Integrated platform for design and verification of digital FIR filters

Sharma Udit

2007

Sharma, U. (2007). Integrated platform for design and verification of digital FIR filters.
Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/3520>

<https://doi.org/10.32657/10356/3520>

Nanyang Technological University

Downloaded on 26 Apr 2023 18:24:16 SGT

**An Integrated Platform for Design and Verification
Of
Digital FIR Filters**

Sharma Udit

School of Electrical & Electronic Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of
Master of Engineering

2007

Acknowledgements

I would like to take this opportunity to thank my supervisor, Associate Professor Jong Ching Chuen, for his supervision throughout this research programme. In particular, it was his patience and motivation that paved the way for me. His invaluable suggestions enriched my research experience and made it a fruitful journey.

I would also like to thank Associate Professor Chang Chip Hong, for getting me acquainted with this exciting research topic, for all the invaluable suggestions and timely discussions during the development of our software and the organization of the experimental results, and for tirelessly helping me to correct and improve the manuscripts of the papers. It was his endless energy that kept me motivated through out this process

I would like to thank Ms. Xu Fei, Mr. Chen Jiajia, Mr. Shibu Menon, Mr. Mathias Faust and Mr. Markus Rahle for their help, suggestions and collaboration pertaining to the work in this research programme.

I would like to express my gratitude to my parents for their unwavering support and encouragement. I am also grateful to my friend, Sonali, for her constant reminders and enquiries about the progress of my research.

Last but not least, to the staff of Centre for High Performance Embedded Systems, I wish to convey my appreciation to all of them for their kind and friendly assistance.

Abstract

FIR filter is an ineluctable component of many DSP applications. Many different structures of FIR filters exist and each choice provides some trade-off between the hardware complexity and the achievable throughput rate. One of the popular structures is the transposed direct form where the input is fed to all the coefficient multipliers in parallel and the regularity of the linear accumulation makes it highly amenable to implementation through a silicon compiler. The core operation of a transposed direct form FIR filter can be modeled as a general Multiple Constant Multiplications (MCM) problem in high-level synthesis. MCM refers to a group of multiplications of different constants with the same variable or with delayed versions of the same variable. The effectiveness of common subexpression elimination (CSE) algorithm to reduce the adder cost the MCM block directly influences the performance of the application specific integrated circuit (ASIC) when the filter solution is coded in hardware description language and synthesized.

Many algorithms have been developed to minimize the cost of implementing an application specific filter. The optimization criteria considered are typically measured in terms of logic complexity and logic depth based on the number of discrete adders. For a given set of coefficients, some algorithms work better than others and the logic complexity and logic depth information given by the algorithms have no clear correlation to the physical implementation results. For highly constrained specifications, designers may need to manually implement various solutions generated by different algorithms in order to find one that meet their expectation. Hand coding each filter is a time consuming and onerous task. It was therefore imperative to develop an integrated environment to facilitate automatic generation of synthesizable VHDL codes from these solutions.

This work aimed at creating a unified platform to automate the generation of VHDL codes for the physical synthesis of FIR filters in both direct and transposed direct form filters. The aim was to provide designers and researchers with a tool to analyze the physical performances of different filter solutions in terms of VLSI area, delay and power consumption. Many benchmark filters were collected for such evaluation. This platform also performs a functional verification of the VHDL codes by running parallel simulations in a simulator and comparing the final output with its own response to randomly generated stimuli, thereby avoiding manual supervision.

CONTENTS

ACKNOWLEDGEMENTS	i
ABSTRACT	ii
CONTENTS	iii
LIST OF FIGURES	vii
LIST OF TABLES	ix
LIST OF ABBREVIATIONS	x

CHAPTER 1

INTRODUCTION	1
---------------------	----------

1.1	Motivation	1
1.2	Objectives	3
1.3	Major Contributions	4
1.4	Organization of the Thesis	5

CHAPTER 2

BACKGROUND AND EXISTING WORKS	7
--------------------------------------	----------

2.1	Linear-Phase FIR Filter	7
2.2	Signed Digit Representations	9
2.3	Adders	10
2.3.1	Standard Adders	10
2.3.2	Carry Propagation Adder	11
2.3.2.1	Ripple Carry Adder (RCA)	11
2.3.2.2	Carry Look-ahead Adder (CLA)	12
2.4	Carry-Save Adder Tree	14
2.5	Correction Vector	14
2.6	Merged Arithmetic Architecture	16
2.7	Reduction Trees	19
2.7.1	Wallace Tree	19
2.7.2	Dadda Tree	21
2.8	Common Subexpression Sharing	22

2.8.1	BIN	23
2.8.2	CSD	24
2.8.3	BHM	24
2.8.4	MRAG	24
2.8.5	C1	25
2.8.6	NRSCSE	25
2.8.7	PASKO	25
2.8.8	CRAH-2	26
2.8.9	CRA	26
2.8.10	MBPT	26
2.8.11	BCSE	27
2.8.12	4BCSE	27
2.8.13	HARTLEY	27

CHAPTER 3

DEVELOPMENT OF sys-FIR 29

3.1	sys-FIR Overview	29
3.2	Inputs to sys-FIR	32
3.2.1	Direct Form	32
3.2.2	Transposed Direct Form	34
3.3	Common Subexpression Elimination in sys-FIR	38
3.4	sys-FIR Planning and Design Considerations	39
3.4.1	Symmetric/Asymmetric Implementation	39
3.4.2	Input Range Analysis	41
3.4.3	Inherent Robustness	41
3.4.4	Automatic Verification	42
3.5	sys-FIR System Description	42
3.5.1	Implementation	43
3.5.1.1	Filter Structure	43
3.5.1.2	Correction Vector	44
3.5.1.3	Handling Vertical/Oblique Subexpressions	45

3.5.2	sys-FIR User Interfaces	46
3.5.3	sys-FIR Features	50
3.5.3.1	Modularity	50
3.5.3.2	Forward Compatibility	53
3.5.3.3	Verification	55
3.5.3.4	Debugging	55
3.5.3.5	Script Generation for DC	56
3.5.3.6	Handling of Special Cases	56
3.5.3.7	Modified-RCA and CLA	57
3.5.3.8	Optimization of CSA Trees	60
3.5.3.9	Use of Recursive Adder Structures	63
3.5.3.10	Uniform Naming Convention	63
3.5.3.11	Input Range Compatibility	63
3.5.3.12	Redundancy Removal	64

CHAPTER 4

RESULTS AND DISCUSSION 65

4.1	The Design Flow	65
4.2	Direct Form Implementation	68
4.3	Transposed Direct Form Filters	73
4.3.1	VLSI metrics comparison for FIR filters generated with various CSE algorithms	74
4.3.2	Direct form vs Transposed direct form Comparison	75
4.3.3	Research on novel CSE techniques	77
4.3.4	Different Input Length Comparison	77
4.3.5	Power Analysis	78

CHAPTER 5

CONCLUSIONS AND FUTURE WORK 82

5.1 Conclusions 82

5.2 Recommendations for Future Work 83

AUTHOR’S PUBLICATIONS 86

REFERENCES 87

APPENDIX I (Filter Database) a

APPENDIX II (CRAH2 MATLAB link to TFF Kernel) b

APPENDIX III (s-FCG Output) g

APPENDIX IV (Sample Top-level VHDL File created by sys-FIR) l

List of Figures

Fig. 2.1	Direct Form FIR Filter	9
Fig. 2.2	Transposed Direct Form FIR Filter	9
Fig. 2.3	Half Adder	11
Fig. 2.4	Full Adder	11
Fig. 2.5	Full Adder Block Diagram	11
Fig. 2.6	3-bit RCA	12
Fig. 2.7	16-bit CLA	13
Fig. 2.8	Signed Integers Arithmetic	15
Fig. 2.9	Sign Extension	15
Fig. 2.10	Alternative to Sign Extension	16
Fig. 2.11	Derivation of the merged adder structure	18
Fig. 2.12	CSA Tree Reduction using Wallace's Strategy	20
Fig. 2.13(a)	Adding 6-bit numbers using Wallace Tree	21
Fig. 2.13(b)	Adding 6-bit numbers using Dadda Tree	21
Fig. 2.14	Example Illustrating Subexpression Sharing	22
Fig. 2.15	Full Precision CSD Multiplier	23
Fig. 2.16	Multiplier using Common Subexpression Sharing	23
Fig. 3.1	sys-FIR Overview	31
Fig. 3.2	26-tap Filter Coefficient File	32
Fig. 3.3	Sample Direct Form internal standard File (DFF)	33
Fig. 3.4(a)	CRA Output	34
Fig. 3.4(b)	CRAH2 Output	34
Fig. 3.5	Sample Transposed Direct Form internal standard File (TFF)	35
Fig. 3.6	Sample Coefficient Set with V/O Subexpression	36
Fig. 3.7	V/O Subexpression Declaration in TFF	37
Fig. 3.8	Subexpression Sharing	38
Fig. 3.9	Transposed Direct Form Implementation	40
Fig. 3.10	6-Tap Symmetric Filter	40
Fig. 3.11	Filter Structure implemented by sys-FIR	43

Fig. 3.12	ADDNDELAY Block with CSA option	44
Fig. 3.13	Issue of Correction Vector Implementation	45
Fig. 3.14	CSVOS Block for Vertical/Oblique Subexpressions	46
Fig. 3.15	sys-FIR: Top-Level GUI	47
Fig. 3.16	DFF/TFF Generation GUI	48
Fig. 3.17	s-VHDG GUI	49
Fig. 3.18	MATLAB Code Structure	51
Fig. 3.19	VHDL Generation	52
Fig. 3.20	TFF Generator Kernel	54
Fig. 3.21	Bit Pattern for CPA	57
Fig. 3.22	Standard-RCA vs. Modified-RCA	58
Fig. 3.23	CLA implementation for bit pattern in Fig. 3.21	59
Fig. 3.24	4-bit Standard-CLA logic	59
Fig. 3.25	4-bit Modified-CLA logic	60
Fig. 3.26	Shifting Operation	61
Fig. 3.27	Carry Save Adder Tree Reduction	61
Fig. 3.28	Recursive Adder Implementation	63
Fig. 4.1	Filter Design Flow using sys-FIR	67
Fig. 4.2	Hybrid Column Compression for Direct Form FIR Architecture	69
Fig. 4.3	Carry Look-ahead Adder for Final Accumulation	70
Fig. 4.4	64-tap FIR Filter Layout	71
Fig. 4.5	Direct vs Transposed Direct Form Delay Comparison	76
Fig. 4.6	Direct vs Transposed Direct Form Area Comparison	76
Fig. 4.7	CSE Dependent Area Requirements of Tommi_n281_16bit Filter	77
Fig. 4.8	Input Signal Wordlength Impact on Filter Size	78
Fig. 4.9	Synopsys Dynamic Power Simulation Flow	79
Fig. 4.10	Dynamic Power Consumption of filter samueli01_n25	80
Fig. 4.11	Dynamic Power Consumption of filter samueli02_n60	81

List of Tables

Table 2.1	Maximum tree height $n(h)$ for a h -level Carry Save Adder tree	19
Table 3.1	Hardware Utilization in Example of Fig 3.27	62
Table 4.1	Hardware Costs of Different CSA tree Reduction Techniques	70
Table 4.2	64-tap FIR Filters Power Consumption	72
Table 4.3	FIR Filters Comparison	73
Table 4.4	VLSI Metrics of dempster_n25 Filter from Solutions of Different CSE algorithms	74
Table 4.5	VLSI Metrics of Tommi_n281_16bit Filter from Solutions of Different CSE algorithms	75

List of Abbreviations

BCSE	– Binary Common Subexpression Elimination
CLA	– Carry Look-Ahead Adder
CPA	– Carry Propagation Adder
CRA	– Contention Resolution Algorithm
CSA	– Carry Save Adder
CSD	– Canonical Signed Digit
CSE	– Common Subexpression Elimination
DFF	– Direct Form internal standard File
FA	– Full Adder
FCF	– Filter Coefficient File
FIR	– Finite Impulse Response
HA	– Half Adder
IIR	– Infinite Impulse Response
MBPT	– Multirooted Binary Partition Tree
MRAG	– Modified Reduced Adder Graph
NRSCSE	– Non Recursive Signed Common Subexpression Elimination
RCA	– Ripple Carry Adder
s-ISFG	– sys-FIR Internal Standard File Generator
s-VHDG	– sys-FIR VHDL Generator
s-FCG	– sys-FIR Frontend Constraints Generator
s-TVG	– sys-FIR Test Vector Generator
TCF	– Transposed Direct Form Coefficient File
TFF	– Transposed Direct Form internal standard File

Chapter 1

Introduction

1.1 Motivation

Due to the non-ideal effects of analog circuits, such as DC offset, device non-linearities, proneness to temperature, supply drifts and phase distortions, the signal processing function for many real-time applications systems has moved away from the traditional analog hardware to digital implementation. When compared to the traditional analog techniques, digital signal processing (DSP) provides various advantages such as improved stability, lower equipment production costs, greater flexibility and higher performance. These advantages have lead to an increased use of DSP in a whole range of applications.

The objective of most wireless and DSP applications is to provide a standard specification for low complexity, low cost, low power consumption and high data rate. For example, in a wireless communication system, various modulation and demodulation schemes are employed. A generic modulator is designed to generate different modulation signals and a generic demodulator demodulates the respective modulated signals [GUN99]. The modulation and demodulation process involves a number of filters, including the pulse shaping filters and various image rejection filters. In short, digital filter is a key and ubiquitous component in any communication system. As the frequency of the signal to be processed digitally increases, the performance of the general purpose DSP devices becomes a limiting factor in system design, owing to the high power consumption and low number of operations that can be performed in a single sampling interval. A frequently adopted solution to this problem is to use application specific integrated circuit (ASIC), which can be customized for improved performance owing to the dedicated nature of the operating environment. The disadvantage of ASIC is its rigidity to follow changing standard

specifications or adaptation to peripheral changes. In between the two extremes are flexible soft IP cores synthesizable by any silicon compiler into ASIC solutions.

There are two types of digital filters – Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters. Direct and Transpose direct forms are two prominent FIR filter architectures used in the industry. FIR filters are preferred to IIR counterparts owing to various factors, the most important of which is stability. Different FIR filter structures have been explored in [HAW96]. The most important problem related to hardware design of an FIR filter is the multiplier block. A multiplier occupies enormous area and dissipates huge amount of power owing to its complexity in hardware. Research is currently going on in designing multiplierless filters. New algorithms to design and optimize multiplierless filters have been reported in many papers [ZHO95], [JUH01].

Efficient multiplierless realization of non-recursive digital filters with constant fixed point coefficients has been an area of pervasive research interest due to its widespread applications in digital signal processing, telecommunications, computer graphics and control [HAW96, DEM95]. The trend towards increasing sampling rates and high resolutions of analog to digital converters for RF, data acquisition and video signal processing applications has driven the design constraints of Finite Impulse Response (FIR) digital filters beyond the traditional consideration of reduced silicon cost. High throughput rate, low power dissipation and increased programmability are among the benchmark performance criteria. Dedicated hardware optimized for a fixed coefficient set meet the stringent goals of sampling rate and power budget at reduced flexibility and scalability. For applications that demand expandable, single-platform, multiservices flexibility, the emergence of fast reconfigurable FPGAs [HER01] such as the recent Xilinx XC6200 and Virtex FPGA family has made run-time reconfigurable ASIC a reality and choice technology for high-speed programmable FIR filters. Both IC design approaches require the automation techniques to perform combinatorial optimization of formidable complexity with drastically different emphasis on power, delay and area cost than software compilers for general purpose DSPs. The ongoing trend in digital signal processing is putting far greater stress on development of more powerful algorithms in order to deliver the desired performance.

While this has effectively resulted in a lot of research to optimize the filter coefficient data to improvise the speed and power efficiency of the filter, it was observed that researchers have to spend a lot of time hard coding the filter structures based on their algorithm to measure their performance with respect to the industry and academic benchmarks. This work demonstrates a new integrated platform which can be used to generate VHDL codes for FIR filters in both direct and transposed direct form. The software makes use of various optimization techniques like ‘hybrid column compression’, ‘merged arithmetic architecture’ and ‘common subexpression elimination’ to optimize the filter structures for the best possible hardware solution. It is therefore very amenable to efficient compilation of silicon solutions and forms the basis for exploitation by high-level synthesis tools targeted at the next-generation run-time reconfigurable ASICs.

1.2 Objectives

This objective of this work is to shed some light on a new design methodology for automating the development of optimized ASIC cores for the implementation of a specific classes of FIR filters – direct form and multiplierless transpose direct form FIR filters with constant coefficients in Canonical Signed Digit (CSD) representation. A significant part of this project is geared towards developing an electronic design automation (EDA) tool to automate the generation of synthesizable hardware description codes for various optimization options of multiplierless transpose direct form FIR filters. The codes generated can be synthesized into the targeted hardware platform using industry standard EDA tools. The ease of synthesis and the support for scalability of this tool will enable a more rigorous evaluation of the performances of FIR filters. This fully automated and algorithm independent platform helps to expedite the process of filter implementation irrespective of its size and complexity and is meant to save precious time for researchers to pace up their design and analysis cycles. Being a unified framework of FIR filter architectural solutions, the generated RTL codes in both direct and transposed direct form can be subjected through an ASIC flow using the same standard cell library to benchmark the physical performance of various common subexpression elimination algorithms. The evaluation and analysis of the VLSI area, delay and power provides a good insight

into the unexpected shortcomings of these algorithms. The tool was designed keeping in mind the following important factors:

- User-friendliness, for an “easy to use” experience.
- Parameterization, to give the user the flexibility to modify the filter structures to suit his design needs.
- Robustness, to ensure the tool can work on a broad range of filter sets.
- Verification, to ensure the functional correctness of the output

It is hoped that based on the performance results of the new tool, industry designers as well as researchers would be able to use it to their advantage and will be able to select the most suitable filter implementations for their designs in a very short time.

