

Exercise Manual

for

Advanced Verilog Design Techniques

Requirements to Complete All Exercises

Quartus® II software version 10.1 or later, ModelSim®-Altera® version 6.6c or later

Use the link below to download the design files for the exercises:

https://www.intel.com/content/www/us/en/programmable/customertraining/ILT/Advanced_Verilog_Design_Techniques_21_3_v1.zip

Exercise 1

Use if-else statements
efficiently

Objective:

- Adjust state machine output logic to use `if-else` statements efficiently

Note: Be sure to completely read the instructions for each step and sub-step in this lab manual. Each step first summarizes what you will be doing in that step before providing complete instructions. Use the lines next to each step (____) to keep track of your progress or to check off completed steps in the exercises.

If you have any questions or problems, please ask the instructor for assistance.

Conditional statements, such as `if-else`, are behavioral Verilog statements that are executed sequentially inside procedural (i.e. `initial`, `always`) blocks. When using conditional `if` statements, care must be taken when coding for two reasons:

1. Mutual exclusivity: unnecessary chaining and nesting of `if-else` statements can lead to more complex logic equations. This, in turn, will increase the use of logic resources and will make routing more difficult.
2. Open/closed `if-else`: Open `if-else` statements are `if-else` statements for which output values have not been defined for all possible input conditions. Open `if-else` statements in combinatorial processes generate latches which lead to more complex logic equations and thus increased logic and routing usage. By closing `if-else` statements, or defining output values for all possible input conditions, your logic becomes smaller and more predictable with regards to timing.

This exercise examines both of these issues and how they affect logic synthesis.

Table 1.


Counter implementation	Number of LEs
Original <code>if-else</code> structure	
Modified <code>if-else</code> structure	

Step 1: Open if_else project and examine code

- ____ 1. Unzip the lab project files, if necessary. In an Explorer window, go to **C:/altera_trn/Adv_Verilog**. Delete any old lab file folders that may already exist there. Double-click the executable file (**Advanced_Verilog_Design_Techniques_12_1_v1.exe**) found in that location. If you cannot find this file, ask your instructor for assistance. In the WinZip dialog box, just click **Unzip** to automatically extract the files to the directory mentioned above, creating a folder named **AVER12_1**. This will be your lab installation directory.
- ____ 2. Start the Quartus II software (version 10.1 or later; use the newest version available), from the **Start** menu (**All Programs** → **Altera <version_number>** → **Quartus II <version_number>**; use the 64-bit version if using an Altera training laptop or desktop) or from a shortcut on the desktop.
- ____ 3. From the **File** menu, select **Open Project** (*be sure to not select **Open** which opens an individual file, not a project*). **Open** the project **if_else.qpf** located in the **<lab_install_directory>/Ex1** directory.
- ____ 4. Open the Verilog file for the project. From the **File** menu, select **Open** and select **if_else.v**, or just double-click **if_else** in the **Project Navigator** on the left.

The code shows the output logic of a state machine using nested and chained if statements. We'll talk much more about the creation of state machine logic later.

Step 2: Synthesize design and examine results

- ____ 1. Synthesize the design. Click  in the toolbar or, from the **Processing** menu, go to **Start** and select **Start Analysis & Synthesis**.
- ____ 2. Record the results. After the compilation has finished examine the **Flow Summary** section of the **Compilation Report**. How many logic elements (LEs) are used? Record the result in **Table 1**.
- ____ 3. Open the **RTL Viewer**. From the **Tools** menu, go to **Netlist Viewers** and select **RTL Viewer**.

*Do you notice the latches on the outputs (**out1**, **out2**, **out3**)? Unintentional latches like these can complicate timing analysis and most often do not accurately reflect what the designer had in mind.*
- ____ 4. Close the **RTL Viewer**.
- ____ 5. Open the **Technology Map Viewer**. From the **Tools** menu, go to **Netlist Viewers** and select **Technology Map Viewer (Post-Mapping)**.

*This view displays how the Quartus II software actually implemented the `if-then` equation in logic. Notice the output of the 3 LCELLs (**LOGIC_CELL_COMB**) feeding the output pins (**IO_OBUF**) are also feeding back into the inputs of the 3 LCELLs. Again, this is unnecessary routing and design complexity.*

- ____ 6. Close the **Technology Map Viewer**.


Step 3: Modify `if_else.v` using multiple `if-else` statements

- ____ 1. Bring the `if_else.v` file to the foreground, if necessary.
- ____ 2. Change the module name to `if_else_new`.
- ____ 3. Modify the code so that it uses multiple `if` statements instead of prioritized `if-else` statements. Also, ensure all `if-else` statements are closed using one of the techniques discussed.
- ____ 4. Save the file as `if_else_new.v`.

Step 4: Synthesize design and examine results

- ____ 1. Make `if_else_new.v` file the top-level module in the project. With `if_else_new.v` in the foreground, from the **Project** menu, select **Set as Top-Level Entity** (near the bottom of the menu).

This causes the compiler focus to be `if_else_new` instead of `if_else`.

- ____ 2. Synthesize the design. Click  in the toolbar or, from the **Processing** menu, go to **Start** and select **Start Analysis & Synthesis**.
- ____ 3. Record the results. After the compilation has finished, again examine the **Flow Summary** section of the **Compilation Report**. Record the new number of LEs used in **Table 1**.

