Introduction to Digital Design
Using Digilent FPGA Boards
— Block Diagram / Verilog Examples

Richard E. Haskell
Darrin M. Hanna

Oakland University, Rochester, Michigan

LBE Books
Rochester Hills, MI

ISBN 978-0-9801337-9-0

Online Version

Published by LBE Books, LLC
1202 Walton Boulevard
Suite 214
Rochester Hills, MI 48307

www.lbebooks.com

Preface

A major revolution in digital design has taken place over the past decade. Field programmable gate arrays (FPGAs) can now contain over a million equivalent logic gates and tens of thousands of flip-flops. This means that it is not possible to use traditional methods of logic design involving the drawing of logic diagrams when the digital circuit may contain thousands of gates. The reality is that today digital systems are designed by writing software in the form of hardware description languages (HDLs). The most common HDLs used today are VHDL and Verilog. Both are in widespread use. When using these hardware description languages the designer typically describes the *behavior* of the logic circuit rather than writing traditional Boolean logic equations. Computer-aided design tools are used to both *simulate* the Verilog or VHDL design and to *synthesize* the design to actual hardware.

This book assumes no previous knowledge of digital design. We use 30 examples to show you how to get started designing digital circuits that you can implement on a Xilinx Spartan3E FPGA using either the Digilent BASYS™ system board that can be purchased from www.digilentinc.com for $59 or the Digilent Nexys-2 board that costs $99. We will use Active-HDL from Aldec to design, simulate, synthesize, and implement our digital designs. A free student edition of Active-HDL is available from Aldec, Inc. (www.aldec.com). To synthesize your designs to a Spartan3E FPGA you will need to download the free ISE WebPACK from Xilinx, Inc. (www.xilinx.com). The Xilinx synthesis tools are called from within the Aldec Active-HDL integrated GUI. We will use the ExPort utility to download your synthesized design to the Spartan3E FPGA. ExPort is part of the Adept software suite that you can download free from Digilent, Inc. (www.digilentinc.com). A more complete book called *Digital Design Using Digilent FPGA Boards – Verilog / Active-HDL Edition* is also available from Digilent or LBE Books (www.lbebooks.com). This more comprehensive book contains over 75 examples including examples of using the VGA and PS/2 ports. Similar books that use VHDL are also available from Digilent or LBE Books.

Many colleagues and students have influenced the development of this book. Their stimulating discussions, probing questions, and critical comments are greatly appreciated.

Richard E. Haskell
Darrin M. Hanna

Introduction to Digital Design Using Digilent FPGA Boards — Block Diagram / Verilog Examples

Table of Contents

Introduction – Digital Design Using FPGAs	1
Example 1 – Switches and LEDs	6
Example 2 – 2-Input Gates	11
Example 3 – Multiple-Input Gates	15
Example 4 – Equality Detector	20
Example 5 – 2-to-1 Multiplexer	22
Example 6 – Quad 2-to-1 Multiplexer	25
Example 7 – 4-to-1 Multiplexer	30
Example 8 – Clocks and Counters	37
Example 9 – 7-Segment Decoder	42
Example 10 – 7-Segment Displays: *x7seg* and *x7segb*	47
Example 11 – 2's Complement 4-Bit Saturator	55
Example 12 – Full Adder	60
Example 13 – 4-Bit Adder	65
Example 14 – *N*-Bit Adder	68
Example 15 – *N*-Bit Comparator	70
Example 16 – Edge-Triggered D Flip-Flop	*Available only in print vesion*
Example 17 – D Flip-Flops in Verilog	
Example 18 – Divide-by-2 Counter	
Example 19 – Registers	
Example 20 – *N*-Bit Register in Verilog	
Example 21 – Shift Registers	
Example 22 – Ring Counters	
Example 23 – Johnson Counters	
Example 24 – Debounce Pushbuttons	
Example 25 – Clock Pulse	
Example 26 – Arbitrary Waveform	
Example 27 – Pulse-Width Modulation (PWM)	
Example 28 – Controlling Position of a Servo	
Example 29 – Scrolling the 7-Segment Display	
Example 30 – Fibonacci Sequence	

Appendix A – Aldec Active-HDL Tutorial 109
Part 1: Project Setup 109
Part 2: Design Entry – sw2led.bde 113
Part 3: Synthesis and Implementation 116
Part 4: Program FPGA Board 120
Part 5: Design Entry – gates2.bde 122
Part 6: Simulation 128
Part 7: Design Entry – HDE 132
Part 8: Simulation – gates2 135

Appendix B – Number Systems *Available only in print vesion*
B.1 Counting in Binary and Hexadecimal
B.2 Positional Notation
B.3 Fractional Numbers
B.4 Number System Conversions
B.5 Negative Numbers

Appendix C – Basic Logic Gates
C.1 Truth Tables and Logic Equations
C.2 Positive and Negative Logic: De Morgan's Theorem
C.3 Sum of Products Design
C.4 Product of Sums Design

Appendix D – Boolean Algebra and Logic Equations
D.1 Boolean Theorems
D.2 Karnaugh Maps

Appendix E – Verilog Quick Reference Guide 175

Introduction

Digital Design Using FPGAs

The first integrated circuits that were developed in the early 1960s contained less that 100 transistors on a chip and are called small-scale integrated (SSI) circuits. Medium-scale integrated (MSI) circuits, developed in the late 1960s, contain up to several hundreds of transistors on a chip. By the mid 1970s large-scale integrated (LSI) circuits containing several thousands of transistors had been developed. Very-large-scale integrated (VLSI) circuits containing over 100,000 transistors had been developed by the early 1980s. This trend has continued to the present day with 1,000,000 transistors on a chip by the late 1980s, 10,000,000 transistors on a chip by the mid-1990s, over 100,000,000 transistors by 2004, and up to 1,000,000,000 transistors on a chip today. This exponential growth in the amount of digital logic that can be packed into a single chip has produced serious problems for the digital designer. How can an engineer, or even a team of engineers, design a digital logic circuit that will end up containing millions of transistors?

In Appendix C we show that any digital logic circuit can be made from only three types of basic gates: AND, OR, and NOT. In fact, we will see that any digital logic circuit can be made using only NAND gates (or only NOR gates), where each NAND or NOR gate contains four transistors. These basic gates were provided in SSI chips using various technologies, the most popular being transistor-transistor logic (TTL). These TTL chips were the mainstay of digital design throughout the 1960s and 1970s. Many MSI TTL chips became available for performing all types of digital logic functions such as decoders, adders, multiplexers, comparators, and many others.

By the 1980s thousands of gates could fit on a single chip. Thus, several different varieties of *programmable logic devices* (PLDs) were developed in which arrays containing large numbers of AND, OR, and NOT gates were arranged in a single chip without any predetermined function. Rather, the designer could design any type of digital circuit and implement it by connecting the internal gates in a particular way. This is usually done by opening up fuse links within the chip using computer-aided tools. Eventually the equivalent of many PLDs on a single chip led to *complex programmable logic devices* (CPLDs).

Field Programmable Gate Arrays (FPGAs)

A completely different architecture was introduced in the mid-1980's that uses RAM-based lookup tables instead of AND-OR gates to implement combinational logic. These devices are called *field programmable gate arrays* (FPGAs). The device consists of an array of *configurable logic blocks* (CLBs) surrounded by an array of I/O blocks. The Spartan-3E from Xilinx also contains some blocks of RAM, 18 x 18 multipliers, as well as Digital Clock Manager (DCM) blocks. These DCMs are used to eliminate clock distribution delay and can also increase or decrease the frequency of the clock.

Each CLB in the Spartan-3E FPGA contains four slices, each of which contains two 16 x 1 RAM look-up tables (LUTs), which can implement any combinational logic function of four variables. In addition to two look-up tables, each slice contains two D flip-flops which act as storage devices for bits. The basic architecture of a Spartan-3E FPGA is shown in Fig. 1.

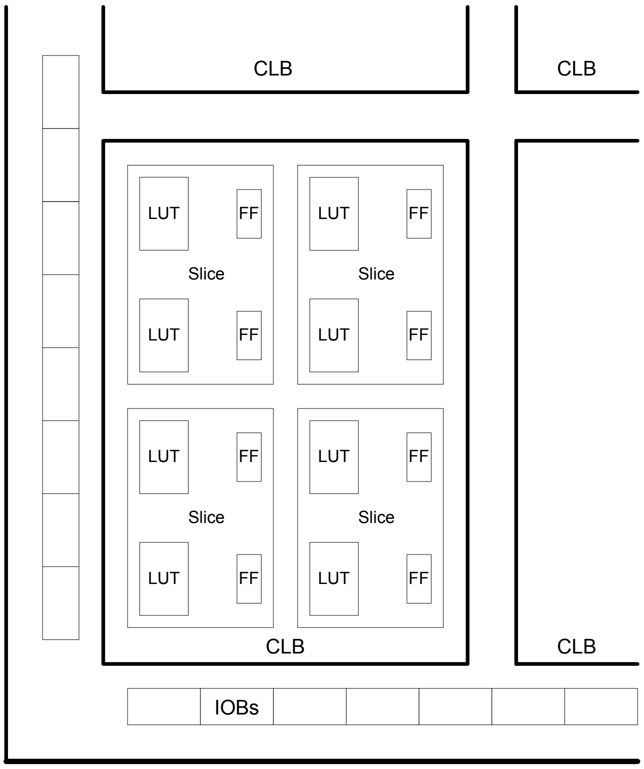


Figure 1 Architecture of a Spartan-3E FPGA

The BASYS board from Digilent contains a Xilinx Spartan3E-100 TQ144 FPGA. This chip contains 240 CLBs arranged as 22 rows and 16 columns. There are therefore 960 slices with a total of 1,920 LUTs and flip-flops. This part also contains 73,728 bits of block RAM. Half of the LUTs on the chip can be used for a maximum of 15,360 bits of distributed RAM.

By contrast the Nexys-2 board from Digilent contains a Xilinx Spartan3E-500 FG320 FPGA. This chip contains 1,164 CLBs arranged as 46 rows and 34 columns. There are therefore 4,656 slices with a total of 9,312 LUTs and flip-flops. This part also contains 368,640 bits of block RAM. Half of the LUTs on the chip can be used for a maximum of 74,752 bits of distributed RAM.

In general, FPGAs can implement much larger digital systems than CPLDs as illustrated in Table 1. The column labeled *No. of Gates* is really equivalent gates as we have seen that FPGAs really don't have AND and OR gates, but rather just RAM look-up tables. (Each slice does include two AND gates and two XOR gates as part of carry and arithmetic logic used when implementing arithmetic functions including addition and multiplication.) Note from Table 1 that FPGAs can have the equivalent of millions of gates and tens of thousands of flip-flops.

Table 1 Comparing Xilinx CPLDs and FPGAs

Xilinx Part	No. of Gates	No. of I/Os	No. of CLBs	No. of Flip-flops	Block RAM (bits)
CPLDs					
9500 family	800 – 6,400	34 – 192		36 - 288	
FPGAs					
Spartan	5,000 – 40,000	77 – 224	100 – 784	360 – 2,016	
Spartan II	15,000 – 200,000	86 – 284	96 – 1,176	642 – 5,556	16,384 – 57,344
Spartan IIE	23,000 – 600,000	182 – 514	384 – 3,456	2,082 – 15,366	32,768 – 294,912
Spartan 3	50,000 – 5,000,000	124 – 784	192 – 8,320	2,280 – 71,264	73,728 – 1,916,928
Spartan-3E	100,000 – 1,600,000	108 – 376	240 – 3,688	1,920 – 29,505	73,728 – 663,552
Virtex	57,906 – 1,124,022	180 – 512	384 – 6,144	2,076 – 26,112	32,768 – 131,072
Virtex E	71,693 – 4,074,387	176 – 804	384 – 16,224	1,888 – 66,504	65,536 – 851,968
Virtex-II	40,960 – 8,388,608	88 – 1,108	64 – 11,648	1,040 – 99,832	73,728 – 3,096,576

Modern Design of Digital Systems

The traditional way of designing digital circuits is to draw logic diagrams containing SSI gates and MSI logic functions. However, by the late 1980s and early 1990s such a process was becoming problematic. How can you draw schematic diagrams containing hundreds of thousands or millions of gates? As programmable logic devices replaced TTL chips in new designs a new approach to digital design became necessary. Computer-aided tools are essential to designing digital circuits today. What has become clear over the last decade is that today's digital engineer designs digital systems by writing software! This is a major paradigm shift from the traditional method of designing digital systems. Many of the traditional design methods that were important when using TTL chips are less important when designing for programmable logic devices.

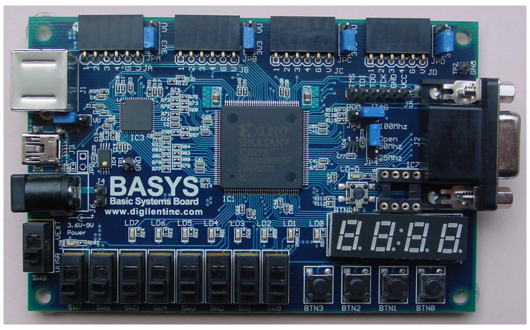
Today digital designers use *hardware description languages* (HDLs) to design digital systems. The most widely used HDLs are VHDL and Verilog. Both of these hardware description languages allow the user to design digital systems by writing a program that describes the behavior of the digital circuit. The program can then be used to both *simulate* the operation of the circuit and *synthesize* an actual implementation of the circuit in a CPLD, an FPGA, or an application specific integrated circuit (ASIC).

Another recent trend is to design digital circuits using block diagrams or graphic symbols that represent higher-level design constructs. These block diagrams can then be *compiled* to produce Verilog or VHDL code. We will illustrate this method in this book.

We will use Active-HDL from Aldec for designing our digital circuits. This integrated tool allows you to enter your design using either a block diagram editor (BDE) or by writing Verilog or VHDL code using the hardware description editor (HDE). Once your hardware has been described you can use the functional simulator to produce waveforms that will verify your design. This hardware description can then be synthesized to logic equations and implemented or mapped to the FPGA architecture.

We include a tutorial for using Active-HDL in Appendix A. A free student version of Active-HDL is available on their website.[1] We will use Xilinx ISE for synthesizing our VHDL designs. You can download a free version of ISE™ WebPACK™ from the Xilinx website.[2] This WebPACK™ synthesis tool can be run from within the Aldec Active-HDL development environment as shown in the tutorial in Appendix A. The implementation process creates a *.bit* file that is downloaded to a Xilinx FPGA on the BASYS board or Nexys-2 shown in Fig. 2. The BASYS board is available to students for $59 from Digilent, Inc.[3] This board includes a 100k-gate equivalent Xilinx Spartan3E FPGA (250k-gate capacity is also available), 8 slide switches, 4 pushbutton switches, 8 LEDs, and four 7-segment displays. The frequency of an on-board clock can be set to 25 MHz, 50 MHz, or 100 MHz using a jumper. There are connectors that allow the board to be interfaced to external circuits. The board also includes a VGA port and a PS2 port. The use of these ports are described in a different book.[4] Another more advanced board, the Nexys-2 board, is also available to students for $99 from Digilent. The Nexys-2 board is similar to the BASYS board except that it contains a 500k- or 1200k-gate equivalent Spartan 3E FPGA, a Hirose FX2 interface for additional add-on component boards, 16 MB of cellular RAM, 16 MB of flash memory, a 50 MHz clock and a socket for a second oscillator. The Nexys-2 is ideally suited for embedded processors.

All of the examples in this book can be used on both the BASYS board and the Nexys-2 board. The only difference is that you would use the file *basys2.ucf* to define the pinouts on the BASYS board and you would use the file *nexys2.ucf* to define the pinouts on the Nexys-2 board. Both of these files are available to download from www.lbebooks.com. Table 2 shows the jumper settings you would use on the two boards.


(a)

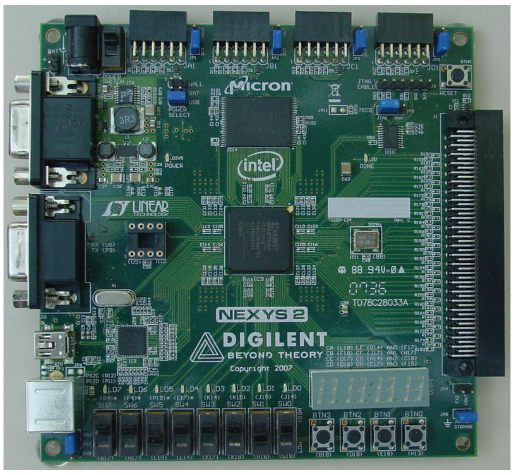

(b)

Figure 2 (a) BASYS board, (b) Nexys-2 Board

[1] http://www.aldec.com/education/
[2] http://www.xilinx.com
[3] http://www.digilentinc.com
[4] *Digital Design Using Digilent FPGA Boards – Verilog / Active-HDL Edition*; available from www.lbebooks.com.

Table 1.2 Board Jumper Settings

BASYS Boad	Nexys-2 Board
Set the JP3 jumper to JTAG	Set the POWER SELECT jumper to USB
Remove the JP4 jumper to select a 50 MHz clock	Set the MODE jumper to JTAG

Verilog

Verilog is based on the C programming language but it is *not* C. Verilog is a *hardware description language* that is designed to model digital logic circuits. It simply has the same syntax as the C programming language but the way it behaves is different. In this book we begin by using the Active-HDL block diagram editor to draw logic circuits using basic gates. When you *compile* these block diagrams Active-HDL will generate the corresponding Verilog code. The block diagram representing your logic circuit can then be used as a module in a higher-level digital design. This higher-level design can then be compiled to produce its corresponding Verilog code. This hierachical block diagram editor will make it easy to design top-level designs.

Sometimes it will be easier to design a digital module by writing a Verilog program directly rather than drawing it using gates. When you do this you can still use the block diagram for this module in higher-level designs. We will illustrate this process in many of our examples.

Just like any programming language, you can only learn Verilog by actually writing Verilog programs and simulating the designs using a Verilog simulator that will display the waveforms of the signals in your design. This is a good way to learn not only Verilog but digital logic as well.

A companion book[5] that uses VHDL instead of Verilog is available from Digilent or www.lbebooks.com. More comprehensive Verilog and VHDL books are also available.[6,7]

[5] *Introduction to Digital Design Using Digilent FPGA Boards – Block Diagram / VHDL Examples*, LBE Books, 2009.

[6] *Digital Design Using Digilent FPGA Boards – Verilog / Active-HDL Edition*, LBE Books, 2009.

[7] *Digital Design Using Digilent FPGA Boards – VHDL / Active-HDL Edition*, LBE Books, 2009.

Example 1

Switches and LEDs

In this example we will show the basic structure of a Verilog program and how to write logic equations for 2-input gates. Example 1a will show the simulation results using Aldec Active-HDL and Example 1b will show how to synthesize the program to a Xilinx FPGA on the BASYS or Nexys-2 board.

Prerequisite knowledge:
None
Learned in this Example:
Use of Aldec Active-HDL – Appendix A

1.1 Slide Switches

The slide switches on the BASYS and Nexys-2 boards are connected to pins on the FPGA through a resistor *R* as shown in Fig. 1.1. The value of *R* is 4.7 kΩ on the BASYS board and 10 kΩ on the Nexys-2 board. When the slide switch is down it is connected to ground and the input *sw*[*i*] to the FPGA is read as a logic 0. When the slide switch is up it is connected to 3.3 V and the input *sw*[*i*] to the FPGA is read as a logic 1.

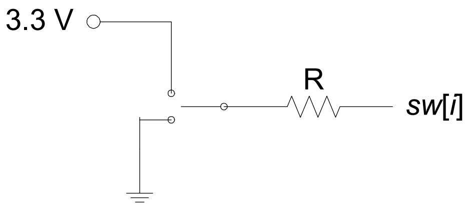


Figure 1.1 Slide switch connection

There are eight slide switches on the BASYS and Nexys-2 boards. The eight pin numbers on the FPGA corresponding to the eight slide switches are given in a *.ucf* file. The file *basys2.ucf* shown in Listing 1.1 defines the pin numbers for all I/O on the BASYS board. Note that we have named the slide switches *sw*[*i*], *i* = 0:7, which correspond to the switch labels on the board. We will always name the slide switches *sw*[*i*] in our top-level designs so that we can use the *basys2.ucf* file without change. Because the pin numbers on the Nexys-2 board are different from those on the BASYS board we will use a different file called *nexys2.ucf* to define the pin numbers on the Nexys-2 board. The names of the I/O ports, however, will be the same for both boards. Therefore, all of the examples in this book can be used with either board by simply using the proper *.ucf* file when implementing the design. Both of these *.ucf* files can be downloaded from www.lbebooks.com.

1.2 LEDs

A light emitting diode (LED) emits light when current flows through it in the positive direction as shown in Fig. 1.2. Current flows through the LED when the voltage on the *anode* side (the wide side of the black triangle) is made higher than the voltage on the *cathode* side (the straight line connected to the apex of the black triangle). When current flows through a lighted LED the forward voltage across the LED is typically between +1.5 and +2.0 volts. If voltage *V2* in Fig. 1.2 is less than or equal to voltage *V1* then no current can flow through the LED and therefore no light will be emitted. If voltage *V2* is greater than voltage *V1* then current will flow through the resistor *R* and the LED. The resistor is used to limit the amount of current that flows through the LED. Typical currents needed to light LEDs range from 2 to 15 milliamps.

Listing 1.1 basys2.ucf

```
# Pin assignment for LEDs
NET "ld<7>" LOC = "p2" ;
NET "ld<6>" LOC = "p3" ;
NET "ld<5>" LOC = "p4" ;
NET "ld<4>" LOC = "p5" ;
NET "ld<3>" LOC = "p7" ;
NET "ld<2>" LOC = "p8" ;
NET "ld<1>" LOC = "p14" ;
NET "ld<0>" LOC = "p15" ;

# Pin assignment for slide switches
NET "sw<7>" LOC = "p6";
NET "sw<6>" LOC = "p10";
NET "sw<5>" LOC = "p12";
NET "sw<4>" LOC = "p18";
NET "sw<3>" LOC = "p24";
NET "sw<2>" LOC = "p29";
NET "sw<1>" LOC = "p36";
NET "sw<0>" LOC = "p38";

# Pin assignment for pushbutton switches
NET "btn<3>" LOC = "p41";
NET "btn<2>" LOC = "p47";
NET "btn<1>" LOC = "p48";
NET "btn<0>" LOC = "p69";

# Pin assignment for 7-segment displays
NET "a_to_g<6>"  LOC = "p25"  ;
NET "a_to_g<5>"  LOC = "p16"  ;
NET "a_to_g<4>"  LOC = "p23"  ;
NET "a_to_g<3>"  LOC = "P21"  ;
NET "a_to_g<2>"  LOC = "p20"  ;
NET "a_to_g<1>"  LOC = "p17"  ;
NET "a_to_g<0>"  LOC = "p83"  ;
NET "dp"  LOC = "p22"  ;

NET "an<3>" LOC = "p26";
NET "an<2>" LOC = "p32";
NET "an<1>" LOC = "p33";
NET "an<0>" LOC = "p34";

# Pin assignment for clock
NET "mclk" LOC = "p54";
```

There are two different ways that an I/O pin of an FPGA can be used to turn on an LED. The first is to connect the FPGA pin to *V2* in Fig. 1.2 and to connect *V1* to ground. Bringing the pin (*V2*) high will then turn on the LED. To turn off the LED the output pin would be brought low. This is the method used for the LEDs *ld*[7] – *ld*[0] on the BASYS and Nexys-2 boards.

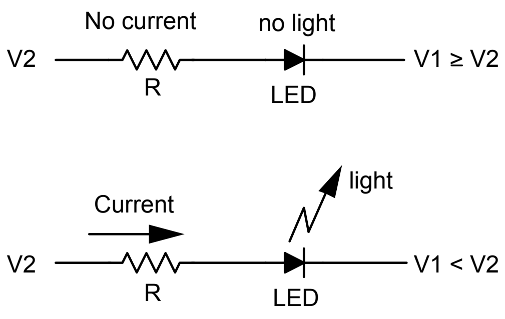


Figure 1.2 Turning on an LED

The second method is to connect the FPGA pin to *V1* in Fig. 1.2 and to connect *V2* to a constant voltage. Bringing the pin (*V1*) low will then turn on the LED. To turn off the LED the output pin would be brought high. This voltage should be equal to *V2* to make sure no current flows through the LED. This second method is the method used for the 7-segment displays on the BASYS and Nexys-2 boards. Examples 9 and 10 will show how to display hex digits on the 7-segment displays.

1.3 Connecting the Switches to the LEDs

Part 1 of the tutorial in Appendix A shows how to connect the input switches to the output LEDs using the block diagram editor (BDE) in Active-HDL. The result is shown in Fig. 1.3.

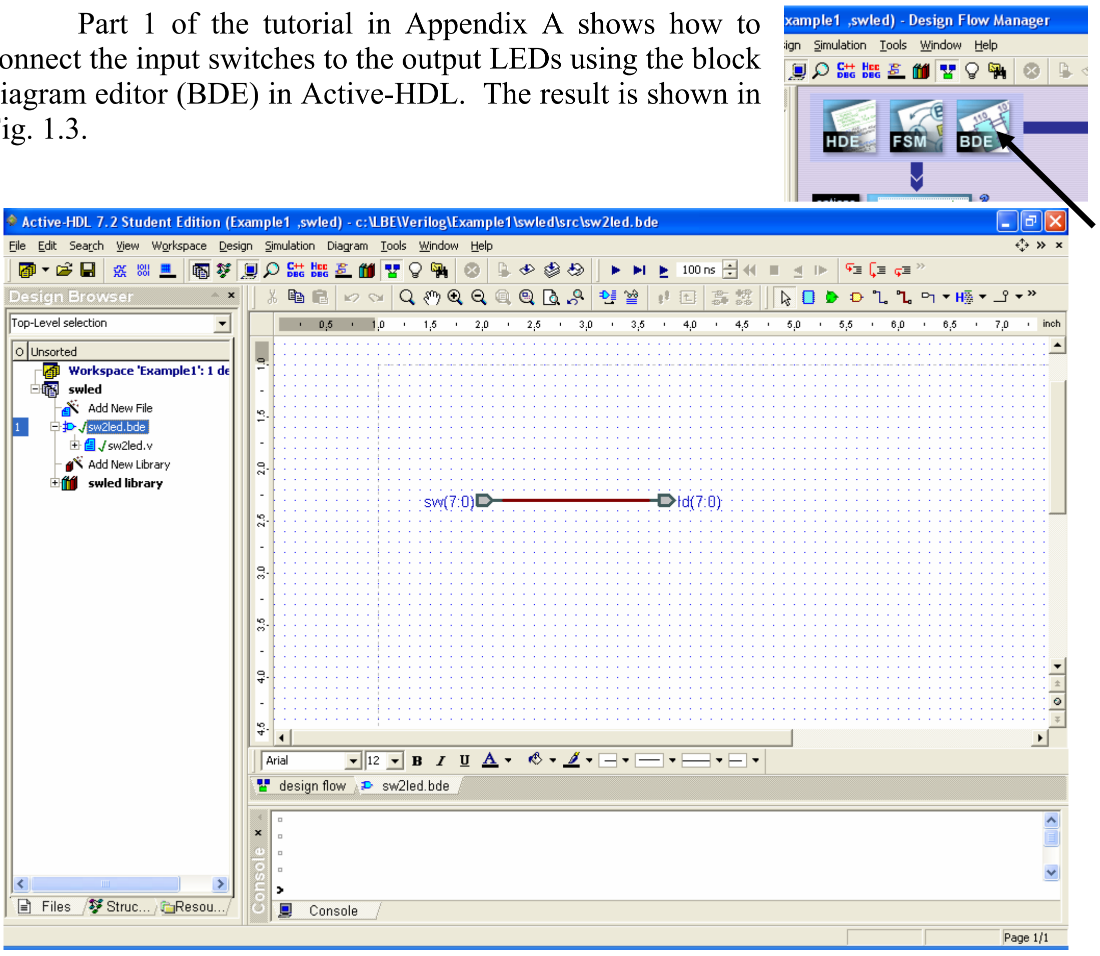

xample1 ,swled) - Design Flow Manager
HDE
FSM
BDE
Active-HDL 7.2 Student Edition (Example1 ,swled) - c:\LBE\Verilog\Example1\swled\src\sw2led.bde
File Edit Search View Workspace Design Simulation Diagram Tools Window Help
Design Browser
Top-Level selection
Unsorted
Workspace 'Example1': 1 de
swled
Add New File
sw2led.bde
sw2led.v
Add New Library
swled library
sw(7:0)
ld(7:0)
Files
Struc...
Resou...
design flow
sw2led.bde
Console
Page 1/1

Figure 1.3 Connecting the eight switches to the eight LEDs

Compiling the file *sw2led.bde* generates the Verilog file *sw2led.v* shown in Listing 1.2. Alternatively, by selecting the hardware description editor (HDE) the module statement and port declarations are automatically generated but you will need to write your own *assign* statement. This can lead to the simpler Verilog program shown in Listing 1.3 where we have combined the module statement and port declarations in a single module statement that conforms to the 2001 Verilog standard. This format makes it easier to see the input and output signals. We can also write a single *assign* statement to replace the two *assign* statements in Listing 1.2. It is unnecessary to define the intermediate bus *BUS7*[7:0] and because *sw* and *ld* are the same size we don't need to include the [7:0] in the *assign* statement.

Listing 1.2 sw2led.v

```
// Title        : sw2led
module sw2led (sw,ld) ;

// ------------ Port declarations --------- //
input [7:0] sw;
wire [7:0] sw;
output [7:0] ld;
wire [7:0] ld;

// ----------- Signal declarations -------- //
wire [7:0] BUS7;

// ----------- Terminals assignment --------//
//           ---- Input terminals ---         //
assign BUS7[7:0] = sw[7:0];

//             ---- Output terminals ---        //
assign ld[7:0] = BUS7[7:0];

endmodule
```

Listing 1.3 sw2led2.v

```
// Title        : sw2led2
module sw2led2 (
input wire [7:0] sw ,
output wire [7:0] ld
) ;

assign ld = sw;

endmodule
```

In Parts 2 and 3 of the tutorial in Appendix A we show how to synthesize, implement, and download the design to the FPGA board. In summary, the steps you follow to implement a digital design on the BASYS or Nexys-2 board are the following:

1. Create a new project and design name.
2. Using the BDE create a logic diagram.
3. Save and compile the *.bde* file.
4. Optionally simulate the design (see Example 2).
5. Synthesize the design selecting the Spartan3E family and the 3s100etq144 device for the BASYS board and the 3s500efg320 device for the Nexys-2 board.
6. Implement the design using either *basys2.ucf* or *nexys2.ucf* as the custom constraint file. Check *Allow Unmatched LOC Constraints* under *Translate* and uncheck *Do Not Run Bitgen* under *BitStream*. Select *JTAG Clock* as the start-up clock under *Startup Options*.
7. Use *ExPort* to download the *.bit* file to the FPGA board.

At this point the switches are connected to the LEDs. Turning on a switch will light up the corresponding LED.

Problem

1.1 The four pushbuttons on the BASYS and Nexys-2 boards are connected to pins on the FPGA using the circuit shown in Fig. 1.4. The value of R is 4.7 kΩ on the BASYS board and 10 kΩ on the Nexys-2 board. When the pushbutton is up the two resistors pull the input down to ground and the input *btn*(*i*) to the FPGA is read as a logic 0. When the pushbutton is pressed the input is pulled up to 3.3 V and the input *btn*(*i*) to the FPGA is read as a logic 1. Create a *.bde* file using Active-HDL that will connect the four pushbuttons to the rightmost four LEDs. Compile and implement the program. Download the *.bit* file to the FPGA board and test it by pressing the pushbuttons.

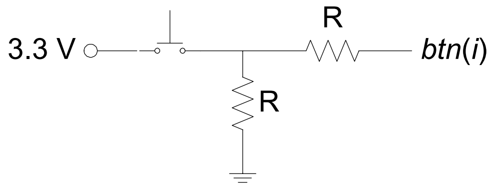


Figure 1.4 Pushbutton connection

Example 2

2-Input Gates

In this example we will design a circuit containing six different 2-input gates. Example 2a will show the simulation results using Aldec Active-HDL and Example 2b will show how to synthesize the program to a Xilinx FPGA on a Digilent board.

Prerequisite knowledge:
Appendix C – Basic Logic Gates
Appendix A – Use of Aldec Active-HDL

2.1 Generating the Design File *gates2.bde*

Part 4 of the tutorial in Appendix A shows how to connect two inputs *a* and *b* to the inputs of six different gates using the block diagram editor (BDE) in Active-HDL. The result is shown in Fig. 2.1. Note that we have named the outputs of the gates the name of the gate followed by an underscore. Identifier names in Verilog can contain any letter, digit, underscore _, or $. The identifier can not begin with a digit or be a keyword. Verilog is *case sensitive*.

The name of this file is *gates2.bde*. When you compile this file the Verilog program *gates2.v* shown in Listing 2.1 is generated. We have modified the module statement to conform to the 2001 Verilog standard as described in Example 1.

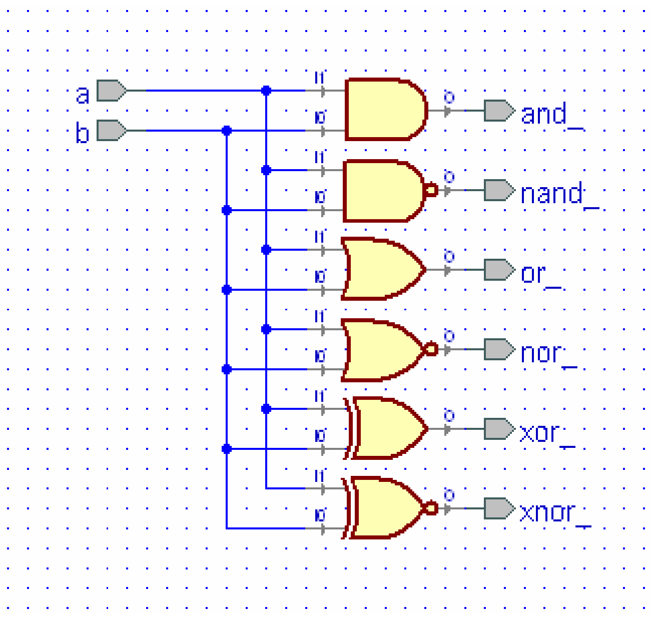


Figure 2.1 Circuit diagram for Example 2

Listing 2.1 gates2.v

```
// Example 2a: gates2
module gates2 (
input wire a,
input wire b,
output wire and_,
output wire nand_,
output wire nor_,
output wire or_,
output wire xnor_,
output wire xor_
) ;

assign and_ = b & a;
assign nand_ = ~(b & a);
assign or_ = b | a;
assign nor_ = ~(b | a);
assign xor_ = b ^ a;
assign xnor_ = ~(b ^ a);

endmodule
```

The logic diagram in Fig. 2.1 contains six different gates. This logic circuit is described by the Verilog program shown in Listing 2.1. The first line in Listing 2.1 is a comment. Comments in Verilog follow the double slash //. All Verilog programs begin with a *module* statement containing the name of the module (*gates2* in this case) followed by a list of all input and output signals together with their direction and type. We will generally use lower case names for signals. The direction of the input and output signals is given by the Verilog statements *input*, *output*, or *inout* (for a bi-directional signal). The type of the signal can be either *wire* or *reg*. In Listing 2.1 all of the signals are of type *wire* which you can think of as a wire in the circuit in Fig. 2.1 where actual voltages could be measured. We will describe the *reg* type in Example 5.

To describe the output of each gate in Fig. 2.1 we simply write the logic equation for that gate preceded by the keyword *assign*. These are *concurrent* assignment statements which means that the statements can be written in any order.

2.2 Simulating the Design *gates2.bde*

Part 4 of the tutorial in Appendix A shows how to simulate this Verilog program using Active-HDL. The simulation produced in Appendix A is shown in Fig. 2.2. Note that the waveforms shown in Fig. 2.2 verify the truth tables for the six gates. Also note that two clock stimulators were used for the inputs *a* and *b*. By making the period of the clock stimulator for the input *a* twice the period of the clock stimulator for the input *b* all four combinations of the inputs *a* and *b* will be generated in one period of the input *a*.

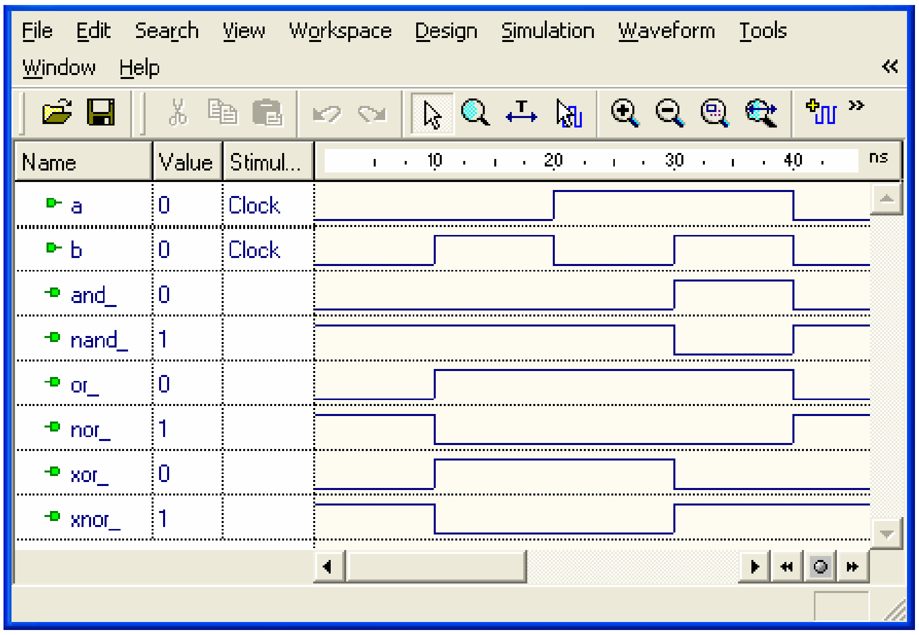


Figure 2.2 Simulation of logic circuit in Fig. 2.1

2.3 Generating a Top-Level Design

Part 5 of the tutorial in Appendix A shows how to create a top-level design for the *gates2* circuit. In order to use the constraint files *basys2.ucf* or *nexys2.ucf* described in Example 1 we must name the switch inputs *sw*[*i*] and the LED outputs *ld*[*i*]. This top-level design, as created in Part 5 of Appendix A is shown in Fig. 2.3. The module *gates2* in Fig. 2.3 contains the logic circuit shown in Fig. 2.1. Note that each wire connected to a bus must be labeled to identify its connection to the bus lines.