1.3 Major Contributions

- (1) A novel EDA environment named **sys-FIR** was developed to create synthesizable VHDL codes which describe the hardware of direct form and transpose direct form FIR filters. Given the filter specifications in the form of filter coefficients, the codes that have been developed in MATLAB create structural VHDL codes which can be synthesized using any of the standard EDA tools like Synopsys Design Compiler, Project Navigator etc.
- (2) sys-FIR allows the user to vary the instances and some parameters of the benchmark filters as well as to generate the solutions of the same filter coefficient sets using different available optimization algorithms. sys-FIR on one hand empowers the designers to be flexible in terms of design requirements and on the other hand saves them a lot of precious time and resources. Various filter structures were implemented to check the robustness of the tool. An unbiased and fair environment was incorporated in sys-FIR to conduct the functional testing of the codes after the HDL generation. An inbuilt random input generator was incorporated to develop input vectors for dynamic power analysis as well as robust RTL verification.

- (3) A benchmark database was compiled from many FIR filters cited in the literatures and simulated using the sys-FIR-Modelsim verification and simulation utility. The information gathered covers filter coefficients in different number systems such as Canonical Signed Digit (CSD), frequency response, gain/attenuation, input word length and special applications of the filter, if any.
- (4) A 64-tap digital fixed filter, used in digital HDTV applications was implemented using sys-FIR in a Standard Cell based ASIC design. It can operate in frequencies over 100MHz. The design was optimised for efficient power distribution, minimum placement congestion and zero routing violations.

1.4 Organization of the Thesis

This thesis is arranged into five chapters.

In Chapter 1, the motivation for the work and the objectives together with the major contributions of the work are presented.

In Chapter 2, the basic filter fundamentals are covered. A literature review on Finite Impulse Response digital filters, which includes both Direct as well as Transposed Direct Form filters, is included. The various computational units in filters, such as binary adders, Ripple Carry Adders (RCA), Carry Look-Ahead Adders (CLA) and Carry Save Adders (CSA) are reviewed next. Common Subexpression Elimination (CSE), a design methodology which helps in reduction of redundancy in filter implementation is explained next. The various available CSE algorithms like BHA, BHM, CRAs, ITM, NRSCSE etc. are reviewed.

In Chapter 3, newly developed platform – **sys-FIR** for the generation of digital FIR filters is presented. The development phases of this software are discussed next, followed by the details of the tool and the various features that it incorporates.

In Chapter 4, the focus is on the design flow used for the FIR filter design which encapsulates sys-FIR as a key component. A practical benchmark filter, which

has been synthesized by this new flow in direct form, is discussed next. The VLSI performance of a whole range of filters, which have been designed using sys-FIR in transposed direct form using different CSE algorithms, is analyzed. The main performances metrics analyzed include area, power and delay. The varying impacts of different CSE algorithms on a filter structure are also investigated and discussed.

Chapter 5 concludes the work and suggests potential area for future work.

Chapter 2

Background and Existing Works

This chapter discusses the various research topics related to the foundation of this work. Linear-phase FIR filter theory is briefly summarized in Section 2.1 followed by the two different FIR filter implementation methodologies – direct and transposed direct form. The two forms of signed digit representations are then discussed and differentiated. Since adders are the most basic design entities used in the implementation of multiplierless filters, Section 2.3 focuses on different standard adder structures. Theory on correction vectors, merged arithmetic architecture and common subexpression sharing is presented next. The use of merged arithmetic architecture and common subexpression elimination algorithms has brought novelty as well as efficiency in the performance of sys-FIR.

2.1 Linear-phase FIR Filter

The frequency response of a linear-phase FIR filter [LYO04] can be described by a real-valued zero-phase frequency response, $H_R(\omega)$ function and a real-valued phase function, $\theta(\omega)$, as in Eq. (2.1).

$$H(\omega) = H_R(\omega) e^{j\theta(\omega)} \quad (2.1)$$

Since $|e^{j\theta(\omega)}| = 1$, $|H(\omega)| = |H_R(\omega)|$. Ignoring the linear phase term, $e^{-j\omega(N-1)/2}$, and N th order linear phase FIR filter can be written as Eq. (2.2).

$$H_R(\omega) = a_0 + 2 \sum_{m=1}^M a_m T_m(\omega) \quad (2.2)$$

where a_m is a function of the filter coefficient h_m and $T_m(\omega)$ is a trigonometric function.

For symmetric impulse response, if N is odd, $M = (N-1)/2$. $a_0 = h_M$ and $a_n = h_{M-n} = -h_{M+n}$ and $T_m(\omega) = \cos(\omega m)$. If N is even, $M = N/2$. $a_0 = 0$ and $a_n = h_{M-n} = h_{M+n}$ and $T_m(\omega) = \cos(\omega(m-1/2))$. If the filter response is anti-symmetrical, $T_m(\omega)$ is a sine function instead of the cosine function and $a_0 = 0$.

Traditional N -tap FIR filter with the impulse response coefficients, h_k can be implemented in either the direct form or transpose direct form. The mathematical expression for a direct form filter is:

$$y_n = \sum_{k=0}^{N-1} h_k x_{n-k} \quad (2.3)$$

where y , x , h and N are output, input, filter coefficient and filter length. n is infinite.

This means that, the output y_n is not valid until the N^{th} tap is filled up.

The transpose direct form is also mathematically equivalent to Eq. 2.3. However, the hardware representation is different.

Direct mapping of the transfer function in Eq. 2.3 gives direct form FIR structure, as shown in Fig. 2.1. In this structure, each of the delayed versions of input signal is multiplied with its corresponding filter coefficient, and the output of the filter is the summation of the results of N multiplications. Generally, for an N -tap direct form FIR structure, it requires $(N-1)$ additions and N multiplications. As shown in Fig. 2.1, the number of multiplications and additions in the structure is proportional to the number of filter taps, which leads to excessive area and power consumption. In [HAW96], it is mentioned that a direct form structure with a tree of adders (merging of the $N-1$ additions) gives superior performance, compared to a direct form structure without using adder tree, owing to the logarithmically increasing delay with increasing filter length.

Fig. 2.2 demonstrates the transpose direct form FIR filter structure. The main difference between the two forms is the tap delay line. In transpose direct form, unlike direct form, input is fed to all the taps simultaneously and the delayed output from the previous tap is added to the signal from the multiplier block in each tap. It is evident that the critical path in the transpose direct form depends on the multiplier block of individual taps and final adder in each tap. The multiplier block consists of

multiplications of the input signals with the coefficients, which are constants. Therefore, this block is called the Multiple Constant Multiplication (MCM) block. As the multiplications in the MCM block involve constants, optimization can be performed to reduce its hardware size. This optimization is regarded as the MCM problem [POT96].

Fig. 2.1 Direct Form FIR Filter

Fig. 2.2 Transpose Direct Form FIR Filter

2.2 Signed Digit Representations

The Canonical Signed Digit (CSD) and Minimal Signed Digit (MSD) are two commonly used number representations in digital filter coefficient synthesis [PAR01]. Both systems are radix-2 signed digit system with the digit set $\{1 \ 0 \ \underline{1}\}$ where $\{\underline{1}\}$ denotes the value -1 . They have the same property that for a given constant, the CSD/MSD representation has the minimum number of non-zero digits. This property guarantees the least number of adders/subtractors used for a given constant

multiplication. It is found that, the CSD/MSD representation can reduce about 33% of non-zero digits compared with the binary representation [PAR01]. CSD representation has an additional property that no two non-zero digits are adjacent and the representation is unique for a given binary number.

The CSD representation for a given constant is unique, while the same constant may have more than one MSD representations. For example, integer 178 is represented as {0101010010} in CSD representation, while it can also be expressed as {0011010010} or {0010110010} in MSD representation. The CSD representation is a sub-set of the MSD representation. There exist algorithms for synthesizing CSD multipliers to reduce the number of carry-save adders to meet the specification of the frequency response of the filter.

A filter coefficient in CSD or MSD form is usually expressed in terms of a sum of Signed Power of Two (SPT) terms e.g. $0.5625_{10} = 0.10010000_2$ has 2 SPT terms -2^{-1} and 2^{-4} .

2.3 Adders

Adders are the most basic and yet essential components in FIR filters. The following sections briefly review the various types of adders used in filter implementation.

2.3.1 Standard Adders

Fig. 2.3 and Fig. 2.4 show the simplest adder blocks – Half Adder and Full Adder respectively. Half adder adds two input bits whereas full adder takes the input carry bit into account. $A \oplus B$ gives the sum of a half adder and carry is logically equal to AB . A full adder comprises of two full adders connected as shown in Fig. 2.5. The Boolean equations for full adder are given in Eqs. (2.4) and (2.5).

$$\text{Sum } (S) = A \oplus B \oplus C_{in} \quad (2.4)$$

$$\text{Carry out } (C_{out}) = AB + C_{in}(A \oplus B) \quad (2.5)$$

The carry path gives the critical path for maximum delay in a full adder. This effect can be observed clearly in a Ripple Carry Adder (RCA) described in the next section.

Fig. 2.3 Half Adder

Fig. 2.4 Full Adder

Fig. 2.5 Full Adder Block Diagram

2.3.2 Carry Propagation Adder (CPA)

A carry propagation adder (CPA) is used to add the final addends of the Carry-Save Adder (CSA) tree (Section 2.4). Two extreme cases of CPA are:

- (1) Maximum delay but least hardware complexity (RCA).
- (2) Minimum delay but high hardware complexity (CLA).

2.3.2.1 Ripple Carry Adder (RCA)

The simplest way to construct an n -bit adder is to cascade n 1-bit full adders. This adder is called a Ripple Carry Adder because the carry ripples through the n stages. The sum of the n^{th} bit cannot be performed until the carry C_{n-1} is evaluated. Fig. 2.6 shows a 3-bit RCA made up of 3 Full Adders (FAs).

Fig. 2.6 3-bit RCA

2.3.2.2 Carry Look-ahead Adder (CLA)

The carry look-ahead adder is a fast parallel adder because it calculates carry signals in advance, based on the input signals. It is based on the fact that a carry signal will be generated in two cases: (1) when both bits A_i and B_i are 1, or (2) when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1. Thus, one can write,

$$C_{OUT} = C_{i+1} = A_i.B_i + (A_i \oplus B_i).C_i \quad (2.6)$$

Eq. (2.6) can also be expressed as Eq. (2.7).

$$C_{i+1} = G_i + P_i.C_i \quad (2.7)$$

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i \quad (2.8)$$

where $G_i = A_i.B_i$ (Generate term) (2.9)

and $P_i = (A_i \oplus B_i)$ (Propagate term) (2.10)

Let us assume that the delay through an AND gate is one-gate delay and through an XOR gate is two-gate delays. Notice that the Propagate and Generate terms depend on the input bits only and thus will be valid after two-gate and one-gate delay, respectively. If Eq. (2.7) is used to calculate the carry signals, there is no need to wait for the carry to ripple through all the previous stages to find its proper value. Thus CLA is a faster adder compared to RCA. ELM adder [KEL92] is considered to be one of the fastest adders, which is based on the concept of CLA [NAG96].

Eqs. (2.11) to (2.14) give the Boolean equations for the carry outputs in a CLA. Eq. (2.14) requires a five-input AND gate. Most technology libraries restrict the fan-in to maximum of 4 as the delay and complexity increases with increase in fan-in. Thus two additional signals – Group Generate and Group Propagate are output from a CLA block. This is illustrated in Fig. 2.7.

$$C_1 = G_0 + P_0.C_0 \quad (2.11)$$

$$C_2 = G_1 + P_1.C_1 = G_1 + P_1.G_0 + P_1.P_0.C_0 \quad (2.12)$$

$$C_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.C_0 \quad (2.13)$$

$$C_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.C_0 \quad (2.14)$$

The Carry Look-ahead Adder can be broken up into two modules: (1) the Partial Full Adder, PFA, which generates S_i , P_i and G_i as defined by Eqs. (2.8), (2.9) and (2.10) above; and (2) the Carry Look-ahead Logic, which generates the carry-out bits according to Eqs. (2.11) to (2.14).

The disadvantage of the Carry Look-ahead Adder is that the carry logic gets quite complicated for more than 4 bits. For that reason, carry-look-ahead adders are usually implemented as 4-bit modules and are used in a hierarchical structure to realize adders that have multiples of 4 bits. Figure 2.7 shows the block diagram of a 16-bit CLA adder. The circuit makes use of the same CLA Logic Block as the one used in the 4-bit adder. Each 4-bit adder provides a Group Propagate and Generate signals, which is used by the CLA Logic Block. The Group Propagate P_G and Group Generate G_G of a 4-bit adder will have the expressions of Eq. (2.15) and Eq. (2.16).

$$P_G = P_3.P_2.P_1.P_0 \quad (2.15)$$

$$G_G = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 \quad (2.16)$$

The Group Propagate P_G and Generate G_G will be available after 3 and 4 gate delays, respectively (one or two additional delays than the P_i and G_i signals).

Fig. 2.7 16-bit CLA

Notice that the carry-out bit, C_{i+1} , of the last stage will be available after four delays (two gate delays to calculate the Propagate signal and two delays as a result of the AND and OR gate). The Sum bit will thus be available after two additional gate delays (due to the XOR gate) or a total of six gate delays after the input signals A_i and

B_i have been applied. The advantage is that these delays will be the same, independent of the number of bits one needs to add, in contrast to the RCA.

2.4 Carry-Save Adder Tree

The multiplier block and the linear accumulation in the direct form FIR filter can be merged readily [ZHI03]. The result is a large irregular adder tree. These tree structures can also be formed during transposed direct form implementation owing to filter coefficients with high non-zero terms. In direct form, this large adder tree can be column compressed [ZHO95] by cost effective carry-save adders to flatten the entire filter structure to a CSA tree of full adder (FA) and half adder (HA) cells with only one CPA. Similar approach can be used in the MCM block implementation in transposed direct form implementation. The CSA tree reduction techniques and their implementation using sys-FIR are explained in Section 2.6 and 3.5.3.8 respectively.

2.5 Correction Vector

If the arithmetic involves signed integers or when subtraction is needed, two factors must be considered, the use of the 2's complement representation and the sign extension. This is not necessary if unsigned integers are used for calculations. It is assumed that the input signal to the filter is in the 2's complement form. The problem involved in dealing with signed integers is shown in Fig. 2.8. This problem is tackled by sign-extending all the signed numbers so that every number is right aligned. This is shown in the example given in Fig. 2.8 with the italicized '1'.

			Expected
+ 1 0 1 → -3			-3
+ 1 0 0 1 → -7			-7
1 1 1 0 → -2			-10
Incorrect Calculation with Signed Integers			
+ 1 1 0 1 → -3			-3
+ 1 0 0 1 → -7			-7
1 0 1 0 0 → -10			-10
In ect Calculation with Signed Integers and Sign Extension			

Fig. 2.8 Signed Integers Arithmetic

Correction vector caters to the need for sign extension. As an SPT symbolizes right shift, the input signal is shifted by different amounts in the filter. Thus sign extension plays a vital role in the accurate implementation of filter architecture. This is important in the multiplier block that employs a CSA tree. With a correction vector an adder tree will be capable of handling signed integers in 2's complement term with very little additional hardware cost. The technique involved in the implementation of the hardware required to calculate the correction vector has been explained below.

If there are numbers shifted by different amount in the CSA tree, the MSB of each shifted number must be duplicated so that all numbers are left aligned. The MSB can either be a '0' or '1', which depends on the input vector. However, as shown in Fig. 2.9, many dots will be added to the dot matrix and many extra adders will be needed just for the sign extension.

4bit					1	0	1	1
8bit	1	1	1	1	1	0	1	1

Fig. 2.9 Sign Extension

To avoid this large and unnecessary overhead, the MSB is inverted and 1 is added to the MSB i.e. $(1-x_{k-1} = \text{not } x_{k-1})$. The positions to the left of the MSB are filled up with ones. If the result is computed, the same answer is obtained as if the sign is duplicated, as can be seen in Fig. 2.10.

Copy the MSB	x_{-4}	x_{-4}	x_{-4}	x_{-4}	x_{-4}	x_{-5}	x_{-6}	x_{-7}
Invert the MSB					$\neg x_{-4}$	x_{-5}	x_{-6}	x_{-7}
Add a constant	1	1	1	1	1			
Result	x_{-4}	x_{-4}	x_{-4}	x_{-4}	x_{-4}	x_{-5}	x_{-6}	x_{-7}

Fig. 2.10 Alternative to Sign Extension

In order to fill up multiple rows of signed integer, the ones can be added up in advance and the MSB can be inverted without sign extension. Thus, a constant value which depends on the shifts can be precalculated and added at the end of first tap. This constant value will have either a '0' or '1' and thus simplified constant adders with one of the input pins tied to a '0' or a '1' can be used to implement the addition of correction vector [MAY].

2.6 Merged Arithmetic Architecture

As described in Section 2.1, an N -tap filter can be realized into a direct or a transposed direct form. Each coefficient is mapped to a sum of Signed Power of Two (SPT) terms. The accumulation is performed by a carry-save adder (CSA) tree as suggested by Reutz [REU89] to reduce the number of carry-propagate adder (CPA) to one. The use of SPT terms reduces the complexity of the general-purpose multiplier per tap to that of a simpler and faster constant multiplication. Nevertheless, the small number of SPT terms per filter coefficient implies the necessary use of one or several CPAs for the addition of two operands. CPA is slow and it contributes approximately 25%-35% to the critical path delay of a multiplier [KOL97]. The constant coefficient multipliers can be decomposed to make the merging of the multiplication and final accumulation processes possible. Using the column-compression technique in tree-structured multiplier, as described below, the entire FIR filter structure can be flattened to N - registers and a CSA tree with only one CPA.

