

Tutorial — Using Quartus II CAD Software

This tutorial introduces the basic features of Quartus[®] II software. It shows how the software can be used to design and implement a circuit specified using the Verilog hardware description language.

1 Getting Started

Each logic circuit, or subcircuit, being designed in Quartus II is called a *project*. The software works on one project at a time and keeps all information for that project in a single directory (folder) in the file system. To begin a new logic circuit design, the first step is to create a directory to hold its files. As part of the installation of the Quartus II software, a few sample projects are placed into a directory called *\qdesigns*. As part of the installation of QUIP, a directory *\quartus_tutorial* is created, which contains this tutorial as well as the directory *verilog_files* that contains the Verilog files used in the tutorial. To hold the design files for this tutorial, we will use a directory *\quartus_tutorial1*. The running example for this tutorial is a simple adder/subtractor circuit, which is defined in the Verilog hardware description language.

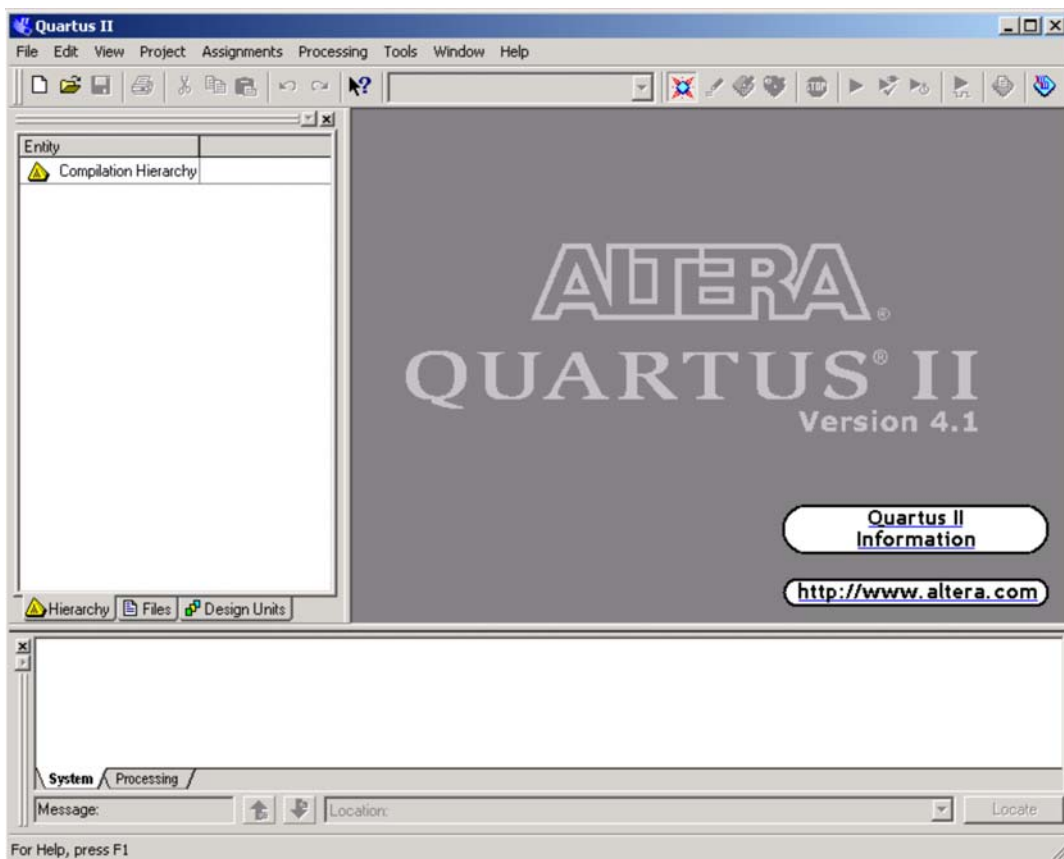


Figure 1. The main Quartus II display.

Start the Quartus II software. You should see a display similar to the one in Figure 1. This display consists of several windows that provide access to all the features of Quartus II, which the user selects with the computer

mouse. Most of the commands provided by Quartus II can be accessed by using a set of menus that are located below the title bar. For example, in Figure 1 clicking the left mouse button on the menu named **File** opens the menu shown in Figure 2. Clicking the left mouse button on the entry **Exit** exits from Quartus II. In general, whenever the mouse is used to select something, the *left* button is used. Hence we will not normally specify which button to press. In the few cases when it is necessary to use the *right* mouse button, it will be specified explicitly. For some commands it is necessary to access two or more menus in sequence. We use the convention **Menu1 | Menu2 | Item** to indicate that to select the desired command the user should first click the left mouse button on **Menu1**, then within this menu click on **Menu2**, and then within **Menu2** click on **Item**. For example, **File | Exit** uses the mouse to exit from the Quartus II system. Many Quartus II commands can be invoked by clicking on an icon displayed in one of the toolbars. To see the list of available toolbars, select **Tools | Customize | Toolbars**. Once a toolbar is opened, it can be moved using the mouse, and icons can be dragged from one toolbar to another. To see the Quartus II command associated with an icon, position the mouse over the icon and a tooltip will appear that displays the command name.

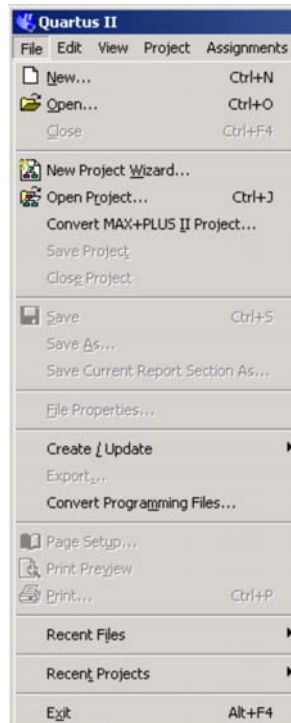


Figure 2. An example of File menu.

It is possible to modify the appearance of the Quartus II display in Figure 1 in many ways. Section 7 shows how to move, resize, close, and open windows within the main Quartus II display.

1.1 Quartus II On-Line Help

Quartus II provides comprehensive on-line documentation that answers many of the questions that may arise when using the software. The documentation is accessed from the menu in the **Help** window. To get some idea of the extent of documentation provided, it is worthwhile for the reader to browse through the **Help** menu. For instance, selecting **Help | How to Use Help** gives an indication of what type of help is provided.

The user can quickly search through the Help topics by selecting **Help | Search**, which opens a dialog box into which key words can be entered. Another method, context-sensitive help, is provided for quickly finding

documentation for specific topics. While using most applications, pressing the F1 function key on the keyboard opens a Help display that shows the commands available for the application.

2 Starting a New Project

To start working on a new design we first have to define a new *design project*. Quartus II makes the designer's task easy by providing support in the form of a *wizard*. Select **File | New Project Wizard** to reach a window that indicates the capability of this wizard. Press **Next** to get the window shown in Figure 3. Set the working directory to be *quartus_tutorial1*. The project must have a name, which is usually the same as the top-level design entity that will be included in the project. Choose *addersubtractor* as the name for both the project and the top-level entity, as shown in Figure 3. Press **Next**. Since we have not yet created the directory *quartus_tutorial1*, Quartus II displays the pop-up box in Figure 4 asking if it should create the desired directory. Click **Yes**, which leads to the window in Figure 5. This window makes it easy to specify which existing files (if any) should be included in the project. Assuming that we do not have any existing files, click **Next**, which leads to the window in Figure 6. Here, we can specify the type of device in which the designed circuit will be implemented. Choose StratixTM as the target device family. We can let Quartus II select a specific device in the family, or we can choose the device explicitly. We will take the latter approach, so make sure that **Specific device selected** in 'Available devices' list is selected under "Target device". Under "Available devices", choose the device called EP1S10F484C5. Press **Next** to go to the window shown in Figure 7. Here, one can specify any third-party tools that should be used. A commonly used term for CAD software for electronic circuits is *EDA tools*, where the acronym stands for electronic design automation. This term is used in Quartus II messages that refer to third-party tools, which are the tools developed and marketed by companies other than Altera. Since we will rely solely on Quartus II, we will not choose any other tools. Press **Next**. Now, a summary of the chosen settings appears in the screen shown in Figure 8. Press **Finish**, which returns to the main Quartus II window, but with *addersubtractor* specified as the new project, in the display title bar, as indicated in Figure 9.

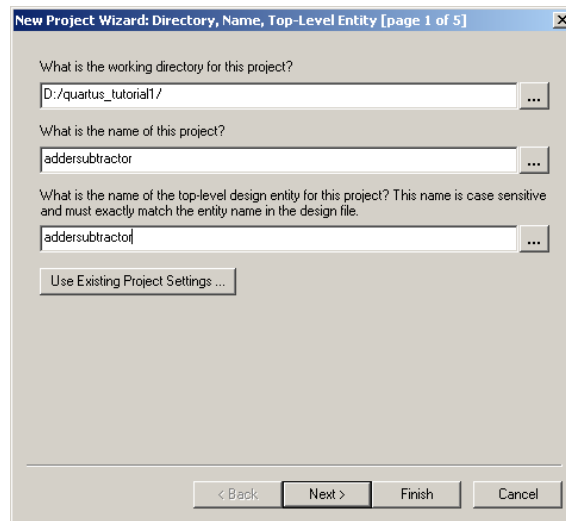


Figure 3. Creation of a new project.

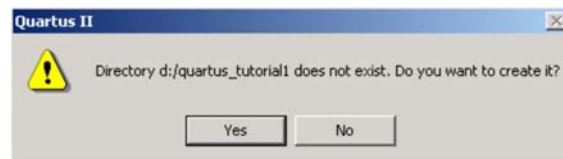


Figure 4. Quartus II can create a new directory for the project.

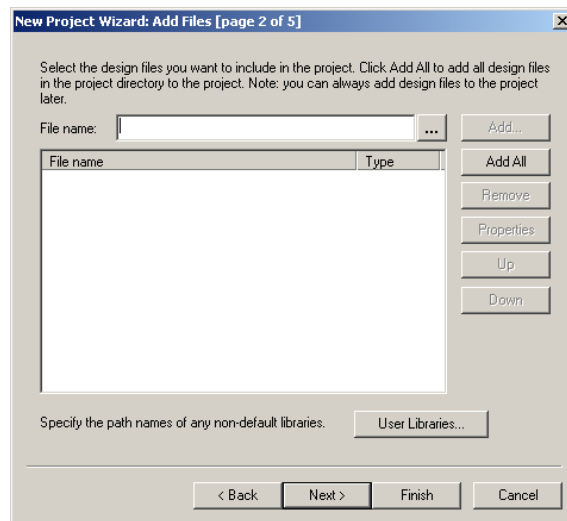


Figure 5. The wizard can include user-specified design files.

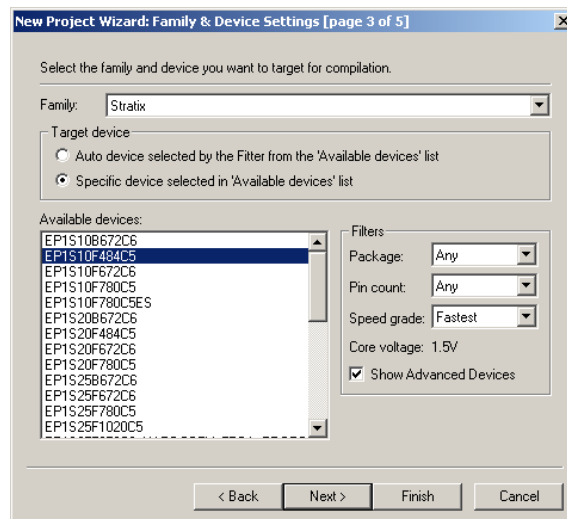


Figure 6. Choose the device family and specify a particular device.

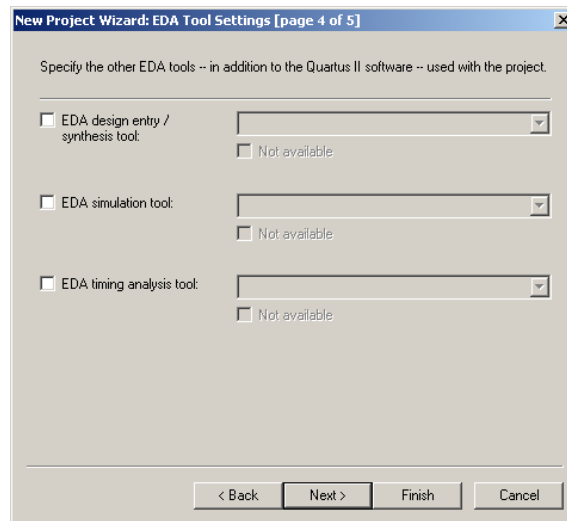


Figure 7. Other EDA tools can be specified.

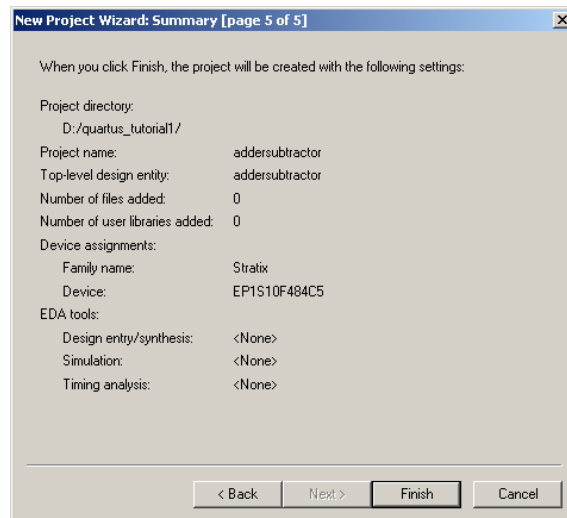


Figure 8. Summary of the project settings.

