

**DESIGN AND IMPLEMENTATION OF DIFFERENT MULTIPLIERS USING  
VHDL**

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE OF

**Bachelor of Technology  
in  
Electronics and Communication Engineering**

By  
**MOUMITA GHOSH**



**Department of Electronics and Communication Engineering**

**National Institute of Technology**

**Rourkela**

2007



**National Institute of Technology  
Rourkela**

**CERTIFICATE**

This is to certify that the thesis entitled, “DESIGN AND IMPLEMENTATION OF DIFFERENT MULTIPLIERS USING VHDL ” submitted by Ms Moumita Ghosh in partial fulfillments for the requirements for the award of Bachelor of Technology Degree in Electronics and Communication Engineering at National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by her under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University / Institute for the award of any Degree or Diploma.

Date:

Prof. Dr. K.K Mahapatra  
Dept. of Electronics and Communication Engineering  
National Institute of Technology  
Rourkela - 769008

## ACKNOWLEDGEMENT

I would like to articulate my profound gratitude and indebtedness to my project guide **Prof.Dr. K.K Mahapatra** who has always been a constant motivation and guiding factor throughout the project time in and out as well. It has been a great pleasure for me to get a opportunity to work under him and complete the project successfully.

I wish to extend my sincere thanks to **Prof.Dr. G Panda**, Head of our Department, for approving our project work with great interest.

I would also like to mention Mr. Jitendra Behera, Ms Durga Digdarshani and Mr Sushant Pattnaik M.Tech Student, for their cooperation and constantly rendered assistance.

It is my pleasure to refer VHDL, Acrobat Reader and Microsoft Word exclusive of which the whole process, right from simulation to compilation of this report would have been impossible.

An undertaking of this nature could never have been attempted with our reference to and inspiration from the works of others whose details are mentioned in references section. I acknowledge my indebtedness to all of them. Last but not the least, my sincere thanks to all of my friends who have patiently extended all sorts of help for accomplishing this undertaking.

# CONTENTS

Abstract.....	05
List of	
Figures.....	ii
List of Tables.....	iii
Introduction.....	09
VHDL .....	14
Filters.....	26
Type of filters.....	29
Adders.....	31
Binary multipliers.....	41
Results.....	54
Conclusion.....	57
References.....	59

## ABSTRACT:

Low power consumption and smaller area are some of the most important criteria for the fabrication of DSP systems and high performance systems. Optimizing the speed and area of the multiplier is a major design issue. However, area and speed are usually conflicting constraints so that improving speed results mostly in larger areas. In our project we try to determine the best solution to this problem by comparing a few multipliers.

This project presents an efficient implementation of high speed multiplier using the shift and add method, Radix\_2, Radix\_4 modified Booth multiplier algorithm. In this project we compare the working of the three multiplier by implementing each of them separately in FIR filter.

The parallel multipliers like radix 2 and radix 4 modified booth multiplier does the computations using lesser adders and lesser iterative steps. As a result of which they occupy lessr space as compared to the serial multiplier. This a very important criteria because in the fabrication of chips and high performance system requires components which are as small as possible.

In our project when we compare the power consumption of all the multipliers we find that serial multipliers consume more power. So where power is an important criterion there we should prefer parallel multipliers like booth multipliers to serial multipliers. The low power consumption quality of booth multiplier makes it a preffered choice in designing different circuits

In this project we first designed three different type of multipliers using shift snd method, radix 2 and radix 4 modified booth multiplier algorithm. We used different type of adders like sixteen bit full adder in designing those multiplier. Then we designed a 4 tap delay FIR filter and in place of the multiplication and additions we implemented the

components of different multipliers and adders. Then we compared the working of different multipliers by comparing the power consumption by each of them.

The result of our project helps us to choose a better option between serial and parallel multiplier in fabricating different systems. Multipliers form one of the most important component of many systems. So by analyzing the working of different multipliers helps to frame a better system with less power consumption and lesser area.

# **CHAPTER 1**

**INTRODUCTION**  
**LOW POWER CONSUMPTION**

## INTRODUCTION

Multipliers are key components of many high performance systems such as FIR filters, microprocessors, digital signal processors, etc. A system's performance is generally determined by the performance of the multiplier because the multiplier is generally the slowest element in the system. Furthermore, it is generally the most area consuming. Hence, optimizing the speed and area of the multiplier is a major design issue. However, area and speed are usually conflicting constraints so that improving speed results mostly in larger areas. As a result, a whole spectrum of multipliers with different area-speed constraints have been designed with fully parallel. Multipliers at one end of the spectrum and fully serial multipliers at the other end. In between are digit serial multipliers where single digits consisting of several bits are operated on. These multipliers have moderate performance in both speed and area. However, existing digit serial multipliers have been plagued by complicated switching systems and/or irregularities in design. Radix  $2^n$  multipliers which operate on digits in a parallel fashion instead of bits bring the pipelining to the digit level and avoid most of the above problems. They were introduced by M. K. Ibrahim in 1993. These structures are iterative and modular. The pipelining done at the digit level brings the benefit of constant operation speed irrespective of the size of the multiplier. The clock speed is only determined by the digit size which is already fixed before the design is implemented.



# CHAPTER 2

VHDL :THE LANGUAGE

## **EXPERIMENTAL**

Many DSP applications demand high throughput and real-time response, performance constraints that often dictate unique architectures with high levels of concurrency. DSP designers need the capability to manipulate and evaluate complex algorithms to extract the necessary level of concurrency. Performance constraints can also be addressed by applying alternative technologies. A change at the implementation level of design by the insertion of a new technology can often make viable an existing marginal algorithm or architecture.

The VHDL language supports these modeling needs at the algorithm or behavioral level, and at the implementation or structural level. It provides a versatile set of description facilities to model DSP circuits from the system level to the gate level. Recently, we have also noticed efforts to include circuit-level modeling in VHDL. At the system level we can build behavioral models to describe algorithms and architectures. We would use concurrent processes with constructs common to many high-level languages, such as if, case, loop, wait, and assert statements. VHDL also includes user-defined types, functions, procedures, and packages." In many respects VHDL is a very powerful, high-level, concurrent programming language. At the implementation level we can build structural models using component instantiation statements that connect and invoke subcomponents. The VHDL generate statement provides ease of block replication and control. A dataflow level of description offers a combination of the behavioral and structural levels of description. VHDL lets us use all three levels to describe a single component. Most importantly, the standardization of VHDL has spurred the development of model libraries and design and development tools at every level of abstraction. VHDL, as a consensus description language and design environment, offers design tool portability, easy technical exchange, and technology insertion

## VHDL: The language

An entity declaration, or entity, combined with architecture or body constitutes a VHDL model. VHDL calls the entity-architecture pair a design entity. By describing alternative architectures for an entity, we can configure a VHDL model for a specific level of investigation. The entity contains the interface description common to the alternative architectures. It communicates with other entities and the environment through ports and generics. Generic information particularizes an entity by specifying environment constants such as register size or delay value. For example,

```
entity A is
    port (x, y: in real; z: out real);
    generic (delay: time);
end A;
```

The architecture contains declarative and statement sections. Declarations form the region before the reserved word **begin** and can declare local elements such as signals and components. Statements appear after *begin* and can contain concurrent statements. For instance,

```
architecture B of A is
    component M
        port (j : in real ; k : out real);
    end component;

    signal a,b,c real := 0.0;

begin
    "concurrent statements"
end B;
```

The variety of concurrent statement types gives VHDL the descriptive power to create and combine models at the structural, dataflow, and behavioral levels into one simulation model. The structural type of description makes use of component instantiation statements to invoke models described elsewhere. After declaring components, we use them in the component instantiation statement, assigning ports to local signals or other ports and giving values to generics. `invert: M port map ( j => a ; k => c);` We can then bind the components to other design entities through configuration specifications in VHDL's architecture declarative section or through separate configuration declarations.

