



MAX+PLUS II
Laboratory Exercise Manual
for
Introduction to Verilog

Introduction to Verilog Lab Overview

Objective : Build a sequential 8 X 8 multiplier

The objective of the following exercises is to build an 8 X 8 multiplier. The input to the multiplier consists of two 8-bit multiplicands (a[7:0], b[7:0]) and the output from the multiplier is a 16-bit result (result[15:0]). Additional outputs are a done bit (DONE_FLAG) and seven signals to drive a seven segment display, (A,B,C,D,E,F,G).

There are several methods of implementing a multiplier, the method chosen for the Verilog labs is the sequential multiplier method. This 8 X 8 multiplier requires four clock cycles to perform the full multiplication. During each cycle, a pair of 4-bit portion of the multiplicands are multiplied in a 4 X 4 multiplier. The multiplication result of these 4 bit slices are then accumulated. At the end of the four cycles, the full composed 16-bit result can be read at the output.

The following equations illustrate the mathematical principles supporting this implementation:

$$\begin{aligned}
 \text{result}[15:0] &= a[7:0] * b[7:0] \\
 &= ((a[7:4] * 2^4) + a[3:0] * 2^0) \\
 &\quad * ((b[7:4] * 2^4) + b[3:0] * 2^0) \\
 &= ((a[7:4] * b[7:4]) * 2^8) \\
 &\quad + ((a[7:4] * b[3:0]) * 2^4) \\
 &\quad + ((a[3:0] * b[7:4]) * 2^4) \\
 &\quad + ((a[3:0] * b[3:0]) * 2^0)
 \end{aligned}$$

Figure 1 in the following page illustrates the top level block diagram of the 8 X 8 multiplier.

The labs are structured as a bottom-up design approach. In each of the first seven exercises, you will use targeted features of the Verilog language to build the individual components of the 8 X 8 multiplier. Then, in exercise 5 you will put everything together in a top level design. You will then compile and simulate to verify the completeness of your design.

Good luck and have fun going through the exercises!

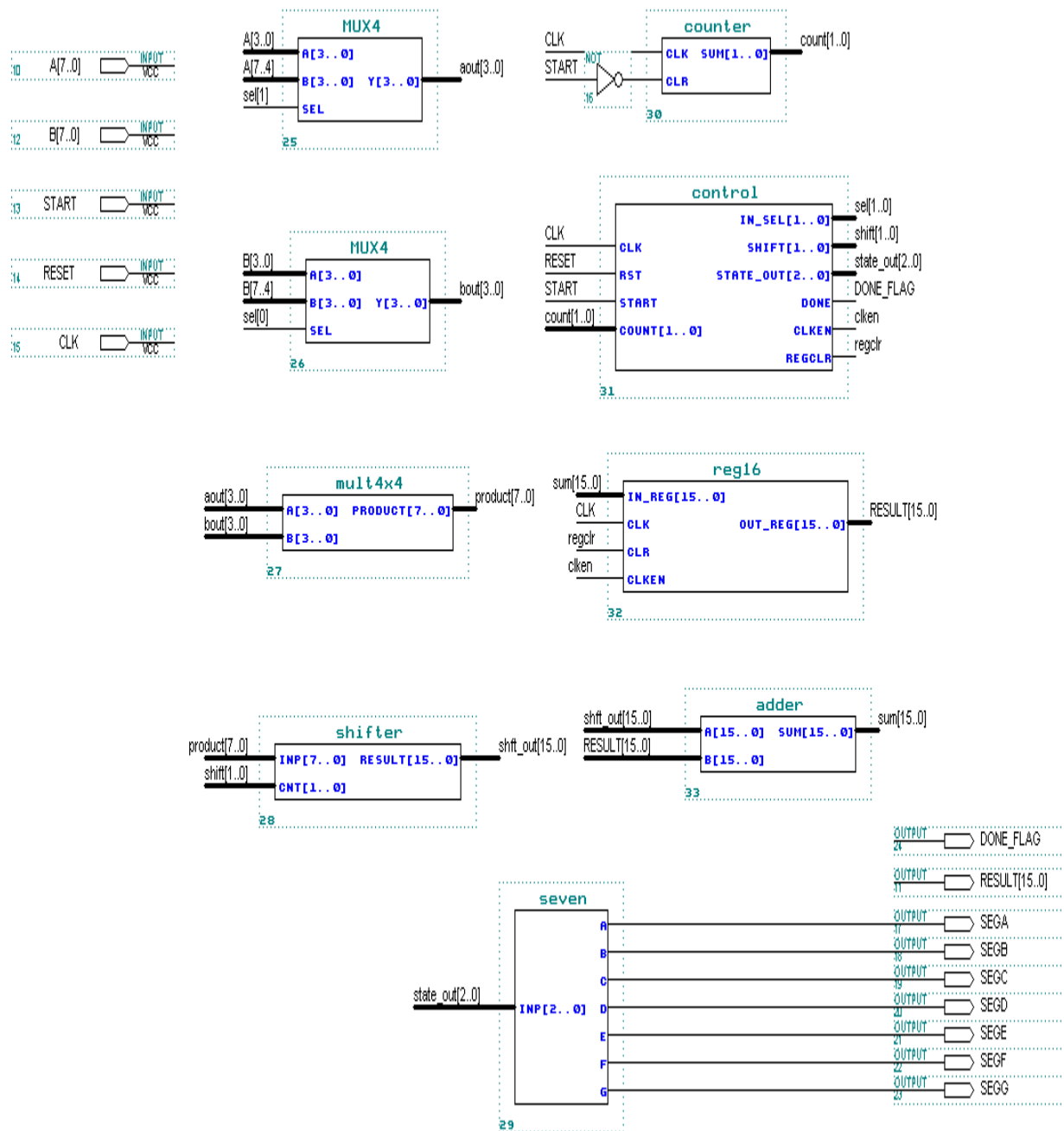


Figure 1 - 8 X 8 multiplier top level design block diagram

Exercise 1

Exercise 1

Part 1:

Objective: Build a 16-bit adder

The 16-bit adder can be constructed using the + operator. It is used to perform the additions in the following equation:

$$\begin{aligned} \text{result}[15:0] &= a[7:0] * b[7:0] \\ &= ((a[7:4] * b[7:4]) * 2^8) \\ &\quad + ((a[7:4] * b[3:0]) * 2^4) \\ &\quad + ((a[3:0] * b[7:4]) * 2^4) \\ &\quad + ((a[3:0] * b[3:0]) * 2^0) \end{aligned}$$

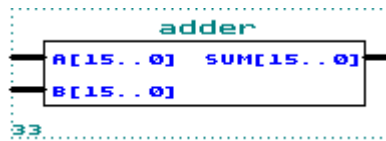




Figure 1-1a.