The FIR filter given in [CHE99] is used as an example to illustrate the merging process. In the paper, Chen and Wilson developed a 15-tap halfband filter with normalized passband and stopband edge frequencies at 0.1 and 0.4, respectively. The wordlength of the coefficients is 14 bits and no more than 3 SPT terms per

coefficient are chosen to represent the filter coefficients. The SPT terms of the filter coefficients are summarized as follows.

$$h(0) = h(14) = -2^{-9}$$

$$h(1) = h(13) = 0$$

$$h(2) = h(12) = 2^{-6} - 2^{-9} - 2^{-14}$$

$$h(3) = h(11) = 0$$

$$h(4) = h(10) = -2^{-4} + 2^{-7} - 2^{-14}$$

$$h(5) = h(9) = 0$$

$$h(6) = h(8) = 2^{-2} - 2^{-8} + 2^{-13}$$

$$h(7) = 2^{-1} - 2^{-3} + 2^{-5}$$

With the filter coefficients represented in the above form, the multiplierless multiplication can be achieved by shifting (rewiring) the delayed version of the input signals to the right by the numbers of bits dictated by the powers of the SPT terms. Using the dot notation where each dot represents a binary variable of increasing significance with its position from the right, the adder structure obtained by merging the multiplication with the final accumulation is derived in Fig. 2.11 (assuming an input signal length of 6 bits for brevity).

Fig. 2.11 Derivation of the merged adder structure

The example shows that the hardware complexity of the merged operation block is only a small fraction of the complexity of the filter structure realized by using (n) general multipliers and $(n-1)$ adders. The full-adder based tree-structured multiplier creates a greater opportunity for optimization by silicon compiler when integrating long (high order) filter into a single chip from the perspective of power/delay/area efficiency measures for the targeted technology of implementation. This CSA tree is to be compressed to a two-operand addition. The maximum number of stages needed to arrive at this last stage can be found by solving the recurrence function [PAR00] shown below:

$$n(h) = \lceil 3n(h-1)/2 \rceil \quad (2.17)$$

where $n(h)$ denotes the maximum height of the adder tree and h denotes the maximum number of stages needed.

The numerical value of $n(h)$ for each h is summarized in Table 2.1.

Table 2.1 Maximum tree height $n(h)$ for a h -level Carry Save Adder tree

h	$n(h)$	h	$n(h)$	h	$n(h)$
0	2	7	28	14	474
1	3	8	42	15	711
2	4	9	63	16	1066
3	6	10	94	17	1599
4	9	11	141	18	2398
5	13	12	211	19	3597
6	19	13	316	20	5395

From Table 2.1, it is estimated that the number of stages needed to compress the merged structure is five. There are several ways for the adder-tree compression process, which differ in the allocation of FAs and HAs. Two of these tree structures have been explained in the next section.

2.7 Reduction Trees

A tree is needed to merge the inputs together. Two commonly used reduction trees - Wallace Tree and Dadda Tree are described below.

2.7.1 Wallace Tree

In Wallace tree, the number of operands is reduced at the earliest opportunity [PAR00]. That means, if there are m dots in a column, Full Adders are used immediately for that column. This tends to minimize the overall delay by making the final CPA as short as possible.

The stages of the adder reduction for the FIR merged arithmetic structure using the Wallace strategy is shown in Fig. 2.12. Here each * on the left figure denotes an input bit and each number in the right figure represents the number of stages needed to generate that specific bit. Each FA reduces three equally weighted input bits (in the same column) to one higher order carry bit and one sum bit for the next stage. Several FAs can be allocated to each column and all full adders of one

stage execute in parallel. Therefore, the latency for every stage is identical and equals to the delay of a full adder.

Fig. 2.12 CSA Tree Reduction using Wallace's Strategy

As shown in the Fig 2.12, five stages are needed to compress the adder tree to the final-stage of two operands. A final CPA is required for the two-operand addition.

2.7.2 Dadda Tree

In Dadda trees, the number of operands to the next lower number in Table 2.1 is reduced using fewest FAs and HAs possible [DAD65] i.e. Dadda tree makes use of the “As late as possible” strategy for adder-tree reduction. The big advantage of the Dadda tree is that it needs fewer adders. Unfortunately the final CPA gets very long.

Fig. 2.13 shows the difference between the Wallace Tree and the Dadda Tree reduction process and the corresponding total hardware cost involved in both cases, where each dot denotes one bit.

Fig. 2.13(a) Adding 6-bit numbers using Wallace Tree

Fig. 2.13(b) Adding 6-bit numbers using Dadda Tree

In this particular example, for an addition of 6 six-bit numbers both Wallace tree and Dadda tree structures consume a total of 28 FAs and 1 HA.

2.8 Common Subexpression Sharing

As shown by Hartley in his work [HAR96], in cases where several multipliers are present in the network of operators for instance in the MCM block of an FIR filter, the savings achieved by identifying common subexpressions can be as much as 50% of the total number of operators. Any 2 coefficients can be found to have common subexpressions as shown in example of Fig 2.14.

Fig. 2.14 Example Illustrating Subexpression Sharing

Subexpression sharing has been used for purposes such as critical path reduction [IQB93] as well as circuit minimization [HAR91]. Hartley gives an example of a multiplication of a 16 bit number by the CSD value {10100100101001}. The resulting multiplier is shown in Fig. 2.15 in which circles represent adders with the indicated number of bits. The notation $\gg n$ denotes right shift of the operand by n bits. Fig. 2.16 shows the use of common subexpression sharing which reduces the number of adders from 5 in a full precision adder to 4 in a CSE based multiplier.

Fig. 2.15 Full Precision CSD Multiplier

Fig. 2.16 Multiplier using Common Subexpression Sharing

For this work, CSE has been applied to transposed direct form implementation only. The following sections describe briefly the various algorithms currently being used by researchers for achieving the goal of generating hardware and speed efficient filters.

2.8.1 BIN

BIN stands for binary algorithm. It is the most basic representation of the filter coefficients. It simply transforms the decimal coefficients to binary coefficients without detecting common subexpressions for elimination. This performance of this baseline algorithm acts as the reference to benchmark the improvement gained by other algorithms for the MCM block.

2.8.2 CSD

CSD represents coefficients in Canonical Signed Digit form ([JOC01]). This algorithm is similar to BIN in the sense that it is also a baseline algorithm without any specific effort to maximize sharing of common subexpressions. CSD is a special ternary number system where any integer is represented with a string of 0, 1 and -1 digits and no two non-zero digits are adjacent to each other. It is a minimal and canonical signed digit representation and hence is commonly used as the starting point for many algorithms.

2.8.3 BHM

There are several algorithms based on graph data structure [BUL91, DEM95, DEM02], which aim at optimizing the filter structures by reducing the number of arithmetic operators used. Bull and Horrocks introduced the concept of primitive operators in [BUL88] and [BUL91]. The BH algorithm [BUL91] makes use of the decimal representation of coefficient set for optimizing the logic complexity of the multiplier block. Later certain modifications were made to BH algorithm. The use of subtractors to reduce the number of logic operators required and the reduction of each partial sum by factors of two until it becomes odd were the most prominent ones. The modified Bull and Horrocks algorithm with these modifications is called BHM [DEM95].

2.8.4 MRAG

MRAG stands for Modified Reduced Adder Graph 2-bit CSE algorithm [FEI05c]. This algorithm is a modified version of the RAG- n algorithm. To increase the opportunity of generating new coefficients in the optimal part, the power-of-two multiples of the fundamentals are put in the search pool besides the fundamentals. In the heuristic part, only the fundamentals are sought in the generation of distance-2 coefficients.

2.8.5 C1

C1 [DEM95] is designed to find the lowest logic depth. This algorithm uses the Reduced Adder Graph algorithm (RAG-n) or BHM algorithm (when the wordlength of the coefficients is long) to design a multiplier block. This algorithm generates a graph using all the required coefficients plus all cost-1 coefficients up to twice the value of the maximum coefficient. The logic depth of this graph is the 'target depth'. It then eliminates all cost-1 coefficients that are not used to create any of the required coefficients from this graph.

2.8.6 NRSCSE

NRSCSE stands for Non Recursive Signed Common Subexpression Elimination algorithm [BOE02]. While the recursive utilization of a common subexpression generates a high logic depth into the digital structure, the NRSCSE algorithm allows the designer to overcome this problem by using each subexpression once. The NRSCSE algorithm searches for the non-recursive signed common subexpressions that must be eliminated from the original CSD array. Compared with the most popular multiplierless algorithms, NRSCSE offers the best relation area – frequency operation, or number of adders - logical depth along with optimal runtime operation due to its simplicity.

2.8.7 Pasko

Pasko algorithm [RPD99] is yet another algorithm for solving the MCM problem. It is also based on the principle of common subexpression elimination. It combines the exhaustive search for multiple pattern identification with a steepest descent approach for pattern selection. The results show a significant reduction in either arithmetic operations or hardware necessary to implement those operations combined with satisfactory runtimes.

2.8.8 CRAH-2

CRAH-2 stands for contention resolution algorithm for 2-bit horizontal CSE [FEI05b]. The CRAH-2 algorithm manages two-bit common subexpressions and aims at achieving the minimal logic depth as the primary goal. CRAH-2 provides a leeway to break away from the local minimum and the flexibility of varying optimization options through a new data structure called the admissibility graph. Reduction in the number of logic operators is used directly as the criterion of the appraisal.

2.8.9 CRA

CRA or the Contention Resolution Algorithm is the generalization of CRAH-2 that can be used for the elimination of higher weight common subexpressions as well as horizontal, vertical and oblique common subexpressions of any weight [FEI05a]. While the existing CSE algorithms use horizontal common subexpressions predominantly due to the symmetry of linear phase FIR filters, CRA exploits both the intra- and inter-coefficient redundancy to reduce the logic depth and logic complexity further. The ability to handle subexpression elimination in both dimensions is unique to this algorithm.

2.8.10 MBPT

MBPT stands for Multirooted Binary Partition Tree algorithm [JCV06]. The adder tree for the Multiple Constant Multiplication (MCM) block of an N -tap digital filter is modeled as a directed acyclic graph called the Multirooted Binary Partition Tree (MBPT). This visually appealing data structure provides an opportunity to dynamically allocating subexpressions at bit level. Based on the direct correlation between MBPT and the circuit topology, a maximum likelihood disjunctive decomposition algorithm is proposed to synthesize a reduced MBPT for the design of low complexity FIR filters. The proposed method can be viewed as an efficient divide-and-conquer approach with dynamic tracking of the statistic of weight-two

subexpressions. As isomorphic subgraphs represent common subexpressions, both weight two are eliminated implicitly in the graph synthesis process.

2.8.11 BCSE

BCSE stands for Binary Common Subexpression Elimination algorithm [MAV06]. This algorithm optimizes the multiplier block of a FIR filter based on the binary form of its coefficients. It eliminates only three different types of common subexpressions. It is reported in [JOC01] that the elimination of subexpressions in the binary form of the coefficients is better than in the canonical signed digit (CSD) form of the coefficients.

2.8.12 4BCSE

4BCSE stands for 4-bit Binary Common Subexpression Elimination algorithm [SMI06]. The functionality of this algorithm is similar to that of BCSE except for the fact that it eliminates three more types of subexpressions than BCSE.

2.8.13 HARTLEY

In [HAR96] the author shows that if the multiplier is represented in Canonical Signed Digit (CSD) form, the number of additions (or subtractions) used will be a minimum. In the case where several multipliers are present in a network of operators, for instance in an FIR filter, the savings achieved by identifying common subexpressions can be as much as 50% of the total number of operators. The asymptotic frequency of the most common subexpression was analyzed mathematically, and it was shown that sharing the two most commonly encountered subexpressions can be expected to lead to a 33% saving of the number of additions.

This chapter briefly covered Linear-phase FIR Filter theory. It also described the 2 FIR filter implementation styles – Direct Form and Transposed Direct Form. Filter optimization techniques like Merged Arithmetic Architecture and Common

Subexpression Elimination were also discussed. The next Chapter focuses on the integrated platform –sys-FIR, its structure and features.

Chapter 3

Development of sys-FIR

The recent research in digital filters has lead to the development of a whole range of common subexpression elimination algorithms, which remove redundancy from filter structures. Each algorithm has some unique features that make it suitable for a certain filter application. The problem now lies with the time required to verify and validate these algorithms. Moreover, it is becoming increasingly difficult to provide a standardized solution to all the filter structures. So there lies a need to develop a universal platform, which can generate synthesizable filter codes based on these algorithms to provide fast analysis on the suitability of an algorithm. sys-FIR, an integrated platform for the generation and verification of digital FIR filters in both direct and transposed direct form, is a step in this direction. This chapter provides the complete details of sys-FIR, including an overview of the system, its input and output formats, modules, features and functionalities.

3.1 sys-FIR Overview

Fig. 3.1 provides an overview of the integrated platform, sys-FIR. The inputs to sys-FIR constitute the filter coefficient file, CSE algorithm (if transposed direct form is chosen), CPA adder choice and input signal width. The whole process has been divided into two steps:

a) Generation of sys-FIR Internal Standard File: The Filter Coefficient File (FCF) is the primary input to the system. FCF is then converted into either ‘sys-FIR Direct Form Internal Standard File (DFF)’ or ‘sys-FIR transposed Direct Form Internal Standard File (TFF)’ format by an internal standard file generator (s-ISFG), based on the implementation style chosen by the user. These file formats are explained in Section 3.2. The generation of TFF has an intermediate step that involves the selection of one of the 13 Common Subexpression Elimination algorithms which

have been described in Section 2.8. This decision is based on the user defined input – ‘*CSE Algorithm*’.

b) Generation of VHDL Codes: Once the internal standard file has been created, the next step is to use it to generate the synthesizable VHDL codes. The filters created by any of the CSE algorithms have fixed architecture. Sys-FIR will generate structural VHDL code to describe the corresponding architecture without any modification. The sys-FIR VHDL generator or s-VHDG utility performs this task. The generation of the VHDL files is also dependent on the type of adder selected by the user and stored in the variable – ‘*Adder Selection*’. The available options for this variable have been described in Section 3.5.1.1. The ‘*Verification*’ variable controls the VHDL verification stage. The sys-FIR Test Vector Generator or s-TVG utility generates a random set of test vectors which are used by the simulator ModelSim [MEN06a] to run functional verification on the filter VHDL codes. The number of such test vectors is again controlled by the user. The verification process has been explained in Section 3.5.3.3.

Fig 3.1 sys-FIR Overview

3.2 Inputs to sys-FIR

The filter coefficient file (FCF) provided to sys-FIR is a simple text file which contains the coefficient values in either CSD or Binary format. A sample file containing the coefficients of a 26-tap filter is shown in Fig. 3.2.

Sample Filter Coefficient File (FCF)														
0	0	0	0	0	0	0	0	0	0	0	0	-1	0	
0	0	0	0	0	0	0	0	1	0	-1	0	0	0	
0	0	0	0	0	0	0	0	0	1	0	1	0	0	
0	0	0	0	0	0	0	0	-1	0	0	1	0	0	
0	0	0	0	0	0	0	-1	0	0	0	0	-1	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	1	0	0	0	0	0	-1	0	-1	
0	0	0	0	0	1	0	-1	0	1	0	-1	0		
0	0	0	0	0	-1	0	0	-1	0	1	0	0		
0	0	0	0	-1	0	-1	0	0	0	1	0	-1		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	1	0	-1	0	1	0	-1	0	1	0	1		
0	1	0	-1	0	0	0	0	1	0	0	0	1		
0	1	0	-1	0	0	0	0	1	0	0	0	1		
0	0	1	0	-1	0	1	0	-1	0	1	0	1		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	-1	0	-1	0	0	0	1	0	-1		
0	0	0	0	0	-1	0	0	-1	0	1	0	0		
0	0	0	0	0	1	0	-1	0	1	0	-1	0		
0	0	0	0	0	0	0	0	0	0	-1	0	-1		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	-1	0	0	0	0	-1	0		
0	0	0	0	0	0	0	0	-1	0	0	1	0	0	
0	0	0	0	0	0	0	0	0	1	0	1	0	0	
0	0	0	0	0	0	0	0	1	0	-1	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	-1	0		

Fig. 3.2 26-tap Filter Coefficient File

Once the FCF is input into sys-FIR, it is converted into two types of internal standard file formats based on the user specified preference in terms of implementation style: direct form or transposed direct form. It is important to have standardized formats in order to avoid any compatibility issues with the newer algorithms which will be incorporated into sys-FIR in the future. The compatibility issue is further explained in Section 3.5.3.2.

3.2.1 Direct Form

An example of a 15-tap sys-FIR Direct Form Internal Standard Coefficient File (DFF) has been shown in Fig. 3.3. Information about each non-zero term is present in a separate row in the file, which has three columns. The first column gives the tap number. The second column contains the horizontal position (index) of the non-zero bits in the CSD matrix. The sign of the non-zero bits is stored in the third

column which is either '1' or '-1'. In case of symmetrical filters, the symmetric taps are placed together in DFF e.g. In Fig 3.3, Tap 1 is followed by the symmetric Tap 15. This ensures that sys-FIR recognizes this filter as a symmetric one while parsing its DFF.

sys-FIR Direct Form Internal Standard Coefficient File (DFF)		
Tap No.	Position	Sign
1	9	-1
15	9	-1
3	6	1
13	6	1
3	9	-1
13	9	-1
3	14	-1
13	14	-1
5	4	-1
11	4	-1
5	7	1
11	7	1
5	14	-1
11	14	-1
7	2	1
9	2	1
7	8	-1
9	8	-1
7	13	1
9	13	1
8	1	1
8	3	-1
8	5	1

Fig. 3.3 Sample Direct Form internal standard File (DFF)

The coefficients can be easily reconstructed from the file shown in Fig. 3.3. Taking the example of Tap 8, there are 3 nonzero digits in the coefficient for this tap. The first one is present at position one followed by the next one at position three and the final one at position five. The number of zeroes to the right of the final non-zero digit depends on the maximum shift of the coefficients. In this example it would be 14. So Tap 8 would be {1 0 1 0 1 0 0 0 0 0 0 0 0 0}.

