ECSE 436 - Signal Processing Hardware
Verilog/Signal Tap II Introduction Handout

Prepared by TA Dylan Watts

This handout will present a brief introduction on how to create Block Symbols via Verilog to use in your .bdf design files, and how to test your design using Signal Tap II. These are two programs built into Quartus II to create and test design files. An alternative to Verilog would be VHDL, which is similar but uses slightly different syntax. Signal Tap II can also be substituted via Model Sim. If you are more comfortable with VHDL and Model Sim you may use these in your designs.

Verilog

Verilog, like VHDL, is a hardware description language which automatically translates a low-level code into gates for use in FPGA and ASIC design. For Lab 1, we will walk through the code for two different components used to create the bit stream generator for our Rate ½ Convolutional Encoder:

- **counter4**: A module which takes a clock as input and outputs a 4-bit vector Y which increments on each clock cycle from 0000 to 1111.
- **bitgenerator**: A module which will take as input a 4-bit select line SEL, and will output different bit-values depending on the specific 4-bit select line input sequence.

For these two modules we will go through the code line-by-line, highlighting some important syntax and aspects, eventually turning the files into Blocks for use in your design.

Part 1: The Code

Counter 4

module counter4 (CLK, Y);	// In order to start a new module you must first declare it's name and parameters. In this case we wish to create a counter4 module, which takes as input a clock CLK and outputs a bit-vector Y. Similar to other coding languages like Java, we must end each statement with a semi-colon.

input CLK; *output reg [3:0] Y;*	// We must now specify our inputs and outputs. CLK will act as our clock input. This is only one bit so we do not need to specify a vector length. Output Y on the hand is a vector of length 4, specified by [3:0]. Since it is just a counter we do not need to specify whether is signed, unsigned, etc. Therefore it is simply a regular output
always@(posedge CLK) begin	// For our counter to operate we need to specify an event which will trigger the increment of Y. In this case at every positive (rising) clock edge, we will begin our operation (note: there is no semi-colon. The begin-end combo acts like the brackets of method in Java).
Y <= Y + 1;	// Pretty straightforward here, at every clock pulse Y is incremented via the operation Y + 1.
End	// Required to end the body of our method.
Endmodule	// In order to indicate the completion of our module, like the end statement above, we need to add this.

Bitgen

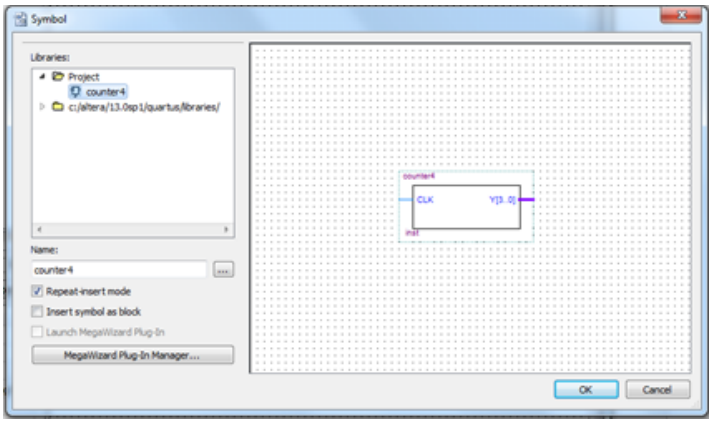
module bitgen (SEL,bitstream);	// Similar to the counter, specifying the name of the module and it's parameters
input [3:0] SEL; *output reg bitstream;*	// Specifying inputs and outputs
always @ (SEL) begin	// Our trigger event – at every new select line input evaluate the body of the method
bitstream = 0;	// Initializing the default value of the output bit to be 0
case (SEL) 4'b0001 : bitstream = 1'b1; 4'b0010 : bitstream = 1'b0; 4'b0011 : bitstream = 1'b0; 4'b0100 : bitstream = 1'b0; 4'b0101 : bitstream = 1'b1; 4'b0110 : bitstream = 1'b1; 4'b0111 : bitstream = 1'b0;	// Here we are identifying a set of special cases for the input vector SEL. If the input vector SEL is equivalent to any of the cases indicated on the left of the colon, a specific bit will be produced as output (ex: if the SEL line is 0101, the output bit will be 1).

```
  4'b1000 : bitstream = 1'b1;
 endcase

end                                    // Closing out the always method.

endmodule                              // Finishing out our module.
```

