BARREL SHIFTER

A miniproject report
submitted in partial fulfillment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

ELECTRONICS & COMMUNICATION ENGINEERING

Submitted by

P. Sumanth Kumar	**(08R01A04B2)**
P. Kishore Kumar	**(08R01A04B3)**
Y. Anji Reddy	**(08R01A04C7)**

Under The Esteemed Guidance Of

Asst. Professor N.V. Rama Krishna.

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

CMR INSTITUTE OF TECHNOLOGY
(AFFILIATED TO JAWAHERLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD)
(AN ISO 9001:2000 CERTIFIED ISTITUTION)
KANDLAKOYA, MEDCHAL ROAD, HYDERABAD - 501401.

2011 – 2012

CERTIFICATE

CMR INSTITUTE OF TECHNOLOGY

This is to certify that the miniproject work entitled

"BARREL SHIFTER"

is a bonafide work carried out by the student in partial fulfillment of the requirements for the degree of **BACHELOR OF TECHNOLOGY** in **ELECTRONICS & COMMUNICATION ENGINEERING** During the academic year 2011-2012.

P. Sumanth Kumar	**(08R01A04B2)**
P. Kishore Kumar	**(08R01A04B3)**
Y. Anji Reddy	**(08R01A04C7)**

(N.V. Rama Krishna)
Asst. Professor
Internal guide

(Prof. A. Balaji Nehru)
Head of the department
Electronics & Communication Engineering

External examiner

DECLARATION

We hereby declare that the results embodied in this Dissertation entitled "**BARREL SHIFTER**" is carried out by us for the partial fulfillment of the project requirements for the award of degree.

We have not submitted to any other University /Institute for the award of any degree.

P. Sumanth Kumar **(08R01A04B2)**

P. Kishore Kumar **(08R01A04B3)**

Y. Anji Reddy **(08R01A04C7)**

CERTIFICATE

This is to certify that the Dissertation entitled "**BARREL SHIFTER**" is a bonafide work done by Y. Anji Reddy(Roll No: 08R01A04C7), P. Sumanth Kumar (Roll No: 08R01A04B2), P. Kishore Kumar(RollNo:08R01A04B3) final year students of B.Tech (ECE), in partial fulfillment of the requirements for the award of the degree of Bachelor Of Technology in Electronics & Communication Engineering, submitted to the Department of Electronics and Communication Engineering, CMR INSTITUTE OF TECHNOLOGY, Hyderabad. The Project work has been carried out at CMR INSTITUTE OF TECHNOLOGY, Hyderabad under my supervision and guidance.

DATE: **SIGNATURE**

(Head of the department)

External Examiner

ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of any task would be incomplete without a mention of the people, who made it possible and whose guidance and encouragement crown all the efforts with success. i would like to take this opportunity to express my deep sense of gratitude and extend my best wishes to all the people who have guided inspired and motivated me during this project and given me immense pleasure to acknowledge their cooperation.

I offer my sincere thanks to **Mr. N.V. Rama Krishna** project guide for readily responding to the request to do the project at **CMR INSTITUTE OF TECHNOLOGY**, Hyderabad. I highly indebted to him, who was not only shown atmost patience, but fertile in suggestions vigilant in the directions of error and who has infinitley helpful.

I express my deep sense of gratitude **to Asso. Prof. N.V.Rama Krishna** , project internal guide for her help, through provoking discussions invigorating suggestions extended to me with immense care zeal throughout the work. I am also thankful for her interminable help in overcoming all hurdles in flow of my project.

I am thankful to **Prof. A. BALAJI NEHRU**, Head of the department ECE for his constant source of encouragement and inspiration for me to strive hard and give my best to anything undertaken.

No small amount of gratitude would be sufficient to **Dr. M. JANGA REDDY,** principal, CMR institute of technology, for his kind encouragement.

P. Sumanth Kumar **(08R01A04B2)**

P. Kishore Kumar **(08R01A04B3)**

Y. Anji Reddy **(08R01A04C7)**

LIST OF CONTENTS

LIST OF FIGURES

CHAPTER – 1

INTRODUCTION

1.1 Introduction

A barrel shifter is a digital circuit that can shift a data word by a specified number of bits. It can be implemented as a sequence of multiplexers. In this implementation, the output of one MUX is connected to the input of the next MUX in a way that depends on the shift distance. The number of multiplexers required is $n*\log_2(n)$, for an n bit word. Four common word sizes and the number of multiplexers needed are listed below:

- 64-bit — $64 * \log_2(64) = 64 * 6 = 384$
- 32-bit — $32 * \log_2(32) = 32 * 5 = 160$
- 16-bit — $16 * \log_2(16) = 16 * 4 = 64$
- 8-bit — $8 * \log_2(8) = 8 * 3 = 24$

Basically, a barrel shifter works to shift data by incremental stages which avoids extra clocks to the register and reduces the time spent shifting or rotating data (the specified number of bits are moved/shifted/rotated the desired number of bit positions in a single clock cycle). A barrel shifter is commonly used in computer-intensive applications, such as Digital Signal Processing (DSP), and is useful for most applications that shift data left or right - a normal style for C programming code.

Rotation (right) is similar to shifting in that it moves bits to the left. With rotation, however, bits which "fall off" the left side get tacked back on the right side as lower order bits, while in shifting the empty space in the lower order bits after shifting is filled with zeros.

Data shifting is required in many key computer operations from address decoding to computer arithmetic. Full barrel shifters are often on the critical path, which has led most research to be directed toward speed optimizations. With the advent of mobile computing, power has become as important as speed for circuit designs. In this project we present a range of 32-bit barrel shifters that vary at the gate, architecture, and environment levels.

CHAPTER – 2

FUNCTION OF BARREL SHIFTER

Each shifter will be designed as a 16-bit shifter that receives a 16-bit input data value along with a two's compliment encoded shift value, and will produce a 16- bit shifted result. This section will describe the internal design characteristics for each shifter.

2.1 Architecture

There are two common architectural layouts for shifts, array shifter and logarithmic shifters. An array shifter(Fig. 1) decodes the shift value into individual shift bit lines that mesh across all input data values. At each crossing point, a gate will either allow or not allow the input data value to pass to the output line, controlled by a shift bit line. The advantage of this design is that there is only ever one gate between the input data lines and the output data lines, so it is fast. The disadvantages of this design are the requirement for a decoder, and the fact that each input data line sees a load from every shift bit line.

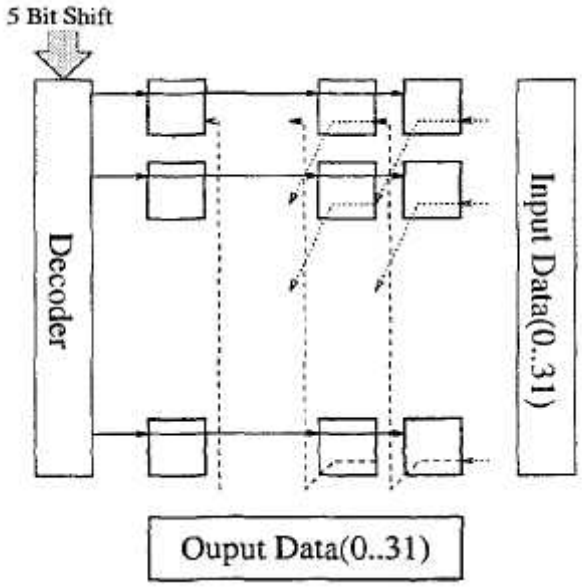


Fig. 1: Structure of an array shifter.

In a logarithmic shifter(Fig. 2*),* the shifter is divided into *loga(n)* stages, where *n* is the input data length. Each bit of the encoded shift value is sent to a different stage of the shifter. Each stage handles a single, powerof- two shift. The input data will be shifted or not shifted by each of the stages in sequence. Five stages would be required when considering **32** bit data. The advantage of a log shifter is that it has small area and does not require a decoder, but the disadvantage is that there are five levels of gates separating the input data from the output data.

2.2 Gate Type

There are two types of gates that are required for these shifters: the array shifter requires switches that will either propagate or not propagate an input data bit, and the log shifter requires 2-to-1 muxes to propagate either a shifted or a non-shifted bit.

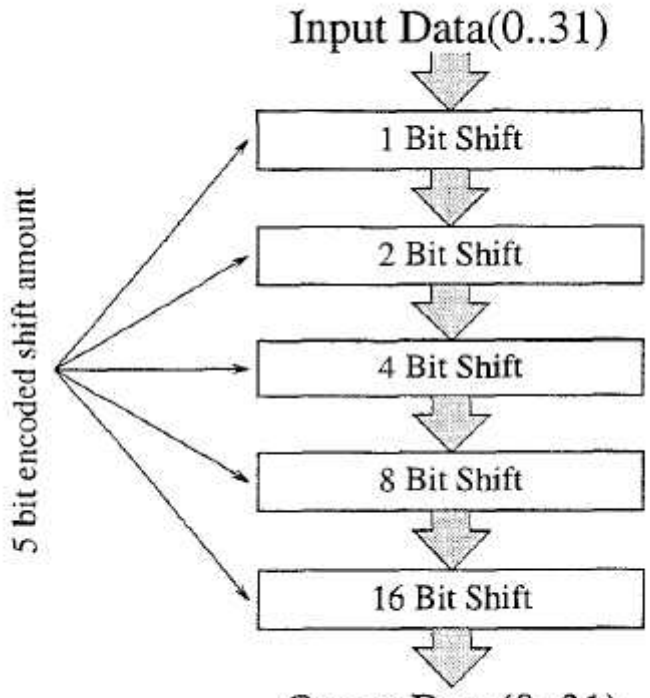


Fig. 2: Structure of a log shifter.

In this project we will consider two types of CMOS switches: (1) ntype pass transistor switch; and **(2)** a full transmission gate switch; and we will consider four types of mux designs: (1) n-type pass transistor mux; (2) full transmission mux;

(3) a static CMOS mux; and **(4)** a dynamic logic mux. The pass transistor and transmission gates are simple and fast, but will require occasional buffering to strengthen the signals in the log shifters. The static and dynamic gates are self buffering so no additional buffers are needed, but contain more transistors. The dynamic gate design is the only type that requires a clock signal for a precharge stage. Figure **3** shows schematics for each gate design.

2.3 Right Rotation

Right rotation is similar to right shifting, except that additional hardware is required to determine which values get shifted into the upper bits of the output. We consider three options: (1) a wrap around least significant bit for right rotation; (2) a sign bit for arithmetic right shifting; and (3) *GND* for signed magnitude right shifting. **A** 3-to-1 control mux was added to each wrap around bit line. This mux allows either the rotation wrap-around bit, the sign bit, or *GND* to be selected.

2.4 Left Rotation

Left rotations can be accomplished by rotating right 32 - *Rotatereft* bits. *Rotate,i,ht* can be calculated by taking the two's-compliment of the *Rotateleft* value, which requires inverting all the *Rotateleft* bits and adding one. The inversion of the *Rotateleft* bits can be accomplished by adding muxes that pass either the shift bit or its inverse. The addition of one to *Rotateleft* can be accomplished in two ways: (1) include a 5 bit incrementor; or (2) add an additional one-bit shift stage.

2.5 Left Shifting

Left shifting can be performed by right shifting 16 – *Shift left* bits, and including an additional row of pull down gates that mask out the lower *n* bits of the *n* bit left shift. **A** second method is to initially reverse the input data bits and perform a right shift of length *Shiftleft,* and finally reverse the output bits.

2.6 Applications

- Digital Signal Processing
- Array Processing
- Graphics
- Database Addressing
- High Speed Arithmetic Processors

Role of barrel shifter in DSP

In digital signal processing (DSP) chips used in processors, a very large number of mathematical operations, including multiplications and additions, need to be performed at high speed. To accomplish this, high speed data path components are required on the DSP chip.

A DSP chip that performs arithmetic operations is a math processor. This processor is coupled to a system bus to receive and send data over the bus to other components in the computer system. An ALU (Arithmetic Logic Unit) is one of the main components that performs arithmetic operations in the math processor within the DSP chip. In order to improve the performance of the ALU, data is often manipulated prior to its introduction into the ALU. For example, from the system bus, incoming data can be normalized by a right shifter that scales down the magnitude of the number represented by the incoming data to make the number more manageable for later operations performed by the ALU. The ALU will then perform the required operation, such as addition, logical AND/OR and EX-OR functions, on the shifted data.

The result of the ALU operation is provided to a left shift device which shifts the ALU result to the left. This has the effect of scaling the number by a proper amount to restore normalization. The re-normalized result of the ALU operation is then placed on the system bus.

One type of shifter used in digital signal processing applications is a barrel shifter that will shift a plurality of bits in a single clock cycle. Barrel shifters are particularly advantageous in high performance applications in comparison to serial shifters which would require N clock cycles to shift a number by N bits. A barrel shifter forms a high-speed data path component that is very useful in high-performance applications.

A design goal of a DSP chip designer is to provide the requisite high performance while reducing its power consumption. Power consumption is problematical in barrel shifters since data passes through the barrel shifter even when the barrel shifter is not being used to shift the data, as is often the case. In addition to a loss of power, sending data which will not be shifted through the barrel shifter produces added delay.

There is a need for a low-power, high-performance barrel shifter that will shift a plurality of bits in one clock cycle when desired, but will also pass data through the barrel shifter without delay or loss of power when the data is not to be shifted.

This and other needs are met by the present invention which provides an arrangement for performing arithmetic operations in a processor. This arrangement includes an arithmetic logic unit and a barrel shifter. The arithmetic logic unit (ALU) performs arithmetic operations on data provided as input to the ALU. The ALU has an input for receiving the data and an ALU output at which M-bit results of the arithmetic operations on the data are produced. The barrel shifter includes a barrel shifter input connected to the ALU output, and a shift circuit connected to the barrel shifter input. The shift circuit selectively shifts the M-bit results and produces shifted M-bit results at a shift circuit output.

A barrel shifter output is coupled to the shift circuit output. Shifted M-bit results are produced at the barrel shifter output when the shift circuit is selected to shift the M-bit results.

Unshifted M-bit results are produced at the barrel shifter output when the shift circuit is selected not to shift the M-bit results. The barrel shifter input includes an isolation circuit that isolates the shift circuit from the ALU output when the shift circuit is selected not to shift the M-bit results.

The isolation circuit of the present invention reduces the consumption of power since the shift circuit is isolated when no shifting is required. The highly capacitive nodes in the shift circuit are therefore not switched when the shift circuit is not used to shift the results from the ALU.

In certain preferred embodiments of the invention, the barrel shifter input includes tristate buffers that receive as input the M-bit results from the ALU. The outputs of the tristate buffers are connected to the input of the shift circuit. When the tristate buffers are enabled, and a shift by the

shift circuit is desired, the tristate buffers pass the M-bit results to be shifted into the shift circuit. When the M-bit results are not to be shifted, however, a signal causes the tristate buffers' outputs to float. This effectively isolates the shift circuit from the ALU output when the shift circuit is not to shift the M-bit results. Power consumption by the shift circuit is reduced since the internal signal nodes with high capacitance are prevented from being switched due to the floated outputs of the tristate buffers.

An especially low-power embodiment of the present invention provides an isolation circuit that includes an input multiplexer that has a plurality of tristate buffers. A first one of these tristate buffers has an input coupled to receive all M-bits of the M-bit results from the ALU output, and an output at which the M-bits of the M-bit results are produced as a function of the shift control signal, this output being coupled to the shift circuit.

The input multiplexer also has a second tristate buffer having an input coupled to receive the (M-P) most significant bits of the M-bit results, and P bits set at a logical zero, where P is less than M, to form a P-left-shifted-bit results.

By providing a multiplexer at the input, which chooses between a zero-shifted M-bit result to be input to the shift circuit, or an M-bit result that has already been shifted by P bits to be input to the shift circuit, the number of multiplexers used in the shift circuit can be reduced by one-half thus reducing the capacitance of the switch circuit by approximately half. This reduces the power consumption by approximately one-half when the shift circuit is switched. Another feature provided by certain preferred embodiments of the present invention is the integration of the output multiplexer of the barrel shifter within the shift circuit. This integration reduces the delay from the input to the output.

The foregoing and other objects, features, aspects and advantages of the present invention will become more apparent from the following detailed description of the present invention when taken in conjunction with the accompanying drawings.

Role of barrel shifter in array processing

This invention relates to a system for storing and processing an array of data-elements, such as pixel data.

In particular, the invention is concerned with such a system which comprises a memory having a plurality of memory locations each having a capacity of a predetermined number B (e.g. 32) bits and a processing means for processing data elements and operable to read the data-elements from and/or write the data elements to the memory. Although the memory and processing means are capable of dealing with the predetermined number B of bits (e.g. 32), in some applications, the data-elements may have a lesser resolution (e.g. 16 or 8 bits). In such cases, it would be possible to use only 16 or 8 bits of the 32 bits available for each data-element. However, the memory would not then be used to its full capacity. Also, in a demand-paged dual memory system which pages are swapped from a paging memory into the first-mentioned memory, pages of data-elements would need to be swapped more often than is necessary.

