4 Programmable Logic Devices: CPLDs and FPGAs with VHDL Design

OUTLINE

OBJECTIVES

Upon completion of this chapter, you should be able to:

- Explain the benefits of using PLDs.
- Describe the PLD design flow.
- Understand the differences between a PAL, PLA, SPLD, CPLD, FPGA and an ASIC.
- Explain how a graphic editor and a VHDL text editor are used to define logic to a PLD.
- Interpret the output of a simulation file to describe logic operations.
- Interpret VHDL code for the basic logic gates.

INTRODUCTION

As you can imagine, stockpiling hundreds of different logic ICs to meet all the possible requirements of complex digital circuitry became very difficult. Besides having all of the possible logic on hand, another problem was the excessive amount of area on a printed-circuit board that was consumed by requiring a different IC for each different logic function. In many cases, only one or two gates on a quad or hex chip were used.

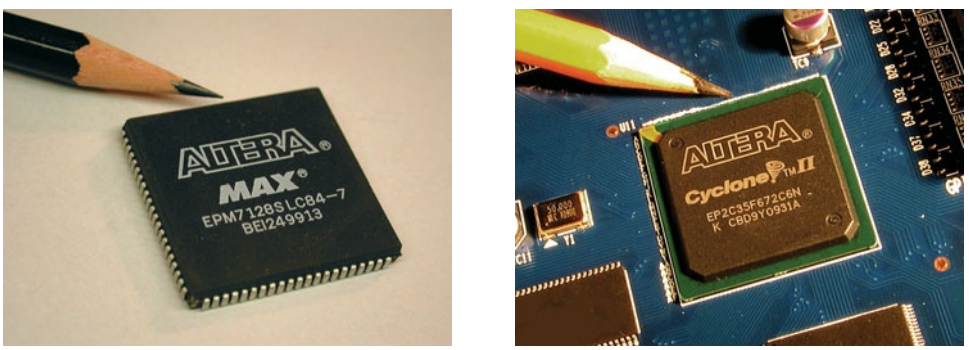
Then came "programmable logic"—the idea that implementing all logic designs using 7400- or 4000-series ICs is no longer needed. Instead, a company will purchase several user-configurable ICs that will be customized (i.e., programmed) to perform the specific logic operation that is required. These ICs are called **programmable logic devices (PLDs)**.

4–1 PLD Design Flow

Samples of two PLDs are shown in Figure 4–1. They contain thousands of the basic logic gates plus advanced sequential logic functions inside a single package. This internal digital logic, however, is not yet configured to perform any particular function. One way to configure it is for the designer to first use PLD computer software to draw the logic that he or she needs implemented. This is called **CAD** (computer-aided design). The PLD software then performs a process called **schematic capture**, which reads the graphic drawing of the logic and converts (compiles) it to a binary file that accurately describes the logic to be implemented. This binary file is then used as an input to a programming process that electronically alters the internal PLD connections (synthesizes) to make it function specifically as required. Hundreds, or even thousands, of digital logic ICs will be replaced by a single PLD.

Another way to define the logic to be programmed into the PLD is to use a high-level language called Hardware Description Language (HDL). A specific form of HDL used by several manufacturers is called **VHDL**, which stands for VHSIC Hardware Description Language (where VHSIC stands for Very High-Speed Integrated Circuit). In this case, the inputs, outputs, and logic processes are defined using statements based on the C programming language. This method is somewhat more difficult to learn, but depending on the logic, it can be a more powerful—and simpler—tool with which to define complex or repetitive logic.

Figure 4–2 illustrates the design flow. First we need to define the digital logic problem that we want to solve. Once we have a good understanding of the problem, we can develop the equations to use in solving the logic operation that we want the circuit to perform.



(a) (b)

Figure 4–1 Sample PLDs: (a) Altera MAX CPLD; (b) Altera Cyclone FPGA.

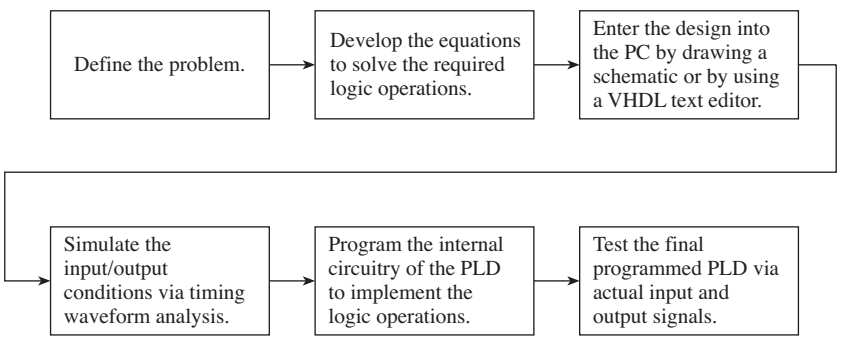


Figure 4–2 PLD product design flow.

After we have completed that work on paper, we will enter the design into a personal computer (PC) by drawing a schematic diagram using the CAD tools provided with the PLD software. In some cases, the design will instead be entered using the VHDL text editor provided. After the PC has analyzed the design, it will allow us to perform a simulation of the actual circuit to be implemented. To do this, we specify the input levels to our circuit, and we observe the resultant output waveforms on the PC screen using the waveform analysis tool provided.

If the computer simulation shows that our circuit works correctly, we can program the logic into a PLD chip that is connected by a cable to the back of our PC. The final step would be to connect actual inputs and outputs to the chip to check its performance in a real circuit.

To illustrate the power of a PLD, let's consider the logic circuit required to implement $X = \overline{A}B + \overline{B + C}$. Figure 4–3 shows the circuitry required to implement the logic using 7400-series ICs. As shown, we would need four different ICs to solve this equation. Wires are shown connecting one gate of each IC to one gate of the next IC until the logic requirements are met.

To solve this same logic using a PLD, we would draw the schematic or use VHDL to define the logic, then program that into a PLD. One possible PLD that could be used to implement this logic is the Altera EPM7128S (see Figure 4–4). After completing the steps listed in Figure 4–2, the internal circuitry of the PLD is configured (in this case) to input A, B, and C at pins 29, 30, and 31 and output to X at pin 73. The PLD software selected which pins to use, and as you can see, only a small portion of the PLD is actually used for this circuit.

This particular PLD is an 84-pin IC in a plastic leaded chip carrier (PLCC) package having 21 pins on a side. The notch signifies the upper left corner of the IC. Pin 1 is located in the middle of the upper row adjacent to a small indented circle;

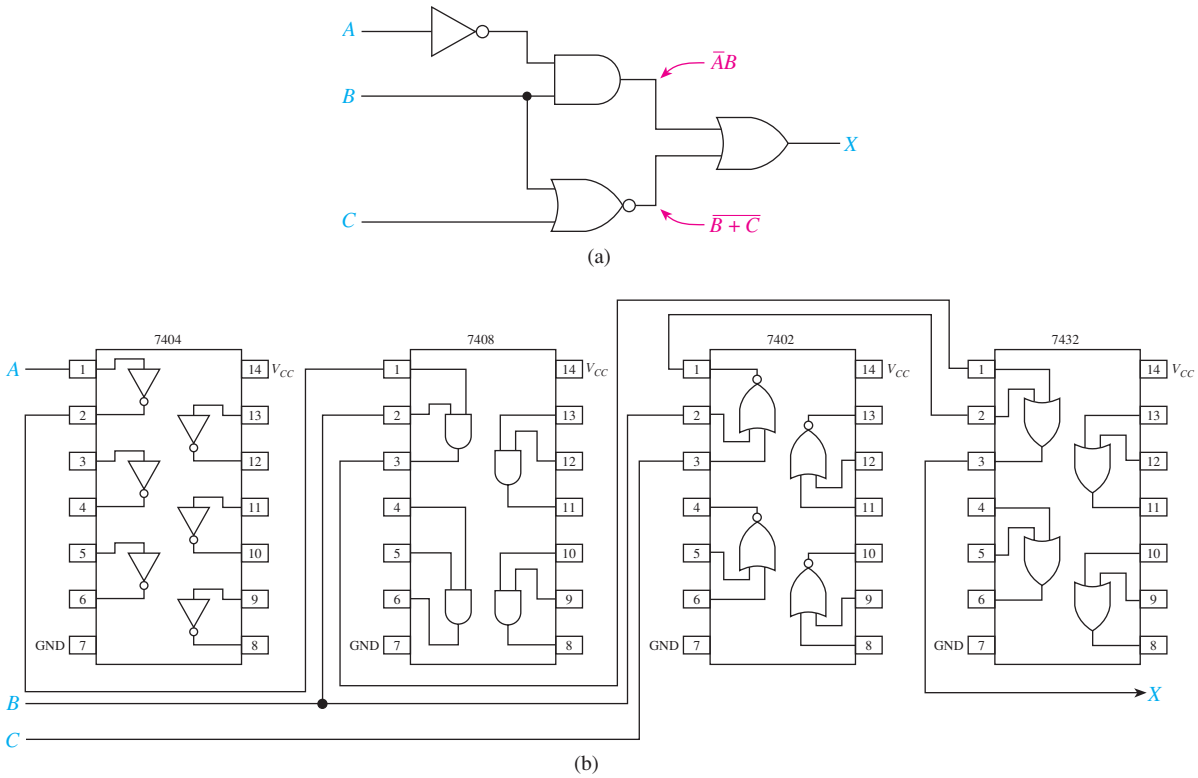


Figure 4–3 Implementing the equation $X = \overline{A}B + \overline{B + C}$ using 7400-series logic ICs: (a) logic diagram; (b) connections to IC chips.

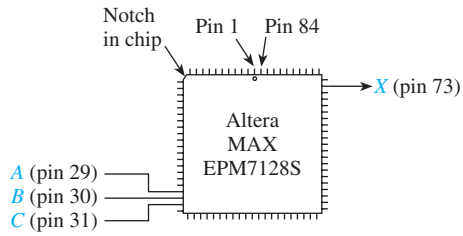


Figure 4–4 Implementing the equation $X = \overline{A}B + \overline{B + C}$ using a PLD.

subsequent pin numbers are counted off counterclockwise from there. (A photograph of this particular chip is shown in Figure 4–1[a].)

As you may suspect, the price of a PLD is higher than a single 7400-series IC, but we've only used a small fraction of the PLD's capacity. We could enter and program hundreds of additional logic equations into the same PLD. The only practical limitation is the number of input and output pins that are available. Many PLDs are erasable and reprogrammable, allowing us to test many versions of our designs without ever changing ICs or the physical wiring of the gates.

We will learn design entry and waveform simulation in this chapter, and we will continue to explore PLD examples and problems throughout the remainder of this text.

One of the leading manufacturers of PLDs is Altera Corporation. Altera offers a full line of CPLDs, FPGAs, and ASICs (all explained in Section 4–2). This manufacturer of programmable logic was chosen for this textbook because they are an industry leader and offer a high level of support to colleges and universities. They also provide a free download version of their design and development software called Quartus II: Web Edition, which we will use throughout the text to design and simulate FPGA-based logic circuits.

PLD development boards that attach directly to the USB port of a PC are available so that you can experience programming and debugging actual PLD ICs. These development boards allow you to program and reprogram repeatedly, so they are a great option for all of your digital experimentation. Typically, a PLD development board will contain a CPLD or an FPGA, a USB port to connect to your PC, and several I/O switches and LEDs to test your design. The board that we will use throughout this textbook is the Altera DE2 Development and Education board. This, and several other development boards, are available through the Altera University Program. Figure 4–5 shows the DE2 development board.

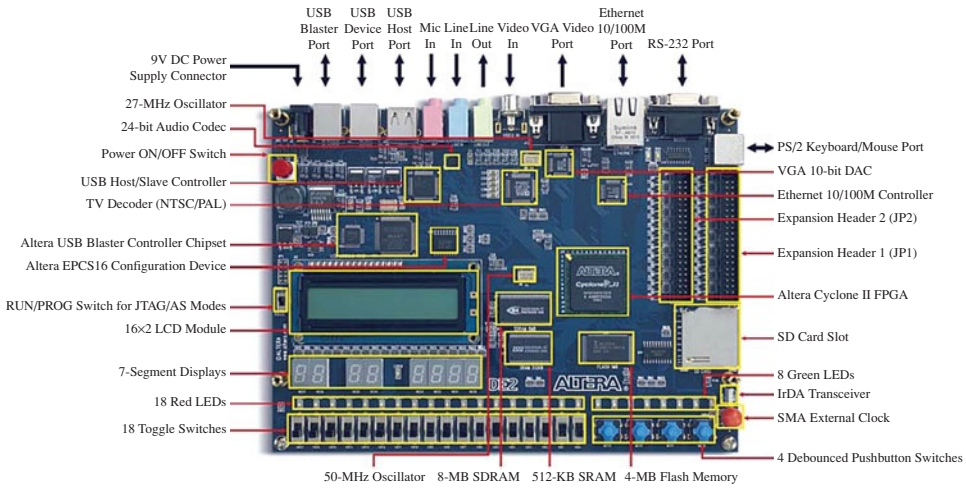


Figure 4–5 The Altera DE2 Development and Education board. (Courtesy of Altera Corporation.)

4–2 PLD Architecture

Basically, there are four types of PLDs: simple programmable logic devices (SPLDs), complex programmable logic devices (CPLDs), field-programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs).

The SPLD

The **SPLD** is the most basic and least expensive form of programmable logic. It contains several configurable logic gates, programmable interconnection points, and may also have memory flip-flops. (Flip-flops are covered in Chapter 10.) To keep logic diagrams easy to read, a one-line convention has been adopted, as shown in Figure 4–6, which is just a small part of an SPLD, showing two inputs and four outputs. (A typical SPLD like the PAL in Figure 4–9 has 16 inputs plus their complements and 8 outputs.) As you can see in Figure 4–6, the A input is split into two different lines: A, and its complement $\overline{A}$. (The triangle symbol is a special type of inverter having two outputs: a true and a complement.) The same goes for the B input and any others that are on the SPLD. The W, X, Y, and Z AND gates are programmable to have any of those four lines ($A, \overline{A}, B, \overline{B}$) as inputs.

The internal SPLD interconnect points are either made or not made by the PLD programming software. In Figure 4–6, the inputs to the W AND gate are connected to A and B. (The connections are shown by a dot.) The inputs to the X AND gate are connected to A and $\overline{B}$, and so on. The outputs of these AND gates are called the **product terms**, because W is the Boolean product of A and B and X is the Boolean product of A and $\overline{B}$.

The product terms in Figure 4–6 are not very useful by themselves. The circuit is made more effective by adding an OR gate to the structure, as shown in Figure 4–7. This new configuration is the foundation for a **programmable array logic (PAL)**–type SPLD. As Figure 4–7 shows, by OR-ing the four product terms together, we now have the Boolean sum of the four product terms, simply called the **Sum-of-Products (SOP)**. The SOP is the most common form of Boolean equation used to represent digital logic. (For more on SOPs, see Section 5–6.)

The **programmable logic array (PLA)** goes one step further by providing *programmable* OR gates for combining the product terms. Figure 4–8 shows a small portion of a PLA. In this illustration, the PLA provides two SOP equations. The inputs to the first OR gate are programmed to connect to all four product terms ($X = AB + A\overline{B} + \overline{A}B + \overline{A}\,\overline{B}$). The inputs to the second OR gate are programmed to connect to only the first and third product terms ($Y = AB + \overline{A}B$).

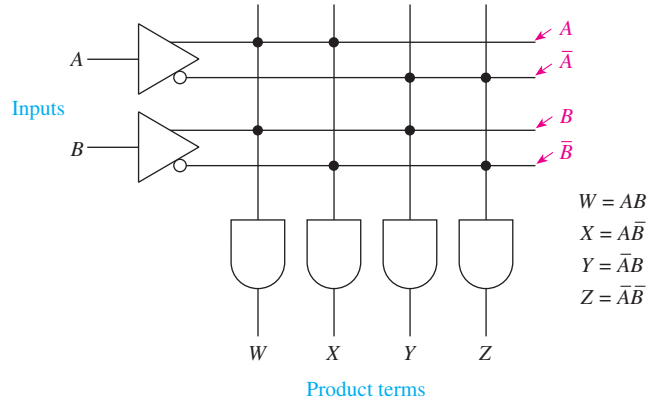


Figure 4–6 One-line convention for PLDs.

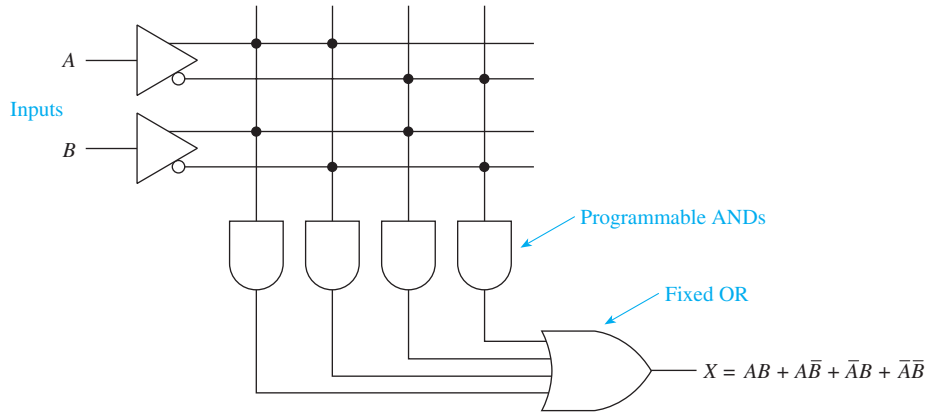


Figure 4–7 PAL architecture of an SPLD.

