Appendix C

Tutorial 2 — Implementing Circuits in Altera Devices

In this tutorial we describe how to use the physical design tools in Quartus II. In addition to the modules used in Tutorial 1, the following Quartus II modules are introduced: Fitter, Floorplan Editor, and Timing Analyzer. To illustrate the procedures involved, we will first implement the *example_verilog* project created in Tutorial 1 in a MAX 7000 CPLD.

C.1 Implementing a Circuit in a MAX 7000 CPLD

Select File | Open Project and browse to the directory *designstyle*2, which contains the Verilog design example used in Tutorial 1. As depicted in Figure C.1, select the *example_verilog* project (Quartus II project files have the filename extension *.qpf*) and click Open.

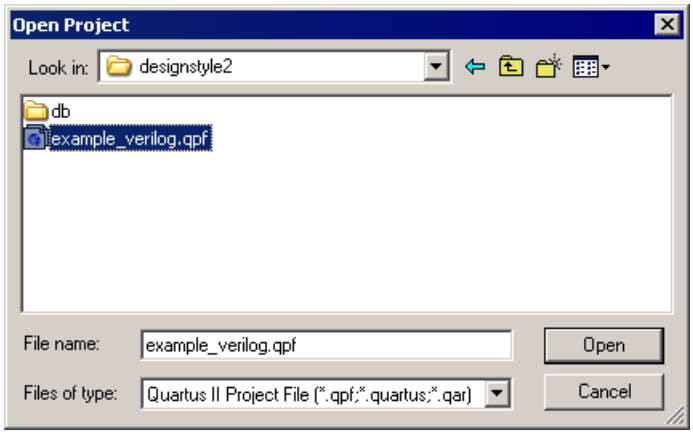


Figure C.1. Opening the *example_verilog* project.

C.1.1 Selecting a Chip

In Tutorial 1 we used the Compiler to perform the synthesis operations, which generated the information needed for functional simulation. Now, we will implement the design in a CPLD and then use timing simulation.

To specify which chip to use, select Assignments | Device to open the window shown in Figure C.2. To select the MAX 7000 CPLD family, click on the pull-down menu in the box labeled Family and select MAX7000S. The S at the end of the name refers to the members of the MAX 7000 family that are in-system programmable. Methods of CPLD programming are discussed in Chapter 3, in section 3.6.4. Note that in some cases Quartus II will display the message "Device family selection has changed. Do you want to remove all pin assignments?" Click Yes to close this pop-up box.

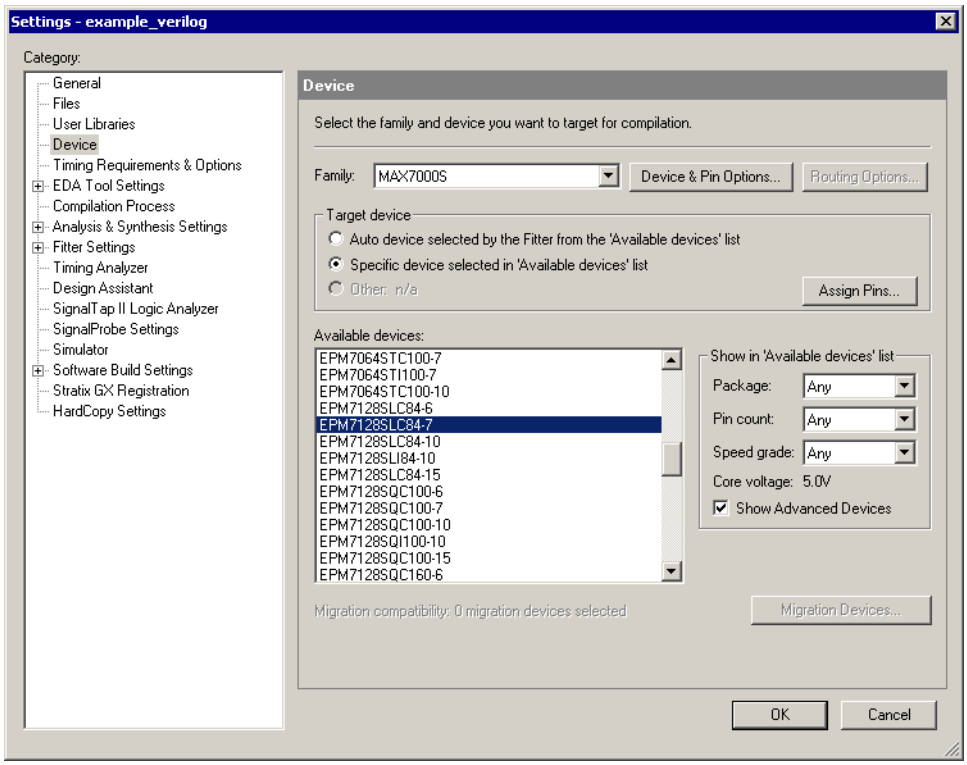


Figure C.2. Selecting a MAX7000S device.

In the Target device box you can specify that Quartus II should automatically select a device during compilation. The ability to have a chip chosen automatically is sometimes convenient for the designer. However, in this case we wish to select a specific chip, so click on Specific device selected in 'Available devices' list.

The available chips in the MAX 7000S family are displayed in the box labeled Available devices. One available chip is the EPM7128SLC84-7 (if this device is not listed, change the Speed Grade item in the Filter box to Any). The meaning of the chip name is as follows: The EPM7 means that the chip is a member of the MAX 7000 family, and the 128 gives the number of macrocells in the chip. The designator LC84 indicates an 84-pin PLCC package; this type of package is described in section 3.6.3. The -7 gives the *speed grade*. We discuss speed grades in Appendix E. As indicated in Figure C.2, click on the EPM7128SLC84-7 device, then click OK to close the Settings window. We have chosen this chip because it is provided on an Altera development board that is discussed in Appendix D.

C.1.2 Compiling the Project

In Appendix B we ran just the synthesis tools in Quartus II, by using the command Processing | Start | Start Analysis & Synthesis. Now, we wish to run not only the synthesis tools, but also a number of other tools that implement the circuit in the target device. To invoke all the needed tools, select Processing | Start Compilation, or use the toolbar icon that looks like a solid purple triangle. This runs in sequence four of the modules in Figure B.16: Synthesis, Fitter, Assembler, and Timing Analyzer. As we saw in Tutorial 1, the compilation progress through each Quartus II module is displayed in the Status window on the left side of the Quartus II display. After the Analysis & Synthesis module converts the Verilog code into a circuit that comprises macrocells, the Fitter module chooses locations on the device for these macrocells.

When compilation is finished, the compilation report displayed in Figure C.3 is produced. As we said in Tutorial 1, there is a lot of useful information in this report. Click on the small + symbol to expand the Fitter section of the report, and then click on the Fitter Equations section to reach the display in Figure C.4. Scroll through this part of the report to see the logic expressions implemented by our circuit. At the bottom of the report the output f is given as

$$\mathsf{f} = \mathsf{OUTPUT}(\mathsf{A1L6});$$

This means that f appears on an output pin, and that output is defined by the logic expression called A1L6, which is realized as indicated near the top of the Fitter Equations section in Figure C.4. These expressions properly implement our logic function $f = x_1x_2 + \overline{x}_2x_3$.

C.1.3 Performing Timing Simulation

Timing simulation is done by using the same procedure that we described in Tutorial 1 for functional simulation. Select Assignments | Settings and click on the Simulator item, as shown in Figure B.24. Open the drop-down list next to Simulation mode and change this setting from Functional to Timing.

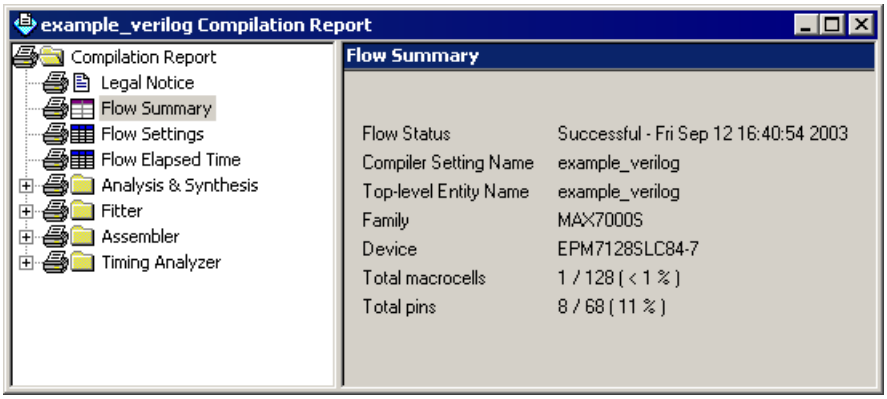


Figure C.3. The compilation summary.

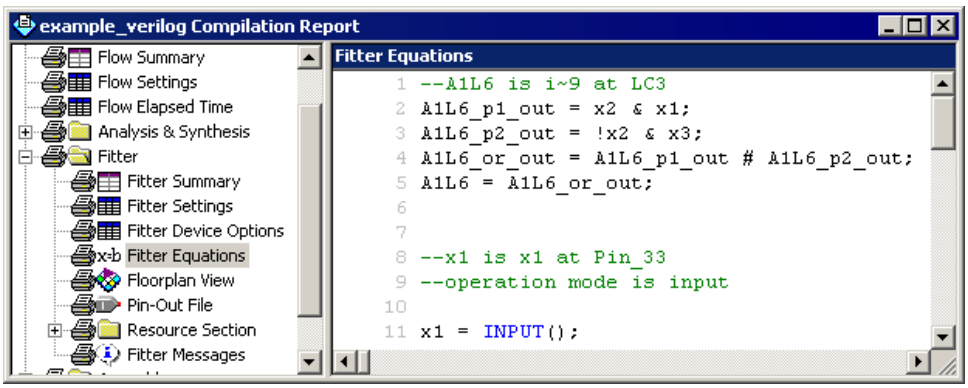


Figure C.4. The Fitter Equations section.