It may therefore be considered expedient to split the whole memory into two for 16-bit data, or four for 8-bit data, and thus overlay whole sub-arrays of the data-elements one on top of another. This would make available the whole capacity of the memory, but would suffer from the disadvantage that severe complications would arise when swapping, for example, just one page of 16-bit or 8-bit data between the memory and paging memory, because it would be necessary to select only half or a quarter of the stored data at each memory location for transfer from the memory to the paging memory, and it would be necessary to mask off half or three-quarters of the memory when transferring a page of data from the paging memory to the first memory.

In order to overcome this problem, in accordance with the invention, the processing means is operable in a mode for processing data-elements having a predetermined number b (e.g. 16) bits not greater than half of said predetermined number B and being operable to read the data-elements from and/or write the data elements to different bit levels (e.g. L(0), L(1)) of the memory locations so that a plurality of data-elements can be stored at such a memory location and so that at no memory location is there stored data-elements from more than one page.

In one embodiment, data elements which are adjacent in at least one direction in the data-array are stored at different bit levels in the memory. However, in a preferred embodiment, the data-elements are stored in the memory in aligned groups of N (e.g. 16) data-elements, and data elements of groups which are adjacent in at least one direction in the data-array are stored at different bit levels in the memory. In this case, in one arrangement, each memory location is divided into two, that is to say, the number b (e.g. 16) of bits of a data-element is half of the number B (e.g. 32) of bits of a memory location, and the data-elements are arranged in patches each of two groups with the data-elements of the two groups being stored at two respective different bit levels (e.g. bits 0-15, bits 16-31). In another arrangement, each memory location is divided into four, that is to say the number b (e.g. 8) of bits of a data-element is a quarter of the number B (e.g. 32) of bits of a memory location, and the data-elements are arranged in patches of four groups with the pixels of the four groups being stored at four respective different bit levels (e.g. bits 0-7, bits 8-15) Preferably, the system is operable in at least two modes selected from the divided-into-two mode, the divided-into-four mode, and a mode in which all B of the bits at a memory location are used to stored each data-element. In this case, the memory may be addressed by an address having bits whose significance varies in accordance with the selected mode of operation, and the system may further comprise a funnel shifter which receives the address bits whose significance can change and a mode selection signal and which outputs address bits appropriate to the selected mode and a level signal indicative of the bit level in the memory of the data-element to be accessed.

During reading of the memory the processing means may be operable to read all of the data at a memory location, the system further comprising means for supplying a shift signal dependent on the level signal to the processing means and the processing means may be operable to bit-shift the read data by an amount dependent on the shift signal to that the data-element to be processed occupies predetermined bit positions.

During writing to the memory, in at least said first-mentioned mode the processing means is preferably operable to duplicate the bits to be written at the different levels and the system may comprise means for supplying a partial write-enable signal dependent on the level signal for controlling the memory so that the data-element is written only to the appropriate bit levels of the memory.

In the preferred embodiment, the processing means comprises a plural number N (e.g. 16) of processors equal in number N to the number of data-elements in a group, where the processors are capable of accessing in parallel all of the data-elements of a group. In this case upon reading of a group which is misaligned with respect to an aligned group, the shift signal supplying means is preferably operable to supply to the processors a respective shift signal for each data-element in the misaligned group which is dependent upon the position of the data-element in the group and the misalignment of the group.

Also, upon writing of a group which is misaligned with respect to an aligned group, the partial write-enable signal supplying means is preferably operable to supply to the memory a respective partial write-enable signal for each data-element in the misaligned group which is dependent on the position of the data-element in the group and the misalignment of the group.

CHAPTER – 3

AN INTRODUCTION TO XILINX 9.1i

The ISE 9.1i provides Xilinx PLD designers with the basic design process using ISE 9.1i. In this chapter you will understande of how to create, verify, and implement a design.

This chapter contains the following sections:

- "Getting Started"
- "Create a New Project"
- "Create an HDL Source"
- "Design Simulation"
- "Create Timing Constraints"
- "Implement Design and Verify Constraints"
- "Reimplement Design and Verify Pin Locations"
- "Download Design to the Spartan™-3 Demo Board"

3.1 Getting Started

Software Requirements:- ISE 9.1i

Hardware Requirements:- Spartan-3 Startup Kit, containing the Spartan-3 Startup Kit Demo Board.

Starting the ISE Software

To start ISE, double-click the desktop icon,

or start ISE from the Start menu by selecting:

- **Start**
- **All Programs**
- **Xilinx ISE 9.1i**
- **Project Navigator**

Note: Your start-up path is set during the installation process and may differ from the one above.Accessing Help

At any time during the tutorial, you can access online help for additional information about the ISE software and related tools.

To open Help, do either of the following:

- Press **F1** to view Help for the specific tool or function that you have selected orhighlighted.
- Launch the **ISE Help Contents** from the Help menu. It contains information about creating and maintaining your complete design flow in ISE.

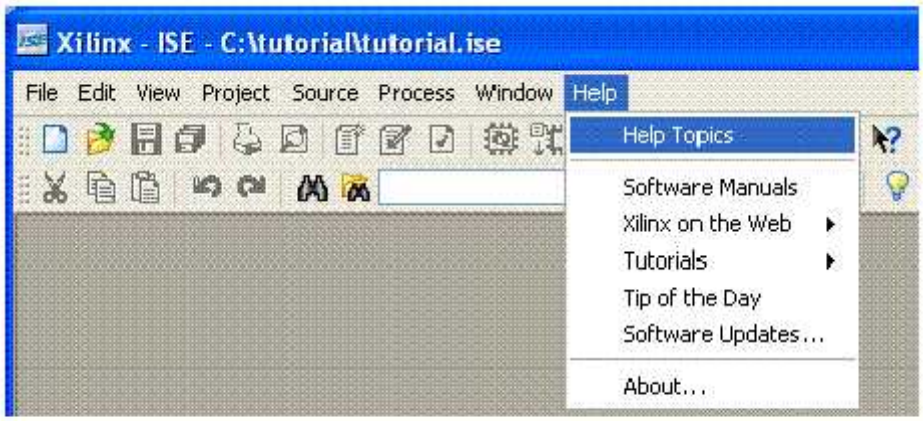


Figure 1: **ISE Help Topics**

3.2 Create a New Project

Create a new ISE project which will target the FPGA device on the Spartan-3 Startup Kit demo board.

To create a new project:

1. Select **File**
New Project... The New Project Wizard appears.

2. Type **tutorial** in the Project Name field.

3. Enter or browse to a location (directory path) for the new project. A tutorial subdirectory is created automatically.

4. Verify that **HDL** is selected from the Top-Level Source Type list.

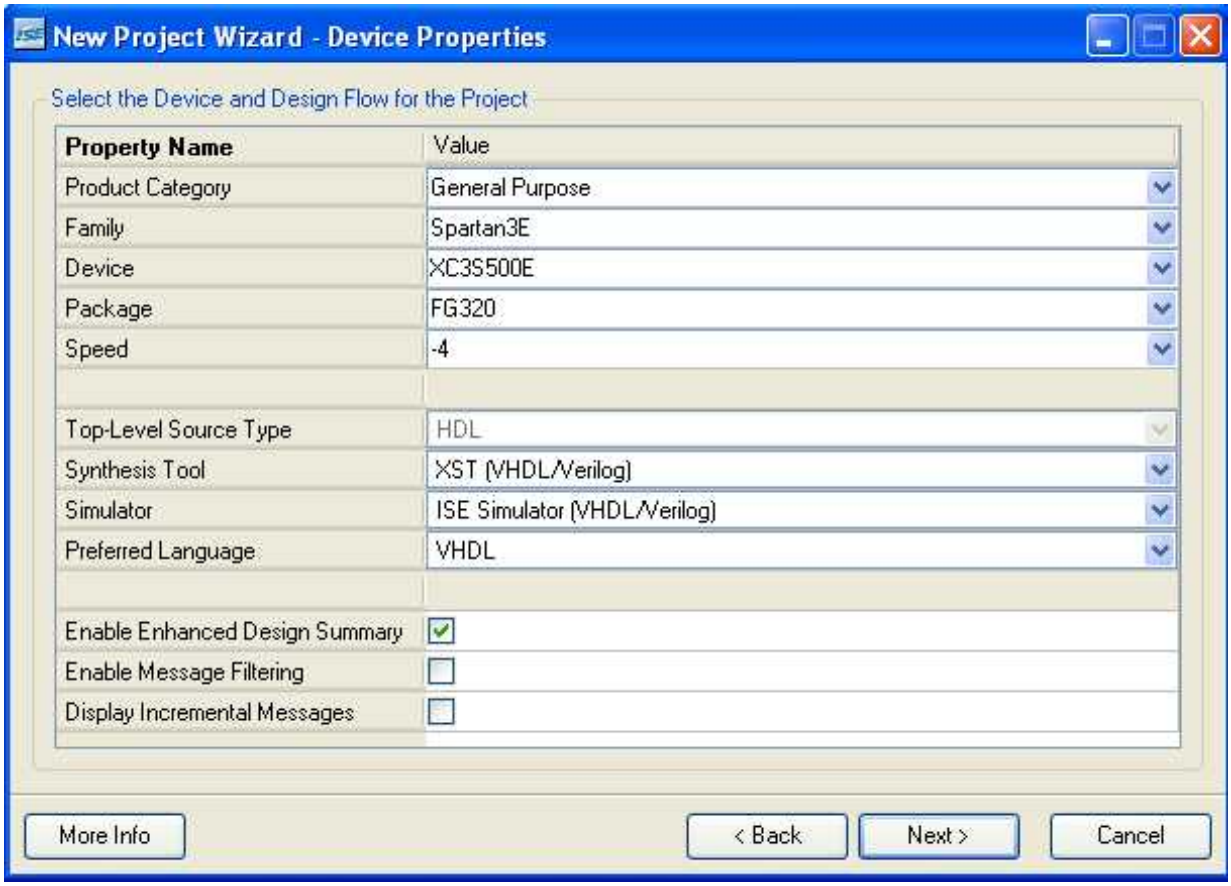
5. Click **Next** to move to the device properties page

6. Fill in the properties in the table as shown below:

- Product Category: **All**
- Family: **Spartan3**
- Device: **XC3S200**
- Package: **FT256**
- Speed Grade: **-4**
- Top-Level Source Type: **HDL**
- Synthesis Tool: **XST (VHDL/Verilog)**
- Simulator: **ISE Simulator (VHDL/Verilog)**
- Preferred Language: **VHDL** (or **Verilog**)
- Verify that **Enable Enhanced Design Summary** is selected.

Leave the default values in the remaining fields.

When the table is complete, your project properties will look like the following:



7. Click **Next** to proceed to the Create New Source window in the New Project Wizard. At the end of the next section, your new project will be complete.

3.3 Create an HDL Source

In this section, you will create the top-level HDL file for your design. Determine the language that you wish to use for the tutorial. Then, continue either to the "Creating a VHDL Source" section below, or skip to the "Creating a Verilog Source" section.

Creating a VHDL Source

Create a VHDL source file for the project as follows:

1. Click the **New Source** button in the New Project Wizard.
2. Select **VHDL Module** as the source type.
3. Type in the file name **counter**.
4. Verify that the **Add to project** checkbox is selected.
5. Click **Next**.
6. Declare the ports for the counter design by filling in the port information as shown below:

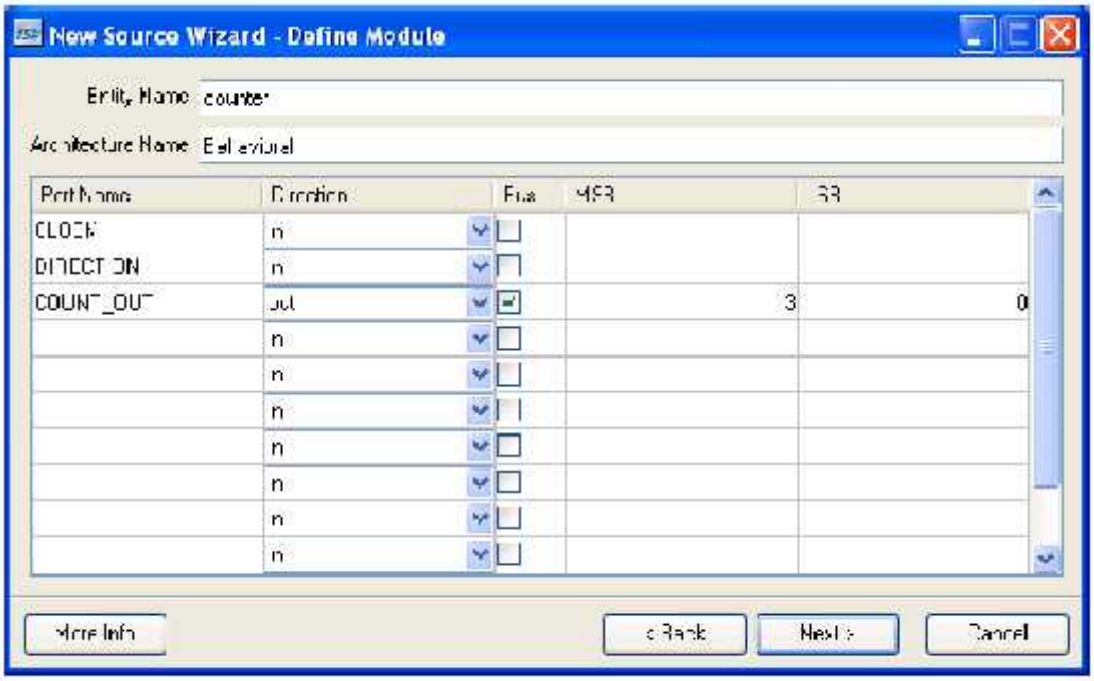


Figure 3: **Define Module**

7. Click **Next**, then **Finish** in the New Source Wizard - Summary dialog box to complete the new source file template.

8. Click **Next**, then **Next**, then **Finish**.

The source file containing the entity/architecture pair displays in the workspace, and the counter displays in the Source tab, as shown below:

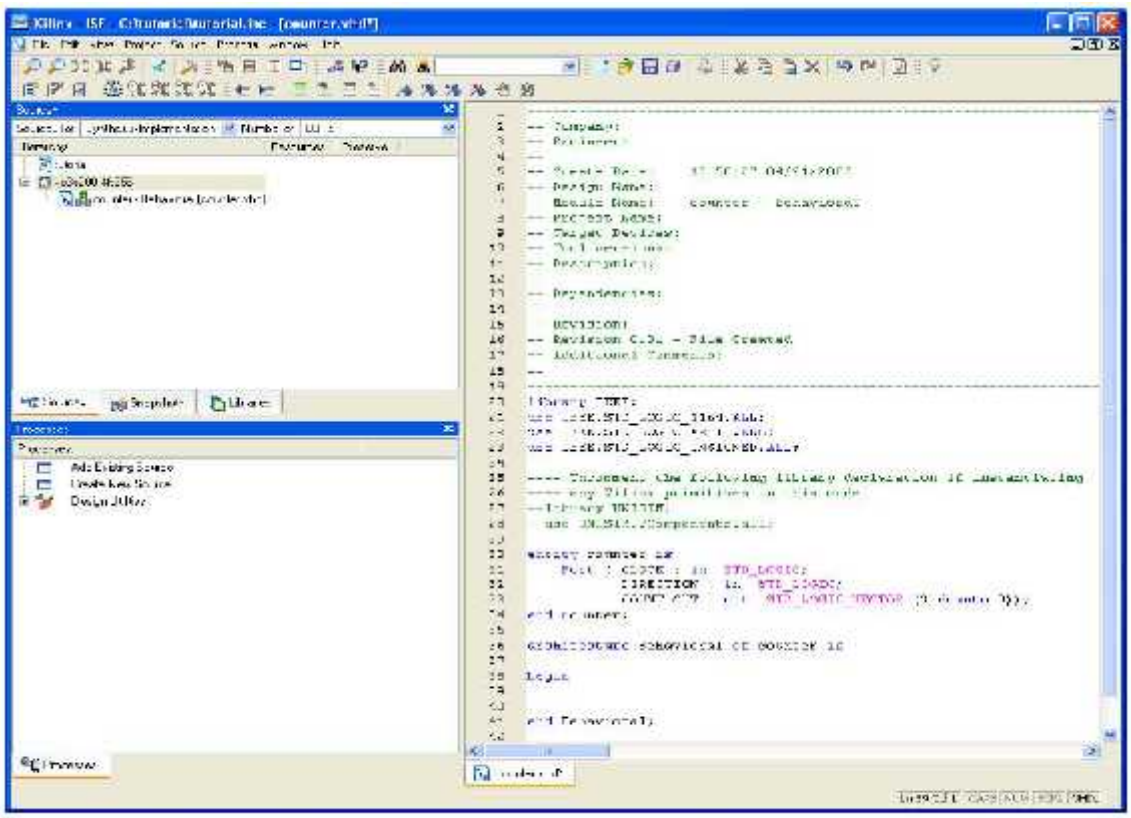


Figure 4: **New Project in ISE**

Using Language Templates (VHDL)

The next step in creating the new source is to add the behavioral description for the counter. To do this you will use a simple counter code example from the ISE Language templates and customize it for the counter design.

