Specialized Multiplier Circuits

Approved by

Specialized Multiplier Circuits

A Dissertation Presented to the Graduate Faculty of

Lyle School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Doctor of Philosophy

with a

Major in Computer Engineering

by

Jason Moore

(M.S., Computer Science, Southern Methodist University)
(B.S., Computer Systems Engineering, University of Arkansas)

December 20 2014

Acknowledgment

My dissertation would not have been possible without the support, encouragement, and help of many people. To each of you, I owe a deep gratitude.

Dr. Mitch Thornton and Dr. David Matula served as my mentors and advisors during the course of my work. Their guidance was essential.

I thank my grandmother, Iris Butler, who was always there with financial support when money ran short. And I express thanks to my brothers, Rodney, Lynn, and Roger who offered suggestions, counsel, and sounding boards –whether I asked or not.

To my dearest, June, you have been a constant and tireless uplifting force in my life. Your encouragement and belief in me bolsters my confidence. Thank you.

I do not have words to articulate how much my parents, Jerry and Jennifer, have supported and influenced me throughout my life. I thank them for their timeless wisdom –though it was not always followed, it was always appreciated. And without their unconditional love, I could have never made it this far. Thank you, both.

Moore, Jason B.S., University of Arkansas, 2001
M.S., Southern Methodist University, 2004

Specialized Multiplier Circuits

Advisor: Dr. Mitchell A. Thornton and Dr. David W. Matula

Doctor of Philosophy conferred December, 20, 2014

Dissertation completed December, 4, 2014

This document presents an *n*-bit dual recoded squarer and a 2*n*-bit dual recoded radix-4 squarer as well as an *n*-bit and 2*n*-bit integer radix-4 multiplier circuits using dual recoded radix-4 squarers. The radix-4 dual recoding squarer reduces the number of bit product terms employed in the previously known squaring methods obtained by either Booth radix-4 recoded multiplication or by radix 2 squaring. Employing the dual recoded radix-4 procedure for design of a squaring circuit introduces a significant reduction in power and area. Architecturally, radix-4 dual recoded squaring uses only the 1's complement representation which allows for a simpler PPG structure as compared to the 2's complement representation required for Booth radix-4 multiplication. Based on the above squarer, an *n*-bit radix-4 multiplier is designed to reduce power and size compared to using a traditional multiplier circuit while a similarly designed radix-4 2*n*-bit multiplier can be used for a gain in area. Finally, an architecture for a combinational floating point multiplier and squarer is described for the purpose of producing a low power floating point square with small area requirements. In order to take advantage of the squarer power improvements with a minimal increase in area, the multiplier and squarer are combined into one circuit. Shared circuitry among the units provides justification for inclusion of a dedicated squarer since a small amount of additional circuitry is required and the power savings for squaring computations is significant as compared to the use of a general-purpose multiplier to generate a squared value.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Dedication

This dissertation is dedicated in loving memory of Dr. Ronald Wayne Skeith. Dr. Skeith was my undergraduate advisor, mentor and friend. His influence shaped my life and career.

Chapter 1

INTRODUCTION

Squaring is the symmetric case of multiplication where both multiplicand and multiplier are the same operand reducing squaring to a unary operation. This symmetry allows partial product arrays for squaring to be reduced to about half the size and half the depth of comparable multiplication partial product arrays for both radix 2 and recoded radix 4 arrays. There are many applications such as fast exponentiation, Euclidean distance computation, digital signal processing applications such as adaptive filtering, vector quantization, image compression, and pattern recognition, sum of squares statistical computations, and rounding of floating point square root, where incorporating a supplemental squaring circuit may be attractive to avoid the time and power requirements of performing all squaring operations in a large multiplier.

Multiplying is the sum of adding one number which is referred to as the multiplicand, to itself a specific number of times which is the multiplier. The traditional algorithm for doing this called for sequential additions of the multiplicand based on the multiplier as illustrated in Figure 1.1.

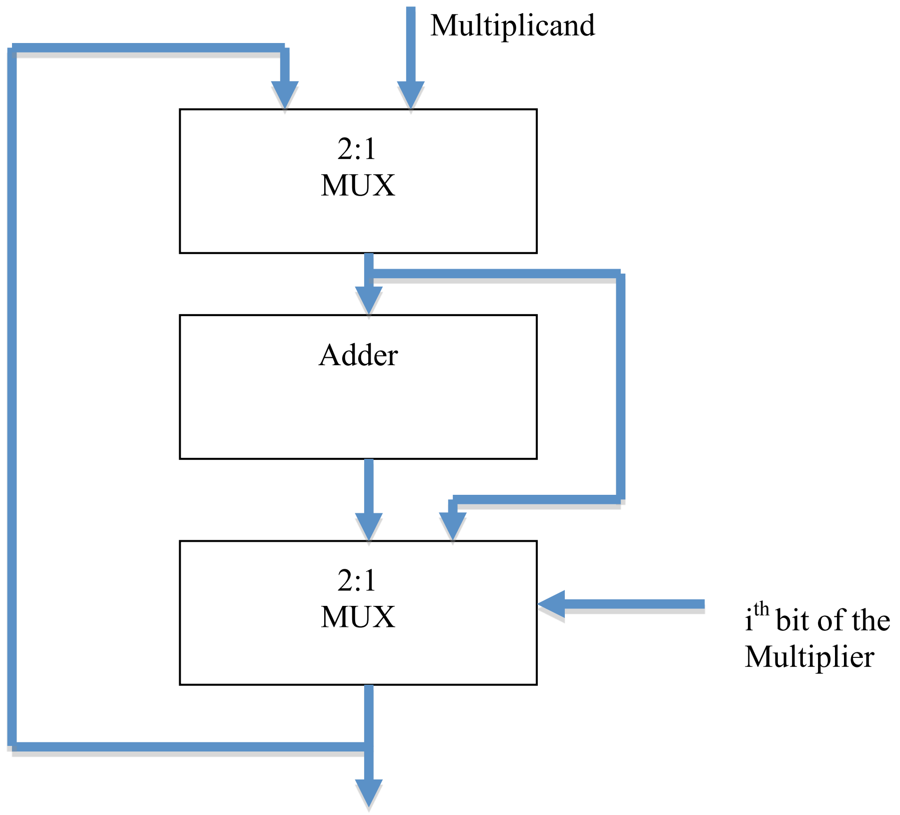

Figure 1.1: Data Path of a Traditional Multiplier Circuit

As well as building general purpose multipliers, left shifters were used to multiply by powers of 2. [It89] These shifters can be thought of as the first special purpose multipliers.

1.1 Current Methods

For binary squaring of the n-bit operand $Q = q_{n-1}q_{n-2}\cdots q_0$, the n^2 bit product terms of the multiplication array may be reduced to $\frac{(n+1)n}{2}$ terms employing and for $i > j$. This reduction in multiplication size for binary squaring has been noted in the literature [EL04, Pa00]. For example, the 5 x 5 bit multiplication partial product array for $Q\ x\ Q$ illustrated in Figure 1.1.1 reduces for such a binary squaring operation to the skewed triangular array of Figure 1.1.2.

				q_4	q_3	q_2	q_1	q_0	
				q_4q_0	q_3q_0	q_2q_0	q_1q_0	q_0q_0	q_0
			q_4q_1	q_3q_1	q_2q_1	q_1q_1	q_0q_1		q_1
		q_4q_2	q_3q_2	q_2q_2	q_1q_2	q_0q_2			q_2
	q_4q_3	q_3q_3	q_2q_3	q_1q_3	q_0q_3				q_3
q_4q_4	q_3q_4	q_2q_4	q_1q_4	q_0q_4					q_4
s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0	

Figure 1.2: Bit Product Array for a 5x5-bit Multiplication Implementation of the Squaring Operation

				q_4	q_3	q_2	q_1	q_0	
			q_4q_0	q_3q_0	q_2q_0	q_1q_0	0	q_0	q_0
		q_4q_1	q_3q_1	q_2q_1	0	q_1			q_1
	q_4q_2	q_3q_2	0	q_2					q_2
q_4q_3	0	q_3							q_3
q_4									q_4
s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0	

Figure 2.2.1.3: Skewed Triangular Bit Product Array for 5-bit Squaring

Note that each row of the triangular array for n-bit squaring may be realized as an "i^{th} multiplicand prefix" that is conditionally selected by the squaring operation's "multiplier" bit q_i. The rows in Figure 1.1.2 thus form "partial squares" analogous to the partial products of multiplication that may be formalized as follows.

Observation 1: Let $Q = q_{n-1}q_{n-2}\cdots q_0$ with $Q_i = \left\lfloor \frac{Q}{2^i} \right\rfloor$ for $i = 1,2,\cdots,n$. Then

$$Q^2 = \sum(4Q_i + q_i)q_i 4^i \qquad (2.1)$$

Proof: With $Q_i = 2Q_1 + q_0$, then $Q^2 = 4Q_1^2 + (4Q_1 + q_0)q_0$. Since $Q = 2Q_{i+1} + q_i$, $Q_i^2 = 4Q_{i+1}^2 + (4Q_{i+1} + q_i)q_I$ and iterative substitution yields equation (2.1).

□

Definition 2: Let $Q = q_{n-1}q_{n-2} \cdots q_0$ with $Q_i = \frac{Q}{2^i} = q_{n-1}q_{n-2} \cdots q_i$ for $0 \leq i \leq n-1$. Then the *ith partial square* $(4Q_{i+1} + q_i)q_i$ is the product of the ith multiplicand factor $(4Q_{i+1} + q_i)$ and the ith select bit q_i.

There are three distinct results that may be desired for the square of an *n*-bit integer.

- *2n*-bit integer square: this is the double precision exact 2*n*-bit square Q^2 in a 2*n*-bit word. This is the result provided by summing the bit product terms illustrated in Figure 2.2. This full result is useful in implementations of multiple precision arithmetic.

- *n*-bit integer square: The result is the square module the word size that is Q^2 mod 2^n for an *n*-bit word. This is the typical result desired for implementing an *n*-bit integer arithmetic expression.
- *n*-bit floating point square: For a floating point number with a normalized significand of *n*-bits, this is the normalized and rounded *n*-bit result. The *n*-bit floating point square is determined from the 2*n*-bit integer square by appropriate rounding to obtain the normalized *n*-bit result.

The integer result is effectively the n-bit low order part of the $2n$-bit square and the floating point result is effectively the high order part, possibly overlapping the low order part since the square of a normalized n-bit significand may have length $2n - 1$ bits. Let us first consider the integer word size example of the 16-bit integer square, contrasting the binary multiplication array with the binary squaring array.

Example 1: Consider evaluation of the square of the 16-bit argument $Q = 50{,}571 = 1100\ 0101\ 1000\ 1011_2$. The full 32-bit square is $(50{,}571)^2 = 2{,}557{,}426{,}041$, which in binary is $Q^2 = 1001\ 1000\ 0110\ 1111\ 0011\ 1001\ 0111\ 1001_2$. Table 1.1.1 illustrates the 16×16 bit multiplication array for determining the 16-bit integer result $Q^2 = 0011\ 1001\ 0111\ 1001_2 \bmod 2^{16}$. This example array corresponds to the low order triangular portion of Figure 1.1.1.

Table 1.1.1: 136 Entries

1100	0101	1000	101**1**
1000	1011	0001	0**1**1
0000	0000	000**0**	00
0010	1100	0**1**01	1
0000	000**0**	0000	
0000	0**0**00	000	
000**0**	0000	00	
1**1**00	0101	1	
1000	1011		
0000	000		
0010	11		
0000	0		
0000			
0000			
000			
11			
1			
0011	1001	0111	1001

Determination of the *n*-bit integer word square for *n* even by binary squaring requires summation of just $\frac{n}{2}$ partial squares. Table 1.1.2 illustrates this summation for our 16-bit example, where there are just 72 entries in the 16x8-bit triangular array compared to 136 entries in the 16x16-bit triangular array of Table 1.1.1. Table 1.1.2 follows the pattern illustrated in the low order triangular portion of Figure 1.1.2, where the small zeroes here denote forced zeroes.

Table 1.1.2: 72 Entries

Binary Squaring – 16-bit Square (Integer word result)

(1100 0101 1000 1011)^2	
[1000 1011]	(selector bits)
1000 1011 0001 01o1	1
0001 0110 0010 o1	1
0000 0000 00o0	0
0101 1000 o1	1
0000 00o0	0
0000 o0	0
00o0	0
o1	1
0011 1001 0111 1001	Square

For computing the full 2*n*-bit integer square $Q^2 = (q_{n-1}q_{n-2}\cdots q_0)^2$, the skewed triangular array of partial squares $(4Q_{i+1} + q_i)q_i$ illustrated in Figure 4.1.2 may be consolidated to form fewer rows by moving up the diagonals of the left half of the $2n{\times}n$ bit array. Incorporating the half-adder relation $q_{i+1}q_i + q_{i+1} = 2q_{i+1}q_i + q_{i+1}\overline{q_i}$ for $0 \leq n-2$, we can obtain a $2n{\times}\left\lceil\frac{n}{2}\right\rceil$ nearly symmetric triangular array of $\binom{n+1}{2}$ bit product terms [EL04, Pa00]. Figure 1.1.1 illustrates the consolidated triangular arrays for the calculations 5- and 6-bit squares. For partial square design of this consolidated $\left\lceil\frac{n}{2}\right\rceil$ row array note that the *n*-bit right halves of the *2n*-bit rows may be formed by using bits

q_i for $i = 0,1,\cdots,\left\lceil\frac{n}{2}\right\rceil - 1$ as select bits for n-bit partial squares and the n-bit left halves of the rows may be similarly formed by using $q_{\left\lfloor\frac{n}{2}\right\rfloor}$ through q_{n-1} as select bits for appropriately modified n-bit partial squares.

						q_4	q_3	q_2	q_1	q_0	
q_4	q_4q_3	$q_4\overline{q_3}$	q_4q_2	q_4q_1	q_4q_0	q_3q_0	q_2q_0	$q_1\overline{q_0}$	0	q_0	q_0
q_3			q_3q_2	$q_3\overline{q_2}$	q_3q_1	$q_2\overline{q_1}$	q_1q_0				q_1
q_2					q_2q_1						q_2

							q_5	q_4	q_3	q_2	q_1	q_0	
q_5	q_5q_4	$q_5\overline{q_4}$	q_5q_3	q_5q_2	q_5q_1	q_5q_0	q_4q_0	q_3q_0	q_2q_0	$q_1\overline{q_0}$	0	q_0	q_0
q_4			q_4q_3	$q_4\overline{q_3}$	q_4q_2	q_4q_1	q_3q_1	$q_2\overline{q_1}$	q_1q_0				q_1
q_3					q_3q_2	q_3q_2	q_2q_1						q_2

Figure 2.2.1.1.1: Partial Product Consolidation for 5- and 6-bit Squaring

The binary squaring operation thus reduces the area measured in bit product terms by a factor of two compared to multiplication for generation of either the n-bit or $2n$-bit square, and computation of the n-bit floating point square is similarly simplified. The reduced size and depth of the array of terms to be accumulated for binary squaring is slightly better than the reduced size obtained by Booth radix-4 recoding, and the bit product terms are specified without the conditional shifting and/or complementation required to calculate the partial products of Booth radix-4 recoding.

1.2 Testing

All of the circuits presented are compared to a carry-save multiplier. When testing the squaring circuits, both the multiplier and square circuits tested with values 0 to 2^{16}

and 2^{16} random values for bit widths greater than 16. When testing the multiplication circuits, the circuits were tested with 2^{16} random combinations of inputs.

1.2.1 Test Bench Generation

In order to produce the Verilog files with test inputs, I wrote a Java application that generates test files. The call to the application is as follows:

```
VerilogTest fileName bitWidth <option>
```

The required inputs are file name and bit width. The bit width can be 16, 32, or 64 depending on the selected option. The option input is optional however if no option is provided, the program defaults to an unsigned square. The other options are multiply, signed, and floating. If the selected option is the default and the bit width is 16, all possible values are used. For all other options except for floating, 2^{16} random values between 0 and $2^{bitWidth}$ are used. The floating point options are slightly different in order to test all possible exponents. The significands are random numbers between 0 and 2^{23} for single precision and 0 and 2^{52} for double while the exponents are cycled through. The complete implementation for automatic test set generation can be found in Appendix A.

1.2.2 Carry-Save Array Multiplier

An array multiplier is a full-tree multiplier with a one-sided reduction tree in which the final addition performed by a carry-save adder. A carry save multiplier is a good choice as a comparison circuit because much like the dual recoded radix-4 square circuit, it produces partial products and then adds them together. The multiplier is comprised of 2-input AND gates, an array of carry-save adders, and a ripple-carry array. The 2-input AND gates are used to produce the partial products while the carry-save adders are used to add the partial products together and the ripple-carry adder is used for the final addition.

Chapter 2

SQUARING

2.1. Booth Recoding

The radix 4 recoding procedure utilizes Booth recoding. Let $P = d_{\lfloor \frac{n}{2} \rfloor} d_{\lfloor \frac{n}{2} \rfloor - 1} \cdots d_0$ with $d \in \{-2, -1, 0, 1, 2\}$ be the Booth recoded radix 4 representation of $Q = q_{n-1} q_{n-2} \cdots q_0$ [1, 4]. It is important to recall how Booth radix-4 digit d_i of P is determined by the three bits $q_{2i+1} q_{2i} q_{2i-1}$ of Q as can be seen from Table 2.1.1. Bit q_{-1} is considered a 0.

Table 2.1.1: Radix-4 Booth Recoding [1, 4]

Binary	Booth Digit
100	$\bar{2}$
110, 101	$\bar{1}$
000, 111	0
001, 010	1
011	2

The use of Booth radix-4 recoding for 16-bit integer multiplication for *Q x P* requires 88 entries and 9 rows as illustrated in Table 2.1.3 for the *n*-bit product. This is a considerable reduction of the 136 entries and 16 rows for the radix-2 integer partial product array but provides no additional benefit for the squaring operation. A radix-2 squaring circuit was described in [3] resulting in 72 entries as illustrated in Table 2.1.4.

Table 2.1.2: Shows how $Q = 1100\ 0101\ 1000\ 1011_2$ is recoded to be represented by $P = (1\bar{1}012\bar{2}1\bar{1}\bar{1})_4$

$Q = 1100\ 0101\ 1000\ 1011_2$	
11	$\bar{1}$
101	$\bar{1}$
001	1
100	$\bar{2}$
011	2
010	1
000	0
110	$\bar{1}$
001	1
$P = (1\bar{1}012\bar{2}1\bar{1}\bar{1})_4$	

Table 2.1.3: Booth Recoded Radix 4 Multiplication

	(selector digits)
0011 1010 0111 0100	$\bar{1}$
1110 1001 1101 00	$\bar{1}$
0101 1000 1011	1
0011 1010 00	$\bar{2}$
0001 0110	2
0010 11	1
0000	0
00	$\bar{1}$
0100 0000 1000 0101	2's comp bits
0011 1001 0111 1001	Square

Table 2.1.4: Booth Recoded Radix-2 Squaring

$(1100\ 0101\ 1000\ 1011)^2$ [1000 1011]	(selector bits)
1000 1011 0001 01o1	1
0001 0110 0010 o1	1
0000 0000 00o0	0
0101 1000 o1	1
0000 00o0	0
0000 o0	0
00o0	0
o1	1
0011 1001 0111 1001	Square

2.2. Consolidating Partial Squares: The 2n-bit Dual Recoded Radix 4 Square

Three changes to the squaring circuit need to be made in order to support finding the 2*n*-bit square instead of the *n*-bit square.

1. The input must be padded with one zero. (sign bit)
2. The sign extension must be calculated.
3. The circuit requires $\lceil \frac{n}{2} \rceil$ PSG's instead of $\lceil \frac{n}{4} \rceil$ PSG's.

Figure 2.2.1 shows change 1 being made in the data path of the PSG circuit. In addition to padding the input with a leading zero, we must now account for the sign of each Booth recoded digit. We have chosen to cumulate all of these and add them to the total together. The sign extension for the 2's complement of this (n - 3) bit array is $q_{n-3}0q_{n-5}\cdots 0q_1$.

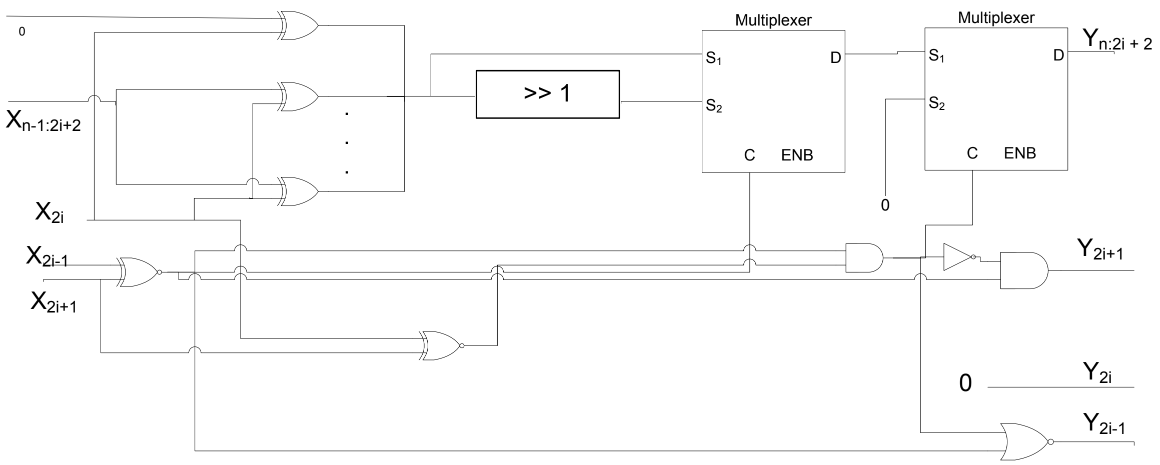

Figure 2.2.1: PSG Data Path for 2n-bit Recoded Radix-4 Squaring Circuit

Table 2.2.1: Radix-4 Squaring (Recoded) 2n-bit square

1100 0101 1000 1011 $(d_7d_6d_5d_4d_3d_2d_1d_0)_4 = (\bar{1}012\bar{2}1\bar{1}\bar{1})_4$	Selection Digits
10 0111 0100 1110 10o1 1001 1101 0011 10o1 01 1000 1011 00o1 0011 1010 01o0 11 0001 01o0 0110 00o1 00 00o0 10o1	$\bar{1}$ $\bar{1}$ 1 $\bar{2}$ 2 1 0 $\bar{1}$
-(o0o0 o0o1 o0o1 o1)	2's comp bits (negative)
1001 1000 0110 1111 0011 1001 0111 1001 (9 8 6 15 3 9 7 9)$_{16}$	Square 2,557,411,328

As can be observed from Table 2.2.1, the output from the first PSG and sign extension bit array can be consolidated into one 2*n*-bit array shown in Figure 2.2.2. Further investigation of Figure 2.2.2 shows that other rows can be consolidated as shown in Figure 2.2.3 for 12-bits.

