

To the Graduate Council:

I am submitting herewith a master thesis written by Zijun Shen entitled "A Development Environment for Reconfigurable Computing ." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Donald W. Bouldin

Prof. Donald W. Bouldin, Major Professor

We have read this thesis
and recommend its acceptance:

Robertz Badurkheim

Paul R. J.

Accepted for the Council:

Cawminkel

Associate Vice Chancellor
and Dean of The Graduate School

**A DEVELOPMENT ENVIRONMENT FOR
RECONFIGURABLE COMPUTING**

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Zijun Shen

December 1996

ACKNOWLEDGEMENT

I would like to thank my advisor Dr. Donald W. Bouldin for introducing me to VLSI design theory and his very helpful insights and suggestions during the course of this project. Thanks also goes to my committee members, Dr. Robert E. Bodenheimer and Dr. Paul B. Crilly for their help during the development of this thesis. I am grateful to the College of Engineering, University of Tennessee, for providing financial support during my graduate studies. Acknowledgements are due to my supervisor at CE, Dr. Denise Ford Jackson, for her support and understanding. Further, thanks to Senthilvelu Natarajan, Amiya Anand Chokhavalala, Gentry Jackson, and Dan Marzahl for providing test cases for this thesis. Finally, I would greatly thank my wife Min Wu, my parents, and my brothers, for their encouragement, understanding, and patience.

ABSTRACT

Reconfigurable Computing is a cutting-edge technology used in VLSI and FPGA design and it refers to a system that contains hardware whose functionality can be changed many times. In this thesis, the author establishes a complete set of design paths to do reconfigurable computing using the Engineers' Virtual Computer (EVC); proves established methods; and develops both hardware and software designs to evaluate differences in execution time between them. This is done by running hardware designs on the EVC and simultaneously executing software equivalent designs on a Sun Sparc 10 station. The hardware and software designs are developed using VHDL, a hardware description language, and the programming language C, respectively. By using COSSAP, a DSP CAD toolset, VHDL and C code are generated from the application's block diagram level. The designs are then synthesized using Synopsys FPGA Compiler with Xilinx 4000 FPGA technology. Xilinx XACT tools are used to perform the physical place and route for a Xilinx 4010 device. The result quantitatively determines that reconfigurable hardware using FPGAs is capable of providing a significant performance improvement over the Sun Sparc 10 processor for DSP applications.

TABLE OF CONTENTS

CHAPTER	PAGE
1. Introduction	1
1.1 Motivation	1
1.2 Objectives	5
1.3 Chapter Contents	7
2. Background	8
2.1 Reconfigurable Logic	8
2.2 SRAM Based FPGA and Dynamically Reconfigurable Logic . . .	9
2.3 Reconfigurable Computing	14
2.4 Engineers' Virtual Computer	16
3. Design Methodology and Tutorial	20
3.1 Design Flow Chart	21
3.2 Tutorial: Using Synopsys and the Engineer's Virtual Computer .	24

CHAPTER	PAGE
3.2.1 Example	24
3.2.2 Using the EVC as Reconfigurable Computing System	28
4. Test Cases For Reconfigurable Computing and Results	31
4.1 Test Case I: Hamming Encoder	32
4.2 Test case II: FIR Filter	42
4.3 Some more Test Cases	54
4.4 Results And Discussions	57
5. SUMMARY, CONCLUSION AND FUTURE WORK	61
5.1 Summary	61
5.2 Thesis Conclusions	62
5.2.1 Conclusions	62
5.2.2 Suggestions	64
5.3 Future Work	65
BIBLIOGRAPHY	67

CHAPTER	PAGE
APPENDIX	69
A. Using Viewlogic and the Engineer's Virtual Computer	70
B. Synthesizing a Xilinx FPGA Using the Synopsys FPGA Com- piler	75
C. VHDL and C Template Files of Using the EVC	81
VITA	92

LIST OF TABLES

TABLE	PAGE
4.1 <i>Performance of Pseudo-Random Number Generator</i>	55
4.2 <i>Performance of Priority Encoder</i>	56
4.3 <i>Performance of Right Shifter</i>	57
4.4 <i>Performance of Multiple Function Comparator</i>	58
4.5 <i>Speedup Results for All Test Cases</i>	59

LIST OF FIGURES

FIGURE	PAGE
1.1 <i>EVC Configuring Flow Chart.</i>	6
2.1 <i>Basic XC4000 Interconnection Block Diagram</i>	10
2.2 <i>The Xilinx Configurable Logic Block</i>	11
2.3 <i>The Operation Chart for EVC</i>	17
2.4 <i>SBus Slave Interface Schematic Symbol</i>	19
3.1 <i>Design Flow Chart For The EVC — Method I</i>	22
3.2 <i>Design Flow Chart For The EVC — Method II</i>	23
3.3 <i>Structural VHDL For Hamming Encoder</i>	29
4.1 <i>Modules Of Two Hamming Encoders And The EVC</i>	35
4.2 <i>Gate_Level Simulation For Two Parallel Hamming's Encoders</i>	35
4.3 <i>Top Level Schematic For Two Parallel Hamming Encoders</i>	36
4.4 <i>Top Level Schematic For One Hamming Encoder Algorithm</i>	37
4.5 <i>Gate Level Schematic For One Hamming Encoder Algorithm</i>	38

FIGURE	PAGE
4.6 <i>Gate Level Schematic for Two Parallel Hamming Encoders</i>	39
4.7 <i>Execution Time-Lines and Speedup.</i>	41
4.8 <i>Realization of FIR Filters</i>	42
4.9 <i>A Bit_Vector FIR Filter of Third Order</i>	43
4.10 <i>3rd Order FIR Filter</i>	44
4.11 <i>A Single Tap of A Bit_Vector FIR Filter</i>	45
4.12 <i>3rd Order FIR Filter VHDL Testbench Use by SDS And VSS . . .</i>	46
4.13 <i>Simulation Of 3rd Order FIR Filter By Using SDS And VSS . .</i>	47
4.14 <i>Top Level Schematic For FIR Filter</i>	48
4.15 <i>Schematic Of SBus Interface For FIR Filter</i>	49
4.16 <i>Top Level Module Of FIR</i>	50
4.17 <i>Sub-Level Module of FIR Module</i>	50
4.18 <i>Top Level Module of A Single Tap of a Bit_vector FIR filter . . .</i>	51

CHAPTER 1

Introduction

Reconfigurable computing is an area that is gaining popularity and general interest recently. The advantage of reconfigurable computing is the effective “hardwiring” of software and the potential dramatic performance gains due to this technology. This thesis demonstrates methods for programming and using a reconfigurable computing system. Some DSP algorithms such as a Finite Impulse Response (FIR) filter, two parallel Hamming Encoders, and several logic operators will be constructed and tested. This thesis includes the performance results of an Engineer’s Virtual Computer (EVC), a reconfigurable computing system, in a SUN Sparc 10 Workstation.

1.1 Motivation

Logic that is one-time configurable is obviously limited. One reason is because everything that the product can do is designed in its base configuration. The other reason is because existing logic may not meet the application’s requirement soon after being delivered to the customers.

Much effort has gone into the development of specialized digital hardware systems that allow the efficient implementation of digital signal processing algorithms and logic operations because of their widespread applications. These specialized hardware systems seek to meet the stringent computational and I/O demands that are often placed on hardware by DSP algorithms. Typically, these specialized hardware systems have taken the form of either a programmable processor with specialized functional units or a single application design with limited or no programmability. The purpose of the first approach is to provide application flexibility while still achieving a modest level of performance. The goal of the second approach is to achieve the highest performance level possible by sacrificing flexibility. These two approaches represent the extremes of implementation possibilities and conflicting goals. That is, system flexibility and maximum performance are generally not harmonious and represent a tradeoff that designers must make. What is needed is an approach that is able to combine the desirable characteristic of system flexibility with the equally important goal of achieving a high level of system performance. Such an approach can be obtained by using reconfigurable hardware.

Reconfigurable hardware refers to the use of any electronic hardware system that can be changed in structure either statically (between applications) or dynamically (during an application) without the addition of physical hardware elements. It can be implemented using any technology that allows the structure of the hardware to change using only electrical signals. Early examples of recon-

figurable hardware can be seen in the application of programmable logic devices (PLDs) and systems incorporating sets of fixed components whose interconnections can be changed by reconfiguring a crossbar or interconnection device.

In this thesis, reconfigurable hardware refers to the use of the Field Programmable Gate Arrays (FPGAs) as the basic hardware element to be reconfigured. FPGAs generally consist of sets of flexible gates, registers, and memories whose function and interconnections are controlled through the loading of SRAMs. A reconfigurable gate could, for example, be made to behave as either an OR or an AND gate depending upon the status of the SRAM configuration bits attached to the gate [14]. FPGAs are able to support both static and dynamic reconfigurations since all that is needed to change the hardware is to reload the controlling SRAMs with the appropriate configuration bits.

The advantage of the reconfigurable hardware approach for digital signal processing system implementation is that application flexibility is provided by allowing the actual hardware design to change rather than by adding software programmable hardware to a design. That is, to change the application implemented by a reconfigurable hardware system, the system need only be reconfigured through the reloading of the configuration SRAM bits. This allows a completely new hardware design to be implemented. No external hardware needs to be added to the initial design to provide this flexibility. The flexibility is provided by the reconfigurability of the FPGA itself and is thus utilized by changing the

entire design at the basic hardware level. Thus, the designer need not consider the programmable hardware elements and can hence concentrate on an optimized hardware design. This is an advantage since many designs suffer in performance when extra flexibility is achieved through the addition of software-programmable circuitry. The addition of software-programmable circuitry requires extra area on the chip and often increases the critical path of the design. With reconfigurable hardware, no such penalty applies and hence greater performance can be achieved. On this basis, reconfigurable hardware has been selected as a potentially attractive method of implementing digital signal processing systems.

We define reconfigurable computing systems as those machines that use the reconfigurable hardware[9], or more precisely, machines that use the reconfigurable aspects of Field Programmable Gate Arrays (FPGAs) to implement an algorithm. Researchers throughout the world have shown that computationally intensive software algorithms can be transposed directly into hardware design for extreme performance gain.

In the context of this thesis, the development environment of reconfigurable computing or reconfigurable hardware refers to the Engineers' Virtual Computer or EVC, which consists of Field-Programmable Gate Arrays (FPGAs). FPGAs can be configured in only nanoseconds, and a product's function can be customized infinite times by configuring FPGAs. The basic idea for configuring hardware objects, algorithms implemented as dynamically downloadable hardware designs, is

shown in Figure 1.1 [11]. From Figure 1.1, the EVC is a typical environment for reconfigurable computing. The objective of this thesis is to develop this reconfigurable computing environment. Specifically, this thesis seeks to answer the question of whether or not EVCs are a useful and justifiable alternative for implementing digital signal processing systems. Do they provide a significant performance increase over software-programmable-based methods (in specific, the C code software_only solution on SUN Sparc 10 workstation) while still maintaining the advantage of flexibility? This thesis will show that they do.

1.2 Objectives

There are multiple objectives of this thesis. First, it is to provide the environment and methods that can do reconfigurable computing efficiently. Second, it establishes design methodologies for doing reconfigurable computing and proves the correctness of these design paths. Third, it will show what kinds of algorithms should and should not be configured into firmware. Finally, differences in execution time for specific algorithms will be given based on the EVC's and Sun Sparc-10 workstation's performances.

EVC Configuring Flow Chart

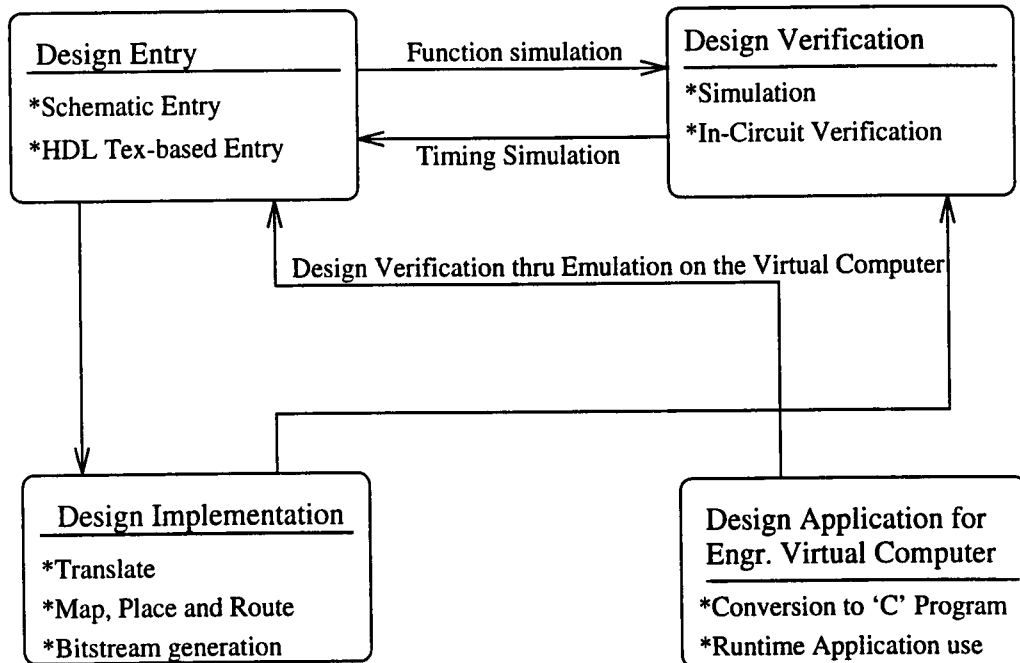


Figure 1.1: *EVC Configuring Flow Chart.*

1.3 Chapter Contents

This thesis is organized as follows: Chapter Two will present more background on the nature of reconfigurable computing and the EVC. Chapter Three will then illustrate the design flow for hardware and software designs. Also, several methods for doing reconfigurable computing will be given. Chapter Four deals with example DSP applications and their performance results when implemented on the EVC in comparison to their software equivalent implementations. These applications are used to show the potential gains of using reconfigurable hardware in DSP applications. Chapter Four also indicates the design methods of Chapter Three are valid. Finally, Chapter Five will discuss conclusions and further work to be done in this area.

CHAPTER 2

Background

In this chapter, some basic concepts and terminologies closely related to reconfigurable computing and the Engineers' Virtual Computer are discussed. The concepts introduced in this chapter will be used in later chapters.

2.1 Reconfigurable Logic

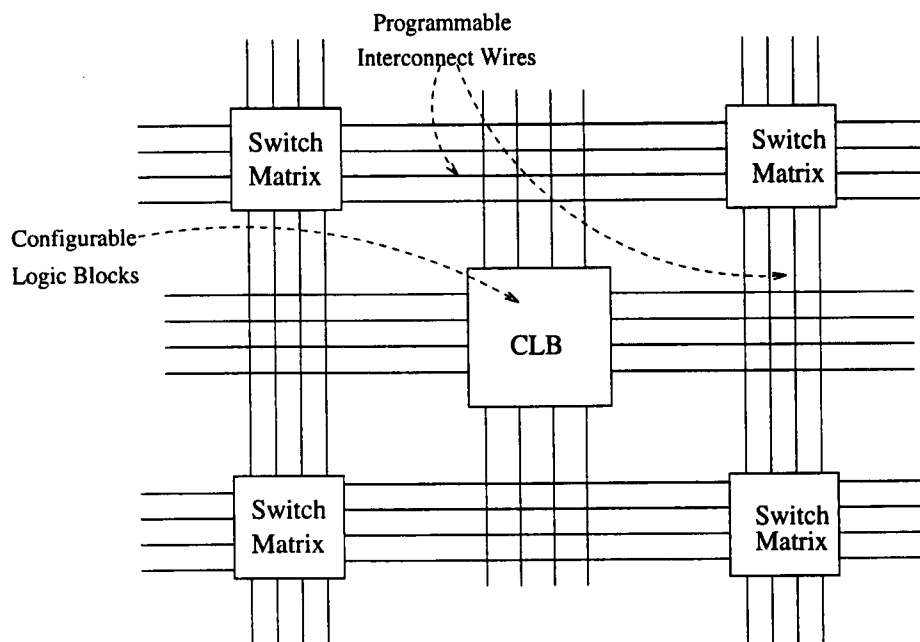
Reconfigurable logic refers to a product whose function may be customized many times [3]. Logic that is configurable is obviously limited because its base configuration has to encompass all possible variants. One technique for producing a product whose function may be extended beyond its original design objectives is to base that product on devices that can be reprogrammed. For instance, a PLA (Programmable Logic Array) device could be used to implement a state machine that performs a certain sequence of operations. Based on the inputs presented to it, a PAL device could also be used to implement Boolean equations and to generate appropriate outputs. In all of these cases, the functions of the truth

table, state machine, or Boolean equations can be modified by simply exchanging the programmable device with an upgraded version.

Another technique would be to use non-volatile memory devices to store firmware programs for use by a microprocessor or a microcontroller. The non-volatile memory could use a PROM, EPROM, EEPROM, or FLASH devices. Similarly, programmable logic devices can employ PLD, EPLD, EEPLD, or FLASH-PLD components. In all of these cases, from a board-level perspective, the board itself would be classified as reconfigurable. However, from a device-level viewpoint, PROMs would fall into the category of “configurable”, while more sophisticated devices, EPROMs, EEPROMs, and FLASHs, would be categorized as “reconfigurable” (similarly is the case of PLDs versus EPLDs, EEPLDs, and FLASH-PLDs). Additionally, EE-based and FLASH-based components may be referred to as in-system programmable (ISP), because they can be reprogrammed while remaining resident on the circuit board [7].