1. Place the cursor just below the begin statement within the counter architecture.

2. Open the Language Templates by selecting **Edit**

- **Language Templates…**

Note: You can tile the Language Templates and the counter file by selecting **Window**

- **Tile Vertically** to make them both visible.

3. Using the "+" symbol, browse to the following code example:

VHDL

- **Synthesis Constructs**
- **Coding Examples**
- **Counters**
- **Binary**
- **Up/Down Counters**
- **Simple Counter**

1. With Simple Counter selected, select **Edit**
2. **Use in File**, or select the **Use Template** in **File** toolbar button. This step copies the template into the counter source file.

4. Close the Language Templates.

Final Editing of the VHDL Source:

1. Add the following signal declaration to handle the feedback of the counter output below the architecture declaration and above the first begin statement:

```
signal count_int : std_logic_vector(3 downto 0) :"0000";
```

2. Customize the source file for the counter design by replacing the port and signal name placeholders with the actual ones as follows:

   - replace all occurrences of <clock> with CLOCK
   - replace all occurrences of <count_direction> with DIRECTION
   - replace all occurrences of <count> with count_int

3. Add the following line below the end process; statement:

```
COUNT_OUT <= count_int;
```

4. Save the file by selecting **File Save**.

When you are finished, the counter source file will look like the following:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Uncomment the following library declaration if instantiating
-- any Xilinx primitive in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
entity counter is
Port ( CLOCK : in STD_LOGIC;
DIRECTION : in STD_LOGIC;
COUNT_OUT : out STD_LOGIC_VECTOR (3 downto 0));
end counter;
```