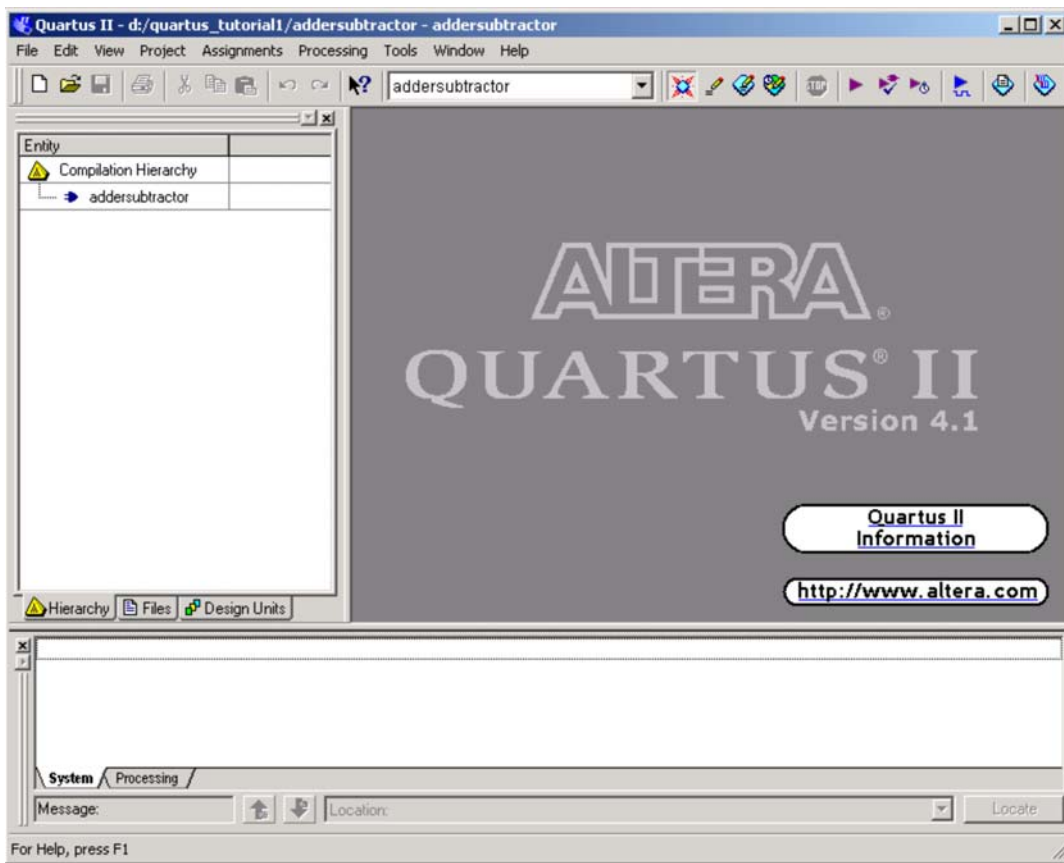


Figure 9. The Quartus II display for the created project.

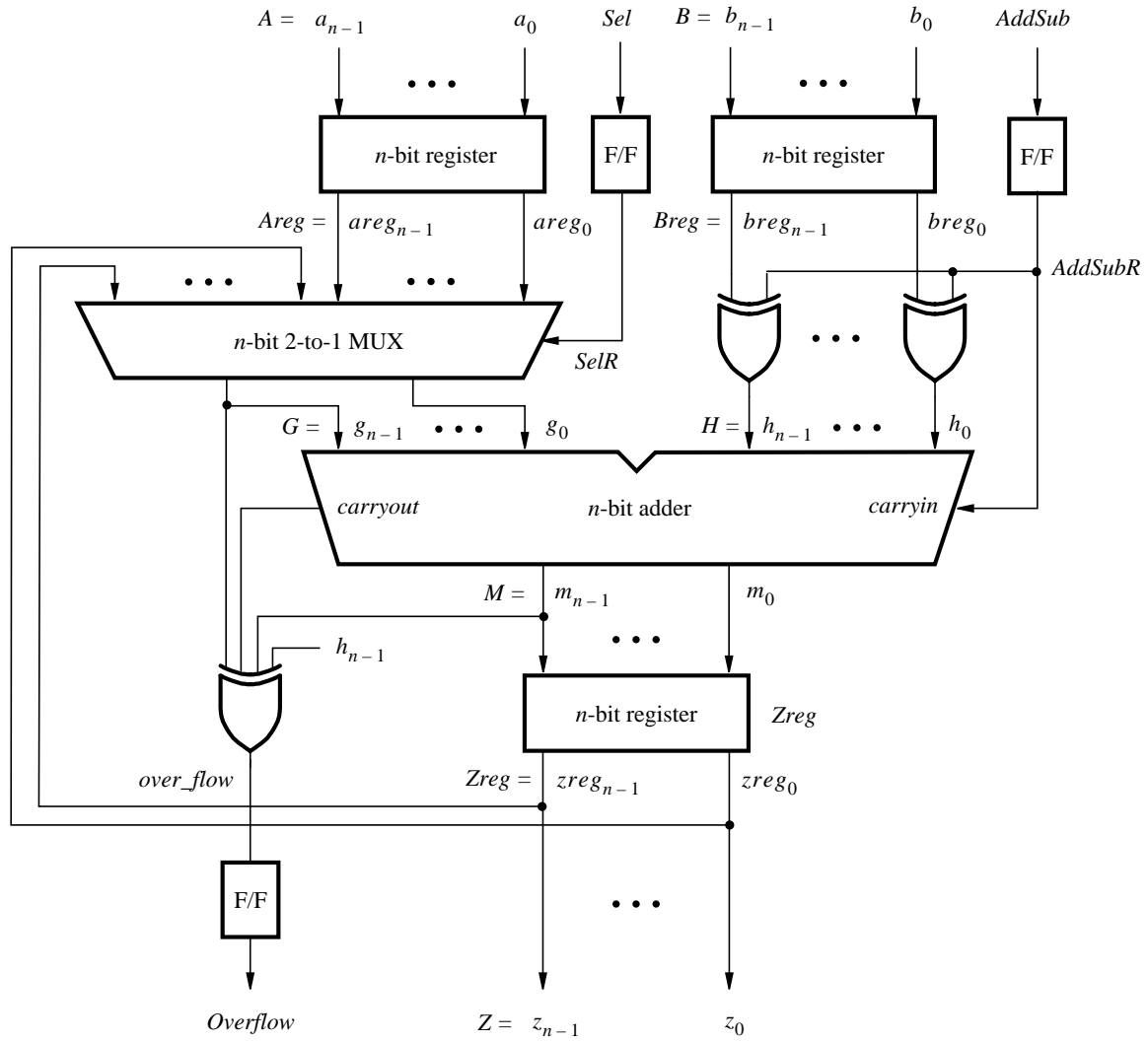


Figure 10. The adder/subtractor circuit.

3 Design Entry Using Verilog Code

As a design example, we will use the adder/subtractor circuit shown in Figure 10. The circuit can add, subtract, and accumulate n -bit numbers using the 2's complement number representation. The two primary inputs are numbers $A = a_{n-1}a_{n-2} \cdots a_0$ and $B = b_{n-1}b_{n-2} \cdots b_0$, and the primary output is $Z = z_{n-1}z_{n-2} \cdots z_0$. Another input is the $AddSub$ control signal which causes $Z = A + B$ to be performed when $AddSub = 0$ and $Z = A - B$ when $AddSub = 1$. A second control input, Sel , is used to select the accumulator mode of operation. If $Sel = 0$, the operation $Z = A \pm B$ is performed, but if $Sel = 1$, then B is added to or subtracted from the current value of Z . If the addition or subtraction operations result in arithmetic overflow, an output signal, $Overflow$, is asserted.

To make it easier to deal with asynchronous input signals, we will load them into flip-flops on a positive edge of the clock. Thus, inputs A and B will be loaded into registers $Areg$ and $Breg$, while Sel and $AddSub$ will be loaded into flip-flops $SelR$ and $AddSubR$, respectively. The adder/subtractor circuit places the result into register $Zreg$.

```

// Top-level module
module addersubtractor (A, B, Clock, Reset, Sel, AddSub, Z, Overflow);
    parameter n = 16;
    input [n-1:0] A, B;
    input Clock, Reset, Sel, AddSub;
    output [n-1:0] Z;
    output Overflow;
    reg SelR, AddSubR, Overflow;
    reg [n-1:0] Areg, Breg, Zreg;
    wire [n-1:0] G, H, M, Z;
    wire carryout, over_flow;

// Define combinational logic circuit
    assign H = Breg ^ {n{AddSubR}};
    mux2to1 multiplexer (Areg, Z, SelR, G);
    defparam multiplexer.k = n;
    adderk nbit_adder (AddSubR, G, H, M, carryout);
    defparam nbit_adder.k = n;
    assign over_flow = carryout ^ G[n-1] ^ H[n-1] ^ M[n-1];
    assign Z = Zreg;

// Define flip-flops and registers
    always @(posedge Reset or posedge Clock)
        if (Reset == 1)
            begin
                Areg <= 0; Breg <= 0; Zreg <= 0;
                SelR <= 0; AddSubR <= 0; Overflow <= 0;
            end
        else
            begin
                Areg <= A; Breg <= B; Zreg <= M;
                SelR <= Sel; AddSubR <= AddSub; Overflow <= over_flow;
            end
    endmodule

... continued in Part b

```

Figure 11. Verilog code for the circuit in Figure 10 (Part a).

The required circuit is described by the Verilog code in Figure 11, which specifies $n = 16$. Note that the top Verilog module is called *addersubtractor* to match the name given in Figure 3, which was specified when the project was created. This code can be typed into a file by using any typical text editor that stores ASCII files, or by using Quartus II text editing facilities. While the file can be given any name, it is a common designers' practice to use the same name as the name of the top-level Verilog module. The file name must include the extension *v*, which indicates a Verilog file. So, we will use the name *addersubtractor.v*. For convenience, we provide the required file in the directory *qdesigns*. Copy this file into the project directory *quartus tutorial1*.

3.1 Using the Quartus II Text Editor

This section shows how to use the Quartus II text editing facilities. You can skip this section if you prefer to use some other text editor to create the *addersubtractor.v* file, or if you have chosen to copy the file from the *qdesigns* directory.


```

// k-bit 2-to-1 multiplexer
module mux2to1 (V, W, Sel, F);
    parameter k = 8;
    input [k-1:0] V, W;
    input Sel;
    output [k-1:0] F;
    reg [k-1:0] F;

    always @(V or W or Sel)
        if (Sel == 0) F = V;
        else F = W;
endmodule

// k-bit adder
module adderk (carryin, X, Y, S, carryout);
    parameter k = 8;
    input [k-1:0] X, Y;
    input carryin;
    output [k-1:0] S;
    output carryout;
    reg [k-1:0] S;
    reg carryout;

    always @(X or Y or carryin)
        {carryout, S} = X + Y + carryin;
endmodule

```

Figure 11. Verilog code for the circuit in Figure 10 (Part *b*).

Select **File | New** to get the window in Figure 12, choose **Verilog HDL File**, and click **OK**. This opens the Text Editor window. The first step is to specify a name for the file that will be created. Select **File | Save As** to open the pop-up box depicted in Figure 13. In the box labeled **Save as type** choose **Verilog HDL File**. In the box labeled **File name type** type *addersubtractor*. Put a checkmark in the box **Add file to current project**. Click **Save**, which puts the file into the directory *quartus_tutorial1* and leads to the Text Editor window shown in Figure 14. Maximize the Text Editor window and enter the Verilog code in Figure 11 into it. Save the file by typing **File | Save**, or by typing the shortcut **Ctrl-s**.

Most of the commands available in the Text Editor are self-explanatory. Text is entered at the *insertion point*, which is indicated by a thin vertical line. The insertion point can be moved either by using the keyboard arrow keys or by using the mouse. Two features of the Text Editor are especially convenient for typing Verilog code. First, the editor can display different types of Verilog statements in different colors, which is the default choice. Second, the editor can automatically indent the text on a new line so that it matches the previous line. Such options can be controlled by the settings in **Tools | Options | Text Editor**.

3.1.1 Using Verilog Templates

The syntax of Verilog code is sometimes difficult for a designer to remember. To help with this issue, the Text Editor provides a collection of *Verilog templates*. The templates provide examples of various types of Verilog statements, such as a **module** declaration, an **always** block, and assignment statements. It is worthwhile to browse through the templates by selecting **Edit | Insert Template | Verilog HDL** to become familiar with this resource.

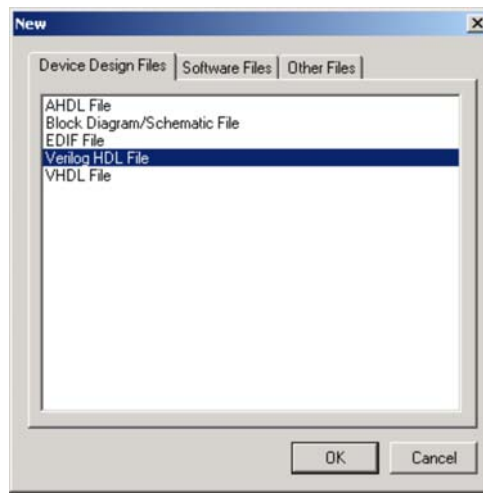


Figure 12. Choose to prepare a Verilog file.

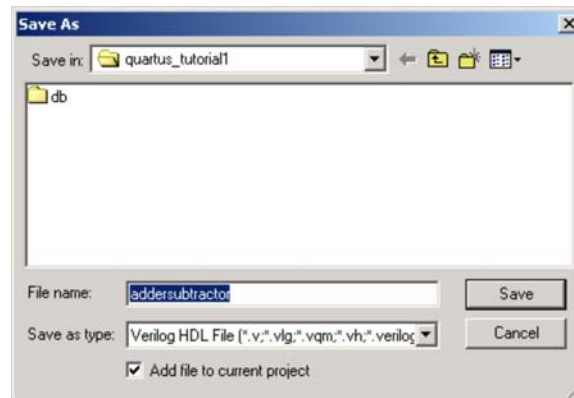


Figure 13. Name the file.

3.2 Adding Design Files to a Project

As we indicated when discussing Figure 5, you can tell Quartus II which design files it should use as part of the current project. To see the list of files already included in the *addersubtractor* project, select **Assignments | Settings**, which leads to the window in Figure 15. As indicated on the left side of the figure, click on the item under **Files**. An alternative way of making this selection is to choose **Project | Add/Remove Files in Project**.

If you used the Quartus II Text Editor to create the file and checked the box labeled **Add file to current project**, as described in Section 3.1, then the *addersubtractor.v* file is already a part of the project and will be listed in the window in Figure 15. Otherwise, the file must be added to the project. If not already done, place a copy of the file *addersubtractor.v* into the directory *quartus_tutorial1*, by getting it from the directory *qdesigns* or by using a file that you created using some other text editor. To add this file to the project, click on **Add...** button in Figure 15 to get the pop-up window in Figure 16. Select the *addersubtractor.v* file and click **Open**. Quartus II indicates the selected file in the **Files** window of Figure 15. Click **OK**. The *addersubtractor.v* file is now included in the project. We should mention that in many cases Quartus II is able to automatically find the right files to use for each entity referenced in Verilog code, even if the file has not been explicitly added to the project. However, for complex projects that involve many files it is a good design practice to specifically add the needed files to the

project, as described above.

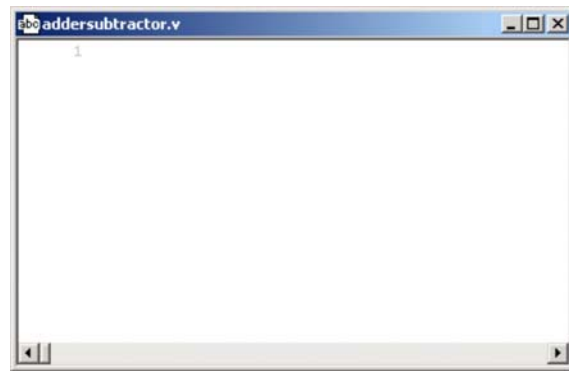


Figure 14. Text Editor window.