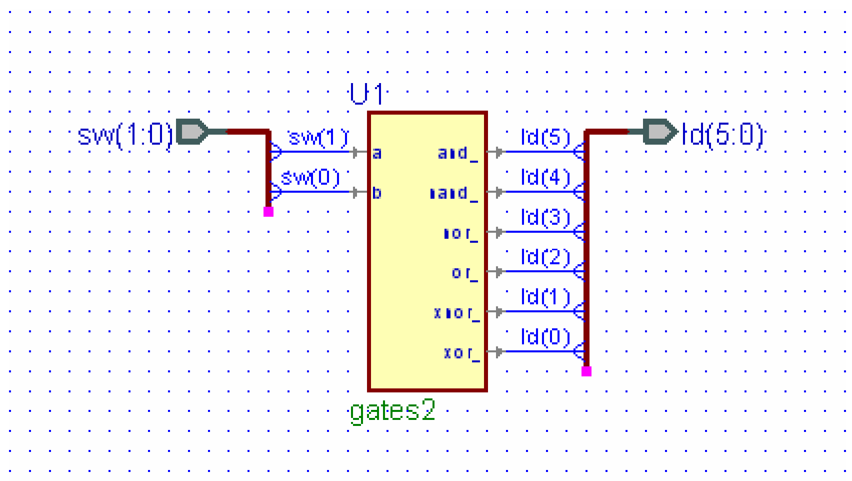


Figure 2.3 Top-level design for Example 2

Compiling the top-level design shown in Fig. 2.3 will generate the Verilog program shown in Listing 2.2. The inputs are now the two rightmost slide switches, *sw*[1:0], and the outputs are the six right-most LEDs *ld*[5:0]. To associate these inputs and outputs with the inputs *a* and *b* and the six output in the *gates2* module in Fig. 2.1 and Listing 2.1 we use the Verilog instantiation statement

```
gates2 U1
(      .a(sw[1]),
       .and_(ld[5]),
       .b(sw[0]),
       .nand_(ld[4]),
       .nor_(ld[3]),
       .or_(ld[2]),
       .xnor_(ld[1]),
       .xor_(ld[0])
);
```

This Verilog instantiation statement begins with the name of the module being instantiated, in this case *gates2* from Listing 2.1. This is followed by an arbitrary name for this module in the top-level design. Here we call it U1. Then in parentheses the inputs and outputs in Listing 2.1 are associated with corresponding inputs and outputs in the top-level design in Fig. 2.3. Note that we connect the input *a* in Listing 2.1 to the input *sw*[1] on the FPGA board. The input *b* in Listing 2.1 is connected to *sw*[0] and the outputs *and_*, *nand_*, *or_*, *nor_*, *xor_*, and *xnor_* are connected to the corresponding LED outputs *ld*[5:0].

Follow the steps in the tutorial in Appendix A and implement this design on the FPGA board. Note that when you change the settings of the two right-most slide switches the LEDs will indicate the outputs of the six gates.

Listing 2.2 gates2_top.v

```
// Example 2b: gates2_top
module gates2_top (sw,ld) ;
input wire [1:0] sw;
output wire [5:0] ld;

gates2 U1
(      .a(sw[1]),
       .and_(ld[5]),
       .b(sw[0]),
       .nand_(ld[4]),
       .nor_(ld[3]),
       .or_(ld[2]),
       .xnor_(ld[1]),
       .xor_(ld[0])
);
```

Example 3

Multiple-Input Gates

In this example we will design a circuit containing multiple-input gates. We will create a logic circuit containing 4-input AND, OR, and XOR gates. We will leave it as a problem for you to create a logic circuit containing 4-input NAND, NOR, and XNOR gates.

Prerequisite knowledge:
Appendix C – Basic Logic Gates
Appendix A – Use of Aldec Active-HDL

3.1 Behavior of Multiple-Input Gates

The AND, OR, NAND, NOR, XOR, and XNOR gates we studied in Example 1 had two inputs. The basic definitions hold for multiple inputs. A multiple-input AND gate is shown in Fig. 2.19. *The output of an AND gate is HIGH only if all inputs are HIGH*. There are three ways we could describe this multiple-input AND gate in Verilog. First we could simply write the logic equation as
.

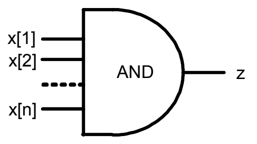


Figure 3.1
Multiple-input AND gate.

```
assign z = x[1] & x[2] & ... & x[n];                (3.1)
```

Alternatively, we could use the & symbol as a *reduction operator* by writing

```
assign z = &x;                                      (3.2)
```

This produces the same result as the statement (3.1) with much less writing. Finally, we could use the following *gate instantiation statement* for an AND gate.

```
and(z,x[1],x[2],...,x[n]);                          (3.3)
```

In this statement the first parameter in the parentheses is the name of the output port. This is followed by a list of all input signals.

A multiple-input OR gate is shown in Fig. 3.2. *The output of an OR gate is LOW only if all inputs are LOW*. Just as with the AND gate there are three ways we can describe this multiple-input OR gate in Verilog. We can write the logic equation as
.

```
assign z = x[1] | x[2] | ... | x[n];
```

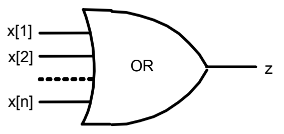


Figure 3.2
Multiple-input OR gate.

or we can use the | symbol as a *reduction operator* by writing

```
assign z = |x;
```

or we can use the following *gate instantiation statement* for an OR gate.

```
or(z,x[1],x[2],...,x[n]);
```

A multiple-input NAND gate is shown in Fig. 3.3. *The output of a NAND gate is LOW only if all inputs are HIGH*. We can write the logic equation as
.

```
assign z = ~(x[1] & x[2] & ... & x[n]);
```

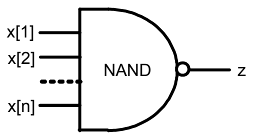


Figure 3.3
Multiple-input NAND gate.

or we can use the ~& symbol as a *reduction operator* by writing

```
assign z = ~&x;
```

or we can use the following *gate instantiation statement* for an OR gate.

```
nand(z,x[1],x[2],...,x[n]);
```

A multiple-input NOR gate is shown in Fig. 3.4. *The output of a NOR gate is HIGH only if all inputs are LOW*. We can write the logic equation as
.

```
assign z = ~(x[1] | x[2] | ... | x[n]);
```

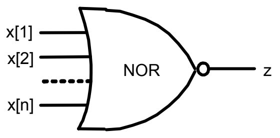


Figure 3.4
Multiple-input NOR gate.

or we can use the ~| symbol as a *reduction operator* by writing

```
assign z = ~|x;
```

or we can use the following *gate instantiation statement* for an OR gate.

```
nor(z,x[1],x[2],...,x[n]);
```

A multiple-input XOR gate is shown in Fig. 3.5. What is the meaning of this multiple-input gate? Following the methods we used for the previous multiple-input gates we can write the logic equation as
.

```
assign z = x[1] ^ x[2] ^ ... ^ x[n];
```

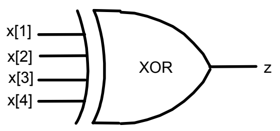


Figure 3.5
Multiple-input XOR gate.

or we can use the ^ symbol as a *reduction operator* by writing

```
assign z = ^x;
```

or we can use the following *gate instantiation statement* for an OR gate.

```
xor(z,x[1],x[2],...,x[n]);
```

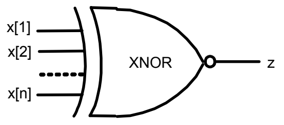


Figure 3.6
Multiple-input XNOR gate.

We will create a 4-input XOR gatge in this example to determine its meaning but first consider the multiple-input XNOR gate shown in Fig. 3.6. What is the meaning of this multiple-input gate? (See the problelm at the end of this

example for the answer.) Following the methods we used for the previous multiple-input gates we can write the logic equation as
.

```
    assign z = ~(x[1] ^ x[2] ^ ... ^ x[n]);
```

or we can use the ~^ symbol as a *reduction operator* by writing

```
    assign z = ~^x;
```

or we can use the following gate *instantiation statement* for an XOR gate.

```
    xnor(z,x[1],x[2],...,x[n]);
```

3.2 Generating the Design File *gates4.bde*

Use the block diagram editor (BDE) in Active-HDL to create the logic circuit called *gates4.bde* shown in Fig. 3.7. A simulation of this circuit is shown in Fig. 3.8. From this simulation we see that *the output of an XOR gate is HIGH only if the number of HIGH inputs is ODD*.

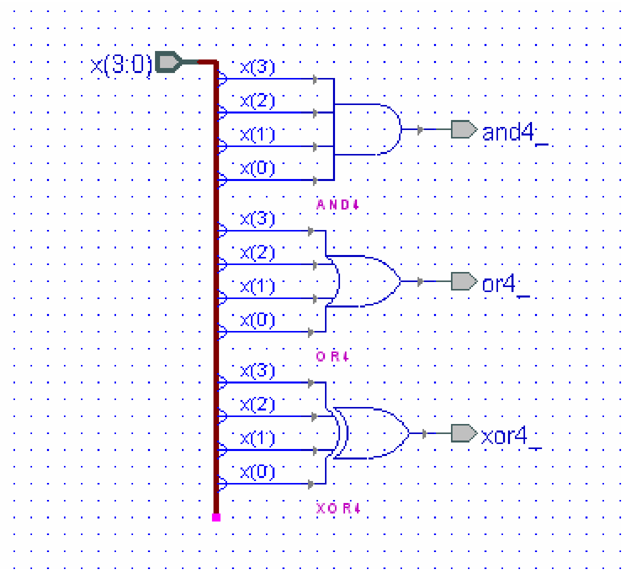


Figure 3.7 Block diagram for *gates4.bde*

If you look at the file *gates4.v* that is generated when you compile *gates4.bde* you will see that Active-HDL defines separate modules for the 4-input AND, OR, and XOR gates and then uses a Verilog instantiation statement to "wire" them together.

Alternatively, we could use the HDE editor to write the simpler Verilog program called *gates4b.v* shown in Listing 3.1 that uses *reduction operators* to implement the three 4-input gates. This Verilog program will produce the same simulation as shown in Fig. 3.8.

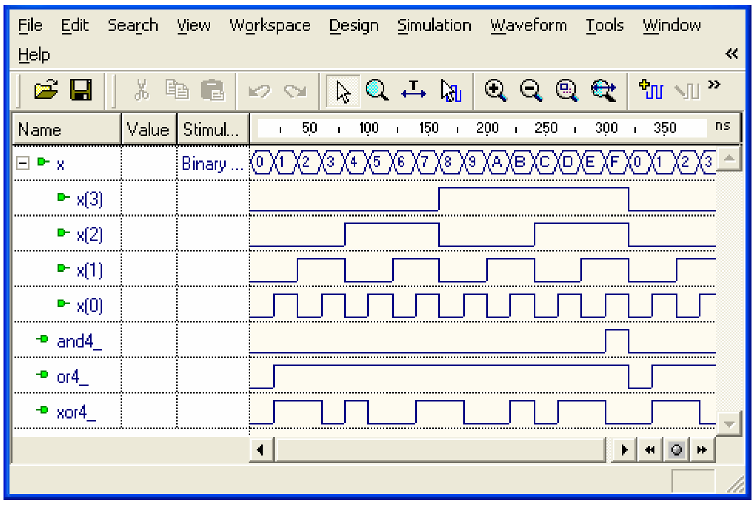

File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Name
Value
Stimul...
50 100 150 200 250 300 350 ns
x
Binary ...
0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3
x(3)
x(2)
x(1)
x(0)
and4_
or4_
xor4_

Figure 3.8 Simulation of the design *gates4.bde* shown in Fig. 3.7

Listing 3.1: gates4b.v

```
// Example 2: 4-input gates
module gates4b (
input wire [3:0] x ,
output wire and4_ ,
output wire or4_ ,
output wire xor4_
);

assign and4_ = &x;
assign or4_ = |x;
assign xor4_ = ^x;

endmodule
```

3.3 Generating the Top-Level Design *gates4_top.bde*

Fig. 3.9 shows the block diagram of the top-level design *gates4_top.bde*. The module *gates4* shown in Fig. 3.9 contains the logic circuit shown in Fig. 3.4. If you compile *gates4_top.bde* the Verilog program *gates4_top.v* shown in Listing 3.2 will be generated. Compile, synthesize, implement, and download this design to the FPGA board.

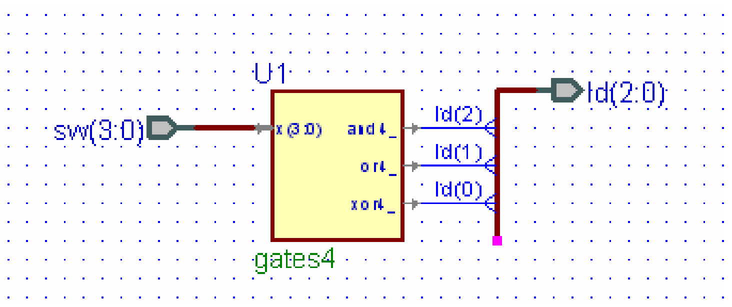


Figure 3.9 Block diagram for the top-level design *gates4_top.bde*

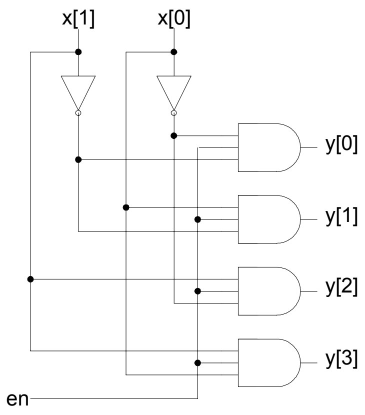
Listing 3.2: gates4_top.v

```
// Example 2: 4-input gates - top level
module gates4_top (
input wire [3:0] sw ,
output wire [2:0] ld
);

gates4 U1
(
       .and4_(ld[2]),
       .or4_(ld[1]),
       .x(sw),
       .xor4_(ld[0])
);

endmodule
```

Problem

3.1 Use the BDE to create a logic circuit containing 4-input NAND, NOR, and XNOR gates. Simulate your design and verify that *the output of an XNOR gate is HIGH only if the number of HIGH inputs is EVEN*. Create a top-level design that connects the four inputs to the rightmost four slide switches and the three outputs to the three rightmost LEDs. Implement your design and download it to the FPGA board.

3.2 The circuit shown at the right is for a 2 x 4 decoder. Use the BDE to create this circuit and simulate it using Active-HDL. Choose a counter stimulator for *x*[1:0] that counts every 20 ns, set *en* to a forced value of 1, and simulate it for 100 ns. Make a truth table with (*x*[1], *x*[0]) as the inputs and *y*[0:3] as the outputs. What is the behavior of this decoder?



Example 4

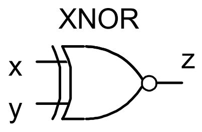
Equality Detector

In this example we will design a 2-bit equality detector using two NAND gates and an AND gate.

Prerequisite knowledge:
Appendix C – Basic Logic Gates
Appendix A – Use of Aldec Active-HDL

4.1 Generating the Design File *eqdet2.bde*

The truth table for a 2-input XNOR gate is shown in Fig. 4.1. Note that the output *z* is 1 when the inputs *x* and *y* are equal. Thus, the XNOR gate can be used as a 1-bit equality detector.



z = ~(x ^ y)
z = x ~^ y

x	y	z
0	0	1
0	1	0
1	0	0
1	1	1

Figure 4.1 The XNOR gate is a 1-bit equality detector

By using two XNOR gates and an AND gate we can design a 2-bit equality detector as shown in Fig. 4.2. Use the BDE to create the file *eqdet2.bde* using Active-HDL.

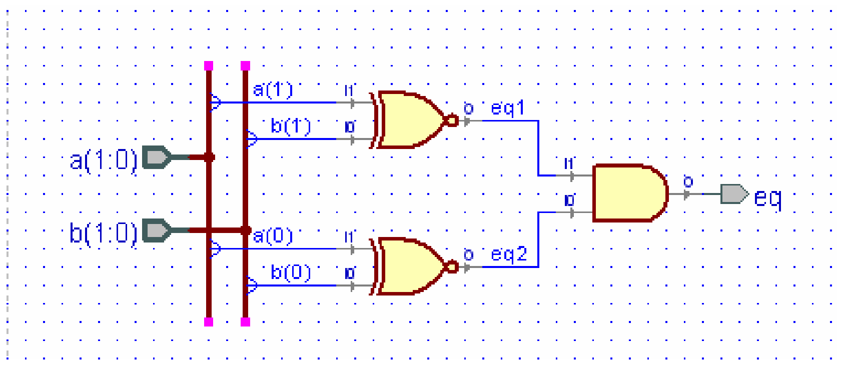


Figure 4.2 Block diagram of a 2-bit equality detector, *eqdet2.bde*

If you compile the file *eqdet2.bde* Active-HDL will generate the Verilog program *eqdet2.v* shown in Listing 4.1. A simulation of *eqdet2.bde* is shown in Fig. 4.3. Note that the output *eq* is 1 only if *a*[1:0] is equal to *b*[1:0].

Listing 4.1: eqdet2.v

```
// Title        : eqdet2
module eqdet2 (
input wire [1:0] a,
input wire [1:0] b,
output wire eq
) ;

wire eq1;
wire eq2;

assign eq1 = ~(b[1] ^ a[1]);
assign eq2 = ~(b[0] ^ a[0]);
assign eq = eq2 & eq1;

endmodule
```

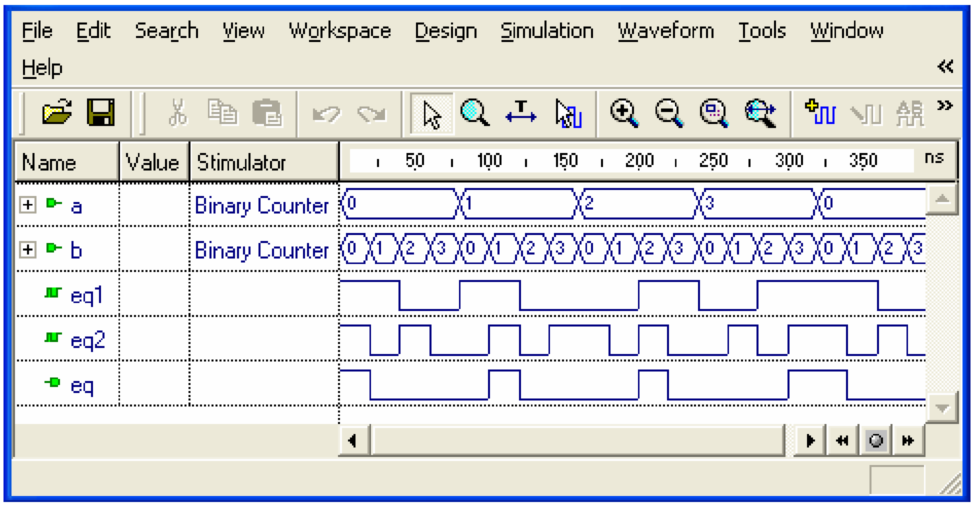

File Edit Search View Workspace Design Simulation Waveform Tools Window
Help
Name Value Stimulator
50 100 150 200 250 300 350 ns
a Binary Counter
b Binary Counter
eq1
eq2
eq

Figure 4.3 Simulation of the 2-bit equality detector, *eqdet2.bde*

Create a top-level design called *eqdet2_top.bde* that connects *a*[1:0] and *b*[1:0] to the rightmost four slide switches and connects the output *eq* to *ld*[0]. Implement your design and download it to the FPGA board.

Example 5

2-to-1 Multiplexer: *if* Statement

In this example we will show how to design a 2-to-1 multiplexer and will introduce the Verilog *if* statement. Section 5.1 will define a multiplexer and derive the logic equations for a 2-to-1 multiplexer. Section 5.2 will illustrate the use of two versions of the Verilog *if* statement.

Prerequisite knowledge:
Karnaugh Maps – Appendix D
Use of Aldec Active-HDL – Appendix A

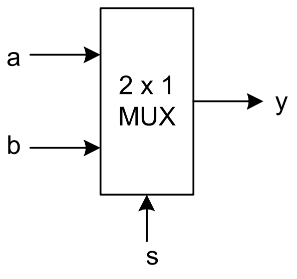
5.1 Multiplexers

An *n*-input multiplexer (called a *MUX*) is an *n*-way digital switch that switches one of *n* inputs to the output. A 2-input multiplexer is shown in Fig. 5.1. The switch is controlled by the single control line *s*. This bit selects one of the two inputs to be "connected" to the output. This means that the logical value of the output *y* will be the same as the logical value of the selected input.

From the truth table in Fig. 5.1 we see that *y* = *a* if *s* = 0 and *y* = *b* if *s* = 1. The Karnaugh map for the truth table in Fig. 5.1 is shown in Fig. 5.2. We see that the logic equation for *y* is

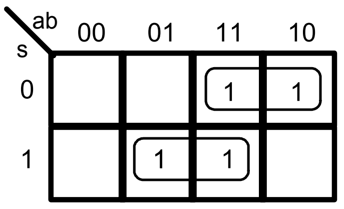
```
y =  ~s & a | s & b                   (5.1)
```

Note that this logic equation describes the circuit diagram shown in Fig. 5.3.



s	a	b	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Figure 5.1 A 2-to-1 multiplexer



y = ~s & a | s & b

Figure 5.2
K-map for a 2-to-1 multiplexer

Use the BDE to create the block diagram *mux21.bde* shown in Fig. 5.3 that implements logic equation (5.1). Compiling *mux21.bde* will generate a Verilog file, mux21.v, that is equivalent to Listing 5.1. A simulation of *mux21.bde* is shown in Fig. 5.4. Note in the simulation that *y* = *a* if *s* = 0 and *y* = *b* if *s* = 1.

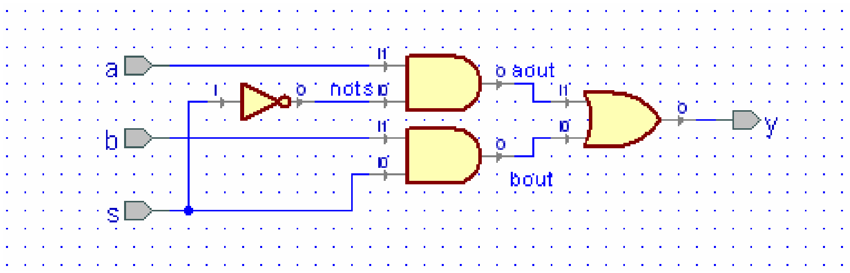
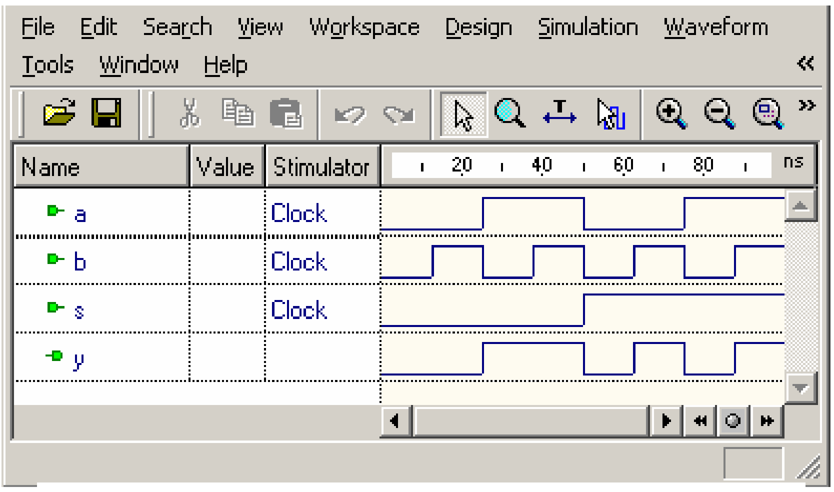


Figure 5.3 Block diagram for a 2-to-1 multiplexer, *mux21.bde*

Listing 5.1 Example5a.v

```
// Example 5a: 2-to-1 MUX using logic equations
module mux21a (
input wire a ,
input wire b ,
input wire s ,
output wire y
);

assign y = ~s & a | s & b;

endmodule
```



Name	Value	Stimulator
a		Clock
b		Clock
s		Clock
y		

Figure 5.4 Simulation of the 2-to-1 MUX in Fig. 5.3

5.2 The Verilog *if* statement

The behavior of the 2 x 1 multiplexer shown in Fig. 5.1 can be described by the Verilog statements

```
if(s == 0)
   y = a;
else
   y = b;
```

The Verilog ***if*** statement must be cont ained in an *always* block as shown in Listing 5.2. Note that *y* must be declared to be of type **reg** because it is assigned a value within the *always* block. The notation @(*) in the *always* statement is equivalent to @(*a*,*b*,*s*) where *a*, *b*, *s* is called the *sensitivity list*. Any time any of these input values change the *if* statement within the *always* block is executed. The use of the * notation is a convenience that prevents you from omitting any of the signals or inputs used in the *always* block. A Verilog program can contain more than one *always* blocks, and these *always* blocks are executed concurrently. The Verilog code in Listing 5.2 will be compiled to produce the logic circuit shown in Fig. 5.3. A simulation of the Verilog code in Listing 5.2 will produce the same waveform as shown in Fig. 5.4.

Listing 5.2 Example4b.v

```
// Example 4b: 2-to-1 MUX using if statement
module mux21b (
input wire a ,
input wire b ,
input wire s ,
output reg y
);

always @(*)
   if(s == 0)
       y = a;
   else
       y = b;

endmodule
```

Create a top-level design called *mux21_top.bde* that connects *a* and *b* to the rightmost two slide switches, connects *s* to *btn*[0], and connects the output *y* to *ld*[0]. Implement your design and download it to the FPGA board. Test the operation of the multiplexer by changing the position of the toggle switches and pressing pushbutton *btn*[0].

Example 6

Quad 2-to-1 Multiplexer

In this example we will show how to design a quad 2-to-1 multiplexer. In Section 6.1 we will make the quad 2-to-1 multiplexer by wiring together four of the 2-to-1 multiplexers that we designed in Example 5. In Section 6.2 we will show how the quad 2-to-1 multiplexer can be designed using a single Verilog *if* statement. Finally, in Section 6.3 we will show how to use a Verilog parameter to define a generic 2-to-1 multiplexer with arbitrary bus sizes.

Prerequisite knowledge:
Example 5 – 2-to-1 Multiplexer

6.1 Generating the Design File *mux42.bde*

By using four instances of the 2-to-1 MUX, *mux21.bde*, that we designed in Example 5, we can design a quad 2-to-1 multiplexer as shown in Fig. 6.1. Use the BDE to create the file *mux24.bde* using Active-HDL. Note that you will need to add the file *mux21.bde* to your project.

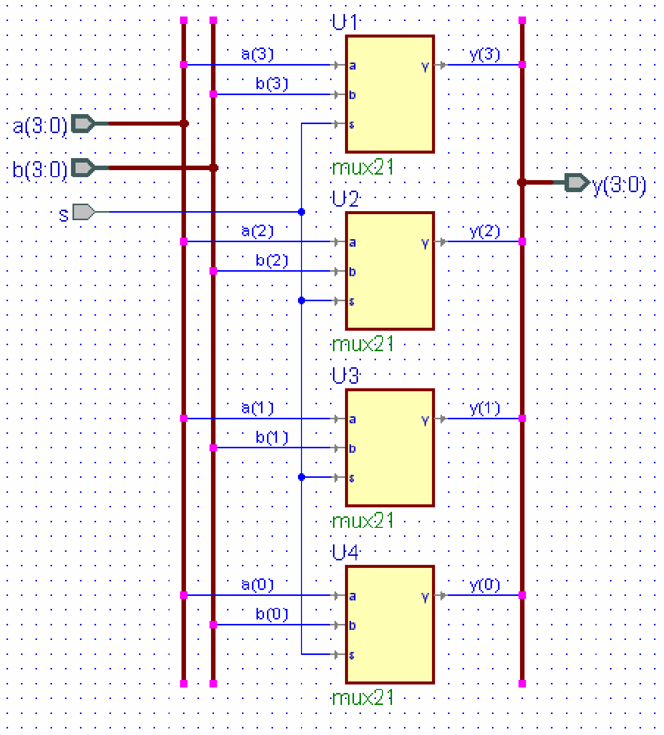


Figure 6.1 The quad 2-to-1 MUX, *mux24.bde*, contains four 2-to-1 MUXs

If you compile the file *mux24.bde* Active-HDL will generate the Verilog program *mux24.v* shown in Listing 6.1. A simulation of *mux24.bde* is shown in Fig. 6.2. Note that the output *y*[3:0] will be either *a*[3:0] or *b*[3:0] depending on the value of *s*.

Listing 6.1 Example6a.v

```
// Example 6a: mux24
module mux24 (
input wire s;
input wire [3:0] a;
input wire [3:0] b;
output wire [3:0] y;
) ;

mux21 U1
(       .a(a[3]),
        .b(b[3]),
        .s(s),
        .y(y[3])
);

mux21 U2
(       .a(a[2]),
        .b(b[2]),
        .s(s),
        .y(y[2])
);

mux21 U3
(       .a(a[1]),
        .b(b[1]),
        .s(s),
        .y(y[1])
);

mux21 U4
(       .a(a[0]),
        .b(b[0]),
        .s(s),
        .y(y[0])
);

endmodule
```

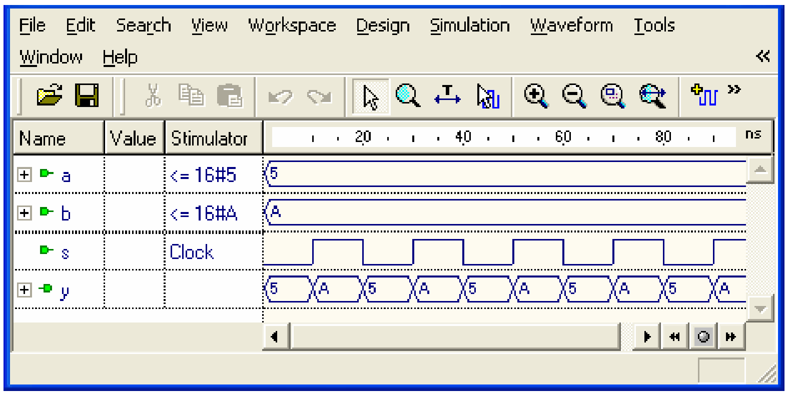


Figure 6.2 Simulation of the quad 2-to-1 MUX in Fig. 6.1

Use the BDE to create the top-level design called *mux21_top.bde* shown in Fig. 6.3. Note that *a*[3:0] are connected to the four leftmost slide switches, *b*[3:0] are connected to the rightmost four slide switches, and *y*[3:0] are connected to the rightmost LEDs. Also note that *s* is connected to *btn*[0], and the input *btn*[0:0] must be declared as an array, even though there is only one element, so that we can use the constraint file *basys2.ucf* or *nexys2.ucf* without change. Implement your design and download it to the FPGA board. Test the operation of the quad 2-to-1 multiplexer by setting the switch values and pressing pushbutton *btn*[0].

If you compile the file *mux24_top.bde* Active-HDL will generate the Verilog program *mux24_top.v* shown in Listing 6.2. A simulation of *mux24_top.bde* is shown in Fig. 6.4.

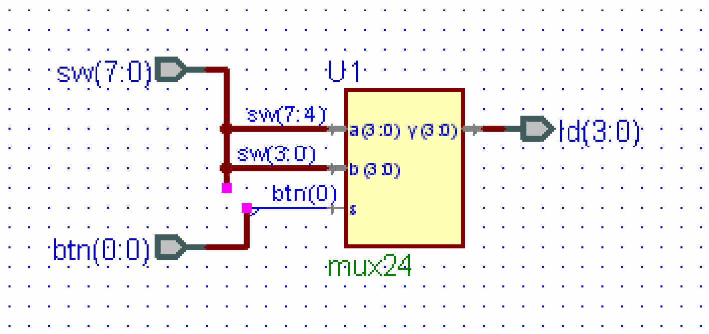


Figure 6.3 Top-level design for testing the quad 2-to-1 MUX

Listing 6.2 Example6b.v

```
// Example 6b: mux24_top
module mux24_top (
input wire [0:0] btn;
input wire [7:0] sw;
output wire [3:0] ld;
) ;

mux24 U1
(      .a(sw[7:4]),
       .b(sw[3:0]),
       .s(btn[0]),
       .y(ld)
);
endmodule
```

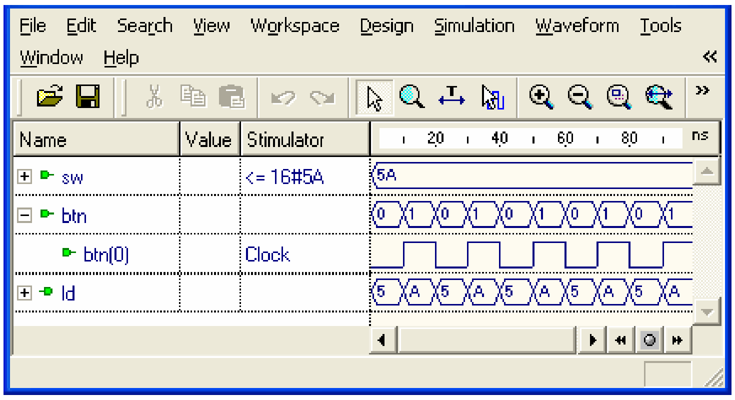


Figure 6.4 Simulation of *mux24_top.bde* in Fig. 6.1

6.2 A Quad 2-to-1 Multiplexer Using an *if* Statement

In Listing 5.2 of Example 5 we used a Verilog *if* statement to implement a 2-to-1 MUX. Listing 6.3 is a direct extension of Listing 5.2 where now the inputs and outputs are 4-bit values rather that a single bit. The Verilog program shown in Listing 6.3 will produce the same simulation as shown in Fig. 6.2. The module *mux24b* defined by the Verilog program in Listing 6.3 could be used in place of the *mux24* module in the top-level design in Fig. 6.3

Listing 6.3 mux24b.v

```
// Example 6c: Quad 2-to-1 mux using if statement
module mux24b(
input wire [3:0] a,
input wire [3:0] b,
input wire s,
output reg [3:0] y
);

always @(*)
      if(s == 0)
         y = a;
      else
         y = b;

endmodule
```

6.3 Generic Multiplexers: Parameters

We can use the Verilog parameter statement to design a generic 2-to-1 multiplexer with input and output bus widths of arbitrary size. Listing 6.4 shows a Verilog program for a generic 2-to-1 MUX.

Note the use of the *parameter* statement that defines the bus width *N* to have a default value of 4. This value can be overridden when the multiplexer is instantiated as shown in Listing 6.5 for an 8-line 2-to-1 multiplexer called *M8*. The parameter override clause is automatically included in the module instantiation statement when you copy it in Active-HDL as shown in Listing 6.5. We will always use upper-case names for parameters. The simulation of Listing 6.5 is shown in Fig. 6.5.

If you compile the Verilog program *mux2g.v* shown in Listing 6.4 it will generate a block diagram for this module when you go to BDE. If you right-click on the symbol for *mux2g* and select *Properties*, you can change the default value of the parameter *N* by selecting the *Parameters* tab and entering an actual value for *N*.

Listing 6.4 mux2g.v

```
// Example 6d: Generic 2-to-1 MUX using a parameter
module mux2g
#(parameter N = 4)
(input wire [N-1:0] a,
 input wire [N-1:0] b,
 input wire s,
 output reg [N-1:0] y
);

always @(*)
      if(s == 0)
         y = a;
      else
         y = b;

endmodule
```

Listing 6.5 mux28.v

```
// Example 6e: 8-line 2-to-1 MUX using a parameter
module mux28(
input wire [7:0] a,
input wire [7:0] b,
input wire s,
output wire [7:0] y
);

mux2g #(
  .N(8))
M8  (.a(a),
  .b(b),
  .s(s),
  .y(y)
);

 endmodule
```

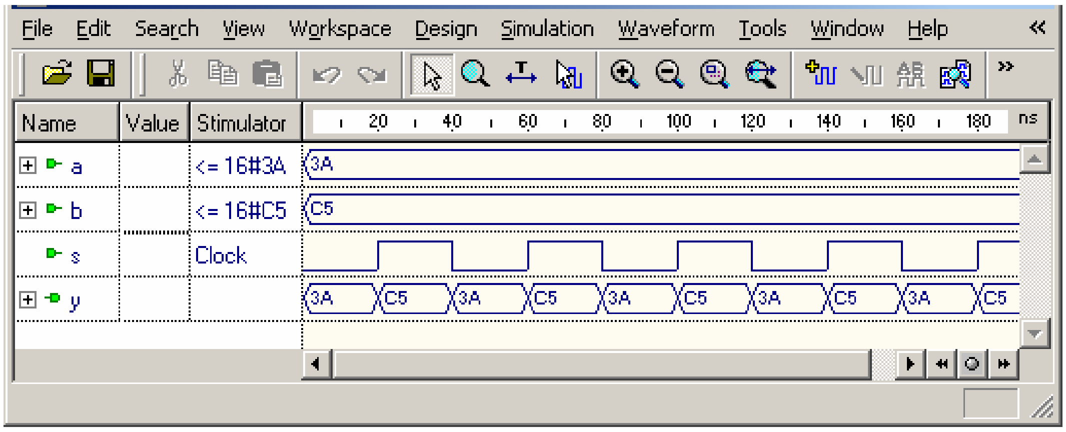

File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Name Value Stimulator
20 40 60 80 100 120 140 160 180 ns
a <= 16#3A 3A
b <= 16#C5 C5
s Clock
y 3A C5 3A C5 3A C5 3A C5 3A C5

Figure 6.5 Simulation result from the Verilog program in Listing 6.5

Example 7

4-to-1 Multiplexer

In this example we will show how to design a 4-to-1 multiplexer. In Section 7.1 we will make a 4-to-1 multiplexer by wiring together three of the 2-to-1 multiplexers that we designed in Example 5. In Section 7.2 we will derive the logic equation for a 4-to-1 MUX. In Section 7.3 we will show how a 4-to-1 multiplexer can be designed using a single Verilog *case* statement and in Section 7.4 we design a quad 4-to-1 multiplexer.

Prerequisite knowledge:
Example 5 – 2-to-1 Multiplexer

7.1 Designing a 4-to-1 MUX Using 2-to-1 Modules

A 4-to-1 multiplexer has the truth table shown in Fig. 7.1 By using three instances of the 2-to-1 MUX, *mux21.bde*, that we designed in Example 5, we can design a 4-to-1 multiplexer as shown in Fig. 7.2. Use the BDE to create the file *mux41.bde* using Active-HDL. Note that you will need to add the file *mux21.bde* to your project.

s1	s0	z
0	0	c0
0	1	c1
1	0	c2
1	1	c3

Figure 7.1
Truth table for a 4-to-1 MUX

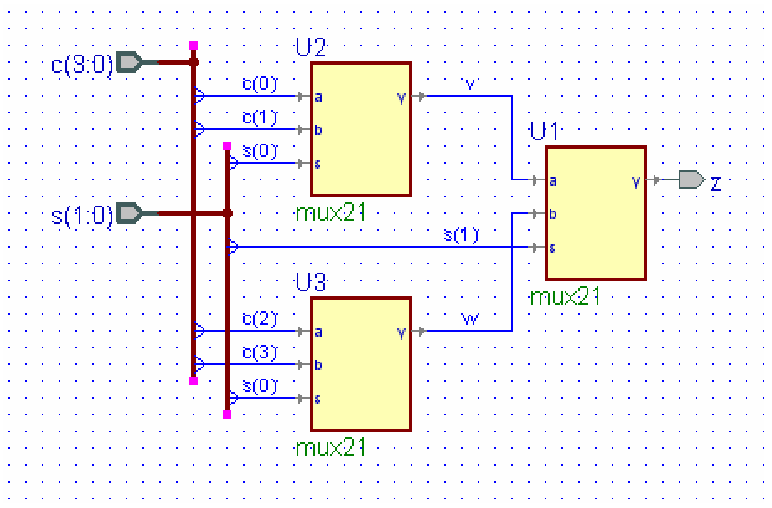


Figure 7.2 The 4-to-1 MUX, *mux41.bde*, contains four 2-to-1 MUXs

In Fig. 7.2 when *s*[1] = 0 it is *v*, the output of U2 that gets through to *z*. If *s*[0] = 0 in U2 then it is *c*[0] that gets through to *v* and therefore to *z*. If *s*[0] = 1 in U2 then it is *c*[1] that gets through to *v* and therefore to *z*.

