ANALYSIS AND IMPLEMENTATION OF VARIOUS ADDERS AND MULTIPLIERS WITH POWER & PERFORMANCE PERSPECTIVES

Thesis report submitted in the partial fulfilment of the requirement for the award of the degree of

MASTER OF TECHNOLOGY

IN

VLSI DESIGN & CAD

Submitted by

Puneet Bhardwaj Dhammi

Roll No.: 600961017

Under the Guidance of

Ms. Harpreet Vohra

Assistant Professor, ECED

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

THAPAR UNIVERSITY, PATIALA (PUNJAB) - 147004

JULY – 2011

DECLARATION

I hereby declare that the work which is being presented in the thesis entitled, **"Analysis and Implementation of MAC Unit with Power & Performance Perspectives"** in partial fulfilment of the requirement for the award of degree of M.Tech (VLSI Design & CAD) at Electronics and Communication Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Ms. Harpreet Vohra,** Assistant Professor, ECED.

The matter presented in this report has not been submitted in any other University/Institute for the award of my degree.

Date: 15th July, 2011

PUNEET DHAMMI
Roll.No. 600961017

It is certified that the above statement made by the student is correct to the best of my knowledge and belief.

Ms. HARPREET VOHRA
Assistant Professor ECED,
Thapar University,
Patiala -147004

Counter Signed by:

(Dr. A. K. Chatterjee)
Professor & Head, ECED
Thapar University, Patiala

(Dr. S. K. Mohapatra)
Dean of Academics Affairs
Thapar University, Patiala

ACKNOWLEDGEMENT

To discover, analyse and to present something new is to venture on an untraded path towards and unexplored destination is an arduous adventure unless one gets a true torchbearer to show the way. I would have never succeeded in completing my task without the cooperation, encouragement and help provided to me by various people. Words are often too less to reveals one's deep regards. I take this opportunity to express my profound sense of gratitude and respect to all those who helped me through the duration of this thesis. I acknowledge with gratitude and humility my indebtedness to **Ms. Harpreet Vohra, Assistant Professor, ECED,** Thapar University, Patiala under whose guidance I had the privilege to complete this thesis. I wish to express my deep gratitude towards her for providing individual guidance and support throughout the thesis work.

I convey my sincere thanks to **Dr. A. K. Chatterjee, Professor & Head of Electronics & Communication Department,** Thapar University, Patiala for his encouragement and cooperation.

I express my heartfelt gratitude towards **Dr. Alpana Agarwal, Assistant Professor & PG coordinator, ECED,** Thapar University, Patiala, for her valuable guidance, encouragement, inspiration and the enthusiasm with which she solved my difficulties in VLSI Design & CAD Lab.

I would also like to thank all my friends Amandeep, Simran, Aditya, Harpreet and all staff members who were always there at the need of the hour and provided with all the help and facilities, which I required for the completion of my thesis.

My greatest thanks are to all who wished me success especially my parents. Above all I render my gratitude to the Almighty who bestowed self-confidence, ability and strength in me to complete this work for not letting me down at the time of crisis and showing me the silver lining in the dark clouds.

PUNEET DHAMMI

ABSTRACT

The goal of this thesis is to analyse and compare various adder and multiplication schemes for high-speed and low power operations. Since the various filter designs found in the Digital Signal Processing applications, require computationally efficient Multiply and Accumulate operations so the blocks with the desired characteristics have to be chosen carefully. Various techniques have been proposed to design multipliers which are efficient in terms of performance, low power consumption and area.

The multiplier is a fairly large block of a computing system. The amount of circuitry involved is directly proportional to the square of its resolution i.e. a multiplier of size n bits has O (n^2) gates. For multiplication algorithms performed in DSP applications latency and throughput are the two major concerns from delay perspective. Latency is the real delay of computing a function, a measure of how long the inputs to a device are stable is the final result available on outputs. Throughput is the measure of how many multiplications can be performed in a given period of time; multiplier is not only a high delay block but also a major source of power dissipation. That's why if one also aims to minimize power consumption, it is of great interest to reduce the delay by using various delay optimizations.

There have been many algorithms proposals in literature to perform accumulation-multiplication, each offering different advantages and having trade-offs in terms of speed, circuit complexity, area and power consumption. In this thesis, a new architecture for a high-speed MAC is designed in which, the computations of multiplication and accumulation are combined and a hybrid-type CSA structure is used to reduce the critical path and improve the output rate. It uses Modified Booth's Algorithm based on 1's complement number system. A modified array structure for the sign bits is used to increase the density of the operands. A carry look-ahead adder (CLA) is inserted in the CSA tree to reduce the number of bits in the final adder.

It has been found that Booth Wallace multiplier is most efficient among all, giving optimum delay, power and area for multiplication. Low power modified Booth recoder and pipelining techniques have been used to reduce power and delay.

CONTENTS

LIST OF FIGURES

FIG. NO.	TITLE OF FIGURE	PAGE NO.

LIST OF TABLES

TERMINOLOGY

DSP	Digital Processing System
MAC	Multiply and Accumulate
FFT	Fast Fourier Transform
LSD	Least Significant Digit
LSB	Least Significant Bit
CSA	Carry Save Adder
CLA	Carry Look-ahead Adder
PP	Partial Product
FA	Full Adder
BE	Booth Encoder
MBR	Modified Booth Recoder
MBA	Modified Booth Algorithm
BS	Booth Selector
WT	Wallace Tree
LCD	Liquid Crystal Display
CLC	Carry look-Ahead Circuit
OC	Output Carry
FPGA	Field Programmable Gate Array
NGD	Native Generic Database
CLBs	Configurable Logic Blocks
LUT	Look Up Table

Chapter 1
INTRODUCTION

1.1 Motivation

In the past few decades ago, the electronics industry has been experiencing an unprecedented spurt in growth, thanks to the use of integrated circuits in computing, telecommunications and consumer electronics. We have come a long way from the single transistor era in 1958 to the present day ULSI (Ultra Large Scale Integration) systems with more than 50 million transistors in a single chip [1].

As the performance of processors has increased, the demand for high speed arithmetic blocks has also increased. With clock frequencies approaching 1 GHz, arithmetic blocks must keep pace with the continued demand for more computational power. The purpose of this thesis is to present methods of implementing high speed binary multiplication. In general, both the algorithms used to perform multiplication, and the actual implementation procedures are addressed. The emphasis of this thesis is on minimizing the latency, with the goal being the implementation of the fastest multiplication blocks possible.

1.2 NEED FOR LOW POWER DESIGN

There are various interpretations of the Moore's Law that predicts the growth rate of integrated circuits. One estimate places the rate at 2X for every eighteen months. Others claim that the device density increases ten-fold every seven years. Regardless of the exact numbers, everyone agrees that the growth rate is rapid with no signs of slowing down. New generations of processing technology are being developed while present generation devices are at very safe distance from the fundamental physical limits. A need for low power VLSI chips arises from such evolution forces of integrated circuits. The Intel 4004 microprocessor, developed in 1971, had 2300 transistors, dissipated about 1 watts of power and clocked at 1 MHz. Then come the Pentium in 2001, with 42 million transistors, dissipating around 65 watts of power and clocked at 2.40 GHz [1].

While the power dissipation increases linearly as the years go by, the power density increases exponentially, because of the ever-shrinking size of the integrated

circuits. If this exponential rise in the power density were to increase continuously, a microprocessor designed a few years later, would have the same power as that of the nuclear reactor. Such high power density introduces reliability concerns such as, electro-migration, thermal stresses and hot carrier induced device degradation, resulting in the loss of performance.

Another factor that fuels the need for low power chips is the increased market demand for portable consumer electronics powered by batteries. The craving for smaller, lighter and more durable electronic products indirectly translates to low power requirements. Battery life is becoming a product differentiator in many portable systems. Being the heaviest and biggest component in many portable systems, batteries have not experienced the similar rapid density growth compared to the electronic circuits. The main source of power dissipation in these high performance battery-portable digital systems running on batteries such as note-book computers, cellular phones and personal digital assistants are gaining prominence. For these systems, low power consumption is a prime concern, because it directly affects the performance by having effects on battery longevity. In this situation, low power VLSI design has assumed great importance as an active and rapidly developing field.

At the circuit design level, considerable potential for power savings exists by means of proper choice of a logic style for implementing combinational circuits. This is because all the important parameters governing power dissipation switching capacitance, transition activity, and short circuit currents are strongly influenced by the chosen logic style.

1.3 Logic Style Requirements for Low Power

According to the formula:

$$P_{dyn} = V^2_{dd} \cdot f_{clk} \cdot \sum_n \alpha_n \cdot c_n + V_{dd} \cdot \sum_n i_{sc_n} \qquad 1.1$$

The dynamic power dissipation of a digital CMOS circuit depends on the supply voltage $V_{dd,}$ the clock frequency f_{clk} the node switching activities α_n, node capacitances c_n, the node short- circuit currents i_{sc_n}, and the number of nodes n. A reduction of each of these parameters results in a reduction of dissipated power. However, clock frequency f_{clk} reduction is only feasible at the architecture level, whereas at the circuit level frequency is usually regarded as constant in order to fulfil some given

throughput requirement. All the other parameters are influenced to some degree by the logic style applied. Thus, some general logic style requirements for low-power circuit implementation can be stated at this point.

1.3.1 Switched capacitance reduction

Capacitive load, originating from transistor capacitances (gate and diffusion) and inter- connect wiring, is to be minimized. This is achieved by having as few transistors and circuit nodes as possible, and by reducing transistor sizes to a minimum. In particular, the number of (high- capacitive) inter-cell connection and their length (by the circuit size) should be kept minimal. Another source for capacitance reduction is found at the layout level. Transistor downsizing is an effective way to reduce switched capacitance of logic gates on noncritical signal paths. For that purpose, a logic style should be robust against transistor downsizing, i.e., correct functioning of logic gates with minimal or near minimal transistor sizes must be guaranteed.

1.3.2 Supply voltage reduction

The supply voltage and the choice of logic style are indirectly related through delay-driven voltage scaling. That is, a logic style providing fast logic gates to speed up critical signal paths allows a reduction of the supply voltage in order to achieve a given throughput. For that purpose, a logic style must be robust against supply voltage reduction, i.e., performance and correct functioning of gates must be guaranteed at low voltages as well. This becomes a severe problem at very low voltages of around 1 V and lower, where noise margins become critical.

1.3.3 Switching activity reduction

Switching activity of a circuit is predominantly controlled at the architectural and register transfer level (RTL). At the circuit level, large differences are primarily observed between static and dynamic logic styles. On the other hand, only minor transition activity variations are observed among different static logic styles and among logic gates of different complexity, also if glitching is concerned.

1.3.4 Short-circuit current reduction

Short-circuit currents (also called dynamic leakage currents or overlap currents) may vary by a considerable amount between different logic styles. They also strongly depend on input signal slopes (i.e., steep and balanced signal slopes are better) and thus on transistor sizing. Their contribution to the overall power consumption is rather limited but still not negligible (10–30%), except for very low voltages$V_{dd} \leq V_{tn} + |V_{tp}|$, where the short-circuit currents disappear. A low-power logic style should have minimal short-circuit currents and, of course, no static currents besides the inherent CMOS leakage currents [2].

1.4 Organization

This report is organized as follows:

CHAPTER I: INTRODUCTION − This chapter introduces power consumption issues in the area of VLSI. This chapter also summarizes the need of low power design in the today's era of scaling down of technologies and nanotechnology. Finally, this thesis chapter explains organization of the thesis.

CHAPTER II: LITERATURE SURVEY− This chapter explains different algorithms of adders, multipliers and MAC unit and finally the optimized one's are selected on the basis of performance measures in designing high performance MAC Unit.

CHAPTER III: MAC UNIT DESIGN AND IMPLEMENTATION – This chapter explains the design of MAC architecture by using optimized adder and multiplication schemes.

CHAPTER IV: ANALYSIS AND IMPLEMENTATION – This chapter provides the analysis and implementation of various adders, multipliers and optimized MAC Unit on performance perspectives.

CHAPTER V: CONCLUSION- This chapter finally concludes that which adder and multiplier are best suited to our application and is also tabulated on various constraints.

1.5 Tools Used

The tools used in the thesis are as follows:

Simulation Software:

- Xilinx 9.2i and Design Complier are used for synthesis and analysis.
- ModelSim 6.1e is used for modelling and simulation.

Hardware Used:

Xilinx Spartan 3E (Family), XC3S500E (Device), FG320 (Package) FPGA device.

Chapter 2

LITERATURE SURVEY

2.1 Introduction to Adder

In nearly all digital IC designs today, the addition operation is one of the most essential and frequent operations. Instruction sets for DSP's and general purpose processors include at least one type of addition. Other instructions such as subtraction and multiplication employ addition in their operations, and their underlying hardware is similar if not identical to addition hardware. Often, an adder or multiple adders will be in the critical path of the design, hence the performance of a design will be often be limited by the performance of its adders. When looking at other attributes of a chip, such as area or power, the designer will find that the hardware for addition will be a large contributor to these areas. It is therefore beneficial to choose the correct adder to implement in a design because of the many factors it aspects in the overall chip. In this chapter we begin with the basic building blocks used for addition, then go through different algorithms and name their advantages and disadvantages.

2.2 Basic Adder Blocks

2.2.1 Half Adder

The Half Adder (HA) is the most basic adder. It takes in two bits of the same weight, and creates a sum and a carryout. If the two inputs a and b have a weight of 2^i (where i is an integer), sum has a weight of 2^i, and carryout has a weight of $2^{(i+1)}$. Equations 2.1 and 2.2 are the Boolean equations for sum and carryout, respectively.

$$\text{sum} = a \oplus b \tag{2.1}$$

$$\text{carry} = a.b \tag{2.2}$$

2.2.2 Full Adder

The Full Adder (FA) is useful for additions that have multiple bits in each of its operands. It takes in three inputs and creates two outputs, a sum and a carryout. The inputs have the same weight, 2^i, the sum output has a weight of 2^i, and the carryout output has a weight of $2^{(i+1)}$.The FA differs from the HA in that it has a carryin as one of its inputs, allowing for the cascading of this structure which is explored below in

Section 2.3.1. Equations 2.3 and 2.4 are the Boolean equations for the FA *sum* and FA *carryout*, respectively. In both those equations cin means carryin.

$$\text{sum}_i = a \oplus b \oplus \text{cin}_i \quad 2.3$$

$$\text{carry}_i = a_i.b_i + b_i.\text{cin}_i + a_i.\text{cin}_i \quad 2.4$$

2.2.3 Partial Full Adder

The Partial Full Adder (PFA) is a structure that implements intermediate signals that can be used in the calculation of the carry bit. It is an extension of FA which include the signals generate (g), kill (k), and propagate (p).When g=1, it means carryout will be 1 (generated) regardless of carryin. When k=1, it means carryout will be 0 (killed) regardless of carryin. When p=1, it means carryout will equal carryin (carryin will be propagated). Table 2.1 reflects these three additional signals, with a comment on the carryout bit in an additional column. Equations 2.5 − 2.7 are the Boolean equations for generate, kill, and propagate, respectively. It should be noted that for the propagate signal, the XOR function can also be used, since in the case of a, b=1, the generate signal will assert that carryout is 1. The Boolean equations for the sum and carryout can now be written as functions of g, p, or k shown by Equations 2.8 and 2.9. Figure 2.1 shows a circuit for creating the generate, propagate, and sum signals. It is a partial full adder because it does not calculate the carryout signal directly; rather, it creates the signals needed to calculate the carryout signal.

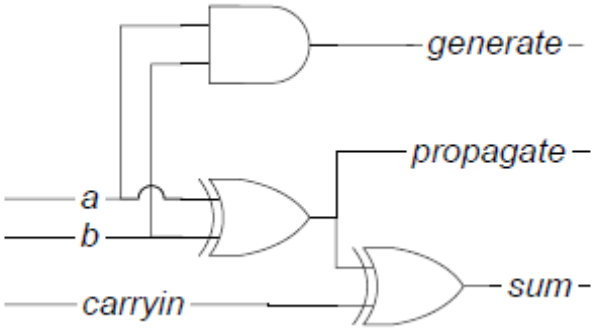


Figure 2.1: Gate Schematic for a Partial Full Adder (PFA)

$$\text{generate}_i(g_i) = a_i.b_i \quad 2.5$$

$$\text{kill}_i(k_i) = \bar{a_i}.\bar{b_i} \quad 2.6$$

$$\text{propagate}_i(p_i) = a_i + b_i = a_i \oplus b_i \quad 2.7$$

$$sum_i = p_i \oplus cin_i \quad (2.8)$$

$$carryout_{i+1} = a_i.b_i + b_i.carryin_i + a_i.carryin_i \quad (2.9)$$

Table 2.1: Extended Truth Table for a 1-bit adder

Inputs			**Outputs**					
carryin	**a**	**b**	**carryout**	**sum**	**g**	**k**	**p**	**Carry Status**
0	0	0	0	0	0	1	0	delete
0	0	1	0	1	0	0	1	propagate
0	1	0	0	1	0	0	1	propagate
0	1	1	1	0	1	0	1	generate/propagate
1	0	0	0	1	0	1	0	delete
1	0	1	1	0	0	0	1	propagate
1	1	0	1	0	0	0	1	propagate
1	1	1	1	1	1	0	1	generate/propagate

2.3 Adder Algorithms

2.3.1 Ripple Carry Adder

The Ripple Carry Adder (RCA) is one of the simplest adders to implement. This adder takes in two N-bit inputs (where N is a positive integer) and produces (N + 1) output bits (an N-bit sum and a 1-bit carryout). The RCA is built from N full adders cascaded together, with the carryout bit of one FA tied to the carryin bit of the next FA. Figure 2.2 shows the schematic for an N-bit RCA. The input operands are labeled a and b, the carryout of each FA is labeled cout (which is equivalent to the carryin (cin) of the subsequent FA), and the sum bits are labelled sum. Each sum bit requires both input operands and cin before it can be calculated. To estimate the propagation delay of this adder, we should look at the worst case delay over every possible combination of inputs. This is also known as the critical path. The most significant sum bit can only be calculated when the carryout of the previous FA is known. In the worst case (when all the carryout's are 1), this carry bit needs to ripple across the structure from the least significant position to the most significant position. Figure 2.3 has a darkened line indicating the critical path.

Hence, the time for this implementation of the adder is expressed in Equation 2.10, where $t_{RCAcarry}$ is the delay for the carryout of a FA and t_{RCAsum} is the delay for the sum of a FA.

$$\text{Propagation Delay}(t_{RCAprop}) = (N-1).t_{RCAcarry} + t_{RCAsum} \qquad 2.10$$

From Equation 2.10, we can see that the delay is proportional to the length of the adder. An example of a worst case propagation delay input pattern for a 4 bit ripple carry adder is where the input operands change from 1111 and 0000 to 1111 and 0001, resulting in a sum changing from 01111 to 10000.

From a VLSI design perspective, this is the easiest adder to implement. One just needs to design and lay out one FA cell, and then array N of these cells to create an N-bit RCA. The performance of the one FA cell will largely determine the speed of the whole RCA. From the critical path in Equation 2.10, minimizing the carryout delay ($t_{RCAcarry}$) of the FA will minimize $t_{RCAprop}$. There are various implementations of the FA cell to minimize the carryout delay [17].

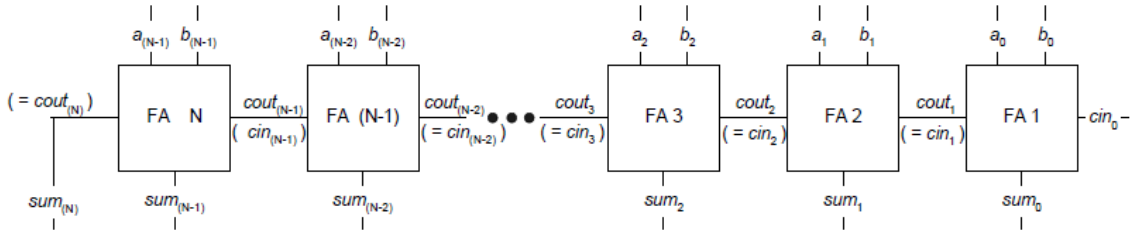


Figure 2.2: Schematic for an N-bit Ripple Carry Adder

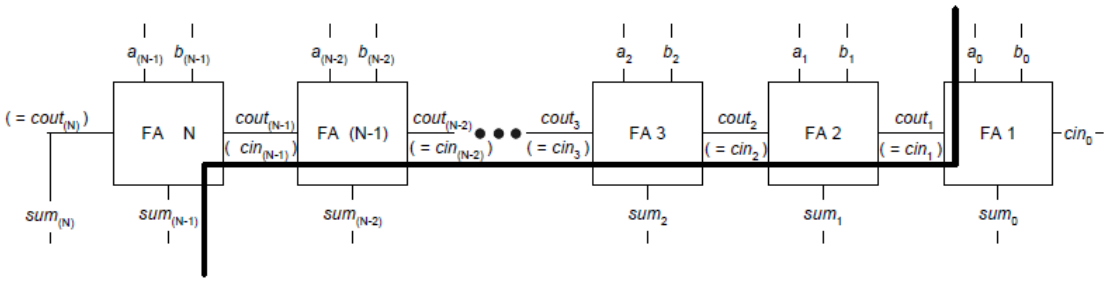


Figure 2.3: Critical path for an N-bit Ripple Carry Adder

2.3.2 Carry Skip Adder:

From examination of the RCA, the limiting factor for speed in that adder is the propagation of the cout bit. The Carry Skip Adder (CSKA, also known as the Carry Bypass Adder) addresses this issue by looking at groups of bits and determines whether

this group has a carryout or not [18]. This is accomplished by creating a group propagate signal ($p_{CSKAgroup}$) to determine whether the group carryin ($carryin_{CSKAgroup}$) will propagate across the group to the carryout ($carryout_{CSKAgroup}$). To explore the operation of the whole CSKA, take an N-bit adder and divide it into N/M groups, where M is the number of bits per group. Each group contains a 2-to-1 multiplexer, logic to calculate M sum bits, and logic to calculate $p_{CSKAgroup}$. The select line for the mux is simply the $p_{CSKAgroup}$ signal, and it chooses between $carryin_{CSKAgroup}$ or $cout_4$.

To aid the explanation, we refer the reader to Figure 2.4, which shows the hardware for a group of 4 bits (M=4) in the CSKA. There are four full adders cascaded together and each FA creates a carryout (cout), a propagate (p) signal, and a sum (sum not shown). The propagate signal from each FA comes at no extra hardware cost since it is calculated in the sum logic (the hardware is identical to the sum hardware for the PFA shown in Figure 2.1). For the $carryout_{CSKAgroup}$ to equal $carryin_{CSKAgroup}$, all of the individual propagates must be asserted (Equations 2.11 and 2.12). If this is true then $carryin_{CSKAgroup}$ skips" past the group of full adders and equals the $carryout_{CSKAgroup}$. For the case where $p_{CSKAgroup}$ is 0, at least one of the propagate signals is 0. This implies that either a delete and/or generate occurred in the group. A delete signal simply means that the carryout for the group is 0 regardless of the carryin, and a generate signal means that the carryout is 1 regardless of the carryin. This is advantageous because it implies that the carryout for the group is not dependent on the carryin. No hardware is needed to implement these two signals because the group carryout signal will reflect one of the three cases (a d, g or group p occurred). The additional hardware to realize the group carryout in Figure 2.4 is accomplished with a 4-input AND gate and a 2-to-1 multiplexer (mux). In general, an M-input AND gate and a 2-to-1 mux are required for a group of bits, including the logic to calculate the sum bits.

$$p_{CSKAgroup} = p_0 . p_1 . p_2 . p_3 \quad (2.11)$$

$$carryout_{CSKAgroup} = carryin_{CSKAgroup} . P_{CSKAgroup} \quad (2.12)$$

In examining the critical path for the CSKA, we are primarily concerned whether the carryin can be propagated (“skipped") across a group or not. Assuming all input bits come into the adder at the same time, each group can calculate the group propagate signal (mux select line) simultaneously. Every mux then knows which signal to pass as the carryout of the group. There are two cases to consider after the mux select line has been determined. In the first case, $carryin_{CSKAgroup}$ will propagate to the carryout. This means

$p_{CSKAgroup}$=1 and the carryout is dependent on the carryin. In the second case, the carryout signal of the most significant adder will become the group carryout. This means $p_{CSKAgroup}$=0 and the carryout is independent of the carryin. If we isolate a particular group (as in Figure 2.4), the second case (signal $cout_4$) always takes longer because the carryout signal must be calculated through logic, whereas the first case ($carryin_{CSKAgroup}$) requires only a wire to propagate the signal. Looking at the whole architecture, however, this second case is part of the critical path for only the first CSKA group. Since the second case is not dependent on the group carryin, all the groups in the CSKA can compute the carryout in parallel. If a group needs its carryin ($p_{CSKAgroup}$=1), then it must wait until it arrives after being calculated from a previous group. In the worst case, a carryout must be calculated in the first group, and every group afterwards needs to propagate this carryout. When the final group receives this propagated signal, then it can calculate its sum bits. Figure 2.5 shows a 16-bit CSKA with 4-bit groups and Figure 2.6 shows a darkened line indicating the critical path of the signals in the 16-bit CSKA.