Step 1 (Create new text file and set to current project)

1. Create a new VERILOG text file. 
2. Under **File Type:** Choose **Text Editor File**. Click **OK**.
3. Save new VERILOG text file to <path>\altera_trn\ver\lab1\adder.v.
Go to File menu and Choose Save As.
4. Set adder.v as the current project. 
5. Write your code.

Remember to use the same input and output port names as shown in Figure 1-1a.



Step 2 (Save and check design)

Save and check adder.v.




Save and check checks for syntax and semantic errors. The compiler should stop on the Compiler Netlist Extractor if there are no error messages.

Step 3 (Do a functional compilation)

1. Bring the compiler to the foreground. 
2. Go to **Processing** menu and Turn on **Functional SNF Extractor**.
3. Run the compiler. 

Step 4 (Do a functional simulation)

1. The stimulus file has been created for you to verify the functionality of your design. If you are interested in learning how to create your own stimulus file, please go to the Appendix of this manual.
2. Open simulator. 
3. In the simulator window, next to Simulation Input:, it should say adder.scf. If it doesn't, **Go to File menu and Choose Inputs/Outputs. Choose adder.scf.**
4. Click on **START**.
5. Check to see if you get the same results shown in Figure 1-1b.

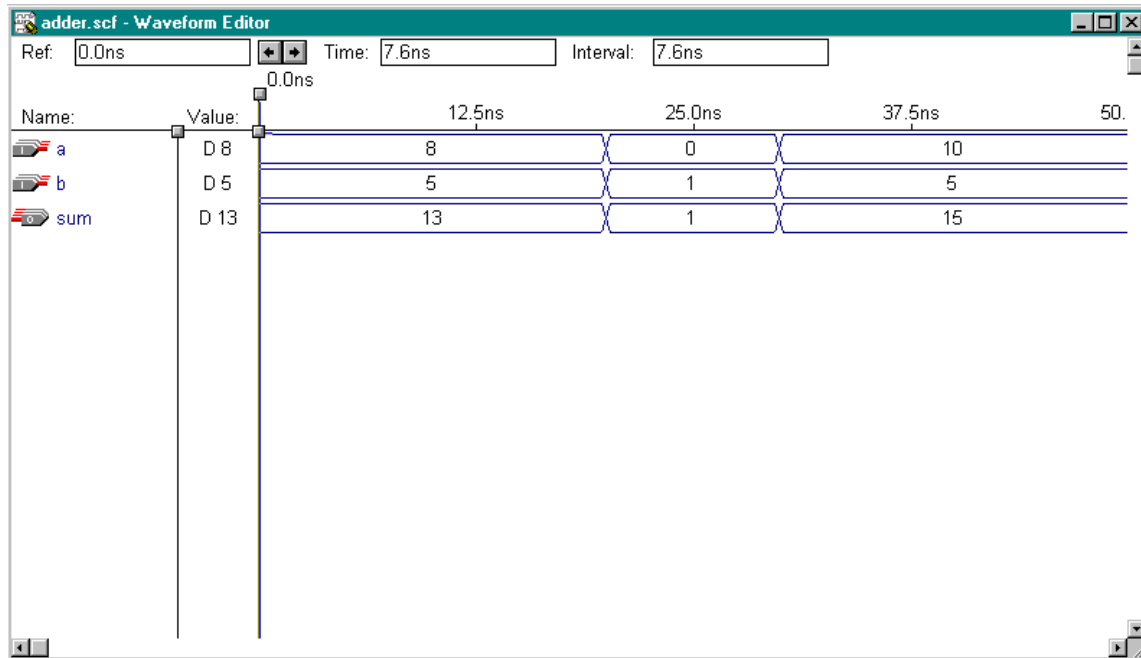


Figure 1-1b.

Part 2:

Objective: Build a 4x4 multiplier

The last component you will build is a 4x4 multiplier.

The 4x4 multiplier will be used to perform the 4-bit slice multiplication operation of the following equation:

$$\begin{aligned}
 \text{result}[15:0] &= a[7:0] * b[7:0] \\
 &= ((a[7:4] * b[7:4]) * 2^8) \\
 &\quad + ((a[7:4] * b[3:0]) * 2^4) \\
 &\quad + ((a[3:0] * b[7:4]) * 2^4) \\
 &\quad + ((a[3:0] * b[3:0]) * 2^0)
 \end{aligned}$$

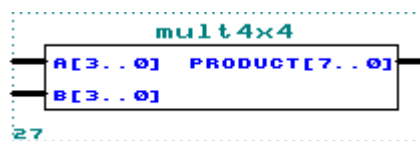





Figure 1-2a.

Step 1 (Create new text file and set to current project)

1. Create a new VERILOG text file. 
2. Under **File Type:** Choose **Text Editor File**. Click **OK**.
3. Save new VERILOG text file to <path>\altera_trn\ver\lab1\mult4x4.v.
Go to File menu and Choose Save As.
4. Set mult4x4.v as the current project. 
5. Write your code.



Remember to use the same input and output port names as shown in Figure 1-2a.

Step 2 (Save and check design)


Save and check mult4x4.v. 

Save and check checks for syntax and semantic errors. The compiler should stop on the Compiler Netlist Extractor if there are no error messages.

Step 3 (Do a functional compilation)

1. Bring the compiler to the foreground. 
2. Go to **Processing** menu and Turn on **Functional SNF Extractor**.
3. Run the compiler. 

Step 4 (Do a functional simulation)

1. The stimulus file has been created for you to verify the functionality of your design. If you are interested in learning how to create your own stimulus file, please go to the Appendix of this manual.
2. Open simulator. 
3. In the simulator window, next to Simulation Input:, it should say mult4x4.scf. If it doesn't, **Go to File menu and Choose Inputs/Outputs. Choose mult4x4.scf.**
4. Click on **START**.
5. Check to see if you get the same results shown in Figure 1-2b.

