

EFFICIENT HARDWARE IMPLEMENTATION
OF FINITE IMPULSE RESPONSE FILTERS
IN FIELD PROGRAMMABLE
GATE ARRAYS

By

KAH-HOWE TAN

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

2000

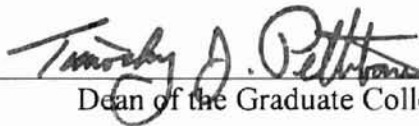
Submitted to Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2002

EFFICIENT HARDWARE IMPLEMENTATION
OF FINITE IMPULSE RESPONSE FILTERS
IN FIELD PROGRAMMABLE
GATE ARRAYS

Thesis Approved:



Thesis Adviser



Dean of the Graduate College

Dedication

*To
My parents*

ACKNOWLEDGMENTS

I would like to express my sincere thanks to my advisor, Dr. Michael A. Soderstrand for giving me guidance in my study. I want to thank Dr. Michael A. Soderstrand and Department of Electrical and Computer Engineering for providing me with this research opportunity. I would like to extend my appreciation to Dr. Keith A. Teague and Dr. Louis G. Johnson for their invaluable knowledge and guidance.

I wish to give my specially appreciation to all of my colleagues in Digital Signal Processing and Communication (DSP&C) groups. I also want to thanks all my friends for supporting me and encouraging me all the time while I am pursuing my degree.

Lastly, I would like to thank my family who always support and encourage me.

TABLE OF CONTENTS

Chapter	Page
1. Introduction.....	1
1.1 Introduction.....	1
1.2 Contribution of thesis.....	2
1.3 Thesis Organization	3
2. Background	5
2.1 Digital system	5
2.2 Finite Impulse Response Filter	6
2.3 Canonical Signed Digit representation and Dempster-Macleod Technique.....	8
2.4 Field Programmable Gate Array.....	12
2.4.1 FPGA background	13
2.4.2 Common types of FPGAs.....	14
2.4.3 Xilinx FPGAs.....	15
2.4.4 Virtex series Architecture	18
3. Procedure for Implementation	21
3.1 Introduction.....	21
3.2 Running GUI program	23
3.3 Running TCL script in Synplify_Pro.....	24
3.4 Running Xilinx Design Manager.....	26
3.5 Implementation in XSV800 board.....	28
4. Translation of filter parameters	29
4.1 Introduction.....	29
4.2 The flow of FIR filter parameter translation.....	31
4.3 Computation of CSD data objects.....	35
4.4 Computation of DM data objects.....	38
5. VHDL coding for FIR filter design	41
5.1 Introduction.....	41
5.2 Basic HDL Terminology.....	43

Chapter	Page
5.3 Behavior coding of FIR filter.....	44
5.3.1 Coding on the filter parameters declaration.....	44
5.3.2 Coding on the filter behavior function file	45
5.3.3 Coding on the top level of FIR filter.....	48
6. Implementation and comparison.....	53
6.1 Introduction.....	53
6.2 Implementation	54
6.2.1 FIR filter implementation example.....	54
6.2.2 Simulation result	55
6.2.3 Experimental result	56
6.2.4 Hardware costs.....	58
6.3 Results and Comparisons.....	60
7. Conclusions and Future Work.....	63
7.1 Conclusions.....	63
7.2 Future Work	64
Bibliography.....	66
Appendix A.....	69
Appendixes B.....	74
Appendix B-1	74
Appendix B-2.....	75
Appendix B-3.....	77
Appendix C.....	79
Appendix D.....	79

LIST OF TABLES

Table	Page
2.1 Comparison of commercial FPGAs	15
4.1 An example of High-pass FIR filter coefficients.....	36
6.1 Hardware costs for an example of Low-pass filter	59
6.2 Specification for four different types of FIR filter.....	60
6.3 Hardware costs in Xilinx FPGAs for four different types of FIR filter.....	61
6.4 Comparison of hardware costs.....	62

LIST OF FIGURES

Figure	Page
2.1 Digital system block diagram	5
2.2 Filter structure: Transpose Direct Form.....	7
2.3 Filter structure: Direct Form	8
2.4 Binary representation of number 217 multiplied by input A.....	9
2.5 CSD representation of number 217 multiplied by input A	10
2.6 DM representation of number 217 multiplied by input A	11
2.7 Symmetrical array, hierarchical PLD, row-based and sea-of-gates.....	14
2.8 Overview of Virtex FPGAs series	16
2.9 Virtex architecture overview.....	19
2.10 Virtex Input/Output Block (IOBs).....	19
2.11 Virtex configurable logic block (CLBs)	20
3.1 Overall flow of implementation.....	22
3.2 Sample GUI program.....	23
3.3 Sample image of Synplify_Pro software	25
3.4 Xilinx Design Manager setting windows.....	26
3.5 Xilinx Design Manager.....	27
3.6 Xilinx Design Manager Implementation option window	27
4.1 Overall flow of the parameters translation.....	33
5.1 Top-to-bottom levels of VHDL files	42
5.2 Flow chart for FIR filter coefficient multiplier.....	47
5.3 Top level of FIR filter	48
5.4 Flow chart for top-level of FIR filter	49
5.5 Structure of transpose direct form FIR filter with odd number order.....	50
5.6 Structure of transpose direct form FIR filter with even number order	51
5.7 Basic structure of 1 st order FIR filter in transpose direct form	52
6.1 Overall structure of the hardware implementation	55
6.2 Comparison of original specification and simulation result	56
6.3 Interface between testing equipment and Xess XSV800 board.....	57
6.4 Comparison of Implementation data with original specification.....	58
7.1 Example of sharing filter coefficient	64

Chapter 1

1. Introduction

1.1 Introduction

Digital equipment such as mobile phones and audio equipment are widely used in today's world. However, the purity of signal will be a major consideration for designing these types of equipment. All of this equipment needs filters for filtering out unnecessary noise. There are two types of filter available, analog filters and digital filters. Comparing these, digital filters will have better controllability than analog filters, as analog filters need passive elements in their design. Passive elements vary in time, which make analog filters have less performance over time compared to digital filters. However, digital filters do have the advantage of simple hardware structure as we can represent digital filters easily by using adders and storage elements.

There are two different types of digital filters. These are Infinite Impulse Response (IIR) filters and Finite Impulse Response filters (FIR). Each type has its own advantages and disadvantages. FIR filter design is chosen for implementation in this thesis due to its unique filter characteristics such as linear phase, finite duration and symmetric filter coefficients. However, FIR filter design has the disadvantage of having a high filter order, which means the hardware requirement is generally greater. This becomes the major concern for the designers. This thesis discusses an efficient way to implement a FIR filter by using less hardware. The hardware saving is achieved by

introducing Canonical Sign Digit (CSD) approach and a new implementation approach based on factoring the CSD that was developed by Dempster and Macleod (the DM Technique). Field programmable gate arrays (FPGAs) are chosen to be the main hardware for implementation as FPGAs can be re-configured easily.

1.2 Contribution of thesis

A Matlab CSD and DM FIR filter design Graphic User Interface (GUI) program is created for FIR filter design targeting for Xilinx FPGAs. This GUI program consists of two parts; the first part is contributed by Husinga [1] and Leong [2], which is the design optimization of FIR filter using the Canonical Sign Digit representation (CSD) and the Dempster-Macleod (DM) technique in Matlab. The second part of this GUI program, which is addressed in this thesis, is to translate the optimized filter coefficients obtained using the GUI program into VHSIC (Very High-Speed Integrated Circuits) Hardware Description Language (VHDL) behavioral codes and further implement these codes in Xilinx FPGAs.

This thesis consists of three main objectives. The first objective is to translate the FIR filter parameters obtained from the optimized outputs of Leong's GUI program [2] into VHDL code that can represent a FIR filter. Chapter 4 in this thesis provides detailed discussion on the translation of the optimized filter parameters. The second objective of this thesis is to develop general VHDL codes for describing the specified behavior of a FIR filter. These VHDL codes are then combined with the parameter file that is translated from the Matlab program for implementation of FIR filter. These VHDL codes can be used for different technologies as they are behaviorally written. Chapter 5 includes more detailed discussion of these VHDL codes. The third objective of this thesis is to

implement the FIR filter design into the Xess XSV800 prototyping board to verify the validity of the design. This thesis mainly concentrates on Virtex XCV800HQ240-4 FPGAs but other Xilinx FPGAs technology can also be implemented using the same behavioral VHDL codes for FIR filter design. Chapter 6 discusses implementation of the FIR filter design into the Xess XSV800 board and comparison of hardware savings between the CSD representation and the DM technique is included.

1.3 Thesis Organization

Chapter 2 discusses the background material related to FIR filter design and FPGAs. The first two sections discuss the background of digital systems and FIR filters. Section 2.3 describes the background of the CSD representation and the DM technique. Section 2.4 includes the background of FPGAs and Xilinx FPGAs used for this thesis. Chapter 3 discusses the hardware implementation procedures on Xess XSV800 prototyping board, where this board uses Xilinx Virtex XCV800HQ240-4 chip as its FPGAs. These implementation procedures include hardware optimization in Leong's GUI program [2] and the interface between Leong's [2] optimized results with the FIR filter behavioral VHDL codes contributed in this thesis. Lastly, a bit stream file is created using implementation option in Xilinx Design Manager. This bit stream file can be downloaded into Xess XSV800 prototyping board for hardware implementation. Chapter 4 discusses the translation of the filter parameters. A Matlab routine is created to translate the parameters of the FIR filter into VHDL codes. Section 4.2 discusses the overall flow of the routine of how the translations take place. Section 4.3 and 4.4 discuss about how to translate the parameters into VHDL codes for the CSD representation and the DM technique respectively. Chapter 5 introduces the behavioral VHDL codes for

FIR filter design. A brief idea of basic HDL terminology is introduced in this chapter. Section 5.3 describes how the behavioral VHDL codes are done for FIR filter design. Chapter 6 covers an example of implementation as well as the results, which the implementation obtains from Xess XSV800 prototyping board. Section 6.3 includes a comparison of the hardware saving using CSD approach and DM approach. Chapter 7 concludes this thesis and future work is discussed in this chapter.

Chapter 2

2. Background

2.1 Digital system

Digital systems are sampled data systems that operate in discrete time instead of continuous time condition. Digital systems use computational processes and algorithms where a sequence of numbers acting as input signals and the system transforms the input sequence signals into a sequence of output signals according to the system characteristics [3]. However, in a real world application, most signals are in analog form. Therefore, analog to digital converters (ADC) and digital to analog converters (DAC) play an important role for digital system performance.

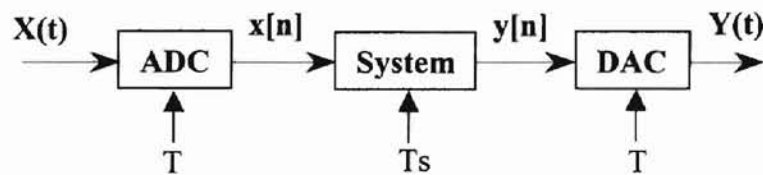


Figure 2.1 Digital system block diagram

Figure 2.1 shows the basic block diagram of a digital system. In this block diagram, continuous signals $X(t)$ are the input of the system. ADC converts the continuous signal's information into a number of digital sequences $x[n]$. $x[n]$ are sampled according to sample period (T) and $x[n]$ are used as the sequence inputs of the digital system. The digital system samples the signals at period T_s and transforms the input $x[n]$

to $y[n]$ based on the computational processes or algorithms defined for the digital system. A DAC is required for converting the digital sequence $y[n]$ back to analog signals $Y(t)$ so that the information is useful for the real world application. Also note that $y[n]$ is sampled at sample period T . ADC and DAC conversion is a complicated process in digital signal processing. There are several ADC available on the market such as parallel (flash) converter, successive approximation ADC, voltage-to-frequency ADC and integrating ADC. 'Delta sigma modulator with one bit DAC' is the most common DAC available. In this thesis, Xess XSV800 prototyping board is the main hardware that we are targeting. An audio CODEC chip (AK 4520a) on this board has the ability to perform the task of ADC and DAC conversion. The digital filter is considered as a digital system. FIR filters are the main target hardware in this thesis. The following section discusses the general idea of a FIR filter and filter structure, which we choose for hardware implementation.

2.2 Finite Impulse Response Filter

A digital Filter is one of the most basic blocks required in a digital signal processing system. A digital filter's impulse response can be either finite duration or infinite duration. The FIR filter that we focus on is a non-recursive, linear phase, constant group delay, symmetry and finite duration digital system [4]. Digital FIR filters design is easier to handle because FIR filters only have non-recursive property where the current output does not depend on the previous output. In this case, we can specify the number of bits we want for the input and output of the filter. The digital filter designer will have more controllability over the FIR filter design. However, one disadvantage of FIR filters over IIR filters is that FIR filters require a higher order in order to achieve certain filter specification. In this case, the higher order FIR filter requires more hardware to

implement. However, FIR filter design is often preferable for designers because of its unique characteristics. We can represent a general FIR filter in the following equation:

$$y(n) = \sum_{k=0}^{N-1} b_k x(n-k) \quad (2.1)$$

$x(n-k)$ is the input of the signal delay by k samples multiplied by the filter coefficient b_k and the output $y(n)$ is the discrete time at instance n . The summation runs from $k=0$ to $k=N-1$, where N is the number of taps for the filter [5].

There are many different structures to represent a FIR filter. Two common structures are the Transpose Direct Form and the Direct Form, both of which can perform a linear phase characteristic. Figure 2.2 shows a general Transpose Direct Form filter, where the input X is fed directly to each filter coefficients. The delay elements are located after filter coefficients. Figure 2.3 shows the Direct Form filter. In this figure, the X inputs need to pass through delay elements before multiplying the filter coefficients. Transpose Direct form structure has a higher sampling rate for implementation as Transpose Direct form structure consists of a shorter critical path compared to Direct Form structure. Therefore, Transpose Direct form structure is selected as the main filter structure so that the filter implementation will have a better sampling rate.

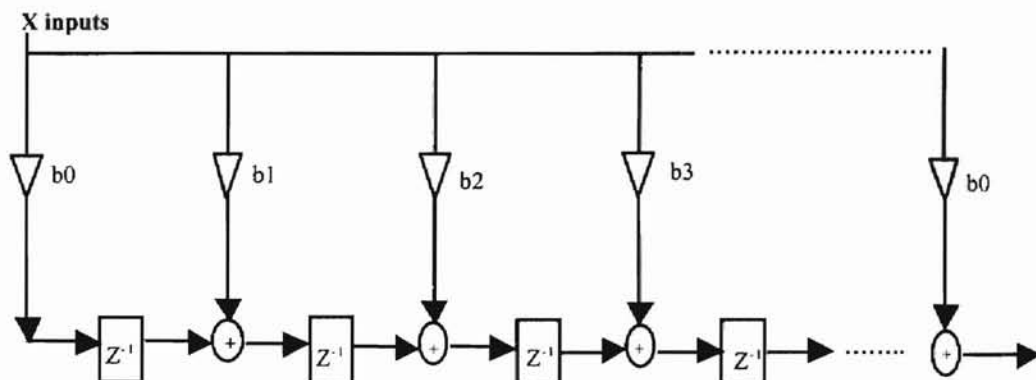


Figure 2.2 Filter structure: Transpose Direct Form

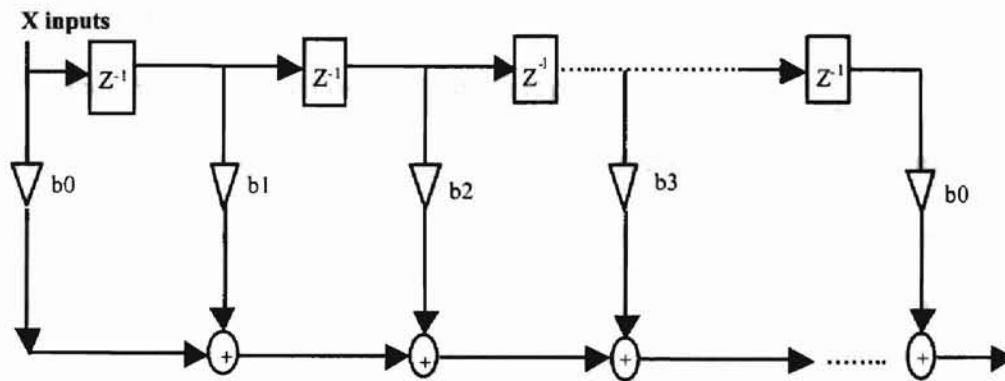


Figure 2.3 Filter structure: Direct Form

2.3 Canonical Sign Digit representation and Dempster-Macleod Technique

We can represent a FIR filter by simply using shifts, add/subtract arithmetic operations and delay elements in hardware sense. However, there are different ways to represent the filter coefficients. The most common way of representation is binary representation, where the set of representation is bounded under $\{0,1\}$. In binary representation, the main arithmetic required is shift and add. Later, the introduction of Canonical Sign Digit representation (CSD) gives more freedom in representing a number. The main contribution of CSD in hardware saving of FIR filter is CSD can represent a binary number using lesser number of non-zero bits. Signed digit (SD) number system is introduced to achieve the above purpose. The possible set of codes represented in CSD is $\{-1, 0, 1\}$. CSD representation assures that there are no two consecutive non-zero bits. In this case, the coefficient numbers are represented using fewer non-zero digits. The following example illustrates how the CSD representation works compared to binary representation. Take number 217 as the filter coefficient. In addition, we assume the filter coefficient and the filter input are a nine-bit number.

In binary, we can represent 217 as $2^0 + 2^3 + 2^4 + 2^6 + 2^7$ or 0 1 1 0 1 1 0 0 1 for a nine bit number. We need five non-zero bits to represent 217.

Take $A = 1 = 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1$ as the input of the filter coefficient multiplier. For binary representation, we start by shifting the input A by 0 time, and add the result by shifting the input A by 3 and then the output is added by shifting the input A 4 times. The output will continue adding the shifted input according to this equation: $(A \ll 0) + (A \ll 3) + (A \ll 4) + (A \ll 6) + (A \ll 7)$, where \ll represent the left shift. Figure 2.4 shows how the shifting and adding is done for this example. In this case, four adders are needed to represent number 217 multiplied by A.