The dataflow style makes wide use of a number of types of concurrent signal assignment statements, which associate a target signal with an expression and a delay. The list of signals appearing in the expression is the sensitivity list; the expression must be evaluated for any change on any of these signals. The target signals obtain new values after the delay specified in the signal assignment statement. If no delay is specified, the signal assignment occurs during the next simulation cycle:

```
c <= a + b after delay;
```

VHDL also includes conditional and selected signal assignment statements. It uses block statements to group signal assignment statements and makes them synchronous with a guarded condition. Block statements can also contain ports and generics to provide more modularity in the descriptions. We commonly use concurrent process statements when we wish to describe hardware at the behavioral level of abstraction. The process statement consists of declarations and procedural types of statements that make up the sequential program. Wait and assert statements add to the descriptive power of the process statements for modeling concurrent actions:

```
process
begin
    variable i : real := 1.0;
wait on a;
    i = b * 3.0;
```

```
c <= i after delay;  
end process;
```

Other concurrent statements include the concurrent assertion statement, concurrent procedure call, and generate statement. Packages are design units that permit types and objects to be shared. Arithmetic operations dominate the execution time of most Digital Signal Processing (DSP) algorithms and currently the time it takes to execute a multiplication operation is still the dominating factor in determining the instruction cycle time of a DSP chip and Reduced Instruction Set Computers (RISC). Among the many methods of implementing high speed parallel multipliers, there is one basic approach namely Booth algorithm.

Power consumption in VLSI DSPs has gained special attention due to the proliferation of high-performance portable battery-powered electronic devices such as cellular phones, laptop computers, etc. DSP applications require high computational speed and, at the same time, suffer from stringent power dissipation constraints.

Multiplier modules are common to many DSP applications. The fastest types of multipliers are parallel multipliers. Among these, the Wallace multiplier is among the fastest. However, they suffer from a bad regularity. Hence, when regularity, high-performance and low power are primary concerns, Booth multipliers tend to be the primary choice.

Booth multipliers allow the operation on signed operands in 2's-complement. They derive from array multipliers where, for each bit in a partial product line, an encoding scheme is used to determine if this bit is positive, negative or zero. The Modified Booth algorithm achieves a major performance improvement through radix-4 encoding. In this algorithm each partial product line operates on 2 bits at a time, thereby reducing the total number of the partial products. This is particularly true for operands using 16 bits or more.

# **CHAPTER 3**

**FILTERS**  
**TYPES OF FILTERS**

## **FILTERS:**

Digital filters are very important part of DSP. Infact their extraordinary performance is one of the key reasons that DSP has become so popular. Filters have two uses: signal separation and signal restoration. Signal separation is needed when the signal has been contaminated with interference, noise or other signals. For example imagine a device for measuring the electrical activity of a baby's heart (EKG) while in the womb. The raw signal will be likely to be corrupted by the breathing and the heartbeat of the mother. A filter must be used to separate these signals so that they can be individually analyzed.

Signal restoration is used when the signal has been distorted in some way. For example, an audio recording made with poor requirement may be filtered to better represent the sound as it actually occurred. Another example is of deblurring of an image acquired with an improper focused lens, or a shaky camera.

These problems can be attacked with either digital or analog filters. Which is better? Analog filters are cheap, fast and have a large dynamic range both in amplitude and frequency. Digital filters in comparison are vastly superior in the level of performance that can be achieved. Digital filters can achieve thousand of times better performance than an analog filter. This makes a dramatic difference in how filtering problems are approached. With analog filters, the emphasis is on handling limitations of the electronics such as the accuracy and stability of the resistors and capacitors. In comparison digital filters are so good that the performance of the filter is frequently ignored. The emphasis shifts to the limitations of the signals and the theoretical issues regarding their processing.

It is common in DSP to say that a filter input and output signals are in time domain. This is because signals are usually created by sampling at regular intervals of time. But this is not the only way sampling can take place. The second most common way of sampling is

at equal intervals in space. For example imagine taking simultaneous readings from an array of strain sensors mounted at one centimeter increments along the length of an aircraft wing. Many other domains are possible; however, time and space are by far the most common. When you see the term time domain in DSP, remember that it may actually refer to samples taken over time, or it may be a general reference to any domain that the samples are taken in.

Every linear filter has an impulse response, a step response and a frequency response. Each of these responses contains complete information about the filter, but in a different form. If one of three is specified, the other two are fixed and can be directly calculated. All three of these representations are important, because they describe how the filter will react under different circumstances.

The most straightforward way to implement a digital filter is by convolving the input signal with the digital filter's impulse response. All possible linear filters can be made in this manner. When the impulse response is used in this way, filters designers give it a special name: the filter kernel. There is also another way to make digital filters, called recursion. When a filter is implemented by a convolution, each sample in the output is calculated by weighting the samples in the input, and adding them together. Recursive filters are an extension of this, using previously calculated values from the output, besides points from the input. Instead of using a filter kernel, recursive filters are defined by a set of recursion coefficients. For now the important point is that all linear filters have an impulse response, even if you don't use it to implement the filter. To find the impulse response of a recursive filter, simply feed in the impulse and see what comes out. The impulse responses of recursive filters are composed of sinusoids that exponentially decay in amplitude. In principle, this makes their impulse responses infinitely long. However the amplitude eventually drops below the round off noise of the system, and the remaining samples can be ignored. Because of these characteristics, recursive filters are also called Infinite impulse response or IIR filters. In comparison, filters carried out by convolution are called Finite impulse response or FIR filters.



The impulse response is the output of a system when the input is an impulse. In this same manner, the step response is the output when the input is a step. Since the step is the integral multiple of the impulse response. This provides two ways to find the step response: (1) feed a step waveform into the filter and see what comes out. (2) Integrate the impulse response. The frequency response can be found out by taking the DFT of impulse response.

### **Time domain Parameters**

It may not be obvious why the step response is of such concern in time domain filters. You may be wondering why the impulse response isn't the important parameter. The answer lies in the way that the human mind understands and processes information. Remember that the step, impulse and frequency responses all contain identical information, just in different arrangements. The step response is useful in time domain analysis because it matches the way humans view the information contained in the signals.

For example, suppose you are given a signal of some unknown origin and asked to analyze it. The first thing you will do is divide the signal into regions of similar characteristics. You can't stop from doing this; your mind will do that automatically. Some of the regions may be smooth; others may have large amplitude peaks; others may be noisy. This segmentation is accomplished by identifying the points that separate the regions. This is where the step function comes in. the step function is the purest way of representing a division between two dissimilar regions. It can mark when an event starts or when an event ends. It tells you that whatever is on the right. This is how the human mind views time domain information: a group of step functions dividing the information into region of similar characteristics. The step response, in turn, is important because it describes how the dividing lines are being modified by the filter.

## Frequency domain parameters

The purpose of the filters is to allow some frequencies to pass unaltered, while completely blocking other frequencies. The pass band refers to those frequencies that are passed, while stop band contains those frequencies that are blocked. The transition band is between. A fast roll-off means that the transition band is very narrow. The division between the pass band and transition band is called the cut off frequency. In analog filter design, the cut off frequency is usually defined at less standardized, and it is common to see 99%, 90%, 70.7% and 50% amplitude levels defined to be the cut off frequency.

## Types of filters

High pass, band pass and band reject filters are designed by starting with a low pass filter, and then converting it into the desired response. For this reason, most discussions on filter design only give examples of low pass filters.

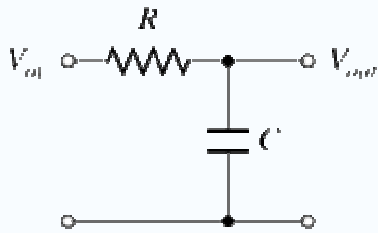
## Low pass filters

Low-pass filter is a filter that passes low frequencies but attenuates (or reduces) frequencies higher than the cutoff frequency. The actual amount of attenuation for each frequency varies from filter to filter. It is sometimes called a high-cut filter, or treble cut filter when used in audio applications.