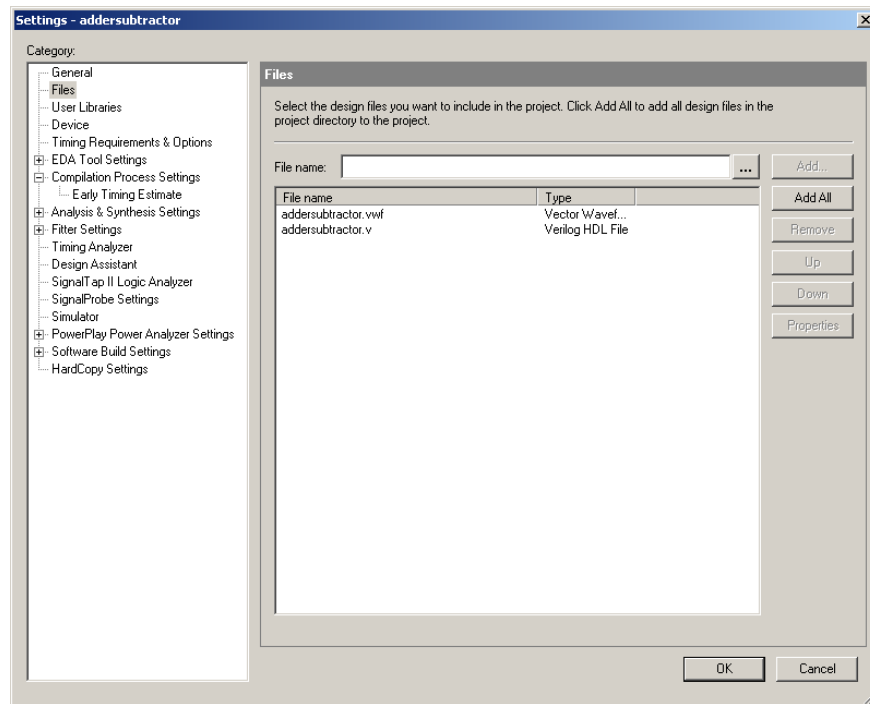


Figure 15. Settings window.

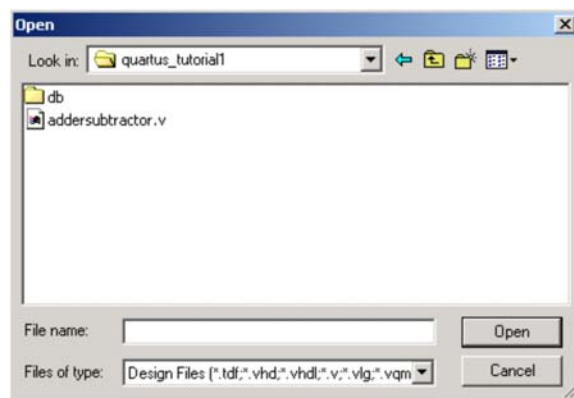


Figure 16. Select file name.

4 Compiling the Verilog Code

The Verilog code is processed by a number of CAD tools that analyze the code and generate an implementation of it for the target chip. In Quartus II these tools are controlled by the application program called the *Compiler*.

Run the Compiler by selecting **Processing | Start Compilation**, or by using the toolbar icon that looks like a solid purple triangle. As the compilation moves through various stages, its progress is reported in the window on the left side. Successful (or unsuccessful) compilation is indicated in a pop-up box. Acknowledge it by clicking OK, which leads to the Quartus II display in Figure 17, in which we have expanded the Compilation Hierarchy in the top left corner to show the hierarchy that comprises all modules in the *addersubtractor* design. In the message window, at the bottom of Figure 17, Quartus II displays various messages. In case of errors, there will be appropriate messages given.

When the compilation is finished, Quartus II produces a compilation report. A window showing this report, displayed in Figure 18, is opened automatically. The window can be resized, maximized, or closed in the normal way, and it can be opened at any time either by selecting **Processing | Compilation Report** or by clicking on the corresponding icon in the toolbar (which looks like a white sheet on top of a blue chip). The report includes a number of sections listed on the left side of its window. Figure 18 displays the Compiler Flow Summary section, which indicates that only a miniscule amount of chip resources are needed to implement this tiny circuit on the selected FPGA chip.

The Compilation Report provides a lot of information that may be of interest to the designer. It indicates the speed of the implemented circuit. A good measure of the speed is the maximum frequency at which the circuit can be clocked, referred to as *f_{max}*. This measure depends on the longest delay along any path between two registers clocked by the same clock. Quartus II performs a timing analysis to determine the expected performance of the circuit. It evaluates several parameters, which are listed in the Timing Analyzer section of the Compilation Report. Click on the small + symbol next to Timing Analyzer to expand this section of the report, as shown in Figure 18. Clicking on **Timing Analyzer Summary** displays the table in Figure 19. The last entry in the table shows that the maximum frequency for our circuit implemented on the specified chip is 260.89 MHz. You may get a different value of *f_{max}*, dependent on the specific version of Quartus II software installed on your computer. To see the paths in the circuit that limit the *f_{max}*, click on the Timing Analyzer item **Clock Setup: 'Clock'** in Figure 19 to obtain the display in Figure 20. This table shows that the critical path begins at the flip-flop *AddSubR* and ends at the flip-flop *Overflow*.

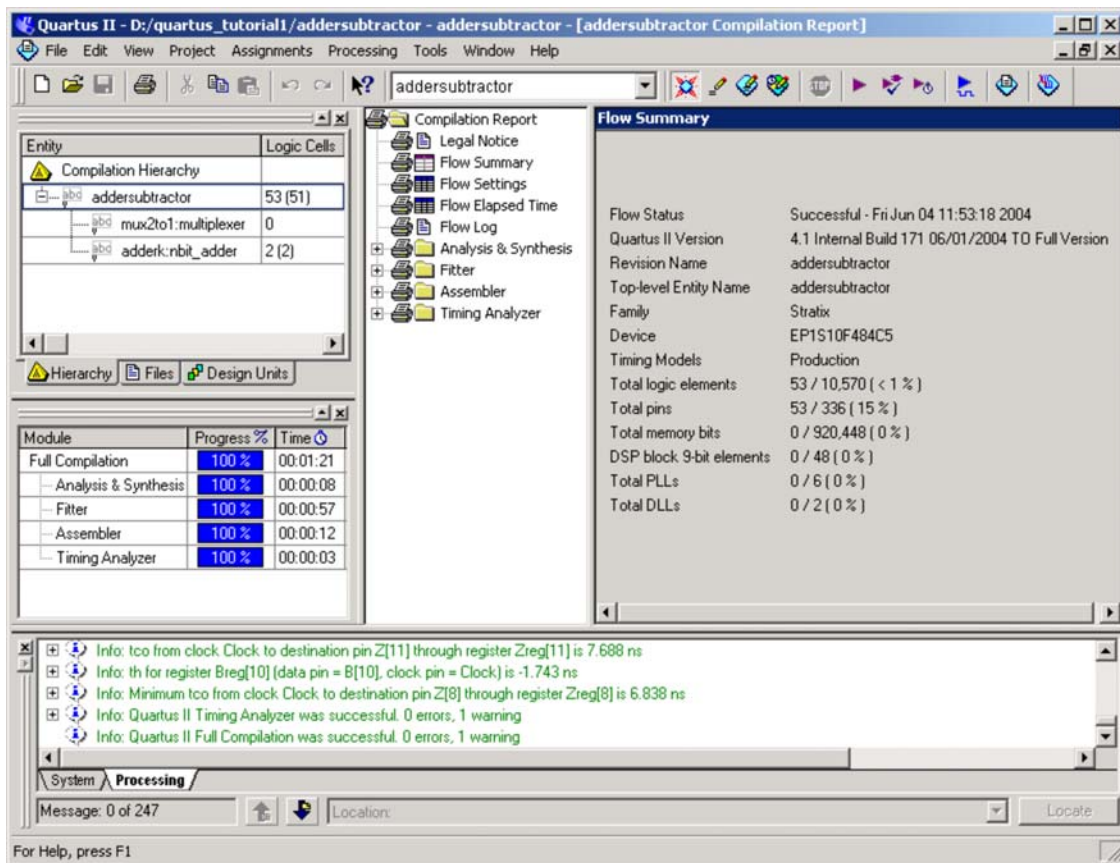


Figure 17. Display after a successful compilation.

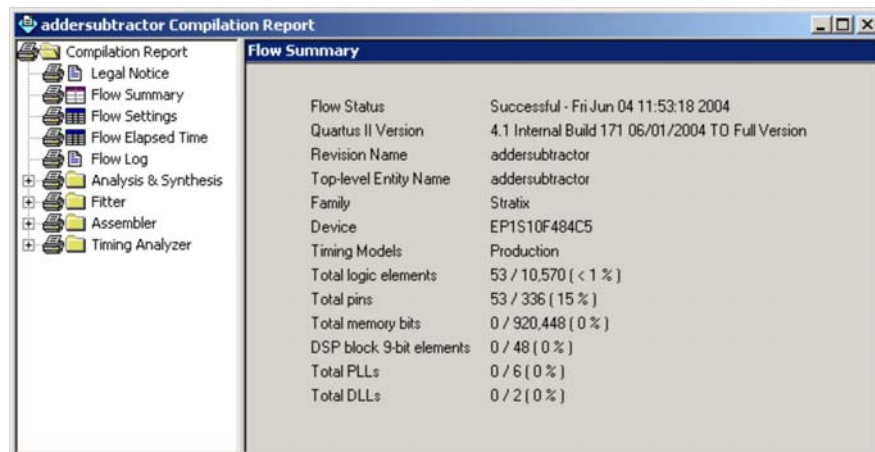


Figure 18. Compilation report.

addersubtractor Compilation Report									
<ul style="list-style-type: none"> Compilation Report Legal Notice Flow Summary Flow Settings Flow Elapsed Time Flow Log Analysis & Synthesis Fitter Assembler Timing Analyzer <ul style="list-style-type: none"> Timing Analyzer Settings Timing Analyzer Summary Clock Settings Summary Clock Setup: 'Clock' tsu tco th Minimum tco Timing Analyzer Messages 	Timing Analyzer Summary								
	Type	Slack	Required Time	Actual Time	From	To	From Clock	To Clock	Failed Paths
	1 Worst-case tsu	N/A	None	2.872 ns	B[14]	Breg[14]		Clock	0
	2 Worst-case tco	N/A	None	7.688 ns	Zreg[11]	Z[11]	Clock		0
	3 Worst-case th	N/A	None	-1.743 ns	B[10]	Breg[10]		Clock	0
	4 Worst-case Minimum tco	N/A	None	6.838 ns	Zreg[8]	Z[8]	Clock		0
	5 Clock Setup: 'Clock'	N/A	None	253.55 MHz (period = 3.944 ns)	Zreg[12]	Overflow~reg0	Clock	Clock	0
	6 Total number of failed paths								0

Figure 19. Summary of timing analysis.

Clock Setup: 'Clock'						
	Slack	Actual fmax (period)	From	To	From Clock	To Clock
1	N/A	253.55 MHz (period = 3.944 ns)	Zreg[12]	Overflow~reg0	Clock	Clock
2	N/A	256.08 MHz (period = 3.905 ns)	Zreg[8]	Overflow~reg0	Clock	Clock
3	N/A	258.53 MHz (period = 3.868 ns)	Zreg[13]	Overflow~reg0	Clock	Clock
4	N/A	260.62 MHz (period = 3.837 ns)	Zreg[9]	Overflow~reg0	Clock	Clock
5	N/A	261.03 MHz (period = 3.831 ns)	AddSubR	Overflow~reg0	Clock	Clock
6	N/A	264.06 MHz (period = 3.787 ns)	Zreg[10]	Overflow~reg0	Clock	Clock
7	N/A	264.62 MHz (period = 3.779 ns)	Self	Overflow~reg0	Clock	Clock
8	N/A	267.24 MHz (period = 3.742 ns)	Zreg[11]	Overflow~reg0	Clock	Clock
9	N/A	267.31 MHz (period = 3.741 ns)	Zreg[4]	Overflow~reg0	Clock	Clock
10	N/A	269.54 MHz (period = 3.710 ns)	Zreg[14]	Overflow~reg0	Clock	Clock
11	N/A	282.01 MHz (period = 3.546 ns)	Zreg[15]	Overflow~reg0	Clock	Clock
12	N/A	282.33 MHz (period = 3.542 ns)	Zreg[1]	Overflow~reg0	Clock	Clock
13	N/A	285.39 MHz (period = 3.504 ns)	Breg[0]	Overflow~reg0	Clock	Clock
14	N/A	287.94 MHz (period = 3.473 ns)	Zreg[2]	Overflow~reg0	Clock	Clock
15	N/A	291.72 MHz (period = 3.428 ns)	Zreg[12]	Zreg[13]	Clock	Clock

Figure 20. Critical paths.

The table in Figure 19 also shows the measurements of other timing parameters. While f_{max} is a function of the longest propagation delay between two registers in the circuit, it does not indicate the delays with which output signals appear at the pins of the chip. The time elapsed from an active edge of the clock signal at the clock source until a corresponding output signal is produced (from a flip-flop) at an output pin is denoted as the t_{co} parameter at that pin. In the worst case, the t_{co} in our circuit is 7.69 ns. Click on t_{co} in the Timing Analyzer section to view the table given in Figure 21. The first entry in the table shows that it takes 7.69 ns for a signal to propagate from bit 14 in register $Zreg$ to the output pin z_{14} . The other two parameters given in Figure 19 are setup time, tsu , and hold time, th ,

tco					
	Slack	Required tco	Actual tco	From	To
					From Clock
1	N/A	None	7.688 ns	Zreg[11]	Z[11] Clock
2	N/A	None	7.568 ns	Zreg[7]	Z[7] Clock
3	N/A	None	7.467 ns	Zreg[9]	Z[9] Clock
4	N/A	None	7.356 ns	Zreg[1]	Z[1] Clock
5	N/A	None	7.256 ns	Zreg[6]	Z[6] Clock
6	N/A	None	7.212 ns	Zreg[0]	Z[0] Clock
7	N/A	None	7.136 ns	Zreg[2]	Z[2] Clock
8	N/A	None	7.128 ns	Zreg[4]	Z[4] Clock
9	N/A	None	7.114 ns	Zreg[12]	Z[12] Clock
10	N/A	None	7.084 ns	Zreg[13]	Z[13] Clock
11	N/A	None	7.082 ns	Overflow~reg0	Overflow Clock
12	N/A	None	6.932 ns	Zreg[10]	Z[10] Clock
13	N/A	None	6.921 ns	Zreg[15]	Z[15] Clock
14	N/A	None	6.917 ns	Zreg[14]	Z[14] Clock
15	N/A	None	6.907 ns	Zreg[3]	Z[3] Clock
16	N/A	None	6.904 ns	Zreg[5]	Z[5] Clock
17	N/A	None	6.838 ns	Zreg[8]	Z[8] Clock

Figure 21. The *tco* delays.

An indication of where the circuit is implemented on the chip is available by selecting **Assignments | Timing Closure Floorplan**. Figure 22 depicts the result, highlighting in color the logic elements used to implement the circuit. To make the image appear as shown you may have to select **View | Field View**, so that the tool does not show the details of the chip resources. The floorplan view can be enlarged by maximizing the window and selecting **View | Fit in Window** (shortcut Ctrl-w), and it can be expanded to fill the screen by clicking the **Full Screen** icon in the toolbar.

