NOTE TO USERS

LATCH-BASED DECIMATION FIR FILTERS

BY

Muhammad Ali Khan, B.Eng. (Electronics),

N.E.D. University of Engineering & Technology, Karachi, Pakistan

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Applied Science

Department of Electronics

Carleton University

Ottawa, Ontario, Canada

Your file Votre référence
ISBN: 0-494-13444-5
Our file Notre référence
ISBN: 0-494-13444-5

ABSTRACT

This thesis presents different methods of implementing decimation digital FIR filters and develops a method to achieve higher speed and lower power in these filters.

This performance improvement comes from using latches as the storage and delay producing elements rather than conventional flip-flops.

By using latches one can save up to 50% of the storage area and storage power budget as compared to conventional designs. The exception is one circuit presented in a recent paper that has been retimed to have the same area efficiency, unfortunately its maximum throughput is halved.

This thesis also presents a novel graphical method of showing pipeline-type digital operations through a time-space diagram. This diagram is used at different occasions to demonstrate the behaviors of decimation FIR filters in terms of time and space.

This thesis is dedicated to my father

Muhammad Hanif Khan,

for all the love and encouragement he gave me throughout my life.

Acknowledgement

I would like to experss my sincere gratitude to my supervisor, Professor John Knight, for his assisitance, encouragement and patience during my thesis work. His guidance and supervision allowed me to exercise my full potential and creativity in this thesis.

I would like to thank my friend Ashraf Siddiqi who helped me to resolve my Matlab issues and my fellow research group member Fred Ma for reviewing my thesis.

I would like to acknowledge Carleton University and Micronet for their financial support during this work.

Finally, I am deeply greatful to my parents, my sister and my wife who always support me with their unconditional love.

Table of Contents

List of Figures

List of Symbols

A_t = Add time, Assuming no Overlap.

b = Filter Coefficient.

CPA = Carry Propagate Adder.

CSA = Carry Save Adder.

Clk = Clock.

C_{out} = Carry out.

DSP = Digital Signal Processing.

EN = Enable Signal for Flip-Flop

FIR = Finite Impulse Response.

FA= Full Adder.

FF = Flip-Flop.

f_s = Sampling Frequency.

f_N = Nyquist Frequency.

HA = Half Adder.

hbf_N = Half-Band Filter's Nyquist Frequency.

Hbfs = Half-Band Filter's Sampling Frequency.

IIR = Infinite Impulse Response.

kHz = Kilo Hertz.

k = Length of Sample in Terms of Bits (used in CSAs).

Ln = Negative Level Sensitive Latch.

Lp = Positive Level Sensitive Latch.

MHz = Mega Hertz.

MSB = Most Significant Bit.

M_t = Multiply time.

M = Decimating Factor. Note : Also used to show the most significant bit in Verilog notation (Chapter 2).

M2C = 2's Complement of Most Significant Bits.

m = Maximum Number of Coefficients.

N = Number of Taps in a FIR Filter .

n = Sample Number.

nf_N = New Nyquist Frequency.

Q = Output of a Flip-Flop.

Tw = Transition Band Width.

T = Propagation Delay Through a Single Full Adder.

Tap = A Connection to the Data Stream Passing Through the Filter.
The Tap Feeds a Multiply Accumulate Unit.

TFF = Toggle Flip-Flop.

x = Input Samples.

y = Output Samples.

Φ = Phase Angle.

$\sum$ = Sum.

Chapter 1 Introduction

1.1 Introduction

During recent years, design for low power has become an important issue with the advent of portable equipment such as cellular phones, organizers, pagers, etc. Decimation filters perform an important role in different low power applications. Oversampled analog-to-digital conversion has become popular in wireless and audio applications because it achieves increased performance and flexibility by shifting signal processing complexity from the analog to the digital domain [1]. Oversampling uses much higher sampling rate than the bandwidth of interest. As a result, simple low-order analog anti-aliasing filters can be used with a more gradual transition width between passband and stopband. After that, decimating filters can efficiently perform further filtering and even down conversion, as is common in many new communication systems [2].

1.2 Thesis Motivation and Objectives

The decimation filter is a useful type of FIR filter. In this thesis we focus on different methods of designing decimation filters. Decimation is used to reduce the output sampling rate of a system compared to the input sampling rate. One common motivation for decimation is the reduction in processing cost. Generally the calculation and/or memory required to implement a DSP system is proportional to the sampling rate, and systems with lower sampling rates are more economical as compared to the systems with higher sampling rates, providing one does not sample below Nyquist's limit.

Lower computational complexity could be achieved for a required filter specification with the help of a multistage decimation filter structure [5]. Through decimation, the output data rate of a filter can be reduced, which results in a lower input data rate in subsequent filter stages. Through this technique, steeper skirts (transition widths) can be obtained with fewer taps [24].

Flip-flops are very important modules in digital design. In filter design, the necessary delay elements z^{-1} are generally created with flip-flops. One reason almost all practical and commercial digital filters possess a lot of flip-flops, is because the number of flip-flops is directly proportional to the length; i.e., the number of taps required to build the filter. Flip-flops are power hungry components. They consume a large fraction of the total power dissipated by the filter.

The main objective of this thesis was to find a way to reduce the silicon area and power dissipation of flip-flops by using latches instead. Latches are used to hold data like flip-flops but the main difference is that latches respond to the clock level while flip-flops are edge triggered. It is usually harder to design circuits in synchronous systems using latches instead of flip-flops.

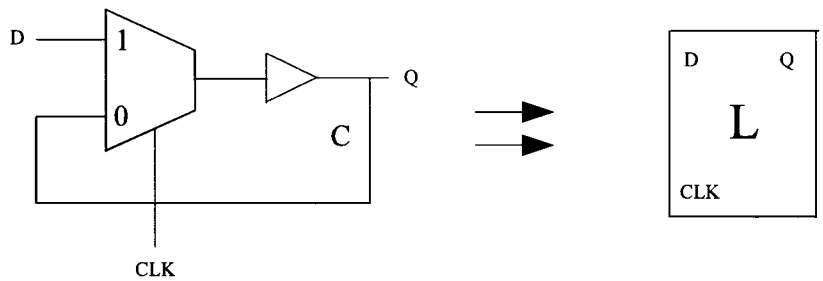

Figure 1.1: Level Sensitive Latch.

Flip-flops are made usually of latches i.e., most flip-flops consist of two latches. That is why latches consume half the area and less power than flip-flops (Figure 1.2).

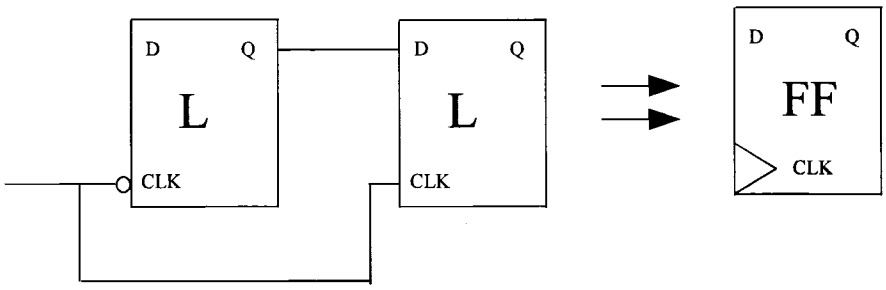

Figure 1.2: Edge Triggered Flip-flop construction.

This thesis presents a successful way of implementing latches in decimating FIR filters. By using latches the storage area and power budget can be reduced up to 50% as compared to conventional designs. However one paper has a retimed design that does match the latch-based filter in terms of area, but latch-based filter can run faster.

1.3 Thesis Organization and Outline

This thesis is organized to acquaint readers with the design and implementation of digital decimation FIR filters. Chapter 2 reviews the fundamentals of digital filters, their classification, comparison, etc. It also elaborates on design architectures for designing digital FIR filters. This chapter also elaborates on some computing modules, such as pipelined adder trees and carry save adders. Chapter 3 introduces decimation FIR filters. Basic concepts needed to understand decimation filters are discussed. This includes basic decimation and multistage decimation filtering. Chapter 4 discusses different practical implementation methods for decimation FIR filters. This includes in-depth operational analysis using time-space diagrams. Matlab/Verilog verification of a novel filter presented in this thesis is also included. Chapter 5 compares results of different implementations of decimation FIR filters. It also outlines the conclusions drawn from this thesis, and indicates the contributions of this research, including future research areas.

Chapter 2: Digital Filters

2.1 Introduction

In this chapter an introduction to digital filters is presented. Since the thesis is concerned with FIR filter implementation, the chapter highlights some common circuits for these filters. This chapter explains different common forms of implementation of digital FIR filters. As one of the forms of implementation heavily depends upon the different architecture of adders, a discussion has been provided about adder architecture that would improve the performance of this form. This discussion on adders also describes pipelining and its advantages. A section on the limitation of latches is included at the end of this chapter as background, because the main focus of the thesis is latch-based decimation FIR filters.

2.2 Digital Filters

In many signal processing systems, the analog input signal first goes through an analog anti-aliasing filter. It is then digitized using an ADC (analog-to-digital converter) which provides successive sampled values of the input signals as binary numbers. These digital samples (x_0, x_1, x_2, x_3...) are often fed to a digital filter, where they are processed resulting in a digital output (y_0, y_1, y_2, y_3...) from the filter. This filtered digital output may then be converted into analog signals by using a DAC (digital to analog converter) or further processed as a digital signal.

Digital filters have certain advantages. They can be designed and implemented in many forms, e.g., FPGA's. They can also be programmed; FIR filters in particular can have their characteristics changed easily by just changing the filter coefficients without any hardware

changes. Analog filters need to have components changed physically.

2.2.1 Classification of Digital Filters

Digital filters are classified into two major types. One type is the FIR (Finite Impulse Response) filters. In this type of filter, the current output value y(n) (where n is the nth sample) is calculated solely from the current and 'm' previous samples of the input x. This type of filter is said to be non-recursive and can be represented mathematically as

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) + ... + b_mx(n-m) \qquad (2.1)$$

The second type of digital filter is called an IIR (Infinite Impulse Response) filter. In this type of filter, the current output value y(n) can be thought of as being calculated from all previous input samples. An IIR filter can be represented mathematically as a FIR filter with an infinite number of coefficients.

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) + ... + b_\infty x(n-\infty) \qquad (2.2)$$

2.2.1.1 IIR (Infinite Impulse Response) Filters

IIR filters are also called recursive filters, reflecting the literal meaning of recursive, i.e., 'running back', and referring to the fact that they feedback previously calculated output values. Hence, the equation for the recursive filters can be expressed as a finite expression containing a finite number of input values {x(n), x(n-1), x(n-2),...} and also output values like {y(n-1), y(n-2),...}. This will be illustrated using the two-term filter described below, which has one feedback term, a_1.

$$y(n) = b_0x(n) + a_1y(n-1) \qquad \text{but} \qquad (2.4)$$

$$y(n-1) = b_0x(n-1) + a_1y(n-2) \qquad \text{substitute into (2.4)}$$

$$y_0(n) = b_0x(n) + a_1[\, b_0x(n-1) + a_1y(n-2)\,] \qquad (2.4)$$

similarly

$$y_0(n) = b_0x(n) + a_1b_0x(n-1) + a_1 [\, b_0x\,(n-2) + a_1y(n-3)\,] \qquad (2.5)$$

After an infinite number of substitutions, the recursive filter becomes equivalent to one without feedback but with an infinite number of coefficients.

The impulse response of a digital filter is the output sequence from the filter when a unit impulse is fed into the input. For a sampled signal, the unit impulse is a single value of 1 at time 0 with all other inputs zero.

The terms FIR (Finite Impulse Response) filters and IIR (Infinite Impulse Response) filters refer to the difference in impulse response of the two types of filters. FIR filters are those whose impulse response is non zero for a finite period of time, whereas IIR filters are those whose impulse response goes on forever because of the recursive terms.

2.2.1.2 FIR (Finite Impulse Response) Filters

This thesis discusses only FIR filters. The input output relation of FIR filters can be given in the z transform domain as

$$y(z) = (b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots\ldots + b_m z^{-m} / 1)\, x(z) \qquad (2.6)$$

The '1' in the denominator would contain the feedback coefficients if there were feedback. A FIR filter calculates it output from the current input value and a finite number of previous input values. They do not use previous output values to compute the current output value. Hence, the z-transform equation (2.6) above does not possess poles except at the origin, which is equivalent to having no feedback. This absence of feedback means that a FIR filter is oscillation free and unconditionally stable.

2.2.2 Comparison Between FIR and IIR Filters

It would seem that IIR filters should be larger then FIR filters as the former need both input and previous output values to calculate the current value. On the contrary FIR filters are larger than similar IIR filters. They are often one to two orders of magnitude larger than IIR filters for a similar filter response.

FIR filters are simple and relatively well understood, whereas IIR filters have more complex algorithms and are much more sensitive to non-idealities like quantization error. IIR filters are subject to oscillations called limit cycles, caused by the quantization of the digital signals being fed back. FIR filters have no limit cycles.

One of the qualities of FIR filters is that they can be easily be made linear phase. Linear phase means that the phase shift of all the frequency components passing through is proportional to the frequency (Figure 2.2.2). This is equivalent to saying that group delay through the filter is the same at all frequencies.

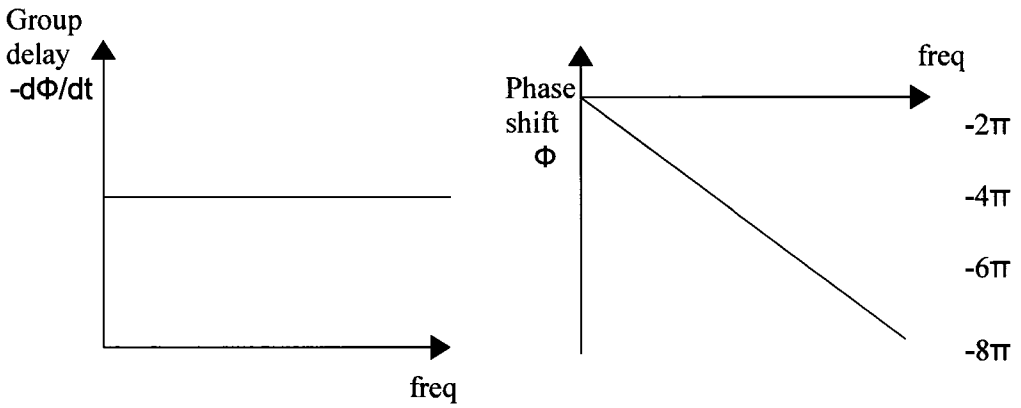

Figure 2.2.2: Phase Diagram for FIR Filters.

A FIR filter has a linear phase if it possesses symmetrical coefficients. This means

$$b_0 = b_{m-1}\ ,\ b_1 = b_{m-2},\ldots$$

If there are an odd number of coefficients then the only unpaired coefficient is the one in the center.

2.3 Digital Filter Design

This section discusses different methods used to design a FIR filter. First, one must design the filter theoretically to find the tap coefficients for a given specification. Matlab is an excellent tool for this. Once one gets the coefficients, the second task is to design the filter hardware. There are some basic computing modules used in almost every normal FIR filter, e.g., multipliers, adders, and delay producing modules. The designer usually has a goal of reducing circuit size and power consumption. This can be accomplished in part by reducing the number of modules.

There are different techniques to design efficient FIR filters. Some techniques reduce the number of multipliers. Some implementations arrange adders to gain speed. This thesis takes a different approach. We achieve efficiency be using level-sensitive latches in place of edge- triggered flip-flops. In the following sections we will discuss different implementations of FIR filters.

2.3.1 Direct-Form Implementation

The type of topology of Figure 2.4.1 is a direct form FIR filter [6]. Data samples $\{x_0, x_1, x_2, \ldots\}$ are passed through the delay components z^{-1}. Then they are multiplied by different coefficients, e.g.,. $\{b_0, b_1, b_2 \ldots\}$ to generate a product of the data and a particular coefficient. These products are added to get the final output. The equation for the circuit of Figure 2.4.1 can be written as

$$Y(z) = (b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3} + b_4 z^{-4} + \ldots)X(z). \qquad (2.7)$$

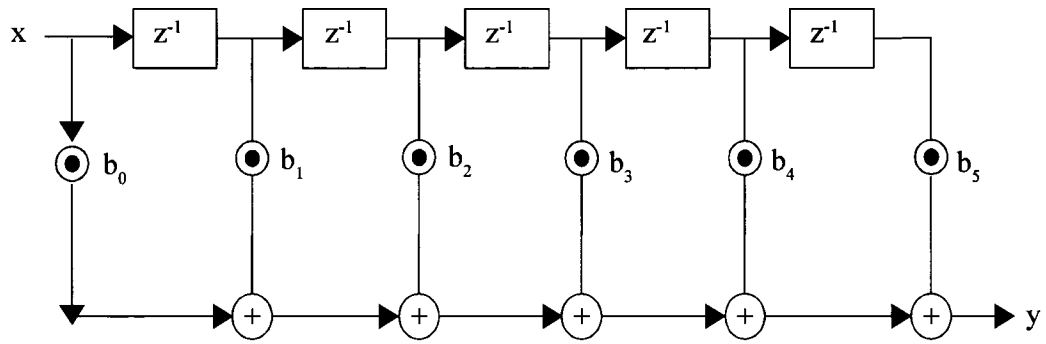

Figure 2.3.1: Direct Form FIR Filters.

This is the simplest form of FIR filters. It uses a large multi-term adder at the bottom. Therefore, one must run the clock slowly enough to allow the sum to propagate through the whole chain of adders between cycles. This is necessary because all the terms are added at once. In practical designs, different methods are used to increase the speed of the adders. These faster adders are discussed in detail in Section 2.4.

2.3.2 Transposed Direct-Form Implementation

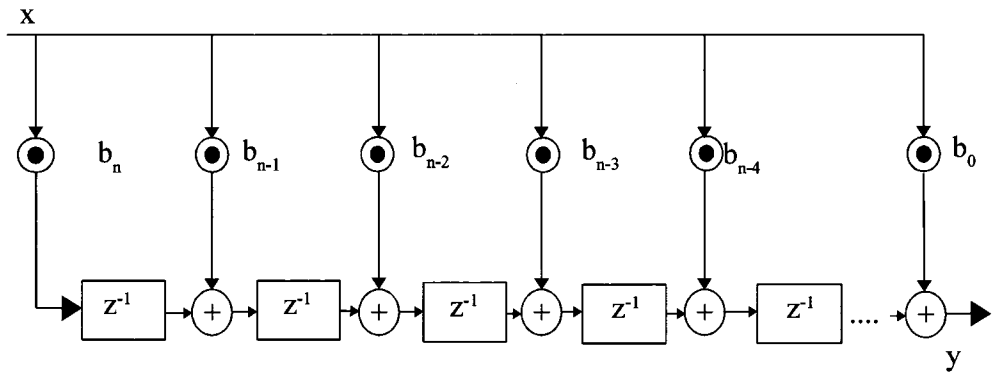

Figure 2.3.2: Transposed Direct-Form FIR Filters.

Figure 2.3.2 shows is an alternate method of implementing a direct-form FIR filter [8]. The equation for this circuit can be written as

$$Y(z) = (b_n \ldots + b_4 z^{-4} + b_3 z^{-3} + b_2 z^{-2} + b_1 z^{-1} + b_0)\, X(z). \qquad (2.8)$$

The data samples are multiplied by the coefficients $\{b_n, \ldots b_4, b_3, b_2, b_1\}$. The products are then added by adders inserted between the delay registers. It is not necessary to wait for all the products to be added before sending in the next sample. The registers between the adders gives the required z^{-1} delay and forms a pipeline for the adders.

2.3.3 Folded FIR Filters

One advantage of FIR filters is that we almost always have symmetrical coefficients $b_0 = b_{m-1}$, $b_1 = b_{m-1}$., etc. This is necessary to have linear phase. This property can be used to reduce the number of multipliers. Folded FIR filters use this property to reduce the number of multipliers by 50% without affecting the operation or performance of the filter. The data is multiplied with all the coefficients and the products are sent to two different positions in the delay-and-add path. The following is an example of a fold-back scheme (Figure 2.3.3).

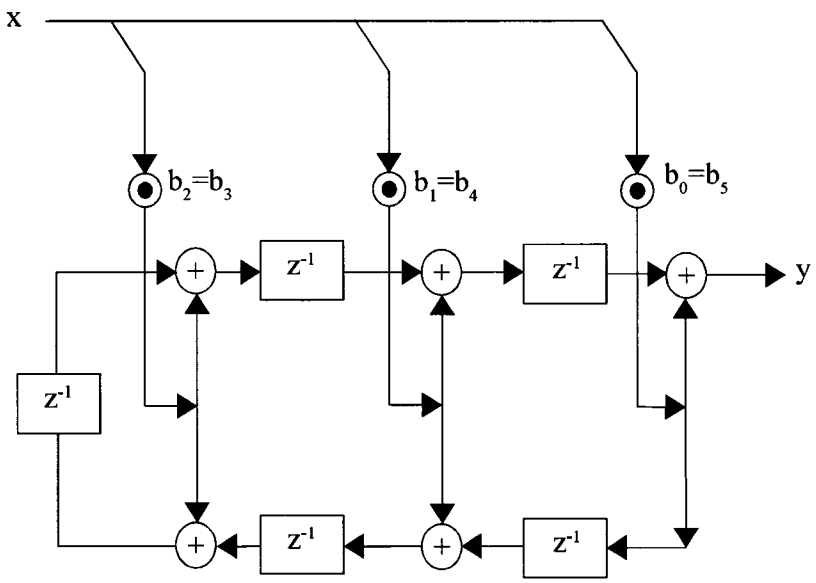

Figure 2.3.3: Alternate Direct-Form Folded FIR Filter.

All the filters described in this thesis can be folded. In many of the later diagrams the folding is omitted for clarity, since it is orthogonal to the point of the thesis.

2.4 Adders For Direct Form FIR Filters

The direct form FIR filters use long adders to add the products. This addition could be done faster using different forms such as adder trees. These adders trees could allow higher throughput through pipelining. These aspects of adder trees are discussed in the following sections.

2.4.1 Direct-Form with Carry-Propagate Adder Tree

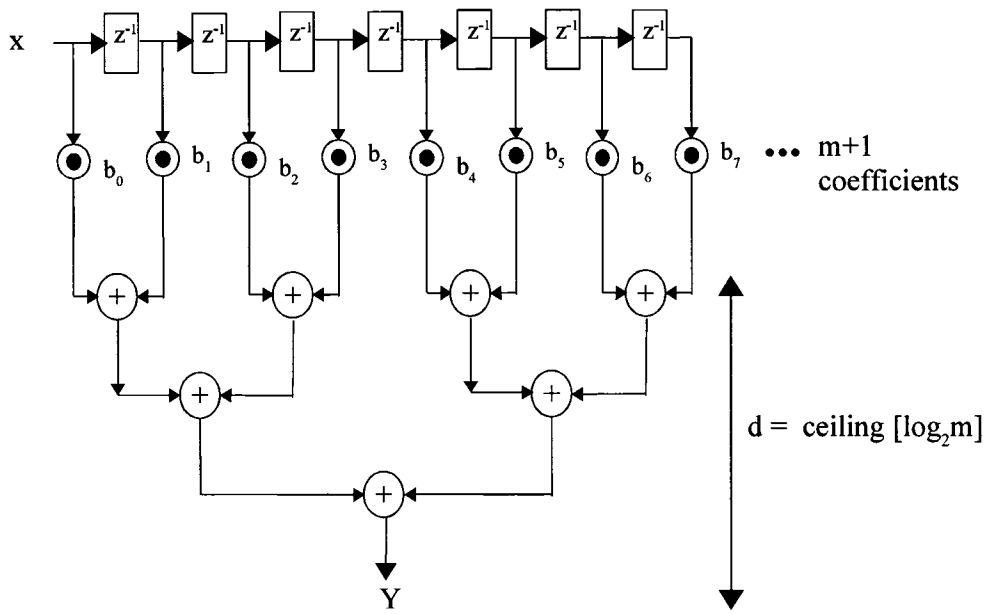

Figure 2.4.1: Direct Form FIR Filters with Adder Tree.

To help overcome the low maximum clock speed of the direct form (Section 2.3.1), we can use an adder tree (Figure 2.4.1). It performs some additions in parallel rather than serially. The circuit in Figure 2.4.1 consists of eight taps and a three-stage adder tree. The adders are still limited as they have to wait for the carry propagation inside each adder. There are different techniques used to speed up this process inside the adders. The adder tree circuit reduces the relative addition delay by a factor of

$$\text{ceiling } [\log_2 m]/m$$

2.4.4.1 Adder Tree with Pipelining

The delay can be further improved by pipelining the adder tree. Registers are placed between the different levels of the adder tree. This increases latency, but also increase throughput. Figure 2.4.4.1 shows an FIR filter with a pipelined adder tree. In the next section, we show how pipelining affects latency and increases throughput.

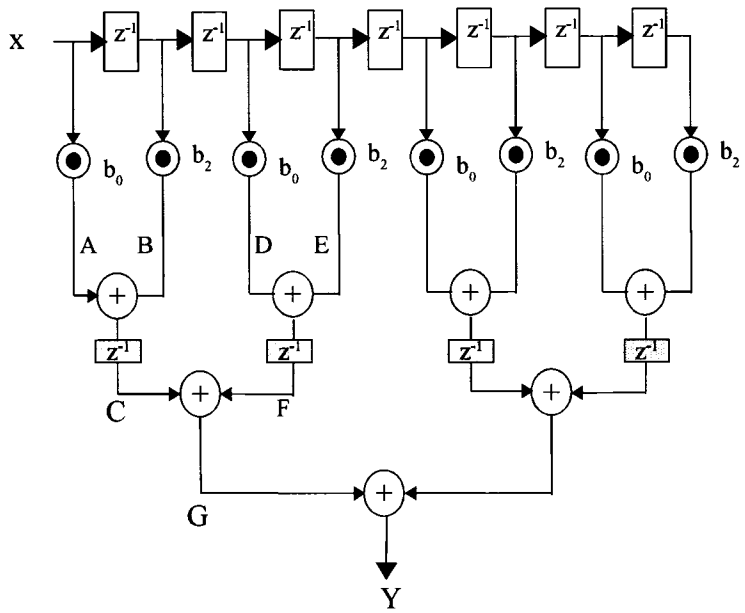

Figure 2.4.4.1: Direct Form FIR Filters with Pipelined Adder Tree.

The circuits in Figure 2.4.4.1 contains eight taps and an adder tree. It has pipeline registers after the first level of adder tree. The addition details are shown in Figure 2.4.4.2 using 3-bit numbers. The two products A and B are added to get C. D and E are also added to get F. C and F are then added in the next level of the adder tree to get G.