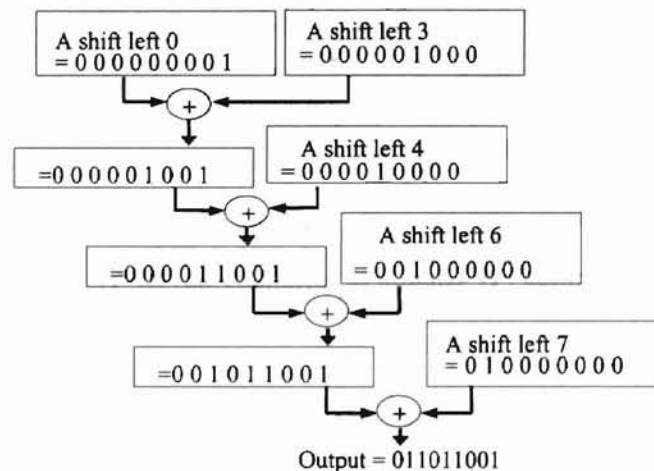


Figure 2.4 Binary representation of number 217 multiplied by input A

In CSD representation, we represent 217 as $2^0 - 2^3 - 2^5 + 2^8$ or 1 0 0 -1 0 -1 0 0 1 for a nine-bit number. We need only four non-zero bits to represent 217.

Taking the same A as input, for CSD representation, we can represent the multiplication by using this equation: $(A \ll 0) - (A \ll 3) - (A \ll 5) + (A \ll 8)$, where \ll represent the left shift. Figure 2.5 shows how the shifting and adding perform in this

CSD representation example. In this case, only one adder and two subtractors are needed to represent number 217 multiplied by the input A. We can see the major difference between CSD representation and binary representation is that CSD representation uses a subtractor. In this thesis, we assume that adder and subtractor requires the same amount of hardware to generate. Clearly, in this example, CSD representation saves one adder in hardware over the binary representation.

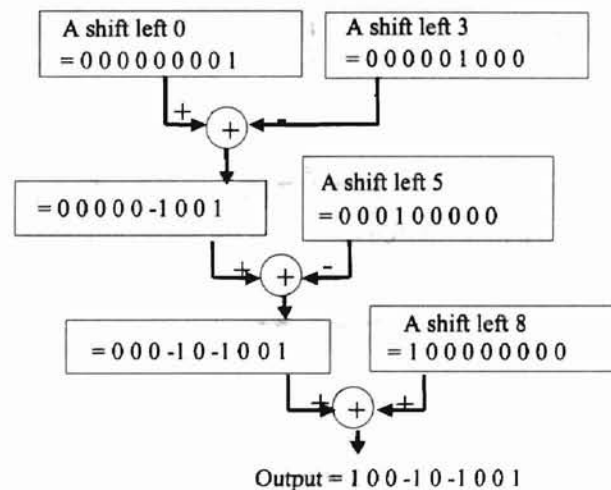


Figure 2.5 CSD representation of number 217 multiplied by Input A

CSD representation has been proven by many researchers regarding its potential in hardware saving. However, the work of Dempster and Macleod [6][7][8][9] has proven that some of the CSD number representations can give more hardware savings by using cascading. We refer to this technique as Dempster-Macleod (DM) technique. Taking the previous example number 217 as our filter coefficient, we can further illustrate how the DM technique can perform a better savings compared to the CSD representation. The first step in using the DM technique is to find the prime factor of the filter coefficient. For this example, the prime factors for 217 are 7 and 31. Hence, we can represent the nine-bit number of 7 as 00000100-1 and 31 as 00010000-1. Figure 2.6 shows how the shifting

and adding/subtracting is done using DM technique. In this case, only two subtractors are required to represent number 217 multiplied by the input A. We save one more adder hardware over the CSD representation.

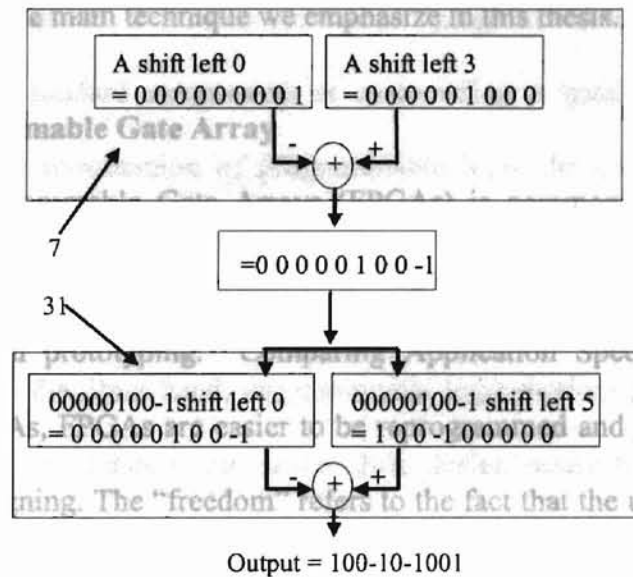


Figure 2.6 DM representation of number 217 multiplied by input A

In paper [10], we proved that comparing two to the power of eight numbers of combinations represented in binary, CSD and DM require the average number of adders 3.004, 1.8125 and 1.60156 respectively. The average savings achieved by using the CSD representation over binary is 39% and the DM technique over binary representation is 46%. There are even more obvious savings for two to the power of sixteen numbers of combinations. The CSD representation has 36% of savings over binary and the DM technique has 56% savings over binary. In Leong's [2] DM algorithm used in the GUI program, the DM technique will always perform an equal or better hardware saving over the CSD representation. Although the DM technique will always perform an equal or

better hardware saving than the CSD representation, we still give the user option to choose the CSD representation in the GUI program. The main reason for allowing the user to choose whether or not to involve the DM technique is because the DM technique relies on a cascade structure that can result in increased delay. However, the DM technique will be the main technique we emphasize in this thesis.

2.4 Field Programmable Gate Array

Field Programmable Gate Arrays (FPGAs) is commonly used nowadays. The reason for FPGAs being so popular is its re-configurable ability, which gives the designer more flexibility in prototyping. Comparing Application Specific Integrated Circuit (ASIC) with FPGAs, FPGAs are easier to be reprogrammed and give the designer more “freedom” in designing. The “freedom” refers to the fact that the user can implement any digital system he wants using the basic function in FPGAs. Altera, Actel, Quicklogic and Xilinx are several different types of FPGAs available commercially. Each of them has a different type of architecture, logic block type and programming technology. For instance, Xilinx FPGAs uses symmetrical array architecture, look-up table type of logic block and static RAM programming technology where Altera uses hierarchical-PLD architecture, PLD block type of logic block and EPROM programming technology. In this thesis, we are using Xilinx FPGAs as our target technology. Xess XSV 800 Virtex prototyping board (with Xilinx Virtex XCV800 HQ240 chip on board) is the main FPGA used for testing the design of FIR filter in this thesis. However, similar implementation can be done for other Xilinx FPGAs by just changing the technology parameters.

2.4.1 FPGA background

Programmable logic devices are devices that provide the designer the ability to reprogram the devices repeatedly. Field Programmable Gate Arrays (FPGAs) is a programmable device with a common purpose set of functions that can be reconfigured for different applications. Custom logic circuit design needs to be customizing at the board level using standard components or customizing at gate level using ASIC [11]. However, after the introduction of programmable logic devices, the designer has the ability to customize his design faster and more reliably. ASIC design may take up a certain amount of time before the manufacturer can produce the chip for the designer to test his design. On the other hand, programmable logic devices provide re-configurable ability that makes the designer customize their design easily by using Computer Aid Design (CAD) software. This is one of the reasons why FPGAs are commonly used device nowadays. FPGAs re-configurable ability gives the designer more flexibility in prototyping. The first programmable device that became popular on the market was Programmable Read Only Memory (PROM) from Harris and Monolithic Memories [12][13]. Later on, Monolithic Memories introduced field programmable PLA (FPLA). Then, the PAL architecture was developed by Monolithic Memories, which caused a great impact on the TTL market. There are two version of PROM available; they are Mask-Programmable Chip and Field Programmable Chip. Mask Programmable Chips are used for larger logic circuits and mostly programmed by the manufacturer. Field Programmable Chips are targeted for the regular user, which are more a cost effective solution for PLD.

The FPGA chip contains a number of identical logic cells that are interconnected in matrix form with wires and programmable switches. Each logic cell can have its own functionality depending on how the users program the cell. Users can specify the functionality by selecting the switches that interconnect the cells. One can design complex logic circuits by using the array of logic cell and interconnected switches.

2.4.2 Common types of FPGAs

Four different types of FPGAs are currently available commercially. Figure 2.7 [12] shows the graphic view of different FPGAs' classes; which are symmetrical arrays, hierarchical PLD, row based and sea-of-gates FPGAs.

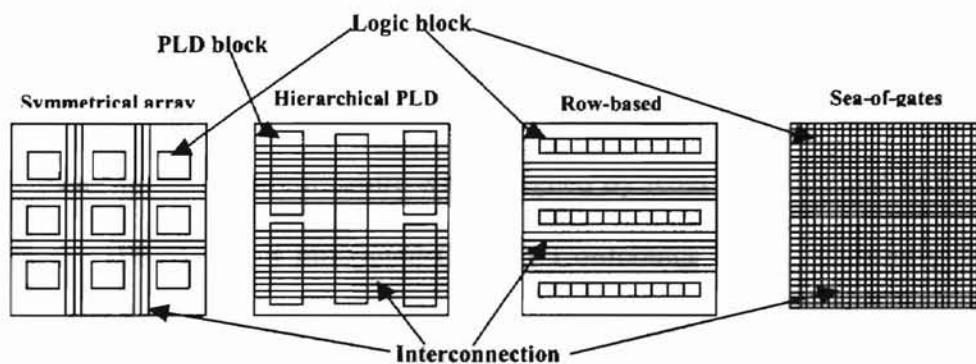


Figure 2.7 Symmetrical array, hierarchical PLD, row-based and sea-of-gates

Each of them has different kind of way to interconnect as well as the way to reprogram each block. As shown in Figure 2.7 [12], each class has a standard block and interconnection except that hierarchical PLD has only PLD and interconnection. In addition, there are four different technologies in use for the basic block. They are “static RAM (SDRAM) cells, anti-fuse, EPROM transistor and EEPROM transistors” [12]. SRAM technology uses pass-transistor, transmission gates or multiplexers for controlling

the cell. This technology can have a very fast circuit reconfiguration but the size of the chips is not compatible. On the other hand, anti-fuse technology is less expensive than SRAM technology but it can only be programmed once. EPROM has the advantage of reprogramming, which saves a lot in resources. Table 2.1 shows the comparison of four different commercial FPGAs' architectures, logic block types and programming technologies [12].

Company	Architecture	Logic Block Type	Programming Technology
Actel	Row-based	Multiplexer-based	Anti-fuse
Altera	Hierarchical PLD	PLD block	EPROM
Quick Logic	Symmetrical array	Multiplexer-based	Anti-fuse
Xilinx	Symmetrical array	Look-Up-Table (LUT)	Static RAM

Table 2.1 Comparison of commercial FPGAs

2.4.3 Xilinx FPGAs

“The first Xilinx architecture was designed by Ross Freeman and introduced in a paper in the 1986 Custom Integrated Circuits Conference” [13]. Currently there are a great number of devices developed by Xilinx. For example, Virtex II platform FPGAs, Virtex series, Spartan-II FPGAs, Spartan-XL FPGAs, Spartan FPGAs, XC4000XL/XLA, XC4000XV and others. Each of the FPGAs has its own specifications, which give the users more choices to select the devices suitable for their design purpose. In this thesis, we mainly concentrate on the Virtex series of Xilinx FPGAs.

Virtex Series: Xilinx Company introduced this series in 1998 and it is the first system with one million gates on the system. The architecture of the system is redefined to include a set of power features that deal with board level problems for high performance system design. Virtex E-series is another design introduced by Xilinx in

1999, which has three millions gates on the system. A Later Virtex EM device was introduced in 2000, which was the new FPGAs chip that used an advanced copper process. Figure 2.8 shows an overview of Virtex FPGAs series [14]. There are several interesting features that Virtex series have. The following is information obtained from the Xilinx official web page [14] about Virtex series FPGAs.

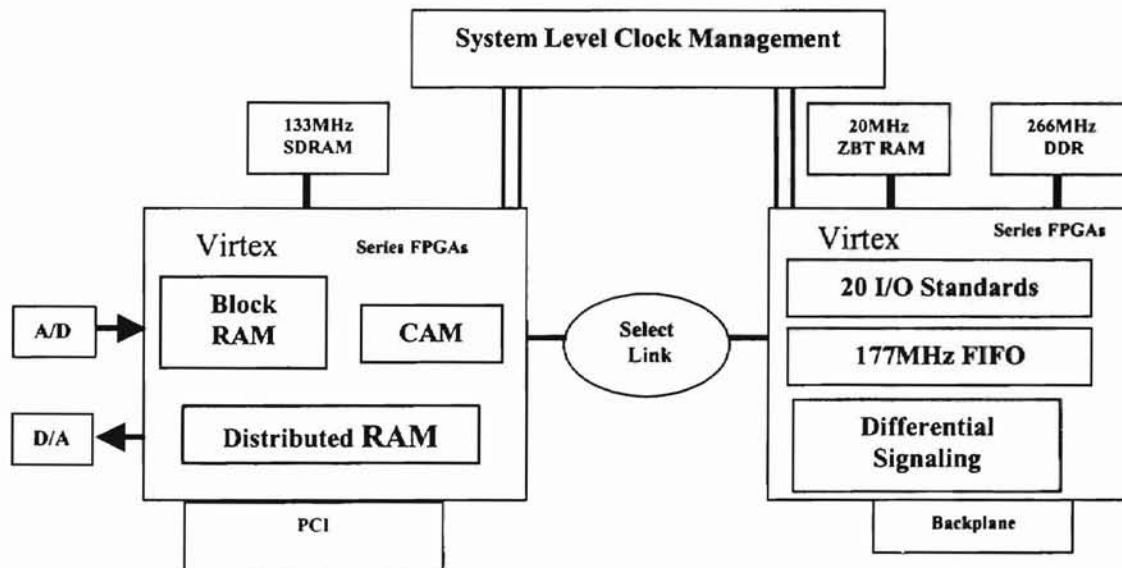


Figure 2.8 Overview of Virtex FPGAs series

The first feature is the System level and clock management, where there are eight high performance delay lock loops (DLLs) in Virtex series. DLLs are introduced so that higher bandwidth rate is allowed in this system. By using the DLL circuitry, precise synchronization of external and internal clocks can be performed. Next is the Static RAM (SRAM) in Virtex series. Static RAM will retain data bits if power is being supplied. On the other hand, dynamic RAM (DRAM) stores the data bits in the cells that contain a capacitor and a transistor. As SRAM does not have to be periodically refreshed, SRAM can have faster access to data. The drawback is that SRAM is more expensive. SRAMs are usually used for high performance speed systems. In Virtex, 133MHz of SRAM is

available. Besides Static RAM, Virtex has ZBT®/NoBL™ SRAM to prevent the latency between read and write data bus operation. ZBT and NoBL are high-speed synchronous pipeline SRAMs. By using these SRAM, one can maximize the bus bandwidth and data throughput. In this case, the system performance will be optimized. The main purpose of using these SRAMs is for networking, telecommunications applications, test equipment, DSP application, embedded memory, secondary cache and other such high-speed memory applications. One unique feature in Virtex is the Double Data Rate (DDR) SDRAM. DDR SDRAM is considered as one of the next generation SDRAM. The major difference between the DDR SDRAM with the regular RAM is that its speed is at least 200 MHz. DDR SDRAM can activate output on both the rising and falling edge of the system clock rather than on just the rising edge. In this case, there is potential to double the output speed. DDR has more advanced synchronization circuitry than regular SDRAM. Block SelectRAM is another feature available in Virtex series. It is also called dual-port RAM. Block SelectRAM is considered as one of the effective resources without sacrificing the existing distributed SelectRAM memory. Block SelectRAM memory is designed to synchronize with the system to perform accurate timing analysis. Each dedicated memory is available in 4K blocks and each block is operating as a fully synchronous true dual-port memory. Each port can read and write on independent clocks and can be configured differently. That is one of the reasons why RAM can be used as buffers for high-speed data streams and for funneling data to different width/speed combinations. Content Addressable Memory (CAM) in Virtex series is defined as “a storage array designed to quickly find the location of a particular stored value” [14]. The way CAM functions is CAM uses the input to compare with the data memory, and CAM

has the ability to find out if an input value matches one or more values stored in the array. Some features of CAM are described as follow. Firstly, CAM stores words in an array. The write mode of CAM is comparable, but the read mode is different. CAM input is looking for data instead of using data address line to pinpoint the storage value. When a match is found, the output is the address in the array. The distributed RAM of Virtex series is built on chips. In the early years, the FPGAs from Xilinx were using external SRAMs and DRAMs instead of on chip RAM. The 20 I/O standards allow system designers to interface with any device with zero translation delay. It also reduces system cost by eliminating external translators, and enabling the lowest power design possible. The FIFOs are used to buffer data on and off chip, or caches for high-speed parallel searches. Differential Signaling in Virtex is used because of comprehensive bandwidth requirements for high performance systems that exceed 100 Gbits per second. Differential signaling can have high bandwidth while reducing power, increasing noise immunity, and decreasing EMI emissions. Besides that, Virtex also has A/D, D/A conversion, Peripheral Component Interconnect, SelectLink and Backplane features.

2.4.4 Virtex series Architecture

This section is included to further explain the use of Xilinx Virtex series FPGAs. All this information is based on Xilinx Virtex product data sheets [15]. There are two major configurable elements in Virtex series; they are configurable logic blocks (CLBs) and input/output blocks (IOBs). CLBs are the basic elements used to perform basic functional blocks. IOBs are used for interfacing within the internal CLBs block with the package pins. Figure 2.9 is the reference figure from Xilinx Virtex product data sheets about the overview of Virtex architecture. As we can see, the BRAMs are located on two

sides of CLBs, where BRAMs provide more memory spaces for Virtex users. Versa ring is used to provide better routing resources for the devices. There are Delay Look Loops (DLLs) for better timing management.

Figure 2.10 shows the basic input/output block for Virtex series FPGAs. In basic IOBs, there are three storage elements, which can be used as a D flip-flops or level sensitive latches. Each of these storage elements uses the same clock signal but all three have their own clock enable signals. All of the IOBs pads have protected circuits to prevent electrostatic discharge or over-voltage transients.