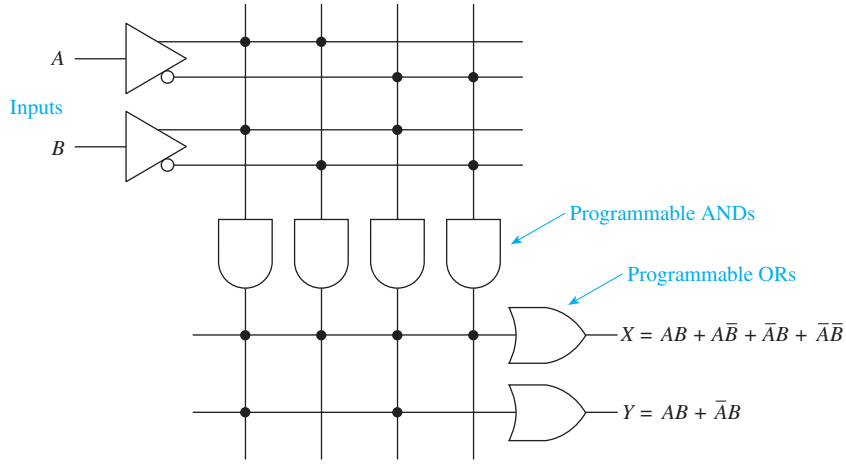


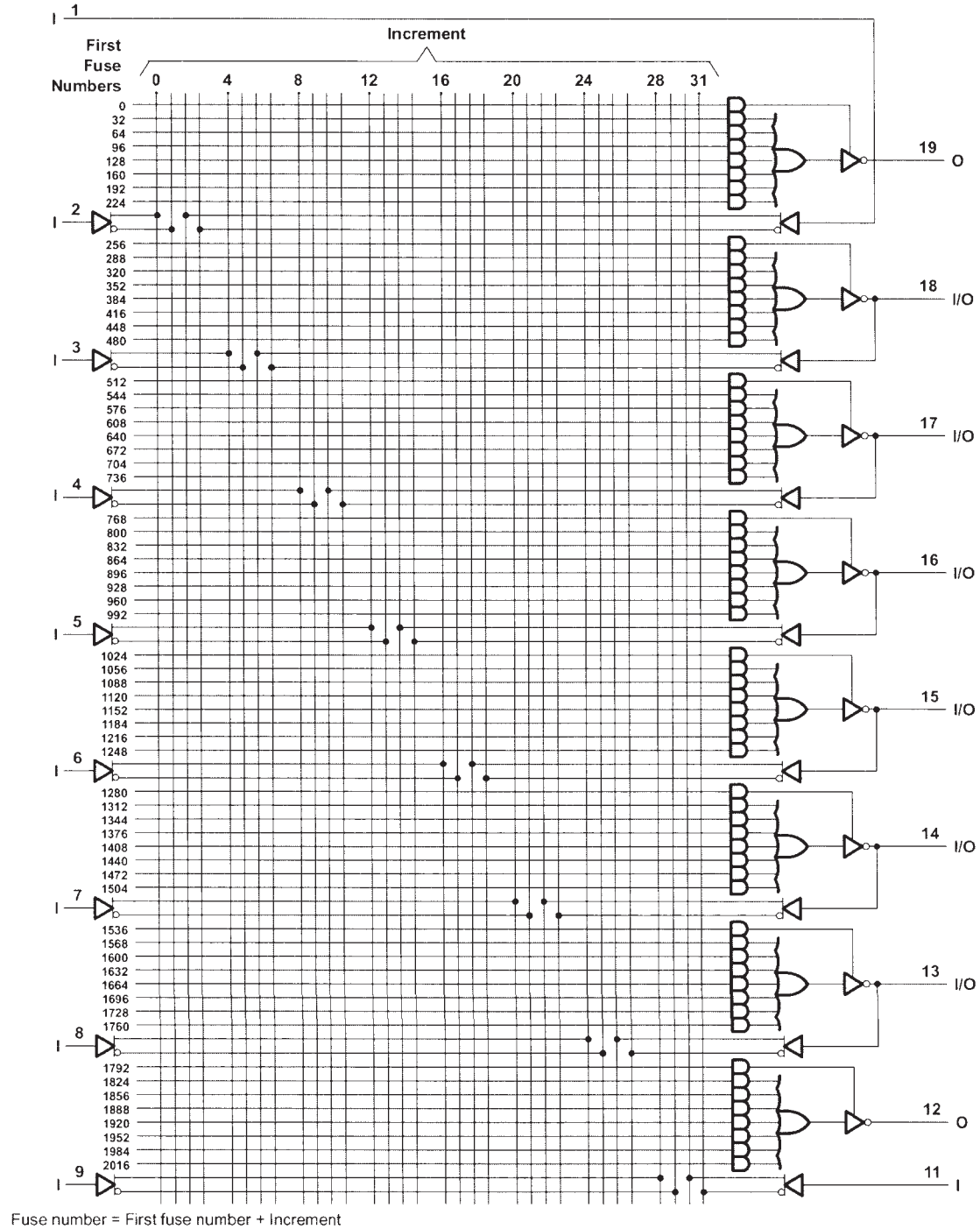
Figure 4–8 PLA architecture of an SPLD.

Some SPLDs also contain a flip-flop memory section and data-steering circuitry. Flip-flop memory circuitry is used in a type of digital circuitry called *sequential logic*. This type of logic is a form of digital memory that changes states based on previous logic conditions and specific logic control inputs. (Sequential logic is covered in detail in Chapters 12 and 13.) The data-steering circuitry takes care of input and control signal interconnections and logic output destinations.

PAL16L8

A sample of a typical PAL device is the PAL16L8 shown in Figure 4–9. The number 16 in the part number signifies that it has 16 inputs. The 8 signifies 8 outputs and the letter L means that the outputs are active-LOW. An active-LOW output is one that goes LOW instead of HIGH when activated. Ten of the inputs in the figure are labeled with the letter I. Each of these can provide the true and the complement of the level placed on the pin. The other 6 inputs are labeled I/O. This means that they can be used as an input or an output. To come up with a total of 8 outputs, the other 2 dedicated outputs labeled O are provided on pins 12 and 19.

logic diagram (positive logic)



Fuse number = First fuse number + Increment

Figure 4–9 The PAL16L8 SPLD logic diagram. (Courtesy of Texas Instruments)

The CPLD

The **CPLD** is made by combining several PAL-type SPLDs into a single IC package, as shown in Figure 4–10. Each PAL-type structure is called a *macrocell.* Each macrocell has several I/O connection points, which go to the chips' external leads. The macrocells are all connected to control signals and to each other via the programmable interconnect matrix shown in the center of the structure.

The Altera MAX 7000S series is an example of a CPLD family. These CPLDs perform the functions of thousands of individual logic gates. They also feature a **nonvolatile** characteristic, meaning that when power is removed from the chip, they will remember their programmed logic and interconnections. (This type of memory is called EEPROM or Flash memory and is covered in Chapter 16.) These ICs can be repeatedly programmed to implement new designs or correct faulty ones, thus eliminating the need to rewire circuitry or buy new logic.

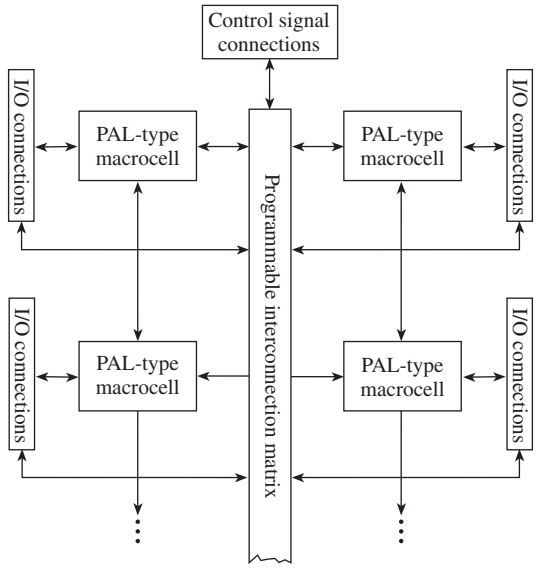


Figure 4–10 Internal structure of a CPLD.

The FPGA

As the name implies, a **Field-Programmable Gate Array (FPGA)** is an array of gates interconnected in a row-column matrix that can be programmed in the field by a computer via a USB connection. The FPGA differs from the CPLD in that, instead of solving the logic design by interconnecting logic gates, it uses a **look-up table** (LUT) method to resolve the particular logic requirement. This allows PLD manufacturers to form a more streamlined design, creating a much denser and faster PLD. Besides having thousands of internal logic elements, FPGAs have hundreds of I/O pins with programmable internal interconnects and storage registers. The Altera Cyclone© series is an example of an FPGA family.

To see how a look-up table works, refer to Figures 4–11(a) and (b). In Figure 4–11(a), the conventional logic for the equation $X = ABCD + A\overline{B}C\overline{D} + \overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$ is implemented using 7400-series ICs. In this case, X is HIGH for three different combinations of the four inputs (X is HIGH when $ABCD = 1111$ or 1010 or 0000).

Figure 4–11(b) shows the same logic implemented in an FPGA LUT. An LUT operates similar to a truth table in that it provides for all possible input combinations and produces a HIGH when the desired combinations of 1s and 0s are provided at the inputs. In Figure 4–11(b), the routing of the logic levels is controlled by the 15 cascaded data selectors (trapezoid symbols). They are actually multiplexers, which are

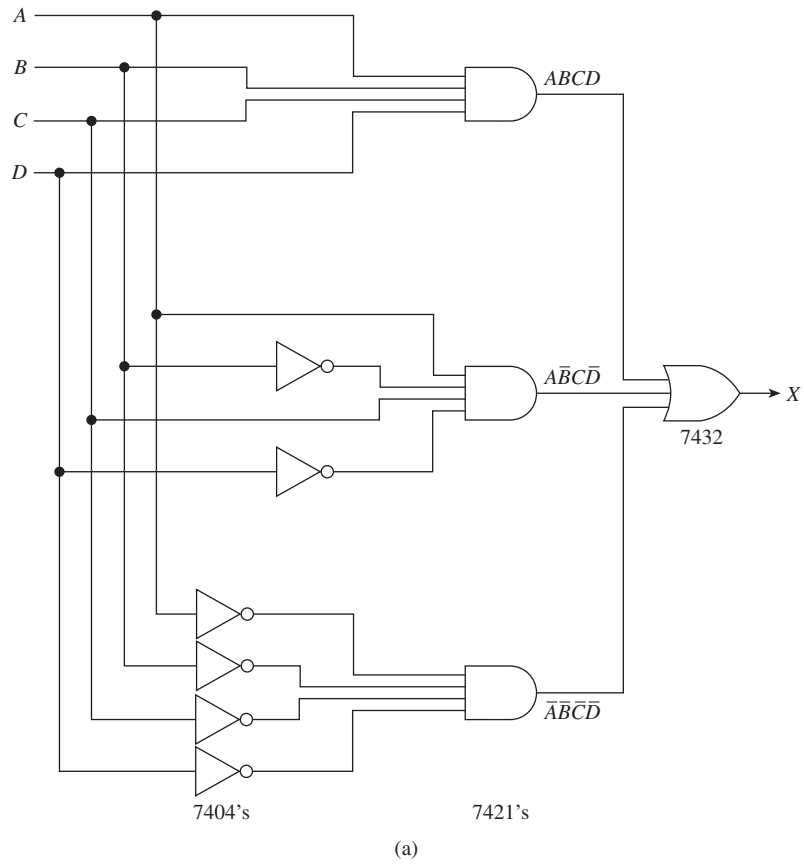


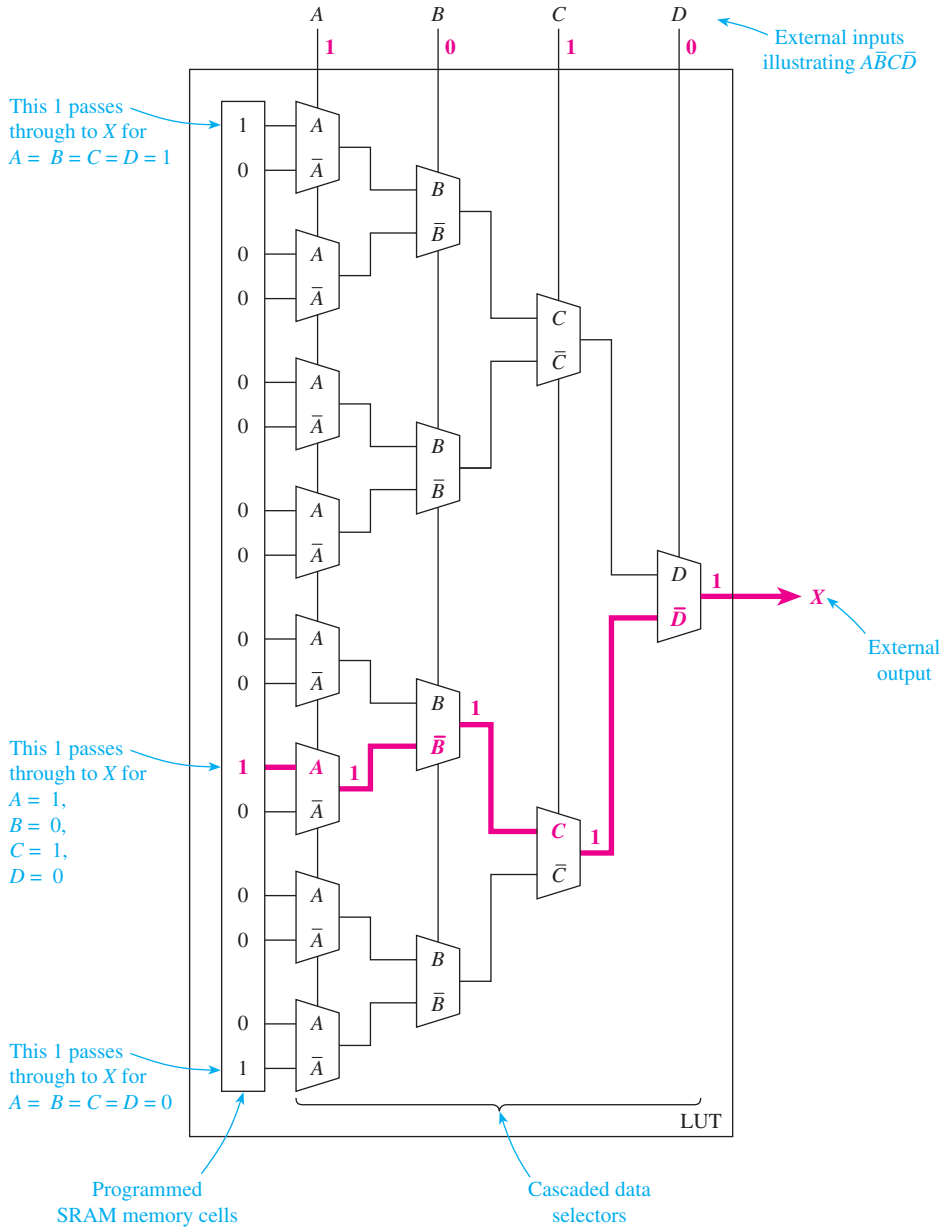
Figure 4–11 $X = ABCD + A\overline{B}C\overline{D} + \overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$: (a) implemented using 7400-series ICs; (b) implemented within a LUT of an FPGA (showing the flow for $A\overline{B}C\overline{D}$).

covered in Chapter 8, but for now all we need to understand is that when the control input A, B, C, or D is HIGH, the logic level on the TRUE input is passed through from left to right. When it is LOW, the logic level on the complement input is passed through. The external A control input actually controls eight data selectors: B controls four, C controls two, and D controls one.

This illustration of a LUT shows the flow of logic when the inputs are set at $A = 1, B = 0, C = 1$ and $D = 0$. In this case, since $A = 1$, then all logic levels connected to the eight TRUE As are passed through. Therefore, by just looking at the highlighted data path, a 1 is passed through to the B data selector. Now, since the B data selector control input is 0, then the data passes through the $\overline{B}$ to the C data selector, and so on. The end result of this path is that a 1 passes through to X when $ABCD = 1010$. To confirm that you understand this logic, follow the logic for $ABCD = 1111$ and then for $ABCD = 0000$ to see that these conditions are also met.

As you can see, the result at X is dependent on the logic levels programmed into the SRAM (static random-access memory) memory cells (covered in Chapter 16). These memory cells are volatile and will need to be reinitialized along with the internal interconnections and registers each time the FPGA is powered on. Although CPLDs have the advantage of being non-volatile, FPGAs are much denser and faster so are used more often in middle to high-end applications.

The FPGA that is on the Altera DE-2 Development board shown in Figure 4–5 is the Cyclone EP2C35F672C6N. It contains 33,216 look-up tables and has 475 pins dedicated for input/output to external circuitry. According to the *ordering Information* at the Altera Cyclone Web site, the *672* in the part number indicates the number of pins



(b)

Figure 4–11 *Continued*

and the letter *F* denotes that it is a FineLine Ball Grid Array (BGA). In order to provide for 672 pins, the BGA pins are on the bottom of the IC setup as 26 rows by 26 columns. (The four outside corner pins are left off.)

The ASIC

Once a logic design has been created and tested on an FPGA, and if there is a large quantity demand, the design can be transferred to an **application-specific integrated circuit (ASIC)**. ASICs are available that are pin compatible and functionally equivalent to their corresponding FPGA product. An important feature of ASICs is that the logic function is hard-coded into the IC, making them non-volatile, so the user does not have to reconfigure the IC at each power-on.

*4–3 Using PLDs to Solve Basic Logic Designs

So, the next obvious question is "How do I design logic with a PLD?" We will use the Quartus® II software to design and simulate solutions modeled after Altera FPGAs. Then, if your laboratory has the PLD programmer boards like the DE-2 shown in Figure 4–5, you can test the actual operation of the FPGA with switches and lights. Even without the boards, however, the design and simulation software is a great learning tool for digital logic.