Use the input waveforms for $x1$, $x2$, and $x3$ that were drawn with the Waveform Editor in Tutorial 1 as inputs for the timing simulation. Select Processing | Start Simulation to run the simulation. When it is completed, the simulation report is displayed. Part of this report is shown in Figure C.5. Select View | Fit in Window to see the complete time range of the waveforms. Compare these waveforms to those shown in Figure B.25. The timing simulation produces the same results as the functional simulation in Tutorial 1 except that the times at which changes in *f* occur are now determined by the timing characteristics of the EPM7128SLC84-7 chip.

We can use the vertical reference line in the display to determine the exact time when f changes value. To do this select View | Snap to Transition, so that your mouse pointer will align perfectly with an edge on any waveform. Click and drag the vertical reference line to the point where f first changes to 1, as shown in the figure. The box labeled Master Time Bar now displays 27.5 ns, meaning that it takes 7.5 ns for the change in $x3$, which occurs at 20 ns, to cause a change in *f*. This result is a reflection of the -7 speed grade of the chip, which is specified as having a delay from an input to an output pin of 7.5 ns.

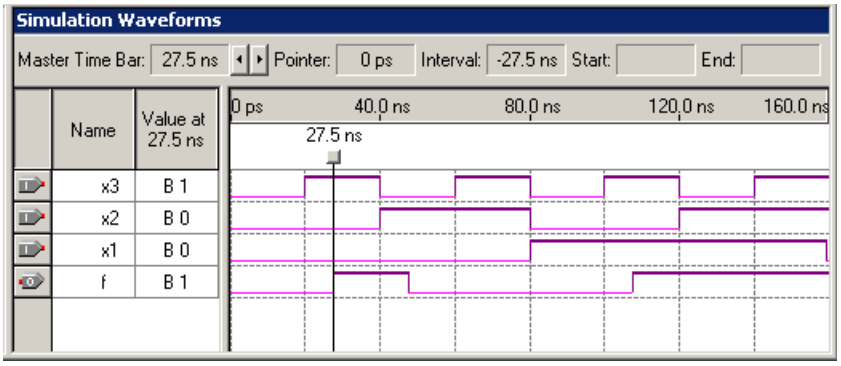


Figure C.5. The Timing Simulation Report.

C.1.4 Using the Floorplan Editor

In addition to examining the equations in the compilation report, another way to view the implementation results is to use the Floorplan Editor. Select Assignments | Timing Closure Floorplan to open the window shown in Figure C.6. Another way to open this window is to click on the corresponding icon in the toolbar. To make the window look like the one in the figure, it may be necessary to change the setting in the Floorplan tool by selecting View | Interior Cells, which causes the macrocells in the device to be displayed.

Figure C.6 shows some of the macrocells in the EPM7128SLC84-7 chip. As we describe in Appendix E, the macrocells are organized into logic array blocks (LABs), where each LAB contains 16 macrocells. To see larger or smaller views of the LABs, click on the magnify buttons in the vertical toolbar; left-click to enlarge the image and right-click to reduce it. To display different sections of the chip, use the window scroll bars.

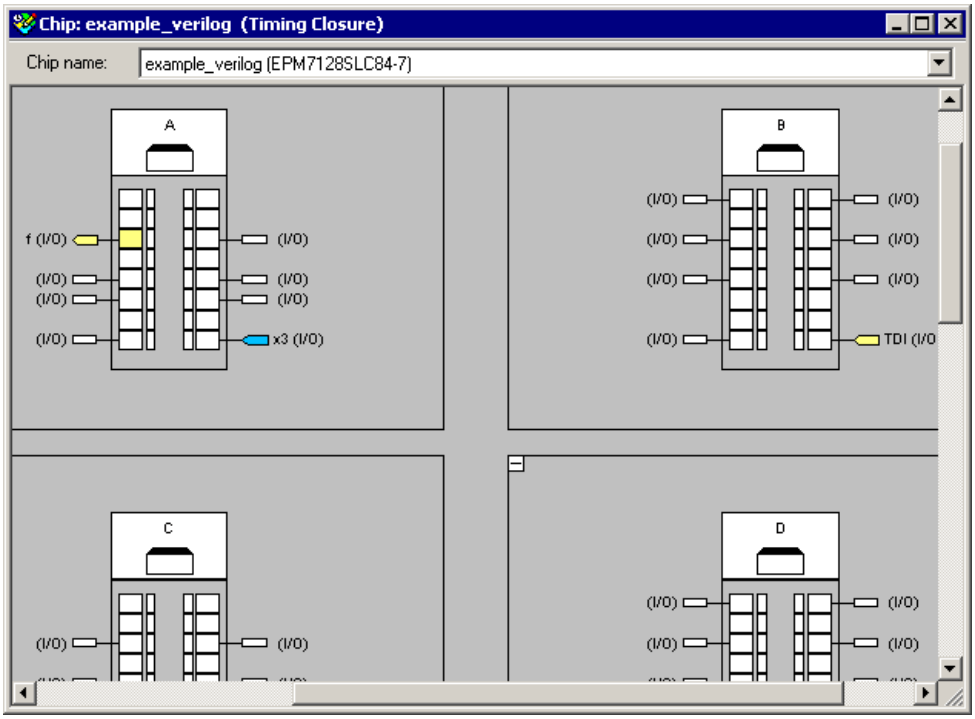


Figure C.6. The Timing Closure Floorplan display.

The Floorplan Editor uses different colors to indicate macrocells that are used in a circuit and macrocells that are unused. For our small example three pins are used for the three inputs to the circuit, and one macrocell provides the circuit output. Adjust the display so that the macrocell that produces the output *f* is visible, as depicted in Figure C.7. Click on this macrocell to select it. The Floorplan Editor can draw lines that indicate which other macrocells the selected macrocell is connected to by choosing View | Routing | Show Node Fan-In. It is also possible to see what logic function is implemented in the selected node by selecting View | Equations. As seen in the figure, this choice displays the logic expressions from the compilation report in the bottom part of the Floorplan window.

Instead of displaying the macrocells, the floorplan tool can alternatively display a picture of the pins on the chip package. To change to this view, select View | Package Top. This leads to the display in Figure C.8. To close the report file equation viewer, select again View | Equations to toggle off this feature.

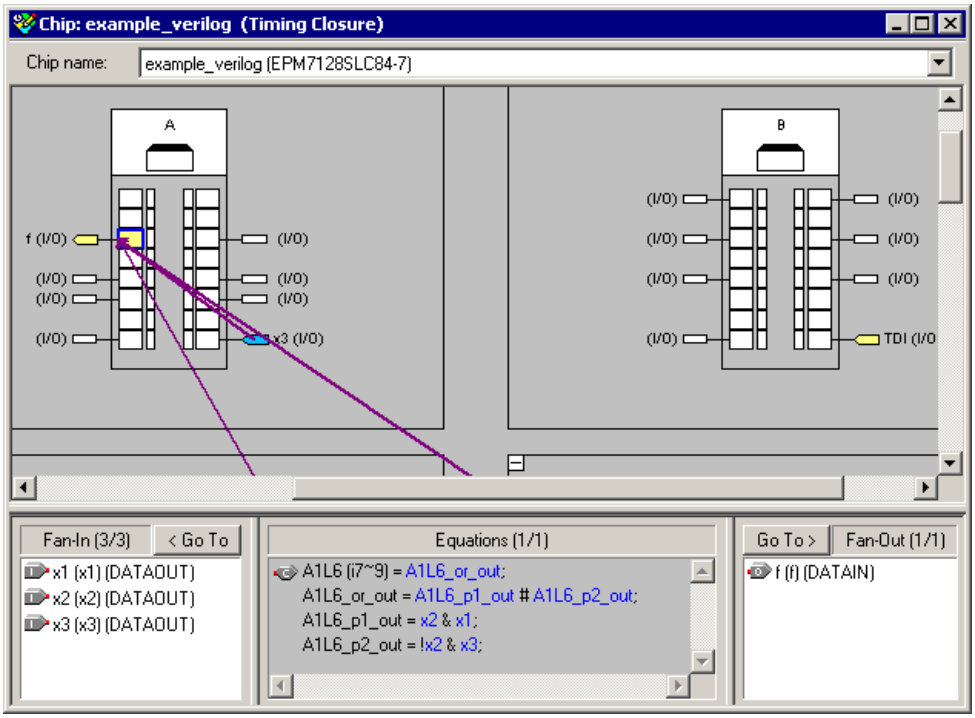


Figure C.7. Viewing node fan-in and equations.