2.2 SRAM Based FPGA and Dynamically Reconfigurable Logic

A very basic definition of a digital FPGA (Field Programmable Logic Array) would be an integrated circuit consisting of a set of gates such as ANDs, ORs, XORs, etc., latches, and registers (D-flipflops), whose interconnection and function can be altered through the reprogramming of configuration bits in the



Note: This figure is adapted from figure 9 of Xilinx 1994 Programmable Logic Data Book

Figure 2.1: *Basic XC4000 Interconnection Block Diagram*

FPGA. An example of reconfigurable interconnection is shown in the Figure 2.1

Figure 2.2 [14] shows the internal structure of a CLB (Configurable Logic Block), which provides the programmable combinatorial logic capabilities of the Xilinx chip. Each configurable logic block provides a combinatorial logic section, two flip-flops, and an internal control section. CLBs communicate with other Xilinx resources by utilizing five logical inputs (.a, .b, .c, .d, and .e), a common clock input (.k), an asynchronous direct reset input (.rd), an enable clock (.ec), a direct data input (.di), and two outputs (.x and .y). All inputs may be driven from interconnect resources adjacent to the blocks, and both outputs may drive

these same interconnect resources [14].

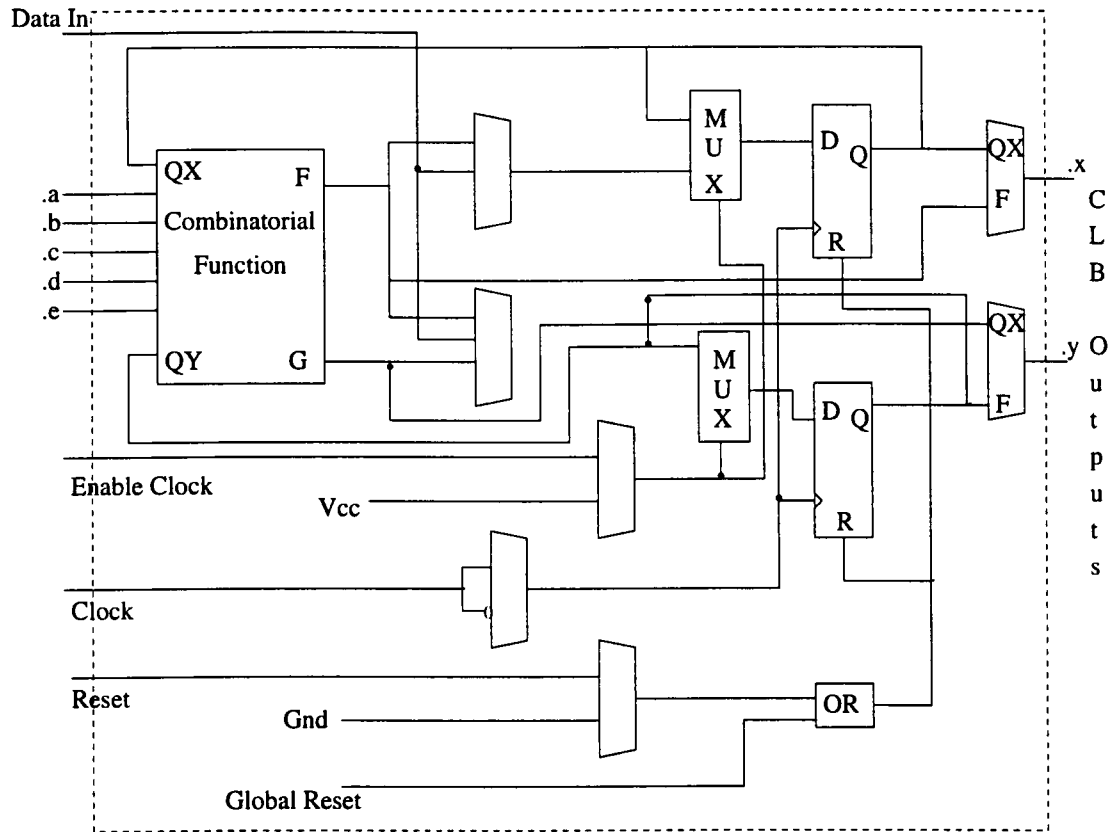


Figure 2.2: *The Xilinx Configurable Logic Block*

Internal to the CLB, data input for either flip-flop may be provided from the F or G combinatorial outputs or the direct block input. Both CLB flip-flops share the reset and clock signals. Flip-flops are reset by a low input on the reset input, and may latch data either on the positive or negative clock edge as selected

by the designer. A low input on the clock enable inhibits the latching of data into the flip-flops regardless of the clock input [14].

Implementation of boolean functions within the combinatorial logic portion of the CLB is accomplished through a 32 by 1 look-up table. Selected inputs to the CLB provide the look-up address. This technique allows the generation of two independent logic functions of up to 4 variables, or a single function of up to 5 variables within one CLB. Some functions of up to 7 variables are possible. Inputs to the combinatorial function may be provided through direct inputs to the CLB or the outputs of the flip-flops. Outputs of the combinatorial function may either provide direct CLB outputs, or inputs to the CLB flip-flops.

FPGAs perform the function of a custom circuit, like a gate array, and are user programmable [1]. They are currently being used as glue logic, for test and verification logic in system designs, for adaptable system designs and more recently as co-processing devices. FPGAs are also used to emulate other component architectures, and are applicable for rapid prototyping [12] [13]. In the context of this thesis, Xilinx XC4010 FPGA is used as co-processing device and could be used as fast rapid prototyping. XC4010 FPGA belongs to the catalog of SRAM based FPGAs.

An SRAM-programmable FPGA is programmed by loading configuration memory cells from an external source. The configuration memory cells control the logic and interconnections that perform the application function of the FPGA.

There is no separate RAM area on the chip, the memory cells are distributed among the logic they control.

The advent of SRAM-based FPGAs in the late 1980s and early 1990s presented a new capability to the electronics fraternity: dynamically reconfigurable logic, which means designs can be reconfigured “on-the-fly” [3]. FPGAs contain a large number of diverse logic gates and registers, which can be connected together in widely different ways to achieve a desired function. SRAM-based variants augment the capabilities of standard FPGAs by allowing new configuration data to be down-loaded into the device by the main system in a fraction of a second. In the case of these devices, a few of the external pins are dedicated to the task of loading the data, including an enable control and clock and data inputs. When the enable control is placed in its active state, edges on the clock are used to load the device’s SRAM with a stream of 0s and 1s which are presented to the serial data input.

The versatility of these devices opens the floodgates to a wealth of possibilities [8]. All these devices are circuit boards whose interfaces can be configured to meet a variety of communications protocols or act as digital signal processors (DSPs) that can be modified to process data in a variety of ways. As alternative protocols become available or improved algorithms are invented, the patterns that are used to configure the FPGAs can be modified to take full advantage of these new developments.

However, the true power of these devices, which are referred to as in-circuit reconfigurable (ICR), resides in their ability to be reconfigured on-the-fly. For example, when a system is first turned on, it might configure all of the FPGAs on a circuit board to perform diagnostic functions, both on themselves and on the circuit board. After the diagnostic checks have been completed, the system can dynamically reconfigure the FPGAs to fulfill the main function of the design.

2.3 Reconfigurable Computing

Reconfigurable computing is a relatively new term that can be taken to refer to a variety of diverse aspects of dynamically reconfigurable logic. However, for the most part, reconfigurable computing is generally accepted to refer to a combination of hardware and software that allows a product to emulate different hardware architectures [3]. More specifically, reconfigurable computing describes computer and controller systems that can be reconfigured ‘on-the-fly’ to meet the needs of the target application. Applications range from ‘number crunching’ and computing algorithms (including DSP algorithms as shown in the following chapters) to user interfaces, to communication systems interfaces and protocols.

Reconfigurable computing technology has the ability to modify a computer system’s hardware architecture in real time [3]. Reconfigurable computing systems are computing platforms whose architecture can be modified by the software to

suit the application at hand. Within the application program, a software routine is written to download a digital design (chip design) directly into the RPU (Reconfigurable Processing Unit) of the reconfigurable computer. It is like having a custom processor chip manufactured for a specific application program. Most Reconfigurable Computing Systems are plug-in boards made for standard computers such as PCs and Workstations. The Reconfigurable board acts as a co-processor to the main MPU (Micro Processor Unit). Calculations normally done within the MPU are now carried out in the RPU instead [11] [9]. The particular FPGA used in Reconfigurable Computing is a SRAM-based FPGA. This class of FPGAs allows one to reconfigure the inside over and over again.

More general Reconfigurable Computers are used either for computations as attached processors or co-processors. Currently, a number of reconfigurable computing plug-in boards (for PC and workstations) are available. These devices act as co-processors to relieve computationally intensive algorithms from the CPU. As reconfigurable computing matures, aspects of current plug-in strategies will migrate directly to the motherboard of the computing system. The Engineers' Virtual Computer is one typical example of these devices.

2.4 Engineers' Virtual Computer

The EVC is one of an entire family of reconfigurable computing products created by the Virtual Computer Corporation (VCC). This product consists of a standard, single slot SBus card that can be plugged into any SBus compatible platform, including SUN Sparc stations. The card features one Xilinx reconfigurable logic device, interface and protocol logic, a VCC proprietary local interface for the addition of optional modules such as an available 2 MB SRAM module.

The active components of the EVC consist of a Xilinx 4000 series device along with a PAL, a EPROM, buffers (FCT245 and 244 components), a 74FCT621 and a high speed oscillator. The PAL contains the logic necessary to control I/O and to configure the Xilinx. The EPROM contains a simple boot program which initializes the board. The 74FCT621 is an open collector buffer and is used to set the interrupts. Xilinx 4000 series FPGAs are SRAM based FPGAs [11].

The EVC is a reconfigurable computer because it uses the reconfigurable aspects of FPGAs to implement an algorithm. The algorithms are partitioned into a sequence of hardware implementable subroutines. These subroutines represent the serial behavior of the algorithm and are executed sequentially.

The EVC SBus card is plugged in one of the SBus slots (slot 3) of Sun Sparc 10 workstation (Sun1). The SBus is the interface between the EVC and Sun1. Hardware designs are downloaded from Sun1 through the SBus into the EVC.

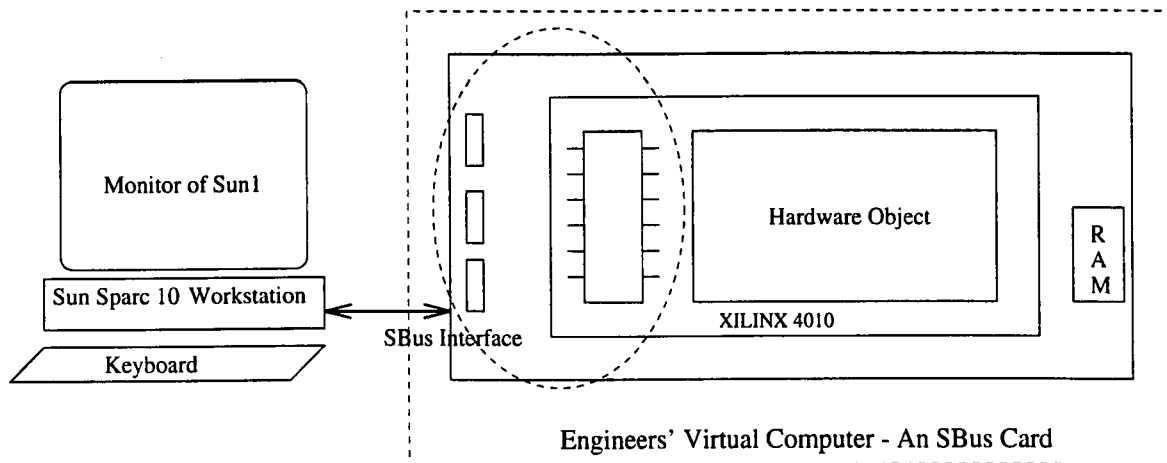


Figure 2.3: *The Operation Chart for EVC*

After the EVC gets the inputs data from Sun1, it will execute the logic functions and computations of the hardware objective design, and then write outputs back to Sun1 through the SBus. This is illustrated in Figure 2.3. Through this setup, the EVC uses the keyboard and monitor of Sun1 as input and output devices. This way, the EVC can work as an independent processor or as a co-processor of Sun1.

The theory of EVC operation is: On power-up, the EVC resets the Xilinx in a powered down state. The workstation looks at each SBus slot and verifies whether or not a board is present. When it looks at the slot the EVC is in, it will read from the EVC's EPROM. To read from the EPROM, the PAL decodes the byte address, enables the EPROM, enables the buffers and buffer direction, and places the data on the bus and performs a byte acknowledge.

Following power up, the Xilinx device can be reset at any time by reading a byte at address 0x8000 in the SBus address space of that SBus slot. The Xilinx can only be programmed after the part is reset (although the Xilinx is reset at boot time). Programming the Xilinx is done by the PAL when there is a byte write operation at address 0x8003 [11].

Once the Xilinx is configured, it is responsible for the SBus interface. After configuration, the PAL takes care of the buffer direction giving the correct direction by way of the SB_DTDIR signal on pin 4 of the Xilinx [11].

The SBus interface plays an important role in using the EVC. Figure 2.4 [11] shows a schematic symbol of the EVC SBus Slave Interface (slvif). A great more detail and examples using the EVC SBus Slave Interface for the hardware designs are discussed in the Appendix A.

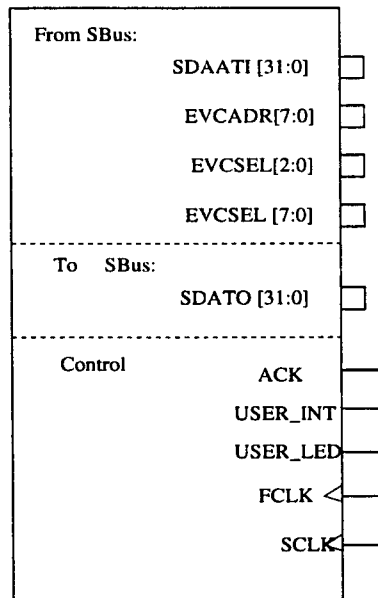


Figure 2.4: *SBus Slave Interface Schematic Symbol*

CHAPTER 3

Design Methodology and Tutorial

The EVC allows the engineer to download a design on an actual FPGA by providing a downloading process which uses 'C' code functions. In this way, we can download a design to the FPGA that is installed on the EVC card [11]. In this chapter, the design path for developing a reconfigurable computing system and for configuring the EVC are provided. By following these completed methods, many applications were implemented and tested successfully on the EVC. This chapter is organized as follows: First, two sets of complete design paths or methods to configure the EVC are established. Then the design implementation processes for one particular application are given. Finally, special requirements in using the EVC as a reconfigurable computing system are highlighted.

Some more applications are included in Appendix A to serve as examples to help the reader understand better the processes of using the EVC. A quick reference of using Synopsys FPGA Compiler can be found in Appendix B. Also, a complete set of template files for both structural VHDL and C code are provided in the Appendix C.

3.1 Design Flow Chart

In the design flow chart Figure 3.1, COSSAP is used as the design entry which allows designers to implement DSP application with the EVC at the block diagram or the system level. This established design path will be proven in the next few chapters.

The design flow for the example in this chapter follows the design flow chart in Figure 3.2. This design path is similar to the one in Figure 3.1 except that the VHDL code and the DSP code in this example are provided manually by the designers. In Figure 3.2, the application function is described using VHDL and then is synthesized using the Synopsys toolset into the Xilinx 4000 FPGA technology. Xilinx XACT tools are used to perform the physical place and route and obtain a configuration file for a Xilinx 4010 device. The file format is then converted so that it can be downloaded to the EVC. Software is then written in C to configure the EVC, download data to it, and retrieve data from it for display purposes. Software corresponding to the hardware function is also written, and their execution results are compared.