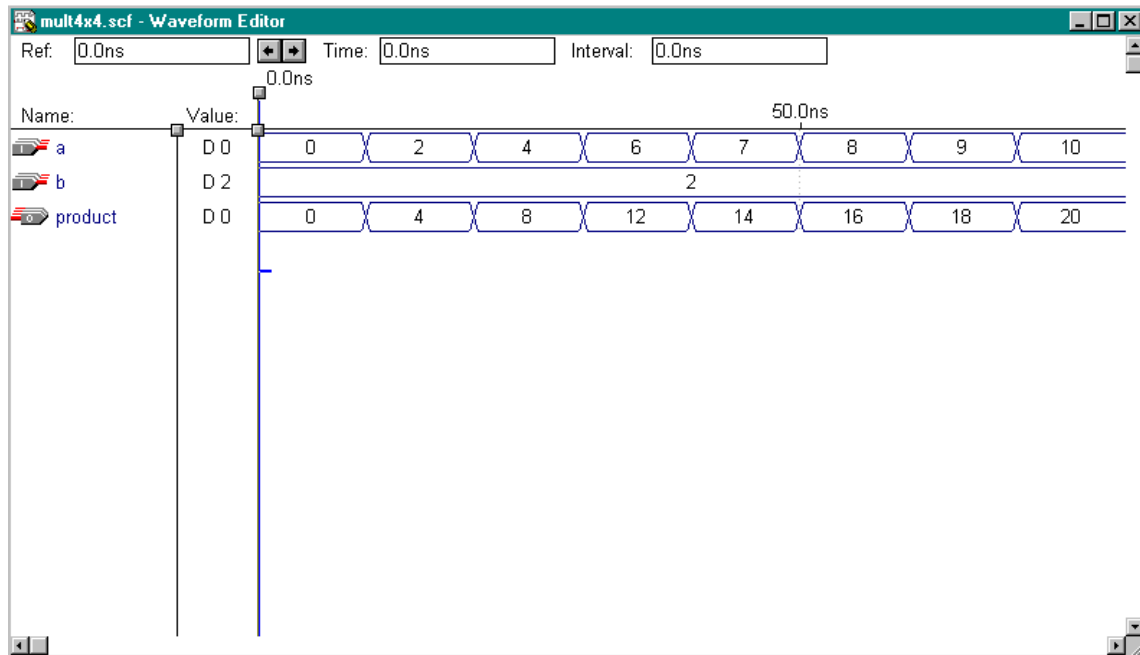


Figure 1-2b.

Exercise 2

Exercise 2

Part 1:

Objective: Build a four input 2:1 multiplexer using IF-ELSE statement

The first component you will build is a four input 2:1 multiplexer.

The input to the multiplexer consists of two 4-bit data buses (a[3:0] and b[3:0]). The output (y[3:0]) is a[3:0] if the select control (sel) is low (0). The output is b[3:0] if sel is high (1).

The four input 2:1 multiplexer will be used in the top level design for selecting the 4-bit slices a[7:4], a[3:0], b[7:4], and b[3:0] as inputs to the 4 X 4 multiplier.

The following equations are for your reference:

$$\begin{aligned}
 \text{result}[15:0] &= a[7:0] * b[7:0] \\
 &= ((a[7:4] * 2^4) + a[3:0] * 2^0) \\
 &\quad * ((b[7:4] * 2^4) + b[3:0] * 2^0) \\
 &= ((a[7:4] * b[7:4]) * 2^8) \\
 &\quad + ((a[7:4] * b[3:0]) * 2^4) \\
 &\quad + ((a[3:0] * b[7:4]) * 2^4) \\
 &\quad + ((a[3:0] * b[3:0]) * 2^0)
 \end{aligned}$$

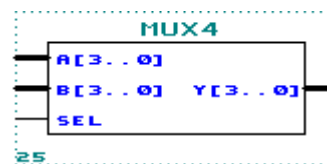


Figure 2-1a.


Step 1 (Create new text file and set to current project)

1. Create a new VERILOG text file. 
2. Under **File Type:** Choose **Text Editor File**. Click **OK**.
3. Save new VERILOG text file to <path>\altera_trn\ver\lab2\mux4.v. **Go to File menu and Choose Save As.**
4. Set mux4.v as the current project. 

5. Write your code.



Remember to use the same input and output port names as shown in Figure 2-1a.

Step 2 (Save and check design)

Save and check mux4.v. 

Save and check checks for syntax and semantic errors. The compiler should stop on the Compiler Netlist Extractor if there are no error messages.

Step 3 (Do a functional compilation)

1. Bring the compiler to the foreground. 
2. Go to **Processing** menu and Turn on **Functional SNF Extractor**.
3. Run the compiler. 

Step 4 (Do a functional simulation)


1. The stimulus file has been created for you to verify the functionality of your design. If you are interested in learning how to create your own stimulus file, please go to the Appendix of this manual.
2. Open simulator. 
3. In the simulator window, next to Simulation Input:, it should say mux4.scf. If it doesn't, **Go to File menu and Choose Inputs/Outputs. Choose mux4.scf.**
4. Click on **START**.
5. Check to see if you get the same results shown in Figure 2-1b.



Figure 2-1b.

Part 2:

Objective : Build an 8-bit to 16-bit shifter using IF-ELSE statement and shifting operators

In this exercise, you will build an 8-bit to 16-bit shifter using IF-ELSE statements and the shift-left operator (\ll). This shifter will be capable of perform three types of shifter operations: no shift, left shift by 4 bit positions, and left shift by 8 bit positions.

The input to the shifter consists of a single 8-bit data bus ($in[7:0]$). The shift operation is controlled by the control signal $cnt[1:0]$.

When $cnt[1:0] == 0$, the no shift operation will be selected.

When $cnt[1:0] == 1$, the left shift by 4 operation will be selected.

When $cnt[1:0] == 2$, the left shift by 8 operation will be selected.

When $cnt[1:0] == 3$, the no shift operation will be selected.

This 8-bit to 16-bit shifter will be used to perform the $* 2^0$ (no shift), $* 2^4$ (left shift by 4 bit positions), and $* 2^8$ (left shift by 8 bit positions) operations of the following equation:

$$\begin{aligned} \text{result}[15:0] &= a[7:0] * b[7:0] \\ &= ((a[7:4] * 2^4) + a[3:0] * 2^0) \\ &\quad * ((b[7:4] * 2^4) + b[3:0] * 2^0) \\ &= ((a[7:4] * b[7:4]) * 2^8) \\ &\quad + ((a[7:4] * b[3:0]) * 2^4) \\ &\quad + ((a[3:0] * b[7:4]) * 2^4) \\ &\quad + ((a[3:0] * b[3:0]) * 2^0) \end{aligned}$$

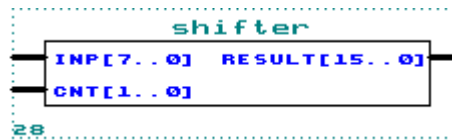





Figure 2-2a.

Step 1 (Create new text file and set to current project)

1. Create a new VERILOG text file. 
2. Under **File Type:** Choose **Text Editor File**. Click **OK**.
3. Save new VERILOG text file to <path>\altera_trn\ver\lab2\shifter.v.
Go to File menu and Choose Save As.
4. Set shifter.v as the current project. 
5. Write your code.