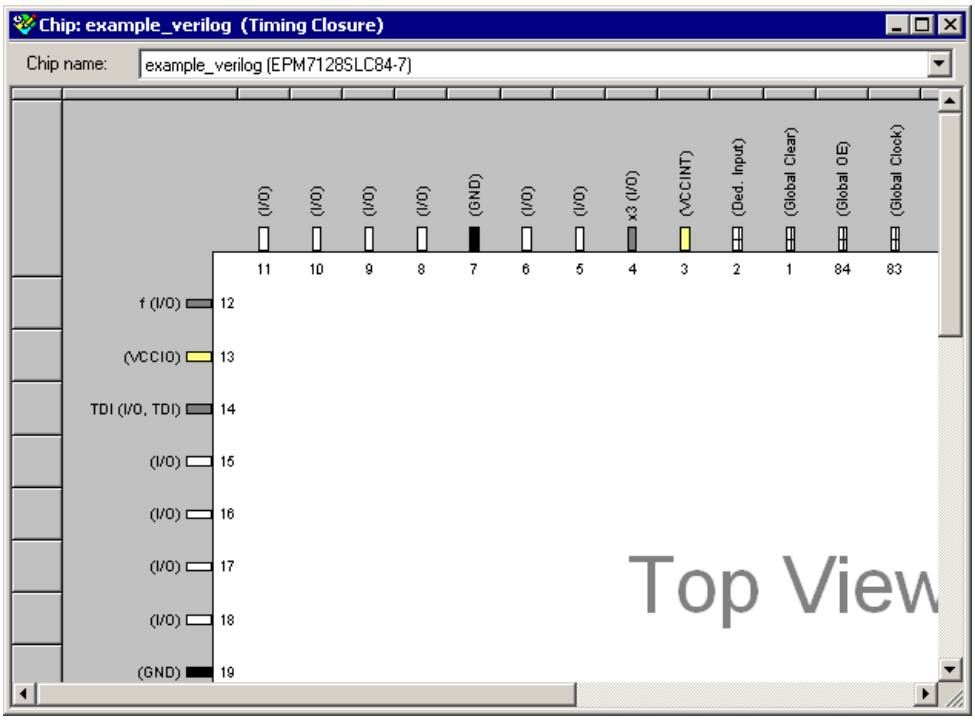


Figure C.8. The package top view.

The Floorplan tool is not essential in the CAD flow described above. It just provides a graphical view of the information contained in the compilation report. We will describe a different use of the Floorplan tool in Appendix D, in which it will be used to modify the implementation results produced by the Compiler, instead of just displaying them.

We have now completed the implementation of the *example_verilog* project in a MAX 7000 chip. Close the project.

C.2 Implementing a Circuit in a Cyclone FPGA

The CAD flow used to implement a circuit in a Cyclone FPGA is the same as that used for the MAX 7000 CPLD. We show in Chapter 4 that multilevel logic synthesis is an effective optimization strategy when targeting designs to lookup table-based FPGAs. Figure 4.54 gives Verilog code for a seven-variable logic function used to illustrate the benefits of multilevel synthesis. In this section we will create a new design project, named *example_verilog*2, which represents the Verilog code in that figure.

Create a new project in a directory named *tutorial*2*multilevel*, and use the name *example_verilog*2 for both the project name and the name of the top-level entity. Select the Cyclone family and let the compiler choose a specific device.

Create a Verilog design file called *example_verilog*2 that comprises the code from Figure 4.54, as displayed in Figure C.9a. Compile the project. After successful compilation, in the compilation report expand the Fitter section and click on Fitter Equations. At the bottom of this section in the report, the output f is specified as

$$\mathsf{f} = \mathsf{OUTPUT}(\mathsf{A1L3});$$

As shown in Figure C.9b the logic expression for A1L3 implements f in a multilevel logic form. The first level of logic is specified as

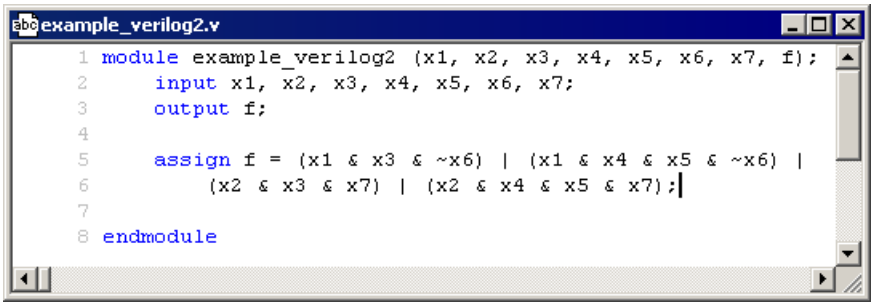
$$A1L2 = x_6x_2x_7 + \overline{x}_6(x_1 + x_2x_7)$$

We show in Appendix E that the logic cell in the Cyclone FPGA is a 4-input lookup table (LUT) that can implement any four-input function. Since the expression above has four inputs, it can be realized in one logic cell in the device. This cell provides an input to the next-level expression

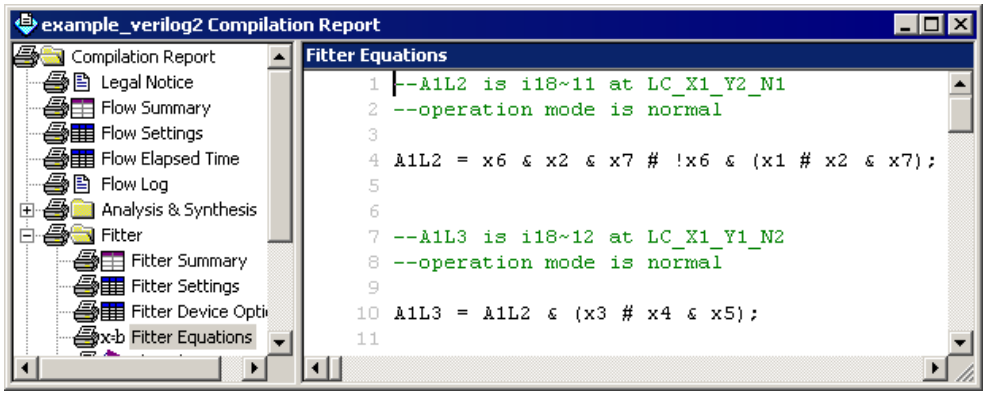
$$A1L3 = A1L2(x_3 + x_4x_5)$$

This expression also has four inputs, and can therefore be realized in a single cell. Thus, f is implemented as two cascaded logic cells. The reader is encouraged to verify that the expression for A1L3 properly implements the function specified in Figure C.9a.

Having implemented the design in the Cyclone device, perform a timing simulation (as explained in section C.1.3) to gain a feeling for the timing characteristics of the Cyclone device. Once a project has been compiled for the target device, it can be downloaded into a chip by using Quartus II. The procedure for programming a chip is described in Appendix D.



(*a*) The Verilog source code.



(*b*) The Fitter Equation report.

Figure C.9. The *example_verilog*2 source code and implementation.

C.3 Implementing an Adder using Quartus II

In section 5.5 we show how an n-bit ripple-carry adder can be specified in Verilog code. In this section we show how the ripple-carry adder can be implemented using the Quartus II system. Create a new project, *adder16*, in a directory *tutorial*2*addern*. We will implement the adder circuit in a Cyclone FPGA. Thus, in the New Project Wizard window shown in Figure B.7, select the Cyclone family. Choose Yes under the question Do you want to assign a specific device, and click the Next button. In the wizard's screen that comes next choose the EP1C6F256C7 (if this device is not listed, change the Speed Grade item in the Filter box to Any).

C.3.1 The Ripple-Carry Adder Code

Verilog code for the n-bit adder is given in Figure C.10. It takes the carry-in signal, *carryin*, plus two n-bit numbers, X and Y, as inputs and produces the n-bit output sum, S, and carry-out signal, *carryout*. The code uses the parameter n, so that the adder can be parameterized to work for any value of n. In this example, n is set to 16. In the code the vector C is used to represent the intermediate carries between the stages in the adder. A for loop is used to create n full-adders that comprise the ripple-carry adder.

Type the code in Figure C.10 into the Text Editor, as explained in Section B.4.2, and save the file in the *tutorial2\addern* directory using the name *adder16.v*. Compile the circuit. The compilation report is shown in Figure C.11.

```
module  adder16 (carryin, X, Y, S, carryout);
    parameter n = 16;
    input  carryin;
    input  [n−1:0] X, Y;
    output  [n−1:0] S;
    output  carryout;
    reg  [n−1:0] S;
    reg  [n:0] C;
    reg  carryout;
    integer  k;

    always @(X or Y or carryin)
    begin
        C[0] = carryin;
        for (k = 0; k <= n−1; k = k+1)
        begin
            S[k] = X[k] ^ Y[k] ^ C[k];
            C[k+1] = (X[k] & Y[k]) | (C[k] & X[k]) | (C[k] & Y[k]);
        end
        carryout = C[n];
    end

endmodule
```

Figure C.10. Verilog code for a ripple-carry adder.

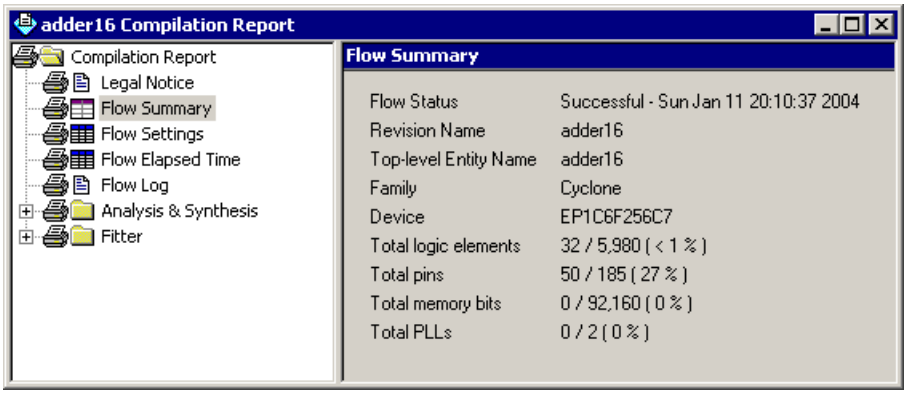


Figure C.11. The compilation report summary.

C.3.2 Simulating the Circuit

To test the correctness of the circuit, we will perform timing simulation. For brevity only a few test vectors will be used, but in a real design situation more extensive testing would be required.

