Appendix D

Tutorial 3

Using MAX+plusII with the 2^{nd} Edition of

Fundamentals of Digital Logic with VHDL Design

This tutorial introduces more advanced capabilities of the MAX+plusII system. We show how hierarchical VHDL code is organized and compiled and illustrate how multibit signals in arithmetic circuits are represented using the CAD tools. Examples using the building blocks in the library of parameterized modules (LPM) are presented, as well as examples of sequential circuits. In addition to the CAD tool applications that are used in Tutorials 1 and 2, this tutorial introduces the Timing Analyzer application.

D.1 Design Using Hierarchical VHDL Code

In section 5.2 we show how an n-bit ripple-carry adder is constructed using n instances of the full-adder circuit. In this section we show how the ripple-carry adder can be described with hierarchical VHDL code.

D.1.1 The Full-Adder Subcircuit

For storing the files used in this tutorial, we created the directory d:\max2work\tutorial3. To enter the full-adder code, in the Manager window select File | New and create a new Text Editor file. Code for the full-adder is given in Figure 5.22. Type this code, which is reproduced in Figure D.1, into the Text Editor. Save the file in the *tutorial3* directory with the name *fulladd.vhd*.

D.1.2 The Ripple-Carry Adder Code

Next create another new Text Editor file to hold the VHDL code for the ripple-carry adder. An example of code for a four-bit adder is shown in Figure 5.23. The code in Figure D.2 gives an n-bit version of the ripple-carry adder code, which uses n instances of the full-adder subcircuit. The code takes the carry-in

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
    PORT ( Cin, x, y  : IN    STD_LOGIC ;
           s, Cout    : OUT   STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
    s <= x XOR y XOR Cin ;
    Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y) ;
END LogicFunc ;
```

Figure D.1: VHDL code for the full-adder.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY addern IS
    GENERIC ( n : INTEGER := 16 ) ;
    PORT ( Cin     : IN    STD_LOGIC ;
           A, B    : IN    STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           Sum     : OUT   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           Cout    : OUT   STD_LOGIC ) ;
END addern ;

ARCHITECTURE Structure OF addern IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO n−1) ;
    COMPONENT fulladd
        PORT ( Cin, x, y   : IN    STD_LOGIC ;
               s, Cout     : OUT   STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    FA_0: fulladd PORT MAP ( Cin, A(0), B(0), Sum(0), C(1) ) ;
    G_1: FOR i IN 1 TO n−2 GENERATE
        FA_i: fulladd PORT MAP ( C(i), A(i), B(i), Sum(i), C(i+1) ) ;
    END GENERATE ;
    FA_n: fulladd PORT MAP ( C(n−1), A(n−1), B(n−1), Sum(n−1), Cout ) ;
END Structure ;
```

Figure D.2: VHDL code for a 16-bit ripple-carry adder.

Cin plus two n-bit numbers A and B as inputs and produces the n-bit output *Sum* and carry-out *Cout*. The code uses the GENERATE statement that is introduced in section 6.6.4. It allows the adder to be parameterized to work for any value of n. In this example, n is set to the value 16.

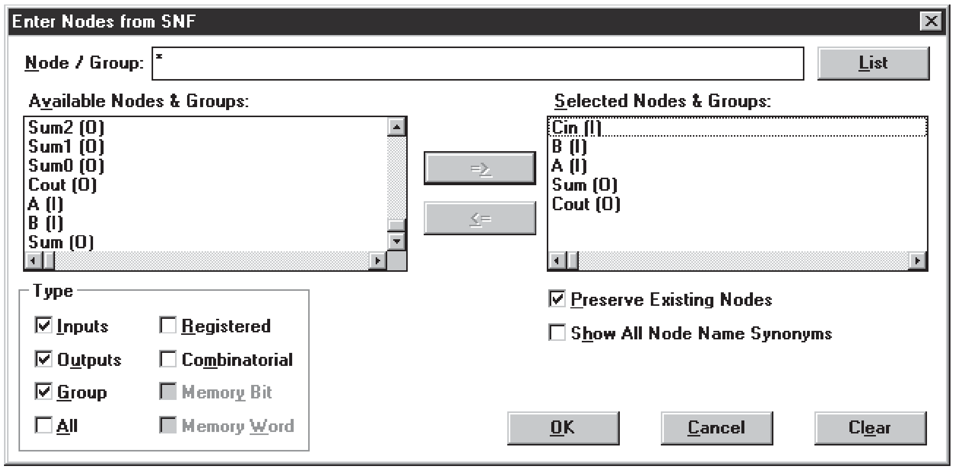
In the architecture the signal C is declared, which is used to represent the intermediate carry signals between the stages in the adder. A component declaration statement is included for the *fulladd* subcircuit defined in Figure D.1. The statement labeled *FA_0* instantiates the least-significant stage of the adder, the generate statement instantiates the next 14 stages, and the statement labeled *FA_n* instantiates the most-significant stage.

Type the code in Figure D.2 into the Text Editor and save the file using the name *addern.vhd*. Use the Ctrl+Shift+j shortcut command to set *addern* as the name of the current project. We will synthesize a circuit to implement this project in a MAX 7000 CPLD. Select Assign | Device to open the Device dialog box. In the drop-down menu labeled Device Family, select MAX7000S. For this project we selected the EPM7128SLC84-7 chip because this chip is provided on the Altera development board, which is discussed in section C.3. If the EPM7128SLC84-7 chip is not listed in the Device dialog box, click on the option at the bottom of the dialog box that specifies Show only fastest speed grades. Turning this option off by clicking on it results in all speed grades of the chips available to the designer being shown in the Devices box (the copy of MAX+plusII included with the book allows the user to select only a limited subset of the chips that are available from Altera).

Click OK to return to the Text Editor and then open the Compiler module. Since we will use timing simulation for this project, make sure that Timing SNF Extractor is turned on in the Processing menu. Run the Compiler. When analyzing the code, the Compiler automatically searches for code to define the *fulladd* component in the file *fulladd.vhd*. If the Compiler reports errors in your code, fix them.

We will perform a timing simulation to determine the speed of operation of the ripple-carry adder in the chosen device. Open the Waveform Editor and then select Node | Enter Nodes from SNF to open the window shown in Figure D.3. Click on the List button and scroll down in the Available Nodes & Groups box until the node *Cin* is visible. Click on the button marked => to copy *Cin* into the Selected Nodes & Groups box. Scroll further down in the Available Nodes & Groups box and select *Cout*. Finally, select the nodes *A*, *B*, and *Sum*, which are displayed at the bottom of the list of Available Nodes & Groups. These nodes represent the 16-bit signals in the VHDL code in Figure D.2. MAX+plusII uses the term *Group*, or *Bus*, for multibit nodes. Click OK to return to the Waveform Editor.

Select File | End Time and set the total simulation time to 250 ns. Select Options | Grid Size to set the guideline spacing to be 25 ns. Save the file with the name *addern.scf*. Use the Ctrl+w shortcut command so that the entire time range, from 0 to 250 ns, is displayed in the Waveform Editor window, as


Enter Nodes from SNF

Node / Group: *

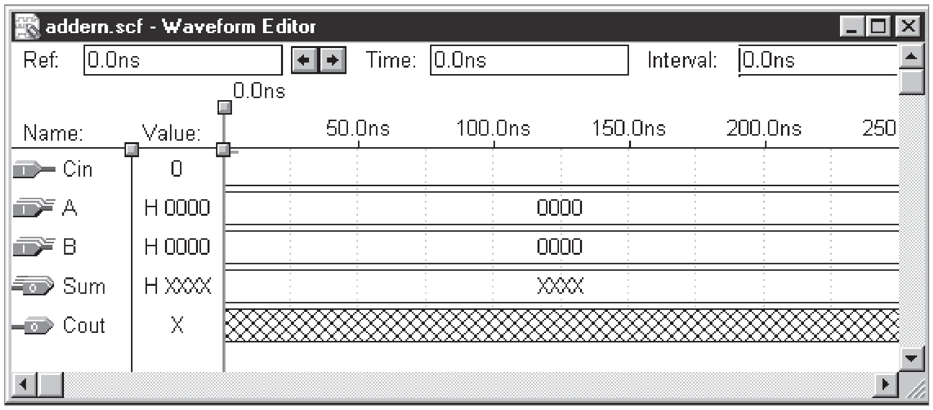
Available Nodes & Groups:
Sum2 (O)
Sum1 (O)
Sum0 (O)
Cout (O)
A (I)
B (I)
Sum (O)

Selected Nodes & Groups:
Cin (I)
B (I)
A (I)
Sum (O)
Cout (O)

Type
[✓] Inputs
[✓] Outputs
[✓] Group
[ ] All
[ ] Registered
[ ] Combinatorial
Memory Bit
Memory Word

[✓] Preserve Existing Nodes
[ ] Show All Node Name Synonyms

Figure D.3: Importing single-bit and multibit node names.



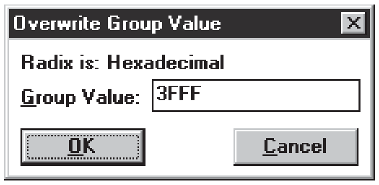
Name:	Value:	
Cin	0	
A	H 0000	0000
B	H 0000	0000
Sum	H XXXX	XXXX
Cout	X	

Figure D.4: The waveform display for the 16-bit adder.

depicted in Figure D.4.