2.4.4.2 Comparison Between Normal and Pipelined Adder Trees

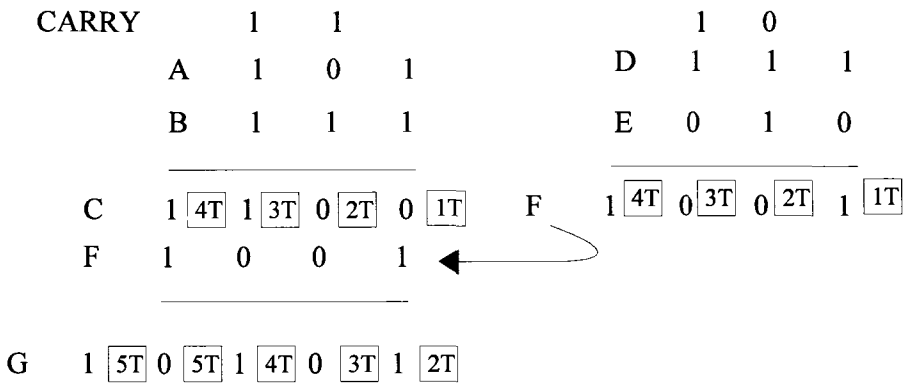

Figure 2.4.4.2.a: Timing Inside Normal Adder Tree.

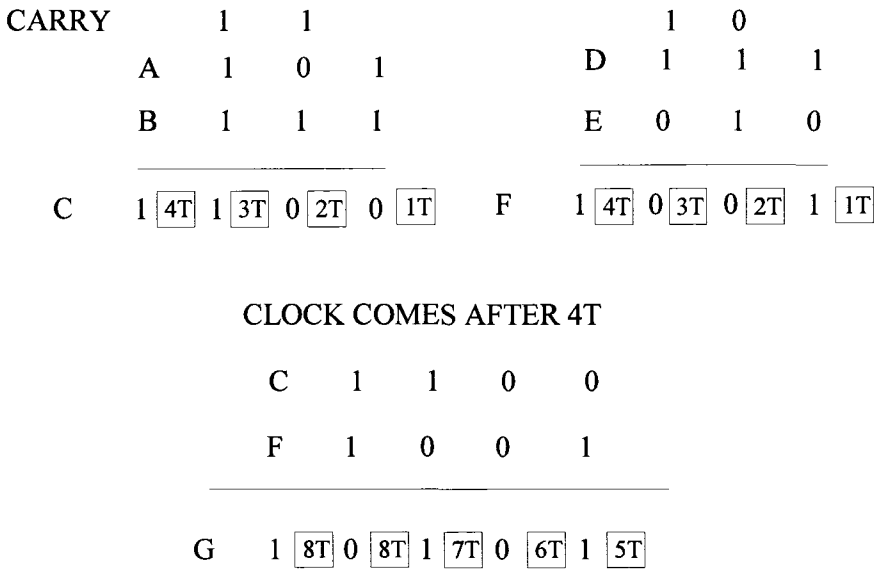

Figure 2.4.4.2.b: Timing Inside Pipelined Adder Tree.

Figure 2.4.4.2.a shows the adder tree operation with respect to time with no pipelining. Let T be the propagation delay through a single full adder. Expression 1T, 2T ,3T, etc., depicts the circuit delay through one full adder, two full adders etc., The last bit of F and G are available after 4T delays. However the first bit of F + C will be stable after a delay of 2T. Therefore, a two-level adder tree computes the sum G in 5T units of times.

Figure 2.4.4.2.b shows the effect of pipelining. The circuit adds A and B, and registers the result C. It also adds D and E and registers the result F. This means that the partial bit results of the addition of A + B and D + E are not immediately available for the next level of addition, i.e., between C and F. C and F are added in the cycle after A + B and D + E are done, i.e., after four units of time. Adding C and F costs another four units of time. Hence, the total time to add with pipelining is 8T.

One assumes here that the clock runs at the maximum speed which allows additions to complete. Comparing both cases, the result G is obtained after five units of time (5T) without pipelining, and after eight units of time 8T with pipelining.

The time used to add A + B and D + E is equal to four units. This means we can bring in the next sample of A + B and D + E after four units of time when we use pipelining registers. Without pipelining the next sample should come after five units of time. Pipelining makes it possible to bring the new sample in one unit time earlier. Hence, the throughput has increased with the use of the pipelined adder tree.

In general, including the multiplier delay, with m + 1 coefficients the tree is d = ceiling [$\log_2 m$] deep. Without pipelining, the clock cycle must be longer than M_t+dA_t. Where A_t is the propagation delay of a complete adder, and M_t is the multiplier delay. With pipelining, the clock period need only be longer than M_t+A_t.

This adder tree circuit can be made even faster by using a different method of addition called carry-save addition as shown in the next section.

2.4.5 Carry-Save Adders

In a carry-save adder (CSA) tree the carry is deferred and added separately rather the propagated to the end as in a normal carry-propagate adder (CPA). This allows a shorter total delay. Only one carry-propagate adder is needed at the end to sum up any outstanding carry bits. The following section presents the basic carry-save adder and an example circuit of a carry-save adder tree.

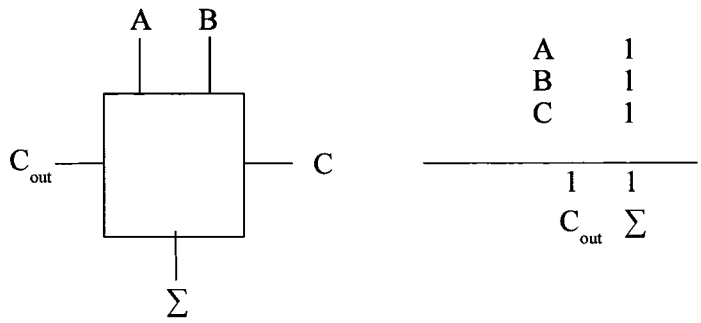

Figure 2.4.5: Full Adder Used in the Carry-Save Adder Cell.

Figure (2.4.5) shows the basic full adder used to create a carry-save adder cell. In this figure we add three bits A, B and C of three different words. The C input is used for a third input bit rather than an incoming carry bit. The addition of these three bits give a two bits output. One is the sum and the other represents the carry out. These basic adders are joined together to form a carry-save adder block to add k-bit words.

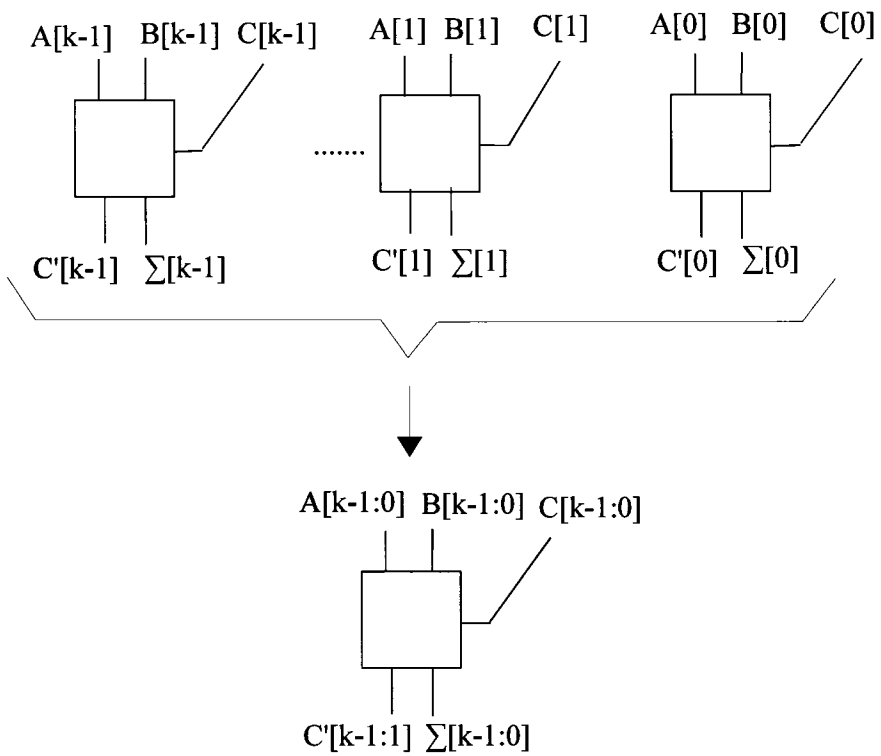

Figure 2.4.5.1: Basic Carry-Save Adder Cell Made from Full Adders. Note: The Subscripts Follow Verilog Notation.

2.4.5.1 Basic Cell for Carry Save Adders

Figure 2.4.5.1 shows how to join together single bit adders to form a carry-save addition for three k-bit numbers. When these carry-save adders are combined to form an adder tree, the appropriate weight of every bit is very important, since the sums and carries are added separately. To keep track of their bit positions, a Verilog type notation is used. C[k:1] indicates a vector with all indices between k and 1 inclusive. C[k:1] and ∑ [k-1:0] represent the bit positions of the Carry and Sum, respectively. The carry has no bit at position 0. The sum has no bit at position *k*. The following section explains how the bits are manipulated when they are added by a carry-save adder tree. A carry-save adder tree is first illustrated for positive numbers. Subsequently an adder tree is illustrated for 2's complement negative numbers.

2.4.6 Carry Save Addition Process

This section discusses the carry-save addition process in detail. The process is different for positive an negative numbers. This is because the sign bit in the negative numbers is added separately. Both processes are discussed below.

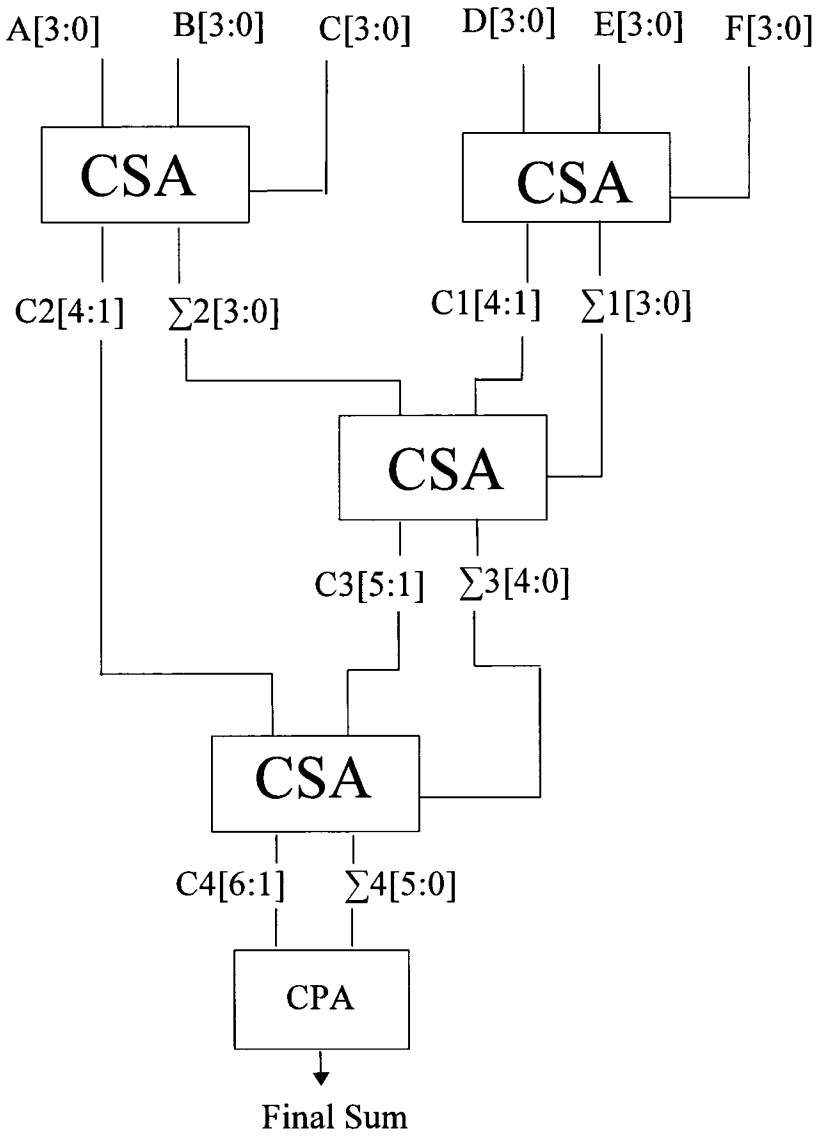

Figure 2.4.6.1: Carry Save Adder Tree for Positive Numbers.
Note: This adds six four-bit numbers.

2.4.6.1 Example for Carry-Save Addition with Positive Numbers

In this example six different positive numbers are summed with carry-save adders (Figure 2.4.6.1). Each number is a nibble, i.e, four bit long. If we add them normally then their sum should be 56 as shown below.

A = 1 0 0 1 = 9
B = 1 1 0 1 = 13
C = 0 0 0 1 = 1
D = 1 1 1 1 = 15
E = 1 0 0 0 = 8
F = 1 0 1 0 = 10

$\sum$ = 111000 = 56

Figure 2.4.6.1.a shows how the sums and carries are calculated separately, so there is no carry propagation delay. They are combined in a later stage. The nibbles ABC and DEF are added separately to give vectors $\sum_1$ and C_1, and $\sum_2$ and C_2 . Note that the carries are shifted and their proper position or weight is given by using a Verilog type notation, e.g., C_1 [4:1] and C_2 [4:1].

1 1 1 1	A [3:0]		1 0 0 1	D[3:0]	
1 0 0 0	B [3:0]		1 1 0 1	E[3:0]	
1 0 1 0	C [3:0]		0 0 0 1	F[3:0]	
1 1 0 1	$\sum_2$ [3:0]		0 1 0 1	$\sum_1$ [3:0]	
1 0 1 0	C_2 [4:1]	1 0 0 1		C_1 [4:1]	

Figure 2.4.6.1.a: Bit Positions in CSA I.

The first level generates four words. We can add only three terms per cell in a carry save adder. So first we add $\sum_1$ [3:0], C_1 [4:1] and $\sum_2$ [3:0] and we leave C_2 [4:1] for the time being. This addition will provided us $\sum_3$ [4:0] and C_3 [5:1] (Figure 2.4.6.1.b).

0 1 0 1	$\sum_1$ [3:0]	
1 0 0 1	C_1 [4:1]	(Note the shift)
1 1 0 1	$\sum_2$ [3:0]	
1 1 0 1 0	$\sum_3$ [4:0]	
0 0 1 0 1	C_3 [5:1]	

Figure 2.4.6.1.b: Bit Positions in CSA II.

The terms $\sum_3$ [4:0] and C_3 [4:1] with the left over term C_2 [4:1] are added with the help of another carry save adder cell. The result from this cell will be $\sum_4$ [4:0] and C_4 [5:1] (Figure 2.4.6.1.c).

1 1 0 1 0	$\sum_3$ [4:0]	
0 1 0 1	C_3 [5:1]	
1 0 1 0	C_2 [4:1]	
0 0 0 1 0 0	$\sum_4$ [5:0]	**(Added with carry propagate**
0 1 1 0 1 0	C_4 [6:1]	**adder)**
0 1 1 1 0 0 0	= 56	

Figure 2.4.6.1.c: Bit Positions in CSA III.

$\sum_4$ [5:0] and C_4 [6:1] are the final sum and carry terms we get out of the carry-save adder tree. These terms are added with a normal carry-propagate adder to get the final output, which is 56 in this example.

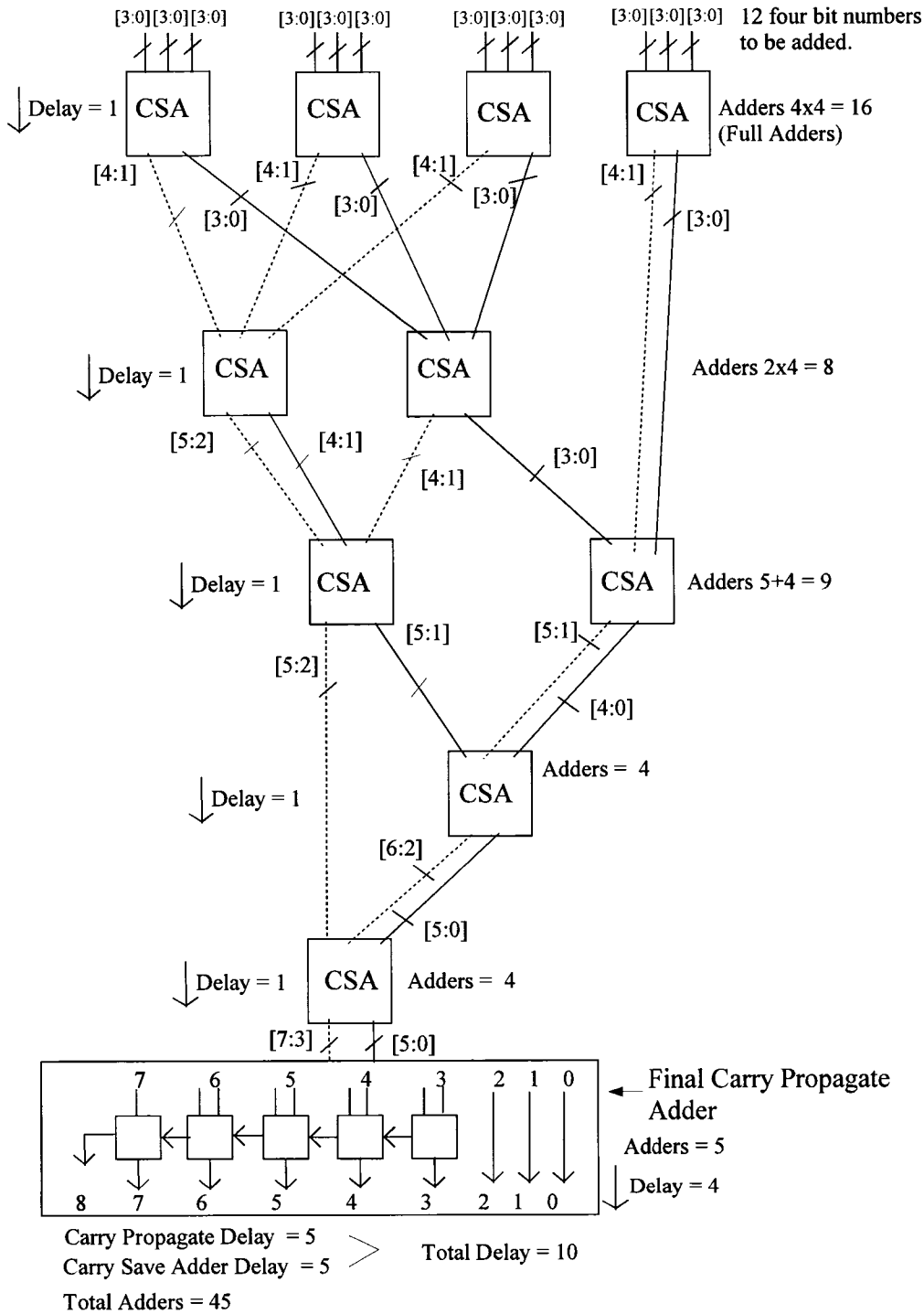

Figure 2.4.6.2: Carry-Save Adder Tree Stitching.

2.4.6.2 Carry-Save Adder Tree Stitching

Figure 2.4.6.2 illustrates a larger carry-save adder tree. It adds 12 samples of data, where each sample is a nibble (four bits long). The sums are propagated through a solid line while the carries propagate via dotted lines. The maximum time delay at each level is noted, along with the number of individual adders used at a particular level. Again a Verilog type notation is used for the bit positioning, and a carry propagate adder is used at the end for final clean up.

Notice that the bottom CSA has input data of bit positions [5:2], [6:2] and [5:0]. This means the bits [1] and [0] are not added and are fed through directly. Similarly the carry-propagate adder does nothing to the three low-order bits.

2.4.6.3 Example for Carry-Save Addition with Negative Numbers

An n+1 bit negative number is represented in 2's complement as

$$-2^{n}x_{n} + 2^{n-1}x_{n-1} + \ldots 2^{1}x_{1} + 2^{0}x_{0}.$$

The sign leading bits are added separately as positive numbers and negated at the end. In the example shown in Figure 2.4.6.3we have used six different 2's complement numbers with carry save adders. Each number is four bits long. If we add them normally, then their sum should be -24.

2^3 * sign bit

A' = 1 0 0 1 = -8+1 = -7
B' = 1 1 0 1 = -8+5 = -3
C' = 0 0 0 1 = -0+1 = 1
D = 1 1 1 1 = -8+7 = 1
E = 1 0 0 0 = -8+0 = -8
F = 1 0 1 0 = -8+2 = -6

$\sum$ = 101000 = -40+16 = -24

Figure 2.4.6.3: CSA with Negative Numbers.

The main difference between the addition of positive numbers and negative numbers is that with negative numbers the addition of the sign bits has to be considered separately. Figure 2.4.6.3.a shows the addition of the data bits only. Figure 2.4.6.3.b shows how the sign bits are added separately.

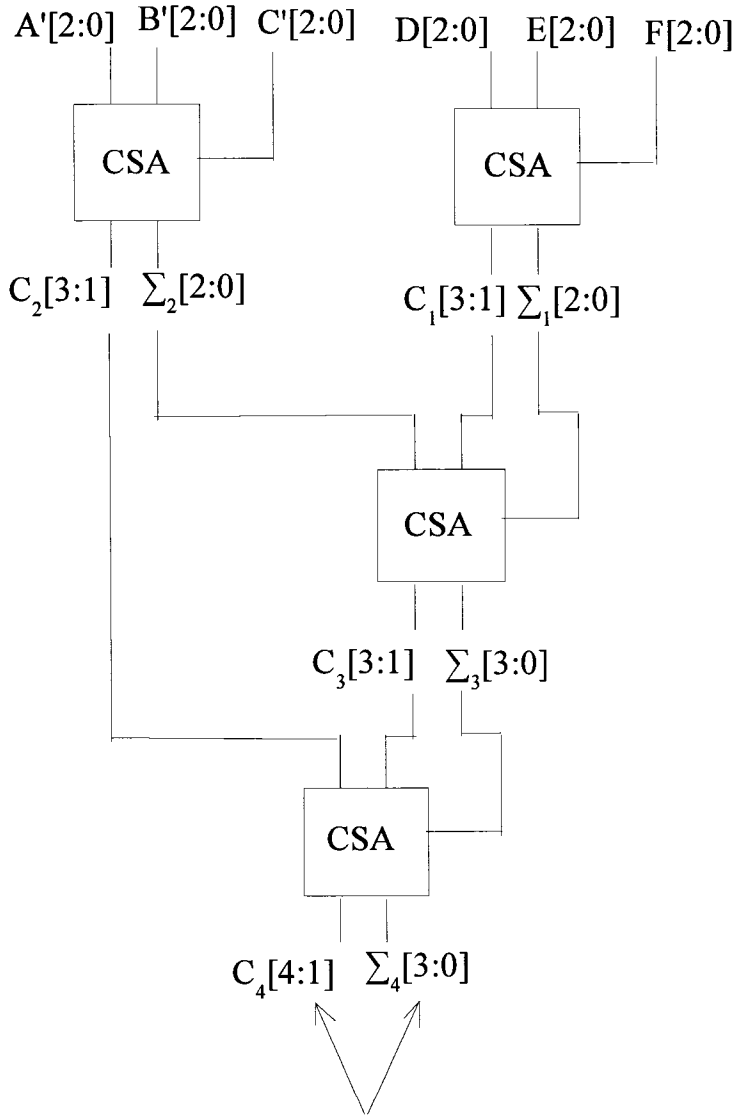

Figure 2.4.6.3.a: Carry-Save Adder Tree for Negative Numbers.
Note: This adds the data bits only.

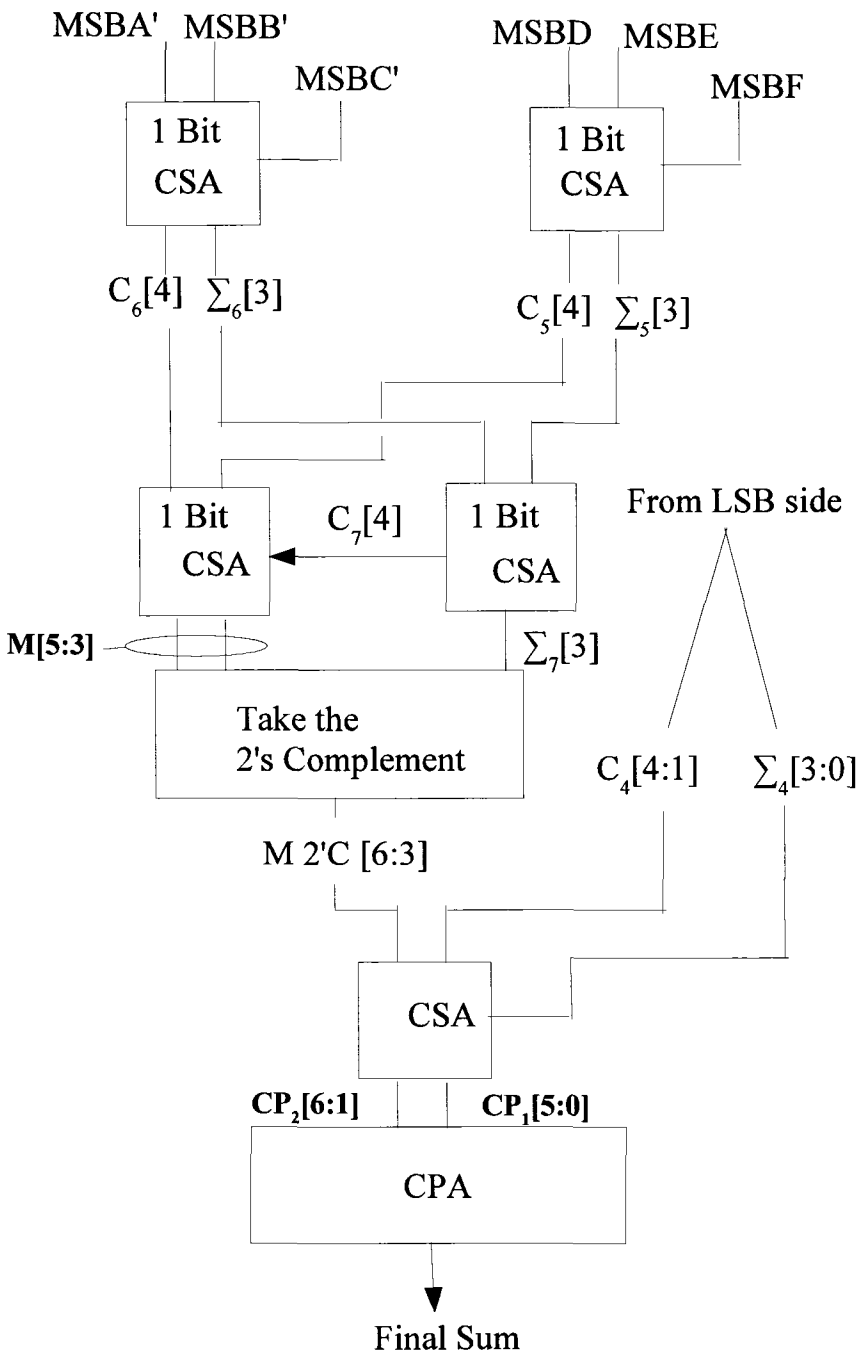

Figure 2.4.6.3.b: Carry-Save Adder Tree for Negative Numbers.
Note: This adds the sign (MSB) bits and adds them to the (LSB).

