

# VIRTEX: *Under the Hood*

## An Insider's Guide to Xilinx FPGA Devices

Jesse Jenkins, Austin Lesea & Peter Alfke

Second Edition: July, 2016,  
Austin Lesea

# Table of Contents

Introduction

Chapter 1

Chapter 2 Getting to Virtex

Chapter 3 Virtex Architectures

Chapter 4 Xilinx Design Software

Chapter 5 Configuration, Reconfiguration & Security

Chapter 6 Virtex Memory

Chapter 7 Virtex Arithmetic Structures

Chapter 8 Virtex. Timing, Models and Closure

Chapter 9 Digital Clock Management

Chapter 10 High Speed Transceivers & the MGT

Chapter 11 SelectIO, DCI and ChipSync

Chapter 12 Embedded Microprocessors

Chapter 13 Virtex FPGA devices in DataCom

Chapter 14 Virtex Family Memory Controllers

Chapter 15 DSP with Virtex FPGA devices

Chapter 16 Virtex Under the Hood

Glossary

## *Second Edition*

*It has been quite a while since this book has been written: it was badly in need of a complete overhaul. As Peter Alfke would have said, “technology advances at about seven times that of human years – so we should think of it in ‘dog-years.’ A technology 3 years old is now an adult. One 10 years old is retired, and no longer in use.” Such a revision is a fair bit of work, so I will get on with it (before it, too, grows old). I hope you find this book to be all that you expect it to be.*

*Austin Lesea*

## Dedication

Peter Alfke: a good friend, and the engineer’s engineer.  
(1913 – 2011)

## Chapter 1 Introduction

### Overview and Review of Design Basics

With the integration of a large number of functions in integrated circuit (IC) devices, architectures to efficiently utilize these functions in end user systems were developed. The introduction of programmable logic arrays (PLA), programmable array logic (PAL), Complex Programmable Logic Devices (CPLD) and Field Programmable Logic Array (FPGA) devices evolved swiftly becoming the primary design method for most of today's digital designers. Designing with FPGA devices became the dominant method because the need for high speed and dense gate count capability lets designers get their systems to market astonishingly fast at a low initial cost. This book should serve to get beginners into programmable logic design, but it goes further adding dimensions of understanding for experienced FPGA designers, as well. The authors are pleased to present a book that takes you a bit deeper than most books of this nature. Indeed, we will be taking you for a tour "under the hood" of today's fastest growing FPGA families, the Virtex lines of FPGA devices.

In this chapter, we "set the stage" for the subsequent chapters by showing how a basic FPGA architecture is developed, and outline the essential requirements its design software must deliver. After that, succeeding chapters detail Xilinx commercial products and explain their inner workings. A lot of examples will be considered throughout the text, but typically we will restrict discussion to some key architectural facet for that example, as the myriad of design files for any substantial design constitutes a book in itself. First, let's quickly review simple logic design principles.

### Quick Logic Review

Because FPGA devices are the framework for logic design, it makes sense to review what logic design is basically all about. Here, we go into a quick review of logic design – combinational and sequential – with the intention of simply refreshing some of the basic ideas most readers learned from a college class or good book on that subject. Along the way, we will introduce some useful FPGA concepts.

Logic design is concerned with Boolean switching variables that can take on the binary values of zero or one. Operations on binary values fall within the subset of linear algebra called Boolean algebra, after George Boole, its developer. Logic functions of Boolean variables are described with algebraic expressions, logic gates and truth tables. They can also be described using abstract design languages such as VHDL and Verilog.

Figure 1-1 shows the basic gates that most readers are familiar with, and also shows each truth table. Simple logic functions of input variables can be built from the individual gates, where complex logic functions can be built from connections of several such gates. Specific attention was given in undergraduate texts to develop a standard method of creating general logic functions from two level Sum of Product (SOP) structures that translates directly from a logic function truth table.

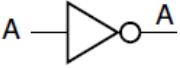
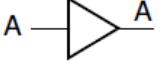
 INVERTER	 AND	 OR	 XOR																																																			
<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>A</th><th><math>\bar{A}</math></th></tr> </thead> <tbody> <tr> <td>0</td><td>1</td></tr> <tr> <td>1</td><td>0</td></tr> </tbody> </table>	A	$\bar{A}$	0	1	1	0	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>A</th><th>B</th><th>AB</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	AB	0	0	0	0	1	0	1	0	0	1	1	1	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>A</th><th>B</th><th><math>A+B</math></th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>1</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	$A+B$	0	0	0	0	1	1	1	0	1	1	1	1	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>A</th><th>B</th><th><math>A \oplus B</math></th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>1</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	$A \oplus B$	0	0	0	0	1	1	1	0	1	1	1	0
A	$\bar{A}$																																																					
0	1																																																					
1	0																																																					
A	B	AB																																																				
0	0	0																																																				
0	1	0																																																				
1	0	0																																																				
1	1	1																																																				
A	B	$A+B$																																																				
0	0	0																																																				
0	1	1																																																				
1	0	1																																																				
1	1	1																																																				
A	B	$A \oplus B$																																																				
0	0	0																																																				
0	1	1																																																				
1	0	1																																																				
1	1	0																																																				
 BUFFER	 NAND	 NOR	 XNOR																																																			
<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>A</th><th>A</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td></tr> </tbody> </table>	A	A	0	0	1	1	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>A</th><th>B</th><th><math>\bar{A}\bar{B}</math></th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>1</td></tr> <tr> <td>0</td><td>1</td><td>1</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	$\bar{A}\bar{B}$	0	0	1	0	1	1	1	0	1	1	1	0	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>A</th><th>B</th><th><math>\bar{A}+\bar{B}</math></th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>1</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	$\bar{A}+\bar{B}$	0	0	1	0	1	0	1	0	0	1	1	0	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>A</th><th>B</th><th><math>\bar{A} \oplus \bar{B}</math></th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>1</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	$\bar{A} \oplus \bar{B}$	0	0	1	0	1	0	1	0	0	1	1	1
A	A																																																					
0	0																																																					
1	1																																																					
A	B	$\bar{A}\bar{B}$																																																				
0	0	1																																																				
0	1	1																																																				
1	0	1																																																				
1	1	0																																																				
A	B	$\bar{A}+\bar{B}$																																																				
0	0	1																																																				
0	1	0																																																				
1	0	0																																																				
1	1	0																																																				
A	B	$\bar{A} \oplus \bar{B}$																																																				
0	0	1																																																				
0	1	0																																																				
1	0	0																																																				
1	1	1																																																				

Figure 1-1 Basic Logic Gates and Truth Tables

Ways to configure the gates into desired functions of greater complexity are needed. From Boolean principles (Table 1-1), two level SOP designs are defined. In the SOP format, a function is described as a network of AND gate outputs logically OR'ed together. The argument here is simply that Boolean variables occur in combinations, so only one combination may be present at any point in time. This corresponds to a single row on a truth table. To completely define the function, it is only necessary to describe the only combinations that result in an output of logic one. Under that situation, the zero values occur automatically.

1.  $A^*B = B^*A$
2.  $A + B = B + A$
3.  $A^*(B^*C) = (A^*B)^*C$
4.  $A + (B + C) = (A + B) + C$
5.  $A^*(B + C) = A^*B + A^*C$
6.  $A + B^*C = (A + B)^*(A + C)$
7.  $A^*(A + B) = A$
8.  $A + (A^*B) = A$
9.  $/A^*B^*C = /A + /B + /C$  DeMorgan
10.  $/A + B + C = /A^*/B^*/C$  DeMorgan
11.  $A^*0 = 0$
12.  $A + 0 = A$
13.  $A^*1 = A$
14.  $A + 1 = 1$
15.  $A^*A = A$
16.  $A + A = A$
17.  $A^*/A = 0$
18.  $A + /A = 1$

Table 1-1 Basic Boolean Logic Properties (+ = OR, \* = AND, / = INVERT)

With all such combinations identified by logically ANDing the appropriate variables, we create a set of partial results, where each AND gate output delivers the result of a single row of the function's output. It only remains to attach the AND gate outputs to the inputs of an OR gate to produce the desired function. Let's see an example, as a refresher.

#### Example 1-1 Three Input Majority Function

Table 1-2 describes a function of three variables A, B, C, where the output produces a one if the majority of the inputs are logical ones. This function may look familiar. If you rename the inputs  $A_i$ ,  $B_i$ ,  $C_i$  it is precisely the truth table for the “carry bit” of a 2 bit adder (where  $C_i$  is the carry in).

A	B	C	F(A,B,C)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 1-2 Majority Function Truth Table

This function is described by the equation:

$$F(A,B,C) = /A*B*C + A*/B*C + A*B*/C + A*B*C$$
Equation 1-1

There are four entries that result in a one, and the above equation first decodes the independent row numbers with an AND expression, then ORs them together. We have not minimized the results, but our goal is to simply define the basic underlying process. Our notation here is that \* is AND, / is inversion, and + is logical OR.

Using rows nine and ten from Table 1-1 (DeMorgan's theorems), we can rewrite Equation 1-1 as follows:

$$F(A,B,C) = /(/(A*B*C) */(A*/B*C) */(A*B*/C) */(A*B*C))$$
Equation 1-2

Figure 1-2 shows the two solutions drawn with logic gates.

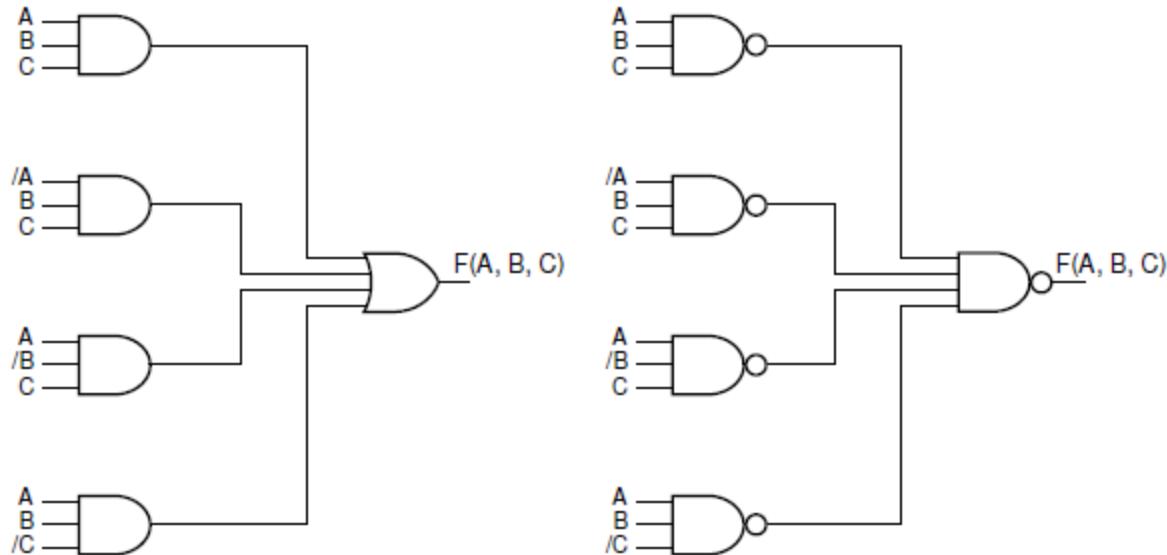


Figure 1-2 Majority Function with AND/OR and with NAND/NAND

### Beginning Development of a Logic Cell

The result on the right in Figure 1-2 uses only one kind of gate. An FPGA will be a framework of building blocks that create a wide range of designs. In order to identify the key qualities of the building blocks within it, we seek a way to describe as many logic functions as possible, with a single effective building block. It may not be evident, at this point, that small, uniform building blocks are important, but consider the densities attainable by today's memory cells. Memory cells have tracked Moore's Law, for decades. We wish to attain densities that track this kind of evolution. Memories taught us that density and regularity are critical. Gate array techniques taught us that functionally complete logic cells are required. NAND gates are functionally complete. Let's briefly discuss gate arrays.

Gate arrays come in a lot of forms, but the simplest is the sea of gates, which is a misnomer. Most gate arrays are really a "sea of transistors". Early gate arrays consisted of columns of alternating N-channel and P-channel transistors that could be wired together

with metal creating, NAND gates (Figure 1-3). Other gates could be built, but often, they were made by cascading NANDs. The NAND is a good choice because it has the quality of being functionally complete. Functional completeness means any logic function can be constructed from gates of a single type. For instance, Equation 1-1 requires three types of gates – AND, OR and INVERT.

Equation 1-2 requires only one type of gate – NAND gates. You may recall this, but probably haven't considered its importance.

With FPGA devices, we seek to create a framework of logic cells to build a very large number of user designs. It would be desirable if the fabric for that framework scaled like memories over time, but had the logic flexibility of gate arrays. These are noble goals, to be sure. Incidentally, Figure 1-3 shows the basic, simple structure of a 2 input NAND gate constructed from four transistors. This became the unit of density for gate array designs, and came to haunt FPGA marketers desiring to compare the two, over time. Also note the functionally complete demonstration shown on the right hand side of Figure 1-3. Building an inverter, AND function and OR function is all that is needed.

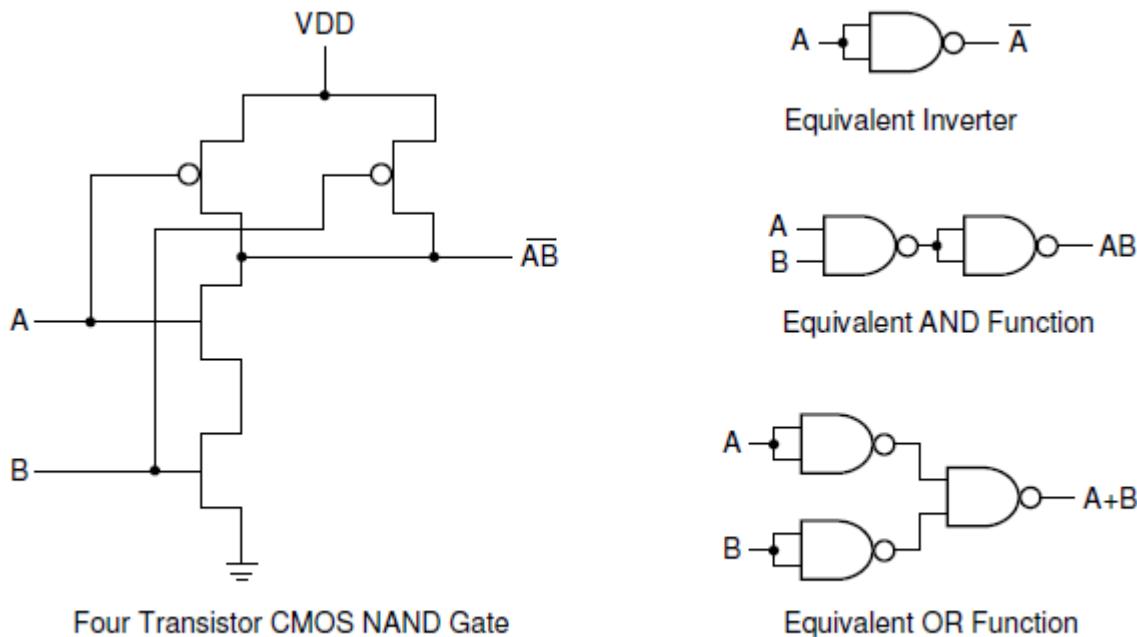


Figure 1-3 CMOS Two Input NAND Gate and demonstration of Functional Completeness

A minimum target quality for a “fabric” logic cell is to be functionally complete. In fact, one developer took that exact level and attempted unsuccessfully to create FPGAdesigns with it – three times. Clearly, simply building up NAND gates wasn’t quite enough. Let’s go further.

Figure 1-4 shows another simple design, with four data inputs ( $D_3, D_2, D_1, D_0$ ), two control inputs ( $S_1, S_0$ ) and a single output, DataOut.

This structure is called a 4:1 multiplexer (MUX). The idea behind a MUX is simply that data is present on the four inputs, and we choose a single value to be delivered to the DataOut pin by “selecting” it with a binary combination assigned to the signals  $S_1$  and  $S_0$ .

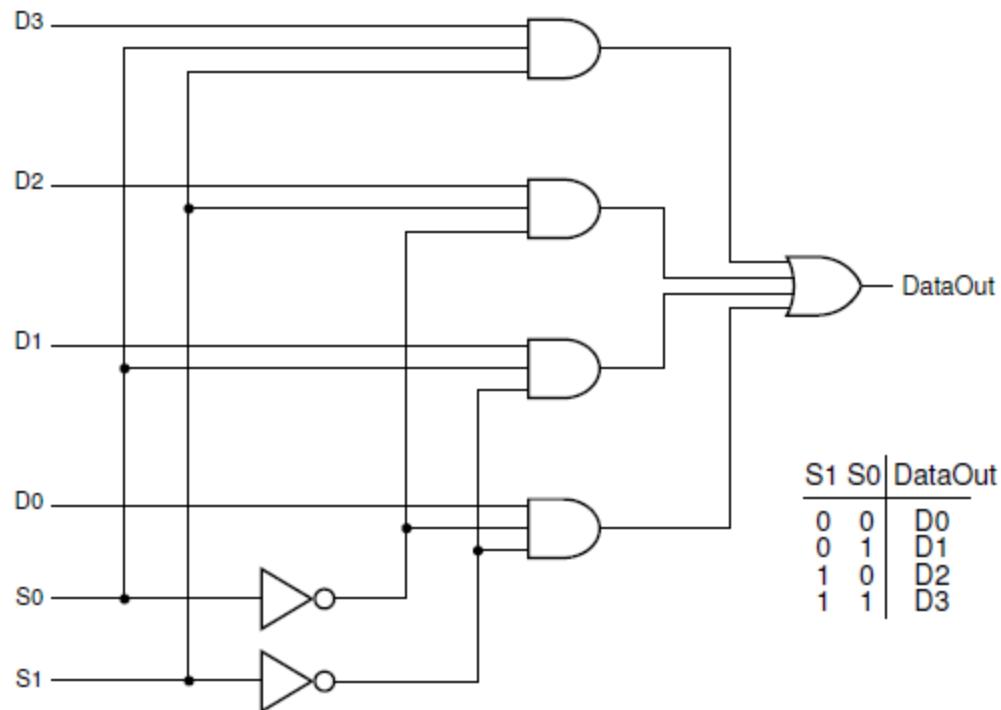


Figure 1-4 4:1 Multiplexer

Multiplexers are key building blocks of digital design, and can be constructed very efficiently out of transistors in a minimum area on a chip. Note the truth table for the 4:1 MUX shown in Figure 1-4. We simply show the combinations of S1 and S0, and the “D” value that appears on the DataOut line. This allows us to make a simple discovery. If we assign constants to the D3, D2, D1 and D0 inputs, we can create the truth tables for the basic logic gates shown in Figure 1-1. The AND function is found by assigning a logic one to the combination where S1S0 = 11 and zero every place else. OR is found by assigning logic zero when S1S0 = 00 and one every place else. The inverse can be handled by tying S1=S0, and putting a logic one on the output when S1S0 are zero, and a logic zero on the output when S1S0 are at one. Figure 1-5 shows that there are sixteen different functions of two input variables.

A	B	F1	A	B	F2	A	B	F3	A	B	F4	A	B	F5	A	B	F6	A	B	F7	A	B	F8
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	1	0	1	0	1	0	1	0	1
1	0	0	1	0	0	1	0	1	1	0	1	1	0	0	1	0	0	1	1	0	1	1	0
1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	1

A	B	F9	A	B	F10	A	B	F11	A	B	F12	A	B	F13	A	B	F14	A	B	F15	A	B	F16
0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1
0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	1	0	1	0	1	0	1	0	1
1	0	0	1	0	0	1	0	1	1	0	1	1	0	0	1	0	0	1	0	1	1	0	1
1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	1

Figure 1-5 Sixteen Functions of Two Binary Variables

This is an interesting capability. If we create a logic cell based on the ability of a multiplexer to produce all possible logic functions of its select inputs, we have what is known as a universal logic function. Are universal logic functions functionally complete? Yes. They can create the NAND function, which is functionally complete, and that is sufficient to verify that they meet the minimum requirement for a uniform logic fabric.

Given that this structure has some good qualities for producing logic functions, how might we add programmability to the multiplexer? The answer is to assign the inputs not as fixed constants, but as programmable constants. One such method is flip flops. By attaching flip flops to the inputs of the multiplexer, we have the ability to assign logic values to the flip flops. These values become the particular truth table row contents selected by the multiplexer. We are literally placing the truth table of the function we seek into the flip flops, to create the logic function. Figure 1-6 shows the flip flops attached to the multiplexer, in the guise of a shift register, streamlining the bit delivery with minimum extra connections.

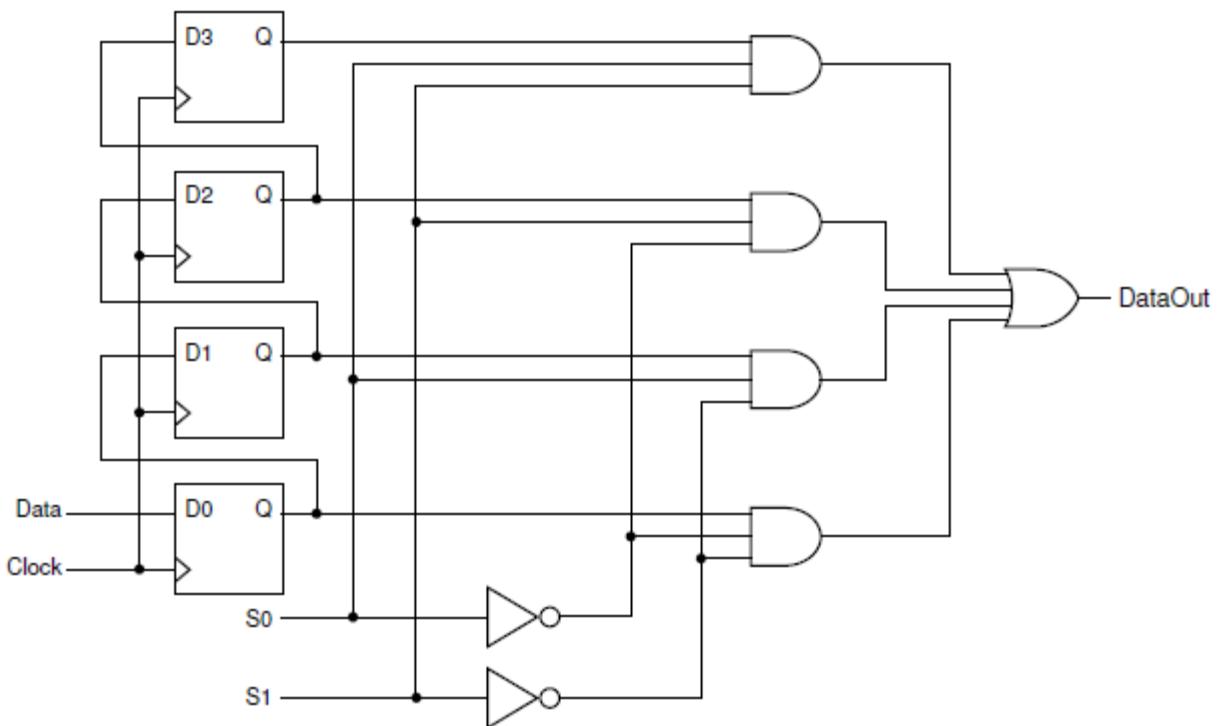


Figure 1-6 Adding Flip Flops to the Multiplexer

It is interesting that to produce a two input AND function with the above multiplexer, we require four flip flops, four AND gates, two inverters and a single four input OR gate. Such is the price for programmability! Based on this observation, it is important to make everything as small as possible, usually adding the benefit of making them fast, too.

If we take this structure as our building block, we must build everything up from two input functions. This can impact performance dramatically. Stacking two input gates to attain larger functions results in *pyramiding*, whereby a partial logic function is created and the partial result feeds into more gates to create the complete function. It would be nice to

create a logic cell capable of doing the most commonly occurring logic operations in a single cell. To arrive at this, statistical studies are needed.

Most of today's logic cells are found to be efficient with a sixteen or sixty four flip flop/multiplexer structure. When combined with the sixteen flip flops, the entire structure is frequently called a *lookup table*, or LUT4 (look up table with four inputs). Recent devices use a LUT6 (6 inputs for sixty four flip flop/multiplexer structure. LUTs are also sometimes called *function generators*, or *universal logic functions*. Our proposed LUT can form any logic function of four (or six) input variables applied to the select inputs on the multiplexer. This logic structure appears to be quite effective for building up combinational functions. We can envision a "sea of LUTs" that may be configured to become the pieces of our logic designs, but what about storing logic variables and building state machines?

#### *Additional Logic Cell Requirements and Configuration Memory*

Figure 1-7 is a classic NAND gate constructed flip flop. It clearly shows why the gate array world typically declares a flip flop to be valued at six to nine gates. Note that two input NANDs are one gate, 3 input gates are 1.5 gates, and we have 6 times 1.5 gates, or nine equivalent gates here. If, for every copy of the NAND gates shown in Figure 1-7, we had to insert a copy of Figure 1-6, we would want to avoid sequential design at all cost!

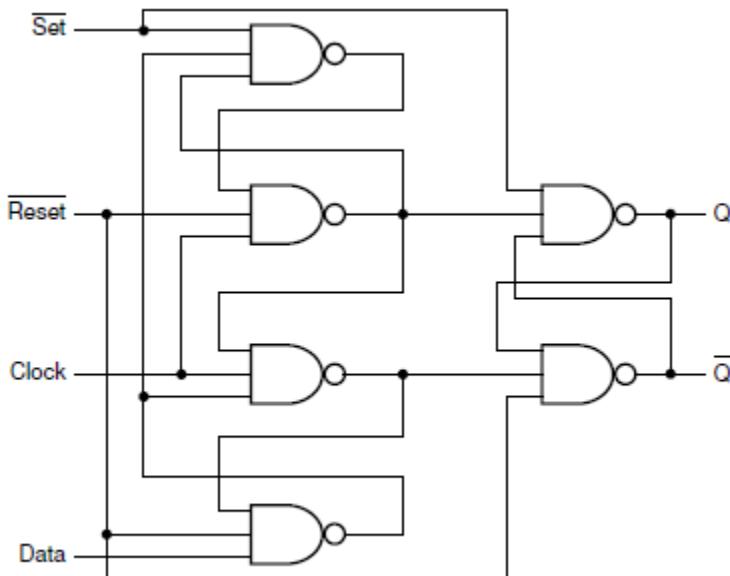


Figure 1-7 Six NAND gate Edge Triggered D Flip Flop

As suggested earlier, we seek the greatest flexibility with the minimum area, if possible. What makes the most sense is to efficiently create a minimal function flip flop, and include it in our LUT based building block. Figure 1-8 shows a more reasonable D type flip flop structure that we can attach to the LUT structure. We can't just tie the LUT output to the D input of the flip flop, as that would restrict all of our designs to be totally sequential. It is best to permit a choice, on a cell by cell basis of whether a combinational function is needed, or a sequential one. Such a "choosing" device is our old friend, the multiplexer –

only, this time we need but a 2:1 multiplexer.

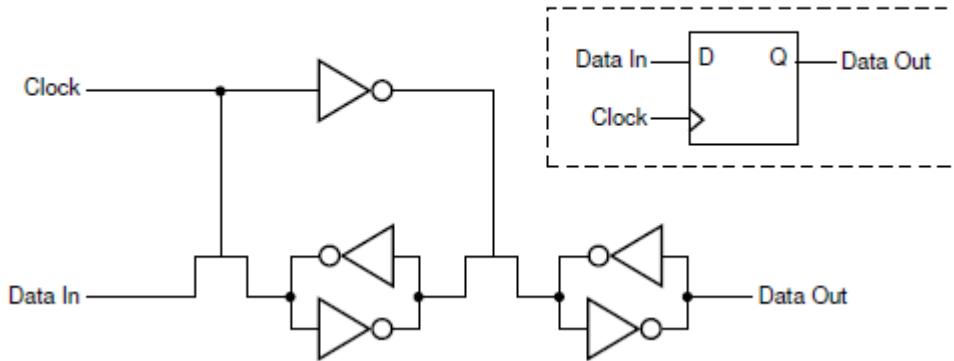


Figure 1-8 Smaller D-type flip flop

At this point, let's consider the proposed cell. It has four logic variable inputs, and a clock input for the D flip flop. We need sixteen storage cells within it dictating the logic function through the multiplexer and an additional bit selecting whether the function is combinational or sequential – a “mode select bit”.

The 16 storage cells that determine the LUT function are clumsy to describe, so we let's show a simpler cell where the 4 input variables, the clock and the output of the cell are all we display. The 16 storage cells are strung together as a shift register, and each set of 16 cascade with the mode select bit to other such cells throughout the whole chip, becoming the configuration memory. Figure 1-9 shows the complete picture. Note the trapezoidal multiplexer with a visible selection input. A fairly standard symbol convention at Xilinx is that trapezoidal multiplexers have their select inputs coming from configuration memory. Rectangular multiplexers have their inputs explicitly shown, and as a rule are dynamic during user operation. Also, note the shorthand symbol on the right hand side of Figure 1-9, where we collapse all the detail out of the figure on the left, to a little box labeled logic cell. The abbreviation serves us well, with more complex diagrams. We maintain the clock and output as bold, for continuity here. In the “logic cell” version, the configuration memory access is not shown. The configuration memory, although not shown is very important, because that is ultimately your design! We arrived at five inputs and one output on our logic cell. The inputs are labeled W, X, Y, Z and CK. The output will sometimes be called “output”, Logic Cell, or simply “F”, the function.

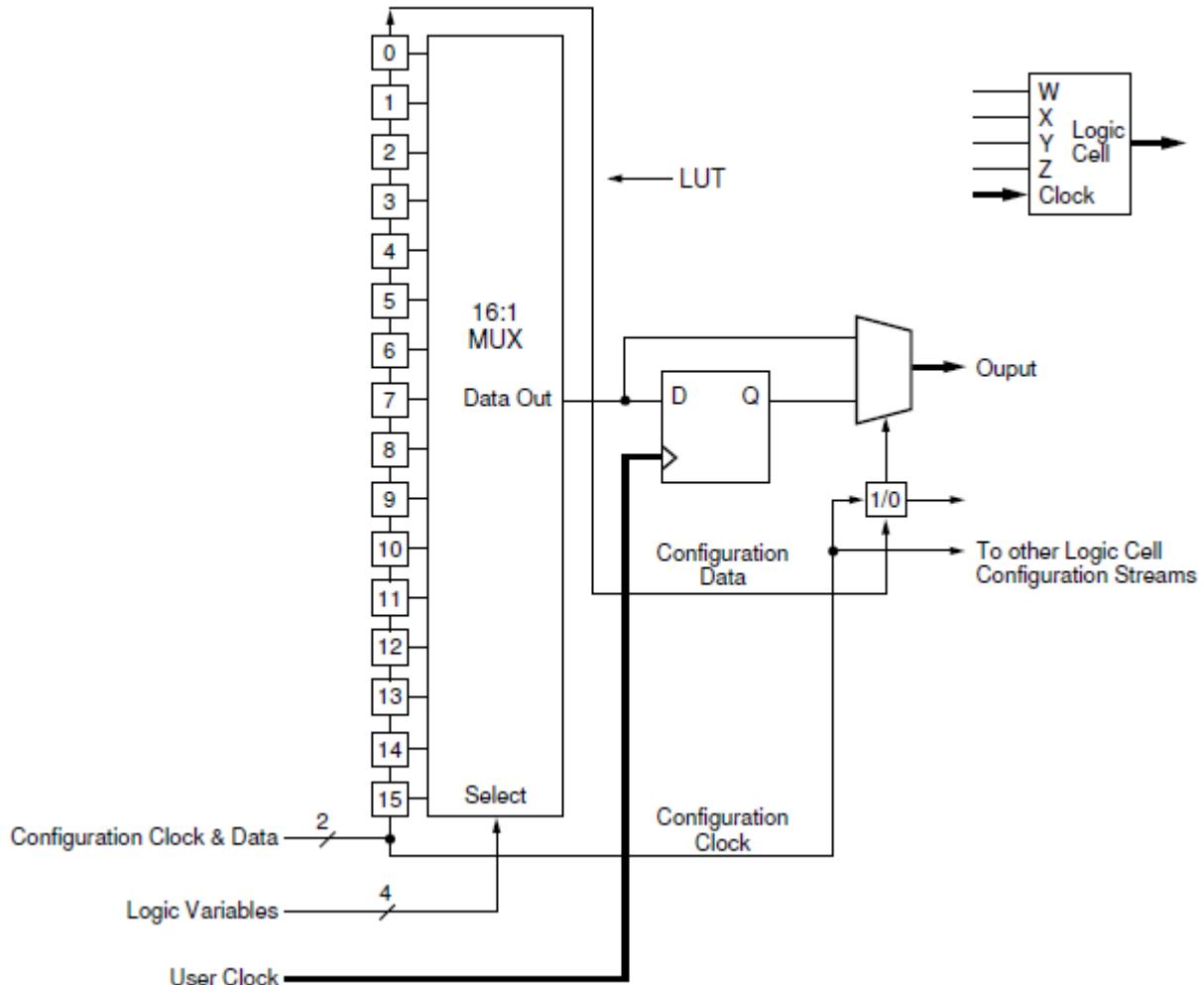


Figure 1-9 A Basic Logic Cell Structure and Symbol

#### Connection Framework and the Complete Array

The configuration memory is a large shift register that permits configuring every LUT based logic cell in an array constructed from such cells. What we haven't done yet, is figure out how to make logic connections between the various logic cells. For this, we need a connection framework. There are many options on how to build a connection framework, but in order to get at a basic, complete FPGA structure, let's pick one of the simplest ones – rows and columns of metal lines. Figure 1-10 shows such a framework. We have five inputs and one output for our logic building block, or logic cell. Configuration inputs don't count. We would like for each logic cell to have the potential to connect to any other logic cell, in general. It is recognized that any given design will not need to have every logic cell be connectable to all other cells, but the potential for such connections is desirable. A tradeoff must be made. One choice is to install two horizontal and four vertical lines structured as a grid, and embed the logic cells into that grid.

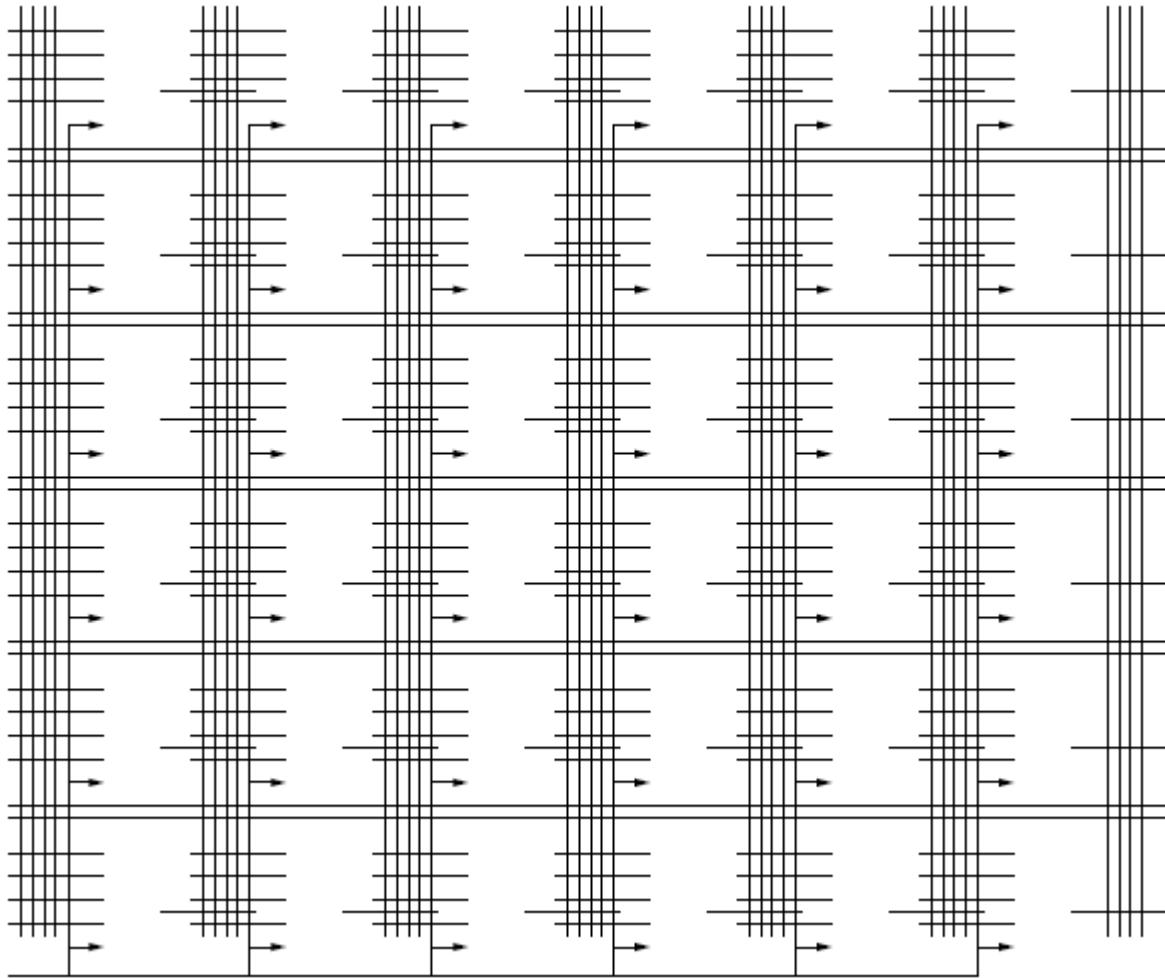


Figure 1-10 Connection Framework

At this point, let's experiment to see if it works, and makes sense for future designs. The connection framework must provide connections for the logic cell five inputs and single output. Other connections are possible, some better than others. Figure 1-11 shows the combined structure, with I/O pins added into the connection framework. We haven't settled yet on how connections are made between the logic cells and the connection framework, but let's simply say that where horizontal or vertical lines cross, there is a connection possibility by means of some technology. For simplicity, let's call it "dot" technology. We can define it better, later.

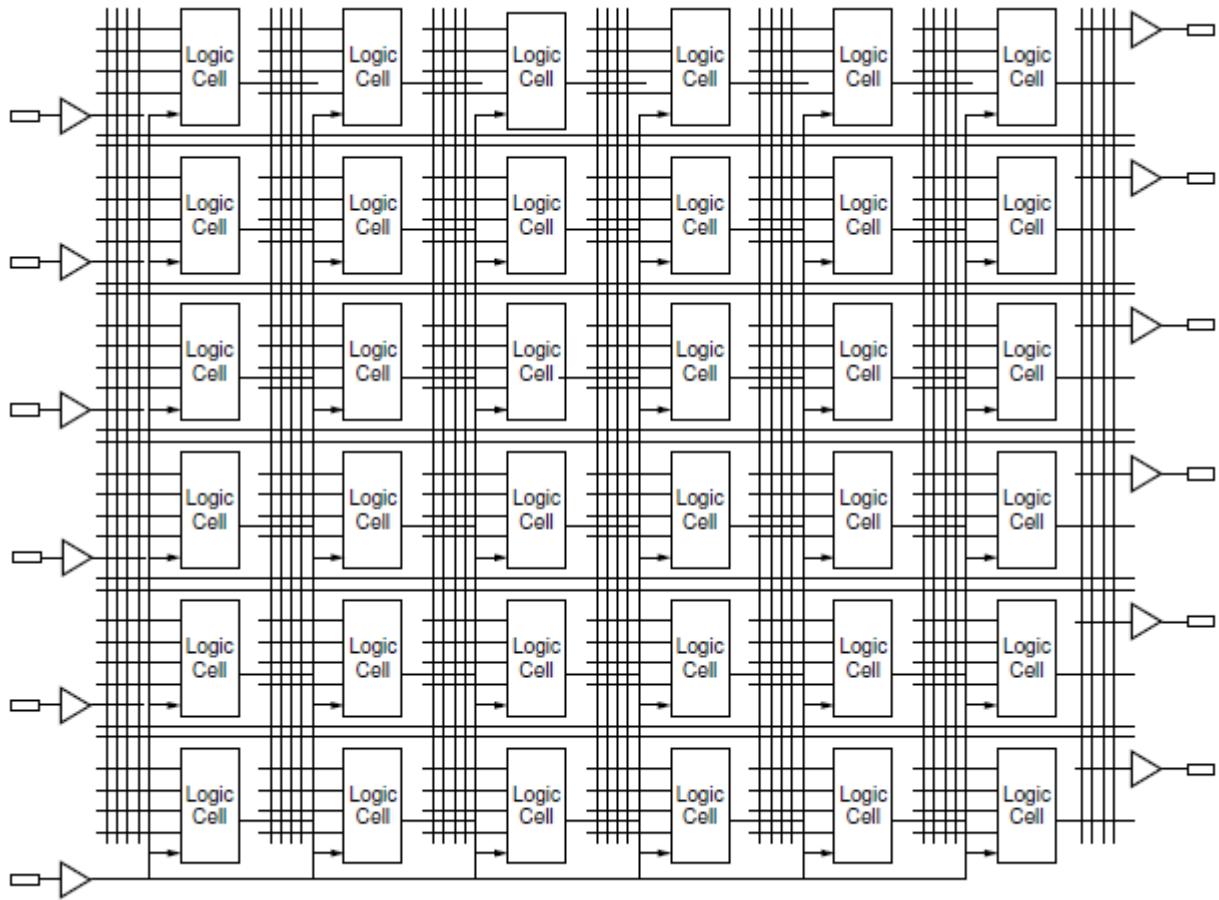


Figure 1-11 Logic Cells embedded in the Connection Framework

Taking the device shown in Figure 1-11, let's do the experiment of connecting up a simple sequential design – a shift register, as a test case for the architecture. It also gives us a feeling for what the design software must do.

#### Example 2 - Building a Shift Register in the Trial Architecture

A shift register is a simple state machine, mentioned earlier in Figures 1-6 and 1-9. We show just the two input signals DataIN and CLOCK, and the output signal DataOUT with the connected four flip flops in Figure 1-12.

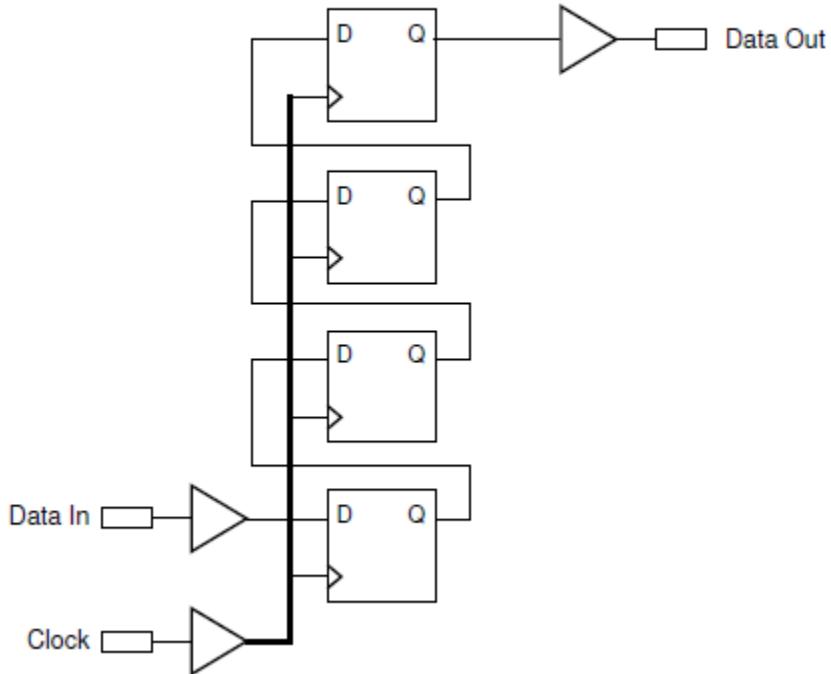
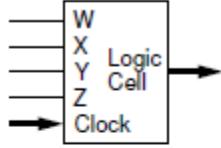


Figure 1-12 Simple Shift Register

In this situation, we want to build the shift register into the programmable framework of Figure 1-11. We must configure four logic cells to behave as simple D flip flops, assign an input to DataIN, attach the CLOCK signal to the Clock net on Figure 1-11 and assign an output to a pin on the right. Let's identify a function that attaches a logic cell input to the D flip flop input, as a direct connection. Examine Figure 1-5 entry F6. This table shows that the B variable matches the output in every row. Expand this idea to the four input logic cell, and there will be a function so that the input matches the output. For F6 in Figure 1-5, this is the B variable. Figure 1-13 shows how the four input logic cell may be configured to do a similar thing. In this case, the 16:1 data multiplexer is loaded with the values in the table in Figure 1-13, and the logic cell Z variable input is the connection site. Note that the trapezoidal multiplexer shown in Figure 1-9 must select the flip flop Q output to be the logic cell output.

WXYZ	F=Z
0000	0
0001	1
0010	0
0011	1
0100	0
0101	1
0110	0
0111	1
1000	0
1001	1
1010	0
1011	1
1100	0
1101	1
1110	0
1111	1



The logic cell symbol is a rectangle divided into two sections. The left section has four input lines labeled W, X, Y, and Z, and one output line. The right section is labeled "Logic Cell". An arrow points from the output line of the cell to the right.

Output MUX selects Flip Flop

Figure 1-13 Logic Cell Symbol for the Shift Register Cells

Now, all that remains is to see if we can assign the cells and “connect the dots” to get a design. Figure 1-14 shows one attempt at building the shift register. Let’s note a couple of things. The four lower left Logic Cells have been renamed to build the function  $F=Z$ , so they simply pass the  $Z$  input of the Logic Cell to the  $D$  input on the flip flop in the cell. The CLOCK signal (external) is assigned to the global clock net within the array, and actually visits each Logic Cell, even the unused ones. DataIN is assigned to the input pin right above the CLOCK pin, and the first little “dot” is labeled “1”, with each consecutive dot labeled in order, somewhere in the neighborhood of the dot.

This little example shows manual assignment of the functions to the cell array, which is called *placement*. The assignment of the little “dots” is forming the connections within the logic array, or defining the paths the signals will take as they form the target function. Forming the connections is a process called *routing*.

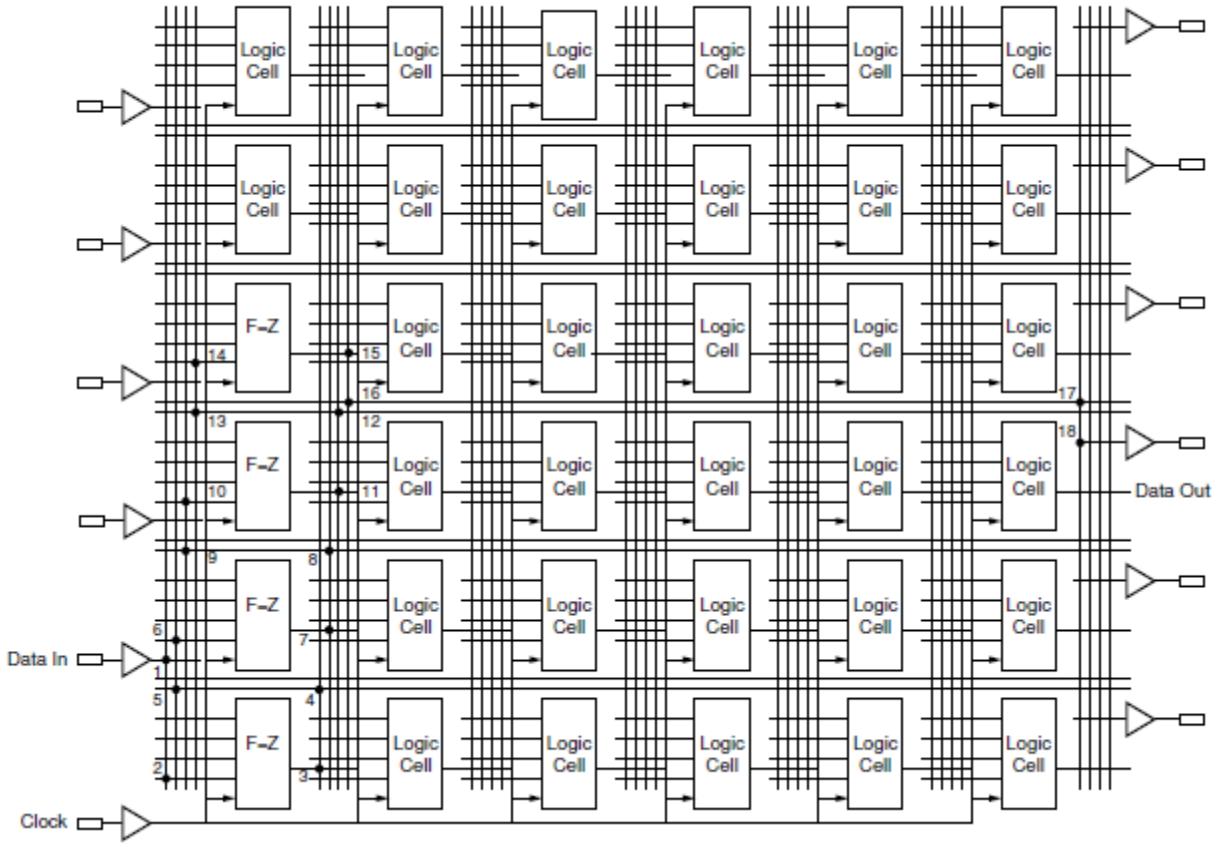


Figure 1-14 Shift Register Assigned and Connected into the Logic Array

We actually introduced a simple design rule into this process. For instance, wherever you see a “dot” on a column, you will discover that it only has one signal source on that column. It is possible to deliver the signal to more than one logic cell input, if needed, but never are two sources allowed to drive the same line. The signal on the input buffer for DataIN drives “dot” 1, which in turn connects to “dot” 2, then enters the bottom left Logic Cell labeled F=Z. The output of this cell drives the first column to the right of the cell, connecting with “dot” 3. This signal heads north to “dot” 4, then west to “dot” 5, then north to “dot” 6 and enters the second Logic Cell labeled F= above the first. This pattern repeats, until the fourth cell delivers its output to “dot” 15, which heads south to “dot” 16, then east, across the array to “dot” 17. Finally, a column connection is attached to “dot” 18, connecting the signal to the output buffer. The output buffer drives the output pad called DataOUT.

Examining the connection columns on the left side of the array, we discover that all four were used to make connections. Had we chosen to park a fifth shifter cell above the fourth, we would have been unable to connect it with that placement. We would have had our first encounter with routing *congestion*.

We placed and routed the last example manually, and managed to lock two cells in the top left side of the array so that they couldn't form any additional logic unless it used either the single DataIN input, or the three feeding back flip flops. Let's see what happens when

we arrange the cells along the bottom of the array, and attempt to do the same basic shift register function.

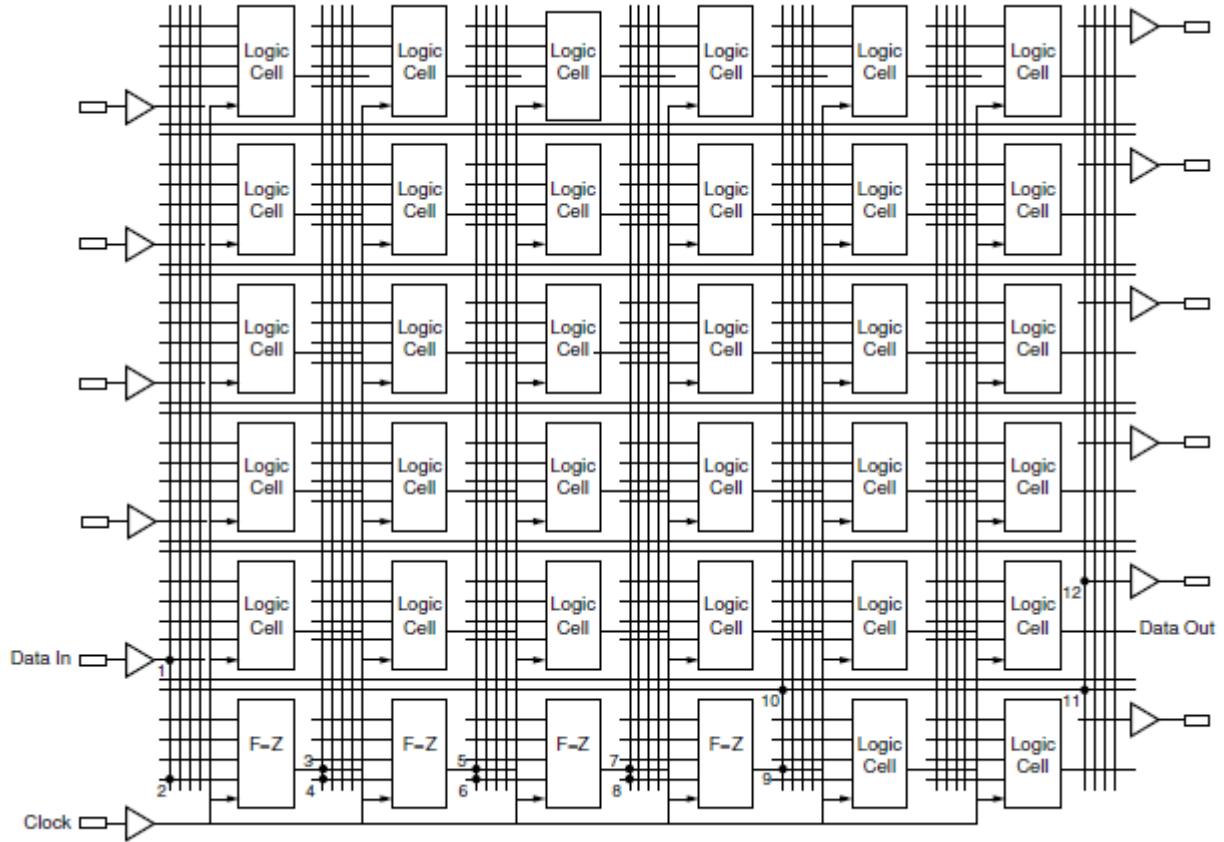


Figure 1-15 Alternate Version of the Shift Register

Similar to Figure 1-14, we deliver CLOCK to all cells, and DataIN arrives on the input right above it. This time, the four bottom left cells are all configured as simple D flip flops, so the Data IN signal gets a “dot” at 1, another at 2, enters the first cell, connecting its output to the next cell’s input on “dots” 3 and 4, proceeding similarly to the right. After four cells are connected, the output on “dot” 9 heads north to “dot” 10, where it connects to a horizontal line and connects with “dot” 11 onto a vertical column heading north, again. To go out, “dot” 12 makes the connection to the output buffer driving DataOUT.

Comparing the two solutions, the first had congestion on the leftmost four vertical connecting columns of the connection framework, locking out the top two cells in the first cell column. The first arrangement used exactly the same number of cells, and required eighteen “dots” to arrive at the selected output site. The second arrangement distributed the grid usage more uniformly across the chip. Taking only twelve dots and four logic cells, the second solution used up one vertical interconnect in each of six groups of four vertical connection structures. The remaining unused vertical connections are available for other changes as the design grows. Note that it would be very easy to add another two cells at the bottom of the array, to expand the shifter, if needed. We would like design software to make choices similar to the version in Figure 1-15, where we will have options to easily

make future edits and avoid congestion.

## ► Design Software Basics

Design software must do a large number of tasks. Among them are the following:

1. Capture the design
2. Eliminate redundant logic
3. Provide a framework to communicate constraints
4. Slice up the design and make assignments to logic cells
5. Place the logic cells at appropriate locations
6. Connect the logic cells to functionally emulate the original logic
7. Verify that the whole process occurred without error

That's a fairly hefty set of tasks, typically done by several software modules designed to cooperatively exchange data, as each module performs transformations on the design. The ultimate result of all modules is to finalize the transformation into the logic cell array configuration bitstream. It is worth recognizing that the programmable device, itself, is a design constraint. We will attempt to place arbitrary designs into a well-defined number of logic cells and make connections among them to form the final design.

As another example, let's consider some of the above list, while pondering how to "pour" Figure 1-16 into Figure 1-11.

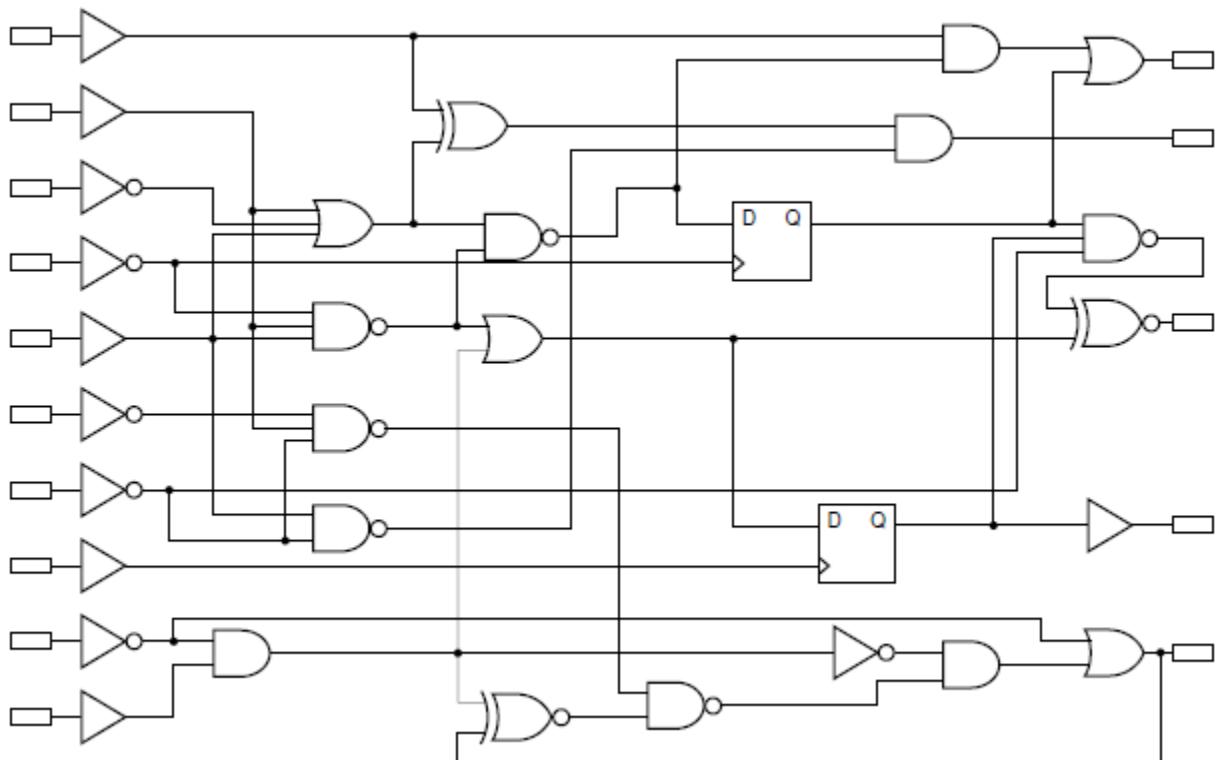


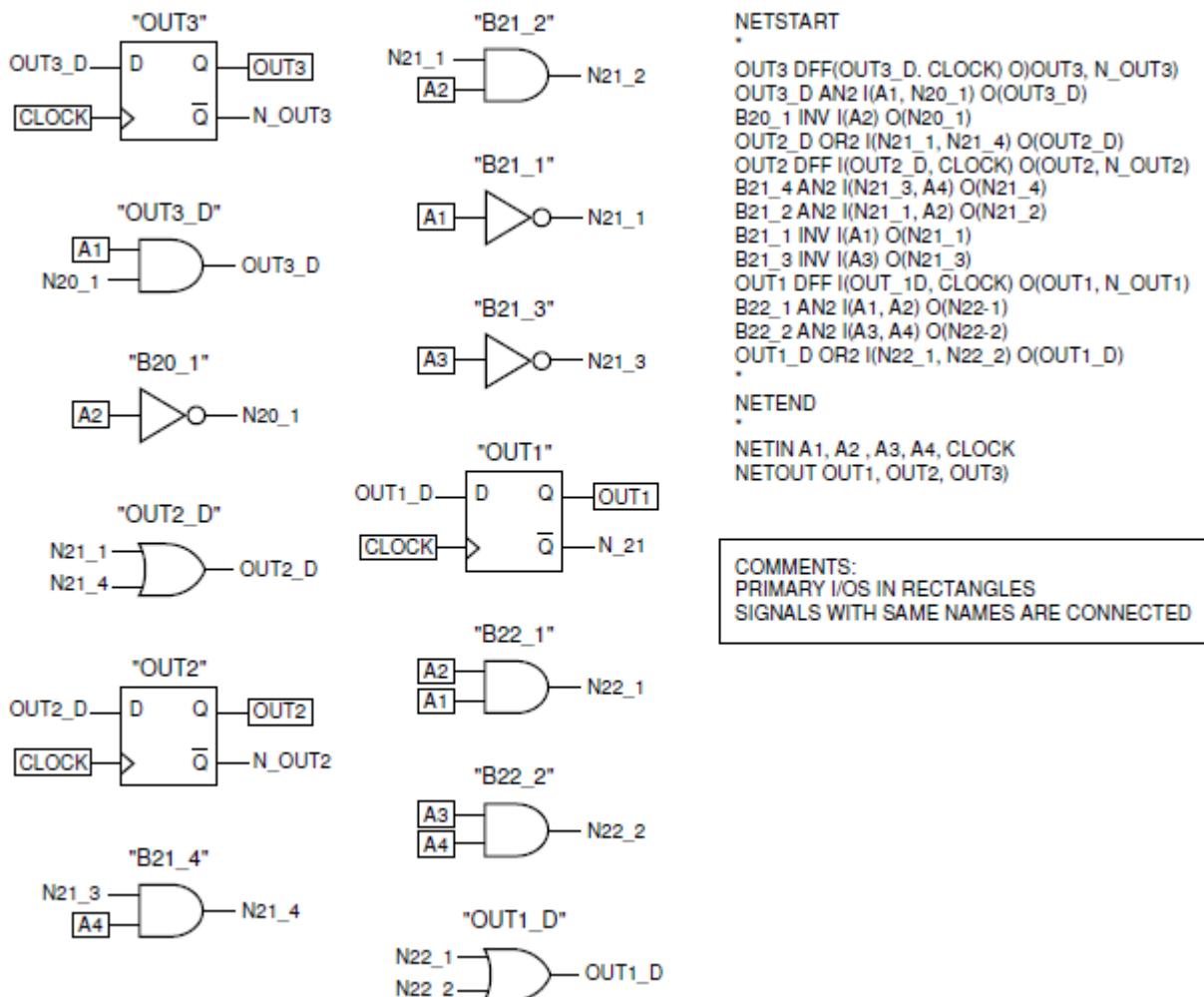
Figure 1-16 Some Weird Circuit

For starters, Figure 1-16 is just some logic. We have absolutely no idea about what it is

supposed to do. We don't know which aspect is more important to get correct than any other aspect of it. It looks like a mess. This is a good observation, because it puts our point of view right in the same perspective as would be had by design software (if it had a point of view)! Design software cannot determine what you are trying to do. It cannot correct your design to better embody your intention. It can only deal with what you have created. In very many ways, design software is like a computer compiler. You create an abstract set of actions called a "program" (think C, C++ or JAVA) and a compiler turns your abstract commands into a set of bits that are understood at the binary level (machine language) by the target computer. In fact, the design software is often referred to as "compiling" the design. Like computer compilers, FPGA compilers deal with data structures that are symbolic program representations. For FPGA compilers, most of the data structures are called netlists.

## Netlists

Netlists are often text files that explicitly describe a circuit in terms of functions and connections, at some level. Gate array netlists ultimately resolve your design into connections between transistors. FPGA netlists ultimately resolve your design into library items that neatly fit into logic cells. Figure 1-17 shows a netlist for a logic structure (note: not Figure 1-16).



## Figure 1-17 Logic and a Netlist Representation

Examining Figure 1-16 and pondering whether it might “fit into” Figure 1-11, opens up a set of choices, decisions and optimizations that must follow. First, the design software must make “big” decisions like:

1. Are there enough input and outputs available on the target FPGA to accommodate the user’s design?
2. Are there enough logic resources (logic cells) available to accommodate it?
3. Are there enough global resources (clocks, resets, sets, tri-state controls)?
4. Are there enough connection resources (vertical and horizontal lines)?

The list can grow, but those four are a good starting point. Number one is resolved by simply inspecting Figure 1-16, and observing that it has eight logic inputs, two clock inputs and six outputs. Figure 1-11 only has five inputs, a clock and 6 outputs. Hence, we don’t have a match. If another part with the same general programmable logic cells / interconnect were available with more I/O resources, it might be a candidate for fitting. Items 2, 3 and 4 are often called the *fabric*. Figure 1-11 appears to have sufficient fabric.

If we assume that Figure 1-16 did not fail the I/O test, then we might be asking if Figure 1-11 has enough logic resources to satisfy the needs of the design. That is a bit tougher. First, we need to make sure that Figure 1-11 is described with the absolute minimum number of logic cells. If we can minimize the design, to reduce logic requirements, then the probability of getting the design to fit a particular FPGA is greater. This has the added effect of producing a design that might fit on a smaller part. As a rule, smaller parts are cheaper. Slower parts are also cheaper, so a general direction to head – for cost – is to select the slowest, smallest part that satisfies the needs of our design. If we are successful, we have the added advantage many times of having a larger part, and/or a faster part available to use, should we need more logic or more speed at some future time. Minimizing your design before starting to place and route has lots of benefits, so we are strongly motivated to reduce it as much as possible. To accomplish that, we use all possible means. By this, we mean standard Boolean reduction, Karnaugh maps, multiple function Quine-McCluskey minimization, and finally, netlist reduction. The only requirement is that we maintain the intended functionality.

### Netlist Reduction

Netlist reduction is present in many tools like, Verilog and VHDL. It consists of a set of substitution results that are heuristic in nature, and have evolved over years of observing equivalent circuit substitutions. Usually, the heuristics are a set of rules and a sequence for applying them. Here are a few simple ones, having big rewards, but are distinct from classic Boolean reduction:

1. eliminate gates that are driven, but whose output is unused
2. eliminate gate inputs that are tied to constant “1” or “0”
3. eliminate inverters attached to the Q output of flip flops, if /Q is available

Surprisingly, rules like these have big payoffs. For instance, using number 1, it is possible to identify a gate, recognize that its output goes “nowhere”, remove it, then notice that

other gates which were only driving the now eliminated gate, have unused outputs and may also be eliminated. This can sometimes continue until big sections of circuitry vanish. The reduction process can be sometimes thwarted, if a cross coupled gate set is in the mix, and the coupled gates are circularly connected, but ultimately the whole thing goes nowhere. Really good netlist optimizers must expand the scope of the reduction tool to identify “big” sets of gates that do nothing.

Rule 2 basically reduces gate inputs, and in an environment with four input logic cells, this is very important.

Rule 3 simply observes that the needed logic variable, /Q already exists. Typically, inversion tends to be “free”, as LUT structures are good at dealing with the variable or its complement, just as nimbly. Other technologies like gate arrays, can’t take advantage of that type of reduction.

The term “netlist reduction” is seldom used, but is more accurate than the more frequently used term netlist optimization or netlist minimization. Really good netlist optimizers can have hundreds of rules and the results can be staggering. They are worth their purchase price.

### Function Assignment or Mapping

After reduction, the next step is to identify groups of logic functions to be combined into the logic cells of the target architecture. This can be an iterative process, because the choices made may not - in general - be unique. Let’s examine Figure 1-16 with that in mind.

First, scanning the diagram, we would like to pack as much into each logic cell, as possible. Assigning a logic cell to a simple inverter or to a two input gate is wasteful, if a tighter packed cell would cover the same logic. One strategy might be to scan through the design first, and find sets of gates that have four distinct inputs resolving to one output. Remove those from the circuit, then scan the remaining circuit for three input groups with one output. Remove those, then scan for two input groups, etc. If requirements are found for more than four input logic functions, then pyramiding must be used.

The best bet is to pack the design into the fewest logic cells possible. Sometimes, the software may have to do things like duplicate logic, but the slightly increased design may ultimately fit into fewer logic cells than the directly mapped design. Figure 1-18 shows some groups identified within the structure of Figure 1-16. Two groups cleanly fit a single four input cell, and one needs further “massaging” as it drives multiple outputs. Multiple output groups suggest using logic duplication.

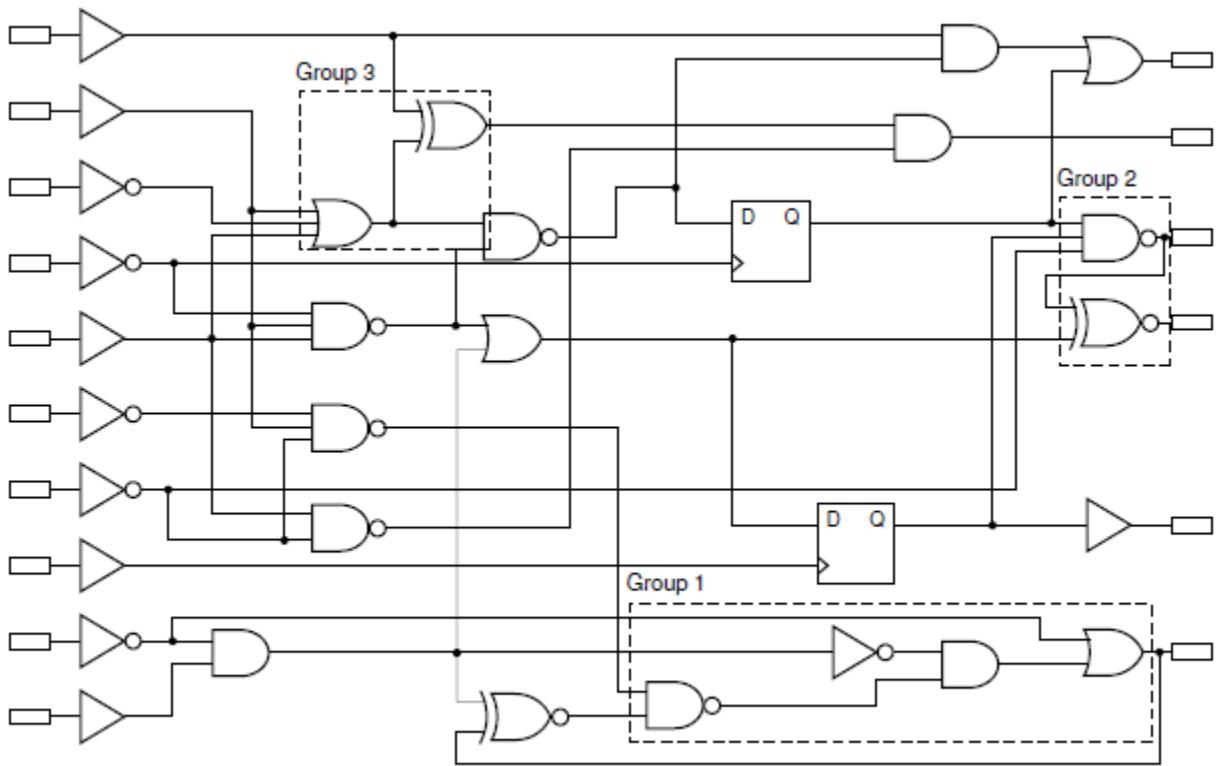


Figure 1-18 Weird Circuit with some Groups Identified

Figure 1-19 shows one way that Group 3 may be handled, by logic duplication. Group 3 has two outputs, but our 4 input logic cell has only one. Both outputs require connection to the three input OR gate, to create their correct function. Figure 1-19 shows one way to accomplish that, by identifying two logic cells, where each is fed by the common three inputs to the OR gate, and each needs to add one additional input on two logic cells to create the equivalent functions. This may not be the way that the Xilinx design software ultimately handles this situation, but shows one approach to the problem.

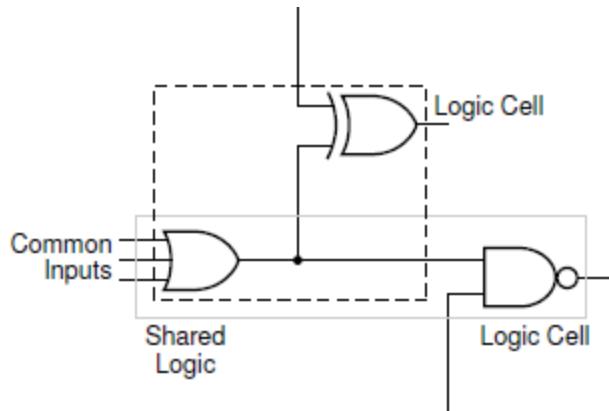


Figure 1-19 Possible Handling for Group 3 with Logic Duplication

Once the groups are identified, the logic may be placed into logic cells. After the whole process is accomplished, a minimum set of logic cells should result. If that number is less

than or equal to the target logic array, the design may be feasible to place and route. Routing completion depends on a good placement, and available capacity in the target FPGA.

From the I/O count, we know that Figure 1-16 will not fit onto the Figure 1-11 logic array. So, there is no point in pursuing that further. However, it is illustrative to continue the process a little further. Figure 1-11 has 36 four input cells. Three groups in Figure 18 will occupy only a single cell, each. There are about 10 remaining gates and flip flops, so it is very likely that the design could easily be placed successfully onto the target logic array.

Whether or not it could be connected is questionable, due to the very limited routing resources. We can't describe the method of arriving at a successful placement and routing here, but we can at least outline the next steps.

### Placement and Routing Strategy

Placement and Routing strategy fall under the general title of physical design, and are treated in depth in Chapter Four, for the Virtex FPGA families. However, in broad terms, the problem is two-fold. First, find an arrangement of logic functions in the FPGA that satisfies a set of design criteria. Then, once one is found, attempt to connect the placed functions so the circuit connects and delivers a solution within the user's timing requirements. Some common ideas that occur here include the following:

1. functions that connect together should be placed nearby, if possible
2. functions that attach to I/O pins should be located near them, if possible
3. functions which must meet specific timing requirements or *critical paths* should be placed and routed, first
4. specific user constraints (such as pin assignments) should be met, if possible

Physical design is frequently modeled with nonlinear mathematics formulated for optimization. Historically, these have often involved the methods like *simulated annealing*, which came out of the development team that brought us the Atomic Bomb, during World War II. It's been around quite a while, withstood the test of time, and through continuous evolution it has been adapted to many different applications. We discuss placement and routing as well as simulated annealing more, in Chapter Four.

### Constraints

The FPGA framework itself is a constraint. However, a number of additional issues can block achieving the desired performance of a design embodied in an FPGA. First, there is the speed limit associated with connecting up logic cells. This includes the metal between the cells and the interconnecting sites ("dots" in Figure 1-14). It cannot be avoided, but can be controlled. Specific assignment of a speed requirement to a section of a design is called a critical path. Second, there is the internal arrangement of relative pieces of logic, or "blocks". This can also be managed, either by the Place and Route tool, or by manual arrangement with a *floorplanning* tool. Third, there are the I/O pins on the device that dictate the physical entry and exit of signals to the design. This aspect is handled by pin assignment software.

There are more constraints than mentioned here, but these are the main ones that must be considered in most designs. Typically, designers must identify their speed requirements then formulate them into an appropriate set of constraints, before doing effective Place and Route. However, for absolute best results, it is generally accepted that the physical design tools perform best with minimum constraints. Minimize or eliminate critical paths, where possible. The freedom to move cell locations and redistribute time delay due to metal, is vital to achieve the best overall performance in most designs. Minimizing critical paths should be kept in mind when selecting constraints for the design.

## Design Verification

There are many aspects to design verification. First, there is simple design rule checking, which is usually done throughout the whole design process. At various steps, the software assures that outputs are not wired together, inputs are not floating, or signals that should be connected are not. An important discipline to adhere to is taking the design completely through the process, without introducing shortcut edits into the design process. For instance, once a designer becomes aware of the netlist structure, there is sometimes the temptation to insert manual edits into netlists and move forward. This can be disastrous, because it may bypass circuit checks that are present in the standard design flow.

A common design verification method is simulation. Simulation creates a model of the circuit inside of a computer. The circuit model is a set of computer data structures describing the behavior of gates, flip flops, I/O connections and other logic items. All binary simulators have the ability to model logic ones and zeroes but, more robust simulators support additional state or node conditions, like high impedance, open drain with resistive pull up, pull down impedance, nodes in transition, and so forth. Often, the ability to model many different conditions in a circuit can give greater insight to the circuit behavior, so those modeling qualities are highly regarded.

The point of simulation is to reflect as close as possible the actual state of the silicon device. Very simply, this is because we are unable to internally observe the actual device in operation, so the simulation model becomes our window into the heart of the device. An interesting description of FPGA design software is the following: think of the software as the “lens” through which we view the design embodied in the device architecture. Very accurate simulation modeling is an important aspect of that software “lens”.

A primary reason that simulation is vital is that when the design is placed and routed onto the device, there will be additional delays in the design, due to the arrangement and connections. These delays are not obvious from the design itself. We need a way to determine if the function mapping, cell arrangement and connections added time delays in such a way that the design operation is useful, or not. If it is not useful, we must determine that quickly, so we can take actions to improve it. The process is iterative. Usually, it involves making small circuit edits, or changes in design constraints. Once the changes are made, we must re-run the placement and routing software, and subsequently assess the new situation.

The process of getting the actual time delays out of the circuit and folding them into the simulation model is called back annotation. Back annotation consists of calculating the time delays in the connections, then placing those delays into special fields of the

simulation netlist, so the simulator can represent the time delays when it runs the simulation. Without this capability, we would be blind when it comes to adjusting our designs.

### Summary and Conclusions

We envision a world where you dream up a design in its most abstract form, push a button and the software delivers the perfectly programmed device embodying your every expectation. Our vision is still a little fuzzy, but we are converging on that reality, faster each year. In this chapter we described some fundamental ideas of FPGA architectures and their supporting design software. Along the way, from the roots of logic design, we developed a logic cell and an array fabric, which could do the basic required functions.

We also showed that it had lots of room for improvement! Things that could be improved include:

1. I/O signals. Simple inputs and outputs created a problem with the Weird design, whereby we couldn't compile it due to insufficient pins of the right type. Having all pins be I/O pins would have resolved that.
2. More routing resources. The vertical and horizontal routing lines were simply, too few.
3. Appropriate global resources. The array had no global set or reset signals (real flip flops use those functions).
4. The Weird design needed two clocks, when we had but one.

It's easy to dream up architectures, then show their deficiencies, and improve them over time. In fact, the last twenty plus years at Xilinx have been just that. The Virtex Families represent the culmination of thirty years of team based research, while partnering with Xilinx customers to create better FPGA devices.

### Looking Ahead

Chapter Two traces the evolution of Virtex architectures. Chapter Three describes the Virtex families from the beginning, through Virtex 5. Chapter Four presents key ideas of Physical Design, and the Virtex design flow. With that understanding behind us, we then tackle additional topics that detail memory architectures, arithmetic structures, I/O circuits, high speed data communication circuits and more. The first chapters show how everything works. Later chapters show what you can do with this powerful technology.

## Chapter 2 Getting To Virtex

The basic ideas presented in Chapter 1 outline the simplest function and interconnection requirements that propelled Xilinx for the first three generations of FPGA devices. As each family arrived, a set of problems would be solved in new and interesting ways. This chapter traces the evolution of the XC2000, XC3000 and XC4000 families. We will be looking quickly at the logic building blocks, interconnect and the I/O capabilities as kind of a “barometer” of how customer needs would drive these solutions. The very first family – the XC2000, was basically defined by Ross Freeman and designed by Bill Carter, early Xilinx pioneers.

### XC2000

The XC2000 consisted of two parts – the XC2018 and the XC2064. The acronym LCA stands for logic cell array, representing the idea of a regular array of logic cells. The logic world had a similar concept developed much earlier, called cellular logic, where logic is created from identical cells, however that term is now viewed as a mathematical or academic term. Xilinx founders struggled with how to refer to their technology. Before calling them field programmable gate arrays (FPGA devices), the LCA expression remained with Xilinx until the Virtex family arrived. Nonetheless, the XC2018 and XC2064 were created from a common fabric, and Figure 2-1 shows the structure of the first FPGA configurable logic block (CLB).

Note that the CLB has four inputs (A, B, C and D) entering a block labeled “Comb. Logic”, which is a lookup table structure similar to that shown in Chapter 1. The rest of the figure is comprised of the flip flop and multiple trapezoidal multiplexers. This was to be a standard Xilinx notation, showing multiplexers with no designated select inputs, as trapezoids. The multiplexer select input is driven from the hidden “configuration space”. Dynamic multiplexers are shown as rectangles, with inputs on the left, outputs on the right and select lines on one end or the other.

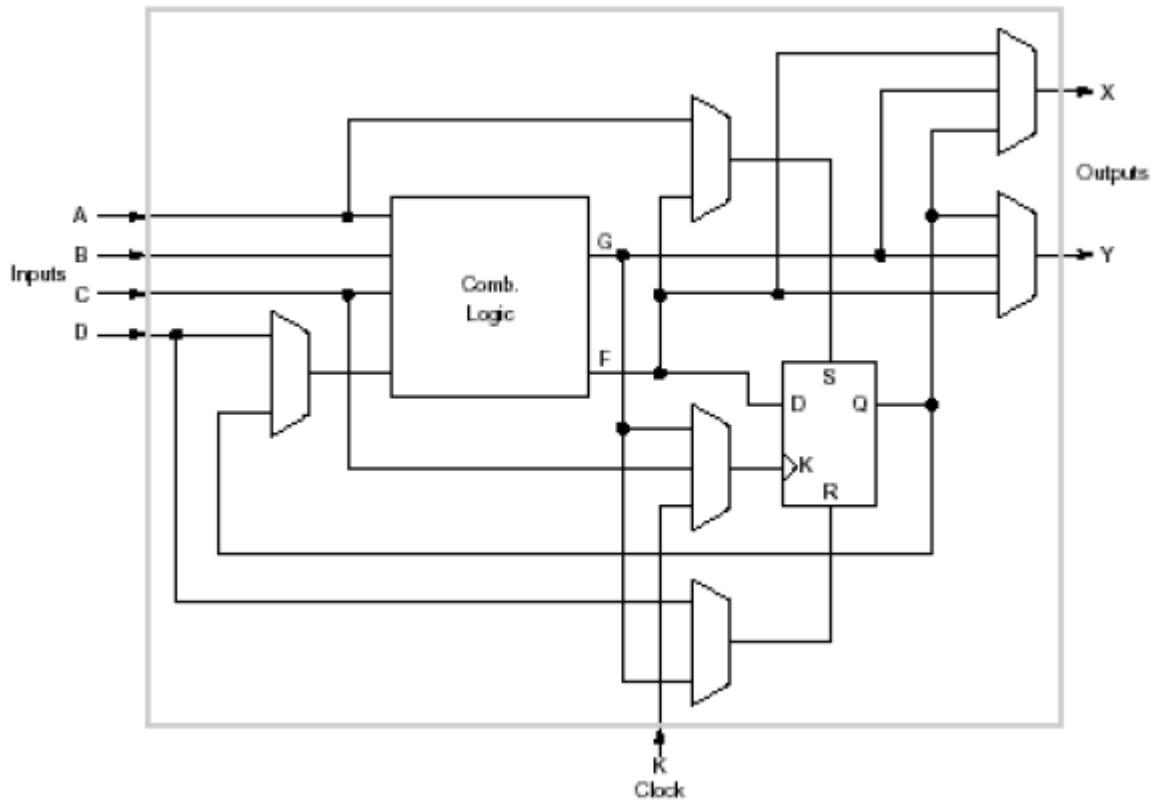


Figure 2-1 XC2000 CLB

As noted, there are two outputs from the CLB – X and Y, and there are two outputs from the LUT – G and F. Those names remain with Xilinx architecture right through early Virtex. It should be obvious at this point, that the “Comb. Logic” LUT can build any function of 4 input variables, but care needs to be taken, as the external inputs A – D, can also be slightly altered. Particularly, D can multiplex with the Q output of the flip flop, so that local feedback occurs within the CLB. This makes for faster switching, than to send the flop Q outside, to be routed back into the CLB doing the same thing. Figure 2-1 shows some additional variations that can be connected for the XC2000 LUT. Some other things to note in Figure 1 are:

1. The A input can be multiplexed through to the flip flop Set.
2. The D input can be multiplexed through to the flip flop Reset.
3. The LUT output G passes through to the X or Y output, or the flip flop clock input.
4. The LUT output F passes through to the X or Y output, or the D input of the flip flop.
5. The flip flop clock comes from the “K” input or the C input, as well as the LUT G output, your choice.

The ability to cross over from LUT outputs (F and G) to CLB outputs (X and Y) through multiplexers, adds to the overall CLB flexibility, and may have some level of symbolic reference to the famous Xilinx “X”.

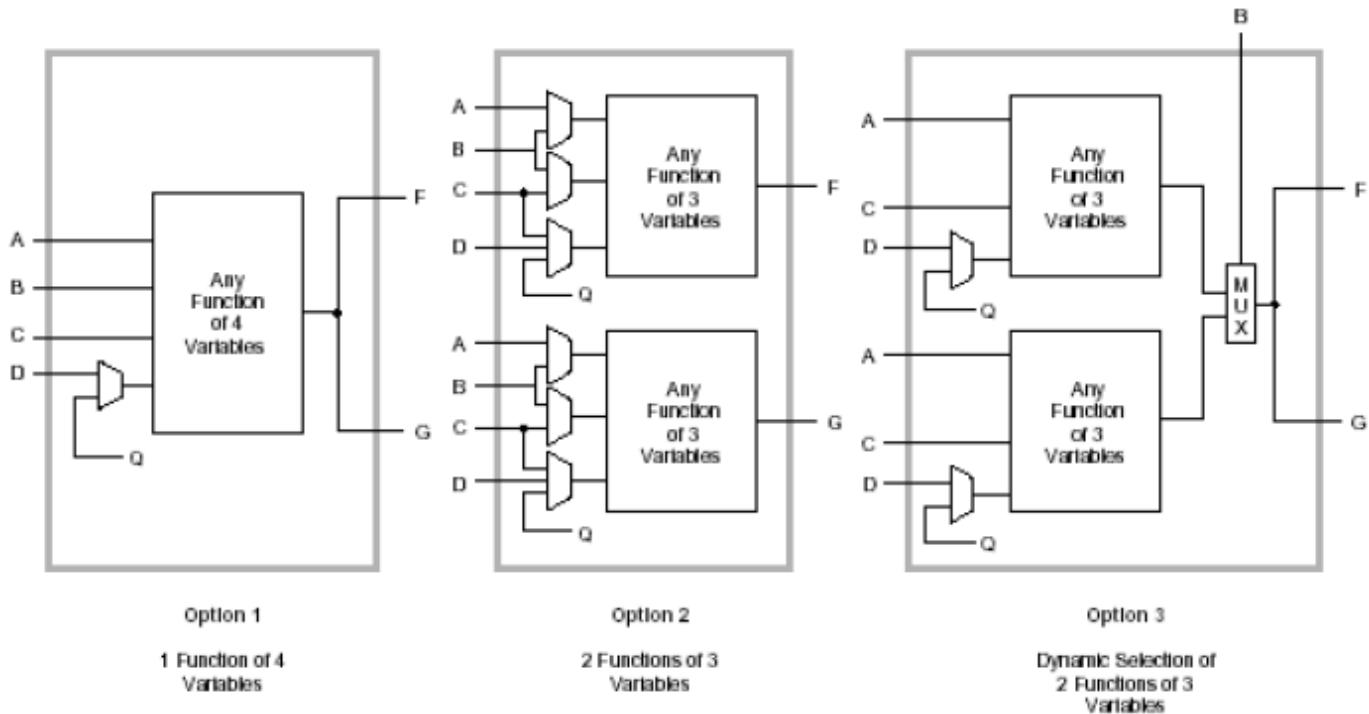


Figure 2-2 Other Optional Function Creation for the XC2000 CLB

Figure 2-2 shows some other connection options for the LUT, but Option 1 is the main default the software tools typically chose. Option 2 (2 functions of 3 variables) and Option 3 (dynamic selection of two functions of 3 variables) were possibly chosen by users that worked at the lowest level, operating with the manual software program known as the FPGA Editor. Although multiple options existed for FPGA design, synthesis was not prevalent at this time, so designers relied on choices offered in a schematic library or manual editing with FPGA Editor. The ABEL design language was occasionally used, also.

The logic cell was a pretty tidy solution. Interconnect was not.

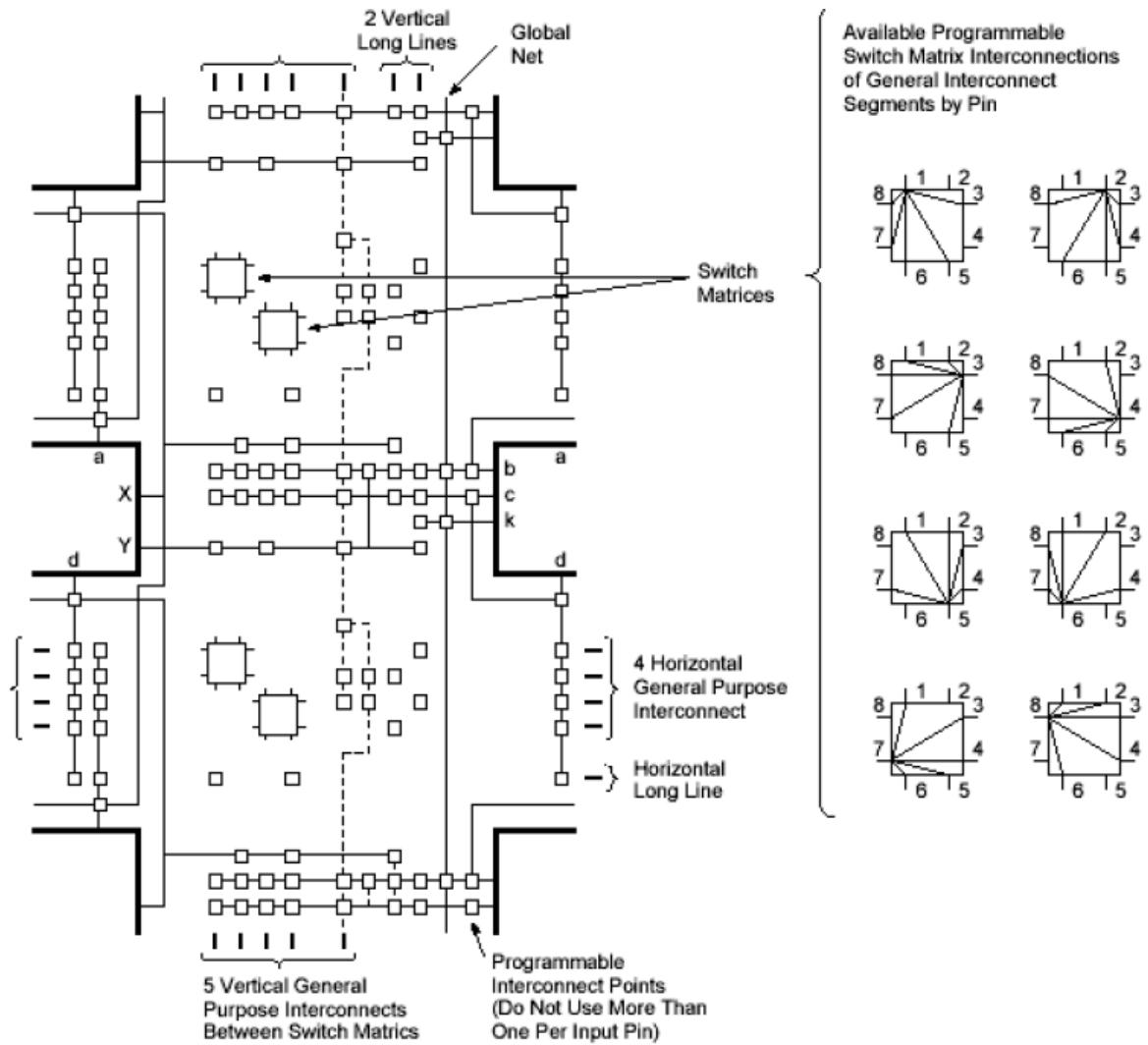


Figure 2-3 XC2000 General Purpose Interconnect

Figure 2-3 shows some of the possibilities for making connections. The heavy bordered box in the middle left and middle right represent the CLBs. The smaller squares with two ticks on each edge are switch matrices, and the tiny little squares sprinkled all around are routing PIPs (Programmable Interconnect Points). Xilinx internal jargon frequently refers to the number and/or arrangement of PIPs as pipulation. Figure 2-3 on the left portion shows lots of optional connect sites for the right side CLB inputs (B, C, K). As well, there are connection options for the CLB outputs (X, Y), and the wording at the bottom suggests that there are vertical metal segments that pass from the south (bottom) towards the north (top). As well, horizontal connection sites pass from east to west.

The right hand side of Figure 2-3 shows a set of eight switch box connection possibilities. They look a bit like little spider webs. The top left one shows connections going from 1 to 3, 5, 6, 7 and 8. The top right one shows connections from 2 to 3, 4, 5, 6 and 8, etc. The idea is that each diagram shows a unique entry site, then several exit sites that are possible. For the top left one, a signal enters on tick#1, and can exit on 3, 5, 6, 7 or 8. This is accomplished by inserting a “PIP” between tick#1 and the other ticks. The PIP is a pass transistor enabled by a storage latch.

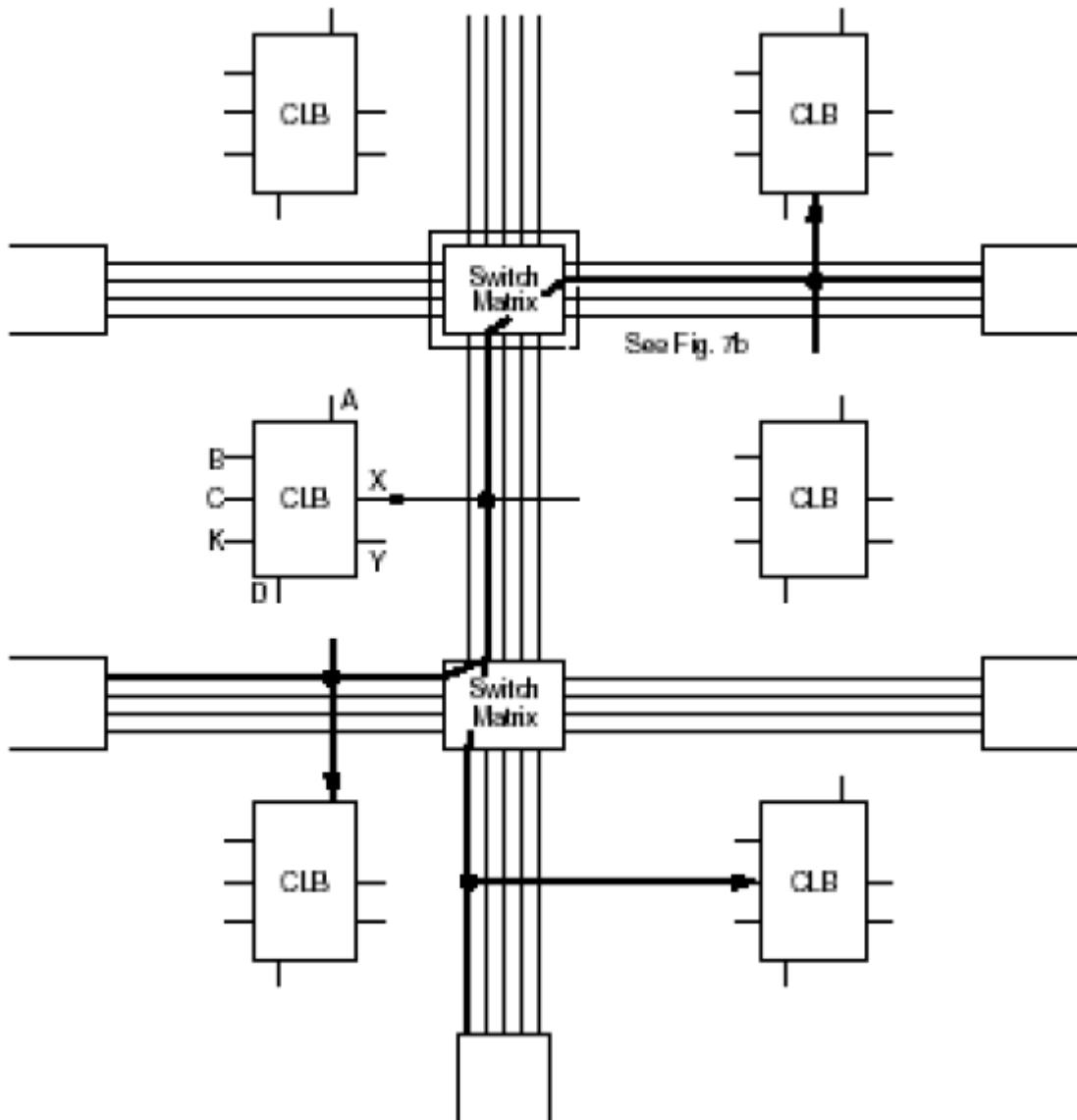


Figure 2-4 Making XC2000 Connections on the General Purpose Interconnect

Figure 2-4 shows a CLB in the center, attaching its X output to a vertical interconnect that goes both north and south through two switch matrices and connecting elsewhere.

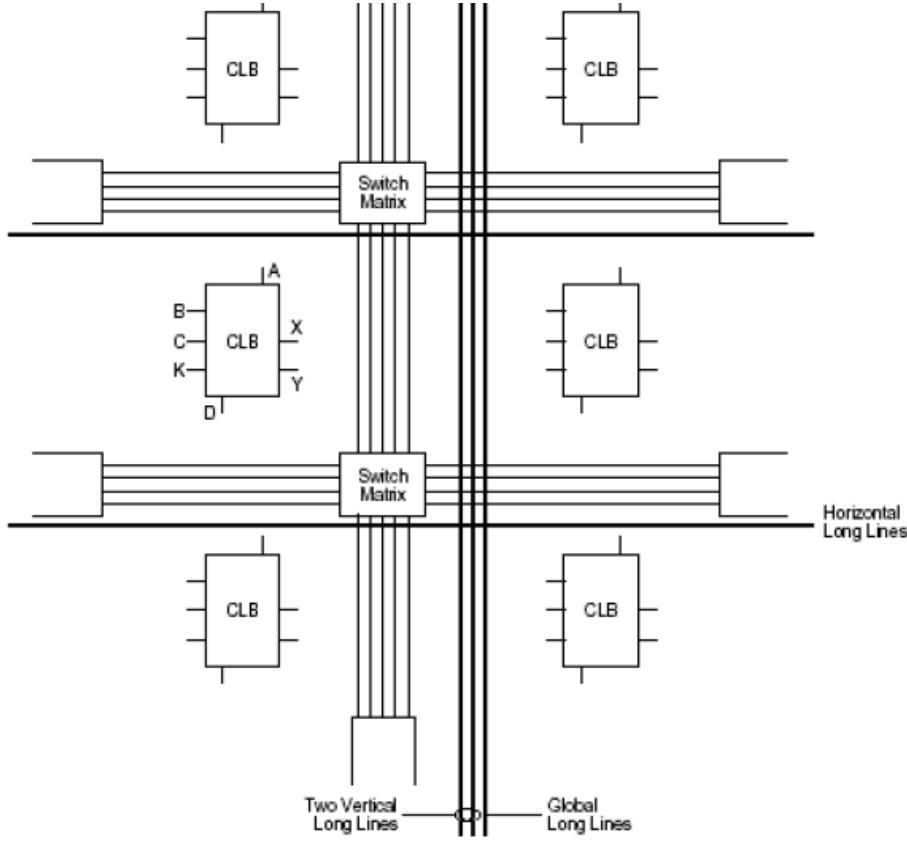


Figure 2-5 XC2000 Long Line Interconnect

Figure 2-5 shows a set of “long line” interconnect wires passing completely across the chip in the vertical and the horizontal directions. These are dear resources, as they are few in number. It was early observed that appropriate placement of functions into neighboring CLBs had a distinct performance payoff, so making efficient neighboring connections was critical. Occasional global connections were allowed, but occurred infrequently compared to neighboring ones. This assumes functions are well placed in the FPGA cell array.

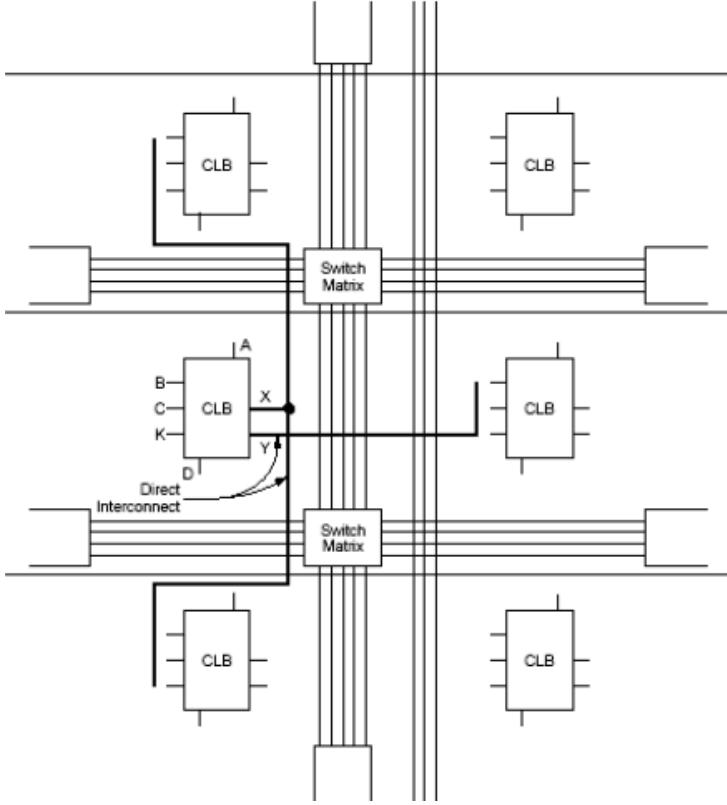


Figure 2-6 Neighbor Connections

Figure 2-6 shows how the CLB X output may make certain connection to its nearest north and south neighbor, whereas the Y output may connect directly to the east neighbor. These connection sites are dedicated, not competing with other signals for access to the General Purpose Interconnect. Design software strongly attempted to place logic using neighboring connections, and reserved G.P. Interconnect and Long Lines for harder problems.

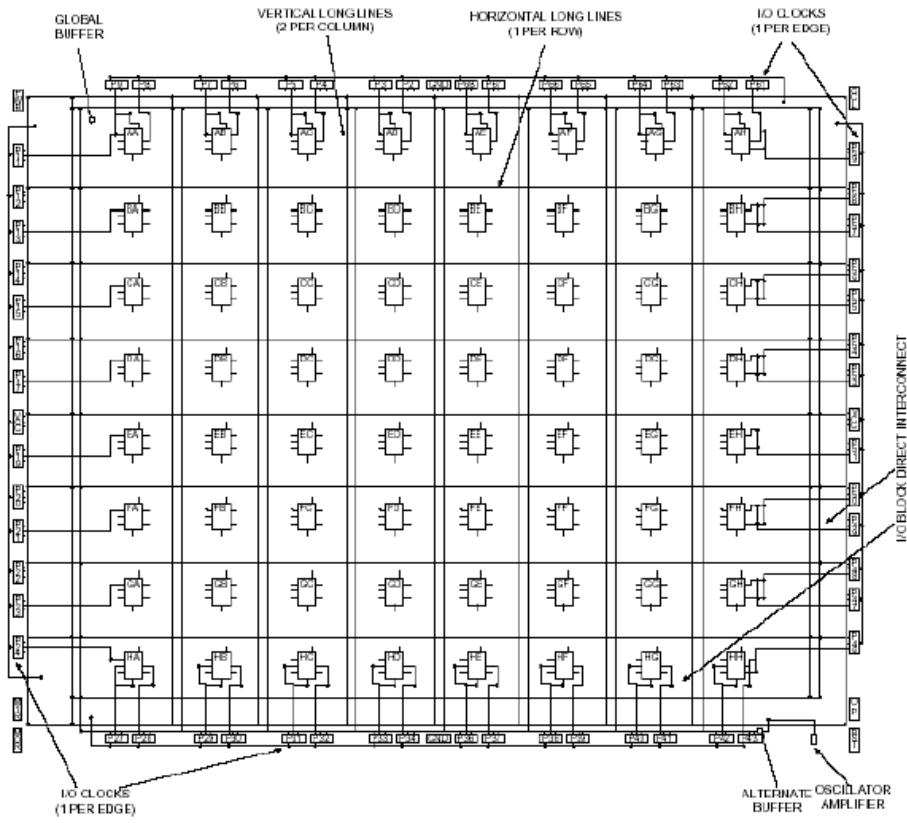


Figure 2-7 XC2000 FPGA “World View”

Figure 2-7 shows how the FPGA Editor displayed the rows and columns of CLBs with respect to the vertical/horizontal interconnect and the I/O pins. Connections are shown as lines, and the above diagram represents a partially routed design. Also shown are global buffers and clock sites. Figure 2-7 represents what the computer aided design (CAD) tool ‘FPGA Editor’ termed a World View.

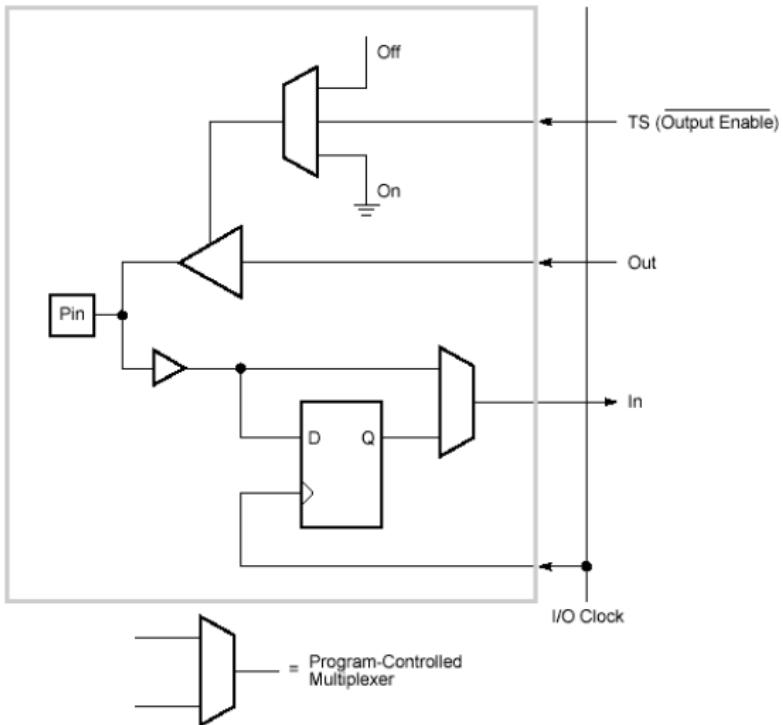


Figure 2-8 XC2000 I/O Cell

The I/O cell of Figure 2-8 shows a very important trend that Xilinx helped to introduce – all pins are I/O pins. Much frustration was experienced using early gate arrays, which offered density mixes of gates in their foundations, with fixed numbers of inputs, outputs and I/O pins. Frequently, getting the right I/O mix required going to a density that cost more, and was underutilized. Those ASIC designers found a welcome resource with FPGA devices, where every pin was an I/O pin. You simply needed to know how many pins were required for the design to know you could get the right mix. As seen in Figure 2-8 tri-stated output existed at every pin, as well as a registered input pin, or bypass entry to the FPGA. The only real downside in the early days was that the output pins were limited to a 4 mA output drive. Having potentially all I/O's came with a price – die size, and that was dealt with by down scaling the current drive.

The XC2000 taught Xilinx and its customers many important lessons. Users needed to rethink their approach to fairly standard logic problems, like how best to design state machines – even counters. Luckily, there exist a number of experienced designers that could recall and modify methods that worked for earlier TTL technologies, where the restriction of having identical parts in 14 and 16 pin DIP packages was shown to be very similar to connecting up CLBs. Hence, the cascade-able counter methods, and “one hot encoded” state machines were called back into duty for building designs within the XC2000 family, much to Xilinx’ benefit. A generally workable approach for many problems was to simply “think of your design” in terms of four input building blocks. It worked. Nonetheless, as the Xilinx audience grew and knowledge of underlying problems evolved, a newer architecture arrived – the XC3000.

## XC3000

Figure 2-9 shows the XC3000 CLB, which offered many changes to the much simpler XC2000 CLB. For starters, the new CLB combined a five input LUT with two outputs, F and G, and two flip flops, with block outputs still being labeled X and Y. Additional static multiplexing assured sufficient connections among the LUT outputs to connect F to X, and to both flip flop D inputs - and G to Y, and also both flip flop D inputs. The LUT inputs are expanded to handle five signals (A – E). An additional “Data In” input connects to multiplexers permitting a direct optional load of a selected flip flop from the routing area. Both flip flops are clocked from the same source, which can be programmed to invert or not. Both flip flops are reset from the same network, which can combine a global reset signal with a logic signal (RD) to permit different resets for different CLBs. New thinking about feeding back flip flops within the CLB to their local LUT added speed to state machine designs (typically counters), and removed those signals from having to compete in the general purpose interconnect area, to simply switch their own input. Clustering local functions within the small CLB region, making it more self-sufficient, and also improved its speed.

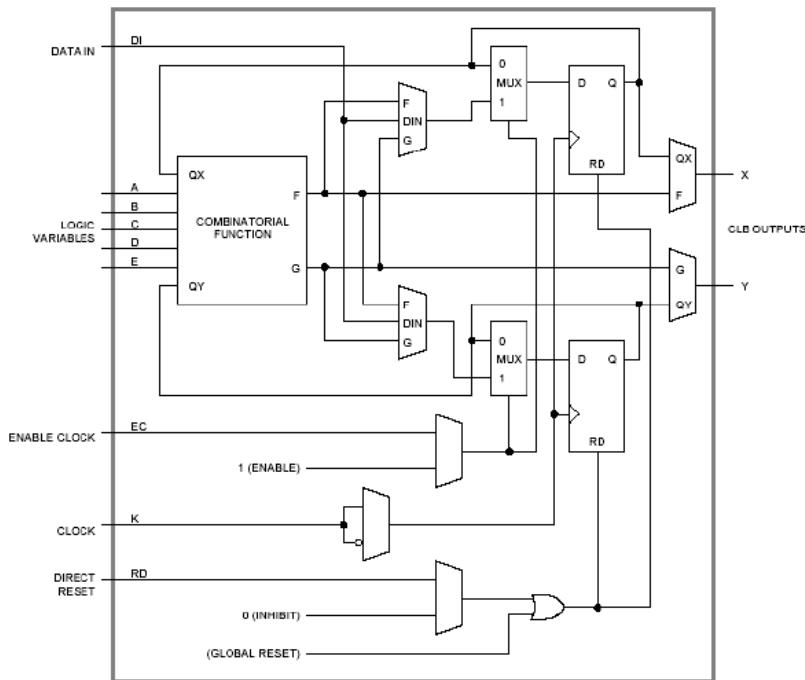


Figure 2- 9 XC3000 CLB

Figure 2-10 shows three different ways to view the XC3000 LUT. First, it can be constructed to deliver two independent functions of four input variables, with state feedback multiplexing adding the QX and QY into the mix of inputs. Second, it can be a single five input LUT – again, with the state outputs adding into the A-E input signal mix, and third, it can be two different 4 input LUTs (same A-D) dynamically switching under the control of the E variable, and driving both the F and G outputs. As you might expect, for software reasons, the default was the five input LUT.

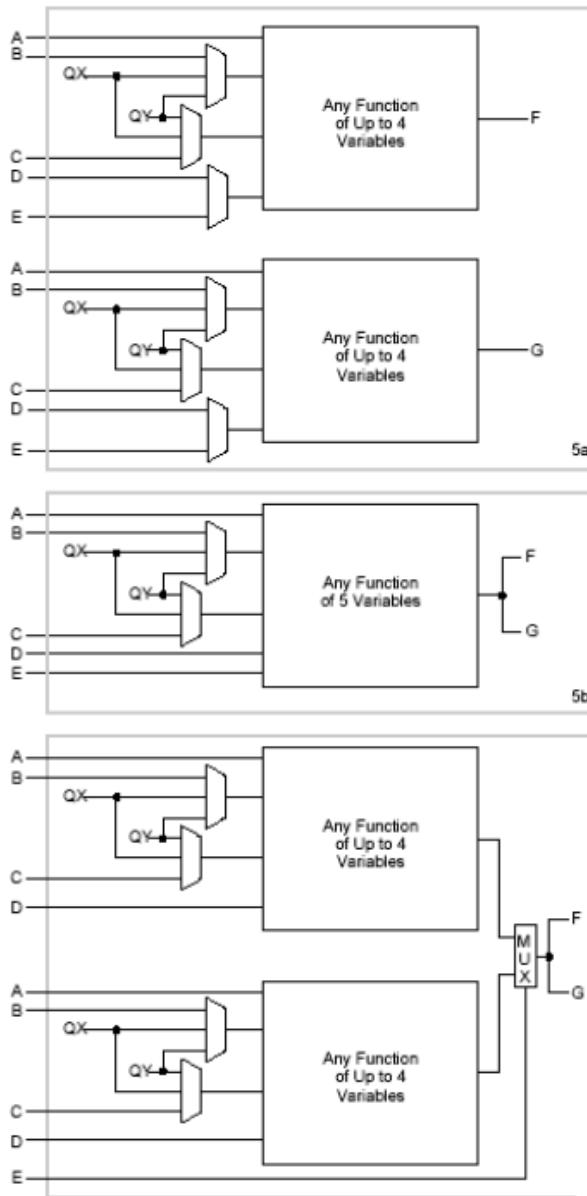


Figure 2-10 Three Ways to Use the XC3000 LUT

Figure 2-11 shows the array of CLBs and switch matrices on a grid of interconnect sites of general purpose interconnect. This structure is similar to that of the XC2000, only this time the PIPs are shown as little dots instead of the tiny boxes earlier.

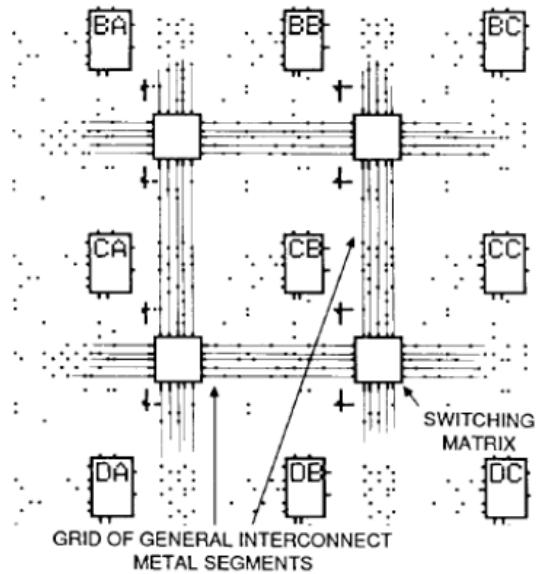


Figure 2-11 XC3000 General Purpose Interconnect

Not mentioned earlier, but more evident in Figure 2-11 is the labeling of individual CLBs, with alphabetic “Cartesian coordinates”, where the origin would be AA, in the upper left hand section (not shown) on this diagram.

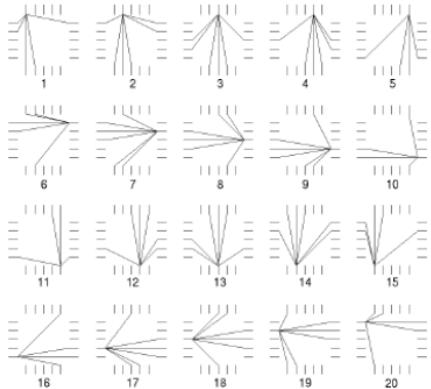


Figure 2-12 XC3000 Switch Matrix Connection Possibilities

The new switch matrix resembles the original, in that pass transistors allow connection between various sites, under control of latched constants held in the configuration space. One thing not mentioned, is the possibility of connecting an incoming signal to two different exit sites, which is possible, but careful attention of loading (speed issue) must be observed. It is also possible to connect from one incoming metal segment to an adjacent segment, on the same side of the switch matrix, by making a “bank shot” across the switch matrix and back to the same starting side, but on another metal segment. (The reference here is to a billiard cue ball being driven into the rail, then angling elsewhere onto the table, to avoid a roadblock). Figure 2-12 shows twenty switch box option patterns.

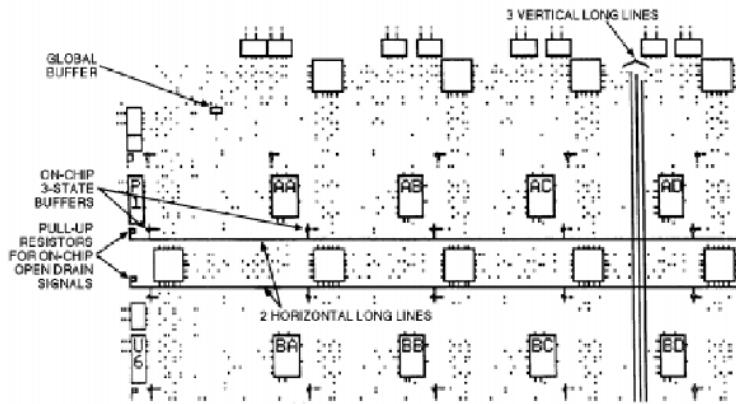


Figure 2-13 XC3000 Long Lines

Figure 2-13 shows the long line sites for XC3000 FPGA devices. Additional detail shows where pullup resistors reside, and where tri-state buffers are located to access the long lines. A key aspect of these interconnections is that tri-state bussing could occur on them, much like the bussing in a microprocessor system. This type of resource proves invaluable later, when designers began seriously placing “soft processors” within FPGA devices.