Figure 4–12 shows the flow of operations required to design, simulate, and program an FPGA. Several methods are actually available to perform the design entry, but we will address the two most common: graphic, and VHDL. The **block (schematic) editor** enables you to connect predefined logic symbols (AND, NAND, OR, etc.) together with inputs and outputs to define the logic operation that you need to implement. The **VHDL editor** is a text editor that helps you to define the logic in a programming language environment. In a text form, you specify the inputs, outputs, and logic operations that you need to implement.

The next step performed by the software is to compile and synthesize the design. A **compiler** is a language and symbol translation program that interprets VHDL statements and logic symbols, then translates them into a binary file that can be used to synthesize, then simulate and program the design into the FPGA IC. The compiler uses several symbol and VHDL library files to obtain the information needed to define the logic entered during the design entry stage. Report files are then generated that describe such things as I/O pin assignments, internal FPGA signal routing, and error messages. **Synthesizing** the design is the process the software completes to develop a model of the PLD's internal electrical connections, which will produce the actual logic functions that will later be simulated, then programmed into the PLD.

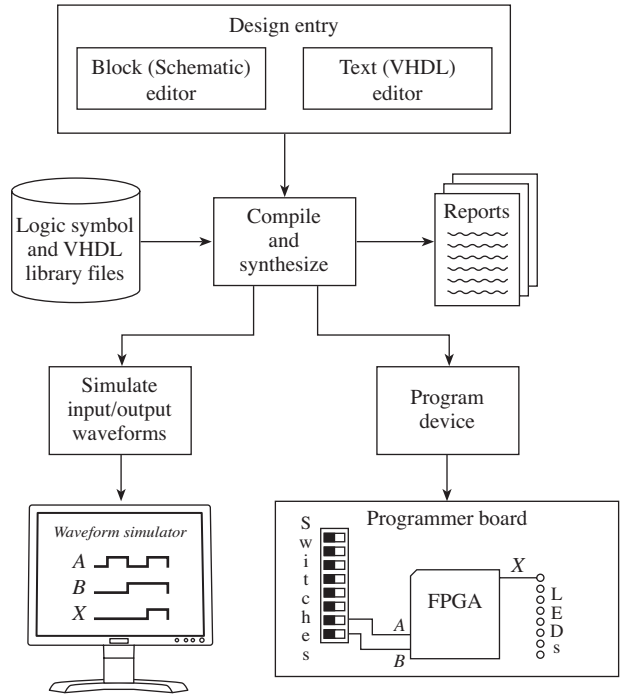


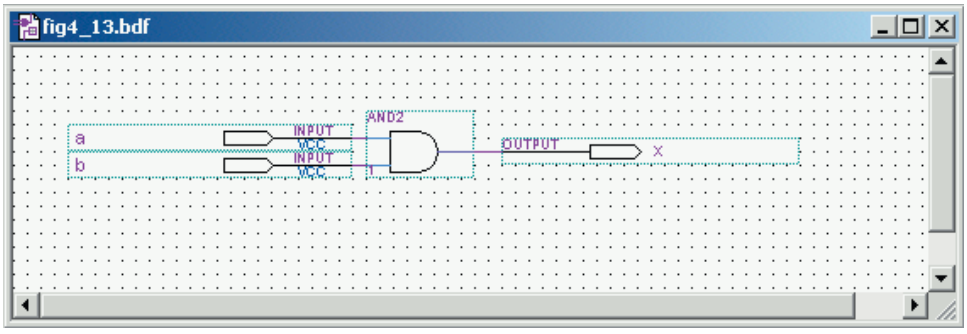
Figure 4–12 FPGA design flow.

*Note: The color bar on the edge of a page indicates that the material in that area covers the implementation of digital logic using PLD hardware and software. This method of logic implementation can be omitted without compromising the thorough coverage of digital electronics presented in the remainder of the book.

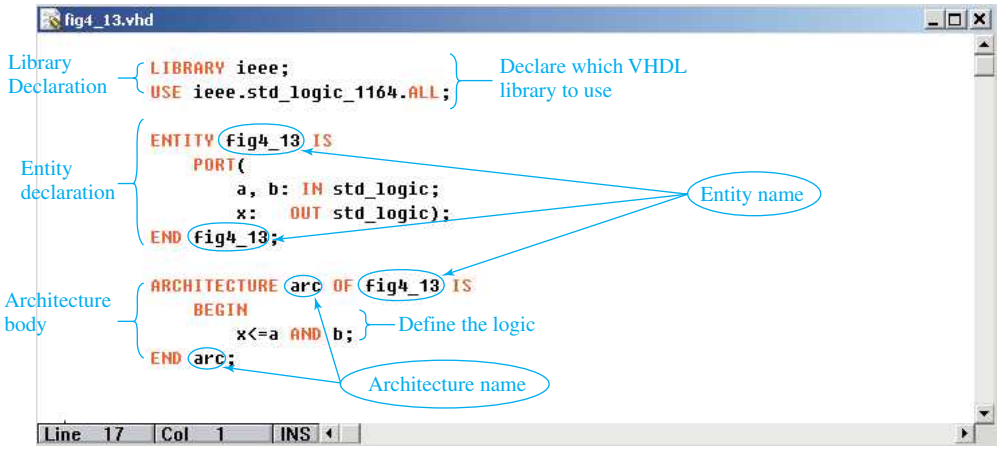
The **waveform simulator** provides a means to check the logic operation of your design. To use it, draw the input waveforms using the CAD tool provided, and the program will show the output response as if these inputs were applied to an actual FPGA. Finally, if you have an FPGA programmer board and the waveform simulation was accurate, you can program the FPGA and test it with actual inputs and outputs.

Quartus® II Software

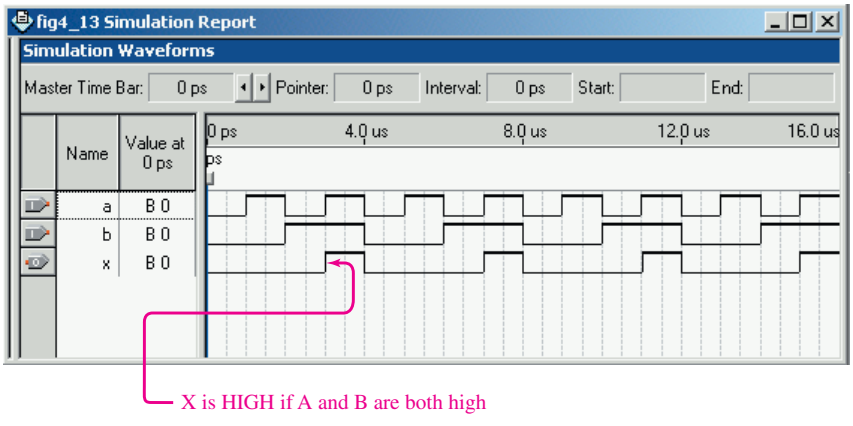
Figures 4–13(a), (b), and (c) are the actual computer screens that you will see when running the Quartus® II software to implement a simple 2-input AND gate following



(a)



(b)



(c)

Figure 4–13 Computer screen displays generated by Quartus® II software for the design of a 2-input AND gate: (a) block editor file; (b) alternative method using the VHDL text editor file; (c) simulation waveform file.

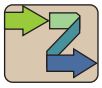
the design flow outlined in Figure 4–12. A tutorial on how to run the software appears in Section 4–4.

Figure 4–13(a) is produced by the *block (schematic) editor.* This method of design allows us to define the inputs, outputs, and circuit logic simply by drawing the logic diagram. This screen shows a 2-input AND gate with two input pins, *A* and *B*, and one output pin, *X*. This circuit was drawn by choosing each circuit component from a library of available symbols and then making each interconnection.

Figure 4–13(b) shows an alternate method of defining the same AND gate design using the *VHDL text editor.* The VHDL program is divided into three sections: **library declaration, entity declaration**, and **architecture body**. As with most computer languages, the first statements of the program are used to declare the library source for resolving and translating the language within the body of the program. In VHDL this is called the *library declaration.* The IEEE standard library (*ieee.std_ logic_1164.ALL*) is used most often by the VHDL compiler to translate references to the inputs, outputs, and logic statements used in the program.

The *entity declaration* defines the input (*a*, *b*) and output (*x*) ports to the CPLD. Note that the entity name (fig4_13) must match the file name (*fig4_13.vhd*) and it appears identically in three locations in the program listing. Also note the use of the underscore in the name because hyphens are not allowed.

The *architecture body* defines the internal logic operations ($x <= a$ AND b) that will be performed on those ports. (The symbol $<=$ means that output *x* receives the value of input *a* ANDed with input *b.*) The architecture name is arbitrary and it appears twice. The one used here is *arc.* As with the entity name, it cannot contain hyphens and it must start with a letter.



Helpful Hint

To make programs easier to read, all VHDL-reserved keywords should be capitalized and all variables should be lower-case.

To make the reading of VHDL programs easier, a formatting convention has been established. Basically, all capitalized words are VHDL-reserved keywords, and all lower-case words and letters are variables. Even though VHDL is not case sensitive, it is good practice for you to follow the convention presented in Figure 4–13(b). For example, writing the equation $x <= a$ AND b as $X <= A$ AND B would make no difference to VHDL, but it is harder to distinguish the keyword AND from the variables *A*, *B*.

You have probably guessed that for defining the action of a simple AND gate, VHDL design is more time-consuming than graphic entry, but we will see in later chapters that it is a much easier way to define logic when the circuits become more complex.

Figure 4–13(c) shows the simulation of the circuit produced by the *waveform simulation editor.* To produce that screen, the waveforms were first drawn for all possible combinations of *A* and *B* (like building a truth table). Then as the simulation is run, the software determines the logic state that would result at *X* for each combination of inputs and shows the result as the *X* waveform.

EXAMPLE 4–1

Figure 4–14 shows five computer screens generated by the Quartus® II software. Each screen produces, or is the result of, a different logic circuit. Determine the Boolean equation that is being implemented in each case.

Solutions:

(a) $X = A + B$

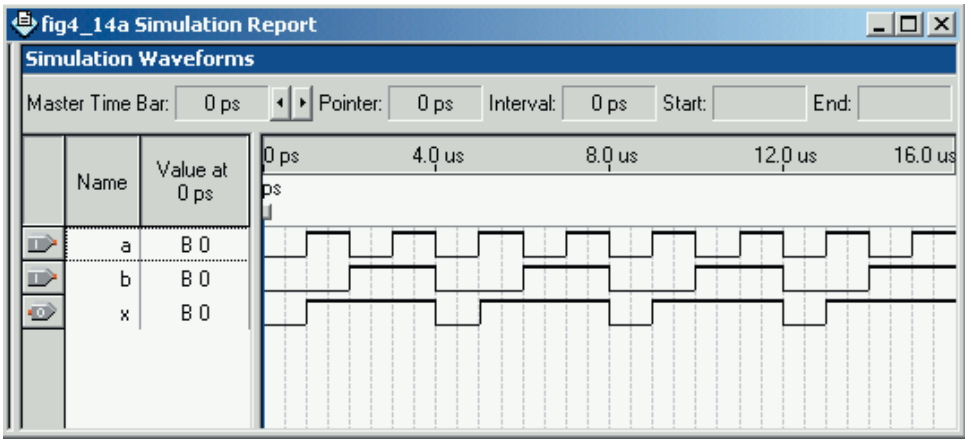
(b) $X = \overline{ABC}$

(c) $X = AB$

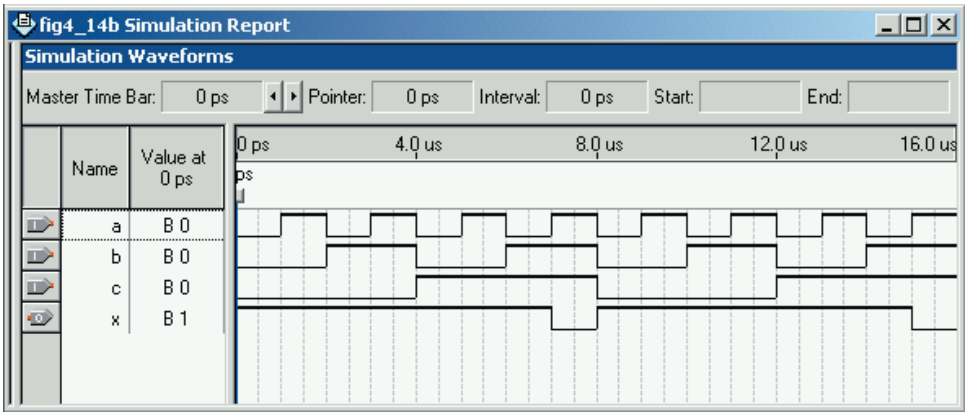
(d) $X = AB + \overline{B}C$

(e) $X = A + (B\overline{C})$

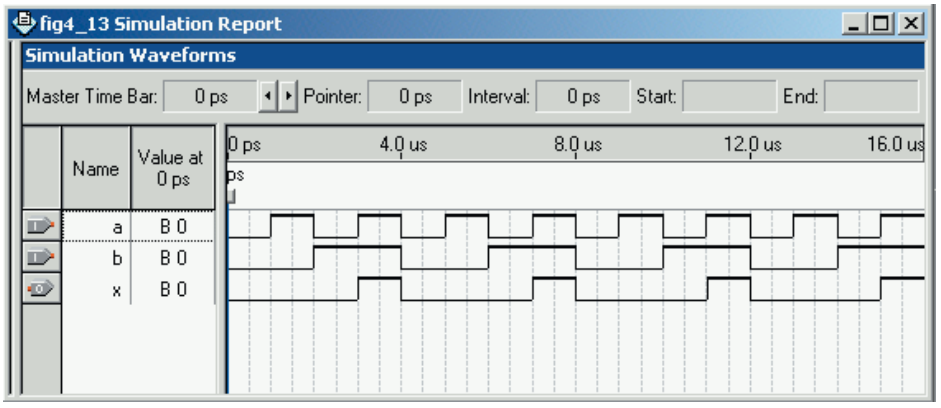
$Y = AB + \overline{B + C}$



(a)

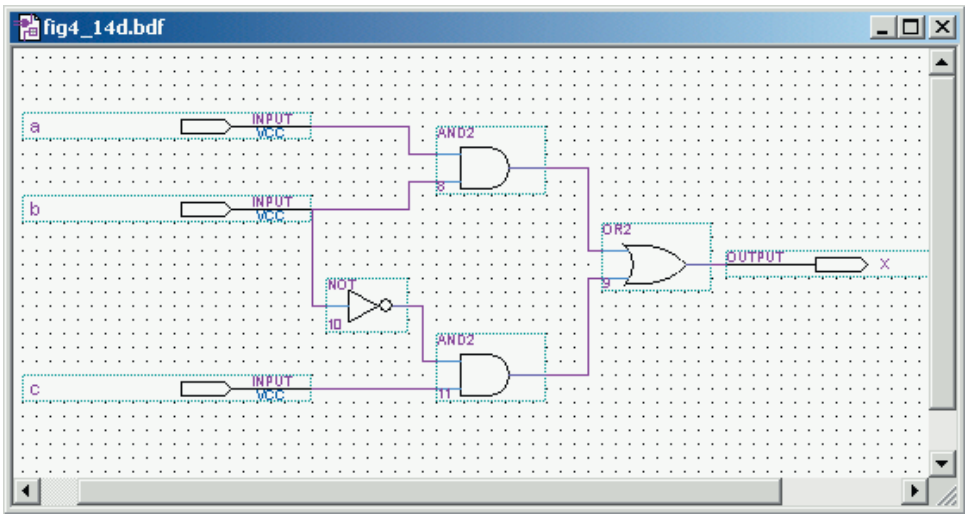


(b)



(c)

Figure 4–14 Computer screens generated by the Quartus® II software for Example 4–1.