If, on the other hand, *s*[1] = 1 in U1 then it is *w*, the output of U3 that gets through to *z*. If *s*[0] = 0 in U3 then it is *c*[2] that gets through to *w* and therefore to *z*. If *s*[0] = 1 in U3 then it is *c*[3] that gets through to *w* and therefore to *z*. Thus you can see that the circuit in Fig. 7.2 will implement the truth table in Fig. 7.1.

When you compile the file *mux41.bde* Active-HDL will generate the Verilog program *mux41.v* shown in Listing 7.1. A simulation of *mux41.bde* is shown in Fig. 7.3. Note that the output *z* will be one of the four inputs *c*[3:0] depending on the value of *s*[1:0].

Listing 7.1 mux41.v

```
// Example 7a: 4-to-1 MUX using module instantiation
module mux41 (
input wire [3:0] c ,
input wire [1:0] s ,
output wire z
);

// Internal signals
wire v;      // output of mux M1
wire w;      // output of mux M2

// Module instantiations
mux21 U1
(      .a(v),
       .b(w),
       .s(s[1]),
       .y(z)
);

mux21 U2
(      .a(c[0]),
       .b(c[1]),
       .s(s[0]),
       .y(v)
);

mux21 U3
(      .a(c[2]),
       .b(c[3]),
       .s(s[0]),
       .y(w)
);
endmodule
```

If you were going to create this top-level design using HDE instead of BDE you would begin by defining the inputs *c*[3:0] and *s*[1:0] and the output *z* and the two wires *v* and *w*. You would then "wire" the three modules together using the three *module instantiation statements* shown in Listing 7.1.

The easiest way to generate this *module instantiation* statement is to first compile the file *mux21.v* from Example 5 using Active-HDL, expand the library icon (click the plus sign), right click on *mux21*, and select *Copy Verilog Instantiation* as shown in Fig. 7.4. Paste this into your top-level *mux41.v* file.

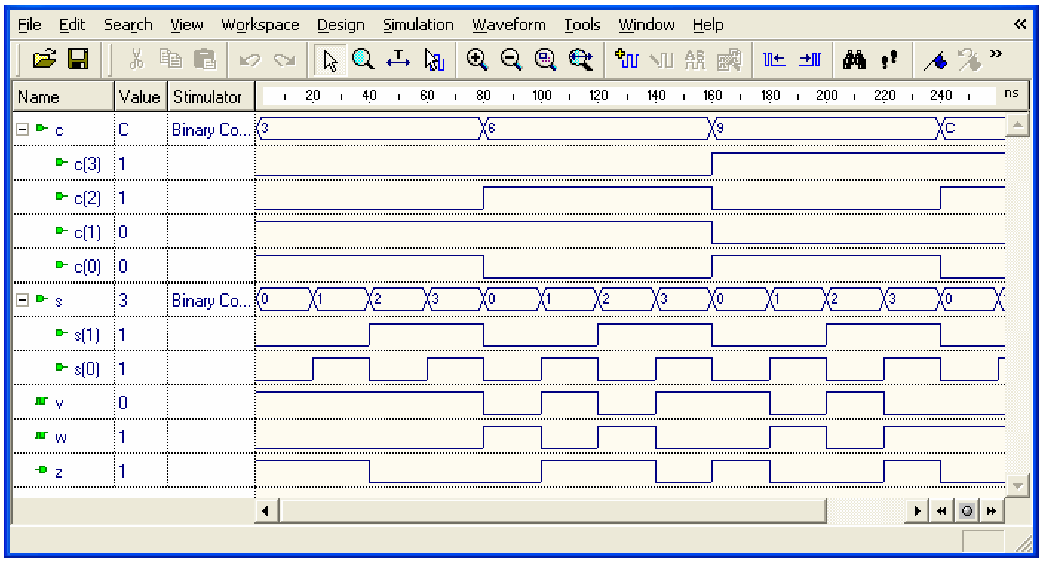
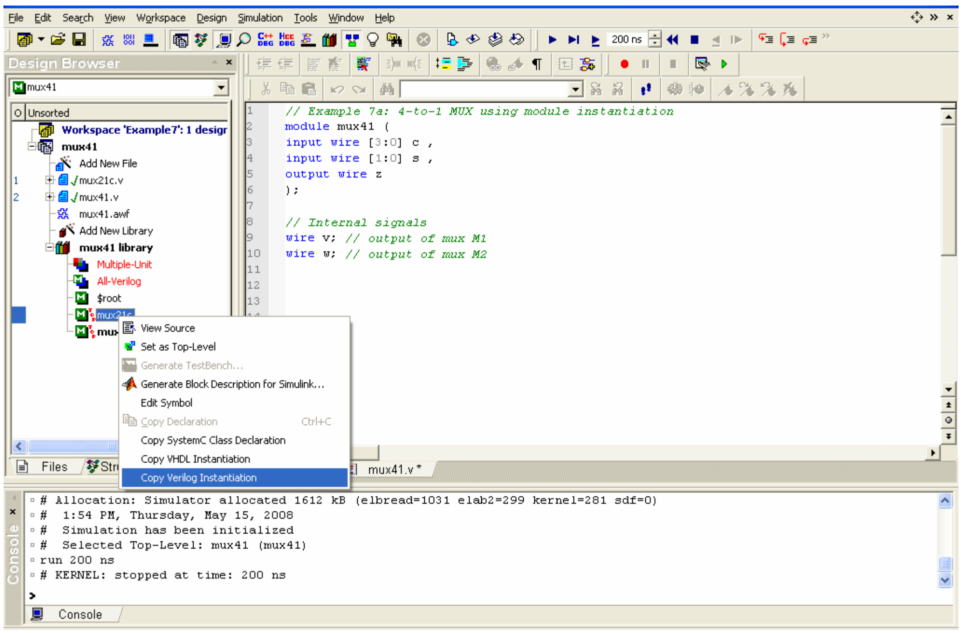

File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Name
Value
Stimulator
c
C
Binary Co...
c(3)
c(2)
c(1)
c(0)
s
3
v
w
z
ns

Figure 7.3 Simulation of the Verilog program in Listing 7.1



```
// Example 7a: 4-to-1 MUX using module instantiation
module mux41 (
input wire [3:0] c ,
input wire [1:0] s ,
output wire z
);

// Internal signals
wire v; // output of mux M1
wire w; // output of mux M2
```

```
# Allocation: Simulator allocated 1612 kB (elbread=1031 elab2=299 kernel=281 sdf=0)
#  1:54 PM, Thursday, May 15, 2008
#  Simulation has been initialized
#  Selected Top-Level: mux41 (mux41)
run 200 ns
# KERNEL: stopped at time: 200 ns
>
```

Figure 7.4 Generating a module instantiation prototype

At this point you would have the statement

```
mux21 Label1 (.a(a),
        .b(b),
        .s(s),
        .y(y)
);
```

Make three copies of this prototype and change the name of *Label1* to U1, U2, and U3 in the three statements. Now you just "wire up" each input and output variable by changing the values in the parentheses to the signal that it is connected to. For example, the mux U1 input *a* is connected to the wire *v* so we would write `.a(v)`. In a similar way the mux input *b* is connected to wire *w* and the mux input *s* is connected to input *s*[1]. The mux output *y* is connected to the output *z* in Fig. 7.2. Thus, the final version of this *module instantiation* statement would be

```
mux21 U1 (.a(v),
        .b(w]),
        .s(s[1]),
        .y(z)
);
```

The other two modules, U2 and U3, are "wired up" using similar module instantiation statements.

7.2 The Logic Equation for a 4-to-1 MUX

The 4-to-1 MUX designed in Fig. 7.2 can be represented by the logic symbol shown in Fig. 7.5. This multiplexer acts like a digital switch in which one of the inputs *c*[3:0] gets connected to the output *z*. The switch is controlled by the two control lines *s*[1:0]. The two bits on these control lines select one of the four inputs to be "connected" to the output. Note that we constructed this 4-to-1 multiplexer using three 2-to-1 multiplexers in a tree fashion as shown in Fig. 7.2.

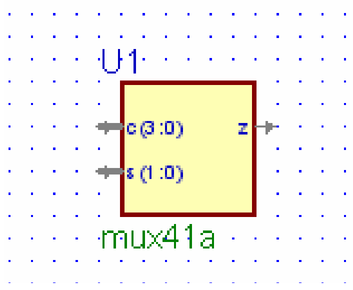


Figure 7.5 A 4-to-1 multiplexer

Recall from Eq. (5.1) in Example 5 that the logic equation for a 2-to-1 MUX is given by

```
y = ~s & a | s & b                    (7.1)
```

Applying this equation to the three 2-to-1 MUXs in Fig. 7.2 we can write the equations for that 4 x 1 MUX as follows.

```
v = ~s0 & c0 | s0 & c1

w = ~s0 & c2 | s0 & c3

z = ~s1 & v | s1 & w

z = ~s1 & (~s0 & c0 | s0 & c1) | s1 & (~s0 & c2 | s0 & c3)
```

or

```
z = ~s1 & ~s0 & c0
  | ~s1 &  s0 & c1                          (7.2)
  |  s1 & ~s0 & c2
  |  s1 &  s0 & c3
```

Equation (7.2) for *z* also follows from the truth table in Fig. 7.1. Note that the tree structure in Fig. 7.2 can be expanded to implement an 8-to-1 multiplexer and a 16-to-1 multiplexer.

A Verilog program that implements a 4-to-1 MUX using the logic equation (7.2) is given in Listing 7.2. A simulation of this program will produce the same result as in Fig. 7.3 (without the wire signals *v* and *w*).

Listing 7.2 mux41b.v

```
// Example 7b: 4-to-1 MUX using logic equation
module mux41b (
input wire [3:0] c ,
input wire [1:0] s ,
output wire z
);

assign z = ~s[1] & ~s[0] & c[0]
         | ~s[1] &  s[0] & c[1]
         |  s[1] & ~s[0] & c[2]
         |  s[1] &  s[0] & c[3];

endmodule
```

7.3 4-to-1 Multiplexer: *case* Statement

The same 4-to-1 multiplexer defined by the Verilog program in Listing 7.2 can be implemented using a Verilog *case* statement. The Verilog program shown in Listing 7.3 does this. The *case* statement in Listing 7.3 directly implements the definition of a 4-to-1 MUX given by the truth table in Fig. 7.1. The *case* statement is an example of a *procedural statement* that must be within an *always* block. A typical line in the *case* statement, such as

```
2: z = c[2];
```

will assign the value of *c*[2] to the output *z* when the input value *s*[1:0] is equal to 2 (binary 10). Note that the output *z* must be of type *reg* because its value is assigned within an *always* clause.

In the *case* statement the alternative value preceding the colon in each line represents the value of the *case* parameter, in this case the 2-bit input *s*. These values are decimal values by default. If you want to write a hex value you precede the number with '*h* as in '*hA* which is a hex value *A*. Similarly, a binary number is preceded with a '*b* as in '*b*1010 which has the same value (10) as '*hA*. Normally, binary numbers are preceded with the number of bits in the number such as 4'*b*107. Using this notation, the number 8'*b*110011 will be the binary number 00110011.

Listing 7.3 mux41c.v

```
// Example 7c:  4-to-1 MUX using case statement
module mux41c (
input wire [3:0] c ,
input wire [1:0] s ,
output reg z
);

always @(*)
      case(s)
            0: z = c[0];
            1: z = c[1];
            2: z = c[2];
            3: z = c[3];
            default: z = c[0];
      endcase
endmodule
```

All *case* statements should include a *default* line as shown in Listing 7.3. This is because all cases need to be covered and while it looks as if we covered all cases in Listing 7.3, Verilog actually defines *four* possible values for each bit, namely 0 (logic value 0), 1 (logic value 1), *Z* (high impedance), and *X* (unkown value).

A simulation of the program in Listing 7.3 will produce the same result as in Fig. 7.3 (without the wire signals *v* and *w*).

7.4 A Quad 4-to-1 Multiplexer

To make a quad 4-to-1 multiplexer we could combine four 4-to-1 MUXs as we did for a quad 2-to-1 multiplexer module in Fig. 6.1 of Example 6. However, it will be easier to modify the *case* statement program in Listing 7.3 to make a quad 4-to-1 MUX. Because we will use it in Example 10 we will define a single 16-bit input *x*[15:0] and we will multiplex the four hex digits making up this 16-bit value.

Listing 7.4 is a Verilog program for this quad 4-to-1 multiplexer. Note that the four hex digits making up the 16-bit value of *x*[15:0] are multiplexed to the output *z*[3:0] depending of the value of the control signal *s*[1:0]. A simulation of this quad 4-to-1 multiplexer is shown in Fig. 7.6 and its BDE symbol is shown in Fig. 7.7.

Listing 7.4 mux44.v

```
// Example 7d: quad 4-to-1 MUX
module mux44 (
input wire [15:0] x ,
input wire [1:0] s ,
output reg [3:0] z
);

always @(*)
      case(s)
          0: z = x[3:0];
          1: z = x[7:4];
          2: z = x[11:8];
          3: z = x[15:12];
          default: z = x[3:0];
      endcase

endmodule
```

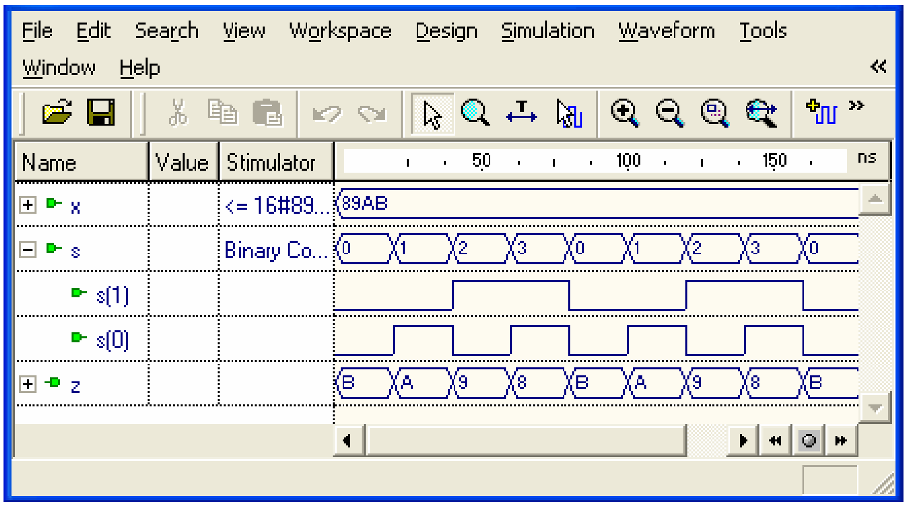


Figure 7.6 Simulation of the quad 4-to-1 MUX in Listing 7.4

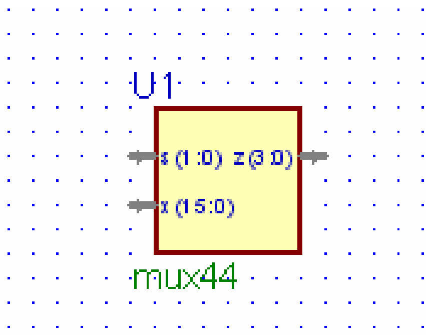


Figure 7.7 A quad 4-to-1 multiplexer

Example 8

Clocks and Counters

The Nexys-2 board has an onboard 50 MHz clock. The BASYS board has a jumper that allows you to set the clock to 100 MHz, 50 MHz, or 25 MHz. All of the examples in this book will assume an input clock frequency of 50 MHz. If you are using the BASYS board you should remove the clock jumper, which will set the clock frequency to 50 MHz. This 50 MHz clock signal is a square wave with a period of 20 ns. The FPGA pin associated with this clock signal is defined in the constraints file *basys2.ucf* or *nexys2.ucf* with the name *mclk*.

In this example we will show how to design an *N*-bit counter in Verilog and how to use a counter to generate clock signals of lower frequencies.

Prerequisite knowledge:

Appendix A – Use of Aldec Active-HDL

8.1 *N*-Bit Counter

The BDE symbol for an *N*-bit counter is shown in Fig. 8.1. If the input *clr* = 1 then all *N* of the outputs *q*[*i*] are cleared to zero asynchronously, i.e., regardless of the value of the input *clk*. If *clr* = 0, then on the next rising edge of the clock input *clk* the *N*-bit binary output *q*[*N*-1:0] will be incremented by 1. That is, on the rising edge of the clock the *N*-bit binary output *q*[*N*-1:0] will count from 0 to *N*-1 and then wrap around to 0.

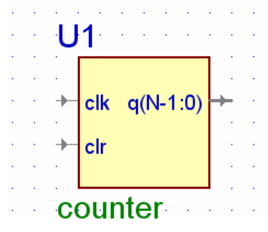


Figure 8.1 An *N*-bit counter

The Verilog program shown in Listing 8.1 was used to generate the symbol shown in Fig. 8.1. Note that the sensitivity list of the always statement contains the phrase

```
posedge clk or posedge clr
```

This means that the *if* statement within the *always* block will execute whenever either *clr* or *clk* goes high. If *clr* goes high then the output *q*[*N*-1:0] will go to zero. On the other hand if *clr* = 0 and *clk* goes high then the output *q*[*N*-1:0] will be incremented by 1.

The default value of the parameter *N* in Listing 8.1 is 4. A simulation of this 4-bit counter is shown in Fig. 8.2. Note that this counter counts from 0 to F and then wraps around to 0. To instantiate an 8-bit counter from Listing 8.1 that would count from 0 – 255 (or 00 – FF hex) you would use an instantiation statement something like

```
counter #(
    .N(8))
cnt16 (.clr(clr),
    .clk(clk),
    .q(q)
);
```

You can also set the value of the parameter *N* from the block diagram editor (BDE) by right-clicking on the symbol in Fig. 8.1 and selecting *Properties* and then the *Parameters* tab.

Listing 8.1 counter.v

```
// Example 8a: N-bit counter
module counter
#(parameter N = 4)
(input wire clr ,
 input wire clk ,
 output reg [N-1:0] q
);

//    N-bit counter
always @(posedge clk or posedge clr)
  begin
    if(clr == 1)
       q <= 0;
    else
       q <= q + 1;
  end

endmodule
```

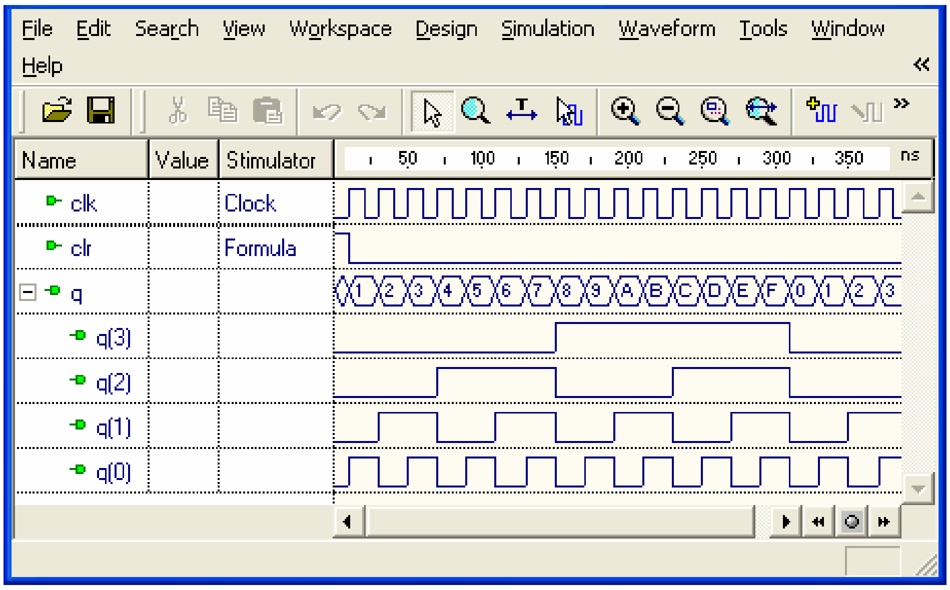


Figure 8.2 Simulation of a 4-bit counter using Listing 8.1

In the simulation in Fig. 8.2 note that the output *q*[0] is a square wave at half the frequency of the input *clk*. Similarly, the output *q*[1] is a square wave at half the frequency of the input *q*[0], the output *q*[2] is a square wave at half the frequency of the input *q*[1], and the output *q*[3] is a square wave at half the frequency of the input *q*[2]. Note how the binary numbers *q*[3:0] in Fig. 8.2 count from 0000 to 1111.

The simulation shown in Fig. 8.2 shows how we can obtain a lower clock frequency by simply using one of the outputs *q*[*i*]. We will use this feature to produce a 24-bit clock divider in the next section.

8.2 Clock Divider

The simulation in Fig. 8.2 shows that the outputs *q*[*i*] of a counter are square waves where the output *q*[0] has a frequency half of the clock frequency, the output *q*[1] has a frequency half of *q*[0], etc. Thus, a counter can be used to divide the frequency *f* of a clock, where the frequency of the output *q*(*i*) is $f_i = f / 2^{i+1}$. The frequencies and periods of the outputs of a 24-bit counter driven by a 50 MHz clock are shown in Table 8.1. Note in Table 8.1 that the output *q*[0] has a frequency of 25 MHz, the output *q*[17] has a frequency of 190.73 Hz, and the output *q*[23] has a frequency of 2.98 Hz.

Table 8.1 Clock divide frequencies

q[*i*]	Frequency (Hz)	Period (ms)
i	50000000.00	0.00002
0	25000000.00	0.00004
1	12500000.00	0.00008
2	6250000.00	0.00016
3	3125000.00	0.00032
4	1562500.00	0.00064
5	781250.00	0.00128
6	390625.00	0.00256
7	195312.50	0.00512
8	97656.25	0.01024
9	48828.13	0.02048
10	24414.06	0.04096
11	12207.03	0.08192
12	6103.52	0.16384
13	3051.76	0.32768
14	1525.88	0.65536
15	762.94	1.31072
16	381.47	2.62144
17	190.73	5.24288
18	95.37	10.48576
19	47.68	20.97152
20	23.84	41.94304
21	11.92	83.88608
22	5.96	167.77216
23	2.98	335.54432

The Verilog program shown in Listing 8.2 is a 24-bit counter that has three outputs, a 25 MHz clock (*clk*25), a 190 Hz clock (*clk*190), and a 3 Hz clock (*clk*3). You can modify this *clkdiv* module to produce any output frequency given in Table 8.1. We will use such a clock divider module in many of our top-level designs.

Listing 8.2 clkdiv.v

```
// Example 8b: clock divider
module clkdiv (
input wire clk ,
input wire clr ,
output wire clk190 ,
output wire clk25 ,
output wire clk3
);
reg [23:0] q;

//  24-bit counter
always @(posedge clk or posedge clr)
  begin
    if(clr == 1)
      q <= 0;
    else
      q <= q + 1;
  end

assign clk190 = q[17];     // 190 Hz
assign clk25 = q[0];       // 25 MHz
assign clk3 = q[23];       // 3 Hz

endmodule
```

Note in Listing 8.2 that we define the internal signal *q*[23:0] of type *reg*. It must be of type *reg* because its value is assigned within an *always* block. The BDE symbol generated by compiling Listing 8.2 is shown in Fig. 8.3. You can edit either Listing 8.2 or the block diagram shown in Fig. 8.3 to bring out only the clock frequencies you need in a particular design. For example, the top-level design shown in Fig. 8.4 will cause the eight LEDs on the FPGA board to count in binary at a rate of about three counts per second. The corresponding top-level Verilog program is shown in Listing 8.3.

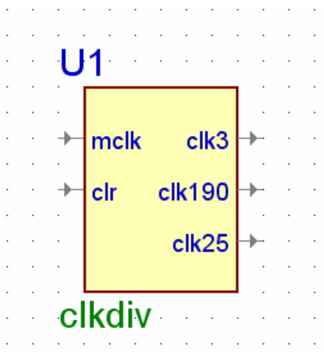


Figure 8.3 A clock divider

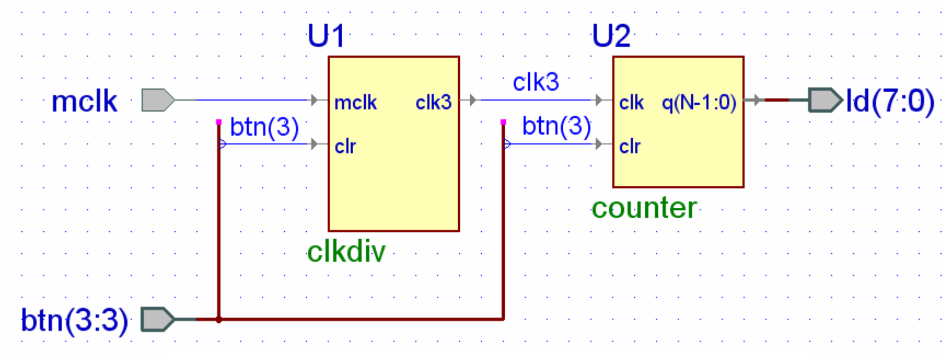


Figure 8.4 Counting in binary on the eight LEDs

Listing 8.3 count8_top.v

```
// Example 8c: count8_top
module count8_top (
input wire mclk;
input wire [3:3] btn;
output wire [7:0] ld;
) ;

wire clk3;

clkdiv U1
(       .clk3(clk3),
        .clr(btn[3]),
        .mclk(mclk)
);

counter
#(      .N(8))
U2
(       .clk(clk3),
        .clr(btn[3]),
        .q(ld[7:0])
);

endmodule
```

Internally, a counter contains a collection of flip-flops. We saw in Fig. 1 of the *Introduction* that each of the four slices in a CLB of a Spartan3E FPGA contains two flip-flops. Such flip-flops are central to the operation of all synchronous sequential circuits in which changes take place on the rising edge of a clock. The examples in the second half of this book will involve sequential circuits beginning with an example of an edge-triggered D flip-flop in Example 16.

Example 9

7-Segment Decoder

In this section we will show how to design a 7-segment decoder using Karnaugh maps and write a Verilog program to implement the resulting logic equations. We will also solve the same problem using a Verilog *case* statement.

Prerequisite knowledge:

Karnaugh maps – Appendix D
case statement – Example 7
LEDs – Example 1

9.1 7-Segment Displays

Seven LEDs can be arranged in a pattern to form different digits as shown in Fig. 9.1. Digital watches use similar 7-segment displays using liquid crystals rather than LEDs. The red digits on digital clocks are LEDs. Seven segment displays come in two flavors: *common anode* and *common cathode*. A common anode 7-segment display has all of the anodes tied together while a common cathode 7-segment display has all the cathodes tied together as shown in Fig. 9.1.

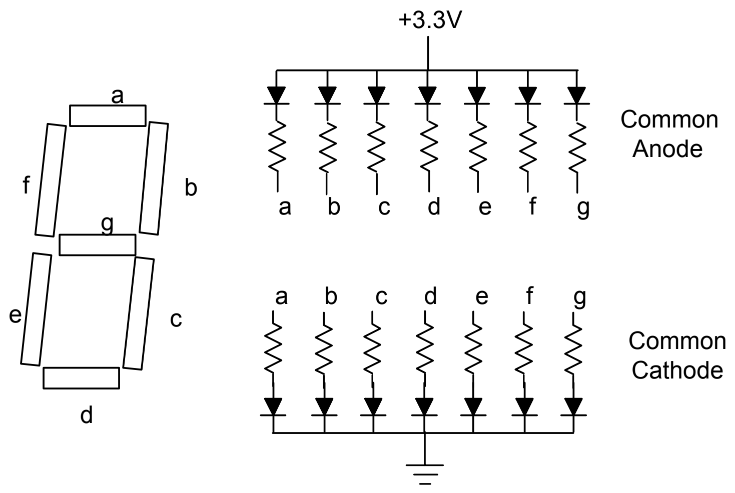


Figure 9.1 A 7-segment display contains seven light emitting diodes (LEDs)

The BASYS and Nexys2 boards have four common-anode 7-segment displays. This means that all the anodes are tied together and connected through a *pnp* transistor to +3.3V. A different FPGA output pin is connected through a 100Ω current-limiting resistor to each of the cathodes, *a* – *g*, plus the decimal point. In the common-anode case, an output 0 will turn on a segment and an output 1 will turn it off. The table shown in

Fig. 9.2 shows output cathode values for each segment *a* – *g* needed to display all hex values from 0 – F.

x	a	b	c	d	e	f	g
0	0	0	0	0	0	0	1
1	1	0	0	1	1	1	1
2	0	0	1	0	0	1	0
3	0	0	0	0	1	1	0
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	0
6	0	1	0	0	0	0	0
7	0	0	0	1	1	1	1
8	0	0	0	0	0	0	0
9	0	0	0	0	1	0	0
A	0	0	0	1	0	0	0
b	1	1	0	0	0	0	0
C	0	1	1	0	0	0	1
d	1	0	0	0	0	1	0
E	0	1	1	0	0	0	0
F	0	1	1	1	0	0	0

1 = off

0 = on

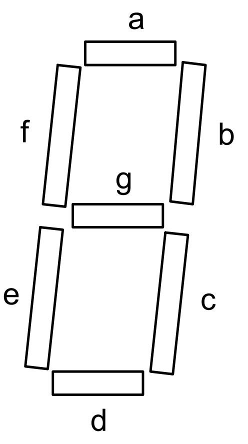


Figure 9.2 Segment values required to display hex digits 0 – F

9.2 7-Segment Decoder: Logic Equations

The problem is to design a *hex to 7-segment decoder*, called *hex7seg*, that is shown in Fig. 9.3. The input is a 4-bit hex number, *x*[3:0], and the outputs are the 7-segment values *a* – *g* given by the truth table in Fig. 9.2. We can make a Karnaugh map for each segment and then write logic equations for the segments *a* – *g*. For example, the K-map for the segment, *e*, is shown in Figure 9.4.

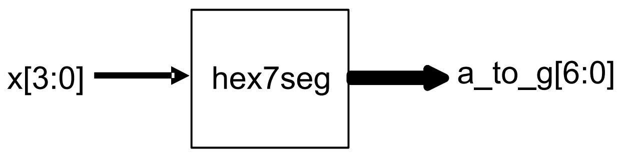


Figure 9.3 A hex to 7-segment decoder

e = ~x3 & x0 | ~x3 & x2 & ~x1 | ~x2 & ~x1 & x0

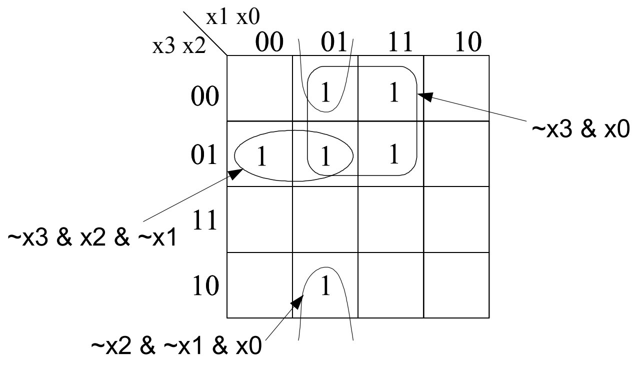
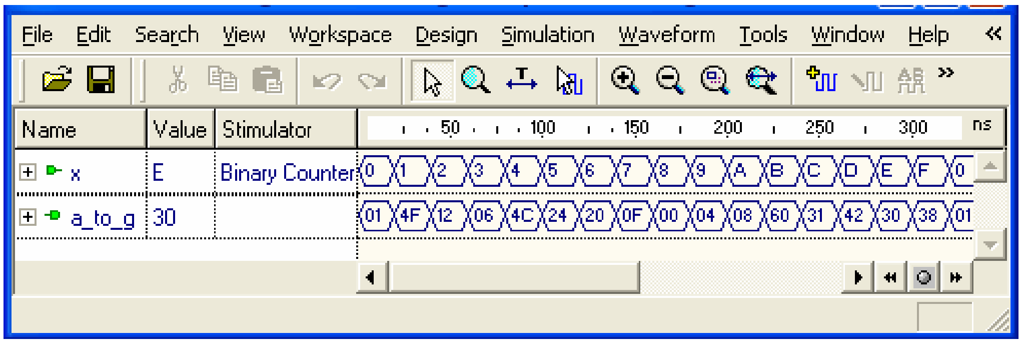


Figure 9.4 K-map for the segment *e* in the 7-segment decoder

You can write the Karnaugh maps for the other six segments and then write the Verilog program for the 7-segment decoder shown in Listing 9.1. A simulation of this program is shown in Fig. 9.5. Note that the simulation agrees with the truth table in Fig. 9.2.

Listing 9.1 hex7seg_le.v

```
// Example 9a: Hex to 7-segment decoder; a-g active low
module hex7seg_le (
input wire [3:0] x ,
output wire [6:0] a_to_g
);

assign a_to_g[6] = ~x[3] & ~x[2] & ~x[1] & x[0]   // a
              | ~x[3] & x[2] & ~x[1] & ~x[0]
              | x[3] & x[2] & ~x[1] & x[0]
              | x[3] & ~x[2] & x[1] & x[0];
assign a_to_g[5] = x[2] & x[1] & ~x[0]           // b
              | x[3] & x[1] & x[0]
              | ~x[3] & x[2] & ~x[1] & x[0]
              | x[3] & x[2] & ~x[1] & ~x[0];
assign a_to_g[4] = ~x[3] & ~x[2] & x[1] & ~x[0]   // c
              | x[3] & x[2] & x[1]
              | x[3] & x[2] & ~x[0];
assign a_to_g[3] = ~x[3] & ~x[2] & ~x[1] & x[0]   // d
              | ~x[3] & x[2] & ~x[1] & ~x[0]
              | x[3] & ~x[2] & x[1] & ~x[0]
              | x[2] & x[1] & x[0];
assign a_to_g[2] = ~x[3] & x[0]                   // e
              | ~x[3] & x[2] & ~x[1]
              | ~x[2] & ~x[1] & x[0];
assign a_to_g[1] = ~x[3] & ~x[2] & x[0]           // f
              | ~x[3] & ~x[2] & x[1]
              | ~x[3] & x[1] & x[0]
              | x[3] & x[2] & ~x[1] & x[0];
assign a_to_g[0] = ~x[3] & ~x[2] & ~x[1]          // g
              | x[3] & x[2] & ~x[1] & ~x[0]
              | ~x[3] & x[2] & x[1] & x[0];
endmodule
```


File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Name Value Stimulator
50 100 150 200 250 300 ns

Name	Value	Stimulator																	
x	E	Binary Counter	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0
a_to_g	30		01	4F	12	06	4C	24	20	0F	00	04	08	60	31	42	30	38	01

Figure 9.5 Simulation of the Verilog program in Listing 9.1

9.3 7-Segment Decoder: *case* Statement

We can use a Verilog *case* statement to design the same 7-segment decoder that we designed in Section 9.2 using Karnaugh maps. The Verilog program shown in Listing 9.2 is a hex-to-seven-segment decoder that converts a 4-bit input hex digit, *0 – F*, to the appropriate 7-segment codes, *a – g*. The *case* statement in Listing 9.2 directly implements the truth table in Fig. 9.2. Recall that a typical line in the *case* statement, such as

```
3: a_to_g = 7'b0000110;
```

will assign the 7-bit binary value, 0000110, to the 7-bit array, `a_to_g`, when the input hex value *x*[3:0] is equal to 3 (0011). In the array `a_to_g` the value `a_to_g`[6] corresponds to segment *a* and the value `a_to_g`[0] corresponds to segment *g*. Recall that in Verilog a string of binary bits is preceded by *n*'b, where *n* is the number of binary bits in the string.

In the *case* statement the value preceding the colon in each line represents the value of the *case* parameter, in this case the 4-bit input *x*. Also note that hex values such as *A* are written as '*hA*.

Recall that all case statements should include a *default* line as shown in Listing 9.2. This is because all cases need to be covered and while it looks as if we covered all cases in Listing 6.2, Verilog actually defines *four* possible values for each bit, namely 0 (logic value 0), 1 (logic value 1), *Z* (high impedance), and *X* (unkown value).

A simulation of Listing 9.2 will produce the same results as shown in Fig. 9.5. It should be clear from this example that using the Verilog *case* statement is often easier than solving for the logic equations using Karnaugh maps.

To test the 7-segment displays on the FPGA board you could create the design *hex7seg_top.bde* shown in Fig. 9.6. This design uses the Verilog program *hex7seg.v* from Listing 9.2 and produces a top-level Verilog program *hex7seg_top.v* equivalent to Listing 9.3. Each of the four digits on the 7-segment display is enabled by one of the active low signals *an*[3:0] and all digits share the same *a_to_g*[6:0] signals. If *an*[3:0] = 0000 then all digits are enabled and display the same hex digit. This is what we do in Fig. 9.6 and Listing 9.3. Making the output *dp* = 1 will cause the decimal points to be off. You should be able to display all of the hex digits from 0 – F by changing the four rightmost switches.

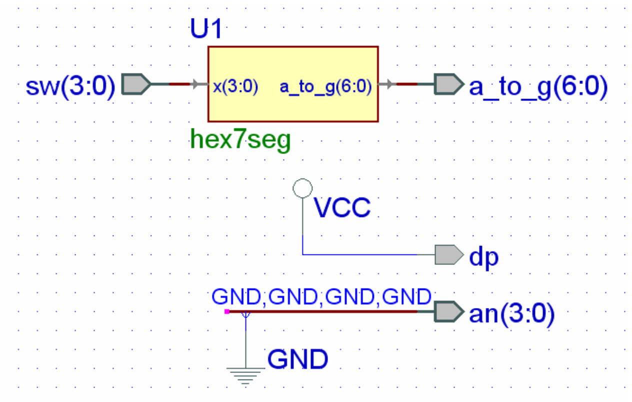


Figure 9.6 Top-level design for testing *hex7seg*

Listing 9.2 hex7seg.v

```
// Example 9b: Hex to 7-segment decoder; a-g active low
module hex7seg (
input wire [3:0] x ,
output reg [6:0] a_to_g
);

always @(*)
   case(x)
        0: a_to_g = 7'b0000001;
        1: a_to_g = 7'b1001111;
        2: a_to_g = 7'b0010010;
        3: a_to_g = 7'b0000110;
        4: a_to_g = 7'b1001100;
        5: a_to_g = 7'b0100100;
        6: a_to_g = 7'b0100000;
        7: a_to_g = 7'b0001111;
        8: a_to_g = 7'b0000000;
        9: a_to_g = 7'b0000100;
        'hA: a_to_g = 7'b0001000;
        'hb: a_to_g = 7'b1100000;
        'hC: a_to_g = 7'b0110001;
        'hd: a_to_g = 7'b1000010;
        'hE: a_to_g = 7'b0110000;
        'hF: a_to_g = 7'b0111000;
        default: a_to_g = 7'b0000001;  // 0
   endcase
endmodule
```

Listing 9.3 hex7seg_top.v

```
// Example 9c: hex7seg_top
module hex7seg_top (
input wire [3:0] sw ,
output wire [6:0] a_to_g ,
output wire [3:0] an ,
output wire dp
);

assign an = 4'b0000;           // all digits on
assign dp = 1;                 // dp off

hex7seg D4 (.x(sw),
      .a_to_g(a_to_g)
);

endmodule
```

Example 10

7-Segment Displays: *x7seg* and *x7segb*

In this example we will show how to display different hex values on the four 7-segment displays.