3.2.2 Transposed Direct Form

One of the primary requirements of sys-FIR is to be able to adapt to new CSE algorithms. A major obstacle in this regard is that most of the algorithms are being designed by geographically distant researchers. It is difficult to request all the researchers to generate the outputs in a standard format. Fig. 3.4(a) and Fig 3.4(b) show the outputs of two CSE algorithms. Fig. 3.4(a) shows the output of CRAH2 [FEI04] and Fig. 3.4(b) presents the output of CRA [FEI05]. It is clear that the outputs generated by the two algorithms are not exactly the same even for the same filter coefficients.

CRA Output
0 0 0 1 0 1 0 0 -1
0 0 0 -1 0 0 0 1 0
0 1 0 1 0 0 -1 0 0
0 0 1 0 0 0 0 0 0
Tap1
1 2 0 0
1 1 3 3
3 2 0 0
Tap2
0 0 0 0
0 0 0 0
0 0 0 0
Tap 3
1 2 0 0
1 1 3 3
3 2 0 0
Tap4
0 0 0 0
0 0 0 0
0 0 0 0

Fig. 3.4 (a) CRA output

CRAH2 output
0 0 0 1 0 1 0 0 -1
0 0 0 -1 0 0 0 1 0
0 1 0 1 0 0 -1 0 0
0 0 1 0 0 0 0 0 0
Tap1
1 2
1 1
0 0
Tap2
0 0
0 0
0 0
Tap 3
1 2
1 1
0 0
Tap4
0 0
0 0
0 0

Fig. 3.4 (b) CRAH2 output

Hence it became necessary to develop an algorithm independent sys-FIR internal file format for the transposed direct form implementation to tackle the following two issues:

1. Compatibility with existing and upcoming CSE algorithms.

2. Avoiding manual intervention for modifying input files leading to unnecessary typo errors and delays.

A typical sys-FIR Transposed Form Internal Standard Coefficient File (TFF) has been shown in Fig. 3.5. The first line of the file identifies the CSE algorithm used for the generation of this file. A '#' sign is used to indicate a comment. Anything followed by '#' is ignored by sys-FIR. Each TFF is divided into three sections which describe the filter topology according to the solution of the CSE algorithm. Each section name begins with an '@' sign.

Fig. 3.5 Sample Transposed Direct Form internal standard File (TFF)

Section 1: This section of the file describes the most elementary subexpressions present in the coefficient set. These are elementary in the sense that they would be later used to create other subexpressions and the coefficients. These subexpressions can be of weight 2 or higher. As shown in Fig. 3.5, Section 1 stores both the binary and integer values of the subexpressions in a row. Some CSE algorithms like CRA can eliminate oblique or vertical (V/O) subexpressions, as explained in Section 2.8.9 and Section 3.5.1.2. The syntax for vertical/oblique subexpressions remains the same as that for the horizontal subexpressions except that it spans as many rows as the height of the vertical or oblique common subexpression, and the first column of these rows is not designated for indicating the integer value of these subexpressions. Instead, an even number is placed in the first column of the first row to identify an inter-coefficient common subexpression, followed by zeros in the first column of subsequent rows until the height of this common subexpression is reached. A unique inter-coefficient common subexpression identifier is assigned to each vertical/oblique common subexpression in ascending order starting from the number 2. The following example further clarifies the concept of vertical/oblique subexpression notation in the input filter specification file, assuming a coefficient set as denoted in Fig. 3.6.

0	0	0	1	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	-1	0	0	-1	0	0	0

Fig. 3.6 Sample Coefficient Set with V/O Subexpression

In Fig. 3.6, the non-zero digits in the shaded boxes constitute one subexpression and the boxes with darkened border constitute the second subexpression. The two non-horizontal subexpressions in this example are explained below. The square brackets below denote the tap shift.

$$\text{a) } 2 \equiv 1 - 1[-3] \ll 2 \qquad \text{b) } 4 \equiv 1 \ll 1 - 1[-1]$$

The corresponding notation in TFF is shown in Fig.3.7.

Vertical/Oblique Subexpressions			
2	0	0	1
0	0	0	0
0	0	0	0
0	-1	0	0
4	1	0	
0	0	-1	

Fig 3.7 V/O Subexpression Declaration in TFF

The first common subexpression is identified by a 2 whereas the second subexpression is identified by a 4. The number of zeros in the first column in the following lines is equal to the number of register delays to be inserted e.g. [-3] denotes a delay of 3 clock cycles between the 2 non-zero terms of the subexpression “2” shown above in (a).

Section 2: The second section is used for higher level subexpressions which can be generated by performing simple add and shift operations on the subexpressions present in Section 1 along with the isolated ‘1’s and ‘-1’s. When the higher weight common subexpression consists of a vertical or an oblique common subexpression, the subexpression name is appended by an ‘i’, e.g., $8 \equiv 4i \ll 3 + 1 \ll 1$. The identifier of the horizontal higher weight common subexpressions is the first column number and is equivalent to its integer value. It is also possible that there might be no higher level subexpressions at all. As shown in Fig. 3.5, in this particular scenario, the section declaration is followed by a blank line.

Section 3: The last section contains all the coefficient values which also include the non-zero non-subexpression digits. The first number of each row is the integer value with the sign of the coefficient. The other numbers of the row are arranged in pairs that describe how to build the coefficient. E.g. in the sample filter of Fig. 3.5, the coefficient of the 5th tap is equal to 2040 i.e. $1 \ll 11 + -1 \ll 3$.

Even if any of the three sections are empty it is mandatory to keep the section header likes “@1. Section” even if no subexpression is specified in Section 1.

3.3 Common Subexpression Elimination in sys-FIR

The theory on common subexpression sharing is explained in Chapter 2, in which it is shown that the coefficients of two taps could have similar subexpressions. In such cases, hardware can be shared between two taps. Consider the example from Chapter 2, which is shown here in Fig. 3.8.

Fig. 3.8 Subexpression Sharing

The subexpressions '1 0 1' and '1 0 0 1' are common to both taps m and n . Note that in a transposed direct form filter the same input signal enters all the taps simultaneously. Thus, both taps, m and n will add the same input signals but shifted by different amounts. Consider the bit string '1 0 1'. In tap m , the output of the adder implementing the subexpression '1 0 1' is shifted by 10 bits to the right but in tap n , it is shifted by 2 bits. The adder for each common subexpression can be implemented by using either a CLA or an RCA.

This elimination of subexpressions has been of great interest to researchers because of the potential savings made by these eliminations in terms of area, speed and power costs. Following is the list of 13 well-known Common Subexpression Elimination (CSE) algorithms which are described in Section 2.8 and are supported by sys-FIR.

- a) BIN
- b) CSD
- c) BHM
- d) MRAG
- e) C1
- f) NRSCSE
- g) PASKO

- h) CRAH2
- i) CRA
- j) MBPT
- k) BCSE
- l) 4BCSE
- m) HARTLEY

3.4 sys-FIR Planning and Design Considerations

The development of sys-FIR was preceded by a thorough understanding of the potential design and implementation issues. In this section, some of those planning issues are discussed.

3.4.1 Symmetric/Asymmetric Implementation

Fig. 3.9 and Fig. 3.10 illustrate two possible transposed direct form filter structures depending on the filter specifications. It was observed that huge hardware saving can be made by taking advantage of the symmetric nature of filters. The implementation of asymmetric filters is straightforward compared to symmetric filters. An asymmetric filter design might consist of a CSA tree followed by a CPA in each tap or an entity with recursive CPA adders. The hardware design of symmetric filters is explained here. Referring to Fig. 3.10, consider a 6-tap filter with the coefficients of Tap 6 identical to Tap 1, Tap 5 identical to Tap 2 and Tap 4 identical to Tap 3. Thus both Tap 6 and Tap 1 should have the same hardware as far as the SPTs are concerned. Tap 1 differs from Tap 6 in the previous tap CPA output. Thus, taps from 6 to 4 will have a CSA tree followed by two CPAs. The first CPA adds the carry and sum outputs from the CSA tree. The second CPA adds the result from the CSA tree and previous tap's output. The output from the first CPA is carried to the symmetric tap. For instance, the output for CPA1 of Tap 5 is carried to Tap 2 where it is added to the previous tap's output (Tap 3 is the previous tap for Tap 2).

Fig. 3.9 Transposed Direct Form Implementation

Fig. 3.10 6-Tap Symmetric Filter

However, it is noticed that in both asymmetrical and symmetrical filters there is a further possibility of saving hardware costs because of the presence of non-symmetric taps which have either exactly the same coefficients or coefficients that differ only by power-of-two multiples (which can be obtained simply by hardwired

shifters). Therefore instead of classifying filters on the basis of their structural symmetry, sys-FIR goes a step further and implements similar taps only once, thereby saving Silicon area.

3.4.2 Input Range Analysis

Since sys-FIR is meant to be a generic tool able to cater to all the digital FIR filter needs, it is important for sys-FIR to have the capability to allow the user to generate a filter for a certain application. Also, the input signal width is a function of the application for which the filter is being designed. This in turn becomes one of the most critical inputs to be provided by the user. Due to the computational limitation of MATLAB described below, sys-FIR could be made compatible for an input range of 2 to 128-bit signals. This wide range has been selected just to ensure that sys-FIR stays robust even when dealing with unusually large input signal lengths.

A few limitations are observed in MATLAB tool, which also put a constraint on the functionality of sys-FIR. Those limitations are listed here to ensure that the future users and developers of sys-FIR are aware of them. It is also important to mention here that sys-FIR does not make use of any special utilities provided by MATLAB. This methodology is adopted to ensure that a future adaptability into any other programming language would be possible for portability reasons.

1. Since only variables of type double are used to represent numbers in MATLAB, if a number is larger than the one that can be handled by the mantissa of a variable of type double, it could impact the accuracy of the results. The mantissa of a double variable in MATLAB is 52 bits wide, so it can store numbers up to 4,503,599,627,370,496 with full precision.

2. For input data wordlengths greater than 30 bits, the small difference between the ends of the dynamic range could not be taken into account. In this particular case the saturated values obtained by the range estimation can be used in the calculations to ensure that a filter could still be generated with reasonably truncation error so that the filtering specifications will not be severely affected.

3.4.3 Inherent Robustness

A lot of brainstorming was carried out on all the possible corner cases that were to be taken care of. It would be practically unfeasible to provide all those corner

cases. The following corner case is just one of the many that can be handled by sys-FIR with ease.

Let $\{0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\}$ be a filter coefficient and $\{*\ * \ * \ * \ * \ * \ * \}$ be a hypothetical input signal being fed into the filter, where ‘*’ denotes a bit. On multiplication, the output would be $\{0\ * \ * \ * \ * \ * \ * \ * \ 0\ 0\ 0\ * \ * \ * \ * \ * \ * \}$. Since the multiplication operation in this particular case creates a single row there is no need for a final CPA implementation. Therefore, it was necessary for sys-FIR to be intelligent enough not to use an adder for such computations since the final result can be obtained just by shift operations, which involve simple wire connections. On the other hand, a similar case involving $\{0\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 1\}$ as one of the coefficients would definitely lead to implementation of a CPA for the final addition.

3.4.4 Automatic Verification

It is obvious that even after achieving a fast implementation of filter structures for rapid prototyping, the bottleneck would shift to the verification stage. Manual intervention for design and library setup, transfer of files and invoking of tool would defeat the whole purpose of sys-FIR in terms of speeding up the process of generating synthesizable VHDL codes. Therefore, automatic verification was considered an essential sign-off step in sys-FIR and it became imperative to integrate this step into the system. This would also help in producing error free VHDL codes thereby enable early detection and elimination of any potential error before the frontend synthesis.

3.5 sys-FIR System Description

In this section, the details of sys-FIR are described. First, the filter structure, as implemented by sys-FIR, is described by giving a detailed analysis of the sys-FIR result for transposed direct form structure. This is followed by the software architecture and execution steps. The top-level graphical user interface (GUI) that gives an easy access to the tool is detailed next. The last section focuses on the various features that make sys-FIR a unique, useful and efficient tool for digital filter design.

3.5.1 Implementation

3.5.1.1 Filter Structure

As shown in Fig. 3.11, the general filter structure generated by sys-FIR is quite different from the unified transposed direct form structure shown in Figs. 3.9 and 3.10 even though the functionality remains exactly the same.

Fig. 3.11 Filter Structure implemented by sys-FIR

The filter structure can be broadly divided into 2 parts, an algorithm dependent *MCM* block and an algorithm independent *ADDNDELAY* block. The *MCM* block contains another entity called the *CSBLOCK* which implements all the common subexpressions created by the CSE algorithm. The implementation can be carried out by using either an RCA or a CLA. Fig. 3.11 shows only the CLA implementation. However, the structure remains the same when RCAs are used. The *ADDNDELAY* section is a long array of registers and adder blocks. The clear demarcation between *MCM* and *ADDNDELAY* blocks enables an easy comparison between the results generated by different CSE algorithms without worrying about the contribution of the non-combination logic (the latches) and the adders in the overall results. An *ADDNDELAY* block with CSA option is implemented as shown in Fig. 3.12.

Fig. 3.12 ADDNDELAY Block with CSA option

As can be seen in Fig. 3.12, the presence of two adjacent latches confirms that a certain tap of this filter had no non-zero term. So no multiplication has taken place for this tap and the input is simply delayed by one cycle.

3.5.1.2 Correction Vector

The theoretical details of correction vector are present in Section 2.5. Although implementation of correction vector helped in the reduction of hardware due to sign extension problem in both direct and transposed direct form implementations, a problem related to the output of an n -tap transposed direct form filter arose due to the use of correction vectors. The presence of correction vector meant the presence of vdd and gnd connections in the last tap. When an input signal was pumped into the filter hardware it was observed that the filter response was incorrect for the first $(n-1)$ clock cycles. As a result an n -tap filter actually started the correct filtering operation only after n clock cycles. This can be seen in the simulation results of a 5-tap filter shown in Fig. 3.13.

Fig. 3.13 Issue of Correction Vector Implementation

As shown in Fig. 3.13, even when an input of zeroes was latched into the filter, the filter outputs were non-zero for 4 clock cycles. Only after the fifth clock cycle, the output returned back to zero. To avoid this issue it was then decided to remove the correction vector implementation from the final tap of the filters and uses 2's complement adders/subtractors with a more precision wordlength estimation of the operators instead.

3.5.1.3 Handling Vertical/Oblique Subexpressions

As described in Section 2.8.9, CRA is capable of generating subexpressions which go beyond a particular tap's boundaries. These are the vertical and oblique common subexpressions. It has been anticipated in the development of sys-FIR that in the future many more CSE algorithms will be developed to exploit the sharing of non-horizontal subexpressions. Therefore, sys-FIR is equipped with the capability of handling such cases. As shown in Fig. 3.14, the CS Block in this particular case is redesigned as the CSVOS (Vertical/Oblique Subexpressions) block by sys-FIR. The number of latches to be inserted between two addends is derived from the input file by calculating the number of taps over which the subexpression crosses.

Fig. 3.14 CSVOS Block for Vertical/Oblique Subexpressions

3.5.2 sys-FIR User Interfaces

It has been kept in mind that users have different preference in terms of the ease of using any software or EDA tool. While some users might prefer a GUI mode others might prefer an interactive step-by-step mode in MATLAB. Therefore both options are provided for all three execution alternatives in sys-FIR, as described below.

1. Start Entire Process from Coefficient to VHDL Code: If chosen, this option will take the user from user-specified filter coefficients to the generation of synthesizable VHDL codes. This forms the top level of sys-FIR and encapsulates the entire process. The top-level GUI for this option is shown in Fig. 3.15.

Fig. 3.15 sys-FIR: Top-Level GUI

2. Start Process from Coefficient to Input File: This option helps in the generation of sys-FIR standard file forms, DFF and TFF, for direct form and transposed direct form implementation respectively. No VHDL file is generated. This option is useful if the user is interested in making a theoretical estimation of the logic depth and logic complexity in terms of the number of discrete adders, or the number of Full Adders (FAs) and Half Adders (HAs), or switching activity of the solutions generated by different algorithms. The unified sys-FIR input file makes writing such estimation program easy without having to concern about the different output formats of different algorithms. The next level GUI for this option is shown in Fig. 3.16.

Fig. 3.16 DFF/TFF Generation GUI

A drop down menu in the GUI gives the user an option to select any of the available algorithms. Apart from the regular features like directory location and the filename input, another useful feature present in the GUI is the “Plot Coefficient” utility that allows the designer to see an impulse response to the filter coefficient set. This utility is provided only in the GUI mode.

The interactive session in MATLAB helps running the system in debugging mode and the batch session helps in generating input files for a collection of filter samples.

3. Start Process from Input File to VHDL code: This option performs the generation of VHDL codes for a user-defined filter in sys-FIR standardized input files. Fig. 3.17 shows the GUI for this option.

Fig. 3.17 s-VHDG GUI

The GUI provides the following options:

1. **output dir:** This is the location where the generated VHDL files are stored.
2. **sys-FIRfile:** This is the location of the sys-FIR Internal Filter Specification file.
3. **Input wordlength:** The input signal width can be defined at this location.
4. **Adder Type for CS:** Two possible options are provided for the case of Common Subexpression (CS) Block : RCA (Ripple Carry Adder) and CLA (Carry Look Ahead Adder). Either one can be chosen based on the application requirement.

