Stony Brook University

Generation of Customized

Time Domain FIR Filter Hardware

A Thesis Presented

by

Yujie Wu

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Electrical Engineering

Stony Brook University

December 2013

Stony Brook University

The Graduate School

Yujie Wu

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Peter Milder, Advisor of Thesis
Assistant Professor, Department of Electrical and Computer Engineering

Sangjin Hong
Professor, Department of Electrical and Computer Engineering

This thesis is accepted by the Graduate School

Charles Taber

Dean of the Graduate School

Abstract

Although finite impulse response (FIR) filtering is a well-known technique, it is still difficult to implement efficiently as hardware because the designer must choose from many application-specific design options, and it is difficult to choose those that best fit the requirements of the system. This thesis describes two design and simulation tools which enable easy implementation and optimization of time-domain FIR filters. The first generates hardware (as synthesizable Verilog) for a designer-specified FIR filter and the second provides a fixed-point simulation environment for the design space (using MATLAB). Both tools are customized based on the user's choices across a parameterized design space. In this thesis, we first design a flexible family of direct form time-domain FIR filters and optimize their adder structures. Then we introduce the accompanying flexible hardware generation tool which can produce synthesizable Verilog based on the user's specifications, and the MATLAB-based fixed-point simulator, which can verify the generator and evaluate the error of the fixed-point implementation. After creating these tools, we use them to carry out synthesis-based experiments to evaluate the tradeoffs among accuracy, area and speed of the time-domain FIR filter. We compare the results with a frequency-domain FIR filter.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I owe my deepest gratitude to my parents, without whose support, I could not realize my dream to study abroad.

Special thanks go to my helpful supervisor, Professor Peter Milder. The guidance and support that he gave truly helped the progression and smoothness of the project. The project would be nothing without his patience in helping me solve the questions encountered in the project.

I also thank my committee member Professor Sangjin Hong for his valuable feedback, which helped improve my thesis.

I would also like to thank Han Chen for allowing me to include his data for frequency domain filter in my evaluation.

Last but not least, I would like to thank my friends at Stony Brook University who made my life enjoyable here in the US. Their encouragement made me adapt to the new college life here quickly.

Yujie Wu

December 2013

Chapter 1

Introduction

Filters, which are used to boost or attenuate frequency components of a signal [1], are a fundamental tool in signal processing [2], communications [3], image processing [4] and many other related areas. A digital filter performs mathematical operations on a sampled, discrete-time signal to reduce or enhance certain aspects of that signal. Among digital filters, finite impulse response (FIR) filters are widely used due to their stability and their linear phase property. Some applications need FIR filters to operate at high frequencies such as video conferencing, whereas some other applications require FIR filters with high throughput and low power such as mobile devices used for communication and audio processing. For many applications, customized hardware implementation as a field-programmable gate array (FPGA) or application-specific integrated circuit (ASIC) is utilized for speed and efficiency (e.g. [5] and [6]).

Hardware realization of FIR filters is not typically studied in a systematic way. A designer must make several choices when implementing a FIR filter to make sure that application requirements are met, including whether to filter in the time or frequency domain, what data type and filter structures to use. Each choice leads to complicated tradeoffs between cost, performance and accuracy, and it is difficult for a designer to determine the best choices.

1.1 Objective

This thesis describes a hardware generator that emits customized time-domain (TD) FIR filter implementations based on the user's specification. These implementations are written as synthesizable Verilog, and are appropriate for FPGA/ASIC. We carry out detailed synthesis-based experiments to study the effects of changing parameters on the area and speed of the filter. We also built a MATLAB model to help us understand how the error changes when we change the parameters. The hardware generator and the MATLAB model allow for evaluation of cost, performance and accuracy. Then we compare the time domain (TD) FIR filters with frequency domain (FD) FIR filters, with the goal of allowing designers to understand the tradeoffs among filter types and parameters.

1.2 Related Work

Optimized software implementations of FIR filters in both frequency domain and time domain have been studied in depth in the past. This includes automated approaches such as [7], low-level algorithmic optimizations such as [8], and adaptive algorithms such as [9]. However, for hardware realization there are significant additional challenges, because the cost and performance requirements are highly application-specific, and different design options are best for different situations. The result is that most existing hardware approaches in the literature are either theoretical or focused narrowly on a single problem. For example previous work [10] and [11] compared time-domain and frequency-domain filter choice only theoretically, and did not allow scaling to high throughput, which is required by modern

applications such as DSP for optical networking (e.g. [12]). [13] is application-specific, comparing the accuracy of frequency domain and time domain implementations of a specific filter (root raised cosine). The comparison is only based on the length and spectral shape of the filter, and does not consider any cost or speed metrics.

Other design techniques, such as work on constant multiplication [14] [15] [16] [17] and retiming [18] can be applied to FIR filtering as well, but are not considered here. Others such as [19] consider dynamic designs where parts of computation can be activated or deactivated at runtime, a technique that could be easily incorporated into the tools discussed in this thesis.

1.3 Organization

The rest of the thesis is organized as follows. In Chapter 2, background on the FIR filter is reviewed and some evaluation metrics we will use are defined. In Chapter 3, the implementation of the time domain FIR filter is described and different structures of the filter are evaluated. Through comparing the speed, area and power, we choose the best structure to base our implementation on. Data type and scaling format are also discussed in this chapter.

Chapter 4 describes the hardware generator which produces synthesizable Verilog code for the TD FIR filter and the MATLAB-based simulation to evaluate filter accuracy. These two tools make implementation and evaluation easier. In Chapter 5, we use these tools to evaluate the time domain filters in terms of accuracy, area and speed, and we compare the results with frequency domain (FD) filters.

Lastly, Chapter 6 concludes this thesis.

Chapter 2

Background

2.1 Time-Domain (TD) FIR Filter

A length M discrete-time FIR filter computes output sequence y_n from input sequence x_n and a set of filter coefficients b_k, where $0 \leq k < M$, according to

$$y_n = \sum_{k=0}^{M-1} b_k x_{n-k}. \quad (2.1)$$

This is implemented by taking the M most recent input elements, scaling them by the corresponding b_k, and computing the sum. A FIR filter has some useful properties. When the coefficient sequence is symmetric, it is easy to design the FIR filter with linear phase. This property is desired for phase-sensitive applications. Further, a FIR filter is inherently stable because it has no feedback so error will not accumulate.

The FIR filter can be implemented using direct form or transposed form. The direct form FIR filter is shown in Figure 2.1. For a filter with M coefficients, the filter must add M values to produce a final output. As M increases, the structure of the adder may affect the speed and area of the whole time-domain FIR filter. To achieve high speed performance, we need to add extra pipeline registers in the adder and carefully design the structure of the adder. Figure 2.2 shows the figure of transposed form FIR filter. It can be constructed from the direct form FIR filter by exchanging the input and output and inverting the direction of signal flow. It has

registers between the adders and can achieve high throughput without adding any extra pipeline registers. But with the increasing of the number of filter coefficients, the latency will increase. To achieve high throughput and reduce latency, we choose to use direct form to implement the FIR filter and add pipeline registers in the adder. The structure of the adder will be discussed in Chapter 3.

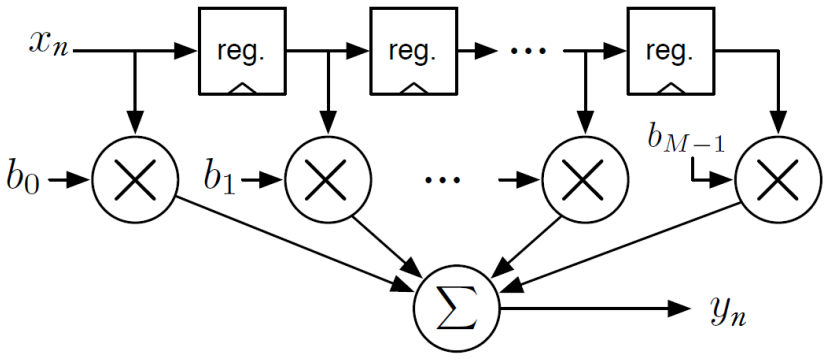

Figure 2.1 Direct Form FIR Filter

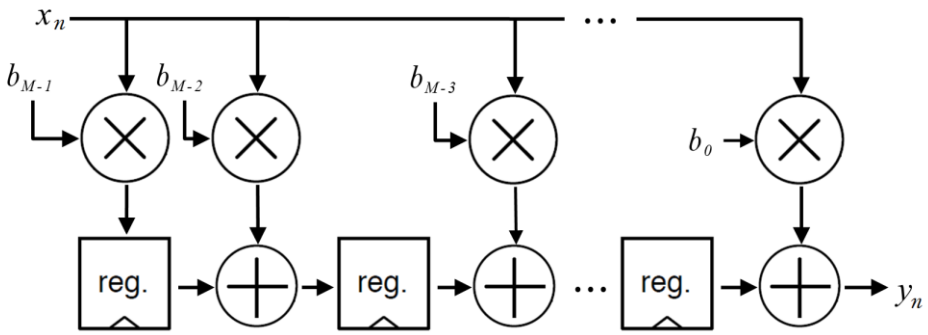

Figure 2.2 Transposed Form FIR Filter

2.2 Frequency-Domain (FD) FIR Filter

In the time domain, the FIR operation is calculated by convolution. Converting to the frequency domain, a convolution is equal to a simple multiplication. The frequency-domain (FD) design we compare with uses the “overlap save” technique [2] to compute the filter

values in the frequency domain. This is illustrated in Figure 2.3.

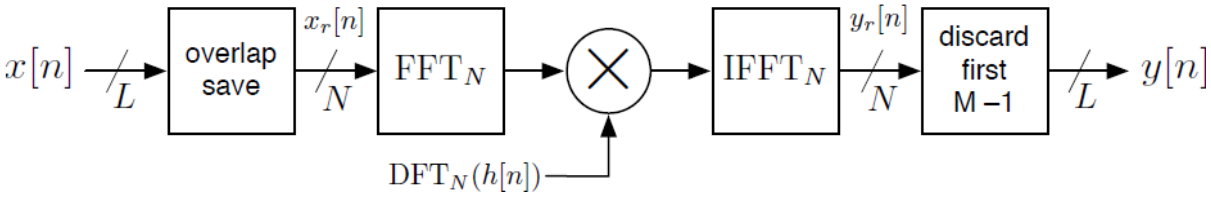

Figure 2.3 Frequency-Domain FIR Filter

2.3 Performance Evaluation Parameters

In Chapter 5, we will evaluate the performance of filters in terms of area, power, speed and error. In Section 2.3.1 – 2.3.4 respectively, we will discuss each of these cost and performance metrics.

2.3.1 Area

In this paper, we implement our designs using Xilinx FPGAs. An FPGA architecture is made up of separate columns of different dedicated hardware resources. This includes clocking resources, DSP slices, block RAMs, CLBs (configurable logic blocks) and I/O resources.

The primary unit of combinational logic of the FPGA is the CLB. Each CLB contains logic units called slices. (In Xilinx 7-Series FPGAs, each CLB contains two slices.) Each slice has four 6-input look-up tables (LUTs) which can also be split into two 5-input LUTs, allowing the slice to be used for two simple logic functions or one more complex function. There are also two flip-flops associated with each LUT, which makes pipelining convenient and fast. There are also dedicated multiplexers which can save LUTs and improve the system speed. Lastly, the slices include a carry chain for implementing fast arithmetic addition and

subtraction [20]. Figure 2.4 shows the carry out is propagated vertically through the four LUTs in a slice. The carry chain propagates from one slice to the slice in the same column in the CLB above. The number of occupied slices is a very important metric for evaluating the area consumed by a design.

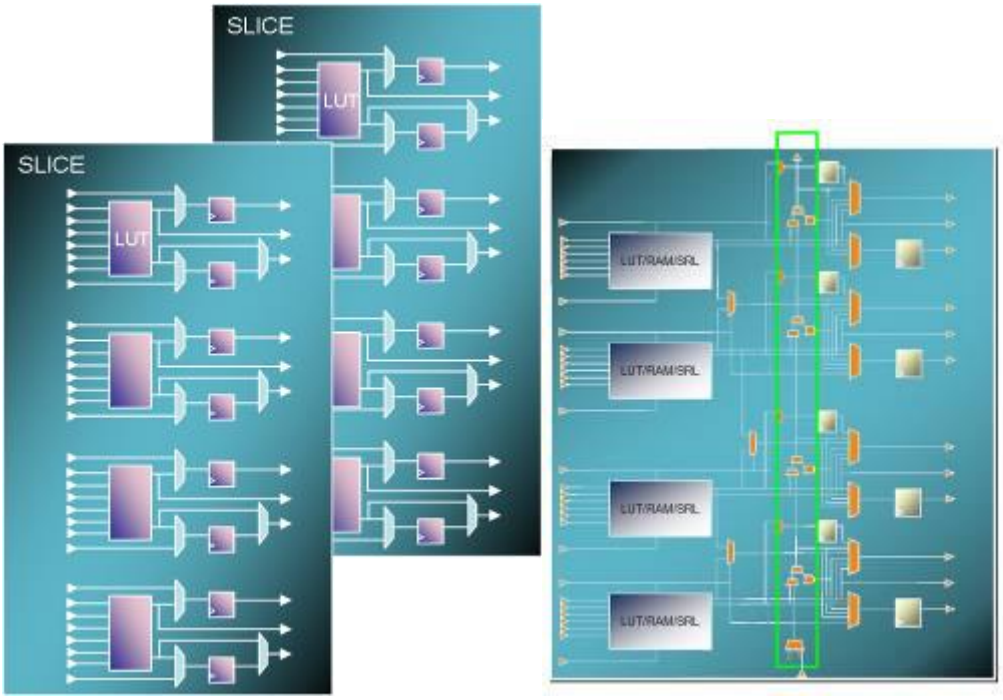

Figure 2.4 CLB Structure and Carry Chain (from [20])

DSP slices are hard arithmetic structures built in to the FPGA, which are ideal for performing certain types of DSP operations. DSP slices can process a large number of mathematical operations repeatedly and quickly on a set of data with low cost, low latency and high performance. Xilinx 7-Series FPGAs contain DSP slices called DSP48E1, which are shown in Figure 2.5. The DSP48E1 slice supports many independent functions, including multiply, multiply accumulate (MACC), multiply add, three-input add, barrel shift, wide-bus multiplexing, magnitude comparator, bit-wise logic functions, pattern detect, and wide counter. The architecture also supports cascading multiple DSP48E1 slices to form wide math

functions, DSP filters, and complex arithmetic without the use of general FPGA logic. The number of DSP48E1 slices used by a design is another important area metric that we will use when evaluating our designs.