(d)

fig4_14e.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY fig4_14e IS
    PORT(
        a, b, c: IN std_logic;
        x, y:    OUT std_logic);
END fig4_14e;

ARCHITECTURE arc OF fig4_14e IS
    BEGIN
        x<= (a OR (b AND NOT c));
        y<= ((a AND b) OR NOT (b OR c));
END arc;
```

Line 17 | Col 1 | INS

(e)

Figure 4–14 *Continued*

*4–4 Tutorial for Using Altera's Quartus® II Design and Simulation Software

To get started, you first need to download the free Quartus® II Web Edition Software. There are several versions available for download. The most appropriate version (and the one used throughout this text) is version 9.1 sp2. The reason for using this version is that when Altera migrated from version 9.1 sp2 to version 10, it needed to drop the capability to create vector waveform files (*vwf* files). These files are used to produce waveform simulations from within the Quartus® II design environment. The main reason a designer would use version 10 (and beyond) is if they have a need to use the highest-end CPLDs and FPGAs that weren't supported by earlier versions of the software. If you need that high level of development, the most current software version will be

*This section is also available as a series of podcast lectures on the textbook companion website.

required. In that case however, to perform waveform simulations, Altera recommends the use of another program called ModelSim® which runs external to the Quartus® II environment. QSIM®, another waveform simulator, runs external to Quartus® II in version 10 but should be internal in later versions. It will look and act just like the vector waveform editor described in this text.

For the best overall learning experience, it is recommended that you download and install the Quartus® II Web Edition version 9.1 sp2. This very popular version will continue to be available to download for many years to come from the Altera archives download site. (https://www.altera.com/download/archives)

In this tutorial we will implement a simple Boolean equation ($X = AB + CD$) to illustrate the steps involved to design, simulate, and program an FPGA using Altera's Quartus® II software.

1. Start the Altera Quartus® II software. The main screen is shown in Figure 4–15.

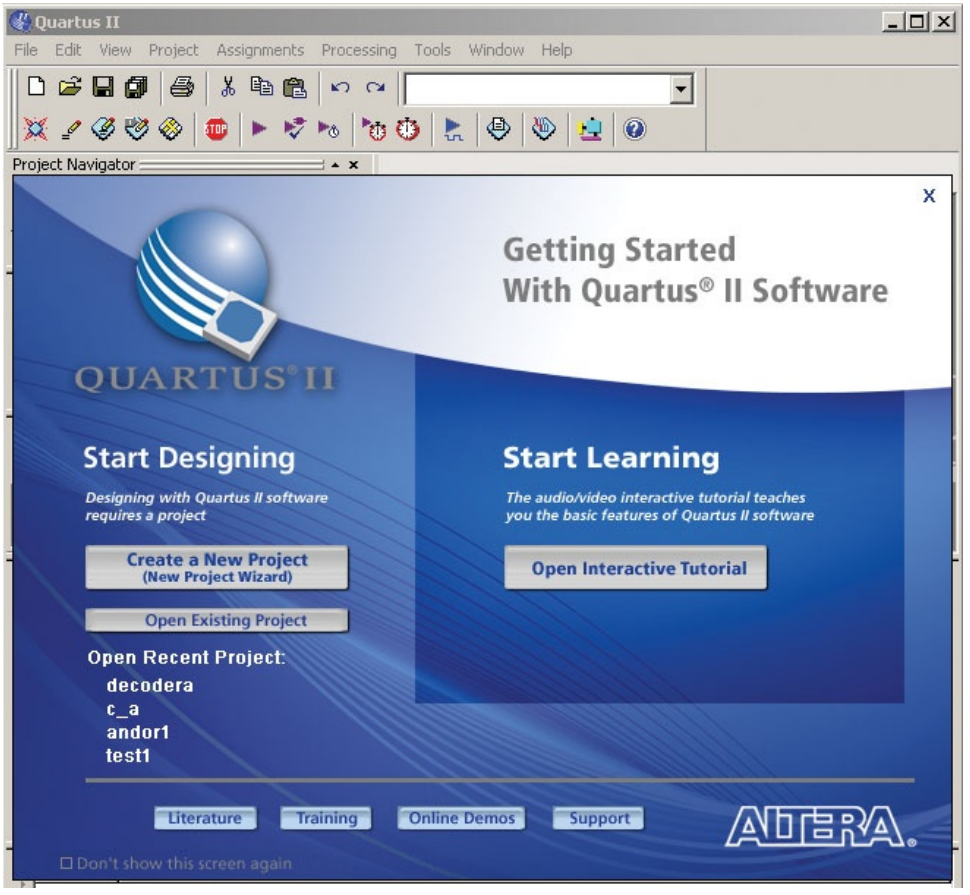


Figure 4–15 Quartus® II main screen. (Courtesy of Altera Corporation.)

Create a New Project

All of our designs will be contained within a "Project." Within the project we will create our design using the *Block Design Editor* to draw a schematic or the *Text Editor* to enter a VHDL program. We will also create a simulation file for the project to test the operation of our circuit before it is programmed into an FPGA.

2. To create a new project:

Press **Create a New Project** then press **Next**

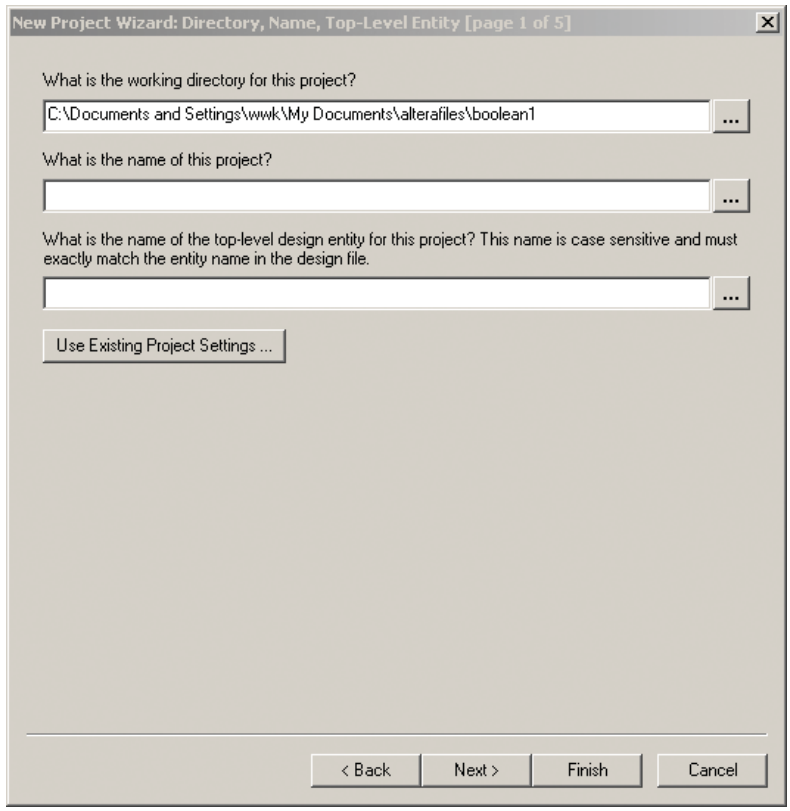


Figure 4–16 The New Project Wizard screen (1 of 5).

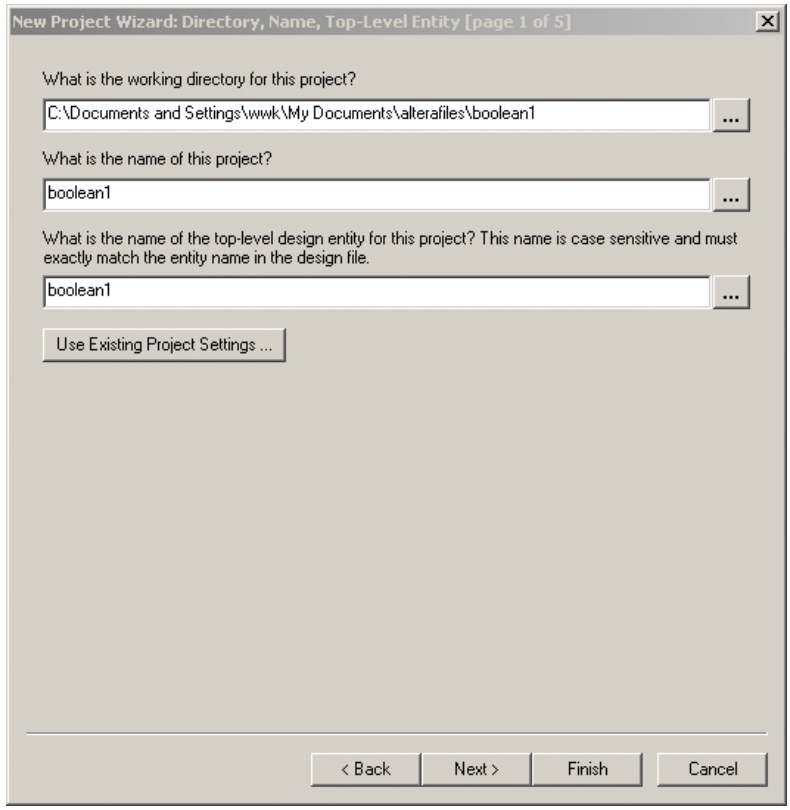


Figure 4–17 The New Project Wizard screen (1 of 5 [*Continued*]).

The New Project Wizard screen is shown in Figure 4–16. The first page of the New Project Wizard asks for the Directory, Name, and Top-Level Entity of the project. A good place to keep all of your projects is in your *MyDocuments* folder (or a removable flash drive). This figure shows a new sub-directory named *alterafiles* and a working directory named *boolean1*. All future FPGA work should be placed in the *alterafiles* subdirectory, and a new working directory (*boolean1* in this case) should be made for each new project.

3. Next you need to fill in a meaningful name and top-level entity for your project. I chose *boolean1* as shown in Figure 4–17. Notice: the name *boolean1* appears on all three lines. Press **Next** and **Yes** to create the new subdirectory.
4. The second wizard screen is shown in Figure 4–18. We have no additional design files to add, so press **Next**.

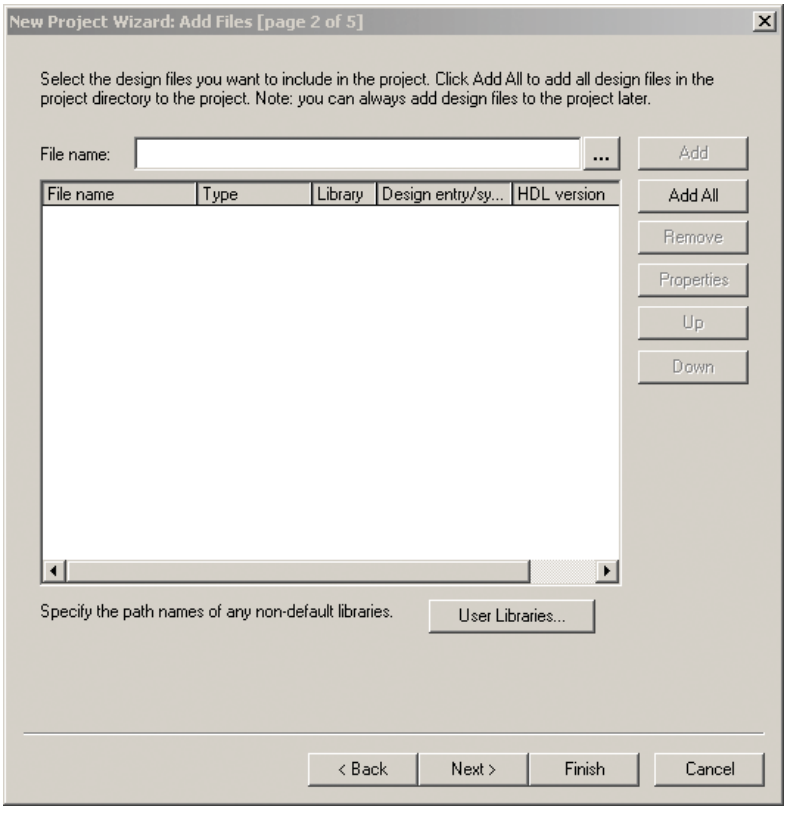


Figure 4–18 The New Project Wizard screen (2 of 5).

5. The third wizard screen is shown in Figure 4–19. This screen will allow us to specify the actual FPGA that we will target for our design. In the drop-down box for the **Family,** select Cyclone II. Place a check in the box for **Specific device**. Highlight the *EP2C35F672C6* and press **Next**.
6. The fourth wizard screen is shown in Figure 4–20. We have no additional EDA tools to use so press **Next** to proceed to the fifth screen.

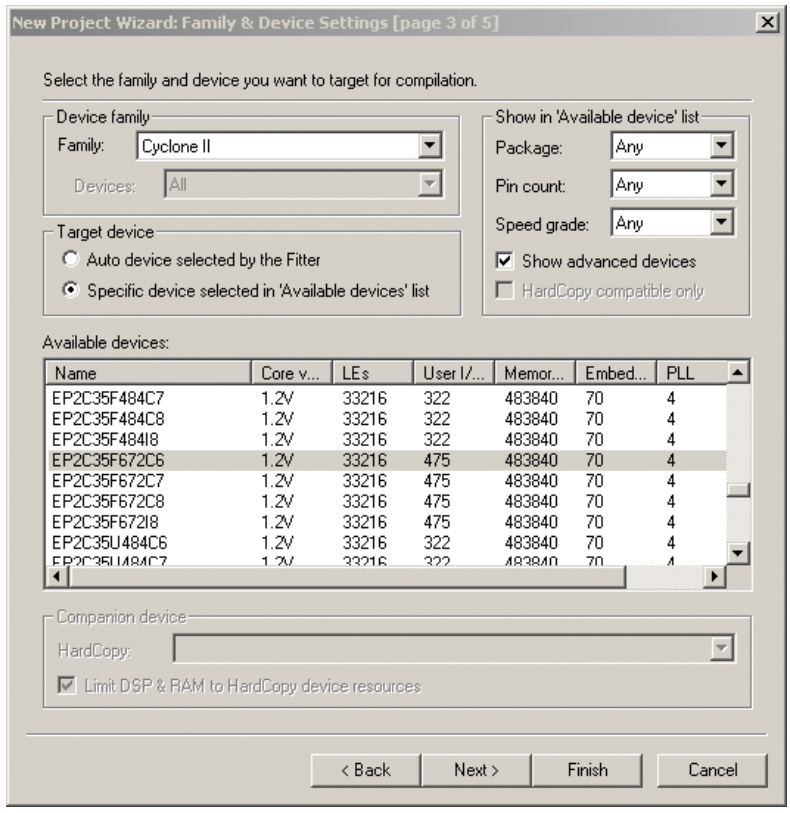


Figure 4–19 The New Project Wizard screen (3 of 5).

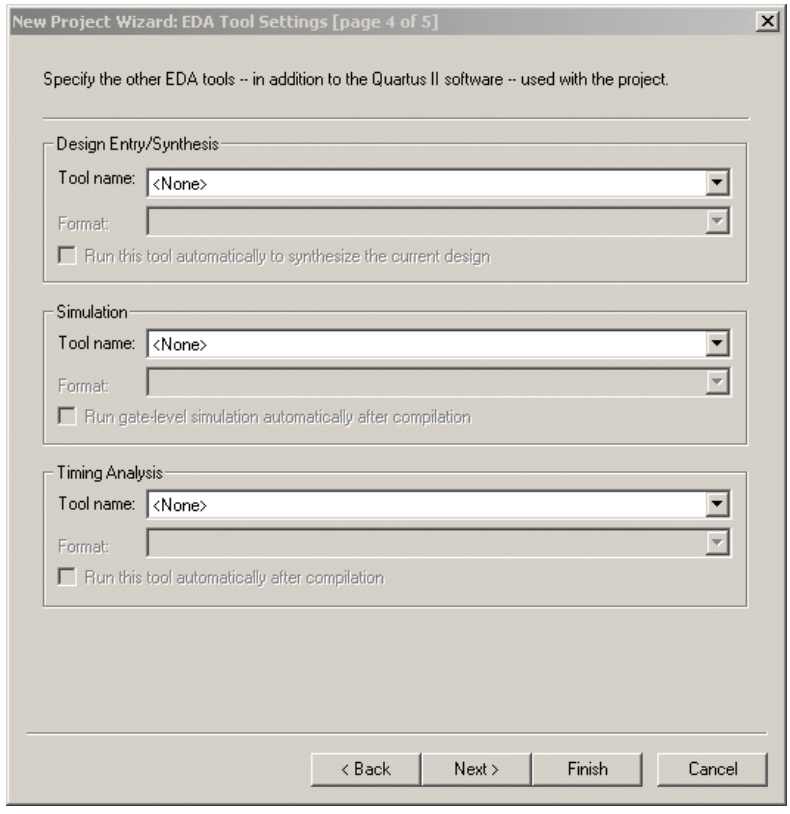


Figure 4–20 The New Project Wizard screen (4 of 5).

7. The fifth wizard screen is shown in Figure 4–21. This shows a summary of all of the choices that we have made. Press **Finish** to complete the New Project Wizard.

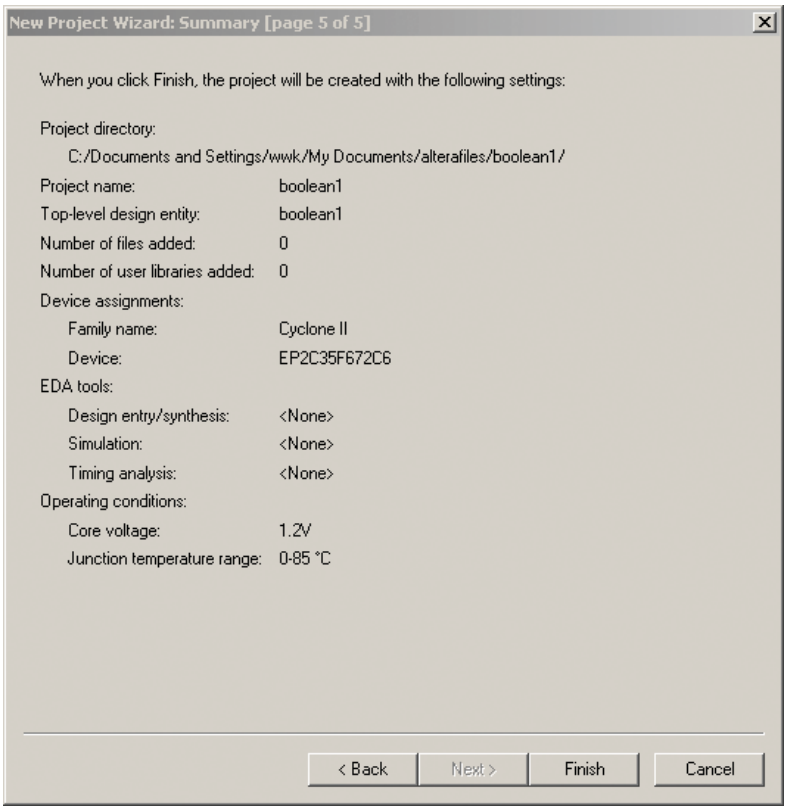


Figure 4–21 The New Project Wizard screen (5 of 5).

Create a Block Design File (*bdf*)

8. To draw the logic circuit for our Boolean equation, we will use the block editor to create a Block Design File by drawing the schematic for the Boolean equation

   $X = AB + CD$.

   Choose **File > New** (see Figure 4–22).

9. Highlight **Block Diagram/Schematic File** and press **OK**. A blank workspace appears. We will draw our digital logic circuit in this workspace.

10. Before drawing the logic circuit we need to name this *bdf* file and save it as part of our project.

    Choose **File > Save As** and enter the **File name** as *boolean1*. Place a *check mark* in the space labeled **Add file to current project** and press **Save** (see Figure 4–23).

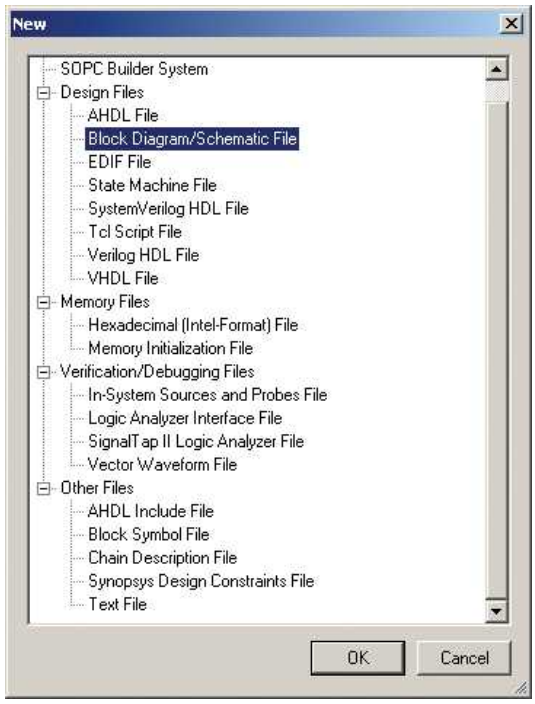


Figure 4–22 The screen used to select a new Block Diagram File.

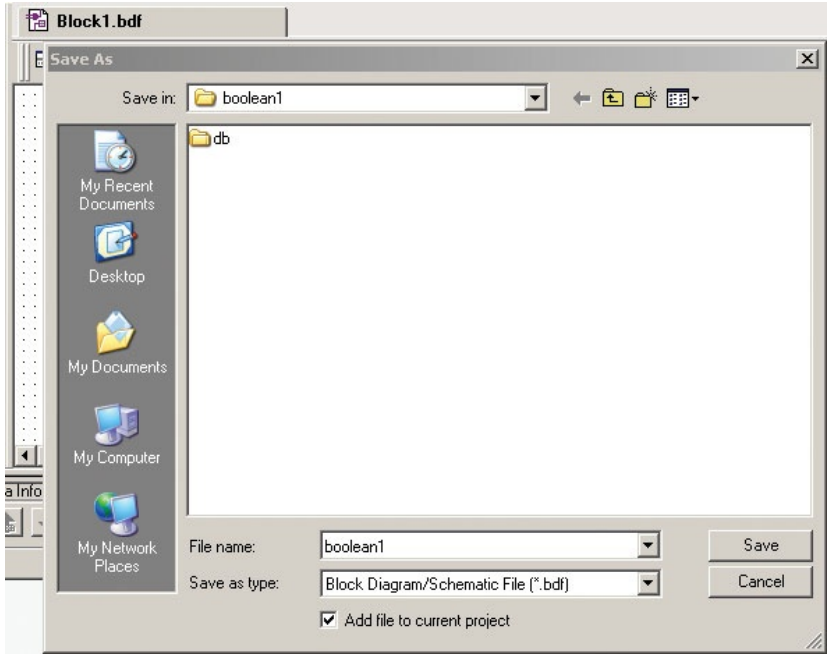


Figure 4–23 Display used to save a new Block Diagram File.

Draw the Digital Logic for the Boolean Equation

11. Right-click the mouse in the empty workspace.

Choose **Insert > Symbol** and type *and2* in the **Name** field and press **OK** (see Figure 4–24).

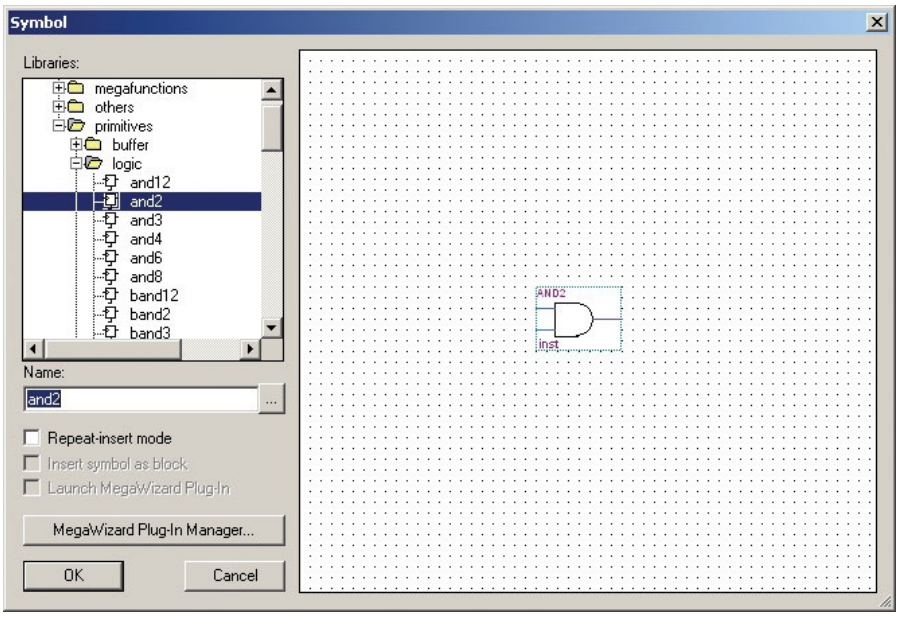


Figure 4–24 Adding a 2-input AND gate to the *bdf* file.

12. Drop the *and2* gate in the *bdf* file workspace by moving your mouse to a suitable location and pressing the left mouse button.

13. To implement the equation $X = AB + CD$ we will need a total of two AND gates and one OR gate. Repeat steps 11 and 12 for another 2-input AND gate (*and2*) and a 2–input OR gate (*or2*).

We also have to provide four input pins for *A*, *B*, *C*, and *D* and one output pin for *X*. Repeat steps 11 and 12 for four input pins (named *input*) and one output pin (named *output*).

The *bdf* workspace should now look like Figure 4–25.

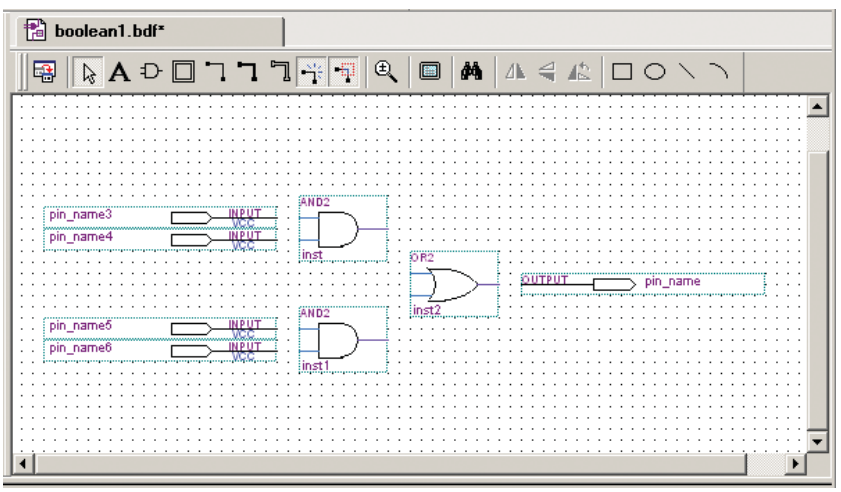


Figure 4–25 Gates and input/output pins inserted into the *bdf* file.

Make the Circuit Connections

14. Before making all of the circuit connections, pin names should be assigned to the four inputs and one output. Double-click on the word *pin_name* inside the first input pin. Enter the lowercase letter *a* for **pin name** and **press OK**. This assigns the name *a* to that pin. Repeat for *b*, *c*, *d*, and *x*.

(*Note:* We use lowercase letters for input and output names to be consistent with the convention used by the VHDL language. We will redesign this logic using VHDL near the end of this tutorial.)

15. We will now make the circuit connections. As you move the mouse pointer close to the end point of any symbol input or output, the pointer automatically becomes a *cross-hair*. This is called the *Smart Drawing Tool*. Press and hold the left mouse button as you drag a connection line from the *a-input* symbol to the input of the first AND gate. Repeat for all of the connections so that the *bdf* file looks like that shown in Figure 4–26.

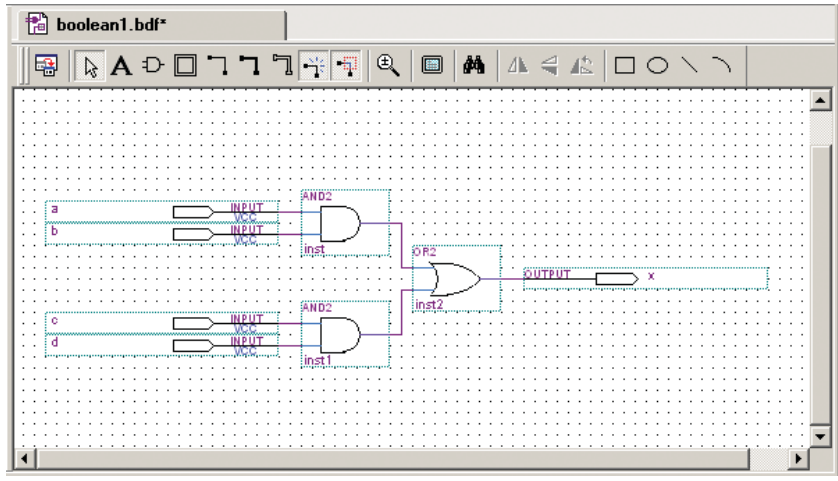


Figure 4–26 The wired *bdf* file.

16. To save the updated *bdf* file:

Choose **File > Save**. (Notice the asterisk is removed from the file name.)

Compiling the Project

17. Now we will compile the project. In this step Quartus® II performs an analysis and synthesis of the *bdf* file to make sure that there are no errors in our logic. It then fits the design to a template of an EP2C35F672C6 FPGA. Finally, it runs an assembler and timing analyzer. To run the compiler:

Choose **Processing > Start Compilation**.

The compilation takes several seconds. When it is complete it should give a message that indicates "Full compilation was successful". (The warnings will be resolved later when we define pin numbers for the input/output) (see Figure 4–27). Press OK.

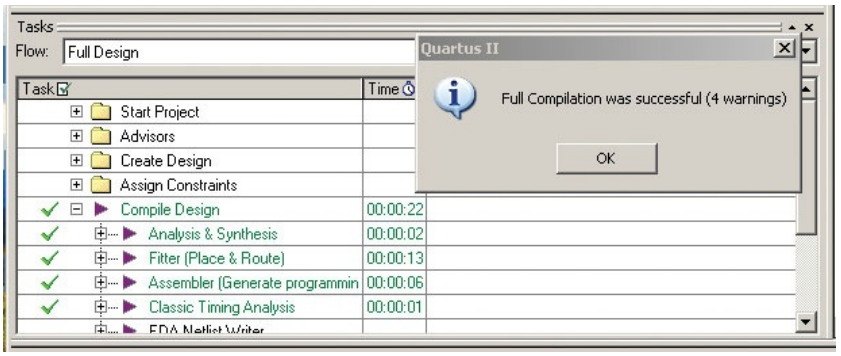


Figure 4–27 Compilation results.

Create a Vector Waveform File (*vwf*) to Simulate the Design*

18. The Vector Waveform File (*vwf*) provides a way for us to draw waveforms that step through all possible combinations of inputs for *a*, *b*, *c*, and *d* and produce the resulting output at *x*. To create a Vector Waveform File:

Choose **File > New > Verification/Debugging Files > Vector Waveform File > OK** (see Figure 4–28).

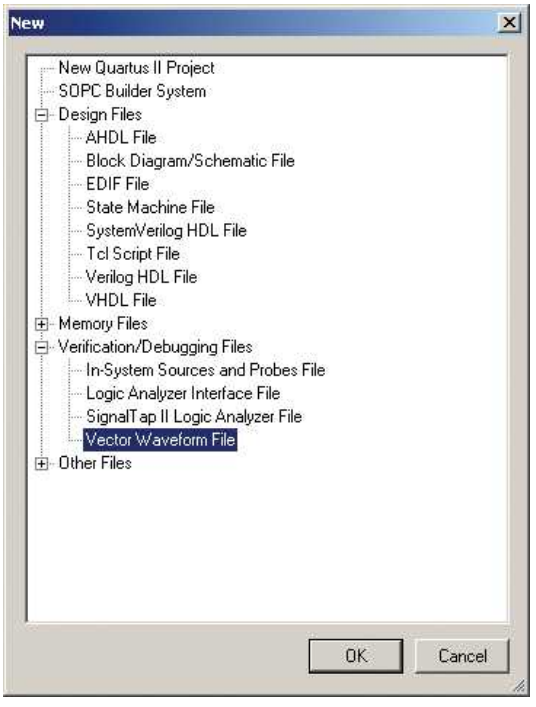


Figure 4–28 The screen used to create a new *vwf* file.

*All *vwf* files in this text were created with **Quartus version 9.1 sp2**. Another alternative to vector waveform simulation is to use **ModelSim®** software. This would require the creation of a VHDL testbench file that could be written after you have a firm understanding of the VHDL language.

19. Before drawing the simulation waveforms we need to name this *vwf* file and save it as part of our project.

Choose **File > Save As** and enter a **file name** of *boolean1*. Place a *check mark* in the space labeled **Add file to current project** and press **Save** (see Figure 4–29).

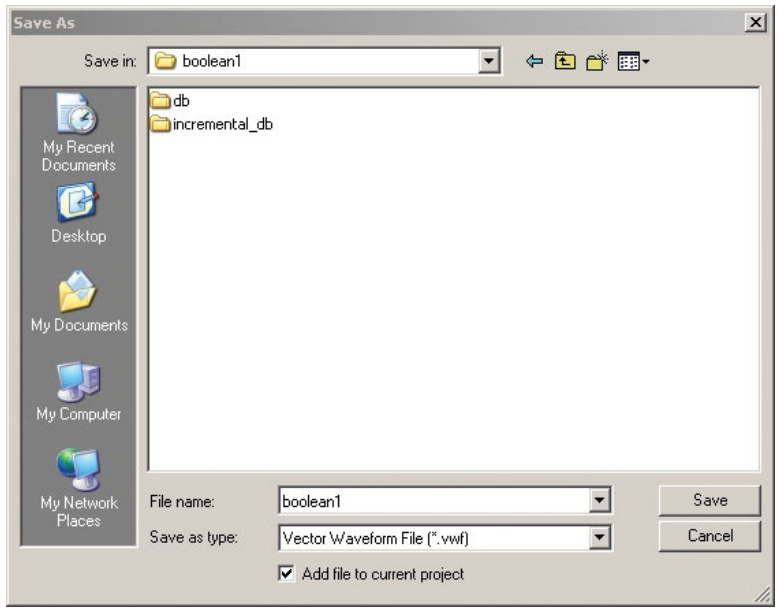


Figure 4–29 The screen display used to save a new Vector Waveform File.

20. To build this simulation file we first need to specify an end time of 16 μs and a grid size of 1 μs for our waveform display:

Choose **Edit > End time** > *16* > *us* > **OK**. Then:

Choose **Edit > Grid Size > Period** > *1* > *us* > **OK** (see Figures 4–30 and 4–31).

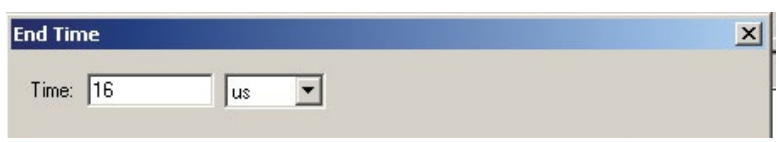


Figure 4–30 Screen used to set the waveform's end time.

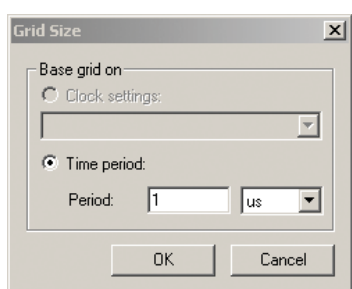


Figure 4–31 Screen used to set the waveform's grid size.

21. To see the entire 16 μs display:

Choose **View > Fit In Window**.

Your *vwf* screen should look like Figure 4–32.

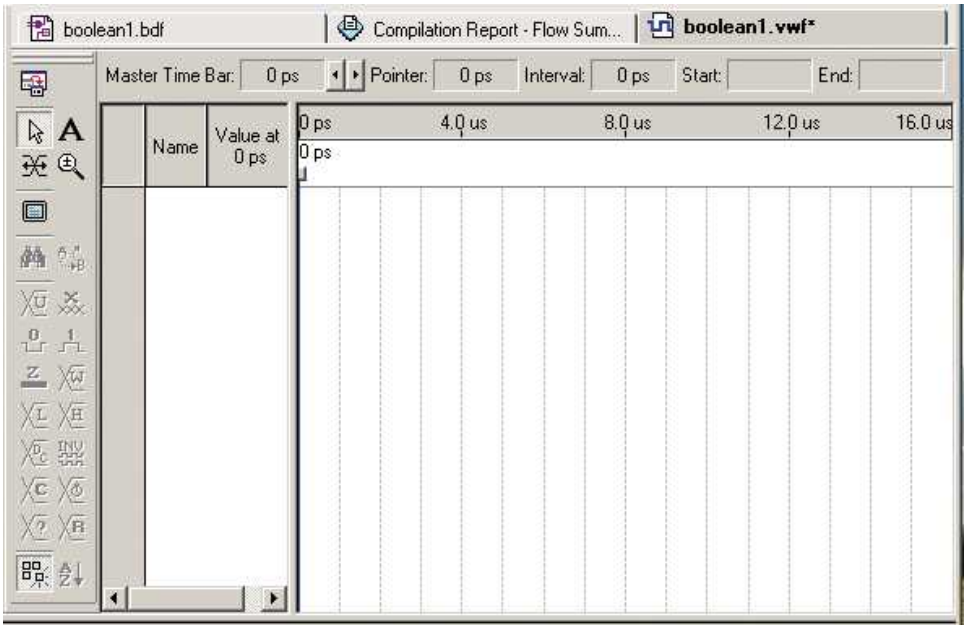


Figure 4–32 The *vwf* screen showing a 16 μs end time and a 1 μs grid size.

Add the Inputs and Outputs to the Waveform (*vwf*) Display

22. We now need to add the inputs and outputs that we want to simulate on the waveform display. The Quartus® II software provides a helpful utility to do this called the "Node Finder."

Choose **View > Utility Windows > Node Finder** (see Figure 4–33).

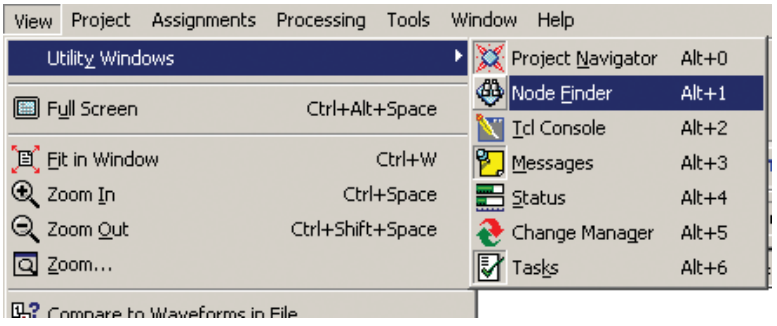


Figure 4–33 Using the Node Finder utility to list inputs and outputs for the *vwf* file.

23. In the Node Finder pop-up window that appears:

Choose **Filter:** *Design Entry (All Names)*.

Press **List** (the display should look like Figure 4–34).

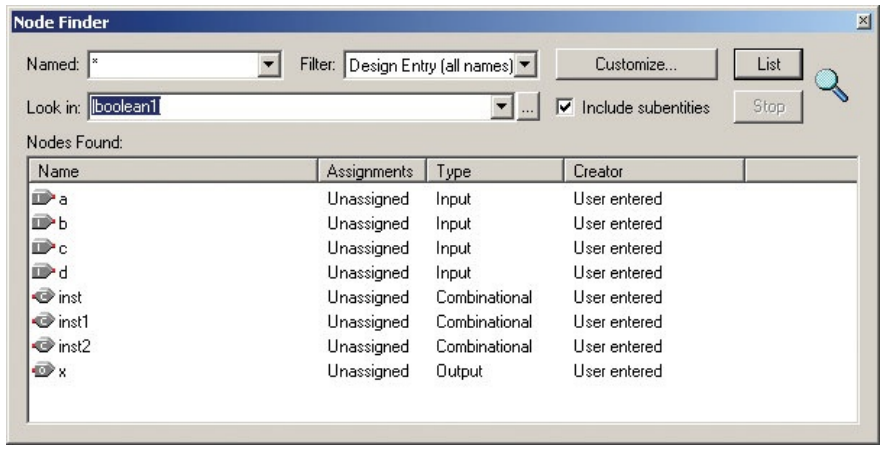


Figure 4–34 The Node Finder screen listing all inputs and outputs of the project.

24. Next we will use the computer mouse to drag the input and output names from the Node Finder screen to the *boolean1.vwf* screen. You can do this by using the mouse to drag each individual input/output with the left mouse button, or you can highlight all five names by holding the CTRL key while you left-click on each of the five input/output names, then drag them all at once (see Figure 4–35).

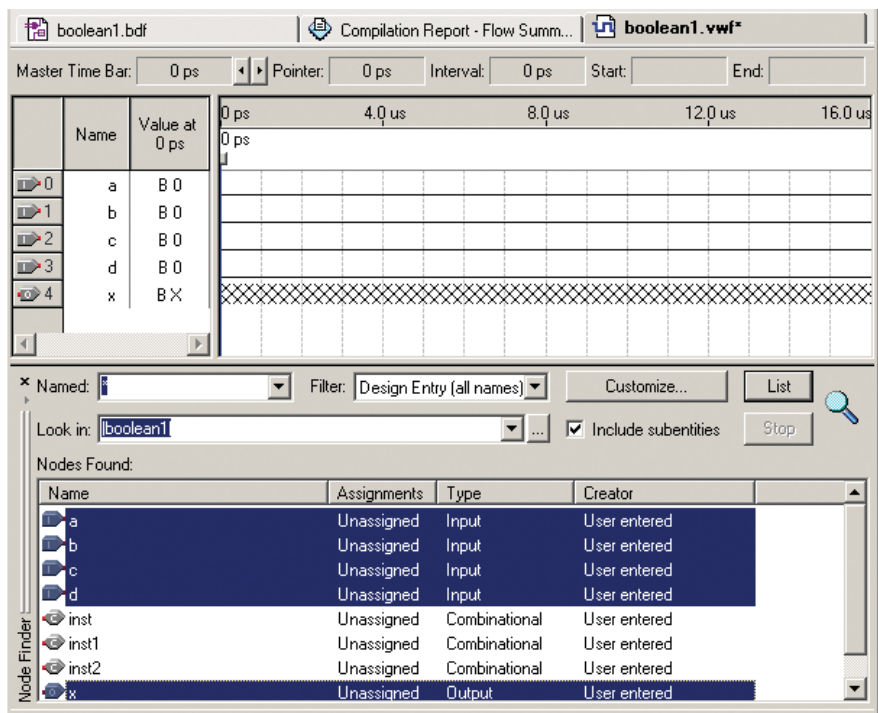


Figure 4–35 Dragging the input/output names from the Node Finder screen to the *vwf* screen.

Create Timing Waveforms for the Inputs

25. In order to test all of the possible combinations for our four inputs we need to create a series of timing waveforms that step through all 16 possible

combinations of input logic levels. The easiest way to do this is to form a binary counter that counts from 0000 up to 1111 just like we did with truth tables in Chapter 3.

In the *vwf* screen, left-click on the first input, *a*, to highlight it.

Choose **Edit > Value > Clock**.

Enter a **period** of *2 us*.

Press **OK**.

The *a*-waveform is shown in Figure 4–36.

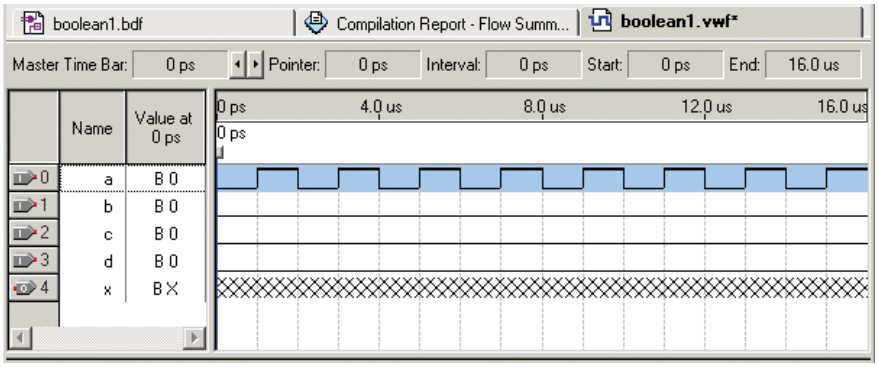


Figure 4–36 The *a*-waveform drawn as a clock with a period of 2 μs.

26. To draw the *b*-waveform as a clock with a period of 4 μs, highlight the *b* input, then:

Choose **Edit > Value > Clock**.

Enter a **period** of *4 us*.

Press **OK**.

27. Repeat for the *c*-waveform (*8 us*) and the *d*-waveform (*16 us*). When completed, the *vwf* screen with all four clock waveforms should look like Figure 4–37.

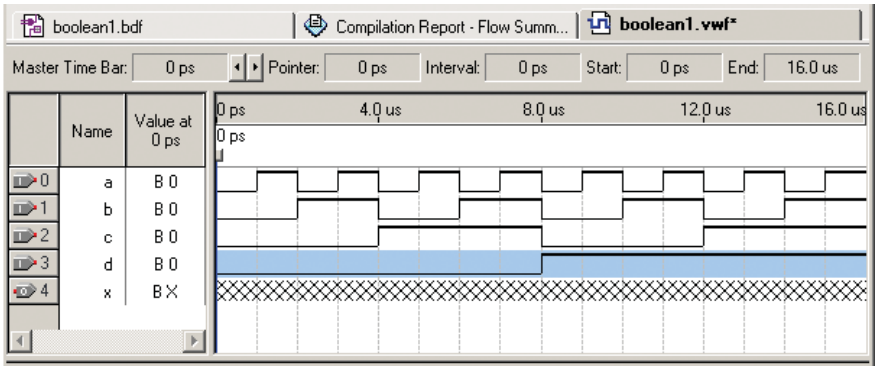


Figure 4–37 Waveforms showing a binary count on the *a*, *b*, *c*, and *d* inputs of the *vwf* file.

28. Save the *vwf* file:

Choose **File > Save**. (Notice the asterisk is removed from the file name.)

Perform a Functional Simulation of the *x*-Output

29. Now that we have the input stimulus defined, the Quartus® II software can use those inputs to determine the level at *x* for each combination of inputs. A functional simulation shows the output waveforms without taking into consideration propagation delays of the internal circuitry. This gives us a simple view of the predicted output so we can check design results.

Choose **Assignments > Settings**.

Then on the left side of the window shown in Figure 4–38 highlight **Simulator Settings**, and for **Simulation Mode** choose **Functional > OK**. Now to create a *netlist* file to enable the simulation:

Choose **Processing > Generate Functional Simulation Netlist > OK**.

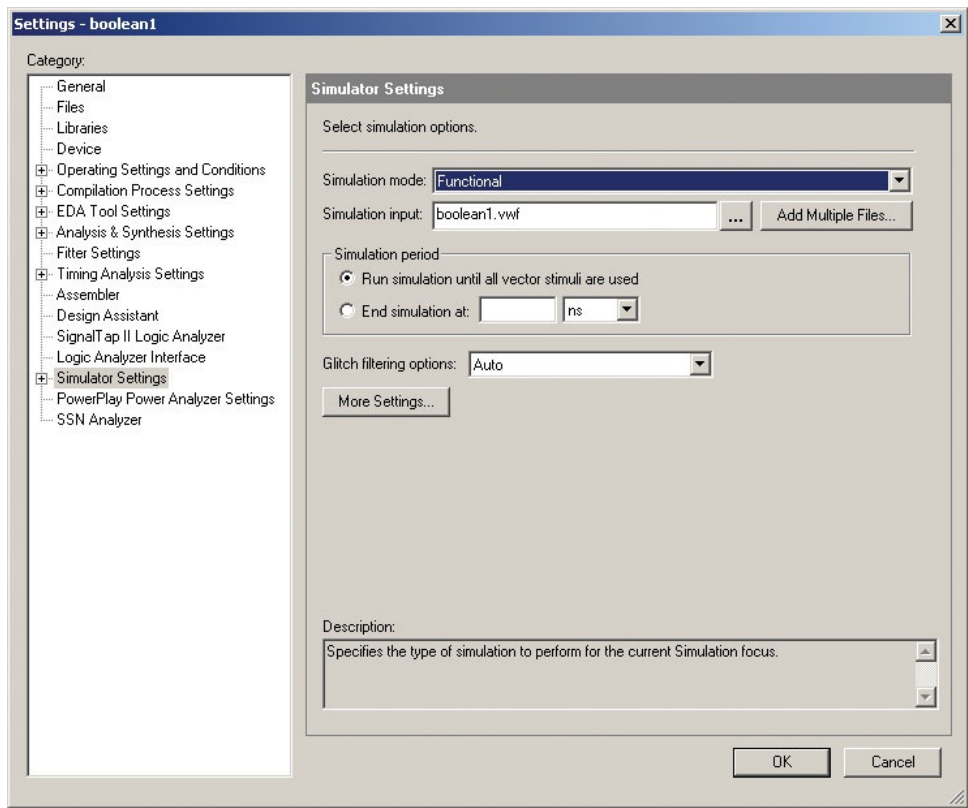


Figure 4–38 The Settings window for specifying the Functional Simulation mode.

30. To process the simulation:

Choose **Processing > Start Simulation**.

After a few moments a message stating "Simulation was successful" should appear.

Press **OK**.

The simulation waveforms are shown in Figure 4–39. (*Note*: You may have to expand the size of the Simulation Waveforms to suit your needs and choose **View > Fit in Window** to see the entire 16 μs waveform.) According to the Boolean equation $X = AB + CD$, X should be HIGH if A AND B are both HIGH OR if C AND D are both HIGH. Study the waveforms to prove to yourself that the simulation shows a valid result.

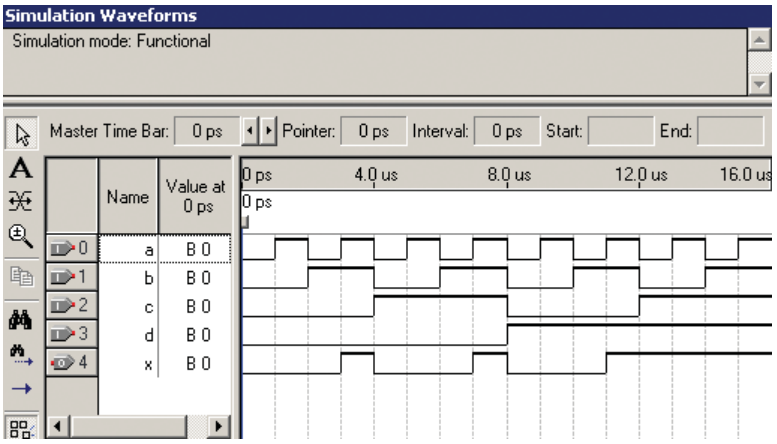


Figure 4–39 Results of the simulation for the Boolean equation $X = AB + CD$.

Programming the FPGA Using the Altera Development and Education Board*

The next step in our development process is to program our logic function into an actual FPGA and test its operation using input switches and an output LED. The development board chosen to perform this task is the Altera DE2. This board has an Altera EPC2C35F672C6 FPGA along with several other I/O devices and memory circuits.

Assigning pins:

31. Previously, when the compiler determined the logic necessary to implement our Boolean equation, it assigned arbitrary pins to our *a*, *b*, *c*, and *d* inputs and our *x* output. However, the DE2 board has several switches, pushbuttons, and LEDs hard-wired to specific pins on the FPGA. Therefore, to exercise our FPGA, we need to assign those specific pin numbers to our inputs and output. Table 4–1 shows a partial list of the pin connections on the FPGA that are hard-wired directly to the I/O on the DE2 board. (A complete list is provided in the DE2 users manual as an Excel *.csv* file.)

TABLE 4–1 EPC2C35F672C6 FPGA Pin Assignments to the DE2 Board (Partial List)

Input Switches		Output LEDs	
Switch Name	**FPGA Pin Number**	**LED Number**	**FPGA Pin Number**
SW0 *A*	N25	LEDR0 *X*	AE23
SW1 *B*	N26	LEDR1	AF23
SW2 *C*	P25	LEDR2	AB21
SW3 *D*	AE14	LEDR3	AC22
SW4	AF14	LEDR4	AD22
SW5	AD13	LEDR5	AD23
SW6	AC13	LEDR6	AD21
SW7	C13	LEDR7	AC21

*The DE2 board is demonstrated in this chapter, but any development board built around an Altera FPGA or CPLD will work.

The pin numbering scheme used in Table 4–1 may seem a little unusual at first, but if you look at the data sheet for our FPGA you see that the IC package is a BGA (Ball Grid Array) set up as 26 rows by 26 columns. The columns are labeled sequentially from 1 to 26, but the rows use the letters *A* through *Y* (skipping *I*, *O*, *Q*, and *X*) then *AA*, *AB*, *AC*, *AD*, *AE*, and *AF*.