Prerequisite knowledge:

Karnaugh maps – Appendix D
case statement – Example 7
LEDs – Example 1

10.1 Multiplexing 7-Segment Displays

We saw in Example 9 that the *a_to_g*[6:0] signals go to all of the 7-segment displays and therefore in that example all of the digits displayed the same value. How could we display a 4-digit number such as 1234 that contains different digits? To see how we might do this, consider the BDE circuit shown in Fig. 10.2. Instead of enabling all four digits at once by setting *an*[3:0] = 0000 as we did in Fig. 9.6 we connect *an*[3:0] to the NOT of the four pushbuttons *btn*[3:0]. Thus, a digit will only be enabled when the corresponding pushbutton is being pressed.

The quad 4-to-1 multiplexer, *mux44*, from Listing 7.4 is used to display the 16-bit number *x*[15:0] as a 4-digit hex value on the 7-segment displays. When you press *btn*[0] if the control signal *s*[1:0] is 00 then *x*[3:0] becomes the input to the *hex7seg* module and the value of *x*[3:0] will be displayed on digit 0. Similarly if you press *btn*[1] and the control signal *s*[1:0] is 01 then *x*[7:4] becomes the input to the *hex7seg* module and the value of *x*[7:4] will be displayed on digit 1. We can make the value of *s*[1:0] depend on the value of *btn*[3:0] using the truth table in Fig. 10.1. From this truth table we can write the following logic equations for *s*[1] and *s*[0].

```
s[1] = btn[2] | btn[3];
s[0] = btn[1] | btn[3];
```

The two OR gates in Fig. 10.2 will implement these logic equations for s[1:0].

btn[3]	btn[2]	btn[1]	btn[0]	s[1]	s[0]
0	0	0	0	X	X
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Figure 10.1 Truth table for generating s[1:0] in Fig. 10.2

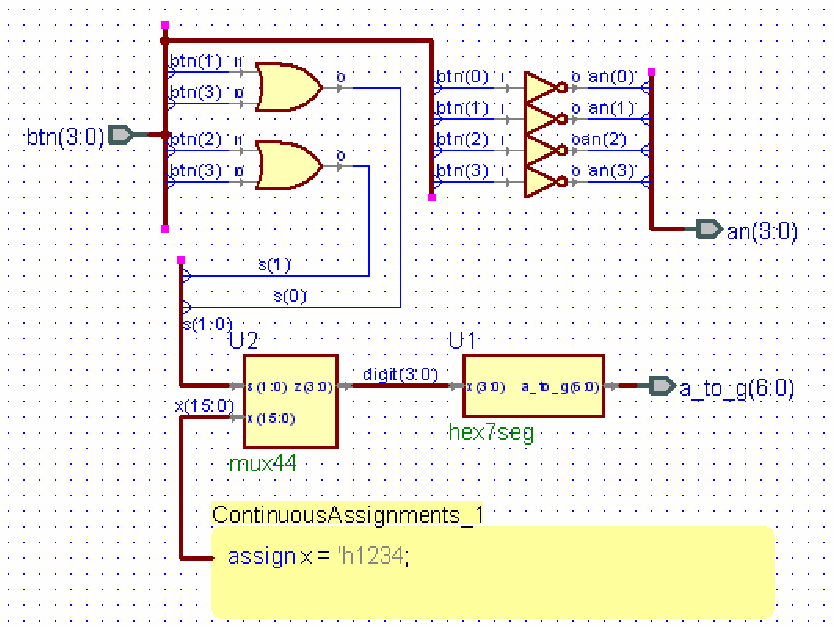


Figure 10.2 BDE circuit *mux7seg.bde* for multiplexing the four 7-segment displays

The Verilog program created by compiling *mux7seg.bde* in Fig. 10.1 is equivalent to the Verilog program shown in Listing 10.1. If you implement the design *mux7seg.bde* shown in Fig. 10.2 and download the *.bit* file to the FPGA board, then when you press buttons 0, 1, 2, and 3 the digits 4, 3, 2, and 1 will be displayed on digits 0, 1, 2, and 3 respectively. Try it.

Listing 10.1 mux7seg.v

```
// Example 10a: mux7seg
module mux7seg (
input wire [3:0] btn,
output wire [6:0] a_to_g,
output wire [3:0] an
);

wire [3:0] digit;
wire [1:0] s;
wire [15:0] x;

assign x = 'h1234;

hex7seg U1
(      .a_to_g(a_to_g), .x(digit));

mux44 U2
(      .s(s), .x(x), .z(digit));

assign s[0] = btn[3] | btn[1];
assign s[1] = btn[3] | btn[2];
assign an = ~btn;

endmodule
```

10.2 7-Segment Displays: *x7seg*

We saw in Section 10.1 that to display a 16-bit hex value on the four 7-segment displays we must multiplex the four hex digits. You can only make it appear that all four digits are on by multiplexing them fast enough (greater than 30 times per second) so that your eyes retain the values. This is the same way that your TV works where only a single picture element (pixel) is on at any one time, but the entire screen is refreshed 30 times per second so that you perceive the entire image. To do this the value of *s*[1:0] in Fig. 10.2 must count from 0 to 3 continually at this fast rate. At the same time the value of the outputs *an*[3:0] must be synchronized with *s*[1:0] so as to enable the proper digit at the proper time. A circuit for doing this is shown in Fig. 10.3. The outputs *an*[3:0] will satisfy the truth table in Fig. 10.4. Note that each output *an*[*i*] is just the maxterm M[*i*] of *q*[1:0].

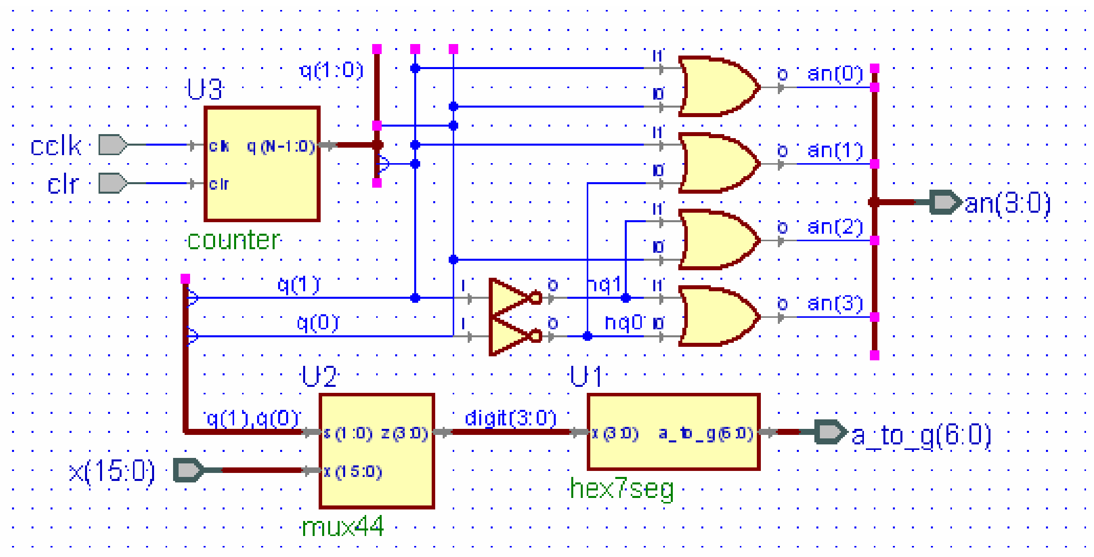


Figure 10.3 BDE circuit *x7seg.bde* for displaying x[15:0] on the four 7-segment displays

q[1]	q[0]	an[3]	an[2]	an[1]	an[0]
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	1

Figure 10.4 Truth table for generating *an*[3:0] in Fig. 10.3

A simulation of *x7seg.bde* is shown in Fig. 10.5. Note how the *an*[3:0] output selects one digit at a time to display the value 1234 on the 7-segment displays. When *x7seg.bde* is compiled it creates a Verilog program that is equivalent to Listing 10.2. The top-level design shown in Fig. 10.6 can be used to test the *x7seg* module on the FPGA board. The Verilog program corresponding to this top-level design is given in Listing 10.3. Note that the *x7seg* module requires a 190 Hz clock generated by the clock divider module *clkdiv* from Example 8.

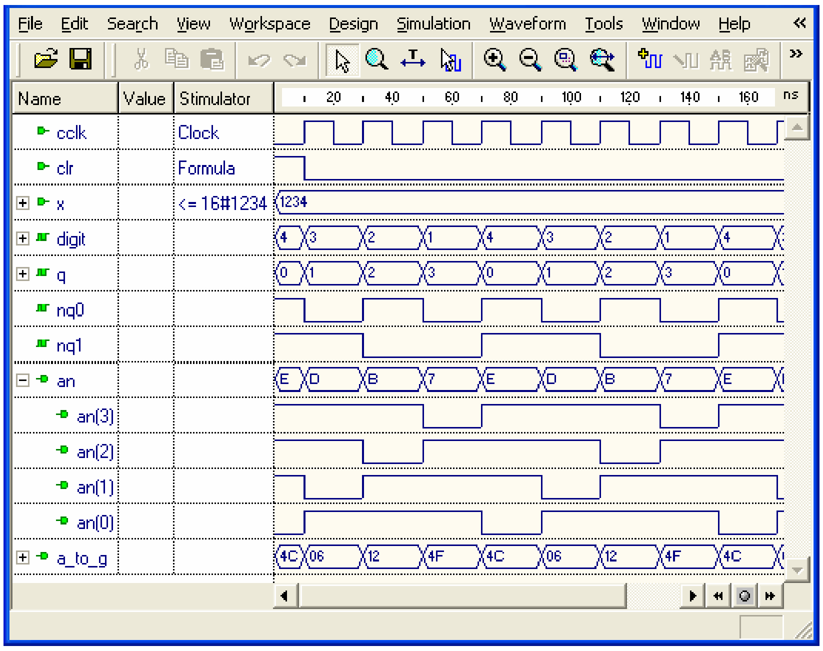


Figure 10.5 Simulation of the *x7segb.bde* circuit in Fig. 10.3

Listing 10.2 x7seg.v

```
// Example 10b: x7seg
module x7seg (
input wire cclk,
input wire clr,
input wire [15:0] x,
output wire [6:0] a_to_g,
output wire [3:0] an
);
wire nq0;
wire nq1;
wire [3:0] digit;
wire [1:0] q;
assign nq1 = ~(q[1]);
assign nq0 = ~(q[0]);
assign an[0] = q[0] | q[1];
assign an[1] = nq0 | q[1];
assign an[2] = q[0] | nq1;
assign an[3] = nq0 | nq1;

hex7seg U1
(      .a_to_g(a_to_g),.x(digit));

mux44 U2
(      .s({q[1:0]}),.x(x),.z(digit));

counter
#(     .N(2)) U3
(      .clk(cclk),.clr(clr),.q(q[1:0]));

endmodule
```

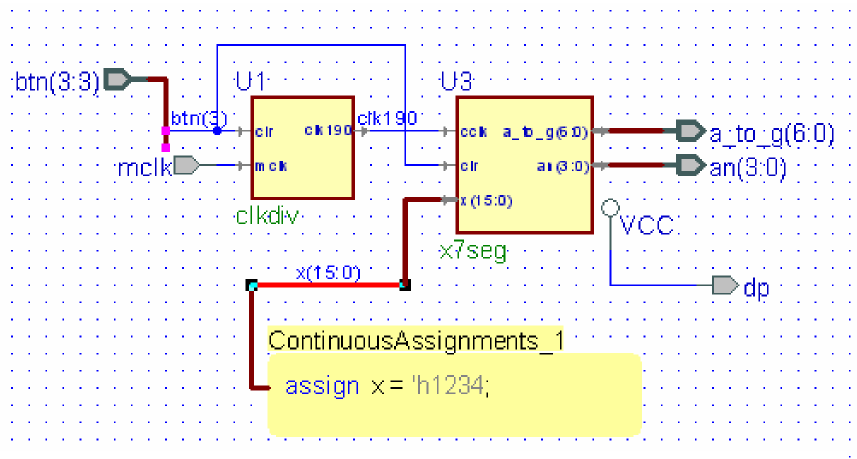


Figure 10.6 Top-level design for testing *x7seg*

Listing 10.3 x7seg_top.v

```
// Example 10c: x7seg_top
module x7seg_top (
input wire mclk,
input wire [3:3] btn,
output wire dp,
output wire [6:0] a_to_g,
output wire [3:0] an
);

wire clk190;
wire [15:0] x;

assign  x = 'h1234;
assign dp = 1;

clkdiv U1
(      .clk190(clk190),
       .clr(btn[3]),
       .mclk(mclk)
);

x7seg U3
(      .a_to_g(a_to_g),
       .an(an),
       .cclk(clk190),
       .clr(btn[3]),
       .x(x)
);

endmodule
```

10.3 7-Segment Displays: *x7segb*

When implementing the circuit for *x7seg* in Fig. 10.3 we must add separate Verilog files to the project for the modules *counter*, *hex7seg* and *mux44*. Alternatively, we can include separate *always* blocks within a single Verilog file. A variation of *x7seg*, called *x7segb*, that displays leading zeros as blanks is shown in Listing 10.4. This is done by writing logic equations for *aen*[3:0] that depend on the values of *x*[15:0]. For example, *aen*[3] will be 1 (and thus digit 3 will not be blank) if any one of the top four bits of *x*[15:0] is 1. Similarly, *aen*[2] will be 1 if any one of the top eight bits of *x*[15:0] is 1, and *aen*[1] will be 1 if any one of the top twelve bits of *x*[15:0] is 1. Note that *aen*[0] is always 1 so that digit 1 will always be displayed even if it is zero.

To test the module *x7segb* you can run the top-level design shown in Listing 10.4 that will display the value of *x* on the 7-segment displays where *x* is defined by the following statement:

```
assign x = {sw,btn[2:0],5'b01010};  // digit 0 = A
```

The curly brackets {--,--} are used for concatenation in Verilog. In this case we form the 16-bit value of *x* by concatenating the eight switches, the three right-most pushbuttons, and the five bits 01010. Note that if all switches are off an A will be displayed on digit 0 with three leading blanks. Turning on the switches and pushing the three right-most pushbuttons will display various hex numbers – always with leading blanks.

Listing 10.4 x7segb.v

```
// Example 10d: x7segb - Display 7-seg with leading blanks
// input cclk should be 190 Hz
module x7segbc (
input wire [15:0] x ,
input wire cclk ,
input wire clr ,
output reg [6:0] a_to_g ,
output reg [3:0] an ,
output wire dp
);

reg [1:0] s;
reg [3:0] digit;
wire [3:0] aen;

assign dp = 1;
// set aen[3:0] for leading blanks
assign aen[3] = x[15] | x[14] | x[13] | x[12];
assign aen[2] = x[15] | x[14] | x[13] | x[12]
                  | x[11] | x[10] | x[9] | x[8];
assign aen[1] = x[15] | x[14] | x[13] | x[12]
                  | x[11] | x[10] | x[9] | x[8]
                  | x[7] | x[6] | x[5] | x[4];
assign aen[0] = 1;      // digit 0 always on

// Quad 4-to-1 MUX: mux44
always @(*)
      case(s)
            0: digit = x[3:0];
            1: digit = x[7:4];
            2: digit = x[11:8];
            3: digit = x[15:12];
            default: digit = x[3:0];
      endcase
```

Listing 10.4 (cont.) x7segb.v

```
// 7-segment decoder: hex7seg
always @(*)
   case(digit)
            0: a_to_g = 7'b0000001;
            1: a_to_g = 7'b1001111;
            2: a_to_g = 7'b0010010;
            3: a_to_g = 7'b0000110;
            4: a_to_g = 7'b1001100;
            5: a_to_g = 7'b0100100;
            6: a_to_g = 7'b0100000;
            7: a_to_g = 7'b0001111;
            8: a_to_g = 7'b0000000;
            9: a_to_g = 7'b0000100;
            'hA: a_to_g = 7'b0001000;
            'hb: a_to_g = 7'b1100000;
            'hC: a_to_g = 7'b0110001;
            'hd: a_to_g = 7'b1000010;
            'hE: a_to_g = 7'b0110000;
            'hF: a_to_g = 7'b0111000;
            default: a_to_g = 7'b0000001;  // 0
   endcase

// Digit select
always @(*)
      begin
            an = 4'b1111;
            if(aen[s] == 1)
                  an[s] = 0;
      end

// 2-bit counter
always @(posedge cclk or posedge clr)
      begin
            if(clr == 1)
                  s <= 0;
            else
                  s <= s + 1;
      end

endmodule
```

Listing 10.5 x7segb_top.v

```
// Example 10e: x7seg_top
module x7segb_top (
input wire clk ,
input wire [3:0] btn ,
input wire [7:0] sw ,
output wire [6:0] a_to_g ,
output wire [3:0] an ,
output wire dp
);

wire [15:0] x;

// concatenate switches and 3 buttons
assign x = {sw,btn[2:0],5'b01010};  // digit 0 = A

x7segb X2 (.x(x),
      .clk(clk),
      .clr(btn[3]),
      .a_to_g(a_to_g),
      .an(an),
      .dp(dp)
);

endmodule
```

Example 11

2's Complement 4-Bit Saturator

In this example we will design a circuit that converts a 6-bit signed number to a 4-bit output that gets saturated at -8 and +7.

Prerequisite knowledge:
Basic Gates – Appendix C
Equality Detector – Example 6
Quad 2-to-1 Multiplexer – Example 6
7-Segment Displays – Example 10

11.1 Creating the Design *sat4bit.bde*

Figure 11.1 shows a circuit called *sat4bit.bde* that was described in the November 2001 issue of NASA Tech Briefs. The circuit will take a 6-bit two's complement number with a signed value between –32 and +31 and convert it to a 4-bit two's complement number with a signed value between –8 and +7. Negative input values less than –8 will be saturated at –8. Positive input values greater than +7 will be saturated at +7.

Note that the two XNOR gates and the AND gate form an equality detector whose output *s* is 1 when *x*[3], *x*[4], and *x*[5] are all equal (see Example 4). This will be the case when the 6-bit input number *x*[5:0] is between -8 and +7. In this case output *y*[3:0] of the quad 2-to-1 MUX will be connected to the input *x*[3:0]. If the top three bits of *x*[5:0] are not equal and *x*[5] is 1 then the input value will be less than -8 and the output *y*[3:0] of the quad 2-to-1 MUX will be saturated at -8. On the other hand if the top three bits of *x*[5:0] are not equal and *x*[5] is 0 then the input value will be greater than +7 and the output *y*[3:0] of the quad 2-to-1 MUX will be saturated at +7.

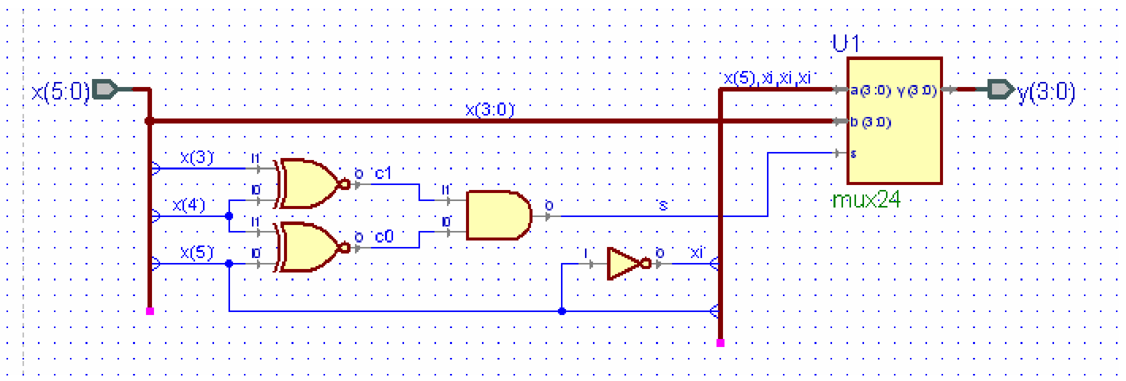


Figure 11.1 Circuit diagram for *sat4bit.bde*

Listing 11.1 sat4bit.v

```
// Example 11a: sat4bit
module sat4bit (
input wire [5:0] x,
output wire [3:0] y
);

wire c0;
wire c1;
wire s;
wire xi;

assign c1 = ~(x[4] ^ x[3]);
assign xi = ~(x[5]);
assign c0 = ~(x[5] ^ x[4]);
assign s = c0 & c1;

mux24 U1
(     .a({x[5],xi,xi,xi}),
      .b(x[3:0]),
      .s(s),
      .y(y)
);

endmodule
```

A top-level design that can be used to test *sat4bit* is shown in Fig. 11.2. The module *x7segb11* is a modification of Listing 10.4 that will display only values between -8 and +7 on the 7-segment display. Listing 11.2 shows the Verilog program for the module *x7segb11*. The input to *x7segb11* is the 4-bit output *y*[3:0] from *sat4bit*. Note that only the two rightmost 7-segment display are enabled. The two leftmost displays are always blank. The *hex7seg always* block in Listing 11.2 has been modified to display the magnitude of the signed value of y[3:0] – 0 to 8. The preceding 7-segment display will either be blank or display a minus sign. The quad 4-to-1 MUX and the new 2-to-1 MUX are used to display the minus sign when *aen*[1] is enabled if *y*[3] is 1; i.e., if *y* is negative.

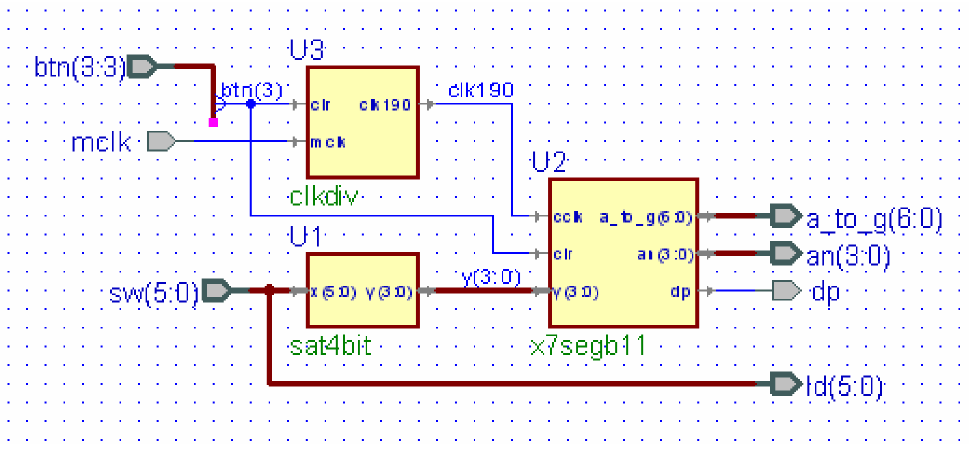


Figure 11.2 Top-level design *sat4bit_top.bde* for testing *sat4bit*

Listing 11.2 x7segb11.v

```
// Example 11b: x7segb11 - test sat4bit
module x7segb11 (
input wire [3:0] y ,
input wire cclk ,
input wire clr ,
output reg [6:0] a_to_g ,
output reg [3:0] an ,
output wire dp
);

reg msel;
reg [6:0] a_g0;
wire [6:0] a_g1;
reg [1:0] s;
reg [3:0] digit;
wire [3:0] aen;

assign a_g1 = 7'b1111110;  // minus sign
assign dp = 1;
assign aen[3] = 0;      // digit 3 always off
assign aen[2] = 0;      // digit 2 always off
assign aen[1] = y[3];   // digit 1 on if negative
assign aen[0] = 1;      // digit 0 always on

// Quad 4-to-1 MUX: mux44
always @(*)
      case(s)
            0: msel = 0;
            1: msel = 1;  // display minus sign
            2: msel = 0;
            3: msel = 0;
            default: msel = 0;
      endcase

// 7-segment decoder: hex7seg
always @(*)
   case(y)
            0: a_g0 = 7'b0000001;
            1: a_g0 = 7'b1001111;
            2: a_g0 = 7'b0010010;
            3: a_g0 = 7'b0000110;
            4: a_g0 = 7'b1001100;
            5: a_g0 = 7'b0100100;
            6: a_g0 = 7'b0100000;
            7: a_g0 = 7'b0001111;
            8: a_g0 = 7'b0000000;    // -8
            9: a_g0 = 7'b0001111;    // -7
            'hA: a_g0 = 7'b0100000;  // -6
            'hb: a_g0 = 7'b0100100;  // -5
            'hC: a_g0 = 7'b1001100;  // -4
            'hd: a_g0 = 7'b0000110;  // -3
            'hE: a_g0 = 7'b0010010;  // -2
            'hF: a_g0 = 7'b1001111;  // -1
            default: a_g0 = 7'b0000001;  // 0
   endcase
```

Listing 11.2 (cont.) x7segb11.v

```
// 2-to-1 MUX
always @(*)
      begin
            if(msel == 1)
                  a_to_g = a_g1;
            else
                  a_to_g = a_g0;
      end

// Digit select
always @(*)
      begin
            an = 4'b1111;
            if(aen[s] == 1)
                  an[s] = 0;
      end

// 2-bit counter
always @(posedge cclk or posedge clr)
      begin
            if(clr == 1)
                  s <= 0;
            else
                  s <= s + 1;
      end

endmodule
```

The Verilog program corresponding to the top-level design in Fig. 11.2 is given in Listing 11.3. Download this top-level design to the FPGA board and observe the output on the 7-segment display for different 6-bit switch inputs.

Listing 11.3 sat4bit_top.v

```
// Example 11c: sat4bit_top
module sat4bit_top (
input wire mclk,
input wire [3:3] btn,
input wire [5:0] sw,
output wire dp,
output wire [6:0] a_to_g,
output wire [3:0] an,
output wire [5:0] ld
);

wire clk190;
wire [3:0] y;

assign ld = sw;

sat4bit U1
(      .x(sw),
       .y(y)
);

x7segb11 U2
(      .a_to_g(a_to_g),
       .an(an),
       .cclk(clk190),
       .clr(btn[3]),
       .dp(dp),
       .y(y)
);

clkdiv U3
(      .clk190(clk190),
       .clr(btn[3]),
       .mclk(mclk)
);

endmodule
```

Example 12

Full Adder

In this example we will design a full adder circuit.

Prerequisite knowledge:
Basic Gates – Appendix C
Karnaugh Maps – Appendix D
7-Segment Displays – Example 10

12.1 Half Adder

The truth table for a half adder is shown in Fig. 12.1. In this table bit *a* is added to bit *b* to produce the sum bit *s* and the carry bit *c*. Note that if you add 1 to 1 you get 2, which in binary is 10 or 0 with a carry bit. The BDE logic diagram, *halfadd.bde*, for a half adder is also shown in Fig. 12.1. Note that the sum *s* is just the exclusive-or of *a* and *b* and the carry *c* is just *a* & *b*. The Verilog program corresponding to the circuit in Fig. 12.1 is shown in Listing 12.1. A simulation of *halfadd.bde* is shown in Fig. 12.2.

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

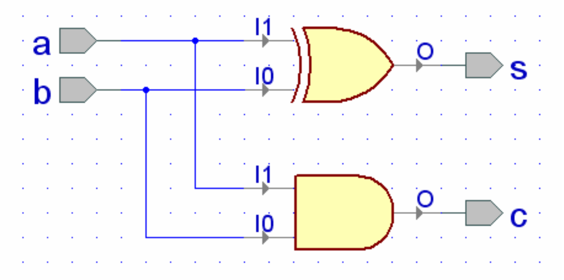


Figure 12.1 Truth table and logic diagram *halfadd.bde* for a half-adder

Listing 12.1 halfadd.v

```
// Example 12a: halfadd
module halfadd (
input wire a,
input wire b,
output wire c,
output wire s
) ;

assign s = b ^ a;
assign c = b & a;

endmodule
```

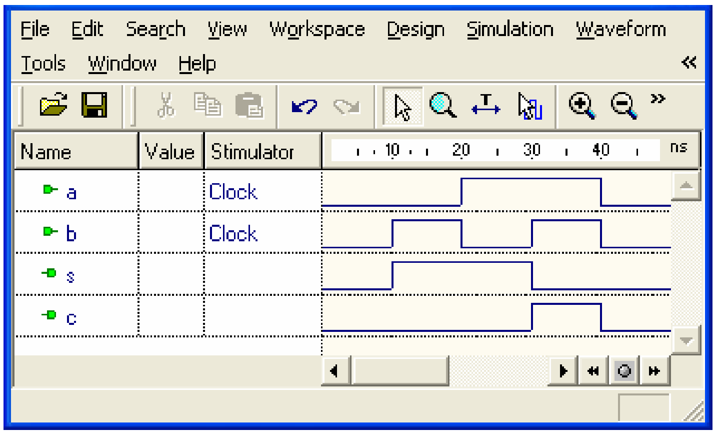


Figure 12.2 Simulation of the half-adder in Fig. 12.1

12.2 Full Adder

When adding binary numbers we need to consider the carry from one bit to the next. Thus, at any bit position we will be adding three bits: a_i, b_i and the carry-in c_i from the addition of the two bits to the right of the current bit position. The sum of these three bits will produce a sum bit, s_i, and a carry-out, c_{i+1}, which will be the carry-in to the next bit position to the left. This is called a *full adder* and its truth table is shown in Fig. 12.3. The results of the first seven rows in this truth table can be inferred from the truth table for the half adder given in Fig. 12.1. In all of these rows only two 1's are ever added together. The last row in Fig. 12.3 adds three 1's. The result is 3, which in binary is 11, or 1 plus a carry.

From the truth table in Fig. 12.3 we can write a sum of products expression for s_i as

c_i	a_i	b_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 12.3
Truth table for a full adder

```
si =  ~ci & ~ai &  bi
    | ~ci &  ai & ~bi         (12.1)
    |  ci & ~ai & ~bi
    |  ci &  ai &  bi
```

We can use the distributive law to factor out $\sim c_i$ from the first two product terms and c_i from the last two product terms in Eq. (12.1) to obtain

```
si =  ~ci & (~ai &  bi | ai & ~bi)
    |  ci & (~ai & ~bi | ai &  bi)                (12.2)
```

which can be written in terms of XOR and XNOR operations as

```
si = ~ci & (ai ^ bi) | ci & ~(ai ^ bi)            (12.3)
```

which further reduces to

```
si = ci ^ (ai ^ bi)                                    (12.4)
```

Fig. 12.4 shows the K-map for c_{i+1} from the truth table in Fig. 12.3. The map shown in Fig. 12.4a leads to the reduced form for c_{i+1} given by

```
ci+1 =  ai & bi | ci & bi | ci & ai                    (12.5)
```

While this is the reduced form, a more convenient form can be written from Fig. 12.4b as follows:

```
ci+1  = ai & bi | ci & ~ai & bi | ci & ai & ~bi
      = ai & bi | ci & (~ai & bi | ai & ~bi)
      = ai & bi | ci & (ai ^ bi)                       (12.6)
```

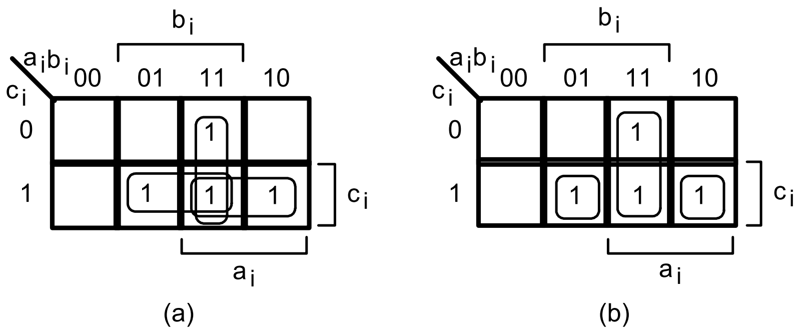


Figure 12.4 K-maps for c_{i+1} for full adder in Fig. 6.2

From Eqs. (12.4) and (12.6) we can draw the logic diagram for a full adder as shown in Fig. 12.5. Comparing this diagram to that for a half adder in Fig. 12.1 it is clear that a full adder can be made from two half adders plus an OR gate as shown in Fig. 12.6.

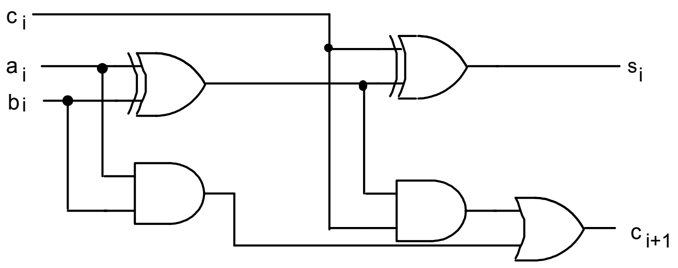


Figure 12.5 Logic diagram for a full adder

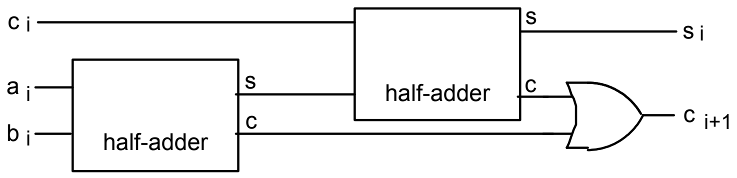


Figure 12.6 A full adder can be made from two half adders plus an OR gate

From Fig. 12.6 we can create a BDE design, *fulladd.bde*, as shown in Fig. 12.7. The Verilog program resulting from compiling this design is equivalent to that shown in Listing 12.2. A simulation of this full adder is shown in Fig. 12.8. Note that the outputs agree with the truth table in Fig. 12.3.

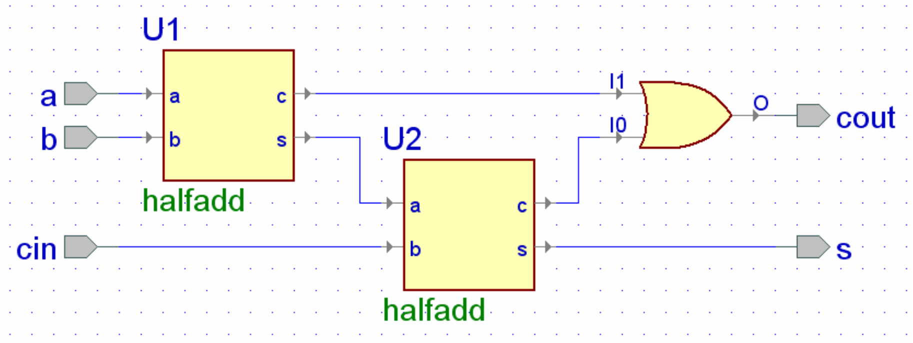


Figure 12.7 Block diagram *fulladd.bde* for a full adder

Listing 12.2 fulladd.v

```
// Example 12b: fulladd
module fulladd (
input wire a,
input wire b,
input wire cin,
output wire cout,
output wire s
) ;

wire c1;
wire c2;
wire s1;

assign cout = c2 | c1;

halfadd U1
(      .a(a),
       .b(b),
       .c(c1),
       .s(s1)
);
halfadd U2
(      .a(s1),
       .b(cin),
       .c(c2),
       .s(s)
);

endmodule
```

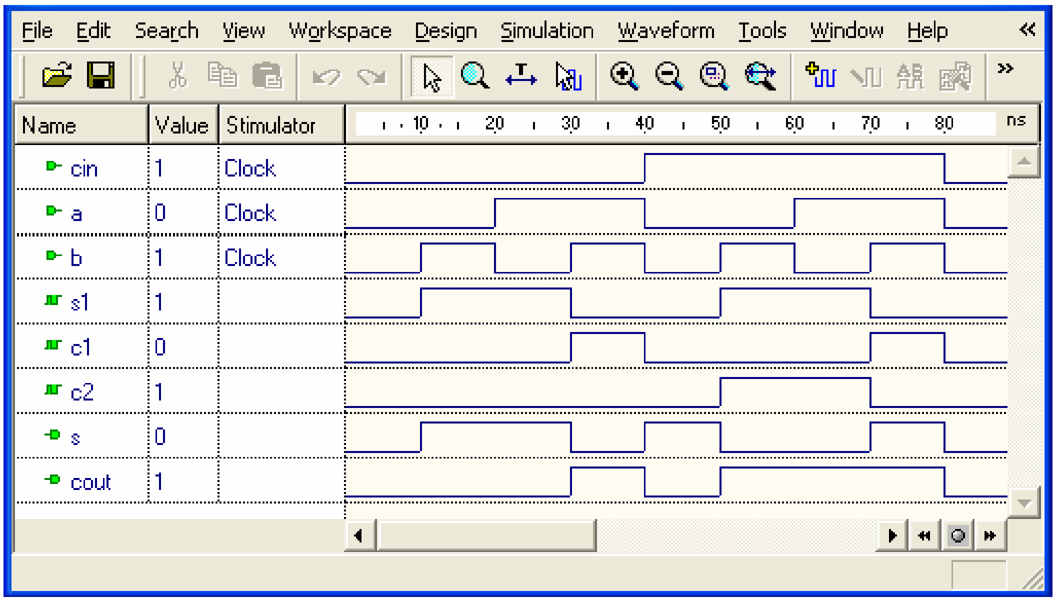

File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Name Value Stimulator
10 20 30 40 50 60 70 80 ns
cin 1 Clock
a 0 Clock
b 1 Clock
s1 1
c1 0
c2 1
s 0
cout 1

Figure 12.8 Simulation of the full adder in Fig. 12.7 and Listing 12.2

Example 13

4-Bit Adder

In this example we will design a 4-bit adder.

Prerequisite knowledge:
Basic Gates – Appendix C
Karnaugh Maps – Appendix D
Full Adder – Example 12

13.1 4-Bit Adder

Four of the full adders in Fig. 12.7 can be combined to form a 4-bit adder as shown in Fig. 13.1. Note that the full adder for the least significant bit will have a carry-in of zero while the remaining bits get their carry-in from the carry-out of the previous bit. The final carry-out, is the *cout* for the 4-bit addition. The Verilog program corresponding to the 4-bit adder in Fig. 13.1 is given in Listing 13.1.