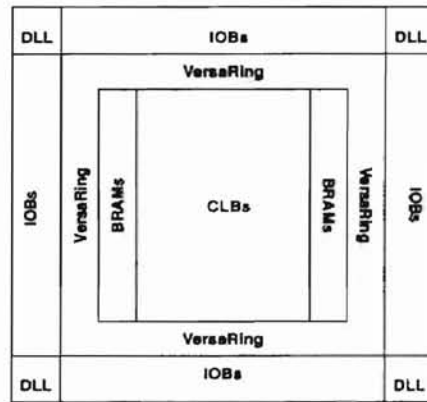


Figure 2.9 Virtex architecture overview

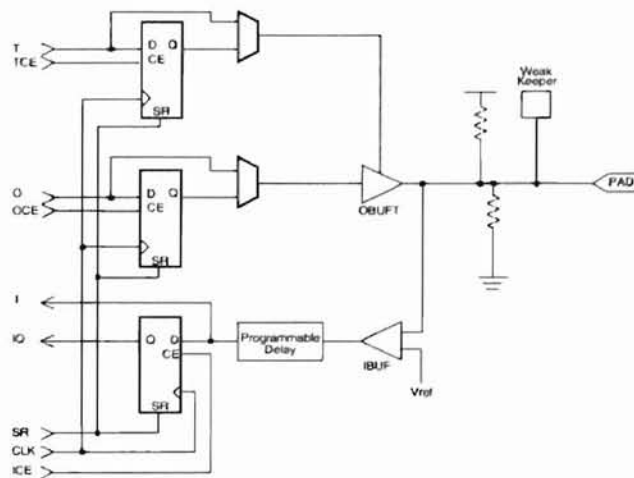


Figure 2.10 Virtex Input/Output Block (IOBs)

Configurable logic block (CLB) is the main element used in Virtex FPGAs. Figure 2.11 is a CLB in Virtex FPGAs. Each CLB has four logic cells; four carry logics and four storage elements. Each logic cell is a 4-input function generator. Virtex uses Look-up tables (LUTs) to implement the function generator. The storage elements can be used as edge-triggered D flip-flops or as level sensitive latches. Slice in Xilinx Virtex FPGAs is considered as half of a CLB. In this thesis, we count the hardware costs in Virtex FPGAs by using slice numbers.

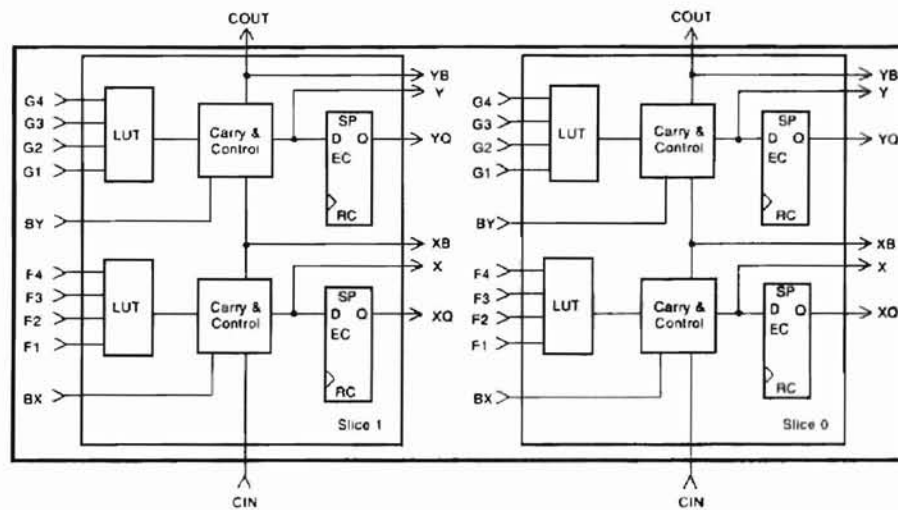


Figure 2.11 Virtex configurable logic block (CLBs)

Chapter 3

3. Procedure for Implementation

3.1 Introduction

Chapter 3 discusses the procedure of the hardware implementation of FIR filters using the VHDL interface written in thesis and Leong's GUI program [2]. Figure 3.1 shows the overall flow of the implementation. The gray portion in Figure 3.1 is contributed by Leong [2] and the white portion in Figure 3.1 is contributed in this thesis. In the white portion of Figure 3.1, *params.vhd* (Appendix B-1) package file is created in Matlab by using the optimized results from Leong's GUI program [2]. Details on how to generate the *params.vhd* (Appendix B-1) package file are included in Chapter 4. By combining the *params.vhd* (Appendix B-1) file with two FIR filter behavioral VHDL codes (*filt.vhd* and *rdfl.vhd*); we can describe a transpose direct form FIR filter. Chapter 5 discusses the FIR filter behavioral VHDL codes. We use Synplify_Pro synthesizer from Synplicity to synthesize the FIR filter behavioral VHDL codes. After synthesizing, Xilinx Design Manager software is used for implementation of the FIR filter. This implementation involves translation, mapping, place and route (PAR), timing simulation and configuration processes. A bit stream file is created after the implementation processes. This bit stream file is then downloaded to the Xilinx FPGAs for implementation. Chapter 6 provides more discussion on the implementation. The hardware implementation in this thesis is done using Xess XSV800 prototyping board

[16]. This board uses Xilinx FPGAs Virtex XCV800HQ240-4 as the main FPGAs chip. In the next few sections, we shall see an example of how FIR filter design is done starting from Leong's GUI program [2] to hardware implementation on XSV800 prototyping board. Although the hardware implementation in this thesis mainly uses Virtex XCV800HQ240-4 as the target technology, the user can also implement other Xilinx FPGAs technology by simply changing the technology module option in the TCL script and also the pin assignment in the constraints file.

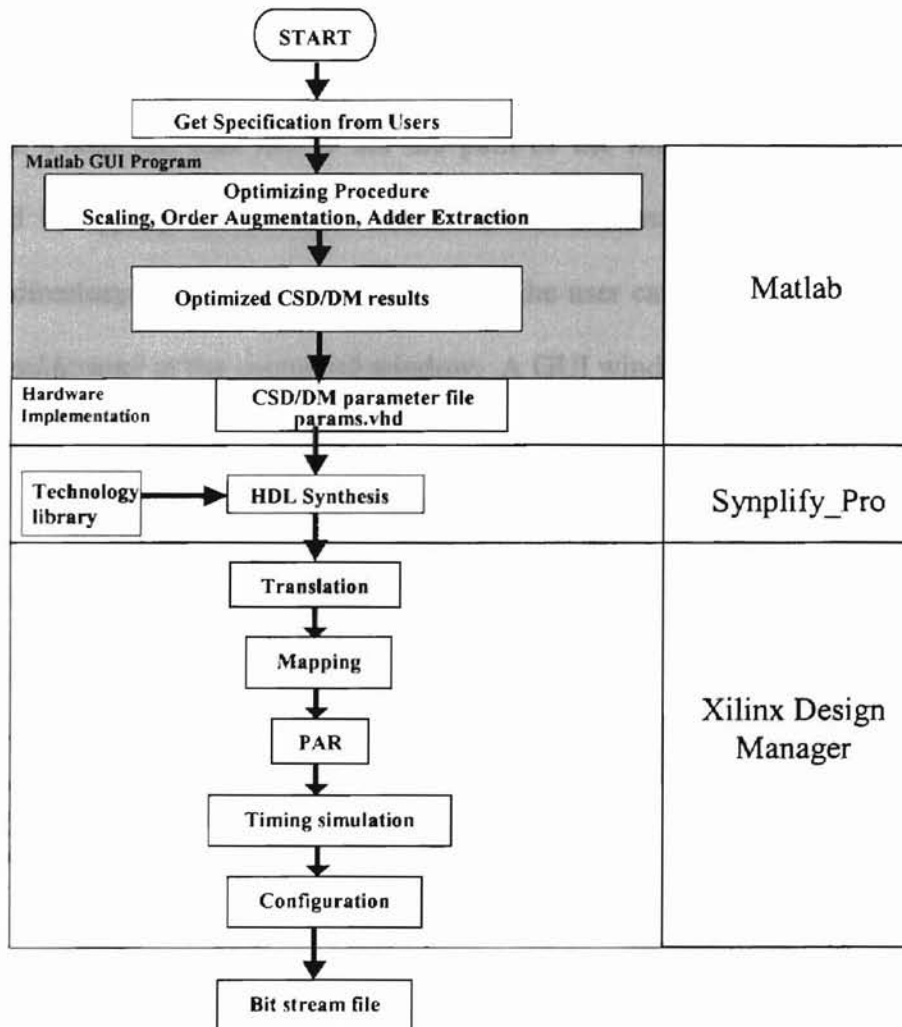


Figure 3.1 Overall flow of implementation

3.2 Running GUI program

Before starting the implementation, make sure that Leong's GUI program [2] and the VHDL behavioral coding written in this thesis is in the directory *c:\csddm*. The reason for doing this is that the VHDL filter parameters file (*params.vhd*) will be copied from the *c:\csddm* directory to *c:\csddm\FIRfilter* sub-directory. This sub-directory in *c:\csddm* directory contains the VHDL behavioral coding of the FIR filter and the VHDL coding of the Codec circuit, TCL script for running the synthesizer in Synplify_Pro and the constraints file for assigning the pins for top-level entity of VHDL coding.

The very first step for running Leong's GUI program [2] is to open the Matlab program. Then, the user has to set the path of the Matlab to *c:\csddm*. This can be achieved by typing *CD c:\csddm* at the Matlab command window, where *CD* is the change directory command in Matlab. Then, the user can run Leong's GUI program by typing *csddesign2* at the command window. A GUI window, which looks like Figure 3.2 pops up.

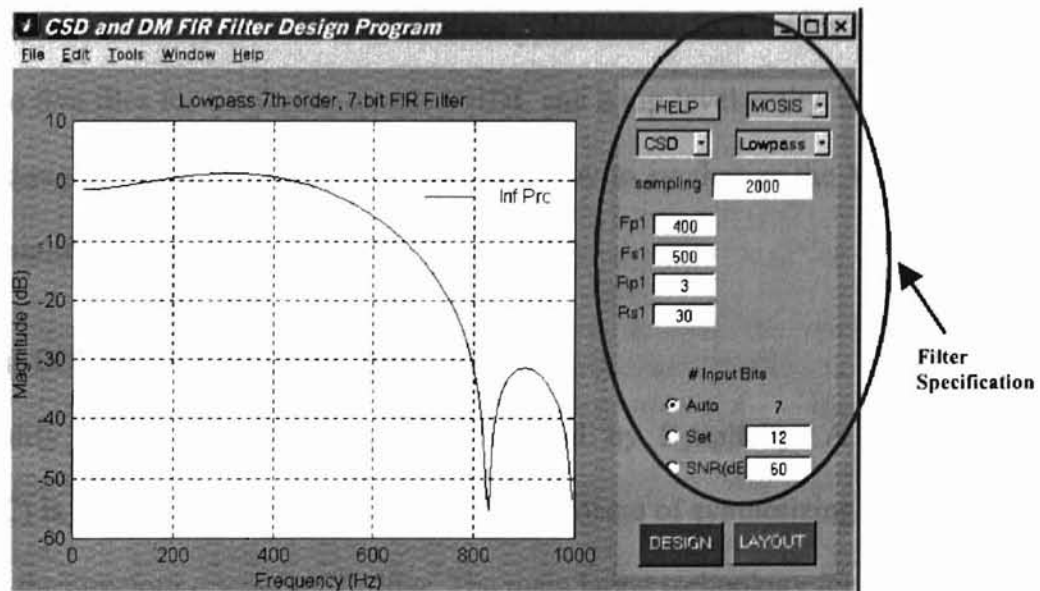


Figure 3.2 Sample GUI program

The next step is to enter the desired specification of the filter. The oval in Figure 3.2 shows the location where filter specifications need to be entered. These specifications include target technology; optimization method, sampling frequency, passband frequency and ripple, stopband frequency and ripple and input bit size. After entering the desired specifications, the user can activate the program by simply clicking the DESIGN button on the GUI program. The GUI program will then go through the CSD representation and the DM technique algorithm and stores the optimized filter parameters. A summary page will pop up after the optimization is done. The user can determine the optimized result by checking the summary page before going on to the hardware implementation. If the user is satisfied with the optimization result, he can start the hardware implementation by simply clicking the LAYOUT button on the GUI window. By doing this, Matlab will run the *layout.m* (Appendix A) file and create a *params.vhd* file in the *c:\csddm\FIRfilter* directory. The *layout.m* file contains an algorithm for generating the filter parameters VHDL file (*params.vhd*). All the files required for the FIR filter design are stored in the *c:\csddm\FIRfilter* directory. These files are three VHDL behavioral codes for FIR filter structure; the five files for the Codec circuit VHDL and a top-level entity VHDL file for the overall design. The following section discusses how to synthesize the VHDL codes using the Synplify_Pro synthesizing tool.

3.3 Running TCL script in Synplify_Pro

Synplify_Pro is a synthesizing tool provided by Synplicity. A TCL script (Appendix C) is written in this thesis to simplify the process of synthesizing. This script is located in the directory *c:\csddm\FIRfilter*. The main target technology for this script is the Xilinx XCV800HQ240, where the user can change the target technology to different

Xilinx technology by simply changing the technology module option in the script file. The user has to open the Synplify_Pro software before running the script. Figure 3.3 shows a general view of the Synplify_Pro GUI.

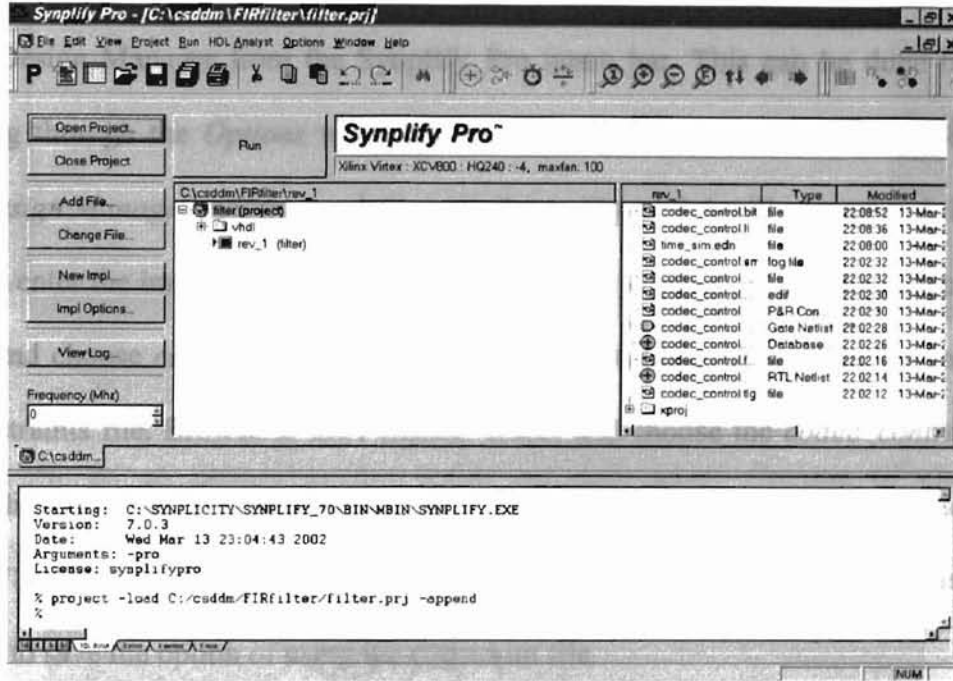


Figure 3.3 Sample image of Synplify_Pro software

The next step is to browse to the *Run* menu bar and look for the *Run TCL script* option in the *Run* menu. After clicking this option, a window pops up and asks for TCL script. The user has to browse the *c:\csddm\FIRfilter* directory and look for a TCL script file called *filter.tcl*. Select this file and click open to activate the synthesizer. The synthesizing process may take some time and it all depends on the complexity of the filter. The synthesizer will create all the necessary files such as **.edf* and **.ncf* files for implementation in a sub-directory called *rev_1* in *FIRfilter* folder. After synthesizing, a window containing the information for running the TCL script pops up. The user can just

close this window and go to the next step of the implementation. The next step of the implementation is to use the Xilinx software called Xilinx Design manager.

3.4 Running Xilinx Design Manager

To make implementation simpler, Synplify_Pro has the option to activate the Xilinx Design Manager from the Synplify_Pro menu bar. This can be done by simply browsing through the *Options* menu bar on Synplify_Pro and selecting the *Xilinx --> Start Design Manager* option. A window similar to Figure 3.4 will pop up and the user is asked to enter the implementation information. In this window, select the constraints file option and choose *custom*. Another window pops up asking the user for the location of the constraints file. Browse to the *FIRfilter* folder and choose the *codec_control.ucf* file (Appendix D) as the constraints file. *codec_control.ucf* file contains the pin assignment for XSV800 boards. This file is not valid for the implementation on other boards. Then click ok to save the option of using the constraint file.

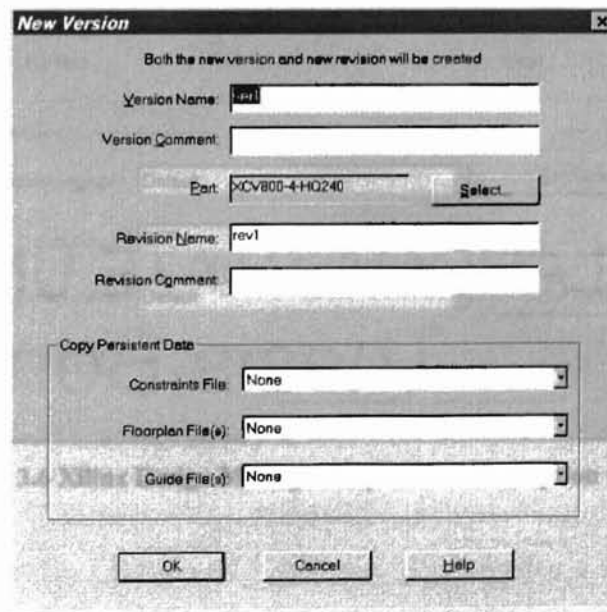


Figure 3.4 Xilinx Design Manager setting windows

After choosing the constraint file, the following window as in Figure 3.5 pops up.

