A-PDF Split DEMO : Purchase from www.A-PDF.com to remove the watermark

CHAPTER 2

OVERVIEW OF FPGA AND EDA SOFTWARE

2.1 INTRODUCTION

Developing a large FPGA-based system is an involved process that consists of many complex transformations and optimization algorithms. Software tools are needed to automate some of the tasks. We use the Web version of the Xilinx *ISE* package for synthesis and implementation, and use the starter version of Mentor Graphics *ModelSim XE III* package for simulation. In this chapter, we give a brief overview of the FPGA device and the S3 prototyping board, and provide short tutorials for the two software packages to "jump-start" the learning process.

2.2 FPGA

2.2.1 Overview of a general FPGA device

A *field programmable gate array* (FPGA) is a logic device that contains a two-dimensional array of generic logic cells and programmable switches. The conceptual structure of an FPGA device is shown in Figure 2.1. A logic cell can be configured (i.e., *programmed*) to perform a simple function, and a programmable switch can be customized to provide interconnections among the logic cells. A custom design can be implemented by specifying the function of each logic cell and selectively setting the connection of each programmable switch. Once the design and synthesis is completed, we can use a simple adaptor cable to download the desired logic cell and switch configuration to the FPGA device and obtain the

Copyright © 2008 John Wiley & Sons, Inc.

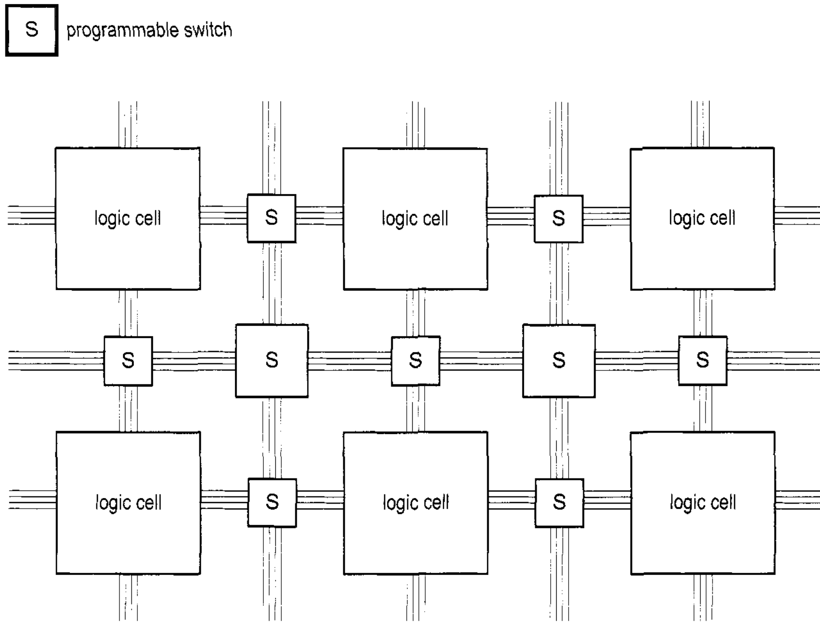
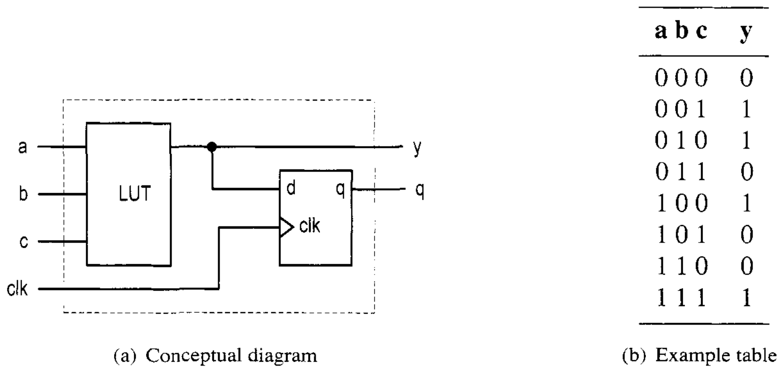


Figure 2.1 Conceptual structure of an FPGA device.



(a) Conceptual diagram

a b c	y
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	1
1 0 1	0
1 1 0	0
1 1 1	1

(b) Example table

Figure 2.2 Three-input LUT-based logic cell.

custom circuit. Since this process can be done "in the field" rather than "in a fabrication facility (fab)," the device is known as *field programmable.*

LUT-based logic cell A logic cell usually contains a small configurable combinational circuit with a D-type flip-flop (D FF). The most common method to implement a configurable combinational circuit is a *look-up table* (LUT). An n-input LUT can be considered as a small 2^n-by-1 memory. By properly writing the memory content, we can use the LUT to implement any n-input combinational function. The conceptual diagram of a three-input LUT-based logic cell is shown in Figure 2.2(a). An example of three-input LUT implementation of $a \oplus b \oplus c$ is shown in Figure 2.2(b). Note that the output of the LUT

can be used directly or stored to the D FF. The latter can be used to implement sequential circuits.

Macro cell Most FPGA devices also embed certain *macro cells* or *macro blocks*. These are designed and fabricated at the transistor level, and their functionalities complement the general logic cells. Commonly used macro cells include memory blocks, combinational multipliers, clock management circuits, and I/O interface circuits. Advanced FPGA devices may even contain one or more prefabricated processor cores.

2.2.2 Overview of the Xilinx Spartan-3 devices

This book uses Xilinx Spartan-3 family FPGA devices. Based on the ratio between the number of logic cells and the I/O counts, the family is further divided into several subfamilies. Our discussion applies to all the subfamilies.

Logic cell, slice, and CLB The most basic element of the Spartan-3 device is a *logic cell* (LC), which contains a four-input LUT and a D FF, similar to that in Figure 2.2. In addition, a logic cell contains a carry circuit, which is used to implement arithmetic functions, and a multiplexing circuit, which is used to implement wide multiplexers. The LUT can also be configured as a 16-by-1 static random access memory (SRAM) or a 16-bit shift register.

To increase flexibility and improve performance, eight logic cells are combined together with a special internal routing structure. In Xilinx terms, two logic cells are grouped to form a *slice*, and four slices are grouped to form a *configurable logic block* (CLB).