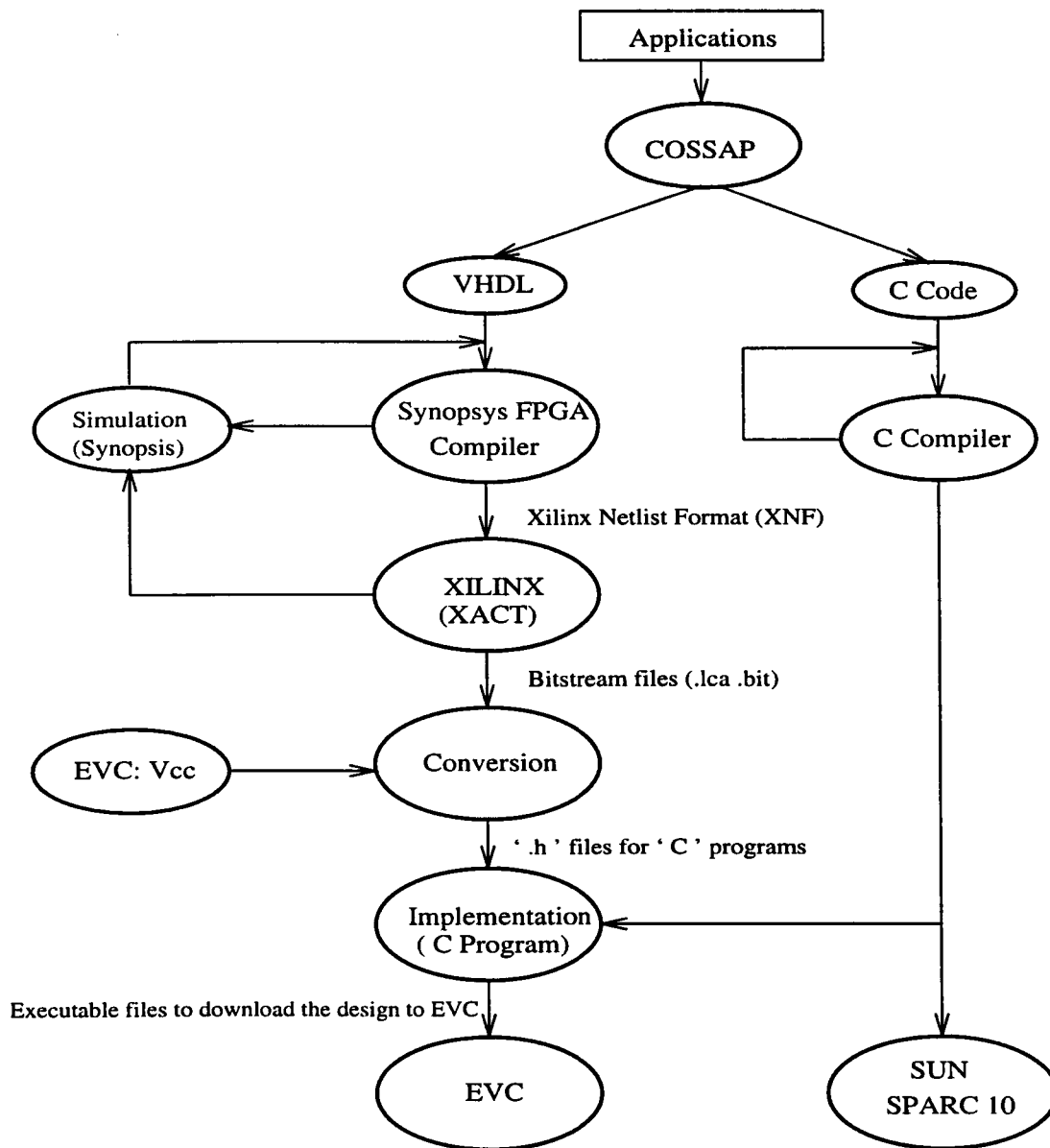


Figure 3.1: *Design Flow Chart For The EVC — Method I*

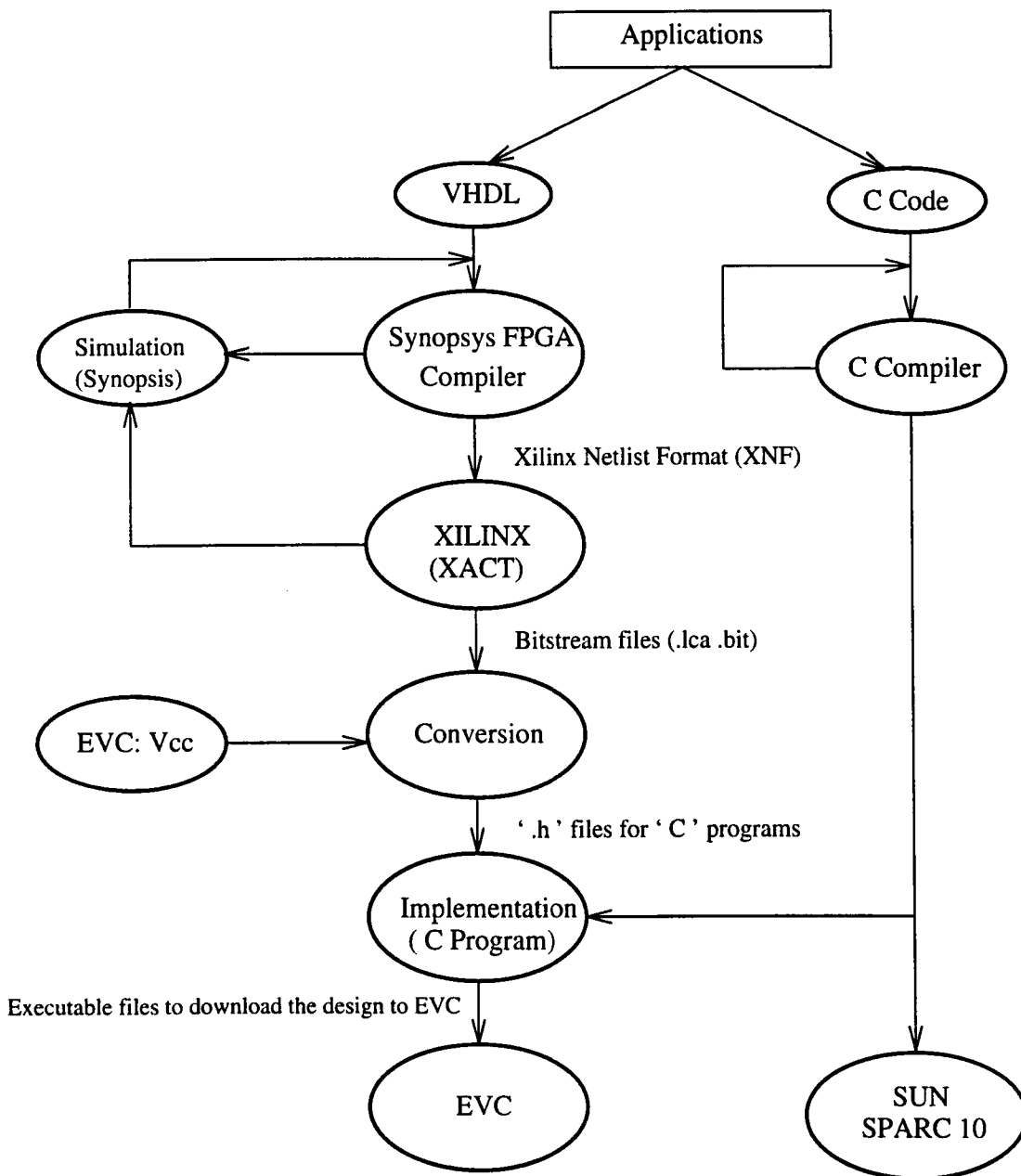


Figure 3.2: *Design Flow Chart For The EVC — Method II*

3.2 Tutorial: Using Synopsys and the Engineer's Virtual Computer

3.2.1 Example

In this section, a complete set of template files for both structural VHDL and C code are given as an example to use Synopsys and the EVC. By following the steps in this tutorial, one can obtain these files in his own directory and then he can run through the whole implementation processes. The included application design is used as a practical example which intends to show how to use the Synopsys FPGA compiler as a design tool for configuring the EVC. The example consists of two Hamming encoders working in parallel which return a string of ASCII characters. The characters will be printed on the Sun1 Sparc 10 workstation's monitor as alphanumeric, but the EVC sees them as 32-bit binary numbers. Each Hamming encoder has an 8-bit binary input to generate a 12-bit binary output. Thus, every group of four ASCII characters contains 0X0X so that the most significant bytes of the two 16-bits are zeros, and the inputs to be encoded will be the binary equivalent of the X's.

Set Up

It is recommended that the reader keeps the exact same directory structure as following.

1. `sun1:/u4/your_login_name % mkdir EVC`
2. `sun1:/u4/your_login_name % cd EVC`
3. `sun1:/u4/your_login_name/EVC % mkdir proj`
4. `sun1:/u4/your_login_name /EVC% cd proj`
5. Copy all the vhd files in this directory: `cp zijun/temp/proj/*.vhd .`
6. Copy the setup file for Synopsys: `cp zijun/temp/proj/.s* .`
7. Copy the script file: `cp zijun/temp/proj/encoder_2.scr`
8. Use an editor to edit the file: `.synopsys_dc.setup`
9. Change my login name 'zijun' to yours, and save the file.
10. List the files to make sure the following VHDL files are in your directory
`sun1:/u4/your_login_name/EVC/proj % ls`
`encoder_2.vhd   ham_code.vhd   sbus_if.vhd`
`evcregl.vhd   module.vhd   encoder_2.scr`
11. `sun1:/u4/your_login_name/EVC/proj % cd demo-13`
12. `sun1:/u4/your_login_name/EVC/demo-13% mkdir ham`
13. `sun1:/u4/your_login_name/EVC/demo-13% cd ham`
14. Copy the c compiler file: `cp zijun/temp/proj/makefile .`
15. Copy C code to your directory: `cp zijun/temp/proj/ham_encoder_2.c`
16. List the all files. `sun1:/u4/your_login_name/EVC/demo-13/ham% ls`
`ham_encoder_2.c   makefile`

Now, you have everything ready and it is recommended that you perform the rest of tutorial step by step.

Invoking The Design Analyzer And Analyzing The Design File

From the EVC/proj directory, run the Synopsys Design Analyzer in the background by entering the following command.

1. `sun1:/u4/your_login_name/EVC/proj % design_analyzer`
2. Select File —> Analyze... from the Design Analyzer menu.
3. Use the left mouse button to click once on File Format —> vhd.
4. Use the left mouse button to click once on evcreg1.vhd.
5. Click OK.
6. Click Cancel.
7. Select File —> Elaborate... from the Design Analyzer menu.
8. Use the left mouse button to click once on WORK and then click OK.
9. Use the left mouse button to click once on evcreg1(rtl).
10. Click OK.
11. Click Cancel.

Perform the same steps as above to the following modules in the following file order:

12. ham_code.vhd
13. module.vhd (Type a 'uniquify' command right after you finish elaborating this module. This could be done by selecting Setup —> Command Window, then, enter a 'uniquify' and return)
14. sbus_if.vhd

15. encoder_2.vhd

Running Script File

Use a script file to compile the design instead of using pull down menus, this will save time to compile the designs. Go to Setup in the Design Analyzer window, select the Execute Script..., then press encoder_2.scr in the File Name field, and then click 'OK'. This will take a while to compile all those modules and you may have to wait 5-10 minutes for this to complete. When a window pops up, click 'Cancel' to stay with the Design Analyzer.

Running XACT and EVC Conversion Function

1. sun1:/u4/user/EVC/proj % xmake -g -p 4010MQ208-5 encoder_2
2. sun1:/u4/user/EVC/proj % xmake encoder_2.mak

This will run all the way to generate a checkreg.bit file. You may have to wait 5-10 minutes for this to complete.

3. sun1:/u4/user/EVC/proj % makebits -b encoder_2

This will generate a rawbits file for the design.

4. sun1:/u4/user/EVC/proj % ../bin/r2h encoder_2

This will generate a header file for inclusion in the 'C' code. This header file is a representation of the logic design.

Running C Compiler

1. `sun1:/u4/your_login_name/EVC/proj%`

`cd /u4/your_login_name/EVC/demo-13/ham`

2. Copy the C header file into current directory:

`cp /u4/your_login_name/EVC/proj/encoder_2.h .`

3. `sun1:/u4/your_login_name/EVC/demo-13/ham% make`

This will compile the 'C' code into an executable that can be run. Remember that this executable can only be run on `sun1.engr.utk.edu`, but you can login from a remote machine and run it.

4. To execute this example, just type: `ham_encoder_2`

Now, you have finished all the operation steps.

3.2.2 Using the EVC as Reconfigurable Computing System

Hierarchy VHDL

The hierarchy order of the VHDL code used in this example is shown in Figure 3.3.

To implement a different design on the EVC, all the `Ham_Code*.vhd` files need to be replaced with the proper design.

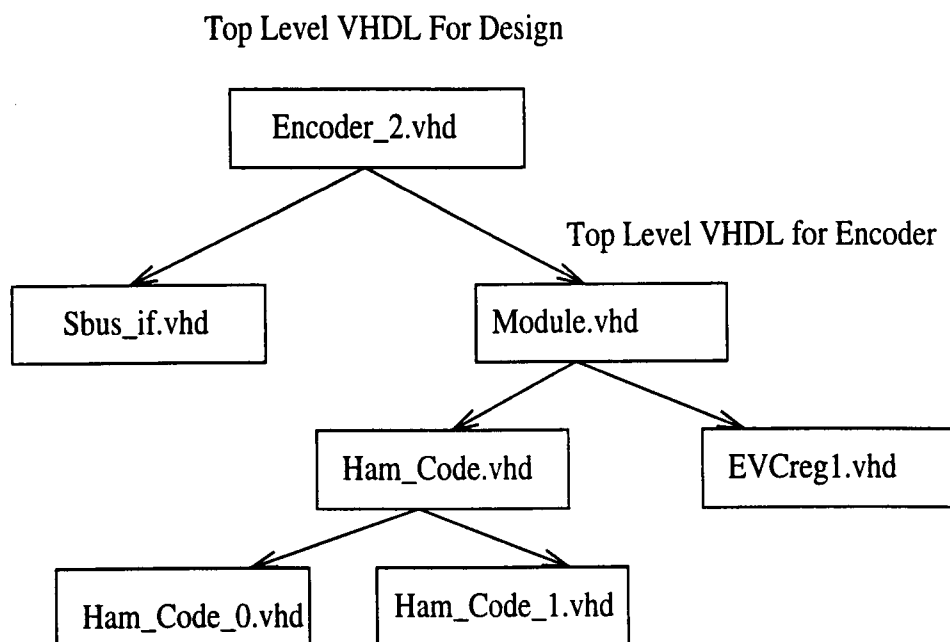


Figure 3.3: *Structural VHDL For Hamming Encoder*

C Programming

The C code for this example is listed in the appendix. This C code can serve as a template for other designs that configure the EVC. The code first checks the results between the hardware and software designs to see whether or not they are correct. Secondly, it calculates the execution times for the hardware design running on the EVC. The last part calculates the execution times of the Sun Sparc 10 workstation and compares the two executions.

Simulation

After a specific design has been successfully described using VHDL, a behavior level simulation should be performed to check the function of the design. This can be done by using Synopsys VHDL Simulation System (VSS). A Register Transfer Level (RTL) simulation should then be done by VSS. Only after the design's functionality has been verified correctly, can designs be plugged in the provided VHDL and C code templates. This will shorten the design time for developing future reconfigurable computing systems. The system level simulation is not necessary for the EVC because the final results will be verified by the software equivalent part of the C code.

CHAPTER 4

Test Cases For Reconfigurable Computing and Results

In this chapter, two distinct digital signal processing applications using the reconfigurable computing system will be presented. The performance of these systems will be discussed and be compared to equivalent implementations using C programs with execution on a Sun Sparc 10 station. The two applications discussed are two parallel Hamming encoders (Hamming Encoders) and a Finite Impulse Response (FIR Filter). The implementation methods for Hamming Encoder design are the same as those presented in Chapter 3.1. While the implementation methods for the FIR filter involve Synopsys' latest CAD tool for developing DSP applications — COSSAP.

The purpose of these test cases are to implement some functions on the EVC and to determine experimentally the speedup provided by the hardware. Also, these two test cases provided methodologies for reconfigurable computing using the Engineers' Virtual Computer. Some more test cases were developed by others following the same methods as the Hamming encoder design. The results of these test cases will be mentioned in this chapter. In each test case, the function

was described using the VHDL and then synthesized using the Synopsys toolset into the Xilinx 4000 FPGA technology as shown in the Figure 3.2 on page 23. Xilinx XACT tools were used to perform the physical place and route and obtain a configuration file for a Xilinx 4010 device. The file was then converted into a format such that it could be downloaded into the EVC. Software was then written in the C language to configure the EVC, download data to it and retrieve data from it for display purposes. Software corresponding to the hardware function was also written, and their execution results are compared.

4.1 Test Case I: Hamming Encoder

The specific function developed in the Hamming Encoder consisted of two parallel Hamming encoders, each accepting an 8-bit parallel input and delivering a 12-bit encoded output. Hamming Encoder algorithm is used extensively in data communication system. Whenever input data is run through this algorithm and run through Hamming Decoder, the output data strings of Hamming Encoder/Decoder can automatically correct one single bit errors which may occur during the communication process. The equations for the Hamming Encoder [5] is:

$$\begin{aligned} \text{DATA_OUT}(0) &= \text{DATA_IN}(0) \text{ XOR } \text{DATA_IN}(1) \\ &\quad \text{XOR } \text{DATA_IN}(3) \text{ XOR } \text{DATA_IN}(4) \end{aligned}$$

```

        XOR DATA_IN(6);
DATA_OUT(1)=DATA_IN(0) XOR DATA_IN(2)
        XOR DATA_IN(3) XOR DATA_IN(5)
        XOR DATA_IN(6);
DATA_OUT(2)=DATA_IN(0);
DATA_OUT(3)=DATA_IN(1) XOR DATA_IN(2)
        XOR DATA_IN(3) XOR DATA_IN(7);
DATA_OUT(4)=DATA_IN(1);
DATA_OUT(5)=DATA_IN(2);
DATA_OUT(6)=DATA_IN(3);
DATA_OUT(7)=DATA_IN(4) XOR DATA_IN(5)
        XOR DATA_IN(6) XOR DATA_IN(7);
DATA_OUT(8)=DATA_IN(4);
DATA_OUT(9)=DATA_IN(5);
DATA_OUT(10)=DATA_IN(6);
DATA_OUT(11)=DATA_IN(7);

```

The equations for the Hamming Encoder show that this algorithm consists of some Exclusive OR logic operations. The hardware implementation for the Hamming encoder was done by using VHDL hardware description language. The structural VHDL were used to implement the reconfigurable computing system. Comparing to the figures in Figure 4.1, the corresponding VHDL structure files

are as in Figure 3.3 on page 29. This is a graphical view of the Hamming Encoder hierarchy along with the icons that might be used to represent each level of the hierarchy.