Below the label Value the Waveform Editor displays the value of the signal waveforms at the point where the reference line is currently situated (the value X for an output signal means that the signal value is unknown, because the simulation has not yet been performed). For multibit signals the displayed signal values have an associated number base, which is indicated by a letter. In Figure D.4 the number base used for the multibit signals *A*, *B*, and *Sum* is hexadecimal, denoted by the letter H. It is possible to change the number base (to binary, octal, or decimal) by double-clicking on the H, but we do not need to do so here. If not already done, activate the Waveform Editing tool by selecting its icon on the left side of the Manager window (the icon is labeled <–|–>). Click the mouse at the 100 ns point on the *A* waveform, drag to 175 ns, and then release the mouse. The Overwrite Group Value dialog box shown in Figure D.5 appears. Type 3FFF into this box and click OK. Next set the value of *A* to 7FFF in the time range from 175 ns to 250 ns and set *B* to the value 0001 in the time range from 50 ns to 250 ns. Save the *addern.scf* file.

Open the Simulator module and click Start to run the timing simulation. The results of the simulation are shown in Figure D.6. Move the reference line in the Waveform Editor to the point in the figure where *Sum* takes the value 4000. Since this sum is produced at 141.5 ns, and *A* changes to 3FFF at 100



Radix is: Hexadecimal
Group Value: 3FFF

Figure D.5: Specifying the value of a multibit signal.

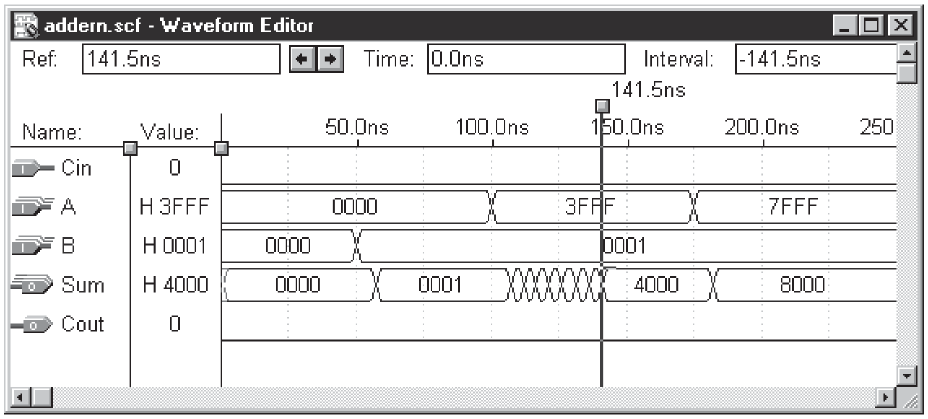


Figure D.6: Timing simulation results for the ripple-carry adder.

ns, the adder requires 41.5 ns to generate the sum.

D.1.3 Alternative Style of Code for the Ripple-Carry Adder

In Figure D.2 the component declaration statement for the *fulladd* subcircuit is included in the architecture. A different style of writing the code is to place the component declaration statement in a package, rather than in the architecture. The package can be included in the same source code file that defines the ripple-carry adder entity. In this example we use a separate file to define the package. Create a new Text Editor file and enter the VHDL code shown in Figure D.7, which defines the package named *fulladd_package*. Save the file with the name *fulladd_package.vhd* and set this as the name of the current project. Open the Compiler module and press the Start button to analyze the code. A message is generated as a reminder that the file being compiled does not contain an architecture.

Open the *addern.vhd* file. As indicated in Figure D.8, add the USE clause

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE fulladd_package IS
    COMPONENT fulladd
        PORT ( Cin, x, y    : IN STD_LOGIC ;
                    s, Cout : OUT  STD_LOGIC ) ;
    END COMPONENT ;
END fulladd_package ;
```

Figure D.7: Declaring the full-adder component in a package.

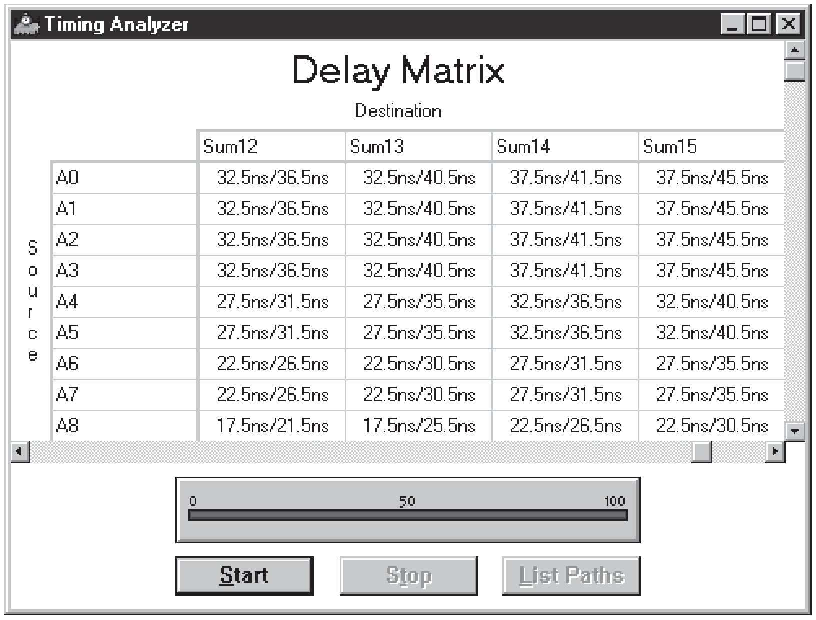
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY addern IS
    GENERIC ( n : INTEGER := 16 ) ;
    PORT ( Cin   : IN   STD_LOGIC ;
           A, B  : IN   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           Sum   : OUT  STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END addern ;

ARCHITECTURE Structure OF addern IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO n−1) ;
BEGIN
    FA_0: fulladd PORT MAP ( Cin, A(0), B(0), Sum(0), C(1) ) ;
    G_1: FOR i IN 1 TO n−2 GENERATE
        FA_i: fulladd PORT MAP ( C(i), A(i), B(i), Sum(i), C(i+1) ) ;
    END GENERATE ;
    FA_n: fulladd PORT MAP ( C(n−1), A(n−1), B(n−1), Sum(n−1), Cout ) ;
END Structure ;
```

Figure D.8: Alternative style of code for the ripple-carry adder.

that includes the *fulladd_ package* package. Delete the *fulladd* component declaration statement from the architecture. Save the file and set it as the current project. Compile the code to verify that there are no errors. The synthesized circuit should be identical to that produced for the code in Figure D.2.

D.1.4 Using the Timing Analyzer Module

The only MAX+plusII module that we have not yet used is the Timing Analyzer. It shows detailed timing information for the circuit synthesized by the Compiler. Select MAX+plusII | Timing Analyzer. Three types of analyses are available, which are listed in the Analysis menu. The type currently selected should be the Delay Matrix. It reports the propagation delays in the circuit from each primary input to each primary output. The other types of analysis are applicable only to circuits that contain storage elements. In the Timing Analyzer window, click the Start button. Figure D.9 shows part of the results produced. In the figure we have sized the window so that four columns of data are visible, and the data has been scrolled to the right to show the propagation delays for nodes *Sum*12 to *Sum*15. The values in each square of the matrix indicate the minimum and maximum delays in the circuit through all paths from each input node to each output node. For instance, the minimum delay from node *A*0 to node *Sum*14



	Sum12	Sum13	Sum14	Sum15
A0	32.5ns/36.5ns	32.5ns/40.5ns	37.5ns/41.5ns	37.5ns/45.5ns
A1	32.5ns/36.5ns	32.5ns/40.5ns	37.5ns/41.5ns	37.5ns/45.5ns
A2	32.5ns/36.5ns	32.5ns/40.5ns	37.5ns/41.5ns	37.5ns/45.5ns
A3	32.5ns/36.5ns	32.5ns/40.5ns	37.5ns/41.5ns	37.5ns/45.5ns
A4	27.5ns/31.5ns	27.5ns/35.5ns	32.5ns/36.5ns	32.5ns/40.5ns
A5	27.5ns/31.5ns	27.5ns/35.5ns	32.5ns/36.5ns	32.5ns/40.5ns
A6	22.5ns/26.5ns	22.5ns/30.5ns	27.5ns/31.5ns	27.5ns/35.5ns
A7	22.5ns/26.5ns	22.5ns/30.5ns	27.5ns/31.5ns	27.5ns/35.5ns
A8	17.5ns/21.5ns	17.5ns/25.5ns	22.5ns/26.5ns	22.5ns/30.5ns

Figure D.9: Using the delay matrix in the Timing Analyser.

is 37.5 ns, and the maximum delay is 41.5 ns. Observe that the propagation delays increase for more significant stages of the adder, as we expect in the ripple-carry structure. The longest delay from any input to any output in the circuit is 45.5 ns. The Timing Analyzer can be set to report only the longest delay by selecting Options | Time Restrictions. Other features of the Timing Analyzer will be discussed in section D.3.3.

We have finished working on the *addern* project, so close all open windows to return to the Manager window.

D.2 Using an LPM Module

In section 5.5.1 we show how to create an adder circuit 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 section D.1.

Create a new Text Editor file and enter the code shown in Figure D.10. It instantiates a 16-bit version of the *lpm_add_sub* module with the same input and output signals used in Figure D.2. Save the file with the name *adderlpm.vhd* and set this file as the current project. Using the Assign menu, select the same device used in the previous example. Run the Compiler to synthesize a circuit that implements the adder.

