

國立交通大學

電子工程學系 電子研究所
碩士論文

高效能且可組態之
子字組平行化乘加器設計



**High-Performance Reconfigurable
Sub-Word Parallel Multiplier-Accumulator Design**

研究生：林宏光

指導教授：黃俊達 博士

中華民國九十五年七月

高效能且可組態之
子字組平行化乘加器設計

**High-Performance Reconfigurable
Sub-Word Parallel Multiplier-Accumulator Design**

研究生：林宏光

Student: Hung-Kuang Lin

指導教授：黃俊達 博士

Advisor: Dr. Juinn-Dar Huang



A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics
College of Electrical & Computer Engineering
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in
Electronics Engineering & Institute of Electronics

July 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年七月

高效能且可組態之 子字組平行化乘加器設計

研究生：林宏光

指導教授：黃俊達 博士

國立交通大學

電子工程學系 電子研究所



本論文提出一個高效能乘加器的設計方法。此乘加器除支援子字組平行化功能之外，還能執行混模運算並具較有彈性的子字組設定。我們提出了一個新的子字平行部份乘積陣列及一個創新的子字平行部份乘積簡化樹以實現子字組平行化。為了利用原本的乘加器硬體，子字組平行化乘加器僅需增加微量的延遲及些許的面積。我們提出的乘加器可動態重組、可合成、可重覆使用且可驗證。我們實做並比較我們的設計及先前的設計。實驗數據顯示，無論在設計延遲、所佔面積、所耗功率，我們的方法在理論上及實務上都改善並且勝過舊方法。


High-Performance Reconfigurable Sub-Word Parallel Multiplier-Accumulator Design

Student: Hung-Kuang Lin

Advisor: Dr. Juinn-Dar Huang

Department of Electronics Engineering &
Institute of Electronics
National Chiao Tung University

ABSTRACT

The logo of National Chiao Tung University is a circular emblem with a gear-like border. Inside the circle, there is a stylized representation of a building or structure, with the letters 'ES' and 'A' visible. The year '1959' is also present at the bottom of the emblem.

This thesis presents the design methodology of a high-performance reconfigurable multiplier-accumulator (MAC) capable of supporting sub-word parallelism (SWP) and additional features such as mixed-mode operation and flexible sub-word combination and mode assignment scheme. In order to perform SWP on the proposed scalar MAC, a new SWP partial product array and a novel speed-optimized SWP partial product reduction tree are proposed. With slight delay and some area overhead, the SWP MAC utilizes essentially the same hardware as the proposed scalar MAC. The whole design is dynamically reconfigurable, fully-synthesizable, reusable, and verifiable. The proposed designs and previous relevant works are implemented and compared. Experimental results demonstrate that the proposed SWP MAC design theoretically and practically improves and outperforms previous works in terms of critical path delay, area cost, and power consumption.

ACKNOWLEDGMENT

誌 謝

能完成這份論文，首先我要感謝指導教授黃俊達老師。老師除了授業及解惑專業知識之外，也教導我許多做研究及做事的方法與態度，這些都對我的未來有裨益。另外感謝 ACAR 實驗室同學翊展、孝恩、維聖、士祐，你們在各方面給的建議與幫助讓我這兩年研究生涯過得充實而不孤單；碩一學弟哲霖、之暉、南興，碩一點五吉祥物詠翔學長，你們是阿達實驗室的快樂泉源，謝謝你們讓我愛上 LAB317B 這小空間；特別感謝碩零學弟建德在這個研究議題上給我的大量支援。謝謝交大電子 92 級的老同學們在工作上給的建言；謝謝建中 329 班及蘆中 326 的一千老朋友們，看到大家都很努力讓見賢思齊。也向其他默默支持我的朋友們說聲感謝。謝謝遠在美國的文玉長時間的擔待，妳的支持是我的最大動力。最後且最重要地，感謝家人的關懷。爺爺及奶媽是我的精神支柱；姊姊一直以來的支持及鼓勵，讓我一生受用；感謝父母親二十多年來的撫養及付出，沒有你們就不會有我與這份論文，我會努力讓你們開心。

我願將這份論文獻給支持我的大家。我愛你們！

CONTENTS

Abstract (Chinese)	I
Abstract (English)	II
Acknowledgment	III
Contents	IV
List of Tables	VII
List of Figures	VIII
Chapter 1 Introduction	1
Chapter 2 Previous Works	4
2.0 Overview	4
2.1 Prerequisites	4
2.1.1 Simple Multiplication & Booth's Algorithm	4
2.1.2 Acceleration of Multiplication Flow	6
2.1.3 Modified Booth's Algorithm (MBA)	7
2.2 Related Works	9
2.2.1 Partial Product Generation (PPG)	9
2.2.2 Three-Dimensional-Method (TDM) PPRT	14
2.2.3 High-Speed Adders	16
2.2.4 Sub-Word Parallelism (SWP)	20
2.3 Summaries of Previous Works	26

Chapter 3 Proposed MAC Designs 27

3.0 Overview 27

3.1 Scalar MAC (SMAC) Design 27

 3.1.0 Specification 27

 3.1.1 Scalar Partial Product Generation (SPPG) 28

 3.1.2 Scalar Partial Product Reduction Tree (SPPRT) 31

 3.1.3 Scalar Carry-Propagate Adder (SCPA) 33

 3.1.4 Summaries of the Proposed Scalar MAC Design 33

3.2 Sub-Word Parallel MAC (SWP MAC) Design..... 34

 3.2.0 Specification 34

 3.2.1 Sub-Word Parallel MAC Execution Flow 35

 3.2.2 Sub-Word Parallel PPG (SWPPG) 36

 3.2.3 Sub-Word Parallel PPRT (SWPPRT) 43

 3.2.4 Sub-Word Parallel CPA (SWCPA) 46

 3.2.5 Summaries of the Proposed SWP MAC Design 49

Chapter 4 Experimental Results 50

4.0 Overview 50

4.1 Implementation 50

4.2 Discussion of Experimental Results 51

 4.2.0 Overview 51

 4.2.1 Delay Comparison 52

 4.2.2 Area Comparison 55

 4.2.3 Power Comparison 58

Chapter 5	Application Notes	60
5.0	Overview	60
5.1	Functionality Enhancement	60
5.1.1	Multiply-Accumulate (MAC) Operation	60
5.1.2	Multiply-Negate (MAN) Operation	62
5.1.3	Unsigned Operation	65
5.1.4	Mixed-Mode Operation	67
5.2	Overflow/Underflow Check for FXP Numbers	69
5.2.1	Fixed-Point (FXP) Representation	69
5.2.2	Maintaining Precision & Accuracy	70
5.2.3	Saturation & Overflow/Underflow for Integers	71
5.2.4	Rounding of Fractions	77
5.3	Reconfigurable Parameters Setup	78
Chapter 6	Conclusions	82
	Future Works	83
	Bibliography	84



LIST OF TABLES

Table 2.1. Selection table of modified Booth's algorithm	8
Table 2.2. Truth table of standard encoding	10
Table 2.3. Truth table of compact encoding	10
Table 2.4. Truth table of race-free encoding	11
Table 2.5. Truth table of <i>LSB_new</i> and <i>hot2</i>	14
Table 3.1. Specification of the proposed SMAC design	27
Table 3.2. Specification of the proposed SWP MAC design	35
Table 3.3. Possible sub-word combinations of the proposed SWP MAC design	35
Table 3.4. Truth table of sign encoding bits and sign bits of PPs	41
Table 4.1. Environment setup for experiments	51
Table 4.2. Critical path delay comparison	52
Table 4.3. Delay overhead on performing SWP	54
Table 4.4. Area cost comparison	55
Table 4.5. Area overhead on performing SWP	57
Table 4.6. Power consumption comparison	58
Table 4.7. Power-delay characteristic comparison	59
Table 5.1. Pseudo MAC instruction types and notations	72
Table 5.2. Pseudo MAC instruction examples	72
Table 5.3. Some available modes for pseudo MAC instructions	73
Table 5.4. Possible saturation conditions using the exempling architecture	76
Table 5.5. Interface of the proposed design	79
Table 5.6. Possible sub-word combinations of the proposed SWP MAC design	80
Table 5.7. Configuration example of KILL signal	80
Table 5.8. Configuration example of MODE signal	81

LIST OF FIGURES

Fig. 2.1. Simple multiplication flow	5
Fig. 2.2. Multiplication flow in three steps	7
Fig. 2.3. Execution flow of MBE multiplication	9
Fig. 2.4. The MBE encoder and decoder in [9]	11
Fig. 2.5. Sign encoding and hot-one modification	13
Fig. 2.6. The concept of TDM	15
Fig. 2.7. An 8-bit carry-select adder example	17
Fig. 2.8. Architecture of a 32-bit hybrid parallel-prefix/carry-select Ling adder	18
Fig. 2.9. Architecture of a 32-bit scalar Fong adder	19
Fig. 2.10. Logic operators used in Fong adder	20
Fig. 2.11. A simplified PPA for 32×32 multiplication in different modes	22
Fig. 2.12. <i>Shared Segmentation</i> PPA for 32×32 multiplication in different modes ...	24
Fig. 3.1. Execution flow of the proposed Scalar MAC design	28
Fig. 3.2. Decoding mcand 1000 in different modes when MBE selects $-2x$	30
Fig. 3.3. FA cell used in the proposed SPPRT	32
Fig. 3.4. The proposed scalar architecture	34
Fig. 3.5. Execution flow of the 32-bit proposed SWP MAC design	36
Fig. 3.6. A 32-bit example of masking and multiplexing on the multiplier	37
Fig. 3.7. Detailed view of the 32-bit proposed SWPPA with a selection example ...	39
Fig. 3.8. SW combinations of the 32-bit proposed SWP MAC design	42
Fig. 3.9. Breaking the FA carry-chain for SWP in SWPPRT	44
Fig. 3.10. FA with carry-in masking used in [10]	45
Fig. 3.11. FA with carry-out masking used in the proposed design	45
Fig. 3.12. A simple 64-bit SWP adder	47

Fig. 3.13. Architecture of a 32-bit Fong adder with reconfigurability	48
Fig. 5.1. Execution flow of two approaches to completing MAC operation	61
Fig. 5.2. MAN flow of method A	62
Fig. 5.3. MAN flow of method B	63
Fig. 5.4. MAN flow of method C	64
Fig. 5.5. An exempling PPA for MAN/MAS operations using method C	65
Fig. 5.6. Adding a PP to perform unsigned operation	66
Fig. 5.7. A representation problem on negation of unsigned numbers	67
Fig. 5.8. Dynamic range comparison among signed, unsigned, and mixed-mode ...	68
Fig. 5.9. A PPA supporting MAN/MAS, unsigned/mixed-mode operation	69
Fig. 5.10. Effect with or without saturation when overflow/underflow occurs	74
Fig. 5.11. Three different rounding schemes	78



CHAPTER 1

INTRODUCTION

Multiply-accumulate (MAC) computation is one of the most frequent operations in DSP applications. A multiplier followed by an accumulator to integrate into a multiplier-accumulator (MAC) unit characterizes a DSP processor. A series of MAC operations has an arithmetic form like coefficient-data, inner product, or matrix computation, and which serves as the core operation in many DSP algorithms such as convolution, finite impulse response (FIR), fast Fourier transform (FFT), discrete cosine transform (DCT), and so many other DSP algorithms also demand extensive MAC operations. Multiplication (MUL), a basic essential arithmetic operation, is regarded as a special case of MAC operation processed in the same MAC unit [1], [2], [3]. Improvements in MAC design therefore significantly benefit the performance of the whole DSP processor according to Amdahl's law. A high performance DSP processor desires a high speed MAC unit with reduced area, low power, and high computational throughput, decided by the specification. To facilitate a high speed MAC design, an architecture using radix-4 modified Booth encoding (MBE) [4] and Wallace partial product reduction tree (PPRT) [5] associated with a high speed carry-propagate adder (CPA) is prevalent. To increase the computational throughput, sub-word parallelism (SWP), a form of single-instruction-multiple-data (SIMD), helps by processing all sub-words (SWs) in parallel and hence providing a performance boost especially for multimedia applications that often require lower-precision operands [6].

Considering the short time to market of a product required in the very era of system-on-a-chip (SoC), a synthesizable, reusable, and verifiable silicon intellectual

property (SIP) with flexible user reconfigurability is popular and utilizes the design reuse concept to help accelerate system integration [7]. Some MAC designs improve the delay of CPA by prudently calculating the signal arrival time of each operand bit, and use the delay profile to configure a faster adder scheme [8], [9]. This indicates the adder scheme highly depends on the chosen cell library and thus usually not suitable for reusable designs.

The previous SWP MAC designs are not speed optimized: the architecture in [6] does not use MBE, resulting more partial products to be accumulated and considerably increasing the latency. A modified Booth-encoded (MBE) MAC architecture in [10], [11] completes SWP using a technique called “shared segmentation” to arrange the partial product array (PPA); however, it forces a regular connection scheme for full-adders (FAs) in the Wallace PPRT, producing lower performance. In addition, the previous designs have a limited functionality either in data format or in SW flexibility.

This thesis presents a synthesizable, reusable, and verifiable high-performance reconfigurable MAC design. The proposed SWP MAC design is obtained, with slight effort and small area overhead, by performing SWP on the proposed scalar design which comprises a high performance MBE, a speed optimized PPRT, and a high speed CPA. The proposed scalar design supports not only the signed operation but also the unsigned and a special mixed-mode operation which forces the multiplicand to be signed and multiplier to be unsigned. Mixed-mode operation provides a larger dynamic range for DSP applications. The proposed scalar design also has better performance in most cases compared with previous scalar MAC designs. As for SWP, the proposed SWP MAC utilizes a novel SWP PPA to advance the performance of SWP PPRT, and takes advantage of a new concept of carry-out masking to facilitate a speed optimized SWP PPRT. Concerning the CPA, a high-performance Fong adder

with SWP capability is integrated into the proposed scalar and SWP designs. The proposed scalar design is superior to related works in most cases while the proposed SWP MAC design not only outperforms previous works in terms of delay, area, and power consumption but also features a more flexible SW combination and mode assignment scheme.

The remainder of this thesis is organized as follows: Chapter 2 briefly describes the previous works that are most relevant to the proposed designs. Chapter 3 details the design methodology of the proposed MAC designs and theoretically compares with previous works. Chapter 4 demonstrates and discusses the experimental results. Chapter 5 explains some important application notes concerning the utilization of the proposed designs. Chapter 6 concludes this thesis. Future works and bibliography are also provided afterward.



CHAPTER 2

PREVIOUS WORKS

2.0 Overview

In this chapter, we review some important previous work relevant to the proposed MAC architecture. Section 2.1 recalls fundamentals and algorithms of multiplication; Section 2.2 concisely describes some related works, theorems, and techniques; Section 2.3 summarizes the previous works and highlights the differences to be described in the next chapter.

2.1 Prerequisites

2.1.1 Simple Multiplication & Booth's Algorithm

Traditional binary multiplication flow is essentially the same as done in decimal multiplication: Logic AND operation is performed on a single bit of the multiplier with each bit of multiplicands; the temporal result, a partial product (PP), always equals the multiplicand itself or zero; the least-significant-bit (LSB) of the PP is aligned to the multiplier bit used. Consequently, if an m -bit by n -bit multiplication is executed, there will be n PPs each with m significant bits. After zero-extending or sign-extending each PP to the most and the least significant ends, an m -bit by n -bit rectangular partial product array (PPA) is formed. Accumulating all PPs produces the final multiplication result. Fig. 2.1 shows the simple multiplication flow of an 8-bit by 8-bit multiplication. Roughly speaking, the number of the significant bits used (x -bit in PPA of Fig. 2.1) is proportional to the amount of hardware required [12].

The flow above is somewhat redundant when a series of zeros shows in the multiplier; it can be further improved. In 1951, Booth introduces a binary multiplication algorithm on the grounds of the add-and-shift concept [12]: the consecutive bits in multiplier affect the generation of partial products. This algorithm is based on two's complement system and thus performs signed multiplication. The fact that shifting alone is faster than addition followed by shifting makes Booth's multiplication faster than traditional ones. Although Booth's algorithm, also referred as radix-2 Booth's algorithm, is not directly applied to modern arithmetic circuits, it serves as a basis in understanding the radix-4 version of this algorithm – modified Booth's algorithm (MBA) [3].

Note that both simple multiplication and Booth's algorithm produce a number of n PPs where n is the bit width of the multiplier, as the eight PPs shown in Fig. 2.1.

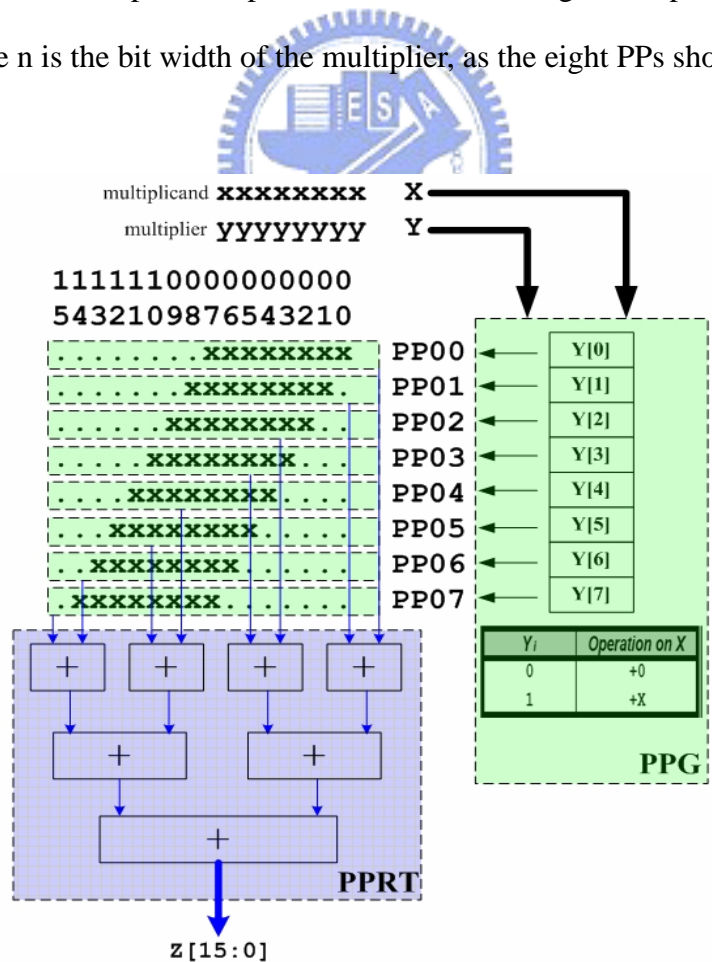


Fig. 2.1. Simple multiplication flow.

2.1.2 Acceleration of Multiplication Flow

The completion of multiplication involves two basic operations – partial product generation (PPG) and their accumulation. Consequently, reducing the number of PPs or accelerating their accumulation helps speed up multiplication [1]. The MBA for reducing the number of PPs will be detailed in the next section.

Accumulation of all PPs implies a series of addition. In theory, we can use a series of carry-propagate adders (CPAs) to accumulate all PPs; the number of addition required is in proportion to the number of PPs. This naïve method is impractical because the delay of a CPA is considerable, let alone the number of PPs grows with the bit width of the multiplier. A better architecture for connecting CPAs exploits some parallelism; this is how we demonstrate in Fig. 2.1. However, the number of the CPA levels still relates to the number of the PPs, incurring longer delay.

As a result, the partial product reduction tree (PPRT) is often utilized. There are plenty of algorithms dedicating to construct a PPRT [5], [14], [15]. One of the most popular constructions of PPRT is the Wallace Tree [5]: use full-adders (FAs), or say (3:2) counters [3], as the building blocks to perform carry-save addition (CSA). It does not work out the addition result at the middle levels of the tree; instead, it just saves each level's carry-out and sum information of CSAs, avoiding the carry propagation which takes a long time. A PPRT, Wallace Tree included, often reduces many rows of PPs until only two rows remain; after summing these two final PPs using a CPA (carry-out demands 1-bit left shift), the product is obtained.

A PPRT speeds up multiplication; multiplication flow is hence frequently sliced into three phases – partial product generation, partial product reduction tree, and carry-propagate adder. Fig. 2.2 exhibits the flow.

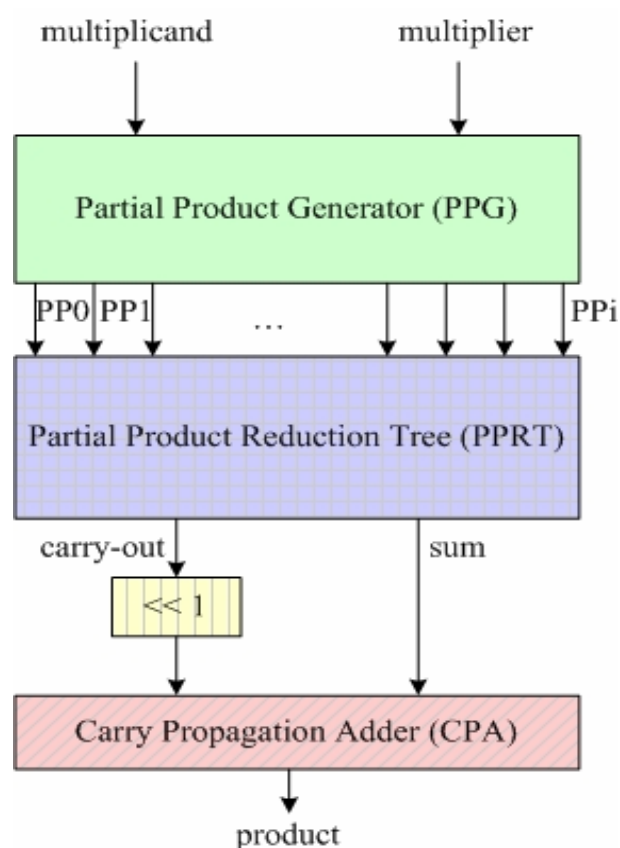


Fig. 2.2. Multiplication flow in three steps.



2.1.3 Modified Booth's Algorithm (MBA)

As mentioned previously, the PPG is dependent with the pattern of multiplier, and the number of PPs is in proportion to the bit width of the multiplier. A PPG that creates a fewer number of PPs will allow the partial product summation to be faster and use less hardware. Given an n -bit multiplier, simple multiplication or Booth's algorithm encodes and ignores/eliminates one multiplier bit for n times, and hence obtains n PPs.

In 1961, MacSorley presents a radix-4 Booth's algorithm based on the concept of original Booth's algorithm and is referred to as modified Booth algorithm (MBA) [4]. Due to the property of radix-4 system, two bits of multiplier are ignored after each encoding, and hence the number of PPs is reduced. Thanks to the property,

modified Booth's encoding (MBE) generates fewer PPs, and is especially useful if groups of consecutive zeros and ones shown in the multiplier. Table 2.1 lists the corresponding behavior for all possible conditions of an encoding triplet [1]. MBA decreases the latency of multiplication through reduction of the number of the PPs and thus the reduction of the levels in the PPRT.

A modified Booth encoded/recoded (we use "encode" in the remaining content) multiplier also consists of three parts: a modified Booth encoder (MBE) associated with the arrangement/alignment of partial product array (PPA) to do PPG, a lower PPRT to accumulate PPs to two, and a fast CPA to sum for the product. Fig. 2.3 displays the execution flow of modified Booth encoded multiplication.

The proposed architecture is theoretically based on MBA. Some most MBA-relevant works will be discussed in the following sections.

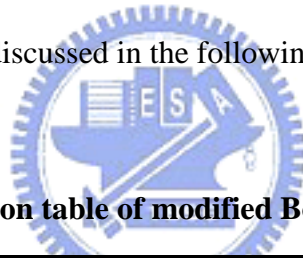


Table 2.1. Selection table of modified Booth's algorithm.