Compare the number of logic elements used. It should be lower, indicating that exclusive and closed `if` statements can produce more efficient logic.

- ____ 4. Verify the improved logic structure in both the **RTL Viewer** and **Technology Map Viewer**.

*In the **RTL Viewer**, you should see no latches, and in the **Technology Map Viewer**, you should see no combinatorial feedback loops.*

Exercise Summary

- Using multiple if-else statements with mutually exclusive conditions can lead to reduction in logic and logic complexity
- Closing `if-else` statements eliminates latches

END OF EXERCISE 1

Exercise 2

Create a modulus up/down
counter

Objective:

- Create an 8-bit up/down counter with a modulus

In this exercise, you will create an 8-bit up/down counter with a modulus using variables.

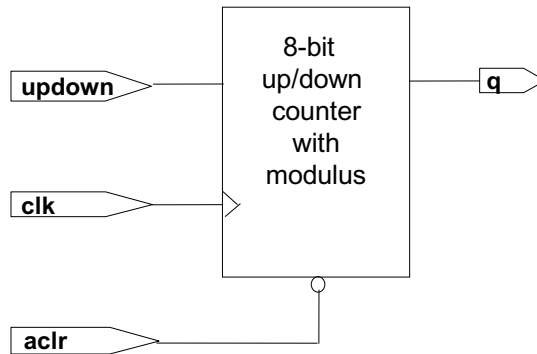


Table 2.

Counter implementation	Number of ALUTs
Original 16-bit counter	
Modulus-100 counter	

Step 1: Open the project and examine the code

- ____ 1. Open the **counters** project. From the **File** menu, select **Open Project**. Browse to the directory <lab_install_dir>\Ex2 and select **counters.qpf**.
- ____ 2. Open the design file **count.v**, either from the **File** menu or the Project Navigator. Quickly investigate this file.

You should see the code for a simple binary up counter.

Step 2: Synthesize the design and examine results

- ____ 3. As in Exercise 1, synthesize the design.
- ____ 4. Record the results. After the compilation has finished, examine the Flow Summary section of the Compilation Report. How many combinational ALUTs (adaptive lookup tables, the main logic blocks found in Stratix® devices) are used for this counter? Record the result in **Table 2**.

Step 3: Modify the design to implement an 8-bit up/down counter with modulus

- ____ 1. Bring **count.v** to the foreground.
 - ____ 2. Change the module name to **count_moda**.
- ____ 3. Modify the design so that the counter has a modulus with a terminating count of 99. You will need to add an additional input control signal named **updown**.

Hint: Think of the behavior of an up/down counter. When counting up, once the counter reaches 99, it must reset itself and continue counting up like normal. But when counting down, once the counter reaches 0, it should reset itself to 99 and start counting down like normal.

- ____ 4. Save the file as **count_moda.v**.
- ____ 5. Set **count_moda** as the top-level entity for the project. From the **Project** menu, select **Set as Top-Level Entity**.




Step 4: Synthesize design and examine results

- ____ 1. Synthesize the design.
- ____ 2. Again record the results. How many ALUTs are now used for the counter? Record your answer in **Table 2**.

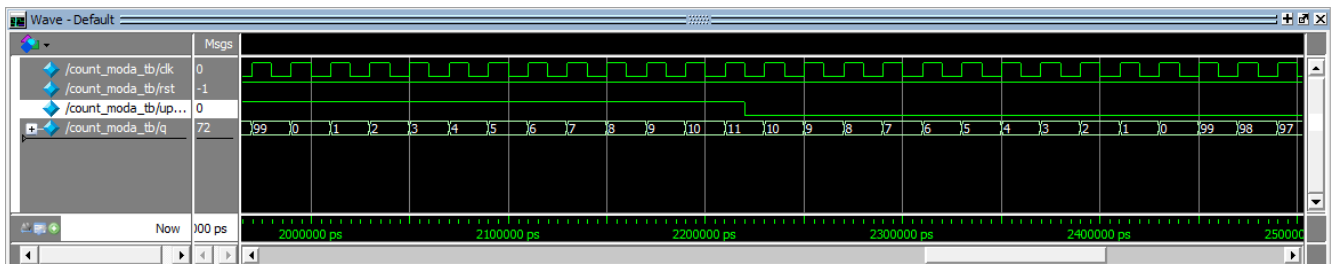
Step 5: Perform a behavioral simulation of the counter

1. Open the ModelSim-Altera simulation tool. From the **Start** menu, again go to **All Programs** → **Altera <version_number>**. Go to the **ModelSim-Altera <version_number>** folder and open **ModelSim-Altera (for the Quartus II software version 12.1, the ModelSim-Altera version number is 10.1b; be sure to use the version of ModelSim-Altera that matches the version of the Quartus II software you are using)**. If the **Welcome** window appears, click **Close**.
2. Set the project directory. From the ModelSim-Altera **File** menu, select **Change Directory**. Browse to the <lab_install_dir>\Ex2 directory. Click **OK**.
3. Run a script file. From the **Tools** menu, go to **Tcl** and select **Execute Macro**. Choose the **setup.do** file and click **Open**.