A high-pass filter is the opposite, and a band pass filter is a combination of a high-pass and a low-pass.

The concept of a low-pass filter exists in many different forms, including electronic circuits (like a *hiss filter* used in audio), digital algorithms for smoothing sets of data, acoustic barriers, blurring of images, and so on. Low-pass filters play the same role in signal processing that moving averages do in some other fields, such as finance; both tools provide a smoother form of a signal which removes the short-term oscillations, leaving only the long-term trend

## Examples of low pass filters



A low-pass electronic filter realized by an RC circuit

A fairly stiff barrier reflects higher frequencies, and so acts as a low-pass filter for transmitting sound waves. When music is playing in another room, the low notes are easily heard, while the high notes are largely filtered out. Similarly, very loud music played in one car is heard as a low throbbing by occupants of other cars, because the closed vehicles (and air gap) function as a very low-pass filter, attenuating all of the treble.

Electronic low-pass filters are used to drive subwoofers and other types of loudspeakers, to block high pitches that they can't efficiently broadcast. Radio transmitters use low pass filters to block harmonic emissions which might cause interference with other communications. An integrator is another example of a low-pass filter. DSL use low-pass and high-pass filters to separate DSL and POTS signals sharing the same pair of wires. Low-pass filters also play a significant role in the sculpting of sound for electronic music as created by analogue synthesizers.

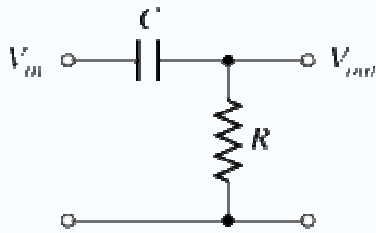
## High pass filter

A high-pass filter is a filter that passes high frequencies well, but attenuates (or reduces) frequencies lower than the cutoff frequency. The actual amount of attenuation for each frequency varies from filter to filter. It is sometimes called a low-cut filter; the terms bass-cut filter or rumble filter are also used in audio applications. A high-pass filter is the

opposite of a low-pass filter, and a band pass filter is a combination of a high-pass and a low-pass.

It is useful as a filter to block any unwanted low frequency components of a complex signal while passing the higher frequencies. Of course, the meanings of 'low' and 'high' frequencies are relative to the cutoff frequency chosen by the filter designer.

### Examples of high pass filters



A passive, analog, first-order high-pass filter, realized by an RC circuit

The simplest electronic high-pass filter consists of a capacitor in series with the signal path in conjunction with a resistor in parallel with the signal path. The resistance times the capacitance ( $R \times C$ ) is the time constant ( $\tau$ ); it is inversely proportional to the cutoff frequency, at which the output power is half the input ( $-3$  dB):

$$f = \frac{1}{2\pi\tau} = \frac{1}{2\pi RC}$$

Where  $f$  is in hertz,  $\tau$  is in seconds,  $R$  is in ohms, and  $C$  is in farads.

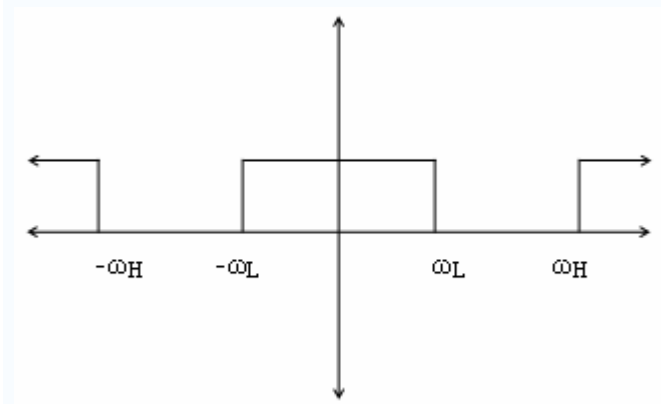
Such a filter could be used to direct high frequencies to a tweeter speaker while blocking bass signals which could interfere with or damage the speaker. A low-pass filter, using a coil instead of a capacitor, could simultaneously be used to direct low frequencies to the woofer.

High-pass and low-pass filters are also used in digital image processing to perform transformations in the spatial frequency domain.

Most high-pass filters have zero gain ( dB) at DC. Such a high-pass filter with very low cutoff frequency can be used to block DC from a signal that is undesired in that signal (and pass nearly everything else). These are sometimes called DC blocking filters.

### **Band reject filters**

In signal processing, a band-stop filter or band-rejection filter is a filter that passes most frequencies unaltered, but attenuates those in a specific range to very low levels. It is the opposite of a band-pass filter. A notch filter is a band-stop filter with a narrow stop band (high Q factor). Notch filters are used in live sound reproduction (Public Address systems, also known as PA systems) and in instrument amplifier (especially amplifiers or preamplifiers for acoustic instruments such as acoustic guitar, mandolin, bass instrument amplifier, etc.) to reduce or prevent feedback, while having little noticeable effect on the rest of the frequency spectrum. Other names include 'band limit filter', 'T-notch filter', 'band-elimination filter', and 'band-rejection filter'.



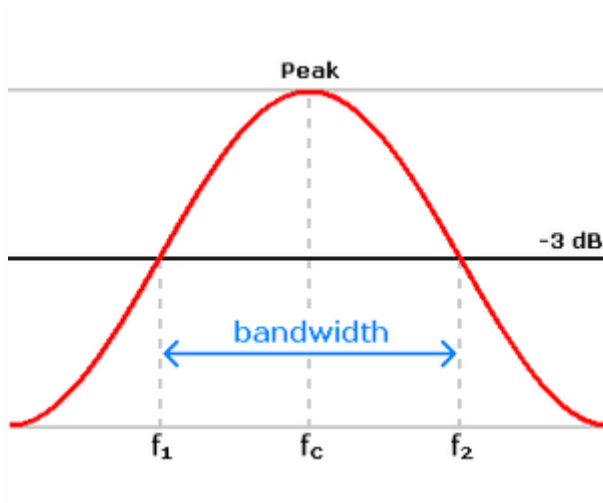
A generic ideal band-stop filter, showing both positive and negative angular frequencies

Typically, the width of the stop band is less than 1 to 2 decades (that is, the highest frequency attenuated is less than 10 to 100 times the lowest frequency attenuated). In the audio band, a notch filter uses high and low frequencies that may be only semitones apart.

### **Band pass filters**

A band-pass filter is a device that passes frequencies within a certain range and rejects (attenuates) frequencies outside that range. An example of an analogue electronic band-pass filter is an RLC circuit (a resistor-inductor-capacitor circuit). These filters can also be created by combining a low-pass filter with a high-pass filter.

An ideal filter would have a completely flat pass band (e.g. with no gain/attenuation throughout) and would completely attenuate all frequencies outside the pass band. Additionally, the transition out of the pass band would be instantaneous in frequency. In practice, no band pass filter is ideal. The filter does not attenuate all frequencies outside the desired frequency range completely; in particular, there is a region just outside the intended pass band where frequencies are attenuated, but not rejected. This is known as the filter roll-off, and it is usually expressed in dB of attenuation per octave or decade of frequency. Generally, the design of a filter seeks to make the roll-off as narrow as possible, thus allowing the filter to perform as close as possible to its intended design. However, as the roll-off is made narrower, the pass band is no longer flat and begins to "ripple." This effect is particularly pronounced at the edge of the pass band in an effect known as the Gibbs phenomenon.



Bandwidth measured at half-power points on a diagram showing power transfer function versus frequency for a band-pass filter.