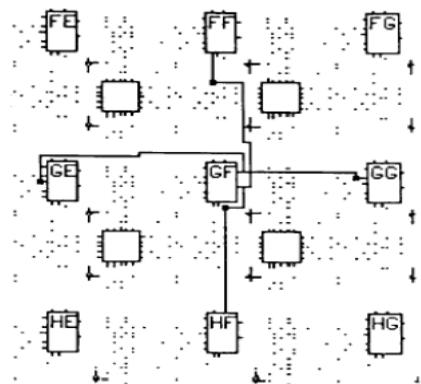


Figure 2-14 XC3000 Neighbor Connections

The importance of local attachment to neighboring logic functions is critical for high speed designs. Time delay from relatively simple connections to the neighboring cells, undermines the performance of a design, if not properly addressed. With the north – south access of the Y outputs and the east-west access of the X outputs, neighboring cells connect without accessing the GP inter connect sites. Cell placement of functions so that this could occur is very important, and many designers didn’t understand these fairly fundamental ideas. Most college textbooks, had largely avoided the topics of time delay, and FPGA devices were too new to be used as the most common example. Clearly, those days are gone, and today’s graduating designers are much more familiar with the placement of logic within their target fabric, than before. Figure 2-15 shows the XC3020, when viewed through the FPGA Editor software. The “speckly” PIPs are regularly shown throughout, the rectangles are the CLBs and the square boxes are the switch matrices. Notice the regularly spaced solid lines passing from the north to the south of Figure 2-15. Those are the vertical long lines. If you look between any two vertical interconnect lines, you will see a regular structure that repeats. For visualization, it should include one CLB,

one switch matrix and all the little PIPs directly around it, until that pattern repeats, again. The region is bounded above and below by a “blank section”, and east and west by the shown long lines. That is what Xilinx design engineers call a CLE Tile. The Configurable Logic Element tile illustrates the method used by the Xilinx team designing FPGA devices. They literally spend about a year carefully designing the tiny, repeatable section that gets stepped and repeated within the connection array, over and over, to build up the FPGA. The same general approach that was used for the XC3000 is fundamentally the approach being used by the high end Virtex parts, today. A key value of this approach is that once the optimized tile is well known and designed, then creating a family of parts is straightforward.

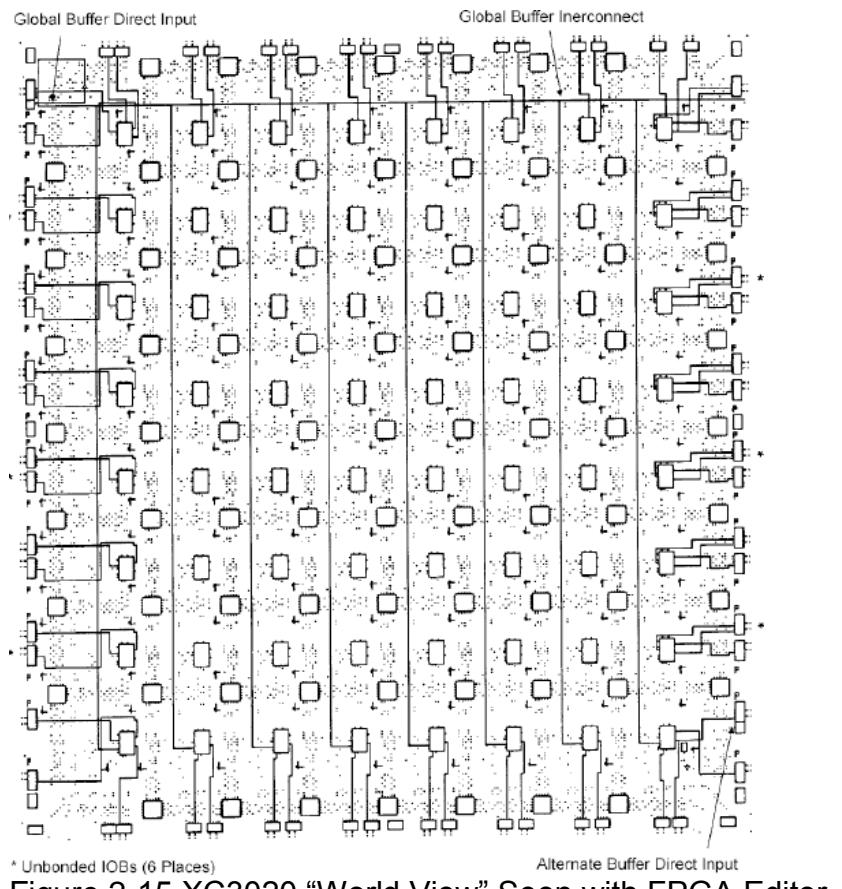


Figure 2-15 XC3020 “World View” Seen with FPGA Editor

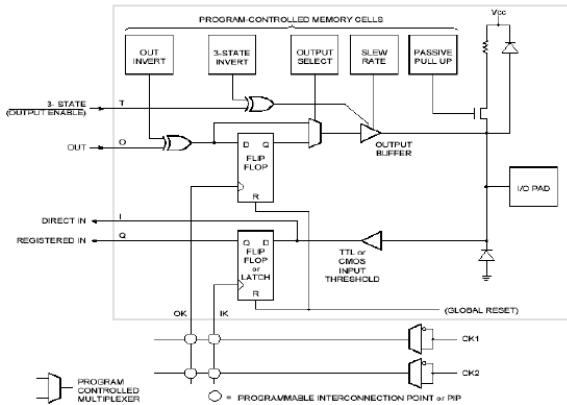


Figure 2-16 XC3000 I/O Cell

Figure 2-16 shows the I/O cell for an XC3000. Each input and output has its “own” optional flip flop/latch, clocked by one of two sources, each on one of two phases. Otherwise, the flip flops can be bypassed. Signals arrive from, and are connect to, the routing fabric on the left hand, and subsequently to the I/O pad on the right. Special memory cells, shown at the top, permit any of the five shown options to occur on an I/O signal. Flexible I/Os became a hallmark of Xilinx FPGA devices, and customers were quick to offer additional advice on future options, that would evolve.

The XC3000 taught Xilinx much. Both the XC2000 and the XC3000 have asymmetrical switch matrices, making life difficult for the Xilinx software team, and ultimately Xilinx customers. This needed to be remedied. When the XC3000 came out, Xilinx undertook a second sourcing arrangement with A.T. & T., which produced some difficulties in the business world. To alleviate that, Xilinx brought out an alternate family that solved two problems – one business, and one technical. The family was named the XC3100 family, and it was very similar to the XC3000, but it included a speedup trick that only Xilinx provided – they charge pumped the PIPs (gates of the pass transistors operated at a slightly higher voltage than the core supply). As signals propagate through the FPGA, the pass transistor that is first encountered, causes a voltage drop (one threshold, or  $V_t$ ), and exposes it to greater jeopardy from noise. To eliminate this, Xilinx designers added a voltage pump that internally raised the VDD for the latches driving the PIP gates, so they would not drop voltage across the pass transistor. This approach was remarkably effective and paid off in several megahertz of additional performance. It was a big success. In fact, the XC3100 was the first FPGA to be PCI compliant and was used by the PCI organization in some of its own demonstration and testing boards.

Other variations of the XC3000 family existed, one that was low power, and one that was nonvolatile. The low power family was somewhat successful, but this market was very cost sensitive, and LCA technology had yet to achieve the price points available today. The nonvolatile FPGA – the XC3720 (logically identical to Figure 2-15) combined EPROM on chip, and self-configured during power up. A desirable feature, for sure, but it ran half the speed of regular XC 3000 parts, at about twice the price. Not a good combination, so it was only briefly marketed.

By the time the XC3000 was in full production, Xilinx application and software engineers

had developed better ways to design. VHDL and Verilog were in their infancy, so schematics and ABEL were the methods available at the time. Offering schematic libraries of familiar functions (like TTL chips) gave designers the building blocks they needed, and produced good results, when connected up within the FPGA devices. Xilinx engineers recognized that there was value to be had in giving customers easy to use, flexible tools, and developed one of the early approaches to procedural design – XBLOCKs. This approach let customers select among a library of functions. The user simply dialed in the correct data widths for the various operands, and the result was a netlist that the Xilinx Design Software could easily connect up. From this point on, Xilinx would be working very closely with third party software developers to create effective solutions for the end designer – both of their customers.

## XC4000

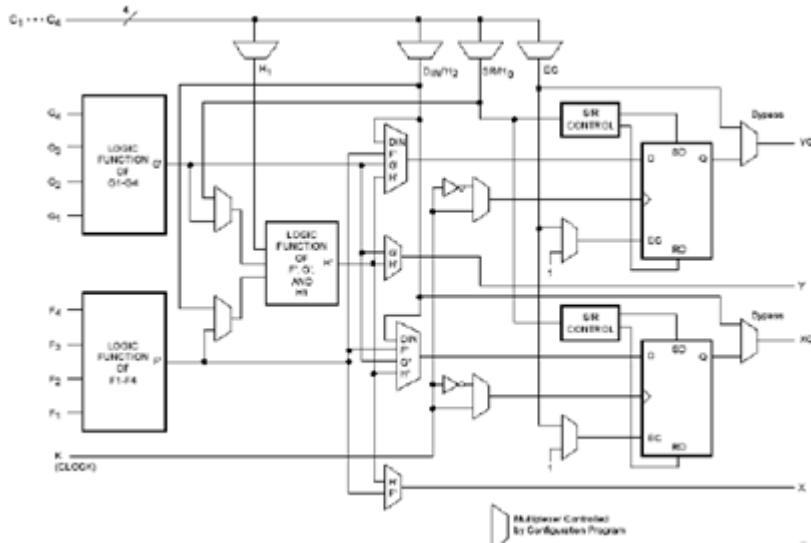


Figure 2-17 One View of the XC4000 CLB

Serious competition arose from nearly a dozen different providers that sought access to the FPGA marketplace, which was developing its own customer base. Some designers came from logic design, where they wanted to integrate more TTL into fewer packages. Some designers came from gate array and standard cell approaches, where they suffered from poor design tools and unforgiving “non-recurring engineering” (NRE) expense. Others came from simply wanting to create new designs in creative ways that had never existed before – like using logic that could be modified, after it had been sold to end customers. New models of the way design should occur were arriving. Along with those new models, other very big changes in voltage levels on printed circuit boards, created demand for multiple XC4000 families, to deal with 5, 3.3 and 2.5 volt signaling standards. The reason that Figure 2-17 is called “one view” of the XC4000 CLB, is that complexity had grown by this time, so it was very difficult to show all the possible functionality available, without confusing new users. Xilinx marketing began showing different views of the CLB logic capabilities, where each view focused on different aspects of the block. Suffice it to say that Figure 2-19 shows another view of Figure 2-17, while focusing on the XC4000 arithmetic capability, versus its logic capability that is highlighted in Figure 2-17. Getting back to Figure 2-17, we see two 4-input LUTs on the left, with Gi and Fi inputs,

and corresponding G and F outputs. In the middle is a third “H” LUT, and on the right are two flip flops. Note that there are now four CLB outputs from this cell, so both flip flops can exit as well as internal combinational signals. The view of the CLB was changing. It is not only a logic resource, but is also a routing resource. Check the top multiplexer - labeled DIN/H2 - and note a direct connection to the XQ CLB output multiplexer labeled “Bypass”. In a sense, the XC4000 CLBs has qualities of the XC 2000 CLB (one 4 input LUT per flip flop) and the XC3000 CLB (combines two flip flops within a single CLB). Naturally, there is a lot more. The added H LUT permits the cascade of the G and F LUTs to make a set of important logic functions that can be as wide as nine inputs. For instance, let’s assume that the truth table for a four input OR is in the F LUT, and another four input OR is in the G LUT. By loading a three input OR gate into the H LUT, the combination delivers a nine input OR to the H LUT output, with OR inputs on  $F_i$ ,  $G_i$  and  $H_1$ , coming from the top section. By altering the H LUT to have a three input NOR input, the result becomes a nine input NOR at the output of H. Loading F and G with the four input AND truth table and H with a three input AND, its output delivers a nine input AND. Changing H to a three input NAND makes the set create a nine input NAND. One more example, would be to load four input EX-OR functions into F and G, and a three input EX-OR into H, creating a nine input EX-OR function within a single CLB. Specifically, this can calculate parity over a byte, which is something very useful for fast data standards like PCI.

The XC 4000CLB was more “opened up” than its predecessors. More entry signals were provided, to more internal sites, via the various multiplexers. This permitted more opportunities for internal paths to make connections and operate faster. Figure 2-18 shows the dual port distributed RAM structure that the F and G LUTs can create. The same basic structure is discussed in Chapter Six on Virtex Memory, but this structure stoked the desire of users needing SRAM within an FPGA. With this structure, it became possible to more efficiently store DSP coefficients than before. It also became possible to make FIFOs, so interfaces across multiple clock domains would be easier. The list goes on and on, about the advantages brought to programmable logic, with RAM.

We will review that list in Chapter Six on Virtex Memory.

In Figure 2-18, we see that the addresses of the two ports are supplied by the F and G LUT inputs. The Datain is a single bit, coming from the D0 multiplexer, which forks into both LUTs. Duplicate contents are kept in each, so that it is possible to read from one, while writing to the other. The F and G LUT inputs become the addresses into the combined memory, and are independent of each other. Examining Figure 2-18 carefully shows that the F address inputs dictate where data is written, for both LUTs. Both F and G address inputs connect to the respective LUT output multiplexers, so it is possible to read simultaneously from different locations, but writing will always be to the address selected by the F inputs. Chapter Six covers more detail on how internal electronics is connected for these structures. At the gate level, it is a very similar capability that moved forward into the Virtex architectures.

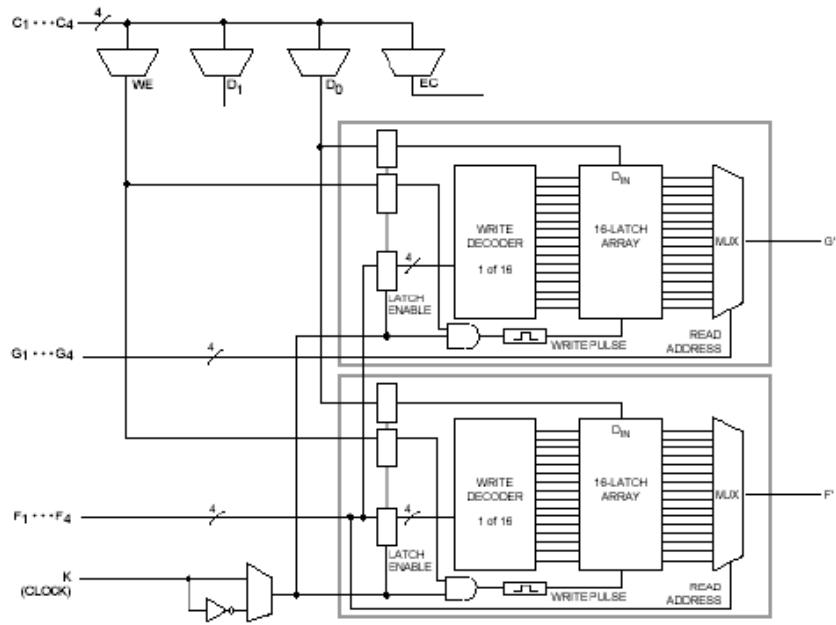


Figure 2-18 More Detailed Distributed CLB Dual Port RAM Structure

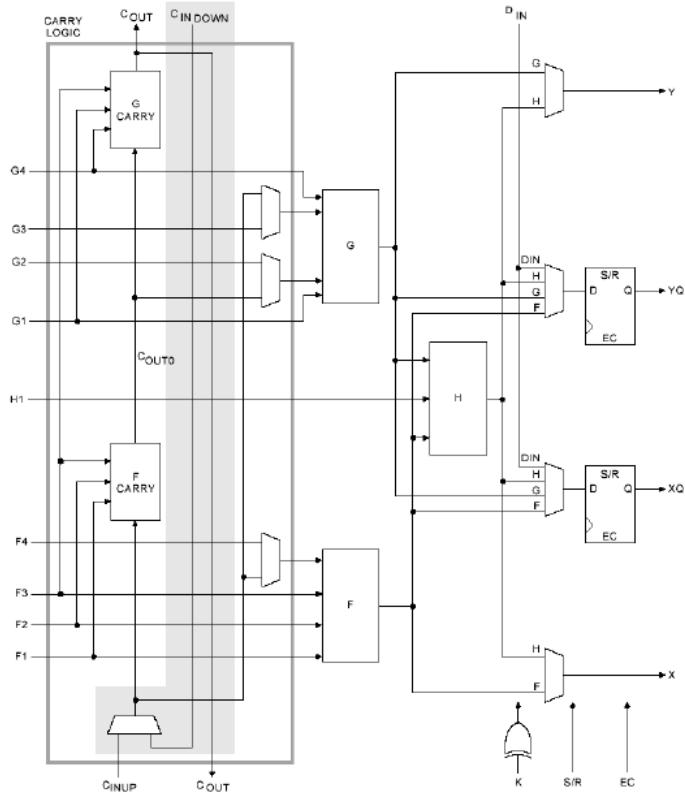


Figure 2-19 XC4000 CLB with Carry Circuitry Exposed

Figure 2-19 shows detail that was not exposed in Figures 2-17 or 2-18, but are in fact, present. If operands are appropriately attached to the F and G inputs, then the LUTs can build up full adders, and additional external circuits can quickly anticipate carry bits, so

that adders in a chain respond more quickly. This is called the fast carry circuitry, and greater detail is shown in Figure 2-20.

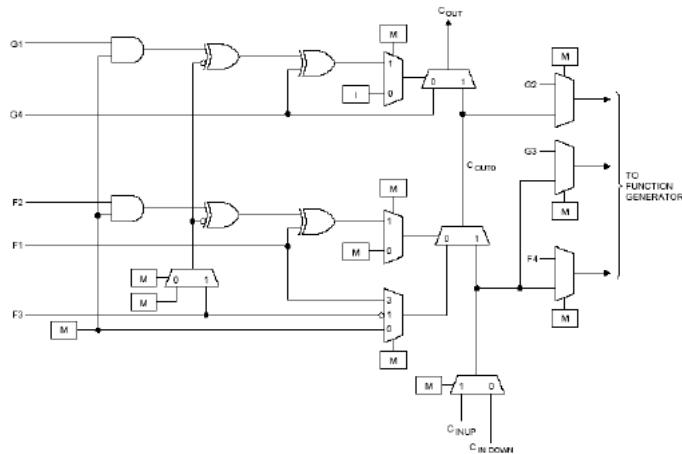


Figure 2-20 XC4000 CLB Fast Carry Logic

Greater detail on arithmetic is contained in Chapter Seven, but it is important to note that placement of adder functions is very important, to achieve the maximum potential speed. Slight variations have been developed for different FPGA families at Xilinx, but Figure 2-21 shows one arrangement, for the XC 4000X families, where adjacent CLBs within columns pass carries quickly up the columns.

Getting to the end of a column without completing the add function was also anticipated. If needed, forwarding carries to adjacent columns occurs in the horizontal direction (dotted lines are general purpose interconnect) for the least significant bit of the adder, then carry forwarding resumes up the columns. As time passed, the Xilinx design software developed so that placement information was automatically inserted into the design. Best results are obtained when the user chooses standard adder primitives with their favorite design method.

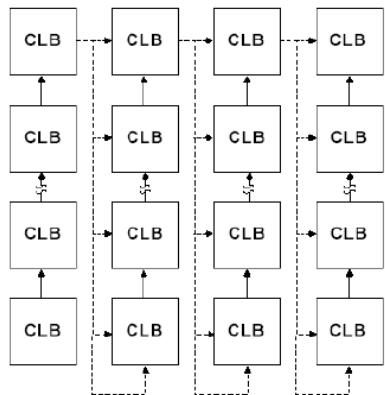


Figure 2-21 XC4000X Carry Propagation

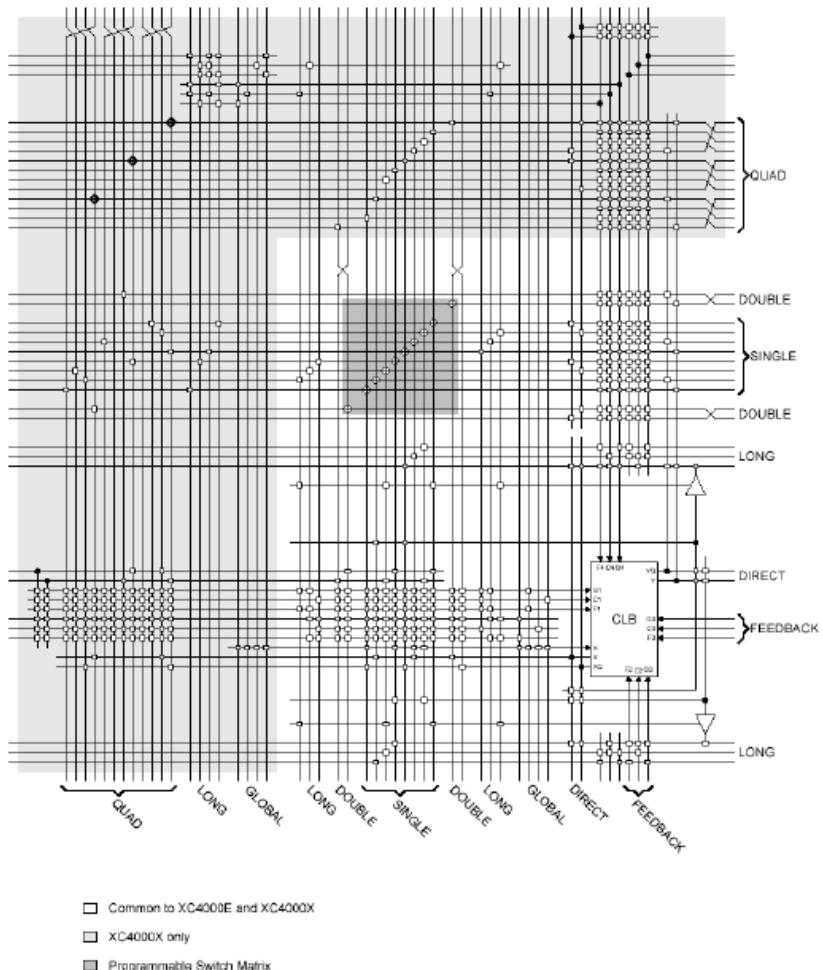


Figure 2-22 XC4000 Spectrum of Interconnection

Figure 2-22 shows an amazing set of connections available to the CLB variables. Along the lines mentioned earlier about “tiling”, you can envision the right hand, mostly unshaded section (containing the CLB) as being the basic tile. It needs to include the darkest shaded square, as that is the Programmable Switch Matrix (PSM). The XC4000 switch matrix – unlike the XC2000 and XC3000 – is very symmetric. The PIP sites within this switch matrix are each capable of connecting the four independent metal segments entering the diamond shaped sites. Standard PIPs are shown as tiny little boxes, where switch matrix PIPs are shown as diamonds. See Figure 2-24.

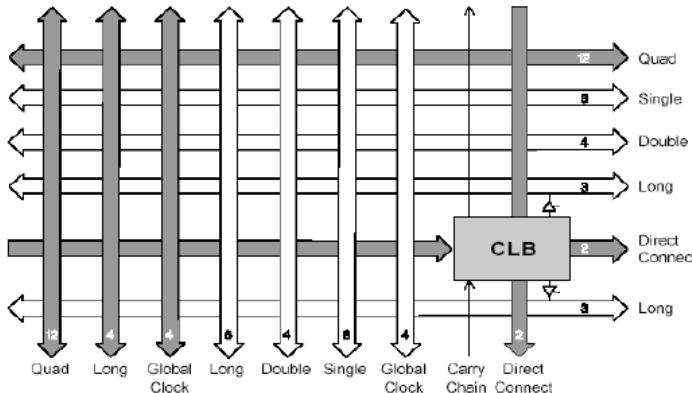


Figure 2-23 High Level routing Diagram of XC4000 Series CLB (Shading for the XC4000X Families Only)

It needs to be stated that the XC4000 architecture was originally developed as a 5 volt powered architecture. As time progressed and technology evolved, it was recast as the XC4000XL and XC4000XV, which were 0.35 and 0.25 micron families, operating with core voltages of 3.3V and 2.5V. The routing was not constant across all densities. Greater densities needed more routing, to support all the logic within the parts. Hence, as shown in Figure 2-22 (see legend), the XC4000XL and XC4000XV had additional routing beyond that offered in the five volt XC4000 family.

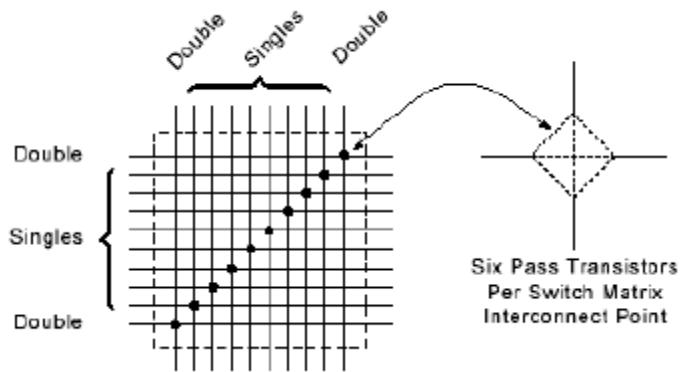


Figure 2-24 XC4000 Programmable Switch Matrix

As just mentioned, the XC4000 PSM had greater symmetry than its predecessors. This gave the software team a much better opportunity to do a great job of routing, with many connection choices being virtually identical.

Note that the dotted lines shown to the right represent the channels of pass transistors, whose sources and drains are permanently connected to the north, south, east and west metal segments. Each pass transistor has its gate driven from a latch in the configuration space. A latch comprises six transistors, so multiply that by six copies, giving 36 transistors per diamond. Such is the price for flexibility! Figure 2-25 shows how “single lines” directly connect between PSMs, but “double lines” skip over every other PSM. Jumping over the PSM is a little less flexible, in that the availability to connect is reduced, but double connection lines are faster. The appearance of the Xilinx “X” in the doubles illustrates that

the double connections are interleaved, so every other one jumps two that are not jumped by the neighboring double. Introducing a hierarchy of connections, where some are local (neighbor to neighbor), then some are less local (urban?), etc. until we have global connections that span the entire chip, gives options.

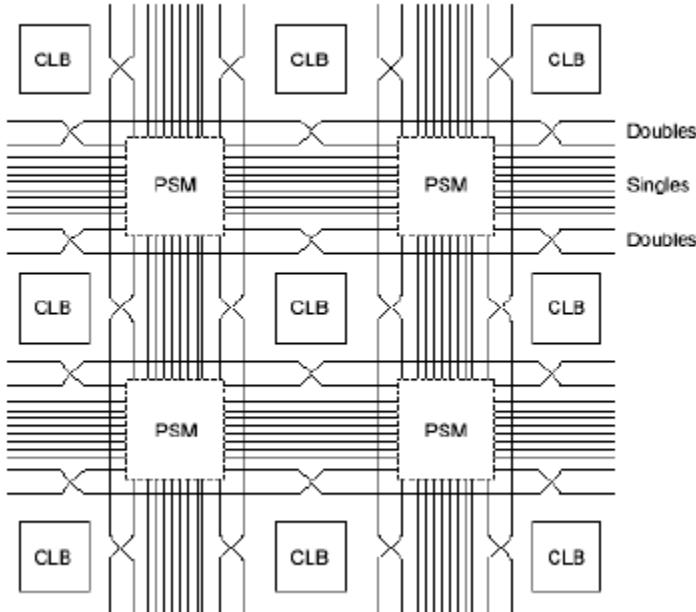


Figure 2-25 Single and Double-Length Lines, with PSMs

The XC4000XL and XV families also introduced “quad lines” capable of jumping over multiple PSMs, being faster, still.

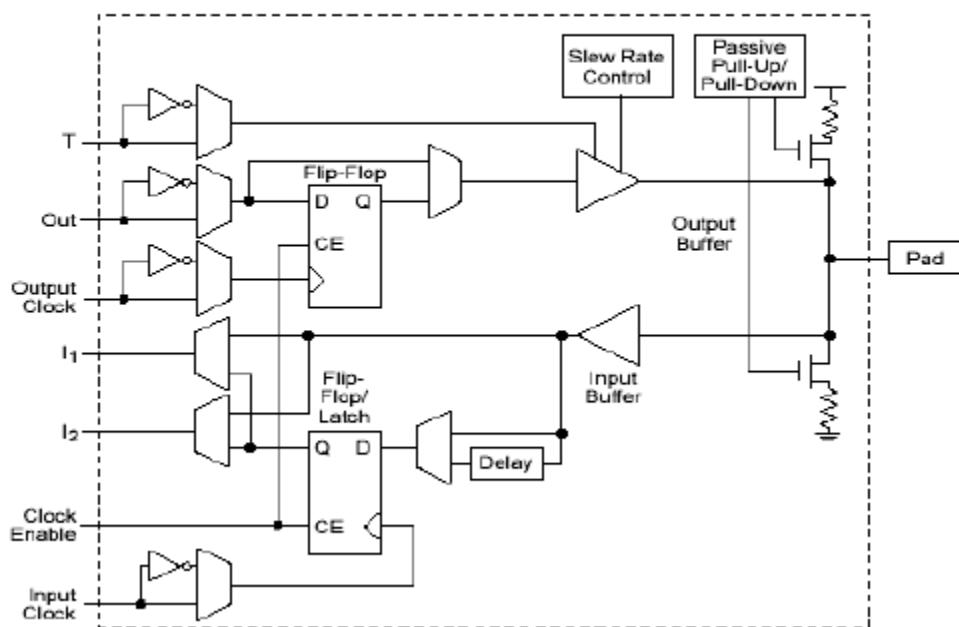


Figure 2-26 XC4000 I/O Cell

The XC4000 I/O Cell (Figure 2-26) has many of the qualities of the XC3000. There are two possible copies of the input signal presented thru multiplexers to I1 and I2, entering the routing area. The tristate, Out, Output Clock and Input Clock are output oriented, but fundamentally similar functions. An optional delay path for the input pin, to the D flip flop input, is available and is new for the XC4000. We will see this feature evolve moving into Virtex and Spartan 3 Families later.

The XC4000 family was released in different versions, while the electronics industry was developing many, new I/O and memory standards that would be designed into its successor, Virtex. But, the XC4000 did deal well with the main standard of its day, PCI.

### Anatomy of a PIP

This section deals briefly with the programmable interconnect point, the PIP. Having segmented interconnection fabric, is a critical factor to the success of Xilinx FPGA devices, and is at the heart of the famous Freeman Patent (reference 1). But, it does present problems, as technology evolves. Figure 2-27 shows the basic, early approach.

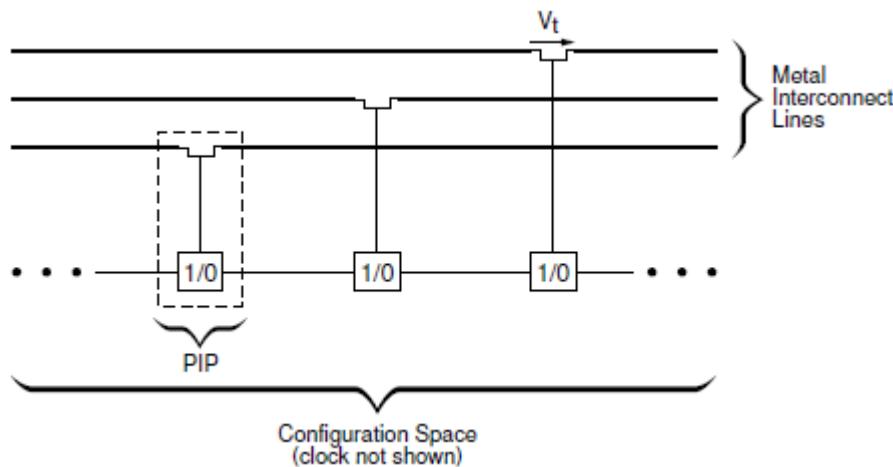


Figure 2-27 Programmable Interconnect Points (PIPs) Connecting Metal Segments

The XC2000 family had a PIP structure pretty much as shown in Figure 2-27. By applying an internal charge pump to the latches within the PIPs, it was possible to drive the pass transistor gates to a higher voltage, so it didn't drop the  $V_t$  voltage between its source and drain. There are issues with this approach in that it can create noise (charge pump). Also, higher voltages typically mean thicker oxides somewhere in the process. Ways to produce programmable connection are clearly important. Note that the PIP in Figure 2-27 is bidirectional, in that a connecting signal can pass in both directions. Figure 2-28 shows another approach, one direction buffering.

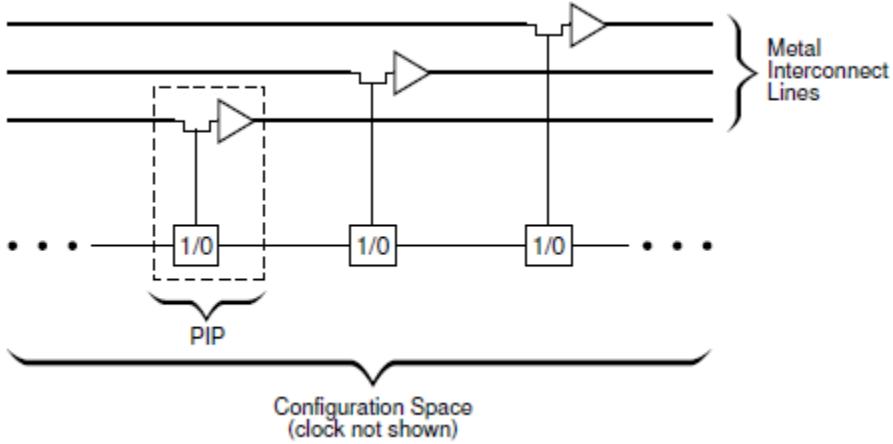


Figure 2-28 Buffered PIP (Not Bidirectional)

Figure 2-28 shows the buffered solution, that restores the  $V_t$  loss, but sacrifices the bidirectional capability, losing an advantage. To regain the bidirectional capability, Figure 2-29 shows another method. In Figure 2-29, instead of a single buffer, we insert back to back buffers with tri-state direction control so that only one buffer is driving.

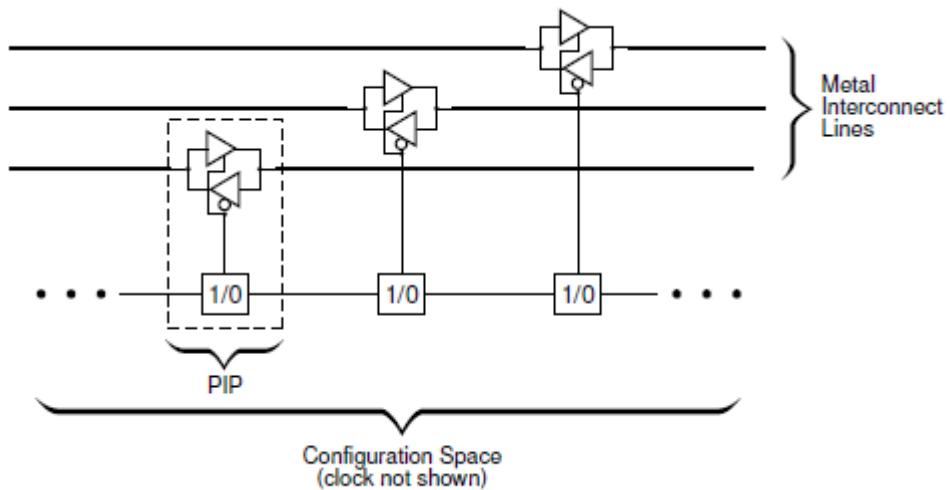


Figure 2-29 Bidirectional Buffered PIP

In Figure 2-29, we maintain the same basic configuration space, but now, each connection site takes on at least eight transistors, where there was a single pass transistor, previously. Clearly, there are a lot of ways to connect things. However, one of the most efficient structures in CMOS digital electronics is the multiplexer. Another approach to connecting signals with buffered multiplexers is shown in Figure 2-30. It is not bidirectional, but it includes independent selection of signals to be attached, and adds a buffer to compensate for the  $V_t$  drop, and speedup the overall structure. This method is used extensively in the Virtex families, with an independent Input Mux and Output Mux for every CLB. Note that the INPUT MUX and OUTPUT MUX can also be connected to form a directional connection facility, which can enjoy buffering on both ends, for speed, and connectible

assignment of metal on either end, to select among signals to connect. The select latches in the configuration space are still shown as little squares with 1/0 to control selection. One input select latch would be active, but multiple output latches might be active.

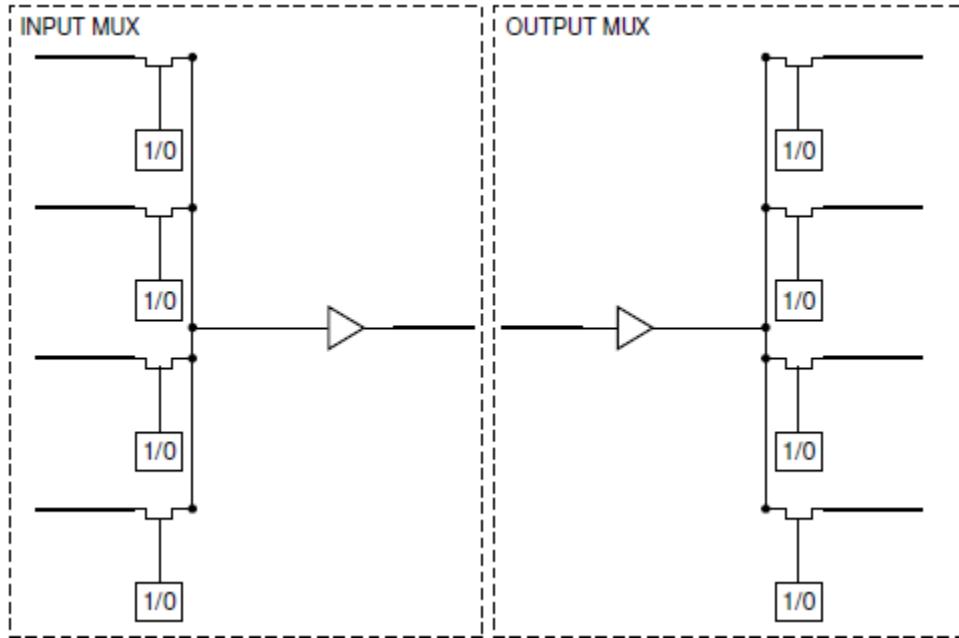


Figure 2-30 PIP Based Multiplexing

### Conclusion - Guessing the Future

As CMOS silicon process features continue to shrink, the voltage swing must also decrease, making creation of programmable circuitry more difficult. Xilinx is currently heading below 90 nanometer processes. In that, area the voltage swings inside a chip will be less than one volt. A CMOS threshold voltage ( $V_t$ ) is typically in the neighborhood of about 0.6 - 0.7 volts. This means silicon transistor switches will not be completely turned on or off, as in previous models. New challenges are presented, moving forward. Moore's Law appears to be intact, but it seems to be running a little slower than in the past. Xilinx continues to push on the bleeding edge of CMOS processes, to supply the latest architectures in the most modern technology to our customers, as we proceed. The path we are staying on is that of Virtex, moving forward.

### References

1. The most famous Xilinx patent is the "Ross Freeman Patent", which introduces segmented interconnection as well as shows how LUTs can be used to build up logic functions. It is US #4,870,302, titled "Configurable Electric Circuit Having Configurable Logic Elements and Configurable Interconnects". It is also referenced as RE34,363 as it is a reissue.
2. Another Ross Freeman patent covers making the LUT into a distributed SRAM, and is US #5,343,406. Its title is "Distributed Memory Architecture for a Configurable Logic Array and Method for Using Distributed Memory". It should be noted that Ross' co-inventor is Hung-Cheng Hsieh.
3. Another patent on using LUTs as SRAM is US#5,414,377 by Philip M. Friedin, and is titled "Logic Block with Look-Up Table for Configuration and Memory."

These are but a few of the patents that were covered by products in this chapter. We will introduce more, as the chapters unfold.

## Chapter 3 Virtex Architectures

### Introduction

Chapter One outlined the basic qualities that comprise a programmable logic solution – both hardware and software. For the first twelve years, Xilinx architectures represented variations on and extensions to the theme described in Chapter One. Those first dozen years are summarized in Chapter Two. Virtex architecture represents a departure from the original theme that began in the early 1998 timeframe.

Times were changing rapidly. Microprocessors and memories were embracing smaller feature CMOS technologies, to gain greater performance and functional capacity. To do that, voltages had to drop, and factors like noise, lots of switching signals and electromagnetic emissions became large issues. Sometimes, there appeared to be a “voltage standard du jour”, even though many of these voltage standards and switching protocols had in fact taken years to define. Designers needed a central clearing house for different voltage standards on their boards, and FPGA devices became the best place for that to occur.

On top of this, Xilinx and its competitors were fighting roadblocks defined by existing architectures. The “programmable interconnect point” (PIP) used in earlier architectures was running out of “steam”. New interconnect architectures were desperately needed. As well, Xilinx users that liked the distributed RAM capabilities of the LCA 4000 family sought greater densities and speeds. To those ends, Xilinx developed the Virtex architectures.

### Virtex

Virtex designers sought to provide a new approach to programmable logic. They recognized that programmable logic elements and programmable interconnect – the “fabric” of FPGA devices was a great value to designers, but this need supplementing with additional fixed resources that had a greater transistor efficiency than fabric had. For a given area, programmability comes at a price – once you settle on what your design is, the flexibility is no longer as precious. Now, designers need greater density, and only optimized “fixed” resources can provide that. Placing just the right “fixed” resources within the fabric results in a much better balance of efficient and flexible structure. This was not trivial.

Chapter Six details the Block RAM structures, but consider the fact that few designers seek exactly the same RAM architecture. Some need it to be 4 bits wide by 1K deep and others need 32 bits wide by 16 K deep. How do you satisfy everybody? It can't be a simple “fixed” resource. It must be efficiently laid out in transistors, but still have the flexibility to adapt to many different usage models. Nonetheless, the block RAM structures are a primary factor that made Virtex a “dynamite” FPGA family. The same performance gains that microprocessors found by incorporating high speed caches were now found in programmable logic, giving similar payback as microprocessors gained. Figure 3-1 shows the basic idea. Block RAMs are shown simplistically, as are the Configurable Logic Blocks (CLBs), IO Blocks (IOBs), the Delay Lock Loops (DLL) and the “Versa Ring” I/O connection fabric. Not detailed in Figure 3-1 are the interconnect structures that exist between the CLBs. The Delay Lock Loops provide a timing resource to manage clocks

applied to larger FPGA devices where signals may span physical distances resulting in substantial clock skews. The core voltage requirement for Virtex is 2.5volts, as it is built on a 0.25 micron CMOS process.

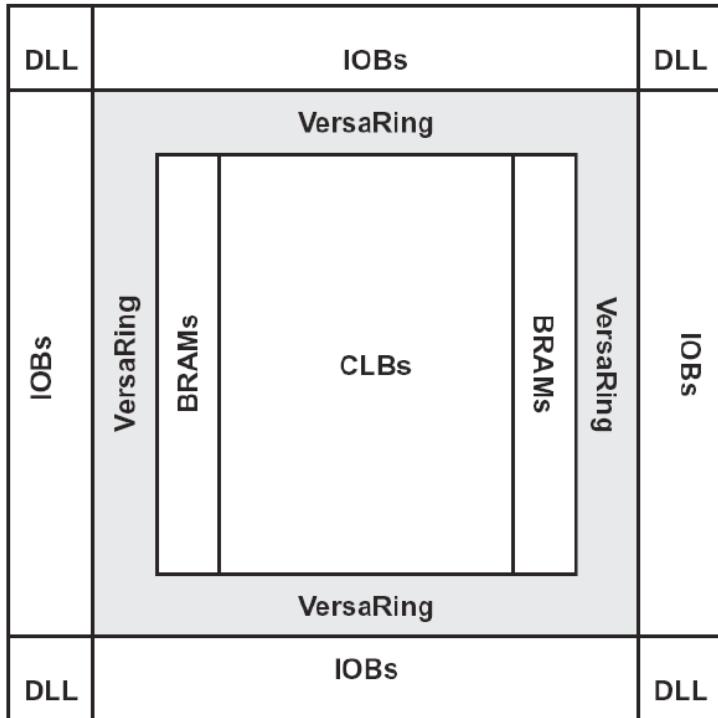


Figure 3-1 Basic Virtex High Level Architecture

Building block definitions evolve with architecture. In the very early architectures, a LUT and flip flop were the building blocks of greatest interest. This was originally referred to as a CLB, but clustering of resources has an advantage – performance. Specifically, if connections can be made naturally among neighboring LUT/flop structures, this can be an advantage from both performance and routing efficiency. Neighboring cells connect through simple multiplexer connections, and avoid presenting signals to the more global routing, where they would have to compete with signal attachments that need to be made over greater distances. Hierarchy became a key factor. Introduce degrees of local interconnection to reduce the need for general connection became a rule. This approach, improved overall performance.

Figure 3-2 shows the elementary clustering used within local “slices” with a Virtex CLB. Multiple Slices comprise a CLB, but even this varies among the more advanced Virtex architectures, as we shall see.

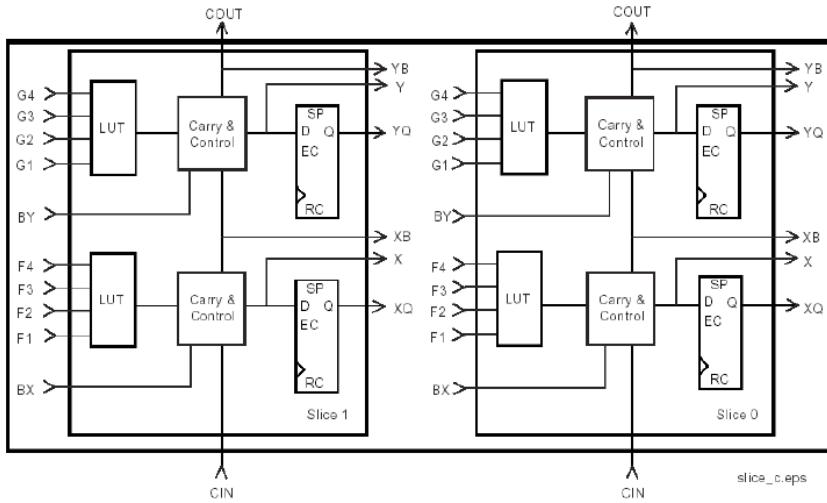


Figure 3-2 Virtex CLB

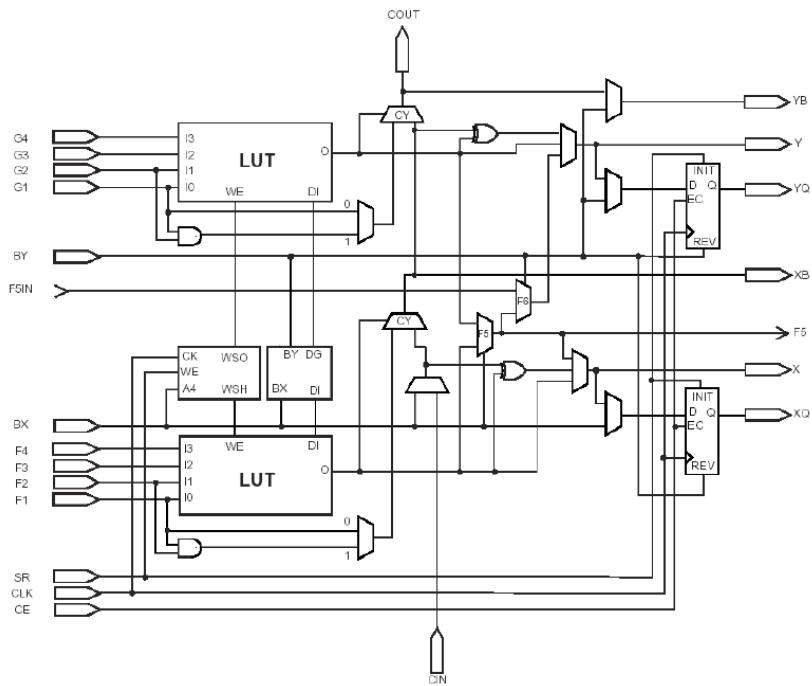


Figure 3-3 Detail of Virtex Slice

Figure 3-3 shows the extensive multiplexing of connections within the Virtex slice. Various connections include usage of the LUTs as fast adders, with extra circuitry included to pass fast carry signals between neighboring LUTs. There are also bypass paths, permitting LUTs to be viewed as routing resources, versus logic functions! (Check the path from BX to either XB or F5 in Figure 3-3). Remember that trapezoidal multiplexers without explicit select inputs have their select inputs in the configuration space, and are not dynamic multiplexers. They simply make a user defined connection after the configuration operation.

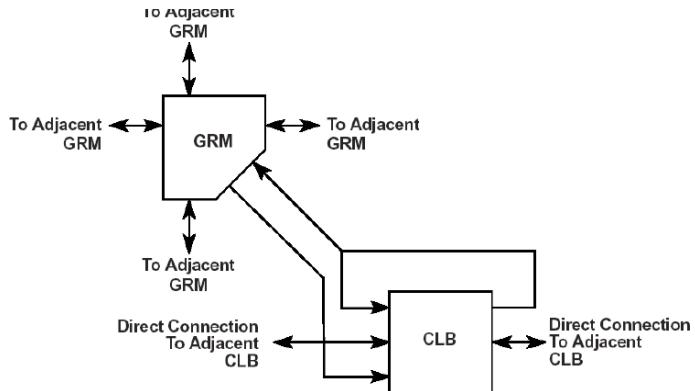


Figure 3-4 Virtex Connection Hierarchy

As shown in Figure 3-4, interconnection can be thought of as connecting within a hierarchy. The feedback around the CLB (multiple slices), provides local connections. Then, there are also connections to adjacent CLBs using multiplexers. Next, there are Global Routing Matrix (GRM) connections, between adjacent GRMs. Figure 3-5 shows the global connections available within the Virtex FPGA, whereby CLBs access tristate lines permitting access clear across the chip.

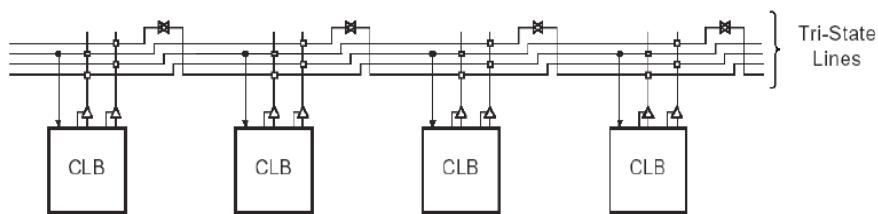


Figure 3-5 Tri-State Line Interconnection

One key aspect of the Virtex approach to building an FPGA is that the number of unbuffered PIPs and short segmented interconnect paths is reduced over previous methods used in the LCA 2000, 3000 and 4000 parts. The use of lots of buffered multiplexers to deliver signals among the slices is critical to the more predictable and fast time delays. Figure 3-6 shows a couple of multiplexers that deliver input signals (IMUX) and direct output signals (OMUX). Multiplexing is the main method of making connections from one slice to another, in general. There is also a very large interconnect array that these feed into and out of, but the facility that singles out which slice signals connect to other slice signals is based on buffered and un-buffered multiplexers. The “doubles” (jump two CLBs) and the “hexes” (jump six CLBs) are also multiplexed accordingly. Figure 3-21, shows another view of signal multiplexing, of the various “jumps”, later.

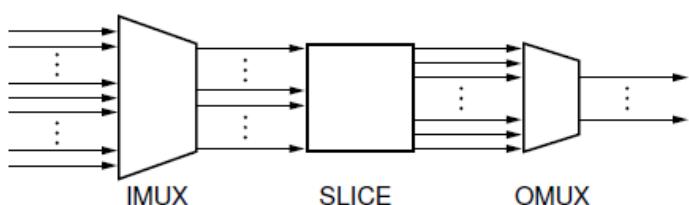


Figure 3-6 IMUX and OMUX within the GRM

An important aspect of interconnect to remember is that the various functional blocks (BRAMs, etc.) and the I/O pins must also be attached to the CLB logic, so the interconnect approach must be generic enough to be modifiable to meet the needs of connecting those resources, also.

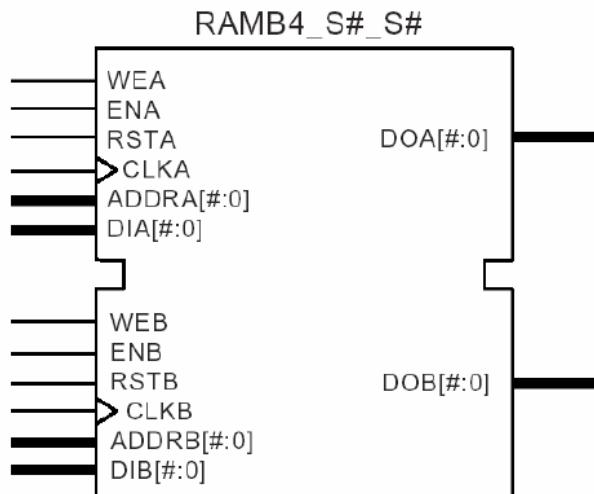


Figure 3-7 Dual Port Block RAM Symbol

Figure 3-7 shows the variable nature of the BRAM structure. Two fully independent data/address ports (A and B) are shown, with completely independent control lines (WE, EN, RST, CLK). The structure is very general. It is possible to simultaneously read and write to any address desired. Contention is possible, and must be accounted for by the user's external design circuitry. However, it is the correct choice to provide this generality. Any restriction that Xilinx designers made on this structure would result in dissatisfying some customers, while pleasing others. With this structure and external logic, it is possible to please almost everybody. The design market has shown this to be the right choice. See Chapter Six for details.

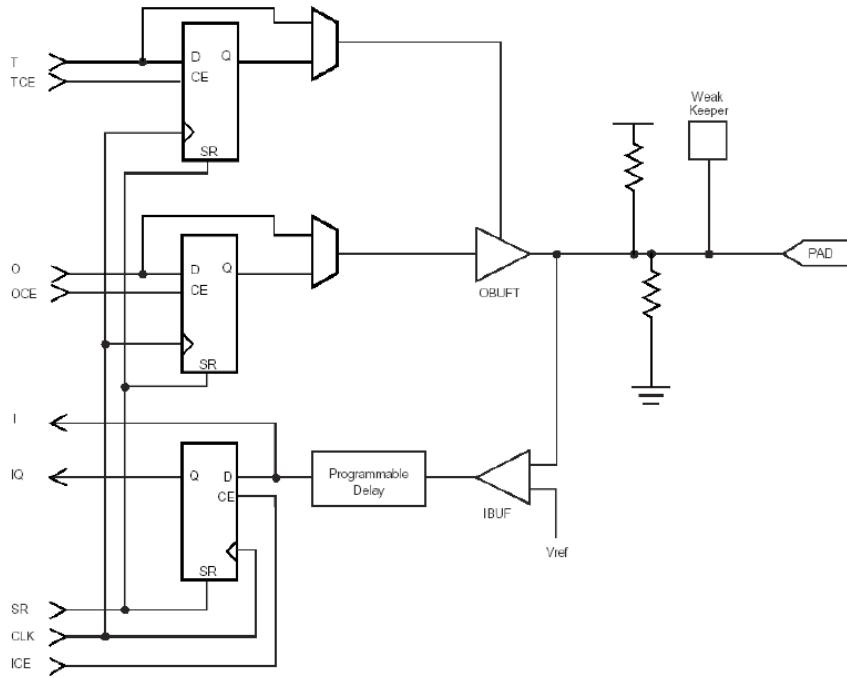


Figure 3-8 Virtex I/O Pins

Xilinx is famous for the flexibility of its I/O pins. Early FPGA devices were a welcome change from ASIC solutions, with the high flexibility of their I/O pins. As time progressed, the I/O pins tracked designer needs. Figure 3-8 shows that signals on the left side of the diagram, coming from the fabric, can land in a flip flop, for pin synchronizing, or bypass the flip flop altogether. The signal is then presented to the output buffer's input, where a Tri-state control option can be driven from an adjacent independent signal. Then, the output is driven to the pad with optional pull up and pull down circuitry, or a weak “keeper” circuit. The “keeper” circuit is optional, and permits tri-stated signals to be gently driven to either logic one or zero. This is a simple CMOS holding circuit, with negligible current driven. The tri-state circuit (top flip flop and bypass multiplexer) is a compelling advantage in the PCI world, where driven signals are referenced to clock edges. This approach, is now being used by other standards, as it maximizes available time on a bus, with “out of band” negotiation removing bus latency, while driving performance higher.

Finally, the input signal has the option of direct access to the routing/logic fabric, registering at the input, or possible setup delay modification. All are good options and can be useful, where needed.

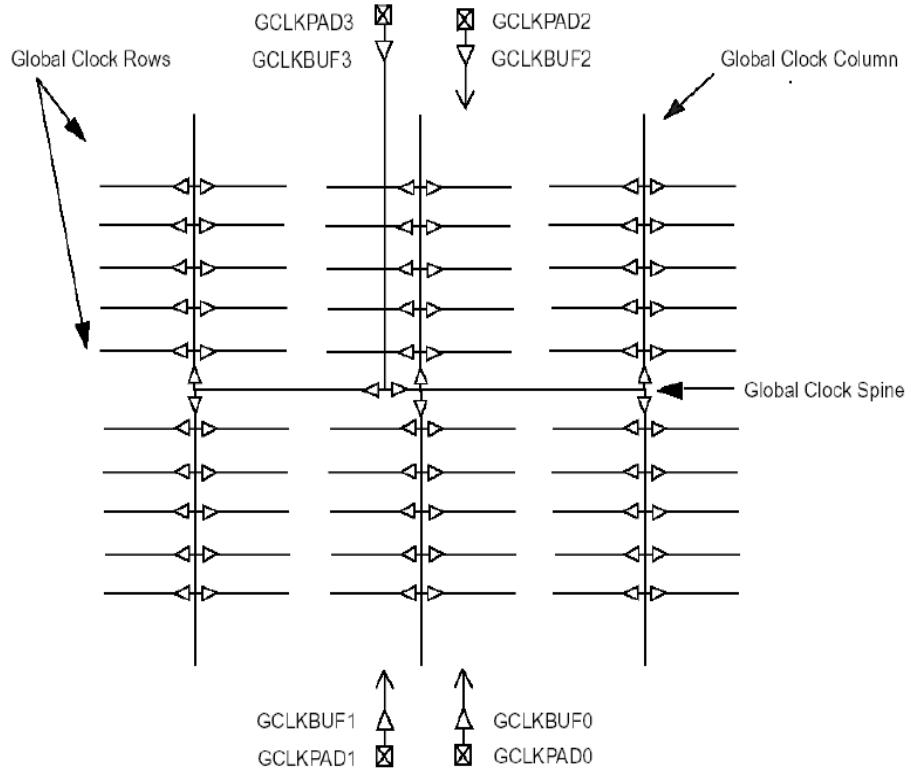


Figure 3-9 Virtex Clock Net

The clock network of Virtex (Figure 3-9) does the “heavy lifting” for clock signal delivery and skew management. Delay Lock Loops provide the ability to tweak and tune the incoming clock, but the clock network, with its uniform delay geometry, manages the consistent clock delivery to

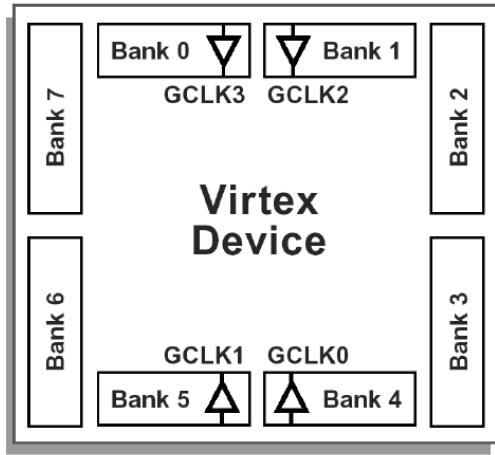


Figure 3-10 Virtex I/O Banking

multiple clocking domains within the chip. This behavior is vital for the solid performance of the FPGA.

To be a general purpose signal translation device requires having the flexibility to translate

signals between many standards, simultaneously. Figure 3-10 shows how the eight banks (two per edge) of I/O signals are arranged to do this. Each I/O bank is independent, so in theory, as many as eight distinct voltage standards can enter and exit the chip at any time. Frequently, designers use less than eight, so often many banks are configured to the same voltages, but any Virtex part can have as many as eight or as few as one voltage standard at the I/O pins. Details for proper termination, and the general management of the I/O pins are handled in the Virtex design software, which helps users manage their I/O resources.

Remember, the Virtex core voltage was 2.5volts for the first version at .25 microns.

### **Virtex E and EM**

Immediately after the release of Virtex, users became enamored of the BRAMs. It became evident very quickly that users needed more, so with little hesitation, Xilinx introduced the Virtex E and Virtex EM families. To provide the greater densities, it was necessary to translate the architecture to a smaller feature CMOS process – 0.18 micron, making the core voltage requirement for this family 1.8 volts. Figure 3-11 shows the basic idea. For many applications, having more BRAM structures within the FPGA, make sense. Figure 3-11 shows that the BRAMs are interleaved between a certain number of CLB/routing tiles, so that logic could be available to solve design problems, or alternately stitch the BRAM pieces together into the specific sizes that designers could best use. Creating structures like first in first out (FIFO) buffers became standard applications. Never before could you buy a FIFO where the input data could be in one arrangement – say 16 bits, and the output data could be another – say 32 bits, and each could be assigned to a different voltage standard, if attached to the I/O pins. Even with smaller processes, the FPGA functionality is loaded to the point that die sizes are relatively large. Additional clock skew requirements are addressed by including more Delay Lock Loops.

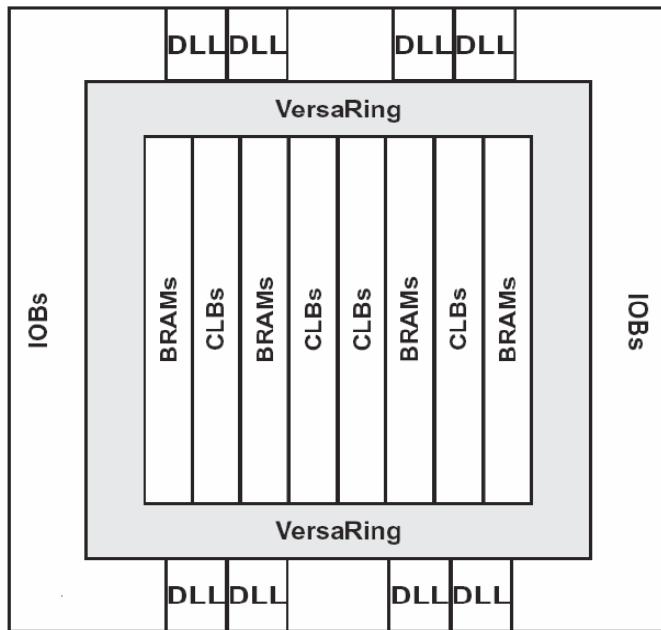


Figure 3-11 Virtex E and Virtex EM Architecture

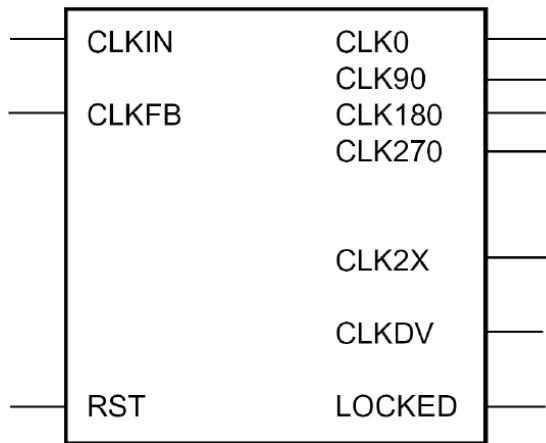


Figure 3-12 Symbolic View of a Delay Lock Loop

Figure 3-12 shows the high level DLL symbol, where the primary input is CLKIN, a feedback site CLKFB is below that, and a Reset input permitting re-initialization of the DLL is below that. With software, it is possible to select signals that track the input, are 90, 180 and 270 degrees out of phase. Clock doubling (2X) and clock dividing (CLKDV) are also possible. Providing these key clocking options takes time, for synchronization, so the Locked signal alerts external signals regarding the attainment of synchronization of the feedback clock with the input clock. The all-digital circuitry is a great step toward improved clocking.

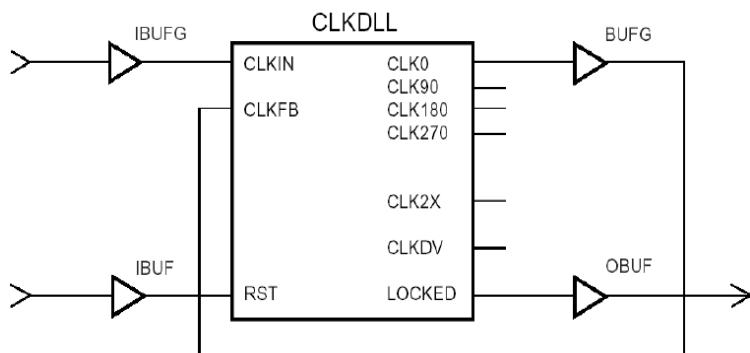
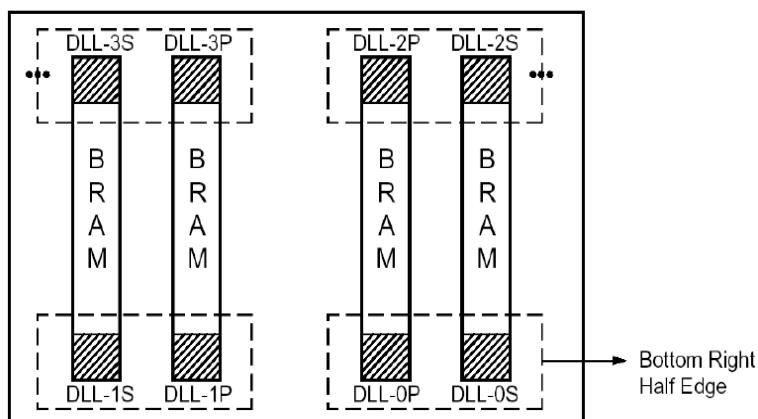


Figure 3-13 Standard Clock Tracking DLL Configuration



### Figure 3-14 Primary and Secondary DLL structures

Figure 3-13 shows a standard configuration, where a clock net within the chip is attached back onto the CLKFB point, and achieves “lock” to the incoming clock. Figure 14 shows the assignment of both primary and secondary DLL structures within a Virtex E/EM array. Note the “P” and “S” designators in Figure 3-14. The physical layout is to place them on the ends of the BRAM columns, in this particular family. That way, they are close to the memories in the center, and available to appropriate I/O structures on the ends. The secondary DLL is typically attached to the same group primary DLL, when forming 4X clocking, by simply setting each DLL to 2X, and cascading them.

## Virtex II

The success of Virtex, Virtex E and Virtex EM gave Xilinx designers the customer feedback needed to drive new directions for embedding flexible, but specific blocks within the programmable fabric. In particular, with both BRAM and fast adder circuits available in the Virtex parts, digital signal processing only needed fast multipliers to make the combination a formidable threat to existing microprocessor DSPs. Being able to calculate a product followed by a sum (multiply-accumulate or “MAC”) then drop the result into a nearby BRAM made all the difference. Abundantly available flip flops within the fabric supplied additional pipelining to smooth out the dataflow, and “voila!” we get a powerful DSP engine, capable of parallel computation other methods cannot not approach. But, that’s not all.

The BRAMs also encouraged users that wanted to create internal microprocessors, capable of rapid data interaction, but more area efficient than straight fabric solutions might yield. Soft microprocessors – “MicroBlaze” and “PicoBlaze” were made available with design support so users could get their code developed and debugged quickly.

With greater available functionality, designers could now create much more elaborate designs – literally whole systems, representing substantial investment in intellectual property. To that end, Xilinx designers inserted encryption circuitry holding the keys for Triple-DES within the volatile part, using a backup battery arrangement to keep the key alive. This allowed designers to have their encrypted configuration patterns held in external PROMs, with decryption occurring during configuration. It also permits encrypted field upgrade of designs.

More industry I/O standards arose, needing support. Meanwhile, Virtex II logic fabric provided greater flexibility and density, with but a few simple adjustments to the original Virtex architecture. For starters, let’s look at Figure 3-15 showing the improved half slice.

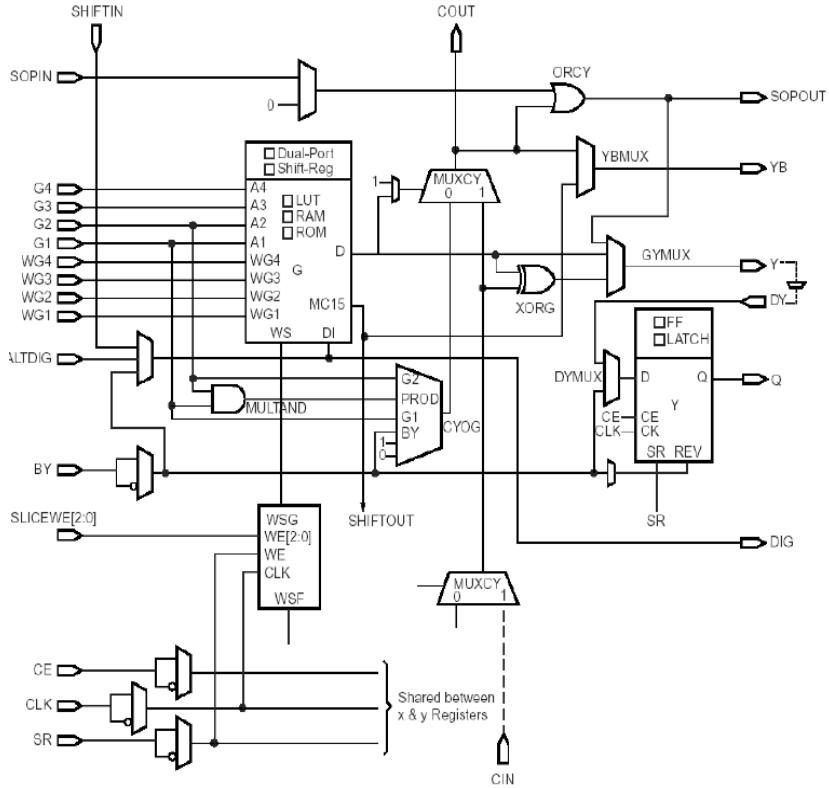


Figure 3-15 Virtex II Half Slice

The main logic engine in Figure 3-15 is the G LUT. Note the check boxes indicating the optional capabilities as a dual port RAM, a 16 bit shift register (SRL 16), a LUT, a RAM or a ROM. The 16 bit shift register is particularly intriguing, and is described in greater detail in Chapter 6, as well as the dual port RAM ability. Address inputs A1-A4 represent the read address, where WG1-WG4 represents the write port address when using the LUT as a dual port RAM. Key flexible capabilities include the vertically directed MUXCYs, which are useful for both arithmetic, and forming wide multiplexers between multiple slices, when appropriately combined. The clock enable (CE), clock (CLK) and set/reset (SR) are shared across both x and y registers within the slice (x is not shown in Figure 3-15).

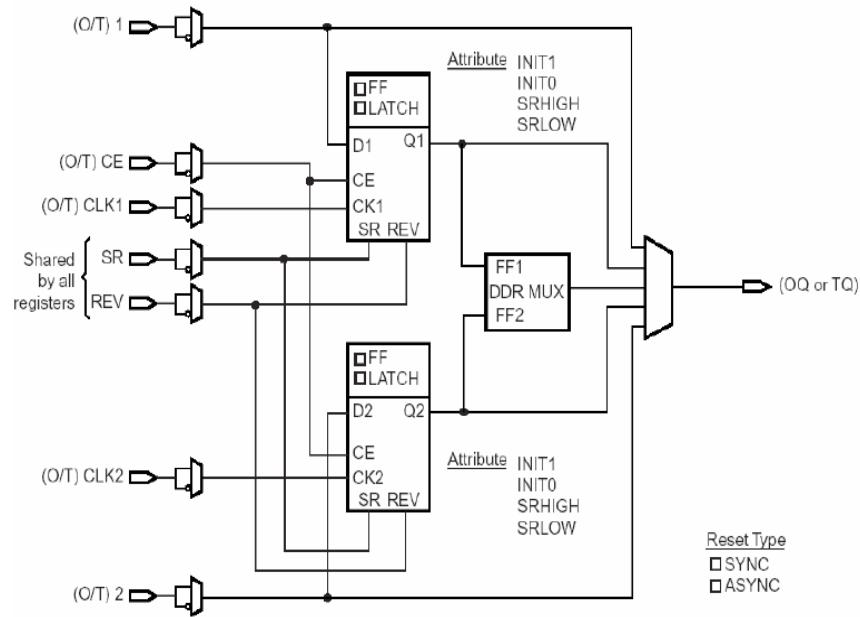


Figure 3-16 Double Data Rate I/O capability

Because memory standards evolved to deliver/accept data on both the rising and falling edges of a clock, circuitry to perform that needed including. Figure 3-16 shows the Virtex II approach, where a DDR multiplexer selects between two flip flops clocked from different clock sources. Figure 3-17 shows how clocking can be derived from either a 180 degree phase inverted single clock, or from the Digital Clock Manager. The DCM – discussed in depth in a Chapter Nine, is critical to manage data setup and hold times with respect to both edges of a clock. Figure 3-17 shows the preferred method of phasing the clock, as it guarantees virtually a perfect 180 degree phase shift on the two clock edges. The DCM evolved from the Virtex DLL, and is detailed in Chapter Nine, but also discussed further in this chapter, later.

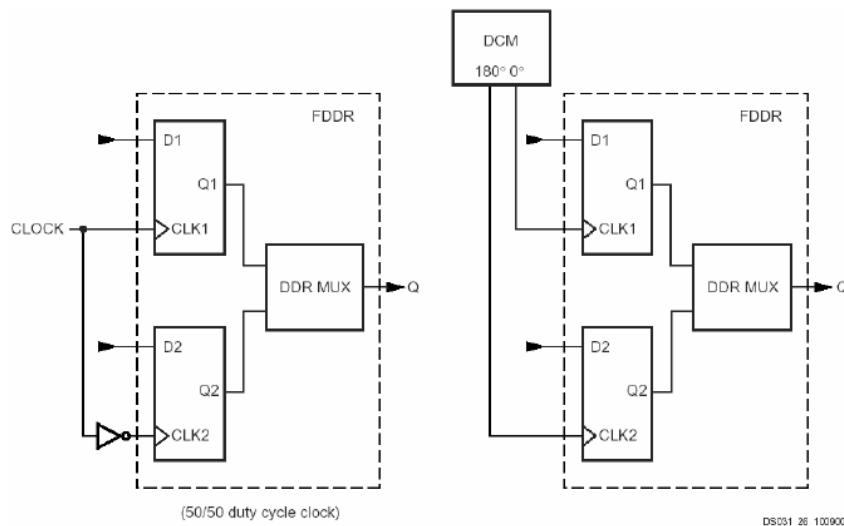


Figure 3-17 DDR Clocking with and without the DCM

Figure 3-18 shows greater detail on the use of double data rate registers, acting on the output register, the tri-state control register and the input register of a standard Virtex II I/O pin. Orchestrating all three is required to extract maximum bandwidth from memory interface standards like HSTL, SSTL and bus standards like PCI. The protocol behavior of each of these has been carefully crafted to provide the maximum data throughput available with that particular industry standard.

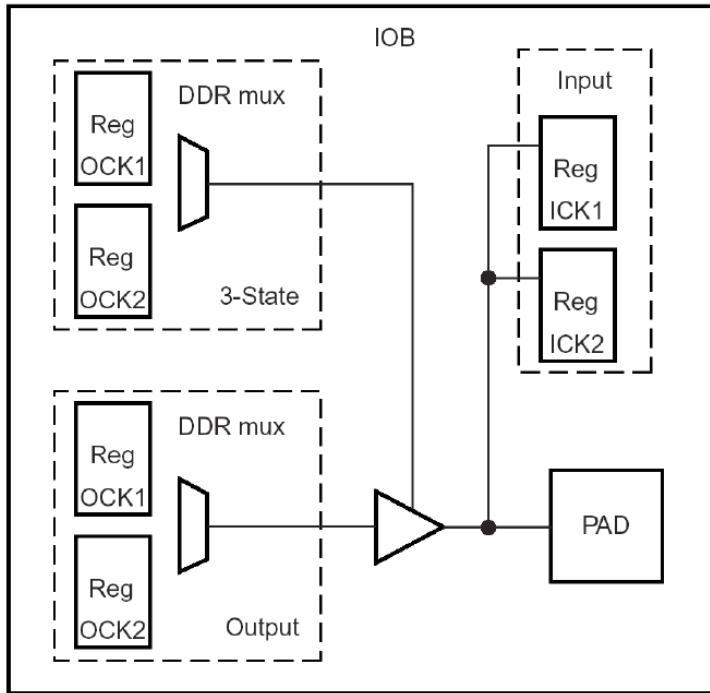


Figure 3-18 DDR facilities associated with an I/O Pad

Figure 3-19 shows some of the interconnect capabilities in the Virtex II CLB. Like Virtex, the architecture provides local feedback sites within the CLBs, as well as fast next neighbor connections, to assure maximum speed of transfer between adjacent slices and CLBs. The switch matrix routing extends the reach of connection to a more general audience of CLBs inside the chip assuring connection needs within the chip are met. Each successive Virtex family modifies the structure initially described in Figures 3-4, 3-5 and 3-6, to suit the densities and new blocks being inserted into the current architecture. Fine details on the routing matrix construction are seldom needed by users and are best displayed in the editor tool, within the Xilinx design software.

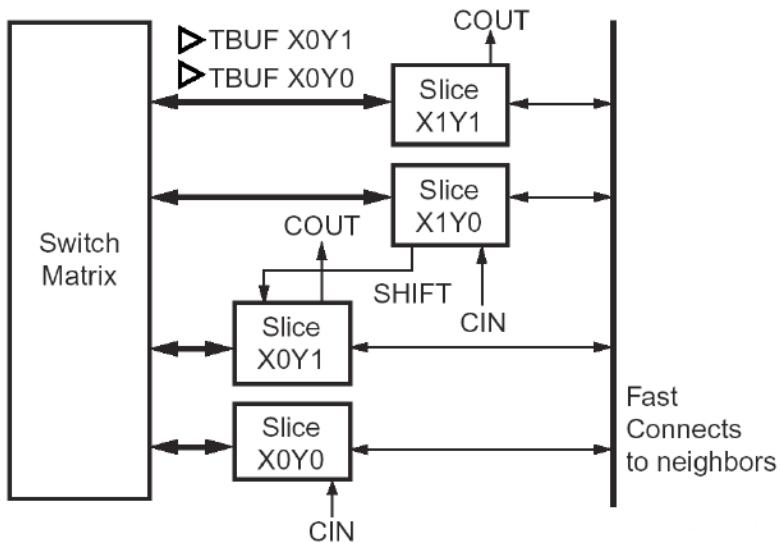


Figure 3-19 CLB and Some Routing Resources

Figure 3-20 shows the special arrangement of CLBs that are located near I/O cells. In particular, the I/O routing tiles must handle the ability to assign specific neighboring pin sites so that they behave properly as parts of a differential pair. Minimum physical distance between adjacent differential pairs is required, to guarantee uniform time delays and correct PCB impedance distances are met. Managing large numbers of paired pins is difficult, so additional software – the pin and area constraint editor (PACE) was developed to aid users in properly assigning pin-outs for best results. Chapter Four will show the software evolution required to support the Virtex architecture evolution.

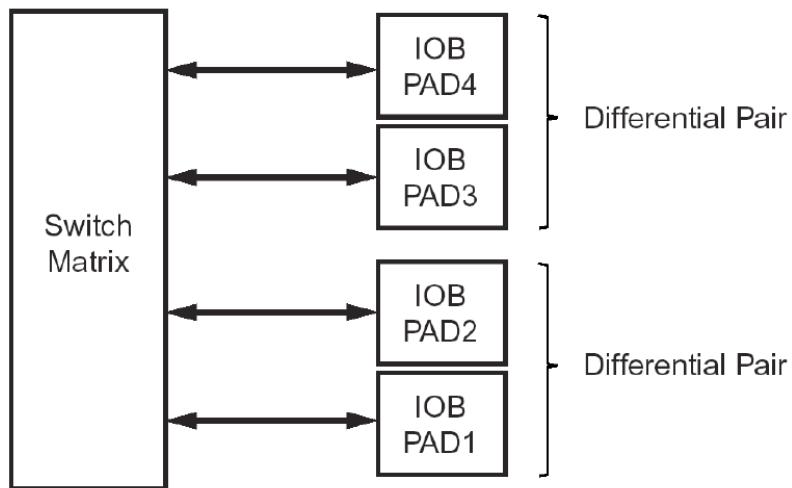


Figure 3-20 Differential Pair I/O Pads and relation to Routing Switch Matrix

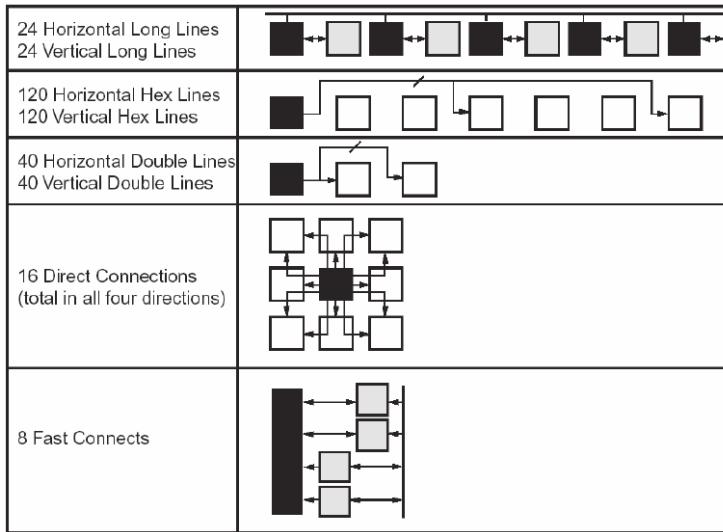


Figure 3-21 Another View of Routing Matrix Connections

Figure 3-21 shows the various possible connections among CLBs in Virtex II. Figures 3-4, 3-5 and 3-6 showed the global routing matrix, with its IMUX, OMUX, Hex connections, double connections and long lines, where Figure 3-21 shows another viewpoint, focusing on the range of possibilities of connections that can be accomplished. This is substantially different from the methods of the early channel routed Xilinx architectures, and is both faster and more uniformly timed, giving designers less frustration in dealing with large timing variations. It still maintains a great deal of regularity, permitting Xilinx chip designers to create modular architectures, while providing consistency to the Xilinx software developers, who deliver capable tools in a reasonable timeframe. Still, variations on this structure are needed to support the various specialty blocks embedded within the fabric.

For instance, additional specialty connection networks are introduced to guarantee that BRAMs and their neighboring multipliers can properly pass data operands as swiftly as possible between the memory and the arithmetic functions. Having the required DSP functionality, and without appropriate connectivity would squander the valuable gains made with the memory and multiplier pairs. Figure 3-22 shows how this is managed within the Virtex II parts. As with the DCM, the BRAM and multipliers are given greater detail in subsequent chapters.

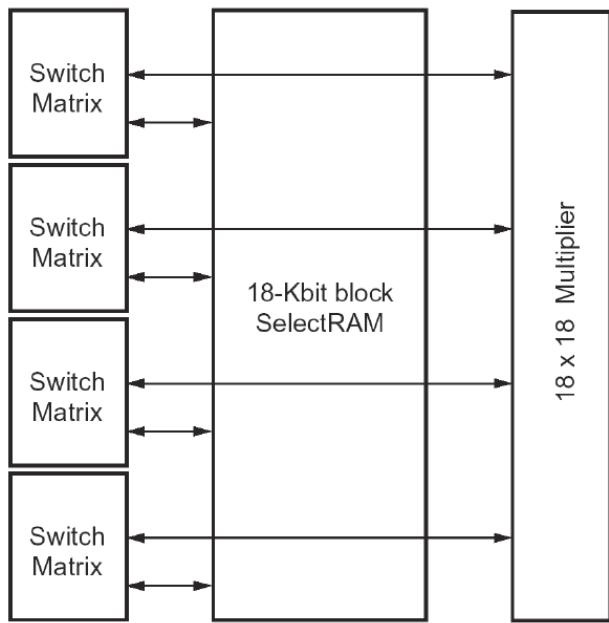


Figure 3-22 BRAM, Multiplier and Switch Matrix Organization

Figure 3-23 simplistically shows the 18 X 18 bit, two's complement multiplier. Data can be treated both asynchronously in a combinational sense, or it can be synchronously managed, as with pipelined systems. Larger family members literally have several dozen multiplier/BRAM pairs, that can effectively “loop unroll” the most devious DSP algorithms, producing extremely fast, real-time solutions to today’s serious problems.

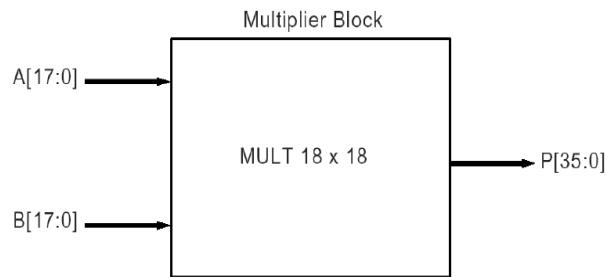


Figure 3-23 Two's Complement 18 X 18 Multiplier Symbol

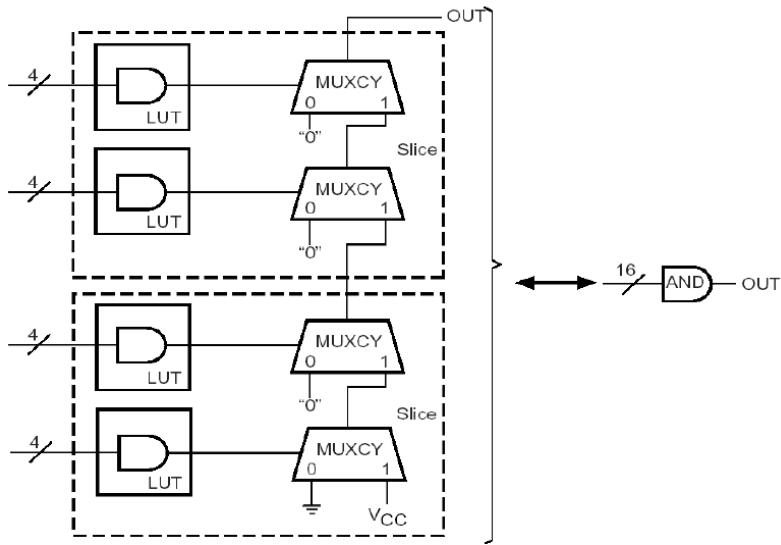


Figure 3-24 Building Large Input AND functions

One interesting paradox of digital design is that some logic functions require lots of operands, and need a fairly “wide” logic structure to create them. Figure 3-24 shows one method that Virtex II FPGA devices employ to build up very wide AND functions. This method, combined with the OR gate (ORCY in Figure 3-15) capabilities within the CLB permit very wide Sum of Products to be created, using the combined LUTs and multiplexers inside the slice.

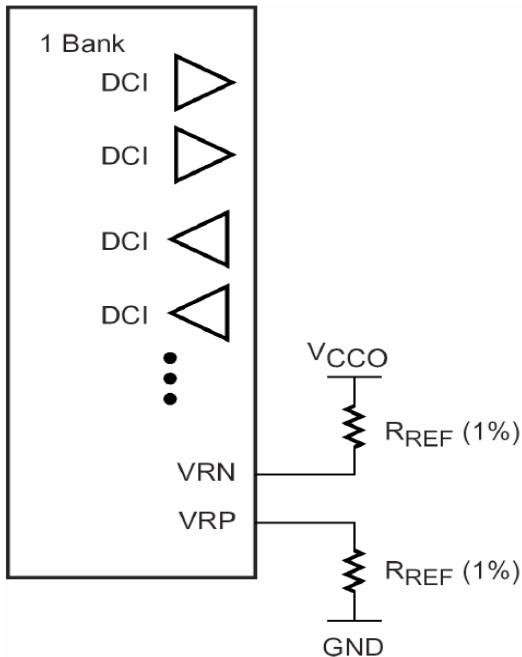


Figure 3-25 DCI Used with Multiple Voltage Standards

Shifting attention to the I/O structures, Figure 3-25 shows the digitally controlled impedance (DCI), which delivers automatic impedance matching into a printed circuit

board by providing external matching resistances to the board characteristic impedance. Knowing the PCB impedance, selected equivalent resistance is attached to the reference sites for a given I/O bank, automatically setting the output impedance of the output pin driver to match the PCB trace impedance, thereby reducing reflections. This automatic circuitry eliminates the need for parallel or series termination on geometrically close I/O pins or balls, which may not accommodate external impedance matching resistor packs, easily. Figure 3-26 shows some of the different configurations that can automatically be terminated using the DCI resistance connections. Chapter Eleven provides more DCI details.

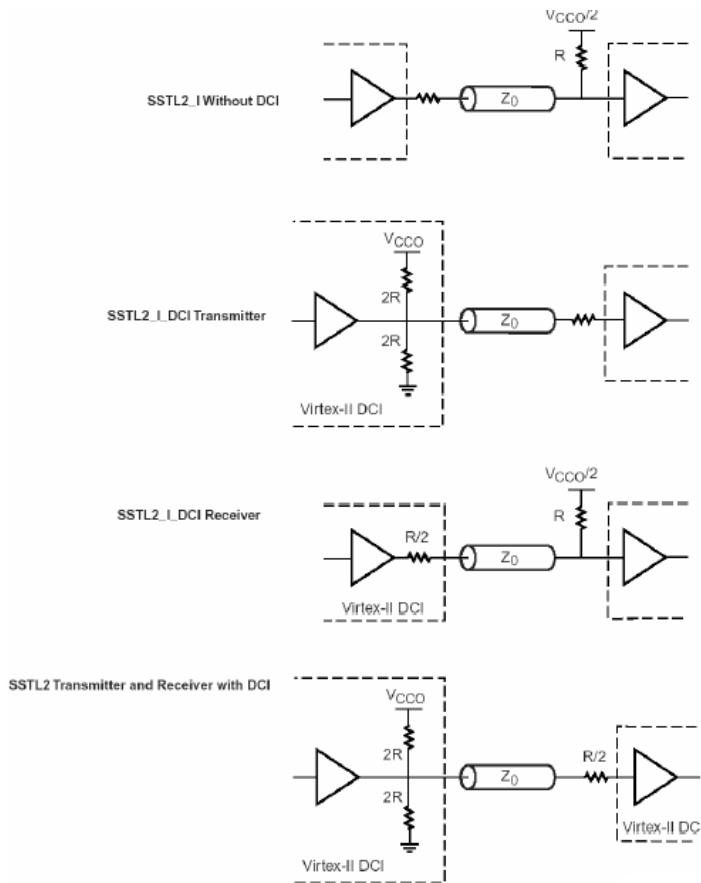


Figure 3-26 Examples of DCI Termination Possibilities

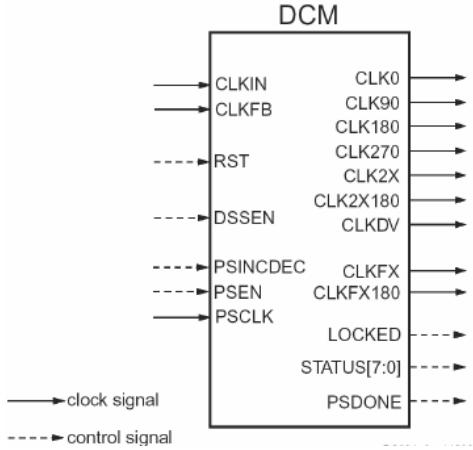


Figure 3-27 DCM High Level Symbol

Note in Figure 3-27, that additional capability beyond the simpler DLL (Figure 3-13) of Virtex has been included in the digital clock manager capability. The DCM has additional status, extra phase shifting capability and extends the lock frequency of the earlier DLL structures. Again, greater DCM details are in Chapter Nine.

Virtex is about adding specific blocks to FPGA fabric, and the most exotic block may be the addition of a microprocessor. This enters the family into the realm of the “system on a chip”, which has been the holy grail of the ASIC world for the previous fifteen years. At last, there is the ability to combine both an efficient programmable processor capability with programmable logic fabric, permitting appropriate partitioning of logic and programming functions, as embedded system designers desire. Figure 3-28 shows the Power PC 405 processor embedded into the Virtex II Pro architecture, adding the “pro” to the family title.

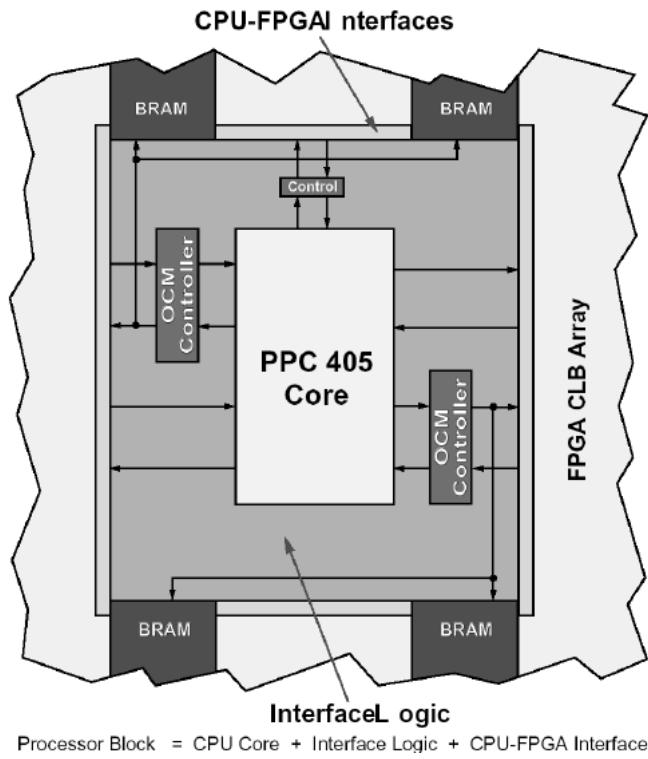


Figure 3-28 Virtex-II Pro Microprocessor Core

The Power PC Xilinx uses, was developed by IBM, but when embedded into the Virtex II fabric, substantial peripheral modification was made assuring full capabilities of the PPC405 are delivered to users. To that end, the I/O pins were laid bare and separated into independent input and output signals. This increased both the connection sites available to the fabric, as well as the flexibility that could be attained. A key internal bussing resource is called the On Chip Memory (OCM) bus, permitting direct access into the BRAMs. Additional interface solutions exist to efficiently access external memory and other on chip resources. To support users and deliver the greatest functionality, additional internal and external memory interfaces were developed for hardware support. In addition, a wide range of software assemblers, compilers, performance enhancement tools are offered as part of an Embedded Development Kit (EDK).

Figure 3-29 gives a better overall picture of Virtex II Pro, as it shows the geometrical arrangement of the various BRAMs, multipliers, PPC405, multi-gigabit RocketI/O Transceivers (aka MGTs), DCMs and so forth. The mix of functions varies among family members, to better suit individual design needs. When loaded with fast 10 Gb/s transceivers, the Virtex II Pro parts are referred to as "Virtex II Pro X".

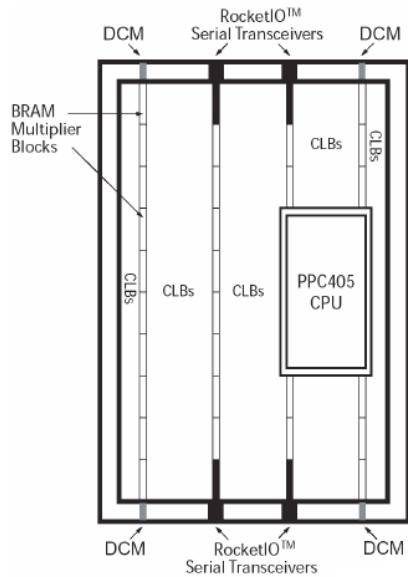


Figure 3-29 Embedding PPC, BRAM, Serial Transceivers, Multipliers and DCMs all in a Single FPGA

### A Pause...

At this point, the Virtex families are well on the way, and we need to take a moment, and look at how the inclusion of the microprocessor as a block failed to materialize the dreams of a universal system on chip. It took Xilinx through Virtex 4, 5 and 6 to recognize the problem.

In spite of Virtex 5 being the most commercially successful family at its time on the 65nm technology node, the vast community of system's engineers were not interested. Why? Well, it did not look like a microprocessor: it was a microprocessor surrounded by the FPGA, which literally made it inaccessible to the designers we had hoped to reach.

### Zynq

The solution to the system on chip issue came about in the 28nm node where Xilinx kept the Virtex Architecture and features, but re-architected the processor side to be an entirely independent (having everything it needed, including its own IO pins) and based on the well accepted ARM™ A7 processor family. “Zynq-7000 AP SoC (all programmable system on chip) devices integrate the software programmability of an ARM-based processor with the hardware programmability of an FPGA, enabling key analytics and hardware acceleration while integrating CPU, DSP, ASSP, and mixed signal functionality on a single device. Zynq-7000 AP SoC’s infuse customizable intelligence into today’s embedded systems to suit your unique application requirements” (from the website). Suddenly, the product exploded its available markets to include all of the traditional microprocessor applications. Wherever a little custom logic and some unique interfaces were needed, here was the solution. From wireless base stations, to automated driver assistance, the Zynq family with its ARM processors, and its Virtex FPGA fabric, was now the new product to beat.

At the same time, the 7 series as it is known, assigned new names to the device families: Virtex, Kintex, Artix, as well as Zynq. The names were meant to indicate the highest performance (Virtex), the best (lowest) cost and performance (Kintex), the lowest overall cost at lower performance (Artix), and the already named system on chip (Zynq). They all derive from the Virtex products, and that allowed a complete new version of the software tools (Vivado) which does not need to support the older devices, which enabled it to devote all of its capabilities to supporting the new families. Unburdened, the new tool provides an improved user experience, and even new ways to create the designs.

## References

Again, going beyond the datasheets, additional detail can be found in the Xilinx patent portfolio. Some key patents for this chapter include the following:

1. U.S. Patent # 5,942,913 by Steve Young, Trevor Bauer, Kamal Chaudhary and Sridhar Krishnamurthy. Although it is primarily concerned with interconnect, it shows in detail how many of the “pieces” connect together, which is a very big part of any design.
2. U.S. Patent #5,889,413 by Trevor Bauer, titled “Lookup Tables Which Double as Shift Registers”.
3. U.S. Patent #6,798,239 by Stephen Douglass, Steven Young, Nigel Herron, Mehul Vashi and Jane Sowards is titled “Programmable Gate Array Having Interconnecting Logic to Support Embedded Fixed Logic Circuitry”. This one simply addresses the issue of how best to introduce a foreign block into the regularly structured FPGA fabric, without undermining its effectiveness and disrupting its effectiveness and speed.
4. U.S. Patent #7,075,332 B1 by Steven Young, Venu Kondapali and Ramakrishnu Tanikella, is titled “Six Input Lookup Table and Associated Memory Control Circuitry for Use in a Field Programmable Gate Array.”

## Chapter 4 Xilinx Design Software

### Introduction

This chapter can never be complete. FPGA design software is extremely complex and shows no sign of becoming simpler. The goals keep changing. For instance, early solutions were considered “good enough” if the design fit the targeted FPGA. Moving forward, assuring it met user required timing for many constraints arrived as an important goal. Keeping the user pin-out constraints was also a key goal in the early days. Today, keeping the pin-outs through a myriad of late arriving edits is vital, but also offering the ability to manage the overall power consumption of the design across many different generations is a goal. Providing fast results (i.e., quick compile) as well as methods for groups of designers to share partial designs on the same project are more recent goals. The process never ends, but continually improves.

This chapter discusses some of the basic ideas that were only suggested in Chapter One, in greater detail. We can’t possibly cover everything, so we will discuss a few key basic principles, and jump very quickly to discussing the essence of more recent approaches. Many of the exact methods used remain trade secrets, but some of the techniques and approaches have made it into both the trade literature as well as the patent office. Clearly, those sources are available to provide greater details, and are specifically called out throughout the chapter, with a list of references at the chapter end.

### A Little History

Early Xilinx design software was primarily the XACT Place and Route package coupled with either ABEL or a schematic capture package from OrCAD or ViewLogic. Simulation was typically ViewSIMTM or PCSilos. Designers were trained in capturing designs using the Xilinx schematic symbol library, compiling, verifying with simulation, and downloading with a serial cable to their device. Many designers skipped simulation, preferring to test designs on the PCB. Some designers even skipped compiling, preferring to hand edit designs with FPGA Editor, the tool that permitted direct access to CLB contents and routing nets. Often, designers started compilations and went home for the evening, as their PCs crunched through the night. Partially routed results arrived by morning, for the designer to assess and manually edit, the next day. Sometimes, the compiles worked, needing no intervention. The first few years Xilinx design software operated in this way. Then, along came PREP (check the glossary). We will return to this story after discussing some more basic ideas.

### Some Basic Ideas

Chapter One outlined some of the requirements for the design software to deliver. Specifically, the design capture portion targets creating and delivering a netlist that describes the connections of various logic functions for the design. Verification of the design is usually done with simulation, where the burden of verifying is on the designer, who must create a sufficient set of input test stimuli (a.k.a. vectors) to “exercise” the model of the design that the simulator derives from the design netlist. This simulation should be done before compilation to assure correct functionality is captured – possibly

with estimated time delays. The more important simulation occurs after design compilation (place and route) with extracted delays back annotated into the netlist, representing actual delays encountered on the FPGA silicon. We only touch lightly on design capture, as our primary discussion focuses on compilation – function placement and routing. See Figure 4-1 for a simplified standard software flow.

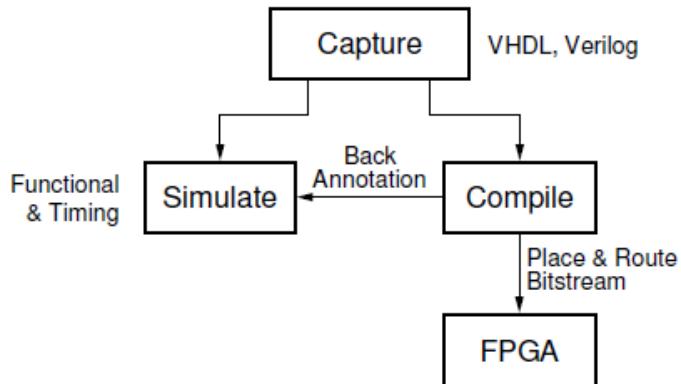


Figure 4-1 Basic FPGA Software Flow

### Synthesis

VHDL and Verilog are the standard methods of creating logic designs in most system designs done these days. Previously, schematic capture was the method, but this became cumbersome as designs expanded beyond the twenty to fifty thousand gate level. Compact text notation with Verilog and VHDL gave compact, flexible and portable designs that operate on a wide range of computers – PCs and workstations. Both VHDL and Verilog offer syntaxes with a very broad range of logic objects, in a parameterizable way. Buses, arithmetic structures, memories and other blocks can be compactly described, and expanded or contracted as needed within a design. The primary requirement was that vendors providing the tools use effective FPGA libraries so resulting netlists are compact and correct. This involves frequent interaction with Xilinx software personnel to assure the tools produce reasonable results. It is an ongoing effort. Because synthesis tools are so powerful and deliver widely varying results, depending on how the netlist is created, a couple of standard approaches have been agreed upon. First, ordinary logic structures are typically inferred by the synthesis tool. This means that the libraries are built so that the synthesizer identifies underlying logic structures and creates netlists that build up the functions from the LUT building blocks fitting well with Xilinx FPGA devices. Second, special blocks are better created by user instantiation. This means that building high speed transceivers can be done with LUT fabric, or with multi-gigabit transceivers (MGT). If you want the MGT, it is best stated explicitly in the code.

Memory and multipliers are two choices where the synthesizer can either assign connections to block resources, or infer results built from LUT fabric. It is the user's responsibility to alter the design to influence the synthesizer's choice. Xilinx applications personnel have created many example design templates showing exactly how to best build those results where instantiation is critical, and offer them by way of application notes and reference designs. Netlist reduction occurs after synthesis, followed by the compilation phase.

## Compilation

### Technology Mapping

The netlist needs to be provided to the compiler after technology mapping. This means appropriate substitutions for the target architecture must be made, to achieve decent reduction in technology cells for the design. That can be done by the synthesis tool, by the backend compiler, or both. Chapter One discussed some of the tactics used, when identifying sections of circuits that efficiently map into the four input LUTs. Nothing says that it is optimal at any point, and may be revisited throughout the flow, as long as the transformed circuit is logically equivalent to the original intention, and faithfully meets the timing requirements of the user. Some choices made by the technology mapper and other modules are not intuitive, and may rely on heuristic methods, analytic methods, or both. Understanding some of the metrics being used by software is a good thing.

### Metrics

Compiling a design to produce a bitstream consists of a lot of steps. Placement is one of many, but a critical one. Before addressing some of the methods and their rationales, it is important to remember that software often works best when driven to meet a goal. To that end, a spectrum of design metrics can be identified and chosen to direct reasonable results. For instance, it is generally possible to place and route a design that is only 50% occupied with the fastest speed grade part. However, that is rarely what users seek, from an economic view. The compiler would generally be successful under those conditions. It is important to agree on basic constraints at the outset, so the compile tools can adjust their strategies accordingly.

Let's consider a case many logic designers may have encountered in previous early logic design classes - Multiple Function “Quine-McCluskey” minimization. In logic reduction methods, an initial cost function is created with a number of gates and inputs. The reduction goal is to select a minimum logic representation collapsing the number of total gates, with the fewest total inputs. FPGA placement goals are typically along the lines of minimizing total wire length of all connections, or total area of assigned functions. Sometimes, the methods are counter intuitive, but choosing to use them is based on how effective they are (i.e., producing good results).

Now, let's consider an example that uses a physical model to convey the idea of a “good” placement metric. Figure 4-2 shows an FPGA grid with some logic blocks identified and connected. A metric that could help evaluate the quality of this arrangement might be the total wire length needed to connect the four cells shown. To calculate that, the design software might add up the sum of all connections in the X direction and all signals in the Y direction, and arrive at a “cost” for this arrangement. This would be the metric to be reduced when evaluating successive arrangements. Two things to note – first, we don't know the real cost for connection because we can't anticipate the congestion, so we work with an assumed wire length cost, which is an estimate. Second, calculating wire length distances in the X and Y directions along a Cartesian grid is called the Manhattan distance (i.e., you move around Manhattan staying on the streets or sidewalks). We will discuss more metrics as we proceed to

look at various approaches.

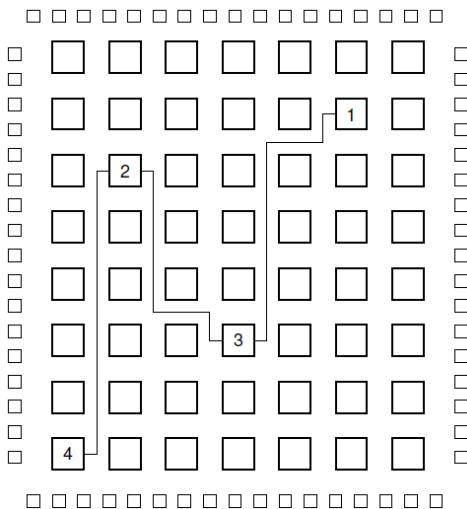


Figure 4-2 Example of Four Cells Placed and Connected

### Placement

Placement is basically finding a best cell arrangement, according to satisfying some metric. A good arrangement meets the user constraints of pinouts and timing, for sure. A better arrangement would tolerate some design editing without totally corrupting the timing of the whole design. There are many approaches. One placement method is simply trying arrangements – randomly – and evaluating them. When the software has done plenty of them – say a thousand - quit and take the best one. It is possible that it never arrives at a good arrangement, but for that thousand trials, the user knows it got the best of them. Clearly, this is simplistic. There must be a better way to approach the problem systematically.

Another approach we might take is to model something in nature, sensitive to positional arrangement. Our intuition is guided by natural principles that result in small, tight arrangements, and we can find such a model in the world of mechanics. For instance, if we know that two cells must connect, then they have a need to communicate, according to the design. In that sense, they exhibit an affinity for each other. One such physical item that exhibits a geometrical affinity is a spring. A spring prefers to be in its un-stretched, rest state. A system of springs would reside in their least stressed state, if all springs were not stretched. The total energy of a set of stretched springs can be calculated by adding up all of the individual stretch distances according to the energy expression:  $\frac{1}{2} K X^2$ . (Thank you, Robert Hooke!) Let's see how this would be applied to the above problem, with Figure 4-3.

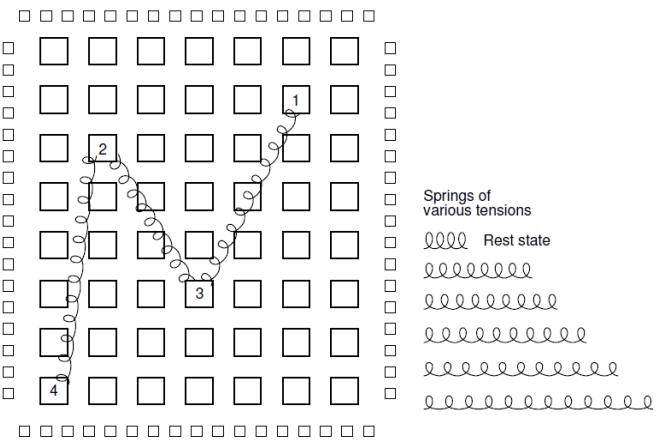


Figure 4-3 Cell Layout with spring “affinities”

The springs in Figure 4-3 could actually be resolved into a vector “force diagram” telling you which way to move the placed cells, to arrive at a lower net “energy” (i.e., more relaxed springs) for this placement. Although not stated earlier, let’s assume that cells 1 and 4 are placed near an edge to satisfy a pinout requirement. We are free to move cells 2 and 3 to arrive at a better solution. Figure 4-4 shows a better placement, with this energy metric. Shortly, we will see that more complete metrics are desirable, having additional dimensions that help identify a better solution to a more appropriate cost function.

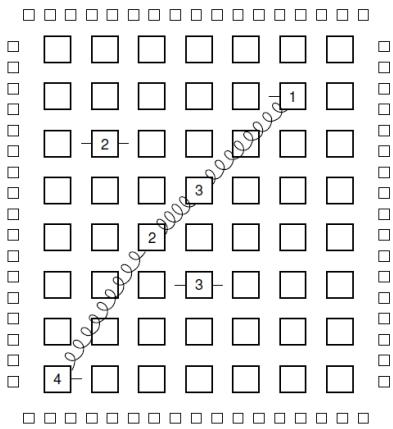


Figure 4-4 Improved Energy Placement (New Positions in Bold)

With many such arrangements to evaluate, the problem can grow very quickly and take substantial time to arrive at a reasonable placement of cells to give a satisfactory solution. In theory, every time a modification of a single cell is made – a move, its impact on the overall system energy needs to be evaluated in order to determine whether or not to accept that move.

There are many algorithms for FPGA placement, so let’s look at another, more famous one known as simulated annealing.

### Simulated Annealing

Simulated annealing was originally developed during World War II by Nicholas Metropolis and others, while solving nonlinear optimization problems during nuclear weapon development. It has remarkably robust properties and nearly always results in a superior placement solution. However, nothing says it is guaranteed to converge in finite time. Nonetheless, it frequently of system provides good partial results that can be made to improve, when suitably modified. It also uses the concept energy, as a metric, so our earlier “spring” example wasn’t a total waste.

Metropolis observed that metallurgy successfully created hard crystalline structures by modifying the system temperature. Shaped hardened steel can be created by repetitively heating and cooling the metal, while shaping it when it is hot. Successive cooling locks in the last shape, which is made when the metal is hot. This is called quenching. At the molecular level, hot atoms are active and more “free” to move around than cold ones. Cold ones tend towards a low energy state, and lock into a crystalline structure when the heat is removed. Adding heat, permits moving atoms from the last locked position, to new positions. So, what does this have to do with FPGA placement? Plenty.

Simulated annealing gives us a method, by modeling natural processes, to arrive at a similar “shape” of cells within our regular array of cells. We can create an analog to temperature, by simply assigning mobility properties to the various placed cells on the FPGA. As we proceed to arrive at improved placements, we can do so by evaluating the new placement, according to system energy and decide whether or not to accept the new placement, or not.

This sounds like it may have merit, but why might it be a preference to assigning little springs to connections between the cells and evaluating it like springs? A key property of simulated annealing is that it can extricate itself from a local minima. Simulated annealing permits occasionally accepting a placement that is not less than the previous placement, but is actually greater. In so doing, it adds the possibility to ultimately find the global minimum for system energy, versus falling into an artificial trap of accepting a local one, and staying there. Figure 4-5 shows what is meant by local minima, where multiple “troughs” exist that might cause an algorithm to stall. The axes on Figure 4-5 are cost versus configurations. As shown, the “current configuration” is headed for a local minima, which might represent an arrangement where logical energy reductions would cause the arrangement to remain there. Annealing permits accepting a placement that arrives at the other side of the “hill”, and finds a subsequent lower cost. Let’s look a little deeper.

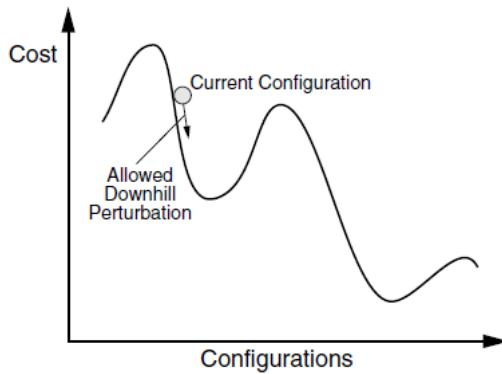


Figure 4-5 Cost versus Configurations – Local Minima

Here is verbally how a simulated annealing placer works. A simulated annealing based placer begins with a random placement of logic blocks and I/O objects. Following this, the random placement is improved iteratively, by choosing pairs of logic blocks or I/Os to swap. The goodness of each swap is evaluated with a cost function. The cost function used is typically designed to minimize estimated wire-length and timing cost. Swaps that decrease placement cost are always accepted. However, swaps that increase cost may or may not be accepted, depending on probabilities. By accepting some swaps that increase cost, the algorithm permits a limited amount of hill climbing which gives an opportunity to free itself from local minima. This process can be stated algorithmically, as follows:

```
M = # of moves to attempt;
T = current value of temperature;
for m = 1 to M{
```

```
    Generate a random cell move;
    Evaluate the change in system energy, "delta E";
    If (delta E < 0){
        /* it's downhill move, accept it */
        accept this move, and update the configuration
    }
    else {
        /* uphill move, maybe accept it */
        accept with probability P = e-delta E/T;
        update configuration if accepted;
    }
}/*end for loop*/
```

The process requires initializing to a first “random placement”, limiting the number of moves, starting temperature, etc. But, it is not conceptually hard, and gives good results. There are many ways to modify the approach and maintain the integrity of the underlying algorithm, also. Simulated annealing asks us to take the counter intuitive step of possibly accepting a placement which is probabilistic, versus simply rejecting it. It makes us consider very carefully what the overall goals of the process are, to identify other choices and options that may yield superior results.

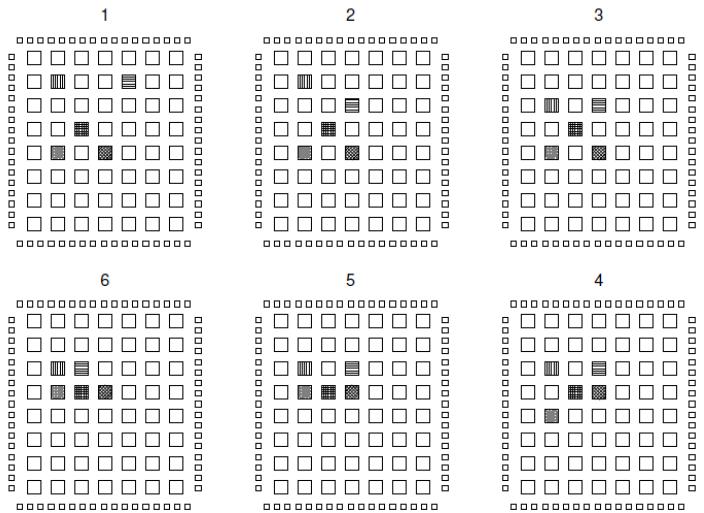


Figure 4-6 Five Simulated Annealing Cell “Swaps”

Figure 4-6 shows five successive versions of a set of placed cells. The first arrangement, labeled “1” is the random initial placement. Number two represents the situation after swapping the upper rightmost cell, with a blank to its lower left. I am using the “spring” process of lower energy, to guide the choices. The third figure swaps the upper left cell down, by exchanging it with a blank that is nearer to the “geometric center” of the cluster. Here, we see a balanced looking set of five cells. The fourth figure swaps the lower right cell with an adjacent blank that is above it. The fifth figure exchanges the lower left cell with a blank above it, resulting in cluster shown. This might be a valid stopping point, as the cells are pretty much as close to each other, as can be permitted. The sixth figure may or may not have merit, exchanging the blank in the top row of the cluster, with the cell in the top right of the cluster. We arbitrarily choose to stop here, as we envision no particularly better arrangement than this one.

### Routing

Now that we have mentioned some of the placement basics, let’s shift the discussion to connection, or routing. Like placement had its roots in history, so does routing. The path that is taken to arrive at a destination is a classic problem of operations research (aka, systems engineering). The traveling salesman problem, or the delivery truck problem of identifying the best sequence of stops to make along a route, to minimize path length has been considered for a long time. It is considered to be solved. Mathematically, it is the dual of the assignment problem, which is how to best assign N tasks to N employees. None of these has the charm of the problem statement originally postulated by Claude Shannon in the first half of the twentieth century, while working for the telephone company, at Bell Labs.

Shannon used chessboards and mazes to model network switching in the telephone system. He arrived at the problem stated as a rat negotiating a maze, seeking a goal of “cheese.” Shannon basically identified key elements: a starting point, a goal and a set of obstacles (ie, the maze partitions). Shannon ending up solving the problem by building the intelligence into the maze with a set of magnetic switches and relays, which is similar to how the switching systems in modern telephone networks have evolved.

The mathematical thinking developed to solve that problem was the starting point for the routing problem. It also influenced subsequent terminology.