A **Zoom Tool**, activated by the icon that looks like a magnifying glass, can be used to enlarge parts of the image even more. Figure 23 shows a zoomed-in view of the floorplan that highlights the implemented circuit. To see the details given in the figure you have to select the **Floorplan Editor** command **View | Interior Cells**. By positioning the cursor on any logic element the designer can see what part of the circuit is implemented in this resource. The floorplan tool has several icons that can be used to view aspects such as fanin and fanout of nodes, connecting paths between nodes, and so on. For more information on using this tool refer to the on-line help, by selecting **Help | Contents | Viewing the Fit in the Floorplan Editor**.

The detailed implementation of the circuit in the form of logic equations is also included in the compilation report. It can be viewed by selecting **Fitter | Equations**. These equations do not necessarily correspond directly to any logic expressions that may have been given in the Verilog design file, because the synthesized circuit is implemented on the FPGA chip in logic elements that constitute lookup (truth) tables (LUTs).

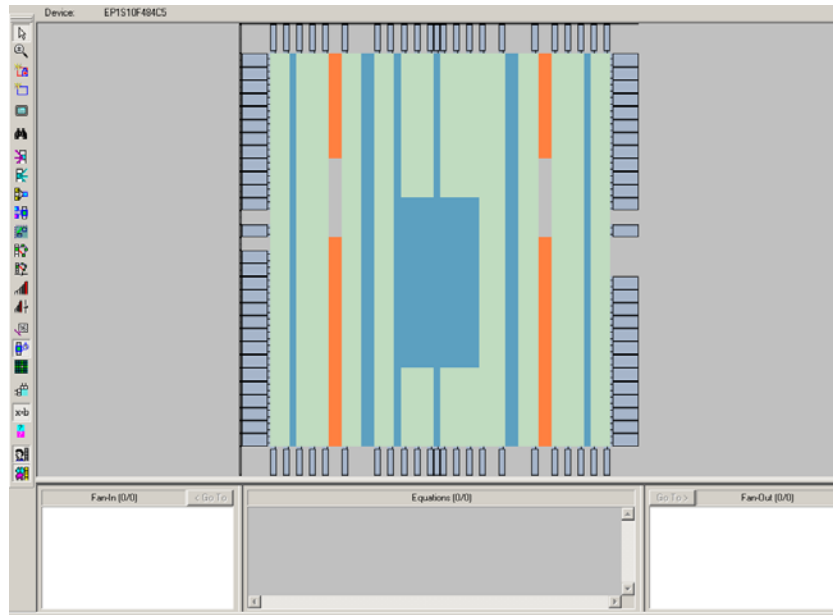


Figure 22. View of the floorplan.

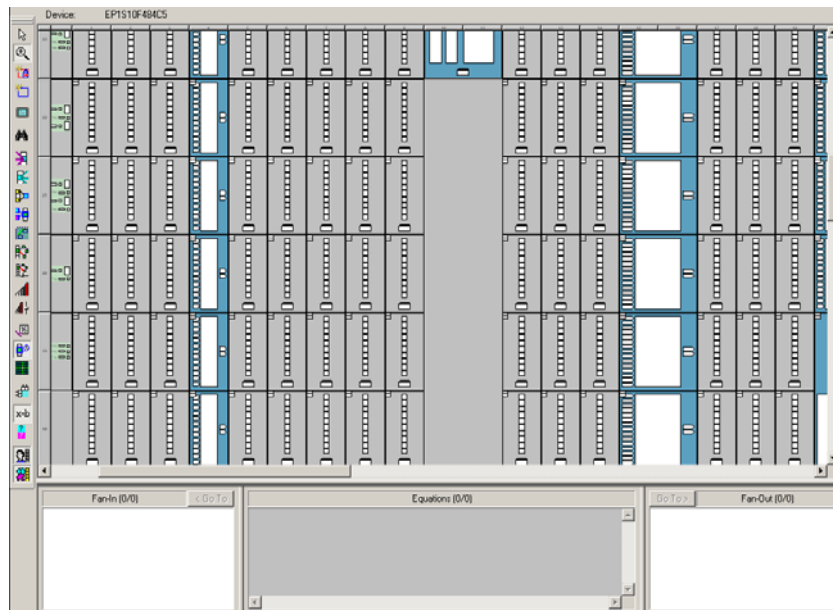


Figure 23. A portion of the expanded view.

4.1 Errors

Quartus II displays messages produced during compilation in the Messages utility window. If the Verilog design file is correct, one of the messages will state that the compilation was successful and that there are no errors.

If the Compiler does not report zero errors, then there is at least one mistake in the Verilog code. In this case a message corresponding to each error found will be displayed in the Messages window. Double-clicking on an error message will highlight the offending statement in the Verilog code in the Text Editor window. Similarly, the Compiler may display some warning messages. Their details can be explored in the same way as in the case of error messages. The user can obtain more information about a specific error or warning message by selecting the message and pressing the F1 function key.

To see the effect of an error, open the file *addersubtractor.v*. Line 21 has the statement

```
assign H = Breg ^ {n{AddSubR}};
```

Replace H with J in this statement, illustrating a typographical error that is easily made because H and J are adjacent on the keyboard. Compile the erroneous design file. Quartus II will display a pop-up box indicating that the compilation was not successful. Acknowledge it by clicking OK. The compilation report summary, given in Figure 24, now confirms the failed result. Click on **Analysis & Synthesis Messages** in this window to have all messages displayed as shown in Figure 25. Double-click on the first error message, which states that variable J is not declared. Quartus II responds by opening the *addersubtractor.v* file and highlighting the erroneous statement as shown in Figure 26. Correct the error and recompile the design.

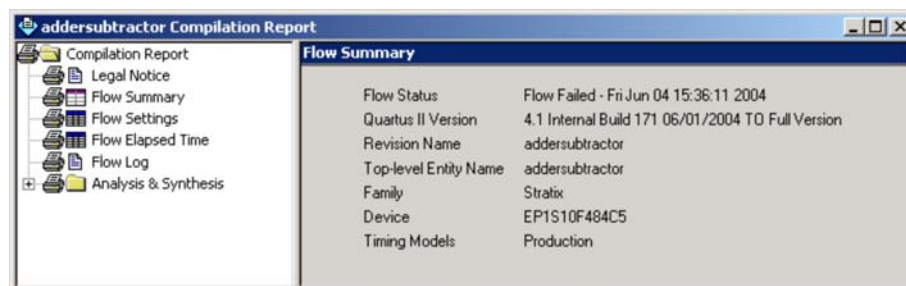


Figure 24. Compilation report for the failed design.

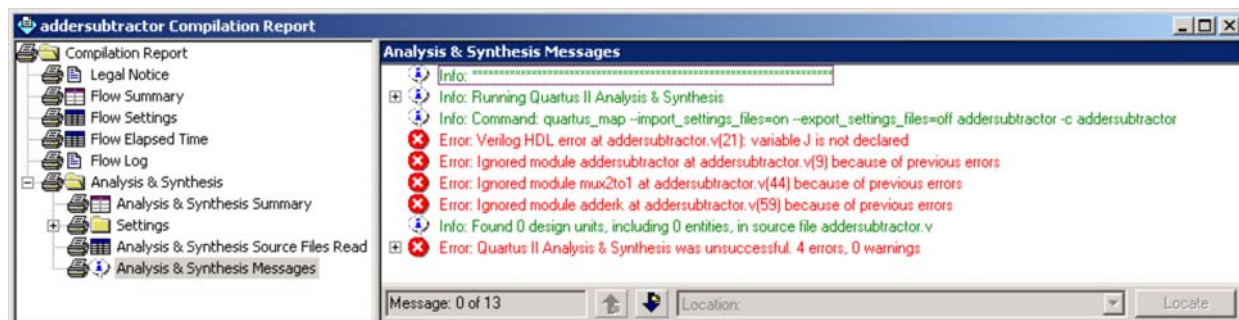


Figure 25. Error messages.

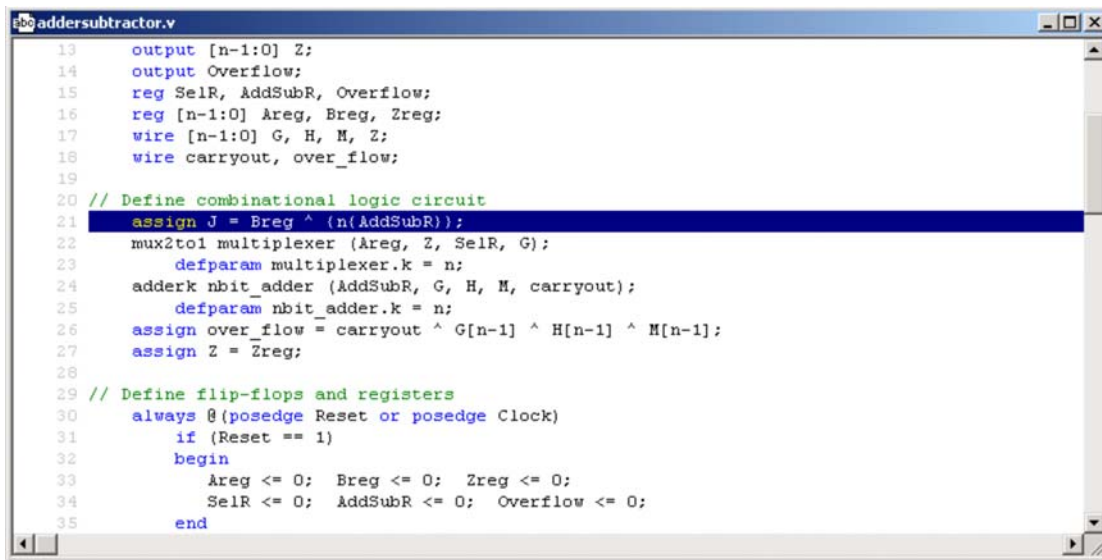


Figure 26. Identifying the erroneous statement.

5 Simulating the Designed Circuit

Quartus II includes a simulation tool that can be used to simulate the behavior of the designed circuit. Before the circuit can be simulated, it is necessary to create the desired waveforms, called *test vectors*, to represent the input signals. We will use the Quartus II Waveform Editor to draw the test vectors.

5.1 Using the Waveform Editor

Open the Waveform Editor window by selecting **File | New**, which gives the window shown in Figure 12. Click on the **Other Files** tab to reach the window displayed in Figure 27. Choose **Vector Waveform File** and click **OK**. (A shortcut for activating the Waveform Editor is available using the icon in the toolbar that looks like a square-wave signal.)

The Waveform Editor window is depicted in Figure 28. Save the file under the name *addersubtractor.vwf*; note that this changes the name in the displayed window. In this figure, we have set the desired simulation to run from 0 to 180 ns by selecting **Edit | End Time** and entering 180 ns in the dialog box that pops up. Selecting **View | Fit in Window** displays the entire simulation range of 0 to 180 ns in the window, as shown. Resize the window to its maximum size.

Next, we want to include the input and output nodes of the circuit to be simulated. Click **Edit | Insert Node or Bus** to open the window in Figure 29. It is possible to type the full hierarchical name of a signal (pin) into the Name box, but it is easier to click on the button labeled **Node Finder** to open the window in Figure 30. The Node Finder utility has a filter used to indicate what type nodes are to be found. Since we are interested in input and output pins, set the filter to **Pins: all**. Click the **List** button to find the input and output nodes as indicated on the left side of the figure. Observe that the input and output signals *A*, *B*, and *Z* can be selected either as individual nodes (denoted by bracketed subscripts) or as 16-bit vectors, which is a more convenient form.

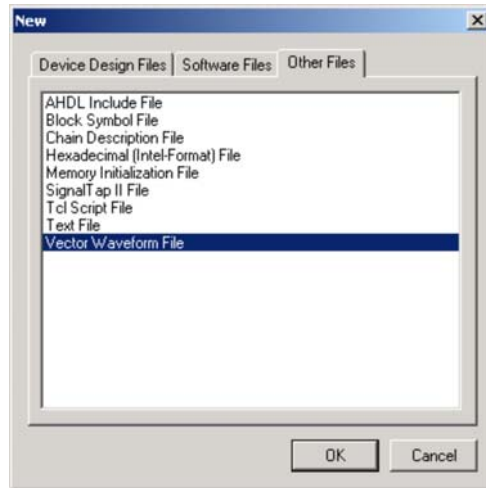


Figure 27. Choose to prepare a test-vector file.

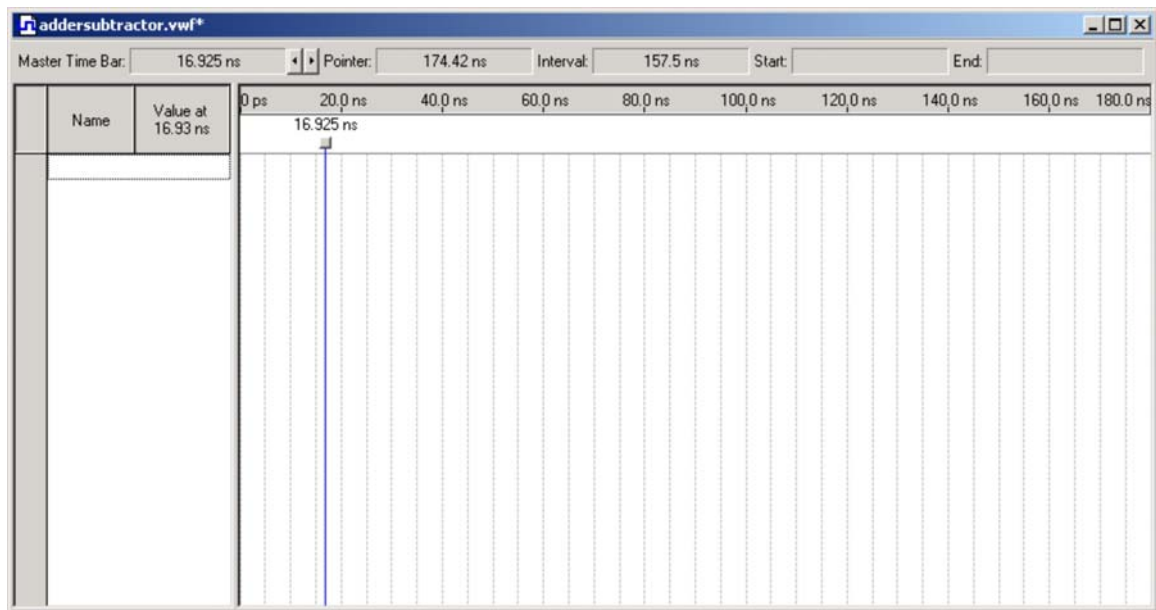


Figure 28. The Waveform Editor window.