			$q_{11}q_{10}q_9q_8$	$q_7q_6q_5q_4$	$q_3q_2q_1q_0$	
$-(q_9\ 0q_7$	$0q_5\ 0q_3$	$0q_1)$ **	** **	** **	** 0*	d_0
		** **	** **	** 0*		d_1
	**	** **	** 0*			d_2
	** **	** 0*				d_3
**	** 0*					d_4
** 0*						d_5

Figure 2.2.1: Partial Product Array for 12-bit Recoded Radix-4 Squaring

The $\left\lceil \frac{n}{2} \right\rceil$ radix 4 partial squares yielding the 2*n*-bit square Q^2 according to Theorem 6 may be consolidated as illustrated in Figure 2.2.2. Let us consider the case $n = 4k$. The $2k$ partial squares selected by digits d_i and d_{k+i} may be consolidated into a single 2*n* bit row for $i = 1, 2, \cdots, k$, yielding $k + 1$ rows to be accumulated. This consolidation allows each partial square to be independently shifted and/or complimented before being loaded into the appropriate portion of a row of the consolidated array. The negative leading portion of row zero can be handled by a 2's complement, yielding an extra unit to be added in column $n + 2$, as shown in Figure 2.2.4. Note if $|d_k| \leq 1$, the unit bit may be inserted in column $n + 2$ of the row for d_k. If $|d_k| = 2$, then $d_k^2 = 4$, and the extra unit may be absorbed by inserting 1000 as the four low order bits of the row for d_k.

				$q_{11}q_{10}q_9q_8$	$q_7q_6q_5q_4$	$q_3q_2q_1q_0$	
	$1\bar{q}_9\ 1\bar{q}_7$	$1\bar{q}_5\ 1\bar{q}_3$	$1\bar{q}_1{*}{*}$	** **	** **	** 0*	d_0
d_4	**	** 0*	** **	** **	** 0*		d_1
d_5	** 0*	00 **	** **	** 0*			d_2
		** **	** 0*				d_3
			1				

Figure 2.2.2: Consolidated Partial Product Array for 12-bit Recoded Radix-4 Squaring

Observation 7: For n = $4k$ with $k \leq 1$, the $2n$-bit square of $Q = q_{n-1}q_{n-2}\cdots q_0$ may be obtained as the sum of a $2n \times \left(\frac{n}{4}+1\right)$ bit array of partial squares obtained by radix 4 recoded selection.

High Order Part Selection Digits		Low Order Part Selection Digits
2's comp	1111 1110 1110 1010 0111 0100 1110 10o1	$\bar{1}$
1	0110 00o1|1001 1101 0011 10o1	$\bar{1}$
0	00 00o0|oo01 1000 1011 00o1	1
$\bar{1}$	10o1|oooo 0011 1010 01o0	$\bar{2}$
	11 0001 10o0	2 (+ carry)
	1001 1000 0110 1111 0011 1001 0111 1001 $(9\ \ 8\ \ 6\ \ 15\ \ 3\ \ 9\ \ 7\ \ 9)_{16}$	Square

Figure 2.2.3: Consolidated Radix 4 Recoded 32-bit Squarer for the Example Case
$Q^2 = (1100010110001011)^2$

However, we would like to for the additional unit for the 2's complement to be in a position where there exists a forced zero in one of the rows. By changing the sign extension from $(1\bar{q}_{n-3}1\bar{q}_{n-5}\ldots1\bar{q}_1)$ to $(1\bar{q}_{n-3}1\bar{q}_{n-5}\ldots1\beta\alpha q_1q_1)$ where $\beta = q_3 \text{ NOR } q_1$ and $\alpha = q_1 \oplus q_3$ pushes a forced 1 into position n + 6 as shown in Figure 2.2.5.

High Order Part Selection Digits		Low Order Part Selection Digits
2's comp	$1\overline{q}_{11}1\overline{q}_9 1\overline{q}_7 1\overline{q}_5 1\overline{q}_3 \beta\alpha q_1 q_1$ ** **** **** **** **o*	d_0
d_5	**** **1*| ** ** **** **** **o*	d_1
d_6	** **o*| oo** ** ** **** **o*	d_2
d_7	**o* |oooo ** ** ** ** **o*	d_3
	** ** ** ** o*	d_4

Figure 2.2.4: Consolidated Radix 4 Recoded 32-bit Square

2.2.1. Implementation

Using the algorithm described above the complete data path for the circuit is shown in Figure 2.2.1.2. The first step is to send the bits into the sign extension, shown in Figure 2.2.1.2, and all of the PSG's in parallel. Next the outputs from these modules are positioned and added in parallel in order to reduce the number of additions. The additions continue until we are left with on 2-n output.

From the description of the sign extension earlier in the chapter, it seems like a simple circuit to implement. However, it does have one complexity, which takes a while to trouble shoot if not found during the design stage. The cause of the issue is that within Booth recoded numbering system negative zero which is represented as 111 is a valid number. Within the squaring algorithm, we assume that 0 is positive. Therefore, we cannot simply negate the sign bit of each Booth recoded digit. The sign extension now characterized as $1Z_{n-3}1Z_{n-1}\cdots 1Z_5\beta\alpha q_1 q_1$ where$Z_x = \overline{q}_x +$

$q_x q_{x-1} q_{x-2}$, $\beta = q_3$ NOR q_1 and $\alpha = (q_1 \oplus q_3) + q_1 q_2 q_3$. Using the distributive law the equation for Z_x can be simplified has follows:

$$Z_x = \overline{q}_x + q_x q_{x-1} q_{x-2}$$

$$Z_x = \overline{q}_x q_x + \overline{q}_x q_{x-1} + \overline{q}_x q_{x-2}$$

$$Z_x = \overline{q}_x q_{x-1} + \overline{q}_x q_{x-2}$$

$$Z_x = \overline{q}_x + q_{x-1} q_{x-2}.$$

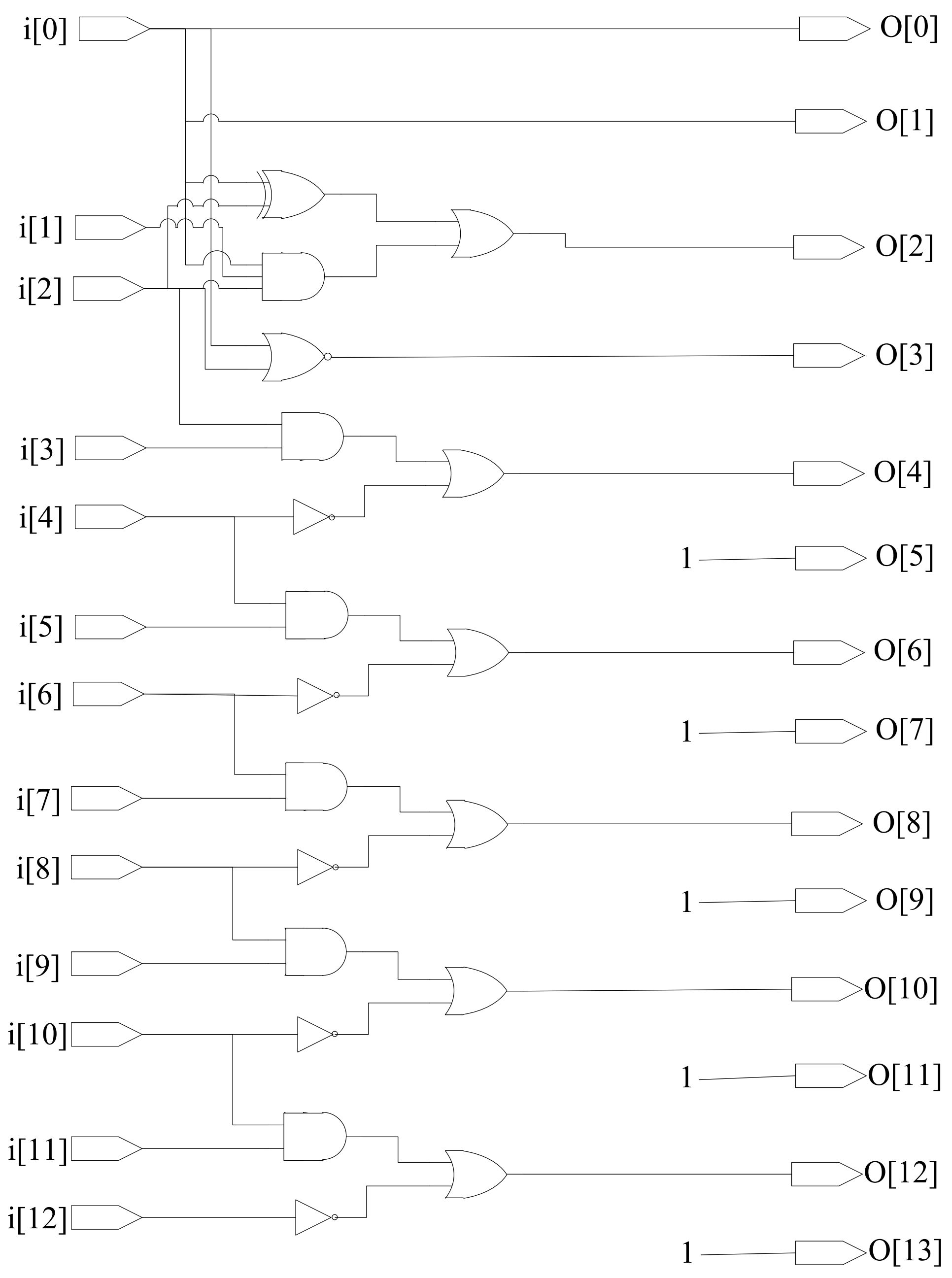

Figure 2.2.1.1: Sign Extension Circuit

Figure 2.2.1.2 illustrates how to create a 16-bit input, 32-bit output full dual recoded radix-4 squaring circuit. The complete circuit consists of $\frac{n}{2}$ partial square generators (PSGs) from Figure 2.2.1, a sign extension circuit from Figure 2.2.1.1, and $\frac{n}{4}$ adders. The PSGs and sign extension circuits are aligned such that outputs from two of these circuits can be concatenated into one input of an adder. For example, the PSG with inputs of n bits is aligned with the sign extension circuit such that the 4 least significant bits of the PSG output is the 4 least significant bits of the square, and bits n to 4 are bits are the first n-4 bits of input 1 of the adder. While the n-2 bit output of sign extension circuit, occupies the final n-2 bits of input 1. When the output bits of two PSGs do not line up so nicely, zeroes are placed between the two outputs to fill in the gaps. This method allows us to avoid combining the outputs by shifting and adding. The adders are arranged such that each level of adders provides the next 4^lbits in the total circuits output where l represents the level of adders.

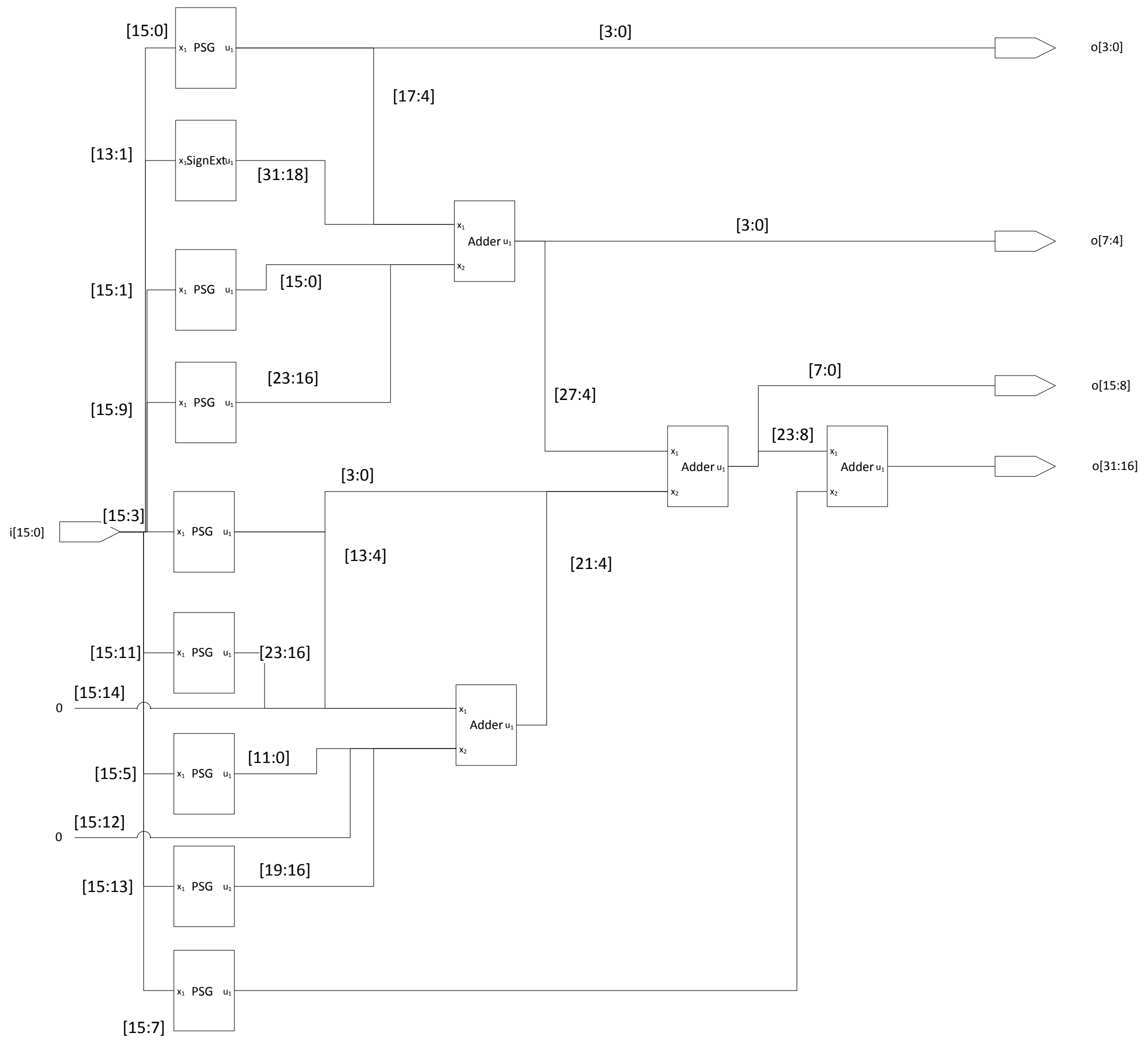

Figure 2.2.1.2: Full Square Circuit

2.2.2. Results

The radix-4 dual recoded squaring circuit and a general purpose multiplier were both implemented in Verilog and mapped to OSU standard cell library [5]. Both circuits were constrained to run with-in a 50ns clock-edge and were implemented for 16, 32, and 64 bit-widths. The charts in Figures 2.2.2.1-2.2.2.3 show a gain in power, leakage power, and area for our customized squaring circuit compared to a multiplier circuit.

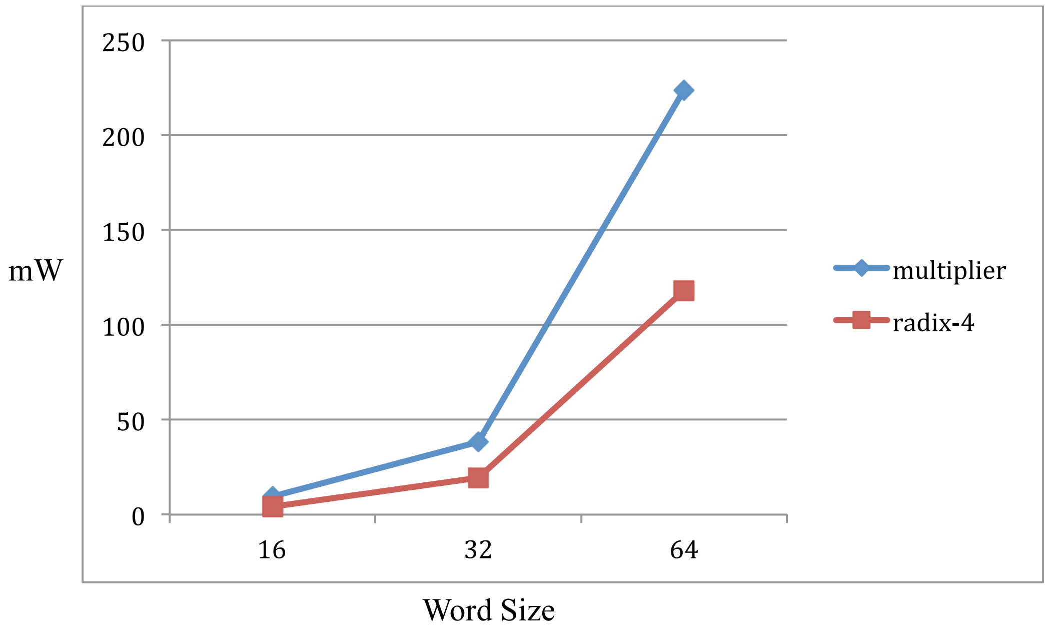

Figure 2.2.2.1: Power (mW) vs Word Size Chart

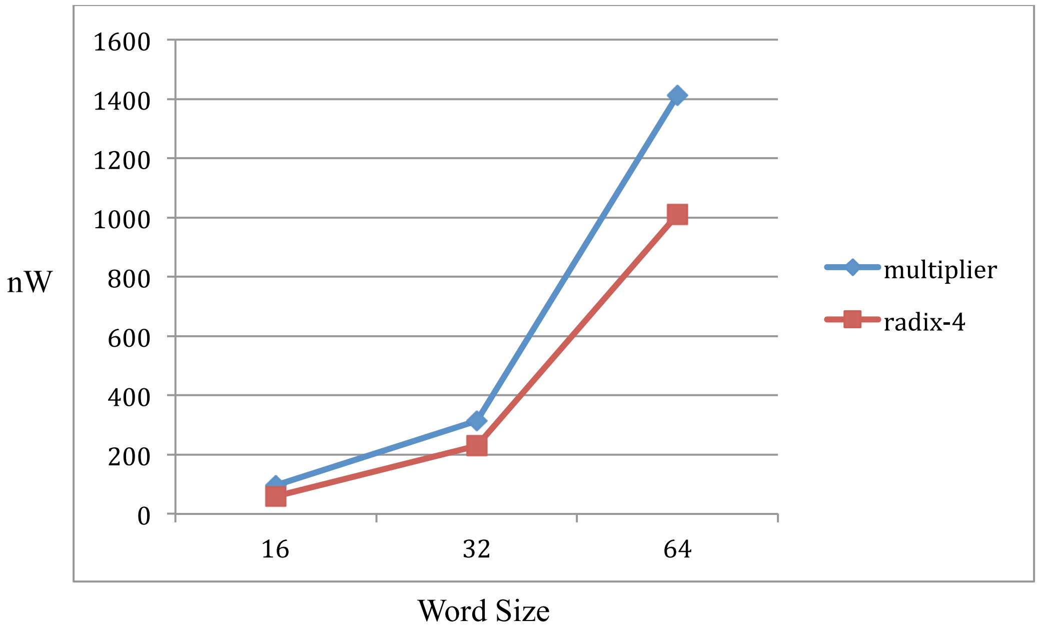

Figure 2.2.2.2: Leakage Power (nW) vs Word Size Chart

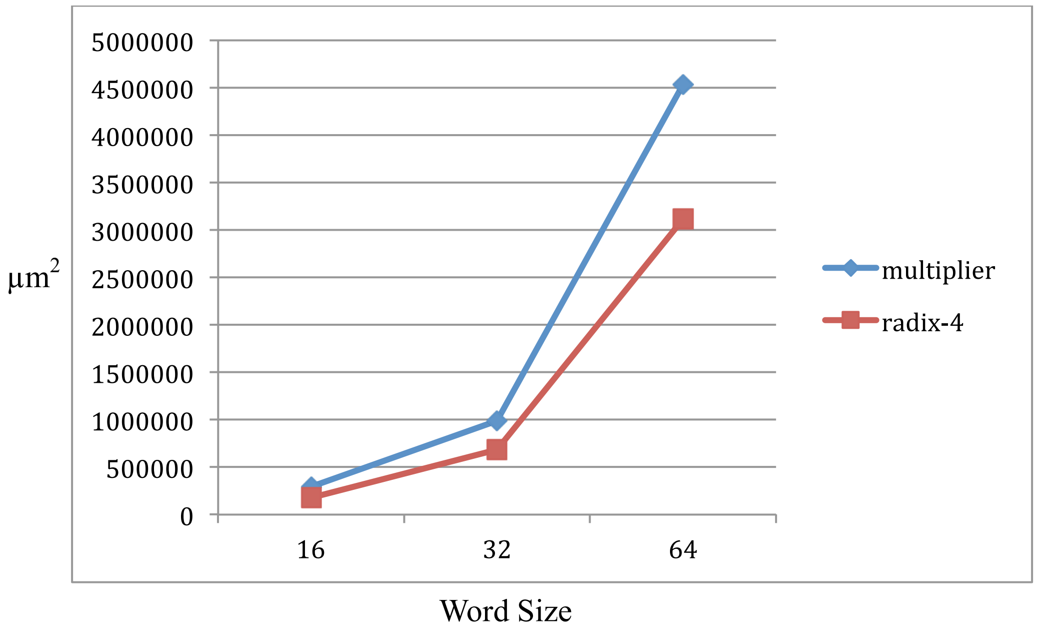

Figure 2.2.2.3: Area vs Word Size Chart

2.3. Dual Recoded Radix-4 Truncated Square

The radix 4 recoding procedure utilizes Booth recoding. Let $P = d_{\lfloor\frac{n}{2}\rfloor} d_{\lfloor\frac{n}{2}\rfloor-1} \cdots d_0$ with $d \in \{-2, -1, 0, 1, 2\}$ be the Booth recoded radix 4 representation of $Q = q_{n-1}q_{n-2} \cdots q_0$ [1, 4]. It is important to recall how Booth radix-4 digit d_i of P is determined by the three bits $q_{2i+1}q_{2i}q_{2i-1}$ of Q as can be seen from Table 2.3.1. Bit q_{-1} is considered a 0.

Table 2.3.1: Radix-4 Booth Recoding [1, 4]

Binary	Booth Digit
100	$\bar{1}$
110, 101	$\bar{1}$
000, 111	0
001, 010	1
011	2

Table 2.3.2; Shows how $Q = 1100\ 0101\ 1000\ 1011_2$ is recoded to be represented by $P = (1\bar{1}0122\bar{2}1\bar{1}\bar{1})_4$

$\boldsymbol{Q = 1100\ 0101\ 1000\ 1011_2}$	
11	$\bar{1}$
101	$\bar{1}$
001	1
100	$\bar{2}$
011	2
010	1
000	0
110	$\bar{1}$
001	1
$\boldsymbol{P = (1\bar{1}0122\bar{2}1\bar{1}\bar{1})_4}$	

The use of Booth radix-4 recoding for 16-bit integer multiplication for $Q \times P$ requires 88 entries and 9 rows as illustrated in Table 2.3.3 for the *n*-bit product. This is a considerable reduction of the 136 entries and 16 rows for the radix-2 integer partial product array but provides no additional benefit for the squaring operation. A radix-2 squaring circuit was described in [3] resulting in 72 entries as illustrated in Table 2.3.4.