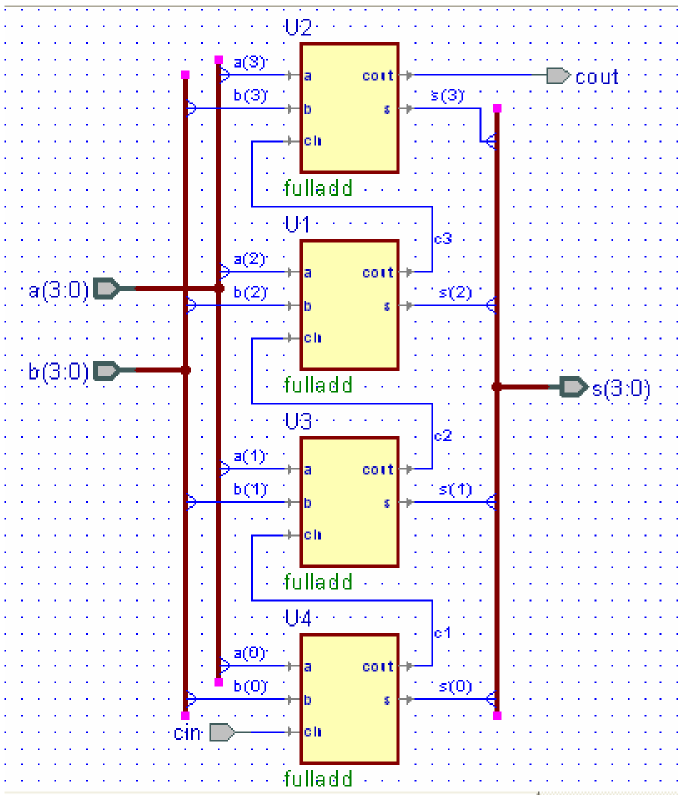


Figure 13.1 Block diagram *adder4.bde* for a 4-bit adder

Listing 13.1 adder4.v

```
// Example 13a: adder4
module adder4 (
input wire cin;
input wire [3:0] a;
input wire [3:0] b;
output wire cout;
output wire [3:0] s;
) ;

wire c1;
wire c2;
wire c3;

fulladd U1
(      .a(a[2]),
       .b(b[2]),
       .cin(c2),
       .cout(c3),
       .s(s[2])
);

fulladd U2
(      .a(a[3]),
       .b(b[3]),
       .cin(c3),
       .cout(cout),
       .s(s[3])
);

fulladd U3
(      .a(a[1]),
       .b(b[1]),
       .cin(c1),
       .cout(c2),
       .s(s[1])
);

fulladd U4
(      .a(a[0]),
       .b(b[0]),
       .cin(cin),
       .cout(c1),
       .s(s[0])
);

endmodule
```

A simulation of the 4-bit adder in Fig. 13.1 and Listing 13.1 is shown in Fig. 13.2. The value of *a* is incremented from 0 to F and is added to the hex value B. The sum *s* is always equal to *a* + *b*. Note that the carry flag, *cout*, is equal to 1 when the correct *unsigned* answer exceeds 15 (or F).

We can test the *adder4* module from Fig. 13.1 and Listing 13.1 on the FPGA board by combining it with the *x7segb* module from Listing 10.4 in Example 10 and the *clkdiv* module from Listing 8.2 from Example 8 to produce the top-level design shown in Listing 13.2. The 4-bit number *sw*[7:4] will be displayed on the first (left-most) 7-

segment display. The 4-bit number *sw*[3:0] will be displayed on the second 7-segment display. These two numbers will be added and the 4-bit sum will be displayed on the fourth (right-most) 7-segment display and the carry bit will be displayed on the third 7-segment display. Try it.

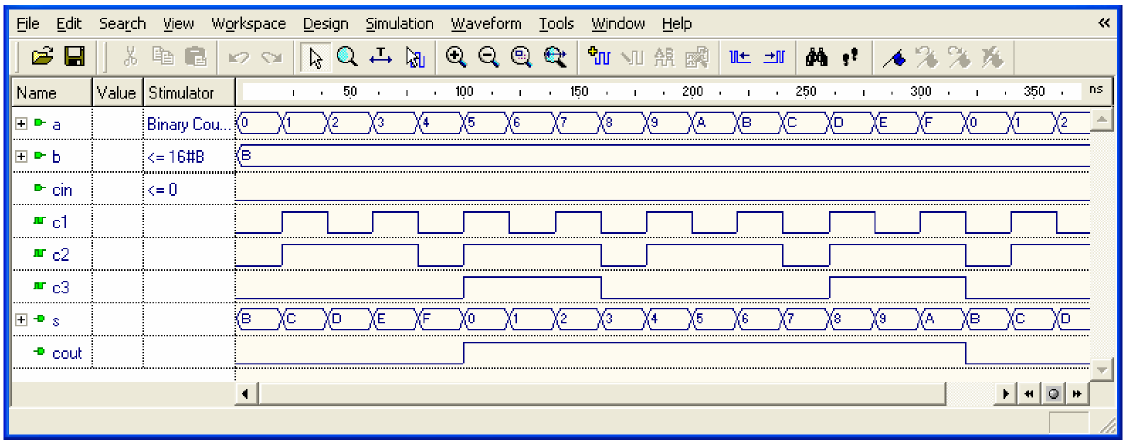

File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Name Value Stimulator
a Binary Cou...
b <= 16#B
cin <= 0
c1
c2
c3
s
cout
50 100 150 200 250 300 350 ns
0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2
B
B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D

Figure 13.2 Simulation of the 4-bit adder in Fig. 13.1 and Listing 13.1

Listing 13.2 adder4_top.v

```
// Example 13b: adder4_top
module adder4_top (
input wire mclk ,
input wire [3:3] btn ,
input wire [7:0] sw ,
output wire [6:0] a_to_g ,
output wire [3:0] an ,
output wire dp ,
output wire [7:0] ld
);

wire clk190, clr, c4, cin;
wire [15:0] x;
wire [3:0] sum;

assign cin = 0;
assign x = {sw,3'b000,c4,sum};
assign clr = btn[3];
assign ld = sw;

adder4 U1 (.cin(cin),.a(sw[7:4]),.b(sw[3:0]),
      .cout(c4),.s(sum));

clkdiv U2 (.mclk(mclk),.clr(clr),.clk190(clk190));

x7segb U3 (.x(x),.cclk(clk190),.clr(clr),
      .a_to_g(a_to_g),.an(an),.dp(dp));

endmodule
```

Example 14

N-Bit Adder

In this example we will design a *N*-bit adder.

Prerequisite knowledge:
4-Bit Adder – Example 13

14.1 4-Bit Adder: Behavioral Statements

It would be convenient to be able to make a 4-bit adder (or any size adder) by just using a + sign in a Verilog statement. In fact, we can! When you write *a* + *b* in a Verilog program the compiler will produce a full adder of the type we designed in Example 12. The only question is how to create the output carry bit. The trick is to add a leading 0 to *a* and *b* and then make a 5-bit temporary variable to hold the sum as shown in Listing 14.1. The most-significant bit of this 5-bit sum will be the carry flag.

A simulation of this program is shown in Fig. 14.1. Compare this with Fig. 13.2.

Listing 14.1 adder4b.v

```
// Example 14a:  4-bit behavioral adder
module adder4b (
input wire [3:0] a ,
input wire [3:0] b ,
output reg [3:0] s ,
output reg cf
);
reg [4:0] temp;

always @(*)
   begin
      temp = {1'b0,a} + {1'b0,b};
      s = temp[3:0];
      cf = temp[4];
   end
endmodule
```

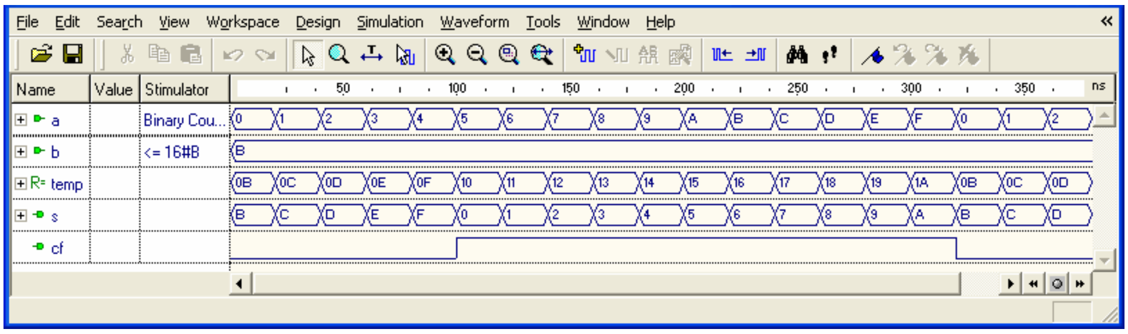

File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Name Value Stimulator
50 100 150 200 250 300 350 ns
a Binary Cou... 0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2
b <= 16#B B
temp 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 0B 0C 0D
s B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D
cf

Figure 14.1 Simulation of the Verilog program in Listing 14.1

14.2 *N* - Bit Adder: Behavioral Statements

Listing 14.2 shows an *N*-bit adder that uses a *parameter* statement. This is a convenient adder to use when you don't need the carry flag. An example of using this as an 8-bit adder is shown in the simulation in Fig. 14.2. Note that when the sum exceeds FF it simply wraps around and the carry flag is lost.

Listing 14.2 adder.v

```
// Example 14b:  N-bit adder
module adder
#(parameter N = 8)
(input wire [N-1:0] a,
 input wire [N-1:0] b,
 output reg [N-1:0] y
);

always @(*)
   begin
      y = a + b;
   end
endmodule
```

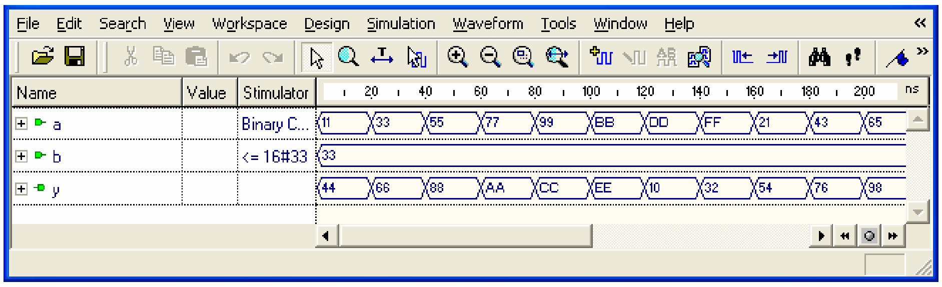

File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Name Value Stimulator
20 40 60 80 100 120 140 160 180 200 ns
a Binary C... 11 33 55 77 99 BB DD FF 21 43 65
b <= 16#33 33
y 44 66 88 AA CC EE 10 32 54 76 98

Figure 14.2 Simulation of the Verilog program in Listing 14.2

The top-level design shown in Fig. 14.3 can be used to test this *N*-bit adder on the FPGA board. In this case we are adding two 4-bit switch settings and observing the sum on the 7-segment display. To set the parameter *N* to 4 right-click on the adder symbol, select *Properties* and click on the *Parameter* tab. Set the actual value of *N* to 4.

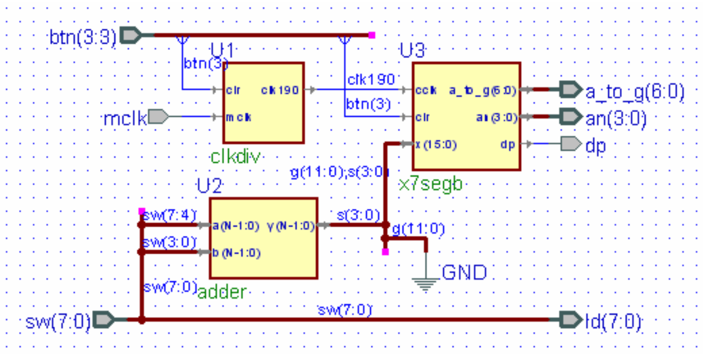


Figure 14.3 Top-level design for testing the N-bit adder on the FPGA board

Example 15

N-Bit Comparator

In this example we will design a *N*-bit comparator.

Prerequisite knowledge:
N-Bit Adder – Example 14

15.1 *N*-Bit Comparator Using Relational Operators

The easiest way to implement a comparator in Verilog is to use the relational and logical operators shown in Table 15.1. An example of using these to implement an *N*-bit comparator is shown in Listing 15.1. A simulation of this program for the default value of $N = 8$ is shown in Fig. 15.1.

Note in the *always* block in Listing 15.1 we set the values of *gt*, *eq*, and *lt* to zero before the *if* statements. This is important to make sure that each output has a value assigned to it. If you don't do this then Verilog will assume you don't want the value to change and will include a latch in your system. Your circuit will then not be a combinational circuit.

Table 15.1 Relational and Logical Operators

Operator	Meaning
==	Logical equality
!=	Logical inequality
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
!	Logical negation
&&	Logical AND
||	Logical OR

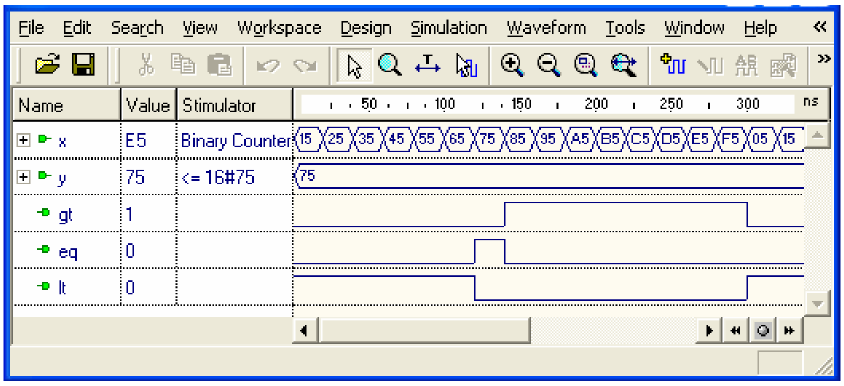

File Edit Search View Workspace Design Simulation Waveform Tools Window Help
Name Value Stimulator
50 100 150 200 250 300 ns
x E5 Binary Counter
15 25 35 45 55 65 75 85 95 A5 B5 C5 D5 E5 F5 05 15
y 75 <= 16#75
75
gt 1
eq 0
lt 0

Figure 15.1 Simulation of the Verilog program in Listing 15.1

Listing 15.1 comp.v

```
// Example 17: N-bit comparator using relational operators
module comp
#(parameter N = 8)
(input wire [N-1:0] x,
 input wire [N-1:0] y,
 output reg gt,
 output reg eq,
 output reg lt
);

always @(*)
begin
   gt = 0;
   eq = 0;
   lt = 0;
   if(x > y)
         gt = 1;
   if(x == y)
         eq = 1;
   if(x < y)
         lt = 1;
end

endmodule
```

You can test this comparator on the FPGA board by creating the BDE block diagram *comp4_top.bde* shown in Fig. 15.2. To make this a 4-bit comparator right-click on the *comp* symbol, select *Properties*, click on the *Parameters* tab, and set the actual value of *N* to 4. You will be comparing the 4-bit number *x*[3:0] on the left four switches with the 4-bit number *y*[3:0] on the right four switches. The three LEDs *ld*[4:2] will detect the outputs *gt*, *eq*, and *lt*. We selected these three LEDs because on the BASYS board they are three different colors. Compile the design *comp4_top.bde*, implement it, and download the *.bit* file to the FPGA board. Test the comparator by changing the switch settings.

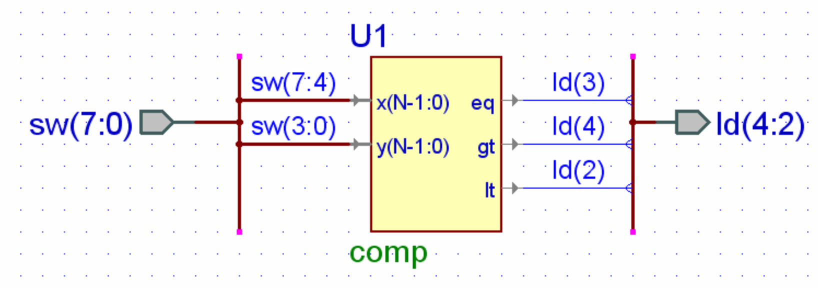


Figure 15.2 Top-level design *comp4_top.bde* to test a 4-bit comparator

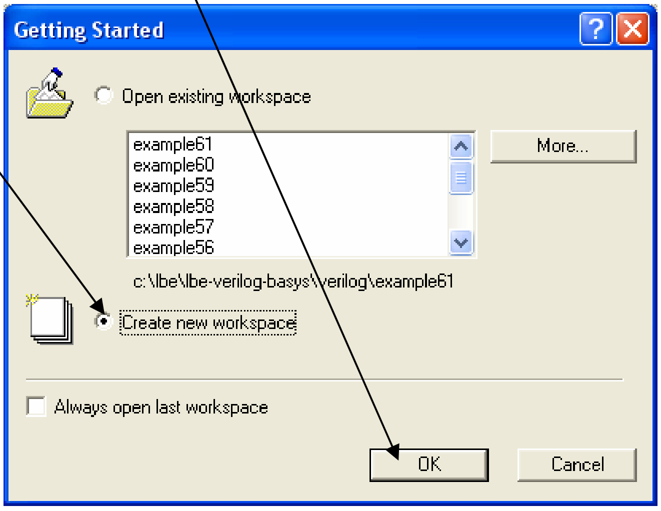
Appendix A

Aldec Active-HDL Tutorial

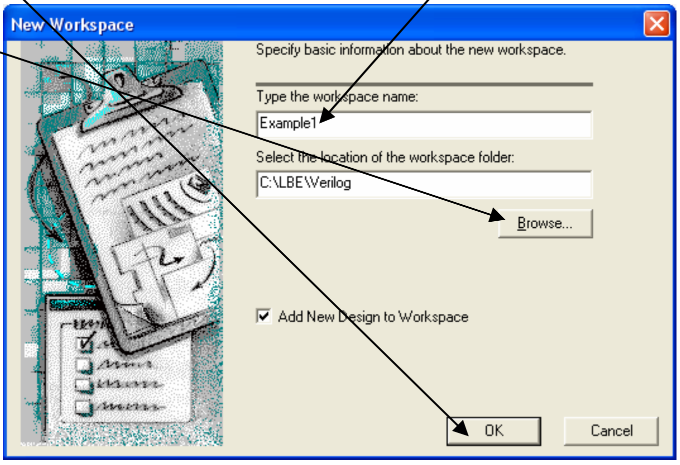
Part 1: Project Setup

Start the program by double-clicking the Active-HDL icon on the desktop.

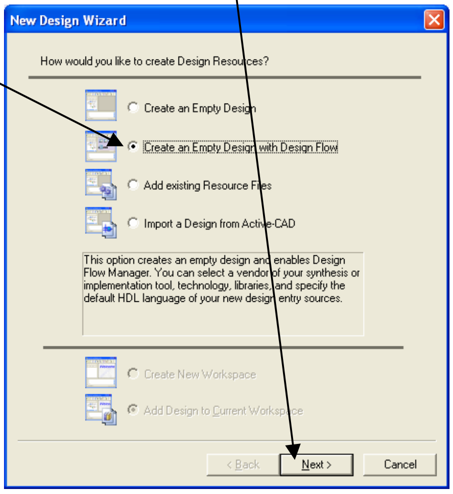
Select *Create new workspace* and click *OK*.



Browse to the directory where you want the project saved, type *Example1* for the workspace name and click *OK*.


New Workspace
Specify basic information about the new workspace.
Type the workspace name:
Example1
Select the location of the workspace folder:
C:\LBE\Verilog
Browse...
Add New Design to Workspace
OK
Cancel

Select *Create an Empty Design with Design Flow* and click *Next*.



New Design Wizard

How would you like to create Design Resources?

- ( ) Create an Empty Design
- (●) Create an Empty Design with Design Flow
- ( ) Add existing Resource Files
- ( ) Import a Design from Active-CAD

This option creates an empty design and enables Design Flow Manager. You can select a vendor of your synthesis or implementation tool, technology, libraries, and specify the default HDL language of your new design entry sources.

- ( ) Create New Workspace
- (●) Add Design to Current Workspace

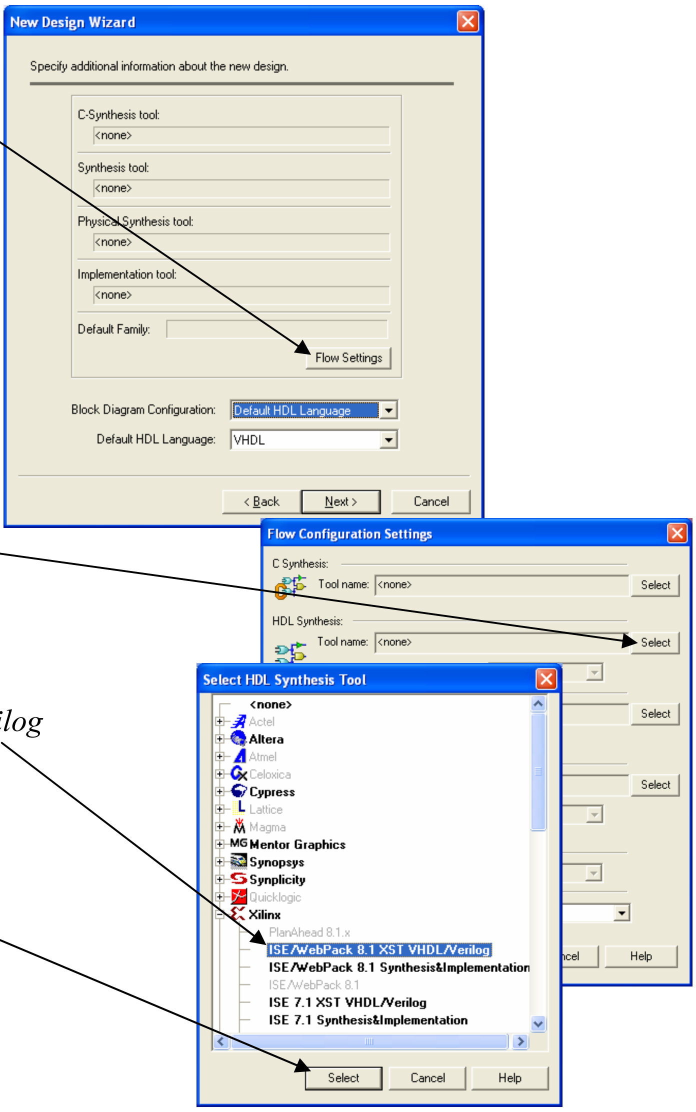
< Back | Next > | Cancel

Click *Flow Settings*

Select *HDL Synthesis*

Select *Xilinx*
ISE/WebPack 8.1 XST VHDL/Verilog

Press *Select*



New Design Wizard

Specify additional information about the new design.

C-Synthesis tool:	<none>
Synthesis tool:	<none>
Physical Synthesis tool:	<none>
Implementation tool:	<none>
Default Family:	

Flow Settings

Block Diagram Configuration: Default HDL Language

Default HDL Language: VHDL

< Back | Next > | Cancel

Flow Configuration Settings

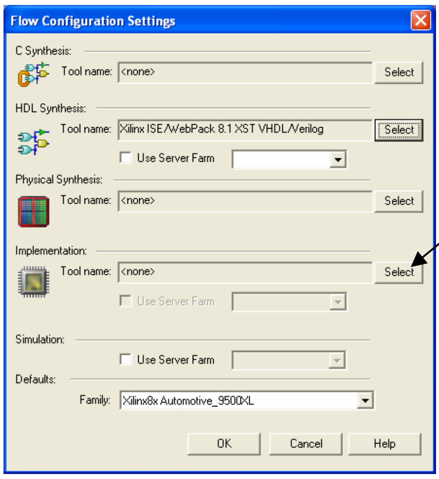
C Synthesis: Tool name: <none> Select

HDL Synthesis: Tool name: <none> Select

Select HDL Synthesis Tool

- <none>
- Actel
- Altera
- Atmel
- Celoxica
- Cypress
- Lattice
- Magma
- Mentor Graphics
- Synopsys
- Synplicity
- Quicklogic
- Xilinx
  - PlanAhead 8.1.x
  - ISE/WebPack 8.1 XST VHDL/Verilog
  - ISE/WebPack 8.1 Synthesis&Implementation
  - ISE/WebPack 8.1
  - ISE 7.1 XST VHDL/Verilog
  - ISE 7.1 Synthesis&Implementation

Select | Cancel | Help


Flow Configuration Settings

C Synthesis: Tool name: <none> Select

HDL Synthesis: Tool name: Xilinx ISE/WebPack 8.1 XST VHDL/Verilog Select
Use Server Farm

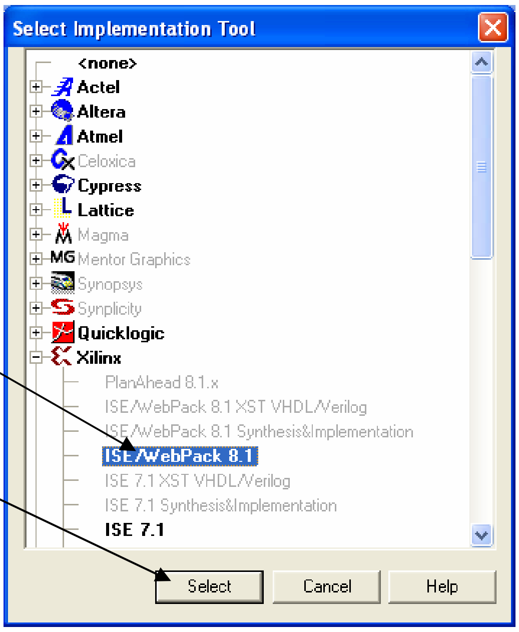
Physical Synthesis: Tool name: <none> Select

Implementation: Tool name: <none> Select
Use Server Farm

Simulation: Use Server Farm

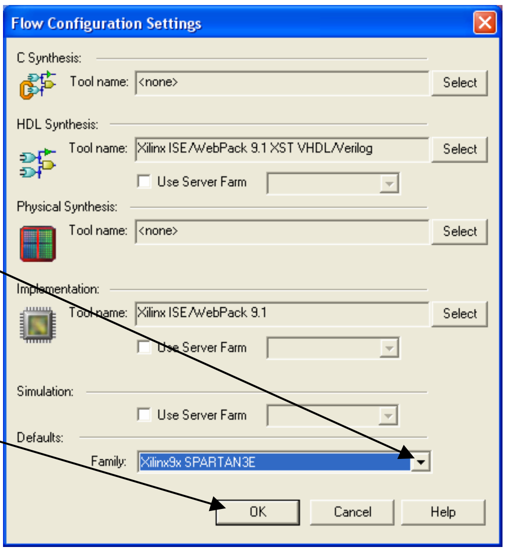
Defaults: Family: Xilinx8x Automotive_9500XL

Select *Implementation*


Select Implementation Tool

Choose *Xilinx ISE/WebPack 8.1*

Press *Select*


Flow Configuration Settings

C Synthesis: Tool name: <none> Select

HDL Synthesis: Tool name: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog Select
Use Server Farm

Physical Synthesis: Tool name: <none> Select

Implementation: Tool name: Xilinx ISE/WebPack 9.1 Select
Use Server Farm

Simulation: Use Server Farm

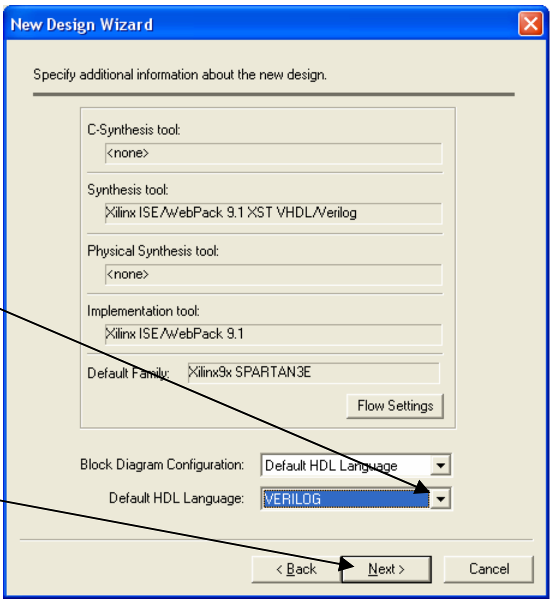
Defaults: Family: Xilinx9x SPARTAN3E

Select *Xilinx9X SPARTAN3E* for Family

Click *Ok*

Select *VERILOG* for the Default HDL Language

Click *Next*


New Design Wizard

Specify additional information about the new design.

C-Synthesis tool: <none>

Synthesis tool: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog

Physical Synthesis tool: <none>

Implementation tool: Xilinx ISE/WebPack 9.1

Default Family: Xilinx9x SPARTAN3E

Flow Settings

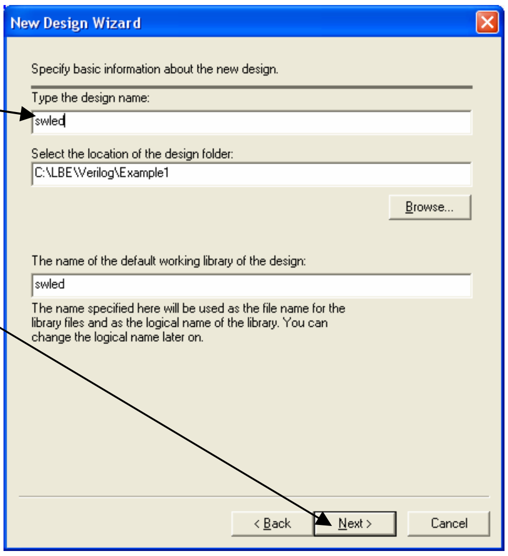
Block Diagram Configuration: Default HDL Language

Default HDL Language: VERILOG

< Back | Next > | Cancel

Type *swled* for the design name

and click *Next*.


New Design Wizard

Specify basic information about the new design.

Type the design name: swled

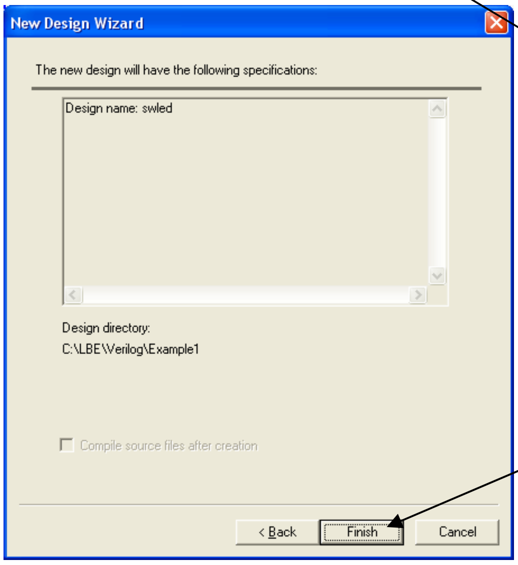
Select the location of the design folder: C:\LBE\Verilog\Example1

Browse...

The name of the default working library of the design: swled

The name specified here will be used as the file name for the library files and as the logical name of the library. You can change the logical name later on.

< Back | Next > | Cancel


New Design Wizard

The new design will have the following specifications:

Design name: swled

Design directory:
C:\LBE\Verilog\Example1

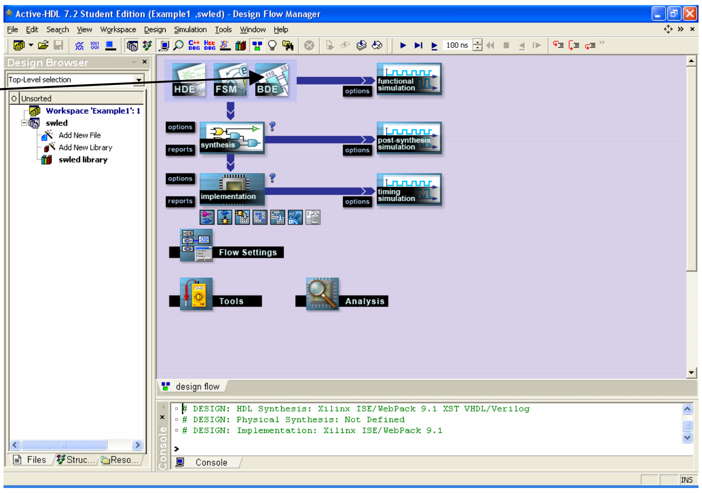
Compile source files after creation

< Back | Finish | Cancel

Click *Finish*.

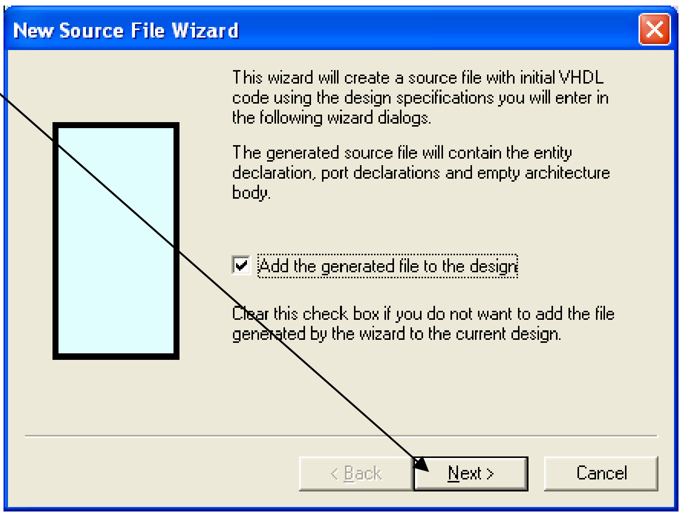
Part 2: Design Entry – sw2led.bde

Click on *BDE*.



```
# DESIGN: HDL Synthesis: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog
# DESIGN: Physical Synthesis: Not Defined
# DESIGN: Implementation: Xilinx ISE/WebPack 9.1
```

Click *Next*.



New Source File Wizard

This wizard will create a source file with initial VHDL code using the design specifications you will enter in the following wizard dialogs.

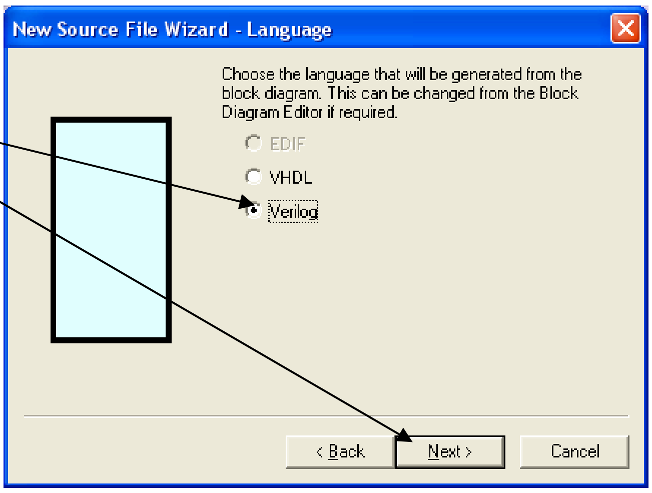
The generated source file will contain the entity declaration, port declarations and empty architecture body.

☑ Add the generated file to the design

Clear this check box if you do not want to add the file generated by the wizard to the current design.

< Back | Next > | Cancel

Select *Verilog* and Click *Next*



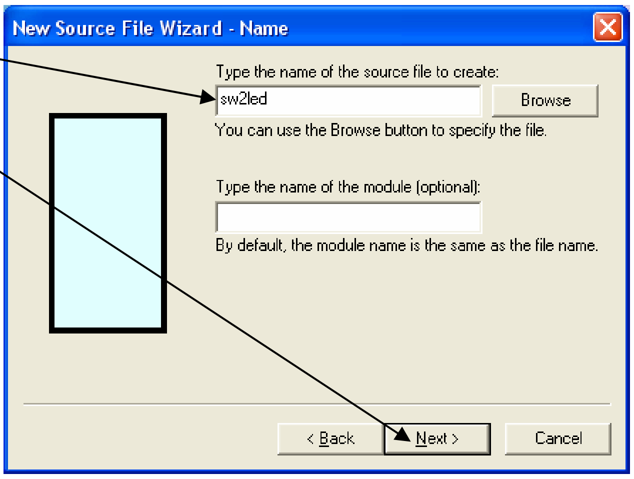
New Source File Wizard - Language

Choose the language that will be generated from the block diagram. This can be changed from the Block Diagram Editor if required.

○ EDIF
○ VHDL
● Verilog

< Back | Next > | Cancel

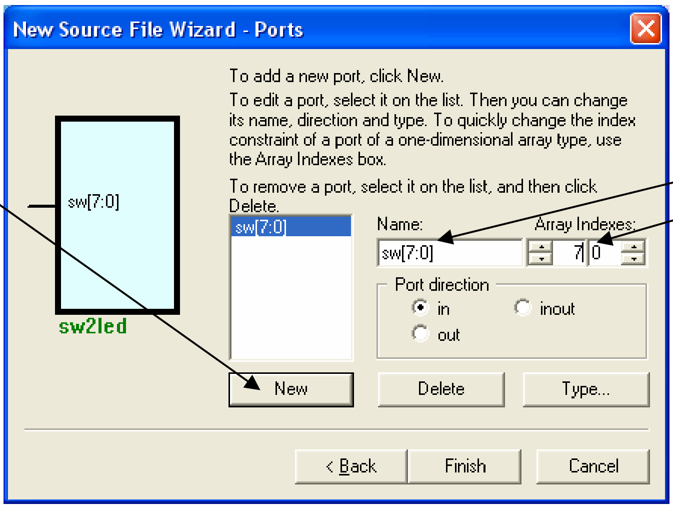
Type *sw2led* and click *Next*.


New Source File Wizard - Name

Type the name of the source file to create:
sw2led
You can use the Browse button to specify the file.
Type the name of the module (optional):
By default, the module name is the same as the file name.

Click *New*.

Type *sw*
Set array indexes to 7:0


New Source File Wizard - Ports

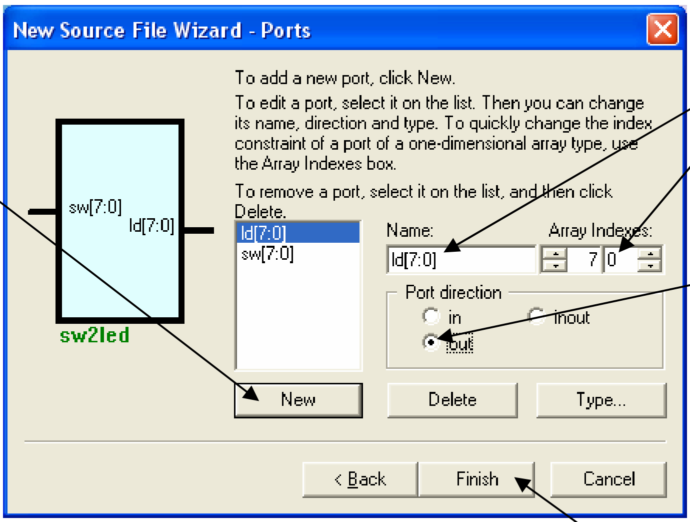
To add a new port, click New.
To edit a port, select it on the list. Then you can change its name, direction and type. To quickly change the index constraint of a port of a one-dimensional array type, use the Array Indexes box.
To remove a port, select it on the list, and then click Delete.

