IMPLEMENTATION OF FIR FILTERS IN HARDWARE DESCRIPTION LANGUAGE (HDL)

By

TONG KIN WAH

FINAL REPORT

Submitted to the Electrical & Electronics Engineering Programme
in Partial Fulfillment of the Requirements
for the Degree
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Universiti Teknologi Petronas
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

IMPLEMENTATION OF FIR FILTERS IN HARDWARE DESCRIPTION LANGUAGE (HDL)

by

Tong Kin Wah

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfillment of the requirement for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Approved by:

Azrina Binti Abd. Aziz
Project Supervisor

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK

June 2006

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

Tong Kin Wah

ABSTRACT

Digital filters are used in digital signal processing (DSP) to improve the quality of a signal, to extract information from signals or to separate two or more signals previously combined. The advancements in VLSI technology have seen the growing popularity of digital filters rather than analog filters. Due to a surge in high performance portable systems, there is a continuous drive for methodologies and approaches of low power and high throughput FIR filter cores. The components of an FIR filter include adders, multipliers, memory unit and control unit. This project intends to compare the performances of different structures of adders and multipliers and integrate these structures to yield a filter which displays the best performance in terms of area, speed and power consumption. The hardware implementation of FIR filters is done using Verilog Hardware Description Language (HDL). All the filter components are modeled using HDL, in which they are then synthesized, implemented and simulated. The simulated design that has been verified is downloaded into Field Programmable Gate Array (FPGA), where Xilinx Virtex-II chip is used. Hardware verification is performed by testing the filter output using a logic analyzer. Important considerations in this project are the selection of appropriate number of bits for input samples and filter coefficients, and also the number representation scheme. The choices made will affect the performance of the filter. This project brings out the importance of exploring varies structures of adders and multipliers that will improve filter performance. This area of study is lacking although there exists innumerable research on advance techniques to implement low power and high throughput filter. The designed FIR filter in this project can be further improved by comparing more structures of adders and multipliers, and incorporating some advance techniques.

ACKNOWLEDGEMENTS

This design project has equipped me with abundance knowledge and it would not be a success without the help of a legion of people. First and foremost, I would like to express my heartfelt gratitude to my supervisor, Azrina, who has not failed to attend to my needs. She is indeed very helpful in attempting to provide solutions to my problems and lead me to the resources that are of great help. I would also want to thank her for directing one of my problems to her friend, Weng Fook Lee, who has actually provided me with suggestions that guide me through the design process.

I am also indebted to three lecturers, Mr. Lo, Mr. Patrick and Dr. Yap, who have helped and guided me much in this project. I want to thank them for spending hours with me in debugging and for their precious piece of advice. Besides, they are patient with all my inquiries and are always willing to lend a helping hand. Not forgetting also to give my thanks to the lab technician, Kak Azira, for the eagerness to help in every way regarding the lab equipment. It would be a tough time without her help in installing the software and obtaining the lab equipment and manuals.

There is another person whom I owe my thanks to – Kuang Sun, who is one of Mr. Lo's FYP students. He is of tremendous help in my project since a part of his project is rather similar to mine. With his help and advice in using the software and lab equipment, a lot of time is saved and more focus can be put into the design. Lastly, I want to take this opportunity to thank everyone who has directly or indirectly involved in this project, be it offering technical information or giving other useful advice. Once again, thank you so much for providing me with a wonderful experience in completing this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ASIC	–	Application-Specific Integrated Circuit
CLA	–	Carry-Look-Ahead Adder
CSA	–	Carry-Save Adder
DF	–	Direct Form
DSP	–	Digital Signal Processing
FIR	–	Finite Impulse Response
FPGA	–	Field Programmable Gate Array
HDL	–	Hardware Description Language
IIR	–	Infinite Impulse Response
IOB	–	Input/Output Block
TDF	–	Transpose Direct Form
VHDL	–	Very high speed HDL
VLSI	–	Very Large Scale Integration

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND OF STUDY

Digital filtering is one of the most important operations in digital signal processing (DSP). Digital filters are widely used in any area where information is handled in digital form or controlled by a digital processor. The continuous growing trend towards digital solutions can be seen in all areas – from electronic instrumentation, control, data manipulation, signals processing, telecommunication to consumer electronics. Due to the advancements in VLSI technology, digital filters are fabricated with greater reliability, smaller size, lower cost, lower power consumption and higher operation speed.

The objectives of using digital filters in DSP are to improve the quality of a signal (for example, to remove or reduce noise), to extract information from signals or to separate two or more signals previously combined. The use of digital filters is especially important to minimize the distortion of the in-band signal components. For instance, digital filter is used in speech synthesis – the Speak and Spell is an example in which it is an electronic learning aid for children and uses the LPC (linear predictive coding) techniques, where the actual human speech to be reproduced later is modeled as the response of a time-varying digital filter to a periodic or random excitation signal.

There is a continuous demand for low power and high throughput FIR filtering cores in DSP architectures. Researches in the literature have developed a number of techniques to implement digital filters in achieving the above purposes. These include the following: use of differential coefficients, wordlength optimization, multirate architectures and dynamic adjustment of filter order [1,2]. Other techniques introduced by researches include coefficient segmentation, block processing and combined segmentation and block processing algorithms, as demonstrated in [3,4,5]. The choice of number representation scheme, investigated in [6,7], can affect the filter performance.

Digital filters are normally modeled using software simulation and then synthesized into corresponding hardware circuit using field programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs). A Hardware Description Language (HDL) provides the framework for the complete logical design. Verilog and VHDL are the two most commonly used HDLs today. Verilog as an HDL was introduced by Cadence Design Systems; they placed it into the public domain in 1990. It was established as a formal IEEE Standard in 1995. The revised version has been brought out in 2001.

Software simulators offer flexible schemes to code the algorithm from a choice of many languages but cannot always offer the speed that a hardware simulator can. Unfortunately, building hardware prototypes to model different systems can be costly and time consuming when constant changes have to be made. Therefore, a middle ground might be found using custom computing platforms or programmable logic. Such systems can offer similar flexibility as software and still retain some or all of the hardware acceleration at the cost of a shorter implementation cycle.

FPGAs are becoming increasingly popular for rapid prototyping of designs with the aid of software simulation and synthesis. Software synthesis tools translate high-level language descriptions of the implementation into formats that may be loaded directly into the FPGAs. An increasing number of design changes through software synthesis become more cost effective than similar changes done for hardware prototypes. In addition, the implementation may be constructed on existing hardware to help further reduce the cost.

1.2 PROBLEM STATEMENT

The requirement of this project title is to implement FIR filters using suitable Hardware Description Language (HDL). The design can then be synthesized into hardware circuit using FPGA. In fact, there are innumerable methodologies and techniques used to implement low power and high throughput FIR filtering cores, as

discussed in [1,2]. The components of a filter include adders, multipliers, memory unit and control unit. Different structures of adders and multipliers will give different performance. Hence, this project aims at modeling the components in HDL and investigating the performance of different structures of adders and multipliers using a simulation tool. The performance is to be viewed in terms of structure size, speed and power consumption.

The adder and multiplier structures, that give the best performance, are to be used in the filter design and the overall filter performance is analyzed. Once software simulation is completed and successful, the final filter design is downloaded into FPGA and verified to ensure that the filter is functioning properly. Performance comparison analyses among various structures of adder and multiplier are lacking since many researches currently focus on the filter implementation techniques. Hence, this project brings out the importance of investigating the structures of adders and multipliers.

1.3 OBJECTIVES

1. To develop software simulations for FIR filters using Verilog HDL.
2. To compare the performance of the different structures of adders and multipliers in relation to area, speed and power consumption.
3. To select the structures of adder and multiplier with the best performance and integrate them with memory unit and control unit to build the overall filter.
4. To select a suitable computational arithmetic (unsigned, signed, fixed or floating point) and the number of bits to represent filter coefficients and input data.
5. To synthesize the filter design into hardware using FPGA and verify its functionality using appropriate equipment.

1.4 SCOPE OF STUDY

1. The concepts and theories of FIR filters are learnt.
2. The design methodology for FIR filters from specifications, coefficients calculation, filter structure, finite wordlength effects to filter implementation are learnt.

3. A suitable data processing style and computational arithmetic for representing the input samples and filter coefficients are decided upon.
4. Each component of the filter (adders, multipliers, memory unit and control unit) is coded into Verilog and their functionalities are verified.
5. Different types of adders and multipliers are explored. The performance of different structures of each component is compared in terms of area, speed and power consumption.
6. All components are integrated to form a complete filter. The final design is verified functionally and a detail analysis is done.
7. The debugged design in HDL is synthesized into corresponding hardware circuit through FPGA.

CHAPTER 2

LITERATURE REVIEW/THEORY

2.1 DIGITAL FIR FILTERS

A filter is essentially a system or network that selectively changes the wave shape, amplitude-frequency and/or phase-frequency characteristics of a signal in a desired manner. A digital filter is a mathematical algorithm implemented in hardware and/or software that operates on a digital input signal to produce a digital output signal for the purpose of achieving filtering objective. Digital filters often operate on digitized analog signals or just numbers, representing some variable.

A simplified block diagram of a real-time digital filter, with analog input and output signals, is given in Figure 1. The bandlimited analog signal is sampled periodically and converted into a series of digital samples, *x(n)*. The digital processor implements the filtering operation, mapping the input sequence, *x(n)*, into the output sequence, *y(n)*, in accordance with a computational algorithm for the filter. The DAC converts the digitally filtered output into analog values which are then analog filtered to smooth and remove unwanted high frequency components [8].

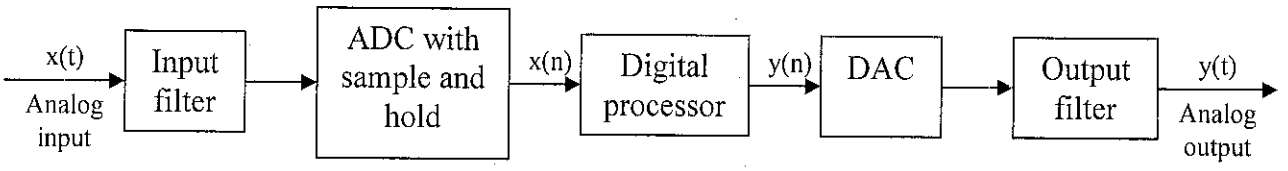


Figure 1 A simplified block diagram of a real-time digital filter with analog input and output signals [8]

Digital filters play important roles in DSP. Compared to analog filters, they are preferred in a number of applications; for example, data compression, biomedical signal processing, speech processing, image processing, data transmission, digital audio and telephone echo cancellation. The advantages and disadvantages of digital filters compared to analog filters are summarized in Table 1.

Table 1 Advantages and disadvantages of digital filters [8]

Advantages	Disadvantages
Can have truly linear phase response.	Speed limitation. Operating speed of digital filters depend on speed of digital processor used and the number of arithmetic operations performed.
Performance of filters does not vary with environmental changes – eliminates the need to calibrate periodically.	
Frequency response can be automatically adjusted if it is implemented using a programmable processor.	
Several input signals or channels can be filtered by one digital filter without replicating the hardware.	Finite wordlength effects. Digital filters are subjected to ADC noise resulting from quantizing a continuous signal and to roundoff noise incurred during computation.
Both filtered and unfiltered data can be saved for further use.	
Can be fabricated small in size and consume low power due to advancements in VLSI technology.	Long design and development times. Hardware development for digital filters can consume a longer time than for analog filters.
More flexible in terms of precision – only limited by the wordlength used.	
Can be made to work over a wide range of frequencies even at very low frequencies.	

Digital filters can be divided into two categories, namely infinite impulse response (IIR) and finite impulse response (FIR) filters. Either type of filter, in its basic form, can be represented by its impulse response sequence, $h(k)$ as in Figure 2. The choice between FIR and IIR filters depends largely on the relative advantages of the two filter types (See Table 2).

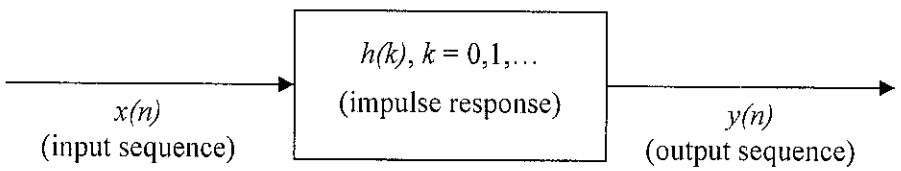


Figure 2 A conceptual representation of a digital filter

Table 2 Comparison between FIR and IIR filters [8]

FIR filter	IIR filter
Can have exactly linear phase response	Nonlinear phase response, especially at band edges
Nonrecursive, always stable	Stability problems
Finite wordlength effects are much less severe	Finite wordlength effects are more severe
Requires more processing time and storage for a given amplitude response specification	Less coefficients leading to less processing time and storage
Filters with arbitrary frequency responses are easier to be synthesized	Analog filters are readily transformed into equivalent IIR filters meeting similar specifications

The basic FIR filter is characterized by the following two equations:

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) \qquad \text{----------} \qquad \text{Equation 1}$$

$$H(z) = \sum_{k=0}^{N-1} h(k)z^{-k} \qquad \text{----------} \qquad \text{Equation 2}$$

where *h(k)* are the impulse response coefficients of the filter, *H(z)* is the transfer function of the filter and *N* is the filter length, which is the number of filter coefficients. The sole objective of most FIR coefficient calculation (or approximation) methods is to obtain values of *h(n)* such that the resulting filter meets the design specifications. Several methods are available to obtain *h(n)* and the most commonly used are window, optimal (Parks-McClellan) and frequency sampling methods. All three lead to linear phase FIR filters.

The number of bits used to represent the input data to the filter and the filter coefficients and in performing arithmetic operations must be small for efficiency and to limit the cost of the digital filter. The problems caused by using a finite number of bits are referred to as finite wordlength effects and can lead to performance degradation of the filter. Finite wordlength effects include [8]:

i) *ADC noise*. ADC quantization noise which results when the filter input is derived from analog signals.

ii) *Coefficient quantization errors*. These result from representing filter coefficients with a limited number of bits.

iii) *Roundoff errors from quantizing results of arithmetic operations*. This may be caused by the wordlength of the processor used.

iv) *Arithmetic overflow*. This occurs when partial sums or filter output exceeds the permissible wordlength of the system.

The computation of output sequence, $y(n)$ involves multiplications, additions/subtractions and delays. Thus, filter implementation needs the following basic components:

i) memory (RAM) to store the present and past input samples, $x(n)$ and $x(n-k)$

ii) memory (RAM or ROM) for storing the filter coefficients, the $h(k)$

iii) multipliers to multiply input samples and filter coefficients

iv) adders to sum the outputs from multipliers

v) control unit to schedule the operations of all components in a filter

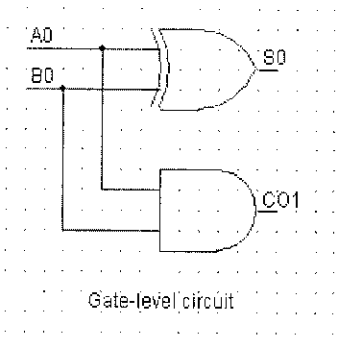
2.2 TWO's COMPLEMENT

Two's complement number representation is used to represent signed numbers. This form of representation is also known as radix complement (RC) representation. Two's complement is selected over other representation schemes because it is able to perform signed addition and multiplication using the same circuitry as in unsigned addition and multiplication. To obtain the two's complement of a number, first complement (negate) all the bits in the number, including the sign bit and all magnitude bits, then add one to the least significant bit of the number. In order to add or multiply two 4-bit operands, signed extension needs to be carried out beforehand, so that the MSB is the sign bit and all four bits are magnitude bits. For example, integer 5 is represented by 00101_2 while integer -5 is represented by 11011_2. Hence, addition of two 4-bit operands requires a 5-bit adder.

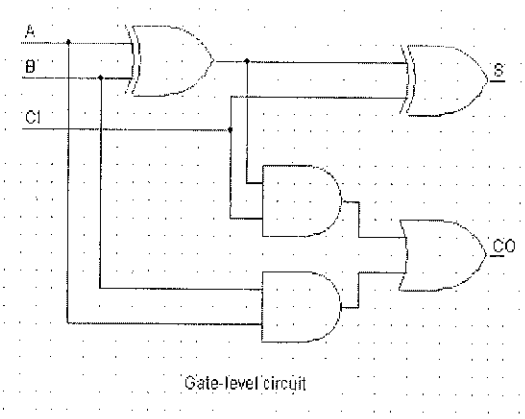
2.3 ADDERS

The iterative design process is used to design adder and subtractor circuits at gate level. Two's complement representation of signed numbers is used so that subtraction can be done using the same circuitry as in addition. The two basic adders are half adder (HA) and full adder (FA). A half adder is capable of adding two 1-bit operands while a full adder can add two 1-bit operands and an input carry. Both adders result in two outputs – a sum and an output carry. The gate-level circuits and equations for half adder and full adder are shown in Figure 3.

Higher bits adders are formed by employing the full adders and half adders where appropriate in an iterative modular design process. Examples of higher bit adders are ripple-carry adders, carry-look-ahead adders, carry-save adders, carry-select adders and carry-skip adders.



Equations:
$S0 = A0 \oplus B0$
$CO1 = A0.B0$
(a)



Equations:
$S = CI \oplus A \oplus B$
$CO = A.B+CI.A+CI.B$
or
$CO = (A \oplus B).CI+A.B$
(b)

Figure 3 Gate-level circuits and equations for (a) half adder and (b) full adder

2.3.1 Carry-Look-Ahead Adder (CLA)

The 4-bit CLA showing the carry-out circuitry is indicated in Figure 4. This figure assumes that there is no input carry at bit position 0. The propagation delay times shown in parentheses for the carry-out bits and the sum bits for the CLA are substantially smaller than that of ripple-carry adder as the number of stages increases. The CLA

contains carry-generate terms ($G_i = A_i.B_i$) and carry-propagate terms ($P_i = A_i+B_i$). From full adder, $CO_{i+1} = A_i.B_i + CI_i.(A_i+B_i)$. The carry bit contains one carry-generate term and one carry-propagate term. When the expression $A_i.B_i$ is 1, the carry-out bit becomes 1 independent of the carry-in bit, CI_i and so the expression $A_i.B_i$ is called the carry-generate term. It generates the carry-out bit [9].