The modules for the whole design consisted of one top level module of the system design (`encoder_2`), a functional module (`module`), a SBus interface (`sbus_if`), two Hamming Encoders (`ham_code`), and one register (`evcreg1`). All of these modules are shown in Figure 4.1. Gate_Level simulation for the two parallel Hamming Encoders is shown in Figure 4.2. The schematic for two parallel Hamming Encoders is shown in Figure 4.3. The schematic for one Hamming Encoder is shown in Figure 4.4. The gate level schematic for one Hamming Encoder algorithm is shown in Figure 4.5. Figure 4.6 shows the top level schematic of the recomputable computing system which includes EVC SBus interface macro.

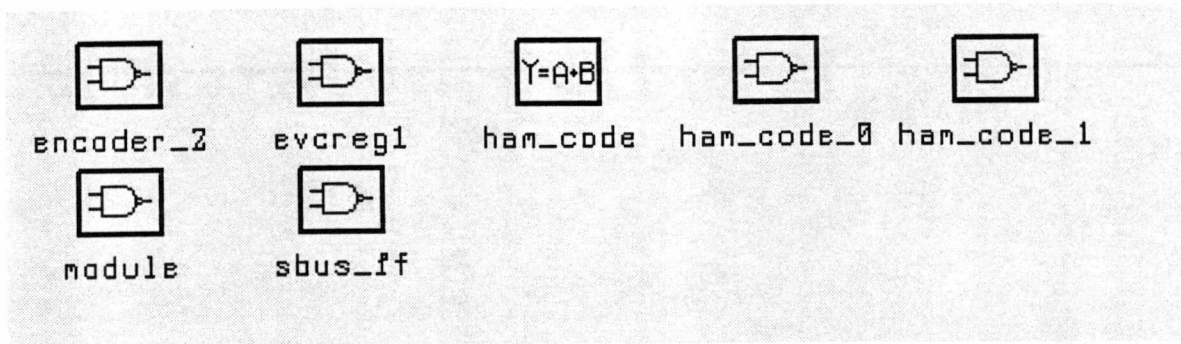


Figure 4.1: Modules Of Two Hamming Encoders And The EVC

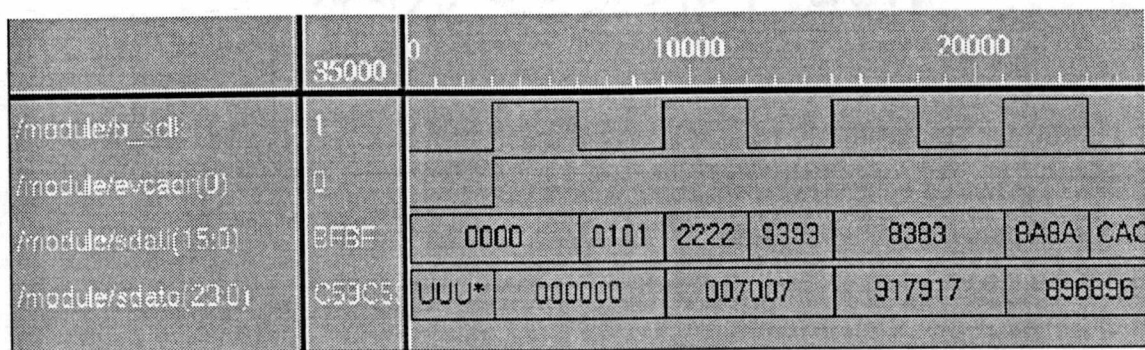


Figure 4.2: Gate_Level Simulation For Two Parallel Hamming's Encoders

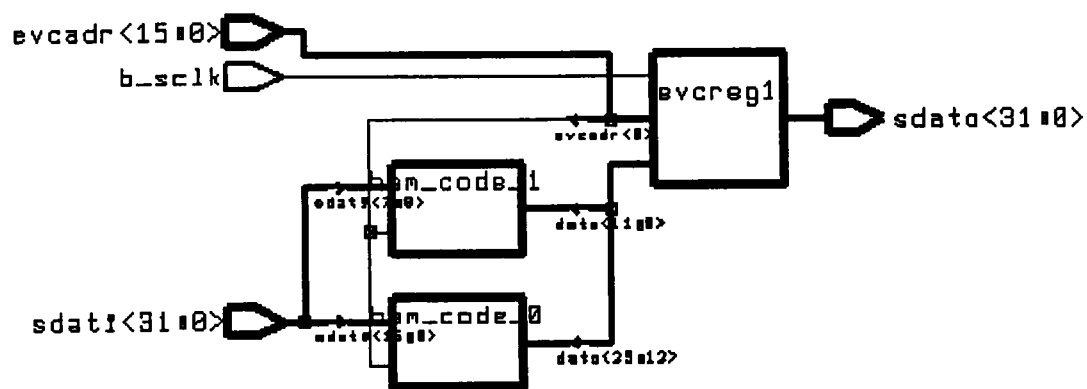


Figure 4.3: *Top Level Schematic For Two Parallel Hamming Encoders*

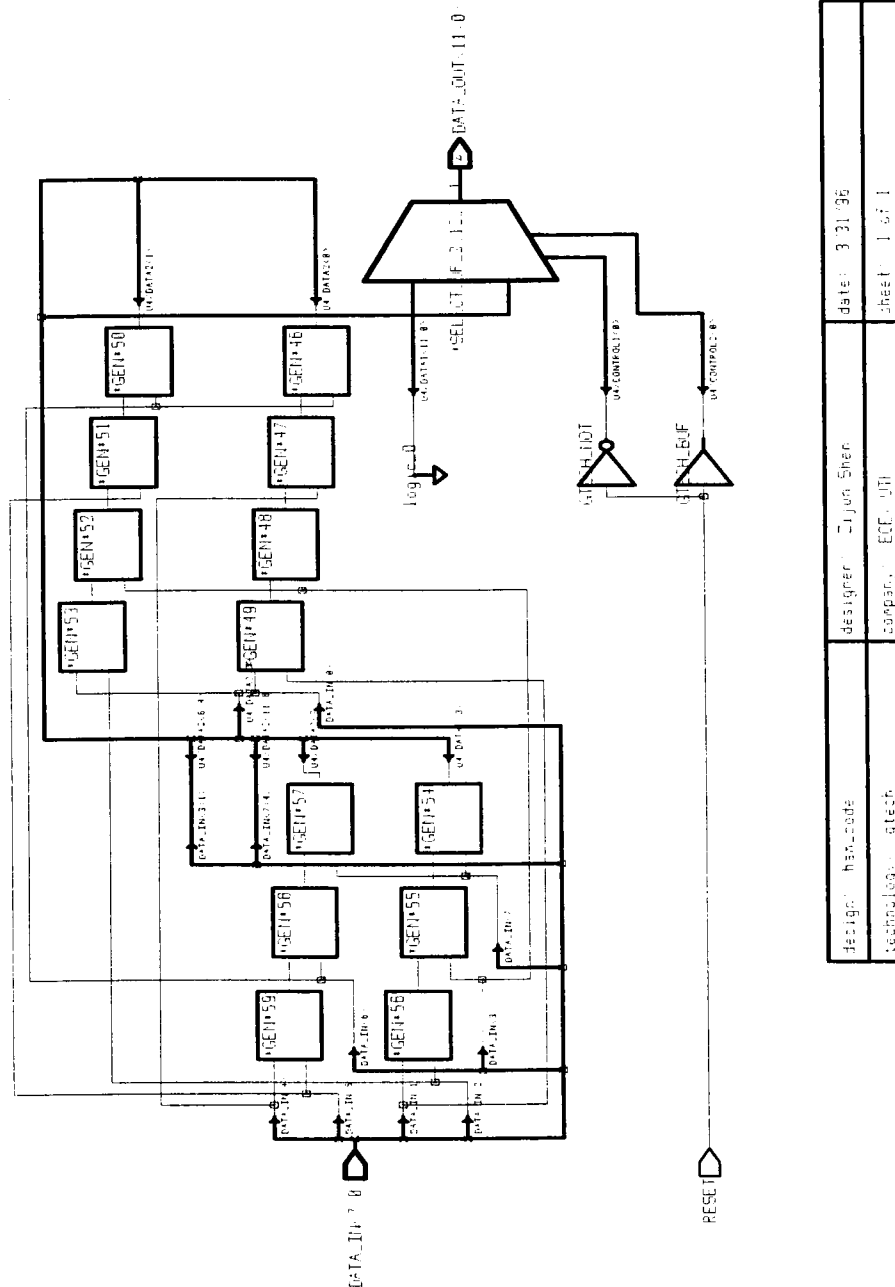
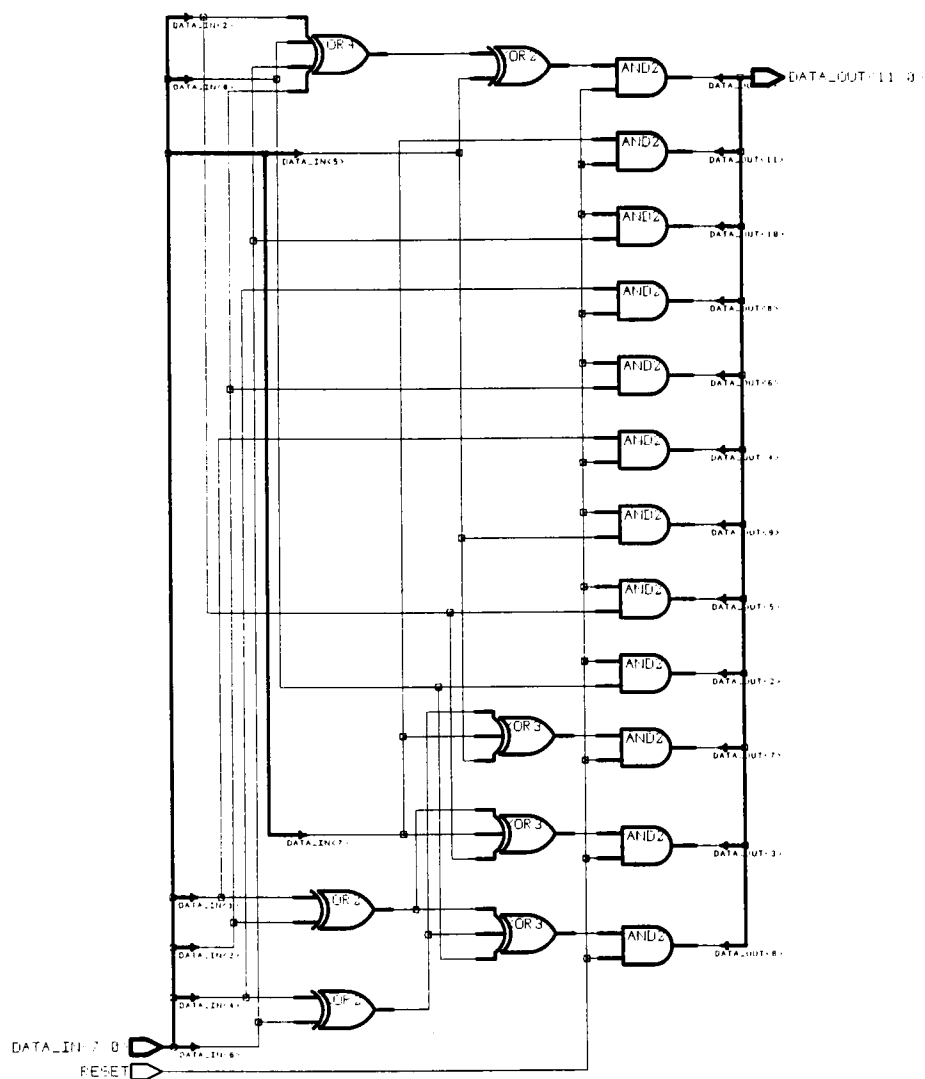


Figure 4.4: Top Level Schematic For One Hamming Encoder Algorithm



design: ham_code_1	designer: Zijun Shen	date: 3/31/96
technology: xprn4000-5	company: ECE, UTR	sheet: 1 of 1

Figure 4.5: Gate Level Schematic For One Hamming Encoder Algorithm

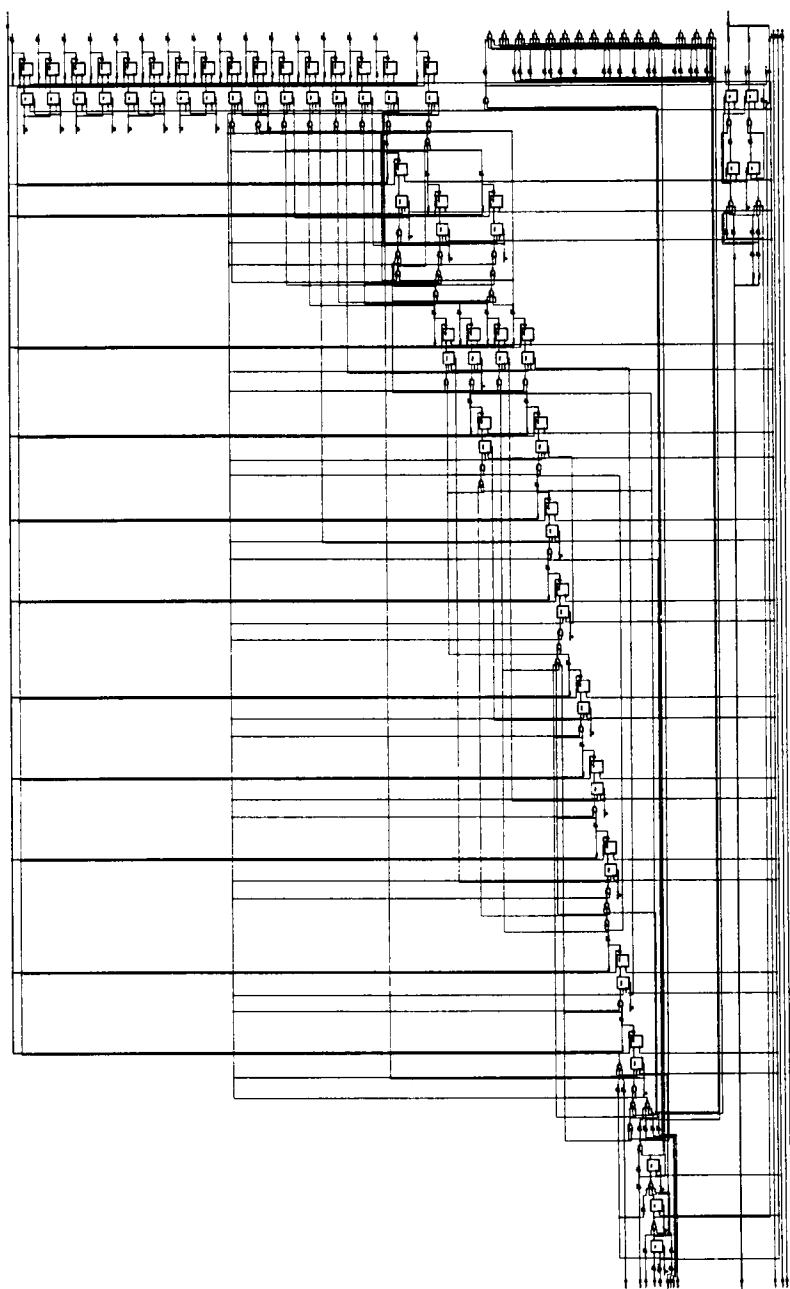


Figure 4.6: *Gate Level Schematic for Two Parallel Hamming Encoders*

The software implementation for the Hamming Encoder was done by using C programming on the Sun Sparc 10 workstation. In the C code (refer to the Appendix of this thesis), void HAM() is defined as the software equivalent solution to mimic the hardware design, which in this case, is the Hamming Encoder algorithm. Function void HAM() is defined by using C code and has nothing to do with hardware objective design; hence, it is the Software-Only Implementation for Hamming Encoder algorithm.

After both hardware and software have been successfully implemented, C code has been written to include both hardware and software implementation. Parts of results for this case are shown in following:

Software-Input: 5f5f Software-Output: 57d57d

Hardware-Input: 5f5f Hardware-Output: 57d57d

Software-Input: 6161 Software-Output: 606606

Hardware-Input: 6161 Hardware-Output: 606606

Software-Input: 6363 Software-Output: 61f61f

Hardware-Input: 6363 Hardware-Output: 61f61f

Download time is 0.160000 seconds

EVC and Software Perform Hamming Code Encoder Algorithms: 80000000 strings

Hardware execute time is: 10.330000 seconds

Hardware execute time + Download time is: 10.490000 seconds

Software execute time is: 41.630001 seconds

EVC saves 31.140001 seconds

The results indicate that the hardware accelerated solution produced a speedup of over 4 times as compared to the software-only solution for an input data string of 80,000,000 characters. We could draw a time line to show the results, refer to Figure 4.7 [2].