5. **Adder Type for MCM:** Again, two possible choices are provided: CLA and RCA for the multiple constant multiplication block. These options ensure that the area-timing trade off can be analyzed before hand and the appropriate adder structure can be handpicked for a certain application.
6. **Adder Type for Addndelay:** Apart from the above two options (i.e. RCA and CLA), another type of adder, Carry Save Adder (CSA), is included in this menu. This is to ensure that if a designer expects trees with high depth to be formed in a filter structure he can make use of the CSA tree structure to reduce the hardware consumption. Final Carry Propagation Adder (CPA) in this case would be a CLA.
7. **Verify Generated Code:** A simple tick in this option enables the tool to verify the VHDL codes generated. The verification environment is discussed later in Section 3.5.3.3.
8. **Random Simulation Steps:** This is another feature that has been added to help in the verification process. It generates the random input signals to be used in the simulation of the generated VHDL codes. Details would be included in Section 3.5.3.3.

3.5.3 sys-FIR Features

This section highlights the unique and useful features embedded in sys-FIR. Each feature has been designed for high value-addition to the tool.

3.5.3.1 Modularity

The generic code for all the MATLAB functions that are used to generate different VHDL entities is shown in the form of a flowchart in Fig. 3.18. Each MATLAB Code has a header section that briefs the user on the expected inputs and outputs of that program. Once the execution begins, the corresponding function checks for any inconsistencies in the inputs. In case no errors are flagged, the program computes the relevant data and stores variables like *port_name*, *component_name*, *signaldef*, *signal_route*, etc. Each VHDL code generating function returns its name and its port declaration when it is invoked. Within all sections in the file, a number of strings are generated. These strings are then passed to a MATLAB script called *vhdlFileGen.m*, which generates the VHDL file using the strings.

Fig. 3.18 MATLAB Code Structure

The code structure for VHDL generation is shown in Fig. 3.19. Various steps involved at this particular stage are articulated below.

1. **File Parsing:** Every TFF/DFP is first parsed for any unforeseen errors. This prevents the carrying forward of inadvertent bugs to the next stage. The design methodology ensures that all the major issues with the input file can be found and fixed as early as possible in the design flow.

Fig. 3.19 VHDL Generation

2. **WidthEst/ WidthEstVO:** Based on the type of subexpressions used (i.e. horizontal or vertical/oblique or both), sys-FIR calls the corresponding utilities: *widthEst* and *widthEstVO*. These functions determine the length of the output signal which is later used in the port mapping, signal definition and port declaration segments of the VHDL code. A brief explanation is given below to understand the complexity involved in adapting sys-FIR for vertical/oblique subexpressions. Assuming a 3-tap filter with coefficients equal to 5, 3 and -12, the coefficients can be described as:

$$C_0 = 5 = 1 \ll 2 + 1 \ll 0$$

$$C_1 = 3 = 1 \ll 2 + -1 \ll 0$$

$$C_2 = -12 = -1 \ll 4 + 1 \ll 2$$

The digits in bold represent the terms of an oblique/vertical subexpression. Assuming an input signal width of 8 bits, in case of a straightforward implementation without any oblique subexpression elimination, the width estimation would give the signal width as

$$N_0 = N_1 = N_2 = 12.$$

But if the vertical subexpression elimination is implemented the estimated widths would change because of the removal of the width-reducing ‘-1’ in C_2 from the subexpressions and the new width estimation would be

$$N_0 = N_1 = 12, N_2 = 13$$

3. **MCM Generation:** At this point, the tool is ready to create VHDL codes for the filter design. The first stage is to output the MCM block. sys-FIR also takes into consideration the user defined preference for the type of adder to be used during the implementation of the MCM block.
4. **ADDNDELAY Generation:** After the generation of the MCM block, sys-FIR generates the ADDNDELAY block. This process also takes into consideration the user defined preference for the type of adder to be used during the implementation of the ADDNDELAY block.

3.5.3.2 Forward Compatibility

As described in Section 3.3, sys-FIR supports whole range of CSE algorithms. Nevertheless there will always be a probability of researchers developing new algorithms for more optimal solutions in terms of the redundancy reduction and subexpression elimination. Therefore, sys-FIR has been developed in a standardized format which allows a user to fit in one’s own algorithm wrapper on top of a kernel structure provided by sys-FIR, as has been shown in Fig. 3.20.

Fig. 3.20 TFF Generator Kernel

This kernel based structure is mandatory to maintain a certain degree of opaqueness to a researcher who is only interested in generating VHDL codes for his own algorithm without getting into the finer details of the code. At the same time the following points have to be taken care of while developing a wrapper for a new algorithm:

- a) The function head has to be in line with the existing convention. The current format is AlgorithmName2sys-FIR e.g. MBPT2sys-FIR()
- b) Three inputs are to be given to this file:
 - 1) coeff (The filter coefficients in integer form).
 - 2) interaction (To issue a message if the file already exists by that name).
 - 3) addName (To append the name by a user defined text).
- c) The output file should contain the information in 3 sections as described in Section 3.2.2.
- d) The filename generated by the new algorithm should be returned from the MATLAB function 'generateSys-FIRFile'.

- e) The new algorithm function should be called from the function 'startAlgorithm.m'.

This five-step methodology will ensure that the new algorithm can be used as an integral part of sys-FIR. An example for CRA is included in Appendix II to show the MATLAB code described above.

3.5.3.3 Verification

Although MATLAB version 6.2 has a special toolbox, SimuLink for interaction with ModelSim, it was not suitable for the simulation requirements of this work. The problem lies with the fact that SimuLink cannot be launched in the background without a GUI, which in-turn needs manual supervision and interference. The chosen methodology for verification does not need this toolbox and runs ModelSim without a GUI. This makes the solution more portable because it is possible to run it with a basic MATLAB version.

The *startModelsim* utility is incorporated in sys-FIR to serve as a sequential verification environment. ModelSim version 6.2c is used for this work. First a normal compilation is done by ModelSim to ensure that all the VHDL codes that have been generated by sys-FIR are syntactically correct and have no compilation problems for the later flow. Second, a random test vector generator (s-TVG) is used to generate input test patterns to simulate the VHDL codes. The number of results generated by s-TVG can be controlled by the *random_simulation_steps* utility. Simultaneously another sys-FIR function mathematically computes the filter response for the same random signals. The final simulation outcome is then compared with the MATLAB computed result. In case an error is detected, it is printed to a report file for future reference and debugging.

3.5.3.4 Debugging

The program *debugFilt* comes in extremely handy when an error is detected at the verification step described in Section 3.5.3.3. Here the objective is to identify the origin of the error generation. The wrong output signal value that was observed after verification could have originated in any of the taps. This utility performs a more detailed mathematical computation and calculates the expected output of each tap of

the filter. These results are then printed to a text file and they can be compared with those provided by the simulator to find the exact location of the faulty node. The *debugFilt* utility is not turned on by default. It is to be used only when sys-FIR flags an error at the verification step.

3.5.3.5 Script Generation for DC

The ultimate objective of sys-FIR is to help the user in comparing different CSE algorithms for which a formal synthesis flow has to be executed. The utility *prepareDC*, which forms the core of the s-FCG (sys-FIR Frontend Constraint generator) module, helps in creating the frontend scripts for the design flow. This utility can be manually modified to include any application specific constraints in terms of timing, area or power. The new constraints would immediately appear in the final shell script written out by sys-FIR. A sample s-FCG output is included in Appendix III. This shell script can be used by Synopsys Design Compiler (DC) in synthesis and hardware optimization steps. Moreover, this utility also empowers the user to store the files at his preferred location. This becomes important when the tool is run in the batch mode.

3.5.3.6 Handling of Special cases

All cases of two's complement adders can be successfully handled by sys-FIR, including the following extreme cases

1. signals (operands or partial sums) that do not overlap at all due to a big shift in one signal.
2. signals that overlap by only one bit.
3. signals for which the input and the output are of the same size, i.e. addition generates no additional bit.

The simplicity of the corner scenarios, like the ones mentioned above, disguises the complexity involved in handling them because of the way they modify the MATLAB arrays and structures in sys-FIR. Accurate pre-estimation of the signal widths by sys-FIR is specially dependent on the precise understanding and implementation of such corner cases.

3.5.3.7 Modified-RCA and CLA

It was observed that given a set of constant CSD coefficients, the empty bit positions for each tap can be predetermined. So using a custom made CPA from the standard libraries will not take advantage of these missing bits. The hardware complexity can be reduced to a great extent in some cases if the missing bits are taken into account. The following Modified-RCA and Modified-CLA structures, which account for the empty bit positions and optimize the hardware of the CPA, were made the default CRA and CLA architectures in sys-FIR.

a) Modified-RCA

Fig. 3.21 shows an example of the sum and carry bits at the final stage of the CSA tree, where ‘*’ represents an unknown bit. In Fig. 3.21, the sum vector ‘a’ has 12 dots but the carry vector ‘b’ has only 4 dots at positions 3, 5, 7 and 8. The carry vector has four bits only. If a standard RCA is used, 1 HA and 11 FAs would be required. Instead, the improved RCA uses 6 HAs and 3 FAs. Fig. 3.22 shows the two implementations for the bit pattern in Fig. 3.21. The improved RCA capitalizes on the missing bits. The three bits in positions ‘0’ to ‘2’ can directly be sent as output. The actual addition starts at bit position ‘3’ where a HA is used in contrast to the standard-RCA (Section 2.3.2.1) where the addition starts at position ‘0’. The hardware allocation and the difference in the number of gates between the two implementations are shown in Fig. 3.22. The advantage of using the improved RCA can be seen at the missing bit positions. ‘W’ indicates a wire connection, ‘H’ signifies a half adder and ‘F’ a full adder.

Bit Position (i)	11	10	9	8	7	6	5	4	3	2	1	0
Sum (a)	*	*	*	*	*	*	*	*	*	*	*	*
Carry(b)				*	*		*		*			

Fig. 3.21 Bit pattern for CPA

11	10	9	8	7	6	5	4	3	2	1	0
*	*	*	*	*	*	*	*	*	*	*	*
0	0	0	*	*	0	*	0	*	0	0	0
Standard RCA Implementation											
F	F	F	F	F	F	F	F	F	F	F	H
Modified-RCA Implementation											
H	H	H	F	F	H	F	H	H	W	W	W
No. of Gates Saved											
3	3	3	0	0	3	0	3	3	2	2	2

Fig. 3.22 Standard-RCA vs. Modified-RCA

b) Modified - CLA

The CLA is also optimized based on a prior knowledge of the positions of constant bits from the CSA tree. This optimization reduces the gate count by a high percentage.

Consider the same example given in Fig. 3.21. All standard libraries have cells with a maximum fan-in of 4. Thus a standard CLA with 12-bit word length will be implemented in two stages with three 4-bit CLAs in the first stage and a partial 3-bit CLA in the second stage, as illustrated in Fig. 3.23. The boolean equations for the Group Generates (GG) and Group Propagates (GP) have been described in Section 2.3.2.2.

Fig. 3.23 CLA implementation for bit pattern in Fig. 3.21

Let us focus on the last four bits in the example, i.e. bit positions '0' to '3'. The input vector 'a' has four input bits whereas 'b' has only one input bit. If a CLA from a standard library is used to implement the 4-bit CLA, the missing bits at bit positions '0', '1' and '2' would have '0' as the bit value. Propagates (p_i) and Generates (g_i) are determined according to Eqs. (2.10) and (2.11). The hardware to implement this 4-bit CLA is shown in Fig. 3.24.

Fig. 3.24 4-bit Standard-CLA Logic

The Modified-CLA exploits these constant inputs to simplify the logic circuits for the propagation ' p_i ' and generate ' g_i '. In the 4-bit CLA that was considered, expect for bit position '3', $p_i = a_i$ and $g_i = 0$. Since $g_i = 0$, the boolean equations for carry outputs can be reduced to the following.

$$c_1 = p_1 c_0$$

$$c_2 = p_1 p_0 c_0$$

$$c_3 = p_2 p_1 p_0 c_0$$

$$gG = g_3$$

$$gP = p_3 p_2 p_1 p_0$$

The hardware for the 4-bit Modified-CLA is shown in Fig 3.25. The standard 4-bit CLA uses 16 gates whereas the improved CLA uses 6 gates only. If the CLA for the entire 12-bit input (given in Fig 3.21) is considered, the number of gates is reduced by 50% in the first stage. The second stage CLA will not have any reduction, as all the inputs are unknown.

Fig. 3.25 4-bit Modified-CLA Logic

3.5.3.8 Optimization of CSA Trees

As described in Section 2.2 the constant coefficient of each tap is approximated as SPT terms. If tap 'N' has three SPT terms, there will be three input

words, the shifts of which are given by the SPT terms. Fig. 3.26 gives the dot notation for the coefficient $h(1) = 2^{-2} + 2^{-4} + 2^{-6}$. Input x is shifted by 2, 4 and 8 bits to the right in the shifter block to give three 8 bit signals sig1, sig2 and sig3, where 'o' indicates a shift and '*' implies the presence of a bit.

Fig. 3.26 Shifting Operation

The shifted signals are then added using merged arithmetic architecture (Section 2.6). For the transposed direct form implementation, Carry-Save Adder (CSA) trees with Wallace tree reduction are used. Fig. 3.27 shows the Wallace tree reduction for the input signals from Fig. 3.26

The reduction can be divided into several stages. At each stage, the maximum number of bits is eliminated by using Full Adders (FAs) and Half Adders (HAs). The reduction is continued till a maximum of two dots is obtained in the tree. In Fig. 3.27, the three partial sums in Stage 1 are reduced to two partial sums in Stage 2. Since the maximum dots per column in Stage 2 are two, the reduction stops.

Fig. 3.27 Carry Save Adder Tree Reduction

Following are the main steps in the optimization of the CSA tree.

- (1) The reduction starts from the LSB, i.e. column '0'.
- (2) If there is 1 dot per column (in Stage N), it is passed to the next stage without any reduction (to Stage N+1). E.g. columns '0' and '1' have one dot each in

- stage 1. They are passed to Stage '2'. This is done in order to avoid any redundant use of adders with one input being tied to '0'.
- (3) Starting from the LSB, if 2 dots per column are found before 3 dots per column, the two dots are passed to the next stage without reduction. E.g. columns '2' and '3' have two dots each. Starting from column '0', we first encounter 2 dots in columns '2' and '3' and then three dots are found in columns '4' to '7'. Since the final addition is done by the Carry Propagation Adder(CPA) which will have two input signals, the two dots are passed to Stage 2 without reduction, i.e. a HA is not used to reduce the two dots in Stage 1.
- (4) A full adder (FA) is used whenever three or more dots are found in a column. Table 3.1 gives the hardware allocated for various numbers of dots per column.

Table 3.1 Hardware Utilization in Example of Fig. 3.27

Dots per Column	Hardware
2	HA
3	FA
4	FA + wire
5	FA + HA
6	FA + FA
7	FA + FA + wire
8	FA + FA + HA

If there are 4 dots per column, an FA is used to account for 3 bits and the remaining one bit is sent to the next stage as wire connection.

- (5) If two dots are encountered after three dots (e.g. Columns '8' to '10' have two dots each but the columns before them have three dots each), a HA is employed for their reduction.

The final result from a CSA tree is a bit pattern which contains two vectors, say 'a' and 'b'. An irregular pattern of bits can be observed. Some bits are missing in one of the final addends – 'b'. Finally, a Carry Propagation Adder (CPA) is used to sum the two addends.

3.5.3.9 Use of Recursive Adder Structures

It is possible that when a user has specified the use of CLA or RCA for adder implementation in *CSBLOCK*, *MCM*, or *ADDNDELAY* there might be more than two operands for addition. In that case, the RCA/CLA adders are invoked repeatedly as shown in Fig. 3.28.

Fig. 3.28 Recursive Adder Implementation

The adder in Fig. 3.28 caters for 6 inputs. A total of 5 CLA adders are required for this operation.

3.5.3.10 Uniform Naming Convention

A highly uniform naming convention has been used so as to ensure that any further improvement of tool could be done with a great ease, e.g., all the MATLAB files which are responsible for generating the VHDL files directly are named with an extension ‘_vhd.m’ in its file name.

3.5.3.11 Input Range Compatibility

Special precaution has been taken in sys-FIR development to ensure that a wide input length could be supported. The current range of input signal that has been tested successfully stands at 2 to 128 bits. More testing would be required for larger signal inputs.

3.5.3.12 Redundancy Removal

The concept of symmetric/asymmetric filters was discussed in Section 3.4.1. While implementing a filter by making use of its symmetrical structure helps in reducing the silicon area costs, it was noticed that there might be many other non-symmetrical taps with similar coefficient values apart from the one which is symmetrical to a certain tap. These taps can be implemented simply by taking out the signal directly from the root-tap or by shifting the output of the root-tap. sys-FIR makes use of this information and does not physically implement any such taps more than once. This further helps in hardware reduction.

This chapter presented a detailed description of sys-FIR including its structure, features and functionality. The uniqueness of sys-FIR was highlighted with the help of its new and innovative features like user-friendliness, robustness, forward compatibility, high modularity etc. The next chapter presents some of the results generated by sys-FIR and the verification of the system.

Chapter 4

Results and Discussion

This chapter is divided into four sections. The first section explains the ASIC design flow that has been developed around sys-FIR for filter generation and verification. A fast implementation of a practical direct form FIR filter through sys-FIR has been described in Section 4.2 along with its physical postlayout synthesis results. Section 4.3 demonstrates the use of sys-FIR to evaluate prelayout synthesis results of transposed direct form filters generated by different optimization algorithms.

4.1 The Design Flow

As explained in the previous chapters, sys-FIR (based on MATLAB codes) has been developed for the generation of synthesizable VHDL codes for a filter structures with a specific set of coefficients and input wordlength.

The VHDL codes can then be functionally verified by using ModelSim simulator [MEN06a, MEN06b, MEN06c] which now forms an integral part of the sys-FIR platform itself. The tool generates a set of random input signals for the Modelsim simulator while simultaneously calculating the expected output signals from the transfer function in MATLAB independently. These two sets of output values (from the circuit and from filter equation) are then compared and a flag is generated if they do not match.