Part 2: The Block

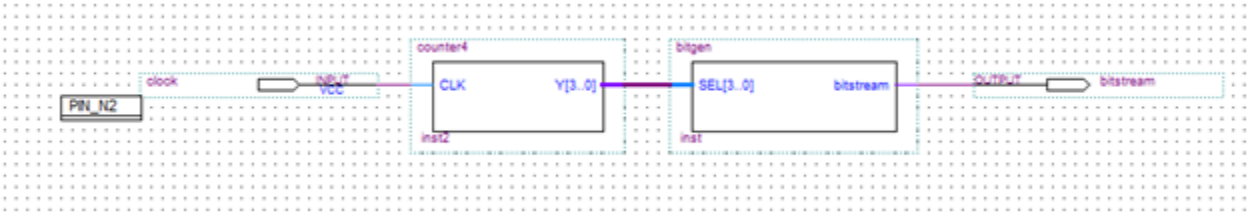
1. To start a new Verilog file, go to **File** => **New** => **Verilog HDL File** (under design files).
2. Copy and paste the code for *counter4* into the file (found at the end of this handout).
3. Save As your file as counter4
4. In your Project Navigator tab to the left, click the Files tab. counter4.v should now show up:

5. Right click counter4.v and go to “Create Symbol File for Current File”
6. Go into your .bdf file now. Try and add a symbol. Your new counter4 symbol should show up now under the “Project” folder.



7. Repeat Steps 1-6 for the bitgen Module.
8. Now to create the input generator: hmm I wonder how these two are going to fit together?
9. Set your .bdf file as the Top Level Entity and compile
10. Using the Pin Planner, set your clock pin to **Pin_N2** – this is the built in 50MHz clock on the DE-II board.
11. Set your .bdf file as the Top Level Entity and compile again.

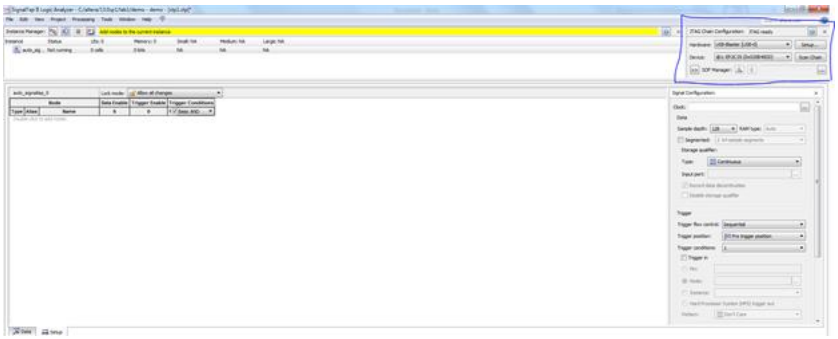
This is what your final design should look like