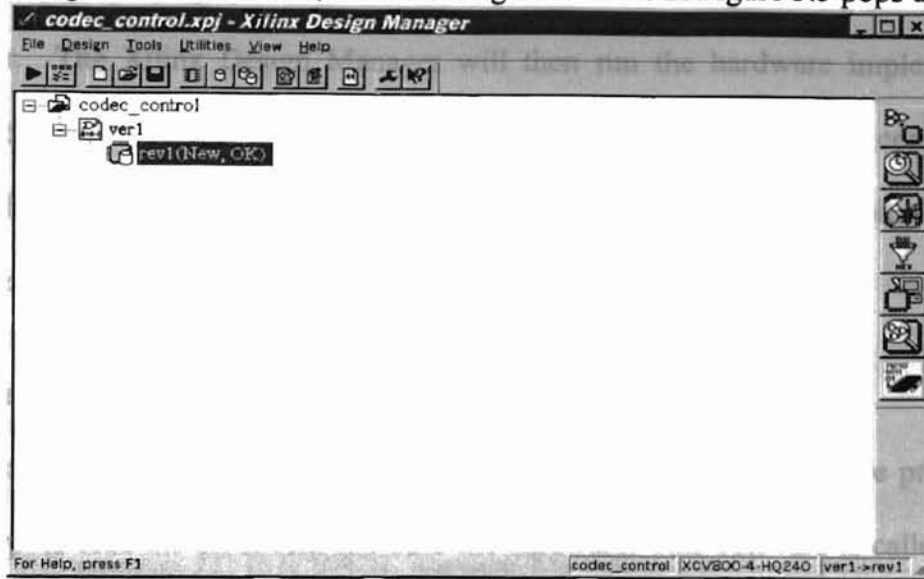


Figure 3.5 Xilinx Design Manager

Then go to *Design* in the menu bar and select *Options* to choose the implementation option. This will call up an option window as shown in Figure 3.6.

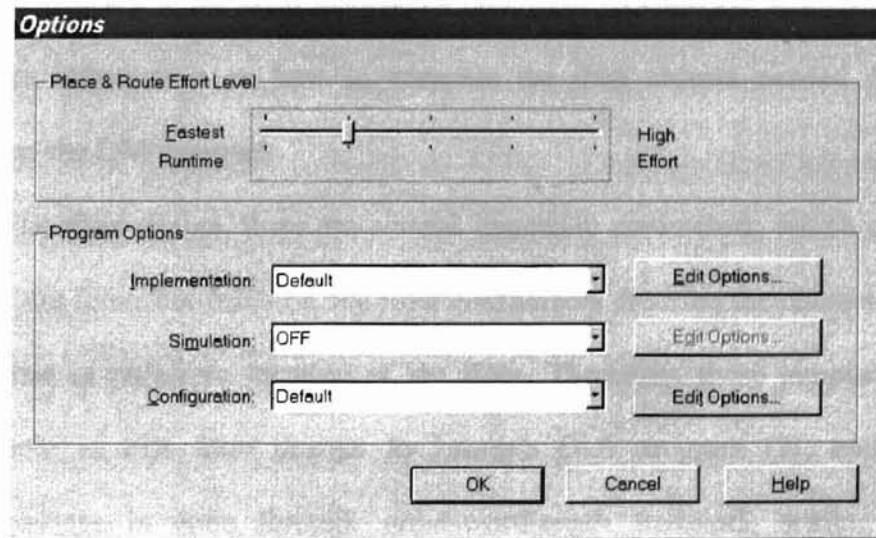


Figure 3.6 Xilinx Design Manager Implementation option window

In this window, the user has to set the simulation option to *Foundation EDIF* and leave the implementation option and configuration option as *Default*.

To start the implementation, the user can just select *Implement* on the *Design* menu bar. The Xilinx Design Manager will then run the hardware implementation, which includes translation, mapping, place and route, time simulation and configuration. The configuration process in Xilinx Design Manager will create a bit stream file (*codec_control.bit*) for implementation.

3.5 Implementation in XSV800 board

To do implementation on XSV800 board, we can use the software provided by Xess Corp. to load the bit stream file onto the hardware. This software is called Xstools [17]. The user just needs to load the bit stream called *codec_control.bit* into the Xstools software for implementation. After downloading the bit stream file into the board, the user can start testing the filter by switching the RESET of flip-flop, which is the DIP switch 8 on board.

Chapter 4

4. Translation of filter parameters

4.1 Introduction

This chapter includes explanation of the filter parameters translation from Leong's GUI program [2] to a VHDL package file called *params.vhd* (Appendix B-1). This file provides information for the FIR filter design. By combining *params.vhd* with another two FIR filter VHDL behavioral codes (*filt.vhd* and *rdfl.vhd*), one can implement the FIR filter on FPGAs. This section gives a brief introduction to the translation of the parameters. Section 4.2 discusses the overall flow of the translation. Both section 4.3 and 4.4 provides information on how to compute the data objects needed for the CSD approach and the DM approach.

In FIR filter design, there are several important parameters, which describes the behavior of the filter. For instance, the filter coefficients describe the characteristic of the filter response or pole/zero location of the filter. Therefore, these parameters play an important role in FIR filter design. In Leong's GUI program [2], optimization in hardware savings is done through order-wordlength trade-off, scaling and adder extraction.

After optimization, Leong's GUI program [2] provides a list of parameters such as filter coefficients (*QC*); filter order (*Nmin*), the CSD representation coefficient matrix (*newtable*), the DM technique coefficient matrix (*cdmopt*) and input bits (*bin*) of the

filter. However, this information needs further processing before it can be implemented. Users can only use the GUI program to simulate and verify if the filter coefficients meet the specifications. Therefore, a translation of these parameters is necessary to convert this information into hardware description language and verify that the Matlab simulation is accurate. In this thesis, the hardware implementation is done in Xilinx FPGAs. VHSIC Hardware Description Language (VHDL) is chosen for describing the behavior of the FIR filter. This chapter will mainly discuss how the parameters are translated into a package file called *params.vhd* (Appendix B-1) that can be used in the behavioral VHDL codes for FIR filter design. In the next chapter, more discussion regarding the behavioral VHDL codes will be included.

The translation of parameters can be done in several ways. Some software packages such as Xtreme DSP developer's kit from Xilinx has the ability to translate certain Matlab simulink blocksets in Matlab into hardware description language (HDL) using Xilinx system generator. These HDL codes can be further synthesized in Synplify_Pro from Synplicity software or FPGA advantage from Mentor Graphics. However, hardware implementation in this thesis is specifically targeted for FIR filter implementation in Xilinx FPGAs. The hardware implementation for FIR filters can be done easily by only using adder/subtractor and delay elements as the main components. There are several different choices of multipliers available for multiplication parts of the FIR filter coefficients. For example, parallel multiplier or booth multiplier can be used for such a purpose. However, for this thesis's application of FIR filter design, the coefficient multiplication is done by simply using shift and add/subtract operations. In

this chapter, translation of filter parameters using the CSD representation and the DM technique are described.

Translating the FIR filter parameters into VHDL codes is based on the concept in Husinga's [1] and Muthya's [18] theses. In their theses, the translations were mainly for CSD representation. In this thesis, several improvements have been made and the DM technique translations are included. More data objects such as cascading stage (*b_stage*), maximum number of shift (*zero_con*), maximum size of adder (*bshift*), number of extended zeros required for filter output which is less than 20 bits (*yex*), shared structure data object (*NN*) and odd or even order data object (*ODD*) are added for better description of FIR filter behavior. Details on the overall translation of the filter parameters are included in the next section.

4.2 The flow of FIR filter parameter translation

This section includes the description of the translation flow from optimized filter parameters in Leong's GUI program [2] to parameters that are recognized in hardware. The parameters' translation is explained briefly in Figure 4.1. The parameters translation is done in the Matlab file called *layout.m* (Appendix A). The following paragraph describes how the translation takes place.

After optimization, the GUI program provides certain information such as the filter coefficients (*QC*); filter order (*Nmin*), the CSD representation coefficient matrix (*newtable*), the DM technique coefficient matrix (*cdmopt*) and input bits (*bin*) for FIR filter hardware implementation. First, the program will check which techniques the user wants to use. User has the option of choosing either the CSD representation or the DM technique before hand. By choosing either of these techniques, the translation procedure

is slightly different. However, in both techniques, the subroutine will create a VHDL package file called *params.vhd* (Appendix B-1), which can further be used to describe the behavior of the digital filter. The *params.vhd* (Appendix B-1) package file is created using the *fopen* and *fprintf* command in Matlab. The *fprintf* command prints the library clause and use clause on the *params.vhd* (Appendix B-1) file. Both techniques' subroutine include library clause and use clause. These clauses are used to define the standard library use for the digital filter design. In addition, *fprintf* command prints design units such as entity declaration, architecture body, package declaration and package body in *params.vhd* (Appendix B-1) file.

Data object is the object declaration of a specific type with certain value. There are three different data classes for data object. They are Constant, Variable and Signal classes [19]. However, in *params.vhd* (Appendix B-1) package file, we only use the constant class to declare the FIR filter's parameters. Ten constant class data objects are used in *params.vhd* (Appendix B-1) package file. These data objects are order of filter (N), input bits (b), cascading stage (b_stage), number of nonzero bits in each stage (n_per_row), locations of nonzero bit ($coeff_vec$), maximum number of shift ($zero_con$), maximum size of adder ($bshift$), number of extended zeros required for filter output which is less than 20 bits (yex), shared structure data object (NN) and odd or even order data object (ODD). All of this information is shared by other design units. The GUI program must evaluate the optimized parameters before the function is chosen. Therefore, the order of the filter (N), input bits (b), number of bits required for representing the multiplier coefficient and filter coefficients are known before the GUI program runs the *layout.m* (Appendix A) file.

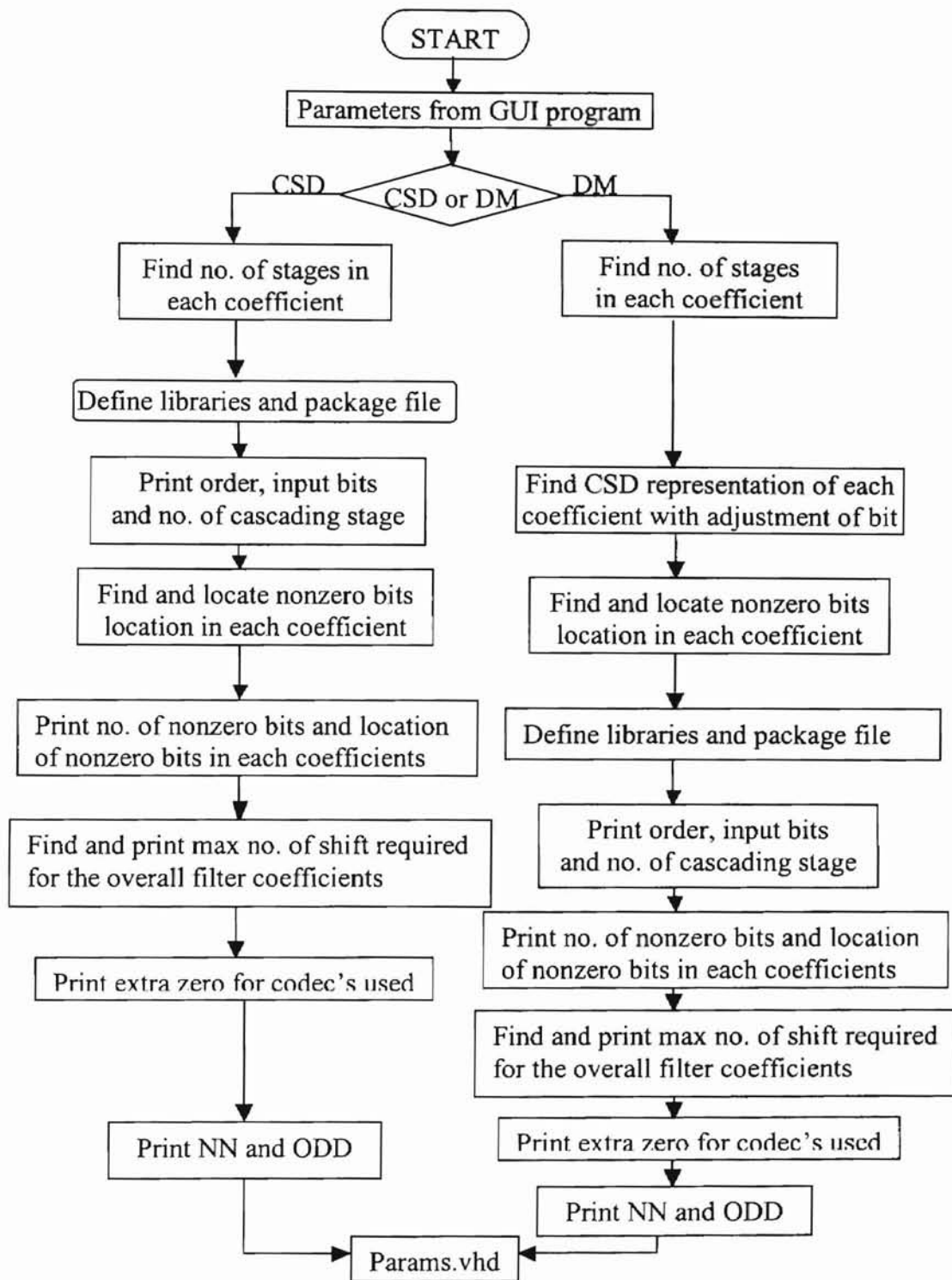


Figure 3.1 Overall flow of the parameters translation

Filter order (N) and inputs bits (b) are standard information that can be printed on *params.vhd* (Appendix B-1) without any process of translation. However, the *layout.m*

file contains algorithms for finding the rest of the data objects such as cascading stage (*b_stage*), number of nonzero bits in each stage (*n_per_row*), locations of nonzero bit (*coeff_vec*), maximum number of shift (*zero_con*), maximum size of adder (*bshift*), number of extended zeros required for filter output which is less than 20 bits (*yex*), shared structure data object (*NN*) and odd or even order data object (*ODD*). The next two sections contain more detailed discussion on how to compute these data objects. After computing all the data objects, the *fprintf* command is used to print all of the information on the *params.vhd* (Appendix B-1) package file. Lastly, the *fclose* command in Matlab is used to close the file printing. Before one can use the *fprintf* command, *fopen* command has to be used to specify where to read the file from or write the file to. The purpose of *layout.m* (Appendix A) file is to create a package file for defining the filter parameters. Therefore, we need to define the *fopen* command as a write function instead of a read function. The following line of code is used to ask Matlab to write the character into the file called *params.vhd* (Appendix B-1).

```
fid=fopen('params.vhd','w');
```

The main command used in the *layout.m* file is *fprintf*. The following example is used to further demonstrate how *fprintf* command can do the job of printing the *params.vhd* (Appendix B-1) package file. This example shows how to print the order of the filter onto the *params.vhd* (Appendix B-1) package file. The following line is the code written in Matlab.

```
fprintf(fid,'CONSTANT N: INTEGER:= %2d;\n',Nmin);
```

In this line, “INTEGER” is used to define “Nmin” as the constant class data object with integer value. “fid” is the file identifier, “%2d” is the optional subtype specifier for

conversion character and “\n” is the special format in Matlab for linefeed. By using the above code, the following line will be printed to the file that the user specifies in the *fopen* command.

CONSTANT N: INTEGER:= Nmin; (where Nmin is the number of order).

The above discussion gives the basic concept of the flow for translating the parameters into a package file that can be used for defining the digital FIR filter parameters.

4.3 Computation of CSD data objects

The main purpose of this section is to show how the computations of data object for the CSD representation take place. All of these codings are included in a Matlab m-file called *layout.m* (Appendix A). Filter order (N) and input bits (b) both data objects do not need any further computation. Both of these data objects can be directly printed on the *params.vhd* (Appendix B-1) package file by using *fprintf* command. Cascading stage (b_stage) is one of the data objects, which needs computation to provide useful information. Cascading stages for the CSD representation are all value one or zero since there is no involvement of cascading structure in CSD representation. The value for b_stage is one when the filter coefficient is not a zero value or else b_stage will be zero. From the optimized filter parameters, we can find the cascading stage by checking the optimized filter coefficients.

Number of nonzero bits in each stage (n_per_row) is the next data object that needs certain computation to provide useful information. The table of CSD coefficients (*newtable*) is already available before *layout.m* is activated. Table 4.1 is an example of the CSD coefficients table.

Coefficient	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
-0.3125 →	0	0	0	0	0	-1	0
-0.046875 →	0	0	0	0	-1	0	1
0.078125 →	0	0	0	0	1	0	1
0.25 →	0	0	1	0	0	0	0
0.25 →	0	0	1	0	0	0	0
0.078125 →	0	0	0	0	1	0	1
-0.046875 →	0	0	0	0	-1	0	1
-0.3125 →	0	0	0	0	0	-1	0

Table 4.1 An example of High-pass FIR filter coefficients

By using the information from the CSD coefficients table, we can make a small routine in Matlab *layout.m* file to count the number of nonzero bits in each coefficient. For example, the first coefficient in the Table 4.1 is -0.3125 , which is 2^{-5} , has only one nonzero bit. To take another example, the second coefficient in Table 4.1 is -0.046875 . There are two nonzero bits, where one of the nonzero bits has a minus sign, the other has a plus sign. As the minus sign nonzero bit has a larger value than the plus sign nonzero bit, the value of this coefficient will be negative. Again, after computation of the number of nonzero bits in each coefficient, they are printed to the *params.vhd* (Appendix B-1) package file using *fprintf* command.

Next data object is the location of nonzero bit (*coeff_vec*). The computation for finding the nonzero bit location is done in the same routine for finding the number of nonzero bit in each coefficient. In the routine, the sign of the nonzero bit is included to preserve the original value of the coefficient. Taking the first coefficient in Table 4.1 as an example, the nonzero bit is located at 2^{-5} . From a hardware point of view, it is only

taking the input signal and shifting the signal five times to the right. Therefore, the value for this `coeff_vec` will be '-5', where the minus sign indicates this filter coefficient is a negative coefficient.

Maximum number of shift (`zero_con`) is the next data object that needs to be computed. This data object is used to preserve the bits required for the filter adders so that no data will be lost during the filtering process. This number can be obtained by finding the maximum number for the `coeff_vec` plus one, which is the maximum shift needed in the multiplying process.

The next data object is maximum size of adder (`bshift`). The way to compute this data object is to add the value of input bit (`b`) and maximum number of shift (`zero_con`). The reason for having `bshift` is to make sure the adding operation in the filter has sufficient adder size. Input data will be lost if the size of the adder in the filter is not big enough.

The next data object is number of extended zeros required for filter output, which is less than 20 bits (`zex`). This data object is used to make sure that the output of the filter is equal to 20 bits. If the filter output is less than 20 bits, then the zero signals are needed to concatenate with the filter output signal so that the filter will have 20 bits of output. The reason for having a 20-bit output is that the Codec on the Xess XSV800 board can only produce 20 bits of output.