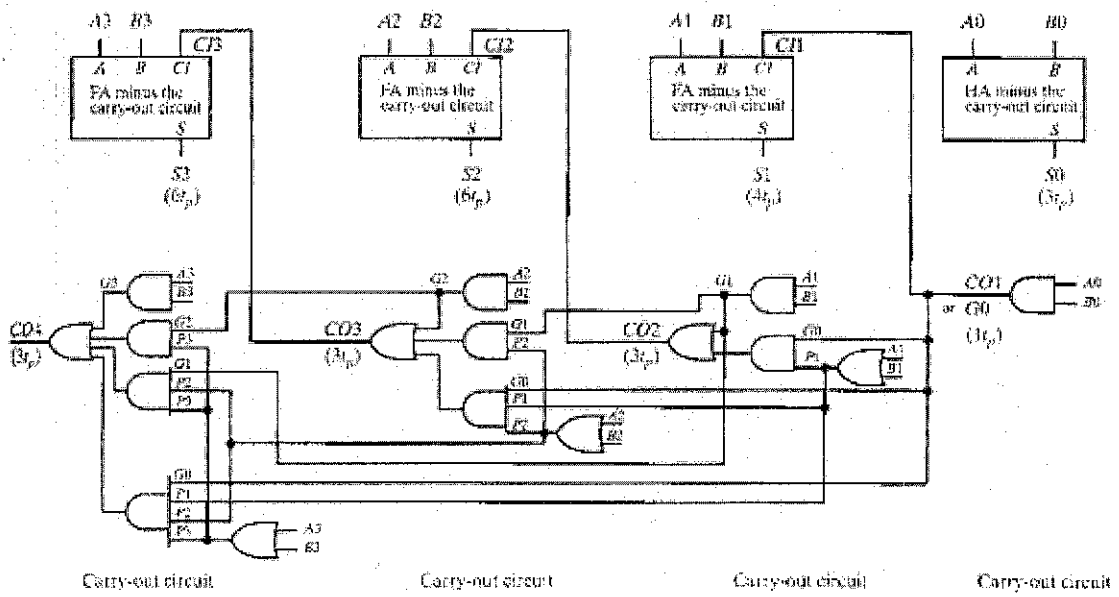


Figure 4 A 4-bit CLA showing carry-out circuitry [9]

When the carry-in bit CI_i is 1 and the expression A_i+B_i is also 1, the carry-out bit becomes 1 and so the expression A_i+B_i is called the carry-propagate term. It propagates or moves the value CI_i to the carry-out bit [9]. The carry-out bit of the non-ripple expandable CLA can be written as follows for each bit position:

Bit position 0:
CO1 = G0 + CI0.P0

Bit position 1:
CO2 = G1 + CI1.P1
= G1 + CO1.P1 $\qquad CI_i = CO_i$
= G1 + G0.P1 + CI0.P0.P1

Bit position 2:
CO3 = G2 + CI2.P2
= G2 + CO2.P2
= G2 + G1.P2 + G0.P1.P2 + CI0.P0.P1.P2

In general, CLA bit position organization scheme for i = 0,1,2...:

$$CO_{i+1} = G_i + G_{i-1}.P_i + G_{i-2}.P_{i-1}.P_i + G_{i-3}.P_{i-2}.P_{i-1}.P_i + \ldots + CI0.P0.P1\ldots P_{i-1}.P_i$$

---------- Equation 3

Since each carry-out bit is in SOP (sum of product) form, each function can be implemented as a 2-level gate circuit that is dependent only on the carry-generate and carry-propagate terms for the current bit position and all the previous (or less significant) bit positions. Since each carry-generate carry-propagate term required only a single gate level of logic, each carry-out function past bit position 0 can be implemented as a 3-level gate circuit with settling time (propagation delay time) of just $3t_p$. This reduces the settling time for the sum bits to only $6t_p$ for any CLA with three or more bits [9].

Three things limit the usefulness of CLA circuitry when it is applied over a large number of stages:

i) The carry-generation term G0 from first bit position must be capable of driving each of the succeeding stages.

ii) Each succeeding stage requires gates with an increasing number of inputs (gates with a higher fan-in).

iii) Gate count increases and thus, cost increases with each additional stage.

Due to these limiting factors, CLA is usually implemented over small groups of bits (such as 4 bits). The carry-look-ahead technique can then be applied again over the groups as they are cascaded [9].

2.3.2 Carry-Save Adder (CSA)

Carry-save adders are designed to add more than two operands. This technique involves cascading full adders such that the carry output of each adder is shifted to the left one bit position and added to an FA in the next row (referred to as carry save) except for the last row. A single RCA (ripple-carry adder) or CLA may be used in the last row. The concept is illustrated below for the addition of five 1-bit operands A0, B0, C0, D0 and E0. The following relationship is used to determine the number of rows of adders required [9].

Number of rows of adders = Number of operands to be added – 1

				A0	Operand 1
				B0	Operand 2
			+	C0	Operand 3
				S10	Sum, Row 1
		CO11			Carry, Row 2 (carry save)
	+			D0	Operand 4
		S21	S20		Sum, Row 2
		CO21			Carry, Row 3 (carry save)
	+			E0	Operand 5
		S31	S30		Sum, Row 3
	CO32	CO31			Carry, Row 4 (carry save)
+	CO43	CO42	CO41		Carry, Row 4 (no carry save)
	S43	S42	S41	S40	Sum, Row 4 (last row)

Extending the concept for more bits, a general block diagram layout for a CSA using FA can be drawn. The diagram layout is illustrated in Figure 5. This type of circuit configuration is also referred to as a Wallace-Tree Summing Network. HA can be used in places where only two bits must be added and the least significant bit is not required for the adder in the last row [9].

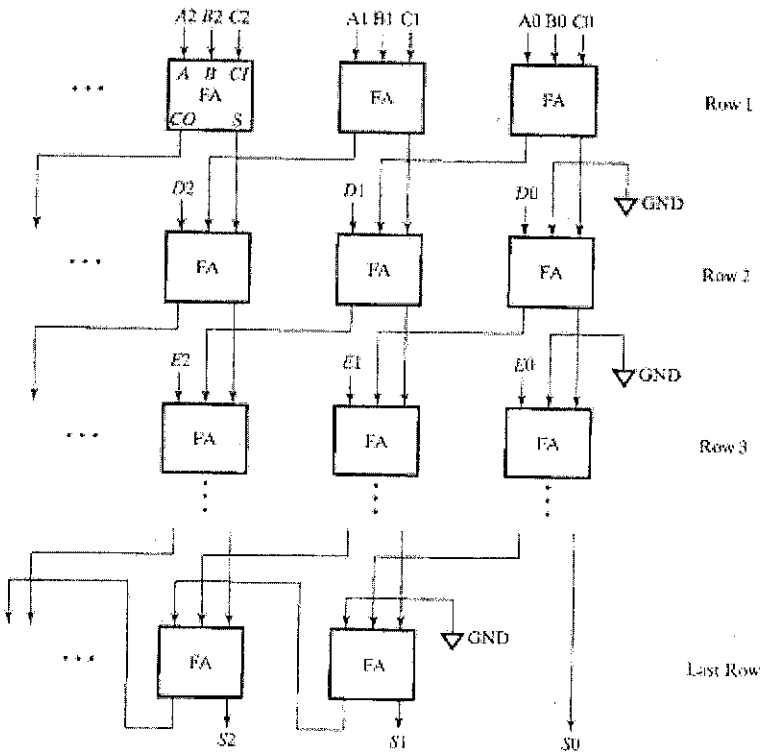


Figure 5 General block diagram layout for a CSA using full adders [9]

2.4 MULTIPLIERS

Multiplication of signed numbers represented by two's complement is not as straightforward as multiplication of unsigned numbers. Multiplication of signed numbers employs an algorithm, either right-shift or left-shift algorithm. In this section, right-shift algorithm will be discussed as this involves less hardware realization. Multiplication with right shifts uses top-to-bottom accumulation as governed by the following equation:

$$p^{(j+1)} = (p^{(j)} + x_j a 2^k)\, 2^{-1} \qquad \text{with } p^{(0)} = 0 \text{ and}$$

$$\text{|——add——|} \qquad\qquad p^{(k)} = p = ax + p^{(0)}2^{-k}$$

$$\text{|—shift right—|}$$

The example in Figure 6 shows a sequential multiplication of two's complement numbers with right shifts. The multiplicand is -10 and multiplier is 11, which yields result -110. For two's complement, arithmetic shift right (ASR) is used to preserve the MSB in which the contents are shifted right by one bit. For example, 1101 becomes 1110 and 0101 becomes 0010. The carry-out is discarded for the addition of previous and current partial products.

Previous partial product

Current partial product

Left-most bit 1 is **NOT** carry-out bit, it is the sign bit produced during ASR

a		1	0	1	1	0				
x		0	1	0	1	1				
$p^{(0)}$		0	0	0	0	0				
$+x_0a$		1	0	1	1	0				
$2p^{(1)}$	1	1	0	1	1	0				
$p^{(1)}$		1	1	0	1	1	0			
$+x_1a$		1	0	1	1	0				
$2p^{(2)}$	1	1	0	0	0	1	0			
$p^{(2)}$		1	1	0	0	0	1	0		
$+x_2a$		0	0	0	0	0				
$2p^{(3)}$	1	1	1	0	0	0	1	0		
$p^{(3)}$		1	1	1	0	0	0	1	0	
$+x_3a$		1	0	1	1	0				
$2p^{(4)}$	1	1	0	0	1	0	0	1	0	
$p^{(4)}$		1	1	0	0	1	0	0	1	0
$+x_4a$		0	0	0	0	0				
$2p^{(5)}$	1	1	1	0	0	1	0	0	1	0
$p^{(5)}$		1	1	1	0	0	1	0	0	1 0

Figure 6 Sequential multiplication of 2's-complement numbers with right shifts [10]

2.4.1 Radix-4 Booth's Multiplier (Booth's Algorithm)

Booth's Algorithm is used to replace strings of 1's in multiplier by +1 and -1. This is the most basic form of Booth Algorithm called radix-2 Booth recoding. There are two ways to speed up the multiplication process:

i) Reducing the number of operands to be added by handling more than one multiplier bit at a time.

ii) Adding the operands faster via parallel/pipelined multi-operand addition using tree and array multipliers.

Radix-4 Booth's recoding is a variation of modified Booth's Algorithm. Table 3 shows the recoding techniques associated with radix-4 Booth's Algorithm. Multiplier bit position is denoted x_i and the recoded version for multiplier is $z_{i/2}$. An example to recode the multiplier is provided below the table. From the example, it can be seen that a 16-bit multiplier is recoded to an 8-bit operand, thus reducing the number of partial products to be added.

Table 3 Radix-4 Booth's recoding [10]

x_{i+1}	x_i	x_{i-1}	y_{i+1}	y_i	$z_{i/2}$	Explanation
0	0	0	0	0	0	No string of 1s in sight
0	0	1	0	1	1	End of string of 1s
0	1	0	0	1	1	Isolated 1
0	1	1	1	0	2	End of string of 1s
1	0	0	-1	0	-2	Beginning of string of 1s
1	0	1	-1	1	-1	End a string, begin new one
1	1	0	0	-1	-1	Beginning of string of 1s
1	1	1	0	0	0	Continuation of string of 1s

Example: $(21\ 31\ 22\ 32)_{four}$

	1	0	0	1	1	1	0	1	1	0	1	0	1	1	1	0	Operand x
(1)	-2		2		-1		2		-1		-1		0		-2		Recoded version z

-a = 1010
-2a = 10100

Shifted 2 bits to the right and sign extended

```
==========================
a              0 1 1 0
x              1 0 1 0
z                -1  -2      Recoded version of x
==========================
p(0)       0 0 0 0 0 0
+z0a       1 1 0 1 0 0
--------------------------
4p(1)      1 1 0 1 0 0
p(1)       1 1 1 1 0 1   0 0
+z1a       1 1 1 0 1 0
--------------------------
4p(2)      1 1 0 1 1 1   0 0
p(2)           1 1 0 1   1 1 0 0
==========================
```

Figure 7 Radix-4 multiplication with modified Booth's recoding [10]

The example in Figure 7 illustrates radix-4 multiplication with modified Booth's recoding of the two's complement multiplier. The multiplicand is 6 whereas the multiplier is -6, which gives -36 as the result. Since the multiplier is 4-bit long, only two additions of partial products are required with radix-4 multiplication. The redundant sign bits in front of the final result can be discarded. Note that right-shift algorithm is used.

An advantage of using modified Booth's recoding technique is that the number of partial products is reduced which in turn reduces the hardware and delay required to sum the partial products. This is because when there is a string of 0 or a string of 1 in the multiplier, only shifting operation is performed, which is faster than addition. Hence, it is often wise to choose one of the two's complement numbers that has fewer changes in 0's or 1's as the multiplier. For instance, consider the two's complement numbers 101001 and 111001 in Table 4. A disadvantage of Booth's Algorithm is that it adds delay into the formation of partial products.

Table 4 Selection of multiplier based on fewer transitions in 0's or 1's

101001	4 changes. From 1 to 0, from 0 back to 1, then back to 0, from 0 to 1 for the last bit.
111001	2 changes. The 1 in bit-3 changes to 0, then 0 in bit-1 changes to 1. Selected as multiplier.

The hardware implementation of radix-4 multiplier requires registers for multiplicand, multiplier and partial product, recoding logic, multiplexer and adder. The simplified block diagram for a radix-4 multiplier based on Booth's recoding is represented in Figure 8.

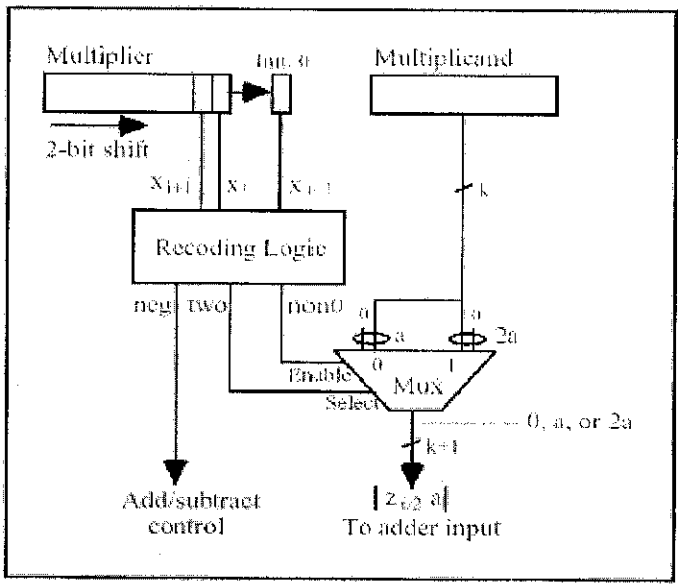


Figure 8 Hardware realization of radix-4 multiplier based on Booth's recoding [10]

Figure 9 shows the recoding logic and multiplexer to generate a partial product. The multiplier group consists of 3 bits of the multiplier (x_{i+1} x_i x_{i-1}). Output of the booth decoder will select 0, M or 2M where M is the multiplicand. The XOR gates are used to generate one's complement by inverting all the bits. If the MSB of the multiplier group is 0, then the partial product will be 0, M or 2M; if the MSB of the multiplier group is 1, then all the bits of the partial product will be inverted. –M or -2M can be generated by adding S=1 in which two's complement of partial product is created. The resulted partial product is then added to the previous partial product stored in a register that are shifted two bits to the right. Normally, CSA will be used so that multiple operands can be added simultaneously.

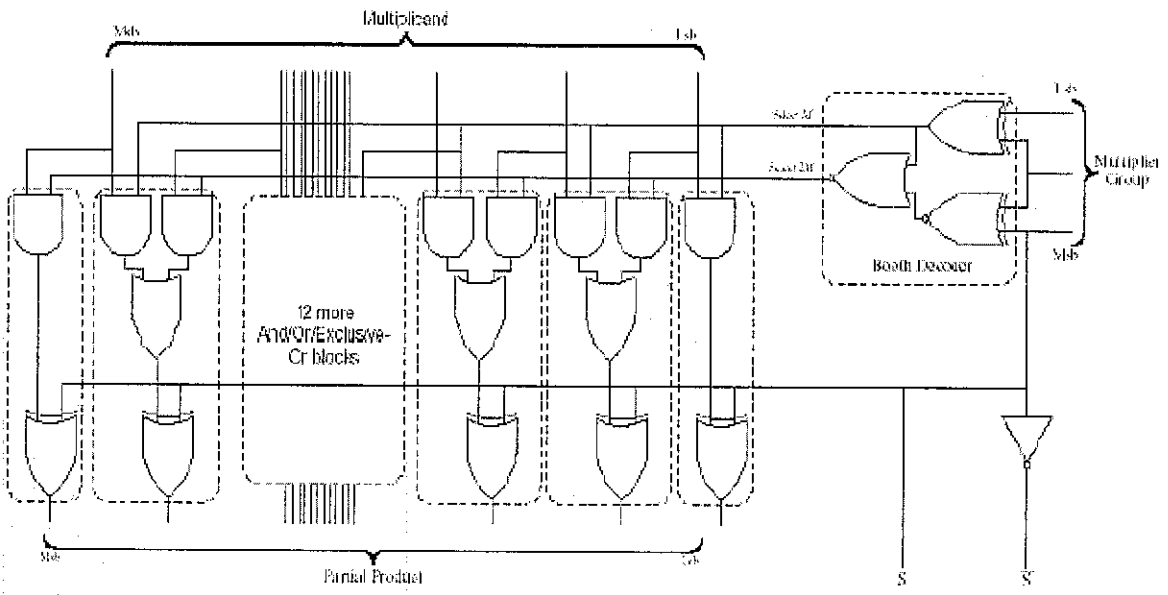


Figure 9 Recoding logic and multiplexer to generate partial products [10]

2.4.2 Baugh-Wooley Array Multiplier

Baugh-Wooley array multiplier is used to multiply positive and negative numbers in two's complement. The principle of this multiplier is that the subtraction can be added by complementing the subtrahend and adding 1. This multiplier has a regular structure and is governed by a final equation derived as follows [11]:

Let us consider two numbers A and B:

$$A = (a_{n-1} \ldots a_0) = -a_{n-1}.2^{n-1} + \sum_{0}^{n-2} a_i.2^i$$

$$B = (b_{n-1} \ldots b_0) = -b_{n-1}.2^{n-1} + \sum_{0}^{n-2} b_i.2^i$$

The product of A and B is given by the following equation:

$$A.B = a_{n-1}.b_{n-1}.2^{2n-2} + \sum_{0}^{n-2}\sum_{0}^{n-2} a_i.b_i.2^{i+j} - a_{n-1}\sum_{0}^{n-2} b_i.2^{n+i-1} - b_{n-1}\sum_{0}^{n-2} a_i.2^{n+i-1}$$

In order to use only adder cells, the negative terms are rewritten as:

$$-a_{n-1}\sum_{0}^{n-2} b_i.2^{i+n-1} = a_{n-1}.\left(-2^{2n-2} + 2^{n-1} + \sum_{0}^{n-2} \overline{b_i}.2^{i+n-1}\right)$$

Hence, the product of A and B becomes:

$$A.B = a_{n-1}.b_{n-1}.2^{2n-2} + \sum_{0}^{n-2}\sum_{0}^{n-2} a_i.b_j.2^{i+j}$$

$$+ b_{n-1}\left[-2^{2n-2} + 2^{n-1} + \sum_{0}^{n-2} \overline{a_i}.2^{i+n-1}\right]$$

$$+ a_{n-1}\left[-2^{2n-2} + 2^{n-1} + \sum_{0}^{n-2} \overline{b_i}.2^{i+n-1}\right]$$

The final equation is:

$$A.B = -2^{2n-1} + \left(\overline{a_{n-1}} + \overline{b_{n-1}} + a_{n-1}.b_{n-1}\right).2^{2n-2}$$

$$+\sum_{0}^{n-2}\sum_{0}^{n-2} a_i.b_j.2^{i+j} + \left(a_{n-1} + b_{n-1}\right).2^{n-1}$$

$$+\sum_{0}^{n-2} b_{n-1}.\overline{a_i}.2^{i+n-1} + \sum_{0}^{n-2} a_{n-1}.\overline{b_i}.2^{i+n-1}$$

---------- Equation 4

since

$$-\left(b_{n-1} + a_{n-1}\right).2^{2n-2} = -2^{2n-1} + \left(\overline{a_{n-1}} + \overline{b_{n-1}}\right).2^{2n-2}$$

A and B are n-bit operands, so their product is a 2n-bit number. Consequently, the most significant weight is 2n-1, and the first term -2^{2n-1} is taken into account by adding a 1 in the most significant cell of the multiplier. Figure 10 shows the structure of a 5-bit Baugh-Wooley multiplier and can be verified using the final equation by substituting n=5. The array comprises of (n-1)*(n-1)+1 full adders, multiplication units (AND gates) and carry propagation adders.