Dm 1[3]	A'_m 1[3]	1 1 1	[2:0]	0 0 1	[2:0]		
E_m 1[3]	B'_m 1[3]	0 0 0	[2:0]	1 0 1	[2:0]		
F_m 1[3]	c'_m 0[3]	0 1 0	[2:0]	0 0 1	[2:0]		
1 Σ_6 [3]	0 Σ_5[3]	1 0 1	Σ_2 [2:0]	1 0 1	Σ_1 [2:0]		
1 C_6 [4]	1 C_5 [4]	0 1 0	C_2 [3:1]	0 0 1	C_1 [3:1]		

Figure 2.4.6.3.c: Bit Positions in Negative CSA I.

Figure 2.4.8.1.c shows how we separate the data and sign bits. To add the data bits, a three bit carry-save adder cell is used. For the sign bit, only a single bit adder is used. In figure 2.4.8.1.c we added the data bits of ABC and DEF separately. $\{\Sigma_1[2:0], C_1[3:1]\}$ and $\{\Sigma_2[2:0], C_2[3:1]\}$ are their respective sums and carries. $\{\Sigma_5[3], C_5[4], \Sigma_6[3], C_6[4]\}$ are the respective sum and carry bits for the sign bit.

		1 0 1	Σ_1 [2:0]
0	Σ_5 [3]	0 0 1	C_1 [3:1]
1	Σ_6 [3]	1 0 1	Σ_2 [2:0]
1	Σ_7 [3]	0 0 1 0	Σ_3 [3:0]
0	C_7 [4]	0 1 0 1	C_3 [4:1]
1	C_5 [4]	0 1 0	C_2 [3:1]
1	C_6 [4]		

0 1 0 1 M[6:3] + 40 Sign bit weights as a positive number
1 0 1 1 M2C[6:3] 2's Comp - 40 Sign bit weights as a negative number

$\{-2^6+...+2^3+2^2\}$

Figure 2.4.6.3.d: Bit Positions in Negative CSA II.

Figure 2.4.6.3.d shows that carry save adders can only add three terms, so $\sum_1$ [2:0], C_1 [3:1], $\sum_2$ [2:0] are added first. C_2 [3:1] is left for the time being. This addition will generate $\sum_3$ [3:0] and C_3 [4:1]. Again, the sum and carry of the sign bits are handled separately. $\sum_5$ [3] and $\sum_6$ [3] are added giving $\sum_7$ [3] .

The bit position [3] of the sign bit addition has only one term i.e., $\sum_7$ [3]; there is no other term at this position to add. Hence, $\sum_7$ [3] is joined as it is with the sum motion of C_7 [4], C_5 [4], and C_6 [4] which is M[5:3]. The word 'M' is used here because these are the most significant bits.

These M[5:3] bits must be 2's complemented in order to follow the algorithm of negative number addition, and this complement is named M2C[5:3].

0 0 1 0	$\sum_3$ [3:0]	
0 1 0 1	C_3 [4:1]	
0 1 0	C_2 [3:1]	
0 1 1 0 0	$\sum_4$ [4:0]	+ 12
0 0 0 1 0	C_4 [5:1]	+ 4
1 0 1 1	M2C [6:3]	- 40
0 1 0 0 0 0	CP_1 [5:0]	**(Added with carry propagate adder)**
1 0 1 1 0 0	CP_2 [6:1]	
1 1 0 1 0 0 0	=	-24

Figure 2.4.6.3.e: Bit Positions in Negative CSA III.

The terms $\sum_3$ [3:0], C_3 [4:1], and left over term C_2 [3:1] are added with another carry save adder cell. The result from this cell will be $\sum_4$ [4:0] and C_4 [5:1]. At this point there are two terms

resulting from the data path and one resulting term from the sign bit path. This is the time to merge both paths together. For this reason we add $\sum_4$ [4:0], C_4 [5:1],and M2C [5:3]. This gives us two final terms CP_1 and CP_2. At the end, CP_1 and CP_2 are summed through a cleanup adder. This carry propagate adder gives us the final result which is -24 for this particular example (Figure 2.4.6.3.e).

Hence carry-save adders are useful and can add faster than carry-propagate adders. The positive and negative (2's complement) numbers can be added as shown in the previous sections. With the use of these carry-save adders direct form FIR filters could operate faster.

2.5 Latch-Based Circuits

The main focus of this thesis is latch-based FIR filters. Latch based circuits have limitations and cannot work well with all type of circuits. However latches were found to be beneficial in the case of decimation FIR filters. Latches were tried with regular FIR filters as well but no significant advantage was achieved. In the following sections we will discuss the limitation of latch-based circuits, with a quick review of some of the design problems.

2.5.1 Symbolic Conventions

Figure 2.5.1 presents some symbolic conventions, which are used in this chapter. We also review the relation between a latch and a flip-flop as discussed in the introduction.

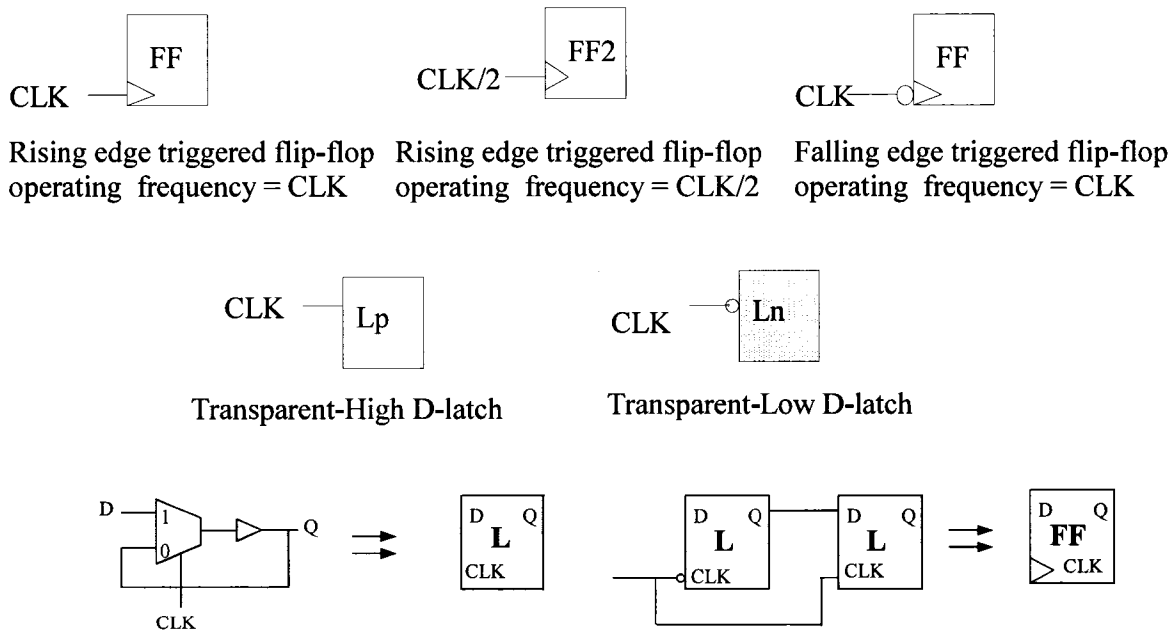

Figure 2.5.1: Symbolic Conventions.

2.6 Limitations of Latch-Based Circuits

This thesis is based on replacement of flip-flops with latches. Latches are seldom used in mainstream digital design. The following section will explain some of the problems.

2.6.1 No Feedback Around a Latch

Latches are transparent for half the clock cycle. During this half cycle feedback around a latch may oscillate or act as a unwanted storage element (Figure 2.6.1). The circuit would become an asynchronous circuit with all its design problems [28].

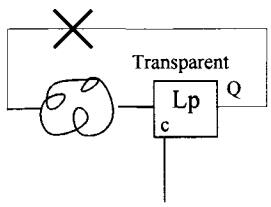

Figure 2.6.1: Feed back around a latch is not recommended.

2.6.2 Upper and Lower Bounds on Timing Constraints

A signal feeding forward through consecutive odd latches becomes a difficult timing problem. If the signal propagates through both latches it will come out one cycle early. However it must propagate through Logic-1 (Figure 2.6.2) and be captured by the first latch. This puts two timing constraints on Logic-1, an upper bound and a lower bound, and satisfying both is very difficult. With flip-flop storage the lower bound is called the hold time and is normally fairly easy to satisfy.

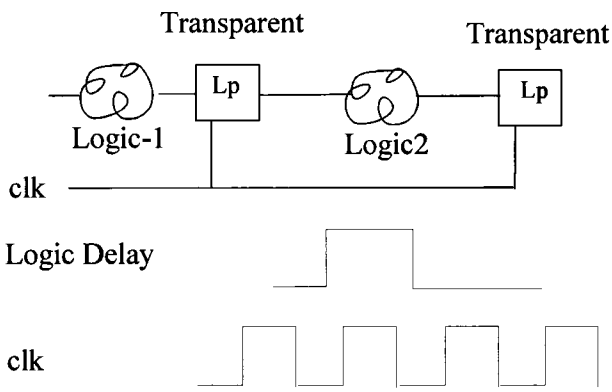

Figure 2.6.2: Two consecutive odd/even latches becomes single latch.

2.6.3 Feedback Loop Limitations

To avoid the double timing constraints a signal leaving an odd latch must pass through an even latch before reaching another odd latch and vice-versa. Loops must go from an odd-phase latch outputs to an even-phase latch input (Figure 2.6.3).

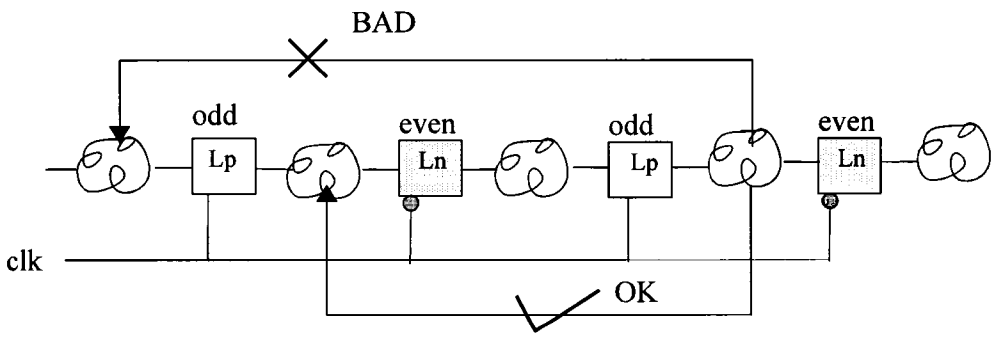

Figure 2.6.3: Feedback Loop Limitations in Latches.

2.6.5 State Graph Restriction.

One of the limitation comes because half the latches are transparent and only half are holding, so only half of the states are in effect at a time.

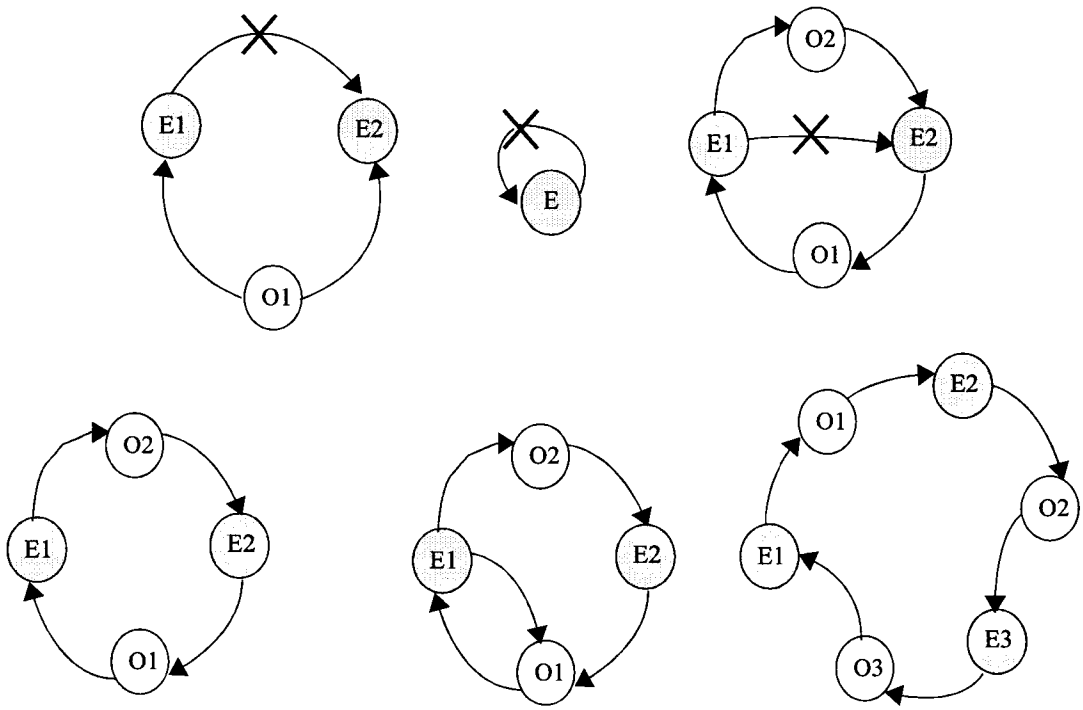

Figure 2.6.4: Latch Limitations II.

While using latches, states graphs should be divided into odd and even states. This is a result of the feedback restriction of the last section. The odd states must feed even states and even states must feed odd states. Thus the state graphs cannot possess wait states and must always have even length loops like doublets, squares and hexagons (figure 2.6.4).

Most circuits do not benefit from using latches. Control circuits have complex state graphs which often use loops with an odd number of states. Pipeline circuits usually do not save storage area by using latches.

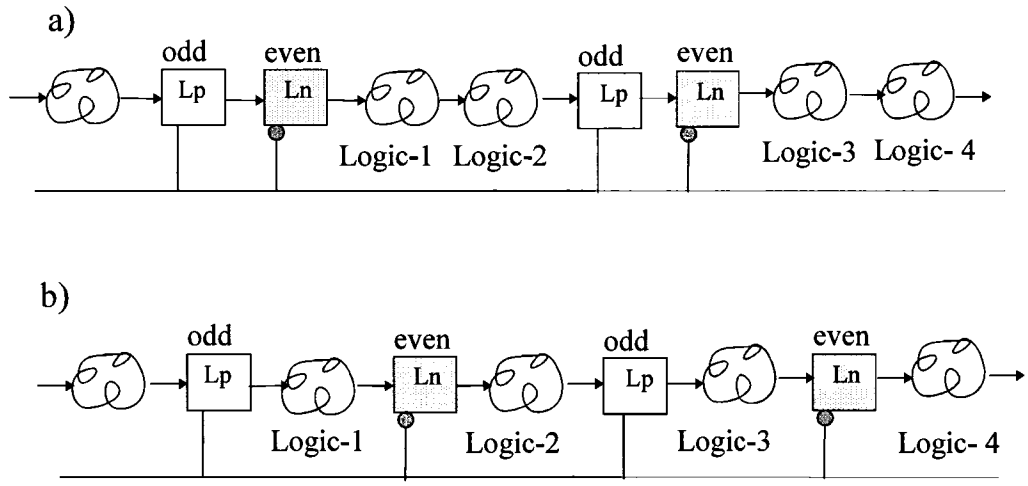

Figure 2.6.4.1: Pipelining with latches.

Figure 2.6.4.1 gives an intuitive idea why storage is not saved. In part a, logic is placed between flip-flop registers shown as two latches. In part b, the logic is place between every alternate latch. These two circuits are almost identical and there is no gain by using latches for pipelining. This turned out to be the case for non-decimating FIR filters.

2.6.5 Complier problem

There are other reasons why designers may prefer not to use latches. These are related to present synthesis tools and are not basic.

The asynchronous reset on a latch is not synthesized logically but require a special compiler directive. These take the form of a comment line which is ignored by the simulator but intercepted by the synthesizer. Comment in Verilog starts as '//' and the reset command is

// synopsis async_set_reset "clear".

The compiler will automatically generate test circuitry for circuits with flip-flop registers (scan chains). Latch circuit must have test circuitry designed manually.

2.7 Summary

Digital filters are advantageous in terms of implementation. The response of the digital filters can be changed easily as compared to analog filters by reprogramming the coefficients. Nearly all FIR filters have symmetric coefficients and can be folded easily to half the number of multipliers. Two forms of FIR filters were discussed. In the transposed direct form storage elements could be used as delay and pipelining. The direct form has a long slow adder chain which can be made faster by pipelining registers and carry-save adders.

Restrictions in state graphs, feedback loops, and synthesis tools makes latch based circuit implementations limited as compared to flip-flop based circuit. Also there are often minimal gains form using latches. However we will show that they have advantages for decimating FIR filter implementation.

CHAPTER 3 Decimation FIR filters

3.1 Introduction

This chapter explains the different aspects of decimation FIR filters. Aliasing is an important aspect of decimation FIR filters for this reason we discuss anti-aliasing filter requirements especially digital anti-aliasing filters. An application of decimation filters is discussed in terms of multistage oversampled filters, with the help of examples. In multistage decimation the use of half-band filters is common, therefore half-band filters are also included in the discussion.

3.2 Basic Decimation Filter

We can create a decimation filter by simply discarding any unwanted results from a regular filter. For example, if the input data rate is 100 MHz, and we want to decimate the data rate by 2, then the output data rate will be 50 MHz (Figure 3.2).

Unwanted signals, noise or undesired channels would be removed with a single filter, and then decimating to the output rate. However this will make an unnecessarily larger filter.

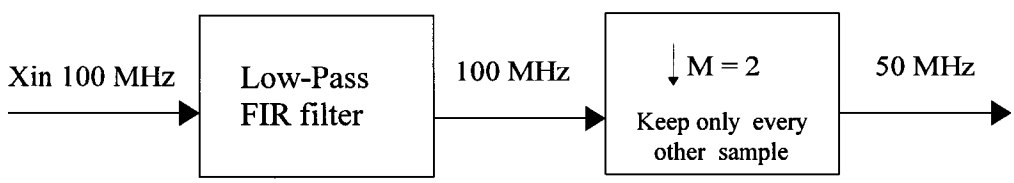

Figure 3.2: A Decimation FIR Filter.

Figure 3.2 shows a basic decimation filter with a decimating factor M = 2. A decimation filter only computes every Mth result. It is often beneficial to decimate in several stages as will be discussed in Section 3.6. Before that it will be helpful to discuss aliasing and half-band filters.

3.3 Aliasing

In sampling theory, when an input signal frequency component exceeds half the sampling frequency (the Nyquist frequency), this frequency will appear to have come from a lower frequency in the digitized signal. Figure 3.1 shows two different aliased signals A and B. To prevent aliasing, these unwanted signals must be filtered out before they are digitized by the analog-to-digital converter (ADC).

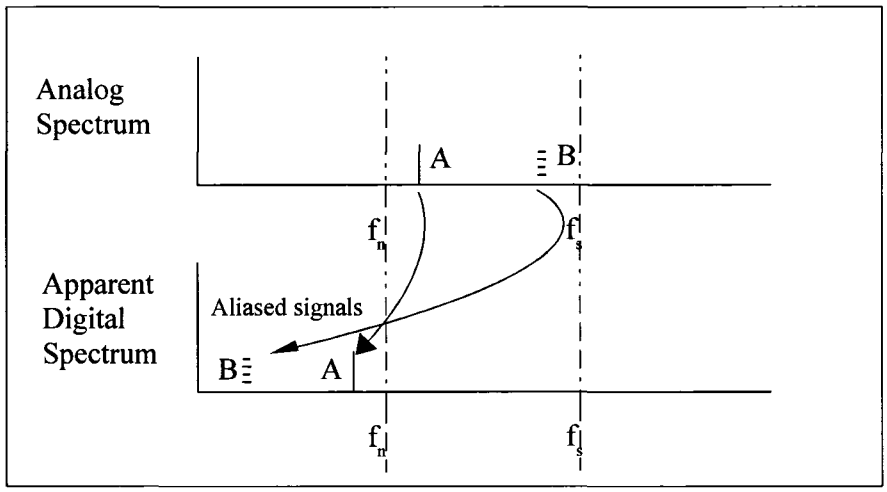

Figure 3.3: Aliasing Across Nyquist Frequency.

3.3.1 Antialising Filter Requirements

Figure 3.2.1 part (a) shows a filter response with an extra pass band. Part (b) shows the input signal after sampling. Sampling in time makes the filter spectrum periodic in frequency. Its spectrum is reflected around the Nyquist frequency. Part (c) shows the overall response after passing through the filter.

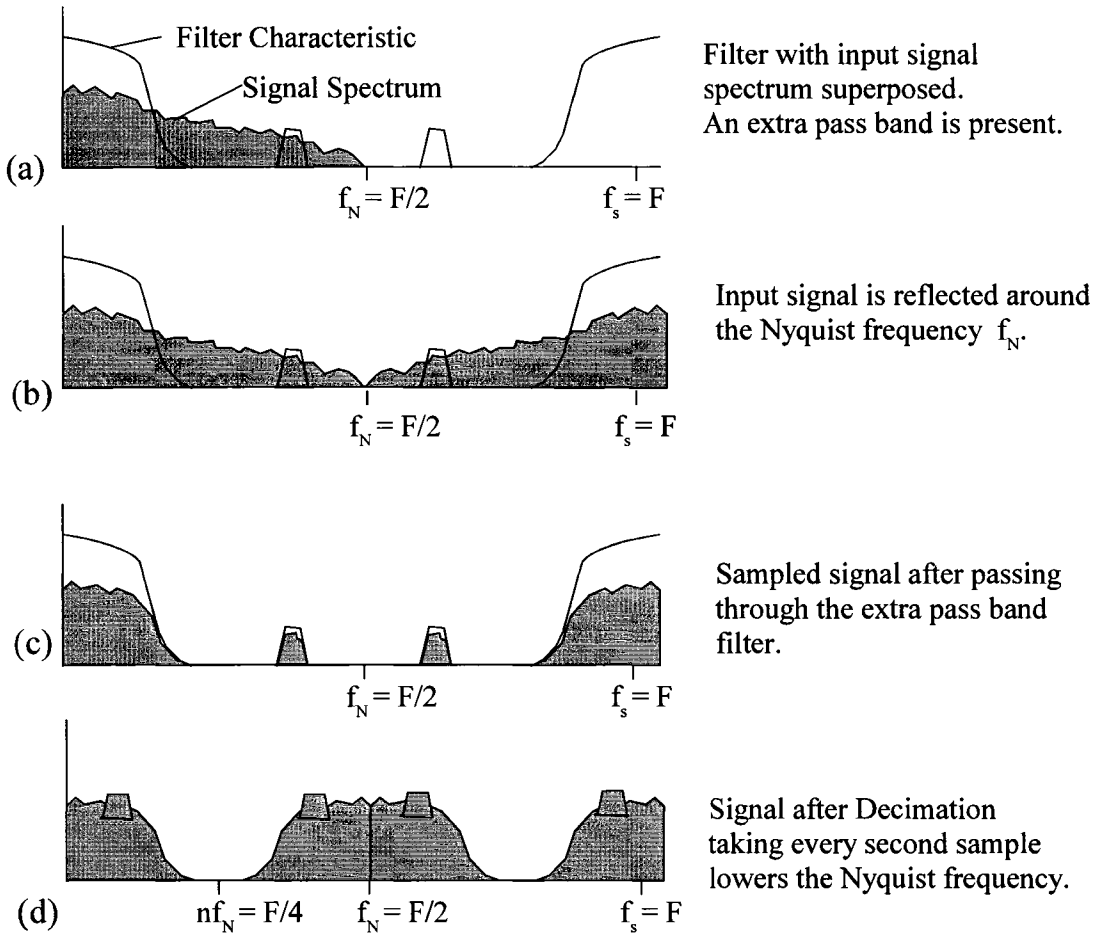

Figure 3.3.1: Decimation Low Pass Filter with Extra Pass Band Above nf_N (New Nyquist Frequency).

The only part of the signal left is that within the low-pass end and that inside the extra pass band. Now when this signal is decimation by 2, taking every second sample lowers the aliased

spectrum. The aliased signal appears in the desired spectrum below $nf_N = F/4$, as shown in part (d). Hence the extra passband must be removed before decimation.

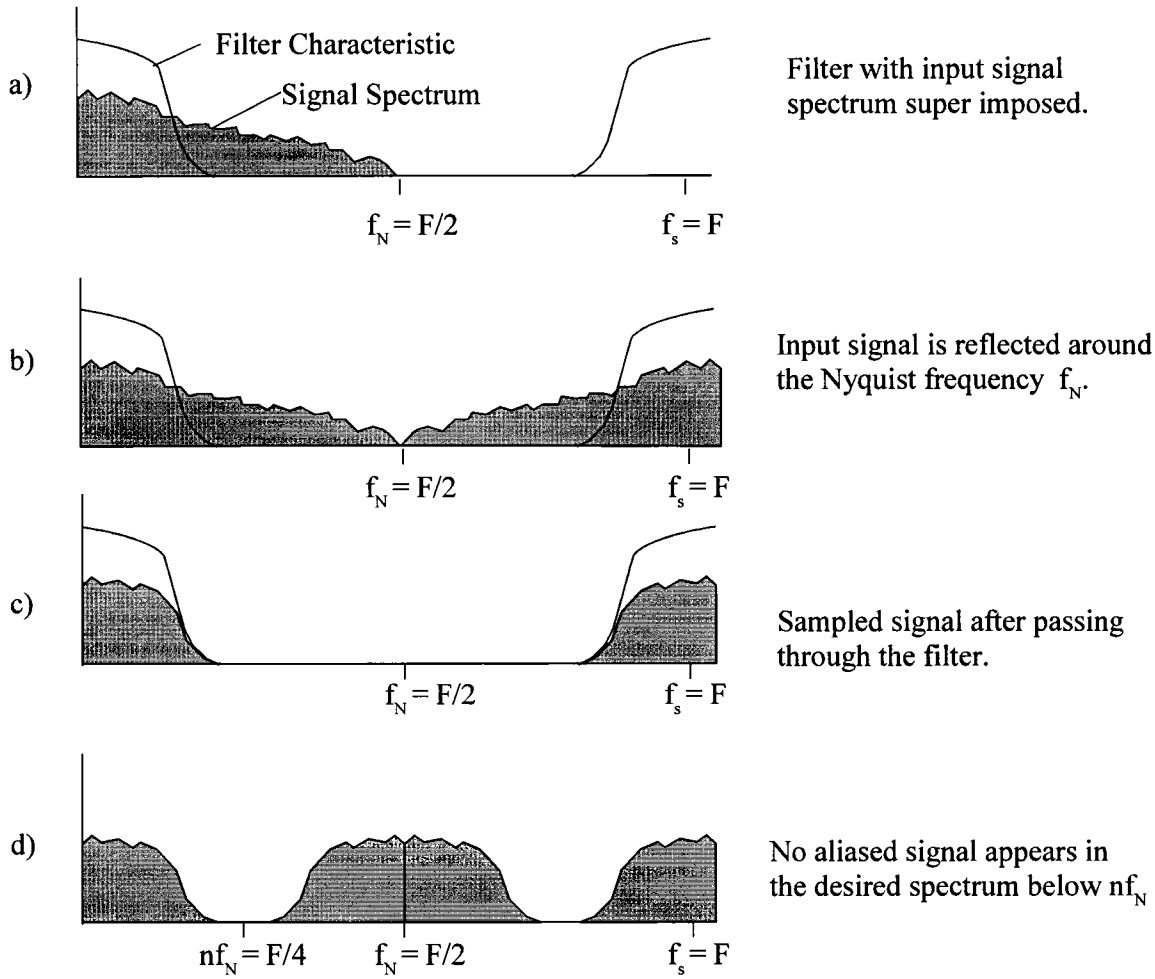

Figure 3.3.2: Decimated Low pass filter with cutoff below F/4.