Name: sw[7:0]
Array Indexes: 7 0
Port direction: [✓ in] inout out

Click *New*.

Type *ld*
Set array indexes to 7:0

Click *out*.


New Source File Wizard - Ports

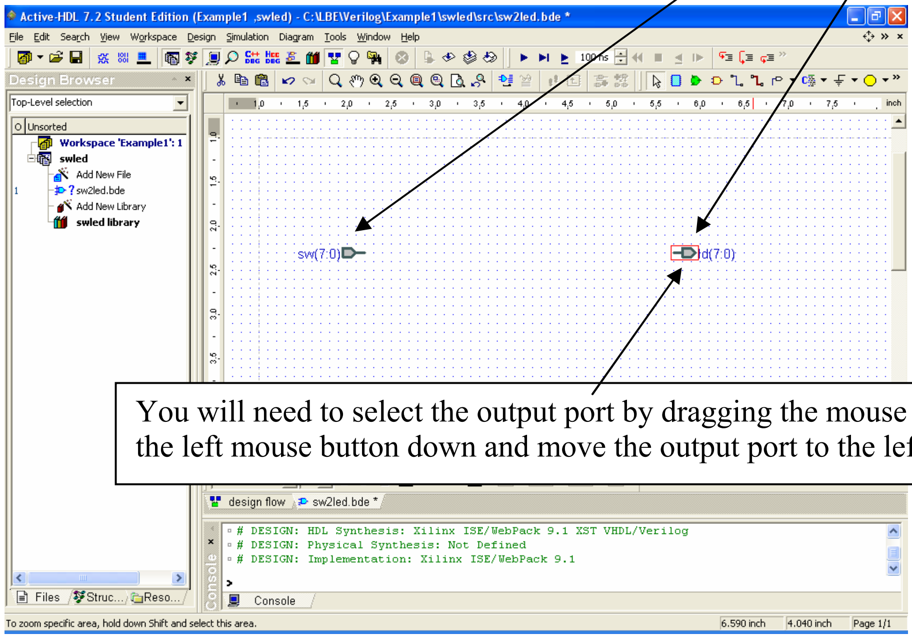
To add a new port, click New.
To edit a port, select it on the list. Then you can change its name, direction and type. To quickly change the index constraint of a port of a one-dimensional array type, use the Array Indexes box.
To remove a port, select it on the list, and then click Delete.

ld[7:0]
sw[7:0]

Name: ld[7:0]
Array Indexes: 7 0
Port direction: in inout [✓ out]

Click *Finish*.

This will generate a block diagram (schematic) template with the *input* and *output* ports displayed.

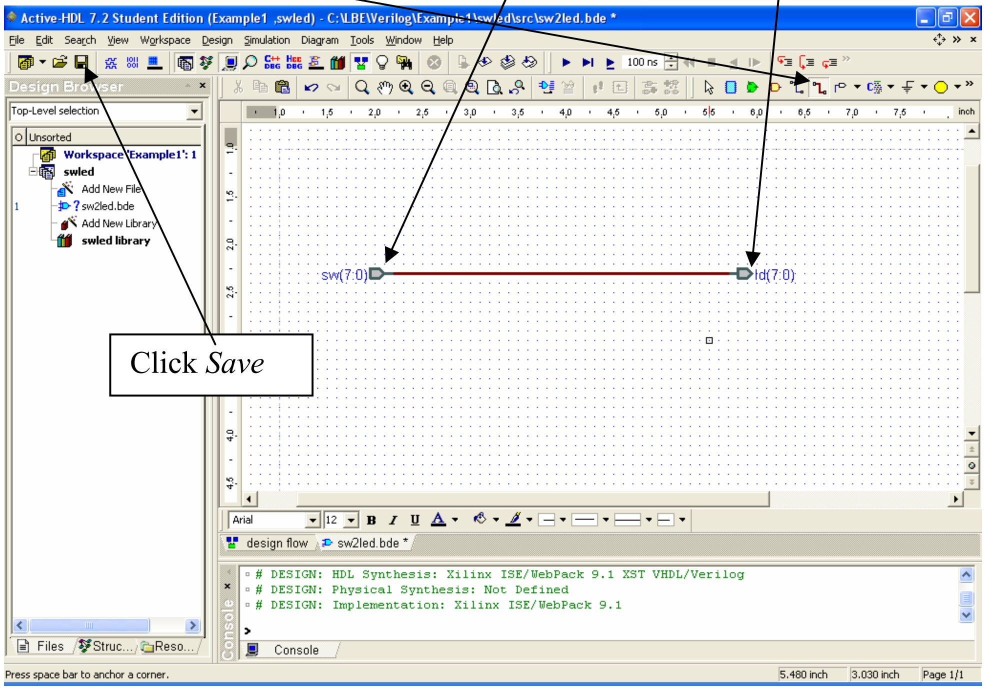


You will need to select the output port by dragging the mouse with the left mouse button down and move the output port to the left.

```
# DESIGN: HDL Synthesis: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog
# DESIGN: Physical Synthesis: Not Defined
# DESIGN: Implementation: Xilinx ISE/WebPack 9.1
```

To zoom specific area, hold down Shift and select this area. 6.590 inch 4.040 inch Page 1/1

Select the *bus* icon and connect the input *sw*[7:0] to the output *ld*[7:0] as shown.

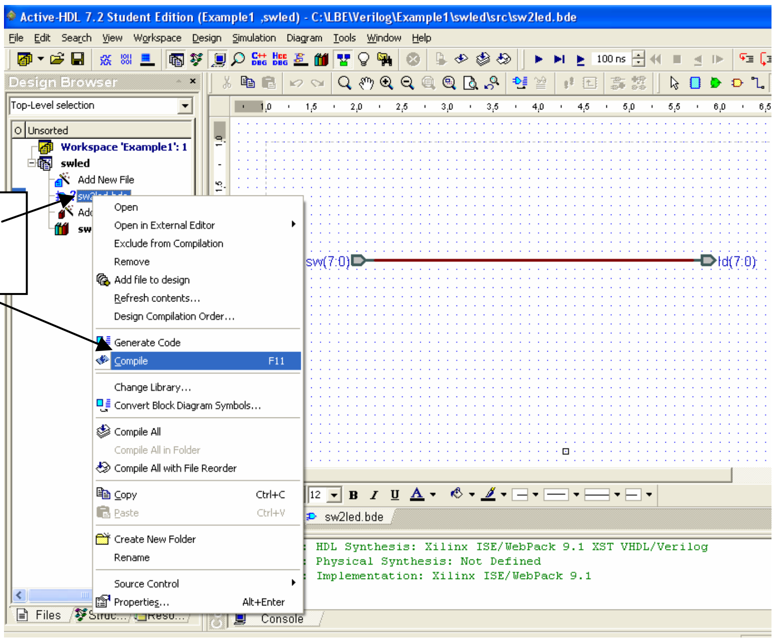


Click *Save*

```
# DESIGN: HDL Synthesis: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog
# DESIGN: Physical Synthesis: Not Defined
# DESIGN: Implementation: Xilinx ISE/WebPack 9.1
```

Press space bar to anchor a corner. 5.480 inch 3.030 inch Page 1/1

Right-click on *sw2led.bde* and select *Compile*

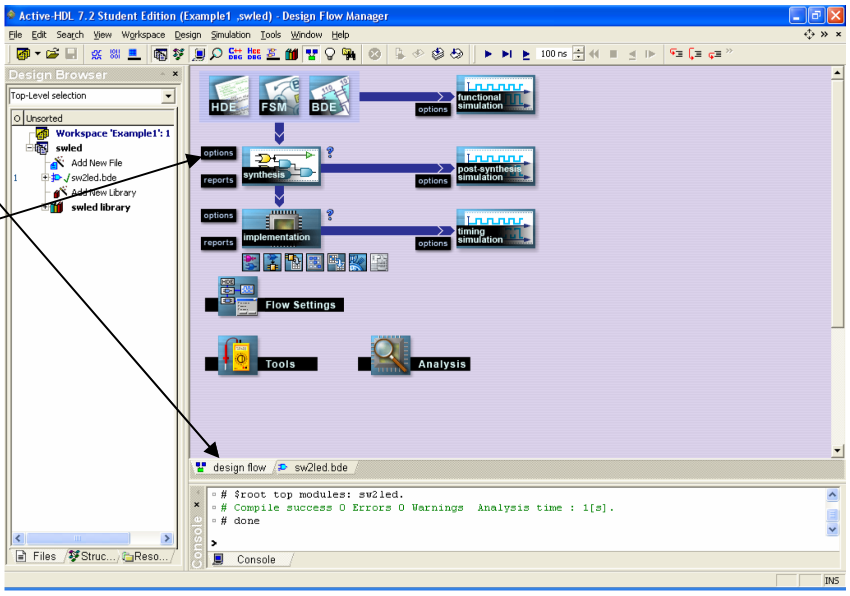


```
HDL Synthesis: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog
Physical Synthesis: Not Defined
Implementation: Xilinx ISE/WebPack 9.1
```

Part 3: Synthesis and Implementation

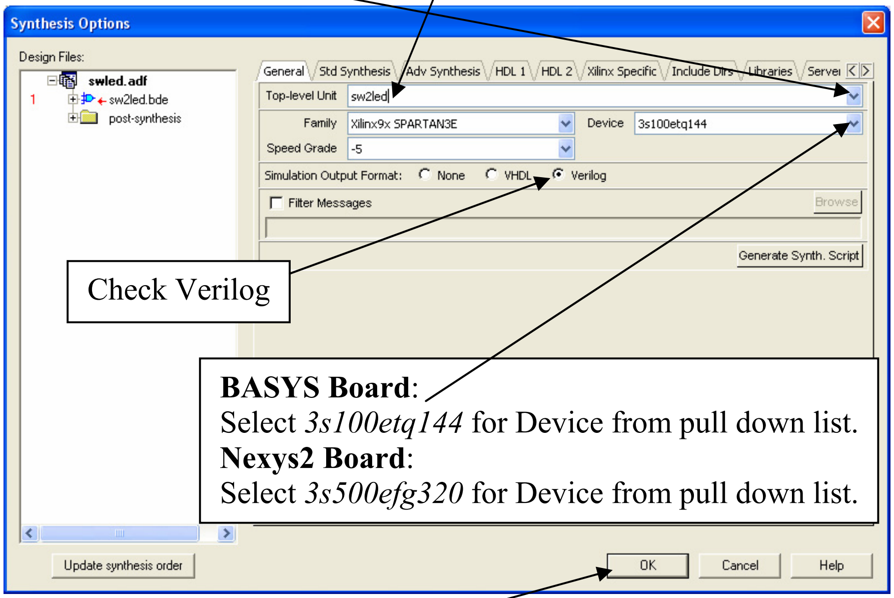
Click *design flow*

Click synthesis *options*



```
# $root top modules: sw2led.
# Compile success 0 Errors 0 Warnings  Analysis time : 1[s].
# done
```

Pull down menu and select *sw2led* for Top-level Unit.



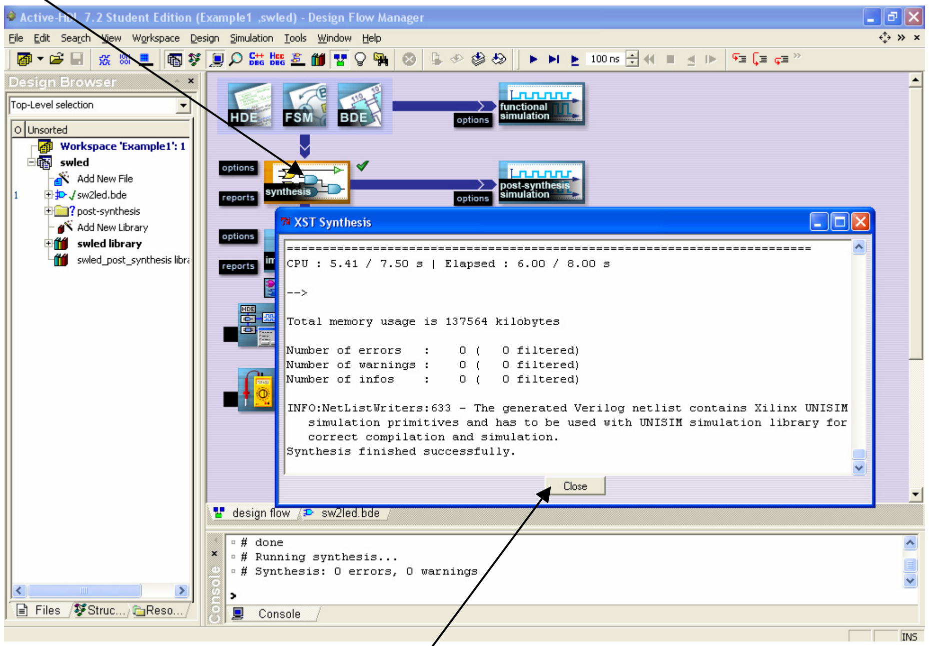
Top-level Unit: sw2led
Family: Xilinx9x SPARTAN3E
Device: 3s100etq144
Speed Grade: -5
Simulation Output Format: None VHDL [✓ Verilog]
Filter Messages

Check Verilog

BASYS Board:
Select *3s100etq144* for Device from pull down list.
Nexys2 Board:
Select *3s500efg320* for Device from pull down list.

Click *Ok*.

Click *synthesis*

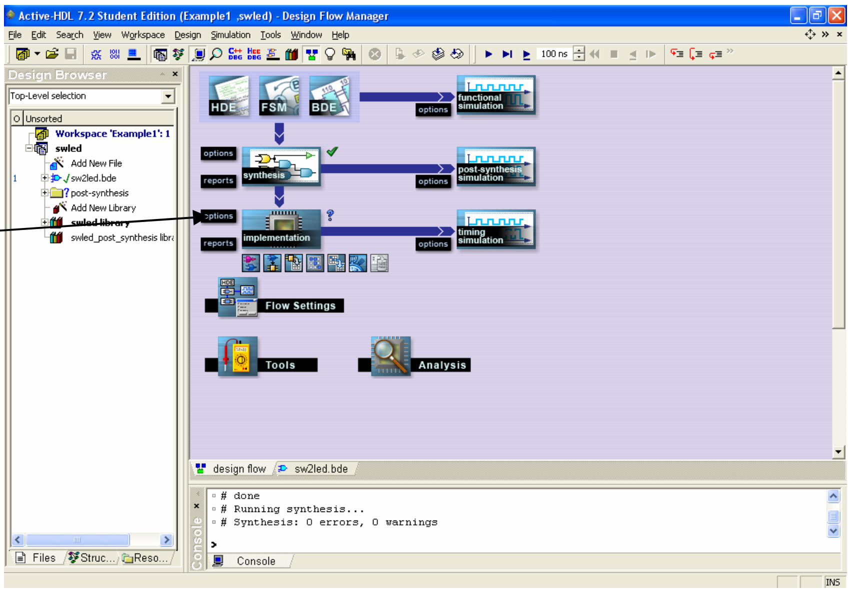


```
================================================================================
CPU : 5.41 / 7.50 s | Elapsed : 6.00 / 8.00 s

-->

Total memory usage is 137564 kilobytes

Number of errors   :    0 (   0 filtered)
Number of warnings :    0 (   0 filtered)
Number of infos    :    0 (   0 filtered)

INFO:NetListWriters:633 - The generated Verilog netlist contains Xilinx UNISIM
   simulation primitives and has to be used with UNISIM simulation library for
   correct compilation and simulation.
Synthesis finished successfully.
```

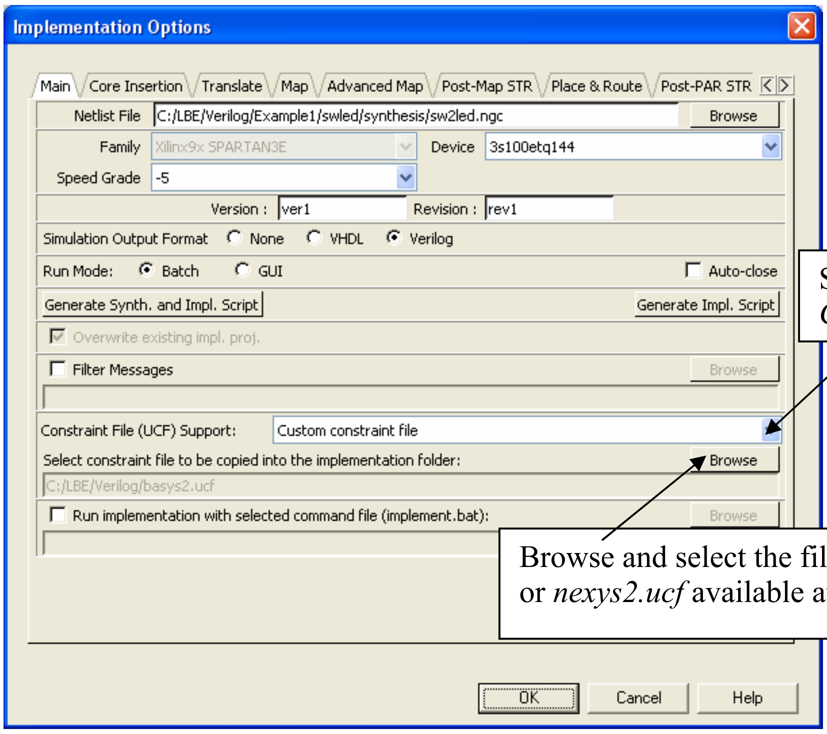
```
# done
# Running synthesis...
# Synthesis: 0 errors, 0 warnings
>
```

After synthesis is complete, click *Close*.

Click *implementation options*



```
# done
# Running synthesis...
# Synthesis: 0 errors, 0 warnings
```

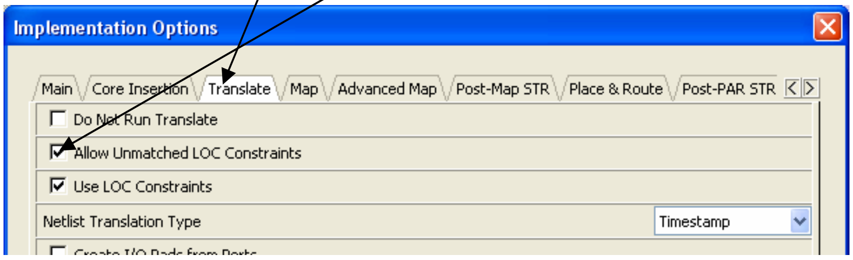


Netlist File: C:/LBE/Verilog/Example1/swled/synthesis/sw2led.ngc
Family: Xilinx9x SPARTAN3E
Device: 3s100etq144
Speed Grade: -5
Version: ver1
Revision: rev1
Simulation Output Format: None VHDL [✓ Verilog]
Run Mode: [✓ Batch] GUI
Auto-close
[✓ Overwrite existing impl. proj.]
Filter Messages
Constraint File (UCF) Support: Custom constraint file
Select constraint file to be copied into the implementation folder:
C:/LBE/Verilog/basys2.ucf
Run implementation with selected command file (implement.bat):

Select *Custom constraint file*

Browse and select the file *basys2.ucf* or *nexys2.ucf* available at www.lbebooks.com

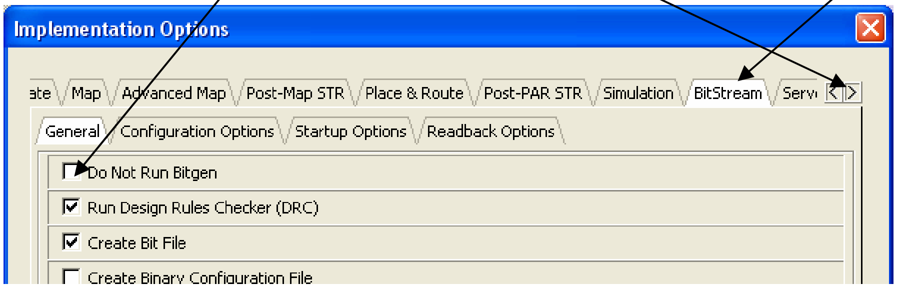
Select *Translate* and check *Allow Unmatched LOC Constraints.*



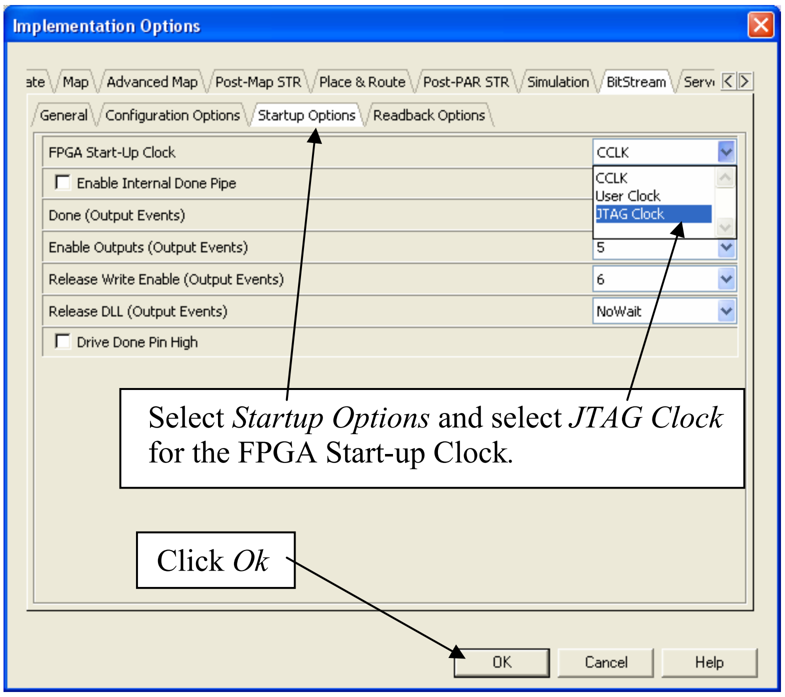
- [ ] Do Not Run Translate
- [x] Allow Unmatched LOC Constraints
- [x] Use LOC Constraints

Netlist Translation Type: Timestamp

Shift for more options…. Select *BitStream* and uncheck *Do Not Run Bitgen.*



- [ ] Do Not Run Bitgen
- [x] Run Design Rules Checker (DRC)
- [x] Create Bit File
- [ ] Create Binary Configuration File

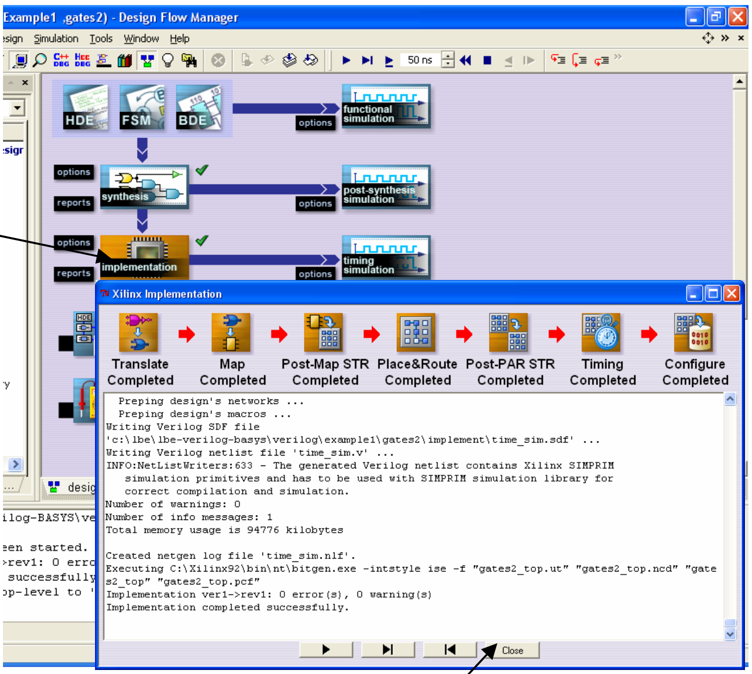


FPGA Start-Up Clock	CCLK
Enable Internal Done Pipe	☐
Done (Output Events)	
Enable Outputs (Output Events)	5
Release Write Enable (Output Events)	6
Release DLL (Output Events)	NoWait
Drive Done Pin High	☐

Select *Startup Options* and select *JTAG Clock* for the FPGA Start-up Clock.

Click *Ok*

Click *implementation*

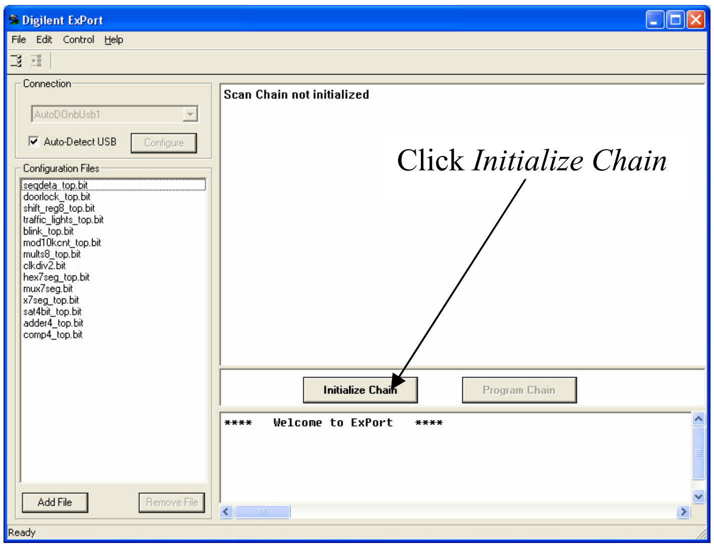


```
  Preping design's networks ...
  Preping design's macros ...
Writing Verilog SDF file
'c:\lbe\lbe-verilog-basys\verilog\example1\gates2\implement\time_sim.sdf' ...
Writing Verilog netlist file 'time_sim.v' ...
INFO:NetListWriters:633 - The generated Verilog netlist contains Xilinx SIMPRIM
   simulation primitives and has to be used with SIMPRIM simulation library for
   correct compilation and simulation.
Number of warnings: 0
Number of info messages: 1
Total memory usage is 94776 kilobytes

Created netgen log file 'time_sim.nlf'.
Executing C:\Xilinx92\bin\nt\bitgen.exe -intstyle ise -f "gates2_top.ut" "gates2_top.ncd" "gate
s2_top" "gates2_top.pcf"
Implementation ver1->rev1: 0 error(s), 0 warning(s)
Implementation completed successfully.
```

When implementation is complete click *Close.*

Part 4: Program FPGA Board

To program the Spartan3E on the BASYS or Nexys-2 boards we will use the **ExPort** tool that is part of the the **Adept Suite** available free from Digilent at http://www.digilentinc.com/Software/Adept.cfm?Nav1=Software&Nav2=Adept Double-click the **ExPort** icon on the desktop.



Scan Chain not initialized

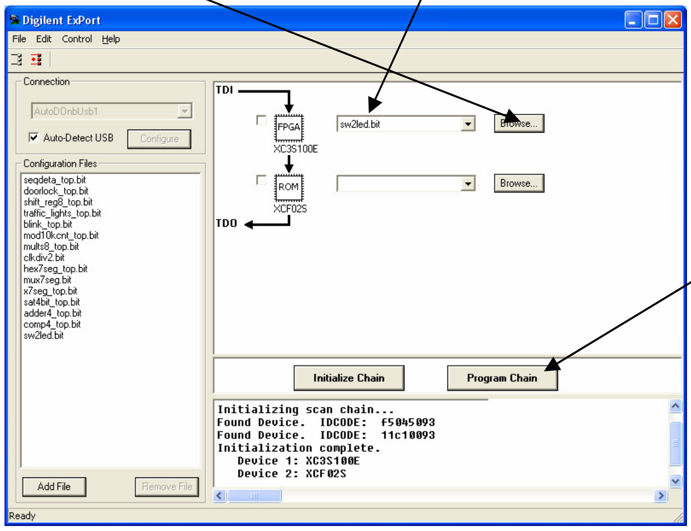
```
seqdeta_top.bit
doorlock_top.bit
shift_reg8_top.bit
traffic_lights_top.bit
blink_top.bit
mod10kcnt_top.bit
mults8_top.bit
clkdiv2.bit
hex7seg_top.bit
mux7seg.bit
x7seg_top.bit
sat4bit_top.bit
adder4_top.bit
comp4_top.bit
```

```
****   Welcome to ExPort   ****
```

Click *Initialize Chain*

Click *Browse* and go to *Example1->swled->implement->ver1->rev1->sw2led.bit*
Select *sw2led.bit*

Click *Program Chain*



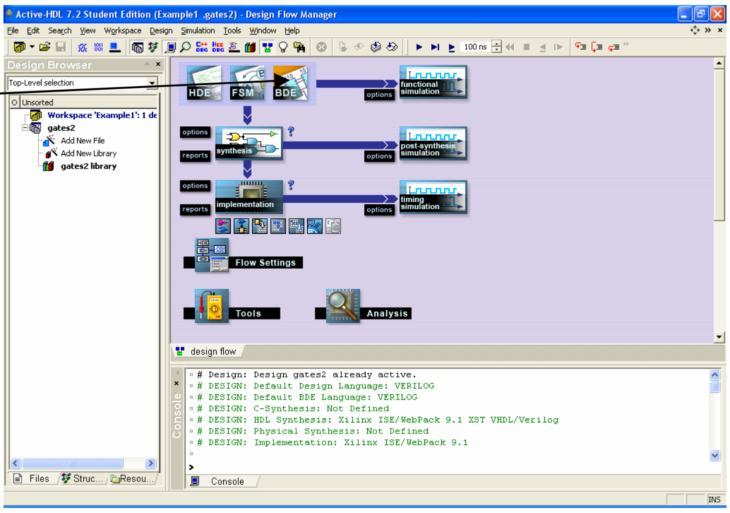
```
seqdeta_top.bit
doorlock_top.bit
shift_reg8_top.bit
traffic_lights_top.bit
blink_top.bit
mod10kcnt_top.bit
mults8_top.bit
clkdiv2.bit
hex7seg_top.bit
mux7seg.bit
x7seg_top.bit
sat4bit_top.bit
adder4_top.bit
comp4_top.bit
sw2led.bit
```

```
Initializing scan chain...
Found Device.  IDCODE:  f5045093
Found Device.  IDCODE:  11c10093
Initialization complete.
   Device 1: XC3S100E
   Device 2: XCF02S
```

Your program is now running on the board. Change the switches and watch the LEDs.

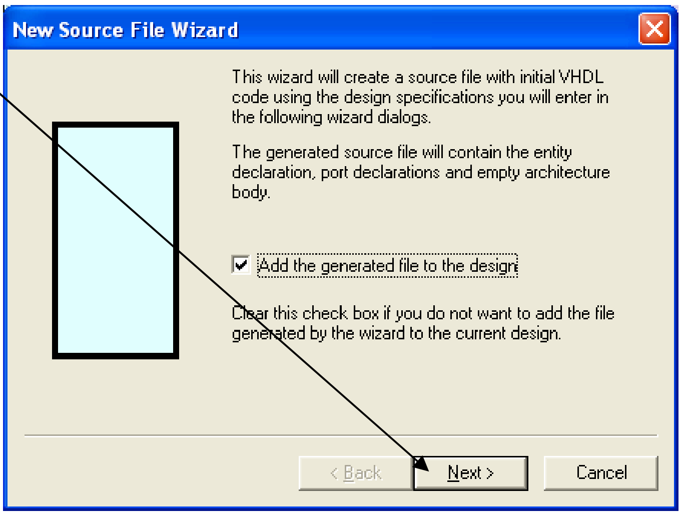
Part 5: Design Entry – gates2.bde

Click on *BDE*.



```
# Design: Design gates2 already active.
# DESIGN: Default Design Language: VERILOG
# DESIGN: Default BDE Language: VERILOG
# DESIGN: C-Synthesis: Not Defined
# DESIGN: HDL Synthesis: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog
# DESIGN: Physical Synthesis: Not Defined
# DESIGN: Implementation: Xilinx ISE/WebPack 9.1
```

Click *Next*.



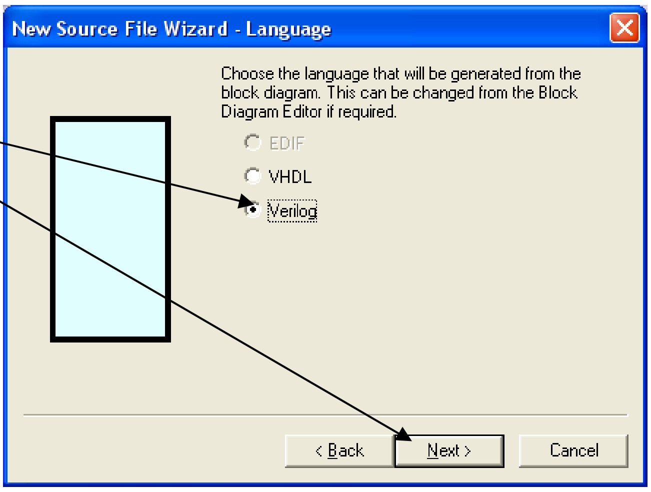
This wizard will create a source file with initial VHDL code using the design specifications you will enter in the following wizard dialogs.

The generated source file will contain the entity declaration, port declarations and empty architecture body.

[✓] Add the generated file to the design

Clear this check box if you do not want to add the file generated by the wizard to the current design.

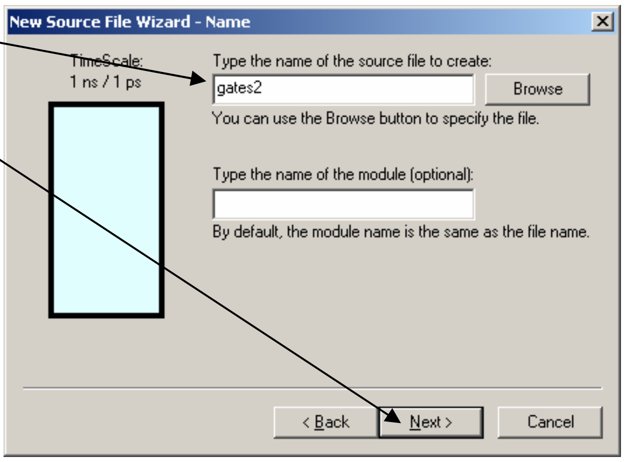
Select *Verilog*
and Click *Next*



Choose the language that will be generated from the block diagram. This can be changed from the Block Diagram Editor if required.

EDIF
VHDL
[✓ Verilog]

Type *gates2* and click *Next*.



New Source File Wizard - Name

TimeScale: 1 ns / 1 ps

Type the name of the source file to create: gates2

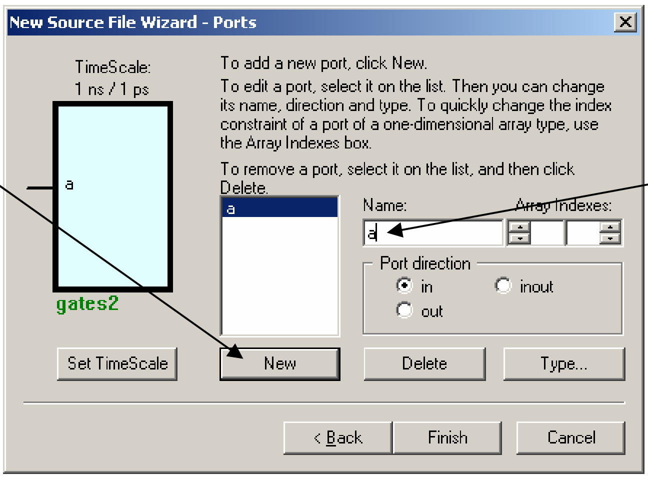
You can use the Browse button to specify the file.

Type the name of the module (optional):

By default, the module name is the same as the file name.

Click *New*.

Type *a*.



New Source File Wizard - Ports

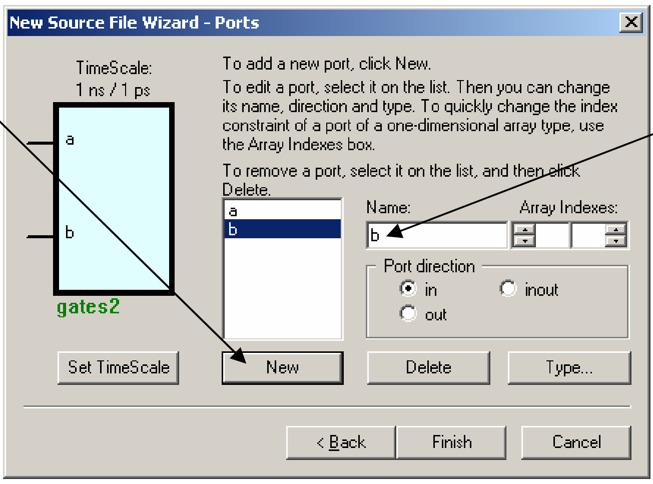
TimeScale: 1 ns / 1 ps

To add a new port, click New.
To edit a port, select it on the list. Then you can change its name, direction and type. To quickly change the index constraint of a port of a one-dimensional array type, use the Array Indexes box.
To remove a port, select it on the list, and then click Delete.

Name: a
Port direction: [✓ in] inout out

Click *New*.

Type *b*.

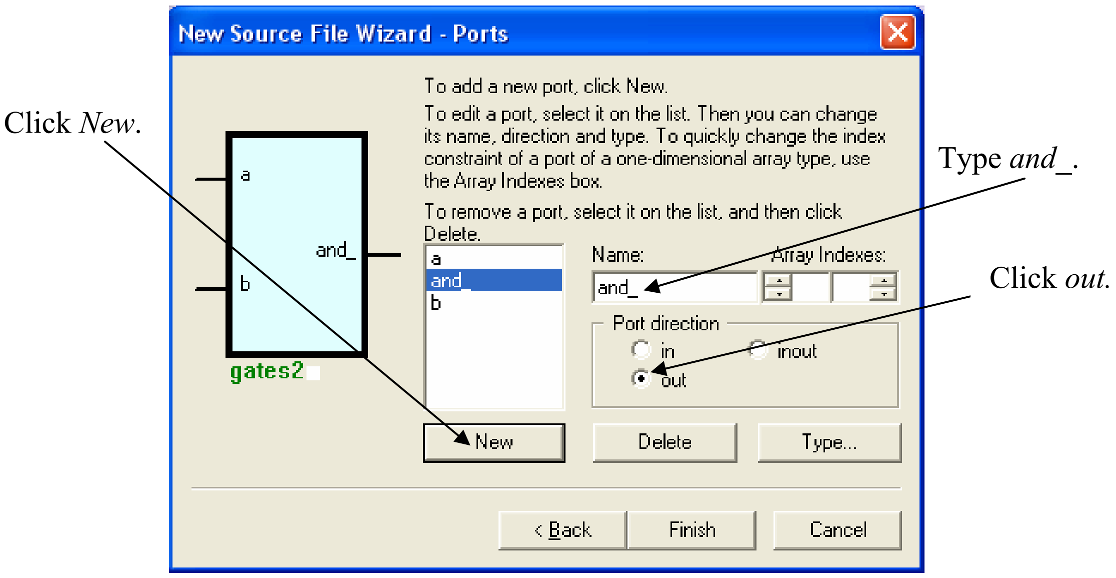


New Source File Wizard - Ports

TimeScale: 1 ns / 1 ps

To add a new port, click New.
To edit a port, select it on the list. Then you can change its name, direction and type. To quickly change the index constraint of a port of a one-dimensional array type, use the Array Indexes box.
To remove a port, select it on the list, and then click Delete.