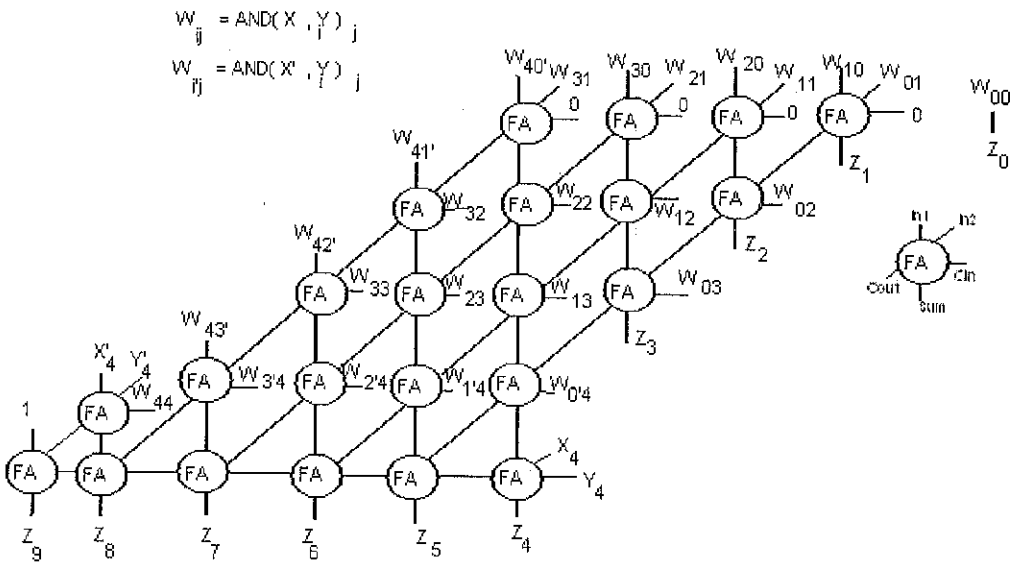


Figure 10 A 5-bit Baugh-Wooley multiplier

CHAPTER 3

METHODOLOGY/PROJECT WORK

3.1 PROJECT FLOW

The flow of the entire project is outlined as seen in Figure 12. The design methodology for an FIR filter starts from filter specifications, coefficients calculations, filter structure, study of finite wordlength effects and finally filter implementation. Specifications of the filter are determined based on the type of filter designed. There are four types of filters, namely low-pass, high-pass, bandpass and bandstop filter. Several methods are available to obtain filter coefficients and the most commonly used are window, optimal and frequency sampling methods. Two most basic FIR filter architectures are direct form (DF) and transpose direct form (TDF), given in Figure 11. In this project, a low-pass FIR filter with DF architecture is designed using Kaiser Window method.

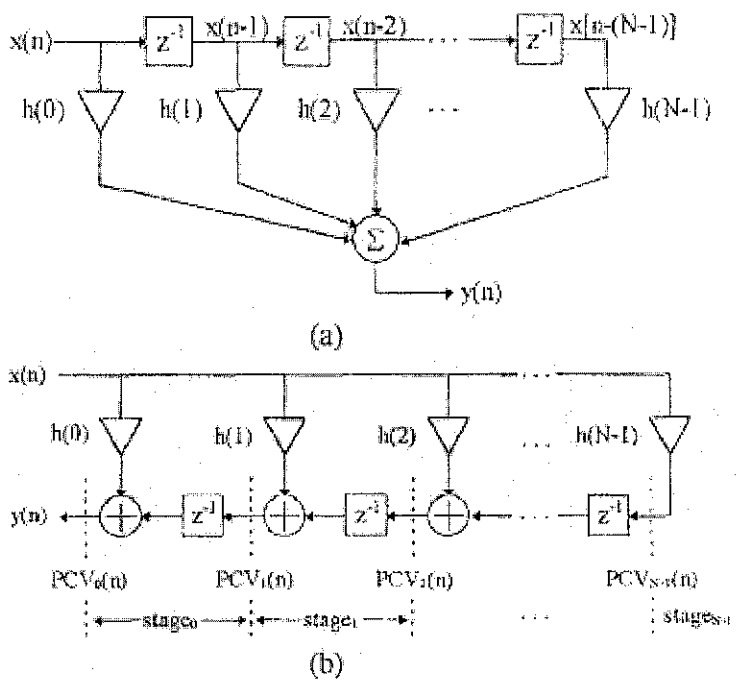


Figure 11 (a) DF FIR filter architecture (b) TDF FIR filter architecture [12]

The different structures of adders and multipliers are explored and some of the structures have already been discussed in the previous chapter. Design description, which is to describe the circuit in terms of its behaviour, can be done in a few levels of abstractions. The lowest level is circuit level with switches as the basic element, followed by gate level, data flow level and lastly, the highest level, which is behavioural level. In common practice, both gate level and data flow level modeling (RTL level) are used because many of the behavioural level constructs are not directly synthesizable. Even if synthesizable, they are likely to yield relatively redundant or wrong hardware. The number of bits used to represent input data and filter coefficients, and also the number representation scheme are important considerations that can affect the filter performance.

A basic FIR filter consists of multipliers, adders and delay units, as can be deduced from Equation 1 (page 7). Depending on the architecture and performance objectives, a filter can also have memory and control unit. Each of the filter components is coded into Verilog and its functionality is verified. Performance comparison is done for different structures of adders and multipliers in view of their propagation delay, area and power consumption. Two different structures of adders and multipliers are compared in this project. The better component structure based on performance is chosen to be integrated into the complete filter design. Functionality of the complete filter is verified through simulation and its performance is tabulated. Once successful, the design is downloaded into FPGA and functionality verification is carried out by analyzing the filter output using a logic analyzer.

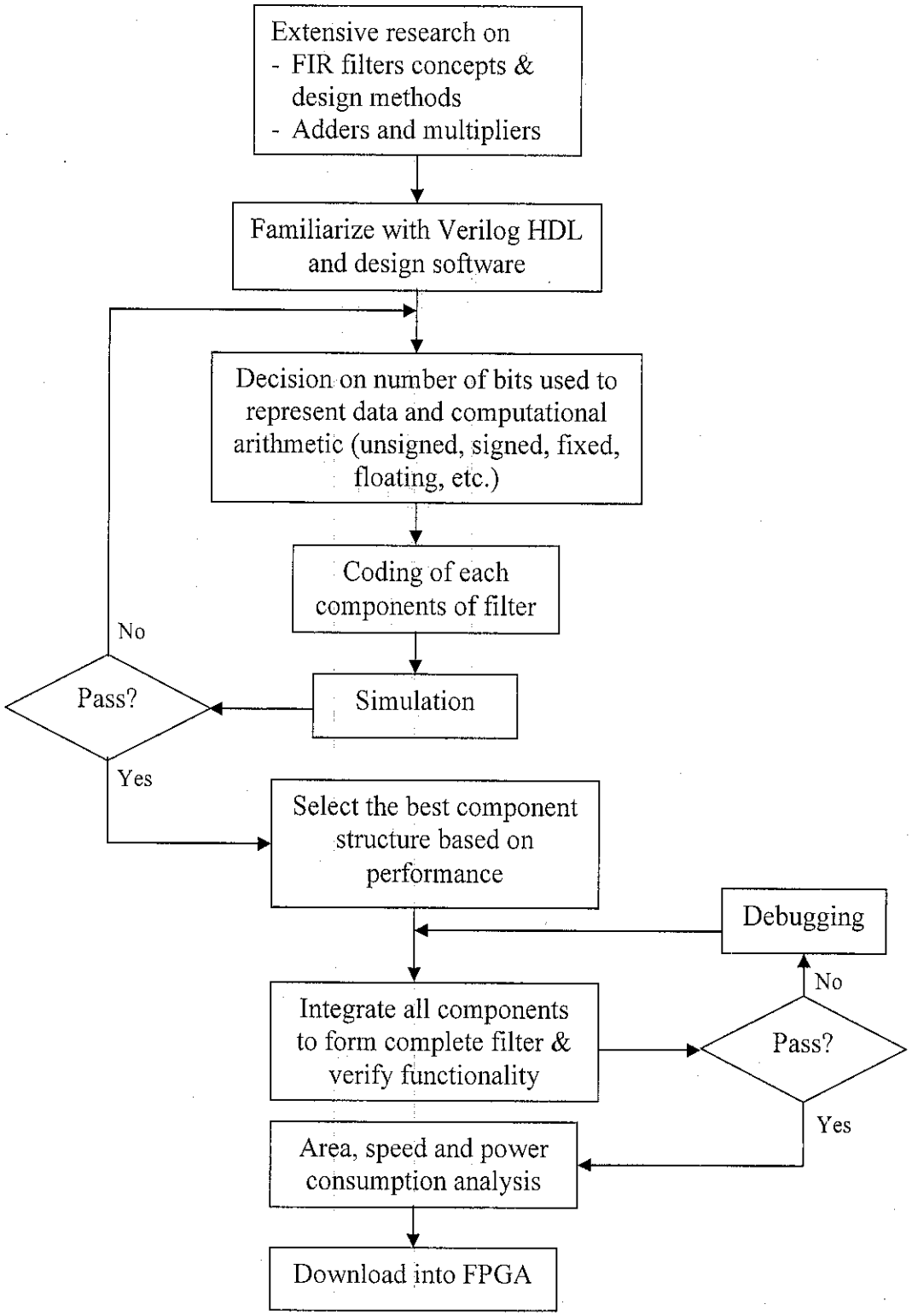


Figure 12 Entire project flow

3.2 BASIC DESIGN METHODOLOGY

Figure 13 indicates the crucial steps in designing small modules of a filter. Each of the small modules is simulated and verified separately to ease debugging task.

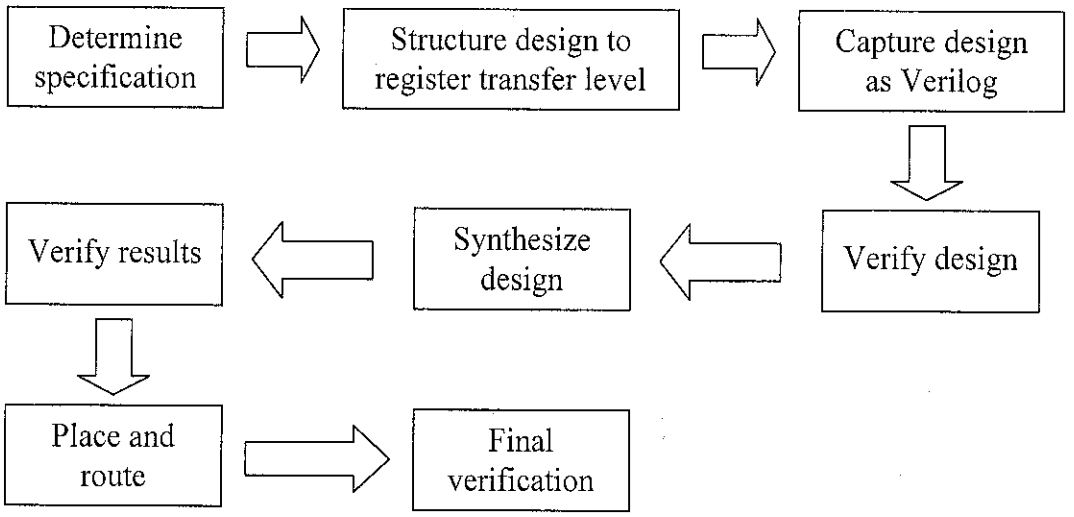


Figure 13 Steps in designing small modules of a filter [13]

1. *Determine specification.* The specification details the behavior and interface of each module in the design. At the module level, the specification includes the following:
   i) A description of the top-level behavior of the module
   ii) A description of all inputs and outputs, their timing and constraints
   iii) Performance requirements and constraints
2. *Structure design to register transfer level (RTL).* This is a logic design phase where a block diagram for the design is determined, which includes registers and functions of combinational logic.
3. *Capture design as Verilog.* Design description can be done based on a few levels of abstraction – the highest is behavioral level, followed by data flow level, gate level and the lowest switch (circuit) level. Many of the behavioral level constructs are not directly synthesizable; even if synthesized they are likely to yield relatively redundant or wrong hardware. The solution is to redo the behavioral modules at lower levels.

4. *Verify design.* This is a pre-synthesis verification process to determine that the design is 100% functionally correct. This process is known as functional simulation.
5. *Synthesize design.* Synthesis tools are used to transform the Verilog design into a gate level design.
6. *Verify results of synthesis.* Gate-level simulation, timing analysis and other techniques are used to verify that the design produced by the synthesis tool is correct and consistent with the Verilog RTL design.
7. *Place and route.* This stage is referred to as physical design where the actual layout of the chip is determined. The gates in the chip are assigned (placement) to positions on the chip and then connected together with wires (routing). Post-place-and-route simulation can then be performed to obtain area and timing information.
8. *Final verification.* A number of final checks are done to ensure that the chip is wired up correctly and is manufacturable. The nature of these checks is beyond the scope of this project.

3.3 BIT REPRESENTATION SCHEME

In this project, the number of bits used to represent input data and filter coefficients is eight bits. Signed numbers will be used with two's complement as the representation scheme. Fixed-point numbers will be employed instead of floating-point which needs a more complex number representation scheme. Area, speed and power consumption analyses are performed by using ModelSim and Xilinx ISE simulation tools.

3.4 IDENTIFICATION OF TOOLS

1. ModelSim and Xilinx ISE simulation tools
2. MATLAB
3. Virtex-II xc2v1000 reference board – an FPGA which enables the filter design to be programmed into.
4. Agilent Technologies 1673G logic analyzer and probes
5. Xilinx JTAG cable

3.5 TASKS ACCOMPLISHED

1. Two structures of adders and multipliers are designed, simulated and their performances are compared. The adders are CLA and CSA while multipliers are radix-4 Booth's multiplier and Baugh-Wooley array multiplier. CLA and Baugh-Wooley multiplier are found to have better performance compared to their counterparts.
2. A DF low-pass, 18th order FIR filter is designed by using adders, multipliers and delay units. The filter is implemented using parallel approach, which eliminates the need of memory and control unit.
3. The performance of the complete filter is analyzed. Its functionality is verified through software simulation, as well as hardware verification.
4. The degree to which the filter can reduce or eliminate high-frequency noise is analyzed using MATLAB.

3.6 PROBLEMS ENCOUNTERED

Throughout this project, some problems and challenges are encountered as discussed briefly below:

1. *Inexperience in employing the different levels of abstractions of Verilog coding.* As mentioned, some codes written in behavioural level may be non-synthesizable. Considerable amount of time is used to debug the faulty codes when simulation fails or gives incorrect output.
2. *Limitation of Virtex-II device.* This device has 172 bonded IOBs. However, both adders accept outputs from 19 multipliers simultaneously, which gives a total of 304 bits for all outputs of multipliers. Limitation of the target device causes simulations to fail for both CLA and CSA. The solution is discussed in the next chapter.
3. *Incapability of 1673G logic analyzer to provide inputs.* The available logic analyzer in the lab is not able to provide inputs to the filter that is downloaded into the Virtex-II chip. Hence, inputs to the filter are provided manually by extending the codes to account for a signal generator module.
4. *Difficulty in predicting the output from the filter.* It can be seen from the codes that filter operation is controlled by the triggering of clock. During hardware testing of

the filter functionality, the onboard clock is utilized and is always running once the board is powered-up. Therefore, it is very hard to compare the output from simulations and output obtained from logic analyzer. A manual push button (available on the board) is used to serve the function of a clock trigger.

3.7 TESTING & TROUBLESHOOTING

A lot of debugging is done on the codes when simulation fails or gives incorrect output. This is often so when behavioural level modeling is used to model the filter components. Behavioural level modeling is inevitable when conditional expressions are employed in the process of designing. Examples of these type of constructs are 'if', 'if-else', 'while' and 'for'. In this case, experience is vital to recognize the way of writing that results in codes that are synthesizable.

All the filter components are simulated and verified to ensure that their intended functionalities are correct before proceeding to the next step in designing. The complete filter does not require much troubleshooting since all lower level modules are functioning correctly. The simulated design is verified through hardware synthesis using FPGA so as to be sure that the filter is working correctly in practical.

CHAPTER 4

RESULTS & DISCUSSION

4.1 FIR FILTER SPECIFICATIONS

A low-pass FIR filter is designed using Kaiser Window with MATLAB 'sptool'. A set of filter specifications is defined in Table 5.

Table 5 Filter specifications

Specifications	Values
Passband frequency, F_p	1000 Hz
Stopband frequency, F_s	2000 Hz
Passband ripple, R_p	0.4455 dB (5%)
Stopband ripple, R_s	40 dB (1%)
Sampling frequency, F_{samp}	8000 Hz

This set of specifications yields an 18th order filter with 19 coefficients altogether. The specifications are chosen such that the number of coefficients is not too big in order to reduce the filter size. The multiplication and addition process carried out by the filter is intended to be parallel so that the throughput and sample rate of the filter can be maximized. Due to the parallelism, the number of coefficients has to be small in order to reduce hardware. FIR filters can also be implemented in sequential in which this approach aims to minimize area requirements through the reuse of as much hardware as possible. However, its bottleneck is low throughput. Direct form (DF) FIR filter is realized in this project.

4.1.1 Analysis of Designed FIR Filter

The defined filter specifications are analyzed to determine the level of filter performance in removing or reducing high-frequency noise. It can be seen in Figure 14 that the generated signal has frequency of 500Hz and random noise has frequencies ranging from 500Hz to 8000Hz. The two signals are combined to create a noisy signal, z, which is then allowed to pass through to the designed filter that ultimately gives filtered

output y. The second plot in Figure 16 resembles the original signal in which the filtered signal is relatively smooth without jagged edges caused by high-frequency noise. Since the cutoff frequency of designed filter is 1500 Hz, any frequencies above this will be significantly suppressed. These suppressed frequencies have negligible amplitudes owing to the 40 dB stopband ripple. However, the filtered output displays a phase lag or termed group delay of nine. The group delay of a filter is a measure of the average delay of the filter as a function of frequency. It is the negative first derivative of the phase response of the filter.

```
%freq of signal=500Hz with sampling freq=8000Hz
f=8000;
t=0:1/f:1;
x=sin(2*pi*500*t);
%to create noise with 16 different frequencies
for k=1:16
    nn(k,:)=0.08*randn(1)*sin(2*pi*k*500*t);
end
sum=0;
for k=1:16
    sum=sum+nn(k,:);
end
z=x+sum;
%filt1 consists of designed filter specs
y=filter(filt1.tf.num,1,z);
m=1:100;

figure(1);
subplot(2,1,1); plot(x(m));
xlabel('Time index n'); ylabel('Amplitude');
title('Signal, x = sin (500\pit)');
subplot(2,1,2); plot(sum(m));
xlabel('Time index n'); ylabel('Amplitude');
title('Random noise, sum');
figure(2);
subplot(2,1,1); plot(z(m));
xlabel('Time index n'); ylabel('Amplitude');
title('Noisy signal, x + sum');
subplot(2,1,2); plot(y(m));
xlabel('Time index n'); ylabel('Amplitude');
title('Filtered signal, y');
```