## FIR FILTERS

Digital filters can be divided into two categories: finite impulse response (FIR) filters; and infinite impulse response (IIR) filters. Although FIR filters, in general, require higher taps than IIR filters to obtain similar frequency characteristics, FIR filters are widely used because they have linear phase characteristics, guarantee stability and are easy to implement with multipliers, adders and delay elements [ 1,2]. The number of taps in digital filters varies according to applications. In commercial filter chips with the fixed number of taps [3], zero coefficients are loaded to registers for unused taps and unnecessary calculations have to be performed. To alleviate this problem, the FIR filter chips providing variable-length taps have been widely used in many application fields [4-61]. However, these FIR filter chips use memory, an address generation unit, and a modulo unit to access memory in a circular manner. The paper proposes two special features called a data reuse structure and a recurrent-coefficient scheme to provide variable-length taps efficiently. Since the proposed architecture only requires several MUXs, registers, and a feedback-loop, the number of gates can be reduced over 20 % than existing chips.

In, general, FIR filtering is described by a simple convolution operation as expressed in the equation (1)

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k] \quad (1)$$

where  $x[n]$ ,  $y[n]$ , and  $h[n]$  represent data input, filtering output, and a coefficient, respectively and  $N$  is the filter order. The equation using the bit-serial algorithm for a FIR filter can be represented as

$$y[n] = \sum_{k=0}^{N-1} \sum_{j=0}^{M-1} (h_j[k] \cdot 2^j) \cdot x[n-k] \quad (2)$$

where the  $h_j$ ,  $N$  and  $M$  are the  $j$ th bit of the coefficient

### Transversal filter

An N-Tap transversal was assumed as the basis for this adaptive filter. The value of  $N$  is determined by practical considerations, [1]. An FIR filter was chosen because of its stability. The use of the transversal structure allows relatively straight forward construction of the filter, Fig.1.

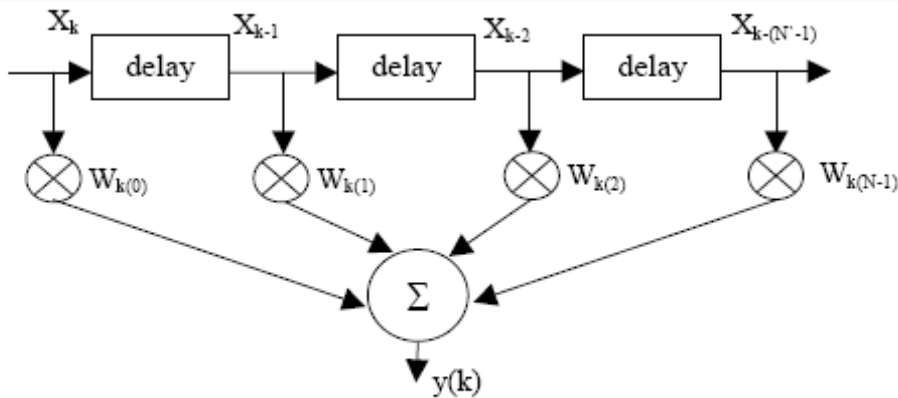


Figure 1. Transversal FIR Filter

As the input, coefficients and output of the filter are all assumed to be complex valued, and then the natural choice for the property measurement is the modulus, or instantaneous



amplitude. If  $y(k)$  is the complex valued filter output, then  $|y(k)|$  denotes the amplitude. The convergence error  $p(k)$  can be defined as follows:

$$p(k) = |y(k) - A| \quad (4)$$

where the  $A$  is the amplitude in the absence of signal degradations.

The error  $p(k)$  should be zero when the envelope has the proper value, and non-zero otherwise. The error carries sign information to indicate which direction the envelope is in error. The adaptive algorithm is defined by specifying a performance/cost/fitness function based on the error  $p(k)$  and then developing a procedure that adjusts the filter impulse response so as to minimize or maximize that performance function.

$$Y_k = \sum_{i=0}^{N-1} w_k(i) x_{k-i} \quad (5)$$

The gradient search algorithm was selected to simplify the filter design. The filter coefficient update equation is given by:

$$W_{k+1} = w_k - \mu e_k x_k \quad (6)$$

Where  $X_k$  is the filter input at sample  $k$ ,  $e_k$  is the error term at sample  $k = p_k \cdot y_k$  and  $\mu$  is the step size for updating the weights value.

# **CHAPTER 4**

**ADDERS**

**HALF ADDER**

**FULL ADDER**

## ADDER

In electronics, an adder is a digital circuit that performs addition of numbers. In modern computers adders reside in the arithmetic logic unit (ALU) where other operations are performed. Although adders can be constructed for many numerical representations, such as Binary-coded decimal or excess-3, the most common adders operate on binary numbers. In cases where two's complement is being used to represent negative numbers it is trivial to modify an adder into an adder-subtractor

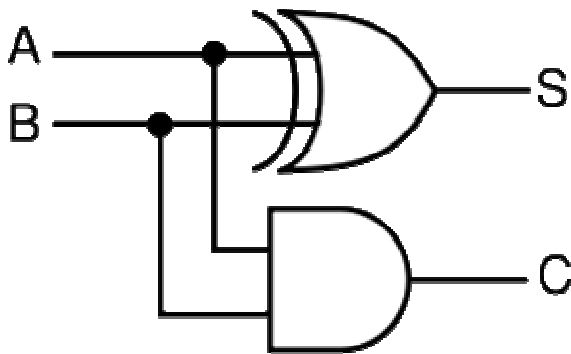
### Types of adders

For single bit adders, there are two general types.

A half adder has two inputs, generally labeled  $A$  and  $B$ , and two outputs, the sum  $S$  and carry  $C$ .  $S$  is the two-bit XOR of  $A$  and  $B$ , and  $C$  is the AND of  $A$  and  $B$ . Essentially the output of a half adder is the sum of two one-bit numbers, with  $C$  being the most significant of these two outputs.

The second type of single bit adder is the full adder. The full adder takes into account a carry input such that multiple adders can be used to add larger numbers. To remove ambiguity between the input and output carry lines, the carry in is labeled  $C_i$  or  $C_{in}$  while the carry out is labeled  $C_o$  or  $C_{out}$ .

### Half adder



Half adder circuit diagram

A half adder is a logical circuit that performs an addition operation on two binary digits. The half adder produces a sum and a carry value which are both binary digits.

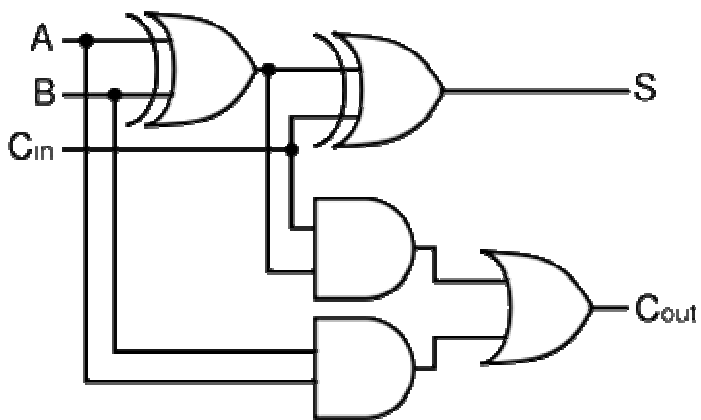
$$S = A \oplus B$$
$$C = A \cdot B$$

Following is the logic table for a half adder:

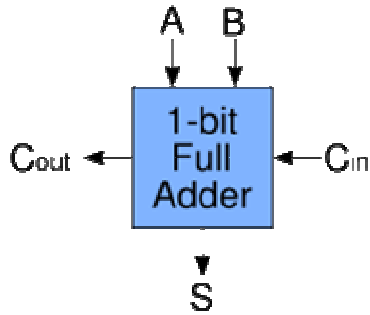
Input Output

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

**Full adder**



Inputs: {A, B, Carry In} → Outputs: {Sum, Carry Out}



Schematic symbol for a 1-bit full adder

A full adder is a logical circuit that performs an addition operation on three binary digits. The full adder produces a sum and carries value, which are both binary digits. It can be combined with other full adders (see below) or work on its own.

$$S = (A \oplus B) \oplus C_i$$

$$C_o = (A \cdot B) + (C_i \cdot (A \oplus B)) = (A \cdot B) + (B \cdot C_i) + (C_i \cdot A)$$

Input Output

<i>A</i>	<i>B</i>	<i>C<sub>i</sub></i>	<i>C<sub>o</sub></i>	<i>S</i>
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Note that the final OR gate before the carry-out output may be replaced by an XOR gate without altering the resulting logic. This is because the only discrepancy between OR and

XOR gates occurs when both inputs are 1; for the adder shown here, one can check this is never possible. Using only two types of gates is convenient if one desires to implement the adder directly using common IC chips.