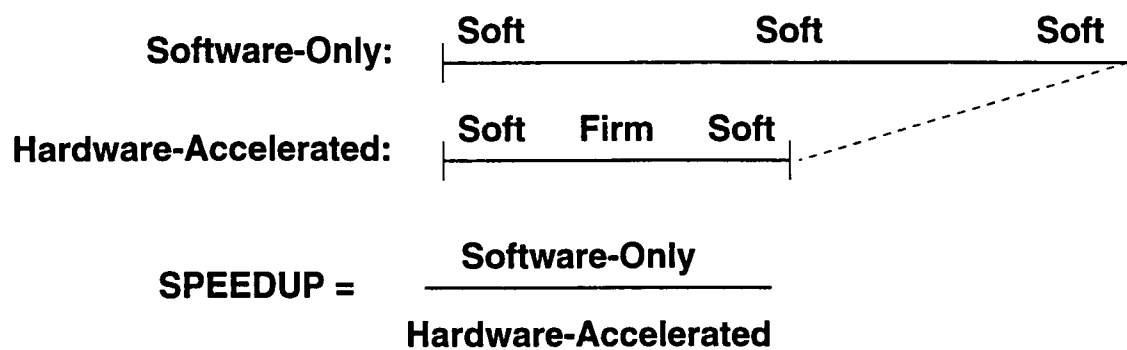


Figure 4.7: *Execution Time-Lines and Speedup.*

$$y(n) = \sum_{k=0}^{N-1} b_k * x(n-k)$$

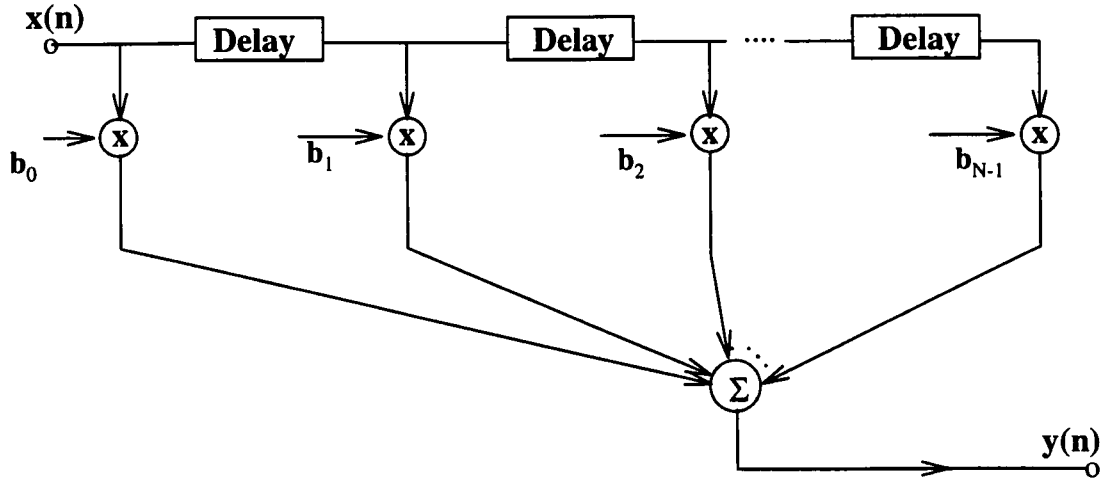


Figure 4.8: *Realization of FIR Filters*

4.2 Test case II: FIR Filter

A popular digital signal processing algorithm that is required in most signal processing systems and that is often used as a benchmark for rating the performance of signal processing systems is the Finite Impulse Response (FIR) filter. The equation for the computation of an N-tap FIR filter is:

$$y(n) = \sum_{k=0}^{N-1} b_k * x(n - k)$$

The realization of FIR filter is shown in Figure 4.8. This is simply the convolution

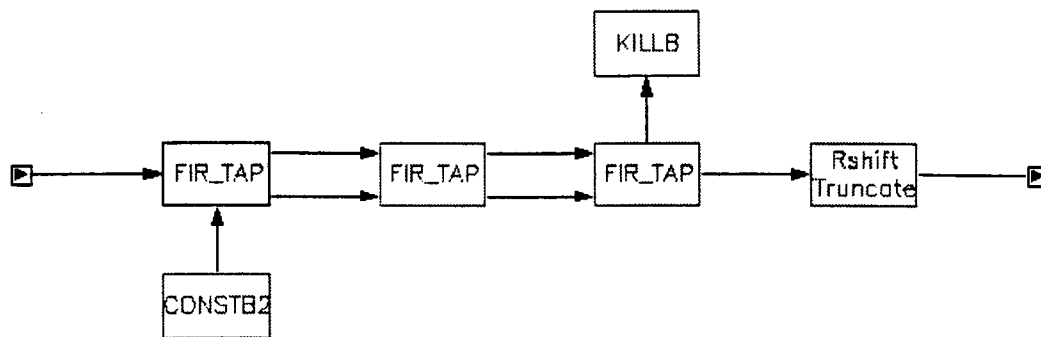


Figure 4.9: *A Bit_Vector FIR Filter of Third Order*

of the input sequence $x(n)$ with the filter coefficients/taps b_k . A signal flow graph for the computation of an N-tap FIR filter also appears in Figure 4.8. As can be seen in the flow graph the computation of an FIR filter requires a series of multiplications which are summed together to form the final output. The computation can be performed sequentially using a multiply-accumulate loop, in full parallel using N multipliers and N-1 adders, or as a combination of sequential-parallel computation [6].

For the FIR design, the order n is 3rd and the word_length N is 8. To implement this 3-tap 8-bit FIR filter, Synopsys DSP development tool COSSAP is used. The design flow chart for the FIR filter design is shown in Figure 3.1 on page 22.

By using COSSAP's graphic editor, the equivalent block diagram for Figure 4.8 was drawn as shown in Figure 4.9. As seen in Figure 4.9, the block diagram

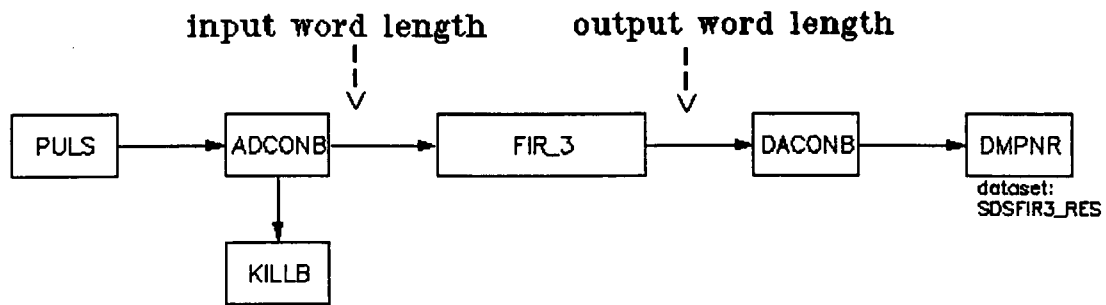


Figure 4.10: 3rd Order FIR Filter

consists of three single taps of a bit_vector FIR filter. This forms the FIR filter block (FIR_3) that appears in Figure 4.10. One single tap of a bit_vector FIR filter is shown in Figure 4.11. A single tap of a bit_vector FIR filter is a primitive block which existed in the COSSAP bit_true library. After executing a set of COSSAP's small programs, a VHDL file for this FIR was generated and was then synthesized by using either the Synopsys FPGA Compiler or the Design Compiler. As shown in the design flow chart Figure 3.1 on page 22, COSSAP CAD tools enable DSP designers to work at the system level while achieving the quality of results necessary for low cost, high performance systems. This methodology consolidates concept development and validation processes at the gate, register-transfer (RT) and system levels [10]. From the viewpoint of design methodology, the design method used in the FIR filter design is more efficient and powerful compared to the method used in the Hamming encoder design. This method for implementing

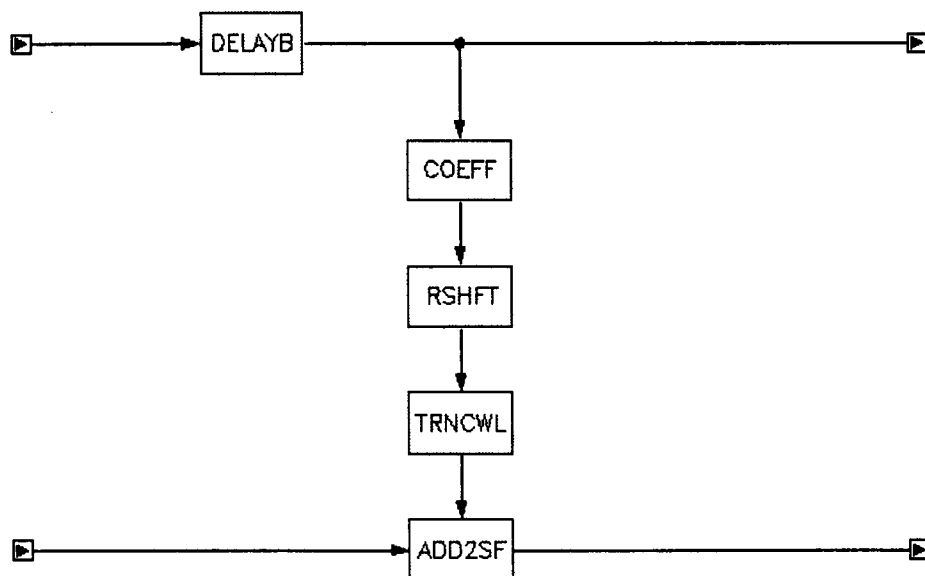


Figure 4.11: *A Single Tap of A Bit-Vector FIR Filter*

the EVC could also be used as fast-prototyping and it is a practical method for reconfigurable computing. Not only can COSSAP automatically generate HDL code and DSP code from the block-diagram specification level, but it also can simulate design at the same level. This provides the capability to target designs quickly. To simulate the behavior of the FIR, a VHDL testbench file for the FIR was generated by using COSSAP 'xvsi' program and this VHDL testbench can be used as a VHDL co-simulation model or a functional simulator in order to simulate the original design at the system or block diagram level. Refer to Figure 4.12, the co-simulation model FIR_3_VHDL was generated from the original design FIR_3. In the FIR filter design, both the COSSAP Stream Driven Simulator (SDS) and the Synopsys VHDL System Simulator (VSS) were performed to simulate the de-

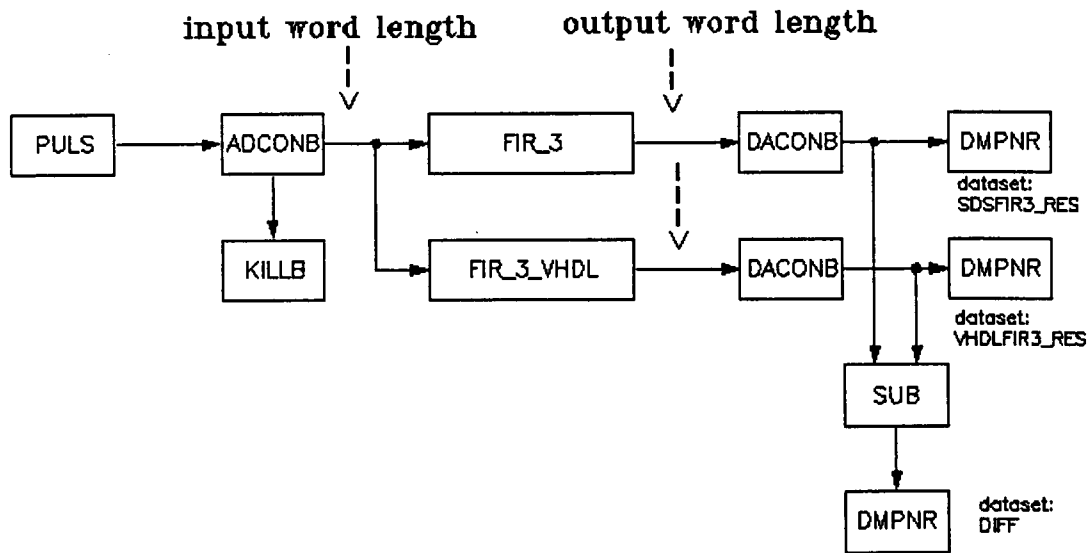


Figure 4.12: 3rd Order FIR Filter VHDL Testbench Use by SDS And VSS

sign, and their results were the same which is shown in Figure 4.13. Also, a report file for simulation results was generated by SDS.

After the design has been verified functionally correct, the FIR could be plugged in the reconfigurable computing system by using the same way as used in the two Hamming encoders design. Again, structural VHDL was written to implement the FIR design into the EVC. The top level design of this case for the EVC is shown in Figure 4.14. It consists of SBus module and FIR module (marked as 'my design' module in the top level schematic). The SBus module is shown in Figure 4.15. The top level design for FIR is shown in Figure 4.16 which consists of three single tap of a bit_vector FIR filter (refer to Figure 4.17.) and a

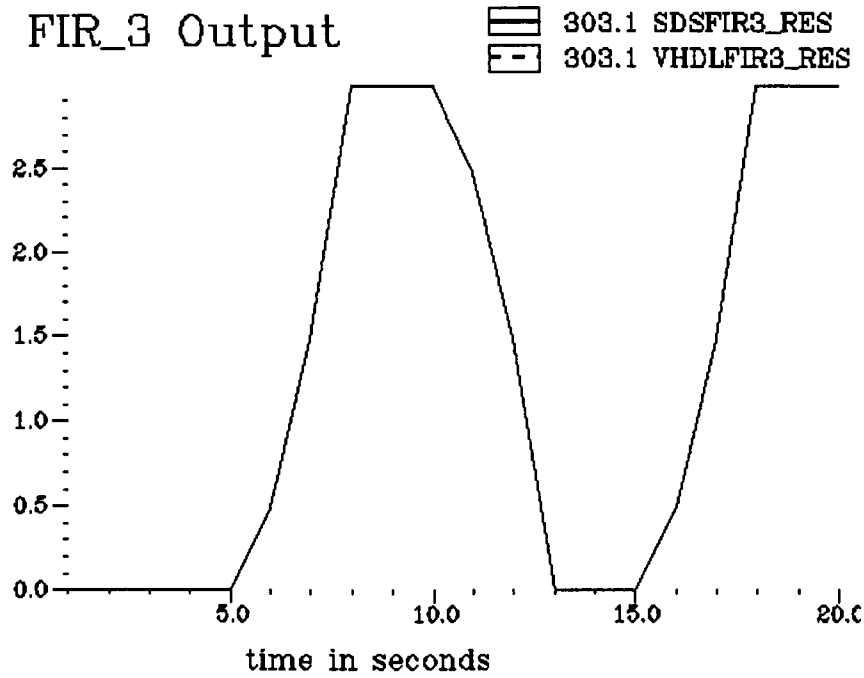


Figure 4.13: *Simulation Of 3rd Order FIR Filter By Using SDS And VSS*

single tap of a bit_vector FIR filters and other modules as shown in Figure 4.17.

A single tap of a bit_vector FIR filter is shown in Figure 4.18.