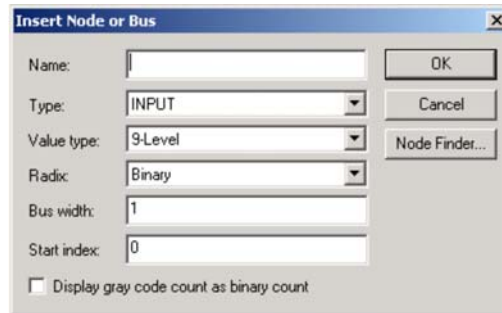


Figure 29. The Insert Node or Bus dialogue.

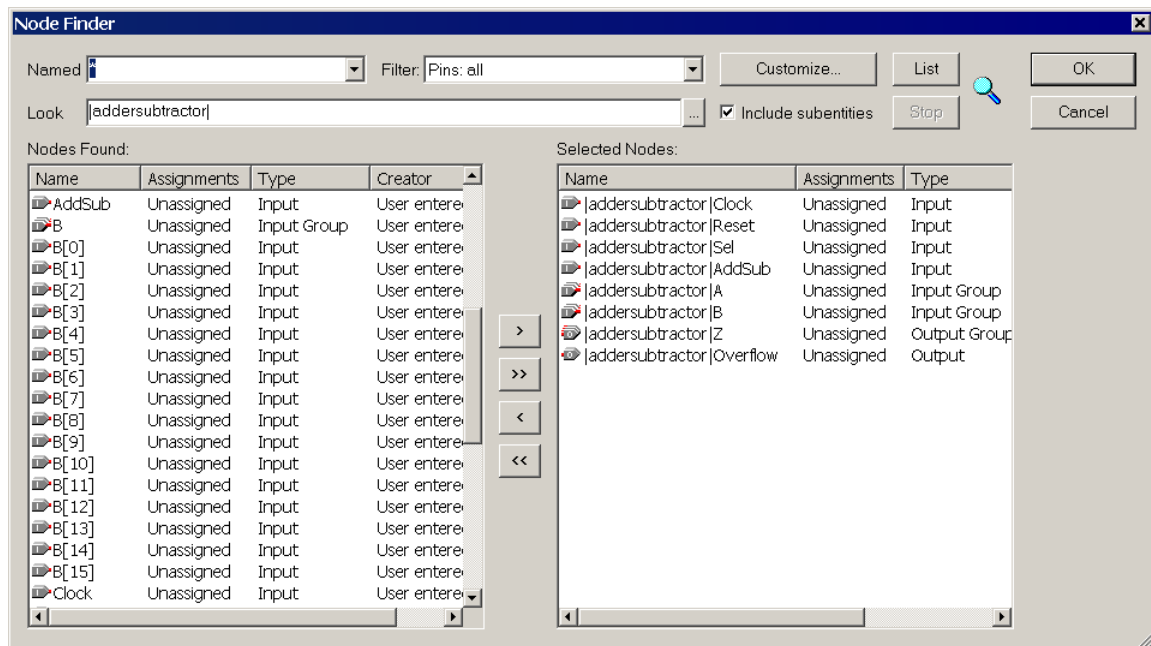


Figure 30. Selecting nodes to insert into the Waveform Editor.

Use the scroll bar inside the Nodes Found box in Figure 30 to find the *Clock* signal. Click on this signal and then click the > sign to add it to the Selected Nodes box on the right side of the figure. Do the same for *Reset*, *Sel*, and *AddSub*. Then choose vectors *A*, *B* and *Z*, as well as the output *Overflow*, in the same way. (Several nodes can be selected simultaneously in a standard Windows manner.) Click OK to close the Node Finder window, and then click OK in the window of Figure 29. This leaves a fully displayed Waveform Editor window, as shown in Figure 31. If you did not select the nodes in the same order as displayed in Figure 31, it is possible to rearrange them. To move a waveform up or down in the Waveform Editor window, click on the node name (in the Name column) and release the mouse button. The waveform is now highlighted to show the selection. Click again on the waveform and drag it up or down in the Waveform Editor.

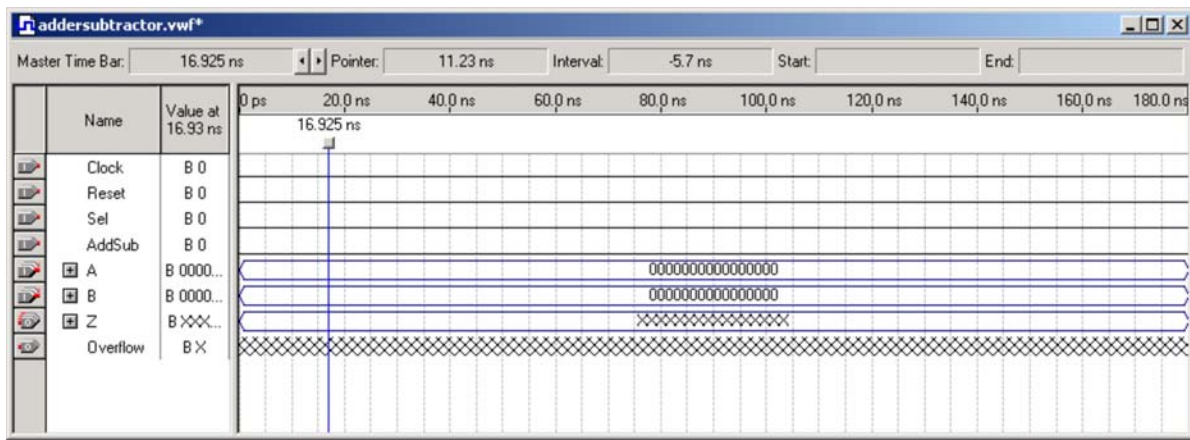


Figure 31. The nodes needed for simulation.

We will now specify the logic values to be used for the input signals during simulation. The logic values at the outputs *Z* and *Overflow* will be generated automatically by the simulator. To make it easy to draw the desired waveforms, the Waveform Editor displays (by default) vertical guidelines and provides a drawing feature that snaps on these lines (which can otherwise be invoked by choosing **View | Snap to Grid**). Observe also a solid vertical line, which can be moved by pointing to its top and dragging it horizontally. This “reference line” is used in analyzing the timing of a circuit, as described later; move it to the *time* = 0 position. The waveforms can be drawn using the Selection Tool, which is activated by selecting the icon in the toolbar that looks like a big arrowhead, or the Waveform Editing Tool, which looks like a two-headed arrow.

To simulate the behavior of a large circuit, it is necessary to apply a sufficient number of input valuations and observe the expected values of the outputs. The number of possible input valuations may be huge, so in practice we choose a relatively small (but representative) sample of these input valuations. We will choose a very small set of input test vectors, which is not sufficient to simulate the circuit properly but is adequate for tutorial purposes. We will use eight 20-ns time intervals to apply the test vectors as shown in Figure 32. The values of signals *Reset*, *Sel*, *AddSub*, *A* and *B* are applied at the input pins as indicated in the figure. The value of *Z* at time t_i is a function of the inputs at time t_{i-1} . When *Sel* = 1, the accumulator feedback loop is activated so that the current value of *Z* (rather than *A*) is used to compute the new value of *Z*. The effect of the test vectors in Figure 32 is to perform the following computation:

$$\begin{aligned}
 t_0 &: \text{Reset} \\
 t_1 &: Z(t_1) = 0 \\
 t_2 &: Z(t_2) = A(t_1) + B(t_1) = 54 + 1850 = 1904 \\
 t_3 &: Z(t_3) = A(t_2) - B(t_2) = 132 - 63 = 69 \\
 t_4 &: Z(t_4) = A(t_3) + B(t_3) = 0 + 0 = 0 \\
 t_5 &: Z(t_5) = A(t_4) - B(t_4) = 750 - 120 = 630 \\
 t_6 &: Z(t_6) = Z(t_5) + B(t_5) = 630 + 7000 = 7630 \\
 t_7 &: Z(t_7) = Z(t_6) + B(t_6) = 7630 + 30000 = 37630 \text{ (overflow)}
 \end{aligned}$$

Initially, the circuit is reset asynchronously. Then for two clock cycles the output *Z* is first the sum and then the difference of the values of *A* and *B* at that time. This is followed by setting both *A* and *B* to zero to clear the contents of register *Z*. Then, the accumulator feedback path is tested in the next three clock cycles by performing the computation

$$Z = A(t_4) - B(t_4) + B(t_5) + B(t_6)$$

using the values of *A* and *B* shown above.

Time	<i>Reset</i>	<i>Sel</i>	<i>AddSub</i>	<i>A</i>	<i>B</i>	<i>Z</i>
t_0	1	0	0	0	0	0
t_1	0	0	0	54	1850	0
t_2	0	0	1	132	63	1904
t_3	0	0	0	0	0	69
t_4	0	0	1	750	120	0
t_5	0	1	0	0	7000	630
t_6	0	1	0	0	30000	7630
t_7	0	1	0	0	0	37630

Figure 32. The required testing behavior.

We can generate the desired input waveforms as follows. Click on the waveform name for the *Clock* node. Once a waveform is selected, the editing commands in the Waveform Editor can be used to draw the desired waveforms. Commands are available for defining the clock, or setting the selected signal to 0, 1, unknown (X), high impedance (Z), don't care (DC), and inverting its existing value (INV). Each command can be activated by using the **Edit | Value** command, or via the toolbar for the Waveform Editor. The Edit menu can also be opened by right-clicking on a waveform name.

With the *Clock* signal highlighted, click on the **Overwrite Clock** icon in the toolbar (the icon depicts a clock). This leads to the pop-up window in Figure 33. Enter the clock period value of 20 ns, make sure that the phase is 0 and the duty cycle is 50 percent, and click **OK**. The desired clock signal is now displayed in the Waveform window.

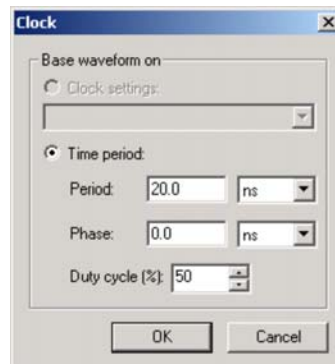


Figure 33. Definition of the clock period, phase and duty cycle.

We will assume, for simplicity of timing, that the input signals change coincident with the negative edges of the clock. To reset the circuit, set $Reset = 1$ in the time interval 0 to 20 ns. Do this by pressing the mouse at the start of the interval and dragging it to its end, which highlights the selected interval, and choosing the logic value 1 in the toolbar. Make $Sel = 1$ from 100 to 160 ns, and $AddSub = 1$ in periods 40 to 60 ns and 80 to 100 ns. This should produce the image in Figure 34.

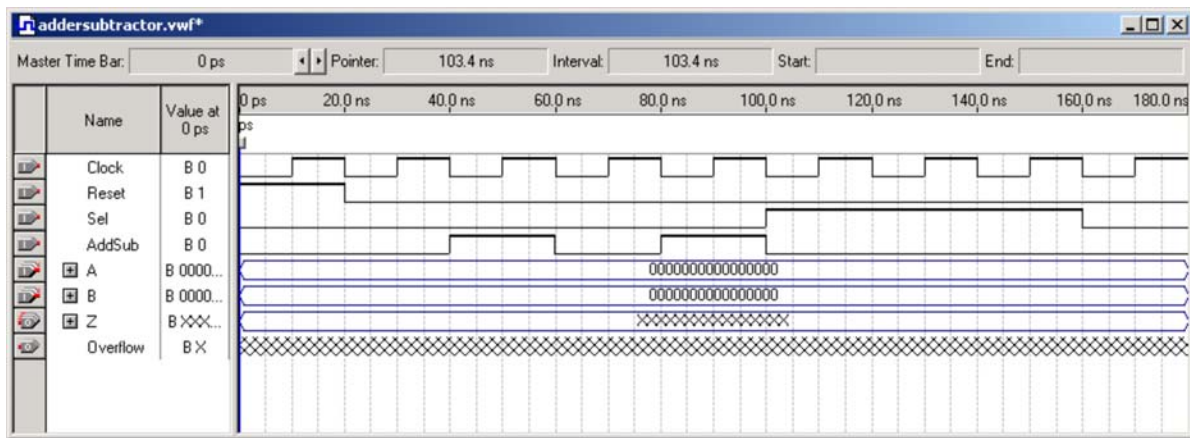


Figure 34. Setting of test values for the control signals.

Vectors *A*, *B*, and *Z* 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 signed decimal numbers, so right-click on *A* and select **Properties** in the pop-up box to get to the window displayed in Figure 35. Choose signed decimal as the radix, make sure that the bus width is 16 bits, and click OK. In the same manner, declare that *B* and *Z* should be treated as signed decimal numbers.

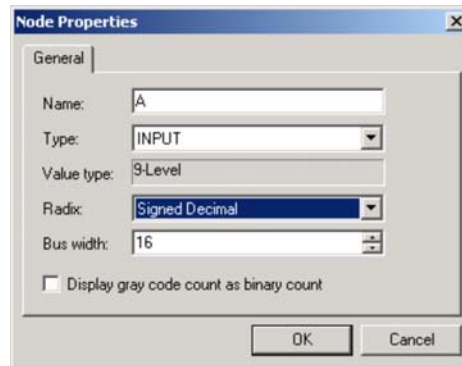


Figure 35. Definition of node properties.

The default value of *A* is 0. To assign specific values in various intervals proceed as follows. Select (highlight) the interval from 20 to 40 ns and press the **Arbitrary Value** icon in the toolbar (it is labeled by a question mark), to bring up the pop-up window in Figure 36. Enter the value 54 and click OK. Similarly, for the subsequent 20-ns intervals set *A* to the values 132, 0, 750, and then 0 to the end. Set the corresponding values of *B* to 1850, 63, 0, 120, 7000, 30000, and 0, to generate the waveforms depicted in Figure 37. Observe that the outputs *Z* and *Overflow* are displayed as having unknown values at this time, which is indicated by a hashed pattern; their values will be determined during simulation. Save the file.

Another convenient mechanism for changing the input waveforms is provided by the Waveform Editing tool. The icon for the tool is in the toolbar; it looks like two arrows pointing left and right. When the mouse is dragged over some time interval in which the waveform is 0 (1), the waveform will be changed to 1 (0). Experiment with this feature on signal *AddSub*.

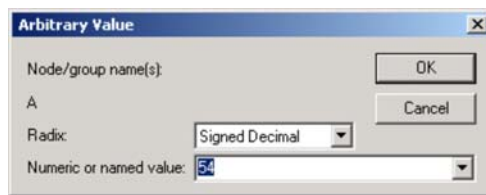


Figure 36. Specifying a value for a multibit signal.