A full adder can be constructed from two half adders by connecting  $A$  and  $B$  to the input of one half adder, connecting the sum from that to an input to the second adder, connecting  $C_i$  to the other input and or the two carry outputs. Equivalently,  $S$  could be made the three-bit xor of  $A$ ,  $B$ , and  $C_i$  and  $C_o$  could be made the three-bit majority function of  $A$ ,  $B$ , and  $C_i$ . The output of the full adder is the two-bit arithmetic sum of three one-bit numbers.

# **CHAPTER 5**

**BINARY MULTIPLIER  
SERIAL AND PARALLEL MULTIPLIERS**

## **BINARY MULTIPLIER**

A Binary multiplier is an electronic hardware device used in digital electronics or a computer or other electronic device to perform rapid multiplication of two numbers in binary representation. It is built using binary adders.

The rules for binary multiplication can be stated as follows

1. If the multiplier digit is a 1, the multiplicand is simply copied down and represents the product.
2. If the multiplier digit is a 0 the product is also 0.

For designing a multiplier circuit we should have circuitry to provide or do the following three things:

1. it should be capable identifying whether a bit is 0 or 1.
2. It should be capable of shifting left partial products.
3. It should be able to add all the partial products to give the products as sum of partial products.
4. It should examine the sign bits. If they are alike, the sign of the product will be a positive, if the sign bits are opposite product will be negative. The sign bit of the product stored with above criteria should be displayed along with the product.

From the above discussion we observe that it is not necessary to wait until all the partial products have been formed before summing them. In fact the addition of partial product can be carried out as soon as the partial product is formed.



Notations:

- $a$  – multiplicand
- $b$  – multiplier
- $p$  – product

Binary multiplication (eg  $n=4$ )

$$p = a \times b$$

$a_{n-1} a_{n-2} \dots a_1 a_0$

$b_{n-1} b_{n-2} \dots b_1 b_0$

$p_{2n-1} p_{2n-2} \dots p_1 p_0$

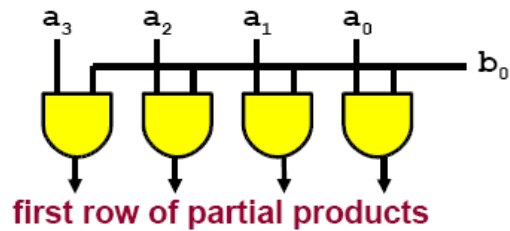
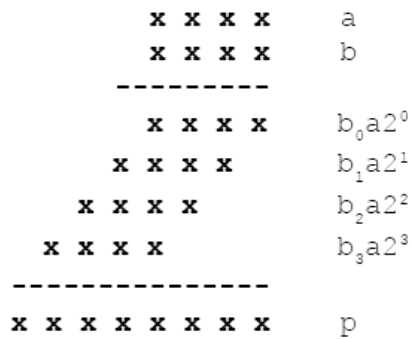
x x x x	a
x x x x	b
-----	
x x x x	b <sub>0</sub> a <sub>20</sub>
x x x x	b <sub>1</sub> a <sub>21</sub>
x x x x	b <sub>2</sub> a <sub>22</sub>
x x x x	b <sub>3</sub> a <sub>23</sub>
-----	
x x x x x x x x	p

# Basic hardware multiplier

## Partial products

In binary, the partial products are trivial –  
*if* multiplier bit = 1, copy the multiplicand  
*else* 0

Use an 'AND' gate!



## MULTIPLY ACCUMULATE CIRCUITS

Multiplication followed by accumulation is a operation in many digital systems , particularly those highly interconnected like digital filters,neural networks, data quantisers, etc.

One typical MAC(multiply-accumulate) architecture is illustrated in figure. It consists of multiplying 2 values, then adding the result to the previously accumulated value, which must then be restored in the registers for future accumulations. Another feature of MAC circuit is that it must check for overflow, which might happen when the number of MAC operation is large .

This design can be done using component because we have already design each of the units shown in figure. However since it is relatively simple circuit, it can also be designed directly. In any case the MAC circuit, as a whole, can be used as a component in application like digital filters and neural networks

## ARCHITECTURE OF A RADIX $2^n$ MULTIPLIER

The architecture of a radix  $2^n$  multiplier is given in the Figure. This block diagram shows the multiplication of two numbers with four digits each. These numbers are denoted as  $V$  and  $U$  while the digit size was chosen as four bits. The reason for this will become apparent in the following sections. Each circle in the figure corresponds to a radix cell which is the heart of the design. Every radix cell has four digit inputs and two digit outputs. The input digits are also fed through the corresponding cells. The dots in the figure represent latches for pipelining. Every dot consists of four latches. The ellipses represent adders which are included to calculate the higher order bits. They do not fit the regularity of the design as they are used to “terminate” the design at the boundary. The outputs are again in terms of four bit digits and are shown by  $W$ 's. The  $l$ 's denote the clock period at which the data appear.

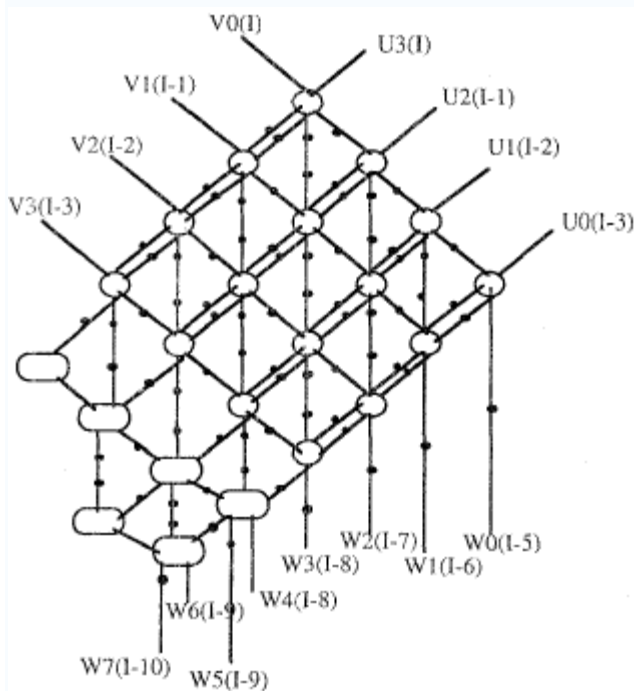


Fig. 1. Radix  $2^{11}$  multiplier architecture

## BOOTH MULTIPLIER

The decision to use a Radix-4 modified Booth algorithm rather than Radix-2 Booth algorithm is that in Radix-4, the number of partial products is reduced to  $n/2$ . Though Wallace Tree structure multipliers could be used but in this format, the multiplier array becomes very large and requires large numbers of logic gates and interconnecting wires which makes the chip design large and slows down the operating speed.

# Booth Multiplication Algorithm

### Booth Multiplication Algorithm for radix 2

Booth algorithm gives a procedure for multiplying binary integers in signed  $-2$ 's complement representation.