Y_{2i+1}	Y_{2i}	Y_{2i-1}	Operation	Explanation
0	0	0	+0	<i>string of 0's</i>
0	0	1	+X	<i>end of 1's</i>
0	1	0	+X	<i>a single 1</i>
0	1	1	+2X	<i>end of 1's</i>
1	0	0	-2X	<i>beginning of 1's</i>
1	0	1	-X	<i>a single 0</i>
1	1	0	-X	<i>beginning of 1's</i>
1	1	1	-0	<i>string of 1's</i>

Note: $2i$ indicates the even bit positions; when $i = 0$, $Y[-1]$ is assume to be zero

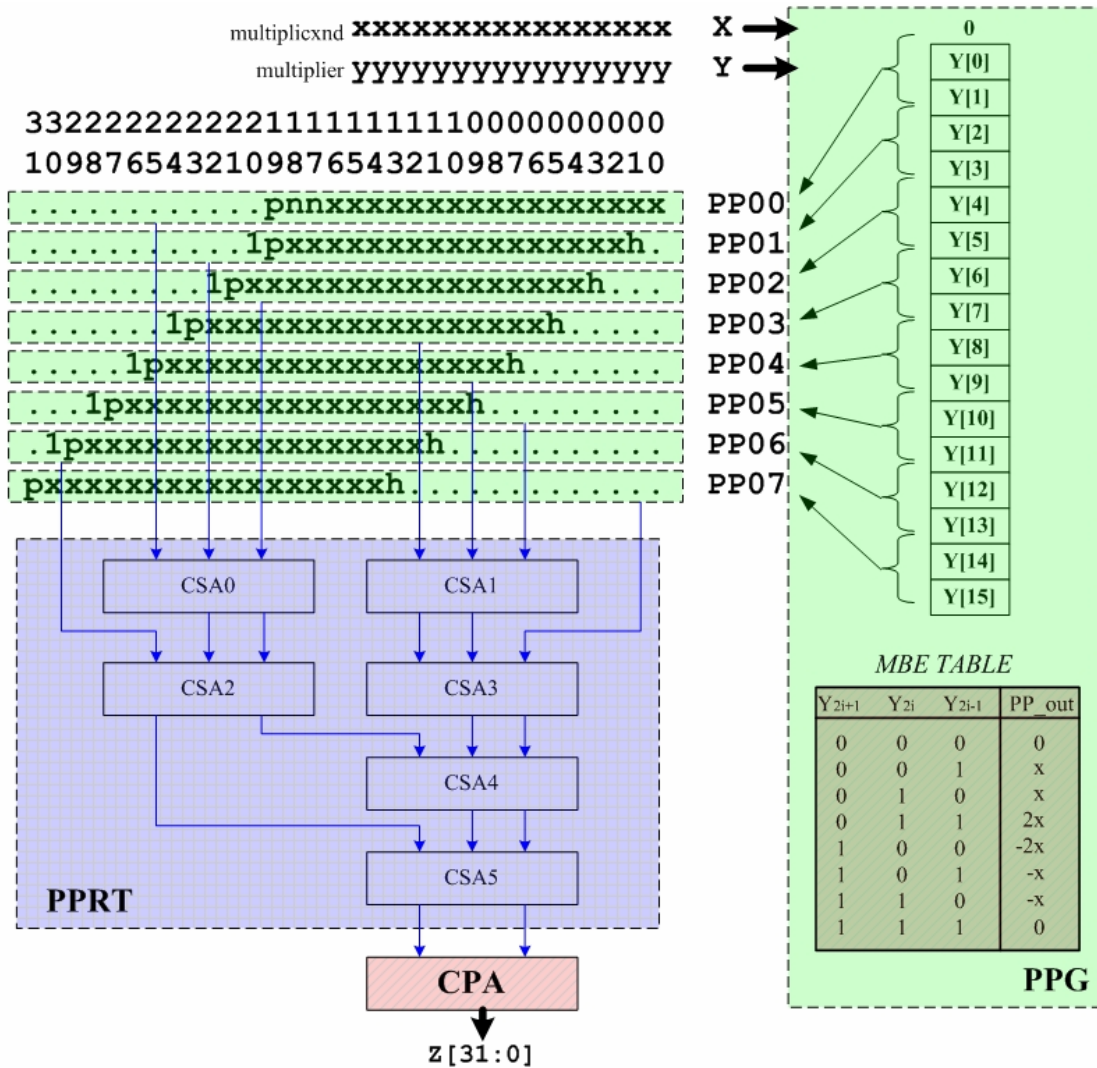


Fig. 2.3. Execution flow of MBE multiplication.

2.2 Related Works

2.2.1 Partial Product Generation (PPG)

Partial product generation is divided into two parts - decoding the multiplicand in correspondence with the encoding of multiplier done by an MBE, and the arrangement and alignment on the MBE outputs to form the PPA.

Concerning the MBE, [16] presents a comparison of energy dissipation among standard, compact, and race-free encoding schemes of an MBE. The race-free

scheme encoded MBE consumes least power because it balances the delay of internal signals and thus avoids glitches/sparks in the circuits. In [9] the race-free MBE is further optimized in terms of timing and area. The spirit of this implementation is to intentionally use “wrong” encoding signals at middle gate levels and corrects the error at final level. The temporal “wrong” logic enables more logic optimization compared to other encoding schemes, leading to a decrease in delay, reduction of area, and less consumption of power. Table 2.2, 2.3, and 2.4 list the truth table of standard, compact, and race-free MBE schemes, respectively. Fig. 2.4 shows the improved encoder and decoder of the MBE in [9].

Table 2.2. Truth table of standard encoding.

Y_{2i+1}	Y_{2i}	Y_{2i-1}	$P1$	$P2$	Z	$M1$	$M2$
0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	0	1	0	0	0
1	0	0	0	0	0	0	1
1	0	1	0	0	0	1	0
1	1	0	0	0	0	1	0
1	1	1	0	0	1	0	0

Table 2.3. Truth table of compact encoding.

Y_{2i+1}	Y_{2i}	Y_{2i-1}	$P1$	$P2$	Neg
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	0	1	1
1	0	1	1	0	1
1	1	0	1	0	1
1	1	1	0	0	1

Table 2.4. Truth table of race-free encoding.

Y_{2i+1}	Y_{2i}	Y_{2i-1}	$P1$	$P2$	Neg	Z
0	0	0	0	1	0	1
0	0	1	1	0	0	1
0	1	0	1	0	0	0
0	1	1	0	1	0	0
1	0	0	0	1	1	0
1	0	1	1	0	1	0
1	1	0	1	0	1	1
1	1	1	0	1	1	1

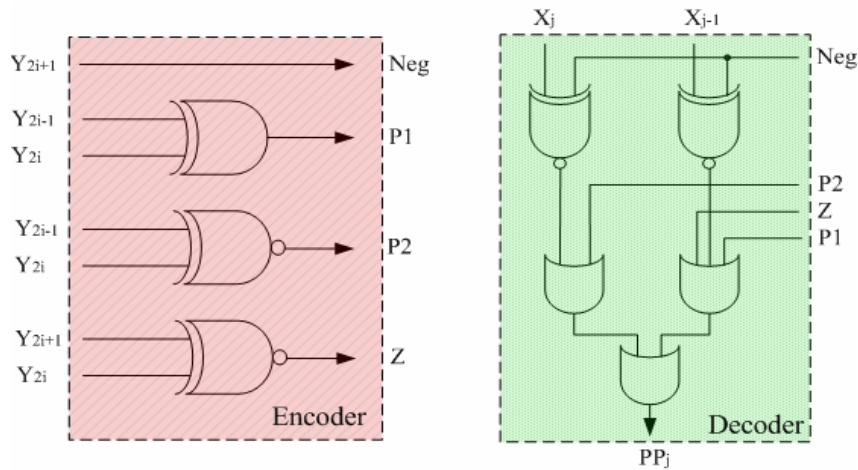


Fig. 2.4. The MBE encoder and decoder in [9].

When MBA is used, the PPs are treated as signed numbers since three negative MBE outputs may be selected as listed in Table 2.1. This suggests sign extension be applied to every PP to ensure a correct result; however, sign extension needs to take considerable extra logic. To deal with, in [17], [18], [19], a technique called sign-encoding (SE) or sign-generation is provided and [12] gives this technique a general description. The concept of SE is depicted in Fig. 2.5 at the MSB end: It begins to presume all PPs are negative and hence one-extension is applied as shown in Fig. 2.5a. Since the extended ones are fixed in position, accumulating all extended

ones in advance produces $\{1,1\}$ in front of the first PP and $\{1,0\}$ for others, as shown in Fig. 2.5b. To correct the presumption, add one to the LSB of each sign-extension string, resulting in the logic in Fig. 2.5c. As a whole, SE exploits the predictability of sign-extension, and cleverly protects from the redundant extension bits simply for correctly representing a sign number. It takes only two or three SE bits, $\{p,n,n\}$ for the first PP and $\{1,p\}$ for others, in front of the original MSB of each PP where n stands for the original sign of each PP; p , the negation of n .

We simulate a multiplier with or without using SE. While SE is used, the power consumption of the PPG and PPRT is saved up to one-third of that without using SE; the improvement rate grows as the bit width increases.

Another problem arises when MBE selects a negative output. Since MBA treats the operands as signed numbers in two's complement (TC) format, if a negative output is selected, we have to negate/two's-complement the bit stream, implying a two's complementer for negation is required. To complete the operation, the ones, also called hot-ones [19], are needed to be added after inverting (one's complementing) the bit stream. It's a waste to let these ones solely for TC be one of the PPRT inputs. Fortunately, due to MBA this can be prevented since the least significant bit (LSB) position of the present PP should align two bits far from the LSB of the preceding PP; two bits space $\{h,h\}$ is saved and can be utilized to locate the hot-one from the preceding PP as shown in Fig. 2.5a at the LSB end. The hot-one may also left shift one bit if MBE selects $2x$ or $-2x$ from the encoding table, but this takes no effort since two bits space are reserved.

In case a random-valued multiplier is being encoded, the hot-one bit may show up in either left or right h position. This irregularity will increase the PPRT latency [9]. In [9], the authors also propose a skill – we refer it to “hot-one modification” in this paper– to regulate the LSB end of all PPs. Observing the fact that the hot-one

logic relates the LSB logic of the present PP as shown in Fig. 2.5b, a truth table as listed in Table 2.5 can be built; the new logic equation of two signals LSB_new and $hot2$ can be expressed as:

$$\begin{aligned}
 LSB_new_i &= x_{LSB} \cdot (y_{2i-1} \oplus y_{2i}) \\
 hot2 &= y_{2i+1} \cdot y_{2i-1} + y_{2i} \cdot y_{2i-1} + x_{LSB} \cdot y_{2i} + x_{LSB}
 \end{aligned}
 \tag{2.1}$$

It arranges all hot-one bits to the left h positions ($hot2$) accompanied with the probable modification on the preceding LSB (LSB_new). As a result, Fig. 2.5c exhibits the arranged, shorter, parallelogram-shaped, more regular PPA to be accumulated in the PPRT.

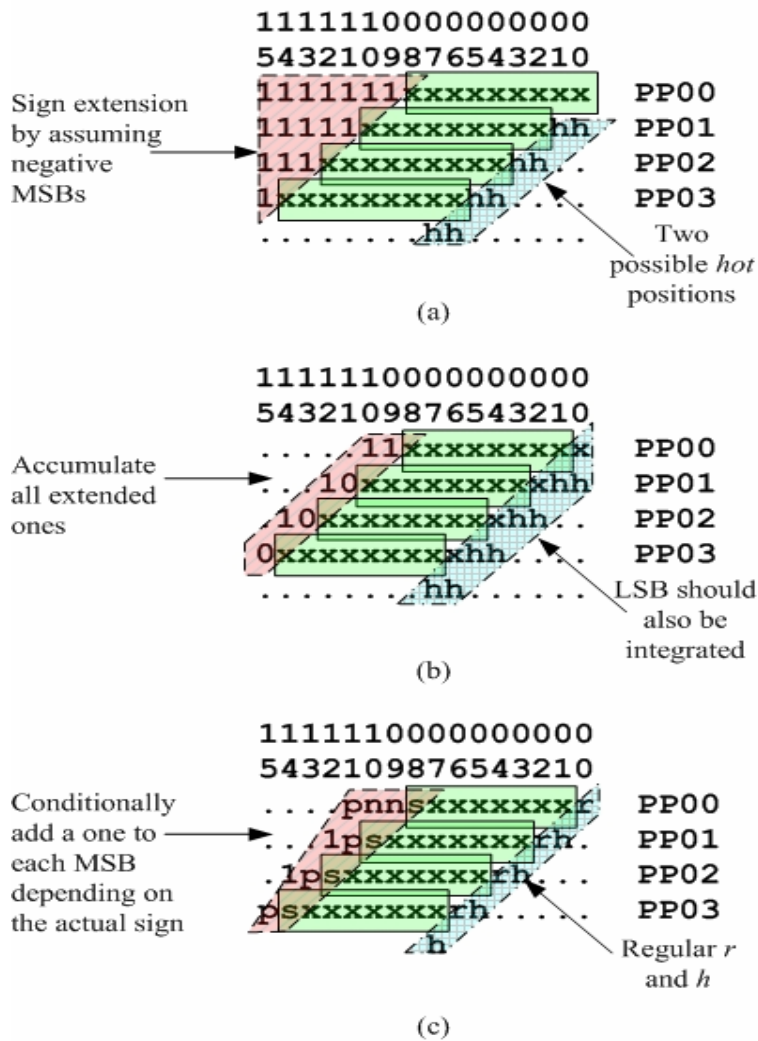


Fig. 2.5. Sign encoding and hot-one modification.

Table 2.5. Truth table of LSB_new and $hot2$.

LSB_old	Y_{2i+1}	Y_{2i}	Y_{2i-1}	$hot2$	LSB_new
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	0	1
1	0	1	1	0	0
1	1	0	0	1	0
1	1	0	1	0	1
1	1	1	0	0	1
1	1	1	1	0	0

2.2.2 Three-Dimensional-Method (TDM) PPRT

In [8], Oklobdzija *et al* present a three-dimensional method (TDM) to build a speed optimized Wallace PPRT. The main idea of this speed optimization can be briefly depicted as Fig. 2.6: In Fig. 2.6a, a common logic implementation of an FA is shown. Without loss of generality, assuming a NAND gate delay to be 1 and an XOR gate delay to be 2, the delay of each input-to-output path can be calculated as shown in Fig. 2.6b. The longest path is from input a or input b to output sum ; sum , therefore, is referred to as the “slow output” in contrast with the “fast output”, $cout$. cin is the “slow input” since it can wait for a slow output. Connecting a “slow output” to a signal requiring a “fast input” (e.g., a) produces the critical path! Take a two-level PPRT for example, the latency of the left configuration in Fig. 2.6c is

more balanced since it connects the “fast output” *cout* to a “fast input” *b*, while the regular configuration on the right side always connects *sum* to *b*, creating a critical path. Exploiting the concept to balance the uneven delay of all paths is the spirit of TDM.

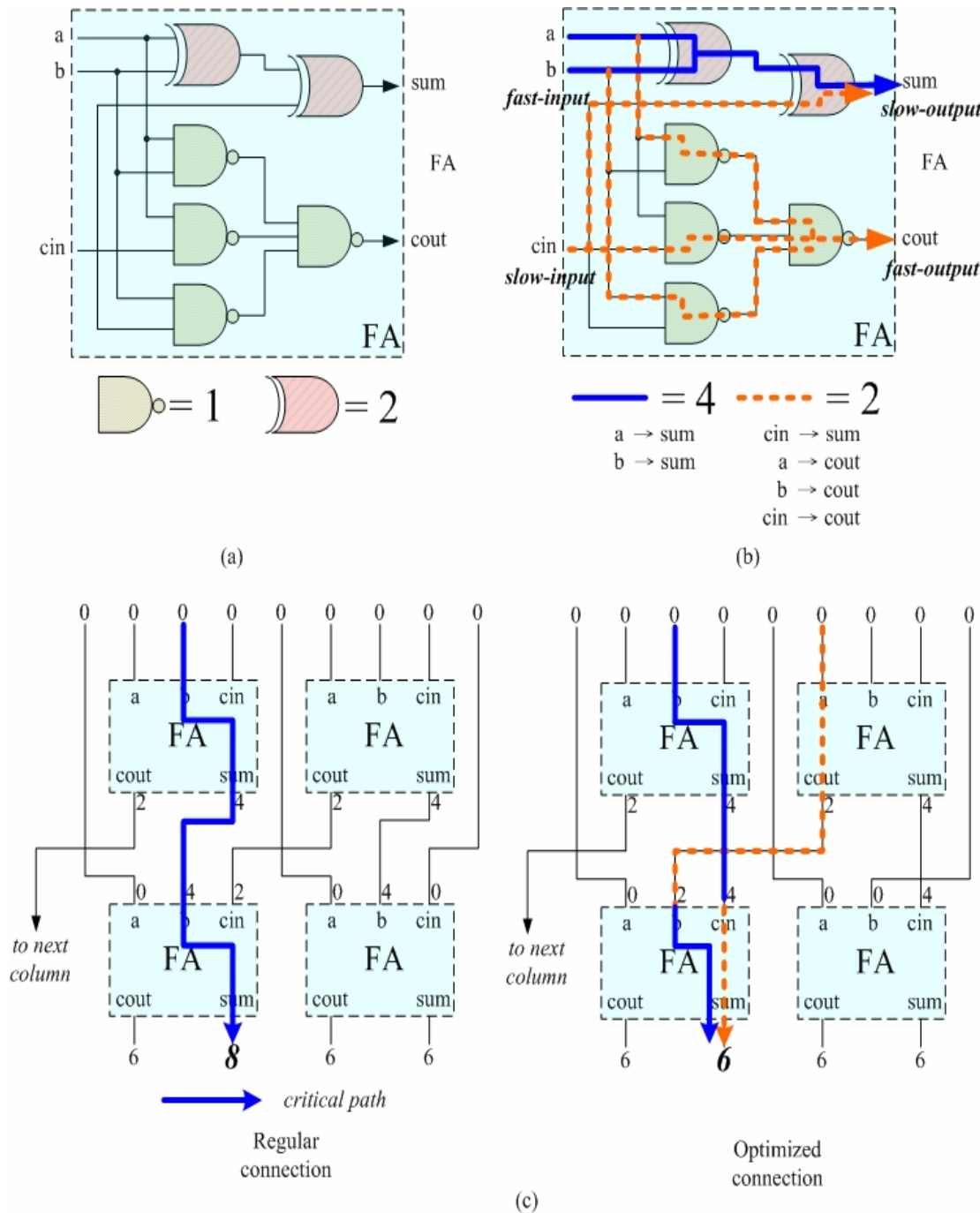


Fig. 2.6. The concept of TDM.

TDM requires the delay information of each cell used in the PPRT and then three-dimensionally constructs a speed optimized PPRT using the cells available. The thought of regulating the PPA in [9] stems from the fact that TDM is also implemented in their work. Irregularity of PPA diminishes the optimization of PPRT in accordance with TDM [9].

Since the TDM takes cell delay information as inputs, the optimized PPRT is cell-dependent and thereby library-dependent. Generally speaking, TDM is a sorting algorithm; we can implement a generator coded in high-level languages to facilitate the generation of the speed optimized PPRT.

Later in [20], Oklobdzija et al. prove that TDM is truly optimized, not just improved.

2.2.3 High-Speed Adders

To complete fast multiplication, it must take a fast PPG, a speed optimized PPRT, and also a high-speed adder. In general, fast addition concerns the fast generation of carries or correct prediction of the behavior of carries. In this section, two fast addition schemes – carry-select addition and prefix addition – are introduced with conceptual description.

Fig. 2.7 demonstrates an 8-bit example of a carry-select adder (CSKA) or a conditional-sum adder (CoSA): an operand is partitioned into several blocks (bit width can be fixed or variable). Instead of waiting carry-out from the block LSBs, a CSKA or CoSA duplicates blocks of MSBs, and calculates the sum of the two blocks in parallel by presuming the carry-in bit to be one or zero, respectively. Since the carry-in must be either one or zero, the correct answer can be selected from one of the two MSBs blocks. A two-to-one multiplexer (MUX2) can simply use the

carry-out from the LSBs block as the selection signal to pick the correct answer. This scheme is fast because every block in Fig 2.7 processes in parallel, so for the 8-bit example, the critical path is the addition time of the LSBs block plus a MUX2 selection time. However, since duplicated hardware is used, the area approximately doubles the normal case that uses only one MSBs block.

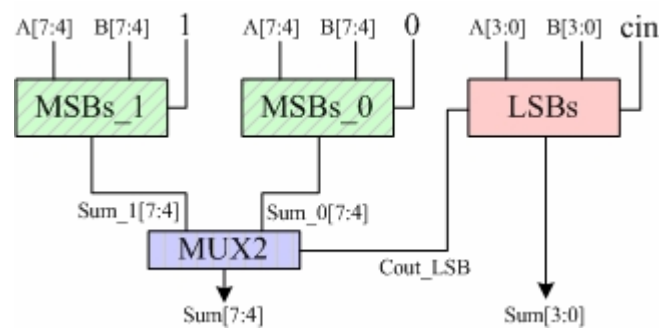


Fig. 2.7. An 8-bit carry-select adder example.

The other popular fashion of fast addition exploits carry lookahead concept that the behavior of carry is actually decided by the carry generation of current inputs or carry propagation of previous carry generation or carry-in. This makes it possible to anticipate the carry. The anticipatory signals are faster because they pass fewer gates, but it takes many more gates to anticipate the proper carry [21]. This concept can be further generalized to parallel prefix computation which observing that block-level generate/propagate signals can also be grouped using prefix operators [3].

Various parallel prefix addition schemes exist such as Brent-Kung [22] and Han-Carlson [23]. In general, the more the parallel-prefix operators are used, the faster the addition completes; however, the actual speed depends on implementation details. A high performance addition scheme on the grounds of Ling addition [24] is presented in [25]. It reduces one logic level over the original Ling Adder in theory

and also minimizes the fan-out of each prefix operator while implemented. In [26] the area is further reduced by fully exploiting the idea of hybrid addition. As a result, considering both in theory or implementation, a high-speed, area-minimized, hybrid Ling adder, which is called “Fong adder” for the remaining context, is presented in [26]. Fig. 2.8 shows a 32-bit architecture of the speed improved hybrid Ling adder [25]: The fan-out of each logic operator is properly taken care of and some modified carry-select adders (MCSAs) are used – the hybrid part – to obtain the result. Fig. 2.9 shows the architecture of a 32-bit Fong adder: Compared to [25], MCSAs with large area are replaced with simple carry select adders (SCSAs) and a ripple-carry adder (RCA), resulting a smaller area cost. This is done by implementing some logic operators working in parallel with prefix operators in each level and hence introduces no timing overhead. Fig 2.10 shows the logic operators used in Fong adder.

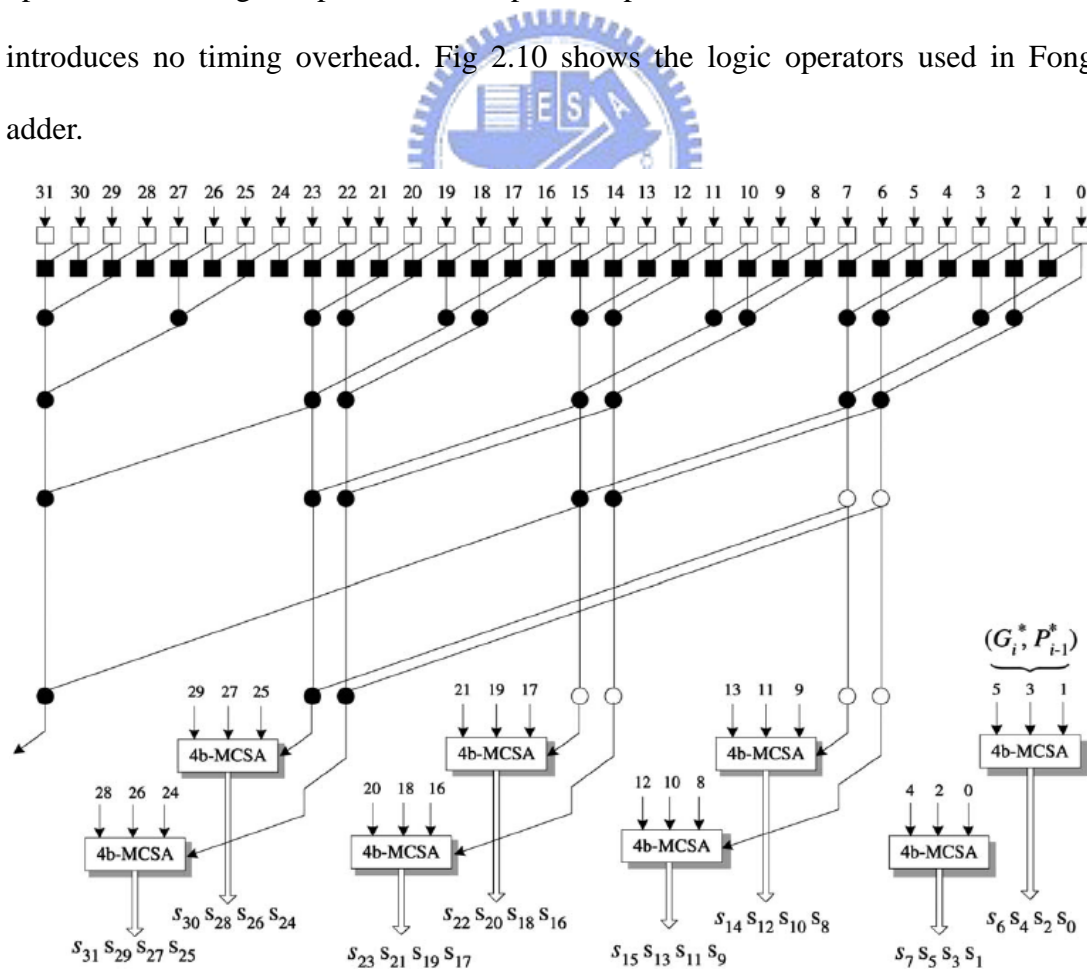


Fig. 2.8. Architecture of a 32-bit hybrid parallel-prefix/carry-select Ling adder.