Open the Waveform Editor and create the same simulation vectors that we

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
LIBRARY lpm ;
USE lpm.lpm_components.all ;

ENTITY adderlpm IS
    PORT ( Cin    : IN    STD_LOGIC ;
           A, B   : IN    STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           Sum    : OUT   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           Cout   : OUT   STD_LOGIC ) ;
END adderlpm ;

ARCHITECTURE Structure OF adderlpm IS
BEGIN
    instance: lpm_add_sub
        GENERIC MAP (LPM_WIDTH => 16)
        PORT MAP ( cin => Cin, dataa => A, datab => B,
                   result => Sum, cout => Cout ) ;
END Structure ;
```

Figure D.10: A 16-bit adder built using the *lpm_add_sub* module.

used in section D.1.2, shown in Figure D.6. Save the waveform file, open the Simulator, and run the timing simulation. The results of the simulation should be as illustrated in Figure D.11. Position the reference line so that it shows the delay incurred by the adder to produce the sum 4000. This sum is produced 22.5 ns after A changes to 3FFF. Comparing this delay to the one in Figure D.6, which is 41.5 ns, we see that the *lpm_add_sub* module generates a faster adder circuit.

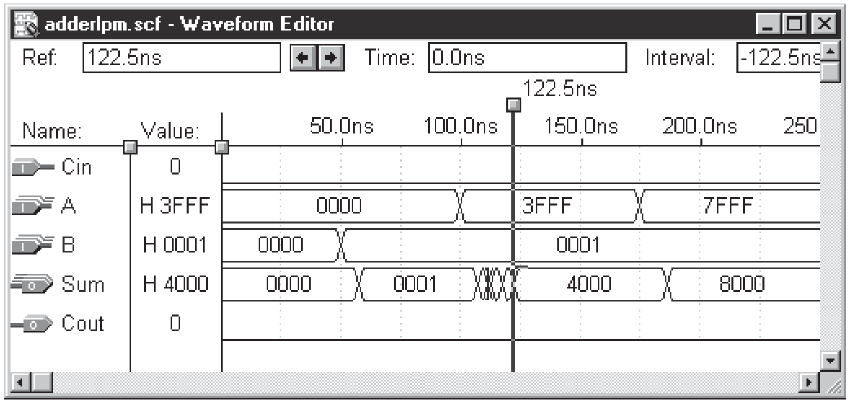
To view the source code file that defines the *lpm_add_sub* module, open the Hierarchy Display module. A simplified picture of the window that appears is given in Figure D.12. It shows that the *lpm_add_sub* module is instantiated using a building block named *addcore*. Double-click on the small icon labeled tdf, which stands for *text design file*, next to addcore:adder. The source code file named *addcore.tdf* is opened in the Text Editor. The code is written using the Altera Hardware Description Language (AHDL), which is another language supported by MAX+plusII. Although AHDL uses a different syntax than VHDL uses, the two languages have enough similarity to enable the reader to understand some of the code. The file includes comments that specify how the *addcore* module is implemented in different types of devices. For devices designated as MAX, such as MAX 7000, *addcore* is implemented as blocks of eight-bit carry-lookahead adders, with ripple-carry between the adder blocks. Hence our 16-bit adder in Figure D.10 is implemented using two 8-bit carry-lookahead adders, with the carry-out of one adder connected to the carry-in of the other. A diagram of this type of adder circuit is shown in Figure 5.17.

Implementation in a FLEX 10K Chip

Close *addcore.tdf* and close the Hierarchy Display. We will now implement the code in Figure D.10 in a FLEX 10K device. Select Assign | Device and choose FLEX10K in the Device Family drop-down menu. For the results shown here, we selected the EPF10K20RC240-4 chip, which is included on the Altera development board described in section C.3.

Open the Compiler and synthesize a circuit that implements the project in the FLEX 10K chip. Run the Timing Simulator to generate the simulation results shown in Figure D.13. Positioning the reference line in the Waveform Editor at the point where *Sum* changes to 4000 shows that 54.9 ns are needed to generate the sum in the FLEX 10K device.

In Appendix E we show that FLEX 10K devices include special-purpose carry logic for implementation of fast adders. The *lpm_add_sub* module can be implemented using this resource by directing the logic synthesis algorithms to optimize the generated circuit for speed. Select Assign | Global Project Logic



Name	Value	
Cin	0	
A	H 3FFF	0000, 3FFF, 7FFF
B	H 0001	0000, 0001
Sum	H 4000	0000, 0001, 4000, 8000
Cout	0	

Figure D.11: Timing simulation results for the code in Figure D.10.

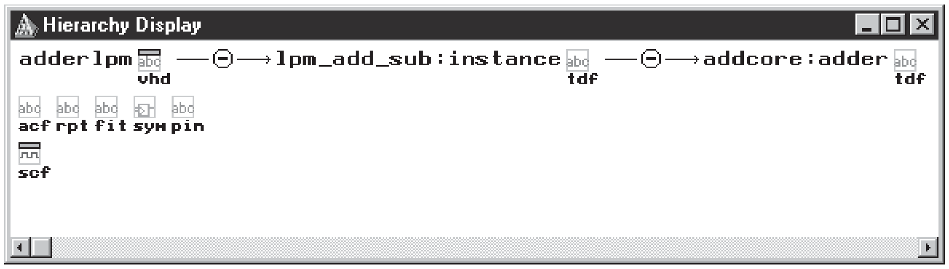


Figure D.12: The Hierarchy Display for the code in Figure D.10.

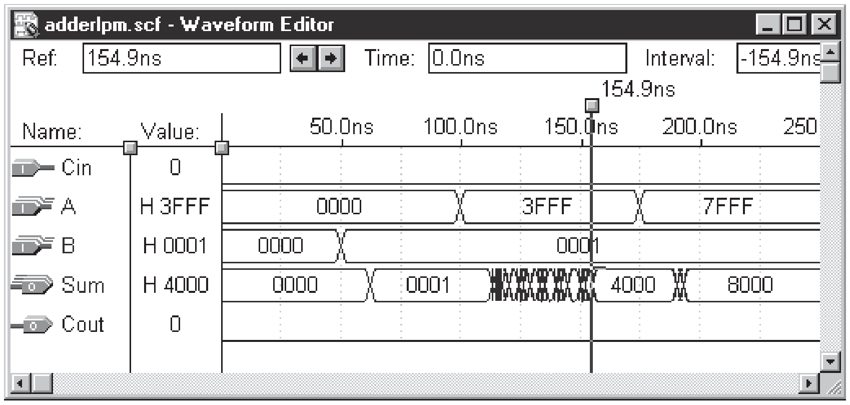

adderlpm.scf - Waveform Editor
Ref: 154.9ns
Time: 0.0ns
Interval: -154.9ns
154.9ns
Name: Value: 50.0ns 100.0ns 150.0ns 200.0ns 250
Cin 0
A H 3FFF 0000 3FFF 7FFF
B H 0001 0000 0001
Sum H 4000 0000 0001 4000 8000
Cout 0

Figure D.13: Timing results when optimized for area in a FLEX 10K device.

Synthesis to open the window shown in Figure D.14. In the Global Project Synthesis Style drop-down menu, select FAST. Run the Compiler to synthesize a circuit optimized for speed and then run the Timing Simulator again. The results, illustrated in Figure D.15, show that only 22.4 ns are needed to generate $Sum = 4000$ when the dedicated carry-logic resources are used.

We have finished working on the *adderlpm* project. Close all open windows to return to the Manager window.

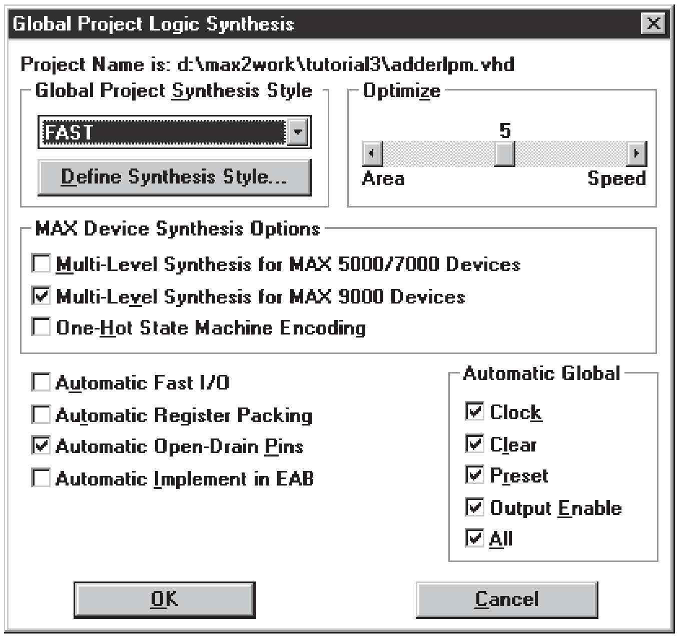
D.3 Design of a Sequential Circuit

This example shows how to implement a sequential circuit using MAX+plusII. The presentation assumes that the reader is familiar with the material in Chapter 7. Figure 7.42 depicts a circuit with a four-bit adder connected to a register that feeds back to the adder. This section shows how to implement the circuit using modules from the LPM library. We will first describe the circuit using a schematic and then give an equivalent design using VHDL code.

D.3.1 Using the Graphic Editor

Select File | New and create a new Graphic Editor file. Save the file with the name *graphic2.gdf*. Use the Ctrl+Shift+j shortcut command to set *graphic2* as the current project. Double-click on the blank space in the Graphic Editor window. Open the *primitives* library by double-clicking on the line in the Symbol Libraries box that ends in prim. Import an *input* symbol using the procedure described in section B.2.2. Import two more *input* symbols and two *output* symbols.