*This .do (ModelSim scripting format) file will compile all of the design files, including a pre-created testbench (named **count_moda_tb.v**), load the simulation, open up the Wave view and add signals to it for monitoring. It will then run the simulation for 3 us.*

4. Is your counter counting up AND down properly? If you don't see anything, zoom full  first. Then, zoom in  or out  as needed to view the waveform.

Your waveform should look like the screenshot below. The counter should roll over from 99 to 0 at 2 us and roll from 0 to 99 at around 2.4 us.



5. From the **Simulate** menu, select **End Simulation**. Click **Yes**.
6. If you need to go back and make corrections to your code in the Quartus II software, please do so, resynthesize, and then re-run **setup.do** in the ModelSim-Altera simulator until the simulation behavior is correct.
7. Close the ModelSim-Altera simulator when finished.

Exercise Summary

- Built a modulus 100 up/down counter based on the examples given in the presentation

END OF EXERCISE 2

Exercise 3

Practice encoding a simple
state machine

Objectives:

- Practice encoding a state machine
- Practice custom encoding a state machine

Table 3.

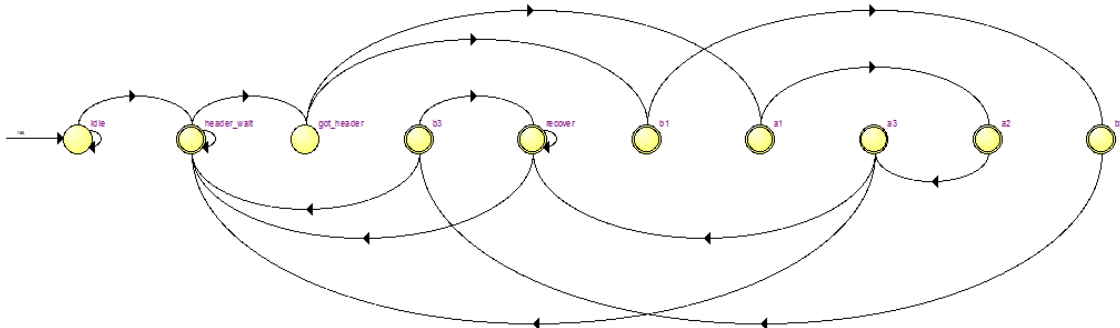
State name	Default encoding	Minimal bits
idle		
header_wait		
got_header		
a1		
a2		
a3		
b1		
b2		
b3		
recover		

Table 4.

<i>State</i>	<i>data_a</i>	<i>data_b</i>	<i>validate</i>	<i>error</i>	<i>Encoding</i>
<i>idle</i>	0	0	0	0	
<i>header_wait</i>	0	0	1	0	
<i>got_header</i>	0	0	0	0	
<i>a1</i>	1	0	0	0	
<i>a2</i>	1	0	0	0	
<i>a3</i>	1	0	1	0	
<i>b1</i>	0	1	0	0	
<i>b2</i>	0	1	0	0	
<i>b3</i>	0	1	1	0	
<i>recover</i>	0	0	0	1	

Step 1: Synthesize a state machine

- ____ 1. From the **File** menu of the Quartus II software, open the project `<lab_install_dir>\Ex3\state_machine.qpf`.
- ____ 2. Synthesize the design as before.
- ____ 3. In the Compilation Report window Flow Summary section, how many combinational ALUTs and dedicated logic registers were used? _____
- ____ 4. From the **Tools** menu, go to **Netlist Viewers**, and select **State Machine Viewer**. Review the transition diagram and table.



- ____ 5. Click the **Encoding** tab at the bottom of the **State Machine Viewer**. Record the encoded values for each of the states in **Table 3** above.

The name of each of the 10 states of the state machine are listed in the **Name** column on the left. The other 10 columns represent the 10 registers used to encode the 10 states. Each register is named after a particular state because in each state, only one register is considered active. For all but one state, the active value for that register is logic 1. For the idle register/state, the active value is logic 0.

Based on this information, what kind of encoding is this? It is actually one-hot encoding. In Altera devices, all internal registers power up low by default. With true one-hot encoding, your idle state would be encoded as 0000000001. This means that you would power up into an undefined state in an Altera device or additional logic would be needed to preset the idle register. To get around this, the Quartus II synthesis tool inverts the first bit of the state machine so that you power up into a known state, 0000000000.

Step 2: Select Minimal Bit Encoding

1. From the **Assignments** menu, select **Assignment Editor**.


This opens up a tool which allows you to add logic options/assignments to your design using a spreadsheet format.

2. In the Assignment Editor, double-click the <<new>> cell in the **To** column of row 1. Type the name **current_state** and hit the **Enter** key.

*Notice that the name **current_state** now appears with a little 'S' to the left of it. This indicates that this is the name of a state machine, or the signal that gets assigned the state names in your code.*

3. In the **Assignment Editor**, double-click the cell in the **Assignment Name** column in the first row. Select **State Machine Processing** from the drop-down menu.