Open the Waveform Editor window. Use Edit | End Time to set the desired simulation to run from 0 to 250 ns. Choose the grid lines to be placed at 25-ns intervals. This is done by selecting Edit | Grid Size, which leads to the window in Figure C.12. Set the period to 50 ns and click OK. Select View | Fit in Window to display the entire simulation range in the window.

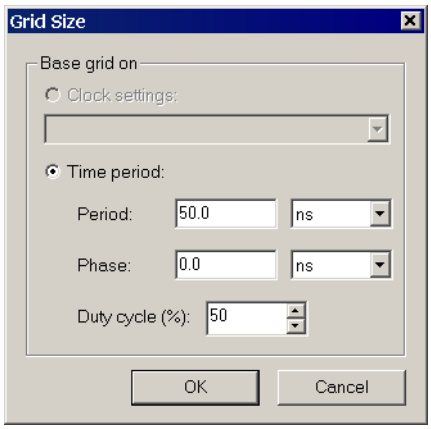


Figure C.12. Setting the spacing of grid lines.

Select Edit | Insert Node or Bus, and then open the Node Finder utility to reach the window in Figure C.13. Set the filter to Pins: all and click List, which displays the input and output nodes as depicted in the figure. Scroll down the list of displayed nodes until you reach *carryin*. Select this node by clicking on it and then clicking the > sign. Next select the X input. Note that this input can be selected either as nodes that correspond to the individual bits (denoted by bracketed subscripts) or as a 16-bit vector, which is a more convenient form. Then, select the input Y and outputs S and *carryout*. This produces the image in the figure. Click OK.

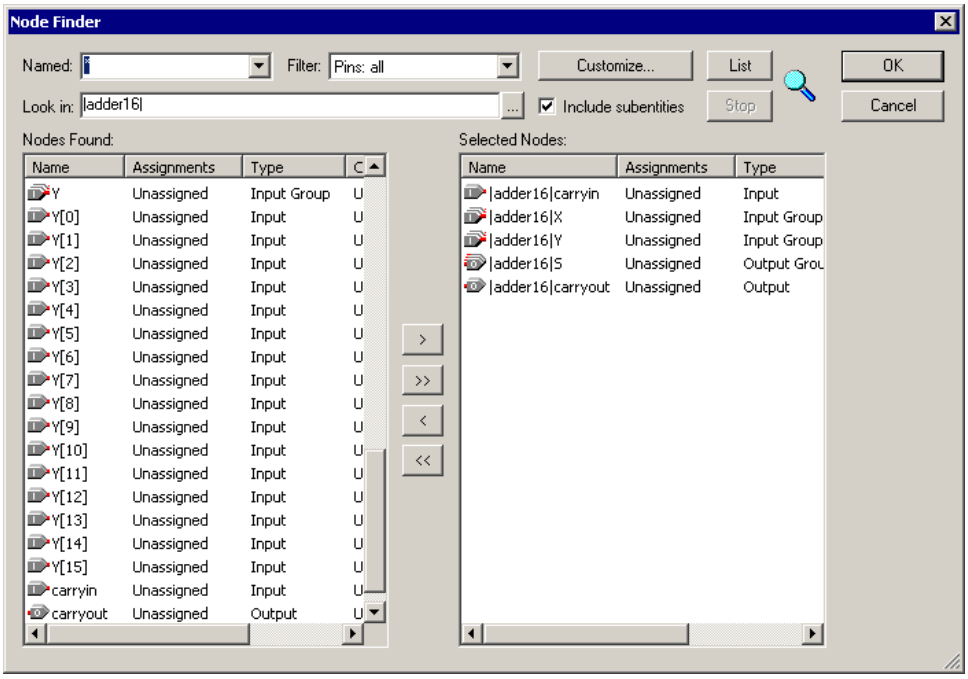


Figure C.13. The Node Finder window.

The Waveform Editor window now looks like the image in Figure C.14. Vectors X, Y, and S are initially treated as binary numbers. They can also be treated as either octal, hexadecimal, signed decimal, or unsigned decimal numbers. For our purpose it is convenient to treat them as hexadecimal numbers, so right-click on X in the Name column and select Properties in the pop-up box to get to the window displayed in Figure C.15. Choose hexadecimal as the radix, make sure that the bus width is 16 bits, and click OK. (Quartus II uses the term *bus* to refer to multibit nodes.) In the same manner, declare that Y and S should be treated as hexadecimal numbers. The resulting waveform display is shown in Figure C.16.

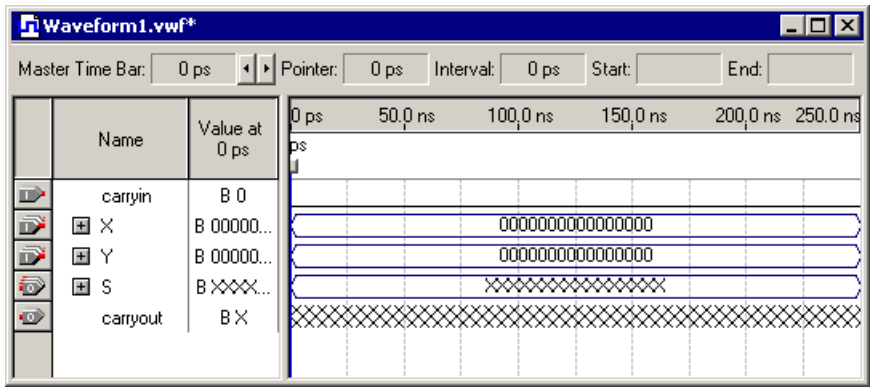


Figure C.14. Selected input and output nodes.

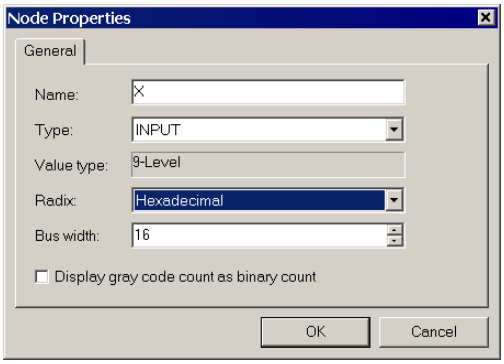


Figure C.15. Defining the characteristics of a node.

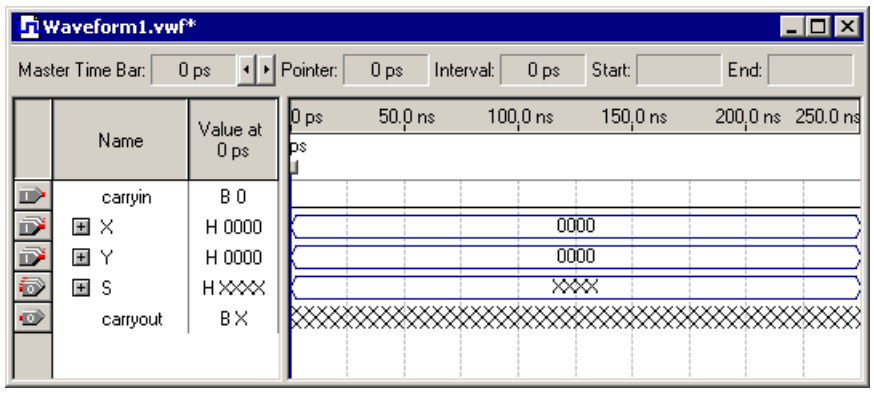


Figure C.16. Using the hexadecimal representation for multibit signals.

We will now set the test values of X and Y. The default value of these inputs is 0. To assign specific values in various intervals proceed as follows. Select (highlight) the interval from 100 to 175 ns of input X. Press the Arbitrary Value icon in the toolbar (it is labeled by a question mark), to bring up the pop-up window in Figure C.17. Enter the value 3FFF and click OK. Then, set X to the value 7FFF in the interval from 175 to 250 ns. Set Y to 0001 in the interval from 50 to 250 ns. Thus, the input waveforms should be as depicted in Figure C.18. Save the file as *adder16.vwf*.

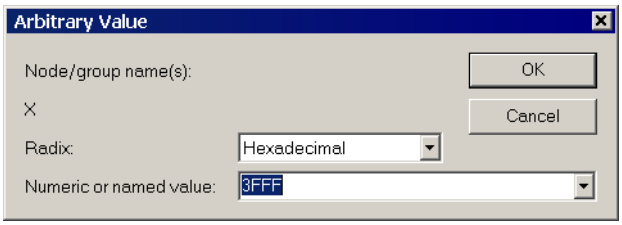


Figure C.17. Assigning the value of a multibit signal.

C.3.3 Timing Simulation

To examine the functionality of the circuit, and determine its speed of operation in the chosen device, we will perform a timing simulation. Select Assignments | Settings | Simulator to reach the window in Figure B.25 and choose Timing as the simulation mode. Run the simulator. The result is given in Figure C.18. It shows considerable delays in producing the correct value $S = 4000$ because the carries are rippling through the adder stages.

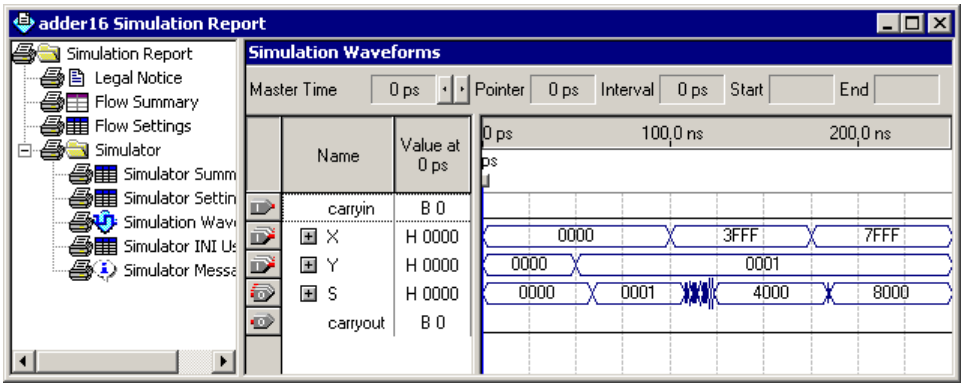


Figure C.18. The result of timing simulation.