Next we need to import two symbols, *lpm_ff* and *lpm_add_sub*, from the LPM library. The *lpm_ff* module is an n-bit register, and the *lpm_add_sub* module is an adder/subtractor subcircuit. One way to import the modules is to open the


Global Project Logic Synthesis

Project Name is: d:\max2work\tutorial3\adderlpm.vhd

Global Project Synthesis Style: FAST

Define Synthesis Style...

Optimize: 5 (Area – Speed)

MAX Device Synthesis Options

- [ ] Multi-Level Synthesis for MAX 5000/7000 Devices
- [✓] Multi-Level Synthesis for MAX 9000 Devices
- [ ] One-Hot State Machine Encoding

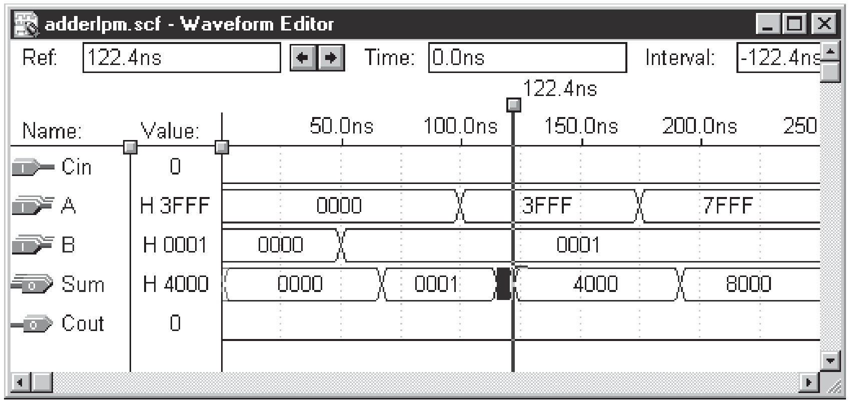
- [ ] Automatic Fast I/O
- [ ] Automatic Register Packing
- [✓] Automatic Open-Drain Pins
- [ ] Automatic Implement in EAB

Automatic Global

- [✓] Clock
- [✓] Clear
- [✓] Preset
- [✓] Output Enable
- [✓] All

OK Cancel

Figure D.14: Setting logic synthesis options.


adderlpm.scf - Waveform Editor

Ref: 122.4ns Time: 0.0ns Interval: -122.4ns

Name:	Value:
Cin	0
A	H 3FFF
B	H 0001
Sum	H 4000
Cout	0

Figure D.15: Timing results when optimized for speed in a FLEX 10K device.

LPM library in the same way as we opened the *primitives* library and then select the appropriate symbols to be imported into the schematic. However, each LPM module has various parameters that have to be set to configure the module. MAX+plusII provides a tool called the *MegaWizard Plug-In Manager* to help configure the modules.

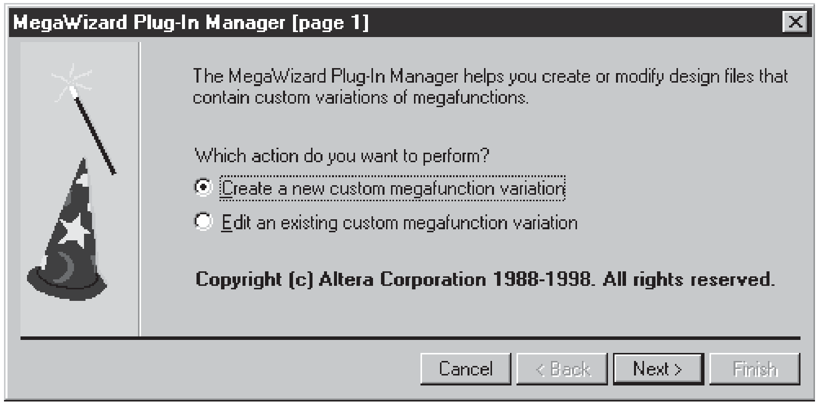
Double-click on a blank space in the Graphic Editor to open the Enter Symbol dialog box. Click on the MegaWizard Plug-In Manager button to open the window shown in Figure D.16. Another way to start this tool is to select File | MegaWizard Plug-In Manager. Click on Next to create a new instance of an LPM module to be imported into the schematic.

In the next page of the MegaWizard Plug-In Manager, shown in Figure D.17, click on the + symbol next to the LPM storage item in the Available Megafunctions box. Then click on LPM_FF to select this module. In addition to creating a symbol for use in a schematic, the MegaWizard tool creates a design file for use with a hardware description language. In Figure D.17 we clicked on VHDL, but we will not use the resulting VHDL file here. It is necessary to provide a name that will be used for the symbol file. We chose the name *Reg4*, as shown in the figure.

Click Next to open the window in Figure D.18. Click on the drop-down menu and set the number of flip-flops in the *lpm_ff* module to 4. The default flip-flop type is now set to D, and the MegaWizard displays the graphical symbol for the *lpm_ff* that will be created for use in the schematic. Click Next to open the window in Figure D.19. Under the item Asynchronous inputs, click to select Clear. Note that the graphical symbol shown for the module now includes the asynchronous reset input. Click on Finish to return to the Enter Symbol dialog box. The name of the *Reg4* symbol is automatically entered in the box labeled Symbol Name. Click OK to return to the Graphic Editor.

Start the MegaWizard tool again to import an instance of the *lpm_add_sub* module. In the screen shown in Figure D.20, click on the + symbol beside LPM arithmetic and then click on LPM_ADD_SUB. Select VHDL for the type of HDL source code file that should be generated. Be sure to make this selection because we will use the generated VHDL file in section D.3.4. As shown in the figure, use the name *Adder4* for the symbol. Click Next to open the window in Figure D.21. Use the drop-down menu to configure the module as a four-bit adder. Click Finish to return to the Enter Symbol box and then click OK.

The symbols that have been imported into the schematic are given in Figure


MegaWizard Plug-In Manager [page 1]

The MegaWizard Plug-In Manager helps you create or modify design files that contain custom variations of megafunctions.

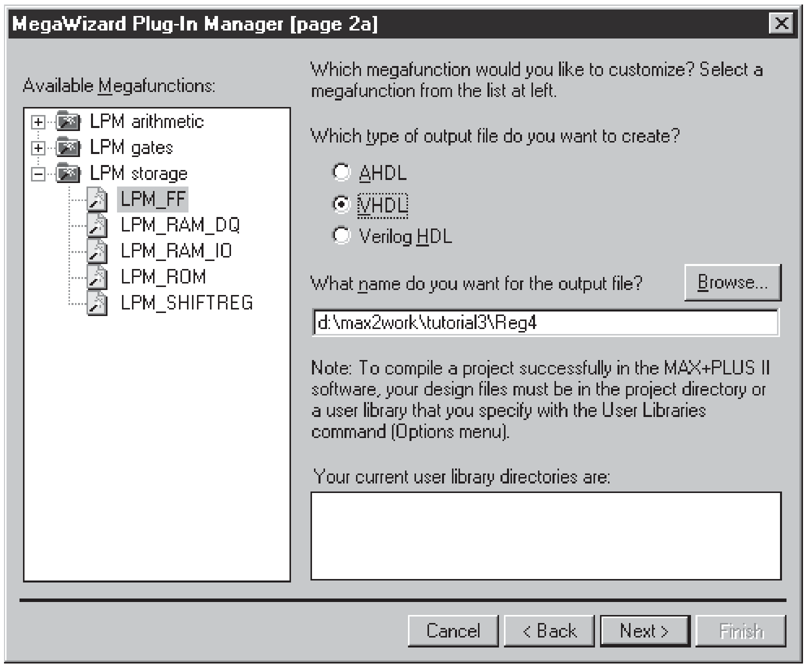
Which action do you want to perform?

[✓ Create a new custom megafunction variation]

Edit an existing custom megafunction variation

Copyright (c) Altera Corporation 1988-1998. All rights reserved.

Figure D.16: Using the MegaWizard Plug-In Manager.



MegaWizard Plug-In Manager [page 2a]

Available Megafunctions:

- LPM arithmetic
- LPM gates
- LPM storage
  - LPM_FF
  - LPM_RAM_DQ
  - LPM_RAM_IO
  - LPM_ROM
  - LPM_SHIFTREG

Which megafunction would you like to customize? Select a megafunction from the list at left.

Which type of output file do you want to create?

AHDL [✓ VHDL] Verilog HDL

What name do you want for the output file? Browse...

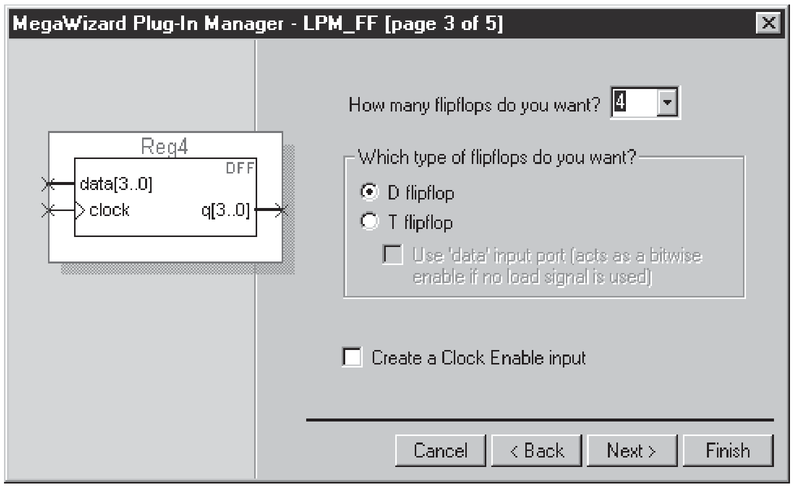
d:\max2work\tutorial3\Reg4

Note: To compile a project successfully in the MAX+PLUS II software, your design files must be in the project directory or a user library that you specify with the User Libraries command (Options menu).

Your current user library directories are:

Figure D.17: Selecting the *lpm_ff* module.

D.22. As indicated in the figure, assign the name *Reset* to the input symbol in the lower-left corner of the schematic and assign the name *Clock* to the input



MegaWizard Plug-In Manager - LPM_FF [page 3 of 5]

How many flipflops do you want? 4