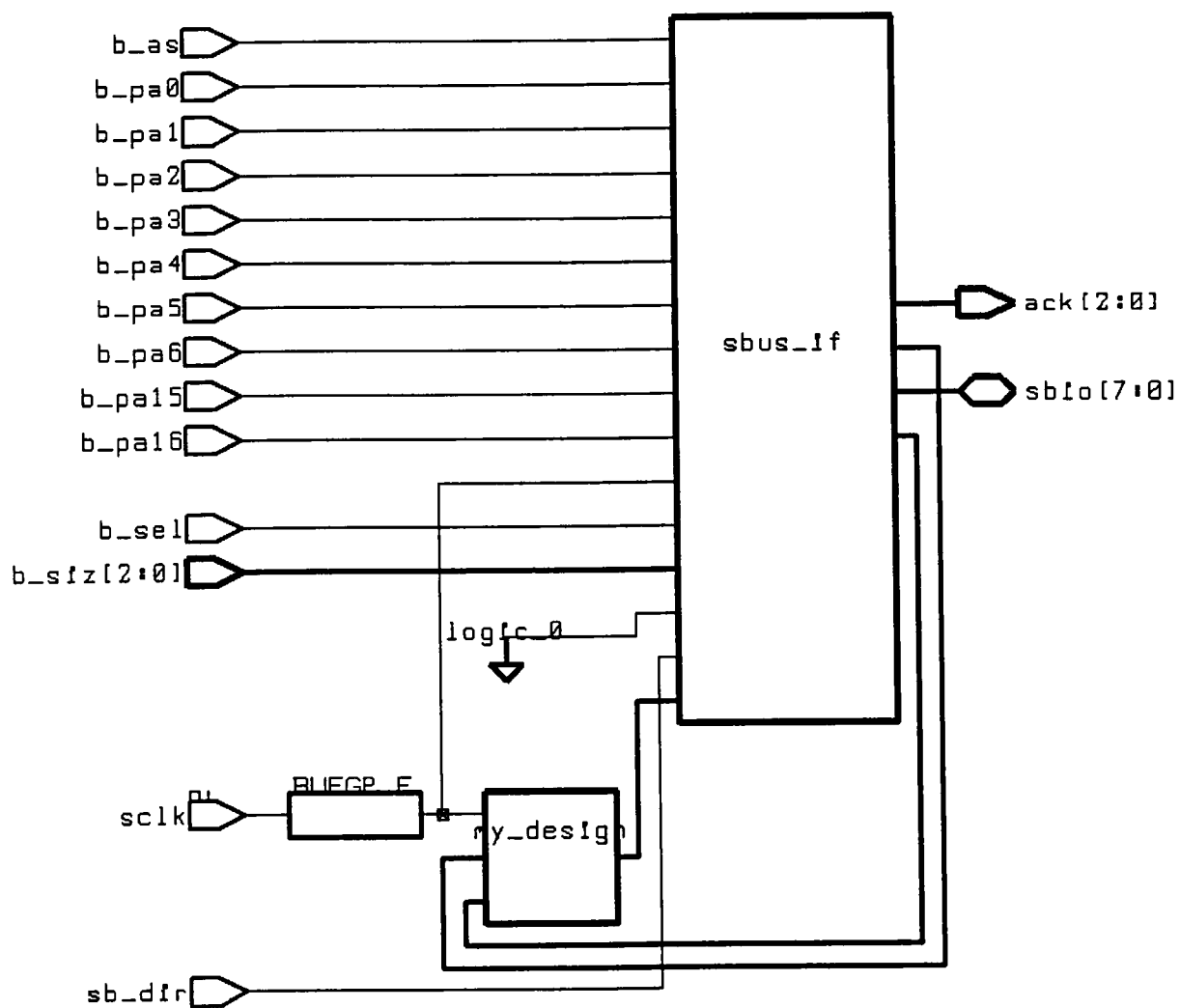


Figure 4.14: *Top Level Schematic For FIR Filter*

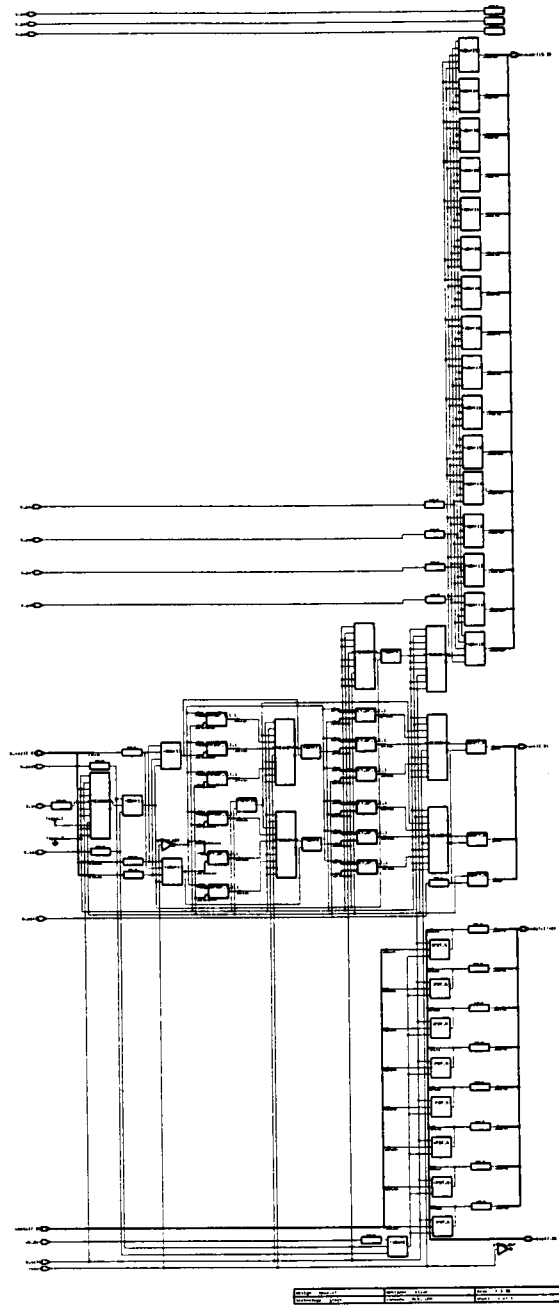


Figure 4.15: *Schematic Of SBus Interface For FIR Filter*

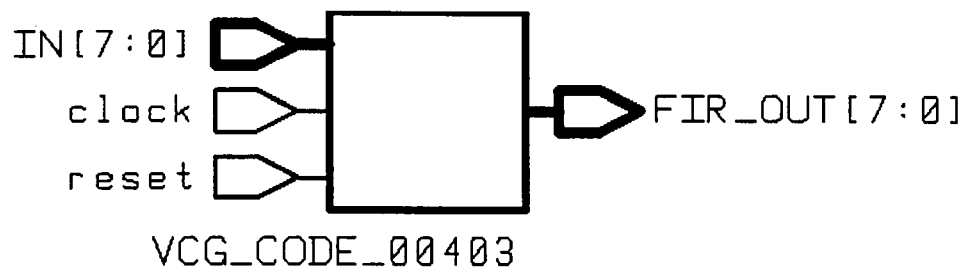


Figure 4.16: *Top Level Module Of FIR*

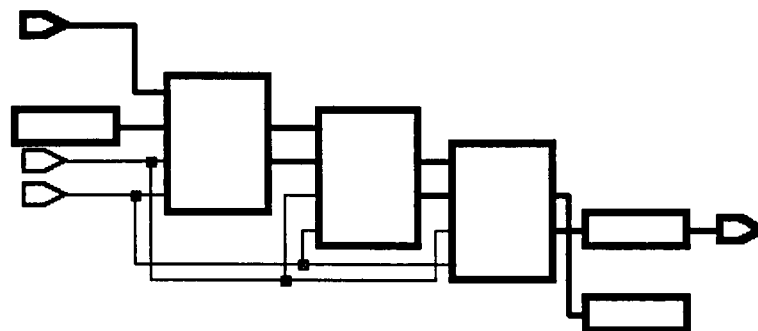


Figure 4.17: *Sub-Level Module of FIR Module*

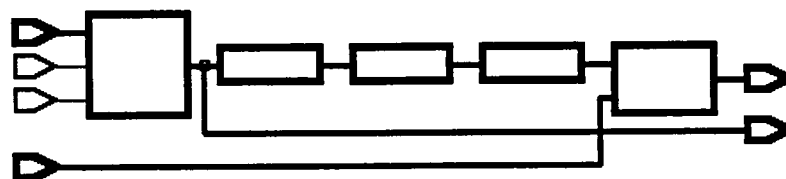


Figure 4.18: *Top Level Module of A Single Tap of a Bit-vector FIR filter*

The software equivalent for the FIR was written for the Sun Sparc 10 workstation. In the C code (refer to Appendix C), function void fir() is defined as the software equivalent solution to mimic the FIR algorithm (hardware design). The hardware design is also included in this C code. Both software and hardware functions obtained an input $x(n)$, then they performed the FIR algorithm independently, and execute times for both were calculated. Here are some results:

For 100 numbers of inputs:

Input: 93 Output: 139

Input: 94 Output: 141

Input: 95 Output: 142

Input: 96 Output: 144

Hardware objective download time is 0.170000 seconds

Calculated time for EVC 100 inputs in 0.010000 seconds

Hardware execute time + Download time is: 0.180000 seconds

Calculated time for software 100 inputs in 0.070000 seconds

For 200 numbers of inputs:

Input: 196 Output: 166

Input: 197 Output: 167

Input: 198 Output: 169

Input: 199 Output: 170

Hardware objective download time is 0.170000 seconds

Calculated time for EVC 200 inputs in 0.090000 seconds

Hardware execute time + Download time is: 0.260000 seconds

Calculated time for software 200 inputs in 0.280000 seconds

For 400 numbers of inputs:

Input: 396 Output: 338

Input: 397 Output: 339

Input: 398 Output: 341

Input: 399 Output: 342

Hardware objective download time is 0.170000 seconds

Calculated time for EVC 400 inputs in 0.200000 seconds

Hardware execute time + Download time is: 0.370000 seconds

Calculated time for software 400 inputs in 1.140000 seconds

For 900 numbers of inputs:

Input: 896 Output: 832

Input: 897 Output: 833

Input: 898 Output: 835

Input: 899 Output: 836

Hardware objective download time is 0.170000 seconds

Calculated time for EVC 900 inputs in 0.390000 seconds

Hardware execute time + Download time is: 0.560000 seconds

Calculated time for software 900 inputs in 5.800000 seconds

The results indicate that with the increase of input numbers, the EVC obtains more speedup compared to the software-only solution. Particularly, when

the numbers of inputs are 900, the hardware accelerated solution produced a speedup of over 9 times as compared to the software-only solution. On the other hand, when the numbers of inputs are 100, the EVC speedup reduced to only that of the software-only solution. Notice, the hardware download time is almost fixed even for different designs and different input numbers, and it is around 0.17 seconds. The calculation time alone for the EVC of 100 input numbers is 0.01 second and this is still 7 times faster than that of the software-only solution. The reasons behind this are given in the next chapter.

4.3 Some more Test Cases

The test cases in the following were developed by others who used the files of Test Case I as template files to create their own designs. Those designs are Pseudo-Random Number Generator, Priority Encoder, Variable Length Right Shifter and Multiple Function Comparator.

Pseudo-Random Number Generator consisted of two parallel 16 bit Linear Feedback Shift Registers (LFSRs), each accepting a 16-bit input and delivering a 16-bit output. The results (as shown in Table 4.1) indicate that the hardware accelerated solution was little bit faster than that of a C program to accomplish the same task.

The specific function developed in Priority Encoder consisted of a single

Table 4.1: *Performance of Pseudo-Random Number Generator*

Execution	Time (seconds)
Software-Only	11.84
Hardware Download	00.16
Hardware Execution	9.02
Time Difference w/o Download	2.82
Time Difference with Download	2.66
Speedup w/o Download	$11.84/9.02 = 1.3$

module that accepted a single 32 bit input. Based on which left-most input bit was high, a 5 bit op-code was outputted from the module. For example, if the least most significant input bit was the only bit that was high, the output would be 5 binary 0's. If the most significant input bit was high, the output would be 5 binary 1's. The 5 bit op-code therefore can be used to cover all 32 possibilities. The test results is shown in Table 4.2.

In the table, the minus sign indicates that this algorithm runs faster on Sun Sparc than it runs on the EVC. In this case, the "speedup" is 0.15, which is less than one. This is true for all the other test cases. For example, in the following tables, when "speedup" is bigger than one, it refers the EVC is the winner.

Table 4.2: *Performance of Priority Encoder*

Execution	Time (seconds)
Software-Only	1.4
Hardware Download	00.18
Hardware Execution	9.04
Time Difference w/o Download	-7.63
Time Difference with Download	-7.81
Speedup w/o Download	$1.4/9.04 = 0.15$

Variable Length Right Shifter consisted of two 8-bit right shift registers controlled by two 8-bit control words. The control words determined how many bits to shift right. The test results is shown in Table 4.3.

Multiple Function Comparator is a 16-bit comparator. Two 16-bit numbers are compared and a three bit code is generated to signify the results of the compare. The two 16-bit numbers both come from a 32-bit source representing the SBUS in the Sparc station where the EVC is located. This 32-bit number is split into the two 16-bit variables labeled A and B. The comparison is then made. If number A is equal to number B, a code of '001' or '1' is generated. If A is greater than B, a code of '010' or '2' is generated. If A is less than B, a code

Table 4.3: *Performance of Right Shifter*

Execution	Time (seconds)
Software-Only	0.33
Hardware Download	0.17
Hardware Execution	0.10
Time Difference w/o Download	0.23
Time Difference with Download	0.06
Speedup w/o Download	$0.33/0.10 = 3.3$

of '100' or '4' is generated. The results, as shown in Table 4.4, indicate that the software solution was over 2x faster than the hardware "accelerated" solution for an input data string of 800,000,000 characters.

4.4 Results And Discussions

Many designs concerning the reconfigurable computing system were successfully developed and tested. From the above test cases, three different results can be obtained. First, algorithms such as the Hamming Encoders, the FIR and the Random Number Generator obtain gains in speed over the Sun Sparc 10

Table 4.4: *Performance of Multiple Function Comparator*

Execution	Time (seconds)
Software-Only	38.89
Hardware Download	00.17
Hardware Execution	94.72
Execution + Download	94.89
Time Difference with Download	-56.00
Time Difference without Download	-56.83
Speedup w/o Download	$38.89/94.72 = 0.41$

Workstation through the use of the EVC. For the Right Shifter, the EVC and Sun Sparc 10 Workstation have almost the same execution time. Finally, for the cases of the Priority Encoder and the 16-bit Comparator, the Sun Sparc 10 workstation have smaller execution times than that of the EVC. Table 4.5 shows the three different kinds of results for several different algorithms.

From these results, it is clear that algorithms which are computationally intensive can be executed quicker when configured into the EVC. Thus, firmware can often execute in less time than that of a software-only solution as depicted

Table 4.5: *Speedup Results for All Test Cases*

Algorithms	Software-Only	Firmware	Time Difference	Speedup
Hamming Encoder	41.11 (sec.)	10.14 (sec.)	30.78 (sec.)	4.05 x
FIR	5.82 (sec.)	0.42 (sec.)	5.24 (sec.)	13.86 x
Right Shifter	0.33 (sec.)	0.10 (sec.)	0.06 (sec.)	3.3 x
RN Generator	11.84 (sec.)	9.02 (sec.)	2.66 (sec.)	1.3 x
Priority Encoder	1.4 (sec.)	9.048 (sec.)	-7.81 (sec.)	0.15 x
Comparator	38.89 (sec.)	94.72 (sec.)	-56.00 (sec.)	0.41 x

by the time-lines in Figure 4.7 on page 41 [2]. In the Hamming encoder and the FIR, the ratio of firmware execution time to software execution time decreases when more input data is used. This is because the data cache of the Sparc 10 is limited to two kilobytes and a large number of inputs will exceed the data cache limitation. In the Hamming Encoders, 80,000,000 input streams flooded the Sparc CPU data cache. Thus, processing all the inputs took a considerable amount of extra time. Other reasons that the EVC runs faster than the Sparc 10 workstation are that the implemented algorithms on the EVC can take advantage of using parallel calculations, which one Sun Sparc CPU is simply not capable of doing. In Hamming encoder design, two Hamming Encoders run parallel on the

EVC, and in FIR design, parallel multipliers were used to calculate convolutions. The parallel multiplier generally gives the best multiplication rates. Hence, when implementing a larger or a more complicated designs into the EVC, a quicker execution time can be expected.

CHAPTER 5

SUMMARY, CONCLUSION AND FUTURE WORK

5.1 Summary

The goal of this thesis was to develop a reconfigurable computing environment, to provide methods that for reconfigurable computing, and to determine quantitatively if reconfigurable hardware using FPGAs is capable of providing a significant performance improvement over the Sun Sparc 10 processor for digital signal processing applications. The success of developing and implementing hardware designs into a reconfigurable computing system in Chapter 4 experimentally proves the accuracy of methods developed in Chapter 3. The performance of test applications indicate an order of magnitude increase in performance for the EVC over the Sun Sparc 10 processor performance. This can generally be expected with FPGA-based reconfigurable systems. Thus these results justify the applications of FPGAs and reconfigurable systems to digital signal processing. Several conclusions can be formed regarding the use of a reconfigurable computing system for digital signal processing based on the example implementations. These conclusions will be discussed next.

5.2 Thesis Conclusions

5.2.1 Conclusions

Custom hardware designs are commonly designed using schematic capture utilities. This requires the designer to think at a much lower level than the actual flow of an algorithm and often immerses the designer in details unrelated to the algorithm being implemented. The approach of this thesis project removes the burden of minor details from the designer by providing methods of implementing algorithms at a level that is closer to the actual algorithm itself. Basically, this thesis develops several design methods of implementing digital signal processing algorithms into a reconfigurable computing system. A set of structural VHDL files and related C programming code serve as template files for configuring the EVC. This allows future users to conveniently implement their firmware into the EVC efficiently. These template files can also allow the EVC to serve as fast-prototyping for different hardware designs. As structural VHDL templates gave users an environment to work at a high level to do reconfigurable hardware designs, Synopsys' COSSAP provides the possibility for users to work at a block diagram or system level in implementing algorithms into hardware designs.

COSSAP, Synopsys' FPGA Compiler, Xilinx XACT software, and the VHDL template files were used to implement an FIR into the EVC. This design method provided an environment for doing reconfigurable computing for signal

processing designs. This state-of-art method greatly shortens product's time-to-market. This is one of the most efficient, as well as latest, design methods for implementing hardware designs into the EVC.

From the results of test cases in Chapter 4, it is concluded that dramatic performance gains can be obtained through the "hardwiring" of a computationally intensive algorithm through a reconfigurable computing system. In comparison to software-only implementations on a Sun Sparc 10 station, the FPGA based hardware implementation is overall faster than the equivalent software-only implementation. Furthermore, for a computationally intensive algorithm, computing several multiplications and accumulations all at once in a firmware design can increase parallelism. A single Sun Sparc processor just cannot do that. Thus, this parallelism can obtain dramatic performance gains for a FPGA over a Sun Sparc processor. This clearly indicates the advantage in the use of reconfigurable computing.

Using the EVC while resident in the system gives the system designer a new degree of freedom, not available with any other type of logic. Hardware can be implemented and changed as easily as software. Design updates or modifications are also easy. Hence, an additional advantage of the use of reconfigurable hardware over ASICs, mentioned briefly in Chapter 2, is the ability of reconfigurable hardware to employ increased application specificity through hardware reconfiguration. A reconfigurable computing system such as the EVC can be made to be

even more application specific than a VLSI ASIC since the specific requirements of the system can be fixed in hardware at the time the system is to be used. For example, ASICs, while designed to be applied to a specific application class, almost always allow some flexibility through the setting of switches or the programming of function registers. This allows the user of the ASIC to adjust certain basic operational parameters specific to his application of the ASIC. This is not necessary with reconfigurable hardware since the necessary parameters can be adjusted by changing the data stream used to program the hardware. This has the advantage of reducing extra circuitry needed to allow the parameter adjustments and may also lead to an increase in performance since the adjustments have been made to actual hardware rather than through the setting of “soft” values. Therefore, the reconfigurable computing system provides a significant performance increase over software-programmable-based methods while still maintaining the advantage of flexibility.