Figure 14 Codes to test the filter performance

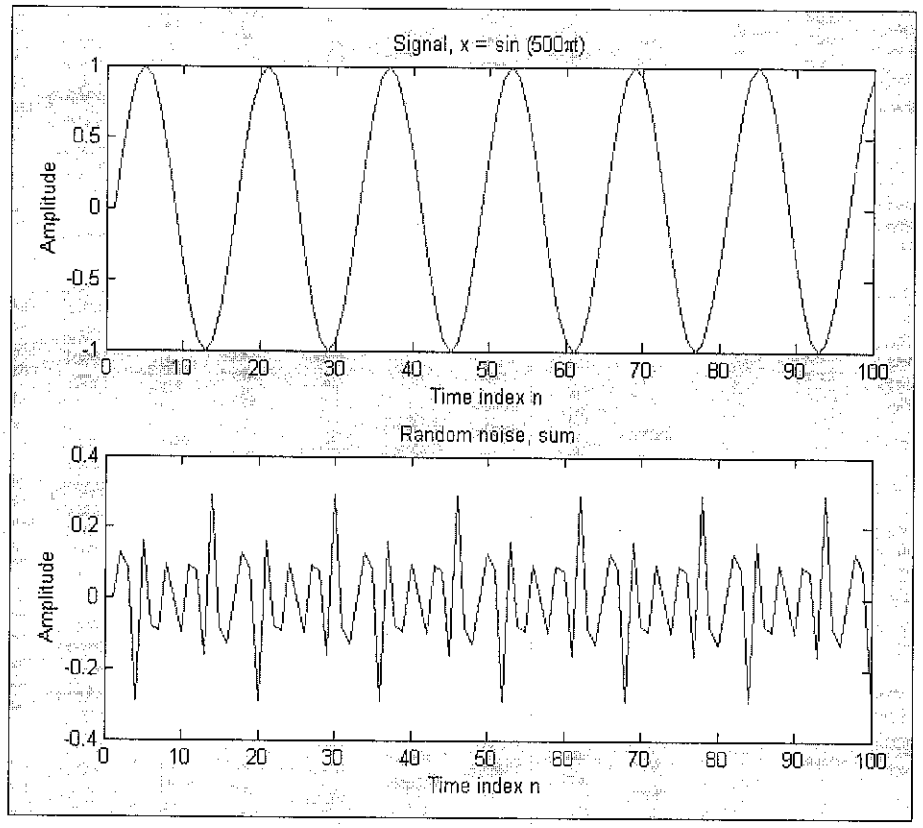


Figure 15 Original signal and generated random noise

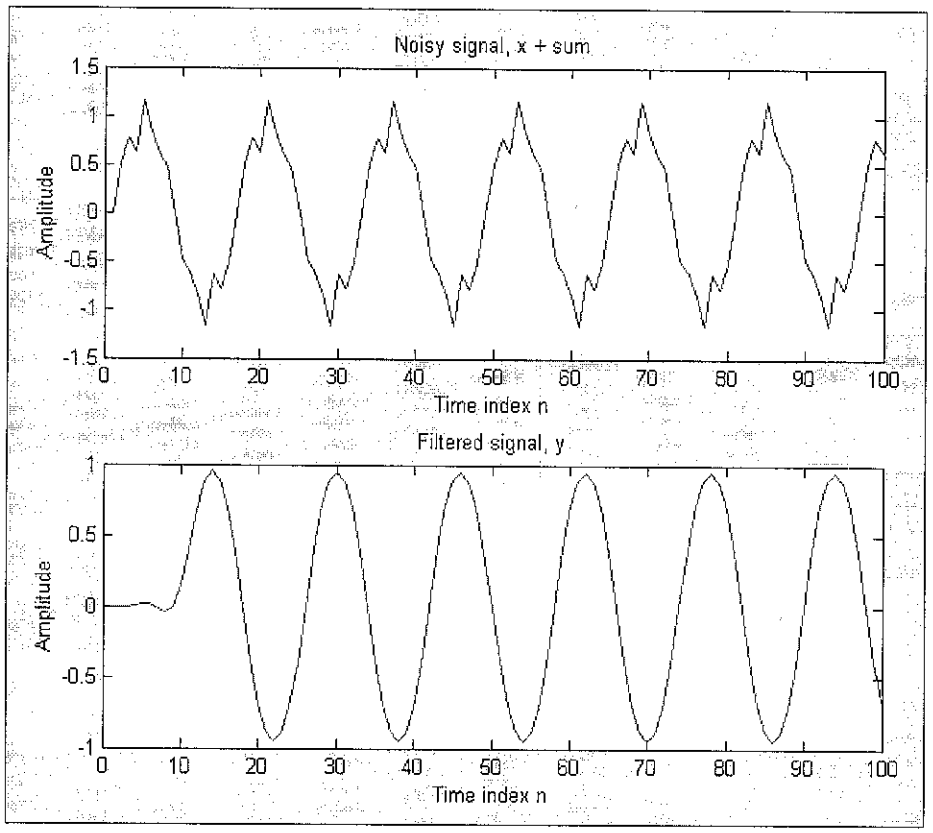


Figure 16 Noisy signal and filtered signal

4.2 VERILOG CODES

This section indicates the associated codes that are used in the filter design. These include codes for Baugh-Wooley array multiplier, CLA, shift register and the complete filter. Note that other Verilog codes associated with radix-4 Booth's multiplier and carry-save adder are included in Appendix A.

4.2.1 Baugh-Wooley Array Multiplier

Variable B (codes in Figure 17) represents the coefficient of the filter and is declared as parameter so that its value can be changed in the complete filter design during instantiation of this module. The following codes illustrate an example which declares B as having the hexadecimal value 02. The test-bench for Baugh-Wooley array multiplier instantiates the module 'Wooley' that declares B as an input port rather than parameter in order to be used for simulation purpose. The complete codes for this multiplier are shown in Figure 34 in Appendix A.

```
`timescale 1ns/1ps
module Wooley(A,P);

input [7:0]A;
output [15:0]P;
parameter [7:0]B = 8'h02;

wire [49:0]U;
wire [6:0]W1,W2;
wire W3,CO;
wire sum0,sum1,sum2,sum3,sum4,sum5,sum6,sum7,sum8,sum9,sum10;
wire sum11,sum12,sum13,sum14,sum15,sum16,sum17,sum18,sum19,sum20;
wire sum21,sum22,sum23,sum24,sum25,sum26,sum27,sum28,sum29,sum30;
wire sum31,sum32,sum33,sum34,sum35,sum36,sum37,sum38,sum39,sum40;
wire sum41,sum42,sum43;
wire cout0,cout1,cout2,cout3,cout4,cout5,cout6,cout7,cout8,cout9,cout10;
wire cout11,cout12,cout13,cout14,cout15,cout16,cout17,cout18,cout19,cout20;
wire cout21,cout22,cout23,cout24,cout25,cout26,cout27,cout28,cout29,cout30;
wire cout31,cout32,cout33,cout34,cout35,cout36,cout37,cout38,cout39,cout40;
wire cout41,cout42,cout43,cout44,cout45,cout46,cout47,cout48,cout49,cout50;
wire cout51,cout52,cout53,cout54,cout55,cout56,cout57;

assign U[0] = A[0] & B[0];
assign U[1] = A[1] & B[0];
assign U[2] = A[2] & B[0];
assign U[3] = A[3] & B[0];
assign U[4] = A[4] & B[0];
assign U[5] = A[5] & B[0];
assign U[6] = A[6] & B[0];
```

Figure 17 Partial codes of Baugh-Wooley multiplier

```
`timescale 1ns/1ps
module Wooley_tst;

reg   [7:0]A,B;
wire  [15:0]P;

Wooley woo(A,B,P);
initial
begin
      A = 8'h00; B = 8'h00;
      #100 A = 8'h01; B = 8'h10;
      #50 A = 8'h11; B = 8'h1a;
      #50 A = 8'h21; B = 8'h2b;
      #50 A = 8'h31; B = 8'h32;
      #50 A = 8'h82; B = 8'h10;
      #50 A = 8'haf; B = 8'h7a;
      #50 A = 8'hc5; B = 8'hbb;
      #50 A = 8'hff; B = 8'hff;
end
initial $monitor($realtime ," A=%h, B=%h, product=%h", A,B,P);

endmodule
```

Figure 18 Test-bench for Baugh-Wooley array multiplier

```
module full_adder(cin,b,a,sum,cout);

input  cin,b,a;
output sum,cout;

wire  S01;
wire  C01;
wire  C02;

half_adder  ha1(a,b,S01,C01);
half_adder  ha2(cin,S01,sum,C02);
assign      cout = C01 | C02;

endmodule
```

Figure 19 Full adder

```
module half_adder(A,B,sum,cout);

input  A,B;
output sum,cout;

assign cout = A & B;
assign sum = A ^ B;

endmodule
```

Figure 20 Half adder

4.2.2 Carry-Look-Ahead Adder (CLA)

Figures 21 and 22 represent 16-bit CLA and 17-bit CLA respectively. As the names imply, a 16-bit CLA is capable of adding two operands that have 16 bits. Note that the Verilog codes for CLA_nsx (4-bit CLA without sign extension), CLA (4-bit CLA with sign extension), CLA_18 (18-bit CLA), CLA_19 and CLA_20 are attached to Appendix A.

```
// 16-bit CLA
module CLA_16(A,B,S);

input  [15:0]A;
input  [15:0]B;
output [16:0]S;

wire CI0 = 0;
wire C01,C02,C03;

CLA_nsx cla1(A[3:0],B[3:0],CI0,S[3:0],C01);
CLA_nsx cla2(A[7:4],B[7:4],C01,S[7:4],C02);
CLA_nsx cla3(A[11:8],B[11:8],C02,S[11:8],C03);
CLA cla4(A[15:12],B[15:12],C03,S[15:12],S[16]);

endmodule
```

Figure 21 16-bit CLA

```
// 17-bit CLA
module CLA_17(A,B,S);

input  [16:0]A;
input  [16:0]B;
output [17:0]S;

wire   C01,C02,C03,C04;
wire   A17,A18,A19,B17,B18,B19,S18,S19,S20;
wire   CI0 = 0;

CLA_nsx cla1(A[3:0],B[3:0],CI0,S[3:0],C01);
CLA_nsx cla2(A[7:4],B[7:4],C01,S[7:4],C02);
CLA_nsx cla3(A[11:8],B[11:8],C02,S[11:8],C03);
CLA_nsx cla4(A[15:12],B[15:12],C03,S[15:12],C04);

assign A17=A[16],A18=A[16],A19=A[16];
assign B17=B[16],B18=B[16],B19=B[16];

CLA cla1({A19,A18,A17,A[16]},{B19,B18,B17,B[16]},C04,{S19,S18,S[17:16]},S20);

endmodule
```

Figure 22 17-bit CLA

4.2.3 Shift Register (Delay Units)

Figure 23 shows the codes for a shift register which consists of instantiations of eighteen flip-flops. The flip-flops serve as delay units for the filter.

```
`timescale 1ns/1ps
module delay(clk,reset,x,y1,y2,y3,y4,y5,y6,y7,y8,y9,
y10,y11,y12,y13,y14,y15,y16,y17,y18);

input clk,reset;
input [7:0]x;
output [7:0]y1,y2,y3,y4,y5,y6,y7,y8,y9,y10;
output [7:0]y11,y12,y13,y14,y15,y16,y17,y18;

flipflop ff1(clk,reset,x,y1);
flipflop ff2(clk,reset,y1,y2);
flipflop ff3(clk,reset,y2,y3);
flipflop ff4(clk,reset,y3,y4);
flipflop ff5(clk,reset,y4,y5);
flipflop ff6(clk,reset,y5,y6);
flipflop ff7(clk,reset,y6,y7);
flipflop ff8(clk,reset,y7,y8);
flipflop ff9(clk,reset,y8,y9);
flipflop ff10(clk,reset,y9,y10);
flipflop ff11(clk,reset,y10,y11);
flipflop ff12(clk,reset,y11,y12);
flipflop ff13(clk,reset,y12,y13);
flipflop ff14(clk,reset,y13,y14);
flipflop ff15(clk,reset,y14,y15);
flipflop ff16(clk,reset,y15,y16);
flipflop ff17(clk,reset,y16,y17);
flipflop ff18(clk,reset,y17,y18);

endmodule
```

Figure 23 Shift register acts as delay units by flip-flop instantiations

```
`timescale 1ns/1ps
module flipflop(clk,reset,x,y);

input clk,reset;
input [7:0]x;
output [7:0]y;
reg [7:0]y;

always @(posedge clk or posedge reset)
begin
        if(reset)
                y <= 0;
        else
                y <= x;
end

endmodule
```

Figure 24 Verilog codes of a D flip-flop

4.2.4 Filter Implementation

The Verilog description for the complete filter and its associated test-bench can be seen in Figures 25 and 26 respectively. During the instantiations of multipliers, the filter coefficients are changed using the syntax found in Figure 25.

```
`timescale 1ns/1ps
module filter(clock,reset,data_in,out);

input clock,reset;
input [7:0]data_in;
output [20:0]out;
reg [7:0]mem;
reg [7:0]data_out;
wire [7:0]y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,y14,y15,y16,y17,y18;
wire [15:0]P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12,P13,P14,P15,P16,P17,P18,P19;
wire [16:0]Ra,Rb,Rc,Rd,Re,Rf,Rg,Rh,Ri;
wire [17:0]Raa,Rbb,Rcc,Rdd,Ree;
wire [18:0]Rff,Rgg;
wire [19:0]Rhh;
wire m16,m18,m19;

//register 'mem' acts as buffer for data storage for one clock cycle
always @(posedge clock or posedge reset)
begin
        if(reset)
                begin
                        mem <= 8'h00;
                        data_out <= 8'h00;
                end
        else
                begin
                        data_out <= mem;
                        mem <= data_in;
                end
end

delay shift_reg(clock,reset,data_out,y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,
                y12,y13,y14,y15,y16,y17,y18);

//instantiations of nineteen multipliers
Wooley #(8'h00) mult1(data_out,P1);
Wooley #(8'h00) mult2(y1,P2);
Wooley mult3(y2,P3);
Wooley mult4(y3,P4);
Wooley #(8'hfe) mult5(y4,P5);
Wooley #(8'hf8) mult6(y5,P6);
Wooley #(8'hfc) mult7(y6,P7);
```

continue...

```
Wooley #(8'h0d) mult0(y7,P8);
Wooley #(8'h25) mult9(y8,P9);
Wooley #(8'h30) mult10(y9,P10);
Wooley #(8'h25) mult11(y10,P11);
Wooley #(8'h0d) mult12(y11,P12);
Wooley #(8'hfc) mult13(y12,P13);
Wooley #(8'hf8) mult14(y13,P14);
Wooley #(8'hfe) mult15(y14,P15);
Wooley mult16(y15,P16);
Wooley mult17(y16,P17);
Wooley #(8'h00) mult18(y17,P18);
Wooley #(8'h00) mult19(y18,P19);

//instantiations of adders that add operands with varying number of bits
CLA_16 cla16a(P1,P2,Ra);
CLA_16 cla16b(P3,P4,Rb);
CLA_16 cla16c(P5,P6,Rc);
CLA_16 cla16d(P7,P8,Rd);
CLA_16 cla16e(P9,P10,Re);
CLA_16 cla16f(P11,P12,Rf);
CLA_16 cla16g(P13,P14,Rg);
CLA_16 cla16h(P15,P16,Rh);
CLA_16 cla16i(P17,P18,Ri);

assign m16 = P19[15];
CLA_17 cla17a(Ra,Rb,Raa);
CLA_17 cla17b(Rc,Rd,Rbb);
CLA_17 cla17c(Re,Rf,Rcc);
CLA_17 cla17d(Rg,Rh,Rdd);
CLA_17 cla17e(Ri,{m16,P19},Ree);
CLA_18 cla18a(Raa,Rbb,Rff);
CLA_10 cla10b(Rcc,Rdd,Rgg);
CLA_19 cla19a(Rff,Rgg,Rhh);

assign r18 = Ree[17], r19 = Ree[17];
CLA_20 cla20a(Rhh,{r19,r18,Ree},out);

endmodule
```

Figure 25 Verilog description for the complete filter

```
`timescale 1ns/1ps
module filter_tb();

reg clock,reset;
reg [7:0]data_in;
wire [20:0]out;
integer i;

parameter offset = 100;
parameter cycle = 20;

filter filt(.clock(clock),.reset(reset),.data_in(data_in),.out(out));

initial
begin
        clock = 0;    reset = 0;    data_in = 8'h00;
        #offset;
        forever #cycle clock = ~clock;
end

initial
begin
        #(offset+cycle) reset = 1;
        #cycle;
        reset = 0;
        data_in = 8'h01;

        for(i=0; i<20; i=i+1)
                begin
                        #(cycle*2);
                        data_in = data_in + 3'd5;
                end
end

initial $monitor($time," clock=%b, reset=%b, input=%h, output=%h", clock,reset,data_in,out);

endmodule
```

Figure 26 Test-bench for the complete filter

4.3 SOFTWARE SIMULATIONS

Functional and timing simulation results for radix-4 Booth's multiplier and Baugh-Wooley multiplier are included in Appendix B.

Simulations for CLA for performance comparison are done based on the overall adder formed by multiple CLA instantiations. However, the large amount of I/Os of overall adder has exceeded the amount of I/Os that the selected device is capable of handling, which causes simulation to fail. Thus, some of the input ports are declared as 'wire' and assigned values internally. To ensure the accuracy of the simulation results in terms of performance criteria, two sets of the number of input ports are chosen, which are one and eight input ports. It can be seen in Tables 7 and 8 that the percentage difference follows a consistent trend for the three performance criteria. All three criteria – path delay, area and power consumption decrease by half when input port increases from one to eight. The respective Verilog codes are attached to Appendix B, shown in Figures 51 and 53, together with the simulation results for both test-benches.

Similar to CLA, the simulations for CSA for performance comparison are done based on the overall adder formed by multiple CSA instantiations. The CSA also encounters the same problem as in the case of CLA. Similar method as in CLA is used to perform simulations on CSA. The Verilog codes for overall adder with one input and eight input ports are included in Appendix B, shown in Figures 59 and 61, together with the simulation results for both test-benches.

4.3.1 Performance Comparisons

The following results are obtained through functional and timing simulations using Xilinx ISE synthesis tool.

Table 6 Performance comparison between multipliers

	Booth's Multiplier	Baugh-Wooley Multiplier	Percentage difference (Baugh-Wooley as reference)
Maximum path delay after place & route (ns)	24.542	25.078	2.14%
Area (no. of slices out of 5120)	78	64	-21.88%
Power consumption (mW)	510.34	481.65	-5.96%

Table 7 Performance comparison between adders with one input port

One input	Carry-look-ahead adder (CLA)	Carry-save adder (CSA)	Percentage difference (CLA as reference)
Maximum path delay after place & route (ns)	27.200	26.090	4.08%
Area (no. of slices out of 5120)	31	51	-64.52%
Power consumption (mW)	570.49	510.34	10.54%

Table 8 Performance comparison between adders with eight input ports

Eight inputs	Carry-look-ahead adder (CLA)	Carry-save adder (CSA)	Percentage difference (CLA as reference)
Maximum path delay after place & route (ns)	37.115	36.205	2.45%
Area (no. of slices out of 5120)	183	245	-33.88%
Power consumption (mW)	817.12	775.55	5.09%

4.3.2 Complete filter

Both the functional and timing simulation results for the complete filter are displayed in Figures 27 and 28. Only part of the results is shown.

```
  0 clock=0, reset=0, input=00, output=xxxxxx
120 clock=1, reset=1, input=00, output=000000
140 clock=0, reset=0, input=01, output=000000
160 clock=1, reset=0, input=01, output=000000  //at this time,input data is stored in register
180 clock=0, reset=0, input=06, output=000000
200 clock=1, reset=0, input=06, output=000000  //input 01 is available at data_out, y[1]
220 clock=0, reset=0, input=0b, output=000000
240 clock=1, reset=0, input=0b, output=000000  //input 06 is available at data_out, y[2]
260 clock=0, reset=0, input=10, output=000000
280 clock=1, reset=0, input=10, output=000002  //y[3]
300 clock=0, reset=0, input=15, output=000002
320 clock=1, reset=0, input=15, output=00000e  //y[4]
340 clock=0, reset=0, input=1a, output=00000e
360 clock=1, reset=0, input=1a, output=000020  //y[5]
380 clock=0, reset=0, input=1f, output=000020
400 clock=1, reset=0, input=1f, output=000022  //y[6]
420 clock=0, reset=0, input=24, output=000022
440 clock=1, reset=0, input=24, output=000000  //y[7]
460 clock=0, reset=0, input=29, output=000000
480 clock=1, reset=0, input=29, output=1fffdb  //y[8]
500 clock=0, reset=0, input=2e, output=1fffdb
520 clock=1, reset=0, input=2e, output=00000f  //y[9]
540 clock=0, reset=0, input=33, output=00000f
560 clock=1, reset=0, input=33, output=000107  //y[10]
580 clock=0, reset=0, input=38, output=000107
600 clock=1, reset=0, input=38, output=0002e4  //y[11]
620 clock=0, reset=0, input=3d, output=0002e4
640 clock=1, reset=0, input=3d, output=000562  //y[12]
660 clock=0, reset=0, input=42, output=000562
680 clock=1, reset=0, input=42, output=000810  //y[13]
700 clock=0, reset=0, input=47, output=000810
720 clock=1, reset=0, input=47, output=000aa6  //y[14]
740 clock=0, reset=0, input=4c, output=000aa6
760 clock=1, reset=0, input=4c, output=000d1a  //y[15]
780 clock=0, reset=0, input=51, output=000d1a
800 clock=1, reset=0, input=51, output=000f88  //y[16]
820 clock=0, reset=0, input=56, output=000f88
840 clock=1, reset=0, input=56, output=001200  //y[17]
860 clock=0, reset=0, input=5b, output=001200
880 clock=1, reset=0, input=5b, output=001480  //y[18]
900 clock=0, reset=0, input=60, output=001480
920 clock=1, reset=0, input=60, output=001700  //y[19]
```