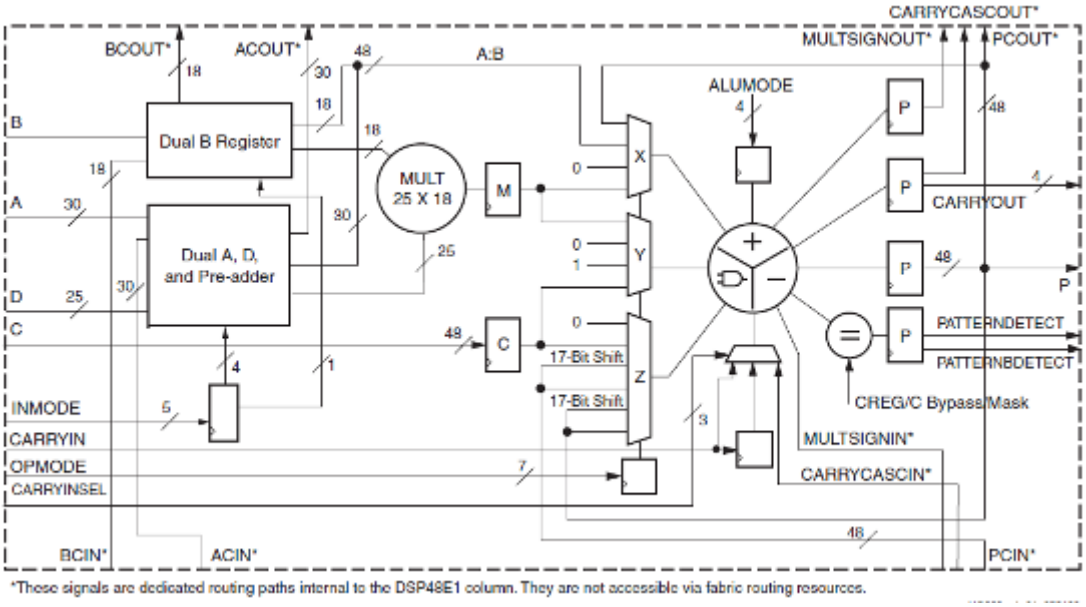

Figure 2.5 FPGA DSP48E1 Slice (from [21])

2.3.2 Power

Power dissipation has two components: dynamic and static. We will consider both parts in our evaluation of the power consumption of our designs.

Dynamic power is dissipated due to the short-circuit current and the switching capacitance. It can be calculated by aCV^2f, where a is the switching activity factor, C is the capacitance being charged or discharged per clock cycle, V is voltage and f is the switching frequency. The short-circuit current is caused by both pMOS and nMOS stacks being partially on.

The static power dissipated comes from leakage and contention current. The source of

the leakage includes subthreshold leakage, gate leakage and junction leakage.

2.3.3 Speed

We use two metrics to evaluate the speed of a design. The first is minimum clock period which is determined by the critical path of the design. We can often optimize it by adding pipeline registers. The second metric is throughput. We define the throughput of the system by

$$\text{throughput} = parallelism \times \max frequency \tag{2.2}$$

Our metric for parallelism (defined later) is equivalent to the number of concurrent input samples per clock cycle. When we report throughput, we determine clock frequency after synthesizing, placing, and routing the design. In Chapter 5, we report throughput in gigasamples per second.

2.3.4 Error

Our hardware designs will use a fixed-point number representation, where the accuracy of the number is related to the number of bits used. It is important to know the relationship between the length of each word and its result on the error of the filter. We use mean squared error (MSE) to quantify error, defined as:

$$\text{MSE} = \frac{1}{N}\sum_{l=1}^{N}\left(X_l - \hat{X}_l\right)^2 \tag{2.3}$$

where X is the fixed-point filter output and $\hat{X}$ is the ideal filter output.

Chapter 3

Design of FIR Filter

In this chapter, we first compare two adder structures: adder-tree and adder-cascade. Then we extend the more-efficient design and use it as our baseline in the following chapters.

3.1 Comparison of Two Adder Structures

Because the adder structures considered here are used in a variety of different types of FIR filters, we must consider variations on the design in terms of the size of the filter, its parallelism, and the data representation used. So, we evaluate the cost and performance of the FIR filters using two adder structures across the following parameters:

a) The number of coefficients of the FIR filter, namely M in the formula (2.1). M is constrained to be a power of 2 in this initial investigation, but later we will allow this to be an arbitrary integer.

b) The number of stages of parallelism w, which is the number of inputs that can go into the system at the same time. This also should be the power of 2 here and no greater than M. Larger values of w lead to designs with higher area cost but higher throughput. Later we expand this parameter to be an arbitrary integer.

c) The number of bits that are used to represent the two's complement fixed-point data. To avoid overflow, we increase this value as needed after each operation. For example, if two B-bit numbers are multiplied, the output requires $2B$ bits. If two B-bit numbers are added,

B+1 bits are required. Later we will also consider scaling the representation to avoid increasing the number of bits needed.

3.1.1 Adder Tree Structure

In typical direct form FIR filters, the input stream is presented to one multiplier input while the coefficients supply the other input. Then an adder tree is used to combine the outputs from all multipliers [21]. As shown in Figure 3.1, the unpipelined adder tree structure has registers only at the input and output of the final adder. The rectangles in the figure represent registers. The pipelined adder tree structure has pipeline registers between different stages of the adder, which is shown in Figure 3.2.

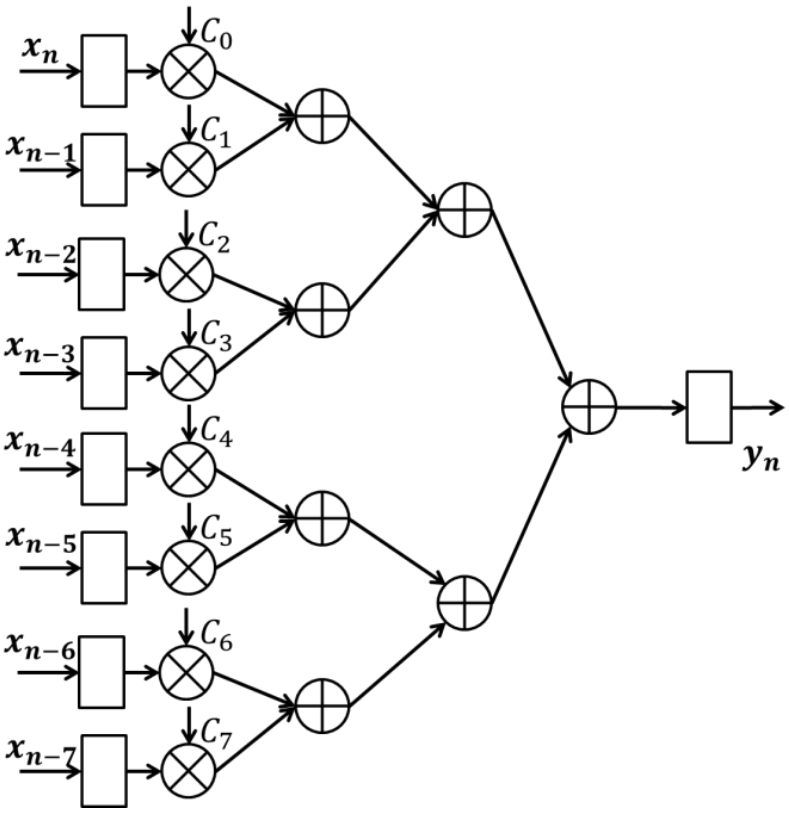

Figure 3.1 Unpipelined Adder Tree Structure

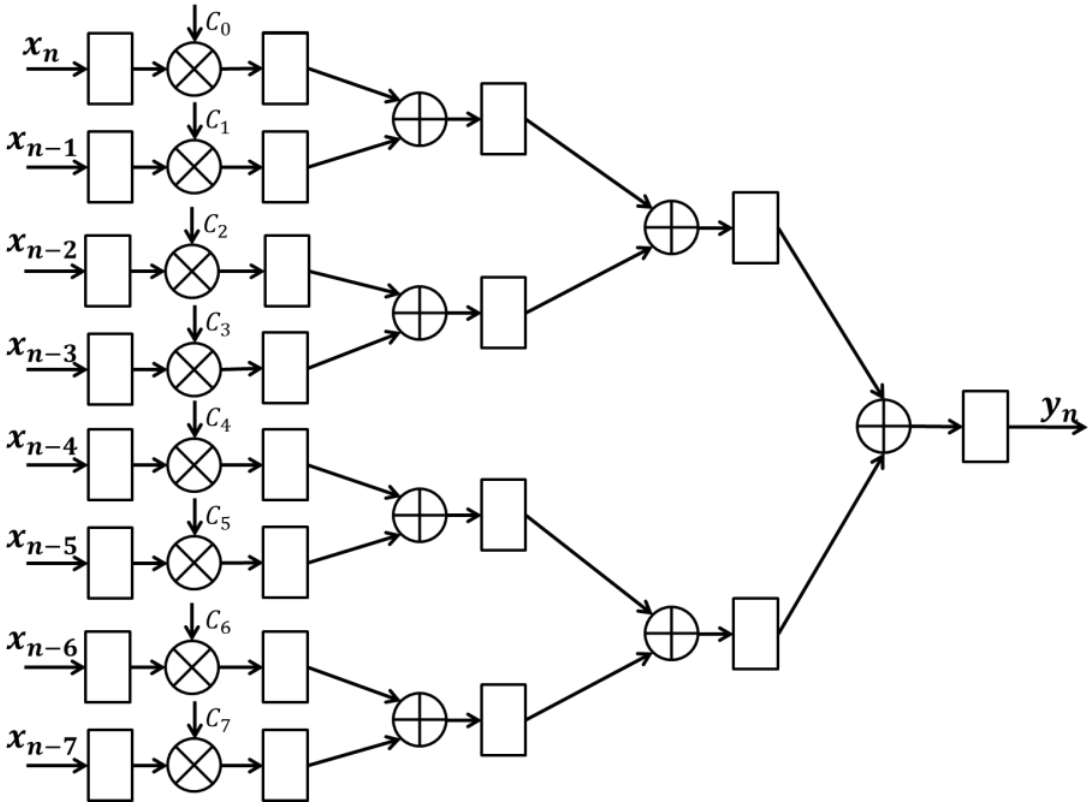

Figure 3.2 Adder Tree with pipelining

3.1.2 Adder Cascade Structure

The DSP48E1 slice has a multiplier and an adder, and it also supports cascading multiple DSP48E1 slices to form DSP filter. The adder cascade structure, shown in Figure 3.3, is designed to take advantage of this. By utilizing the cascade path of the DSP48E1 slices, the adder cascade structure efficiently combines a multiplication and an addition into a single DSP slice. The first five stages use a cascade structure, and after the first five stages, the pipelined adder tree structure is used. An example with eight coefficients is shown in Figure 3.3. The basic unit is sized to take four inputs and use five stages of pipeline. If we increase the size of this basic unit, latency will increase greatly with little changes in the resources required. In this work, this combination cascade and tree structure is called adder cascade

structure for short.

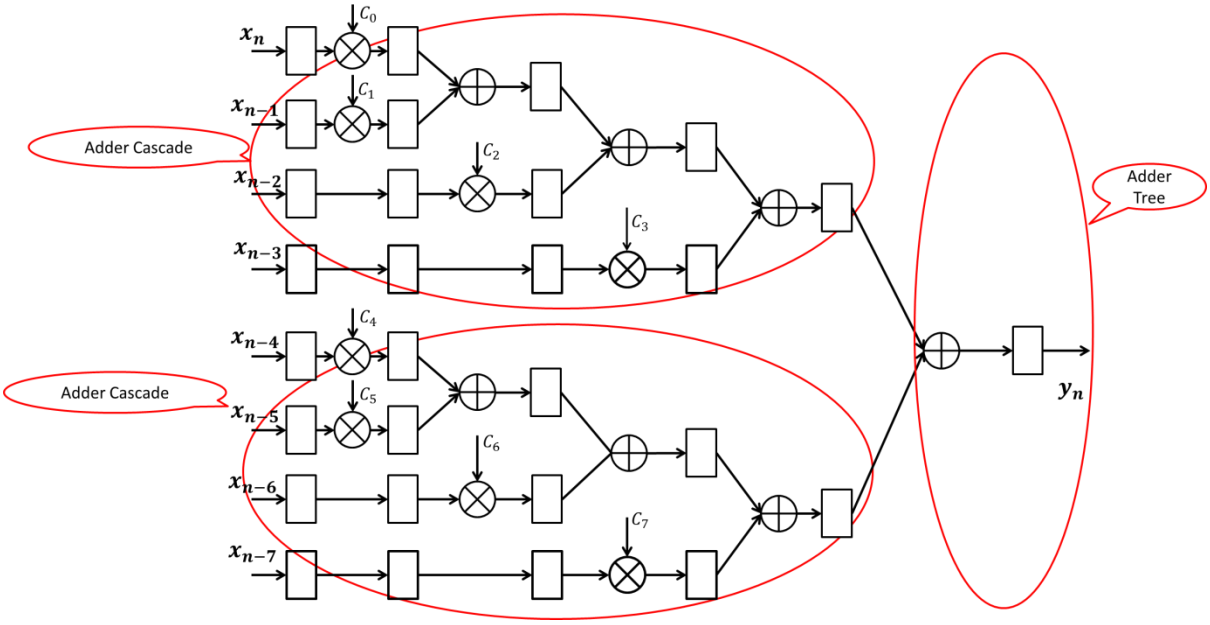

Figure 3.3 Adder Cascade

3.1.3 Performance Comparison

Several instances are simulated and synthesized. They are first simulated behaviorally to check whether the function of the Verilog code is correct. After this, the instances are synthesized, placed and routed. From the place and route report, we can obtain the area of each design instance. From the post-place-and-route static timing report, the clock frequency is obtained. To get the power information and increase the confidence level of the power data, post-route simulation is performed and an SAIF (switching activity interchange format) file is generated. This contains the number of changes on all signals in the design. Then the Xilinx XPower Analyzer is used to estimate dynamic and static power. Because dynamic power depends on clock frequency, each design is set to its maximum frequency when the power is analyzed. Static power is largely determined by the size of FPGA, so two FPGAs with different

sizes are considered to quantify this. The Virtex-6 XC6VLX75T is much bigger than the Spartan-6 XC6SLX9 FPGA, so we expect much higher static power for the Virtex-6 implementation.

There are three parameters we can change to observe the trend of the speed, area and power. They are parallelism, word length and the number of coefficients. Three experiments are designed to find the relationship between the parameters and the performance, and also to compare the performance of the two structures as described above.

The first experiment is to change the number of coefficients i and observe the changes in clock frequency, area and power of each structure while keeping the parallelism and the length of each word constant.

The second experiment is to keep the length of each word and number of coefficients constant, and change the parallelism. Then, we observe the clock frequency, area and power of each FIR structure.