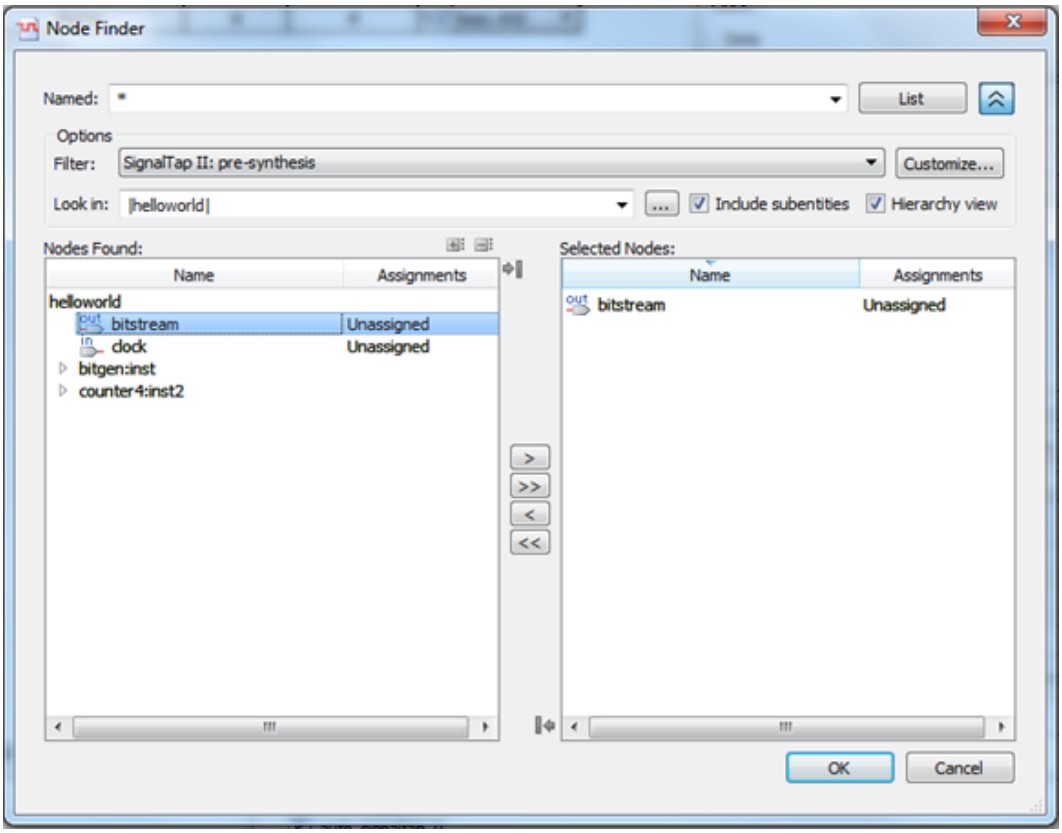
Signal Tap II

In order to test the functionality of our designs we will need to use Signal Tap II or Model Sim to generate waveform files of the bits at the inputs and outputs. I like Signal Tap II because it runs directly off the actual hardware, so you can ensure that it works on the physical device but again, if you can show me it works on Model Sim that works for me. To test our bitstream generator we need to set up a new vector waveform file and run it on the board, which we will now do step by step below:

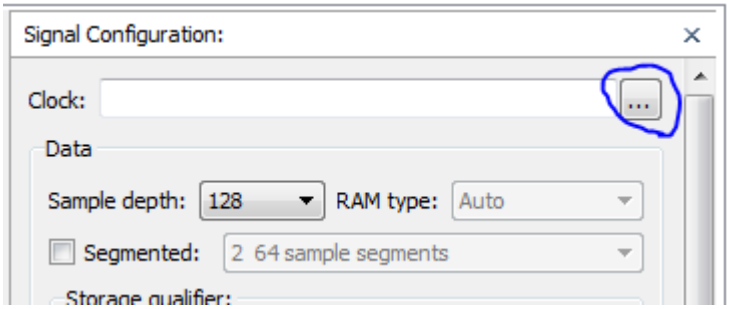
1. Plug your board in and turn it on.
2. Go to **File** => **New** => **Signal Tap II Logic Analyzer File** (under "Verification/Debugging Files).
3. A new window should show up. Make sure that The JTAG device is recognized (top right-hand corner, highlighted in blue). If it isn't come see me.