***This figure is a direct copy of Fig. 8 in [25]**

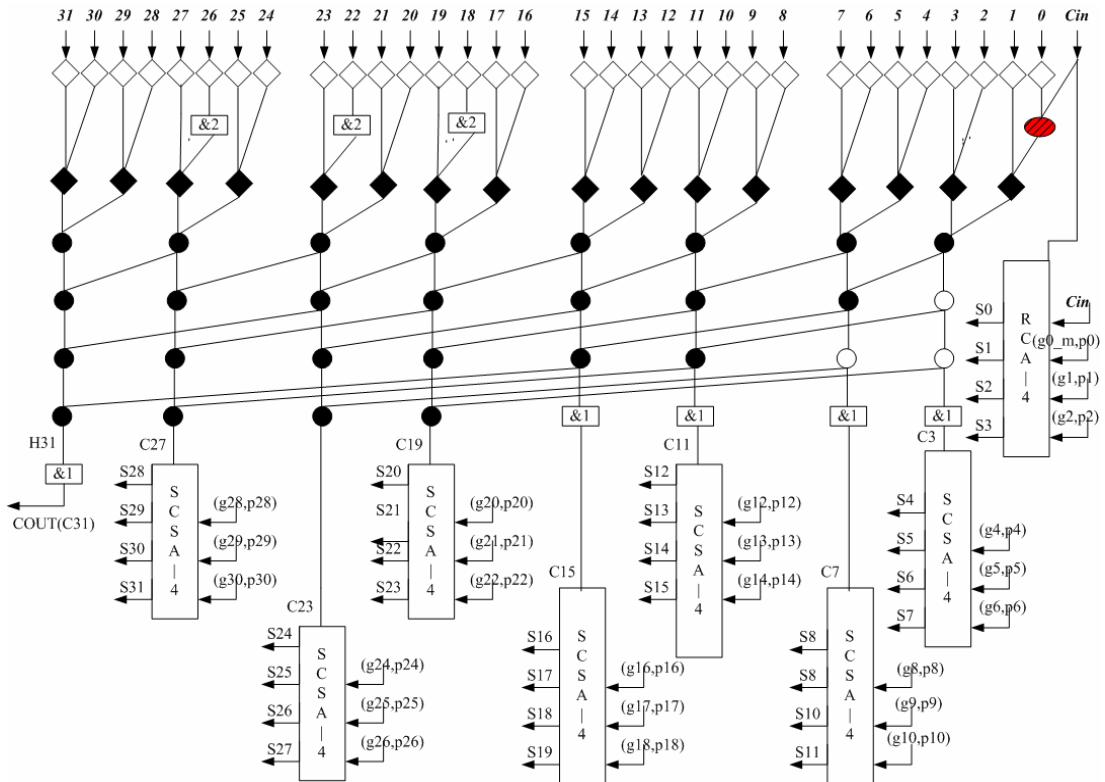


Fig. 2.9. Architecture of a 32-bit scalar Fong adder.

***This figure is a direct copy of Fig. 30 in [26]**

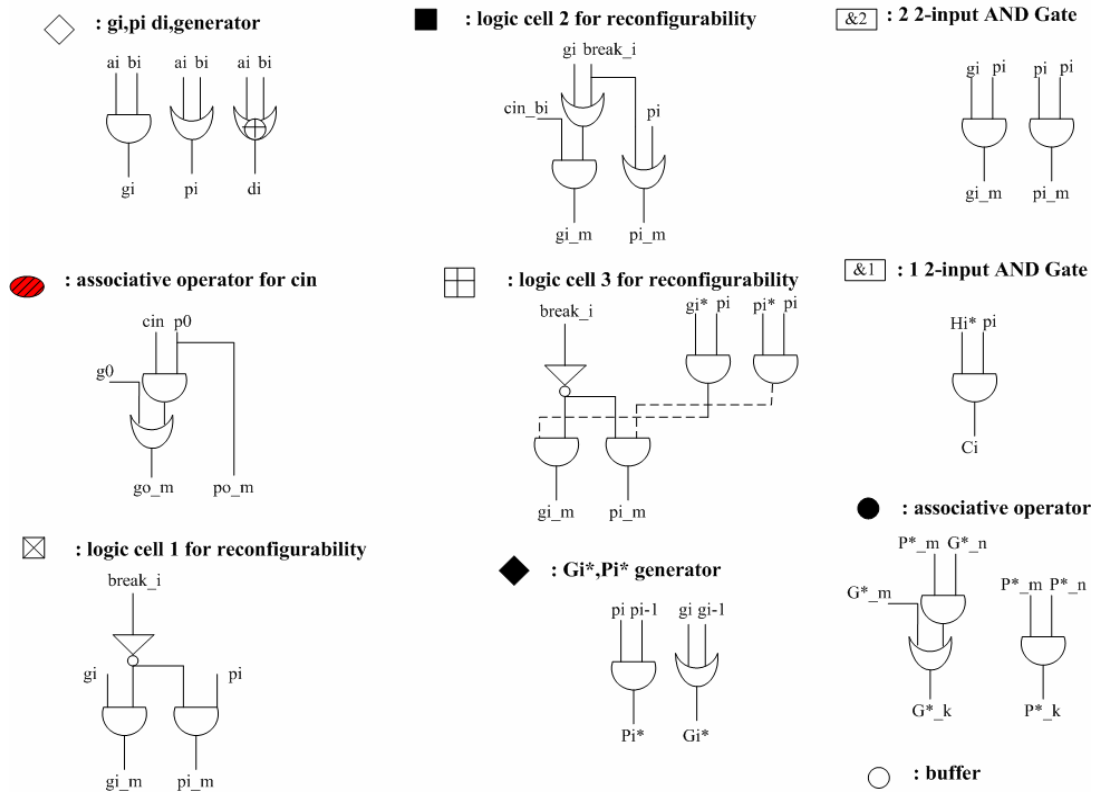


Fig. 2.10. Logic operators used in Fong adder.

2.2.4 Sub-Word Parallelism (SWP)

The utilization of parallel processing leads to a boost in performance. It is a key feature among modern multimedia extensions and DSP processors [6]. A direct implementation of parallel processing is to duplicate hardware such as dual-MAC architecture in ADI-Blackfin® series [27] or quad-MAC architecture in TI-C6000® family [28] DSP processors to increase throughput. However, if given a 16-bit fixed-point (FXP) DSP processor designated for multimedia applications, the original 16-bit datapath is a waste and consumes unwanted power when lower-precision data such as 8-bit pixels are under processing. Duplicating hardware usually damages the hardware utilization rate.

Sub-word parallelism (SWP) or sub-word parallel processing serves as a solution to improve hardware utilization rate and increases throughput by exploiting parallel processing concept. Viewed as a form of Single-Instruction-Multiple-Data (SIMD), SWP is a technique to divide an operand (hardware) into multiple lower-precision ones, conditionally uses the whole or part of the hardware, and thereby raises the hardware utilization rate without introducing significant overhead. For example the same 16-bit scalar hardware can simultaneously process two 8-bit data and hence double the throughput.

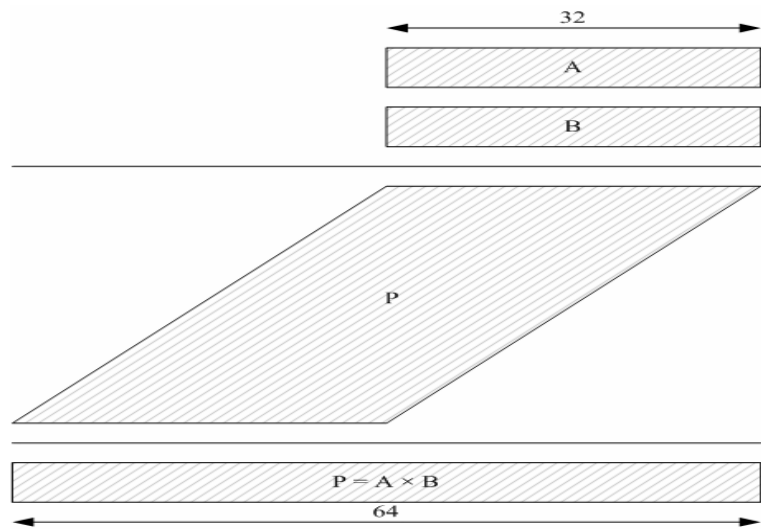
In order for clear and precise explanation, we refer the terms SWP, vectorizing, slicing, segmenting, and partitioning to the same concept as described above, and sub-words (SWs), vectors, slices, segments, and elements are the same product after performing SWP. The term scalar represents a status without utilizing SWP.

SWP concept is of great performance help [29], [30], and SWP datapath units are hence developed. If all units are sub-word parallelized, both scalar and SWP operations can be executed and the computing ability will magnificently increase

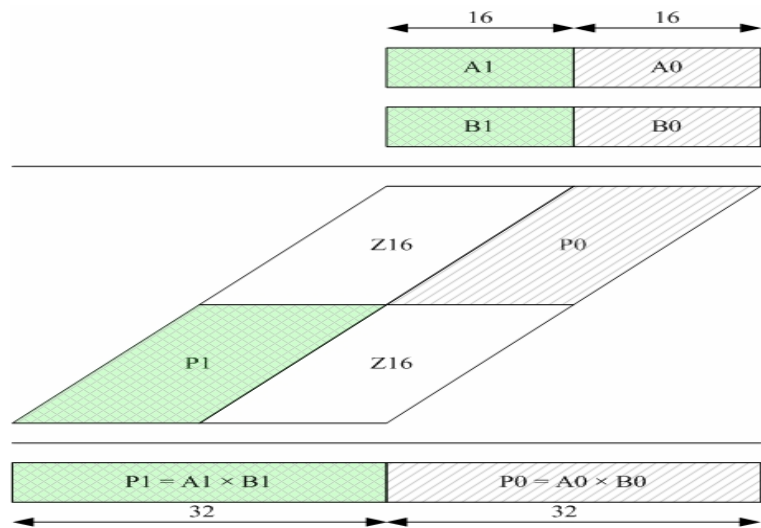
compared to a scalar only architecture. For example, [31], [32], [10], and [26], all propose an SWP adder architecture.

Concerning our work, the SWP multiplier requires an SWP PPG, an SWP PPRT, and also an SWP CPA. The major difference between the scalar and the SWP architecture lies in the existence of the invisible “boundaries” between SWs. As for multiplication, involving PPs accumulation, the carry-out behavior of each SW should be manipulated. In this section, two SWP PPG methods are explained; SWP accumulation will be discussed and compared with the proposed design together in Chapter 3.

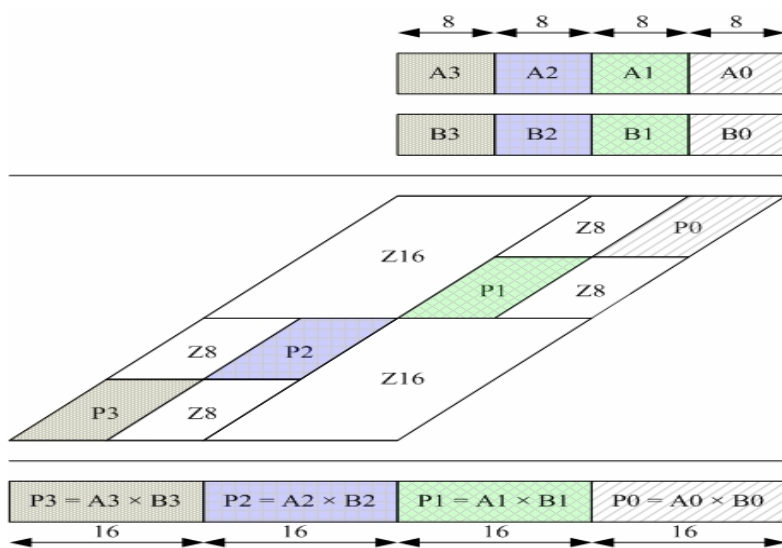
A non-Booth encoded multiplier architecture [6], based on Baugh-Wooley algorithm [33], finds that most bits in the signed PPA overlap those in the unsigned PPA. Concerning SWP, it arranges the PPAs of different SWP modes as shown in Fig. 2.11: Observing that most bits in 8-bit SWP PPA P_0 , P_1 , P_2 , and P_3 in Fig. 2.11c are identical to those of 16-bit PPA P_0 and P_1 in Fig. 2.11b, or 32-bit scalar PPA P_0 in Fig. 2.11a, it indicates most bits in different SWP modes can share with one another. The only effort is on each SW boundary and on managing fields of zeros (Z_8 or Z_{16}). Since the architecture is not modified Booth encoded, it has more PPs and has worse performance in terms of speed; however, a non-MBE architecture usually consumes less power [36]. This MUL/MAC architecture can further functionally integrate the sum-of-square operation into the same PPA without much overhead [34], resulting in a sub-word parallel multiplication and sum-of-square unit (SPMSSU) [35].



(a)



(b)



(c)

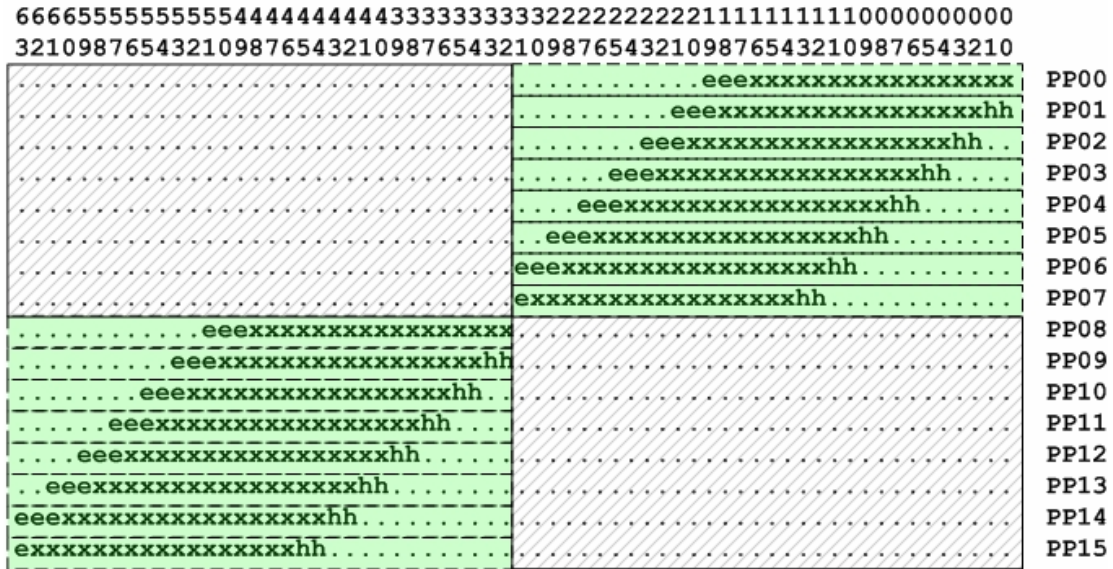
Fig. 2.11. A simplified PPA for 32×32 multiplication in different modes.

A 64-bit fixed-point (FXP) vector MAC architecture capable of supporting multiple precisions is presented in [10] and [11]; it can perform one 64×64 , two 32×32 , four 16×16 , or eight 8×8 bit signed/unsigned MAC operations using essentially the same hardware of a scalar 64-bit modified Booth encoded MAC. These papers also compare different SWP PPA methods and propose one called *shared segmentation*. The shared segmentation method exploits substantially the same concept as done in [6] (described in the preceding paragraph). Most bits in a vector mode overlap with those in another mode, producing similar SWP PPA as Fig. 2.11. It also designs an SWP Wallace PPRT using a special FA at SW boundaries and an SWP CPA using 4-bit CLA blocks. Fig. 2.12 depicts a detailed 32-bit PPA example of the shared segmentation method: Fig. 2.12a, Fig. 2.12b, and Fig. 2.12c illustrates the PPA in 32-bit (scalar), 16-bit, and 8-bit vector mode, respectively. Fig. 2.12d displays the PPs overlap among vector modes; it's clearly shown in the figure that many bits take no effort on selection. It implies there's no need to use a 32-bit, a 16-bit and an 8-bit MBEs to generate three PPs and use three-to-one multiplexers (MUX3s) for selection; All that's required is the 32-bit MBE output associated with some multiplexing at 16-bit and 8-bit vector boundaries.

It just takes some timing and area overhead to “vectorize” a scalar MAC using shared segmentation method. However, this architecture limits the SW combination and places restrictions on constructing the vector PPRT. The vector CPA in this work can also be improved. As a result, the proposed SWP PPA resembles and improves the shared segmentation PPA described in this section.



LEGEND:
 . : null bits or 0
 x : significance bits from PPG
 e : sign encoding bits
 h : hot-ones
 [green box]: significance bit arrays

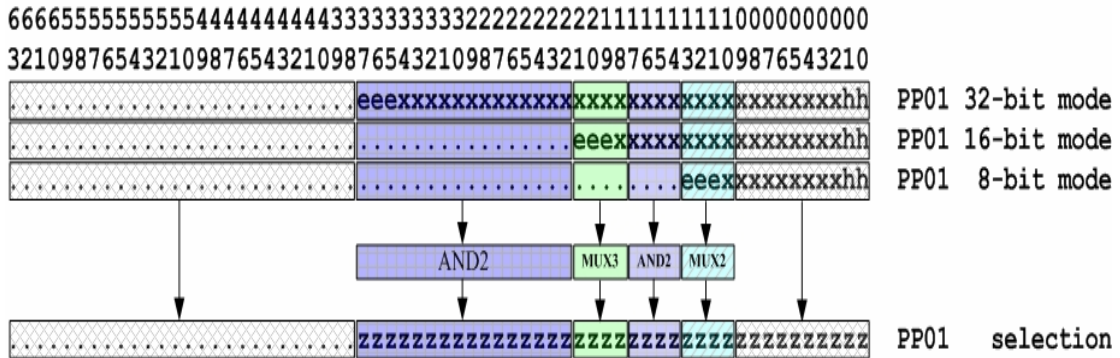
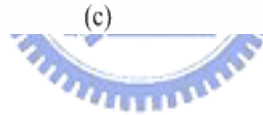


LEGEND:
 . : null bits or 0
 x : significance bits from PPG
 e : sign encoding bits
 h : hot-ones
 [green box]: significance bit arrays
 [hatched box]: Z16

(b)



LEGEND:
 . : null bits or 0
 x : significance bits from PPG
 e : sign encoding bits
 h : hot-ones
 : significant bit array
 : Z16
 : Z8



LEGEND
 . : null bits or 0
 x : significance bits from PPG
 e : sign encoding bits
 h : hot-ones
 z : output PP bits after selection
 : direct selection
 : selection requires a 2-input AND gate (AND2)
 : selection requires a 3-input multiplexer (MUX3)
 : selection requires a 2-input multiplexer (MUX2)

(d)

Fig. 2.12. Shared Segmentation PPA for 32×32 multiplication in different modes.

2.3 Summaries of Previous Works

The multiplication flow of a scalar MBE multiplier can be partitioned into three steps – PPG, PPRT, and CPA. For PPG, a race-free encoding scheme which outperforms other schemes in terms of timing, area, and power consumption is proposed. Sign encoding that prevents sign extension, and hot-one modification that integrates LSB with hot-ones both make the PPA more regular. PPRT often uses levels of FAs to perform carry-save addition, and TDM is an algorithm that helps construct a speed optimized PPRT. The number of PPs after the PPRT is reduced to two. A CPA is used to sum the two PPs to obtain the final product. SWP increases throughput and provides a performance boost in multimedia extensions or DSP processors. Without much overhead, SWP can be applied to MUL/MAC unit by rearranging PPA and the support of SWP accumulation.

The proposed scalar and SWP designs improve and innovate while utilizing some previous works. We'll describe the proposed designs in more detail in the next chapter.

CHAPTER 3

PROPOSED MAC DESIGNS

3.0 Overview

In this chapter, the design methodology of the proposed MAC designs is elaborated. Section 3.1 presents the scalar version of the proposed MAC design: as described in Chapter 2, the MAC unit consists of three parts – PPG, PPRT, and CPA. Based on the scalar MAC architecture, Section 3.2 enunciates the sub-word parallel (SWP) version of the proposed MAC. The differences, improvements and innovations are compared or highlighted in each section and briefly summarized in Section 3.3.

3.1 Scalar MAC (SMAC) Design

3.1.0 Specification

A high performance scalar MAC design which multiplies the N -bit multiplicand ($mcand$) by the N -bit multiplier ($mlier$) with/without accumulating a $2N$ -bit accumulator ($accu$) is proposed. It supports signed/unsigned/mixed-mode operation. Table 3.1 lists the specification of the proposed SMAC. Fig. 3.1 shows the proposed SMAC execution flow. To be noted, the carry-out of final result is also provided.

Table 3.1. Specification of the proposed SMAC design.

Operation: $m_out = accu + mcand \times mlier$ (mode)	
<i>Bit Width of mcand</i>	8/16/32/64
<i>Bit Width of mlier</i>	8/16/32/64
<i>Bit Width of accu</i>	16/32/64/128
<i>Bit Width of m_out</i>	16/32/64/128
<i>Available modes</i>	01:Signed/00:Unsigned/1?:Mixed-mode

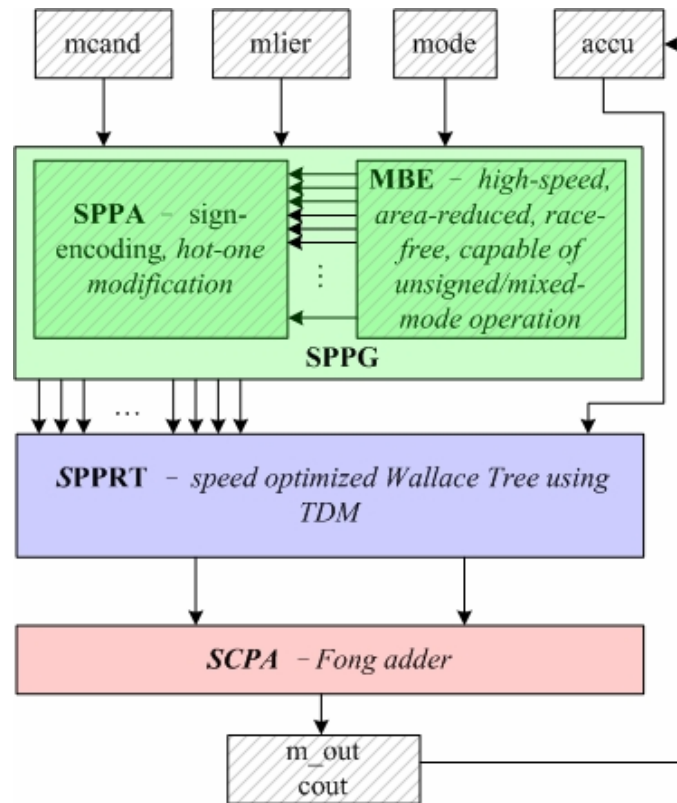


Fig. 3.1. Execution flow of the proposed Scalar MAC design.

3.1.1 Scalar Partial Product Generation (SPPG)

The first phase of SPPG, modified Booth encoding (MBE), is to encode the triplets chosen from the multiplier and then decodes the multiplicand with respect to MBA selection table (Table 2.1). The proposed scalar design favors the race-free concept in [16] that diminishes the energy dissipation, and benefits from the implementation in [9] which saves one logic level and reduces area.

A special operating mode, mixed-mode, is integrated into the proposed scalar design. It forces the multiplicand and the accumulator to be signed, the multiplier to be unsigned, and produces a signed result after operation. Mixed-mode operation has a larger dynamic range, and will be explained in detail in Section 5.4.

However, the MBE scheme in [9] only applies to signed operands. To support unsigned/mixed-mode operation, some modification must be performed on the MBE.

By specification, both unsigned and mixed mode treat *mlier*, the multiplier, as an unsigned number; however, due to two's complement (TC) format natively utilized in MBA, the MSB of *mlier* is the negatively weighted sign bit. It implies $N+1$ bits are required in TC format to fully represent an N -bit unsigned number by forcing the $(N+1)$ th bit, the new MSB and sign bit, to a zero. Owing to the existence of the extra zero, an always positive PP is generated to support unsigned/mixed-mode operation. This is why an N -bit DSP processor with an $(N+1)$ -bit MAC unit supporting unsigned multiplication is frequent.

Briefly speaking, two methods are used to generate the extra PP. The first method uses MBE to generate by assuming $\{0,0,m\}$ as the extra encoding triplet where m stands for the MSB of *mlier*, resulting in a PP equal to zero or *mcand* since the extra triplet is always $\{0,0,0\}$ or $\{0,0,1\}$. The other method uses a similar concept by observing when unsigned/mixed-mode is asserted, a multiplexer with a string of zeros and *mcand* as two inputs and m as the control signal can select the extra PP. The result should be identical with the first method. As a result, both methods help unsigned/mixed-mode operation while neither of them influences on signed operation since the MBE selection of the extra signed-extended triplet $\{m,m,m\}$ or the selection of MUX2s always equals zero. Section 5.1 will detail the way to support unsigned and mixed-mode operation.