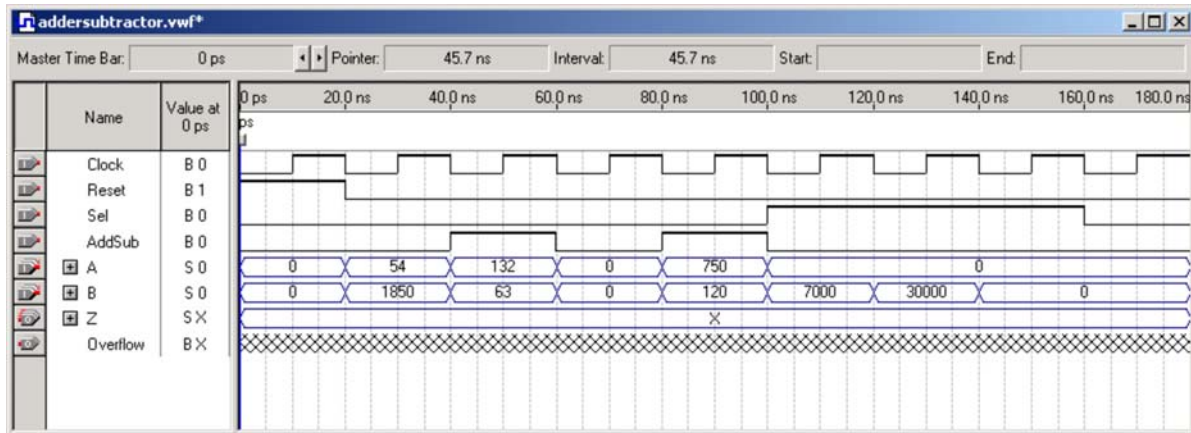


Figure 37. The specified input test vectors.

5.2 Performing the Simulation

A designed circuit can be simulated in two ways. The simplest way is to assume that logic elements and inter-connection wires are perfect, thus causing no delay in propagation of signals through the circuit. This is called *functional simulation*. A more complex alternative is to take all propagation delays into account, which leads to *timing simulation*. Typically, functional simulation is used to verify the functional correctness of a circuit as it is being designed. This takes much less time, because the simulation can be performed simply by using the logic expressions that define the circuit.

To perform the functional simulation, select **Assignments | Settings** to open the Settings window shown in Figure 15. On the left side of this window click on **Simulator** to display the window in Figure 38 and choose **Functional** as the simulation mode. The Quartus II simulator takes the inputs and generates the outputs defined in the *addersubtractor.vwf* file. Before running the functional simulation it is necessary to create the required netlist, which is done by selecting **Processing | Generate Functional Simulation Netlist**. A simulation run is started by **Processing | Start Simulation**, or by using the icon in the toolbar that looks like a blue triangle with a square wave below it. At the end of the simulation, Quartus II indicates its successful completion and displays a Simulation Report illustrated in Figure 39. As seen in the figure, the Simulator creates waveforms for the outputs *Z* and *Overflow*. As expected, the values of *Z* indicate the correct sum or difference of the applied inputs one clock cycle later because of the registers in the circuit. Note that the last value of *Z* is incorrect because the expected sum of 37630 is too big to be represented as a signed number in 16 bits, which is indicated by the *Overflow* signal being set to 1.

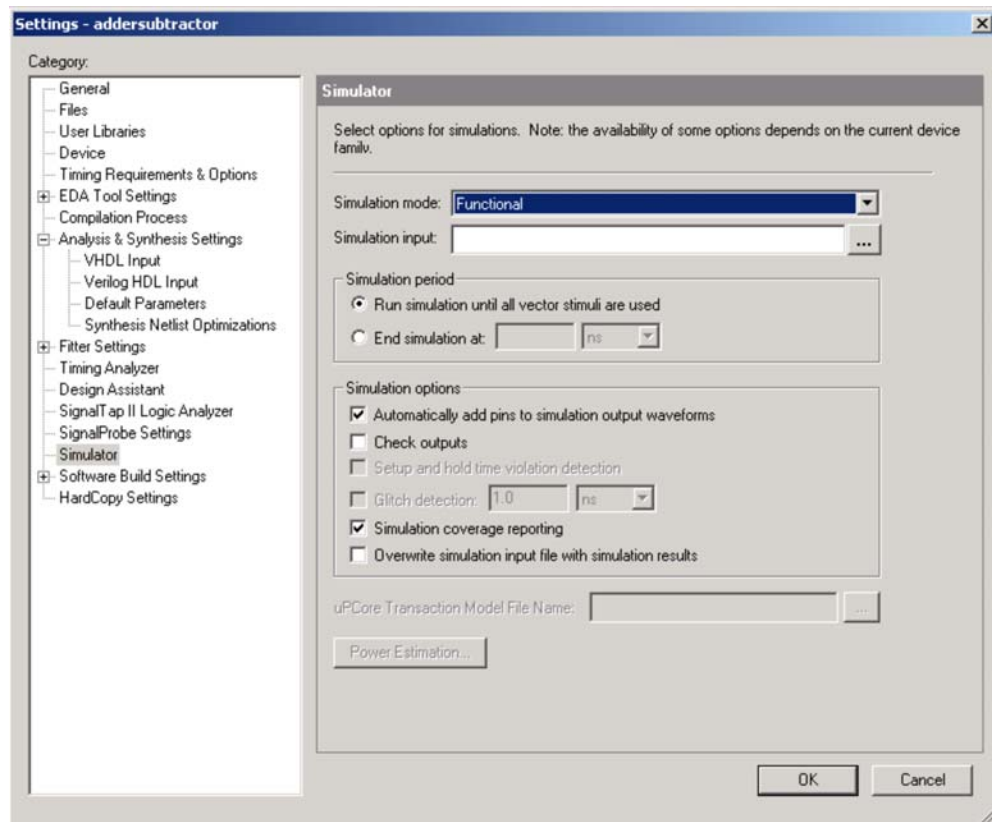


Figure 38. Specifying the simulation mode.

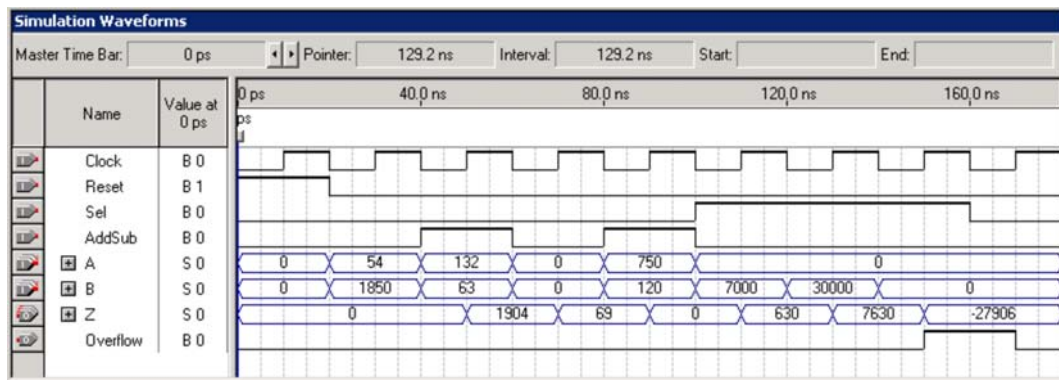


Figure 39. The result of functional simulation.

In this simulation, we considered only the input and output signals, which appear on the pins of the FPGA chip. It is also possible to look at the behavior of internal signals. For example, let us consider the registered signals *SelR*, *AddSubR*, *Areg*, *Breg*, and *Zreg*. Open the *addersubtractor.vwf* file and activate the Node Finder window, as done for Figure 30. The filter in Figure 30 specified Pins: all. There are several other choices. To find the registered signals, set the filter to Registers: post-fitting and press List. Figure 40 shows the result. Select the signals *SelR*, *AddSubR*, *Areg*, *Breg*, and *Zreg* for inclusion in the *addersubtractor.vwf* file, and specify that *Areg*, *Breg*, and *Zreg* have to be displayed as signed decimal numbers, thus obtaining the display in Figure 41. Save the file and simulate the circuit using these waveforms, which should produce the result shown in Figure 42.

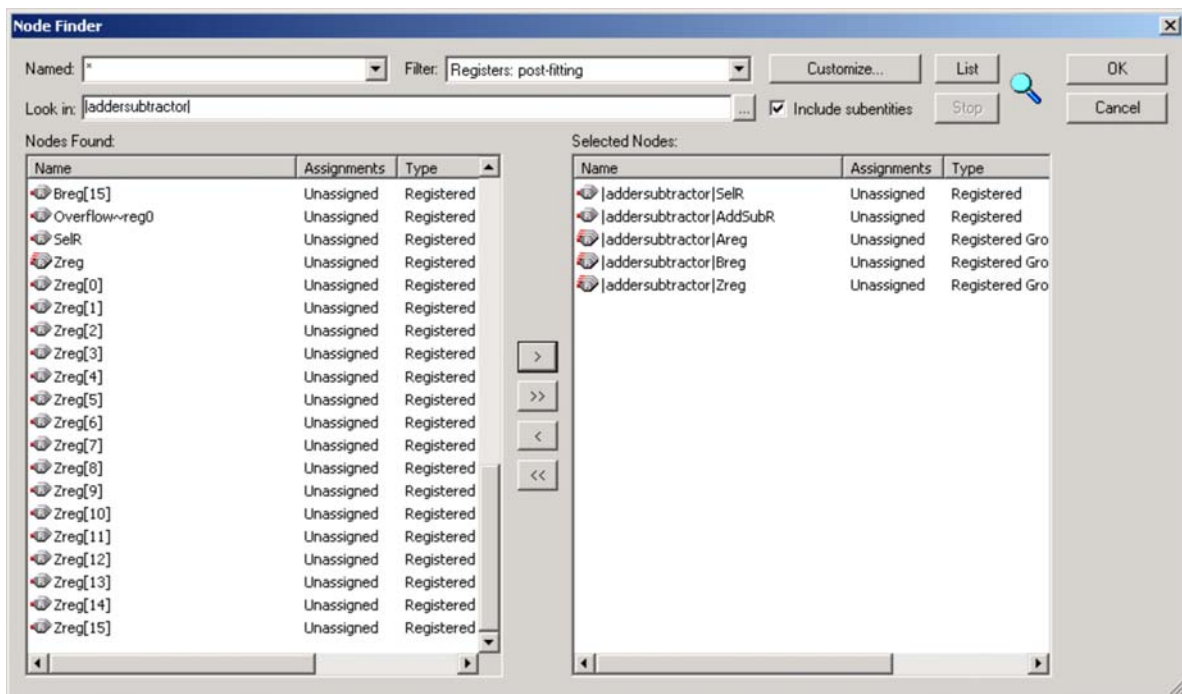


Figure 40. Finding the registered signals.



Figure 41. Inclusion of registered signals in the test.

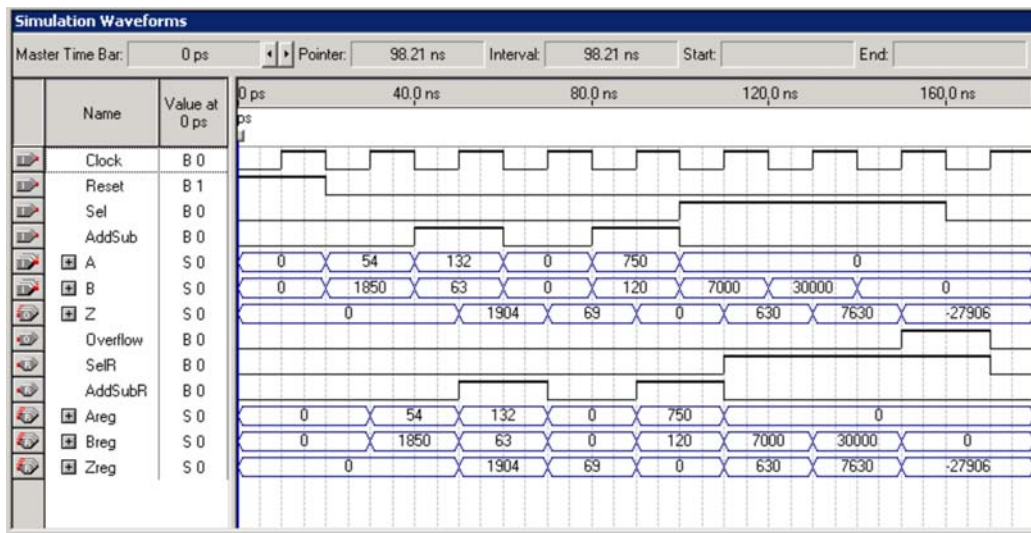


Figure 42. The result of new simulation.

Having ascertained that the designed circuit is functionally correct, we should now perform the timing simulation to see how well it performs in terms of speed. Select **Assignments | Settings | Simulator** to get to the window in Figure 38, choose **Timing** as the simulation mode, and click **OK**. Run the simulator, which should produce the waveforms in Figure 43. Observe that there are delays in loading the various registers as well as longer delays in producing valid signals on the output pins. As an aid in seeing the actual values of the delays, we can use the reference line. Point to the small square handle at the top of the reference line and drag it to the rising edge of the first *AddSubR* pulse, at which time the registers are also loaded, as indicated in the figure. (To make it possible to move the reference line to any point in the waveform display, you may have to turn off the feature **View | Snap on Grid**.) This operation places the reference line at about the 52.8 ns point, which indicates that it takes 2.8 ns to load the registers after the rising edge of the clock (which occurs at 50 ns). The output *Z* attains its correct value some time after this value has been loaded into *Zreg*. To determine this propagation delay to the output pins, drag the reference line to the point where *Z* becomes valid. This can be done more accurately by enlarging the displayed simulation waveforms by using the **Zoom Tool**. Left-click on the display to enlarge it and right-click to reduce it. Enlarge the display so that it looks like the image in Figure 44. (After enlarging the image, click on the **Selection Tool** icon.) Position the reference line where *Z* changes to 1904, which occurs at about 57.2 ns. The display indicates that the propagation delay from register *Zreg* to the output pins *Z* is $57.2 - 52.8 = 4.4$ ns. It is useful to note that even before we performed this simulation, Quartus II evaluated various delays in the implemented circuit and reported them in the **Compilation Report**. As already seen in Figure 19, the worst case *t_{co}* (Clock to Output Delay) for the *Z* output (pin *z₃*) was estimated as 7.69 ns; this delay can be found by zooming into the simulation results at the point where *Z* changes to the value 7630.

In this discussion, we have used the numbers obtained during our simulation run. The user is likely to obtain somewhat different numbers, depending on the version of Quartus II that is used.

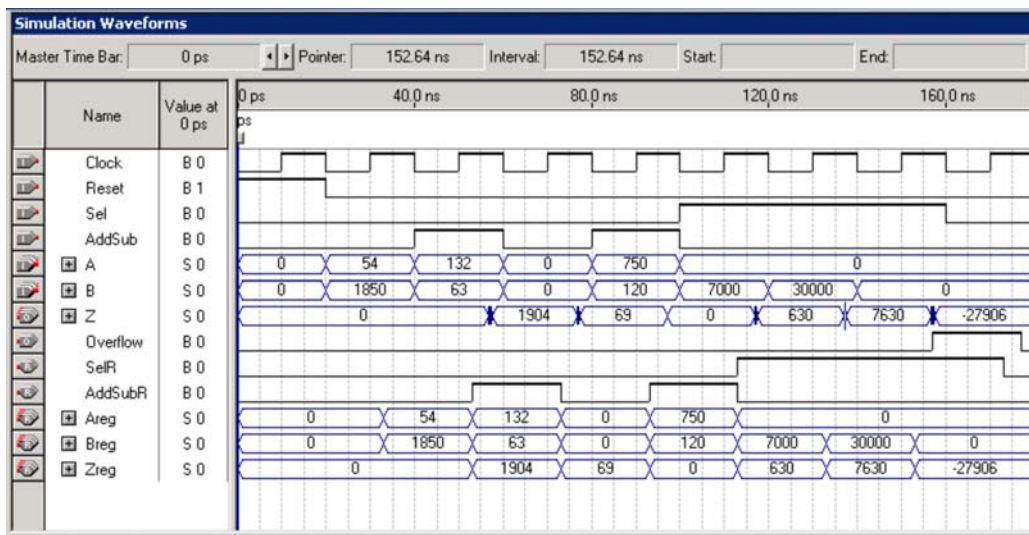


Figure 43. The result of timing simulation.