The third experiment is to change the length of the word while keeping the other parameters constant.

3.1.3.1 Verification

For verification, we generate a given number of random numbers within a user-specified range, and use them in a testbench for behavioral simulation. At the same time, we compute the expected simulation results given the random inputs. After behavioral simulation and post-route simulation, the results are saved in another two files. A comparison program written in C can compare whether the behavioral and post-route simulation results are the same as the

expected results. In this way the correctness of the Verilog code can be tested.

3.1.3.2 Cost and Performance as the Number of Coefficients Changes

In the first experiment, we choose parallelism w=1, 16 bits data length, and the number of coefficients M from 4 to 64. Then we observe the clock frequency, area and power using the procedure described above. For the unpipelined adder tree structure, when the number of coefficient reaches 64, the Xilinx tool is unable to complete synthesis. Table 3.1 shows the results.

In Table 3.1, we observe that the clock frequency of the adder cascade and pipelined adder tree are stable, because they both have pipeline registers that prevent the critical path from growing when the number of coefficients increases. The clock frequency of the unpipelined adder tree structure decreases with the increasing number of coefficients because without pipeline registers, the critical path grows roughly proportionally to the logarithm of the number of coefficients.

Table 3.1 Speed and area versus the number of coefficients

No. of Coefficients		4	8	16	32	64
Adder cascade	**Clock frequency (MHz)**	355.998 (2.809ns)	355.998 (2.809ns)	355.998 (2.809ns)	355.998 (2.809ns)	355.998 (2.809ns)
	Area(Slices)	2	22	50	116	274
	DSP48E1	4	8	16	32	64
Unpipelined adder tree	**Clock frequency (MHz)**	150.399 (6.649ns)	85.572 (11.686ns)	45.254 (21.952ns)	23.881 (41.873ns)	--
	Area (Slices)	7	19	38	76	
	DSP48E1	4	8	16	30	
Pipelined adder tree	**Clock frequency (MHZ)**	355.998 (2.809ns)	355.998 (2.809ns)	355.998 (2.809ns)	355.998 (2.809ns)	355.998 (2.809ns)
	Area(Slices)	17	48	94	203	518
	DSP48E1	4	8	16	32	64

It can also be obtained from Table 3.1 that the area increases with the increasing number of coefficients. Further, the pipelined adder tree structure needs a larger area than the adder cascade structure for the same number of coefficients, which is shown in Figure 3.4.

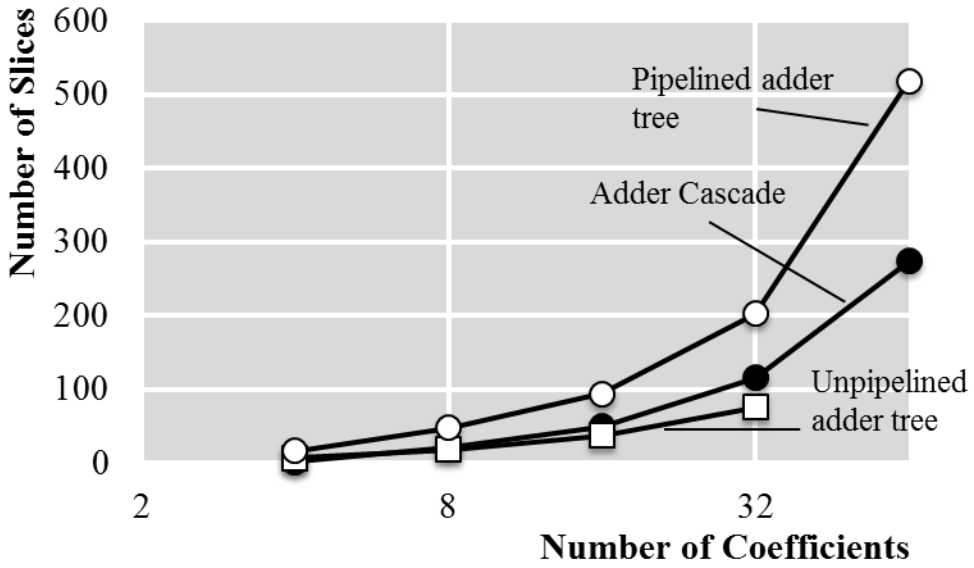

Figure 3.4 Area vs. Number of Coefficients

Because the area and speed performance of the unpipelined adder tree is far worse than the other options, only the other two structures are compared in terms of power. The results are shown in Table 3.2. As expected, the static power of the design on Virtex-6 FPGA is much higher than that on Spartan-6 FPGA, because of the Virtex-6's large area.

Table 3.2 Power versus number of coefficients

No. of Coefficients			**4**	**16**
Adder Cascade	**Power (W)** Virtex	Dynamic	0.215	0.383
		Static	1.298	1.301
	Power (W) Spartan	Dynamic	0.190	0.396
		Static	0.032	0.036
Pipelined Adder Tree	**Power (W)** Virtex	Dynamic	0.185	0.213
		Static	1.297	1.297
	Power (W) Spartan	Dynamic	0.178	0.220
		Static	0.032	0.032

3.1.3.3 Cost and Performance as the Parallelism Changes

In the second experiment, each word is 16 bits and there are 16 coefficients for the system. As the parallelism changes from 1 to 4, the results obtained are shown in Table 3.3.

From the table, we can see that when the parallelism is increasing, the clock frequency does not change for all three structures. This occurs because the parallel designs area built by duplicating modules without changing their internal structure or timing.

Table 3.3 Speed and area versus parallelism

No. of Parallelism		1	2	4
Adder cascade	**Clock frequency**	355.998MHz 2.809ns	355.998MHz 2.809ns	355.998MHz 2.809ns
	Area(Slices)	50	99	171
	DSP48E1	16	32	64
Adder tree without pipeline	**Clock frequency**	45.254MHz 21.952ns	45.254MHz 21.952ns	45.254MHz 21.952ns
	Area(Slices)	38	35	32
	DSP48E1	16	32	64
Pipelined adder tree	**Clock frequency**	355.998MHz 2.809ns	355.998MHz 2.809ns	355.998MHz 2.809ns
	Area(Slices)	94	165	280
	DSP48E1	16	32	64

The result of increasing of parallelism is increasing throughput. Figure 3.5 shows the relationship between area and throughput. The area of both the adder cascade structure and pipelined structure are increasing with the increasing of the throughput, and the adder cascade structure uses less area than pipelined adder tree structure. The area of the unpipelined adder tree structure becomes smaller with the increasing of the throughput. One explanation for this is that the synthesis tool reuses some modules in the system or does some optimization when the parallelism increases, so the area becomes smaller.

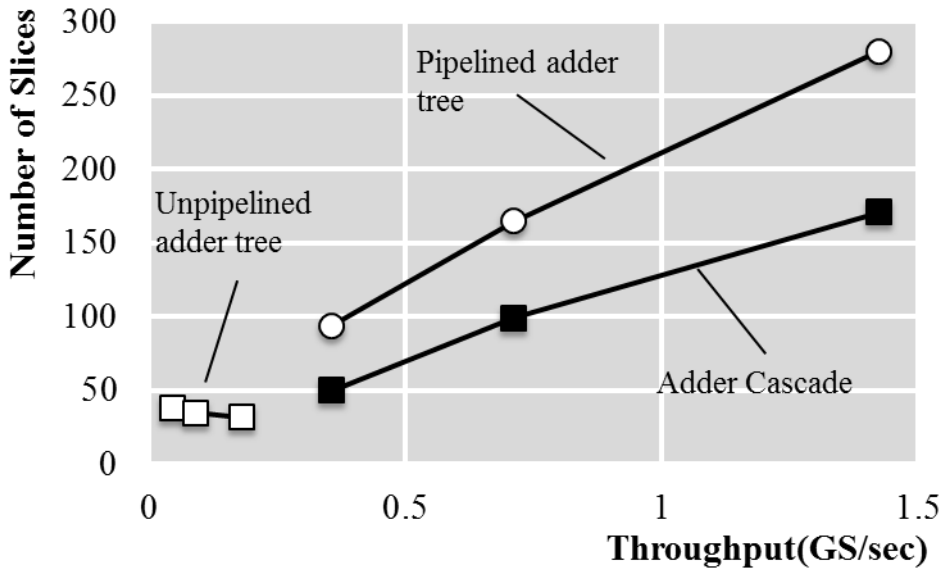

Figure 3.5 Area vs. Throughput

Table 3.4 shows the data for power simulation as the number of coefficients changes. The power for the design with parallelism 4 on Spartan-6 FPGA is not shown because the design is too large to fit in the small FPGA. From the power data, we see that designs with smaller area result in reduced static power.

Table 3.4 Power versus number of coefficients

No. of Parallelism			1	4
Adder Cascade	**Power** Virtex	Dynamic	0.383	0.257
		Static	1.301	1.298
	Power Spartan	Dynamic	0.396	--
		Static	0.036	--
Pipelined Adder Tree	**Power** Virtex	Dynamic	0.213	1.315
		Static	1.297	1.323
	Power Spartan	Dynamic	0.220	--
		Static	0.032	--

3.1.3.4 Cost and Performance as Word Length Changes

For the third experiment, the length of each word is changed from 8 bits to 24 bits while the other parameters stay the same. The number of coefficients is 16 and the parallelism is 1.

The results are shown in Table 3.5 and Table 3.6.

Because the speed of the unpipelined adder tree structure is so slow, we focus on the other two structures. When the length of each word is 20 bits, the speed of both of the two structures decreases and the number of slices increases greatly, but the number of DSP slices remains the same. This is because more resources on the FPGA other than DSP slices are used when the 20-bit calculation is performed, which results in longer critical path. When the length of each word reaches 24 bits, although fewer slices are occupied, more DSP slices are used (since each DSP slices includes a 18×25 bit multiplier). The result is that the critical path becomes shorter and the speed gets higher. Although the area fluctuates at 20 bits and 24 bits of the word length, the general trend is that the area will increase with the increasing of the word length.

Table 3.5 Speed and area versus word length

Length of each Word		**8**	**12**	**16**	**20**	**24**	**28**	**32**
Adder cascade	**Clock frequency (MHz)**	355.998	355.998	355.998	237.248	240.558	179.243	146.499
	Area(Slices)	32	46	50	356	339	326	477
	DSP48E1	16	16	16	16	26	32	62
Adder tree without pipeline	**Clock frequency (MHz)**	45.254	45.254	45.254	60.617	61.323	58.799	109.230
	Area(Slices)	24	31	38	224	164	213	263
	DSP48E1	16	16	16	16	32	32	34
Pipelined adder tree	**Clock frequency (MHz)**	355.998	355.998	355.998	241.429	247.158	181.951	179.019
	Area(Slices)	57	71	94	370	345	452	403
	DSP48E1	16	16	16	16	32	32	40

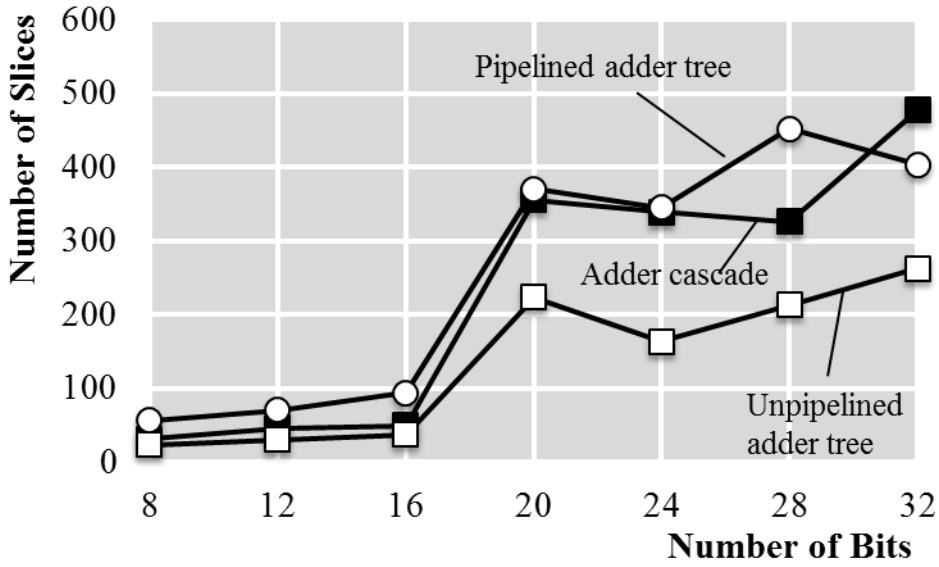

Figure 3.6 Area vs. Number of Bits

From the static power data shown in the Table 3.6, it also can be concluded that when the area increases, the static power will increase.

Table 3.6 Power versus word length

Length of each Word			**8**	**16**	**24**	**32**
Adder Cascade	**Power** Virtex	Dynamic	0.206	0.383	0.408	0.395
		Static	1.297	1.301	1.302	1.302
	Power Spartan	Dynamic	0.210	0.396	0.229	--
		Static	0.032	0.036	0.033	--
Pipelined Adder Tree	**Power** Virtex	Dynamic	0.075	0.213	0.481	0.451
		Static	1.294	1.297	1.304	1.303
	Power Spartan	Dynamic	0.075	0.220	0.426	--
		Static	0.030	0.032	0.037	--

3.1.4 Summary

In this section, we have compared the performance of time-domain FIR filters using three different adder structures. In terms of power, the static power will increase with the increasing of the FPGA size. In terms of area and speed, the unpipelined adder tree structure

uses the least area but is very slow. Both the adder cascade and pipelined adder tree structure have good speed performance. To achieve the same speed, the adder cascade structure uses less area than pipelined adder tree structure. Therefore, the adder cascade structure is the best among the three. Next, we will build upon this structure in our further work and evaluation.

3.2 Design Expansion

In order to make the time-domain FIR filter more general and apply to a wider space of applications, we now expand the design space to increase its flexibility.

3.2.1 Expansion of the Number of Coefficients

The basic unit in the adder cascade structure is a four-input adder cascade unit, as shown in the Figure 3.3. It has five pipeline stages. But when the number of coefficients is not multiple of 4, there will be fewer inputs and stages in the last unit. For example, Figure 3.7 shows the structure of the filter with 11 coefficients. There are 3 basic units in this structure. The last unit has only three inputs. But the number of pipeline stages keeps the same. When the outputs from the three basic units are added together using adder tree, the first two outputs are added first, then added to the third. To guarantee the timing sequence is correct, one more pipeline register is added after the third unit.