Early routers were software modules that started with the placed arrangement of logic cells. Routing proceeded by connecting an input cell to its target, then going from there, to the next target, etc. As connecting proceeded, all nets would either become connected, or it would manifest that certain paths could not be connected. Largely, this was due to the possible path choices being taken by previous connections. The failure condition was termed congestion, and often appeared to visually look like a rat's nest. Even developers of today's modern routers sometimes refer to making the connection as "snatching the cheese." The early, simple routers are also called maze routers. There are many variations on routers, for instance, there are global routers and detail routers, depending on the scope of the signals being considered. Global routers are concerned with identifying general requirements for the path of a signal, like "make a vertical connection, followed by a horizontal connection". The global router doesn't dictate exactly which vertical or horizontal connection to make, which is the job of the detail router.

## **Back to Our Story**

The above diversion was designed to set the stage for the events following the PREP benchmark contest in the early 1990s. The situation had about a dozen programmable logic silicon vendors forming an organization (PREP) to create design benchmarks with appropriate judging criteria. The general goal was to give the purchasing public a rational basis for making purchasing decisions. Part of the ground rules included permitting software created by the silicon vendor, as well as third party CAD providers to be used. Xilinx and its chief rival, Altera were running in tight competition. A third party CAD company named NeoCAD applied its "Foundry" design tools to the Xilinx products and consistently delivered superior performance.

Xilinx had to take them seriously, to evaluate the secret formula they brought to the contest. In the end, Xilinx took the "gentlemen's way out" and bought NeoCAD. That resulted in positive transformation of Xilinx design software, which would set the stage for future releases. The earliest, XACT software was known internal to Xilinx as Automatic Place and Route, APR. The combined NeoCAD software with Xilinx software was called XACTStep, but internally was called Partition, Place and Route, PPR. Adding timing into the mix, the next software was externally called ISE, with the internal name being Place and Route, PAR. Moving the ISE framework forward, it is internally called Pack, Place and Physical synthesis, P3 or P cubed. Each release has added some improved nuance, for overall results, or the user's experience. The arrival of Virtex changed the software field considerably. In particular, eight possible banked voltage standards brought one set of requirements. Offering pre-packaged soft intellectual property, added more requirements. Inserting hard blocks, such as BRAMs, DSPs, MGTs all added even more requirements. Our story continues with the new capabilities. Let's take a look at some of them.

## **Virtex Design Software Developments**

Depending on the family chosen, different approaches are taken to produce good

results for users. The early parts (Virtex, E, EM) primarily introduced multiple banked voltage standards, clock management (DLL) and BRAM. The big hurdle with this family was managing the I/O signals to avoid conflicts with the banking requirements. We will look at the various strategies and see it is really a series of evolutionary improvements, over time.

## Handling Multiple I/O Standards

Virtex, E and EM all arrived very quickly, with each having eight different I/O bank possibilities. The I/O bank requirements include single ended voltage standards, double ended standards and standards requiring application of an external reference voltage (VREF) against which a switching input voltage is compared. Typically, a given voltage standard “prevails” for all of the I/O pins within a particular bank. VREF is also an input, to the I/O bank, occupying a pin within that bank. Output voltage swing is dictated by an external bank attachment to a VCCO pin, so a bank may be affected by a VCCO which determines all output voltage swing levels for the bank, as well as VREF, which is applied to those standards within the bank that require a VREF. Some open drain standards (i.e., GTL) have their VCCO determined externally to the chip, by a pullup to an externally applied voltage. Table 4-1 shows an example of three voltages and their requirements.

IO Standard	Direction	V <sub>REF</sub> Required	V <sub>CCO</sub> Required
Peripheral	Input	No	3.3V
	Output	No	3.3V
Interconnect	Bidirectional	No	3.3V
Gunning	Input	0.8V	No
	Output	No	No
	Bidirectional	0.8V	No
Transceiver	Input	0.75V	No
	Output	No	1.5V
	Bidirectional	0.75V	1.5V
Logic			
High-speed			
Transceiver			
Logic -I			

Table 4-1 Some I/O Standard Voltage Requirements

Reference 7 details the methods chosen to support the handling of the I/O standards, so we will summarize them here, and encourage the reader to investigate further with that paper.

The algorithm chosen is unique in that it combines simulated annealing, weighted bipartite matching and bin packing heuristics. Heuristic methods have become commonplace, with FPGA software, as they improve over time as additional knowledge is gained about the chosen approach. Weighted bipartite matching is a form of the classic assignment problem – i.e., assigning a set of tasks to a set of resources, depending on which is best suited for the task. Weights are set or determined for matching the task to the resource. Bin packing is a systematic process of taking the next item to be loaded into a particular position, and placing them in order of

consideration. Both can be done simply.

Going further, the simulated annealing step places a user's core logic and I/O objects using a cost function with a component that is directed at removing I/O banking rule violations. Following simulated annealing, I/O placement is improved using the weighted bipartite placement algorithm. If annealing and bipartite matching fail to produce a feasible I/O placement, the algorithm enters a constructive packing step where I/O objects are packed within banks in a feasible way. Experiments have verified that the algorithm works effectively and reduces overall wire-length. The process begins with simulated annealing, with logic and I/Os placed randomly. The goodness of the swaps is evaluated by a cost function designed to minimize estimated wire-length and timing cost. Swaps that decrease the cost are always taken, but swaps that increase it are only accepted according to a probability curve, as mentioned earlier. Note that the cost is based on estimated wire-length, because it has not been routed yet. We do not know what the real wire-length is until after routing time. Here is the form of the placement cost function:

$$\text{PlacementCost} = a \text{ WirelengthCost} + b \text{ TimingCost} + g \text{ BankViolationCost}$$

Parameters  $a$ ,  $b$ ,  $g$  are scalar constraints reflecting that term's importance. High values can override that factor in the placement cost, and dominate. The Wirelength cost is based on estimated wire-length. TimingCost is based on user supplied constraints and timing slack allocation. User constraints come from design requirements, and timing slack allocation comes from experience. BankViolationCost is comprised of multiple factors. Each bank will have a tally of conflict costs based on that bank's VREF and VCCO requirements. Signals that use the VREF for a bank have low cost, as well as signals that require no VREF, as both can be accommodated within the same bank. Signals with a different VCCO requirement cannot be accommodated. This needs the idea of a prevailing VREF and prevailing VCCO, which are the values chosen to handle the majority of signals assigned to a bank, or a user constraint for the bank. A simple assignment of zero cost to signals that fit the mold of a bank, and infinite cost for signals that violate the prevailing choices works to provide simplified choices, to influence the direction of the overall cost function. There are more complicated issues, but that generally explains the basic concepts.

## Timing Driven PAR

Reference 4 and 7 describe approaches to constrain place and route to achieve specific timing goals. Reference 4 introduces the concept of timing slack, and an overall timing budget. Slack is simply the difference between the required timing for a net, and the actual timing of it. Slack can be positive, negative, or zero. Zero slack meets the timing requirement.

Positive slack exceeds the requirement, and negative slack simply means that the actual time delay for the specific net exceeds the required time delay. A number of methods exist to budget slack, so that it can be spread around or redistributed among the delay components of a net. Reference 4 focuses on defining a set of slack limits for the network then provides a recipe for "bumping" the limits to adjust them to suit a given overall requirement. Reference 7 outlines a method to perform simultaneous

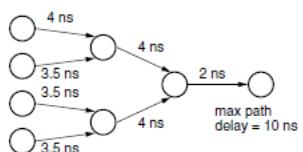
placement and routing. It has been evaluated for multiple architecture types – row based FPGA devices as well as the Virtex “Island” style of FPGA. In this sense, think of clusters of LUTs and flip flops (CLBs or slices) as tiny function islands surrounded by a sea of connection fabric. The approach identifies 6 key ideas affecting results:

1. Cells and nets are uniformly malleable. It is possible to rearrange functions within cells, cell positions and assigned connect choices as needed throughout the place and route process.
2. Simulated annealing is the chosen placement optimization method. It permits modification of the tuning criteria, and is good, in general.
3. Routing occurs during placement. This way, actual connections are evaluated for the current placement. Ripping up and re-routing is permitted.
4. It is not required to route all intermediate placements. It is possible to obtain some partial layouts with complete routing for part of it, and incomplete routing for other sections.
5. Incremental rerouting is used as an attempt to recover from layout perturbations due to replacement, with the target being to reroute previous un-routed nets.
6. Focus on relay-out is based explicitly on critical path improvement with successive perturbations.

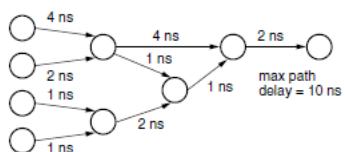
The cost function chosen is: Cost = Wr X R + Wt X T R counts the number of nets lacking complete routing and T measures the worst-case delay on the slowest path in the current placement. Wr and Wt are weights adaptively determined at runtime to balance the two cost components. Details of the rerouting and updating strategy are discussed in Reference 7.

### Run-Time-Conscious Automatic Timing-Driven Layout Synthesis

An effective method used in the Virtex-II family is described in Reference 12. It builds upon earlier ideas, but adds several new ones. It recognizes a discovery, that increased runtime of the layout process is a strong function of the number of constraints being satisfied. Rather than attempt to satisfy a lot of constraints, it attempts to simply make the whole design faster, where faster specifically means operation frequency. By making everything faster, the strategy works with a reduced set of constraints, simplifying and speeding up the process.



*\*Critical\** Example Circuit



*\*Less Critical\** Example Circuit

Figure 4-7 Timing Graphs of Networks

Remember that slack is the difference between required time delay and actual time delay. Zero or positive meets the need, but negative slack for a signal makes it critical. Figure 4-7 shows two groups of signals. These are directed acyclic graphs (DAG), representing signal paths and associated delays. Interconnect delays are shown along the arrowed paths, and logic block delays are represented by the circles. Block delays are assumed to be zero in this example. The top DAG has four possible paths from a vertex (ball) on the left, called sources to a ball on the right, called a sink. Tallying the incremental delays along the four paths, it is seen that the top path adds  $4 + 4 + 2 = 10$  nanoseconds. The three other paths each tally to 9.5 nanoseconds. If we simply say that we wish to reduce the time delay to 90% of the maximum, then all paths would have to be reduced to no more than 9 nanoseconds. This simplistic approach makes all paths for that net critical, as they are all between 9 and 10 nanoseconds. Every one of them exceeds the constraint target. Examining the lower network, there are six paths from the sources to the sink. Again, the maximum path delay shown is 10 nanoseconds, but five other paths are less than 9 nanoseconds, so applying the “90% of the maximum” criteria to 7b results in constraining only one of the paths. This is good, as it sets up a condition of only one critical path to be managed, versus four in the previous example.

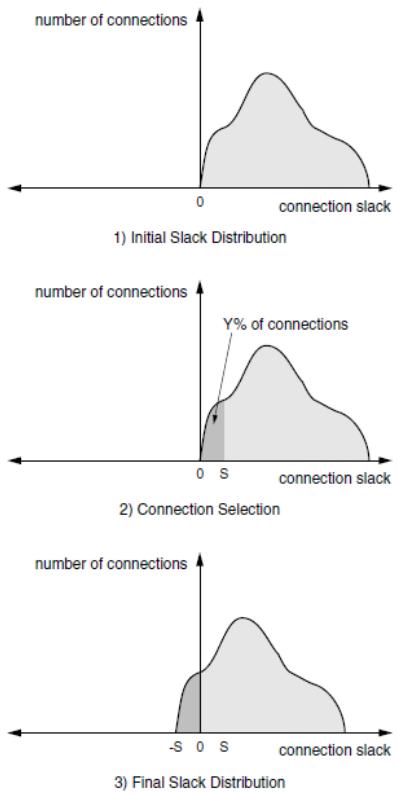


Figure 4-8 Slack Histogram Distributions

Examining Figure 4-8, we see a distribution of slacks for various nets in the design. The top graph represents an initially placed and routed design distribution. If we take Y% of the distribution, and assign it to have negative slack, that effectively shifts the distribution to the left. The bottom curve shows the distribution so shifted. The result is that we identify

a small set of signals that will need to be carefully routed, to improve their slack, but a larger set remains with acceptable slack. The idea is to use this small set of signals to improve the overall time delay of all of them.

Here's how the process works. At placement, connection delay estimates are used (i.e., annealing). At routing, accurate delays are now available. From this layout, we extract the maximum time delay of any path, called  $T_{layout}$ . This corresponds to the initial layout distribution in Figure 4-8, and has no negative slack (yet). Analysis of this first distribution is to identify a slack value,  $S$ , so that  $Y\%$  of the parameters have slack less than or equal to  $S$ . This defines a constraint such that:  $T_{constraint} = T_{layout} - S$

$T_{constraint}$  is the target to meet with the next layout. This produces a new slack distribution, as shown in the bottom curve of Figure 4-8. The approach gives fast, satisfactory results, when typically applied three times. The method relies on a single parameter,  $Y$  to define the constraints for the whole design. Good results have been obtained by using  $Y$  repeatedly at 2% for no more than three passes through a design. A key quality of this method is simply that it provides two important qualities. It's fast in runtime, and very reasonable in performance. Performance focused routing can beat the results given by the Runtime Conscious method, but it comes at substantially greater runtime, than this one takes, and may not be dramatically faster.

## RCT – Resource, Congestion and Timing

The Virtex 4 ASMBL architecture introduced some architecture changes that suggested additional methods might be used for effective place and route. RCT is the framework that evolved to fit those needs. The RCT methodology is currently being successfully used. Its roots can be found in reference 5, on PathFinder by McCurchie and Ebling. A key idea here is that the router should attempt to first optimize every connection required, by simply making the connection. This should give the fastest connection evaluation for all paths, for a given placement. After the paths have been connected, the design is analyzed to identify connection conflicts (read: short circuits). When conflicts are identified, a negotiation process is engaged, where cost is evaluated for all contenders in the conflict. The negotiation is designed to favor the net that requires the connection to make its timing budget. This process is termed overlap removal. Overlap removal can be viewed in general as a priority based negotiation process.

Although substantially more complex than this example, Figure 4-9 shows four different connection possibilities to connect points A-C, B-E, F-D. A distance grid is superimposed to model "time" as a net length, where the upper left, starting case shows a net time delay for the three nets to be a cost of 16 time units. The top right net shows connections being made at a cost of 12 time units, which is 25% faster, but results in a short being introduced, that would need negotiating and typically increase that number. The lower left case shows another result that costs 16, which is no better than the original. The lower right shows the chosen connection that results in a cost of 12, and has no shorts. It is the chosen solution.

Figure 4-10, shows how it might have been discovered more directly, by first just making the connections, which incidentally incurs a cost of 8. Identifying that the B-E net is "responsible" for two shorts, suggests that it be "negotiated" around the other two vertical nets, giving the chosen net shown in the lower right of Figure 4-9.

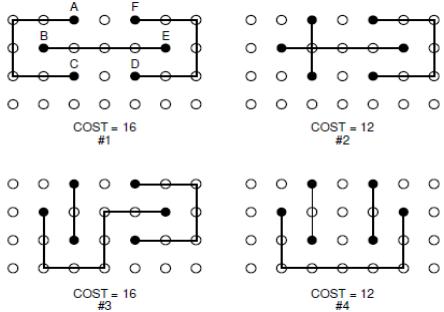


Figure 4-9 Different Routing Strategies

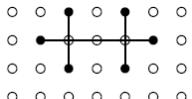


Figure 4-10 Direct Routing, with Shorts

We have only described a few signals interacting, but in real designs, there will be many complex “tangles” that are encountered, taking much greater effort to identify the proper signal negotiations, and deliver the best resolution - but in the end, it works!

### More Complete Software Flow

Figure 4-11 better details some of the steps that we have discussed to this point, than is shown in Figure 4-1. What it still fails to show are the misfires, and attempts that require ripping up and re-routing, which are applicable to many of the design flows discussed, so far. The general goal is to make the software be largely architecture independent, so the benefits of new methods will apply to older architectures and improve them, also. Small things are important. For instance, the RCT method requires a single parameter to drive the layout process, and does not need communication of various calculations across software module boundaries. Simplicity breeds reliability, in generally complicated software. Methods that embody general principals are valuable, and the thinking process on placement and routing has come a long way in the last twenty years, but there is still distance to go. It is not perfect, yet.

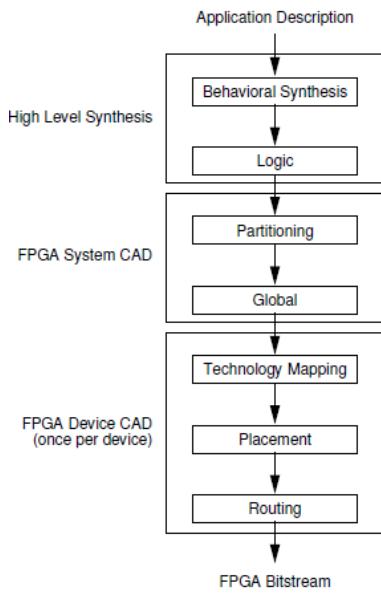


Figure 4-11 More Detailed FPGA Design Software Flow

## New Directions and Moving Forward

Large complex designs formed on large FPGA devices are exposed lots of issues with the general approach used with design software. The ability to create simultaneous switching events with hundreds of signals is possible, and very tough to debug, when on a printed circuit board. Introduction of tiny edits into large designs result in many days of tuning and trimming the PAR effort, to get back to the performance that was already there before the edit.

These types of problems are best not handled after the design is deemed done, but rather, in advance. Planning for predictable issues can be done, and this is a new direction for design software. Identifying issues that occur when integrating multiple blocks onto a single chip, is important – particularly when each block was independently designed by different designers. Using commercially available intellectual property, without it disrupting the rest of the design, or alternately, having its integrity disrupted is just as important.

These and many other issues are here today, and advanced planning is the primary strategy. One example of such a tool is the PlanAhead product originally developed by Hier Design. PlanAhead is an ASIC style floorplanning tool that lets you place and route key function blocks to achieve target performance and lock it in. PlanAhead lets you create, constrain and connect blocks once, and use the results repeatedly. Once designed, the blocks persist with their tuned performance, and drop right in to the PAR flow, saving hours of replacing and rerouting, as the blocks are stitched into the design. Figure 4-12 shows a “before” and “after” floorplan with blocks connected using PlanAhead tools.

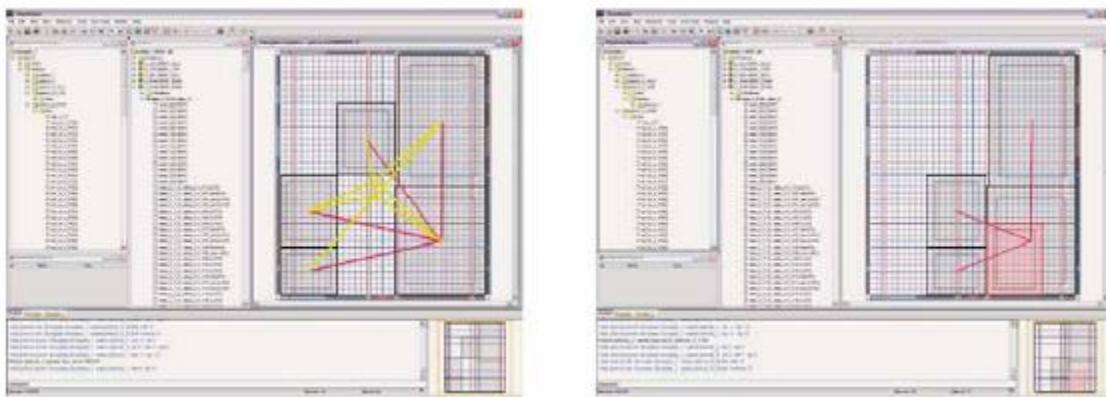


Figure 4-12 Before (left) and After (right) PAR with PlanAhead

We haven't discussed technology mapping much at this point, but it is a block shown in Figure 4-10. It's another area for improvement, where function mapping can be modified at several points in the design flow, so look for more work to be done there, in the future. Lots of opportunity remains for identifying points for additional flexibility and creativity, to offer improved solutions.

Xilinx created a flow from the ground up for the 28nm technology node. The tools had grown, and were inefficient in the new era of multicore processor machines. The result is Vivado, which also supports high level synthesis with the ability to convert c/c++ code into gates (using Verilog as the intermediate step). Vivado does not support any earlier devices. Vivado uses a new synthesis tool, a new analytical places which provides a deterministic solution without any annealing. All of the new modules make use of multicore processors to the extent possible. With each new update (issued quarterly) one sees an improving trend as more and more cores may be used to speed up each step.

### **Afternote**

PREP had both good and bad results. In general, it showed all the players attempting to play by the rules. However, it was very interesting after the dust settled, to find each candidate company literally declaring itself to be the "winner" (and PREP proves it!) by simply identifying an appropriate framework to present their results. Some claimed to be fastest, most economical, most efficient, and so forth. Others had to stretch the spirit and declare results like "most consistent", or "least deviation among results". This just goes to show that you can never predict the outcome of a marketing focused contest. Or, maybe ... you can.

### **References**

1. Simulated Annealing Algorithms: An Overview, Rob A. Rutenbar, IEEE Circuits and Devices Magazine, pp 20-26, January 1989
2. The Timber Wolf Placement and Routing Package, Carl Sechen and Alberto Sangiovanni-Vincentelli, IEEE Journal of Solid State Circuits, pp 510-522, Vol. SC 20, No. 2, April 1985
2. A Detailed Router for Field-Programmable Gate Arrays, Stephen Brown, Jonathan Rose, Zvonko Vranesic, IEEE Transactions on Computer Aided Design, Vol. 11, No. 5, May 1992
3. Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing, Jon Frankle, Design Automation Conference, Proceedings, 29th ACM/IEEE, pp 536-542, 12 June 1992

4. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs, Larry McMurchie and Carl Ebeling, Third International ACM Symposium on Field-Programmable Gate Arrays (FPGA'05), pp 111-117, Monterey, California, USA, 1995
5. Patent No. US 5,659,484, Frequency Driven Layout and Method for Field Programmable Gate Arrays, David Wayne Bennett, Eric Ford Dellinger, Walter A. Manaker, Jr., Carl M. Stern, William R. Troxel, Jay Thomas Young, August 19, 1997
6. Performance-Driven Simultaneous Placement and Routing for FPGA's, Sudip Nag, Rob Rutenbar, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 17, No. 6, June 1998.
7. A Placement Algorithm for FPGA Designs with Multiple I/O Standards, Jason Anderson, Jim Saunders, Sudip Nag, Chari Madabhushi, Rajeev Jayaraman, FPL 2000, LNCS 1896, pp 211-230, copyright Springer-Verlag Berlin Heidelberg, 2000.
8. Patent No. US 6,289,496 B1, Placement of Input-Output Design Objects into a Programmable Gate Array Supporting Multiple Voltage Standards, Jason H. Anderson, James L. Saunders, Madabhushi V.R. Chari, Sudip K. Nag, Rajeev Jayaraman, September 11, 2001.
9. Patent No. US 6,484,298 B1, Method and Apparatus for Automatic Timing Driven Implementation of a Circuit Design, Sudip K. Nag, Kamal Chaudhary, James H. Anderson, Madabhushi V.R. Chari, Sandor S. Kalman, November 19, 2002
10. Run-Time-Conscious Automatic Timing-Driven FPGA Layout Synthesis, Jason Anderson, Sudip Nag, Kamal Chaudhary, Sandor Kalman, Chari Madabhushi, Paul Cheng, pp168-178, FPL 2004, LNCS 3202, copyright Springer-Verlag, Berlin, Heidelberg, 2004
11. Patent No. US 6,851,101 B1, Method for Computing and Using Future Costing Data in Signal Routing, Raymond Kong, James H. Anderson, Feb. 1, 2005

# Chapter 5 Configuration, Reconfiguration and Security

*“Done didn’t go high. What should I do?”*

*-frequently asked question (FAQ) on the Xilinx forum*

## Overview

Delivering the design, in the form of a bitstream to the FPGA presents a set of interesting problems. Some people need the FPGA to configure rapidly, and others can afford to take more time. Some people wish to deliver from existing memory resources and others prefer dedicated EPROMs for FPGA configuration. Some systems contain a single FPGA and others have multiple FPGA devices feeding from a common data stream. As well, some designers may need some aspect of their configuration to occur quickly, while others can be done slowly. On top of this, some users recognize the amount of work involved in creating a design, and wish to secure it by encrypting the bitstream to prevent theft. Creating a single facility for all these designers is not trivial.

This chapter outlines the basics of configuration, and reveals some of the inner workings that are not evident to many users. With this knowledge, additional capabilities to use for designs where reconfiguration is fundamental to the overall solution will be better understood. Finally, the underlying mechanisms used for decrypting bitstreams is discussed.

## Configuration Basics

At the end of place and route, all functions, positions and connections are known. Creation of the bitstream for an FPGA is undertaken by a software module called BitGen. The created file is either downloaded through a cable or programmed into an EPROM for delivery to the FPGA. The programming software can combine bitstreams into a composite if needed for multiple FPGA devices. We discuss more on that later. The configuration bitstream can be viewed as an alternate image of an FPGA. Underneath the logical structure, is a separate bit framework forming the connections, filling the LUTs and BRAMS, as well as various registers within multi-mode clock management tiles (MMCM in the newer devices replaces the DCM), DCMs, multipliers, MGTs and whatever future features evolve.

Figure 5-1 reveals the top, logic layer shown above a lower configuration image of an FPGA. Both images reside on the silicon, side by side, not as shown in Figure 5-1, but it is still a good working model for discussion. This simplified diagram shows the top section containing the IOBs, CLBs, BRAMS, etc. The bottom portion is simplified and shows a JTAG port (lower left), various configuration cells (little squares sprinkled around) and a central rectangle, called “CONFIGURATION CKT”.

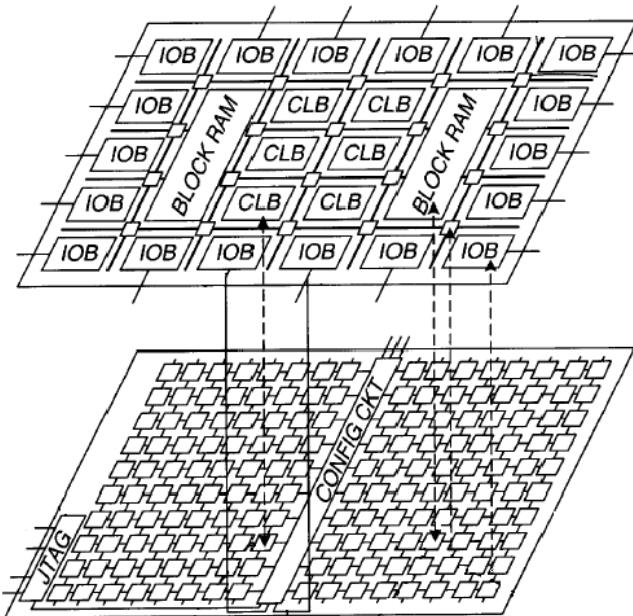


Figure 5-1 Relationship of Configuration Memory to Logic Structure

The configuration circuitry receives the bitstream from the outside world (from a configuration cable, an EPROM or microprocessor), and disperses the bits in an orderly way to the various configuration storage cells. Clearly, there must be a previously agreed upon protocol, and because bitstreams can be very large, there must be recognition that errors may occur in data transmission into the FPGA. All this is handled by the configuration control circuitry. Figure 5-2 gives additional detail more clearly showing the bitstream landing in a central shift register, targeting the central FPGA column. This shifter is of a length called a frame, which will be the fundamental data item transferred. The input data stream is subdivided into consecutive frames. We shall describe the sequence of events in greater detail, shortly, but Figure 5-2 shows that the frame shifter (central rectangle) captures a frame, which is connected in parallel to all cells – both left and right, down the rows. The frame shifter outputs connect to the parallel D inputs of the various latches in the rows. Hence, if a frame value is loaded, all that is needed to transfer it to a column of configuration cells is a clock, attached to each column independently. The configuration process is simply extracting frames of data from the incoming bitstream, identifying the correct columns to receive the frames, dispatch them, and move on until completion.

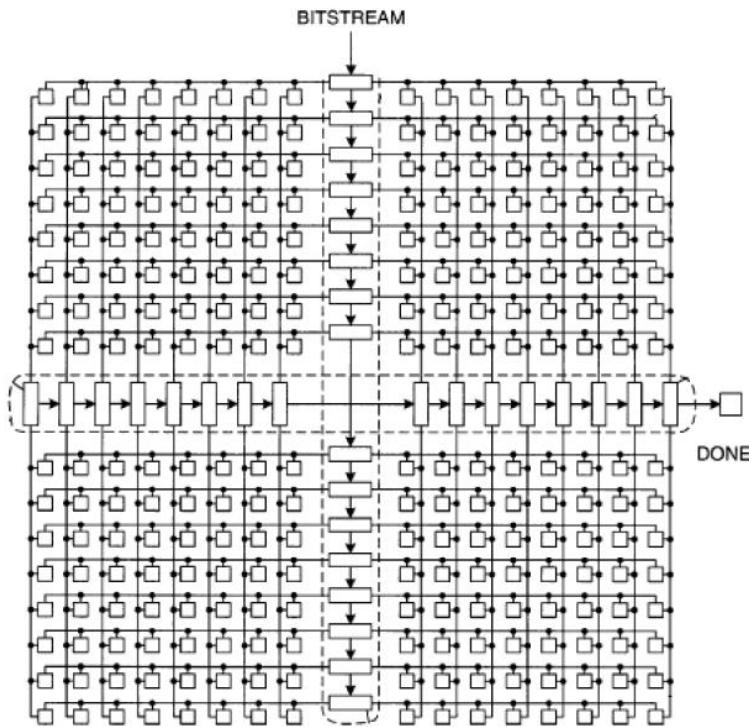


Figure 5-2 Frame Shift Register Relation to Configuration Cells

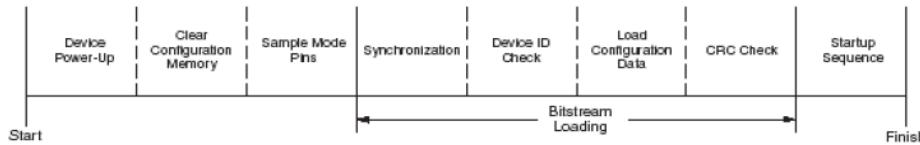


Figure 5-3 Sequence of Actions for Configuration

The operation sequence for typical configuration are summarized with the following action list: (reference Figure 5-3)

1. Device power up
2. Clear the configuration memory
3. Sample the mode pins (described shortly)
4. Synchronize
5. Check device ID
6. Load the configuration data
7. Check the CRC
8. Issue startup sequence

Let's describe the actions in more detail. At power up, the various configuration cells and state machines are not capable of correct electronic action until sufficient voltage and current are available from external power regulators. Depending on the Virtex family, the precise voltages will be different for the various activities, as the silicon processes differ. In general, there is a recognition voltage where internal state machines trigger and take appropriate action. Once the proper voltage arrives, a state

machine triggers and the configuration circuitry automatically cycles through its address space of configuration frames, writing zeroes into all the cells. This eliminates the possibility of internal contention, which might damage the part. This memory resetting occurs twice, to assure proper initialization.

After clearing the FPGA, the control circuitry samples the mode pins, to discover what the method of bit delivery will be. Table 5-1 summarizes the modes for the Virtex 4 family, but the available modes have varied among Xilinx FPGA devices over the years. Table 5-1 represents the most frequently used modes, and a couple of key ideas need to be stated.

Configuration Mode	M2	M1	M0	Data Width	CCLK Direction
Master Serial	0	0	0	1 bit	Output
Slave Serial	1	1	1	1 bit	Input
Master SelectMAP	0	1	1	8 bits	Output
Slave SelectMAP	1	1	0	8 bits	Input
JTAG/Boundary-Scan only	1	0	1	1 bit	—

Table 5-1 Virtex 4 Configuration Mode Options

Like “source synchronous” data transmission, the master modes refer to which device in a chain of FPGA devices presents a configuration clock (CCLK) to the rest of the chain. There will be only one in a given chain. It is possible to have the FPGA internally create a clock, whose frequency is selectable among a set of predefined choices. Because these are defined by internal delay parameters (read: ring oscillators), the user selectable values vary among the Virtex families, which are built from different technologies. As we will see shortly, there are a set of standard configurations of chained FPGA devices that subscribe to the above modes. When multiple FPGA devices are chained together, their composite bitstream is presented to the chain, and a protocol determines which bits go where.

After sampling the mode pins, the configuration controller decides what to do next, as a set of predefined actions. The configuration bitstream includes a synchronization word – a specific 32 bit word, with groups of ones and zeroes that are carefully checked, to guarantee correct reading of the word, at the previously specified clock period. If this is successful, the next step is to make certain that the bits which follow are for the receiving device. Hence, a device ID pattern is next checked against an internally stored constant within the particular Virtex part. Because frame sizes, and bit arrangements vary among the Virtex family, this check is critical, to assure that data frames of one family are not inadvertently loaded – meaninglessly – into another Virtex family member. Possible results include damaging the target part, if done incorrectly.

Assuming synchronization and device ID check went correctly the configuration payload is the next set of arriving bits. These bits include both configuration data – the CLB bits, BRAM bits, etc. – but also instructions and parameters to the configuration controller, for a set of specific actions the configuration controller must handle. See Figure 5-4.

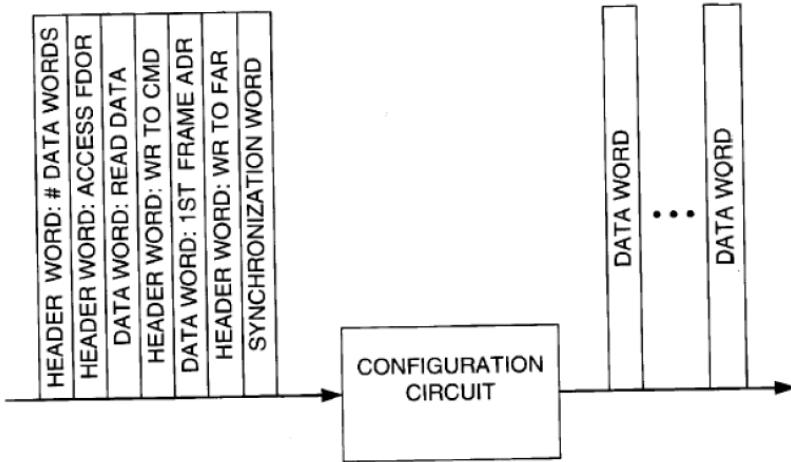


Figure 5-4 Bitstream Data In, Configuration Data Out

From the delivered bitstream, it is possible that data is directed to land at specific address regions (i.e., frame addresses), and it is possible to deliver one set of data, and have it copied into consecutive frames, etc. This permits some level of bitstream compression. Through the duration of bitstream delivery, a CRC state machine within the FPGA tallies a finite field calculation that is unique for the delivered bitstream, and serves as a signature at the end of the process. At the end of the bitstream delivery, the FPGA calculated CRC value must match the one appended to the bitstream at the end of the process, or an error condition is flagged into the configuration controller error register. Done will not go high if this happens. Done is the pin signaling that all went well.

After the CRC is checked, a set of user options is entertained. Two key options are synchronization of internal DCMs (possibly thousands of clocks) and calibration of the DCI circuitry, requiring several trial and error actions, also. Both of these can, optionally, occur before the “done” flag is asserted, so the FPGA apparently is configured, time synchronized and impedance matched to its environment, all within the “configuration timeframe”.

Anyway, the default of “startup” is to simply initialize the BRAMS, enable the outputs and assert done. Let’s look at a set of connection choices available to various FPGA configurations.

## Configuration Options- Serial Mode

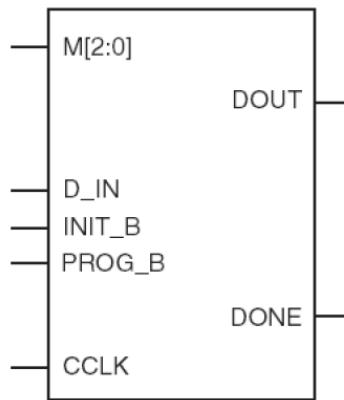


Figure 5-5 General Serial Configuration Model

Figure 5-5 shows the set of input and output pins associated with serial configuration. The mode pins are constants defined by binary values shown in Table 5-1. D\_IN is the bitstream input pin, and typically attaches to a serial PROM, or another FPGA's DOUT. Hence, DOUT is the output site for cascading or "chaining" to another FPGA. The FPGA closest to the bitstream starting point is called the most upstream FPGA, and successive FPGA devices in a chain are more downstream. CCLK can be either an input or an output, depending on the predefined position of the FPGA and the pre-agreed upon position it has within a chain. The master device creates clocks for successive slave devices. PROG\_B is an input that is active low and performs a full chip reset. INIT\_B is a complex I/O pin that when held low delays configuration until being raised high. After configuration begins, INIT\_B is an active low, open drain signal indicating whether a CRC error has occurred. INIT\_B switches functions at the point when the mode inputs are sampled.

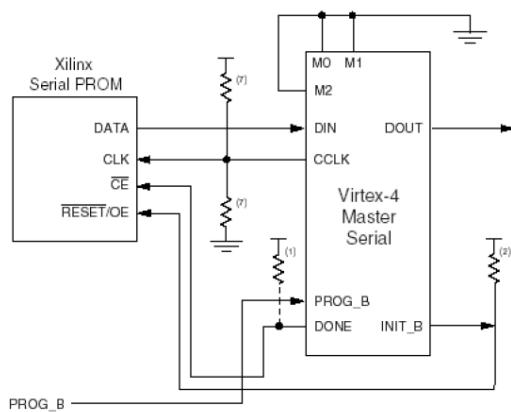


Figure 5-6 Master Serial Mode

Figure 5-6 shows the master serial mode. All mode pins are tied to ground, CCLK comes from within the FPGA and is delivered to a Xilinx Serial PROM, which in turn delivers DATA to the FPGA DIN pin. Note that INIT\_B is pulled up, and fed back to

the active low RESET/OE pin on the PROM, so that configuration restarts, if a CRC error occurs. The DONE signal can be programmed as an “active” or internally pulled up pin, or alternately as an open drain pin, needing a pull up. When cascading FPGA devices, tying their DONE signals together must be performed by having all but the most downstream FPGA configured with their DONE signal as an open drain pin. One pull up resistor for the chain is required.

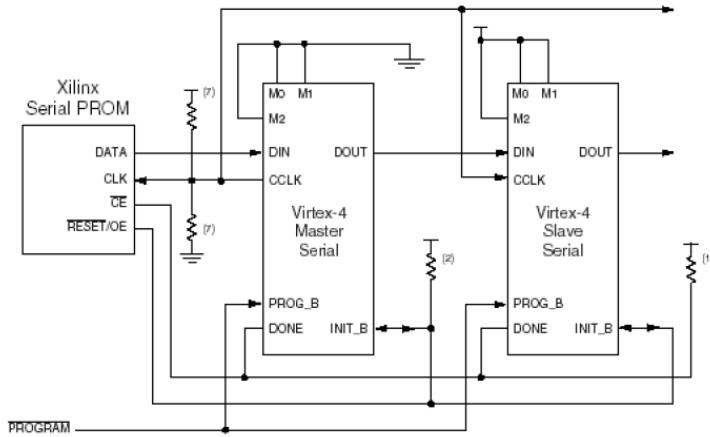


Figure 5-7 Master/Slave Serial Configuration Chain

Figure 5-7 shows a cascade, where the leftmost FPGA is configured like Figure 5-6, but an additional FPGA's DIN is attached to the DOUT of the left FPGA. Here, we also see the upstream FPGA (left) delivering a clock to both the Serial PROM and the downstream FPGA (right). The INIT\_B signals are tied together, and also assert the chip enable (CE) input to the PROM. Note that the input labeled PROGRAM is an external signal which will begin programming, when driven high. In this situation, we see that the upstream FPGA and downstream FPGA DONE pins are joined and terminated with a pull up resistor, so the overall configuration completes when the ensemble DONE goes high. Finally, note that the downstream FPGA is indeed, configured as a serial slave by tying all mode inputs high per the assignment from Table 5-1.

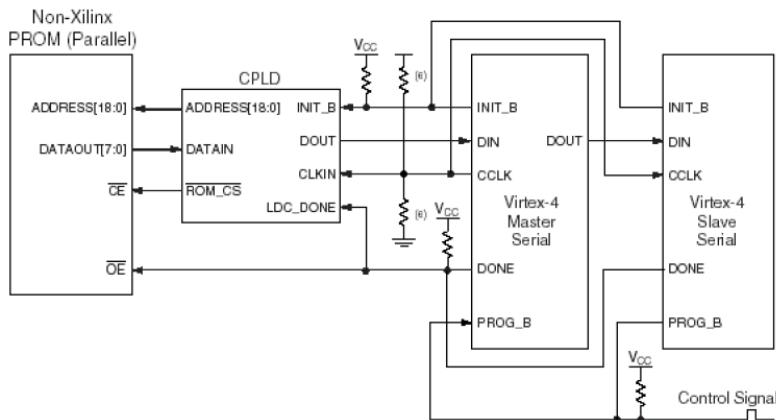


Figure 5-8 Master/Slave Serial Configuration from Parallel EEPROM

Many possibilities exist for configuring FPGA devices serially. The one shown in Figure 5-8 shows a case where a non-Xilinx parallel PROM is made to resemble a serial PROM by inserting an inexpensive CPLD (Xilinx, naturally) between the parallel PROM and the FPGA chain. The DOUT bitstream coming from the PLD now emulates the bitstream coming from a serial PROM. The internals of the CPLD include an address counter, stepping through consecutive PROM addresses, a shift register that serializes the PROM's bytes, and a state machine interacting with both the PROM and the FPGA devices. Similarly, a microprocessor could substitute for the CPLD, extracting data from a PROM and delivering it to the FPGA chain.

### Configuration Options – SelectMAP Mode

SelectMAP mode is the choice, when faster configuration is desired. In this case, data arrives to the FPGA(s) on an 8 bit parallel bus, so speeds can approach 8X that of the serial modes. Figure 5-9 shows the pertinent pins for SelectMAP mode. The mode pins, INIT\_B, CCLK and PROG\_B all have similar functions in SelectMAP configuration. Note that DOUT is not present, as we will not chain through upstream FPGA devices, but rather around them. A new sentinel, BUSY is present to show that this specific FPGA is being programmed, and DONE takes on the usual meaning.

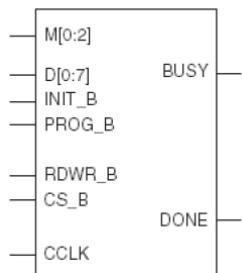


Figure 5-9 General Parallel Configuration Model

Two additional signals are shown in Figure 5-9 called RDWR\_B and CS\_B. CS\_B is an active low chip select, which is applied (typically) to a single FPGA at a time, when delivering the byte stream to that FPGA. RDWR\_B is an input signal controlling the data direction for the byte stream – low for inputting, and high for outputting (i.e., readback).

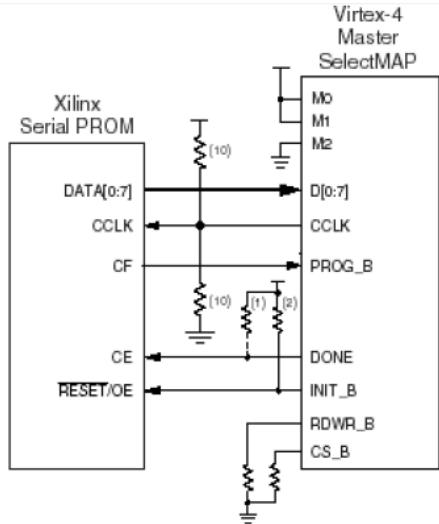


Figure 5-10 Single Master SelectMAP Configuration Model

Figure 5-10 shows a single FPGA getting its bitstream in SelectMAP mode. The eight bit data bus coming from the PROM directly attaches to the eight bit bus on the FPGA. Because this is a single FPGA situation, and no readback will be needed, both RDWR\_B and CS\_B are shown tied low through resistors. This specific connection is a possibility with Xilinx Platform Flash PROMs, which permit parallel outputting, but must be properly designated by a parallel choice, when creating the bitstream with the software.

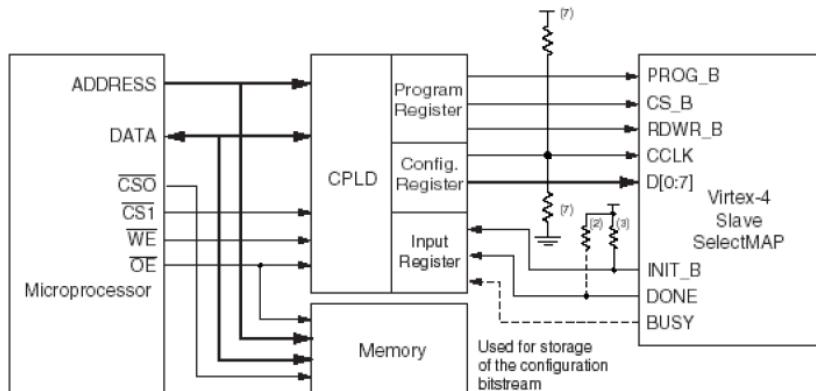


Figure 5-11 Single Master Configuration from Microprocessor and CPLD

Another SelectMAP configuration circuit is shown in Figure 5-11, where the microprocessor DATA bus is buffered through a CPLD, resolving the various handshake signals. In this case, the more general “Memory” is shown as the data source, and may be simply any memory structure (EPROM, Disk, DRAM, SRAM, etc.) attached to the microprocessor bus.

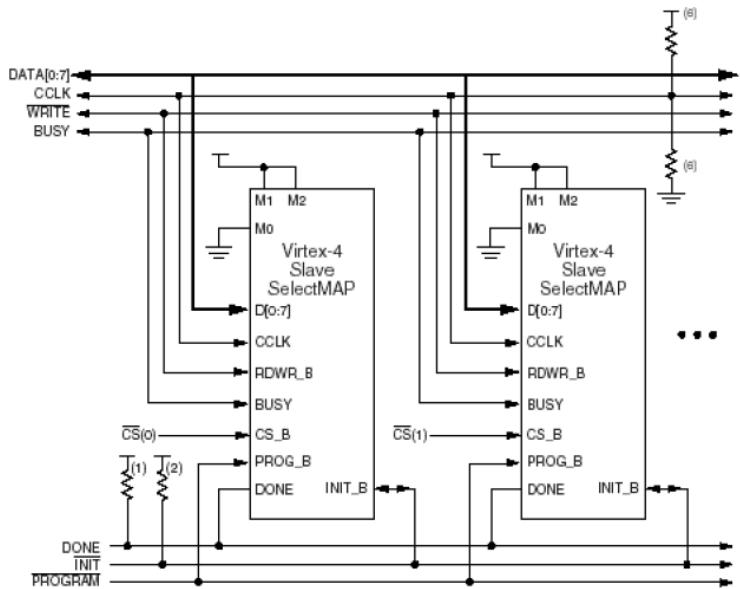


Figure 5-12 Multiple SelectMAP Configured Devices

Figure 5-12 shows a cascade of multiple similarly configured SelectMAP FPGA devices, with appropriately connected control signals, and terminations. Because the chip selects (CS(0), CS(1)) are unique in this bus configuration, we lose the sense of upstream/downstream and in this case, no FPGA is shown driving CCLK, so all participants are slaves.

### Configuration Options – JTAG Mode

The original JTAG (i.e., IEEE 1149.1) test circuitry evolved over time to include the ability to configure programmable logic devices. This was done by a team of device stakeholders seeking to serve customers needing to configure multiple devices, from multiple vendors in a single connection chain. The facility both tests devices, as well as programs them. Figure 5-13 shows such a chain comprised of Virtex 4 FPGA devices, but any Virtex parts or other devices could be included into such a chain.

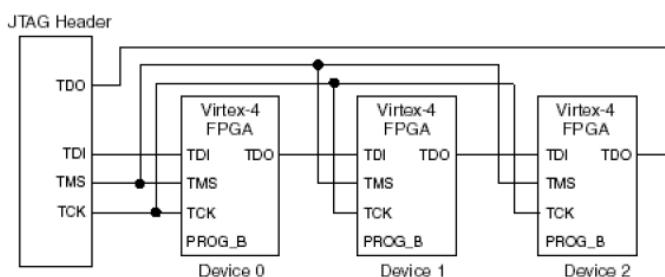


Figure 5-13 JTAG Configuration Scheme

The primary data input signal here is Test Data Input (TDI), conveying the bitstream and commands into the FPGA. Should the bitstream not be for the first FPGA, the

data forwards through onto Test Data Out (TDO), attached to the next FPGA's TDI, and so forth until the last device delivers its TDO back to the original data source site. That source is typically a microprocessor, connected through a configuration cable. The Test Clock (TCK) signal and the Test Mode Select (TMS) signal are used together, to place the internal circuitry into various data delivery modes, programming modes and test modes. We won't detail how the 1532 circuitry works, but it uses the same basic idea as the other serial modes and is similar. It is notable, that JTAG programming is frequently the mode of choice for system prototyping, but switching to another mode (serial or SelectMAP) is preferred when moving the product to manufacturing.

### Under the Hood of the Configuration Controller

Figure 5-14 is a block diagram of the clever circuitry inside the FPGA controlling the overall configuration process. The upper left side shows the General Interface Circuit, which interprets the Mode pins, delivers CCLK and interacts with, or produces the various control signals for data handshake. Below that, is the JTAG circuit that extracts data from TDI and forwards it on, to TDO. Both blocks interface to a 32 bit internal bus, to which are attached multiple other Frame support registers.

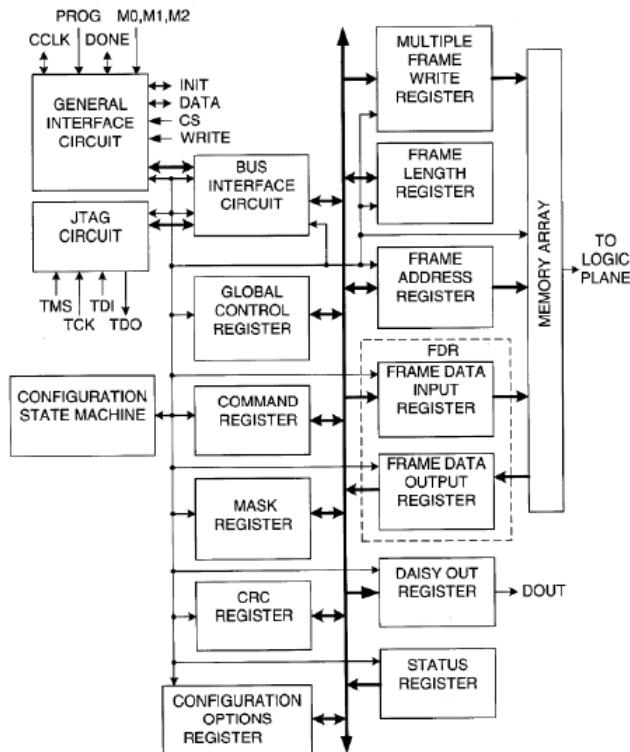


Figure 5-14 Configuration Control Circuit

The CRC circuit is a linear feedback shift register (LFSR) calculating a CRC from a 32 bit finite field polynomial. This is one aspect of the error control circuitry within the configuration circuitry. The Configuration State Machine collaborating with the Command Register directs the actions of the other registers that manage frame data. For instance, the Frame Address Register (FAR) holds the frame address to be written

into, where the Frame Data Input Register holds the current data frame to be written to the configuration memory. The Frame Data Output Register is the feedback path, for configuration verification. The Daisy Out Register holds the data being forwarded to a downstream FPGA. The Configuration Options Register holds the user choices for activities like synchronizing the DCM and initializing the DCI, as well as the defaults for configuration, like enabling the I/O pins. The Status Register holds the current status of the configuration operation (Busy, Done, Error, etc.). Most of the registers are 32 bits wide, as is the central bus.

There are additional state machines – packet processors, masking circuits and so forth. The basic structure is much the same across the various Virtex families, but frame sizes vary among the families, and other small details also differ. This structure is basically a microprocessor, receiving its instructions from the same bitstream arriving with the configuration data. Recalling Figure 5-4, we saw bitstream arriving on the left, with data, parameters and commands embedded, and configuration data exiting on the right. Figure 5-14 details the block previously shown in Figure 5-4. Because of the underlying flexibility in the configuration structure of CLBs, as well as the separation of BRAM control tiles from BRAM data tiles, additional dimensions of configuration are possible – notably reconfiguration. Let's briefly look at that now.

## Reconfiguration

Reconfiguration is the ability to change the configuration. SRAM based technology has been highly desirable in this regard, in that being able to modify a design, an indefinite number of times, is very attractive. Just changing the FPGA contents while it is attached to a board, to correct errors, or do incremental design, is the forte of Xilinx FPGA devices. But, the possibilities for this capability are not limited to those ideas, alone. Here are some more ideas based on reconfigurability:

1. Field upgrade – changing the design when it is installed at a remote site.
2. Feature enhancement – adding features to an already deployed design.
3. Reconfigurable coprocessor- changing a coprocessor function
4. Modify connections on a printed circuit board – reroute existing connections
5. Self reconfiguration (or dynamic reconfiguration) where an FPGA modifies itself
6. Self-correcting circuits – circuits that recover from failure, automatically
7. Others, yet to be imagined, based on partial reconfiguration

Let's explore some of these ideas in greater depth.

Field upgrade is the process of introducing an improved circuit into an existing product. This may be bug fixing, where a bug was discovered after the system shipped. A simple example of this might be that two high order address lines for a microprocessor system were accidentally exchanged, and the new pattern makes them have the correct ordering. It could also be any logic error totally contained within the FPGA. The primary requirement is access to the configuration circuitry from the outside world, in some way. The upgrade could be delivered by a simple portable laptop computer and a JTAG cable, or it could be delivered over the Internet and downloaded through

system circuit access that already exists. Files have even been delivered by email to remote machines which then upgrade themselves without human intervention.

Feature enhancement is similar to field upgrade, but instead of fixing a bug, it is adding something new to the system. One idea would be to offload a microprocessor calculation that is a performance bottleneck, to a hardware version contained within an FPGA, having available logic capacity to perform the task. The primary requirement here – beyond the configuration port access, is that the FPGA was previously connected appropriately into the system to support reconfiguration. Specifically, it would have to be connected to the microprocessor bus(es) as dictated by the required hardware change. Reconfigurable co-processor is a variation on the idea of a feature enhancement. In this case, the task of the FPGA is to perform tasks offloading a microprocessor and act as its “co-processor”. A co-processor usually does a bottlenecking task for a microprocessor, and takes advantage of the FPGA devices fast hardware facilities and pipeline features. Communication between the FPGA and microprocessor for data would be through buses. Control communication could be by agreed upon timing, whereby operands dispatch to the FPGA and somewhat later the microprocessor collects results. Alternately, the FPGA could directly drop results into pre-specified memory locations, or simply assert an interrupt input to the microprocessor, when done. The range of co-processing tasks is vast. It could be any of a broad number of digital signal processing actions – FIR, IIR, FFT, or finite field arithmetic, which is typically not handled well by standard microprocessors. There appears to be no limit to the types of operations such a coprocessor could perform.

Connection modification is almost trivial with an FPGA, unless its 100% packed. Basically, an FPGA can implement a cross point switch, which can connect any input to any output. Other switches, requiring fewer resources, but delivering perfectly acceptable connectivity include Banyon nets used in data communications. Even other topologies exist as choices.

Self-reconfiguration is simply the ability of the FPGA to initiate its own reconfiguration. This is a very interesting proposition, in that it would be possible for the FPGA to be acting as a coprocessor, then become a different function – in real time – as needed. A similar idea here, is a memory hierarchy system where a cache controller recognizes a memory page isn't loaded into cache, automatically loads it, and resumes execution, when completed. Here, the FPGA needs to know what configuration to load, and must be able to perform a partial reconfiguration. Configuration is “partial” because the update controller must remain within the FPGA, for future self reconfiguration episodes. Self reconfiguration is not for the casual designer, at this time and requires substantial design planning. We will look even closer at self-correcting circuits and partial reconfiguration, in the next two sections.

## **Self-Correcting Circuits**

This broad category of designs is based on much theory going back to the time of John von Neumann and associates that wrote about designing highly reliable circuits, from unreliable components. This involved a form of systematic logic replication. If one of the duplicates fails, another will most likely have the correct behavior, so just use that circuit's result. With only two identical circuits, how do you know which one is

wrong? The obvious step is to take three copies of the design, and the highest likelihood is that two of the three will agree, if not all three circuits. The method of selecting uses a logical majority function, which identifies the set that matches. This process is sometimes called voting. Logic created in this way is called triple redundant logic (TRL). Because this style of design makes solutions even more expensive, you might be asking: What does this have to do with FPGA devices?

When SRAM based FPGA devices are placed into an environment with higher degrees of radiation particles (outer space or high altitude aircraft), they are exposed to having an occasional particle collide with an SRAM cell, and change the functionality of the FPGA, by modifying the cell. This is called a soft error upset or SEU. This is a tiny probability, but is a possibility, and designers developed methods to reduce the likelihood of failure, by using triple redundant design, along with a simple reconfiguration process called “scrubbing”. The idea behind scrubbing is simple. Triple redundancy lowers the error rate, but over time, its effectiveness may decrease as circuits are bombarded and fail. An SRAM cell isn't permanently damaged, its state is just scrambled, becoming unknown. By periodically reloading the FPGA fabric SRAM cells, the logic automatically becomes correct again. There are details in how often this needs to be done, but those are discussed in more depth, in Xilinx application notes.

Xilinx engineers have worked with various experts in this field, both measuring SEU occurrence, as well as evaluating effectiveness of triple redundant logic and scrubbing. These methods proved effective enough for Xilinx FPGA devices to be chosen as key technology to participate in recent Mars Rover designs. NASA designers are clearly comfortable enough with these methods to risk vast sums on their reliability. We won't pursue triple redundant logic, but let's discuss scrubbing.

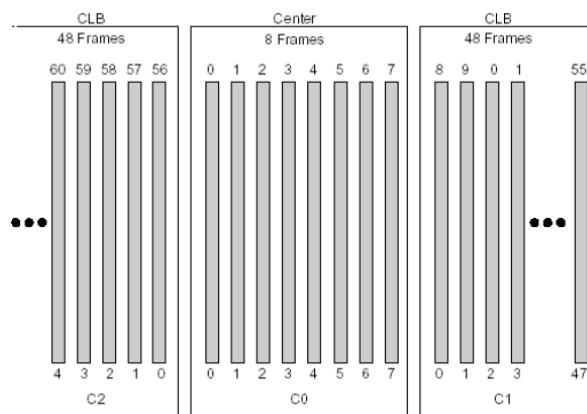


Figure 5-15 Configuration column sequencing

Figure 5-15 shows some Virtex configuration frames, grouped into columns, and labeled with numbers. This is the default sequencing of the frame data to be loaded into the configuration memory columns. Specifically, there are 8 frames in the central column, getting loaded first. Then, to the right of that are 48 frames getting loaded next, and to the left of the central columns are another 48 frames getting loaded after that. The process proceeds, by default, in this fashion until the whole chip configures,

with the outside edges being loaded last. It is possible to alter this sequence, by issuing a bitstream into the part that places the data in a different order. This is because the configuration controller actually uses commands within the bitstream, that dictate what frame address is to be targeted, with the Frame Address Register (FAR) and what data goes there with the Frame Data Input Register (FDIR). For error correcting, a controller could readback configuration data via the Frame Data Output Register (FDOR) and compare contents with corresponding bits in an off-chip EPROM. Then, it could reload defective frames. In fact the latest devices include single error correct double error detect syndromes for each frame so that error may be found and fixed as they occur continuously while in operation.

When added to triple redundant logic, the failure rate approaches nonexistent. Xilinx has spent much effort developing test procedures using both neutron beam and varying altitude measurements to arrive at a suite of solutions that provide many options for designers to choose among. Depending on the configuration bitstream size, possible failure rate desired and a number of other parameters, designers can decide how frequently it makes sense for the scrub circuit to cycle through the configuration space, refreshing bits, or enabling the device itself to find and fix errors.

## Partial Reconfiguration

Partial reconfiguration (PR) is an idea just gaining momentum, as a standard technique. After many attempted design flows, Xilinx has released the following as production worthy: the Partial Reconfiguration in the Vivado Design Suite -[partial-reconfiguration.html](#).

## Bitstream Security

Developing a design in programmable logic, can be a time consuming effort, depending on the constraints and the task at hand. Sometimes, many hundreds of hours are spent exploring tradeoffs and deciding on the best set of design choices. Developers are quite aware that their bitstream – all those dollars wrapped up in time – are at risk during the configuration process, as a design reversing engineer could simply capture the bitstream while it is loading and format it into their own EPROM. They would be in business making copies of the original design, with little investment. Xilinx has identified a number of ways to reduce this problem, giving designers greater assurance of the safety of using programmable logic.

Let's first discuss the general notion of security, and identify that not all end users seek exactly the same level of bitstream security. For instance, as mentioned above, there are design thieves called "cloners" wishing to simply save the R & D expense for a system, and make duplicates at a much cheaper cost. There are also banks (ATM machines), military (air to air communication), government (secure cellphones) and even gambling interests. Each has their own requirements. Let's consider slot machines, briefly. Contrasting to cloners, gaming commission experts insist that the entire contents of any PLD make its configuration memory completely exposed. They don't care about people duplicating the contents. Their security model is based on regulations and random auditing of a gaming floor. When an auditor shows up to inspect a slot machine, he knows what the internal configuration of the programmable

logic memory should be, and interrogates the machine (by programming cable) to verify that it is correct. His primary concern is that the contents cannot be altered – erased or reprogrammed. In fact, they may attempt to do so, and prove whether it can't be done. Their goal is to guarantee the “odds” don't change in the machine due to some form of tampering. Similar criteria might occur for a secure cellphone. Modifying the phone's authentication transaction could be disastrous. If a secure phone landed in the hands of a terrorist, he might create vast damage, so guaranteeing the correct authentication occurs is an important aspect of that security.

Bitstream security is but one aspect of larger security models, and Xilinx' choices for encryption and embodiment of the decrypting circuitry is superior in the marketplace. Let's briefly discuss the use of DES and triple-DES (a.k.a. 3DES) which was first introduced with the Virtex-E family, then subsequently the Virtex-II products.

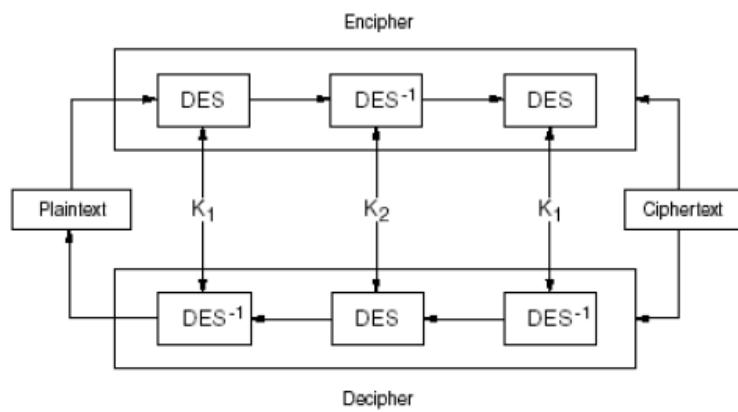


Figure 5-20 Triple DES strategy in Virtex-E and Virtex II

DES was originally developed over thirty years ago by I.B.M. and adopted by NIST as a standard, after modification by the NSA and others. It uses a 56 bit key, to combine with a crafty algorithm to encrypt data. Due to advances in computing power, it was decided to recommend a modification to the DES standard, which involved multiple passes of the data through the DES algorithm, with multiple keys. The most secure version is shown in Figure 5-20. Plaintext (bits) start on the left, and go through the DES algorithm with key K1. This result then goes through the inverse DES algorithm with another key, K2. This second result then goes once more through the DES algorithm with a third key, K3. Each key is a different, random 56 bit value. This results in the same number of bits on the right hand side, as started on the left side. The flow just described represents the top “Encipher” flow in Figure 5-20. Decipherment – what the FPGA will be doing, is shown along the bottom of Figure 5-20, from right to left. Normally, software combines the bitstream with three keys as in the top flow, and an FPGA with 3DES and three identical internal keys performs the bottom flow. As a working model, assume that a factory encrypts the bitstream (top flow) and programs that into an EPROM. The EPROM and the Virtex FPGA are attached to a board and connected. The FPGA then has three 56 bit keys programmed into a key memory, and a tiny battery attached to special terminals that only connect to the key memory.

Power can be denied to the board, but the FPGA still retains its keys due to the battery.

Disconnecting the battery results in immediate loss of the key memory, and the FPGA keys cannot be accessed. To resume, the FPGA would need to have a battery reattached and three 56 bit keys reprogrammed into it. If this were a cloner, he would not have the three keys, and be unable to steal the design.

Much thought has been put into detail on how people with great knowledge of Xilinx products might attack these designs to gain the keys, and additional protocol and circuit steps have been taken to thwart these actions. Including Cipher Block Chaining (CBC) nicely blocks dictionary attacks which focus on using statistical patterns to attack a design. It also nicely stops relocation attacks, where an attacker modifies the bitstream to attempt placing clear bits into an accessible region of the FPGA. As a policy, Xilinx supports users that wish to follow the best security practices. These include using the standard algorithms, which have withstood substantial scrutiny from the cryptography and security communities. Xilinx methods focus on keeping all security in the key, and operating with public algorithms. In fact, much detail on how to understand Xilinx bitstreams is publicly available in the various documents on configuration, and use of the internal logic analyzer (ILA).

Full details on the 3DES and the Advanced Encryption Standard (Virtex 4 and later) are available online, from the NIST website, and are freely available.

However, let's take a closer look at the internal structure of what is attached to what, so you can see how the decryptor connects into the configuration controller. Figure 5-21 shows a high level diagram including the two programming ports (standard and JTAG) along with the key memory and the decryptor. Various paths are available to divert bitstreams to and from the configuration controller and the decryptor, as shown.

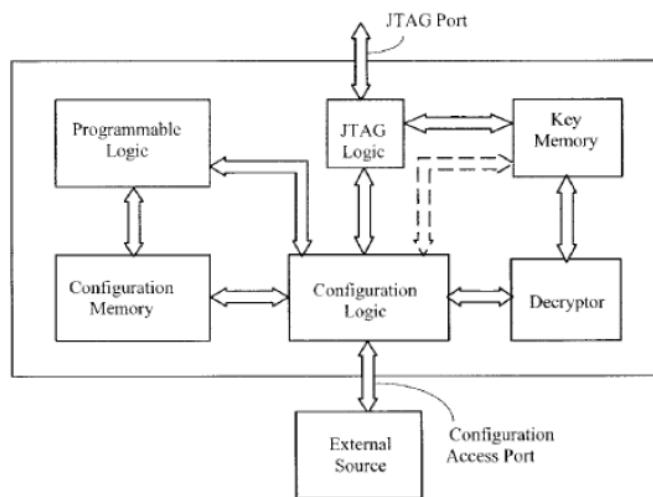


Figure 5-21 High level look at FPGA, Configuration and Decryptor

Figure 5-22 shows greater detail on how the Decryptor is selectively involved with the bitstream. Specifically shown are the encrypted bitstream paths coming in through a 64 bit register, as well as the CBC paths to perform the cipher block chaining. Note that the CBC register must be preloaded before the decryption begins with starting seed values, to maintain generality. As a quick summary, cipher block chaining modifies consecutive blocks of plaintext, EX-ORing the previous block of plaintext with

the current block, prior to running the current block through the encryption mill. In this way, all blocks contains some level of information from all other blocks, increasing both confusion and diffusion aspects of the process. It is vital that the other end of the process reverse this, to obtain correct clear bits. CBC mode is supported by a wide range of commercial applications and can be regarded as an industry standard approach to any encryption protocol.

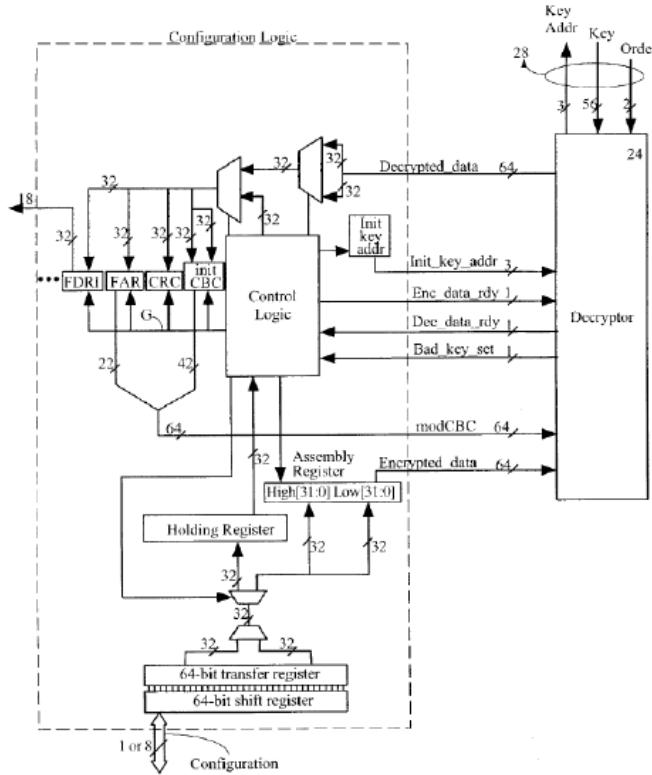


Figure 5-22 Greater Detail on Decryption and Configuration Interaction

During the development of Virtex 4, NIST deemed that DES was appearing to “wear out”. Paul Kocher from Cryptography Research had managed to crack DES by using an attack he developed called differential power analysis and an array of Xilinx FPGA devices. The attack was on a microprocessor executing DES and he managed to identify a 56 bit key. When this sort of thing happens, it is time to move on and adopt a successor. NIST has withdrawn DES as a standard, but retains its endorsement of 3DES. In that light, Xilinx maintains full support of the 3DES standards embodied in Virtex E and Virtex II families, but has moved forward to the newer AES standard in Virtex 4 and later devices. Details on AES can be found at: <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf> The version of AES being used is a 256 bit key version, embodied as custom logic on the FPGA die. CBC has also been retained, as well as the battery backed up key memory. Xilinx software provides appropriate user support to properly create encoded bitstreams and manage keys for development.

An additional user interface permitting direct access into the FPGA internals, called ICAP is available, but must not be used for final designs, as it would permit user access to the clear bits area of the FPGA configuration space. If ICAP is not

instantiated in a design, hardware lockouts block unauthorized user access to the configuration space.

## **Summary and Conclusion**

To many designers, details of the configuration circuitry are a necessary evil. Others have the imagination to see that modifying configurations, securing bitstreams and finding great ways to "tweak and tune" designs "on-the-fly" and in real time provide design dimensions previously only dreamed about. The design team at Xilinx works hard to create user facilities that encourage creativity in these dimensions.

## **Final Comment**

During the original edition of this text, Spartan 3E was introduced, and included a new configuration standard employing off the shelf SPI EPROMS directly attaching to the FPGA. In order to do this, additional pin options and other modes were developed for the configuration controller circuitry. This is a boon to users, as these EPROMs are extremely attractive from a price viewpoint, making this a highly desirable solution. Subsequent devices all now have the SPI flash as an option, along with is by 4, and b8 cousins, providing a standard means of non-volatile storage and configuration.

### **Additional References:**

1. US Patent # 6,429,682 B1, Configuration Bus Interface Circuit for FPGAs, by David Schultz, Larry Hung, Erich Goetting
2. US Patent # 6,366,117(B1),Nonvolatile/Battery-Backed Key in PLD by, Raymond Pang, Jennifer Wong, Scott Frake, Jane Sowards, Venu Kondapalli, Erich Goetting, Steve Trimberger, and Kamesh Rao
3. Additional resources are contained online in XAPP 216, XAPP270 and XAPP 290
4. UG071 Virtex-4 Configuration Guide
5. 7 Series FPGAs Configuration User Guide UG470 (v1.10) June 24, 2015

# Chapter 6 Virtex Memory

## Introduction

Virtex memory structures evolved over several generations. Distributed Select RAM (DSR) was first introduced into XC4000 FPGA architectures as a means of creating small RAM structures that could be cascaded, by using the lookup table structure in the logic elements as data storage. This limited the density of RAM to sixteen bits per module. Users requested denser, more efficient structures to handle DSP and data communication operations without sacrificing vast amounts of logic fabric. Xilinx architects created configurable block memories, to meet that need. In theory, the payoff of including RAM within the FPGA is similar to that of cache memory within a microprocessor. External bus bandwidth is reduced and internal throughput increased, by simply making the data transactions occur within the device, rather than always going external. This also dramatically reduces the I/O activity, gaining improved overall signal integrity (less noise). Before we look deeper into the memory structures, let's take a look at some of the applications these resources are designed to support.

## RAM Applications

Using table lookup is ingrained into most of us, from our early years in grammar school, when we learned the addition and multiplication tables. That method set a standard for computational efficiency, remaining today. When any digital algorithm is created to calculate something, its speed and memory requirements are often compared to doing the same task “once and for all”, and loading the pre-computed answers into a RAM. The solution that runs faster is the choice. With the plummeting costs of electronic memory and disks today, that approach is frequently used. This opens up the first application area for large memories – mathematical lookup tables. Why create a sophisticated state machine to calculate something esoteric, when a program can do it, and place the results into fast RAM. It also delivers uniform time delay to provide the results, which is an advantage, too.

Data structures, like stacks and queues, are easily implemented with SRAM structures, to hold data. Data communication and instrumentation are two application areas often handling ordered data, captured over time, where the sequence must be maintained. First in first out (FIFO) structures are easily built from block RAMs with additional counters and control logic to handle the ordering of data, along with control logic to manage collisions and overflows. Adding dual port memory, where independent reading and writing can occur with the RAM simultaneously, streamlines these designs to meet the needs of today’s high speed world.

Math lookup tables and FIFOs are probably the most frequent uses for RAM, but others exist, among which are:

1. Operand stacks
2. Register files
3. Instruction caches
4. DMA buffers
5. Instruction memories
6. State tables
7. Logic functions
8. Message buffers
9. Virtual channels
10. Video line buffers
11. Digital delay lines
12. RAMDAC color mapping tables
13. Test vector buffers
14. PCI configuration memory
15. Sequential machines

As a running example, we will follow the evolution of FIFOs, through various embodiments of the Virtex RAM resources.

### Distributed Select RAM and the SRL16

Chapter Three discussed the Virtex architectures and overviewed the slice capabilities. Let's look in more depth at the LUT structure, from the point of view of the LUT as a shift register, then with the LUT viewed as a distributed RAM. Figure 6-1 shows the LUT with focus on its ability to shift and be randomly accessed. Note that we have also included (in dashed boxes) the addition of the configuration data bits, so you can see how the logic shown must be both a logic function, as well as a support function.

Examining Figure 6-1, there is a set of 16 storage cells shown as D flip flops. The circuitry is different from a standard D, as discussed in Chapter 1. In fact, it is a latch structure with input capacitance being loaded by a narrow pulse derived from the input clock. This approach reduces area, and saves power while still meeting the storage requirement. Each D latch is driven from a multiplexer providing data from one of three sources at any time: configuration bits, SRAM data bit or a previous shift register bit. Moving forward, we will refer to the LUT storage cells as D flip flops, anyway. The DATAOUT for the LUT is a particular D flip flop (from sixteen) value, defined by the select input address ( $A_0 - A_3$ ) to the LUT. This is the same address as the truth table row selection, but in this context, it is either a shift register position, or a distributed SRAM memory address.

### SRL16

Let's first examine the SRL16 behavior, which is the 16 bit shift register. Data enters the shift register at the top of Figure 6-1, and exits at either the DATAOUT point from the multiplexer, or from the Q of D flip flop #15, where it is presented to the CLB

infrastructure through one of the internal multiplexers. This allows direct cascade of the shift register from one LUT, to another in the “southern” direction. It is possible to build up substantially long shift registers by cascading multiple SRL16s, from slice to slice within a CLB, and between CLBs. Depending on the FPGA family chosen, the cascading of shift registers will vary somewhat. Virtex, Virtex-E, Virtex-EM, Virtex-II and Virtex-II Pro permit cascading of every 16 bit shift register within the CLBs. Spartan 3/E and Virtex 4 restrict this capability to only half the slices within the CLBs having the full support to create shift registers and distributed RAM. Virtex 5 follows the philosophy of Spartan3/E and Virtex 4, but with its six input LUT structure.

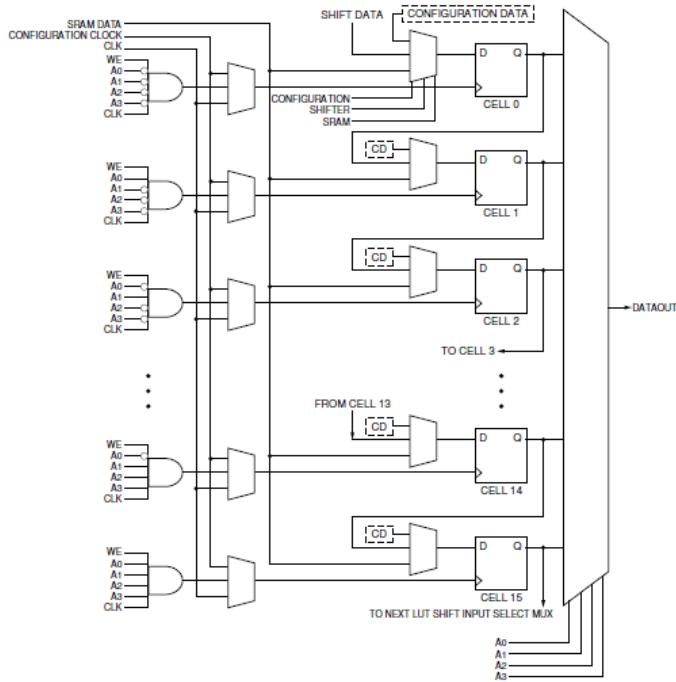


Figure 6-1 Memory Structure of LUT RAM/Shifter

Because the SRL16 structure permits both the DATAOUT exit site for bits as well as the cell 15 Q output, there are some interesting ways to create shift based structures. For instance, you can create linear feedback shift registers (LFSR) consisting of shift register chains interspersed with EX-OR gates, that ultimately feedback upon themselves. The EX-OR gates are formed by connecting to other LUTs with the EX-OR truth table installed. The LFSR structure lends itself to efficient cascades of 16 bit shift registers with shorter chains, as needed using the DATAOUT sites. At the various cascade sites, the EX-OR LUTs are inserted, as dictated by the LFSR polynomial.

Figure 6-2 shows one such LFSR built in the Galois form, where the EX-OR gates are connected “in-line” with the shift register chain. Figure 6-3 shows the other standard form (Fibonacci), where the EX-OR gate(s) are connected out of line with the shifter cells. The attachment sites are based on a mathematical polynomial, and the ordering of flip flops is important to correctly make attachments. Galois and Fibonacci structures are a little different from each other, so care needs to be taken in their attachments. Xilinx has a wide set of application notes on constructing LFSRs, and

the software creates these efficiently, and automatically. Figure 6-4 shows how several SRL 16E type modules might be connected to create a fairly large LFSR. Several standard SRL 16 varieties are available in the design software library.

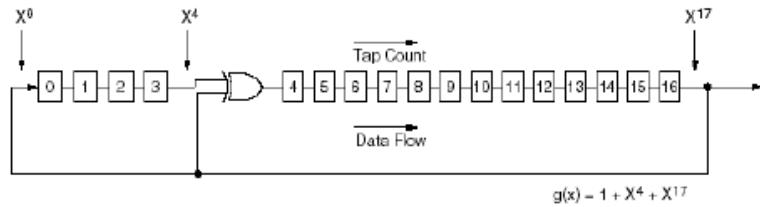


Figure 6-2 Galois style LFSR

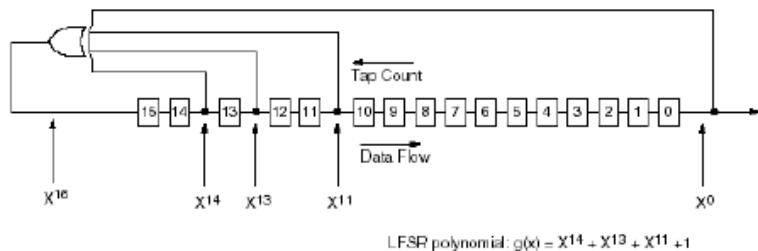


Figure 6-3 Fibonacci style LFSR

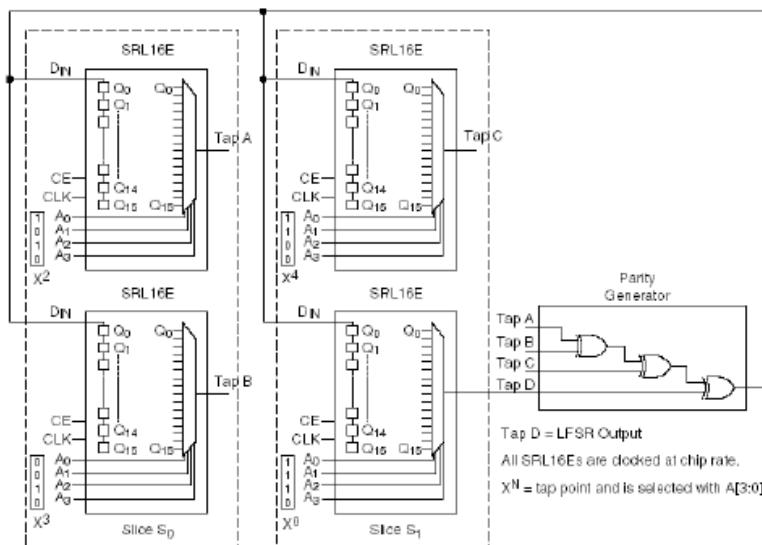


Figure 6-4 Multiple SRL 16E LFSR structure

Naturally, all of the SRL16 structures that apply to earlier Virtex and Spartan architectures are applicable to the Virtex 5 SLICEM and later devices up to UltraScale+™, only larger. Figure 6-5 shows cascading SRL32s for Virtex 5, to illustrate.

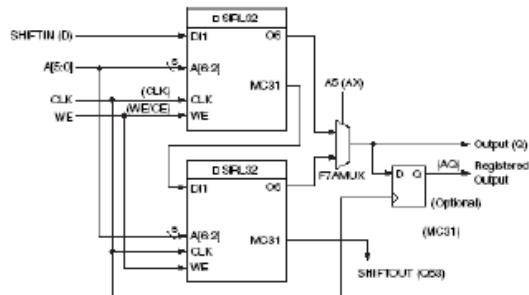


Figure 6-5 Cascaded SRL32 Shift Registers using Virtex 5 SLICEM

Now that we have a feeling for LUT based shift registers, let's see how they can be combined to create a primitive, order maintaining data structure, simply by using several of them with a common clock. Such a structure is shown in Figure 6-6. Data is loaded by simply "pushing" bits into one end of the parallel shifters, and "popping" them from the other. This has the problem of a data structure initially loaded with garbage, so the first data won't be available at the other end (Q15) until the 16th clock issues, completing the first 16 entries into the structure. With some clock management, and the presence of a counter to tally the arrival of the first 16 operands, data can be found at the other end, on consecutive clock loads. This is pretty restrictive. If we load frames of 16 bit data items down the shifter columns, then extract them as we reload others, we can make use of the shift facility.

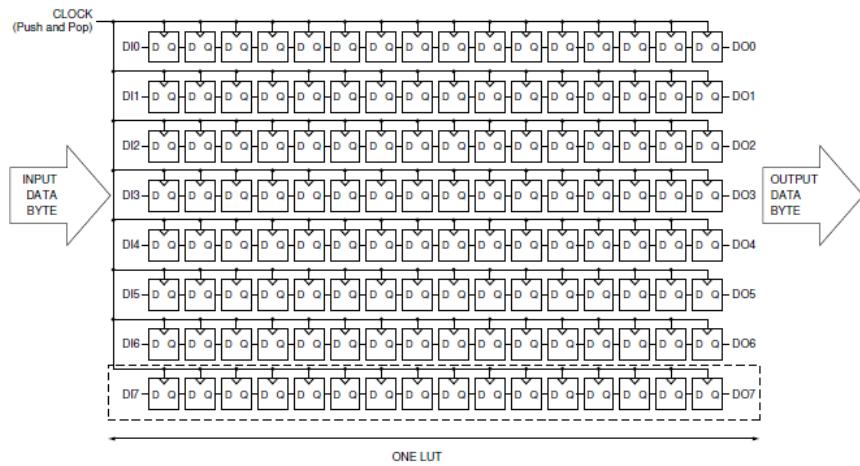


Figure 6-6 Push/Pop Shifter

This structure leaves much to be desired! A superior FIFO can be created by using the RAM capability, so let's refocus our attention on the creation of RAM from the LUT structure, permitting writing and subsequent reading back of a single bit data.

### Distributed Select RAM

When the LUT is configured to be a RAM, additional circuitry comes into play. Figure 6-1 shows that single bit data arrives at the top of the diagram (DI) as SRAM Data,

and attaches to each D flip flop, through its input multiplexer. When the appropriate address is provided, and a write pulse is present (along with the clock), a specific D flip flop is selected and data is written into it, by the only actively decoded clock signal among the set of 16. This is simple, single port writing into the 16 bit file structure. The data bit is read by examining the DATAOUT site, which becomes the effective DO line. Single port RAM capability is straightforward. Binary words are created by paralleling multiple Select RAM modules, controlled by the same address lines. For instance, a 16 byte FIFO is constructed by assembling 8 distributed Select RAM modules, attaching two four bit counters to all eight of them, and using write signals to drive data into the cells. Writing advances one address counter (pushes), while reading advances the other counter (pops), tracking the output site (oldest data). If constructed from a single port RAM, these two addresses would need to be multiplexed. A better structure, using two distributed Select RAM structures per 16 bits, can be created from the dual port capabilities supported in the distributed RAM architecture. We detail this FIFO shortly.

Dual port capability arises when the written data is broadcast to two 16 bit LUT structures, writing into the same address, on each LUT. Independent read-back occurs on one LUT, while both modules are written into the address driven on the write port address. It is termed dual port because it is possible to effectively write into one address and readback from another address, simultaneously. This is not a complete two port memory, which permits independent reading and writing of the memory using two sets of separate control, address and data signals. That capability is reserved for the larger Select Block RAMs. Figure 6-7 shows the single port and dual port structures, at a high level. The RAM 16X1S is single port memory and occupies a single LUT (ala Figure 6-1). The RAM 16X1D occupies two copies of Figure 6-1, with the D inputs tied together, so data is written jointly into both LUT structures. Figure 6-8 expands the detail for the RAM16X1D for a Spartan 3/E FPGA. The notation yX1D means substitute 16 for "y" if a single LUT, 32 if two, etc.

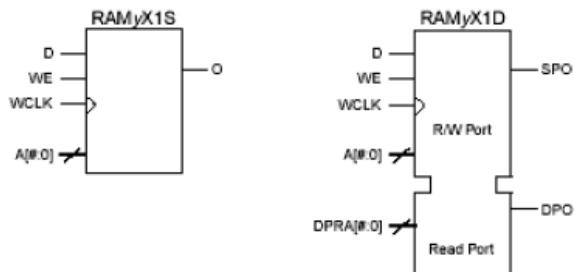


Figure 6-7 Single Port (single LUT) and Dual Port (double LUT) symbols

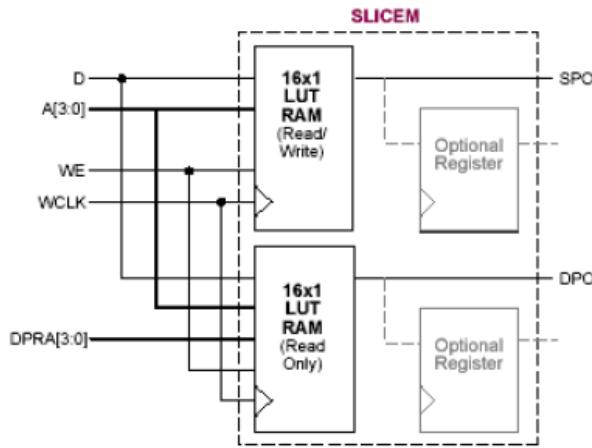


Figure 6-8 Dual port Distributed SRAM detail

Figure 6-9 shows how neighboring LUTs can be cascaded using the CLB multiplexers to make appropriate connections. In this case, the depth is increased from 16 to 32 bits. Virtex (E,EM,II) all have full Distributed RAM within each LUT, where Spartan 3/E , Virtex 4 and Virtex 5 have half of the LUTs being configurable as Distributed RAM and the other half configurable as Logic RAM. Slices (2 LUTs) configured as RAM are tagged as SLICEM, whereas logic slices are tagged as SLICEL.

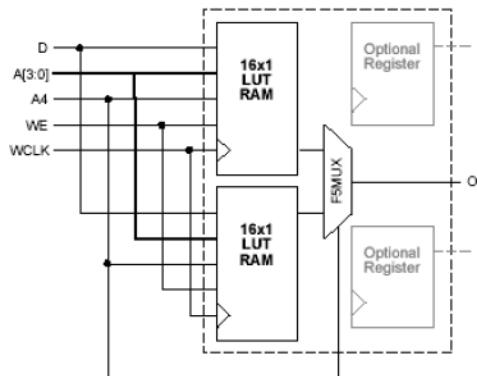


FIGURE 6-9 Cascading the LUTs to 32 bits deep

Figure 6-10 and 6-11 show the more elaborate dual port usage with the Virtex 5 SLICEM. Note that Figure 6-10 is organized as pairs of bits

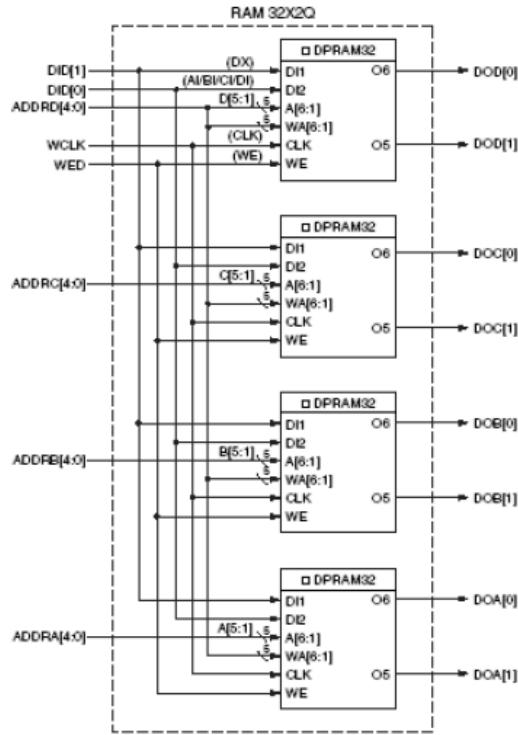


Figure 6-10 32 X 2 Dual Port LUT RAM in Virtex 5

where Figure 6-11 is organized as 32 by six bits.

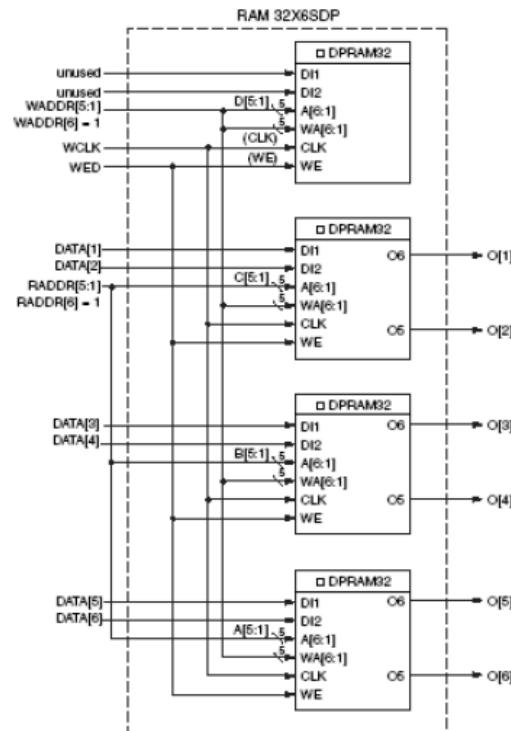


Figure 6-11 32 X 6 Dual Port LUT RAM in Virtex 5

### Synchronous FIFOs

Let's explore the FIFO on the dual port distributed Select RAM. Figure 6-12 shows a higher level of two LUT RAM structures cascaded as a dual port structure. There is a four bit "A" address and a four bit "B" address. We will have a four bit up counter attached to the A lines and another attached to the B lines. A advances when writes occur and B advances when reads occur. As long as B is greater than or equal to A, there will be valid data in the FIFO. This would be the case, until the write counter rolls over from F to 0, when any B greater than 0 would become invalid. Figure 6-13 shows the four bit counter structure.

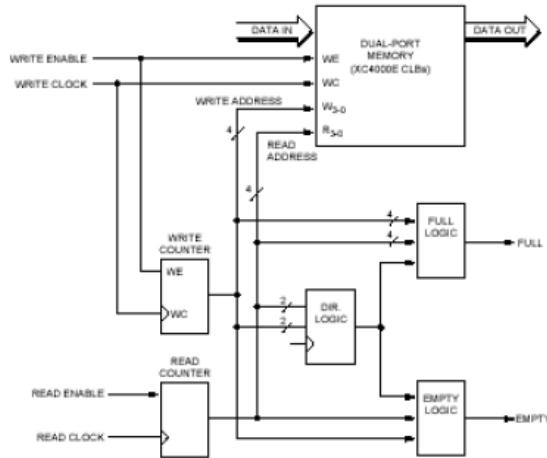


Figure 6-12 16 X 16 Dual Port Select RAM FIFO Structure

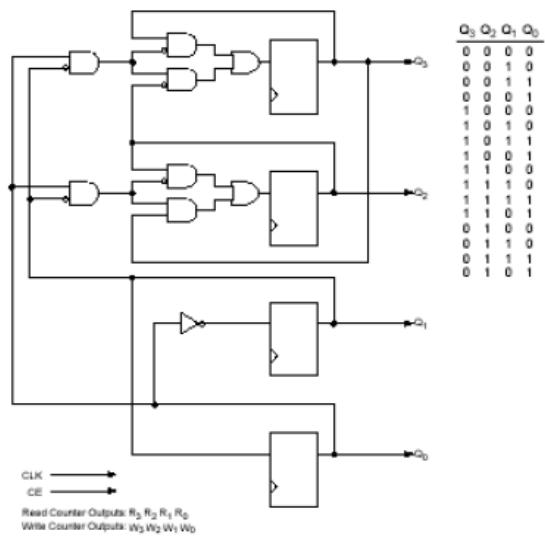


Figure 6-13 Structure for Read/Write Counters

Synchronous FIFO design is straightforward, because the tracking of the two counters is restricted to events occurring on a single clock. Asynchronous counters present the additional problem of deciding when the FIFO is empty and when it is full. Figure 6-14 presents a partial solution to the asynchronous FIFO design problem, with the

inclusion of additional state information like Full (stretched), Full (synchronized), Empty (stretched) and Empty (synchronized). Let's expand details on the asynchronous FIFO.

### Asynchronous FIFO description

Asynchronous First-In-First-Out memories are ideal system building blocks. They are user-friendly on the outside, and keep their internal complexity invisible to the user. Asynchronous FIFOs are the best approach for passing data between independent clock domains within an FPGA. Conceptually, FIFOs are simple, requiring a dual-port memory, with a write-address counter and data inputs on one port, a read-address counter and data outputs on the other port. Only the required Full/Empty handshake control is complicated, requiring careful design, especially when read and write are driven by independent clocks. Full and Empty must be derived from the identity-decoding of the read and write counters. When identical, the FIFO is either empty and forbids further reads, or it is full, forbidding further writes.

This poses two problems:

- detecting the identity of two asynchronously clocked counters
- distinguishing between Full and Empty.

FIFO binary counters should never be decoded or compared, because multiple bit-changes are inherent to the binary coding, inevitably leading to erroneous short decoding spikes. Outputs that are supposed to change simultaneously always have small delay differences. Incrementing a binary counter can create several erroneous ultra-short output codes, for example during the transition from 0111 to 1000. A well-known solution to this problem is a Gray-coded counter, where only one bit changes on any increment. Emile Baudot (1845-1903) first discovered these codes, but Frank Gray, a Bell Labs researcher, patented them in 1953. Creating a synchronous Gray counter is very easy, by starting with a binary counter and adding a Gray-code register to it. Each bit "n" of the Gray register is driven by the EX-OR of bits n and (n+1) of the binary counter. Driving the EX-OR inputs in parallel with the binary D-inputs (instead of from the more obvious binary Q outputs) avoids pipelining delays keeping binary and Gray codes in lockstep.

The two Gray counters, one for the write address and one for the read address can easily be compared for identity, never generating an erroneous decoding spike. Since the leading and trailing edges of the decoded identity signal are the result of two different and asynchronous clocks, the length of the decoded identity signal is indeterminate. The solution to this problem is described later.

The Full or Empty ambiguity of the identity signal can be resolved in various ways. One reliable method is to latch a Direction signal (Figure 6-12) whenever read and write addresses are in adjacent quadrants of their circular address space. This is determined from the two most significant counter bits. When read is one quadrant behind write, this indicates that the FIFO might get empty. When write is one quadrant behind read, this indicates that the FIFO might get full. These two conditions can easily be decoded by combining the two MSBs of both counters in two look-up tables that set or reset the Direction latch, which also must be reset (to empty) upon power-up.

The Direction output becomes a qualifier to convert the decoded identity signal into Full and Empty outputs.

The remaining problem is the asynchronous nature of Full and Empty, described here in detail for the Empty output. The Full signal is generated in the equivalent way. The leading edge of Empty is obviously caused by the read clock incrementing the read counter. Empty is only used to prevent further reads, and its leading edge is inherently synchronous in the read clock domain. The trailing edge of the decoded Empty signal, however, is caused by the write clock incrementing the write counter, and is thus asynchronous in the read clock domain.

The simplest circuit solution is to synchronize Empty to the read clock, using a two-stage synchronizer, to avoid metastability problems. Since the leading edge is inherently synchronous, and must be available as early as possible, it is used to preset the synchronizer flip-flops asynchronously. This synchronizer is the crucial, and perhaps controversial part of the design, and is therefore analyzed and explained here in more detail: the decoded Empty signal can only start soon after the active read clock edge and is terminated as a result of a later write clock edge. The duration of the decoded signal is thus unknown. If the signal is a short spike, because a write operation occurred very soon after empty was detected, there are three possible results:

1. The spike does not preset the synchronizer flip-flops.  
This is acceptable since the FIFO is no longer empty, and there is no reason to stop the following active read-clock edge from initiating a read of the newly written data.
2. The spike does preset the synchronizer flip-flops.  
This is acceptable, since it generates a synchronous Empty output that is two read-clock periods wide. This may be an overly cautious reaction, but causes no harm, only a slight performance loss after running empty.
3. The spike presets only one of the synchronizer flip-flops.  
This is also acceptable, since it generates a one-read-clock period long, perhaps unnecessary, Empty output.

If the decoded signal is as long as one or more read-clock periods, its trailing edge might coincide with an active read-clock edge. This might make the first flip-flop go metastable, but the double synchronization resolves this uncertainty. See Figure 6-14. Should the designer still worry about metastability, a third synchronizer flip-flop can be inserted to eliminate all concerns.

### Asynchronous FIFO Performance

FIFO performance is limited by the propagation delays of the Full and Empty controls. The shortest safe clock period must be longer than the sum of (Gray-counter clock-to-Q) + (Full/Empty decoder) + (asynchronous preset delay of the synchronizer flip-flop) + (set-up time of the circuit that disables the next read or write operation). With proper attention to partitioning and floorplanning, this sum can be minimized. This design does not provide any “almost Full/Empty” or “half Full” control outputs. But such

signals can easily be generated by subtracting the binary counter values, and double-synchronizing the resulting output to eliminate unavoidable decoding glitches. The resulting delay is not a performance limitation, since “partial Full/Empty” controls are not in the critical speed path.

### Asynchronous FIFO Conclusion

This section described a simple and reliable design of a FIFO memory with independent and asynchronous read and write clocks. The design is ideally suited for Virtex, Virtex-II and Spartan 3 architectures, implementing FIFOs with a depth of hundreds to tens of thousands of address locations, and widths up to 144 bits, using one or several BRAMs plus a few CLBs for counting and control. The fundamental design concept was originally published in 1996 as XAPP051, <http://www.xilinx.com/xapp/xapp051.pdf>.

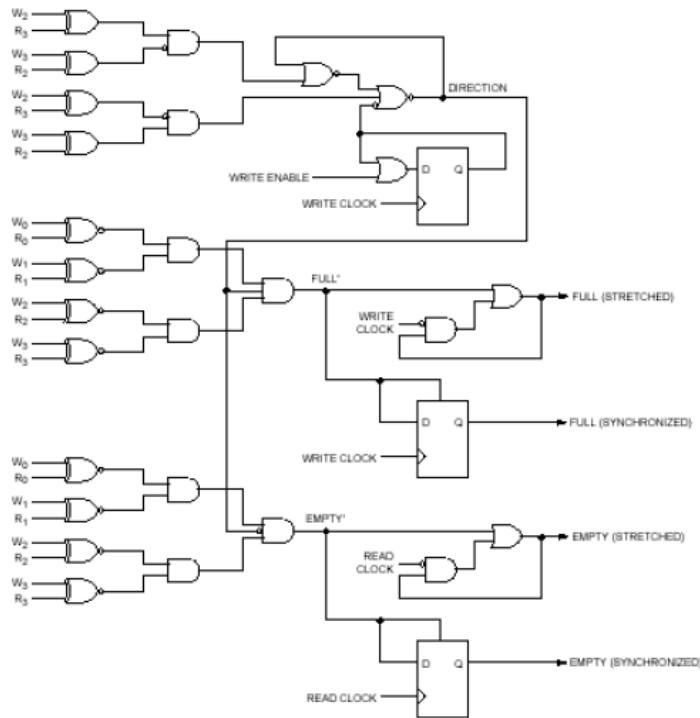


Figure 6-14 Asynchronous FIFO Control Unit

### **Block RAM**

Block RAM was introduced commercially with the original Virtex Family. However, the basic structure has evolved over time (starting even before the Virtex family), with subsequent architectures bringing additional organizations for customers to use. Table 6-1 summarizes the various possible configurations across Virtex families and corresponding process technology. This is important, because shrinking CMOS features efficiently increases BRAM density.

Family	Process	32 K	16 K	8 K	4096	2048	1024	512	256
Virtex	250 nm				X 1	X 2	X 4	X 8	X 16
Virtex E/EM	180 nm				X 1	X 2	X 4	X 8	X 16
Virtex II	150 nm		X 1	X 2	X 4	X 9	X 18	X 36	
Virtex II Pro	130 nm		X 1	X 2	X 4	X 9	X 18	X 36	
Spartan 3/E	90 nm		X 1	X 2	X 4	X 9	X 18	X 36	
Virtex 4	90 nm		X 1	X 2	X 4	X 9	X 18	X 36	
Virtex 5	65 nm	X 1	X 2	X 4	X 8	X 9	X 18	X 36	

Table 6-1 Various Virtex Block RAM Organizations

Note: Spartan 3/E, Virtex II, Virtex 4 and Virtex 5 offer the X9, X18 and X36 options, but are addressed as if they are X8, X16 and X32 organizations. The optional extra bit is viewed as parity or a utility bit.

Before we look inside the structure of the RAM cells and see how they are built, let's look at a small SRAM structure to get a feel for organizational basics. Figure 6-15 shows the basic RAM diagram having an address port (ADDR), a data input port (DIN) and a data output port (DOUT). It also has read (RD), write (WR), clock (CLK) and enable (EN) input control signals. Assume EN is asserted, enabling the RAM in Figure 6-15. The function is straightforward. When RD is asserted, the data at the address driven by the ADDR lines appears at the DOUT port. In some cases, the read completion may be synchronized to the CLK input, or even loaded into an internal latch using the CLK input. Various RAM modules made by other manufacturers, have those sorts of characteristics. When WR is asserted (driven high), the data present on the DIN lines is written into the address currently being driven on the ADDR lines. Usually, this occurs coincidentally with the CLK signal. It is common to have an "asynchronous read" and synchronous write.

This entire set of events is contingent upon the Enable signal being also asserted during the read and write events. If EN is not asserted, then nothing happens and no data enters or exits the module. This is important, because it is possible to use the EN signal for depth expansion when building larger memories. Figure 6-16 shows expanding the "data space" by connecting two 256 X 16 modules into a 256 X 32 structure. With other connections, Figure 6-17 shows connecting two 256 X 16 modules to become a 512 X 16 module. Figure 6-17 requires an additional inverter on one stage. These are the basics of "cascading" RAM structures. Using multiplexers, it is possible to create flexible combinations of RAM modules, meeting designer's needs today. Assume EN is asserted in Figure 6-16, but ADDR8 applied to EN doubles the address space in Figure 6-17.

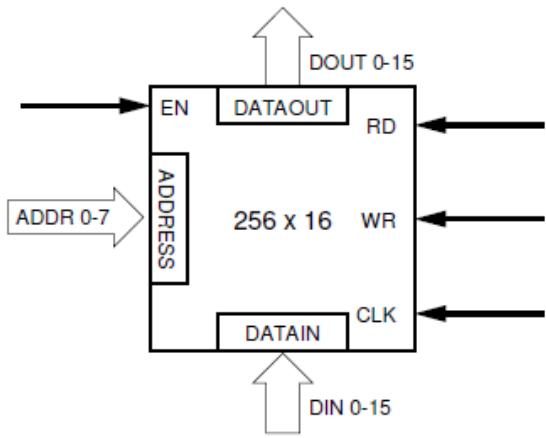


Figure 6-16 A 512 X 16 RAM

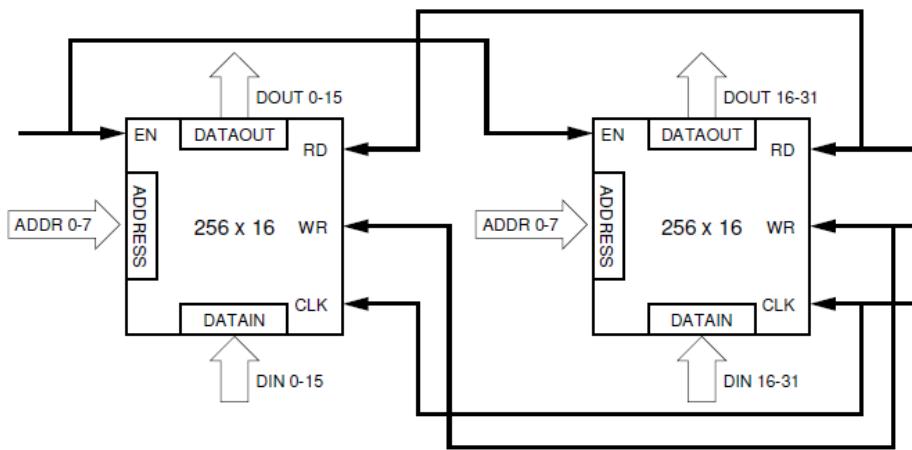


Figure 6-17 A 512 X 16 RAM module from two 256 X 16 modules

Where we have shown the various address and data paths being interconnected, it is more correct to assume multiplexers are present for both address and data path interactions. The multiplexing required to create the many combinations shown in Table 6-1 is formidable, indeed! Because of the complexity of dealing with the specific multiplexer controls, data and address lines, the Xilinx design software implements a wide range of user models for RAM. At the concrete level, these methods are simple VHDL and Verilog templates where the user dictates which address and data lines connect to other user signals. At the abstract level, it is possible for HDLs to infer the RAM structure from the sense of the design file. Often greatest performance is achieved with the more intimate, concrete description.

Now that we have a feeling for the blocks, let's look first at how the blocks are embedded within the LUT fabric, then we will delve into the deeper structures of the dual port block RAMs, themselves.

### Virtex RAM Embedding Structures

The Virtex Family was the first FPGA family to include block RAMs of the style we have been describing. As subsequent families developed, many improvements to the

RAM structure, the cascade logic, parity logic and so forth evolved. Let's look at some variations.

Figure 6-18 shows how the BRAM was envisioned to fit into the interconnect lines, so that control signals could be connected to the logic cells. Multiple kinds of "PIPs" shown as diamonds with "crossovers" and "tees" are overlaid onto the array as connection sites. With this style of interconnect, the BRAMs can be combined like Figures 6-16 and 6-17 to extend width and depth, manipulating the ENA inputs as address lines. Because customers need to create different arrangements of BRAM – like Figures 6-16 and 6-17, but with greater depths and more widths, it became necessary to add additional multiplexing into the BRAM building blocks to provide the choices shown in Table 6-1. Figure 6-19 shows one approach to how output multiplexing can be inserted into the arrangement. A reverse set of circuitry is needed in Figure 6-19 (essentially "de-multiplexing") to perform the various data writes into the memory. From a timing viewpoint, Figure 6-19 reinforces the fact that Data 0 is common to all formats, so it enters and exits every multiplexer. Hence, all formats read at the same speed as Data 0. We will see more on this with Virtex 4. Figure 6-19 more closely describes the Block RAM types that began with the Virtex II parts, extending thru Spartan 3/E and into Virtex 4. Note that the rows are comprised of four 36 bit words, and there are 128 words - hence, 18Kbits. The multiplexing into and out of the Block RAMs to handle the cases with and without parity is not trivial.

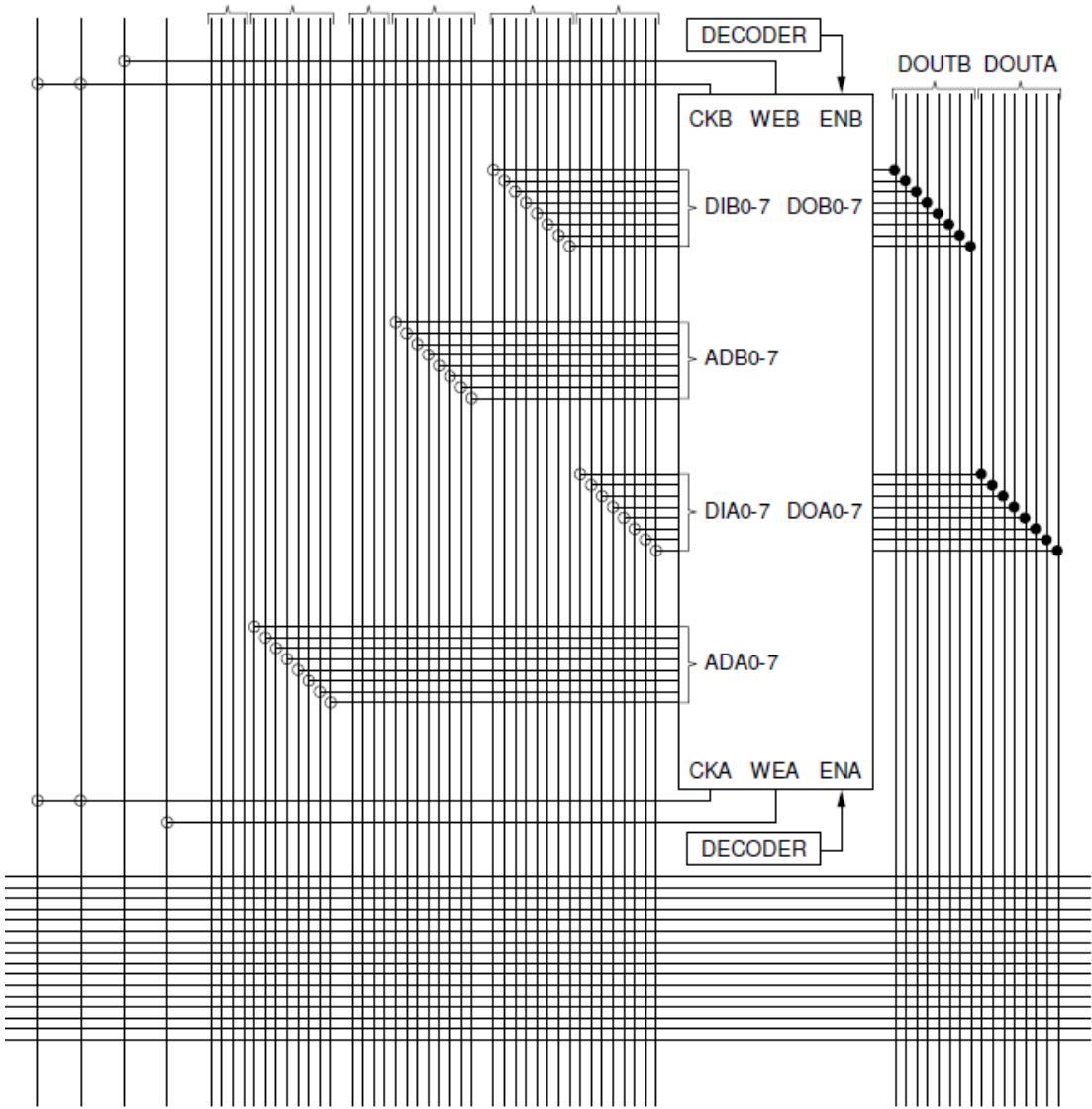


Figure 6-18 Block RAM Specialized Interconnect

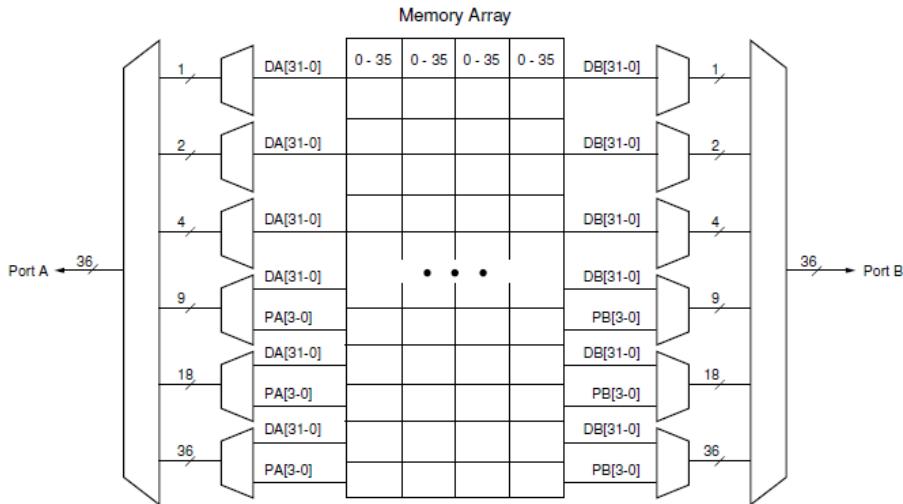


Figure 6-19 Block RAM with Multiplexing Output Structure Detailed

Figure 6-18 shows that the block interconnects with a special routing region outside the Block RAM. Figure 6-19 doesn't focus on that aspect, but the Figure 6-15 Block RAM also needs to interact with a specialized set of routing tiles permitting efficient interaction with the memory structures. Virtex II was the first Virtex architecture to incorporate 18 X 18 multipliers, and it was desired to arrange the BRAMs into close coordination with the multipliers forming extremely effective DSP calculation engines.

Clearly, operands could reside in the BRAMS, exit to adjacent multipliers, be multiplied and/or added to other operands and land back in the BRAMs for future calculations. Data never need go a distance more than a few microns of metal length, and runs very fast! More will be said on that in the arithmetic sections of this book. See Figure 6-20.

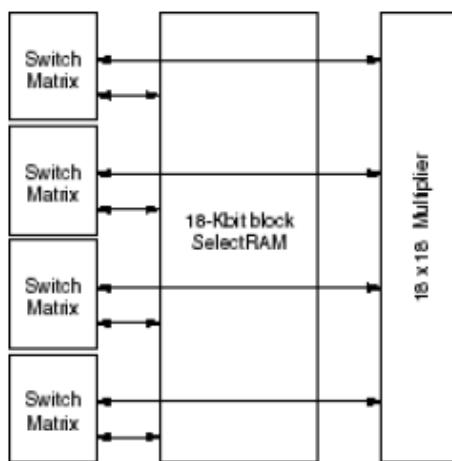


Figure 6-20 Virtex II, Spartan 3 and Virtex 4 Block RAM/Multiplier Structure

The BRAMs are conveniently sized to not violate the regularity of the routing grid. Figure 6-21 shows the arrangement adopted in the Virtex-II family, where the

introduction of a BRAM occurs between columns of two, four and six CLBs. The BRAM height is designed to track incremental CLB heights, as well. The Spartan 3/E and Virtex 4 have somewhat different BRAM arrangements, and their individual datasheets show those structures.

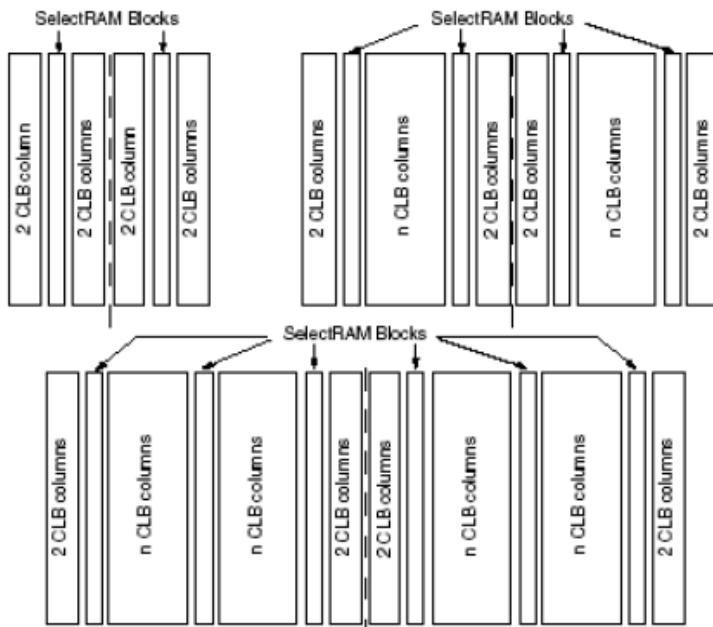


Figure 6-21 Virtex II BRAM Arrangements

### A Look Inside Single and Dual Port RAM

Now that we have seen the high level on how the BRAMs are defined, embedded and cascaded, let's reverse the scope and look into the innards of the memory cell structure. In order to provide a simplified explanation of the functionality of a single port and a dual port BRAM in general, simplified versions of the RAM circuits in the IDT Application note AN-45 by John Mick shown in Figures 6-22 through 6-25 are used. While the actual circuit structures in the Virtex series and Spartan series BRAMs differ from that shown in Figures 6-22 to 6-25 below, these figures illustrate, for educational purposes, generally how BRAMs function.