```
architecture Behavioral of counter is
signal count_int : std_logic_vector(3 downto 0) := "0000";
begin
process (CLOCK)
begin
if CLOCK='1' and CLOCK'event then
if DIRECTION='1' then
count_int <= count_int + 1;
else
count_int <= count_int - 1;
end if;
end if;
end process;
COUNT_OUT <= count_int;
end Behavioral;
```

You have now created the VHDL source for the tutorial project. Skip past the Verilog sections below, and proceed to the "Checking the Syntax of the New Counter Module" section.

Checking the Syntax of the New Counter Module

When the source files are complete, check the syntax of the design to find errors and types.

1. Verify that **Synthesis/Implementation** is selected from the drop-down list in the Sources window.
2. Select the **counter** design source in the Sources window to display the related processes in the Processes window.
3. Click the "**+**" next to the Synthesize-XST process to expand the process group.
4. Double-click the **Check Syntax** process.

Note: You must correct any errors found in your source files. You can check for errors in the Console tab of the Transcript window. If you continue without valid syntax, you will not be able to simulate or synthesize your design.

5. Close the HDL file.

3.4 Design Simulation

Verifying Functionality using Behavioral Simulation

Create a test bench waveform containing input stimulus you can use to verify the functionality of the counter module. The test bench waveform is a graphical view of a test bench.

Create the test bench waveform as follows:

1. Select the **counter** HDL file in the Sources window.
2. Create a new test bench source by selecting **Project New Source**.
3. In the New Source Wizard, select **Test Bench WaveForm** as the source type, and type **counter_tbw** in the File Name field.
4. Click **Next**.
5. The Associated Source page shows that you are associating the test bench waveform with the source file counter. Click **Next**.
6. The Summary page shows that the source will be added to the project, and it displays the source directory, type and name. Click **Finish**.
7. You need to set the clock frequency, setup time and output delay times in the Initialize Timing dialog box before the test bench waveform editing window opens.

The requirements for this design are the following:

- The counter must operate correctly with an input clock frequency = 25 MHz.
- The DIRECTION input will be valid 10 ns before the rising edge of CLOCK.
- The output (COUNT_OUT) must be valid 10 ns after the rising edge of CLOCK.

The design requirements correspond with the values below.

Fill in the fields in the Initialize Timing dialog box with the following information:

- Clock High Time: **20** ns.
- Clock Low Time: **20** ns.
- Input Setup Time: **10** ns.
- Output Valid Delay: **10** ns.
- Offset: **0** ns.
- Global Signals: **GSR (FPGA)**

Note: When GSR(FPGA) is enabled, 100 ns. is added to the Offset value automatically.

- Initial Length of Test Bench: **1500** ns.

Leave the default values in the remaining fields.

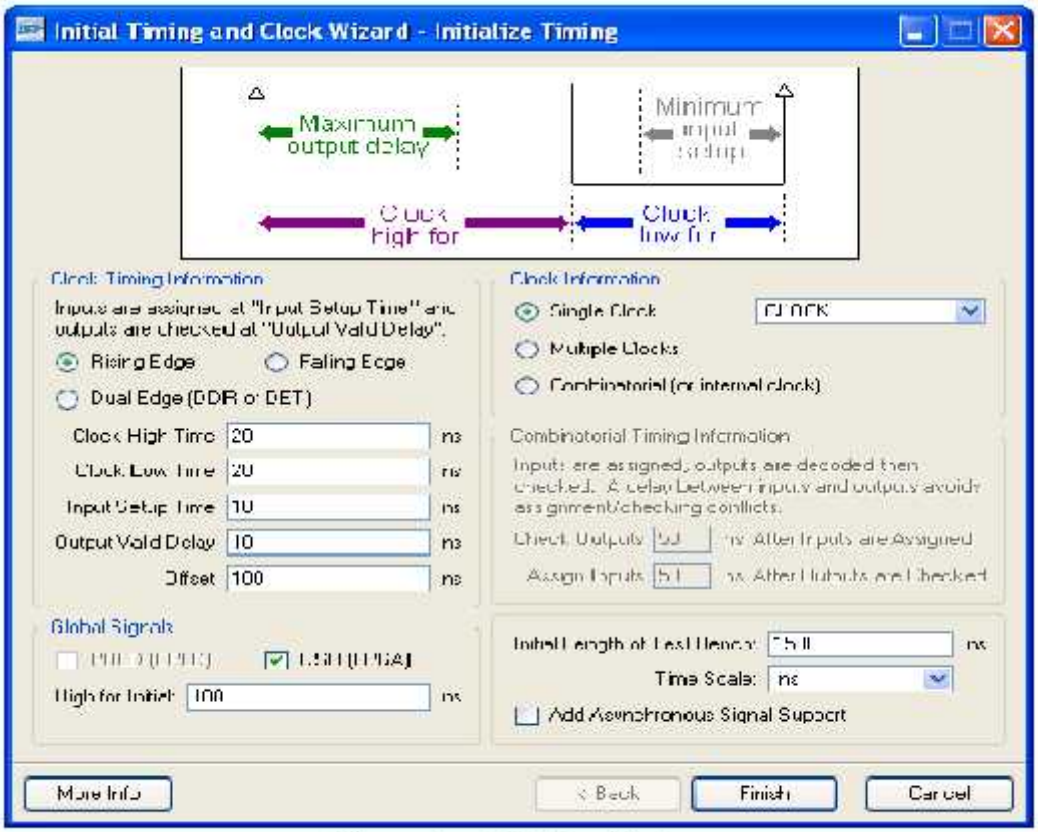


Figure 7: **Initialize Timing**

8. Click **Finish** to complete the timing initialization.

9. The blue shaded areas that precede the rising edge of the CLOCK correspond to the Input Setup Time in the Initialize Timing dialog box. Toggle the DIRECTION port to define the input stimulus for the counter design as follows:

- Click on the blue cell at approximately the **300** ns to assert DIRECTION high so that the counter will count up.
- Click on the blue cell at approximately the **900** ns to assert DIRECTION low so that the counter will count down.

Note: For more accurate alignment, you can use the **Zoom In** and **Zoom Out** toolbar buttons.

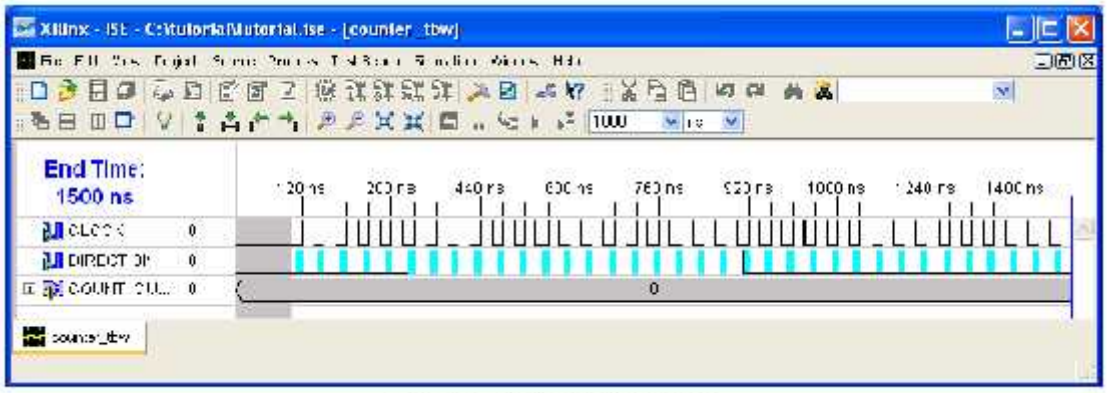


Figure 8: **Test Bench Waveform**

10. Save the waveform.

11. In the Sources window, select the **Behavioral Simulation** view to see that the test bench waveform file is automatically added to your project.

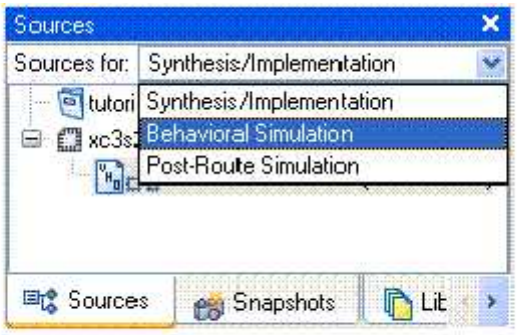


Figure 9: **Behavior Simulation Selection**

12. Close the test bench waveform.

3.5 Simulating Design Functionality

Verify that the counter design functions as you expect by performing behavior simulation as follows:

1. Verify that **Behavioral Simulation** and **counter_tbw** are selected in the Sources window.

2. In the **Processes** tab, click the "+" to expand the Xilinx ISE Simulator process and double-click the **Simulate Behavioral Model** process. The ISE Simulator opens and runs the simulation to the end of the test bench.
3. To view your simulation results, select the **Simulation** tab and zoom in on the transitions.

The simulation waveform results will look like the following:

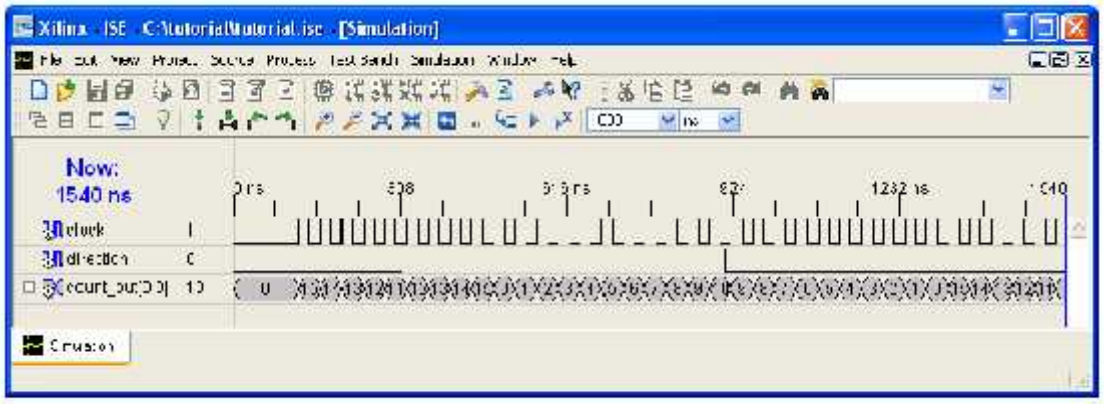


Figure 19: **Simulation Results**

Note: You can ignore any rows that start with **TX**.

4. Verify that the counter is counting up and down as expected.
5. Close the simulation view. If you are prompted with the following message, "You have an active simulation open. Are you sure you want to close it?", click **Yes** to continue. You have now completed simulation of your design using the ISE Simulator.

3.6 Create Timing Constraints

Specify the timing between the FPGA and its surrounding logic as well as the frequency the design must operate at internal to the FPGA. The timing is specified by entering constraints that guide the placement and routing of the design. It is recommended that you enter global constraints. The clock period constraint specifies the clock frequency at which your design must operate inside the FPGA. The offset constraints specify when to expect valid data at the FPGA inputs and when valid data will be available at the FPGA outputs.

3.6.1 Entering Timing Constraints

To constrain the design do the following:

1. Select **Synthesis/Implementation** from the drop-down list in the Sources window.

2. Select the **counter** HDL source file.

3. Click the "+" sign next to the User Constraints processes group, and double-click the

Create Timing Constraints process.ISE runs the Synthesis and Translate steps and automatically creates a User Constraints File (UCF). You will be prompted with the following message:

Figure 11: **Prompt to Add UCF File to Project**

4. Click **Yes** to add the UCF file to your project. The counter.ucf file is added to your project and is visible in the Sources window. The Xilinx constraints Editor opens automatically.

Note: You can also create a UCF file for your project by selecting **Project Create New Source**.

In the next step, enter values in the fields associated with CLOCK in the Constraints Editor Global tab.

5. Select **CLOCK** in the Clock Net Name field, then select the **Period** toolbar button or double-click the empty Period field to display the Clock Period dialog box.

6. Enter **40** ns in the Time field.

3.6.2 Implement Design and Verify Constraints

Implement the design and verify that it meets the timing constraints specified in the previous section.

Implementing the Design

1. Select the **counter** source file in the Sources window.
2. Open the Design Summary by double-clicking the **View Design Summary** process In the Processes tab.
3. Double-click the **Implement Design** process in the Processes tab.
4. Notice that after Implementation is complete, the Implementation processes have a green check mark next to them indicating that they completed successfully without Errors or Warnings.

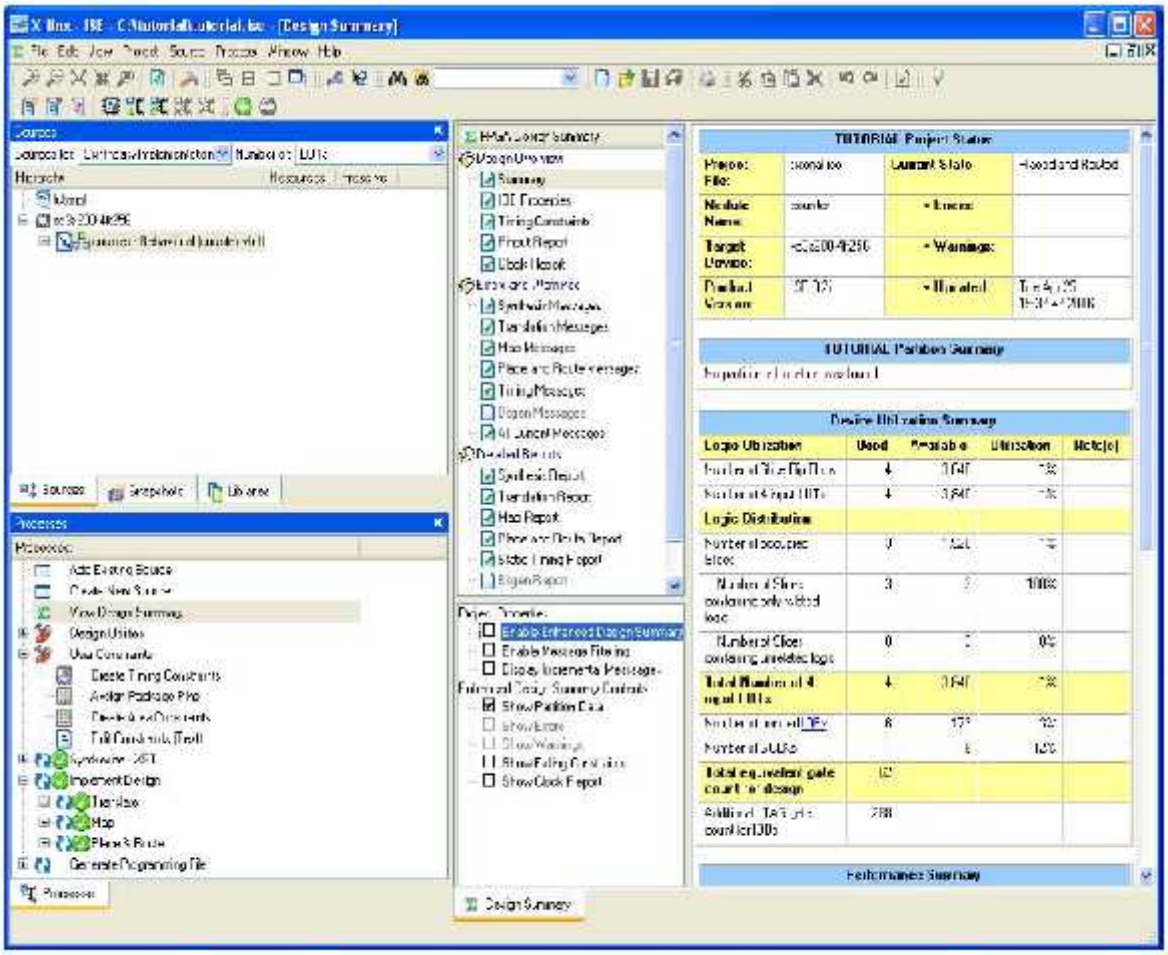


Figure 16: **Post Implementation Design Summary**

5. Locate the **Performance Summary** table near the bottom of the design Summary.
6. Click the **All Constraints Met** link in the Timing Constraints field to view the Timing Constraints report. Verify that the design meets the specified timing requirements.

Assigning Pin Location Constraints

Specify the pin locations for the ports of the design so that they are connected correctly on the Spartan-3 Startup Kit demo board.

To constrain the design ports to package pins, do the following:

1. Verify that **counter** is selected in the Sources window.
2. Double-click the **Assign Package Pins** process found in the User constraints process group. The Xilinx Pinout and Area Constraints Editor (PACE) opens.
3. Select the **Package View** tab.
4. In the Design Object List window, enter a pin location for each pin in the **Loc** column using the following information:
   - CLOCK input port connects to FPGA pin **T9** (GCK0 signal on board)