Point to the small square handle at the top of the reference line and drag it to the point where the S value becomes 4000. A more accurate view can be obtained if the waveform image is enlarged using the Zoom Tool. Enlarge the image to look like the display in Figure C.19. Click on the Selection Tool icon, and drag the reference line as closely as possible to the point where the value 4000 becomes valid.

The change in S from 0001 to 4000 is caused by the X input changing from 0000 to 3FFF, which occurs at 100 ns. As seen in Figure C.19, the output S changes to 4000 at approximately 123.4 ns. Therefore, the propagation delay through the adder, for these particular values of inputs, is estimated to be 23.4 ns. Note that, in this case, the adder performs the operation $3FFF + 1 = 4000$ which involves a carry rippling through most of the stages of the adder circuit. For other values of inputs, the propagation delay may be much smaller. In Figure C.18, we see that the operation $0000 + 0001 = 0001$ is completed in about 8.5 ns.

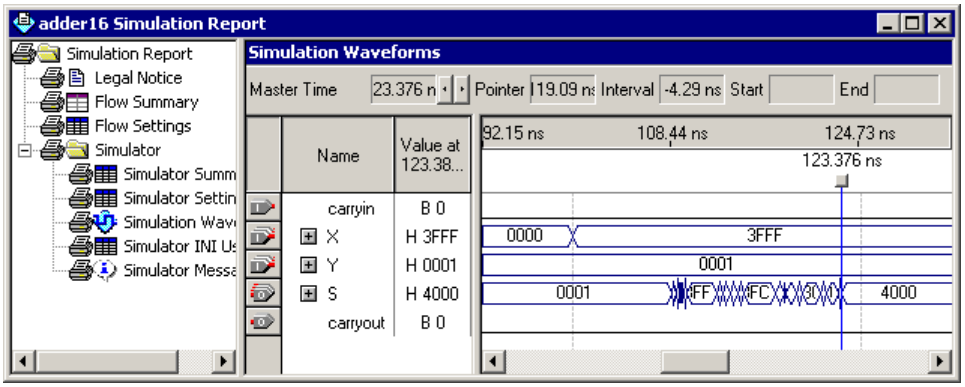


Figure C.19. Detailed results of timing simulation.

When we compile our circuit using Processing | Start Compilation one of the modules executed is the Timing Analyzer. As explained in Chapter 12, this module automatically produces an estimate of the speed of the circuit. Open the compilation report by selecting Processing | Compilation Report or by clicking on its icon. The report includes the derived timing analysis. Click on the small + symbol next to Timing Analyzer to expand this section of the report. Then, click on Timing Analyzer Summary to get the display in Figure C.20. The summary indicates that the estimated worst case propagation delay from an input to output pin, t_{pd}, is 24.7 ns. This longest path starts at the *carryin* input and ends at $S[15]$. Note also that the minimum delay is estimated to be 8.5 ns. More detailed information about the propagation delays along various paths through the circuit can be seen by clicking on tpd on the left side of Figure C.20, which displays the information in Figure C.21. Here, we see that there are several paths along which the propagation delay is close to the maximum, including the one given in the summary in Figure C.20. These longest-delay paths are referred to as *critical paths*.

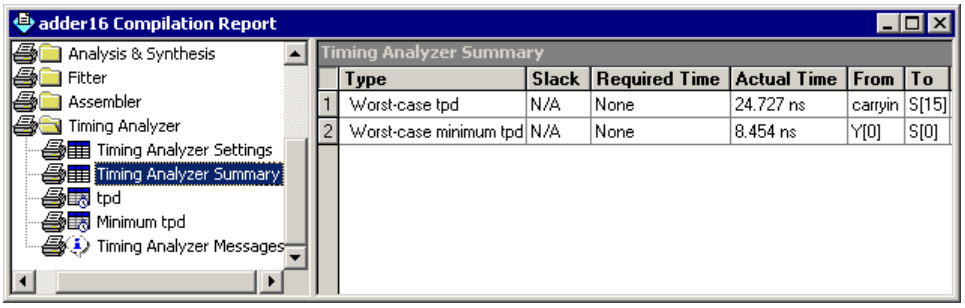


Figure C.20. The worst-case propagation delay.

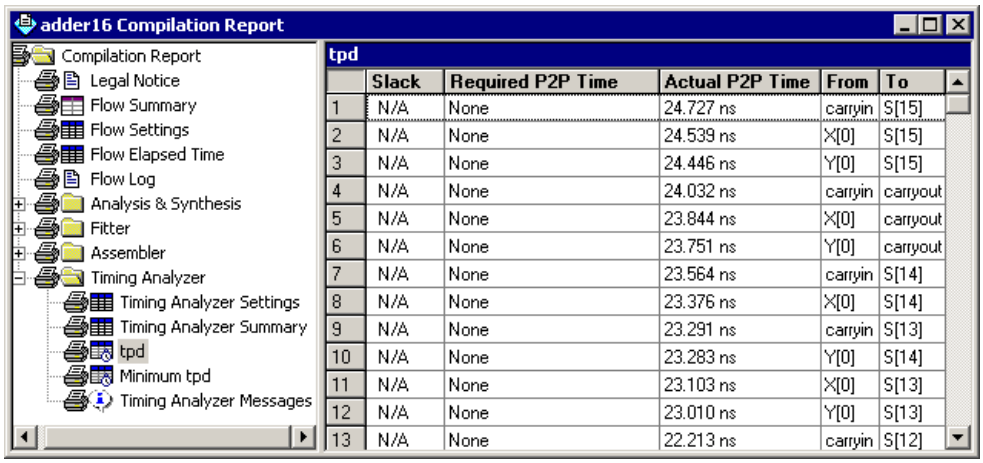


Figure C.21. The critical paths.

The Timing Analyzer performs several types of timing analysis. The results displayed in Figure C.21 give the delays through a combinational circuit, from input pins to output pins. The other types of analysis are applicable only to circuits that contain storage elements, namely flip-flops. This type of analysis is discussed in section C.5.

C.3.4 Implementation in a CPLD Chip

We will now implement the ripple-carry circuit in a CPLD chip. Select Assignments | Device to reach the window in Figure C.22. Choose the MAX 7000S family and select the device EPM7128SLC84-7.

Compile the circuit. Open the Timing Analyzer summary in the compilation report, which is depicted in Figure C.23. Observe that the worst-case propagation delay is now 22.5 ns, which is smaller than the delay observed in Figure C.20. We should not jump to a conclusion about the relative performance of FPGA and CPLD devices, because this circuit is just a small example, and there are many other devices that we could have chosen in our implementation. Also, there are other possibilities in implementing a design, as we will see in the next section.

We have finished working on the *addern* circuit, so close the project.

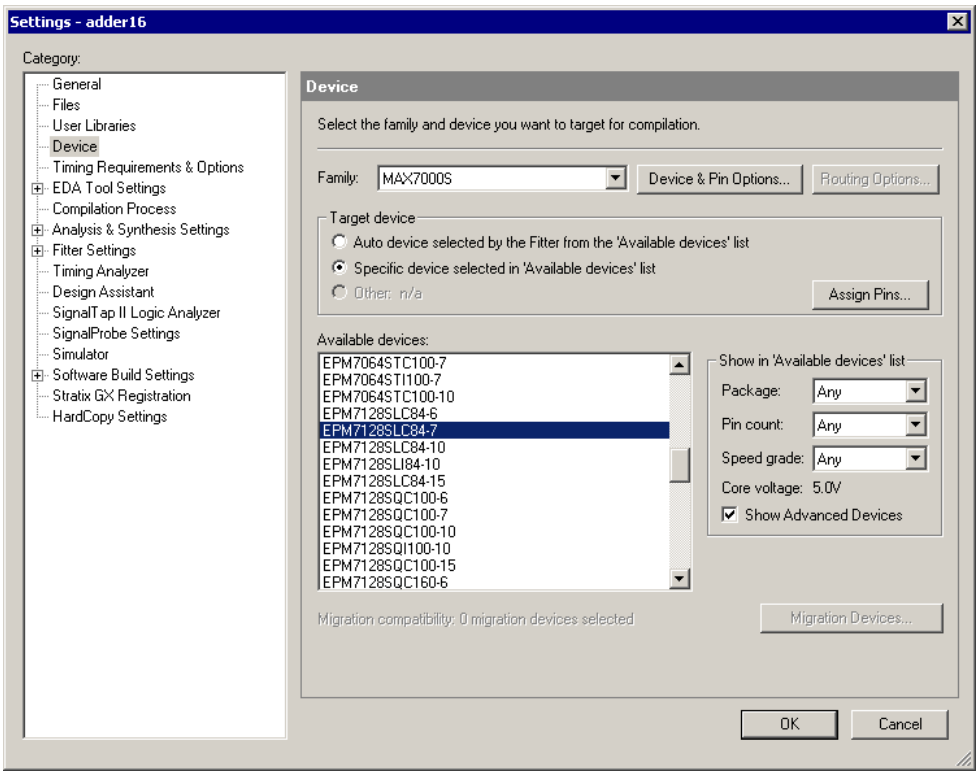


Figure C.22. Specification of the desired device.