Figure 27 Partial results for the functional simulation of the filter test-bench

```
  0 clock=0, reset=0, input=00, output=xxxxxx
 27 clock=0, reset=0, input=00, output=000000
120 clock=1, reset=1, input=00, output=000000
160 clock=1, reset=0, input=01, output=000000
200 clock=1, reset=0, input=06, output=000000
240 clock=1, reset=0, input=0b, output=000000
280 clock=1, reset=0, input=10, output=000000
293 clock=1, reset=0, input=10, output=000002
300 clock=0, reset=0, input=15, output=000002
320 clock=1, reset=0, input=15, output=000002
334 clock=1, reset=0, input=15, output=00000e
360 clock=1, reset=0, input=1a, output=00000e
386 clock=0, reset=0, input=1f, output=000020
400 clock=1, reset=0, input=1f, output=000020
422 clock=0, reset=0, input=24, output=000022
440 clock=1, reset=0, input=24, output=000022
466 clock=0, reset=0, input=29, output=000000
480 clock=1, reset=0, input=29, output=000000
504 clock=0, reset=0, input=2e, output=1fffdb
520 clock=1, reset=0, input=2e, output=1fffdb
548 clock=0, reset=0, input=33, output=00000f
560 clock=1, reset=0, input=33, output=00000f
588 clock=0, reset=0, input=38, output=000107
600 clock=1, reset=0, input=38, output=000107
628 clock=0, reset=0, input=3d, output=0002e4
640 clock=1, reset=0, input=3d, output=0002e4
664 clock=0, reset=0, input=42, output=000562
680 clock=1, reset=0, input=42, output=000562
705 clock=0, reset=0, input=47, output=000810
720 clock=1, reset=0, input=47, output=000810
743 clock=0, reset=0, input=4c, output=000aa6
760 clock=1, reset=0, input=4c, output=000aa6
787 clock=0, reset=0, input=51, output=000d1a
800 clock=1, reset=0, input=51, output=000d1a
825 clock=0, reset=0, input=56, output=000f88
840 clock=1, reset=0, input=56, output=000f88
864 clock=0, reset=0, input=5b, output=001200
880 clock=1, reset=0, input=5b, output=001200
904 clock=0, reset=0, input=60, output=001480
920 clock=1, reset=0, input=60, output=001480
944 clock=0, reset=0, input=65, output=001700
```

Figure 28 Partial results for the timing simulation of the filter test-bench

Table 9 Complete filter performance

	Complete filter using Baugh-Wooley array multipliers and carry-look-ahead adders
Maximum path delay after place & route (ns)	32.133
Area (no. of slices out of 5120)	414
Power consumption (mW)	709.11

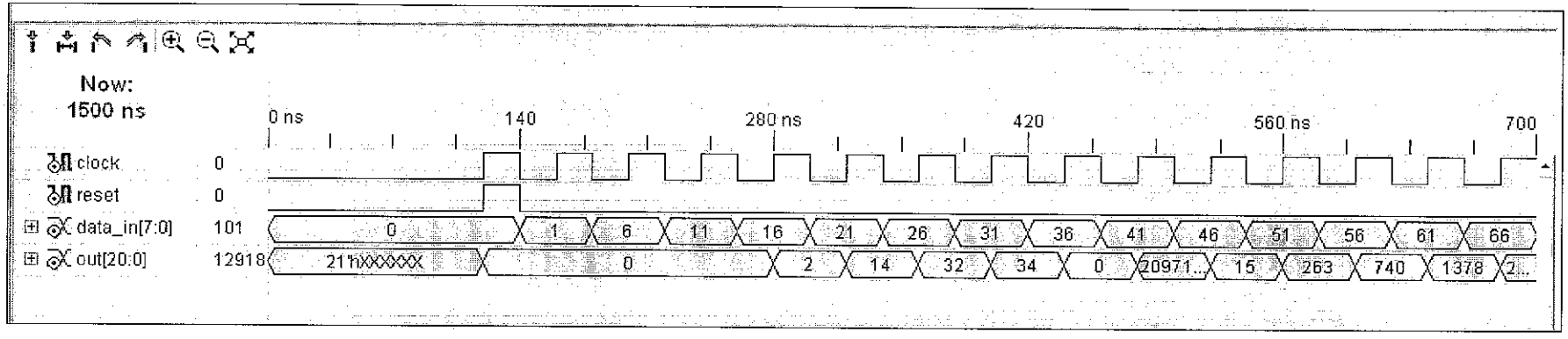


Figure 29 Partial waveforms for the functional simulation of filter test-bench

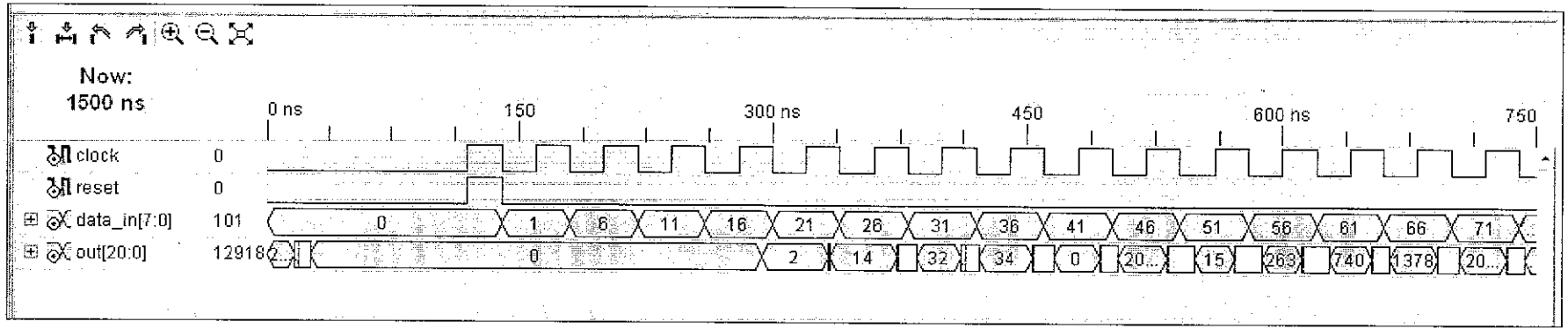


Figure 30 Partial waveforms for the timing simulation of filter test-bench

4.4 HARDWARE SYNTHESIS

The design is programmed into Virtex-II chip and it is tested using a logic analyzer. It is supposed that the logic analyzer provides input to the filter and at the same time, the filter output is observed. Unfortunately, the logic analyzer available is unable to provide input. Thus, the codes are extended to account for the input generator module that is used to provide inputs to the filter manually. This concept is illustrated in Figure 31.

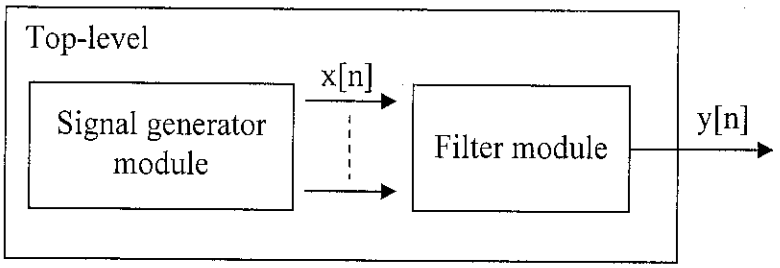


Figure 31 Signal generator module providing inputs to filter

```
/*This program instantiates the signal generator module and
filter module.
*/
`timescale 1ns/1ps
module filter_in(clock,reset,out);

input clock,reset;
output [20:0]out;
wire [7:0]data_in;

input_gen gen(clock,reset,data_in);
filter filt(clock,reset,data_in,out);

endmodule
```

Figure 32 Top-level module

```
//This program generates input data internally to the filter.
`timescale 1ns/1ps
module input_gen(clock,reset,data_in);

input clock,reset;
output [7:0]data_in;
reg [7:0]data_in = 8'h00;

always @(posedge clock or posedge reset)
begin
        if(reset)   data_in <= 8'h00;
        else
             data_in <= data_in + 3'd5;
end

endmodule
```

Figure 33 Verilog codes of signal generator module

4.5 DISCUSSION

The module that describes the radix-4 Booth's multiplier with 8-bit inputs (see Figure 35 in Appendix A) instantiates four 'Boothpar' modules which in turn yield four partial products. All four partial products are summed using a 16-bit CSA. 'Boothpar' module realizes the hardware implementation of recoding logic and multiplexer. In 'CSA_16_booth' module, the 9-bit partial products are required to be shifted accordingly based on the weights of bits in each partial product. Functional and timing simulations for Booth's multiplier are verified and found to be identical.

Baugh-Wooley array multiplier basically consists of AND gates and full adders as reflected by the structure in Figure 10. Functional and timing simulations for Baugh-Wooley multiplier are also verified and found to be identical. From the performance comparison in Table 6, both multipliers have almost similar path delay with Booth's multiplier delay recorded at a slightly lower value. However, the area occupied by Booth's multiplier is 78 slices as compared to 64 slices for Baugh-Wooley multiplier. Power consumption for Baugh-Wooley multiplier is about 30mW less than Booth's multiplier. By looking at the percentage difference, Baugh-Wooley multiplier displays a better performance and hence, it is selected for the filter design.

Basically, for CLA modules, there are multiple instantiations of 'CLA_nsx' modules followed by an instantiation of 'CLA' module. 'CLA_nsx' module performs addition between two 4-bit operands that are not signed extended. On the contrary, 'CLA' module adds two 4-bit operands that are sign extended, where these four bits are the upper four bits of an operand. Sign extension is necessary for the upper four bits in order to obtain the correct result.

Figures 42 and 43 (in Appendix A) show the HDL descriptions for modules 'CLA_nsx' and 'CLA' respectively. It can be seen that the codes are divided into four stages since it is a 4-bit adder in the case of 'CLA_nsx'. The basis to this block of codes is according to the formula given in Equation 3. In the case of 'CLA', there is an extra stage owing to sign extension of operands. Output S4 is the sign bit, which corresponds

to S[16] of top-level module 'CLA_16'. The carry-out bit, CO5 can be discarded since the output range requires only five bits for a 4-bit adder. Higher-order adders can be designed by cascading several 'CLA_nsx' modules with one 'CLA' module for the upper four bits.

The overall adder formed by several CLA instantiations accepts outputs from 19 multipliers simultaneously since the multiplication and addition process is carried out in parallel. Each multiplier output consists of 16 bits, thus there are 304 bits for all outputs of the 19 multipliers. However, the target device has only 172 bonded IOBs. Therefore, a method is used, which is mentioned in 'Software Simulations' section, in order to perform simulations on the adder. Similar problem is encountered by overall adder with several CSA instantiations and the same method is used to resolve it.

The overall adder formed by multiple CSA instantiations (module 'adder_csa' in Figure 59 or 61 in Appendix B) instantiates three 16-bit adders capable of adding five operands, one 16-bit and one 19-bit adder, in which both are capable of adding four operands. This is the best combination of different sizes of adders due to two reasons:

1. If CSA was to add three operands, it will function like a ripple-carry adder, thus the advantage of using CSA cannot be displayed.
2. The more operands that CSA adds, the more number of bits of sign extension is required since adding two operands requires one sign extension. More sign extensions increase hardware.

Functional and timing simulation results for CLA and CSA are done for overall adders that have one and eight input ports. By looking at the performance comparison in Tables 7 and 8, CLA has a significantly smaller area compared to CSA, which are 64.52% and 33.88% less for overall adder with one input port and eight input ports respectively. The trade-offs for the decrease in area are the increase in path delay and power consumption. CLA indicates an increase of 4.08% path delay and 10.54% power consumption for adder with one input port while for adder with eight input ports, an increase of 2.45% path delay and 5.09% power consumption can be observed. It can be

safely said that CLA portrays a better performance compared to CSA judging at the much higher decrease in area. Hence, it is selected for the filter design.

Since the design is an 18th order filter, there are eighteen delay units for the input samples to pass through. The delay units are implemented using D flip-flops where in this design, the input data appears at the output at the positive edge of clock that triggers the flip-flop. In the 'delay' module in Figure 23, it instantiates eighteen flip-flops which are actually cascaded to form a shift register. The HDL description for the complete filter in Figure 25 is rather straightforward. The 'always' construct defines a register that holds an input sample temporarily for one clock cycle before going out to the shift register. The functional and timing simulations for the filter are verified.

In this filter design, memory unit and control unit are omitted because the arithmetic operations are performed in parallel. RAM which is used to store the input samples is replaced by a single register. ROM which is initially suggested to be used to store filter coefficients is not necessary because the coefficients are directly defined as parameter in the multiplier module. Control unit is also not required as the processing of data and output sample, y[n] are all carried out in one clock cycle. The omission of memory unit and control unit introduces simplicity in this design and also the use of less hardware, hence reducing cost.

The functionality of the filter is verified by implementing it into FPGA. During hardware verification, there is a difficulty to predict the filter output because the onboard 24 MHz oscillator is used as clock, which starts running once the board is supplied with power. This problem is highlighted in the preceding chapter. Hence, a manual push button is used in order to test the output of the filter. When the button is pushed, it signifies the triggering of clock and thus, starts the operation of the filter for one clock cycle. The output can then be observed on the logic analyzer for each clock cycle.

CHAPTER 5

CONCLUSION & RECOMMENDATIONS

This project requires the implementation of FIR filter through HDL in which the filter components can be divided into adders, multipliers, memory unit and control unit. Two's complement number representation and eight bits are used to represent input data and filter coefficients. Fixed point numbers are used. In this project, carry-look-ahead adder and carry-save adder are designed and compared. In the case of multiplier, radix-4 Booth's multiplier and Baugh-Wooley array multiplier are designed and compared. Both carry-look-ahead adder and Baugh-Wooley array multiplier display better performance compared to their counterparts. Hence, they are selected to be used in the filter design. The design is an eighteenth order filter and has nineteen filter coefficients. Therefore, the shift register has eighteen D flip-flops cascaded. Memory unit and control unit are omitted because arithmetic operations of the filter are carried out in parallel. The filter employs DF architecture and its performance obtained via simulations is summarized. The complete filter are synthesized, implemented using FPGA and overall functionality is validated through hardware.

Improvements can be made to the current design, which include the following:

i) More structures of adders and multipliers can be compared for their performance.

ii) Other factors that affect the filter performance can be incorporated into the design. These factors include the use of different number representation schemes like sign magnitude and advance techniques like differential coefficient method (DCM).

iii) A combination of sequential and parallel filter implementation approach can be explored to determine the trade-off between consumed area and throughput.

iv) The versatility of this design enables the filter to be modified to other types besides low-pass based on specific applications. However, one limitation is that the verification of the design is rather cumbersome due to the lack of suitable equipment.

REFERENCES

[1] A.T. Erdogan and T. Arslan, "High Throughput FIR Filter Design for Low Power SOC Applications", University of Edinburgh, 2000, pp. 374-378.

[2] A.T. Erdogan and T. Arslan, "Low Power FIR Filter Implementations Based on Coefficient Ordering Algorithm", Proceedings of the IEEE Computer Society Annual Symposium on VLSI Emerging Trends in VLSI Systems Design, 2004.

[3] A.T. Erdogan, M. Hasan and T. Arslan, "Algorithmic Low Power FIR cores", IEE Proc.-Circuits Devices Syst., Vol. 150, No. 3, June 2003, pp. 155-160.

[4] C.H. Wang, A.T. Erdogan, T. Arslan, "High Throughput and Low Power FIR Filtering IP Cores", University of Edinburgh, 2004, pp. 127-130.

[5] A.T. Erdogan and T. Arslan, "Low Power Block Based FIR Filtering Cores", University of Edinburgh, 2003, pp. 341-344.

[6] T. Arslan and A.T. Erdogan, "Low Power Implementation of High Throughput FIR Filters", University of Edinburgh, 2002, pp. 373-376.

[7] A.T. Erdogan, E. Zwyssig and T. Arslan, "Architectural Trade-offs in the Design of Low Power FIR Filtering Cores", IEE Proc.-Circuits Devices Syst., Vol. 151, No. 1, Feb. 2004, pp.10-17.

[8] Emmanuel C. Ifeachor, Barrie W. Jervis, *Digital Signal Processing, A Practical Approach*, 2nd Ed., Prentice Hall, 2002.

[9] Richard S. Sandige, *Digital Design Essentials*, Prentice Hall, 2002.

[10] Prof. Vojin G. Oklobdzija, University of California, "Lecture 9: Multipliers", 11 May 2004, http://lapwww.epfl.ch/courses/comparith/Lectures/VLSI-Arithmetic-Lect-9-Multiplier.pdf

[11] D. Mlynek, "Chapter 6 Arithmetic for Digital Systems", 11 October 1998, http://www.vlsi.wpi.edu/webcourse/ch06/ch06.html

[12] A.T. Erdogan, T. Arslan and D.H. Horrocks, "Low Power Multiplication Schemes for Single Miltiplier CMOS Based FIR Digital Filter Implementations", University of Wales Cardiff, 1997, pp. 1940-1943.

[13] David R. Smith, Paul D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, Prentice Hall, 2001.

[14] T. Arslan, *Chapter 4: VLSI Design*, Institute for System Level Integration/ University of Edinburgh, 2001/2002.

[15] T.R. Padmanabhan, B. Bala Tripura Sundari, *Design Through Verilog HDL*, Wiley Inter-Science, 2004.

[16] Weng Fook Lee, *Verilog Coding for Logic Synthesis*, Wiley Inter-Science, 2003.

[17] Stephen Brown, Zvonko Vranesic, *Fundamentals of Digital Logic with Verilog Design*, McGraw Hill, 2003.

[18] Virtex-II XC2V40/XC2V1000 Reference Board User's Guide.