5.2.2 Suggestions

Some memory modules need to be added onto the EVC board in order to increase the performance of the EVC as well as to configure complicated applications such as image processing algorithms. For some test cases in Chapter 4, some algorithms such as the ‘16-bit Comparator’ and the ‘16-bit Priority Encoder’ run slower than the Sparc because of the Sbus I/O limitations. This is also one of the reasons

that the Hamming Encoders cannot obtain the same speedup gains as the FIR can. If a memory module were on the board, it would be possible to send data to the memory, process the data, collect the results in memory, and then read the results back from the memory. This would make the EVC run much faster.

5.3 Future Work

This thesis has shown the possibility of using reconfigurable hardware based FPGAs to obtain a performance increase over a Sun Sparc 10 processor in some digital signal processing algorithms. All of the test cases used only 7% to 25% of the resources of a XC4010. Future work could profitably be done to implement some larger designs which take 50% to 70% resources of a XC4000. Better results might be obtained by configuring complicated functions on the EVC because a Xilinx FPGA device can be reconfigured dynamically to perform different functions at different times. Thus, future work could also include the implementation of more complicated algorithms into the EVC by taking advantage of FPGAs' abilities of doing dynamic reconfigurable computing. This includes implementation of system self diagnostics (built-in-self-test) or implementation of dual-purpose hardware for a given application. Furthermore, a Graphical User Interface could be developed to make the involved CAD tools and the VHDL template files more user friendly. This software could also allow test results to be shown in a graphic format in order to give users more direct results in the performance of both the

EVC and Sun Sparc 10 station.

BIBLIOGRAPHY

BIBLIOGRAPHY

- [1] N. W. Bergmann and J. C. Mudge. *Comparing the Performance of FPGA-based Custom Computers with General-Purpose Computers for DSP Applications*. IEEE Workshop on FPGAs for Custom Computing Machines, April 1994.
- [2] Bouldin, D. W. ECE 552 notebook and project reports: Hardware acceleration by using the engineer's virtual computer. Technical report, Dept. of Electrical and Computer Engineering, University of Tennessee, Knoxville, March 1996.
- [3] Reconfig Company. Glossary-reconfigurable computing. Technical report, Reconfigurable Computing, <http://www.reconfig.com>, 1996.
- [4] Xilinx Company. *Xilinx Synopsys Interface FPGA User Guide*. Xilinx Company, San Jose, CA, Dec. 1994.
- [5] Hamming, R. W. *Coding and information theory*. Prentice-Hall,, 1980.
- [6] Ludeman,L. C. . *Fundamentals of Digital Signal Processing*. John Wiley and Sons, New York, 1986.
- [7] Oldfield,J. V. and Dorf,R. C. *Field-Programmable Gate Arrays Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems*. Wiley-Interscience Publication, 1995.
- [8] Petersen,R. J. An Assessment of the Suitability of Reconfigurable System for Digital Signal Processing. Master's thesis, Brigham Young University, Sept. 1995.
- [9] Schewel,J. Reconfigurable Computing. Technical report, Virtual Computer Corporation, 1996.
- [10] Synopsys Inc, Mountainview , CA. *COSSAP User's Guide*, Oct. 1995.
- [11] Thornburg,M., Casselman, S., and Schewel,J. *Engineer's Virtual Computer - Technical Reference*. Virtual Computer Corporation, Reseda, CA, August 1994.
- [12] Walters, S. Prototyping asics in reprogrammable hardware meets system requirements. *Computer Design*, page 76, May 1990.
- [13] Weste,N. H. E. and Eshraghian,K. *Principles of CMOS VLSI Design*. Addison-Wesley, 1993.
- [14] Xilinx. The programmable logic data book. San Jose, CA, 1994.

APPENDIX

APPENDIX A

Using Viewlogic and the Engineer's Virtual Computer

Overview and Design Flow Chart of Using EVC

The EVC allows the engineer to test a design on an actual FPGA by providing a simple downloading process that uses 'C' code to test and download the design to the FPGA inside the EVC card [11]. The design flow chart for the EVC is shown in Figure 1.1 on page 6.

Hardware Installation

The EVC card has already been installed inside sun1.engr.utk.edu at SBus slot #3 . No additional hardware configuration and change is needed.

Software Installation

Copy the contents of the directory `cad/course/EVC` to your directory. This directory contains sub-directories, it is recommended that you keep the exact directory structure. You can copy the contents recursively using:

```
sun1:/u4/your_login_name % mkdir EVC
```

```
sun1:/u4/your_login_name % cd EVC
```

```
sun1:/u4/your_login_name/EVC % cp -r cad/course/EVC/* .
```

The directory contains the following sub-directories:

`lib/` : EVC library files `bin/` : EVC executables

`demo-13/` : Contains the following examples

`check/` : Contains the simple register code. (Example 1)

`fib/` : Contains the fibonacci sequence generator. (Example 2)

`t/` : Contains `t.c` which blinks LED. (Example 3)

`viewdraw.ini` : PATH information. Does not need any change.

`wir/,sch/,sym/` : Regular viewlogic files.

Operation

The operation can be summarized as follows (Some of these steps will be explained in greater detail later with reference to an example):

1. Enter the design using any conventional tools (Schematic/HDL).
2. Perform pre-layout simulation for functionality.
3. Add an SBus interface symbol to the schematic.

This will be used to interface the logic within the design to the software through the machine's SBus.

4. Implement the design using Xilinx tools (translate to xnf...and run all steps required until an .rbt file is generated).

5. Run an EVC utility to convert the .rbt file to a .h that can be included in a 'C' code file.

6. Write a simple 'C' code file that includes the header file from step 4.

A set of EVC functions are then used to download the design onto the FPGA and to read the output back. The functions allow the user to download test vectors into the design and read back results through the SBus interface.

7. Compile and execute the 'C' code program.

Once compiled, the executable can be executed as often as required. The program can be written so that it accepts command-line arguments or options which in turn allow for different arguments to be passed to the logic inside the FPGA.

Three Examples and Operation Processes

The following three examples will help you understand the design process of using the EVC.

Example 1: Simple register check

This example is meant to illustrate the steps necessary to download a specific design into the virtual computer and read/write data to it [11]. Bring up the schematic checkreg in the EVC/demo-13/sch directory. [Note: To do so, type `cd demo-13; powerview`. Then, create `project-dir /your_acct/EVC/demo-13` and open `viewdraw` and select `checkreg`.] The schematic represents a simple register to which 32-bit data can be written and read out. The additional logic around the register consists of the interface to the SBus. The SBus interface symbol comes from the EVC library, the others represent `xblox` buses and interface blocks. Do the following:

Type: `xmake checkreg`.

This will start with the `wir/checkreg` file and should run all the way to generate

a checkreg.bit file. You may have to wait 5-10 minutes for this to complete.

Type `makebits -b checkreg`.

This will generate rawbits file for the design.

Type: `../bin/r2h checkreg`.

This will generate a header file for inclusion in the 'C' code. This header file is a representation of the logic design. Change to the check subdirectory [Type: `cd check`] and use an editor to view the checkreg.c file. This file includes the header file from step 4 and will be used to initialize the EVC, download the design, and then transfer data to/from the EVC. The 'C' code fills up a large array with integers, downloads each integer into the virtual computer, and reads it back into a second array. The result is that the first array is copied into the second array. The timer code and function are used to calculate the time taken for the whole operation and compare it with time taken to do the same operation on the Sparc station. The functions used in the 'C' code are explained in the EVC Reference below. Note that the read I/O port address is one more than the write I/O port. Now type: `make`.

This will compile the 'C' code into an executable that can be run. Remember that this executable can only be run on sun1.engr.utk.edu (you can rshell from a remote machine and run it). To execute this example, just type: `checkreg` The following output should be observed:

```
Download time is 0.190000 seconds
Transferred 8000000 bytes in 1.500000 seconds
Transferred 6400000 bytes/sec.
Number of Reps 1
Download time is 0.180000 seconds
Transferred 8000000 bytes in 1.140000 seconds
Transferred 7017543.213886 bytes/sec.
Number of Reps 2
Download time is 0.160000 seconds
Transferred 8000000 bytes in 1.130000 seconds
Transferred 7079646.794441 bytes/sec.
Number of Reps 3
```

Example 2: Fibonacci sequence generator

This example is more complex than the first one and hence the schematic is harder to understand. It is meant to accept an argument from the user such as 5 which would cause the logic to output the 5th fibonacci number. Fibonacci numbers are defined by:

$f(1)=1$

$f(2)=1$

$f(n)=f(n-1)+f(n-2)$

So the sequence goes 1,1,2,3,5,8,...etc. Be sure you are in the demo-13 directory [Type: cd ..] Then, bring up the schematic fib in the EVC/demo-13/sch directory. Run xmake on fib.

This will start with the wir/fib file and should run all the way to generate a fib.bit file. You may have to wait 5-10 minutes for this to complete.

Run makebits -b fib .

This will generate a rawbits file for the design.

Run ../bin/r2h fib .

This will generate a header file for inclusion in the 'C' code. This header file is a representation of the logic design.

View the demo-13/check/fib/fib.c file. This file includes the header file from step 4 and will be used to initialize the EVC, download the design, and then transfer data to/from the EVC. The 'C' code calculates fibonacci numbers using regular equations and outputs the time taken as Sparc time. It also uses the EVC to calculate a specific fibonacci number and outputs the time taken as Virtual time. On average, the EVC saves about 50% of the time compared to Sparc calculation on sun1 (which is a Sparc 10).

Now type "make" to compile the file. Then execute it.

```
sun1:/u4/zijun/temp/EVC/demo-13/fib% fib 3000000
```

```
Sparc time = 0.250000
```

```
The 3000000 Fibonacci number is 0x2dcc7100 (768372992)
```

```
Virtual time = 0.160000
```

```
The 3000000 Fibonacci number is 0x2dcc7100 (768372992)
```

Example 3: Blink LED

In this example, you will see how to use hardware object in 'C' program and how to download your digital design into the EVC. The example blinks the user definable LEDs on the EVC1.

1. Start powerview, and open the schematic t in the EVC/demo-13 directory.

2. Run xmake on t.

This will start with the wir/t file and should run all the way to generate a .bit file. You may have to wait 5-10 minutes for this to complete.

3. Run makebits -b t .

This will generate a rawbits file for the design.

4. Run ../bin/r2h t .

This will generate a header file for inclusion in the 'C' code. This header file is a representation of the logic design.

5. Read the EVC/demo-13/t/t.c file.

This file includes the header files created from step 4 and will be used to initialize the EVC, download the design, and then transfer data to/from the EVC. Type "make" to compile the file and then execute the results. The 'C' code for t.c is in

the appendix. Please focus on the EVC SBus interface process and the using of EVC functions:

1. `address_array= EVCdownload(t,0,3);`

This tells you to download the design t into EVC at Slot 3 `address_array[0]=i%2;`
The above line of code will write the values onto the data lines and toggle EV-CADR0 of the Hardware Object.

2. `EVCreset(3);` Resets the EVC.

Please refer to the 'EVC Sbus interface Reference' in Chapter 2.3 and 'EVC Software Reference' in the following for more detail.

EVC Software Reference

The EVC includes the following software which are used for download hardware design process (C code).

Command: The following commands are provided in the EVC/bin directory:

1. `evcreset`: Resets the EVC.

Usage: `evcreset 3` where the number indicates the SBus slot #.

2. `r2h`: Reads a rawbits file (.rbt) and creates a 'C' header file (.h) of the same name.

Usage: `r2h filename`

Functions: The following functions can be used in the 'C' code. They all take a parameter indicating the SBus slot # (It is 3 in our case):

1. `EVCdownload`: Download configurations to the EVC. Usage: `EVCdownload(<array name>,<device identifier 0=XC4010 or 1=XC4013,<SBus slot #>)`, Example: `EVCdownload(fib,0,3)`

2. `EVCreset`: Resets the EVC. Usage: `EVCreset(<SBus slot #>)`, Example: `EVCreset(3)`

3. `EVCrbtrig`: Toggles RDBK.TRIG on the XC40xx device. Usage: `EVCrbtrig(<SBus slot #>)`, Example: `EVCrbtrig(3)`

4. `EVCreadback`: Returns a single bit of the readback data stream from the XC40xx device. Usage: `EVCreadback(<SBus slot #>)`, Example: `EVCreadback(3)`

APPENDIX B

Synthesizing a Xilinx FPGA Using the Synopsys FPGA Compiler

In this section, an example is given to help tour the Synopsys FPGA Compiler CAD tool [4]. Synopsys FPGA Compiler is used as design tool to implement the reconfigurable computing system, the contents of Chapter 3 are based on this tutorial.

Example

Here is an 8-bit counter that is synthesized into Xilinx XC4000 device by using Synopsys FPGA compiler [4]. This example will help to understand the design process by using Synopsys FPGA Compiler Tools.

Set Up

1. You need to create a working directory: Create a directory called count8 and change to that directory. `sun1:/u4/user % mkdir count8`
`sun1:/u4/user % cd count8`
2. Copy over some files necessary to complete this tutorial:
`/u4/user/count8 % cp -r /usr/xact/tutorial/synopsys/fpga/x4000/vhd .`
`/u4/user/count8 % cp zijun/count8/.synopsys_dc.setup .`
3. Use one of your favorite text editors to open the file: `.synopsys_dc.setup`
4. Change my login name 'zijun' to yours, and save the file.

Invoking The Design Analyzer

1. From the count8 directory, run the Synopsys Design Analyzer in the background by entering the following command.
`sun1:/u4/user/count8 % design_analyzer &`
2. Verify that your Synopsys options were set correctly:

Click Setup —> Defaults...

3. Verify that your settings match the following setting.

search_path = \$SYNOPSYS/libraries/syn \$HOME/count8/vhd/u4/your_login_name/count8/vhd/

Note: Replace your login name to /u4/your_login_name/count8/vhd/ if your_login_name here is not yours.

link_library = xprim_4005-5.db xprim_4000-5.db xgen_4000.db xio_4000-5.db xfpga_4000-5.db

target_library = xprim_4005-5.db xprim_4000-5.db xgen_4000.db xio_4000-5.db xfpga_4000-5.db

symbol_library = xc4000.sdb

4. Change anything which does not match in step 3. Select OK and then Cancel to close the window.

5. Select File —> Quit, and then OK to close the design analyzer window.

Analyze The Design File

1. sun1:/u4/user/count8 % design_analyzer
2. Select File —> Analyze... from the Design Analyzer menu.
3. Use the left mouse button to click once on vhd/ and then click OK.
4. Use the left mouse button to click once on count8.vhd.
5. Click OK.
6. Click Cancel.

Creating The Design File

1. Goes to File—> Elaborate... from the Design Analyzer menu.
2. Scroll the library list and click on work.
3. Click once on count8 (BEHAVIORAL) in the design field.
4. Click OK.
5. Click Cancel after it finishes reading design.

Insert I/O Buffers

Note: Count8 is one-level design.

1. Click the left mouse button on the count8 icon.
2. Click on the down arrow icon to display the design in Symbol View. The down arrow icon is located at left of the Design Analyzer menu.
3. Select the CLEAR, CLOCK and ENABLE input ports by clicking on one with

the left mouse button, and the other two with the middle mouse button. A dotted rectangle indicates that the ports are selected.

4. Select Attributes —> Optimization Directives —> Input Port... from the menu.
5. Click on the box next to the Port is Pad.
6. Select Apply.
7. Click on Cancel.
8. Select the COUT [7:0] bus by clicking on it with the left mouse button.
9. Select Attributes —> Optimization Directives —> Output Port... from the Design Analyzer menu.
10. In the Bus Selector window, click cancel.
11. In the Out Port Attributes dialog box, click on the box labeled 'Port is Pad'.
12. Click on Apply.
13. Click on Cancel.
14. Select Setup —> Command Window... from the the Design Analyzer menu.
15. At the Design analyzer prompt in the Command window, type insert_pads and then return.

Timing

1. Click CLOCK pin by using left mouse button.
 2. Click the following from the attributes menus.
Attributes —> Clocks —> Specify ...
 3. Click on Apply and then on Cancel.
- Look at the symbol of count8. A red waveform appears above the CLOCK pin to indicate the setting of a timing constraint.

Compiling The Design

1. Click Tools —> FPGA Compiler ... from the Design Analyzer menu.
2. Click on Optimization...
3. Make sure the Map Design box is shaded and the Map Effort is Medium.
4. Click OK.
5. Once the design is compiled , click Cancel to close the Compile Log window.
6. Select Cancel to close the FPGA Compiler window.

Evaluating The Results

1. Creating a schematic of the design by selecting the gate picture icon on the left side of the Synopsys Design Analyzer window.
2. Click on the up arrow icon to switch to the Designs View.
3. click on the count8 icon.
4. Select Tools —> FPGA Compiler... from the Design Analyzer menu.
The FPGA Compiler window appears.

5. Select Report....
6. Click the boxes next to FPGA Resources and Timing.
7. Select Apply.
8. Use the scroll bar in the Report Output window to view the design statistics.
9. Select Cancel to close the Report output box.
Note: Do not close the Report dialog box.

10. In the Analysis Report section of the Report dialog box, click on the box to the left of Timing with the left mouse button until this box is shaded.
11. Deselect (do not highlight) the FPGA Resources box.
12. Select Apply. The Report Output window opens. The results reported are worst-case timing delay estimates. The final results cannot be determined until after you run PPR.