   - COUNT_OUT<0> output port connects to FPGA pin **K12** (LD0 signal on board)
   - COUNT_OUT<1> output port connects to FPGA pin **P14** (LD1 signal on board)
   - COUNT_OUT<2> output port connects to FPGA pin **L12** (LD2 signal on board)
   - COUNT_OUT<3> output port connects to FPGA pin **N14** (LD3 signal on board)
   - DIRECTION input port connects to FPGA pin **K13** (SW7 signal on board)

Notice that the assigned pin locations are shown in blue:

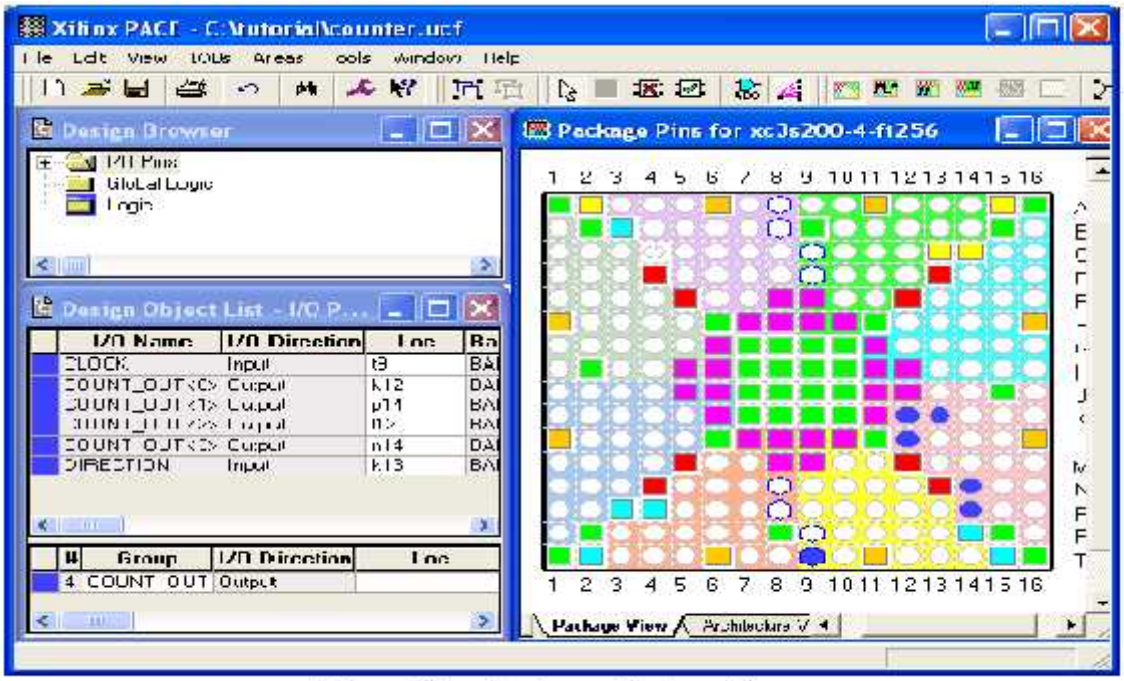


Figure 18: **Package Pin Locations**

5. Select **File**

   Save. You are prompted to select the bus delimiter type based on the synthesis tool you are using. Select **XST Default <>** and click **OK**.

6. Close PACE.

Notice that the Implement Design processes have an orange question mark next to them, indicating they are out-of-date with one or more of the design files. This is because the UCF file has been modified.

3.6.3 Download Design to the Spartan™-3 Demo Board

This is the last step in the design verification process. This section provides simple instructions for downloading the counter design to the Spartan-3 Starter Kit demo board.

1. Connect the 5V DC power cable to the power input on the demo board (J4).
2. Connect the download cable between the PC and demo board (J7).
3. Select **Synthesis/Implementation** from the drop-down list in the Sources window.
4. Select **counter** in the Sources window.
5. In the Processes window, click the "+" sign to expand the **Generate Programming File** processes.
6. Double-click the **Configure Device (iMPACT)** process.
7. The Xilinx WebTalk Dialog box may open during this process. Click **Decline.**
8. Select **Disable the collection of device usage statistics for this project only** and click **OK**.

   iMPACT opens and the Configure Devices dialog box is displayed.

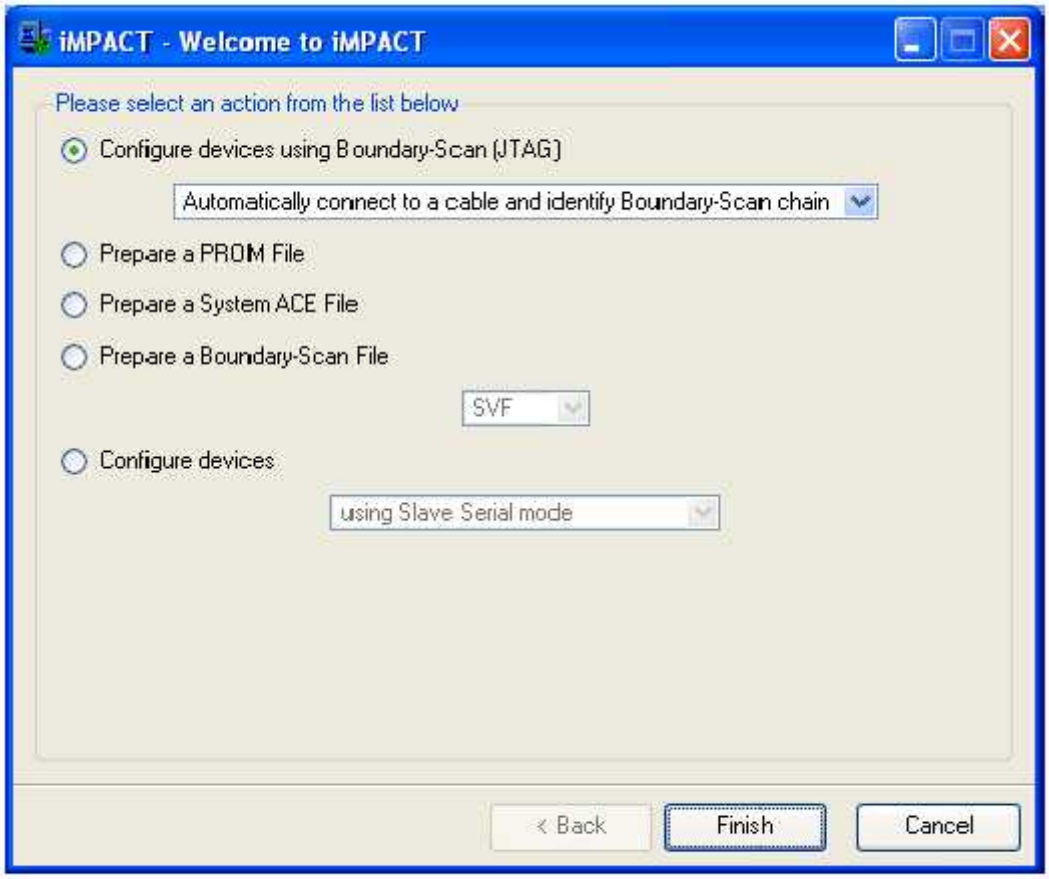
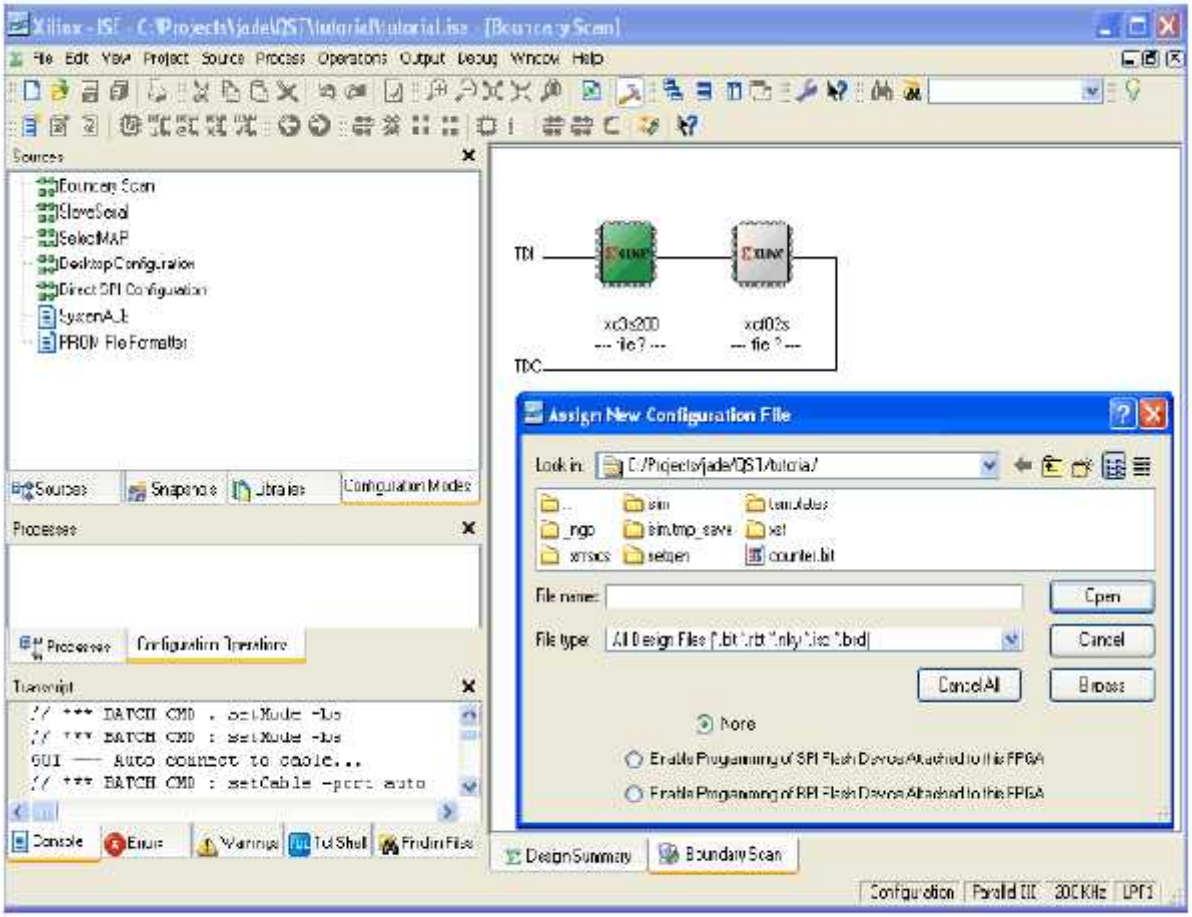


Figure 21: **iMPACT Welcome Dialog Box**

9. In the Welcome dialog box, select **Configure devices using Boundary-Scan (JTAG)**.
10. Verify that **Automatically connect to a cable and identify Boundary-Scan chain** is selected.
11. Click **Finish**.
12. If you get a message saying that there are two devices found, click **OK** to continue. The devices connected to the JTAG chain on the board will be detected and displayed in the iMPACT window.
13. The **Assign New Configuration File** dialog box appears. To assign a configuration file to the xc3s200 device in the JTAG chain, select the counter. bit file and click **Open**.



14. If you get a Warning message, click **OK**.
15. Select **Bypass** to skip any remaining devices.
16. Right-click on the xc3s200 device image, and select **Program...** The **Programming Properties** dialog box opens.
17. Click **OK** to program the device.

When programming is complete, the Program Succeeded message is displayed.

On the board, LEDs 0, 1, 2, and 3 are lit, indicating that the counter is running.

18. Close iMPACT without saving.

Chapter - 4

An Introduction to FPGA

4.1 Introduction

As described in **Architectural Overview**, the Spartan™-3E FPGA architecture consists of five fundamental functional elements:

- **Input/Output Blocks (IOBs)**
- **Configurable Logic Block (CLB) and Slice Resources**
- **Block RAM**
- **Dedicated Multipliers**
- **Digital Clock Managers (DCMs)**

The following sections provide detailed information on each of these functions. In addition, this section also describes the following functions:

- **Clocking Infrastructure**
- **Interconnect**
- **Configuration**
- **Powering Spartan-3E FPGAs**

4.2 Input/output Blocks (IOBs)

For additional information, refer to the "*Using I/O Resources*" chapter in UG331.

IOB Overview

The Input/Output Block (IOB) provides a programmable, unidirectional or bidirectional interface between a package pin and the FPGA's internal logic. The IOB is similar to that of the Spartan-3 family with the following differences:

- Input-only blocks are added
- Programmable input delays are added to all blocks
- DDR flip-flops can be shared between adjacent IOBs

The unidirectional input-only block has a subset of the full IOB capabilities. Thus there are no connections or logic for an output path. The following paragraphs assume that any reference to output functionality does not apply to the input-only blocks. The number of input-only blocks varies with device size, but is never more than 25% of the total IOB count.

Figure 1 is a simplified diagram of the IOB's internal structure. There are three main signal paths within the IOB: the output path, input path, and 3-state path. Each path has its own pair of storage elements that can act as either registers or latches. For more information, see **Storage Element Functions**.

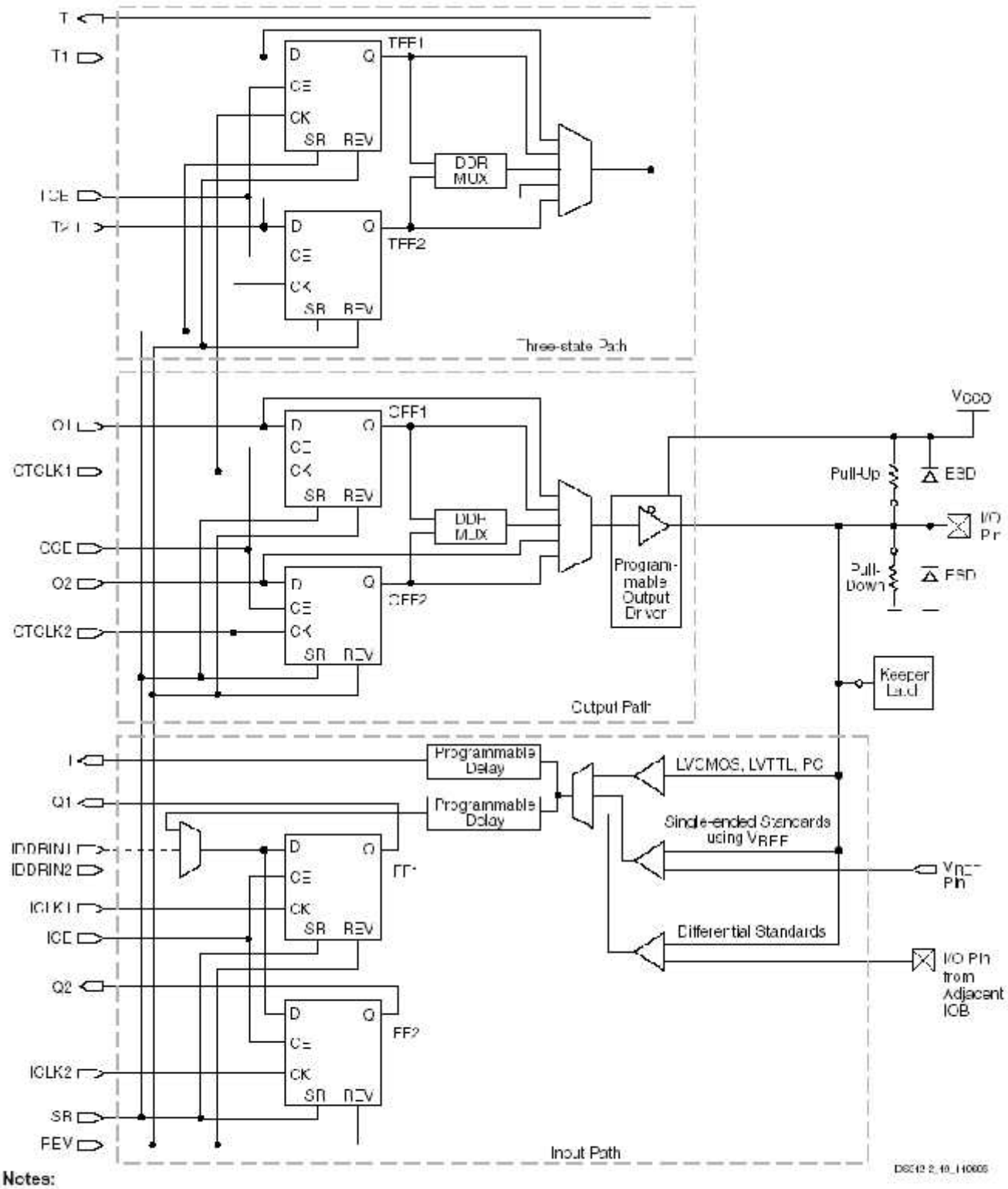
The three main signal paths are as follows:

• The input path carries data from the pad, which is bonded to a package pin, through an optional programmable delay element directly to the I line. After the delay element, there are alternate routes through a pair of storage elements to the IQ1 and IQ2 lines. The IOB outputs I, IQ1, and IQ2 lead to the FPGA's internal logic. The delay element can be set to ensure a hold time of zero (see **Input Delay Functions**).

• The output path, starting with the O1 and O2 lines, carries data from the FPGA's internal logic through a multiplexer and then a three-state driver to the IOB pad. In addition to this direct path, the multiplexer provides the option to insert a pair of storage elements.

• The 3-state path determines when the output driver is high impedance. The T1 and T2 lines carry data from nthe FPGA's internal logic through a multiplexer to the output driver. In addition to this direct path, the multiplexer provides the option to insert a pair of storage elements.

• All signal paths entering the IOB, including those associated with the storage elements, have an inverter option. Any inverter placed on these paths is automatically absorbed into the IOB.



Notes:

1. All IOB control and output path signals have an inverting polarity option within the IOB.
2. IDDRIN1/IDDRIN2 signals shown with dashed lines connected to the adjacent IOB in a differential pair only, not to the FPGA fabric.

Simplified IOB Diagram

4.3 Configurable Logic Block (CLB) and Slice Resources

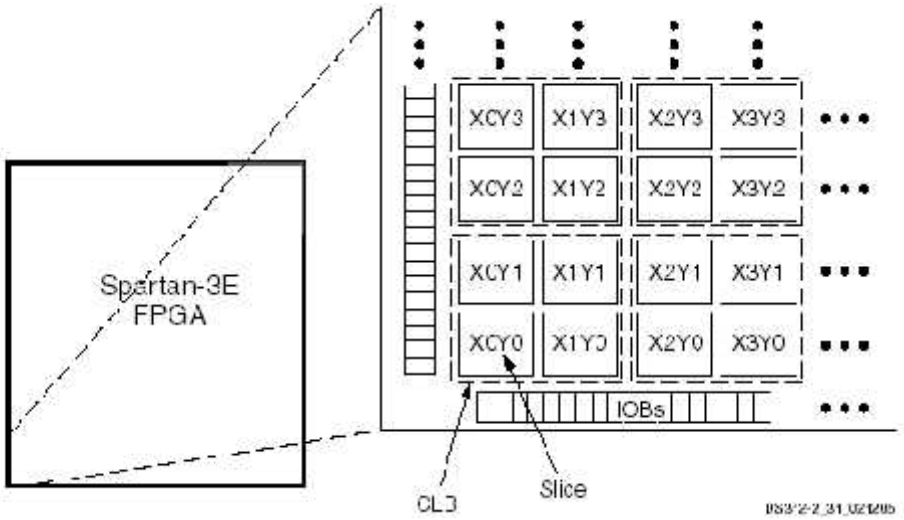
For additional information, refer to the "*Using Configurable Logic Blocks (CLBs)*" chapter in UG331.

CLB Overview

The Configurable Logic Blocks (CLBs) constitute the main logic resource for implementing synchronous as well as combinatorial circuits. Each CLB contains four slices, and each slice contains two Look-Up Tables (LUTs) to implement logic and two dedicated storage elements that can be used as flip-flops or latches. The LUTs can be used as a 16x1 memory (RAM16) or as a 16-bit shift register (SRL16), and additional multiplexers and carry logic simplify wide
logic and arithmetic functions. Most general-purpose logic in a design is automatically mapped to the slice resources in the CLBs. Each CLB is identical, and the Spartan-3E family CLB structure is identical to that for the Spartan-3 family.

CLB Array

The CLBs are arranged in a regular array of rows and columns as shown in Figure 14. Each density varies by the number of rows and columns of CLBs (see Table 9).



CLB Locations

Table 9: **Spartan-3E CLB Resources**

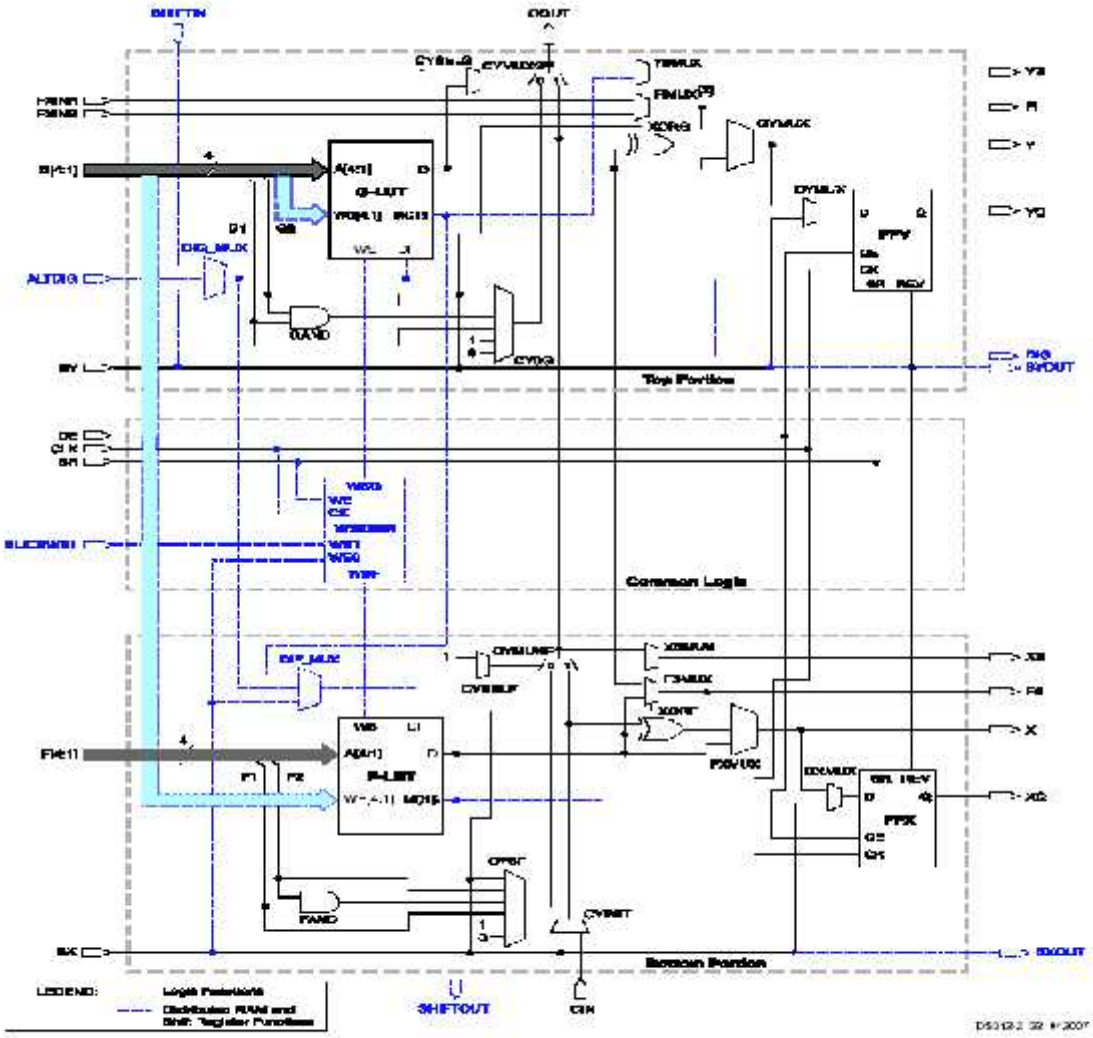
Device	CLB Rows	CLB Columns	CLB Total[1]	Slices	LUTs / Flip-Flops	Equivalent Logic Cells	RAM16 / SRL16	Distributed RAM Bits
XC3S100E	22	16	240	960	1,920	2,160	960	15,360
XC3S250E	34	26	612	2,448	4,896	5,508	2,448	39,168
XC3S500E	46	34	1,164	4,656	9,312	10,476	4,656	74,496
XC3S1200E	60	46	2,168	8,672	17,344	19,512	8,672	138,752
XC3S1600E	76	58	3,688	14,752	29,504	33,192	14,752	236,032

Notes:

1. The number of CLBs is less than the multiple of the rows and columns because the block RAM/multiplier blocks and the DCMs are embedded in the array (see Figure 1 in Module 1).

Slices:

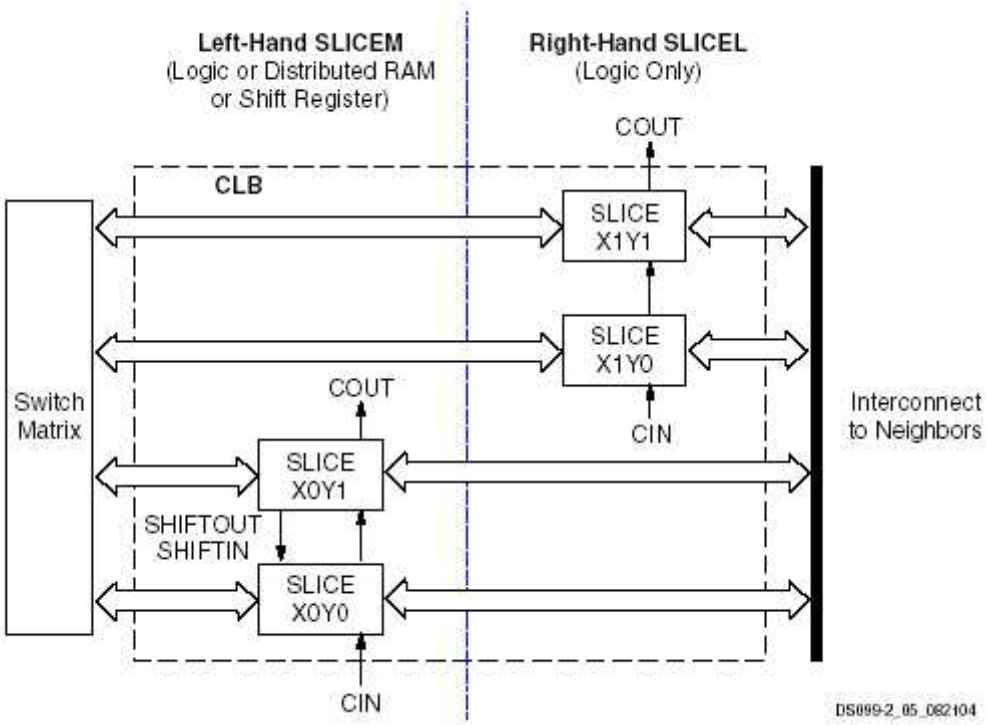
Each CLB comprises four interconnected slices, as shown in Figure 16. These slices are grouped in pairs. Each pair is organized as a column with an independent carry chain. The left pair supports both logic and memory functions and its slices are called SLICEM. The right pair supports logic only and its slices are called SLICEL. Therefore half the LUTs support both logic and memory (including both RAM16 and SRL16 shift registers) while half support logic only, and the two types alternate throughout the array columns. The SLICEL reduces the size of the CLB and lowers the cost of the device, and can also provide a performance advantage over the SLICEM.