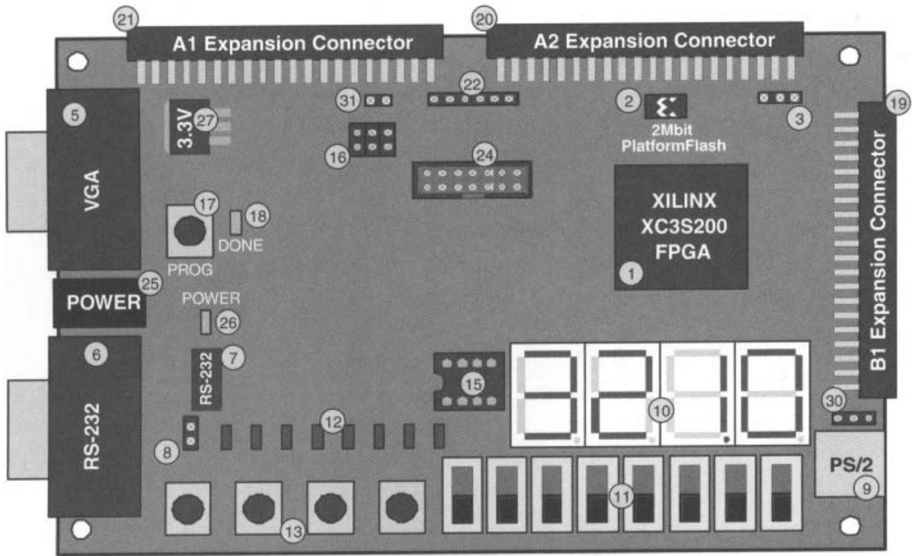
Macro cell The Spartan-3 device contains four types of macro blocks: combinational multiplier, *block RAM*, *digital clock manager* (DCM), and *input/output block* (IOB). The combinational multiplier accepts two 18-bit numbers as inputs and calculates the product. The block RAM is an 18K-bit synchronous SRAM that can be arranged in various types of configurations. A DCM uses a digital-delayed loop to reduce clock skew and to control the frequency and phase shift of a clock signal. An IOB controls the flow of data between the device's I/O pins and the internal logic. It can be configured to support a wide variety of I/O signaling standards.

Devices in the Spartan-3 subfamily Althopugh Spartan-3 FPGA devices have similar types of logic cells and macro cells, their densities differ. Each subfamily contains an array of devices of various densities. The numbers of LCs, block RAMs, multipliers, and DCMs of the devices from the Spartan-3 subfamily are summarized in Table 2.1.

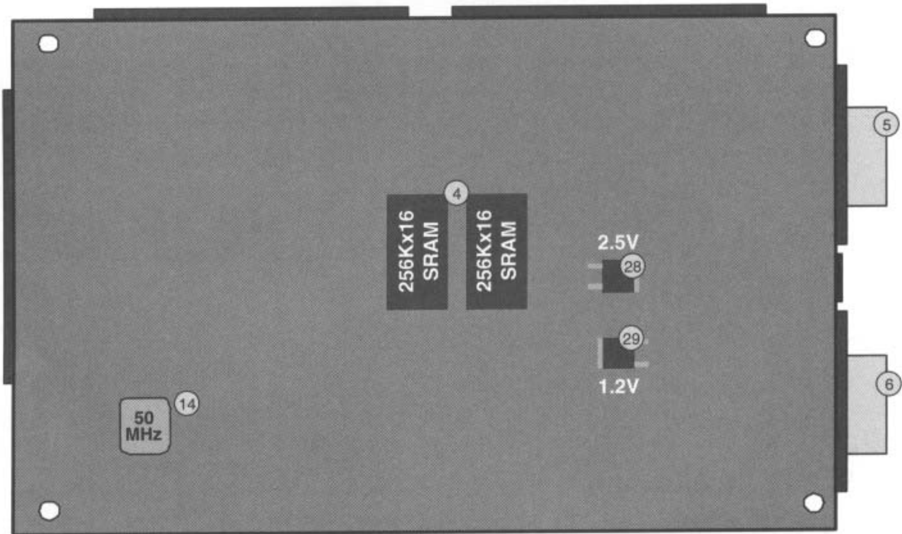
2.3 OVERVIEW OF THE DIGILENT S3 BOARD

The Digilent S3 board is based on a Spartan-3 device (usually an XC3S200) and has an array of built-in peripherals. The simplified layouts of the board are shown in Figure 2.3(a) and (b). The main components and connectors are as follows:

1. Xilinx Spartan-3 XC3S200 FPGA device (XC3S200FT256)
2. 2M-bit Xilinx XCF02S platform flash configuration PROM
3. Jumper to select the configuration source
4. Two 256K-by-16 asynchronous SRAM devices (ISSI IS61LV25616AL-10T).



(a) Top view



(b) Bottom view

Figure 2.3 Layout of an S3 board. (Courtesy of Xilinx, Inc. © Xilinx, Inc. 1994–2007. All rights reserved.)

Table 2.1 Devices in the Spartan-3 family

Device	Number of LCs	Number of block RAMs	Block RAM bits	Number of multipliers	Number of DCMs
XC3S50	1,728	4	72K	4	2
XC3S200	4,320	12	216K	12	4
XC3S400	8,064	16	288K	16	4
XC3S1000	17,280	24	432K	24	4
XC3S1500	29,952	32	576K	32	4
XC3S2000	46,080	40	720K	40	4
XC3S4000	62,208	96	1,728K	96	4
XC3S5000	74,880	104	1,872K	104	4

5. VGA display port
6. RS-232 serial port
7. RS-232 transceiver/voltage-level convertor
8. Second RS-232 transmit and receive channel
9. PS/2 mouse/keyboard port
10. Four-digit seven-segment LED display
11. Eight slide switches
12. Eight discrete LED outputs
13. Four momentary-contact pushbutton switches
14. 50-MHz crystal oscillator clock source
15. Socket for an auxiliary crystal oscillator clock source
16. Jumper to select an FPGA configuration mode
17. Pushbutton switch to force FPGA reconfiguration
18. LED to indicate whether the FPGA is successfully configured
19. 40-pin expansion connector 1 (labeled B1)
20. 40-pin expansion connector 2 (labeled A2)
21. 40-pin expansion connector 3 (labeled A1)
22. JTAG connector for Digilent download cable.
23. Digilent low-cost download cable (included in the S3 kit but not shown in Figure 2.3)
24. JTAG port (to be used with the Xilinx Parallel Cable IV and MultiPRO Desktop Tool, which are not included in the S3 kit)
25. Power connector for an unregulated 5-V power supply (included in the S3 kit)
26. Power-on LED indicator
27. 3.3-V voltage regulator
28. 2.5-V voltage regulator
29. 1.2-V voltage regulator
30. Selector for PS2 port voltage supply (3.3 or 5 V)

2.4 DEVELOPMENT FLOW

The simplified development flow of an FPGA-based system is shown in Figure 2.4. To facilitate further reading, we follow the terms used in the Xilinx documentation. The left portion of the flow is the refinement and programming process, in which a system is transformed from an abstract textual HDL description to a device cell-level configuration

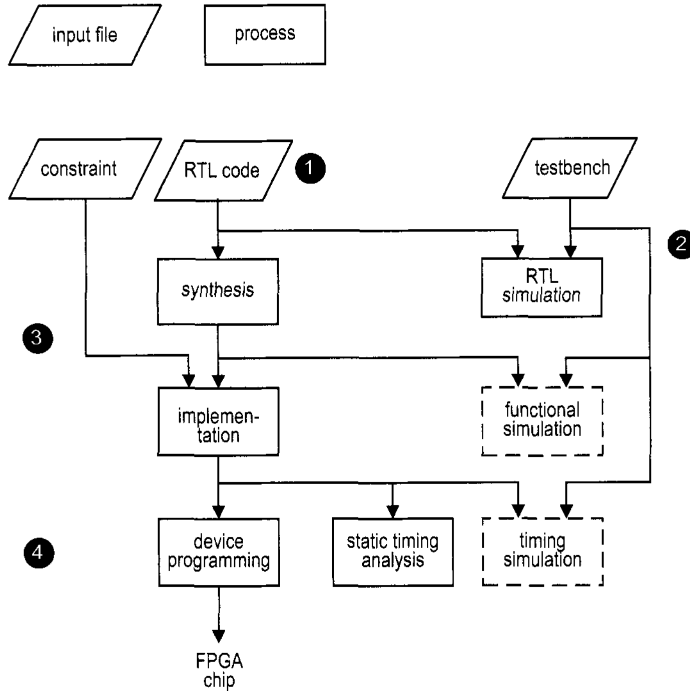


Figure 2.4 Development flow.