Remember to use the same input and output port names as shown in Figure 2-2a.

Step 2 (Save and check design)


Save and check shifter.v. 

Save and check checks for syntax and semantic errors. The compiler should stop on the Compiler Netlist Extractor if there are no error messages.

Step 3 (Do a functional compilation)

1. Bring the compiler to the foreground. 
2. Go to **Processing** menu and Turn on **Functional SNF Extractor**.
3. Run the compiler. 

Step 4 (Do a functional simulation)

1. The stimulus file has been created for you to verify the functionality of your design. If you are interested in learning how to create your own stimulus file, please go to the Appendix of this manual.
2. Open simulator. 
3. In the simulator window, next to Simulation Input:, it should say shifter.scf. If it doesn't, **Go to File menu and Choose Inputs/Outputs. Choose shifter.scf.**
4. Click on **START**.
5. Check to see if you get the same results shown in Figure 2-2b.

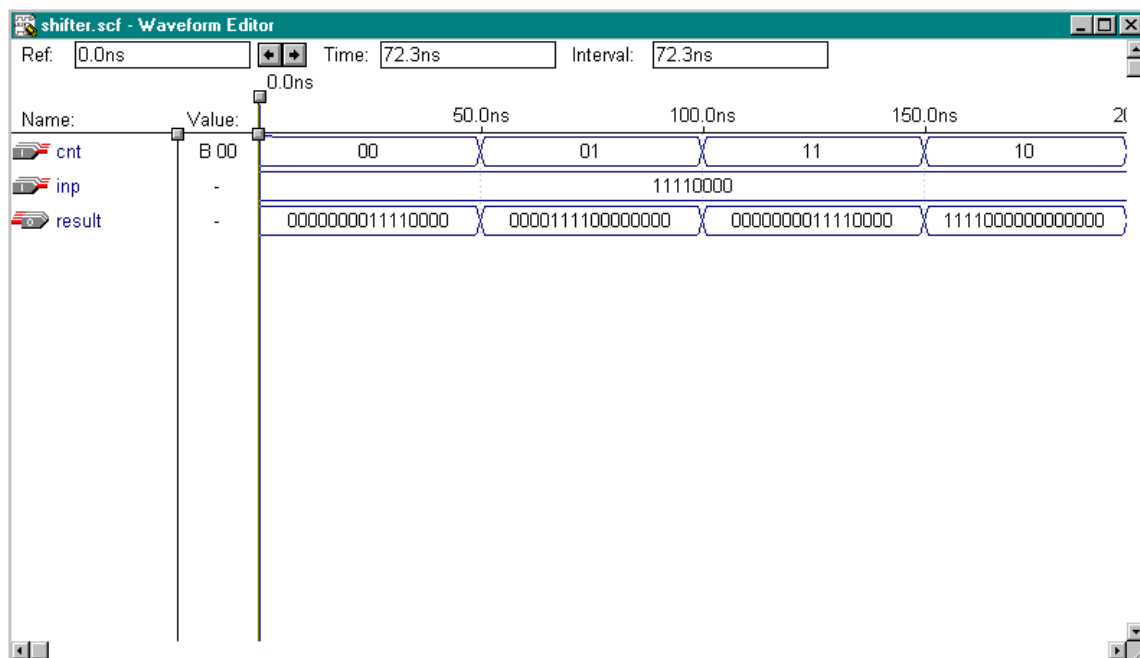


Figure 2-2b.

Exercise 3

Exercise 3

Objective: Build a 7-segment display using *CASE* statement

The 7-segment display shall display 0, 1, 2, 3, and E.

INPUTS			OUTPUTS							DISPLAY
IN2	IN1	IN0	a	b	c	d	e	f	g	
0	0	0	1	1	1	1	1	1	0	0
0	0	1	0	1	1	0	0	0	0	1
0	1	0	1	1	0	1	1	0	1	2
0	1	1	1	1	1	1	0	0	1	3
1	X	X	1	0	0	1	1	1	1	E

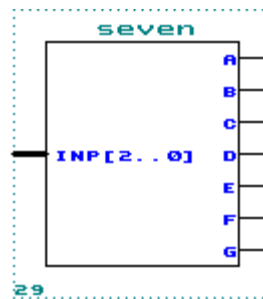




Figure 3-1.

Step 1 (Create new text file and set to current project)

1. Create a new VERILOG text file. 
2. Under **File Type:** Choose **Text Editor File**. Click **OK**.
3. Save new VERILOG text file to <path>\altera_trn\ver\lab3\seven.v.
Go to File menu and Choose Save As.
4. Set seven.v as the current project. 
5. Write your code.

Remember to use the same input and output port names as shown in Figure 3-1.



Step 2 (Save and check design)

Save and check seven.v.




Save and check checks for syntax and semantic errors. The compiler should stop on the Compiler Netlist Extractor if there are no error messages.

Step 3 (Do a functional compilation)

1. Bring the compiler to the foreground. 
2. Go to **Processing** menu and Turn on **Functional SNF Extractor**.
3. Run the compiler. 

Step 4 (Do a functional simulation)

1. The stimulus file has been created for you to verify the functionality of your design. If you are interested in learning how to create your own stimulus file, please go to the Appendix of this manual.
2. Open simulator. 
3. In the simulator window, next to Simulation Input:, it should say seven.scf. If it doesn't, **Go to File menu and Choose Inputs/Outputs. Choose seven.scf.**
4. Click on **START**.
5. Check to see if you get the same results shown in Figure 3-2.