4. Now, double-click on the cell in the **Value** column in the same row. Select **Minimal Bits** in the drop-down menu.

	tatu	From	To	Assignment Name	Value	Enabled	Entity
1	✓		 current_state	State Machine Processing	Minimal Bits	Yes	state_machine
2		<<new>>	<<new>>	<<new>>			

5. Synthesize the design as before. Click **Yes** when asked to save the assignments.

6. In the **Compilation Report** window **Flow Summary** section, how many combinational functions and dedicated logic registers were used this time?

Note that fewer registers are used. This is typically the difference between implementing one-hot encoding versus binary encoding in an FPGA. Because look-up table devices like FPGAs have a limited number of combinational inputs per cell, one-hot encoding uses more registers. However, one-hot encoding sometimes uses less combinational logic to feed those registers since there is only one active bit per state. Meanwhile, binary encoding uses fewer registers but sometimes requires more combinational logic in front of those registers due to more active bits in each state.

7. Use the State Machine Viewer again (may need to close and reopen the viewer) or the **State Machines** section of the **Analysis & Synthesis** folder in the **Compilation Report** to record the encoding scheme of **current_state** into **Table 3** in the **Minimal Bits** column.

Notice that the state diagram is the same as before, but the encoding has changed. You should have 4 state bits being used to encode the 10 states. Is this binary encoding? Sort of. The Quartus II software recognizes that the largest benefit of binary encoding is the reduction in the number of state registers. Thus, using this "minimal bit" encoding style, the number of state registers is minimized, but they may not be in numeric order.

Step 3: Custom encode the state machine

- _____ 1. Open the **state_machine.v** file again. Change the module name to **state_machine_new**. Save the file as **state_machine_new.v**.
- _____ 2. Set **state_machine_new** as the new top-level entity for the project.
- _____ 3. Create a custom encoding style based on **Table 4** above. To do this, base the encoding on the values of the outputs as shown in the presentation.

Remember, in this custom encoded style, the idea is to limit the output decoding logic by basing your state values on the outputs.

Hint: You cannot have two states that use the same encoded value, even if the outputs in the two states are the same. If this happens, you must add an additional state bit to differentiate between the identical states.

- _____ 4. Edit **state_machine_new.v** to encode the state machine based on your values above. Use the parameters to define your custom encoding.
- _____ 5. Return to the **Assignment Editor** and set the **State Machine Processing** logic option to **User-Encoded** for the **current_state** variable. Save the **Assignment Editor** options.
- _____ 6. Synthesize the design.
- _____ 7. In the **Flow Summary** section of the **Compilation Report**, how many combinational ALUTs and dedicated logic registers are used now? _____

*You should see from the State Machine Viewer (close and reopen if necessary) that while the state diagram again has not changed, the encoding has. (If not, make sure you set your **State Machine Processing** logic option correctly.) You should have 5 state bits (5 **current_state** columns) and the encoding should match what you wrote in Table 4. To verify the hardware implementation, check the Technology Map Viewer. Locate each of the outputs (i.e. **error**, **data_a**, **validate**, **data_b**) and trace each output back to its source, through the I/O buffer. You should find that each output is fed by a single register with no combinational logic in between.*

Exercise Summary

- Encoded a state machine automatically and manually using synthesis options
- Reduced logic utilization by custom encoding a state machine