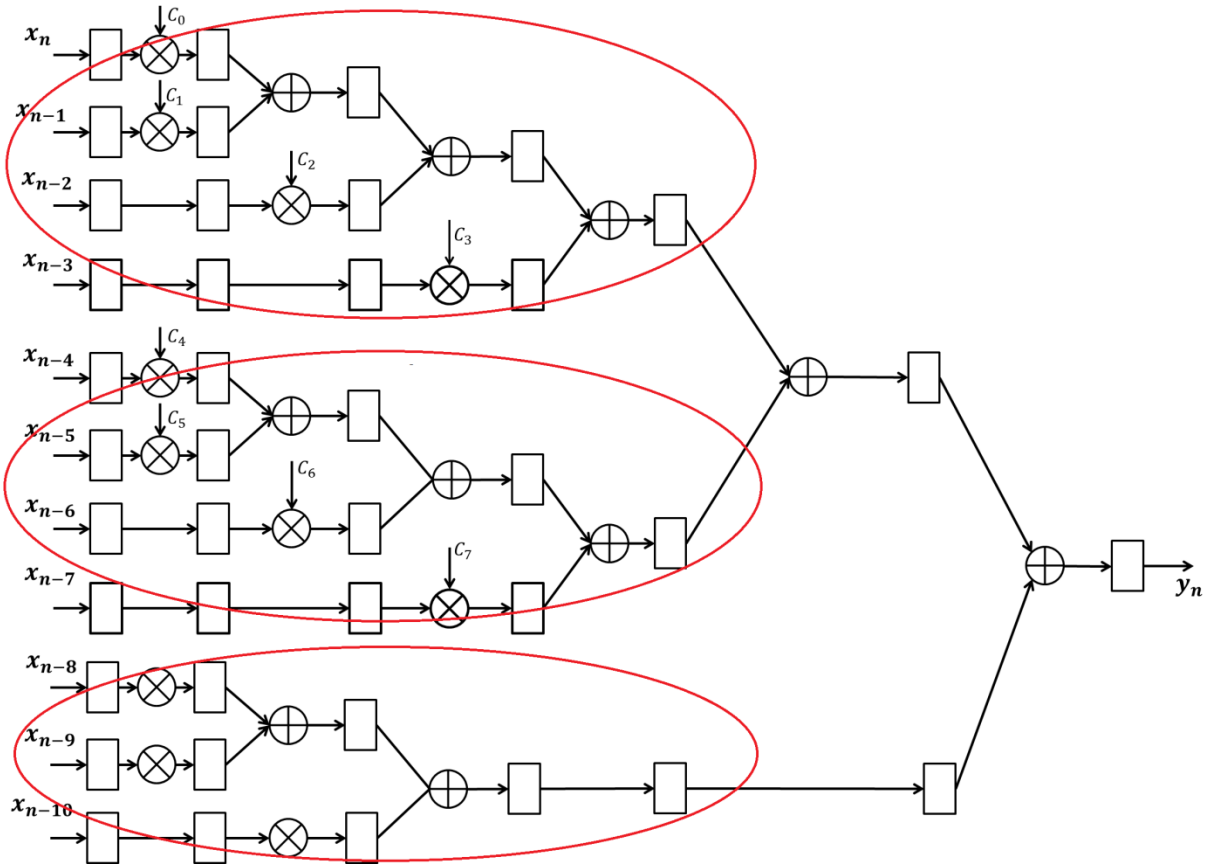

Figure 3.7 Cascade Structure of FIR filter with 11 coefficients

3.2.2 From Real to Complex

Many applications (such as communications systems) require complex inputs and complex coefficients. To enable this, we must allow the multipliers and adders to perform complex arithmetic.

For the adder, one real adder is changed to two adders, one for the real part and the other for the imaginary part. For the multiplier, one real multiplier is changed to four multipliers and two adders to perform the complex multiplication. The Figure 3.8 shows the structure of the complex multiplier. It performs the complex multiplication

$$(a + bi) \times (c + di) = (ac - bd) + (bc + ad)i$$

Two pipeline stages are added (shown as rectangles in the figure) to maintain consistent clock

frequency.

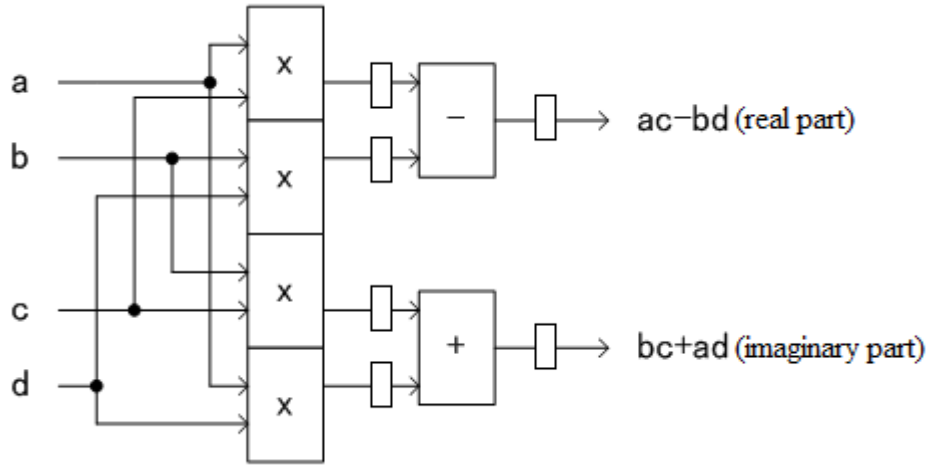

Figure 3.8 Complex Multiply [22]

3.2.3 Scaling Format of Multipliers and Adders

When arithmetic is performed on fixed point data, the maximum possible value of each word grows. In order to prevent data from overflowing (requiring more bits than are available to represent the result), we can either increase the word length of the result or scale data down to fit within the original number of bits. The previous experiments (Section 3.1) utilized the former technique to avoid overflow. However when the number of coefficients becomes large, the length of the output can grow very long. Here we scale the data to avoid overflow and also avoid increasing the length of the output. We make each input and output port of the adder and multiplier the same length, and scale the data to fit. Although precision will decrease because of this scaling, the resources used will decrease, and we can minimize the error by choosing the appropriate scaling format of the multipliers and adders.

In the following chapters, we use scaled arithmetic, that is, after adding or multiplying two *B*-bit values, we keep the most significant *B* bits. Improvements could potentially be

made by using dynamic or other more complicated scaling methods [23], but the advantage of using these methods depends highly on the application.

To make the hardware fixed-point results as accurate as possible under this specific scaling format, the coefficients should be scaled to be as large as possible. Ideally, the designer can choose to scale the coefficients C_i to $C_i/\max(C_i)$, and then filter the inputs using these scaled values. After filtering, the results can then be multiplied by $\max(C_i)$ to restore the original results. (Often this multiplication can be incorporated into the next computational stage after the filter.) The benefit of this technique is to let the coefficients be as large as possible to make full use of the bits available.

3.2.4 Increasing the Parallelism

Previously, we restricted parallelism to be a power of 2. Here we increase flexibility in cost and throughput by allowing it to be any positive integer. To support the parallel structure with streaming width *w*, we replicate the cascade structure *w* times, and for each instance we shift the inputs by appropriate amount. Figure 3.9 gives an example of the structure with 5 coefficients and the parallelism 3. Three outputs are generated at each clock cycle.

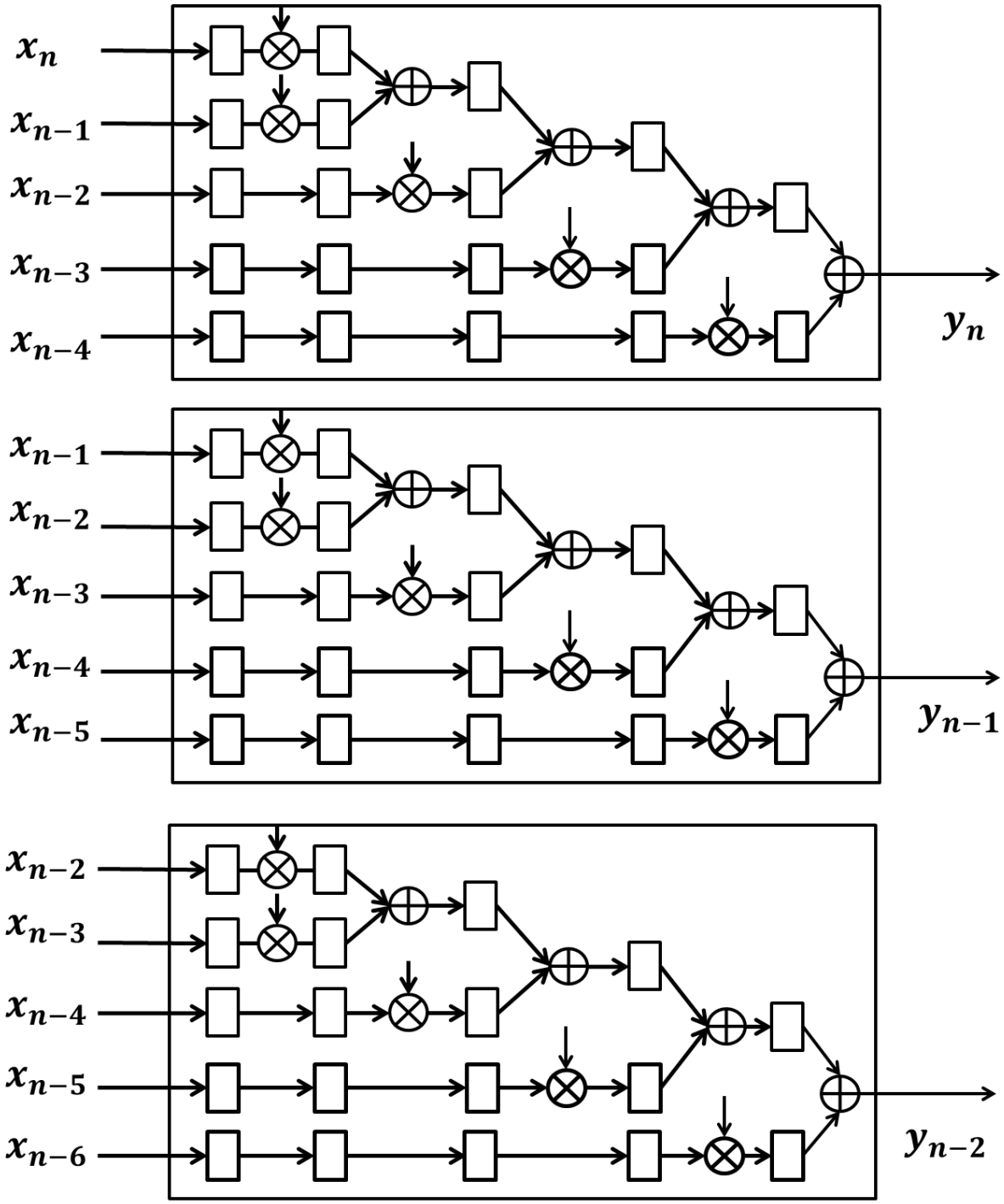

Figure 3.9 The structure of the system with 5 coefficients and 3 levels of parallelism

In the next chapter, we will describe how this flexible design space is implemented in a hardware generation and simulation framework.

Chapter 4
Hardware Generation and Simulation Platform

In this chapter, we build a flexible hardware generation tool using C and a fixed-point simulation environment using MATLAB. The hardware generation tool can generate synthesizable Verilog based on the user's specification. The MATLAB simulation environment is to verify the generated Verilog code and calculate the mean squared error (MSE). Further, a decimal-to-integer converter is built to convert the data to proper format and the hardware generation tool uses these data as inputs.

4.1 Converter

We assume the coefficients and input of the filter are all decimals in the interval (-1, 1). We use B bits to represent each number. We fix the radix point to be one bit to the right of the most significant bit, and the other B-1 bits represent the fractional part of the decimals. If we remove the decimal point, the B-bit number can be viewed as an integer. The converter represents decimals in (-1, 1) as B-bit integers, allowing the hardware to operate on integer representations of the data. For example, if we want to use 6 bits to represent a decimal number 0.632359246225410, it is 0.10100. Removing the decimal point gives 010100, which is 20. If we use 10 bits to represent the same decimal number, it is 0.101000011. Converting

it to integer, it is 323.

The input of the converter is a list of coefficients, the input of the FIR filter and the length of each word B. The output is the set of coefficients and inputs as integers.

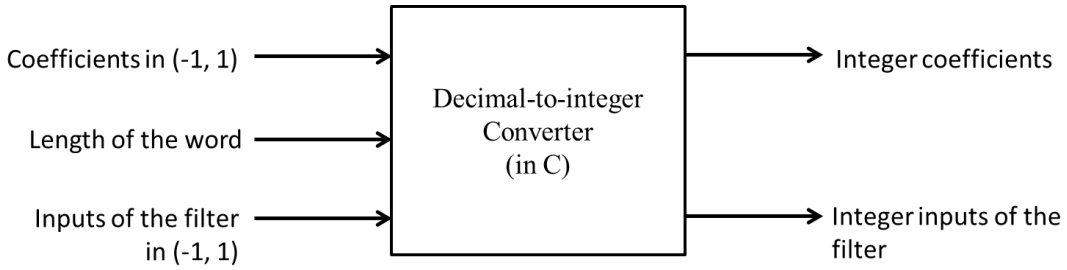

Figure 4.1 The converter schematic

4.2 Generator for FIR Filter

To automatically produce Verilog code for each design instance, a generator for the FIR filter is developed using C code. The input of the generation tool includes:

a) The number of coefficients of the filter (any positive integer greater than 2).

b) The filter coefficients, which have been scaled to integer representation by the converter (see Section 4.1).

c) The length of each word B, which should be positive integer.

d) The number of stages of parallelism *w*, which is the number of inputs that can go into the system at the same time. The *w* should be any positive integer no greater than the number of coefficients.

e) A scaling parameter that controls whether the adders and multipliers should be scaled or unscaled (defined in Section 3.2.3).

f) The input values for the test bench to use.

The output of the generator is synthesizable Verilog code for the TD FIR filter and testbench based on the user's specifications.