The last two data objects are shared structure data object (`NN`) and odd or even order data object (`ODD`). `NN` is determined by the order of the filter structure. If the filter order (`N`) is an odd number, this means that the filter has an even number of filter coefficients. Therefore, `NN` is as follow:

$$NN = (N+1)/2 - 1 \quad (4.1)$$

On the other hand, if the filter order is an even number, NN will be as follow:

$$NN = N/2 \quad (4.2)$$

The odd or even data object (ODD) is determined by the filter order (N) too. ODD will only equal to one or zero by using the remainder function in Matlab. ODD will be equal to zero if the filter order is an even number and is equal to one if the filter order is odd. The *fprintf* command is used to print all of the above data objects to *params.vhd* (Appendix B-1) the package file. Using the *fclose* command to close the file completes the whole package.

4.4 Computation of DM data objects

The computation of DM data objects has some similarity to computation of data objects in CSD representation. The computation is located in the second half of the *layout.m* coding (Appendix A). First, the number of filter order (N) and number of input bits (b) are the same as the CSD representation because the DM technique only causes change in the way to represent the filter coefficients. The next step is to compute b_stage of each coefficient. As discussed in Chapter 2, the DM technique emphasizes using cascading method to save hardware costs. Therefore, information about the number of stages is important for the DM implementation. As mentioned in the last section, cascading values for the CSD representation are only one or zero. However, the cascading values for the DM technique can range from zero up to the number of stages required for representing the filter coefficients. The only information provided from Leong's GUI program [2] before running the *layout.m* program is the decimal form of cascading coefficients. Therefore, in *layout.m* program there is a small routine for finding

the number of cascading stages needed for each coefficient. This information is stored under variable name *bstage*.

Before finding the rest of the data objects for the DM technique, another new table is created so that each decimal form cascading coefficients will have its own CSD representation. The table for the DM technique is call *dhtable*.

The next data object is number of nonzero bits in each stage (*n_per_row*). The calculation of this data object and location of the nonzero bit (*coeff_vec*) data object are found under the same routine. By using the information from the previous small routine, which is *dhtable*, the location and number of nonzero bits in each stage can be found.

We still need to preserve the maximum bits in each coefficient if the cascading is more than one stage in each coefficient. The following description shows the way to adjust the nonzero bit location. First, we need to find the coefficient with more than one cascading stage. In those coefficients with more than one stage, the routine will check the maximum number of bits required for each stage. By deducting the maximum number of bits required for each stage from the maximum number of bits required for each coefficient, the left over bits will be used in last stage of the coefficient. The following is one of the examples to further illustrate the adjustment of nonzero bits location.

Assume the value of filter coefficient as 0.34375. We can use $(2*11)/2^6$ to get 0.34375. The representation for each number 2 and 11 is 0000010 (which is $2/2^6$) and 0010-10-1 (which is $11/2^6$) respectively. However, if we multiply $(2/2^6)$ and $(11/2^6)$, we get 0.00537109375, which is not the same as the actual number (0.34375) that we want. Therefore, we need to do some adjustments to get the correct CSD representation. First the number 2 will be divided by power of 2, which is one bit. In this case, first stage

number 2 will be straight wiring. The left over bits will be 2^5 as the first stage took one bit away. Therefore, the second stage, which is the last stage in this example, number 11 will be divided by 2^5 . By converting the results back to CSD representation, we can get the correct representations for $(2*11)/2^6$, which are 1000000 and 010-10-10. If the cascading stage is more than two, the adjustment of bits will keep on going until the last stage.

After the adjustment, we can continue to find the rest of the data objects using the similar technique discussed in the previous section. However, the maximum number of shift (*zero_con*) has to count is the maximum shift of each coefficient after adjustment of bits. The maximum size of adder (*bshift*) and number of extended zeros required for filter output which is less than 20 bits (*yex*), share structure data object (*NN*) and odd or even order data object (*ODD*) will be using the same computation described in the previous section.

Chapter 5

5. VHDL coding for FIR filter design

5.1 Introduction

This chapter shows how to convert the optimize filter parameters found in Leong's GUI program [2] into hardware description coding that can be realized for Xilinx FPGA chips. The basic hardware components for designing a FIR filter are adders/subtractor and delay elements. As mentioned in Chapter 2, using shift and add/subtract arithmetic operation can do the multiplying of the filter coefficient. The delay elements are simply D flip-flop with asynchronous reset. As discussed in Chapter 2, we are using transposed direct form for the FIR filter design to get a better sampling rate.

The implementation of FIR filter in this thesis is done using three VHDL files that describe the behavior of the filter. These three files are *params.vhd*, *filt.vhd* and *rdfl.vhd*. The *params.vhd* (Appendix B-1) VHDL file is the lower level of the overall design followed by *filt.vhd* and the top level is *rdfl.vhd*. Figure 5.1 describes the top-to-bottom level of the VHDL file for FIR filter design targeted for Xilinx FPGAs.

The *params.vhd* (Appendix B-1) is the main file that describes the VHDL representation of the CSD/DM coefficients table, where the FIR filter parameters are translated into constant declaration. The *filt.vhd* describes the behavior of the CSD/DM FIR filter, where several functions are created for manipulation of the FIR filter behavior.

The top level of the FIR filter files is *rdfl.vhd* and this file calls function and constant from both *params.vhd* (Appendix B-1) and *filt.vhd* (Appendix B-2) files. The way to create *params.vhd* (Appendix B-1) file is already discussed in Chapter 4. In this chapter, we will only discuss how to use these three files to describe the behavior of the FIR filter.

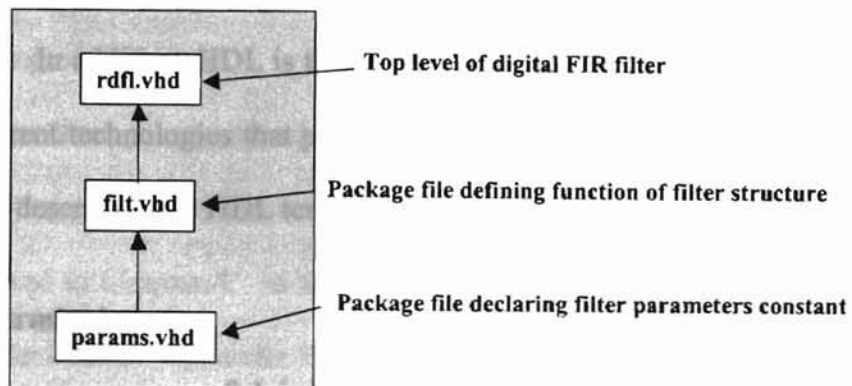


Figure 5.1 Top-to-bottom levels of VHDL files

VHDL is chosen as the hardware description language for coding because VHDL is considered as one of the industry standards. VHDL was first introduced in 1980 by the USA department of Defense (DOD) for having a set of self-documenting circuit designs, which follows a common methodology, and these designs can be reusable with new technologies [20]. Discussion on how these VHDL files were created for implementation purposes is included in this chapter.

There are several advantages of using HDL coding. One of the advantages for using HDL coding is that the users do not need to enter the gate-level description of the design manually. Users can use the available standard library cell or custom library cell defined by themselves to implement their design in HDL multiple times without going through the necessity of entering the gate-level description each time. This is the reason why most industries try to use HDL coding instead of gate-level description for their

design. The second advantage of using HDL coding is that the user can view the design at a higher-level structure to reduce the complexity of the design. This will improve the design quality. The user just needs to deal with his design ideal instead of considering the gate-level components. However, certain circuit designs may need custom library cells to meet the specifications and most companies have their own custom library cells provided for their designers. In addition, HDL is technology independent so users can target their HDL files in different technologies that provide more freedom in circuit designing. The following sections describe basic HDL terminology and the behavior coding of the filter.

5.2 Basic HDL Terminology

The most common way of doing design modeling of a digital system in HDL coding is to make the design into abstract blocks or so called components. Each component is instantiate as a design entity, where an entity is defined as a hardware abstraction of a system. Usually each component is separated into small modules and the whole system is combined together by using design hierarchy and forms a top-level entity. VHDL entity has five different types of design units. There are entity declaration, architecture body, package declaration, package body and configuration declaration. Entity declaration defines the external port information of the component. For example, the input and output signal names of the component. The architecture body defines the internal function of the design. One can define several architecture bodies in one design entity where each architectures has different interconnected components, or a different set of concurrent or sequential statements to describe the design. Package declaration is defined as a common declaration that can be globally used by different design units. For example, constant declaration, subprogram and data types can be included in package

declaration. Package body is defined as the contents of the subprogram that was declared in package declaration. Configuration declaration is the main design unit that is used to bind all associate architecture bodies together [19][20]. In this thesis, the behavior coding of the FIR filter is all based on the basic HDL terminology in designing.

5.3 Behavior coding of FIR filter

5.3.1 Coding on the filter parameters declaration

The FIR filter parameters declaration (*params.vhd*) is generated from the GUI program as discussed in Chapter 4. In this section, we will discuss the functionality of each parameter's declaration. Appendix B-1 is a sample of a FIR filter. As mentioned in Chapter 4, there are ten constant class data objects used for hardware implementation in this thesis.

Data object *N* (filter order) is used to determine the maximum number of filter coefficients, which is used as a parameter for the size of the array for storing the results of input multiplied by each filter coefficient. Data object *b* is used to assign the number of bits required for the input signal. Data object *b_stage* (number of cascading stage) is used as data for controlling how many times looping of stage is necessary in each coefficient. Data object *n_per_row* (number of nonzero bits in each stage) and data object *coeff_vec* (location of nonzero bits) are used to perform the shift and add/subtract arithmetic operations in the filter multiplier. Data object *zero_con* (maximum number of shift for the whole filter) is used as the constant number for the number of zeros necessary for concatenation. Data object *bshift* (sum of maximum number of shift with input bits) is used to assign the number of bits required for the filter in order to preserve

all the data. Data object *yex* is used for truncating the signal so that we can use the Codec on Xilinx board for implementation purposes. The last two data objects *NN* and *ODD* are used for sharing the multipliers in the FIR filter. The sharing of multiplier is achievable as the FIR filter has a symmetric characteristic.

5.3.2 Coding on the filter behavior function file

The filter behavior function file (*filt.vhd*) is used to describe the general structure of a FIR filter. *Filt.vhd* file acts as a package file for top-level entity *rdfl.vhd*, where *rdfl.vhd* describes the overall implementation of FIR filter. Appendix B-2 includes the VHDL coding of the file *filt.vhd*. In this section, a detailed description of each function used to describe the filter structure in *filt.vhd* is included. There are three functions used in file *filt.vhd* for generating the FIR filter structure. These three functions are *snd_shift*, *zero_vec* and *xstage*. The *snd_shift* function is used to perform the shift operation of the filter. This function takes *coeff_vec* and the filter input signal as input and performs the shifting and returns the result of this function with the signal that already shifts based on the number of shifts indicated in *coeff_vec*. The second function *zero_vec* is used to generate zero vectors of the maximum bit numbers. This function is required to accommodate FIR filter with zero coefficients. This function *zero_vec* takes the number of maximum size of adder (*bshift*) as input and returns a zero vector with the size number indicated in *bshift*.

The last function *xstage* is used to perform the overall shift and add/subtract arithmetic operation of each filter coefficients and return the result in an array. This array has the row size of $NN+1$ and column size of *bshift*. Input of *xstage* is the input of filter signal concatenates with zero. The number of zero concatenates is the number of

zero_con. *Zero_con* set up in such a way that the shifting operation will not shift any input signal out as the shifting operation and only shifts the zero signals concatenate at the end of the signal. Figure 5.2 shows how *xstage* can perform the multiplier function of the FIR filter by using shift and add arithmetic operations.

Xstage takes the FIR filter input signal as input and concatenate *zero_con* bits of zero. This signal is called *XS*, where the size of signal is the size of input bits plus the size of the maximum shift needed for the FIR filter. The maximum shift is taken from the results of Leong's GUI program [2]. *Xstage* will perform the shift and add operation for every coefficient in the FIR filter. Therefore, the *xstage* function will loop *NN*. In the loop, the program will check to see if *b_stage* is equal to zero; where *b_stage* is equal to zero indicates that the filter coefficient is zero. In this case, zero vectors are generated using *zero_vec* function. On the other hand, if the *b_stage* is greater than zero, the program will go into a loop where this loop will loop *b_stage* times.

At each stage, there is an internal loop to perform the shift and add operation. The number of loops for this internal loop is based on the number of *n_per_row*. In this internal loop, the program will first check to see if the *coeff_vec* is equal to zero. This condition is set to fit the DM technique, as some of the DM technique coefficients may need straight wiring from input instead of shifting. In this case, if *coeff_vec* is equal to zero and the *b_stage* is equal to zero, *XSI* (internal signal for each shift) is equal to *X* (input signal), else *XSI* equals to *XS* (internal signal for each stage). On the other hand, if the *coeff_vec* is not equal to zero, it will take the positive value of *coeff_vec* and call the function *snd_shift* for shifting. After the program runs shifting or straight wiring routines, both routines go to another condition for checking. This condition is to check

the sign of the *coeff_vec*. By checking the sign of *coeff_vec*, we can know whether we are doing an add/subtract operation. *XS_int* is the internal signal for each cascading stage. After processing all the stages in each coefficient, the data is stored in an array for top-level entity usage.

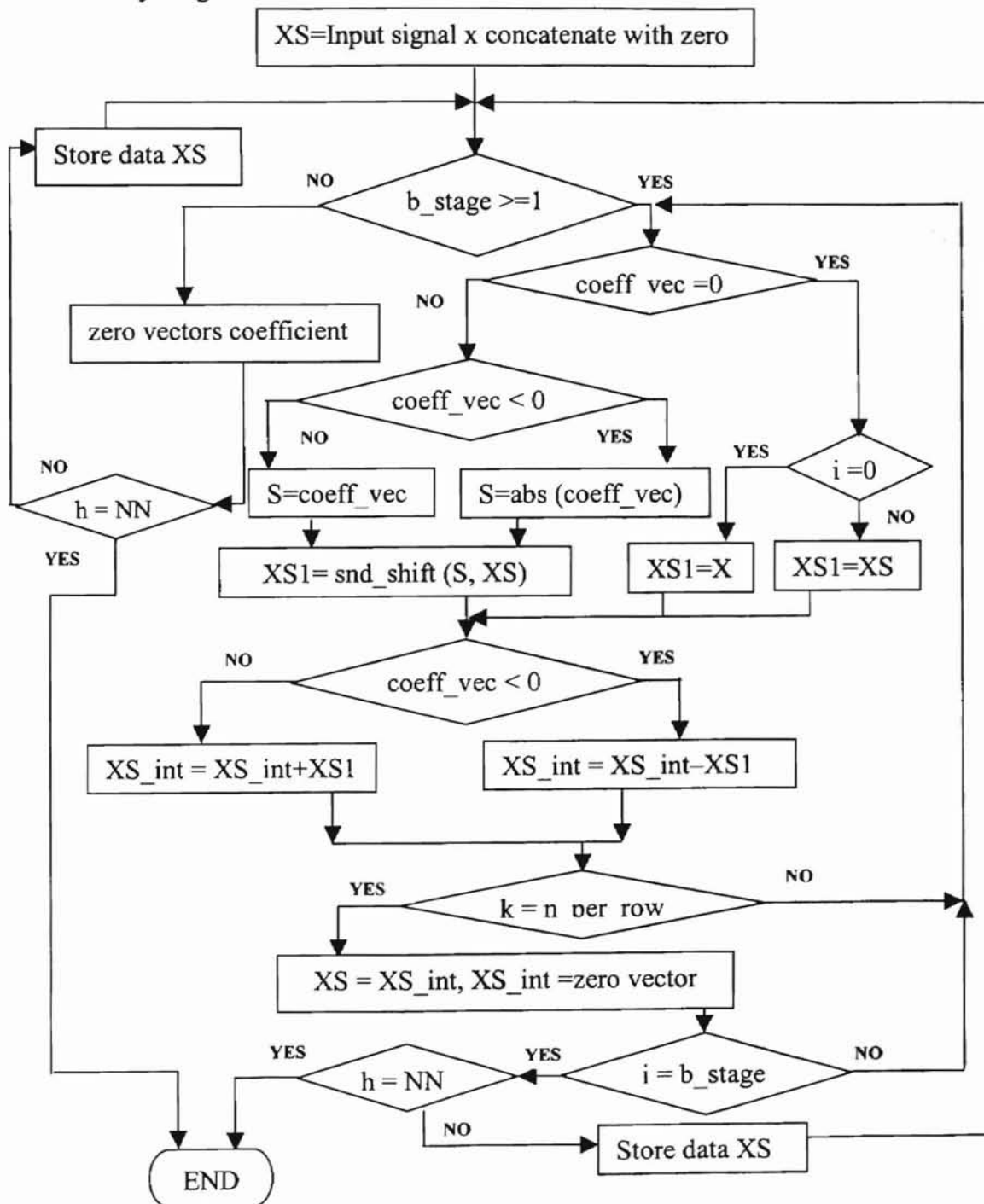


Figure 5.2 Flow chart for FIR filter coefficient multiplier

5.3.3 Coding on the top level of FIR filter

The top level of the entire structure is called *rdfl.vhd*. Appendix B-3 shows the VHDL coding of *rdfl.vhd*. In this top-level structure, *params.vhd* (Appendix B-1) and *filt.vhd* (Appendix B-2) are used as the package file for FIR filter design. The top-level ports are as described in Figure 5.3. The input X is the input of the filter, the size of which depends on the input *bin* from Leong's GUI program [2]. The input clock is the sampling clock for the FIR filter, which in this thesis is set at approximately 48KHz to match the sampling frequency of the Codec on the XSV800 board. The input RESET is used to trigger the asynchronous reset of the D flip-flops. The output, Y_{codec}, is set to 20 bit widths as the Codec on the XSV800 board can only handle 20 bits of DAC output.