and then downloaded to the FPGA device. The right portion is the validation process, which checks whether the system meets the functional specification and performance goals. The major steps in the flow are:

1. Design the system and derive the HDL file(s). We may need to add a separate constraint file to specify certain implementation constraints.
2. Develop the testbench in HDL and perform *RTL simulation*. The RTL term reflects the fact that the HDL code is done at the register transfer level.
3. Perform *synthesis* and *implementation*. The synthesis process is generally known as *logic synthesis*, in which the software transforms the HDL constructs to generic gate-level components, such as simple logic gates and FFs. The *implementation* process consists of three smaller processes: translate, map, and place and route. The *translate process* merges multiple design files to a single netlist. The *map process*, which is generally known as *technology mapping*, maps the generic gates in the netlist to FPGA's logic cells and IOBs. The *place and route process*, which is generally known as *placement and routing*, derives the physical layout inside the FPGA chip. It places the cells in physical locations and determines the routes to connect various signals. In the Xilinx flow, *static timing analysis*, which determines various timing parameters, such as maximal propagation delay and maximal clock frequency, is performed at the end of the implementation process.
4. Generate and download the programming file. In this process, a configuration file is generated according to the final netlist. This file is downloaded to an FPGA device serially to configure the logic cells and switches. The physical circuit can be verified accordingly.

The optional *functional simulation* can be performed after synthesis, and the optional *timing simulation* can be performed after implementation. Functional simulation uses a synthesized netlist to replace the RTL description and checks the correctness of the synthesis process. Timing simulation uses the final netlist, along with detailed timing data, to perform simulation. Because of the complexity of the netlist, functional and timing simulation may require a significant amount of time. If we follow good design and coding practices, the HDL code will be synthesized and implemented correctly. We only need to use RTL simulation to check the correctness of the HDL code and use static timing analysis to examine the relevant timing information. Both functional and timing simulations can be omitted from the development flow.

2.5 OVERVIEW OF THE XILINX ISE PROJECT NAVIGATOR

Xilinx ISE (integrated software environment) controls all aspects of the development flow. *Project Navigator* is a graphical interface for users to access software tools and relevant files associated with the project. We use it to launch all development tasks except ModelSim simulation. The discussion in this section and the tutorial in the next section are based on ISE WebPack version 8.2.

The default ISE window is shown in Figure 2.5. It is divided into four subwindows:

- *Sources window* (top left): hierarchically displays the files included in the project
- *Processes window* (middle left): displays available processes for the source file currently selected
- *Transcript window* (bottom): displays status messages, errors, and warnings
- *Workplace window* (top right): contains multiple document windows (such as HDL code, report, schematic, and so on) for viewing and editing

Each subwindow may be resized, moved, docked, or undocked. The default layout can be restored by selecting View ≻ Restore. Note that a subwindow may contain multiple pages. The tabs at the bottom are used to select the desired page.

Sources window The sources window is used mainly to display files associated with the current project. A typical source window, which corresponds to the design of Listing 2.2, is shown in Figure 2.6. The top drop-down list, labeled Sources for:, specifies the current design view. The `synthesis/implementation` view should be selected since we use ISE only for synthesis and implementation,

There are three tabs at the bottom, labeled Sources, Snapshots, and Libraries. The Sources tab displays the project name, the FPGA device specified, and user documents and design files. The modules are displayed according to the internal design hierarchy. In Figure 2.6, the `eq2` and `eq1` entities reflect the hierarchy of Listing 2.2. The `eq2` module also includes the `eq_s3.ucf` file, which specifies the constraints of the design. We can open a file in the workplace window by double-clicking the corresponding module. A *top-level module* icon can be placed next to a module, as in the `eq2` module, to invoke synthesis and implementation for this particular module.

The Snapshots tab displays project's "snapshots," which are copies of previously stored project files. The Libraries tab shows all libraries associated with the project.

Processes window The processes window displays the processes available. The display is *context sensitive* and the available processes are based on source type selected in the sources window. For example, the `eq2` module, which is set as the top-level module,

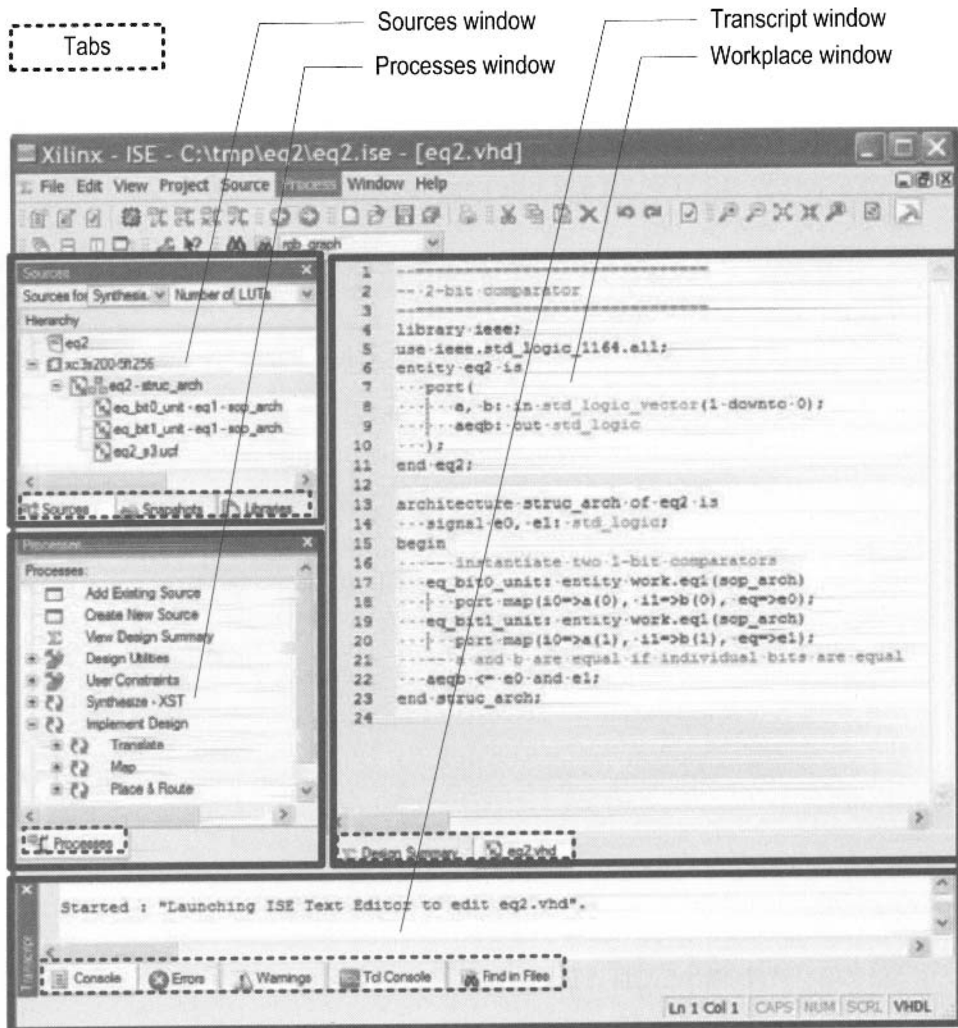


Figure 2.5 Typical ISE window.