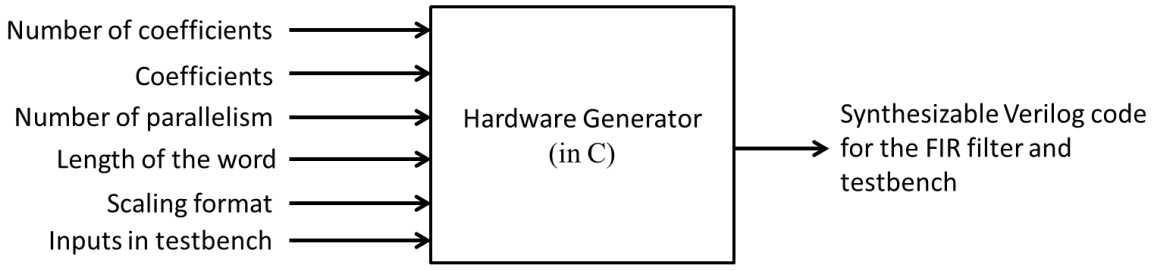

Figure 4.2 Hardware generator inputs and outputs

4.3 MATLAB Simulator

By allowing control of the number of bits in the fixed-point data format used, the generator allows the user to specify a tradeoff between cost and accuracy. However, it can be difficult for users to understand how their choice of the number of bits will affect the overall accuracy of the computation. To measure this, we construct a fixed-point simulation environment using MATLAB. This simulation takes the input data, the coefficients and number of bits as its input. The model then simulates the filter using the same fixed-point options as the hardware filter, and outputs the same results as the hardware simulation. This has two purposes. First, we use this to verify that our hardware realization was produced correctly. Second, it enables easy comparison between the fixed-point results and MATLAB's double-precision convolution. We quantify error in terms of mean squared error (MSE) (defined in Section 2.2).

4.4 Summary of Platform and Simulation Procedure

In this chapter, we described a generation tool and a simulation environment which allow a designer to easily understand the cost and performance implications of the various options available, and to enable automatic generation of the implementation. First, the automatic hardware generator produces synthesizable Verilog code for a FIR filter and a simulation testbench based on the user's specification. Second, the MATLAB-based simulation platform allows easy bit-exact simulation of the generated hardware system and compares the results with the double precision convolution results to obtain the error.

When we perform an experiment, we can give the two platforms the same set of input data and coefficients, simulate the hardware design and compare the simulation results with the MATLAB results to check the correctness of the design. Then we synthesize, place, and route the design to get the timing and resource utilization information. At the same time, simulations run in MATLAB allow us to measure accuracy and error.

Chapter 5

Evaluation

In this chapter, we evaluate the performance of the TD filter in terms of accuracy, speed and area, and compare it with another implementation method using frequency domain (FD) methods (as defined in Section 2.2). For each set of experiments, we first use the generator to generate Verilog code of the TD filter based on the specification, and simulate it using Xilinx ISE Design Suite. Then we put the same coefficient and input data into the MATLAB simulator to check the correctness and calculate the MSE. After making sure the Verilog code is correct based on our specification, we synthesize and place and route the design on a Xilinx Virtex-7 FPGA (xc7vx980t) to get the timing and resource utilization information. Then, we compare with data for the FD filter from [24].

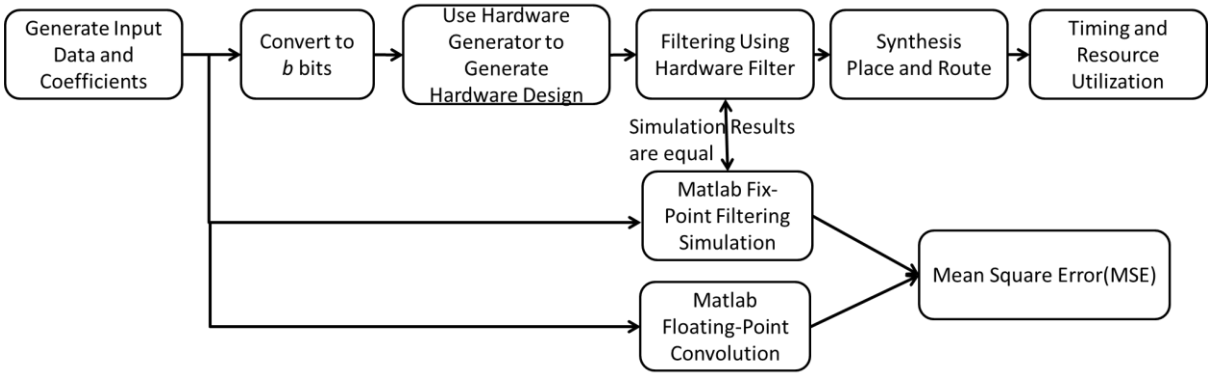

Figure 5.1 Summary of Procedure

5.1 Error Measurement

To quantify the error of fixed-point hardware implementations of TD FIR filters, we use

the MATLAB simulation environment described in Chapter 4.3. We simulate two different types of coefficients: (1) random; and (2) root raised cosine (RRC) filter used in communication systems (e.g. [3]) with fixed point data format from 8 to 32 bits. For each, we simulate designs with 31, 127, and 511 coefficients. For the designs with 31, 127, and 511 coefficients, 100, 500, and 1000 random complex input data respectively are used during simulation. We choose the number of coefficients to be odd in order to keep the RRC filter symmetric. For RRC coefficients, we average the error over 10 sets of the random complex input data for each design. For the others, we average the error over 10 different random sets of coefficients and random input data for the random coefficients.

Figure 5.2 to Figure 5.4 shows the mean squared error for TD (white marker) and FD (grey marker) filter with 31, 127 and 511 random coefficients (squares) and RRC coefficients (triangles). The x-axis shows the number of bits used in the fixed-point format and the y-axis (log scale) shows the MSE.

The error of the TD design with RRC coefficients is always about 2 times smaller than the error of the same design with random coefficients because the RRC coefficients are purely real-valued while the random coefficients are complex. The FD designs have lower error with RRC coefficients than with random coefficients, because the FFT of the RRC coefficients has many values which round to 0 or 1, which increases the accuracy.

We can observe that for the design with 31 RRC coefficients, the FD design needs 1 or 2 more bits than the TD to get to the same error. When the number of coefficients increases to 127 and 511, the FD designs have roughly the same precision as TD designs. For the designs

with 31 random coefficients, the FD needs 2-3 more bits to get the same error as TD. With increasing of number of coefficients, the difference between TD and FD increases to 3 to 4 bits.

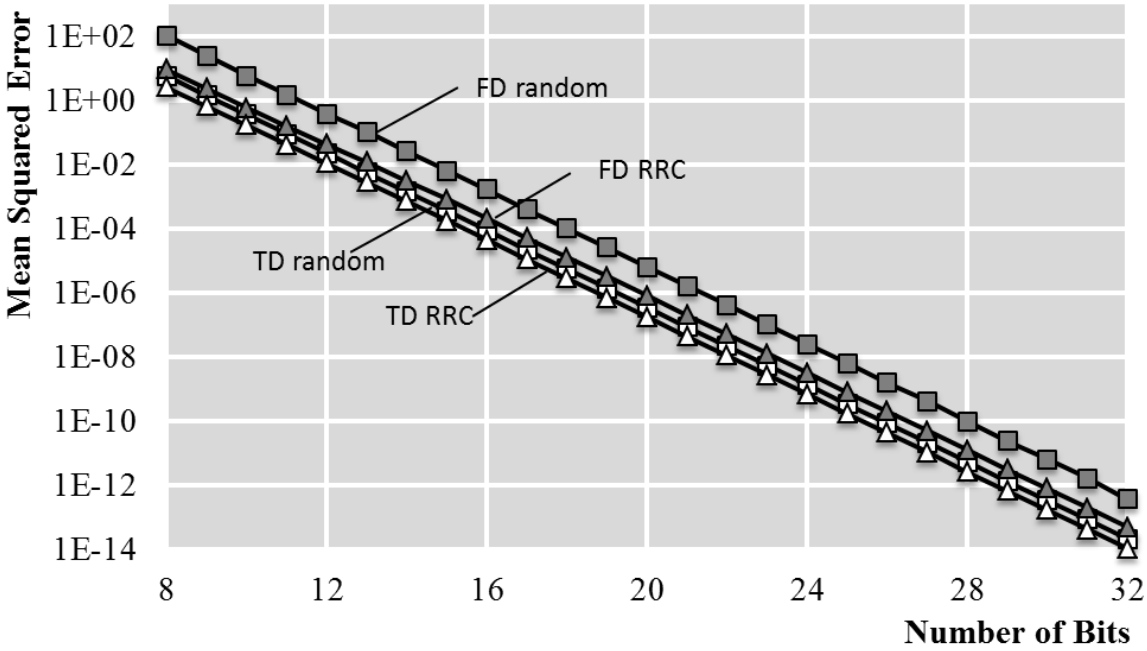

Figure 5.2 Mean squared error versus bits for TD (white) and FD (grey) with 31 random (squares) and RRC coefficients (triangles).

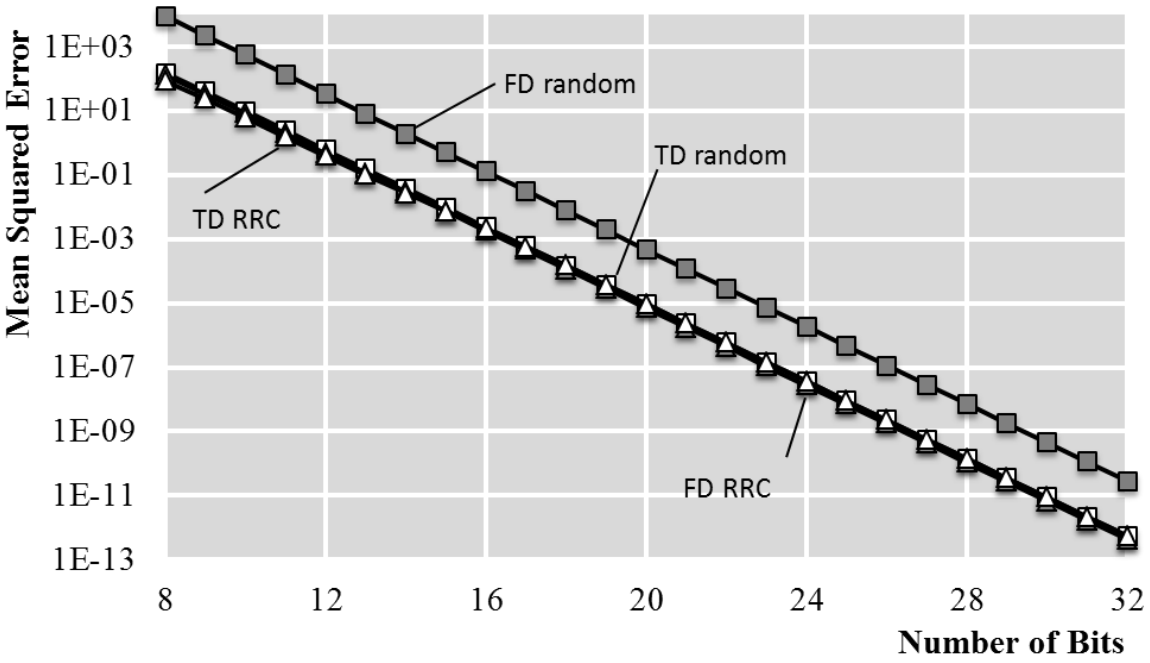

Figure 5.3 Mean squared error versus bits for TD (white) and FD (grey) with 127 random (squares) and RRC coefficients (triangles).

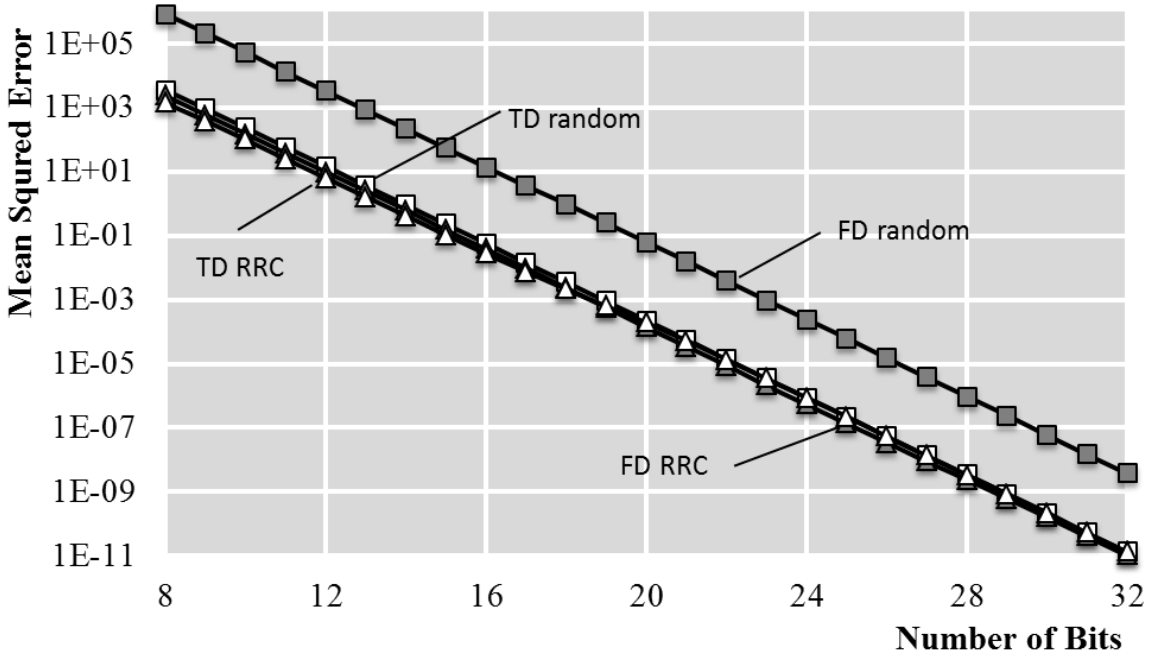

Figure 5.4 Mean squared error versus bits for TD (white) and FD (grey) with 511 random (squares) and RRC coefficients (triangles).

Next, we perform another experiment where we fix the number of bits to 20, increase the

number of coefficients from 11 to 91 with step size 10, and observe the trend of error for TD filter and compare it with FD filter. Random coefficients are used. The error is averaged over 10 different random sets of coefficients and random input.