Figure 4–40 shows a close-up photograph of the switches and LEDs we will be using. [Inputs *a* and *b* are shown LOW; inputs *c* and *d* are shown HIGH. The red LED used for output *x* (LEDR0) is just above switch SW0.]

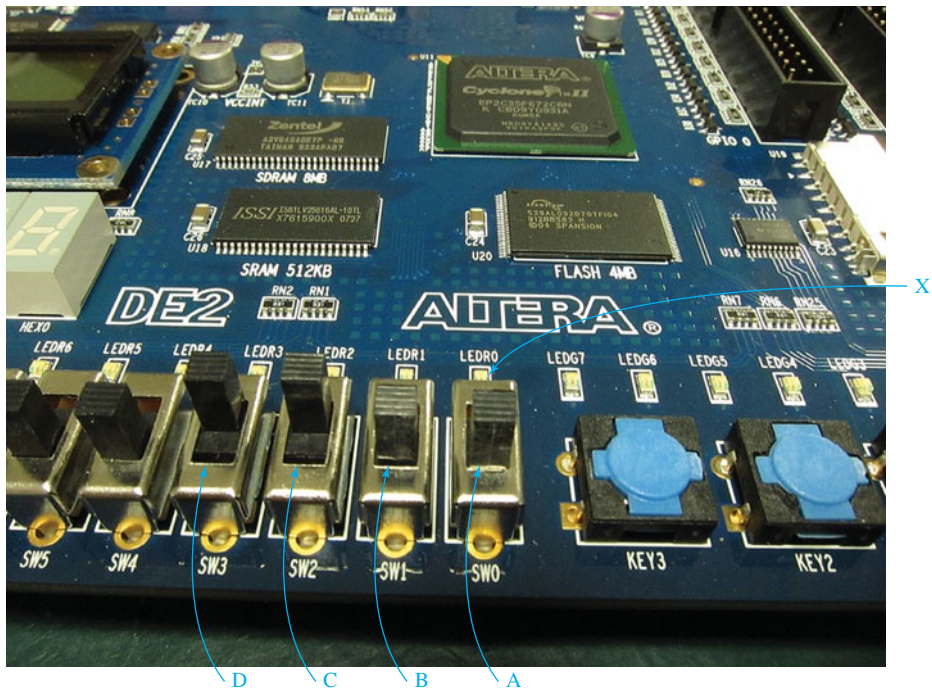


Figure 4–40 DE2 board switches and LED used for testing our Boolean logic.

Pin assignments are made by using the Assignment Editor.

Choose **Assignments > Pins**.

The pin assignment window is shown in Figure 4–41.

In the **Location** column, enter the pin numbers from Table 4–1 for *a*, *b*, *c*, *d*, and *x*. (Shortcut: Just type *N25*, *N26*, etc. in each location.) The completed table is shown in the bottom section of Figure 4–42. The top section of Figure 4–42 shows that the pin assignments were made automatically to the schematic *bdf* file.

Re-compile the project:

32. Now that we have defined specific pin assignments, we need to re-compile the project so that Quartus® will map our logic in the optimum FPGA location and connect the internal input/output to the correct external pins.

Choose **Processing > Start Compilation**.

After a successful compilation, we are ready to program the FPGA.

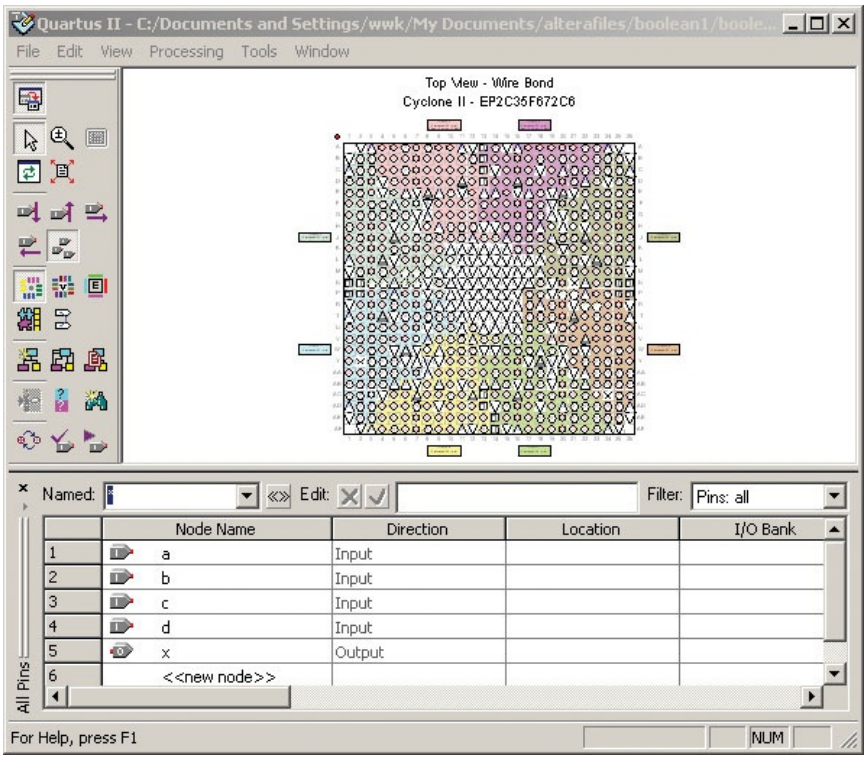


Figure 4–41 The pin assignments window.

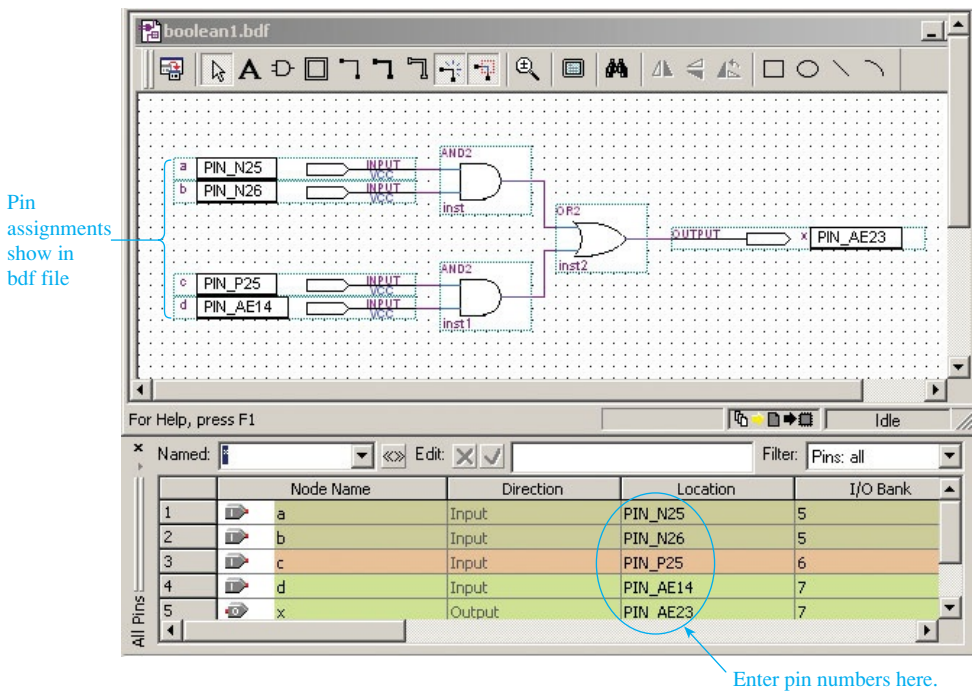


Figure 4–42 The completed pin assignments (bottom) and *bdf* file (top) showing assigned pins.

Program the FPGA on the DE2 board:

33. The final step is to program the FPGA that is on our DE2 board. If this is the first time that this host computer has been used with this software, you will need to follow the instructions in the DE2 user's manual for installing the

USB driver for the DE2 board. This driver facilitates communications with the JTAG interface that is provided on the board. The acronym JTAG stands for Joint Test Action Group. This is an IEEE standard that defines a method for testing and transferring data into digital circuitry.

Connect the USB cable from your board to the host computer and apply power to the DE2.

Choose **Tools > Programmer**.

The programmer window is shown in Figure 4–43. If this is the first time using this host computer for programming FPGAs, you may have to choose **Hardware Setup** to specify that you are using the **USB-Blaster**. Also be sure to select **Mode: JTAG**.

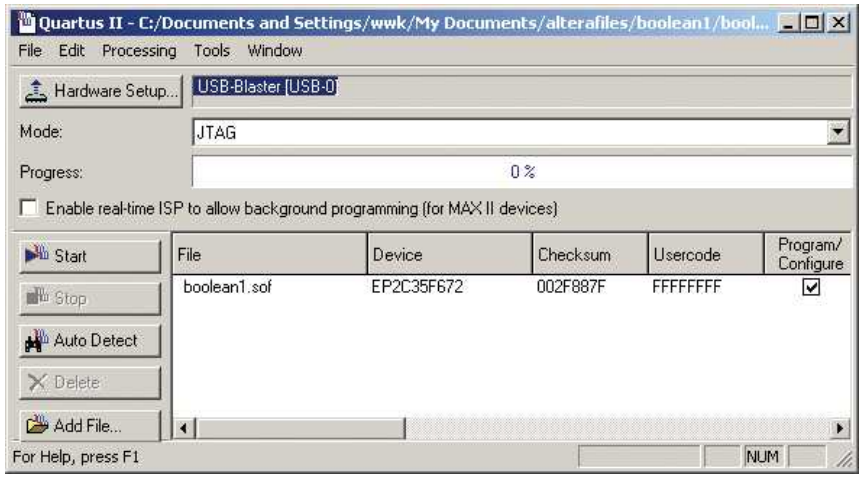


Figure 4–43 The programmer window for downloading our *Boolean.sof* file to the FPGA via the USB-Blaster cable using the JTAG programming mode.

Choose **Start** in the programmer window to begin the programming process.

When the Progress window shows 100%, the device programming is complete, and it is time to test our logic.

Test the logic on the DE2 board:

34. Think back to the Boolean equation that we are implementing: $X = AB + CD$. This means that if A and B are both HIGH or C and D are both HIGH, the LED at X will come on. Test the logic in the FPGA by sliding the appropriate switches. You should see the LED only comes on for a HIGH A and B or a HIGH C and D.

VHDL Design Entry

In this section, we will create the design for *boolean1* ($X = AB + CD$) using the VHDL text editor instead of the block design (schematic) editor. After we define the inputs, outputs, and Boolean equation using the VHDL text editor, we will then recompile the project and check the simulation to be sure that the same output results are implemented.

(*Note:* The following steps assume that you are still working in the *boolean1* project created in steps 1–34. If not, reopen the project by choosing:

File > Recent Projects > *c:* \ … \ *booleanl.*

[If this is a new project to be implemented using VHDL, go back to steps 1–7 to create a new project first.])

35. To get a blank VHDL Text Editor screen:

Choose **File > New > VHDL File > OK** (see Figure 4–44).

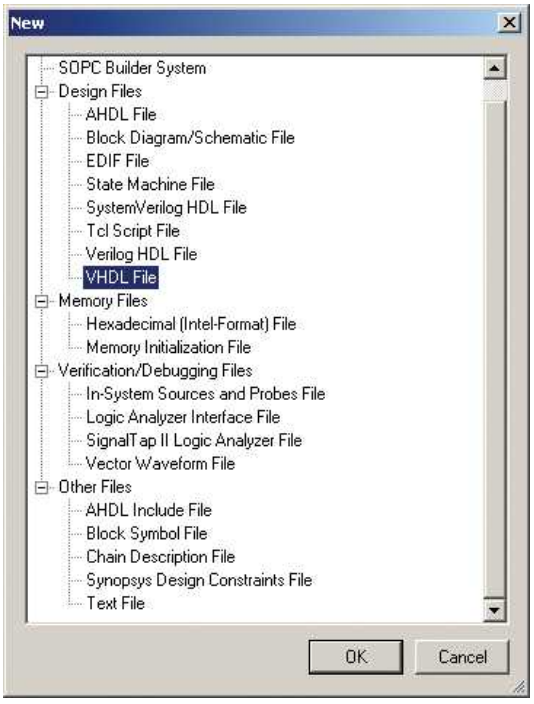


Figure 4–44 Window used to get a blank VHDL text editor screen.

36. Type in the VHDL program for $X = AB + CD$ as shown in Figure 4–45.

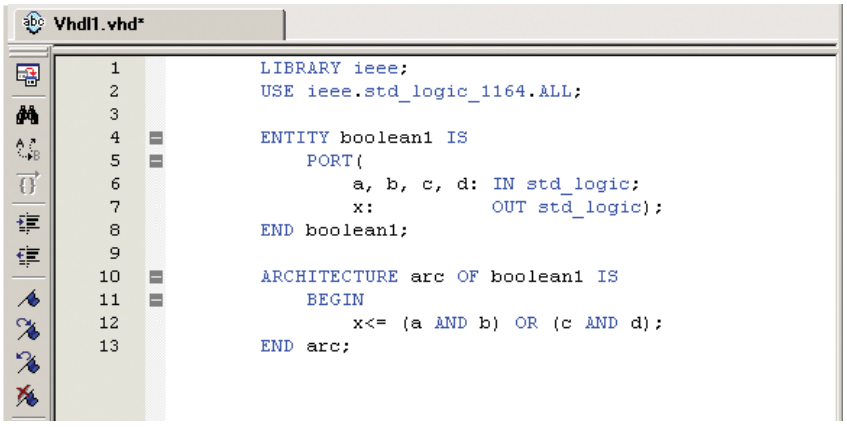


Figure 4–45 The VHDL program listing.

37. To save the VHDL program as part of the current project:

Choose **File > Save As > File name:** ***booleanl.***

Add a check mark next to: **Add file to current project** then press **Save** (see Figure 4–46).

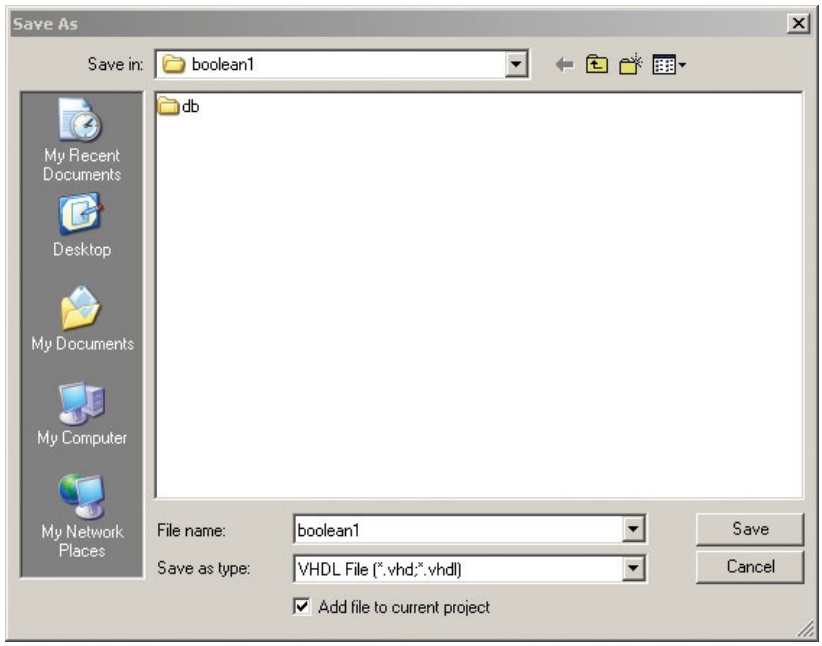


Figure 4–46 Saving the VHDL program as part of the current project.

38. Now we want to compile the program to check for errors. However, since we have already compiled a design for this project using the Block Design File *booleanl.bdf* we need to remove it from the current project or else there will be a conflict error because the project won't know which design to use. To remove the *bdf* file from the project:

Choose **Assignments > Settings**.

Highlight the **Category *Files***.

Highlight the **File name *boolean1.bdf*** then press **Remove > OK** (see Figure 4–47).

(*Note:* This does not delete the *bdf* file from your computer; it only keeps it from being compiled with the *vhd* file and eliminates the conflict that would occur. Later you could use the **Assignments Settings** to **Add** the *bdf* file back and remove the *vhd* file.)

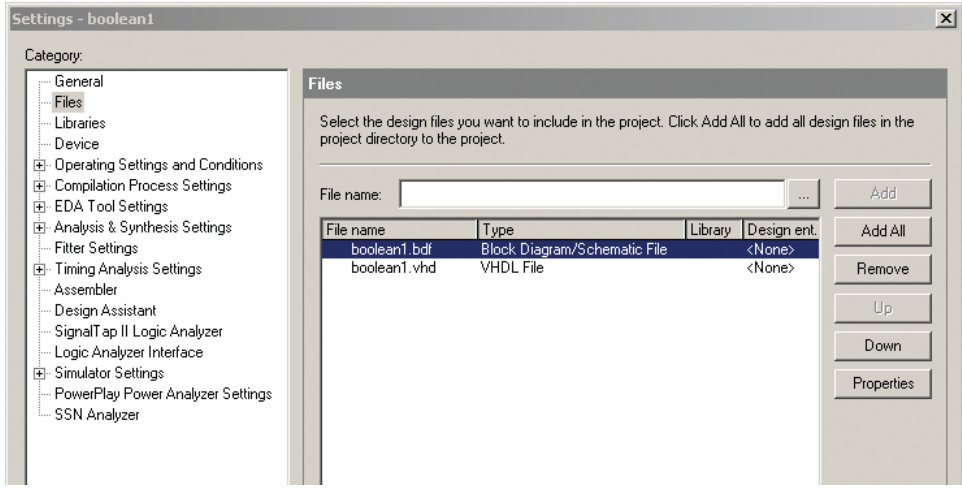


Figure 4–47 Removing the *bdf* file from the current project.

39. To compile the project:

Choose **Processing > Start Compilation**.

After a successful compilation press **OK**.

40. Now you can follow the steps previously outlined to perform a simulation and then program the FPGA IC.

(*Note:* The pin assignments previously made for this project will apply to the design created using VHDL. Also, you don't need to re-create the Vector Waveform File *boolean1.vwf*.) To open the previously created one:

Choose **File > Open > File Name: *boolean1.vwf* > Open.** (*Note:* **Files of type: All files** must be Highlighted to see the *vwf* files as a choice.) Then follow the steps outlined previously for performing a simulation.)

4–5 FPGA Applications

The logic design problems in this section will be solved using the tools provided in the Quartus® II software program. If you haven't already done so, you must work step by step through the tutorial instructions presented in Section 4–4. In each of the examples that follow, your goal is to design the logic circuit, perform a simulation of your circuit, and then, if you have a programmer board, you should download your results and test it on an actual FPGA with switches and LEDs.

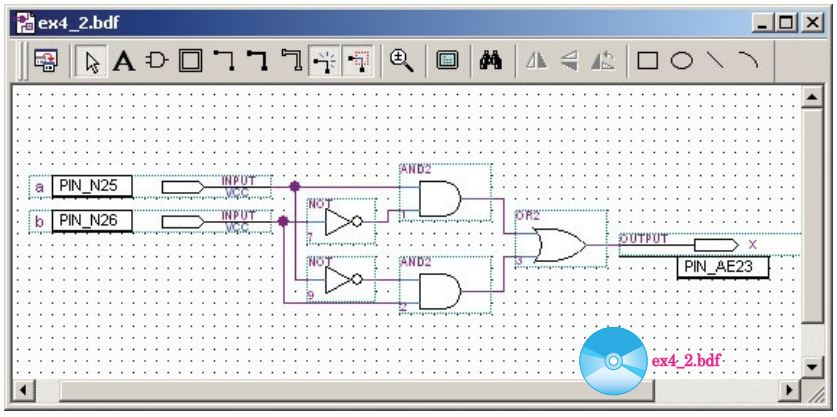
EXAMPLE 4–2

Use Altera Quartus® II software to design the FPGA logic to implement the Boolean equation $X = A\overline{B} + \overline{A}B$.

(a) Design the logic using the block editor to create a Block Design File (*bdf*) called *ex4_2.bdf*.

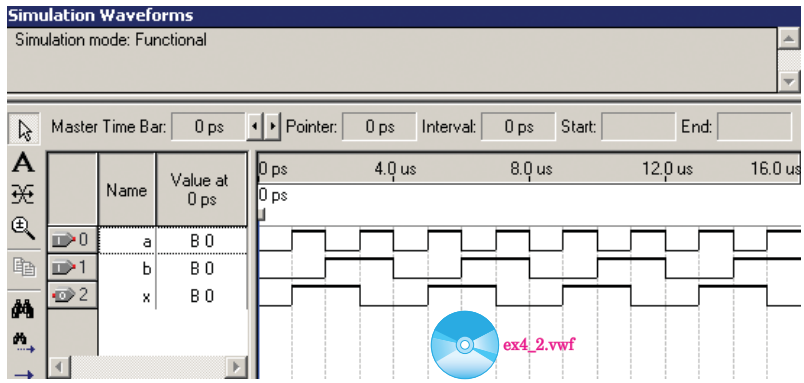
(b) Test the operation of the CPLD logic by using the waveform editor to create a Vector Waveform File (*vwf*) called *ex4_2.vwf*. The simulation should show all possible combinations of inputs.

Solution: The results of the design are shown in Figures 4–48(a) and (b). (The project files for all examples can be found on the textbook companion website.)



(a)

Figure 4–48 Solution to the equation $X = A\overline{B} + \overline{A}B$: (a) Block Design File; (b) Vector Waveform File.



(b)

Figure 4–48 *Continued*

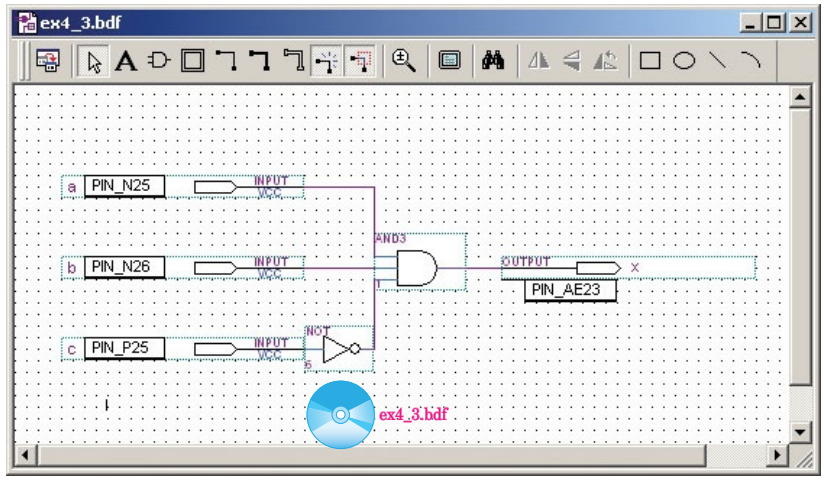
EXAMPLE 4–3

Use Altera Quartus® II software to design the FPGA logic to implement the Boolean equation $X = AB\overline{C}$.

(a) Design the logic using the block editor to create a Block Design File (*bdf*) called *ex4_3.bdf.*

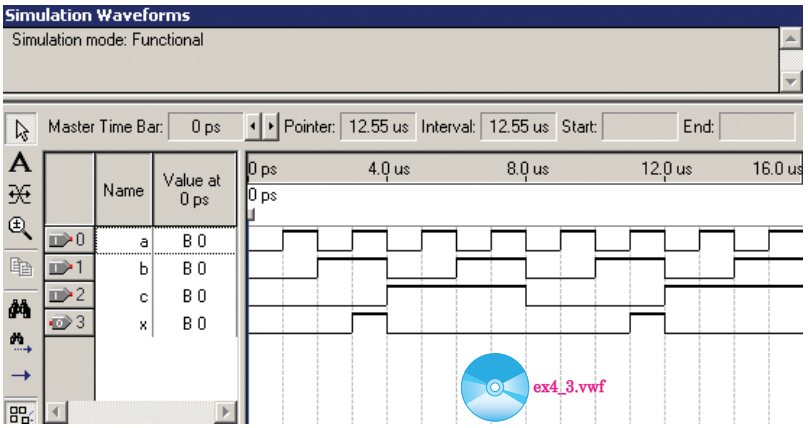
(b) Test the operation of the FPGA logic by using the waveform editor to create a Vector Waveform File (*vwf*) called *ex4_3.vwf.* The simulation should show all possible combinations of inputs.

Solution: The results of the design are shown in Figures 4–49(a) and (b). (The *bdf* and *vwf* files can also be found on the textbook companion website.)



(a)

Figure 4–49 Solution to the equation $X = AB\overline{C}$: (a) Block Design File; (b) Vector Waveform File.



(b)

Figure 4–49 *Continued*

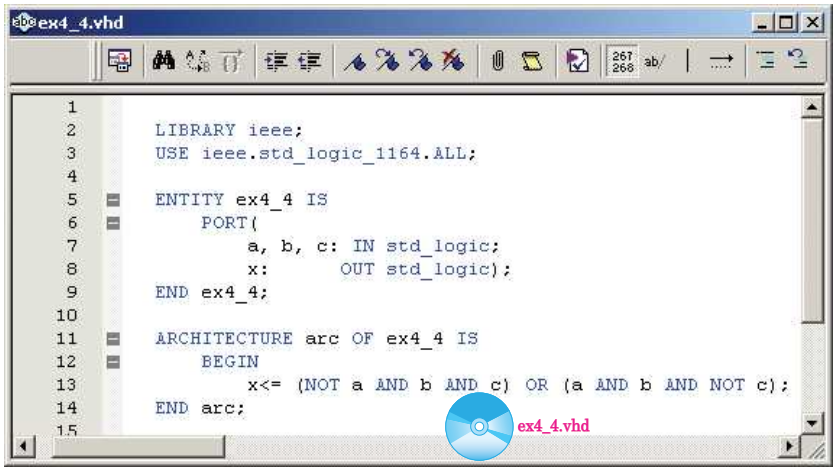