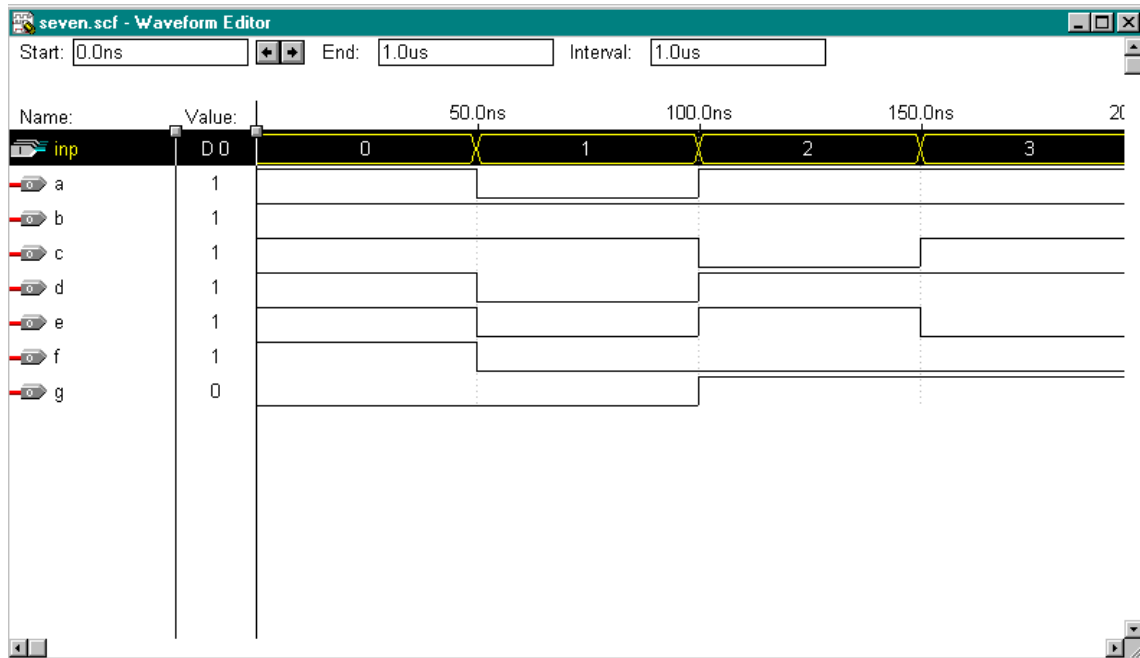


Figure 3-2.

Exercise 4

Exercise 4

Part 1:

Objective: Build a 16-bit register

It is used to store intermediate results.

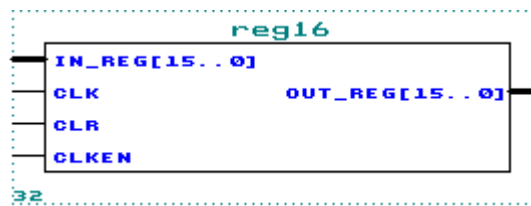




Figure 4-1a.

Step 1 (Create new text file and set to current project)

1. Create a new VERILOG text file. 
2. Under **File Type:** Choose **Text Editor File**. Click **OK**.
3. Save new VERILOG text file to <path>\altera_trn\ver\lab4\reg.v.

Go to File menu and Choose Save As.

4. Set reg.v as the current project. 

5. **Functionality of the 16-bit register:**

If clr==1 && clken==0, then in_reg will be loaded into register at the rising edge of clock.


Otherwise, if clr==0, then the register is cleared.

Note: This is a synchronous clear and clock enable register.

6. Write your code.



Remember to use the same input and output port names as shown in Figure 4-1a.

Step 2 (Save and check design)


Save and check reg.v. 

Save and check checks for syntax and semantic errors. The compiler should stop on the Compiler Netlist Extractor if there are no error messages.

Step 3 (Do a functional compilation)

1. Bring the compiler to the foreground. 
2. Go to **Processing** menu and Turn on **Functional SNF Extractor**.
3. Run the compiler. 

Step 4 (Do a functional simulation)

1. The stimulus file has been created for you to verify the functionality of your design. If you are interested in learning how to create your own stimulus file, please go to the Appendix of this manual.
2. Open simulator. 
3. In the simulator window, next to Simulation Input:, it should say reg.scf. If it doesn't, **Go to File menu and Choose Inputs/Outputs. Choose reg16.scf.**
4. Click on **START**.
5. Check to see if you get the same results shown in Figure 4-1b.

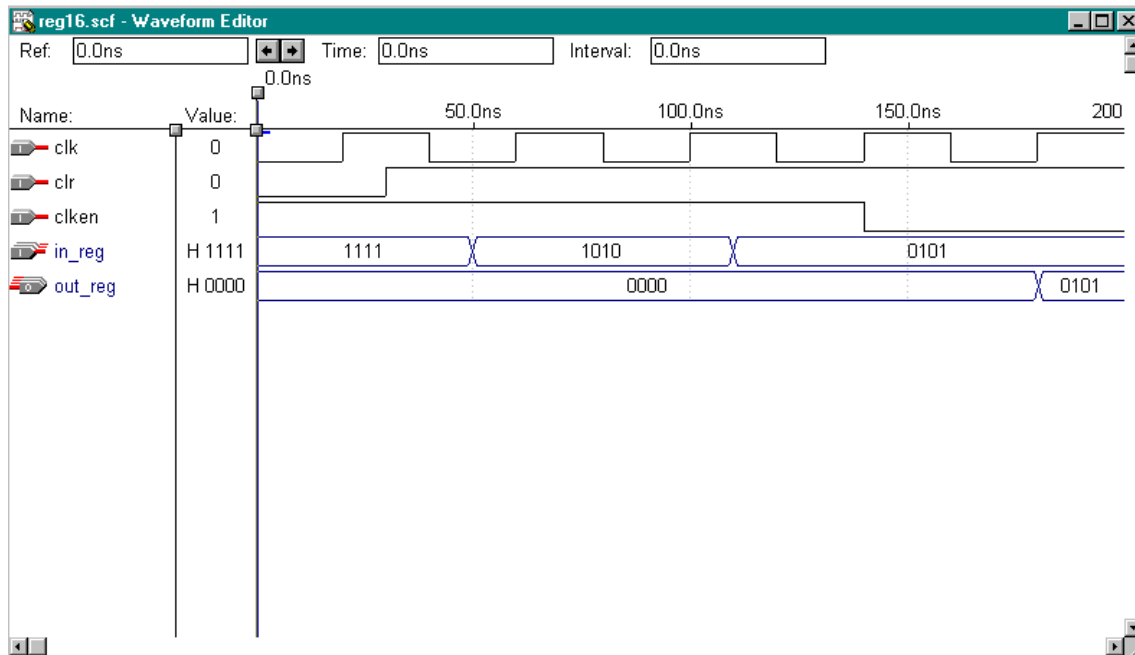


Figure 4-1b.

Part 2:

Objective: Build a 2-bit counter

The 2-bit counter can be constructed using the +1 operation. It is used to help the state machine track the cycles of the sequential multiplication.

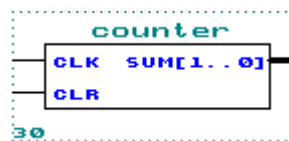




Figure 4-2a.

Step 1 (Create new text file and set to current project)

1. Create a new VERILOG text file. 
2. Under **File Type:** Choose **Text Editor File**. Click **OK**.
3. Save new VERILOG text file to <path>\altera_trn\ver\lab4\counter.v.
Go to File menu and Choose Save As.

4. Set counter.v as the current project. 

5. Functionality of the 2-bit counter:


If clr==1, then the counter counts at the rising edge of clock.

Otherwise, if clr==0, then the counter is cleared.

Note: This is a synchronous clear.