Using either method, the logic of the extended triplet $\{s,s,m\}$ or the extended bit s is dependent with m , the MSB of *mlier*, and the assigned mode under execution. If naming *mode[1]* as *mix* (1: mixed-mode; 0: signed/unsigned mode) as well as *mode[0]* as *tc* (1: signed-mode; 0: unsigned-mode), the logic of s is derived as:

$$s = m \cdot tc \cdot (\sim mix) \tag{3.1}$$

In the proposed SPPG, the first method is utilized; besides, signed encoding is also integrated into the MBE, resulting in an N-bit-input and (N+2)-bit-output MBE. Fig 3.2 demonstrates why the output PP requires two-bit extension: assume a 4-bit operand, 1000, is the *mcand*, and the current encoding triplet is {1,0,0} (-2x); it indicates the negation of *mcand* followed by one-bit left shift is to be performed. Due to the need of one-bit left shift, a 5-bit temporary data is required, as shown in the second and third rows in the figure. The bit in bit position 5 is used to save the correct sign that may shift out 5-bit data boundary. If the operating mode is different, this saved bit may differ even if LSBs are the same. Moreover, this bit is also useful for sign encoding. Six bits are hence required for correct representation.

However, the logic of the extended two bits relates to the operating mode, two 2-input AND gates (AND2) are needed at the most significant two bits of the *mcand* to generate these two extended bits. These AND2s are added in the decoder in Fig. 2.4 while there's no logic change on the remaining LSBs. This modification increases a little delay and is still area reduced.

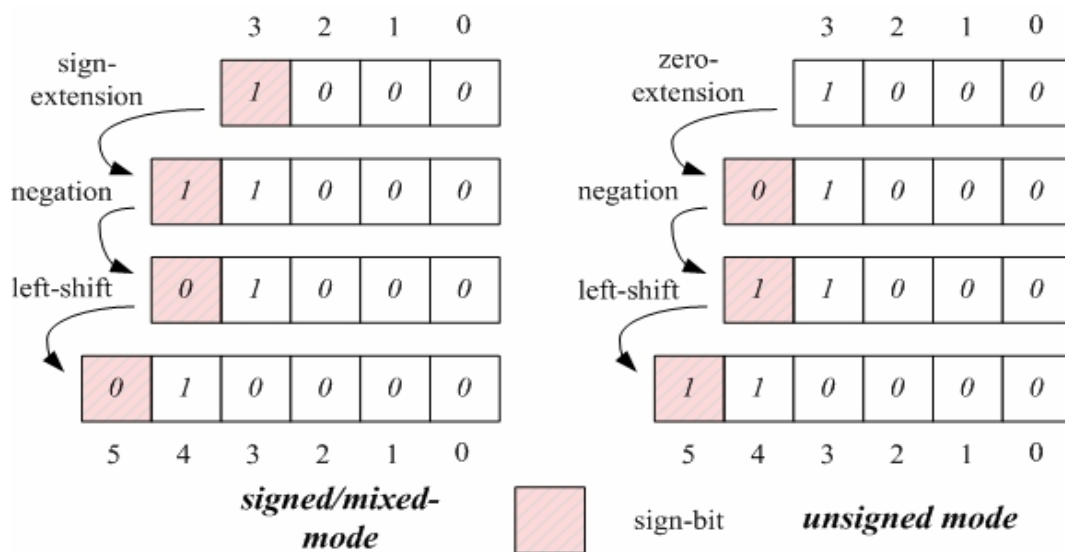


Fig. 3.2. Decoding *mcand* 1000 in different modes when MBE selects -2x.

The second phase of SPPG, arranging scalar partial product array (SPPA), is to properly arrange the PPs generated from MBE. Two techniques, sign encoding (SE) and hot-one modification, are used to arrange the proposed SPPA.

As mentioned in Section 2.2.1, SE is done by replacing the sign-extension bits with $\{p, n, n\}$ for the first PP and $\{1, p\}$ for others, where n stands for the sign bit of the PP and $p = \sim n$. This technique reduces the number of sign-extension bits to two or three and then considerably saves the area and power consumption as bit width grows.

Hot-one modification aligns the hot-one bits, obtained by two's complementing of the preceding PP, all to the left position (*hot2*) with a slight logic change on the LSB of the preceding PP. It makes the LSB end of the PPA shorter and regular.

Both techniques help the proposed SMAC create a narrower-width SPPA which occupies less area, consumes less power, and assists the speed optimization of TDM PPRT. The proposed SPPG is architecturally similar to the PPG in [9].

3.1.2 Scalar Partial Product Reduction Tree (SPPRT)

Three-dimensional method (TDM) [8] is utilized to construct the proposed SPPRT with the architecture of Wallace Tree. A full-adder (FA) is the basic cell to build levels of CSAs. Fig. 3.3 shows the FA cell used in the proposed SPPRT. Concerning TDM, it takes the delay information of each cell used in the tree. Instead of using logic cells like XOR, AND, and OR to build an FA, the SPPRT directly uses the standard high speed FA cell provided by the cell library. This helps not only simplify the generation algorithm but also estimate the delay more accurately. All that is required is to look up in the cell library databook [37] for the delay of six

paths in an FA (*a-to-sum*, *b-to-sum*, *cin-to-sum*, *a-to-cout*, *b-to-cout*, and *cin-to-cout*). A simple software generator is developed to connect the FAs in the SPPRT using TDM.

TDM can be further optimized if the arrival time of each input bit of PPRT is given. It implies that this optimization is cell library dependent and hence and is hard to be reusable. Considering the proposed design, it is easy to obtain reusability. Although the delay information is cell library dependent, to look it up and send it into the software generator to rebuild another SPPRT is effortless since only a standard FA cell is used. However, it's not suitable to use the whole input signal delay profile to build the SPPRT since the synthesizer may generate different SPPG netlist each time the timing constraint varies. The ever-changing delay profile makes the PPRT not speed optimized and perhaps not reusable. As a remedy, logic optimization is left for the synthesizer to make. Since the delay profile is unpredictable and eventually a kind of estimation, the proposed scalar design simply assumes all signals arrive to the SPPRT simultaneously, leading to a reusable TDM SPPRT.

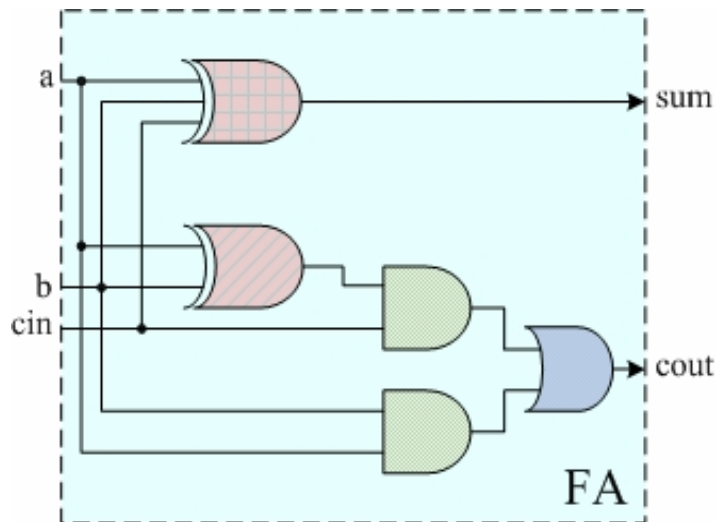
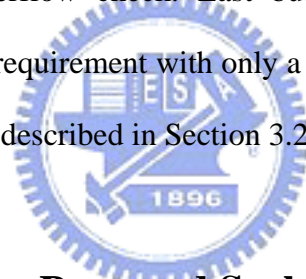


Fig. 3.3. FA cell used in the proposed SPPRT.

3.1.3 Scalar Carry-Propagate Adder (SCPA)

Both adders in [8] and [9] exploit the input operand delay profile to configure a hybrid adder scheme to accelerate addition and reduce area. This again is cell library dependent and hence is hardly reusable. For the proposed scalar design, architectural optimization using delay profile is not recommended. Each bit of two operands of the SCPA hypothetically leaves the SPPRT and arrives at the same time. Fong adder [26] is implemented as the SCPA. The architecture of a 32-bit Fong adder has been shown in Fig. 2.9. There are three main reasons that Fong adder is utilized. First, it outperforms most other adders in terms of delay while it minimizes area cost compared to similar architectures. Second, the carry-out bit is provided so as to perform overflow/underflow check. Last but not least, Fong adder also supports SWP that meets our requirement with only a slight delay and area overhead. The proposed SWP scheme is described in Section 3.2.



3.1.4 Summaries of the Proposed Scalar MAC Design

Fig 3.4 displays the proposed scalar architecture. It is partitioned into SPPG, SPPRT, and SCPA. In SPPG, a race-free encoding scheme is utilized with a high-speed and area-reduced MBE implementation supporting signed, unsigned, and mixed-mode operation. Sign encoding and hot-one modification are applied on the proposed SPPA. In SPPRT, a speed optimized reusable PPRT exploiting TDM is built. As for SCPA, Fong adder is used. Note the figure actually shows the multiplier design. It can easily perform MAC operation simply by feeding the multiplication result into SPPRT as another PP. The proposed SWP design utilizes essentially the same hardware of the proposed scalar design. The way to perform SWP is described in the next section.

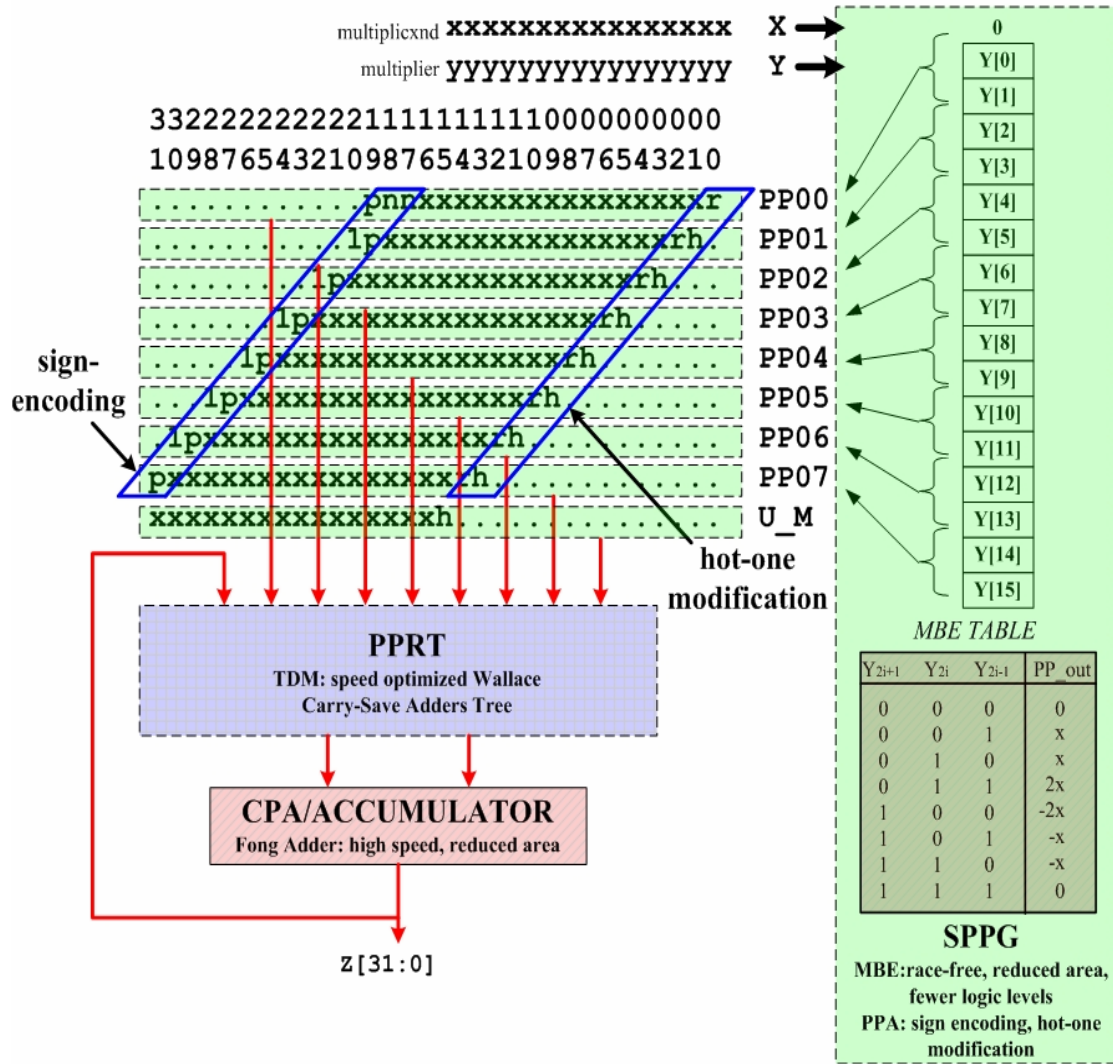


Fig. 3.4. The proposed scalar architecture.

3.2 Sub-Word Parallel MAC (SWP MAC) Design

3.2.0 Specification

A high performance sub-word parallel MAC (SWP MAC) design based on the SMAC architecture is proposed. Table 3.2 lists the specification of the SWP MAC. Kill signals separate SWs and each SW independently processes in its unique mode. Table 3.3 lists the possible sub-word combinations. The detailed SWP reconfiguration scheme is provided in Section 5.3.

Table 3.2. Specification of the proposed SWP MAC design.

Operation: $m_out = accu + mcand \times mlier (mode)(kill)$	
<i>Bit Width of mcand</i>	8/16/32/64
<i>Bit Width of mlier</i>	8/16/32/64
<i>Bit Width of accu</i>	16/32/64/128
<i>Bit Width of m_out</i>	16/32/64/128
<i>Bit Width of a Basic SW</i>	Input:8/Output:16
<i>Bit Width of Each Kill</i>	1
<i>Bit Width of Each Mode</i>	2
<i>Available mode</i>	01:Signed/00:Unsigned/1?:Mixed-mode; independence among all sub-words

Table 3.3. Possible sub-word combinations of the proposed SWP MAC design.

Possible Sub-Word Combinations	
16-bit	(16)
	(8,8)
32-bit	(32)
	(8,8,8,8)
	(8,8,16)
	(16,16)
	(16,8,8)
64-bit	A 64-bit SWP MAC is viewed consisting of two independent 32-bit SWP MACs; it has $5 \times 5 = 25$ possible combinations

3.2.1 Sub-Word Parallel MAC Execution Flow

Fig 3.5 shows the execution flow of the proposed SWP MAC: it is still partitioned into three main parts – SWPPG, SWPPRT, and SWCPA. To apply SWP, some modification should be made in each part – mostly lies in the preprocessing of SWPPG. SWPPG is described in Section 3.2.2; SWP accumulation is divided into

SWPPRT and SWCPA and explained in Section 3.2.3 and 3.2.4, respectively.

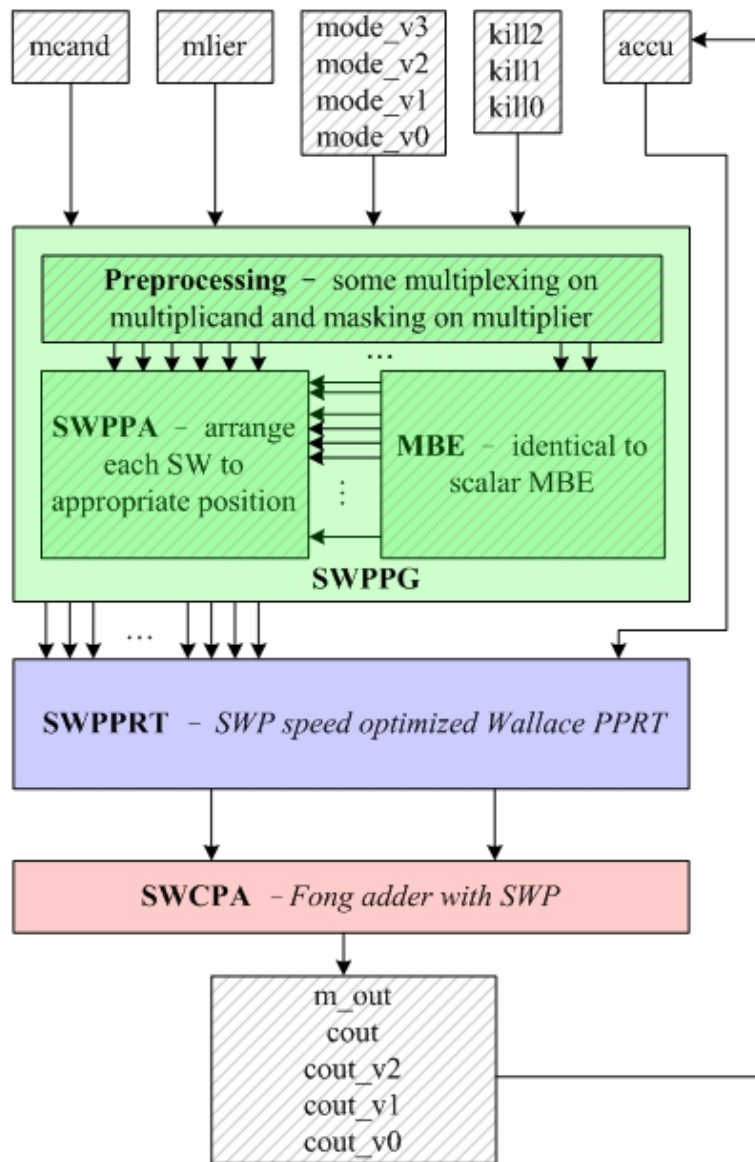


Fig. 3.5. Execution flow of the 32-bit proposed SWP MAC design.

3.2.2 Sub-Word Parallel PPG (SWPPG)

The proposed SWPPG has an identical MBE scheme as used in scalar PPG. The difference lies in the preprocessing on the input operands and the arrangement of the sub-word parallel partial product array (SWPPA). The additional logic for SWP processes mostly in parallel with the SPPG; this enhancement incurs only a

slight timing overhead and some area overhead.

Operand preprocessing consists of two parts – masking and multiplexing on the multiplier and multiplexing on the multiplicand. Fig 3.6 shows a 32-bit example of masking and multiplexing on the multiplier: The bottom SW_0 is the 32-bit multiplier in scalar mode. There is a zero assumed to the right of the LSB for the use of first encoding triplet while there are two $s0$ bits, for the use of unsigned/mixed-mode operation, extended to the left of MSB where $s0$ is generated according to Eq. (3.1). These bits are necessary to complete MBE operation. When SWP modes are under execution, the assumed zero and the extend s bits should be appended to each SW as done in scalar mode. For instance in the top row of Fig.3.6, zeros are assumed at $mlier[-1]$, $mlier[7]$, $mlier[15]$, and $mlier[23]$, and s bits are extended to the left of each SW's MSB. This modification results in 3-bit overlap between SWs, and some bits differ among SWP modes. Therefore mode-dependent multiplexing (selection) or zero-masking are required at these bit positions.

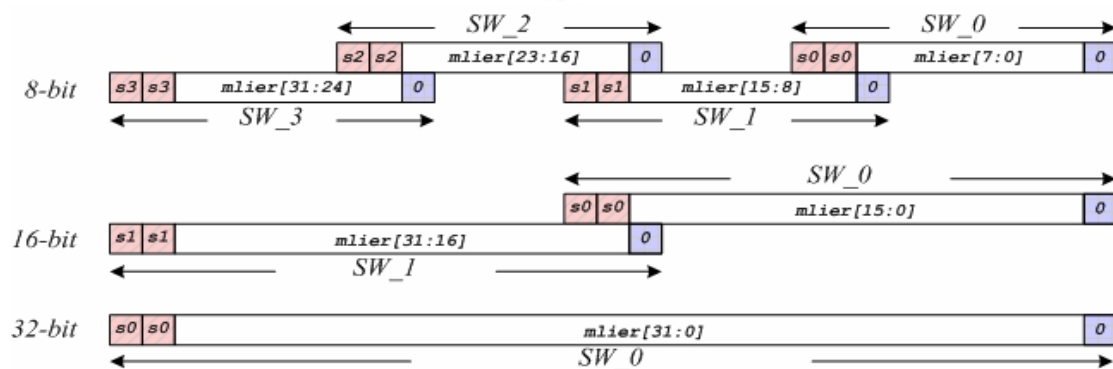


Fig. 3.6. A 32-bit example of masking and multiplexing on the multiplier.

To take an example, the fifth encoding triplet in 32-bit or 16-bit mode is $mlier[9:7]$; in 8-bit mode, $mlier[7]$ should be masked to a zero, resulting a $\{mlier[9:8],0\}$ encoding triplet. This demonstrates the necessity of zero-masking

between SW boundaries. Concerning multiplier multiplexing, it is important to note that the overlapped triplets $\{s0, s0, mlier[7]\}$ between SW_0 and SW_1 in 8-bit mode for the use of unsigned/mixed-mode correction, is not sent to the MBE; instead, the correction PP is generated, simply using an 8-bit MUX2. This multiplexing eliminates the ambiguity in selecting which triplet to MBE. The MSB of SW_0 , $mlier[7]$ in this case, is the selection signal of the MUX2, i.e. when $mlier[7]$ equals 1-bit one, $mcand[7:0]$ as correction PP for SW_0 is required. The same idea can be applied to each SW boundary, avoiding using some 8-bit or 16-bit MBEs to generate correction PPs. All this is required is the scalar 32-bit MBEs.

As for preprocessing on multiplicand, the proposed SWPPA arranges PPA of each SW similar to what has been explained in Section 2.2.4. Some bits overlap and remain the same among different SWP modes while some bits, especially bits at SW boundaries, vary and require mode-dependent multiplexing (selection). The difference is in sign encoding (SE) bits plus one bit saved for the sign of PP and the hot-one modification bits. Fig. 3.7 shows the detailed view of the 32-bit proposed SWP PPA: Fig 3.7a shows the SWP PPA in scalar mode in which we can see 17 PPs including accumulator; SE bits and hot-one modification bits are also shown. Fig 3.7b displays the SWP PPA in 16-bit mode: the SE bits and the sign-bit of PP08 shares those of scalar PP08 while the hot-one modification bits don't share; in contrast, PP01 has a same hot-one modification bits while it differs in SE bits and the sign-bit. Fig 3.7c depicts the 8-bit SWP PPA. Clearly, it tells that the difference mainly lies at SW boundaries. Fig 3.7d exemplifies the selection of PP01 among different modes. Although there exists three modes, only three bit positions actually require a 3-to-1 multiplexer (MUX3) for selection; some take AND2s or MUX2s while some do not demand any selection. As a note, even in the 64-bit proposed SWP design, the proposed SWP PPA requires still MUX3s for worst-case positions.