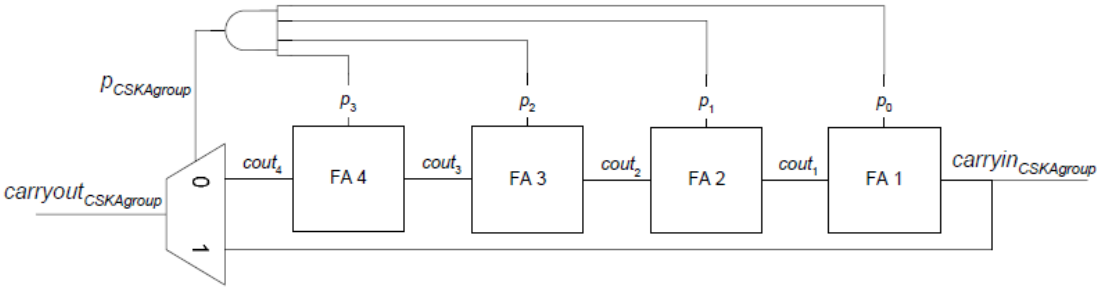


Figure 2.4: One group in a Carry Skip Adder, in this case M=4

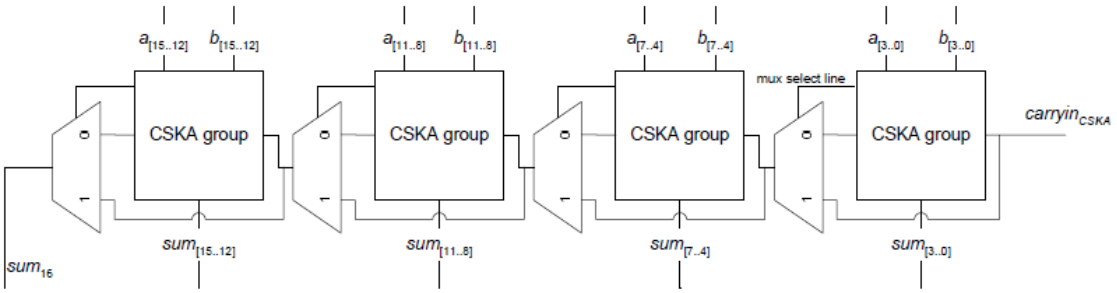


Figure 2.5: A 16-bit Carry Skip Adder N=16, M=4

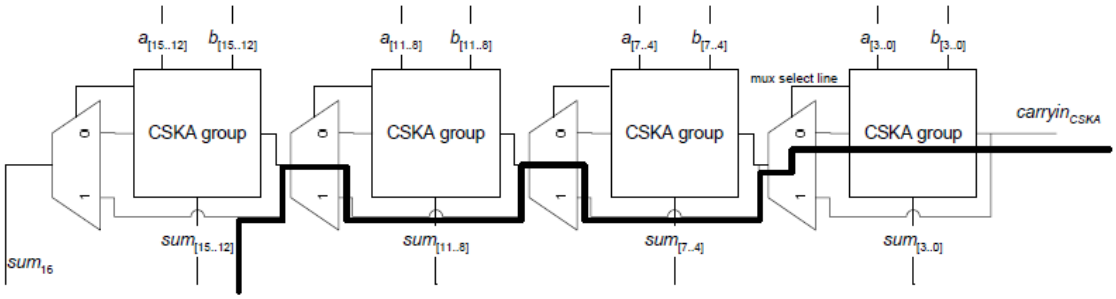


Figure 2.6: Critical path through 16-bit CSKA

If we assume a 16-bit CSKA with 4-bit groups, with each group containing a 4-bit RCA for the sum logic, then the worst case propagation delay through this adder is expressed in equation 2.13. In this equation, $t_{RCAcarry}$ and t_{RCAsum} are the delays to calculate the carryout and sum signals of an RCA, respectively. Each group has 4 bits, so the delay through the first group has 4 RCA carryout delays. This carryout of the first group potentially propagates through 3 muxes, where one mux delay is expressed as $t_{muxdelay}$. Finally, when the carryout signal reaches the final group, the sum for this group can be calculated. This is represented by the final two components of Equation 2.4.

$$t_{CSKA16} = 4 * t_{RCAcarry} + 3 * t_{muxdelay} + 3 * t_{RCAcarry} + t_{RCAsum} \quad 2.13$$

For Equation 2.13, there are some assumptions about the delay through the circuit. First, we assume in the first CSKA group that the group propagate signal is calculated before the carryout of the most significant adder. Thus, the mux for this first group is waiting for the carryout. For the final CSKA group, we assume that it takes longer for sum15 to be calculated than for sum16 to be calculated. Once the carryin for this last group is known, the delay for sum16 is the delay of the mux; for sum_{15} it is a delay of $3*t_{RCAcarry} + t_{RCAsum}$ (3 ripples through the adder before the last sum bit can be calculated). For an N-bit CSKA, the critical path equation is expressed in Equation 2.5. M represents the number of bits in each group. There are N/M groups in the adder, and every mux in this group except for the last one is in the critical path. As in Equation 2.13, Equation 2.14 assumes that each group contains a ripple carry adder.

$$t_{CSKAN_N} = M * t_{RCA_{carry}} + \left(\frac{N}{M} - 1\right) t_{mux_{delay}} + (M-1) * t_{RCA_{carry}} + t_{RCA_{sum}} \quad 2.14$$

From a VLSI design perspective, this adder shows improved speedup over a RCA without much area increase. The additional hardware comes from the 2-to-1 mux and group propagate logic in each group, which is about 15% more area. One drawback to this structure is that its delay is still linearly dependent on the width of the adder,

therefore for large adders where speed is important, the delay may be unacceptable. Also, there is a long wire in between the groups that $carryout_{CSKAgroup}$ needs to travel on. This path begins at the carryout of the first CSKA group and ends at the carryin to the final CSKA group. This signal also needs to travel through $\left(\frac{N}{m}-1\right)$ muxes, and these will introduce long delays and signal degradation if pass gate muxes are used. If buffers are required in between these groups to reproduce the signal, then the critical path is lengthened. An example of a worst case delay input pattern for a 16-bit CSKA with 4-bit groups is where the input operands are 1111111111111000 and 0000000000001000. This forces a carryout in the first group that skips through the middle two groups and enters the final group. This carryin to the final group ripples through to the final sum bit (sum_{15}). To determine the optimal speed for this adder, one needs to find the delay through a mux and the carryout delay of a FA. It is one of these two delays that will dominate the delay of the whole CSKA. For short adders ($\leq$ 16 bits), the $t_{carryout}$ of a FA will probably dominate delay, and for long adders the long wire that skips through stages and muxes will probably dominate the delay.

2.3.3 Carry Look Ahead Adder

From the critical path equations in Sections 2.2.1 and 2.2.2, the delay is linearly dependent on N, the length of the adder. It is also shown in Equations 2.10 and 2.14 that the $t_{carryout}$ signal contributes largely to the delay. An algorithm that reduces the time to calculate $t_{carryout}$ and the linear dependency on N can greatly speed up the addition operation. Equation 2.9 shows that the carryout can be calculated with g, p, and carryin. The signals g and p are not dependent on carryin, and can be calculated as soon as the two input operands arrive. Weinberger and Smith invented the Carry Look Ahead (CLA) Adder [19]. Using Equation 2.9, we can write the carryout equations for a 4-bit adder. These equations are shown in Equations 2.15−2.18, where c_i represents the carryout of the i^{th} position ($0 \leq i \leq (N-1)$), and g_i, p_i represent the generate and propagate signal from each PFA). The equations for c_2, c_3 and c_4 are obtained by substitution of c_1, c_2 and c_3, respectively. These equations show that every carryout in the adder can be determined with just the input operands and initial carryin (c_3). This process of calculating c_i by using only the p_i, g_i and c_0 signals can be done indefinitely, however, each subsequent carryout generated in this manner becomes increasingly difficult because of the large number of high fan-in gates [20].

$$c_1 = g_0 + p_0.c_0 \quad (2.15)$$

$$c_2 = g_1 + p_1.c_1 = g_1 + p_1.g_0 + p_1.p_0.c_0 \quad (2.16)$$

$$c_3 = g_2 + p_2.c_2 = p_2.g_1 + p_2.p_1.g_0 + p_2.p_1.p_0.c_0 \quad (2.17)$$

$$c_4 = g_3 + p_3.c_3$$

$$= g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.g_0 + p_3.p_2.p_1.p_0.c_0 \quad (2.18)$$

The CLA adder uses partial full adders as described in Section 2.1.3 to calculate the generate and propagate signals needed for the carryout equations. Figure 2.6 shows the schematic for a 4-bit CLA adder. The logic for each PFA block is shown in Figure 2.3. The CLA logic block implements the logic in Equations 2.15−2.18, and the gate schematic for this block is in Figure 2.7. For a 4-bit CLA adder the 4th carryout signal can also be considered as the 5th sum bit.

Although it is impractical to have a single level of carry look-ahead logic for long adders, this can be solved by adding another level of carry look-ahead logic. To achieve this, each adder block requires two additional signals: a group generate and a group propagate. The equations for these two signals, assuming adder block sizes of 4 bits, are shown in Equations 2.19 and 2.20. A group generate occurs if a carry is generated in one of adder blocks, and a group propagate occurs if the carryin to the adder block will be propagated to the carryout. Figure 2.11 shows the gate schematic of the two additional signals.

$$\text{group generate} = g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.c_3 \quad (2.19)$$

$$\text{group propagate} = g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.c_3 \quad (2.20)$$

With multiple levels of CLA logic, carry look-ahead adders of any length can be built. The size of an adder block in a CLA adder is usually 4 bits because it is a common factor of most word sizes and there is a practical limit on the gate size that can be implemented [20]. To illustrate the use of another level of CLA logic, Figure 2.8 shows the schematic for a 16-bit CLA adder. There is a second level of CLA logic which takes the group generate and group propagate signals from each 4-bit adder subcell and calculates the carryout signals for each adder block. If an adder has multiple levels of CLA logic, only the final level needs to generate the c_4 signal. All other levels replace this c4 signal with the group generate and group propagate. The CLA logic for this 16-bit adder is identical to the CLA logic for the 4-bit adder in Figure 2.7; therefore the equations for the carryout signals are in Equations 2.15−2.18.

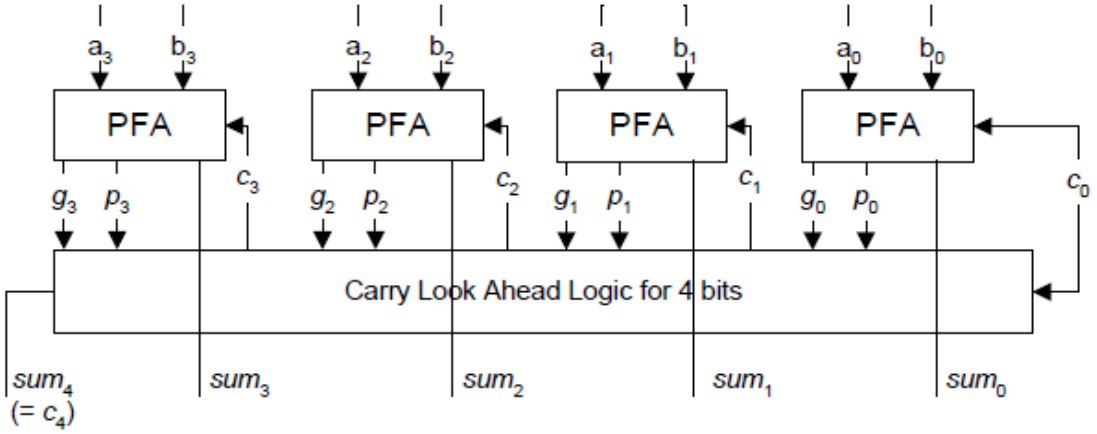


Figure 2.7: 4-bit carry look-ahead adder

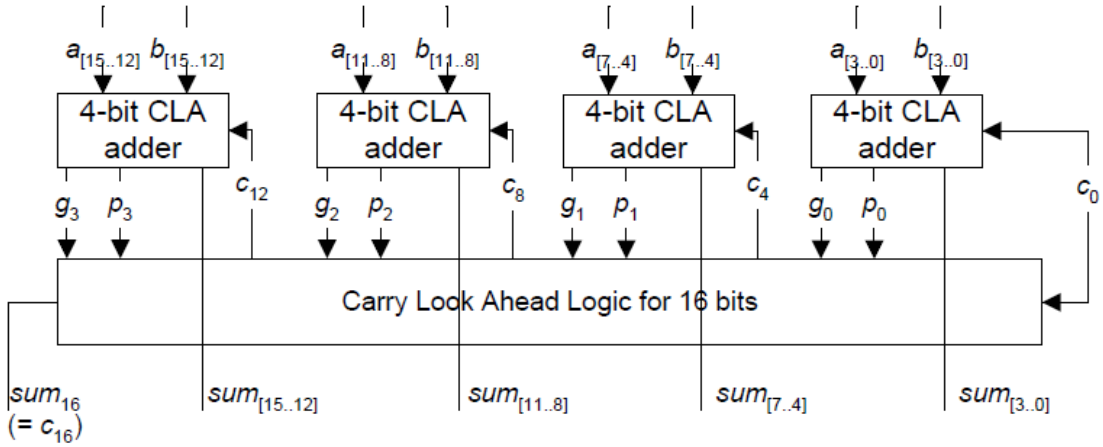


Figure 2.8: Schematic for a 16-bit CLA adder

A third level of CLA logic and four 16-bit adder blocks can be used to build a 64-bit adder. The CLA logic would create the c_{16}, c_{32}, and c_{48} signals to be used as carryins to the 16-bit adder blocks and the c_{64} as the sum_{64} signal. If a design calls for an adder of length 32, a designer can simply use two 16-bit adder blocks and the first two carryout signals (c_{16}, c_{32}) from the third level of CLA logic. The identical hardware in the CLA logic, coupled with the fact that the adder blocks can be instantiated as subcells, makes building long adders with this architecture simple.

Determining the critical path for a CLA adder is difficult because the gates in the carry path have different fan-in. To get a general idea, we first assume that all gate delays are the same. The delay for a 4-bit CLA adder then requires one gate delay to calculate the propagate and generate signals, two gate delays to calculate carry signals, and one

gate delay to calculate the sum signals; this equates to four gate delays. For a 16-bit CLA adder there is one gate delay to calculate the propagate and generate signal (from the PFA), two gate delays to calculate the group propagate and generate in the first level of carry logic, two gate delays for the carryout signals in the second level of carry logic, and one gate delay for the sum signals. The second level of carry logic for the 16-bit CLA adder contributes an additional two gate delays over the 4-bit CLA adder, thus increasing the total to six gate delays. Continuing in this manner (a 64-bit add takes eight gate delays, a 256-bit add takes ten gate delays), we see that the delay for a CLA adder is dependent on the number of levels of carry logic, and not on the length of the adder. If a group size of four is chosen, then the number of levels in an N-bit CLA is expressed in Equation 2.21 and in general the number of levels in a CLA for a group size of k is expressed in Equation 3.22. For an N-bit CLA adder, each level of carry logic introduces two gate delays in addition to a gate delay for the generate and propagate signals and a gate delay for the sum. The total gate delay is expressed in Equation 2.23, which shows that the delay of a CLA adder is logarithmically dependent on the size of the adder. This theoretically results in one of the fastest adder architectures.

$$\text{CLA levels (with group size of 4)} = \lceil \log_4 N \rceil \quad 2.21$$

$$\text{CLA levels (with group size of k)} = \lceil \log_k N \rceil \quad 2.22$$

$$\text{CLA gate delay} = 2 + 2.\lceil \log_k N \rceil \quad 2.23$$

From a VLSI design perspective, this adder may take more time to implement, but there still exists regularity with the architecture that allows building long adders fairly easily. The reuse of the CLA logic definitely contributes to the feasibility of building a long adder without additional design time. Also, after an adder is built, it can be used as a subcell, as is done with the 4-bit adders as blocks in the 16-bit CLA adder. A drawback to CLA adders are their larger areas. There is a large amount of hardware dedicated to calculating the carry bits from cell to cell. However, if the application calls for high performance, then the benefits of decreased delay can outweigh the larger area.

2.3.4 Carry Select Adder

Adding two numbers by using redundancy can speed addition even further. That is, for any number of sum bits we can perform two additions, one assuming the carryin is 1 and one assuming the carryin is 0, and then choose between the two results once the actual carryin is known. This scheme, proposed by Sklanski in 1960, is called

conditional-sum addition [21]. An implementation of this scheme was first realized by Bedrij and is called the Carry Select Adder (CSLA) [22].

The CSLA divides the adder into blocks that have the same input operands except for the carryin. Figure 2.9 shows a possible implementation for a 16-bit CSLA using ripple carry adder blocks. The carryout of the first block is used as the select line for the 9-bit 2-to-1 mux. The second and third blocks calculate the signals $sum_{16} - sum_8$ in parallel, with one block having its carryin hardwired to 0 and another hardwired to 1. After one 8-bit ripple adder delay there is only the delay of the mux to choose between the results of block 2 or 3. Equation 2.24 shows the delay for this adder. The 16-bit CSLA can also be built by dividing it into even more blocks. Figure 2.10 shows the block diagram for the adder if it were divided into 4-bit RCA blocks. Equation 2.25 expresses the delay for this structure.

$$t_{CSLA_{16a}} = t_{8bitRCA} + t_{(9bitmux)} \quad 2.24$$

$$t_{CSLA_{16b}} = t_{4bitRCA} + 3.t_{(5bitmux)} \quad 2.25$$

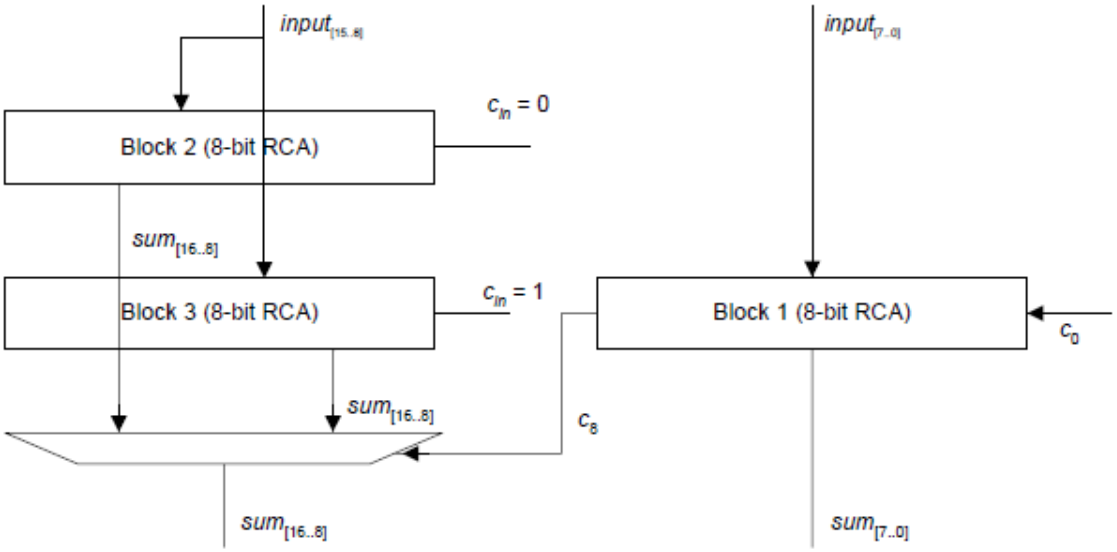


Figure 2.9: Schematic for a 16-bit CSLA with 8-bit RCA blocks

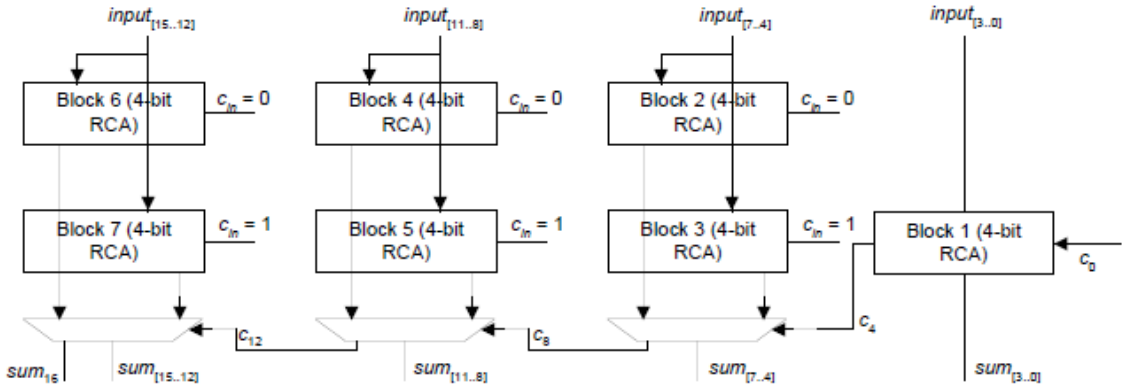


Figure 2.10: Schematic for a 16-bit CSLA with 4-bit RCA blocks

The CSLA can use any of the adder structures discussed in the previous sections as subcells. The delay ultimately comes down to the speed of the adder subcell used and the speed of the muxes used to select the sum bits. A general equation for this adder is expressed in Equation 2.28, where N is the adder size, and k is the group size of each adder subcell.

$$t_{CSLA_N} = t_{k-bitadder} + \frac{N}{k}.t_{((k+1)bitmux} \quad 2.26$$

The CSLA described so far is called the Linear Carry Select Adder, because its delay is linearly dependent on the length of the adder. In the worst case, the carry signal must ripple through each mux in the adder. Also, notice that the subcells are done with their addition at the same time, yet the more significant bits are waiting at the input of the mux to be selected. From a VLSI design perspective, the CSLA uses a large amount of area compared to the other adders. There is hardware in this architecture which computes results that are thrown away on every addition, but the fact that the delay for an add can be replaced by the delay of a mux makes this architecture very fast. Also, the Linear CSLA has regularity that makes it easier to layout.

2.3.5 Prefix Adder

Binary carry-propagating addition can be efficiently expressed as a prefix computation. Adders in which the computation of carries is based on the prefix equations are naturally called ‘prefix adders’. Those which compute multiple sub-terms in parallel by exploiting the associativity property are called ‘parallel prefix adders’. For n-bit addition, n a power of 2, a minimum depth prefix adder comprises $1 + \log_2 n$ unate logic gates, plus 1 non-unate logical stage [23].

The first stage of the adder computes carry generate (g), propagate (p) and kill (k) terms for each bit according to the relations as similar to the PFA:

$$\text{generate}_i(g_i) = a_i . b_i \quad 2.27$$

$$\text{kill}_i(k_i) = \overline{a_ı + b_ı} \quad 2.28$$

$$\text{propagate}_i(p_i) = a_i + b_i = a_i \oplus b_i \quad 2.29$$

The g and k terms are then combined in $\log_2 n$ logical stages to produce carries (c_i) into each bit position using the iterative relation:

$$c_{i+1} = g_i + \overline{k_ı} . c_i \quad 2.30$$

The final stage computes sum bits (s_i) as:

$$s_i = p_i \oplus c_i \quad 2.31$$

The first and last stages are intrinsically fast because they involve only simple operations on signals local to each bit position. The intermediate stages embody the long-distance propagation of carries, so the performance of the adder hinges on this part. In a prefix adder this part is constructed of nodes which perform the prefix operation "●". Parallel Prefix Adders (PPA) are variations of the well-known carry look ahead adder (CLA). Similar to a CLA they employ the three-stage structure shown in Fig. 2.11. The only difference between a CLA and a PPA lies in the second stage which is responsible for the generation of the carry signals of the binary addition. In a PPA the prefix operator "●" is introduced and the carry signal generation is treated as a prefix problem [12]. The operator is defined by the equation:

$$(g_{in_1}, p_{in_1}) \bullet (g_{in_2}, p_{in_2}) = (g_{in_1} + p_{in_1} . g_{in_1}, p_{in_1} . p_{in_2}) \quad 2.32$$

In this equation, "●" is applied on two pairs of bits (g_{in1}, p_{in1}) and (g_{in2}, p_{in2}). These pairs represent generate and propagate signals used in the addition. The output of the operator is a new pair of bits generated as described in Equation 2.32. With the use of this operator a prefix network is constructed and the second stage of the adder is the implementation of this network. Fig. 2.12 illustrates how the operator is used in the construction of the prefix network.

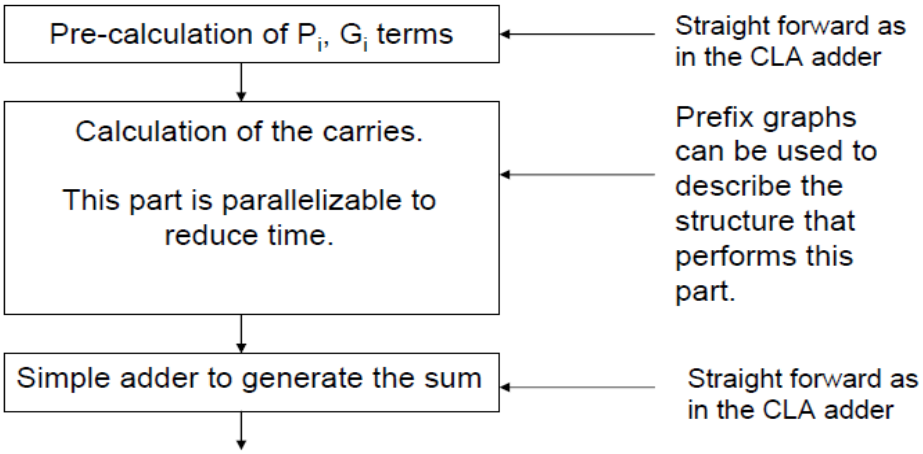
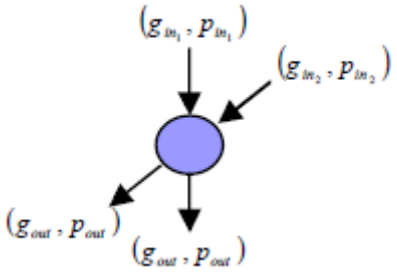


Figure 2.11: Parallel Prefix Adder 3-stage structure



$$(g_{in_1}, p_{in_1}) \circ (g_{in_2}, p_{in_2}) = (g_{in_1} + p_{in_1}.g_{in_2}, p_{in_1}.p_{in_2})$$

Figure 2.12: The "●" operator used in the prefix network of a PPA

- **Prefix Addition**

  In a prefix problem, n inputs $x_{n-1}, x_{n-2}, \dots, x_0$ and an arbitrary associative operator ● are used to compute n outputs$y_i = x_i \bullet x_{i-1} \bullet \dots \bullet x_0$, i=0,…,n-1. Thus each output y_i is dependent on all inputs x_i of same or lower magnitude ($j \leq i$). Carry propagation in binary addition is a prefix problem. Prefix addition can be expressed as follows:

$$g_i = \begin{cases} a_0.b_0 + a_0.c_0 + b_0.c_0 & \text{if } i = 0, \\ a_i.b_i & \text{otherwise} \end{cases} \quad 2.33$$

$$p_i = a_i \oplus b_i \quad 2.34$$

$$(G_{i:k}^{\,l}, P_{i:k}^{\,l}) = (G_{i:j}^{\,l-1}, P_{i:j}^{\,l-1}).(G_{j:k}^{\,l-1}, P_{j:k}^{\,l-1})$$
$$= (G_{i:j}^{\,l-1} + P_{i:j}^{\,l-1}.G_{j:k}^{\,l-1}, P_{i:j}^{\,l-1}.P_{j:k}^{\,l-1}) \quad 2.35$$

$$c_{i+1} = G_{i:0}^{\,m} \quad 2.36$$

$$s_i = p_i \oplus c_i \quad 2.37$$

For $i = 0, \ldots, n-1$ and $l = 1, \ldots, m$, where a_i and b_i are the operand input, g_i and p_i, the generate and propagate, c_i the carry and s_i the sum output signals at bit position i. c_0 and c_n correspond to the carryin c_{in} and carryout c_{out}, respectively. $G_{i:k}^{l}$ and $P_{i:k}^{l}$ denote the group generate and propagate signals for the group of bits i,…,k at level l. The ● operator is repeatedly applied according to a given prefix structure of m levels (i.e. depth m) in order to compute the group generate signal $G_{i:0}^{m}$ for each bit position i. Prefix structures and adders can nicely be visualized using directed acyclic graphs (DAGs) with the edges standing for signals or signal pairs and the nodes representing the four logic operators depicted in Figure 2.13.