6. Write your code.
Remember to use the same input and output port names as shown in Figure 4-2a.

Step 2 (Save and check design)


Save and check counter.v. 

Save and check checks for syntax and semantic errors. The compiler should stop on the Compiler Netlist Extractor if there are no error messages.

Step 3 (Do a functional compilation)

1. Bring the compiler to the foreground. 
2. Go to **Processing** menu and Turn on **Functional SNF Extractor**.
3. Run the compiler. 

Step 4 (Do a functional simulation)

1. The stimulus file has been created for you to verify the functionality of your design. If you are interested in learning how to create your own stimulus file, please go to the Appendix of this manual.
2. Open simulator. 
3. In the simulator window, next to Simulation Input:, it should say counter.scf. If it doesn't, **Go to File menu and Choose Inputs/Outputs. Choose counter.scf.**
4. Click on **START**.
5. Check to see if you get the same results shown in Figure 4-2b.

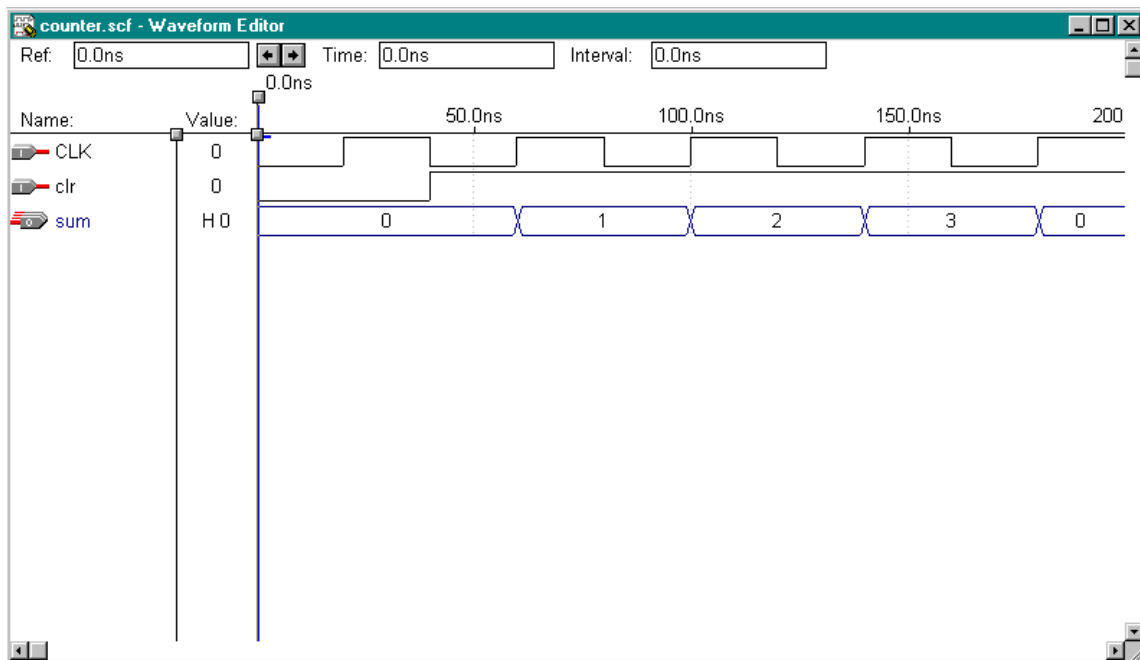


Figure 4-2b.

Exercise 5

Exercise 5

Objective: Putting all together by instantiating the lower-level components

You have now completed building all of the components necessary to build the 8x8 multiplier, except for the controlling state machine. Due to time, the controlling state machine has been written for you and is located in `c:\alter\int_Verilog\lab6\control.v`.

This state machine will manage all the operation that occurs within the 8 X 8 multiplier.

The state machine will perform the $((a[3:0] * b[3:0]) * 2^0)$ multiplication in the first cycle (LSB state) after the input signal *start* becomes a '1'. This intermediate result is saved in an accumulator.

In the second clock cycle (MID state), the $((a[3:0] * b[7:4]) * 2^4)$ multiplication is performed. The multiplication result is added with the content of the accumulator and clocked back into the accumulator.

In the third clock cycle (MID state), the $((a[7:4] * b[3:0]) * 2^4)$ multiplication is performed. The multiplication result is added with the content of the accumulator and clocked back into the accumulator.

In the fourth clock cycle (MSB state), the $((a[7:4] * b[7:4]) * 2^8)$ multiplication is performed. The multiplication result is added with the content of the accumulator and clocked back into the accumulator. This result is the final result:

$$\begin{aligned} \text{result}[15:0] &= a[7:0] * b[7:0] \\ &= ((a[7:4] * b[7:4]) * 2^8) \\ &\quad + ((a[7:4] * b[3:0]) * 2^4) \\ &\quad + ((a[3:0] * b[7:4]) * 2^4) \\ &\quad + ((a[3:0] * b[3:0]) * 2^0) \end{aligned}$$

NOTE: There are two inputs to the state machine *start* and *count*[1:0]. The *start* signal is a single cycle high-true signal. When *start* becomes a '1', it indicates that multiplication can begin at next clock cycle. The *start* signal can only be asserted for one clock cycle. The *start* signal shall stay a '0' until next 8 x 8 multiplication is to be performed. The *count*[1:0] signal is the output of a free running 2-bit counter. The *count*[1:0] signal is synchronously initialized by the *start* signal. *Count*[1:0] is used by the state machine to track the cycles of the multiplication.

Please also note that this is NOT the optimal design. The state machine design as you see it is intended for exercising your VERILOG skills and not the ability to perform optimum solution.