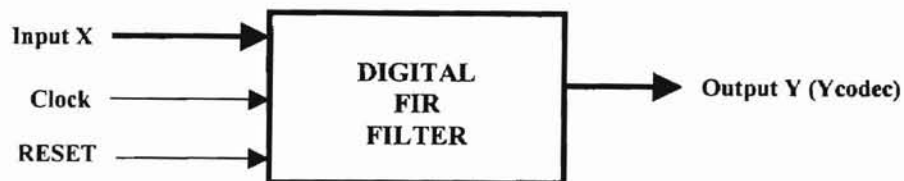


Figure 5.3 Top level of FIR filter

Figure 5.4 shows the overall flow of top-level of FIR filter design. The *result_final* signal is the input of the filter X pads with *zero_con* number of zeros. As discussed in section 5.3.2, the padding of zero is to assure that there is no signal lost after the shift and add/ subtract arithmetic operation. The *rdfl.vhd* will then call *xstage* function in *filt.vhd* by taking *result_final* as input to find the value after multiplying each filter coefficient. The result is then stored in *XS* where *XS* is an array with row size of $NN+1$ and column size of *bshift*. The next step is to generate the overall structure of the FIR filter.

Mishra, N. et al. / International Journal of Electrical and Electronics Engineering

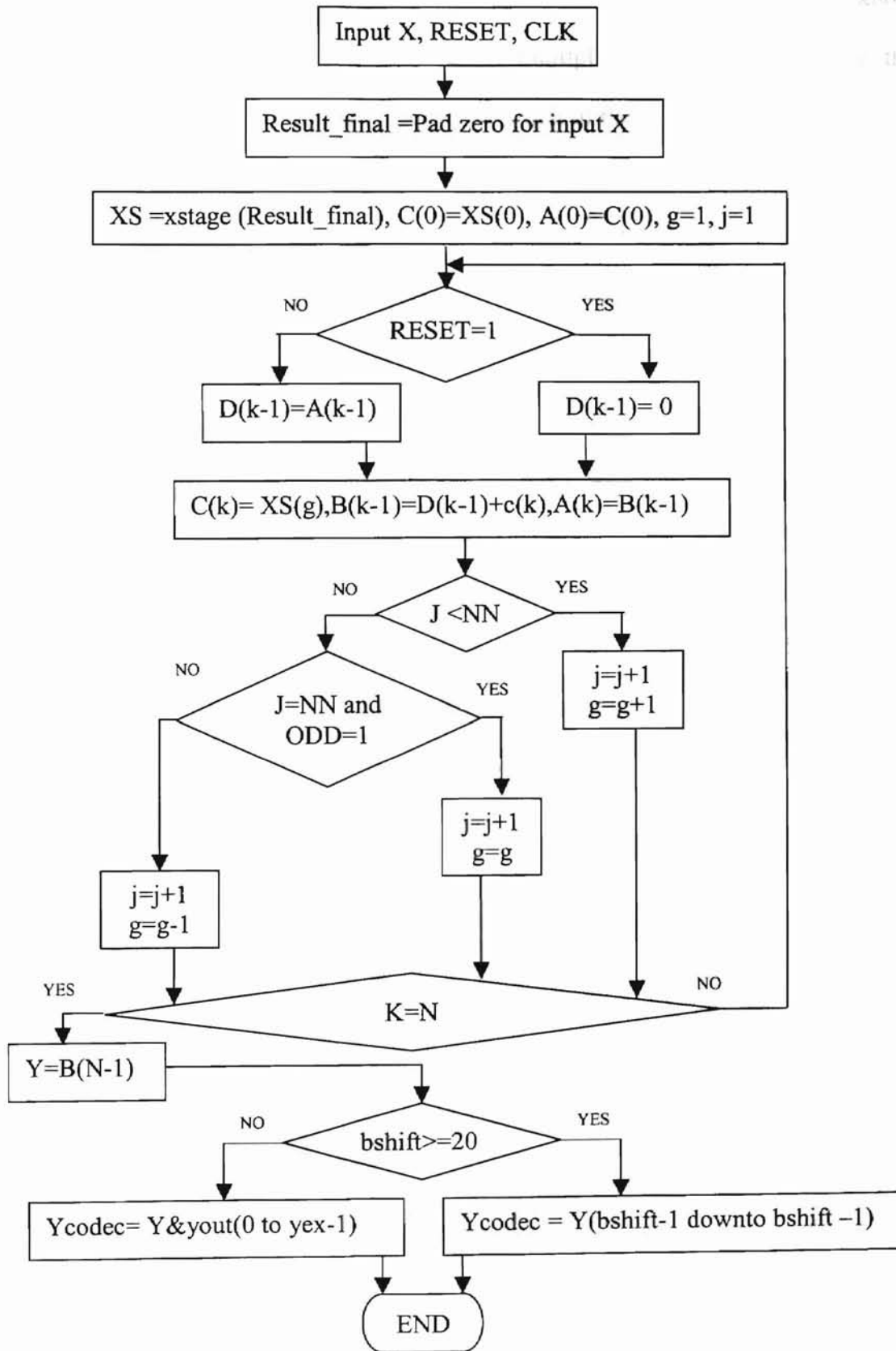


Figure 5.4 Flow chart for top-level of FIR filter

Before going into the overall structure of the FIR filter, it is important to know that we can save more hardware by sharing the multipliers of the filter. Due to the symmetrical characteristic of FIR filter, we can save half of the hardware requirements by sharing the multiplier circuit. Figure 5.5 describes a fifth order FIR filter in transpose direct form, where the filter coefficient number is even. Therefore, we can share the multiplier structure as described in Figure 5.5. In this case, the filter order is an odd number where NN is equal to two and ODD is equal to one based on the equation 4.1 discussed in Chapter 4.

For an even number order filter, we can use the structure described in Figure 5.6, which is an example of a sixth order FIR filter. The $b3$ multiplier is a stand-alone filter coefficient without sharing. In this case, NN is equal to three and ODD equal to zero based on the equation 4.2 discussed in Chapter 4.

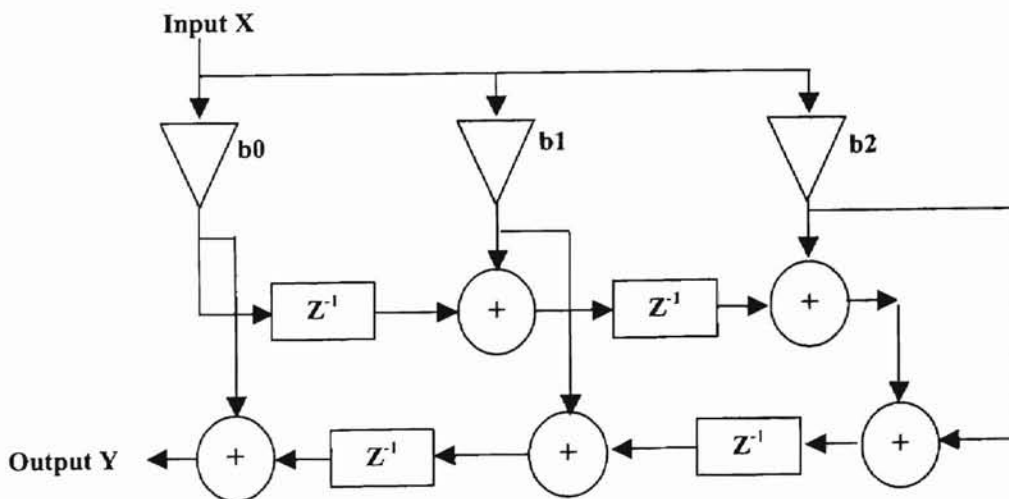


Figure 5.5 Structure of transpose direct form FIR filter with odd number order

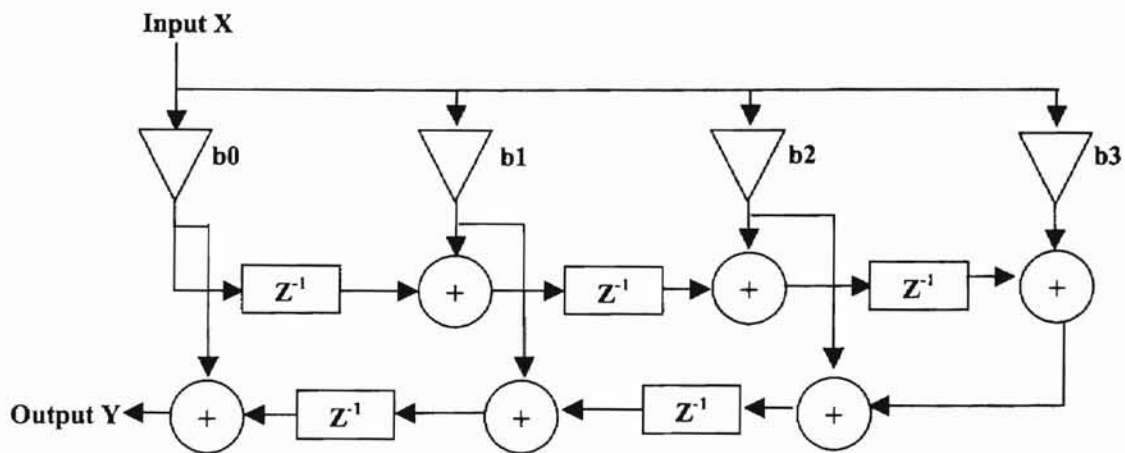


Figure 5.6 Structure of transpose direct form FIR filter with even number order

Although we can save half of the hardware by sharing the multiplier circuit, the structural adders and delays are still required. Therefore, we need to loop for $N+1$ times in generating the structural adders and delays. Figure 5.7 describes a basic structure of a first order FIR filter. From this figure, the first output signal $XS(0)$ after multiplying with the first coefficient is connected with internal signal $C(0)$ and this signal is then connected to $A(0)$. Then the behavior code will go into a loop of N times. In the loop, the delay element (Z^{-1}) will pass zero signals to the $D(0)$ internal signal if the RESET is high. On the other hand, $D(0)$ internal signal will pass anything from the previous $A(0)$ internal signal. Internal signal $C(1)$ will be connected to $XS(1)$, which is the output of the second multiplier. This signal is then added with the $D(0)$ signal to produce $B(0)$ internal signal. The condition for checking variable j with NN is to determine when to share the filter multiplier circuit. After looping N times, the output of internal signal $B(N-1)$ will be the final output (Y) of the overall filter. In Figure 5.4, there is one more condition of checking the constant data object *bshift* because the Codec can only hold 20 bits of output signal.

Therefore, if the output Y is more than 20 bits, we will have to truncate the output signal to 20 bits to accommodate the Codec on XSV800 board.

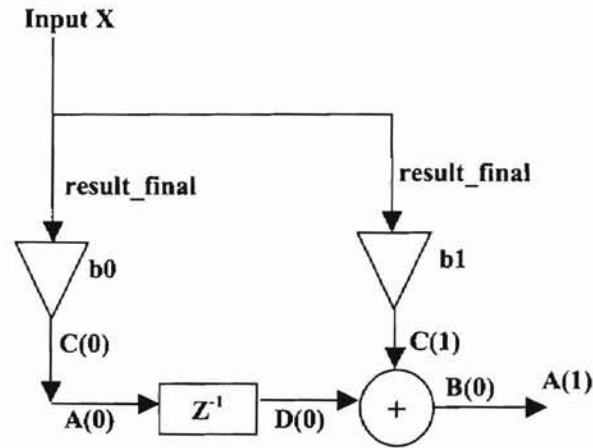


Figure 5.7 Basic structure of 1st order FIR filter in transpose direct form

Chapter 6

6. Implementation and comparison

6.1 Introduction

This chapter shows that Leong's GUI program [2] using CSD representation and DM technique is valid in the hardware sense. Therefore, an example of the hardware implementation in Xilinx FPGAs using the CSD representation and the DM technique is included for verification purposes.

A Low-pass filter example is included in section 6.2. This section discusses the simulation in Xilinx Logic simulator and actual implementation in Xilinx FPGAs. Data are taken from the actual hardware to compare with the original specification. The hardware implementation costs for both the CSD representation and the DM technique are included for comparison purposes. The hardware cost is the number of slices required in Xilinx FPGAs for implementing the FIR filter. This cost includes the hardware required for the filter multiplier circuits, structural adder /subtractor and delay elements.

In section 6.3, the comparison of hardware costs for the CSD representation and the DM technique are included. Four different types of FIR filter with specific filter specifications are used for this comparison purpose. Two different Low-pass filter specifications are tested for comparison of the standard hardware costs calculated in Leong's GUI program [2] with the actual hardware costs.

6.2 Implementation

The implementation is done on the Xess XSV800 board, where the main FPGA used in Xess XSV800 board is Virtex XCV800HQ240-4. The actual implementation of hardware is done to verify the filter by taking data from the output of the Codec on Xess XSV800 board.

The following example is used to further illustrate the implementation of the FIR filter in hardware.

6.2.1 FIR filter implementation example

This section uses a Low-pass FIR filter as an example for illustration. This example includes simulation results obtained from Xilinx Logic simulator, and actual experiment data from XSV800 board using the CSD representation. Experiment data are taken for comparison with the original specification. The specification of the filter is as follow:

Sampling frequency = 48000KHz
Passband frequency = 11000KHz
Stopband frequency = 13000KHz
Passband ripple = -3dB
Stopband ripple = -30dB

By using the CSD representation option from Leong's GUI program [2], we get the following filter coefficients.

$$b = \begin{bmatrix} -0.015625 & -0.015625 & 0.0078125 & 0.0234375 & 0.003906025 \\ -0.015625 & 0.0078125 & 0.0351560625 & 0 & -0.0507810625 \\ 0.0078125 & 0.1757810625 & 0.265625 & 0.1757810625 & 0.0078125 \\ -0.0507810625 & 0 & 0.0351560625 & 0.0078125 & -0.015625 \\ 0.0039060625 & 0.0234375 & 0.0078125 & -0.015625 & -0.015625 \end{bmatrix}$$

After going through optimization algorithm in Leong's GUI program [2], the filter order is twenty-four and input bits for the filter is nine. The size of the filter output signal

is eighteen bits. Note that we need to feed the output of the filter to the Codec DAC on XSV800 board. Therefore, the output of the filter is padded with two extra zeros to get the 20-bit size. Figure 6.1 is the overall RTL view of the hardware implementation of the Low-pass filter. This RTL view includes the FIR filter circuit, Codec circuit and a control circuit for Codec. This RTL view is captured from the synthesizing software, Synplify_Pro 7.0.

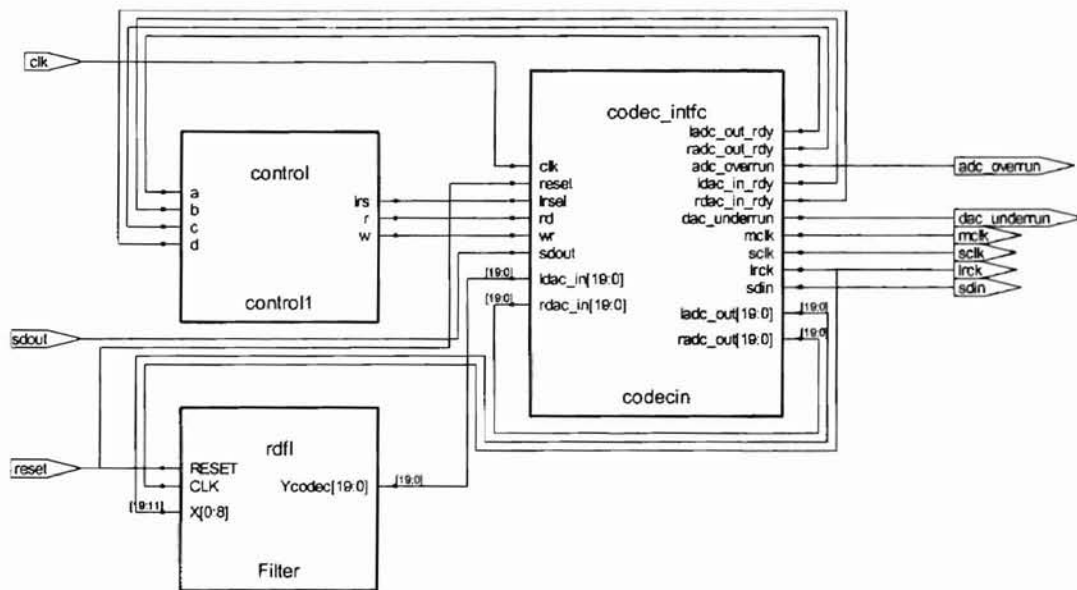


Figure 6.1 Overall structure of the hardware implementation

6.2.2 Simulation result

By using Xilinx Logic simulator, we can verify the FIR filter design. The simulation is verified by taking the filter circuit instead of the overall circuit. After implementation using Xilinx Design Manager, a *time_sim.edn* file is created. This file contains the simulation information for the filter design. This file is loaded into Xilinx Logic simulator for simulation purposes. The filter design is verified by checking the impulse response of the filter. Figure 6.2 shows the magnitude plot of the Low-pass filter. The following steps are used to get this magnitude response plot. First, we take the

impulse response of the filter using Xilinx Logic Simulator. Then we take two hundred points Fast Fouries Transform of this response. Lastly, we take the results and convert them into magnitude in decibels. Figure 6.2 only takes 100 points of the results where the other half will be a duplicate image of the first half of the frequency response. As we can see, the simulation result matches the original specification. This again proved that the design of this filter works.

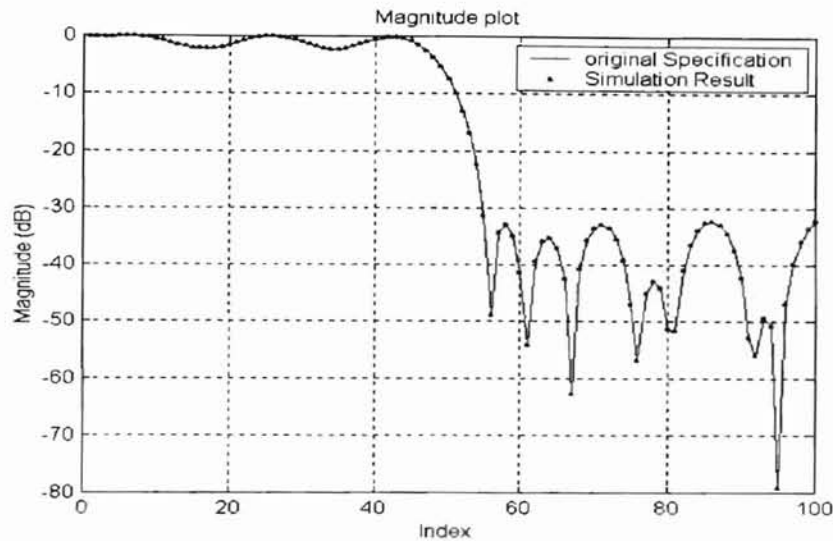


Figure 6.2 Comparison of original specification and simulation result

6.2.3 Experimental result

Figure 6.3 shows how the testing of the design takes place. The main equipment used for testing is Function Generator (HP 3314A), Spectrum Analyzer (HP 3585A), Digital Oscilloscope (TDS 3052) and Xess XSV800 prototyping board. The Function Generator will act as the input source for the filter. The output signal from Function Generator is fed to the Digital Oscilloscope so that we can obtain the input magnitude of the source signal. The output signal from the Function Generator is also fed to the Codec on the XSV800 board. The Codec converts the analog signal to digital signal. The

digitized signal from the Codec is fed to the FIR filter, which is already programmed in the FPGA. The process of filtering the signal takes place in the FPGA. The processed digital signal from the FPGA is fed to the Codec, which converts the digital signal back to the analog signal. This signal is then sent through the stereo jack on XSV 800 board to the Spectrum Analyzer.