Table 2.3.3: Booth Recoded Radix 4 Multiplication

	(selector digits)
0011 1010 0111 0100	$\bar{1}$
1110 1001 1101 00	$\bar{1}$
0101 1000 1011	1
0011 1010 00	$\bar{2}$
0001 0110	2
0010 11	1
0000	0
00	$\bar{1}$
0100 0000 1000 0101	2's comp bits
0011 1001 0111 1001	Square

Table 2.3.4: Booth Recoded Radix-2 Squaring

$(1100\ 0101\ 1000\ 1011)^2$ [1000 1011]	(selector bits)
1000 1011 0001 01o1	1
0001 0110 0010 o1	1
0000 0000 00o0	0
0101 1000 o1	1
0000 00o0	0
0000 o0	0
00o0	0
o1	1
0011 1001 0111 1001	Square

2.3.1. Algorithm

The radix-4 dual recoded squaring algorithm determines the Booth digits in a right-to-left manner i.e. starting from the least significant bit and moving toward the most significant bit. Let P_i be the integer formed by shifting the radix 4 digit string right i places deleting the low order i digits obtaining

for $i = 0,1,\cdots,\left\lfloor\frac{n}{2}\right\rfloor$, with and . Since then and

for $i = 0,1,\cdots,\left\lfloor\frac{n}{2}\right\rfloor$, we obtain the following.

Observation 1:

$$Q^2 = \sum_{i=0}^{\left\lfloor\frac{n}{2}\right\rfloor-1}\left(8\left(Q_{2i+2} + q_{2i+1}\right)d_i + d_i^2\right)16^i$$

Definition 1: Let with for and let be the i^{th} digit of the Booth recoded radix 4 representation . Then the i^{th} radix-4 *partial square* is where is the i^{th} multiplicand factor for radix-4 dual recoded squaring and d_i is the i^{th} *recoded radix 4 select digit.*

Recall that is effectively the sign bit of the recoded digit d_i, so we obtain the partial square identity

.

It is important to observe that the 2's complement of reduces to the sign extended 1's complement of Q_{2i+2}, as formally summarized in the following.

Theorem 1: Let be the radix-4 dual recoded representation of . Let be the conditionally 1's complemented sign extended leading bits of $Q_{2i+2} = \left\lfloor \frac{Q}{2^i} \right\rfloor$ given for $i = 0, 1, \cdots, \left\lfloor \frac{n}{2} \right\rfloor$ by

Then

$$Q^2 = \sum_{0}^{\left\lfloor \frac{n}{2} \right\rfloor} \left(8Q^*_{2i+2}\left|d_i\right| + d_i^2\right)16^i .$$

From Theorem 1 we observe that the PPG's for recoded radix 4 squaring are simpler than for a Booth recoded radix-4 multiplier since the complements are simply 1's complements. This means that for the *n*-bit integer radix-4 square: (1) we need to employ only the $\left\lceil \frac{n}{4} \right\rceil$ low order half of the Booth-4 select digits in forming the radix-4 dual recoded square array, (2) No complement bit row is needed as in Booth-radix 4 multiplication, and (3) no sign extension of the partial squares are needed for the *n*-bit square.

Table 2.3.5 illustrates the radix-4 dual recoded array for computing . There are only 40 entries in Table 2.3.1.1 compared to the 72 entries for radix 2 illustrated in Table 1.1.4 and compared to the 88 entries for Booth recoded radix 4 multiplication illustrated in Table 1.1.3.

Table 2.3.5: Radix-4 Dual Recoded Squaring

1100 0101 |10|00| 10|11 [$\bar{1}$ 1 $\bar{1}$ $\bar{1}$]	Booth-4 select digits
0111 0100 1110 10o1	$\bar{1}$
1101 0011 10o1	$\bar{1}$
1011 00o1	1
01o0	$\bar{2}$
0011 1001 01 11 10 01	Square

2.3.2. Implementation

Based on the properties of Theorem 1, an efficient Partial Square Generator (PSG) was designed. Figure 2.3.2.1 is a Synopsys screen shot of an 8-bit PSG. The dataflow through the circuit is as follows:

1. All the bits of the input, X, except for the 3 least significant bits are conditionally negated if x_2 is 1,
2. The output from step 1 is left shifted by 1 if x_2 is equal to x_0,
3. If the three least significant bits of X are 0, $Y \equiv 0$ else all but the 3 least significant bits of Y are the output from step 2 and

$$y_2 = \overline{\overline{(x_0 \oplus x_1)}\,\overline{(x_0 \oplus x_2)}(x_0 \oplus x_1)}$$
$$y_1 = 0$$
$$y_0 = \overline{\overline{(x_0 \oplus x_1)}\,\overline{(x_0 \oplus x_2)} + \overline{(x_0 \oplus x_1)}}$$

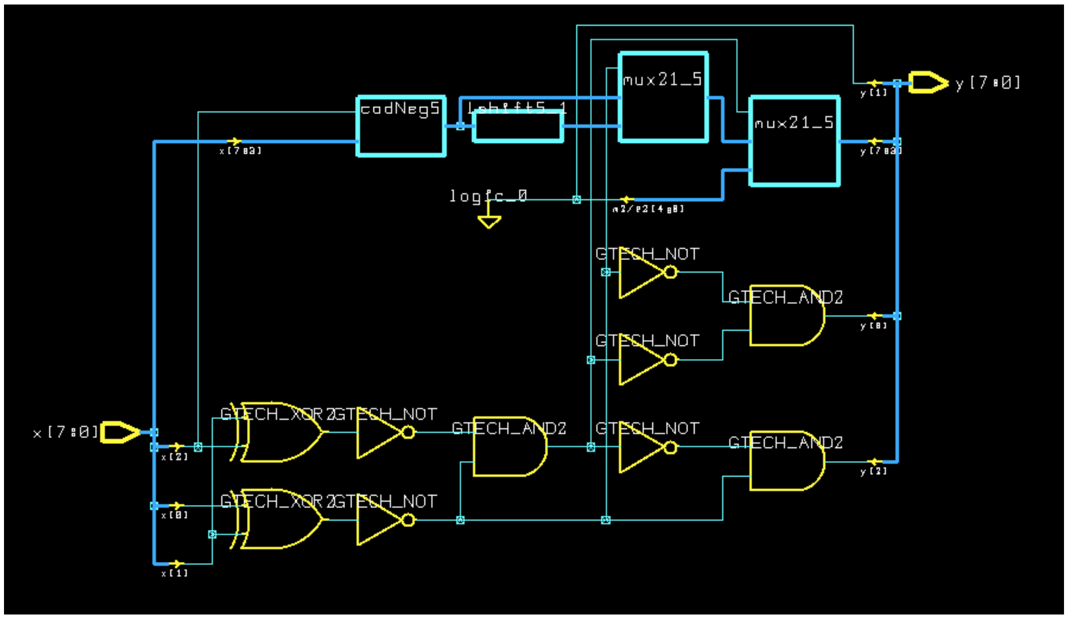

Figure 2.3.2.1: A Synopsys Screen Shot of an 8-bit PSG

Using the PSG design above, the architecture of the squaring unit can be viewed in at least the two following methods depicted in figures 2.3.2.2 and 2.3.2.3. Figure 2.3.2.2 shows the architecture view of building the circuit independent of smaller versions of the circuit. The architecture in Figure 2.3.2.2 is constructed from the following components: $\left\lceil \frac{n}{2} \right\rceil$ PSG's of bit sizes n, n-2, …, 2, and an adder tree of height of $\log_x n$ where x is the bit size of the adders used in the adder tree.

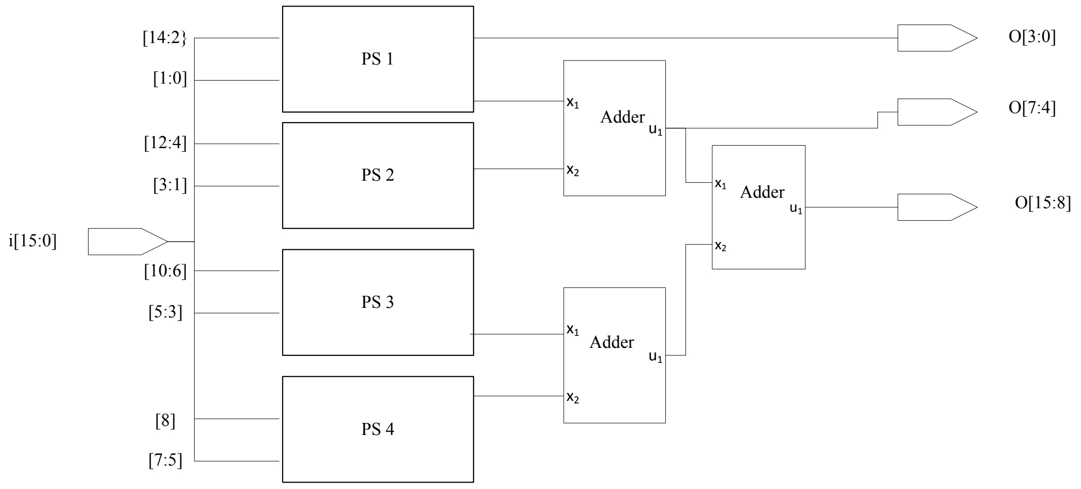

Figure 2.3.2.2: Data Path of 16-bit Input/16-bit Output Radix-4 Dual Recoded Squaring Circuit

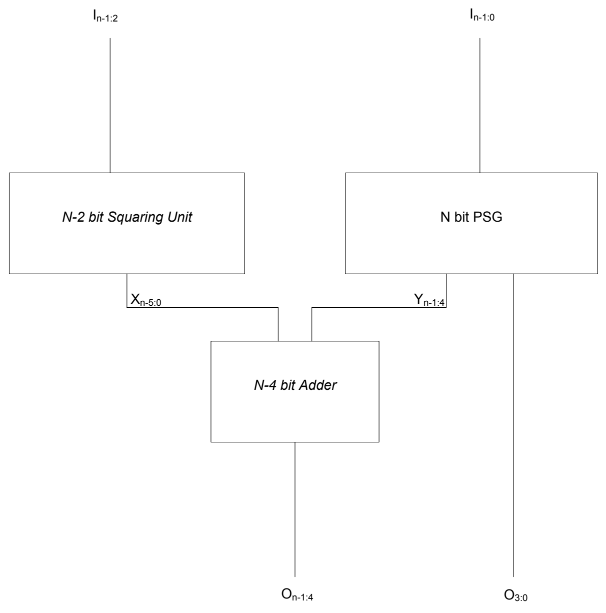

Figure 2.3.2.3: Recursive Data Path for an n-bit Recoded Radix-4 Squaring Circuit

2.3.3. Results

The radix-4 dual recoded squaring circuit and a general purpose multiplier were both implemented in Verilog and mapped to OSU standard cell library [5]. Both circuits were constrained to run with-in a 50ns clock-edge and were implemented for 16, 32, and 64 bit-widths. The charts in Figures 2.3.3.1-2.3.3.3 show a substantial gain in power, leakage power, and area for our customized squaring circuit compared to a multiplier circuit.

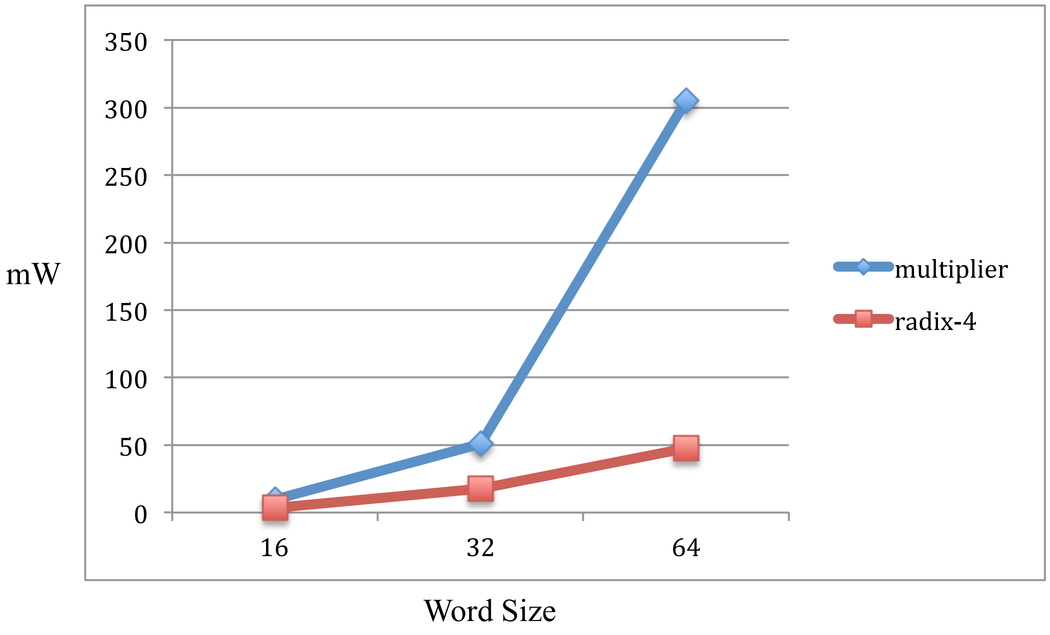

Figure 2.3.3.1: Power (mW) vs Word Size Chart

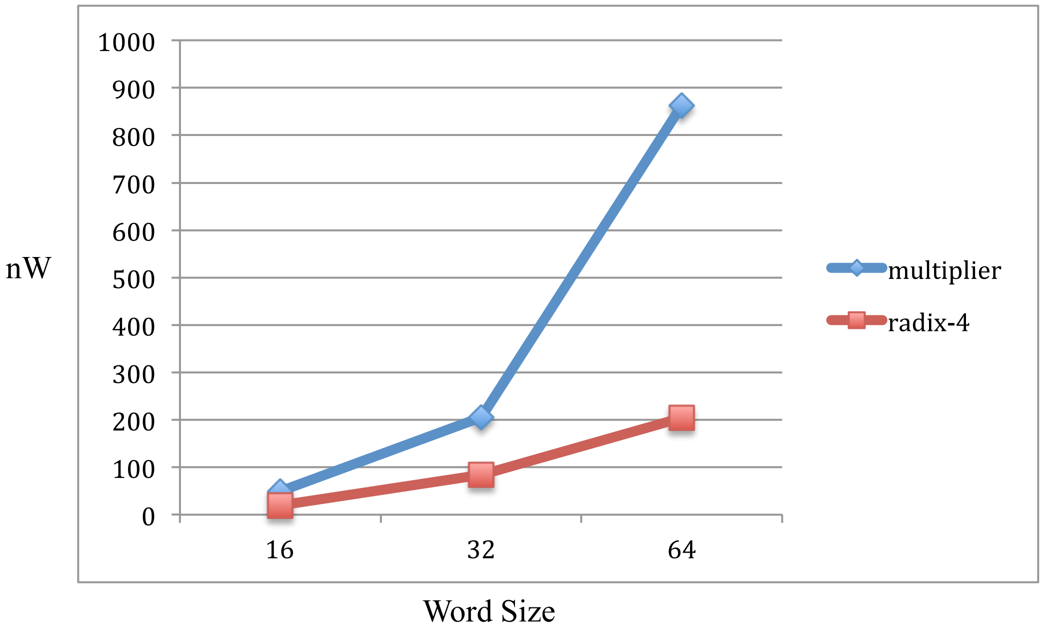

Figure 2.3.3.2: Leakage Power (nW) vs Word Size Chart

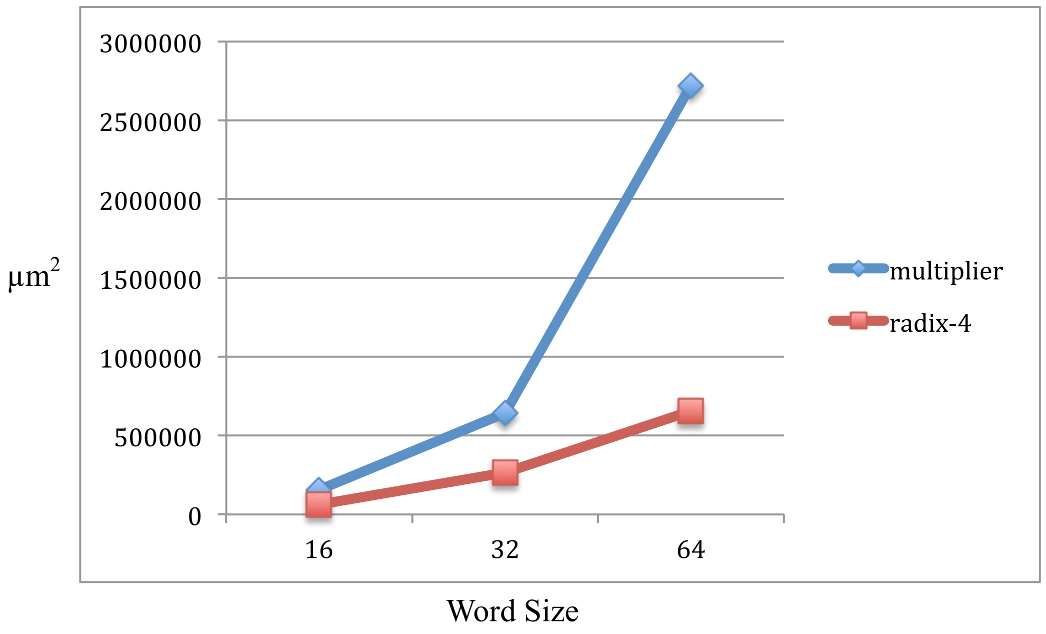

Figure 2.3.3.3: Area vs Word Size Chart

2.4. Floating-Point Square

Although the integer square and general purpose integer multiplier are very useful circuits, must real world calculations, such as finance, business, and science, require floating-point numbers. With this in mind, I used the integer square described in Chapter 3 to design an IEEE 754 standard floating-point squaring unit.

2.4.1. Summary of the IEEE 754 Floating-Point Standard [14]

When discussing the IEEE 754 Floating Point Standard [14], we first need to look at the number are represented. Figure 2.4.1.1 shows the number representation for a single precision floating-point number which is interpreted as follows:

$$\begin{cases} 0.F * 2^{-127} & E = 0 \\ \begin{cases} NaN & F \neq 0 \\ \pm Infinity & F = 0 \end{cases} & E = 255 \\ -1^{S} * 1.F * 2^{E-127} & 0 < E < 255 \end{cases}.$$

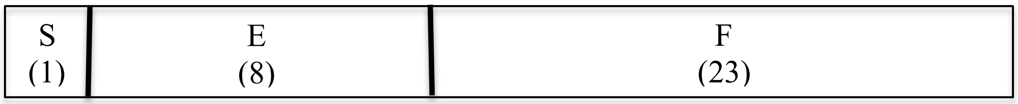

Figure 2.4.1.1: Floating-point Number Representation

Conceptually, floating point multiplication is simple. The exponents are added and multiplying the significands. However, a lot of pre- and post-processing must be performed such as lining up the significands and adjusting the exponents accordingly as well as setting flags in case an exact result can’t be calculated. The next several sections will describe how floating-point multiplier used as a comparison accomplishes these tasks.

2.4.2. Algorithm

The question now becomes how to multiply two numbers represented in IEEE floating-point format. The simplest case is when both of the inputs are normalized. In this case the significands are multiplied together with the most significant half of the product is used as the significant answer and the exponents are added together. Table 2.4.1 illustrates this simple case.

Table 2.4.1: Simple Case Floating-Point Multiplier

Operations	Exponent	Significant
	1000 0011	000 0001 0100 0001 1000 1010
	1000 1000	000 1000 0010 1000 0001 0010
Exponent Add/Significant Multiply	1000 1100	000 1011 1111 0000 1100 1010

The first complication that covered in the above example is if the product of the significands is equal to or greater than 2. The value is 2 or greater if and only if the most significant bit of the product is one. If this occurs, the product must be right shifted by one and the sum of exponents must be incremented by one. A more complicated issue is if one or both of the inputs are not normalized. Since the inputs to the multiplier are not guaranteed to be normalized, the inputs must be checked to see if they are normalized and if not, it must be normalized. Table 2.4.2 illustrates what happens if the above algorithm is used to multiply a normalized number with an unnormalized number.

Table 2.4.2: Normalized/Unnormalized Floating-point (Incorrect)

Operations	Exponent	Significant
	0000 0000	010 0000 0000 0000 0000 0000
	1100 1111	001 0100 1000 0110 0000 0000
Exponent Add/Significant Multiply	0101 0000	011 1001 1010 0111 1000 0000

The input must be left shifted to one place past the most significant 1. The exponent then becomes 1 – the number of shifts. The exponent needs an extra bit to be stored in 2's compliment form. The example shown in Table 2.4.3 shows the multiplication of a normalized and unnormalized floating point numbers.

Table 2.4.3: Normalized/Unnormalized Floating-point Multiplication

Operation	Exponent	Significant
	0000 0000	010 0000 0000 0000 0000 0000
	1100 1111	001 0100 1000 0110 0000 0000
Shift Significant/Adjust Exponent	1 1111 1111	000 0000 0000 0000 0000 0000
	0 1100 1111	001 0100 1000 0110 0000 0000
Product	0100 1111	001 0100 1000 0110 0000 0000

The other special cases of NAN, positive infinity, and negative infinity can be accounted for by detection and assigning the output to the appropriate answer.

2.4.3. Implementation

2.4.3.1. Floating point multiplication

The floating-point multiplication implemented in [15] breaks the multiplication into the following components: preprocessor, special case detector, pre-normalizer, multiplier, exponent adder, normalizer, shifter, rounder, flagger, and assembler. The components are connected as shown in Figure 2.4.3.1. Each of the components will be discussed in detail in the following sections.