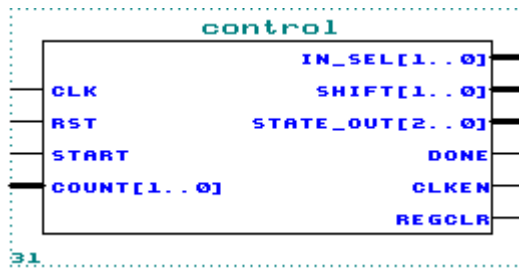
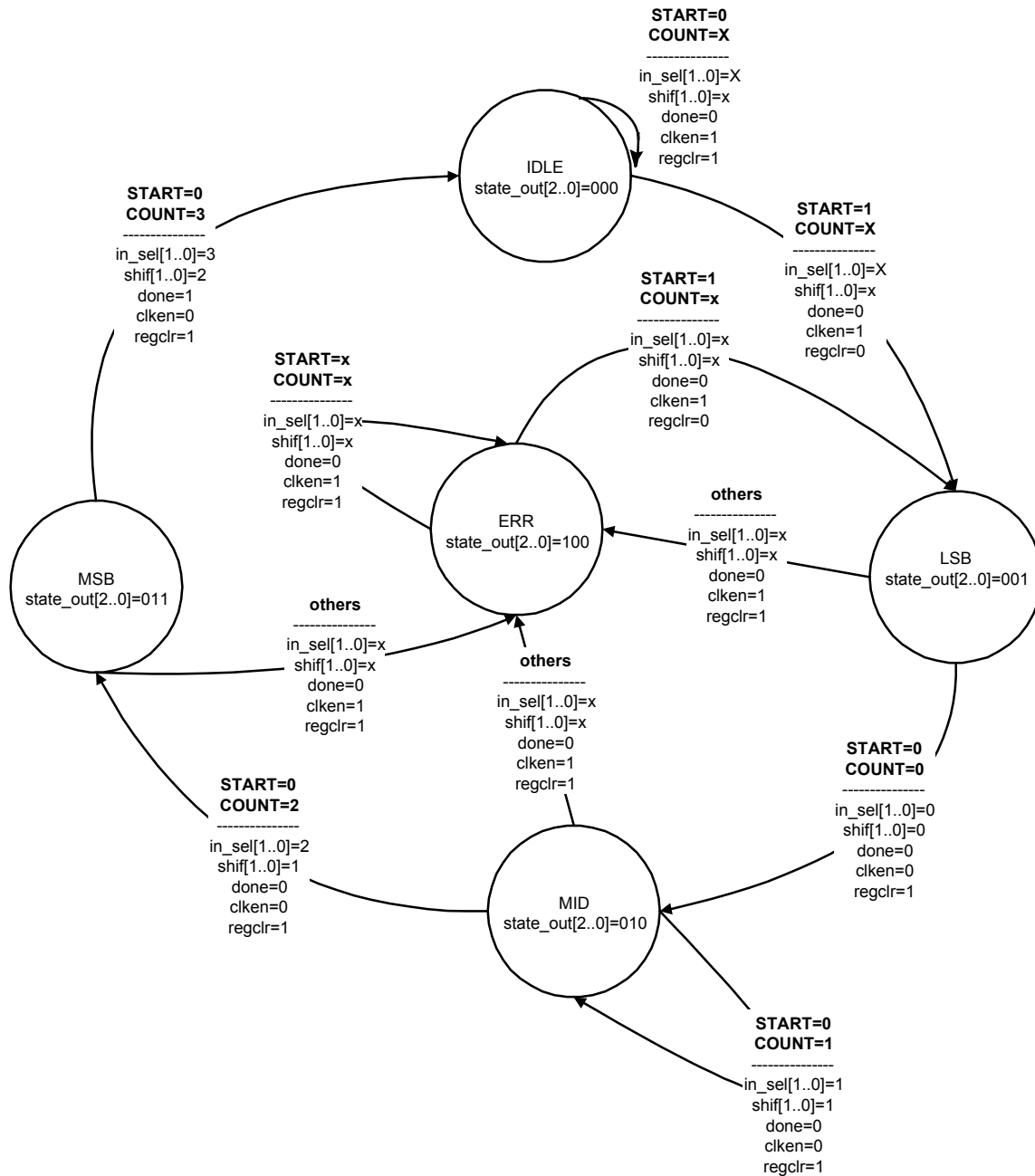


Figure 5.

Refer to the state diagram on the next page.



X means a “don’t care” in this state diagram.

Making use of the knowledge you have gained up to this point, you should instantiate each component in a top-level design and connect all signal as shown in Figure 6-1. You have successfully completed the Introduction to VERILOG class once your top-level design is compiled and simulated correctly.

Congratulations!

You have completed the implementation of the following 8x8 multiplier.

$$\begin{aligned} \text{result}[15:0] &= a[7:0] * b[7:0] \\ &= ((a[7:4] * b[7:4]) * 2^8) \\ &\quad + ((a[7:4] * b[3:0]) * 2^4) \\ &\quad + ((a[3:0] * b[7:4]) * 2^4) \\ &\quad + ((a[3:0] * b[3:0]) * 2^0) \end{aligned}$$