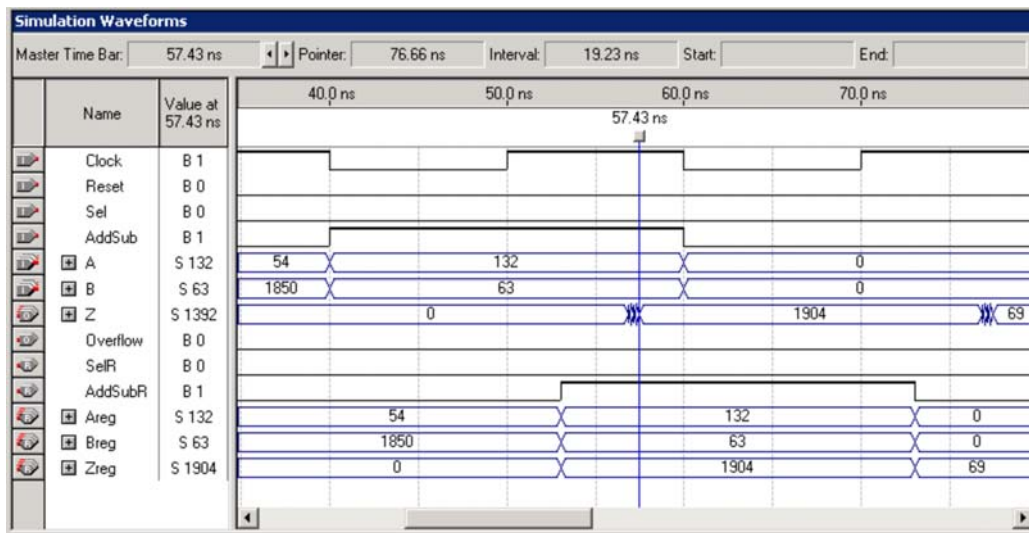


Figure 44. An enlarged image of the simulated waveforms.

6 Using Library Modules

Practical designs often include commonly used circuit blocks such as adders, subtractors, multipliers, decoders, counters, and shifters. Altera provides highly efficient implementations of such blocks in the form of library modules that can be instantiated in Verilog designs. The compiler may recognize that a standard function specified in Verilog code can be realized using a library module, in which case it will automatically *infer* this module. However, these modules can also be instantiated in the design explicitly by the user.

Quartus II includes a *library of parameterized modules (LPM)*. The modules are general in structure and they are tailored to a specific application by specifying the values of general parameters. Select **Help | Megafunc-tions/LPM** to see a listing of the available LPMs. One of them is an adder/subtractor module called *lpm_add_sub megafunction*. Select this module to see its description. The module has a number of inputs and outputs, some

of which may be omitted in a given application. Several parameters can be defined to specify a particular mode of operation. For example, the number of bits in the operands is specified in the parameter LPM_WIDTH. The LPM_REPRESENTATION parameter specifies whether the operands are to be interpreted as signed or unsigned numbers, and so on. Templates on how an LPM can be instantiated in a hardware description language are given in the description of the module. Using these templates is somewhat cumbersome, so Quartus II provides a wizard that makes the instantiation of LPMs easy.

We will use the *lpm_add_sub* module to simplify our previously designed adder/subtractor circuit presented in Figures 11 and 12. The augmented circuit is given in Figure 45. The *lpm_add_sub* module, instantiated under the name *megaddsub*, replaces the adder circuit as well as the XOR gates that provide the input *H* to the adder. Since arithmetic overflow is one of the outputs that the LPM provides, it is not necessary to generate this output with a separate XOR gate.

To implement the new adder/subtractor circuit, create a new directory *quartus_tutorial2*. Use the New Project Wizard to create a new project using the procedure described in Section 2. Call the project *addersubtractor2*, as indicated in Figure 46. Choose the same EP1S10F484C5 device used in the project *addersubtractor* to allow a direct comparison of implemented designs.

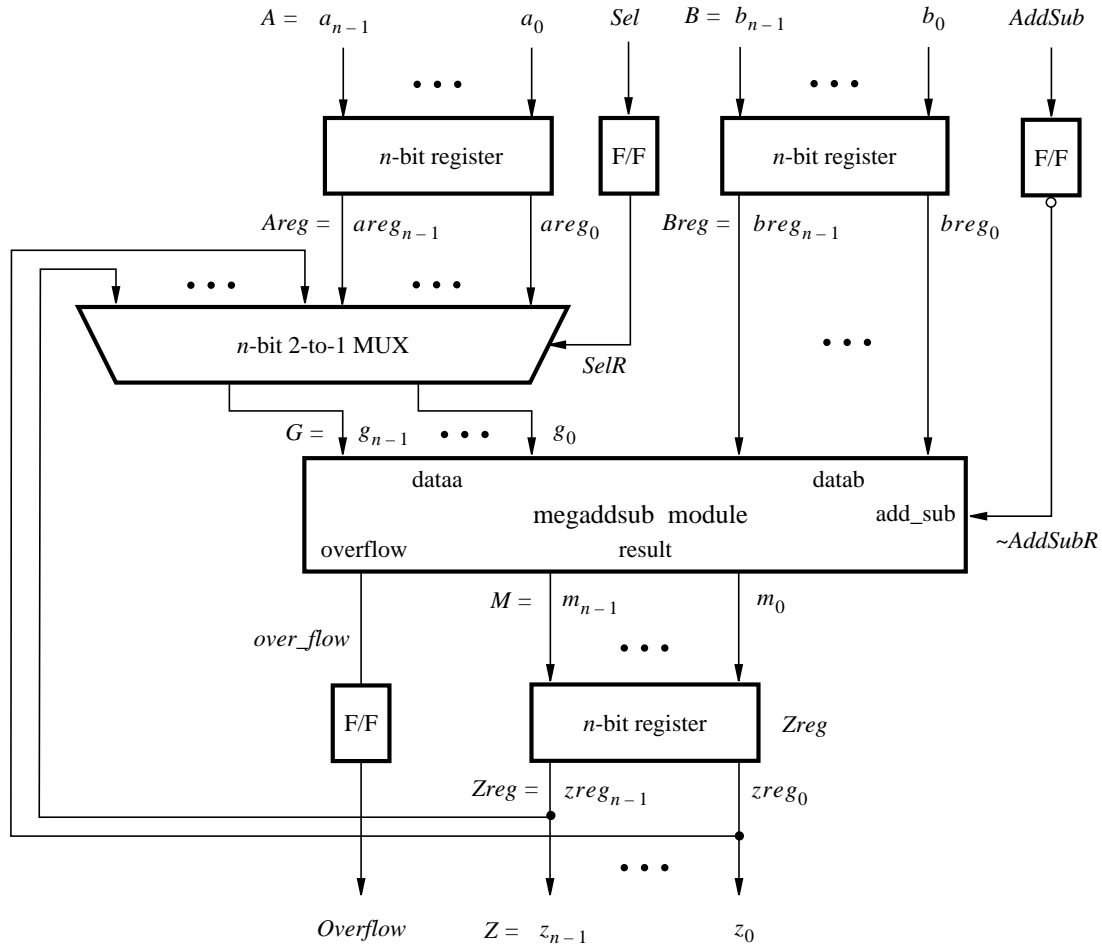


Figure 45. The augmented adder/subtractor circuit.

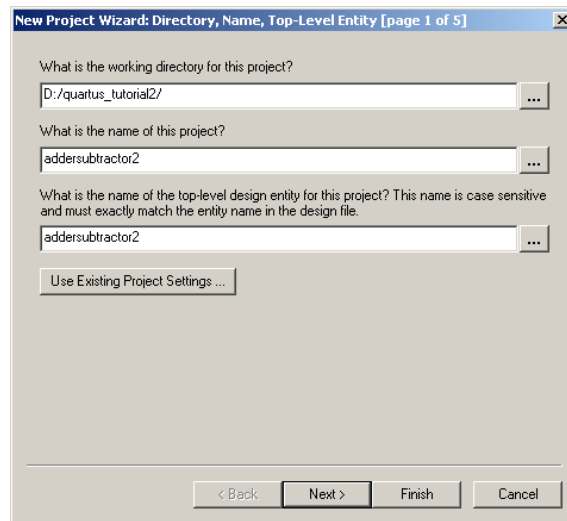


Figure 46. Creation of a new project.

The new design will include the desired LPM subcircuit specified as a Verilog module that will be instantiated in the top-level Verilog design module. The Verilog module for the LPM subcircuit is generated by using a wizard as follows. Select **Tools | MegaWizard Plug-In Manager**, which leads to a sequence of seven pop-up boxes in which the user can specify the details of the desired LPM. In the box shown in Figure 47 indicate **Create a new megafunction variation** and click **Next**. The box in Figure 48 provides a list of the available LPMs. Expand the “arithmetic” sublist and select **LPM_ADD_SUB**. Choose **Verilog HDL** as the type of output file that should be created. The output file must be given a name; choose the name *megaddsub.v* and indicate that the file should be placed in the directory *quartus_tutorial2* as shown in the figure. Press **Next**. In the box in Figure 49 specify that the width of the data inputs is 16 bits. Also, specify the operating mode in which one of the ports allows performing both addition and subtraction of the input operand, under the control of the *add_sub* input. A symbol for the resulting LPM is shown in the top left corner. Note that if *add_sub* = 1 then $result = A + B$; otherwise, $result = A - B$. This interpretation of the control input and the operation performed is different from our original design in Figures 11 and 12, which we will have to account for in the modified design. Click **Next** to reach the box in Figure 50. Specify that the values of both inputs may vary and click **Next**. The box in Figure 51 allows the designer to indicate optional inputs and outputs that may be specified. Since we need the overflow signal, make the **Create an overflow output** choice and press **Next**. In the box in Figure 52 say **No** to the pipelining option and click **Next**. Figure 53 gives a summary which shows the files that the wizard created. We will use the file *megaddsub.v* in our modified design. Figure 54 depicts the Verilog code in this file; note that we have not shown the comments in order to keep the figure small.

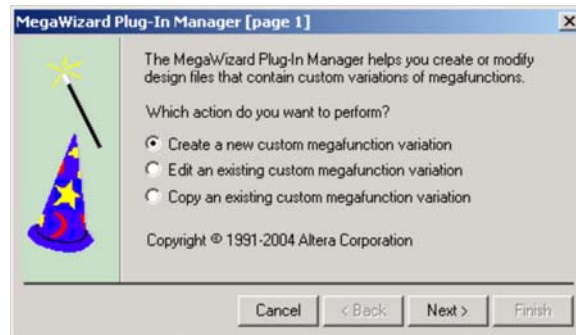


Figure 47. Choose to define an LPM.

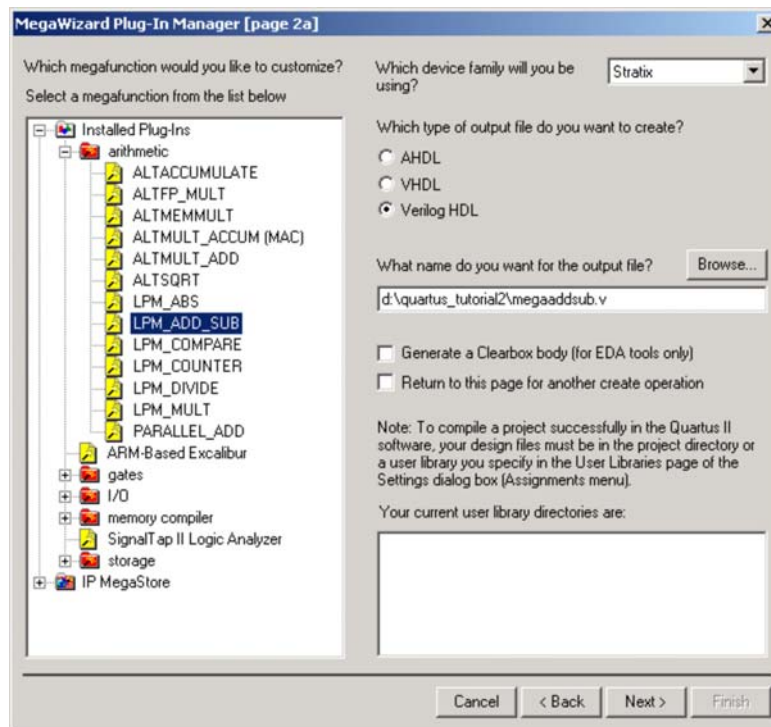


Figure 48. Choose an LPM from the available library.

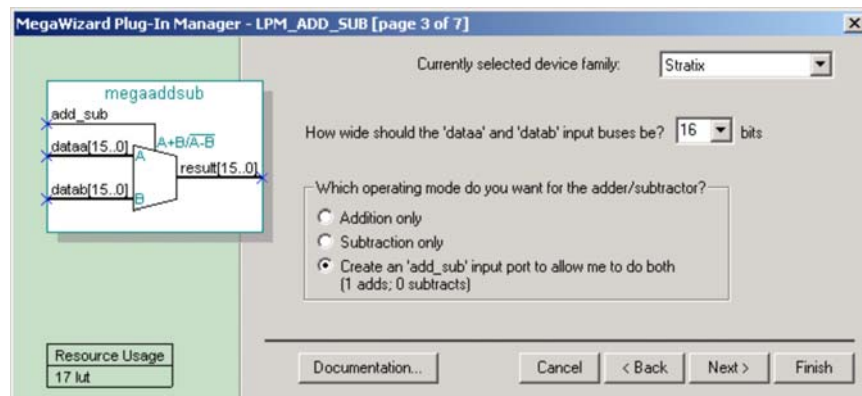


Figure 49. Specify the size of data inputs.

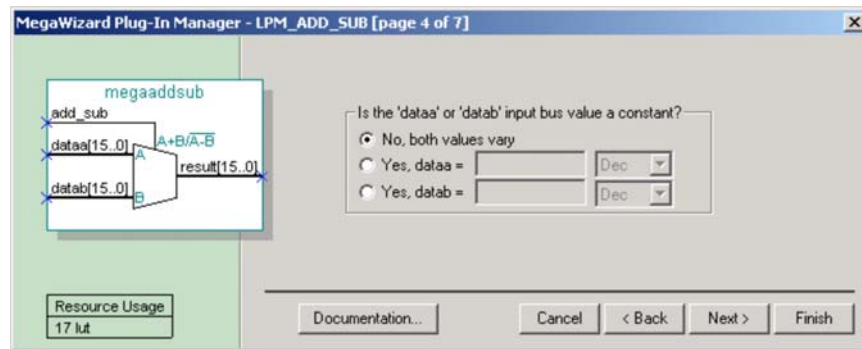


Figure 50. Further specification of inputs.