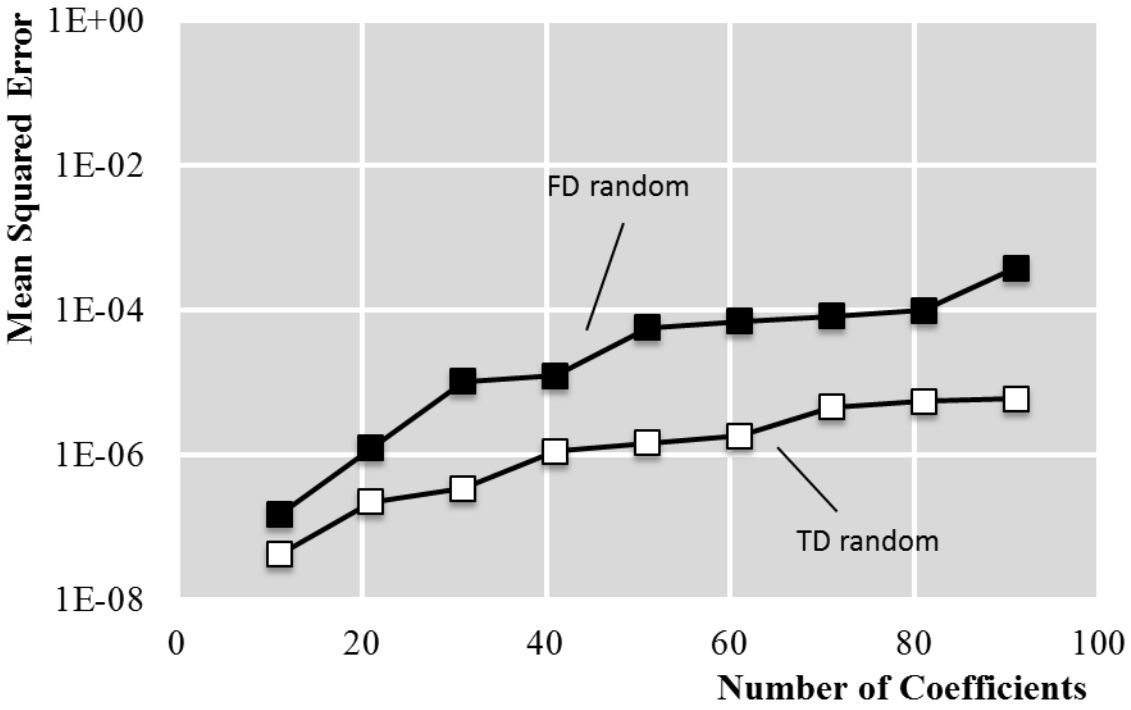

Figure 5.5 Mean squared error versus number of coefficients for TD (white) and FD (grey) with 20 bits

As shown in Figure 5.5, as the number of coefficients increases, error also increases. Also of interest is that the TD curve has apparent "steps," that is, its error increases more when the number of coefficients goes from 11 to 21, 31 to 41 and from 61 to 71 than the others. This is because of the tree-based adder structure shown in Figure 3.7. This structure with M coefficients will have $\lceil \log_2 M \rceil + 1$ adder stages, so each time the number of coefficients increases larger than a power of 2, a new stage is required. Adding this stage means that we need to scale the results one more time, which adds additional error. For

example, the designs with 7 or 8 coefficients have only four adder stages while the design with 9 coefficients needs a fifth stage. This is why the error increases more when the number of coefficients goes from 11 to 21, 31 to 41 and from 61 to 71 than the others.

From the discussion of the results, we can conclude that the error is affected by the number of coefficients, the number of bits used, and the values of the coefficients themselves. By using the provided MATLAB fixed-point simulation environment, a designer can easily find the relationship among error, number of bits and the coefficients for a specific application.

5.2 Area Measurement

Based on the results of the error analysis, the number of bits needed to produce an equal error level for TD and FD varies depending on the specific filter. To evaluate the area of TD and FD fairly, we choose the TD with 12-bit random coefficients, and compare it with FD filter with 12-bit and 16-bit random coefficients. We start the analysis by picking the parallelism w=2, and see how the area changes as the number of coefficients changes. Then, we repeat the experiment with parallelism w=8 and w=16.

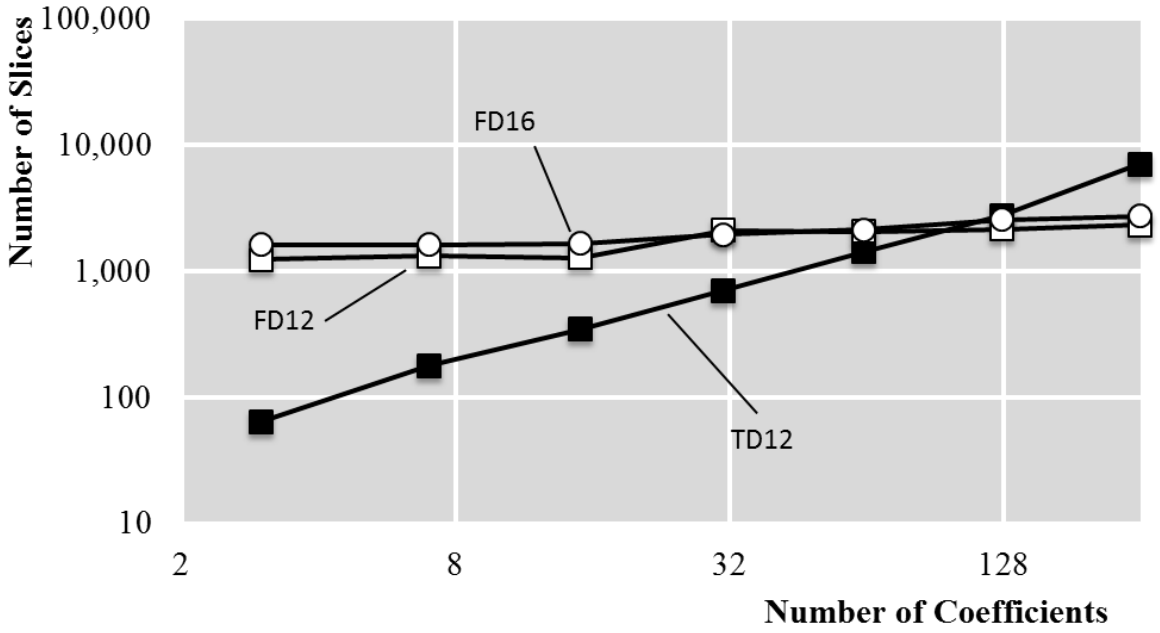

Figure 5.6 Slices for TD and FD filters when parallelism is 2, "TD12" indicates 12-bit TD designs, while "FD12" and "FD16" indicate 12-bit and 16-bit FD designs

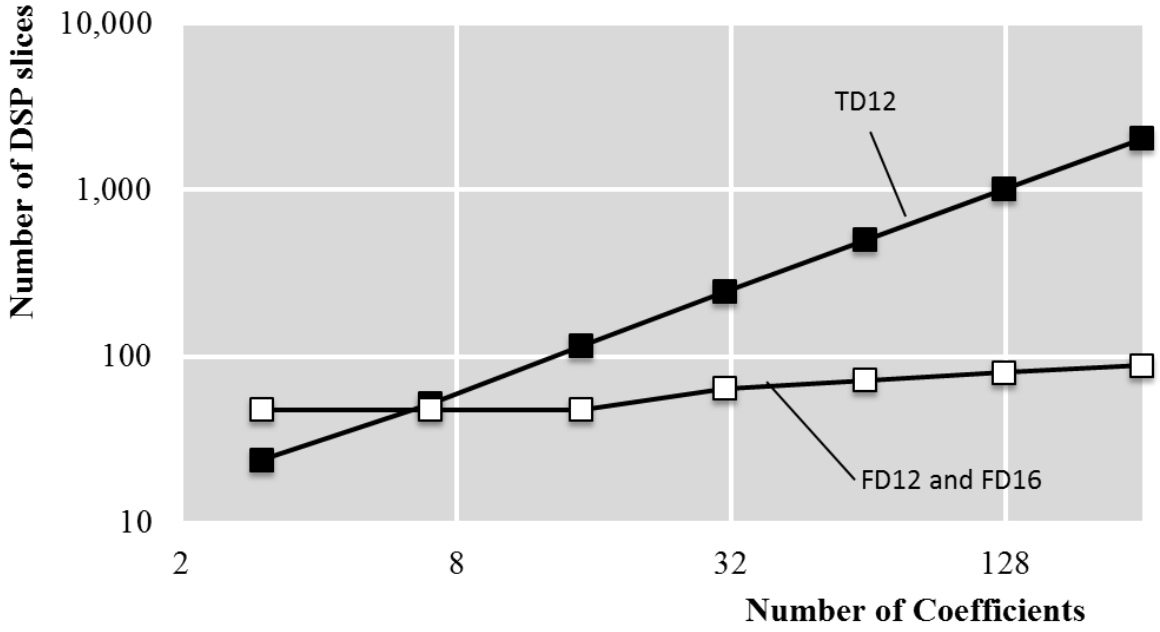

Figure 5.7 DSP slices usage for TD and FD filters when parallelism is 2

Figure 5.6 and Figure 5.7 show the area (in slices and DSP slices) of the TD and FD filters when the parallelism is 2. The number of DSP slices increases linearly in TD because

each DSP slice corresponds to a multiplier. The number of slices increases with the increasing of the number of coefficients, because the TD design will become larger and do more computation, while the slices and DSP slices for the FD design changes relatively little. However, FD designs require block RAMs (BRAMs), memory modules, in the FFT/IFFT which the TD designs do not need at all. With an increasing of the number of coefficients, the number of BRAMs required for FD increases greatly, which is shown in Figure 5.8.

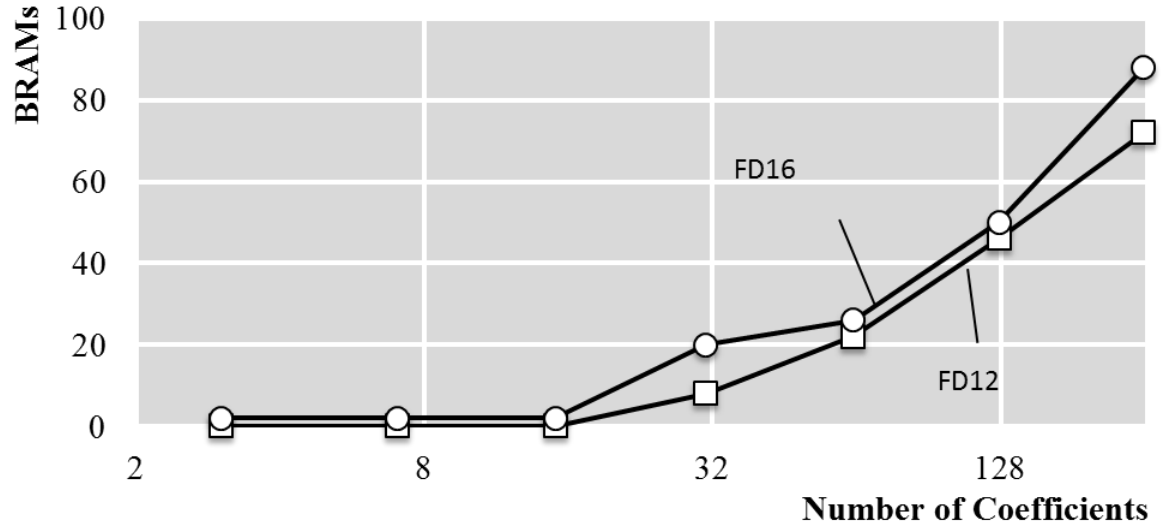

Figure 5.8 BRAMS for FD filters when parallelism is 2

Next, we increase parallelism to 8 and 16, with results shown in Figure 5.9. The TD designs of w=8 and number of coefficients greater than 64, and w=32 and number of coefficients greater than 16 cannot map to the FPGA (xc7vx980t) because of insufficient DSP slices.

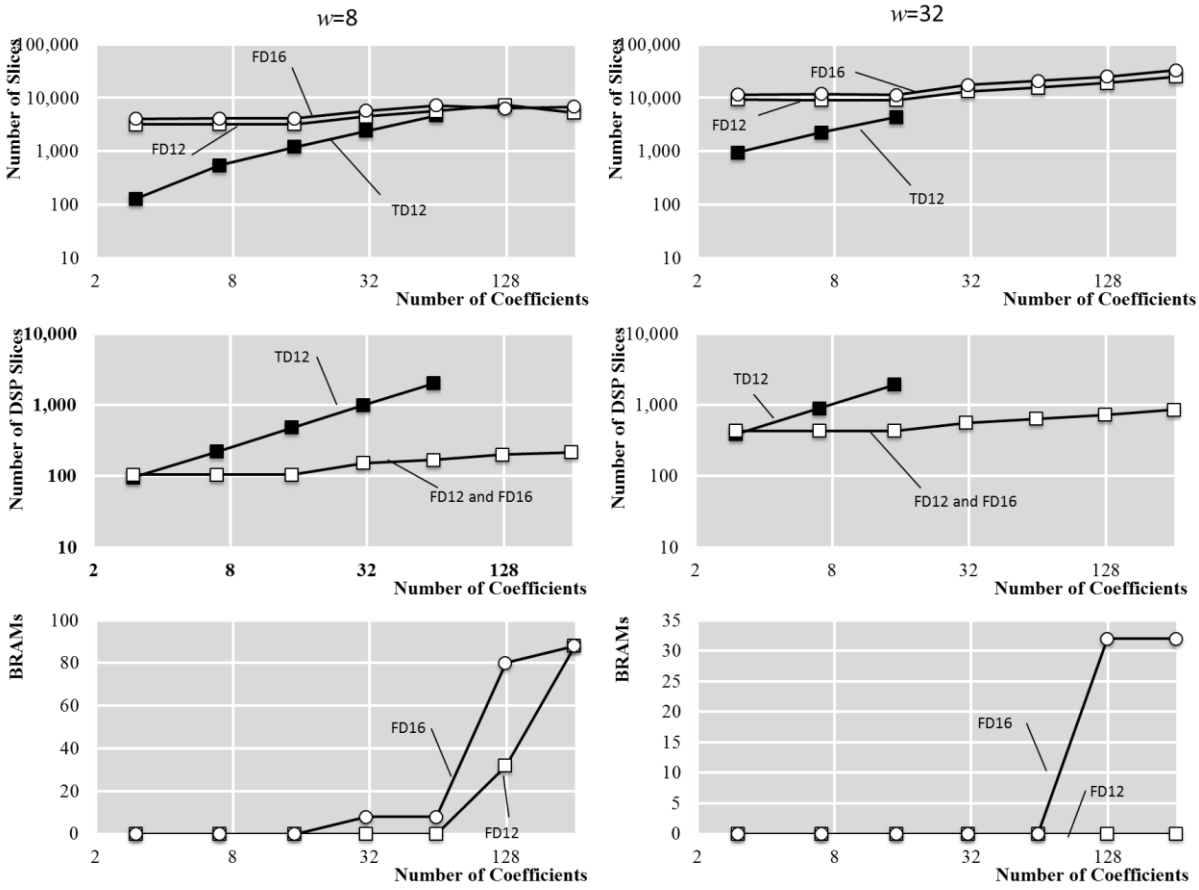

Figure 5.9 Area and Cost for TD and FD filters when parallelism is 8 and 32

The results with increasing the parallelism show the same trend as when parallelism equals. The slices and DSP slices will increase greatly when TD filters have more coefficients, while the BRAMs will be used more when the FD filters have more coefficients. In terms of the cost, when the number of coefficients is small, TD designs are ideal because fewer slices and DSP slices are occupied than FD designs and no BRAMs are used. FD designs are best for designs with a large number of coefficients. Although they require many BRAMs, the slices and DSP slices do not increase much. When the number of coefficients is somewhere in between, there is not necessarily a single better choice between FD and TD. Instead the designer must balance between slices, DSP slices and BRAMs.