Notes:

1. Options to invert signal polarity as well as other options (e.g. clock enable lines) for various primitives are not shown.
2. The index can be 5, 7, or 9, depending on the slice. The upper SLICE has an F8MUX, and the upper SLICEM has an F7MUX. The lower SLICE and SLICEM both have an F6MUX.

Simplified Diagram of the Left-Hand SLICEM



Arrangement of Slices within the CLB

Slice Location Designations

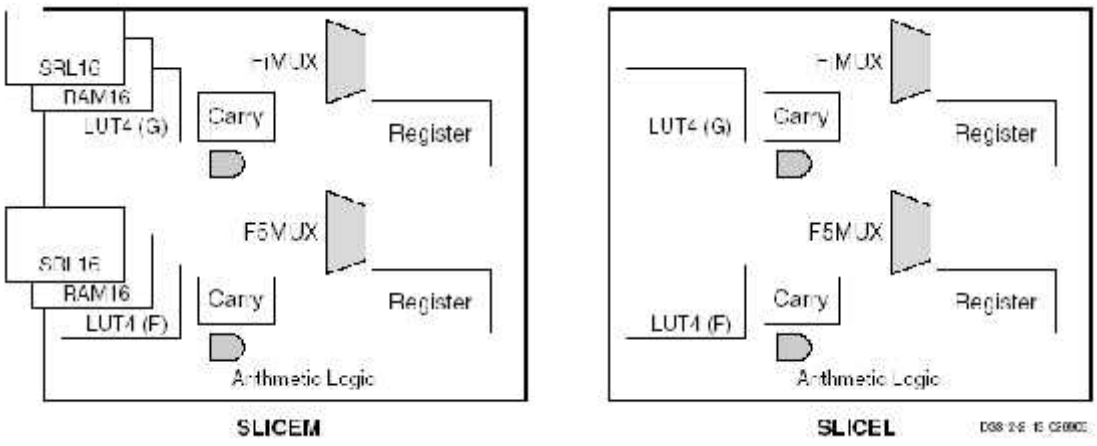
The Xilinx development software designates the location of a slice according to its X and Y coordinates, starting in the bottom left corner, as shown in Figure 14. The letter 'X' followed by a number identifies columns of slices, incrementing from the left side of the die to the right. The letter 'Y' followed by a number identifies the position of each slice in a pair as well as indicating the CLB row, incrementing from he bottom of the die. Figure 16 shows the CLB located in he lower left-hand corner of the die. The SLICEM always as an even 'X' number, and the LICEL always has an odd X' number.

Slice Overview

A slice includes two LUT function generators and two storage elements, along with additional logic, as shown in Figure 17.

Both SLICEM and SLICEL have the following elements in common to provide logic, arithmetic, and ROM functions:

• Two 4-input LUT function generators, F and G
• Two storage elements
• Two wide-function multiplexers, F5MUX and FiMUX
• Carry and arithmetic logic



Resources in a Slice

The SLICEM pair supports two additional functions:
• Two 16x1 distributed RAM blocks, RAM16
• Two 16-bit shift registers, SRL16

Each of these elements is described in more detail in the following sections.

Logic Cells

The combination of a LUT and a storage element is known as a “Logic Cell”. The additional features in a slice, such as the wide multiplexers, carry logic, and arithmetic gates, add to the capacity of a slice, implementing logic that would otherwise require additional LUTs. Benchmarks have shown that the overall slice is equivalent to 2.25 simple logic cells. This calculation provides the equivalent logic cell count

Slice Details

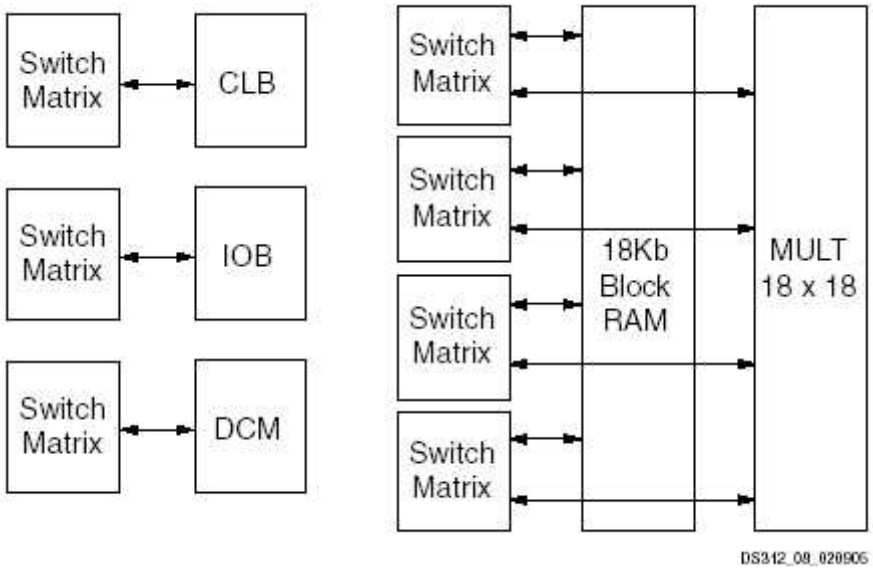
Figure 15 is a detailed diagram of the SLICEM. It represents a superset of the elements and connections to be found in all slices. The dashed and gray lines (blue when viewed in color) indicate the resources found only in the SLICEM and not in the SLICEL. Each slice has two halves, which are differentiated as top and bottom to keep them distinct from the upper and lower
slices in a CLB. The control inputs for the clock (CLK), Clock Enable (CE), Slice Write Enable (SLICEWE1), and Reset/Set (RS) are shared in common between the two halves. The LUTs located in the top and bottom portions of the slice are referred to as "G" and "F", respectively, or the "G-LUT" and the "F-LUT". The storage elements in the top and bottom portions of the slice are called FFY and FFX, respectively. Each slice has two multiplexers with F5MUX in the bottom
portion of the slice and FiMUX in the top portion. Depending on the slice, the FiMUX takes on the name F6MUX, F7MUX, or F8MUX, according to its position in the multiplexer chain. The lower SLICEL and SLICEM both have an F6MUX. The upper SLICEM has an F7MUX, and the upper
SLICEL has an F8MUX. The carry chain enters the bottom of the slice as CIN and exits at the top as COUT. Five multiplexers control the chain: CYINIT, CY0F, and CYMUXF in the bottom portion and CY0G and CYMUXG in the top portion. The dedicated arithmetic logic includes the exclusive-OR gates XORF and XORG (bottom and top portions of the slice, respectively) as well as the AND gates FAND and GAND (bottom and top portions, respectively). See Table 10 for a description of all the slice input and output signals.

4.4 Interconnect

For additional information, refer to the "*Using Interconnect*" chapter in UG331. Interconnect is the programmable network of signal pathways between the inputs and outputs of functional elements within the FPGA, such as IOBs, CLBs, DCMs, and block RAM.

4.5 Overview

Interconnect, also called routing, is segmented for optimal connectivity. Functionally, interconnect resources are identical to that of the Spartan-3 architecture. There are four kinds of interconnects: long lines, hex lines, double lines, and direct lines. The Xilinx Place and Route (PAR) software exploits the rich interconnect array to deliver optimal system performance and the fastest compile times.



Four Types of Interconnect Tiles (CLBs, IOBs, DCMs, and Block RAM/Multiplier)

Chapter 5

An Introduction to Spartan 3E- Kit

5.1 Introduction

The Basys board is a circuit design and implementation platform that anyone can use to gain experience building real digital circuits. Built around a Xilinx Spartan-3E Field Programmable Gate Array and a Cypress EZUSB controller, the Basys board provides complete, ready-to-use hardware suitable for hosting circuits ranging from basic logic devices to complex controllers. A large collection of on-board I/O devices and all required FPGA support circuits are included, so countless designs can be created without the need for any other components.

Four standard expansion connectors allow designs to grow beyond the Basys board using breadboards, user-designed circuit boards, or Pmods (Pmods are inexpensive analog and digital I/O modules that offer A/D & D/A conversion, motor drivers, sensor inputs, and many other features). Signals on the 6-pin connectors are protected against ESD damage and short-circuits, ensuring a long operating life in any environment. The Basys board works seamlessly with all versions of the Xilinx ISE tools, including the free WebPack. It ships with a USB cable that provides power and a programming interface, so no other power supplies or programming cables are required

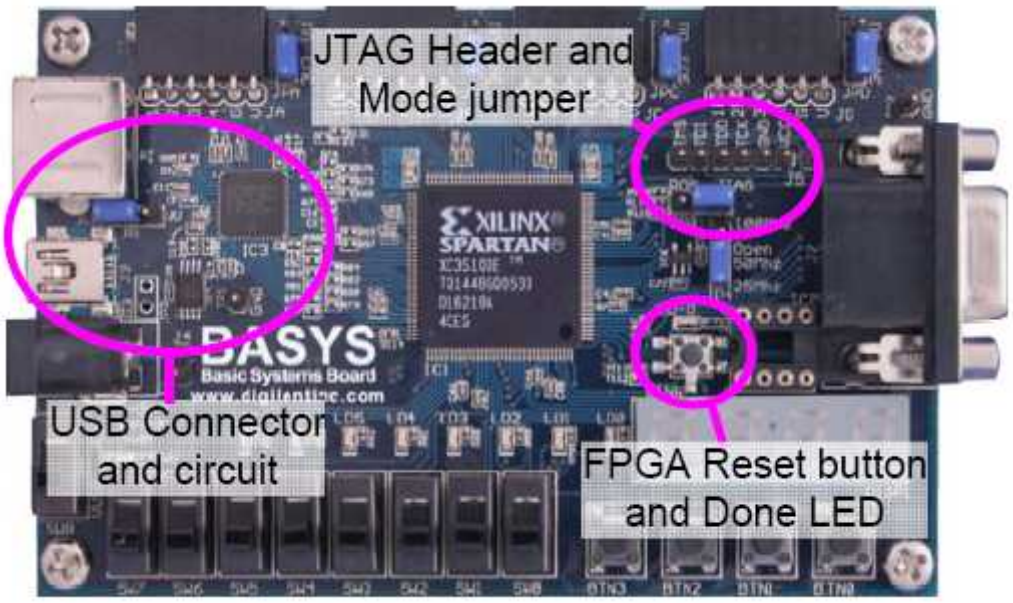


Figure 1. Basys programming circuit locations

5.2 Board Power

The Basys board is typically powered from a USB cable, but a power jack and battery connector are also provided so that external supplies can be used. To use USB power, set the power source switch (SW8) to USB and attach the USB cable. To use an external wall-plug power supply, set SW8 to EXT and attach a 5VDC to 9VDC supply to the center-positive, 2.1/5.5mm power jack. To use battery power, set SW8 to EXT and attach a 4V-9V battery pack to the 2-pin, 100-mil spaced battery connector (four AA cells in series make a good 6+/- volt supply). Voltages higher than 9V on either power connector may cause permanent damage. SW8 can also be used to turn off main power by setting it to the unused power input (e.g., if USB power is used, setting SW8 to EXT will shut off board power without unplugging the USB cable).

Input power is routed through the power switch (SW8) to the four 6- pin expansion connectors and to a National Semiconductor LP8345 voltage regulator. The LP8345 produces the main 3.3V supply for the board, and it also drives secondary regulators to produce the 2.5V and 1.2V supply voltages required by the FPGA. Total board current is dependant on FPGA configuration, clock frequency,

and external connections. In test circuits with roughly 20K gates routed, a 50MHz clock source, and all LEDs illuminated, about 100mA of current is drawn from the 1.2V supply, 50mA from the 2.5V supply, and 50mA from the 3.3V supply. Required current will increase if larger circuits are configured in the FPGA, or if peripheral boards are attached.

5.3 Configuration

After power-on, the FPGA on the Basys board must be configured before it can perform any useful functions. During configuration, a "bit" file is transferred into memory cells within the FPGA to define the logical functions and circuit interconnects. The free ISE/WebPack CAD software from Xilinx can be used to create bit files from VHDL, Verilog, or schematic-based source files.

Digilent's PC-based program called Adept can be used to configure the FPGA with any suitable bit file stored on the computer. Adept uses the USB cable to transfer a selected bit file from the PC to the FPGA (via the FPGA's JTAG programming port). Adept can also program a bit file into an on-board non-volatile ROM called "Platform Flash". Once programmed, the Platform Flash can automatically transfer a stored bit file to the FPGA at a subsequent power-on or reset event if the Mode Jumper is set to ROM. The FPGA will remain configured until it is reset by a power-cycle event or by the FPGA reset button (BTNR) being pressed. The Platform Flash ROM will retain a bit file until it is reprogrammed, regardless of power-cycle events.