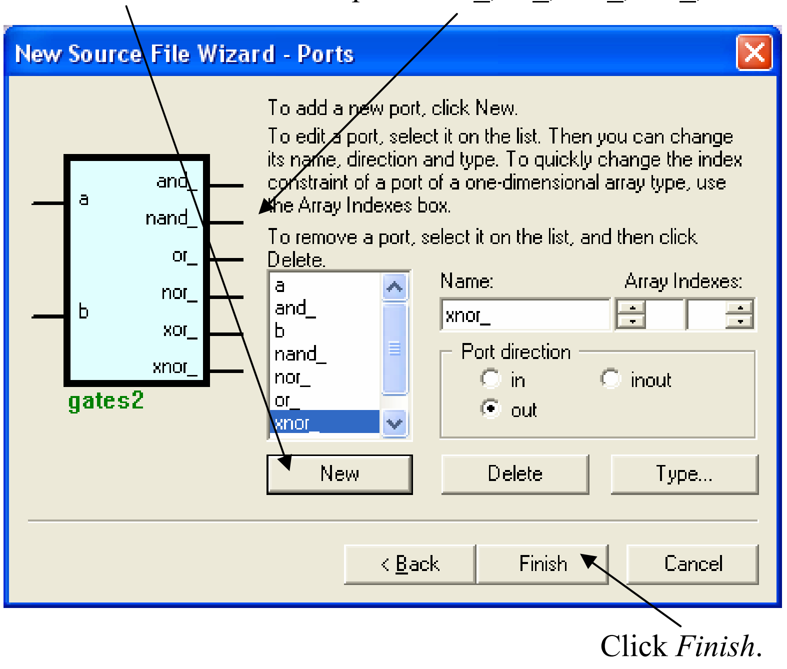
Name: b
Port direction: [✓ in] inout out



To add a new port, click New.
To edit a port, select it on the list. Then you can change its name, direction and type. To quickly change the index constraint of a port of a one-dimensional array type, use the Array Indexes box.
To remove a port, select it on the list, and then click Delete.

Name: and_ Array Indexes:
Port direction: in / inout / [✓ out]

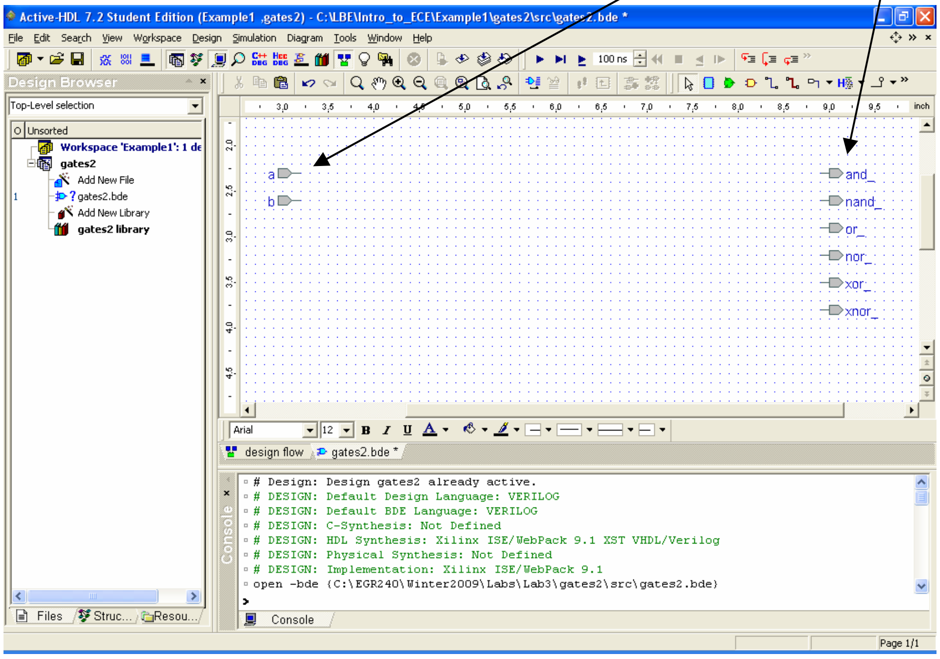
Continue to click *New* and add the outputs *nand_*, *or_*, *nor_*, *xor_*, and *xnor_*.



To add a new port, click New.
To edit a port, select it on the list. Then you can change its name, direction and type. To quickly change the index constraint of a port of a one-dimensional array type, use the Array Indexes box.
To remove a port, select it on the list, and then click Delete.

Name: xnor_ Array Indexes:
Port direction: in / inout / [✓ out]

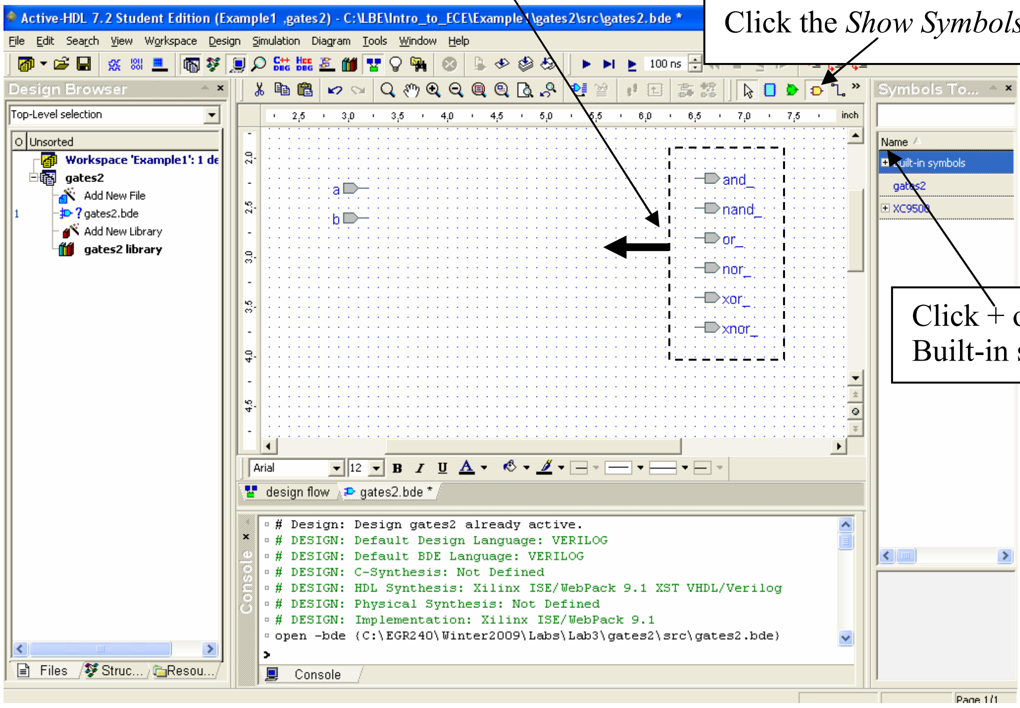
This will generate a block diagram (schematic) template with the *input* and *output* ports displayed.


Active-HDL 7.2 Student Edition (Example1 .gates2) - C:\LBE\Intro_to_ECE\Example1\gates2\src\gates2.bde *
File Edit Search View Workspace Design Simulation Diagram Tools Window Help
Design Browser
Top-Level selection
Unsorted
Workspace 'Example1': 1 de
gates2
Add New File
? gates2.bde
Add New Library
gates2 library
a
b
and_
nand_
or_
nor_
xor_
xnor_
design flow
gates2.bde *
Console
Files
Struc...
Resou...
Page 1/1

```
# Design: Design gates2 already active.
# DESIGN: Default Design Language: VERILOG
# DESIGN: Default BDE Language: VERILOG
# DESIGN: C-Synthesis: Not Defined
# DESIGN: HDL Synthesis: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog
# DESIGN: Physical Synthesis: Not Defined
# DESIGN: Implementation: Xilinx ISE/WebPack 9.1
open -bde {C:\EGR240\Winter2009\Labs\Lab3\gates2\src\gates2.bde}
>
```

Select the output ports by dragging the mouse with the left mouse button down and move the output ports to the left.

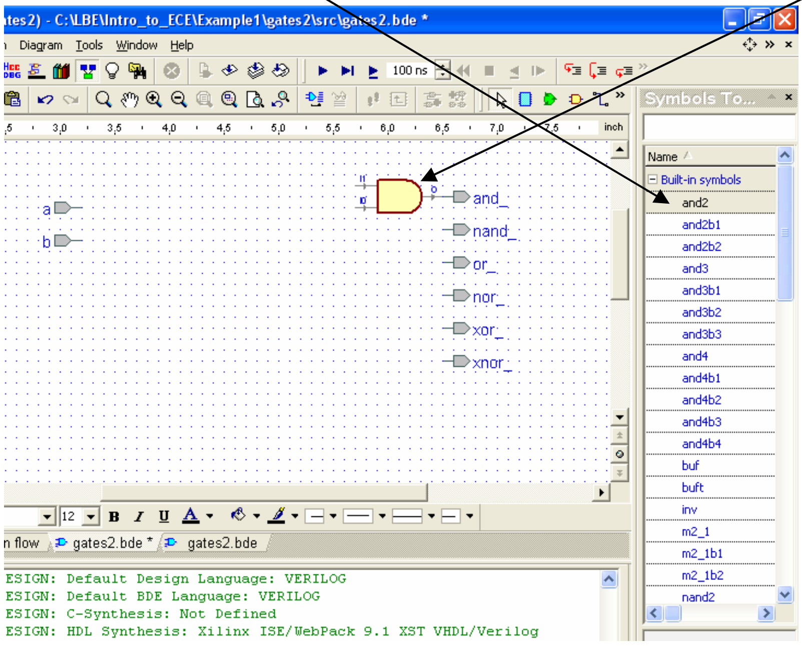
Click the *Show Symbols Toolbox* icon



```
# Design: Design gates2 already active.
# DESIGN: Default Design Language: VERILOG
# DESIGN: Default BDE Language: VERILOG
# DESIGN: C-Synthesis: Not Defined
# DESIGN: HDL Synthesis: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog
# DESIGN: Physical Synthesis: Not Defined
# DESIGN: Implementation: Xilinx ISE/WebPack 9.1
open -bde {C:\EGR240\Winter2009\Labs\Lab3\gates2\src\gates2.bde}
>
```

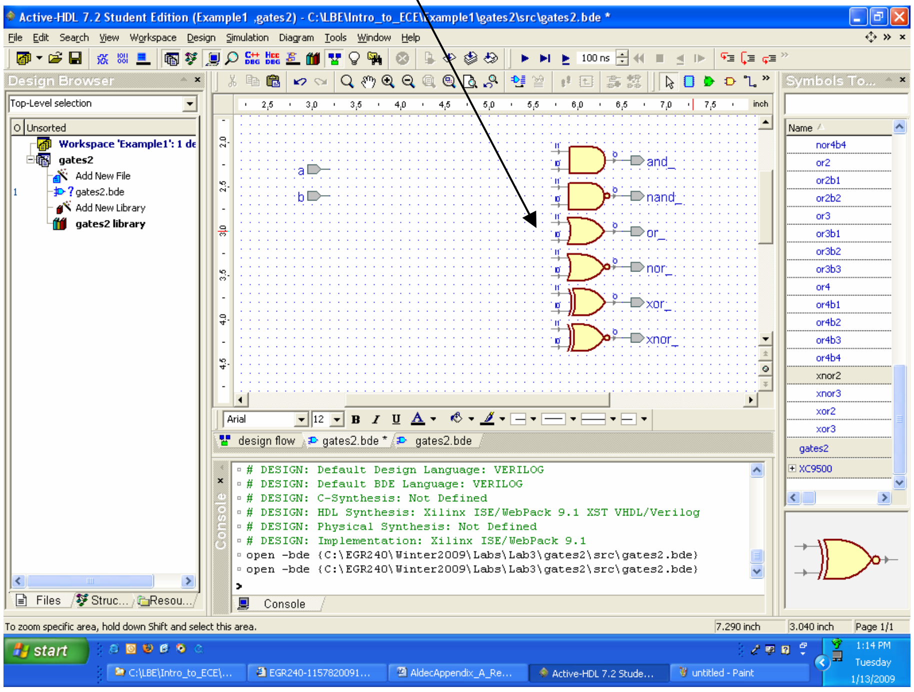
Click + on Built-in symbols

Grab the *and2* symbol with the mouse and drag it to the output port *and_*



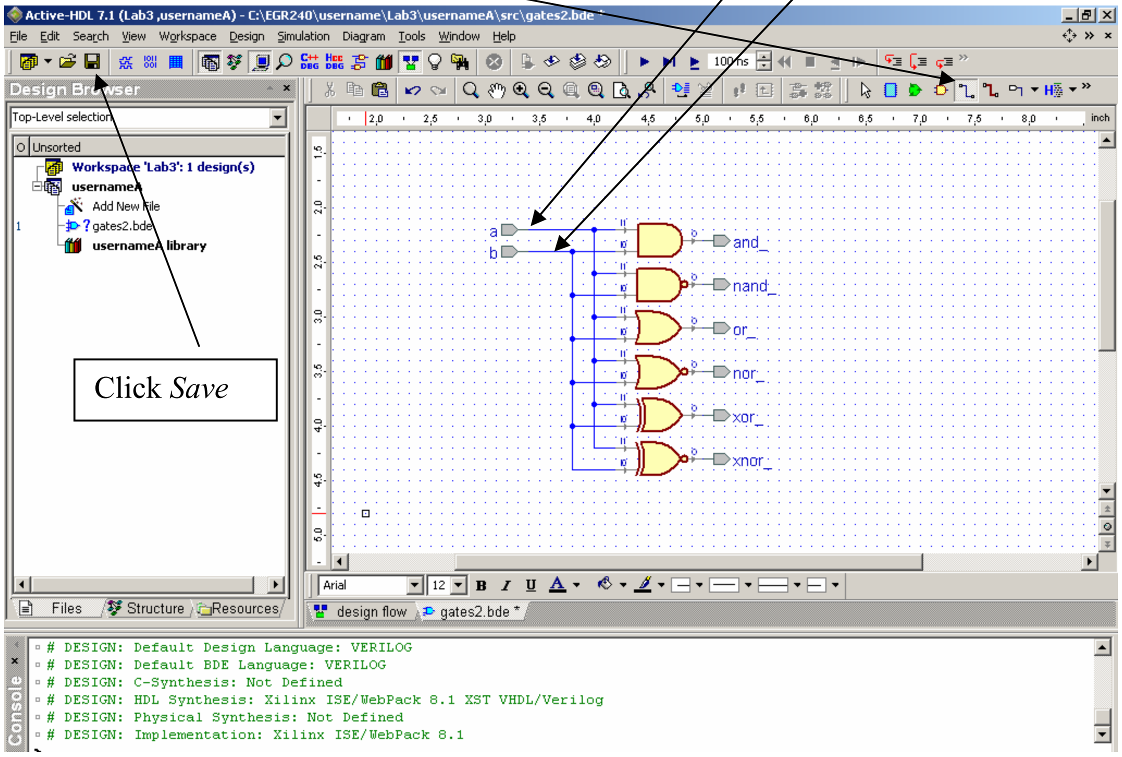
```
ESIGN: Default Design Language: VERILOG
ESIGN: Default BDE Language: VERILOG
ESIGN: C-Synthesis: Not Defined
ESIGN: HDL Synthesis: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog
```

Grab the symbols for *nand2*, *or2*, *nor2*, *xor2*, and *xnor2* and drag them to the appropriate output port, moving the output ports down as necessary.



```
# DESIGN: Default Design Language: VERILOG
# DESIGN: Default BDE Language: VERILOG
# DESIGN: C-Synthesis: Not Defined
# DESIGN: HDL Synthesis: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog
# DESIGN: Physical Synthesis: Not Defined
# DESIGN: Implementation: Xilinx ISE/WebPack 9.1
open -bde {C:\EGR240\Winter2009\Labs\Lab3\gates2\src\gates2.bde}
open -bde {C:\EGR240\Winter2009\Labs\Lab3\gates2\src\gates2.bde}
>
```

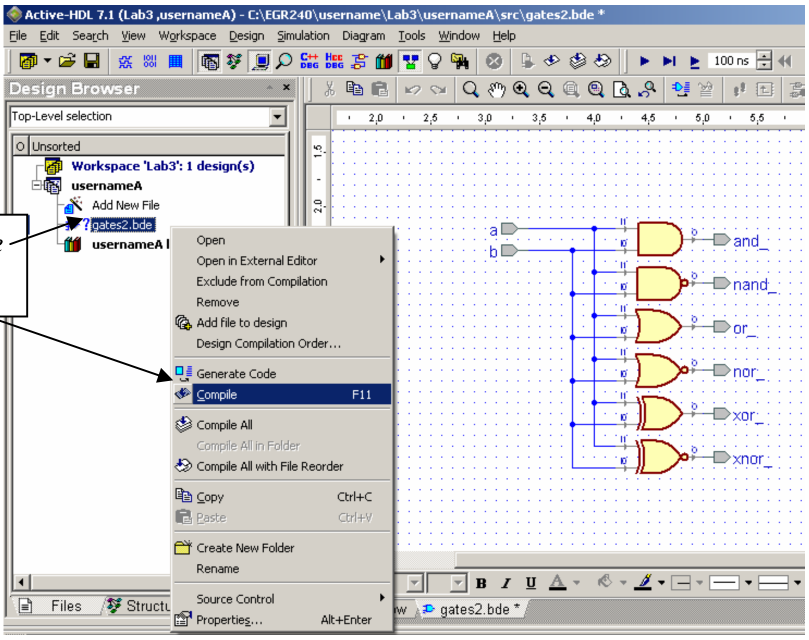
Select the *wire* icon and connect the gate inputs to *a* and *b* as shown.


Active-HDL 7.1 (Lab3 ,usernameA) - C:\EGR240\username\Lab3\usernameA\src\gates2.bde *
File Edit Search View Workspace Design Simulation Diagram Tools Window Help
Design Browser
Top-Level selection
Unsorted
Workspace 'Lab3': 1 design(s)
usernameA
Add New File
gates2.bde
usernameA library
Files Structure Resources
a
b
and_
nand_
or_
nor_
xor_
xnor_
design flow
gates2.bde *
Console

```
# DESIGN: Default Design Language: VERILOG
# DESIGN: Default BDE Language: VERILOG
# DESIGN: C-Synthesis: Not Defined
# DESIGN: HDL Synthesis: Xilinx ISE/WebPack 8.1 XST VHDL/Verilog
# DESIGN: Physical Synthesis: Not Defined
# DESIGN: Implementation: Xilinx ISE/WebPack 8.1
```

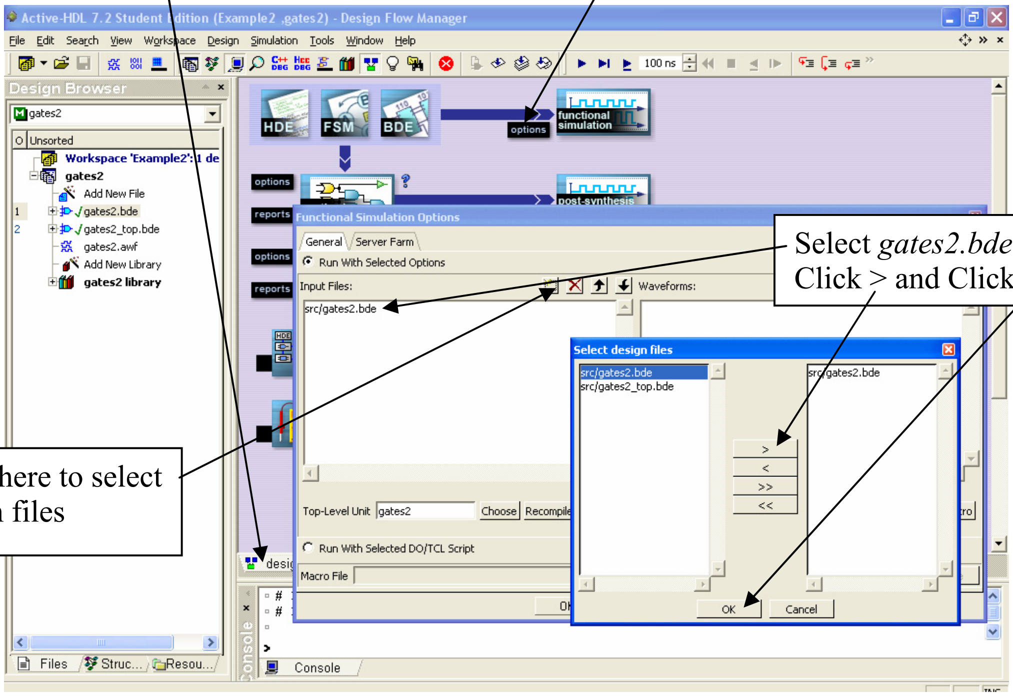
Click *Save*

Right-click on *gates2.bde* and select *Compile*

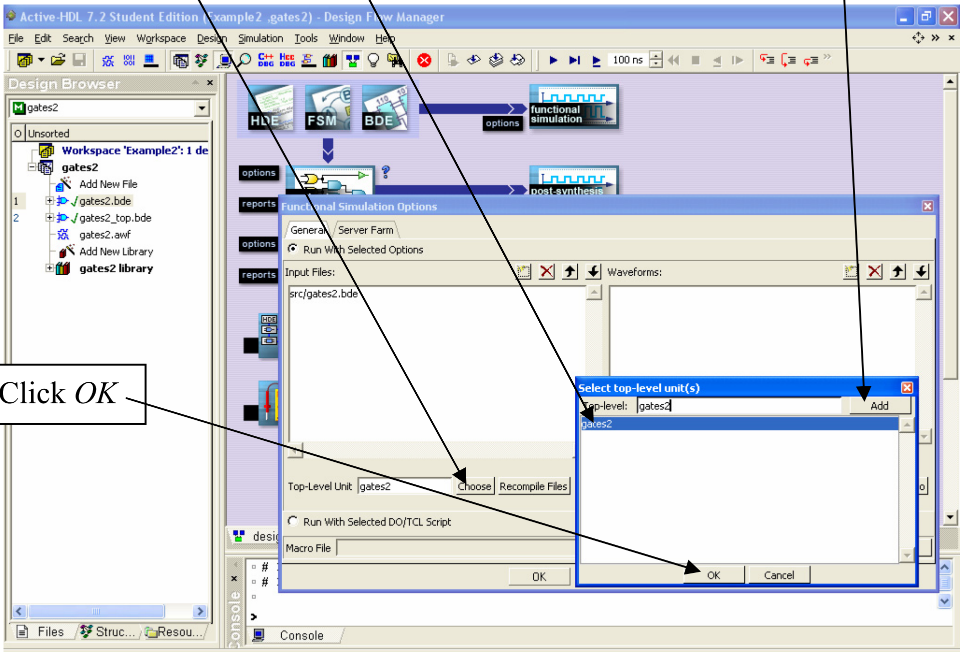


Part 6: Simulation

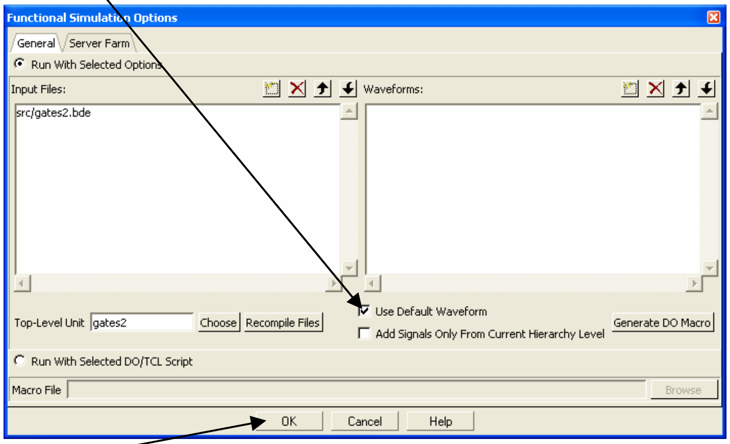
Click *design flow* and then Click *functional simulation options*



Click *Choose*, select *gates2* as the top-level design, and click *Add*.

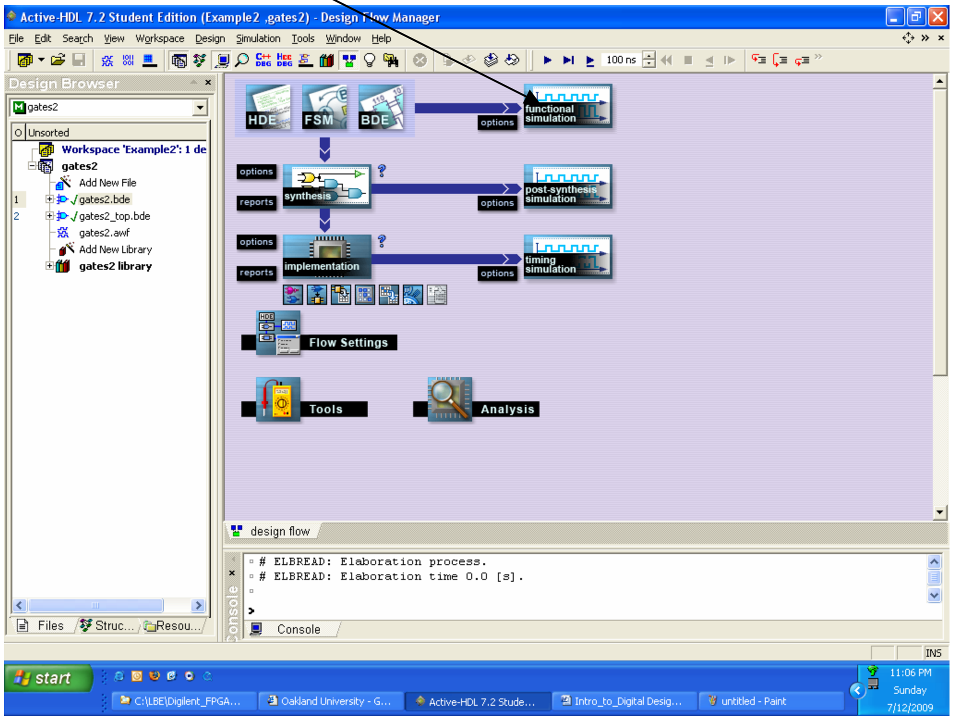


Click *Use Default Waveform*



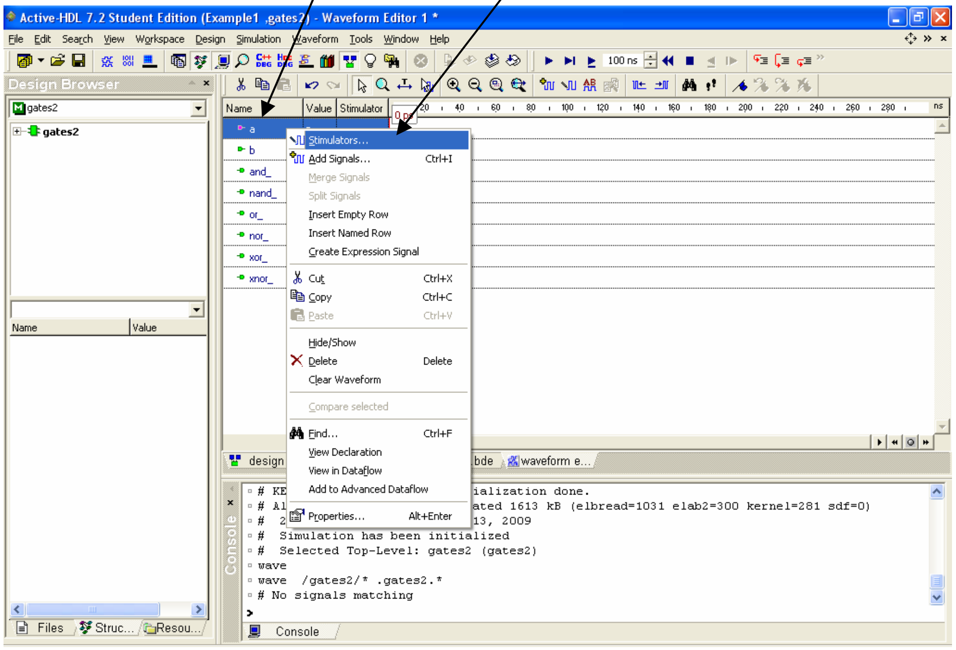
Click *OK*

Click *functional simulation*



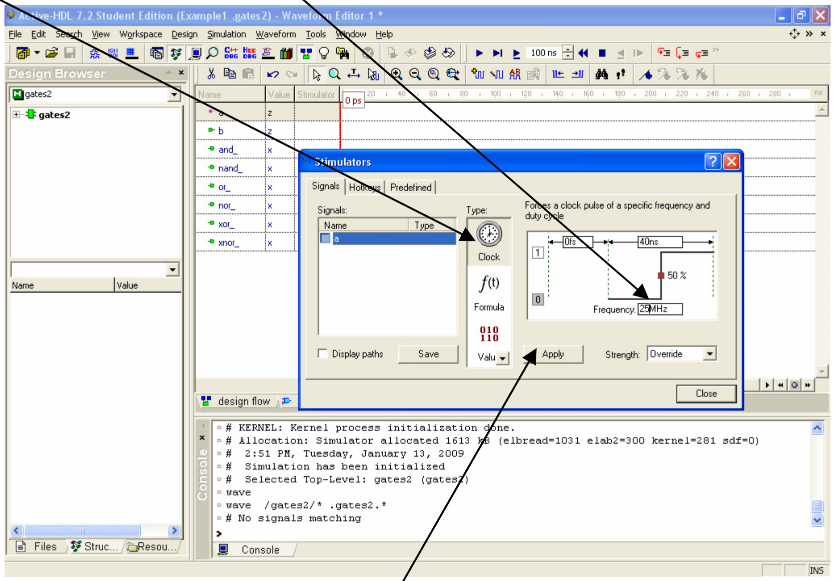
```
# ELBREAD: Elaboration process.
# ELBREAD: Elaboration time 0.0 [s].
```

The waveform window will automatically come up with the simulation already initialized. Make sure the order is *a*, *b*, *and_*, *nand_*, *or_*, *nor_*, *xor_*, *xnor* (grab and drag if necessary). Right-click on *a* and select *Stimulators*.



```
# KE...ialization done.
# Al...ated 1613 kB (elbread=1031 elab2=300 kernel=281 sdf=0)
# 2...13, 2009
# Simulation has been initialized
# Selected Top-Level: gates2 (gates2)
wave
wave /gates2/* .gates2.*
# No signals matching
```

Select *Clock* and set Frequency to 25 MHz

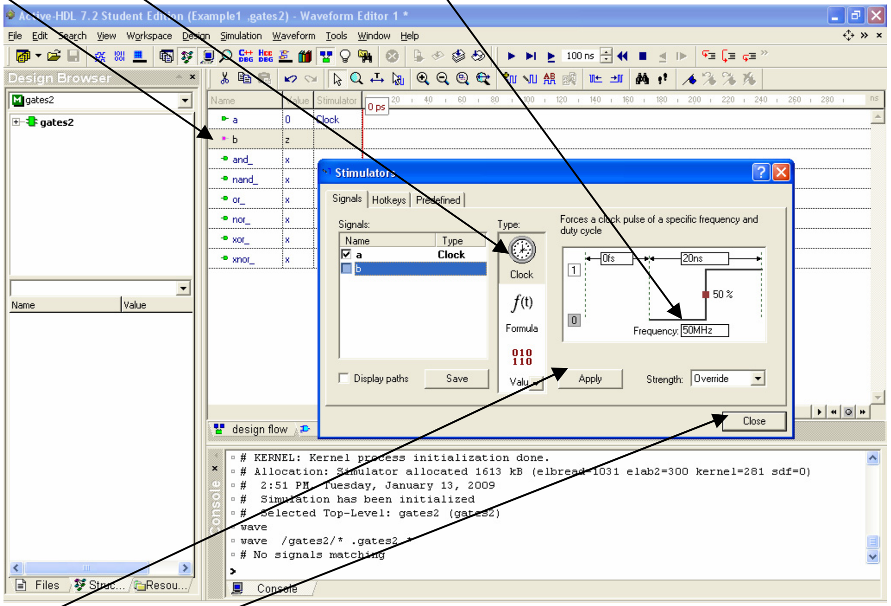

Active-HDL 7.2 Student Edition (Example1 .gates2) - Waveform Editor 1 *
Design Browser
gates2
Stimulators
Signals Hotkeys Predefined
Signals:
Name Type
a
Type:
Clock
Formula
Value
Forces a clock pulse of a specific frequency and duty cycle
0fs
40ns
50 %
Display paths
Save
Apply
Close
design flow
Console

Frequency: 25MHz
Strength: Override

```
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 1613 kB (elbread=1031 elab2=300 kernel=281 sdf=0)
#  2:51 PM, Tuesday, January 13, 2009
#  Simulation has been initialized
#  Selected Top-Level: gates2 (gates2)
wave
wave  /gates2/* .gates2.*
# No signals matching
```

Click *Apply*

Click on *b*, select *Clock* and set Frequency to 50 MHz



Signals:

Name	Type
a	Clock
b	

Forces a clock pulse of a specific frequency and duty cycle

Frequency: 50MHz

50 %

Strength: Override

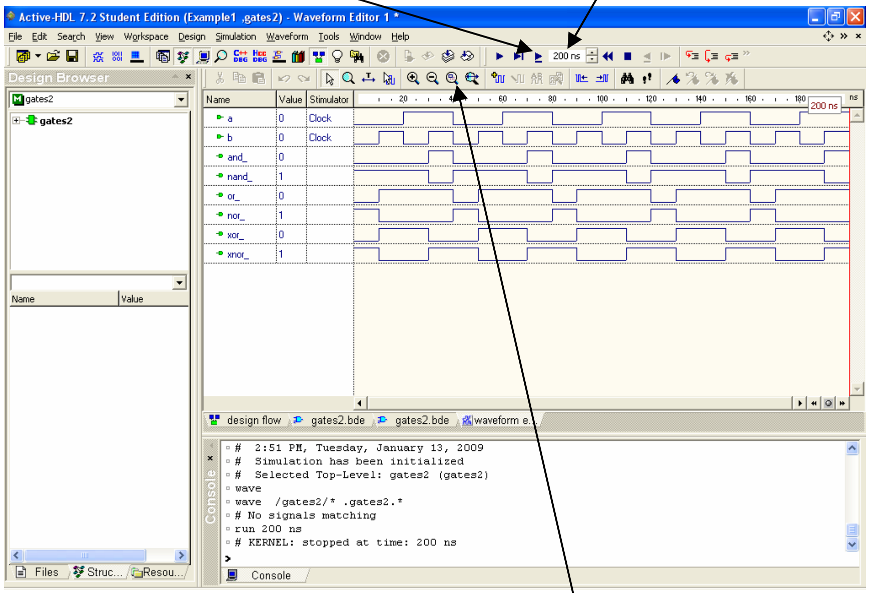
```
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 1613 kB (elbread=1031 elab2=300 kernel=281 sdf=0)
#  2:51 PM, Tuesday, January 13, 2009
#  Simulation has been initialized
#  Selected Top-Level: gates2 (gates2)
wave
wave  /gates2/* .gates2.*
# No signals matching
>
```

Click *Apply*

Click *Close*

Set simulation time to 200 ns

Click here to run simulation


Active-HDL 7.2 Student Edition (Example1 .gates2) - Waveform Editor 1 *
200 ns
design flow
gates2.bde
waveform e...
Console

Name	Value	Stimulator
a	0	Clock
b	0	Clock
and_	0	
nand_	1	
or_	0	
nor_	1	
xor_	0	
xnor_	1	

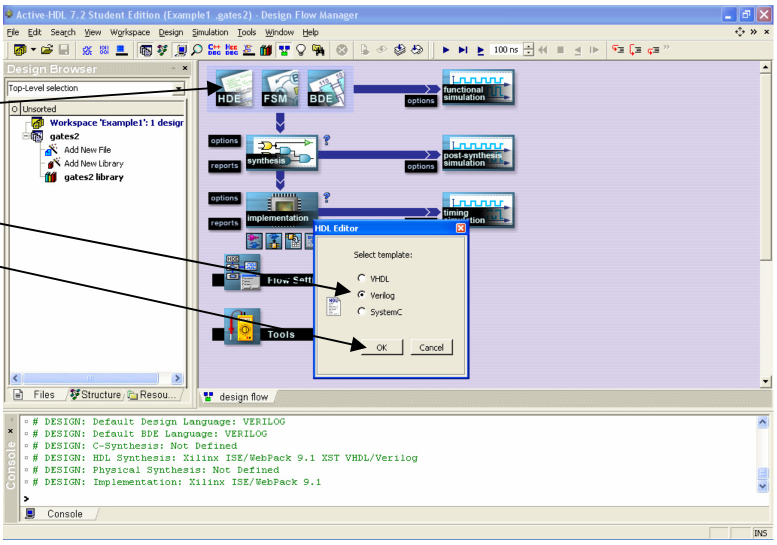
```
#  2:51 PM, Tuesday, January 13, 2009
#  Simulation has been initialized
#  Selected Top-Level: gates2 (gates2)
wave
wave  /gates2/* .gates2.*
# No signals matching
run 200 ns
# KERNEL: stopped at time: 200 ns
>
```

Click *Zoom to Fit.*

Part 7: Design Entry - HDE

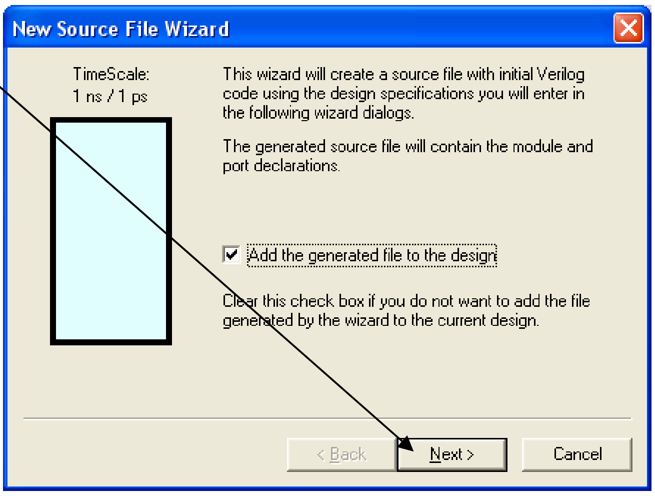
Click on HDE.

Select *Verilog* and Click *OK*.



```
# DESIGN: Default Design Language: VERILOG
# DESIGN: Default BDE Language: VERILOG
# DESIGN: C-Synthesis: Not Defined
# DESIGN: HDL Synthesis: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog
# DESIGN: Physical Synthesis: Not Defined
# DESIGN: Implementation: Xilinx ISE/WebPack 9.1
>
```

Click Next.