66665555555554444444444333333333322222222211111111110000000000
 3210987654321098765432109876543210987654321098765432109876543210

..... pnpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	PP00
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP01
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP02
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP03
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP04
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP05
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP06
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP07
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP08
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP09
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP10
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP11
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP12
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP13
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP14
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	PP15
..... pxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	U_M
..... xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxh.	ACC

LEGEND:

- . : null bits or 0
- x : significance bit from PPG
- n : sign fo each PP in scalar mode
- p : ~n
- h : hot2
- █: significant bits arrays

(a)

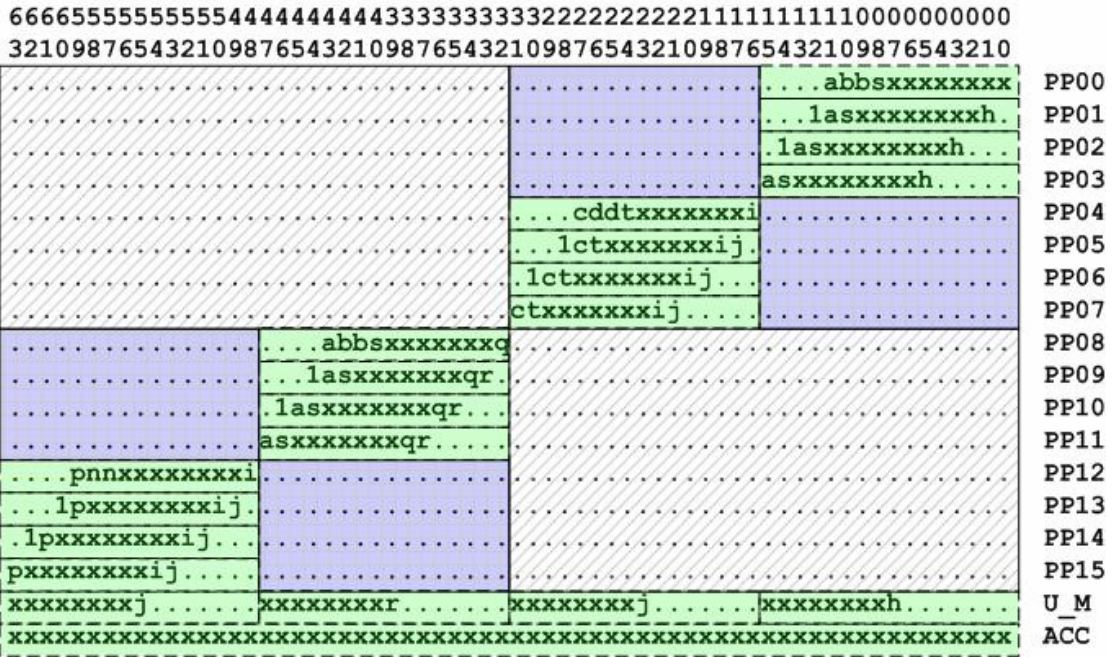
66665555555554444444444333333333322222222211111111110000000000
 3210987654321098765432109876543210987654321098765432109876543210

..... cddtxxxxxxxxxxxxxxxxxxxxxxxx	PP00
..... lctxxxxxxxxxxxxxxxxxxxxxxxxh.	PP01
..... lctxxxxxxxxxxxxxxxxxxxxxxxxh.	PP02
..... lctxxxxxxxxxxxxxxxxxxxxxxxxh.	PP03
..... lctxxxxxxxxxxxxxxxxxxxxxxxxh.	PP04
..... lctxxxxxxxxxxxxxxxxxxxxxxxxh.	PP05
..... lctxxxxxxxxxxxxxxxxxxxxxxxxh.	PP06
..... lctxxxxxxxxxxxxxxxxxxxxxxxxh.	PP07
..... pnpxxxxxxxxxxxxxxxxxxxxxxxxq	PP08
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxqr.	PP09
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxqr.	PP10
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxqr.	PP11
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxqr.	PP12
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxqr.	PP13
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxqr.	PP14
..... lpxxxxxxxxxxxxxxxxxxxxxxxxxqr.	PP15
..... pxxxxxxxxxxxxxxxxxxxxxxxxqr.	U_M
..... xxxxxxxxxxxxxxxxxxxxxxxxxxxr.	ACC

LEGEND:

- . : null bits or 0
- x : significance bit from PPG
- n : sign fo each PP in scalar mode
- p : ~n
- h : hot2
- █: significant bit array
- c : p-bit for 16-bit mode
- d : n-bit for 16-bit mode
- t : MSB of PP
- q : LSB_new for 16-bit mode
- r : hot2 for 16-bit mode
- ▣: Z16

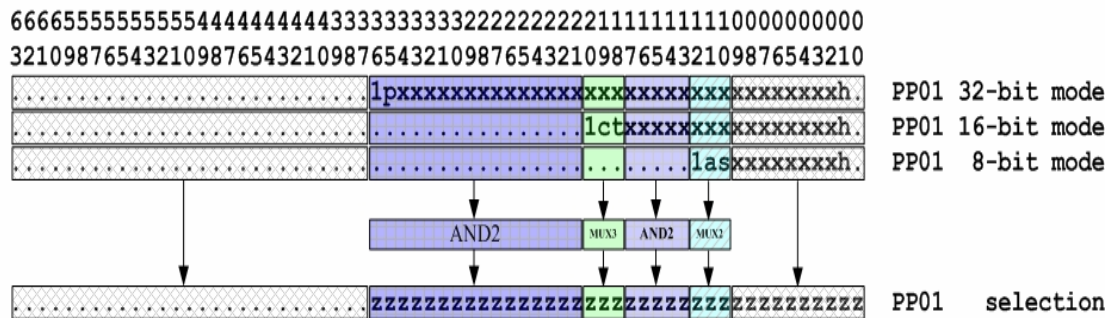
(b)



LEGEND:

.	: null bits or 0	c	: p-bit for 16-bit mode
x	: significance bit from PPG	d	: n-bit for 16-bit mode
n	: sign fo each PP in scalar mode	t	: MSB of temp PP at 16-bit mode
p	: ~n	q	: LSB_new for 16-bit mode
h	: hot2	r	: hot2 for 16-bit mode
a	: p-bit for 8-bit mode		: significant bit array
b	: n-bit for 8-bit mode		: Z16
s	: MSB of temp PP at 8-bit mode		: Z8
i	: LSB_new for 8-bit mode		
j	: hot2 for 8-bit mode		

(c)



LEGEND

.	: null bits or 0		: direct selection
x	: significance bits from PPG		: selection requires a 2-input AND gate (AND2)
p,c	: sign encoding bits		: selection requires a 2-input multiplexer (MUX2)
t,s	: MSB of temp PP		: selection requires a 3-input multiplexer (MUX3)
h	: hot2	z	: output PP bits after selection

(d)

Fig. 3.7. Detailed view of the 32-bit proposed SWPPA with a selection example.

Therefore the preprocessing on multiplicand concerns the generation of SE bits, sign bits of PPs, and hot-one modification bits of each SW and their selection among modes. Table 3.4 lists the truth table of SE bits and sign bits of PPs used in the proposed SWP design. Table 2.5 and Eq. (2.1) have shown the logic of hot-one modification bits. These bits are generated in parallel with scalar MBE without introducing any timing overhead since their logic is not as complicated as an MBE. The area overhead, as demonstrated in Fig. 3.7d, is not huge since most bits share those in the scalar PPA.

Table 3.4. Truth table of sign encoding bits and sign bits of PPs.

<i>tc</i>	Y_{2i+1}	Y_{2i}	Y_{2i-1}	<i>n</i>		<i>s</i>	
				<i>m=0</i>	<i>m=1</i>	<i>m=0</i>	<i>m=1</i>
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	1
0	1	0	0	1	1	1	0
0	1	0	1	1	1	1	1
0	1	1	0	1	1	1	1
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	1
1	0	1	0	0	1	0	1
1	0	1	1	0	1	0	1
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	0
1	1	1	0	1	0	1	0
1	1	1	1	0	0	0	0

tc: 1:signed/0:unsigned; ***Y***: multiplier; ***m***: MSB of SW;
s: sign of corresponding PP; ***n***: SE bit

Thanks to this SWP PPA, the proposed architecture offers more flexible SW combination schemes than previous works if both SWPPRT and SWCPA also support. The SWP combination scheme is controlled by the pre-decoded input *kill* signals. The pre-decoding is performed in parallel with the scalar MBE and thereby does not incur timing overhead. Fig. 3.8 shows the SWP schemes of the 32-bit proposed SWP design: Each *kill* signal conditionally enables/disables the carry-chain. Three *kill* signals provide 8 SW combinations; however, if $\{kill2, kill1, kill0\}$ equals $\{0,0,1\}$, $\{1,0,0\}$, or $\{1,0,1\}$, the middle 16-bit SW obtains a fault PPA since the corresponding PPA has never been generated in this region. Fig. 3.8a to Fig. 3.8e shows the possible five SW combinations; Fig. 3.8f displays an invalid SW combination scheme. For 64-bit design using the proposed architecture, two 32-bit SW halves process in parallel, offering a total of 25 (5x5) different SW combinations.

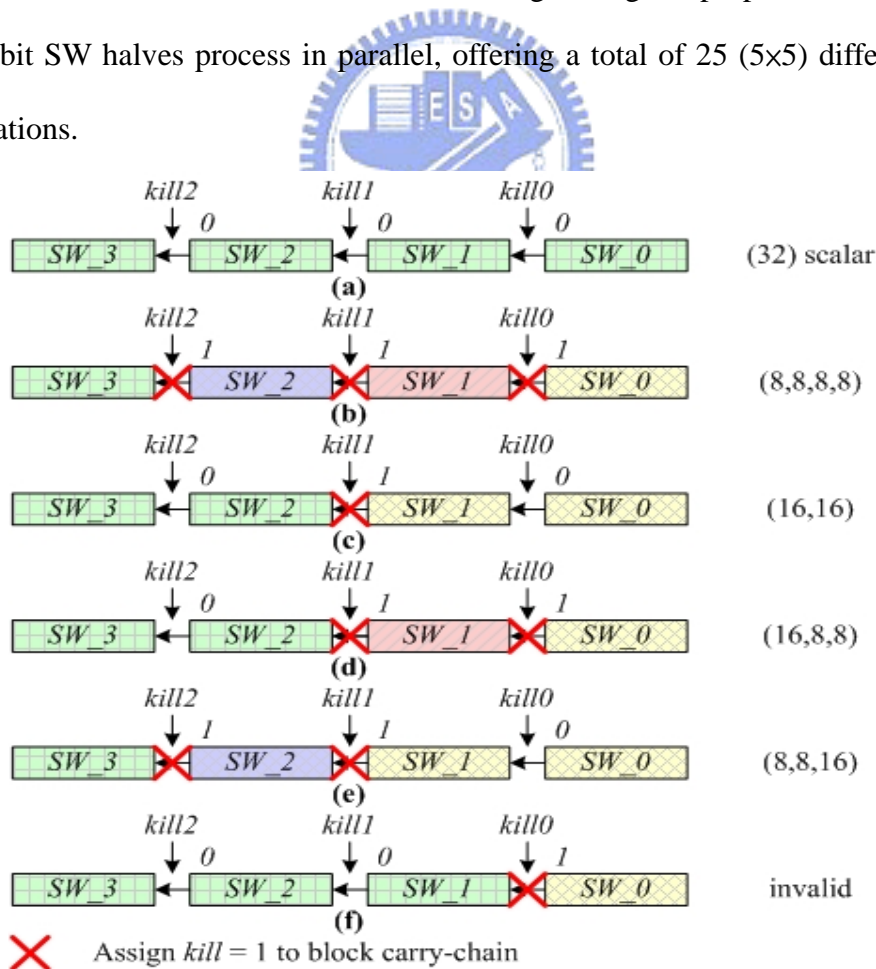
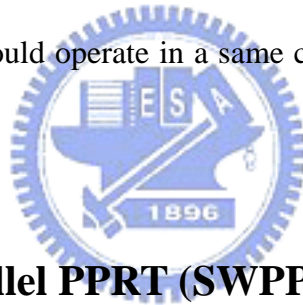


Fig. 3.8. SW combinations of the 32-bit proposed SWP MAC design.

The proposed SWP design is characterized by SWP mode assignment as well; each SW has its own operating mode. To take an example, if a 32-bit SWP operates in 8-bit SWP mode as sketched in Fig. 3.8b, the four SWs don't have to perform the same signed/unsigned/mixed-mode MAC operation at the same time. Instead, each SW assigns its unique *mode* signal, and a total of 81 ($3 \times 3 \times 3 \times 3$) different SW mode assignment schemes are allowed. Moreover a central *mode* signal assigned to all SWs, as used in [10], introduces high fan-out, and which consequently requires buffer insertion. SWP mode assignment ameliorates high fan-out.

Although this modification increases some input ports and places some restrictions on mode assignment, it provides reconfigurability and flexibility for the proposed design. Compared to the 64-bit proposed design, [10] offers only four SW combinations and all SWs should operate in a same central mode, and mixed-mode is not supported.



3.2.3 Sub-Word Parallel PPRT (SWPPRT)

To add SWP in the scalar PPRT, the behavior of carries traversing SW boundaries requires careful manipulation. On the whole, it involves carry-killing (blocking, breaking, disabling, etc) at SW boundaries on each level in the SWPPRT. Both the proposed SWPPRT and the VPPRT in [10] exploit Wallace CSA Tree, using an FA as the basic building block. It implies both designs judiciously manage the carry-out or carry-in of FAs to conditionally break the carry-chain. For example, Fig. 3.9 sketches an image at a SW boundary: Assuming FA_0 is at the MSB of SW_0 and FA_1 is at the LSB of SW_1 , there are two ideas to break the carry-chain – ignoring the carry-in of FA_1 or disabling the carry-out of FA_0 . It implies new FA cells are required, without glue logic, for the use at SW boundaries.

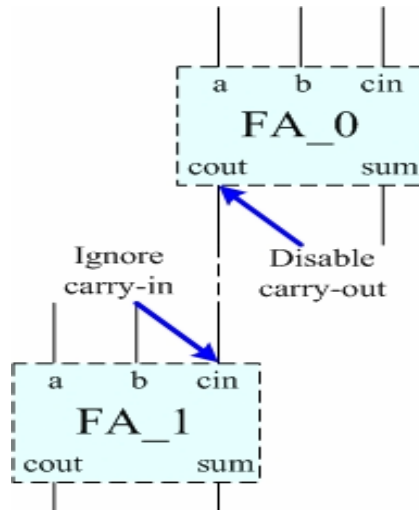


Fig. 3.9. Breaking the FA carry-chain for SWP in SWPPRT.

The first idea is utilized in [10]. Considering an FA used at the LSB of SW₁, the signal *cin* receives the FA carry-out from MSB of the previous SW₀, and hence requires masking on *cin* using the signal *kill*. Fig 3.10 shows the FA with carry-in masking used in [10]. This method does not create a new critical path since the paths *cin* to *sum* and *cin* to *cout* are fast as shown in Fig. 2.6a. Combining *cin* with *kill* using NAND2 incurs no significant delay since the extra gate is in parallel with others. This claim is somehow misleading because it implicitly assumes the delays of all signals are balanced and thereby an FA is always assumed to have its longest path latency all the time. This is often not the case with the real circuit since uneven delay among paths do exist, facilitating the speed optimization using TDM [8]. If using the FA scheme at SW boundaries, in Fig. 3.9 *cout* of FA₀ “must” connect to *cin* of FA₁ and *sum* of FA₀ “must” connect to *a* or *b* of FA₁. This restricted scheme creates a longer critical path going through all *sum* signals and all *a/b* signals since it eliminates the use of TDM to optimize the delay. Furthermore, on the middle levels of Wallace CSA Tree, a lack in *cin* signals for connection of *cout* signals is possible and other FA cells may be required.

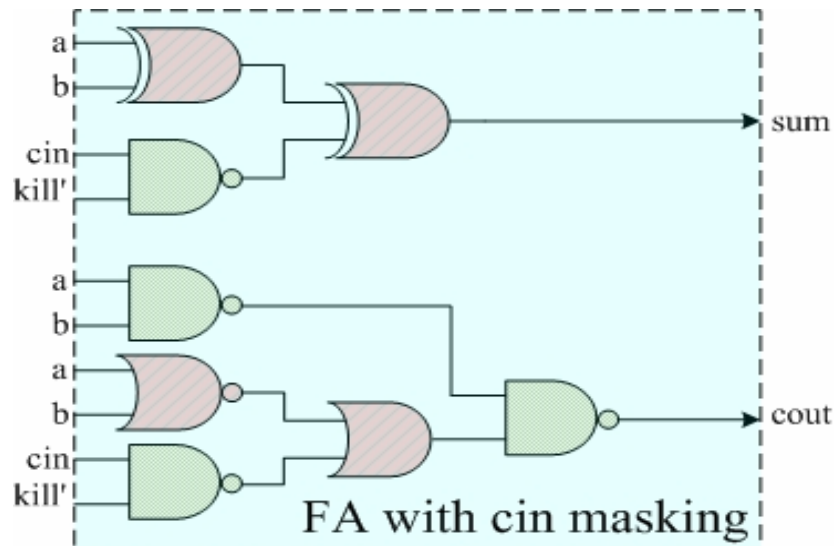


Fig. 3.10. FA with carry-in masking used in [10].

The proposed design utilizes a different idea. Since uneven path latency does exist in the proposed SPPRT, TDM can still be utilized. Generally speaking, the more a flexible signal connection is available, the more the speed of a PPRT is optimized. The proposed SWPPRT concerns conditionally disabling the carry-out as shown in Fig. 3.9. A possible realization of FA with carry-out masking used at the MSB of SW_0 is shown in Fig. 3.11.

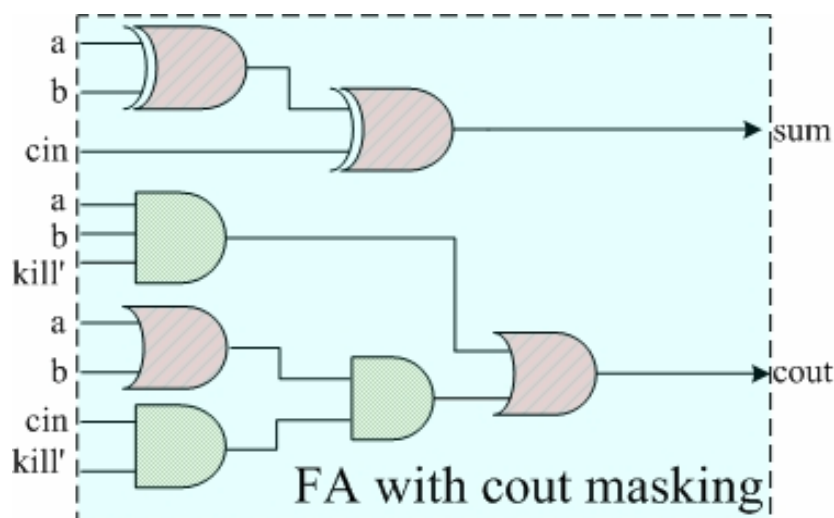


Fig. 3.11. FA with carry-out masking used in the proposed design.

Whenever *kill* is asserted, *cout* must be zero. This modification does not add extra delay to the original FA critical path; however it creates some longer paths compared to the scalar PPRT. It thereby slightly reduces the SPPRT performance since the original FA cells at each MSB of SWs should be replaced by new cells.

This method allows flexible FA connection; TDM is thus still feasible. The proposed SWPPRT outperforms the VPPRT in [10] in theory since TDM speed optimization can still be applied. The SWPPRT has nearly the same performance as the SPPRT. Delay information of the new FA cell is required for TDM; for simplicity, we assume the new FA cell has identical delay information with the original FA cell.

To configure the SW combination scheme, SWPPRT requires identical *kill* signals fed into SWPPA. If properly assigned and connected at SW boundaries, the SWPPRT supports equivalent SW combinations as configured in SWPPA.

3.2.4 Sub-Word Parallel CPA (SWCPA)

There are various SWP adder schemes. The basic idea again is to break the carry-chain across SW boundaries. An easy approach to breaking the carry-chain is to conditionally insert one-bit zero between SWs to both operands. This *annihilates* the carry-chain. When a carry is required, a 1-bit one is simply inserted between SWs to either operand, serving as *propagate* signal without affecting the result. This approach is less relevant to the adder architecture; however, the delay overhead is considerable as bit width grows. If taking consecutive 16 bits as a basic adder block size, a 32-bit SWP adder requires one inserted bit; a 64-bit adder, three; a 128-bit adder, seven. This enlarges the bit width of the CPA to a number unequal to the power of two, and which deteriorates the performance of CPA since in most architectures the block size usually equals the power of two. Fig 3.12 sketches a simple 64-bit SWP adder scheme. The *kill* signal controls the *annihilate* or *propagate* behavior of the carry.

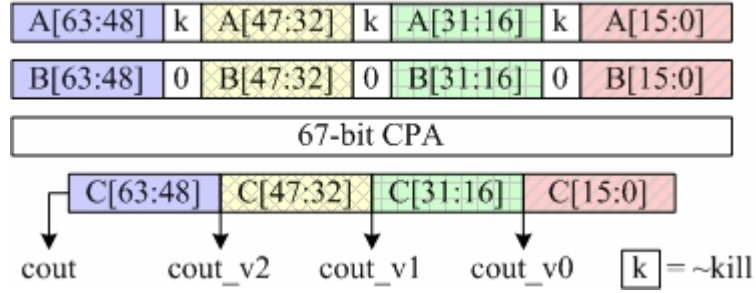


Fig. 3.12. A simple 64-bit SWP adder.

AN SWP adder using 4-bit carry-lookahead generator (CLG) is implemented in [10]. Similar to the SWP method in VPPRT, an AND2 is added to mask the carry-in of the CLG without additional delay. This CLA logic is expressed as:

$$\begin{aligned}
 cout_0 &= g_0 + p_0 \cdot cin \cdot (\sim kill) \\
 cout_1 &= g_1 + g_0 \cdot p_1 + p_0 \cdot p_1 \cdot cin \cdot (\sim kill) \\
 cout_2 &= g_2 + g_1 \cdot p_2 + g_0 \cdot p_1 \cdot p_2 + p_0 \cdot p_1 \cdot p_2 \cdot cin \cdot (\sim kill) \\
 g[3:0] &= g_3 + g_2 \cdot p_3 + g_1 \cdot p_2 \cdot p_3 + g_0 \cdot p_1 \cdot p_2 \cdot p_3 \\
 p[3:0] &= p_0 \cdot p_1 \cdot p_2 \cdot p_3,
 \end{aligned}
 \tag{3.2}$$

where $g[3:0]$ and $p[3:0]$ stand for 4-bit CLG *generate* and *propagate* signal from bit 0 to bit 3, respectively. This adder enjoys the merits of a scalar CLA without large overhead.

As for Fong adder exploited in the proposed SCPA, it also enhances for SWP with minor area and timing overhead because extra operators for breaking the carry-chain are added only at boundary bit positions and work in parallel with the original operators as shown in Fig. 3.13. Fong adder is capable of supporting flexible SW combinations with a basic block size. Concerning the proposed SWP MAC design, we choose a size of 16 bits for lowest granularity. The *break* signals in Fong adder control the behavior of carries across SW boundaries, and thus configure the SWP scheme. The logic of *break* signals are identical to *kill* signals used in the proposed

SWP MAC, and hence meets our specification without any efforts. The carry-out signal of each SW is provided for further possible use. As far as the carry-in signal *cin* of each SW is concerned, Fong adder sets some restrictions to configure scalar or SWP schemes; however, using our design methodology, the negation of *kill* signals equal the *cin* signals of Fong adder at the corresponding bit positions. Compared with SWP CLA design in [10], Fong adder has better performance in terms of delay and area since the optimized Ling addition is essentially faster than CLA by reducing one logic level, and the hybrid adder architecture contributes to a smaller area. The advantage manifests itself as the bit width increases [26].

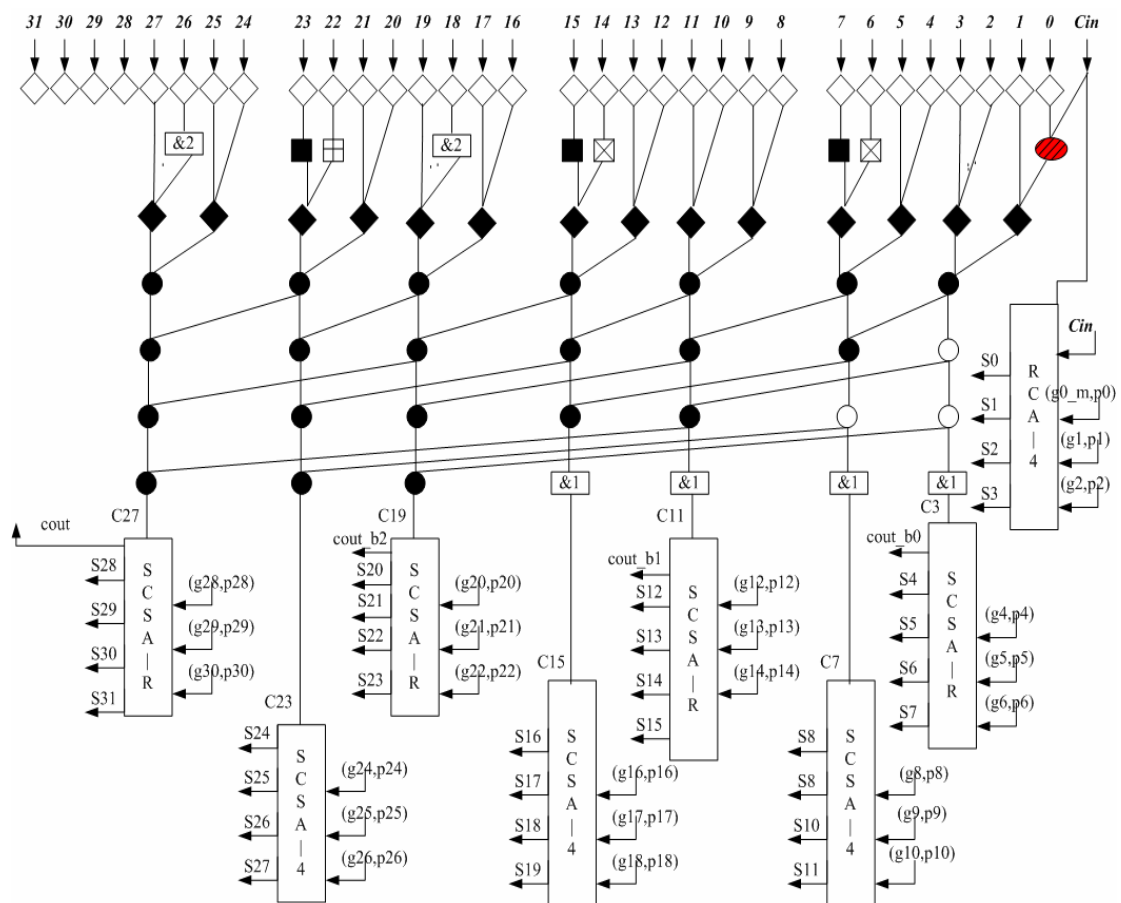
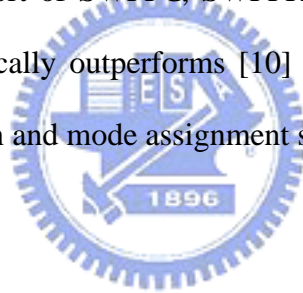


Fig. 3.13. Architecture of a 32-bit Fong adder with reconfigurability.