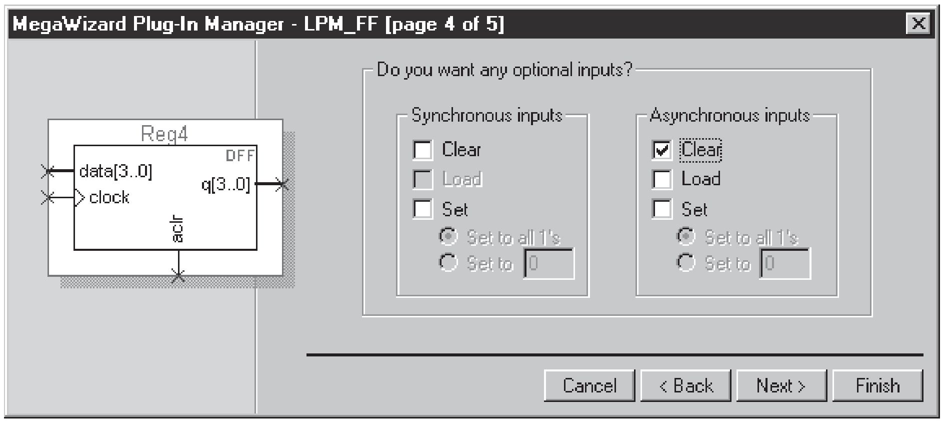
Which type of flipflops do you want?

[✓ D flipflop] T flipflop

Use 'data' input port (acts as a bitwise enable if no load signal is used)

Create a Clock Enable input

Figure D.18: Configuring the *lpm_ff* module.

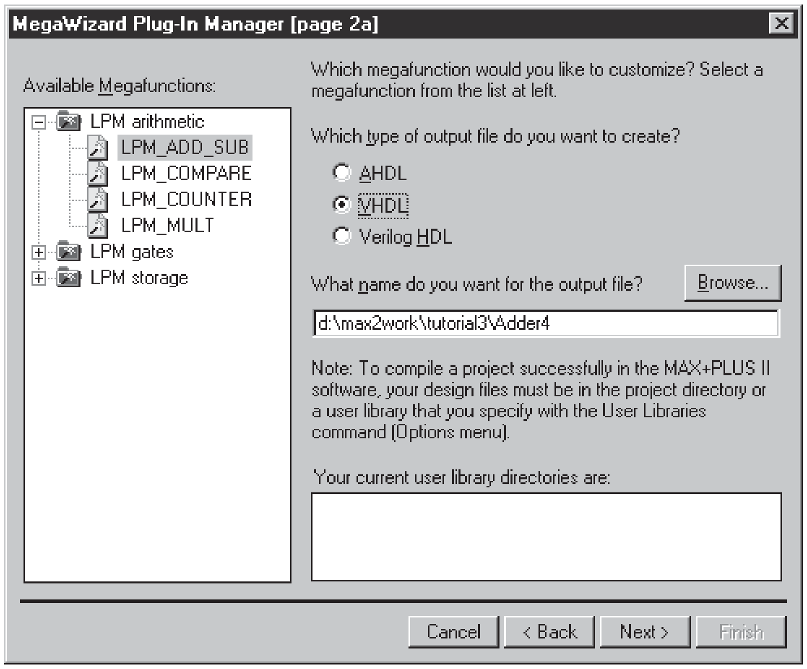

MegaWizard Plug-In Manager - LPM_FF [page 4 of 5]

Do you want any optional inputs?

Synchronous inputs: [ ] Clear, [ ] Load, [ ] Set (Set to all 1's / Set to 0)

Asynchronous inputs: [✓ Clear], [ ] Load, [ ] Set (Set to all 1's / Set to 0)

Figure D.19: Adding a clear input on the *lpm_ff* module.


MegaWizard Plug-In Manager [page 2a]

Available Megafunctions:

LPM arithmetic: LPM_ADD_SUB, LPM_COMPARE, LPM_COUNTER, LPM_MULT; LPM gates; LPM storage

Which megafunction would you like to customize? Select a megafunction from the list at left.

Which type of output file do you want to create?

AHDL [✓ VHDL] Verilog HDL

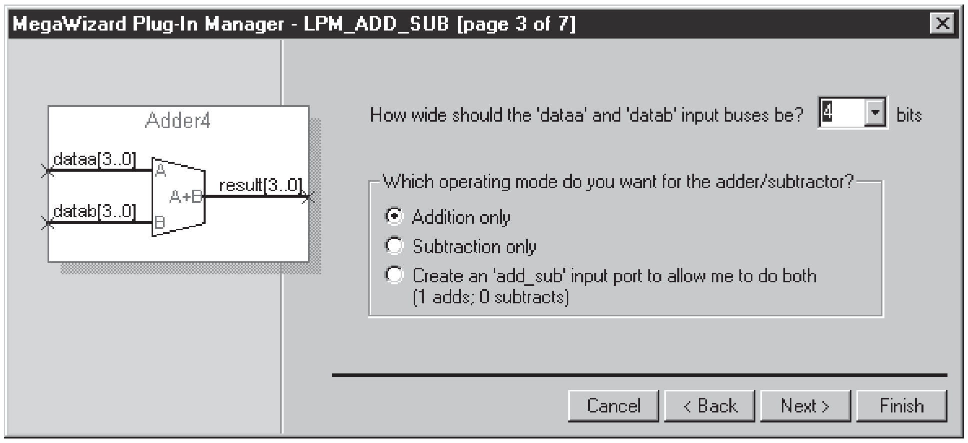
What name do you want for the output file? d:\max2work\tutorial3\Adder4

Note: To compile a project successfully in the MAX+PLUS II software, your design files must be in the project directory or a user library that you specify with the User Libraries command (Options menu).

Your current user library directories are:

Figure D.20: Importing the *lpm_add_sub* module.

symbol above that. The third input symbol is used for the circuit's four-bit input, named *Data*. Use the syntax *Data* [3..0] for the four-bit signal. Assign the name *Sum* [3..0] to the output symbol in the top-right corner of the schematic; we use this symbol to show the sum produced by the adder in a timing simulation. Assign the name *RegSum* [3..0] to the other output symbol.


MegaWizard Plug-In Manager - LPM_ADD_SUB [page 3 of 7]

How wide should the 'dataa' and 'datab' input buses be? 4 bits

Which operating mode do you want for the adder/subtractor?
- [✓ Addition only]
- Subtraction only
- Create an 'add_sub' input port to allow me to do both (1 adds; 0 subtracts)

Figure D.21: Configuring the *lpm_add_sub* module.

Connecting Nodes with Wires and Names

If not already done, activate the Selection tool in the Graphic Editor by clicking on the icon that looks like an arrowhead on the left edge of the window. As explained in section B.2.2, the Selection tool allows the the Graphic Editor to change automatically between the modes of selecting a symbol or drawing wires to interconnect symbols.

Draw a wire from the *Clock* input symbol to the clock node on the *Reg4* symbol and another wire from *Reset* to the *aclr* node on *Reg4*. Next place the

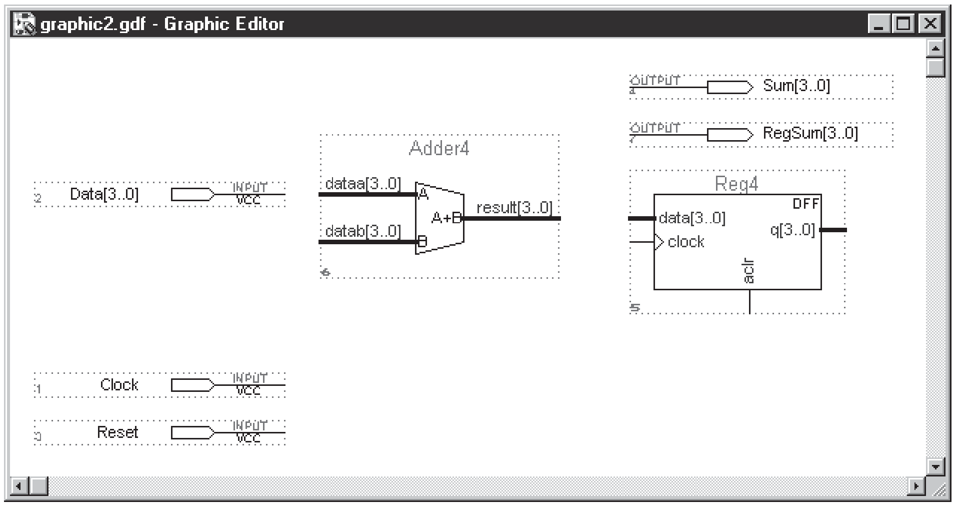


Figure D.22: The symbols imported into the schematic.

mouse on top of the pinstub for the node *dataa*[3..0] on the *Adder4* symbol. Draw a wire from this pinstub to the left until it reaches the pinstub on the *Data*[3..0] input symbol. The wire is drawn as a bold line to indicate that it represents a multibit signal. MAX+plusII uses the term *bus wire* for multibit wires. The Graphic Editor automatically creates a bus wire when the wire is drawn starting at a multibit node such as *dataa*[3..0]. It is possible to manually select the style of line that should be drawn by clicking the right mouse button and selecting the Line Style menu item.

Draw a bus wire from the output of *Adder4* to the *data*[3..0] node on *Reg4*. Draw another wire to connect this node to the *Sum*[3..0] output symbol. Finally, draw a bus wire from the output of *Reg4* back to the *datab*[3..0] node on *Adder4*. To complete the schematic, we need to connect the output of *Reg4* to the *RegSum*[3..0] output symbol. Instead of drawing a wire to make this connection, we will illustrate a different way of connecting the nodes in the schematic. The Graphic Editor allows the user to attach a label to a wire. To attach a label to the bus wire between the output on *Reg4* and the input on *Adder4*, click the mouse somewhere on the wire. The wire is highlighted to show that it is selected, and a small cursor appears on the wire. Type the label *RegSum*[3..0] and observe that this label appears on the wire. In Figure D.23 we attached the label at a point below the *Adder4* symbol, but the label can be placed anywhere on the wire. The Graphic Editor now considers the bus wire to be physically connected to the *RegSum*[3..0] output symbol, just as if a wire were drawn between them. We say that the nodes are interconnected *by name*, rather than by drawing a wire. Most schematic capture tools allow nodes to be interconnected in this manner. In large schematics, connecting nodes by name allows the schematic diagram to appear less cluttered because it means that

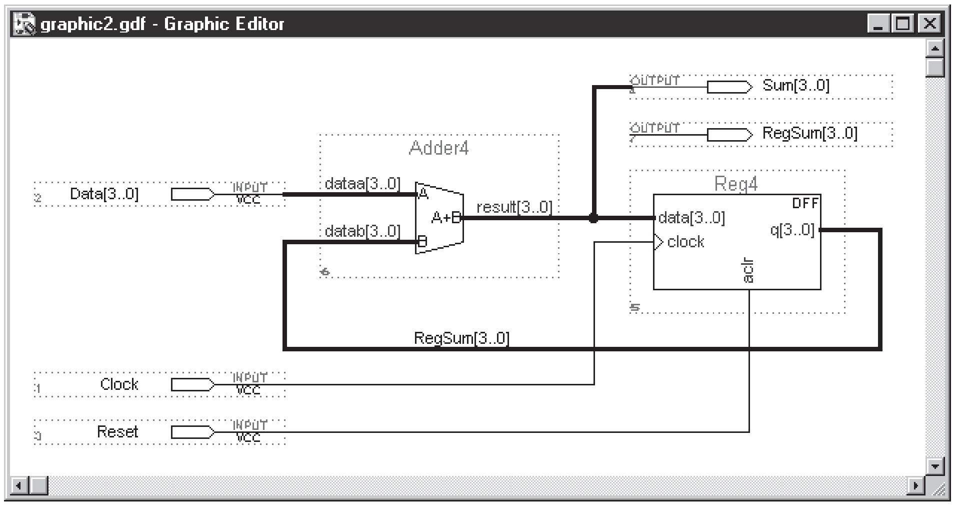


Figure D.23: The completed schematic.

fewer wires have to be drawn.

For completeness we should also mention an alternative way to specify the name (label) of a bus wire. It can be specified using individual signal names separated by commas. In this example we could type the label as follows: $\boxed{RegSum3,\ RegSum2,\ RegSum1,\ RegSum0}$. This style of label allows a bus wire to be composed of any group of node names. For instance, if nodes x, y, and z exist in a design, then a three-bit bus could have the label $\boxed{x,\ y,\ z}$.