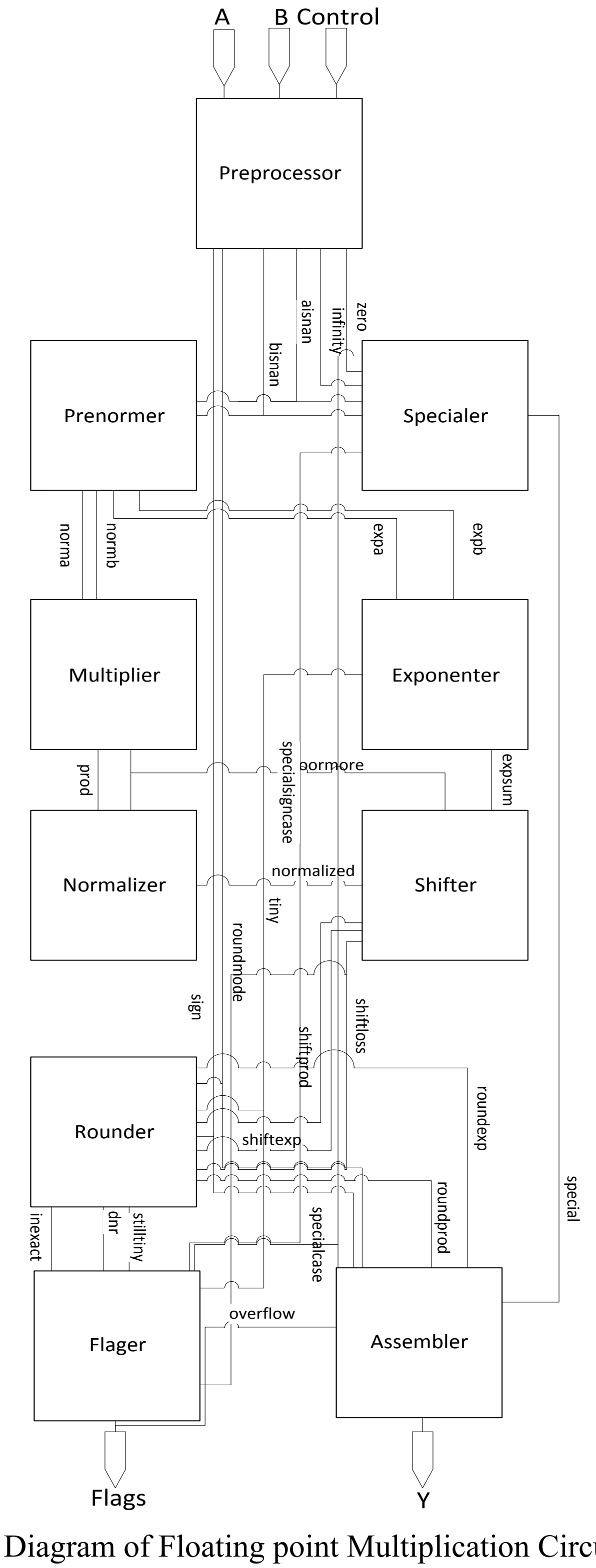

Figure 2.4.3.1: Diagram of Floating point Multiplication Circuit [15]

2.4.3.1.1. Preprocessor

Before the two values can be multiplied, some preprocessing must be done. The inputs are represented by A and B for the floating point numbers and *control* for the rounding mode. The rounding mode is defined by the two least significant bits of the *control* input. First, the potential sign of output can be calculated by taking the XOR of the most significant bit of A and B. The preprocessor circuit must also determine if either A or B is one of the special cases discussed in Section 2.4.1. To determine wither either A or B is zero, all of the bits excluding most significant of each input is NORed and then outputs from the NOR of each input are ORed together producing the *zero* output. This gives the equation: $zero = \begin{pmatrix} (\sim|A[wsig-1]\&\sim|A[width-1{:}width-wexp-1])| \\ (\sim|B[wsig-1]\&\sim|B[width-1{:}width-wexp-1]) \end{pmatrix}$

where *wsig* and *wexp* stand for the bit width of the significant and bit width of the exponent. A and B both checked to determine whether or not it is infinity or NAN. First all of the bits of the exponent are checked to see if everyone is 1. This is done by ANDing all the exponent bits together. Then NOT of the output NOR gate used earlier to determine if the significant is zero is ANDed with output of the AND gate to detect NAN and the output of the earlier NAND gate ANDed with the AND gate used to determine if all bits of the exponent are 1 to detect infinity. These equations further explain this:

$$aisnan = \sim(\sim|A[wsig-1])\ \&\ (\&A[width-1{:}width-wexp-1])$$

$$bisnan = \sim(\sim|B[wsig-1])\ \&\ (\&B[width-1{:}width-wexp-1])$$

$$infinity = \big((\sim|A[wsig-1]\ )\&(\&A[width-1{:}width-wexp-1])\big)\ |$$
$$\big((\sim|B[wsig-1]\ )\&(\&B[width-1{:}width-wexp-1])\big).$$

The preprocessor must also determine if either of the inputs is not normalized. An input is not normalized if the exponent is equal but the significant is not. In order to calculate this, the NOR gates used find the *zero* output. For both *A* and *B*, this equates to the NOT of the NOR of bits [wsig-1:0] ANDed with the NOR of bits [width-1: width-wexp-1]. The entire circuit for the processor is shown in Figure 2.4.3.2.

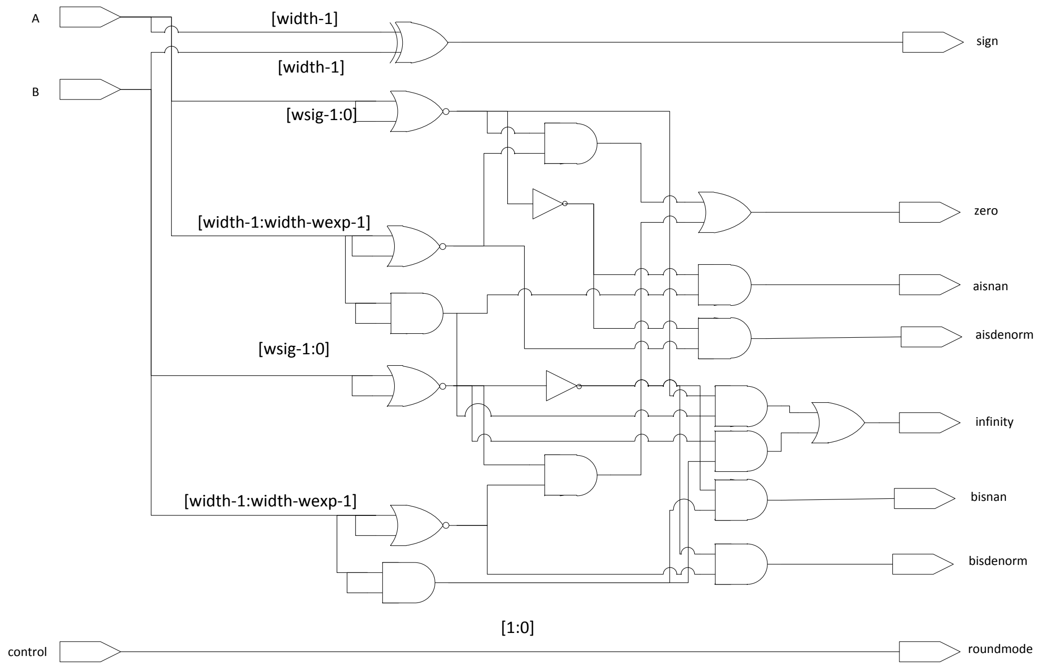

Figure 2.4.3.2:Pre-processor [15]

2.4.3.1.2. Special Case Detector

The outputs *aisnan, bisnan, zero,* and *infinity* from the preprocessor are inputs to the special case detector as well as *A* and *B*. The *specialsigncase* is defined as $aisnan \mid bisnan \mid (zero \; \& \; infinity)$ and *specialcase* is defined as$aisnan \mid bisnan \mid infinty \mid zero$. *Invalid* equates to$zero \; \& \; infinity$. If *invalid* is 1 the *specialsign* is most significant bit *A* if the following expression, which is referred to as *aishighernan* in [15], *aisnan* is true plus the significant of *A* is greater than or equal to the significant *B* or *bisnan* is false and is the most significant bit of *B* if it is false. The value to which the output *special* is assigned can best be described using the flowchart in Figure 2.4.3.3. The entire circuit for the special case detector is shown in Figure 2.4.3.4.

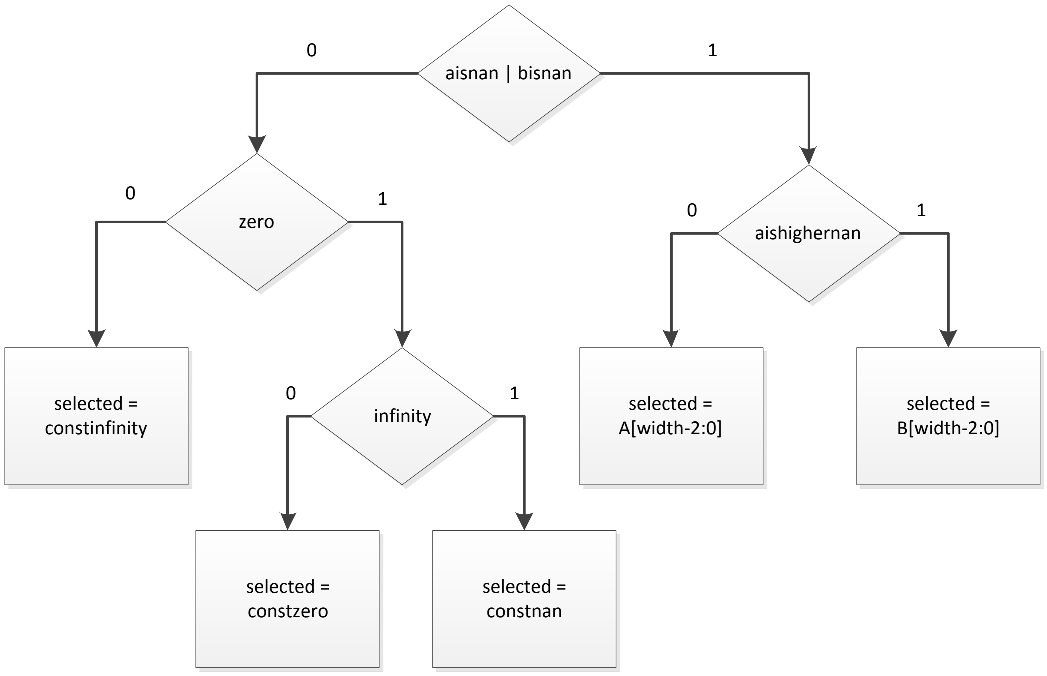

Figure 2.4.3.3: Flowchart for *selected*

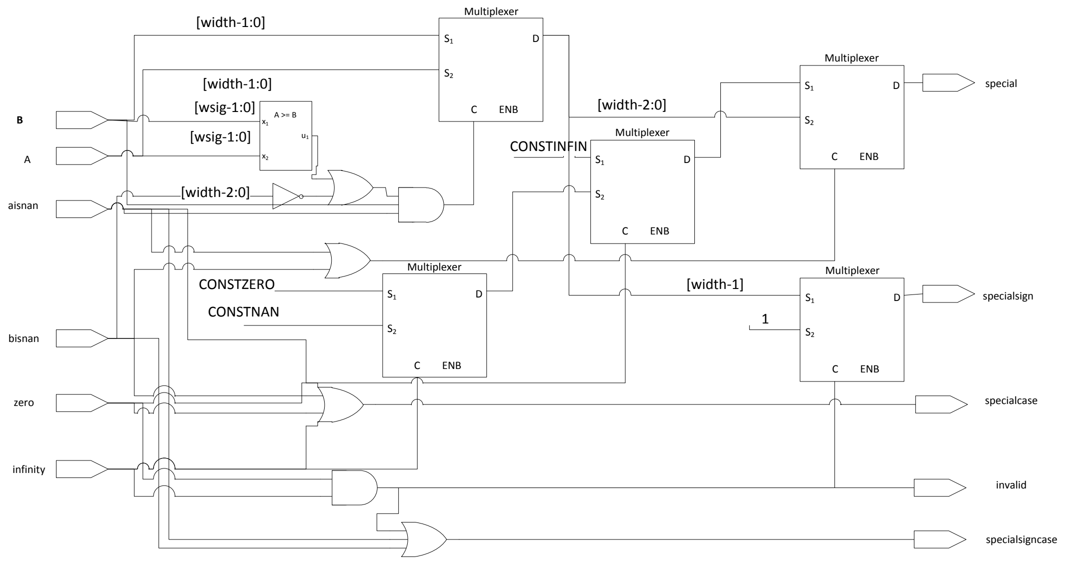

Figure 2.4.3.4: Special Case Detector [15]

2.4.3.1.3. Pre-normalizer

Using the inputs *A* and *B* as well as the outputs from the preprocessor *aisdenorm* and *bisdenorm* to normalize *A* and *B* such that the significant value is between [1, 2). If the input is unnormalized, the significant is left shifted by the width minus the bit position of the most significant 1. This pre-normalization shift function is shown in Figure 2.4.3.5. If the significant is shifted, the exponent must be adjusted such that the number still represents the same value. The entire pre-normalized circuit is shown in Figure 2.4.3.6.

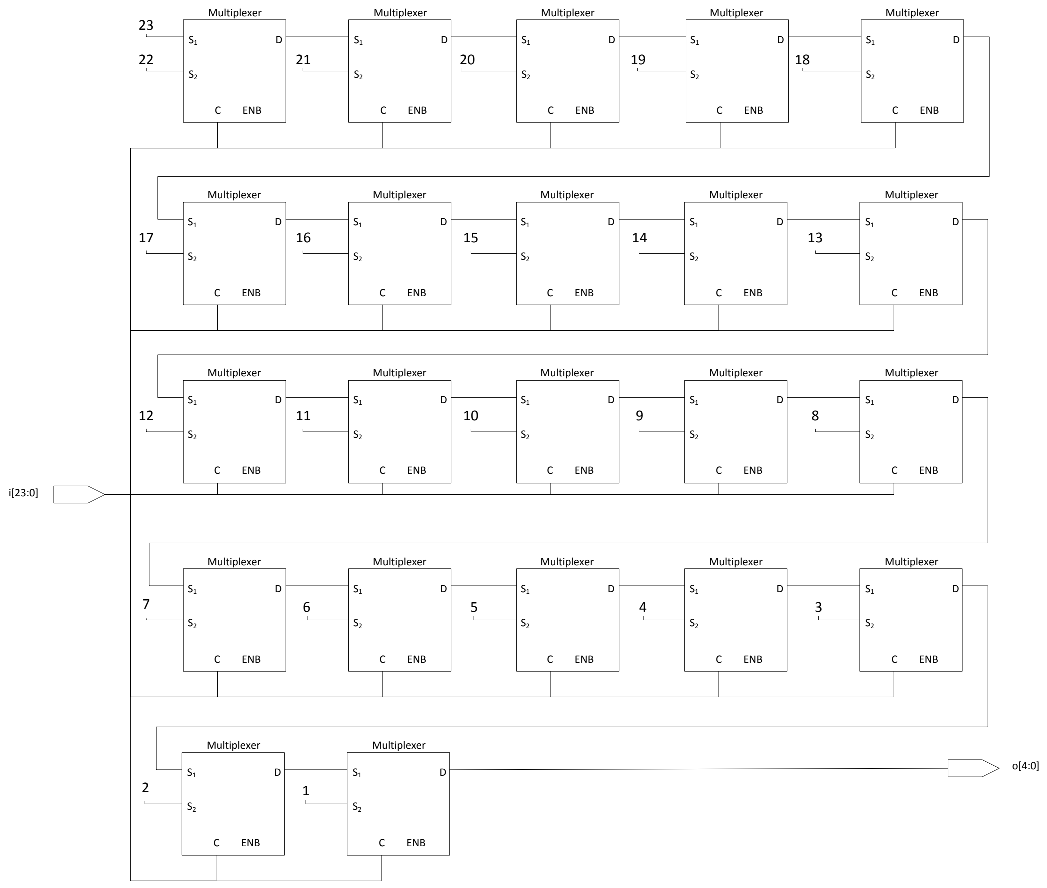

Figure 2.4.3.5: Pre-norm Shifter [15]

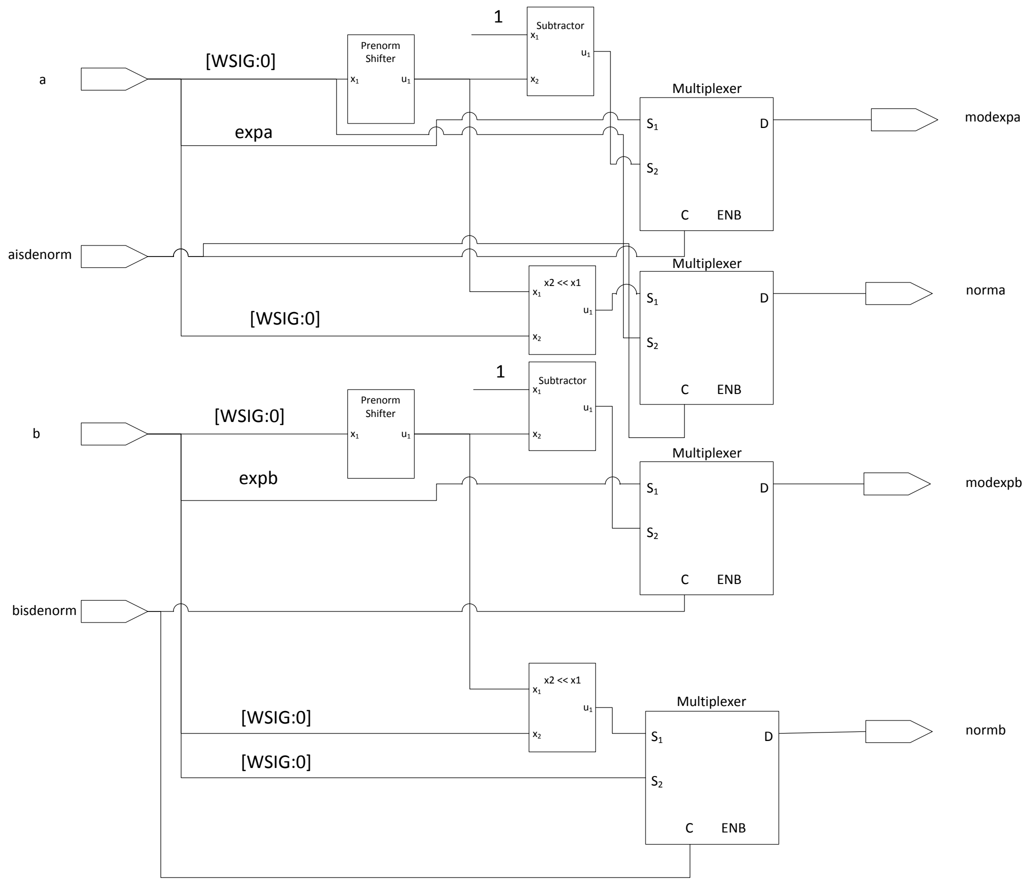

Figure 2.4.3.6: Pre-normalizer [15]

2.4.3.1.4. Multiplier

The multiplier circuit used is the carry save multiplier described in Chapter 1. The multiplier takes the normalized significands as inputs. In addition to outputting the product, the most significant bit of the product is used to determine if the product is equal to or greater than two.

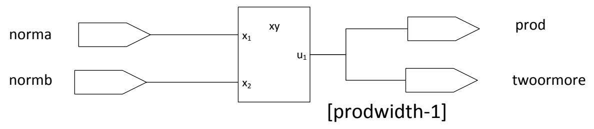

Figure 2.4.3.7: Multiplier [15]

2.4.3.1.5. Exponent Adder

In addition to multiplying the significands, the exponents need to be added. However, the exponents need to be adjusted by the bias and whether or not the product from the multiplier is more than two. The circuit in Figure 2.4.3.8 implements the following equation derived from equation 2.4.1.1:$expsum = expa + expb - (bias - twoormore)$. The bias must be subtracted from the sum of *expa* and *expb* because both have *bias* added to them. Therefore, the equation becomes$(expa - bias) + (expb - bias) + bias = expa + expb - bias$. If *twoormore* from the multiplier in Section 2.4.2.1.4 is 1, the exponent needs to incremented thus producing the equation:$expsum = expa + expb - (bias - twoormore)$. The output *tiny* is used to determine if the *expsum* is less than one. Since *expsum* is in 2's complement form, *tiny* is one if the most significant bit of *expsum* is 1 or if all of the other bits of *expsum* are 0.

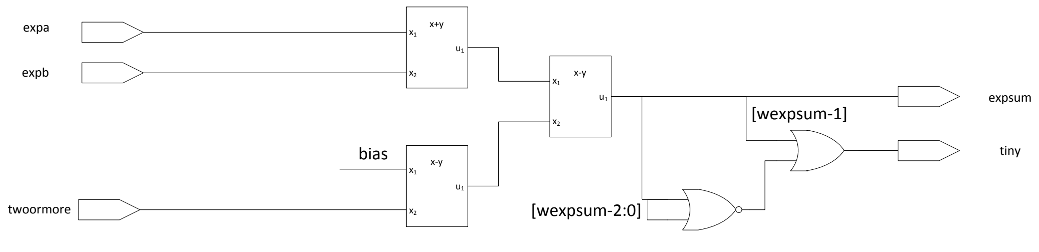

Figure 2.4.3.8: Exponent Adder [15]

2.4.3.1.6. Normalizer

If the needed the *product* from the multiplier is normalized. If both *tiny* and *twoormore* are both true than the product is already normalized, *normalized* is set equal to *prod*. Since *twoormore* represents an overflow of the product and *tiny* represents an overflow of the sum of the exponents, the product is left shifted by 1 if one of them is true or left shifted by 2 if neither one is true. This is done to place the most significant bit of the possible output in the most significant bit of the product.

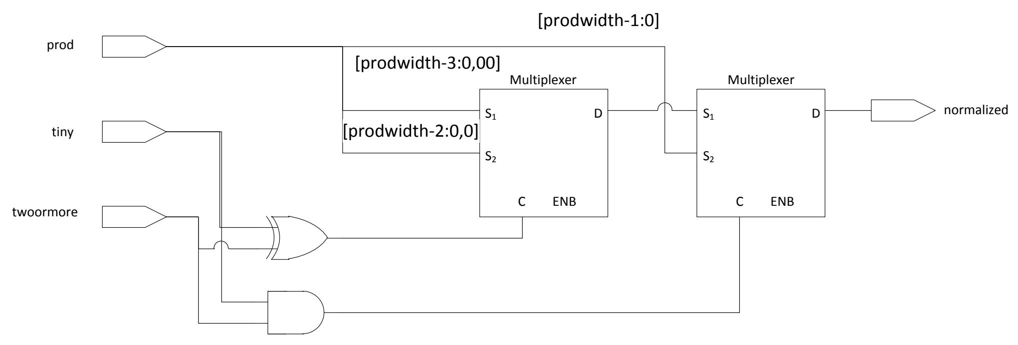

Figure 2.4.3.9: Normalizer [15]

2.4.3.1.7. Shifter

If the product is unnormalized which is indicated by *tiny* being true, then the significant may need to be right shifted in order to properly represent the true value. This is done by shifting the magnitude *expsum*. Since for *tiny* to be true the *expsum* must be less than or equal than 0, *normalized* should be right shifted by the 2’s complement of *expsum* if the most significant bit of *expsum* is 1. If *expsum* is greater than zero, *shiftexp* is set equal to *expsum* otherwise *shiftexp* is set to 0. It is also necessary to determine whether or not there was any loss of accuracy. *Shiftloss* is used to store this information. In order for *shiftloss* to be true, a shift must be necessary and either the shift causes *prodshift* to be zero or if any of the bits lost due to a shift is 1. The implementation from [15] is shown in Figure 2.4.3.10.