Figure 2.14 shows the general prefix adder structure. The square (□) and diamond (◊) nodes form the pre-processing and post-processing stages. The black nodes (●) evaluate the prefix operator ● and the white nodes (○) pass the signals unchanged to the next level in the prefix carry-propagation stage, which is visualized by prefix graphs in the sequel. These prefix graphs consist of n columns (i.e. bit positions) and m rows (i.e. prefix levels) of black or white nodes, where each row corresponds to one black node delay. The top and bottom margins of a column reflect the input and output signal arrival times for that particular bit. Because white nodes do not contain any logic, they are neglected in graph size measures [28].

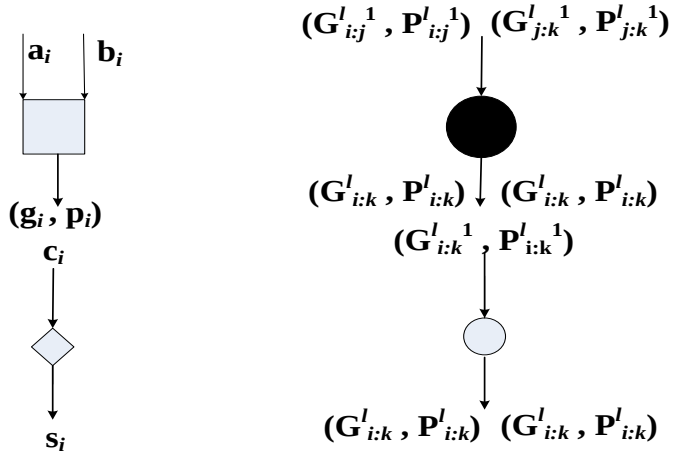


Figure 2.13: Prefix Adder logic operators

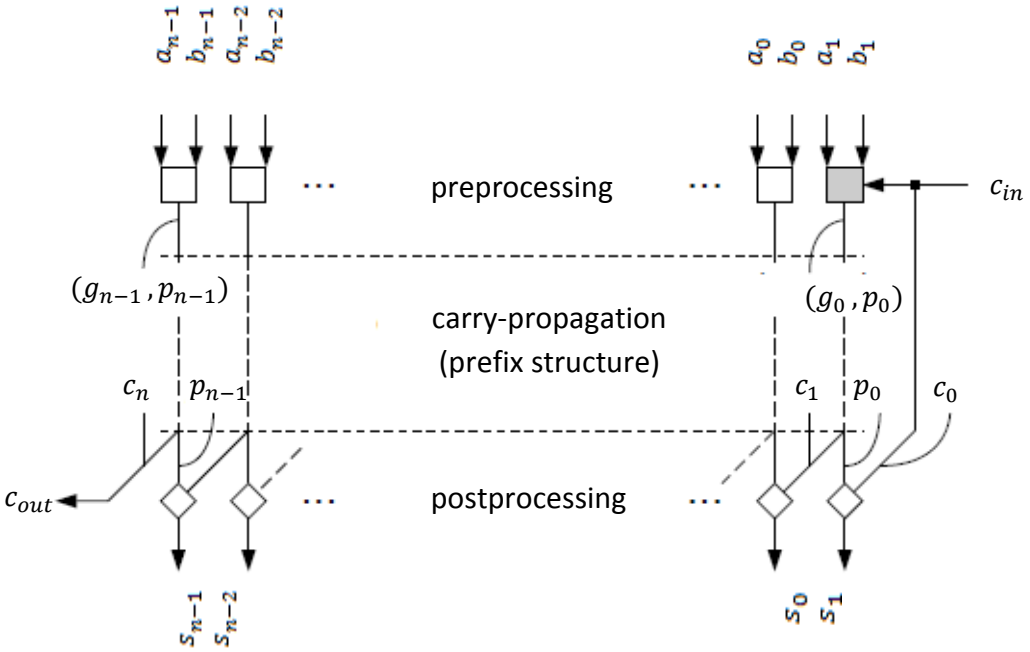


Figure 2.14: Prefix Adder structure

The structure of the prefix network determines the type of the prefix adder. Over the years several classical topologies have been proposed which optimize for one of the following parameters: area, logic-depth size, fan-out and interconnect count. The Sklansky adder topology offers a minimum depth prefix network, at the expense of fan-out and area [21]. The Kogge-Stone adder has minimum depth and fan-out but maximum area and interconnects count [25]. Ladner and Fischer proposed a general way to construct the prefix network which includes the minimum depth case of the Sklansky topology with improved area requirement [24]. The Brent-Kung adder has minimum area but maximum depth [26]. The Han-Carlson adder combines the Brent-Kung and the Kogge-Stone structures in order to balance between logic depth and interconnect count [27]. Knowles proposed a way to construct adders of minimum depth with all possibilities of fan-out count ranging from the minimum case seen in the Kogge-Stone to the maximum fan-out seen in the Sklansky adder [10].

2.4 Summary

This chapter presents various adder algorithms and analyses them with respect to speed and area. The ripple carry adder is the obvious solution for lowest area and lowest speed requirements. The carry-skip adder realizes a considerable speed-up at only small area increase. However it is not suited because of the complex block size computation and the inherent logic redundancy, which disallows automatic circuit optimization.

Redundancy removal is possible only at considerable area hardware overhead due to duplicate sum computation and selection circuitry. The resulting carry look-ahead adders come with different carry propagation schemes offering a high variety of high performance adder circuits. A 32-bit CLA adder architecture is then chosen to be used in designing MAC unit.

Table 2.2: Adder Comparison: N is the length of the adder and k is the group size, if applicable

Adder	Delay
RCA	N
CSKA	N/k
CLA	$\log_4 N$
CSLA	$\frac{\log_4 N}{\frac{N}{k}}$

2.5 Basics of Multipliers

Multipliers play an important role in today's digital signal processing and various other applications. With advances in technology, many researchers have tried and are trying to design multipliers which offer either of following – high speed, low power consumption, regularity of layout and hence less area or even combination of them in one multiplier. Thus making them suitable for various high speed, low power, compact VLSI implementation. The common multiplication method is add and shift algorithm. Multiplication is a mathematical operation that at its simplest is an abbreviated process of adding an integer to itself, a specified number of times. A number (multiplicand) is added to itself a number of times as specified by another number (multiplier) to form a result (product).

Multiplication hardware often consumes much time and area compared to other arithmetic operations. Digital signal processors use a multiplier/MAC unit as a basic building block [29] and the algorithms they run are often multiply-intensive. Multiplication-based operations such as Multiply and Accumulate (MAC) are currently implemented in many Digital Signal Processing (DSP) applications such as convolution, Fast Fourier Transform (FFT), filtering and in microprocessors in its arithmetic and logic unit. In this chapter, we discuss different architectures for multiplication and the methods

that improve speed and/or area. Also, it is important to consider these methods in the context of VLSI design. It is beneficial to find structures that are modular and easy to layout. Many of the architectures described in this chapter will be used in the implementation of the multiply-accumulate unit.

2.5.1 Multiplication Definition

To perform an M-bit by N-bit multiplication as shown in Figure 2.15, the M-bit multiplicand $A = a_{(M-1)}a_{(M-2)} \dots a_1a_0$ is multiplied by the N-bit multiplier $B = b_{(N-1)}b_{(N-2)} \dots b_1b_0$ to produce the M+N-bit product $P = p_{(M+N-1)}p_{(M+N-2)} \dots p_1p_0$.The unsigned binary numbers A and B can be expressed by Equations 2.38 and 2.39. The equation for the product is defined in Equation 2.40 [4].

$$A = \sum_{i=0}^{M-1} a_i . 2^i \tag{2.38}$$

$$B = \sum_{i=0}^{N-1} b_i . 2^i \tag{2.39}$$

$$P = A.B = \left(\sum_{i=0}^{M-1} a_i . 2^i\right).\left(\sum_{i=0}^{N-1} b_i . 2^i\right) = \sum_{i=0}^{M-1}\sum_{i=0}^{N-1} (a_i b_i . 2^{i+j}) \tag{2.40}$$

With two's complement multiplication, both numbers are signed and the result is signed. If A and B are signed binary numbers, they are expressed by Equations 2.41 and 2.42. The equation for the product is defined in Equation 2.43.

$$A = -a_{M-1} . 2^{M-1} + \sum_{i=0}^{M-2} a_i . 2^i \tag{2.41}$$

$$B = -b_{N-1} . 2^{N-1} + \sum_{i=0}^{N-2} b_i . 2^i \tag{2.42}$$

$$P = A.B = \left(-a_{M-1} . 2^{M-1} + \sum_{i=0}^{M-2} a_i . 2^i\right).\left(-b_{N-1} . 2^{N-1} + \sum_{i=0}^{N-2} b_i . 2^i\right) \tag{2.43}$$

Any multiplier can be divided into three stages: Partial productions generation stage, partial products addition stage and the final addition stage. In the first stage, the multiplicand and the multiplier are multiplied bit by bit to generate the partial products. In this stage, a second-order Booth encoding algorithm is usually used instead to reduce the number of partial products to half. The second stage is the most important, as it is the

most complicated and determines the speed of the overall multiplier. In the last stage, the two-row outputs of the tree are added using any high speed adder such as look-ahead adder to generate the output result.

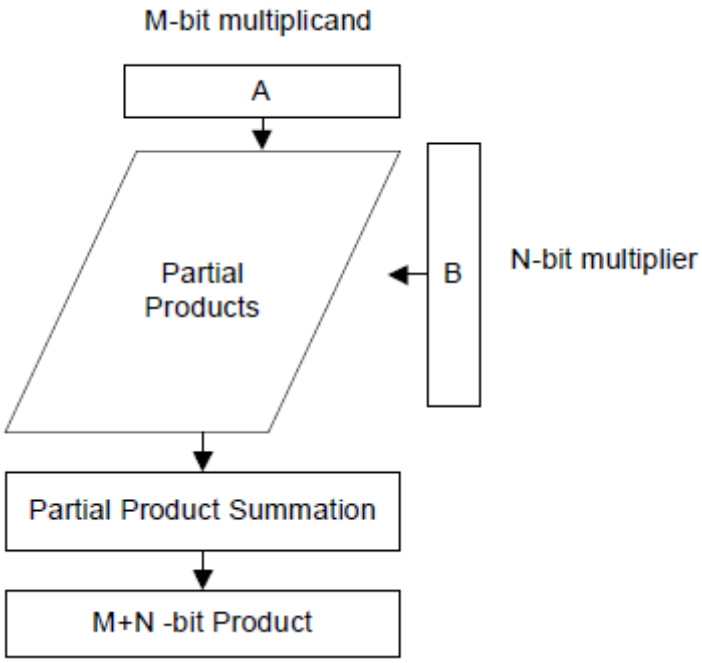


Figure 2.15: Generic Multiplier Block Diagram.

2.5.2 Array Multiplier

From Equation 2.40, each multiplicand is multiplied by a bit in the multiplier, generating N partial products. Each of these partial products is either the multiplicand shifted by some amount, or 0. This is illustrated in Figure 2.16 for an M×N multiply operation. This figure can map directly into hardware and is called the array multiplier. The generation of partial products consists of simple AND'ing of the multiplier and the multiplicand. The accumulation of these partial products can be done with rows of ripple adders. Thus, the carry out from the least significant bit ripples to the most significant bit of the same row, and then down the “left side" of the structure. Figure 2.17 shows a 4x4 unsigned array multiplier [17].

The partial products are added in ripple fashion with half and full adders. A full adder's inputs require the carryin from the adjacent full adder in its row and the sum from a full adder in the above row. Rabaey [17] states that finding the critical path in this structure is non-trivial, but once identified, results in multiple critical paths. It requires a

lot of time to optimize the adders in the array since all adders in the multiple critical paths need to be optimized to result in any speed increase (this implies optimization of both the sum and carryout signals in a full adder). The delay basically comes down to a ripple delay through a row, and then down a column, so it is linearly proportional$(t_d \approx M + N)$ to the sum of the sizes of the input operands.

Conventional linear array multipliers consist of rows of carry-save adders (CSA). In a linear array multiplier, as the data propagates down through the array, each row of CSA's adds one additional partial-product to the partial sum. Since the intermediate partial sum is kept in a redundant, carry-save form there is no carry propagation. This means that the delay of an array multiplier is only dependent upon the depth of the array, and is independent of the partial-product width. Their high performance and regular structure have perpetuated the use of array multipliers for VLSI math co-processors and special purpose DSP chips. The main disadvantage of the array multiplier is the worst-case delay of the multiplier proportional to the width of the multiplier. As operand sizes increase, linear arrays grow in size at a rate equal to the square of the operand size. This is because the number of rows in the array is equal to the length of the multiplier, with the width of each row equal to the width of multiplicand. The large size of full arrays typically prohibits their use, except for small operand sizes, or on special purpose math chips where a major portion of the silicon area can be assigned to the multiplier array. Another problem with array multipliers is that the hardware is underutilized. As the sum is propagated down through the array, each row of CSA's computes a result only once, when the active computation front passes that row. Thus, the hardware is doing useful work only a very small percentage of the time. This low hardware utilization in conventional linear array multipliers makes performance gains possible through increased efficiency. For example, by overlapping calculations pipelining can achieve a large gain in throughput.

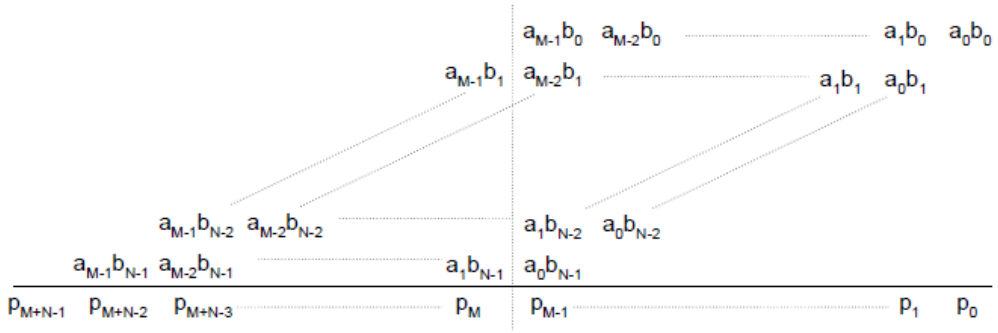


Figure 2.16: Partial product array for an M × N multiplier.

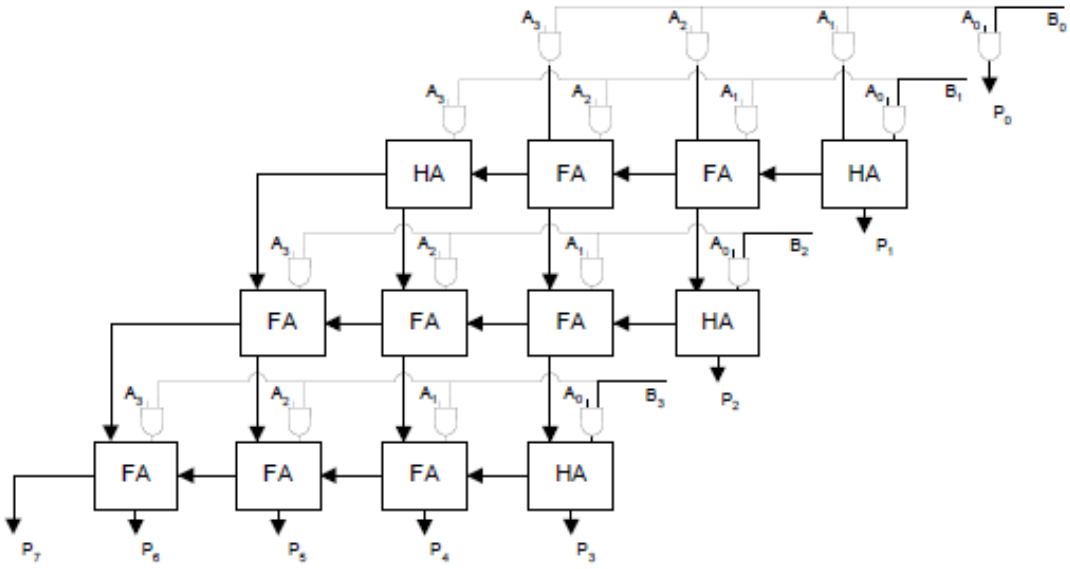


Figure 2.17: Array multiplier block diagram for a 4x4 multiplier [17]

2.5.3 Tree Multiplier

The tree multiplier reduces the time for the accumulation of partial products by adding all of them in parallel, whereas the array multiplier adds each partial product in series. The tree multiplier commonly uses CSAs to accumulate the partial products.

2.5.3.1 Wallace Tree

The reduction of partial products using full adders as carry-save adders (also called 3:2 counters) became generally known as the “Wallace Tree" [32]. This architecture reduces the partial products at a rate of $\log_{3/2} N/2$. Figure 2.18 shows an example of tree reduction for an 8×8-bit partial product tree. The ovals around the dots represent either a full adder (for three circled dots) or a half adder (for two circled dots). This tree is reduced to two rows for a carry-propagate adder after four stages. There are many ways to reduce this tree with CSAs, and this example is just one of them. By using carry-save adder the need of carry propagation in the adder is avoided and latency of one addition is equal to gate delay of adder.

2.5.3.2 Carry Save Compressor

The disadvantage for both the Wallace structures from a VLSI perspective is the irregularity in connections between the stages. Weinberger [33] introduced a structure he called “4:2 carry-save adder module" which takes in four inputs of weight 2^i, one input of

weight 2^{i+1}, and produces one output of weight 2^i and two outputs of weight 2^{i+1}. Figure 2.19 shows the block diagram for a 4:2 compressor using two 3:2 compressors. This structure actually takes 5 inputs and produces 3 outputs, where one input is a carryin and one output is a carryout. These exist to allow the chaining together of 4:2s for rows of bits. The two carryouts generated are of the same weight, and the important feature is that there is no “rippling" effect from the carryin to the 2nd carryout. For a given set of inputs in_3, in_2, in_1, and in_0, the signal c_{out} must not change when c_{in} changes. This is the reason it is an effective carry save structure, and it maintains the speed in reducing a tree of partial product bits.

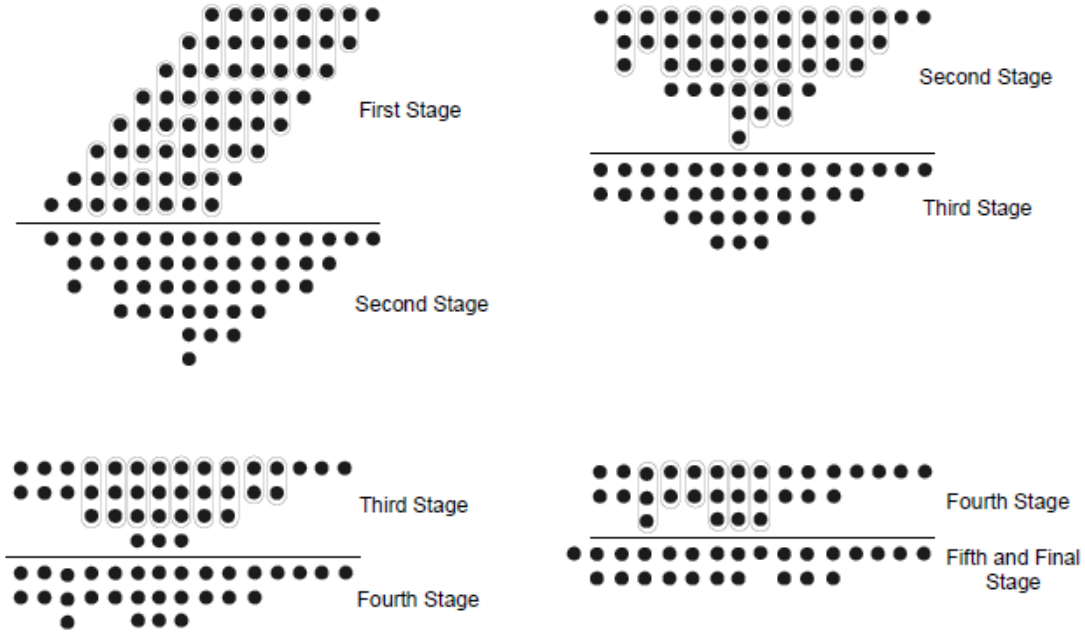


Figure 2.18: Wallace Tree for an 8 × 8-bit partial product tree

There are various ways to implement the 4:2 compressor, the simplest of which is cascading two full adder cells together. This structure was popularized by Santoro [30], who built a 64 × 64-bit multiplier. The interconnection of these full adders must not make the 2nd carryout dependent on the carryin. Since it takes two full adders to build one 4:2, the speed of each 4:2 compressor is limited by the delay of 3 XOR gates in series [34]. Thus, a 4:2 is slightly faster than two full adders (which is four XOR gate delays). Thus, the tree compression is slightly faster using 4:2s with a trade-off of a little more hardware. It reduces at a rate of $\log_2(\frac{N}{2})$, and there is also the added benefit of regular layout.

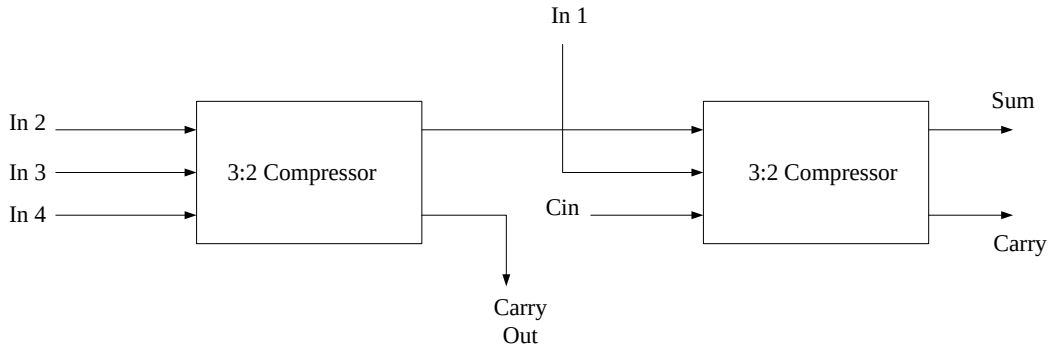


Figure 2.19: Block Diagram of 4:2 compressor

2.5.4 Partial Product Generation Methods

Another method to improve the speed of the multiplication operation is to improve the partial product generation step. This can be done in two ways:

1) Generate the partial products in a faster manner.
2) Reduce the number of partial products that need to be generated.

The first option can only be achieved if a different architecture for partial product generation is used. Considering that a bit is generated with just an AND gate delay, it seems that the partial product bits are already calculated in the fastest way possible. The second option will most likely take longer than an AND gate, but the reduced number of partial products results in a smaller tree and therefore reduces the time during the tree reduction step. As long as the time saved during tree reduction is greater than the extra time it takes to generate fewer partial products, it is beneficial to implement this option.

2.5.4.1 Booth's Algorithm

` A simple method to reduce the number of partial products in a multiplication operation is called Booth's algorithm [35]. Through recoding, potentially only half the number of partial products must be generated. The algorithm begins by looking at two bits of the multiplier at a time, and then determines what partial product to generate based on these two bits. This algorithm works for both signed and unsigned numbers. Table 2.3 shows Booth's algorithm and a decoding scheme. The decoder needs to select the correct partial product based on B_i and B_{i-1}. For a negative multiplicand, hardware that supports subtractions is needed, and for the positive multiplicand a 0 needs to be appended to the most significant bit to guarantee a positive result. Koren (Section 6.1, [20]) states two drawbacks to Booth's algorithm. One is that the variable number of add/subtract

operations leads to a difficult design. For example, there are some cases where you must add all the partial products (all non-zero partial products generated), and other cases where only two non-zero partial products are generated, hence only one add operation is needed. The second drawback is the inefficiency of the algorithm when there are isolated 1's. For example, in an 8 × 8-bit multiplication operation, a multiplier of 01010101 would normally generate 4 non-zero partial products (each 1 in the multiplier replicates the multiplicand), but with Booth's algorithm a total of 8 non-zero partial products are generated. In hardware, the design must always take into account the worst case, so with this algorithm there are still potentially 8 partial products that will be generated which results in no improvement in terms of the hardware.

2.5.4.2 Modified Booth's Algorithm

MacSorley [36] solved the drawbacks of Booth's algorithm by presenting an improved version which is popularly known as Modified Booth Encoding (also Radix-4 Modified Booth Encoding). It recodes two multiplier bits at a time, and uses a third bit as a reference. Figure 2.20 illustrates the manner in which the bits are recoded, with $B = b_{(N-1)}b_{(N-2)} \dots b_1 b_0$ as the original multiplier and $D = d_{(N-1)}d_{(N-2)} \dots d_1 d_0$ as the new recoded multiplier. At each step 3 bits of Multiplier B, b_{i+1}, b_i, b_{i-1}, are examined and corresponding d_i is calculated. B is always appended on the right with zero ($b_{-1} = 0$), and n is always even (B is sign extended if needed). The product A.B is then obtained by adding n/2 partial products. Table 2.3 shows what the recoded multiplier values are, and the partial products they generate. The Booth decoder takes these recoded multiplier bits and selects the correct partial product. Bewick [37] shows an implementation for this scheme, and a block diagram of this implementation is shown in Figure 2.21.