END OF EXERCISE 3

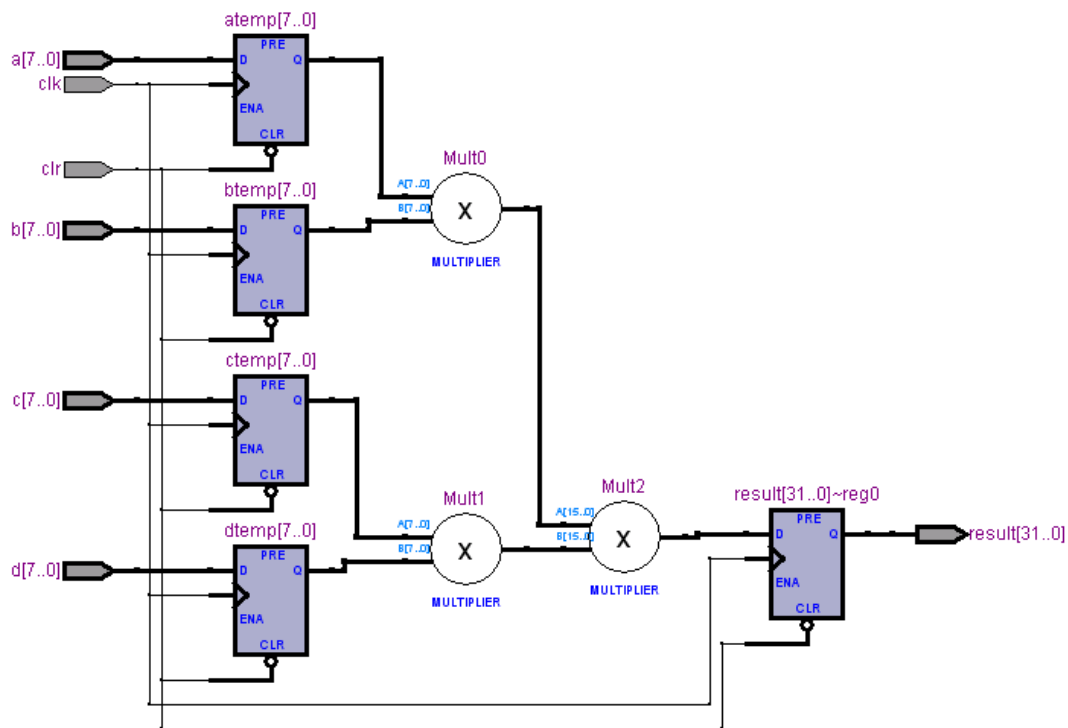
Exercise 4

Create Class I and II
testbenches

Part 1

Objective

- To create a Class I testbench to simulate a multiplier design



A 4-input multiplier design

Step 1: Open ModelSim-Altera project

- _____ 1. If it's not already open, open the ModelSim-Altera simulation tool.
- _____ 2. Open the ModelSim **three_mult_test** project. From the **File** menu, select **Open**. Change the file type to **Project Files (*.mpf)**. Browse to the **<lab_install_dir>\Ex4** directory and open **three_mult_test.mpf**.
- _____ 3. Examine the **three_multb** design file. From the **File** menu, again select **Open**. If necessary, change the file type to **HDL Files** or **Verilog Files**. Open the **three_multb.v** file.

*This is a 4-input multiplier that requires 3 multiplier blocks to implement. Recall that for every operator in the code (in this case *), a logic block will be synthesized to implement that operator. Notice it takes 2 cycles for the multiplier to produce a result due to the non-blocking assignments.*

- _____ 4. Examine the **three_mult_tb** testbench file. Open the design file **three_mult_tb.v** in the text editor.

This is the skeleton for the testbench that you will be creating. All the sections you will be coding are commented and (mostly) in the order in which you will be filling them in, so you know where things go. After reading a step below, check the comments to see in which section it belongs. This will help you stay organized.

Note: You will be filling in the testbench block by block (mostly initial blocks). You CAN of course combine blocks together to make things more efficient (as long as the relative timing stays the same), but for instruction purposes, you will code each block separately.

Step 2: Set default timing

- _____ 1. Define the timescale. For this testbench, set a timescale of 1 ns and a precision of 1 ns.

That should be plenty of for this exercise. Remember, unnecessarily using smaller timescales means an increase in memory usage and simulation time.

- _____ 2. Define a 20 ns clock period. Use the **`define** compiler directive to create a **CLOCK_PERIOD** macro of **20**.

Notice the empty module declaration has been done for you. (You get to do all of the hard stuff!)

Step 3: Create testbench signals

For the signal names, use the names suggested below, as the script files you will run later assume these names. If you would prefer to name them differently, just be aware you will have to manually change the script.

- _____ 1. Create the data inputs. Create four 8-bit variable vectors named **a**, **b**, **c**, and **d** (like in the multiplier block).
- _____ 2. Create the clock and reset. Create a single bit variable **clk** signal and a single bit variable **clr_n** signal.
- _____ 3. Create a 32-bit net signal named **result** to connect to the multiplier output.

*This signal must be a net data type since it will be driven by an output of the **three_multb** module instantiation. Sub-module inputs can be driven by net or variable data types, but sub-module outputs and bidirectionals must drive to net data types.*

Step 4: Define clock and reset patterns

- _____ 1. Define a **20-ns clock pattern** for **clk** that will be used to drive everything in the design. Use either method shown in the presentation (i.e. `initial` block/`forever` loop or `initial` and `always` blocks).
- _____ 2. Define a pattern for **clr_n** that has it **power up asserted** and then gets **released after 40 ns**.

Step 5: Define patterns for data inputs

- _____ 1. Initialize all four data inputs to 1.
Since this is a multiplier, the inactive states for an input would be a 1.
- _____ 2. Define a counting pattern to be used for the **a** input. For input **a**, create a count pattern with the following criteria:
 - a. The pattern should wait until the reset is de-asserted before counting begins. You can use simple delay (brute force) or event control sensitive to a rising edge reset (adjusts with the reset delay) to do this.
 - b. Once counting begins, the **a** input should count on every 3rd falling edge clock cycle with no end (*Hint: the first count value occurs on the 3rd falling edge and repeats from there forever*).

- _____ 3. Define a non-regular pattern for the **b** input using the following table.

At simulation time	Decimal value of b should be
100	2
160	3
200	4
500	5
800	7

Step 6: Instantiate the DUT

- _____ 1. Instantiate the **three_multb** block into the design connecting the module instantiation ports to their corresponding testbench wires or reg variables.

Your Class I testbench for Part I of this exercise is now complete. Now, you will run it to see if you set it up correctly!

Step 7: Compile the design and fix any errors

- _____ 1. Before you compile your code, you must create a working library in the ModelSim tool into which the design will be compiled. From the **File** menu, go to **New** and select **Library**. If not set already, set both the **Library Name** and the **Library Physical Name** to **work**. Make sure **a new library and a logical mapping to it** is selected. Click **OK**.


When you compile, the ModelSim-Altera tool creates a functionally-equivalent, executable version of your code for simulation. All ModelSim projects must have a work library.