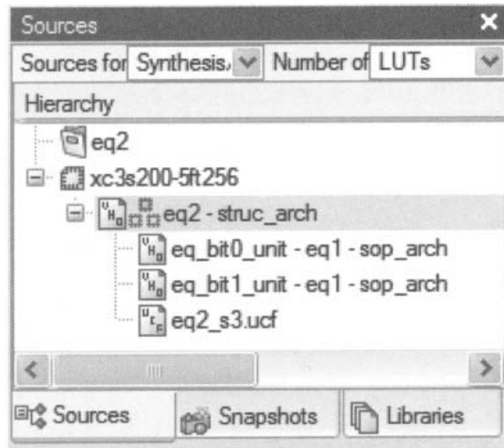


Figure 2.6 Typical source window.

is selected in Figure 2.6. The available processes are displayed in the processes window, as shown in Figure 2.7. Some processes may also contain several subprocesses. We can initiate a process by clicking on the corresponding icon. ISE incorporates the "auto make" technology, which automatically runs the processes necessary to get to the desired step. For example, when we initiate the Generate Programming File process, ISE automatically invokes the Synthesize and Implement Design processes since file generation is dependent on the implementation result, which, in turn, is dependent on the synthesis result.

Transcript window The transcript window is used to display the progress of a process and relevant messages. The Console page displays errors, warnings, and information messages. An error is signified by a red X mark next to the message and a warning is signified by a yellow ! mark. The Warnings and Errors pages display only warning and error messages.

Workplace window The workplace window is for users to view and edit various types of files. We use it to perform two main tasks. The first task is to view and edit the HDL and constraint files. The default editor is the *ISE Text Editor*, which is a simple text editor with features to assist creation of the HDL code. The second task is to check the design summary and various reports.

2.6 SHORT TUTORIAL ON ISE PROJECT NAVIGATOR

Xilinx ISE consists of an array of software tools, but detailed discussion of their use is beyond the scope of this book. We present a short tutorial in this section to illustrate the basic development process. There are four major steps:

1. Create the design project and HDL codes.
2. Create a testbench and perform RTL simulation.
3. Add a constraint file and synthesize and implement the code.
4. Generate and download the configuration file to an FPGA device.

These steps follow the general development flow discussed in Section 2.4.

We use the 2-bit comparator discussed in Chapter 1 in the tutorial. The codes are repeated in Listings 2.1 and 2.2.

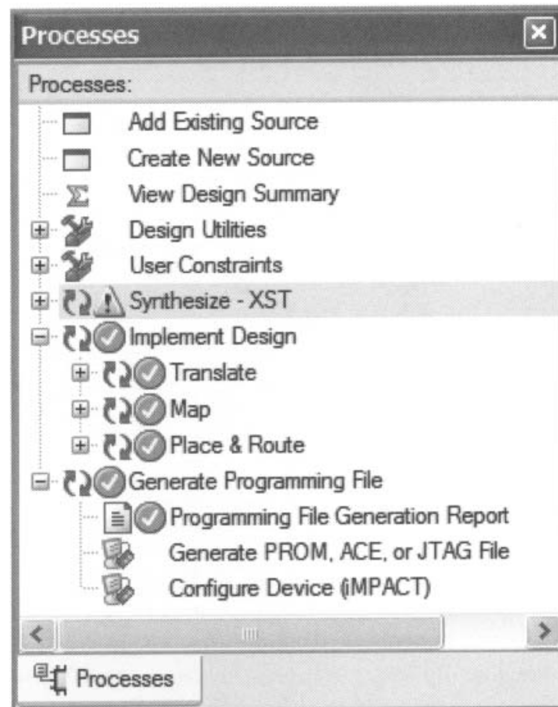


Figure 2.7 Typical processes window.

Listing 2.1 Gate-level implementation of a 1-bit comparator

```
library ieee;
use ieee.std_logic_1164.all;
entity eq1 is
   port(
      i0, i1: in std_logic;
      eq: out std_logic
   );
end eq1;

architecture sop_arch of eq1 is
   signal p0, p1: std_logic;
begin
   -- sum of two product terms
   eq <= p0 or p1;
   -- product terms
   p0 <= (not i0) and (not i1);
   p1 <= i0 and i1;
end sop_arch;
```

Listing 2.2 Structural description of a 2-bit comparator

```
library ieee;
use ieee.std_logic_1164.all;
entity eq2 is
```

```
   port(
5     a, b: in std_logic_vector(1 downto 0);
      aeqb: out std_logic
   );
end eq2;

10 architecture struc_arch of eq2 is
   signal e0, e1: std_logic;
begin
   -- instantiate two 1-bit comparators
   eq_bit0_unit: entity work.eq1(sop_arch)
15    port map(i0=>a(0), i1=>b(0), eq=>e0);
   eq_bit1_unit: entity work.eq1(sop_arch)
      port map(i0=>a(1), i1=>b(1), eq=>e1);
   -- a and b are equal if individual bits are equal
   aeqb <= e0 and e1;
20 end struc_arch;
```

2.6.1 Create the design project and HDL codes

There are three tasks in this step:

- Create a project.
- Add or create HDL files.
- Check the HDL syntax.

Create a project An ISE project contains basic information of a design, which includes the source files and a target device. A new project can be created as follows:

1. Select Start ≻ All Programs ≻ Xilinx ISE ≻ Project Navigator (or wherever ISE resides) to launch the ISE project navigator.
2. In Project Navigator, select File ≻ New Project. The New Project Wizard - Create New project dialog appears. Enter the project name as eq2 and the location, and verify that `HDL` is selected in the Top-level Source Type field. Click Next.
3. The New Project Wizard - Device Properties dialog appears. We need to enter the desired target device in this dialog. This information can be found in FPGA board manual or by checking the marking on the top of the FPGA chip. For a typical S3 board, select the following:
   - Product Category: `All`
   - Family: `Spartan3`
   - Device: `XC3S200`
   - Package: `FT256`
   - Speed: `-4`

   We also need to verify that the Xilinx XST software is selected for synthesis:
   - Synthesis Tool: `XST (VHDL/Verilog)`
4. Click Next a few times to go through the remaining dialogs and then click Finish to complete the creation.

After a project is created, we can create or add the relevant HDL files and a constraint file.