[19] Software manual for Xilinx, http://www.xilinx.com/support/software_manuals.htm

APPENDICES

APPENDIX A

1. Baugh-Wooley Array Multiplier

```
`timescale 1ns/1ps
module Wooley(A,P);

input [7:0]A;
output [15:0]P;
parameter [7:0]B = 8'h02;

wire [48:0]W;
wire [6:0]W1,W2;
wire W3,C0;
wire sum0,sum1,sum2,sum3,sum4,sum5,sum6,sum7,sum8,sum9,sum10;
wire sum11,sum12,sum13,sum14,sum15,sum16,sum17,sum18,sum19,sum20;
wire sum21,sum22,sum23,sum24,sum25,sum26,sum27,sum28,sum29,sum30;
wire sum31,sum32,sum33,sum34,sum35,sum36,sum37,sum38,sum39,sum40;
wire sum41,sum42,sum43;
wire cout0,cout1,cout2,cout3,cout4,cout5,cout6,cout7,cout8,cout9,cout10;
wire cout11,cout12,cout13,cout14,cout15,cout16,cout17,cout18,cout19,cout20;
wire cout21,cout22,cout23,cout24,cout25,cout26,cout27,cout28,cout29,cout30;
wire cout31,cout32,cout33,cout34,cout35,cout36,cout37,cout38,cout39,cout40;
wire cout41,cout42,cout43,cout44,cout45,cout46,cout47,cout48,cout49,cout50;
wire cout51,cout52,cout53,cout54,cout55,cout56,cout57;

assign W[0] = A[0] & B[0];
assign W[1] = A[1] & B[0];
assign W[2] = A[2] & B[0];
assign W[3] = A[3] & B[0];
assign W[4] = A[4] & B[0];
assign W[5] = A[5] & B[0];
assign W[6] = A[6] & B[0];

assign W[7] = A[0] & B[1];
assign W[8] = A[1] & B[1];
assign W[9] = A[2] & B[1];
assign W[10] = A[3] & B[1];
assign W[11] = A[4] & B[1];
assign W[12] = A[5] & B[1];
assign W[13] = A[6] & B[1];

assign W[14] = A[0] & B[2];
assign W[15] = A[1] & B[2];
assign W[16] = A[2] & B[2];
assign W[17] = A[3] & B[2];
assign W[18] = A[4] & B[2];
assign W[19] = A[5] & B[2];
assign W[20] = A[6] & B[2];
```

continue...

```
assign W[21] = A[0] & B[3];
assign W[22] = A[1] & B[3];
assign W[23] = A[2] & B[3];
assign W[24] = A[3] & B[3];
assign W[25] = A[4] & B[3];
assign W[26] = A[5] & B[3];
assign W[27] = A[6] & B[3];

assign W[28] = A[0] & B[4];
assign W[29] = A[1] & B[4];
assign W[30] = A[2] & B[4];
assign W[31] = A[3] & B[4];
assign W[32] = A[4] & B[4];
assign W[33] = A[5] & B[4];
assign W[34] = A[6] & B[4];

assign W[35] = A[0] & B[5];
assign W[36] = A[1] & B[5];
assign W[37] = A[2] & B[5];
assign W[38] = A[3] & B[5];
assign W[39] = A[4] & B[5];
assign W[40] = A[5] & B[5];
assign W[41] = A[6] & B[5];

assign W[42] = A[0] & B[6];
assign W[43] = A[1] & B[6];
assign W[44] = A[2] & B[6];
assign W[45] = A[3] & B[6];
assign W[46] = A[4] & B[6];
assign W[47] = A[5] & B[6];
assign W[48] = A[6] & B[6];

assign W1[0] = (~A[0])&B[7];
assign W1[1] = (~A[1])&B[7];
assign W1[2] = (~A[2])&B[7];
assign W1[3] = (~A[3])&B[7];
assign W1[4] = (~A[4])&B[7];
assign W1[5] = (~A[5])&B[7];
assign W1[6] = (~A[6])&B[7];

assign W2[0] = A[7]&(~B[0]);
assign W2[1] = A[7]&(~B[1]);
assign W2[2] = A[7]&(~B[2]);
assign W2[3] = A[7]&(~B[3]);
assign W2[4] = A[7]&(~B[4]);
```

continue...

```
assign W2[5] = A[7]&(~B[5]);
assign W2[6] = A[7]&(~B[6]);

wire gnd=0;
wire high=1;
assign W3=A[7]&B[7];
assign P[0]=W[0];
full_adder fa1(gnd,W[1],W[7],P[1],cout0);
full_adder fa2(gnd,W[2],W[8],sum0,cout1);
full_adder fa3(gnd,W[3],W[9],sum1,cout2);
full_adder fa4(gnd,W[4],W[10],sum2,cout3);
full_adder fa5(gnd,W[5],W[11],sum3,cout4);
full_adder fa6(gnd,W[6],W[12],sum4,cout5);
full_adder fa7(gnd,W2[0],W[13],sum5,cout6);
full_adder fa8(W[14],cout0,sum0,P[2],cout7);
full_adder fa9(W[15],cout1,sum1,sum6,cout8);
full_adder fa10(W[16],cout2,sum2,sum7,cout9);
full_adder fa11(W[17],cout3,sum3,sum8,cout10);
full_adder fa12(W[18],cout4,sum4,sum9,cout11);
full_adder fa13(W[19],cout5,sum5,sum10,cout12);
full_adder fa14(W[20],cout6,W2[1],sum11,cout13);
full_adder fa15(W[21],cout7,sum6,P[3],cout14);
full_adder fa16(W[22],cout8,sum7,sum12,cout15);
full_adder fa17(W[23],cout9,sum8,sum13,cout16);
full_adder fa18(W[24],cout10,sum9,sum14,cout17);
full_adder fa19(W[25],cout11,sum10,sum15,cout18);
full_adder fa20(W[26],cout12,sum11,sum16,cout19);
full_adder fa21(W[27],cout13,W2[2],sum17,cout20);
full_adder fa22(W[28],cout14,sum12,P[4],cout21);
full_adder fa23(W[29],cout15,sum13,sum18,cout22);
full_adder fa24(W[30],cout16,sum14,sum19,cout23);
full_adder fa25(W[31],cout17,sum15,sum20,cout24);
full_adder fa26(W[32],cout18,sum16,sum21,cout25);
full_adder fa27(W[33],cout19,sum17,sum22,cout26);
full_adder fa28(W[34],cout20,W2[3],sum23,cout27);
full_adder fa29(W[35],cout21,sum18,P[5],cout28);
full_adder fa30(W[36],cout22,sum19,sum24,cout29);
full_adder fa31(W[37],cout23,sum20,sum25,cout30);
full_adder fa32(W[38],cout24,sum21,sum26,cout31);
full_adder fa33(W[39],cout25,sum22,sum27,cout32);
full_adder fa34(W[40],cout26,sum23,sum28,cout33);
full_adder fa35(W[41],cout27,W2[4],sum29,cout34);
full_adder fa36(W[42],cout28,sum24,P[6],cout35);
full_adder fa37(W[43],cout29,sum25,sum30,cout36);
full_adder fa38(W[44],cout30,sum26,sum31,cout37);
full_adder fa39(W[45],cout31,sum27,sum32,cout38);
full_adder fa40(W[46],cout32,sum28,sum33,cout39);
full_adder fa41(W[47],cout33,sum29,sum34,cout40);
full_adder fa42(W[48],cout34,W2[5],sum35,cout41);
full_adder fa43(W1[0],cout35,sum30,sum36,cout42);
full_adder fa44(W1[1],cout36,sum31,sum37,cout43);
full_adder fa45(W1[2],cout37,sum32,sum38,cout44);
full_adder fa46(W1[3],cout38,sum33,sum39,cout45);
full_adder fa47(W1[4],cout39,sum34,sum40,cout46);
full_adder fa48(W1[5],cout40,sum35,sum41,cout47);
full_adder fa49(W1[6],cout41,W2[6],sum42,cout48);
full_adder fa50(W3,~A[7],~B[7],sum43,cout49);
full_adder fa51(A[7],B[7],sum36,P[7],cout50);
full_adder fa52(cout50,cout42,sum37,P[8],cout51);
full_adder fa53(cout51,cout43,sum38,P[9],cout52);
full_adder fa54(cout52,cout44,sum39,P[10],cout53);
full_adder fa55(cout53,cout45,sum40,P[11],cout54);
full_adder fa56(cout54,cout46,sum41,P[12],cout55);
full_adder fa57(cout55,cout47,sum42,P[13],cout56);
full_adder fa58(cout56,cout48,sum43,P[14],cout57);
full_adder fa59(cout57,cout49,high,P[15],C0);

endmodule
```

Figure 34 Baugh-Wooley multiplier with instantiations of full adders

2. Radix-4 Booth's Multiplier

```
//Radix-4 Booth multiplier with 8-bit input operands, generating
//16-bit result.
module Booth(A,B,R);

input  [7:0]A;
input  [7:0]B;
output [15:0]R;
wire  [8:0]P1,P2,P3,P4;
wire  [8:0]B1;

assign B1 = B << 1;
Boothpar par1(A,B1[2:0],P1);
Boothpar par2(A,B1[4:2],P2);
Boothpar par3(A,B1[6:4],P3);
Boothpar par4(A,B1[8:6],P4);

CSA_16_booth csa(P1,P2,P3,P4,R);

endmodule
```

Figure 35 Radix-4 Booth's multiplier with 8-bit inputs

```
/*This program implements the recoding logic and
multiplexer for radix-4 Booth's algorithm to generate
partial products.
*/
module Boothpar(A,B,P);

input [7:0]A;
input [2:0]B;
output [8:0]P;
wire A8,P9;
wire [8:0]out;

/*sign extension needed for the case when MSB of multiplicand is 1
and recoded version of 3-bit multiplier group is 1(M=1) and also
MSB of multiplier group is 0.
*/
assign A8=A[7];                                          // sign extension
assign M = B[0]^B[1];
assign M2 = ~(M | (B[1]~^B[2]));  // 2*multiplicand
assign out[0] = (M & A[0]) ^ B[2];
assign out[1] = ((M2 & A[0]) | (M & A[1])) ^ B[2];
assign out[2] = ((M2 & A[1]) | (M & A[2])) ^ B[2];
assign out[3] = ((M2 & A[2]) | (M & A[3])) ^ B[2];
assign out[4] = ((M2 & A[3]) | (M & A[4])) ^ B[2];
assign out[5] = ((M2 & A[4]) | (M & A[5])) ^ B[2];
assign out[6] = ((M2 & A[5]) | (M & A[6])) ^ B[2];
assign out[7] = ((M2 & A[6]) | (M & A[7])) ^ B[2];
assign out[8] = ((M2 & A[7]) | (M & A8)) ^ B[2];
assign P = out + B[2];
assign P9 = (M2 & A8) ^ B[2];

endmodule
```

Figure 36 Recoding logic and multiplexer to generate partial products

```
/*This program adds four 16-bit operands, creating a 16-bit CSA.
The 9-bit input operands are internally signed extended to 16 bits.
Input operands are shifted left accordingly before addition to
implement the Booth's algorithm.
*/
module CSA_16_booth(A,B,C,D,S);

input  [8:0]A;                    // P1
input  [8:0]B;                    // P2<<2
input  [8:0]C;                    // P3<<4
input  [8:0]D;                    // P4<<6
output [15:0]S;
wire   S16,S17;
wire   sum0,sum1,sum2,sum3,sum4,sum5,sum6,sum7,sum8,sum9,sum10;
wire   sum11,sum12,sum13,sum14,sum15,sum16,sum17,sum18,sum19,sum20;
wire   sum21,sum22,sum23,sum24,sum25,sum26,sum27,sum28,sum29,sum30;
wire   cout0,cout1,cout2,cout3,cout4,cout5,cout6,cout7,cout8,cout9,cout10;
wire   cout11,cout12,cout13,cout14,cout15,cout16,cout17,cout18,cout19,cout20;
wire   cout21,cout22,cout23,cout24,cout25,cout26,cout27,cout28,cout29,cout30;
wire   cout31,cout32,cout33,cout34,cout35,cout36,cout37,cout38,cout39,cout40;
wire   cout41,cout42,cout43,cout44,cout45,cout46;
wire   gnd=0;

full_adder    fa1(gnd,A[0],gnd,sum0,cout0);
full_adder    fa2(gnd,A[1],gnd,sum1,cout1);
full_adder    fa3(gnd,A[2],B[0],sum2,cout2);
full_adder    fa4(gnd,A[3],B[1],sum3,cout3);
full_adder    fa5(C[0],A[4],B[2],sum4,cout4);
full_adder    fa6(C[1],A[5],B[3],sum5,cout5);
full_adder    fa7(C[2],A[6],B[4],sum6,cout6);
full_adder    fa8(C[3],A[7],B[5],sum7,cout7);
full_adder    fa9(C[4],A[8],B[6],sum8,cout8);
full_adder    fa10(C[5],A[8],B[7],sum9,cout9);
full_adder    fa11(C[6],A[8],B[8],sum10,cout10);
full_adder    fa12(C[7],A[8],B[8],sum11,cout11);
full_adder    fa13(C[8],A[8],B[8],sum12,cout12);
full_adder    fa14(C[8],A[8],B[8],sum13,cout13);
full_adder    fa15(C[8],A[8],B[8],sum14,cout14);
full_adder    fa16(C[8],A[8],B[8],sum15,cout15);
```

continue...

```
full_adder    fa17(gnd,sum0,gnd,S[0],cout16);
full_adder    fa18(cout0,sum1,gnd,sum16,cout17);
full_adder    fa19(cout1,sum2,gnd,sum17,cout18);
full_adder    fa20(cout2,sum3,gnd,sum18,cout19);
full_adder    fa21(cout3,sum4,gnd,sum19,cout20);
full_adder    fa22(cout4,sum5,gnd,sum20,cout21);
full_adder    fa23(cout5,sum6,D[0],sum21,cout22);
full_adder    fa24(cout6,sum7,D[1],sum22,cout23);
full_adder    fa25(cout7,sum8,D[2],sum23,cout24);
full_adder    fa26(cout8,sum9,D[3],sum24,cout25);
full_adder    fa27(cout9,sum10,D[4],sum25,cout26);
full_adder    fa28(cout10,sum11,D[5],sum26,cout27);
full_adder    fa29(cout11,sum12,D[6],sum27,cout28);
full_adder    fa30(cout12,sum13,D[7],sum28,cout29);
full_adder    fa31(cout13,sum14,D[8],sum29,cout30);
full_adder    fa32(cout14,sum15,D[8],sum30,cout31);

full_adder    fa33(gnd,sum16,cout16,S[1],cout32);
full_adder    fa34(cout32,sum17,cout17,S[2],cout33);
full_adder    fa35(cout33,sum18,cout18,S[3],cout34);
full_adder    fa36(cout34,sum19,cout19,S[4],cout35);
full_adder    fa37(cout35,sum20,cout20,S[5],cout36);
full_adder    fa38(cout36,sum21,cout21,S[6],cout37);
full_adder    fa39(cout37,sum22,cout22,S[7],cout38);
full_adder    fa40(cout38,sum23,cout23,S[8],cout39);
full_adder    fa41(cout39,sum24,cout24,S[9],cout40);
full_adder    fa42(cout40,sum25,cout25,S[10],cout41);
full_adder    fa43(cout41,sum26,cout26,S[11],cout42);
full_adder    fa44(cout42,sum27,cout27,S[12],cout43);
full_adder    fa45(cout43,sum28,cout28,S[13],cout44);
full_adder    fa46(cout44,sum29,cout29,S[14],cout45);
full_adder    fa47(cout45,sum30,cout30,S[15],cout46);
full_adder    fa48(cout46,cout31,cout15,S16,S17);

endmodule
```

Figure 37 CSA for Booth's multiplier to sum all partial products

```
`timescale 1ns/1ps
module Booth_tst;

reg   [7:0]A,B;
wire [15:0]R;

Booth booth1(A,B,R);
initial
begin
      A = 8'h00; B = 8'h00;
      #100 A = 8'h01; B = 8'h10;
      #50 A = 8'h11; B = 8'h1a;
      #50 A = 8'h21; B = 8'h2b;
      #50 A = 8'h31; B = 8'h32;
      #50 A = 8'h83; B = 8'h30;
      #50 A = 8'ha1; B = 8'h1a;
      #50 A = 8'hdc; B = 8'h9b;
      #50 A = 8'hfb; B = 8'hc2;
end
initial
$monitor($realtime ," A=%b, B=%b, product=%h", A,B,R);

endmodule
```

Figure 38 Test-bench for radix-4 Booth's multiplier

3. Carry-Save Adder (CSA)

```
/*This program adds four 16-bit operands, creating a 16-bit CSA.
Each operand is sign-extended to generate the 16th and 17th bit.
(One bit sign extension for addition of two operands)
*/
module CSA_16(A,B,C,D,S);

input  [15:0]A;
input  [15:0]B;
input  [15:0]C;
input  [15:0]D;
output [17:0]S;       // S184519 not needed as output,hence declared as wire

wire   A16,A17,B16,B17,C16,C17,D16,D17,S18,S19;
wire   sum0,sum1,sum2,sum3,sum4,sum5,sum6,sum7,sum8,sum9,sum10;
wire   sum11,sum12,sum13,sum14,sum15,sum16,sum17,sum18,sum19,sum20;
wire   sum21,sum22,sum23,sum24,sum25,sum26,sum27,sum28,sum29,sum30;
wire   sum31,sum32,sum33,sum34;
wire   cout0,cout1,cout2,cout3,cout4,cout5,cout6,cout7,cout8,cout9,cout10;
wire   cout11,cout12,cout13,cout14,cout15,cout16,cout17,cout18,cout19,cout20;
wire   cout21,cout22,cout23,cout24,cout25,cout26,cout27,cout28,cout29,cout30;
wire   cout31,cout32,cout33,cout34,cout35,cout36,cout37,cout38,cout39,cout40;
wire   cout41,cout42,cout43,cout44,cout45,cout46,cout47,cout48,cout49,cout50;
wire   cout51,cout52;
wire   gnd=0;

assign A16=A[15],B16=B[15],C16=C[15],D16=D[15];
assign A17=A[15],B17=B[15],C17=C[15],D17=D[15];
full_adder    fa1(C[0],A[0],B[0],sum0,cout0);
full_adder    fa2(C[1],A[1],B[1],sum1,cout1);
full_adder    fa3(C[2],A[2],B[2],sum2,cout2);
full_adder    fa4(C[3],A[3],B[3],sum3,cout3);
full_adder    fa5(C[4],A[4],B[4],sum4,cout4);
full_adder    fa6(C[5],A[5],B[5],sum5,cout5);
full_adder    fa7(C[6],A[6],B[6],sum6,cout6);
full_adder    fa8(C[7],A[7],B[7],sum7,cout7);
full_adder    fa9(C[8],A[8],B[8],sum8,cout8);
full_adder    fa10(C[9],A[9],B[9],sum9,cout9);
full_adder    fa11(C[10],A[10],B[10],sum10,cout10);
full_adder    fa12(C[11],A[11],B[11],sum11,cout11);
full_adder    fa13(C[12],A[12],B[12],sum12,cout12);
full_adder    fa14(C[13],A[13],B[13],sum13,cout13);
full_adder    fa15(C[14],A[14],B[14],sum14,cout14);
full_adder    fa16(C[15],A[15],B[15],sum15,cout15);
full_adder    fa17(C16,A16,B16,sum16,cout16);
full_adder    fa18(C17,A17,B17,sum17,cout17);
```

continue...

```
full_adder    fa19(gnd,sum0,D[0],S[0],cout18);
full_adder    fa20(cout0,sum1,D[1],sum18,cout19);
full_adder    fa21(cout1,sum2,D[2],sum19,cout20);
full_adder    fa22(cout2,sum3,D[3],sum20,cout21);
full_adder    fa23(cout3,sum4,D[4],sum21,cout22);
full_adder    fa24(cout4,sum5,D[5],sum22,cout23);
full_adder    fa25(cout5,sum6,D[6],sum23,cout24);
full_adder    fa26(cout6,sum7,D[7],sum24,cout25);
full_adder    fa27(cout7,sum8,D[8],sum25,cout26);
full_adder    fa28(cout8,sum9,D[9],sum26,cout27);
full_adder    fa29(cout9,sum10,D[10],sum27,cout28);
full_adder    fa30(cout10,sum11,D[11],sum28,cout29);
full_adder    fa31(cout11,sum12,D[12],sum29,cout30);
full_adder    fa32(cout12,sum13,D[13],sum30,cout31);
full_adder    fa33(cout13,sum14,D[14],sum31,cout32);
full_adder    fa34(cout14,sum15,D[15],sum32,cout33);
full_adder    fa35(cout15,sum16,D16,sum33,cout34);
full_adder    fa36(cout16,sum17,D17,sum34,cout35);

full_adder    fa37(gnd,sum18,cout18,S[1],cout36);
full_adder    fa38(cout36,sum19,cout19,S[2],cout37);
full_adder    fa39(cout37,sum20,cout20,S[3],cout38);
full_adder    fa40(cout38,sum21,cout21,S[4],cout39);
full_adder    fa41(cout39,sum22,cout22,S[5],cout40);
full_adder    fa42(cout40,sum23,cout23,S[6],cout41);
full_adder    fa43(cout41,sum24,cout24,S[7],cout42);
full_adder    fa44(cout42,sum25,cout25,S[8],cout43);
full_adder    fa45(cout43,sum26,cout26,S[9],cout44);
full_adder    fa46(cout44,sum27,cout27,S[10],cout45);
full_adder    fa47(cout45,sum28,cout28,S[11],cout46);
full_adder    fa48(cout46,sum29,cout29,S[12],cout47);
full_adder    fa49(cout47,sum30,cout30,S[13],cout48);
full_adder    fa50(cout48,sum31,cout31,S[14],cout49);
full_adder    fa51(cout49,sum32,cout32,S[15],cout50);
full_adder    fa52(cout50,sum33,cout33,S[16],cout51);
full_adder    fa53(cout51,sum34,cout34,S[17],cout52);
full_adder    fa54(cout52,cout35,cout17,S18,S19);

endmodule
```