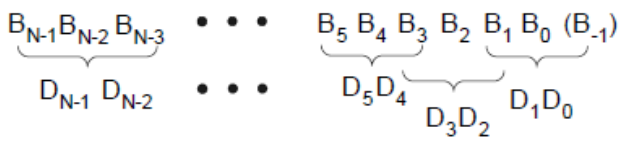


Figure 2.20: Recoded multiplier using Modified Booth Encoding

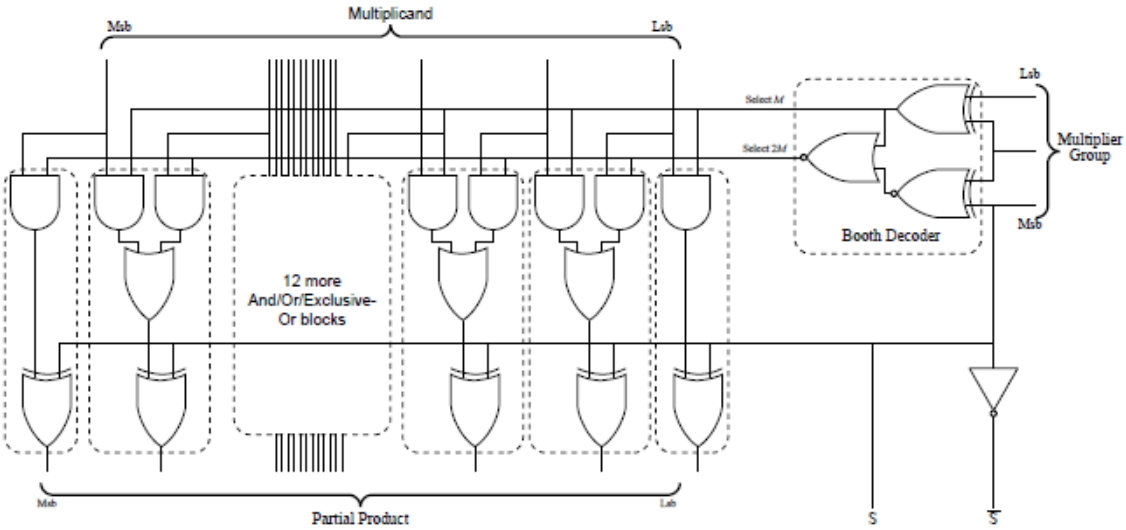


Figure 2.21: Bewick's implementation of the Booth encoder and decoder

Table 2.3
Modified Booth Algorithm [9]

B_{i+1}	B_i	B_{i-1}	Recoded Digit	Recoded Bits D_{i-1}	Recoded Bits D_i	Partial Product
0	0	0	0	0	0	0*Multiplicand
0	0	1	+1	0	1	+1*Multiplicand
0	1	0	+1	0	1	+1*Multiplicand
0	1	1	+2	1	0	+2*Multiplicand
1	0	0	-2	1	0	-2*Multiplicand
1	0	1	-1	0	1	-1*Multiplicand
1	1	0	-1	0	1	-1*Multiplicand
1	1	1	0	0	0	0*Multiplicand

2.5.5 Summary

The multiplication algorithms are compared in terms of their time complexity as well as area complexity. Time complexity refers to the total number of clock cycles needed during the execution process of the multiplier. This in turn depends on the number of add and shift operation the particular multiplier algorithm requires which is a function of the number of bits required to represent the operands (i.e. n). The time complexity of array multipliers is proportional to the operand word length (O(n)), but the time complexity of tree multipliers is proportional to the logarithm of the operand word length ($O(\log n)$) [3]. The area complexity of both types of multipliers, when physically implemented, is proportional to the square of the operand word length ($O(n^2)$) [4]. In parallel multipliers number of partial products to be added is the main parameter that

determines the performance of the multiplier. To reduce the number of partial products to be added, Modified Booth algorithm is one of the most popular algorithms. To achieve speed improvements Wallace Tree algorithm can be used to reduce the number of sequential adding stages. Further by combining both Modified Booth algorithm and Wallace Tree technique we can see advantage of both algorithms in one multiplier. However with increasing parallelism, the amount of shifts between the partial products and intermediate sums to be added will increase which may result in reduced speed, increase in silicon area due to irregularity of structure and also increased power consumption due to increase in interconnect resulting from complex routing. On the other hand, serial-parallel multipliers compromise speed to achieve better performance for area and power consumption. So selection of a parallel or serial multiplier actually depends on the nature of application.

Table 2.4
Comparison between Multipliers [31]

Multiplier Type	**Speed**	**Circuit Complexity**	**Layout**	**Area**
Array	Low/Medium	Simple	Regular	Smallest
Booth	High	Complex	Irregular	Medium
Wallace	Higher	Medium	More irregular	Large
Booth Wallace	Highest	More complex	Medium regularity	Largest

2.6 MAC UNIT

In the majority of digital signal processing (DSP) applications, the critical operations usually involve many multiplications and/or accumulations. So, for a real-time signal processing, a high speed and high throughput Multiplier-Accumulator (MAC) is always a key to achieve a high performance digital signal processing system. In the last few years, the main consideration of MAC design is to enhance its speed. This is because speed and throughput rate is always the concern of digital signal processing system. But for the epoch of personal communication, low power design also becomes another main design consideration. This is because battery energy available for these portable products limits the power consumption of the system. Therefore, the main motivation of this research is to investigate various pipelined multiplier/accumulator architectures and

circuit design techniques that are suitable for implementing high throughput signal processing algorithms and at the same time achieve low power consumption.

In any multiplication algorithm, the operation is decomposed in three steps. Each partial product shows a multiple of the multiplicand to be summed to the final result. Currently almost all high-speed multipliers use a modified Booth multiplication algorithm. All DSP algorithms would require some form of the Multiplication and Accumulation Operation. This is the most important block in DSP systems which are composed of an adder, multiplier and the accumulator. Usually adders implemented in DSPs are Ripple Carry Adders, Carry-Select or Carry-Save adders, as speed is of utmost importance in a DSP. Basically the multiplier will multiply the inputs and give the results to the adder, which will add the multiplier results to the previously accumulated results. This operation eases the computation of the most important formula i.e., $b(n).x(n-k)$ which is needed in filters, Fourier analysers, etc. The inputs for the MAC are supposed to be fetched from some memory location and fed to the multiplier block of the MAC, which will perform multiplication and give the result to adder which will accumulate the result and then if needed will also store the result into a memory location. This entire process is to be achieved in a single clock cycle [6].

The MAC unit determines the speed of the overall system; it always lies in the critical path. In order to improve the speed of the MAC unit, there are two major bottlenecks that need to be considered. The first one is the partial products reduction network that is used in the multiplication block and the second one is the accumulator. Both of these stages require addition of large operands that involve long paths for carry propagation. To speed up the multiplication process implements both the multiplication and the accumulation operations within the same functional block by merging the accumulator with the multiplication circuit using tree architectures for the partial products reduction network [7].

Many researchers have attempted in designing MAC architecture with high speed computational performance and low power consumption. Elguibaly proposed a fast pipelined implementation to lower the MAC architecture's critical delay [8]. Murakami et al. adopted the half array implementation to design a high-speed and area-effective MAC architecture [9]. Raghuneth et al. made use of a carry-save multiplier that can simplify sign extension and saturation, and further applies it on MAC architecture to reduce the unit's area and power consumption [10]. Li Hsun proposed a low-power Multiplication-Accumulation Computation (MAC) unit using the radix-4 Booth algorithm, by reducing

its architectural complexity and minimizing the switching activities [11]. Kwon et al. developed a merged MAC unit based on fast 5:2 compressors instead of 3:2 and 4:2 compressors [12]. Fayed et al. proposed new data merging architecture for high speed multiply accumulate units [13, 14]. The architecture can be applied on binary trees constructed using 4:2 compressor circuits. Increasing the speed of operation is achieved by taking advantage of the available free input lines of the compressor circuits, which result from the natural parallelogram shape of the generated partial products and using the bits of the accumulated value to fill in these gaps.

The general construction of the MAC operation can be presented by this equation

$$Z = A \times B + Z \qquad 2.44$$

where, the multiplier A and multiplicand B are assumed to have n bits each and the addend Z has (2n+1) bits. The basic MAC Unit is made up of a multiplier and an accumulator as shown in Figure 2.1.The multiplier can also be divided into the partial products generator, summation tree, and final adder. This construct leads to four basic blocks to implement. The summation network represents the core of the MAC unit. This block occupies most of the area and consumes most of the circuit power and delay. Several algorithms and architectures are developed in attempt to optimize the implementation of this block.

A general hardware architecture of this MAC is shown in Figure 2.22. It executes the multiplication operation by multiplying the input multiplier and the multiplicand. This is added to the previous multiplication result as the accumulation step. The basic MAC operation comprises of a multiplication which can be divided into three operational steps. The first is radix-2 Booth encoding in which a partial product is generated from the multiplicand and the multiplier. The second is adder array or partial product compression to add all partial products and convert them into the form of sum and carry. The last is the final addition in which the final multiplication result is produced by adding the sum and the carry. If the process to accumulate the multiplied results is included, a MAC consists of four steps, as shown in Figure 2.23, which shows the operational steps explicitly.

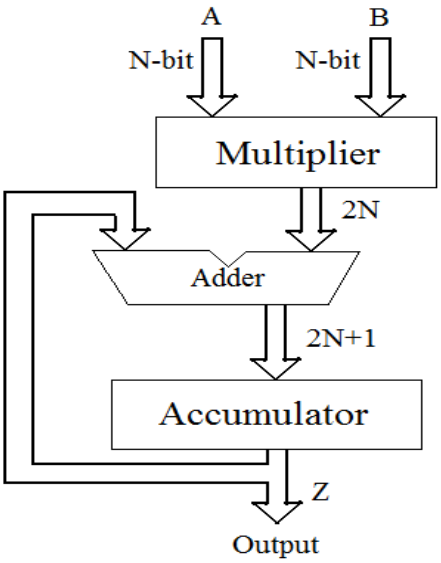


Figure 2.22: Basic MAC Architecture

The N-bit 2's complement binary number A can be expressed as-

$$A = -2^N - a_{N-1} - \sum_{i=0}^{N-3} a_i 2^i \qquad ; \qquad a_i \in 0,1 \tag{2.45}$$

If Equation 2.45 is expressed in base-4 type redundant sign digit form in order to apply the radix-2 Booth's algorithm, it would be [5].

$$A = \sum_{i=0}^{\frac{N}{2}-1} d_i 4^i \tag{2.46}$$

$$d_i = -2a_{2i+1} + a_{2i} + a_{2i-1} \tag{2.47}$$

If Equation 2.46 is used, multiplication can be expressed as

$$A \times B = \sum_{i=0}^{\frac{N}{2}-1} d_i 2^{2i} B \tag{2.48}$$

If these equations are used, therefore mentioned multiplication–accumulation results can be expressed as

$$P = A \times B + Z = \sum_{i=0}^{\frac{N}{2}-1} d_i 2^i B + \sum_{j=0}^{2N-1} z_j 2^j \tag{2.49}$$

Each of the two terms on the right-hand side of Equation 2.49 is calculated independently and the final result is produced by adding the two results. The MAC architecture implemented by Equation 2.49 is called the standard design [18].

If N-bit data are multiplied, the number of the generated partial products is proportional to. In order to add them serially, the execution time is also proportional to N. The architecture of a multiplier, which is the fastest, uses radix-2 Booth encoding that generates partial products and a Wallace tree based on CSA as the adder array to add the partial products. If radix-2 Booth encoding is used, the number of partial products, i.e., the inputs to the Wallace tree, is reduced to half, resulting in the decrease in CSA tree step. In addition, the signed multiplication based on 2's complement numbers is also possible. Due to these reasons, most current used multipliers adopt the Booth encoding.

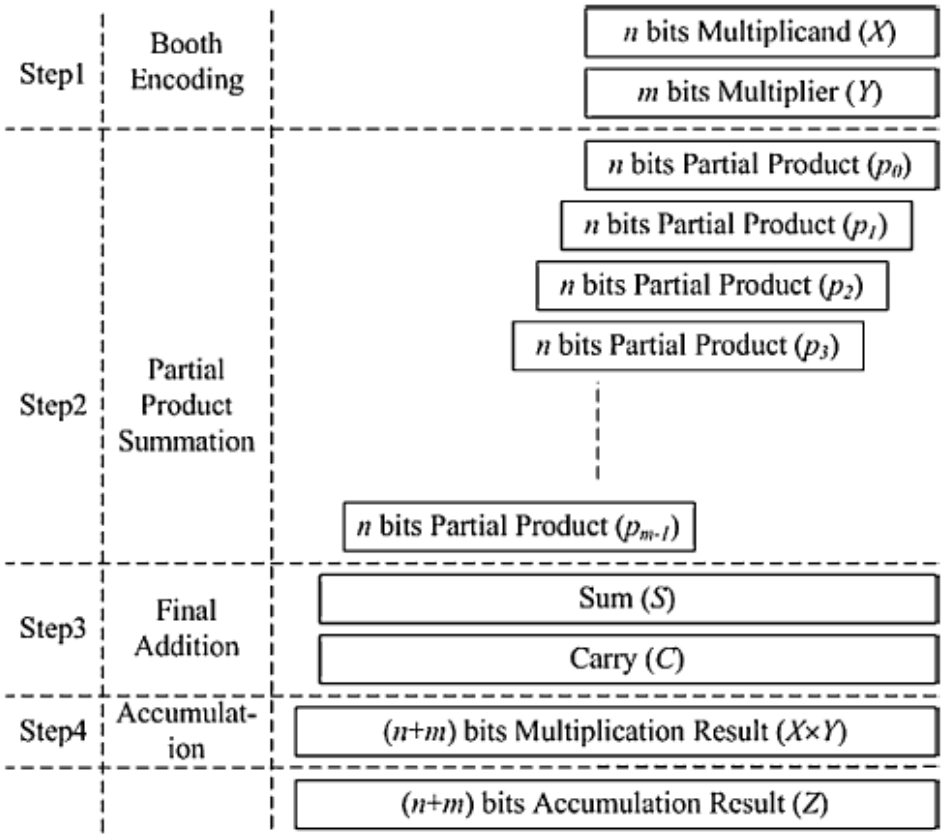


Figure 2.23: Basic steps for deriving a MAC Unit [15]

2.6.1 Derivation of MAC Arithmetic

1) Basic Concept: If an operation to multiply two N–bit numbers and accumulate into a 2N-bit number is considered, the critical path is determined by the 2N-bit accumulation operation. If a pipeline scheme is applied for each step in the

standard design of Figure 2.23, the delay of the last accumulator must be reduced in order to improve the performance of the MAC. The overall performance of the MAC is improved by eliminating the accumulator itself by combining it with the CSA function. If the accumulator has been eliminated, the critical path is then determined by the final adder in the multiplier. The basic method to improve the performance of the final adder is to decrease the number of input bits. In order to reduce this number of input bits, the multiple partial products are compressed into a sum and a carry by CSA. The number of bits of sums and carries to be transferred to the final adder is reduced by adding the lower bits of sums and carries in advance within the range in which the overall performance will not be degraded. A 2-bit CLA is used to add the lower bits in the CSA. In addition, to increase the output rate when pipelining is applied, the sums and carries from the CSA are accumulated instead of the outputs from the final adder in the manner that the sum and carry from the CSA in the previous cycle are inputted to CSA. Due to this feedback of both sum and carry, the number of inputs to CSA increases, compared to the standard design and [16]. In order to efficiently solve the increase in the amount of data, CSA architecture is modified to treat the sign bit.

2) Equation Derivation: The aforementioned concept is applied to 2.5 to express the MAC arithmetic. Then, the multiplication would be transferred to a hardware architecture that complies with the concept, in which the feedback value for accumulation will be modified and expanded for the new MAC.

First, if the multiplication in Equation 2.48 is decomposed and rearranged, it becomes

$$A \times B = d_0 2B + d_1 2^2 B + d_2 2^4 B + \cdots + d_{\frac{N}{2}-1} 2^{N-2} B \qquad 2.50$$

If Equation 2.50 is divided into the first partial product, sum of the middle partial products, and the final partial product, it can be re-expressed as Equation 2.51. The reason for separating the partial product addition as Equation 2.51 is that three types of data are fed back for accumulation, which are the sum, the carry, and the pre-added results of the sum and carry from lower bits.

$$A \times B = d_0 2B + \sum_{i=1}^{\frac{N}{2}-2} d_i 2^{2i} B + d_{\frac{N}{2}-1} 2^{N-2} B \qquad 2.51$$

Now, the given concept is applied to Z in Equation 2.49. If is first divided into upper and lower bits and rearranged, Equation 2.52 will be derived. The first term of the right-hand side in Equation 2.52 corresponds to the upper bits. It is the value that is fed back as the sum and the carry. The second term corresponds to the lower bits and is the value that is fed back as the addition result for the sum and carry

$$Z = \sum_{i=0}^{N-1} z_i 2^i + \sum_{i=N}^{2N-1} z_i 2^i \qquad 2.52$$

The second term can be separated further into the carry and sum term as

$$\sum_{i=N}^{2N-1} z_i 2^i = \sum_{i=0}^{N-1} z_{N+i} 2^i 2^N = \sum_{i=0}^{N-2} (c_i + s_i) 2^i 2^N \qquad 2.53$$

Thus Equation 2.52 is finally separated into three terms as

$$Z = \sum_{i=0}^{N-1} 2^i + \sum_{i=0}^{N-2} c_i 2^i 2^N + \sum_{i=0}^{N-2} s_i 2^i 2^N \qquad 2.54$$

If Equation 2.51 and Equation 2.54 are used, the MAC arithmetic in Equation 2.49 can be expressed as

$$P = \left(d_0 2B + \sum_{i=1}^{\frac{N}{2}-2} d_i 2^{2i} B + d_{\frac{N}{2}-1} 2^{N-2} B \right) + \left(\sum_{i=0}^{N-1} z_i 2^i 2^N + \sum_{i=0}^{N-2} c_i 2^i 2^N + \sum_{i=0}^{N-2} s_i 2^i 2^N \right) \qquad 2.55$$

If each term in Equation 2.55 is matched to the bit position and rearranged, it can be expressed as Equation 2.56 which is the final equation for the MAC. The first parenthesis on the right is the operation to accumulate the first partial product with the added result of the sum and carry. The second parenthesis is the one to accumulate the middle partial products with the sum of CSA that was fed back. Finally, the third parenthesis expresses the operation to accumulate the last partial product with the carry of the CSA.

$$P = \left(d_0 2B + \sum_{i=0}^{N-1} z_i 2^i \right) + \left(\sum_{i=1}^{\frac{N}{2}-1} d_i 2^{2i} B + \sum_{i=0}^{N-2} c_i 2^i 2^N \right) + \left(d_{\frac{N}{2}-1} 2^{N-2} B + \sum_{i=0}^{N-2} s_i 2^i 2^N \right) \qquad 2.56$$

2.6.2 MAC Architecture [15]

If the MAC process explained in the previous section is rearranged, it would be as Figure 2.25, in which the MAC is organized into three steps. When compared with Figure 2.23, it is easy to identify the difference that the accumulation has been merged into the process of adding the partial products. Another big difference from Figure 2.23 is that the final addition process in step 3 is not always run even though it does not appear explicitly in Figure 2.25. Since accumulation is carried out using the result from step 2 instead of that from step 3, step 3 does not have to be run until the point at which the result for the final accumulation is needed.

The hardware architecture of the MAC to satisfy the process in Figure 2.25 is shown in Figure 2.24. The N-bit MAC inputs, A and B, are converted into an (N+1)-bit partial product by passing through the Booth encoder. In the CSA and accumulator, accumulation is carried out along with the addition of the partial products. As a result, N-bit S, C and Z (the result from adding the lower bits of the sum and carry) are generated. These three values are fed back and used for the next accumulation. If the final result for the MAC is needed, P [2N-1: N] is generated by adding S and C in the final adder and combined with P [N-1: 0] already generated.

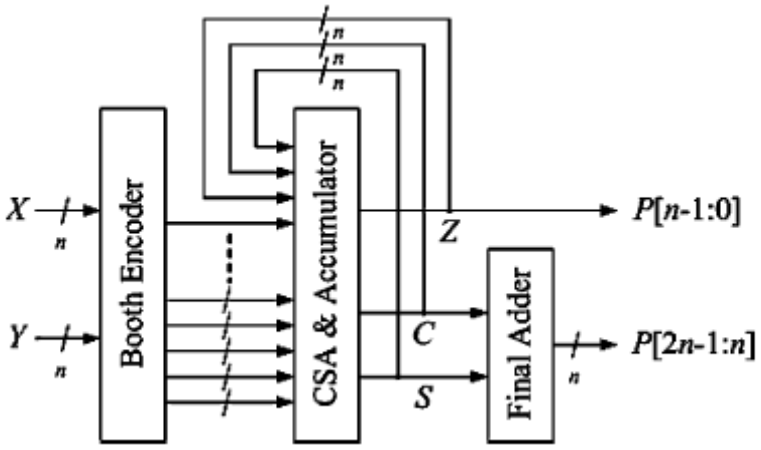


Figure 2.24: Hardware architecture of MAC

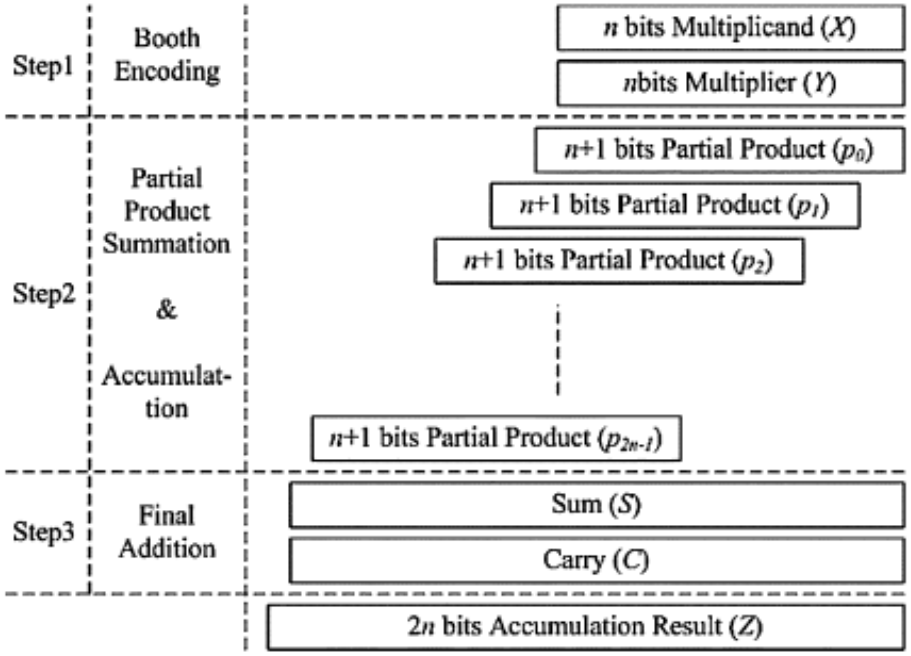


Fig 2.25 Arithmetic operation of MAC [15]

Chapter 3

MULTIPLY ACCUMULATE UNIT DESIGN & IMPLEMENTATION

The Multiply-Accumulate (MAC) unit performs the Multiply instruction and the MAC instruction, which are essential for all DSP processors. The MAC unit determines the speed of the overall system; it always lies in the critical path. Many researchers have attempted in designing MAC architecture with high computational performance and low power consumption. In order to improve the speed of the MAC unit, there are two major bottlenecks that need to be considered. The first one is the partial products reduction network that is used in the multiplication block and the second one is the accumulator. Both of these stages require addition of large operands that involve long paths for carry propagation. To speed up the multiplication process implements both the multiplication and the accumulation operations within the same functional block by merging the accumulator with the multiplication circuit using tree architectures for the partial products reduction network [22]. This chapter discusses the design and implementation of the multiply-accumulate unit which is pipelined into three stages in order to achieve high performance.

3.1 Design and Implementation:

The MAC unit is divided into three pipe stages so it has the ability to be clocked at a faster rate. The latency is three cycles for a MAC or multiply instruction, but the throughput is one operation per cycle. Figure 3.1 shows a block diagram of the MAC unit. The first pipe stage contains the Booth encoding, generation of partial products (Booth decoding), and two rows of 4:2 compressors to begin accumulation of the partial products. The second pipe stage continues accumulation of the partial products with a row of 4:2 compressors and a row of half adders. After this pipe stage the partial product tree is reduced to two rows of 32 bits. The third and final pipe stage contains a 40-bit carry propagate adder (CPA) to complete the MAC or multiply instruction, the 40-bit accumulator, and the accumulator shifter. This stage is the critical pipeline stage of the MAC unit.