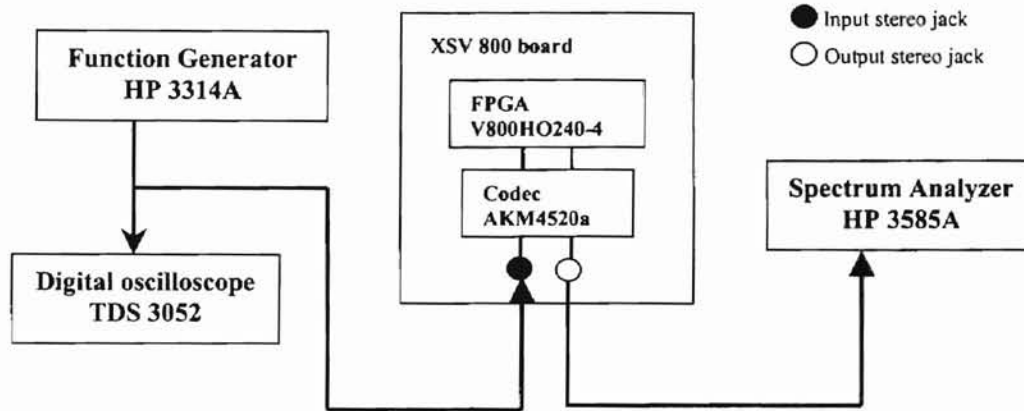


Figure 6.3 Interface between testing equipment and Xess XSV800 board

The way we test the experimental result of the FIR filter on the board is by taking the output of the Codec to the Spectrum Analyzer and the input source to the Digital Oscilloscope to measure the input source magnitude. In this experiment, 100 points of data are taken for verification purposes from DC to 24KHz. This can be achieved by changing the frequency of Function Generator. In this case, the step size for each data point is 240Hz to get 100 points between DC to 24KHz. The magnitude plot on Figure 6.4 shows the frequency response of the implementation results. As we can see, the experiment met the original specifications except that some of the data points are slightly

off in the Stopband region. This may be due to inaccuracy of readings from the spectrum analyzer, as the magnitude of the Stopband region is too small to measure.

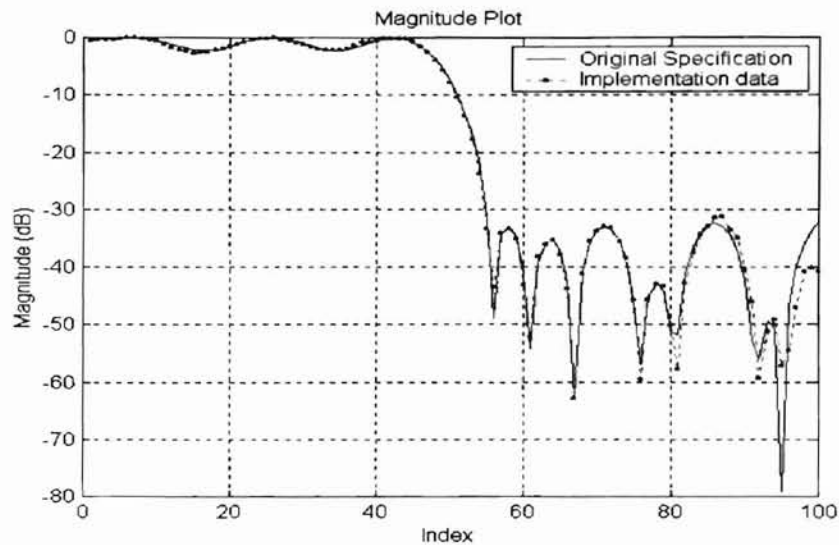


Figure 6.4 Comparison of Implementation data with original specification

6.2.4 Hardware costs

The hardware costs are the main concern of this thesis. The CSD representation and the DM technique are introduced for saving hardware in the FIR filter. The hardware costs saving occur mainly on the hardware required for filter multiplier. As discussed in Chapter 2, we know that the DM technique will have the same or better hardware saving in filter multiplier compared to the CSD representation. However, the structural adder/subtractor and delay element will have the same hardware requirement for both the CSD representation and the DM technique.

In the previous example, the CSD representation and the DM technique have the following costs for hardware shown in Table 6.1. The hardware costs include the filter multiplier, structural adder/ subtractor and delay elements. The first part of Table 6.1 shows the overall costs of the filter using the actual bit size of the filter output required

instead of 20-bit output. The second part of Table 6.1 shows the overall costs of filter with the Codec circuit. As we can see the “number of flip flops”, which acts as the delay elements, is the same for both the CSD representation and the DM technique as both techniques have the same filter order. The number of delay elements required in transpose direct form structure is equal to the number of filter order. However, the size of the delay element for each stage is different. The size of delay element depends on the bit size required for each stage in the filter without truncation. Therefore, the whole filter may have a different size of delay element at each stage. The “number of slices” in Table 6.1 indicates the total number of slices needed for the overall hardware implementation. The “number of 4 input LUTs” in Table 6.1 indicates the number of logic cells needed for the filter implementation. Although each slice has 2 LUTs, the “number of slices” may not be half of the “number of 4 input LUTs” because more LUTs may be needed for getting better routing. In the summary page of Leong’s GUI program [2], the number of shared adders for this filter is 33 and 32 adders respectively for the CSD representation and the DM technique. By looking at the first part of Table 6.1, we can see from the “number of slices” required for filter implementation, the DM technique has a 5.84% of savings over the CSD representation.

Technique		Number of Slices	Number of flip flops	Total number of 4 Input LUTs
Without Codec Circuit	CSD	274	387	448
	DM	258	387	423
With Codec Circuit	CSD	347	490	532
	DM	329	490	506

Table 6.1 Hardware costs for an example of Low-pass filter

6.3 Results and Comparisons

Table 6.2 presents the specifications of four FIR filters for Table 6.3. Table 6.3 shows a list of hardware costs comparison for four basic types of FIR filters, namely Low-pass, High-pass, Band-pass and Band-reject FIR filters. From this table, the savings of the DM technique compared to the CSD representation ranges from 1.79% to 5.84% by comparing the “number of slices” required. The number of “average adder size” is calculated by dividing the “number of 4 input LUTs” by the number of “shared adders”. This average size of adders is the actual size for each adder required for each stage in the filter as in some stage in the filter may not need larger bit sizes to preserve the signals’ information. We can see from Table 6.3 that the average adder size is smaller in the DM technique compared to the CSD representation. The average saving of adder size ranges from 0.08% to 2.65% compared to the DM technique from the CSD representation. The “average delay size” is calculated by taking the “number of flip flops” divided by the number of delays required for the filter. The number of delays is equal to the number of the filter order. The number of “average delay size” is smaller than the output bits of the filter, which indicates that in some filter stages the actual delay size required is smaller than the filter output size and still preserves the signal information.

Low-pass filter: Spec: Fsamp=48KHz Fp=11KHz Fs=13KHz Rp=-3dB Rs=-30dB Input bits=9 Output bit =18 Filter order=24	High-pass filter: Spec: Fsamp=48KHz Fp=11KHz Fs=12KHz Rp=-1dB Rs=-40dB Input bits=13 Output bit =26 Filter order=78	Band-pass filter: Spec: Fsamp=48KHz Fs1=5KHz Fp1=7KHz Fp2=9KHz Fs2=11KHz Rp=-3dB Rs1=Rs2=-40dB Input bits=8 Output bit =19 Filter order=31	Band-reject filter: Spec: Fsamp=48KHz Fs1=9KHz Fp1=9.5KHz Fp2=10KHz Fs2=10.5KHz Rp1=Rp2=-3dB Rs1=-40dB Input bits=9 Output bit =18 Filter order=116
---	---	--	---

Table 6.2 Specification for four different types of FIR filter

Filter type	Technique	Total Adder	Shared Adders	Number of Slices	Number of flip flops	Total number of 4 Input LUTs	Ave. delay size	Ave. Adder size
Low-pass filter	CSD	45	33	274	387	448	16.13	13.58
	DM	43	32	258	387	423	16.13	13.22
High-pass filter	CSD	159	118	1284	1921	2265	24.63	19.19
	DM	157	117	1261	1921	2225	24.63	19.02
Band-pass filter	CSD	67	41	371	538	527	17.36	12.85
	DM	63	39	363	538	501	17.36	12.84
Band-reject filter	CSD	173	110	1080	1923	1387	16.58	12.61
	DM	172	109	1056	1923	1356	16.58	12.44

Table 6.3 Hardware costs in Xilinx FPGA for four different types of FIR filter

Two different Low-pass filter specifications are tested for comparison of the standard hardware costs calculated in Leong's GUI program [2] with the actual hardware costs shown in Table 6.4. The hardware costs calculated in Leong's GUI program [2] are 962 and 840 respectively before the DM technique optimization and after the DM technique optimization in first specification. There is a 12.68% of hardware saving in this specification by using DM technique. In the second specification, the hardware costs calculated in Leong's GUI program [2] are 735 and 342 respectively before the DM technique optimization and after the DM technique optimization. There is 53.47% of hardware saving in this specification. The actual hardware costs are calculated using the "number of slices" needed for implementation. By comparing the "number of slices" before optimization and after optimization, there are 17.35% and 54.97% of hardware savings respectively for the specifications in Table 6.4. Therefore, the VHDL interface itself was responsible for 1.5% to 4.67% additional savings beyond that provided by the Leong's GUI program [2].

Specification	Before DM technique optimization				After DM technique optimization			
	No. of slices	No. of flip flops	Total no. of 4 Input LUTs	Hardware costs	No. of slices	No. of flip flops	Total no. of 4 Input LUTs	Hardware costs
fsamp=20KHz fp=5KHz fs=7KHz Rp=-1.5dB Rs=-40dB	219	255	373	962	181	257	300	840
fsamp=20KHz fp=5KHz fs=8KHz Rp=-3dB Rs=-40dB	191	151	334	735	86	108	137	342

Table 6.4 Comparison of hardware costs

Chapter 7

7. Conclusions and Future Work

7.1 Conclusions

The implementation example in Chapter 6 illustrates that we can use the CSD representation and the DM technique to implement a FIR filter. The implementation results match the specification of the user. This thesis also included the comparison of hardware costs by using the CSD representation and the DM technique in FIR filter design for four different basic types of FIR filters with specific filter specification. The comparison showed that the savings of hardware range from 1.79% to 5.84% for these four FIR filters. Besides that, two Low-pass filter examples in Chapter 6 showed that the VHDL interface itself is responsible for an additional hardware savings of 1.5% to 4.67% by using the DM technique. Due to no limitation of user specification in Leong's GUI program [2], there is no comparison included in this thesis of overall hardware savings for all FIR filters. The behavioral VHDL coding for FIR filter in Chapter 5 is generally written so that these codes can be used for implementation in other technologies. In conclusion, the DM technique achieves a better hardware savings than the CSD representation. However, as mentioned before, speed of filter in the DM technique may not perform as well as the CSD representation.

7.2 Future Work

The implementations in this thesis are mainly targeted for Xilinx Virtex XCV800HQ240-4 FPGAs. More technologies such as MOSIS or SOI can be considered in future work. Besides that, the VHDL behavioral coding in this thesis generated the hardware representation for half of the filter coefficients and shared the other half of the coefficients due to the symmetrical characteristic of FIR filter. However, there is no sharing of hardware in between the filter coefficient that may have the similar number of shifts. For example, the coefficients 0.65625 can share its hardware structure with the coefficient 0.625, as coefficient 0.65625 is the sum of 2^{-1} , 2^{-3} and 2^{-5} and coefficient 0.625 is just the sum of 2^{-1} and 2^{-3} . We can use coefficient 0.625 as the main structure and add a 2^{-5} to get coefficient 0.65625. Figure 7.1 illustrates this example in hardware view. In this case, the actual hardware required for implementing these two filter coefficients is only two adders instead of three adders. Therefore, more sharing of hardware between the filter coefficients can be achieved if the filter has the characteristic described above.

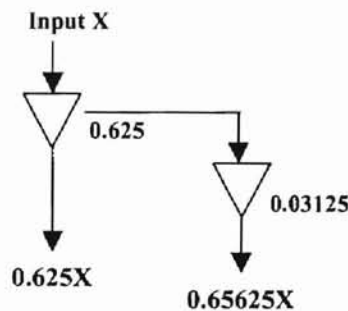


Figure 7.1 Example of sharing filter coefficient

Transpose direct form filter structure is used as the main structure of the FIR filter in this thesis. However, there may be other structures that may have more hardware

savings. Future work may consider different structures that have more savings in implementing the FIR filter. We did not take the Signal to Noise ratio (SNR) option in Leong's GUI program [2] into consideration. Future work may consider implementing a different VHDL interface, which has SNR ability built in.

Bibliography

- [1] Dannielle L Husinga, “ Design of Optimized Filters Using CSD Coefficient Representation”, Master’s Thesis, ECE Dept, UC Davis, California, Mar. 1996.
- [2] Wen Fung Leong, “Optimizing FIR filters coefficients using CSD representation and DM technique”, Master’s Thesis, ECEN Dept, Oklahoma State University, Oklahoma, May 2002.
- [3] Herman J. Blinchikoff & Anotol I. Zverev, “Filter in the time and frequency domains”, Robert E. Krieger Publishing Company, 1987.
- [4] Alan V. Oppenheim, Ronald W. Schafer, “Discrete-time signal processing”, Prentice Hall, 1999.
- [5] Litwin. L . “FIR and IIR digital filter,” IEEE Potential, Volume 19, Issue 4,pp. 28-31, Oct.- Nov. 2000.
- [6] Dempster, A.G.; Macleod, M.D, “Use of minimum-adder multiplier blocks in FIR digital filters”, Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on Circuit and System, Vol.42, Issue: 9, pp.569 –577, Sept., 1995.
- [7] Dempster, A.G.; Macleod, M.D., “Comments on Minimum number of adders for implementing a multiplier and its application to the design of multiplierless digital filters”, Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on Circuit and System, Vol.45 Issue: 2, pp.242 –243, Feb. 1998.

- [8] Dempster, A.G.; Macleod, M.D., "Comparison of fixed-point FIR digital filter design techniques", *Circuits and Systems II: Analog and Digital Signal Processing*, IEEE Transactions on Circuit and System, Vol.44 Issue: 7, pp.591 –593, July 1997.
- [9] Soderstrand, M.A.; Johnson, L.G.; Arichanthiran, H.; Hoque, M.D.; Elangovan, R., "Reducing hardware requirement in FIR filter design", *Acoustics, Speech, and Signal Processing*, 2000. ICASSP '00. Proceedings. 2000 IEEE International Conference on, Vol.6, pp.3275 –3278, 2000.
- [10] Kah-Howe Tan; Wen Fung Leong; Kadam, S., Soderstrand, M.A.; Johnson, L.G., "Public-domain MATLAB program to generate highly optimized VHDL for FPGA implementation", *Circuits and Systems*, 2001. ISCAS 2001. The 2001 IEEE International Symposium on Circuit and System, Vol. 4, 2001 pp.514 –517, May, 2001.
- [11] FPGA general information, <http://www.andraka.com/whatisan.htm>
- [12] FPGA general information, <http://www.vcc.com/fpga.html>
- [13] John V. Oldfield; Richard C. Dorf, "Field-Programmable Gate Arrays: Configurable Logic for rapid prototyping and implementation of digital systems", John Wiley & Sons, Inc, 1995.
- [14] Xilinx official web page, <http://xilinx.com>
- [15] Xilinx product data sheets, <http://xilinx.com/partinfo/ds003.htm>
- [16] Xess Corp. XSV800 prototyping board manual, http://www.xess.com/manuals/xsv-manual-v1_1.pdf
- [17] XSV tools manual, http://www.xess.com/manuals/xstools-v4_0.pdf

- [18] Muthya S. Patharlanka, "Automatic Generation of VHDL for Optimized Filters Using CSD", Master's Thesis, ECE Dept, UC Davis, 1996.
- [19] Jayaram Bhasker, "A VHDL Primer," Englewood Cliff, New Jersey, Prentice-Hall, 1992.
- [20] Douglas J. Smith, "HDL chip design: A Practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or verilog", Madison, Doone Publications, 2000.