To program the Basys board, attach the USB cable to the board (if USB power will not be used, attach a suitable power supply to the power jack or battery connector on the board, and set the power switch to VEXT). Start the Adept software, and wait for the FPGA and the Platform Flash ROM to be recognized. Use the browse function to associate the desired .bit file with the FPGA, and/or the desired .mcs file with the Platform Flash ROM. Right-click on the device to be programmed, and select the "program" function. The configuration file will be sent to the FPGA or Platform Flash, and the software will indicate whether programming was successful. The "configuration done" LED (LD_D) will also illuminate after the FPGA has been successfully configured.

5.4 Oscillators

The Basys board includes a primary, user settable silicon oscillator that produces 25MHz, 50MHz, or 100MHz based on the position of the clock select jumper at JP4. A socket for a second oscillator is provided at IC7 (the IC7 socket can accommodate any 3.3V CMOS oscillator in a half-size DIP package). The primary and secondary oscillators are connected to global clock input pins at pin 54 and pin 53 respectively

5.5 User I/O

Four pushbuttons and eight slide switches are provided for circuit inputs. Pushbutton inputs are normally low and driven high only when the pushbutton is pressed. Slide switches generate constant high or low inputs depending on position. Pushbuttons and slide switches all have series resistors for protection against short circuits (a short circuit would occur if an FPGA pin assigned to a pushbutton or slide switch was inadvertently defined as an output). Eight LEDs and a four-digit seven segment LED display are provided for circuit outputs. LED anodes are driven from the FPGA via current-limiting resistors, so they will illuminate when a logic '1' is written to the corresponding FPGA pin. A ninth LED is provided as a power-indicator LED, and a tenth LED (LD-D) illuminates any time the FPGA has been successfully programmed.

5.6 PS/2 Port

The 6-pin mini-DIN connector can accommodate a PS/2 mouse or keyboard. Most PS/2 devices can operate from a 3.3V supply, but some older devices may require a 5VDC supply. A jumper on the Basys board (JP1) selects whether 3.3V or VU is supplied to the PS/2 connector. For 5V, set JP1 to VU and ensure that Basys is powered with a 5VDC wall plug supply. For 3.3V, set the jumper to 3.3V. For 3.3V operation, any board power supply (including USB) can be used. Both the mouse and keyboard use a two-wire serial bus (clock and data) to communicate with a host device. Both use 11-bit words that include a start, stop and odd parity bit, but the data packets are organized differently, and the keyboard interface allows bi-directional data transfer.

The clock and data signals are only driven when data transfers occur, and otherwise they are held in the "idle" state at logic '1'. The timings define signal requirements for mouse-to-host communications and bi-directional keyboard communications. A PS/2 interface circuit can be implemented in the FPGA to create a keyboard or mouse interface.

5.7 Dumping ProcedureProgramming through JTAG:

For programming the FPGA we need a JTAG cable which is a 6 pin cable converted to a parallel port cable connected to CPU, So the FPGA is programmed through this cable. And this type of programming is called "flash programming".

Connecting the USB Cable

The kit includes a standard USB Type A/Type B cable, similar to the one shown in Figure . The actual cable colour might vary from the picture.

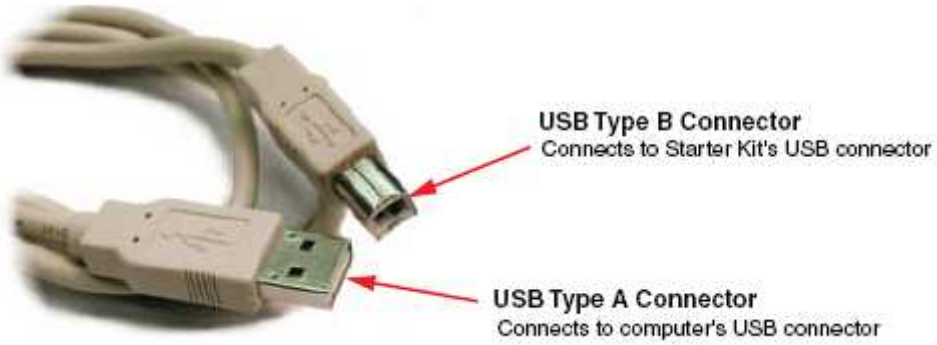


Figure 18 **Standard USB Type A/Type B Cable**

The wider and narrower Type A connector fits the USB connector at the back of the computer. After installing the Xilinx software, connect the square Type B connector to the Spartan-3E Starter Kit board, as shown in Figure 19 . The USB connector is on the left side of the board, immediately next to the Ethernet connector. When the board is powered on, the Windows operating system should recognize and install the associated driver software.

When the USB cable driver is successfully installed and the board is correctly connected to the PC, a green LED lights up, indicating a good connection.

Programming via iMPACT

After successfully compiling an FPGA design using the Xilinx development software, the design can be downloaded using the iMPACT programming software and the USB cable. To begin programming, connect the USB cable to the starter kit board and apply power to the board. Then, double-click **Configure Device (iMPACT)** from within Project Navigator, as shown in Figure20.

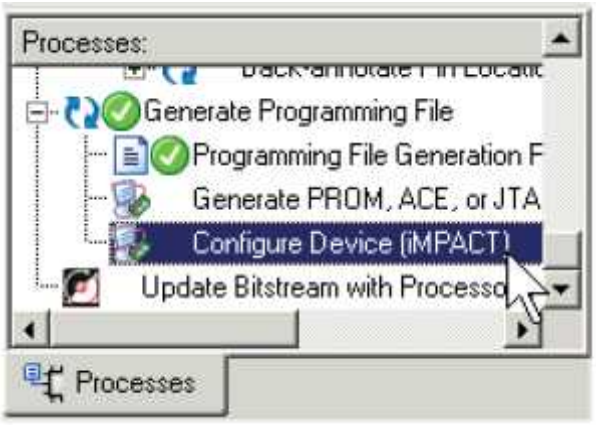


Figure 20 : **Double-Click to Invoke iMPACT**

If the board is connected properly, the iMPACT programming software automatically recognizes the three devices in the JTAG programming file, as shown in Figure 21. If not already prompted, click the first device in the chain, the Spartan-3E FPGA, to highlight it. Right-click the FPGA and select **Assign New Configuration File**. Select the desired FPGA configuration file and click **OK**.

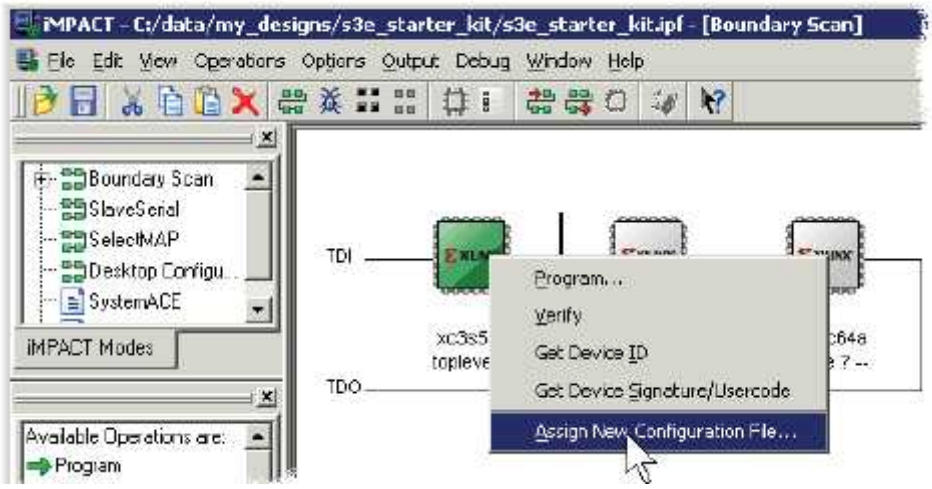


Figure 21 : **Right-Click to Assign a Configuration File to the Spartan-3E FPGA**

If the original FPGA configuration file used the default StartUp clock source, CCLK, iMPACT issues the warning message shown in Figure 22. This message can be safely ignored. When downloading via JTAG, the iMPACT software must change the StartUP clock source to use the TCK JTAG clock source.

Figure 22 : **iMPACT Issues a Warning if the StartUp Clock Was Not CCLK**

To start programming the FPGA, right-click the FPGA and select **Program**. The iMPACT software reports status during programming process. Direct programming to the FPGA takes a few seconds to less than a minute, depending on the speed of the PC's USB port and the iMPACT settings.

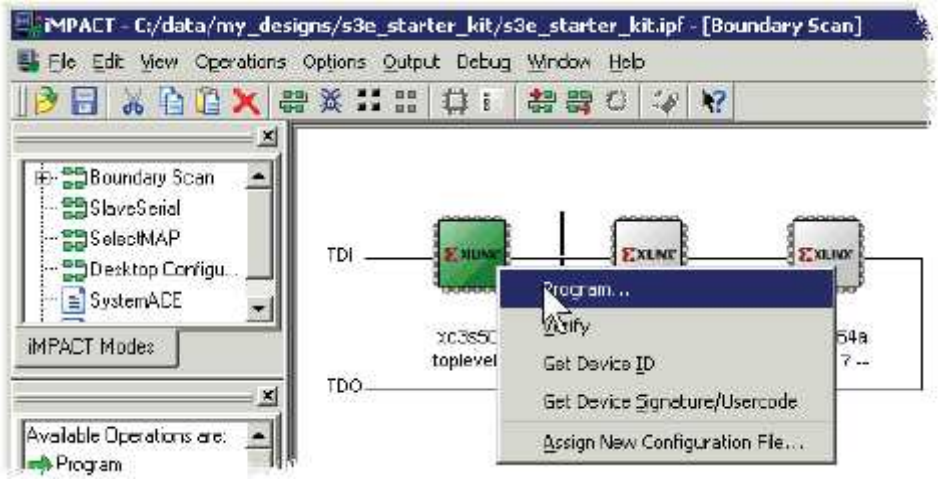


Figure 23: **Right-Click to Program the Spartan-3E FPGA**

When the FPGA successfully programs, the iMPACT software indicates success, as shown in Figure 24 The FPGA application is now executing on the board and the DONE pin LED (see Figure 17) lights up.

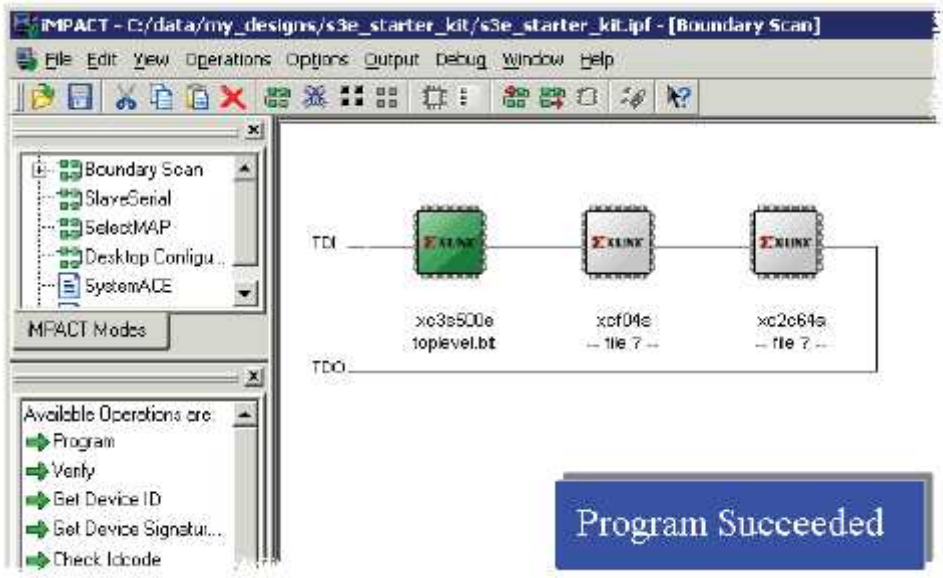


Figure 24 : **iMPACT Programming Succeeded, the FPGA's DONE Pin is High**

CHAPTER – 6

RESULTS

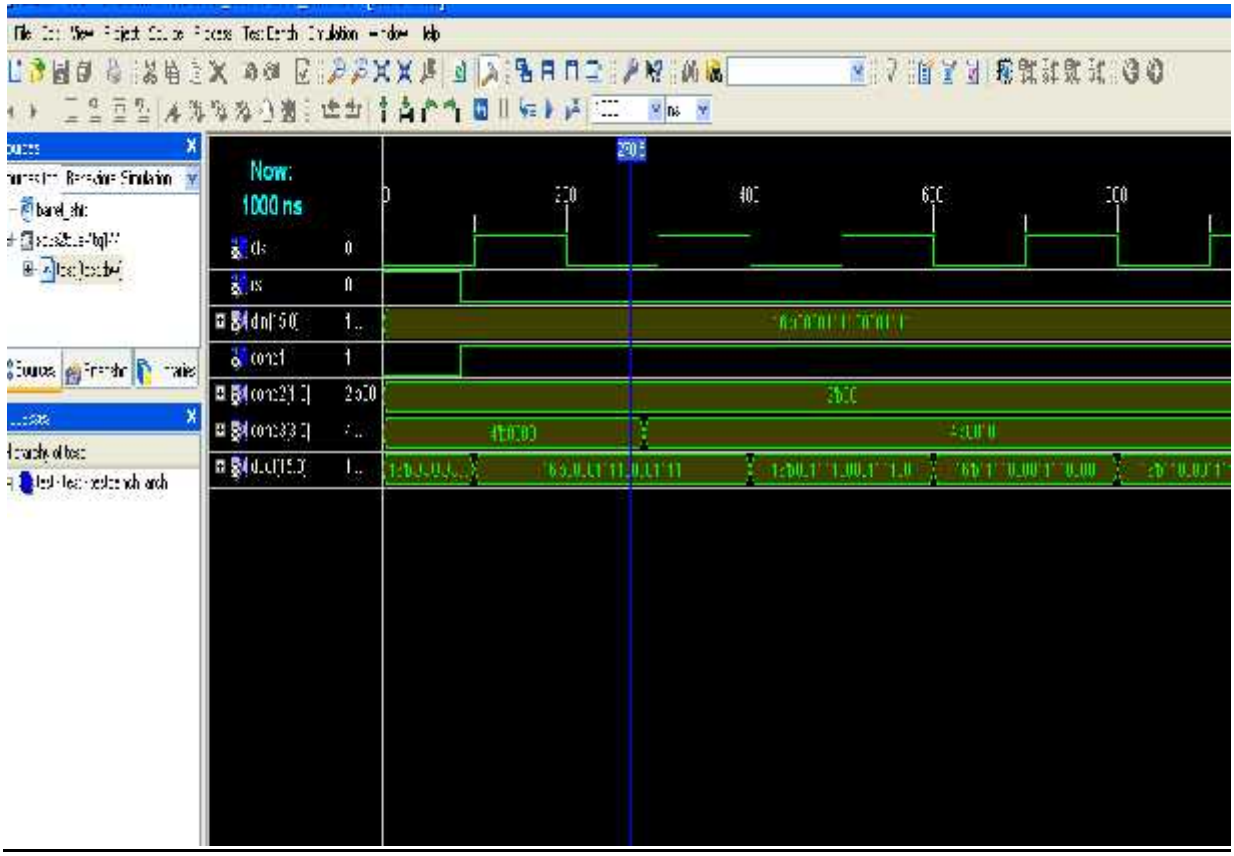


Fig 6.1: simulation result for left shift

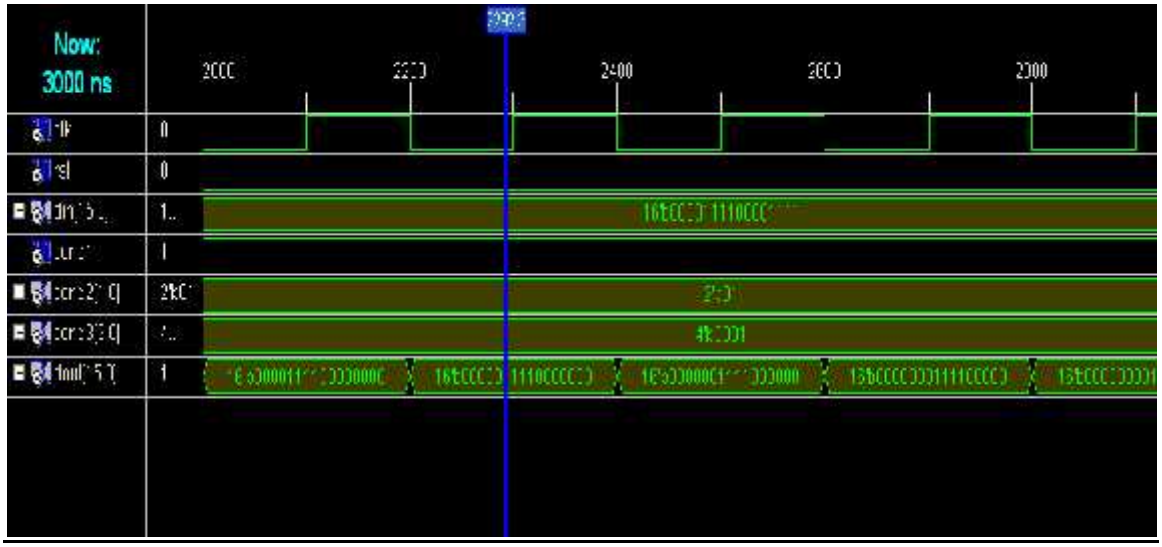


Fig 6.2: Simulation results for right shift

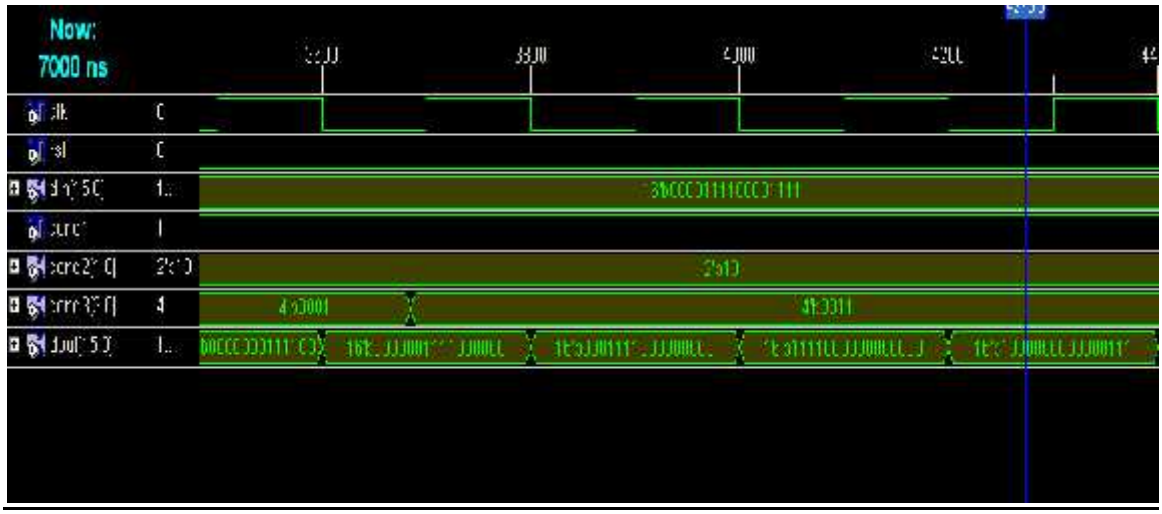


Fig 6.3 Simulation results for circular left shift

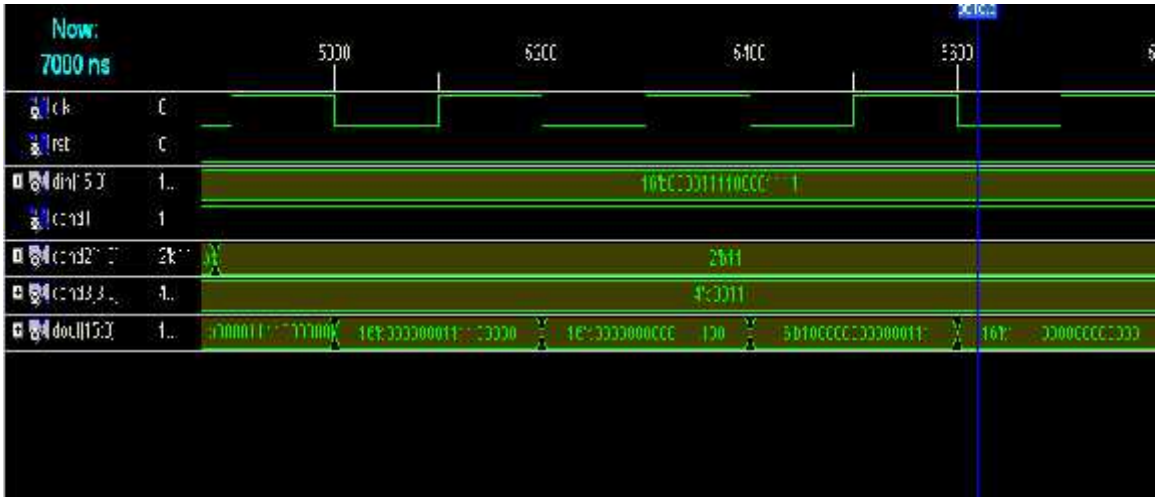


Fig 6.4: simulation results for circular right shift

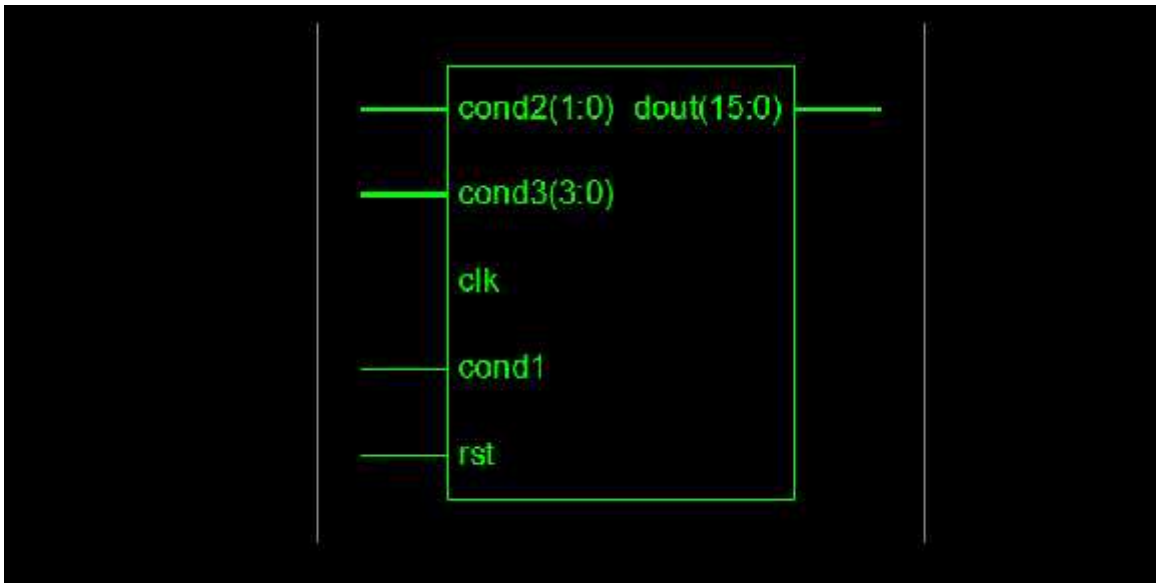


Fig 6.5:Block diagram of Barrel shifter

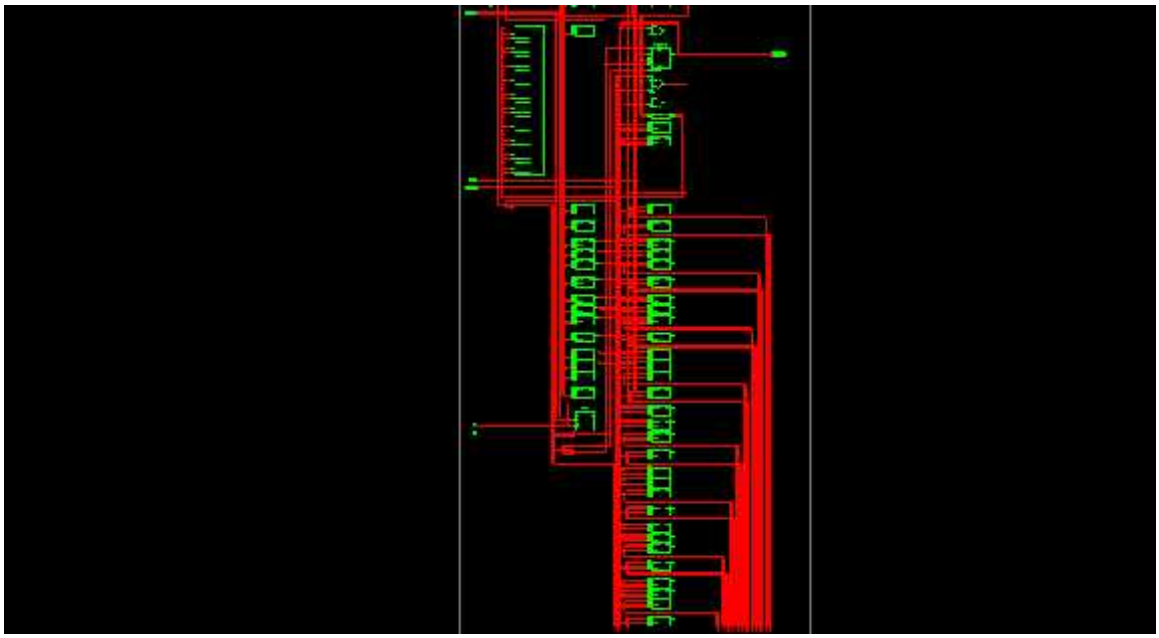


Fig 6.6: Register Transfer Logic for Barrel shifter

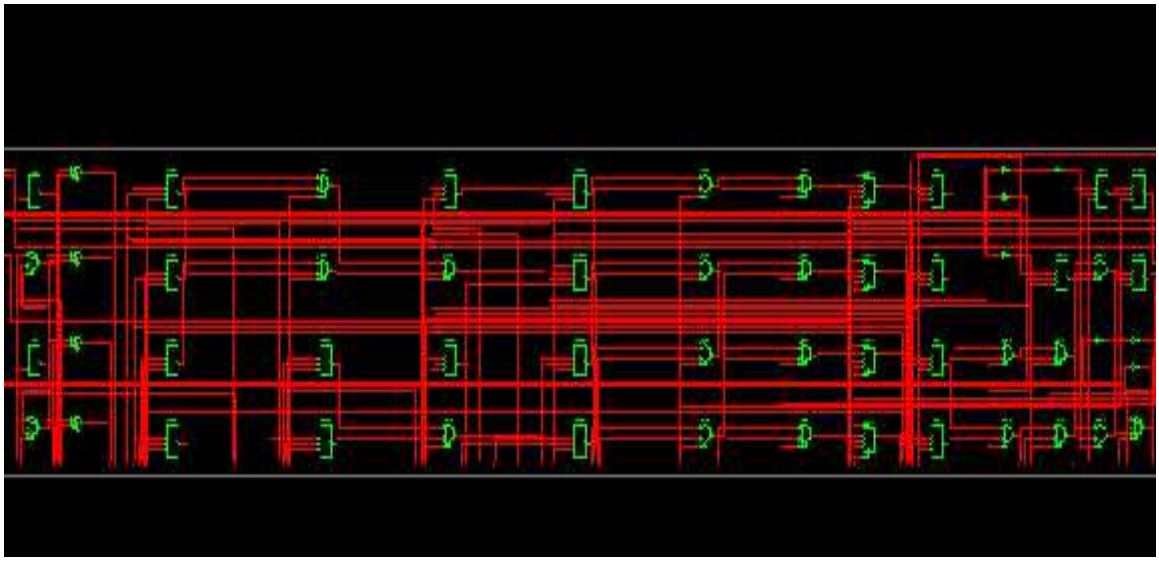


Fig6.7:technology schematic for barrel shifter

Module Name:	barrel_shft	**• Errors:**	No Errors
Target Device:	xc3s250e-4tq144	**• Warnings:**	8 Warnings
Product Version:	ISE 9.1i	**• Updated:**	Sun Jul 19 13:37:13 2015

BARREL_SHFT Partition Summary

No partition information was found.

Device Utilization Summary

Logic Utilization	Used	Available	Utilization	Note
Number of Slice Flip Flops	49	4,896	1%	
Number of 4 input LUTs	246	4,896	5%	
Logic Distribution				
Number of occupied Slices	58	2,448	6%	
Number of Slices containing only related logic	58	58	100%	
Number of Slices containing unrelated logic	0	58	0%	
Total Number of 4 input LUTs	311	4,896	6%	
Number used as logic	246			
Number used as a route-thru	65			
Number of bonded IOBs	35	108	23%	
Number of GCLKs	1	24	4%	
Total equivalent gate count for design	2,776			
Additional JTAG gate count for IOBs	1,248			

Performance Summary

Fig 6.8: Design summary for Barrel shifter

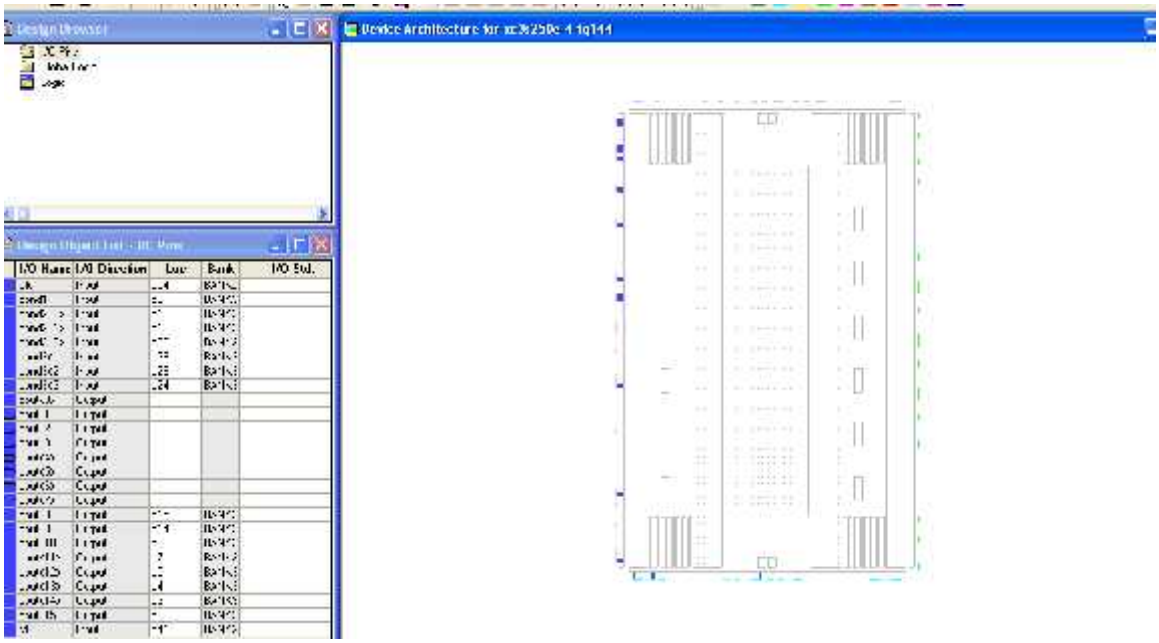


Fig 6.9 Pin assignment for barrel shifter

SYNTHESIS REPORT

Release 9.1i - xst J.30

--> Parameter TMPDIR set to ./xst/projnav.tmp

CPU : 0.00 / 0.33 s | Elapsed : 0.00 / 0.00 s

--> Parameter xsthdpdir set to ./xst

CPU : 0.00 / 0.33 s | Elapsed : 0.00 / 0.00 s

--> Reading design: barrel_shft.prj

TABLE OF CONTENTS

==========

=

* Synthesis Options Summary *