***This figure is a direct copy of Fig. 47 in [26]**

3.2.5 Summaries of the Proposed SWP MAC Design

Based on the proposed scalar MAC design, the proposed SWP MAC applies SWP to SPPG, SPRT and SCPA. In SPPG, SWP is done by preprocessing on operands and carefully arranging the SWPPA. In SPRT, SWP is done by replacing SW boundary FA cells with a new FA supporting carry-out masking. For SCPA, Fong adder performs SWP by adding some logic operators working in parallel with original operators. The timing and area overhead mostly lies in SWPPG since multiplexing and masking requires several levels of logic. As for SWPRT, the novel method still facilitates the use of TDM to generate SWP speed optimized Wallace Tree. The SWP Fong adder has nearly the same performance as a scalar Fong adder. Due to the support of SWPPG, SWPRT, and SWCPA, the proposed SWP MAC not only theoretically outperforms [10] but also innovatively features more flexible SW combination and mode assignment schemes.



CHAPTER 4

EXPERIMENTAL RESULTS

4.0 Overview

In this chapter, we provide the experimental results of the proposed design. Section 4.1 elaborates the environment for implementation. Section 4.2 provides the data and statistics of the experiment and discusses the experimental results.

4.1 Implementation

To acquire delay and area estimates, the scalar architecture in [10] consisting of an MBE with the encoding scheme in Table 2.2, a regularly connected Wallace PPRT, and a CPA using 4-bit CLA blocks is rebuilt; Synopsys DesignWare IPs (DWIPs) [38] – *DW02_mult* (scalar multiplier) and *DW02_prod_sum1* (scalar MAC) using *wall* synthesis model (MBE-Wallace architecture) – are also chosen for synthetical result comparison. VMAC in [10], an SWP MAC design, is rebuilt for comparison. All designs are implemented in Verilog HDL on register transfer level (RTL) with a same coding style, and then synthesized using Synopsys *Design Compiler* [39] in Artisan 0.18 um standard cell library for UMC 0.18 um silicon technology with and a relatively conservative wireload model *wl10*. The FA cell used in PPRT and the delay information for TDM algorithm exploit the cell *ADDFHX4*, a high speed standard FA cell [37]. All other cells are optimized by the synthesizer. Synopsys *PrimePower* [40] is used for power analysis. In order to prove the correctness of functionality, all designs are simulated by Cadence *Verilog-XL* simulator [41] with patterns cycling through all possible combinations, and the

results are compared to behavior models in Verilog format provided by DWIP. As for verification, Novas *nLint* [42] is used for design rule check with a reusable rule set according to [7]; TransEDA *Verification Navigator* [43], for code coverage analysis; Cadence *Conformal* equivalence checker [44], for logic equivalence (formality) checking between original designs and gate-level netlists. These EDA tools ensure the robustness of designs. Table 4.1 lists the environment for our experiment.

Table 4.1. Environment setup for experiments.

Simulation Environment	
Coding	<i>Verilog</i> HDL
Simulator	Cadence <i>Verilog</i> ® -XL
Synthesizer	Synopsys <i>Design Compiler</i> ®
Power Analyzer	Synopsys <i>PrimePower</i> ®
Cell Library	Artisan <i>UMC 0.18 μm</i> technology
Wire Load Model	UMC <i>wl10</i>
Verification	
Design Rule Check	Novas <i>nLint</i> ® with strict rule set
Equivalence Check	Cadence <i>Encounter</i> ™ <i>Conformal</i> ® Equivalence Checker
Coverage Analysis	TransEDA <i>Verification Navigator</i> ®

4.2 Discussion of Experimental Results

4.2.0 Overview

All results are shown in tabular form with discuss under the tables; besides, the improvement rate of each comparison relative to the proposed design are also provided in percentage.

The result of critical path delay in worst case, area cost at critical timing, and power consumption will be reported and compared at the following sections.

4.2.1 Delay Comparison

Table 4.2 reports the critical path delay in nano-second of all designs in worst case: Table 4.2a lists the delay of scalar multiplier designs; Table 4.2b lists the delay of scalar MAC designs; Table 4.2c lists the delay of SWP multiplier designs; Table 4.2d lists the delay of SWP MAC designs.

Table 4.2. Critical path delay comparison.

(a). Scalar multiplier designs.

Max Delay of SMUL(ns)	DW01_mult (wall)		SMUL in [10]		Proposed SMUL
8-bit	4.06	13.79%	4.35	19.54%	3.50
16-bit	5.25	10.48%	5.92	20.61%	4.70
32-bit	6.18	5.83%	7.61	23.52%	5.82
64-bit	7.40	6.35%	9.11	23.93%	6.93

(b). Scalar MAC designs.

Max Delay of SMAC(ns)	DW02_prod_sum1 (wall)		SMAC in [10]		Proposed SMAC
8-bit	4.31	16.01%	4.74	23.63%	3.62
16-bit	5.55	12.97%	6.43	24.88%	4.83
32-bit	6.50	10.15%	7.65	23.66%	5.84
64-bit	7.81	11.01%	9.37	25.83%	6.95

These two tables clearly show the high-speed advantage of the proposed scalar design. The proposed design on average accelerates DWIPs by approximately 10% and [10] by more than 20%. TDM PPRT contributes the most while Fong adder also has a relatively short delay. The delay of the proposed SPPG is a little longer due to the enhancement for mixed-mode. If removed, the proposed design will have an even better performance. The design in [10] is the slowest since TDM is not applied;

DWIPs outperform [10] by instantiating the same high speed FA *ADDFHX4* cells from the cell library. The delay difference between each two rows in each table is approximately the same since doubling the bit width incurs two more levels of CSA delay into PPRT which is about 1.2 ns in this case. In two tables the corresponding entry relates since adding an accumulator into PPRT incurs one more level of CSA delay or sometimes no delay; this is the feature of Wallace Tree.

(c). SWP multiplier designs.

Max Delay of SWP MUL(ns)	Proposed SMUL		VMUL in [10]		Proposed SWP MUL
16-bit	4.70	-2.98%	6.02	19.60%	4.84
32-bit	5.82	-5.15%	7.75	21.03%	6.12
64-bit	6.93	-4.33%	9.31	22.34%	7.23

(d). SWP MAC designs.

Max Delay of SWP MAC(ns)	Proposed SMAC		VMAC in [10]		Proposed SWP MAC
16-bit	4.83	-4.14%	6.56	23.32%	5.03
32-bit	5.84	-5.82%	7.82	20.97%	6.18
64-bit	6.95	-4.60%	9.65	24.66%	7.27

Table 4.2c and Table 4.2d manifest the high-speed feature of the proposed SWP designs. In all cases, they outperform the SWP designs in [10], and even outperform the scalar designs of DWIPs. The theoretical benefits of the proposed SWPPRT using TDM are also realized. The delay of the proposed scalar design is also compared. It is clear that our SWP method incurs less than 6% timing overhead.

Table 4.3 reports the delay overhead on performing SWP. Table 4.3a shows the case with the designs in [10]. Table 4.3b shows the case with the proposed designs.

Table 4.3. Delay overhead on performing SWP.

(a). Designs in [10].

SWP Delay Overhead of [10]	SMUL	VMUL	Overhead	SMAC	VMAC	Overhead
8-bit	4.35	N/A	N/A	4.74	N/A	N/A
16-bit	5.92	6.02	1.69%	6.43	6.56	2.02%
32-bit	7.61	7.75	1.84%	7.65	7.82	2.22%
64-bit	9.11	9.31	2.20%	9.37	9.65	2.99%

(b). The proposed designs.

SWP Delay Overhead of the proposed	SMUL	SWP MUL	Overhead	SMAC	SWP MAC	Overhead
8-bit	3.50	N/A	N/A	3.62	N/A	N/A
16-bit	4.70	4.84	2.98%	4.83	5.03	4.14%
32-bit	5.82	6.12	5.15%	5.84	6.18	5.82%
64-bit	6.93	7.23	4.33%	6.95	7.27	4.60%

8-bit SWP designs are not available since a size of 8-bit is chosen as the multiplier/MAC basic block. The SWP 8-bit design is thus equivalent to a scalar 8-bit design.

These two tables indicate that the delay overhead on performing SWP in [10] is less than the proposed design. Both the shared segmentation method and the proposed design incurs at most a MUX3 delay for the SWPPA, and both CPA designs have a similar overhead on SWP. This condition thereby infers the decrease in SWPPRT performance since a lower performance FA cell is used to replace the FAs at SW boundaries with a same estimation of delay profile. The degradation of TDM is reasonable; however, the actual delay still significantly outperforms [10]. [10] with a smaller delay overhead is not good because it implicitly assumes all FA

cells introduce a critical timing. The PPRT performance in both scalar and SWP designs is said to be equally slow.

4.2.2 Area Comparison

Table 4.4 reports the area cost in square-micro-meter of all designs at critical timing: Table 4.4a lists the area cost scalar multiplier designs; Table 4.4b lists the area cost of scalar MAC designs; Table 4.4c lists the area cost of SWP multiplier designs; Table 4.4d lists the area cost of SWP MAC designs.

Table 4.4. Area cost comparison.

(a). Scalar multiplier designs.

Total Cell Area of SMUL (μm^2)	DW01_mult (wall)		SMUL in [10]		Proposed
8-bit	11104	-20.81%	15202	11.76%	13415
16-bit	39587	-3.15%	51217	20.27%	40835
32-bit	133335	-3.17%	183334	24.97%	137563
64-bit	469149	-3.57%	710822	31.64%	485917

(b). Scalar MAC designs.

Total Cell Area of SMAC (μm^2)	DW02_prod_sum1 (wall)		SMAC in [10]		Proposed
8-bit	13734	-15.26%	18422	14.07%	15830
16-bit	43070	-6.57%	59010	22.22%	45898
32-bit	141897	-4.25%	196640	24.77%	147932
64-bit	496389	-3.43%	709248	27.61%	513423

These two tables show that the proposed design has nearly the same area cost except for 8-bit, compared with DWIPs. This can be explained that for the 8-bit

design, the area overhead for mixed-mode weighs heavily due to the area is still small, and the area reduction advantage of Fong adder is obscure. The effect of hybrid adder architecture manifests itself as bit width increases while the proposed MBE features area reduction as well. The proposed design is faster and enhanced with mixed-mode operation with approximately the same area. The proposed design significantly outperforms [10] owing to different MBE and PPRT schemes. When doubling the bit width, the area becomes three to four times as large as the original area, and which meets theoretical inference. In two tables the corresponding entry relates since adding an longer accumulator into PPRT incurs one more level of longer CSA logic, and is also with a multiple of three to four as the size doubles.

(c). SWP multiplier designs.

Total Cell Area of SWP MUL (μm^2)	Proposed SMUL		VMUL in [10]		Proposed SWP MUL
16-bit	40835	-23.35%	59282	15.04%	50368
32-bit	137563	-23.49%	210162	19.17%	169883
64-bit	485917	-24.61%	813822	25.60%	605508

(d). SWP MAC designs.

Total Cell Area of SWP MAC (μm^2)	Proposed SMAC		VMAC in [10]		Proposed SWP MAC
16-bit	45898	-20.91%	66825	16.96%	55494
32-bit	147932	-21.21%	222638	19.46%	179302
64-bit	513423	-22.19%	818144	23.32%	627349

These two tables demonstrate that the proposed SWP design outperforms the design of [10] in terms of area cost with approximately 20% overhead. Most overhead is introduced by the SWPPG. Although the proposed SWPPA avoids using

dedicated MBEs to generate PP for each bit width, sign encoding bits, sign bit, hot-one modification bits still need to be generated by designated logic. The area overhead for SWPPRT and SWCPA is not considerable, especially as bit width grows.

Table 4.5 reports the area overhead on performing SWP. Table 4.5a shows the case with the designs in [10] while Table 4.5b shows the case with the proposed designs.

Table 4.5. Area overhead on performing SWP.

(a). Designs in [10].

SWP Area Overhead of [10]	SMUL	VMUL	Overhead	SMAC	VMAC	Overhead
8-bit	15202	N/A	N/A	18422	N/A	N/A
16-bit	51217	59282	15.75%	59010	66825	13.24%
32-bit	183334	210162	14.63%	196640	222638	13.22%
64-bit	710822	813822	14.49%	709248	818144	15.35%

(b). The proposed design.

SWP Area Overhead of the proposed	SMUL	SWP MUL	Overhead	SMAC	SWP MAC	Overhead
8-bit	13415	N/A	N/A	15830	N/A	N/A
16-bit	40835	50368	23.35%	45898	55494	20.91%
32-bit	137563	169883	23.49%	147932	179302	21.21%
64-bit	485917	605508	24.61%	513423	627349	22.19%

These two tables imply that the proposed SWP design has less than double of the overhead of [10]. This is reasonable since hot-one modification is applied to

each SW boundary in each mode. It's a trade-off between speed optimized PPRT and a reduced area SWPPA.

4.2.3 Power Comparison

Estimation of power consumption is performed by *PrimePower* [40]. For designs of a same size, an identical file with a number of 10,000 random patterns plays as the stimulus for estimation. Power estimation is only applied to MAC designs. Simulation results of scalar designs and the proposed SWP design executed in 8-bit SWP mode are reported in milli-Watt in Table 4.6: Table 4.6a lists the power consumption at critical timing as shown in Table 4.2 whereas Table 4.6b lists the power consumption of the same designs processed at a relatively loose timing of 20 nano-second.

Table 4.6. Power consumption comparison.

(a). Power consumption at critical timing.

Power at $T_{crit.}$ (mW)	SMAC in [10]		DW02_prod_sum1 (wall)		SWP MAC		SMAC
8-bit	4.03	17.44%	3.21	-6.30%	N/A		3.43
16-bit	15.22	88.41%	8.99	11.27%	9.48	17.39%	8.08
32-bit	34.30	45.90%	27.28	16.04%	28.52	21.31%	23.51
64-bit	153.70	92.61%	87.89	10.14%	97.97	22.77%	79.80

(b). Power consumption at $T = 20$ (ns).

Power at $T = 20$ ns (mW)	SMAC in [10]		DW02_prod_sum1 (wall)		SWP MAC		SMAC
8-bit	0.96	53.62%	0.69	11.47%	N/A		0.62
16-bit	4.90	150.46%	2.50	27.76%	2.39	22.24%	1.96
32-bit	13.14	90.99%	8.88	29.10%	8.83	28.34%	6.88
64-bit	72.07	159.43%	34.37	23.72%	35.67	28.40%	27.78

Except for the 8-bit comparison with DWIP at critical timing, the proposed designs enjoy a less power consumption. The proposed scalar designs outperform DWIPs even if the area cost of the proposed design is a little bit larger. For some cases, the proposed SWP designs even outperform scalar DWIPs. Race-free encoding for the MBE accounts for the phenomenon.

The proposed design thereby is high-speed, moderate-area, and power-reduced. The power-delay (PD) characteristic is also calculated and listed in Table 4.7 to demonstrate the superiority of the proposed design.

Table 4.7. Power-delay characteristic comparison.

Power-Delay (mW-ns)	SMAC in [10]		DW02_prod_sum1 (wall)		SWP MAC		SMAC
8-bit	19.09	53.77%	13.85	11.56%	N/A		12.41
16-bit	97.86	150.83%	49.88	27.85%	47.70	22.25%	39.02
32-bit	262.40	91.11%	177.32	29.15%	176.25	28.37%	137.30
64-bit	1440.17	159.67%	686.42	23.77%	712.24	28.42%	554.61

CHAPTER 5

APPLICATION NOTES

5.0 Overview

In this chapter, we discuss some important application issues when utilizing the proposed MAC design. Section 5.1 details some frequently used DSP arithmetic operations that can be easily enhanced or extended using the proposed architecture; Section 5.2 provides some overflow/underflow check skills with respect to some common fixed-point (FXP) number representation formats for DSP applications; Section 5.3 describes the way to flexibly reconfigure parameters of the proposed designs to meet users' requirement.



5.1 Functionality Enhancement

5.1.1 Multiply-Accumulate (MAC) Operation

MAC operation is a DSP frequently used operation and is essentially the same as multiplication. Multiplication is treated as a special-case MAC operation without accumulation. A dedicated MAC unit is usually developed to integrate MAC operation with multiplication. In order to implement MAC operation in multiplication time, there are essentially two approaches:

- I. Integrating accumulator data into PPRT as another PP.*
- II. Adding accumulator data after multiplication.*

It seems *approach I* can considerably reduce the delay along the critical path since it does not incur another CPA delay as *approach II* does. Instead, *approach I* takes only a level of CSA delay or in some cases no delay; for example, A PPRT

with five or six inputs both take three CSA levels for reduction; this is the characteristic of Wallace Tree. Fig. 5.1 illustrates the flow of these two approaches.

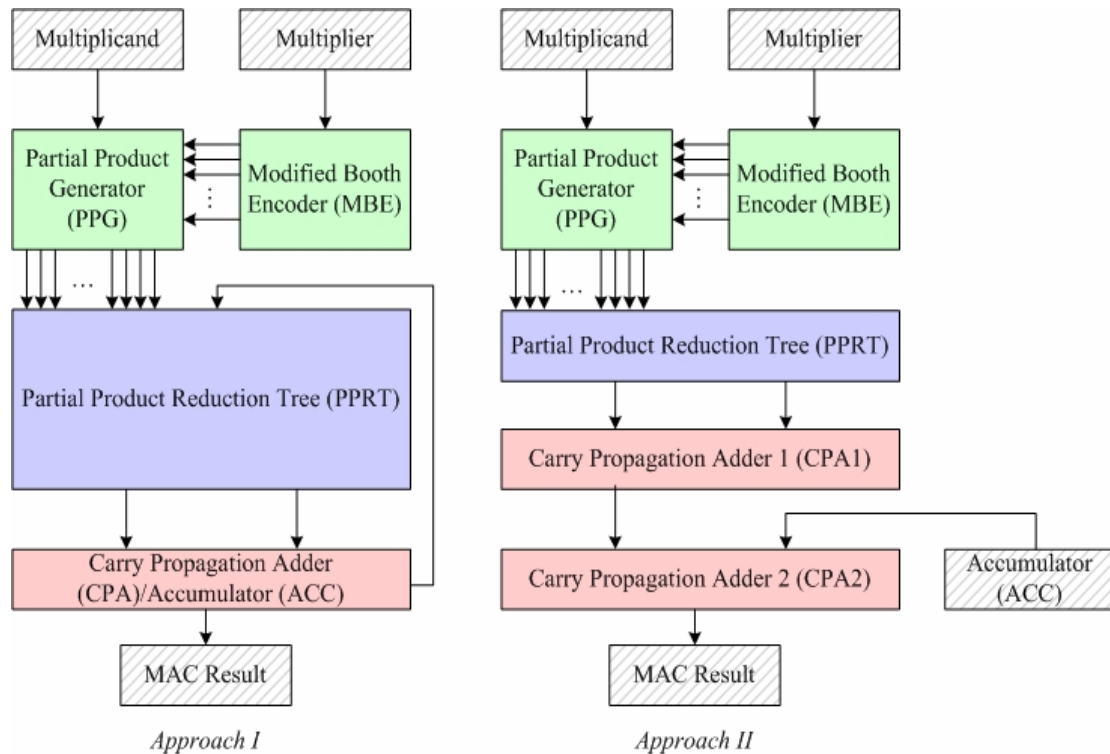


Fig. 5.1. Execution flow of two approaches to completing MAC operation.

However, if a MAC unit with complicated modes and functionalities for DSP application is under consideration, *approach II* is sometimes a better alternative because modification on inputs or maybe some internal temporal signals must be performed to meet the specification assigned. It takes more and complex control signals when using *approach I* to control the temporal data since the all PPG, PPRT, and CPA may require their own control signals. The generation of control signals not only influences performance but increases the design complexity. The choice between *approach I* and *approach II* is a tradeoff.

As a reminder, the input and output (I/O) of PPRT is often a good place to insert pipeline registers if needed.

5.1.2 Multiply-Negate (MAN) Operation

In practical, sometimes the negated multiplication result is required. For example, many DSP processors supports MAC operation with the negated multiplication result (this is another example why we may use *approach II* in the preceding section). This multiply-negate (MAN) or multiply-subtract (MAS) operation has different implementation. Three different feasible methods are described as follows:

- A. *Negation/Two's complement is performed after multiplication:* This is a naïve method since a two's complemter consisting of an inverter associated with an incrementer will unavoidably be used along the critical path. To improve, we can negate the final two partial products from the bottom level of CSA output in PPRT and simply use another level of CSA to sum the special '2' solely for two's complementing use. An incrementer delay is therefore replaced with a level of CSA delay. The naïve flow and the improved flow are shown in Fig. 5.2.

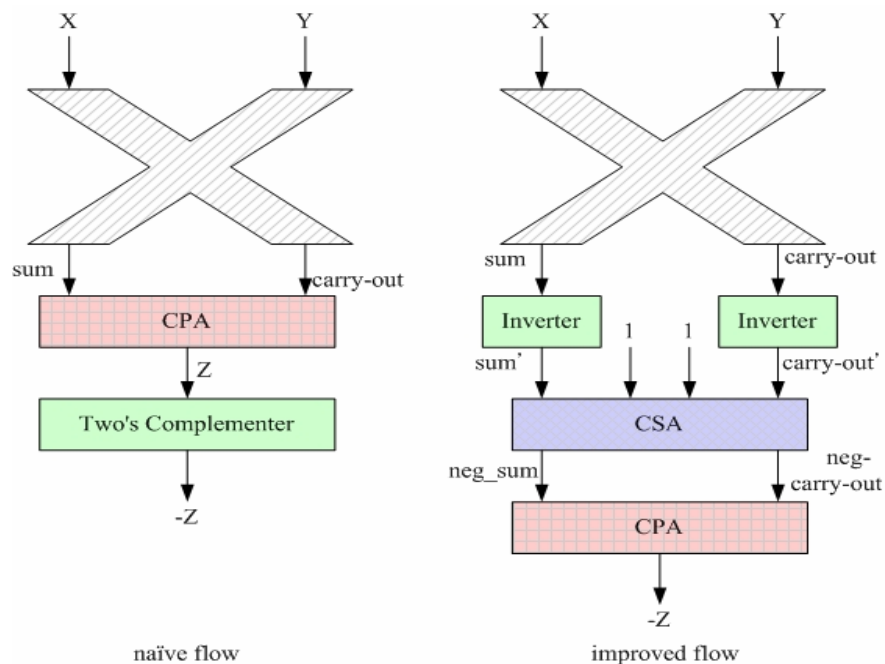


Fig. 5.2. MAN flow of method A.

B. *Negation/Two's complement is manipulated by user:* Compared with *method A*, we now pay attention to the input end. If we can negate either the multiplicand or the multiplier before multiplication, we will get the negated result afterwards. Intuitively, this can be done by performing two's complement on one of the two operands, but this again jeopardizes the performance. Fig. 5.3 shows this modification on input. Another way is to use an instruction such as negation to deal with the problem; unfortunately, it takes one or more clock cycles. The last resort is to mental-calculate the negated operand; this way, however, loses dynamics and user-friendliness.

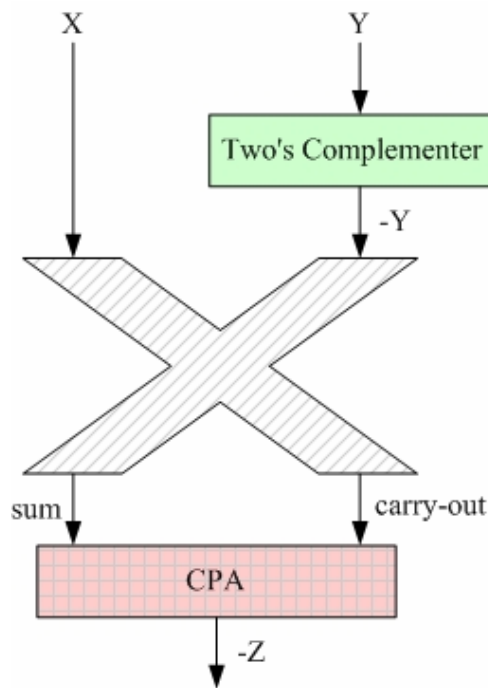


Fig. 5.3. MAN flow of method B.