EXAMPLE 4–4

Use Altera Quartus® II software to design the FPGA logic to implement the Boolean equation $X = \overline{A}BC + A\overline{BC}$.

(a) Design the logic using the block editor to create a VHDL File (*vhd*) called *ex4_4.vhd*.

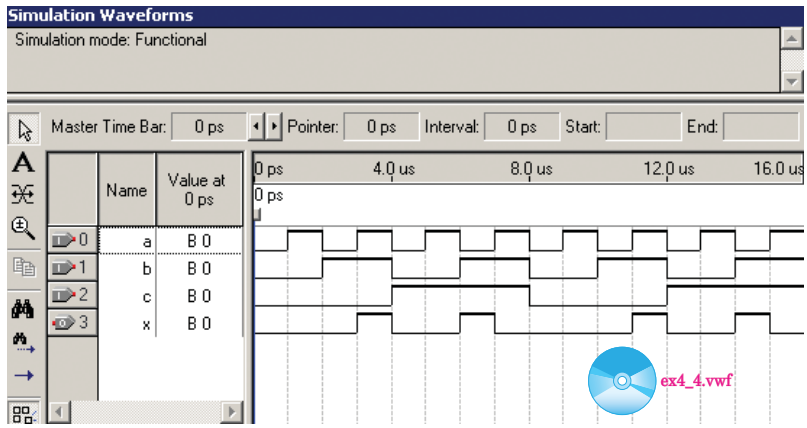
(b) Test the operation of the FPGA logic by using the waveform editor to create a Vector Waveform File (*vwf*) called *ex4_4.vwf*. The simulation should show all possible combinations of inputs.

Solution: The results of the design are shown in Figures 4–50(a) and (b). (The *vhd* and *vwf* files can also be found on the textbook companion website.)



(a)

Figure 4–50 Solution to the eqution $X = \overline{A}BC + A\overline{BC}$: (a) VHDL program; (b) Vector Waveform File.



(b)

Figure 4–50 *Continued*

Summary

In this chapter, we have learned that

1. PLDs can be used to replace 7400- and 4000-series ICs. They contain the equivalent of thousands of logic gates. CAD tools are used to configure them to implement the desired logic.

2. The two most common methods of PLD design entry are (graphic) entry and VHDL entry. To use graphic entry, the designer uses CAD tools to draw the logic to be implemented. To use VHDL entry, the designer uses a text editor to write program descriptions defining the logic to be implemented.

3. PLD design software usually includes a logic simulator. This feature allows the user to simulate levels to be input to the PLD, and it shows the output simulation to those input conditions.

4. Most PLDs are erasable and reprogrammable. This allows users to test many versions of their logic design without ever changing ICs.

5. Basically, there are four types of PLDs: SPLDs, CPLDs, FPGAs, and ASICs. SPLDs use the PAL or PLA architecture. They consist of several multiinput AND gates whose outputs feed the inputs to OR gates and memory flip-flops. CPLDs consist of several interconnected SPLDs. FPGAs are the most dense form of PLD, solving logic using a look-up table to determine the desired output. ASICs are functionally equivalent to FPGAs but their logic is permanently hard-coded into the IC.

Glossary

Architecture Body: The section in a VHDL program defining the logic functions to be implemented.

ASIC (application-specific integrated circuit): ASICs are functionally equivalent and pin compatible with their sister FPGA. Used for large quantity

applications, their logic is hard-coded, making them a non-volatile version of an FPGA.

Block Editor: A software tool provided as part of the PLD development package. It provides a way to enter designs by drawing a schematic to create a Block Design File.

CAD: Computer-Aided Design. This type of design uses a computer to aid in the drawing and logic development of a logic circuit. It eliminates many of the manual, time-consuming tasks once associated with logic design.

CPLD: Complex Programmable Logic Device. A PLD consisting of more than 100 interconnected SPLDs. A single chip can be programmed to implement hundreds of logic equations and operations.

Compiler: A language translation software module used by CPLD development systems to convert a schematic or VHDL code into a binary file to represent the digital logic to be implemented.

Entity Declaration: The section of a VHDL program defining the input and output ports.

FPGA: Field-Programmable Gate Array: The most dense form of PLD. It uses a look-up table to resolve its logic operations. Its main disadvantage is that most FPGAs are volatile, losing their memory when power is removed.

Library Declaration: The section of a VHDL program declaring the software libraries to be included in the program. These libraries are used by the compiler to resolve references to the various program commands.

Look-Up Table: Used by FPGA logic to determine the output level of a circuit based on the combinations of logic levels at its inputs. It is constructed as a truth table except that its outputs are only HIGH for specific combinations of inputs solving the given logic product terms.

Nonvolatile: Internal memory is maintained even when power is removed from the IC.

PAL: Programmable Array Logic: Its basic structure contains multiple inputs to several AND gates, the outputs of which are connected to a series of fixed ORs.

PLA: Programmable Logic Array: Its basic structure contains multiple inputs to several AND gates, the outputs of which are connected to a series of programmable ORs.

PLD: Programmable Logic Device: An IC containing thousands of undefined logic functions. A software development tool is used to specify (i.e., program) the specific logic to be implemented by the IC. PLD is the general term used to represent PLAs, PALs, SPLDs, CPLDs, and FPGAs.

Product Terms: Input variables that are ANDed together (e.g., ABC, $AB\overline{C}$).

Schematic Capture: A method used by PLD software to input a design that is defined by a schematic.

SPLD: Simple Programmable Logic Devices: A programmable, digital logic IC containing several PAL or PLA structures with internal interconnections and memory registers.

Sum-of-Products (SOP): Two or more product terms that are ORed together (e.g., $AB\overline{C} + A\overline{CD} + BCD$).

Synthesize: The creation of a model of the PLD's internal electrical connections that will produce the actual logic functions defined by the user.

VHDL: VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. A programming language used by PLD software to define a logic design by specifying a series of I/O definitions and logic equations.

VHDL Editor: A software program facilitating entry of text-based instructions comprising the VHDL program.

Waveform Simulator: The part of a PLD software development tool that allows users to simulate the input of several signals to a logic circuit and observe its response in a Vector Waveform File.

Problems

Section 4–1

4–1. How does programmable logic differ from discrete digital logic like the 7400 series?

4–2. What are two common ways to configure or define logic to PLD programming software?

4–3. What does HDL stand for in the acronym VHDL?

4–4. List the six steps in the PLD design flow.

4–5. How many different ICs would it take to implement the following equations?

(a) $X = AB + \overline{B\overline{C}}$

(b) $Y = \overline{A\overline{B}} + B\overline{C} + \overline{\overline{C + D}}$

4–6. How is pin 1 identified in the PLCC package style used for the PLD in Figure 4–4?

4–7. What is the purpose of the PLD programmer boards shown in Figure 4–5?

Section 4–2

4–8. How many product terms are in the following equations?

(a) $X = A\overline{C} + BC + \overline{A}C$

(b) $Y = A\overline{B}C + B\overline{C}$

(c) $Z = AB\overline{C} + ACD + \overline{B}C\overline{D}$

4–9. How does a PLA differ from a PAL?

4–10. Redraw the PLA circuitry of Figure 4–8 to implement the following SOP equations:

(a) $X = \overline{A}B + \overline{A}\,\overline{B} + A\overline{B}$

(b) $Y = AB + \overline{A}B$

4–11. Why is it advantageous to use a CPLD or ASIC that is nonvolatile?

4–12. Refer to the data sheets in Appendix B (or the manufacturer's Web site) to determine the number of usable gates and macrocells in each of the following CPLDs:

(a) Altera MAX EPM7128S

(b) Xilinx XC95108

4–13. Instead of interconnecting logic gates, the FPGA solves its logic requirements by using what method?

4–14. Draw a 2-input look-up table (LUT) similar to Figure 4–11(b) for the equation $X = \overline{A}\,\overline{B} + A\overline{B}$.

4–15. Because most FPGAs are volatile, what must be done each time they are powered up?

Section 4–3

4–16. What are the two most common methods of design entry for FPGA development software?

4–17. What is the function of the compiler in FPGA development software?

4–18. What is the purpose of the three pin stubs in the *bdf* file shown in Figure 4–13(a)?

4–19. VHDL allows the user to enter the logic design via a _____ editor.

4–20. Define the purpose of the following three VHDL program segments:

(a) Library

(b) Entity

(c) Architecture

4–21. Write the VHDL entity declare for a three-input AND gate.

4–22. Write the VHDL architecture for a three-input AND gate.

4–23. Draw the logic circuit to be implemented by the following VHDL architecture body:

```
ARCHITECTURE arc OF p4_23 IS
BEGIN
x <= (a AND (b OR c));
y <= (a OR NOT b) AND NOT (b AND c);
z <= NOT (b AND c) OR NOT (a OR c);
END arc;
```

FPGA Problems

The following problems will be solved using the Altera Quartus® II software. You will be asked to solve the design using the block design entry method or the VHDL design entry method. In either case you will demonstrate the circuit operation by producing a Vector Waveform File (*vwf*) that exercises all possible inputs to your circuit. The final step, if you have a programmer board like the DE-2, is to download your design to an FPGA and demonstrate its operation to your instructor.

Section 4–4

C4–1. Use an FPGA to implement the following Boolean equation: $X = A\overline{B}$.

(a) Create a Block Design File called *prob_c4_1.bdf* to define the logic circuit.

(b) Create a Vector Waveform File called *prob_c4_1.vwf* to test the operation of your design by showing the output waveform for all possible input conditions.

(c) Build a truth table for the Boolean equation.

(d) Download the design to the FPGA on your programmer board and demonstrate its operation by monitoring the output LED as you step through all switch combinations shown in your truth table from part (c).

C4–2. Use an FPGA to implement the following Boolean equation: $X = AB + \overline{A}\,\overline{B}$.

(a) Create a Block Design File called *prob_c4_2.bdf* to define the logic circuit.

(b) Create a Vector Waveform File called *prob_c4_2.vwf* to test the operation of your design by showing the output waveform for all possible input conditions.

(c) Build a truth table for the Boolean equation.

(d) Download the design to the FPGA on your programmer board and demonstrate its operation by monitoring the output LED as you step through all switch combinations shown in your truth table from part (c).

C4–3. Use an FPGA to implement the following Boolean equation: $X = A\overline{B}C$.

(a) Create a Block Design File called *prob_c4_3.bdf* to define the logic circuit.

(b) Create a Vector Waveform File called *prob_c4_3.vwf* to test the operation of your design by showing the output waveform for all possible input conditions.

(c) Build a truth table for the Boolean equation.

(d) Download the design to the FPGA on your programmer board and demonstrate its operation by monitoring the output LED as you step through all switch combinations shown in your truth table from part (c).

C4–4. Use an FPGA to implement the following Boolean equation: $X = A\overline{B}C + \overline{A}\,\overline{B}C$.

(a) Create a VHDL File called *prob_c4_4.vhd* to define the logic circuit.

(b) Create a Vector Waveform File called *prob_c4_4.vwf* to test the operation of your design by showing the output waveform for all possible input conditions.

(c) Build a truth table for the Boolean equation.

(d) Download the design to the FPGA on your programmer board and demonstrate its operation by monitoring the output LED as you step through all switch combinations shown in your truth table from part (c).