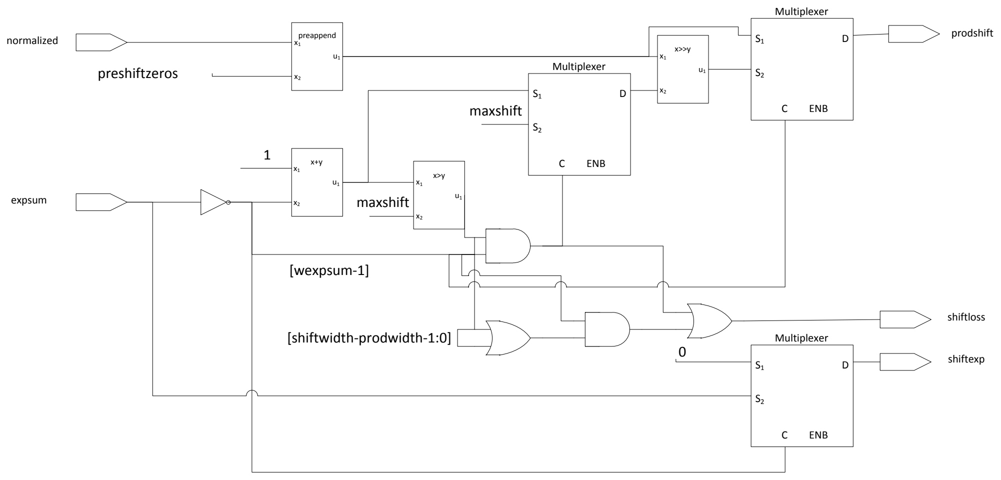

Figure 2.4.3.10: Normalizer [15]

2.4.3.1.8. Rounder

The multiplier described in Section 2.4.2.1.8 produces an output that is bit-width twice as wide as the bit widths of the inputs. However, the IEEE standard only allows for the output to be stored in the same bit width as the inputs. Since it sometimes not possible to store the exact result, the output must be rounded. The rounding in the floating-point multiplier implementation in [15] is based on the rounding methods presented in [16]. The multiplier allows four different types of rounding modes: round to nearest even, round to zero, round to positive infinity, and round to negative infinity. The rounding mode is determined by the two least significant bits of *control*. Table 2.4.4 shows the relation between *control* and the rounding mode.

Table 2.4.4: Rounding Modes [15]

Control	Rounding Mode
00	Round to Nearest Even
01	Round to Zero
10	Round to Positive Infinity
11	Round to Negative Infinity

The upper half of the shifted product is used as the basis for the answer which will be referred to as *unrounded*. The purpose of the Rounder is to determine whether or not to add 1 to *unrounded*. The simplest rounding method is round to zero because 1 is never added *unrounded* as the least significant bits are always truncated which will be referred to as *lowerBits*. The only time that 1 is added to *unrounded* is if at least one of the *lowerBits* is 1 and the rounding mode is round to nearest even and the least significant bit of *unrounded* is 1, the rounding mode is round to positive infinity and the sign bit is 0, or the rounding mode is round to negative and the sign bit is 1. This leads to following Boolean expression:

$$roundUp = |lowerBits \,\&\, \begin{pmatrix} {\sim}round[1]{\sim}round[0]{\sim}unrounded[0] \mid \\ round[1]{\sim}round[0]sign \mid \\ round[1]round[0]{\sim}sign \end{pmatrix}$$ which can

be simplified to:

$roundUp = |lowerBits \;\&\; \begin{pmatrix} \sim round[1] \sim round[0] \sim unrounded[0]| \\ round[1](round \oplus sign) \end{pmatrix}$. The exponent will need to be adjusted if the significant needs to roundup and adding one to the significant could cause the significant to be greater than or equal to 2. If this is the case, the exponent will need to be incremented and the significant left shifted by one. Since the exponent might be changed, it must be rechecked to see if the number is still tiny and must also be checked for overflow. According to the IEEE standard [14], once the value has been rounded, it is now longer exact. If the value is inexact, it must be detected which can be done by$|lowerBits\ |roundUp|round[0]$. As the initial inputs were checked in to see if they were denormal as described in Section 2.4.2.1.1, the rounded product must also be checked. Figure 2.4.3.11 shows how all the above Boolean expressions were implemented in what [14] called a rounder.

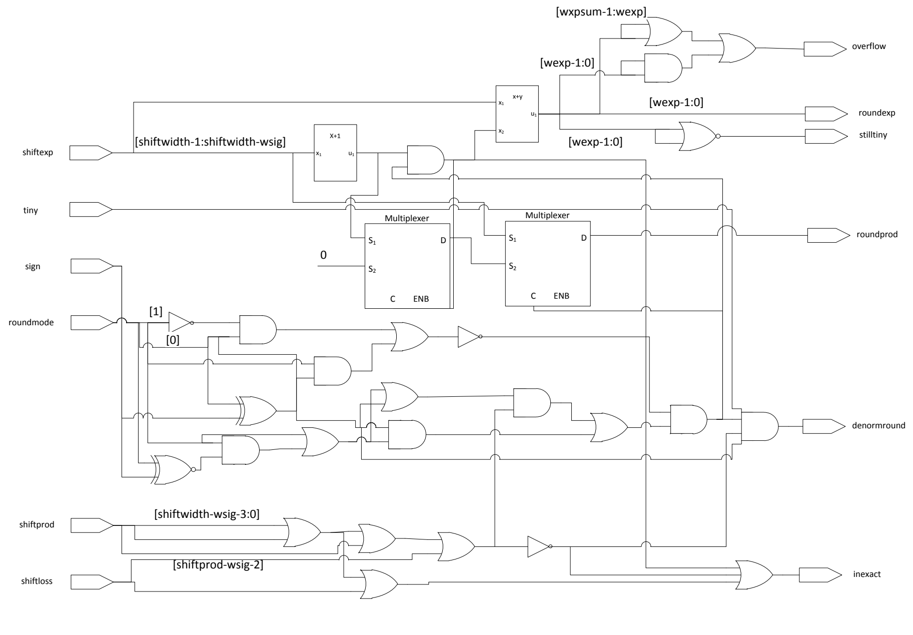

Figure 2.4.3.11: Rounder [15]

2.4.3.1.9. Flagger

As stated in 2.4.1, flags must be set in the event that an exact answer cannot be calculated. The first of such situations is a divide by zero but since this a multiplier a division by zero cannot happen therefore the divide by zero flag is set to zero. Another such case is infinity times zero which is labeled invalid. This is actually determined in the Special Case Detector in Section 2.4.2.1.2 and only needs to be assigned to the appropriate flag here. Only if it is not a special case as defined in Section 2.4.2.1.2 can the flags *underflow*, *overflow*, and *inexact* be true. The underflow flag is set if *tiny* from Section 2.4.2.1.5, *underflow* which is the OR of *shiftloss* from Section 2.4.2.1.8 and *inexact* from Section 2.4.2.1.8, and not *overflow* from Section 2.4.2.1.8. The *overflow* output is then just the *overflow* input into the Flagger from Section 2.4.2.1.8. The *inexact* input is the OR of *shiftloss* from Section 2.4.2.1.8 and *inexact* from Section 2.4.2.1.8. The *inexact* flag is the OR of the *inexact* input, *underflow* input, and the *overflow* input. The Flagger is shown in Figure 2.4.3.12.

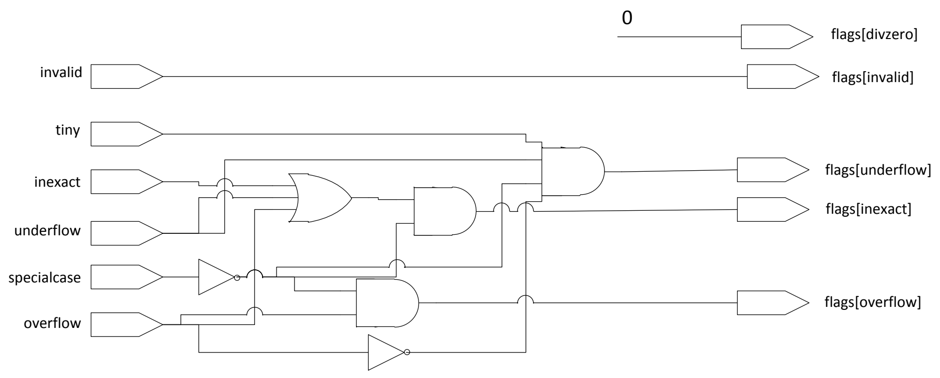

Figure 2.4.3.12: Flagger [15]

2.4.3.1.10. Assembler

Once all of the possible answers have been calculated, the right answer must be chosen which is accomplished by the Assembler. The Assembler is composed mainly of multiplexers. See Figure 2.4.3.13. The inputs *specialsigncase*, *sign*, *roundmode,* and *overflow* are to select the correct output. If *specialsigncase* is 1, the most significant bit or sign bit is *specailsign* otherwise it is set to *sign*. The logic for determining which values

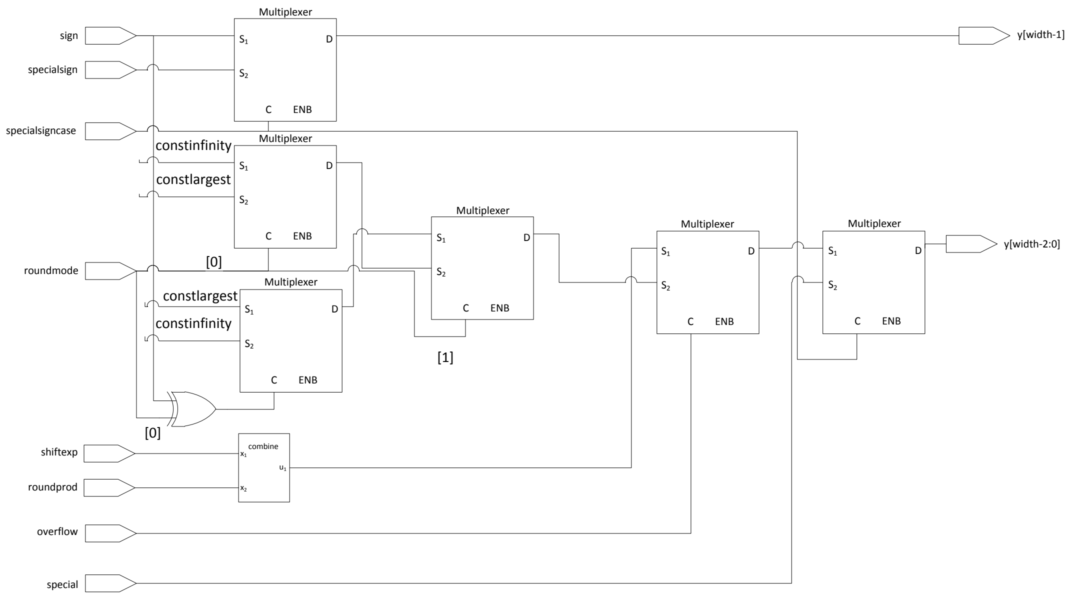

Figure 2.4.3.13: Assembler [15]

2.4.3.2. Floating-Point Squarer

Some optimizations can be implemented when focusing on the square as opposed to a general multiplication. First since there is only one input, checking for special values become simpler. The results for special value inputs are shown in Table 2.4.3.1.

Table 2.4.3.1: Special Values

Input	Output
NAN	NAN
+INFINITY	+INFINITY
-INFINITY	+INFINITY
UNNORMALIZED	UNNORMALIZED

One major change and a few minor changes were made to the floating-point multiplier discussed earlier. The first change was in the preprocessor. The preprocessor is simplified to the following Boolean expressions:

$$zero = \sim|A[wsig-1]\&\sim|A[width-1{:}width- \ wexp-1]$$

$$aisnan = \sim(\sim|A[wsig-1])\ \&\ (\&A[width-1{:}width-wexp-1])$$

$$infinity = \big((\sim|A[wsig-1]\ )\&(\&A[width-1{:}width-wexp-1])\big)\ .\ [23]$$

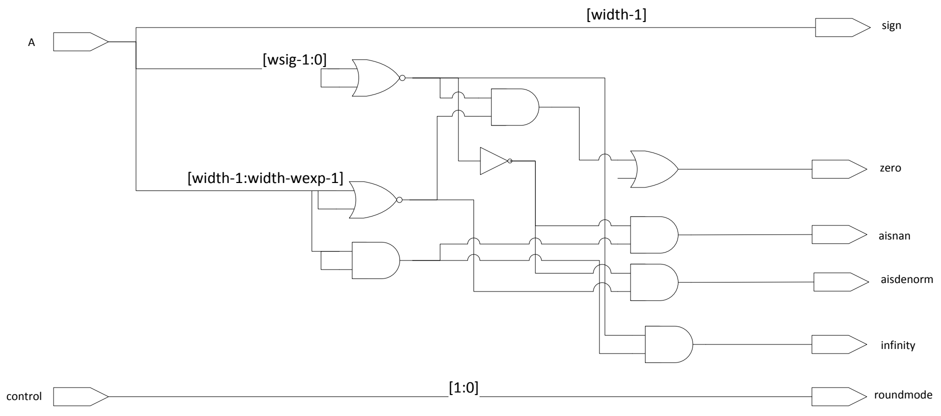

Figure 2.4.3.2.1: Simplified Pre-processor

The simplified special case detector is described as follows. The *specialsigncase* is defined as $aisnan \mid (zero \,\&\, infinity)$ and *specialcase* is defined as$aisnan \mid infinty \mid zero$. *Invalid* equates to$zero \,\&\, infinity$. If *invalid* is 1, the *specialsign* is most significant bit of *A*.[23]

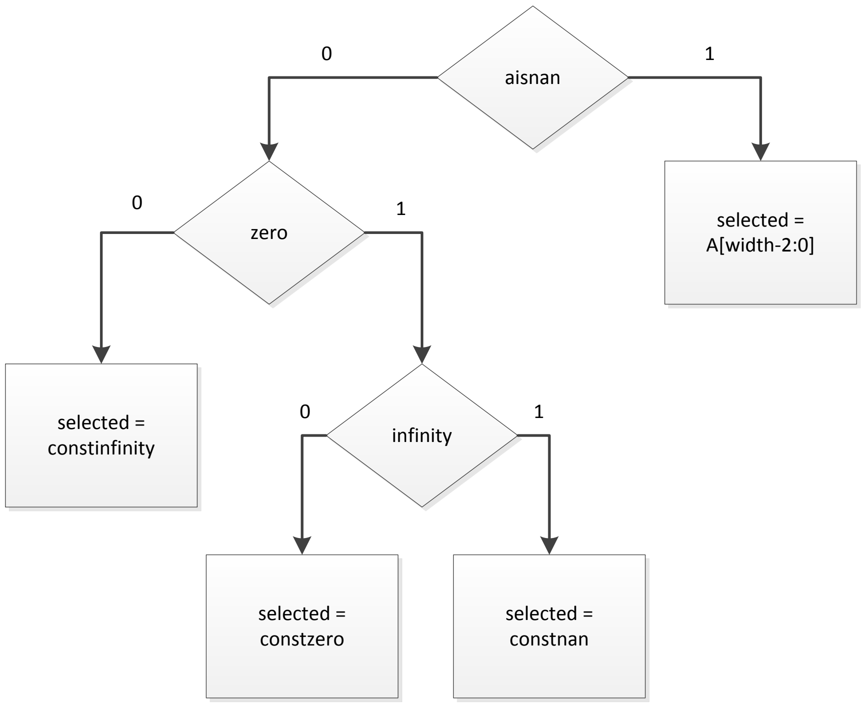

Figure 2.4.3.2.2: Flowchart for selected [23]

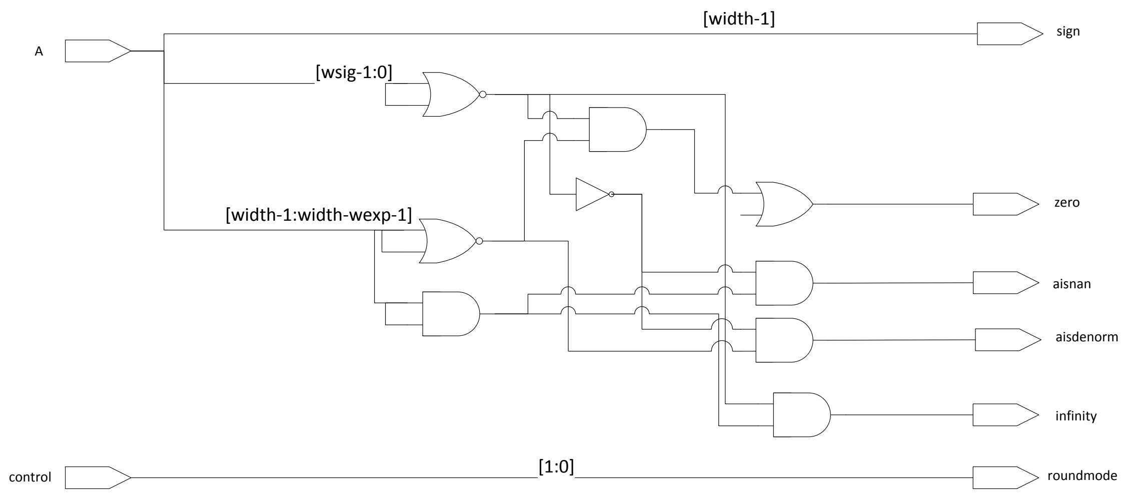

Figure 2.4.3.2.3: Simplified Special Case Detector [23]

The pre-normalized for the square is half the size of the pre-normalized for the multiplier. The circuit has only one pre-normalized shifter, subtractor, and shifter as well as 2 multiplexers. The pre-normalized now produces two outputs *modexpa* and *norma*. The new circuit is shown in figure 2.4.3.2.4.

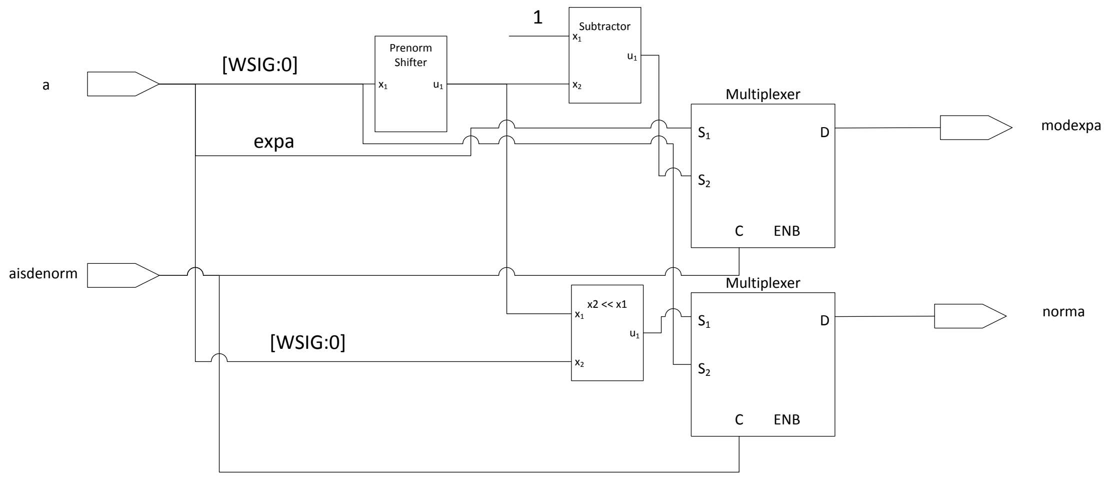

Figure 2.4.3.2.4: Simplified Pre-normalizer [23]

The adder in the exponent adder is replaced with a left shift by 1. The multiplier is replaced by a full dual recoded radix-4 square. The squaring circuit has bit width 24 and 54 for single precision and double precision respectively.

2.4.4. Results

The floating-point radix-4 dual recoded squaring circuit and a general purpose floating-point multiplier were both implemented in Verilog and mapped to OSU standard cell library [5]. Both circuits were constrained to run with-in a 50ns clock-edge and were implemented for single and double precisions. The charts in Figures 2.4.4.1-2.4.4.3 show a substantial gain in power, leakage power, and area for our customized squaring circuit compared to a multiplier circuit.