In some schematics it is necessary to connect the individual signals within a bus to other nodes. For example, if a bus exists called $C\,[3..0]$, an individual signal in this bus can be accessed by assigning an appropriate label to a node. For instance, to connect a node to the right-most bit in $C\,[3..0]$, the node would be given the label $C\,[0]$. An example of a schematic that uses labels in this manner appears in Figure 10.33 in section 10.2.5.

D.3.2 Synthesizing a Circuit and Using the Timing Simulator

Save the completed schematic. Use the Assign menu to choose a device from the MAX 7000S family. For the results shown here, we selected the EPM7128SLC84-7 device. Compile the design. Open the Waveform Editor and use Node | Enter Nodes from SNF to select the nodes shown in Figure D.24. Set the total simulation time to 500 ns and set the grid size to 25 ns. Set *Reset* to 1 for the first 50 ns of the simulation time and then leave *Reset* at 0 for the rest of the time. To enter the waveform for the clock signal, point the mouse at the name of the *Clock* waveform in the Waveform Editor display and click the right mouse button. In the pop-up menu, select Overwrite | Clock to open the dialog box shown in Figure D.25. The box specifies that the *Clock* signal has the initial value 0 and its period is equal to twice the grid size (50 ns). Click OK, and the Waveform Editor automatically draws the periodic clock signal.

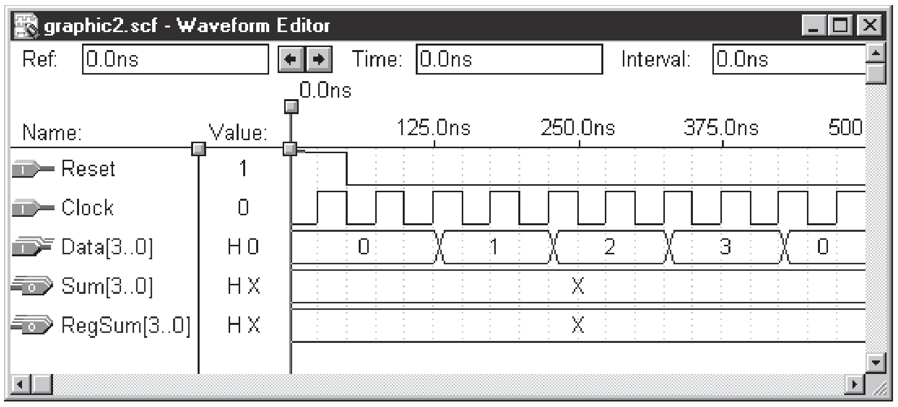


Figure D.24: The input waveforms.

The next step is to draw the waveform for the *Data*[3..0] input. In a typical digital system, input signals change values shortly after the active clock edge because they are the outputs of a register controlled by the same clock. In this example all changes in the *Data*[3..0] input will occur 5 ns after a positive clock edge. Change the grid size to 5 ns. Use the magnifying glass button to zoom in on the Waveform Editor display and draw the waveform shown in Figure D.24 for *Data*[3..0]. In this waveform changes to the *Data*[3..0] signal occur at 130 ns, 230 ns, 330 ns, and 430 ns. Save the waveform file and run the Timing Simulator to generate the simulation results given in Figure D.26. Observe that 7.5 ns are needed to generate a sum from the adder, which is then clocked into the register at the next active clock edge.

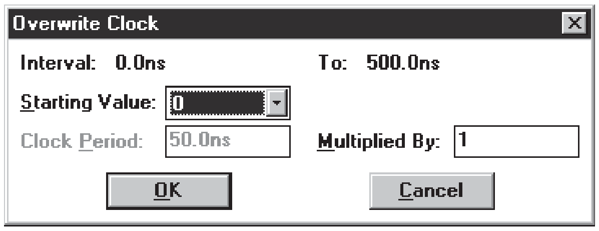
D.3.3 Using the Timing Analyzer

We introduced the Timing Analyzer in section D.1.4 and showed that it can display propagation delays in a circuit. Another use of the Timing Analyzer is to calculate the minimum clock period for which a sequential circuit will operate correctly. Open the Timing Analyzer and select Analysis | Registered Performance to open the window in Figure D.27. Click the Start button. The Timing Analyzer reports that the circuit operates correctly with a minimum clock period of 17 ns. This clock period accounts for all propagation delays in the circuit and for the setup time at the data input on *Reg4*. The Timing Analyzer can also be used to check for setup and hold time violations in a circuit by selecting Analysis | Setup/Hold Matrix.

We have now finished working with the *graphic2* project.

D.3.4 Using VHDL Code

Figure D.10 shows an example of instantiating an LPM module in VHDL code. It uses the GENERIC parameters to configure the module. Another way to do this is to make use of the MegaWizard Plug-In Manager. In Figures D.16 to D.19, we used this tool to create an instance of the *lpm_ff* module. In addition to creating a symbol file for use in the *graphic2* schematic, the MegaWizard created two other files, called *Reg4.vhd* and *Reg4.cmp*. *Reg4.vhd* represents a subcircuit in the same way that *fulladd.vhd* in Figure D.1 represents a subcircuit. Figure



Interval: 0.0ns To: 500.0ns
Starting Value: 0
Clock Period: 50.0ns Multiplied By: 1

Figure D.25: Creating a clock waveform.

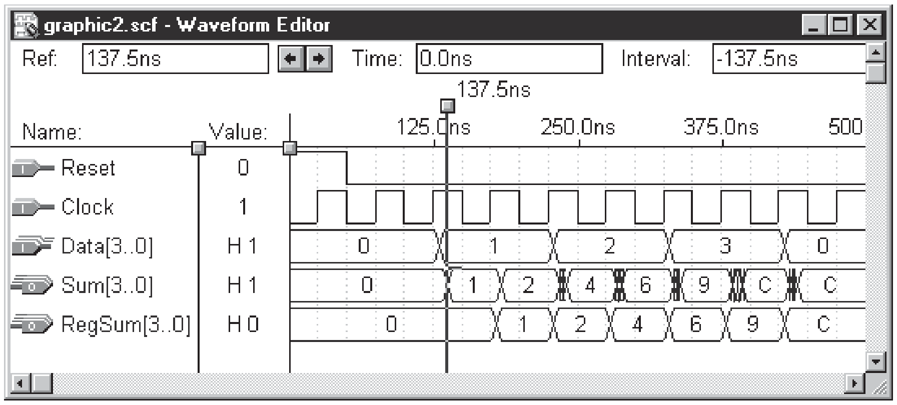
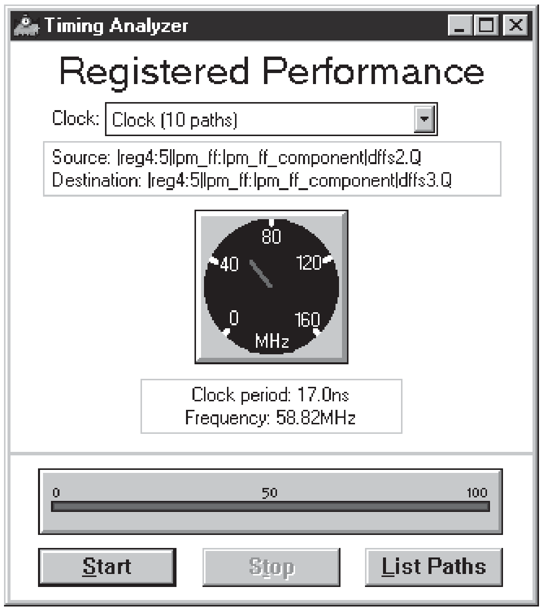


Figure D.26: The results produced by timing simulation.

D.28 shows VHDL code that is equivalent to the *graphic2* schematic. *Reg4* is declared as a component in the architecture. This component declaration statement was copied and pasted from the *Reg4.cmp* file. The adder in the circuit is specified using the + operator in the statement "Sum <= Data + RegSum". The register is created by instantiating the *Reg4* component. The configuration of the *lpm_ff* module using its GENERIC parameters is not shown in the code, because this is done in the *Reg4.vhd* file created by the MegaWizard



Source: |reg4:5|lpm_ff:lpm_ff_component|dffs2.Q
Destination: |reg4:5|lpm_ff:lpm_ff_component|dffs3.Q

Clock period: 17.0ns
Frequency: 58.82MHz

Figure D.27: Maximum clock frequency estimate.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY example3 IS
     PORT ( Clock, Reset   : IN       STD_LOGIC ;
            Data           : IN       STD_LOGIC_VECTOR(3 DOWNTO 0) ;
            RegSum         : BUFFER   STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END example3 ;

ARCHITECTURE Behavior OF example3 IS
     SIGNAL Sum : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
     COMPONENT Reg4
          PORT ( clock   : IN    STD_LOGIC;
                 q       : OUT   STD_LOGIC_VECTOR (3 DOWNTO 0) ;
                 aclr    : IN    STD_LOGIC ;
                 data    : IN    STD_LOGIC_VECTOR (3 DOWNTO 0) ) ;
     END COMPONENT ;
BEGIN
     Sum <= Data + RegSum ;
     R1: Reg4 PORT MAP ( Clock, RegSum, Reset, Sum ) ;
END Behavior ;
```