I will illustrate the booth algorithm with the following example:

Example,  $2_{\text{ten}} \times (-4)_{\text{ten}}$

$0010_{\text{two}} * 1100_{\text{two}}$

#### Step 1: Making the Booth table

I. From the two numbers, pick the number with the smallest difference between a series of consecutive numbers, and make it a multiplier.

i.e., 0010 -- From 0 to 0 no change, 0 to 1 one change, 1 to 0 another change, so there are two changes on this one

1100 -- From 1 to 1 no change, 1 to 0 one change, 0 to 0 no change, so there is only one change on this one.

Therefore, multiplication of  $2 \times (-4)$ , where  $2_{\text{ten}}$  ( $0010_{\text{two}}$ ) is the multiplicand and  $(-4)_{\text{ten}}$  ( $1100_{\text{two}}$ ) is the multiplier.

II. Let  $X = 1100$  (multiplier)

Let  $Y = 0010$  (multiplicand)

Take the 2's complement of  $Y$  and call it  $-Y$

$-Y = 1110$

III. Load the  $X$  value in the table.

IV. Load 0 for  $X-1$  value it should be the previous first least significant bit of  $X$

V. Load 0 in  $U$  and  $V$  rows which will have the product of  $X$  and  $Y$  at the end of operation.

VI. Make four rows for each cycle; this is because we are multiplying four bits numbers.

U	V	X	X-1
0000	0000	1100	0

Load the value  
1<sup>st</sup> cycle  
2<sup>nd</sup> cycle  
3<sup>rd</sup> Cycle  
4<sup>th</sup> Cycle

### Step 2: Booth Algorithm

Booth algorithm requires examination of the multiplier bits, and shifting of the partial product. Prior to the shifting, the multiplicand may be added to partial product, subtracted from the partial product, or left unchanged according to the following rules:

Look at the first least significant bits of the multiplier " $X$ ", and the previous least significant bits of the multiplier " $X - 1$ ".

1 0 0 Shift only

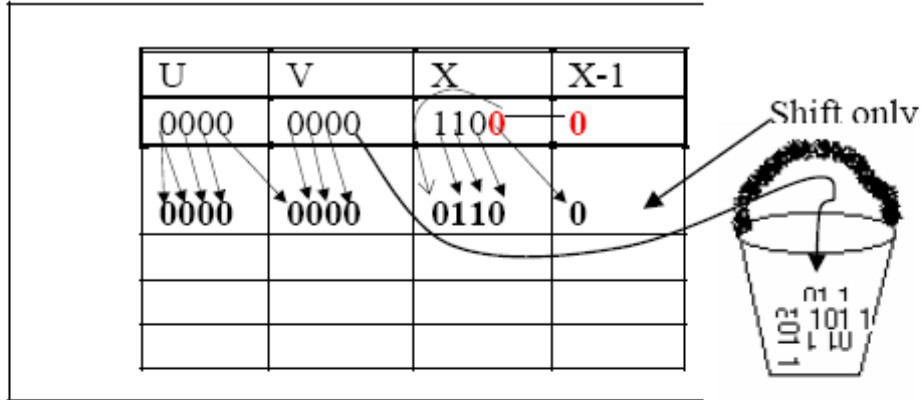
1 1 Shift only.

0 1 Add  $Y$  to  $U$ , and shift

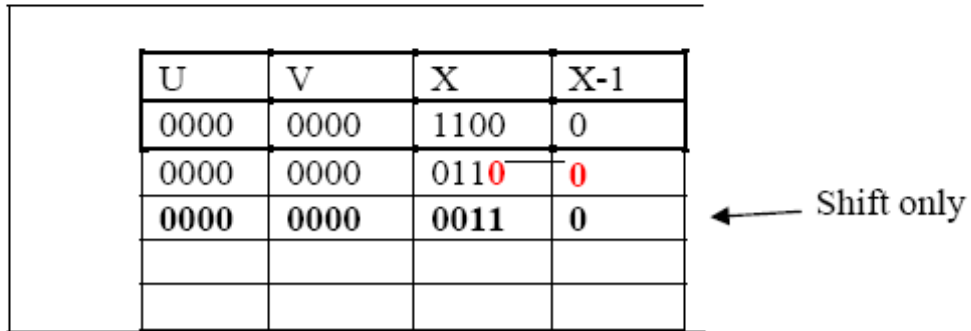
1 0 Subtract  $Y$  from  $U$ , and shift or add  $(-Y)$  to  $U$  and shift

II Take  $U$  &  $V$  together and shift arithmetic right shift which preserves the sign bit of 2's complement number. Thus a positive number remains positive, and a negative number remains negative.

III Shift X circular right shift because this will prevent us from using two registers for the X value.



Repeat the same steps until the four cycles are completed.



U	V	X	X-1
0000	0000	1100	0
0000	0000	0110	0
0000	0000	0011	0
<b>1110</b>	<b>0000</b>	<b>0011</b>	<b>0</b>
<b>1111</b>	<b>0000</b>	<b>1001</b>	<b>1</b>

← Add -Y (0000 + 1110 = 1110)  
 ← Shift

U	V	X	X-1
0000	0000	1100	0
0000	0000	0110	0
0000	0000	0011	0
1110	0000	0011	0
1111	0000	1001	1
<b>1111</b>	<b>1000</b>	<b>1100</b>	<b>1</b>

← Shift only

We have finished four cycles, so the answer is shown, in the last rows of U and V which is:  $11111000_{two}$

Note: By the fourth cycle, the two algorithms have the same values in the Product register.

### Booth multiplication algorithm for radix 4

One of the solutions of realizing high speed multipliers is to enhance parallelism which helps to decrease the number of subsequent calculation stages. The original version of the Booth algorithm (Radix-2) had two drawbacks. They are: (i) The number of addsubtract operations and the number of shift operations becomes variable and becomes inconvenient in designing parallel multipliers. (ii) The algorithm becomes inefficient when there are isolated 1's. These problems are overcome by using modified Radix4 Booth algorithm which scan strings of three bits with the algorithm given below:

- 1) Extend the sign bit 1 position if necessary to ensure that  $n$  is even.
- 2) Append a 0 to the right of the LSB of the multiplier.
- 3) According to the value of each vector, each Partial Product will be 0,  $+y$ ,  $-y$ ,  $+2y$  or  $-2y$ .

The negative values of  $y$  are made by taking the 2's complement and in this paper Carry-look-ahead (CLA) fast adders are used. The multiplication of  $y$  is done by shifting  $y$  by one bit to the left. Thus, in any case, in designing a  $n$ -bit parallel multipliers, only  $n/2$  partial products are generated.

$X(i)$	$X(i-1)$	$X(i-2)$	$y$
0	0	0	+0
0	0	1	+ $y$
0	1	0	+ $y$
0	1	1	+2 $y$
1	0	0	-2 $y$
1	0	1	- $y$
1	1	0	- $y$
1	1	1	+0

Table I Radix4 Modified Booth algorithm scheme for odd values of  $i$ .