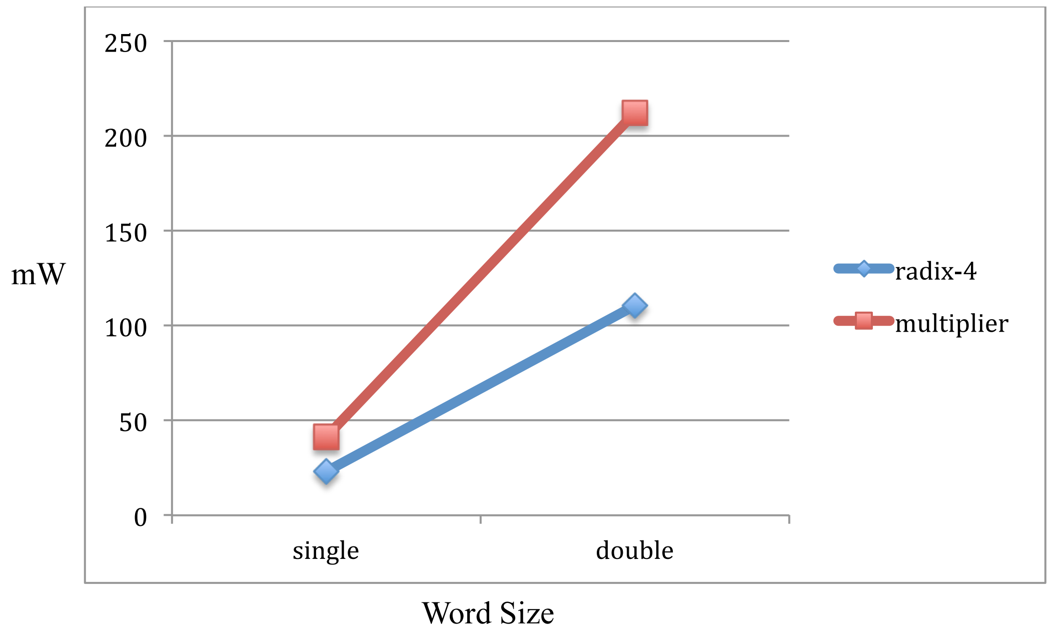

Figure 2.4.4.1: Power (mW) vs Word Size chart

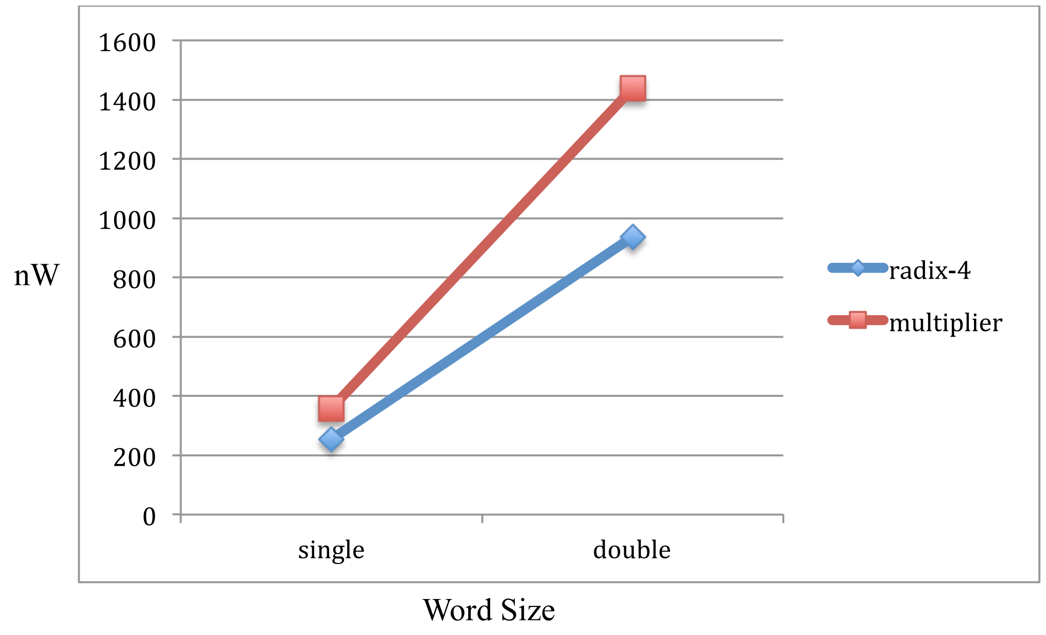

Figure 2.4.4.2: Leakage Power(nW) vs Word Size Chart

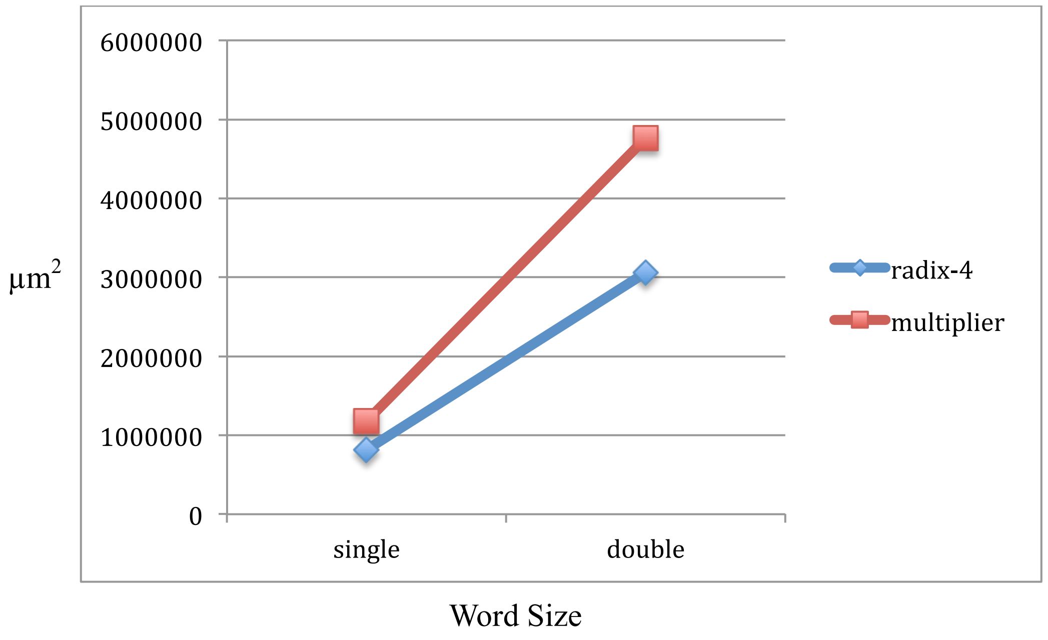

Figure 2.4.4.3: Area vs Word Size Chart

Chapter 3

MULTIPLICATION

3.1. Parallel Multipliers

A parallel multiplier is a multiplier in which partial products of the multiplicand are produced and then added together. The partial products can be added together serially or in tree structure. Two common types of parallel multipliers are the Pezaris and the Booth multipliers.

3.1.1. Pezaris Array Multiplier

The Pezaris array multiplier is generally used for two's complement multiplication. The partial products are generated by AND gates while the summation section of the adder consist of four different types of full adders. Two of the full adders are positively weighted while the other two full adders are negatively weighted. Below are the Boolean expressions for the carry and sum of all four adder types where type 0 and 1 are positively weighted while type 2 and 3 are negatively weighted.[18] The sum output is the same for all full types: $s = a \oplus b \oplus c_{in}$. The carry out for each type is $c_{out} = ab + ac_{in} + bc_{cin}$ for types 1 and 3 and $c_{out} = ab + a\overline{c_{in}} + b\overline{c_{in}}$ for types 1 and 2. [18] The output from the array multiplier can also be read serially has the least significant bits are calculated before the most significant bits.

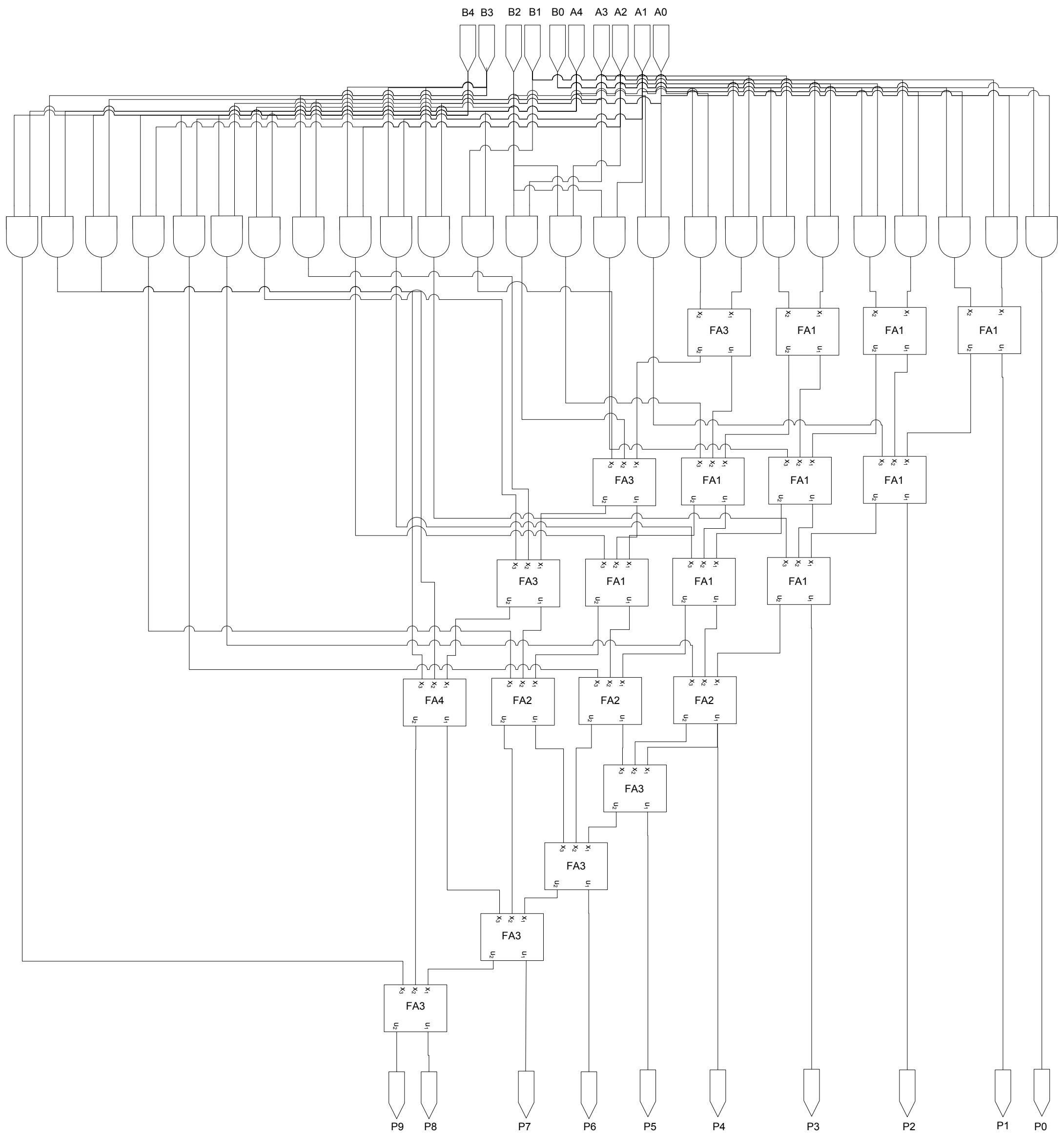

Figure 3.1.1.1: Data path of a Pezaris Multiplier [18]

3.1.2. Booth Multiplier

In 1951 Andrew Booth presented a new algorithm for multiplication in [19]. The true strength of his algorithm was that it was able to multiply two numbers efficiently no matter the sign of either number. A radix-2 recoding was used in which 00 represented 0, 01 represented 1, 10 represented -1, and 11 represented 0. The initial product is created by adding leading zeroes to the multiplier. When the Booth digit is 1, the multiplicand is added to the left half of the current product and then an arithmetic shift right is performed on the current product. When the Booth digit is -1, the multiplicand is subtracted from the left half of the current product and then an arithmetic shift right is performed on the current product. When the Booth digit is 0, on an arithmetic shift right is performed. Table 3.1.1 shows an example of the Booth algorithm by multiplying -5 and 3 to produce -15. [19]

Table 3.1.1: Booth multiplication of -5 and 3

0000 1011	Booth-4 select digits
1101 1011	-1 – Subtract
1110 1101	Arithmetic Shift right (ASR)
1111 0110	0 – ASR
0010 0110	1 – Add
0001 0011	ASR
1110 0011	-1 - Subtract
1111 0001	ASR
1111 0001	Product (-15)

The Booth multiplier can be used in mobile devices for reduction in area and power consumption.[20] In [20], the authors are able to make gains in area and power consumption through the use of counterflow organization and a merged arithmetic/shifter unit. The counterflow organization is when data and commands flow in opposite directions across the pipeline.[20]

While the initial Booth algorithm is not a parallel multiplier, it is the basis of the Booth recoded multiplier which is a parallel multiplier. The Booth recoded multiplier is used in digital signal processing as well as image compression. [21, 22] The basic design for a Booth recoded multiplier is to produce the product in two stages. The first stage all of the partial products are calculated and the second stage the partial product are added together. For the following design description, a radix-4 Booth recoded multiplier is assumed.

The partial products stage consists of two circuit types: a Booth encoder and a partial product generator (PPG). The Booth encoder takes as inputs 3 bits from the multiplier and outputs 3 control signals: negate, shift, and zero. The inputs to the PPG are the multiplicand and the 3 control outputs of the Booth encoder. The output of the PPG, which is called the partial product, can be any of the following zero, multiplicand, negated multiplicand, shifted multiplicand, or shifted and negated multiplicand.

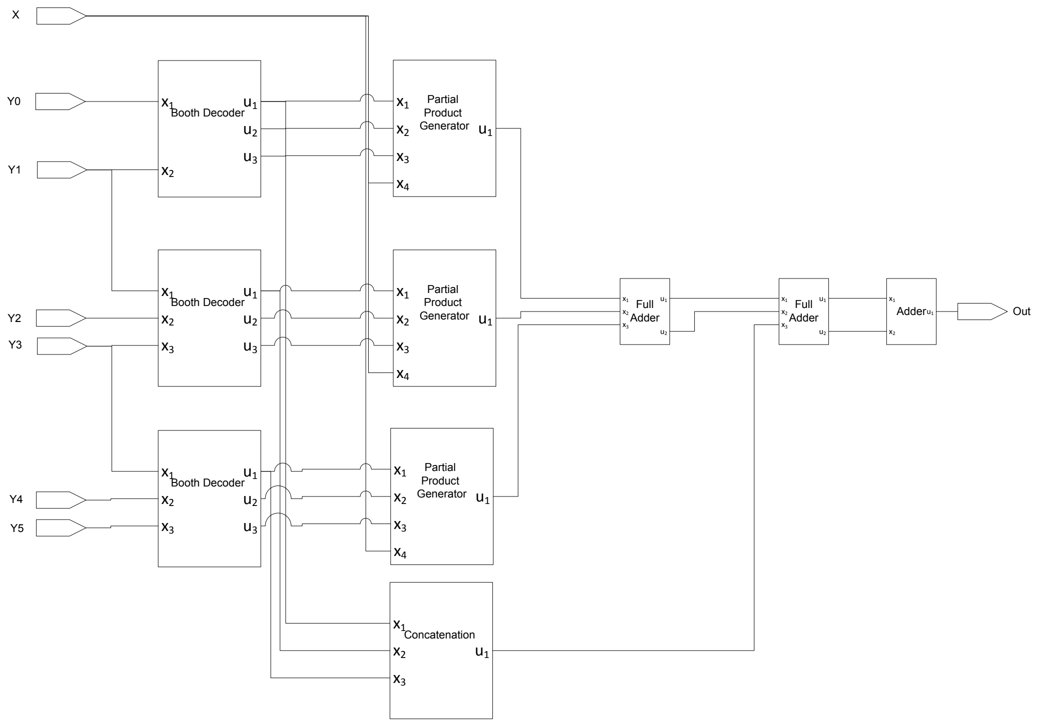

Figure 3.1.2.1: Data path of a Booth Multiplier [21]

3.2. Tree Multipliers

3.2.1. Dadda Multiplier

In 1965, Luigi Dadda presented a new tree multiplier which has been used in high speed floating point multipliers and regular connectivity. [25, 26] The Dadda tree has worst case *O(n)* where *n* is the bit width of the inputs. Like many other tree multipliers, the first step is to form partial products by ANDing together of the two inputs as shown in Figure 3.2.1.1. However, unlike other multipliers the Dadda tree multiplier consolidates partial products as late as possible. [24] Dadda used a dot notation to illustrate his algorithm which is shown in Figure 3.2.1.2. Each dot represents a bit of a partial product. Each rectangle box around a column of dots represents a full adder if 3 are enclosed and a half-adder if the box encloses 2 dots. The last rectangle represents a carry propagate adder. The diagonal lines represent carries. As shown in Figure 3.2.1.3, a Dadda tree multiplier takes 4 full adders, 2 half adders, and 6-bit carry propagate adder.

$$\begin{array}{r}
a_3 a_2 a_1 a_0 \\
\times\ b_3 b_2 b_1 b_0 \\
\hline
a_3 b_0 a_2 b_0 a_1 b_0 \\
a_3 b_1 a_2 b_1 a_1 b_1 \quad\quad \\
a_3 b_2 a_2 b_2 a_1 b_2 \quad\quad\quad\quad \\
a_3 b_3 a_2 b_3 a_1 b_3 \quad\quad\quad\quad\quad\quad
\end{array}$$

Figure 3.2.1.1: Initial Partial Product for a Dadda Multiplier

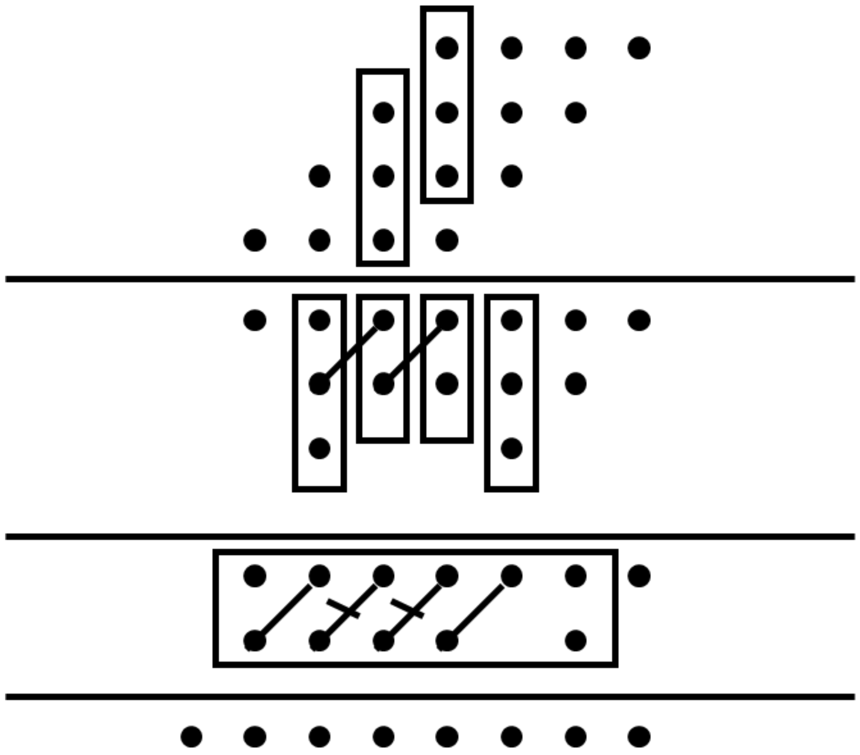

Figure 3.2.1.2: Dot Diagram of a Dadda Tree multiplier

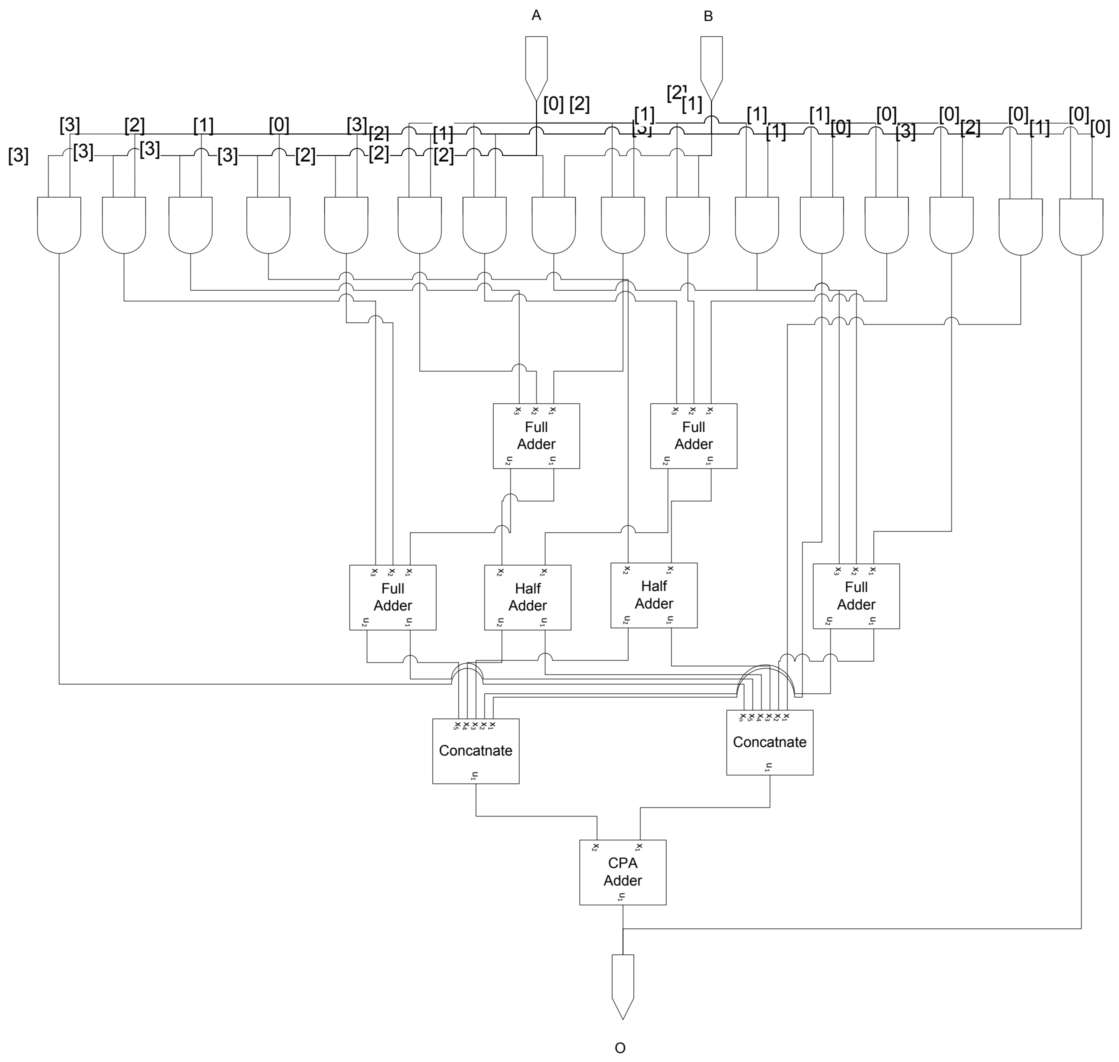

Figure 3.2.1.3: Data Path for a Dadda Tree Multiplier

3.2.2. Wallace Multiplier

Like the Dadda tree multiplier, the Wallace tree multiplier starts by ANDing all the input bits to create partial product, but unlike the Dadda multiplier the Wallace multiplier reduces the number of partial products as early as possible. This is done by grouping the initial partial products into groups of three to feed into full adders and this continues until there only two partial products left which are added together in carry propagate adder. This data path is show in Figure 3.2.2.1. [27]

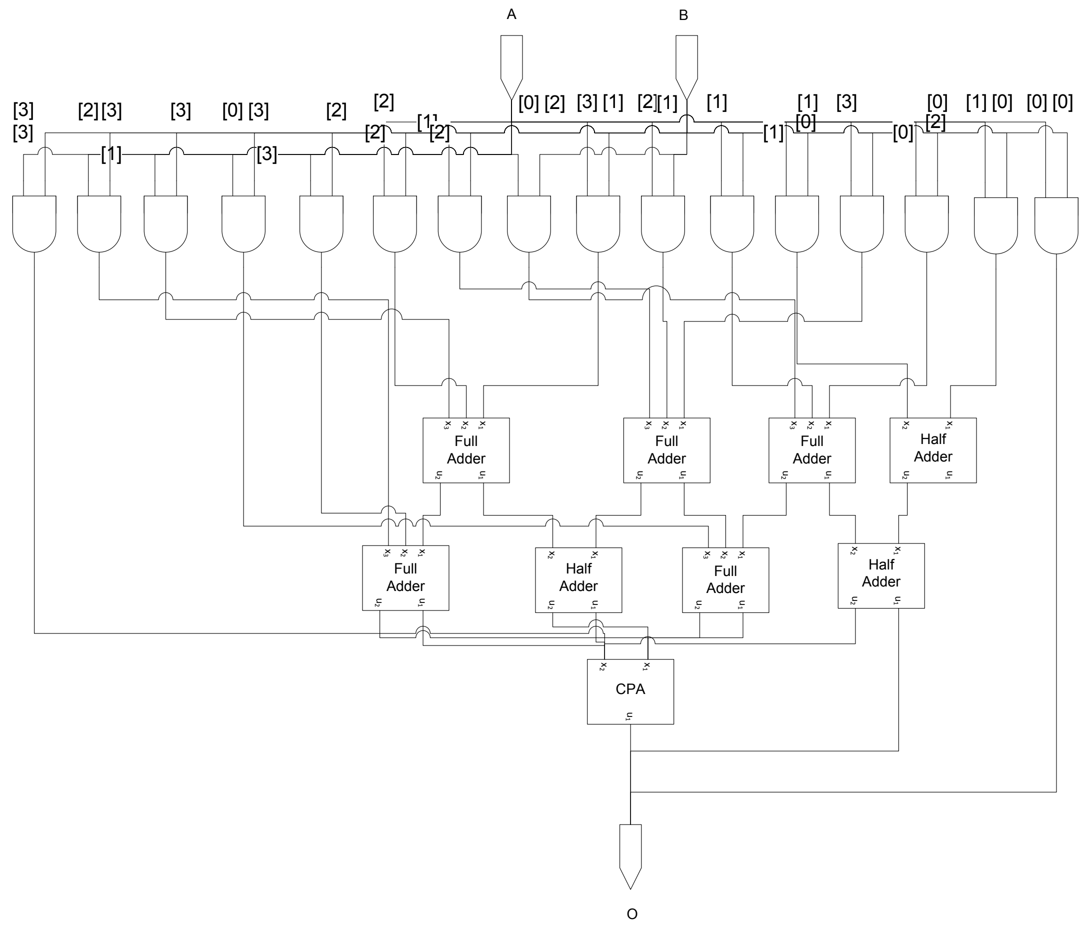

Figure 3.2.2.1: Data Path for a Wallace Tree Multiplier

Over the years since the Wallace multiplier was introduced, improvements have been made. In [28], replacing the carry propagation adder with parallel prefix adder was able to improve time delay. Other proposed improvements include the use of 4:2 and 5:2 compressors instead of 3:2 compressors, i.e. full adders, and a carry select adder instead of a carry propagate adder. [29, 30]

While the parallel and tree multipliers have been presented separately, principles of each can be used together. In [31], a Wallace tree was presented with the initial partial products are created through Booth encoding and partial product generators as opposed to AND gates.