C. *Negation/Two's complement is performed in multiplication run time:* We are to perform:

$$C = -(A \times B). \quad (5.1)$$

By deduction and inspection, Eq. (5.1) is re-written as:

$$C = (-A) \times B. \quad (5.2)$$

Now replace $-A$ by $\neg A + 1$; we can rewrite Eq. (5.2) as:

$$C = (-A) \times B = (\neg A + 1) \times B = (\neg A) \times B + B. \quad (5.3)$$

Eq. (5.3) indicates we can conditionally invert the multiplicand and simply view the multiplier as another PP. Fortunately sometimes the existence of the extra PP doesn't introduce any timing overhead. This is the characteristic of Wallace PPRT. Fig. 5.4 shows the flow of *method C* and Fig. 5.5 depicts an example of PPA when MAN/MAS operation is under execution. Note the accumulator (*ACC*) and the sign-extended multiplier for negation (*Mlier*) are added at the bottom of the original PPA.

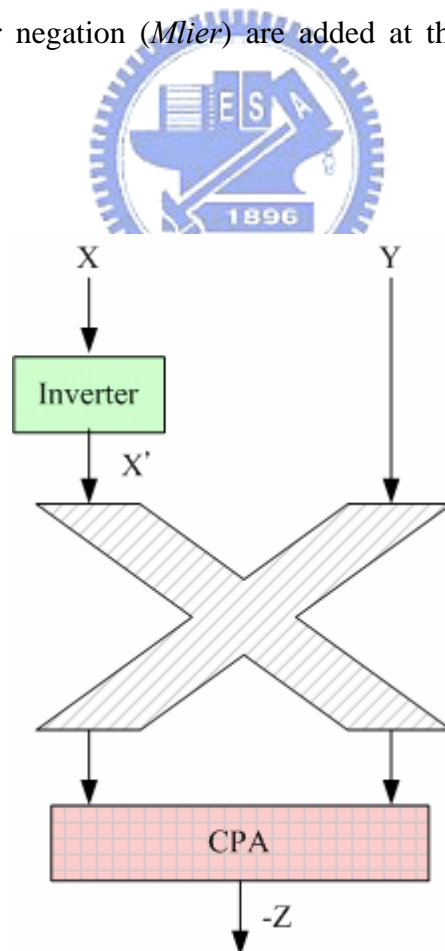


Fig. 5.4. MAN flow of method C.

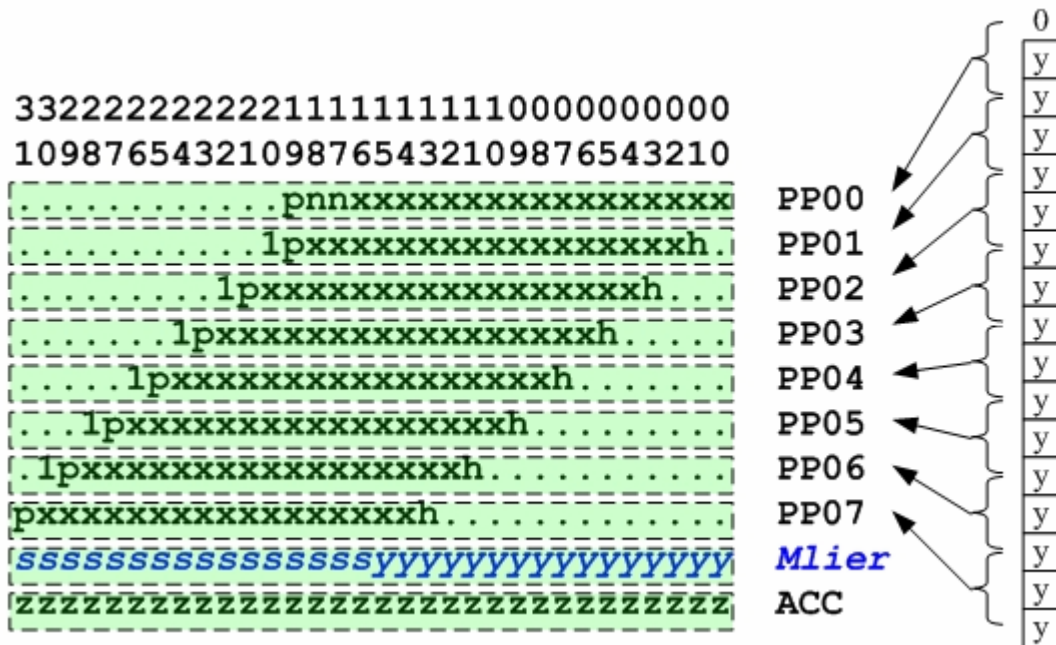


Fig. 5.5. An exempling PPA for MAN/MAS operations using method C.

5.1.3 Unsigned Operation

Substantially, all signals in a circuit or design are simply bit streams; the meaning of a bit stream depends on how a user treats or interprets. For a digital system, unsigned number representation is native and important; for example, most floating point formats represent numbers in a sign magnitude form, completely separating the mantissa (significance) multiplication from the sign handling [12]. Therefore unsigned multiplication must be supported in a DSP multiplier.

Booth's algorithm and modified Booth's algorithm were originally developed to cope with signed multiplication in TC format. The proposed design, based on MBA, then requires some modification to support unsigned multiplication. Section 3.1 has introduced two ways to support unsigned/mixed-mode operation by generating an extra PP. They are supplemented as follows.

The first way generates the extra PP by the MBE, assuming $\{0,0,m\}$ as the encoding triplet where m stands for the MSB of the multiplier. The triplet then

always equals $\{0,0,0\}$ or $\{0,0,1\}$, and the extra PP thereby always equals zero or the unsigned multiplicand. After proper alignment, this PP helps unsigned multiplication since the extra PP ensures a positive result. This modification has no influence on signed mode since sign-extension of m is performed, and $\{m,m,m\}$ is sent to the MBE. The extra PP is hence destined to zero and does not affect the result.

The other way renders that in TC format, the MSB of each operand is the negatively-weighted sign bit. If using TC format to represent an unsigned number, one extra bit is required as the new MSB and sign for each operand. If unsigned mode is under execution, the new MSB is zero-filled; otherwise, sign extension is still applied and does not affect TC representation.

Either way has a similar idea that the negatively-weighted sign bit should be especially taken care of by appending extra bits. The logic of the appending bits for the extra triplet using the first way equal the logic of the new MSB and sign in the second way, and which has been expressed in Eq. (3.1). Fig. 5.6 shows the PPA enhanced with unsigned multiplication: a PP U_M , generated from either way for correction, locates at the bottom of the PPA.

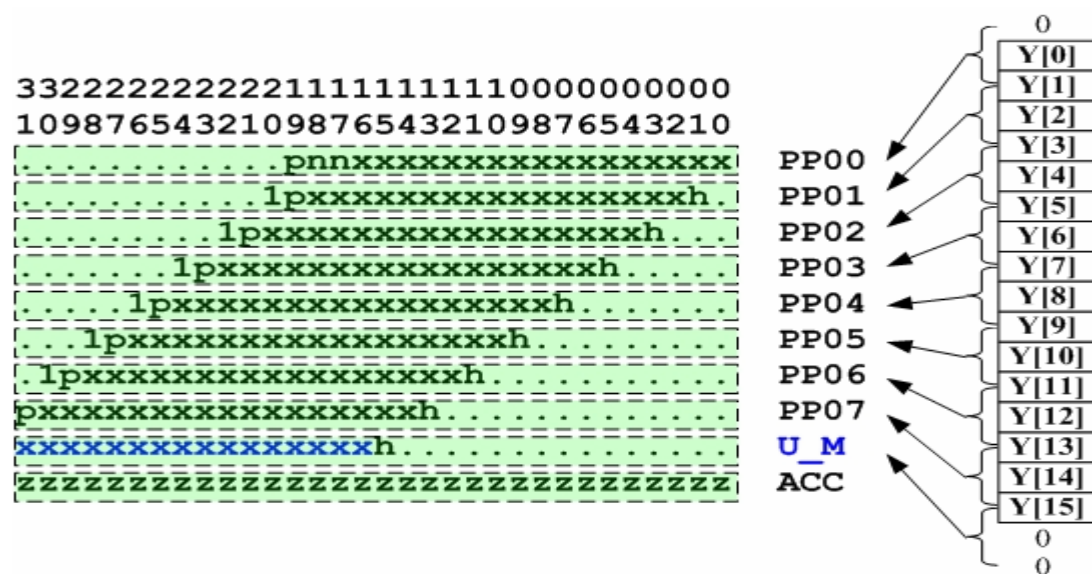


Fig. 5.6. Adding a PP to perform unsigned operation.

Appended operand bits are necessary not only for supporting unsigned MAC operation but also for performing MAN operation (described in Section 5.2) on unsigned numbers. The reason is demonstrated in Fig.5.7: Actually, it takes $N+1$ bits to represent an N -bit unsigned number after negation. Therefore, it is also of great help when performing the MAN/MAS operations on unsigned numbers.

The difference of the two ways lies in the number of appended bits for each operand. Considering the bit width of the generated PP from the MBE, the proposed design appends two bits for each PP as shown in Fig. 3.2. Either way thereby has no difference in the PPA, PPRT and CPA.

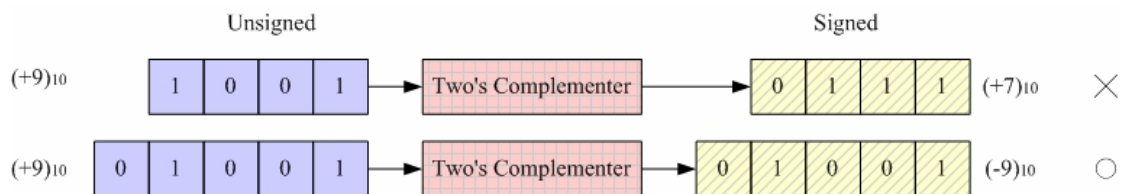


Fig. 5.7. A representation problem on negation of unsigned numbers.

5.1.4 Mixed-Mode Operation

Sign magnitude and TC are two different attempts on representing negative numbers. Both formats, inevitably, trade the MSB significance for a sign bit. This loses the dynamic range especially when representing unsigned numbers since the MSBs should always be zero-filled.

To retain a larger dynamic range, some DSP processors [27] support a special operation mode, called mixed-mode in this thesis, to perform a signed multiplicand operated with a unsigned multiplier, and the product or the accumulator data is still signed. Fig. 5.8 shows the comparison on dynamic range among signed, unsigned, and mix-mode operation: mixed-mode benefits from both the TC representation and a larger dynamic range.

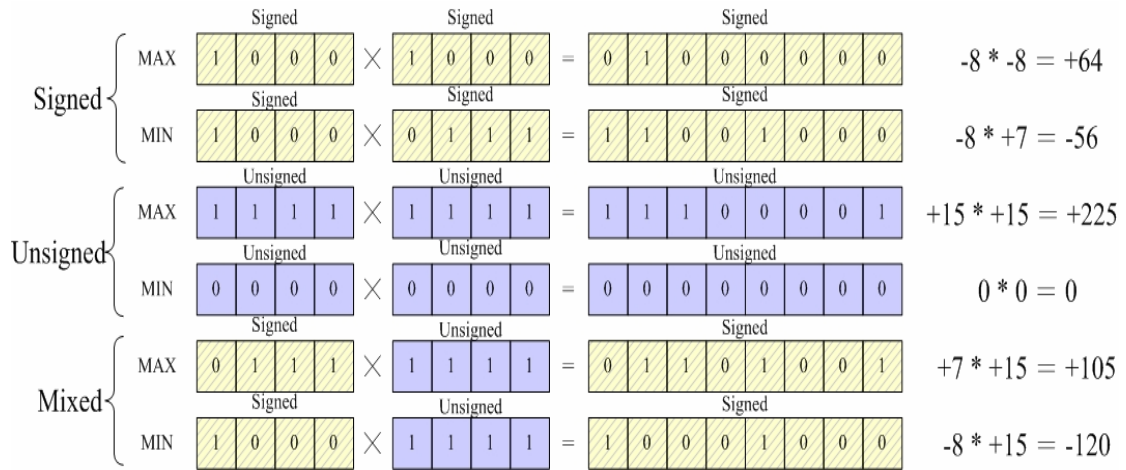


Fig. 5.8. Dynamic range comparison among signed, unsigned, and mixed-mode.

To implement mixed-mode multiplication, it's exactly the same as the modification done for unsigned mode. The multiplier is unsignedly represented; zero-extension is performed. Hence, one extra PP for mixed-mode multiplication is generated and accumulated in the PPRT. Since the result is signedly represented in TC format, the extra PP does not ensure a positive result. Fig. 5.9 shows the PPA that supports MAN/MAS operation, unsigned operation, and mixed-mode operation: the correction PP for mixed-mode is located at the exactly same place of unsigned correction PP; hence they are combine into a single PP, U_M , for unsigned and mixed-mode correction.

As a result, three different numeric FXP data formats – signed, unsigned, and mixed-mode – are integrated into the proposed designs. The overhead on PPRT, as mentioned before, may sometimes be ignored; however there's a little delay added along the critical path since three possible operating modes are under consideration. If mixed-mode is removed, the proposed designs will have even better performance.

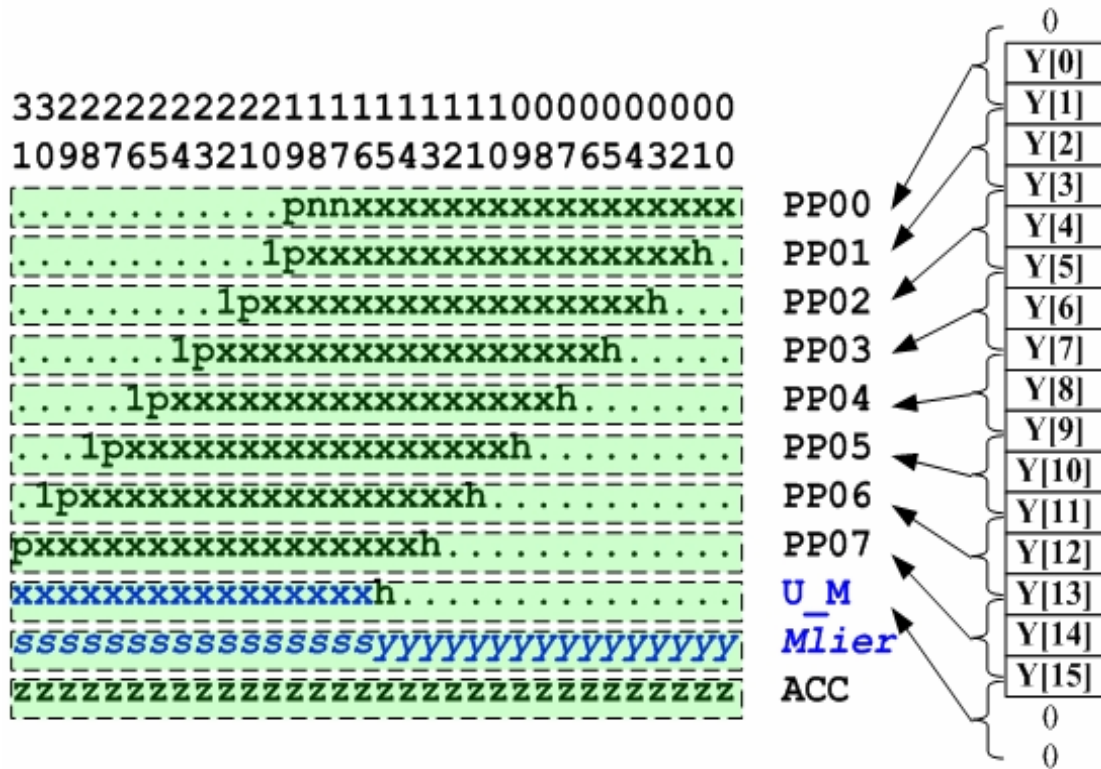


Fig. 5.9. An PPA supporting MAN/MAS, unsigned/mixed-mode operation.

5.2 Overflow/Underflow Check for FXP Numbers

5.2.1 Fixed-Point (FXP) Representation

Compared with floating point (FLP) representation, the fixed-point (FXP) numeric format is say to be fixed since the radix point is assumed to be “fixed” at some bit positions. Taking an N -bit stream for instance, if the “virtual” radix point is at the right of LSB, it is exactly the same as how we interpret the integer data format, and is denote as $N.0$ signed/unsigned integer format. To represent fractions, the radix point can be set at any place except at the right of LSB. The weights of each bit position to the left side of the radix point are larger than one; positions to the right side, smaller. If there are k bits to the left side of the radix point, it is denoted as $k.(N-k)$ format. Most DSP processors use $0.N$ format to represent unsigned fractions and $1.N-1$ format for signed fractions.

FXP representation also features that most mathematic/scientific operations such as addition and multiplication shares integer arithmetic, i.e. even if a fraction number is under consideration, the same adder or multiplier hardware still obtains the correct answer, provided the result is correctly interpreted. For FLP processors, the FLP datapath can not share FXP datapath since the FLP numbers is represented in another data format such as IEEE-754 standard.

FXP representation has a narrower dynamic range than FLP representation, a FXP processor; however, it is less expensive than a FLP processor. FXP processors thereby prevail in DSP applications.

5.2.2 Maintaining Precision & Accuracy

In general, when multiplying two N -bit FXP operands, it takes $2N$ -bit to represent the product without introducing any error. This is sometimes referred to as the law of conservation of bits [2]. The $2N$ -bit register will eventually be insufficient if it is accumulated for some times. Even worse, what if we want to store the $2N$ -bit result back to N -bit registers or memories? When the result of an arithmetic operation exceeds the range of the destination register, important information can be lost.

These precision-related problems occur because the number of bits required to represent the result exceed that of the intrinsic system data format. If not properly controlled, the result goes wrong. Plenty of techniques are developed to resolve the problem such as *saturation*, *input-scaling*, *accumulator with guard bits*, *rounding*, and *truncation*. In the following text we'll discuss saturation and rounding in detail because they are frequent and almost supported in all DSP processors.

Considering the fact that addition and multiplication increase the operand width and the full width result is impractical when operands go on, programmers have

to decide the significant bits of the result. For a $2N$ -bit multiplier product in an N -bit system, the higher half N bits are often viewed as the significant part for fractions while lower half are often significant for integers. This selection retains higher precision from the original data. However, N -bit data width can not fully represent a $2N$ -bit product; some inspections are thus required during half-part selection in order to avoid error and maintain accuracy or higher precision.

Overflow/Underflow check is different between integers and fractions. In the following section, a MAC unit operating on 16-bit FXP data (X and Y) and producing a 32-bit product (M) that may be added or subtracted from a 40-bit accumulator (ACC) will be used as an example to briefly explain how to perform overflow/underflow check. The exempling MAC architecture is typical in modern DSP processors such as in Analog Device's Blackfin™ DSP processors [27].

5.2.3 Saturation & Overflow/Underflow for Integers

In Section 5.2.3 and 5.2.4, a set of pseudo assembly MAC instructions is utilized to demonstrate the way to perform saturation or rounding. Table 5.1 lists the basic instruction types and notations; Table 5.2 demonstrates some pseudo MAC instruction examples; Table 5.3 details some available modes that can be supported by the proposed MAC design.

As far as a FXP MAC is concerned, overflow/underflow may happen when accumulating ACC to M . When overflow/underflow is asserted, saturation means that the overflowed/underflowed data is not viewed as the final result; instead, to maintain higher precision, incorrect data is replaced by the maximal/minimal representable value (still incorrect). To take an example, consider adding base-10 numbers in a system where numbers cannot be larger than two digits in size. If we add the numbers

55, 30, and 20, the result is 5, because two digits are not sufficient for representing the correct result of 105. If saturation mode is applied, we replace 5, which is 100 away from the correct answer, with the maximal representable number 99, which is only 6 away from the correct result [2]. Saturation practically maintains higher accuracy. Fig. 5.10 shows the effect with or without saturation when overflow/underflow occurs.

Table 5.1. Pseudo MAC instruction types and notations.

Instruction Type	Description
$M.HF = X.HF * Y.HF (MODE);$	MUL to a data register half
$M = X.HF * Y.HF (MODE);$	MUL to a data register
$ACC += X.HF * Y.HF (MODE);$	MAC/MAS/MUL to ACC
$M.HF = (ACC += X.HF * Y.HF) (MODE);$	MAC/MAS/MUL to ACC and a data register half
$M = (ACC += X.HF * Y.HF) (MODE);$	MAC/MAS/MUL to ACC and a data register
Note: .HF stands for a register half; it is either .H for high part or .L for low part. (MODE) can be chosen from Table 5.3; += stands for MAC/MAS/MUL operation, respectively.	

Table 5.2. Pseudo MAC instruction examples.

Example	Description
$M.L = X.L * Y.H;$	Multiply the lower half of X with the higher half of Y; treat both operands as a signed fraction; store the result to the lower half of M.
$M = X.H * Y.H (SI);$	Multiply the higher half of X with the higher half of Y; treat both operands as a signed integer; store the result to M.
$M.H = (ACC = X.H * Y.L) (UF);$	Multiply the higher half of X with the lower half of Y; treat all operands as a unsigned fraction; store the result to ACC and the higher half of M.
$M = (ACC += X.L * Y.L) (MF);$	Multiply the higher half of X with the higher half of Y; accumulate in ACC; treat X, ACC, and M as a signed fraction; Y, as a unsigned fraction; store the result to ACC and M.

Table 5.3. Some available modes for pseudo MAC instructions.

MODE	16-bit	32-bit
Default	Format: 1.15 * 1.15 -> 1.15 Range: 0x8000 ~ 0x7FFF Meaning: Multiply two signed 1.15 format numbers; after 1-bit left shift correction, (accumulate and then) round and saturate the result in signed 1.15 format, and store to a register half.	Format: 1.15 * 1.15 -> 1.31 Range: 0x8000_0000 ~ 0x7FFF_FFFF Meaning: Multiply two signed 1.15 format numbers; after 1-bit left shift correction, (accumulate and then) saturate the result in signed 1.31 format, and store to a register.
Unsigned Fraction (UF)	Format: 0.16 * 0.16 -> 0.16 Range: 0x0000 ~ 0xFFFF Meaning: Multiply two unsigned 0.16 format numbers; (accumulate and then) round and saturate the result in signed 0.16 format, and store to a register half.	Format: 0.16 * 0.16 -> 0.32 Range: 0x0000_0000 ~ 0xFFFF_FFFF Meaning: Multiply two unsigned 0.16 format numbers; (accumulate and then) saturate the result in unsigned 0.32 format, and store to a register.
Signed Integer (SI)	Format: 16.0 * 16.0 -> 16.0 Range: 0x8000 ~ 0x7FFF Meaning: Multiply two signed 16.0 format numbers; (accumulate and then) saturate the result in signed 16.0 format, and store to a register half.	Format: 16.0 * 16.0 -> 32.0 Range: 0x8000_0000 ~ 0x7FFF_FFFF Meaning: Multiply two signed 16.0 format numbers; (accumulate and then) saturate the result in signed 32.0 format, and store to a register.
Unsigned Integer (UI)	Format: 16.0 * 16.0 -> 16.0 Range: 0x0000 ~ 0xFFFF Meaning: Multiply two unsigned 16.0 format numbers; (accumulate and then) saturate the result in unsigned 16.0 format, and store to a register half.	Format: 16.0 * 16.0 -> 16.0 Range: 0x0000_0000 ~ 0xFFFF_FFFF Meaning: Multiply two unsigned 16.0 format numbers; (accumulate and then) saturate the result in unsigned 32.0 format and store to a register.
Truncation (T)	Format: 1.15 * 1.15 -> 1.15 Range: 0x8000 ~ 0x7FFF Meaning: Multiply two signed 1.15 format numbers; after 1-bit left shift correction, (accumulate and then) truncate (and saturate) the result in signed 1.15 format and store to a register half.	Truncation is meaning less for 32-bit result; Same as Default mode
Unsigned Fraction with Truncation (TUF)	Format: 0.16 * 0.16 -> 0.16 Range: 0x0000 ~ 0xFFFF Meaning: Multiply two unsigned 0.16 format numbers; (accumulate and then) truncate (and saturate) the result in signed 0.16 format, and store to a register half.	Truncation is meaning less for 32-bit result; Same as unsigned fraction mode.
Mixed Mode Fraction (MF)	Format: 1.15 * 0.16 -> 1.15 Range: 0x8000 ~ 0x7FFFF Meaning: Multiply a signed 1.15 format number with an unsigned 0.16 format number; (accumulate and then) round and saturate the result in signed 1.15 format, and store to a register half.	Format: 1.15 * 0.16 -> 1.31 Range: 0x8000_0000 ~ 0x7FFF_FFFF Meaning: Multiply a signed 1.15 format number with an unsigned 0.16 format number; (accumulate and then) saturate the result in signed 1.31 format, and store to a register.
Mixed Mode Integer (MI)	Format: 16.0 * 16.0 -> 16.0 Range: 0x8000 ~ 0x7FFF Meaning: Multiply a signed 16.0 format number with an unsigned 16.0 format number; (accumulate and then) saturate the result in signed 16.0 format, and store to a register half.	Format: 16.0 * 16.0 -> 32.0 Range: 0x8000_0000 ~ 0x7FFF_FFFF Meaning: Multiply a signed 16.0 format number with an unsigned 16.0 format number; (accumulate and then) saturate the result in signed 32.0 format and store to a register.