5.3 Speed Measurement

In this section, we evaluate speed in terms of throughput. We define the throughput in Chapter 2.2.3 as parallelism (samples per clock cycle) times frequency (GHz), producing gigasamples per second.

With increasing parallelism, throughput will also increase, as long as the clock frequency stays roughly constant (due to pipelining). To evaluate the speed of the FD and TD filters, we use the same designs as in Chapter 5.2. We first increase the number of coefficients with parallelism 2 to see the trend of the speed. Then, we repeat the experiment with parallelism 8 and parallelism 16.

Figure 5.10 shows the throughput of each design. When the number of coefficients increases, the throughput decreases only slightly for the TD design. (The pipelining prevents larger decreases, but routing delays increase as the number of coefficients grows.) For the FD design, the throughput fluctuates highly when parallelism is 8 and 32. This is due to inconsistencies in clock frequency, which are caused by the synthesis tool's struggles with large designs. In designs with high parallelism, some smaller designs have larger routing delays than much larger designs.

By increasing the parallelism, the filters can reach very high throughput. The throughput of the TD filter with 16 coefficients and parallelism 32 can reach 10 billion words per second. In most cases, the TD filters have slightly higher throughput than the equivalent FD filters.

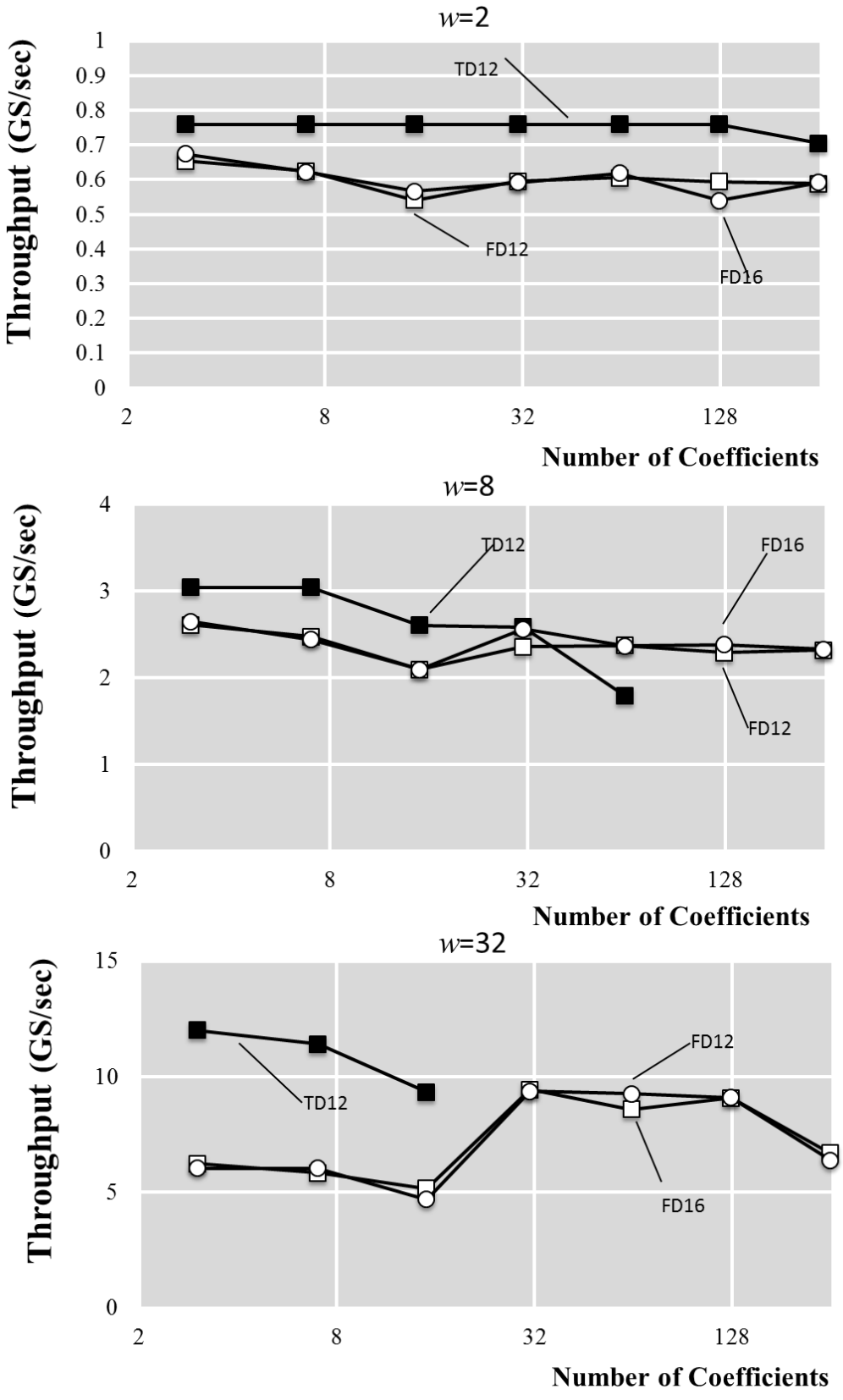

Figure 5.10 Throughput for TD and FD filters

5.4 Summary

In this chapter, we evaluated the cost and speed of the generated time-domain FIR filters, and compared them to frequency-domain FIR filters. We demonstrated that accuracy relates to many factors, not only the number of bits and the number of coefficients, but also the coefficients themselves. For a specific application, a designer can easily compare various options using the MATLAB simulation platform described in Chapter 4. As to the cost, when the number of coefficients increases, the amount of slices and DSP slices will increase quickly for TD filters. FD filters require a lot of memory when the filter size increases, but they have much lower requirements of DSP slices and slices than TD. So, TD filters are best for small designs while FD excels for large ones. For the speed, the TD filter has better throughput than the FD filter overall. When we increase the parallelism, throughput of both designs increases greatly.

We cannot make a general conclusion as to which filter is “better” because the parameters are application-dependent. We approach this problem by providing tools to conveniently evaluate and implement all options so designers can select the best choices for their requirements.

Chapter 6
Conclusions

Although FIR filtering is widely used in signal processing and related fields, efficient implementation of FIR filters in hardware can be difficult because the designer must make variety of choices based on application requirements. In this paper we first compared two adder structures for the direct form time domain (TD) FIR filter, namely the adder tree and adder cascade structure. The adder cascade structure mapped better to our FPGA because it is uses the FPGA's resources more efficiently. So this structure was then used as the baseline for the TD FIR filter design.

Then we developed a hardware generator. This tool automatically produces TD FIR filter implementations in Verilog, based on the user's specification, and includes a customized testbench module. We also built a MATLAB model to verify the design generated by the generator and measure the error. These two tools can take input parameters such as the filter size, filter coefficients, data type and parallelism. Using these two tools, we carried out synthesis-based experiments to evaluate error, area and speed of the filter as parameters change. We also compared the time domain FIR filters with other implementations of frequency domain (FD) FIR filters.

Our results allowed us to reach several interesting conclusions. First, we saw that the error of an implementation is affected not only by the data representation and the number of

coefficients, but also by the values of the coefficients themselves. Generally, the error increased with the increasing of the number of coefficients and with the decreasing of the number of bits (length) of the coefficients. We used the random coefficients and the root raised cosine filter coefficients to compare the TD filter and FD filter accuracy. For the TD filter, the random coefficients and the RRC coefficients had the same error level. The FD RRC filter can reach almost the same accuracy as the TD RRC filter, but the FD random filter needed three to four more bits to reach the same accuracy as TD random filter.

In terms of area, our results show that the TD FIR filter is suitable for filters with a low number of coefficients, because it needed less logic than FD FIR filters and it did not require BRAMs. For designs with a larger number of coefficients, FD filters were more suitable because the number of slices and the DSP slices required increased much more slowly than for the TD implementation. In terms of throughput, the TD filter was slightly better than FD.

With these tools and through the evaluation of their results, we give system designers the ability to easily optimize and reason about tradeoffs among cost, performance and accuracy related to FIR filter design.

Reference

[1] A. V. Oppemheim, A. S. Willsky and H. S. Nawab, Singals and Systems, Prentice Hall, 1997.

[2] A. V. Oppenheim, R. W. Schafer and J. R. Buck, Discrete-time signal processing, Upper Saddle River, New Jersey, USA: Prentice Hall, 1999.

[3] G. J. Proakis and M. Salehi, Digital Communications, Fifth Edition, New York, NY, USA: McGraw-Hill Higher Education, 2008.

[4] T. J. Terrell and R. J. Simpson, "Two-dimensional FIR filter for digital image processing," *Journal of the Institution of Electronic and Radio Engineers*, vol. 56(3), pp. 103-106, 1986.

[5] E. S. Chung, P. A. Milder, J. C. Hoe and K. Mai, "Single-Chip Heterogeneous Computing:Does the Future Include Custom Logic, FPGAs, and GPGPUs?," *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 225-236, 2010.

[6] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pp. 37-47, 2010.

[7] A. Gacic, M. Puschel and J. M. Moura, "Fast automatic implementations of FIR filters," *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 2,

pp. 541-544, 2003.

[8] L. R. Rabiner, " A simplified computational algorithm for implementing FIR digital filters," *IEEE Transactions on Acoustics, Speech and Signal Processing*, pp. 259-261, 1977.

[9] E. Turajlic and O. Bozanovic, "A novel adaptive FIR filter algorithm," in *2012 IX International Symposium on Telecommunications (BIHTEL)*, 2012.

[10] B. Spinnler, "Equalizer design and complexity for digital coherent receivers," *IEEE Journal of Selected Topics in Quantum Electronics*, vol. 16 no. 5, pp. 1180-1192, Sept. 2010.

[11] G. Grandmaison, J. Belzile, C. Thibeault and F. Gagnon, "Frequency domain filter using an accurate reconfigurable FFT/IFFT core," in *IEEE Northeast Workshop on Circuits and Systems*, 2004.

[12] P. Watts, R. Waegemans, M. Glick, P. Bayvel and R. Killey, "An FPGA-based optical transmitter design using real-time DSP for advanced signal formats and electronic predistortion," *Journal of Lightwave Technology*, Vol. 25, no.10, pp. 3089-3099, 2007.

[13] S. Israa, M. Amine, B. G. Leonardo, Q. Juan and N. A. Josef, "Frequency Domain vs. Time Domain Filter Design of RRC Pulse Shaper for Spectral Confinement in High Speed Optical Communications," in *14.2013 ITG Symposium Proceedings on Photonic Networks*, Leipzig, Germany, 2013.

[14] Y. Voronenko and M. Puschel, "Multiplierless multiple constant multiplication," *ACM*

Transactions on Algorithms, Vol. 3, no.2, 2007.

[15] H. Kang and I. Park, "FIR filter synthesis algorithms for minimizing the delay and the number of adders," *IEEE Journal on Circuits and Systems—II:Analog and Digital Signal Processing,* Vol. 48, no.4, August 2001.

[16] A. G. Dempster, S. S. Dimirsoy and I. Kale, "Designing multiplier blocks with low logic depth," in *IEEE International Symposium on Circuits and Systems (ISCAS),* 2002.

[17] J. Park, K. Muhammad and K. Roy, " High-performance FIR filter design based on sharing multiplication," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* Vol. 11,No.2 , pp. 244-253, 2003.

[18] D. Yagain and V. A. Krishna, " FIR filter design based on retiming automation using VLSI design metrics," in *2013 International Conference on Technology, Informatics, Management, Engineering, and Environment (TIME-E),* 2013.

[19] S.-J. Lee, J.-W. Choi, S. W. Kim and J. Park, " A Reconfigurable FIR Filter Architecture to Trade Off Filter Performance for Dynamic Power Consumption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* Vol. 19,No.12, pp. 2221-2228, 2011.

[20] Xilinx, "7 Series FPGA Overview," Xilinx, June 2011. [Online]. Available: http://www.xilinx.com/csi/training/7_series_FPGA_overview.htm.

[21] Xilinx, "7 Series DSP48E1 Slice User Guide," 30 January 2012. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf..

[22] T. Wada, "64 point Fast Fourier Transform Circuit (Version 1.0)," 9 September 2006. [Online]. Available: http://www.ie.u-ryukyu.ac.jp/~wada/design07/spec_e.html. [Accessed 9 Octorber 2013].

[23] R. Koutsoyannis, P. A. Milder, C. R. Berger, M. Glick, J. C. Hoe and M. Puschel, "Improving fixed-point accuracy of FFT cores in O-OFDM systems," in *ICASSP*, 2012.

[24] Y. Wu, H. Chen, G. Bischof and P. Milder, "Generation of customized FIR filter hardware", under review.