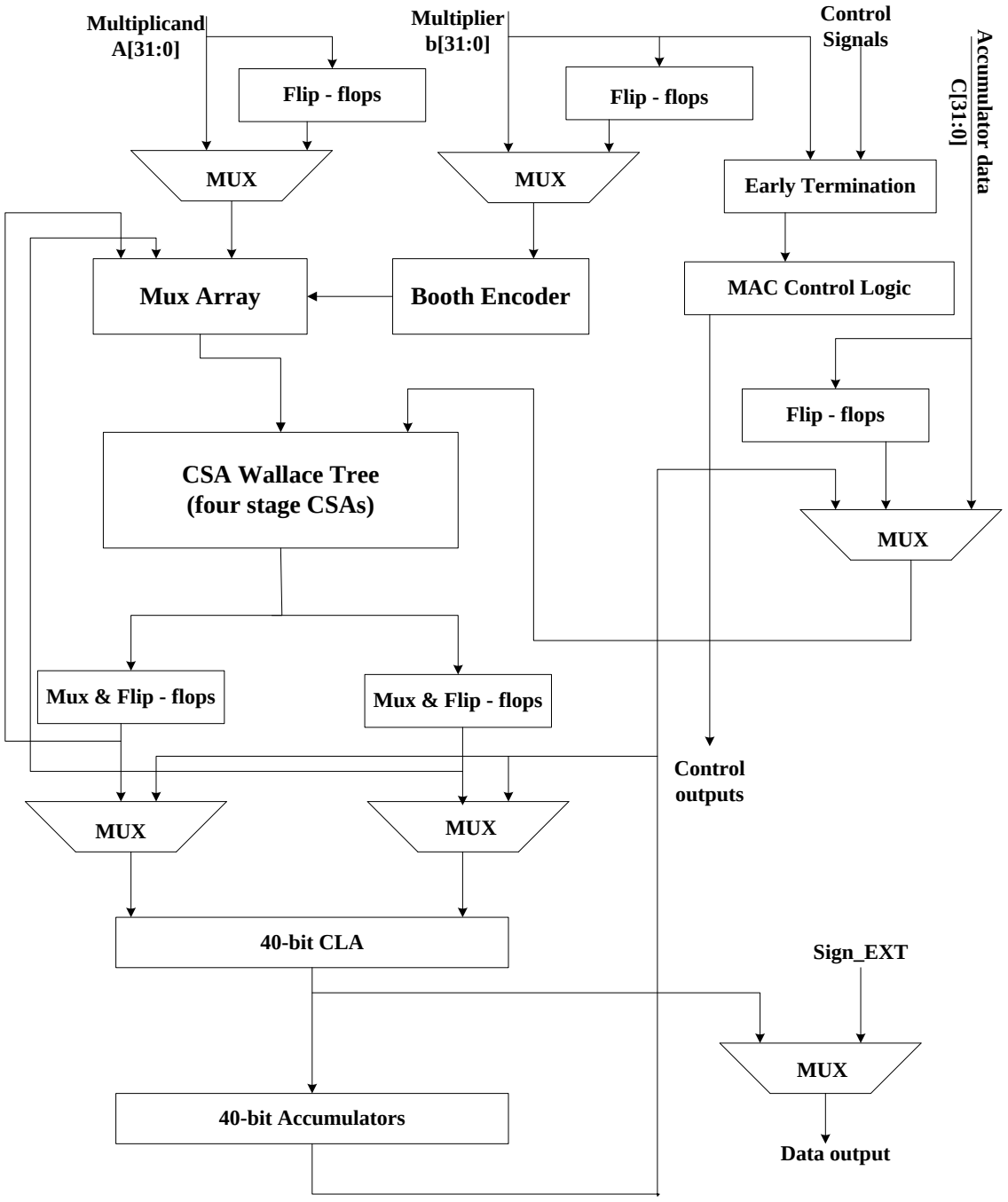


Figure 3.1: Block diagram for MAC unit

3.2 Partial Product Generation

The first step in a multiplication is to generate the partial product bits. In this implementation, modified Booth encoding is chosen because of the simple task of generating the multiplicand and twice the multiplicand for the partial products. The benefits of Booth encoding in multipliers have been questioned because of the efficiency of 4:2 counters versus the complex logic of Booth encoders and selectors [38]. In order to achieve an area/speed balance, we implement the partial product tree, as for this unit it is

not in the critical pipe stage, with Booth encoders because it saves area and the speed will be faster than the critical pipeline stage.

Sign extension

The multiplicand determines the width of the partial product tree, and the multiplier determines the height. Using Modified-Booth encoding for these two signed numbers, results in a tree that is nine rows high. The width of the tree is 32 bits wide because the final result of a 16×16-bit multiplication is 32 bits. Also, each partial product generated needs to be sign extended, and also needs to add an additional bit in the least significant position for negative partial products (the "invert and add 1" for negative numbers in the two's complement number system). There are a total of eight partial products, and each one is 17 bits wide (before sign extension). The additional bit in the least significant position of each partial product increases the height of the tree to nine. Figure 3.2 shows the partial product tree with sign extension.

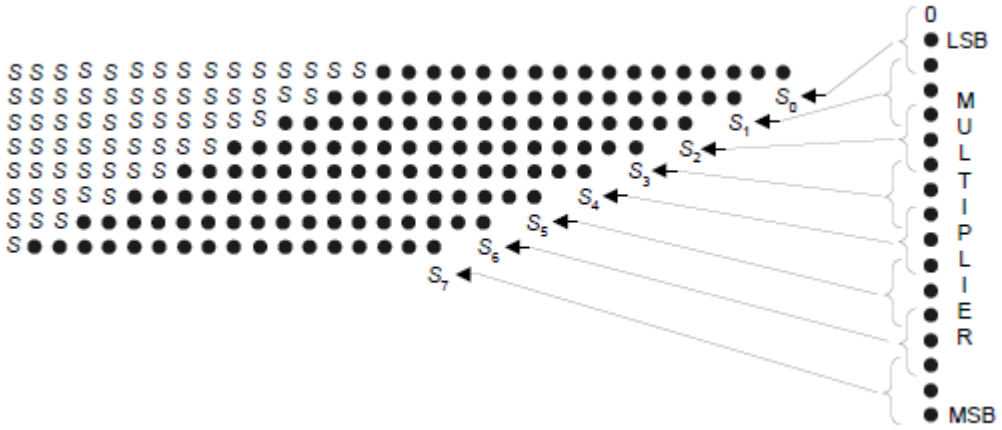


Figure 3.2: Partial Product Tree with sign extension

Although necessary, sign extension in the partial product rows is inefficient because it requires additional hardware to add the sign bits. Also, sign extension presents a large load to the most significant bit of each partial product row, because the most significant bit needs to be replicated many times. This would significantly slow down the partial product generation step. In order to alleviate this sign extension problem, we use a method that reduces the sign extension bits to a constant, and instead add this constant to the partial product tree during every multiplication operation. Appendix A [25] has a thorough explanation for this method, and goes through an example for a 16×16-bit multiplier. Figure 3.3 shows and explains the steps taken to reduce the number of bits to

add in the partial product tree for the 16×16-bit signed multiplication needed in this MAC unit. After the fourth step the partial product bits are ready for accumulation.

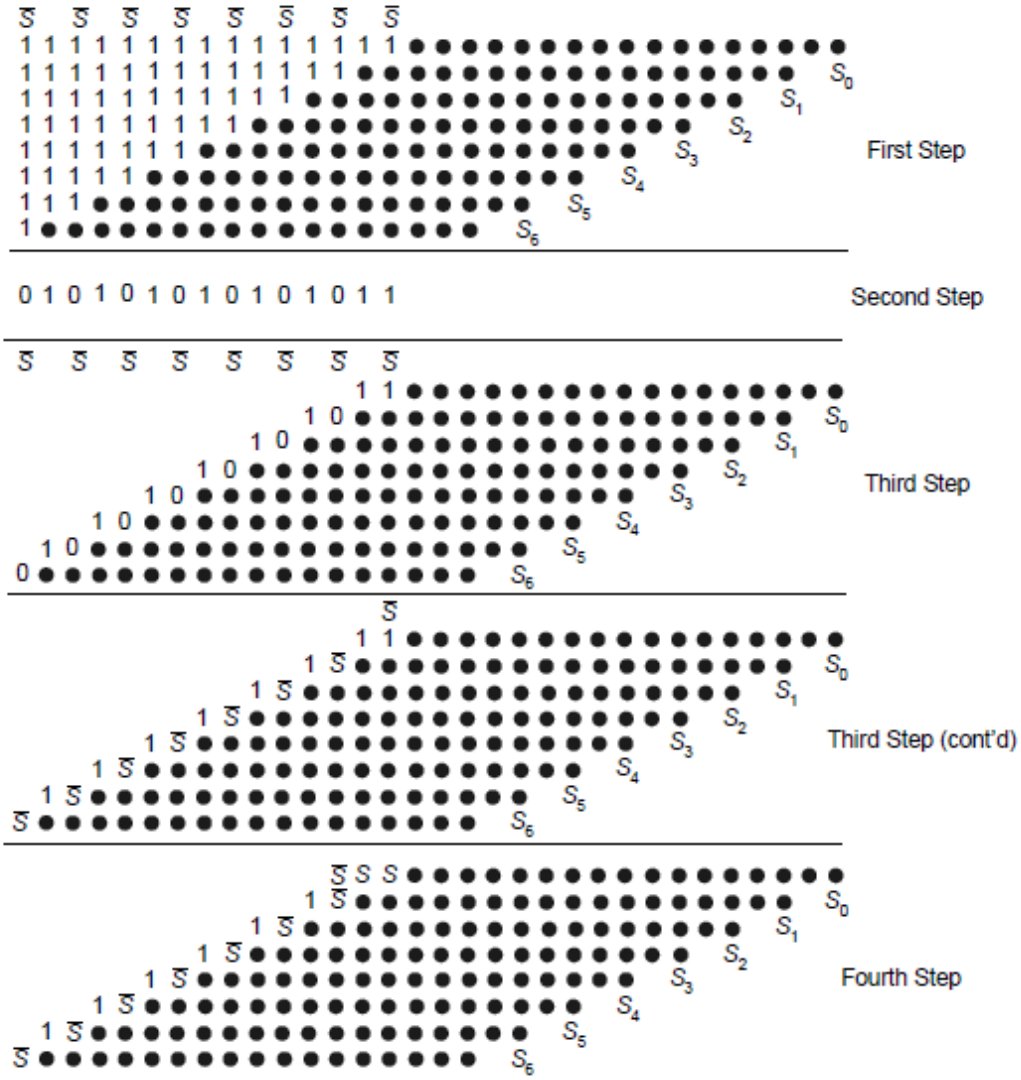


Figure 3.3: Reducing sign extension in a partial product tree: Each step in this figure has a dot diagram of the partial product tree with the sign extension bits. The S7 bit is omitted to reduce the height of the tree and will be added at a later pipe stage. The first step involves assuming all the partial products are negative, and sign extending with 1's. If the partial product turns out to be positive, the string of ones can be converted back to leading 0's by adding a 1 in the least significant position. This is achieved by adding the inverted sign bit, S. S = 0 for a negative partial product, thus keeping the string of ones; and S = 1 for a positive partial product, thus turning the extension into a string of zeroes. The second step adds all the sign extension bits into a single constant. In the figure this step omits the other dots in the partial product tree. The third step fills in this constant with the rest of the bits. In the hardware, these 1's in the constant will be hardwired to V_{dd}. The fourth step reduces the height of the tree by absorbing the S into the first partial product. After the fourth step, the partial product tree is ready to be accumulated.

Booth Encoders and Partial Product Selectors

This section describes how to generate the partial product tree in the fourth step of Figure 3.3. Each partial product row is generated with a Booth encoder and a partial product row selector (Booth selector). Its block diagram is shown in Figure 3.4. The truth table for the Booth encoder is in Table 3.1, its equations expressed in Equations 3.1−3.4, and its gate schematic is in Figure 3.5. The encoder has three inputs and produces four outputs, which are select signals for the partial product selector. The select signals from the Booth encoder have drivers at the end to drive the long wire and large amount of logic in the partial product row selector.

$$\text{selectm} = m_1 \oplus m_2 \quad 3.1$$

$$\text{select2m} = \bar{m}_2 . m_1 . m_0 + m_2 . \bar{m}_1 . \bar{m}_0 \quad 3.2$$

$$\text{select0} = \bar{m}_2 . \bar{m}_1 . \bar{m}_0 + m_2 . m_1 . m_0 \quad 3.3$$

$$\text{sign} = m_2 \quad 3.4$$

The partial product row selector contains 17 partial product bit selectors and an inverter at the end to invert the sign bit. The truth table for one partial product bit is in Table 3.2, its equation in Equation 3.5.

$$\text{ppbit} = \text{sign} \oplus (\text{selectm} . \text{mcand} + \text{select2m} . (2 . \text{mcand}) + \text{select0} . \text{Gnd}) \quad 3.5$$

It takes as inputs the four output signals from the Booth encoder and two bits of the multiplicand, and it outputs one bit. The logic for this cell is a 3-to-1 selector that chooses between the multiplicand(mcand), twice the multiplicand($2 . \text{mcand}$), or zero. There is also the option to invert the result that this selector chooses. The sign control signal either inverts the partial product bit or not. In our implementation of the MAC unit, the partial product bit selector is arrayed horizontally 17 times to build a block called a partial product row selector, and the detailed block diagram is in Figure 3.4. The largest result that this row can output is twice the multiplicand (at most a left shift of one), hence the cascade of 17 cells. In Table 3.2 there are four inputs (for a maximum of 16 unique combinations), but there are only eight entries because the assertion of a select signal is mutually exclusive amongst the other two (as described by the Booth encoder). Only selectm, select2m or select0 can be asserted at any time.

Table 3.1: Truth table for the Booth encoder

Inputs			**Outputs**			
m_2	m_1	m_0	$selectm$	$select2m$	$select0$	$sign$
0	0	0	0	0	1	0
0	0	1	1	0	0	0
0	1	0	1	0	0	0
0	1	1	0	1	0	0
1	0	0	0	1	0	1
1	0	1	1	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	1	1

Table 3.2: Truth table for the partial product bit selector

Inputs				**Outputs**
$selectm$	$select2m$	$select0$	$sign$	$pp\ bit$
0	0	1	0	0
0	1	0	0	$2.mcand$
1	0	0	0	$mcand$
0	0	1	1	1
0	1	0	1	$\overline{2.mcand}$
1	0	0	1	$\overline{mcand}$

3.3 Partial Product Accumulation

After the partial products are generated, the next step is to accumulate them. The goal is to reduce the tree to carry-save format, which is two rows of bits. When the tree is in this format, a carry propagate adder (as discussed in Chapter 2) produces the final sum. We primarily use 4:2 compressors because of their speed and regularity in layout. The partial product tree in the fourth step of Figure 3.3 is reduced with two rows of 4:2 compressors (which can cover a tree of height 8). The S7 bit makes the height of the tree 9 bits high, and two rows of 4:2's would not be sufficient, so we delay the addition of this bit to a later pipe stage. These two rows of 4:2's, the Booth encoder, and partial product row selectors are in the first pipe stage. Its outputs are fed into registers for the next pipe stage. There are a total of 96 registers at the end of the first pipe stage (92 registers for the outputs of the 4:2 compressors, 2 registers for the S sign extension bits, and 1 register

each for S3 and S7). The second pipe stage continues the tree reduction. The tree now has a height of five rows (four rows from the outputs of the 4:2 compressors from the first pipe stage, and one row for the S7 bit). One row of 26 4:2 compressors is used for the first four rows of bits, which compresses into two rows of bits. It is possible to use a row of 32 4:2 registers in this pipe stage, but we observe that the bits in the least significant portion of this pipe stage are already in carry-save format. After the 4:2 compressor reductions, there are now three rows in the tree with the S7 bit. The outputs of the 26 4:2 compressors are fed into a row of half adders to further reduce the height to two rows. The row of half adders (used as carry-save adders) reduces the height so the S7 bit will fit into carry-save format for the third pipe stage. There are a total of 62 registers at the end of the second pipe stage (35 registers for the bits from the half adders, 26 registers for the outputs of the 4:2 compressors that aren't fed into half adders, and 1 register for the S7 bit). Figure 3.4 shows the reduction of the partial product tree into carry-save format.

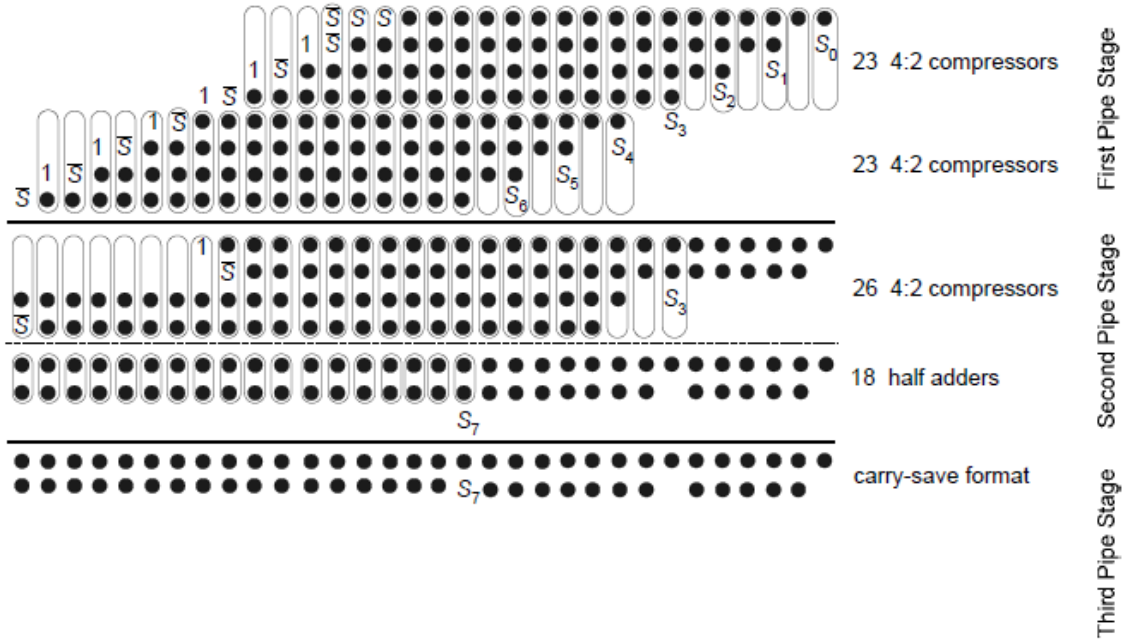


Figure 3.4: Partial Product tree reduction with 4:2 compressors and half adders

3.4 Final Stage of MAC Unit

The third and final stage of the MAC unit completes the accumulation of the partial product tree. It contains a row of 40 AND gates, a row of 40 full adders, a 40-bit carry select adder, and a 40-bit right shifter. During the execution of this final stage, the result of either a multiply/MAC instruction or a shift instruction is selected, thus the

propagation delays for these instructions occur in parallel and the critical path is one of the two. This stage also contains 40 registers for the accumulator, and 16 registers for the multiplication result. The dot diagram for these three rows is in Figure 3.5. The outputs of the full adders are in carry-save format and are fed into the 40-bit adder for the final addition. Since two 40-bit quantities are a large amount to add, we implement the carry-propagate adder with a carry-select adder to achieve a low propagation delay.

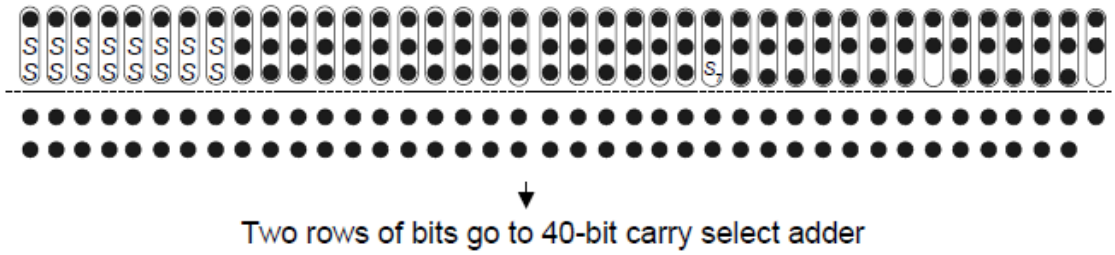


Figure 3.5: Dot diagram of bits in the third pipe stage

3.5 Summary

This chapter presents the MAC unit design and implementation for the high speed DSP applications. It utilizes Booth encoding, 4:2 compressors for partial product tree reduction, and a 40-bit carry select adder for final accumulation.

Chapter 4

ANALYSIS & IMPLEMENTATION

4.1 Adder Implementation

Implementation of 32-bit addition using ripple carry, carry select, carry look-ahead and parallel prefix adders has been done using ModelSim 6.1e tool. The designs of each type are as described in section 2.1. Following are the speed and power results using Design Vision compiler of Synopsys using 180 nm technology at 25MHz.

4.1.1 Ripple Carry Adder

The ripple carry adder is constructed by cascading FA blocks in series. The carry out of one stage is fed directly to the carry in of the next stage. For n-bit parallel adder, it requires n full adders. As the delay increases with the bit length these adders are not very much efficient when large bit numbers are used.

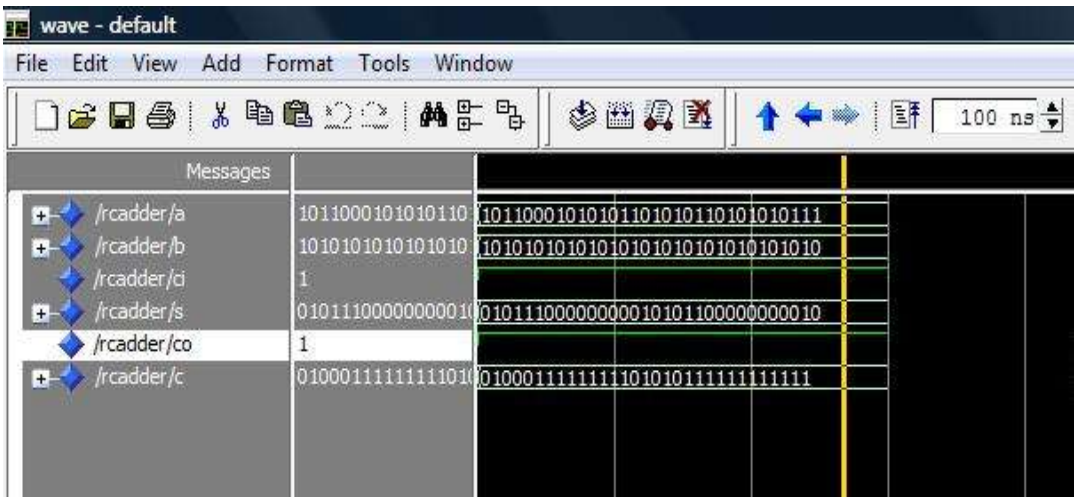


Figure 4.1: Simulation result of 32-bit Ripple Carry Adder

4.1.2 Carry Skip Adder

These adders are composed of ripple carry adder blocks of fixed size and a carry skip chain. The sizes of the blocks are chosen so as to minimize the length of the carry. For designing 32-bit adder, the carry chain block is divided into 8 blocks each comprising of 4-bit RCA and carry skip chain implementing using FA's and

muxes respectively.

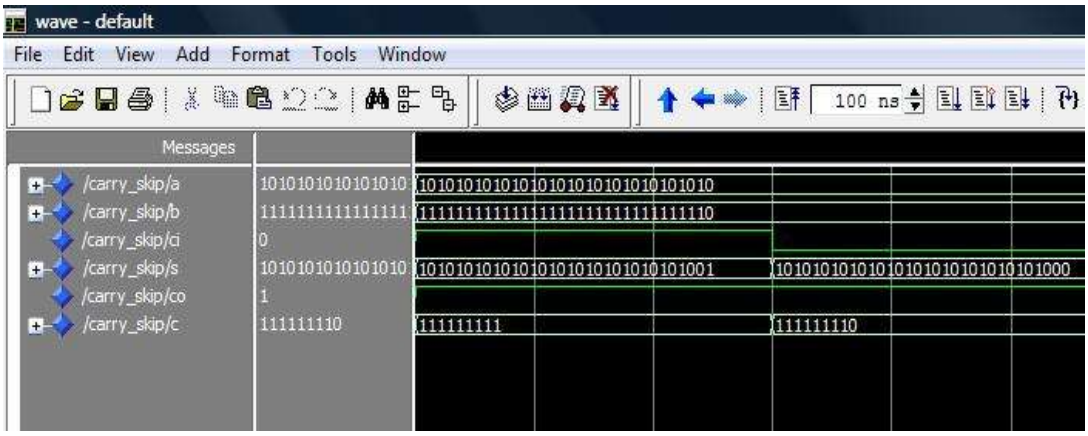


Figure 4.2: Simulation result of 32-bit Carry Skip Adder

4.1.3 Carry Look-ahead Adder

As it takes a long time for the carry to ripple through each stage of the FA in the ripple carry adder, to make the circuit faster, the carry is split into two portions: one is called propagate carry and the other one is generate carry. For each stage of the FA, a propagate carry bit P and a generate carry bit G are added. As P and G having simpler functions compared to the carry out of FA, they can be generated faster than this. In order to produce the propagate and generate carry for the group as soon as possible, the 4-bits of propagate and generate carries are grouped into one carry look ahead unit (CLAU). The design of the 32-bit CLAU unit is shown in Figure 4.9.

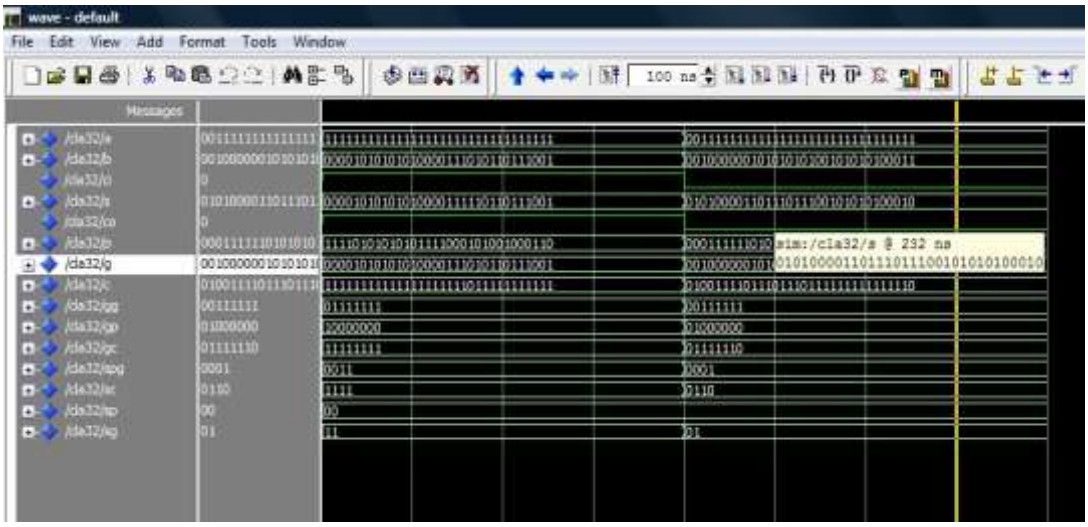


Figure 4.3: Simulation result of 32-bit Carry Look-ahead Adder

4.1.4 Carry Select Adder

This adder is composed of two 4-bit ripple carry adders per section in which both sum and carry are calculated for the two alternatives of the input carry '0' and '1'. The carry out of each section determines the carry in of the next section, which then selects the appropriate ripple carry adder where the first section has a carry in of '0'.