Figure D.28: VHDL code equivalent to the *graphic2* project.

tool.

The reader should type the code in Figure D.28 into a Text Editor file, compile it, and simulate the resulting circuit. The results should be identical to those produced for the *graphic2* project.

D.4 Design of a Finite State Machine

In section 8.1 we show a simple Moore-type finite state machine 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 D.29. VHDL code that describes the machine appears in Figure 8.29; it is reproduced in Figure D.30. Create a new Text Editor file and enter the code shown in Figure D.30. Save the file with the name *simple.vhd* and set this as the name of the current project.

D.4.1 Implementation in a CPLD

Use the Assign menu to select a MAX 7000S device. We chose the EPM7128SLC84-7 chip for the results presented here. Run the Compiler to synthesize a circuit

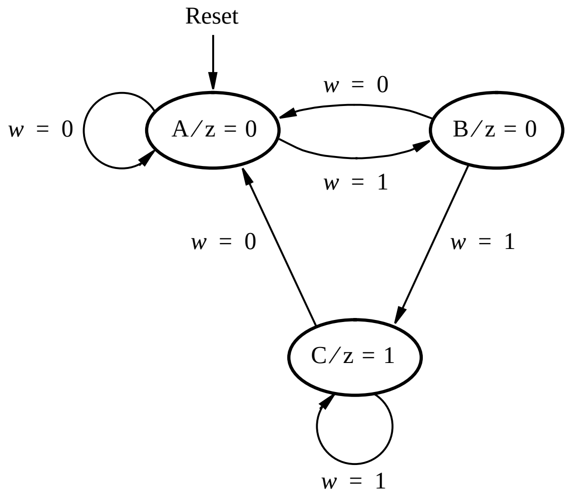


Figure D.29: State diagram of a Moore-type FSM.

for the FSM. Open the Compiler report file and scroll down until the section that shows the state assignment is visible, as illustrated in Figure D.31. Two state variables are used, named y_2 and y_1. The codes assigned to the three states, $A = 00$, $B = 01$, and $C = 10$ are the same ones shown in Figure 8.6.

Scroll down further in the report file to see the logic expressions shown in Figure D.32. These expressions can be derived by hand by considering the state diagram and state assignment. For example, state variable y_1 has the value 1 only in state B, and the state diagram specifies that the machine changes to state B only if it is currently in state A and $w = 1$. Since A has the code $y_2y_1 = 00$, then y_1 should be set to 1 if $y_2 = y_1 = 0$ and $w = 1$. Hence the expression for y_1 in Figure D.32 is

$$y_1 = w\overline{y}_1\overline{y}_2$$

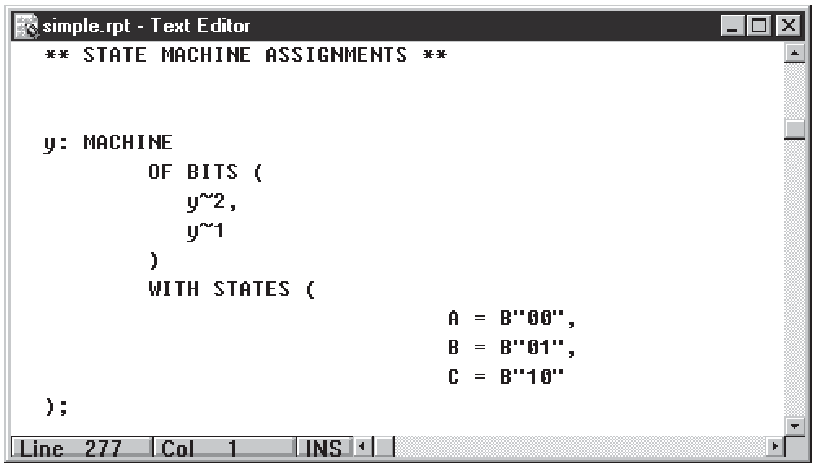
The expressions for y_2 and z are derived similarly.

Open the Waveform Editor and import the nodes *Resetn*, *Clock*, w, z, and y, as shown in Figure D.33. 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. Create the periodic *Clock* signal as described in section D.3.2. Draw the waveform for w shown in the figure. Each change in w occurs 5 ns after a positive clock edge; use a grid size of 5 ns to draw this waveform. Save the file and run the Timing Simulator to generate the results shown. 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. It shows that changes in the FSM's state occur 2.5 ns after an active clock edge and that an additional 7 ns are needed to change the value of the z output.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY simple IS
    PORT ( Clock, Resetn  : IN    STD_LOGIC ;
           w              : IN    STD_LOGIC ;
           z              : OUT   STD_LOGIC ) ;
END simple ;

ARCHITECTURE Behavior OF simple IS
    TYPE STATE_TYPE IS (A, B, C) ;
    SIGNAL y : STATE_TYPE ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= B ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= C ;
                    END IF ;
                WHEN C =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= C ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;
    z <= '1' WHEN y = C ELSE '0' ;
END Behavior ;
```

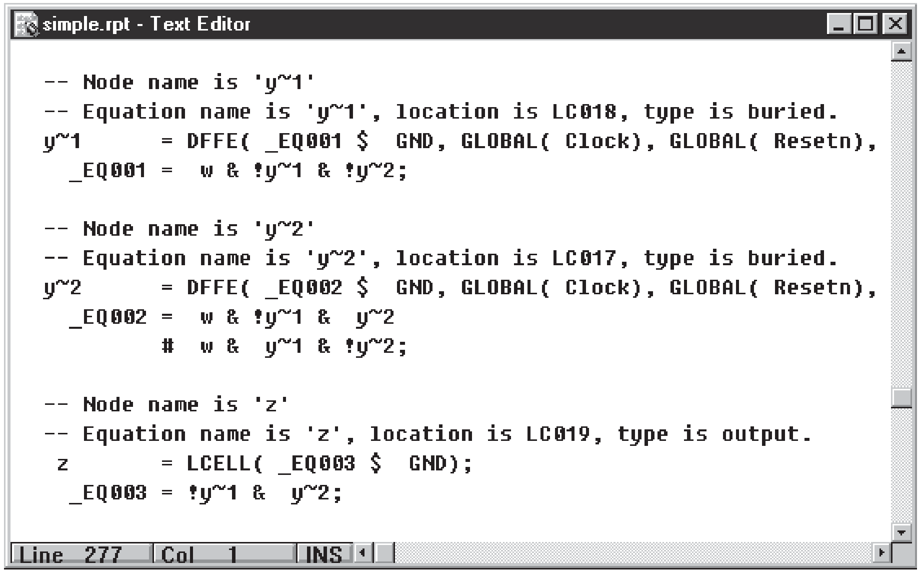
Figure D.30: VHDL code for the FSM in Figure D.29.

Open the Timing Analyzer. Select Analysis | Registered Performance and click the Start button. The analysis reports that the FSM operates correctly with a maximum clock frequency of 125 MHz.