TimeScale:
1 ns / 1 ps

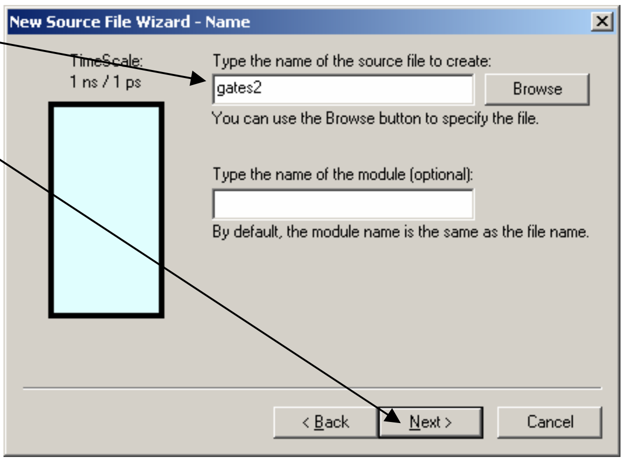
This wizard will create a source file with initial Verilog code using the design specifications you will enter in the following wizard dialogs.

The generated source file will contain the module and port declarations.

[✓] Add the generated file to the design

Clear this check box if you do not want to add the file generated by the wizard to the current design.

Type *gates2* and click *Next*.



TimeScale:
1 ns / 1 ps

Type the name of the source file to create: gates2

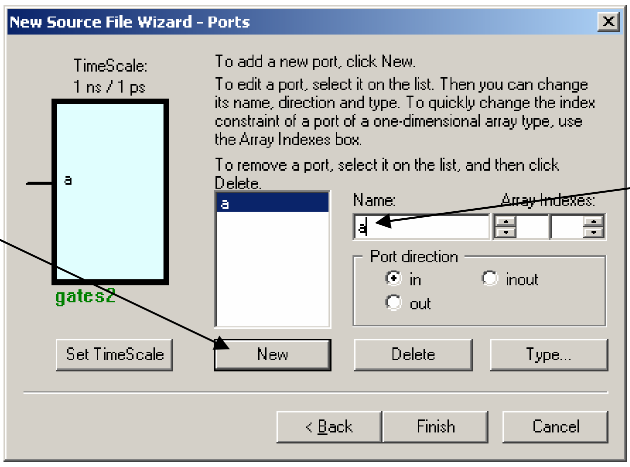
You can use the Browse button to specify the file.

Type the name of the module (optional):

By default, the module name is the same as the file name.

Click *New*.

Type *a*.


New Source File Wizard - Ports

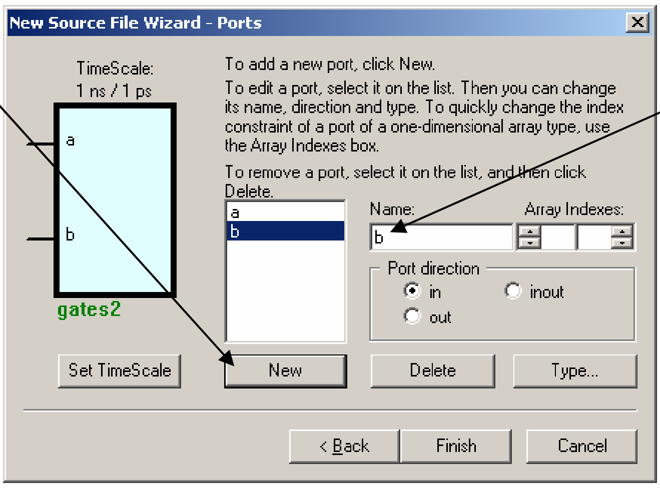
TimeScale:
1 ns / 1 ps

To add a new port, click New.
To edit a port, select it on the list. Then you can change its name, direction and type. To quickly change the index constraint of a port of a one-dimensional array type, use the Array Indexes box.
To remove a port, select it on the list, and then click Delete.

Name: a
Array Indexes:
Port direction: [✓ in] inout out

Click *New*.

Type *b*.


New Source File Wizard - Ports

TimeScale:
1 ns / 1 ps

To add a new port, click New.
To edit a port, select it on the list. Then you can change its name, direction and type. To quickly change the index constraint of a port of a one-dimensional array type, use the Array Indexes box.
To remove a port, select it on the list, and then click Delete.

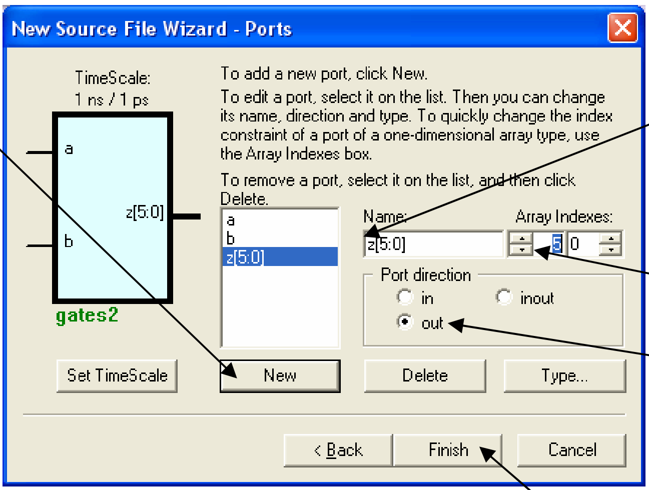
Name: b
Array Indexes:
Port direction: [✓ in] inout out

Click *New*.

Type *z*.

Set Array Indexes to 5:0.

Click *out*.


New Source File Wizard - Ports

TimeScale:
1 ns / 1 ps

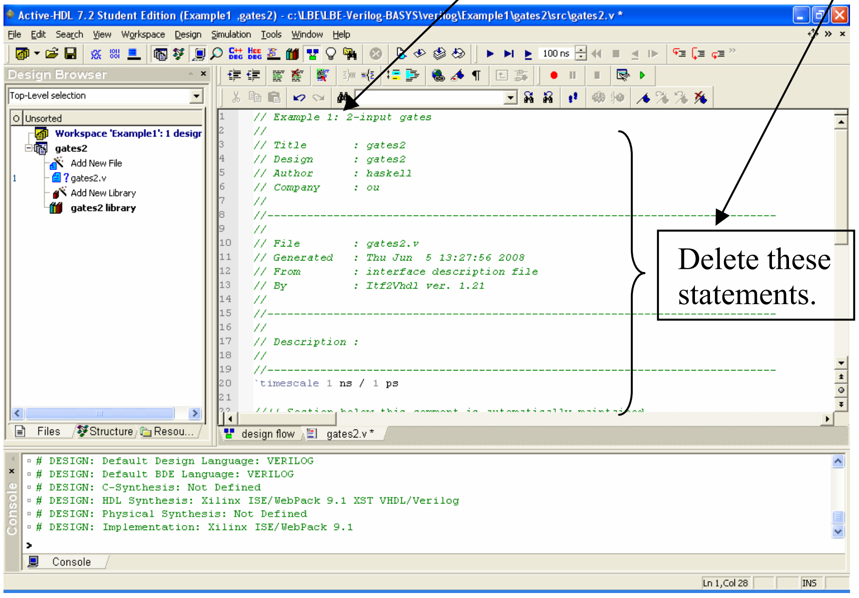
To add a new port, click New.
To edit a port, select it on the list. Then you can change its name, direction and type. To quickly change the index constraint of a port of a one-dimensional array type, use the Array Indexes box.
To remove a port, select it on the list, and then click Delete.

Name: z[5:0]
Array Indexes: 5 0
Port direction: in inout [✓ out]

Click *Finish*.

This will generate a Verilog template with the input and output signals filled in. Delete all the comments and replace them with the single comment

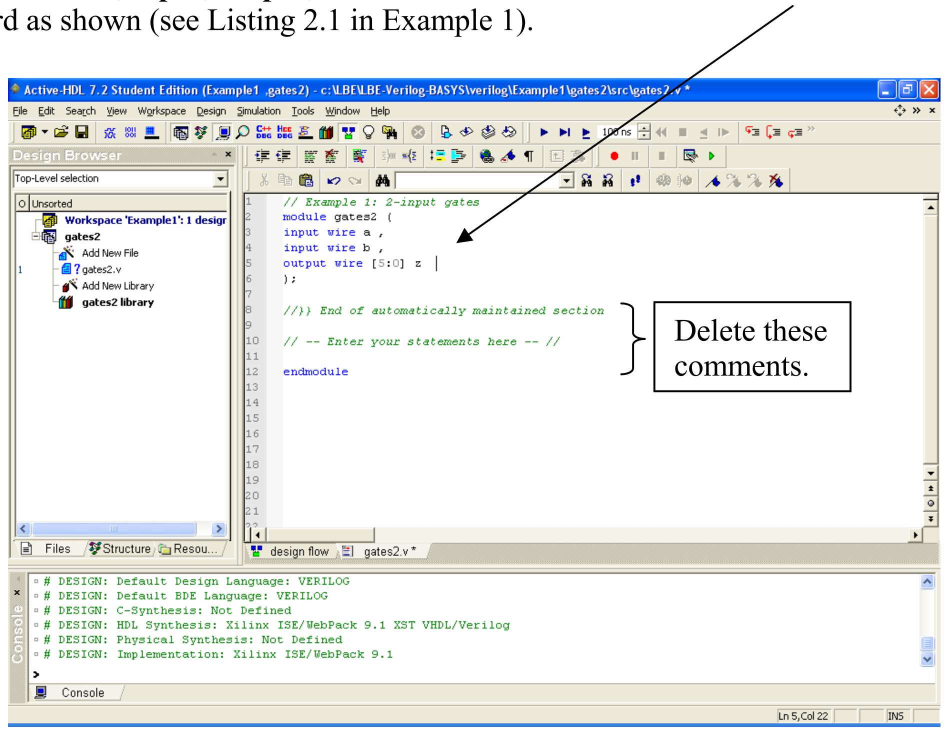
```
// Example 1: 2-input gates
```


Active-HDL 7.2 Student Edition (Example1 ,gates2) - c:\LBE\LBE-Verilog-BASYS\verilog\Example1\gates2\src\gates2.v *
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
Top-Level selection
Unsorted
Workspace 'Example1': 1 design
gates2
Add New File
gates2.v
Add New Library
gates2 library
Files Structure Resou...
design flow
gates2.v *
Console
Ln 1, Col 28
INS
Delete these statements.

```
// Example 1: 2-input gates
//
// Title       : gates2
// Design      : gates2
// Author      : haskell
// Company     : ou
//
//-----------------------------------------------------------------------------
//
// File        : gates2.v
// Generated   : Thu Jun  5 13:27:56 2008
// From        : interface description file
// By          : Itf2Vhdl ver. 1.21
//
//-----------------------------------------------------------------------------
//
// Description :
//
//-----------------------------------------------------------------------------
`timescale 1 ns / 1 ps
```

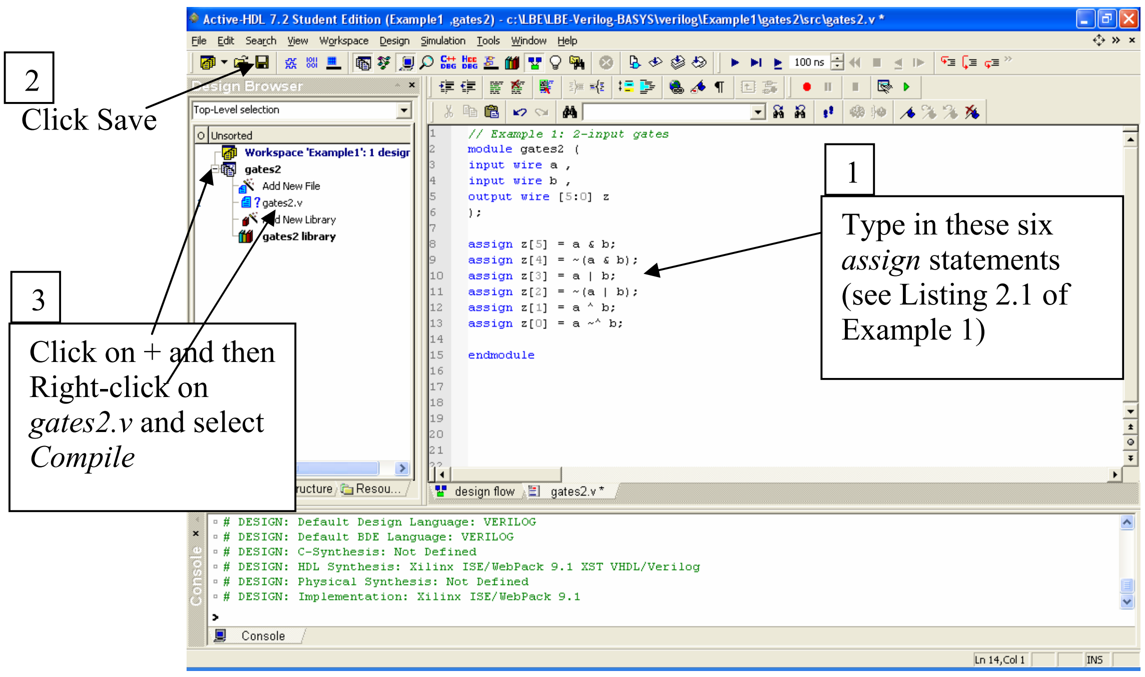
```
# DESIGN: Default Design Language: VERILOG
# DESIGN: Default BDE Language: VERILOG
# DESIGN: C-Synthesis: Not Defined
# DESIGN: HDL Synthesis: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog
# DESIGN: Physical Synthesis: Not Defined
# DESIGN: Implementation: Xilinx ISE/WebPack 9.1
>
```

Edit the **module**, **input**, **output**, and **wire** statements to conform to the 2001 Verilog standard as shown (see Listing 2.1 in Example 1).


Active-HDL 7.2 Student Edition (Example1 ,gates2) - c:\LBE\LBE-Verilog-BASYS\verilog\Example1\gates2\src\gates2.v *
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
Top-Level selection
Unsorted
Workspace 'Example1': 1 design
gates2
Add New File
gates2.v
Add New Library
gates2 library
Files Structure Resou...
design flow
gates2.v *
Console
Ln 5, Col 22
INS
Delete these comments.

```
// Example 1: 2-input gates
module gates2 (
input wire a ,
input wire b ,
output wire [5:0] z
);

//}} End of automatically maintained section

// -- Enter your statements here -- //

endmodule
```

```
# DESIGN: Default Design Language: VERILOG
# DESIGN: Default BDE Language: VERILOG
# DESIGN: C-Synthesis: Not Defined
# DESIGN: HDL Synthesis: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog
# DESIGN: Physical Synthesis: Not Defined
# DESIGN: Implementation: Xilinx ISE/WebPack 9.1
>
```


Active-HDL 7.2 Student Edition (Example1 .gates2) - c:\LBE\LBE-Verilog-BASYS\verilog\Example1\gates2\src\gates2.v *
Design Browser
Workspace 'Example1': 1 design
gates2
Add New File
gates2.v
Add New Library
gates2 library
design flow
gates2.v *
Console

2 Click Save

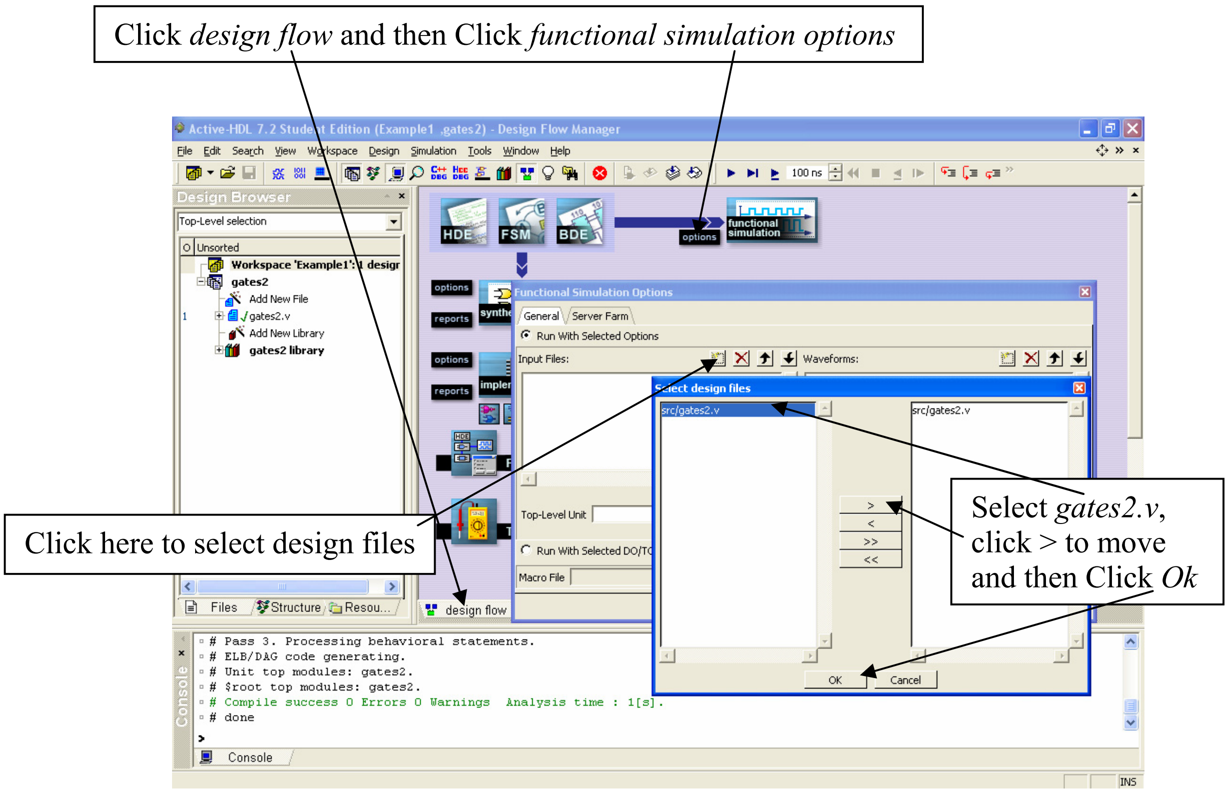
3 Click on + and then Right-click on *gates2.v* and select *Compile*

1 Type in these six *assign* statements (see Listing 2.1 of Example 1)

```
// Example 1: 2-input gates
module gates2 (
input wire a ,
input wire b ,
output wire [5:0] z
);

assign z[5] = a & b;
assign z[4] = ~(a & b);
assign z[3] = a | b;
assign z[2] = ~(a | b);
assign z[1] = a ^ b;
assign z[0] = a ~^ b;

endmodule
```

```
# DESIGN: Default Design Language: VERILOG
# DESIGN: Default BDE Language: VERILOG
# DESIGN: C-Synthesis: Not Defined
# DESIGN: HDL Synthesis: Xilinx ISE/WebPack 9.1 XST VHDL/Verilog
# DESIGN: Physical Synthesis: Not Defined
# DESIGN: Implementation: Xilinx ISE/WebPack 9.1
```

Part 8: Simulation – gates2



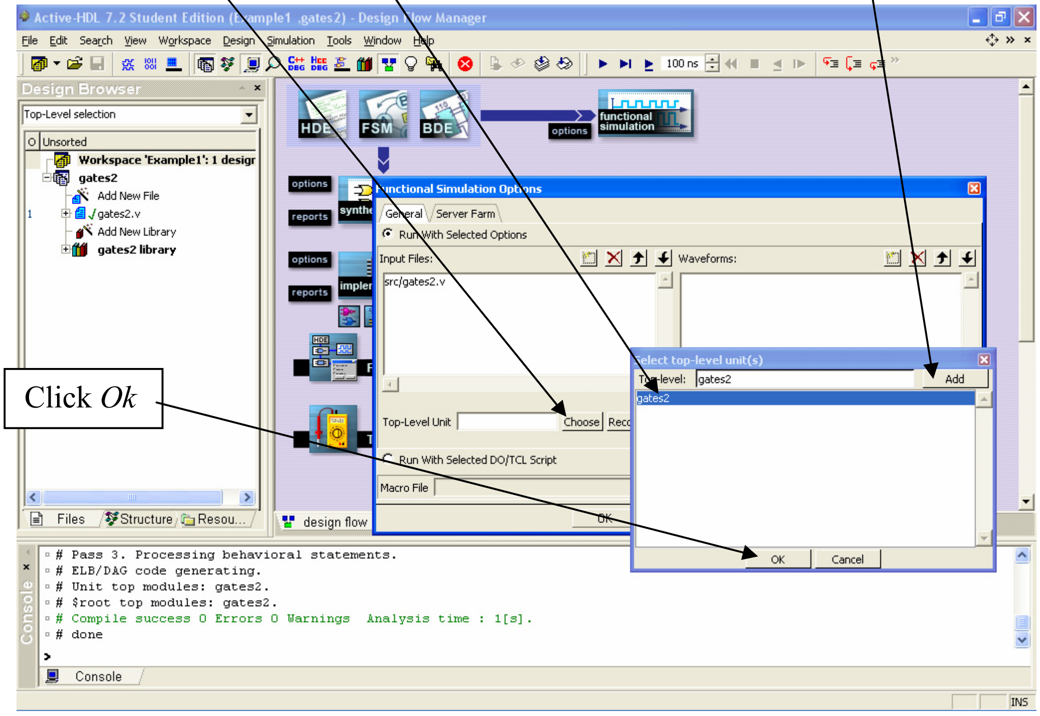
Click *design flow* and then Click *functional simulation options*

Click here to select design files

Select *gates2.v*, click > to move and then Click *Ok*

```
# Pass 3. Processing behavioral statements.
# ELB/DAG code generating.
# Unit top modules: gates2.
# $root top modules: gates2.
# Compile success 0 Errors 0 Warnings  Analysis time : 1[s].
# done
```

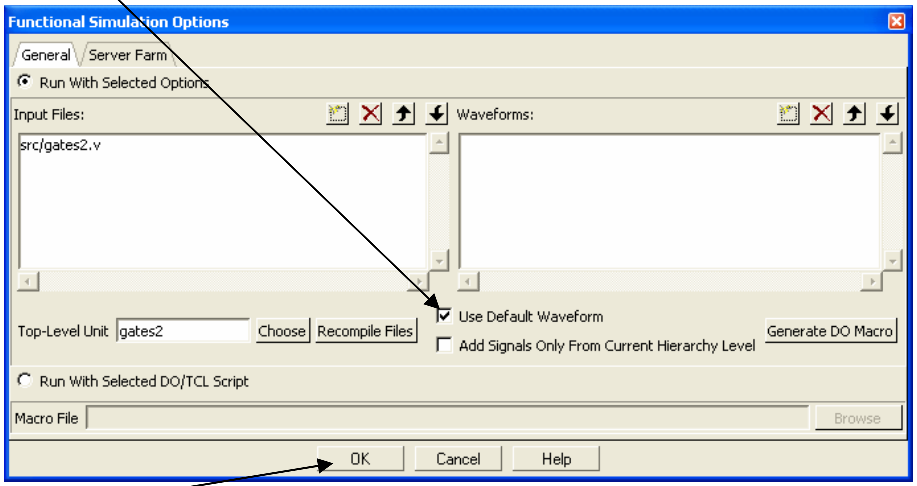
Click *Choose*, select *gates2* as the top-level design, and click *Add*.



```
# Pass 3. Processing behavioral statements.
# ELB/DAG code generating.
# Unit top modules: gates2.
# $root top modules: gates2.
# Compile success 0 Errors 0 Warnings  Analysis time : 1[s].
# done
>
```

Click *Ok*

Click *Use Default Waveform*



(•) Run With Selected Options

Input Files: src/gates2.v

Top-Level Unit: gates2

[✓] Use Default Waveform

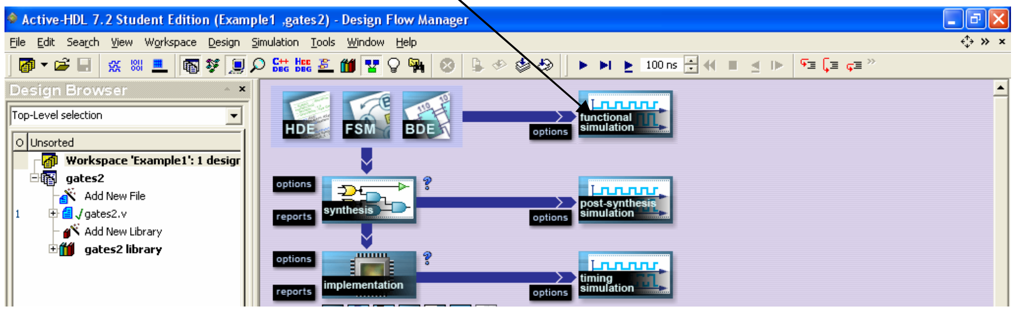
[ ] Add Signals Only From Current Hierarchy Level

( ) Run With Selected DO/TCL Script

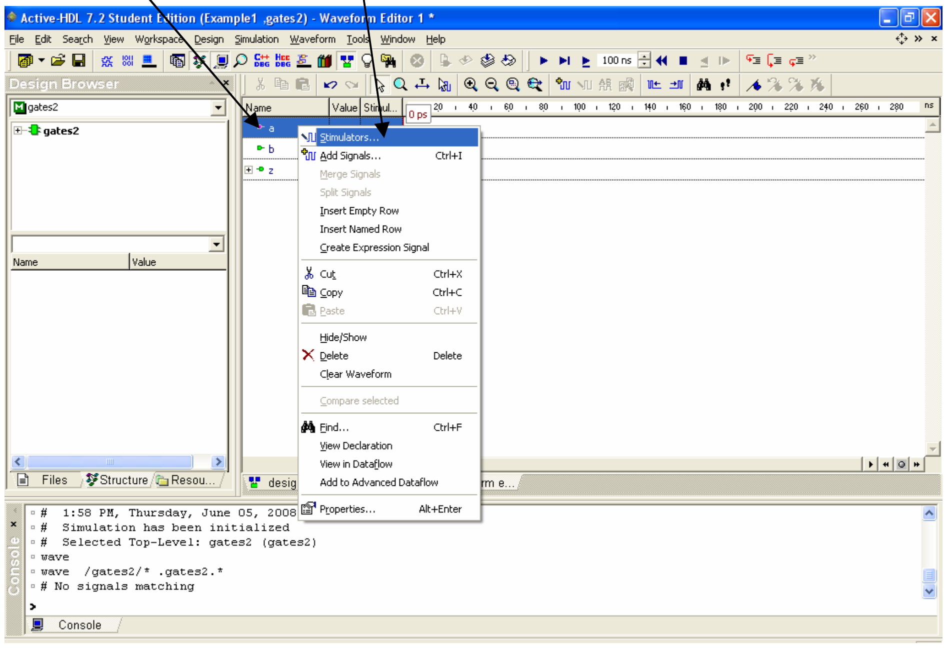
Macro File:

Click *Ok*

Click *functional simulation*

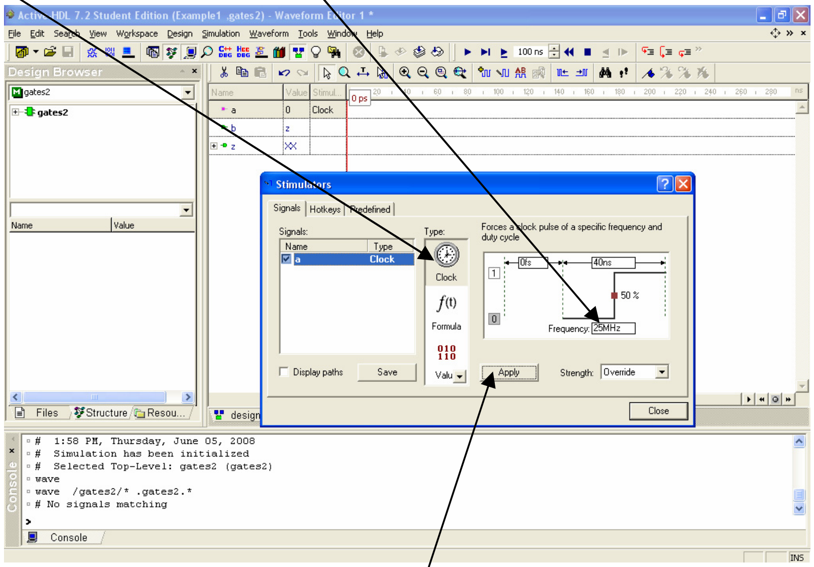


The waveform window will automatically come up with the simulation already initialized. Make sure the order is *a*, *b*, *z* (grab and drag if necessary). Right-click on *a* and select *Stimulators*.



```
# 1:58 PM, Thursday, June 05, 2008
# Simulation has been initialized
# Selected Top-Level: gates2 (gates2)
wave
wave /gates2/* .gates2.*
# No signals matching
>
```

Select *Clock* and set Frequency to 25 MHz


Active-HDL 7.2 Student Edition (Example1 .gates2) - Waveform Editor 1 *
Design Browser
Stimulators
Signals
Hotkeys
Predefined
Clock
Formula
Display paths
Save
Apply
Close
Forces a clock pulse of a specific frequency and duty cycle
Console

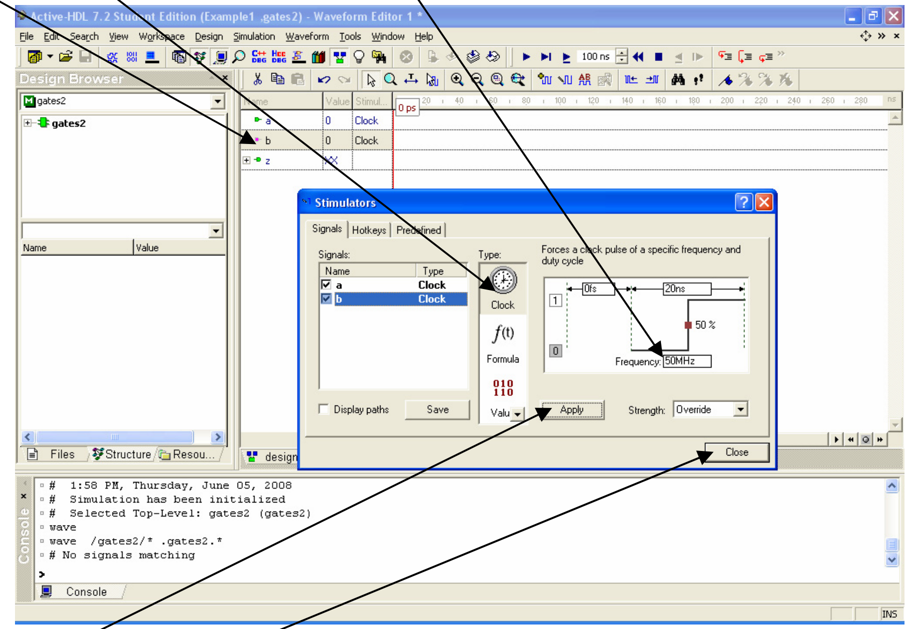
Name	Type
a	Clock

Frequency: 25MHz
Strength: Override

```
# 1:58 PM, Thursday, June 05, 2008
# Simulation has been initialized
# Selected Top-Level: gates2 (gates2)
wave
wave /gates2/* .gates2.*
# No signals matching
>
```

Click *Apply*

Click on *b*, select *Clock* and set Frequency to 50 MHz


Active-HDL 7.2 Student Edition (Example1 .gates2) - Waveform Editor 1 *
Design Browser
Stimulators
Signals
Hotkeys
Predefined
Clock
Formula
Display paths
Save
Apply
Close
Forces a clock pulse of a specific frequency and duty cycle
Console

Name	Type
a	Clock
b	Clock

Frequency: 50MHz
Strength: Override

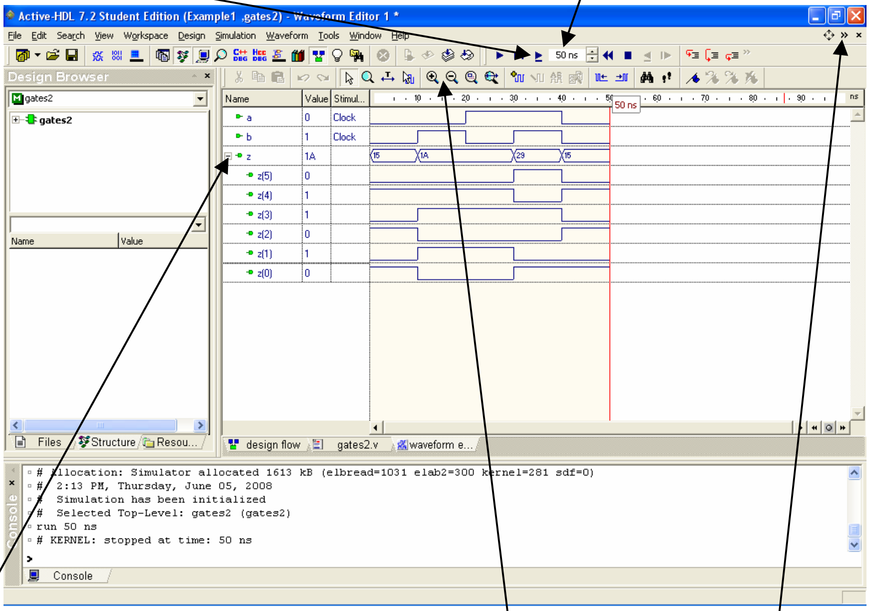
```
# 1:58 PM, Thursday, June 05, 2008
# Simulation has been initialized
# Selected Top-Level: gates2 (gates2)
wave
wave /gates2/* .gates2.*
# No signals matching
>
```

Click *Apply*

Click *Close*

Set simulation time to 50 ns

Click here to run simulation


Active-HDL 7.2 Student Edition (Example1 .gates2) - Waveform Editor 1 *
File Edit Search View Workspace Design Simulation Waveform Tools Window Help
50 ns
Design Browser
gates2
Name
Value
Stimul...
a
0
Clock
b
1
Clock
z
1A
z(5)
z(4)
z(3)
z(2)
z(1)
z(0)
Files
Structure
Resou...
design flow
gates2.v
waveform e...
Console

```
# Allocation: Simulator allocated 1613 kB (elbread=1031 elab2=300 kernel=281 sdf=0)
#  2:13 PM, Thursday, June 05, 2008
#  Simulation has been initialized
#  Selected Top-Level: gates2 (gates2)
run 50 ns
# KERNEL: stopped at time: 50 ns
>
```

Click + sign to show all elements of *z*.

Study the waveforms for various magnifications.

To print out this waveform you can detach it by clicking >> here and then press *Alt Prnt Scrn* to copy it to the clipboard. Then paste it in a *.doc* file and print.

Appendix E

Verilog Quick Reference Guide

Category	Definition	Example

Identifer Names	Can contain any letter, digit, underscore _, or $
	Can not begin with a digit or be a keyword
	Case sensitive
		`q0`
		`Prime_number`
		`lteflg`

Signal Values	0 = logic value 0 1 = logic value 1 z or Z = high impedance x or X = unknown value	

Numbers	d = decimal
	b = binary
	h = hexadecimal
	o = octal
		`35 (default decimal)`
		`4‘b1001`
		`8’a5 = 8’b10100101`

Parameters	Associates an identifer name with a value that can be overridden with the **defparam** statement	`#(`**`parameter`**` N = 8)`

Local parameters	Associates an identifer name with a constant that cannot be directly overridden	**`localparam`**` [1:0] s0 = 2’b00,`
		`s1 = 2’b01, s2 = 2’b10;`

Nets and Variables Types	**wire** (used to connect one logic element to another)
	reg (variables assigned values in **always** block)
	integer (useful for loop control variables)
		`wire`` [3:0] d;`
		`wire`` led;`
		`reg`` [7:0] q;`
		`integer`` k;`

Module	**module** module_name
	[#(parameter_port_list)]
	(port_dir_type_name,{ port_dir_type_name }
	);
	[**wire** declarations]
	[**reg** declarations]
	[**assign** assignments]
	[**always** blocks]
	
	endmodule
		`module`` register`
		`#(`**`parameter`**` N = 8)`
		`(`**`input wire`**` load ,`
		`input wire`` clk ,`
		`input wire`` clr ,`
		`input wire`` [N-1:0] d ,`
		`output reg`` [N-1:0] q`
		`);`
		
		`always`` @(`**`posedge`**` clk `**`or posedge`**` clr)`
		`if``(clr == 1)`
		`q <= 0;`
		`else if``(load)`
		`q <= d;`
		`endmodule`

Logic operators	~ (NOT)
	& (AND)
	| (OR)
	~(&) (NAND)
	~(|) (NOR)
	^ (XOR)
	~^ (XNOR
		`assign`` z = ~y;`
		`assign`` c = a & b;`
		`assign`` z = x | y;`
		`assign`` w = ~(u & v);`
		`assign`` r = ~(s | t);`
		`assign`` z = x ^ y;`
		`assign`` d = a ~^ b;`

Reduction operators	& (AND)
	| (OR)
	~& (NAND)
	~| (NOR)
	^ (XOR)
	~^ (XNOR
		`assign`` c = &a;`
		`assign`` z = |y;`
		`assign`` w = ~&v;`
		`assign`` r = ~|t;`
		`assign`` z = ^y;`
		`assign`` d = ~^b;`

Arithmetic operators	+ (addition)
	- (subtraction)
	* (multiplication)
	/ (division)
	% (mod)
		`count <= count + 1;`
		`q <= q – 1;`

Verilog Quick Reference Guide (cont.)

Relational operators	==, !=, >, <, >=, <=, ===, !==	**assign** lteflg = (a <= b);
		assign eq = (a == b);
		if(clr == 1)

Shift operators	<< (shift left)
	>> (shift right)
		c = a << 3;
		c = a >> 4;

always block	**always** @(<sensitivity list>)
	always @(*)
		always @(*)
		begin
		  s = a ^ b;
		  c = a & b;
		end

if statement	**if**(expression1)
	begin
	  statement;
	end
	else if (expression2)
	begin
	  statement;
	end
	else
	begin
	  statement;
	end
		if(s == 0)
		   y = a;
		else
		   y = b;

case statement	**case**(expression)
	  alternative1: **begin**
	        statement;
	      end
	  alternative2: **begin**
	        statement;
	      end
	  [**default**: **begin**
	        statement;
	      end
	endcase
		case(s)
		   0: y = a;
		   1: y = b;
		   2: y = c;
		   3: y = d;
		   default: y = a;
		endcase

for loop	**for**(initial_index; terminal_index; increment)
	begin
	  statement;
	end
		for(i=2; i<=4; i=i+1)
		      z = z & x[i];

Assignment operator	= (blocking)
	<= (non-blocking)
		z = z & x[i];
		count <= count + 1;

Module instantiation	Module_name instance_name(.port_name(expr)
	  {,.port_name([expr])});
		hex7seg d7R(.d(y),
		            .a_to_g(a_to_g)
		);

Parameter override	**defparam** instance_name.parameter_name = val;	**defparam** Reg.N = 16;