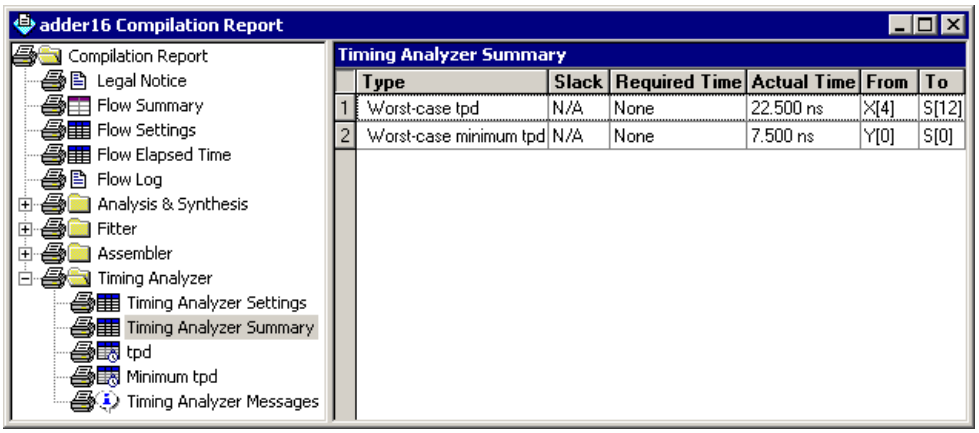


Figure C.23. The worst-case delay using a CPLD

C.4 Using an LPM Module

In section 5.5.1 we discuss how an adder circuit can be implemented by using the *lpm_add_sub* module in the library of parameterized modules (LPM). In this section we compare the adder circuit produced by the *lpm_add_sub* module to the ripple-carry adder implemented in the previous section. Create a new project, *adder16_lpm*, in a directory *tutorial2\adderlpm*. Choose the same FPGA chip as in section C.3.

The easiest way to instantiate an LPM module is by means of a wizard. Select Tools | MegaWizard Plug-in Manager to activate the wizard. A number of pop-up boxes will appear in which we can specify

the features of the desired module. In the screen shown in Figure C.24 choose to create a new variation of a megafunction, and click Next. In the screen in Figure C.25 select the LPM_ADD_SUB module. Make sure that the Cyclone family is indicated at the top right, and also select the entry Verilog HDL as the type of file to create. Let the output file be named *megadd.v*. (The filename extension, *v*, will be added automatically.) Click Next. In Figure C.26, specify that a 16-bit adder circuit is required. Click Next to reach the screen in Figure C.27. Indicate that both inputs can vary and click Next. In Figure C.28 specify that both carry input and output signals are needed. Observe that the wizard displays a symbol for the adder which includes the specified inputs and outputs. In the screen in Figure C.29 decline the pipelining option. The last screen is given in Figure C.30, which indicates the files generated by the wizard. Click Finish. We are interested only in the *megadd.v* file, so make sure that this in the only file selected by a check mark.

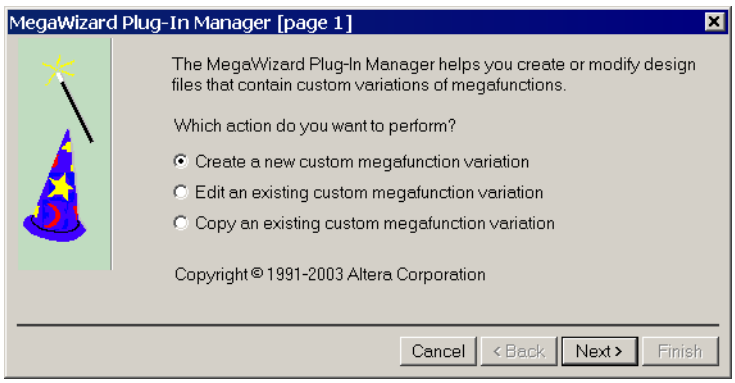


Figure C.24. Choose to create an LPM instance.

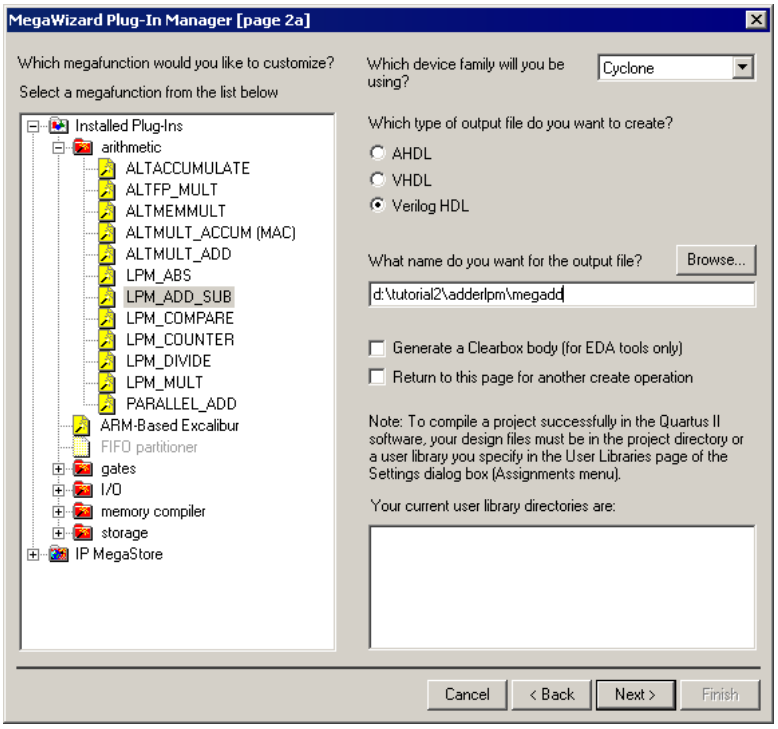


Figure C.25. Select the LPM and its Verilog specification.

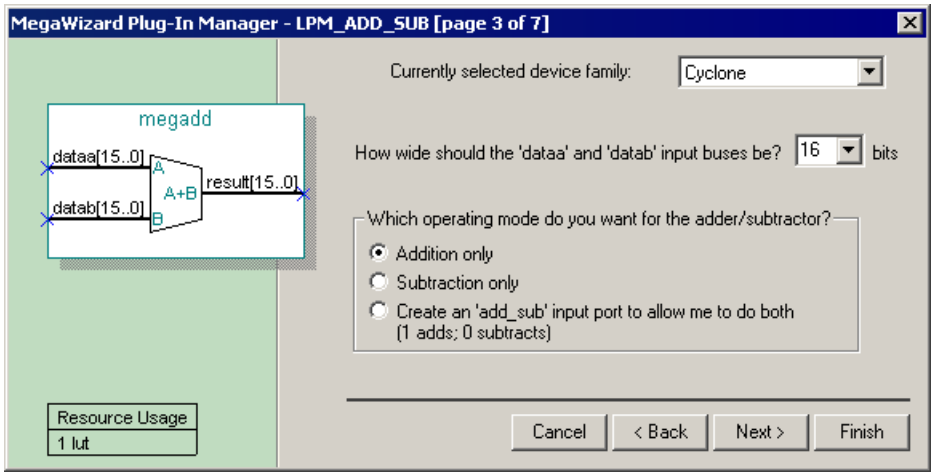


Figure C.26. Choose the adder option and the number of bits.

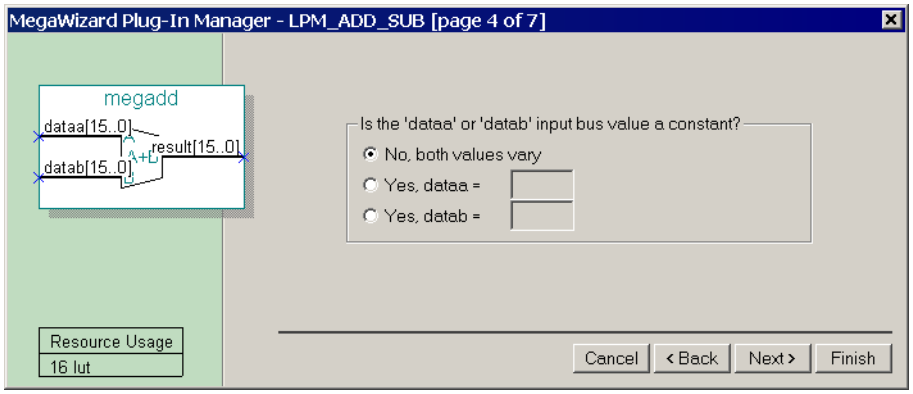


Figure C.27. Choose both inputs to be variable.

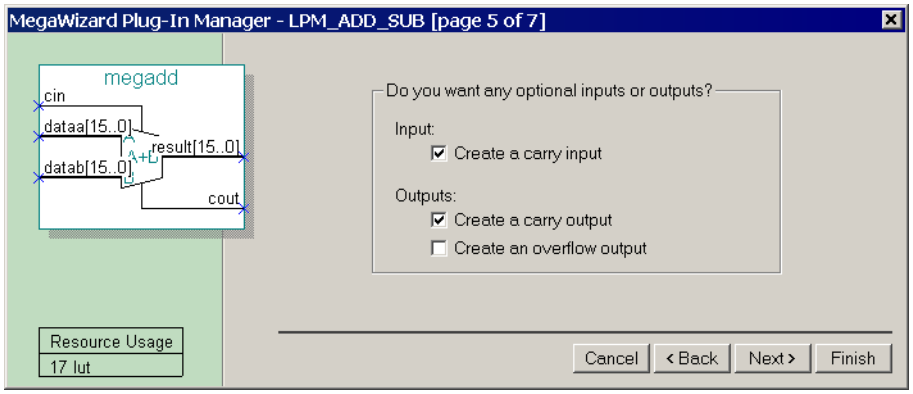


Figure C.28. Include carry input and output connections.