These functionally verified VHDL codes are then synthesized by using Synopsys Design Compiler [SOL]. The constraints can be written manually or with the help of sys-FIR *preparedc* utility as explained in Chapter 3. For this work, the hardware optimization was done targeting TSMC 0.18 μ m CMOS technology library. The constraints are primarily related to clock frequency, input/output port delays and input/output port capacitances. A script file has been written in UNIX shell [ELL] to repeat the optimization procedure to find the maximum clocking frequency where the slack is met. Synopsys Power Compiler is then used to perform dynamic power analysis. For the results shown in Section 4.2, a gate level netlist in Verilog was

generated, corresponding to the clock period which gave a marginal positive slack. In addition to the netlist, reports were generated which gave the details about the silicon area from Design Compiler. Synopsys Apollo was used for the backend phase of the project (floorplanning, placement and routing). RC extraction was done using Star RC extractor and back annotated timing analysis was done by feeding the Star RC outputs and the netlist into Synopsys Primetime tool. The extraction and timing analysis were performed for worst-case scenarios to ensure the validity of the results in adverse operating conditions.

The design flow for the FIR filter implementation and evaluation using sys-FIR is given in Fig. 4.1.

Figure 4.1 Filter Design Flow using sys-FIR

4.2 Direct Form Implementation

A benchmark design of a 64-tap bandpass filter that operates at a sampling rate in excess of 100MHz and dissipates power in the order of mW was selected for implementation by using sys-FIR. This was to ensure that sys-FIR not only facilitates comparison between CSE algorithms but also delivers functionally correct and synthesizable VHDL codes, which can be instantly plugged into the ASIC design flow discussed in Section 4.1.

The 64-tap filter implemented by using sys-FIR in this work forms a key component of a system that performs the front-end signal processing functions in a quadrature modulator and demodulator for high bit rate digital video applications [YJD90]. The direct form structure can meet this high throughput rate requirement with far lower power consumption due to the elimination of a large number of power hungry carry propagation adders (CPAs). The design is carried out using a dedicated merged arithmetic architecture, where the multiplication of the input signal with the filter coefficients have been decomposed into signed additions implemented as a carry-save adder (CSA) tree. A hybrid column compression technique for the CSA tree has been employed to reduce its hardware complexity and the power dissipation of the final CPA. The reduced wordlength CPA is implemented with modified carry look-ahead logic.

Using the dot matrix notation where each dot represents a binary variable of increasing significance with its position from the right, the adder structure of the 64-tap FIR filter obtained by merging the multiplication with the final accumulation is shown in the first stage of Fig. 4.2. The hybrid tree reduction scheme [ZHI03] used is a combination of Wallace and Dadda strategy and involves 12 stages of CSAs demarcated by the horizontal lines in Fig. 4.2. The hybrid reduction method partitions the dot matrix map into two sub-blocks and then applies an independent Wallace and Dadda strategies on each sub-block. Table 4.1 shows the comparison of three column compression methods on the 64-tap FIR filter. While Dadda strategy uses less hardware for the CSA tree, it has massive interconnects and requires a 24-bit long CPA. If the CPA is implemented with fast CLA with block factor of 4, the critical path is lengthened by an additional delay of a 4-bit CLA stage. The hybrid method

yields the same 14-bit CPA as the Wallace strategy but it uses less hardware than Wallace reduction. The number of interconnects is modest for its CSA tree.

Fig. 4.2 Hybrid Column Compression for Direct Form FIR Architecture

Table 4.1: Hardware Costs of Different CSA tree Reduction Techniques

	Wallace	Dadda	Hybrid
Full Adder	1525	1518	1526
Half Adder	65	27	36
Number of interconnects	155	2793	1091
Wordlength of final CPA	14	23	14
CLA hierarchy (with 4-bit CLAs)	2	3	2

The CSA tree generates a 23-bit sum vector and a 13-bit carry vector. 9 LSB bits of the sum vector are directly hardwired and its remaining bits and the carry vector are fed to the 14-bit CLA. The CLA is implemented with 4-bit CLA logic blocks in two stages, as shown in Fig. 4.3.

Fig. 4.3 Carry Look-ahead Adder for Final Accumulation

The entire design was carried out on a Solaris platform running on an Ultrasparc driven SunBlade1000 dual processor workstation.

Initially, the total area occupied by the standard cells in the netlist was estimated to be $235,000 \mu\text{m}^2$ by Synopsys Design Compiler. With the estimated routing complexity, a core utilization ratio of 0.75 was chosen. This measures up to a total core area of $313,600 \mu\text{m}^2$. The core dimensions are $560 \mu\text{m}$ by $560 \mu\text{m}$. A major part of this extra area is allocated for routing channels between rows. A Row/Core ratio of 0.9 was chosen to define the width of these channels. A core to side gap of $30 \mu\text{m}$ ensures enough routing space for special Vdd and Gnd nets. The IO pads significantly increases the chip size to $1353.2 \mu\text{m}$ by $1353.2 \mu\text{m}$, giving a final layout area of 1.83 mm^2 . The layout of the chip is shown in Fig. 4.4. The design is pad-dominant as can be seen from the area occupied by the dotted boxes which are the pad

openings used for bonding the silicon to the package. A 40-pin DIP package can be used for final chip assembly.

Fig. 4.4 64-tap FIR Filter Layout

An incremental synthesis approach was adopted while generating the frontend synthesized netlist. The netlist optimization was performed using a highly conservative wire load model till the EDA tool flagged a positive slack. Finally, the design had a marginally positive slack at a clock period of 9ns. Back annotated timing analysis, performed on the backend netlist using the parasitic information generated from the final layout, showed a critical path of 7.89 ns. Therefore the chip can operate at 125 MHz at 1.8V based on the post-layout simulation results.

The dynamic power analysis was performed using a sample of 2500 randomly generated 8-bit input vectors. These vectors, along with a switching activity information file generated by Design Compiler were then passed to the ModelSim simulator for creating a back annotated SAIF file. This file along with the top gate level netlist was then used to run Power Compiler and the results are shown in Table 4.2.

Table 4.2: 64-tap FIR Filters Power Consumption

Cell Internal Power	1.1496 mW	81%
Net Switching Power	261.3972 μ W	19%
Total Power	1.4110 mW	100.00%

This power analysis was conducted at 1.8V and 125MHz. The results show that a large portion of the total power consumption of this circuit is contributed by the switching activity of the cells, and only about 20% of the power dissipation comes from the interconnects.

Table 4.3 shows the results of the transposed direct form filter chips designed by module generator gate array compiler and the direct form merged arithmetic architecture generated by sys-FIR. The programmable filter belongs to a different category with varying coefficient but it serves as a good reference to assess the achievable power consumption. Albeit major technology difference for the same filter specifications, the filter generated by sys-FIR, is able to operate at high speed of 125 MHz without compromising on other key design metrics like area and power by applying hybrid tree reduction technique on a direct form filter structure, although a significant amount of the improvement is also due to the use of advanced process technology.

Table 4.3: FIR Filters Comparison

Design Style	Module Generator[ISH90]	Gate Array Compiler[YJD90]	Module Generator[HAT90]	sys-FIR Generator
Clock Rate	13.5 MHz	100 MHz	100 MHz	125 MHz
Supply	Unspecified	Unspecified	5 V	1.8 V
Coeff. Type	Fixed	Fixed	Programmable	Fixed
Wordlength	8	14	12	14
Technology	1.2 μm CMOS	0.8 μm BICMOS	0.9 μm CMOS	0.18 μm CMOS
No. of Taps	15,15,19,15	64	80	64
Transistor Count	N/A	187k MOS 22k NPN	240k MOS	113.2k MOS
Power	N/A	6 W	3.1 W	1.41 mW

4.3 Transposed Direct Form Filters

Keeping in mind the final application of sys-FIR, it was mandatory to prove its robustness as well as its efficiency. The VHDL codes of the filter circuits for a total of 28 different filter specifications from Appendix I – ‘Filter Database’ were automatically generated and synthesized with various parametric options, and verified by sys-FIR. All frontend activities, including the generation, verification and synthesis of the VHDL codes, were performed with a short turnaround time of less than 80 man-hours. All 28 architectures were generated in direct form and in transposed direct form (with 13 available CSE algorithms) and with both CRA and CLA adder structures. The input bit width was also varied to ensure that the software is compatible for different input lengths. The bottleneck of generating and verifying filters now lies with the synthesis process and has been driven farther from creating HDL and verification. All filter structures merely took a few minutes in MATLAB generation. This speed is dependent on three factors:

1. Filter size.
2. Matlab Version.
3. Processor Speed.

For this project, MATLAB ver7.2.0.232 (R2006a) was used on Intel P4 processors (3.4GHz). Relatively, most of the time was spent on synthesizing these

filters using Design Compiler. A staggering 2352 (28-specs $\times$ 14-algorithms $\times$ 3-wordlengths $\times$ 2-adder_options) filters were created and optimized. (Here, '14-algorithms' refers to 13 CSE algorithms in transposed direct form implementation and the direct form implementation). It is beyond the scope of this thesis to present all the timing, area and power results but it is imperative to show in what possible ways researchers can use sys-FIR. sys-FIR has opened up new avenues for comparison of filters. Following are the various ways in which filter selection can be done based on different parameters. All the results shown here correspond to filters generated for 8-bit input signals unless otherwise stated.

4.3.1 VLSI metrics comparison for FIR filters generated with various CSE algorithms

It was observed that the solutions obtained by using different CSE algorithms can have a large impact on the physical VLSI performance metrics of a certain filter but it could not be concluded that the same algorithm can be universally applied to all the filters for better results. A better alternative would be to use sys-FIR to quickly assess the suitability of an algorithm for a certain filter. This has been shown with the help of Table 4.4.

Table 4.4: VLSI Metrics of dempster_n25 (Appendix I) from Solutions of Different CSE Algorithms

Algorithm	Combinational area(μm^2)	Noncombinational area (μm^2)	Total Cell Area (μm^2)	Max delay (ns)
4BCSE	114870.9	38283.41	153150.8	10.08
BCSE	112851.5	38283.41	151131.7	10.58
BHM	116580.8	38283.41	154860.5	10.21
BIN	140562.5	38283.41	178840.6	9.32
C1	113447.0	38283.41	151727.1	13.18
CRAH2	120589.2	38283.41	158868.9	9.37
CSD	130718.7	38283.41	168997.8	8.41
HARTLEY	118586.5	38283.41	156866.4	9.02
MBPT	129328.3	38283.41	167607.3	9.49
MRAG	111421.1	38283.41	149701.3	11.39
NRSCSE	116274.4	38283.41	154554.5	9.05
PASKO	116876.8	38283.41	155156.6	9.84

It is clear from Table 4.4 that for the filter used in [DEM95] while the simplistic CSD implementation is the best possible option in terms of timing, it has

high combinational area requirements. On the other hand a baseline normal binary transposed direct form implementation on an average consumes about 27% more area than a transposed direct form implementation using any other CSE algorithm. In this particular case “HARTLEY” seems to be generating the most optimum design with a critical path delay of 9.02 ns and a total cell area of $156866.4 \mu\text{m}^2$

Table 4.5: VLSI Metrics of Tommi_n281_16bit Filter (Appendix I) from Solutions of Different CSE Algorithms

Algorithm	Combinational area(μm^2)	Noncombinational area (μm^2)	Total Cell Area (μm^2)	Max delay (ns)
4BCSE	591770.6	453339.8	1045172	13.12
BCSE	592571.3	453339.8	1045973	12.96
BHM	605573.1	453339.8	1058986	14.23
BIN	728669.8	453339.8	1182179	10.7
C1	599329.4	453339.8	1052736	10.64
CRA	593039.3	453339.8	1046436	11.00
CSD	689263.8	453339.8	1142715	10.83
HARTLEY	629233.9	453339.8	1082650	11.51
MBPT	613282.9	453339.8	1066690	10.99
MRAG	591707.5	453339.8	1045108	14.13
NRSCSE	606204.8	453339.8	1059608	11.53
PASKO	597303.0	453339.8	1050703	10.83

Tommi_n281_16bit filter (Appendix I) is a large filter with 281 taps. It can be deduced from Table 4.5 that “C1” is the more promising CSE solution for this filter which performs comparatively well in terms of both timing and area metrics.

4.3.2 Direct form vs Transposed direct form Comparison

While it is obvious that direct form structures are more area efficient in comparison to transposed direct form filters due to the fixed size of registers [ZHI03], the area-delay tradeoff is not predictable. It is possible that a certain filter when implemented in transposed direct form may be able to meet its area requirement while still being much faster than in direct form. On the other hand it is also possible that a direct form implementation of a particular filter might be fast enough while still being much smaller than transposed direct form. The decision between direct and transposed direct form structures can be easily determined using sys-FIR at a very early stage of filter design. The trade-off analysis can now be made easily and more accurately with sys-FIR. Fig. 4.5 and Fig. 4.6 give a small comparison between the metrics of a direct

and transposed direct form implementation of certain filters. The results shown in Fig. 4.5 and Fig. 4.6 have been obtained from selected filter specifications out of the 28 filter specifications given in Appendix I. The filters are selected to cover a wide range of filter orders, the smallest one being an 11 tap filter and the largest one has 281 taps.

Fig. 4.5 Direct vs Transposed Direct Form Delay Comparison

Fig. 4.6 Direct vs Transposed Direct Form Area comparison

The comparisons between the two filter structures show that direct form implementation, irrespective of the filter size is an outright winner in terms of the area mainly because of the lower non-combinational areas. The selection based on delay is not as straightforward. While the direct form implementation is still suitable for

smaller filters, in large filters it can be concluded that transposed direct form is a much better choice because of shorter critical paths between registers.

4.3.3 Research on novel CSE techniques

The ease of adding additional wrappers to the existing standard file generator (ISFG) can also be used advantageously to analyze the strengths and weaknesses of new algorithms. In the graph of Fig. 4.7, different CSE algorithms are compared based on the combinational area requirements of a 281-tap filter.

Fig 4.7 CSE Dependent Area Requirements of Tommi_n281_16bit Filter

The filter Tommi_n281_16bit (Appendix I) was generated using all the available CSE algorithms and synthesized in DC using exactly same set of constraints. The graph clearly shows that HARTLEY is not the best option in this case. Some possible alternatives are C1, CRA and MRAG.

4.3.4 Different input length comparison

The capability of sys-FIR to instantly generate a filter for different wordlength input signal is useful to interface the same filter solution to different resolutions of analog-to-digital converter (ADC). It helps the user to identifying the impact of ADC resolution on the area-timing-power performances of the filter. Moreover, it also enable fast prototyping of the same filter specification and structure for different input data wordlengths.

In the Fig. 4.8, the increases in areas on the same filter due to input signal width modification are shown for four CSE algorithms: CRA, PASKO, HARTLEY and NRSCSE.

Fig. 4.8 Input Signal Wordlength Impact on Filter Size

4.3.5 Power Analysis

All the power analysis was conducted by running a dynamic power simulation using the recommended Synopsys back annotated power simulation flow [SOL] shown in Fig. 4.9.

Fig. 4.9 Synopsys Dynamic Power Simulation Flow

The forward annotation file is created using *rtl2saif* command. The *saif* file contains the information about all the nodes present in the design. A test vector file along with the *saif* is then used by the simulator to generate the switching activity observed on the various nodes due to the input test vectors. This switching activity is then output at regular intervals in the form of a back annotated *saif* files. These files are then fed back into Power Compiler to determine the average dynamic power consumption in the circuit. For every back annotated saif file a corresponding power result is generated by the tool.

The dynamic power results generated for a benchmark filter samueli01_n25 [SAM89] which was implemented for all the available CSE algorithms in sys-FIR are shown in Fig. 4.10, which clearly shows that the RCA based implementation consumes less power than CLA based implementation irrespective of the algorithm used because of low switching activity and smaller number of signal nodes in RCA adders.

Fig. 4.10 Dynamic Power Consumption of filter samueli01_n25 (Appendix I)

As shown in Fig. 4.10, CRA algorithm has been most efficient in eliminating the common subexpressions and reducing the logic depth of this filter, which in turn minimizes the switching activity and glitch power propagation. However, it is important to note that no general conclusion can be drawn on the basis of the results shown for only samueli01_n25 filter. Fig 4.11 shows the dynamic power consumption of filter samueli02_n60. In contrast to samueli01_n25, BHM outperforms the other algorithms, when used with RCA adders. For CLA adders, PASKO provides the best solution.

Fig 4.11 Dynamic Power Consumption of filter samueli02_n60 (Appendix I)

In this chapter, some of the possible ways in which sys-FIR can be used to draw realistic conclusions while selecting a particular CSE algorithm have been demonstrated. Generation of a huge database of filters served two purposes. First, it helped to ensure the robustness of sys-FIR and secondly it proved the practical application of sys-FIR in common subexpression elimination research. An application specific filter was also generated in direct form, using sys-FIR. It was then implemented by using the standard ASIC flow to validate the VHDL codes generated by sys-FIR. The next chapter covers the conclusions drawn from this work and some suggestions have been made for further improvement of sys-FIR.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The use of digital filters in DSP applications is indispensable. Although programmable filters based on DSP cores are flexible, they are not suitable for mobile applications that require low power consumption and high throughput rate. Since the flexibility of a multiplier is not necessary an advantage in such and many other applications, application-specific FIR filters are frequently designed to meet the stringent constraints. Many techniques exist to enhance the performance, reduce the silicon area and lower the power consumption of FIR filters. Elimination of common subexpressions (CSE) is one well-known and effective methodology to improve the overall performance metrics of a digital filter. Many CSE algorithms have been proposed by different researchers, each have its unique features and strengths. In order to tackle this hard problem efficiently, the cost metric assumed by these algorithms are often too simplistic and the relative merits of different algorithms may not correlate well with the results of physical implementation. To add to the problem, it is generally acknowledged in this research community that there is a lack of a common benchmark suite for a fair evaluation of the quality of different solutions to this multiple constant multiplication problem. This work fills the gap by developing an integrated platform to realize application-specific FIR filters with very short design turnaround time and enable different optimization algorithms to be speedy and efficiently compared for their VLSI metrics. Some of these existing CSE algorithms were studied in this work. A novel integrated platform named sys-FIR that forms the very core of this work has been described. This thesis summarizes the useful features of this software, along with its various potential utilizations. The tool stands apart for its robustness and speed. It has been shown that sys-FIR can be a powerful tool in the hand of researchers actively involved in the field of digital filters and DSP

applications. A fully automated VHDL code generator program for direct and transposed direct form FIR filter implementation from any filter coefficient set has been developed and documented. It can translate a given filter specification into different transpose form filter designs based on 13 built-in CSE algorithms. With sys-FIR, an accurate pre-estimate of the gate level silicon view of the design can be readily obtained.