Create a new HDL file If a file does not exist, we must create a new source file. The procedure to create a new HDL file is:

1. Select Project ≻ New Source. The New Source Wizard - Select Source Type dialog appears. Select VHDL Module and type the file name, eq2. Click Next.
2. The next dialog appears. This dialog allows us to enter port names. These names are then later embedded in the HDL code. Enter the I/O port information according to Listing 2.2. Click Next.
3. Click Finish and a new HDL text editor window appears in the workplace window. The software automatically generates the HDL skeleton, which includes a comment header, library clauses, an entity declaration, and an empty architecture body.
4. By default, ISE version 8.2 generates the following library clauses:

   ```
   use IEEE.STD_LOGIC_ARITH.ALL;
   use IEEE.STD_LOGIC_UNSIGNED.ALL
   ```

   The two libraries are not IEEE standard and should be replaced with

   ```
   use ieee.numeric_std.all;
   ```

   This issue is explained in Section 3.2.2.
5. Use the editor to enter the HDL code in Listing 2.2 and save the file.
6. Repeat the process to create another file for the code in Listing 2.1.

Add existing files If a file already exists, it can be added to the project as follows:

1. Select Project ≻ Add Source. A dialog window appears.
2. Go to the desired directory and select the desired files. Click Open and a new dialog appears.
3. Click OK to complete the addition. These files now appear in the sources window of the project navigator.

Check the code syntax After completing a new HDL file, we need to check the syntax of the code:

1. Select the desired file in the source window.
2. In the processes window, click the + icon next to Synthesize to expand the process hierarchy.
3. Double-click the Check Syntax process.

The bottom transcript displays the progress of the process and reports errors and warnings, which are started with a red X and yellow ! marks. Double-clicking the message leads to the offending line in the file. We can correct the problem, save the file, and repeat the syntax checking process until all syntax errors are eliminated.

2.6.2 Create a testbench and perform the RTL simulation

The testbench functions as a virtual lab bench. It consists of the HDL module to be tested and a code segment to generate the stimulus. The RTL simulation verifies operation of the HDL module in the host computer. ISE contains a built-in ISE simulator and can launch the *ModelSim* simulator manufactured by Mentor Graphics Corporation. Since the latter is more robust and versatile, we use it in the book. Although ModelSim can be invoked from ISE Project Navigator, we treat it as an individual software tool and illustrate its use in Section 2.7.

2.6.3 Add a constraint file and synthesize and implement the code

There are three tasks in this step:

- Add a constraint file.
- Perform synthesis and implementation.
- Check the design summary.

Add a constraint file *Constraints* are certain conditions imposed on the synthesis and implementation processes. For our purposes, the main type of constraint is the pin assignment of a top-level I/O port and the minimal clock rate. During the implementation process, an I/O signal of the top-level module must be mapped to a physical pin of the FPGA device. Since the peripherals' I/O signals are already permanently connected to the designated FPGA's pins on the prototyping board, we must ensure that the signals are mapped to the corresponding pins. The other type of constraint is about timing, which specifies the minimal clock frequency to facilitate the oscillator of the board.

The constraint information is stored in a text file with an extension of .ucf (for the user constraint file). In the eq2 circuit, we can connect the a and b ports to four switches and the aeqb port to an LED to verify the physical operation of the circuit. For the S3 board, the corresponding pins are F12, G12, H14, H13, and K12. The constraint file becomes

```
# 4 slide switches
NET "a<0>"  LOC = "F12"  ;  # switch 0
NET "a<1>"  LOC = "G12"  ;  # switch 1
NET "b<0>"  LOC = "H14"  ;  # switch 2
NET "b<1>"  LOC = "H13"  ;  # switch 3
# led
NET "aeqb"  LOC = "K12"  ;  # led 0
```

Note that the # sign is used for a comment and the text after it is ignored. This file must be added to the design in the sources window.

There are several ISE tools to specify and generate the constraint file. Since all of our experiments are done in the same prototyping board, the constraints (i.e., pin assignment and clock frequency) remain the same. A constraint template file that includes all connected I/O peripheral signals of the S3 board is provided in the Appendix. One easy method to create a constraint file is simply to copy and edit the template file according to the I/O port names of the current design. The procedure to create the .ucf file for the eq2 circuit is:

1. Copy the template constraint file and rename it eq2_s3.ucf.
2. Follow the procedure in Section 2.6.1 to add the new constraint file to the eq2 module in the sources window.
3. Select the constraint file.
4. In the processes window, click the + icon next to User Constraints to expand the process hierarchy.
5. Double-click the Edit Constraints (Text) process to launch the ISE text editor.
6. Rename the I/O names as needed and then delete the unused pin assignments.
7. Save the file.

The default option of ISE version 8.2 only allows the pin assignments of the existing top-level I/O ports. If unused pin assignments are not deleted from the ucf template, error messages will be generated. We can override the default option as follows:

1. Select the top-level HDL file.
2. Right-click the Implement Design process in the processes window and then select Properties... from the menu. A dialog window appears.
3. In the dialog window, check the Allow Unmatched LOC Constraints option and then click OK.

After this option is turned on, we can use the same ucf template for all designs as long as the same I/O port names are kept in the top-level module, and we don't need to edit the ucf file each time.

Perform synthesis and implementation Invoking the synthesis and implementation procedure is very simple:

1. Select the module to be synthesized and make sure that it is designated as the top-level module (with a green square next to the module icon).
2. Double-click the Implement Design process in the processes window.
3. Although the syntax is checked earlier, the code may contain constructs that cannot be synthesized or may lead to poor implementation (such as a combinational loop). The error and warning messages are displayed in the console tab of the transcript window.
4. Correct the problems and repeat the simulation and synthesis processes if needed.

Check the design summary As the project progresses, a report is generated in each process. These reports and key statistics are summarized in a design summary window. We can check the size of the resulting circuit (in terms of the numbers of slices, FFs, and LUTs) and, for a sequential circuit, check whether the clock rate meets the timing constraints. The summary can be invoked by double-clicking the View Design Summary process in the processes window. The summary for the eq2 circuit is shown in Figure 2.8. We can check the use of slices, LUTs, and so on, in the Device Utilization Summary portion. A more detailed report can be invoked by clicking the corresponding link.

2.6.4 Generate and download the configuration file to an FPGA device

The last step is to generate the configuration file and download the file to the FPGA device. There are three tasks in this step:

- Connect the download cable.
- Generate the configuration file.
- Download the configuration file.

The S3 kit comes with a parallel-port JTAG download cable, and the following discussion is based on this cable. The procedures for other cables are similar and detailed instructions can be found in their manuals.

Connect the download cable The procedure to prepare the board is as follows:

1. Make sure that the *PROM* and the *Mode* jumpers (labeled 3 and 16 in Figure 2.3) are in their default setting (as the board is shipped).
2. Connect the power cable.
3. Connect one end of the download cable to the parallel port of a PC and connect the other end to the JTAG port (labeled 22 in Figure 2.3) on the S3 board.

Generate the configuration file Generating a configuration file is very straightforward:

1. Make sure that the top-level module is selected in the source window.
2. Click Generate Programming File in the processes window.

After this process is completed, a configuration file, eq2.bit, is generated.