- _____ 2. Compile all Verilog files. From the **Compile** menu, select **Compile All**.

*Check the **Transcript** window to see if you got any errors or warnings. If you get an error, double-click the error message to open a message window that will give you more details on what exactly is wrong. Of course, ask the instructor if you are having trouble figuring out what is wrong.*

*Each time you edit/fix your code, save the files and run **Compile All** again until you get no errors.*

Step 8: Simulate the design and verify visually

To help with setting up the simulation, a simulation configuration named **three_mult_sim** has already been created for you. You can find it in the ModelSim-Altera **Project** window under the Verilog source files (it has a little ModelSim icon ). This simulation configuration defines simulation-related items such as the top-level module for simulation, any additional resource libraries being referenced and the default simulation resolution. By using a simulation configuration, you don't have to specify this information each time you start a simulation.

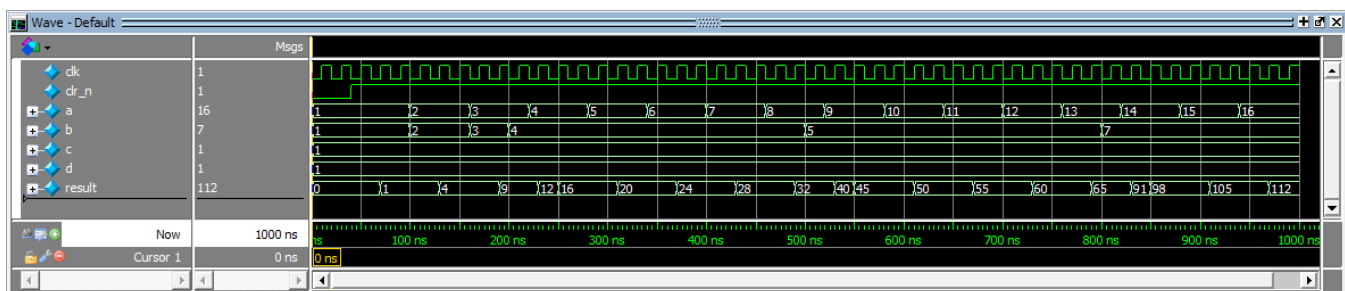
1. Start the simulation. In the **Project** window (if you don't see this window, enable it in the **View** menu), right-click the **three_mult_sim** configuration and select **Execute**.

This loads your design into the simulator and sets the simulation time to time 0. Again, if you get any errors, read the message to determine the problem. After fixing, execute the simulation configuration again until it loads with no errors (timescale warnings are fine).

To accelerate verification, a ModelSim macro file (**sim.do**) file has been created for you. As in exercise 2, it opens the Wave window, adds the testbench signals to it (formatting them in the process), and advances the simulation. You can open this file if you are curious to see how it does this.

The **sim.do** file uses the `run -all` command to run the simulation until there is no more stimulus. Normally, you'd have to stop the simulation manually, but the testbench uses a `$stop` command at the bottom to halt simulation at 1 us. Of course, this is not the end of the stimulus, but it should be long enough to see if your code is working.

2. Run **sim.do**. From the **Tools** menu, go to **Tcl** and select **Execute Macro**. Choose **sim.do** and click **Open**.



The Wave window (shown above) should be open so you can view the results of the simulation. If you can't see it all, use the zoom control options in the toolbar or right-click anywhere in the waveform.

- ____ 3. Verify the correct behavior of your stimulus. Visually check against the results above.

Did your simulation function correctly? Check to see that it is multiplying the four inputs correctly. Remember there is a full cycle between when the inputs are sampled (on a rising edge) and when the result appears. If the simulation results are incorrect, check your stimulus.

If you have to change the testbench file, do the following:

- *Change the file and save.*
- *In the Transcript window, type `abort`. This kills the macro file that was running.*
- *From the **Simulate** menu, select **End Simulation**.*
- ***Compile All** again.*
- *Execute the simulation configuration and the **sim.do** file again.*

Part 2**Objective:**

- *Enhance the Class I testbench from Part 1 to create a complete Class II testbench*

Step 1: Prepare to update the testbench

- _____ 1. End your previous simulation if it is still running. From the **Simulate** menu, select **End Simulation**. Click **Yes**.

*To make it easy to test your stimulus and check against expected results read from text files, you must first disable the stimulus you defined for inputs **a** and **b** in Part 1.*

- _____ 2. Disable the stimulus for the **a** and **b** inputs. Open the **three_mult_tb.v** file, if it is not already open. Comment out the stimulus for **a** and **b**, leaving the input initialization assignments un-commented. An easy way to comment code is to highlight the uncommented section of code in the ModelSim text editor, right-click the selected code, go to **More** and select **Comment Selected**.

Step 2: Use the provided data file as input stimulus

A data file has already been created and provided with this exercise for you to use as input stimulus. You will use system tasks to read these values into internal memories so can apply them to the multiplier.

- _____ 1. Review the input stimulus. Open the data file **input.dat** in the ModelSim (or any) text editor. You'll need to change the file type to **All Files** to see it.