Appendix

The code in this appendix is a Verilog description of a time-domain FIR filter instance created by the generator described in this thesis. A random testbench for the instance is also generated. The time-domain FIR filter design instance has 4 coefficients, 2 stages of parallelism, and word length is 12 bits. The 4 coefficients are random complex numbers. They are
0.5521103740 – 0.4267403185j
-0.2835474610 – 0.1517075151j
-0.5101168752 + 0.1216162592j
-0.0605792403 – 0.8772545457j
After being converted to 12 bits, the coefficients become
1130 – 873j
-580 – 310j
-1044 + 249j
-124 – 1796j
The coefficients in integer are assigned in Verilog code below.

```
module FIR_timedomain(clk, x_inRe1, x_inIm1, x_inRe2, x_inIm2,
y1Re, y1Im, y2Re, y2Im);
parameter sample_width=12;
parameter K=2;

input clk;
input signed [sample_width-1:0] x_inRe1, x_inIm1, x_inRe2,
x_inIm2;
output signed [sample_width-1:0] y1Re, y1Im, y2Re, y2Im;

reg signed [sample_width-1:0] y1Re, y1Im,  y2Re, y2Im;
reg signed [sample_width-1:0] x_nmRe [2**K-1:0];
reg signed [sample_width-1:0] x_nmIm [2**K-1:0];

wire signed [2*sample_width:0] a11Re[3:0],a11Im[3:0],
a21Re[3:0],a21Im[3:0];
wire signed [sample_width:0] b11Re, b11Im, b21Re, b21Im;
wire signed [sample_width:0] c11Re, c11Im, c21Re, c21Im;
wire signed [sample_width:0] d11Re, d11Im, d21Re, d21Im;

reg signed [sample_width-1:0] a11Re_R[3:0],a11Im_R[3:0],
a21Re_R[3:0],a21Im_R[3:0];
```

```
reg signed [sample_width-1:0] b11Re_R, b11Im_R, b21Re_R, b21Im_R;
reg signed [sample_width-1:0] c11Re_R, c11Im_R, c21Re_R, c21Im_R;
reg signed [sample_width-1:0] d11Re_R, d11Im_R, d21Re_R, d21Im_R;
reg signed [sample_width-1:0] x_nm_11R1Re[3:2], x_nm_11R2Re,
x_nm_11R1Im[3:2], x_nm_11R2Im;
reg signed [sample_width-1:0] x_nm_21R1Re[3:2], x_nm_21R2Re,
x_nm_21R1Im[3:2], x_nm_21R2Im;

wire signed [sample_width-1:0] C0Re, C0Im, C1Re, C1Im, C2Re, C2Im,
C3Re, C3Im;
integer z;

assign C0Re = 12'd1130;
assign C1Re = -12'd580;
assign C2Re = -12'd1044;
assign C3Re = -12'd124;
assign C0Im = -12'd873;
assign C1Im = -12'd310;
assign C2Im = 12'd249;
assign C3Im = -12'd1796;


always @(posedge clk) begin
   x_nmRe[1] <= x_inRe1;
   x_nmIm[1] <= x_inIm1;
   x_nmRe[0] <= x_inRe2;
   x_nmIm[0] <= x_inIm2;
   for (z=0;z<2**K-2;z=z+1)begin
   x_nmRe[z+2] <= x_nmRe[z];
   x_nmIm[z+2] <= x_nmIm[z];
   end
end

always @(posedge clk) begin

   x_nm_11R1Re[2] <= x_nmRe[2];
   x_nm_11R1Re[3] <= x_nmRe[3];
   x_nm_11R2Re <= x_nm_11R1Re[3];
   a11Re_R[0] <=
{a11Re[0][2*sample_width],a11Re[0][sample_width+12-2:12]};
   a11Re_R[1] <=
```

```
{a11Re[1][2*sample_width],a11Re[1][sample_width+12-2:12]};
   a11Re_R[2] <=
{a11Re[2][2*sample_width],a11Re[2][sample_width+13-2:13]};
   a11Re_R[3] <=
{a11Re[3][2*sample_width],a11Re[3][sample_width+14-2:14]};
   b11Re_R <= {b11Re[sample_width],b11Re[sample_width+1-2:1]};
   c11Re_R <= {c11Re[sample_width],c11Re[sample_width+1-2:1]};
   d11Re_R <= {d11Re[sample_width],d11Re[sample_width+1-2:1]};

   x_nm_11R1Im[2] <= x_nmIm[2];
   x_nm_11R1Im[3] <= x_nmIm[3];
   x_nm_11R2Im <= x_nm_11R1Im[3];
   a11Im_R[0] <=
{a11Im[0][2*sample_width],a11Im[0][sample_width+12-2:12]};
   a11Im_R[1] <=
{a11Im[1][2*sample_width],a11Im[1][sample_width+12-2:12]};
   a11Im_R[2] <=
{a11Im[2][2*sample_width],a11Im[2][sample_width+13-2:13]};
   a11Im_R[3] <=
{a11Im[3][2*sample_width],a11Im[3][sample_width+14-2:14]};
   b11Im_R <= {b11Im[sample_width],b11Im[sample_width+1-2:1]};
   c11Im_R <= {c11Im[sample_width],c11Im[sample_width+1-2:1]};
   d11Im_R <= {d11Im[sample_width],d11Im[sample_width+1-2:1]};

   x_nm_21R1Re[2] <= x_nmRe[1];
   x_nm_21R1Im[2] <= x_nmIm[1];
   x_nm_21R1Re[3] <= x_nmRe[2];
   x_nm_21R1Im[3] <= x_nmIm[2];
   x_nm_21R2Re <= x_nm_21R1Re[3];
   a21Re_R[0] <=
{a21Re[0][2*sample_width],a21Re[0][sample_width+12-2:12]};
   a21Re_R[1] <=
{a21Re[1][2*sample_width],a21Re[1][sample_width+12-2:12]};
   a21Re_R[2] <=
{a21Re[2][2*sample_width],a21Re[2][sample_width+13-2:13]};
   a21Re_R[3] <=
{a21Re[3][2*sample_width],a21Re[3][sample_width+14-2:14]};
   b21Re_R <= {b21Re[sample_width],b21Re[sample_width+1-2:1]};
   c21Re_R <= {c21Re[sample_width],c21Re[sample_width+1-2:1]};
   d21Re_R <= {d21Re[sample_width],d21Re[sample_width+1-2:1]};
   x_nm_21R2Im <= x_nm_21R1Im[3];
```

```
   a21Im_R[0] <=
{a21Im[0][2*sample_width],a21Im[0][sample_width+12-2:12]};
   a21Im_R[1] <=
{a21Im[1][2*sample_width],a21Im[1][sample_width+12-2:12]};
   a21Im_R[2] <=
{a21Im[2][2*sample_width],a21Im[2][sample_width+13-2:13]};
   a21Im_R[3] <=
{a21Im[3][2*sample_width],a21Im[3][sample_width+14-2:14]};
   b21Im_R <= {b21Im[sample_width],b21Im[sample_width+1-2:1]};
   c21Im_R <= {c21Im[sample_width],c21Im[sample_width+1-2:1]};
   d21Im_R <= {d21Im[sample_width],d21Im[sample_width+1-2:1]};

   end

Mul mul110
(.clk(clk), .xRe(x_nmRe[0]), .xIm(x_nmIm[0]), .CRe(C0Re), .CIm
(C0Im), .yRe(a11Re[0]), .yIm(a11Im[0]));
Mul mul111
(.clk(clk), .xRe(x_nmRe[1]), .xIm(x_nmIm[1]), .CRe(C1Re), .CIm
(C1Im), .yRe(a11Re[1]), .yIm(a11Im[1]));
Mul mul112
(.clk(clk), .xRe(x_nm_11R1Re[2]), .xIm(x_nm_11R1Im[2]), .CRe(C
2Re), .CIm(C2Im), .yRe(a11Re[2]), .yIm(a11Im[2]));
Mul mul113
(.clk(clk), .xRe(x_nm_11R2Re), .xIm(x_nm_11R2Im), .CRe(C3Re),
.CIm(C3Im), .yRe(a11Re[3]), .yIm(a11Im[3]));
Mul mul212
(.clk(clk), .xRe(x_nm_21R1Re[2]), .xIm(x_nm_21R1Im[2]), .CRe(C
2Re), .CIm(C2Im), .yRe(a21Re[2]), .yIm(a21Im[2]));
Mul mul213
(.clk(clk), .xRe(x_nm_21R2Re), .xIm(x_nm_21R2Im), .CRe(C3Re),
.CIm(C3Im), .yRe(a21Re[3]), .yIm(a21Im[3]));
Mul mul210
(.clk(clk), .xRe(x_inRe1), .xIm(x_inIm1), .CRe(C0Re), .CIm(C0I
m), .yRe(a21Re[0]), .yIm(a21Im[0]));
Mul mul211
(.clk(clk), .xRe(x_nmRe[0]), .xIm(x_nmIm[0]), .CRe(C1Re), .CIm
(C1Im), .yRe(a21Re[1]), .yIm(a21Im[1]));
add add111
(.clk(clk), .x1Re(a11Re_R[0]), .x1Im(a11Im_R[0]), .x2Re(a11Re_
R[1]), .x2Im(a11Im_R[1]), .yRe(b11Re), .yIm(b11Im));
```

```
add add112
(.clk(clk), .x1Re(b11Re_R), .x1Im(b11Im_R), .x2Re(a11Re_R[2]),
 .x2Im(a11Im_R[2]), .yRe(c11Re), .yIm(c11Im));
add add113
(.clk(clk), .x1Re(c11Re_R), .x1Im(c11Im_R), .x2Re(a11Re_R[3]),
 .x2Im(a11Im_R[3]), .yRe(d11Re), .yIm(d11Im));
add add211
(.clk(clk), .x1Re(a21Re_R[0]), .x1Im(a21Im_R[0]), .x2Re(a21Re_
R[1]), .x2Im(a21Im_R[1]), .yRe(b21Re), .yIm(b21Im));
add add212
(.clk(clk), .x1Re(b21Re_R), .x1Im(b21Im_R), .x2Re(a21Re_R[2]),
 .x2Im(a21Im_R[2]), .yRe(c21Re), .yIm(c21Im));
add add213
(.clk(clk), .x1Re(c21Re_R), .x1Im(c21Im_R), .x2Re(a21Re_R[3]),
 .x2Im(a21Im_R[3]), .yRe(d21Re), .yIm(d21Im));
always @(posedge clk) begin
   y1Re <= {d11Re[sample_width],d11Re[sample_width+1-2:1]};
   y1Im <= {d11Im[sample_width],d11Im[sample_width+1-2:1]};
   y2Re <= {d21Re[sample_width],d21Re[sample_width+1-2:1]};
   y2Im <= {d21Im[sample_width],d21Im[sample_width+1-2:1]};
   end
endmodule


module Mul(clk, xRe, xIm, CRe, CIm, yRe, yIm);
parameter sample_width = 12;
input clk;
input signed [sample_width-1:0] xRe, xIm, CRe, CIm;
output signed [2*sample_width:0] yRe, yIm;
reg signed [2*sample_width:0] yRe, yIm;
reg signed [2*sample_width-1:0] f1, f2, f3, f4;
always @(posedge clk) begin
f1 <= xRe * CRe;
f2 <= xRe * CIm;
f3 <= xIm * CRe;
f4 <= xIm * CIm;
end
always @* begin
yRe = f1 - f4;
yIm = f2 + f3;
end
```

```
endmodule


module add(clk, x1Re, x1Im, x2Re, x2Im, yRe, yIm);
parameter sample_width = 12;
input clk;
input signed [sample_width-1:0] x1Re, x1Im, x2Re, x2Im;
output signed [sample_width:0] yRe, yIm;
reg signed [sample_width:0] yRe, yIm;
always @* begin
yRe = x1Re + x2Re;
yIm = x1Im + x2Im;
end
endmodule


/*`timescale 1ns / 1ps
module tb;
//Inputs
reg clk;
reg signed [11:0] x_inRe1, x_inIm1, x_inRe2, x_inIm2 ;

//Outputs
wire signed [11:0] y1Re, y1Im, y2Re, y2Im ;

integer file;

//Instantiate
FIR_timedomain uut (
   .clk(clk),
   .x_inRe1(x_inRe1),
   .x_inIm1(x_inIm1),
   .x_inRe2(x_inRe2),
   .x_inIm2(x_inIm2),
   .y1Re(y1Re),
   .y1Im(y1Im),
   .y2Re(y2Re),
   .y2Im(y2Im)
   );

initial clk = 0;
```

```
always #10 clk = ~clk;
initial begin
   file=$fopen("outputv.txt","w");
   // Initialize Inputs
   $monitor("outputRe=%d, outputIm=%d\n outputRe=%d,
outputIm=%d",y1Re, y1Im, y2Re, y2Im);
@(posedge clk) #1
x_inRe1<=-468;
x_inIm1<=-567;
x_inRe2<=222;
x_inIm2<=1241;
@(posedge clk) #1
x_inRe1<=-915;
x_inIm1<=1376;
x_inRe2<=-244;
x_inIm2<=-694;
@(posedge clk) #1
x_inRe1<=-359;
x_inIm1<=267;
x_inRe2<=-1056;
x_inIm2<=876;
@(posedge clk) #1
x_inRe1<=437;
x_inIm1<=-1282;
x_inRe2<=-1332;
x_inIm2<=-1733;
@(posedge clk) #1
x_inRe1<=-795;
x_inIm1<=1163;
x_inRe2<=-231;
x_inIm2<=-1538;
@(posedge clk) #1
x_inRe1<=1369;
x_inIm1<=-257;
x_inRe2<=252;
x_inIm2<=1380;
@(posedge clk) #1
x_inRe1<=-1049;
x_inIm1<=-1622;
x_inRe2<=-352;
x_inIm2<=1878;
```

```
@(posedge clk) #1
x_inRe1<=-699;
x_inIm1<=1093;
x_inRe2<=319;
x_inIm2<=863;
@(posedge clk) #1
x_inRe1<=121;
x_inIm1<=1267;
x_inRe2<=-1892;
x_inIm2<=-1579;
@(posedge clk) #1
x_inRe1<=214;
x_inIm1<=929;
x_inRe2<=-2047;
x_inIm2<=1301;
@(posedge clk) #1
x_inRe1<=1466;
x_inIm1<=919;
x_inRe2<=658;
x_inIm2<=-1769;
@(posedge clk) #1
x_inRe1<=749;
x_inIm1<=178;
x_inRe2<=-1459;
x_inIm2<=-2013;
@(posedge clk) #1
x_inRe1<=-1145;
x_inIm1<=-1190;
x_inRe2<=2019;
x_inIm2<=32;
@(posedge clk) #1
x_inRe1<=-1918;
x_inIm1<=-620;
x_inRe2<=-1013;
x_inIm2<=1231;
@(posedge clk) #1
x_inRe1<=1482;
x_inIm1<=-114;
x_inRe2<=1508;
x_inIm2<=-1958;
@(posedge clk) #1
```

```
x_inRe1<=1596;
x_inIm1<=-1075;
x_inRe2<=290;
x_inIm2<=1341;
@(posedge clk) #1
x_inRe1<=-1262;
x_inIm1<=-603;
x_inRe2<=-1487;
x_inIm2<=-513;
@(posedge clk) #1
x_inRe1<=1958;
x_inIm1<=269;
x_inRe2<=520;
x_inIm2<=308;
@(posedge clk) #1
x_inRe1<=-1825;
x_inIm1<=-1099;
x_inRe2<=-111;
x_inIm2<=-399;
@(posedge clk) #1
x_inRe1<=-1456;
x_inIm1<=-461;
x_inRe2<=1073;
x_inIm2<=798;
@(posedge clk) #1
x_inRe1<=-938;
x_inIm1<=-1129;
x_inRe2<=-484;
x_inIm2<=-1802;
@(posedge clk) #1
x_inRe1<=-115;
x_inIm1<=-245;
x_inRe2<=-270;
x_inIm2<=677;
@(posedge clk) #1
x_inRe1<=1438;
x_inIm1<=-770;
x_inRe2<=-1296;
x_inIm2<=604;
@(posedge clk) #1
x_inRe1<=-211;
```

```
x_inIm1<=366;
x_inRe2<=-706;
x_inIm2<=-1668;
@(posedge clk) #1
x_inRe1<=1178;
x_inIm1<=288;
x_inRe2<=-989;
x_inIm2<=1031;
$fclose(file);
$finish;
end
always @(*) begin
$fmonitor(file,"%d+j%d\n%d+j%d",y1Re,y1Im, y2Re, y2Im);
end
endmodule
```