13. Select Cancel when reviewing the Report Output window.
The following steps will save the estimated area and timing results to a report file.

14. Locate the Send Output to field at the bottom of the Report dialog box.
15. Select File.
16. Place your cursor in the File field.
17. Double-click to highlight the default report file name.
18. Type count8.timing
19. Select Apply —> Cancel to close the Report dialog box...

Note: Do not close the FPGA Compiler window.

Saving The Design

1. Select the count8 icon with the left mouse button.
2. Select the File —> Save from the Design Analyzer menu.
The system saves the file as count8.db
3. in the FPGA Compiler window, select FPGA Cells to Gates
4. Select Cancel to close the FPGA Log window.
5. Select Cancel to close the FPGA Compiler window.

Setting The Design Part Type

To select a particular part for the count8 design, type the following command at the Design Analyzer prompt in the Command window:

`set_attribute count8 "part" -type string "4005pc84-5" and return.`

To allow the XACT software more freedom during placement and routing, type the following command at the Design Analyzer prompt in the Command window.
`set_attribute find (design,"*") "xnfout_use_blkname" -type boolean false and return.`

Saving The Design File As An sxnf File

1. Click File —> Save as...
2. Click in the field next to File Name.
3. Change the .db extension to .sxn.
4. Click on the bar next File Format.
5. Select XNF.
6. Make sure the Save All Designs in Hierarchy box is shaded (highlighted).
7. Select OK.

Exit The Design Analyzer

1. Select File —> Quit
2. Click OK to exit.

Note: Although one can put most of above commands into a script file(see appendix), the above steps will help us to understand how Synopsys FPGA Compiler works. After one is familer with all above steps, he can create his own script file which will save time. The other Synopsys FPGA Compiler related examples here are executed by script files.

Running XILINX XACT

You are now done with the Synopsys FPGA Compiler and are ready to use the XACT Development System. Do the following command at your unix prompt.

`sun1:/u4/user/count8 % xmake count8.sxn`

This will give you the result all the way down to the .bit file of your design. You could check the RPT file to view the PPR and other information of the design. To obtain detailed post-placement and post-routing timing information about your design, do the following command at Unix prompt.

```
sun1:/u4/user/count8 % xdelay count8
```

This will create a short report file — count8.dly, open this file, check and compare the difference of the timing result which you get from count8.timing. The wire-load models and the mapping of the X-BLOX modules account for the difference in delay from that of count8.timing.

APPENDIX C

VHDL and C Template Files of Using the EVC

Template Files

```
-----  
-- Top Level Desgin for Hamming Encoder using the EVC  
-- date: 2-Feb. 96  
-- design: Zijun Shen  
-----
```

```
library ieee;  
use ieee.std_logic_1164.all;  
use IEEE.std_logic_unsigned.all;  
  
entity encoder_2 is  
  
    port( sclk : in std_logic;  
          sb_dir : in std_logic;  
          b_sel : in std_logic;  
          b_as : in std_logic;  
          b_pa16 : in std_logic;  
          b_pa15 : in std_logic;  
          b_pa6 : in std_logic;  
          b_pa5 : in std_logic;  
          b_pa4 : in std_logic;  
          b_pa3 : in std_logic;  
          b_pa2 : in std_logic;  
          b_pa1 : in std_logic;  
          b_pa0 : in std_logic;  
          b_siz : in std_logic_vector(2 downto 0);  
          ack : out std_logic_vector(2 downto 0);  
          sbio : inout std_logic_vector(31 downto 0));  
  
end encoder_2;
```

architecture rtl of encoder_2 is

component BUFGP_F

```
    port( o : out std_logic;  
          i : in std_logic);  
end component;
```

component module

```
    port( b_sclk : in std_logic;  
          sdata : out std_logic_vector(31 downto 0);  
          evcadr : in std_logic_vector(15 downto 0);  
          sdati : in std_logic_vector(31 downto 0));  
end component;
```

component sbus_if

```
    port( b_sclk : in std_logic;  
reset : in std_logic;  
          sdata : in std_logic_vector(31 downto 0);  
          evcadr : out std_logic_vector(15 downto 0);  
          sdati : out std_logic_vector(31 downto 0);  
sb_dir : in std_logic;  
b_sel  : in std_logic;  
b_as   : in std_logic;  
b_pa16 : in std_logic;  
b_pa15 : in std_logic;  
b_pa6  : in std_logic;  
b_pa5  : in std_logic;  
b_pa4  : in std_logic;  
b_pa3  : in std_logic;  
b_pa2  : in std_logic;  
b_pa1  : in std_logic;  
b_pa0  : in std_logic;  
b_siz  : in std_logic_vector(2 downto 0);  
ack    : out std_logic_vector(2 downto 0);  
sbio   : inout std_logic_vector(31 downto 0));  
end component;
```

```
signal evcadr : std_logic_vector(15 downto 0);  
signal sdati  : std_logic_vector(31 downto 0);  
signal sdata  : std_logic_vector(31 downto 0);  
signal b_sclk : std_logic;  
signal zero   : std_logic;
```

```

begin

    inst0: BUFGP_F port map (b_sclk, sclk);

    inst1: sbus_if port map (b_sclk, zero, sdata, evcadr, sdati, sb_dir, b_sel,
                             b_as, b_pa16, b_pa15, b_pa6, b_pa5, b_pa4, b_pa3,
                             b_pa2, b_pa1, b_pa0, b_siz, ack, sbio);

    inst2: module port map (b_sclk,sdata, evcadr, sdati);

    zero <= '0';

end rtl;

-----
-- module.vhd, user's define module
-- date: 31-Jan. 96
-- design: Zijun Shen
-----

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity module is
    port( b_sclk   : in std_logic;
          sdata    : out std_logic_vector(31 downto 0);
          evcadr   : in std_logic_vector(15 downto 0);
          sdati    : in std_logic_vector(31 downto 0));
end module;

architecture rtl of module is
    -----BOX-----
    component ham_code
        port(   RESET      : in std_logic ;
              DATA_IN    : in std_logic_vector(7 downto 0);
              DATA_OUT    : out std_logic_vector(11 downto 0));
    end component;

    component evcreg1
        port( b_sclk   : in std_logic;

```

```

        regin      : in std_logic_vector(23 downto 0);
        evcadr     : in std_logic_vector(15 downto 0);
        sdato      : out std_logic_vector(31 downto 0));
end component;

signal dato      : std_logic_vector(23 downto 0);
begin
U1:  ham_code
      port map ( RESET = > evcadr(0),
        DATA_IN = > sdati(7 downto 0) ,
        DATA_OUT = > dato(11 downto 0));
U2:  ham_code
      port map ( RESET = > evcadr(0),
        DATA_IN = > sdati(15 downto 8),
        DATA_OUT = > dato(23 downto 12));
U3: evcreg1
      port map ( b_sclk, dato, evcadr,sdato);
-----
end rtl;

-----
-- Hamming Encoder
-- date: 31-jan-96 15:06:33
-- zijun
-----

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ham_code is
port(
    RESET      : in std_logic ;
    DATA_IN   : in std_logic_vector(7 downto 0);
    DATA_OUT   : out std_logic_vector(11 downto 0));

end ham_code;

architecture behavior of ham_code is

begin
    process(RESET,DATA_IN)

```

```

begin
  if RESET = '0' then
    DATA_OUT<="000000000000";
  else
    DATA_OUT(0)<=DATA_IN(0) XOR DATA_IN(1)
    XOR DATA_IN(3) XOR DATA_IN(4)
    XOR DATA_IN(6);
    DATA_OUT(1)<=DATA_IN(0) XOR DATA_IN(2)
    XOR DATA_IN(3) XOR DATA_IN(5)
    XOR DATA_IN(6);
    DATA_OUT(2)<=DATA_IN(0);
    DATA_OUT(3)<=DATA_IN(1) XOR DATA_IN(2)
    XOR DATA_IN(3) XOR DATA_IN(7);
    DATA_OUT(4)<=DATA_IN(1);
    DATA_OUT(5)<=DATA_IN(2);
    DATA_OUT(6)<=DATA_IN(3);
    DATA_OUT(7)<=DATA_IN(4) XOR DATA_IN(5)
    XOR DATA_IN(6) XOR DATA_IN(7);
    DATA_OUT(8)<=DATA_IN(4);
    DATA_OUT(9)<=DATA_IN(5);
    DATA_OUT(10)<=DATA_IN(6);
    DATA_OUT(11)<=DATA_IN(7);

    end if;
  end process;
end behavior;

```

```

/*****
/* C CODE FOR TWO PARALLEL HAMMING CODE ENCODERS */
/* The two Hamming encoders working in parallel */
/* EVCdownload--Hardware Design */
/* Ham()--Software Equivalent for Hardware Design */
/* Date: Feb. 16, 96 */
/* Written by: Zijun Shen */
*****/
#include
#include
#include
#include
#include
#include "encoder_2.h"

```

```

#define MAX    10000000
#define NUMREPS    100

float    timer();
int (*EVCdownload());
int EVCreset();

struct in_byte {
    unsigned char inbyte[3];
} inbytes;

struct in_bit {
    unsigned int inbit_23 : 1;
    unsigned int inbit_22 : 1;
    unsigned int inbit_21 : 1;
    unsigned int inbit_20 : 1;
    unsigned int inbit_19 : 1;
    unsigned int inbit_18 : 1;
    unsigned int inbit_17 : 1;
    unsigned int inbit_16 : 1;
    unsigned int inbit_15 : 1;
    unsigned int inbit_14 : 1;
    unsigned int inbit_13 : 1;
    unsigned int inbit_12 : 1;
    unsigned int inbit_11 : 1;
    unsigned int inbit_10 : 1;
    unsigned int inbit_9 : 1;
    unsigned int inbit_8 : 1;
    unsigned int inbit_7 : 1;
    unsigned int inbit_6 : 1;
    unsigned int inbit_5 : 1;
    unsigned int inbit_4 : 1;
    unsigned int inbit_3 : 1;
    unsigned int inbit_2 : 1;
    unsigned int inbit_1 : 1;
    unsigned int inbit_0 : 1;
};

struct out_bit {
    unsigned int outbit_23 : 1;
    unsigned int outbit_22 : 1;
    unsigned int outbit_21 : 1;
    unsigned int outbit_20 : 1;
    unsigned int outbit_19 : 1;
};

```

```

        unsigned int outbit_18 : 1;
        unsigned int outbit_17 : 1;
        unsigned int outbit_16 : 1;
        unsigned int outbit_15 : 1;
        unsigned int outbit_14 : 1;
        unsigned int outbit_13 : 1;
        unsigned int outbit_12 : 1;
        unsigned int outbit_11 : 1;
        unsigned int outbit_10 : 1;
        unsigned int outbit_9 : 1;
        unsigned int outbit_8 : 1;
        unsigned int outbit_7 : 1;
        unsigned int outbit_6 : 1;
        unsigned int outbit_5 : 1;
        unsigned int outbit_4 : 1;
        unsigned int outbit_3 : 1;
        unsigned int outbit_2 : 1;
        unsigned int outbit_1 : 1;
        unsigned int outbit_0 : 1;
};

union u_in {
    unsigned long int EVCi:24;
    struct in_bit inbits;
    struct in_byte inbytes;
} uin;

union u_out {
    unsigned long int EVCo:24;
    struct out_bit outbits;
} uout, EVCoout;

/*-----BOX-----*/
/* Software Equivalent Function for the Hamming encoders,
   it mimics the ham_code.vhd */
void HAM(struct in_bit *inbits, struct out_bit *outbits)
{
    outbits->outbit_0 = inbits->inbit_0 ^
        inbits->inbit_1 ^ inbits->inbit_3 ^
        inbits->inbit_4 ^ inbits->inbit_6;
    outbits->outbit_1 = inbits->inbit_0 ^
        inbits->inbit_2 ^ inbits->inbit_3 ^
        inbits->inbit_5 ^ inbits->inbit_6;

```

```

outbits->outbit_2 = inbits->inbit_0;
outbits->outbit_3 = inbits->inbit_1 ^
    inbits->inbit_2 ^ inbits->inbit_3 ^
    inbits->inbit_7;
outbits->outbit_4 = inbits->inbit_1;
outbits->outbit_5 = inbits->inbit_2;
outbits->outbit_6 = inbits->inbit_3;
outbits->outbit_7 = inbits->inbit_4 ^
    inbits->inbit_5 ^ inbits->inbit_6 ^
    inbits->inbit_7;
outbits->outbit_8 = inbits->inbit_4;
outbits->outbit_9 = inbits->inbit_5;
outbits->outbit_10 = inbits->inbit_6;
outbits->outbit_11 = inbits->inbit_7;

outbits->outbit_12 = inbits->inbit_8 ^
    inbits->inbit_9 ^ inbits->inbit_11 ^
    inbits->inbit_12 ^ inbits->inbit_14;
outbits->outbit_13 = inbits->inbit_8 ^
    inbits->inbit_10 ^ inbits->inbit_11 ^
    inbits->inbit_13 ^ inbits->inbit_14;
outbits->outbit_14 = inbits->inbit_8;
outbits->outbit_15 = inbits->inbit_9 ^
    inbits->inbit_10 ^ inbits->inbit_11 ^
    inbits->inbit_15;
outbits->outbit_16 = inbits->inbit_9;
outbits->outbit_17 = inbits->inbit_10;
outbits->outbit_18 = inbits->inbit_11;
outbits->outbit_19 = inbits->inbit_12 ^
    inbits->inbit_13 ^ inbits->inbit_14 ^
    inbits->inbit_15;
outbits->outbit_20 = inbits->inbit_12;
outbits->outbit_21 = inbits->inbit_13;
outbits->outbit_22 = inbits->inbit_14;
outbits->outbit_23 = inbits->inbit_15;

}
/*-----End BOX-----*/

/* main function */
main()

```

```

{
    float time1, time2, time3, time4, tmp;
    int i, j ;
    unsigned long int inEVC;
    unsigned long int outEVC;

    register int *write_to_evc, *read_from_evc;

    /* Download Hardware Object configurations to the EVCis,
       setup timer for this process */

    time1 = timer();
    write_to_evc = EVCdownload(encoder_2, 0, 3);
    time2 = timer();
    read_from_evc = write_to_evc + 1;

    /* Compare the Results between Software solution and
       Hardware solution */

    for (i=1; i < NUMREPS; i+=2) {
        uin.inbytes.inbyte[0] = 0;
        uin.inbytes.inbyte[1] = i;
        uin.inbytes.inbyte[2] = i;

        HAM((struct in_bit *) &(uin.inbits),
            (struct out_bit *) &(uout.outbits));

        *write_to_evc = uin.EVCi;
        EVCout.EVCo = *read_from_evc;

        printf("Software-Input: %x Software-Output: %x\n",
            uin.EVCi, uout.EVCo);
        printf("Hardware-Input: %x Hardware-Output: %x\n",
            uin.EVCi, EVCout.EVCo);
        printf("\n");
    } /* end for(), end Compare the Results */

    printf("Hardware Design's Download time is
           %f seconds\n", time2-time1);
    printf("\n");
    printf("EVC and Software Perform Hamming's
           Code Encoder Algorithem: %d strings ", 8*MAX );
    printf("\n");

```

```

printf("\n");

/* Setup timer for the Hardware--EVC1 caculation process */

time3 = timer();
for (i=1; i < MAX; i+=1)    {
    uin.inbytes.inbyte[0] = 0;
    uin.inbytes.inbyte[1] = 0;
    uin.inbytes.inbyte[2] = i;

    *write_to_evc = uin.EVCi;
    EVCoout.EVCo = *read_from_evc;

} /* end for() */

time4 = timer();
printf("Hardware execute time is: %f seconds\n",
      time4-time3);

printf("\n");
tmp = time4-time3+time2-time1;
printf("Hardware execute time + Download time is:
%f seconds\n",
      tmp);

printf("\n");
time1 = timer();

/* Setup timer for Software caculation process */

for (i=1; i < MAX; i+=1)    {
    uin.inbytes.inbyte[0] = 0;
    uin.inbytes.inbyte[1] = 0;
    uin.inbytes.inbyte[2] = i;

    HAM((struct in_bit *) &(uin.inbits),
        (struct out_bit *) &(uout.outbits));

} /* end for() */

time2 = timer();
printf("Software execute time is:
      %f seconds\n", time2-time1);
printf("\n");
printf("EVC saves %f seconds\n", time2-time1-tmp);
printf("\n");

```

```
} /* end main() */
```

```
float timer()  
{  
    float timer;  
    float secs,tarray[2];  
    struct rusage rused;  
  
    getrusage(RUSAGE_SELF,&rused);  
    timer = rused.ru_utime.tv_sec  
           + rused.ru_stime.tv_sec  
    + .000001*( rused.ru_utime.tv_usec  
               + rused.ru_stime.tv_usec);  
    return(timer);  
}
```

VITA

Zijun Shen was born in Changshia City, Hunan Province, Mainland China on April 24, 1964. After graduating from the 15th High School of Wuhan City in 1983, he entered Wuhan Institute of Technology for his undergraduate studies. He was awarded a Bachelor of Engineering degree in Electrical Engineering in July of 1987. He had worked as an electrical engineer in the Computer Center, Central-South China Architecture Design Institute since August, 1987. In August of 1994, he joined the University of Tennessee for the graduate program in Electrical and Computer Engineering Department. He was awarded a Master of Science degree in Electrical Engineering in August of 1996.