Figure 6-22 shows an SRAM that is different in arrangement from Figure 6-21. Here we have only the storage capability, and none of the SRL16 or configuration shifting. This is a 16 X 1 single port SRAM structure. A0 - A3 are the supplied address inputs, and we see four rows of four columns of storage cells. Assuming the inverters are built from CMOS transistor pairs, then the back to back inverters store the data and there are pass transistors permitting writing and reading of the data, to the target cell. This structure is also frequently called a coincident select RAM structure. Note that A0 - A1 decodes selecting the target row, and A2-A3 decodes selecting the target column. Whichever cell resides at the combined address is selected within the array.

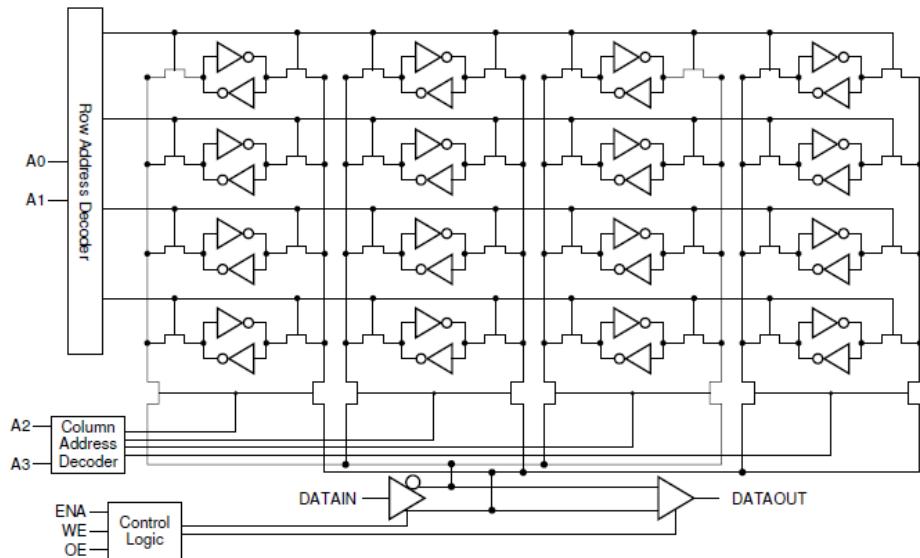


Figure 6-22 Sixteen Bit Single Port SRAM

Figures 6-23 and 6-24 help explain the read and write action. Assume a row is selected by the row decoder of Figure 6-22, producing a logic one on a particular row address (Figure 6-23). This enables the Left Row Transistor and the Right Row Transistor in Figure 6-23. On the left and right transistors, one terminal connects to the storage cell, and the other terminal connects to a pair of transistors whose gates are driven by the column address decoder (bottom left of Figure 6-22). Only one column is selected for a particular four bit address, and only one row. Where the row and column intersect, the SRAM cell at that co-ordinate is either read or written into. The Left and Right Transistors are sometimes called the access transistors.

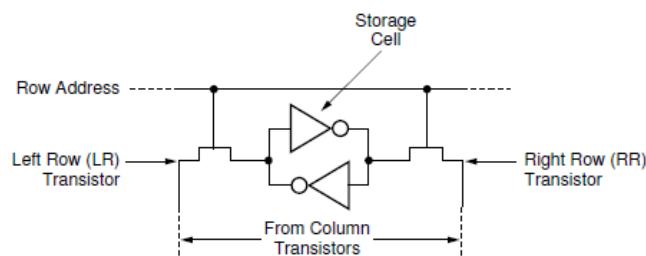


Figure 6-23 Storage Cell of a Single Port SRAM

Let's take a closer look at the row and column intersect relationship with Figure 6-24, which is a trimmed and adjusted version of Figure 6-22, removing 15 of the SRAM cells and focusing on the intersection of one row and one column, as shown with the bold row and column select lines coming from the respective decoders. If data is present on the Data In line, and the appropriate “enable” and write signals are asserted, then the Data In transfers to the two complementary outputs of the write driver. This driver is attached to the Left Column (LC) and Right Column (RC) transistors. Note that the LC and RC transistors both have their gates turned on by simply being attached to the Column select line coming from the Column Decoder. The inverted terminal of the Data In buffer drives through the LC transistor and the

positive terminal drives through the RC transistor to opposite sides of the SRAM cell, through the respective LR and RR transistors. This consistently passes data into the cell, where it stabilizes after the write enable signals vanish. In the structure shown, Data Out is sensed and forwarded out when enabled. The output driver is also a complementary amplifier, sensing both directions of the cell, through the same LR, RR, LC and CR access transistors.

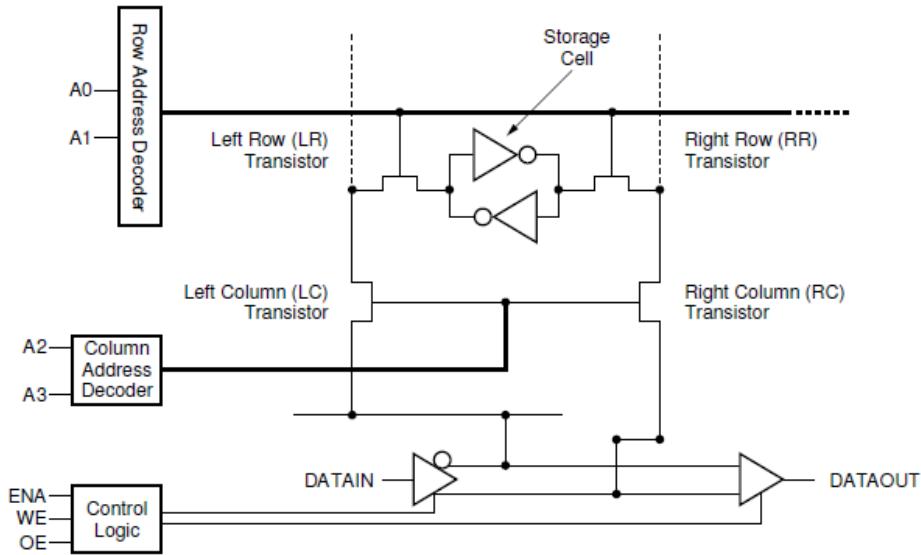


Figure 6-24 Close-up of SRAM Read/Write Circuitry

This has been a fairly lengthy and detailed discussion of a tiny, single port SRAM array structure about the storage size of a LUT! Even the earliest Virtex architecture had relatively large, dual ported BRAM, but the previous explanation vastly simplifies the workings of the dual port SRAM structure, which is now shown in Figure 6-25. Incidentally, Figure 6-25 was also inspired by IDT AN-45 by John Mick. Precisely the same number of cells is in Figure 6-25 as Figure 6-22, but an additional Read/Write access port is now present on the diagram. Duplicate sets of column and row read/write circuits and independent sets of row and column address decoders are present, as well. This complicates the operation some, but the underlying behavior is essentially the same for Figure 6-25 as it was for Figure 6-22, so we can discuss the dual port aspects with greater ease. In the dual port structure, we have an “A port” and a “B port”. This is consistent with Virtex standard naming.

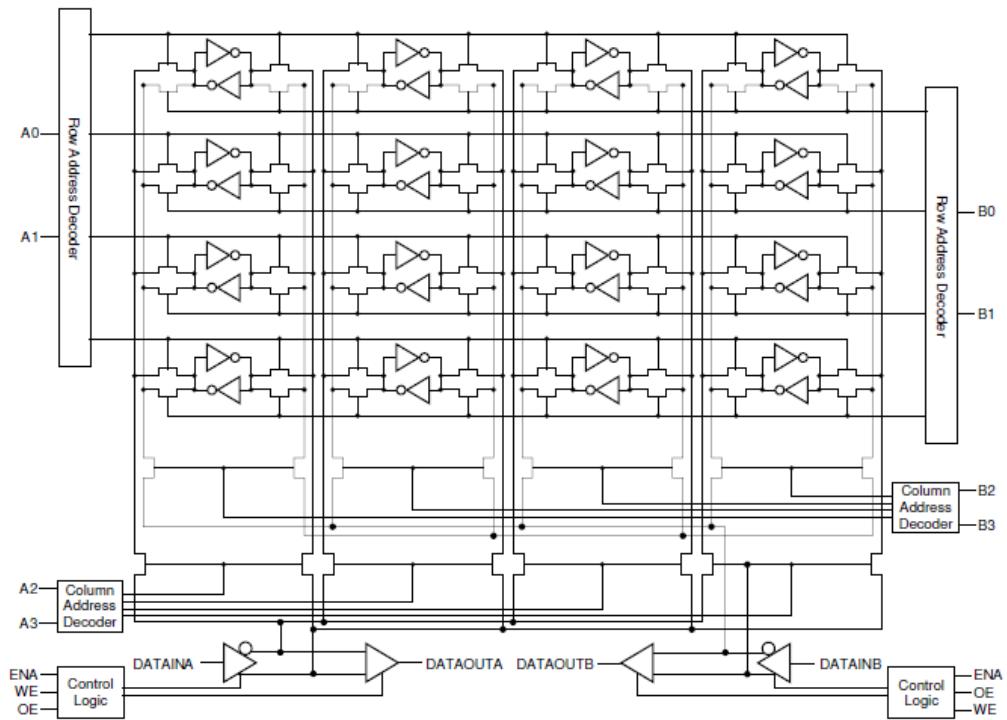


Figure 6-25 Dual Port SRAM Structure

The key differences are attributed to the duplication of overall circuitry for both ports. Note that the right and left row access transistors for both the A and B port are attached directly to the storage cell (back to back inverters). This means that the A port can select a specific row and column for reading or writing, independent of the address selected by the B port. Hence, it is possible to read and write from two different addresses at the same instant. This makes building a FIFO, quick and easy.

There is one issue – a small, but important one. With sixteen independent addresses and two sets of reads and two sets of writes, it is possible that occasionally the two addresses match and both sides attempt to write into the same address. The result is some level of signal contention, creating an unknown data state in a specific address. Should Xilinx introduce circuits that would forbid that behavior? To do so would add time delay to every BRAM transition. It would invariably introduce a solution that would please some designers and frustrate others. Hence, the Xilinx solution is to provide the dual port feature in its most flexible, fastest approach and let the individual designer introduce external logic resolving the issue in their chosen way.

Adding some form of signal arbitration is one approach to resolving this, predefining the competition “winner” is another, and simply flagging the existence of two simultaneous writes to the same address is a third. Other options exist, for designers to select among. It may also be important to remember that sixteen addresses is a tiny space, and RAM collisions would probably be substantially less frequent as the address space expands. The end application dictates the solution complexity. For instance, a BRAM used for handling text files might be more sensitive to corrupt data than a BRAM handling graphic data, where a tiny data error might go unnoticed.

## Different Port Configurations – Bit Swizzling

Virtex BRAMs can have one port configured in a different way than another. That means that the port multiplexing automatically converts data formats by simply storing into one port – say A in one format and reading back from port B in another. This capability is inherent in the flexible structures being offered, but can be invaluable in a number of different situations. For instance, simply interfacing two processors of different data widths through one of the memories (acting as a mailbox) can automatically reformat the data between them. When transmitting data in one format to a receiving engine of another, this can be a tedious, inefficient operation, where words are disassembled and reassembled to suit the needs of the receiver. Virtex BRAMs do this automatically.

## Virtex BRAM FIFO

Now that we have described the dual port nature of the BRAM, let's look at how that impacts the solution for a FIFO. As described earlier, a FIFO typically has two pointers, a Push counter recording data writes and a Pop counter recording data reads. With a dual port BRAM, Port A might be dedicated to pushing, while port B directs writing addresses. That is about it, in the simplest sense. We won't have the problem mentioned earlier about writing to the same address, as each port does an independent operation. However, in this case, we have the problem of figuring out the “edge” conditions for the FIFO. Specifically, it makes no sense to read from a FIFO that is empty, and it makes little sense to overwrite a FIFO that has filled to capacity. Figure 6-26 shows the signals made available for a 511 X 36 synchronous FIFO.

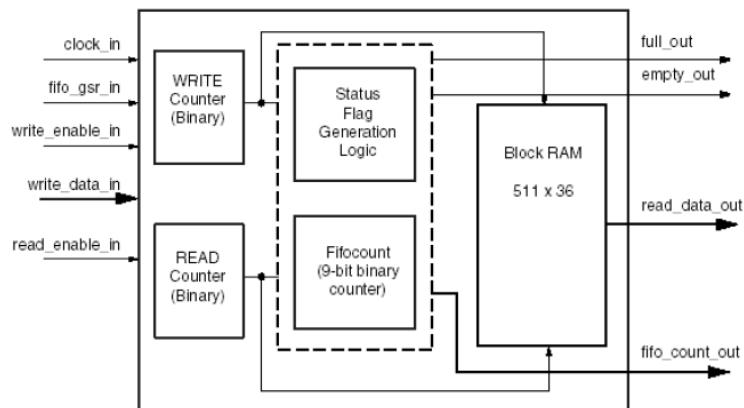


Figure 6-26 512 X 36 FIFO in Virtex BRAM

## Virtex 4 BRAM

The Virtex 4 BRAM represents the original Virtex strategy with a number of compelling refinements. Figure 6-27 shows the basic 18K BRAM and all dual signals made available for each port. In addition, several new signals are present expanding the capabilities of earlier Virtex models, while maintaining a high level of upward compatibility, so users can easily migrate their designs upward within the Virtex Families.

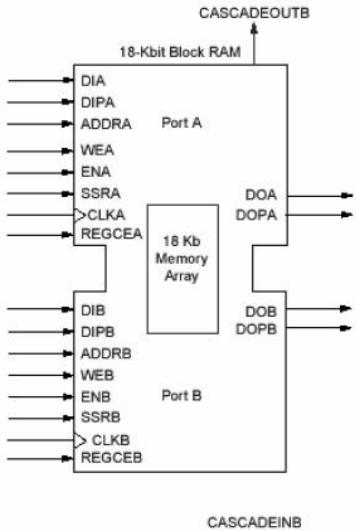


Figure 6-27 Virtex 4 BRAM Symbol

For instance, Figure 6-28 shows the presence of input address registering, and optional output data registering. Earlier Virtex, Virtex E and Virtex EM architectures did not have this structure, and made greater use of logic fabric to accomplish this. Spartan 3 introduced the output register, as well as several different modes of operation that are adopted by Virtex 4.

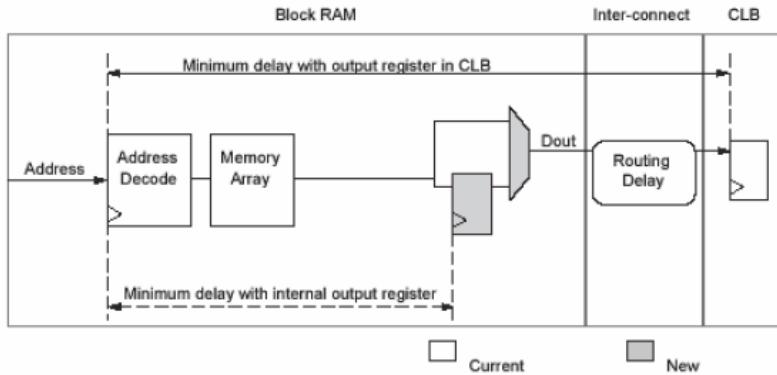


Figure 6-28 Virtex 4 BRAM Output Register

In particular, this architectural update presents choices that can be made when each BRAM port simultaneously accesses the same address. Specifically, it can be designated that a “Read First” operation will capture the current address contents into the port output register before a simultaneously arriving write clobbers the data with its new contents. It is also possible to just write on top of the data being read, or alternately to hold the data out register, throughout the write cycle preserving whatever was there last. Detailed timing relationships for all cases are shown in the Virtex 4 datasheets.

Figure 6-29 shows the addition of cascade multiplexing associated with the narrowest

BRAM configuration. This permits direct interconnection of BRAM modules within the same column to automatically attach to neighboring (north and south) BRAM cells, forming deep cascades of BRAM blocks. It avoids attaching signals into the more general interconnect grid, and permits very deep structures to be formed for single bit BRAMs up and down the inter-CLB columns. Additional data width would be created by using nearest neighbor BRAM modules to create those bits. This circuitry is intended to give the best performance and the best efficiency possible for creating deep BRAM data structures. Connection is by direct attachment of one BRAM's CASCADEOUT to the next BRAM's CASCADEIN.

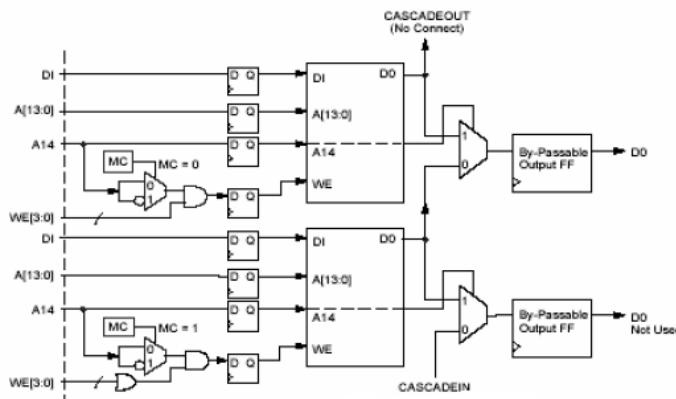


Figure 6-29 Virtex 4 BRAM Output Multiplexer/Cascade Circuitry

Other structures rely on the general interconnect fabric to form their connections.

Additional points of flexibility are that the input control signals to the BRAM can have their polarity selected. This includes clocks, enables, write enables and set/reset signals. It is also possible to designate whether the output register contains a particular value at initialization, for completeness.

As with all Virtex BRAMs, the blocks can be initialized to any binary contents the user desires in the design files. These values are subsequently forward into the BRAM configuration image that gets loaded. This capability cannot be overstressed for convenience when building sophisticated circuits using the BRAMs. For instance, if creating DSP designs in Virtex FPGA devices, it is often required to initialize large numbers of parameters and filter coefficients into BRAM before calculation can proceed. If these blocks were contained in an ASIC, or external RAM blocks, they would power up with binary garbage inside and need to be initialized before operation. With Virtex FPGA devices, this is automatically loaded into the configuration file, and then initialized using the configuration circuitry. Other approaches would require either creating logic to do this, or including microprocessor code specifically targeted to load this memory.

### Virtex 4 Synchronous FIFO Hardware

Automatic support for synchronous FIFOs is included in the Virtex 4 BRAM hardware. This is restricted to 4K X 4, 2K X 9, 1K X 18 and 512 X 36 FIFO configurations. Not

much of a restriction! The general block diagram level FIFO structure is shown in Figure 6-30. Included is the BRAM core, the read and write pointers (counters), the various error, status and control signals that can be used with whatever protocol might be desired. Asynchronous FIFO behavior can be achieved by including additional asynchronous FIFOs on each port, constructed from LUT RAM.

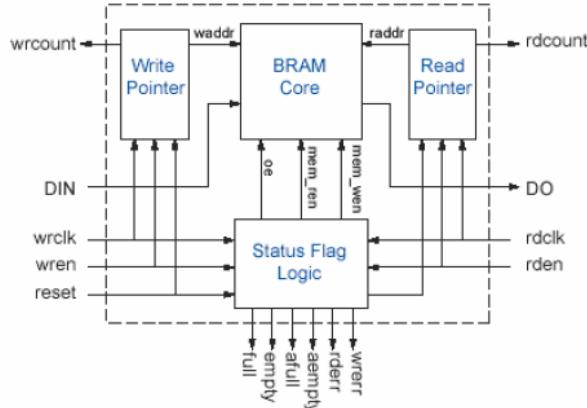


Figure 6-30 Virtex 4 FIFO Structure

The chosen configurations are even less restrictive when it is seen that designers can combine them to build customized FIFOs, as the 8K X 4 FIFO shown in Figure 6-31, using a simple NOR function and direct attachments of the various control signals.

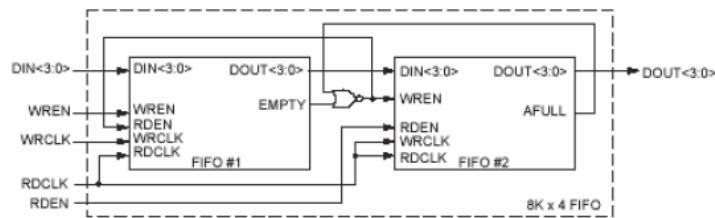


Figure 6-31 Virtex 4 BRAM Configured as 8K X 4 FIFO

Figure 6-32 shows an alternate connection of two 512 X 36 BRAMS connected to form a 512 X 72 BRAM, achieving the cascade with two AND gates, two OR gates and two inverters, built from fabric.

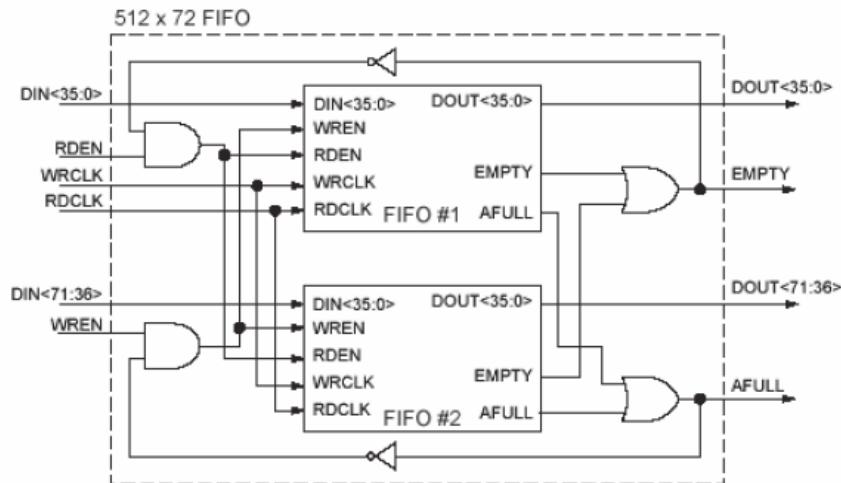


Figure 6-32 Two Virtex 4 BRAMs Configured as a 512 X 72 FIFO

### Virtex 5 BRAM, FIFO and ECC Capabilities

#### Virtex 5 BRAM

Virtex 5 offers a more elaborate BRAM than earlier models, with a basic building block being a 36Kbit dual port BRAM, depicted in Figure 6-33. This version comes with the standard two port – A and B access and controls, and easily cascades with more modules to create larger structures with minimal requirement from the FPGA fabric, as noted by the cascade signaling available. Figure 6-34 shows a Virtex 5 BRAM organized as a single module, independent read and write BRAM with 64 bit wide data, which plays well with the wider DSP48E structure in the Virtex 5 family. Figure 6-35 shows the optional capabilities of the Virtex 5 BRAM, which are similar to the Virtex 4 (Figure 6-28). Again, both designs offer fully independent dual port reading and writing, and Figure 6-35 illustrates the simplicity of cascading modules to create deeper structures.

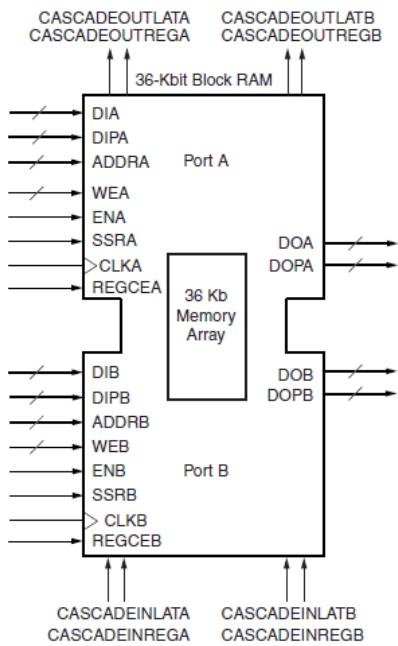


Figure 6-33 Virtex 5 Dual Port BRAM Symbol

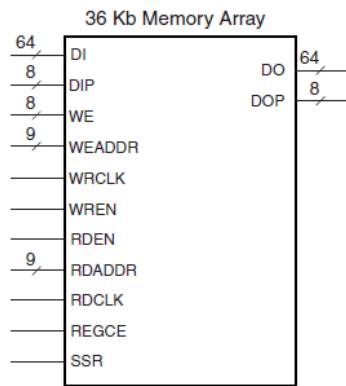


Figure 6-34 Virtex 5 BRAM Organized by 64 Bits

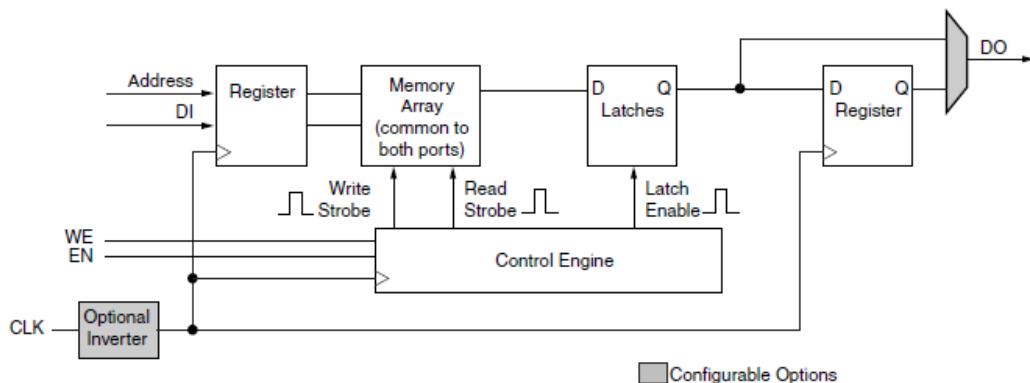


Figure 6-35 Virtex 5 BRAM Configurable Options

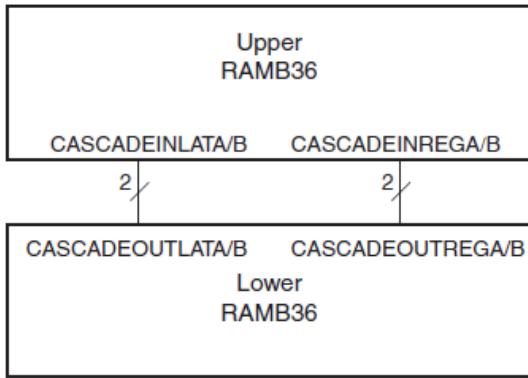


Figure 6-36 Conceptual Virtex 5 BRAM Cascade

Figure 6-37 reveals some of the underlying circuitry to accomplish the cascading ability. The fabric supplies the control signals and address/data paths, but the output ports are combined in the multiplexer, which pulls the lower data to be made available at the common switch of the upper block multiplexer before delivery to the external fabric. With this structure, multiple block cascading is possible.

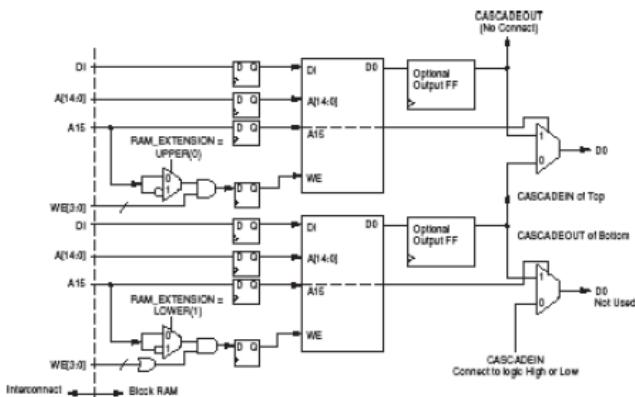


Figure 6-37 Virtex 5 BRAM Output Multiplexer/Cascade Circuitry

### Virtex 5 BRAM FIFO

Virtex 5 provides automatic FIFO creation, with special circuitry to handle the read/write pointers, control signals and status flags. It is a more complete solution than that provided by the Virtex 4 family. There are two primary operating modes – standard and “first word fall through” (FWFT). Figure 6-38 shows a read cycle in both modes. Of particular importance is the quick availability of data at the output port, when in the FWFT mode.

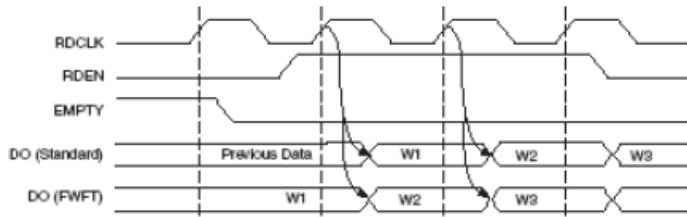


Figure 6-38 FIFO Timing with and without Fall Through

Classic status flags of “empty”, “full”, “almost empty”, “almost full”, “read error” and “write error” are manifested automatically by the controller. There is a well-defined description of their behaviors, when operating in the two modes (standard and FWFT). Cascade for greater depth is similar to Figure 6-31 and greater width is similar to Figure 6-32. The Virtex 5 FIFO makes managing data across clock domains substantially trivial.

### Virtex 5 ECC

Figure 6-39 shows the Hamming error correction paths that generate and check data transfers into the BRAM when organized as 512 by 72 bits. This situation has 64 bits of data appended with 8 bits of hamming code, getting written into the BRAM, then read back and checked for data integrity. Read errors will be flagged and presented to the fabric. Note that this configuration operates with independent read and write ports. When operating in the “full scrub” mode, the circuitry will automatically correct incorrect data as it is read from the memory. Other operations are possible and both VHDL and Verilog templates are available for use with the Xilinx design software.

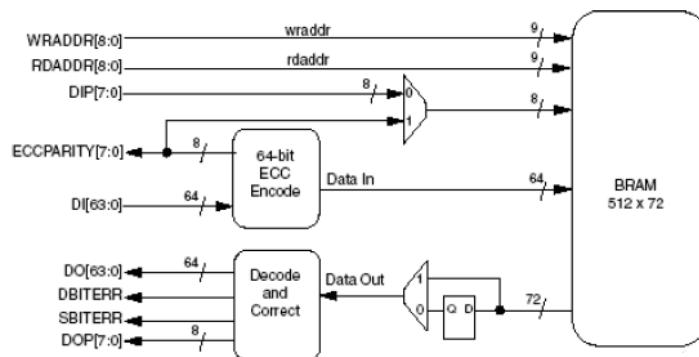


Figure 6-39 Virtex 5 64-bit ECC Capability

### Closing Comments

The complexity of the Virtex family of RAMs would make this document much longer to go any deeper. Xilinx technical documents – datasheets, application notes and white papers provide additional details needed to use the RAMs, in well described methods.

Xilinx has several application notes regarding FIFOs in FPGA devices, and the tools design these automatically. However, for an in depth discussion of asynchronous

FIFOs, see “Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons” by Clifford E. Cummings and Peter Alfke. This paper was the winning submission at SNUG 2002, and is available for download at [www.sunburst-design.com](http://www.sunburst-design.com).

## References

1. U.S. #5,933,023, FPGA Architecture Having RAM Blocks with Programmable Word Length and Width and Dedicated Address and Data Lines, by Steve Young
2. U.S. #6,346,825, Block RAM with Configurable Data Width and Parity for Use in Field Programmable Gate Array, by Raymond Pang, Steve Young, and Trevor Bauer
3. U.S. #6,297,665 B1, FPGA Architecture with Dual-Port Deep Look-Up Table RAMs, by Trevor Bauer and Steve Young
4. AN-45 “Introduction to IDT’s FourPortTM SRAM”, by John R. Mick, 1999.
5. UG190 Virtex 5 Handbook
6. 7 Series FPGAs Memory Resources User Guide, UG473 (v1.11) November 12, 2014

## Chapter 7 Virtex Arithmetic Structures

### Introduction

Binary arithmetic has been performed by logic for decades. Transitioning arithmetic to LUTs, was natural for Xilinx FPGA devices. Early LCA 2000 and LCA 3000 family architectures managed to progress in this regard. However, the LCA 4000 added two crucial items that got both Xilinx users and Xilinx designers headed toward more serious arithmetic, and eventually to high speed digital signal processing. These two items were including high speed paths for fast carries between LUTs, and modifying LUTs to become small SRAM structures. The two developments went hand in hand. Additional improvements were made in Virtex, in both fabric as well as special purpose computing blocks, as we shall see in this chapter.

High speed adding is vital for arithmetic. It is the fundamental arithmetic task. All else derives from it. But, reducing the I/O bandwidth needed to manage operands, by supplying local storage was also as important. That being the case, it is no wonder that placing multipliers and BRAM in Virtex FPGA devices triggered DSP, and made fast processing available to anyone willing to do the designs.

### Adding

Two's complement integer binary arithmetic has become the standard for arithmetic, when done at the gate and flip flop level in many programmable and ASIC environments. Hence, a large number of methods exist for doing those basic operations. A very important subcategory of these operations are developed for distributed arithmetic (DA), which will be mentioned again, later. Our focus here will be on how Virtex FPGA devices implement arithmetic operations, versus why these choices have been made by the signal processing community. Because everything derives from the adder, we start there.

The standard equations for a binary adder cell describe an incremental sum bit and a corresponding carry term, as follows:

$$S_i = A_i \text{ XOR } B_i \text{ XOR } C_i \quad \text{Equation 7-1}$$

$$C_{i+1} = A_i B_i + (A_i \text{ XOR } B_i) C_i \quad \text{Equation 7-2}$$

This is one of many formulations for an intermediate bit within a larger adder. The equation for the sum bit is simply the exclusive or each input bit (operands A and B and the carry in from the previous stage). The carry out stage for position "i" becomes the carry in stage for position "i + 1". The least significant bit will have  $C_i = 0$ , and the most significant bit will produce  $C_{i+1}$  which will be an overflow variable for the adder. Focus here is on the intermediate carries, representing the general case.

By implementing the equations directly into LUTs, each equation only uses about half a LUT, which is not particularly efficient, and the standard routing connections

may introduce intolerable additional delays. This is the “ripple adder” implementation, which, when implemented in straight logic would take about  $2n + 1$  levels of gates to produce a carry out for the highest order of an  $n$ -bit adder. That’s relatively slow. See Figure 7-1.

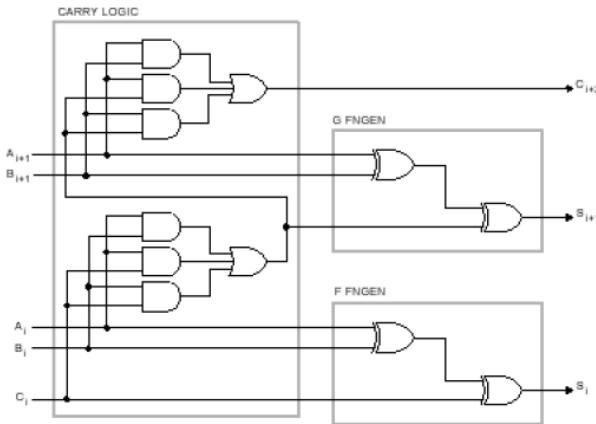


Figure 7-1 Basic Ripple Adder

### Carry Chains

In an effort to improve things, carry look ahead arrangements were investigated, but the circuitry to perform the logic was prohibitive. Of course, that changed with Virtex 5, but in the meantime, a tradeoff was sought by Xilinx architects and designers. The strategy chosen was to identify slice connections and small amounts of efficient, additional circuitry that could produce very fast ripple carry circuitry that could be neatly woven into the slice and interconnect tile fabric. Table 7-1 shows a key observation. This approach was discovered with an earlier family, the XC5200 family, which pre-dates Virtex, but was found to be very effective. Early Virtex (through Virtex 4) fast carry solutions stem from this.

A	B	C	C <sub>OUT</sub>
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 7-1 Effective Carry Operation

As noted in Table 7-1, when  $A = B$ , Cout will be equal to the input variable A (or B). When A is not equal to B, then Cout equals the CIN. This suggests a simple additional XOR gate, along with a multiplexer wired into the slice at judicious sites

will improve the situation by creating a quick carry out circuit. The multiplexer input variables come from the LUT input, and CIN. This works in parallel with the LUT circuitry, producing fast results. Figure 7-2 shows the basic idea.

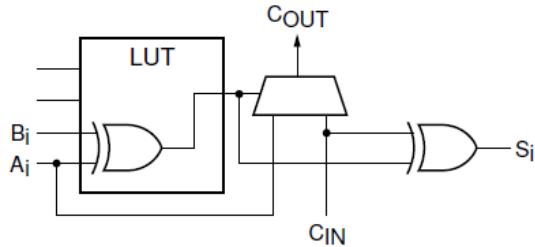


Figure 7-2 Basic Carry Chain Idea

In order to better see the circuitry, Figure 7-3 identifies the circuit region where the connections can be found and made most obvious.

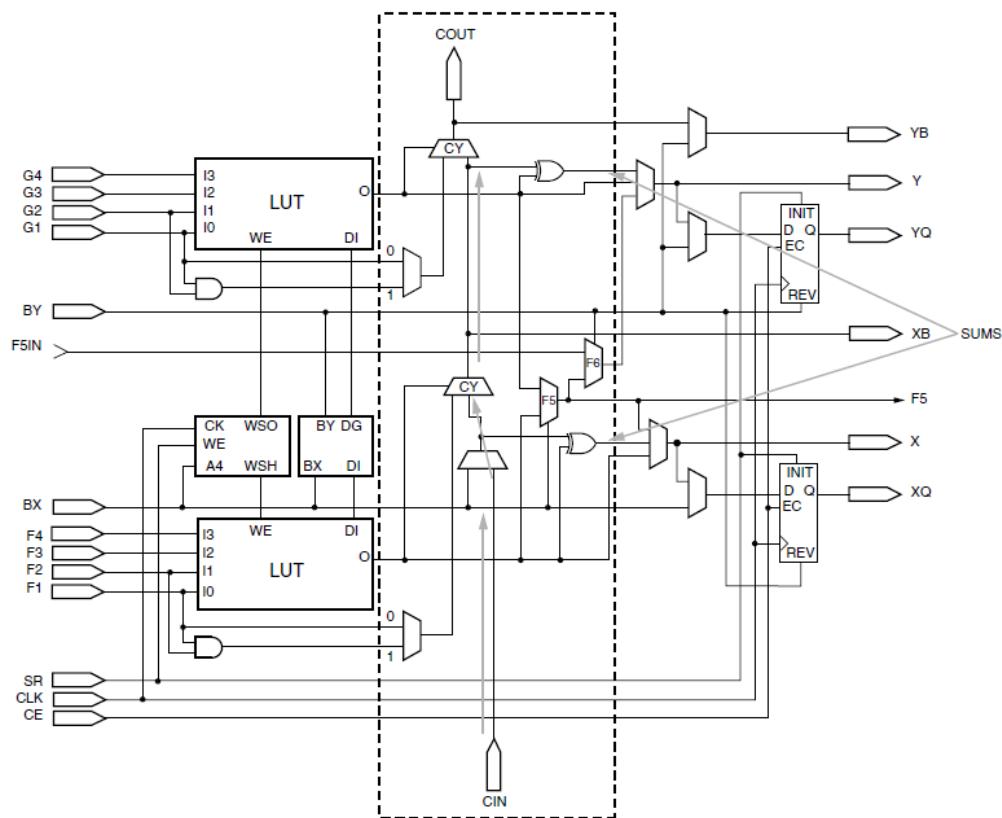


Figure 7-3 Carry Chain Exposed in Virtex Slice

The EX-OR sites shown and pointed at by arrows, expose the sites where consecutive sum bits are produced. Entry to the carry chain is at the bottom of the Slice, where Cin enters the first multiplexer, and passes up the multiplexer chain to the second multiplexer labeled CY. The signal can optionally exit the chain onto XB, or proceed upward as the input to the next logic cell's carry chain. Hence, most effective and highest performance for extended adders is obtained by placing them within columns

and adjacent to each other in bit succession order. In this configuration, remember that the LUT will contain one EX-OR function, and the second required EX-OR, to complete the sum, is an EX-OR pointed at, in Figure 7-3.

## Subtracting

Binary subtraction is basically the same as addition, where the negative representation of a number is added to another, to produce the difference. Figure 7-4 shows how such a structure is built. Be sure to note that one operand already resides in the flip flops ( $Q_0, Q_1$ ) and the other arrives from elsewhere ( $D_0, D_1$ ). In this figure, we also use the Slice AND gates to gain additional carry control. The operation is summarized in Table 7-2, where four operations are shown possible: Load the data, load the negative of the data, add the current register contents to new data, and subtract the incoming data from the current register contents. The regularity suggests exactly how to extend the functionality, in a general way.

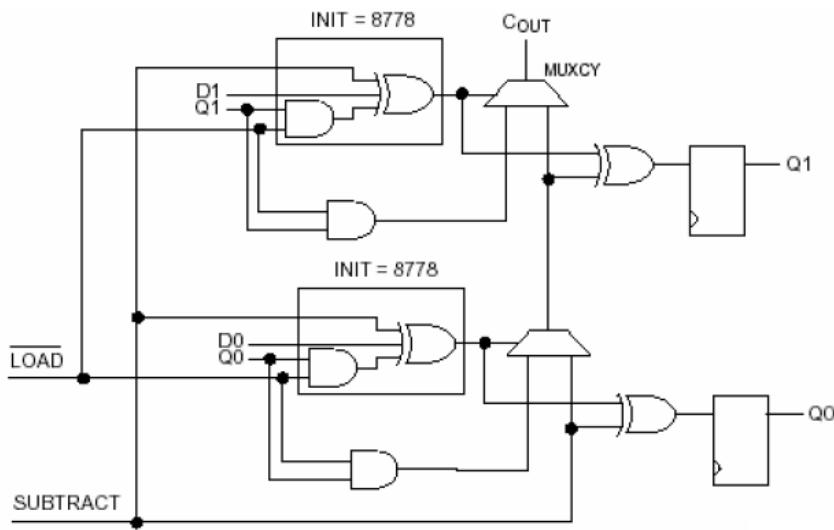


Figure 7-4 Combined Adder and Subtractor

LOAD	SUBTRACT	Function
0	0	$Q \leq D$
0	1	$Q \leq -D$
1	0	$Q \leq Q + D$
1	1	$Q \leq Q - D$

Table 7-2 Adder/ Subtractor Functionality

## Counting

Using this same basic approach, a convenient method to design high speed counters is possible. If an Adder or Subtractor has one variable set to zero, and adds the output of an accumulator to that zero operand, with the Carry in set to logic one, the result is an incrementer. By subtracting a logic one, the device becomes a decremener. Up counters can be thought of as registered incrementers, and down counters are

registered decrementers. Figure 7-5 shows such a structure, which can be cascaded to very large lengths. Note that it is loadable, cascade able, synchronous and counts Up if the Up/Down is driven to logic 1. It counts down if the Up/Down control input is driven to logic zero. The Carry chain becomes the counter cascade operation, and when properly arranged within Virtex columns, such counters can hit the speed limit of the silicon, with ease.

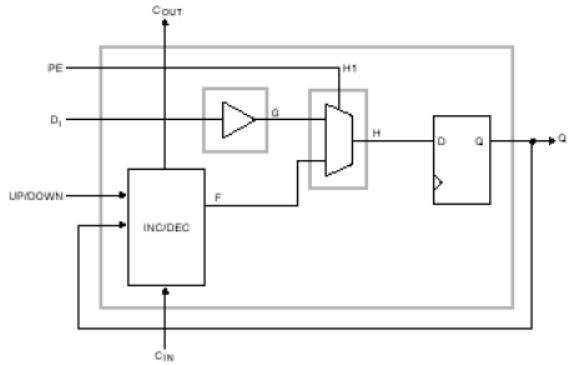


Figure 7-5 Up/Down Count Cell

Users are frequently concerned that the provided resources are designated to perform tasks that their design requirements cannot use. Hence, Xilinx designers and application engineers make every effort to provide a number of useful alternative solutions for the wide variety of common design needs. Fast counters make that list, when built using adders with fast carry chains.

### **Virtex 5 Carry Look Ahead**

While still maintaining the general flavor of the basic fast carry chain, the Virtex 5 slices allow intervening entry sites from the O5 and O6 LUT outputs to influence the COOUTs. See Figure 7-6.

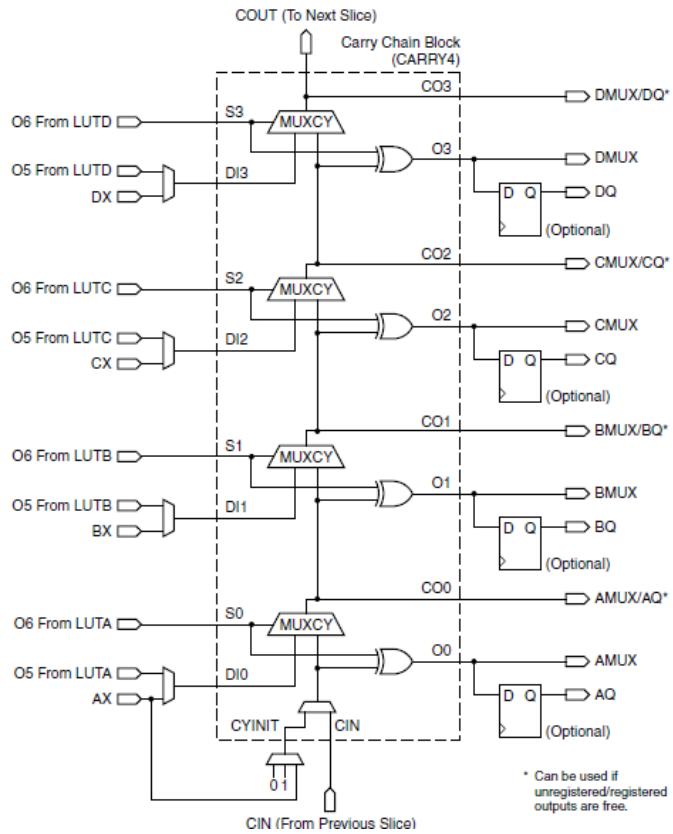


Figure 7-6 Virtex 5 Carry Look-ahead Circuitry

In this situation, it is assumed that classic “Generate” and “Propagate” functions are built inside the LUTs, with the deeper LUTs making this more efficient.

### LUT based Multipliers

Binary multiplication is a process of bit ANDing, shifting and adding. The process is described in detail in many logic design textbooks, with John Wakerly's Digital Design, Principles and Practices being one of the best, and likely the most complete. Readers unfamiliar with binary arithmetic and how it is built in hardware might do well to review the basics before proceeding. A simple binary, integer multiplication structure is shown in Figure 7-7. In this case, a four bit operand ( $A_0-A_3$ ) is multiplied by a two bit operand ( $B_0-B_1$ ), producing a six bit result. The following calculation illustrates the process:

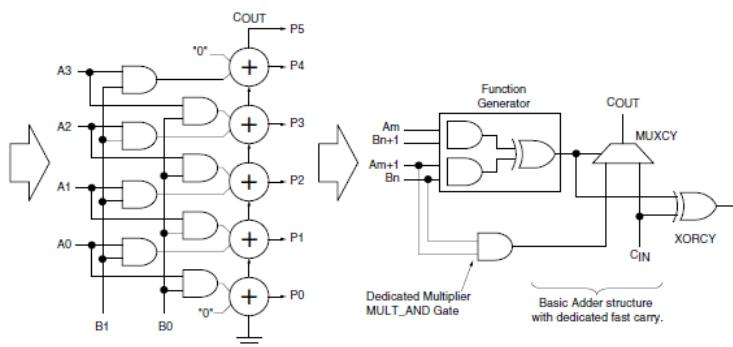


Figure 7-7 N X 2 Multiplier

Circles with pluses represent full adders, so the structure shown presents identical cells which logically AND two specific bit pairs, then add them together, to produce the output product. The least significant cell has the bottom AND omitted, and a logic zero (shown as ground) entering the Carry In site. The most significant cell shows the Carry out delivering P5, the most significant bit of the product.

The circuitry to the right of Figure 7-7 shows how the AND terms can merge right into the adder and carry circuitry in a simple way. Each logic cell within a slice produces a partial product of the overall solution. Xilinx design software offers users the choice of automatically producing multiplication structures in this way, or alternately with Block Multipliers, which we discuss shortly. Figure 7-8 isolates the circuitry in the Virtex slice that contains the functionality shown on the right hand of Figure 7-7. Similar circuitry resides within the rest of the Virtex families where Carry chain support is present. There are some variations among the family members, with Virtex being the simplest.

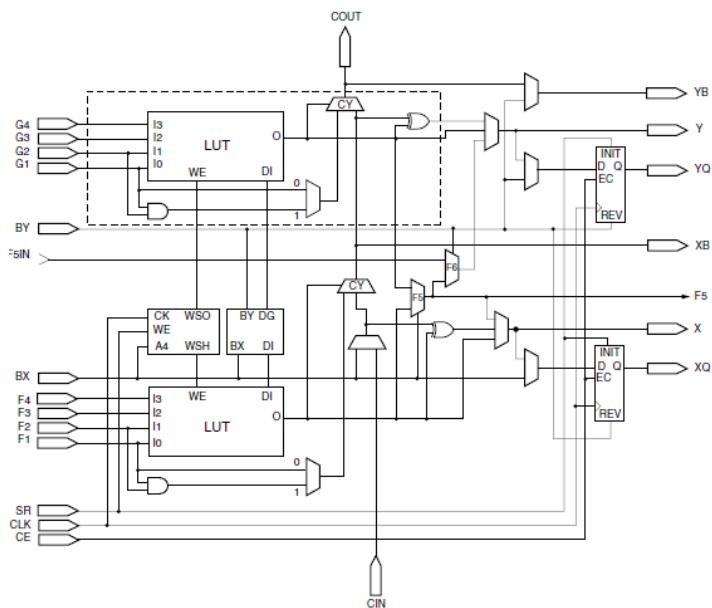


Figure 7-8 Special LUT Multiplication Support

### Lookup Table Arithmetic

As suggested earlier, it is possible to pre-compute binary functions of interest in a wide variety of applications. For instance, binary trigonometric values can be pre-calculated and loaded in an orderly way into RAM, then used as needed for say, DSP calculations. Sines, cosines, logarithms and other standard functions can be created and loaded into tables that operate very fast, at run time. Also, calculations that are “out of the ordinary”, like Galois Field calculations can be loaded into tables and used for data communication, cryptography or storage applications. That would include Viterbi encoding/decoding and polynomial based error correction codes. Switch boxes for scrambling (Sboxes) used in Triple DES or AES encryption, or Finite Field

logarithms used in elliptic curve cryptography are other interesting functions to load into tables. Choice of the RAM (BRAM or Distributed) depends on data widths, depths and which would serve best.

Some very nice work was done by Stanley White (and others) where key aspects of a computation are pre-computed, and stored in memory, to eliminate operations like real-time multiplication, altogether. These methods – distributed arithmetic - were particularly useful in the early Virtex architectures, before including multipliers. See the references at the chapter end. Less interest is directed along these lines, recently. Nonetheless, the prudent designer should blend resources, as dictated by the design, mixing fabric based, RAM based and “hard feature” based elements.

## Multiplier Blocks

Providing fixed multiplier structures is a tough architectural choice. Various user communities have differing needs. For instance, automotive or industrial applications may need only 8-10 bit operands, where digital music developers prefer 24 bits. Military radar signal processors have operated effectively with 4-6 bits. In the latter case, the extreme high speed of the signals managed is limited by how fast a converter can provide an adequate estimate of a number. Nonetheless, Xilinx designers recognized the value of creating fixed resource multipliers to satisfy the multiplication needs of users, and developed blocks to do just that.

The choice was 18 bit, two’s complement multipliers. Designers needing less could still use the 18 bit units, by zero or sign padding. Designers needing more bits achieve it by combining several multipliers with fabric based adders, and/or fabric based multipliers, as desired.

Multipliers were introduced in Virtex-II, and found their way into Virtex-II Pro/X, Spartan 3/E and Virtex 4. Naturally, they reside appropriately next to BRAMs, to provide operands and accept results at high speed. The combination works very well. Table 7-3 summarizes the two styles of Multiplier blocks, and Figure 7-9 shows the two configurations that are possible – combinational and registered.

Primitive	A Width	B Width	P Width	Signed/Unsigned	Output
MULT18X18	18	18	36	Signed (Two’s Complement)	Combinatorial
MULT18X18S	18	18	36	Signed (Two’s Complement)	Registered

Table 7-3 Multiplier Primitives

The chief differences between combinational and registered are simply using the output register that is present, which is bypassed for the combinational version. Hence, the registered version operates with additional clock (C), clock enable (CE) and reset (R) control inputs. These are directly attached to the output register.

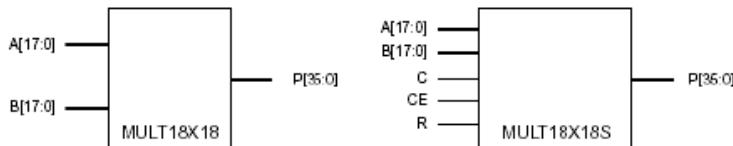


Figure 7-9 Combinational and Registered 18 bit Multipliers

Table 7-4 shows the available choices in the Spartan 3 family. The smallest part delivers four multiplier blocks, and the largest provides 104. They are arranged into specific columns, along with BRAMs, and their physical layout is such that they match evenly into integral numbers of CLBs, so they are stitched seamlessly into the logic fabric, and do not disrupt the Xilinx design software. As shown in the Memory chapter, the routing differs in the neighborhood of an inserted function block, in a way that is consistent with the standard CLB array, and still supports the needs of the function block.

Device	Multiplier Columns	Multipliers
XC3S50	1	4
XC3S200	2	12
XC3S400	2	16
XC3S1000	2	24
XC3S1500	2	32
XC3S2000	2	40
XC3S4000	4	96
XC3S5000	4	104

Table 7-4 Spartan 3 Multiplier Availability

We won't detail how two's complement multipliers work, other logic textbooks (i.e., Wakerly), cover that work, but let's look at some variations beyond simply providing two 18 bit operands and obtaining a 36 bit product. Figure 7-10 shows a combinatorial block embedded into a structure that multiplies a 22 bit operand by a 16 bit operand, producing a 38 bit result. In this example, 18 of the 22 bits of A are attached to the multiplier block along with all sixteen bits of the B operand. The same sixteen bits are shown multiplied by four of the 22 bits of A (circled X), delivering a 20 bit partial result. The final product is created by adding 16 of the bits from the lower multiplier to 34 bits of the block multiplier, giving 34 bits of the end product, and appending four bits from the bottom multiplier. The bottom multiplier could be created from a LUT based multiplier (ala Fig. 7-6). Details are found in XAPP 467 on the Xilinx website.

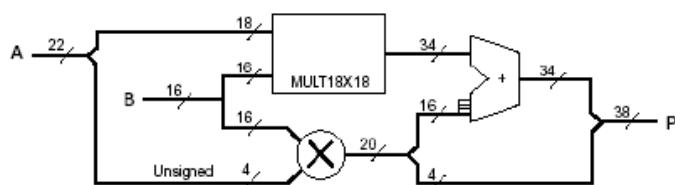


Figure 7-10 22 X 16 bit Multiplier

Figure 7-11 shows a method of expanding multiplication to accommodate 35 bit input operands, and produce 70 bit products. As shown, it combines four multiplier blocks, with pipeline stages of registers required to correctly stage the data. The output registers shown on the multipliers are actually the registers of the multipliers. Additional stages are inserted on the top two legs, to synchronize results, when the bottom two products must be added together. Additional synchronizing is needed before and after the huge bit operand adder is driven by the partial products. Lots of circuitry is used, and speed is impacted, but large multipliers are clearly possible, even in small Spartan 3 parts.

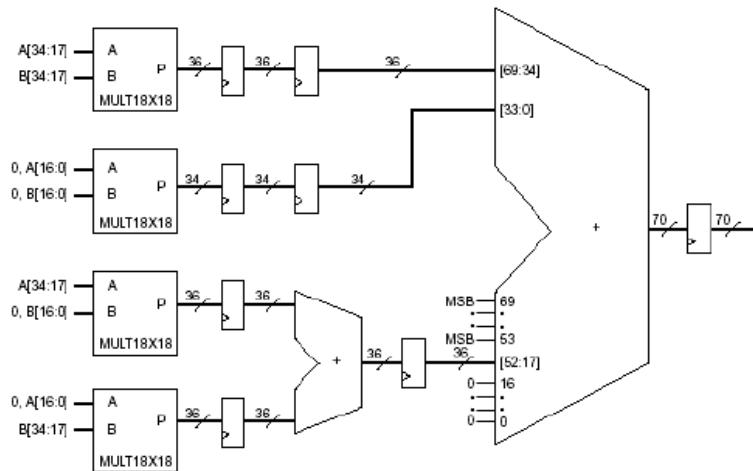


Figure 7-11 35 X 35 bit Multiplier

### Other Multiplier Uses

Users - always concerned with getting best value for their money – are interested in additional functions that can be performed with special function blocks. Just as adders can be used to build fast counters, multipliers also have additional uses. Here is a brief list:

1. Multiply several shorter numbers, with one multiplication unit
2. Perform division, by repeated multiplication
3. Shift a fairly arbitrary number of positions, also called barrel shifting
4. Return the two's complement of a binary number
5. Return the magnitude of a two's complement number

XAPP 467 outlines the solutions for these applications, and VHDL and Verilog code are available for users.

### Virtex 4 DSP 48

Multiply and accumulate (MAC) is largely, the primary task for DSP. Examining the limits of DSP calculations in silicon, reveals that interconnect and addition affect the performance of DSP calculations as well as multiplication. The Virtex 4 approach to the problem is by introducing calculation engines specifically for performing a wide range of standard DSP calculations. Those engines are called DSP 48 slices (and

tiles), shown in Figure 7-12.

The approach taken deviates from earlier methods. Rather than relying on fast carry chains for addition speed, high speed 48 bit adders, with additional cascade capability, were defined. In fact, this was done in collaboration with an outside company specializing in creating fast arithmetic structures in silicon. The multiplier is a two's complement, modified Booth multiplier, but it is “opened up” so that sign bits can propagate faster to carry decision logic, and resulting partial products merge into the final adder stage, which accommodates three input variables, rather than just two.

In Figure 7-12, multiply is toward the left and add/subtract is toward the right, as expected. One departure from earlier Xilinx FPGA symbols is that solid grey multiplexers have their input selection statically determined by the configuration memory. White multiplexers can be dynamically selected, as the design dictates. Both multiplexers are shown as trapezoids. The DSP 48 architecture is reminiscent of bit slice microprocessor components that were popular in the 1970’s - 1980’s. Defined operand ports (A,B,C) provide paths for input and output variables. Within the block are registers, multiplexers, a multiplier and an adder/subtractor. Cascading ports exist, so data can flow between adjacent slices, minimizing connection through the FPGA routing framework. Control of the operand paths and the operations to be performed are under the control of an instruction (Opmode), decoded within the slice.

An important aspect of the DSP 48 structure addresses two key requirements of arithmetic processing that can be devastating for real time DSP – overflow and rounding. Managing the accuracy of complex calculations is critical for high quality results, and Xilinx design engineers addressed both topics in user selectable ways. Details will be described later, but the choices available for C operand options, and the data widths are important aspects that help resolve overflow and rounding issues. Before looking in detail at Figure 7-12, let’s first direct our attention to Figure 7-13, a much simpler version doing the same general task.

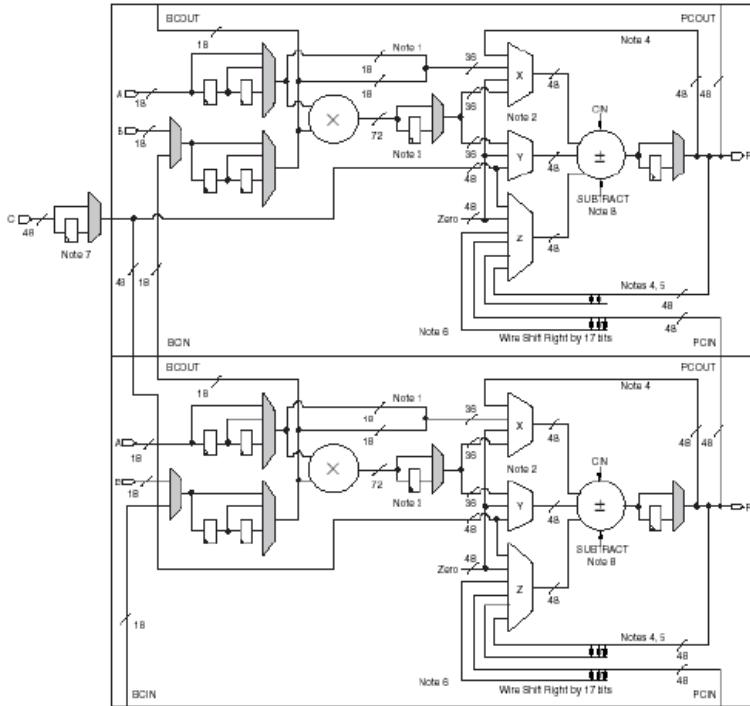


Figure 7-12 Virtex 4 DSP 48 Tile Consisting of Two DSP 48 Slices

The math portion of the DSP48 slice consists of an 18 by 18 two's complement multiplier followed by three 48-bit data path multiplexers (with outputs X, Y and Z) followed by a three input 48-bit adder/subtractor.

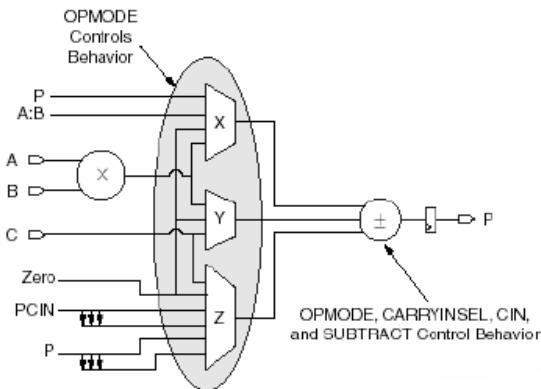


Figure 7-13 Simplified DSP 48 Slice

The data and control inputs to the DSP48 slice drive the arithmetic portions directly, or are optionally registered, once or twice, to assist the construction of different, highly pipelined DSP algorithms. The data inputs A and B can be registered once or twice. The other data and control inputs can be registered only once. Full speed operation is obtained when pipelining the registers.

The output of the adder/subtractor is a function of its inputs. The inputs are driven by

the upstream multiplexers, multiplier array and carry select logic. Equation 7-3 summarizes the combination of X, Y, Z and CIN by the adder/subtractor. The CIN, X multiplexer output and Y multiplexer output are always added together. The combined result can be selectively added to or subtracted from the Z multiplexer output.

$$\text{Adder Out} = Z +/-(X + Y + \text{CIN}) \quad \text{Equation 7-3}$$

Equation 7-4 describes a typical use where A and B are multiplied and the result is added to, or subtracted from, the C register.

$$\text{Adder Out} = C +/-(A \times B + \text{CIN}) \quad \text{Equation 7-4}$$

Selecting the multiplier function requires both X and Y multiplexer outputs to feed the adder. The two 36-bit partial products from the multiplier are sign extended to 48 bits before being sent to the adder/subtractor.

Figure 7-13 shows the seven OPMODE bits control selection of the 48-bit data paths by the three multiplexers feeding each of the three inputs to the adder/subtractor. In all cases, the 36-bit input data to the multiplexers is sign extended, forming 48-bit operands into the adder/subtractor. Based on 36-bit operands and a 48-bit accumulator output, there are 12 guard bits available to manage over flow. The number of multiply/accumulates possible before this structure overflows is  $2^{12} = 4096$ . Combinations of OPMODE, SUBTRACT, CARRYINSEL and CIN control the adder/subtractor functionality. We will look deeper into these combinations shortly.

#### Zooming In (to Figure 7-12)

Examining the 18 bit A input variable, we see it feeds a multiplexer, along with two pipeline delayed copies of the A operand. Both pipeline registers share common enables (CEA), resets (RSTA) and clocks (not shown). This pipelining provides fast access paths for quick loading, without needing to directly attach the operands from fabric, where routing might degrade performance. See Figure 7-14.

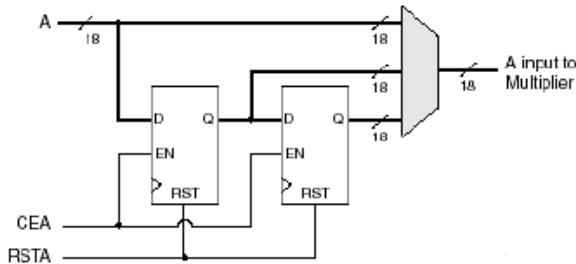


Figure 7-14 A Input Logic

Figure 7-15 shows a similar arrangement for the B operand, except that the B input bus (18 bits) is multiplexed with a cascaded B operand input bus (BCIN), permitting the chosen B variable to be both multiplied as well as forwarded out on a BCOUT bus (not shown in Figure 7-15). This special structure permits broadcasting one variable through various tiles, as chosen, without engaging FPGA fabric to form connections.

Clocks, enables and resets are independent of those for the A variable, but have the same general capabilities.

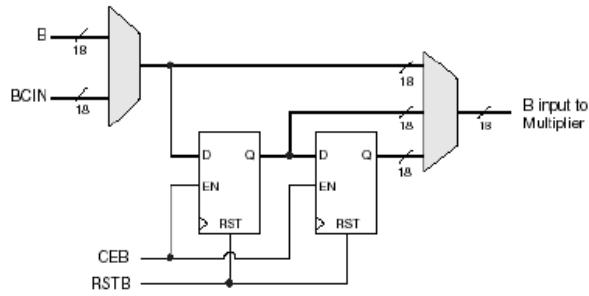


Figure 7-15 B Input Logic

Figure 7-16 shows the C variable input logic. Note that unlike A and B, it is always 48 bits wide. C also has but a single level of optional pipelining, single enable and reset, but two choices for clocking. The C variable is used as both a third operand for addition, as well as supporting rounding operations, as we shall see. An important aspect of C to remember is that it is shared across both DSP slices within the DSP48 tile. It can be used by either, neither or both slices, but holds only one value.

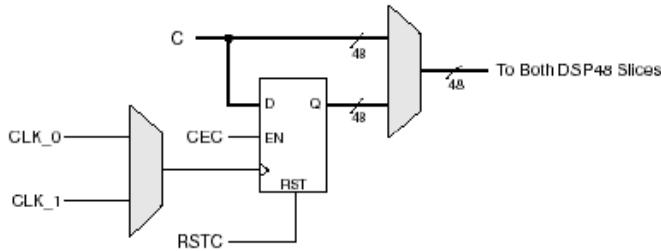


Figure 7-16 C Input Logic

Jumping over all the arithmetic logic, to the output stage, we see Figure 7-17, which takes the adder output site (P), multiplexed with its registered version, to the Slice Output site. The Slice Output feeds back, within the slice (X and Z multiplexer inputs), as well as forwarding to the other slice within the tile on the PCOUT to PCIN connections. Note that the feedback through the X multiplexer allows accumulation without multiplication, where feedback through the Z multiplexer is for accumulation with multiplication. Not shown in Figure 7-17, but evident in Figure 7-12 is the ability to take the Slice Output, truncate the low order 17 bits, and sign extend the most significant 17 bits, for alignment.

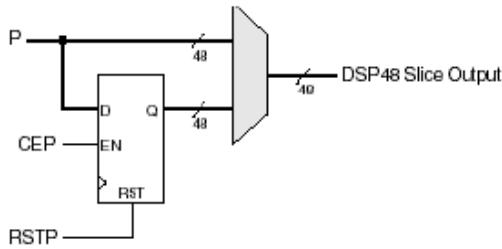


Figure 7-17 P Output Logic

Figure 7-18 shows that key control bits, which multiplex in through grey (static) multiplexers, can also provide dynamic behavior, as they are multiplexed with registered versions of the same value. That means, these important signals can be continuously changed (top input), or can be registered, to keep the same function between clock signals, for extended duration. As shown, all three functions share the same reset signal (RSTCTRL). Not evident here, is that all registers share a common clock, but the SUBTRACT register has an independent enable (CECINSUB), which works with fabric logic to customize carry input conditions.

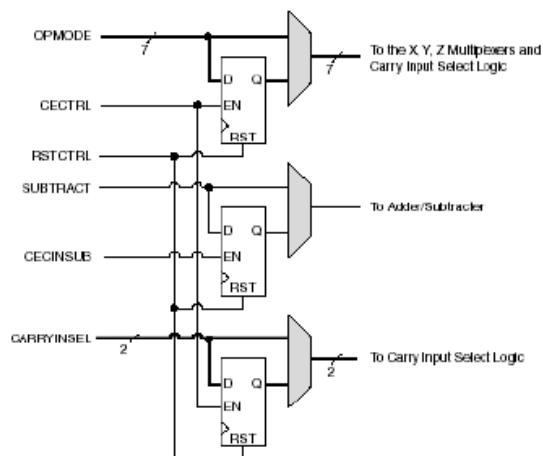


Figure 7-18 OPMODE, SUBTRACT and CARRYINSEL Port Logic

Figure 7-19 shows the two's complement multiplier. It differs from that shown in Figure 7-9, for Virtex II, in that it delivers partial results which combine in the adder stage through the X and Y multiplexers. This arrangement effectively “opens up” the multiply accumulate (MAC) operation, for speed purposes. Both Partial Product 1 and Partial Product 2 must be added together to deliver a complete product.

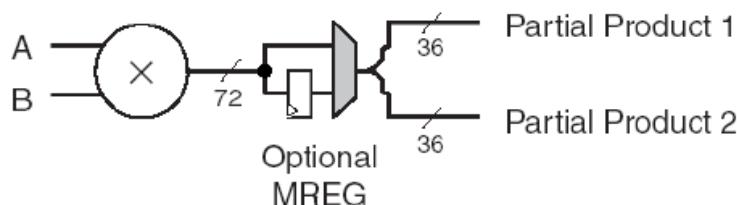


Figure 7-19 Two's Complement Multiplier with Optional MREG

Table 7-5 shows the binary value that must be applied within the OPMODE to control the X multiplexer in a DSP48 tile. Note that both DSP48 slices will get the same OPMODE. As shown, the four options are Zero, Partial Product 1, P or A concatenate B. All four X choices are defined.

OPMODE Binary			X Multiplexer Output Fed to Add/Subtract
Z	Y	X	
XXX	XX	00	ZERO (Default)
XXX	01	01	Multiplier Output (Partial Product 1)
XXX	XX	10	P
XXX	XX	11	A concatenate B

Table 7-5 OPMODE Control of X Multiplexer

Table 7-6 summarizes the OPMODE control of the Y multiplexer. Only three choices are possible, with Zero, C, and Partial Product 2 giving results. “10” is not a legal selection, and when Y has “01” on it, X must also have “01”, so both provide their respective partial products to their outputs.

OPMODE Binary			Y Multiplexer Output Fed to Add/Subtract
Z	Y	X	
XXX	00	XX	ZERO (Default)
XXX	01	01	Multiplier Output (Partial Product 2)
XXX	10	XX	Illegal selection
XXX	11	XX	C

Table 7-6 OPMODE Control of Y Multiplexer

Table 7-7 summarizes the Z multiplexer output. Various inputs shown in Figure 11 are directed to the Z output when the specific 3 bit Z field of the OPMODE is driven according to Table 7-7. Only six combinations are legal.

OPMODE Binary			Z Multiplexer Output Fed to Add/Subtract
Z	Y	X	
000	XX	XX	ZERO (Default)
001	XX	XX	PCIN
010	XX	XX	P
011	XX	XX	C
100	XX	XX	Illegal selection
101	XX	XX	Shift (PCIN)
110	XX	XX	Shift (P)
111	XX	XX	Illegal selection

Table 7-7 OPMODE Control of Z Multiplexer

Table 7-8 essentially provides the “instruction set” for the DSP48 tile. Combining the various combinations of Tables 7-5, 7-6 and 7-7 result in the operations summarized in Table 7-8. The Adder/Subtractor Output field defines exactly what occurs when sets of OPMODE bits are applied. The Hex and Binary OPMODE bits reveal the “machine language” for the DSP engine. Several of the Adder/Subtractor input combination results rely on CIN. Carry management for high speed logic can be critical, as seen earlier, with Virtex slices. Additional flexibility to handle the combination of MAC within the DSP48 is in order.

Hex OPMODE	Binary OPMODE	XYZ Multiplexer Outputs and Adder/Subtractor Output			
[6:0]	Z Y X	Z	Y	X	Adder/Subtractor Output
0x00	000 00 00	0	0	0	$\pm CIN$
0x02	000 00 10	0	0	P	$\pm(P + CIN)$
0x03	000 00 11	0	0	A:B	$\pm(A:B + CIN)$
0x05	000 01 01	0	Note 1		$\pm(A \times B + CIN)$
0x0c	000 11 00	0	C	0	$\pm(C + CIN)$
0x0e	000 11 10	0	C	P	$\pm(C + P + CIN)$
0x0f	000 11 11	0	C	A:B	$\pm(A:B + C + CIN)$
0x10	001 00 00	PCIN	0	0	$PCIN \pm CIN$
0x12	001 00 10	PCIN	0	P	$PCIN \pm (P + CIN)$
0x13	001 00 11	PCIN	0	A:B	$PCIN \pm (A:B + CIN)$
0x15	001 01 01	PCIN	Note 1		$PCIN \pm (A \times B + CIN)$
0x1c	001 11 00	PCIN	C	0	$PCIN \pm (C + CIN)$
0x1e	001 11 10	PCIN	C	P	$PCIN \pm (C + P + CIN)$
0x1f	001 11 11	PCIN	C	A:B	$PCIN \pm (A:B + C + CIN)$
0x20	010 00 00	P	0	0	$P \pm CIN$
0x22	010 00 10	P	0	P	$P \pm (P + CIN)$
0x23	010 00 11	P	0	A:B	$P \pm (A:B + CIN)$
0x25	010 01 01	P	Note 1		$P \pm (A \times B + CIN)$
0x2c	010 11 00	P	C	0	$P \pm (C + CIN)$
0x2e	010 11 10	P	C	P	$P \pm (C + P + CIN)$
0x2f	010 11 11	P	C	A:B	$P \pm (A:B + C + CIN)$
0x30	011 00 00	C	0	0	$C \pm CIN$
0x32	011 00 10	C	0	P	$C \pm (P + CIN)$
0x33	011 00 11	C	0	A:B	$C \pm (A:B + CIN)$
0x35	011 01 01	C	Note 1		$C \pm (A \times B + CIN)$
0x3c	011 11 00	C	C	0	$C \pm (C + CIN)$
0x3e	011 11 10	C	C	P	$C \pm (C + P + CIN)$
0x3f	011 11 11	C	C	A:B	$C \pm (A:B + C + CIN)$
0x50	101 00 00	Shift (PCIN)	0	0	$Shift(PCIN) \pm CIN$
0x52	101 00 10	Shift (PCIN)	0	P	$Shift(PCIN) \pm (P + CIN)$
0x53	101 00 11	Shift (PCIN)	0	A:B	$Shift(PCIN) \pm (A:B + CIN)$
0x55	101 01 01	Shift (PCIN)	Note 1		$Shift(PCIN) \pm (A \times B + CIN)$
0x5c	101 11 00	Shift (PCIN)	C	0	$Shift(PCIN) \pm (C + CIN)$
0x5e	101 11 10	Shift (PCIN)	C	P	$Shift(PCIN) \pm (C + P + CIN)$
0x5f	101 11 11	Shift (PCIN)	C	A:B	$Shift(PCIN) \pm (A:B + C + CIN)$
0x60	110 00 00	Shift (P)	0	0	$Shift(P) \pm CIN$
0x62	110 00 10	Shift (P)	0	P	$Shift(P) \pm (P + CIN)$
0x63	110 00 11	Shift (P)	0	A:B	$Shift(P) \pm (A:B + CIN)$

Table 7-8 OPMODE Control Bits and Adder/Subtractor Function

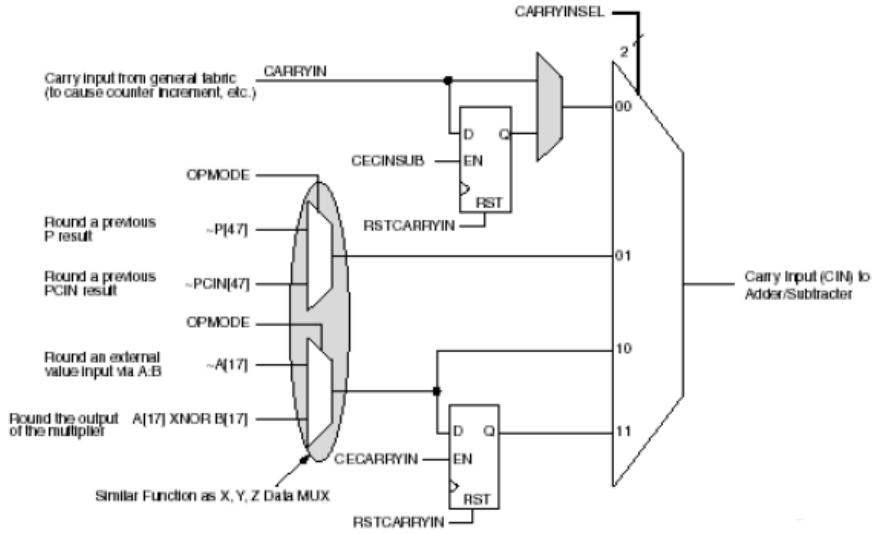


Figure 7-20 Carry Input Logic Feeding the Adder/Subtractor

Figure 7-20 shows the Carry Input (CIN) options to the Adder/Subtractor. Four options are selectable depending on the CARRYINSEL bits, which are shared across the whole DSP48 tile. The top multiplexer input, which is a carry input from the fabric, is one of generality. It is the logical choice for use, when building large fast counters with the DSP48. The top input is also the choice for designs needing design freedom, for rounding. Other choices are available using the multiplexers enclosed in the shaded oval of Figure 7-20. We pursue their use, shortly.

## Rounding

Let's briefly discuss the existence of many different rounding strategies. The simplest is truncation. Truncation discards the bits beyond a certain point within the range of bits considered to be our "word". This is usually done in the least significant bit area, having the advantage of speed. Truncation can discard significant value, and result in significantly biased results. Hence, other rounding methods evolved. Consistently rounding up or down, also produces biased results, over many calculation iterations. An interesting rounding strategy involves randomly rounding up or down, which has properties of its own.

One desirable method, for today's DSP solutions, is rounding away from zero, or symmetric rounding. Another term for this is rounding towards plus or minus infinity. In this situation, if a result is half or greater than the chosen interval, it is rounded up. If the result is less than half, it is rounded down. The overall away from zero tends to not introduce bias into the final result as much as other methods. This is where the C register enters the picture. Depending on the word size being multiplied and accumulated, an appropriate C register value accomplishes the task.

Table 7-9 gives a set of examples, of symmetric rounding under various conditions. Remember that the "binary point" is an artificial position, which designers must track

independently. Examples are given for a range of positive and negative values present on the multiplier output, which is an appropriate rounding site, as addition occurs after it.

Multiplier Output (Decimal)	Multiplier Output (Binary)	C Value	Internally Generated CIN	Multiplier Plus C Plus CIN	After Truncation (Binary)	After Truncation (Decimal)
2.4375	0010.0111	0000.0111	1	0010.1111	0010	2
2.5	0010.1000	0000.0111	1	0011.0000	0011	3
2.5625	0010.1001	0000.0111	1	0011.0001	0011	3
-2.4375	1101.1001	0000.0111	0	1110.0000	1110	-2
-2.5	1101.1000	0000.0111	0	1101.1111	1101	-3
-2.5625	1101.0111	0000.0111	0	1101.1110	1101	-3

Table 7-9 Symmetric Rounding Examples

Looking at the first row in Table 7-9, we have the value 2.4375. Based on the symmetric rounding strategy, this is less than 2.5, so it should round to 2.0. The appropriate task is to add a value from the C register (shown as 0000.0111) to the current multiplier, along with a carry in of logic 1, then truncate.

The first row:

0010.0111 multiplier output

0000.0111 Current C Register Value

1 Current Carry In

0010.1111 Intermediate Answer

0010 Symmetrically rounded, after truncation

The second row:

0010.1000 multiplier output

0000.0111 Current C Register Value

1 Current Carry In

0011.0000 Intermediate Answer

0011 Symmetrically rounded, after truncation

Third Row:

0010.1001 multiplier output

0000.0111 Current C Register Value

1 Current Carry In

0011.0001 Intermediate Answer

0011 Symmetrically rounded, after truncation

Fourth Row:

1101.1001 multiplier output

0000.0111 Current C Register Value

0 Current Carry In

1110.0000 Intermediate Answer

1110 Symmetrically rounded, after truncation

Fifth Row:

1101.1000 multiplier output  
 0000.0111 Current C Register Value  
 0 Current Carry In  
 1101.1111 Intermediate Answer  
 1101 Symmetrically rounded, after truncation

Sixth Row:

1101.0111 multiplier output  
 0000.0111 Current C Register Value  
 0 Current Carry In  
 1101.1110 Intermediate Answer  
 1101 Symmetrically rounded, after truncation

We see a pattern in these examples. In each case, the value loaded in the C register is the same (0...01..1). The exact value depends on where the binary point is chosen to be. The Carry In is related to the sign of the multiplier output. If it is negative, CIN will be zero, and if it is positive, CIN will be one. The idea here is like adding the value 0.5 if a positive fraction is half or more, and subtracting 0.5 if the fraction is less than negative half. It's convenient that the carry in, when added to 0000.0111 gives 0000.1000, which lets us load a single value into C, and simply add either zero or one, to modify its effect.

Now, getting back to Figure 7-20, we notice that the multiplexer selections in the shaded oval are making their selection based on the high order bits of various arithmetic results – the sign bits. The top multiplexer, within the oval, is selecting either the highest order bit from a previous result, or the highest order result from a cascaded input result from a neighboring slice within the tile. The lower multiplexer within the oval, selects either the high order of the concatenated A:B (i.e., A(17)), or the EXNOR of A(17) with B(17), which is the sign of the product of A times B. Recall that EXNOR produces a one when inputs match, which gives a one value, when the product of A and B will be positive. The relationship between the sign bits and the rounding process is subtle, and important. Table 7-10 summarizes the important relationships between the operand sources and the CIN multiplexer.

CARRYINSEL[1:0]	OPMODE	Carry Source	Comments
00	XXX XX XX	CARRYIN	General fabric carry source (registered or not)
01	Z MUX output = P or Shift(P)	$\sim P[47]$	Rounding P or Shift(P)
01	Z MUX output = PCIN or Shift(PCIN)	$\sim PCIN[47]$	Rounding the cascaded PCIN or Shift(PCIN) from adjacent slice
10	X and Y MUX output = multiplier partial products	$A[17] \text{xnor } B[17]$	Rounding multiplier (MREG pipeline register disabled)
11	X and Y MUX output = multiplier partial products	$A[17] \text{xnor } B[17]$	Rounding multiplier (MREG pipeline register enabled)
10	X MUX output = A:B	$\sim A[17]$	Rounding A:B (not pipelined)
11	X MUX output = A:B	$\sim A[17]$	Rounding A:B (pipelined)

Table 7-10 OPMODE and CARRYINSEL Control Carry Source

## High Level Viewpoint

Figure 7-21 ties the details discussed so far together, into the symbol that is used when instantiating a DSP48 module into a design. Clearly shown are the various external input and output sites for both data and control signals. Of particular note, are three 18 bit input buses, two 48 bit input buses, a single 18 bit output bus, and two 48 bit output buses. Lots of parallelism is evident from the symbol. An important point to remember is that these tiles can be further cascaded, to produce combined results.

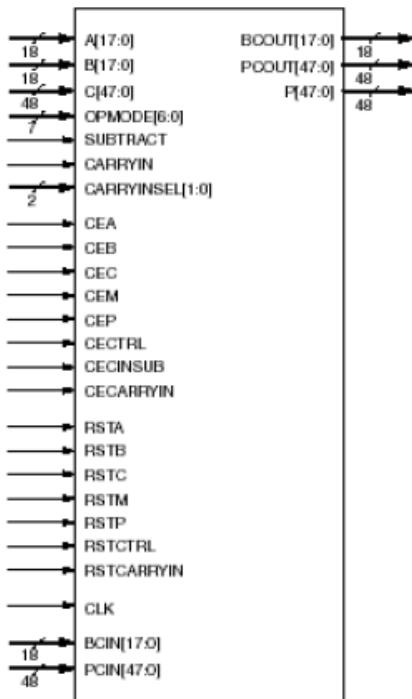


Figure 7-21 DSP48 Primitive Symbol

## Cascading Example – 35 Bit Multiplier

Building larger multipliers is also a valuable degree of freedom, similar to what was shown in Figure 7-11, earlier, when using block multipliers. The multiplier in a DSP48 is an 18 bit, two's complement multiplier. Cascading them takes some finesse, as the solution will be done in pieces, and merging them together, must maintain the integrity of the two's complement numbers being used. Figure 7-22 shows what is entailed.

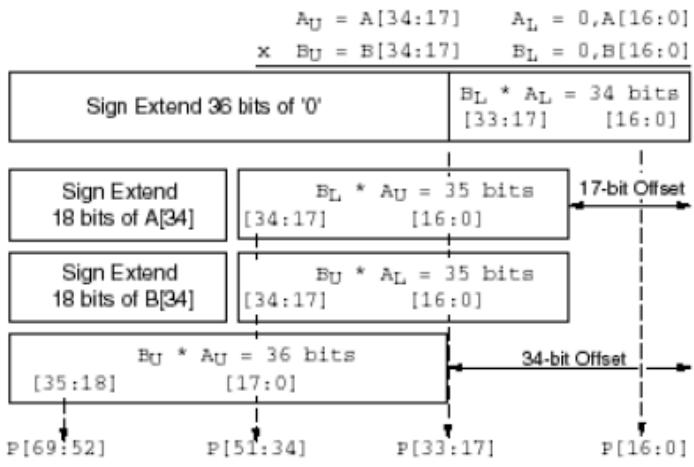


Figure 7-22 35 Bit Multiplier using Multiple DSP48 Tiles

For starters, we subdivide the input operands into lower and upper pieces. The lower A operand ( $0, A[16:0]$ ) is designated as AL as the lower B operand ( $0, B[16:0]$ ) is BL. Likewise, the upper A operand ( $A[34:17]$ ) is AU and the upper B operand ( $B[34:17]$ ) is BU. The first step is to multiply the respective lower pieces, which only carry 17 bits of payload. We have artificially inserted zeroes into the high order bits of the lower pieces, to make our two's complement multiplier supply us with a partial product that maintains the sign and magnitude integrity of the least significant partial portion of the result. This is a consequence of the fact that the multiplier is a two's complement multiplier, that assumes high order bits are sign bits, and corrects accordingly. We are forcing the multiplier to give us an answer that is consistent with the two's complement method, which makes no correction over the lower piece of the partial result. This result has 34 bits of actual value, so 36 bits of zero padding to the overall result expand that partial result to 70 bits.

The next operation is to multiply the BL times AU, to produce another partial result of 35 bits, which is left shifted by 17 bits and sign extended by 18 bits, totaling 70 bits. In this case, the overlap of the partial result extends into the region where the sign bit can be properly handled by the two's complement multiplier/adder structure. The third row of Figure 7-22 does the same operation on AL times BU, with the same shuffling of the data for sign correction and alignment. The last partial product produced multiplies AU by BU, which is 36 bits long, and is left shifted by 34 bits, to construct the 70 bit field. The final step is to add up all the aligned, shifted and extended partial products to produce the 70 bit result. This example shows why the “right shift” by 17 bits of PCIN in Figure 7-12 exists. Sign extension and right shift are part of the DSP48 tile, largely to permit creation of larger multipliers, if needed.

### DSP48 Timing

Figure 7-23 summarizes the key timing relationships for the data paths within the DSP48 slice.

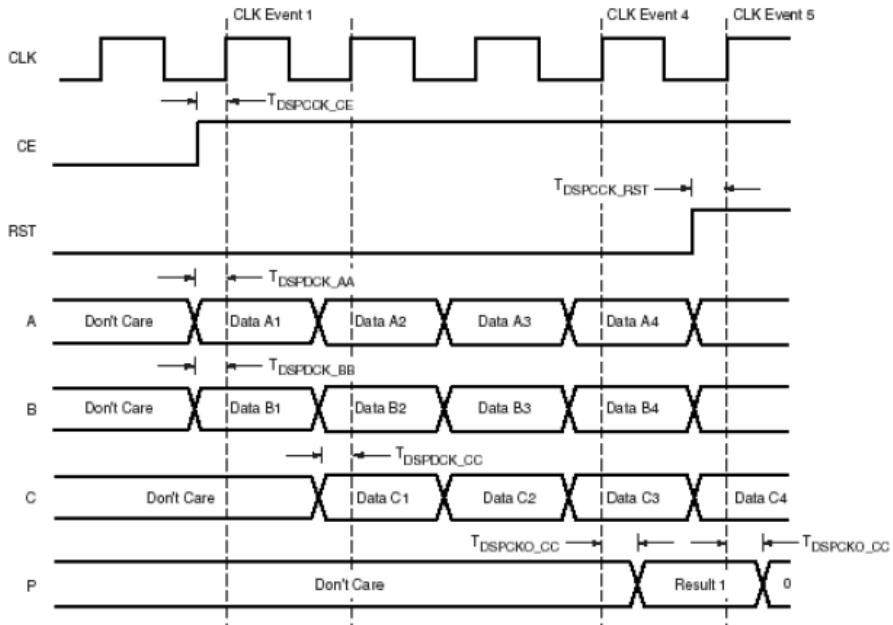


Figure 7-23 DSP 48 Timing and Timing Parameters

The clock is shared by all, but be aware that the C register can select one of two clocks. Multiple other register groups have independent clock enables, and resets, but this diagram represents a composite timing, which must be reinterpreted, if distinctly different enables and resets are used. This diagram uses both pipeline stages for the A and B registers, and the single stage for the C register. Hence, A1 and B1 are loaded on the second clock. They forward to the pipeline second stage on the third clock, as C1 is loaded. By the third clock, they are presented to the multiplier, and by the fourth clock, the partial products are combined with the original C register, producing a multiply-add ( $A \times B + C$ ), which is the result. Timing values for the various parameters shown are in the Virtex 4 Datasheet.

### Other DSP48 Applications

Now, let's look at some other things you can do with a DSP48. Clearly, adding, subtracting and multiplication are easy. Just select the right OPMODE, assign clocks, controls and data to appropriate ports and "step back." Division can be accomplished, also. There are at least two methods, repeated subtraction and repeated multiplication. Both are detailed in the Extreme DSP Design Considerations guide, on the Xilinx website. Counting is basically accomplished as mentioned earlier, by loading a value into a register and adding one to it on each clock cycle, accumulating the result in the output register.

Multiplexing can be done through the X, Y and Z registers, which are dynamic. By assigning zeroes to appropriate paths, disabling the pass through on the adder, and modifying OPMODE bits permits selecting various data fields within the DSP48. Observing Figure 7-12 and selecting appropriate OPMODE bits from Table 7-8 gives the possibilities. Remember that static multiplexers are shown shaded in gray.

Figure 7-24 shows the structure for a circuit that can calculate the square root of a number. This involves loading your operand into register C, constant of 10000000 into the A register, then executing a state machine that will converge to the square root after a number of clocks.

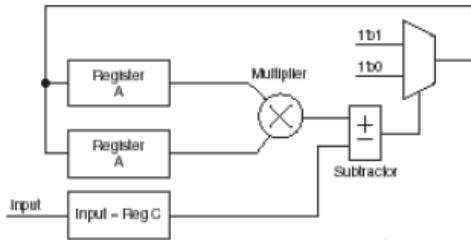
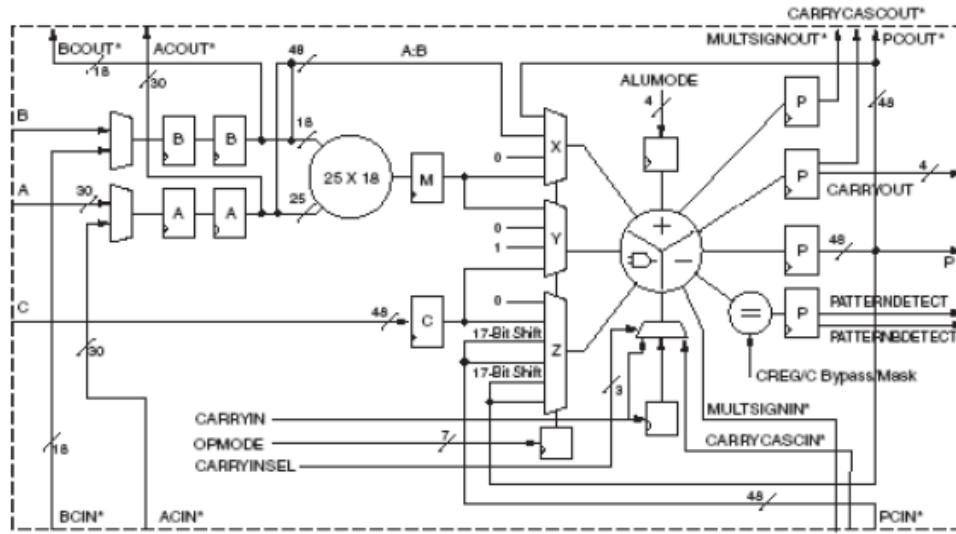


Figure 7-24 Square Root Logic

The state machine must be crafted to identify the integer and fraction portions of the answer, and the number of clocks varies for convergence, but those are execution details. Calculating a square and adding to another square, followed by a square root circuit permits calculating RMS values of signals, another common DSP task.

### Virtex 5 DSP48E Capabilities

Figure 7-25 shows a high level version of the DSP48E, which is an expanded version of half of Figure 7-12. The DSP48E builds on the ideas first introduced with the DSP48, making it even more powerful.



\*These signals are dedicated routing paths internal to the DSP48E column. They are not accessible via fabric routing resources.

Figure 7-25 Virtex 5 DSP48E Tile

Of particular note are the expansion of the A variable data width, extra registering for the C variable, more CARRYINSEL multiplexer choices, additional logic capabilities within the ALU core, and fast operand comparison. Figure 7-27 shows a simplified structure, for 48 bit operands.

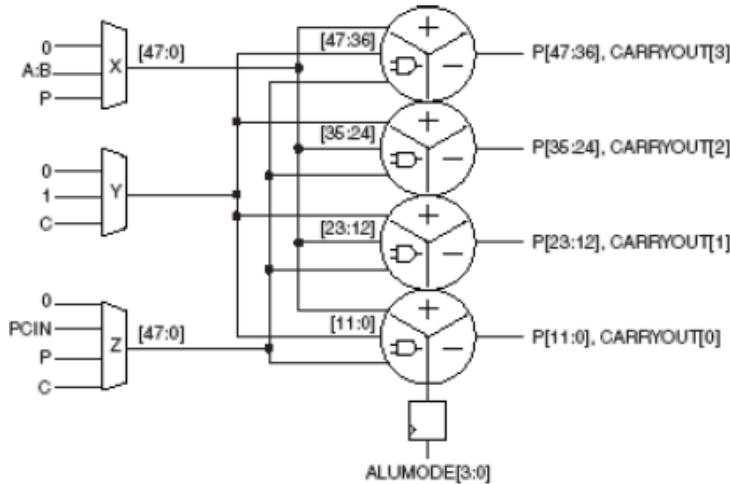


Figure 7-27 Simplified DSP48E Functional Diagram

Figure 7-28 shows a simplified diagram that focuses on pattern detection and masking of subfields, at the same time, using an externally applied mask.

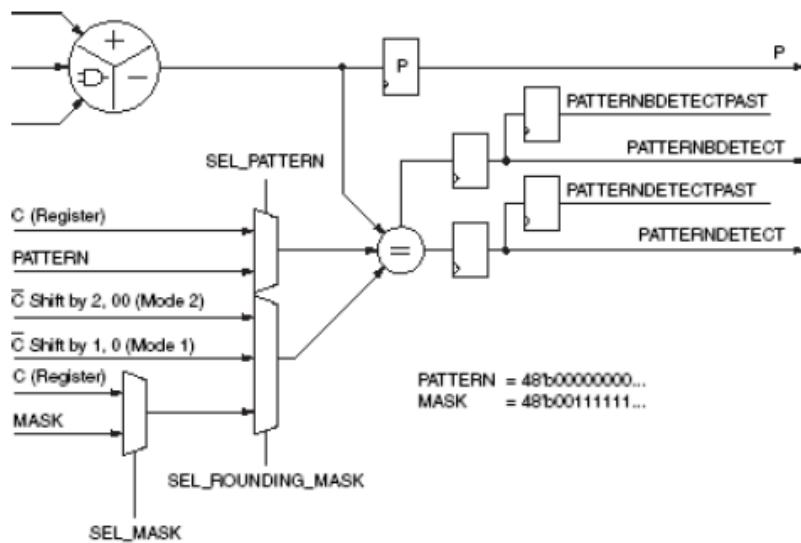


Figure 7-28 DSP48E Pattern Match Capabilities

Figure 7-29 shows how pattern detect outputs across multiple DSP48E slices can be combined to create “Overflow” and “Underflow” condition variables that help manage the accuracy of calculations.

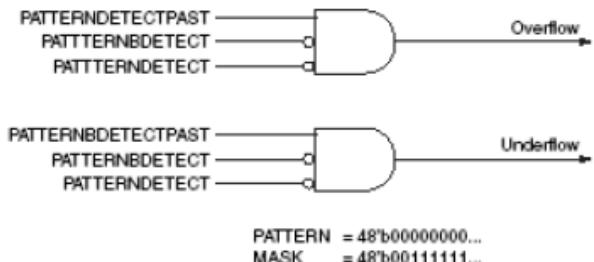


Figure 7-29 DSP48E Over and Underflow detection

Applications previously described for the DSP48 are all possible with the DSP48E. We will not show the instruction table, timing diagrams and descriptions of rounding that were done earlier, but rather make reference to the Virtex 5 User Guide, which includes an extensive detailed description of the DSP48E.

## Conclusions

Xilinx continues to improve the arithmetic capabilities of its FPGA devices, as well as the software required to implement them. Core generators for various regular arithmetic software exist to make using these features a pleasure, with certainty that the best performance and least area results. Additional detail on these structures can be found in Chapter 16 on DSP, or in the various datasheets and application notes on the Xilinx website.

## Additional References

1. XAPP013 "Using the Dedicated Carry Logic in XC4000E" details the carry chain, and those details hold for the Virtex Families, also.
2. US Patent #5,481,206 "Circuit for Fast Carry and Logic", by Bernie New and Kerry Pearce provides fast carry chain details.
3. XAPP 215 "Design Tips for HDL Implementation of Arithmetic Functions", describes use of the AND gate with the carry chain. It also includes full VHDL and Verilog listings of the various arithmetic operations.
4. US patent # 6,427,156, "Configurable Logic Bloc with AND Gate for Efficient Multiplication in FPGAs", by Ken Chapman and Steve Young gives circuit details of the AND gate used with the carry chain.
5. XAPP 467 "Using Embedded Multipliers in Spartan 3 FPGAs" details operation of the multipliers, which are very similar for all Virtex II, Spartan 3/E and Virtex 4 architectures.
6. Full operation detail of the DSP48 logic engine is provided in the XtremeDSP Design Considerations User Guide (UG073) freely available on the Xilinx website.
7. Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review, Stanley A. White, IEEE ASSP Magazine, July 1989, pp. 4-19.
8. Publications section of: [www.andraka.com](http://www.andraka.com), which is the Andraka Consulting group. Further developments on distributed arithmetic are available for download. Focus is on DSP with FPGAs.
9. Virtex 5 User Guide (UG190)
10. Accelerating Design Productivity with 7 Series FPGAs and DSP Platforms, WP406 (v1.1) February 22, 2013, Tom Hill

# Chapter 8 Virtex Timing, Models and Closure

"The software is the lens through which the user views the FPGA Architecture . . ."

- Bill Carter, Former Xilinx CTO and first FPGA designer

## Timing Overview

Basic timing principals hold for all digital logic, and Virtex FPGA devices are no different in that respect. Flip flop setup times, clock to out delays and interconnect time delay are all factors for describing the timing behavior of a design embodied in any Xilinx Virtex FPGA. Hence, the incremental time delays associated with various logic and interconnect elements all apply – as well as the timing properties of the various embedded features. Each element is best described by a timing model that defines essential behaviors and associates corresponding time delays to those behaviors.

This chapter identifies many of the timing parameters, and gives a feel for how those are used to assess the performance of a design configured into the FPGA. This is not a particularly useful endeavor of itself, but when combined with the directives available to the Place and Route (PAR) software, defines how performance requirements are dictated to the design software to achieve the end user's goals. Finally, a brief discussion on how you might approach achieving your timing goals is discussed.

## The Basics

Fundamentals account for 99% of all timing analysis, so revisiting them as a starting point, is appropriate.

### Logic Delay

Logic delay is simply the time delay to produce a stable, usable output from when an input variable to a logic element switches, until its output is valid and remains so. This covers the case, where intermediate transients might exist on a logic element output, after the inputs have stabilized. A typical logic gate that might do this could be an EX-OR gate, simultaneously switching from both inputs equal to logic zero, to logic one. In theory, there should be a constant output of zero on the gate output, but in practice, there is often a brief glitch, as one input usually switches before the other, creating the momentary output transition up, then back down. The propagation time delay is usually defined as the time delay from an input transitioning until the output first assumes its correct stable output value. Figure 8-1 shows TPD for a simple sum of products in a single LUT. TPD holds for the LUTs, multiplexers and other combinational FPGA structures, as a timing parameter type.

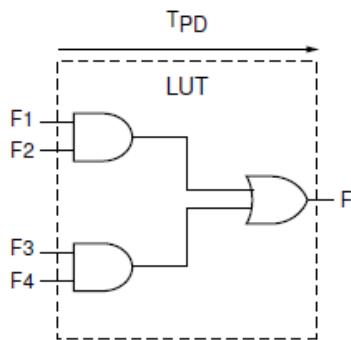


Figure 8-1 Simple Logic Propagation Time

## Flip Flop Timing

Flip Flop timing is a bit more complex. The Flip Flop itself is usually simply described by three parameters: setup time ( $T_{SU}$ ), clock to output delay ( $T_{CO}$ ) and hold time ( $T_H$ ). These are the basic properties of the flip flop, which are most frequently described by characterizing the flip flops in a laboratory. The parameters are generally specified in logic datasheets, and the understanding to gain from this is that the time values are presented as ranges, where there is typically a minimum and a maximum observed, under various conditions. Hence, when the specification is given, the most important limit is described in a datasheet, which tells the designer that the value of this parameter will never be more than (i.e., a “max”) is stated under all of the conditions described in the datasheet. Likewise, if a parameter is specified as a minimum, it will never be less than the stated value, over all conditions described in the datasheet. Now, let's look at what the parameters are.

The setup time is the required amount of time that the input excitations must be stable before the delivery of a clocking event. In Xilinx FPGA devices, this is most frequently the time that data applied to a D flip flop must be stable before a clocking event arrives, to successfully switch the flip flop. This has become a bit more complex, with Double Data Rate flip flops being typical in today's high speed memory and data communications standards. Figure 8-2 shows the simple D flip flop case.

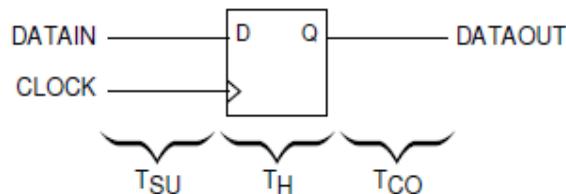


Figure 8-2 Flip Flop Timing Parameters

Hold time is the required amount of time input data must remain stable after the clocking event. In most cases zero is the desired value, to simplify logic. Clock to output ( $T_{CO}$ ) is the time from the clocking event to stable binary presence at the flip flop output. The upper limit on switching for a flip flop is its toggle frequency ( $F_{TOG}$ ).  $F_{TOG}$  is calculated as the inverse of the sum of  $T_{CO}$  and  $T_{SU}$ .  $F_{TOG}$  is an indicator of the absolute fastest a flip flop can switch, and is frequently the speed of the least significant bit of a binary counter.  $F_{MAX}$  is modified from  $F_{TOG}$  by increasing the setup delay as a sum of all appropriate additional delays (routing and logic) folded into the same type of calculation as  $F_{TOG}$ .  $F_{MAX}$  varies with the design.

## **Timing Models**

Timing models exist, to identify specific functions that can have associated time delays. Examples of this include the Slices, Distributed RAM, SRL16, BRAM, Multipliers, DCMs and MGTs. DCMs and MGTs are so complex that they will not be dealt with here, but a visit to the datasheets reveals all of the subordinate timing aspects of each function. Let's start by looking at the values split out for a Slice.

### Slice Timing

Figure 8-3 shows a simplified Virtex 4 Slice diagram which represents either SliceL or SliceM. The LUTs, flip flops and various multiplexers are present, and have representative values describing their individual time delays.

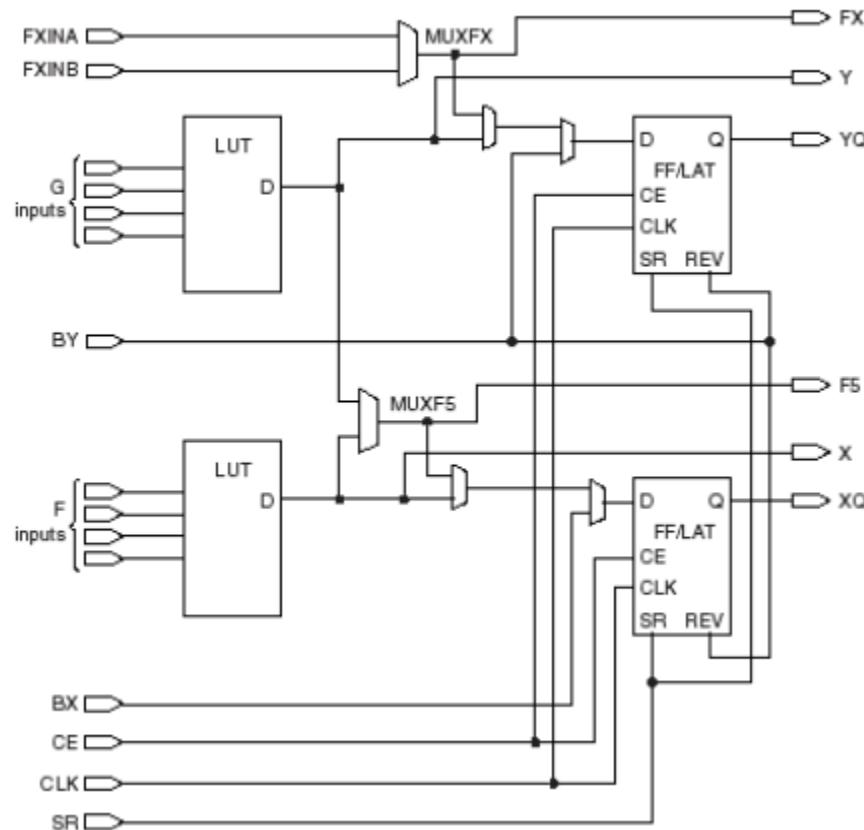


Figure 8-3 Virtex 4 SliceL/M Timing Structure

Figure 8-4 shows some basic time relationships between various signals entering/exiting the Slice timing model. Specifically described are the flip-flop timing attributes  $T_{CECK}$ ,  $T_{DICK}/T_{FXCK}$ ,  $T_{SRCK}$  and  $T_{CKO}$ .

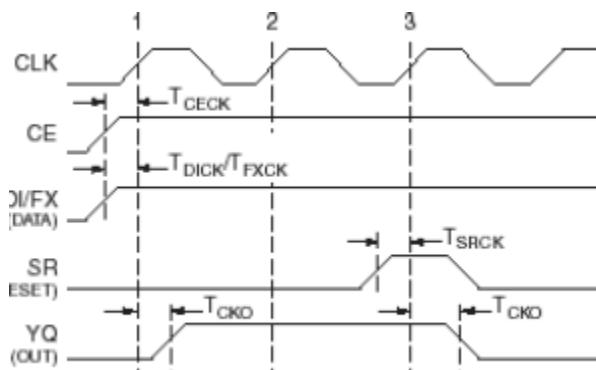


Figure 8-4 General Slice Timing Characteristics

Table 8-1 defines the time delays associated with the basic slice timing.

Parameter	Function	Description
<b>Combinatorial Delays</b>		
$T_{ILO}$	E/C inputs to X/Y outputs	Propagation delay from the E/C inputs of the slice, through the look-up tables (LUTs), to the X/Y outputs of the slice.
$T_{IF5}$	E/C inputs to F5 output	Propagation delay from the E/C inputs of the slice, through the LUTs and MUXF5 to the F5 output of the slice.
$T_{IF5X}$	E/G inputs to XMUX output	Propagation delay from the E/G inputs of the slice, through the LUTs and MUXF5 to the XMUX output of the slice.
Parameter	Function	Description
$T_{IF6Y}$	$F_{XIN\ A}/F_{XIN\ B}$ inputs to YMUX output	Propagation delay from the $F_{XIN\ A}/F_{XIN\ B}$ inputs, through F6MUX to the YMUX output of the slice.
$T_{INA/F}/T_{INB/F}$	$F_{XIN\ A}/F_{XIN\ B}$ inputs to FX output	Propagation delay from the $F_{XIN\ A}/F_{XIN\ B}$ inputs, through F6MUX to the FX output of the slice.
<b>Sequential Delays</b>		
$T_{CKO}$	FFClock (CLK) to XQ/YQ outputs	Time after the clock that data is stable at the XQ/YQ outputs of the slice sequential elements (configured as a flip-flop).
$T_{CKLO}$	Latch Clock (CLK) to XQ/YQ outputs	Time after the clock that data is stable at the XQ/YQ outputs of the slice sequential elements (configured as a latch).
<b>Setup and Hold for Slice Sequential Elements</b>		
$T_{SCK}$ = Setup time (before clock edge) $T_{HCK}$ = Hold time (after clock edge)		
$T_{DICK}/T_{CKDX}$	BX/BY Inputs	Time before Clock (CLK) that data from the BX or BY inputs of the slice must be stable at the D-input of the slice sequential elements (configured as a flip-flop).
$T_{FXDCK}/T_{CKFX}$	$F_{XIN\ A}/F_{XIN\ B}$ input	Time before Clock (CLK) that data from the $F_{XIN\ A}$ or $F_{XIN\ B}$ inputs of the slice must be stable at the D-input of the slice sequential elements (configured as a flip-flop).
$T_{CECK}/T_{CKCE}$	CE input	Time before Clock (CLK) that the CE (Clock Enable) input of the slice must be stable at the CE-input of the slice sequential elements (configured as a flip-flop).
$T_{SRCK}/T_{CKSR}$	SR/BY inputs	Time before Clock (CLK) that the SR (Set / Reset) and the BY (Rev) inputs of the slice must be stable at the SR/Rev-inputs of the slice sequential elements (configured as a flip-flop). Synchronous set/reset only.
<b>Set/Reset</b>		
TRPW		Minimum Pulse Width for the SR (Set / Reset) and BY (Rev) pins.
TRQ		Propagation delay for an asynchronous Set/Reset of the slice sequential elements. From SR/ BY inputs to XQ/ YQ outputs.
FTOG		Toggle Frequency - Maximum Frequency that a CLB flip-flop can be clocked: $1/(T_{CH}+T_{CL})$ .

Table 1 SliceL/M timing parameter summary

The single parameter that most people think of as the “Tp” for the slice is TILO. Other commonly interesting parameters from Table 8-1 are TIF5 toggle frequency). These adders indicate very basic timing limits for common operations. When building state machines, the combinational delay parameters, the flip flop timing and routing delays can dictate the design performance. If the specific design module can be captured within a single CLB and use only local connection resources, then the time delays shown in Table 8-1 will be very near the final results. Only fractional nanosecond delays will be added to connect circuits within the CLB. Designs requiring external routing require delay extraction after place and route to determine performance.

Most of the other parameters in Table 8-1 represent delays of the various multiplexers, and the flip-flops. Specifically, it is important to realize that Virtex flip flops all include a clock enable input, which was described earlier, but does include an element of time delay to drive the CE input with a logic or global resource. It becomes another powerful, but complicating element in determining the performance of a logic module.

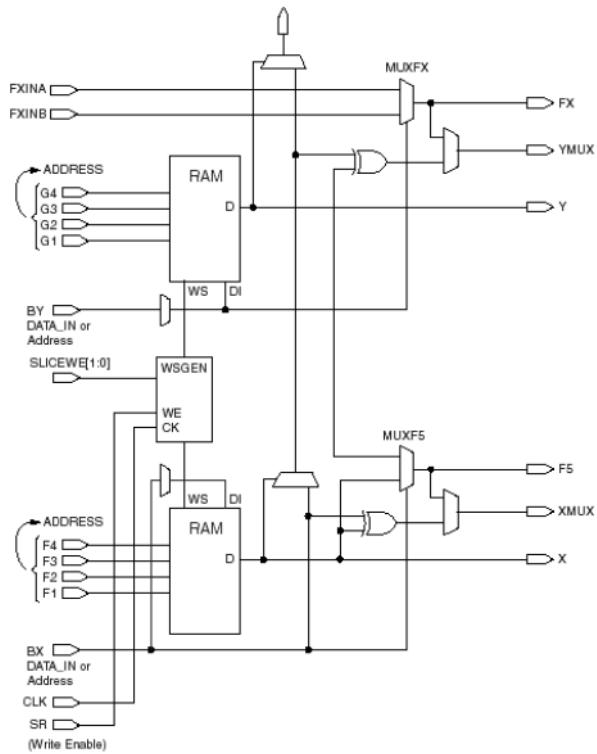


Figure 8-5 Simplified Virtex-4 SliceM Distributed RAM

Figure 8-5 shows another viewpoint of the Virtex-4 SliceM. Specifically, the F and G inputs are denoted as address inputs to the LUT RAMs, and the X and Y outputs are the LUT RAM outputs. Key time delays are from valid address to valid output for a “read”, and from DATA\_IN combined with write enable (WE) for a write operation.

Table 8-2 summarizes the various timing parameters for synchronous operation of the Distributed (LUT) RAM structure. Various entry sites (CLK, WE, BX) and output sites (X, F5, XMUX) are worked into the names of the parameters to remember which signals are being defined with respect to each other, in the table. Figure 8-6 shows the parameters of interest depicted graphically in a timing diagram that encompasses multiple storage and retrieval cycles.

Parameter	Function	Description
<b>Sequential Delays for Slice LUT Configured as RAM (Distributed RAM)</b>		
T <sub>SHCKO</sub>	CLK to X	Time after the Clock (CLK) of a Write operation that the data written to the distributed RAM is stable on the X output of the slice.
T <sub>SHCKOF5</sub>	CLK to F5 output (WE active)	Time after the Clock (CLK) of a Write operation that the data written to the distributed RAM is stable on the F5 output of the slice.
<b>Setup and Hold for Slice LUT Configured as RAM (Distributed RAM)</b>		
T <sub>S</sub> = Setup time (before clock edge) T <sub>WH</sub> = Hold time (after clock edge)		The following descriptions are for setup times only.
T <sub>DH</sub> /T <sub>DH</sub>	BX/BY configured as data input (DI)	Time before the clock that data must be stable at the BX/BY input of the slice.
T <sub>AS</sub> /T <sub>AH</sub>	F/G Address inputs	Time before the clock that address signals must be stable at the F/G inputs of the slice LUT (configured as RAM).
T <sub>WS</sub> /T <sub>WH</sub>	WE input (SR)	Time before the clock that the write enable signal must be stable at the WE input of the slice LUT (configured as RAM).
<b>Clock CLK</b>		
T <sub>WC</sub>		Minimum clock period to meet address write cycle time.

Table 8-2 Distributed RAM Timing Parameters

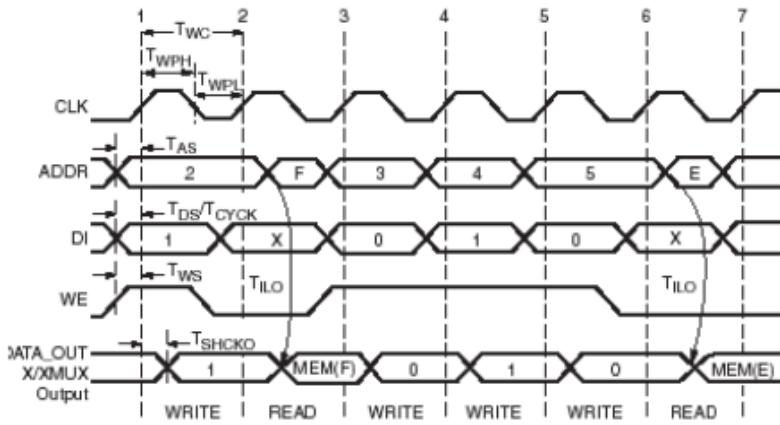


Figure 8-6 Virtex-4 Distributed RAM Timing Characteristics

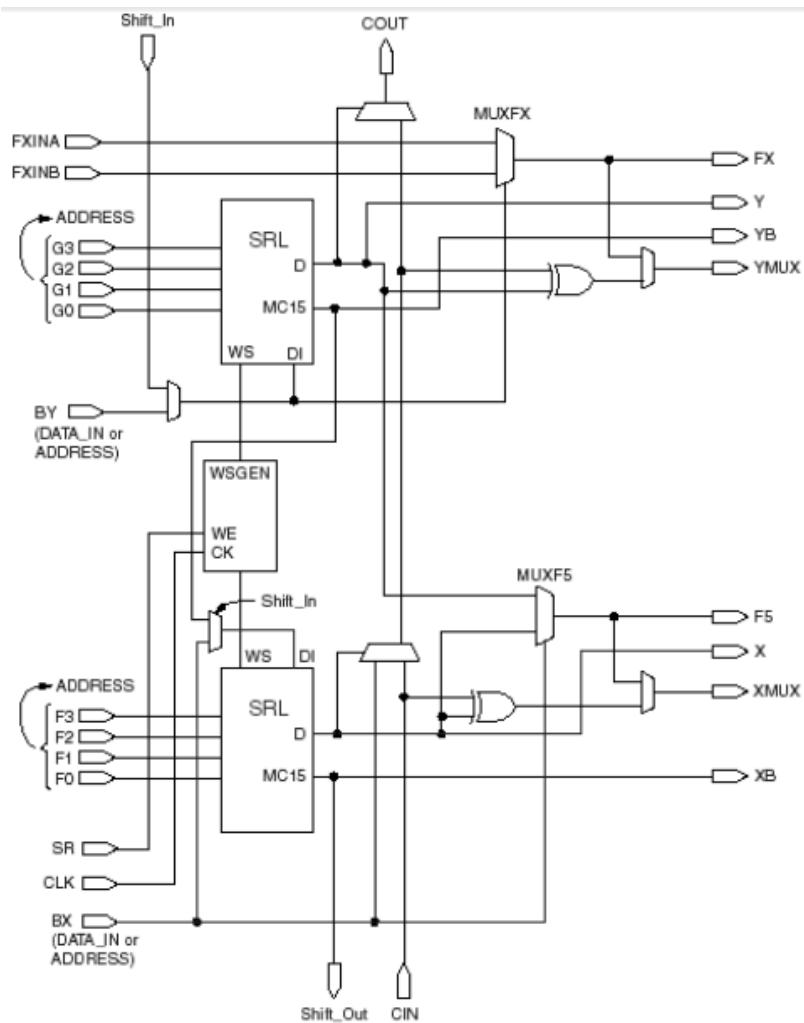


Figure 8-7 Virtex-4 SliceM SRL Timing Model

Figure 8-7 refocuses the SliceM on the shift register model, isolating those key signals that interact when creating basic SRL16 type structures. Specifically identified are the DATA\_IN, Shift\_In (multiplexer) and Shift\_Out signal sites. XB and YB are shown as the MC15 sites on both SliceM LUTs. The cascading shown also identifies how the 16 bit shifters cascade to simplistically create a 32 bit (or less) shift structure.