# CHAPTER 6

ANALYSIS  
VHDL CODES FOR MULTIPLIER

## ANALYSIS

### VHDL code for sixteen bit adder

---

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity sixteenbit_fa is
    Port ( a : in  STD_LOGIC_VECTOR (15 downto 0);
          b : in  STD_LOGIC_VECTOR (15 downto 0);
          -- cin : in  STD_LOGIC;
          yout : out STD_LOGIC_VECTOR (15 downto 0);
          cout : out STD_LOGIC);
end sixteenbit_fa;

architecture Behavioral of sixteenbit_fa is
    signal s: std_logic_vector(15 downto 0);
    signal carry1: std_logic_vector(16 downto 0);
    COMPONENT twobit_add
        PORT(
            a : IN std_logic;
            b : IN std_logic;
            cin : IN std_logic;
```

```

        sum : OUT std_logic;
        cout : OUT std_logic
    );
END COMPONENT;

begin
carry1(0)<='0';
g1 : for i in 0 to 15 generate
    f0 : twobit_add PORT MAP(a(i), b(i),carry1(i),yout(i), carry1(i+1));
--    inter_carr<=carry(i+1);
end generate g1;
cout<=carry1(16);
end Behavioral;

```

### **VHDL code for array multiplier**

```

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity mult64 is
    Port ( clk:in std_logic;
          --rst:in std_logic;

```

```

        a : in std_logic_vector(7 downto 0);
        b : in std_logic_vector(7 downto 0);
        prod : out std_logic_vector(15 downto 0));
end mult64;

```

architecture Behavioral of mult64 is

```

constant n:integer :=8;
subtype plary is std_logic_vector(n-1 downto 0);
type pary is array(0 to n) of plary;
signal pp,pc,ps:pary;

```

```

begin

```

```

pgen:for j in 0 to n-1 generate
pgen1:for k in 0 to n-1 generate
pp(j)(k)<=a(k) and b(j);
end generate;
pc(0)(j)<='0';
end generate;
ps(0)<=pp(0);
prod(0)<=pp(0)(0);
addr:for j in 1 to n-1 generate
addc:for k in 0 to n-2 generate
ps(j)(k)<=pp(j)(k) xor pc(j-1)(k) xor ps(j-1)(k+1);
pc(j)(k)<=(pp(j)(k) and pc(j-1)(k)) or
(pp(j)(k) and ps(j-1)(k+1)) or
(pc(j-1)(k)and ps(j-1)(k+1));
end generate;
prod(j)<=ps(j)(0);
ps(j)(n-1)<=pp(j)(n-1);

```

```

end generate;
pc(n)(0)<='0';

addlast:for k in 1 to n-1 generate
ps(n)(k)<=pc(n)(k-1) xor pc(n-1)(k-1) xor ps(n-1)(k);
pc(n)(k)<=(pc(n)(k-1) and pc(n-1)(k-1)) or
      (pc(n)(k-1) and ps(n-1)(k)) or
      (pc(n-1)(k-1)and ps(n-1)(k));
end generate;
prod(2*n-1)<=pc(n)(n-1);
prod(2*n-2 downto n)<=ps(n)(n-1 downto 1);

end Behavioral;

```

### **VHDL code for booth encoder**

```

-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use ieee.std_logic_arith.all;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

```

entity booth\_mult is

```
Port ( a : in STD_LOGIC_VECTOR (7 downto 0);  
      b : in STD_LOGIC_VECTOR (7 downto 0);  
      yout : out STD_LOGIC_VECTOR (15 downto 0);  
      ovf: out std_logic);
```

end booth\_mult;

architecture Behavioral of booth\_mult is

COMPONENT booth\_encoder

```
PORT(  
      a : IN std_logic_vector(7 downto 0);  
      arg : IN std_logic_vector(2 downto 0);  
      pprod : OUT std_logic_vector(15 downto 0)  
      );  
END COMPONENT;
```

COMPONENT sixteenbit\_fa

```
PORT(  
      a : IN std_logic_vector(15 downto 0);  
      b : IN std_logic_vector(15 downto 0);  
      yout : OUT std_logic_vector(15 downto 0);  
      cout : OUT std_logic  
      );  
END COMPONENT;
```

```
signal pp1,pp2,pp3,pp4,s1,s2,s3,sum1,sum2,sum3: std_logic_vector(15 downto 0);
```

```
signal st: std_logic_vector(2 downto 0);
```

```
signal k1,k2,k3: std_logic;
```

begin

```
st<=b(1 downto 0)&'0';
```

```
u0: booth_encoder PORT MAP(a,st,pp1);
```

```
u1: booth_encoder PORT MAP(a,b(3 downto 1),pp2);
```

```

u2: booth_encoder PORT MAP(a,b(5 downto 3),pp3);
u3: booth_encoder PORT MAP(a,b(7 downto 5),pp4);
s1<=pp2(13 downto 0)&"00";
s2<=pp3(11 downto 0)&"0000";
s3<=pp4(9 downto 0)&"000000";
u4: sixteenbit_fa PORT MAP(pp1,s1,sum1,k1);
u5: sixteenbit_fa PORT MAP(sum1,s2,sum2,k2);
u6: sixteenbit_fa PORT MAP(sum2,s3,yout,ovf);

end Behavioral;

```

#### **VHDL code for booth multiplier radix 4**

```

-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use ieee.numeric_std.all;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity booth_encoder is
    -- generic(N : integer:=8);

```

```

Port ( a : in std_logic_vector(7 downto 0);
      arg : in std_logic_vector(2 downto 0);
      pprod : out std_logic_vector(15 downto 0));
end booth_encoder;

```

architecture Behavioral of booth\_encoder is

```

function encoder(arg1: std_logic_vector(2 downto 0);data:std_logic_vector(7
downto 0))

```

```

return std_logic_vector is

```

```

variable temp,temp1,temp2: std_logic_vector(8 downto 0);

```

```

variable sign: std_logic;

```

```

begin

```

```

    case arg1 is

```

```

        when "001"|"010" =>

```

```

            if data <0 then

```

```

                temp:='1'& data;

```

```

            else

```

```

                temp:='0'&data;

```

```

            end if;

```

```

        when "011" =>

```

```

            if data<0 then

```

```

                temp1:='1'&data;

```

```

                temp:=temp1(7 downto 0)&'0';

```

```

            else

```

```

                temp:='0'&data(6 downto 0)&'0';

```

```

            end if ;

```

```

        when "100" =>

```

```

            if data<0 then

```

```

                temp1:='1'&data;

```

```

                temp2:=(not temp1)+"00000001";

```

```

                temp:=(temp2(7 downto 0)&'0');

```



```

        else
            temp1:='0'&data;
            temp2:=(not temp1)+"000000001";
            temp:=(temp2(7 downto 0)&'0');
        end if;
    when "101"|"110" =>
        if data < 0 then
            temp1:='1'&data;
            temp:=not(temp1)+"000000001";
        else
            temp1:='0'&data;
            temp:=(not temp1)+"000000001";
        end if;
    when others =>
        temp:="000000000";
        --"(others=>'0');

    end case;
    return temp;
end encoder;

signal s1: std_logic_vector(8 downto 0);
signal s2: std_logic;

begin
s1<=encoder(arg,a);
--s2<=s1(8);

--pprod<=s2&s2&s2&s2&s2&s2&s2&s2&s1(7 downto 0);
pprod<=sxt(s1,16);
end Behavioral;

```

## VHDL code for FIR filter

---

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity fir is
--    GENERIC (n: INTEGER :=4; m:INTEGER := 8);
    PORT(x: in std_logic_vector(7 downto 0);
         clk, rst: in std_logic;
         y: out std_logic_vector(15 downto 0));
end fir;

architecture Behavioral of fir is
    COMPONENT booth_mult
        PORT(
            a : IN std_logic_vector(7 downto 0);
            b : IN std_logic_vector(7 downto 0);
            yout : OUT std_logic_vector(15 downto 0);
            ovf: out std_logic
        );
    END COMPONENT;
    COMPONENT sixteenbit_fa
        PORT(
```

```

        a : IN std_logic_vector(15 downto 0);
        b : IN std_logic_vector(15 downto 0);
        yout : OUT std_logic_vector(15 downto 0);
        cout : OUT std_logic
    );
END COMPONENT;
type registers is array (2 downto 0) of std_logic_vector(7 downto 0);
    type coefficients is array (3 downto 0) of std_logic_vector(7
downto 0);
    type product is array (3 downto 0) of std_logic_vector(15 downto
0);

    signal reg:registers;
    constant          coef:coefficients
:=("00000001","00010110","00110011","00110000");
    signal c,p:std_logic;
        --    signal acc: std_logic_vector(15 downto 0):=(others=>'0');
        signal prod,acc:product;
    signal sign: std_logic;
        signal c1,p1:std_logic_vector(2 downto 0);

begin
--
--    variable acc,prod:
--        signed(2*m-1 downto 0):=(others=>'0');
--        variable sign: std_logic;
--        begin
--
--
--