In a nutshell, a number of novel ideas have been proposed through the project:

- (1) An EDA environment just for the generation and verification of filter design.
- (2) Use of Merged Arithmetic architecture for column compression and improved carry propagation adders for direct form filter structure.
- (3) Unify the representation of adder and tapped delay line topology of different graph-dependance and common subexpression elimination algorithms.
- (4) Implementation of less conservative range estimation for the accumulator and delay elements for transposed direct form structure based on the priori knowledge of the constant coefficients.

The results section in Chapter 4 discusses how sys-FIR helps in choosing a particular CSE algorithm for a certain filter, in selecting direct versus transposed direct form implementation and the effect of input signal length on the size of various filters. Moreover, a comprehensive database of published filters has been used to validate sys-FIR to ensure its robustness and accuracy.

5.2 Recommendations for Future Work

The work presented in this thesis has opened some new options for future activities.

1. Adding Intelligence to sys-FIR

Although sys-FIR speeds up the process of generating filters in both direct and transposed direct forms it lacks any sort of intelligence. It is completely dependent on the user defined inputs. Neither can it make any decisions on its own nor can it make

any suggestions to the user in terms of the CSE algorithm to be used to implement a particular filter. sys-FIR could be far more powerful if it has some inherent decision making capabilities based on the various metrics that it generates. Quite a few options could be possible in that scenario:

- a) Decision based on power consumption results.
- b) Decision based on Silicon area results.
- c) Decision based on timing.
- d) Decision based on adder costs.

In this case, instead of selecting a particular algorithm for the generation of a filter, the user would be able to specify the metric requirements of the filter and sys-FIR would select the CSE algorithm that provides the closest match to these requirements.

2. Integration with Frontend tools

sys-FIR has been linked to Modelsim to run verification. Further integration is possible if the tool can be linked to frontend synthesis tools like Synopsys Design Compiler and Power Compiler. This would help to remove any manual intervention at this stage of IC design. Moreover the power results could be used by sys-FIR to make certain decisions as described in 1.

3. Integration with Backend tools

Once a comparison has been made among various CSE algorithms and a final structure has been synthesized, the next logical step would be to integrate the backend tools like Cadence Encounter, Synopsys Star RC and Synopsys Primetime into sys-FIR to generate the final routed database that can be used for fabrication. This integration is required only for filters which have been optimized in terms of all the given parameters.

4. Robustness

The robustness of any software always depends on the number of corner cases that have been executed on the tool. While all the possible precautions were taken during the development of sys-FIR, it is always probable that a certain exceptional case might have been missed. Therefore it is recommended that further tests should be carried out on sys-FIR to find out any bugs and fix them immediately.

5. Adder Modifications

Currently sys-FIR provides only two options in terms of CPA implementation – High speed CLAs and Area efficient RCAs. It is possible that the adder structures developed by various EDA vendors might perform better than the above mentioned adder structures. Therefore, it is recommended that the software be adapted to include more adder structures to provide more flexibility to the user in terms of adder selection.

Author's Publications

1. Zhi Ye, Ravi Kumar Satzoda, Sharma Udit, Naveen Nazimudeen and Chip Hong Chang, "Performance Evaluation of Direct Form FIR Filter with Merged Arithmetic Architecture." in *Proc. 2004 IEEE workshop on Electronic Design, Test and Applications*, DELTA, Perth, Australia, pp. 407-409, January, 2004.
2. Sharma Udit, Fei Xu, Chip Hong Chang, C.C.Jong and K.S.yeo, "sys-FIR: A Compiler For Evaluating VLSI Performance Metyrics of Reduced Adder Cost FIR Filters," in *Proc. IEEE Symposium on Low Power, High Speed Chips (Cool Chips VIII - 2005)*, Yokohama, Japan, 2005.

References

- [BOE02] E. I. Boemo, M. Martinez-Peiro and L. Wanhammar, "Design of high-Speed multiplierless filters using a nonrecursive signed common subexpression algorithm," *IEEE Trans. Circuit Syst.*, vol. 49, no. 3, pp. 196-203, Mar. 2002.
- [BUL88] D. R. Bull, "Primitive operator digital filter synthesis using a shift biased algorithm," in *Proc. IEEE Int. Symp. Circuits and Systems*, vol. 2, pp. 1529–1532, Helsinki, Finland, Jun. 1988
- [BUL91] D. R. Bull and D. H. Horrocks, "Primitive operator digital filters," *IEE Proc. G*, vol. 138, no. 3, pp. 401-412, June 1991.
- [CHE99] C.L. Chen and A.N. Willson Jr., "A trellis search algorithm for the design of FIR filters with signed-powers-of-two coefficients." *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing*, Vol: 46, No. 1, pp. 29 –39, Jan. 1999.
- [DAD65] L.Dadda, "Some schemes for parallel multipliers," *Alta. Freq.*, Vol. 34, pp. 346-356, May 1965.
- [DEM95] A.G.Dempster and M.D. Macleod, "Use of minimum-adder multiplier blocks in FIR digital filters", *IEEE Trans. on Circuits and Systems – II: Analog and Digital Signal Processing*, Vol. 42, No. 9, pp. 569-577, Sep. 1995.
- [DEM02] S. S. Demirsoy, A. G. Demspster and I. Kale, "Power analysis of multiplier blocks," in *Proc. IEEE Int. Symp. on Circuits and Syst. I*, vol. 1, pp. 297-300, Phoenix AZ., May 2002.
- [DEL04] Z. Ye, Ravi Kumar Satzoda, Udit Sharma, N. Nazimudeen and C. H. Chang, "Performance Evaluation of Direct Form FIR filter with Merged Arithmetic Architecture" *IEEE International Workshop on Electronic Design, Test and Applications (DELTA)*, Jan. 2004.
- [ELL] "Unix Shells by Example" by Ellie Quigley.
- [FEI04] F. Xu, C. H. Chang and C. C. Jong, "Efficient Algorithms for Common Subexpression elimination in Digital Filter Design," *Proc. IEEE Int. Conf. on Speech, Acoustics, and Signal Processing*, Montreal, Canada, Vol. V, pp. 137-140, May 2004.
- [FEI05a] C. H. Chang, F. Xu and C. C. Jong, "I²CRA: Contention resolution algorithm for intra- and inter-coefficient common subexpression elimination," Special Session on Low Complexity Digital Filter Design Techniques and Their Applications, in *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS-2005)*, Kobe, Japan, pp. 1823-1826, May 23-26, pp. 2005.
- [FEI05b] F. Xu, C.H. Chang and C. C. Jong, "Contention resolution algorithm for common subexpression elimination in digital filter design," *IEEE Transactions on Circuits and Systems-II Express Brief*, vol. 52, no. 10, pp. 695-700, October 2005.

[FEI05c] C.-H. Chang F. Xu and C.-C. Jong. Modified reduced adder graph algorithm for multiplierless fir filters. *IEE Electronics Letters*, vol. 41, no. 6, pp. 302-303, March 2005.

[GUN99] J.E.Gunn, K.S.Barron, and W.Ruczcyk, "A low-power DSP core-based software radio architecture", *IEEE Journal on Selected Areas in Communications*, Vol. 17, No.4, pp-574-590, April 1999.

[HAR96] R.I. Hartley, "Subexpression sharing in filters using canonical signed digit multipliers" *IEEE Trans. on Circuits and Systems- II: Analog and Digital Signal Processing*, Vol. 43, No.10, Oct. 1996.

[HAT90] M. Hatamian and S. Rao, "A 100 MHz 40-tap programmable FIR filter chip", in Proc. IEEE Int. Symp. Circuits Syst., pp. 3053-3056, 1990.

[HAW96] R.A.Hawley, B.C.Wong, T.Lin, J.Laskowski and H.Samueli, "Design techniques for silicon compiler implementations of high-speed FIR digital filters", *IEEE Journal of Solid-state Circuits*, Vol.31, No.5, pp. 656-667, May 1996.

[HER01] J.P. Heron, R. Woods, S. Sezer and R. H. Turner, "Development of a run-time reconfiguration system with low reconfiguration overhead", *Jour. of VLSI Signal Processing*, vol. 28, pp. 97-113, 2001.

[IQB93] Z. Iqbal, M. Potkonjak, S. Dey, A. Parker, "Critical path minimization using retiming and algebraic speed up", *Proc. 30th ACM/IEEE Design Automation Conference*, pp 573-577, 1993.

[ISH90] M. Ishikawa et Al., "Automatic layout synthesis for FIR filters using a silicon compiler", *IEEE International Symposium on Circuits and Systems*, pp.2588-2591, 1990.

[JAI91] R. Jain, P. T. Yang and T. Yoshino, "FIRGEN: A computer-aided design system for high performance FIR filter intergrated circuits," *IEEE Transactions On Signal Processing*, vol. 39, no. 7, pp. 1655-1668, Jul. 1991.

[JAN02] Y. H. Jang and S. J. Yang, "Low-power CSD linear phase FIR filter structure using vertical common sub-expression," *Electronics Letters*, vol. 38, no. 15, pp. 777-779, Jul. 2002.

[JCV06] C. Jiajia, C.H. Chang and A. P. Vinod, "Maximum likelihood disjunctive decomposition to reduced multirooted drag for fir filter design" *IEEE International Symposium on Circuits and Systems (ISCAS)*, Athens, Greece 2006.

[JOC01] A. Y. Jeffrey O. Coleman, "Fractions in the canonical-signed-digit number system," *Conference on Information Sciences and Systems*, The Johns Hopkins University, pp. 2, March 2001.

[JUH01] J.Yli-Kaakinen and T.Saramaki, “A systematic algorithm for the design of multiplierless FIR filters”, *The 2001 IEEE International Symp. on Circuits and Systems (ISCAS 2001)*, Vol: 2, pp. 185-188, May 2001.

[KOL97] R.K. Kolagotla, H.R. Srinivas and G.F. Burns, “VLSI implementation of a 200 MHz 16x16 left-to-right carry-free multiplier in 0.35um CMOS technology for next-generation DSPs”, *Proc. of the IEEE Int. Conf. on Custom Integrated Circuits*, pp. 469-472, 1997.

[KEL92] Kelliher, T.P. Owens R.M. Irwin M.J. Hwang T., “ELM-a fast addition algorithm discovered by a program”, *IEEE Trans. On Computers*, Vol. 41, Issue 9, pp.1181 - 1184 Sept. 1992.

[LYO04] R. G. Lyons, “Understanding Digital Signal Processing”.

[MAY] “Design of High Speed Multiplierless FIR Filter” – Diploma Thesis by Matthias May, Patricia Hane, NTU Jan 2003.

[MEN06a] Mentor Graphics Corporation, <http://www.mentor.com>. ModelSim SE Reference Manual, software version 6.2c edition, August 2006.

[MEN06b] Mentor Graphics Corporation, <http://www.mentor.com>. ModelSim SE Tutorial, software version 6.2c edition, August 2006.

[MEN06c] Mentor Graphics Corporation, <http://www.mentor.com>. ModelSim SE User’s Manual, software version 6.2c edition, August 2006.

[NAG96] C. Nagendra, M.J. Irwin and R.M. Owen, “Area-time-power tradeoffs in parallel adders,” *IEEE Trans. On Circuits and Systems – II: Analog and Digital Signal Processing*, vol. 43, no. 10, pp. 689-702, Oct. 1996.

[PAR00] B.Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2000.

[PAR01] I.C.Park and H.J.Kang, “Digital filter synthesis based on minimal signed digit representation”, *Proc. IEEE Design Automation Conference*, pp. 468 –473, 2001.

[POT96] M. Potkonjak, M. B. Srivastava, and A. P. Chandrakasan, “Multiple constant multiplications: Efficient and versatile framework and algorithms for exploring common subexpression elimination,” *IEEE Trans. Computer-Aided Design*, vol. 15, no. 2, pp. 151-165, Feb. 1996.

[REU89] P.Reutz, “The architectures and design of a 20-MHz real-time DSP chip set,” *IEEE Jour.l of Solid-State Circuits*, Vol. 24, pp. 338-348, Apr. 1989 .

[RPD99] R. Pasko, P. Schaumont, V. Derudder, S. Vernalde and D. Ďuračková, “A new algorithm for elimination of common subexpressions,” *IEEE Transactions on Computer-Aided Design*, vol. 18, no. 1, pp. 58-68, Jan. 1999.

- [SAM89] H. Samueli, "An improved search algorithm for the design of multiplier-less FIR filters with powers-of-two coefficients," *IEEE Transactions on Circuits and Systems*, vol. 36, no. 7, pp. 1044-1047, Jul. 1989.
- [SCH99] P. Schaumont, R. Pajsko, V. Derudder, S. Vernalde, D. Durajckova, "A new algorithm for elimination of common subexpressions", *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, Vol. 18, No. 1, Jan 1999.
- [SCH99] M.J.Schulte, J.E. Stine, J.G. Jansen, "Reduced power dissipation through truncated multiplication", *Proc. IEEE Alessandro Volta Memorial Workshop on Low Power Design*, pp. 61-69, March 1999.
- [SMI06] K. P. G. Smitha. 4-bit binary common subexpression elimination. "A New Binary Common Subexpression Elimination Method for Implementing Low Complexity FIR Filters," *International Symposium on Circuits and Systems* (Under Review) December 2006.
- [SOL] Synopsys Online Documentation (SOLD)
- [VIN06] A. P. Vinod, and E. M. Lai, "Low power and high-speed implementation of FIR filters for software defined radio receivers," *IEEE Transactions on Wireless Communications*, vol. 5, iss. 7, pp. 1669-1675, Jul., 2006.
- [YJD90] T. Yoshiono, R. Jain. P.T.Yang, H. Davis, W. Gass and A. H. Shah, "A 100-MHz 64-tap FIR digital filter in 0.8-um BiCMOS gate array", *IEEE Journal Of Solid-State Circuits*, Vol. 25, No. 6, Dec. 1990.
- [ZHI03] Z. Ye and C. H. Chang, "A hybrid CSA tree with merged arithmetic architecture for FIR filter", *Proc. 3rd Int. Symp. on Image and Signal Processing and Analysis (ISPA 2003)*, Sep. 2003.
- [ZHO95] Z.Wang, G.A.Jullien and W.C. Miller, "A new design technique for column compression multipliers", *IEEE Trans. on Computers*, Vol. 44, No.8, pp. 962-970, Aug 1995.

Appendix I: Filter Database

Filter	Pass band	Stop band	Reference
dempster_n25	0.25	0.3	DEM02
jain_n11	-	-	JAI91
jang_n11	-	-	JAN02
laskowski_n43_16bit	0.2	0.3	LAS92
laskowski_n43_52bit	0.2	0.3	LAS92
parksMcClellan_n26_16bit	.2 pi	.25 pi	VIN03
parksMcClellan_n26_52bit	.2 pi	.25 pi	VIN03
pasko_n121_16bit	0.783	0.74	RPD99
pasko_n121_52bit	0.783	0.74	RPD99
samueli01_n25	0.15	0.25	SAM89
samueli02_n60_16bit	0.021	0.07	SAM89
samueli02_n60_52bit	0.021	0.07	SAM89
Tommi_n281_16bit	0.5Hz	49.5Hz	RAI94
Tommi_n281_52bit	0.5Hz	49.5Hz	RAI94
vinod_n15_16bit	135.44 kHz	541.67 khz	VIN03
vinod_n15_51bit	135.44 kHz	541.67 khz	VIN03
YCLim_63_16bit	0.1	0.14	YON83
YCLim_63_52bit	0.1	0.14	YON83
YCLim_n121_16bit	0.4	0.37	YON83
YCLim_n121_52bit	0.4	0.37	YON83
YCLim_n36	0.23	0.29	YON83
YCLim_n37	0.23	0.29	YON83
Zhao_n25_16bit	0.15	0.25	ZHA88
Zhao_n25_52bit	0.15	0.25	ZHA88
KP1	0.15	0.25	KAN00
KP2	0.15	0.2	KAN00
KP3	0.1	0.15	KAN00
KP4	0.1	0.12	KAN00