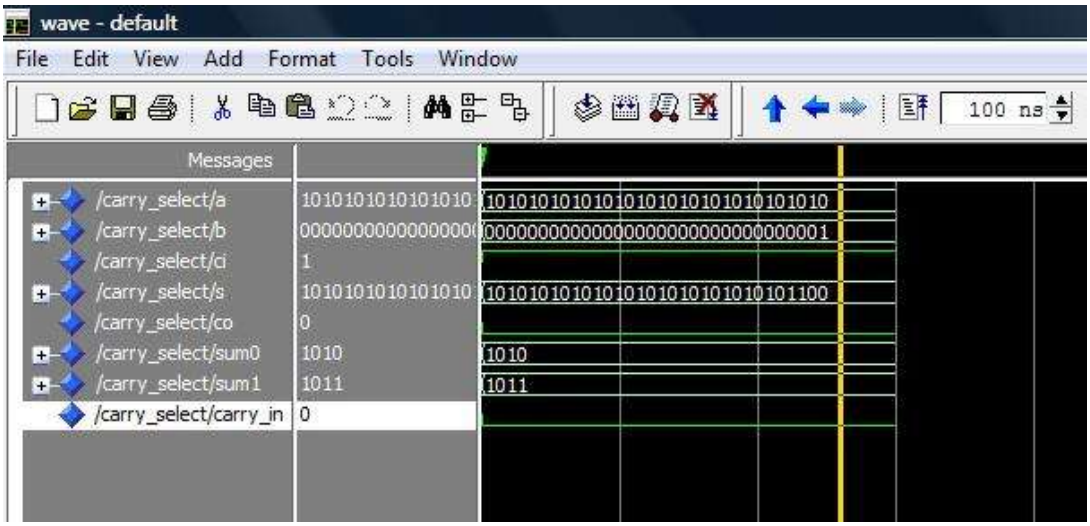


Figure 4.4: Simulation result of 32-bit Carry Select Adder

4.1.5 Parallel Prefix Adder

The equations of the well-known CLA adder can be formulated as a parallel prefix problem by employing a special operator "●". This operator is associative hence it can be implemented in a parallel fashion. A Parallel Prefix Adder (PPA) is equivalent to the CLA adder but the two differ in the way their carry generation block is implemented. The parallel prefix adder employs the 3-stage structure of the CLA adder where the improvement is in the carry generation stage which is the most intensive one.

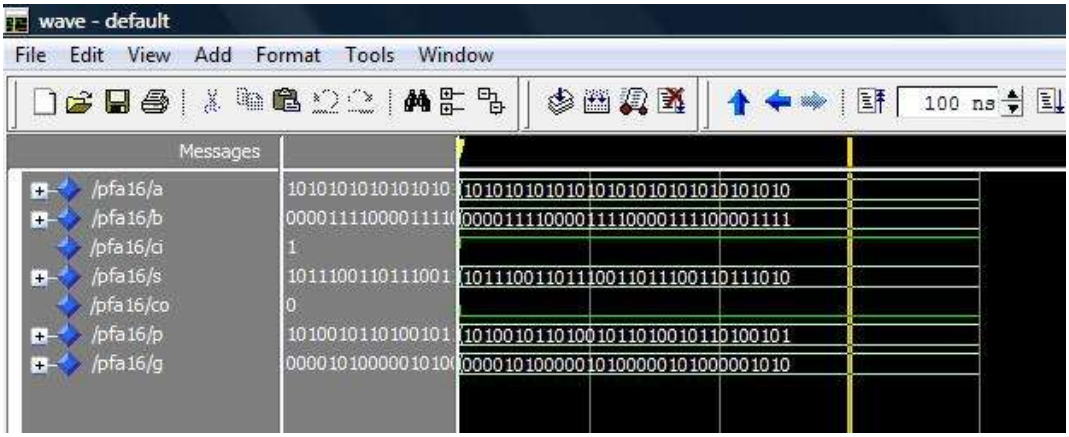


Figure 4.5: Simulation result of 32-bit Parallel Prefix Kogge-Stone Adder

Table 4.1: Comparison of 32-bit adders for Various Performance Measures

	Ripple Carry Adder	**Carry Skip Adder**	**Carry Look-ahead Adder**	**Carry Select Adder**	**Parallel Prefix Adder**
Power	123.90 µW	134.62 µW	183.05 µW	248.55 µW	254.9 µW
Delay	3.06 ns	4.32 ns	0.57 ns	2.67 ns	0.41 ns

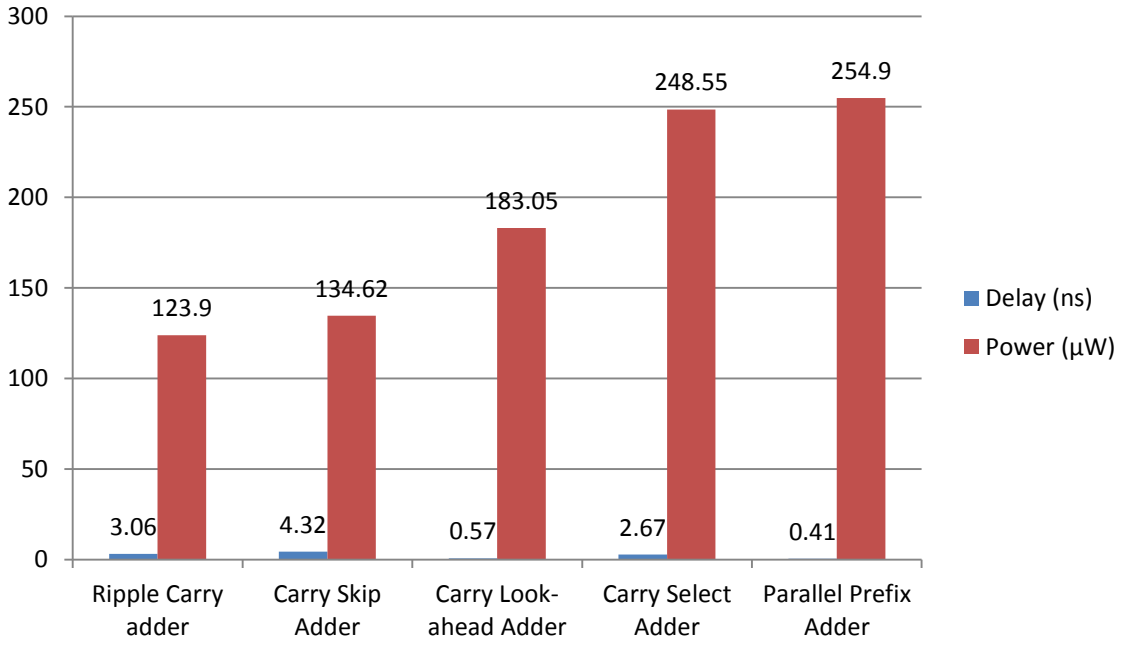


Figure 4.6 Performance Analysis of various adders

4.2 Multiplier Architecture

Multiplier architecture consists of MBR, Wallace tree (4:2 compressor) and a carry look-ahead adder. The first block generates partial products on the basis of Booth Algorithm by considering three consecutive bits of multiplier. Partial products so formed are added using 4:2 compressors which have a regular structure. Wallace tree include half adders, full adders, 4:2 carry save adders in the same stage, reducing partial product at the same time. Block diagram of this architecture is shown in Figure 4.7. The outputs of Wallace tree i.e. final sum and carry are added using carry look-ahead adder and adder give the final product.

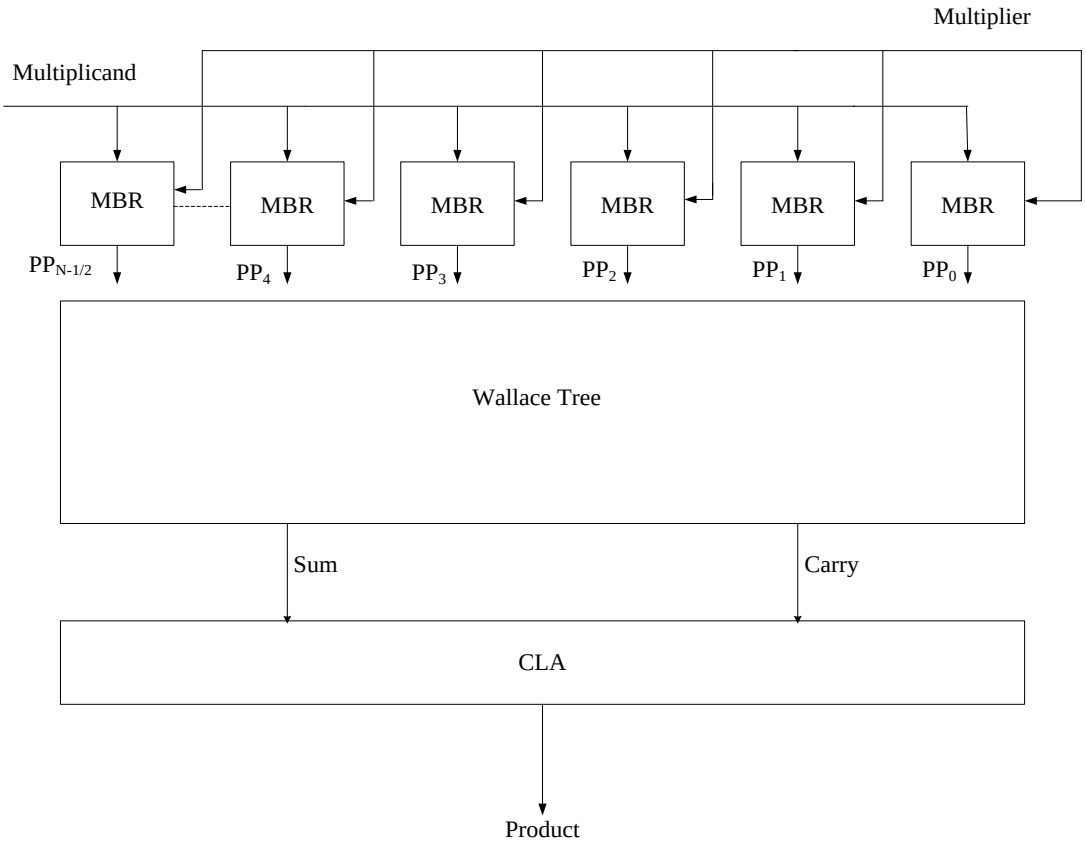


Figure 4.7: Block diagram of Booth encoded Wallace tree multiplier

Implementation of 16 × 16 Bit Modified Booth Wallace Multiplier

In 16 × 16 bit multiplier, eight MBR generate eight partial products of 17-bit. The multiplicand comes from the left to go into eight Modified Booth recoders. Each recoder takes 3 bits from the multiplier with a '0' appended at the right end. MBR has a 17-bit output. Each MBR output is shifted to its correct position and sign extended. Block

diagram of this multiplier is shown in Figure 4.8.

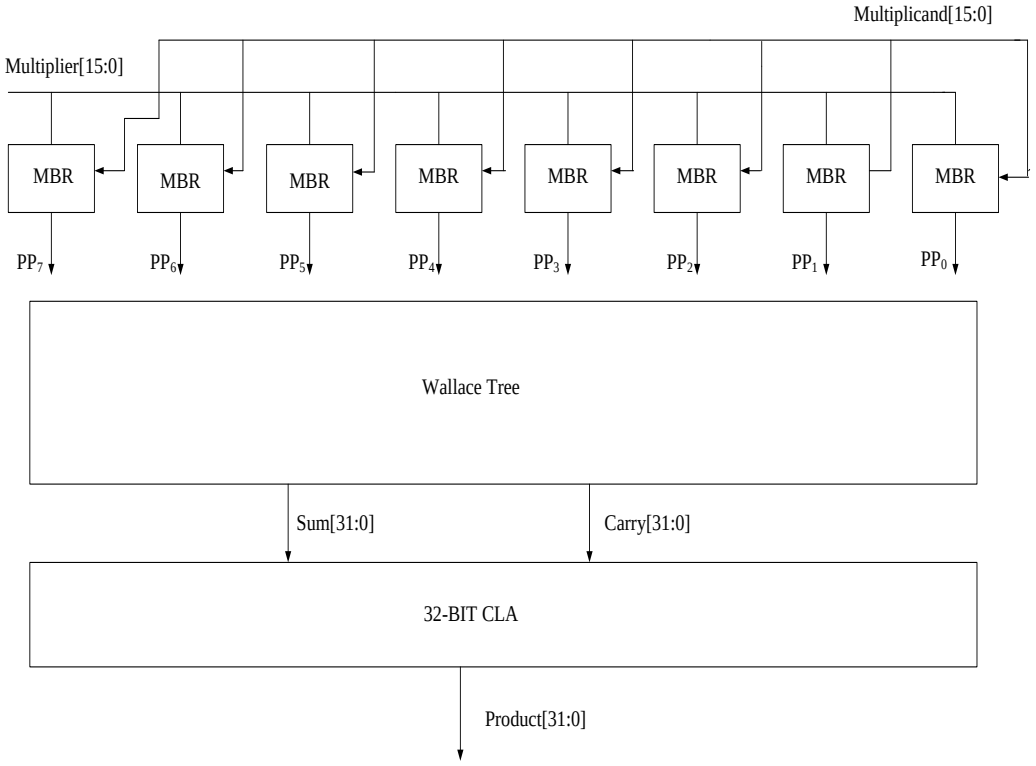


Figure 4.8: Block diagram of 16 × 16 bit modified Booth Wallace multiplier

Wallace tree includes three rows of 4:2 compressors. The first row 4:2 compressors adds partial products PP_0, PP_1, $PP_2$and PP_3, the second row adds partial products PP_4, PP_5, PP_6 and PP_7 and the third one add the sum outputs from the first rows and the carry bits from 4:2 compressors in the right column. Figure 4.9 shows the 4:2 compressor organization for adding eight partial products. The carry and sum outputs of last row of 4:2 compressor are added with 32-bit CLA with the carry output bits shifted left one bit position to add with the sum bits. Figure 4.10 shows 32-bit CLA which requires three CLC levels and eleven carry look-ahead units. This adder was designed by adding a single third-level CLC and one OC circuit to two 16-bit CLAs minus their respective OC circuits. Second level CLC uses the group P and G outputs from the first level CLCs as inputs and provides the carry outputs C_4, C8, C12, C20, C24 and C_{28}. Third level CLC uses the group P and G outputs from the second level CLCs as inputs and provides the carry output C_{16}. Also, the P and G group outputs from the third-level CLC circuit cover carry generation and propagation for all 32 bits and, by using an OC circuit, can combine these two outputs with C_0 to produce carry output C_{32}. Thus final 32-bit product is obtained.

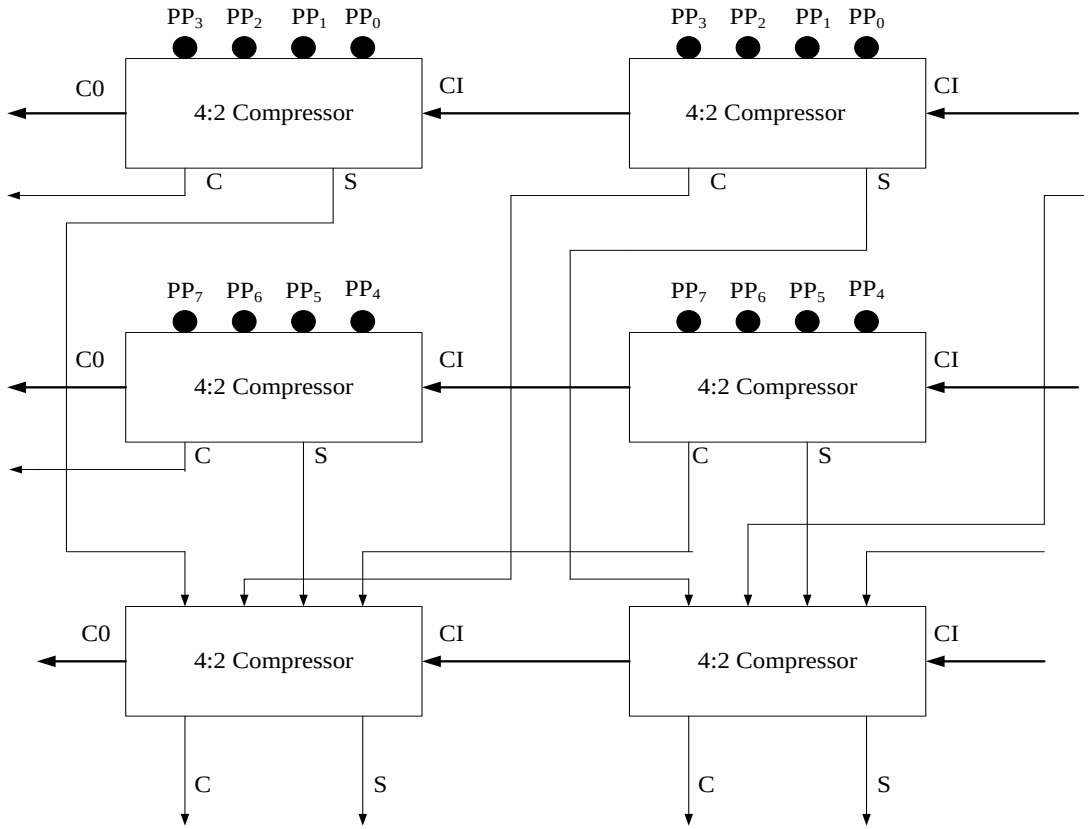


Figure 4.9: 4:2 Compressor organization for eight partial products

Implementation of 32 × 32 Bit Modified Booth Wallace Multiplier

In 32 × 32 bit multiplier, sixteen MBR generate sixteen partial product of 33-bit. Sixteen PP are reduced using seven rows of 4:2 compressors. Figure 4.12 shows the 4:2 compressor organization for adding sixteen partial products. Block diagram of 32 × 32 bit Booth Wallace multiplier is shown in Figure 4.11.

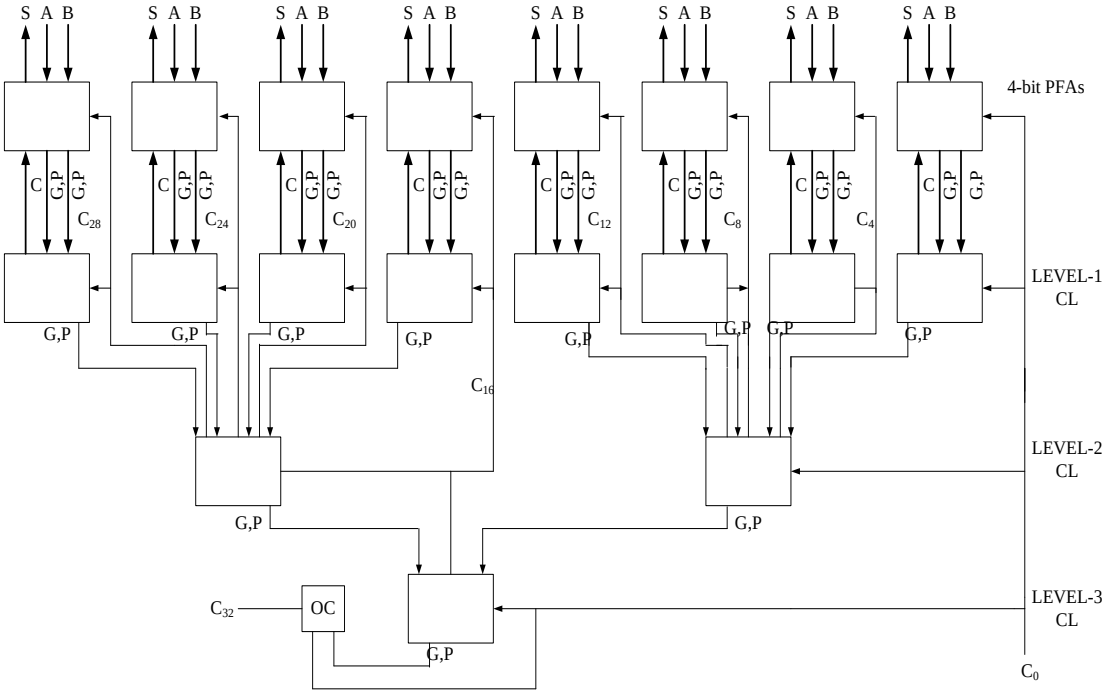


Figure 4.10: 32-bit CLA

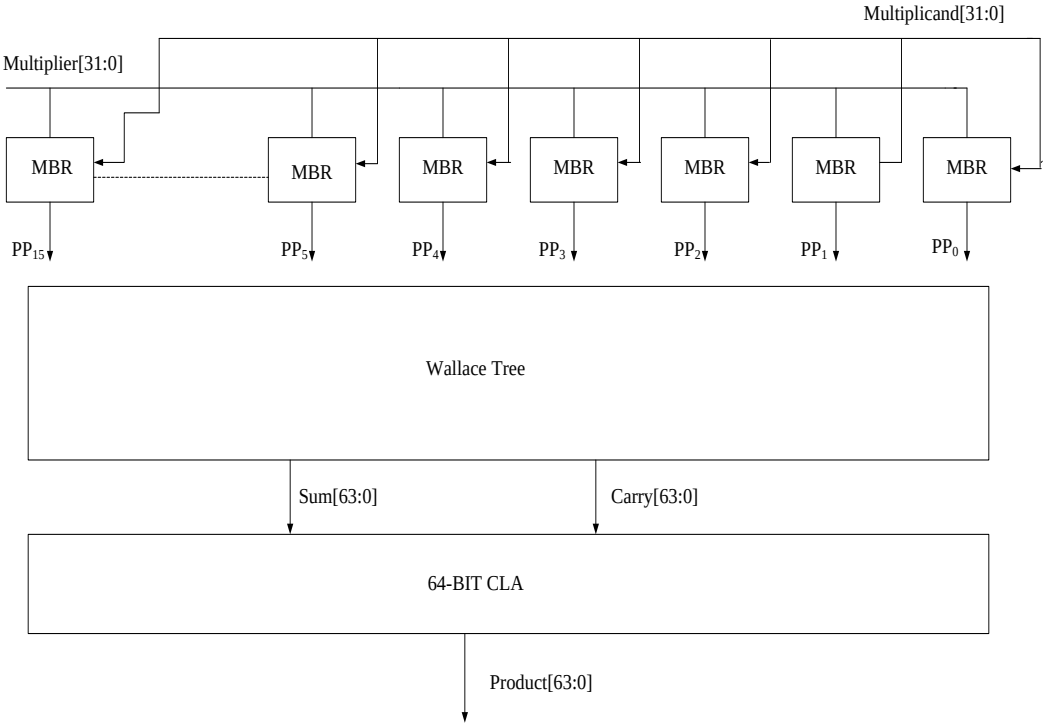


Figure 4.11: Block diagram of 32 × 32 bit modified Booth Wallace multiplier

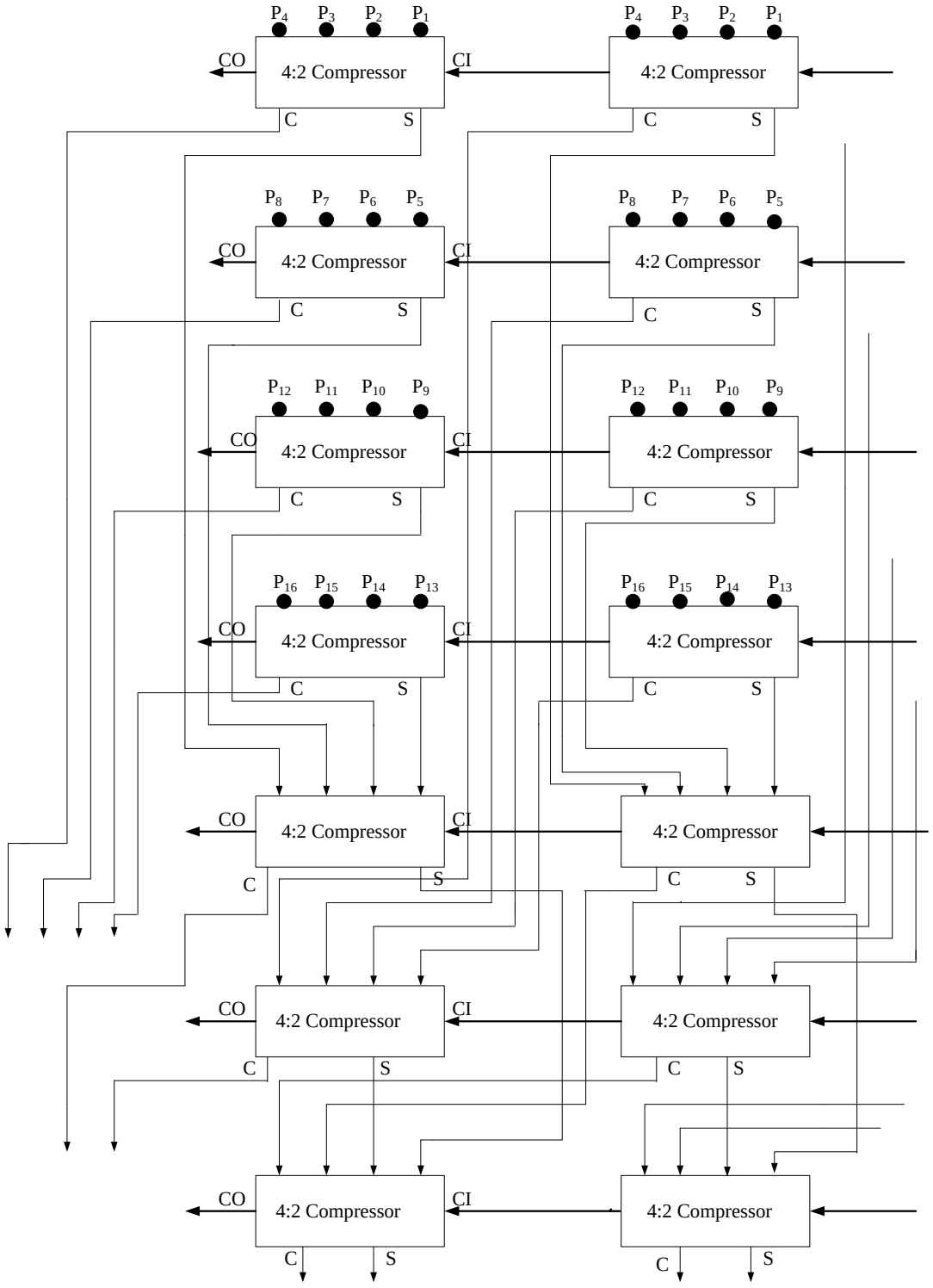


Figure 4.12: 4:2 compressor organization for sixteen partial products

The carry and sum outputs of last row of 4:2 compressor are added with 64-bit carry look-ahead adder with the carry output bits shifted left one bit position to add with the sum bits. Figure 4.13 shows 64-bit CLA.