Parameter	Function	Description
Sequential Delays for Slice LUT Configured as SRL (Select Shift Register)		
$T_{REG}$	CLK to X/Y outputs	Time after the Clock (CLK) of a Write operation that the data written to the SRL is stable on the X/Y outputs of the slice.
$T_{CKSH}$	CLK to Shift_out	Time after the Clock (CLK) of a Write operation that the data written to the SRL is stable on the Shift_out or XB/YB outputs of the slice.
$T_{REGF5}$	CLK to F5 output	Time after the Clock (CLK) of a Write operation that the data written to the SRL is stable on the F5 output of the slice.
$T_{REGXB}/T_{REGYB}$	CLK to XB/YB outputs	Time after the Clock (CLK) of a Write operation that the data written to the SRL is stable on the XB/YB outputs of the slice.
Setup/Hold for Slice LUT Configured as SRL (Select Shift Register)		
$T_{xxS}$ = Setup time (before clock edge) $T_{xxH}$ = Hold time (after clock edge)		The following descriptions are for setup times only.
$T_{WS}/T_{WH}$	CE input (WE)	Time before the clock that the write enable signal must be stable at the WE input of the slice LUT (configured as SRL).
$T_{DS}/T_{DH}$	BX/BY configured as data input (DI)	Time before the clock that the data must be stable at the BX/BY input of the slice.

Table 8-3 Virtex-4 SRL Timing Parameter Identification

Figure 8-8 identifies all the Table 8-3 parameters in a timing diagram. The setup times for the Write Enable ( $T_{WS}$ ), Shift\_In ( $T_{DS}$ ) are referenced to the CLK signal. Note the change in Data Out (D) as the Address changes, in a time delay of  $T_{ILO}$ , mentioned earlier in the general

SliceM/L timing model.

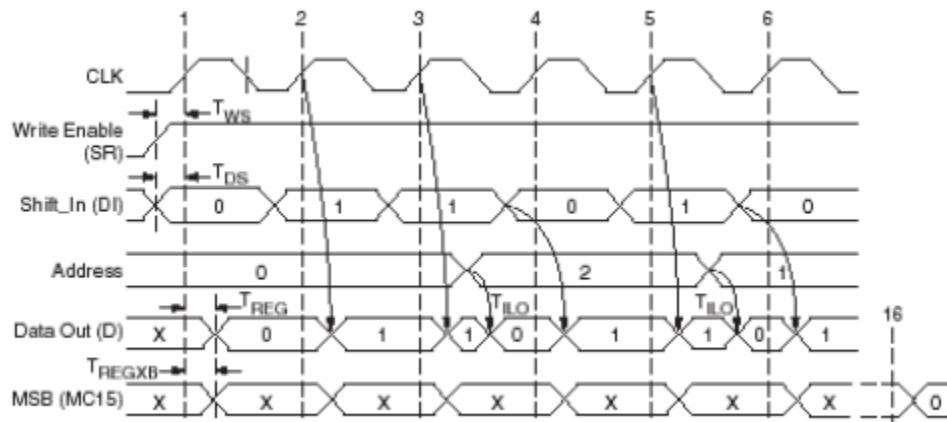


Figure 8-8 Virtex-4 SliceM SRL Timing Characteristics

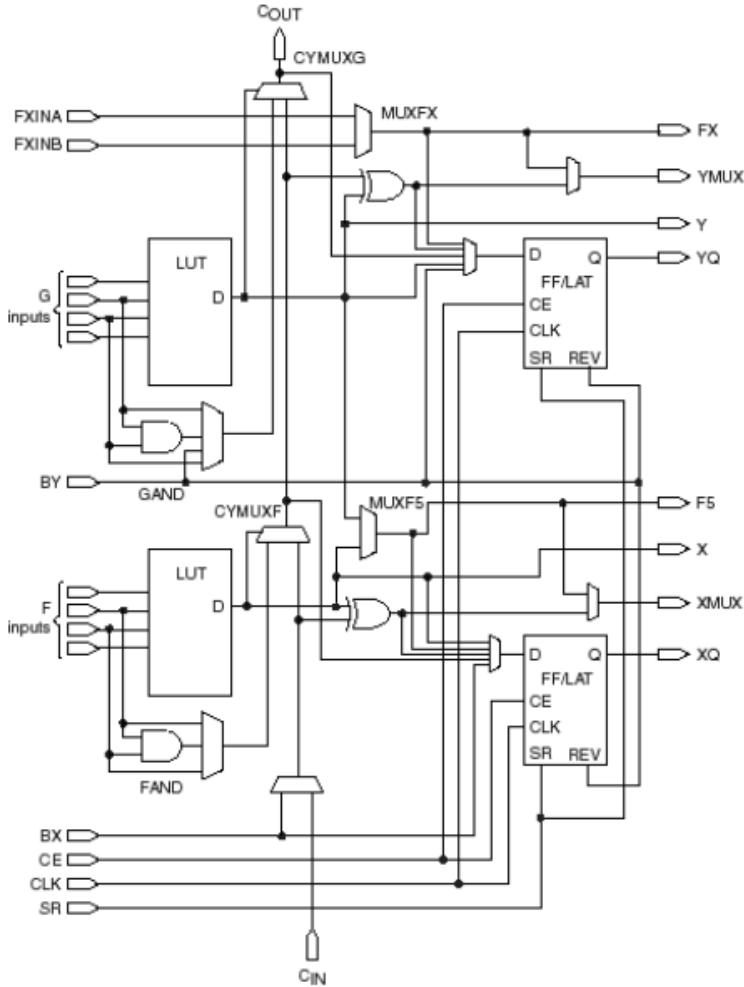


Figure 8-9 Simplified Virtex-4 Slice Carry-Chain Diagram

Fast addition and high speed counter designs use the carry chain (Figure 8-9) to propagate data from one logic cell to another, within a slice, between them and between CLBs. Several of the timing paths are simple combinational paths, such as  $T_{BYP}$ , which is the pass through time delay from  $C_{IN}$  to  $C_{OUT}$ . A more complex path is  $T_{OPCYF}$  or  $T_{OPCYG}$  which describe the delay through the respective LUTs to control of the  $C_{out}$  multiplexers. Naturally, the setup times for data attaching to the D flip flops has a requirement depending on the exact path taken, which may be through the carry circuitry. Table 8-4 and Figure 8-9 summarize the various carry chain paths, and associated time delays.

Parameter	Function	Description
<b>Sequential Delays for Slice LUT Configured as Carry Chain</b>		
$T_{BXY}/T_{BYC}$	BX/BY input to $C_{OUT}$ output	Propagation delay from the BX/BY inputs of the slice, to $C_{OUT}$ output of the slice.
$T_{BYP}$	$C_{IN}$ input to $C_{OUT}$ output	Propagation delay from the $C_{IN}$ input of the slice, to $C_{OUT}$ output of the slice.
$T_{FANDCY}/T_{GANDCY}$	F/G input to $C_{OUT}$ output	Propagation delay from the F/G inputs of the slice, to $C_{OUT}$ output of the slice using FAND (product).
$T_{OPCYF}/T_{OPCYG}$	F/G input to $C_{OUT}$ output	Propagation delay from the F/G input of the slice to $C_{OUT}$ output of the slice.
$T_{OPX}/T_{OPY}$	F/G input to XMUX/YMUX output	Propagation delay from the F/G inputs of the slice, to XMUX/YMUX output of the slice using XOR (sum).
<b>Setup/Hold for Slice LUT Configured as Carry Chain</b>		
$T_{xxS}$ = Setup time (before clock edge) $T_{xxH}$ = Hold time (after clock edge)	The following descriptions are for setup times only.	
$T_{CINCK}/T_{CKCIN}$	$C_{IN}$ Data inputs (DI)	Time before Clock (CLK) that data from the $C_{IN}$ input of the slice must be stable at the D-input of the slice sequential elements (configured as a flip-flop). Figure 5-27 shows the worst-case path.

Table 8-4 Slice Carry-Chain Timing Parameters

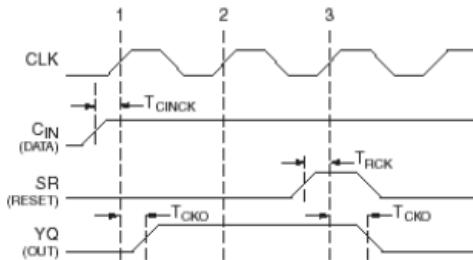


Figure 8-9 Slice Carry-Chain Timing Characteristics

### BRAM Timing

Block RAM timing is a bit more complex than Distributed RAM timing, as the additional enabling and output register management add more factors to consider. Also, Figure 8-10 reveals additional complexity of the NET delays added to the basic parameters when connecting into the BRAM structure.

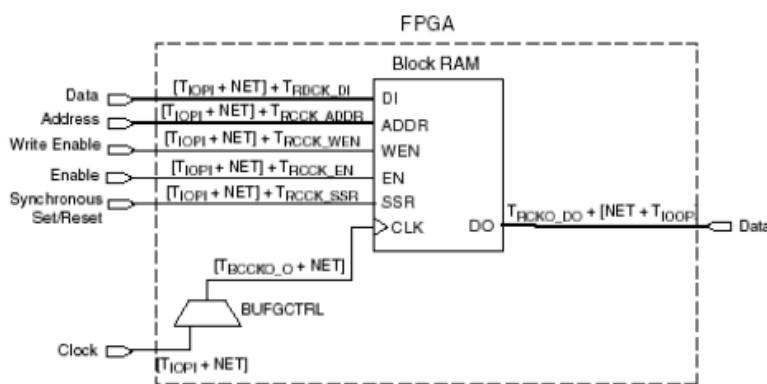


Figure 8-10 BRAM Timing Parameter Identification

The optional behavior mode choices for the BRAM are discussed in more detail Chapter Six, but the timing diagram in Figure 8-11 exposes some of the transactions for various Read and Write cycles. The timing shown illustrates the BRAM outputs first disabled, then issuing a READ, followed by a WRITE, followed by another READ, then a “reset” and an output disable. You can track the various addresses and corresponding data going in and out, on the diagram. Be sure to note that the “reset” condition specifically loads a user chosen value into the output register.

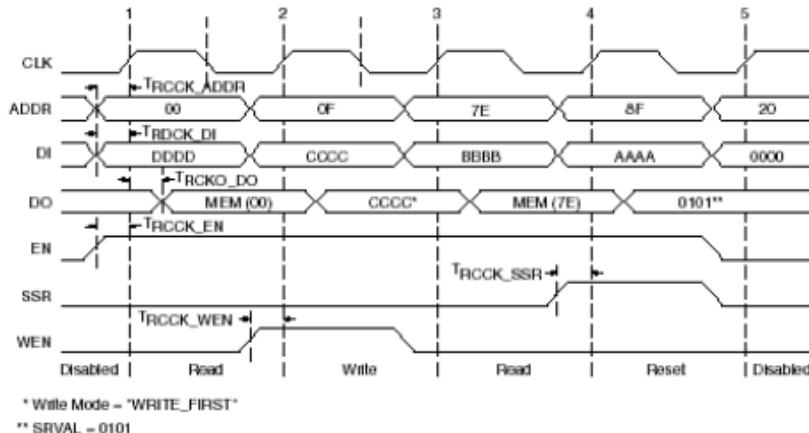


Figure 8-11 BRAM Timing Diagram

Table 8-5 defines all the parameters for both the BRAM, as well as the output register, which may be used in several ways. The requirements for stability on the address being targeted as well as the data setup requirements are straightforward. The Enable and Write Enable are very similar parameters to those of commercially available SRAM modules. Hence, a specific amount of time is needed to enable the data to the BRAM outputs, as well as strobing data into the module. Note that two output delays are associated with the clock – one, for the situation where the data directly exits the BRAM and another when the data is delivered to the output register on the BRAM output lines.

Figure 8-10 is not so much a timing model as simply an identification of sites on the BRAM module that will incur time delays and clarify where additional time delay for interconnection (NET) may impact performance.

Parameter	Function	Control Signal	Description
<b>Setup and Hold Relative to Clock (CLK)</b>			
$T_{RCK_X}$ = Setup time (before clock edge) $T_{RCKH_X}$ = Hold time (after clock edge)	The following descriptions are for setup times only.		
$T_{RCKC\_ADDR}/T_{RCKD\_ADDR}$	Address inputs	ADDR	Time before the clock that address signals must be stable at the ADDR inputs of the block RAM.
$T_{RDCK\_DI}/T_{RCKD\_DI}$	Data inputs	DI	Time before the clock that data must be stable at the DI inputs of the block RAM.
Parameter	Function	Control Signal	Description
$T_{RCKC\_EN}/T_{RCKE\_EN}$	Enable	EN	Time before the clock that the enable signal must be stable at the EN input of the block RAM.
$T_{RCKC\_SSR}/T_{RCKC\_SSR}$	Synchronous Set/Reset	SSR	Time before the clock that the synchronous set/reset signal must be stable at the SSR input of the block RAM.
$T_{RCKC\_WEN}/T_{RCKC\_WEN}$	Write Enable	WEN	Time before the clock that the write enable signal must be stable at the WEN input of the block RAM.
$T_{RCKC\_REGCE}/T_{RCKC\_REGCE}$	Optional Output Register Enable	REGCE	Time before the clock that the register enable signal must be stable at the REGCE input of the block RAM.
<b>Sequential Delays</b>			
$T_{RCKO\_DO} \text{ (Max)}$	Clock to Output	CLK to DO	Time after the clock that the output data is stable at the DO outputs of the block RAM (without output register).
$T_{RCKO\_DO} \text{ (Min)}$	Clock to Output	CLK to DO	Time after the clock that the output data is stable at the DO outputs of the block RAM (with output register).

Table 8-5 BRAM Timing Parameters

Surprisingly, the timing for the BRAM is independent of the configuration chosen. This was described in Chapter Six, but the key idea for BRAM configuration timing is simply that access to data line “D0” is common to all configurations, setting the access time for all.

### Multiplier Timing

Figure 8-12 shows a simplistic model for the Two’s complement, 18 X 18 multiplier. The key timing parameter for the multiplier is from stable arrival of input variables until the result is present on the product side of the module. Unlike the BRAM, the multiplier exhibits a different speed depending on the size of the operands being multiplied. This is shown in

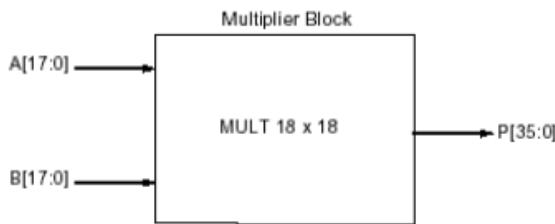


Figure 8-12 18 X 18 Multiplier Model

Figure 8-13, where the time delay is shown to follow a curve, depending on the pins being driven at the multiplier output. This is a simplification, as the number of pins is a discrete variable, and not continuous, as shown. Nonetheless, the nature of a combinational multiplier is one where partial products are added together, and that

requires a time delay for accumulating the correct carries as the result proceeds to be developed.

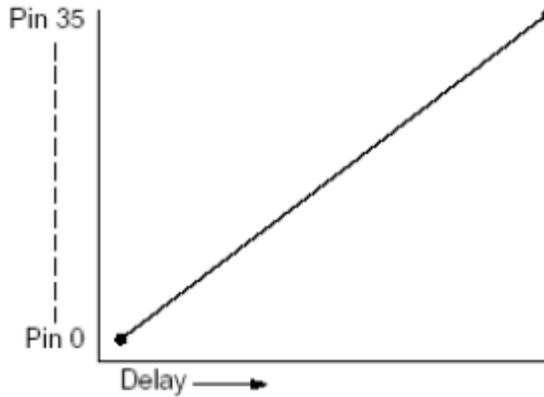


Figure 8-13 Multiplier Time delay versus Output Pins

Figure 8-14 illustrates the incremental nature of the result forming for the multiplication of two four bit operands. Observe the additional time delay to produce each bit of the final product, in succession. P0 arrives very early, but we watch the consecutive arrival of each product bit, as P1 subsequently progresses to P2 in order until finally P7 completes the results. Because most algorithms require the complete result before proceeding, our timing models must account for the worst case arrival of the longest bit of the result, in general.

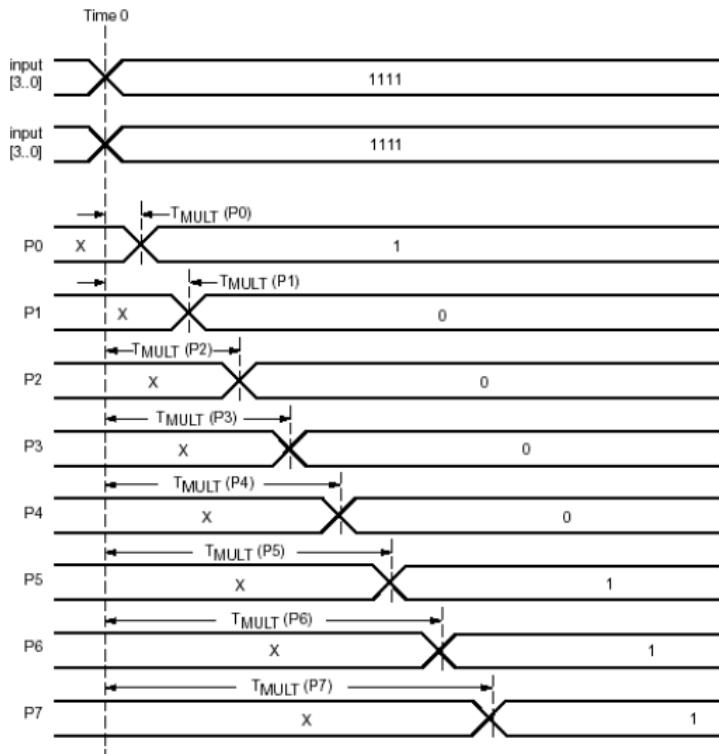


Figure 8-14 Timing Parameters for a 4 X 4 multiplier

The various product time delays are presented in a table, for each product, accounting for speed grades. Naturally, the Xilinx design software accounts for all of this detail, on your behalf.

### Interconnect Time Delay

Interconnect time delay in Virtex families may seem like a mystery. The timing models do not explicitly reveal the time delay of the various interconnect lines within the architecture. There are many reasons for this. First, the delays for lines like direct interconnect to neighboring CLBs are typically very small. Second, the interconnect delay must be de-rated by the circuit loading encountered by the connection lines to logic and other elements. For instance, when a hex line interconnects to a long line, the combined time delays must be adjusted. They must be de-rated for loading of other multiplexers and buffers encountered. Third, only the design software can actually tally all of these incremental delays and loading factors to deliver an accurate time delay value. Hence, the interconnect time delay is most evident in the static timing reports produced by the design software, or in the design view program, revealing routes taken to connect the design. Finally, the various Virtex and Spartan families are not identical in available user interconnection until the 28 nanometer node. Newer products are now identical in architecture in order to gain tool efficiencies.

The connection models differ among the families. Figure 8-15 should impress the reader of the level of complexity interconnect has. The nomenclature of Figure 8-15 has the logic slice described as a “CLE”, to which signals exit on the OMUX and enter on the IMUX. These type multiplexers are briefly described at the end of Chapter Three. These multiplexers also feed into and are driven from the large innocuous block labeled “INTERCONNECT”. Figure 8-16 shows additional detail on the INTERCONNECT block, which is a veritable “rat’s nest” of available connections. Examining Figure 8-16 makes imminently clear the reason Virtex FPGA devices need large numbers of metal layers to make connections in a small area! Hence, this complexity is largely why details on the interconnect time delays remain unpublished, as specific parameters.

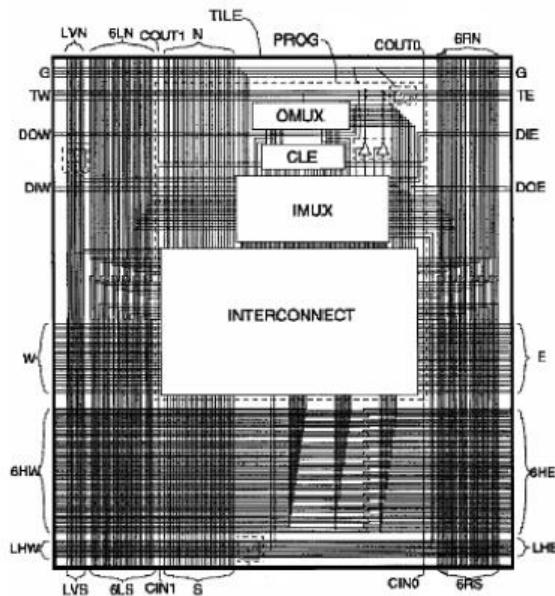


Figure 8-15 Virtex Style Logic Tile

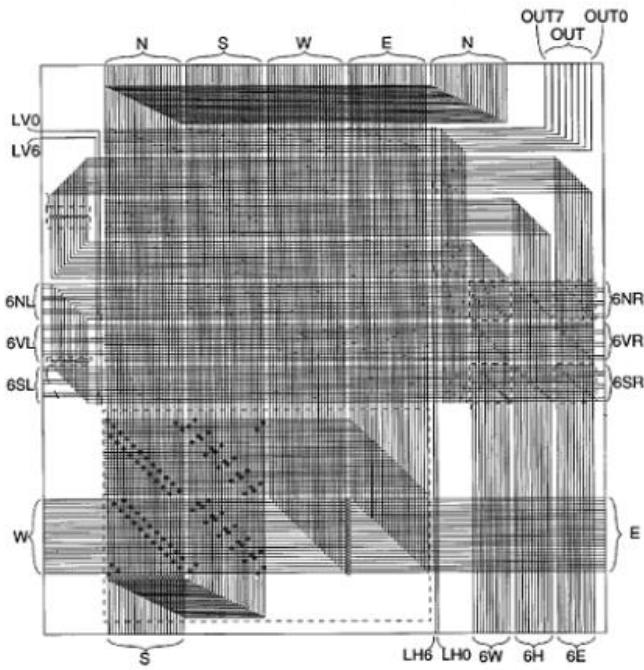


Figure 8-16 Greater Detail on the INTERCONNECT Block

## Timing Reports

Xilinx design software maintains data bases of the various incremental time delays for the different logic fabrics and special modules. As mentioned earlier, designs exist in the software primarily in the form of netlists, where functionality and connections are evident. As the netlist is formed in the software, associations to time delays for the logical units arise, as do the time delays for the various connection paths chosen. Static timing is made evident as the design progresses through the software to the implementation phase. Several different timing reports are produced for the design, depending on how far the user chooses to “compile” it. For instance, right after synthesis, there is a timing report produced, but it has little detail about the final product, so lacks accuracy. As more is known about the final design, more is learned about the real timing, and the static timing report becomes refined. After the part is fully placed, routed and a programming file produced, everything is known and the timing is a fully accurate model of the real circuit time behavior.

## Timing Strategies

“Timing” for a design is not usually a single parameter, but in many cases, the fastest clock rate possible –  $F_{MAX}$  is the primary single target. Much of Xilinx design software focuses on maximizing  $F_{MAX}$  but systems using multiple clocks make that very hard to analyze. Let’s look at two tactics that can improve overall timing in a system focusing on maximizing  $F_{MAX}$ . The first tactic is load splitting, and the second tactic is pipelining.

### Load Splitting (also known as Fanout Reduction)

Load splitting is simply recognizing that driving multiple loads from a signal adds time delay due to both routing as well as end logic drive requirement. It takes current to charge metal up to a logic one, and that metal exists as interconnect as well as transistor gate capacitance. If a driving signal must deliver its logic payload to 32 flip

flop reset inputs, then each reset contributes a fraction of the whole in time delay, and connections between all the players add capacitance, as well. By splitting the signal into two drivers, each with 16 flip flops and their corresponding interconnection, each driver encounters roughly half what a single driver would encounter. It speeds up the transaction.

Our analysis is simplistic with load splitting, as it assumes that the logic for the driver is still only represented once. That is, the same logic drives two buffers. In some cases, load splitting may require logic replication. Logic replication means two copies of the same logic must be created to drive independent drivers which in turn deliver to the end load – say our set of flip flop resets. Because this only occurs with heavily loaded sections of designs, its impact on logic capacity is usually minimal.

One problem with logic replication is getting synthesis tools to do it. Synthesis tools employing netlist optimization, are designed to recognize duplication and eliminate it. That being the case, they easily recognize duplication, and act accordingly. This is a classic case, where users and developers have arrived at ways of either flagging sections of code to not be optimized, or introduce “tricks” that disguise the logic as not being duplicated. Renaming signals, or visiting I/O pins often solve the problem depending on the synthesis vendor. Synthesis tools can do load splitting for you, but if you want a specific result you need to properly indicate what you want to that tool.

## Pipelining

Pipelining has had its biggest impact in arithmetic unit design, but it is a common technique that can be used to increase  $F_{MAX}$ . The basic idea stems right from the definition of  $F_{MAX}$ . Here is a simple version:

$$F_{MAX} = 1/(T_{LOGIC} + T_{ROUTING} + T_{su} + T_{co})$$

We see that this is an expansion in complexity of our  $F_{TOG}$  expression, and we immediately see that  $F_{MAX}$  is less than  $F_{TOG}$  because of the logic and routing delays added in.  $F_{MAX}$  would get closer to  $F_{TOG}$ , if logic and routing delays could be reduced. Figure 8-17 shows a simplified case that may be able to use pipelining for an advantage.

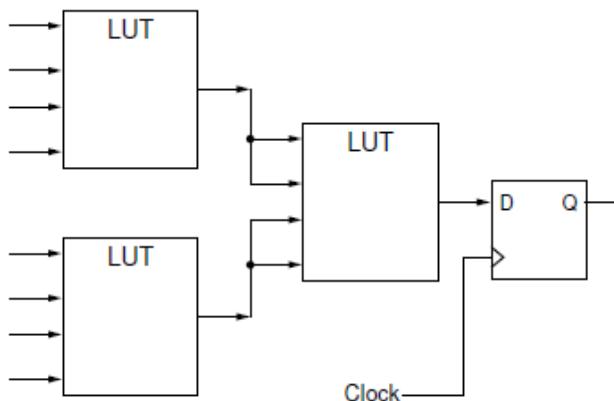


Figure 8-17 Simplified State Transition Logic

In this case, the speed at which the flip flop can switch – the  $F_{MAX}$  – is limited by the time delays of the flip flops, as well as the propagation delay of the two layers of LUTs

driving its D flip flop. It's easy to envision the TILO for the LUTs, but also included will be the interconnection delay between the LUTs. We can speed things up, by introducing pipeline registers between the LUTs, as shown in Figure 8-18.

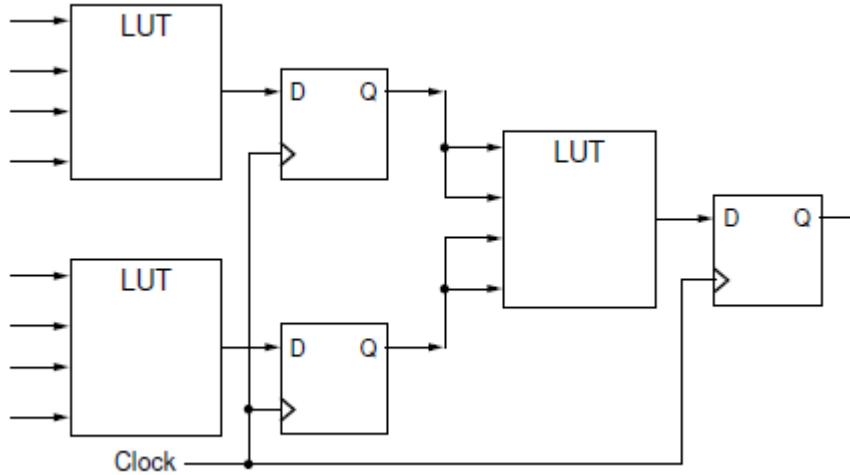


Figure 8-18 Simplified State Transition Logic with Pipelining

In Figure 8-18, we simply introduce flip flops between the cascaded LUTs, so the delay presented to all three flip flops is now the LUT TILO with its associated connection delay, most likely within the slice. This lets us speed up the clock, as the primary delay is just the interconnect between slices, which is primarily a design software placement problem.

From a resources viewpoint, Figure 8-17 was probably not using its available slice flip flop resources, anyway, so no real assets are lost by pipelining. The  $F_{MAX}$  increases in Figure 8-18. The only downside, which may not be important, is that the latency time may have increased. Latency is the time between when data enters the state machine, and the state machine is giving the right output, or residing in the correct state. It will take more clocks for the right hand flip flop to arrive at its correct state with pipelining, than without. However, the clock speed will be faster, so that additional latency may be negligible. Being able to process data at a faster clock rate may outweigh the potential latency issue. Another technique, similar to pipelining is called *retiming*. Retiming is typically performed by the synthesis software, and involves examining all the logic delays between flip flops on the same clock network. Identifying long combinational delays that may be re-balanced to have similar delays between flip flops is the target. Often, this involves balancing the combinational delays by moving flip flops – to permit the final solution to run at a faster clock time. Again, this must be done carefully. Retiming can actually reduce time penalty (in elapsed time).

If done properly, latency in cycles will be constant. There are many little tricks to obtaining the highest speed, but load splitting and pipelining along with managing DCMs cover a lot of important ground. Now, let's consider the high level topic of timing closure.

## Timing Closure

Timing closure is simply the process of getting the timing results you want out of a design. First off, it is important to know what you want. If you have no idea of what

speed can be obtained, then you simply get what the design software delivers. However, if you know what you want, then you should be aware of timing constraints that can be offered to the design software to provide guidance. As suggested earlier, constraints make the problem greater for the design software, but the overriding goal is to give customers what they seek. We only discuss the general approach taken, leaving detail to the references.

Topping the list is to use proper HDL coding procedures. This primarily means to use synchronous (one clock) design as much as possible, and avoid elaborate hierarchy that serves to confuse timing issues. Asynchronous design only complicates the analysis, further blurring appropriate choices to be made. Multiple clocks also confuse the situation, but orderly boundary crossing (i.e., FIFOs) help a lot. Each synthesis tool operates with a library of choices. Dictating appropriate choices is important at the outset, to obtain best results for the PAR tools later. In fact, many synthesis tools provide for high level choices to be made with regard to constraints, so they provide a suitable netlist for the PAR tools. This can only be learned by investigating the particular synthesis tool literature.

After coding style comes global constraint. This is a broad topic, but the main idea is to choose feasible global constraints. An infeasible constraint would be along the lines of selecting an  $F_{MAX}$  faster than an  $F_{TOD}$ . Feasibility of the constraints will be known early, often right after synthesis. The synthesizer provides an estimate of what might be achieved with a timing summary derived from a first netlist. If the synthesized netlist is within specification, the design is likely feasible to meet requirements after PAR. If the design is infeasible right after synthesis, it should be recoded, or have the constraints revised. Two controls to be managed at a high level are the synthesis constraints and the PAR effort level. Each synthesizer provides unique constraints, so that means learning your particular third party tool constraints. Most of these are summarized in the Xilinx Constraints Guide.

PAR operates at different *effort levels*. An effort level is simply how much time the tool takes to arrive at a target result. You can pick a low, medium or high effort level for your PAR runs. Chapter Four discussed the various approaches taken by the evolving software, so you can envision that it may iterate between placement and routing, with various evaluations of slack, simulated annealing, etc.

Often times, a key *critical path* emerges from a design that permits focusing the synthesis and PAR tools on the set of signals comprising that path. The critical path can be a leading indicator for whether a specific version of a synthesized netlist is pushed forward into the PAR stage. Hopefully, an acceptable netlist with sufficient proximity to the global goals is met, with at least an approximation of the final value for the critical paths being met. Should this be the case, but still require further tuning, floorplanning and manual intervention may be an acceptable approach. Xilinx has offered floorplanning tools from the beginning, and they are still a popular method, today. The idea here is “divide and conquer.” By synthesizing blocks of the design and identifying block level timing requirements, the problem can be reduced. A great tool for this is the device view tools discussed in Chapter Four.

Two blocks can be independently synthesized, placed, routed and analyzed. When each smaller, placed and routed block is faster than need be, the two may then be stitched together. The additional timing penalty attributed to the stitching can then be analyzed. Whole designs can be constructed in this manner, if needed. The approach gives great control, but many want to only work at the high level and drive the global

constraints to get their results. Of course, the other way out of a problem can be to pick a larger, faster part, but that costs money.

## Conclusions

Timing is a vast topic in digital FPGA devices. We have only touched on some of the fundamentals. To become a great designer will take effort, training, study and experience. To that end, Xilinx offers a vast set of design examples, tutorials, and design guides, constraint manuals and so on. Some of the more important ones are listed in the references below. One of the best approaches to gaining insight into the timing of designs is to work through many small designs, fully understanding the produced timing results. Literally, this means starting with a gate or two, then adding a flip flop or two, then build shifters, counters, adders and so on, to study the effects of little changes. As each design grows, the timing changes, and insight regarding the results occurs. Taking a short cut around these small experiments often results in a longer path to your goal of becoming a good FPGA designer. To reiterate, the basics account for about 99% of the process.

## References

These references should fill in many details left unanswered in this chapter. Reference 1 alone, has 888 pages on the topic of constraints. It discusses Xilinx software constraints, but also covers third party synthesis tool constraints, so you can know how choices made with a Synopsys or a Synplicity synthesis tool can achieve your design goals. Other references detail blocks not discussed and tools obliquely mentioned.

1. ISE 10.1i Constraints Guide
2. RocketIO Transceiver User Guide (ug024.pdf)
3. Extreme DSP User Guide (ug073)
4. XAPP462 Using Digital Clock Managers (DCMs) in Spartan-3 FPGAs
5. UltraFast Design Methodology Guide for the Vivado Design Suite, UG949 (v2016.2) June 8, 2016

# Chapter 9 High Speed Clock Management

## Overview

The size of today's FPGA device is substantially larger than what was available in the past, in both a logic capacity and physical area. If there was ever an environment where "distance equals time", it is in an integrated circuit, and managing the signal arrival to the various sites inside is a daunting problem. Carefully managing both the data and clocking signals for thousands of flip flops could be a nightmare, if it were not for the anticipation of this situation by Xilinx designers. The problem is reduced by more than half, by combining a clock network as discussed in the architecture chapters, with digital clock management.

## Clock Resources Architecture

The FPGA device requires a versatile clocking network to provide a clock to all tiles which ensures proper synchronous operation. Local, regional, and global regions are defined, with their appropriate delay matched clock buffers along with resources to manage the phase and frequencies of the clocks.

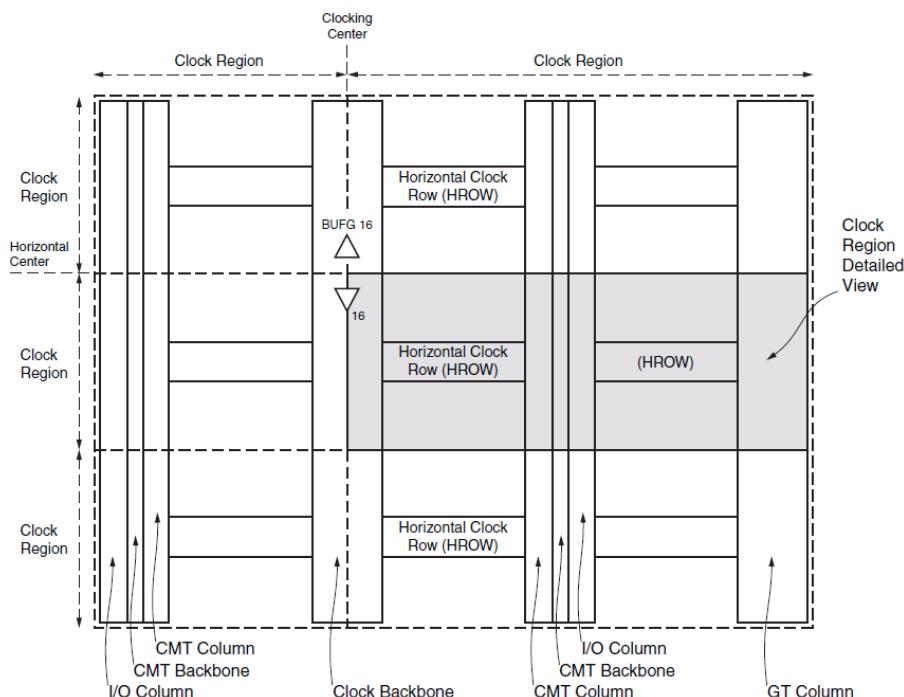


Figure 9-2 7 Series FPGA High-Level Clock Architecture View

The distribution of clocks provides for a rich set of features for both local, regional, and global clock references. The buffers provide clock enable and synchronous clock switching without creating 'runt' pluses (illegal clock periods that are too short, either high or low).

## The Clock Management Tile (CMT)

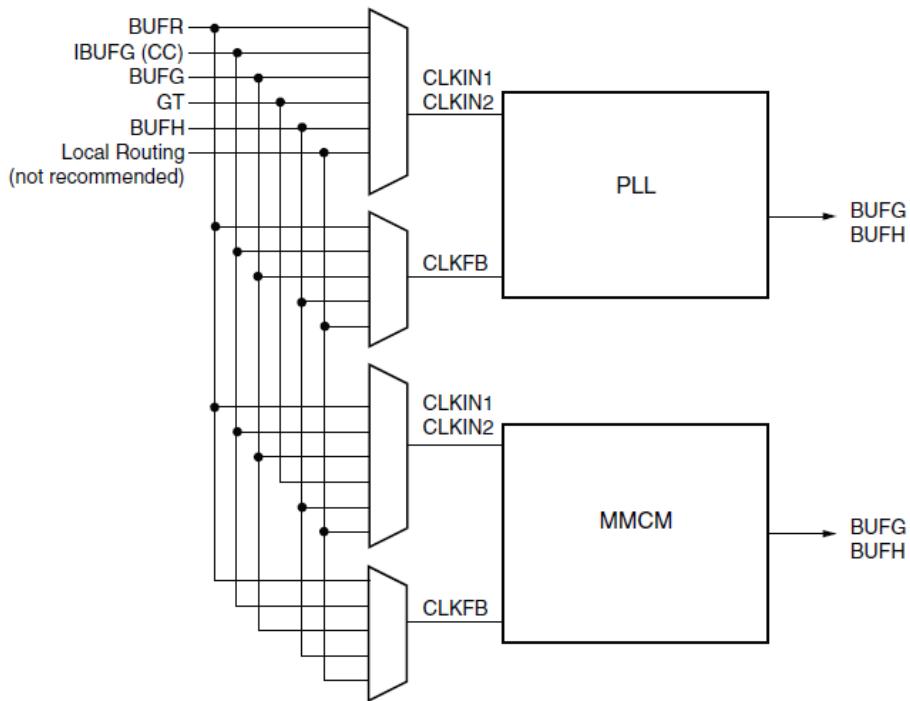


Figure 9-1 The Clock Management Tile

Figure 9-1 shows the 7-series Virtex family solution for clock management, the Clock Management Tile (CMT) which includes a mixed-mode clock manager (MMCM) and a phase-locked loop (PLL). The PLL contains a subset of the MMCM functions. At the core of the 7 series FPGA devices CMT is the architecture similar to the Virtex-5 and Virtex-6 FPGA devices with enhanced functions and capabilities. Each circuit has its own characteristic properties and brings capabilities permitting users to increase state machine speeds, and improve reliability. Although they share some common attributes, each is a little different in their target goal, and will be discussed separately.

The first Virtex FPGA devices included only the Digital Delay Lock Loop (DLL) with the clocking network to produce synchronized timing solutions. The overall behavior is similar to that of a phase lock loop (PLL), except all actions occur digitally in the DLL, and the frequency range of locking is typically different. As more was learned about the need for time management, the DLL evolved into the Digital Clock Manager in Virtex II and Spartan 3/E. Virtex 4 refined these modules even more, and also added a set of new modules called Phase Matched Clock Dividers (PMCD). The all-digital nature of the DCM led to repeatability and stability, yet suffered from a discrete time step which led to unavoidable jitter. With the 7 series, the DCM is replaced by a digitally assisted and auto calibrated phase lock loop (PLL) which has all the benefits of the DCM, with less intrinsic jitter, as well as jitter filtering.

### Virtex 7 Clock-Capable Inputs

External user clocks must be brought into the FPGA on differential clock pin pairs called clock-capable (CC) inputs. Clock-capable inputs provide dedicated, high-speed access to the internal global and regional clock resources. Clock-capable inputs use

dedicated routing and must be used for clock inputs to guarantee timing of various clocking features. General-purpose I/O with local interconnects should not be used for clock signals. Each I/O bank is located in a single clock region and includes 50 I/O pins. Of the 50 I/O pins in each I/O bank in every I/O column, there are four clock-capable input pin pairs (a total of eight pins). Each clock-capable input:

- Can be connected to a differential or single-ended clock on the PCB
- Can be configured for any I/O standard, including differential I/O standards
- Has a P-side (master), and an N-side (slave)

Single-ended clock inputs must be assigned to the P (master) side of the clock-capable input pin pair. If a single-ended clock is connected to the P-side of a differential clock pin pair, the N-side cannot be used as another single-ended clock pin—it can only be used as a user I/O.

## **Global Clocking Resources**

Global clocks are a dedicated network of interconnect specifically designed to reach all clock inputs to the various resources in an FPGA. These networks are designed to have low skew and low duty cycle distortion, low power, and improved jitter tolerance. They are also designed to support very high frequency signals. Understanding the signal path for a global clock expands the understanding of the various global clocking resources. The global clocking resources and network consist of the following paths and components:

- Clock Tree and Nets - GCLK
- Clock Regions
- Global Clock Buffers

### Clock Tree and Nets – GCLK

7 series FPGA device clock trees are designed for low-skew and low-power operation. Any unused branch is disconnected. The clock trees can also be used to drive logic resources such as reset or clock enable. This is mostly used for high fanout/load nets. In the 7 series FPGA device architecture, the pin access of the global clock lines are not limited to the logic resources clock pins. The global clock lines can drive pins in the CLB other than CLK pins (for example: the control pins SR and CE). Applications requiring a very fast signal connection and large load/fanout benefit from this architecture.

### Clock Regions

7 series devices improve the clocking distribution by the use of clock regions. Each clock region can have up to 12 global clock domains. These 12 global clocks can be driven by any combination of the 32 global clock buffers available in a monolithic device or SLR. The dimensions of a clock region are fixed to 50 CLBs tall (50 IOBs) and spanning the left or right side of the die. In 7 series devices, the clock backbone splits the device into a left or right side. The backbone is not located in the center of the die. By fixing the dimensions of the clock region, larger 7 series devices can have more clock regions. The 7 series FPGA devices supply from 4 to 24 clock regions.

## Global Clock Buffers

There are 32 global clock buffers in every 7 series device. A CCIO input can directly connect to any global clock buffer in the same half of the device. Each differential clock pin pair can connect to either a differential or single-ended clock on the PCB. When used as a differential clock input, the direct connection comes from the P-side of the differential input pin pair. When used as a single-ended clock input, the P-side of the pin pair must be used because a direct connection only exists on this pin. If a single-ended clock is connected to the P-side of a differential pin pair, then the N-side cannot be used as another single-ended clock pin. However, it can be used as a user I/O.

CMTs in the top half of the device can only drive the BUFGs in the top half of the device and CMTs in bottom half can only drive BUFGs in the bottom half. Similarly, only BUFGs in the same half of the device can be used as feedback to the CMTs in the same half of the device. Gigabit transceivers (GTs) can only directly connect to MMCMs/PLLs when the CMT column extends into regions that also contain a full column of GTs and I/Os. The Virtex-7T and Virtex-7XT devices have these full columns.

Global clock buffers allow various clock/signal sources to access the global clock trees and nets. The possible sources for input to the global clock buffers include:

- Clock-capable inputs
- Clock Management Tile (CMT) consisting of mixed-mode clock managers (one MMCM and one PLL per CMT) driving BUFGs in the same half of the device.
- Adjacent global clock buffer outputs (BUFGs)
- General interconnect
- Regional clock buffers (BUFRs)
- Gigabit transceivers

The 7 series FPGA device clock-capable inputs can drive global clock buffers indirectly through the vertical clock network that exists in the clock backbone column. The 32 BUFGs are organized into two groups of 16 BUFGs in the top and bottom of the device. Any resources (for example, GTX transceivers) connecting to the BUFGs directly have a top/bottom limitation. For example, each MMCM in the top can only drive the 16 BUFGs residing in that top of the device. Similarly, the MMCMs in the bottom drive the 16 BUFGs in the bottom. All global clock buffers can drive all clock regions in 7 series devices. However, only 12 different clocks can be driven in a single clock region. A clock region (50 CLBs) is a branch of the clock tree consisting of 25 CLB rows up and 25 CLB rows down. A clock region spans halfway across the device. The clock buffers are designed to be configured as a synchronous or asynchronous glitch-free 2:1 multiplexer with two clock inputs. There is a dedicated path (routing resource) for BUFG cascading to allow for more than two clock input selections. The 7 series FPGA devices control pins provide a wide range of functionality and robust input switching.

In the 7 series FPGA devices clocking architecture BUFGCTRL multiplexers and all derivatives can be cascaded to adjacent clock buffers within the group of 16 in the upper and lower half of the device, effectively creating a ring of 16 BUFGMUXes

(BUFGCTRL multiplexers) in the upper half and another ring of 16 in the lower half. Figure 9-3 shows a simplified diagram of cascading BUFGs.

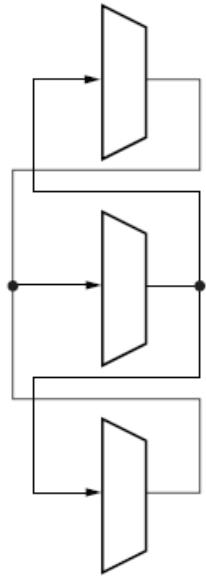


Figure 9-3 Cascading BUFG

### Regional Clocking Resources

Regional clock networks are clock networks independent of the global clock network. Unlike global clocks, the span of a regional clock signal (BUFR) is limited to one clock region, one I/O clock signal drives a single bank. These networks are especially useful for source-synchronous interface designs. The I/O banks in 7 series devices are the same size as a clock region. To understand how regional clocking works, it is important to understand the signal path of a regional clock signal. The regional clocking resources and network in 7 series devices consist of the following paths and components:

- Clock-Capable I/O
- I/O Clock Buffer—BUFIO
- Regional Clock Buffer—BUFR
- Regional Clock Nets
- Multi-Region Clock Buffer—BUFMR/BUFMRCE
- Horizontal Clock Buffer—BUFH, BUFHCE
- High-Performance Clocks

Please consult the 7 series clock management user's guide for the details.

### **Back to the Clock Management Tile**

In 7 series FPGA devices, the clock management tile (CMT) includes a mixed-mode clock manager (MMCM) and a phase-locked loop (PLL). The PLL contains a subset of the MMCM functions. At the core of the 7 series FPGA device CMT is the architecture similar to the Virtex-5 and Virtex-6 FPGA device with enhanced functions and capabilities. The CMT backbone can be used to chain CMT clocking functions; however, there are limitations to placement, distance, and connectivity resources.

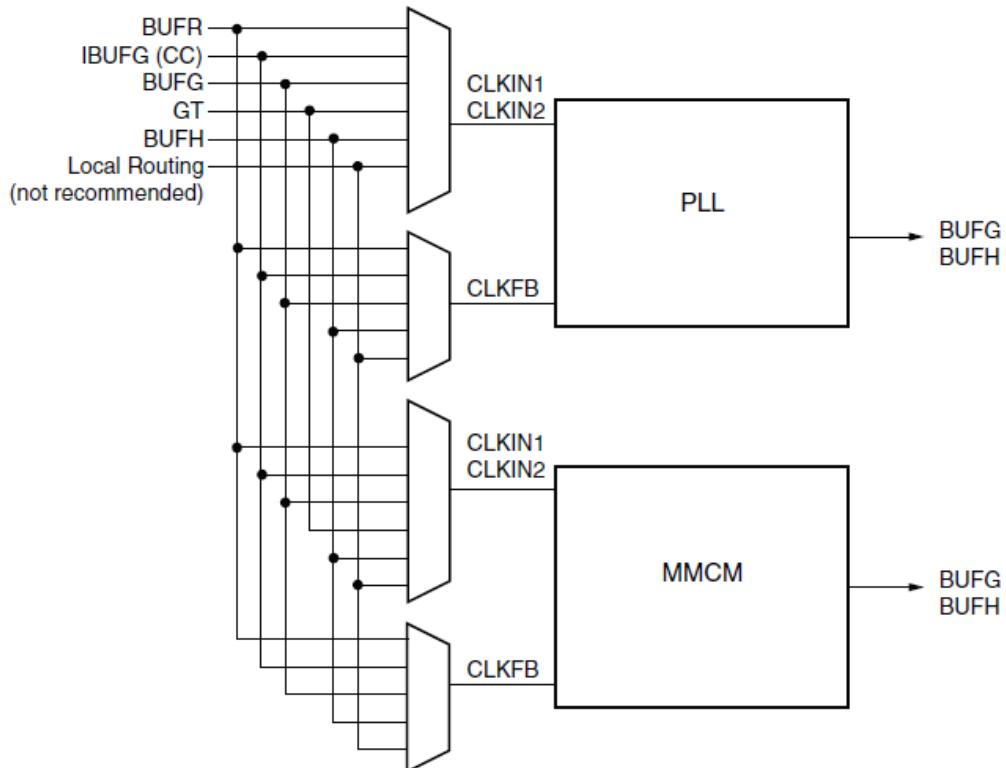


Figure 9-4 Block Diagram of the 7 Series FPGA device CMT

The CMT diagram (Figure 9-4) shows a high-level view of the connection between the various clock input sources and the MMCM/PLL. In 7 series FPGA devices, the clock input connectivity allows multiple resources to provide the reference clock(s) to the MMCM/ PLL. The number of output counters (dividers) is eight with some of them capable of driving out an inverted clock signal ( $180^\circ$  phase shift). For backward compatibility with DCMs, nine independent outputs can be selected for mapping the DCM outputs directly into the MMCM. 7 series FPGA devices MMCMs have infinite fine phase-shift capability in either direction and can be used in dynamic phase-shift mode. The resolution of the fine phase shift depends on the VCO frequency. Fractional divide functionality in increments of  $1/8^{\text{th}}$  (0.125) for CLKFBOUT and CLKOUT0 are available to support greater clock frequency synthesis capability. 7 series FPGA devices have added spread spectrum capabilities to the MMCMs. If the MMCM spread spectrum feature is not used, a spread spectrum on an external input clock will not be filtered and thus passed on to the output clock.

### MMCMs and PLLs

7 series devices contain up to 24 CMT tiles. The MMCMs and PLLs serve as frequency synthesizers for a wide range of frequencies, serve as a jitter filters for either external or internal clocks, and de-skew clocks. The PLL in the 7 series FPGA devices, a subset of the MMCM functionality, is based on the MMCM and not necessarily based on previous PLL designs. The additional features supported by the MMCM are:

- Direct HPC to BUFR or BUFI0 using CLKOUT[0:3]
- Inverted clock outputs (CLKOUT[0:3]B)
- CLKOUT6
- CLKOUT4\_CASCADE

- Fractional divide for CLKOUT0\_DIVIDE\_F
- Fractional multiply for CLKFBOUT\_MULT\_F
- Fine phase shifting
- Dynamic phase shifting

With such a generous wealth of features and capabilities, there is the temptation to use them all. That is generally a bad idea. The simplest clocking solution (one global clock) is always the easiest and best to implement. Next would be fractionally related synchronous global clocks with synchronous domain crossings. Unfortunately, there are some clocks which must be at some specified frequency, and others at other rates, so that asynchronous clock crossings with asynchronous global clocks may be unavoidable. In those cases, pay attention to the clock domain crossings (CDC), and use the correct CDC primitives.

### [Video: cross-clock-domain-checking-cdc-analysis](#)

How the Clocks are Managed Depends on the Use Mode. System Synchronous or Source Synchronous. These use cases are discussed below.

### System Synchronous Design

To align the phases, the PLL or MMCM is used to insert delays to match phases. The accuracy is specified in the data sheet, and is generally +/- 100 ps or so and is maintained over temperature and voltage variations to guarantee that timing is met.

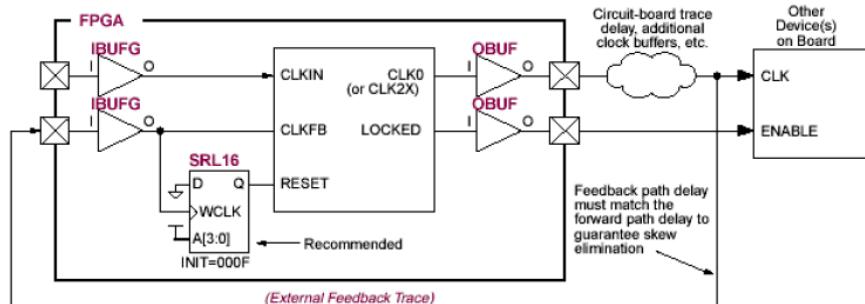


Figure 9-5 Eliminating Skew on External Clock Signal

Figure 9-6 shows a standard pipeline of two logic blocks in a system, sharing the same clock, but one block's data output feeds the next block's data input. This style of design has been mainstream for about 50-60 years in digital electronics, and is called System Synchronous design.

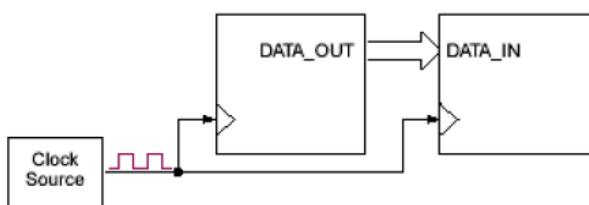


Figure 9-6 System-Synchronous Applications

Figure 9-7 shows another approach, called Source Synchronous, where data is forwarded - along with a clocking signal - from one block to another. The data source is also the clock source, so there is local synchronization at every data exchange. High speed data communications and multiphase memory interface protocols have adopted this method very successfully, and it reduces data bandwidth loss due to clock skews within a system. More discussion on both occurs in the memory interfacing and data communication chapters.

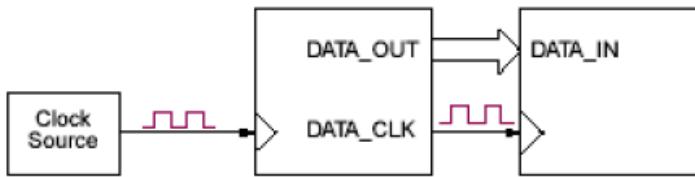


Figure 9-7 Source-Synchronous Applications

Figure 9-8 exposes the timing problem of System Synchronous design, whereby the data delay must be managed for the whole system, so that all setup times with respect to the System Synchronous clock must be met with respect to a rising clock (usually) and delayed data arrangement. The bottom two waveforms show first, a Source Synchronous clock arriving in phase with the data, then (bottom) a phase shifted version. The phase shifted version is a 90 degree phase shifted clock, capable of capturing data on both a rising and a falling clock edge, thereby doubling effective data transmission rate.

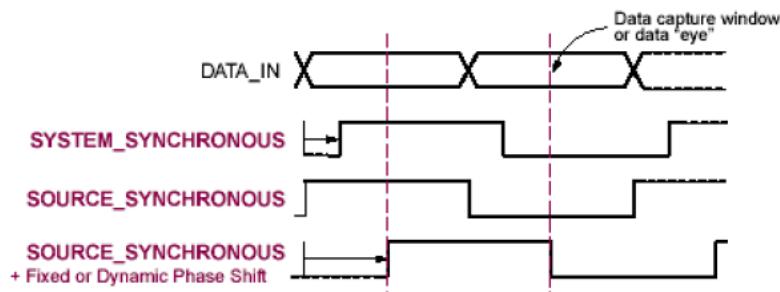


Figure 9-8 System Synchronous and Source Synchronous Timing

Vital to effective Source Synchronous double data rate transmission is the ability to create clocks that have 50% duty cycle. Knowing where the 90, 180 and 270 degree phase points are on a clock, permit duty cycle correction fairly easily. But having that ability now permits designs to have the same setup time for a rising edge clock as for a falling edge clock. If these values were different, standard synchronous design methods would much harder to use in a double data rate system.

### Accuracy Limitations – Jitter

Jitter. It's the bane of all synchronization methods, today. There are many variations of clock jitter, and a number of factors that result in its existence. Power supply noise, thermal drift and simultaneous switching outputs (SSO) can all claim a piece of the jitter picture. Compensating for jitter can substantially reduce potential overall performance. Figure 9-9 shows the basic clock jitter situation. In a simplistic definition, it is a deviation from the ideal clock. But, that's important. All classic synchronous

design methods require the designer to identify the setup, hold and clock to output time for the behavior of flip flops with respect to the clocking event.

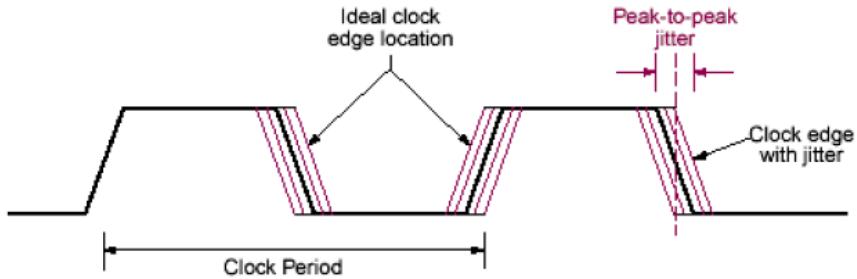


Figure 9-9 Clock Jitter

Jitter makes that be a probabilistic determination, so performance is lost because the design must be altered to account for what might happen with an unpredictable clock. The main two contributors are peak-to-peak jitter and cycle to cycle jitter. Each robs performance in a different way.

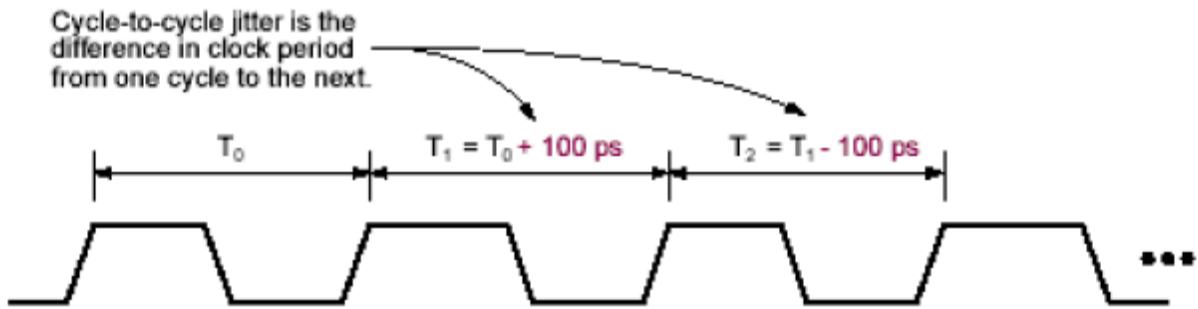


Figure 9-10 Cycle to Cycle Jitter Example

Figure 9-10 shows cycle to cycle jitter, where a baseline “ideal” cycle has  $T_0$  as its period. A successive period increases by 100 ps and a subsequent cycle decreases by 100 ps, back to the amount of  $T_0$ . In this case, the jitter is specified as  $\pm 100$  ps, which means that the clock period will never exceed this deviation range, over millions of clocks, from one cycle to the next. Peak to peak jitter indicates the earliest and latest transition times compared to the ideal clock transition time over many clocks as compared with a mythical perfect reference clock.

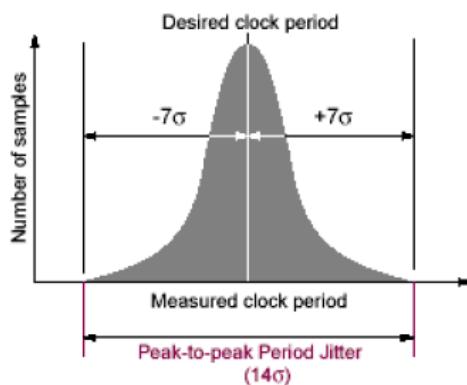


Figure 9-11 Peak to Peak Period Jitter Distribution

Figure 9-11 shows the probability distribution for peak-to-peak jitter. The CMT is specified with a jitter deviation of +/- 7 sigma, to a 14 sigma peak-to-peak period jitter. A 14 sigma peak-to-peak jitter, equates to a maximum typical bit error rate of  $1.28 \times 10^{-12}$ . Jitter is almost always bounded, so it in practice is never this large. It is true that the longer you measure the peak to peak value, it increases. As a matter of practicality, measurement intervals of tens of minutes should suffice.

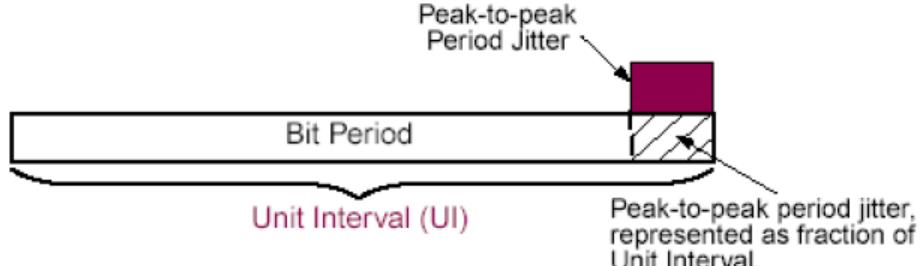


Figure 9-12 Period Jitter Specified as a Fraction of a Unit Interval

Another industry standard way of describing jitter geometrically is shown in Figure 9-12. Here, we show the amount of time allocated to a bit. This is called a Unit Interval (UI). In Figure 9-12, we see that the jitter occupies a fraction of that UI. If we were passing data on both ends of the clock period, we would simply allocate half the clock period as the bit period, and identify a fraction of jitter on each end of the cycle, or alternately assign half the clock period to each bit period.

### Calculating Total Jitter

Because jitter is a composite of many things, it is important to keep straight the primary contributing components. For starters, there is typically jitter present on the incoming clock. Then, we will attach that to a CMT, which adds a specific amount of jitter, too. However, this does not add in the straight arithmetic sense. Peak to peak jitter adds in a root mean square (RMS) type calculation, as shown below.

#### **Peak-to-Peak**

$$JITTER_{TOTAL} = \sqrt{(JITTER_{INPUT})^2 + (JITTER_{SPEC})^2}$$

#### **Peak-to-Peak Deviation**

$$JITTER_{TOTAL} = \pm \sqrt{\frac{(JITTER_{INPUT})^2 + (JITTER_{SPEC})^2}{2}}$$

where

$JITTER_{INPUT}$  = The input period jitter, measured at the clock input pin of the FPGA

$JITTER_{SPEC}$  = The DLL clock output period jitter, as specified in the Spartan-3 Data Sheet for the appropriate output port

$JITTER_{TOTAL}$  = The expected total output period jitter

The expressions described above, permit a generalization, by noting that the output jitter from one level becomes the input jitter to the next. Because that will be squared, you will typically just take the square root of the sum of all the squares for the peak-to-peak jitter, and take the square root of the average of the sum of all squares for the peak-to-peak deviation. This sort of arrangement is shown in Figure 9-13, below.

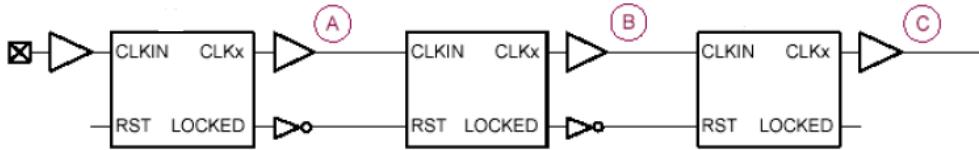


Figure 9-13 Calculating Jitter for Cascaded PLL or MMCM

Although we only touched on the topic of partial reconfiguration in Chapter 5, it is interesting to note that the Dynamic Reconfiguration Port (i.e. ICAP) allows the CMT to be dynamically programmed while the PLL or MMCM is in operation. More on this is in the various Virtex configuration documents and software literature. As you might guess, a “wizard” is provided to help with the adjusting.

#### Additional References

1. U.S. Patent #6,191,613 B1, Programmable Logic Device with Delay-Locked Loop, by David P. Schultz, Lawrence C. Hung and F. Erich Goetting
2. U.S. Patent #6,289,068, Delay Lock Loop with Phase Shifter, by Joseph Hassoun, F. Erich Goetting and John D. Logue
3. U.S. Patent # 6,775,342, Digital Phase Shifter by Steve Young, John Logue, Andrew Percey, Erich Goetting and Alvin Ching Comment: subsequent improvement patents by Austin Lesea, Andrew Percey, Steve Young and Trevor Bauer have improved DCMS to the point that they are (were) truly a pleasure to use. Check U.S. Patents #6384,647, 6,737,925, and 6,775,342
4. XAPP462 “Using Digital Clock Managers (DCM) in Spartan-3 FPGAs” by Steve Knapp presents a particularly thorough and insightful summary of using DCMS and includes a very nicely written description on how jitter affects user performance.
5. 7 Series FPGAs Clocking Resources User Guide, UG472 (v1.11.2), June 12, 2015

# Chapter 10 High Speed Transceivers: MGT, GTP and PCIe

## Introduction

The practice of taking parallel binary data and converting it to a serial stream for transmission over long distances is an old art. The first examples of serial transmission systems are those used in telegraphy (i.e. Morse Code). Later, circuit innovations led to the explosion of digital transmission systems in the 1960's and 1970's for long distance, and local telephony (T1 and E1 asynchronous transmission systems). Following that, digital microwave radio systems filled in the gaps where wires and fibers could not go.

Back-planes for intra-system communications resemble small versions of telephony digital transmission and switching systems. They were a natural consequence of passing large amounts of data from one processing element to another.

FPGA devotees lagged only slightly, but now lead the field in the adoption of fast serial link technologies. The Virtex 7 family capabilities support up 96 GTH transceivers for short range (~ 40 inches, depending on the speed up to 13.1 Gb/s) asynchronous (Syntonomous) links. See the glossary for definitions of the tonus and chronous terms used here. See Figure 10-1.

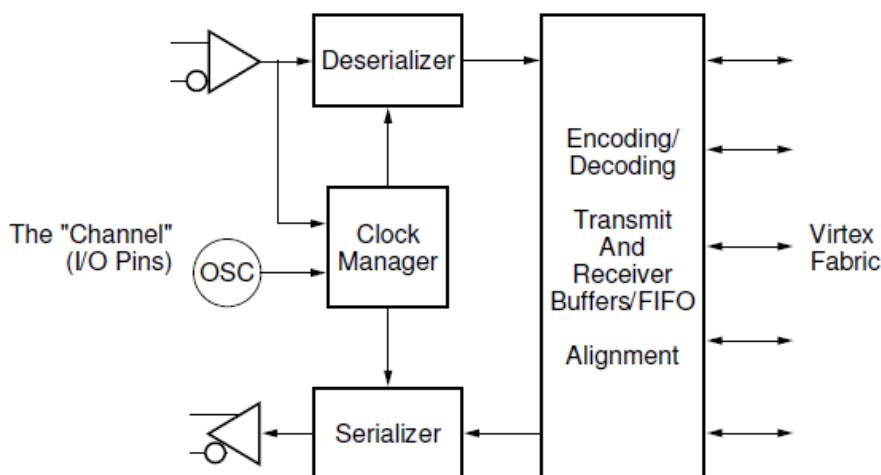


Figure 10-1 Basic High Speed Transceiver (SERDES)

This chapter divides the high-speed transceiver into multiple parts. First, we discuss how the transmit symbol, the bit, must be shaped to pass through the transmission channel. Next, we discuss the transmitter itself. Third is the channel discussion, which in a classical text, might be the first and most important section covered. Fourth, comes the receiver, followed by digital post-processing. The fifth section concerns timing and synchronization – as serial back-plane channels exist in several modes of timing and synchronization. Figure 10-2 captures some greater detail on the various sub blocks within the transceiver, just mentioned. Next we discuss errors, or impairment. After this, we cover issues in simulation and verification of the complete communications channel, which is an absolute requirement for the speeds we are talking about here. Finally, we tour the inside of the RocketI/O Multigigabit Transceivers and take a look “under the hood.”

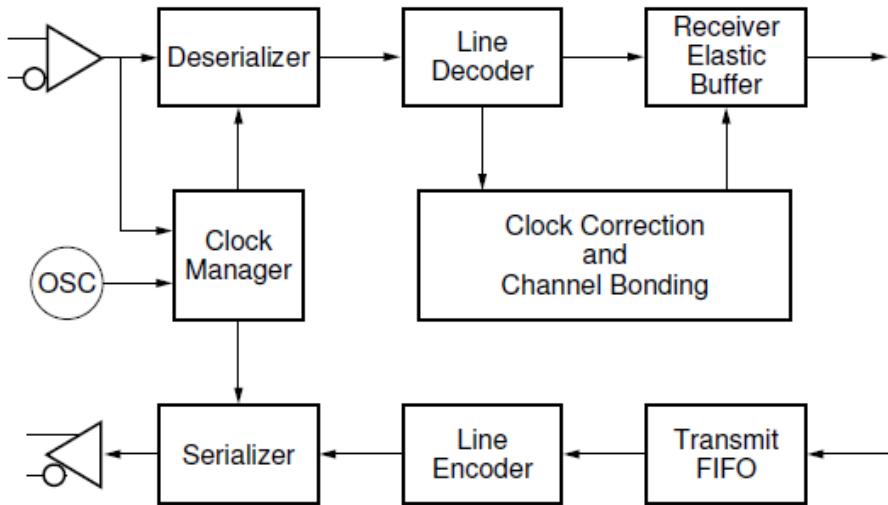


Figure 10-2 More Detailed Transceiver Block Diagram

### Pre-processing

Before the parallel digital data is serialized, there are some options to consider. Does the channel pass direct current, or must the signals be DC balanced? How many repeating symbols are allowed in a sequence before synchronization is lost (also known as run length)? Should the data be encoded so errors are automatically removed (corrected) upon reception?

#### NRZ: Balancing Ones and Zeroes

Non Return to Zero (NRZ) codes means that the serial data stream does not have a DC component. Zero DC component means there are a balanced, or equal number of ones and zeroes making up the serial data stream. Such a channel is equivalent to an AC coupled path with some high frequency behavior. The power spectral density of the serial stream needs to respect that high frequency behavior, and not have persistent low frequency components. Low frequency components are a direct result of having long strings of ones or zeroes, or long strings of very low-density ones or zeroes. Long consecutive strings can introduce a charging bias into the data channel that can alter the receiver's ability to distinguish ones from zeroes. In order to prevent arbitrary data from causing these problems, the transmit data is usually encoded before transmission. Figure 10-3 shows some NRZ data and Figure 10-4 shows some Return to Zero (RZ) data.

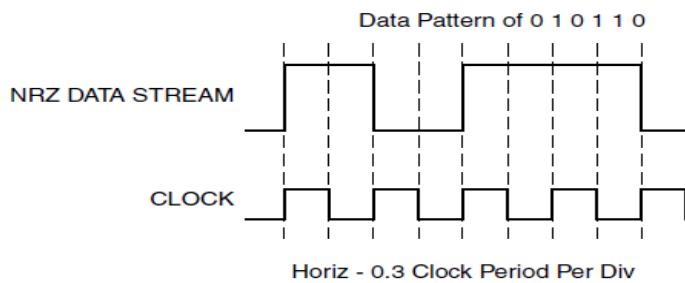


Figure 10-4 Non-Return to Zero Data and Clock

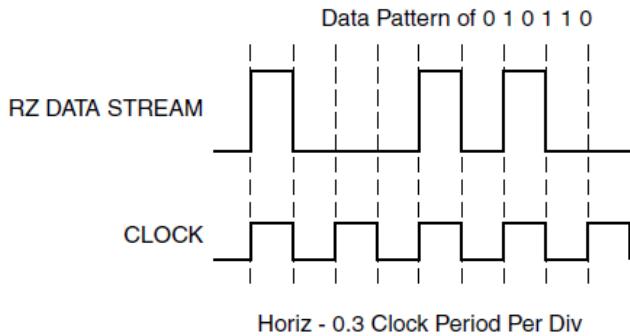


Figure 10-5 Return to Zero Data and Clock

### Encoding schemes: Scrambling, NbMb Coding and Substitution Coding

#### Scrambling

Scrambling takes the transmission data, and Exclusive ORs it with a pseudo random (repeating) sequence. These are constructed as linear feedback shift registers (LFSRs) as discussed in earlier chapters. As long as the serial data itself does not synchronize to the code (unlikely, but possible), the resultant data are half ones and zeroes. For example, if the data itself to be scrambled is the same sequence that would be generated by feeding this scrambler with all 1's, or all 0's, then the "scrambled" output will be all 1's, or all 0's. This is obviously not what we wanted, so recognition that this is possible often results in some other rule of formatting such that it is not possible to occur. Such a rule might be to make the frame length always even, which would prevent these sequences (which are always of odd length) from occurring.

#### NbMb Coding

NbMb coding is most common in data transmission systems. Increasing the number of bits sent is usually not an issue, to derive a benefit. In this case, imagine the 8b10b code, which replaces an eight bit byte, with a ten bit symbol. Using a lookup table, bytes are mapped to ten bit symbols meeting the run length requirement of the high pass channel. The run length requirement of the channel is the limit on the number of consecutive ones or zeroes to avoid introducing a DC offset into the physical channel.

These types of codes also allow for side channel communications for information of interest to the receiver. Side Channel information includes transmitter status, clock synchronization words, and other communications at a much lower rate than the data itself. See Table 10-1 for some 8b10b mappings.

ABCDE FGH (8b)	abcdei fghj (10b)
01011 000	010110 0100
01001 001	010011 0010
01001 100	010011 1001
01001 001	010011 0010
01001 110	010011 1100
01011 000	010110 0100
00100 011	010010 0110
00110 001	001101 0010

Table 10-1 8b/10b Coding Example

(Note: ABCDE FGH = 8 bit input code, abcdei fghj = 10 bit output code)

Substitution codes recognize a “bad” string of data, like a long string of zeroes, and replace them with preformatted data blocks. These are recognized at the receiver as a “replace this a long string of zeroes” with another symbol command. One example is B8ZS (binary 8 zero substitution) code, commonly used on T1 links to replace 8 zeroes in a row with a code that is recognized at the receiver. See Additional Reference 1.

8b10b is a very common code used today, by many different transmission protocols. Another industry standard is the 64b/66b coding, which uses an alternate strategy for arriving at the substitutions.

#### Error coding: FEC codes

In order to detect and correct errors at the receiving end, additional coding can be added to the sent data. A common method is to add a checksum so the receiver can determine if the received data block is in error. If an error is detected, some systems will request a data re-transmission. This is commonly known as the ACK/NAK channel for error control after acknowledge and negative acknowledge commands that get used. Unfortunately, as the error rate increases, the ACK/NAK channel eventually stops being able to effectively deliver information, as almost every packet transmitted has an error.

One improvement on ACK/NACK is to send additional bits, allowing the receiver to detect if a bit error has occurred, and to correct it, as well. A simple system is to organize the bytes into blocks and form byte parity for every byte (adds one bit per byte). Then, add a block parity for each bit of each byte in the block (append a final byte with parity at the end of the block that is the parity for all bit zeroes, bit ones, etc. of the bytes). This final byte is called a longitudinal parity check. Upon data block receipt, it is reorganized into its original form of bytes with parity in a block. If any one byte parity is incorrect, an error in that byte has occurred. The longitudinal parity at the end then indicates which bit is in error, permitting correction. Longitudinal parity check also fails if the error rate gets too high, or the block lengths are too long.

Other more powerful forward error correcting (FEC) codes are even better, but have a much greater algorithmic complexity. Some of the best of these are the Bose-Chaudhuri-Hocquenghem (BCH) codes. Those codes are well defined in the IEEE literature, or standard textbooks on high speed data communications. See Additional Reference 2.

## The Transmitter

The transmitter takes parallel data words that have been prepared for sending, serializes them, and shapes them to match the channel. To do this, you need a channel bit rate clock, typically derived from the byte or word clock by a phase locked loop (PLL). Symbol, or bit shaping is also needed, as the channel usually has requirements for near and far end cross talk, as well as needing to overcome high frequency losses.

### Transmit PLL, Clock Phase Noise and Transmit Jitter

The transmit PLL is typically provided with a reference clock from the parallel data clock domain. This clock is usually multiplied by 10 or 20 to create the transmitted bit rate clock. Any phase noise (jitter) on the reference clock directly adds to the phase noise or jitter in the transmitted data, reducing the channel error margin. A phase noise free clock, is desired. Although a PLL has a low pass jitter filter response (reducing high frequency jitter from the source clock), this filtering can't eliminate all jitter.

Some low frequency jitter passes through directly.

### Transmit Symbol Shaping (Equalization or Pre-emphasis)

The transmitted symbols, bits in our case, must be shaped before being sent. Symbols with excessive high frequency information (fast rise and fall times) may cause excessive cross talk. Symbols with too slow a rise or fall time may be attenuated by the channel high frequency roll-off, and be "un-receivable" at the far end. If too fast is bad, and too slow is bad, what is "just right"? It turns out that for any given channel length (which is the same as saying any given channel's high frequency loss), there is an optimal amount of high frequency energy that can be added to the transmit symbol for best reception. This is known as pre-emphasis, or equalization. The highest frequency components of the transmitted symbols are emphasized to exactly match the losses of that high frequency energy in the length of the channel. See Figures 10-5 and 10-6.

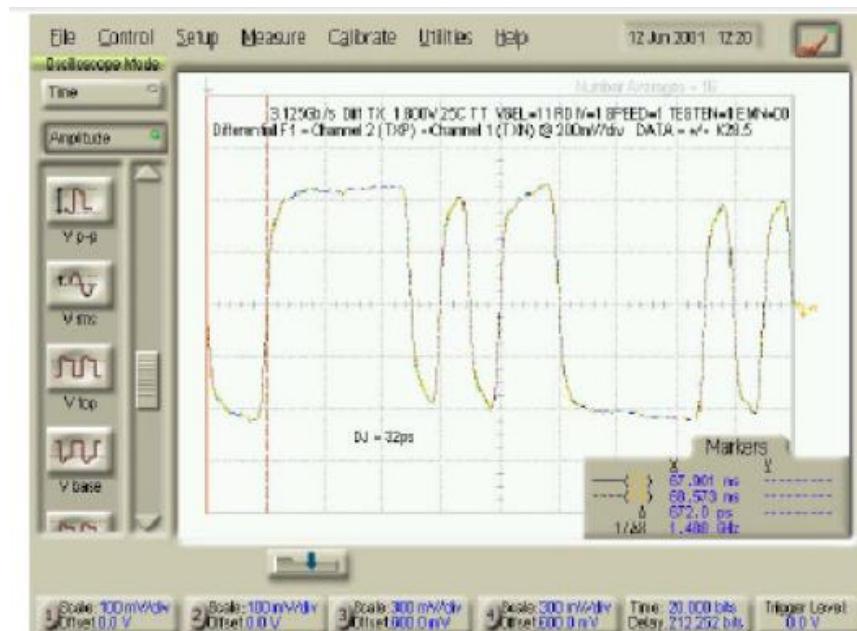


Figure 10-5 Symbol with No Pre-Emphasis

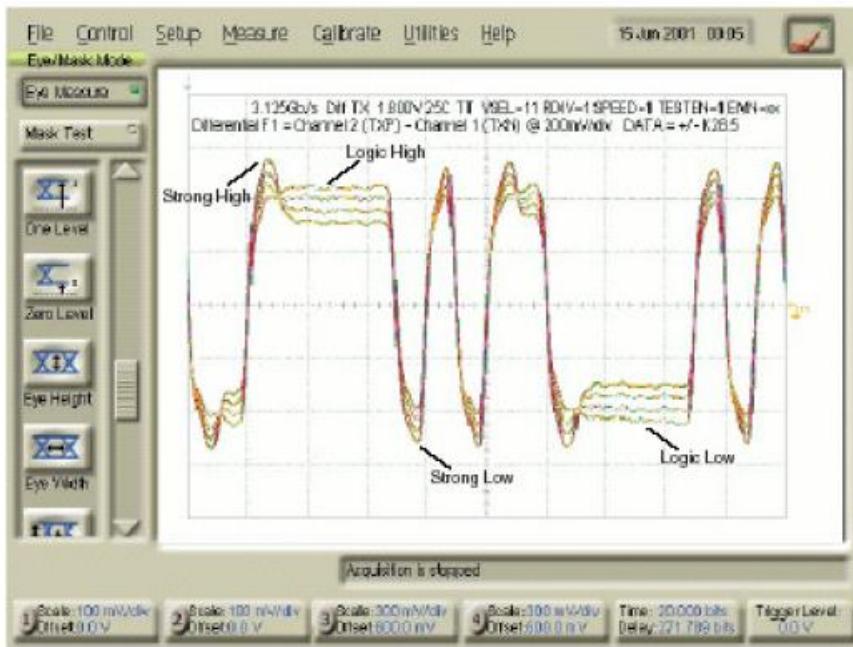


Figure 10-6 Symbol with Pre-emphasis

The signals right at the transmitter output appear distorted, but the resulting symbols at the receiver appear as if they did not encounter any channel losses. They are much cleaner than without pre-emphasis at the transmitter. Figure 10-7 provides greater detail on the relative drive strengths of the pre emphasized signals, at various times during pulse transmission, illustrating the strong high, logic high, strong low and logic low levels used.

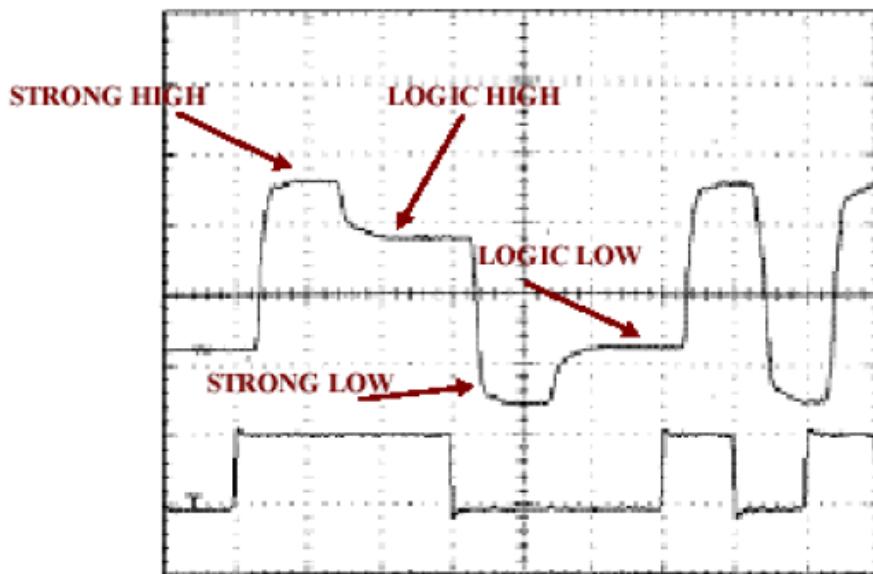


Figure 10-7 Zoom in on Pre-emphasized Pulses (bottom trace without)

Communications theory explains how a symbol only requires a certain bandwidth to be recovered error free at the receiver. Any more bandwidth is wasteful, and with any less bandwidth, the received data cannot be recovered without errors. This is known as the raised-cosine (or root-raised) matched Nyquist channel response. Figures 10-8, 10-9, 10-10 and 10-11 show both time and frequency domains of these functions.

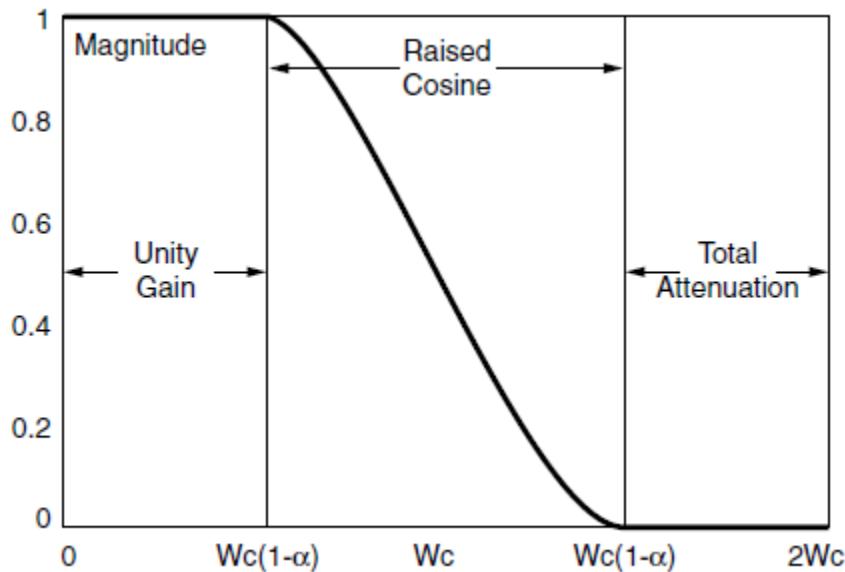


Figure 10-8 Idealized Raised Cosine Frequency Transfer Function

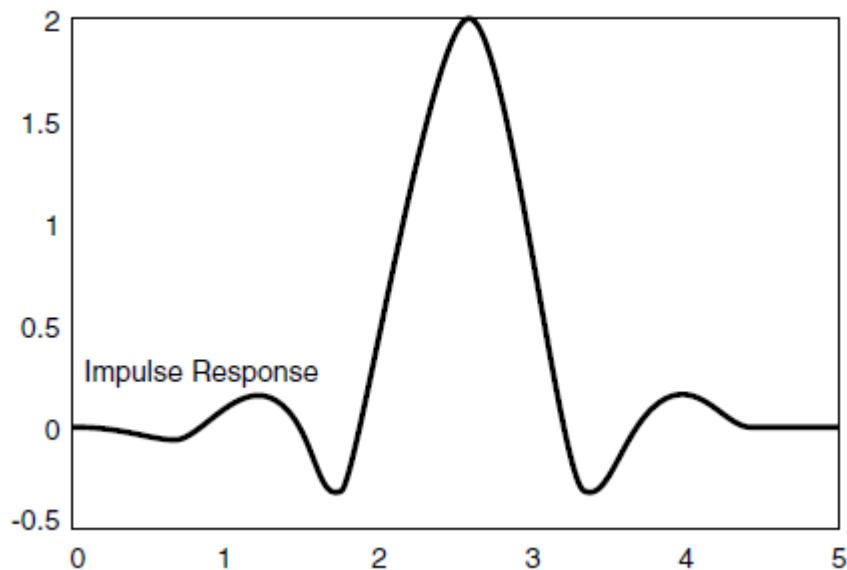


Figure 10-9 Idealized Raised Cosine Impulse Response

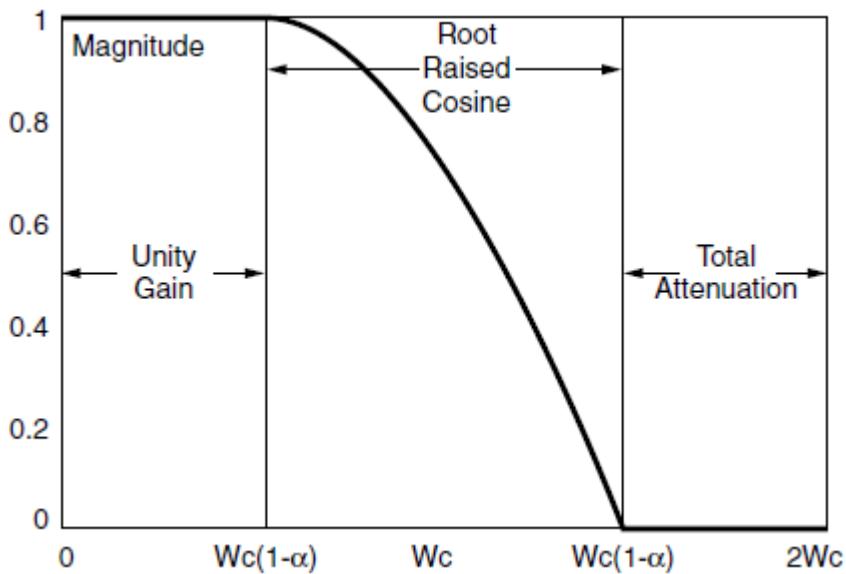


Figure 10-10 Idealized Root Raised Cosine Frequency Response

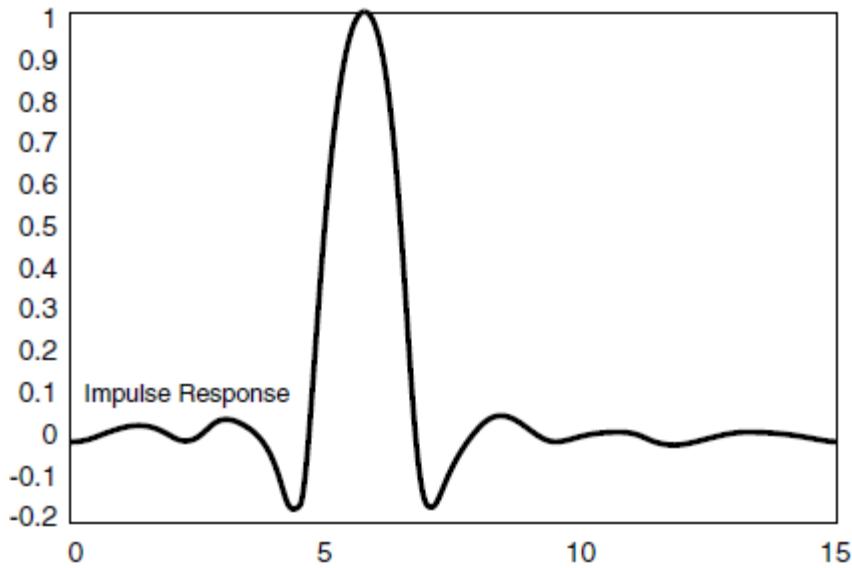


Figure 10-11 Idealized Root Raised Cosine Impulse Response

These are ideal functions to pre-filter symbols, before channel delivery.

### The Channel

Now that we have properly encoded and shaped a transmit symbol, we can send it over the back plane (the channel) to the receiver. We must also consider the frequency response (loss), the channel distortion, the nature of a differential transmission system, and finally, the transmission distance.

### Loss

The frequency response of the channel is a critical part of the system. In fact, the entire transmitter and receiver are designed for a specific set of channels, and not others. Backplane losses are such that the primary corruption is high frequency attenuation. Additionally, impedance mismatching occur at vias, and connectors unless the printed circuit board design is pristine. Even then, reflections, and miss-matches are common.

Channel loss has the effect of “smearing” transmitted symbols. For consecutive symbols, previous signal tails stretch onto successive signal rising edges, resulting in inter-symbol-interference (ISI). Figure 10-12 shows this effect. The top signal shows the ideal pulse on the left, with its corresponding “smeared” version in the channel, stretched out over multiple time slots. The bottom curve shows successive pulses, where the channel version blends together creating a signal that is much less digital looking.

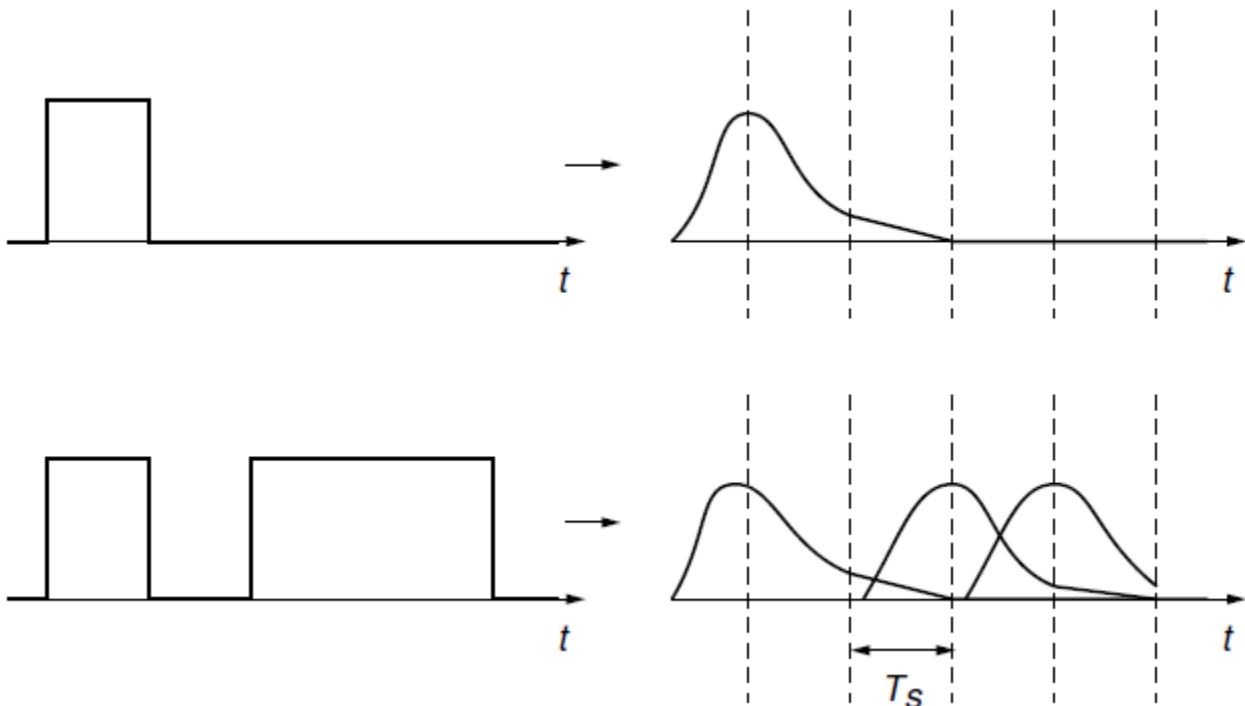


Figure 10-12 Simple Inter-Symbol-Interference (ISI)

#### Group Delay Distortion

ISI is a direct result of the channel group delay distortion. Group delay distortion is the behavior that causes lower frequencies to travel at a different speed over the channel, than higher frequencies. Although there are ways to pre-distort transmitted symbols to compensate for ISI, they are beyond the scope of this discussion. Such techniques are commonly used in digital microwave systems because radio frequency spectrum is too precious to be wasted for any reason. Figure 10-13 shows a frequency response for group (or envelope) delay distortion.

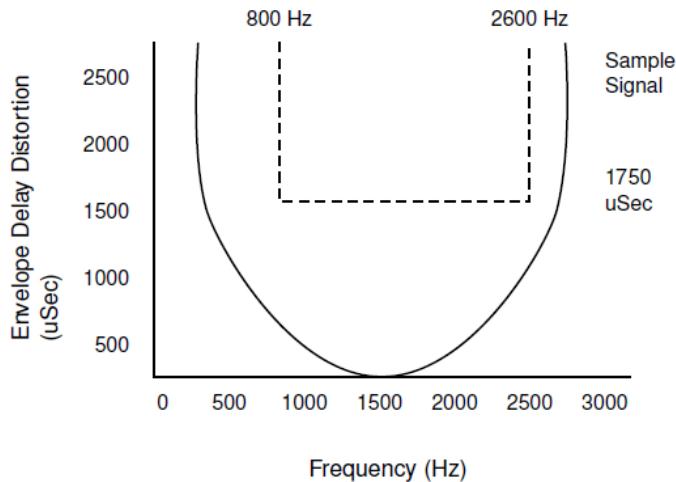


Figure 10-13 Envelope (Group) Delay Distortion

#### Differential transmission

The serial back-plane schemes we are discussing are exclusively differential in nature. Differential transmission systems can provide reliable, error free communications in noisy environments. Any common mode noise will be suppressed by the receiver, in a properly designed differential system. Often, engineers provide two separate 50-ohm transmission lines and call it “differential”. It isn’t. It’s simply two single ended 50-ohm transmission lines. Such a system has no differential noise rejection whatsoever, and is prone to many problems. See the top traces in Figure 11-6 in Chapter 11.

A properly designed differential transmission system is beyond the scope of this discussion, but is simply stated as two wires located the proper distance apart to conduct the signals to the far end. For a practical system, there are two main ways this is done, micro-strip printed circuit board lines, and strip lines. Figure 10-14 shows some of the trace topologies used to deliver differential signaling. Table 10-2 provides some guidelines on their relative merits.

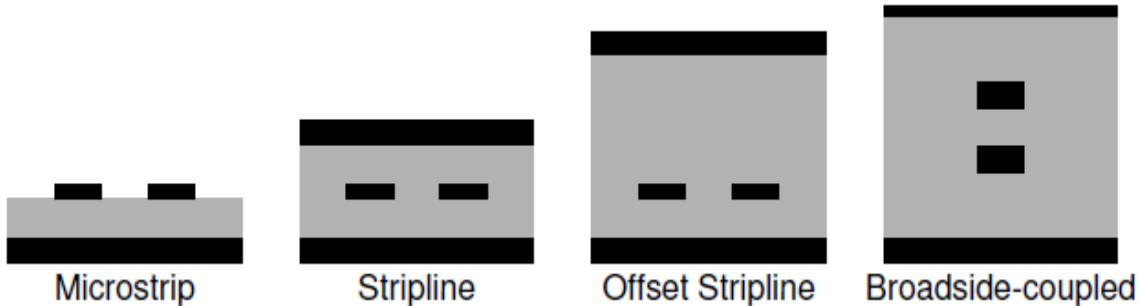


Figure 10-14 Various Differential Trace Topologies

Type	Pros	Cons
Microstrip	Less loss than internal traces	Only two layers (top and bottom) More susceptible to interference
Stripline	Better shielding More possible layers	More amplitude loss per inch in high frequency signals than microstrip
Offset Stripline	Useful if non-symmetrical Stack-up is needed. Can be used to limit the number of power/gnd planes.	If used to save layers, the offset area above the traces should be kept free of other traces and must be free of parallel traces.
Broadside-coupled	Very tight coupling	The broadside coupled is difficult to manufacture because of tight tolerances and it is not recommended for multi-gigabit operation.

Table 10-2 Relative Merits for the Differential Trace Topologies

Remember that any time the transmission line goes from a differential mode, to a single ended mode, there is opportunity for noise to be converted from the common mode to the differential mode. Differential mode noise will not be properly suppressed by the receiver, and must be avoided. Best results are obtained with a completely differential channel for the entire distance. See Figure 10-15. That being said, there still exists the possibility of ISI in a differential channel, as shown below in Figure 10-16.

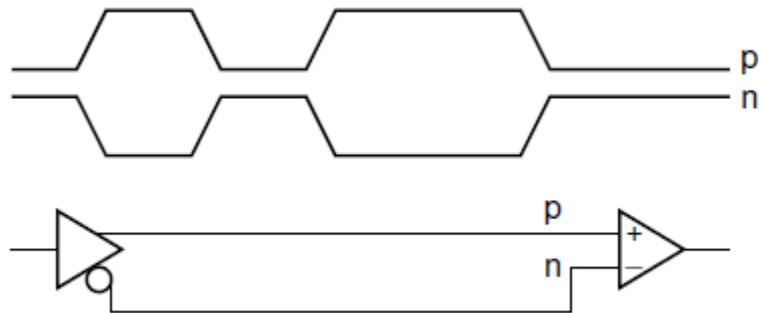


Figure 10-15 Differential Signals, and Complementary Driver/Receiver

Figure 10-16 shows the effect of a long set of logic ones driven in the middle of the top diagram. To the right of the bottom diagram, we see the inability of the line to properly discharge to effectively deliver logic levels due to ISI. This is not a single ended driver effect, alone. Differential circuits are also susceptible to ISI.

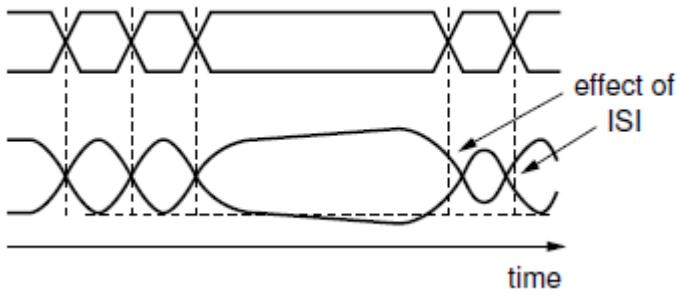


Figure 10-16 Differential Inter-Symbol-Interference

### How long is the channel?

A question often asked is “How long can a channel be?” Naturally, this is the wrong question. The question that should be asked is “How much loss and what signal shape is allowed in the channel?” The length is of little use: the manufacturer may have used an exotic printed circuit board material with thick copper traces to minimize dielectric and surface attenuation. All of these loss management tricks cost more money. It’s better to know the channel can tolerate a 6-dB roll-off at 3 GHz, for example, and then compare that frequency response with the actual system back plane.

Additionally, how many errors are being allowed for the loss or length given? It is of little use to quote the length (or loss) at which the channel has no more margin, and starts to have errors. Margin is defined as the difference between proper operation, and where operation ceases to be useful. Margin may be measured in dB when referring to the signal level, or in terms of unit intervals or picoseconds when referring to input jitter tolerance, or in terms of frequency when referring to input frequency offset tolerance. “6dB of receive margin” defines that the receive signal may vary by as much as 6 dB from some reference level, and still be recovered error free.

Of more interest than knowing where a channel is broken, is asking where the operation of the channel is still error free? Error free operation from +3 dBm to -15 dBm would imply a 18 dB margin. The upper and lower limits might also be useful not only for system loss, but if there were any active amplifiers (or repeaters) in the link.

### **The Receiver**

The receiver must do the following tasks:

1. Sense the incoming symbol.
2. Shape it to maximize the eye opening (details shortly).
3. Sample the waveform to determine a one or a zero.
4. Tolerate symbol sequences of the expected unchanging length.
5. Withstand channel jitter without producing errors.
6. Not transfer incoming jitter to the recovered clock, so jitter accumulates.

### Receive Symbol Equalization

Just like some equalization was good at the transmitter, in a perfect world receive equalization would be designed to exactly match one half of what was achievable in the transmitter. This is the “root-cosine” overall response that was referred to earlier, as the transmitter response should be equal to the receiver response. Each should be the square root of the ideal filter response having zero ISI. An ideal Nyquist channel

response is seldom achieved for a back-plane system, due to complexity. Instead, the receiver attempts to “open the eye” by applying some small amount of high frequency gain, similar to what was applied to the output symbols to compensate for channel loss. Eye diagrams are discussed later in the Error Management section.

Unfortunately, neither the transmit equalization, nor the receive equalization are the proper root-cosine shape, and neither do anything for group delay distortion. As a result, back-plane systems have substantial ISI contributions to the jitter, which are usually called the deterministic jitter, and directly detract from the overall link margin. Here margin is referring to difference between adding no jitter, and adding jitter until the onset of errors (where errors reach some defined rate, such as 1E-6, which is commonly used). Having tolerance to as much as 5 unit intervals of jitter occurring at some frequency would be considered as having a 5 unit interval margin at this jitter frequency.

Jitter is usually added to a standard data pattern. The data pattern itself will have its own deterministic jitter, which needs to be the same in order to make comparative measurements between different devices.

### Slicing (time and amplitude)

The other necessary function of the receiver is to recover the transmitted clock information, in order to synchronize recovered data bits at the receiver. To recover clock timing, a phase locked loop is synchronized to the frequency and phase of the incoming data stream. The receiver timing must track the transmitter low frequency noise and jitter, but suppress the high frequency jitter so as not to add jitter to the recovered clock. See Figure 10-17. The phase lock loops (PLLs) in most monolithic receivers are identical to those in the transmitter, having identical phase tracking responses.

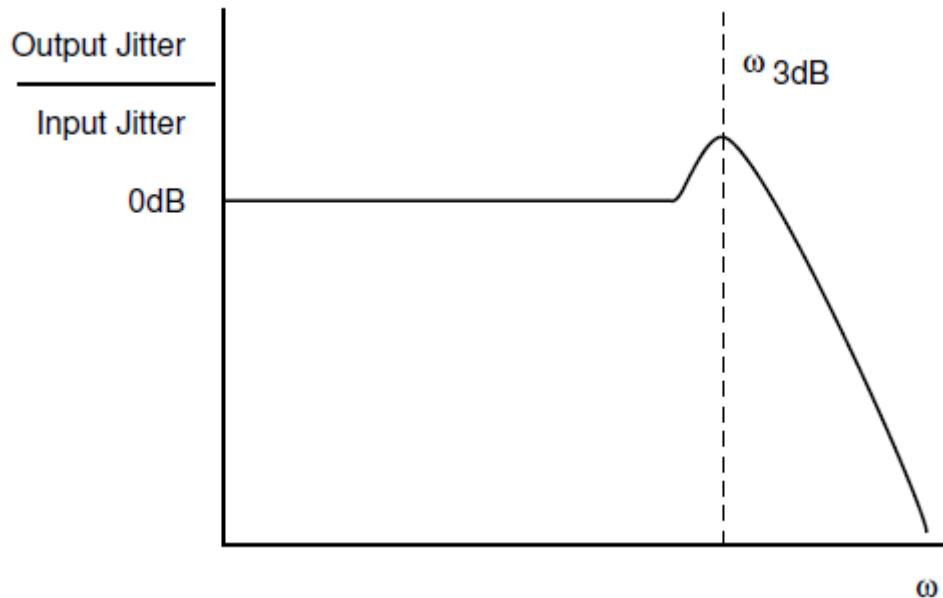


Figure 10-17 Jitter Transfer Curve

The number of received symbols arriving without recovered clock drift to the point of error when another symbol arrives, is directly related to the quality factor (Q) of the receive PLL resonator. If the PLL has a very high quality resonator (crystal, SAW filter) the receiver can persist without more information, remaining locked and error free. For simple back-plane serial transceivers, silicon RC, transistor, or LC resonators are used, having Q's (quality factor of the oscillator) far less than crystals or SAW filters.

### Run Length

A typical Q is on the order of five to ten. This implies that the number of like symbols in a row should be less than five to ten bits. As a rough approximation, the required Q must be on the order of the number of unchanging symbols desired. So, for a hundred consecutive zeroes, a Q of one hundred is typically required. If the format of the data stream is never allowed to drop below a 50% density in a few bits (like never more than five bits in a 8b10b system), then occasional runs of 50 or more bits in a row may be tolerated without error, even with a low Q receiver. The function of capturing the data and the clock is commonly known as Clock and Data Recovery CDR).

### Jitter Tolerance

It is desirable to track the low frequency phase noise and variations to obtain the lowest possible bit error rate, or greatest margin. In doing so, the transmitter's own phase noise can be ignored (tracked), and the incoming reference clock at the receiver can have more jitter at low frequencies (i.e., a cheaper clock source). The larger the value of the added jitter tolerated before the onset of errors (margin), generally the better the link will perform in the real world.

Phase Lock Loops are not jitter free devices. They operate with feedback and filters that continuously sample the incoming signal and make adjustments. Those adjustments become additional, small sources of jitter in themselves. Hence, the receiver PLL requires filtering to reduce jitter increases that arise.

### Jitter Transfer

If the PLL has insufficient jitter suppression, or worse yet, has jitter gain at some frequency, then a chain of such systems won't be stable, and errors will result. See Figure 10-17 at  $w = w_{3DB}$ . In many back-plane systems, jitter transfer is of little importance, and the recovered clock is used to load the receiver data into a local FIFO buffer. The system clock removes data from the FIFO, and recovered clock jitter is eliminated. This approach requires all transceivers be clocked from the same system reference (i.e., a Syntonomous system).

### Post-processing

After the receiver has distinguished ones from the zeroes, the bits are assembled into words for post processing. First, transmit coding is removed, and run length substitutions are reversed. Then, error detection and correction are performed.

### Untangling the data

Similar to the transmit coding, the data must be unscrambled, NbMb decoded, and any substitution codes replaced with the correct values. If a scrambler is used, there must be either a synchronizing sequence sent to recover scrambler/descrambler

synchronization. More often, a self-synchronizing descrambler is used. Unfortunately, scramblers can result in error multiplication as a single channel error may create multiple errors in the unscrambled data stream.

The NbMb decoding is also a table lookup process, just like the coding, but error multiplication can be eliminated. The decode table includes all “wrong” codes referenced to their nearest neighbor (best guess) recovered byte. For example, if the recovered word is 0xZZZh, the nearest byte code this matches is 0xYYh. It also could be 0xWWh. Thus half of the time, the error is actually corrected!

A substitution code replaces the received word with the desired word, which may actually be null information (no data at all).

### Removal of Run Length Coding

If there is additional coding to compensate for long strings of zeroes or ones, these also are recognized at this time.

### FEC Code Processing

After all of the previous processing is done, there may be error detection, or error correction processing to be performed, as well. This topic is well covered in any number of textbooks. Additional Reference three by Richard E. Blahut is particularly thorough.

### Synchronization and Timing

As already discussed, there are a few commonly used synchronization and timing models used in back-plane serializer designs. These are discussed below.

#### Single channel, Syntonus

By far the most common is the single channel - a transmitter connected to a receiver, where the byte or word clock is sent to both ends of every link. This clock provides reference to both the transmitter, and the receiver FIFO clocks. The syntonus approach prevents FIFO buffers from overflowing or underflowing (also known as slipping). The down side is that a low speed, low jitter clock must be sent to all transceivers used.

#### Single channel, Synchronous

One mode not used for back-planes, is the single channel synchronous case where the receive clock is in phase with the receive data. This requires a separate differential pair for just the clock, and is frequently not used in back planes.

#### Single Channel, Asynchronous

Often the receiver cannot have the same clock as the transmitter. This happens when communicating between two separate boxes, but may also occur in a system, by design. In the asynchronous case, each transceiver has its own clock, close in frequency (some parts per million different) to the others. In any such system, continuous data transmission eventually overflows the receiver FIFO. In any such system, bytes are added to compensate for the maximum clock frequency difference.

These are known as “stuff” bytes or padding and are ignored at the receiver end. In this way, the transmitter never overflows the receiver FIFO.

### Multiple Channel, Channel Bonding

When more than one channel is required for higher capacity, it is desirable to synchronize transmitted words across several transmitters to multiple delivered words, at the receivers. Virtex serial multi-gigabit transceivers (MGTs) provide this feature by using channel bonding. 8b10b codes are used for communicating information from the transmitters to the receivers to synchronize the recovered words into the correct order. Figure 10-18 shows the basic idea. The top diagram shows four words in the consecutive time order of TSRQP, where T exits the transmitter first, followed by S, then R, Q and P in succession. As the words enter the channel, they encounter different electrical distances and arrive at the receiver. The trick here is that the symbol S is inserted into the data stream as a marker, so that the receiver will know when coordinated data is supposed to be isolated. The diagram shows that the R symbol on lane 1 is behind the rest of the R symbols in the other three channels, after recognition of S. The solution is to modify the delivery of the Read clock by one data cell for “lane 1”, to track the correct word boundary, and establish a “bond” between the channel lanes, accordingly.

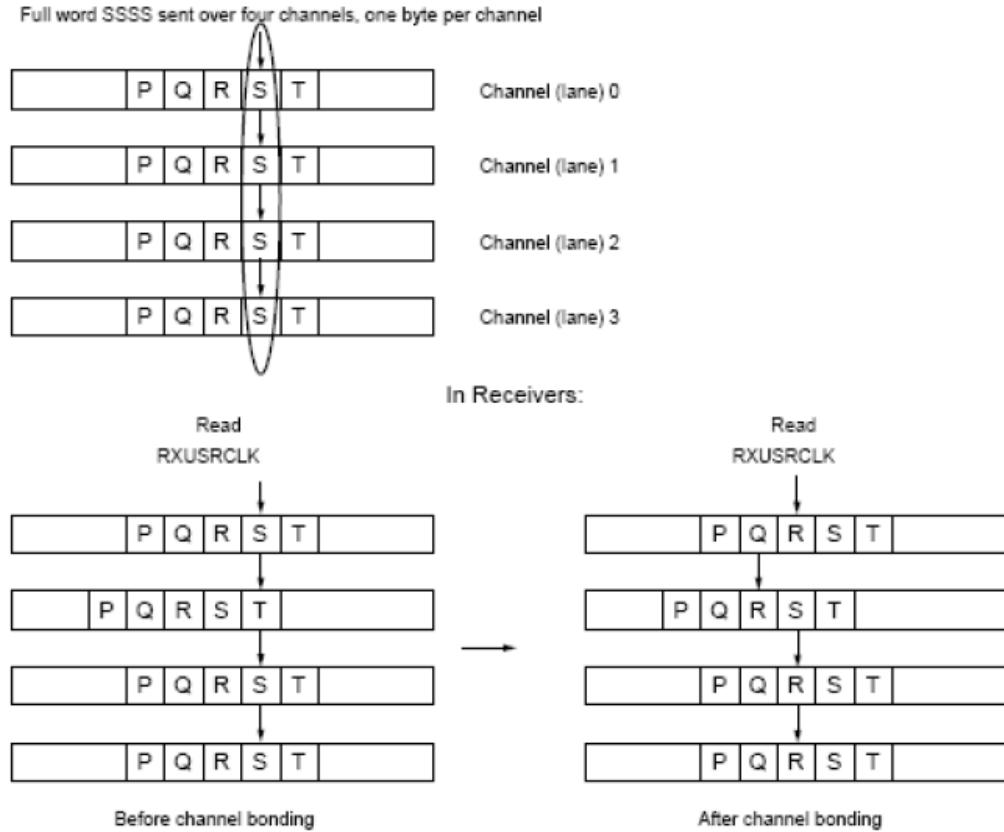


Figure 10-18 Channel Bonding

### Error management

In any serial channel, there will eventually be a bit error. Bit errors are usually a bad thing, and if there is no error detection or correction present (as is often the case),

errors are preferred not to happen at all. This is a real problem: there will be errors (inevitable), but no errors are desired, and no error detection or correction is present. The solution to this problem is a compromise. The link is designed to have no bit errors under normal operation, and the only time bit errors occur is when an extraordinary event occurs. This is the philosophy chosen by Xilinx, as well as most telephony equipment providers. It is a commonly used trade-off. The negative is that the transmitter, channel, and receiver must adhere to strict guidelines to assure they have sufficient margin to be error free. Simulation of the channel is strongly advised to prove the system will work, once built.

### Bit Error Rate (BER) as a Metric

Bit error rate is one of the most commonly misunderstood metrics. By the previous discussion, if errors are designed to happen only in extraordinary cases, what meaning does a BER of 10-12 have? In fact, it has no meaning at all. A link that actually has a “normal” or dribbling bit error rate of 10-12 is officially broken. It does not meet the criteria outlined above for a link without error detection or correction.

BER statistics requires tens, if not hundreds, of errors to characterize. This is simply because a rate defines a slope. A slope is not accurate until it consists of many points. In the communications industry, generally samples of 10 points (about -50%, +180% for 95% confidence) to 16 points (about +/-20% for 95% confidence) are considered the absolute minimum to determine an instantaneous rate. Such a rate is only accurate for the very short period (10 or 16 samples of errors). To properly characterize the BER, one requires many more errors to have the confidence that you are observing all of the degradation that is present. A BER of 10-12 on a 3.125 Gb/s link will take a long time just to get one error, about 5.33 minutes. You will likely lose your patience (or someone will interrupt the experiment) before enough sample points are taken for adequate accuracy.

Typically BER testers assume a normal distribution (Gaussian), and make simplifying assumptions. These assumptions may grossly over or under estimate the link performance. Either way, the equipment is very expensive.

### Use of Eye Patterns

By observing the eye pattern over time, which is an oscilloscope trace of the data triggered by the recovered clock, we can see how much margin there is in both amplitude, and time. The margin present is a gross indication of link performance, but as most engineers will say, they can “tell a good eye pattern when they see it”. The eye pattern could have a 10-3 bit error rate and still appear perfect, so it is required that both error rate and eye pattern tests are done. Figure 10-19 shows a sample eye diagram, for a single bit Unit Interval.

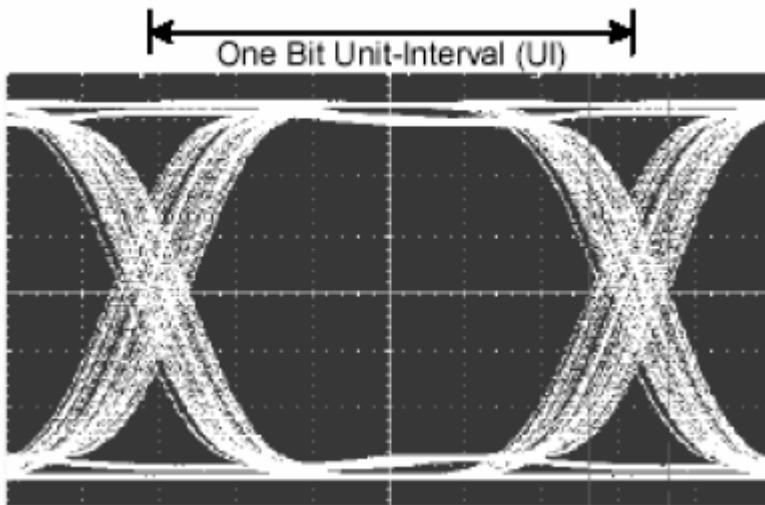


Figure 10-19 Eye pattern

An eye pattern is usually taken on a sampling or digital oscilloscope, and overlays the traces of multiple signal cycles. Figure 10-20 shows how the composite of several such signals results in a trace with fuzzy edges.