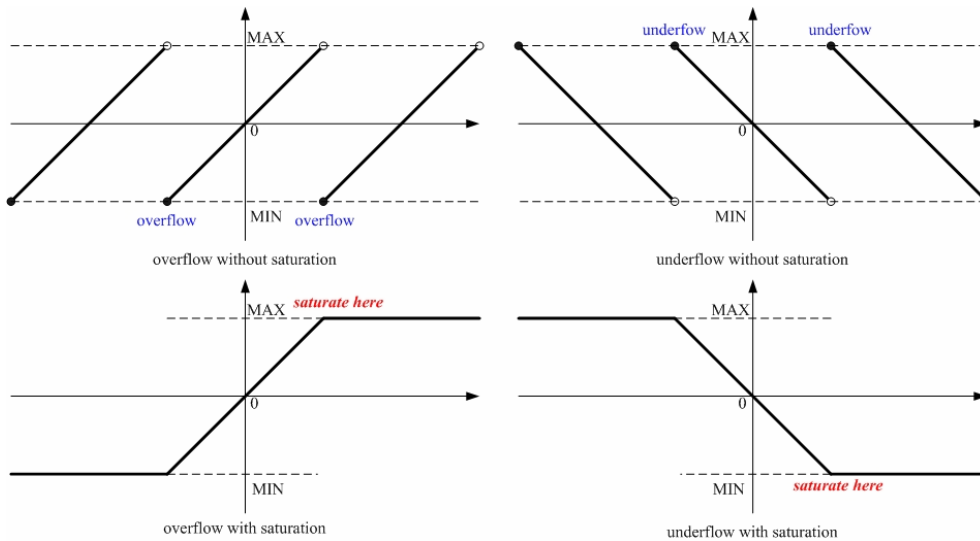


Fig. 5.10. Effect with or without saturation when overflow/underflow occurs.

Possible saturation conditions are listed in Table 5.4 and described as follows:

- *40-bit signed integer:* Overflow/Underflow may occur when accumulating M to ACC . The check scheme is identical to that for signed addition. If two operands are with the same sign, the sign of the result is inspected. Overflow happens when two positive numbers sum to a negative result; it is saturated to the maximum of the 40.0 signed integer format, $0x7F_FFFF_FFFF$. Underflow happens when two negative numbers sum to a positive result; it should saturate to the minimum, $0x80_0000_0000$.
- *32-bit signed integer:* The system may store the 40-bit MAC result to M ; overflow/underflow happens when the 40-bit ACC can not be fully represented. It occurs when $ACC[39:32]$, the guard bits, are not all the same as $ACC[31]$, the sign bit. If $ACC[39:31]$ equal to 9 ones or 9 zeros, saturation is unnecessary; if not, $ACC[39]$ decides saturated value. The maximum of a 32.0 signed integer is $0x7FFF_FFFF$; the minimal, $0x8000_0000$.

- *16-bit signed integer*: The system may store the 40-bit MAC result to *M.H* or *M.L* from $M[15:0]$ or $ACC[15:0]$. If one of the bits in $M[31:16]$ or $ACC[39:16]$ doesn't equal the sign bit, $M[15]$ or $ACC[15]$, it indicates the 32-/40-bit result is different from the 16-bit result and thereby overflow/underflow happens, i.e. if $M[31:15]$ or $ACC[39:15]$ are all ones or all zeros, no saturation is required; if not, $M[31]$ or $ACC[39]$ determines the saturated value. The maximum of a 16.0 signed integer is 0x7FFF; the minimum, 0x8000.
- *40-bit unsigned integer*: The logic of the *carry-out* bit of the 40-bit CPA is equivalent to the logic of this overflow condition since it indicates 40 bits are not enough for representation. Underflow occurs when operating MAN/MAS operation with a negative result; this should use one extra bit to check as shown in Fig. 5.7. If supported, it resembles the underflow of 40-bit signed integer. The maximum of a 40.0 unsigned integer is 0xFF_FFFF_FFFF; the minimum, 0x00_0000_0000.
- *32-bit unsigned integer*: Overflow concerns the positive sign extension and the *carry-out* of the 40-bit CPA. It demands the 9-bit value $\{cout, ACC[39:32]\}$ equals zero. Underflow check resembles that of 32-bit signed integer. As a note, multiplication in this case asserts no overflow due to the law of conservation of bits. The maximum of a 32.0 unsigned integer is 0xFFFF_FFFF; the minimum, 0x0000_0000.
- *16-bit unsigned integer*: Similar to 32-bit unsigned integer overflow check, it requires to check whether each of the 25 bits, $\{cout, ACC[39:16]\}$ or each of the 16 bits, $M[31:16]$, equals zero, depending on the source. Underflow check resembles that of 16-bit signed integer. The maximum of a 16.0 unsigned integer is 0xFFFF; the minimum, 0x0000.

Table 5.4. Possible saturation conditions using the exempling architecture.

Type	Width	Overflow	Max.	Underflow	Min.
SI	40	M[31]=ACC[39]=0 and result ACC[39]=1	0x7fffffff	M[31]=ACC[39]=1 and result ACC[39]=0	0x800000000
	e.g.	ACC += X.L * Y.L (SI); // ACC_old = 0x 7fc0000000; // X.L = 0x8000; // Y.L = 0x8000; /* M= 0x40000000; The accumulated result is 0x800000000 which overflows. The result should saturate to 0x7fffffff */		ACC += X.L * Y.L (SI); // ACC_old = 0x8000000000; // X.L = 0x0001; // Y.L = 0xffff; /* M = 0xffffffff; The accumulated result is 0x7fffffff which underflows. The result should saturate to 0x8000000000 */	
	32	^ACC[39:31]=1 and ACC[39] = 0	0x7fffffff	^ACC[39:31]=1 and ACC[39] = 1	0x80000000
	e.g.	M = (ACC += X.L * Y.L) (SI); // ACC_new = 0x 7fc0000000; /* Overflow condition asserts; the result saturates to 0x7fffffff */		M = (ACC += X.L * Y.L) (SI); // ACC_new = 0x8000000000; /* Underflow condition asserts; the result saturates to 0x800000000 */	
	16	^ACC[39:15] = 1 and ACC[39] = 0, or ^M[31:15] = 1 and M[31] = 0	0x7fff	^ACC[39:15] = 1 and ACC[39] = 1, or ^M[31:15] = 1 and M[31] = 1	0x8000
	e.g.	M.L = X.L * Y.H (SI); // X.L = 0x8000; // Y.H = 0x8000; /* M = 0x40000000; Overflow asserts; the result saturates to 0x7fff */		M.L = X.L * Y.H (SI); // X.L = 0x0004; // Y.H = 0x8000; /* M = 0xffff0000; Underflow asserts; the result saturates to 0x8000 */	
UI	40	cout = 1;	0xffffffff	depends on system	0x000000000
	e.g.	ACC += X.L * Y.L (UI); // ACC_old= 0xffc0000000; // X.L = 0x8000; // Y.L = 0x8000; /* M = 0x40000000; The carry-out of the accumulated result is 1 which overflows. The result saturates to 0xffffffff */		ACC -= X.L * Y.L (UI); // ACC_old= 0x0000000000; // X.L = 0x0001; // Y.L = 0x0001; /* M = 0x00000001; ACC_new equals -1 in decimal which saturates to 0x0000000000 */	
	32	{cout,ACC[39:32]}=1	0xffffffff	depends on system	0x00000000
	e.g.	M = (ACC += X.L * Y.L) (UI); // ACC_new= 0x0200000000; /* Overflow asserts; the result saturates to 0xffffffff */		M = (ACC -= X.L * Y.L) (UI); /* M = 0x00000001; ACC_new equals -1 in decimal which saturates to 0x0000000000 */	
	16	{cout,ACC[39:16]}=1; or M[31:16] = 1	0xffff	depends on system	0x0000
e.g.	M.L = X.L * Y.H (UI); // X.L = 0x0002; // Y.H = 0xffff; /* M = 0x0001ffc; Overflow asserts; the result saturates to 0xffff */		M.L = (ACC -= X.L * Y.L) (UI); /* M = 0x00000001; ACC_new equals -1 in decimal which saturates to 0x0000000000 */		

5.2.4 Rounding of Fractions

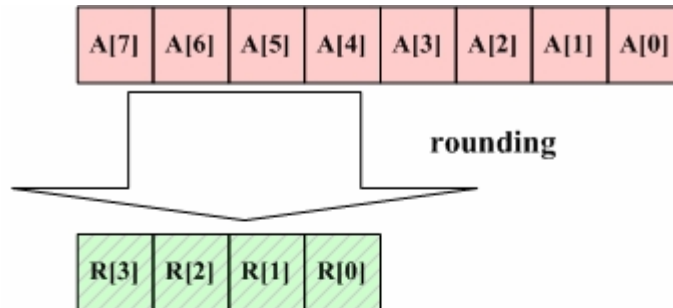
Unlike integers, the significant bits of fractions are the higher half. If it is to store back a $2N$ -bit result to an N -bit register, the system eventually discards the lower N bits. However, in order to maintain higher precision, some techniques are developed to deal with such condition and in general called rounding.

There are three chief rounding schemes: *biased-rounding*, *unbiased-rounding*, and *truncation*. For biased and unbiased rounding, if the lower half to be discarded is larger than a half, the system rounds up, i.e. it adds one to the new LSB, $M[16]$ or $ACC[16]$; when it is smaller than a half, the system rounds down by simply discarding the lower half. The two rounding schemes differ only in the case that the lower half equals the midpoint value. For biased rounding, this value is always rounded up; biased rounding thereby always rounds to the nearest 0 or 1 and is also nicknamed *round-to-nearest*. The result on average is biased to a value slightly larger than a half. Unbiased rounding, also called *round-to-nearest-even*, has a different way to deal with the midpoint value. It rounds the value to the nearest even point, and the rounding direction depends on the LSB of the higher half, $M[15]$ or $ACC[15]$, not always upward. This scheme consequently has no bias if the system process on random data, and is also called *convergent rounding*.

Considering DSP algorithms, we often use unbiased rounding scheme to round fractions; however, some application such as GSM algorithm uses biased rounding [27]. It requires some logic to perform rounding before writing the result to a system register half. If the destination register bit width is long enough, rounding is meaningless.

An easiest way to avoid any rounding logic is to totally get rid of the lower half. This technique is referred to as *truncation* and sometimes called *round-to-zero*.

In contrast with the other two schemes, truncation is biased downward to a smaller average. Fig. 5.11 depicts three rounding schemes.



Biased (Round-to-Nearest)

$$R[3:0] = A[7:4] + A[3];$$

Unbiased (Round-to-Nearest-Even/Convergent)

```

If (A[3:0] != 1000)
  R[3:0] = A[7:4] + A[3];
If (A[3:0] = 1000)
  R[3:0] = A[7:4] + A[4];

```

Truncation (Round-to-Zero)

$$R[3:0] = A[7:4];$$

Fig. 5.11. Three different rounding schemes.

5.3 Reconfigurable Parameters Setup

The proposed design has been detailed in previous sections; this section describes the way to reconfigure the parameters of the proposed design in tabular form. Table 5.5 lists the I/O interface of the proposed design. Table 5.6, identical to Table 3.3, lists again the possible SW combination schemes of the proposed design. Table 5.7 and Table 5.8 give some examples to configure the *kill* and *mode* signals, respectively. Some points or exception to be noticed are list as notes at the bottom of each Table.

Table 5.5. Interface of the proposed design.

MAC	Scalar	16-bit	32-bit	64-bit
MULTIPLICAND	mcand[N-1:0]	mcand[15:0]	mcand[31:0]	mcand[63:0]
MULTIPLIER	mlier[N-1:0]	mlier[15:0]	mlier[31:0]	mlier[63:0]
ACCUMULATOR	accu[2N-1:0]	accu[31:0]	accu[63:0]	accu[127:0]
MODE	mode[1:0]	mode_v0[1:0] mode_v1[1:0]	mode_v0[1:0] mode_v1[1:0] mode_v2[1:0] mode_v3[1:0]	mode_v0[1:0] mode_v1[1:0] mode_v2[1:0] mode_v3[1:0] mode_v4[1:0] mode_v5[1:0] mode_v6[1:0] mode_v7[1:0]
KILL	N/A	kill	kill0 kill1 kill2	kill0 kill1 kill2 kill3 kill4 kill5 kill6
RESULT	m_out[2N-1:0]	m_out[31:0]	m_out[63:0]	m_out[127:0]
CARRY-OUT	cout	cout_v0 cout	cout_v0 cout_v1 cout_v2 cout	cout_v0 cout_v1 cout_v2 cout_v3 cout_v4 cout_v5 cout_v6 cout
<p>Note: N is the bit width of scalar input operands</p> <p>Note: v0, v1, ..., v7 indicate SWs in order; v7 aligns to MSB; v0, LSB</p> <p>Note: For all MODE signal: 1?: mixed-mode; 00: unsigned; 01: signed</p> <p>Note: kill is inserted between SWs; kill2 between v2 and v3, and the like</p>				

Table 5.6. Possible sub-word combinations of the proposed SWP MAC design.

Possible Sub-Word Combinations	
16-bit	(16)
	(8,8)
32-bit	(32)
	(8,8,8,8)
	(8,8,16)
	(16,16)
	(16,8,8)
64-bit	A 64-bit SWP MAC is viewed consisting of two independent 32-bit SWP MACs; it then has $5 \times 5 = 25$ possible combinations

Table 5.7. Configuration example of KILL signal.

KILL	kill6	kill5	kill4	kill3	kill2	kill1	kill0
16-bit SWP MAC							
(16)	N/A	N/A	N/A	N/A	N/A	N/A	0
(8,8)	N/A	N/A	N/A	N/A	N/A	N/A	1
32-bit SWP MAC							
(32)	N/A	N/A	N/A	N/A	0	0	0
(16,16)	N/A	N/A	N/A	N/A	0	1	0
(8,8,8,8)	N/A	N/A	N/A	N/A	1	1	1
(8,8,16)	N/A	N/A	N/A	N/A	1	1	0
64-bit SWP MAC							
(64)	0	0	0	0	0	0	0
(32,32)	0	0	0	1	0	0	0
(16,16,16,16)	0	1	0	1	0	1	0
(8,8,8,8,8,8,8,8)	1	1	1	1	1	1	1
(16,16,32)	0	1	0	1	0	0	0
(8,8,8,8,32)	1	1	1	1	0	0	0
(8,8,16,32)	1	1	0	1	0	0	0
Note: List only some possible conditions							
Note: An illegal input will be redirected to scalar mode by default							

Table 5.8. Configuration example of *MODE* signal.

<i>MODE</i>	<i>SW_7</i>	<i>SW_6</i>	<i>SW_5</i>	<i>SW_4</i>	<i>SW_3</i>	<i>SW_2</i>	<i>SW_1</i>	<i>SW_0</i>
16-bit SWP MAC								
(16)	N/A	N/A	N/A	N/A	N/A	N/A	○	×
(8,8)	N/A	N/A	N/A	N/A	N/A	N/A	○	○
32-bit SWP_MAC								
(32)	N/A	N/A	N/A	N/A	○	×	×	×
(16,16)	N/A	N/A	N/A	N/A	○	×	○	×
(8,8,8,8)	N/A	N/A	N/A	N/A	○	○	○	○
(8,8,16)	N/A	N/A	N/A	N/A	○	○	○	×
64-bit SWP MAC								
(64)	○	×	×	×	×	×	×	×
(32,32)	○	×	×	×	○	×	×	×
(16,16,16,16)	○	×	○	×	○	×	○	×
(8,8,8,8,8,8,8,8)	○	○	○	○	○	○	○	○
(16,16,32)	○	×	○	×	○	×	×	×
(8,8,8,8,32)	○	○	○	○	○	×	×	×
(8,8,16,32)	○	○	○	×	○	×	×	×
○: Configurable ×: Can't configure;should be identical with the nearest ○ on the left side Note: Incorrect assignment of mode may cause a wrong result								

CHAPTER 6

CONCLUSIONS

In this thesis, we present the design methodology of a high-performance reconfigurable modified Booth encoded MAC unit. It is capable of supporting sub-word parallel (SWP) operation which enhances computational throughput of many DSP algorithms especially for multimedia applications. The scalar version of the proposed design comprises a high-speed, area-reduced, and race-free MBE; a speed optimized Wallace PPRT using TDM; and a high speed, area-minimized Fong adder. Using essentially the same hardware, SWP is performed on the scalar MAC by applying some preprocessing to operands associated with a new arrangement of the SWPPA, and with the support of carry-chain blocking when accumulating all partial products. A novel full-adder carry-out masking concept is proposed to build the SWPPRT, facilitating the use of TDM. The SWP version Fong adder inherits its scalar merits and supports identical SW combinations with our requirement. The proposed SWP design innovatively features the flexible sub-word combination and mode assignment scheme with nearly same delay and modest area overhead compared with the proposed scalar design. The proposed designs are fully-synthesizable in a reusable and verifiable design style. Experimental results demonstrate that the proposed scalar and SWP designs, for most cases, outperform the designs of *DesignWare*® IP [38] and of [10] in terms of critical path delay, area cost, and power consumption.

FUTURE WORKS

We are developing a generator to generate the RTL codes of the proposed MAC designs in Verilog HDL format. Testbench for verification, synthesis script, and user's manual will also be generated. All output files depend on the user reconfigurable inputs. We are also analyzing the pros and cons of replacing the scalar MAC units in multiple-MAC DSP processors by a proposed SWP MAC in order to design a high-performance MAC unit.



BIBLIOGRAPHY

- [1] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, pp. 698, pp. 484, pp. 488, John Wiley & Sons, 1999.
- [2] P. Lapsley, J. Bier, A. Shoham and E. Lee, *DSP Processor Fundamentals: Architectures and Features*, p. 9, p. 35, p. 47, Berkeley Design Technology Inc., 1996
- [3] B. Parhami, *Computer Arithmetic Algorithms and Hardware Design*, pp. 204-205, pp. 149-151, pp. 133-134, pp. 98-99, Oxford University Press, New York, 2000.
- [4] O. L. MacSorley, "High-speed arithmetic in binary computers", *Proc. IRE*, vol. 49, pp. 67-91, 1961.
- [5] C. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Trans. on Electronic Computers*, vol.13, pp. 14-17, 1964.
- [6] S. Krithivasan and M. J. Schulte, "Multiplier Architectures for Media Processing," *Proc. 37th Asilomar Conf. Signals, Systems, and Computers*, pp. 2193-2197, Nov. 2003.
- [7] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-Chip Designs*, Kluwer Academic Publishers, third edition, 2002.
- [8] V. G. Oklobdzija, D. Villegier, and S. S. Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach," *IEEE Trans. Computers*, vol. 45, no. 3, pp. 294--305, March 1996.
- [9] W.-C. Yeh and C.-W. Jen, "High-Speed Booth Encoded Parallel Multiplier

- Design," *IEEE Trans. Computers*, vol. 49, no. 7, pp. 692-701, July 2000.
- [10] A. Danysh and D. Tan, "Architecture and Implementation of a Vector/SIMD Multiply-Accumulate Unit," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 284-293, Mar., 2005.
- [11] D. Tan, A. Danysh, M. Liebelt, "Multiple-Precision Fixed-Point Vector Multiply-Accumulator Using Shared Segmentation," *arith*, p. 12, 16th IEEE Symposium on Computer Arithmetic (ARITH-16 '03), 2003.
- [12] G. W. Bewick, "Fast Multiplication: Algorithms and Implementation," PhD dissertation, pp. 14-16, appendix A, pp. 13-14, Stanford University, Department of Electrical Engineering, Feb., 1994.
- [13] A. D. Booth, "A Signed Binary Multiplication Technique," *Quarterly J. Mechanical and Applied Math.*, vol. 4, pp. 236-240, 1951.
- [14] L. Dadda, "Some Schemes for Parallel Multipliers," *Alta Frequenza*, pages 349-356, March 1965.
- [15] M. Santoro, "Design and Clocking of VLSI Multipliers", PhD dissertation, Stanford University, Department of Electrical Engineering, 1989.
- [16] R. Fried, "Minimizing Energy Dissipation in High-Speed Multipliers," *Proc. 1997 Int'l Symp. Low Power Electronics and Design*, pp. 214-219, 1997.
- [17] M. Annaratone and W. Z. Shen, "The Design of an LSI Booth Multiplier," Carnegie Mellon University Thesis report (CS), no. 150, 1984.
- [18] A. A. Farooqui and V. G. Oklobdzija, "General Data-Path Organization of a MAC Unit for VLSI Implementation of DSP Processors," *Proc. 1998 IEEE Int'l Symp. Circuits and Systems*, vol. 2, pp. 260-263, 1998.
- [19] S. Vassiliadis, E.M. Schwarz, and B.M. Sung, "Hard-Wired Multipliers with Encoded Partial Products," *IEEE Trans. Computers*, vol. 40, no. 11, pp. 1181-1197, Nov. 1991.

- [20] P. F. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi, "Optimal circuits for parallel multipliers," *IEEE Transactions on Computers*, vol. 47, no. 3, pp. 273-285, Mar. 1998.
- [21] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, pp241-249, Morgan Kaufman Publishers, Inc., 2nd Edition, 1998.
- [22] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Transactions on Computers*, vol. 31, no. 3 pp.260-264, 1982.
- [23] T. Han, D. A. Carlson, and Steven P. Levitan, "Fast Area Efficient VLSI Adders," *IEEE International Conference on Computer Design*, pages 418-422, October 1987.
- [24] H Ling, "High-Speed Binary Adder," *IBM J. Res. Develop.*, vol. 25, no. 3, pp156-166, May 1981.
- [25] G. Dimitrakopoulos and D. Nikolos, "High-Speed Parallel-Prefix VLSI Ling Adders," *IEEE Trans. Computers*, vol. 54, No.2, Feb. 2005.
- [26] Y. -C. Fong, "A High-Speed Area-Minimized Reconfigurable Adder Design," Master's thesis, National Chiao Tung University, Department of Electronics Engineering, Jul. 2006.
- [27] Analog Devices, *Blackfin® Processor Hardware Reference*, revision 3.0, Sep., 2004. Available from www.analog.com.
- [28] Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide*, revision F, Oct. 2000. Available from www.ti.com.
- [29] C. G. Lee and M. G. Stoodley, "Simple Vector Microprocessors for Multimedia Applications," *Proc. 31st Ann. ACM/IEEE Int'l Symp. Microarchitecture*, pp. 25-36, 1998.
- [30] R. B. Lee, "Multimedia Extensions for General-Purpose Processors," *Proc.*

Signal Processing Systems (SIPS '97), pp. 9-23, Nov. 1997.

- [31] N. Burgess, "PAPA—Packed Arithmetic on a Prefix Adder For Multimedia Applications," *Proc. IEEE Int'l Conf. Application-Specific Systems, Architectures and Processors*, pp. 197-207, July 2002.
- [32] A. A. Farooqui, V. G. Oklobdzija, and F. Chehrazi, "Multiplexer Based Adder for Media Signal Processing," *Proc. 1999 Int'l Symp. VLSI Technology, Systems, and Applications*, pp 100-103, June 1999.
- [33] C. R. Baugh and B. A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Transactions on Computers*, vol. 22, pp. 1045--1047, December 1973.
- [34] M. J. Schulte, L. P. Marquette, S. Krithivasan, E. G. Walters, and J. Glossner, "Combined Multiplication and Sum-of-Squares Units," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 204–214, June, 2003.
- [35] Shankar Krithivasan, Michael J. Schulte, John Glossner, "A Subword-Parallel Multiplication and Sum-of-Squares Unit," *isvlsi*, p. 273, *IEEE Computer Society Annual Symposium on VLSI Emerging Trends in VLSI Systems Design (ISVLSI'04)*, 2004.
- [36] T. K. Callaway and E. E. Swamlander, Jr., "Power-Delay Characteristics of CMOS Multipliers," *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, pp. 26-32, 1997.
- [37] Artisan Components, *UMC 0.18 μ m L180 Process 1.8-Volt Sage-XTM Standard Cell Library Databook*, release 2.0, pp. 32-33, Nov. 2003.
- [38] Synopsys Inc., *DesignWare® Building Block IP Documentation Overview*, Jan. 17, 2005.
- [39] Synopsys Inc., *Design Compiler® User Guide*, version W-2004. 12, Dec.,

2004.

[40] Synopsys Inc., *PrimePower® Manual*, version W-2004. 12, Dec., 2004.

[41] Cadence Design Systems Inc., *Verilog®-XL User Guide*, version 3.4, Jan., 2002.

[42] Novas Software Inc., *nLint® User Guide and Tutorial*, version 2.2, Dec., 2004.

[43] TransEDA Technology Ltd., *Verification Navigator® User Guide*, version 2005.03, Mar., 2005.

[44] Cadence Design Systems Inc., *Encounter™ Conformal® Equivalence Checking User Guide*, version 5.1, June, 2005.