EQ2 Project Status			
Project File:	eq2.ise	**Current State:**	Placed and Routed
Module Name:	eq2	• **Errors:**	No Errors
Target Device:	xc3s200-5ft256	• **Warnings:**	No Warnings
Product Version:	ISE, 8.1i	• **Updated:**	Sun Jan 21 18:04:45 2007

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of 4 input LUTs	1	3,840	1%	
Logic Distribution				
Number of occupied Slices	1	1,920	1%	
Number of Slices containing only related logic	1	1	100%	
Number of Slices containing unrelated logic	0	1	0%	
Total Number of 4 input LUTs	1	3,840	1%	
Number of bonded IOBs	5	173	2%	
Total equivalent gate count for design	6			
Additional JTAG gate count for IOBs	240			

Performance Summary			
Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints		

Detailed Reports					
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Sat Jan 20 22:22:32 2007	0	0	0
Translation Report	Current	Sat Jan 20 22:22:46 2007	0	0	0
Map Report	Current	Sat Jan 20 22:23:00 2007	0	0	2 Infos
Place and Route Report	Current	Sat Jan 20 22:23:18 2007	0	0	1 Info
Static Timing Report	Current	Sat Jan 20 22:23:30 2007	0	0	2 Infos
Bitgen Report					

Figure 2.8 Design summary.

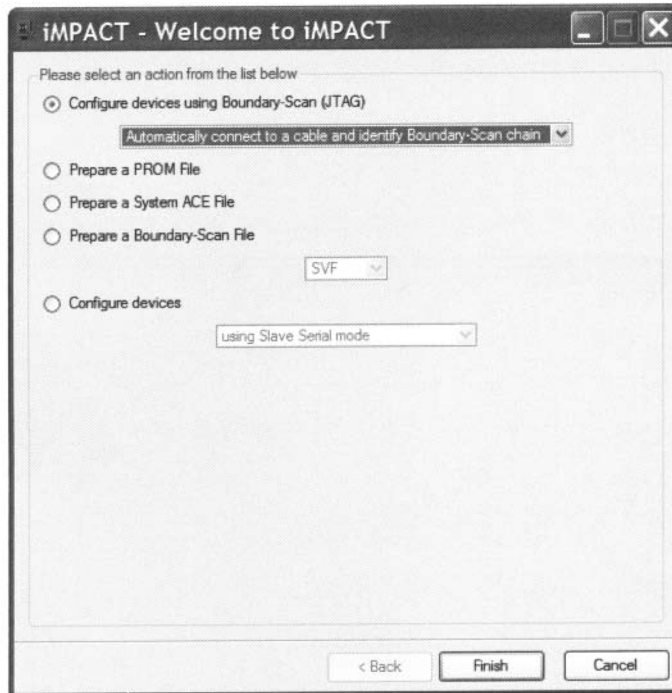


Figure 2.9 iMPACT welcome dialog.

Download the configuration file Downloading the configuration file to an FPGA device is done by a software tool known as *iMPACT*, which can be invoked from ISE Project Navigator. The procedure is

1. In the processes window, click the + sign to expand the Generate Programming File hierarchy.
2. Double-click the Configure Device (iMPACT) process. The Welcome to iMPACT dialog appears, as shown in Figure 2.9. Check Configure devices using Boundary-Scan (JTAG) and verify that Automatically connect to a cable and identify Boundary-Scan chain is selected in the drop-down list. Click Finish.
3. If a message indicating that two devices are found is displayed, click OK to continue.
4. The main iMPACT window, along with the Assign New Configuration File dialog, appears, as shown in Figure 2.10. The devices connected to the JTAG chain on the board should be detected and displayed.
5. Select the eq2.bit file and click Open to assign this configuration file to the xc3s200 device in the JTAG chain.
6. If a warning message appears, ignore it and click OK.
7. Select Bypass to skip the other device.
8. Right-click on the xc3s200 device image, and select Program .... The Programming Properties dialog opens. Click OK to program the device.
9. The Program Succeeded message appears when the downloading process is completed.

Now the FPGA device is configured and we can test the circuit with the switches and observe the output LED.

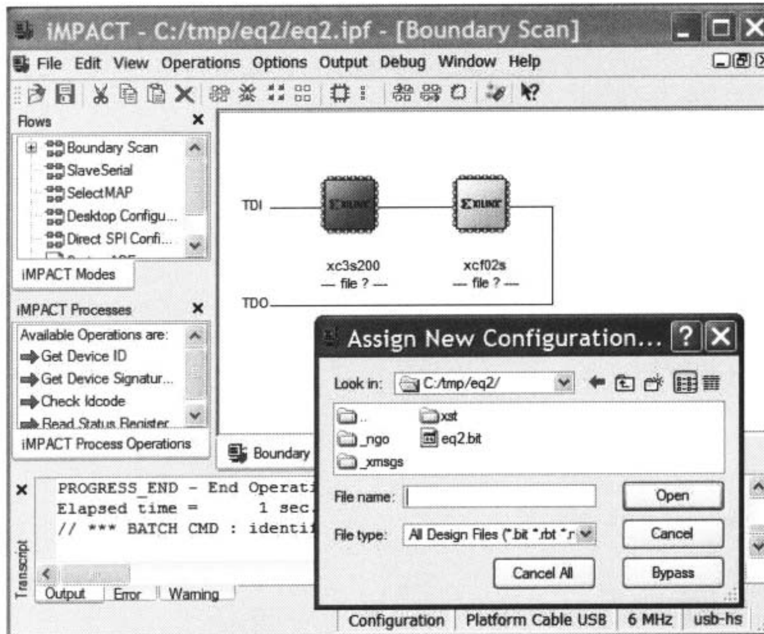


Figure 2.10 iMPACT main window.

An alternative way to configure the FPGA is to download the configuration file to a PROM and load the configuration file from the PROM. More information may be found in the sources cited in the Bibliographic section.

2.7 SHORT TUTORIAL ON THE MODELSIM HDL SIMULATOR

The ModelSim software is an HDL simulator manufactured by Mentor Graphics Corporation and can run independently without ISE. The discussion in this section is based on ModelSim XE III Starter version 6.0d.

The default ModelSim window is shown in Figure 2.11. It is divided into three subwindows: Transcript window (bottom), Workspace window, and multiple document interface (MDI) window. The Workspace window displays information on the current process. The bottom tab is used to select the desired process page, which can be Project, Library, Sim, and so on. The Transcript window keeps track of command history and messages. It can also be used as a command-line interface to enter ModelSim commands. The MDI window is an area to display HDL text, waveform, and so on. The bottom tab selects the desired pages.

Each subwindow may be resized, moved, docked, or undocked. Additional windows may appear for some operations. The default layout can be restored by selecting Window ≻ Initial Layout.