Figure 3.2.2 shows the response of the filter without an extra passband, part (a). There is a clean spectrum after the low pass filter part (c). Due to this we do not have any aliased signal in the spectrum after decimation by 2 in section (d). Thus the filter must be a low-pass filter with negligible output above F/4. Otherwise the decimation sampling will alias frequencies above F/4 to below F/4.

3.4 Decimation Before and After Filtering.

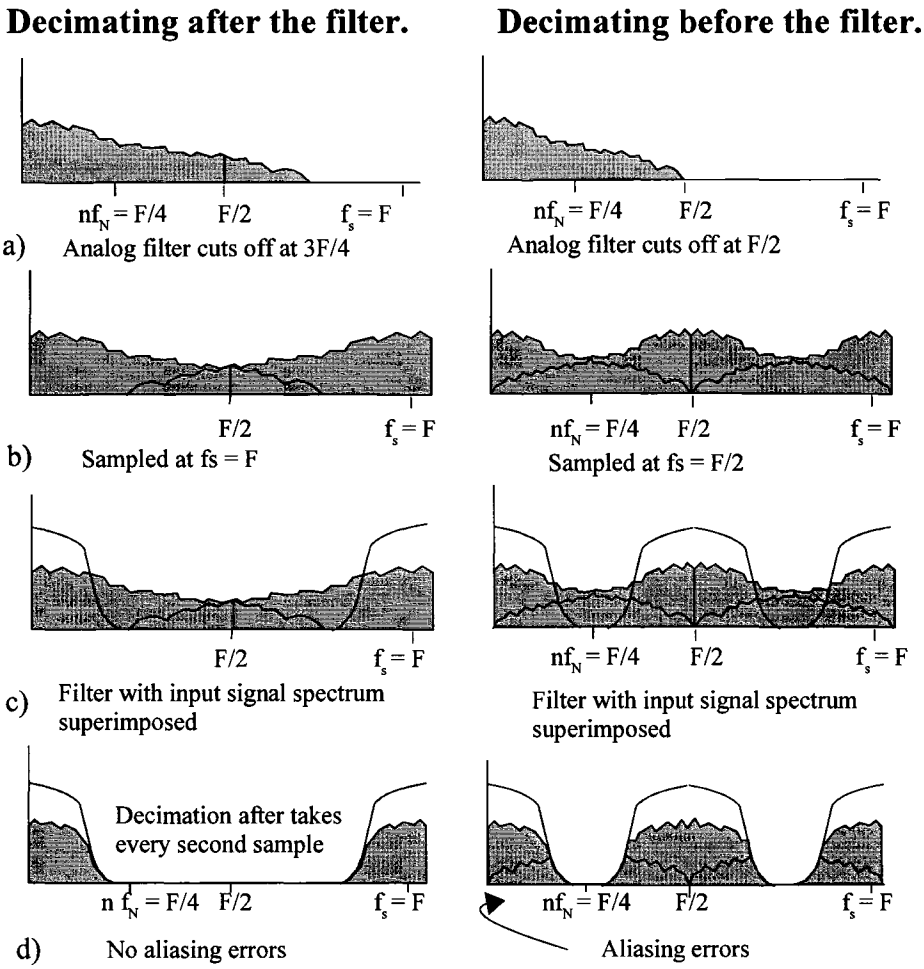

Figure 3.4: Difference Between Decimation Before and After the filter.

Figure 3.3 shows a comparison between decimation before and after the filtering. In decimation before filtering, every second input sample is taken, in decimation after the filtering we take every second output sample. In the graphs of part (a), the right side is filtered more than the left side, but this filtering is not enough. The filter at right side is sampled at F/2 which causes aliasing as shown in the part (b). Part (c) shows the filter response superposed on the input signal spectrum. Section (d) shows the final filtered output. The filter in which decimation was done

before filtering has aliasing errors, whereas the filter in which decimation was done after filtering has no aliasing errors.

Usually low-pass filtering of the signal is required prior to down sampling in order to enforce the Nyquist criteria at the post-decimation rate. For example, consider a signal sampled at a rate of 300 MHz, whose highest frequency component is 100 MHz; this is less than the Nyquist frequency of 150 MHz. In order to reduce the sampling rate by a factor of three to 100 MHz, there must be no component greater then 50 MHz, which is the Nyquist frequency of the reduced rate.

Since the original signal has component of 100 MHz, it must be low-pass filtered prior to down sampling in order to remove all the components above 50 MHz.

3.5 Half-band Filters

A half-band filter is a subset of the FIR filter family. A half-band filter has a frequency response symmetric about $f_N/2$. A half-band filter has only an odd length impulse response.

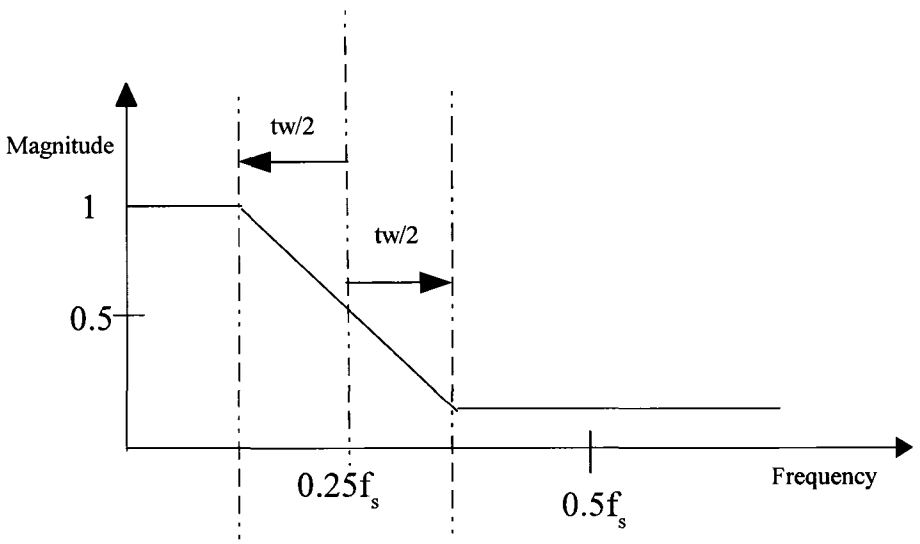

Figure 3.5: Magnitude Response of a Half-band filter

Furthermore the odd coefficients are zero except the middle one, which is equal to 0.5 [14]. Because of the zero coefficients, fewer taps are required, which reduces the hardware and consumes less power.

Half-band filters have a frequency response of 0.5 at the frequency of $0.25f_s$. Thus as in Figure 3.5 frequencies up to $\{0.25f_s + tw/2\}$ will alias down to $\{0.25f_s - tw/2\}$, where *tw* is the transition band width. This means half-band filter must be followed by a low pass filter which has negligible response above $\{0.25fs - tw/2\}$.

3.6 Multistage Decimation Filters

In filter design a narrow transition width is usually desired. However this requires many taps. To reduce the number of taps some compromises have to be made. For example a filter with a cut-off frequency starting at 100 MHz with roll off between 100 MHz and 200 MHz uses fewer taps than a filter with a cut-off frequency starting at 100 MHz with a roll off between 100 MHz and 150 MHz [13].

Using a multistage filter for decimation can reduce computation and storage [23]. A multi-stage filter can be used to achieve a certain specification by performing decimation between stages. The first stage decimating filter can relax the specification for the subsequent filter, and can be used as a pre-filter to remove out of band noise; more stringent specifications can be accommodated by the subsequent filters.

When decimation is done in the first stage filter, it reduces the output data rate of this filter, which causes a reduction in the input data rate of the subsequent filter. In FIR filters if the input data rate of the filter is reduced to half, then the relative transition width of that filter will be reduced to half of the relative transition width for the maximum data rate.

Figure 3.6 shows the response of two identical filters with different sampling rates. The filter in plot (a) has a f_N of 400 MHz with a transition width of 100 MHz to 200 MHz.

Plot (b) shows the response of an identical filter where the f_N has been reduced by a half to nf_N i.e., from 400 MHz to 200 MHz. In this case, for the same *N* number of taps, the transition width will be reduced to half, i.e., from 50 MHz to 100 MHz.

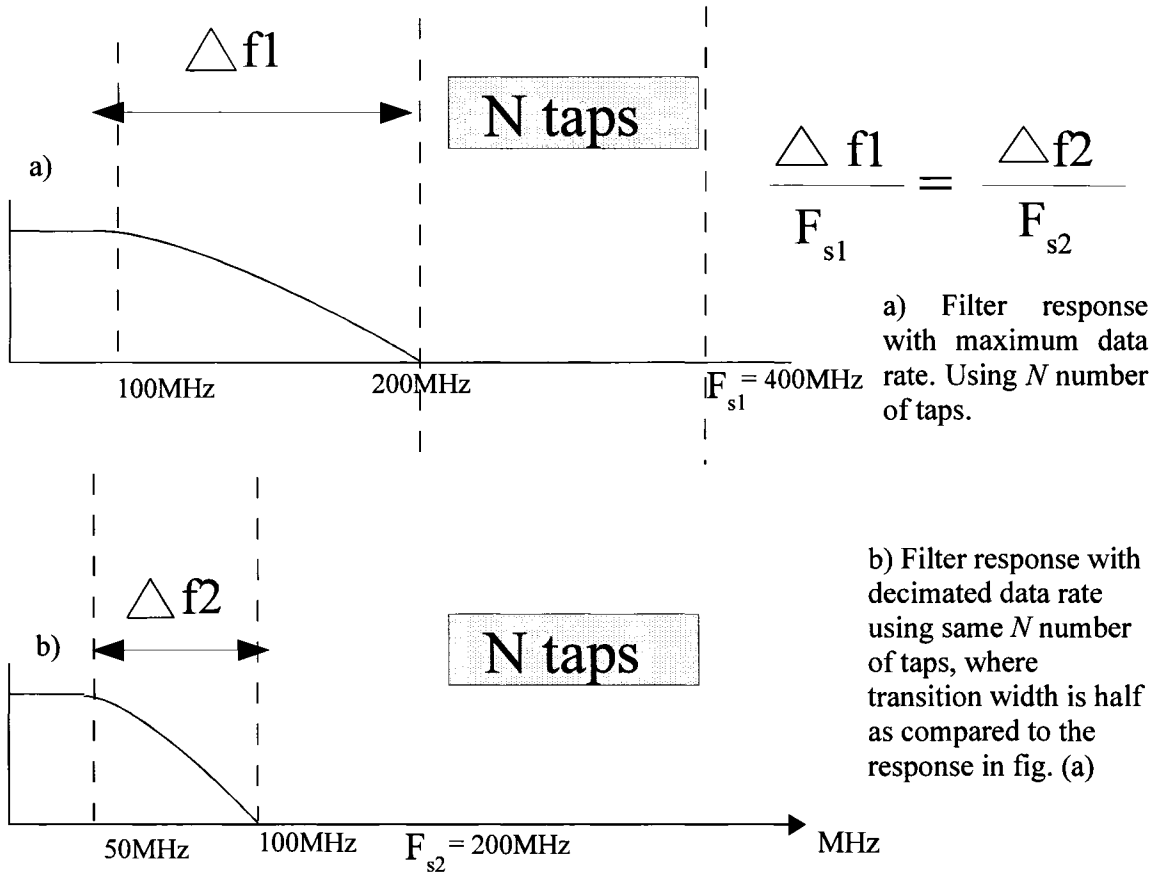

Figure 3.6: Transition Width Related to Number of Taps.

The power consumed by the filter is directly proportional to the number of taps as well as the operating frequency. Using multistage decimation filters, the total numbers of taps can be reduced. Subsequent filters will also operate at lower frequencies, thus further reducing power consumption.

Each filter removes only that part of the signal that would alias into the pass band of the next stage. Thus the relative transition band slope in each filter is less and each filter is much simpler. In particular, the analog filter can often be a single RC stage.

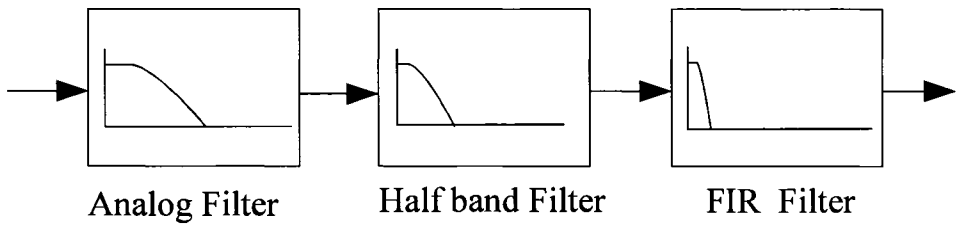

Figure 3.6.1: Multistage Decimation Filter.

Figure 3.6.1 shows the operation of a multistage decimation filter, where several filter stages have been used to reduce the total number of taps. Each filter removes the part of the undesired signal that will alias into the desired band. Therefore the transition band of the first stage filter could be a much wider percentage of the sampling rate. This first stage will be an analog anti-aliasing filter; the wide transition region means it can be simpler and less expensive.

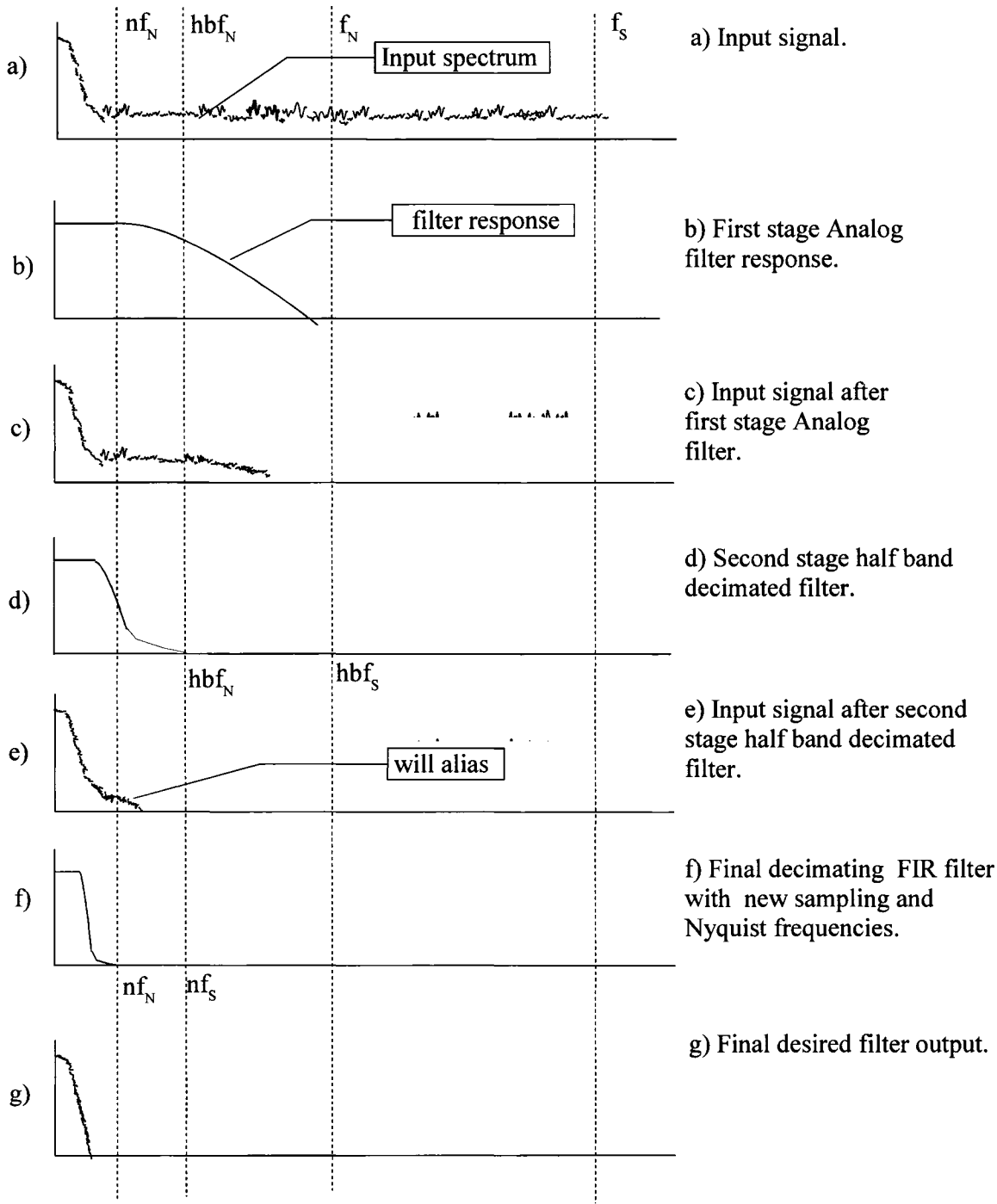

Figure 3.7: Multistage Decimation Filter Example I.

3.7 Multistage Filter Example I

Figure 3.6 shows the signal and filter response of the different stages of a multistage decimation filter. Part (a) shows the input signal, where f_s is the input sampling frequency and f_N is the input signal Nyquist frequency. This signal is fed into a first stage analog filter. Part (b) shows the filter response of the first stage analog filter. Part (c) shows the output of the first stage analog filter. Here the signal will be sampled at frequency f_s. F_N is the associated Nyquist frequency. Everything above f_N has been removed by the first stage filter.

The signal shown in Part (d) is the response for the next stage filter. The Nyquist frequency f_{NY} will become the input sampling frequency of this second stage filter, which is a half-band filter. This is a low-pass filter which passes half of the available output bandwidth, and is discussed in section 3.4. The output sampling frequency of the half-band filter is denoted by hbf_s and its Nyquist frequency is denoted by hbf_N. The half-band filter will decimate the output of first stage filter by 2.

Part (e) shows the output of the second-stage half-band filter. There are still remnants of the signal above the half-band Nyquist frequency. This signal is sent to the third-stage filter. This means that the Nyquist frequency of the half-band filter output will become the input sampling frequency of third stage filter. Some signal appears above nf_N which will alias back below nf_N. The final low-pass FIR filter must remove this aliasing.

Part (f) shows the filter response of the third stage regular FIR filter, where nf_s represents the new sampling frequency at the output and nf_N is the new Nyquist frequency.

Part (g) shows the output signal of final stage FIR filter. The frequencies of this signal are very low compared to the original input signal. An expensive high order filter is required for such a sharp frequency response at the original sampling frequency f_s, but multistage filtering has made it much smaller and lower power. A practical example of such a filter is presented in the following section.

3.8 : Multistage Filter Example II

This example from Analog Devices elaborates the benefits of two stage filters with half-band filter performing the decimation. The filter in this example is designed for an application with an input clock frequency of 1.2 MHz which requires a low pass FIR filter with a cutoff frequency of 50 kHz and a stop band frequency of 100 Hz, a pass band ripple of 0.01 dB and a stop band attenuation of 120 dB. In order to design a filter with these specifications a 189 taps single-stage filter is required [13].

However these specifications could be easily achieved by a two-stage decimation filter with less than half the number of taps, as shown in figure 3.8.

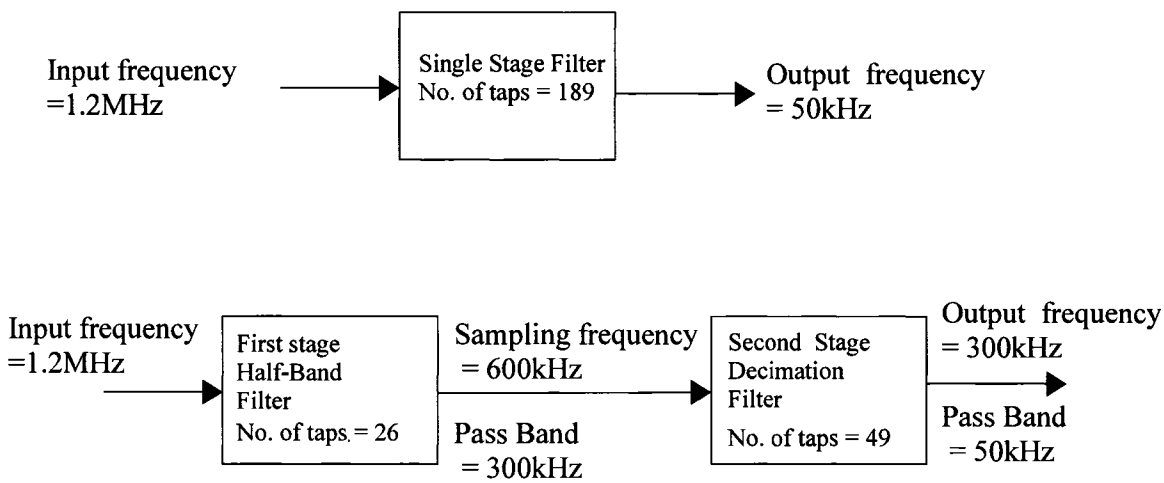

Figure 3.8: Decimation Filtering in Stages.

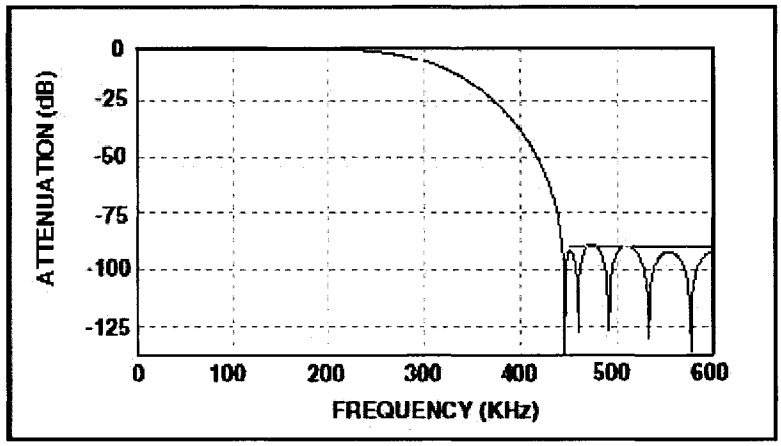

Figure 3.8.1: Half-band Filter Response.

The first stage filter is a half-band filter with a bandwidth of fs/4 and a wide transition width. Hence if we use a clock frequency of 1.2 MHz the band width of the half-band filter will be 300 kHz. Figure 3.8.1 [13] shows the response of half-band decimation filter. The cost of this filter is 52 taps which is reduced to only 26 taps only because half of the tap weights are zero in a half-band filter.

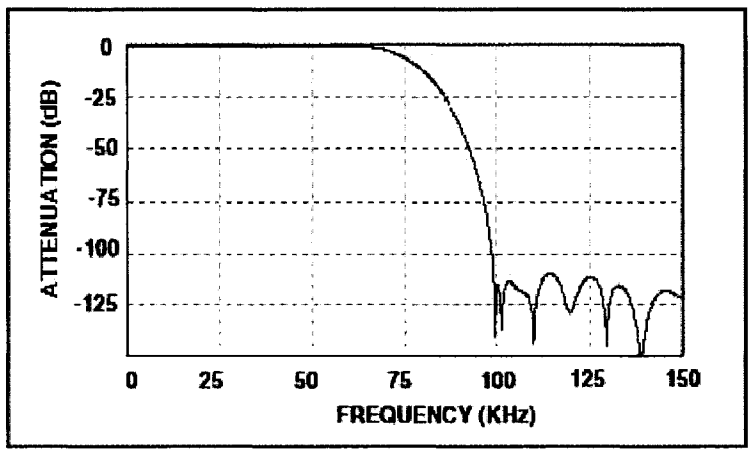

Figure 3.8.2: 2nd Stage Low Pass Filter Response.

After the first stage of digital filtering, the data stream is decimation by two which makes the input data rate to the 2nd stage filter 1.2 MHz/2 or 600 kHz. The goal was to achieve a narrow transition width in the second stage. This is now possible having done decimation in the first stage. Now fewer of taps will be required to achieve the desired transition width.

With a cutoff frequency of 50kHz and a stop band frequency of 100 Hz, the number of taps used in the 2nd stage is 49. Figure 3.8.2 [13] shows the filter response of 2nd stage FIR filter. If we add the number of taps in both stages then it will become 75. This is a big saving as the number of taps for the same specification with a single stage filter was 189.

3.9 Summary

To avoid aliasing decimation should be done after filtering. Decimation filters are an important branch of FIR filters. Multistage decimation is advantageous because the desired filter response could be achieved with a fewer number of taps. Half-band filters are often used in multistage decimation filters because all odd tap are zero except one. Multistage decimation is used to simplify low-pass filters with oversampled data. A filter sampled at $f_s/2$ has half the number of taps for the same transition band width as one sampled at f_s. Thus lower cost sharp low-pass filters can be build using decimation.

CHAPATER 4 Implementation of Decimation FIR Filters

4.1 Introduction

This chapter gives analysis of different design techniques for decimating FIR filters. In the beginning decimation filter design is described for both direct and transposed forms. After that more practical approaches are discussed. These designs are taken both from industry and the literature. A new graphical method of design analysis using a 'time-space diagram' is also introduced. The later part of this chapter explain the newly designed latch-based FIR filter and some practical implementation of half-band filters as well.

4.2 Simple Decimation Filter

As discussed earlier, decimation filters have a sample rate at the output of 1/M that at the input, where M is the decimating factor. For example if M=2, one takes every other sample. To avoid aliasing, the filter must be low pass and have negligible response at Fs/2M, where Fs is the original sampling frequency. For M = 2, a block at the end selects every second sample. The decimation output will have a Nyquist frequency of f_N = F/4. Figure 4.2.1 shows the direct form. Figure 4.1.2 shows the alternate direct form which will be used for the following explanation.

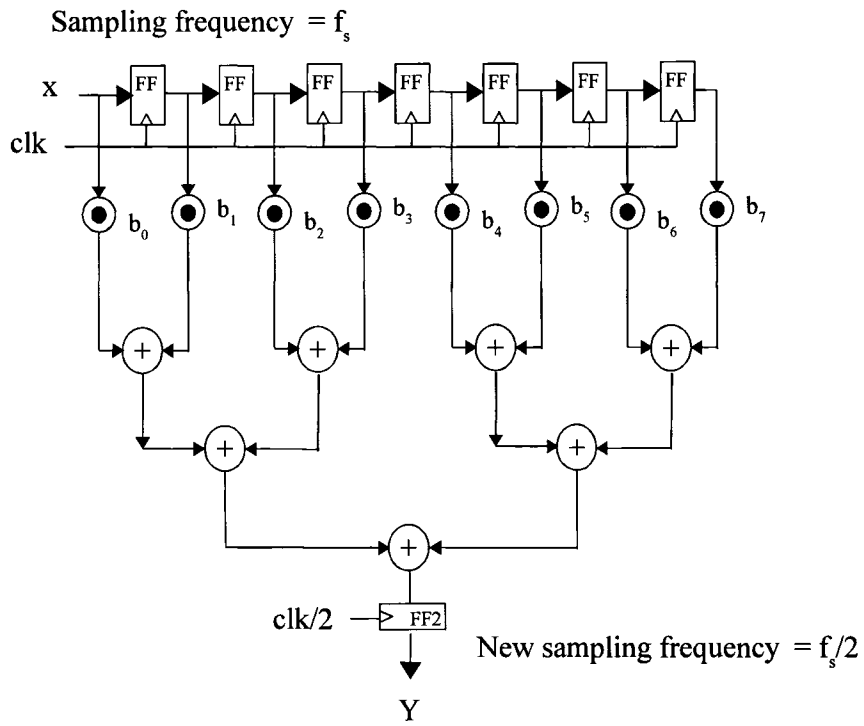

Figure 4.2.1: Simple Decimation Filter Using Direct Form.