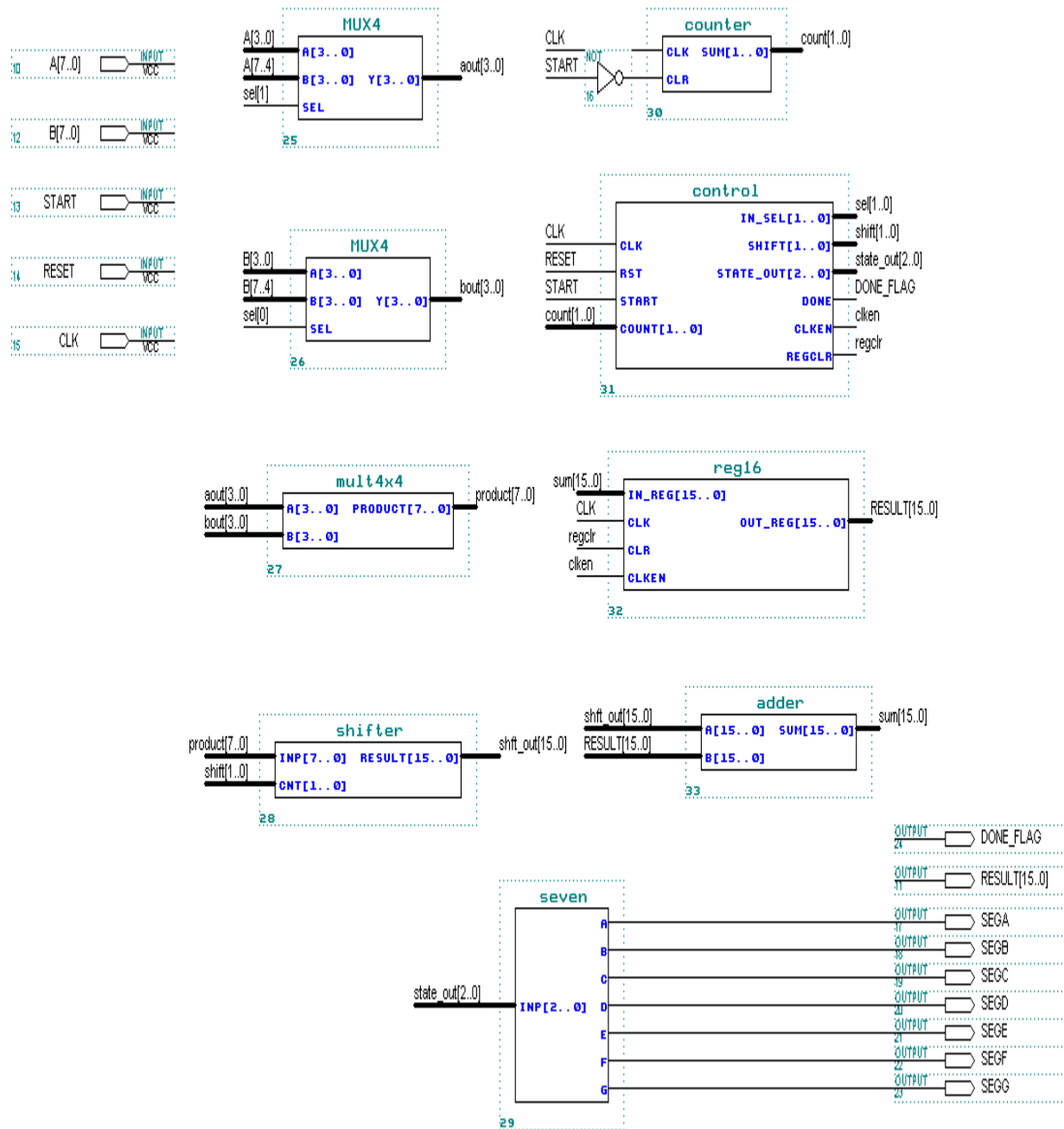





Figure 5-1.

Step 1 (Create new text file and set to current project)

1. Create a new VERILOG text file. 
2. Under **File Type:** Choose **Text Editor File**. Click **OK**.
3. Save new VERILOG text file to <path>\altera_trn\ver\lab5\mult8x8.v.
Go to File menu and Choose Save As.
4. Set mult8x8.v as the current project. 
5. **The lower-level components have been created in different directories than the current directory. In order for MAX+plus II to find these lower-level components, it must have a search path. Therefore, you must do the following:**
 - a) Go to **Options menu** and Choose **User Libraries**.
 - b) Choose the directory structure <path>\altera_trn\ver\lab1. Click on **ADD**.
 - c) Choose the directory structure <path>\altera_trn\ver\lab2. Click on **ADD**.
 - d) Choose the directory structure <path>\altera_trn\ver\lab3. Click on **ADD**.
 - e) Choose the directory structure <path>\altera_trn\ver\lab4. Click on **ADD**.
 - f) Click **OK**.
7. Write your code.


Remember to use the same input and output port names as shown in Figure 5-1.

Step 2 (Save and check design)


Save and check mult8x8.v. 

Save and check checks for syntax and semantic errors. The compiler should stop on the Compiler Netlist Extractor if there are no error messages.


Step 3 (Do a functional compilation)

1. Bring the compiler to the foreground. 
2. Go to **Processing** menu and Turn on **Functional SNF Extractor**.
3. Go to **Processing** menu and Turn on **Preserved All Node Name Synonyms**.

This option will preserve buried signal names.

4. Run the compiler. 

Step 4 (Do a functional simulation)

1. The stimulus file has been created for you to verify the functionality of your design. If you are interested in learning how to create your own stimulus file, please go to the Appendix of this manual.
2. Open simulator. 
3. In the simulator window, next to Simulation Input:, it should say mult8x8.scf. If it doesn't, **Go to File menu and Choose Inputs/Outputs. Choose mult8x8.scf.**
4. Click on **START**.
5. Check to see if you get the same results shown in Figure 5-2.

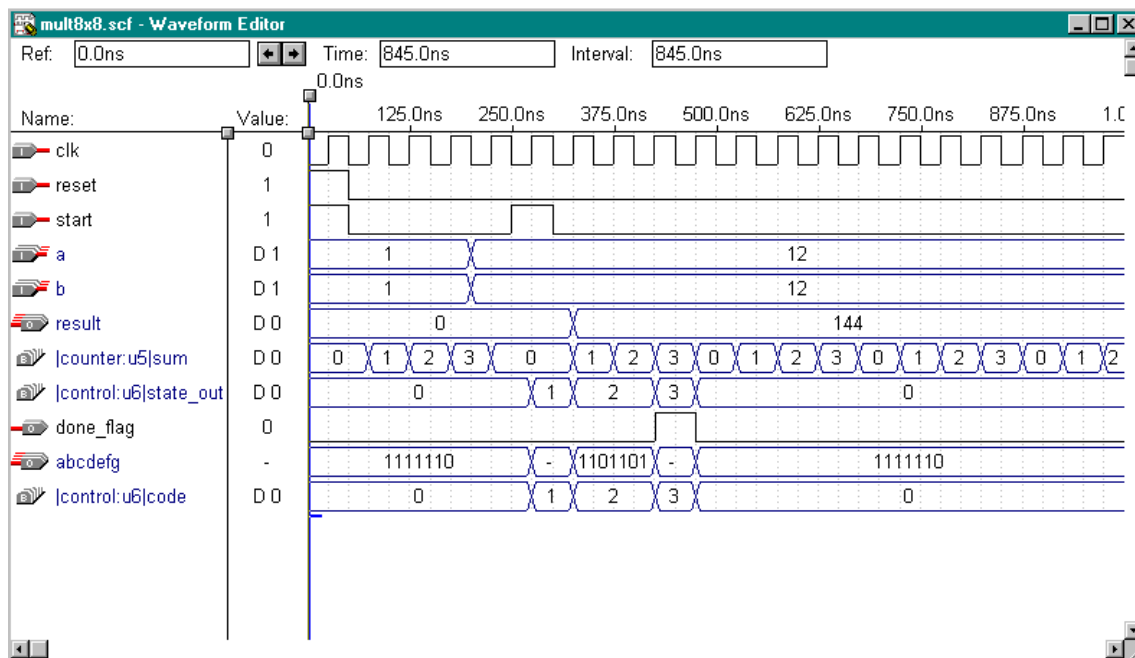




Figure 5-2.

APPENDIX

How to create a stimulus file:

1. Create a new waveform stimulus for simulation. 
2. Under **File Type**: Choose **Waveform Editor File**
3. Click **OK**.
4. Save file as mux4.scf. **File menu -> Save As**
5. Enter nodes into the waveform file. **Node menu -> Enter Nodes from SNF**
 - a. Go to **Type** section.
 - b. Turn on **Inputs**, **Outputs**, and **Groups**.
 - c. Click on **List**.
 - d. Highlight the nodes (you want to see in the waveform display) listed on **Available Nodes & Groups**:
 - e. Click on \Rightarrow to bring nodes to **Selected Nodes & Groups**.
 - f. Click on **OK**.
4. Set the inputs to the appropriate values.
5. Open simulator. 
6. Click on **START**.