3.3. Integer Multiplication Using Squaring Units

As mobile devices become a larger and larger part of our lives and more mobile devices are announced seemingly daily such as the Kindle and iPad, power consumption of electronic devices becomes more and more important. Capitalizing on the power savings of the Radix-4 Recoded squaring unit, a multiplier was designed using the squaring unit. The purpose of this design is to produce a low power multiplier with minimal speed lose. With a lower power multiplier, battery life of mobile devices should increase.

3.3.1. Algorithm

The design is based on the following mathematical identity.

$$\begin{aligned}
a \times b &= \tfrac{1}{4}(4ab) \\
a \times b &= \tfrac{1}{4}(2ab + 2ab) \\
a \times b &= \tfrac{1}{4}(2ab + 2ab + a^2 + b^2 - a^2 - b^2) \\
a \times b &= \tfrac{1}{4}(a^2 + 2ab + b^2 - (a^2 - 2ab + b^2)) \\
a \times b &= \tfrac{1}{4}((a+b)^2 - (a-b)^2)
\end{aligned}$$

From this identity, the following data path shown in Figure 3.3.1.1 was created.

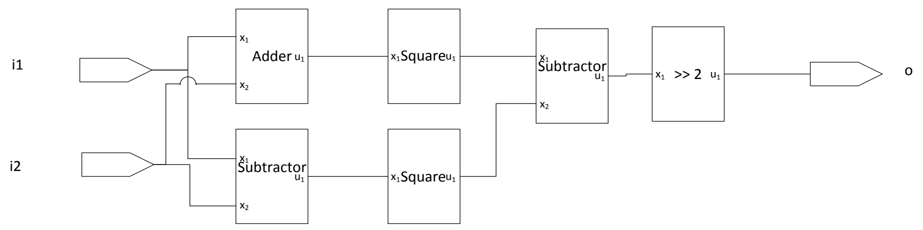

Figure 3.3.1.1: Data Path for Integer Multiplication Using Squaring Units

3.3.2. Unsigned Multiplication

3.3.2.1. Implementation

Slight changes needed to be made to the implementation of the Radix-4 squaring unit. The issue during implementation that caused this was that the inputs for the squaring unit no longer have bit widths that are multiples of 4. This occurs due to the need of an overflow bit for the results of the adder and subtractor. One possible solution to this is to divide the output of the adder and the output of the subtractor by 2 before the values are squared. However, this adds a new issue. Since we are dealing with integer numbers and not real numbers, truncating becomes an issue when dividing the inputs to the squaring circuits by 2. This issue is shown the following example of multiplying 3 times 2 in Table 3.3.1.2.1. In this example, 3 times 2 equals 4 which we all know is incorrect. This error occurs because when we 5 divided by 2 should be 2.5 instead of 2 and 1 divided by 2 should be .5 instead of 0. The squares of 2.5 and .5 are 6.25 and .25 respectively. By subtracting .25 from 6.25, the correct answer for 3 times 2 is calculated. In order to fix this issue, the *n* most significant bits of the inputs to the squaring circuits are added to the outputs of the squaring circuits. This adds if and only if the input has been truncated. This method gives you the output as if the value was truncated after it was squared. Since we only add the x when $x[0] = 1$, then the square is equal to the truncated squared value, x^2 plus .25. The following mathematical proof shows that our method works

$$x^2 + .25 = (x - .5)^2 + x$$

$$x^2 + .25 = x^2 - x + .25 + x$$

$$x^2 + .25 = x^2 + .25$$

In order to prove this identity in more general terms, the following proof, where s represents the number right shifts and is an positive integer greater than 0, is needed.

$$x^2 + \frac{1}{2^{2s}} = \left(x - \frac{1}{2^s}\right)^2 + \frac{x}{2^{s-1}}$$

$$x^2 + \frac{1}{2^{2s}} = x^2 - \frac{2x}{2^s} + \frac{1}{2^{2s}} + \frac{x}{2^{s-1}}$$

$$x^2 + \frac{1}{2^{2s}} = x^2 + \frac{1}{2^{2s}}$$

From the above equation, if a number was right shifted by 2 before being squared, the squared value would have to have to add x and $\frac{x}{2}$ if both bits shifted to the left are 1. Figure 3.3.2.1 illustrates what the data path of this circuit would look like. Note that the circuit equates to this equation: $\lfloor i^2 \rfloor = i[16:1]^2 + i[0]i[16:1]$.

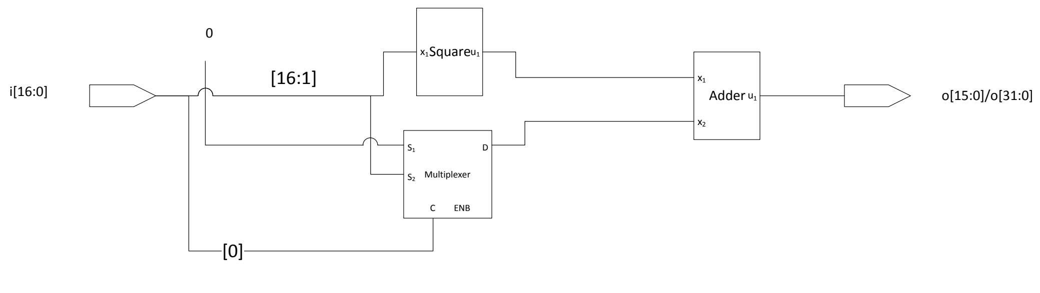

Figure 3.3.2.1: Example Truncation Correction Data Path

Table 3.3.2.1.2 revisits our example 3 times 2. Adding these corrections to the data path shown in Figure 3.3.2.1, the circuit shown in Figure 3.3.2.2 was created.

Table 3.3.1: Truncation Example

Operation	Binary Value	Binary Value
Inputs	0000000000000011	0000000000000010
Add/Subtract	0000000000000101	0000000000000001
Right Shift by 1	0000000000000010	0000000000000000
Square	0000000000000100	0000000000000000
Subtract	0000000000000100	

Table 3.3.2: Truncation Correction Example

Operation	Binary Value	Binary Value
Inputs	0000000000000011	0000000000000010
Add/Subtract	0000000000000101	0000000000000001
Right Shift by 1	0000000000000010	0000000000000000
Square	0000000000000100	0000000000000000
Conditional Add	0000000000000110	0000000000000000
Subtract	0000000000000110	

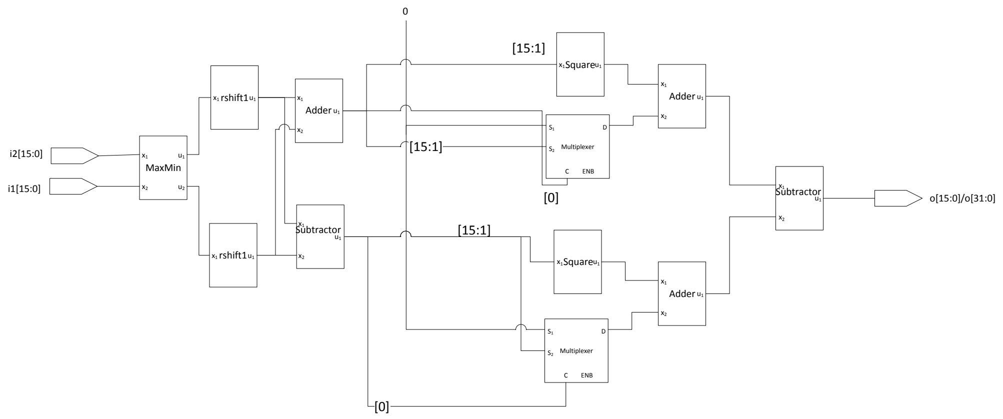

Figure 3.3.2.2: Radix-4 Dual Booth Recoded Multiplier by Squaring

The above circuit however was taking too much power and area. I was able to optimized the circuit by replacing the second 2-1 MUX in all of the PSG’s with AND gates to conditionally zero out the output as well as replacing the 2-1 MUXs in the truncation correction with AND gates. This optimization was able to improve the area but not the power consumption.

3.3.2.2. Results

The general purpose integer multiplier created from radix-4 dual recoded squaring circuits and a general purpose multiplier were both implemented in Verilog and mapped to OSU standard cell library [5] for the for the lower n-bits of the result. Both circuits were constrained to run with-in a 50ns clock-edge and were implemented for 16, 32, and 64 bit-widths. The charts in Figures 3.3.2.3-3.3.2.5 show a gain in power, leakage power, and area for our customized radix-4 based multiplier compared to a multiplier circuit.

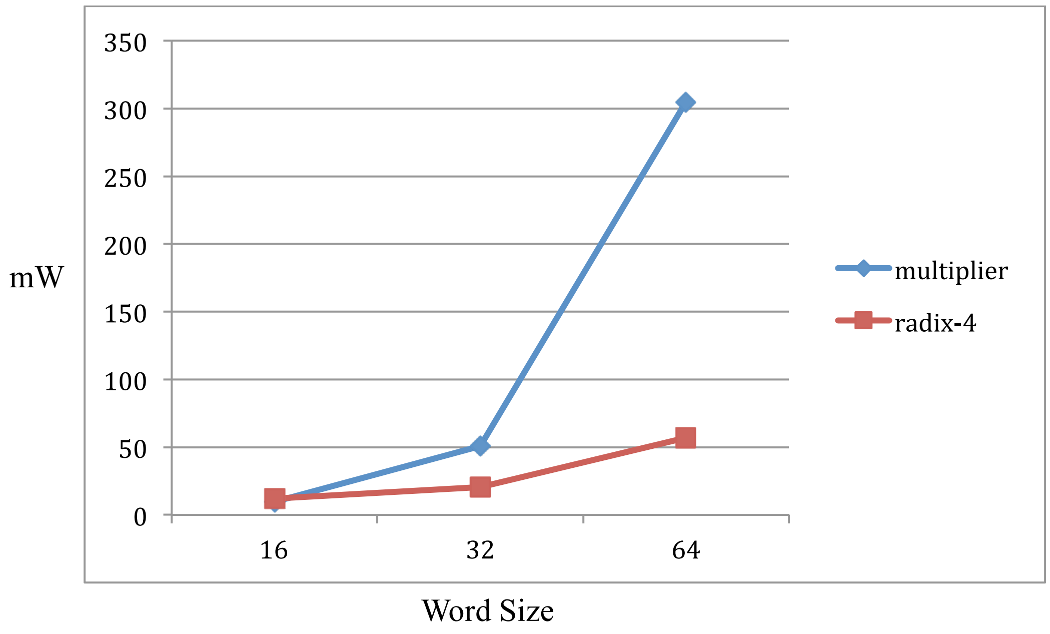

Figure 3.3.2.3: Power (mW) vs Word Size Chart

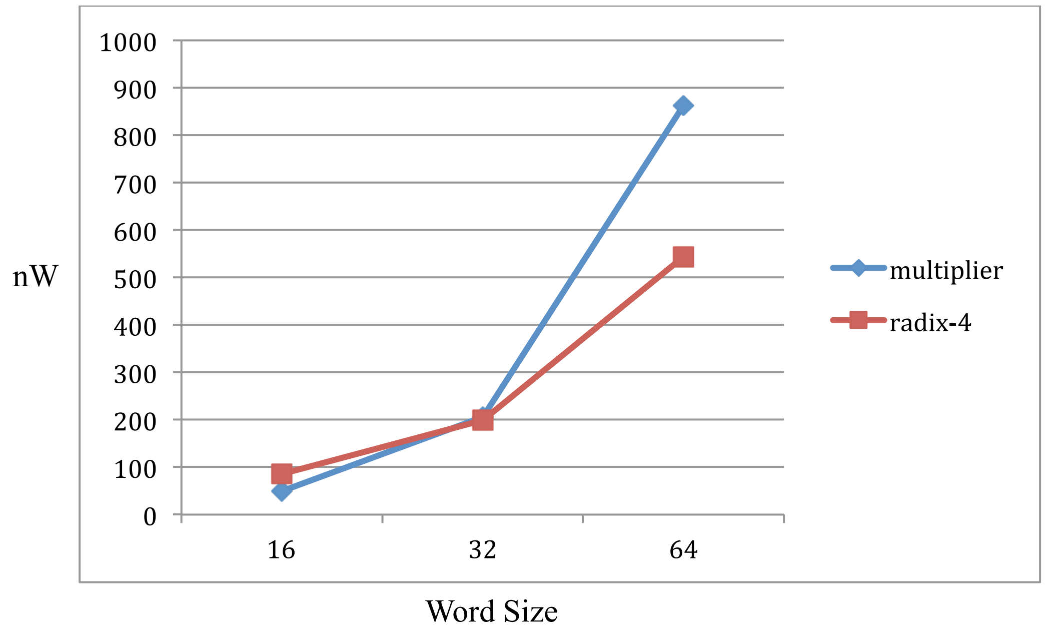

Figure 3.3.2.4: Leakage Power(nW) vs Word Size Chart

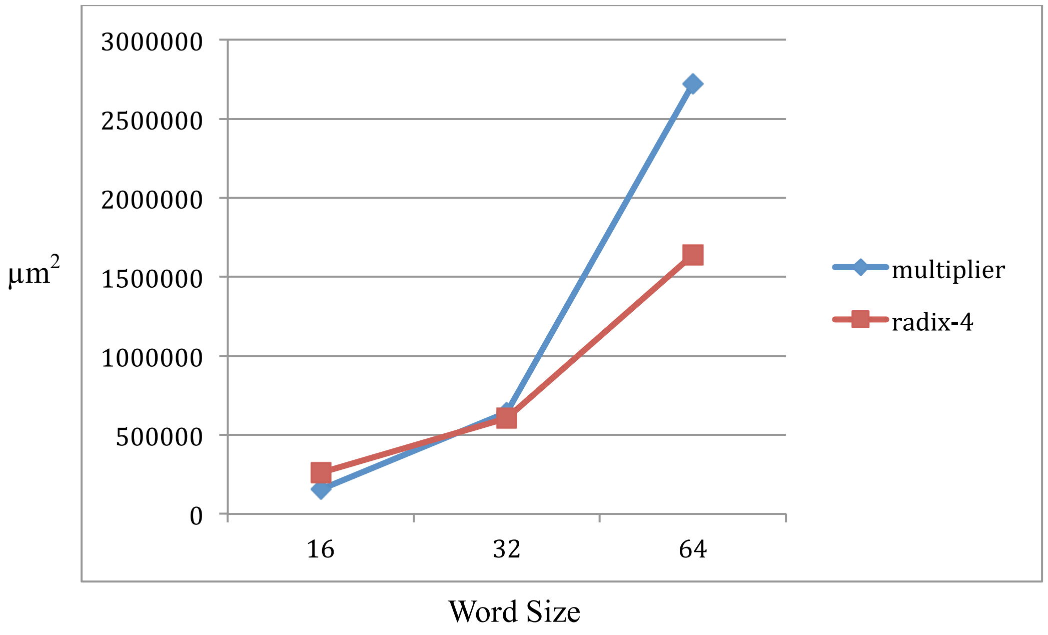

Figure 3.3.2.5: Area vs Word Size Chart

The general purpose integer multiplier created from radix-4 dual recoded squaring circuits and a general purpose multiplier were both implemented in Verilog and mapped to OSU standard cell library [5] for a n-bits of the result. Both circuits were constrained to run with-in a 100ns clock-edge and were implemented for 16, 32, and 64 bit-widths. The charts in Figures 3.3.2.6-3.3.2.8 show a loss in power but gains in leakage power and area for our customized radix-4 based multiplier compared to a multiplier circuit.

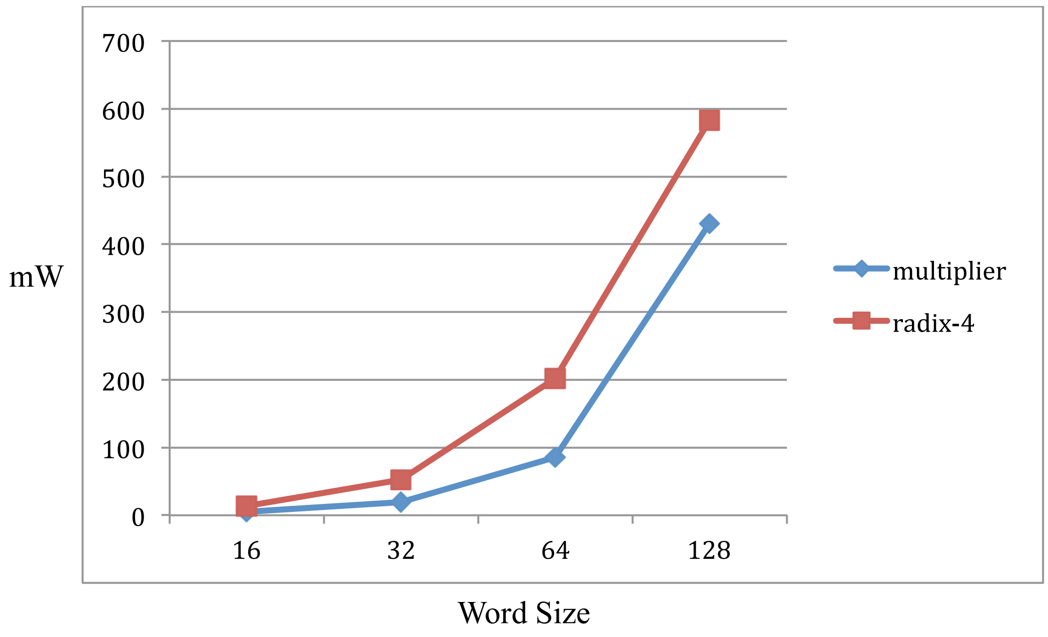

Figure 3.3.2.6: Power (mW) vs Word Size Chart

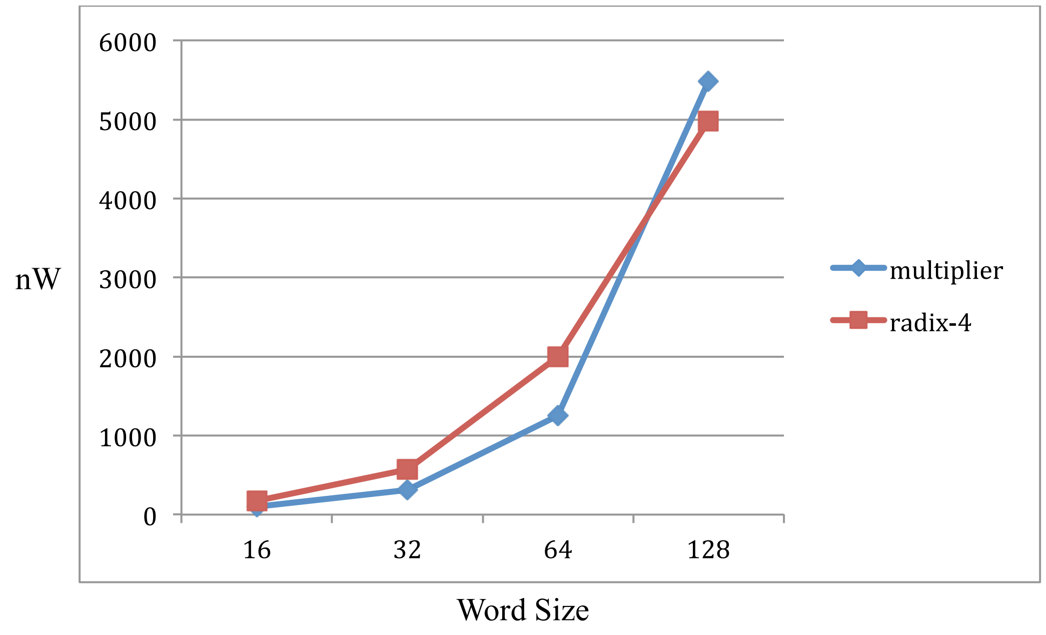

Figure 3.3.2.7: Leakage Power(nW) vs Word Size Chart

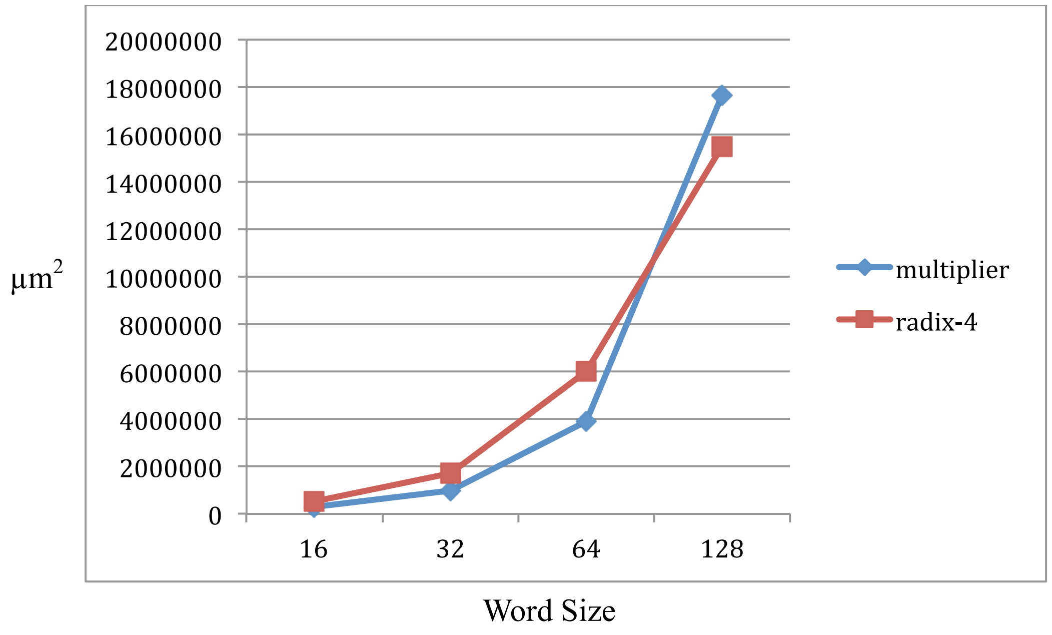

Figure 3.3.2.8: Area vs Word Size Chart

3.3.3. Signed Multiplication

3.3.3.1. Implementation

While the unsigned general purpose multiplier is useful, a signed multiplier would be even more useful. The initial thought of building a signed multiplier might be to just adding 2's complement circuits to the inputs and the outputs. While that configuration will work, it is not the optimal configuration.