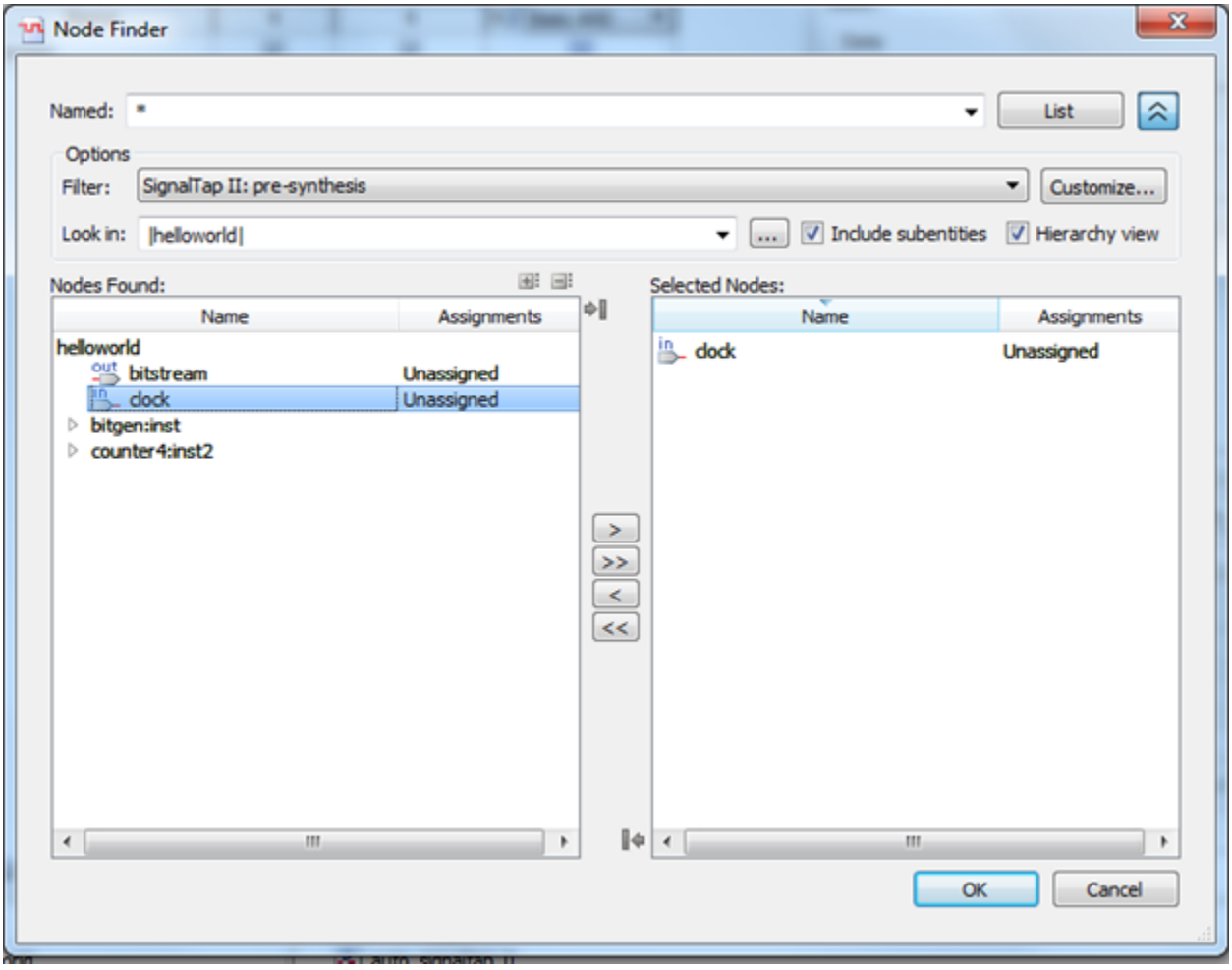
4. Double Click to add nodes. Make sure your **Filter** is set to **SignalTap II Pre-synthesis** NOT postfitting. Click **List** and your input/output pins should show up.
    - NOTE: Make sure you "Look in" your .bdf file, in this case |helloworld|. If your pins do not show up properly then you did not set your .bdf file as the top-level entity when compiling – this is a super common mistake. If you do not see the window below, go back to your design file, set the .bdf file as the top level entity, compile, and double click to add nodes again. You should get the screen below.
5. Click on the bitstream node and the **>** button in the middle, then OK. This will be the node we want to measure.