```
** STATE MACHINE ASSIGNMENTS **


y: MACHINE
        OF BITS (
           y~2,
           y~1
        )
        WITH STATES (
                               A = B"00",
                               B = B"01",
                               C = B"10"
);
```

Figure D.31: State assignment for the VHDL code in Figure D.30.



```
-- Node name is 'y~1'
-- Equation name is 'y~1', location is LC018, type is buried.
y~1      = DFFE( _EQ001 $  GND, GLOBAL( Clock), GLOBAL( Resetn),
  _EQ001 =  w & !y~1 & !y~2;

-- Node name is 'y~2'
-- Equation name is 'y~2', location is LC017, type is buried.
y~2      = DFFE( _EQ002 $  GND, GLOBAL( Clock), GLOBAL( Resetn),
  _EQ002 =  w & !y~1 &  y~2
         #  w &  y~1 & !y~2;

-- Node name is 'z'
-- Equation name is 'z', location is LC019, type is output.
 z       = LCELL( _EQ003 $  GND);
  _EQ003 = !y~1 &  y~2;
```

Figure D.32: Logic expressions for the code in Figure D.30.

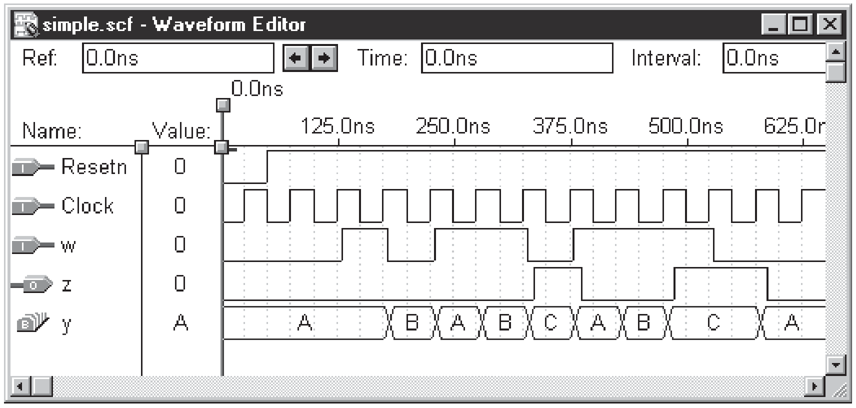
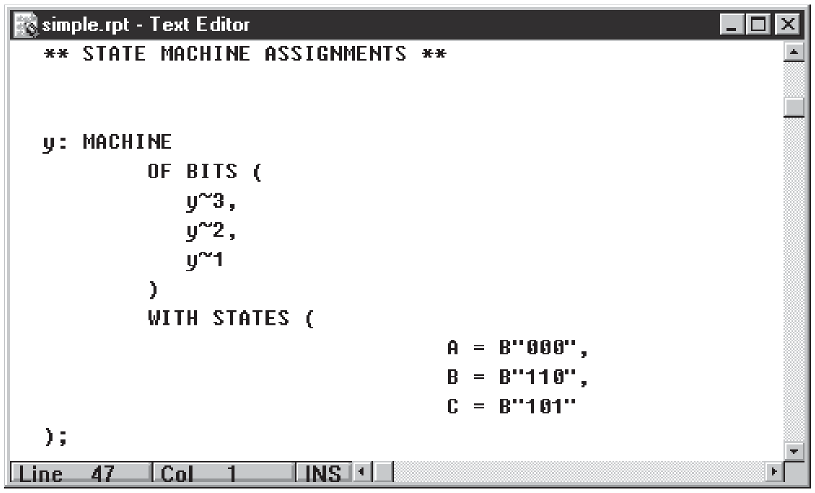


Figure D.33: Timing simulation for the code in Figure D.30.

D.4.2 Implementation in an FPGA

To see how the FSM is implemented in an FPGA, use the Assign menu to select a device in the FLEX 10K family. We chose the EPF10K20RC240-4 for the results shown here. Run the Compiler to synthesize a new circuit for the FSM.

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. Open the Compiler report file and scroll down to the section depicted in Figure D.34. It shows that three state variables are used for the FLEX 10K implementation. The state assignment is the same as the one-hot encoding except that state variable y_3 is complemented. As discussed in section 8.8.1, this



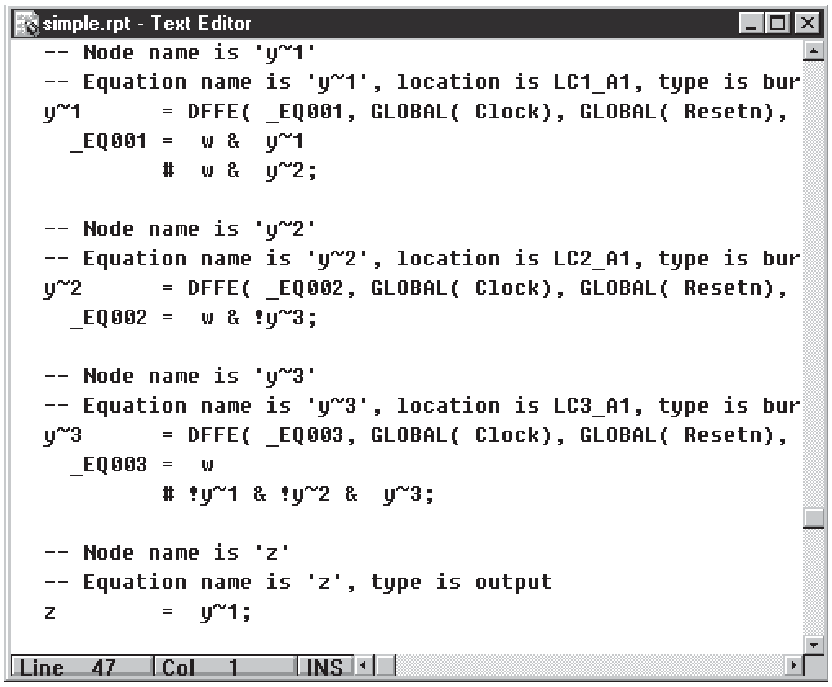
```
** STATE MACHINE ASSIGNMENTS **



y: MACHINE
        OF BITS (
           y~3,
           y~2,
           y~1
        )
        WITH STATES (
                                  A = B"000",
                                  B = B"110",
                                  C = B"101"
);
```

Figure D.34: State assignment using a FLEX 10K chip.

modified form of one-hot encoding provides a simple reset mechanism because it assigns the code 000 to the starting state A.

The logic expressions for the circuit are displayed in Figure D.35. They can be derived by first assuming that one-hot encoding is used, such that $A = 100, B = 010$, and $C = 001$. For this encoding, derive expressions for y_3, y_2, y_1, and z. Since the actual codes use the opposite values for y_3, then complement variable y_3 in each of the derived expressions and, in addition, complement the entire expression for y_3. We leave the detailed derivation as an exercise for the reader.

Rerun the Timing Simulator using the input waveforms in Figure D.33. Use the reference line in the Waveform Editor to show that changes in the state variables occur 6.4 ns after an active clock edge. Changes in the z output require an additional 6.7 ns. Use the Timing Analyzer to see that the FSM runs correctly at the same maximum clock rate as in the MAX 7000S device, namely, 125 MHz. While the Compiler always uses one-hot encoding when the target chip is a FLEX 10K device, it can also use this encoding for a MAX 7000 device. The reader can experiment by again selecting a MAX 7000S chip and using the logic synthesis dialog box (see Figure D.14) to enable the One-Hot State Machine Encoding option.



```
-- Node name is 'y~1'
-- Equation name is 'y~1', location is LC1_A1, type is bur
y~1      = DFFE( _EQ001, GLOBAL( Clock), GLOBAL( Resetn),
  _EQ001 =  w &  y~1
         #  w &  y~2;

-- Node name is 'y~2'
-- Equation name is 'y~2', location is LC2_A1, type is bur
y~2      = DFFE( _EQ002, GLOBAL( Clock), GLOBAL( Resetn),
  _EQ002 =  w & !y~3;

-- Node name is 'y~3'
-- Equation name is 'y~3', location is LC3_A1, type is bur
y~3      = DFFE( _EQ003, GLOBAL( Clock), GLOBAL( Resetn),
  _EQ003 =  w
         # !y~1 & !y~2 &  y~3;

-- Node name is 'z'
-- Equation name is 'z', type is output
z        =  y~1;
```

Figure D.35: Logic expressions using a FLEX 10K chip.

D.5 Concluding Remarks

In Tutorials 1, 2, and 3, we have introduced many of the most important features of MAX+plusII. However, many other features are available. The reader is encouraged to learn about the more advanced capabilities of the CAD system by exploring the various commands and on-line help provided in each application.