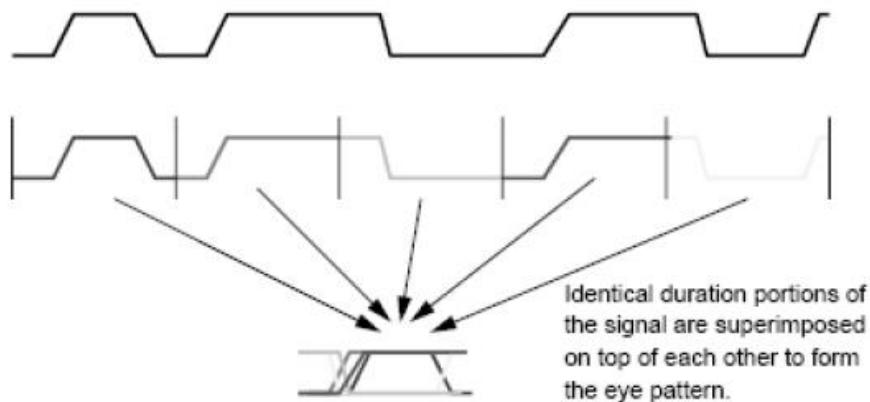


Figure 10-20 Eye Pattern Composition

Uncertainty in the various rising and falling edges due to jitter and ISI can cause the “eye” to close up. Figure 10-21 shows a nice crisp eye diagram, for an NRZ signal (eye wide open).

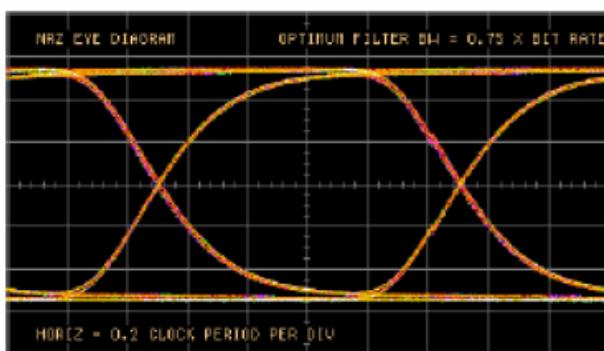


Figure 10-21 NRZ Signal Eye Diagram

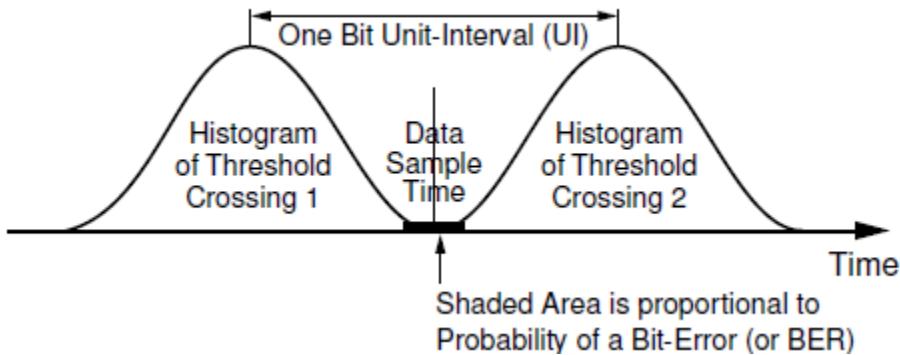
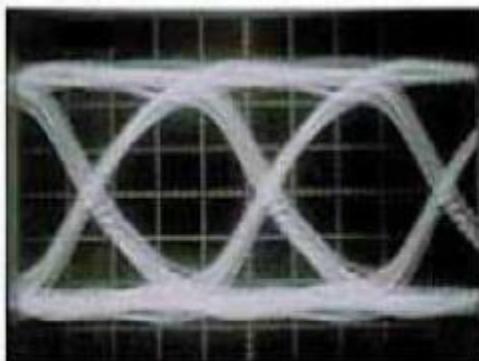
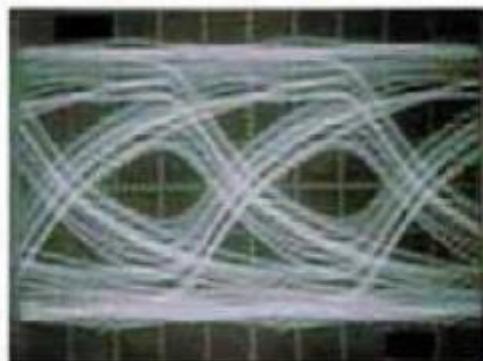


Figure 10-22 Distribution of Threshold Crossings for a One IU Eye Diagram

Figure 10-22 shows a histogram taken over many samples of a target signal. Note the shaded, overlapping area of the bimodal distribution, indicating a region of Bit Error.



Pre-Emphasis



No Pre-Emphasis

Figure 10-23 Eye Diagrams, with and without Pre Emphasis

Figure 10-23 shows a received eye diagram, with pre-emphasis and without pre-emphasis. In particular, note that the left side diagram is more “open”, displaying a crisper UI, for a bit, than the right side shows. The best receiver strategy is to open the eye, then attempt to sample the bit as near the center, as possible, providing ample setup and hold time for bit capture into the receiver buffer/shifter.

Much can be determined from eye diagrams. Figure 10-24 shows recognition of ISI and jitter, by identifying various thicker regions of the eye waveform.

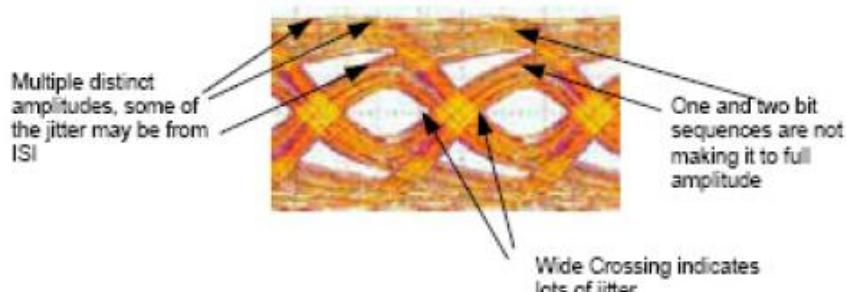


Figure 10-24 Eye Diagram Details Indicate Signal Quality

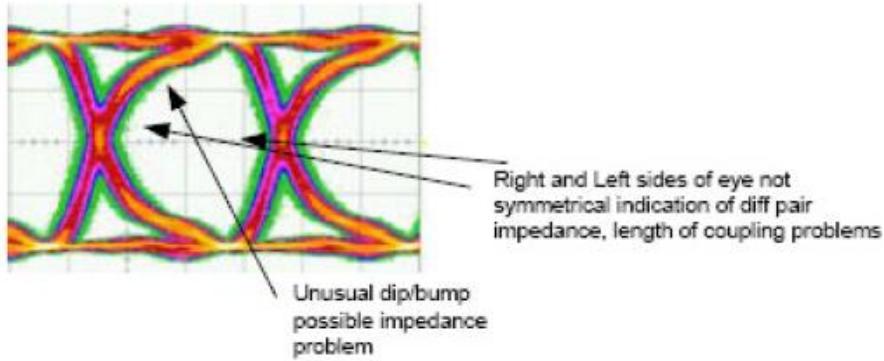
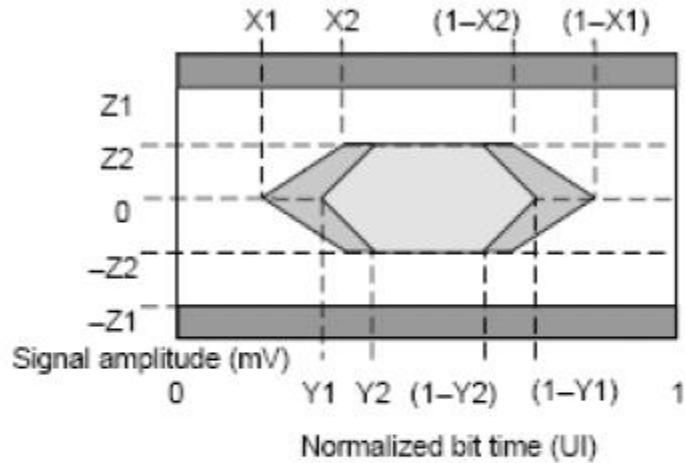


Figure 10-25 Eye Diagram Details Indicate Channel Quality

Figure 10-25 indicates asymmetric eye diagrams can determine channel qualities like impedance or driver/receiver issues. Finally, Figure 10-26 indicates some of the required/expected eye quantities for various industry standard signaling protocols. It gives the required voltage levels and positions for various eye “points of interest.”



Data rate (Gbps)	X1 (UI)	X2 (UI)	Y1 (UI)	Y2 (UI)	Z1 (mV)	Z2 (mV)
3.125 (XAUI)	0.180	0.305	0.300	0.425	800	100
2.5 (Infiniband)	0.205	0.380	0.325	0.500	800	87.5
1.25 (802.3z)	0.231	0.356	0.375	0.500	800	100

Figure 10-26 Eye Masks for Various Standard Protocols

Additionally, tests of an unstressed link are not very interesting, as no errors should ever occur. Stress tests, performed as the clocks are jittered, while the length is made longer than intended, and with noise added to the system allows the designer to gain confidence in the eventual system link performance. Stress testing is a good indicator of how robust the channel is, and exposes potential weakness early, for correction.

## **Simulation and Verification**

For any link, full logic simulation is first done to ensure that the bytes sent are the same bytes as those received. This level of functional simulation exposes all errors regarding mode selection, coding options, scrambling, and channel bonding. Once the logic is known to be good, a circuit level simulation should be done to assure that the analog performance of the link is met with sufficient margin. Xilinx works with third party simulation experts to supply appropriate circuit models for full board level simulations at various modeling levels. Xilinx design personnel create both IBIS, IBIS-AMS, and SPICE simulation models, and work with customers to properly assess their design capabilities and needs.

Spice Models of both the transmitter and the receiver, are used with (encrypted) spice netlists of the back plane and its connectors. Xilinx has pre-run tens of thousands of simulations with various combinations of solution, the simulation effort is reduced to perhaps a final verification of the system once the printed circuit board parameters are extracted or measured. We will discuss a little more on verification, shortly.

For specific details on the transceivers, see the references, below.

### **Additional References**

1. High-Speed Serial I/O Made Simple: A Designer's Guide, with FPGA Applications, Abhijit Athavale, available at: <http://www.xilinx.com/publications/books/serialio/serialio.htm>
2. Algebraic Codes for Data Transmission, Cambridge University Press, 2003, Richard E. Blahut.
3. UG024, RocketIOTM Transceiver User Guide UG076, Virtex 4 RocketIOTM Transceiver User Guide
4. High-Speed Digital Design: A Handbook of Black Magic, PTR Prentice Hall, Howard W. Johnson and Martin Graham
5. High-Speed Signal Propagation: Advanced Black Magic, PTR Prentice Hall, Howard W. Johnson and Martin Graham
7. UG196, Virtex 5 RocketIOTM GTP Transceiver User Guide UG197, PCI Express Endpoint Block User Guide
8. 7 Series FPGAs GTX/GTH Transceivers User Guide, UG476 (v1.11.1) August 19, 2015

# Chapter 11 SelectIO, DCI and ChipSync

## Introduction

Change is the hallmark of programmable logic, and the I/O pins are where very large change is most evident. The single requirement of any electronic data transaction is that the sending and receiving partners understand each other's logical voltage levels. Because designers must frequently interface between devices (microprocessors, memories, EPROMs and logic) of differing voltage standards, FPGA devices with multiple I/O banks became a necessary requirement for quick product development. Virtex FPGA devices handle a multitude of different voltage standards, with developed I/O technology going beyond simple level translators.

**SelectIO** is the original set of I/O standards provided in Virtex architectures, along with double data rate (DDR) flip flops. As this was supplemented with additional standards, it became also known as "SelectIO Ultra", but most users simply call it SelectIO, meaning the set of standards for the particular Virtex Family FPGA of interest.

**Digitally Controlled Impedance** (DCI) is the capability for an I/O pin to be self-terminating, without requiring external resistor packs. This is important, because it is very difficult to get just the right impedance physically close enough to the I/O pins of large pin count packages, to be most effective. This patented circuitry is unique to Xilinx, and has found its way into the hearts and minds of thousands of designers.

**ChipSync** was added to the Virtex-4 Family, by adding additional I/O time management capabilities. ChipSync is also mentioned in the chapter on Memory Interfacing, as it makes interfacing to DDR SDRAM and QDR-II SRAM substantially easier than before.

We look at these three topics in greater depth in this chapter, and note that they frequently interact with each other. At the chapter close, you will find out where to learn more.

## SelectIO

The original Virtex supported about 16 different voltage standards. Virtex 7 supports about 60. A lot has happened since the initial introduction of Virtex FPGA devices, as technology has collapsed its transistor sizes and voltage levels. See Table 11-1. IO Standard Virtex 7 series.

LVTTL (Low Voltage TTL)  
LVCMS (Low Voltage CMOS)  
LVDCI (Low-Voltage Digitally Controlled Impedance)  
LVDCI\_DV2  
HSLVDCI (High-Speed LVDCI)  
HSTL (High-Speed Transceiver Logic)  
HSTL\_I and HSTL\_I\_18  
HSTL\_I\_12  
HSTL\_I\_DC1 and HSTL\_I\_DC1\_18  
HSTL\_II and HSTL\_II\_18  
HSTL\_II\_DC1 and HSTL\_II\_DC1\_18  
HSTL\_II\_T\_DC1 and HSTL\_II\_T\_DC1\_18  
DIFF\_HSTL\_I and DIFF\_HSTL\_I\_18

DIFF\_HSTL\_I\_DCI and DIFF\_HSTL\_I\_DCI\_18  
 DIFF\_HSTL\_II and DIFF\_HSTL\_II\_18  
 DIFF\_HSTL\_II\_DCI and DIFF\_HSTL\_II\_DCI\_18  
 DIFF\_HSTL\_II\_T\_DCI and DIFF\_HSTL\_II\_T\_DCI\_18  
 HSTL Class I (1.2V, 1.5V, or 1.8V)  
 Differential HSTL Class I  
 HSTL Class II  
 Differential HSTL Class II  
 HSTL\_II\_T\_DCI (1.5V or 1.8V) Split-Thevenin Termination (3-state)  
 SSTL (Stub-Series Terminated Logic)  
 SSTL15\_R, SSTL135\_R, DIFF\_SSTL15\_R, DIFF\_SSTL135\_R  
 SSTL18\_I, DIFF\_SSTL18\_I  
 SSTL18\_I\_DCI, DIFF\_SSTL18\_I\_DCI  
 SSTL18\_II, SSTL15, SSTL135, DIFF\_SSTL18\_II, DIFF\_SSTL15, DIFF\_SSTL135  
 SSTL18\_II\_DCI, SSTL\_15\_DCI, SSTL135\_DCI, DIFF\_SSTL18\_II\_DCI,  
 DIFF\_SSTL\_15\_DCI, DIFF\_SSTL135\_DCI  
 SSTL18\_II\_T\_DCI, SSTL15\_T\_DCI, SSTL135\_T\_DCI, DIFF\_SSTL18\_II\_T\_DCI,  
 DIFF\_SSTL15\_T\_DCI, DIFF\_SSTL135\_T\_DCI  
 SSTL12, SSTL12\_DCI, SSTL12\_T\_DCI, DIFF\_SSTL12, DIFF\_SSTL12\_DCI,  
 DIFF\_SSTL12\_T\_DCI  
 SSTL18, SSTL15, SSTL135, SSTL12  
 Differential SSTL18, SSTL15, SSTL135, SSTL12  
 SSTL18, SSTL15, SSTL135, or SSTL12 (T\_DCI) Termination  
 HSUL\_12 (High Speed Unterminated Logic)  
 HSUL\_12 and DIFF\_HSUL\_12  
 HSUL\_DCI\_12 and DIFF\_HSUL\_12\_DCI  
 HSUL\_12  
 Differential HSUL\_12  
 MOBILE\_DDR (Low Power DDR)  
 LVDS and LVDS\_25 (Low Voltage Differential Signaling)  
 Mini-LVDS (Mini Low Voltage Differential Signaling)  
 PPDS (Point-to-Point Differential Signaling)  
 TMDS (Transition Minimized Differential Signaling)  
 BLVDS (Bus LVDS)

Table 11-1 Summary of Various Virtex 7 series Family I/O Standards

The number of standards is substantial, so we won't detail them all in this chapter. Of particular note in Table 11-1 is the presence of the suffix "DCI" at the end of some standards. Those standards support the Digitally Controlled Impedance (DCI), which was introduced with Virtex-II and will be discussed in more depth shortly. Specific Xilinx product datasheets detail the various standards, but we will look closer at three important ones – SSTL, HSTL, and LVDS. Those three standards have had huge impact on Virtex FPGA family sales.

### Stub Series Terminated Logic (SSTL)

As suggested in the introduction, voltage standards have changed much. It's sometimes hard to understand the choice to shift away from the "good old" standards, like 5volt and 3.3 volt logic, but various manufacturers approached new problems in different ways. For instance, DRAM manufacturers optimized memory processes for very high density and speed. Doing that with small feature CMOS meant they would be dealing with very thin Silicon Oxides along with the small transistor gate widths.

Architecturally, they chose to operate with both clock edges, giving double data rate devices, but they had to select their I/O architecture to work with the new processes and not forsake the speed they had attained. They arrived at the set of standards called Stub Series Terminated Logic (SSTL).

SSTL is a set of standards developed to operate at different voltage swings, which are denoted by the number at the end of the SSTL prefix. For instance, SSTL3 is a 3.3V version, where SSTL2 is a 2.5V version. The “dash number” like -I or -II dictates the current drive associated with that specific standard. Large capacitive loads typically need greater current to charge and discharge PCB capacitance. Figure 11-1 shows some selected SSTL families of standards. Note the termination resistances, the terminating voltage (VTT) and the reference voltage, VREF.

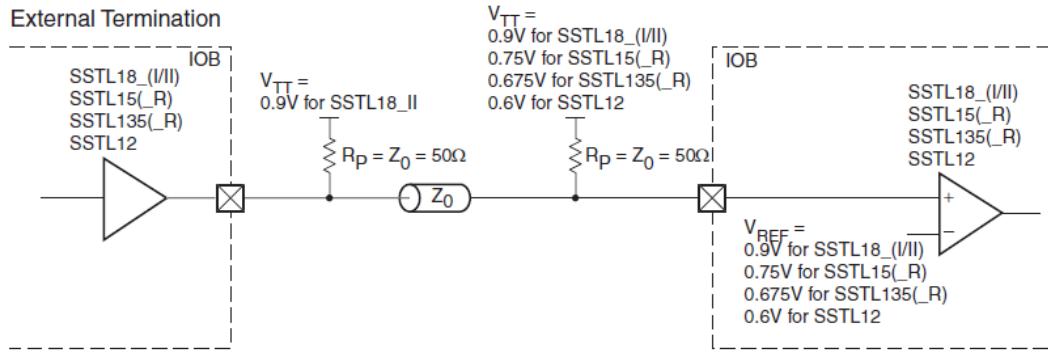


Figure 11-1 Selected SSTL Families of Standards, Non-DCI

The VREF distinguishes the switching point between a logic one and zero, which is half the switching range of the driver. For both cases, note that VTT is the termination voltage, and typically matches the reference voltage, VREF.

### High Speed Transceiver Logic (HSTL)

High Speed Transceiver Logic (HSTL) is a switching standard common to several types of memories. Operating in the 1.5V range, it is a single ended standard that is received by a D.C. biased input comparator constructed from a differential receiver with a constant VREF on one input leg. There are several classes of HSTL, with different current drives, VREFs and termination impedances. Figure 11-2 shows HSTL families of standards. There is not much difference in the input or output structure, except for setting the VCCIO appropriately, externally attaching a VREF, and attaching termination as dictated by the standard. Current drive is one factor that makes a difference. HSTL is an EIA/JESD 8-6 standard, originally sponsored by IBM.

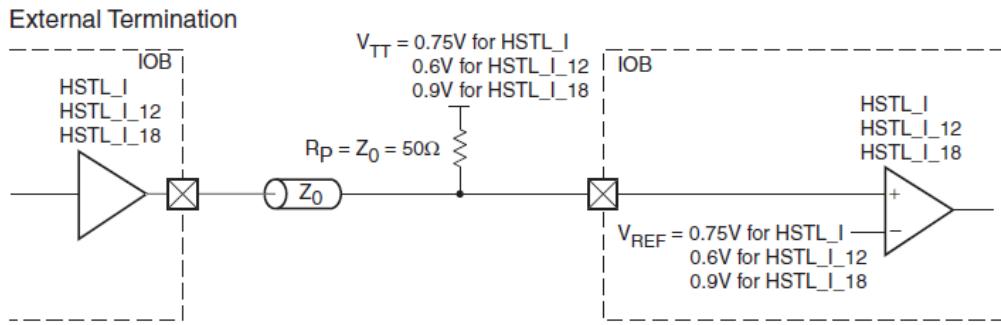


Figure 11-2 HSTL-I Selected Families of Standards, Non-DCI

### Low Voltage Differential Signaling (LVDS)

Low Voltage Differential Signaling emerged as the chosen standard for backplane interfacing in high speed data communications equipment over recent times. It's a CMOS standard providing many benefits of earlier bipolar emitter coupled logic (ECL), but operates at positive voltages. There are several versions of LVDS, depending on what the target application requires. For instance, Multi-Drop LVDS targets broadcasting of data to multiple receivers, where Bus LVDS targets bidirectional transmission. Each modifies the basic LVDS termination and voltage swings, to a degree. Let's first examine the basic LVDS structure, shown in Figure 11-3.

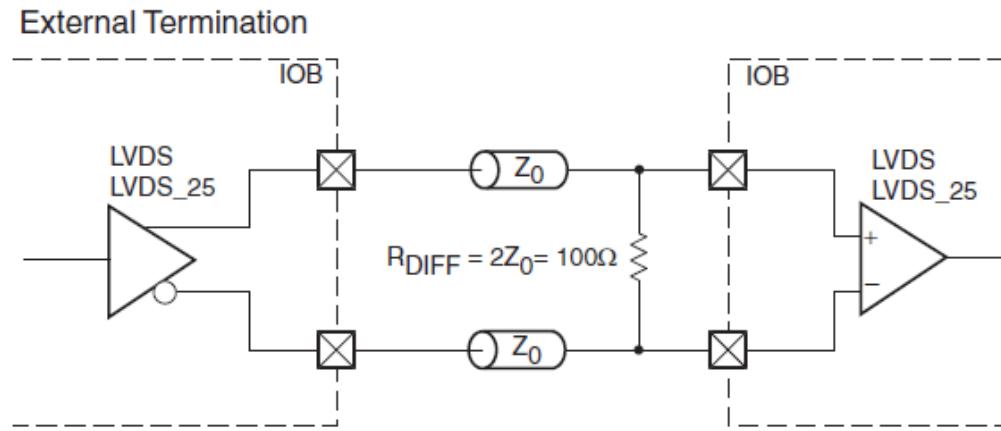


Figure 11-3 Typical point to point LVDS

As shown, the current drive requirements are 3.5 mA in both drive and sink directions, with receiver termination of 100 ohms across the differential inputs. VDD is 2.5V for this standard, with a nominal common voltage of 1.25V, with the output high as 1.38V and the output low voltage at about 1.03. This provides the differential value of 350 mV developed across the input termination, as shown.

Note that LVDS devices may also operate at 3.3 or 1.8 volts if so designed. A 3.3 volt powered device is compatible with the Xilinx FPGA 2.5 volt powered LVDS bank.

A small voltage swing delivers high speed, but comes at the price of noise immunity, which is addressed by the common mode rejection ratio (CMRR) of the differential signaling. The idea behind differential signaling is that common noise sources corrupt both the positive and negative asserted signals the same. This can be due to adjacent

signal or clock coupling, or power supply noise. Either way, noise corrupts both legs of the transmitter fairly uniformly. The receiver senses both signals, subtracts them (eliminating the noise) to produce the output signal. Still, to gain best benefit, it is incumbent on the board designer to carefully layout both signal paths in uniform, “strip-line” fashion to maintain the benefit and obtain maximum data rate. Failure to match signal paths can skew the common noise contribution, and aggravate the noise, versus cancel it.

A common speed for LVDS is 622 Mb/sec. For backplane transmission, multiple gigabits of bandwidth can be achieved by supplying multiple such channels – say ten channels, delivering cumulative 6.22 Gb/sec. Figure 11-4 shows Spice simulations of differential pulse transmission and burst transmission. The top traces show differential inputs overlapping on the absolute top, and the received pulse below that. Note the insertion of differential mode “glitches” passing through to the output. This was mentioned in Chapter Ten. Traces below that show 622 Mb/sec complementary inputs, and corresponding outputs in the lowest trace. Xilinx application notes XAPP 230, 231, 232, 233 and 243 detail various aspects of several LVDS types.

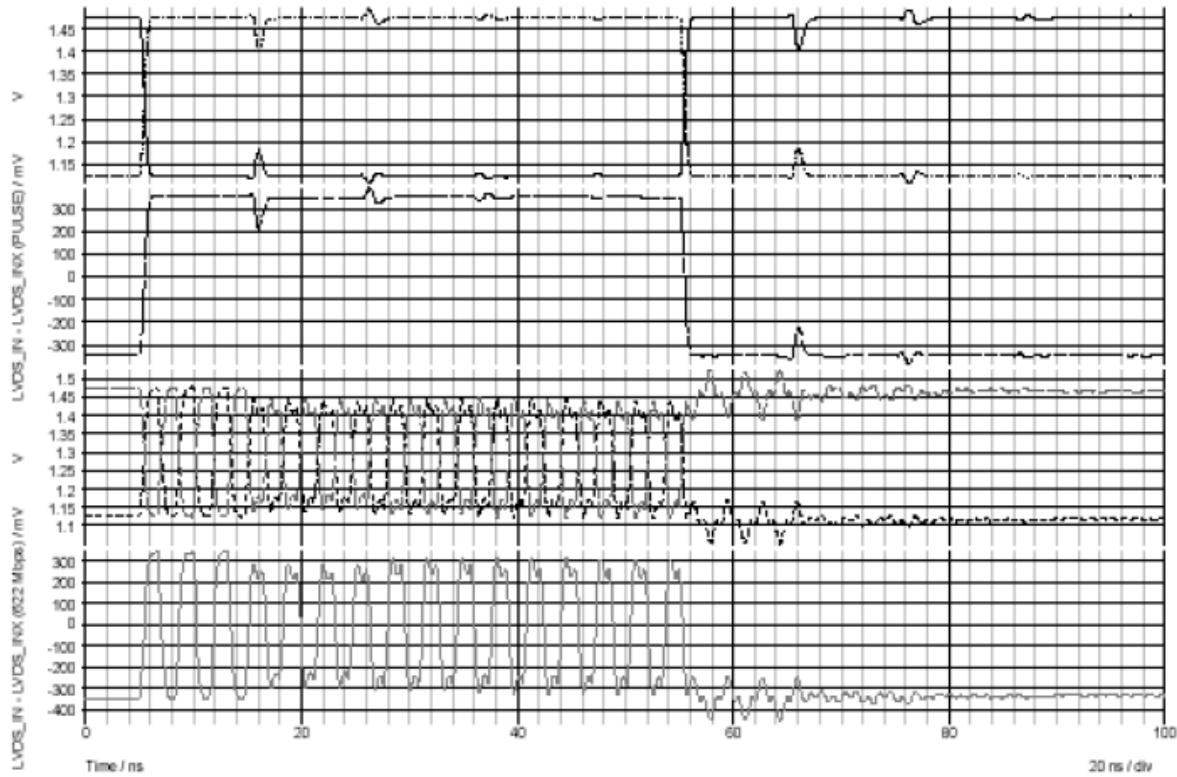


Figure 11-4 Spice Simulations of LVDS Pulse and Burst Transmission

Multi-Drop LVDS is shown in Figure 11-5, where a single terminating resistance is shown across the ensemble of receivers. The signal levels are the same for Multi-Drop, as standard LVDS, but termination conditions may vary depending on the PCB layout and trace impedances. A more realistic depiction is shown in Figure 11-6, with twenty outputs being driven, and a distributed termination network of resistors.

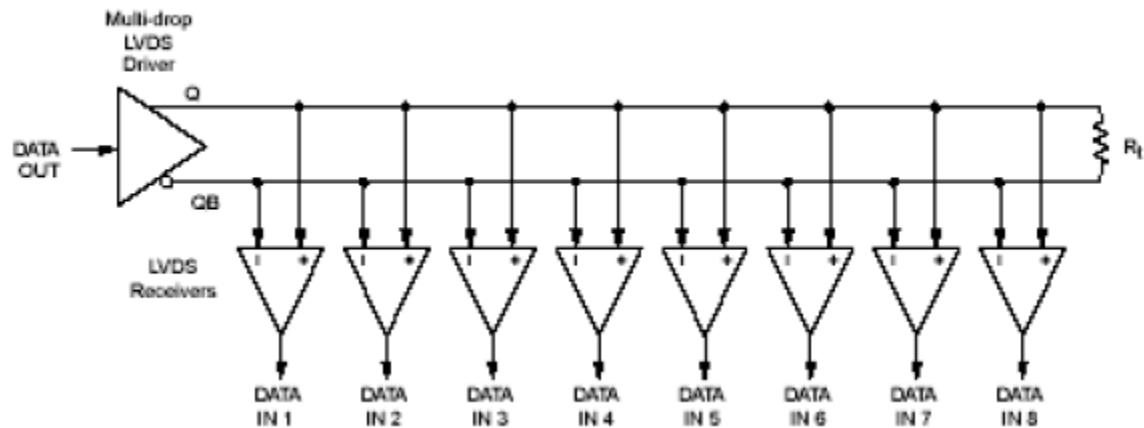


Figure 11-5 Typical Multi-Drop LVDS Arrangement

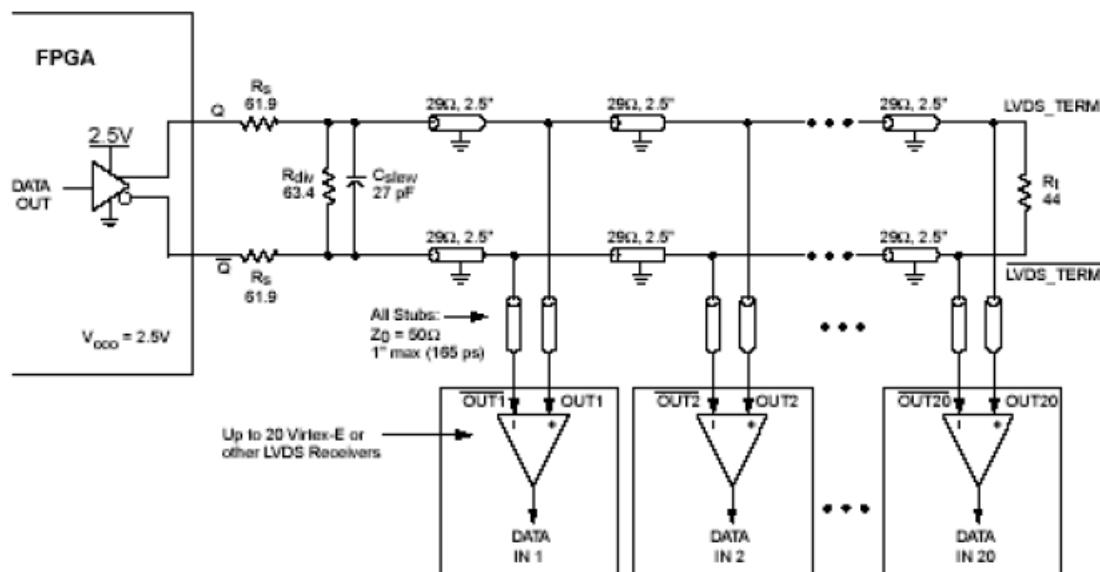


Figure 11-6 Twenty Loads of Multi-Drop LVDS

XAPP 231 shows the time delays associated with the various received signals in a high distribution structure as shown in Figure 11-6

### Putting it All Together

We won't be showing the full blown Virtex 7 I/O structure, at this point, but simply the basic Virtex type structure. Figure 11-7 is taken right out of the I/O patent for the original Virtex family, and you can see multiple represented voltage standards shown on the various boxes. The data I/O pad is in the upper left, and the large box on the right is the JTAG circuitry.

Simplistically, many of the "push pull" standards can be driven from the same output, with the VCCIO set to the appropriate limiting voltage. Input threshold detection is a bit trickier, but single ended receivers needing a VREF can generally be derived from the same transistors. Fast receivers like LVDS and LVPECL can't work that way, as both differential inputs need to handle high speed. There is substantial transistor

sharing among the standards, to save area. Nonetheless, this gives an idea of how much circuitry is behind an I/O pin in the simplest family! Details are provided in Additional Reference 1.

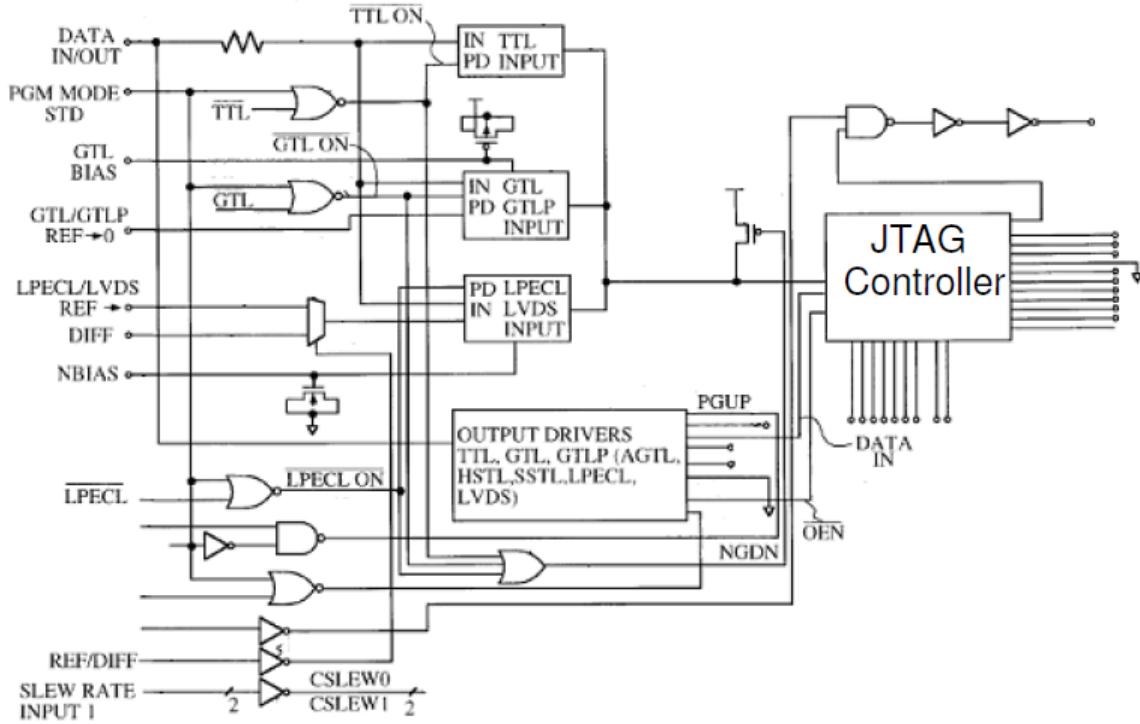


Figure 11-7 Virtex Multi I/O Standard Pin

Chapter Four on design software discussed some issues the Place and Route tools deal with while using the I/O banks. When designing, it is the user's responsibility to identify which signals are to be associated with the sets of I/O standards. Designs with hundreds of pins make this a daunting task. To aid users, there is substantial documentation on constraints and a special tool to handle pin and constraints to ease the handling.

This completes the basic introduction to SelectIO, but be aware that it also includes the logic capability of double data rate flip flops, which we now inspect in greater detail. Figure 11-8 Bidirectional I/O Pad with DDR Capability Figure 11-8 shows the flip flop logic available at every I/O pin, for all Virtex families. The upper right shows two input flip flops, with both "D's" attached to the pad. Separate clocks (CK1, CK2) permit different clock attachments (typically a clock and its complement). One D input can capture data on its clock rising edge, and if so chosen, the other can capture data on the complement clock (effectively the original's falling edge). The output flip flops are shown multiplexed into the output buffer, and the 3-state control flip flops are multiplexed into the 3-state of the output buffer.

Although there is no specific multiplexer select signal shown, it is usually described as under the control of the clocking signal of the flip flops attached on the multiplexer inputs. This is not quite the situation. A closer model of the behavior is that the multiplexer is merged into a specialized flip flop. In Chapter One, we saw a flip flop (Figure 1-8) having the D input entering a master latch on a clock rising edge, then the master contents transferred to the slave latch on the clock falling edge. In Chapter

Six, we mentioned how a LUT cell (Figure 6-1) could be created from a latch strobed by a self-timed clock. The DDR flip flop builds on those ideas. It has a multiplexer, attaching to a single slave latch, to form the structure. In Figure 11-8, DATA1 enters the circuit, and forwards through two pass gate transistors under the control of a strobe produced by CLK1, time skewed to arrive on the gates of the two pass gate transistors. The right side of the two pass gate transistors connect to the D Input of a simple latch. Six different data sources attach to that D Input. DATA1 and DATA2 are the synchronous data, arriving on CLK1 and CLK2 edges (for rising and falling events). The other paths are from set/reset circuits or configuration initialization bits (not shown). In summary, the Nor-inverter structures produce pulses, the pass transistors multiplex and the output inverters latch, to create the DDR flip flop.

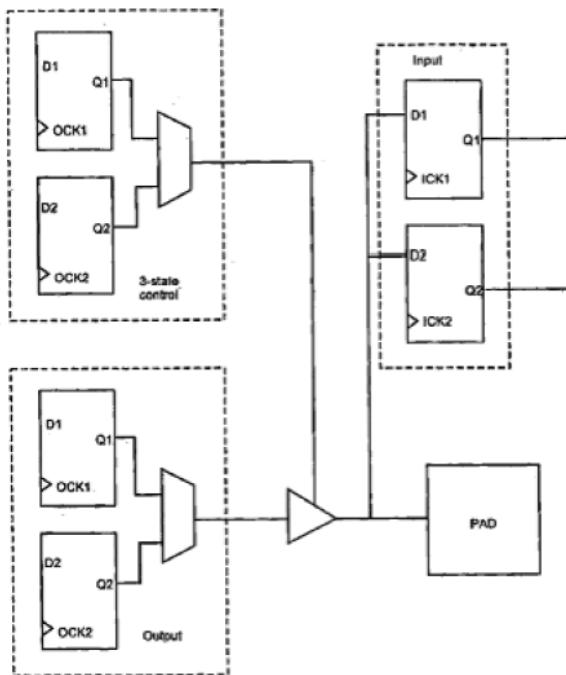


Figure 11-8 Bidirectional I/O Pad with DDR Capability

Clearly, all this detail exists before the level translation shown back in Figure 11-7.

### **Digitally Controlled Impedance (DCI)**

Earlier sections showed the elaborate termination required to successfully operate high speed I/O pins. Key to effective operation is locating these impedances close to the appropriate driver/receiver to mitigate reflections. Physically, this is hard to do with hundreds of I/O sites on ball grid packages. There isn't enough package edge "real estate" to accomplish the task. The other tack of attaching resistors to the underside of the chip, is also tough. Even customized resistor packages still presented problems. As often occurs, Xilinx designers found a better way – terminate within the package.

The resulting solution is called Digitally Controlled Impedance (DCI). The basic idea is shown in Figure 11-9.

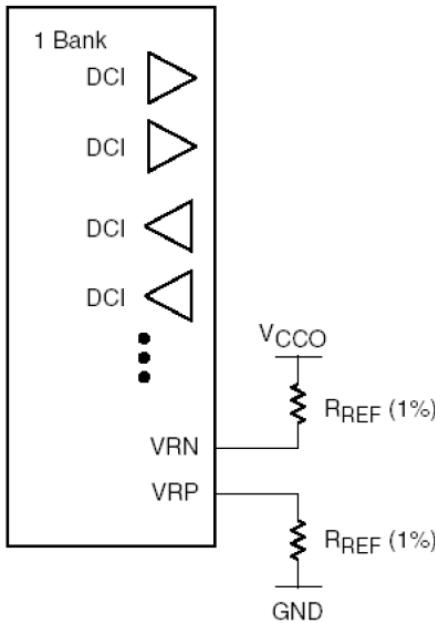


Figure 11-9 DCI in Virtex-II

DCI operates in a banked fashion, where I/O pins are provided for external attachment of a pair of reference resistors. It's true that external resistors are still required, but the resistors attach externally to  $V_{CCO}$  and ground, and are selected to match the characteristic impedance (ie,  $Z_0$ ) of the PCB. Internal circuitry measures this value, and tunes an impedance matching circuit inside the chip, that is present at every I/O pin, to track the external resistors. This is a substantial reduction in the number of required external terminations. Only one resistor pair per bank does the trick. The main DCI operation is done by selecting pull up and pull down transistors within the FPGA to arrive at an equivalent impedance internally tied to the pin. Figure 11-10 shows a bit of the high level construction.

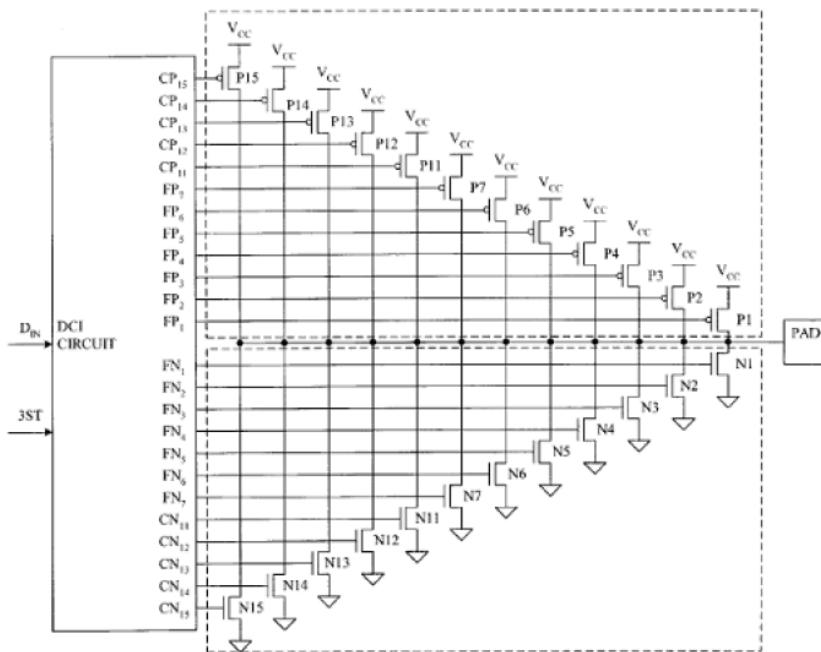


Figure 11-10 Impedance Selecting Transistor ‘Tree’

In Figure 11-10, the N and P transistors have their gates driven from sites labeled for coarse (C) and fine (F). Transistors will mimic resistors under the control of the DCI control circuitry, as we shall see shortly.

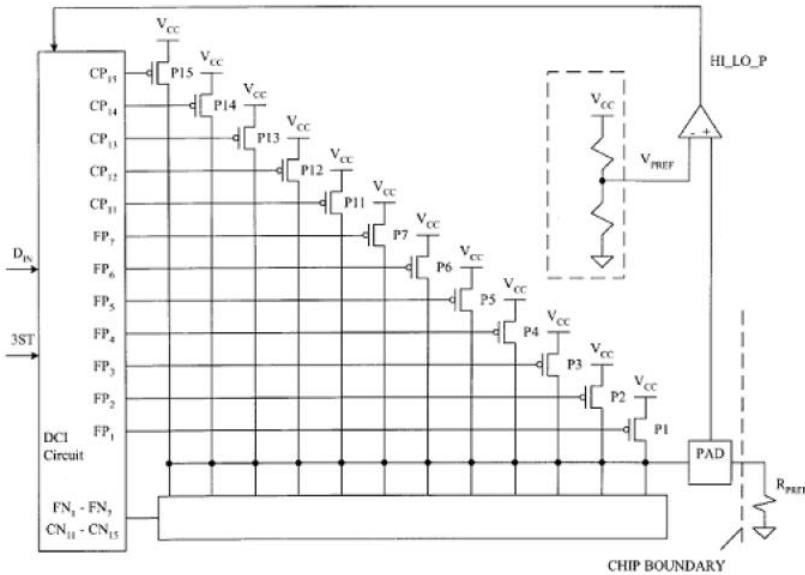


Figure 11-11 N-Channel Transistor Selection

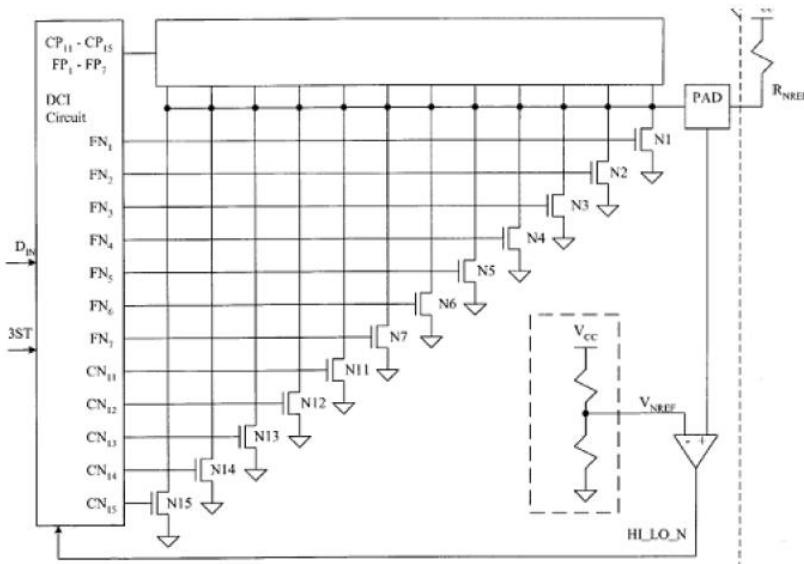


Figure 11-12 P-Channel Transistor Selection

Figures 11-11 and 11-12 show how internal comparators match against the external termination resistance to select an appropriate transistor configuration from the P-Channel and N-Channel transistor “trees,” creating appropriate matching values inside the FPGA. This results in an identical selection setting for each I/O pin so selected within the same I/O bank. Hence, all pins on a given bank need only access one pair of external matching resistors. The selection process is a digital set of operations.

The comparison selection process initially occurs right after configuration completes, and an outline of the steps is shown in Figure 11-20. In spirit, the process resembles the DLL time delay process, where a coarse value is first selected and followed by fine grain choices successively converging on a final value. In the DCI process, coarse impedance transistors are first switched in, then finer transistors complete the process. Typically, this is done with transistor sizing to vary the impedance (you probably forgot that “transistor” was a contraction for the words “transfer resistor”).

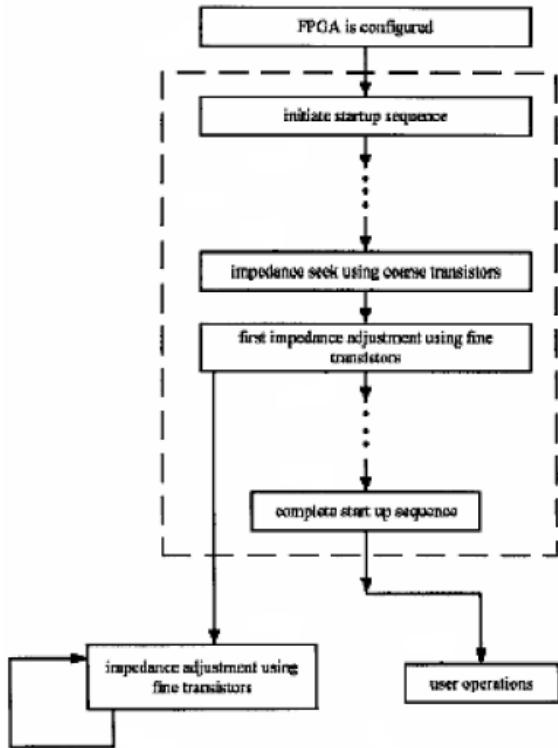


Figure 11-13 DCI Impedance Matching Strategy

The resulting impedance selection gives very decent I/O behavior. Figure 11-14 is an example of DCI in and out of action. Dark traces show the signal with DCI engaged and light traces show it without. It clearly makes a difference! See XAPP 659 for further details and examples. See Additional References 2 and 3 for more detail.

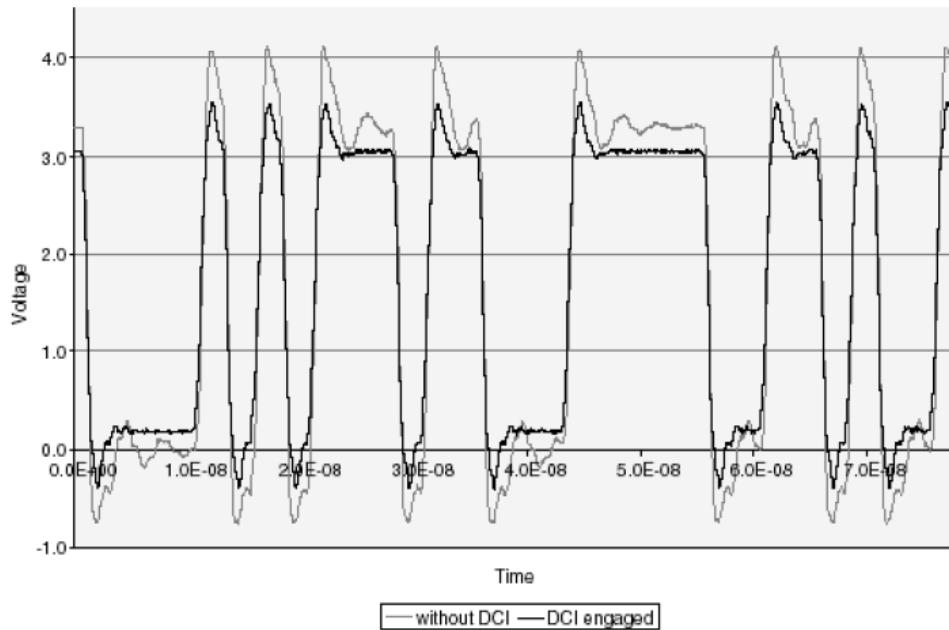


Figure 11-14 SSTL2C2 With and Without DCI Termination

## ChipSync

In order to distinguish the I/O voltage, impedance and drive capability from the other timing based features, the ChipSync title was created for the additional advanced capabilities present in Virtex 4. The roots can be seen in XC4000 FPGA devices and early Virtex FPGA devices that included input delay selection as an option. ChipSync includes Idelay, Bitslip, ISERDES and OSERDES capabilities. We consider them one at a time.

### Idelay

Idelay is the ability to add incremental delay to an input signal, arriving from a pin. As you might expect, it does that with a silicon based delay line (i.e. consecutive inverter/buffers). The question is: how do you dictate what the delay will be? We saw methods for delay selection in Chapter Nine, and as you might guess, similar methods apply. In the case of Idelay, an external clock source defines a time frame (i.e., a clock period) to measure against. Idelay circuitry is most accurate, with a 200 MHz clock applied to the appropriate input pin, attached to an Idelay controller. The controller captures one period of the 200 MHz clock, by isolating its rising and falling edges.

Once captured, the number of delay elements for the period are defined. The controller adjusts the Idelay voltage with a servo, tuning the period to have 64 delay taps. The number of delay taps are then known. If exactly 200 MHz is applied, there will be 64 subdivisions of the period, to be selected among. Any resulting tap delay is about 78 picoseconds. You don't have to supply an external 200 MHz clock, as a DCM can "up multiply" a slower clock, and deliver it to the Idelay controller.

Virtex ASMBL architecture is distinctive among Virtex family architectures. I/Os appear in columns distributed across the chip. Within each column, is a set of I/O pins, and also that column's Idelay controller. Figure 11-15 shows where the Idelay block lies with respect to the other input circuitry.



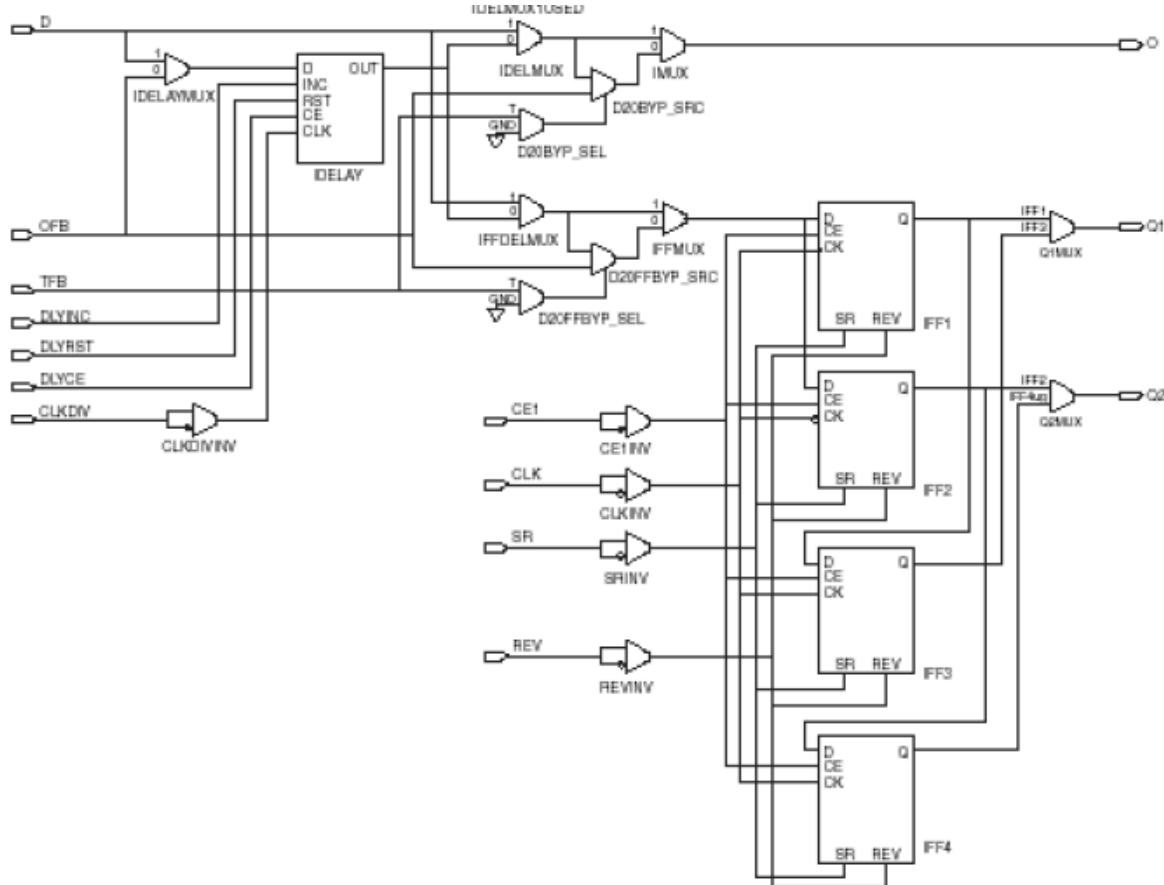


Figure 11-15 Input Logic Block Diagram

The Idelay block symbol itself, is shown in Figure 11-16. Note that it requires reset, clock enable and other clocking resources. This is because the delay taps need selecting within the module.

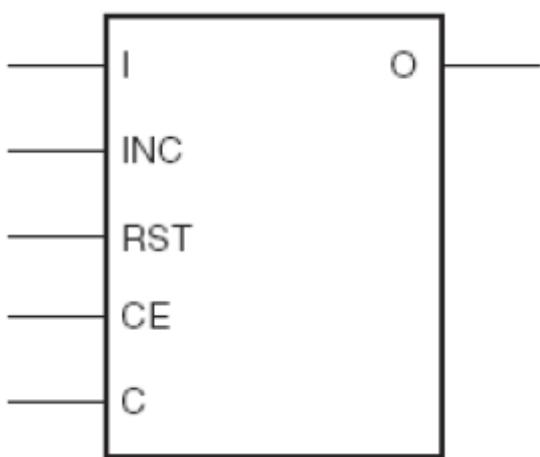


Figure 11-16 Idelay Block Symbol

When CE and INC are high, the internal delay is incremented on each successive clock applied. With CE high and INC low, internal delay is decremented one delay unite. When CE is low, it remains at the current delay selection. There are a total of

64 delay steps inside. The Idelay controller block symbol is shown in Figure 11-17. Note that it requires the attachment of the external Reference clock, to establish the 5 nanosecond period for time measurements (i.e., 200 MHz REFCLK). RST is active after configuration completes. RDY indicates the servo has locked. RDY from IDELAYCTRL can attach to CE on the IDELAY module. REFCLK can be shared across multiple IDELAYCTRL units, and RDY signals from several IDELAYCTRL modules can be combined in the fabric, to synchronize them.

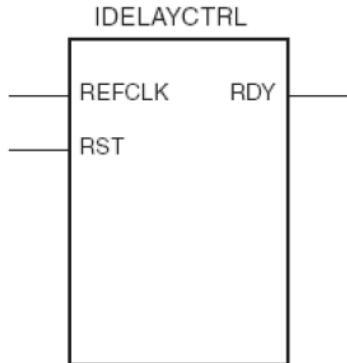


Figure 11-17 Idelay Control Block Symbol

The effect of Idelay on an input signal is shown in Figure 11-18. Note the clock enable and increment signal are applied, resulting in the output being delayed by “one click” of the delay line (~78 picoseconds) within the Idelay module. This delay is shown as dotted lines on the O-Before and O-After signals. After the RST signal is applied, the output resumes tracking the original time delay of the module.

Idelay can be modified manually, using the configuration port, or circuitry can be constructed to automatically insert delay as needed for a design. This will be described with an ISERDES module to perform phase alignment, along the lines of channel bonding with the MGTs, as described in Chapter Ten.

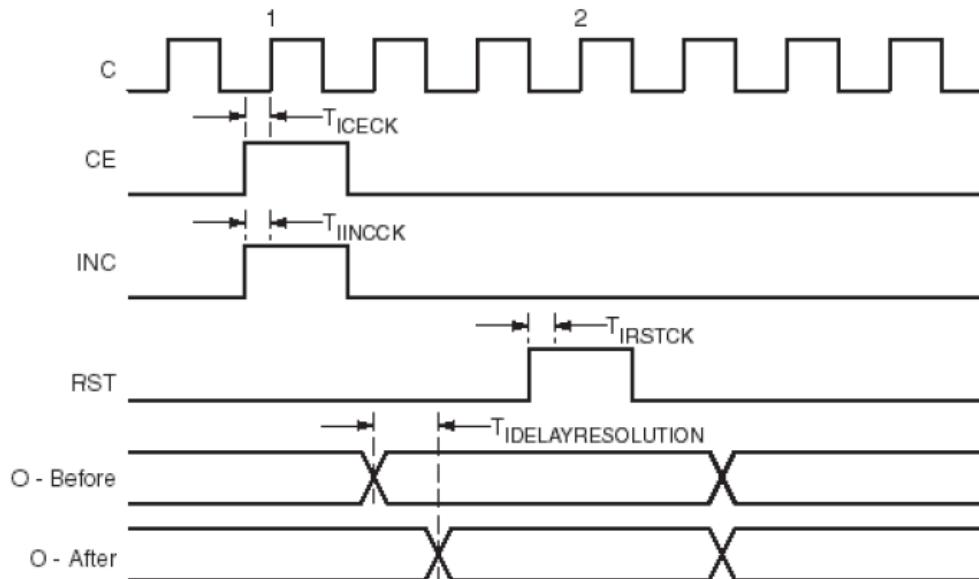


Figure 11-18 Effect of Idelay on Input Signal

## ISERDES

Serial data transmission is so common, that it makes sense to provide high speed serializer/deserializer (SERDES) resources, and starting with Virtex 4, devices deliver just that, at every I/O pin. The functionality is partitioned into two aspects – ISERDES and OSERDES. ISERDES is shown in Figure 11-19. The overall goal is to accept serial input data at the “D” pin, and turn it into 6 bit parallel data bits at the Q1-Q6 pins. Note that IDelay is offered as an input option, to the ISERDES, to potentially delay information with respect to the applied clock. Clearly, a six bit shifter resides within the Serial to Parallel Converter block. Additionally, we see a clock enable module, the BITSLIP module and other assorted control signals managing the ISERDES block. One or two clock enables are available. The SHIFTIN1/2 and SHIFTOUT1/2 are cascade signals used to expand the ISERDES up to 8 or ten bits out.

ISERDES targets two key markets – networking and memory design. Key features exist that are distinct for each market, and are enabled when the appropriate interface type is used in the software. In particular, for memory interfaces, BitSlip is disabled, and OCLK is enabled. With source synchronous memory interfaces, the clock is synchronized with the memory, and DDR applications use both edges to transfer data. With networking, it is common that “channel bonding” is used, whereby multiple serial channels need to be trained to identify where each is defining data word boundaries. For that case, BitSlip is enabled.

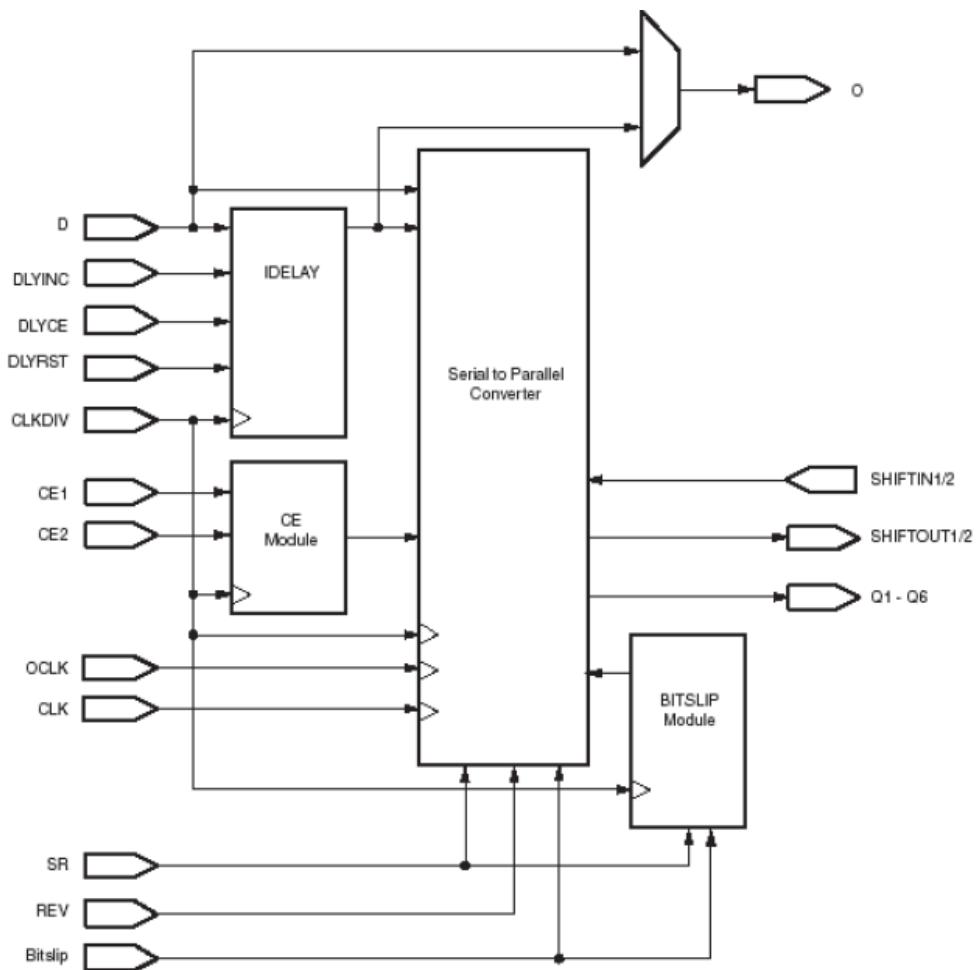


Figure 11-19 ISERDES Block Diagram

It is also possible that greater than six bits are desired for the parallel output of the ISERDES. To accommodate that, there exists two ISERDES modules at each I/O pin – one master and one slave. By cascading the master into the slave, parallel expansion occurs. Figure 11-20 shows ISERDES width expansion.

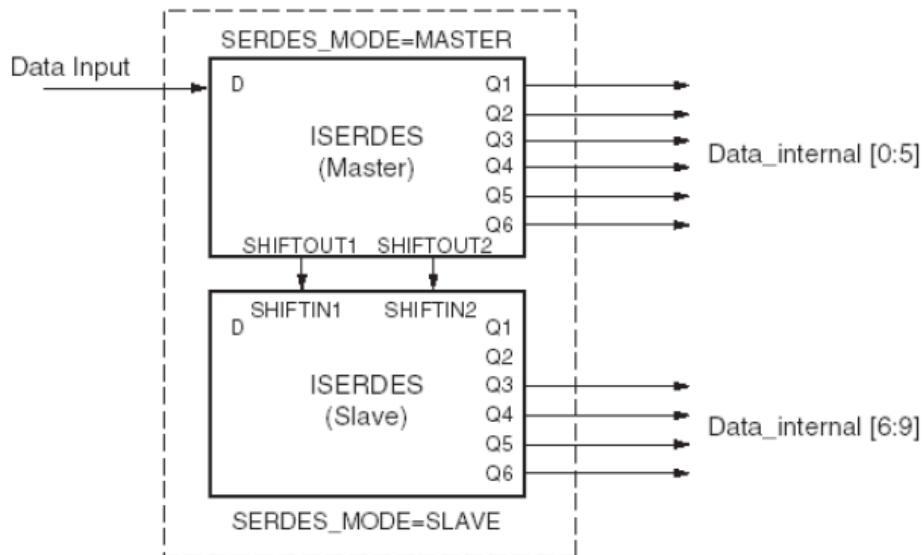


Figure 11-20 ISERDES Width Expansion

For DDR operation, the width expansion can be up to ten bits. For SDR operation, the expansion is up to eight bits. Figure 11-20 shows the ten bit expansion, with Q6 on the Slave ISERDES driving the MSB. One comment on the name: ISERDES would be more properly called IDES as it is an Input DESerializer. Alternately it might have been called SIPO, for Serial In, Parallel Out. Oh well, marketing!

### BitSlip

BitSlip is neat. It permits a serial input channel to shift an incoming pattern, to obtain alignment. Parallel channels can be tuned to eliminate misalignment among them, by carefully “slipping” individual channels until they properly align. Typically, this is done with a training pattern and state machine that dictates the different “slip” values for each separate channel. The tables shown in Figure 11-28 show the basic idea. The left table shows operation with SDR mode and the right table shows it with DDR mode.

Bitslip Operation in SDR Mode		Bitslip Operation in DDR Mode	
Bitslip Operations Executed	Output Pattern (8:1)	Bitslip Operations Executed	Output Pattern (8:1)
Initial	10010011	Initial	00100111
1	00100111	1	10010011
2	01001110	2	10011100
3	10011100	3	01001110
4	00111001	4	01110010
5	01110010	5	00111001
6	11100100	6	11001001
7	11001001	7	11100100

Figure 11-21 BitSlip Examples

To understand the SDR table, think of a left shifting barrel shifter. The initial value of the SDR Mode table has: 10010011 as the starting point. If you left shift that value, end around, it becomes 00100111, which is shown in the second row, where the BitSlip module “slips” by one position. The next row shows the slip by two positions from the initial, and so forth. The DDR mode is more devious. The slip format involves alternating, a barrel shift right, followed by a left barrel shift of three. One right and two left is a net shift of two left, on two successive clock events, but arrived at in a “non-intuitive” fashion. In Figure 11-28, note that the Initial state for SDR is different than for DDR, so right at the beginning, the examples don’t appear to track, as might be expected. The symbolic example below shows what the DDR BitSlip pattern is doing.

X7 X6 X5 X4 X3 X2 X1 X0 original pattern  
 X0 X7 X6 X5 X4 X3 X2 X1 (rotate one right)  
 X5 X4 X3 X2 X1 X0 X7 X6 (rotate three left)  
 X6 X5 X4 X3 X2 X1 X0 X7 (rotate one right)  
 X3 X2 X1 X0 X7 X6 X5 X4 (rotate three left)  
 X4 X3 X2 X1 X0 X7 X6 X5 (rotate one right)  
 X1 X0 X7 X6 X5 X4 X3 X2 (rotate three left)  
 X2 X1 X0 X7 X6 X5 X4 X3 (rotate one right)

---

X7 X6 X5 X4 X3 X2 X1 X0 (rotate three left –original pattern)

The BitSlip logic is a fairly straightforward rotating shift register for SDR. For DDR, it incorporates two shift registers, intertwined with many multiplexers to accomplish the alternating right followed by three left rotates. Keep in mind that the shifting event for DDR alternates between rising and falling edges of the clock, so “bit slipping” needs to respond to both edges, as it proceeds.

### ChipSync Example

Let’s detail the SDR approach for multiple channels, by using the example(s) included in XAPP 700 “Dynamic Phase Alignment for Networking Applications”. Figure 11-22 shows the basic problem of aligning multiple channels. In concept, this is similar to channel bonding, except channel bonding manipulates memory pointers and flags to identify correct data boundaries. BitSlip actually aligns the data.

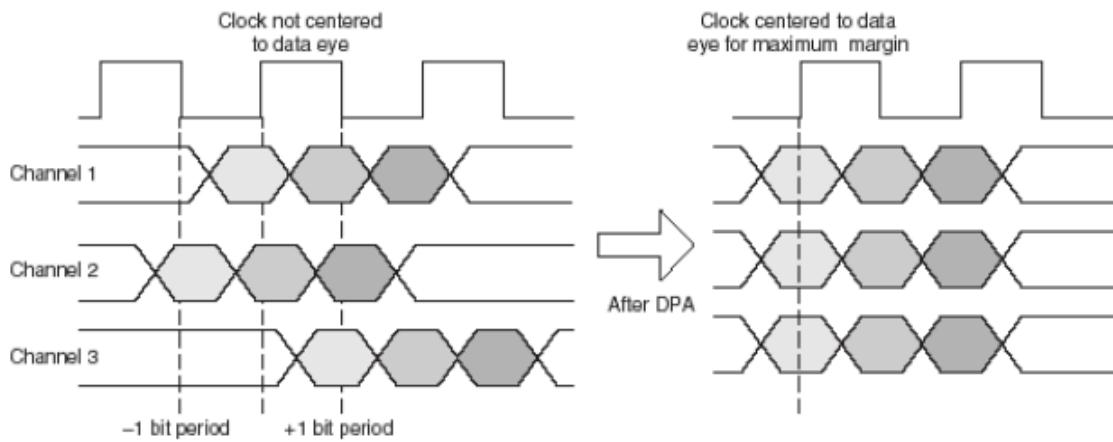


Figure 11-22 Results of Dynamic Phase Alignment

We see three different channels, with successive bits at varying relationships to the incoming clock on the left diagram, and all three channels lock together and synchronize with the leading clock edge on the right hand side.

Figure 11-23 shows the problem being solved – the alignment of sixteen different data channels, with a timeshared set of logic that does the alignment. The alignment occurs with a training pattern applied in source synchronous fashion, which means the pattern has a good mix of data transitions, permitting clock extraction. The basic process can be thought of as two staged – first, bit alignment (centering), then word alignment. Centering is done with Idelay, to place the data appropriately centered with respect to the clock (i.e. “middle of the eye”). Once the data is centered on one channel, it must be centered on the others, taken in succession. Then, after all data are centered, BitSlip is invoked to make sure the “right” bits are aligned within a word. Some of these ideas were described in Chapter Ten.

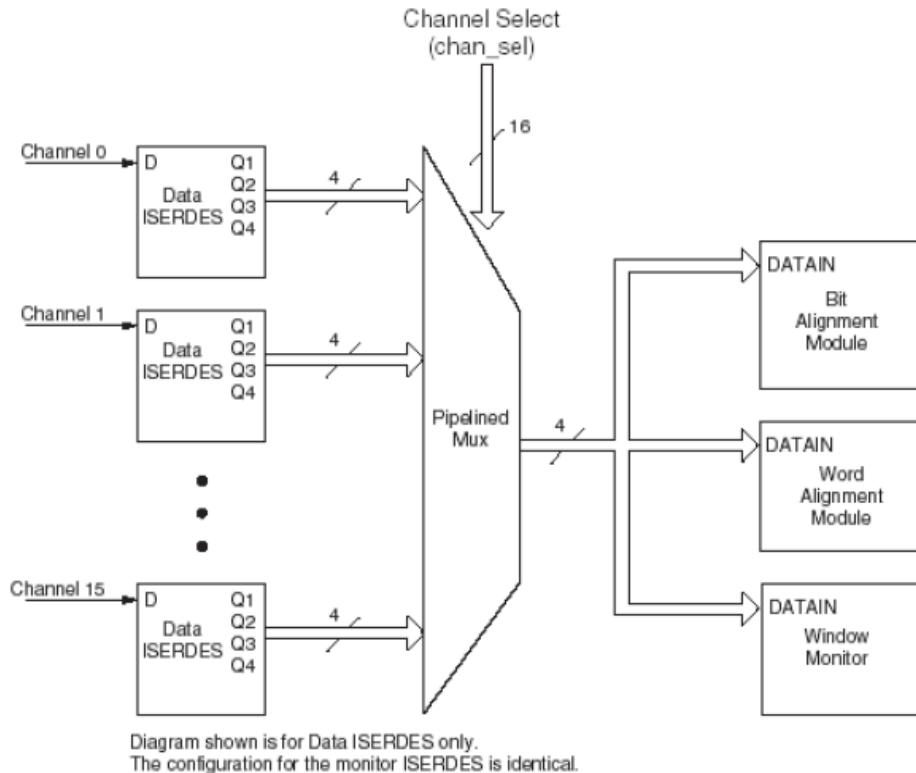


Figure 11-23 Timeshared Training of Control Modules Across 16 Channels

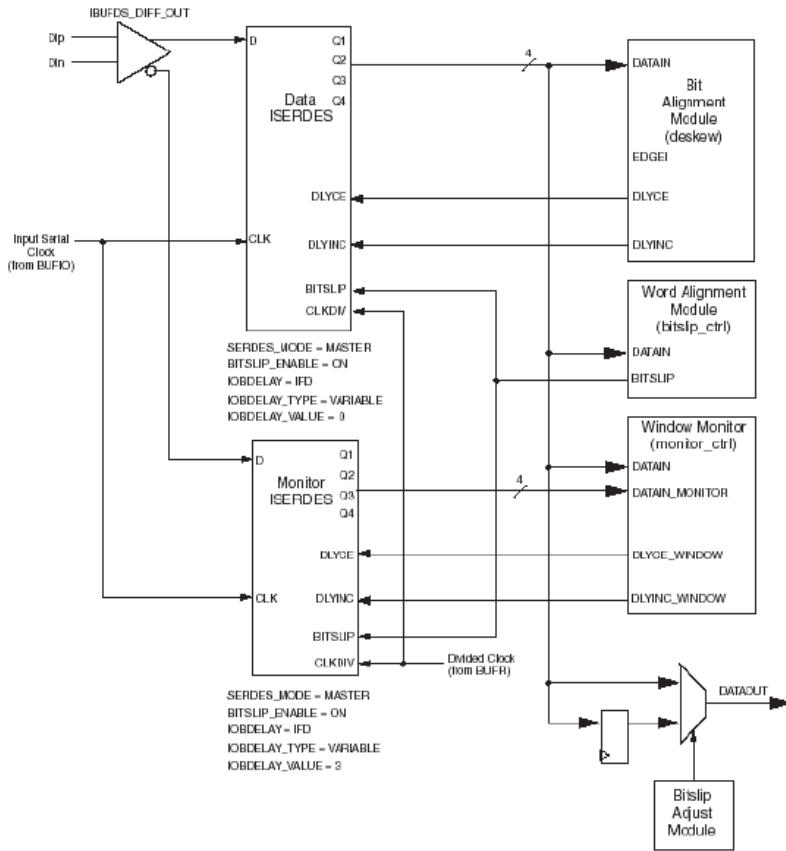


Figure 11-24 Modules of One Dynamic Phase Alignment Channel

Figure 11-24 shows a summary of the actions taken by the Idelay module to identify the leading and falling edges of a bit. After finding the two edges, Idelay has effectively captured a bit period, so the center need only be identified as half the total delay.

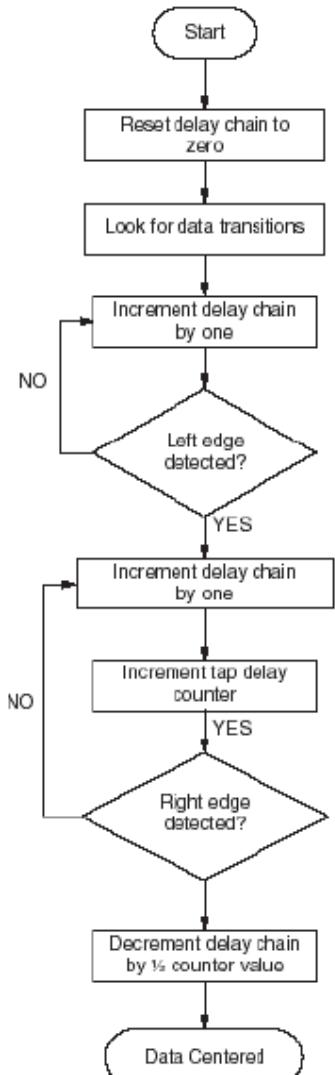


Figure 11-25 Idelay Bit Alignment Strategy

Figure 11-26 shows how the subsequent BitSlip behavior aligns the centered data bits into the appropriate word, for completion of the synchronization process.

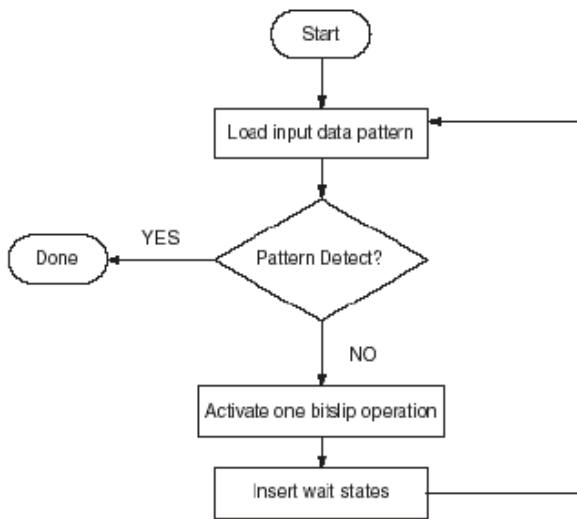


Figure 11-26 Word Alignment via BitSlip Operation

XAPP 700 gives greater detail, as well as the design modules to perform the task. Note that the process would ideally result in a final, synchronized data word, but at high speed in many systems, delay can drift. Hence, the process is “dynamic” and periodically updates the synchronization assuring phase alignment remains locked. This behavior is described by the sections of the design that refer to “Window Monitoring”, which uses ISERDES logic that is “free” due to differential I/O pins not using all their ISERDES blocks.

## OSERDES

Output flexibility is available with the Output Serializer/Deserializer-OSERDES. Actually, it would be more appropriate to call it an Output SERializer (i.e., OSER), as it converts parallel data into serial data. Others might call it a Parallel In, Serial Out, or PISO. Figure 11-27 shows the basic structure.

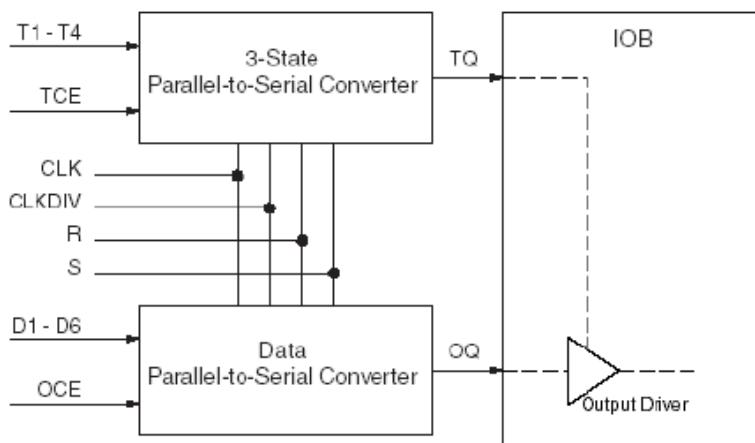


Figure 11-27 OSERDES Block Diagram

Note that the input data being delivered is six bits wide, but it is possible to expand the OSERDES to handle 10 bits of data as in Figure 11-28.

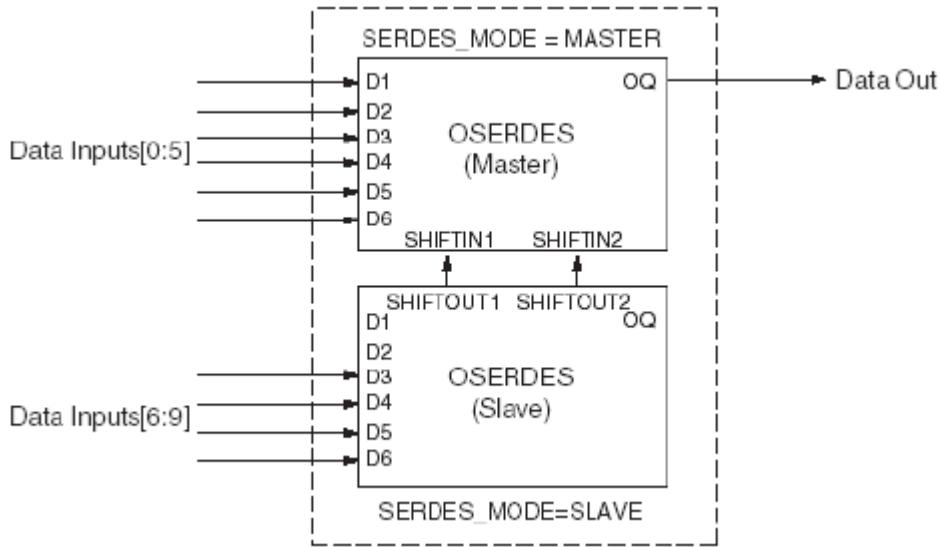


Figure 11-28 OSERDES with Expansion

In Figure 11-27, we see that both CLK and CLKDIV are provided. This makes it possible to do multiple data transactions in DDR format with the OSERDES. Table 11-2 shows the relationship between the CLK, CLKDIV and the various possible SDR and DDR formats. Note that although six input bits are possible, as few as two may be chosen, or as many as ten bits can be handled by cascading a master OSERDES with the available slave OSERDES within the same I/O Tile. Management of the tri-state control of the output pin is handled through the three state parallel to serial converter.

Input Data Width Output in SDR Mode	Input Data Width Output in DDR Mode	CLK	CLKDIV
2	4	2X	X
3	6	3X	X
4	8	4X	X
5	10	5X	X
6	-	6X	X
7	-	7X	X
8	-	8X	X

Table 11-2 CLK and CLKDIV Relationship to Data in OSERDES

Figure 11-29 shows an example of a six bit SDR data transmission. In the diagram, data bits D1A – D6A are presented and successively shifted out on six consecutive clocks. This is followed by D1B-D6B on the next six clocks, and so forth.

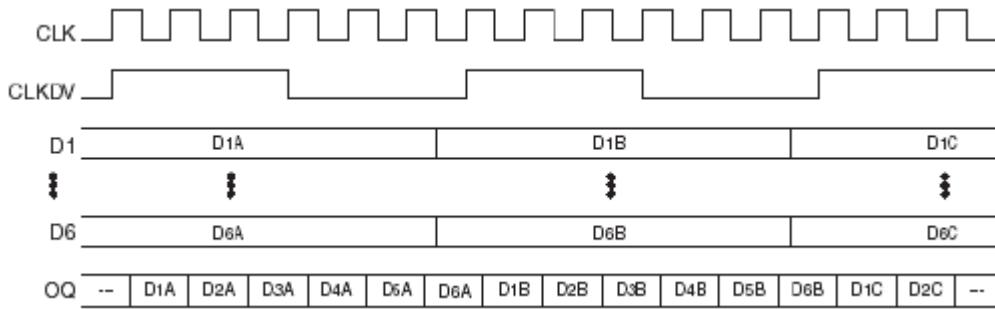


Figure 11-29 Six Bit SDR Data Transmission

Figure 11-30 shows a similar set of data transfers in DDR mode.

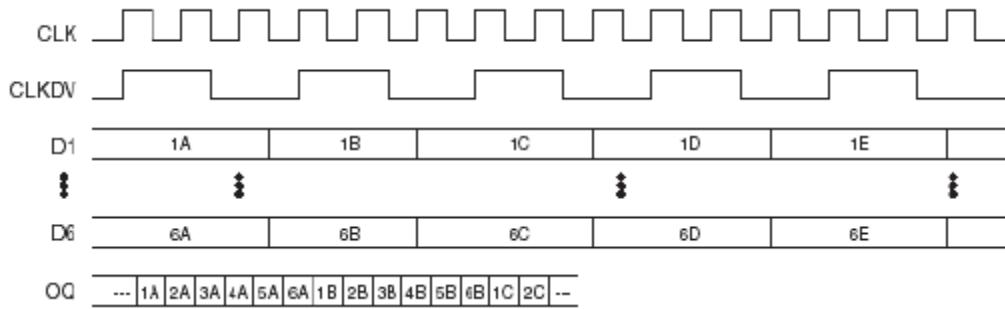


Figure 11-30 Six Bit DDR Data Transmission

In Figure 11-30, data bits D1A-D6A transfer alternating on rising and falling clock edges. This is followed by data bits D1B-D6B, and so forth. ISERDES and OSERDES are designed to work together for a complete bidirectional, self-aligning data stream resource, or be used independently on separate channels, as needed.

## Conclusion

Switching speed continues to increase. Virtex Family SelectIO supports that in a big way. As discussed, designers need to address signal integrity with careful layout, termination and signal management techniques. It is no wonder that the data communication world has focused their attention on fast serial links – if there is a problem, they know where it will be. But, problems go beyond just signaling and terminating. Early planning for device pin-outs has become critical. Partitioning designs so that the right logic accesses the correct ports, and gets to/from the correct pin sets – in advance - is a problem that didn't exist in earlier times. To address those needs, Xilinx has engaged with many third party tool developers that help arrive at a “best pin-out” before the design is placed and routed. Board level simulations, IBIS and Spice modeling are all part of the complete picture.

## Additional References

1. U.S. Patent #6,218,858, Programmable Input/output Circuit for FPGA use in TTL, GTL, GTIP, LVPECL and LVDS Circuits, by Suresh Menon, Yogendra Bobra, Atul Ghia and Arch Zaliznyak

2. U.S. Patent #6,489,837 B2, Digitally Controlled Impedance for I/O of an Integrated Circuit Device, by David Schultz, Suresh Menon, Eunice Hao, Jason Bergendahl and Jian Tan
3. U.S. Patent #6,445,245 B1, Digitally Controlled Impedance for I/O of an Integrated Circuit Device, by David Schultz, Suresh Menon, Eunice Hao, Jason Bergendahl and Jian Tan.
4. XAPP230, The LVDS IO Standard
5. XAPP231, Multi-Drop LVDS with Virtex-E FPGAs
6. XAPP233, Multi-Channel 622 Mb/s LVDS Data Transfer for Virtex-E Devices
7. XAPP243
8. XAPP704, Virtex-4 High-Speed Single Data Rate LVDS Transceiver
9. XAPP705, Virtex-4 High-Speed Dual Rate LVDS Transceiver
10. U.S. Patent#6,525,565 (B2)
11. U.S. Patent#6,777,980 (B1)
12. 7 Series FPGAs SelectIO Resources User Guide, UG471 (v1.6) September 18, 2015

## Chapter 12 Embedded Microprocessors

### Introduction

Since the LCA 3000, designers have been creating soft microprocessors to embed into FPGA devices. They were cute. They worked, but many would be hard pressed to argue that it was more cost effective to build the processor in an FPGA, versus the cheaper, CMOS single chip 8 bit processors that were available in volume. Yet, users did it anyway. Processors like the 6502, the 8051 and the Z80 found their way into FPGA devices, and a cottage industry grew up supplying cross assemblers and debug utilities for these designs.

As time progressed, Xilinx began offering microprocessor (uP) designs for users that wanted them. The first was actually an application note / reference design called the "KCPSM" named after its designer, the Ken Chapman Programmable State Machine. Ken based this design on one a college professor developed, but it was simple, effective and efficient, particularly when adapted to the Virtex architecture. KCPSM took advantage of using the LUTs as register arrays. Ken went on to create an assembler, and investigated many tradeoffs of using a processor approach to designs, versus direct hardware solution (ie, Verilog or VHDL). KCPSM has since evolved into what is called PicoBlaze, and will be detailed shortly.

Seeking a more complete, 32 bit solution, Goran Bilski developed the original MicroBlaze soft processor. This was the first soft processor to be offered commercially at Xilinx, and the company chose to develop a complete support package for it, with assembler, C compilers, debuggers, full documentation and trained company support. It continues to evolve and was recently the topic of a Microprocessor Report issue, which noted that it now offered floating point multiply. In a sense, the name MicroBlaze begat the name PicoBlaze, the smaller predecessor, although they are in fact unrelated.

The PowerPC architecture was developed back in the 1990's as a collaboration between Apple, IBM and Motorola. Both IBM and Motorola developed their particular spins on this RISC architecture, with Apple probably being the largest customer for it, until recent announcements otherwise. Nonetheless, over recent years, the PowerPC 405 model has been the processor of choice primarily in the data communication field. Given Xilinx market direction at the time of this decision, as well as the partnership arrangement during the development of the Virtex-II Pro family, with IBM, it became a natural choice. We will take a look at this solution also, along with supporting hardware and software infrastructure.

Before moving into the architectures, it should be pointed out that the original motivation of using processors inside of FPGA devices involved at least two prime factors – integration and solution tradeoff. Integration is obvious, minimize the total number of chips, reduce board area, reduce power, increase reliability, and so forth. Solution tradeoff was hinted at earlier. Some tasks require the raw speed of brute force hardware designed for a special purpose. Other tasks are not hardware intensive, and may be best served by sharing a processor operating slower, but more area efficiently than the fabric might perform. Being able to assign slower tasks to the more efficient processor and faster tasks to the high-speed fabric, or built in function blocks makes sense. This is also at the heart of including a processor choice. Being able to select among processor options is another critical factor.

Finally, with the move to the ARM processor ecosystem (arguably the most popular processors in the world today as they are in every smart phone, the 7 series Xilinx Zynq device finally got the FPGA+uP equation correct. What we did not do previously was to recognize that to cater to the uP market, we must look EXACTLY like a uP!

## PicoBlaze

Figure 12-1 shows the PicoBlaze soft processor module combined with an on chip BRAM. Several different embodiments of PicoBlaze exist, with varying instruction sets and instruction bit widths. Earlier models incorporated 16 bit instruction paths, but the Figure 12-1 model uses 18 bits, which fits nicely with Virtex BRAMs in the Virtex II, Virtex 4 and Spartan 3 families. We will discuss instructions, shortly.

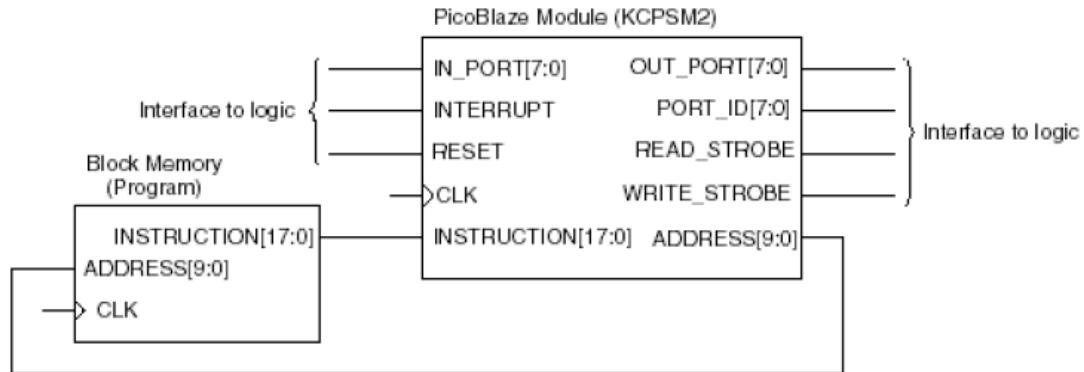


Figure 12-1 PicoBlaze Module and Block Memory

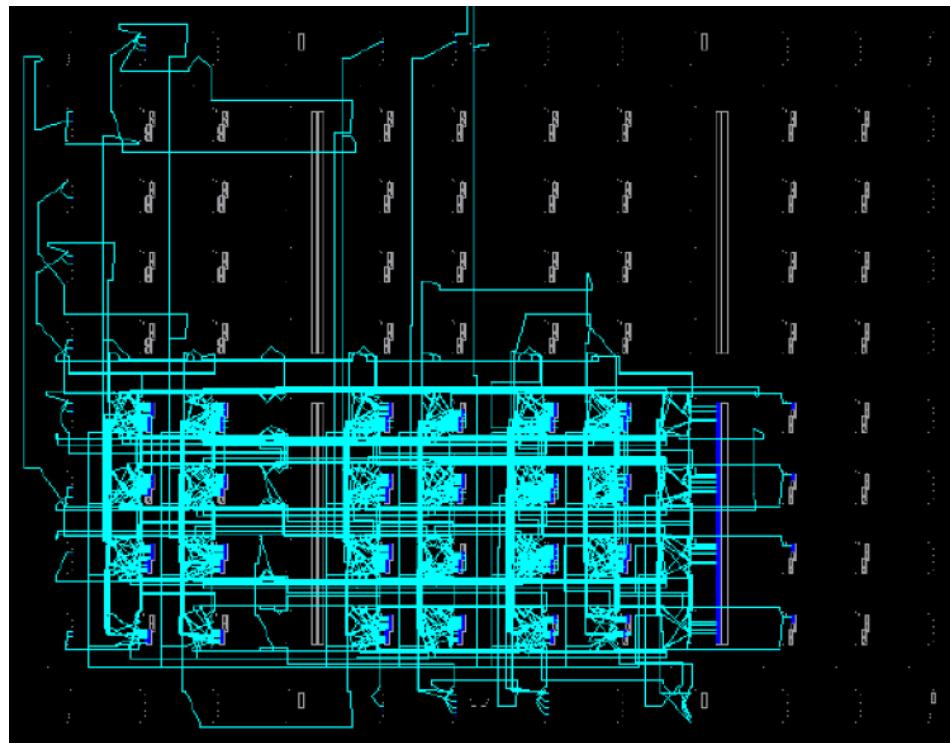


Figure 12-2 PicoBlaze in an XC2V40 Virtex II

Figure 12-2 shows a view of PicoBlaze compiled onto a small Virtex II FPGA, where it clearly occupies less than half the chip. Figure 12-3 shows the same design compiled onto an XC2V6000, where PicoBlaze could easily fit in between cracks left behind by bigger designs. “There’s always room for PicoBlaze”, might be a convenient slogan.

PicoBlaze is the first processor entry, where designers might ask the question: Could I better use my FPGA area budget creating solutions from brute force hardware, or might a better solution partition some functions into a soft processor, operating slower, but handling many such slow moving tasks in an efficient way? This partitioning process is definitely up to the designer, but one of the aims of this chapter is to show ways to take advantage of processors combined within the fabric – either hard modules or soft ones.

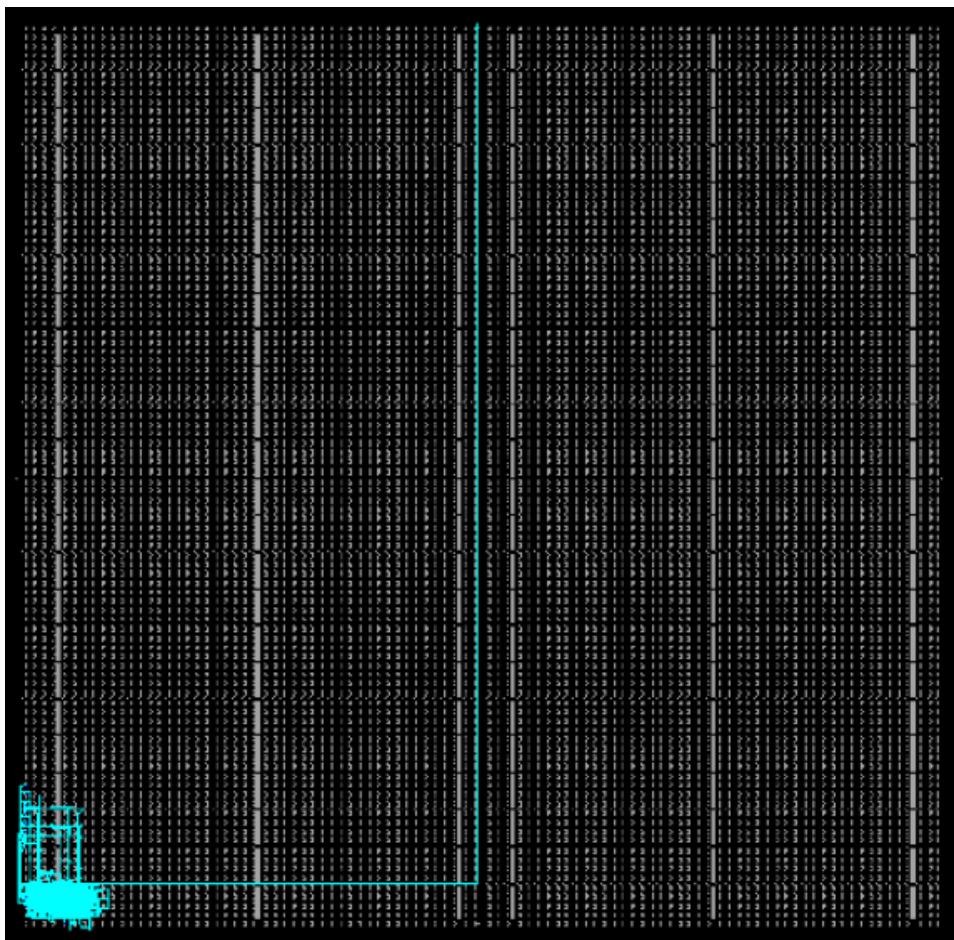


Figure 12-3 PicoBlaze in XC2V6000 Virtex II

Figure 12-4 shows a more detailed diagram of the PicoBlaze architecture, identifying the various register files, ALU, program counter, address stack, I/O ports and control signals. PicoBlaze is constructed to take advantage of using LUT RAM for register files and using the fast adder chains contained in the CLBs. Functionality is targeted to make best use of available resources contained in the fabric. However, users are free to modify the design as required. There is no path to write into the program memory, so this constitutes a basic Harvard architecture. Data is typically stored in external BRAM, with instructions coming from the module labeled ROM/RAM,

meaning a BRAM is dealt with as a read only facility. Operands are eight bits, so the ALU, and all data paths track that width. There are 256 output addresses and 256 input addresses. Cycle times vary with the compilation and part speed, but speeds in excess of 150 MIPS are common.

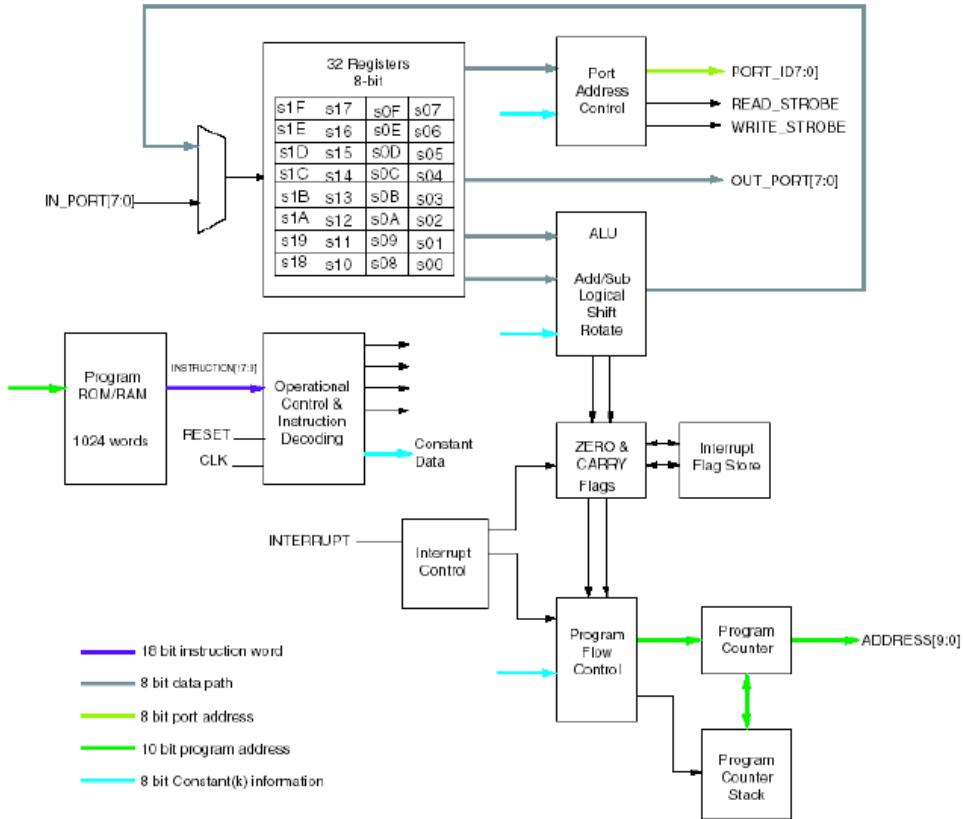


Figure 12-4 PicoBlaze Architecture

One of the key attributes of this architecture, is its use of eight bit constants or “immediate operands” embedded into the instruction word, to take quick action when operating on other data (masks, etc.). Figure 12-5 shows the entire design can fit into as few as 77 slices. PicoBlaze requires two clock cycles per instruction. PicoBlaze provides an interrupt that is handled with a PC stack, and appropriate enable/disable instructions. External logic can expand management of the interrupt input. The stack is fifteen entries deep, so interrupt nesting is possible, to that extent. XAPP 213 on PicoBlaze includes details on writing interrupt service routines to support this.

PicoBlaze supports 49 instructions, in the usual assortment of Add, Subtract, AND, OR, EX-OR, Jump, Jump to Subroutine, etc. Most offer a version with an immediate constant, to reduce cycles and speed things up. Code space is 1024, 18-bit instructions which fit in a single 18K bit BRAM.

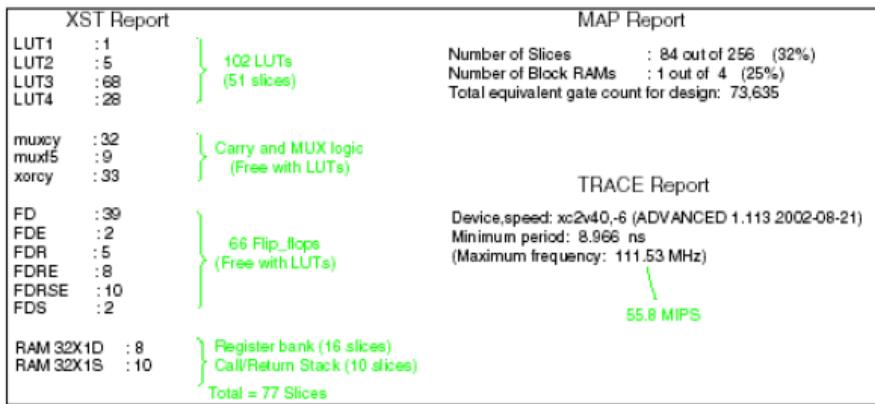


Figure 12-5 Performance and Resource Utilization of PicoBlaze in XC2V40

Because it is a “soft” processor, PicoBlaze may be altered to suit a designer’s needs. Indeed, the CPLD version of PicoBlaze has shown that when the design compiles (VHDL or Verilog) along with the users code, the design automatically collapses (in synthesis) to supporting just the instructions being used, unless otherwise constrained. The other direction is also possible, whereby the design is augmented with additional capability, folded into the fabric with the rest of the design. Other mixes are also possible, as the design is basically elastic at the architect’s will. Key to supporting this flexibility is at least, minimal support tools that permit modifying the assembler along with the architecture. At least two assemblers are offered for free. Standard simulation using third party simulators like ModelSim, Synopsys, Cadence, Synplify are all available and work well.

Although Xilinx has a great response to PicoBlaze, its primary support is reserved for 32 bit users of MicroBlaze and the newer Zynq ARM processors to which we now direct our attention.

## MicroBlaze

MicroBlaze evolved out of the academic efforts of engineer, Goran Bilski. Initially developed as a good, basic 32 bit soft RISC type architecture, it has gained in stature through the addition of an organized external busing structure, and a sophisticated tool suite for developing serious applications. All registers, instructions, data paths and address paths are thirty two bits wide. Figure 12-6 shows the MicroBlaze Core Block diagram. Note the available instruction cache (I-Cache) and data cache (D-Cache) permitting rapid fetch/store for both commands and data. Clearly, on board BRAM is a resource to be exploited. At the time of this writing, MicroBlaze has progressed to include the following:

- 3 or 5 stage pipeline support
- Native AXI-4 support
- AXI Coherency Extension (ACE) Support
- Cache line word length: 4, 8 or 16
- Area or Speed Optimized configuration option
- Support for Memory Management Unit
- Low latency Interrupt mode support
- Fault Tolerance, including Error Correction Codes (ECC) and Lockstep support
- MPU mode for region protection for secure RTOS applications

- Instruction and Data Caches
- Cache size configurable: 2kB - 64kB (Block RAM based)
- Local Memory Bus (LMB) Instruction and Data side interface
- Hardware Barrel Shifter
- Hardware Multiplier and Divider
- Up to 16 AXI Stream interfaces
- Floating Point Unit (Single Precision, IEEE 754 compatible)
- Processor Version Register
- Re-locatable Base Vectors
- Support for Sleep mode and Sleep instruction
- Extended Debug Support: Performance Monitoring, Performance Trace, Non-Intrusive profiling

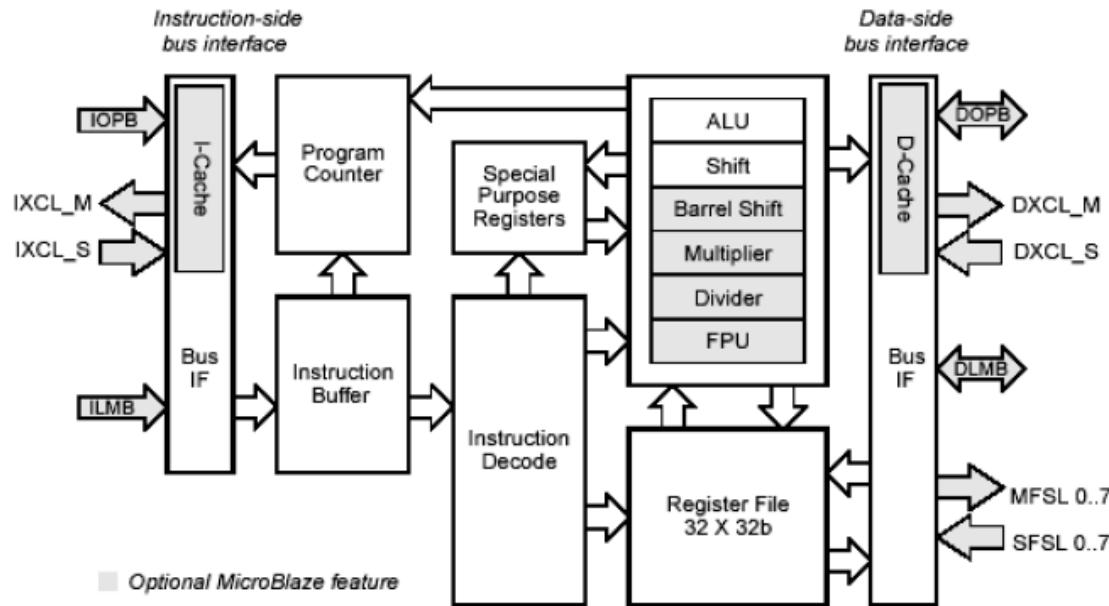


Figure 12-6 MicroBlaze Core Block Diagram

Being a “soft” processor core again means the design can be altered to suit the user’s needs. Gray areas in Figure 12-6 represent aspects that are optional. We can’t describe all the functions in great detail but the Additional References point to that literature. Let’s simply identify the key functions and summarize their capabilities.

### Register File

The Register File is comprised of 32 addressable, 32 bit registers. Some have additional alternate functions beyond general purpose. For instance, Register 0 is a read only all zero constant. Register 14 holds interrupt return addresses. Register 16 holds break return addresses while Register 17 holds hardware exception return addresses. In addition, Register 17 can be general purpose, if not using hardware exceptions. The other registers within the file are simply 32 bit general purpose storage.

## Arithmetic and Logic Unit

The ALU is a 32 bit arithmetic and logic unit supporting the instruction set of MicroBlaze. It provides a full complement of 32 bit integer add, subtract, shift, Boolean operations, as well as multiply and divide. Recently added are optional capabilities for handling floating point add, subtract, multiply and divide, all with the IEEE 754 single precision standard. The standard for rounding is “round to the nearest”.

## Special Purpose Registers

There are five special purpose registers. The Program Counter is one. The Machine Status Register (MSR) has eleven specified bits and 21 reserved bits. The specified bits describe the condition of the machine, like “carry”, “exception in progress”, “break in progress”, “division by zero” and so forth. The Exception Address Register (EAR) holds the address of the instruction which resulted in the exception. Typically, an exception is a misaligned memory load or store, but data alignment must be corrected during operation, and this register helps isolate the offending operation. The Exception Status Register (ESS) holds specific bit fields that can help recover from an exception. Several of the bits indicate “illegal address”, “illegal op-code”, “floating point exception”, etc. The Floating Point Status Register (FSR) has five specific bits that indicate “invalid operation”, “divide by zero”, “overflow”, “underflow” and “de-normalized operand error”.

## Instruction Buffer

MicroBlaze implements a classic, three cycle (fetch, decode, execute) pipeline. Although most instructions execute uniformly, branch delay slots are incorporated to assure proper execution of previous instructions. A pre-fetch buffer is included to correct for wait state effects when dealing with slower memory.

## Caches

MicroBlaze supports both data and instruction caches. Both are direct mapped. Both can be user configured, as well as the Tag memory. Note that if the main memory is BRAM, adding BRAM caches will not speed things up. BRAM cache should be used with slower external memories, and also with the memory management unit which allows MicroBlaze to run and support Linux.

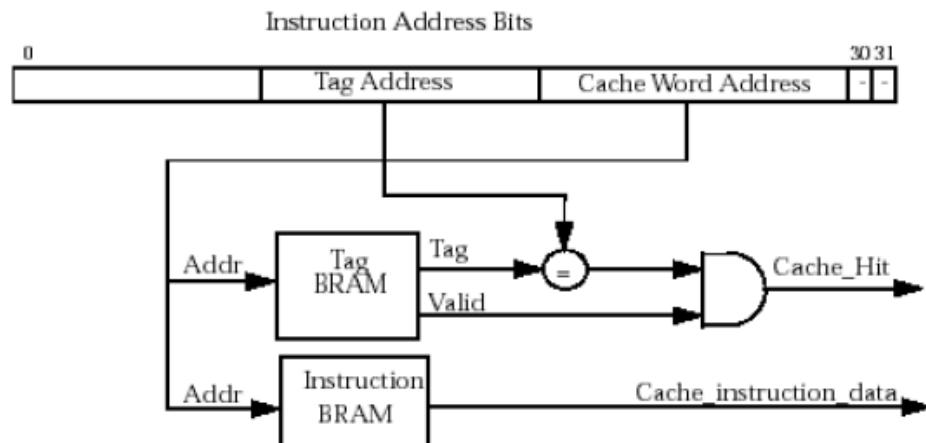


Figure 12-7 OPB Instruction Cache Organization

Figure 12-7 shows how an Instruction Address coming off the Instruction On-Chip Peripheral Bus (IOPB) maps to the appropriate instruction BRAM results. Cache software maintains the Tag BRAM and the Instruction BRAM, as a rule. MicroBlaze supports special cache command instructions for cache maintenance. To facilitate easy interface with all hard and soft IP, the AXI Bus is used by Xilinx throughout, so MicroBlaze now fully supports AXI interfaces, masters and slaves, to external memory, peripherals, and custom blocks (IP cores).

Designers are encouraged to make their solutions “blaze” (i.e., fast). To do that, they will need to identify sections of their design that can bottleneck, versus sections that operate at reasonable speeds. Performance monitoring tools help identify bottlenecks, but if you assume that such blocks have been identified, the next step is to create special purpose hardware engines – accelerators – to replace the slow code. Once that is done, you need only to marry them to the processor.

### **MicroBlaze Support**

MicroBlaze and the ARM processors are supported by the same basic software tool, which will be discussed in a later section.

### **PowerPC405 (PPC405)**

#### Architectural Philosophy

As suggested earlier, the PowerPC was developed by an industry consortium seeking to create a serious RISC processor capable of handling a broad spectrum of different commercial applications. Anticipating such needs as a complete personal computer processor, the portable laptop and embedded applications, different Architecture Levels were envisioned. In terms of levels, there is for starters, the User Instruction Set Architecture (UISA), which includes the problem state, or user environment. This includes the set of instructions and registers that are available to be manipulated by the user’s programs, as well as data types, floating point memory conventions and in general, the architecture viewed by the user. The UISA is common to all levels, to assure that user programs remain transportable across architectures.

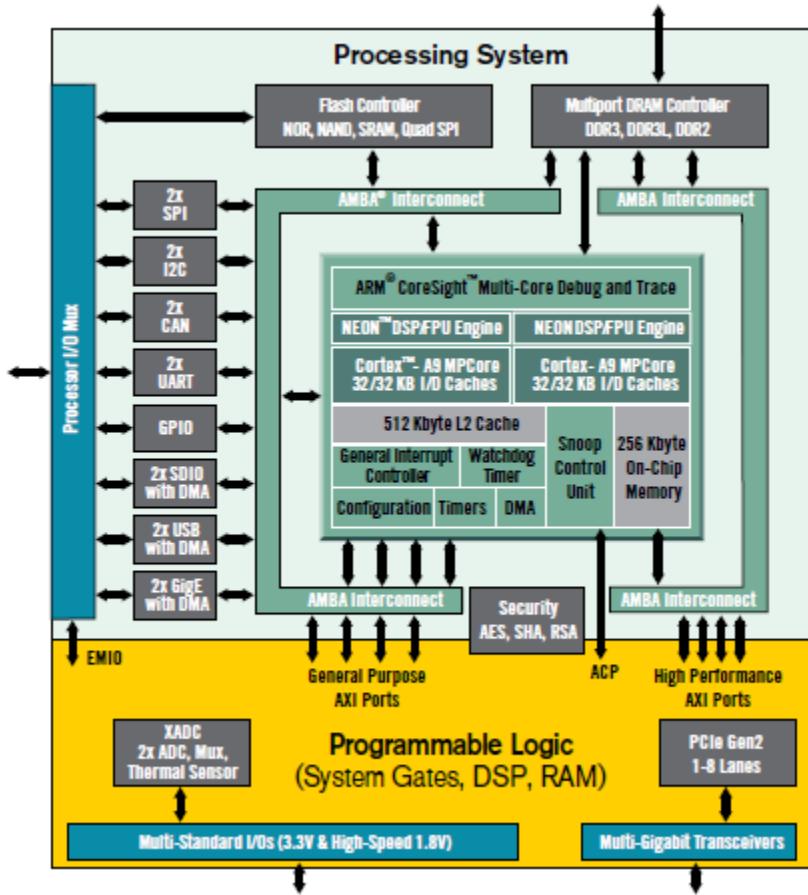
The Virtual Environment Architecture (VEA) defines additional user level functionality outside typical user level requirements. The VEA includes defining time-base resources from a user perspective, cache models and cache control instructions, and a memory model for multiple devices to access memory. The VEA conforms to the UISA. Finally, the Operating Environment Architecture (OEA) adds in supervisor level resources needed by an operating system. The OEA includes synchronization requirements, exception handling, memory management and defines time base resources from a supervisor level. The OEA conforms to the VEA, thereby also conforming to the UISA.

There is latitude within the PPC Architecture Levels, to accommodate variation. For instance, some registers, register bit fields and instructions are optional. It is also possible to create additional registers and supervisory commands. Hardware resources can be implemented that perform the same task as instructions, but must deliver identical results. There are areas outside the scope of the PPC architecture that may be defined for different architectural variations. For instance, the System Bus Interface is not specifically defined signals and protocols for bus transactions. It is also possible to implement bus specific exceptions that trigger into the OEA. Cache design

(size, structure, replacement policy) are not defined. Having and Instruction-Cache and a Data-Cache is not required. Details of pipelining and hardware structure are not defined. There is plenty of room for modification and improvement, while maintaining full compatibility over the required UIISA. Xilinx chose to implement the Embedded PowerPC model as the largest Xilinx user base best fit that model at the time. Since then, the ARM Ecosystem became the dominant embedded system microprocessor choice, so no more will be said about the PPC.

## Zync AP SoC

At the 28 nanometer node, the Virtex FPGA device fabric was combined with the hardened ARM Cortex A7 32 bit microprocessor platform and peripherals to provide a complete “System on a Chip” (SoC). In Figure 12-8 we see the two sides of the new device, the processor system (PS) and the programmable logic (PL).



12-8 Zync All Programmable SoC

First, the PS has a complete set of IO pins dedicated and available without using any FPGA fabric resources. In fact, its power is separate so that there is no single point of failures. Next, the PS side has a rich set of commonly used peripherals hardened so that most systems need only the special peripherals or accelerators the specific application requires. Finally, the PS to PL interface uses the AXI bus to seamlessly allow wide bandwidth interfacing between the two.