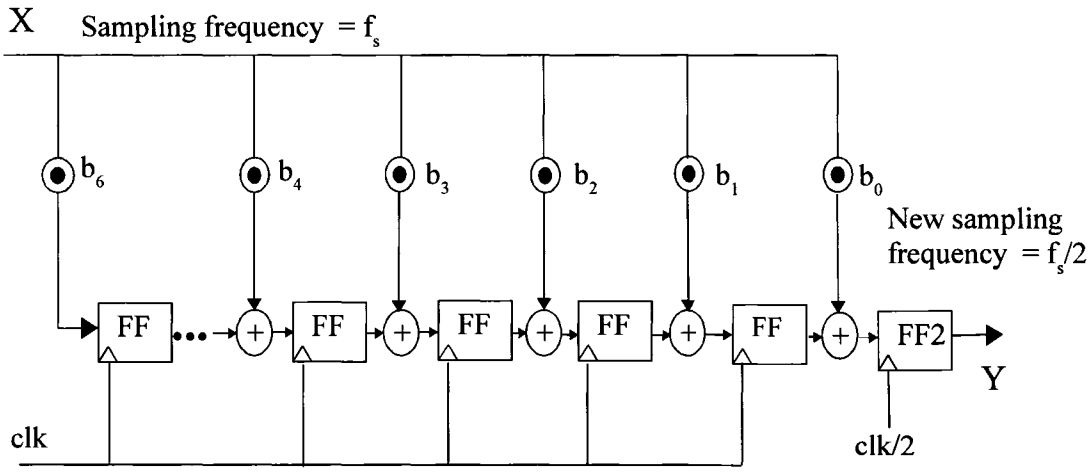

Figure 4.2.2: Simple Decimation Filter Using Alternate Direct Form.

Figure 4.2.2 shows an 8 tap FIR filter where $\{b_0,b_1,....b_6\}$ are the coefficients of the filter. The delay elements are created using flip-flops, activated on the rising clock edge . If $\{x_0, x_1, x_2, ...\}$ are the input samples, then the normal output $\{y_0, y_1, y_2, ...\}$ is given by the following equations.

$$y_0 = ... + b_6 x_{n-6} + b_5 x_{n-5} + b_4 x_{n-4} + b_3 x_{n-3} + b_2 x_{n-2} + b_1 x_{n-1} + b_0 x_{n-0} \quad (4.1)$$

$$y_1 = ... + b_6 x_{n-7} + b_5 x_{n-6} + b_4 x_{n-5} + b_3 x_{n-4} + b_2 x_{n-3} + b_1 x_{n-2} + b_0 x_{n-1} \quad (4.2)$$

$$y_2 = ... + b_6 x_{n-8} + b_5 x_{n-7} + b_4 x_{n-6} + b_3 x_{n-5} + b_2 x_{n-4} + b_1 x_{n-3} + b_0 x_{n-2} \quad (4.3)$$

$$y_3 = ... + b_6 x_{n-9} + b_5 x_{n-8} + b_4 x_{n-7} + b_3 x_{n-6} + b_2 x_{n-5} + b_1 x_{n-4} + b_0 x_{n-3} \quad (4.4)$$

$$y_4 = ... + b_6 x_{n-10} + b_5 x_{n-9} + b_4 x_{n-8} + b_3 x_{n-7} + b_2 x_{n-6} + b_1 x_{n-5} + b_0 x_{n-4} \quad (4.5)$$

$$y_5 = ... + b_6 x_{n-11} + b_5 x_{n-10} + b_4 x_{n-9} + b_3 x_{n-8} + b_2 x_{n-7} + b_1 x_{n-6} + b_0 x_{n-5} \quad (4.6)$$

In the case of a decimation FIR filter, every alternate output sample is discarded. Only the following samples are needed.

$$y_0 = ... + b_6 x_{n-6} + b_5 x_{n-5} + b_4 x_{n-4} + b_3 x_{n-3} + b_2 x_{n-2} + b_1 x_{n-1} + b_0 x_{n-0} \quad (4.1)$$

$$y_2 = ... + b_6 x_{n-8} + b_5 x_{n-7} + b_4 x_{n-6} + b_3 x_{n-5} + b_2 x_{n-4} + b_1 x_{n-3} + b_0 x_{n-2} \quad (4.3)$$

$$y_4 = ... + b_6 x_{n-10} + b_5 x_{n-9} + b_4 x_{n-8} + b_3 x_{n-7} + b_2 x_{n-6} + b_1 x_{n-5} + b_0 x_{n-4} \quad (4.5)$$

$$y_6 = ... + b_6 x_{n-12} + b_5 x_{n-11} + b_4 x_{n-10} + b_3 x_{n-9} + b_2 x_{n-8} + b_1 x_{n-7} + b_0 x_{n-6} \quad (4.7)$$

This is simple, but not a efficient design. The computations done to calculate omitted output samples are wasted. The following sections discusses how to avoid doing those unneeded calculations, thus saving resources.

4.3 Interleaving Samples for Decimation.

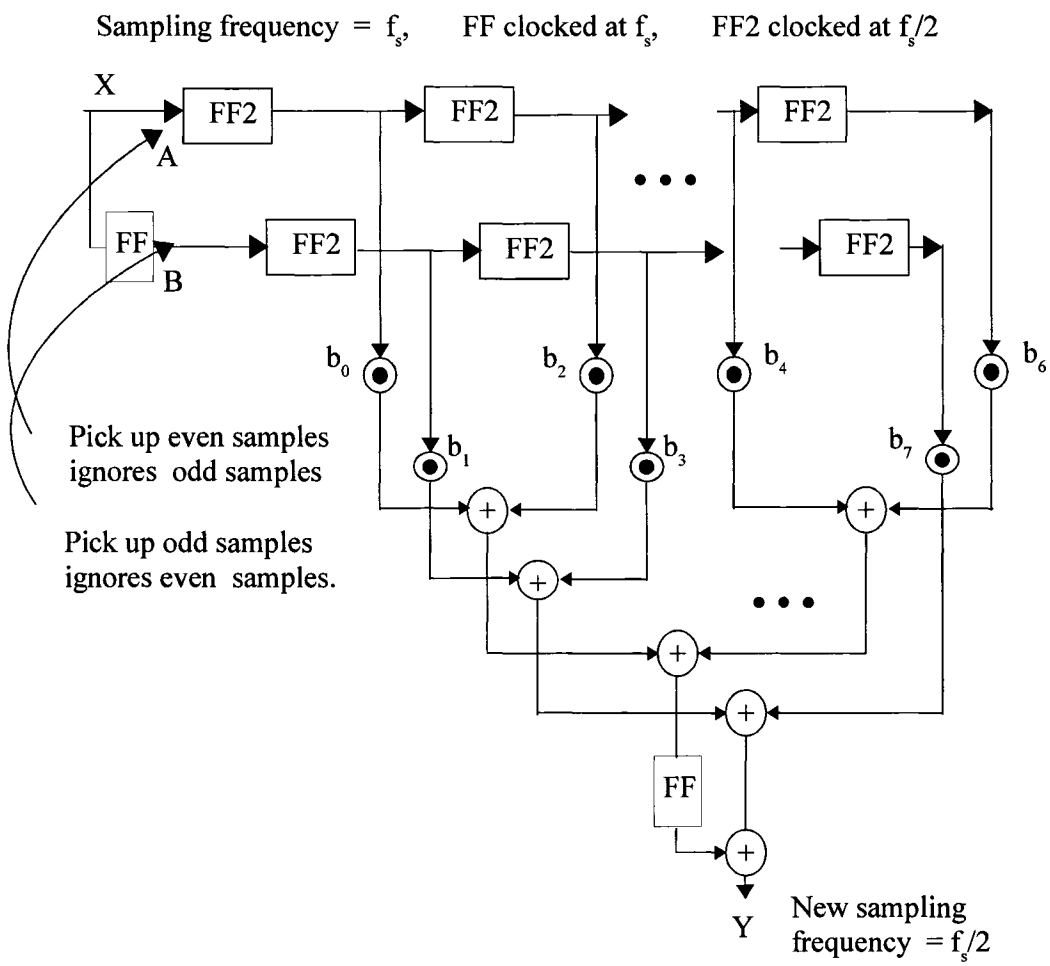

Figure 4.3: Interleaving Samples for Decimation.

Figure 4.3 shows a method of interleaving the samples to avoid redundant calculations. In this organization, the registers 'FF2' are clocked at half the sampling frequency. The 'FF' register selects odd samples at point B and sends them down the lower chain . The even samples are selected by the upper chain at point A. An extra 'FF' flip-flop is used at the end to match the timing. This shows how the samples are interleaved. One must not throw out any input samples. Another circuit using the same concept is described in the next section.

4.4 Decimation FIR filter Implementation I

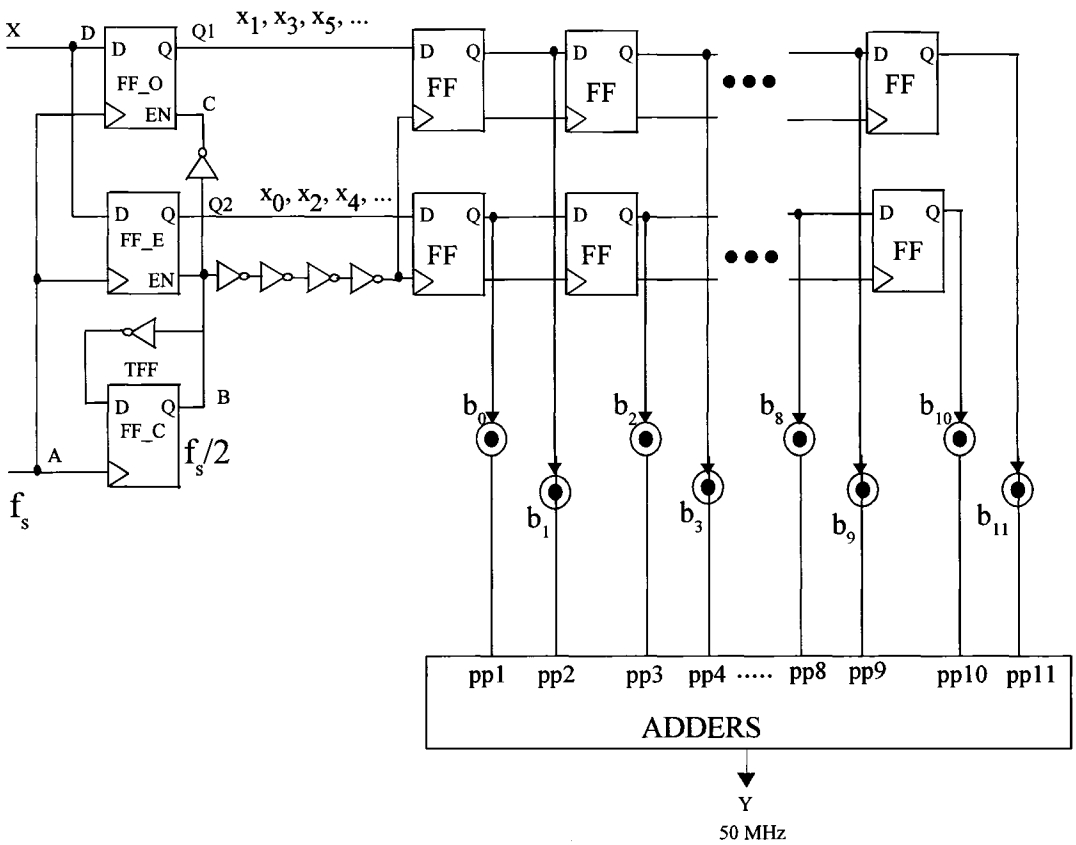

Figure 4.4.1: Decimation FIR Filter Implementation I.

A practical decimation filter (ALTERA Corporation) [12] using the direct form of FIR filter is shown in Figure 4.4.1. The filter uses some additional flip-flops for clock division and input sample manipulation. The inverter chain is used to ensure that the flip-flop chain is clocked only after Q1 and Q2 are stable.

In this direct form, the data is first delayed and then multiplied by the coefficients to produce products. These products are summed by a large adder at the bottom. This big adder could be removed by using the alternate direct form, which is discussed in the next section.

The input clock is divided by flip-flops FF_C (Figure 4.4.1). The output of FF_C serves as a clock for the entire filter at half of the input clock rate. If the input sample rate is f_s for example, then only FF_C must run at the f_s; the rest of the FIR filter runs at $f_s/2$. This $f_s/2$ clock is used to enable FF_E and, after an inverter FF_O. FF_E and FF_O have an input frequency of f_s but they are enabled at $f_s/2$. FF_E and FF_O are used to split the data into even and odd streams respectively. The clock position and data values at different points in the circuit are shown in Figure 4.4.2.

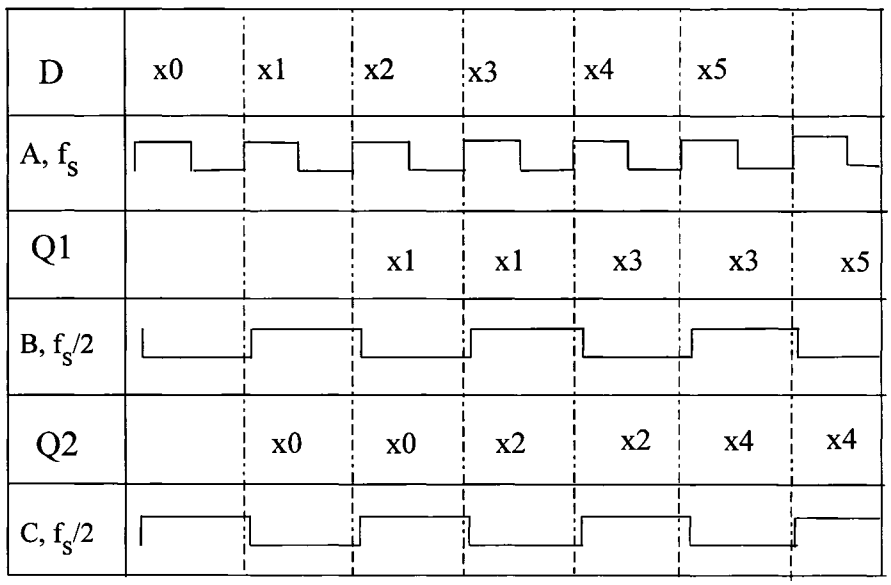

Figure 4.4.2: Data Sample Positioning at Different Points with Respect to the clock For Decimated FIR Filter Using Direct Form.

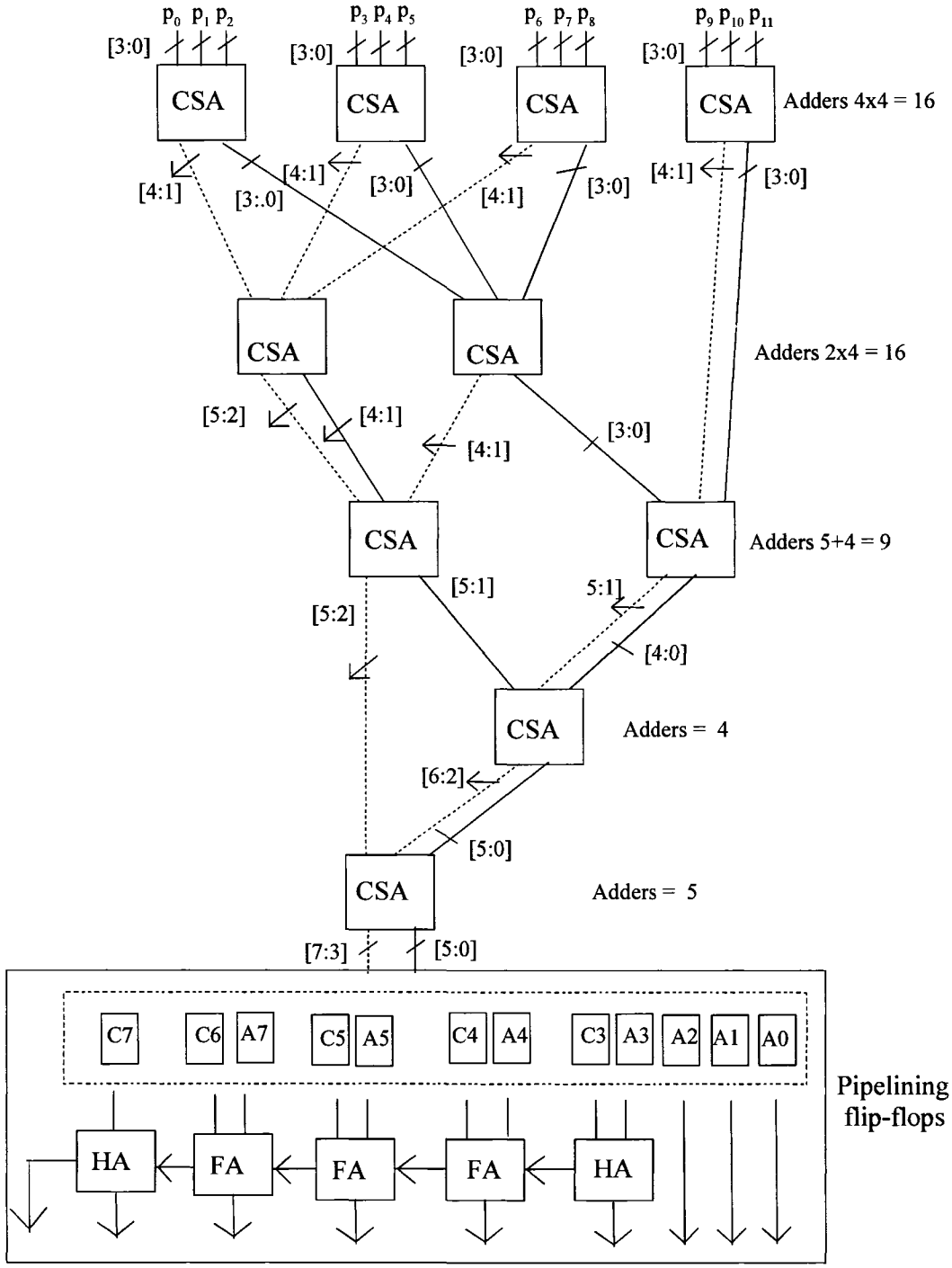

Figure 4.4.3: Carry Save Adder Tree with Pipelining.

The products p_0, p_1, p_2,... p_{11} are added by a carry-save adder tree. Figure 4.4.3 shows the organization of a carry-save adder tree. The difference between this Figure 4.4.3 and Figure 2.4.6.2 is that we have introduced pipelining in this carry save adder tree. This allows the clock to run faster, but it costs additional flip-flops. Furthermore this pipelining shown will still be slower than the alternate direct form in which the delay producing flip-flops are placed in between the adders. The next section discusses these type of filters.

4.5 Decimation FIR Filter Implementation II

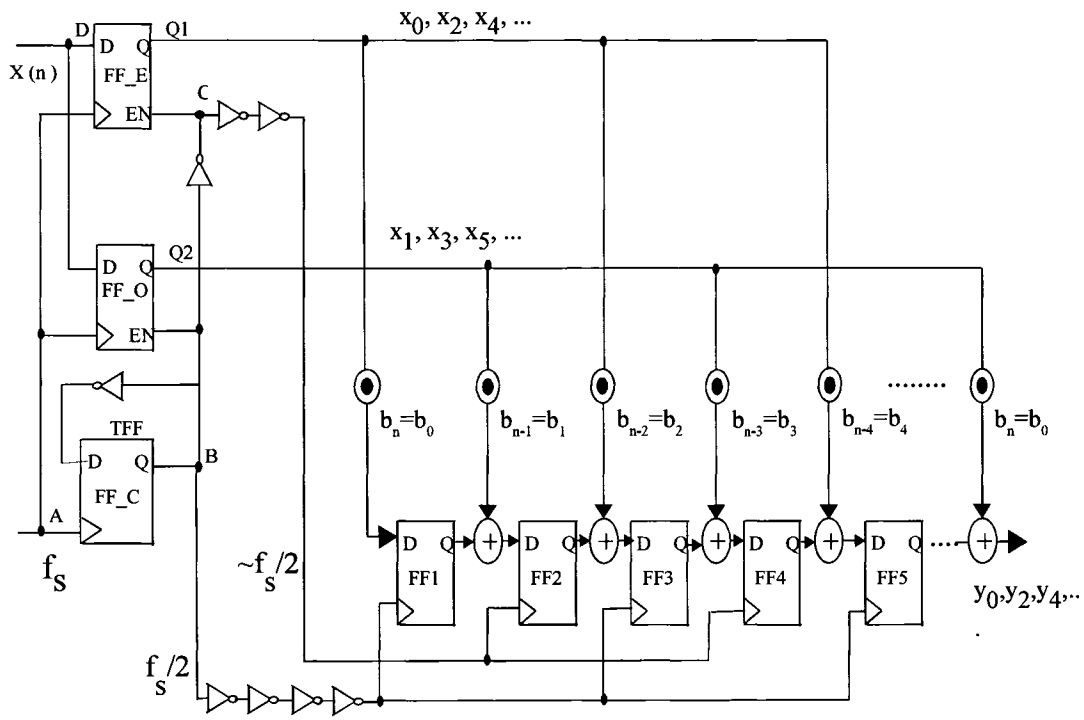

Figure 4.5: Decimation FIR filter Implementation II.

The circuit shown in Figure 4.5 is a decimating filter with a decimating factor of 2. The data streams are divided into even and add streams as done in Section 4.4. The flip-flops are working on alternate clock edge. The output contains only y_0, y_2, y_4,... (even output samples) because all the odd output samples are not calculated.

The big advantage of the alternate direct form is that the large adder used in the previous example is avoided. Thus this alternate form could be clocked faster because it does not have an adder tree delay.

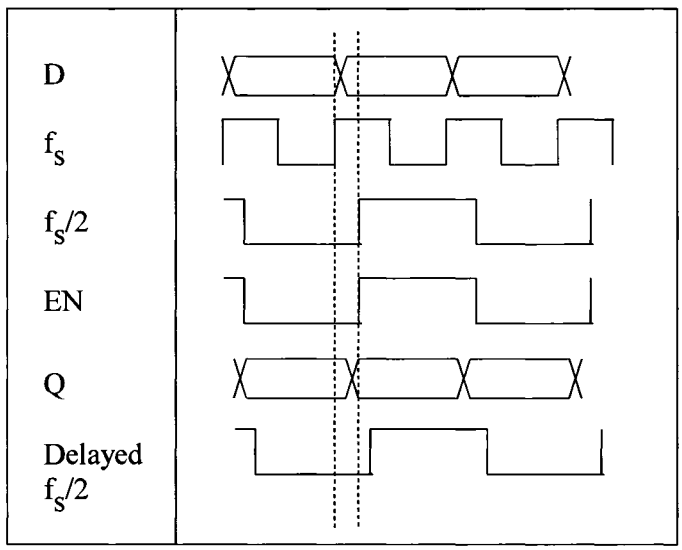

Figure 4.5.1: Clock and data timing for Decimation FIR II Implementation.

The clocking strategy is the same as for the decimation FIR filter implementation I (Section 4.4). The inverter chain is used to ensure that the flip-flop chain is clocked only after Q1 and Q2 are stable. They are needed because there is a race between Q1 and Q2, and the $f_s/2$ clock (Figure 4.5.1).

4.6 Decimation FIR filter Implementation III

The circuit shown in Figure 4.6 is taken from a paper [16]. It is comparable with the latch-based decimation FIR filter presented in this thesis in terms of required flip-flop / latch area and power. However it has a much slower maximum throughput.

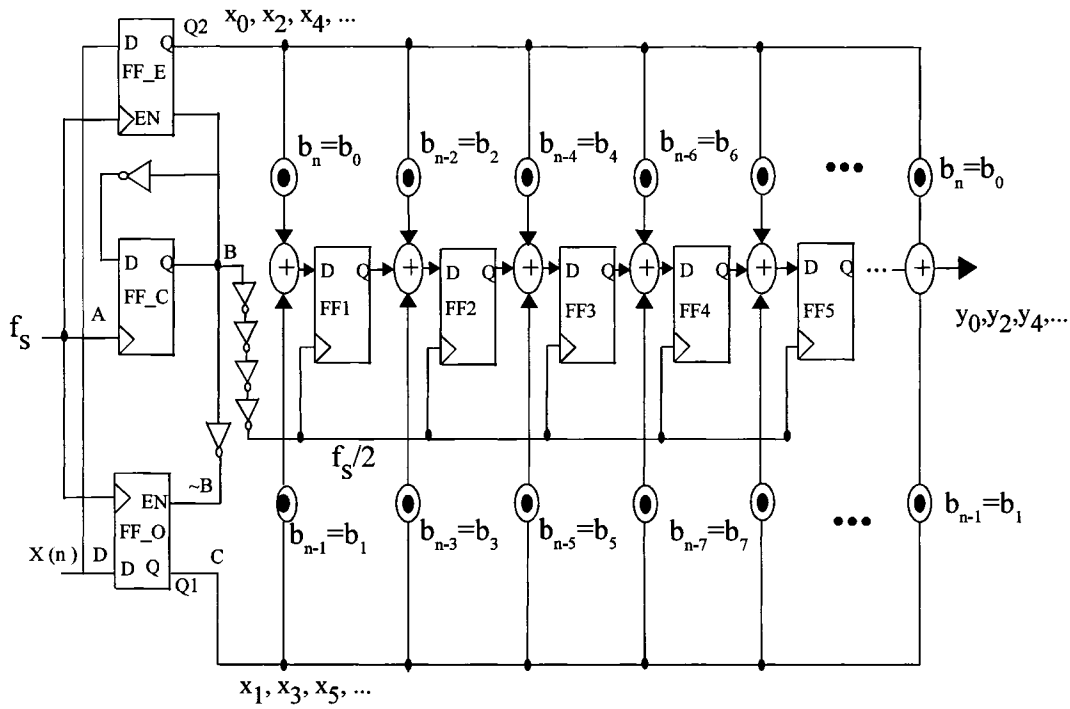

Figure 4.6: Decimation FIR Filter Implementation III.

The above circuit (Figure 4.6), decimates by 2. Clocking and decimation is done in the same manner as it was done for implementation I (Section 4.4) and implementation II (Section 4.5). FF_C divides the input clock frequency while FF_E and FF_O split the input samples into even and odd stream respectively. In this alternate direct form, there is one flip-flop for two taps so the storage area and power is reduced by a factor of two.