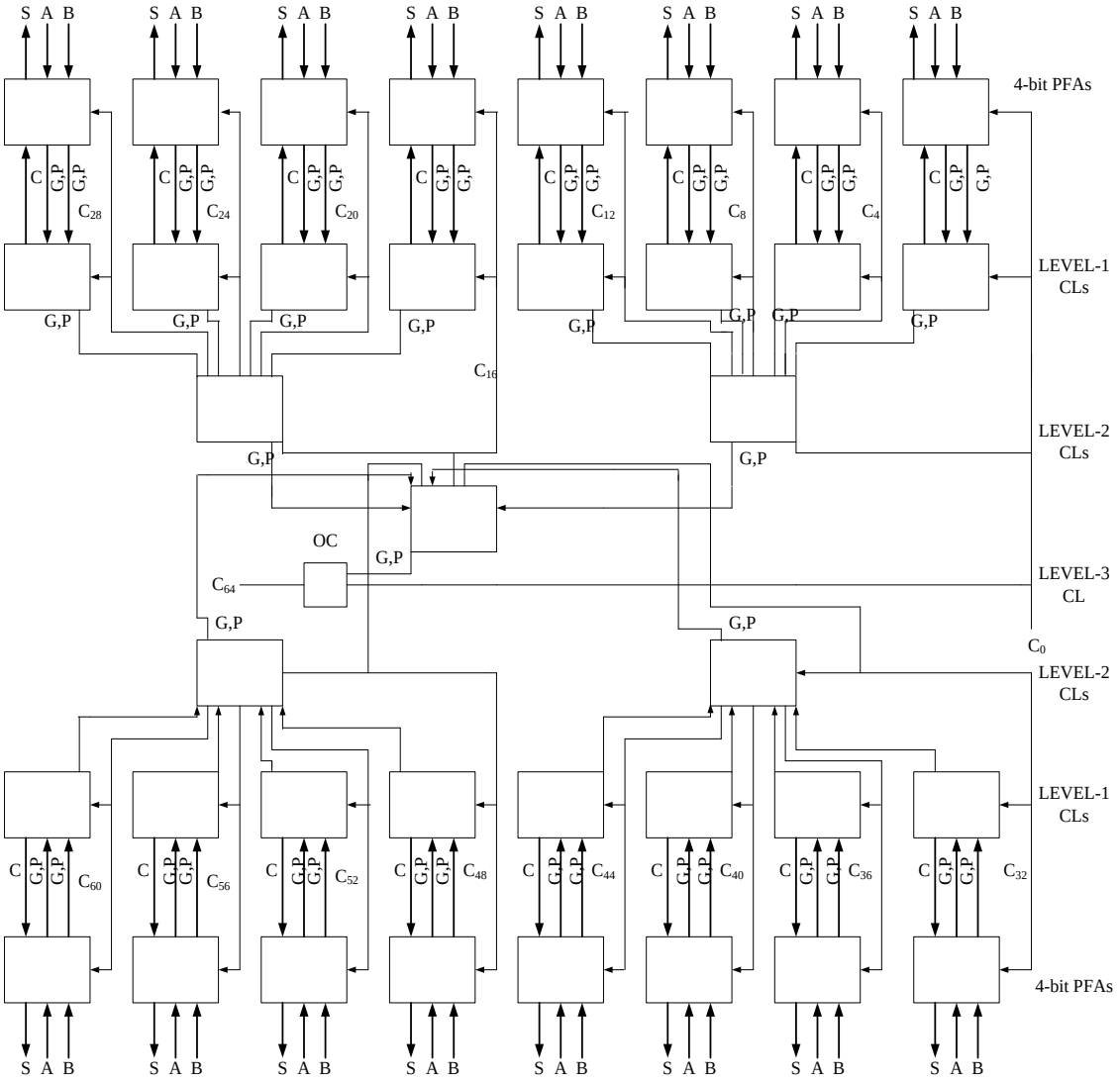


Figure 4.13: 64 bit CLA

This adder was designed by adding a single third-level CLC and one OC circuit to four 16-bit CLAs minus their respective OC circuits. Third-level CLC uses the group *P* and *G* outputs from the four second-level CLCs as inputs and provides the carry outputs C_{16}, C_{32}, and C_{48}. Also, the *P* and *G* group outputs from the third-level CLC circuit cover carry generation and propagation for all 64 bits and, by using an OC circuit, can combine these two outputs with C_0 to produce carry output C_{64}.

4.3 Results of Multiplier Blocks

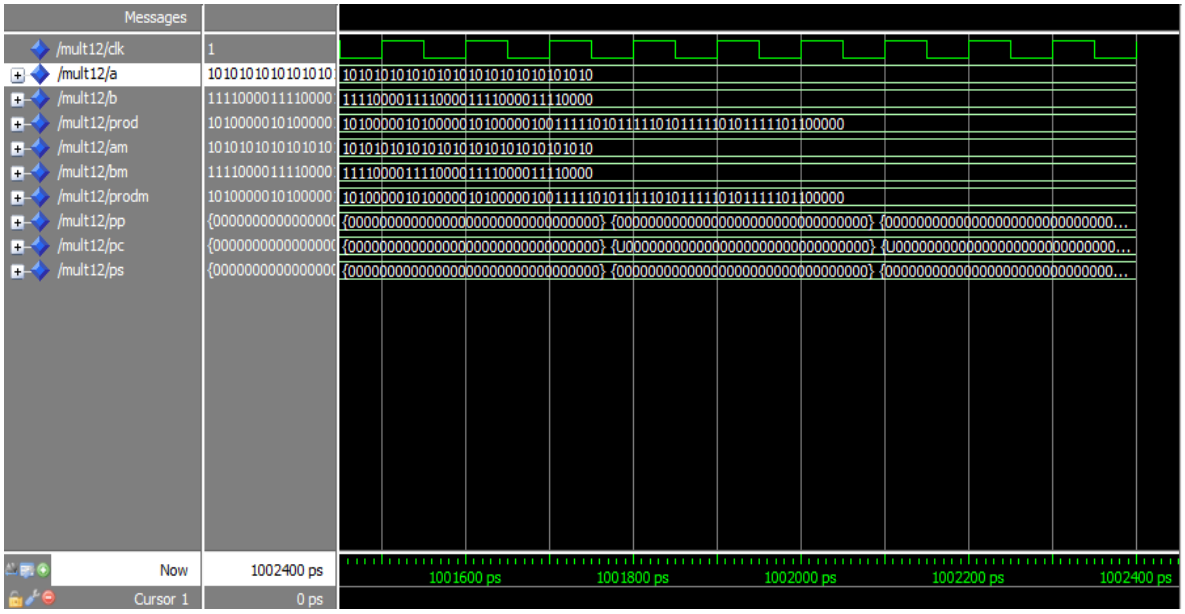


Figure 4.14: Simulation result of 32×32-bit Array multiplier

Description:

a:		Input data 32-bit
b:		Input data 32-bit
clk:		clock input
prod:		Output data 64-bit
a	=	10101010101010101010101010101010
b	=	11110000111100001111000011110000
prod	=	10100000101000001010000010011111
		01011111010111110101111101100000

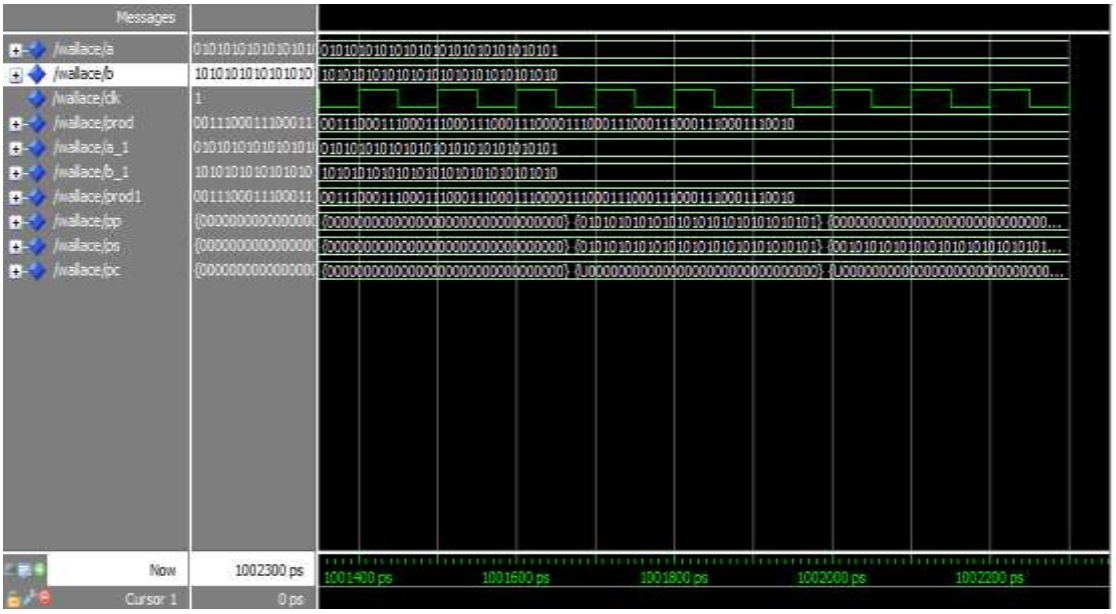


Figure 4.15: Simulation result of 32×32-bit Wallace multiplier

Description:

a:		Input data 32-bit
b:		Input data 32-bit
clk:		Input clock
z:		Output data 64-bit
a	=	01010101010101010101010101010101
b	=	10101010101010101010101010101010
z	=	0011100011100011100011100011100011100011100011100011100011100010

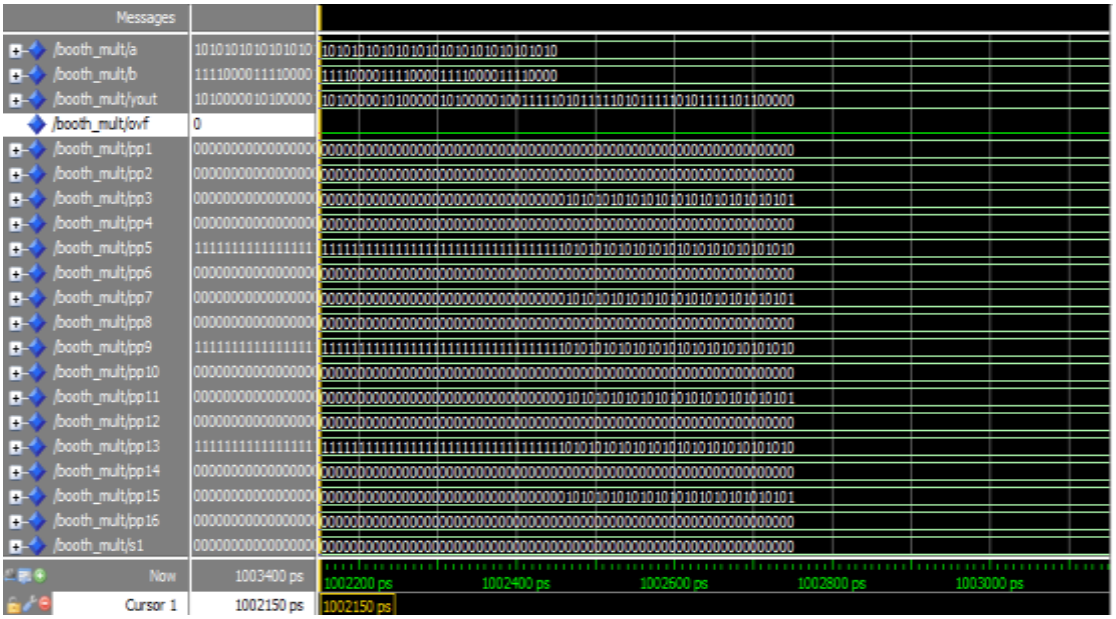


Figure 4.16: Simulation result of 32×32-bit Booth multiplier

Description:

a:		Input data 32-bit
b:		Input data 32-bit
yout:		Output data 64-bit
a	=	10101010101010101010101010101010
b	=	11110000111100001111000011110000
yout	=	10100000101000001010000010011111
		01011111010111110101111101100000

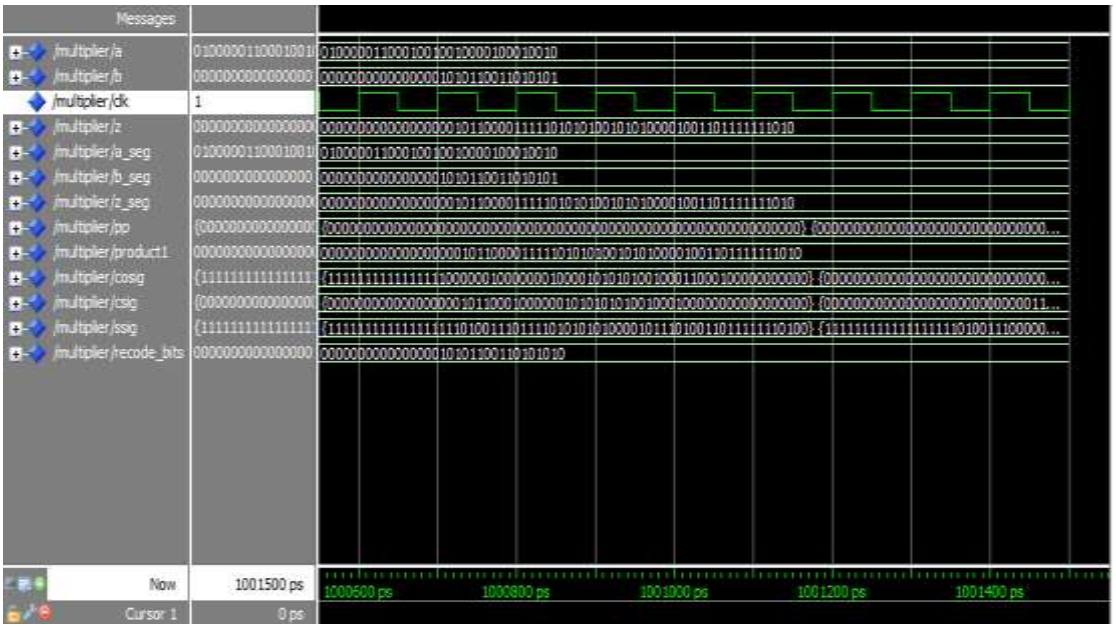


Figure 4.17: Simulation result of 32×32-bit modified Booth Wallace multiplier

Description:

a:		Input data 32-bit
b:		Input data 32-bit
clk:		clock input
z:		Output data 64-bit
a	=	01000001100010010010000100010010
b	=	00000000000000001010110011010101
c	=	00000000000000000010110000111110
		10101001010100001001101111111010

Table 4.2: Comparison of 32×32 bit multipliers for Various Performance Measures

	Array multiplier	**Booth multiplier**	**Wallace multiplier**	**Booth Wallace multiplier**
Area(LUTs)	3456	4067	4378	4167
Delay	36.34ns	22.45ns	15.76ns	13.38ns
Power	89mW	98mW	108mW	102mW

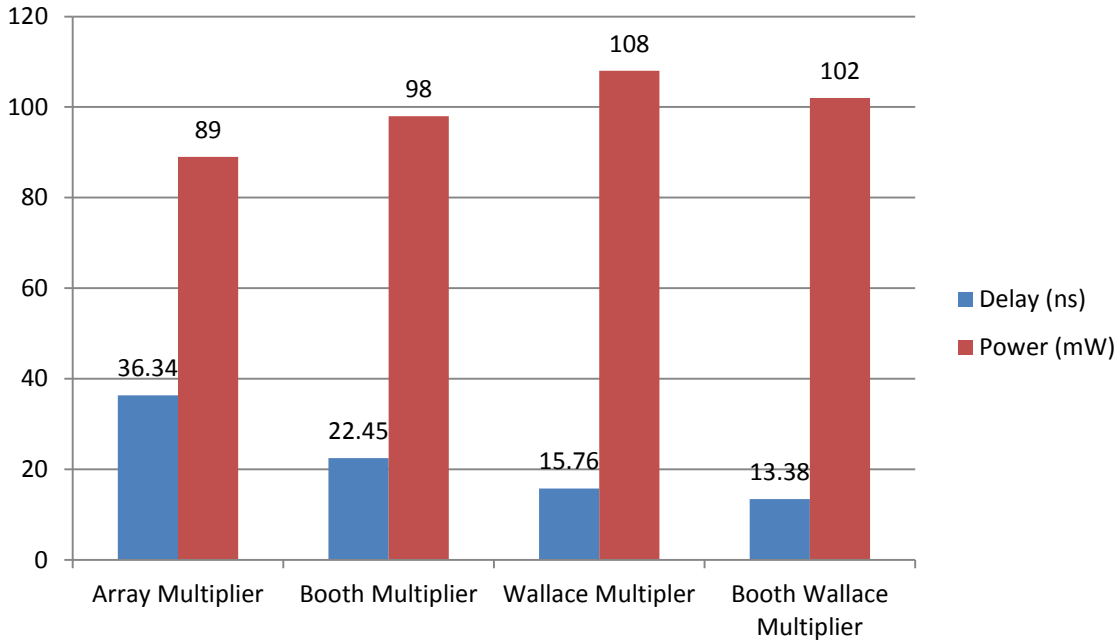


Figure 4.18 Performance Analysis of various multipliers

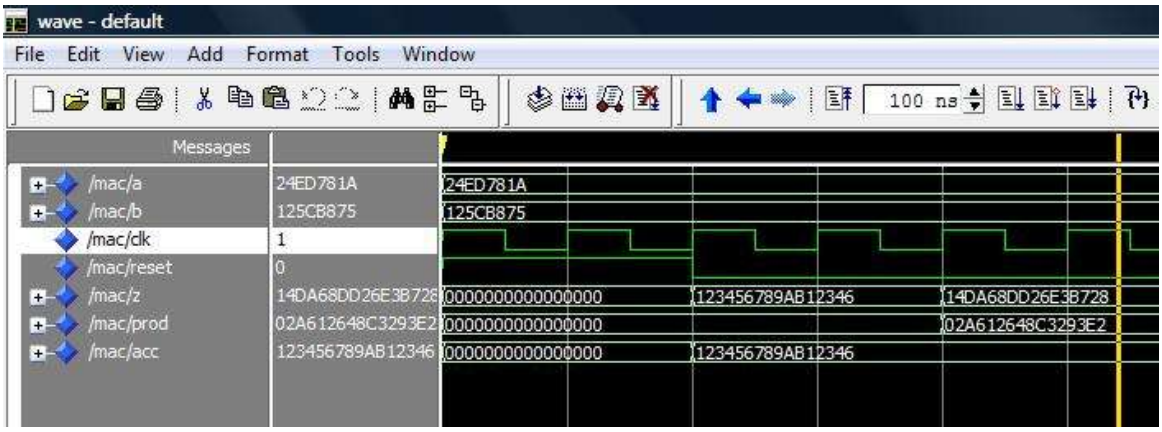


Figure 4.19: Simulation result of 32-bit MAC Unit

Chapter 5
CONCLUSION

The designs of various adders and multipliers have been implemented on Spartan XC3S500-5-FG320. The computation delay for 32-bit CLA, used in the design of MAC Unit is 0.57 ns and 8×8-bit, 16×16-bit and 32×32-bit Booth Wallace multiplier are 8.78 ns, 10.13 ns and 13.38 ns respectively. The performance of 32×32-bit multiplier increases by employing 3-stage pipelining technique. For 32×32 MAC, the gate delay in the architecture is 7.556 ns with nearly 18% device utilization (number of slices: 481 out of 2352 (20%) and number of 4-input LUTs: 850 out of 4704(18%).

By employing pipelining, speed increases by the factor of 1.7 with penalty on area increasing by factor of 1.11. However, speed improvement ratio is larger than the area increase ratio. Hence, it is concluded that pipelining improves performance of multiplier. Power of multiplier is reduced by using low power Modified Booth Recoder (MBR). Further, comparison of different adders and multipliers is done on various performance measures. All comparison is based on the synthesis reports keeping one common base for comparison. That means we target same FPGA device (part number and speed grade) with same design constraints implied for the synthesis of each multiplier. Delay of Wallace tree and Booth Wallace multiplier is almost same and is the least but power consumption of Wallace tree multiplier is high compared to Booth Wallace multiplier. Power consumption of Booth Wallace is more than array and Booth Multiplier. Hence, Booth Wallace multiplier and CLA adder for accumulation are used in our design of 32×32-bit MAC Unit.

Various attempts were made to achieve significant improvement over existing data formats. An extensive simulation and comparison between the data formats was made and results were summarized for analysis. All the designs are made at architectural level keeping one common base for comparison and the same can be done at circuit level where variations in V_t can be made for low power applications.

Future Scope

As an attempt to develop, arithmetic algorithm and architecture level optimization techniques for low power high-speed multiplier design, techniques presented in this thesis has achieved good results. However, there are limitations in this work and several future research directions are possible as follows:

1. One possible direction is radix higher than radix-4 recoding. Only radix-4 recoding is considered in this thesis work as it is a simple and popular choice. Higher-radix recoding further reduces the number of PPs and thus has the potential of power saving.
2. In order to enhance the performance, higher order compressors like 7:2, 9:2 can be used to accumulate the partial products.
3. Deep level pipeline architecture can be used for speed improvements.

All the designs are made at architectural level keeping one common base for comparison and the same can be done at circuit level where variations in V_t can be made for low power applications.

The pipeline technique is widely used to improve the performance of digital circuits. A simple calculation reveals that close to 33 gate-delays are needed to complete a multiplication of 32-bit operands without pipelining. From this number nine gate delays belong to the Wallace structure (4:2 compressor), ten gate delays to the MBR and fourteen gate delays belong to the 64-bit carry look-ahead adder. Now, with addition of three registers, each carrying one-gate delays we come up with overall 36 gate delays. Hence, if we split this delay into three, then each stage of the pipeline takes about twelve gate delays to complete.

REFERENCES

[1] A. P. Chandrakasan, S. Sheng, And R. W. Brodersen, "Low Power CMOS Digital Design," IEEE Journal of Solid-state Circuits, vol. 27, no. 04, pp. 480-484, April 1999.

[2] Kaushik Roy, Sharat C. Prasad, Low-Power CMOS VLSI Circuit Design, John Wiley & Sons, Inc, 2000.

[3] I. Koren, Computer Arithmetic and Algorithms, Brookside Court Publishers, 1998.

[4] M.J. Schulte, P.I. Balzola, A. Akkas, and R.W. Brocato, "Integer Multiplication with Overflow Detection or Saturation", IEEE Trans. Computers, vol. 49, no. 7, July 2000.

[5] C.S. Wallace, "A suggestion for a fast multiplier", IEEE Trans. Electron. Comp., vol. EC-13, pp 14-17, Feb. 1964.

[6] Kihak Shin, Ik Kyun Oh, Sang Min, Beom Seom Ryu, Kie Young Lee and Tae Won Cho "A Multi-Level Approach to Low Power Mac Design" IEEE Trans. VLSI systems, vol. 48 , pp 361- 763, 1999.

[7] Shanthala S, Cyril Prasanna Raj, Dr. S. Y. Kulkarni, Design and VLSI Implementation of Pipelined Multiply Accumulate Unit, Second International Conference on Emerging Trends in Engineering and Technology, ICETET-09.

[8] F. Elguibaly, "A fast parallel multiplier-accumulator using the modified Booth algorithm", IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing, vol. 47, pp. 902-098, Sept. 2000.

[9] H. Murakami, et al." A multiplier-accumulator macro for a 45 MIPS embedded RISC processor," IEEE J. Solid-State Circuits, vol. 31, pp. 1067-1071, July 1996.

[10] R. K. J. Raghunath, et al. "A compact carry-save multiplier architecture and its applications," Proc. IEEE 40th Midwest Symposium. Circuits and Systems, vol. 2, pp. 794-797, Aug. 1997.

[11] Li-Hsun Chen; Chen, O.T.C., "A multiplication-accumulation computation unit with optimized compressors and minimized switching activities" Circuits and Systems, 48th Midwest Symposium, MWSCAS.2005, Page(s):1223 - 1226, 2005.

[12] Ohsang Kwon, K. Nowka and E. E. Swartzlander, “A 16-bit x 16-bit MAC design using fast 5:2 compressors,” Proc. of IEEE International Conference on Application Specific Systems, Architectures, and Processors, pp. 235 –243, 2000.

[13] Ayman Fayed, Walid Elgharbawy, and Magdy Bayoumi, “A merged multiply accumulate for high-speed signal processing application,” ICASSP IEEE 2004.

[14] Ayman Fayed , Walid Elgharbawy, and Magdy Bayoumi, “A data merging technique for high-speed low-power multiply accumulate units,” IEEE 2002.

[15] Young-Ho Seo, Member, IEEE, and Dong-Wook Kim, Member, IEEE, “A New VLSI Architecture of Parallel Multiplier Accumulator Based on Radix-2 Modified Booth Algorithm”, IEEE transactions on very large scale integration (VLSI) systems, vol. 18, no. 2, February 2010.

[16] A. R. Omondi, Computer Arithmetic Systems. Englewood Cliffs, NJ: Prentice-Hall, 1994.

[17] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, Digital Integrated Circuits, A Design Perspective, Prentice Hall, Upper Saddle River,NJ, 2003.

[18] A. Chandrakasan, W. J. Bowhill, and F. Fox, Design of High-Performance Microprocessor Circuits, IEEE Press, Piscataway, NJ, 2001.

[19] A.Weinberger and J. Smith, “A logic for high-speed addition," in National Bureau of Standards, 1958, pp. 3-12.

[20] I. Koren, Computer Arithmetic Algorithms, A.K Peters, Ltd., Natick, MA, 2002.

[21] J. Sklansky, “Conditional-sum addition logic”, IRE Transaction on Electronic Computers, vol.EC-9, pp. 226-231, 1960.

[22] O.J. Bedrij, “Carry-select adder”, IRE Transaction on Electronic Computers, June 1962.

[23] S. Knowles, “A Family of Adders”, Proc. 14th Symposium Computer Arithmetic, pp. 30-34, Apr. 1999.

[24] R.E. Ladner and M.J. Fisher, “Parallel Prefix Computation,” J. ACM, vol. 27, no. 4, pp. 831-838, Oct. 1980.

[25] P.M. Kogge and H.S. Stone, “A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations,” IEEE Trans. Computers, vol. 22, no. 8, pp. 786-792, Aug. 1973.

[26] R.P. Brent and H.T. Kung, “A Regular Layout for Parallel Adders,” IEEE Trans. Computers, vol. 31, no. 3, pp. 260-264, Mar. 1982.

[27] T. Han and D. Carlson, “Fast Area-Efficient VLSI Adders,” Proc. Symposium Computer Arithmetic, pp. 49-56, May 1987.