Appendix A

```
% This m file is used to generate a parameter package file for FIR filter in VHDL
%-----
%Check if user choose for CSD representation or DM technique
if dmval ==2
%Following lines created the params.vhd for CSD technique

%Created stage parameter for CSD representation. In this case, bstage=0 if filter
coeff.=0 else bstage=1
for i=1:Nmin+1
    if QC(i)==0
        bstage(i)=0;
    else
        bstage(i)=1;
    end
end

%Print the following lines onto the params.vhd file
fid=fopen('params.vhd','w');
fprintf(fid,'Library IEEE;\n');
fprintf(fid,'use IEEE.std_logic_1164.all;\n');
fprintf(fid,'use IEEE.std_logic_unsigned.all;\n\n');
fprintf(fid,'PACKAGE params IS\n\n');
fprintf(fid,'TYPE int_arr1 IS ARRAY(natural range <>) OF INTEGER;\n');
fprintf(fid,'TYPE int_arr2 IS ARRAY(natural range <>) OF INTEGER;\n');
fprintf(fid,'TYPE int_arr3 IS ARRAY(natural range <>) OF INTEGER;\n\n');
fprintf(fid,'CONSTANT N: INTEGER:= %2d;\n',Nmin);
fprintf(fid,'CONSTANT b: INTEGER:=%2d;\n',bin);

%print b_stage onto the params.vhd file
fprintf(fid,'CONSTANT b_stage: int_arr1(0 to %3d):=(' ,Nmin);
for i=1:Nmin+1
    if i<=Nmin
        fprintf(fid,'%3d, ',bstage(i));
    else
        fprintf(fid,'%3d',bstage(i));
    end
end
fprintf(fid,');\n');

%Find the number of nonzero bit in each coefficient and locate the nonzero bit in each
coefficient
[M,N]=size(newtable);
g=1; t=1;
for i=1:M
    if bstage(i)==0

    else
        numrow(t)=0;
        for j=1:bmin
            if newtable(i,j)==-1
                numrow(t)=numrow(t)+1;
                posi(g)=-(j-1);
                g=g+1;
            elseif newtable(i,j)==1
                numrow(t)=numrow(t)+1;
                posi(g)=(j-1);
                g=g+1;
            end
        end
        t=t+1;
    end
end

%print n_per_row onto the params.vhd file
fprintf(fid,'CONSTANT n_per_row: int_arr2(0 to %3d):= (' ,t-2);
for i=1:t-1;
    if i<t-1
        fprintf(fid,'%3d, ',numrow(i));
```

```

else
    fprintf(fid, '%3d', numrow(i));
end
end
fprintf(fid, ');\n');

%print coeff_vec onto the params.vhd file
fprintf(fid, 'CONSTANT coeff_vec: int_arr3(0 to %3d):=(' ,g-2);
for i=1:g-1;
    if i<g-1
        fprintf(fid, '%3d, ', posi(i));
    else
        fprintf(fid, '%3d', posi(i));
    end
end
fprintf(fid, ');\n');

%Check max number of shift needed in the whole filter
s=1; maxb=0;
for i=1:t-1
    if numrow(i)~=0
        for j=1:numrow(i)
            maxbb=abs(posi(s));
            s=s+1;
            if maxbb >=maxb
                maxb=maxbb;
            end
        end
    end
end

%print zero_con, bshift onto the params.vhd file
fprintf(fid, 'CONSTANT zero_con: INTEGER:= %2d;\n', maxb+1);
fprintf(fid, 'CONSTANT bshift: INTEGER:= %2d;\n', bin+maxb+1);

%Pad zero if bshift <20 for codec usage
if (bin+maxb+1)<=20
    yextra=20-(bin+maxb+1);
else
    yextra=0;
end
fprintf(fid, 'CONSTANT yex: INTEGER:= %2d;\n', yextra);

ODD=rem(Nmin, 2);
if ODD==1
    NN=(Nmin+1)/2-1;
else
    NN=Nmin/2;
end
fprintf(fid, 'CONSTANT NN: INTEGER:= %2d;\n', NN);
fprintf(fid, 'CONSTANT ODD: INTEGER:= %2d;\n\n', ODD);

fprintf(fid, 'END package;');
fclose(3);
%End of generating CSD params.vhd

%-----
else
%Following lines created the params.vhd for DM technique
clear bstage cdmoptnew dmtable
cdmoptnew=cdmopt;
[M,N]=size(cdmoptnew);

%Find number of stage in each coefficient
for i=1:M;
    t=0;
    for j=1:N;
        if cdmopt(i,j)~=0 ;
            t=t+1;
        end
    end
end
end

```

```

    bstage(i)=t;
end

%Flipping the negative sign of the first section in cdmopt to the end section in cdmopt
for i=1:M;
    if (cdmopt(i,1)<0) & (bstage(i)>1);
        if (cdmopt(i,1)==0) & (cdmopt(i,2)==0);
            for j=N:-1:1;
                if cdmopt(i,j)==0;
                    cdmoptnew(i,1)=abs(cdmopt(i,1));
                    cdmoptnew(i,j)=-cdmopt(i,j);
                    break
                else
                    end
            end
        else
            end
        end
    end
end

%create dhtable;
tsize=0;
for i=1:M;
    for j=1:N;
        if cdmoptnew(i,j)==0;

            else
                tsize=tsize+1;
            end
        end
    end
end
dhtable=zeros(tsize,bmin);

%Finding csd representation for all dm coefficients
t=1;
for i=1:M
    if bstage(i)==0;

    elseif (bstage(i)==1);
        x=cdmoptnew(i,1)/(2^(bmin-1));
        dhtable(t,:)=csd(x,1,(bmin-1));
        t=t+1;
    elseif (bstage(i)==2) & (cdmoptnew(i,2)==0);
        x=cdmoptnew(i,1)/(2^(bmin-1));
        dhtable(t,:)=csd(x,1,(bmin-1));
        t=t+1;
    else
        pwl=0;
        for j=1:bstage(i);
            if j==bstage(i);
                x=cdmoptnew(i,j)/(2^(bmin-pwl-1));
                dhtable(t,:)=csd(x,1,(bmin-1));
            else
                g=0;
                h=0;
                while g==0;
                    if cdmoptnew(i,j)<=(2^h);
                        g=1;
                    end
                    h=h+1;
                end
                x=cdmoptnew(i,j)/(2^(h-1));
                dhtable(t,:)=csd(x,1,(bmin-1));
                pwl=pwl+h-1;
            end
            t=t+1;
        end
    end
end
end
end

```

```

%Getting position of the nonzero bits also number of nonzero bits in each small section
of csdconstant
[u,y]=size(newtable);
g=1; t=1;
for i=1:u
    if bstage(i)==0

    else
        for e=1:bstage(i)
            numrow(t)=0;
            for j=1:bmin
                if dmtable(t,j)==-1
                    numrow(t)=numrow(t)+1;
                    posi(g)=-(j-1);
                    g=g+1;
                elseif dmtable(t,j)==1
                    numrow(t)=numrow(t)+1;
                    posi(g)=(j-1);
                    g=g+1;
                end
            end
            t=t+1;
        end
    end
end

%Print the following lines onto the params.vhd file
fid=fopen('params.vhd','w');
fprintf(fid,'Library IEEE;\nuse IEEE.std_logic_1164.all;\n');
fprintf(fid,'use IEEE.std_logic_unsigned.all;\n\n');
fprintf(fid,'PACKAGE params IS\n\n');
fprintf(fid,'TYPE int_arr1 IS ARRAY(natural range <>) OF INTEGER;\n');
fprintf(fid,'TYPE int_arr2 IS ARRAY(natural range <>) OF INTEGER;\n');
fprintf(fid,'TYPE int_arr3 IS ARRAY(natural range <>) OF INTEGER;\n\n');
fprintf(fid,'CONSTANT N: INTEGER:= %2d;\n',Nmin);
fprintf(fid,'CONSTANT b: INTEGER:=%2d;\n',bin);

%print b_stage onto the params.vhd file
fprintf(fid,'CONSTANT b_stage: int_arr1(0 to %3d):=(' ,M-1);
for i=1:M;
    if i<M
        fprintf(fid,'%3d,',bstage(i));
    else
        fprintf(fid,'%3d',bstage(i));
    end
end
fprintf(fid,');\n');

%print n_per_row onto the params.vhd file
fprintf(fid,'CONSTANT n_per_row: int_arr2(0 to %3d):=(' ,t-2);
for i=1:t-1;
    if i<t-1
        fprintf(fid,'%3d,',numrow(i));
    else
        fprintf(fid,'%3d',numrow(i));
    end
end
fprintf(fid,');\n');

%print coeff_vec onto the params.vhd file
fprintf(fid,'CONSTANT coeff_vec: int_arr3(0 to %3d):=(' ,g-2);
for i=1:g-1;
    if i<g-1
        fprintf(fid,'%3d,',posi(i));
    else
        fprintf(fid,'%3d',posi(i));
    end
end
fprintf(fid,');\n');
posinew=posi;

```



```

%Check max number of shift needed in the whole filter
s=1; maxb=0; y=1;
for i=1:M
    if bstage(i)~=0
        d=bstage(i);
        maxbb=0;
        for j=1:d
            maxn=0;
            maxnn=0;
            if j>=2
                for k=1:numrow(y)
                    if k>=2
                        maxnn=abs(posi(s));
                    else
                        maxn=abs(posi(s));
                    end
                    if maxnn>maxn
                        maxn=maxnn;
                    end
                    s=s+1;
                end
                maxbb=maxbb+maxn;
            else
                for k=1:numrow(y)
                    if k>=2
                        maxnn=abs(posi(s));
                    else
                        maxn=abs(posi(s));
                    end
                    if maxnn>maxn
                        maxn=maxnn;
                    end
                    s=s+1;
                end
                maxbb=maxn;
            end
            if maxbb >=maxb
                maxb=maxbb;
            end
            y=y+1;
        end
    end
end
end
fprintf(fid,'CONSTANT zero_con: INTEGER:=%2d;\n',maxb+1);
fprintf(fid,'CONSTANT bshift: INTEGER:= %2d;\n',bin+maxb+1);

%Pad zero if bshift <20
if (bin+maxb+1)<=20
    yextra=20-(bin+maxb+1);
else
    yextra=0;
end
fprintf(fid,'CONSTANT yex: INTEGER:= %2d;\n',yextra);

ODD=rem(Nmin,2);
if ODD==1
    NN=(Nmin+1)/2-1;
else
    NN=Nmin/2;
end
fprintf(fid,'CONSTANT NN: INTEGER:= %2d;\n',NN);
fprintf(fid,'CONSTANT ODD: INTEGER:= %2d;\n\n',ODD);
fprintf(fid,'END package;');
fclose(3)

%End of generating DM params.vhd
end

%Copy file to the directory for hardware implementation
copyfile('C:\csddm\params.vhd','c:\csddm\FIRfilter\params.vhd')

```

Appendix B

Appendix B-1

Sample params.vhd file:

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

PACKAGE params IS

TYPE int_arr1 IS ARRAY(natural range <>) OF INTEGER;
TYPE int_arr2 IS ARRAY(natural range <>) OF INTEGER;
TYPE int_arr3 IS ARRAY(natural range <>) OF INTEGER;

CONSTANT N: INTEGER:= 24;
CONSTANT b: INTEGER:= 9;
CONSTANT b_stage: int_arr1(0 to 24):=(1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1);
CONSTANT n_per_row: int_arr2(0 to 22):=( 1, 1, 1, 2, 1, 1, 1, 2, 3, 1, 4, 2, 4, 1, 3, 2, 1, 1, 1, 2, 1, 1, 1);
CONSTANT coeff_vec: int_arr3(0 to 37):=(-6, -6, 7, 5, -7, 8, -6, 7, 5, 8, -4, 6, -8, 7, 2, -4, -6, 8, 2, 6, 2, -4, -6,
8, 7, -4, 6, -8, 5, 8, 7, -6, 8, 5, -7, 7, -6, -6);
CONSTANT zero_con: INTEGER:= 9;
CONSTANT bshift: INTEGER:= 18;
CONSTANT yex: INTEGER:= 2;
CONSTANT NN: INTEGER:= 12;
CONSTANT ODD: INTEGER:= 0;

END package;
```

Appendix B-2

Sample filt.vhd file:

```
--This file is used to generate the behavior of the FIR filter. Three functions are used to perform the behavior of the --
--filter.
--(1) snd_shift: use to perform the shift function of the filter.
--(2) zero_vec: use to generate a zero vector that its size equal to the maximum adder size.
--(3 )xstage: use to perform each tap of the filter coefficient multiplication with input X
-----

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.Numeric_STD.all;
use IEEE.std_logic_unsigned.all;
use work.params.all;

PACKAGE filt IS

TYPE twod_data_mat IS ARRAY(natural range <>) OF unsigned(0 TO bshift-1);

FUNCTION snd_shift ( s : INTEGER; v: unsigned(0 to bshift-1) ) RETURN unsigned;
FUNCTION zero_vec ( a : INTEGER) RETURN unsigned;
FUNCTION xstage( X: unsigned(0 TO bshift-1) ) RETURN twod_data_mat;

END package;

PACKAGE BODY filt IS
-----
FUNCTION snd_shift ( s : INTEGER; v: unsigned(0 to bshift-1) ) RETURN unsigned IS
VARIABLE result : unsigned(0 to bshift-1);
VARIABLE R,k: INTEGER;

BEGIN
k:=0;
R:=bshift-1;
FOR i IN 0 TO R LOOP
    IF i<= s-1 THEN
        result(i):=v(0);
    ELSE
        result(i):=v(k);
        k:=k+1;
    END IF;
END LOOP;
RETURN result;
END snd_shift;

-----
FUNCTION zero_vec ( a : INTEGER ) RETURN unsigned IS
VARIABLE result: unsigned (0 to bshift-1);

BEGIN

FOR i IN 0 TO bshift-1 LOOP
result(i):='0';
END LOOP;
RETURN result;
END zero_vec;
-----
```

```

FUNCTION xstage( X: unsigned(0 TO bshift-1) ) RETURN twod_data_mat IS
VARIABLE data_mat: twod_data_mat (0 to NN) ;
VARIABLE tot,np,s: INTEGER;
VARIABLE XS, XS1,XS_int: unsigned(0 TO bshift-1);

BEGIN
tot:=0;
np:=0;
XS_int:=zero_vec(bshift);
FOR h IN 0 to NN LOOP
  IF b_stage(h)>= 1 THEN
    XS:= X;
    FOR i IN 0 to (b_stage(h)-1) LOOP
      FOR k IN 0 to (n_per_row(np)-1) LOOP
        IF coeff_vec(tot) = 0 THEN
          XS1:= X;
        ELSE
          XS1:=XS;
        END IF;
        ELSE
          IF coeff_vec(tot)<0 THEN
            s:=abs(coeff_vec(tot));
          ELSE
            s:=coeff_vec(tot);
          END IF;
          XS1:= snd_shift(s,XS);
        END IF;
        IF (coeff_vec(tot)<0) THEN
          XS_int:=XS_int-XS1;
        ELSE
          XS_int:=XS_int+XS1;
        END IF;
        tot :=tot+1;
      END LOOP;
      np:= np+1;
      XS:=XS_int;
      XS_int:=zero_vec(bshift);
    END LOOP;
  ELSE
    XS:= zero_vec(bshift);
  END IF;
  data_mat(h):=XS;
END LOOP;
RETURN data_mat;
END xstage;
-----
END filt;

```

Appendix B-3

Sample rdfl.vhd file:

```
-- FIR FILTER DESIGN USING CSD/DM technique. --
-- This file is the top level of the overall FIR filter. --
-- This file calls the functions and constants from the params.vhd and filt.vhd files. --
-----
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.Numeric_STD.all;
use IEEE.std_logic_unsigned.all;
use work.params.all;
use work.filt.all;

ENTITY rdfl IS

    PORT(
        X:IN unsigned(0 to b-1);
        RESET: IN std_logic;
        CLK: IN std_logic;
        Ycodec: OUT unsigned(19 downto 0)

    );
END rdfl;

ARCHITECTURE behave OF rdfl IS
Signal Y:unsigned(bshift-1 downto 0);
Signal A,B,C,D:twod_data_mat(0 to N);
Signal result: unsigned(0 to zero_con -1);
Signal result_final : unsigned(0 to bshift-1);
Signal yout: unsigned(0 to yex);
BEGIN

name:process(X,RESET,CLK,A,B,C,D)

VARIABLE ZERO: unsigned(0 to bshift-1);
VARIABLE XS: twod_data_mat(0 to NN);
VARIABLE j,g: integer;
BEGIN

FOR i IN 0 TO zero_con-1 LOOP
    result(i) <='0';
END LOOP;

IF yex >0 THEN
FOR i IN 0 TO yex LOOP
    yout(i)<='0';
END LOOP;
END IF;
result_final <= (X & result);

ZERO:=zero_vec(bshift);
XS:=xstage(result_final);

        C(0) <= XS(0);
        A(0) <= C(0);
```

```

j:=1;
g:=1;
filter:FOR k IN 1 TO N LOOP

  IF (RESET = '1') THEN
    D(k-1) <= ZERO;
  ELSIF rising_edge(CLK) THEN
    D(k-1) <= A(k-1);
  END IF;
  C(k) <= XS(g);
  B(k-1) <= D(k-1) + C(k);
  A(k) <= B(k-1);
  if j < NN THEN
    j:=j+1;
    g:=g+1;
    elsif (j=NN AND ODD=1) THEN
      g:=g;
      j:=j+1;
    elsif (j=NN AND ODD=0) THEN
      g:=g-1;
      j:=j+1;
    else
      g:=g-1;
      j:=j+1;
    end if;

END LOOP;

Y <= B(N-1);
  IF bshift >= 20 THEN
    Ycodec <= Y(bshift-1 downto bshift-20);

  ELSE
    Ycodec <= (Y & yout(0 to yex-1));
  END IF;
END PROCESS name;

END behave;

```

Appendix C

TCL script for Virtex XCV800HQ240-4 FPGAs (filter.tcl)

```
#!/bin/wish
project -new filter
add_file -vhdl "params.vhd"
add_file -vhdl "filt.vhd"
add_file -vhdl "rdfl.vhd"
add_file -vhdl "codec.vhd"
add_file -vhdl "clkgen.vhd"
add_file -vhdl "channel.vhd"
add_file -vhdl "codec_intfc.vhd"
add_file -vhdl "control.vhd"
add_file -vhdl "codec_control.vhd"
set_option -technology Virtex
set_option -part XCV800
set_option -speed_grade -4
set_option -package HQ240
set_option -top_module codec_control
project -save
project -run
```

Appendix D

Pin assignment filter implementation in Xess XSV800 prototyping board

```
#
# Assign pin for XSV800 prototyping board from Xess Corp.
#
NET "clk" LOC = "P89";
NET "mclk" LOC = "P3";
NET "lrck" LOC = "P4";
NET "sclk" LOC = "P5";
NET "sdin" LOC = "P6";
NET "sdout" LOC = "P7";
NET "reset" LOC = "P140";
NET "adc_overnrun" LOC = "P152";
NET "dac_underrun" LOC = "P154";
```

VITA

Kah-Howe Tan 2

Candidate for the degree of

Master of Science

Thesis: EFFICIENT HARDWARE IMPLEMENTATION OF FINITE IMPULSE
RESPONSE FILTERS IN FIELD PROGRAMMABLE GATE ARRAYS

Major field: Electrical Engineering

Biographical:

Personal Data: Born in Kuching, Sarawak, Malaysia, on February 1, 1977, the son of Tan and Chong.

Education: Graduated from Chung Hua Middle School No.1, Kuching, Sarawak, Malaysia in December 1995; received Bachelor of Science degree in Electrical Engineering from Oklahoma State University, Stillwater, Oklahoma in July 2000. Completed the requirements for the Master of Science degree with a major in Electrical Engineering at Oklahoma State University in May, 2002.

Experience: Employed by Electrical and Computer Engineering Department at Oklahoma State University as a graduate research assistant and teaching assistant, 2000 to present.

Professional Memberships: IEEE society.