```

```

a1:booth_mult PORT MAP(coef(0),x,acc(0),p);
m1:for i in 1 to 3 generate
    sign <=acc(3)(15);
    a2: booth_mult PORT MAP(coef(i),reg(3-i),prod(i-1),p1(i-1));
    a3: sixteenbit_fa PORT MAP(acc(i-1),prod(i-1),acc(i),c1(i-1));
end generate m1;

process(clk,rst,sign)
begin
    if (rst='1') then
        for i in 2 downto 0 loop
            for j in 7 downto 0 loop
                reg(i)(j)<='0';
            end loop;
        end loop;
    elsif(clk'event and clk='1')then
        reg(0)<=reg(1);
        reg(1)<=reg(2);
        reg(2)<=x;
    end if;

-- if ((sign=prod(prod'left)) then
-- if (acc(acc'left)/= sign))then
-- acc <=(acc'left => sign,others=> not sign);
-- end if;

end process;
-- reg <= x& reg(n-2 downto 1);

```

```
---- end if;
process(clk,rst)
begin
if(rst='1') then
y<="0000000000000000";
elsif(clk'event and clk='1')then
y<= acc(3);
end if;
end process;
end Behavioral;
```

# CHAPTER 7

RESULTS

## RESULTS OF DIFFERENT MULTIPLIERS

### ARRAY MULTIPLIER

Number of Slices	229
Number of 4 input LUTs	302
Number of bonded INPUT	16
Number of bonded OUTPUT	16
CLB Logic Power	104mW

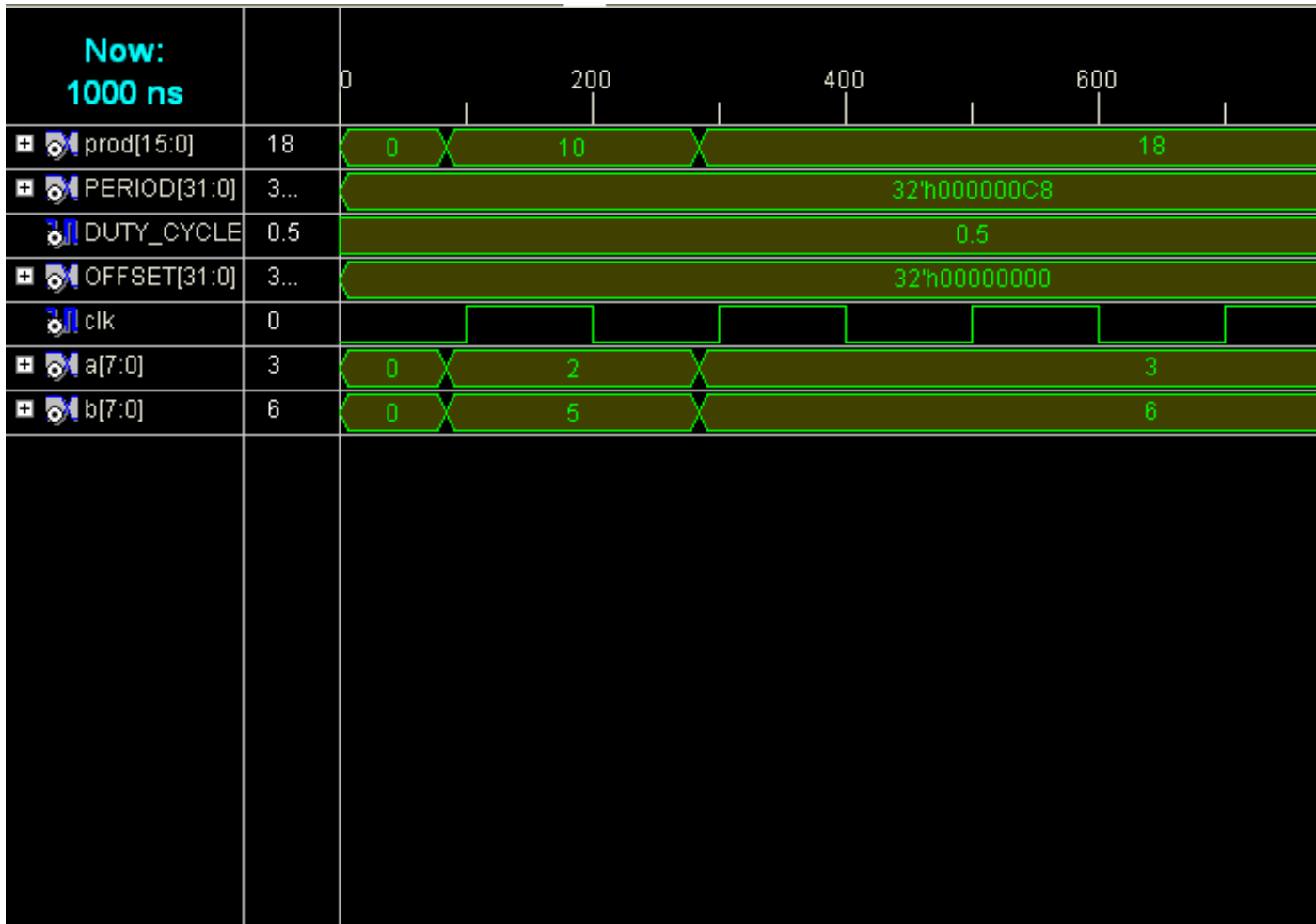
### RADIX 2 BOOTH MULTIPLIER

Number of Slices	130
Number of 4 input LUTs	249
Number of bonded INPUT	16
Number of bonded OUTPUT	17
CLB Logic Power	79mW

### RADIX 4 BOOTH MULTIPLIER

Number of Slices	229
Number of 4 input LUTs	302
Number of bonded INPUT	16
Number of bonded OUTPUT	16
CLB Logic Power	47mW

# MULTIPLIER OUTPUT





# **CHAPTER 8**

**CONCLUSION**

## CONCLUSION

Our project gives a clear concept of different multiplier and their implementation in tap delay FIR filter. We found that the parallel multipliers are much option than the serial multiplier. We concluded this from the result of power consumption and the total area. In case of parallel multipliers, the total area is much less than that of serial multipliers. Hence the power consumption is also less. This is clearly depicted in our results. This speeds up the calculation and makes the system faster.

While comparing the radix 2 and the radix 4 booth multipliers we found that radix 4 consumes lesser power than that of radix 2. This is because it uses almost half number of iteration and adders when compared to radix 2.

When all the three multipliers were compared we found that array multipliers are most power consuming and have the maximum area. This is because it uses a large number of adders. As a result it slows down the system because now the system has to do a lot of calculation.

Multipliers are one the most important component of many systems. So we always need to find a better solution in case of multipliers. Our multipliers should always consume less power and cover less power. So through our project we try to determine which of the three algorithms works the best. In the end we determine that radix 4 modified booth algorithm works the best.

## REFERENCES

Websites referred:

1. [www.wikipedia.com](http://www.wikipedia.com)
2. [www.howstuffwork.com](http://www.howstuffwork.com)
3. [www.xilinx.com](http://www.xilinx.com)

Books referred:

1. Circuit Design using VHDL, by Pedroni , page number 285-293.
2. VHDL by Sjöholm Stefan
3. VHDL by B Bhaskar
4. Digital Signal Processing by Johny R Johnson ,PHI publications.
5. Digital Signal Processing by Vallavraj & Salivhanan, TMH publications.