## PL Side

The PL consists of:

1. Dual-Core ARM Cortex-A9 Based Application Processor Unit (APU)
2. CPU frequency: Up to 667 MHz
3. Coherent multiprocessor support
4. ARMv7-A architecture
5. TrustZone security
6. Thumb-2 instruction set
7. Jazelle RCT execution Environment Architecture
8. NEON media-processing engine
9. Single and double precision Vector Floating Point Unit (VFPU)
10. CoreSight technology and Program Trace Macrocell (PTM)
11. Timer and Interrupts
12. Three watchdog timers
13. One global timer
14. Two triple-timer counters
15. 32 KB Level 1 4-way set-associative instruction and data caches (independent for each CPU)
16. 512 KB 8-way set-associative Level 2 cache (shared between the CPUs)
17. Byte-parity support On-Chip Memory
18. On-chip boot ROM
19. 256 KB on-chip RAM (OCM)
20. Byte-parity support
21. External Memory Interfaces
22. Multiprotocol dynamic memory controller
23. 16-bit or 32-bit interfaces to DDR3L, DDR3,
24. DDR2, or LPDDR2 memories
25. ECC support in 16-bit mode
26. 1 GB of address space using single rank of 8-, 16-, or 32-bit-wide memories
27. Static memory interfaces
28. 8-bit SRAM data bus with up to 64 MB support
29. Parallel NOR flash support
30. ONFI1.0 NAND flash support (1-bit ECC)
31. 1-bit SPI, 2-bit SPI, 4-bit SPI (quad-SPI), or
32. two quad-SPI (8-bit) serial NOR flash
33. 8-Channel DMA Controller
34. Memory-to-memory, memory-to-peripheral, peripheral-to-memory, and scatter-gather transaction support
35. Two 10/100/1000 tri-speed Ethernet MAC peripherals with IEEE Std 802.3 and IEEE Standard 1588 revision 2.0 support
36. Scatter-gather DMA capability
37. Recognition of 1588 rev. 2 PTP frames
38. GMII and RGMII interfaces
39. Two USB 2.0 OTG peripherals, each supporting up to 12 Endpoints
40. USB 2.0 compliant device IP core
41. On-the-go, high-speed, full-speed, and low-speed modes support
42. Intel EHCI compliant USB host
43. 8-bit ULPI external PHY interface
44. Two full CAN 2.0B compliant CAN bus
45. CAN 2.0-A and CAN 2.0-B and ISO 118981-1 standard compliant
46. External PHY interface
47. Two SD/SDIO 2.0/MMC3.31 compliant controllers

- 48. Two full-duplex SPI ports with three peripheral chip selects
- 49. Two high-speed UARTs (up to 1 Mb/s)
- 50. Two master and slave I2C interfaces
- 51. GPIO with four 32-bit banks, of which up to 54 bits can be used with the PS I/O (one bank of 32b and one bank of 22b) and up to 64 bits (up to two banks of 32b) connected to the programmable logic (PL)
- 52. Up to 54 flexible multiplexed I/O (MIO) for peripheral pin assignments
- 53. High-bandwidth interconnect connectivity within PS and between PS and PL
- 54. ARM AMBA AXI based
- 55. QoS support on critical masters for latency and bandwidth control

Perhaps this list is excessive, but not when compared to the requirements of systems today. What is not in this list may be implemented in the PL side.

### PL Side

In keeping with its Virtex roots, the Zynq device has:

- 1. Configurable Logic Blocks (CLB)
  - a. Look-up tables (LUT)
  - b. Flip-flops
  - c. Cascadeable adders
- 2. 36 Kb Block RAM
  - a. True Dual-Port
  - b. Up to 72 bits wide
  - c. Configurable as dual 18Kb
- 3. DSP Blocks
  - a. 18 x 25 signed multiply
  - b. 48-bit adder/accumulator
  - c. 25-bit pre-adder
- 4. Programmable I/O Blocks
  - a. Supports LVCMOS, LVDS, and SSTL
  - b. 1.2V to 3.3V I/O
- 5. Programmable I/O delay and SerDes
- 6. JTAG Boundary-Scan Standard 1149.1 Compatible Test Interface
- 7. Two 12-Bit Analog-to-Digital Converters
  - a. On-chip voltage and temperature sensing
  - b. Up to 17 external differential input channels
- 8. One million samples per second maximum conversion rate
- 9. Automotive Temperature Range
- 10. Serial Transceivers
  - a. Up to 4 receivers and transmitters
  - b. Supports up to 6.6 Gb/s data rates
- 11. PCI Express® Block Root Complex and Endpoint configurations
  - a. Supports up to Gen2 speeds
  - b. Supports up to 4 lanes
- 12. XADC or Sysmon 10 bit Analog to Digital Converter

Combine the PL with the PS, and quite simply you have the best of both worlds. Please refer to the excellent Zynq Book (see references) for a complete discussion of how to use this device.

## XADC or Sysmon

As early as Virtex 5, the system monitor or *Sysmon* block was introduced. The XADC is the basic building block that enables analog mixed signal (AMS) functionality which is new to 7 series FPGAs. By combining high quality analog blocks with the flexibility of programmable logic, it is possible to craft customized analog interfaces for a wide range of applications. See the references or [www.xilinx.com/ams](http://www.xilinx.com/ams) for more information.

### XADC Overview

The XADC includes a dual 12-bit, 1 Mega sample per second (MSPS) ADC and on-chip sensors. The ADCs and sensors are fully tested and specified (see the respective 7 series FPGAs data sheet). The ADCs provide a general-purpose, high-precision analog interface for a range of applications. Figure 12-9 shows a block diagram of the XADC. The dual ADCs support a range of operating modes, for example, externally triggered and simultaneous sampling on both ADCs and various analog input signal types, for example, unipolar and differential. The ADCs can access up to 17 external analog input channels.

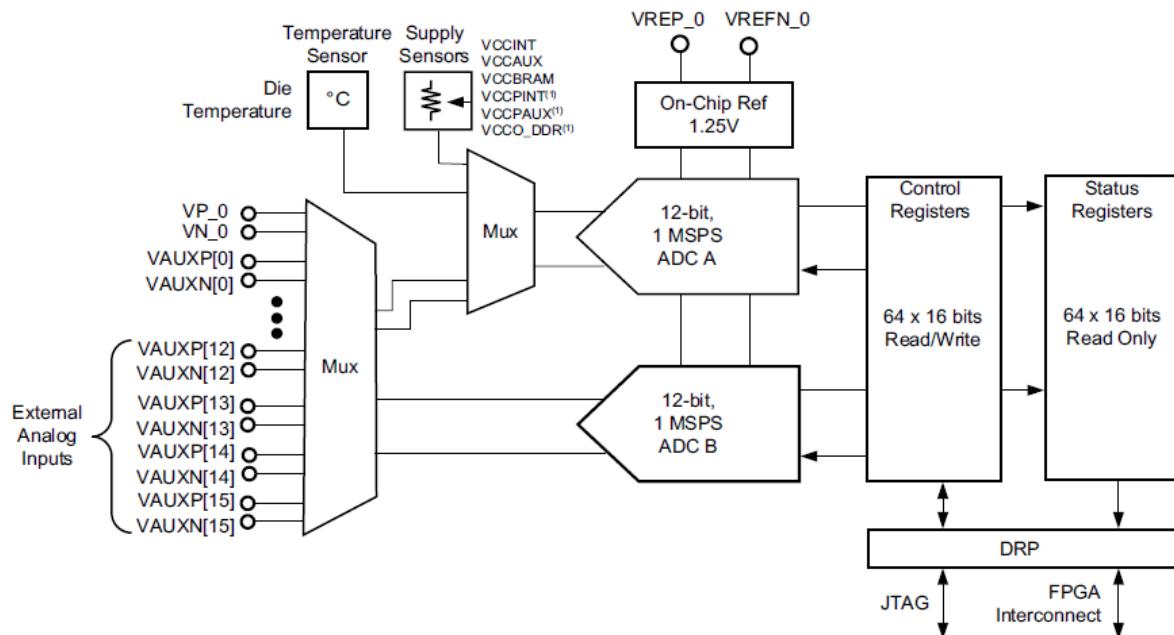


Figure 12-9 XADC Block Diagram

The XADC may monitor internal power supply nodes, and die temperature. The minimum and maximum sample registers keep these values, along with the most recent reading, and an averaged value (if programmed). The sample and hold bandwidth is ~ 1 GHz, so transients of a nanosecond in duration may be ‘caught’ in the long term by sampling techniques. In this way, the integrity of the power supply system may be verified (no voltages out of the recommended for the operation).

## Conclusions and Directions

The Illiac IV was moved from Purdue University to Moffett Field in Santa Clara, to reside in a dirigible hangar. In case you are new to computing, the Illiac IV was a multiple computer array that sliced problems up and divided the workload across an array of identical processors, to achieve extremely high computation rates for solving problems like weather prediction, or modeling weapon behavior. One Zynq device on a desktop could easily do the same thing, today. Problems that took an act of the U.S. Congress to fund, can actually be built by an individual, as a hobby. Indeed, a PicoBlaze based array processor can be built for around \$5.00. Array processing is just one application embedded processor FPGA devices make possible, and practical. As discussed in the chapter on configuration, fault tolerance for FPGA fabric is accomplished with triple modular redundancy and configuration scrubbing. Commercial fault tolerant computing first happened when Tandem Computers invented lockstep processing. That made very robust computer structures capable of exhibiting “nonstop” computing, by marrying together two processors that auto checked results as they proceeded. This was already done in Virtex II processors.

Lock-stepping two MicroBlaze is easily done. Dataflow processing has generally been studied in academic labs. Embedded computing makes custom dataflow processors a reality. Likewise, for systolic processors. Even mythical processors like the DLX RISC used as an example in the Hennessy and Patterson computer architecture textbook are possible to build up and use in universities. Commercially, all of these ideas are now possible, and make a lot of sense, particularly with regard to high speed multimedia processing done in conjunction with the Internet. But, the ability to give deep understanding to students and users, by making intimate access points within the architecture available for inspection and measurement is also a great value. At the time of this edition, 82 car models are designing advanced driver assistance systems (ADAS) using Xilinx Zync SoC devices.

References (available on the Xilinx website):

1. XAPP213 PicoBlaze 8 Bit Microcontroller
2. XAPP627 PicoBlaze 8-Bit Microcontroller for Virtex-II Devices
3. MicroBlaze Processor Reference Guide (UG984)
4. Xilinx Software Development Kit (XSDK) (UG782)
5. The Zynq Book, [www.zynqbook.com](http://www.zynqbook.com)
6. 7 Series FPGAs and Zynq-7000 All Programmable SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide UG480 (v1.8) May 24, 2016

# **Chapter 13 Virtex Family FPGA Devices in Data Communications**

## **Introduction and Overview**

Xilinx design engineers were at the right place and the right time, with the data communications market. As high density XC4000 family parts were impacting the market, several very influential data communication companies were seriously embracing FPGA devices, and were more than willing to discuss specific needs for creating compelling solutions to urgent problems. This included influencing the selection of BRAM sizes and configurations, selection of I/O voltage standards, encoding formats for high speed transceivers, and the selection of the ARM processors as Xilinx' choice for fixed embedded processors. Additional guidance has been gained from various DSP gurus and that impact is also evident in the Virtex families of FPGA devices.

In data communications, it was crystal clear that programmable fabric and choice IP blocks would permit early adoption of data standards that were not yet official, and grant leeway to system designers to modify choices, if the standards changed before adoption. Adaptability to the standards in real time, as they evolved gave FPGA devices an incredible edge to become indispensable in this market.

Because data communications is so broad of a term, this chapter will present a few high level principles and definitions, then summarize the current bounty of Xilinx' available data communications solutions. Think of this chapter as an encapsulation of key Xilinx data communications literature – some of which is listed at the chapter end, for further study. An important consideration, however, is that it will represent a snapshot in time of what has been released and available on the Xilinx website. The list may be updated in the future, but readers can find additional material by searching [www.xilinx.com](http://www.xilinx.com).

## **Basic Data Communication Ideas**

The High Speed Transceiver chapter discussed the basics of a data communication channel. This included various signaling methods, encoding data, error correction, data pre-emphasis and a number of system issues that need to be pre-agreed on by both the "sender" and the "receiver". We will not revisit those ideas here, but rather, build upon them. A very large number of data communication standards are concerned with high level protocols and packet formatting. One of the most common ways of describing data communication activities is with the familiar OSI protocol diagram, or "stack". See Figure 13-1.

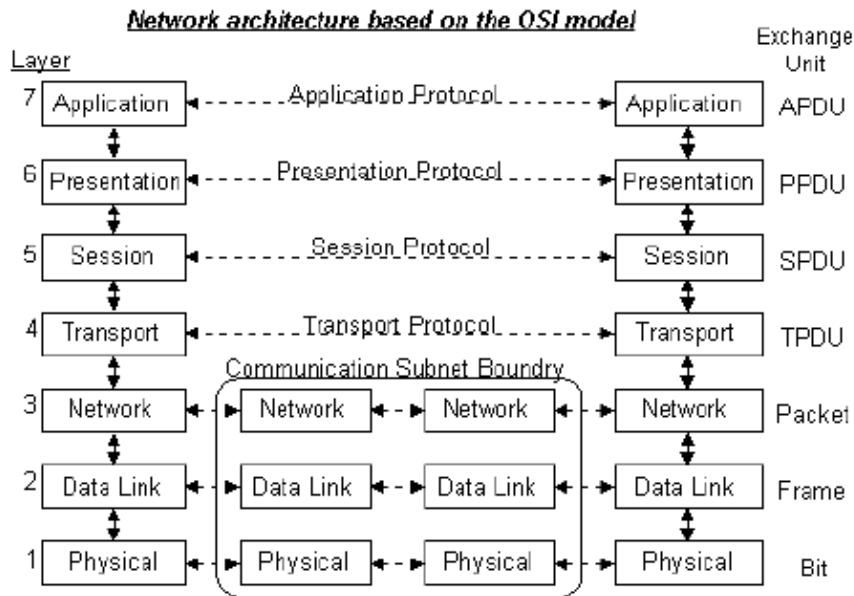


Figure 13-1 OSI Model of Data Communications

Very generally, the lowest OSI level is concerned with details of the physical transmission of data – voltage levels, connector pins, cabling quality, termination, and so forth. The very highest level is concerned with using the data that is exchanged. Between those levels are varying degrees of data formatting, packet creation, error encoding, assembly, disassembly, file creation and overall formatting and unformatting. Of particular interest is the simple fact that Virtex family FPGA devices can participate at every OSI level on the chart, from the bottom physical level to the highest abstract level on the chart. We will see examples illustrating that in this chapter.

With all of the jargon injected into standard OSI discussions, it is sometimes forgotten that the primary mission is to send and receive a file of data called “the payload”. When the payload is fragmented into smaller portions and wrapped with header information having destination addresses, error checking fields and packet ordering information, it assumes a less personal nature. The file is handled by the various protocol engines encountered, according to predefined rules, which are successful in disassembling the payload, transporting the pieces, then reassembling the original file – intact!

It’s a true miracle of modern technology. And the Internet is a testimony to the success of all this digital “massaging”. We will look at several categories of data communications solutions in this chapter. For starters, we consider some key building blocks that exploit the advantages of combining fast, specific function hard blocks within the FPGA, under the title of Data Communication Building Blocks. Then, we will see how small subsystems can be created to perform standard, medium sized functions. Finally, we will take a look at how Virtex parts – particularly Virtex 7can create complete data communication systems.

## Data Communication Building Blocks

Let's first look at some Virtex capabilities that work well with a large number of data communication applications. This list is tiny compared to Virtex capabilities, but let's overview basic content addressable memories (CAMs), clock/data alignment and serializer/deserializers. These are typical "day to day" functions found in a wide variety of data communication systems.

### *Building CAMs*

XAPP 201 overviews the basics of building a content addressable memory, and summarizes other capabilities detailed in XAPP202, 203 and 204. Figure 13-2 focuses on comparing a simple RAM with a CAM. As shown, the RAM has a 10 bit address path, and reads out an eight bit data path. Omitted is the write mode for the RAM, as well as any control signals. The idea here is that a RAM delivers the data contents of a selected address. A CAM, operates a bit differently. In this case, eight bits are delivered into the CAM in the form of a match query. The data bits are compared within the CAM, and if they match, the CAM manifests a "match" signal, indicating the presence of the data, and an "address" value that is associated with the data value. Hence, the data input value corresponds to the "content" being sought, and the match/address results are the corresponding response. In data communications, packets frequently arrive into the data communication subsystem, where a data field located before the beginning of the payload is isolated, and dropped into a CAM to produce a forwarding address for that particular packet. This type of action can happen in a single clock cycle, which dramatically accelerates this type of "address forwarding". Another CAM type of mapping could occur with a disk controller, whereby the data bus contains a filename, and the "address" value might be the disk track and sector, where the file is stored. Clearly, the appropriate data items need to be pre-loaded into the CAM prior to query, and the CAM needs to be initially flushed so there will be no false matches, right at power on time. It's true that delivering the address out is its primary mission, but identifying whether there is a valid "match" at query time, also must be fast. If there is no match, the system needs to identify that the transaction failed, so a processor can load the correct address into the CAM, for future queries, quickly.

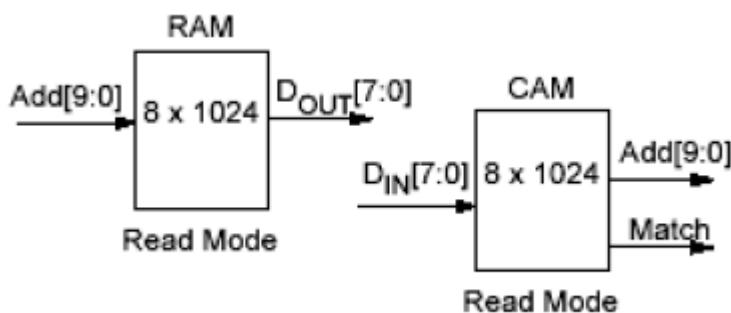


Figure 13-2 Comparing RAM and CAM

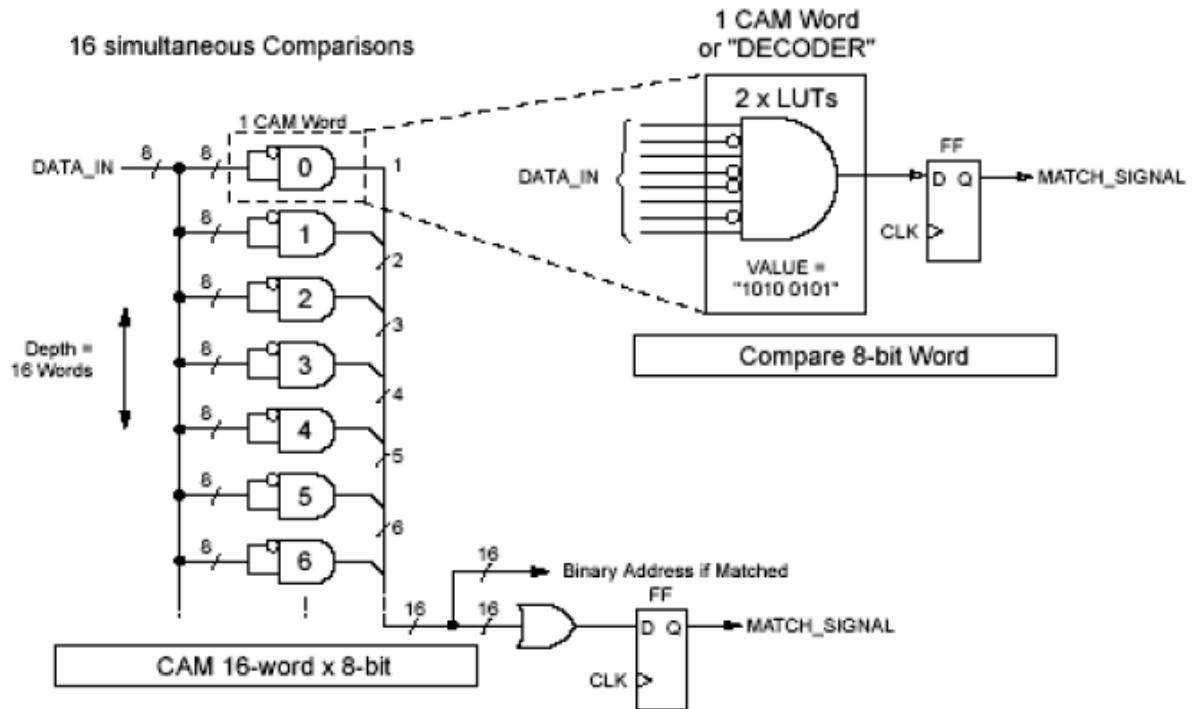


Figure 13-3 Eight bit comparator

Figure 13-3 shows how a LUT can be preloaded so that a specific applied address will supply the match condition. This is performing sixteen simultaneous compares, and only one should match. The operation is thought of as a logical AND of incoming bits. Figure 13-4 better shows how an eight bit input data value is split into two four bit fields into two LUTs, and the results combined with the fast carry multiplexers. In this situation, the top LUT contains 15 zeroes, and a single one loaded into the LUT at the address equal to the applied binary value on the address lines ( $A[3:0]$ ). The same, or a different binary nibble would be applied to the second LUT address lines, and again, 15 zeroes are loaded, with a single one at the address that we wish to compare. When both four bit fields select their respective address holding a one, they will produce ones that will drive the fast carry multiplexers, to deliver a “match” to the nearby flip flop. Figure 13-5 expands this detail, showing multiple such words being simultaneously queried.

The beauty of this approach becomes apparent, when it is realized that CAMs can be constructed in this manner, to adapt to the desired number and width of data items needed. It is possible to expand in both width and depth, as desired, and those are detailed in the Additional Reference application notes.

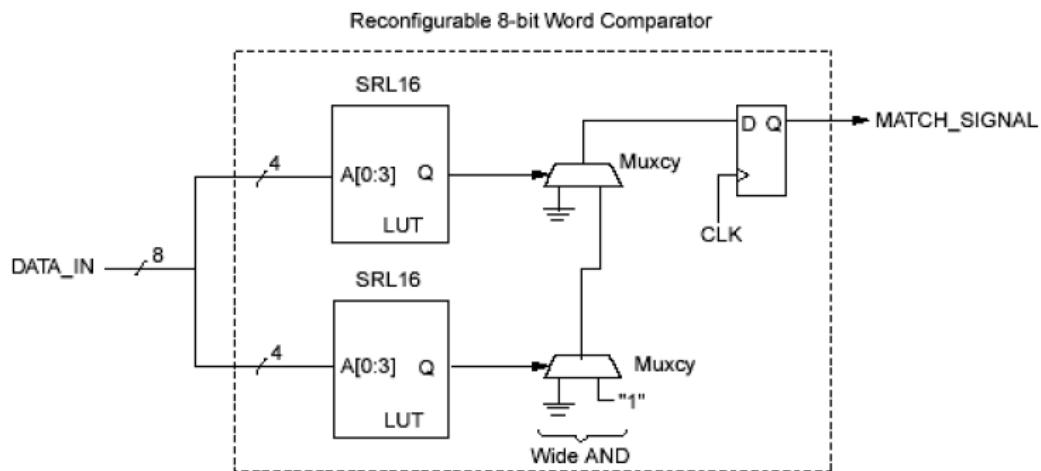


Figure 13- 4 Two LUT Eight bit word Comparator

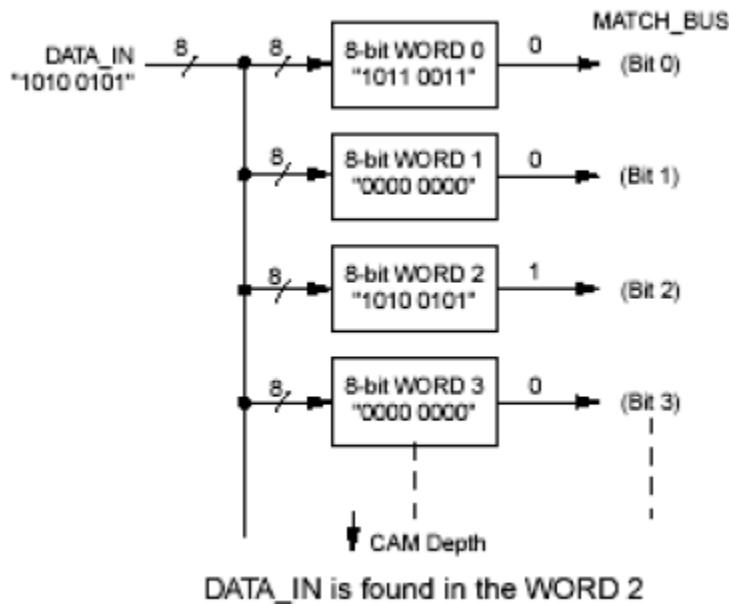


Figure 13-5 Querying Multiple Eight Bit Words Simultaneously

Table 13-1 describes some of the configurations that can be created, and how many resources are needed to accomplish the task. Note that LUT RAM and BRAM can be used to build CAM structures, and can be combined, where typically the LUT RAM does the "match" and the BRAM supplies the contents sought.

Reference Design	Depth (number of words)	Width (number of bits)	CAM Block (bits)	Read		Write	number of SRL16E or RAM16x1	number of BlockRAM	number of TBUF
				Match	Encode				
XAPP 204	32	8	256	4.5 ns	11.5 ns	2 x 11.5 ns	16	2	
XAPP 204	128	8	1K	5.5 ns	15 ns	2 x 15 ns	64	8	32
XAPP 204	256	8	2K	8.5 ns	19 ns	2 x 19 ns	128	16	64
XAPP 203	16	16	256	7.5 ns	7.5 ns	16 x 8.5 ns	64		
XAPP 203	32	16	512	8 ns	8 ns	16 x 10 ns	128		
XAPP 203	128	40	5K	12 ns	12 ns	16 x 14 ns	1280		32
XAPP 203	256	24	6K	12.5 ns	12.8 ns	16 x 15 ns	1536		64
XAPP 202	256	16	4K	16 x 12 ns	12 ns	12 ns	256		
XAPP 202	4096	16	64K	16 x 20 ns	20 ns	20 ns	4096		

Table 13-1 Various CAM Solutions Found in Xilinx Application Notes

### Clock to Data Phase Alignment

ChipSync, which is found in Virtex 4 and later FPGA devices focuses on aligning input clocks and data, but a similar capability can be obtained using CMT's with phase shifter circuits and fabric based alignment. Figure 13-6 shows the basic idea, where different data arrives from different external devices, on unsynchronized input data clocks.

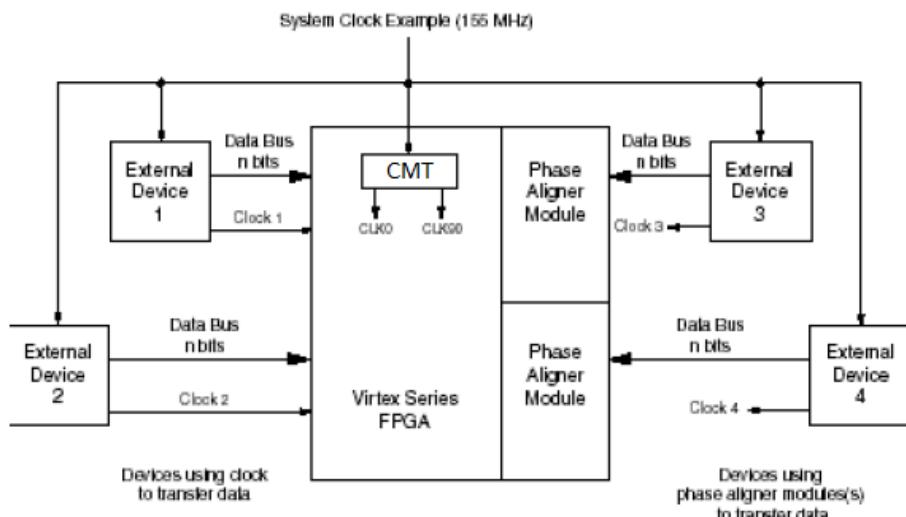


Figure 13-6 Typical Data to Clock Phase Alignment Application

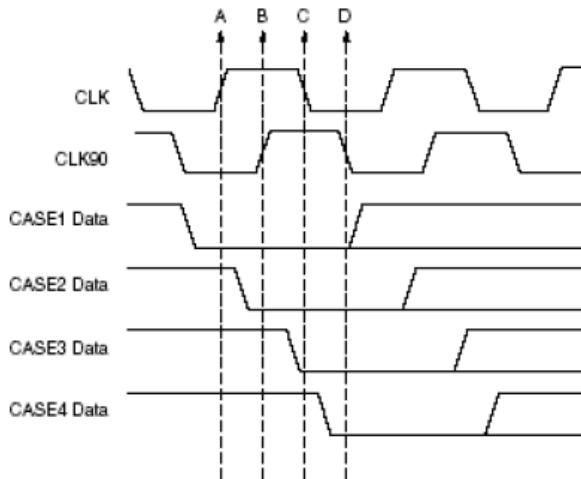


Figure 13-7 Multiphase Data Clocking

Figure 13-7 shows how a clock and its ninety degree phase shifted version creates four unique clock edges – A, B, C and D.

Assigning input data to a corresponding input clock, to best meet the setup time for that data needs to be done. Carefully observe the assigned clocks. The middle and the right hand flip flop columns all attach to the input CLK signal, except on the bottom row. Clocks down the left hand column attach to either CLK or CLK90 (A and B in Figure 13-7), for the first Figure 13-8 Input Stage two flip flops. The bottom two flip flops are also connected to CLK and CLK90, but with inversion at the flip flop, to accomplish the 180 and 270 degree phase shifts (C and D in Figure 13-8). The middle column of flip flops and the right hand column synchronize the results to the CLK signal, with the two flip flop shift structure in each row eliminating possible metastability that might occur if setup time is missed by the left hand flip flop in that row. After the data lands in the various flip flops, it is possible to construct a set of eight comparators that identify where the best data will be, and select among them with a multiplexer that presents the selection to the rest of the data communication circuit. Those details are covered further in XAPP 225 and are appropriate for all Virtex parts and Spartan II and later families.

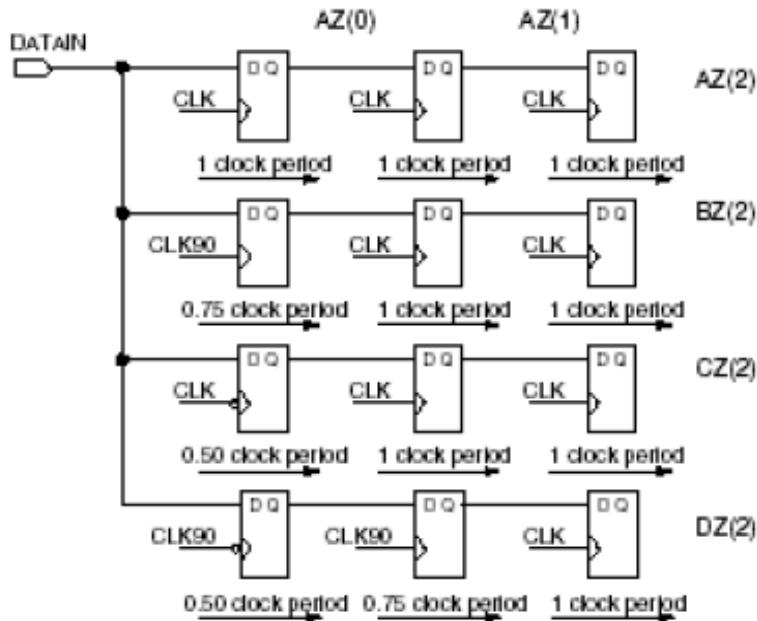


Figure 13-8 Data Alignment

Virtex ChipSync capability doing dynamic phase alignment is detailed in XAPP 707. In this situation, multiple channels are aligned using the ISERDES and Bitslip feature, to permit multiple words and channels to properly track a clock and maintain word integrity across multiple channels.

#### Using Block SelectRAM Memories as Serializer/Deserializers

Block SelectRAM can be effectively used as both a SERDES structure as well as a RAM buffer, as described in XAPP 690. Let's first revisit the basics. Figure 13-9 shows a generic “deserializer”. Basically, it's a shift register, with some support logic. The support logic is a counter that identifies when data frames are complete, and a comparator that examines the shifted contents to alert when specific bits are present in the register. Frame detection from the counter will be combined with bit matching in the comparator, to flag special frames, characters or bits that occur at key times in a protocol.

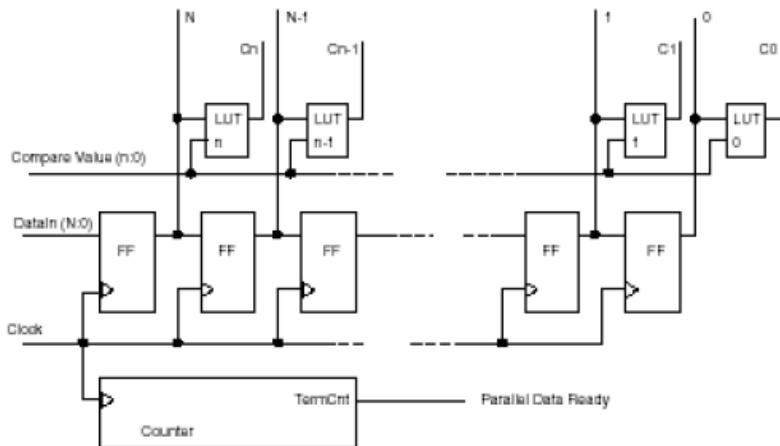


Figure 13-9 Generic Deserializer

Figure 13-10 shows how the shifter might be constructed with a BRAM. Assume the BRAM address is controlled by a counter pointing at address zero. Assign input data on a single input bit, say the least significant of eight. Then, feed back the output data, shifted over one bit onto the next higher order input bit. On successive write clocks, the first data bit enters the BRAM, is latched, then feedback to the second input data position. The next clock loads the second bit into the first bit's position, and reloads the first bit into the second bit position, within the word. The third data bit arrives and is clocked into the low order position, at the same time, the second bit enters the second position, and the first bit assumes the third bit position. This proceeds until a word is complete – say eight bits. Then, the address counter advances to the next word, and the process proceeds. Due to the large number of configurations possible with the BRAM, it is possible to load different word lengths, as needed. Figure 13-10, suggests such an “n bit” word.

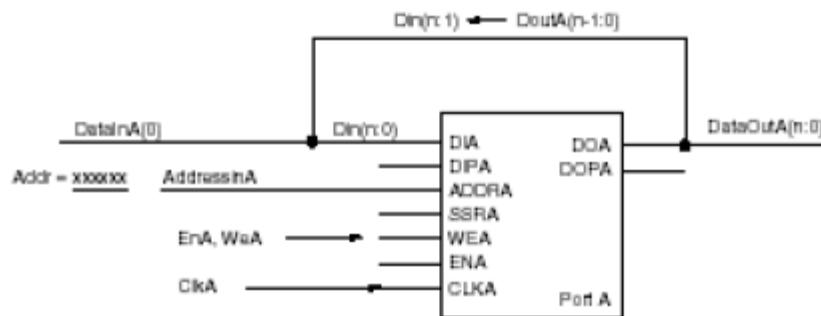


Figure 13-10 Single Port Deserializer

Combining the ideas of Figure 13-9 and Figure 13-10, we obtain Figure 13-11. The Address counter is on the top left side of the diagram, feeding the BRAM A port address input. The output of the BRAM feeds a pattern matcher (similar to the CAM circuit, earlier built from LUTs). The bottom portion of Figure 13-11 includes a five bit counter, as the data frame in this example is a 32 bit word.

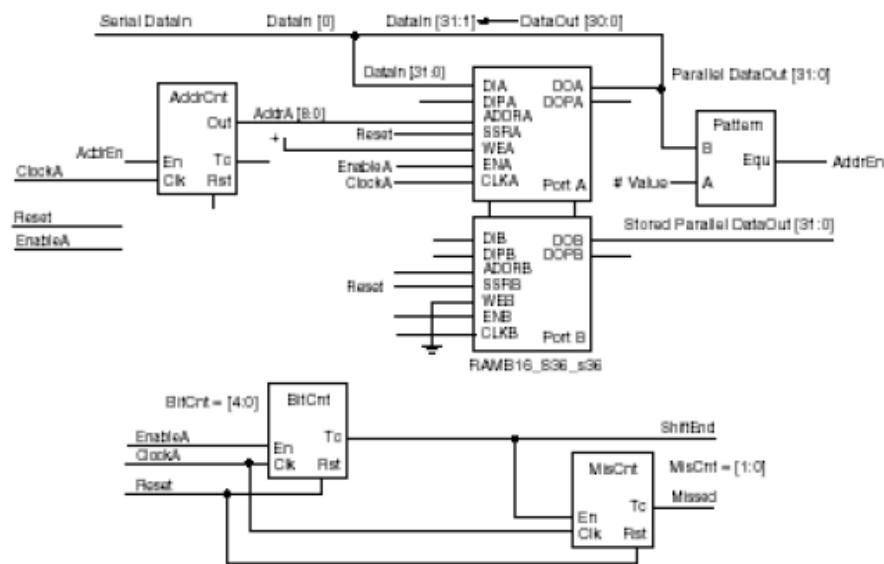


Figure 13-11 Single Port Deserializer with Second Port Readout and Pattern Matching

Figure 13-12 shows how the shifter core in Figure 13-11 can be modified to capture double data rate values into two BRAM ports, and the data interleaved to build up the words, by combining their outputs.

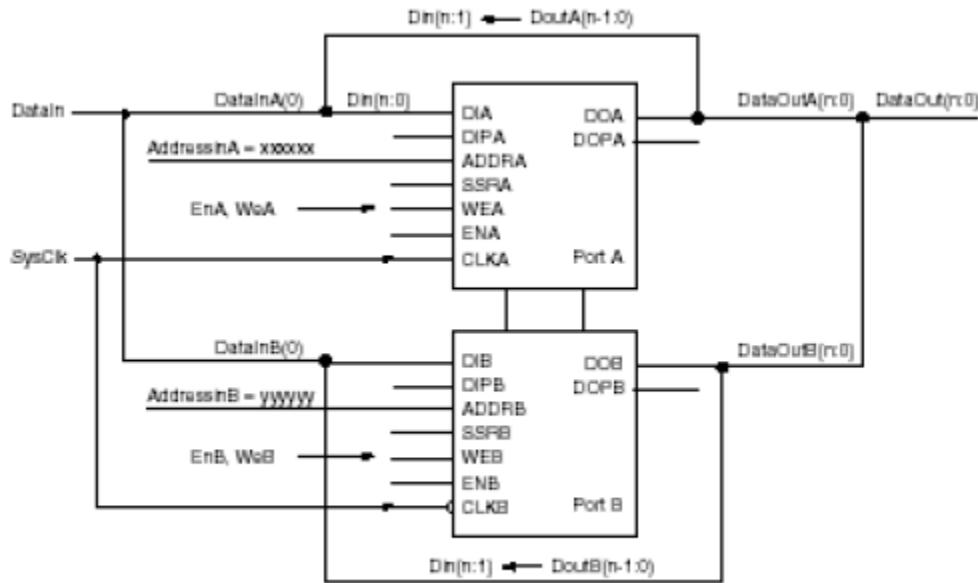


Figure 13-12 Dual Port Serializer with Interleaved DDR Output

Figure 13-13 provides the “other side” of the SERDES, the “deserializer”. The shifting structure, is the LUT flip flop, as before, with the loading control done in the LUT portion. Again, a counter is needed to determine when the shifter has been loaded and shifted to depletion, so another data item can be subsequently loaded for another transfer.

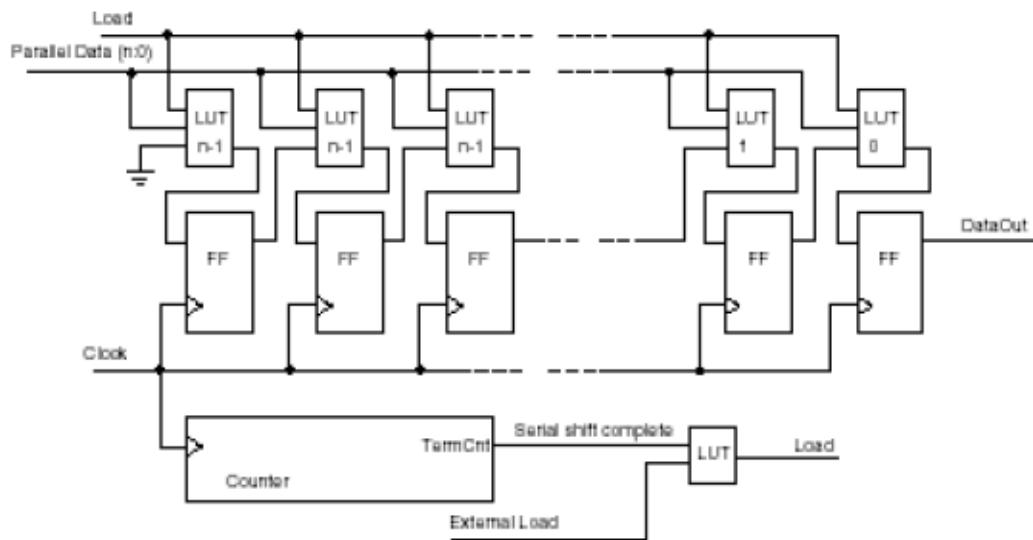


Figure 13-13 Generic Serializer

Building the structure with the BRAM is shown in Figure 13-14. Data enters through the DataIn multiplexer, and is shifted by assigning separate data out lines back to the data in lines of the BRAM. Shifting is multiplexed to occur, when not loading, on successive clocks.

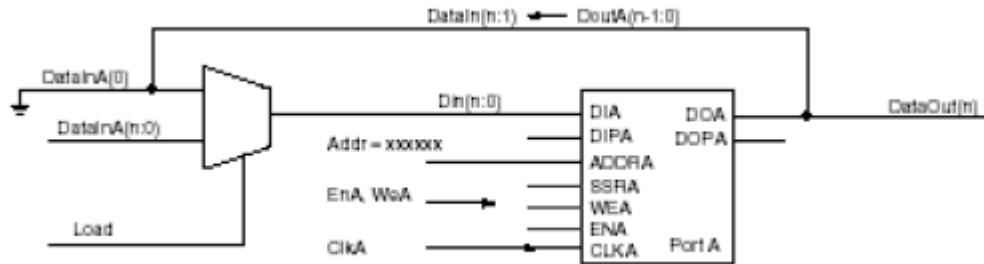


Figure 13-14 Single Port Serializer

It's also possible to combine both ports to obtain a double data rate output structure, along the lines of Figure 13-12. These details are completely described in XAPP 690.

For very high speed, designers will typically use multi gigabit transceivers (MGT) to perform the “serdes” function, but it is a welcome feature that slower speed channels need not waste this valuable resource, which can accomplish a similar slower version built from FPGA fabric and BRAMs.

### **Small Data Communication Subsystems**

Data communication systems can be viewed as hierarchical connections of smaller subsystems. We now shift our focus to slightly larger components that are concerned with switching input data streams to output data streams, which covers a lot of ground in the data communication world. First, we'll look at a high speed buffered crossbar switch design. Then, we will look at a serial backplane interface to shared memory, and finally, we will look at a mesh reference design. These three applications are discussed extensively in XAPP 240, 648 and 698. Other topologies are possible and have been built, including Benes networks and Banyan switches.

#### High Speed Buffered Crossbar Switch Design

Figure 15 shows a basic crossbar switch architecture. A simple crossbar switch consisting of “n” input lines and “m” output lines can make any connection from any input to any output through a set of  $n \times m$  connections. Many times, the input set matches the output set, so a common structure has a matrix of  $n \times n$  switches, permitting any connection from an input to any output. It is also possible to assign any input to multiple outputs, if desired. Figure 13-16 shows a small FIFO structure assigned to each cross-point, permitting data buffering between the inputs and outputs. Virtex family FPGA devices are particularly capable of creating this type of interconnect fabric

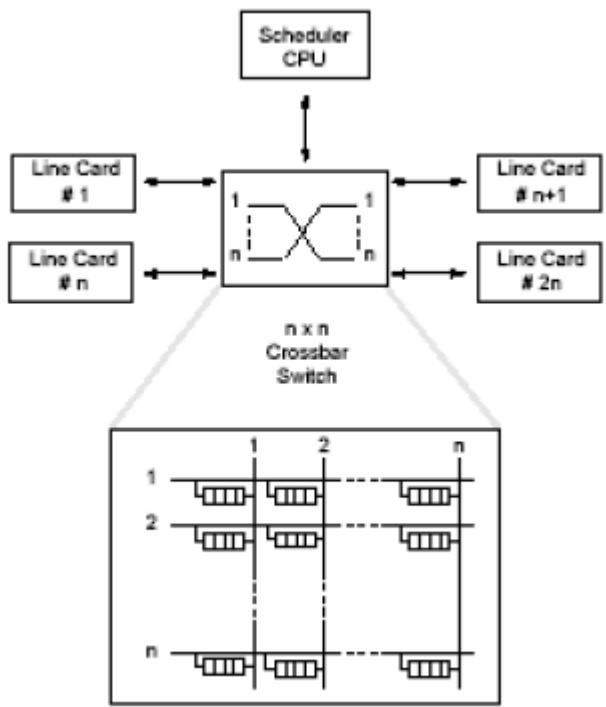


Figure 13-15 Buffered Crossbar Switch Architecture

by supplying dual port BRAMs that can build significant data buffers at the connect sites of the switch. High speed data routers benefit significantly from this building block, because it is inherently fast, “non-blocking” and scalable. Speed is largely gained through the sheer simplicity of the connections. The FIFO can mitigate head of line (HOL) blocking, and the regularity of the structure permits easy extension in both input and output directions.

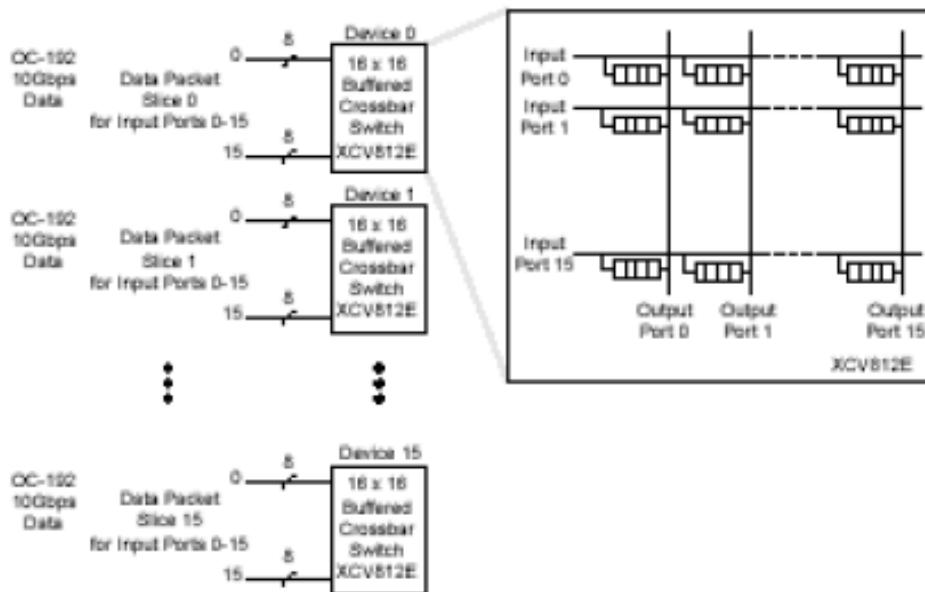


Figure 13-16 OC-192 16 x 16 Buffered Crossbar Switch

Figure 13-16 shows how multiple smaller Virtex devices can be assigned to create a high speed cross-point switch. It is typical of data communications to solve the bandwidth needs by replicating duplicate channels, and splitting data among them, for routing. The data is later reformatted as needed. This structure is capable of hitting 160 Gb/s as an aggregate bandwidth.

Figure 13-17 expands the internal detail to show how the switch matrix is built from multiple FIFOs getting multiplexed with the cascade/carry muxes, to produce the selected internal data paths. Multiple levels of flow control are also possible, with control signals PX\_AF manifesting as the FIFOs near a full condition. The FIFOs built from the BRAM structures are 36Mb in capacity, but timing needs only about 1Gb. This leaves ample available extra capacity to manage various latencies, as flow control becomes distributed through the switch, as well as other on board PCB time delays that may enter the equation.

Although this application was originally developed for the Virtex E family, additional advantage can be gained with the later families. Greater BRAM depth in the can add buffer capacity, and the built-in FIFOs on later BRAMs makes faster, more uniform behavior at very high speeds. Although this application is frequently done by ASICs in many systems, the ability to create a flexible, high speed solution on FPGA devices brings a clear advantage for updating and modifying overall behavior. FPGA devices permit additional dimensions of altering the various protocols used in assigning switch connections for service, which may not be possible with fixed ASIC structures.

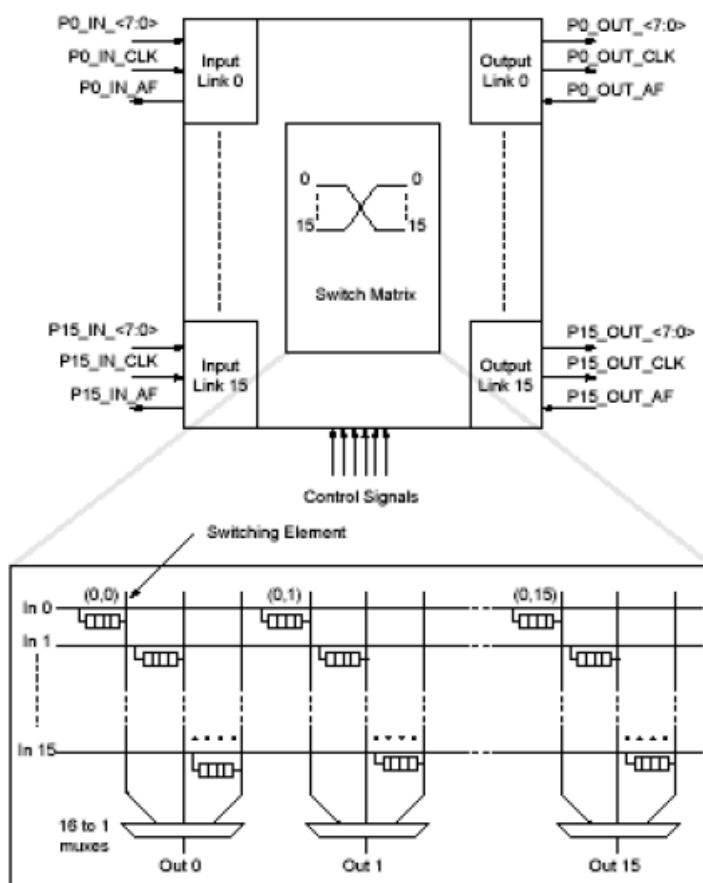


Figure 13-17 Buffered Crossbar Switch Internal Detail

## Serial Backplane Interface to Shared Memory

High speed data communication systems use massive memory stores to buffer files and packets, as they assemble, transmit, disassemble, re-transmit, etc. In earlier times, this capacity was met with the wide bandwidth of processor to memory buses. As high speed CMOS technologies evolved, switching noise of wide buses moving large voltage swings has been a problem for both bandwidth and error management. This problem has been largely resolved by moving to even faster, low voltage swing, serial interfaces, using differential signaling, to reduce noise by common mode rejection. To that end, developers gravitated to serial backplane formats to interchange data between line cards, and memories. XAPP 648 outlines a Xilinx reference design that uses the open-source Aurora Protocol Engine, to achieve a high speed serial backplane interface to a shared memory. Figure 13-18 show the basic arrangement.

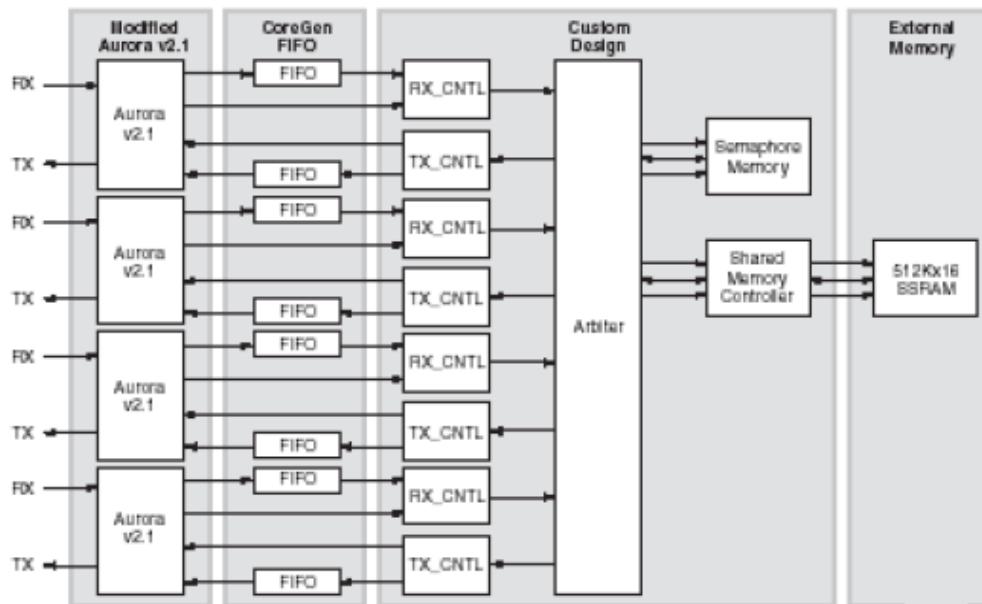


Figure 13-18 Serial Backplane Interface to Shared Memory Block Diagram

In Figure 13-18, Virtex FPGA devices are used to interface four serial channels to a common 512K x 16 SyncBurst SRAM. Clearly, independent input and output FIFOs are included in each receiver and transmitter data path, using CoreGen created FIFO structures in Virtex BRAM. Because the larger memory is shared, and has multiple sources vying for access, it delivers uniform service among the channels. The target protocol is packet based, so an additional “semaphore” memory holds pointer and access control information, to lock the shared memory for the prescribed packet transaction length. Full details are described in XAPP 648, but the design can be expanded and contracted to suit user needs. Some standard applications that might require this type of structure include:

1. Message passing between line cards
2. Shared storage for bridges in protocol converters
3. RAID controller mapping tables
4. Multiprocessor data storage

## Mesh Reference Design

Another interconnect topology is the mesh topology, shown in Figure 13-19.

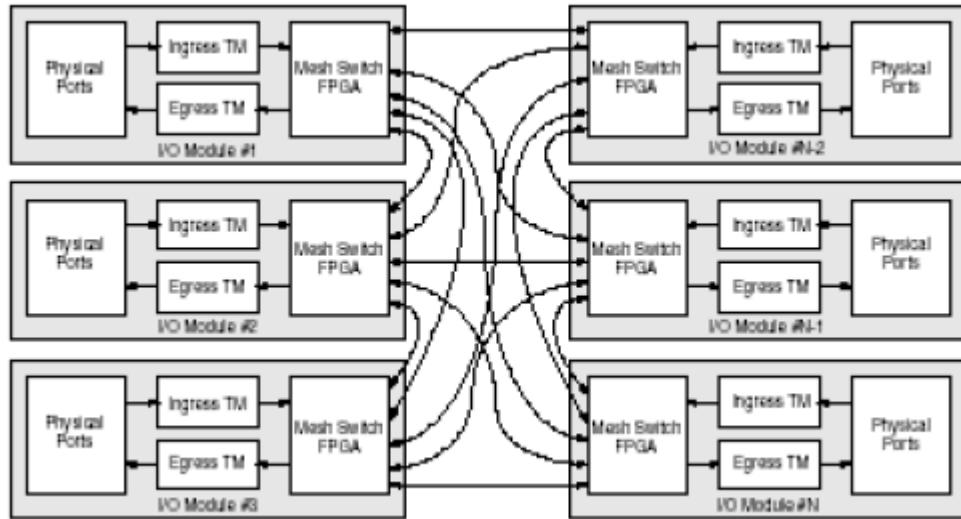


Figure 13-19 Mesh Topology

XAPP 698 details a full reference design for a Mesh Fabric switch. In the mesh topology, there is a private connection between every pair of modules within the switch. Once data connection is established, it may be transferred at the speed limit of the specific channel existing between module pairs.

## **Complete Data Communication Systems**

Including BRAM, MGTs and Power PCs on a single die with differential driver/receiver pairs, puts it entirely within the Virtex family realm to perform all seven levels of the OSI model for data communication (Figure 13-1). That being the case, it makes sense to at least outline some of the complete data communication systems that have been designed and verified on Virtex family FPGA devices. To that end, let's first look at a small web server, then a gigabit reference design, and finally the Virtex Tri-mode Ethernet MAC. All designs are fully documented and available for users to take as a starting point for their own variations.

### Web Server Using MicroBlaze

Using the inexpensive student Artix-7 device board, Arty™, one may create a web server. See:

<https://reference.digilentinc.com/learn/programmable-logic/tutorials/arty-getting-started-with-microblaze-servers/start>

This guide takes you through placing all the IP cores required to build a Linux based server. The PL includes the programmable logic, configuration logic, and associated embedded devices. The PS comprises the processor unit, on-chip memory, external memory interfaces, and peripheral connectivity interfaces including two gigabit Ethernet controllers (GEM), which access PL signals through the extended multiplexed I/O (EMIO) interface to connect different physical interfaces.

## Ethernet Performance and Jumbo Frame Support with PL Ethernet with Zynq SoC

A complete system is described in XAPP1082: PS and PL Ethernet Performance and Jumbo Frame Support with PL Ethernet in the Zynq-7000 AP SoC.

In the designs provided with this application note, the PS-GEM0 is connected to the Marvell PHY through the reduced gigabit media independent interface (RGMII), which is the default setup for the ZC706 board. The focus of this application note is the design of additional Ethernet ports. The designs described in this application note are:

- PS Ethernet (GEM1) that is connected to a 1000BASE-X or SGMII physical interface in PL through an EMIO interface
- PL Ethernet implemented as soft logic in PL and connected to the 1000BASE-X or SGMII physical interface in PL

Figure 13-20 shows the various Ethernet implementations on the ZC706 board. Note: The three Ethernet links cannot be active at the same time because the ZC706 board offers only one SFP cage for the 1000BASE-X or SGMII PHY. The PS-GEM0 is always tied to the RGMII Marvell PHY. The PS-GEM1 and the PL Ethernet share the 1000 BASE-X or SGMII PHY so only two Ethernet Links can be active at a given time. The 1000BASE-X/SGMII PHY and the GTX transceiver are part of the AXI Ethernet core for PL Ethernet design.

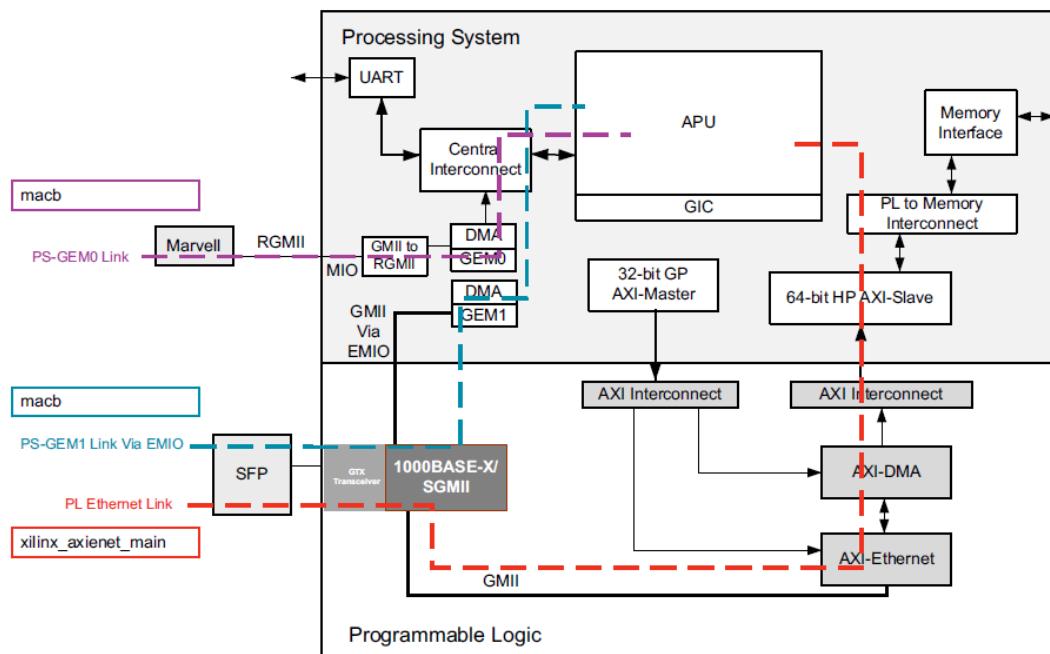


Figure 13-20 Zynq-7000 AP SoC Ethernet Interface

The software architecture is shown in Figure 13-21.

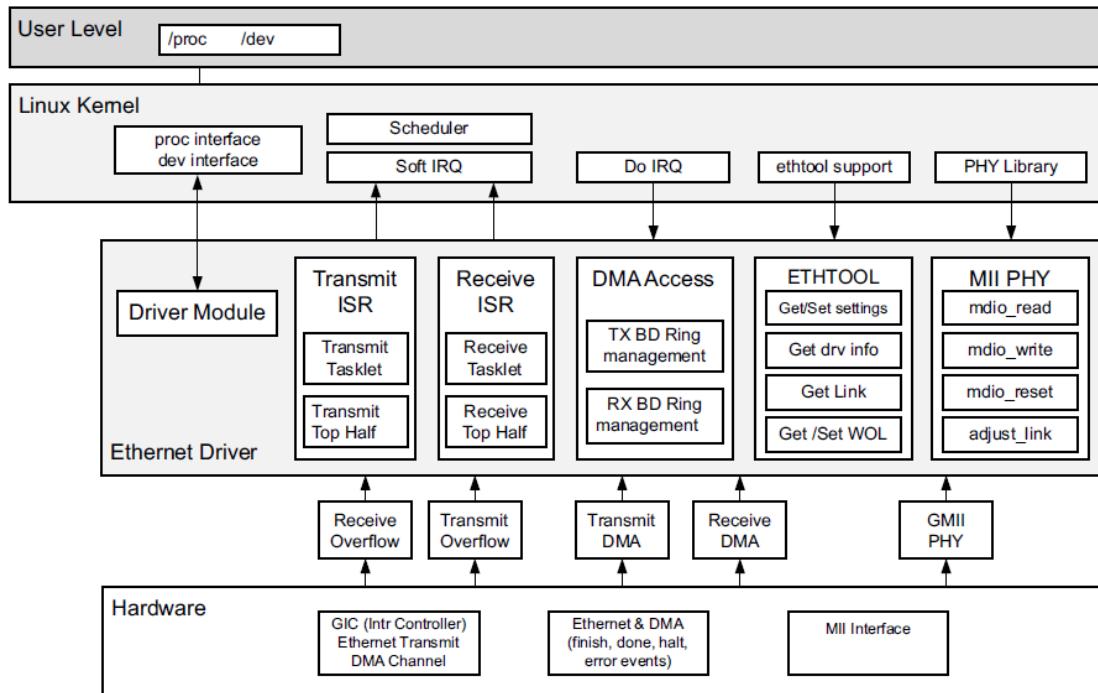


Figure 13-21 Software Architecture

### Comments on the Future

The field of data communications is moving in new directions, with The Xilinx SDAccel Development Environment. Data center operators constantly seek more server performance. Currently they develop applications with easy-to-program multicore CPUs and GPUs but CPU performance/watt is hitting the wall and GPU performance/watt is hitting the wall as well. Designers working on high-volume data-center applications including key acceleration, image recognition, speech transcription, encryption, and text search want GPU ease-of-programming but with hardware that will give them low power consumption, high throughput, and the lowest possible latency. However, there's a significant problem with scalability of multicore-CPU and GPU accelerators: developers would like to target simple full height plug-in PCIe® boards to use as application accelerators in data center servers. These boards can be configured to run high power graphics cards but customers want to target 25W or less to maximize scalability and minimize total power footprint.

Recent studies conducted jointly by Xilinx® and ETH Zurich, a Swiss university, have shown that FPGA-based application acceleration can achieve up to 25X better performance per watt and 50-75x latency improvement compared to CPU/GPU implementations while also providing excellent I/O integration (PCI, DDR4 SDRAM interfaces, high-speed Ethernet, etc.). In other words, FPGA devices provide the heart of what's needed for power-efficient hardware application acceleration on one chip while providing solutions that are below the 25W per board targets.

### Additional References

Because most of these designs are very serious attempts at providing users with a working solution, the supporting literature is lengthy. Many of the application notes/reference designs alone are longer than this brief chapter.

1. XAPP 201 An Overview of Multiple CAM Designs in Virtex Family Devices
2. XAPP 202 Content Accessible Memory (CAM) in ATM Applications
3. XAPP 203 Designing Flexible Fast CAMs with Virtex Family FPGAs
4. XAPP 204 Using Block RAM for High Performance CAMs
5. XAPP 225 Data to Clock Phase Alignment
6. XAPP 240 High-Speed Buffered Crossbar Switch Design Using Virtex-EM Devices
7. XAPP 247 Serial Digital Interface (SDI) Physical Layer Implementation
8. XAPP 264 Building OPB Slave Peripherals using System Generator for DSP
9. XAPP 289 Common Switch Interface CSIX-L1 Reference Design
14. XAPP 759 Configurable Physical Coding Sublayer
10. XAPP 764 Connecting Xilinx FPGAs to the Philips A-rate Fibre Optic Transceiver
11. BACKGROUNDER, SDACCEL DEVELOPMENT ENVIRONMENT

[http://www.xilinx.com/publications/prod\\_mktg/sdx/sdaccel-backgrounder.pdf](http://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf)

# **Chapter 15 Virtex Family Memory Controllers**

## **Introduction**

Microprocessor and microcontroller systems require sufficient memory to hold data and instructions for the task at hand. Early processors included the circuitry to directly attach to arrays of memory chips, but the problem became too complex and expensive. Varieties of memory evolved, which had distinct data handshakes, and no processor could interface to them all. For a while, memory controller chips existed, as well as ASIC and CPLD solutions. Again, the number of data handshakes (voltages, timing, drive strength, termination), was unwieldy. The situation got entirely too complex when five volts was no longer the standard.

With smaller featured silicon, the various memory technologies redefined their handshake needs, to suit the process sweet spots that they embraced to deliver their primary memory capability. Shrinking silicon features warranted smaller voltage swings, but bandwidth requirements grew. Data transactions were seldom single item transfers, and bursting became the norm, to reduce addressing and control overhead out of the memory cycles. Desperate for even greater performance at a given technology, memory developers began transferring data on both edges of a clock.

Clocking also became a major point of focus, to deliver clean data, when clocks are very fast, and noise must be rejected. Designers needed standard solutions, to common high performance memory structures, and Virtex family FPGA devices presented all the requisite features needed to solve the problem. An interesting evolution to trace is the features within the Virtex families and the prevailing memory standards. Both voltage standards and timing facilities have tracked well with the direction of memory technology. Earlier standards like SSTL-1 and HSTL-1 worked fine in the Virtex, -E and –EM families to support the available memories. As the memory structures evolved, so did Virtex capability.

Today's memories are well served with the improved SelectIO+ standards, and the new ChipSync capabilities that are designed to handle the timing requirements of the newer standards.

## **Overview**

This chapter presents the basic principles used in designing memory interfaces with Virtex FPGA devices. To that end, there are some general ideas to first discuss, followed by a high level discussion of several of the common solutions that exist – DDR3 SDRAM, and QDR-II+ SRAM. We conclude the discussion, by providing a table summarizing the general attributes of the offered solutions, and providing a reference summary so the reader can find the detailed solutions available on the Xilinx website.

## **Basics- Timing and Data Alignment**

The simplest memory interface is probably the basic static RAM interface. The handshake is simply driving a set of address lines with the necessary voltage levels and the target address, while asserting a chip enable (CE), a R/~W signal and directing the data bus transceivers according to the direction of R/~W. Of course, there are setup time requirements, and access time requirements that must be met by the logic, but it is very nearly that simple. Unfortunately, SRAM tends to be fast and simple, but not as dense as most applications require, so dynamic RAM is more attractive.

Versions of SRAM are still attractive, as densities there have increased with shrinking technology, but DRAM families are a primary focus with today's memory solutions. Standard DRAMs are seldom used, today, being replaced by Synchronous DRAMs (SDRAM), and more recently by a variety called Double Data Rate SDRAM (DDR SDRAM). Figure 15-1 shows the basic idea, for a Virtex based source synchronous memory interface.

Although we cover DDR3 here, DDR4 is newer, and is supported in devices after Virtex 7 (Virtex Ultra Scale at 20nm, and UltraScale+ at 16nm).

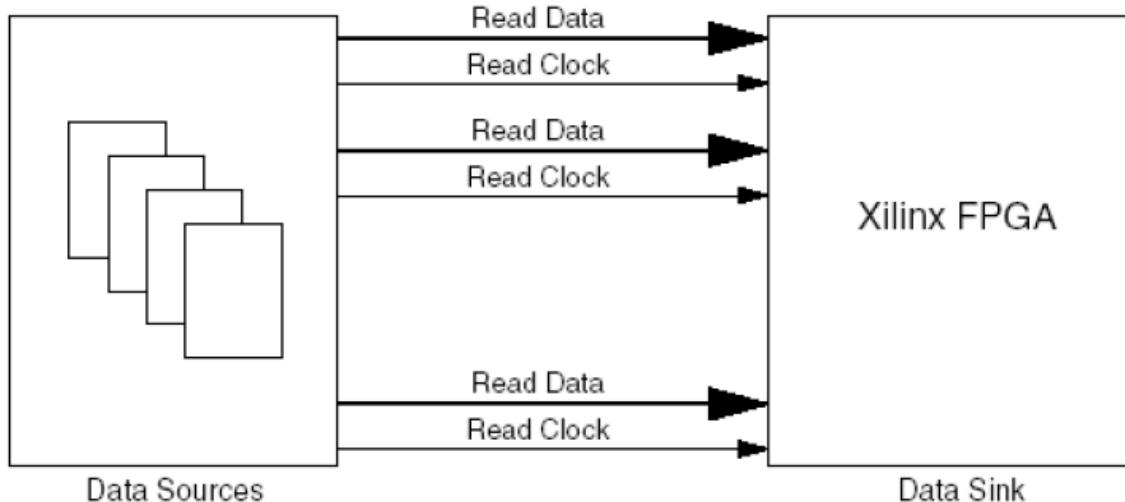


Figure 14-1 Source Synchronous Memory Interface Reading Data

As noted, multiple memories (data sources) are available to deliver or receive their data, interacting with the FPGA. To do this for multiple sources requires multiple independent I/O banks, capable of recognizing appropriate voltages, and each with independent timing to track the specific memory being referenced.

An appropriate way to deal with memory design is to attack the problem modularly, thereby reducing the problem to the key requirements of specific modules. To serve that goal, we resort to a generalized subdivision of requirements, similar to the OSI model of data communications. One such representation is shown in Figure 14-2.

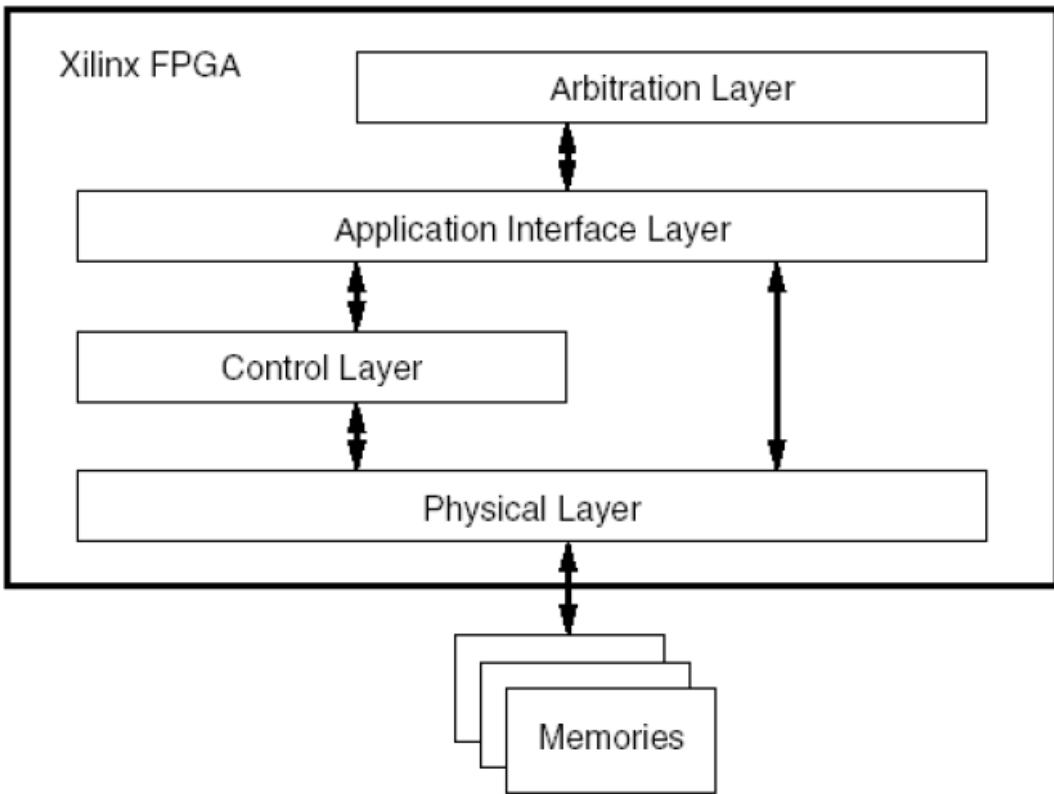


Figure 14-2 Modular Memory Interface Representation

This approach requires identifying aspects of the design and assigning the requirement to special features of the FPGA. For instance, the I/O pin voltage assignments must match the target memory requirements. That programming must be made in the FPGA design software. The timing for data, address and control signals must match the memory needs, and is assigned to FPGA timing control features - DCM, ChipSync, or fabric, as needed and available. Another aspect of the “control” portion of the design for DDR is that the memory attributes require being initialized, and data transfers occur as prescribed by commands. This memory programming gives the flexibility to dictate the size of data burst, and setup other memory parameters. Interfacing to the user’s “back end” with arbitration among multiple memory blocks is typically relegated strictly to the FPGA fabric.

One early Virtex capability that has remained with the family is the double data rate output register, available at each IOB. Figure 14-3 shows how the DDR IOB can deliver the appropriate clock to the memory, for a DDR-1 type memory interface.

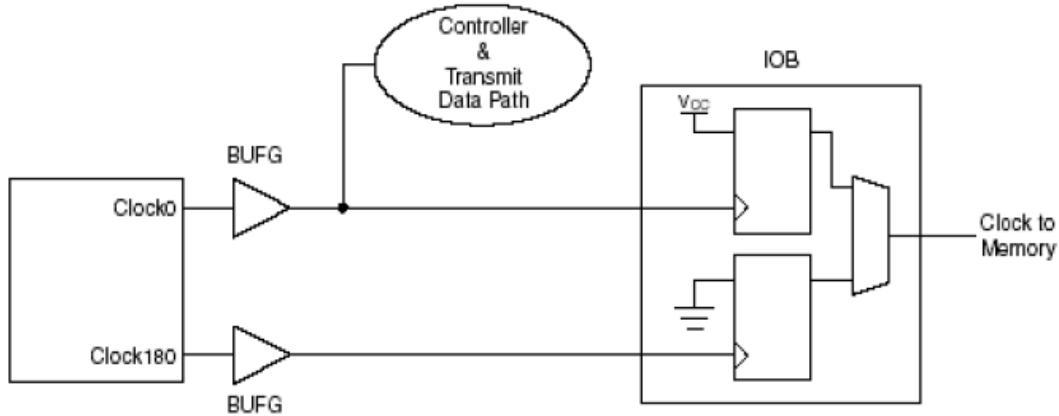


Figure 14-3 Clock Generation for DDR Memory

As shown, both the Clock0 and the Clock180 outputs are attached to BUFG drivers, then attached respectively to the two edge triggered flip flops, which toggle on the rising edge. Because each flip flop is initialized out of phase (VCC to one and GND to the other), they will remain out of phase through every clock cycle, and the DDR multiplexer combines them to deliver the clock to the Memory. By using the same clock for latching data into output registers, and various command and control values into registers, all actions maintain their by relative synchronization, as they track through nearly identical time delays. Proper timing design requires a detailed timing budget analysis. Figure 14-4 shows the relationship between timing and control signals for an older DDR-1 design, and Table 14-1 shows the timing budget and margin analysis for a 200 MHz design.

The time values shown in Table 14-1 are in picoseconds, where the clock cycle time is 5 nanoseconds (alternately, 5000 picoseconds). In a DDR transaction, the standard approach is to align the delivered data to be centered between clocking events. Assuming a 50% duty cycle, this permits data being clocked on a rising edge to have the same setup and hold requirements as data clocked on the falling edge. In Figure 14-4, there are “uncertainties” and “margins”. Uncertainty diminishes margin. All effort should increase margin and decrease uncertainty. Jitter is an example of uncertainty.

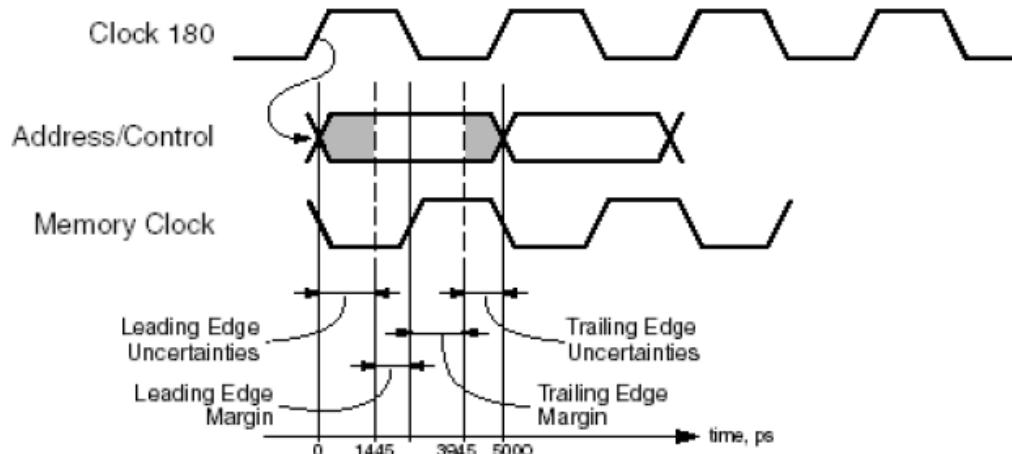


Figure 14-4 Address and Control Signal Uncertainty and Margin Identification

Parameter	Value	Leading-Edge Uncertainties	Trailing-Edge Uncertainties	Meaning
T <sub>CLOCK</sub>	5000			Clock period
T <sub>CLOCK_SKW</sub>	50	50	50	Minimal skew, since right/left sides are being used and the bits are close together
T <sub>PACKAGE_SKW</sub>	65	65	65	Using same bank reduces package skew
T <sub>SETUP</sub>	750	750	0	Setup time from memory data sheet
T <sub>HOLD</sub>	750	0	750	Hold time from memory data sheet
T <sub>PCB_LAYOUT_SKW</sub>	50	50	50	Skew between layout lines on the board
T <sub>PHASE_OFFSET_ERROR</sub>	140	140	140	Offset between different phases of the clock. Taken from the parameter CLKOUT_PHASE
T <sub>DUTY_CYCLE_DISTORTION</sub>	0	0	0	Duty-cycle distortion does not apply
T <sub>JITTER</sub>	0	0	0	Since the clock and address are generated using the same clock, the same jitter exists in both. Therefore, it does not need to be included
Total Uncertainties		1055	1055	
Command Window	2500	1445	3945	Worst-case window of 2500 ps

Table 14-1 Address and Control Signal Margin Analysis

In terms of writing the data, Figure 14-5 shows a high level diagram of the signal relationships. The memory clock is taken as 0 degrees, the data is written with respect to 90 and 270 degree events, with the DQS strobe tracking the memory clock. Figure 14-6 shows how data is multiplexed out onto a DDR IOB, using the 90 and 270 degree phase delays from the DCM delivered clock.

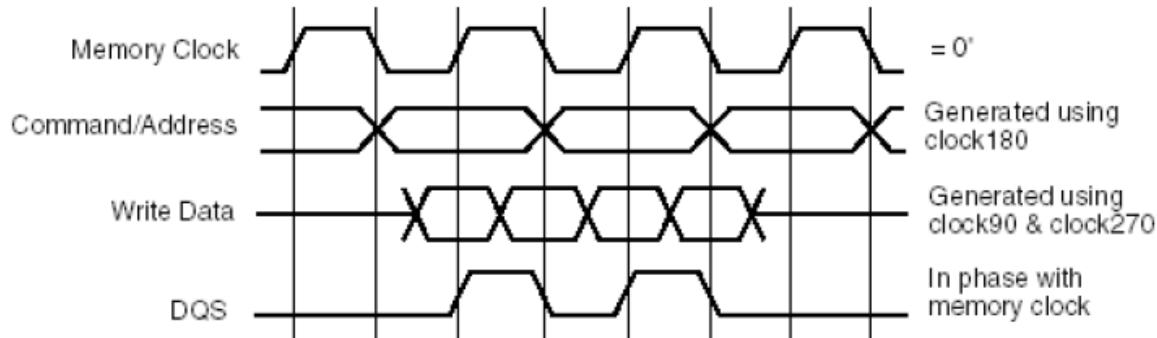


Figure 14-5 Write Data, Command, Address, DQS Strobe and Clock Signals

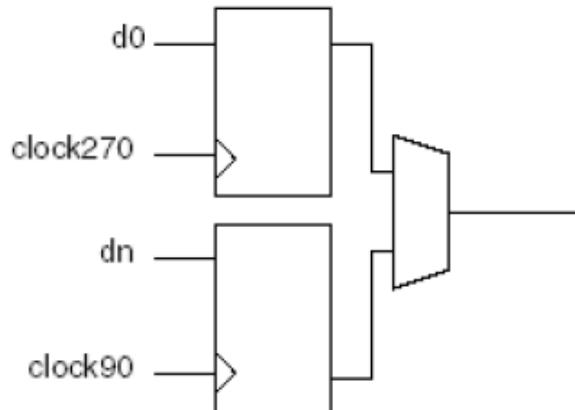


Figure 14-6 DDR IOB Register Driving “Write” Data

Similar to the Address and Control Signal margin analysis is the Write Data margin analysis, shown in Figure 14-7. Again, all variable delay parameters need to be identified and accounted for, which includes both package variables as well as printed circuit board delays.

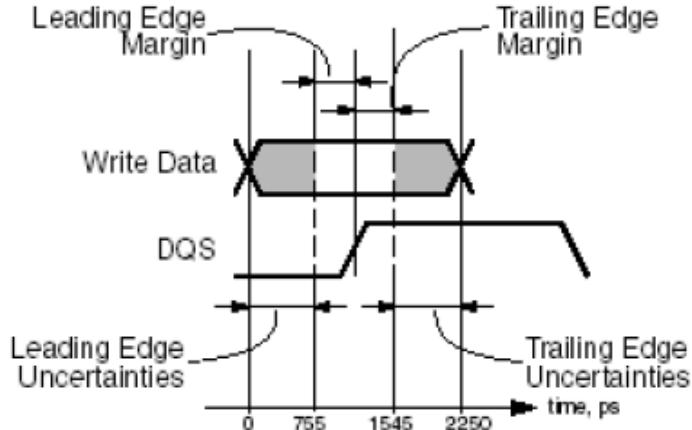


Figure 14-7 Write Data Uncertainty and Margin Identification

Parameter	Value	Leading-Edge Uncertainties	Trailing-Edge Uncertainties	Meaning
$T_{CLOCK}$	5000			Clock period
$T_{CLOCK\_PHASE}$	2500			Clock phase
$T_{DCD}$	250			Duty cycle distortion of clock to memory
$T_{DATA\_PERIOD}$	2250			Total data period, $T_{CLOCK\_PHASE} - T_{DCD}$
$T_{CLOCK\_SKEW}$	50	50	50	Minimal skew, since right/left sides are being used and the bits are close together
$T_{PACKAGE\_SKEW}$	65	65	65	Skew due to package pins and board layout. This can be reduced further with tighter layout
$T_{SETUP}$	450	450	0	Setup time from memory data sheet
$T_{HOLD}$	450	0	450	Hold time from memory data sheet
$T_{PCB\_LAYOUT\_SKEW}$	50	50	50	Skew between layout lines on the board
$T_{PHASE\_OFFSET\_ERROR}$	140	140	140	Offset error between different clocks from the same DCM. Taken from the parameter CLKOUT_PHASE
$T_{JITTER}$	0	0	0	The same DCM is used to generate the clock and data. Hence they jitter together
Total Uncertainties	1205	755	755	Worst case for leading and trailing can never happen simultaneously
Window	740	755	1495	Total worst-case window is 740 ps

Table 14-2 Summarizes values, again for a 200 MHz memory clock.

Table 14-2 Write Data Margin Analysis DDR memories are called “source synchronous”, which means that the synchronizing signal is delivered by the data source in any data transaction. Both ends of the transaction typically have delay lock loop synchronizing blocks, and the FPGA may be creating the clock for the whole system. Whichever side is providing data must also provide the synchronization signal along with the data, to the receiving end, and that pulse is DQS. Managing DQS so that it is properly aligned and centered to the data can be a problem. Multiple ways to do this exist. One way is to use PCB metal to create the delay. Another is to use an external time delay module. Still another is to cascade LUTs to create a delay that can be tuned to meet the desired value. This is shown in Figure 14-8. The time delay for

a signal using LUT based method will be approximately one to seven TILO values. More could be cascaded, as needed (not recommended!).

Using the local feedback sites within a CLB would minimize routing delays, and make for very uniform delay taps.

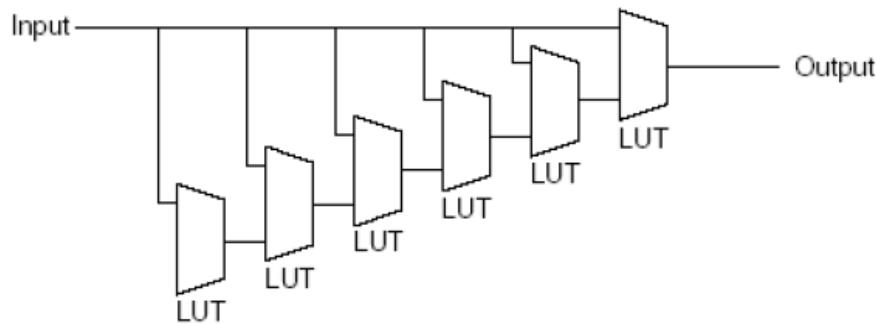


Figure 14-8 LUT Based Time Delay Circuit

When reading the data, the signal that synchronizes it is the DQS, just mentioned. It is not a free running clock, but occurs once, with each data item. Hence, the data must be delivered to a flip flop with appropriate strobe alignment, to “catch” the data. Figure 14-9 shows the Read Data Valid Uncertainty and Margin Identification, and Table 14-3 provides the appropriate Read Data Valid Analysis.

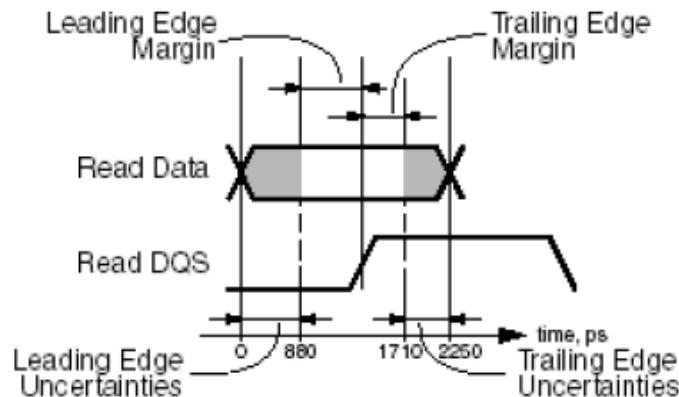


Figure 14-9 Read Data Valid Uncertainty and Margin Identification

Parameter	Value	Leading-Edge Uncertainties	Trailing-Edge Uncertainties	Meaning
T <sub>CLOCK</sub>	5000			Clock period
T <sub>PHASE</sub>	2500			Clock phase
T <sub>MEM_DCD</sub>	250			Duty cycle distortion from memory DLL
T <sub>DATA_PERIOD</sub>	2250			Total data period, T <sub>PHASE</sub> - T <sub>MEM_DCD</sub>
T <sub>DQSQ</sub>	500	500	0	Strobe-to-data distortion
T <sub>PACKAGE_SKEW</sub>	65	65	65	This parameter depends on the exact package. Since the 8 data bits are close together, skew is less than this
T <sub>SETUP</sub>	240	240	0	Setup time from Virtex-II Pro data sheet for -6 part. Taken from T <sub>DICK</sub> parameter.
T <sub>HOLD</sub>	-50	0	-50	Hold time from Virtex-II Pro data sheet. Taken from T <sub>CKDI</sub> parameter.
T <sub>JITTER</sub>	100	0	0	Data and strobe jitter together, since they are generated off the same clock
T <sub>LOCAL_CLOCK_LINE</sub>	25	25	25	Observed skew is lower than this value, since loading is light and all bits are close together
T <sub>PCB_LAYOUT_SKEW</sub>	50	50	50	Skew between data lines on the board
T <sub>QHS</sub>	450	0	450	Hold skew factor for DQ
Uncertainties		880	540	
Window	830	880	1710	Worst-case window of 830 ps

Table 14-3 Read Data Valid Margin Analysis

In order to capture data and make it available for a user application, multiple capture registers are needed. Figure 14-10 shows how a set of four such registers would operate, moving multiple data operations and forwarding to a user interface. To simplify the interface to a user application, a set of FIFOs minimize the issues that can arise in timing. See Figure 14-11.

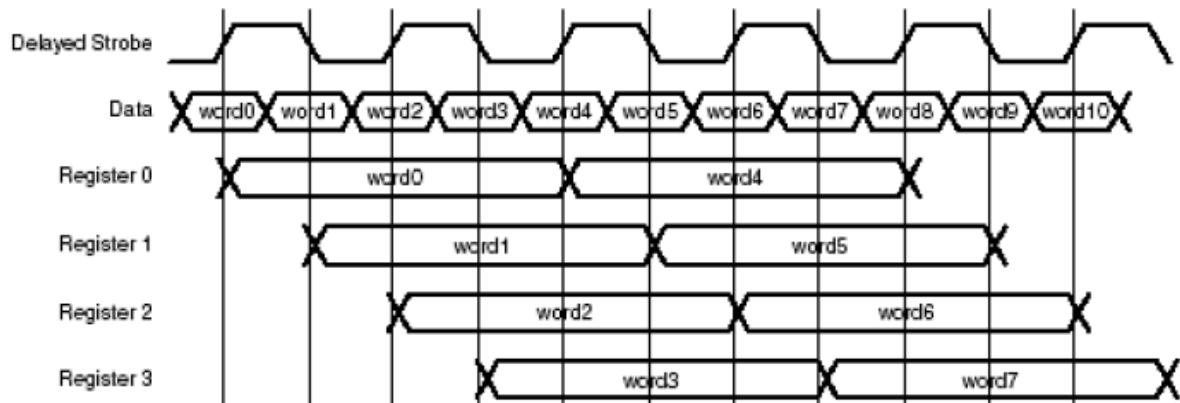


Figure 14-10 Data Capture into Four Register

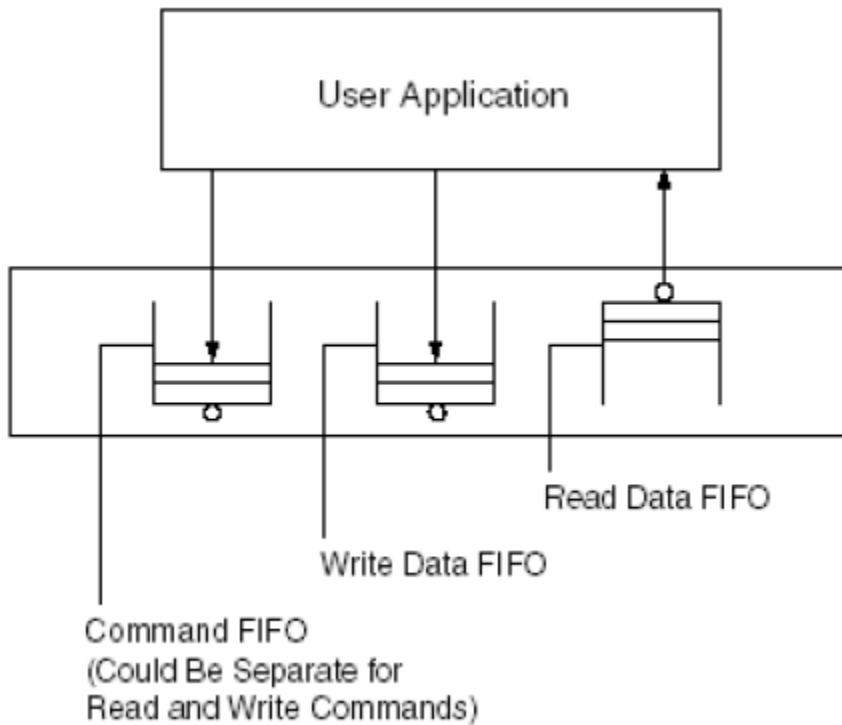


Figure 14-11 Using FIFOs to Interface to the User Application

Note in Figure 14-11, the presence of a Command FIFO, which delivers the Read and Write commands, as well as the initialization information, after power up. Timing and alignment techniques just described are appropriate for the early Virtex families, but after Virtex 4 offers additional capability – ChipSync, making it easier to delay the input data, than to align the DQS strobe. Figure 14-12 details two situations for using the Idelay capability of ChipSync to sandwich an incoming pulse, the DQS. Remember, that Idelay is designed to operate with an external 200 MHz clock, calibrating its internal time delay taps to ~78 picoseconds each. There are 64 possible taps, and the task of the Idelay controller is to delay an incoming signal to a given number of taps.

Typically, we seek to find the rising and falling edge of a pulse, then split the time difference between edges, to closely estimate the pulse center. The strategy here is to “train” the Idelay circuit with several successive, but artificial “Read” requests (i.e. Dummy\_rd\_en), which are used to identify a tap setting for Idelay, resulting in the data being centered around the DQS edge. Actual reading will be done with the internal FPGA clock, after the training. Figure 14-13 shows the DQS edge detection strategy logic. Note from Figure 15-12, that two cases must be resolved by the logic – dummy read occurring in the middle of DQS and dummy read occurring after DQS. Also, due to the speed limitation on Idelay taps, there is a lower limit on what clock speeds can work, around 100 MHz. All 64 taps of Idelay, at ~78 picoseconds each is roughly 5 nanoseconds, which is about half a period for 100 MHz.

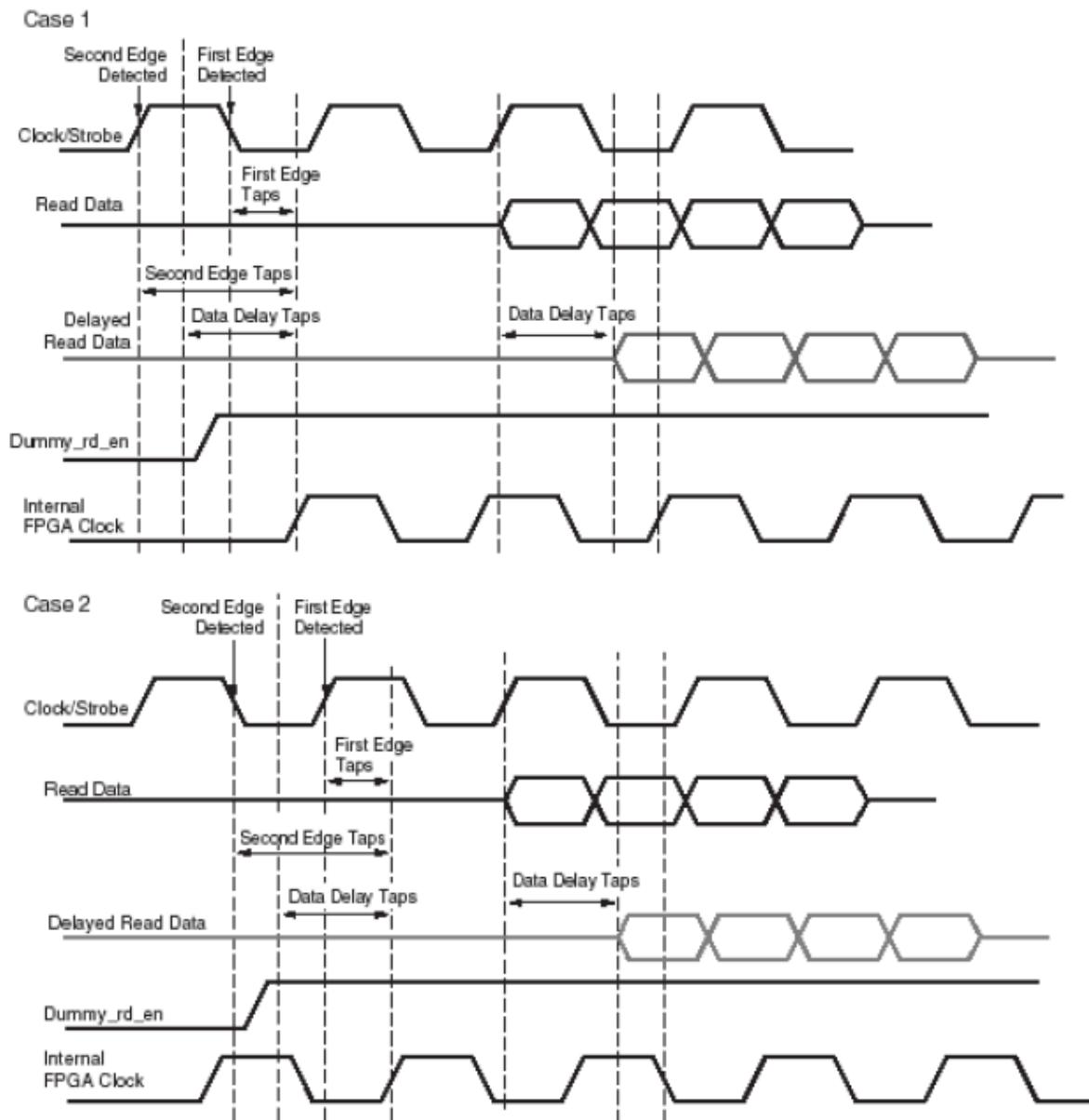


Figure 14-12 Clock Strobe (DQS) Center to FPGA Clock Phase Detection

The Edge Detection logic task is to identify the number of taps needed to find the DQS signal center position. Once found, the Data Idelay Tap Control Logic connects to the input Data's Idelay circuitry and shifts the data by the same number of taps. Figure 14-14 shows that circuitry. It is important to recognize that the right hand logic in Figure 14-13 connects to the lower right hand logic of Figure 14-14, shifting the incoming data, so it synchronizes with the FPGA internal clock.

The technique of using Idelay to center the data is called direct clocking. Also shown in Figure 14-12, on the right are two Read Data FIFOs – one capturing on the rising clock edge and the other capturing on the falling clock edge. These FIFOs can be built from SRL16 LUT RAM structures, and the outputs forwarded to the FPGA fabric.

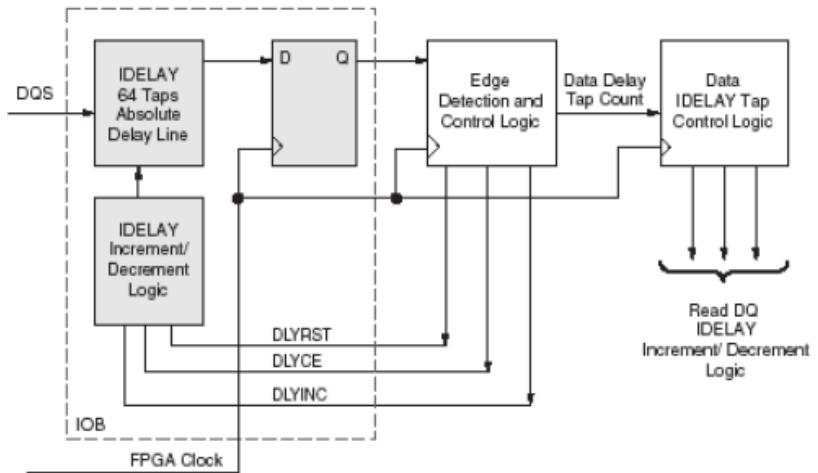


Figure 14-13 Strobe Edge Detection

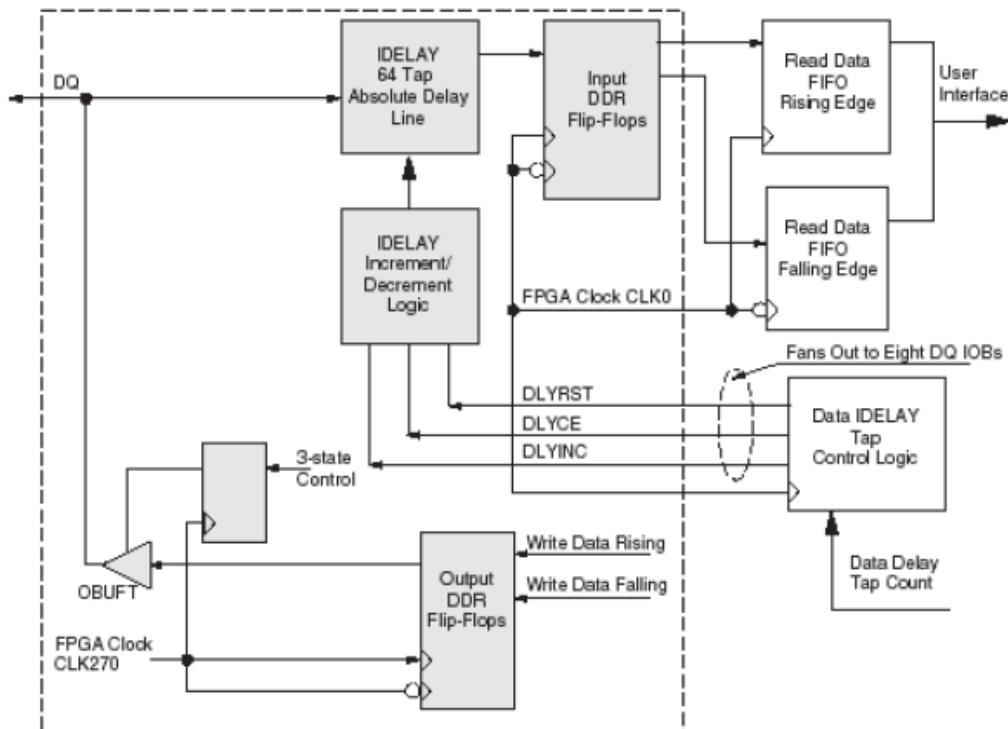


Figure 14-14 Read/Write Data Path

## VIRTEX 7 DDR3 SDRAM Controller

A challenging aspect of the design is the logic-fabric-based implementation of the memory controller. With increasing DDR memory data rates (2X the memory clock rate), the logic-fabric-based controller that manages memory commands and data flow must run at higher rates as well. With each successive device generation, the memory clock rates for each DDR architecture generation have increased at a faster pace than logic fabric performance, creating a gap between required memory clock rates and the rates at which logic-fabric-based controllers can be clocked (see Figure 14-15).

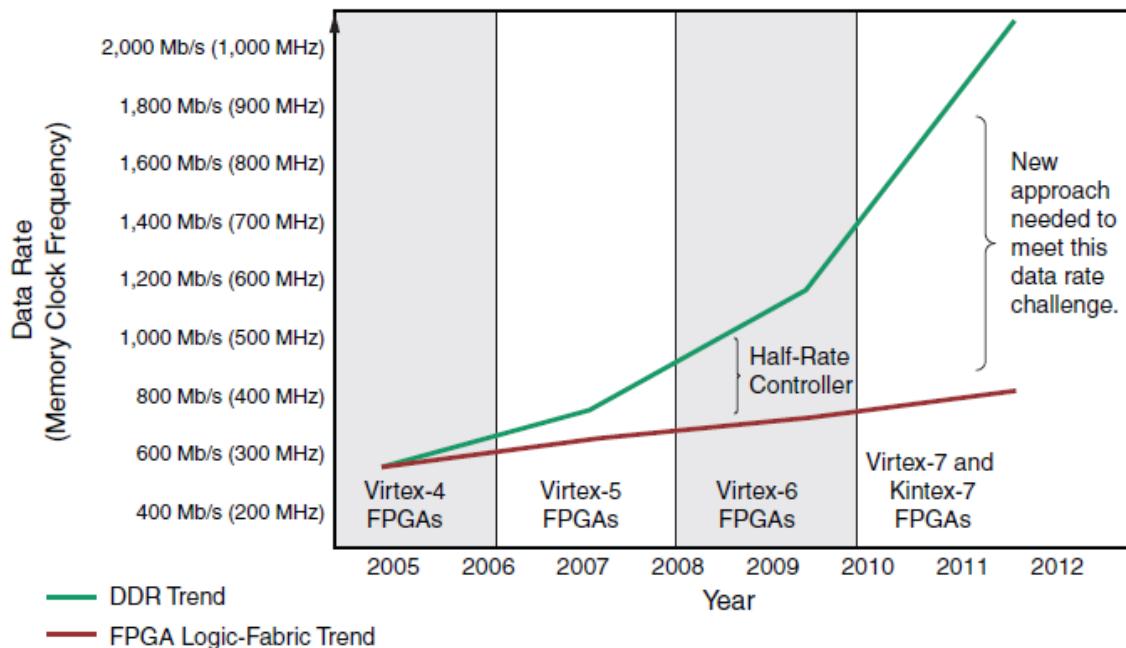


Figure 14-15 Data Rate Challenge for Logic-Fabric-Based Controllers

In previous-generation products (such as Virtex-6 FPGA devices), a DDR3 controller running at half the rate of the memory clock is implemented to meet the logic fabric timing limitations. This implementation provides a good match between the data rate capabilities of the I/O and the logic fabric clock rate capabilities needed for the controller implementation. In the 7 series and Zynq devices, however, a new architecture bridges the wider gap between DDR3 data rates of up to 1.866 Gb/s (needed in high-performance applications) and the logic fabric clock rate.

### Memory Interface Architecture

Xilinx has improved the architecture of the PHY layer memory interface and controllers to achieve data rates of 1.866 Gb/s for mid speed grade FPGA devices and AP SoC's. The efficiency of the DDR3 controller design has also been improved to achieve higher effective bandwidth.

As shown in Figure 14-16, the architecture of the memory controller and interface incorporates three functional modules: the PHY, the DDR3 memory controller, and the user interface.

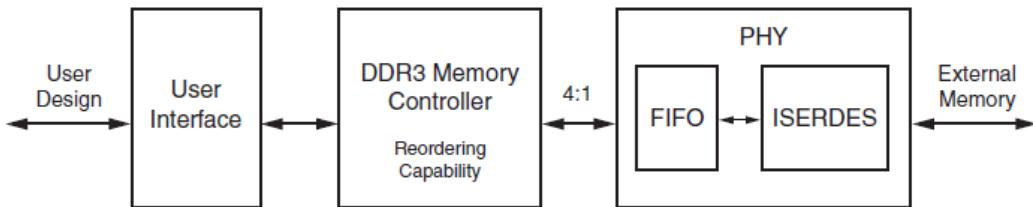


Figure 14-16 DDR3 Controller and Interface Architecture

For DDR3 interfaces that run at a data rate of 1.866 Gb/s, the clock rate of the logic-fabric-based controller needs to be  $\frac{1}{4}$  the memory clock rate to implement controller state machines in the logic fabric that can keep up with the 933 MHz DDR3 memory clock rate. The improved PHY architecture makes this possible with a dedicated FIFO that provides the gearbox capability to decouple the memory clock rate from the logic-fabric-based controller clock rate at an appropriate ratio. This ratio can be 4:1, or for lower memory clock rates, 2:1. DDR3 interfaces that require clock rates of over 800 MHz (1,600 Mb/s) are suitable for 4:1 decoupling ratios, while other interfaces like QDR-II+ or RLDRAM II running at rates below 550 MHz can be better matched with a 2:1 decoupling ratio.

#### Optimizing the Physical Layer for Higher Data Rates

The PHY is responsible for capturing read data and for transferring write data from the controller to the external memory devices. The I/O drivers must provide sufficient drive strength and switching speed for data rates up to 1.866 Gb/s supported by the highest speed grade DDR3 devices. An innovative I/O architecture ensures drive capability with excellent signal integrity at these high rates. A 2V pre-driver voltage option (VCCAUX\_IO) was needed to ensure sufficient signal gain to meet the drive requirements for the 1.866 Gb/s rates. See Figure 14-17.



Figure 14-17 2.0V Pre-driver (VCCAUX\_IO) Voltage Supply Option

In addition to enabling the capabilities of the I/O driver and receiver to switch at these high data rates, the read data capture and real-time calibration require a dedicated clocking circuit that can manage the initial calibration of the clock to the data valid window and maintain this relationship over changes in voltage and temperature during system operation. 7 series FPGA devices and Zynq-7000 AP SoC's incorporate a new clocking structure as part of the clock management tile (CMT) called the Phaser that has all the built-in capabilities to control and maintain the clock-to-read-data timing with up to 7 ps precision. Figure 14-18 outlines the I/O block and CMT-to-logic fabric signal relationship that constitutes the basis of the PHY architecture. The system clock is driving the phase-locked loop (PLL) in the CMT, and that in turn drives the clock through the BUFG for the memory controller. A PLL output also drives the Phaser

control and the PHASER\_IN and PHASER\_OUT blocks. The PHASER\_IN performs two primary functions: it delays the DQS, which is then used to capture the data (DQ in Figure 5) in the ISERDES; and it controls the transfer of the data from the ISERDES to the IN\_FIFO block. The FIFO has two modes, 1:2 and 1:1. The 1:2 mode further decouples the high data rates of the I/O from the logic fabric that needs to run at a lower rate. With the 1:4 ISERDES and 1:2 FIFO transfer ratios, the data is effectively running single data rate at  $\frac{1}{4}$  of the memory clock rate. This is a necessity when implementing data rates higher than 1,600 Mb/s or 800 MHz. The memory controller needs to run only at  $\frac{1}{4}$  of the 800 MHz rate. This 1:4 decoupling of the two clock systems (memory and controller) provides the benefit of easier timing closure. For 1.866 Gb/s data rates or 933 MHz clock rates, the memory controller runs at only 233 MHz. The logic-fabric-based controller state machine can meet timing if the clock rates are kept in this range.

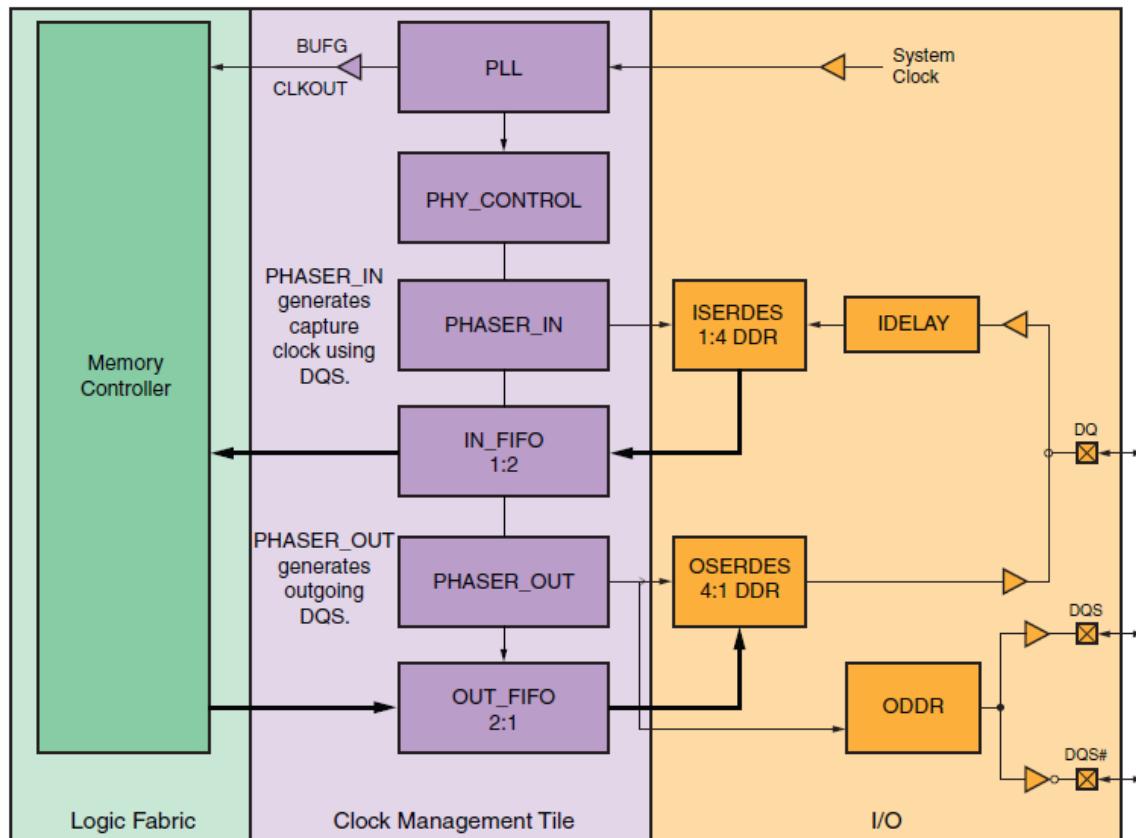


Figure 14-18 PHY Architecture and Data Transfer to the Logic-Fabric-Based Memory Controller

The output data path and clocking are similar to the input path, but the PHASER\_OUT and OUT\_FIFO are used to clock and control the data out transfer. The PHASER\_OUT also controls the generation of the outgoing DQS signal.

## QDR-II+ SRAM Interfaces

Quad data rate SRAM raises the performance bar substantially. Primarily targeted at the data communication applications - memory buffers, look up tables, traffic managers and linked lists, these devices manage four data transactions in a single clock period. Clocking at 633 MHz, 36 bit QDR-II+ SRAM devices can transfer up to 22.8 Gb/sec.

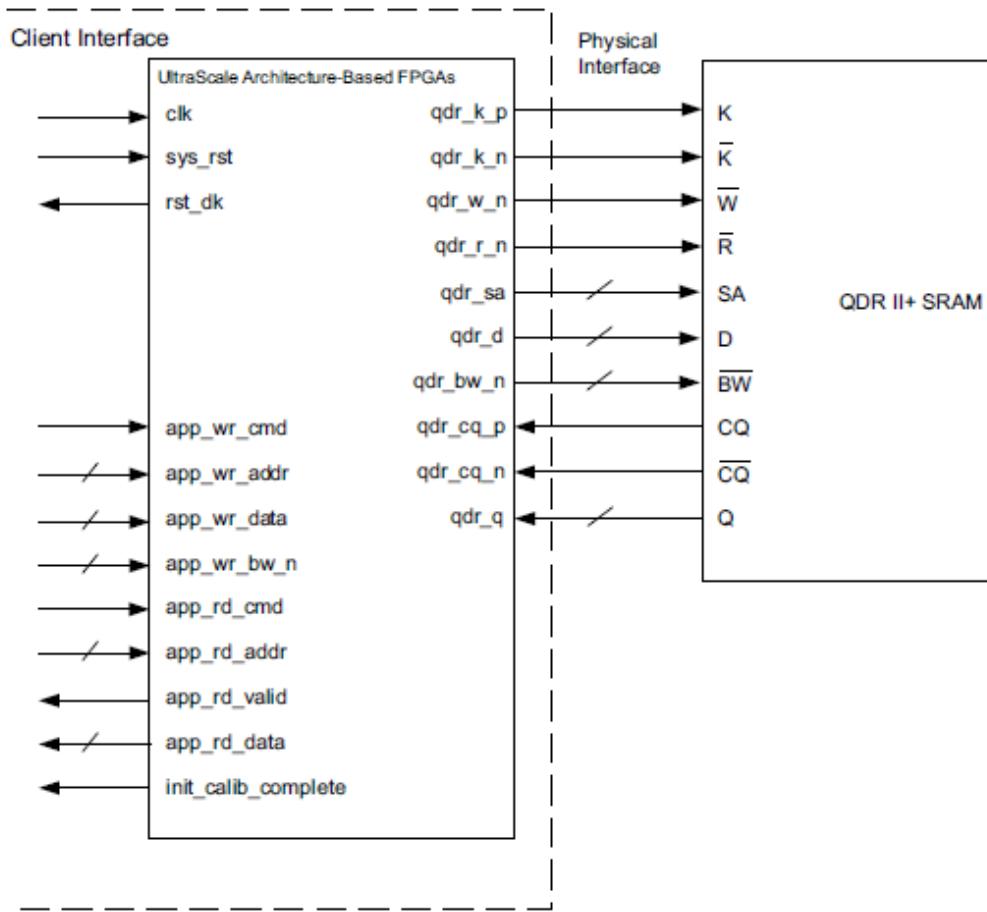


Figure 14-19 High-Level Block Diagram of QDR II+ Interface Solution on UltraScale+

The physical layer includes the hard blocks inside the FPGA and the soft calibration logic necessary to ensure optimal timing of the hard blocks interfacing to the memory part. These hard blocks include:

- Data serialization and transmission
- Data capture and deserialization
- High-speed clock generation and synchronization
- Coarse and fine delay elements per pin with voltage and temperature tracking

The soft blocks include:

- Memory Initialization – The calibration modules provide an initialization routine for the particular memory type. The delays in the initialization process can be bypassed to speed up simulation time if desired. The QDR II+ memories do not require an elaborate initialization procedure. However, you must ensure that the Doff\_n signal is provided to the memory as required by the vendor. The

- QDR II+ SRAM interface design provided by the memory wizard drives the Doff\_n signal from the FPGA. After the internal MMCM has locked, the Doff\_n signal is asserted High for 100  $\mu$ s without issuing any commands to the memory device. For memory devices that require the Doff\_n signal to be terminated at the memory and not be driven from the FPGA, you must perform the required initialization procedure.
- Calibration – The calibration modules provide a complete method to set all delays in the hard blocks and soft IP to work with the memory interface. Each bit is individually trained and then combined to ensure optimal interface performance. Results of the calibration process is available through the Xilinx debug tools. After completion of calibration, the PHY layer presents raw interface to the memory part.

Please refer to the references for more details.

### Conclusions and Further Reading

From a highly detailed analysis viewpoint, we have only skimmed the surface of several application notes. The full picture, as well as the design files, are available for users to download, and run their own simulations, or build their systems. Xilinx also maintains a web page referred to as the Memory Corner, where the latest memory interfaces and revisions to older designs can be found. In all, every memory base appears to be covered.