We present a short tutorial in this section to illustrate the basic simulation process. There are three steps:

1. Prepare a simulation project.
2. Compile the HDL codes.
3. Perform a simulation and examine the waveform.

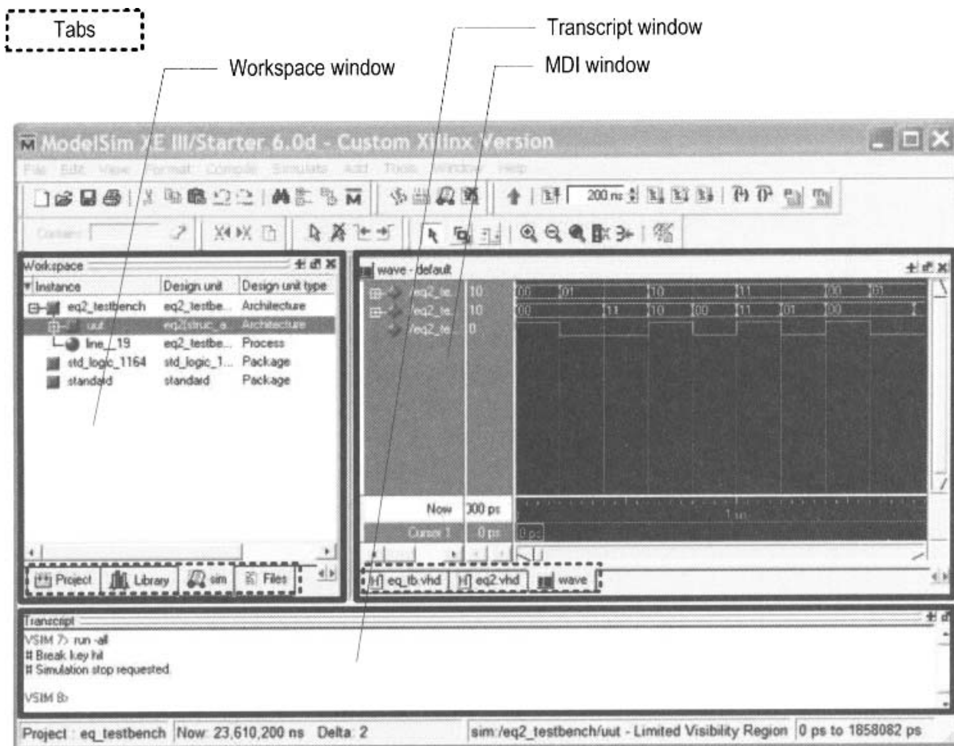


Figure 2.11 Typical ModelSim window.

We use the 2-bit comparator testbench discussed in Chapter 1 for the tutorial, and the code is repeated in Listing 2.3. An additional assertion statement,

```
assert false
   report "Simulation Completed"
   severity failure;
```

is added to the end of the process. It generates an "artificial failure" and stops the simulation.

Listing 2.3 Testbench of a 2-bit comparator

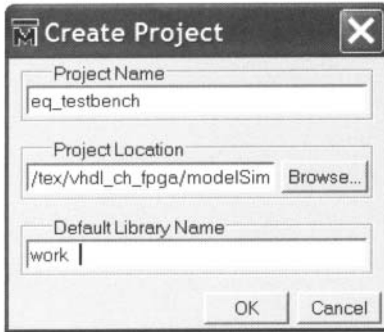
```
library ieee; use ieee.std_logic_1164.all;
entity eq2_testbench is
end eq2_testbench;

architecture tb_arch of eq2_testbench is
   signal test_in0, test_in1: std_logic_vector(1 downto 0);
   signal test_out: std_logic;
begin
   -- instantiate the circuit under test
   uut: entity work.eq2(struc_arch)
      port map(a=>test_in0, b=>test_in1, aeqb=>test_out);
   -- test vector generator
   process
```

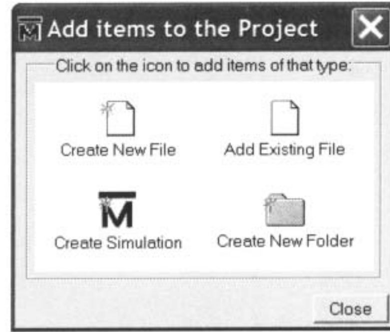
```
   begin
15    -- test vector 1
      test_in0 <= "00";
      test_in1 <= "00";
      wait for 200 ns;
      -- test vector 2
20    test_in0 <= "01";
      test_in1 <= "00";
      wait for 200 ns;
      -- test vector 3
      test_in0 <= "01";
25    test_in1 <= "11";
      wait for 200 ns;
      -- test vector 4
      test_in0 <= "10";
      test_in1 <= "10";
30    wait for 200 ns;
      -- test vector 5
      test_in0 <= "10";
      test_in1 <= "00";
      wait for 200 ns;
35    -- test vector 6
      test_in0 <= "11";
      test_in1 <= "11";
      wait for 200 ns;
      -- test vector 7
40    test_in0 <= "11";
      test_in1 <= "01";
      wait for 200 ns;
      -- terminate simulation
      assert false
45       report "Simulation Completed"
         severity failure;
   end process;
end tb_arch;
```

Prepare a simulation project A ModelSim simulation project consists of the library definition and a collection of HDL files. A testbench is an HDL program and can be created by using the ISE text editor, as discussed in Section 2.6.1. Alternatively, ModelSim also has a built-in editor. We assume that all HDL files are already constructed. The procedure to create a project is as follows:

1. Select Start ≻ All Programs ≻ ModelSim XE III 6.0d ≻ ModelSim (or wherever ModelSim resides) to launch the ModelSim program.
2. Select File ≻ New ≻ Project and the Create Project dialog appears, as shown in Figure 2.12(a). Enter the project name as `eq_testbench`, select the project location, and set Default Library Name to `work`. Click OK. A blank Project page appears in the main window and the Add items to the project dialog appears, as shown in Figure 2.12(b).
3. In the Add items to the project dialog, click Add Existing File and add the necessary HDL files. Click OK. The project tab appears in the workplace subwindow and displays the selected files, as shown in Figure 2.13.



(a) `Create Project` dialog



(b) `Add items` dialog

Figure 2.12 New project dialogs.

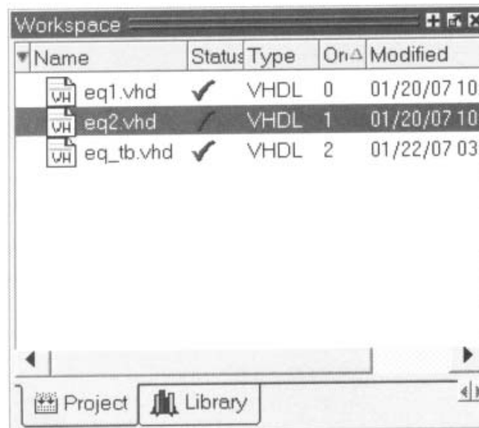


Figure 2.13 Project tab of the workplace panel.

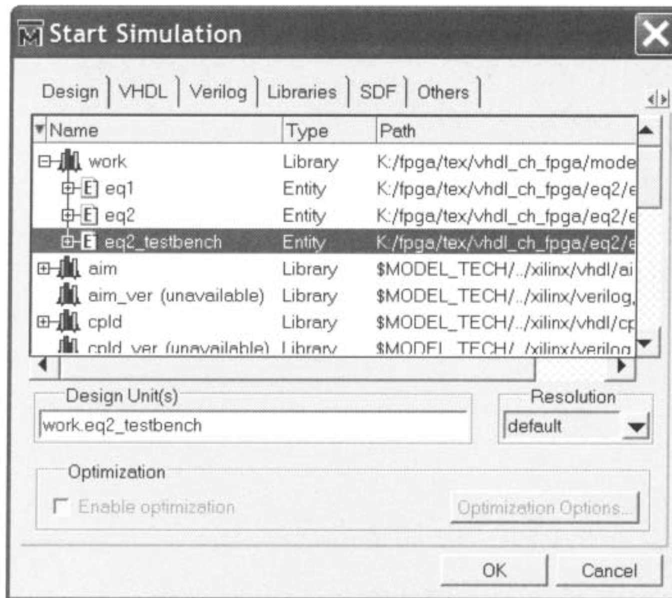


Figure 2.14 Simulate dialog.