*For simplicity, the stimulus file has a pair of values per line, with a total of 16 pairs. You will apply each pair of values to the **c** and **d** multiplier block inputs.*

- _____ 2. Create a memory array to hold the stimulus. At the top of the testbench file, locate the commented section that says to **Declare testbench internal signals**. Create a **memory array** that is **8 bits wide** and **32 words deep**.
- _____ 3. Write a procedural block to read in stimulus and apply it to inputs **c** and **d**. Locate the code in the corresponding commented section. Your code should do the following:
- Read the file **input.dat** into the memory.
 - Wait until the **clr_n** signal is de-asserted.
 - Read the first two memory addresses into **c** and **d** on the falling edge of the clock.
 - Create a loop that will read the remaining 15 pairs of values and assign them to **c** and **d** on falling clock edges. You should **delay 5 clock cycles** between each read. Don't forget to add a loop index integer variable as well.

- _____ 4. Extend the length of the simulation. At the bottom of the testbench file, increase the time for the simulation to stop from 1 μ s (1000) to 2 μ s (2000).
- _____ 5. Instead of working on the expected outputs right away, try compiling and running a simulation to verify that your stimulus is working correctly. Use the same steps from Part 1 to do this (compile; execute the configuration; run the script file).

*Remember that now, **a** and **b** should not be changing, but **c** and **d** should get loaded with all 16 values stored in **input.dat**. Once you have verified that you are reading data successfully (remember to end the simulation when complete), you are ready to move on to include a check against the expected results.*

Step 3: Use the provided data file as expected results

A data file with expected outputs has already been created and provided with this exercise for you. Like with the input stimulus, you will use system tasks to read these values into internal memories so can compare them to the multiplier output.

- _____ 1. Review the expected results data. Open the data file **expected.dat** in the ModelSim (or any) text editor.

*This file contains a single column of expected values, essentially the multiplied results of **input.dat** (since **a** and **b** are stuck at 1).*

- _____ 2. Create a memory array to hold the stimulus. At the top of the testbench file, create a memory array named **exp_res** that is **8 bits wide** and **16 words deep**.

- _____ 3. Write a procedural block to read in stimulus and compare it to the multiplier result. Locate the code in the corresponding commented section. Your code should do the following:

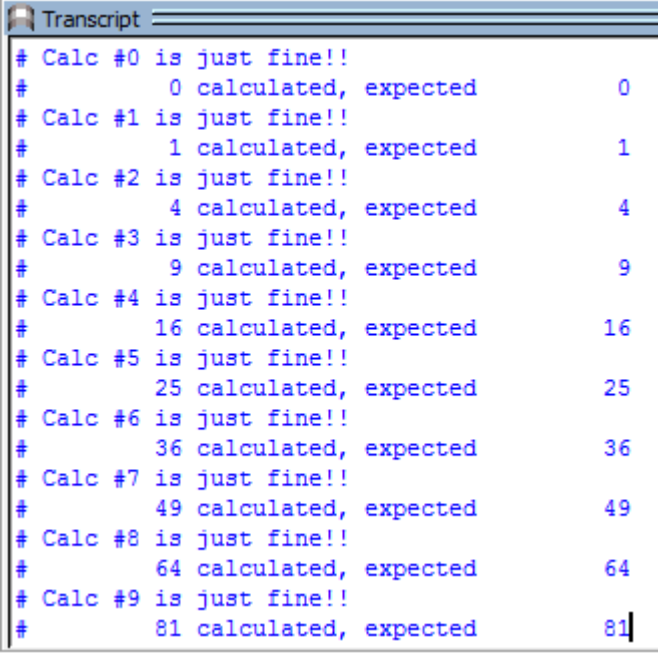
- a. Read the file **expected.dat** into the **exp_res** memory.
- b. Wait until the **clr_n** signal is de-asserted.
- c. Since there are two register stages in the multiplier block, you must wait for the same falling edge clock cycle as the input and then 2 additional clock cycles before you can read the first output value. A compare task has been provided to compare the values you read. The task definition is found near the bottom of the testbench file. You should pass it the address of the expected value being checked along with the result and the expected value, in that order. For example:

```
compare(0, result, exp[0]);
```

The task prints out messages to indicate whether the check was successful or not.

- d. Create a loop that will read the remaining expected values and use the compare task to check them against the result. You will need another index integer variable. Be sure to delay 5 clock cycles between each check and compare (like the input).

4. Again, compile all files, execute the simulation configuration, and run the simulation. Check the simulator **Transcript** window to see if the compare routines caught any errors. If everything is correct, you should see results like these:



```
Transcript
# Calc #0 is just fine!!
#      0 calculated, expected      0
# Calc #1 is just fine!!
#      1 calculated, expected      1
# Calc #2 is just fine!!
#      4 calculated, expected      4
# Calc #3 is just fine!!
#      9 calculated, expected      9
# Calc #4 is just fine!!
#     16 calculated, expected     16
# Calc #5 is just fine!!
#     25 calculated, expected     25
# Calc #6 is just fine!!
#     36 calculated, expected     36
# Calc #7 is just fine!!
#     49 calculated, expected     49
# Calc #8 is just fine!!
#     64 calculated, expected     64
# Calc #9 is just fine!!
#     81 calculated, expected     81
```

Step 4 (Optional)

1. If you have extra time, try changing the input stimulus file to include data for 3 or all 4 inputs. Of course, this will require changing the routine that reads the file, increasing the memory array size, and changing the expected results routine.