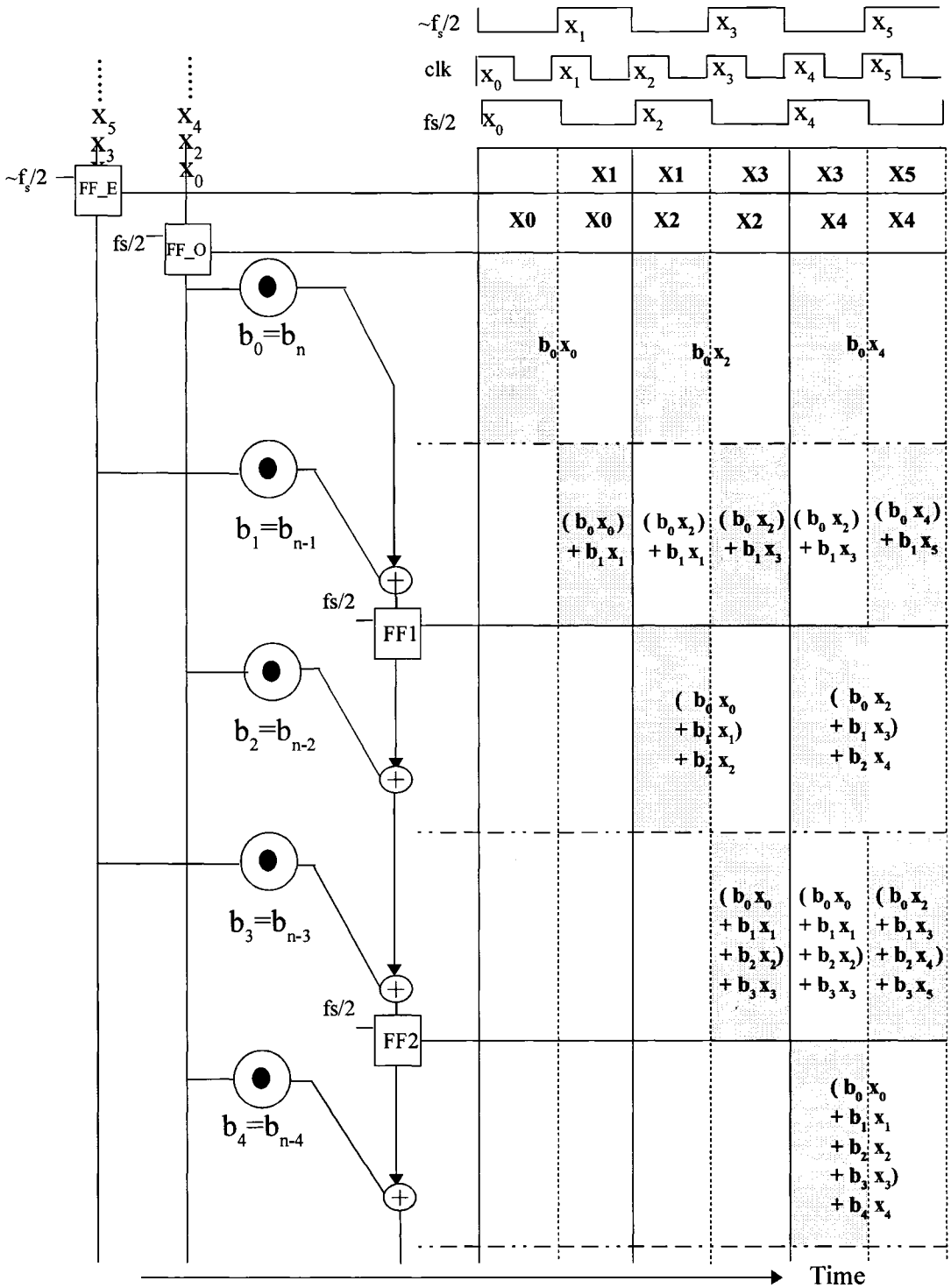

Figure 4.6.1: Time Space Digram for Decimation FIR filter Implementation III.

4.6.1 Time-space Diagram for Decimation FIR Filter Implementation III

This section introduces a novel graphical method for describing pipeline type operations. In this diagram the time is represented on the x-axis and the space (flow through the components of the pipeline) is represented on the y-axis. Figure 4.6.1 shows the data at any point in the circuit in a particular clock cycle. In the first row with shading, b_0x_0 is being calculated at the start of the circuit in the first two clock cycles. In the second shaded row b_1x_1 being calculated in the second clock cycle and the brackets indicate (b_0x_0) is transferred in from the row above.

Along the space axis, FF_E and FF_O are shown with their respective enables at B and ~B replaced by equivalent clocks. This simplifies the diagram and is acceptable if there is no clock skew between $f_s/2$ and f_s. Also the abstract 3-input adders are split into two realistic 2-input adders in this diagram. Since $b_0=b_n$, $b_1=b_{n-1}$, ..., in this figure we are using b0, b1, b2, ... to avoid writing the longer subscripts which cause clutter..

The circuit of Figure 4.6.1 is partially redrawn in Figure 4.6.2 with some details removed for clarity. Figure 4.6.2 shows how the signal flows inside the multiply and adder and how much time is consumed by the different modules.

At time T0, the flip-flop FF_E will hold the first sample x_0 which will appear at point A in the time-space diagram. It will be multiplied by b_0 through MUL0 during time T0. At T1 x_1 will appear at the output of FF_E located at point B in the time-space diagram. Then x_1 will be multiplied by b_1 taking about 75% of the time T1 until point C in the time-space diagram (assuming ADD2 will take 25% of T1). ADD2 will start adding the output of MUL0 and MUL1 at point C, so the output of MUL0 is not needed before point C. Hence the operation of MUL0 can be overlapped with the operation of MUL1 during T1 until point C.

The important thing to be noted here is that the operations of MUL1 and ADD2 cannot be extended beyond point D because this is the time for the next clock edge of $f_s/2$. Hence MUL0 has roughly twice as much time to multiply as compared to MUL1.

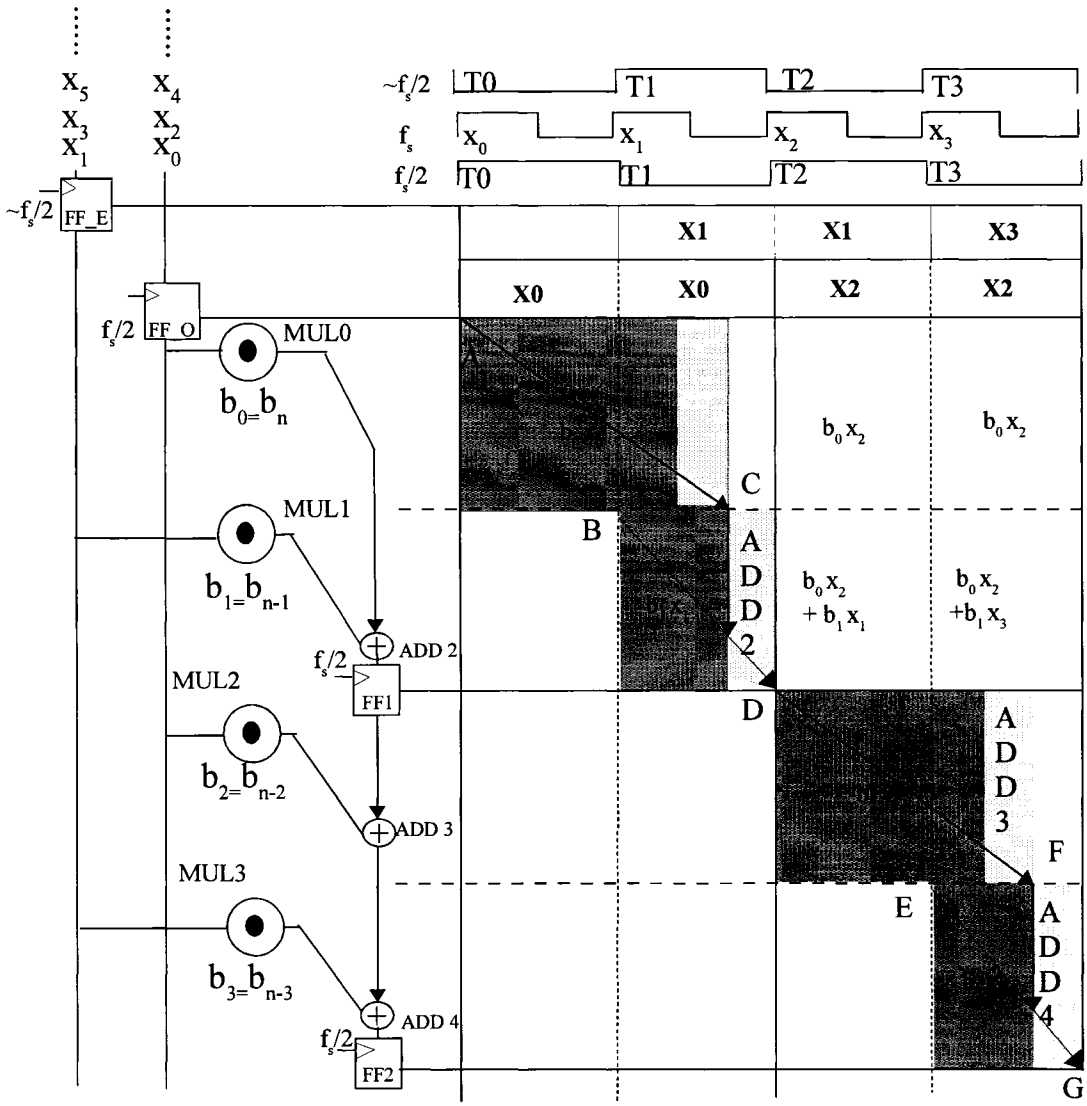

Figure 4.6.2: Detailed Time Space Diagram for Decimation FIR Filter Implementation III.

In the same manner the multiplication of x_2 with b_2 will start at point D. The multiplication is done by MUL2 during T2, but may be overlapped with MUL3 during T3. However MUL3 cannot be extended beyond T3, because of the clock edge of $f_s/2$. This phenomenon will continue through out the filter where every even multiplier has more time and every odd multiplier has relatively less time for multiplication.

Due to this the minimum time period of the clock will be determined by the multiply and accumulate (add) timing of the even multipliers and adders. Thus the filter does not get any benefit from the multiplication overlap advantage of the odd multipliers. This is the bottle neck of implementation III for designing regular decimation filters. However some filter with smaller odd coefficients can take advantage of this overlap multiplication.

4.7 A Novel Decimation FIR filter Implementation IV

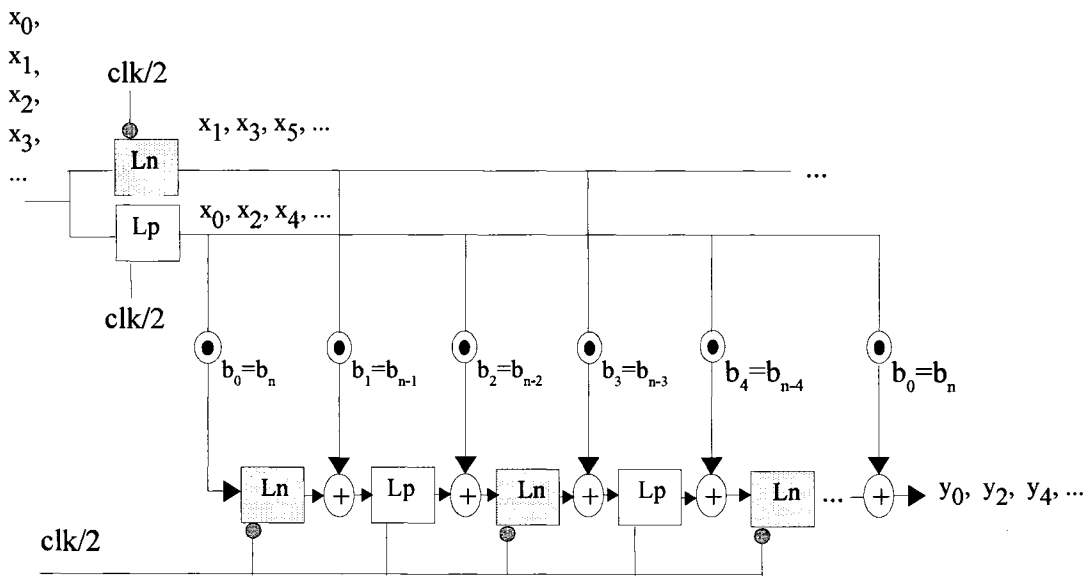

Figure 4.7: Newly Designed Latch Based Decimation FIR Filter Implementation IV.

The circuit in Figure 4.7 shows a novel technique for designing a decimation FIR filter It follows the principle of the transposed direct-form FIR filters and has the same transfer function as the design in Figure 4.6. The coefficients are symmetrical. Note that in Chapter 5 the actual implementation is done by folding to reduce the number of multipliers by half. The folding is not shown to make the explanation simpler and more understandable.

The basic difference between this circuit and previous implementations is that, level-sensitive transparent latches are used as delay producing elements, while the previous designs use edge-triggered flip-flops. The following section explains the detailed function of this new circuit.

4.7.1 Time-Space Digram for Decimation FIR Filter Implementation IV

The latch-based filter will be explained using the space-time diagram. In Figure 4.7.1, time is on the X-axis and space (signal travel along the pipeline) is on the Y-axis. In this Figure {N0, N1, N2, ...} are transparent-low latches where {P0, P1, P2, ...} are transparent-high latches. {b_0, b_1, b_2, ...} are filter coefficients. The 'L' at the top of a box shows that the data entering on the top of the 'L' is latched; the output stays constant, regardless of changes in the input, until the clock phase changes. 'T' shows that the latch on the left side is in the transparent mode and the contents of the box can proceed down to the the next box in the space direction. Because the coefficients are symmetrical, we use b_0, b_1, b_2, ... instead of b_n, b_{n-1}, b_{n-2} ... when showing the signal flow in the time-space diagram.

The left column shows latches instead of the flip-flops shown in Figure 4.6.2. Note, that the data path has been split into two parts, i.e., the data is multiplied only with those coefficients which are necessary to get even or odd output samples. Those products which are not needed because of decimation are not calculated. Figure 4.7.1 shows the data flow inside the filter. The formula in the box is the data that will be presented to the output latch, at or before the end of the f_s cycle. The bracketed part is what is stored in, or transmitted through, the previous latch.

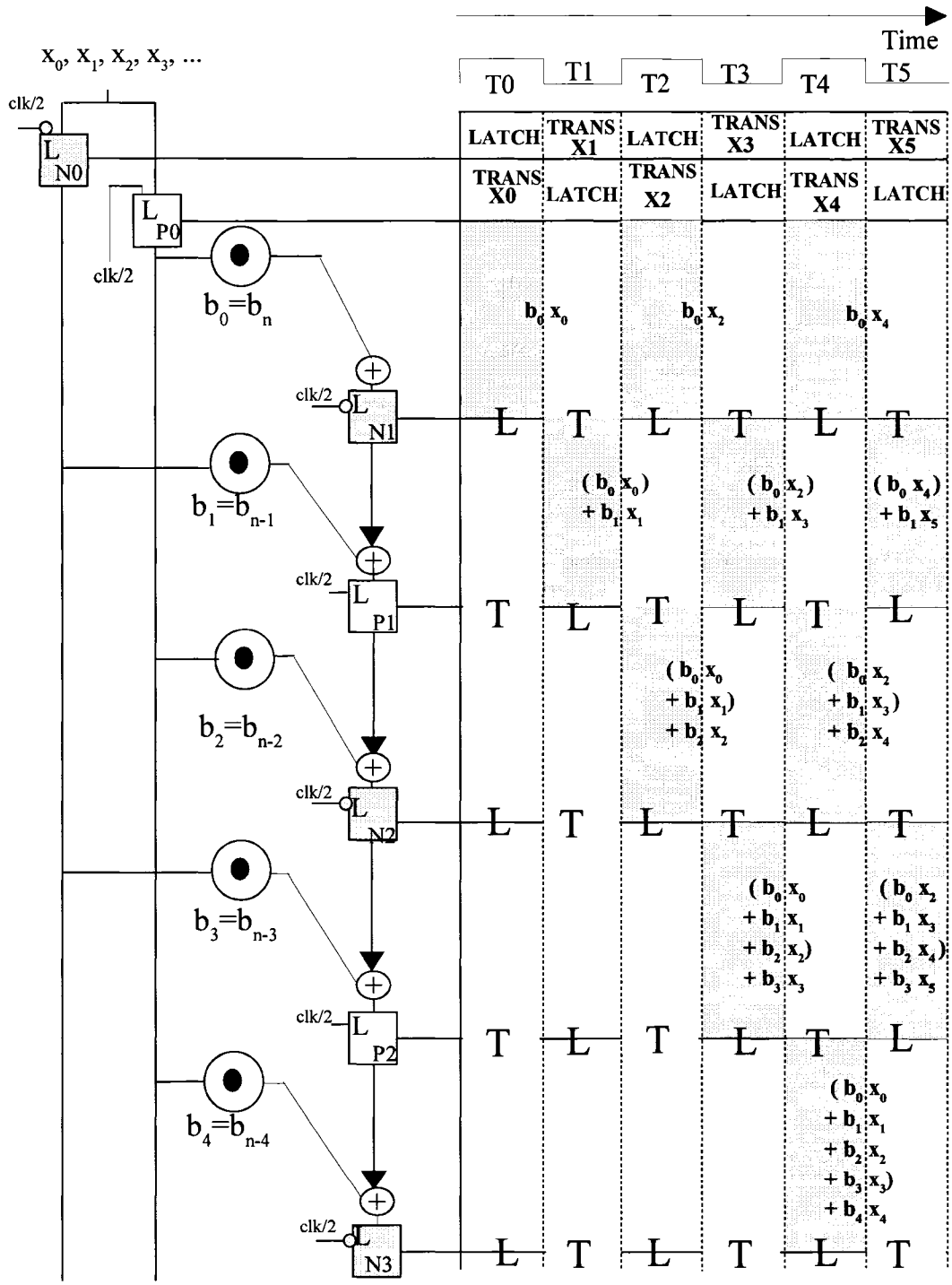

Figure 4.7.1: Time Space Diagram for Decimation FIR Filter Implementation IV.

The first two latches, N0 and P0, are used to make sure that the data synchronizes with the clock. The incoming data is then split and multiplied with a different set of coefficients. The data coming out of the latch P0 is multiplied with the even coefficients and the data coming out of the latch N0 is multiplied with the odd coefficients.

Let us analyze the circuit (Figure 4.7.1.1). At T0 the clock is at the positive level. At this point the latch P0 is transparent and the first data sample x_0 will pass through this latch and get multiplied with the first coefficient b_0. The product b_0x_0 is now ready at the input of latch N1 but cannot get through as N1 is in latch mode.

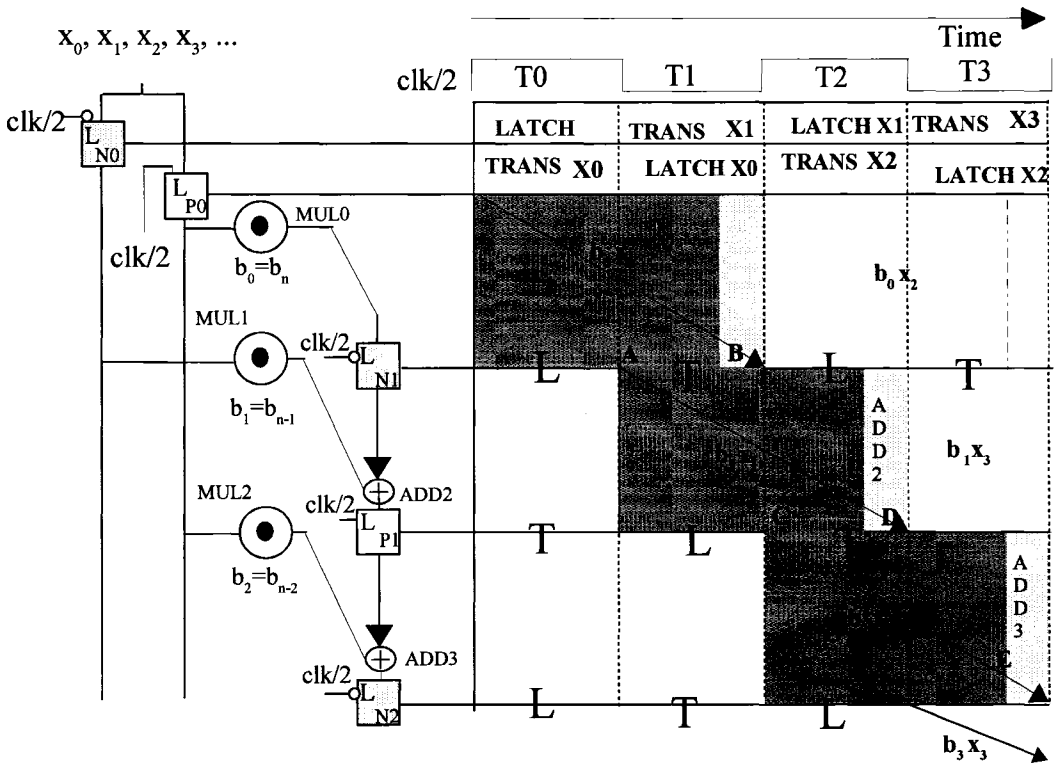

Figure 4.7.1.1: Detailed Time Space Digram for Decimation FIR Filter Implementation III.

At T1 the latch P0 will be in latch mode whereas N0 will become transparent. The second sample x_1 will pass through N0 and will be multiplied with b_1. This multiplication will take time and will be completed at, say, point B during T1. At point B, latch N1 is in transparent mode. The product b_0x_0 will now get through N1 and will be added through ADD2 to the new product b_1x_1. However b_0x_0 will be latched in T2 so the add in ADD2 does not need to be completed until point D. In the case of the FIR implementation III (Section 4.6), this addition had to be finished at point C as it has to be completed before the rising clock edge. This is the real advantage that this FIR IV implementation has over the FIR III implementation. In that implementation, the even coefficient multiplies can use most of the second cycle, but the odd multiply-add must be completed within a clock cycle.

At the start of T2, the new sample x_2 will be multiplied by b_2. The previous result $(b_0x_0 + b_1x_1)$ will be available during some of T2 and all of T3. Thus b_2x_2 only has to finish at point E in T3 to give time to the addition $(b_0x_0 + b_1x_1) + b_2x_2$. At clock phase T3, the new sample x_3 will start multiplying with b_3 resulting in the product b_3x_3. As discussed, this multiplication will take time inside T3 and T4.

T1, T2, ..., each represent one f_s cycle. Each multiply has 100% of one f_s cycle plus roughly 75% of the next f_s cycle to complete. This leaves 25% of an f_s cycle to do the addition. In reality many FIR filters combine the addition in the last stage of the multiply. This is called a multiply accumulate operation. It means the final answer is available when the multiply is finished so each multiply would have two f_s cycles to complete.

This process will go on for the rest of the taps. All the unused products like b_0x_1, b_1x_2, b_2x_3, b_3x_4 and b_4x_5 are not calculated, resulting in saving of power. The decimation output equations from section 4.2 are repeated for comparison. One can see them forming along the diagonals.

$$y_0 = \ldots + b_6 x_{n-6} + b_5 x_{n-5} + b_4 x_{n-4} + b_3 x_{n-3} + b_2 x_{n-2} + b_1 x_{n-1} + b_0 x_{n-0} \quad (4.1)$$

$$y_2 = \ldots + b_6 x_{n-8} + b_5 x_{n-7} + b_4 x_{n-6} + b_3 x_{n-5} + b_2 x_{n-4} + b_1 x_{n-3} + b_0 x_{n-2} \quad (4.3)$$

$$y_4 = \ldots + b_6 x_{n-10} + b_5 x_{n-9} + b_4 x_{n-8} + b_3 x_{n-7} + b_2 x_{n-6} + b_1 x_{n-5} + b_0 x_{n-4} \quad (4.5)$$

The replacement of the latches with flip-flops is a valuable advantage of this circuit over the circuits of implementation I and II. For example, if the data entering the filter is 8 bits long and the filter requires 100 taps (a reasonable length for a FIR filters), then 800 flip-flops are required. Every flip-flop roughly consists of two latches. So the number of latches will be 1600. If latches are used instead of flip-flops, the same operation takes 800 latches. This saves both circuit area and power.

Implementation III, requires only 400 flip-flops. Therefore the thesis circuit has only a speed advantage over implementation III. However implementation III could equal the speed of implementation IV in some cases where the odd coefficients are small as compared to the even coefficients. One example is a decimation half-band filters where the odd coefficients can be multiplied quickly. Such decimation half-band filters are discussed in the next section.

4.8 Decimation Half-band Filter implementations.

As discussed in Section 3.5, half-band filters have every odd coefficient zero except the middle one which has a value of 0.5. No multipliers are needed for odd coefficients and the one needed in the middle (0.5) is done easily by shifting the data bits right by one position. Figure 4.8.1 shows a half-band decimation filter designed with the help of latches.

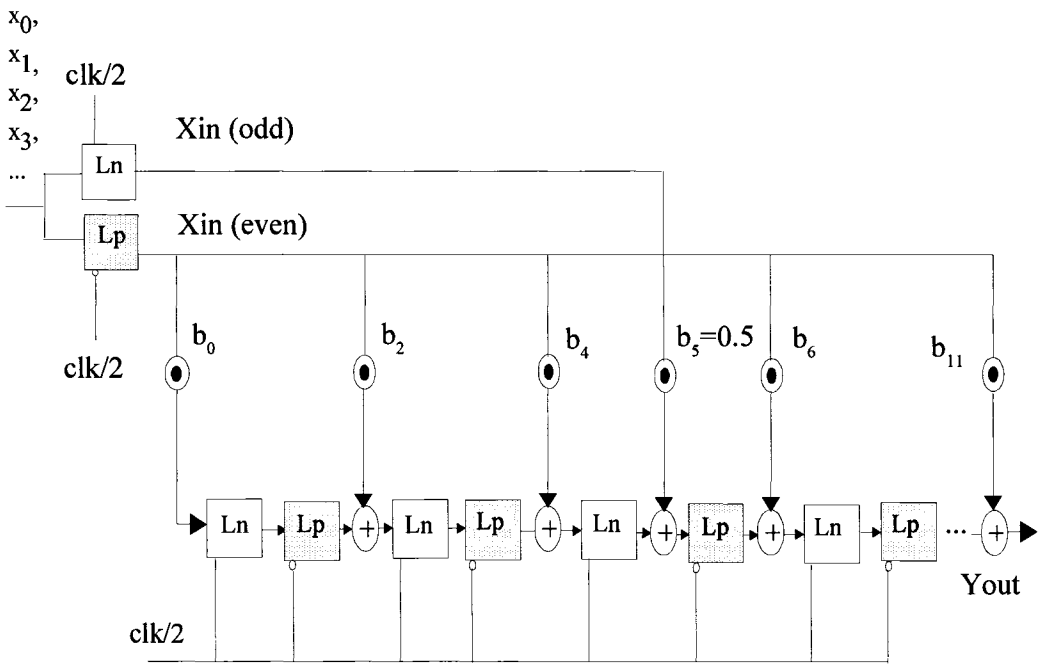

Figure 4.8.1: Latches Based Half-band Decimation Filter.

Most of the odd multipliers have been taken out of the circuit as every odd coefficient is zero. The odd stream has only one tap with a value 0.5, which can be done with a right shift.