Figure 39 16-bit CSA adding four operands

```
/*This program adds five 16-bit operands, creating a 16-bit CSA.
Each operand is sign-extended to generate the 16th,17th and 18th bit.
(One bit sign extension for addition of two operands)
*/
module CSA_16_5(A,B,C,D,E,S);

input  [15:0]A,B,C,D,E;
output [18:0]S;

wire A16,A17,A18,B16,B17,B18,C16,C17,C18,D16,D17,D18,E16,E17,E18;
wire S19,S20,S21;
wire sum0,sum1,sum2,sum3,sum4,sum5,sum6,sum7,sum8,sum9,sum10;
wire sum11,sum12,sum13,sum14,sum15,sum16,sum17,sum18,sum19,sum20;
wire sum21,sum22,sum23,sum24,sum25,sum26,sum27,sum28,sum29,sum30;
wire sum31,sum32,sum33,sum34,sum35,sum36,sum37,sum38,sum39,sum40;
wire sum41,sum42,sum43,sum44,sum45,sum46,sum47,sum48,sum49,sum50;
wire sum51,sum52,sum53,sum54,sum55,sum56;
wire cout0,cout1,cout2,cout3,cout4,cout5,cout6,cout7,cout8,cout9,cout10;
wire cout11,cout12,cout13,cout14,cout15,cout16,cout17,cout18,cout19,cout20;
wire cout21,cout22,cout23,cout24,cout25,cout26,cout27,cout28,cout29,cout30;
wire cout31,cout32,cout33,cout34,cout35,cout36,cout37,cout38,cout39,cout40;
wire cout41,cout42,cout43,cout44,cout45,cout46,cout47,cout48,cout49,cout50;
wire cout51,cout52,cout53,cout54,cout55,cout56,cout57,cout58,cout59,cout60;
wire cout61,cout62,cout63,cout64,cout65,cout66,cout67,cout68,cout69,cout70;
wire cout71,cout72,cout73,cout74,cout75,cout76;
wire gnd=0;

assign A16=A[15],B16=B[15],C16=C[15],D16=D[15],E16=E[15];
assign A17=A[15],B17=B[15],C17=C[15],D17=D[15],E17=E[15];
assign A18=A[15],B18=B[15],C18=C[15],D18=D[15],E18=E[15];
full_adder    fa1(C[0],A[0],B[0],sum0,cout0);
full_adder    fa2(C[1],A[1],B[1],sum1,cout1);
full_adder    fa3(C[2],A[2],B[2],sum2,cout2);
full_adder    fa4(C[3],A[3],B[3],sum3,cout3);
full_adder    fa5(C[4],A[4],B[4],sum4,cout4);
full_adder    fa6(C[5],A[5],B[5],sum5,cout5);
full_adder    fa7(C[6],A[6],B[6],sum6,cout6);
full_adder    fa8(C[7],A[7],B[7],sum7,cout7);
full_adder    fa9(C[8],A[8],B[8],sum8,cout8);
full_adder    fa10(C[9],A[9],B[9],sum9,cout9);
full_adder    fa11(C[10],A[10],B[10],sum10,cout10);
full_adder    fa12(C[11],A[11],B[11],sum11,cout11);
full_adder    fa13(C[12],A[12],B[12],sum12,cout12);
full_adder    fa14(C[13],A[13],B[13],sum13,cout13);
full_adder    fa15(C[14],A[14],B[14],sum14,cout14);
```

continue…

```
full_adder    fa16(C[15],A[15],B[15],sum15,cout15);
full_adder    fa17(C16,A16,B16,sum16,cout16);
full_adder    fa18(C17,A17,B17,sum17,cout17);
full_adder    fa19(C18,A18,B18,sum18,cout18);

full_adder    fa20(gnd,sum0,D[0],sum19,cout19);
full_adder    fa21(cout0,sum1,D[1],sum20,cout20);
full_adder    fa22(cout1,sum2,D[2],sum21,cout21);
full_adder    fa23(cout2,sum3,D[3],sum22,cout22);
full_adder    fa24(cout3,sum4,D[4],sum23,cout23);
full_adder    fa25(cout4,sum5,D[5],sum24,cout24);
full_adder    fa26(cout5,sum6,D[6],sum25,cout25);
full_adder    fa27(cout6,sum7,D[7],sum26,cout26);
full_adder    fa28(cout7,sum8,D[8],sum27,cout27);
full_adder    fa29(cout8,sum9,D[9],sum28,cout28);
full_adder    fa30(cout9,sum10,D[10],sum29,cout29);
full_adder    fa31(cout10,sum11,D[11],sum30,cout30);
full_adder    fa32(cout11,sum12,D[12],sum31,cout31);
full_adder    fa33(cout12,sum13,D[13],sum32,cout32);
full_adder    fa34(cout13,sum14,D[14],sum33,cout33);
full_adder    fa35(cout14,sum15,D[15],sum34,cout34);
full_adder    fa36(cout15,sum16,D16,sum35,cout35);
full_adder    fa37(cout16,sum17,D17,sum36,cout36);
full_adder    fa38(cout17,sum18,D18,sum37,cout37);

full_adder    fa39(gnd,sum19,E[0],S[0],cout38);
full_adder    fa40(cout19,sum20,E[1],sum38,cout39);
full_adder    fa41(cout20,sum21,E[2],sum39,cout40);
full_adder    fa42(cout21,sum22,E[3],sum40,cout41);
full_adder    fa43(cout22,sum23,E[4],sum41,cout42);
full_adder    fa44(cout23,sum24,E[5],sum42,cout43);
full_adder    fa45(cout24,sum25,E[6],sum43,cout44);
full_adder    fa46(cout25,sum26,E[7],sum44,cout45);
full_adder    fa47(cout26,sum27,E[8],sum45,cout46);
full_adder    fa48(cout27,sum28,E[9],sum46,cout47);
full_adder    fa49(cout28,sum29,E[10],sum47,cout48);
full_adder    fa50(cout29,sum30,E[11],sum48,cout49);
full_adder    fa51(cout30,sum31,E[12],sum49,cout50);
full_adder    fa52(cout31,sum32,E[13],sum50,cout51);
full_adder    fa53(cout32,sum33,E[14],sum51,cout52);
full_adder    fa54(cout33,sum34,E[15],sum52,cout53);
full_adder    fa55(cout34,sum35,E16,sum53,cout54);
full_adder    fa56(cout35,sum36,E17,sum54,cout55);
full_adder    fa57(cout36,sum37,E18,sum55,cout56);
full_adder    fa58(gnd,cout18,cout37,sum56,cout57);

full_adder    fa59(gnd,sum38,cout38,S[1],cout58);
full_adder    fa60(cout58,sum39,cout39,S[2],cout59);
full_adder    fa61(cout59,sum40,cout40,S[3],cout60);
full_adder    fa62(cout60,sum41,cout41,S[4],cout61);
full_adder    fa63(cout61,sum42,cout42,S[5],cout62);
full_adder    fa64(cout62,sum43,cout43,S[6],cout63);
full_adder    fa65(cout63,sum44,cout44,S[7],cout64);
full_adder    fa66(cout64,sum45,cout45,S[8],cout65);
full_adder    fa67(cout65,sum46,cout46,S[9],cout66);
full_adder    fa68(cout66,sum47,cout47,S[10],cout67);
full_adder    fa69(cout67,sum48,cout48,S[11],cout68);
full_adder    fa70(cout68,sum49,cout49,S[12],cout69);
full_adder    fa71(cout69,sum50,cout50,S[13],cout70);
full_adder    fa72(cout70,sum51,cout51,S[14],cout71);
full_adder    fa73(cout71,sum52,cout52,S[15],cout72);
full_adder    fa74(cout72,sum53,cout53,S[16],cout73);
full_adder    fa75(cout73,sum54,cout54,S[17],cout74);
full_adder    fa76(cout74,sum55,cout55,S[18],cout75);
full_adder    fa77(cout75,sum56,cout56,S19,cout76);
full_adder    fa78(cout76,cout57,gnd,S20,S21);

endmodule
```

Figure 40 16-bit CSA adding five operands

```
/*This program adds four 19-bit operands, creating a 19-bit CSA.
Each operand is sign-extended to generate the 19th and 20th bit.
*/
module CSA_19(A,B,C,D,S);

input  [18:0]A,B,C,D;
output [20:0]S;

wire A19,A20,B19,B20,C19,C20,D19,D20,S21,S22;
wire sum0,sum1,sum2,sum3,sum4,sum5,sum6,sum7,sum8,sum9,sum10;
wire sum11,sum12,sum13,sum14,sum15,sum16,sum17,sum18,sum19,sum20;
wire sum21,sum22,sum23,sum24,sum25,sum26,sum27,sum28,sum29,sum30;
wire sum31,sum32,sum33,sum34,sum35,sum36,sum37,sum38,sum39,sum40;
wire cout0,cout1,cout2,cout3,cout4,cout5,cout6,cout7,cout8,cout9,cout10;
wire cout11,cout12,cout13,cout14,cout15,cout16,cout17,cout18,cout19,cout20;
wire cout21,cout22,cout23,cout24,cout25,cout26,cout27,cout28,cout29,cout30;
wire cout31,cout32,cout33,cout34,cout35,cout36,cout37,cout38,cout39,cout40;
wire cout41,cout42,cout43,cout44,cout45,cout46,cout47,cout48,cout49,cout50;
wire cout51,cout52,cout53,cout54,cout55,cout56,cout57,cout58,cout59,cout60;
wire cout61;
wire gnd=0;

assign A19=A[18],B19=B[18],C19=C[18],D19=D[18];
assign A20=A[18],B20=B[18],C20=C[18],D20=D[18];
full_adder    fa1(C[0],A[0],B[0],sum0,cout0);
full_adder    fa2(C[1],A[1],B[1],sum1,cout1);
full_adder    fa3(C[2],A[2],B[2],sum2,cout2);
full_adder    fa4(C[3],A[3],B[3],sum3,cout3);
full_adder    fa5(C[4],A[4],B[4],sum4,cout4);
full_adder    fa6(C[5],A[5],B[5],sum5,cout5);
full_adder    fa7(C[6],A[6],B[6],sum6,cout6);
full_adder    fa8(C[7],A[7],B[7],sum7,cout7);
full_adder    fa9(C[8],A[8],B[8],sum8,cout8);
full_adder    fa10(C[9],A[9],B[9],sum9,cout9);
full_adder    fa11(C[10],A[10],B[10],sum10,cout10);
full_adder    fa12(C[11],A[11],B[11],sum11,cout11);
full_adder    fa13(C[12],A[12],B[12],sum12,cout12);
full_adder    fa14(C[13],A[13],B[13],sum13,cout13);
full_adder    fa15(C[14],A[14],B[14],sum14,cout14);
full_adder    fa16(C[15],A[15],B[15],sum15,cout15);
full_adder    fa17(C[16],A[16],B[16],sum16,cout16);
full_adder    fa18(C[17],A[17],B[17],sum17,cout17);
full_adder    fa19(C[18],A[18],B[18],sum18,cout18);
full_adder    fa20(C19,A19,B19,sum19,cout19);
full_adder    fa21(C20,A20,B20,sum20,cout20);
```

continue...

```
full_adder     fa22(gnd,sum0,D[0],S[0],cout21);
full_adder     fa23(cout0,sum1,D[1],sum21,cout22);
full_adder     fa24(cout1,sum2,D[2],sum22,cout23);
full_adder     fa25(cout2,sum3,D[3],sum23,cout24);
full_adder     fa26(cout3,sum4,D[4],sum24,cout25);
full_adder     fa27(cout4,sum5,D[5],sum25,cout26);
full_adder     fa28(cout5,sum6,D[6],sum26,cout27);
full_adder     fa29(cout6,sum7,D[7],sum27,cout28);
full_adder     fa30(cout7,sum8,D[8],sum28,cout29);
full_adder     fa31(cout8,sum9,D[9],sum29,cout30);
full_adder     fa32(cout9,sum10,D[10],sum30,cout31);
full_adder     fa33(cout10,sum11,D[11],sum31,cout32);
full_adder     fa34(cout11,sum12,D[12],sum32,cout33);
full_adder     fa35(cout12,sum13,D[13],sum33,cout34);
full_adder     fa36(cout13,sum14,D[14],sum34,cout35);
full_adder     fa37(cout14,sum15,D[15],sum35,cout36);
full_adder     fa38(cout15,sum16,D[16],sum36,cout37);
full_adder     fa39(cout16,sum17,D[17],sum37,cout38);
full_adder     fa40(cout17,sum18,D[18],sum38,cout39);
full_adder     fa41(cout18,sum19,D19,sum39,cout40);
full_adder     fa42(cout19,sum20,D20,sum40,cout41);

full_adder     fa43(gnd,sum21,cout21,S[1],cout42);
full_adder     fa44(cout42,sum22,cout22,S[2],cout43);
full_adder     fa45(cout43,sum23,cout23,S[3],cout44);
full_adder     fa46(cout44,sum24,cout24,S[4],cout45);
full_adder     fa47(cout45,sum25,cout25,S[5],cout46);
full_adder     fa48(cout46,sum26,cout26,S[6],cout47);
full_adder     fa49(cout47,sum27,cout27,S[7],cout48);
full_adder     fa50(cout48,sum28,cout28,S[8],cout49);
full_adder     fa51(cout49,sum29,cout29,S[9],cout50);
full_adder     fa52(cout50,sum30,cout30,S[10],cout51);
full_adder     fa53(cout51,sum31,cout31,S[11],cout52);
full_adder     fa54(cout52,sum32,cout32,S[12],cout53);
full_adder     fa55(cout53,sum33,cout33,S[13],cout54);
full_adder     fa56(cout54,sum34,cout34,S[14],cout55);
full_adder     fa57(cout55,sum35,cout35,S[15],cout56);
full_adder     fa58(cout56,sum36,cout36,S[16],cout57);
full_adder     fa59(cout57,sum37,cout37,S[17],cout58);
full_adder     fa60(cout58,sum38,cout38,S[18],cout59);
full_adder     fa61(cout59,sum39,cout39,S[19],cout60);
full_adder     fa62(cout60,sum40,cout40,S[20],cout61);
full_adder     fa63(cout61,cout41,cout20,S21,S22);

endmodule
```

Figure 41 19-bit CSA adding four operands

4. Carry-Look-Ahead Adder (CLA)

```
/*This program adds two 4-bit operands, creating a 4-bit adder.
No sign extension to the operands, hence only suitable for
addition of unsigned numbers.
*/

module CLA_nsx(A,B,CI0,S,CO4);

input  [3:0]A;                          // Input-four bits
input  [3:0]B;
input  CI0;
output [3:0]S;
output CO4;                             // Carry-out bit

wire   CO1,CO2,CO3;
wire   G0,G1,G2,G3;
wire   P0,P1,P2,P3;
wire   c1,c2,c3,c4,c5,c6,c7,c8,c9,c10;
wire   ss1,ss2,ss3,ss4;

assign ss1 = A[0] ^ B[0];
assign S[0] = CI0 ^ ss1;
assign G0 = A[0] & B[0];
assign P0 = A[0] | B[0];
assign c1 = P0 & CI0;
assign CO1 = G0 | c1;

assign ss2 = A[1] ^ B[1];
assign S[1] = CO1 ^ ss2;
assign G1 = A[1] & B[1];
assign P1 = A[1] | B[1];
assign c2 = G0 & P1;
assign c3 = P0 & P1 & CI0;
assign CO2 = G1 | c2 | c3;

assign ss3 = A[2] ^ B[2];
assign S[2] = CO2 ^ ss3;
assign G2 = A[2] & B[2];
assign P2 = A[2] | B[2];
assign c4 = G1 & P2;
assign c5 = G0 & P1 & P2;
assign c6 = P0 & P1 & P2 & CI0;
assign CO3 = G2 | c4 | c5 | c6;

assign ss4 = A[3] ^ B[3];
assign S[3] = CO3 ^ ss4;
assign G3 = A[3] & B[3];
assign P3 = A[3] | B[3];
assign c7 = G2 & P3;
assign c8 = G1 & P2 & P3;
assign c9 = G0 & P1 & P2 & P3;
assign c10 = P0 & P1 & P2 & P3 & CI0;
assign CO4 = G3 | c7 | c8 | c9 | c10;

endmodule
```

Figure 42 4-bit CLA without sign extension

```
/*This program adds two 4-bit operands, creating a 4-bit adder.
The two operands are sign-extended to create 5-bit operands,
generating C05, which is insignificant to the result.
*/

module CLA(A,B,CI0,S,S4);

input  [3:0]A;                          // Input=four bits
input  [3:0]B;
input  CI0;
output [3:0]S;
output S4;                              // Carry-out bit

wire   A4,B4;
wire   C01,C02,C03,C04,C05; // C05 is for overflow due to sign extension
wire   G0,G1,G2,G3,G4;
wire   P0,P1,P2,P3,P4;
wire   c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15;
wire   ss1,ss2,ss3,ss4;

assign A4=A[3], B4=B[3];            // sign bit - sign extension

assign ss1 = A[0] ^ B[0];
assign S[0] = CI0 ^ ss1;
assign G0 = A[0] & B[0];
assign P0 = A[0] | B[0];
assign c1 = P0 & CI0;
assign C01 = G0 | c1;

assign ss2 = A[1] ^ B[1];
assign S[1] = C01 ^ ss2;
assign G1 = A[1] & B[1];
assign P1 = A[1] | B[1];
assign c2 = G0 & P1;
assign c3 = P0 & P1 & CI0;
assign C02 = G1 | c2 | c3;

assign ss3 = A[2] ^ B[2];
assign S[2] = C02 ^ ss3;
assign G2 = A[2] & B[2];
assign P2 = A[2] | B[2];
assign c4 = G1 & P2;
assign c5 = G0 & P1 & P2;
assign c6 = P0 & P1 & P2 & CI0;
assign C03 = G2 | c4 | c5 | c6;

assign ss4 = A[3] ^ B[3];
assign S[3] = C03 ^ ss4;
assign G3 = A[3] & B[3];
assign P3 = A[3] | B[3];
assign c7 = G2 & P3;
assign c8 = G1 & P2 & P3;
assign c9 = G0 & P1 & P2 & P3;
assign c10 = P0 & P1 & P2 & P3 & CI0;
assign C04 = G3 | c7 | c8 | c9 | c10;

assign S4 = C04 ^ ss4;
assign G4 = A4 & B4;
assign P4 = A4 | B4;
assign c11 = G3 & P4;
assign c12 = G2 & P3 & P4;
assign c13 = G1 & P2 & P3 & P4;
assign c14 = G0 & P1 & P2 & P3 & P4;
assign c15 = P0 & P1 & P2 & P3 & P4 & CI0;
assign C05 = G4 | c11 | c12 | c13 | c14 | c15;

endmodule
```

Figure 43 4-bit CLA with sign extension

```
// 18-bit CLA
module CLA_18(A,B,S);

input  [17:0]A,B;
output [18:0]S;
wire   A18,A19,B18,B19,S19,S20;
wire   C01,C02,C03,C04;
wire   CI0 = 0;

CLA_nsx clan1(A[3:0],B[3:0],CI0,S[3:0],C01);
CLA_nsx clan2(A[7:4],B[7:4],C01,S[7:4],C02);
CLA_nsx clan3(A[11:8],B[11:8],C02,S[11:8],C03);
CLA_nsx clan4(A[15:12],B[15:12],C03,S[15:12],C04);

assign A18=A[17],A19=A[17];
assign B18=B[17],B19=B[17];

CLA cla1({A19,A18,A[17:16]},{B19,B18,B[17:16]},C04,{S19,S[18:16]},S20);

endmodule
```