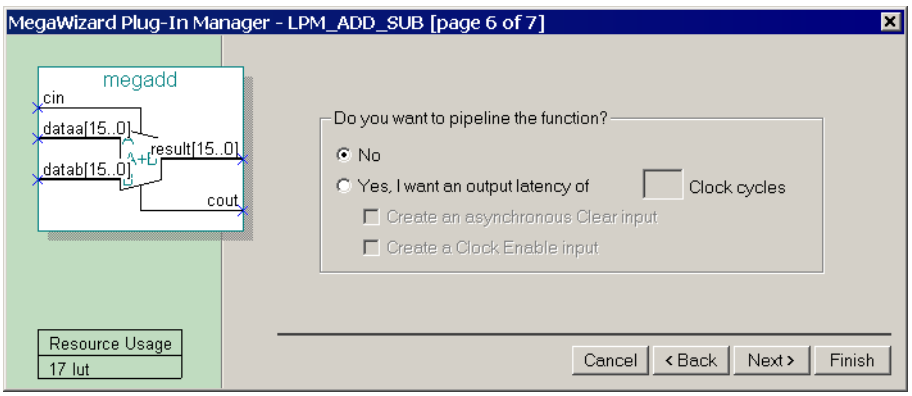


Figure C.29. Decline the pipelining option.

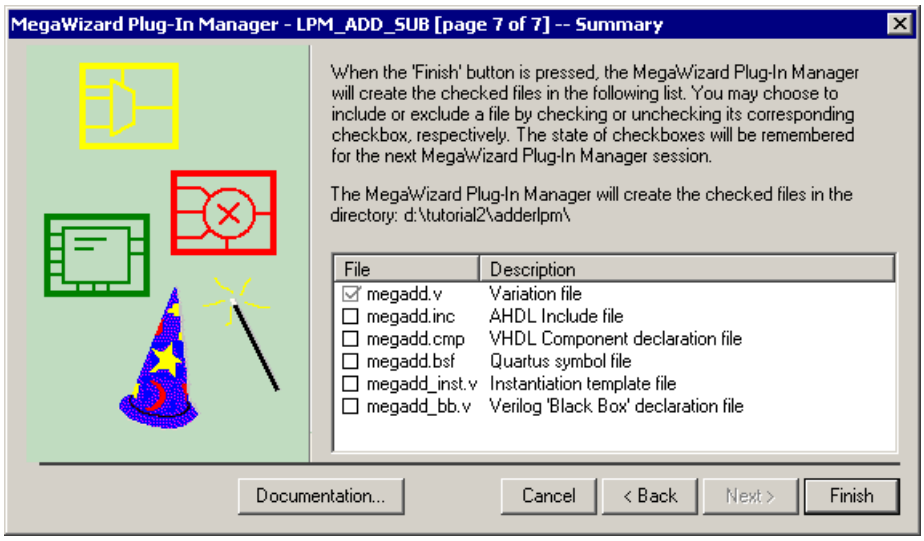


Figure C.30. Files generated by the wizard.

The *megadd* module is shown in Figure C.31. (We have removed the comments to make the figure smaller.) The top-level Verilog code that instantiates this module is given in Figure C.32. Enter this code into a file called *adder16_lpm.v*.

Compile the design. A summary of the timing analysis is shown in Figure C.33. In this design, the worst-case propagation delay is 13.4 ns. Clearly, the adder implementation by means of an appropriate LPM is superior to our generic specification in Figure C.10. The reason that this adder is much faster than our previously created ripple-carry adder is that the LPM makes use of special circuitry in the FPGA for performing addition. We discuss such circuitry, often called a *carry-chain*, in Section 5.4. We may conclude that a designer should normally use an LPM if a suitable module exists in the library. Close the *adder16_lpm* project.

```
module  megadd (dataa, datab, cin, result, cout);
   input  [15:0] dataa;
   input  [15:0] datab;
   input  cin;
   output  [15:0] result;
   output  cout;
   wire  sub_wire0;
   wire  [15:0] sub_wire1;
   wire  cout = sub_wire0;
   wire  [15:0] result = sub_wire1[15:0];

   lpm_add_sub  lpm_add_sub_component (
                .dataa (dataa),
                .datab (datab),
                .cin (cin),
                .cout (sub_wire0),
                .result (sub_wire1));
      defparam
         lpm_add_sub_component.lpm_width = 16,
         lpm_add_sub_component.lpm_direction = ”ADD”,
         lpm_add_sub_component.lpm_type = ”LPM_ADD_SUB”,
         lpm_add_sub_component.lpm_hint = ”ONE_INPUT_IS_CONSTANT=NO”;
endmodule
```

Figure C.31. Verilog code for the *megadd* module.

```
module  adder16_lpm (carryin, X, Y, S, carryout);
   input  carryin;
   input  [15:0] X, Y;
   output  [15:0] S;
   output  carryout;

   megadd  adder_circuit (.cin(carryin), .dataa(X), .datab(Y),
               .result(S), .cout(carryout));
endmodule
```

Figure C.32. Verilog code that instantiates the LPM adder module.

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To
1	Worst-case tpd	N/A	None	13.242 ns	Y[0]	S[15]
2	Worst-case minimum tpd	N/A	None	9.079 ns	X[0]	S[0]

Figure C.33. The worst-case delay for the *adder16_lpm* circuit.

C.5 Design of a Finite State Machine

This example shows how to implement a sequential circuit using Quartus II. The presentation assumes that the reader is familiar with the material in Chapter 8. In section 8.1 we show a simple Moore-type finite state machine (FSM) that has one input, w, and one output, z. Whenever w is 1 for two successive clock cycles, z is set to 1. The state diagram for the FSM is given in Figure 8.3; it is reproduced in Figure C.34. Verilog code that describes the machine appears in Figure 8.29; it is reproduced in Figure C.35. Create a new project, *simple*, in the directory *tutorial2\fsm*. Create a new Text Editor file and enter the code shown in Figure C.35. Save the file with the name *simple.v*.

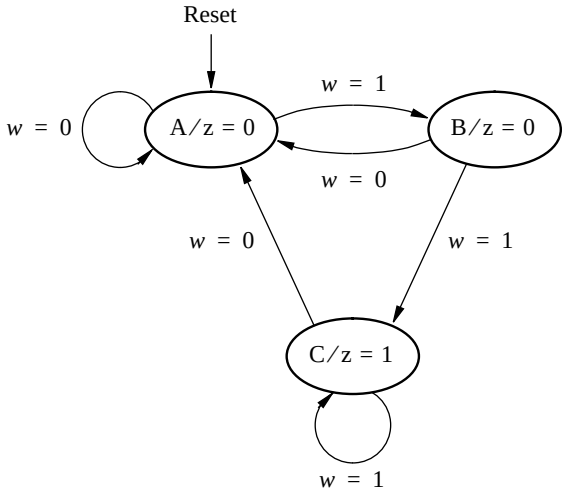


Figure C.34. State diagram of a Moore-type FSM.

```
module  simple (Clock, Resetn, w, z);
    input  Clock, Resetn, w;
    output  z;
    reg  [2:1] y, Y;
    parameter  [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

    // Define the next state combinational circuit
    always @(w or y)
        case (y)
            A:  if (w)    Y = B;
                else      Y = A;
            B:  if (w)    Y = C;
                else      Y = A;
            C:  if (w)    Y = C;
                else      Y = A;
            default:      Y = 2'bxx;
        endcase

    // Define the sequential block
    always @(negedge Resetn or posedge Clock)
        if  (Resetn == 0)   y <= A;
        else   y <= Y;

    // Define output
    assign  z = (y == C);

endmodule
```

Figure C.35. Verilog code for the FSM in Figure C.34.

C.5.1 Implementation in a CPLD

Select the same MAX 7000S device as in section C.1. Compile the circuit. Open the Waveform Editor and import the nodes *Resetn*, *Clock*, *w*, and *z*. These nodes are found by setting the Node Finder filter to Pins: all. We also want to see the behavior of the state variables, which are implemented by means of flip-flops. To find these nodes, set the Node Finder filter to Registers: post-fitting and click List. The Node Finder displays two nodes, as shown in Figure C.36. Import both of these nodes into the Waveform Editor. Set the total simulation time to 650 ns and set the grid size to 25 ns. Set *Resetn* $= 0$ during the first 50 ns, and then set *Resetn* $= 1$. To enter the waveform for the clock signal, click on the name of the *Clock* waveform in the Waveform Editor display. With the signal highlighted, click on the *Overwrite Clock* icon in the toolbar (the icon depicts a clock). This causes the pop-up window in Figure C.37 to appear. Set the clock period to be 50 ns, make sure that the phase is 0 and the duty cycle is 50 percent, and click OK. The defined clock signal is now displayed in the Waveform Editor window, as depicted in Figure C.38. Next, draw the waveform for w as indicated in the figure. Save the file, under the name *simple.vwf*. Run the Timing Simulator to get the result shown in Figure C.39.