Compile the HDL code The *compile* term here means to convert the HDL code into ModelSim internal format. In VHDL, the compiling is done on the *design unit* basis. Each entity and architecture is considered as one design unit. The procedure is:

1. Highlight the `eq1` file and right-click the mouse. Select Compile ≻ Compile Selected. Note that the compiling should be started from the modules at the bottom of the design hierarchy. The progress and messages are displayed in the transcript window.
2. If the file contains no syntactical error, a check mark shows up. Otherwise, an X mark shows up. Click the red error line in the transcript window to locate the errors. Correct the problems, save the file, and recompile the file.
3. Repeat the preceding steps to compile the `eq2` file and then the `eq_tb` file.

Perform a simulation and examine the waveform After compiling the testbench and corresponding files, we can perform the simulation and examine the resulting waveform. This corresponds to running the circuit in a virtual lab bench and checking the waveform in a virtual logic analyzer. The procedure is:

1. Select Simulate ≻ Simulate and the Simulate dialog appears.
2. In the Design tab, find and expand the work library, which is the one defined when we create the project. All compiled units are displayed, as shown in Figure 2.14.
3. Load `eq2_testbench` by double-clicking the corresponding icon. The sim tab appears in the workplace window and the corresponding page displays the structure of the `eq2_testbench` module, as shown in Figure 2.15. An object window, which contains the signals in the selected module, may also appear.
4. Highlight the uut unit and right-click the mouse. Select Add ≻ Add to Wave. This adds all the signals of the uut unit to the waveform page. The waveform page appears in the MDI window.
5. If necessary, rearrange the signals order and set them to proper format (decimal, hex, and so on.).

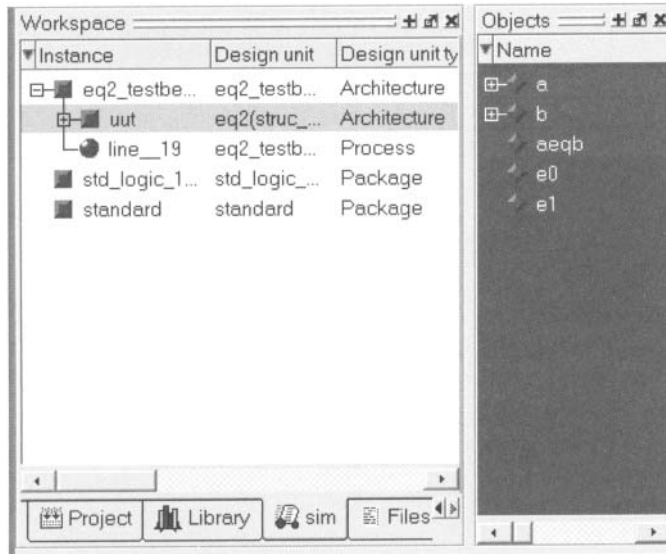


Figure 2.15 Sim panel of the workplace panel.

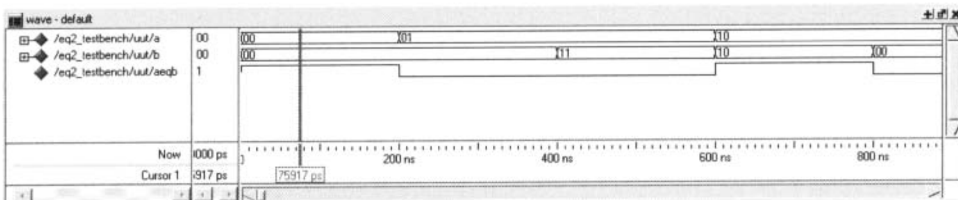


Figure 2.16 Waveform window.

6. Select Simulate ≻ Run. There are several commands to control the simulation: Restart (restart the simulation), Run (run the simulation one step), Continue run (resume the run from the interrupt), Run All (run the simulation forever), and Break (break the simulation). These commands are also shown as icons at the top of the window.
7. The waveform window displays the simulated result, shown in Figure 2.16. We can scroll the window, zoom in, or zoom out to check the correctness of the design.

2.8 BIBLIOGRAPHIC NOTES

Both Xilinx ISE and Mentor Graphics ModelSim are complex software packages, and their documentation exceeds several thousand pages. Most documentation can be accessed via the Help menu. ISE has a short 30-page tutorial, *ISE 8.1i Quick Start Tutorial*, and a more comprehensive 170-page tutorial, *ISE In-Depth Tutorial*. ModelSim also has a similar tutorial, *ModelSim Tutorial*. These tutorials provide an overview on all features of the software package. Relevant information for the Spartan-3 device can be found in its data sheets, *DS099 Spartan-3 FPGA Family: Complete Data Sheet*, which includes the detailed

Table 2.2 Truth table of a 2-to-4 decoder with enable

input			**output**
en	$a(1)$	$a(0)$	*bcode*
0	–	–	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000

explanation on the logic cells and macro cells. *The Design Warrior's Guide to FPGAs* by Clive Maxfield provides a comprehensive review of FPGA-related issues. The detailed layout and I/O connectors of the S3 board can be found in *Spartan-3 Starter Kit Board User Guide*. Information on other prototyping boards can be found in their manuals.

2.9 SUGGESTED EXPERIMENTS

2.9.1 Gate-level greater-than circuit

The greater-than circuit compares two inputs, a and b, and asserts an output when a is greater than b. We want to create a 4-bit greater-than circuit from the bottom up and use only gate-level logical operators. Design the circuit as follows:

1. Derive the truth table for a 2-bit greater-than circuit and obtain the logic expression in the sum-of-products format. Based on the expression, derive the HDL code using only logical operators.
2. Derive a testbench for the 2-bit greater-than circuit. Perform a simulation and verify the correctness of the design.
3. Use four switches as the inputs and one LED as the output. Synthesize the circuit and download the configuration file to the prototyping board. Verify its operation.
4. Use the 2-bit greater-than circuits and 2-bit equality comparators and a minimal number of "glue gates" to construct a 4-bit greater-than circuit. First draw a block diagram and then derive the structural HDL code according to the diagram.
5. Derive a testbench for the 4-bit greater-than circuit. Perform a simulation and verify the correctness of the design.
6. Use eight switches as the inputs and one LED as the output. Synthesize the circuit and download the configuration file to the prototyping board. Verify its operation.

2.9.2 Gate-level binary decoder

An n-to-2^n binary decoder asserts one of 2^n bits according to the input combination. The functional table of a 2-to-4 decoder with an enable signal is shown in Table 2.2. We want to create several decoders using only gate-level logical operators. The procedure is as follows:

1. Determine the logic expressions for the 2-to-4 decoder with enable and derive the HDL code using only logical operators.
2. Derive a testbench for the decoder. Perform a simulation and verify the correctness of the design.