#### References:

1. Achieving High Performance DDR3 Data Rates WP383 (v1.2) August 29, 2013 Adrian Cosoroaba
2. UltraScale Architecture-Based FPGAs Memory IP v1.2 LogiCORE IP Product Guide Vivado Design Suite PG150 June 8, 2016
3. 7 Series FPGAs Memory Interface Solutions User Guide UG586 March 1, 2011
4. <http://www.xilinx.com/products/technology/memory-interfacing.html>

# Chapter 15 Digital Signal Processing with Virtex FPGA Devices

## Introduction

Digital Signal Processing (DSP) has deep roots. Filtering signals with discrete calculations clearly goes back to before 1960, but it was probably around then that the IEEE and the British IEE publications began providing a lot of published literature on the topic. Bell Labs was certainly the center of the DSP universe in those days. Some have suggested that Euler was aware of the Discrete Fourier Transform, but such things may be tough to substantiate. Although we can trace the beginnings of FPGA-like structures to the work of John von Neumann in the 1940's, it's likely that DSP is even older. Nonetheless, mathematicians have been discussing the task of slicing problems up and calculating a result from tiny interconnected building blocks for a long time. "Divide and conquer" goes back even further.

Xilinx got started with the very early FPGA devices, but the XC4000 family was the first to have a logic block that included adder chains. The importance of high-speed adders cannot be overstressed. Addition is at the heart of most DSP operations, with multiplication coming in next. The combination of the two makes for a powerful capability. The flexibility of creating just the right size of adder/multiplier blocks is also important, in that designers need never build more logic than is required. It also lets lots of short cuts be taken, resulting in the ability to speed up the overall process. DSP on Virtex family FPGA devices is also intuitively satisfying because it is possible to directly translate signal flow diagrams to hardware. You can "point at" the logic that is the adder, and the logic that is the multiplier, etc. The presence of flexible BRAM or Distributed RAM resources for holding operands inside the chip, also reduces the need to go off chip for intermediate value retention. The presence of flip flops in every slice element permits direct pipelining of intermediate results to gain the fastest clock speeds.

This chapter looks at some of the DSP building blocks, and the support flows that are available for algorithm development and verification. The Xilinx DSP "toolbox" is vast and powerful. The dream of putting high speed DSP into the hands of every engineer is at hand. The tools and methods permit novice and expert engineers to achieve performance that microprocessor style DSPs envy.

## Building Blocks

Figure 15-1 shows common building blocks, derived from adders and multipliers. In each case, operands defined as A,B or C are manipulated to produce a result, as the diagram shows. Much of the 1960s and 1970s were concerned with developing effective DSP without multipliers and so it was with Xilinx FPGA devices, early on.

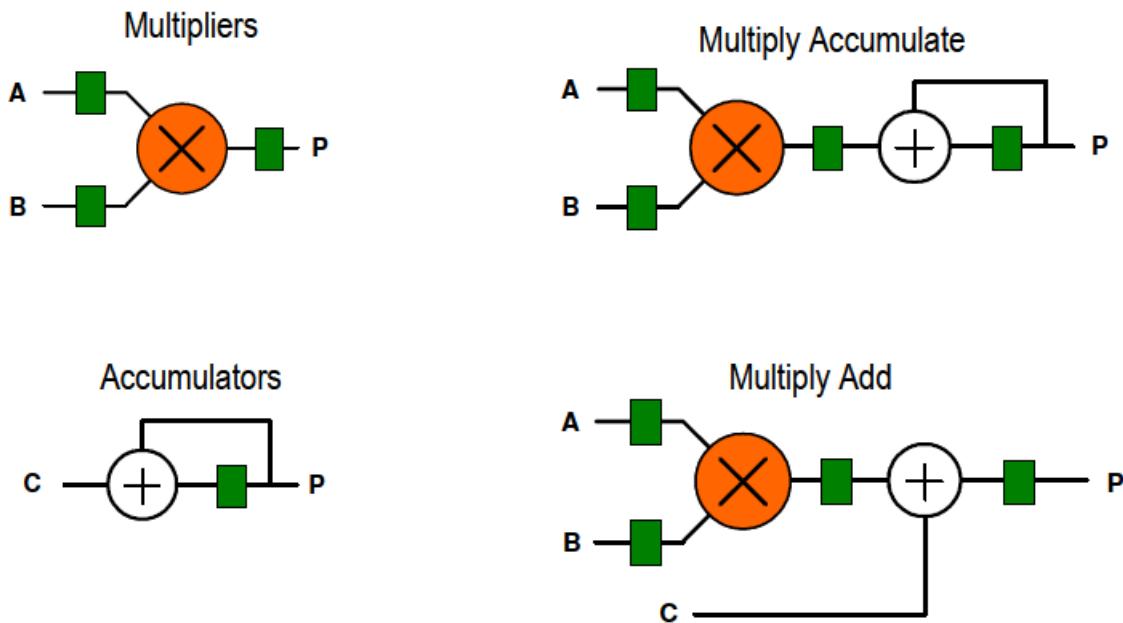


Figure 15-1 Common DSP Building Blocks

Other methods included extensive pre-calculating of fixed coefficients, or partial products that could be loaded into LUTs, to minimize calculation delays, wherever possible. While those methods are still available, most designers prefer to use the available calculation resources for the above four operations. Still, loading BRAMs with pre-calculated coefficients for Fast Fourier Transforms (FFT) or Discrete Cosine Transforms (DCT) are still common and effective. This is also true of bottlenecking calculations, where a BRAM can hold pre-computed values that would take a long time, if done by brute force hardware. Off line pre-computing in advance is a standard solution to these type of problems.

## Basic DSP Operations

### FIR Filters

Let's look at a few of the types of standard DSP functions that many users need, every day. Equation 15-1 shows the standard expression for a Finite Impulse Response (FIR) Filter.

$$y(k) = \sum_{n=0}^{N-1} a(n)x(k-n) \quad k = 0, 1, \dots$$

Equation 15-1

The  $x(k)$  are the input samples,  $y(k)$  are the outputs,  $N$  is the number of filter coefficients and  $a(n)$  are the filter coefficients. Pictorially, this can be shown in a number of different ways, where a common one is shown in Figure 15-2.

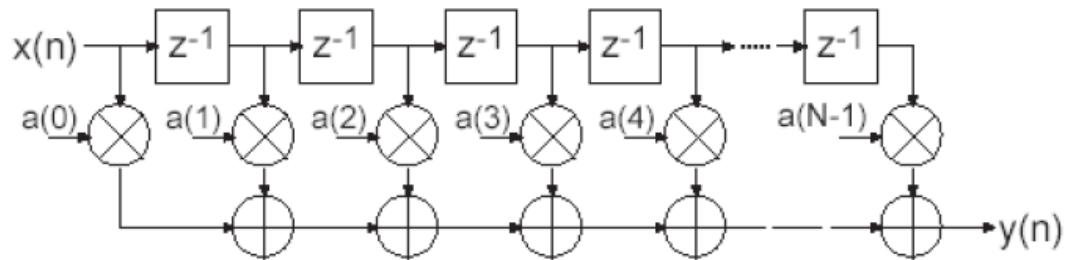


Figure 15-2 Tapped Delay Line FIR Filter

Although we assume the reader is familiar with DSP, let's simply reassure that the circles with X's are multipliers, the circles with +'s are adders and the boxes with Z-1 are delay elements, in classic z-transform fashion. Samples enter on the left, ripple through the various multiply/add stages and ultimately present correct data to the output site,  $y(n)$ . This represents what DSP engineers call an N-tap FIR filter. The general structure relates the taps along the delay line to poles and zeroes of the transfer function, on the unit circle. Figure 15-2 constitutes one form of a signal flow diagram, identifying the intermediate calculations along the way. In some Virtex realizations, the arrangement of the hardware aligns well with the signal flow diagram, which can be convenient. Using the method mentioned in the Arithmetic Chapter on distributed arithmetic, Equation 15-1 can be implemented without using multipliers. This involves pre-computing multiple partial sums of the coefficients and storing them into a RAM. That way, multiplications do not bottleneck the results, which may find new bottlenecks either in the connection routing, or the carry chain speed forming various additions. Appendix 15-1 summarizes basic distributed arithmetic for simple FIR filters. Figure 15-3 shows one means of performing this sort of distributed calculation.

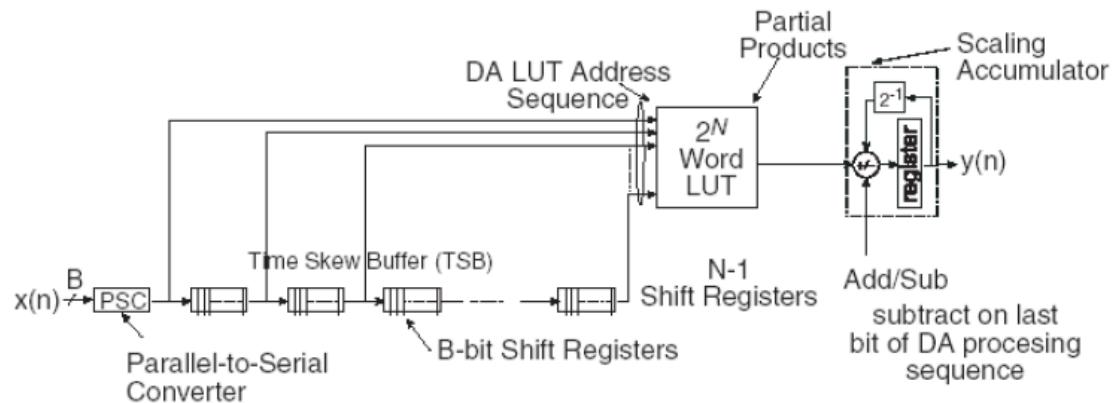


Figure 15-3 Serial Distributed Arithmetic FIR Filter

Note that the data becomes serialized along the way, from the initial parallel to serial converter, then subsequently passes through successive shift registers. The shift registers conveniently target SRL16 LUTs. The shift sites then form addresses into the  $2N$  Word LUT RAM, subsequently feeding into a scaling accumulator. This type of filter garnered great attention, when the Virtex parts were first launched, as they exceeded standard DSP performance from Texas Instruments, Motorola and Analog Devices. Figure 15-4 provides a feeling for the sort of speed tradeoffs that can be made by modifying the number of bits in the distributed arithmetic approach compared

to that of a standard multiply accumulate (MAC) approach. The left hand chart compares to one MAC and the right hand chart compares to two MACs.

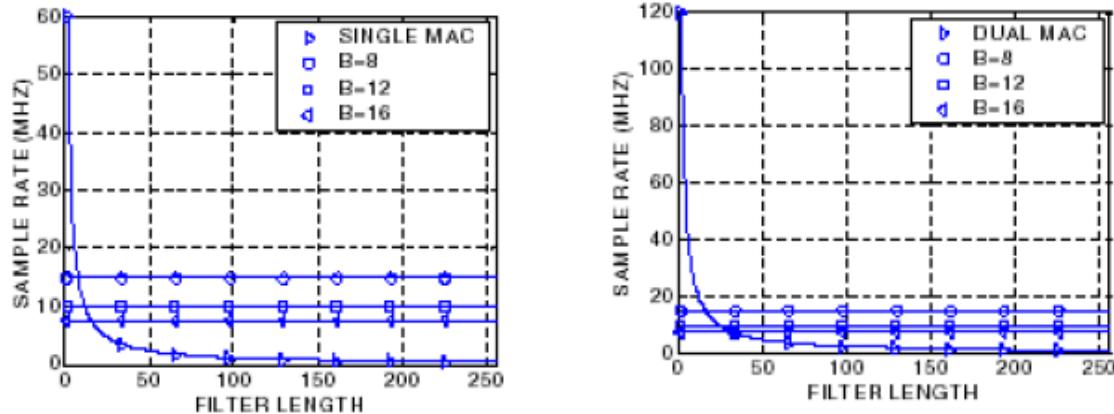


Figure 15-4 Sample Rate for Single and Dual MAC FIR Versus DA FIR at 120 MHz for Various Precisions

Using standard approaches from signal theory, many filters have symmetric frequency representation, permitting circuitry reduction by exploiting that symmetry. Figure 15-5 shows a nine tap FIR filter realization exploiting Odd symmetry. The advantage of these structures is the elimination of multipliers, which occurs by virtue of combining the mirror image frequency components suitably arranged to combine samples, then multiply the sum once, versus individually multiplying them at the expense of more multiplications.

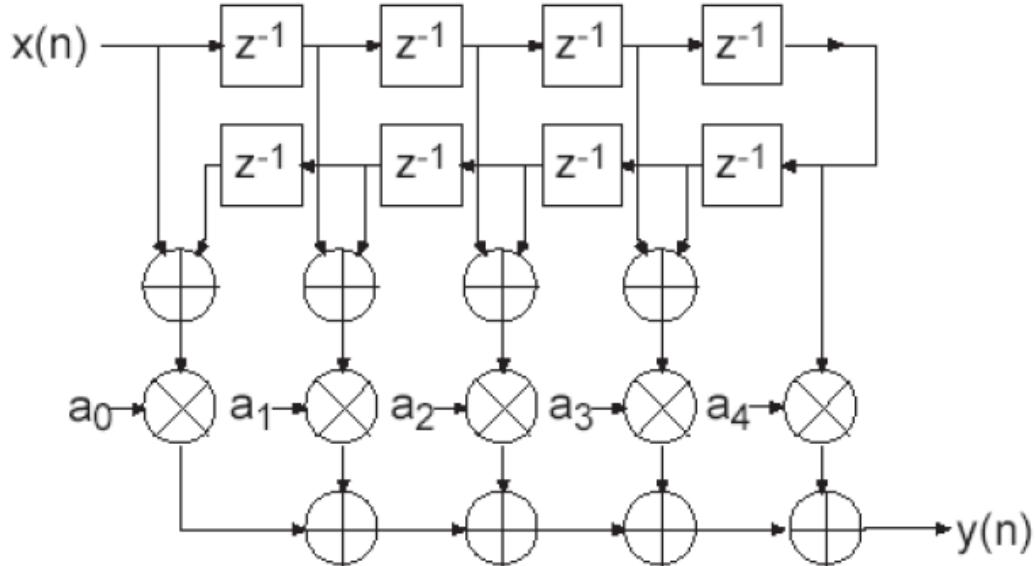


Figure 15-5 Exploiting Coefficient Symmetry for Odd Number Filter Taps

Clearly, low pass, band pass and high pass filters can be developed. The classic trick of re-circulating the filtered results back through the filter multiple times, to create sharper response also works, if time is available.

## Cascaded Integrator Comb Filters (CIC) (Hogenauer)

CIC filters permit up and down conversion of digital data. Figure 15-6 shows a decimation filter (down converter) and Figure 15-7 shows an interpolation filter (up converter). Both structures can be easily pipelined, and are multiplier-less. Note that the filter is built primarily from cascades of adders and registers. Elimination of multipliers plays well, for high speed.

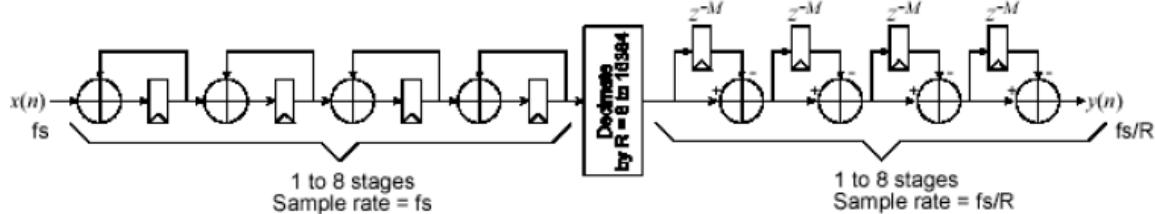


Figure 15-6 CIC Decimation Filter

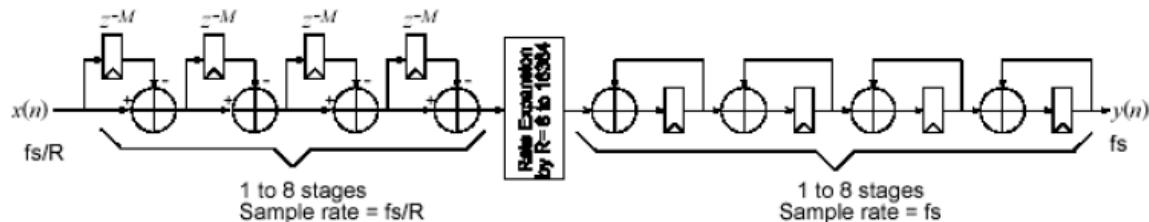


Figure 15-7 CIC Interpolation Filter

Note the reversal of the signal directions and placement of registers between adder stages. This is reminiscent of Fibonacci and Galois LFSR structures discussed earlier, but with real adders. CIC filters are basically low pass structures, and can introduce significant aliasing into the results, so this is a factor to be dealt with in some applications. A very nice book by Frederick J. Harris on "Multirate Signal Processing", explores CIC structures in depth. Xilinx offers a flexible LogiCore CIC Filter, which may be downloaded and used with the Core Generator system tool.

## Transforms

Discrete Fourier Transforms (DFT), Fast Fourier Transforms (FFT) and Discrete Cosine Transforms (DCT) are also possible, and are offered as Xilinx LogiCores. Figure 15-8 shows a straightforward implementation of a one dimensional DCT, which uses a set of FIR filters to develop the DCT coefficients.

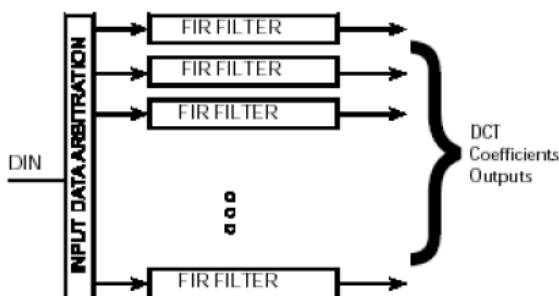


Figure 15-8 1-D Discrete Cosine Transform Engine

In this situation, Equation 2 describes what is being implemented:

$$X(k) = \alpha(k) \sum_{n=0}^{N-1} x(n) \cos\left[\frac{\pi(2n+1)k}{2N}\right] \quad 0 \leq k \leq N-1$$

Where

$$\alpha(0) = \sqrt{\frac{1}{N}}, \quad \alpha(k) = \sqrt{\frac{2}{N}} \text{ for } 1 \leq k \leq N-1$$

Equation 16-2

As you might guess, there is also a reverse operation – the inverse DCT, which recovers the original data, from the coefficient set calculated above. Depending on the structure chosen, and the number of bits calculated per clock, fabric can be saved. Table 15-1 gives an idea of this tradeoff.

Clocks per Sample	Sample Rate	Silicon Area (# slices)
6	$f_{clk}$	982
9	$(f_{clk}/9)x8$	767
17	$(f_{clk}/17)x8$	589

Table 15-1 Sample Rate (Throughput) versus Fabric Utilization Tradeoff

Discrete Cosine Transforms are important because they are a key requirement for data compression schemes like MPEG-4, for video. Compressing multidimensional data can have huge impact on bandwidth for applications like local area networks, or cellphone base stations, which must handle massive data loads.

### Wavelet Transforms

Wavelets have been popular recently. Partially, this is because they lend themselves very well to real world signal characteristics, which are not continuous and stationary, as required by standard Fourier analysis. Their history goes back at least to Dennis Gabor in the 1940's. We won't go into much wavelet theory here, but suffice it to say that one particular discrete wavelet transform developed by Stephane Mallat has been quite popular. It defines two filter functions  $g(k)$  and  $h(k)$ . The outputs of the filter functions are combined successively in a pyramided structure spanning multiple stages, where each pyramid level produces the coefficients of the wavelet transform. JPEG 2000 compression is based on wavelet transforms, so it may also be applicable to cell-phone applications, where pictures suitably compressed will save substantially on bandwidth.

## Data Communication Encoding/Decoding

Error coding and decoding is often associated with DSP, mostly because of the numerical capabilities of DSP microprocessors. Inherently integer calculations, error coding/decoding are probably best done with straight logic, sometime configured to perform finite field calculations as in Reed-Solomon encoding/decoding. Viterbi and Turbo codes can require intense hardware solutions, to operate at full speed. One way to obtain encoding/decoding solutions is to simply buy them. Xilinx offers intellectual property (IP) with a special “sign once” licensing arrangement, and Table 15-2 illustrates the flexibility of its Reed-Solomon decoder LogiCore.

	ATSC 1	ATSC 2	DVB1	DVB2	CCSDS	G.709	G.709 2-channel	ETSI- BRAN	IEEE- 802.16d
Generator Start	0	0	0	0	112	0	0	0	0
h	1	1	1	1	11	1	1	1	1
k	187	187	188	188	223	239	239	239	239
n	207	207	204	204	255	255	255	255	255
Polynomial	285	285	285	285	391	285	285	285	285
Symbol Width	8	8	8	8	8	8	8	8	8
Erasure Decoding	No	No	No	Yes	No	No	No	No	No
Optimization	Area	Speed	Speed	Speed	Speed	Speed	Speed	Speed	Speed
Clock Periods Per Symbol	1	1	1	2 <small>(note 6)</small>	1	1	1	1	1
Variable Block Length	No	No	No	No	No	No	No	Yes	Yes
Number of Channels	1	1	1	1	1	1	2	1	1
Puncture Patterns	0	0	0	0	0	0	0	0	4
Processing Delay <sup>1</sup>	294	294	204	179	660	204	406	204	357
Latency <sup>1</sup>	508	508	415	380	925	486	926	Variable	Variable
Xilinx Part	XC2VP2-6	XC2VP2-6	XC2VP2-6	XC2VP4-6	XC2VP2-6	XC2VP2-6	XC2VP2-6	XC2VP2-6	XC2VP2-6
Area (Slices) <sup>3</sup>	758	764 <sup>5</sup>	633	1455	1110	557	870 <sup>7</sup>	681	877
Block Memories	2	2	2	2	3	2	2	3	4
Maximum Clock Frequency <sup>2,3</sup>	121MHz	129MHz	135MHz	119MHz	124MHz	132MHz	204MHz	137MHz	130MHz

**Notes:**

1. Measured in symbol periods.
2. Higher frequencies may be attainable by setting the packfactor option on the mapper to -c 100 rather than -c 1.
3. Area and max clock frequencies are provided as a guide. They may vary with new releases of the Xilinx implementation tools, etc.
4. All results use the create\_RPM option which switches on the use of RLOCs within the design.
5. Note that the “speed” option generally does not have a large effect on slices used. This depends on the code parameters and the device selected.
6. Set to 2 to keep Processing Delay <= n.
7. Many of these slices contain only registers and the LUTs will still be available for user logic.

Table 16-2 Reed-Solomon Decoder Implementation Details

Note that the LogiCore solution supports several data communication standards, offers some degree of flexibility- puncture options, erasure decoding, and multiple channel support. The lower table entries also show variation in required logic, as well as speed.

## Development Flows

An early goal for DSP was to provide a wide spectrum of tools capable of solving most DSP problems, for a potentially large customer base. Naturally, this includes hardware savvy system engineers, but it was also recognized that many DSP engineers are borderline mathematical theoreticians. This presented a bit of a problem, in that FPGA design tends to be more intimate with real world logic delays and resource usage than many of those designers wished to be. The solution, was to engage with these engineers and discover the tools they would prefer and adapt our FPGA design flows to their needs. The basic progression of tool development to date has been first, HDL, then introducing Core Generator and more recently System Generator. The latter tool operates in the MatLab/Simulink environment used by many DSP algorithm developers, and has been a very successful partnership between Xilinx and The Math Works. Let's discuss these basic flows and see where they lead.

### HDL (RTL) Flows

Verilog and VHDL have been the primary hardware definition languages supported by Xilinx toolset. They present very transportable, abstract formats for describing the functionality of a design. This is powerful, but can also present problems. It is possible to write a design description that is open to interpretation, resulting in less than desirable results. That being the case, a general rule has evolved to define designs as close to the real world fabric blocks as possible, to remove ambiguity and get recognizable, controllable resulting netlists. That flow is best described as the register transfer level (RTL) flow. RTL permits greater user level of control, at the expense of complete abstraction. This may or may not be good. If you don't want to know what is happening with your design (fully abstract), then you will have to be content with what happens to it in the compile process. Otherwise, it is advisable to learn the RTL versions of VHDL and Verilog.

One key idea for high level description of a design is to use pipelining. This has the advantage of building fast, predictable designs, running at high clock rates. The downside is that the results may require many clock cycles before the first result is available at the output (i.e. latency). Simplistically, pipelining a design is simply identifying logic sites that may be registered (i.e. break the logic, attach to a D flip flop, and reattach the flip flops Q output on the other side of the break). This lets the PAR software identify small logic “chunks”, interleave registers, and repeat. It fits very nicely onto Virtex architectures. Being able to write your RTL HDL to do this simplifies DSP design much.

### Vivado System Generator for DSP

System Generator is a DSP design tool from Xilinx that enables the use of the MathWorks model-based Simulink design environment for FPGA design. Previous experience with Xilinx FPGA devices or RTL design methodologies are not required when using System Generator. Designs are captured in the DSP friendly Simulink modeling environment using a Xilinx specific blockset. The System Generator design

can then be imported into a Vivado IDE project using the IP Catalog. System Generator lets designers create signal flow/block diagrams in the standard Simulink environment, where they can pick from a set of available blocks, manage data types, and interconnect them to produce digital systems. Once connected, the systems can be simulated to determine correctness, then forwarded for synthesis to create an FPGA bitstream. If designers need blocks beyond the abundant Xilinx blocks, they can create and import them. In addition to the Xilinx blocks, there are also many reference designs that can be imported into System Generator designs, with details given on the Xilinx website.

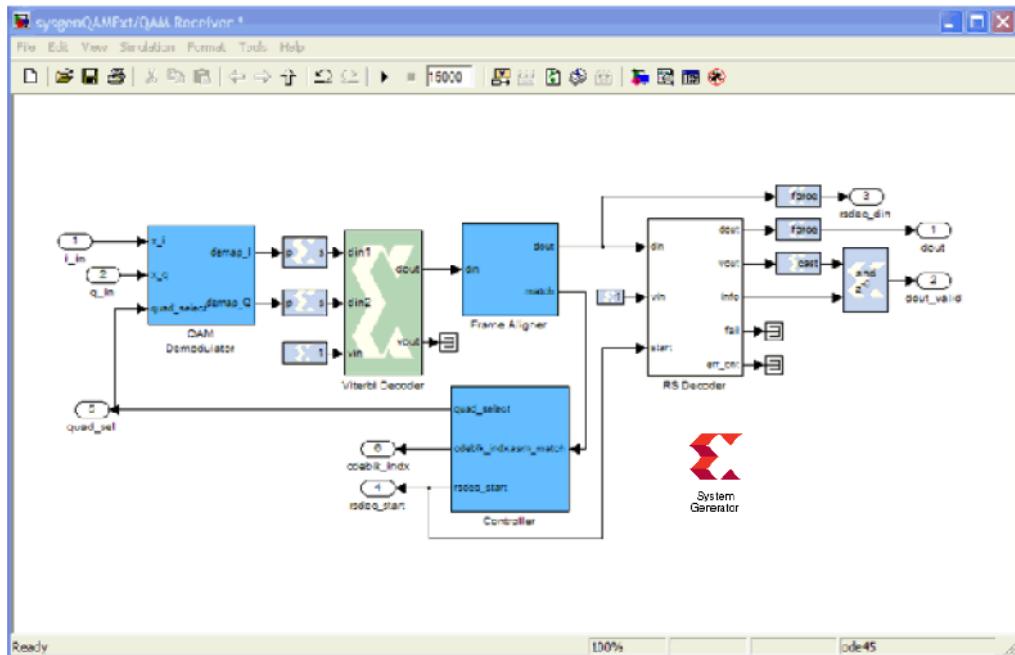


Figure 15-9 System Generator

#### Xilinx DSP Block Set

Over 90 DSP building blocks are provided in the Xilinx DSP blockset for Simulink. These blocks include the common DSP building blocks such as adders, multipliers and registers. Also included are a set of complex DSP building blocks such as forward error correction blocks, FFTs, filters and memories. These blocks leverage the Xilinx IP core generators to deliver optimized results for the selected device.

#### FIR Filter Generation

System Generator includes a FIR Compiler block that targets the dedicated DSP48E1 and DSP48E2 hardware resources in the 7 series and UltraScale devices to create highly optimized implementations. Configuration options allow generation of single rate, interpolation, decimation, Hilbert, and interpolated implementations. Standard MATLAB functions such as fir2 or the MathWorks FDATool can be used to create coefficients for the Xilinx FIR Compiler.

## Support for MATLAB

Included in System Generator is an MCode block that allows the use of non-algorithmic MATLAB for the modeling and implementation of simple control operations.

## Hardware Co-Simulation

System Generator provides accelerated simulation through hardware co-simulation. System Generator will automatically create a hardware simulation token for a design captured in the Xilinx DSP blockset that will run on supported hardware platforms. This hardware will co-simulate with the rest of the Simulink system to provide up to a 1000x simulation performance increase.

## System Integration Platform

System Generator provides a system integration platform for the design of DSP FPGA devices that allows the RTL, Simulink, MATLAB and C/C++ components of a DSP system to come together in a single simulation and implementation environment. System Generator supports a black box block that allows RTL to be imported into Simulink and co-simulated with either ModelSim or Xilinx Vivado simulator, and provides a Vivado HLS block that allows integration and simulation of C/C++ sources.

## **Did Someone Mention C/C++?**

With Vivado HLS, one is able to convert C/C++ functions into RTL designs which may then be imported into your design. The advantages are many, both in time to develop, but also in time to functionally verify, and time to test. Through the use of pragmas (comments in the code) one directs the conversion to pipeline, expand and parallelize inner loops, etc.

## References

1. DS240 Xilinx LogiCore Distributed Arithmetic FIR Filter V9.0
2. Xilinx LogiCore Cascaded Integrator-Comb (CIC) Filter V3.0
3. DS252 Xilinx LogiCore Reed-Solomon Decoder V5.1
4. Core Generator User Guide V2.1
5. System Generator User Guide V7.1 (online at Xilinx website)
6. Building Custom FIR Filters Using System Generator (J. Hwang, J. Ballagh)
7. WP 212 DSP Co-Processing in FPGAs: Embedding High-Performance, Low-Cost DSP Functions, (S. Zack)
8. Xtreme DSP User Guide (ug073)
9. Multirate Signal Processing for Communication Systems, PTR Prentice Hall, Frederick J. Harris
10. Applications of Distributed Arithmetic to Digital Signal Processing, S.A. White, IEEE ASSP Magazine, Vol. 8, No. 9, September, 1989
11. The Role of Distributed Arithmetic in FPGA-based Signal Processing, Les Mintzer (available on the Xilinx website, search for "theory1.pdf")
12. FPGAs Make a Radar Signal Processor on a Chip a Reality, Ray Andraka
13. UltraScale Architecture-Based FPGAs Memory IP v1.2 LogiCORE IP Product Guide Vivado Design Suite PG150 June 8, 2016
14. <https://www.xilinx.com/products/design-tools/vivado/integration/sysgen.html>
15. Vivado Design Suite User Guide Model-Based DSP Design Using System Generator UG897 (v2016.2) June 8, 2016
16. <http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>

## **Chapter 17 Virtex: Under the Hood**

Xilinx hit the three thousand patent mark at approximately its thirtieth anniversary. As this second edition is published, Xilinx will be thirty three years new. Virtex 7 is in full production, and Virtex UltraScale, and UltraScale Plus is sampling. It makes sense to take our bearings and consider where we have come, and where we are heading.

### **Technology Directions**

Inside Xilinx, the Virtex families are all named after famous mountains. The earliest Virtex family was named after the largest mountain in North America (hint: think Alaska). Virtex 4 is named after the largest mountain in the lower 48 states. Virtex 5 was named after the largest peak in the Cascade Range. It's probably true that each family represented a new challenge, never previously undertaken by a programmable logic company.

Xilinx has held the market lead for quite a while, and maintains its position pushing the state of the art. With the 7 series at 28nm, Xilinx took a commanding lead over all of its competitors. With 16nm UltraScale Plus, Xilinx has had zero competition for over one year as our largest competitor was acquired by another company, and continues to delay its answer to the challenge.

Virtex brought on new challenges of embedding various function blocks – BRAMs, multipliers, MGTs, DSPs and processors as the CMOS "process mountain" got steeper and steeper. This is not an exaggeration. The original 250 nanometer Virtex family process was simple compared to the current 16 nanometer process. The number of process steps – implants, oxide layers, metal layers, etc. looks like a wedding cake, with layer upon layer stacked higher and higher. It will only be getting thicker, moving forward, and new process challenges appear with every family.

The cost of developing products skyrocketed over the last decade, with one chip mask set costing roughly a ten million dollars. This doesn't include the design, layout and process development manpower - just the masks. As you might guess, mistakes are very expensive. Figure 17-1 shows the original Virtex, 250 nanometer process diagram, and Figure 17-2 shows the Virtex 4, 90 nanometer process diagram. These pictures are taken with a scanning electron microscope, and sliced with focused ion beam technology. Figure 17-3 shows a Virtex 5, 65 nanometer cross section.

The "key firsts" of including copper interconnect metal (for speed) and low K dielectric (for speed), were both introduced with Virtex II Pro. As you know from the discussion of timing, in Chapter Eight, tiny decreases in nanoseconds result in large bandwidth increases in mega/gigahertz. All of these steps are expensive, but key Xilinx customers sought this added performance, and Xilinx responded to serve their needs.

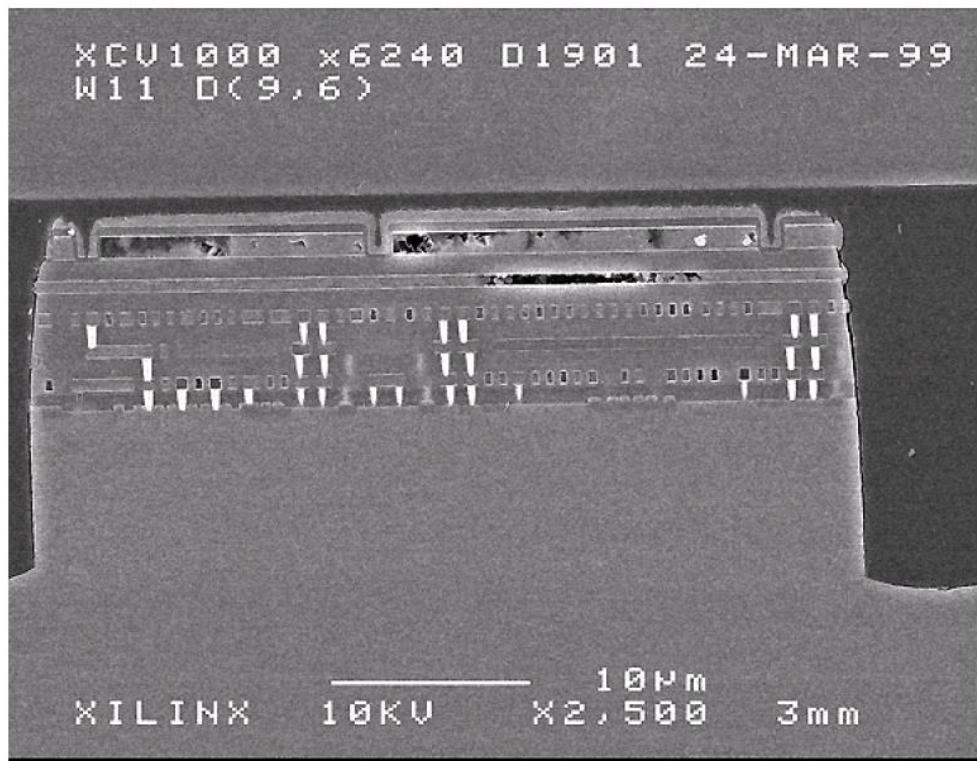


Figure 17-1 Virtex 250 Nanometer Cross Section

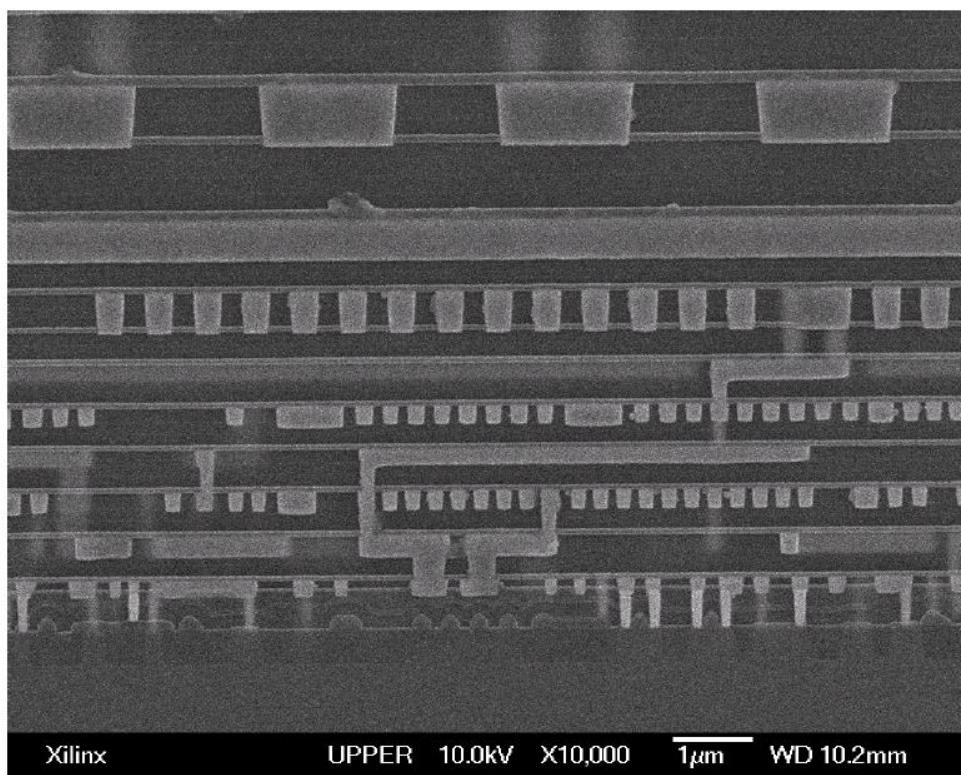


Figure 17-2 Virtex 4 90 Nanometer Cross Section

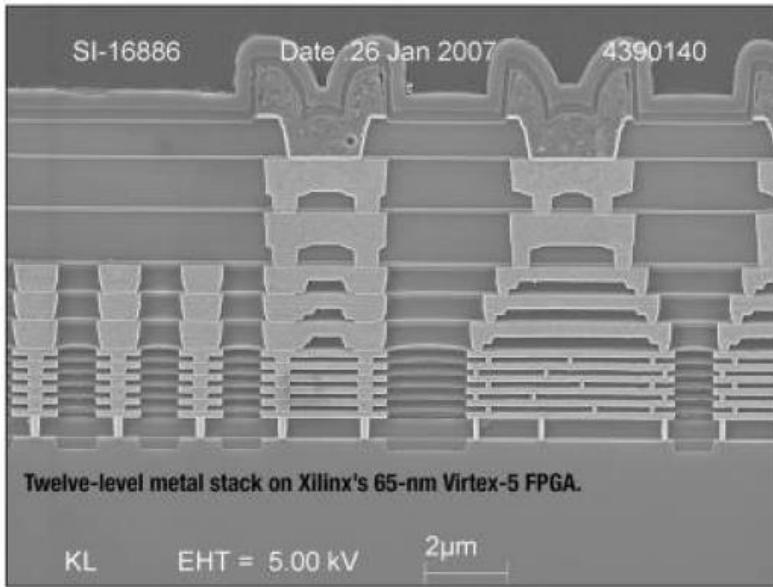


Figure 17-3 Virtex 5 65 Nanometer Process Cross Section

Process economics changed much over Xilinx' lifetime. In early days, it was easy to predict when volume manufacture would amortize the cost of new equipment to predict pending price reductions for these expensive processes. Recent processes violate those models. Equipment costs are rapidly increasing, and it takes more time (years) to provide cheaper, more cost effective solutions. Xilinx has learned much, finding new ways to reduce costs and make products more affordable than before.

Below 90 nanometers, CMOS transistor behavior changes much, too. No longer is the technology inherently low power. No longer is there a significant time when one transistor blocks current flow, while the other passes it. At 65 and 45 nanometers, all transistors are turned on, all of the time. Behavior is more like emitter coupled logic (ECL) which was notorious for both blazing speed and blazingly hot devices! Such is the price for innovation. It is time to find new ways to reduce power, and attain even greater speed. Nonetheless, Xilinx is rising to the challenge, by partnering with multiple industry leading process development partners.

### Dimensions of Excellence

As process features shrunk, other concerns and possibilities arose. For instance, power consumption became a concern below 130 nm, and extra precautions needed to be taken. One successful approach to current leakage management has been the use of multiple oxide layers in the process. Thinner oxides tend to "leak" more, but transistors built with thin oxides tend to be faster. To that end, Xilinx designers and process developers used thin oxides only where necessary - speed paths. Thicker oxides were used on I/O cells and in the configuration areas, where the pattern is stored, but not frequently moved.

Smaller features brought concern from aerospace and high reliability markets regarding "single event upset" from cosmic particles (see Chapter 5). Xilinx designers

and process engineers, working with outside test labs, have tested small feature Spartan and Virtex parts with neutron beams and high altitude test facilities to assure absolute minimum in cell disturbs. To that end, Virtex parts being in the famous Mars Rover are a testament to this work.

The area of reconfigurable logic - whereby the FPGA assumes different configurations at different points in its operation - is an on-going area of research that is interesting to academic institutions, commercial enterprises and the team at Xilinx Labs, whose charter includes exploiting this vast, untapped capability. Internally, Xilinx is organized with specific business units focusing on embedded systems and digital signal processing. Creating better user interfaces and more effective tools for designers to develop their algorithms swiftly and effectively is an on-going goal. As you might expect, the design software continues to evolve. New mappings appropriate for the Virtex 5 six LUT structure, and its impact on end user designs will be under investigation for some time to come.

## **Conclusions**

At this point, we draw the text to a close, but you should be aware that there is more to come. As future Virtex parts roll out, this text may be revised and updated adding in new features and explaining more application areas where these parts can be used. We won't tell you the name of the next product.

## Virtex Glossary

**Accuracy** is the magnitude of the maximum difference between a real value and its numerical representation. Accuracy is a major concern for DSP algorithms.

**Active Interconnect** is Xilinx' fourth generation segmented routing technology, providing fully buffered routing, at both ends of connection segments. Active Interconnect substantially homogenizes path delays for more uniform connections and predictable performance.

**AES** Acronym for the NIST Advanced Encryption Standard. AES was developed with the specific intention of replacing its predecessor standards, DES and Triple DES.

**AGP** Advanced Graphics Port is an industry standard graphics interfacing protocol. AGP electrical parameters resemble PCI in both timing and drive capability.

**Aliasing** Under sampling discrete signals in DSP results in incorrect data that may not be faithfully reconstructed. This condition is called aliasing. Under sampling disregards the Nyquist sampling requirement.

**Anti-aliasing** refers to pre-filtering data prior to sampling, so that its bandwidth is at most half of the sampling frequency.

**APR** Automatic Place and Route describes both the general software flow in many ASIC environments as well as an early version of the main software in the Xilinx XACT product. Note: also called AP&R.

**APU** Auxiliary Processor Unit is a block of circuitry attached to the PowerPC405 processor core to facilitate the integration and high performance operation of custom co-processors and hardware accelerator functions.

**ASIC** Application Specific Integrated Circuit. Typically either a gate array or standard cell based integrated circuit, but many varieties exist. Frequently, ASIC is used interchangeably with gate array, standard cell, or simply "semi-custom" integrated circuit.

**ASMBL** Advanced Silicon Modular BLock architecture is the columnar block and fabric architecture used since Virtex 4. Placing I/O pins uniformly across the package along the columns results in unprecedented signal integrity in flip chip packages, for the high speed requirements these families target.

**Assignment problem** The assignment problem is a classic system theory problem of making the best task assignment to a set of resources. The traveling salesman problem is the mathematical dual of the assignment problem. It is sometimes called "weighted bipartite matching."

**ASSP** is an acronym for Application Specific Standard Product. This is any number of readily available, completed designs originally created using ASIC technology. The vendor is a third party that undertakes developing an ASIC, to do a marketable task, and sells the result as a finished product.

**Asynchronous** means literally “without a clock”. This is a catchall term that includes state machines using multiple clocking sources, state machines without any clock (state transition responding to input and/or state changes) and more. Note that a state machine that has only one clock input, but uses both edges of the clock is also asynchronous, because it includes two clocking events, unless every flip flop switches on both edges, which is the same thing as doubling the clock frequency, which would be synchronous.

**ATG Automatic Test Generation** covers a whole spectrum of methods of creating test vectors for designs, with the goal of creating tests that can distinguish a good device from a bad one. Ideally, it also targets making the distinction in a minimum set of applied input stimuli (vectors).

**AXI Bus** A standard developed by ARM to interconnect their processors and other blocks. Licensed by Xilinx and used as a standard interface on both soft (programmable logic blocks) and hardened (ASIC) blocks.

**Back annotate** Back annotation is basically placing more information back into something which had less. In most FPGA situations, this refers to placing time delay values into the design netlist, after PAR has completed its task of placement and routing. The netlist at that time reflects real world delays that are useful for timing simulation and static timing analysis. Occasionally, back annotate is used to describe specific pin assignment back into the high level design, if the software dictates pin-out.

**Backplane** The medium used to interconnect a number of circuit boards. Typically refers to a special, heavy-duty printed or discrete wired circuit board.

**Bandwidth** is the maximum data rate that an application can process or transfer over an interface or the spectrum occupied by the interface. Bandwidth is often measured in bits per second (rate) or Hertz (interface).

**BER** is an acronym for Bit Error Rate. BER is a figure of merit used in communication to describe channel quality. BER measures the number of errors detected at a receiver in a given length of time, sometimes specified as a percentage of received bits; sometimes specified in exponential form (i.e. 10E-8 indicates 1 bit error in  $10^8$  bits.)

**BERT** Bit Error Rate Tester is high speed equipment for measuring BER

**Big-endian** is the representation of a multi-byte value that has the most significant byte of any data field stored at the lowest memory address, which is also the address of the larger field. A pun as we have both big and little ones.

*Bin packing* is simply assigning connections consecutively once their terminal block is determined. It is an aspect of routing.

*BIST* Built In Self test is a broad category of circuits that can be included in a design, operating automatically to determine the electrical condition of a design. An early method of doing BIST was called signature analysis.

*Bit and Cycle-true Modeling* System Generator produces simulation results that are bit-true and cycle-true to the hardware it generates. To say a simulation is bit-true means that at the boundaries (i.e. interfaces between System Generator blocks and non-System Generator blocks), a value produced in simulation is bit-for-bit identical to the corresponding value produced in hardware. To say a simulation is cycle-true means that at the boundaries, corresponding values are produced at corresponding times. The boundaries of the design are the points at which System Generator gateway blocks exist. When the tool generates hardware, Gateway In blocks become top-level input ports where Gateway Out blocks become top-level output ports.

*Bitslip* is the capability to align bits within a word by individually introducing Idelay delays to each input channel. It involves a state machine, the Idelay facility and a training pattern to identify word synchronization.

*BGA* Ball Grid Array is a packaging technology using tiny solder balls that attach to a printed circuit board. It is a surface mount technology.

*Boundary Scan* This is a test technology which involves adding circuitry to the periphery (boundary) of the chip, right at the pins. The most common type of boundary scan circuitry is IEEE 1149.1 or “JTAG” Boundary Scan. There is usually a boundary scan controller that responds to external testing commands, and a bidirectional capture register capable of sampling/driving data right at the device pins. Boundary scan circuitry is an effort to systematically improve the testability of an integrated circuit.

*BPSK* BiPhase Shift Keying is a digital frequency modulation technique for sending data over a coaxial cable network.

*BRAM* Acronym for Block Random Access Memory.

*Bridge* An interface to facilitate communication between devices using communication protocols. Bridges can occur between buses, where a PCI bridge expands the slot capacity of a PCI bus. Bridges can connect buses or data channels of different protocols, also. Often an application of Virtex FPGA devices using RocketIO high-speed serial transceivers and/or parallel I/Os.

*BSP* Board Support Package is a collection of software and hardware resources designed to facilitate development of applications for microprocessors.

*BUFG* is an FPGA global buffer characterized by high speed and high drive. BUFGs typically propagate fast control signals, such as clocks.

*Cache* is high speed memory located electrically “near” to a processor. Typically, processors operate directly out of level one (L1) cache, and access a slower more “distant” level two (L2) cache when correct data isn’t present in the L1 cache. L1 is usually located within the processor block and L2 outside. Additional levels may be used, and all are coordinated by a “cache controller”.

*CAM* Content Addressable Memory is a specialized type of memory incorporating high speed comparison into the basic fabric. CAM is usually a key ingredient for associative memories that excel at high speed search.

*Capacity* is a measure of the total capabilities of an FPGA. This includes the logic fabric, the I/O’s, all block resources (BRAM, ALU, MGT, etc.) See also, *utilization*.

*Carry-chain* is a Xilinx FPGA architectural feature for efficient, high speed addition. Xilinx has many patents on carry chains. See also, “*fast carry*”.

*Carry Look Ahead* is a technique possible to produce fast results with the Virtex slices that include special circuits to form classic “generate” and “propagate” intermediate results. These combine to increase the speed of additions, which thereby speedup subtractions and counters, as well.

*Cascaded integrator-comb (C/C)* filters are multi-rate filters used for realizing large sample rate changes in digital systems. Both decimation and interpolation structures are supported. CIC filters have no multipliers, consisting of only adders, subtractors and registers. They are typically used in applications having a sample rate faster than the signal bandwidth.

*CBC* Cipher Block Chaining is a cryptographic technique, where successive blocks of encrypted data are EX-ORed with a previous block, to add further confusion to the encryption process.

*CDR Clock & Data Recovery* is the ability to extract a clock from specially encoded data at the receiving end. 8b10b encoding is an example of an encoding format that assures the correct run of 1’s and 0’s occurs in the transmitted stream so that the receiver can create a clock from the data stream.

*Channel bonding* Channel bonding is the set of actions taken with multiple data streams that can arrive skewed from each other, whereby a receiver can correctly identify bits of the same data token, although they may actually reside in different input clock cycles.

*Charge pumps* are internal circuitry designed to produce a higher internal voltage than is supplied by external pins. Charge pumps employ high speed, free running oscillators, and switched capacitor banks to produce the voltage. Higher voltage applied to transistor gates (as on PIPs) can eliminate the threshold drop encountered from source to drain.

*Checksum* is a method of assuring integrity of a binary file, by simply adding all of the bits within the file, and producing a unique sum that reflects the number of bits stored, or transmitted. It is often chosen because it is easy and fast, and simple compare to a separately transmitted sum reveals whether or not the file transferred or stored properly. It is not robust enough to recover from an error, which must be resolved by other means – typically retransmission. Sometimes, the addition overflows the size of a chosen word for the sum, so it is possible to “alias” the sum, but is a rare occurrence.

*ChipScope PRO* ChipScope PRO is the updated version of ChipScope ILA to include access to embedded PowerPC resources. ChipScope ILA (Integrated Logic Analyzer) is a set of core logic functions that create circuitry to embed a logic analyzer within a Virtex part.

*ChipSync* ChipSync is the set of I/O features in later Virtex FPGA devices that deal with signal timing. ISERDES, OSERDES, Idelay and Bitslip are all components of ChipSync.

*CLB* Configurable Logic Block is one level of configurable logic in an FPGA device. However, CLB contents have varied over FPGA history with regard to how much logic is in the block. Early FPGA CLBs consisted of one, two or three LUTs with flip flops. Current Virtex family CLBs consist of multiple slices, grouped into a CLB. The slice functionality today is closer to the CLB functionality of the past. See also “*slice*”.

*CLE* Configurable Logic Element is a common Xilinx design term for a LUT and flip flop. CLE is most often used when referring to the CLB circuit layout. See also *Logic Cell*.

*Clock skew* is the accumulating delay of clock signals as they are routed across distance. The same effect, to different degrees, is seen on integrated circuits, printed circuit boards and systems.

*CML* Current Mode Logic, one of several different switching technologies that operate with current versus voltage. Emitter Coupled Logic (ECL) and Positive ECL as well as LVDS are examples of CML.

*CMOS* is a strange abbreviation for Complementary Symmetry Metal Oxide Semiconductor technology. Logic gates built from CMOS use but two types of transistors, and the technology scales well with shrinking processes. Small size, high speed, low cost and low power are common attributes.

*CMRR* Common Mode Rejection Ratio is a quality of differential switching signals. This is the ability to subtract out noise signal which may exist on both the positive asserted and negative asserted components of the signal. It is typically specified in decibels (dB).

*CMT* Clock Management Tile. Contains the PLL and other clocking functions in later Virtex devices.

*Commas* are special separator symbols in a data stream. 8b10b encoding supports several comma symbols.

*Common Mode* The aspect of differential signaling that is not complementary. See also *CMRR*.

*Configuration* is the act of loading the various LUT and PIP contents into the SRAM based configuration space. Multiple configuration modes exist, such as slave serial, master parallel and JTAG.

*Configuration Memory/Space* is the set of SRAM bits that define the Virtex functionality. It is an array of SRAM cells that contain the LUT and PIP contents.

*Congestion* A highly crowded routing condition is congested. Congested routing is intolerant to modification due to limited available choices.

*Constraint* is any of a number of design restrictions encountered while designing with programmable logic. Constraints occur in many dimensions. For instance, restricting the design to a specific number of pins is a constraint. Specifically assigning signals to particular pins is another set of constraints. Requiring chosen clock frequencies or dictated time delays are timing constraints. Area and power constraints also exist. Constraints are what makes writing FPGA Place and Route software so much fun. Global constraints like “period” are usually easiest to use and give high payoff.

*Constraint File* is specifically designed to communicate user requirements to the FPGA design software. It is a common repository for users to isolate their special needs, and a place for the design software to find them.

*Context* has many meanings, but the primary one used with Virtex FPGA devices is with regard to microprocessors. The context of a processor is the binary state of all registers and function units at a point in time. This includes the state of the control unit, as well as the data units and register set.

*Context switch* is the action of modifying the processor context from one set of values to another, at a point in time. A context switch occurs when a subroutine is entered, and returning to original context occurs upon exit from the subroutine. In a multiprocessor environment, the processor context switches when it leaves one processing domain and enters another. Switching between applications, or from an application to an operating system state would be other examples. Any of these usually involve saving the context (register, state values, flags, etc.) to external memory, for switching back in the future.

*Cores* are ready made functions optimized for Xilinx FPGA devices. Cores range in complexity from simple arithmetic operators such as adders, accumulators and multipliers to system level building blocks such as filters, transforms, FIFOs and memories.

*CoreConnect* is an interconnection internal bus developed by IBM. It eases integration and reuse of the processor, system and peripheral blocks. Elements of CoreConnect architecture include the processor local bus (PLB), the on-chip peripheral bus (OPB), a bus bridge and a device control register bus. Later replaced by AXI bus.

*Core Generator* is Xilinx design software that provides a set of procedural solutions that users may elaborate as they need. Core Generator includes several DSP filter functions and many mathematical operations.

*Co-simulation* is a misnomer for combining simulated operation with simply operation. Large blocks (Processors, DSP48, MGT) are complex to model in a simulator. By using actual hardware to “model” their behavior in a simulation environment that resolves lower design elements into netlist and data structures, much simulation time can be saved and reliable results gained. The hardware blocks are actually “operating”, versus simulating hence, the misnomer. Co-simulation is not trivial. It requires complex software and coordination between modeled circuits and real circuits to maintain time and function integrity, but the speed payoff is worth the effort.

*Critical path* is a signal in a section of combinatorial logic that limits the speed of the logic. Storage elements begin and end a critical path, which may include I/O logic.

*Crosstalk* is a type of noise whereby the simple switching presence of one signal disrupts the clean behavior of another. The disrupting signal is termed the “aggressor” and the disrupted signal is the “victim”.

*CRC Cyclic Redundancy Checksum* is an error detection calculation. CRC is used in a number of ways in data communication. FPGA devices specifically use CRC to check that configured bitstreams are error free, prior to asserting the done signal (a semaphore).

*CSP Chip Scale Packaging* is a small form factor, surface mounted, ball grid technology whereby the package perimeter slightly exceeds the die perimeter.

*Cycle to Cycle Jitter* is the worst-case difference in clock period between adjacent clock cycles in the collection of clock periods sampled. In a histogram of cycle-cycle jitter, the mean value is zero.

*DA Distributed Arithmetic* describes a number of methods whereby calculation occurs incrementally over several stages. Typically, it involves recognizing a key factorization that may eliminate bottlenecking steps like reducing multiplications and additions. The standard paper on this topic was by Stanley White, and is referenced in Chapter 15.

*DAC Digital to Analog Converter.*

*DAG (see Directed Acyclic Graph).*

*DC Balanced* in data communications refers to encoding schemes that avoid charging up a transmission line by recoding data so that long strings of ones or long strings of zeroes never occur. Long consecutive strings tend to charge a data line, which may affect its future ability to respond to switching. 8b/10b and 64b/66b encoding are two attempts at DC balancing.

*DC Coupling* describes interconnection between a transmitter and receiver without series capacitance. Short distance transmission can be DC coupled, but longer distances can cause transmission lines to accumulate charge, thereby biasing the reception on the other end.

*DCI* Digitally Controlled Impedance. See *XCITE*.

*DCM* Digital Clock Manager is a timing resource block that includes a digital delay lock loop, a digital phase selector and a digital frequency synthesizer. DCMs are capable of deriving many user selectable timing signals from a user's clock input. Frequency multiplication, division and clock delay are key capabilities. Replaced by the Multi-Mode Clock Manager (MMCM) in the Clock Management Tile (CMT) in later Virtex devices.

*DCR* Data and Clock Recovery is the ability to recreate a clock from a properly encoded data stream. One such encoding is the popular 8b10b encoding format.

*DCT* Discrete Cosine Transform is a popular DSP operation on video images. MPEG 4 includes DCT as one element in its calculation suite. Xilinx offers DCT solutions through our third party partners, and also uses DCT in many video reference designs.

*DDR* Double Data Rate is a data transfer protocol where data transfers on both clock edges. DDR is a standard for both memory and data communications.

*DDS* Direct Digital Synthesizer is also known as a numerically controlled oscillator (NCO). Xilinx offers a DDS LogiCORE.

*Debug Monitor* Embedded software specifically designed as a debugging tool. Debug monitors usually reside in ROM and communicate with the debugger via serial port or network connection. The debug monitor provides a set of primitive commands to view and modify memory locations and registers, create and remove breakpoints, and execute your program.

*Debugger* is a software development tool used to test and debug embedded software. The debugger runs on a host computer and connects to the target through a serial port or network connection. Using a debugger, you can download software to the target for immediate execution. You can also set breakpoints and examine the contents of specific memory locations and registers.

*Decimate* Originally, to take every tenth item, such as soldiers in the Roman army. Since then, the idea is to sample data and periodically discard samples.

*Delivery Truck Problem* See traveling salesman.

*Density* is a relative measure used to define the amount of logic a device can implement, often expressed as a number of logic gates. This approach was originally taken by ASIC manufacturers and carries over into programmable logic, often with some difficulty.

*DES* Data Encryption Standard (DES) was the first government sponsored, commercially embraced data standard for encryption. DES helped launch ecommerce over the last few decades. It has been supplanted by 3DES and AES. *3DES* Triple DES is an extension of DES, whereby three successive 56 bit keys are applied to the data. The application involves using the first key with incoming clear text (or clear bits, with Xilinx configuration data). Then, an inverse DES algorithm is applied with the second 56 bit key. Finally, a third pass of the data occurs, using DES once again, and a third 56 bit key. In theory, it is a “misnomer” and should be 2DES and IDES, but that’s marketing!

*Design rule* A design rule is a general term for requirements to assure circuits will work. Excluding floating inputs, forbidding tied outputs and limits to electrical loading are common design rules. See also, DRC.

*Detail Router* is a software module that makes low level connections for a placed circuit. See also maze router.

*Deterministic Jitter* Deterministic jitter is the component attributable to the channel data pattern. Different digital patterns have different spectral content. These differing spectra give rise to varying amounts of signal jitter.

*Dhrystone* is a synthetic benchmark for defining processor performance. Very early benchmarks were called “Whetstones”, and were based on real world workload. The strange spelling is a bit of a pun on the original. Other humorous variations have been called “Mhoiststones” and “Dhampstones”, but the humor stretches thin. Dhrystones were first used for rating engineering workstations and had an element of graphics calculation along with scientific instruction mix. One advantage for Dhrystones is that they are not typically memory intensive, so they often fit nicely in L1 caches, where processors usually perform their best.

*Dictionary Attack* is a cryptographic term whereby a judiciously chosen set of keys are applied to a device, to attempt to “crack” it. For Virtex parts, this would be the set of keys found in Virtex II (3DES) and later Virtex (AES). Code book chaining (CBC) reduces the likelihood of this attack.

*Differential Signaling* uses complementary signals to transmit data. Differential signaling offers faster data rates at reduced signal swing, with a higher signal to noise ratio than single ended signaling.

*Direct connect* is a term for nearest CLB neighbor interconnect segment. See single, double, quad and hex.

*Direct Clocking* This is the technique of delaying input data, to be properly captured by a clock, using the Idelay facility within later Virtex devices. The tap delay is determined by the phase difference between the rising edge of the internal clock and the center of the chosen clock pulse/strobe. See XAPP701.

*Directed Acyclic Graph* is a mathematical description frequently used to model circuits. The model is graphic, consisting of nodes and vertices, and an acyclic graph forbids loop back.

*Distributed Arithmetic* (DA) is a class of calculations that is area efficient and avoids explicit multiplications by pre-calculating partial results and storing them in RAM for subsequent assembly as a calculation proceeds. DA FIR realizations are capable of supporting very high data rates.

*Dispersion* is the smearing of a signal or waveform as a result of transmission through a non-ideal medium. Signal components travel at different velocities according to their frequency. Dispersion is the result. All transmission media are non-ideal.

*DFS* Digital Frequency Synthesis is one component of a DCM module.

*DLL* Delay Lock Loop is an all digital timing device using many uniform time delays and elaborate control circuitry to delay a clock signal so its main switching edge arrives uniformly at many places on a chip at once. The DLL is a key component of a Digital Clock Manager, and is designed to work in close cooperation with the clock trees and clock multiplexing structures of Virtex FPGA devices. Replaced later by the PLL and the MMCM.

*DMA Direct Memory Access* is the set of circuitry in a microprocessor that permits high speed direct access to memory. It often works with a discipline of interleaved access called cycle stealing.

*DMIPS* Dhystone Millions of Instructions Per Second. See Dhystone.

*Double connect* is a term for a short hop interconnect segment. It is literally jumping over two neighboring CLBs to access the next physical layer of CLB choices. See direct, single, quad and hex.

*DPS* Digital Phase Shifter is a portion of a Xilinx DCM that can introduce programmable time delay into various clocking signals. Feature is preserved in the MMCM.

*DRAM* Dynamic Random Access Memory is a general category of small cell memory structure requiring periodic refreshing in order to retain its contents even though it remains powered up. Synchronous DRAM (SDRAM) is a more recent version, introducing clocking requirements.

*DDR SDRAM* is still another more recent version introducing faster bandwidth by virtue of passing data on both clock edges.

*DRC* Design Rule Check is a discipline of verifying circuits usually with software. It encompasses a number of checks, including loading, proper connection to primary I/Os, correct identification of signal direction, and a whole host of other pertinent checks. See also, design rule.

*DSP* Digital Signal Processing includes a whole host of calculation algorithms, including digital filtering, discrete signal transforms, decision circuits and any number of calculations where sequences of consecutive signal samples are modified. Digital delay would also be a form of DSP. Alternately, a specialized microprocessor for performing DSP is also referred to as a “DSP.”

*DSP48* Virtex 4 DSP slice including multipliers, adders, multiplexers and registers for high speed DSP calculation. Also handles rounding, sign extension and various DSP housekeeping activities.

*DSP48E* Virtex 5 DSP slice with additions beyond the Virtex 4 DSP48 slice.

*DSS* Digital Spread Spectrum; also Digital Satellite System.

*DUC* Digital Up Converter modulates a base band signal to an intermediate frequency, for use as input to a digital to analog converter. It has a configurable data path comprising a cascade of two poly-phase interpolator FIR filters, a CIC filter, a DDS and a mixer.

*DCD* Duty Cycle Distortion Transistors are not perfect. Switching speed differences between n-channel and p-channel devices results in altering an input signal duty cycle to something different at the output. Usually, this is most evident in simple circuits that should resist distortion, like clock drivers.

*Dynamic current* is switching current that is driving a signal. In theory, CMOS only dissipates power by virtue of its dynamic current component. See also static current.

*Dynamic Power* dissipation is associated with the flow of dynamic current. This power is also related to capacitance being charged by the dynamic current.

*Dynamic range* Dynamic range is the range of numbers that can be represented before overflow or under flow occur in DSP calculations. This determines when the values need to be scaled.

*ECC* Error Correction Codes come in many forms. Parity and Hamming codes fit the model of error correction, but the basic idea is to judiciously add a few bits to data that will increase the likelihood of correctly recreating the original data after transmission across a noisy channel.

*EDA* Electronic Design Automation.

*EDIF* Electronic Data Interchange Format was developed to create a netlist based descriptor for integrated circuits. It presents different “views” of a design, based on a specific interest. For example, there is a logic view, and a test view. The idea was to provide all the necessary information to design, model, simulate and test devices with a common format.

*EDK* Embedded Development Kit is a collection of software tools to support an embedded microprocessor such as PPC405 or MicroBlaze.

*Effort* as used in FPGA design software, translates directly into time. High effort place and route is generally higher quality, as the tool tries more arrangements and routings to obtain the best possible solution. Low effort place and routing is usually quick, and not close to optimal.

### *EMC* Electro Magnetic Compatibility

*EMI* Electro Magnetic Interference is basically, electronic noise created from switching signals on a printed circuit board, a cable, an antenna or other signal medium. The amount of interference an electronic device can emit is regulated by various government agencies.

*Equalization* is amplification or attenuation of certain frequency components of a signal. Equalization is used to counteract the effects of non-ideal transmission media.

*Eye Diagram* of a signal overlays the signal waveform over many cycles. Each cycle waveform is aligned to a common timing reference, typically a clock. An eye diagram provides a visual indication of the voltage and timing uncertainty associated with the signal. It can be generated by synchronizing an oscilloscope to the timing reference.

*Eye Mask* The size of the eye opening in the center of an eye diagram indicates the amount of voltage and timing margin available to sample this signal. For a particular electrical interface, a fixed reticule or window could be placed over the eye diagram showing how the actual signal compares to minimum criteria window, known as the eye mask. If a margin rectangle with width equal to the required timing margin and height equal to the required voltage margin fits into the opening, then the signal has adequate margins. Voltage margin can often be traded off for timing margin.

*Fabric* The arrangement and physical relationship of components or constituent elements of something. Used with FPGA devices, fabric generally refers to the logic elements and the interconnect hardware that make up the FPGA device.

*FAE* Field Applications Engineers are digital design engineers trained to be expert with Xilinx programmable logic in order to assist Xilinx customers with their designs. Xilinx FAEs tend to be magicians capable of handling multiple different designs with different customers simultaneously.

*Fast carry* describes special FPGA paths to quickly determine and deliver addition carries are called fast carries. Xilinx has many fast carry patents. See also carry chain and carry look ahead.

*Faults* are basically broken circuits. Testing is designed to recognize faults in a digital circuit by applying a set of inputs so that an output that is incorrect is exposed. Fault simulation is a method of observing the switching behavior of a netlist and inserting broken nets into the design, to determine whether the applied vectors can produce resulting vectors that are different from correct resulting vectors. Common fault models insert “stuck at 0” or “stuck at 1” nodes into the design. Sticking two nodes together (bridging fault) is common for testing very regular structures like memories.

*FEC* Forward Error Correction is a means of pre- encoding data for future recovery after corruption. Many means exist, all of which reduce available message bandwidth. See also ECC.

*FFT* Fast Fourier Transform is the general class of DSP solutions to calculate the Discrete Fourier Transform, where data restricted to blocks of  $2N$  samples can be frequency transformed with high speed and minimal storage requirements. The Inverse FFT (IFFT) has similar qualities. In earlier days, FFTs fell into two versions named by their developers – Cooley and Tukey developed one, while Gentleman and Sande developed another.

*FIFO* First In, First Out is a data structure frequently implemented in RAM with an external controller. It has the useful property of independent read and write behavior, that permits writes to be clocked in one clock domain and reads to be clocked in another. Virtex 4 and Virtex 5 offer specialty hardware to implement FIFO using BRAM, where other Virtex products offer standard fabric with BRAM to create FIFOs. FIFOs are the standard solution to marrying high speed data paths across different clocking regions without problems. Mathematically, FIFOs are referred to as queues.

*FIR* Finite Impulse Response filters are key building blocks in DSP. They have no feedback, and typically produce results depending on just a few stored results (low latency). They are unconditionally stable, and come in a number of different styles. Whereas linear filters are usually described in terms of poles and zeroes, FIR filters are more frequently described in terms of taps.

*Flag* is also known as a semaphore. Flags can be either hardware (flip flops) or “software” (register bits or RAM cell bits).

*Flip chip* is packaging technology, where the die is inverted onto a substrate, so it’s “topside” is face down, and its bottom substrate is up. Connections are made by tiny solder balls attached to the die that bond to the substrate of the package. Low inductance is a key quality of this technology giving great results for large number of fast switching signals. Decreases ground bounce. See sparse chevrons.

*Flip-flop* is a standard logic storage cell. Many types exist, but the most silicon efficient appears to be the D flip-flop. Others exist, but are not common in programmable logic, such as the RS and JK. It’s common in programmable logic to include flip flops that can be modified for greater efficiency to other configurations such as a “T flip flop” or a transparent “D latch”. Flip flops require some form of data input and a clocking event. Rising edge, falling edge and dual edge flip flops are variations.

*FLOPS* FLoating Operations Per Second are a metric for high speed calculation, that include both a multiplication and an addition, but specifically with floating point numbers. Surprisingly, in floating point format, multiplication is often easier than addition. Part of the issue is operand normalization and managing overflows.

*Floor planner* is a design software tool for graphically arranging the logic contents of an FPGA fabric manually. The process strongly resembles the same procedures used in printed circuit board layout and manual editing. The terms “rip” and “reroute” are colorful, self explaining verbs in the world of floorplanning. Sometimes, the term floorplanning is used for the visualization process of geometrically defining the presence and orientation of blocks on the FPGA “floor”.

*Flush* is a process of declaring memory contents invalid. Cache memories can be flushed by simply setting all associated “valid bits” to zero. In video memory structures, flushing may involve writing appropriately “blank” data into the video RAM.

$F_{MAX}$  is the maximum operating frequency is a term applied in a number of ways. For a simple state machine like a counter, it describes the fastest continuous, error free clocking operation of the state machine. Other state machines follow suit, as long as there is a single clock. For a flip flop,  $F_{MAX}$  is context dependent upon the feedback from the flip flops output back to its input. The net delay of the feedback added to the switching requirements of the feedback dictate the  $F_{MAX}$ .  $F_{MAX}$  is always less than or equal to  $F_{TOG}$ .

*Footprint* is used in many ways, but specifically with BGAs, it is the mechanical location of the balls on a BGA package.

*FPGA* Field Programmable Gate Array is an obligatory acronym for this glossary. Early FPGA devices included programmable logic cells with programmable interconnection. Xilinx pioneered this field and first provided segmented interconnect, which gave good results for making connections. Virtex FPGA devices improved this by introducing highly buffered, hierarchical interconnect, with multiple I/O standards, many choices on specialty blocks (BRAMs, Multipliers, MGTs, Microprocessors), as well as timing resources (DCM, ChipSync, etc.) to give designers flexibility and performance to solve most logic system problems encountered.

### *FPGAs* FPGA Devices (noun form)

*FPGA Editor* is a graphic design tool displaying an FPGA floorplan that permits users to manually modify their designs. Replaces by the floor planner in Vivado.

*FPU* Floating Point Unit is frequently a portion of a microprocessor, handling floating point multiplication, addition, and division. Most microprocessors inherently handle fixed point or integer calculations. Some microprocessors perform floating point strictly in software, so an FPU might be viewed as a type of accelerator.

*Frame* is an element of FPGA configuration, which roughly correlates to a column or sub-column of CLBs and associated interconnection. The exact size varies between FPGA families, but is often referred to when discussing details of configuration.

*FSK* Frequency Shift Keying is a type of signal modulation. With FSK, a logical one is represented by a different frequency than a logical zero.

*F<sub>TOG</sub>* is the symbolic descriptor for fastest flip flop toggling frequency. It is the upper limit on flip flop switching speed, where it changes state on every clocking event.

*Function generator* is another term for lookup table.

*Functionally complete* is a mathematical property of a logic element to build any logic function from that element, alone. NAND gates are functionally complete, as are NOR gates. LUTs are functionally complete, also.

*Gate* has many definitions. First, a gate is a logic building block which can typically be a NAND, NOR, OR, AND, EX-OR, etc. In CMOS technology, the term gate is usually meant to describe a 2 input NAND gate consisting of four transistors connected. Gate counts, or gate equivalents are given in terms of two input CMOS gates. Usable gates and system gates are variations from the CMOS definition.

*Gate Array* is an ASIC technology where foundations of unconnected transistor columns are offered in fixed sizes and pin outs for users to build completed solutions. Users are provided cell libraries, capture tools, simulation tools and assorted design rule checking software to prepare their netlists so that manufacturers can complete the design by adding metal connections to the transistor columns. The software flow is similar to that of an FPGA design flow.

### *GBT GigaBit Transceiver*

*Global Router* is software that makes general assignments for signals, with respect to route direction. The global router does not make specific routing metal assignments. See “detail router”.

*Glue* is a generic term for logic that has no specifically identifiable function other than “holding together” other better defined blocks. Glue can include gates, registers, multiplexers, decoders, etc. Adders, multipliers and RAMs are examples of blocks. Glue is used to logically “stitch” blocks together. Logic fabric can be thought of as glue.

*GRM Global Routing Matrix* is a switching resource where various segmented interconnect lines get routed through. See also PSM. *GTL Gunning Transceiver Logic* is a type of logic that obtains speed by virtue of a very small signal swing. Another version, *GTL+* is similar and was defined originally by Intel and used in Pentium processors.

*GTP Gigabit Transceiver for low Power* is one Virtex 5 option for a high speed transceiver favoring power over speed.

*GTX Gigabit Transceiver for high speed* is one Virtex 5 option for a high speed transceiver favoring speed over power.

### *GUI Graphical User Interface*

*H.264* is a popular international standard for video decompression. Xilinx offers intellectual property for this decoder through third parties.

*H.323* is International Telecommunications Union data communications protocol for a broad class of teleconferencing applications. H.323 covers items like Voice Over Internet Protocol (VOIP), video, FAX, etc.

*Hardware Over Sampling* Oversampling is a common technique in DSP that can simplify filter design in some cases. It involves sampling a signal at a rate greater than the signal bandwidth dictates by the Nyquist sampling theorem. See CIC filter.

*Hardware in the Loop* see Co-simulation.

*HDL* Hardware Definition Language is a general term for high level, abstract design languages like Verilog and VHDL. The concept can extend to other syntaxes.

*Hex* is a term for a medium hop interconnect segment. Literally jumping over six neighboring CLBs to access the next physical layer of CLB choices. See direct, single, double and quad.

*Hill climbing* is a type of algorithm capable of getting out of local minima. Simulated annealing is one method.

*HLL* Common abbreviation of High Level Language. Any of a number of programming languages, including C, C++, Java, Fortran, BASIC, TRAC, Pascal, Python, Forth, etc. Language support for embedded processors is critical for users to make best use of them.

*HSTL* High Speed Transceiver Logic is another I/O standard sometimes found on EPROMs and other technology.

*IBIS* I/O Buffer Information Specification is a standard for electronic behavioral specifications of integrated circuit input/output analog characteristics. At the heart of an IBIS model is a table of voltage versus current values and timing. IBIS models are complete enough to include process variation information, to assess performance range and deviations with temperature, voltage and process. IBIS is an industry standard modeling format and is critical for accurate board level simulations.

*IBUF* Input BUffer is the input receiver at the pad of an FPGA. IBUF is also a library component assigned to FPGA input pads.

*ICAP* Internal Configuration Access Port

*Idelay* Idelay is the input delay circuitry that can adjust the delay of an input signal to best align to the “open eye” of a received clock.

*Idle Pattern* During many data communication protocols, there is a training period, whereby the clock is extracted and locked. A similar pattern is used in between data transmission episodes, so that clocks persist in their locked condition. That pattern is called an idle pattern.

*IEEE 802.x* Institute of Electrical and Electronics Engineers Standard 802 and its many subcategories. Some examples follow:

IEEE 802. 1 High Level Interface

IEEE 802. 10 Standards for Interoperable LAN Security

IEEE 802. 11 Wireless LAN

IEEE 802. 14 Cable-TV Based Broadband Communication Network

IEEE 802.15 Wireless Personal Area Network

IEEE 802. 16 Broadband Wireless Access

IEEE 802. 3 CSMA/CD

IEEE 802. 4 Token Bus

IEEE 802. 6 Metropolitan Area Network

*IEEE 1149.1* is the IEEE standard that covers in system testing. It evolved out of an earlier effort referred to as JTAG – Joint Test Action Group. Basic ideas include a required set of four pins and a minimum, specific instruction capability, such as Bypass, Sample/Preload and Extest. IEEE 1532 is a successor to IEEE 1149.1, which involves using the IEEE 1149.1 pins and state machines to perform in system programming.

*ILA* Integrated Logic Analyzer (as in ChipScope ILA debugging tool).

*iMPACT* is a Xilinx command line and GUI based tool that enables you to configure your PLD designs using Boundary-Scan, Slave Serial and Select Map configuration modes, as well as the MultiPRO Desktop Programmer. You can use iMPACT to download, readback and verify design configuration data as well as create PROM, SVF, STAPL, System ACE CF and System ACE MPM programming files. Replaced by Hardware Manager in Vivado.

*IMUX* Multiplexer inputs to a CLB.

*Instantiate* is the act of creating an “instance” of something. In a design, instantiation might be creating an instance of an adder, from a library. The action typically distinguishes two different instances, avoiding confusion. Instantiation in programming is often the process of replacing variables with specific values.

*Interrupt* is an asynchronous electrical input from a peripheral to a processor, automatically altering the control flow of program execution. When the interrupt occurs, the current processor state is saved and control transfers to an interrupt service routine. The standard procedure upon completing the service routine is to restore the processor state and resume execution at the point of interrupt.

*Interrupt Latency* is the time delay between interrupt assertion and service routine entry.

*Interrupt Service Routine* is software written to identify the cause of an interrupt and take appropriate action for providing service.

*Interrupt Vector* is a hardware facility whereby the interrupting device inserts a value onto the processor bus, becomes the address for its service routine. In some cases, it is the exact address where in others, it is a symbolic reference to another mapping table for the real address.

*I/OB* Input Output Buffer. I/O driver for an FPGA pin pad. See also IBUF.

*IP* Intellectual Property in programmable logic, this usually means a configuration bitstream that solves a particular problem and is commercially available. An example would be an MPEG-4 decoder block. Xilinx offers a wide range of IP, along with third party partners that provide solutions as well. All Xilinx IP is free to use in Xilinx components. See specific legal terms on the Xilinx legal web page.

*ISE* Integrated Software Environment is the name of Xilinx software that initially supported the Virtex family FPGA devices. Replaced by Vivado for Virtex 7 and later devices.

*ISERDES* Xilinx marketing term for Input SERDES, but the functionality is that of an input deserializer. Part of the later Virtex devices ChipSync feature set.

*ISI* Inter-symbol Interference is a form of data corruption stemming from data-dependent channel characteristics.

*Isochronous* refers to a communication protocol based on time slices rather than handshaking. For example, a process might have 20 percent of total bus bandwidth. During its time slice, that process can stream data.

*ISP* In System Programmable describes the ability to program a configuration pattern into a PLD while it's attached to a printed circuit board.

*Jitter* The jitter of a periodic signal is the delay between the expected transition of the signal and the actual transition. Jitter is a zero mean random variable. When worst case analysis is undertaken, the maximum value of this random variable is used. See also deterministic, cycle to cycle and period jitter. Note that jitter gets corrected (incorrectly) as 'jitters' by most spell checkers.

*Jitter Tolerance* is defined as the peak-to-peak amplitude of sinusoidal jitter applied on the input that causes a predefined, acceptable loss at the output. For example, jitter applied to the input of an OC-N equipment interface that causes an equivalent 1dB optical power penalty.

*Jitter Transfer* is defined as the ratio of jitter on the output of a device to the jitter applied on the input of the device, versus frequency. Jitter transfer is important in applications where the system is utilized in a loop-timed mode, where the recovered clock is used as the source of the transmit clock.

*JTAG* Joint Test Action Group is the name of the boundary scan founding organization that eventually evolved the test standard currently better known as IEEE 1149.1.

*K-character* When using 8b10b encoding, K-characters are used to carry specific control or information designations (such as data alignment).

*Kernel* has two meanings found in FPGA designs and devices. First, for embedded systems, the “heart” of an operating system is often referred to as its kernel. In DSP, the primary calculation is called a kernel. The kernel calculation for an FFT would be a complex butterfly calculation.

*LAN* Local Area Network is one of many data communication systems characterized by high speed and restricted geographical domain – typically, a small distance between network nodes. Ethernet would be a common example of a LAN, but others exist.

*Latency* is the time delay between the arrival of an event and the occurrence of the corresponding effect or result of the event. Pipelining a design by introducing flip flops introduces latency by requiring more clock cycles. See also, Interrupt latency.

*LDT* Lightning Data Transport is a chip-to-chip interconnect standard that provides a bandwidth from 6.4 Gb/sec per eight wire link width, and can support up to 32 links.

*LFSR* Linear Feedback Shift Register is a finite state machine capable of producing pseudo random sequences. They exhibit a repeating pattern (cycle), and are usually one of two varieties – Galois or Fibonacci. LFSRs are fast, logically simple and efficient, and used in many ways.

*LIFO* Last In, First Out (see Stack)

*Linear Decoding* is a method of addressing a circuit using a single address line.

*Linker* A linkage editor is often also called a “linker”. This software module performs the task of linking various re-locatable modules of code together, so that all variable accesses and branches are defined.

*Little-endian* A representation of a multi-byte value that has the least significant byte of any multi-byte data field stored at the lowest memory address, which is also the address of the larger field.

*Locking* A lock constraint in the placement constraint file (PCF) locks a component. A lock routing constraint specifies that the current routing cannot be changed or un-routed. A lock placement constraint specifies that placed components cannot be unplaced, moved or deleted.

*Logic Analyzer* is a real time hardware debugging tool capable of recording many logic signals versus time, and displaying chosen sets of them. A critical capability of a logic analyzer is its ability to “trigger” on chosen combinations and/or sequences of the signals, to reduce the data being captured, and focus on timeframes of interest. This ability lets designers investigate the series of activities leading up to a trigger, or alternately inspect the resulting activities after a trigger. Each signal is associated with a “channel” of the logic analyzer, and key attributes include channel number, channel depth and capture modes (synchronous, asynchronous). Some logic analyzers include disassemblers, so program execution can be traced, when attached to a microprocessor.

*Logic cell* The standard use of logic cell in FPGA parlance is a four input LUT and its D flip flop. The six input LUT may change this, in time. See also CLE.

*Logic Cell Array* Also known as LCA, was the early reference to Xilinx FPGA architecture. XC2000, XC3000, XC3100 and the various XC4000 families are termed LCA families, as opposed to Virtex families.

*LogiCORE* is Xilinx productized design cores. LogiCORE is also available as part of the System Generator block set.

*Logic element* is a building block, defining the logic in a design. These elements are typically primitives – that is, flip-flops, AND gates, and such elements - or macros, higher level combinations of primitives.

*Logic fabric* In FPGA devices, logic fabric consists of the CLB contents and the routing resources within the FPGA. Pre-Virtex products primarily consisted of logic fabric. This is also simply known as fabric.

*Logic gate* Classic AND, OR, NAND, NOR, EX-OR gates are all logic gates. As a general term, the CMOS two input NAND gate is the standard logic gate for capacity counting. See also gate.

*Long lines* are the longest internal connection resources within an FPGA, spanning the width or height of the die in early FPGA devices. Virtex family long lines have varying spans, depending on the family. Long lines are frequently described as horizontal or vertical denoting on chip arrangement.

*Loop back* is a mode of data transmission usually used for diagnostics. Loop back can be done within an MGT, by tying the out going data stream back into the incoming data stream. If it loops back totally within a device, it is sometimes called “local loop back.” Other loop back paths can go just outside the device, by tying the output pins back to the input pins. Full loop back would be sending the data across the data channel, where the receiver simply echoes it right back to the sender. Test activities to establish the integrity of a data channel, or identify sources of error involve looping back.

*LUT* Look Up Tables are key function generation resources in FPGA devices.

*LVCMOS* Low Voltage CMOS is any of the JEDEC CMOS standards below 5 volts. This includes 3.3v, 2.5v, 1.8v, 1.5v and 1.2v CMOS.

*LVDS* Low Voltage Differential Signaling is a signaling method using differential current mode signaling. Several versions exist, including Bus LVDS and Multi-drop LVDS.

*LVTTL* Low Voltage Transistor Transistor Logic is the 3.3V JEDEC TTL standard. Many CMOS products are designed to switch at input thresholds that match those of LVTTL, which is still a common logic technology.

*MAC* Media Access Controller is the section of a data communications link that controls access to the outside medium.

*MAC (MACC)* Multiply and ACCumulate is a standard operation in DSP.

*Manhattan Distance* is a colorful term used to describe metal distances on integrated circuits that are constrained to X and Y coordinates. Standard cell and gate array ASICs usually restrict connections to vertical and horizontal channels. FPGA devices adopted the same approach, in general. You can think of getting between sites in a large city as being constrained to the sidewalks and streets, which are typically X and Y grid oriented.

*Mapping* is the process of assigning a design's logic elements to the specific physical elements that actually implement logic functions in a device.

*Master mode* describes one mode an FPGA configuration circuit may be in. Typically, the first device in a chain of devices will be in "master mode". It comes with certain signal forwarding responsibilities.

*Maze Router* is the classic, detail router that originally modeled connection paths as pursuing a goal (a.k.a. "the cheese") inside a maze. Early work by Claude Shannon defined the problem mathematically.

*Mb* Megabit, often 1,048,576 bits ( $2^{20}$ ,  $2^{20}$ ), and not 1,000,000

*MB* Mega Byte, see above as often not 1,000,000

*Memory coherency* is the action of maintaining the integrity of a multi-port memory, so that corrupt data does not persist. The coherency strategy involves deciding who "wins" if two simultaneous writes occur, or when one address is being read from, while the same address is being written into, which value does the reading entity receive. Different policies exist. Many memory coherency strategies employ a special kind of flag (or semaphore) quaintly known as a "dirty bit", associated with a memory block to indicate corruption.

*Memory map* is a data structure that relates various virtual addresses to real addresses. Using a memory map, it is possible to treat a disk as if it were a large RAM. This would be a virtual memory. Caching also involves memory mapping, whereby memory blocks may exist in both external memory as well as in a high speed, local memory. The memory map tells where to look. Most memory mapping is managed by software.

*Memory Mapped I/O* Different processors distinguish I/O devices from memory addresses in various ways. One way to perform I/O is to simply assign memory addresses to an I/O device, so data stores going to that address, they actually transfer to an output device. Similarly, reads of a memory address become data input from the outside world, when the address selects the input device. This process maps I/O devices into the memory address space.

*Mesochronous* describes clocks that have the same frequency, but may have different clock phases.

*Metastability* An undesirable, marginally stable output state that can occur in logic circuitry. These states can occur in such bi-stable storage elements as registers, latches and memories when the setup and/or hold times are not met in relation to their trigger or clock input.

*MGT* Multi Gigabit Transceiver is a high speed serial communication block available in selective Virtex-II and higher FPGA models. See also GBT, GTP and GTX.

*MicroBlaze* is a 32 bit processor design that Xilinx offers customers seeking the convenience of using a “soft processor” created from the Virtex logic fabric and BRAMs. It comes with complete EDK support.

*Microcontroller* As with microprocessor, microcontroller is not trivial to define. However, microcontrollers tend to have smaller word sizes than microprocessors, are Harvard (separate instruction and data memories) and may access only local control storage. See also microprocessor. Microprocessor as a rule, fall into two categories – von Neumann (i.e. Princeton) or Harvard. These categories focus on where data and instructions reside, but multiple classifications exist, with another one being CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer). Even others exist, such as MIMD, SIMD, systolic, dataflow, and so forth. Clearly, books are written on this topic. PPC405 would be an example of a RISC, as is MicroBlaze, as are the ARM processors. See microcontroller.

*Microstrip* is a printed circuit layout discipline for high speed.

*MIPS* Millions of Instructions Per Second is a simple measure of processor performance.

*Miss* is the action of accessing memory that should reside in a cache, only to discover that it is not present. The consequences of a cache miss are that the correct block of data must be found in slower, external memory, and placed into the cache. Often, that requires placing the new block on top of an already occupied slot in the cache, so care must be taken to minimize future misses by replacing the least useful block with the newer block. A result of poor replacement strategy is the high overhead activity known as thrashing, whereby the benefits of high speed local block copies are lost.

*MMU* Memory Management Unit is a common microprocessor module that assures that various memory blocks are resident. MMUs usually work in conjunction with cache controllers. Some MMUs implement virtual memory, so that large archival storage (i.e., disks) appear as part of a standard memory space.

*Modulo arithmetic* (Or "clock arithmetic") is a kind of integer arithmetic that reduces all numbers to one of a fixed set [0...N-1] (this would be "modulo N arithmetic") by effectively repeatedly adding or subtracting N until the result is within this range. Finite field theory describes this in detail.

*MOSFET* Metal Oxide Semiconductor Field Effect Transistor

*MSB* Most Significant Byte

*MSPS* Million Samples Per Second

*MTU* Maximum Transmission Unit is the largest unit of data that can be transmitted on any particular physical medium.

*Multiplexer* (MUX) is a hardware device that enables two or more signals (analog or digital) to be transmitted over the same circuit by temporarily combining them into a single signal. On the receiving end, the signals are separated by a de-multiplexer.

*FDM* Frequency Division Multiplexing: each signal is assigned a different frequency.

*TDM* Time Division Multiplexing: each signal is assigned a fixed time slot in a fixed rotation. See isochronous.

*STDM* Statistical Time Division Multiplexing: time slots are assigned to signals dynamically to make better use of bandwidth

*WDM* Wavelength Division Multiplexing: each signal is assigned a particular wavelength; used on optical fiber

*Net* is a logical connection between two or more symbol instance pins. After routing, the abstract concept of a net is transformed to a physical connection called a wire.

*Net delay* is delay time associated with connecting two logic cells. Net delay is typically the resistor capacitor (RC) time delay, along with any intrinsic buffer delays encountered.

*Netlist* is a text description of the circuit connectivity. It is basically a list of connectors, a list of instances, and for each instance, a list of the signals connected to the instance terminals. In addition, the netlist contains attribute information (such as the instance function, or time delay values).

*Netlist Optimization* is a design improvement method performed on the netlist version of the design, by a set of rules and substitutions. See optimization.

*NIST* National Institute of Standards and Technology is a federal standards body. For example, AES and DES are NIST standards for encryption.

*Nonmaskable Interrupt* Many interrupt signals can be blocked by software, with a “disable interrupt” command. This introduces the risk of some interrupting events, which must be serviced, being blocked. Nonmaskable interrupts introduce a scheme that software cannot block.

*NRE* Non Recurring Expense (sometimes called Non Recurring Engineering Expense) is the idea of charging a single fee for developing and creating an initial prototype of a circuit. It can be used in many ways, but most frequently it is used with ASIC designs, and the primary component is the IC mask creation fee. In reality, well over 90% of all ASIC designs incur subsequent mask fees for repairs to be made, so calling it NRE is somewhat bogus.

*OC* Optical Carrier. The transmission speeds defined in the SONET/SDH specification. OC defines transmission by optical devices, and STS is the electrical equivalent.

### *OCM* On Chip Memory

*OCM Controller* supports attachment of additional memory to the instruction and data caches that can be accessed at performance levels matching the cache arrays.

*OC-192* Optical Carrier 192 (10 Gbits/Sec) in the SONET/SDH standard.

*Odelay* is a tunable output delay available on I/O pins for later Virtex devices.

*OMUX* The multiplexing output structure of a CLB is termed the OMUX.

*OPB* On-chip Peripheral Bus is architected to alleviate system bottlenecks by reducing capacitive loading on the Processor Local Bus (PLB). OPB supports lower performance peripherals such as I2C, UART, GPIO, etc. This IBM terminology and standards is replaced by the ARM AXI bus in 7-series and later devices.

*Optimization* is a term used to describe the process of transforming a design to reduce area, increase speed, or both. A more apt term would be improvement, as optimization suggests achieving the limit in improvement.

*OSERDES* Output SERDES is an element of ChipSync that takes parallel data and shifts it to an output pin. A more apt term would be OSER, as it is in fact an output serializer.

*OSI RM* Open Systems Interconnect Reference Model is a model of network architecture and a suite of protocols (a protocol stack) to implement it. OSI RM was developed by ISO in 1978 as a framework for international standards in heterogeneous computer network architecture. The OSI architecture is a series of non-proprietary protocols and specifications that are split into seven layers, from lowest to highest: 1 physical layer, 2 data link layer, 3 network layer, 4 transport layer, 5 session layer, 6 presentation layer, 7 application layer. Each layer uses the layer immediately below it and provides a service to the layer above. In some implementations a layer may itself be composed of sub-layers.

*PAR* is a Xilinx software module responsible for cell Placement And Routing, which are key aspects of the physical synthesis process.

*Partitioning* is the process of splitting a single design among multiple devices, or resources.

*PCI* Peripheral Component Interconnect is a fast, fully synchronous bus designed primarily for personal computers, but used in many ways. Key ideas include detailed compliance requirements and incident wave switching of signals. Originally designed for 32 bit transfers at 33 MHz clocking, it currently has many other varieties.

*PCle* PCI express is a newer standard that achieves higher performance by offering a more “serial” format based on unidirectional single bit data lanes.

*PCS* Physical Coding Sub-layer

*PECL* Positive Emitter Coupled Logic is an all positive, differential current mode signaling standard primarily used for high speed clocking.

*Period Jitter* is the worst-case deviation from the theoretic perfect clock period of all clock cycles in the collection of clock periods sampled (which is usually 100,000 to 1M+ samples for specification purposes). In a histogram of period jitter, the mean value is the clock period. Typically specified as peak to peak, and/or RMS (root mean square).

*PGA* Pin Grid Array package Originally, FPGA devices were called PGA or programmable gate arrays.

*Phase noise* is a type of jitter. It is often related to an untimely edge arrival, due to added noise.

*Phase shift* is another way of describing time delay.

*Physical design* is the concrete (i.e. not abstract) actions of placing and routing.

*Physical synthesis* is synthesis at the RTL level, which includes target architecture layout delay information. The architecture layout information better guides the synthesis tools than simple wire loading models used to predict time delay. Unlike ASICs, which present a continuum of time delay, there are fewer timing choices in FPGA architectures, quickly determining feasibility of a particular choice.

*Pinout* is a diagram that indicates how I/Os are terminated to pins in a connector. It may also be a list assigning device functions to specific BGA balls or pins.

*PIP* Programmable Interconnect Points are the programming sites that make or break connections in FPGA devices. The simplest PIP is a pass transistor with its gate driven from a configuration cell. More elaborate PIPs exist, resembling multiplexers, or transmission gates. PIPs are discussed in the first three chapters of this book.

*Pipulation* is a Xilinx pun on the word “population” applied to PIPs. Pipulation describes a set of bits in configuration space devoted to connection.

*Pitch* In integrated circuits, pitch is a distance between two like things. For instance, the distance between two balls on a ball grid array is called a pitch. The distance between two metal lines on an IC is also called pitch. Most frequently, it is described from center to center of the respective things.

*Pipeline processing* is breaking a task into multiple smaller tasks, which proceed both in sequence and in parallel, with multiple operation units.

*Pipelining* is the action of breaking a design into smaller steps that can be operated on sequentially and in parallel, by introducing register stages appropriately to stage intermediate results. Pipelining can result in faster clock rates, at the expense of greater initial latency time, as the pipeline fills.

*Placing* is the action of assigning specific functions to exact locations in the logic fabric. See also routing.

*PlanAhead Floorplanning* software originally developed by Hier Design.

*PLB* Processor Local Bus is the internal, on chip bus for the PPC405.

*PLCC* Plastic Leaded Chip Carrier is a type of surface mount package.

Plesiochronous systems have multiple clocks very nearly the same frequency, but the frequency difference introduces a slowly varying phase difference among clocks. The combination of atomic clocks world-wide is considered plesiochronous. All information is sent to BIPD (Bureau International Pois et Dures – International standards body of weights and measures ibn Paris, France), and a month later, the error from what is considered the actual frequency and phase is sent back to each reference clock host.

*PLL* Phase Lock(ed) Loop is a timing control device capable of tracking an incoming clock signal and tracking it, while producing an error signal. MGTs typically use phase lock loops, which are analog in nature.

*PMA* Physical Media Attachment

*PMCD* Phase Matched Clock Divider

*PPP* (also P3) Pack, Place and Physical Synthesis

*PPR* Partition, Place and Route

*PQFP* Plastic Quad Flat Pack is a type of surface mount package.

*PRBS* Pseudo-Random Bit Sequence is most frequently the output of an LFSR.

*Precision* is the maximum nonzero bits representable as a binary number. It measures the number of bits that can be used to represent each number.

*Pre-emphasis* is a type of equalization that boosts the signal level at one point (say, the transmitter) in anticipation of it being subsequently reduced in the channel, to compensate for the loss that will occur in the channel.

*PREP* PRogrammable Electronics Performance Company was a programmable industry consortium of companies to promote benchmarking programmable logic parts. There were nine agreed upon benchmarks that were used to assess density and performance. Although the rules of the game were mutually agreed upon, they were still open to interpretation and served marketing better than engineering, ultimately.

*Process* can mean multiple different things, but the primary meaning in this text, is CMOS process. Virtex family parts evolved from 0.25 micron down to 65 nanometer in Virtex 5, which include the parts discussed in this text. At the time of this revision, we are designing at 7 nanometers. Mentioning the size attribute simply designates the gate size of the CMOS transistors being used. The last International Technology Roadmap for Semiconductors (ITRS) report has been issued, so Moore's Law is officially broken. From the gate size, experts can determine other attributes like oxide thicknesses, metal widths and so forth. In reality, today's processes are hybrids, covering aspects of many processes, but the smallest process represented is usually described as the "process".

*PSM* Programmable Switch Matrix

*Pyramiding* is multiply layered combination logic with a wide set of first layer gates, reducing the number of inputs at each consecutive layer, to make a triangular shape of gates. Usually, more than two levels of gates.

*Q* See Quality Factor.

*Quad* is a term for a medium hop interconnect segment. A "quad" is literally jumping over four neighboring CLBs to access the next physical layer of CLB choices. See direct, single, double and hex.

*Quality Factor* The quality factor of a filter is a measure of the distance between the upper and lower frequency points and is defined as the center frequency divided by the bandwidth. As the pass band gets narrower around the same center frequency, the Q becomes higher.

*QDR RAM* Quad Data Rate RAM is a RAM architecture capable of transferring four data items per clock cycle.

*QFP Quad Flat Pack*

*QOR Quality Of Results* is a general set of merit figures for design software.

*Quantization* is the action of assigning a value to a signal sample. Quantization is limited to the accuracy and precision available in the A/D converter, and always introduces some level of quantization error.

*Quenching* is a stabilizing step in simulated annealing where the modeled temperature (corresponding to energy) is reduced. This limits the number of possible moves thereafter and helps guarantee the process ends. It is the mathematical equivalent of placing hot metal into a cooling liquid.

*RAM* is the acronym for Random Access Memory. Most often, this means a static memory, needing no refresh circuitry. Other than high speed and low power, it is characterized by the ability to apply any binary address to the device and read or write at random, as needed.

*Random jitter* is caused by power supply noise, resistive (thermal) noise, poor signal integrity, and temperature variations. Basically, if anything is not done perfectly, jitter is the result.

*Range* is the difference between the most negative and most positive representable numbers.

*Rats nest* consists of lines that are point to point connections between un-routed pins on a given net. It may also be a graphic description of tangled, congested design routing resembling a rodent lair.

*RCT Resource, Congestion and Timing* is a more recent strategy used in recent Virtex design software. Based to some degree on the Pathfinder algorithm, one big idea is to quickly evaluate a placement for acceptance by determining whether meeting connection constraints would be feasible, if each net could be assigned the best possible path. If the placement meets that test, negotiation for which connections get the best ones is based on need, to meet constraints. This saves much time routing unusable placements.

*Register* In programmable logic, a logical storage element as little as a single flip flop (usually D type), but arbitrarily larger, as a set of such flip flops. In microprocessors, a set of flip flops usually of the width of the typical data word for that processor. Resolution is the smallest nonzero number representable.

*Retiming* is the action of redistributing combinational time delays across flip flop boundaries to attempt balancing the clock cycle time, for performance. It is usually performed by synthesis tools.

*RocketIO* Serial Transceivers are multi-gigabit serial transceivers developed by the team formerly known as “Rocket Chips”, a company purchased by Xilinx.

*Router* is design software that does routing, or connecting. The term is also used for data communication applications that manage connections between various sites. Routing is the action of defining which physical wire segments in the FPGA that interconnect logic cells.

*Route-through* is the action of making a connection right through a CLB or slice, versus going through the standard interconnect paths. In a congested design, for non-timing critical paths, route-through paths are commonly used.

*RTL* Register Transfer Language is a sub-dialect of an HDL, where low level structure may be identified. It is the least abstract level, and typically closest to the target architectures physical facilities. Although less compact than more abstract code, RTL offers greater user control over final results. The added control may be at the expense of transportability.

*Run Length* refers to the number of consecutive logic ones or zeroes that can exist in a run length limited code. 8b10b coding has a specific run length, that permits clock extraction from an encoded data stream.

*Sample rate* The speed at which data samples are collected is the sample rate. Typically, the standard DSP limit is the Nyquist sampling rate, which is twice the frequency of the fastest signal being sampled, for faithful reconstruction. Sampling faster is called over sampling, and sampling slower is called under sampling. A side effect of under sampling is aliasing, which restricts faithful reconstruction of the original signal from the sampled data.

*Sampling* is the action of collecting samples of a signal. Typically, it involves an A/D converter, operating at the Nyquist sampling rate, or faster. It can also include sampling digital signals, such as the 16X sampling of data in a UART, which permits data transmission without a clock being explicitly sent.

*SDH* See SONET

*SDRAM* Synchronous Dynamic Random Access Memory is a clocked version of DRAM.

*Security bits* can be context dependent, for its meaning. In basic security, they are typically read and write protect bits, inhibiting access to the payload – typically configuration data. The idea can extend to including cryptographic keys or initialization vectors, for a crypto algorithm.

*Segmented Interconnect* is Xilinx renowned interconnect structure permitting networks to be incrementally constructed from smaller segments.

*SERDES* is the contraction for Serializer/de-serializer.

*Setup time* is the standard timing parameter for flip flops whereby input data must remain stable for a period of time prior to the clocking event, to always successfully switch. Failure to meet setup time for all the flip flops attached to a clock can result in unreliable switching. See also, metastability.

*Simplex* is simply one way communication.

*Simulated Annealing* is an algorithm for solving nonlinear systems that involves modeling the process of annealing a metal. It involves identifying energy levels, associating them to the system structure and invoking an annealing schedule, with specific rules for accepting or rejecting a particular outcome of the process. It was developed at Los Alamos during the World War II development of nuclear weapons. Although the early paper has multiple authors, it is generally attributed to Nicholas Metropolis. See also, quenching.

*Single Ended* is a type of simple signaling involving one wire and ground. This is opposed to differential signaling.

*S/I* Signal Integrity describes the environment in which the signals must exist. It covers the various techniques and design issues that ensure signals are undistorted and do not cause problems to themselves, to other components in the system, or to the other systems in proximity.

*SJ* Sinusoidal Jitter has a periodic form and is related to the data patterns in the system. Sinusoidal jitter is a component of deterministic jitter. Also referred to as deterministic jitter or DJ.

*SIMD* Single Instruction, Multiple Data defines a type of parallel processing whereby multiple operands are being calculated simultaneously by the same instruction. Some versions of “systolic processing” fit this model.

*Single* is the term for a short hop interconnect segment. Singles literally jump over one neighboring CLB to access the next physical layer of CLB choices. See direct, double, quad and hex.

*Skew* is the difference among the arrival time of bits transmitted at the same time. Bits can be elements of a parallel bus or members of a differential pair. A single clock can exhibit skew due to the delay encountered as the clock attaches to various targets.

*Skip Characters* is a special comma character (“K-character”) used by the transceiver to align the serial data on a byte/half-word boundary (depending on the protocol used), so that the serial data is correctly decoded into parallel data.

*Slack* is extra available time, beyond the requirement.

*Slave mode* is a configuration mode for an FPGA, that is not at the beginning of a chain of devices. It includes a specific set of signal responsibility actions different from “master mode”.

*Slice* is a term describing a specific portion of a CLB, at least one LUT and associated flip flop. The term varies among Virtex and Spartan families, depending on the family.

*SliceL* is a slice that uses a LUT that does not include memory or shifter capability. Spartan 3 was first to include this capability, which was included in Virtex 4 and Virtex 5, also. It recognizes that die area may be wasted by including full capabilities at every site, when it may not be used.

*SliceM* is the complement to SliceL, adding the SRAM capability to LUTs along with the bit shift option.

*SMT* Surface Mount Technology is a class of IC packaging technology where the chip is attached to the printed circuit surface, versus penetrating the surface with “through hole” technology.

*SNR* Signal to Noise Ratio is a measure of signal quality, usually measured in dB.

*SoC* System on a Chip is a common term for an integrated circuit made with multiple system level blocks – processors, RAMs, ALUs, etc. An SoC is the framework for building a system. The definition holds for both programmable and ASIC ICs.

*SOIC* Small Outline Integrated Circuit is a class of surface mount technology packaging that is particularly small. Chip Scale Packaging (CSP) fits this model well.

*Sonet* Synchronous optical network is a format allowing different types of formats to be transmitted on one line. SONET is a long term solution for a “mid-span-meet between vendors” approach. The other major advantage is that SONET allows ADDING and DROPPING signals with a single multiplexer. Known as SDH or Synchronous Digital Hierarchy Internationally.

*Source synchronous* Source synchronous is a general data transmission protocol frequently used in memories that simply means that the transmitter of data also provides the clock for the same data. This is particularly important when different data is associated with each clock edge. In bidirectional transmission, the clock sourcing is based on which end is sourcing the data, so it trades off during transactions.

*SPICE* is an industry standard electronic simulation software package capable of very accurate transistor/circuit level simulation over a wide range of voltage and temperature environments. It was originally developed by UC Berkeley personnel.

*Sparse chevron* is a Xilinx term for the arrangement of tessellated groups of I/O pins with associated internal and I/O power pins and grounds. The arrangement is particularly effective to improve signal integrity in the ASMBL architectures in Virtex 4 and later devices.

*Spartan* is the cost reduced version of Xilinx FPGA device architectures. The reduction is done by streamlining the included blocks and the fabric. Versions of Spartan 3 are not discussed in this text. Spartan was replaced by Artix at 28nm. Spartan has been resurrected, and has devices planned for in future families to address the low cost market opportunities.

*SPROM* Serial Programmable Read Only Memory is the most common device for holding an FPGA configuration bitstream.

*SRAM* Static Random Access Memory is a RAM that holds its binary contents as long as the power remains applied. It needs no additional control circuitry to periodically refresh its contents. Typically high speed, what it gains in simplicity, it loses in density over Dynamic RAM products. See BRAM.

*SRL 16* is the sixteen bit shift capability of some Virtex family LUTs. SRL32 is present in LUT6 devices.

*SSO* Simultaneously Switching Outputs is one form of system noise attributed to multiple signals switching at the same time. There is some latitude in the term “same time”, as multiple signals switching within a short time of each other tend to contribute to an ensemble average of noise that dramatically reduces by simply spreading their switching events over time.

*SSTL* Stub Series Terminated Logic is a logic standard.

*Stack* Also known as LIFOs (Last In, First Out), stacks are data structures that can be built from hardware using BRAMs and pointers. Stacks are uncommon for use as hardware solutions, but can be useful as peripherals to microprocessors that can operate with an external data stack.

*Stack Pointer* is a counter and state machine that tracks the position of data in a RAM implemented FIFO. The term can also be generated in a software model, but in hardware, it includes all the control variables capable of managing the stack, which can be complex. FIFOs are discussed in detail in several places in this text.

*Standard Cell* is a semi-custom or ASIC technology for users to create designs from a library of cell primitives that have a uniform height, and all interconnect between cells occurs in (typically) horizontal routing channels. Most hardened (fixed in silicon) tiles in Virtex have standard cell content (BRAM, FIFO, MGT, CMT, etc.). This approach is closer to full custom than gate arrays, where fixed foundations containing columns of transistors are simply modified by metal. Standard cell designs have a common design flow with capture, layout, back annotated simulation, and so forth.

*Static current* is the component of device current that flows into or out of the device, when it is powered up, but no inputs are switching. It is primarily attributed to a phenomenon of transistors termed leakage. See also dynamic current.

*Stripline* is a high speed printed circuit layout discipline.

*STS* Synchronous Transport Signal is the name of basic signals in the SONET specification. STS-1 is the lowest component, operating at 51.84 Mbps.

*Switchbox* is a block that connects a set of inputs to a set of outputs. Connections are typically made with a configuration bitstream that enables/disables connections among the various I/O lines. These may be implemented in various ways, PIPs, muxes, etc.

*Symmetric Rounding* is a style of rounding which rounds “away from zero”. It is frequently chosen to reduce a type of bias that other rounding algorithms (say, truncation) may introduce. It is one of several options with the DSP48 module.

*Synthesis* is a process starting from a high level of logic abstraction and automatically creating a lower level of logic abstraction using a library of available primitives.

*Syntonus* is defined as many clocks running at nearly the same frequency, but not in phase. The “same” frequency must be specified to some accuracy.

*Sysmon* System Monitor analog to digital converter.

*SystemACE* System Advanced Configuration Environment consists of two components: ACE Flash memory module and ACE Controller chip. System ACE is a space-efficient, pre-engineered, high-density configuration bitstream manager for multi-FPGA systems. The system can be upgraded or debugged by either exchanging the ACE Flash module or reprogramming the module in-system. Replaced by a small Zynq device after Virtex 6.

*System energy* In the placement problem, positions of logic cells are modeled as having energy. The sum of all such cell energies is the system energy. The usual goal is to modify the placement to reduce the system energy.

*System gates* is a capacity counting measure that converts all resources into two input NAND gate equivalents. For instance, flip flops are usually tallied at six, two input NAND gates each, as are SRAM bits. Blocks like multipliers and adders are all converted into their equivalent logic capacity, to give an idea of what gate array capacity would be needed to make the equivalent function. There is wide variation among the various FPGA vendors with regard to gate counting. With Zynq, the capacity of the FPGA device became accepted (so many system gates is equal to so many CLB's).

*System Generator* Also known as “System Generator for DSP” is a toolset chain combining Core Generator and MatLab tool flows.

*TCL* Tool Command Language is a script oriented utility many software developers prefer.

*TCP* Transmission Control Protocol is one of the main protocols in TCP/IP networks. IP protocol deals only with packets. TCP allows two hosts to establish a connection and exchange streams of data. TCP guarantees that delivery of data and packets will be delivered in the same order in which they were sent.

*Technology Mapping* is the general action of translating abstract logic description into a specific technology. For instance, technology mapping of a logical four input AND gate would map to a four input NAND followed by an inverter, for a CMOS gate array. The same function would map into a single four input LUT for most Virtex FPGA devices.

### *TEMAC* Tri-mode Ethernet Media Access Controller

*Termination* is correctly assigning matching impedance to source and receivers to properly condition a signal. In many cases, this is to reduce reflections. Some cases, such as PCI, encourage reflections, so it is important to know the requirement. In data communication, termination would be ending transmission. If abnormal termination occurs, retransmission is usually required. Normal termination typically includes data integrity checking and protocol shutdown.

*Throughput* is usually a measure of successful data transmission, typically described in bits per second. Data communications channels and memories are often described in terms of throughput. In this sense, the measure is for the data payload, versus the payload plus error correction bits. The correction bits are part of the overhead required for successful transmission.

*Tile* A repeated identical circuit is a tile. An FPGA tile is usually a CLB and an associated interconnect section that can be stepped and repeated into a silicon mask to build a family member. Different family members have varying numbers of tiles.

*Tilo* is the logic propagation delay of a LUT. Think of it as the time delay from input through to logical output.

*TLB* Translation Look-aside Buffers are typically built from either RAMs or CAMs depending on the task required. In generally, they operate with cache memories, and their task is to quickly map an address to another address, or simply determine that an addressed variable is present in memory. The action often occurs in parallel with access to the target memory, occurring as the binary value passes by – hence the term “lookaside”.

$T_{PD}$  Propagation Delay Time is the time it takes for a switching input to arrive successfully at an output. Often, it is used to describe the speed of a simple gate, but can also describe larger structures like decoders or multiplexers. The access time from presentation of an address to arrival of the data on a RAM can be viewed as a  $T_{PD}$ , also. The LUT timing parameter Tilo qualifies as a  $T_{PD}$ . We use the term “successfully” to emphasize that the stable output for a stable input is the desired time, recognizing that early arriving glitches are not the desired output, in most cases.

*TQFP* Thin Quad Flat Pack is an inexpensive type of surface mount package.

*Travelling salesman* is a mathematical optimization process for producing the most efficient path. See also assignment problem and delivery truck problem.

*Tri-Mode Ethernet MAC* is an ASIC block designed to work with or without a microprocessor to implement one of three different Ethernet speeds of media access control. Also known as TEMAC.

*Triple Oxide* is the approach taken in Virtex 4 and later Virtex devices to optimize power and performance. This method uses three oxide thicknesses in the process, so high speed paths use thinner oxides, while slower portions use thicker oxides. Oxide thickness is a factor in leakage current, so only using thin oxide where speed is crucial is part of the approach.

### TSOP Thin Small Outline Package

*UART* Universal Asynchronous Receiver Transmitter describes a broad number of serial to/from parallel solutions that are used in communications paths. Variations include 2, 4, 8 and 16X clock sampling on data (due to asynchronous switching), and may be over eight or sixteen bit words. Odd or even parity and delimiting “start” and “stop” bits are common options.

*UCF* User Constraint File is a text file with a specific syntax to describe positional and time constraints for a Xilinx design.

*UI* Unit Interval is a unit of time corresponding to one bit period. A unit interval is the time it takes to send one bit.

*Universal Logic Function* is the capability of some programmable logic functions to become any function of its binary inputs. LUTs are universal logic functions, which is a superset of being functionally complete.

*USART* Universal Synchronous Receiver Transceiver should be USRT, but common usage simply modifies the relation to UART. USARTs differ from UARTs by including a clock. This also eliminates needing to overclock the data for multiple samples per bit, reducing complexity.

*USB* Universal Serial Bus is an external peripheral interface standard for Plug-and-Play communication between a computer and external peripherals over a cable using bi-directional serial transmission at and above speeds of 12 Mbps. Many versions exist, some with improved performance.

*Utilization* In FPGA devices, utilization describes the percentage of used resources relative to the number of available resources. Hence, utilization is a number with many dimensions. Using half the I/O pins and a quarter of the logic is not using half the chip! All resources must be appropriately factored into the net utilization to get an accurate picture of how much available capacity remains. See also, capacity.

*UTOP/A* Universal Test and Operation PHY-Interface for ATM Verilog is a Hardware Description Language (HDL) for electronic design and gate-level simulation.

*VersaRing* is a term describing the I/O to fabric interconnect region originally used in the XC5200 family and initially used in the Virtex family of FPGA devices.

*Virtex* is Xilinx several FPGA families that first supplemented programmable fabric with special, flexible blocks like BRAMs, multipliers and MGTs, while substantially adding more I/O standards. Virtex UltraScale Plus at the 16 nanometer node is the latest, as of this revision of this text.

### VQFP Very Small Quad Flat Package

*VHDL* is compound acronym, VHSIC Hardware Definition Language. VHDL was originally developed by the military to provide a common language to communicate hardware functionality, modeling and simulation. VHDL is one of the two most common HDLs available, with Verilog being the other. See below.

*VHSIC* Very High Scale Integrated Circuit is the military acronym for large integrated circuits.

*Via* are the paths between layers of either printed circuit boards (PCB) or integrated circuits. Typically, vias attach two or more layers of metal. Vias that do not arrive at the surface of either a chip or a PCB are called “blind” vias. Although necessary for most designs, vias are detrimental to signal integrity, in general.

### VLSI Very Large Scale Integration

*VOIP* Voice Over Internet Protocol is the ability to communicate telephone like transmission over the internet.

*VT* is the symbolic reference to a transistor threshold voltage, the minimum required voltage to activate the device.

*Weighted Bipartite Matching* is another term for the assignment problem.

*WebPACK ISE* is the version of ISE which is available off the Xilinx website, for free. There is a WebPack version now for Vivado to support the smaller devices commonly used by students, hobbyists, and other customers.

*World View* was a graphical image of an FPGA produced by the FPGA Editor. Replaced by the floor plan view in Vivado.

*Write back* is a cache memory “write policy” where data written into the cache is not written into main memory until that data line in the cache is replaced.

*Write through* is a cache memory “write policy” where data written into the cache memory is also written into main memory to assure coherency (i.e. always matching).

*XACT* is the original name of Xilinx FPGA design software.

*XACTStep* is a combination of XACT and NeoCAD’s Foundry product.

*XCITE* Xilinx Controlled Impedance TEchnolgy is built in digitally controlled impedance (DCI) matching of all single-ended I/Os for signal integrity. XCITE eliminates the need for external termination resistors on all selected I/Os by incorporating adaptive series and parallel termination impedance on the FPGA itself.

*Xesium Clocking* is Virtex 4 differential internal clocking structures.

*XST* Xilinx Synthesis Technology is a Xilinx design software tool that supports VHDL, Verilog and mixed synthesis in ISE. Vivado replaced XST with an all new synthesis implantation.

*XtremeDSP* slice is a marketing term for the DSP48 block.

*ZBT RAM* Zero Bus Turnaround RAM is a synchronous RAM architecture optimized for networking and telecommunications.

Back Inside Cover

Back Cover