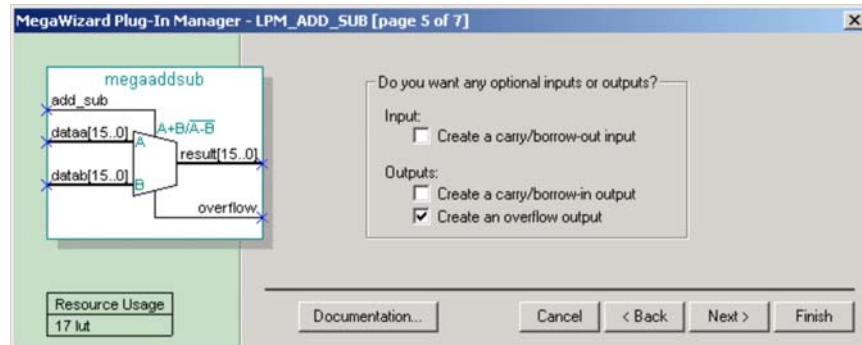


Figure 51. Specify the Overflow output.

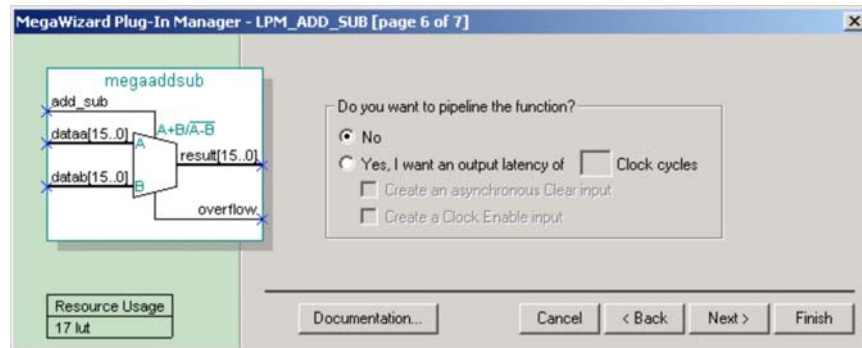


Figure 52. Refuse the pipelining option.

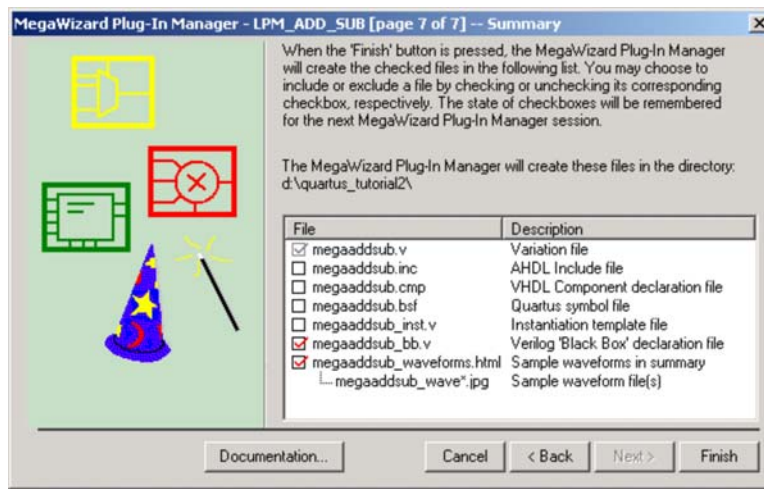


Figure 53. Files created by the wizard.

The modified Verilog code for the adder/subtractor design is given in Figure 55. Put this code into a file `quartus_tutorial2\addersubtractor2.v`. The differences between this code and Figure 11 are:

- The top-level module is now called *addersubtractor2*.
- The **assign** statements that define the *overflow* signal and the XOR gates (along with the signal defined as **wire H**) are no longer needed.
- The *adderk* instance of the adder circuit is replaced by *megaddsub*. Note that the *dataa* and *datab* inputs shown in Figure 49 are driven by the *G* and *Breg* vectors, respectively. Also, the inverted version of the *AddSubR* signal is specified to conform with the usage of this control signal in the LPM.
- The *adderk* module is deleted from the code.

```

// Adder/subtractor module created by the MegaWizard
module megaddsub (
    add_sub,
    dataa,
    datab,
    result,
    overflow);

    input add_sub;
    input [15:0] dataa;
    input [15:0] datab;
    output [15:0] result;
    output overflow;
    wire sub_wire0;
    wire [15:0] sub_wire1;
    wire overflow = sub_wire0;
    wire [15:0] result = sub_wire1[15:0];

    lpm_add_sub lpm_add_sub_component (
        .dataa (dataa),
        .add_sub (add_sub),
        .datab (datab),
        .overflow (sub_wire0),
        .result (sub_wire1));
    defparam
        lpm_add_sub_component.lpm_width = 16,
        lpm_add_sub_component.lpm_direction = "UNUSED",
        lpm_add_sub_component.lpm_type = "LPM_ADD_SUB",
        lpm_add_sub_component.lpm_hint = "ONE_INPUT_IS_CONSTANT=NO",
        lpm_add_sub_component.intended_device_family = "Stratix";
endmodule

```

Figure 54. Verilog code for the ADD_SUB LPM.

To explicitly include the *megaddsub.v* file in the project, select **Project | Add/Remove Files in Project** to reach the window in Figure 15. Specify the file name and click **OK**. The modified design can be compiled and simulated in the usual way.

Compile the design and look at the summary, which is depicted in Figure 56. Observe that the modified design is implemented in 51 logic elements, which is slightly different from the 53 indicated in Figure 18. In general, an LPM offers a highly efficient implementation of the needed function, and is a good choice whenever a suitable LPM exists.

```

// Top-level module
module addersubtractor2 (A, B, Clock, Reset, Sel, AddSub, Z, Overflow);
    parameter n = 16;
    input [n-1:0] A, B;
    input Clock, Reset, Sel, AddSub;
    output [n-1:0] Z;
    output Overflow;
    reg SelR, AddSubR, Overflow;
    reg [n-1:0] Areg, Breg, Zreg;
    wire [n-1:0] G, M, Z;
    wire over_flow;

// Define combinational logic circuit
mux2to1 multiplexer (Areg, Z, SelR, G);
    defparam multiplexer.k = n;
megaddsub nbit_adder (~AddSubR, G, Breg, M, over_flow);
assign Z = Zreg;

// Define flip-flops and registers
always @(posedge Reset or posedge Clock)
    if (Reset == 1)
        begin
            Areg <= 0; Breg <= 0; Zreg <= 0;
            SelR <= 0; AddSubR <= 0; Overflow <= 0;
        end
    else
        begin
            Areg <= A; Breg <= B; Zreg <= M;
            SelR <= Sel; AddSubR <= AddSub; Overflow <= over_flow;
        end
endmodule

// k-bit 2-to-1 multiplexer
module mux2to1 (V, W, Sel, F);
    parameter k = 8;
    input [k-1:0] V, W;
    input Sel;
    output [k-1:0] F;
    reg [k-1:0] F;

    always @(V or W or Sel)
        if (Sel == 0) F = V;
        else F = W;
endmodule

```

Figure 55. Verilog code for the circuit in Figure 45.

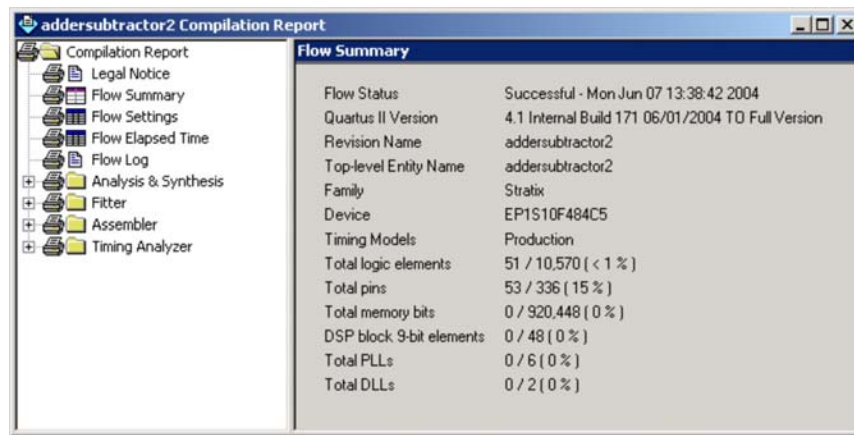


Figure 56. The Compilation results for the code in Figure 55.

7 Quartus II Windows

The Quartus II display contains a number of utility windows, which can be positioned in various places on the screen, changed in size, or closed. In Figure 17, which is reproduced in Figure 57, there are four windows.

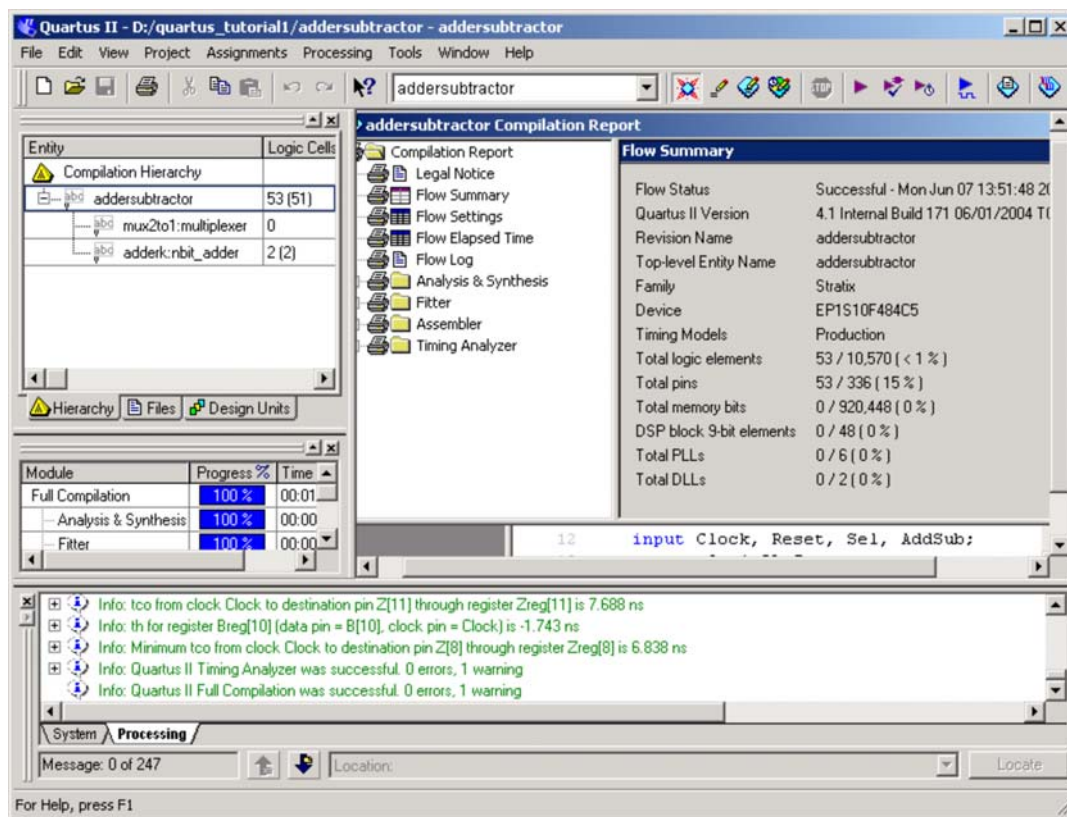


Figure 57. The main Quartus II display.

The Project Navigator window is shown near the top left of the figure. Under the heading Compilation Hierarchy, it depicts a tree-like structure of the designed circuit using the names of the modules in the Verilog code of Figure 11. To see the usefulness of this window, open the previously compiled project *quartus_tutorial1\addersubtractor* to get to the window that corresponds to Figure 57. Now, double-click on the name *adderk* in the hierarchy. Quartus II will open the file *addersubtractor.v* and highlight the Verilog module that specifies the adder subcircuit, as indicated in Figure 58. Next, right-click on the same name and choose **Locate in Last Compilation Floorplan** from the pop-up menu that appears. This causes Quartus II to display the floorplan, as in Figure 23, and highlight the part that implements the adder subcircuit.

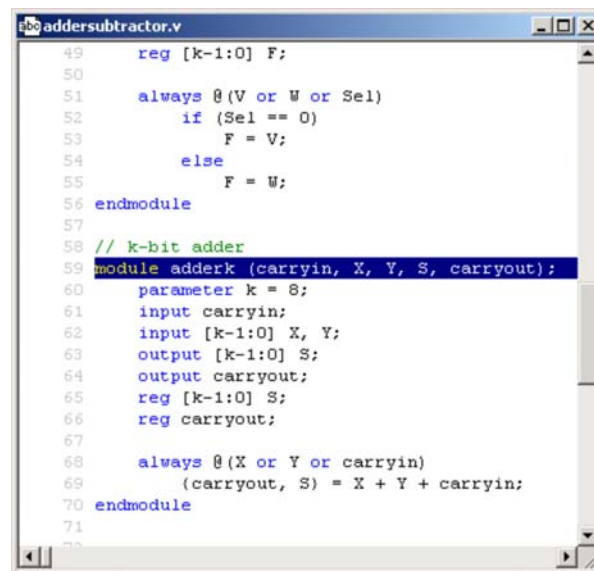
The Status window is located below the Project Navigator window in Figure 57. As you have already observed, this window displays the compilation progress as a project is being compiled by Quartus II.

At the bottom of Figure 57 there is the Message window, which displays user messages produced during the compilation process.

The large area on the right side of the Quartus II display is used for various purposes. As we have seen, it is used by the Text and Waveform Editors. It is also used to display various results of compilation and simulation.

A utility window can be moved by dragging its title bar, resized by dragging the window border, or closed by clicking on the X in the top-right corner. A particular utility window can be opened by using the **View | Utility Windows** command.

The commands available are *context sensitive*, depending on which Quartus II tool is currently being used. For example, when the Text Editor is in use, the Edit menu contains a different set of commands than when another tool, such as the Waveform Editor, is in use.



```
49     reg [k-1:0] F;
50
51     always @(V or W or Sel)
52         if (Sel == 0)
53             F = V;
54         else
55             F = W;
56 endmodule
57
58 // k-bit adder
59 module adderk (carryin, X, Y, S, carryout);
60     parameter k = 8;
61     input carryin;
62     input [k-1:0] X, Y;
63     output [k-1:0] S;
64     output carryout;
65     reg [k-1:0] S;
66     reg carryout;
67
68     always @(X or Y or carryin)
69         (carryout, S) = X + Y + carryin;
70 endmodule
71
```

Figure 58. Locating the *adderk* instance from the Project Navigator.

Copyright ©2004 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

This document is being provided on an "as-is" basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.