Exercise Summary

- Looked at several ways to implement Class I and Class II testbenches
- Generated stimulus using behavioral statements
- Created code to read stimulus and expected results from external files

END OF EXERCISE 4

Exercise 5

Writing parameterized code

Objectives:

- *Create parameterized modules from non-parameterized code*
- *Use generate regions to increase code flexibility*

Step 1: Open project

- _____ 1. Open the project **param_gen**. Reopen the Quartus II software, if necessary. From the **File** menu, open the project <lab_install_dir>\Ex5\param_gen.qpf.

This project contains multiple revisions. We'll edit files in the different revisions to practice using parameters and generate statements as well as to check individually the different parts that will make up the top-level design.

Step 2: Parameterize the non-pipelined multiplier block

- _____ 1. Select the **three_mult** revision. From the top of the Quartus II window, use the drop-down list to select the **three_mult** project revision.

*The Project Navigator hierarchy display should show **three_mult** as the top-level entity in the revision.*

- _____ 2. Open **three_mult.v**. Double-click **three_mult** in the Project Navigator to open the file use the **File** menu.

*This code implements a simple 2-cycle, 4-input multiplier block. The output **result** does not get immediately updated because the inputs require a clock cycle to be stored in temporary variables.*

- _____ 3. Use Verilog parameters to change the implementation from fixed data widths to individually adjustable data widths that can be chosen at compile time. Set the **default** for each data input to **8 bits wide**.

This means all data bus widths through the design must be flexible!

- _____ 4. Save the design and synthesize.

- _____ 5. Check your logic implementation.

- a. Use the RTL Viewer to look at the widths of the I/O buses and the register stages.
- b. Try changing the parameter settings to ensure that the logic responds accordingly.
- c. Check synthesis messages for warnings that would indicate code not being interpreted correctly.

When you have verified your logic is correct, then you are ready to move to the pipeline multiplier.

Step 3: Parameterize the pipelined multiplier block

- ____ 1. Select the **three_mult_pipe** revision. From the top of the Quartus II window, use the drop-down list to select the **three_mult_pipe** project revision.

*The Project Navigator hierarchy display should show **three_mult_pipe** as the top-level entity in the revision.*

- ____ 2. Open the **three_mult_pipe.v** file and review it.

*This code is a pipelined, or 3-cycle, version of the 4-input multiplier block used in the previous step. Additional temporary variables (**abtemp**, **cdtemp**) are used to store the results of $a * b$ and $c * d$, creating an additional pipeline stage for the multiplier.*

- ____ 3. As in the previous step, use Verilog parameters to change the implementation from fixed data widths to individually adjustable data widths that can be chosen at compile time. Set the default for each data input to 8 bits wide.

*Again, all data paths through the design must be flexible! Of course, copying portions of your code over from **three_mult.v** might accelerate this step a bit.*

- ____ 4. Save the design and synthesize.
- ____ 5. As in the previous step, check and test your logic implementation.

Step 4: Use generate region & parameter to select multiplier

- ____ 1. Select the **top_mult** revision from the top of the Quartus II window.

- ____ 2. Open the file **mult.v** from the **File** menu.

*This file should look similar to what you've done in the previous steps. Here you will add an additional parameter and a generate region so that the top-level module (**top_mult**) can select a non-pipelined or pipelined implementation of the 4-input multiplier.*

- ____ 3. Edit the code for **mult.v** to include the following:
- A **PIPELINE** parameter that can be set to 0 or 1.
 - A generate region that uses the **PIPELINE** parameter to choose between instantiating **three_mult** and **three_mult_pipe**. Logically, **PIPELINE** set to 0 should choose the non-pipelined multiplier and be the default, while **PIPELINE** set to 1 should choose the pipelined version.
- ____ 4. Save **mult.v** when finished.

____ 5. Open the top-level file **top_mult.v**.

*Notice that it instantiates **mult** with the PIPELINE parameter set to 0. Also notice that the input data widths have been fixed to 12 bits wide.*

____ 6. Synthesize the design and fix any errors.

____ 7. As before, check and test your logic implementation. Try:

- a. Viewing the design implementation in the RTL Viewer. This time, you will have to burrow down a couple of levels to see the multipliers. (Remember generate regions create structural designs, i.e. hierarchy!) Hover over a hierarchical block, and a down arrow will be attached to the cursor. Double-click the block to descend the hierarchy. Double-click outside of a block to climb back up.
- b. Changing the PIPELINE parameter in **top_mult.v** from 0 to 1.
- c. Changing the top-level bus widths.

Step 5 (Optional)

If there is extra time at the end of class, try using the generate block without instantiations. For instance, the last example in the parameterized code section of the presentation shows the use of the generate region directly with procedural blocks.

*In this optional step, instead of creating **three_mult** and **three_mult_pipe** instantiations, copy their procedural code directly into **mult.v**. Then, your generate region would be selecting between `always` blocks, not module instantiations.*

Exercise Summary

- Added Verilog parameters to add flexibility to designs
- Used generate regions to perform structural modeling through selective instantiation

END OF EXERCISE 5