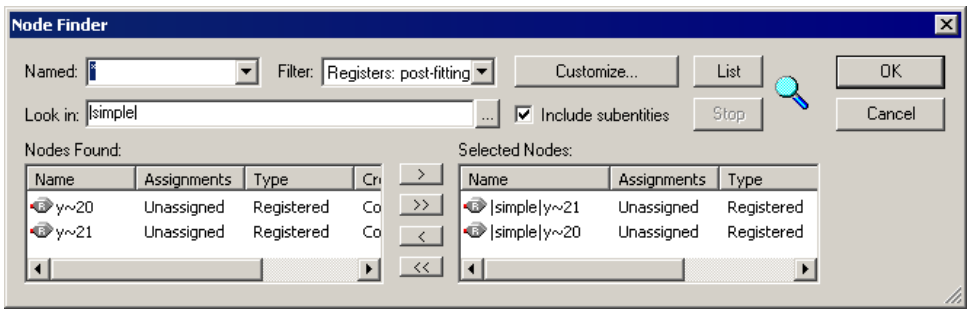


Figure C.36. Nodes that represent the state variables.

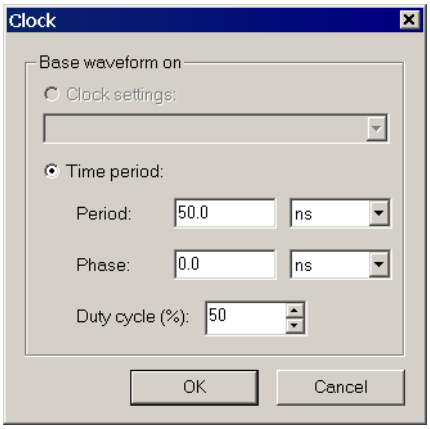


Figure C.37. Setting the *Clock* input.

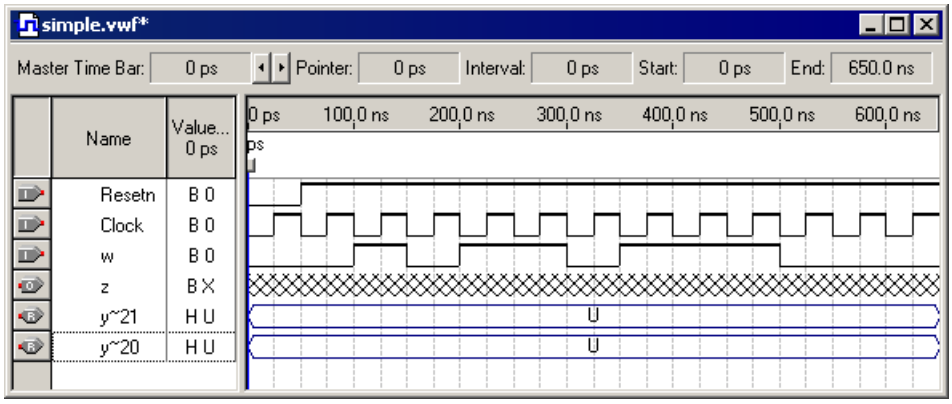


Figure C.38. Input test vectors.

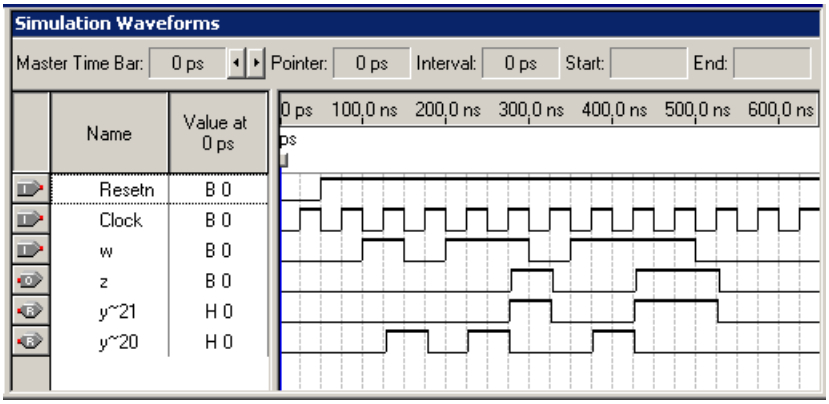
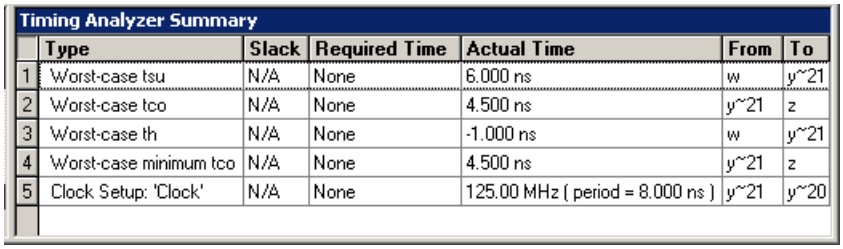


Figure C.39. Timing simulation waveforms.

The FSM behaves correctly, setting $z = 1$ in each clock cycle for which $w = 1$ in the preceding two clock cycles. Examine the timing delays in the circuit, using the reference line in the Waveform Editor. Observe that changes in the FSM's state occur 2.5 ns after an active clock edge and that 4.5 ns are needed to change the value of the z output.

Open the Timing Analyzer summary in the compilation report, which is displayed in Figure C.40. The bottom row indicates that the maximum frequency, which is often called *fmax*, at which the synthesized circuit can operate is 125 MHz. This is a useful indicator of performance. The *fmax* is determined by the longest propagation delay between two registers (flip-flops). The figure also shows the values of some other timing parameters. The worst-case flip-flop setup time, *tsu*, and hold time, *th*, are given. Line 1 in figure C.40 specifies that the w input cannot change within 6 ns of the active clock edge, or else the $y\sim21$ flip-flop may become unstable. Line 3 shows that no input signal in our circuit has to remain stable after the active clock edge, because the worst case hold-time requirement is negative. We explain in section 10.3.2 how flip-flop timing parameters are determined in a target chip. The parameter *tco* indicates the time elapsed from an active edge of the clock signal at the clock pin until an output signal is produced at an output pin. This delay is 4.5 ns for the z output, which is what we also observed in the waveforms in Figure C.39.



Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To
1	Worst-case tsu	N/A	None	6.000 ns	w	y~21
2	Worst-case tco	N/A	None	4.500 ns	y~21	z
3	Worst-case th	N/A	None	-1.000 ns	w	y~21
4	Worst-case minimum tco	N/A	None	4.500 ns	y~21	z
5	Clock Setup: 'Clock'	N/A	None	125.00 MHz (period = 8.000 ns)	y~21	y~20

Figure C.40. Summary of the timing analysis for the FSM circuit.

Note that the states of this FSM are implemented using two state variables. The Verilog code in Figure C.35 specified the present state variables as $y[1]$ and $y[2]$. However, Quartus II gave the names $y\sim20$ and $y\sim21$ to these variables, as we discovered when using the Node Finder. Quartus II uses the names of all inputs and outputs as given in the Verilog code, but it may choose fairly arbitrary names for internal connections.

Two or more binary signals displayed in the Waveform Editor can be combined into a "group" (corresponding to a vector in Verilog terminology) of signals that can be referred to by a single name. Open the *simple.vwf* file and select the $y\sim21$ and $y\sim20$ simultaneously, so that their waveforms are highlighted (make

sure that $y{\sim}21$ is listed *above* $y{\sim}20$, as shown in Figure C.38). Select Edit | Group to reach the pop-up box in Figure C.41. Type y as the group name, choose hexadecimal as the radix, and click OK. This causes y to be used, instead of $y{\sim}21$ and $y{\sim}20$, in the file *simple.vwf*. Perform timing simulation to get the result in Figure C.42. Now, the FSM states are represented by the values of the vector y.

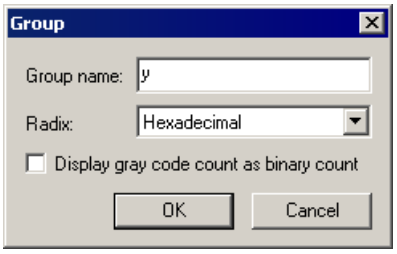


Figure C.41. Grouping of signals.

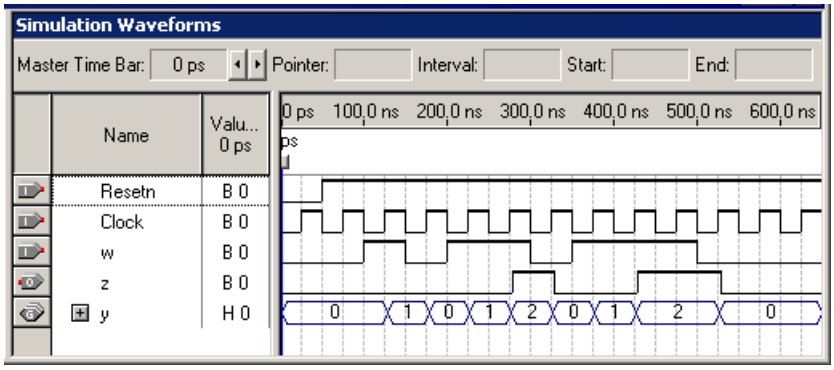


Figure C.42. Waveform displayed as a vector y.

C.5.2 Implementation in an FPGA

In section 8.8 we said that when implementing an FSM in an FPGA, a good strategy is to use one-hot encoding, with one state variable assigned to each state. The Quartus II synthesis tool automatically chooses this approach when targeting an FPGA chip.

The reader is encouraged to recompile the *simple.v* code for the same FPGA chip used in section C.3. Compile the code and observe that three flip-flops are used to implement the FSM. The timing analysis results should show that the circuit will operate at an fmax of about 320 MHz.

C.6 Concluding Remarks

Having completed this and the preceding tutorial, the reader is familiar with many of the most important features of Quartus II. In the next tutorial we will show how the user can manipulate which pins on the target chip are used for a circuit, and how PLD programming is done with Quartus II.