pseudo-random binary sequence** or **PRBS**.)

Connect the shift register clock to KEY[3] and use the momentary key to cycle through the first ten values of the PRBS. The random output should be displayed as two hexadecimal digits.

Checkpoint: You should get to this point by the end of the second week.

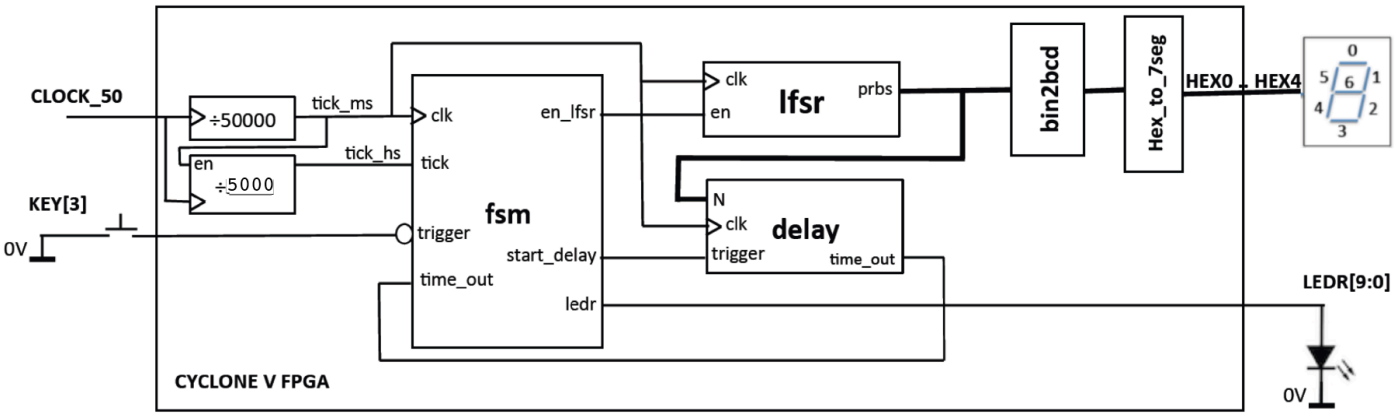
4.0 Experiment 8 (Optional challenge): Starting line delay circuit

The next two experiments are optional. They are designed to provide a challenge to those who finish early, or for those who want to learn more about digital design, Verilog and FPGAs. The two experiments are linked – what you designed in Experiment 8 will be used in Experiment 9.

The goal here is to design a Formula 1 style of race starting lights. The specification of your circuit is:

1. The circuit is triggered (or started) by pressing KEY[3] (don't forget KEY[3] is low active);
2. The 10 LEDs (below the 7-segment displays) will then start lighting up from left to right at 0.5 second interval, until all LEDs are ON;
3. The circuit then waits for a random period of time between 0.25 and 16 seconds before all LEDs turn OFF;
4. You should also display the random delay period in milliseconds on five 7-segment displays.

In order to assist you in designing this circuit without spending too much time, the following overall block diagram of the circuit is provided. You should also download the solution bit-stream for this experiment from the experiment webpage (**ex8sol.sof**) and try it out before attempt it yourself.



In the above diagram, all signals on the left of the block are inputs and the signals on the right are outputs.

The two clock divider circuits provide clock ticks once every 1ms and 0.5sec respectively. Each clock tick should be a positive pulse lasting one period of **CLOCK_50** (i.e. 20ns). The system then use the **tick_ms** signal as the clock of the remaining circuit.

The **LFSR** module produces a pseudo-random binary sequence (**PRBS**), which is used to determine the random delay required. The **enable** signal to the **LFSR** allows this to cycle through a number of clock cycles before it is stopped at a random value.

The **delay** module is triggered after all 10 LEDs are lid, and then provides a delay of **N** clock cycles (at 1ms period) before asserting the **time_out** signal (for 1ms).

The delay value N is fed to the binary to BCD converter, which then drives the 7-segment displays.

There are several design decisions to be made:

1. How many bits LFSR is required?
2. How many bits should you use in the delay module?

The FSM module is the key module to the entire system. You must decide what are the states that are required, draw the state diagram and then map that to Verilog.

5.0 Experiment 9 (Optional challenge): A Reaction Meter

Extend your circuit in Experiment 8 by adding a reaction counter. This should count the time between all the LEDs turning OFF and you pressing **KEY[0]**. The reaction time, instead of the random delay, should be displayed on the 7-segment displays in milliseconds.