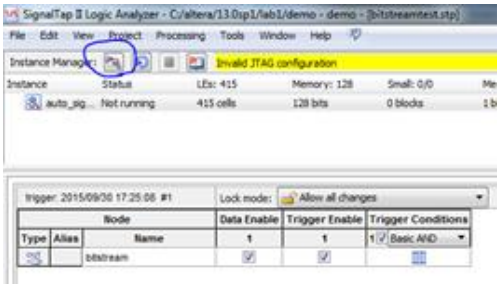
6. Now we need to set our signal configuration. In this case all we need to do is set the Clock. Click the button highlighted in blue



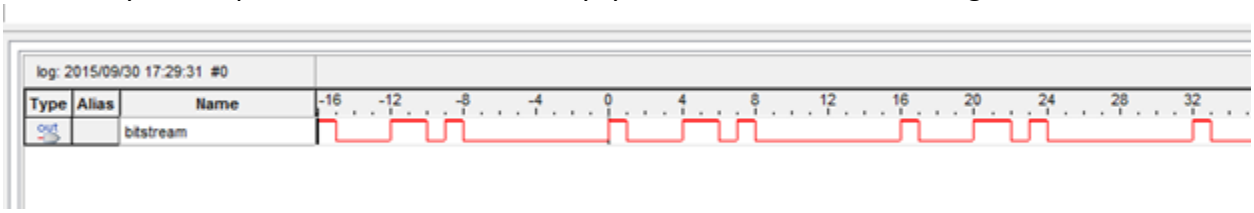
7. A similar window should show up as choosing the nodes for monitoring. In this case, select the **clock** node and press the **>** button, then OK



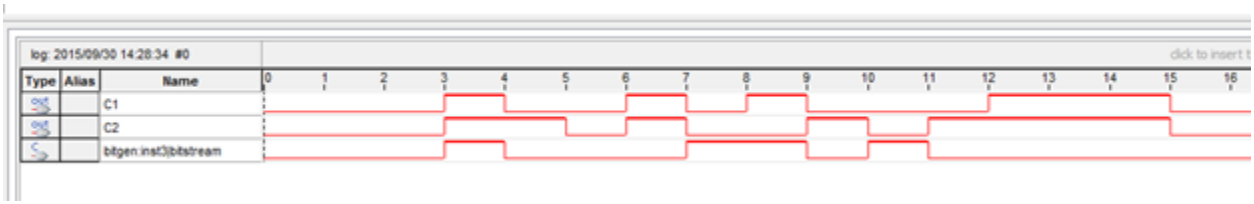
8. Now Save As bitstreamtest, and enable it for the current project.
9. Go back to your design file, and under your Project Navigator Files tab you should now see bitstreamtest.stp.
10. Set your .bdf file to the Top-Level Entity, save all files (by pushing the triple floppydisk button right next to the save button), compile the project, and program the device
11. Go back to your SignalTap II window and hit the “Aquire Data” Button



12. If you completed the tutorial correctly, you should see the following waveform:



You can now hook up your bit stream generator to your Rate ½ Convolutional encoder and finish the Lab 1! Create Pins C1 and C2 at your outputs to monitor the outputs of the encoder in addition to the bitstream, and create a new waveform test to validate your design. You should get the following waveform:



This completes the brief introduction to Quartus II and some of it's features. Make sure you are familiar with how to do all of the steps in Handouts 1 and 2, as you will be using them extensively throughout the remainder of the course!

Appendix – Code for Lab 1 Modules

Counter4

```
module counter4 ( CLK, Y );

input   CLK;
output reg [3:0] Y;

always@(posedge CLK) begin

Y <= Y + 1;

end

endmodule
```

bitgen

```
module bitgen (SEL,bitstream);

input [3:0] SEL;
output reg bitstream;

always @ (SEL)
begin
  bitstream = 0;
  case (SEL)
    4'b0001 : bitstream = 1'b1;
    4'b0010 : bitstream = 1'b0;
    4'b0011 : bitstream = 1'b0;
    4'b0100 : bitstream = 1'b0;
    4'b0101 : bitstream = 1'b1;
    4'b0110 : bitstream = 1'b1;
    4'b0111 : bitstream = 1'b0;
4'b1000 : bitstream = 1'b1;
  endcase
end

endmodule
```