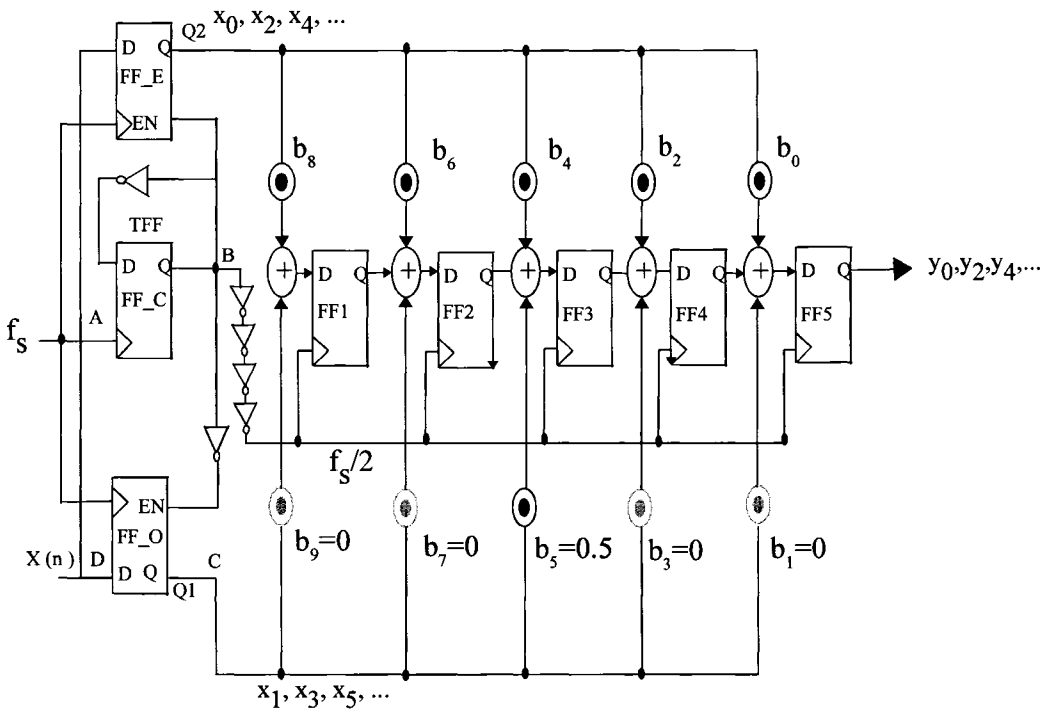

Figure 4.8.2: Half-band Decimation Filter with Implementation III.

Figure 4.8.2 shows a half-band decimation filter using implementation III. The multipliers shown with light gray color have been taken out of the circuit as every odd tap is zero accept the middle tap whose value is 0.5. Because every odd coefficient is zero this circuit will not have any bottleneck for odd samples, and the circuit could run as fast as a latch-based decimation half-band filter could run. Thus implementation III has only a two latch disadvantage over implementation IV for a half-band filter. Not enough for a designer to go through the hassle of using latches.

4.9 Summary

Four different implementations were discussed. All of the implementations used flip-flops as delay producing elements except implementation IV which used latches. The latch-based filter has half the storage area and power of the first two implementations. Its throughput is batter than implementation I and III but is matched by II. A new graphical method called a time-space diagram was introduced to visualize the timing analysis of the implementations. For half-band filters implementation III and IV are essentially equal.

CHAPTER 5 Hardware Design and Verification

5.1 Introduction

In this chapter the detailed hardware design and verification of a latched based FIR filter is discussed. A detailed gate-level design was done for two reasons. One is to verify that the filter functions as desired. The second is to show that it could be implemented using standard commercial tools and standard-cell libraries.

5.2 Design Synthesis for Latch Based Decimation FIR Filter

In digital design the circuit is first written in a HDL (Hardware Description Language); e.g., Verilog, VHDL. The code is then synthesized, with the help of synthesis tools. Different tools are available in the market for synthesis. These tools use pre-designed gate libraries (standard cells) as the target elements when they convert the Verilog code into a gate-level netlist. This gate-level netlist is a real hardware circuit description, the equivalent of a schematic diagram. It is used to run the simulations to verify the synthesis results, and to generate a circuit layout.

The latch-based FIR filter was developed using Verilog HDL and synthesized with the help of the Synopsis Design Analyzer. The synthesis was done using gates in an Artisan library designed for the 0.18μm TSMC (Taiwan Semiconductor Manufacturing Company) CMOS process. ModelSim was used to run initial simulation on the raw Verilog code. NC Verilog was then used to run final simulations with the synthesized gates. The other modules used in the verification, which were not the main filter core (For example 2's complement conversion) were written in Matlab or Simulink. See Figure 5.4.

5.3 Verilog for Synthesis

The Verilog code written for synthesis has two modules, a top module and a multiplier module. All the latches and adders are coded directly in the top module. In Verilog the adders could be synthesized by using a ' + ' sign, however a Verilog multiplier ' * ' cannot be synthesized directly. For this reason a separate multiplier module *var_sign_multiplier.v* (Appendix A) was developed and synthesized. Figure 5.3 shows the circuit connectivity between different modules including the test bench which was not synthesized.

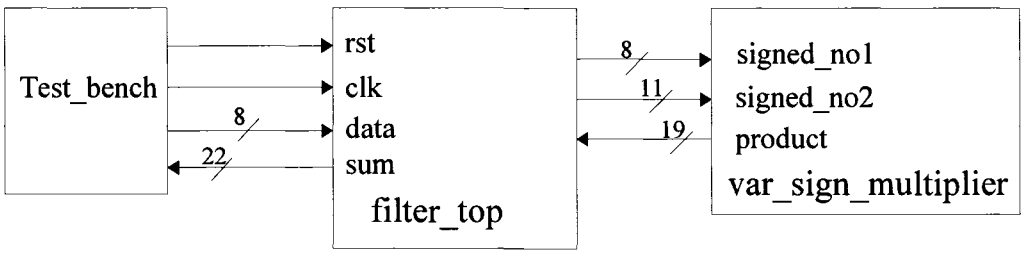

Figure 5.3: Module connectivity for synthesized latch based FIR filter.

Parameter	*Value*
Number of taps	33
Number of coefficient bit	11
Input word length	8
Output word length	23
Number of multipliers	17
Number of adders	32
Number of latches	34

Table 5.1: Design Parameters for Latch Based FIR Filter.

5.3.1 Module Description

According to its name *filter_top* is the top module of the FIR filter. The data entering the top module is 8 bit long and the coefficients used are 11 bits long. The output data consists of 23 bits (Table 5.1). The filter was designed using the folded transposed form. Therefore the the number of multiplier instances is halved. A signed-multiplier module was written to multiply the two given signed numbers. If an 8 bit binary number is to be multiplied with an 11 bit long number then maximum length of the product is going to be 19 bits long. Hence the multiplier module returns 19 bits. The accumulation of the products inside *filter_top* required another 4 bits to avoid truncation error on typical data. In a real application the 23 bit output would likely be truncated on the right, but here we did not want any truncation error inside the Verilog calculations that would cause the results to differ from the comparison filter done in Simulink. See Figure 5.4.

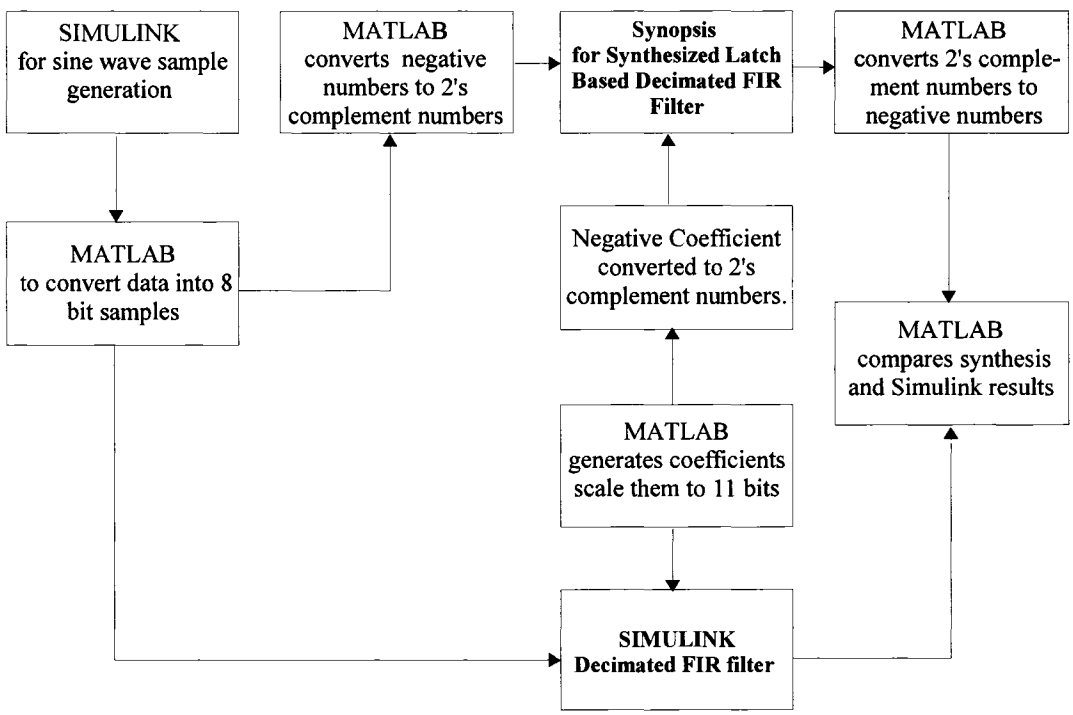

Figure 5.4: System Block Diagram to Compare Synthesis Results with Equivalent Simulink Results.

5.4 Verification of Synthesis Results

To verify the synthesis results of the latch-based FIR filter. The time response of the synthesized filter was compared to that of an identical edge-triggered register-based filter run under Simulink. The coefficients used to synthesize the latch-based filter were exactly those entered into the Simulink filter. Both filters were then fed by same input samples and their outputs were compared, to check the functionality of the latch based decimation FIR filter. A complete block diagram of the whole verification system is shown in figure 5.4.

At first coefficients are generated with the help of Matlab using *Firl*. *Firl* is a command that gives the filter coefficients according to the description provided to it, e.g., *firl*(32, 0.1) provides a 33 tap FIR filter with a cutoff frequency of 0.1 of the Nyquist frequency. These coefficients are then scaled to 11 bit integers to fit the latch-based FIR filter hardware requirement. This was necessary because the filter used integer arithmetic. If all the numbers used in the Simulink model are integers then, if there is no overflow, the Simulink and Verilog models should give identical results. These 11 bit coefficients contains both positive and negative numbers. These coefficients are fed directly to the Simulink filter. In the case of the latch-based FIR filter, the negative coefficients were first converted to equivalent 2's complement integers and are entered into the *filter_top* code.

Simulink generated four sampled sine-waves, added them, and used them as input data. These were converted to 8 bit integers and the latch-based filter data was put in 2's complement form. This data was put in the Verilog *test_bench*. The actual form of the input was not very critical. The object was to show that the filter gave the same result as the Simulink block used as a "gold standard". The time domain was used for verification because it would be much more difficult to check the filter in the frequency domain.

The output of the synthesized latched based filter was in 2's complement integers which were converted back into Matlab's internal format using Matlab. These output samples from the synthesized filter were plotted with the help of Matlab and compared with the output of Simulink

filter. The comparison of the two filter shows that their outputs are almost identical (figure 5.4.1).

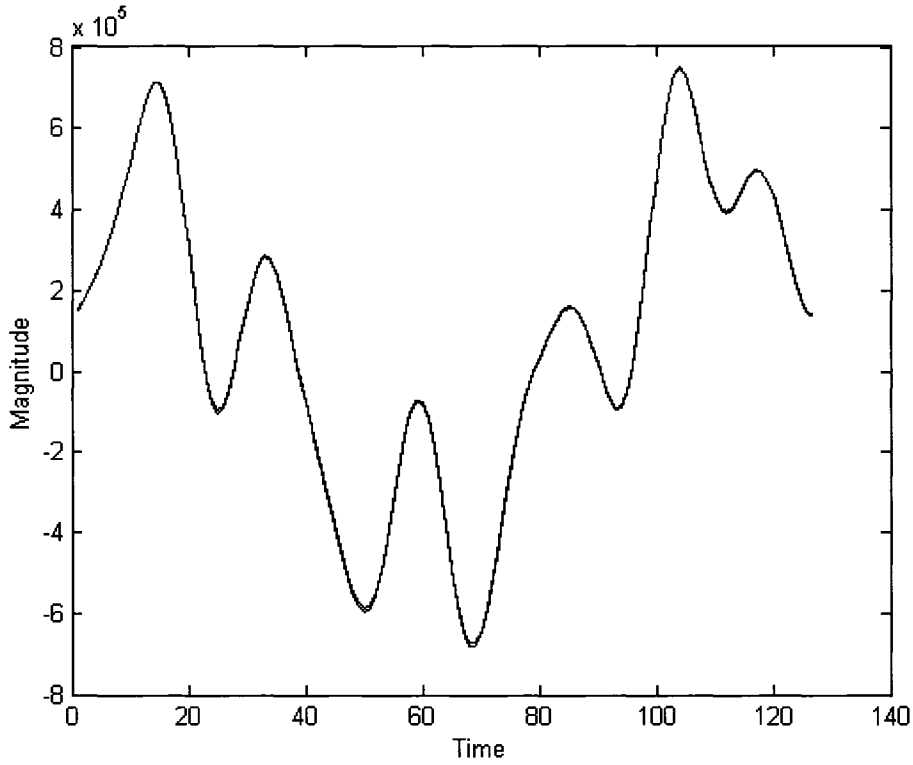

Figure 5.4.1: Synthesis Result compared with Simulink filter in time the domain.

There are very small difference just at the bottom of the most negative valleys. It is suspected that this is caused by a small error in the Verilog code although we were unable to locate it after exhausting effort. It was not caused by overflow because 2's complement overflow causes violent changes from maximum positive to maximum negative [21]. We feel that the error is not of a type that should cause concern about the validity of the general method, or concern that such latch-based circuit cannot be successfully synthesized.

5.5 Summary

The latch-based FIR filter was developed using Verilog HDL and synthesized with the help of the Synopsis Design Analyzer. The synthesis was done using gates in an Artisan library designed for the 0.18μm TSMC CMOS process. Simulation were run on the synthesized circuit. The time response of the synthesized filter was compared to that of an identical edge-triggered register-based filter run under Simulink.

CHAPTER 6: Results

6.1 Introduction

This chapter compares four different decimation filters from the previous Chapter 4. Most of these filters use flip-flops, but a new filter designe uses latches. Different properties of the filters are gathered and compared with the help of a table.

6.2 Overview of Different Decimation FIR Filter Implementations.

These filters architectures are shown here again for the convenience of the readers. The first filter (Figure 6.1) is the direct form [12]. Flip-flops are used as delay elements. A carry-save adder tree without a p-ipeline is used to sum the products.

The second filter (Figure 6.2) is the alternate direct form, which also uses flip-flops for delay elements. The circuit is running with an alternate edge clocking scheme. Adders are placed in between the flip-flops.

The third filter (Figure 6.3) is also based on the alternate direct form [16]. The adders in this circuit are bigger (they add three numbers), but there are fewer flip-flops compared to the one shown in Figure 6.2.

The final filter (Figure 6.4) is designed with the help of latches instead of flip-flops, and is also based on the alternate direct form. This is a new architecture.

Table 6.1 compares the four architectures. All the circuits are clocked at $f_s/2$.

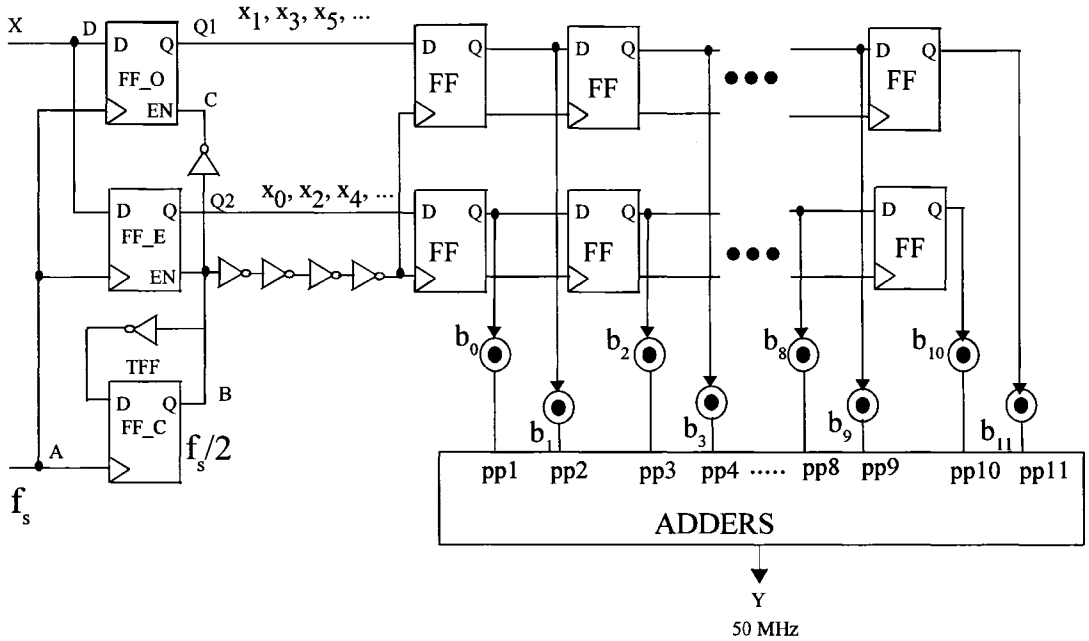

Figure 6.1: Decimation FIR Filter Implementation I.

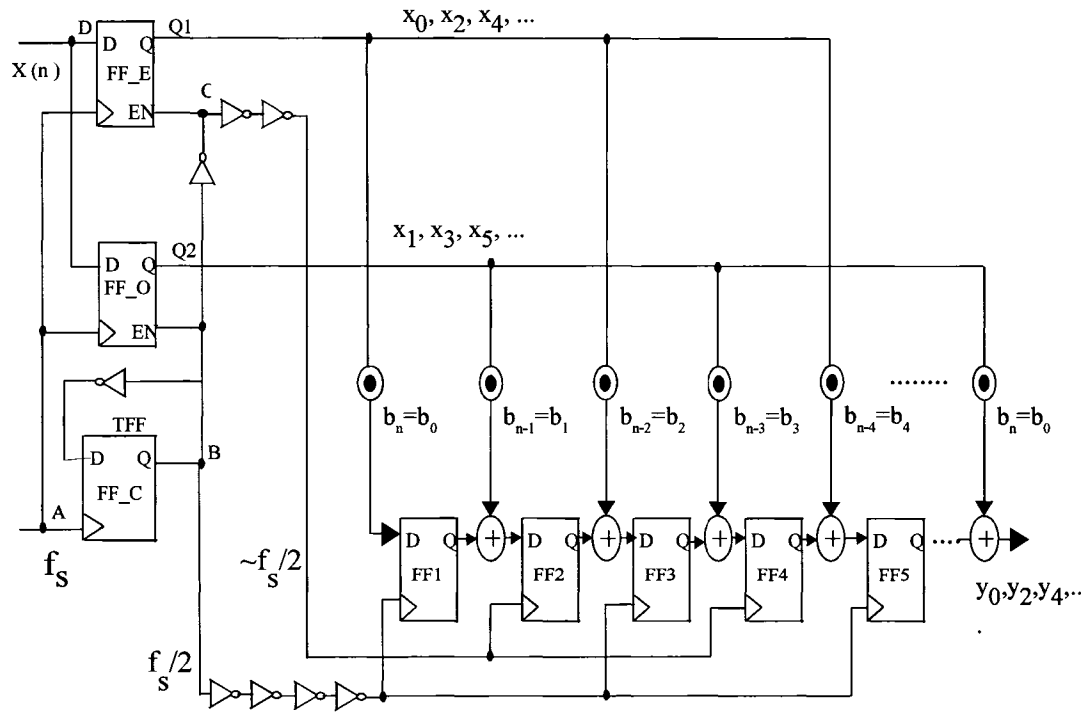

Figure 6.2: Decimation FIR filter Implementation II.

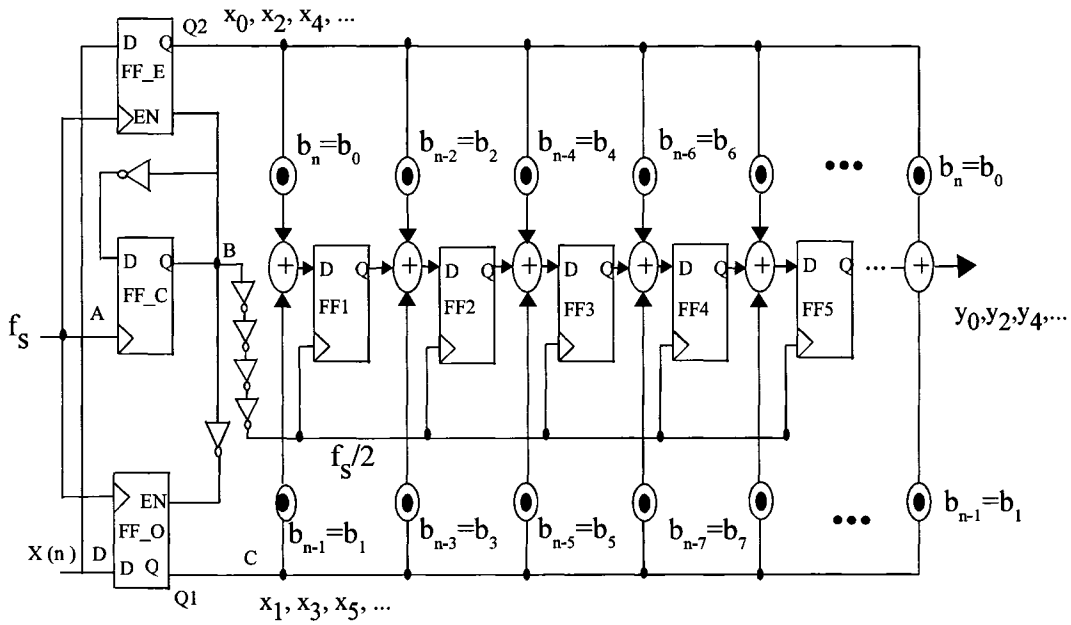

Figure 6.3: Decimation FIR Filter Implementation III.

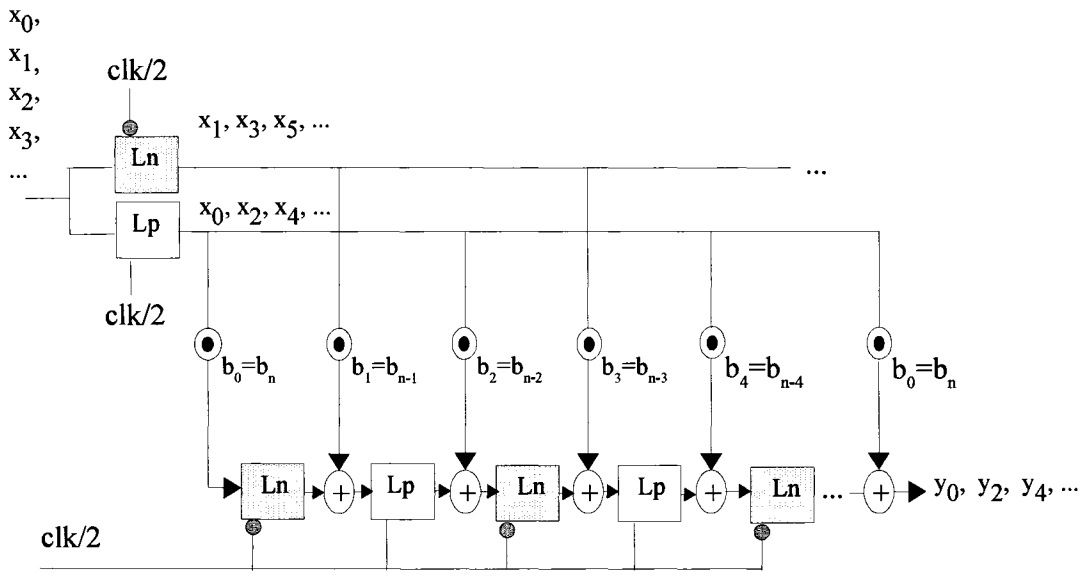

Figure 6.4: Newly Designed Latch Based Decimation FIR Filter Implementation IV.

Parameter	*Decimation filter I*	*Decimation filter II*	*Decimation filter III*	*Decimation Filter IV*
Input Frequency	f	f	f	f
Output Frequency	f/2	f/2	f/2	f/2
Order of the filter.	b	b	b	b
Number of Taps	b+1	b+1	b+1	b+1
Number of Adders	b	b	b	b
Number of Multipliers	(b+1)/2 Note: 1	(b+1)/2	(b+1)/2	(b+1)/2
Number of Latches	2(b+1)+2(bp) Note: 3	2(b+1) Note: 2	(b+1)	(b+1)
Minimum clock period Note: 7	$[M_t+\log_2(b)\ A_t]/2$ Note: 4	$(M_t+A_t)/2$ Note: 5	$(M_t+A_t)/2$ (even) M_t+A_t (odd) Note: 6	$(M_t+A_t)/2$
Decimation Factor	M=2	M=2	M=2	M=2

Notes:

1. Number of multipliers are ≈ half the number of adders because of folding.

2. The term 2(b+1) is used because 2 latches = 1 flip-flop. b represents the number of flip-flops and 2b represents 2 equivalent latches.

3. The term "2(bp)" shows the number of additional latches used by the carry-save adder to meet the pipelining requirements. There are at least four latches (2 flip-flops) for each bit in the word.

4. $\log_2(b)$ refers to how the number of adders depends on the depth of the adder tree.

5. M_t is multiply time. A_t is add time, assuming no overlap.

6.Unless the odd multipliers are much faster then the even (as in half filter) use (M_t+A_t).

7. This is the inverse of the throughput.

Table 6.1: Comparison Between Different Decimation Filter Architectures.

6.2 Comparison Between Different Decimation Filter Design Implementations

The properties of the four filters are compared in table 6.1 The upper row of the table shows the number of taps, adders, multipliers and the input and output frequencies. This thesis is more concerned about the number of delay producing components i.e., flip-flops and latches. The "Number of Latches" row shows that Decimation Filter I contains more latches than Decimation Filter II: This is because Decimation Filter I has extra pipelining flip-flops, which are denoted as *bp* in table 6.1. The related portion of table 6.1 is shown here for the convenience of the readers.

Content to compare	***Decimation filter I***	***Decimation filter II***	***Decimation filter III***	***Decimation Filter IV***
Number of Latches	2(b+1)+2(bp)	2(b+1)	(b+1)	(b+1)

Decimation filters III and IV have half the number of flip-flops compared to Decimation filter I. The clock distribution and latches/flip-flops typically take about half the power in an ASIC. Using this admittedly rough number, saving half the storage would save a quarter of the filter power.

The main operation of FIR filters is the multiply and add operations, done in every clock cycle. That is both multiplication and addition have to be finished within a clock cycle otherwise we have to increase the time period i.e., reduce the speed. The following section will discuss the multiply and add timing of the different implemented filters.