Figure 44 18-bit CLA

```
// 19-bit CLA
module CLA_19(A,B,S);

input  [18:0]A,B;
output [19:0]S;
wire   A19,B19,S20;
wire   C01,C02,C03,C04;
wire   CI0 = 0;

CLA_nsx clan1(A[3:0],B[3:0],CI0,S[3:0],C01);
CLA_nsx clan2(A[7:4],B[7:4],C01,S[7:4],C02);
CLA_nsx clan3(A[11:8],B[11:8],C02,S[11:8],C03);
CLA_nsx clan4(A[15:12],B[15:12],C03,S[15:12],C04);

assign A19=A[18];
assign B19=B[18];

CLA cla1({A19,A[18:16]},{B19,B[18:16]},C04,S[19:16],S20);

endmodule
```

Figure 45 19-bit CLA

```
// 20-bit CLA
module CLA_20(A,B,S);

input  [19:0]A,B;
output [20:0]S;
wire   C01,C02,C03,C04;
wire   CIO = 0;

CLA_nsx clan1(A[3:0],B[3:0],CIO,S[3:0],C01);
CLA_nsx clan2(A[7:4],B[7:4],C01,S[7:4],C02);
CLA_nsx clan3(A[11:8],B[11:8],C02,S[11:8],C03);
CLA_nsx clan4(A[15:12],B[15:12],C03,S[15:12],C04);
CLA cla1(A[19:16],B[19:16],C04,S[19:16],S[20]);

endmodule
```

Figure 46 20-bit CLA

APPENDIX B

1. Radix-4 Booth's Multiplier

```
Finished circuit initialization process.
                  0 A=00000000, B=00000000, product=0000
                100 A=00000001, B=00010000, product=0010
                150 A=00010001, B=00011010, product=01ba
                200 A=00100001, B=00101011, product=058b
                250 A=00110001, B=00110010, product=0992
                300 A=10000011, B=00110000, product=e890
                350 A=10100001, B=00011010, product=f65a
                400 A=11011100, B=10011011, product=0e34
                450 A=11111011, B=11000010, product=0136
```

Figure 47 Results of functional simulation for the test-bench of Booth's multiplier

```
     0  A=00000000, B=00000000, product=xxxx
17.233  A=00000000, B=00000000, product=0000
100     A=00000001, B=00010000, product=0000
111.387 A=00000001, B=00010000, product=0010
150     A=00010001, B=00011010, product=0010
168.961 A=00010001, B=00011010, product=01ba
200     A=00100001, B=00101011, product=01ba
217.258 A=00100001, B=00101011, product=058b
250     A=00110001, B=00110010, product=058b
264.799 A=00110001, B=00110010, product=0992
300     A=10000011, B=00110000, product=0992
318.211 A=10000011, B=00110000, product=e890
350     A=10100001, B=00011010, product=e890
367.327 A=10100001, B=00011010, product=f65a
400     A=11011100, B=10011011, product=f65a
417.197 A=11011100, B=10011011, product=0e34
450     A=11111011, B=11000010, product=0e34
466.458 A=11111011, B=11000010, product=0136
```

Figure 48 Results of timing simulation for the test-bench of Booth's multiplier

2. Baugh-Wooley Array Multiplier

```
Finished circuit initialization process.
0 A=00, B=00, product=0000
100 A=01, B=10, product=0010
150 A=11, B=1a, product=01ba
200 A=21, B=2b, product=058b
250 A=31, B=32, product=0992
300 A=82, B=10, product=f820
350 A=af, B=7a, product=d966
400 A=c5, B=bb, product=0fe7
450 A=ff, B=ff, product=0001
```

Figure 49 Results of functional simulation for the test-bench of Baugh-Wooley multiplier

```
0        A=00, B=00, product=xxxx
16.918   A=00, B=00, product=0000
100      A=01, B=10, product=0000
164.274  A=11, B=1a, product=01ba
200      A=21, B=2b, product=01ba
214.141  A=21, B=2b, product=058b
250      A=31, B=32, product=058b
265.071  A=31, B=32, product=0992
300      A=82, B=10, product=0992
319.016  A=82, B=10, product=f820
350      A=af, B=7a, product=f820
366.985  A=af, B=7a, product=d966
400      A=c5, B=bb, product=d966
419.848  A=c5, B=bb, product=0fe7
450      A=ff, B=ff, product=0fe7
472.321  A=ff, B=ff, product=0001
```

Figure 50 Results of timing simulation for the test-bench of Baugh-Wooley multiplier

3. Carry-Look-Ahead Adder (CLA)

```
/*This program adds the results from the 19 multiplications between
input data and filter coefficients using CLA.
*/
`timescale 1ns/1ps
module adder_cla(M,tsum);

input [15:0]M;
output [20:0]tsum;
wire   [15:0]M1,M2,M3,M4,M5,M6,M7,M8,M9,M10;
wire   [15:0]M11,M12,M13,M14,M15,M16,M17,M18,M19;
wire   [16:0]Ra,Rb,Rc,Rd,Re,Rf,Rg,Rh,Ri;
wire   [17:0]Raa,Rbb,Rcc,Rdd,Ree;
wire   [18:0]Rff,Rgg;
wire   [19:0]Rhh;
wire   m16,r18,r19;

assign M1=M,M2=M,M3=M,M4=M,M5=M,M6=M,M7=M,M8=M,M9=M,M10=M;
assign M11=M,M12=M,M13=M,M14=M,M15=M,M16=M,M17=M,M18=M,M19=M;

CLA_16 cla16a(M1,M2,Ra);
CLA_16 cla16b(M3,M4,Rb);
CLA_16 cla16c(M5,M6,Rc);
CLA_16 cla16d(M7,M8,Rd);
CLA_16 cla16e(M9,M10,Re);
CLA_16 cla16f(M11,M12,Rf);
CLA_16 cla16g(M13,M14,Rg);
CLA_16 cla16h(M15,M16,Rh);
CLA_16 cla16i(M17,M18,Ri);

assign m16 = M19[15];
CLA_17 cla17a(Ra,Rb,Raa);
CLA_17 cla17b(Rc,Rd,Rbb);
CLA_17 cla17c(Re,Rf,Rcc);
CLA_17 cla17d(Rg,Rh,Rdd);
CLA_17 cla17e(Ri,{m16,M19},Ree);
CLA_18 cla18a(Raa,Rbb,Rff);
CLA_18 cla18b(Rcc,Rdd,Rgg);
CLA_19 cla19a(Rff,Rgg,Rhh);

assign r18 = Ree[17], r19 = Ree[17];
CLA_20 cla20a(Rhh,{r19,r18,Ree},tsum);

endmodule
```

Figure 51 Overall adder formed by CLA instantiations with only one input port

```
`timescale 1ns/1ps
module addercla_tst;

reg   [15:0]M;
wire  [20:0]tsum;

adder_cla adder(M,tsum);

initial
begin
      M = 16'h0c1d;
      #50 M = 16'h1111;
      #50 M = 16'h02aa;
      #50 M = 16'hd051;
      #50 M = 16'h0023;
end

initial $monitor($time, " M=%h, totalsum=%h", M,tsum);

endmodule
```

Figure 52 Test-bench for the overall adder with CLA instantiations and one input port

```
`timescale 1ns/1ps
module adder_cla(M1,M2,M6,M7,M11,M12,M16,M19,tsum);

input  [15:0]M1,M2,M6,M7,M11,M12,M16,M19;
output [20:0]tsum;
wire [15:0]M3,M4,M5,M8,M9,M10,M13,M14,M15,M17,M18;
wire [16:0]Ra,Rb,Rc,Rd,Re,Rf,Rg,Rh,Ri;
wire [17:0]Raa,Rbb,Rcc,Rdd,Ree;
wire [18:0]Rff,Rgg;
wire [19:0]Rhh;
wire m16,r18,r19;

assign M3 = 16'h0100;        assign M4 = 16'h1000;
assign M5 = 16'h0002;        assign M8 = 16'h2000;
assign M9 = 16'h0700;        assign M10 = 16'h0080;
assign M13 = 16'h0012;       assign M14 = 16'h0cf0;
assign M15 = 16'h0230;       assign M17 = 16'h1a00;
assign M18 = 16'h0000;

CLA_16 cla16a(M1,M2,Ra);
CLA_16 cla16b(M3,M4,Rb);
CLA_16 cla16c(M5,M6,Rc);
CLA_16 cla16d(M7,M8,Rd);
CLA_16 cla16e(M9,M10,Re);
CLA_16 cla16f(M11,M12,Rf);
CLA_16 cla16g(M13,M14,Rg);
CLA_16 cla16h(M15,M16,Rh);
CLA_16 cla16i(M17,M18,Ri);

assign m16 = M19[15];
CLA_17 cla17a(Ra,Rb,Raa);
CLA_17 cla17b(Rc,Rd,Rbb);
CLA_17 cla17c(Re,Rf,Rcc);
CLA_17 cla17d(Rg,Rh,Rdd);
CLA_17 cla17e(Ri,{m16,M19},Ree);
CLA_18 cla18a(Raa,Rbb,Rff);
CLA_18 cla18b(Rcc,Rdd,Rgg);
CLA_19 cla19a(Rff,Rgg,Rhh);

assign r18 = Ree[17], r19 = Ree[17];
CLA_20 cla20a(Rhh,{r19,r18,Ree},tsum);

endmodule
```

Figure 53 Overall adder formed by CLA instantiations with eight input ports

```
`timescale 1ns/1ps
module adder_cla_tst;

reg   [15:0]M1,M2,M6,M7,M11,M12,M16,M19;
wire  [20:0]tsum;

adder_cla adder(M1,M2,M6,M7,M11,M12,M16,M19,tsum);

initial
begin
      M1=16'h0000;M2=16'h0000;M6=16'h0000;M7=16'h0000;M11=16'h0000;
      M12=16'h0000;M16=16'h0000;M19=16'h0000;
      #50 M1=16'h1000;M2=16'h0200;M6=16'h0330;M7=16'h0efd;M11=16'h1579;
          M12=16'h00f0;M16=16'h6709;M19=16'hafff;
      #50 M1=16'h1200;M2=16'h02f0;M6=16'h1530;M7=16'h0afd;M11=16'h1009;
          M12=16'hc510;M16=16'h6009;M19=16'haf12;
end

//the arguments within $monitor system task should all be in one line
initial $monitor("#%0d, M1=%h, M2=%h, M6=%h, M7=%h, M11=%h, M12=%h, M16=%h, M19=%h, totalsum=%h",
                 $time,M1,M2,M6,M7,M11,M12,M16,M19,tsum);

endmodule
```

Figure 54 Test-bench for the overall adder with CLA instantiations and eight input ports

```
Simulator is doing circuit initialization process.
Finished circuit initialization process.
                  0 M=0c1d, totalsum=00e627
                 50 M=1111, totalsum=014443
                100 M=02aa, totalsum=00329e
                150 M=d051, totalsum=1c7603
                200 M=0023, totalsum=000299
```

Figure 55 Results of functional simulation for CLA with one input port

```
                  0 M=0c1d, totalsum=xxxxxx
                 16 M=0c1d, totalsum=00e627
                 50 M=1111, totalsum=00e627
                 65 M=1111, totalsum=014443
                100 M=02aa, totalsum=014443
                118 M=02aa, totalsum=00329e
                150 M=d051, totalsum=00329e
                166 M=d051, totalsum=1c7603
                200 M=0023, totalsum=1c7603
                214 M=0023, totalsum=000299
```

Figure 56 Results of timing simulation for CLA with one input port

```
#0, M1=0000, M2=0000, M6=0000, M7=0000, M11=0000, M12=0000, M16=0000, M19=0000, totalsum=0061b4
#50, M1=1000, M2=0200, M6=0330, M7=0efd, M11=1579, M12=00f0, M16=6709, M19=afff, totalsum=00b352
#100, M1=1200, M2=02f0, M6=1530, M7=0afd, M11=1009, M12=c510, M16=6009, M19=af12, totalsum=007b05
```

Figure 57 Results of functional simulation for CLA with eight input ports

```
#0, M1=0000, M2=0000, M6=0000, M7=0000, M11=0000, M12=0000, M16=0000, M19=0000, totalsum=xxxxxx
#26, M1=0000, M2=0000, M6=0000, M7=0000, M11=0000, M12=0000, M16=0000, M19=0000, totalsum=0061b4
#50, M1=1000, M2=0200, M6=0330, M7=0efd, M11=1579, M12=00f0, M16=6709, M19=afff, totalsum=0061b4
#70, M1=1000, M2=0200, M6=0330, M7=0efd, M11=1579, M12=00f0, M16=6709, M19=afff, totalsum=00b352
#100, M1=1200, M2=02f0, M6=1530, M7=0afd, M11=1009, M12=c510, M16=6009, M19=af12, totalsum=00b352
#122, M1=1200, M2=02f0, M6=1530, M7=0afd, M11=1009, M12=c510, M16=6009, M19=af12, totalsum=007b05
```

Figure 58 Results of timing simulation for CLA with eight input ports

4. Carry-Save Adder (CSA)

```
/*This program adds all 19 results from multiplications between
inputs and filter coefficients using CSA.
*/
`timescale 1ns/1ps
module adder_csa(M,tsum);

input [15:0]M;
output [20:0]tsum;
wire  [15:0]M1,M2,M3,M4,M5,M6,M7,M8,M9,M10;
wire  [15:0]M11,M12,M13,M14,M15,M16,M17,M18,M19;
wire  [18:0]Ra,Rb,Rc;
wire  [17:0]Rd;
wire  Rd18;

assign M1=M,M2=M,M3=M,M4=M,M5=M,M6=M,M7=M,M8=M,M9=M,M10=M;
assign M11=M,M12=M,M13=M,M14=M,M15=M,M16=M,M17=M,M18=M,M19=M;

CSA_16_5 csa16_5a(M1,M2,M3,M4,M5,Ra);
CSA_16_5 csa16_5b(M6,M7,M8,M9,M10,Rb);
CSA_16_5 csa16_5c(M11,M12,M13,M14,M15,Rc);
CSA_16 csa16a(M16,M17,M18,M19,Rd);
assign Rd18=Rd[17];
CSA_19 csa19a(Ra,Rb,Rc,{Rd18,Rd},tsum);

endmodule
```

Figure 59 Overall adder formed by CSA instantiations with only one input port

```
`timescale 1ns/1ps
module addercsa_tst;

reg   [15:0]M;
wire  [20:0]tsum;

adder_csa adder(M,tsum);

initial
begin
      M = 16'h0c1d;
      #50 M = 16'h1111;
      #50 M = 16'h02aa;
      #50 M = 16'hd051;
      #50 M = 16'h0023;
end

initial $monitor($time, " M=%h, totalsum=%h", M,tsum);

endmodule
```

Figure 60 Test-bench for the overall adder with CSA instantiations and one input port

```
`timescale 1ns/1ps
module adder_csa(M1,M2,M6,M7,M11,M12,M16,M19,tsum);

input [15:0]M1,M2,M6,M7,M11,M12,M16,M19;
output [20:0]tsum;
wire [15:0]M3,M4,M5,M8,M9,M10,M13,M14,M15,M17,M18;
wire [18:0]Ra,Rb,Rc;
wire [17:0]Rd;
wire Rd18;

assign M3 = 16'h0100;
assign M4 = 16'h1000;
assign M5 = 16'h0002;
assign M8 = 16'h2000;
assign M9 = 16'h0700;
assign M10 = 16'h0080;
assign M13 = 16'h0012;
assign M14 = 16'h0cf0;
assign M15 = 16'h0230;
assign M17 = 16'h1b00;
assign M18 = 16'h0000;

CSA_16_5 csa16_5a(M1,M2,M3,M4,M5,Ra);
CSA_16_5 csa16_5b(M6,M7,M8,M9,M10,Rb);
CSA_16_5 csa16_5c(M11,M12,M13,M14,M15,Rc);
CSA_16 csa16a(M16,M17,M18,M19,Rd);
assign Rd18=Rd[17];
CSA_19 csa19a(Ra,Rb,Rc,{Rd18,Rd},tsum);

endmodule
```

Figure 61 Overall adder formed by CSA instantiations with eight input ports

```
`timescale 1ns/1ps
module adder_csa_tst;

reg   [15:0]M1,M2,M6,M7,M11,M12,M16,M19;
wire  [20:0]tsum;

adder_csa adder(M1,M2,M6,M7,M11,M12,M16,M19,tsum);

initial
begin
      M1=16'h0000;M2=16'h0000;M6=16'h0000;M7=16'h0000;M11=16'h0000;
      M12=16'h0000;M16=16'h0000;M19=16'h0000;
      #50 M1=16'h1000;M2=16'h0200;M6=16'h0330;M7=16'h0efd;M11=16'h1579;
            M12=16'h00f0;M16=16'h6709;M19=16'hafff;
      #50 M1=16'h1200;M2=16'h02f0;M6=16'h1530;M7=16'h0afd;M11=16'h1009;
            M12=16'hc510;M16=16'h6009;M19=16'haf12;
end

//the arguments within $monitor system task should all be in one line
initial $monitor("#%0d, M1=%h, M2=%h, M6=%h, M7=%h, M11=%h, M12=%h, M16=%h,
                 M19=%h, totalsum=%h", $time,M1,M2,M6,M7,M11,M12,M16,M19,tsum);

endmodule
```

Figure 62 Test-bench for the overall adder with CSA instantiations and eight input ports

```
Finished circuit initialization process.
                  0 M=0c1d, totalsum=00e627
                 50 M=1111, totalsum=014443
                100 M=02aa, totalsum=00329e
                150 M=d051, totalsum=1c7603
                200 M=0023, totalsum=000299
```

Figure 63 Results of functional simulation for CSA with one input port

```
  0 M=0c1d, totalsum=xxxxxx
 17 M=0c1d, totalsum=00e627
 67 M=1111, totalsum=014443
100 M=02aa, totalsum=014443
114 M=02aa, totalsum=00329e
150 M=d051, totalsum=00329e
167 M=d051, totalsum=1c7603
200 M=0023, totalsum=1c7603
212 M=0023, totalsum=000299
```

Figure 64 Results of timing simulation for CSA with one input port

```
#0, M1=0000, M2=0000, M6=0000, M7=0000, M11=0000, M12=0000, M16=0000, M19=0000, totalsum=0061b4
#50, M1=1000, M2=0200, M6=0330, M7=0efd, M11=1579, M12=00f0, M16=6709, M19=afff, totalsum=00b352
#100, M1=1200, M2=02f0, M6=1530, M7=0afd, M11=1009, M12=c510, M16=6009, M19=af12, totalsum=007b05
```

Figure 65 Results of functional simulation for CSA with eight input ports

```
#0, M1=0000, M2=0000, M6=0000, M7=0000, M11=0000, M12=0000, M16=0000, M19=0000, totalsum=xxxxxx
#23, M1=0000, M2=0000, M6=0000, M7=0000, M11=0000, M12=0000, M16=0000, M19=0000, totalsum=0061b4
#50, M1=1000, M2=0200, M6=0330, M7=0efd, M11=1579, M12=00f0, M16=6709, M19=afff, totalsum=0061b4
#70, M1=1000, M2=0200, M6=0330, M7=0efd, M11=1579, M12=00f0, M16=6709, M19=afff, totalsum=00b352
#100, M1=1200, M2=02f0, M6=1530, M7=0afd, M11=1009, M12=c510, M16=6009, M19=af12, totalsum=00b352
#122, M1=1200, M2=02f0, M6=1530, M7=0afd, M11=1009, M12=c510, M16=6009, M19=af12, totalsum=007b05
```

Figure 66 Results of timing simulation for CSA with eight input ports