Appendix II: CRAH2 MATLAB link to TFF Kernel

```
function fileName = CRAH2sysFIR(coeff,interaction,addName)

% Function requires at least three input argument
if(nargin < 3)
    error(' Please use function in the following way:  fileName =
CRAH2sysFIR(coeff,interaction,addName)');
end;

disp(sprintf('-----START CRAH-2-----'));
disp(sprintf(' Please wait ...'));

fileName = sprintf('sysFIR_CRAH2_%s.txt',addName);
csdCoeff = [];           %includes the coeff in csd form
workDir = pwd;           %save work directory
numberOfShift = [];
sectionOneVector = []; %includes the integer value of the common subexpression
sectionOne = { };       %includes the whole common subexpression
sectionTwo = { };
sectionThree = { };

%save sign of the coeff
signOfCoeff = zeros(1,length(coeff)); %1 = positiv, -1 = negativ
signOfCoeff = 1 + signOfCoeff;
y = find(coeff < 0);
for i = 1: length(y)
    signOfCoeff(y(i)) = -1;
    coeff(y(i)) = -coeff(y(i));
end;

%remove redundance information, because this algorithm has some problem to
%handle which appears twice ore more.
%shift the coeff till all of them are odd or zero
shiftedCoeff = abs(coeff);
y = 0;

for i = 1: length(shiftedCoeff)
    if(shiftedCoeff(i) == 1)           %when coeff = 1, then is shifting = 0
        numberOfShift(i) = 0;
        sectionThree{i} = [1 1 0];
    elseif(shiftedCoeff(i) == 0)       %when coeff = 0, then is shifting = 0
        numberOfShift(i) = 0;
        sectionThree{i} = 0;
    else
        while(mod(shiftedCoeff(i),2) == 0)
            if(shiftedCoeff(i) == 2) %if this coeff is a power of two coeff so write it directly to the section
three
                sectionThree{i} = [2 1 1];
                break;
            else
                shiftedCoeff(i) = shiftedCoeff(i) / 2;
            end
        end
    end
end
```



```

        y = y + 1;
    end;
end;
numberOfShift(i) = y;
y = 0;
end;
end;

reducedCoeff = unique(shiftedCoeff);

%transform the coeff in csd form
for i = 1: length(reducedCoeff)
    tmp = dec2csd(reducedCoeff(i));
    y = size(csdCoeff);
    if(length(tmp) > y(2))
        csdCoeff = [zeros(y(1),length(tmp)-y(2)) csdCoeff];
    elseif(length(tmp) < y(2))
        tmp = [zeros(1,y(2)-length(tmp)) tmp];
    end;
    csdCoeff(i,:) = tmp;
end;

%disp(csdCoeff);

%run CRA algorithm
cd(strcat(workDir,'algorithms\CRAH2'));
try
    [subexpression,cseTable] = CRAH2(csdCoeff);
catch
    disp(sprintf('\n !!!!! ERROR occured in CRAH-2 algorithm !!!!!'));
    disp(sprintf('\n CRAH-2 can not handle this coeff: \n  %s',num2str(coeff)));
    err = lasterror;
    disp(sprintf('\n Error message of CRAH-2: \n\t%s',err.message));
    % check if the variable err.stack had been applied or not
    if(isfield(err,'stack'))
        for i = 1: length(err.stack)
            disp(sprintf(' Error in file:  %s',err.stack(i).file));
            disp(sprintf('  Error name:  %s',err.stack(i).name));
            disp(sprintf('    in Line:  %i',err.stack(i).line));
        end;
    end;
    cd(workDir);
    error('CRAH-2 algorithm stoped, because : "%s"',err.message);
end;
cd(workDir);

%disp('subexpression = ');disp(subexpression);
%disp('cseTable = ');disp(cseTable);

%----- Start transformation form the CRAH2 output to sysFIR form-----
numberOfCoeff = length(reducedCoeff);
noneZeroBits = size(cseTable,1);

lengthOfCsdCoeff = size(csdCoeff,2);

```

```

%fill the first and third section up
for i = 1: numberOfCoeff

    sectionThreeTmp = reducedCoeff(i);
    %get the nonezerodigits of the csdCoeff(i)
    noneZeroDigit = find(csdCoeff(i,:));

    prevNzBit = 0;
    currNzBit = 1;
    checkedNzBit = zeros(1,noneZeroBits);

    if(~(isempty(noneZeroDigit)))
        while(1)
            if(cseTable(currNzBit,1,i) ~= 0)
                comSub = [];
                subexpName = cseTable(currNzBit,1,i);
                while(1)
                    comSub(noneZeroDigit(currNzBit)) = csdCoeff(i,noneZeroDigit(currNzBit));
                    checkedNzBit(currNzBit) = 1;

                    goToNzBit = cseTable(currNzBit,2,i);
                    if(prevNzBit == goToNzBit)
                        %common subexpression has been rebuilt
                        break;
                    else
                        prevNzBit = currNzBit;
                        currNzBit = goToNzBit;
                    end
                end;
            end;
            %check if the common subexpression is the same than in the
            %subexpression description matrix
            subexp = 1;
            subexp = [subexp zeros(1,subexpression(subexpName,2))];
            if(subexpression(subexpName,1) == 1)
                subexp = [subexp -1];
            else
                subexp = [subexp 1];
            end;

            comSubTmp = comSub;
            y = find(comSub);
            comSub = comSub(y(1):end);
            if(~(strcmp(num2str(subexp),num2str(comSub))))
                if(~(strcmp(num2str(subexp),num2str(-comSub))))
                    error(sprintf('rebuilt subexpression is not same than it refers to it in the
subexpressionwhithorder matrix \n\t %s ~= %s',num2str(subexp),num2str(comSub)));
                end;
            end;

            %fill subexpression to section one if it is no already there
            csInt = csd2dec(comSub);
            if(isempty(sectionOneVector)) %check if it is the first subexpression
                sectionOneVector(1) = abs(csInt);
                if(csInt < 0)
                    comSub = -comSub;
                end;
            end;
        end;
    end;
end;

```

```

end;
sectionOne{1} = [abs(csInt) comSub];
else
if(~(ismember(abs(csInt),sectionOneVector)))
m = 1;
while(sectionOneVector(m) < abs(csInt))
m = m + 1;
if(m > length(sectionOneVector))
break;
end;
end;
sectionOneVector = [sectionOneVector(1:m-1) abs(csInt) sectionOneVector(m:end)];
if(csInt < 0)
comSub = -comSub;
end;
sectionOne = [sectionOne(1:m-1) {[abs(csInt) comSub]} sectionOne(m:end)];
end;
end;

%fill subexpression to section three
sectionThreeTmp = [sectionThreeTmp csInt (lengthOfCsdCoeff - size(comSubTmp,2))];

else
%check if at this position is a one or a zero
if(length(noneZeroDigit) >= currNzBit)
%this one is not used for a subexpression. so put it to section
%three
sectionThreeTmp = [sectionThreeTmp csdCoeff(i,noneZeroDigit(currNzBit))
(lengthOfCsdCoeff - noneZeroDigit(currNzBit))];
end;
checkedNzBit(currNzBit) = 1;
end;

y = find(checkedNzBit == 0);
if isempty(y)
%all ones had been find and placed to section three
break;
else
prevNzBit = 0;
currNzBit = y(1);
end;
end;
end;

%add the description to section three
y = find(shiftedCoeff == sectionThreeTmp(1));
if(~(isempty(y)))
for k = 1: length(y)
sectionThree{y(k)} = sectionThreeTmp;
end;
else
error(sprintf(' There is no coefficient like: %d',sectionThreeTmp(1)));
end;
end;
end;

```

```
%shift the coeff back and multiply the sign to each coeff of sectionThree
for i = 1: length(coeff)
    tmp = sectionThree{i};
    tmp(1) = tmp(1) * signOfCoeff(i) * 2^numberOfShift(i);
    for ii = 2: 2: length(tmp)
        tmp(ii) = tmp(ii) * signOfCoeff(i);
        tmp(ii+1) = tmp(ii+1) + numberOfShift(i);
    end;
    sectionThree{i} = tmp;
end;

%----- generate sysFIR file -----

fileName = genSysFIRFile(fileName,'CRAH2',sectionOne,sectionTwo,sectionThree,interaction);
disp(sprintf('-----CRAH2 DONE-----'));
```

Appendix III: s-FCG Output

```
echo ""
#### script running - run_synthesis.scr
echo ""

# Set the variable CELL_LIB_PATH that defines the path to access cell libraries
set CELL_LIB_PATH "/library/Artisan/aci/sc/synopsys/ /library/Artisan/aci/sc/symbols/synopsys/
/library/Artisan/fe_tpz973g_230b/TSMCHOME/digital/synopsys/tpz973g_230a"

# Set the variable CELL_TARGET_LIBRARY that defines the Target libraries
set CELL_TARGET_LIBRARY "typical.db tpz973gtc.db"

# Set the variable CELL_SYMBOL_LIBRARY hat defines the Symbol libraries
set CELL_SYMBOL_LIBRARY "tpz973g.sdb tsmc18.sdb"

## set design parameter
# Set the package list used in the design for synthesis
set PACKAGE_LIST ""

# Set the design list to be synthesized
set DESIGN_LIST [list \
    addndelaycla_10_0_18_0_18_0\
    addndelaycla_10_0_8_0_10_0\
    addndelaycla_12_1_13_0_13_0\
    addndelaycla_12_1_18_0_18_0\
    addndelaycla_13_2_13_0_14_0\
    addndelaycla_13_2_18_0_18_0\
    addndelaycla_14_3_15_0_16_0\
    addndelaycla_14_3_18_0_18_0\
    addndelaycla_16_1_17_0_18_0\
    addndelaycla_16_3_16_0_17_0\
    addndelaycla_16_3_18_0_18_0\
    addndelaycla_8_0_18_0_18_0\
    addndelaycla_i11_0_12_0_13_0\
    addndelaycla_i11_0_18_0_18_0\
    addndelaycla_i11_3_11_0_12_0\
    addndelaycla_i11_3_18_0_18_0\
    addndelaycla_i14_1_15_0_15_0\
    addndelaycla_i14_1_18_0_18_0\
    addndelaycla_i14_3_14_0_15_0\
    addndelaycla_i14_3_18_0_18_0\
    addndelaycla_i8_0_10_0_11_0\
    addndelaycla_i8_0_14_0_14_0\
    addndelaycla_i8_0_18_0_18_0\
    cla_1_1_1_0\
    cla_2_0_0_0\
    cla_2_0_0_1\
    cla_2_0_1_0\
    cla_2_0_1_1\
```

```
cla_2_1_1_0\  
cla_3_0_0_0\  
cla_3_0_0_1\  
cla_3_0_1_1\  
cla_3_1_1_0\  
cla_4_0_0_0\  
cla_4_0_0_1\  
cla_4_0_1_0\  
cla_4_1_0_0\  
cla_4_1_1_0\  
claadder\  
claadder_nc\  
claadder_npg\  
cscla_8_10_0\  
csmadd_8_p7_3\  
latch_10\  
latch_11\  
latch_12\  
latch_13\  
latch_14\  
latch_15\  
latch_16\  
latch_17\  
latch_18\  
latch_8\  
mcmcla_p11_p1_3_p3_0\  
mcmcla_p123_p7_4_p11_0\  
mcmcla_p17_p1_4_p1_0\  
mcmcla_p23_p1_4_p7_0\  
mcmcla_p5_p1_2_p1_0\  
mcmcla_p7_p1_2_p3_0\  
mcmadd_p123_p7_p1_p3\  
csblock\  
mcm\  
addndelay\  
filter\  
]
```

```
# Set the Top-Level design to be synthesized  
set TOP_DESIGN "filter"
```

```
# Set the list of pads to be utilized for synthesis  
set PAD_LIST ""
```

```
## Set the variable MEMORY_LIBRARY as a list with the .db memory files  
set MEMORY_LIBRARY ""
```

```
# Read a technology or symbol library into dc_shell  
# Write a compiled library to disk in Synopsys database format
```

```
# Set the variable search_path that is used by DC to access all necessary libraries.  
set search_path ". $env(SYNOPSYS)/libraries/syn/ /library/Artisan/aci/sc/synopsys/  
/library/Artisan/aci/sc/symbols/synopsys/  
/library/Artisan/fe_tpz973g_230b/TSMCHOME/digital/synopsys/tpz973g_230a ../Mem"
```

```
# Set the variable symbol_library that is used by DC to access the symbol library.
set symbol_library      $CELL_SYMBOL_LIBRARY

# Set the variable target_library that is used by DC to access the target library.
set target_library      [concat $CELL_TARGET_LIBRARY $MEMORY_LIBRARY]

# Set the variable link_library that is used by DC to access the link library.
set link_library        [concat \
{*}                      \
$target_library           \
$synthetic_library       \
$MEMORY_LIBRARY          \
]

# Define the Library to be used for IOPADs
define_design_lib IOLIB018 -path CELL_LIB_PATH

# Set the prefix and suffix to the names of the compiled files
set compile_instance_name_prefix "X"
set compile_instance_name_suffix "X"

# Invoke the script that sets the user defined synthesis options
source ./Script/user_option.scr

# Setting library to work with.
define_design_lib WORK -path ./Work

# Analyze the files in the package list.
# Analyzing HDL files checks the design for errors and converts them into a intermediate format
foreach MODULE "$PACKAGE_LIST" {
    if ![analyze -format vhd1 -lib WORK ../${MODULE}.vhd] {
        puts stderr "Error during analyse of $MODULE."
        echo "Error during analyse of $MODULE." > ./error
        exit 3
    }
}

# Analyze the files in the design list
foreach MODULE "$DESIGN_LIST" {
    if ![analyze -format vhd1 -lib WORK ../${MODULE}.vhd] {
        puts stderr "Error during analyse of $MODULE."
        echo "Error during analyse of $MODULE." > ./error
        exit 3
    }
}

# Elaborate the files in the design list
# Elaboration creates a technology-independent design from the intermediate format
foreach MODULE "$DESIGN_LIST" {
    if ![elaborate ${MODULE} -arch arc${MODULE} -lib WORK -update] {
        puts stderr "Error during elaborate of $MODULE."
        echo "Error during elaborate of $MODULE." > ./error
    }
}
```

```
        exit 3
    }
    write -format db -hierarchy -output ./DB/${MODULE}.db
}

# Check the current design for consistency
check_design

# Remove multiple copies of a design in memory
remove_design -designs

# Read designs into dc_shell
read_file ./DB/${TOP_DESIGN}.db

# Set the working design in dc_shell
set current_design ${TOP_DESIGN}

# Invoke the constraints.scr script file to set the constraints on the top module
source ./Script/constraints.scr

# Removes multiply-instantiated hierarchy in the current_design by creating a
# unique design for each cell instance
uniquify

# Resolve design references
link

# Perform logic and gate level synthesis and optimization on the current design
if ![compile -map_effort high] {
    puts stderr "Error during elaborate of $MODULE."
    echo "Error during elaborate of $MODULE." > ./error
    exit 3
}

# Write a design netlist or schematic from dc_shell to a file
write -format db -hierarchy -output ./DB/SYN_${TOP_DESIGN}.db

# Report the synthesis details
set current_design ${TOP_DESIGN}
report_area > ./Log/Area/${TOP_DESIGN}.area
report_timing > ./Log/Timing/${TOP_DESIGN}.timing
report_power > ./Log/Power/${TOP_DESIGN}.power
report_reference > ./Log/Reference/${TOP_DESIGN}.reference
report_resources > ./Log/Resources/${MODULE}.resources
report_constraint -verbose > ./Log/Constraints/${TOP_DESIGN}.constraints

# Copy the view_command.log to the directory for view_command logging purpose
sh /usr/bin/cp view_command.log ./Log/ViewCommand/view_command.log

# Quit DC after execution of all scripts
quit
```


Appendix IV: Sample Top-level VHDL File created by sys-FIR

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY filter IS
  PORT(
    clk : in std_logic;
    nrst : in std_logic;
    ipsignal : in std_logic_vector(7 downto 0);
    opsignal : out std_logic_vector(16 downto 0));
END filter;

ARCHITECTURE arcfilter OF filter IS

  COMPONENT mcm
  PORT(
    ipsignal : in std_logic_vector(7 downto 0);
    c_output_10 : out std_logic_vector(7 downto 0);
    c_output_9 : out std_logic_vector(8 downto 1);
    c_output_8 : out std_logic_vector(9 downto 2);
    c_output_7 : out std_logic_vector(11 downto 1);
    c_output_6 : out std_logic_vector(13 downto 1);
    c_output_5 : out std_logic_vector(16 downto 7);
    c_output_4 : out std_logic_vector(13 downto 1);
    c_output_3 : out std_logic_vector(11 downto 1);
    c_output_2 : out std_logic_vector(9 downto 2);
    c_output_1 : out std_logic_vector(8 downto 1);
    c_output_0 : out std_logic_vector(7 downto 0));
  END COMPONENT;

  COMPONENT addndelay
  PORT(
    clk : in std_logic;
    nrst : in std_logic;
    c_input_10 : in std_logic_vector(7 downto 0);
    c_input_9 : in std_logic_vector(8 downto 1);
    c_input_8 : in std_logic_vector(9 downto 2);
    c_input_7 : in std_logic_vector(11 downto 1);
    c_input_6 : in std_logic_vector(13 downto 1);
    c_input_5 : in std_logic_vector(16 downto 7);
    c_input_4 : in std_logic_vector(13 downto 1);
    c_input_3 : in std_logic_vector(11 downto 1);
    c_input_2 : in std_logic_vector(9 downto 2);
    c_input_1 : in std_logic_vector(8 downto 1);
    c_input_0 : in std_logic_vector(7 downto 0);
    opsignal : out std_logic_vector(16 downto 0));
  END COMPONENT;
```

```
SIGNAL c_10_con : std_logic_vector(7 downto 0);  
SIGNAL c_9_con : std_logic_vector(8 downto 1);  
SIGNAL c_8_con : std_logic_vector(9 downto 2);  
SIGNAL c_7_con : std_logic_vector(11 downto 1);  
SIGNAL c_6_con : std_logic_vector(13 downto 1);  
SIGNAL c_5_con : std_logic_vector(16 downto 7);  
SIGNAL c_4_con : std_logic_vector(13 downto 1);  
SIGNAL c_3_con : std_logic_vector(11 downto 1);  
SIGNAL c_2_con : std_logic_vector(9 downto 2);  
SIGNAL c_1_con : std_logic_vector(8 downto 1);  
SIGNAL c_0_con : std_logic_vector(7 downto 0);
```

BEGIN

```
    mcm_inst : mcm port map(ipsignal, c_10_con, c_9_con, c_8_con, c_7_con, c_6_con, c_5_con,  
c_4_con, c_3_con, c_2_con, c_1_con,  
    c_0_con);  
    addndelay_inst : addndelay port map(clk, nrst, c_10_con, c_9_con, c_8_con, c_7_con, c_6_con,  
c_5_con, c_4_con, c_3_con, c_2_con, c_1_con,  
    c_0_con, opsignal);
```

END arcfilter;