```
=========================================================================
=
---- Source Parameters
Input File Name                    : "barrel_shft.prj"
Input Format                       : mixed
Ignore Synthesis Constraint File   : NO

---- Target Parameters
Output File Name                   : "barrel_shft"
Output Format                      : NGC
Target Device                      : xc3s250e-4-tq144

---- Source Options
Top Module Name                    : barrel_shft
Automatic FSM Extraction           : YES
FSM Encoding Algorithm             : Auto
Safe Implementation                : No
FSM Style                          : lut
RAM Extraction                     : Yes
RAM Style                          : Auto
ROM Extraction                     : Yes
Mux Style                          : Auto
Decoder Extraction                 : YES
Priority Encoder Extraction        : YES
Shift Register Extraction          : YES
Logical Shifter Extraction         : YES
XOR Collapsing                     : YES
ROM Style                          : Auto
Mux Extraction                     : YES
Resource Sharing                   : YES
Asynchronous To Synchronous        : NO
```

Multiplier Style : auto
Automatic Register Balancing : No

---- Target Options
Add IO Buffers : YES
Global Maximum Fanout : 500
Add Generic Clock Buffer(BUFG) : 24
Register Duplication : YES
Slice Packing : YES
Optimize Instantiated Primitives : NO
Use Clock Enable : Yes
Use Synchronous Set : Yes
Use Synchronous Reset : Yes
Pack IO Registers into IOBs : auto
Equivalent register Removal : YES

---- General Options
Optimization Goal : Speed
Optimization Effort : 1
Library Search Order : barrel_shft.lso
Keep Hierarchy : NO
RTL Output : Yes
Global Optimization : AllClockNets
Read Cores : YES
Write Timing Constraints : NO
Cross Clock Analysis : NO
Hierarchy Separator : /
Bus Delimiter : <>
Case Specifier : maintain
Slice Utilization Ratio : 100
BRAM Utilization Ratio : 100

Verilog 2001 : YES
Auto BRAM Packing : NO
Slice Utilization Ratio Delta : 5
HDL Synthesis Report

Macro Statistics
Adders/Subtractors : 1
32-bit adder : 1
Counters : 1
32-bit up counter : 1
Registers : 17
1-bit register : 17
Multiplexers : 76
1-bit 16-to-1 multiplexer : 60
1-bit 4-to-1 multiplexer : 16

==========
=

==========
=
* Advanced HDL Synthesis *
==========
=

Loading device for application Rf_Device from file '3s250e.nph' in environment C:\Xilinx91i.

==========
=
Advanced HDL Synthesis Report

Macro Statistics

Adders/Subtractors : 1

32-bit adder : 1

Counters : 1

32-bit up counter : 1

Registers : 17

Flip-Flops : 17

Multiplexers : 76

1-bit 16-to-1 multiplexer : 60

1-bit 4-to-1 multiplexer : 16

==========

=

Device utilization summary:

Selected Device : 3s250etq144-4

Number of Slices:	170	out of 2448	6%
Number of Slice Flip Flops:	49	out of 4896	1%
Number of 4 input LUTs:	309	out of 4896	6%
Number of IOs:	25		
Number of bonded IOBs:	25	out of 108	23%
Number of GCLKs:	1	out of 24	4%

Partition Resource Summary:

No Partitions were found in this design.

==========

=

TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
GENERATED AFTER PLACE-and-ROUTE.

Clock Information:

Clock Signal	Clock buffer(FF name)	Load
clk	BUFGP	33
sclk	NONE(i_9)	16

INFO:Xst:2169 - HDL ADVISOR - Some clock signals were not automatically buffered by XST with BUFG/BUFR resources. Please use the buffer_type constraint in order to insert these buffers to the clock signals to help prevent skew problems.

Asynchronous Control Signals Information:

Control Signal	Buffer(FF name)	Load
N0(XST_GND:G)	NONE(i_9)	16
rst	IBUF	16

----------+----------+-------+

Timing Summary:

Speed Grade: -4

Minimum period: 8.965ns (Maximum Frequency: 111.551MHz)

Minimum input arrival time before clock: 8.466ns

Maximum output required time after clock: 4.991ns

Maximum combinational path delay: No path found

Timing Detail:

All values displayed in nanoseconds (ns)

=

Timing constraint: Default period analysis for Clock 'clk'

Clock period: 8.965ns (frequency: 111.551MHz)

Total number of paths / destination ports: 17953 / 66

Delay: 8.965ns (Levels of Logic = 34)

Total 8.965ns (6.625ns logic, 2.340ns route)

(73.9% logic, 26.1% route)

=

Timing constraint: Default period analysis for Clock 'sclk'

Clock period: 7.455ns (frequency: 134.138MHz)

Total number of paths / destination ports: 792 / 16

Delay: 7.455ns (Levels of Logic = 6)
Source: i_0 (FF)
Destination: i_1 (FF)
Source Clock: sclk rising
Destination Clock: sclk rising

Data Path: i_0 to i_1

Cell:in->out	fanout	Gate Delay	Net Delay	Logical Name (Net Name)
FDCPE:C->Q	17	0.591	1.086	i_0 (i_0)
LUT3:I2->O	7	0.704	0.787	cond3<3>_5 (cond3<0>_mmx_out17)
LUT3:I1->O	2	0.704	0.526	cond3<1>11 (cond3<1>17)
LUT3:I1->O	1	0.704	0.499	cond3<2>1131 (cond3<2>211)
LUT3:I1->O	1	0.704	0.000	Mmux_i_1_mux0000_5 (N1414)
MUXF5:I0->O	1	0.321	0.000	Mmux_i_1_mux0000_3_f5 (Mmux_i_1_mux0000_3_f5)
MUXF6:I1->O	1	0.521	0.000	Mmux_i_1_mux0000_2_f6 (i_1_mux0000)
FDCPE:D		0.308		i_1
Total		7.455ns (4.557ns logic, 2.898ns route) (61.1% logic, 38.9% route)		

==========

=

Timing constraint: Default OFFSET IN BEFORE for Clock 'sclk'
Total number of paths / destination ports: 854 / 32

Offset: 8.466ns (Levels of Logic = 7)
Source: cond3<0> (PAD)
Destination: i_1 (FF)
Destination Clock: sclk rising

Data Path: cond3<0> to i_1

Cell:in->out	fanout	Gate Delay	Net Delay	Logical Name (Net Name)
IBUF:I->O	135	1.218	1.470	cond3_0_IBUF (cond3_0_IBUF)
LUT3:I0->O	7	0.704	0.787	cond3<3>_5 (cond3<0>_mmx_out17)
LUT3:I1->O	2	0.704	0.526	cond3<1>11 (cond3<1>17)
LUT3:I1->O	1	0.704	0.499	cond3<2>1131 (cond3<2>211)
LUT3:I1->O	1	0.704	0.000	Mmux_i_1_mux0000_5 (N1414)
MUXF5:I0->O	1	0.321	0.000	Mmux_i_1_mux0000_3_f5 (Mmux_i_1_mux0000_3_f5)
MUXF6:I1->O	1	0.521	0.000	Mmux_i_1_mux0000_2_f6 (i_1_mux0000)
FDCPE:D		0.308		i_1

Total 8.466ns (5.184ns logic, 3.282ns route)
(61.2% logic, 38.8% route)

==========

=

Timing constraint: Default OFFSET OUT AFTER for Clock 'sclk'

Total number of paths / destination ports: 16 / 16

Offset: 4.991ns (Levels of Logic = 1)

Source: i_8 (FF)

Destination: dout<8> (PAD)

Source Clock: sclk rising

Data Path: i_8 to dout<8>

Cell:in->out	fanout	Gate Delay	Net Delay	Logical Name (Net Name)

FDCPE:C->Q 21 0.591 1.128 i_8 (i_8)

OBUF:I->O 3.272 dout_8_OBUF (dout<8>)

Total 4.991ns (3.863ns logic, 1.128ns route)

(77.4% logic, 22.6% route)

==========

=

CPU : 11.44 / 11.81 s | Elapsed : 11.00 / 11.00 s

-->

Total memory usage is 156400 kilobytes

Number of errors : 0 (0 filtered)

Number of warnings : 3 (0 filtered)

Number of infos : 6 (0 filtered)

CONCLUSION:

Hence we have designed the IP Core for Barrel Shifter using VHDL. The simulation has been done using ISE Simulator. The synthesis has been done using XILINX ISE 9.1i.

The bit file has been generated and the output is dumped on the FPGA Device (Spartan3E).

FUTURE SCOPE:

The design has been done for the 16-bit Barrel Shifter. The core can be used to design for further designs of 32-bit , 64-bit and so on to be utilized in DSP Processors and any communication systems like USB transmitters etc.

REFERENCES:

1. www.wikipedia.com
2. www.google.com
3. www.xilinx.com
4. www.digilent.com (for reference manual)
5. Digital Design Principles and Practices by John F. Wakerly, Fourth Edition
6. Advanced VHDL Design by J. Basker