[28] Reto Zimmermann, “Non-Heuristic Optimization and Synthesis of Parallel Prefix Adders”, International Workshop on Logic and Architecture Synthesis (IWLAS’96), Grenoble, December 1996.

[29] J. Eyre and J. Bier, “The evolution of DSP processors”, IEEE Signal Processing Magazine, vol. 17, no. 2, pp. 43-51, Mar. 2000.

[30] M.R. Santoro and M.A. Horowitz, “SPIM: A pipelined 64x64-bit iterative multiplier," IEEE Journal of Solid-State Circuits, vol. 24, no. 2, pp. 487-493, Apr. 1989.

[31] S. Shah, A.J. Al-Khalili, and D. Al-Khalali, “Comparison of 32-bit multipliers for various performance measures, "The 12th International Conference on Microelectronics, pp. 75-80, Oct. 2000.

[32] C. S. Wallace. “A Suggestion for a Fast Multiplier”, IEEE Transactions on Electronic Computers, EC-13:14–17, February 1964.

[33] A. Weinberger, “4:2 Carry-Save Adder Module”, IBM Technical Disclosure Bulletin, vol.23:3811–3814, Jan.1981.

[34] V.G. Oklobdzija, D. Villeger, and S.S. Liu, “A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," IEEE Transactions on Computers, vol. 45, no. 3, pp. 294-305, Mar. 1966.

[35] A.D. Booth, “A signed binary multiplication technique," Quarterly Journal of Mechanics and Applied Mathematics, vol. 2, pp. 236-240, 1951.

[36] O.L. MacSorley, “High-speed arithmetic on binary computers," IRE Transaction on Electronic Computers, vol. 49, pp. 67-91, 1961.

[37] G. W. Bewick, Fast Multiplication: Algorithms and Implementation, Ph.D. thesis, Stanford University, Stanford, CA, Feb. 1994.

[38] P. Bonatto and V.G. Oklobdzija, “Evaluation of booth's algorithm for implementation inparallel multipliers," Signals, Systems and Computers, vol. 1, pp. 608-610, Oct. 1995.

Appendix A

Sign Extension in Booth Multipliers

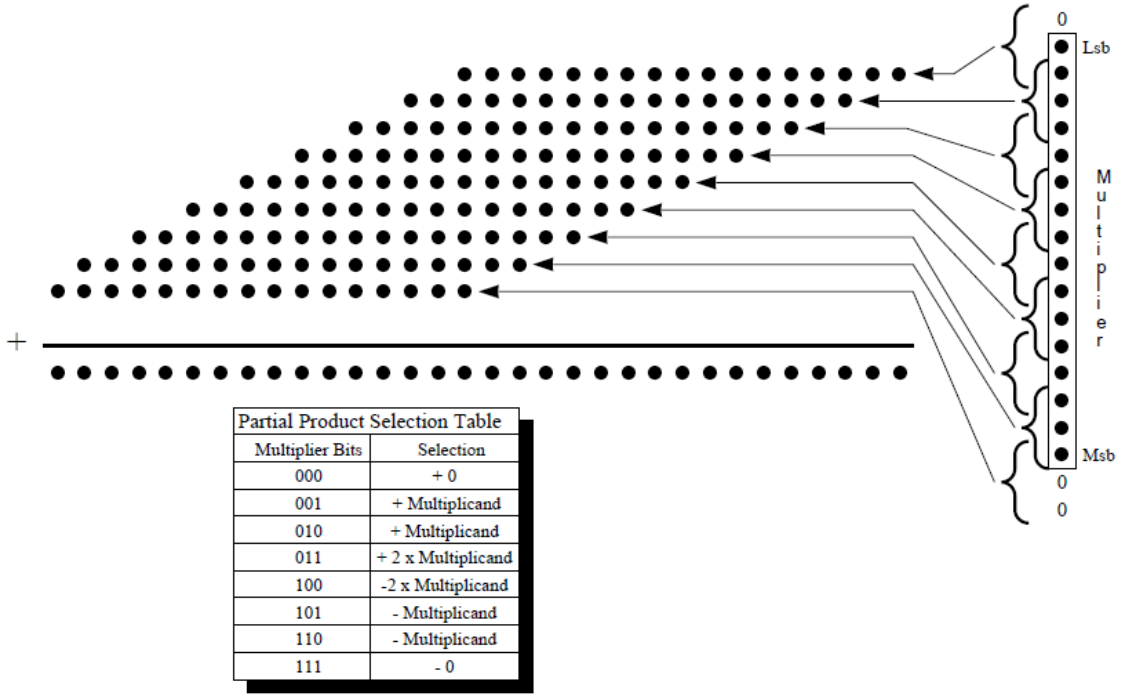
This appendix shows the sign extension constants that are needed when using Booth's multiplication algorithm are computed. The method will be illustrated for the 16x16 bit Booth 2 multiplication example given in Chapter 2. Once the basic technique is understood it is easily adapted to the higher Booth algorithms and also to the redundant Booth method of partial product generation. The example will be that of an unsigned multiplication, but the final section of this appendix will discuss the modifications that are required for signed arithmetic.

A.1 Sign Extension for Unsigned Multiplication

The partial products for the 16x16 multiply example, assuming that all partial products are positive, are shown in Figure A.1. Each partial product, except for the bottom one, is 17 bits long, since numbers as large as 2 times the multiplicand must be dealt with. The bottom partial product is 16 bits long, since the multiplier must be padded with 2 zeroes to guarantee a positive result. Figure A.2 shows the partial products if they all happen to be negative. Using 2's complement representation, every bit of the negated partial products is complemented, including any leading zeroes, and 1 is added at the least significant bit. The bottom partial product is never negated, because the 0 padding assures that it is always positive. The triangle of 1's on the left hand side can be summed to produce Figure A.3, which is exactly equivalent to the situation shown in Figure A.2. Now, suppose that a particular partial product turns out to not be negative. The leading string of ones in that particular partial product can be converted back to a leading of zeroes, by adding a single 1 at the least significant bit of the string. Referring back to the selection table shown in Figure A.1, a partial product is positive only if the most significant bit of the select bits for that partial product is 0. Additionally, a 1 is added into the least significant bit of a partial product only if it is negative. Figure A.4 illustrates this configuration. The S bits represent the 1's that are needed to clear the sign extension bits for positive partial products, and the S bits represent the 1's that are added at the least significant bit of each partial product if it is negative.

A.1.1 Reducing the Height

Finally, the height (maximum number of items to be added in any one column) of the dot diagram in Figure A.4 can be reduced by one by combining the S term of the top partial product with the two leading ones of the same top partial product, which gives the final result, shown in Figure A.5.



Partial Product Selection Table	
Multiplier Bits	Selection
000	+ 0
001	+ Multiplicand
010	+ Multiplicand
011	+ 2 x Multiplicand
100	-2 x Multiplicand
101	- Multiplicand
110	- Multiplicand
111	- 0

Figure A.1: 16 bit Booth 2 multiplication with positive partial products.

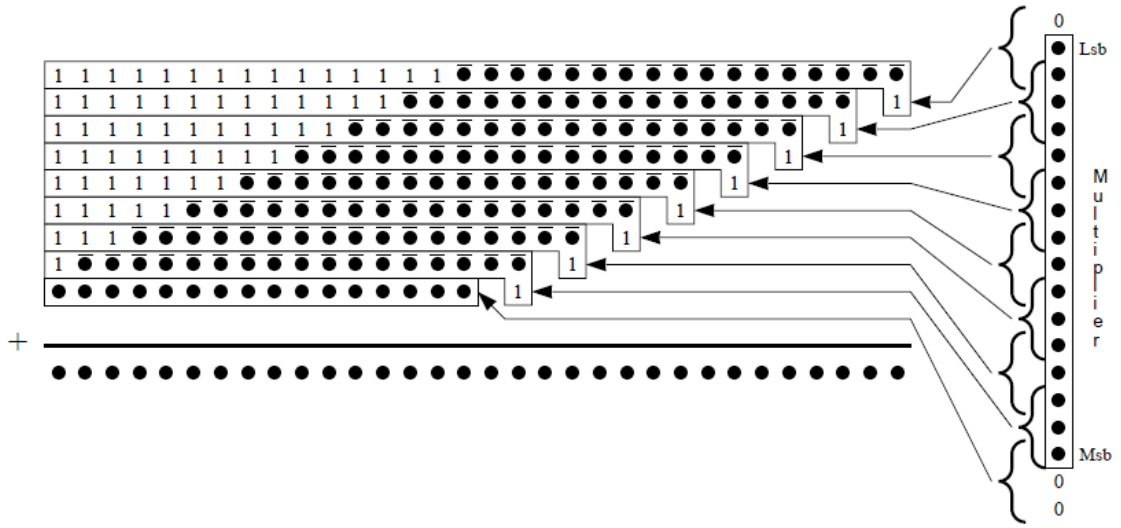


Figure A.2: 16 bit Booth 2 multiplication with negative partial products.

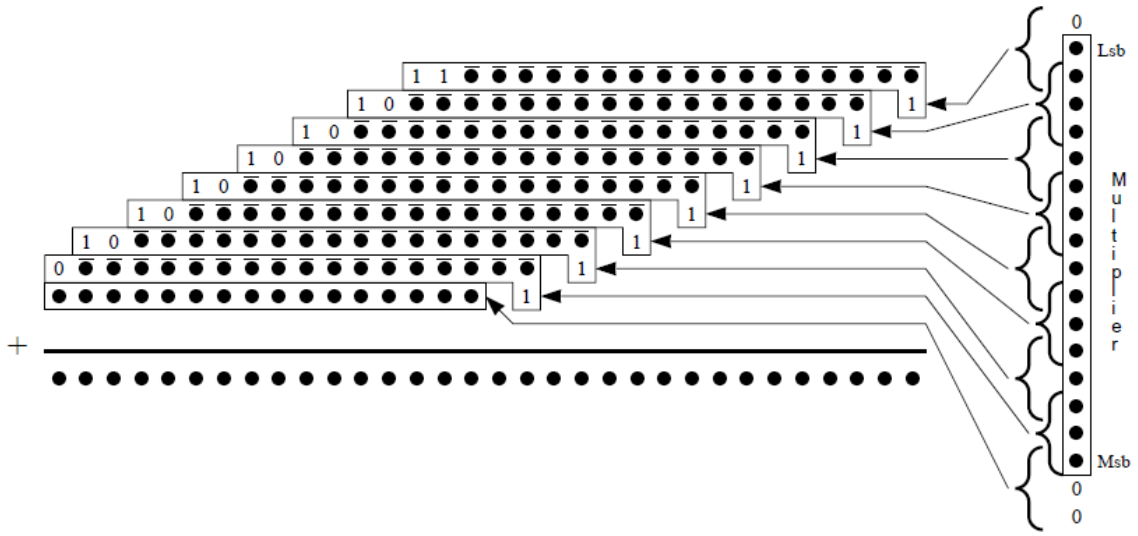


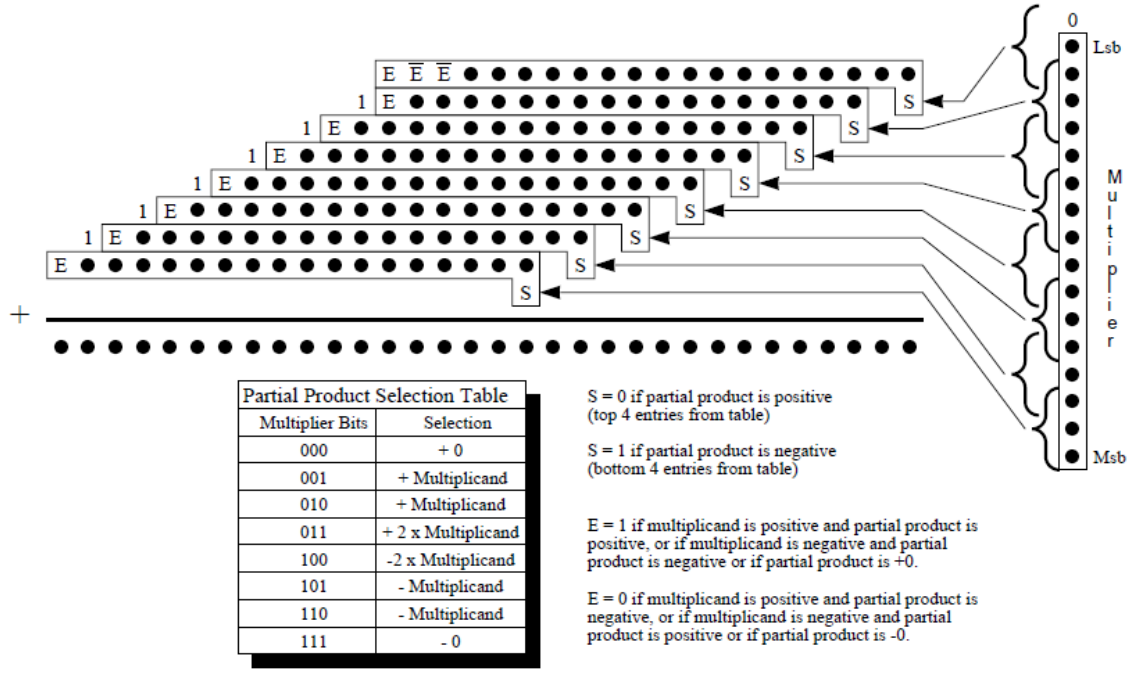
Figure A.3: Negative partial products with summed sign extension.

A.1 Signed Multiplication

The following modifications are necessary for 2's complement, signed multiplication:

1. The most significant partial product (shown at the bottom in all of the preceding figures), which is necessary to guarantee a positive result, is not needed for signed multiplication. All that is required is to sign extend the multiplier to fill out the bits used in selecting the most significant partial product. For the sample 16x16 multiplier, this means that one partial product can be eliminated.
2. When ±Multiplicand (from the partial product selection table) is selected, the 17 bit section of the effected partial product is filled with a sign extended copy of the multiplicand. This sign extension occurs before any complementing that is necessary to obtain −Multiplicand.
3. The leading 1's strings, created by assuming that all partial products were negative, are cleared in each partial product under a slightly different condition. The leading 1's for a particular partial product are cleared when that partial product is positive. For signed multiplication this occurs when the multiplicand is positive and the multiplier select bits chooses a positive multiple, and also when the multiplicand is negative and the multiplier select bits choose a negative multiple. A simple EXCLUSIVE-NOR between the sign bit of the multiplicand and the high order bit of the partial product selection bits in the multiplier generates the one to be added to clear the leading 1's correctly.

The complete 16x16 signed multiplier dot diagram is shown in Figure A.4



Partial Product Selection Table	
Multiplier Bits	Selection
000	+ 0
001	+ Multiplicand
010	+ Multiplicand
011	+ 2 x Multiplicand
100	-2 x Multiplicand
101	- Multiplicand
110	- Multiplicand
111	- 0

S = 0 if partial product is positive (top 4 entries from table)

S = 1 if partial product is negative (bottom 4 entries from table)

E = 1 if multiplicand is positive and partial product is positive, or if multiplicand is negative and partial product is negative or if partial product is +0.

E = 0 if multiplicand is positive and partial product is negative, or if multiplicand is negative and partial product is positive or if partial product is -0.

Figure A.4: Complete signed 16 bit Booth 2 multiplication.

Appendix B

B.1 Introduction

FPGA stands for field programmable gate arrays that can be configured by the customer or designer after manufacturing. Field programmable gate arrays are called this because rather than having a structure similar to a PAL or other programmable device, they are structured very much like gate array ASIC. An FPGA is a device that consists of thousands or even millions of transistors connected to perform logic functions. They perform functions from simple addition and subtraction to complex digital filtering and error detection and correction. Aircraft, automobiles, radar, missiles and computers are just some of the system that uses FPGA. FPGA are truly revolutionary devices that blend the benefits of both hardware and software. They can implement circuits just like hardware, providing huge power, area, and performance benefits over software, yet can be reprogrammed easily to implement wide range of tasks.

Xilinx, Altera and Quicklogic are just few companies that manufacture FPGAs. FPGA architecture consists of three basic capabilities: input/output (I/O) interfaces, basic building blocks, and interconnections. Figure 5.1 shows FPGA architecture. I/O interfaces are the mediums in which data are sent from internal logic to external sources and from which data are received from external sources. The interface signals can be unidirectional or bidirectional, single-ended or differential. Some I/O standards are:

1. GTL (gunning transceiver logic).
2. HSTL (high-speed transceiver logic).
3. LVCMOS (low-voltage CMOS).
4. LVTTL (low-voltage transistor-transistor logic).
5. LVDS (low-voltage differential signalling).

The basic logic building blocks are preconfigured logic or resources used to build design. Xilinx's basic building blocks are called configurable logic blocks (CLBs). Each CLB contains slices, and each slice has lookup tables (LUTs).

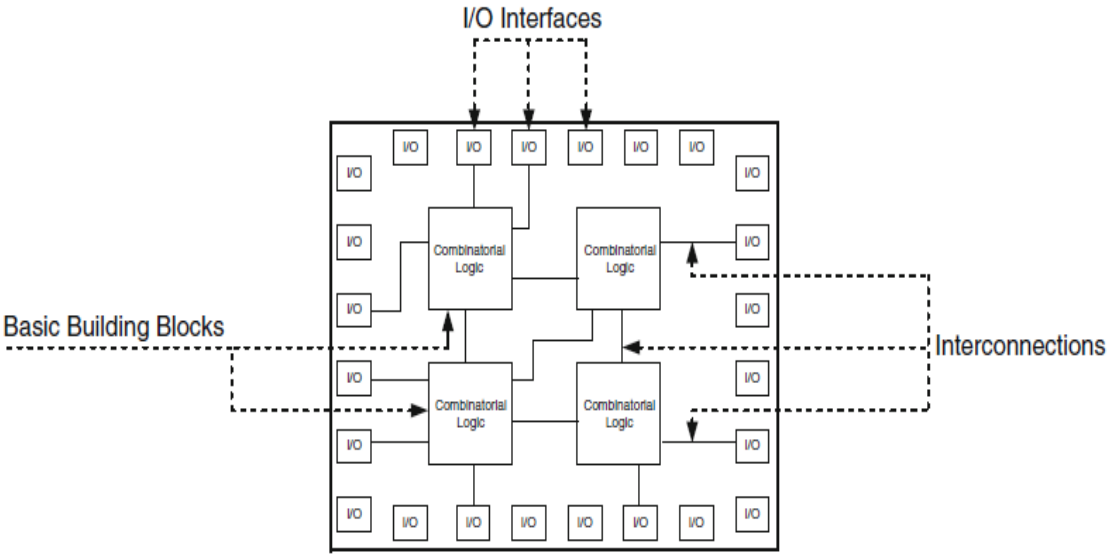


Figure B.1: FPGA Architecture

Interconnection involves connecting the basic building blocks to perform design-specific functions as well as connecting the internal logic to I/O interface.

B.2 FPGA Implementation

The FPGA that is used for the implementation of the circuit is the Xilinx Spartan 3E (Family), XC3S500 / XC3S1600 (Device), FG320 / FG484 (Package), -5 (Speed Grade). The working environment/tool for the design is Xilinx ISE 9.2i. Spartan 3E family member's data summary is shown in Table B.1.

Table B.1: Spartan-3E Data Summary

Device	System Gates	Total CLBs	Total Slices	Max User I/O	Max Diff I/O pairs
XC3S100E	100K	240	960	108	40
XC3S250E	250K	612	2448	172	68
XC3S500E	500K	1164	4656	232	92
XC3S1200E	1200K	2168	8672	304	124
XC3S1600E	1600K	3688	14,752	376	156

B.3 FPGA Design Flow

As the FPGA architecture evolves and its complexity increases, CAD software has become more mature as well. Today, most FPGA vendors provide a fairly complete set of design tools that allows automatic synthesis and compilation from design specifications in hardware specification languages, such as Verilog or VHDL, all the way down to a bit stream to program FPGA chips.

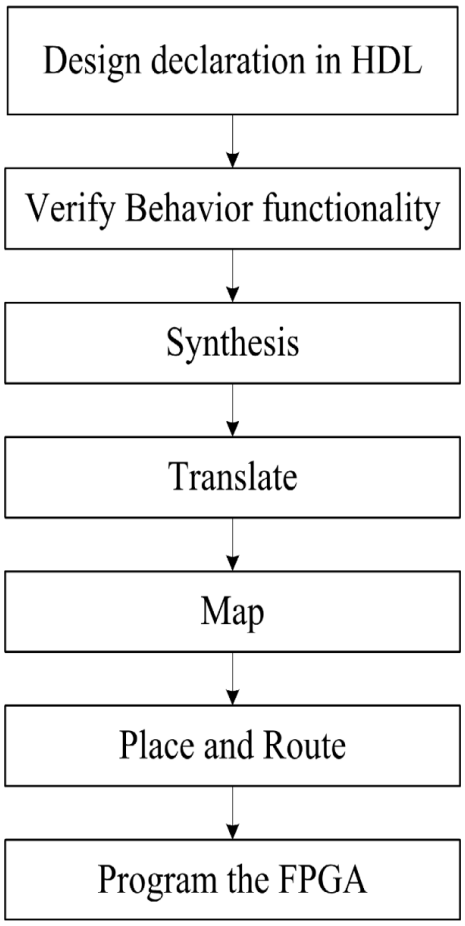


Figure B.2: FPGA Design Flow

A design flow includes the components shown in Figure B.2. Design constraints typically include the expected operating frequencies of different clocks, the delay bounds of the signal path delays from input pads to output pads (I/O delay), from the input pads to registers (setup time), and from registers to output pads (clock-to-output delay).

B.3.1 Design Implementation:

The Design implementation netlist consist of following sub-processes:

1. **Translation:** The Translate process merges all the input netlists and design constraints information's and outputs a Xilinx NGD (Native Generic Database) file.
2. **Mapping:** The Map process is run after the Translate process is complete. Mapping maps the logical design described in the NGD file to the components/primitives (slices/CLBs) present on the target device. The Map process creates an NCD file.

3. **Place and Route:** The place and route (PAR) process is run after the design has been mapped. PAR uses the NCD file created by the Map process to place and route the design on the target FPGA design.
4. **Bitstream Generation:** The collection of binary data used to program reconfigurable logic device is most commonly referred to as a "bit stream," although this is somewhat misleading because the data are no more bit oriented than that of an instruction set processor and there is generally no "streaming".
5. **Functional Simulation:** It is performed on the netlist or code generated by synthesis tool. It is performed before mapping of the design. This simulation process allows the user to verify that design has been synthesized correctly.
6. **Static timing analysis:** Three types of static timing analysis that can be performed are:
    a) Post-fit Static timing analysis: The Analyse Post-Fit Static timing process opens the timing Analyser window, which interactively select timing paths in design for tracing the timing results.
    b) Post-Map Static Timing Analysis: It analyse the timing results of the Map process. Post-Map timing reports can be very useful in evaluating timing performance(logic delay + route delay).
    c) Post Place and Route Static Timing Analysis: Post-PAR timing reports incorporate all delays to provide a comprehensive timing summary. If a placed and routed design has met all timing constraints, then proceed by creating configuration data and downloading a device.

B.4 FPGA Programming

A programming file is generated by running the Generate Programming File process. This process can be run after the FPGA design has been completely routed. The Generate Programming File process runs BitGen, the Xilinx bit stream generation program, to produce a bit stream (.BIT) or (.ISC) file for Xilinx device configuration. The FPGA device is then configured with the .bit file using the JTAG boundary scan method. After the Spartan device is configured for the intended design, then its working is verified by applying different inputs.

B.4.1 LCD Interfacing in FPGA

The Spartan-3E FPGA starter kit board features a 2-line by 16-character Liquid Crystal Display (LCD). The Figure B.3 represents the implementation of LCD Interfacing of 32×32 bit multiplier.

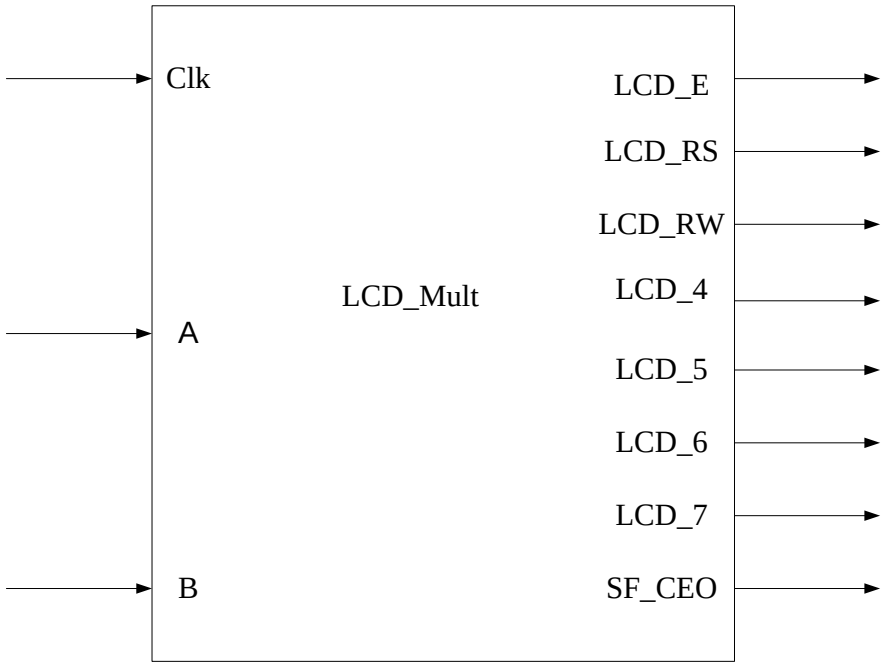


Figure B.3: LCD interfacing

1. 'clk' Input, 'A' Multiplicand, 'B' Multiplier
2. LCD_E: Read/Write Enable Pulse, "0" for disabled, "1" for Read/Write operation enabled. FPGA pin number is M18.
3. LCD_RS: Register Select, "0" for instruction register during write operations, busy flash during read operations, "1" for data for read or write operations. FPGA pin number is L18.
4. LCD_RW: Read/Write Control, "0" for write- LCD accepts data, "1" for read- LCD presents data. FPGA pin number is L17.
5. LCD_4: Data bit DB4, FPGA pin number is R15.
6. LCD_5: Data bit DB5, FPGA pin number is R16.
7. LCD_6: Data bit DB6, FPGA pin number is P17.
8. LCD_7: Data bit DB7, FPGA pin number is M15
9. SF_CE0: SF_CE0 is "1" then, FPGA application has full read/write access to the LCD.