Content to compare	*Decimation filter I*	*Decimation filter II*	*Decimation filter III*	*Decimation Filter IV*
Minimum clock period	$[M_t+\log_2(b)\ A_t]/2$	$(M_t+A_t)/2$	$(M_t+A_t)/2$ (even) M_t+A_t (odd)	$(M_t+A_t)/2$

This is an important section of table 6.1. It shows the minimum clock period that will allow completion of the multiply accumulate operations. The filter throughput will be the inverse of this time.

Decimation filter I has the longest or second largest multiply-add time specification depending on the depth of the adder tree. These adders could be made faster using pipelining which is not used in the table.

Decimation filter II is faster then decimation filter I as it needs $(M_t+A_t)/2$ time to complete multiply-add operation. This is the same as decimation filter IV.

Decimation filter III runs at half the speed of II or IV (It could run faster in some special cases like half-band filters). The reason is that even tap multiply-add operation can be extended to the next clock cycle and hence can take two cycles. But odd tap multiply-add operations have to be completed in one clock cycle.

Decimation filter IV is faster then decimation filter III and equal to II. The multiply-add operation can be extended into the next clock cycle. This means that multiply-add time required per clock cycle is $(M_t+A_t)/2$. Thus the latch based system has both the storage and power saving of filter III and the speed advantage of filter II.

Chapter 7 Conclusion

7.1 Thesis Assertions

This thesis presents a novel method of designing digital decimation digital FIR filters. The main innovation is the successful use of latches instead of flip-flops for storage and delay producing elements.

Several techniques for designing decimation digital FIR filters were compared in terms of performance. Using latches reduce the storage area and storage power up to 50% compared to conventional designs. A recently published implementation [16], however, achieved a similar reduction in area through retiming.

The latch-based decimation FIR filter can also extend the multiplication into the following clock cycle. This allows the clock to run up to 50% faster. Implementation II can also do this but at a cost of twice the storage.

There is some question about the use of latch-based storage in commercial design. This thesis demonstrate the successful synthesis of the circuit using the Synopsis Design Analyzer and Artisan standard cells. The latch-based filter was verified by comparing the time response of the synthesized filter to an identical edge-triggered register-based filter run under Simulink.

A new graphical concept of showing pipe-lined type digital operations with a time-space diagram helped in investigating and comparing filter operations. It simplified tracking of signal propagation in the hardware with respect to time and enabled in depth analysis of different decimation FIR architectures.

7.2 Suggestions for Future Research

The latched base technique was applied to decimation FIR filters only. Initially we attempted to apply the concept to general FIR filters. Unfortunately the result always required as many latches as normal designs. Possible throughput increase were not considered at that time. Only decimation by 2 was investigated in this thesis. Preliminary work indicated the concept would not be difficult to extend to higher factors. Also the same concept might be used to develop FIR filters using a bit-serial approach.

Appendix A

//filter_top.v

```
// The following code is the top module of Decimation FIR filter implementation IV.
// Another module var_sign_multiplier is instantiated in this module.
//filter with latches
`timescale 1ns/1ns
module filter_top(data, clk, rst, sum);
parameter coef_len = 11;

wire [coef_len-1:0] c0 = 11'b11111110001;
wire [coef_len-1:0] c1 = 11'b11111101101;
wire [coef_len-1:0] c2 = 11'b11111100111;
wire [coef_len-1:0] c3 = 11'b11111100000;
wire [coef_len-1:0] c4 = 11'b11111011110;
wire [coef_len-1:0] c5 = 11'b11111100110;
wire [coef_len-1:0] c6 = 11'b00000000000;
wire [coef_len-1:0] c7 = 11'b00000110010;
wire [coef_len-1:0] c8 = 11'b00010000001;
wire [coef_len-1:0] c9 = 11'b00011101101;
wire [coef_len-1:0] c10 = 11'b00101110010;
wire [coef_len-1:0] c11 = 11'b01000000110;
wire [coef_len-1:0] c12 = 11'b01010011110;
wire [coef_len-1:0] c13 = 11'b01100101010;
wire [coef_len-1:0] c14 = 11'b01110011011;
wire [coef_len-1:0] c15 = 11'b01111100101;
wire [coef_len-1:0] c16 = 11'b01111111111;


/* Coefficient
-15 -19 -25 -32 -34 -26 0 50 129 237 370 518 670 810 923 997 1023
997 923 810 670 518 370 237 129 50 0 -26 -34 -32 -25 -19 -15
*/


input clk, rst;
input [7:0] data;
output [28:0] sum;
```

```
wire [18:0] pp0,pp1,pp2,pp3,pp4,pp5,pp6,pp7,pp8,pp9,pp10,pp11,
        pp12,pp13,pp14,pp15,pp16;

reg [7:0] n0,p0;
wire [18:0] in_n1; reg [18:0] n1;
wire [19:0] in_p2; reg [19:0] p2;
wire [20:0] in_n3; reg [20:0] n3;
wire [21:0] in_p4; reg [21:0] p4;
wire [22:0] in_n5; reg [22:0] n5;
wire [22:0] in_p6; reg [22:0] p6;
wire [22:0] in_n7; reg [22:0] n7;
wire [22:0] in_p8; reg [22:0] p8;
wire [22:0] in_n9; reg [22:0] n9;
wire [22:0] in_p10; reg [22:0] p10;
wire [22:0] in_n11; reg [22:0] n11;
wire [22:0] in_p12; reg [22:0] p12;
wire [22:0] in_n13; reg [22:0] n13;
wire [22:0] in_p14; reg [22:0] p14;
wire [22:0] in_n15; reg [22:0] n15;
wire [22:0] in_p16; reg [22:0] p16;
wire [22:0] in_n17; reg [22:0] n17;
wire [22:0] in_p18; reg [22:0] p18;
wire [22:0] in_n19; reg [22:0] n19;
wire [22:0] in_p20; reg [22:0] p20;
wire [22:0] in_n21; reg [22:0] n21;
wire [22:0] in_p22; reg [22:0] p22;
wire [22:0] in_n23; reg [22:0] n23;
wire [22:0] in_p24; reg [22:0] p24;
wire [22:0] in_n25; reg [22:0] n25;
wire [22:0] in_p26; reg [22:0] p26;
wire [22:0] in_n27; reg [22:0] n27;
wire [22:0] in_p28; reg [22:0] p28;
wire [22:0] in_n29; reg [22:0] n29;
wire [22:0] in_p30; reg [22:0] p30;
wire [22:0] in_n31; reg [22:0] n31;
wire [22:0] in_p32; reg [22:0] p32;
wire [22:0] in_n33; reg [22:0] sum;


always @(clk or data or pp0 or in_n1 or in_n3 or in_n5 or in_n7 or in_n9
 or in_n11 or in_n13 or in_n15 or in_n17 or in_n19 or in_n21
 or in_n23 or in_n25 or in_n27 or in_n29 or in_n31
 or in_p2 or in_p4 or in_p6 or in_p8 or in_p10
 or in_p12 or in_p14 or in_p16 or in_p18 or in_p20
 or in_p22 or in_p24 or in_p26 or in_p28 or in_p30 or in_p32 or rst)
```

```
begin
if (rst)
        begin
        n0[7:0] <= 0; p0[7:0] <= 0; n1[18:0] <= 0; p2[19:0] <= 0;
        n3[20:0] <= 0; p4[21:0] <= 0; n5[22:0] <= 0; p6[22:0] <= 0;
 n7[22:0] <= 0; p8[22:0] <= 0; n9[22:0] <= 0; p10[22:0] <= 0;
        n11[22:0] <= 0; p12[22:0] <= 0; n13[22:0] <= 0; p14[22:0] <= 0;
        n15[22:0] <= 0; p16[22:0] <= 0; n17[22:0] <= 0; p18[22:0] <= 0;
        n19[22:0] <= 0; p20[22:0] <= 0; n21[22:0] <= 0; p22[22:0] <= 0;
        n23[22:0] <= 0; p24[22:0] <= 0; n25[22:0] <= 0; p26[22:0] <= 0;
        n27[22:0] <= 0; p28[22:0] <= 0; n29[22:0] <= 0; p30[22:0] <= 0;
        n31[22:0] <= 0; p32[22:0] <= 0; sum[22:0] <= 0;
        end
 if (clk)
        begin
        p0 <= data; p2 <= in_p2;p4 <= in_p4;p6 <= in_p6; p8 <= in_p8;
        p10 <= in_p10; p12 <= in_p12; p14 <= in_p14;p16 <= in_p16;
        p18 <= in_p18; p20 <= in_p20; p22 <= in_p22;p24 <= in_p24;
        p26 <= in_p26; p28 <= in_p28; p30 <= in_p30;p32 <= in_p32;
        end
         if (~clk)
        begin
        n0 <= data; n1 <= in_n1; n3 <= in_n3; n5 <= in_n5; n7 <= in_n7;
        n9 <= in_n9; n11 <= in_n11; n13 <= in_n13; n15 <= in_n15;
         n17 <= in_n17; n19 <= in_n19; n21 <= in_n21; n23 <= in_n23;
        n25 <= in_n25; n27 <= in_n27; n29 <= in_n29; n31 <= in_n31;
        sum <= in_n33;
        end
end



signed_multiplier get_pp0(.signed_no1(p0), .signed_no2(c0), .product(pp0));
signed_multiplier get_pp1(.signed_no1(n0), .signed_no2(c1), .product(pp1));
signed_multiplier get_pp2(.signed_no1(p0), .signed_no2(c2), .product(pp2));
signed_multiplier get_pp3(.signed_no1(n0), .signed_no2(c3), .product(pp3));
signed_multiplier get_pp4(.signed_no1(p0), .signed_no2(c4), .product(pp4));
signed_multiplier get_pp5(.signed_no1(n0), .signed_no2(c5), .product(pp5));
signed_multiplier get_pp6(.signed_no1(p0), .signed_no2(c6), .product(pp6));
signed_multiplier get_pp7(.signed_no1(n0), .signed_no2(c7), .product(pp7));
signed_multiplier get_pp8(.signed_no1(p0), .signed_no2(c8), .product(pp8));
signed_multiplier get_pp9(.signed_no1(n0), .signed_no2(c9), .product(pp9));
signed_multiplier get_pp10(.signed_no1(p0), .signed_no2(c10), .product(pp10));
signed_multiplier get_pp11(.signed_no1(n0), .signed_no2(c11), .product(pp11));
```

```
signed_multiplier get_pp12(.signed_no1(p0), .signed_no2(c12), .product(pp12));
signed_multiplier get_pp13(.signed_no1(n0), .signed_no2(c13), .product(pp13));
signed_multiplier get_pp14(.signed_no1(p0), .signed_no2(c14), .product(pp14));
signed_multiplier get_pp15(.signed_no1(n0), .signed_no2(c15), .product(pp15));
signed_multiplier get_pp16(.signed_no1(p0), .signed_no2(c16), .product(pp16));

// For practical purpose no need to increase the word width beyond 22 bits.

assign in_n1 = pp0,
in_p2 = {n1[18],n1} + {pp1[18],pp1},
in_n3 = {p2[19],p2} + {pp2[18],pp2[18],pp2},
in_p4 = {n3[20],n3} + {pp3[18],pp3[18],pp3[18],pp3},
in_n5 = {p4[21],p4} + {pp4[18],pp4[18],pp4[18],pp4[18],pp4},
in_p6 = {n5[22],n5} + {pp5[18],pp5[18],pp5[18],pp5[18],pp5},
in_n7 = {p6[22],p6} + {pp6[18],pp6[18],pp6[18],pp6[18],pp6},
in_p8 = {n7[22],n7} + {pp7[18],pp7[18],pp7[18],pp7[18],pp7},
in_n9 = {p8[22],p8} + {pp8[18],pp8[18],pp8[18],pp8[18],pp8},
in_p10 = {n9[22],n9} + {pp9[18],pp9[18],pp9[18],pp9[18],pp9},
in_n11 = {p10[22],p10} + {pp10[18],pp10[18],pp10[18],pp10[18],pp10},
in_p12 = {n11[22],n11} + {pp11[18],pp11[18],pp11[18],pp11[18],pp11},
in_n13 = {p12[22],p12} + {pp12[18],pp12[18],pp12[18],pp12[18],pp12},
in_p14 = {n13[22],n13} + {pp13[18],pp13[18],pp13[18],pp13[18],pp13},
in_n15 = {p14[22],p14} + {pp14[18],pp14[18],pp14[18],pp14[18],pp14},
in_p16 = {n15[22],n15} + {pp15[18],pp15[18],pp15[18],pp15[18],pp15},
in_n17 = {p16[22],p16} + {pp16[18],pp16[18],pp16[18],pp16[18],pp16},
in_p18 = {n17[22],n17} + {pp15[18],pp15[18],pp15[18],pp15[18],pp15},
in_n19 = {p18[22],p18} + {pp14[18],pp14[18],pp14[18],pp14[18],pp14},
in_p20 = {n19[22],n19} + {pp13[18],pp13[18],pp13[18],pp13[18],pp13},
in_n21 = {p20[22],p20} + {pp12[18],pp12[18],pp12[18],pp12[18],pp12},
in_p22 = {n21[22],n21} + {pp11[18],pp11[18],pp11[18],pp11[18],pp11},
in_n23 = {p22[22],p22} + {pp10[18],pp10[18],pp10[18],pp10[18],pp10},
in_p24 = {n23[22],n23} + {pp9[18],pp9[18],pp9[18],pp9[18],pp9},
in_n25 = {p24[22],p24} + {pp8[18],pp8[18],pp8[18],pp8[18],pp8},
in_p26 = {n25[22],n25} + {pp7[18],pp7[18],pp7[18],pp7[18],pp7},
in_n27 = {p26[22],p26} + {pp6[18],pp6[18],pp6[18],pp6[18],pp6},
in_p28 = {n27[22],n27} + {pp5[18],pp5[18],pp5[18],pp5[18],pp5},
in_n29 = {p28[22],p28} + {pp4[18],pp4[18],pp4[18],pp4[18],pp4},
in_p30 = {n29[22],n29} + {pp3[18],pp3[18],pp3[18],pp3[18],pp3},
in_n31 = {p30[22],p30} + {pp2[18],pp2[18],pp2[18],pp2[18],pp2},
in_p32 = {n31[22],n31} + {pp1[18],pp1[18],pp1[18],pp1[18],pp1},
in_n33 = {p32[22],p32} + {pp0[18],pp0[18],pp0[18],pp0[18],pp0};

endmodule
```

/* var_sign_mul.v

This is the multiplier module used to multiply the two given signed numbers. If len_no1 is 8 bit long and len_no2 is 11 bit long
then the len_prd will be 18 bit long.
The maximum no 8 bit is binary 10000000 = dec 128
The maximum no 11 bit is binary 10000000000 = dec 1024
The maximum product will be 18 bit long 100000000000000000=dec 131072
Here we are dealing with 2's complement nos. therefore 128 is in fact -128 and 1024 is in fact -1024 and therefore there product will be +131072 hence we have to increase one more bit to the product register in order to keep the last bit zero to recognize it as a +ve no.
The sample length that we are th roughing in the filter is 8 bit long where our coefficient length is 11 bit long. Hence the maximum length of the product will be 18 + 1 bit long for the 2's complement negative nos. scheme.*/

```
`timescale 1ns/1ns
module signed_multiplier(signed_no1, signed_no2, product);
parameter len_no1 = 8;parameter len_no2 = 11;parameter len_prd = 19;

input[len_no1-1:0] signed_no1;
input[len_no2-1:0] signed_no2;
output[len_prd-1:0] product;

wire[len_no1-1:0] signed_no1;
wire[len_no2-1:0] signed_no2;
reg [len_no1-1:0] no1;reg [len_no2-1:0] no2;
reg [len_prd-1:0] product;reg [len_prd-1:0] no3;

always @(signed_no1 or signed_no2)
begin
        if (signed_no1[len_no1-1])
        no1 = (~signed_no1[len_no1-1:0]) + 1;
        else no1 = signed_no1;

        if (signed_no2[len_no2-1])
        no2 = (~signed_no2[len_no2-1:0]) + 1;
        else no2 = signed_no2;
        no3[len_prd-1:0] = no1[len_no1-1:0] * no2[len_no2-1:0];

        if (signed_no1[len_no1-1] ^ signed_no2[len_no2-1])
        product = (~no3[len_prd-1:0])+ 1;
        else
        product = no3;
end
endmodule
```

```
//test_bench.v
//Following is the test bench used for FIR implementation IV.
`timescale 1ns/1ns
module tb_sinewave;

reg clk,rst;
reg [7:0] count_byte;
reg [7:0] count_rst_latch;
reg [7:0] data;
reg [7:0] Mem [0:127];
wire [22:0]sum;

filter_top filter_top1(
        .data(data),
        .clk(clk),
        .rst(rst),
        .sum(sum)
        );

initial

 begin
 $shm_open("Waves_mem.shm");
 $shm_probe("AS");
 count_byte = 0;
 count_rst_latch = 0;


Mem[0]= 8'b00000001; Mem[51] = 8'b00001111; Mem[102]= 8'b00101100;
Mem[1]= 8'b00101001; Mem[52] = 8'b00000101; Mem[103]= 8'b00101000;
Mem[2]= 8'b00111100; Mem[53] = 8'b11110010; Mem[104]= 8'b00100001;
Mem[3]= 8'b01010111; Mem[54] = 8'b11101011; Mem[105]= 8'b00001111;
Mem[4]= 8'b01011110; Mem[55] = 8'b11100011; Mem[106]= 8'b00000101;
Mem[5]= 8'b01011011; Mem[56] = 8'b11100010; Mem[107]= 8'b11101010;
Mem[6]= 8'b01001110; Mem[57] = 8'b11100110; Mem[108]= 8'b11011000;
Mem[7]= 8'b00111111; Mem[58] = 8'b11101110; Mem[109]= 8'b11001101;
Mem[8]= 8'b00101010; Mem[59] = 8'b11110000; Mem[110]= 8'b11001010;
Mem[9]= 8'b00010011; Mem[60] = 8'b11110000; Mem[111]= 8'b11001111;
Mem[10] = 8'b00000010; Mem[61] = 8'b11101101; Mem[112]= 8'b11100101;
Mem[11] = 8'b11110101; Mem[62] = 8'b11011101; Mem[113]= 8'b00000100;
Mem[12] = 8'b11110000; Mem[63] = 8'b11000110; Mem[114]= 8'b00100111;
Mem[13] = 8'b11111000; Mem[64] = 8'b10111011; Mem[115]= 8'b01000001;
Mem[14] = 8'b11111100; Mem[65] = 8'b10110001; Mem[116]= 8'b01010110;
Mem[15] = 8'b00000101; Mem[66] = 8'b10100000; Mem[117]= 8'b01110111;
Mem[16] = 8'b00010100; Mem[67] = 8'b10100010; Mem[118]= 8'b01111100;
```

```
Mem[17] = 8'b00101001; Mem[68] = 8'b10100101; Mem[119]= 8'b01111111;
Mem[18] = 8'b00101000; Mem[69] = 8'b10111011; Mem[120]= 8'b01110001;
Mem[19] = 8'b00100111; Mem[70] = 8'b11001111; Mem[121]= 8'b01011001;
Mem[20] = 8'b00100011; Mem[71] = 8'b11100110; Mem[122]= 8'b00111110;
Mem[21] = 8'b00011100; Mem[72] = 8'b00001000; Mem[123]= 8'b00101011;
Mem[22] = 8'b00010110; Mem[73] = 8'b00100010; Mem[124]= 8'b00001110;
Mem[23] = 8'b00010001; Mem[74] = 8'b00101010; Mem[125]= 8'b11111101;
Mem[24] = 8'b00010011; Mem[75] = 8'b00101001; Mem[126]= 8'b11111010;
Mem[25] = 8'b00011110; Mem[76] = 8'b00101010; Mem[127]= 8'b11111001;
Mem[26] = 8'b00100111; Mem[77] = 8'b00001011;
Mem[27] = 8'b00111100; Mem[78] = 8'b11110100;
Mem[28] = 8'b01001110; Mem[79] = 8'b11010101;
Mem[29] = 8'b01011111; Mem[80] = 8'b10111000;
Mem[30] = 8'b01101001; Mem[81] = 8'b10011111;
Mem[31] = 8'b01101111; Mem[82] = 8'b10001100;
Mem[32] = 8'b01101011; Mem[83] = 8'b10001110;
Mem[33] = 8'b01011000; Mem[84] = 8'b10010011;
Mem[34] = 8'b01001011; Mem[85] = 8'b10011110;
Mem[35] = 8'b00101001; Mem[86] = 8'b11000001;
Mem[36] = 8'b00000100; Mem[87] = 8'b11010101;
Mem[37] = 8'b11100101; Mem[88] = 8'b11100111;
Mem[38] = 8'b11010001; Mem[89] = 8'b11110101;
Mem[39] = 8'b11000111; Mem[90] = 8'b00000011;
Mem[40] = 8'b11000011; Mem[91] = 8'b00001001;
Mem[41] = 8'b11001001; Mem[92] = 8'b00001000;
Mem[42] = 8'b11011101; Mem[93] = 8'b00000000;
Mem[43] = 8'b11111001; Mem[94] = 8'b11111000;
Mem[44] = 8'b00011000; Mem[95] = 8'b11110100;
Mem[45] = 8'b00101011; Mem[96] = 8'b11110010;
Mem[46] = 8'b00111111; Mem[97] = 8'b11111100;
Mem[47] = 8'b01000011; Mem[98] = 8'b11111110;
Mem[48] = 8'b01000011; Mem[99] = 8'b00001111;
Mem[49] = 8'b00111010; Mem[100]= 8'b00100000;
Mem[50] = 8'b00100101; Mem[101]= 8'b00100010;

 clk = 1'b0;
 rst = 1'b0;
 data = 8'b0;
 #1 rst = 1'b1;
 #3 rst = 1'b0;
 #20000 $finish;
 end


always #20 clk = ~clk;
```

```
always@(posedge clk)
$display(" %d ",sum[22:0]);
always@(clk)
 begin
        if (clk)
        begin
         if (count_rst_latch != 35)
         begin
         data = 0;
         count_rst_latch = count_rst_latch + 1;
                  end
                 else
                  begin
                  data = Mem[count_byte];
                  count_byte = count_byte+1;
                  end
        end
                 if (count_byte == 100)
         count_byte = 0;
        if (~clk)
        begin
         if (count_rst_latch != 35)
         begin
         data = 0;
         count_rst_latch = count_rst_latch + 1;
                  end
                 else
 data = Mem[count_byte];
                  count_byte = count_byte+1;
        end
                 if (count_byte == 100)
         count_byte = 0;
end

endmodule
```

References

[1] P. M. Aziz, H. V. Sorenson and J. vander Spiegel, "An Overview of Sigma Delta Converters," *IEEE Signal Processing Magazine*, vol. 13, pp. 61-84, January 1996.

[2] Brian L. Berg and David C. Farden, "Designing Power and Area Efficient Multistage FIR Decimators with the Economical Low Order Filters," Agilent Technologies, Technical Report. *http://eesof tm.agilent.com/pdf/FIR..pdf*

[3] S. Chu and C. S. Burrus: "Multirate Filter Designs Using Comb Filters," *IEEE Trans. Circuits and Sys.*, vol. CAS-31, pp. 913-924, Nov. 1984.

[4] R.E. Crochiere and L.R. Rabiner, *Multirate Digital Signal Processing*, Prentice-Hall , New Jersey, 1983.

[5] Shailesh B. Nerurkar, Khalid H. Abed, Raymond E. Siferd and Vivek Venugopal "Low Power Sigma Delta Decimation Filter," *IEEE Trans. Circuits and Sys.*, vol. CAS-49, pp. 647-650, Aug 2002.

[6] "Distributed Arithmetic FIR filter," Xilinx Inc., Product specification, V9.0, April 2005. *http://www.xilinx.com/ipcenter/catalog/logicore/docs/da_fir.pdf*

[7] E. B. Hogenauer,. "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech, and Signal Processing,* vol. 29, pp. 155–162, April 1981.

[8] "An Introduction to Digital Filers". *Intersil, Application Notes,* January 1999. *http://www.intersil.com/data/an/an9603.pdf*

[9] Y. Gao. "A Comparison Design of Comb Decimators for Sigma-Delta Analog-to-Digital Converters," *Analog Integrated Circuits and Signal Processing*, vol.22, pp. 51–60, January 2000.

[10] M. G. Bellanger, "Digital Filtering by Poly phase Network: Application to Sample-Rate Alteration and Filter Banks," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 24, pp. 109–114, April 1976.

[11] Y. Gao and L. Jia, "A Fifth-Order Comb Decimation Filter for Multi-Standard Transceiver Applications," *IEEE Trans. Circuits and Sys.*, vol. CAS-49, pp. 89-92, May 2002.

[12] "*I*mplementing FIR Filters in FLEX Devices," Application Note 73, Altera Corporation, February 1998.

[13]"AD7725 filtering analog to digital converters data converters," Analog Devices, Note. *http://www.analog.com/en/content/0,2886,760%255F%55F60662,00.html*

[14] Ashok Ambadar, *Analog and Digital Signal Processing,* Second Edition, Brooks/Cole, 1999.

[15] Carol J. Barret, "Low power Decimated Filter Design for Multi Standard Transceiver Applications," Master's Thesis, University of California, 1997.

[16] Prabir C. Maulik, S. Chadha, Wai L. Lee and Philip J. Crawley, "A 16-Bit 250-kHz Delta-Sigma Modulator and Decimation Filter," *IEEE Journal of Solid-State Circuits*, vol. 35, pp. 458- 467, April 2000.

[17] P. P. Vaidyanathan, *Multirate Systems and Filter Banks,* Prentice-Hall, Englewood Cliffs, NJ., 1993.

[18] J. C. Candy, "Decimation for sigma delta modulation," *IEEE Transactions Communications*, vol. 34, pp. 72-76, Jan. 1986.

[19] J. T. Ludwig, S. H. Nawab, and A. P. Chandrakasan, "Low-power digital filtering using approximate processing," *IEEE Journal, Solid-State Circuits*, vol. 31, pp. 395 – 400, Mar. 1996.

[20] C. J. Pan, "A stereo audio chip using approximate processing for decimation and interpolation filters," *IEEE J. Solid-State Circuits*, vol. 35, pp. 45–55, Jan. 2000.

[21] Leland B. Jackson, *Digital Filter and Signal Processing*, Kluwer Academic Publishers, Norwell MA., 1996.

[22] S. Jou, S. Wu and C. Wang, "Low-power multirate architecture for RF digital frequency down converter," *IEEE Trans. Circuits Sys.,* vol. CAS-45, pp. 1487-1494, Nov. 1998.

[23] B. A. White and M. I. Elmasry, "Low-power design of decimation filters for a digital IF receiver," *IEEE Transactions on VLSI Systems,* vol. 8, pp. 339-345, June 2000.

[24] Martin S. Roden, *Analog and Digital Communication Systems,* Prentice-Hall, Englewood Cliffs, NJ., 1991.

[25] Neil H.E. Weste, Kamran Eshraghian, *Principles of CMOS VLSI Design,* Addison-Wesley, Reading MA., 1994.

[26] S. Chu. "Multirate Filter Designs Using Comb Filters," *IEEE Trans. on Circuits and Sys.*, vol. CAS-31, pp. 913–924, November 1984.

[27] K.M. Daugherty, *Analog to Digital Conversion,* McGraw Hill, New York, NY, 1995.

[28] John P. Hayes, *Introduction to Digital Logic Design*, Addison-Wesley, Reading MA., 1993.