Under further investigation, the truncation correction circuit described in 3.3.2.1 is no longer needed. Now that the most significant bit represents the sign of the number, the sum of the two (*n*-1)-bit numbers will at most take *n*-bits to store. Since now the sum and the difference can both be store in *n*-bits, the left shifting can be move to the last operation performed on the data. The new configuration can be shown in Figure 3.3.3.1.

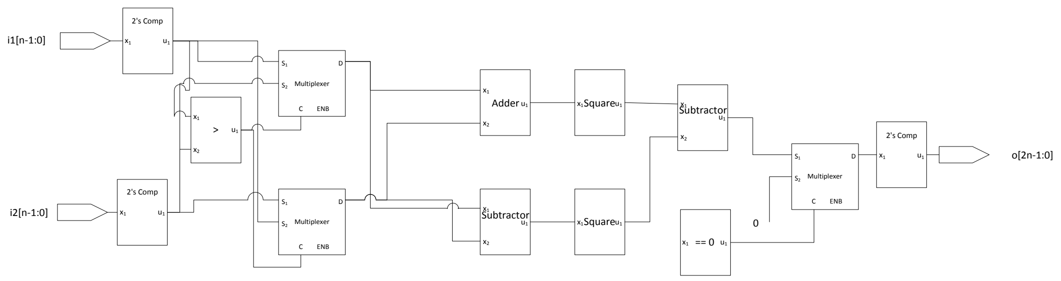

Figure 3.3.3.1: Signed Multiplication Data Path

3.3.3.2. Results

The signed general purpose integer multiplier created from radix-4 dual recoded squaring circuits and a signed general purpose multiplier were both implemented in Verilog and mapped to OSU standard cell library [5] for a n-bits of the result. Both circuits were constrained to run with-in a 100ns clock-edge and were implemented for 16, 32, and 64 bit-widths. The charts in Figures 3.3.3.2 - 3.3.3.4 show a loss in power but gains in leakage power and area for our customized radix-4 based signed multiplier compared to a signed multiplier circuit.

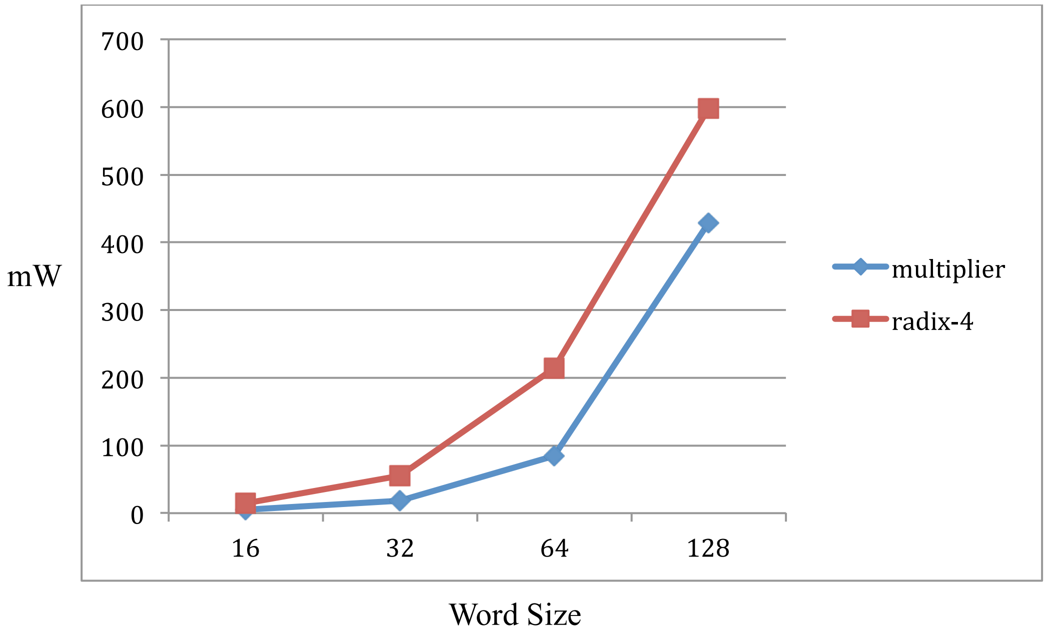

Figure 3.3.3.2: Power (mW) vs Word Size Chart

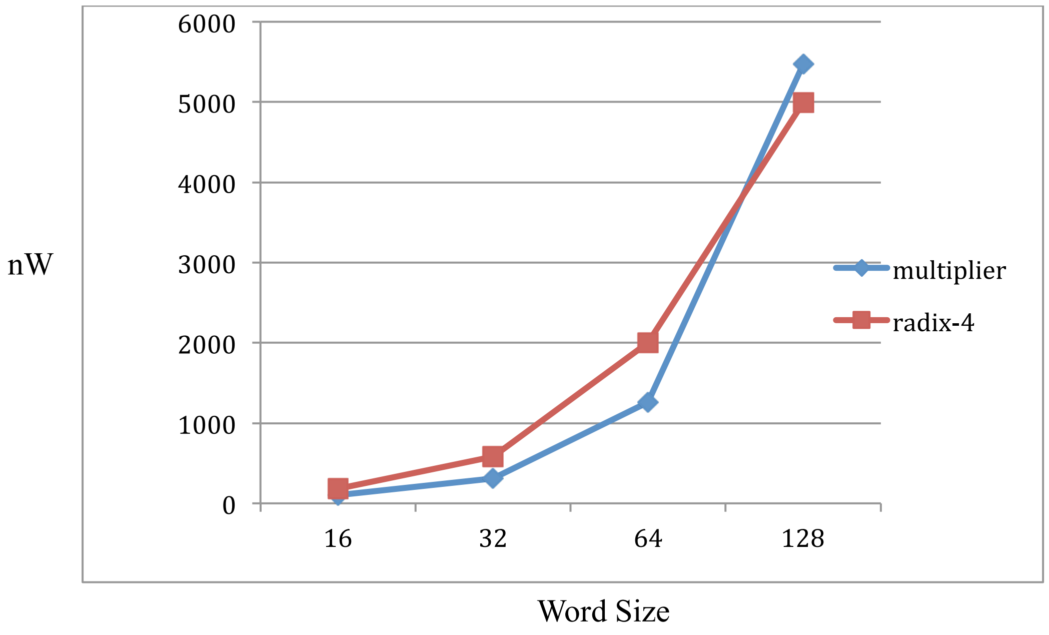

Figure 3.3.3.3: Leakage Power(nW) vs Word Size Chart

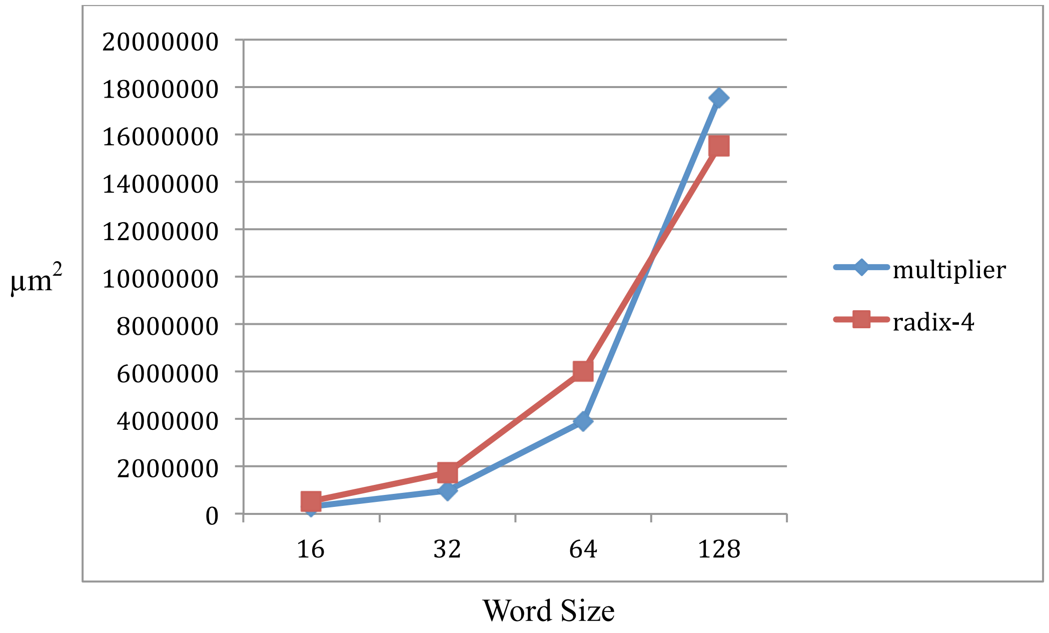

Figure 3.3.3.4: Area vs Word Size Chart

Chapter 4

CONCLUSION

Throughout this paper, we have presented several uses for a dual recoded radix-4 squaring circuit. Although the squaring circuits can be used to build general purpose multipliers, the circuits biggest advantages is as a specialized squaring circuit. When used for calculating the square, improvements are made in area, power, and leakage power. This holds true for the *n*-bit integer square, 2*n*-bit integer square and floating point square. The biggest improvements were seen in the implementations of *n*-bit integer square and floating point square. The improvements in the floating point square are due not only to replacing the multiplier circuit with a dual recoded 2*n*-bit squaring circuit, but also reducing the size of the preprocessor, pre-normalized, special case detector, and exponent adder.

While the multiplier made from the integer square showed a loss in total power and only minimal gains in area and leakage power, the circuit does have value. This designs value is for a circuit that needs to calculate square often and rarely needs the multiplication operation while not having the area for both a dual recoded radix-4 square and a traditional multiplier.

The floating point Booth dual recoded square and a general purpose floating point multiplier could coexist on the same circuit with only a small increase in area. The most obvious way to do this is for the two circuits is to add a MUX to share the normalizer, shifter, rounder, flagger, and assembler. This configuration is shown in Figure 4.1.

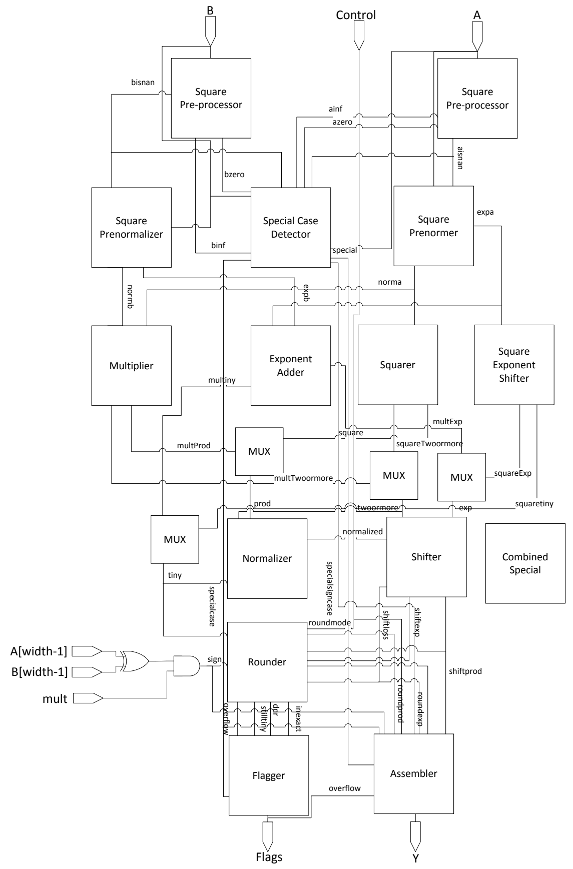

Figure 4.1: Multiplier/Squarer Combination Circuit [23]

Appendix A

AUTOMATIC TESTBENCH GENERATION

```
/**
 * @class VerilogTest
 * @author Jason Moore
 * @description Produces a testbench Verilog file for testing the squaring and multiplier
cicuits
 */

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;

public class VerilogTest {
	private static String[] includes16 = { "fullpps16_1.v", "fullpps16_2.v",
			"fullpps16_3.v", "fullpps16_4.v", "fullpps16_5.v",
			"fullpps16_6.v", "fullpps16_7.v", "fullpps16_8.v",
			"signExt16.v", "fullRadix4_16.v" };
	private static String[] includes32 = { "fullpps32_1.v", "fullpps32_2.v",
			"fullpps32_3.v", "fullpps32_4.v", "fullpps32_5.v",
			"fullpps32_6.v", "fullpps32_7.v", "fullpps32_8.v",
			"fullpps32_9.v", "fullpps32_10.v", "fullpps32_11.v",
			"fullpps16_3.v", "fullpps16_4.v", "fullpps16_5.v",
			"fullpps16_7.v", "fullpps16_8.v", "signExt32.v",
			 "fullRadix4_32.v" };
	private static String[] includes64 = { "fullpps16_2.v", "fullpps16_3.v",
			"fullpps16_4.v", "fullpps16_5.v", "fullpps16_7.v",
			"fullpps16_8.v", "fullpps32_3.v", "fullpps32_4.v",
			"fullpps32_5.v", "fullpps32_6.v", "fullpps32_7.v",
			"fullpps32_8.v", "fullpps32_9.v", "fullpps32_11.v",
			"fullpps64_1.v", "fullpps64_2.v", "fullpps64_3.v",
			"fullpps64_4.v", "fullpps64_5.v", "fullpps64_6.v",
			"fullpps64_7.v", "fullpps64_8.v", "fullpps64_9.v",
			"fullpps64_10.v", "fullpps64_11.v", "fullpps64_12.v",
			"fullpps64_13.v", "fullpps64_14.v", "fullpps64_15.v",
                    "fullpps64_16.v", "fullpps64_17.v", "fullpps64_18.v",
                    "signExt64.v", "fullRadix4_64.v" };
    private static String[] includes16m = { "pps16_1.v", "pps16_2.v",
                    "pps16_3.v", "pps16_4.v", "radix4_16.v", "squareM2_16.v" };
    private static String[] includes32m = { "pps16_2.v", "pps16_3.v",
                    "pps16_4.v", "pps32_1.v", "pps32_2.v", "pps32_3.v",
                    "pps32_4.v", "pps32_5.v", "radix4_32.v", "squareM2_32.v" };
    private static String[] includes64m = { "pps16_2.v", "pps16_3.v",
                    "pps16_4.v", "radix4_16.v", "pps32_2.v", "pps32_3.v",
                    "pps32_4.v", "pps32_5.v", "pps64_1.v", "pps64_2.v",
                     "pps64_3.v", "pps64_4.v", "pps64_5.v", "pps64_6.v",
                     "pps64_7.v", "pps64_8.v", "pps64_9.v", "squareM2_32.v" };
    private static String[] includes16sm = { "pps16_1.v", "pps16_2.v",
                    "pps16_3.v", "pps16_4.v", "radix4_16.v", "signedSquareM2_16.v"
                    };
    private static String[] includes32sm = { "pps16_2.v", "pps16_3.v",
                    "pps16_4.v", "pps32_1.v", "pps32_2.v", "pps32_3.v",
                    "pps32_4.v", "pps32_5.v", "radix4_32.v", "signedSquareM2_32.v"
                    };
    private static String[] includes64sm = { "pps16_2.v", "pps16_3.v",
                    "pps16_4.v", "radix4_16.v", "pps32_2.v", "pps32_3.v",
                    "pps32_4.v", "pps32_5.v", "pps64_1.v", "pps64_2.v",
                     "pps64_3.v", "pps64_4.v", "pps64_5.v", "pps64_6.v",
                     "pps64_7.v", "pps64_8.v", "pps64_9.v", "signedSquareM2_64.v"
                    };
    private static String[] includes24 = { "fullpps32_6.v", "fullpps32_7.v",
                    "fullpps32_8.v", "fullpps32_9.v", "fullpps32_11.v",
                    "fullpps16_2.v", "fullpps16_3.v", "fullpps16_5.v",
                    "fullpps16_7.v", "fullpps16_8.v", "fullpps24_1.v",
                     "fullpps24_2.v", "signExt24.v", "fullRadix4_24.v" };
    private static String[] includes48 = {"fullpps16_2.v", "fullpps16_3.v",
                    "fullpps16_4.v", "fullpps16_5.v", "fullpps16_7.v",
                    "fullpps16_8.v", "fullpps32_2.v", "fullpps32_3.v",
                    "fullpps54_2.v", "fullpps32_5.v", "fullpps32_6.v",
                    "fullpps32_7.v", "fullpps32_8.v", "fullpps32_9.v",
                    "fullpps32_11.v", "fullpps54_1.v","fullpps64_7.v",
                    "fullpps64_8.v", "fullpps64_9.v", "fullpps64_10.v",
                    "fullpps64_11.v", "fullpps64_12.v", "fullpps64_13.v",
                    "fullpps64_14.v", "fullpps64_15.v", "fullpps64_16.v",
                    "fullpps64_17.v", "signExt54.v", "fullRadix4_54.v", "assemble.v",
                    "exponent2.v", "flag.v", "normalize.v", "prenorm2.v",
                    "preprocess2.v", "round.v","shift.v", "special2.v", "fpmul2.v"};
/**
 * @method addIncludes
* @param outputStream
* @param bitWidth
* @param multi
* @throws IOException
* @description: Writes the needed includes
*/
        private static void addIncludes(BufferedWriter outputStream, int bitWidth,
                        int multi) throws IOException {
                String[] includes;
                if (multi == 1) {
                        switch (bitWidth) {
                        case 16:
                                includes = includes16m;
                                break;
                        case 32:
                                includes = includes32m;
                                break;
                        case 64:
                                includes = includes64m;
                                break;
                        default:
                                return;
                        }
                } else if (multi == 2) {
                        switch (bitWidth) {
                        case 16:
                                includes = includes16sm;
                                break;
                        case 32:
                                includes = includes32sm;
                                break;
                        case 64:
                                includes = includes64sm;
                                break;
                        default:
                                return;
                        }
                } else if (multi == 3) {
                        if (bitWidth == 32) {
                                includes = includes24;
                        } else {
                                includes = includes48;
                        }
```

```
		} else {
			switch (bitWidth) {
			case 16:
				includes = includes16;
				break;
			case 32:
				includes = includes32;
				break;
			case 64:
				includes = includes64;
				break;
			default:
				return;
			}
		}
		for (int i = 0; i < includes.length; i++) {
			outputStream.write("`include " + "\"" + includes[i] + "\"");
			outputStream.newLine();
		}
	}

	/**
	 * @method changeValues
	 * @param outputStream
	 * @param bitWidth
	 * @param multi
	 * @throws IOException
	 * @description 2^16 random values are written to the Verilog file
	 */
	private static void changeValues(BufferedWriter outputStream, int bitWidth,
			int multi) throws IOException {
		Random randomGenerator = new Random();
		double numOfValues = 65536.0;
		int values = (int)Math.pow(2, bitWidth);
		//Multiplication
		if (multi == 1) {
			for (double i = 0.0; i < numOfValues; i++) {
				int randomInt1 = randomGenerator.nextInt(new
							Double(values).intValue());
				int randomInt2 = randomGenerator.nextInt(new
							Double(values).intValue());
				outputStream.write("\t\t\ti=" + randomInt1 + ";");
				outputStream.write("\t\t\ti2=" + randomInt2 + ";");
				outputStream.newLine();
```

```
                outputStream.write("\t\t\t#100");
            }
        } else if (multi == 2) { //Signed Multiplication
            for (double i = 0.0; i < numOfValues; i++) {
                int randomInt1 = randomGenerator.nextInt(new
                                Double(values).intValue()) –
                                    (values >> 1);
                int randomInt2 = randomGenerator.nextInt(new
                                Double(values).intValue()) –
                                    (values >> 1);
                outputStream.write("\t\t\ti=" + randomInt1 + ";");
                outputStream.write("\t\t\ti2=" + randomInt2 + ";");
                outputStream.newLine();
                outputStream.write("\t\t\t#100");
            }
        } else if (multi == 3) { //Floating Point Square
            for (int i = 0; i < numOfValues; i++) {
                int control = 3;
                int exp;
                if (bitWidth == 32) { //Single
                    exp = (i%256) << 23;
                    values = (int) Math.pow(2, 23);
                    long randomInt = randomGenerator.nextInt(new
                                Double(values).intValue()) + exp;
                    outputStream.write("\t\t\ti=" + randomInt + ";");
                } else { // Double
                    exp = (i%1024) << 52;
                    values = (int) Math.pow(2, 52);
                    long randomInt = randomGenerator.nextInt(new
                                Double(values).intValue()) + exp;
                    outputStream.write("\t\t\ti=64'd" + randomInt +";");
                }

                if (i < numOfValues/4) {
                    control = 0;
                } else if (i < numOfValues/2) {
                    control = 1;
                } else if (i < (numOfValues * .75)) {
                    control = 2;
                }
                outputStream.write(" control=" + control + ";");
                outputStream.newLine();
                outputStream.write("\t\t\t#100");
            }
```

```
		}
		//Square
		else {
			for (double i = 0.0; i < numOfValues; i++) {
				int randomInt = randomGenerator.nextInt(new
							Double(values).intValue());
				outputStream.write("\t\t\ti=" + randomInt + ";");
				outputStream.newLine();
				outputStream.write("\t\t\t#100");
			}
		}
	}

	public static void main(String[] args) {
		try {
			int multi = 0;
			BufferedWriter outputStream = new BufferedWriter(new
					FileWriter(“simcrct" + args[0] + args[1] + ".v"));
			int bitWidth = Integer.parseInt(args[1]);
			if (args[2].toLowerCase().equals("multiply")) {
				multi = 1;
			} else if (args[2].toLowerCase().equals("signed")) {
				multi = 2;
			} else if (args[2].toLowerCase().equals("floating")) {
				multi = 3;
			}
			addIncludes(outputStream, bitWidth, multi);
			outputStream.write("module stimcrct" + args[1] + ";");
			outputStream.newLine();
			outputStream.write("\treg [" + (bitWidth - 1) + ":0] i;");
			outputStream.newLine();
			//Initial Verilog setup
			if (multi > 0 && multi < 3) {
				outputStream.write("\treg [" + (bitWidth - 1) + ":0] i2;");
				outputStream.newLine();
				outputStream.write("\twire ["+(bitWidth* 2 - 1) + ":0] o;");
				outputStream.newLine();
				outputStream.write("\t" + args[0] + "_" + args[1] +
							" crct(i,i2, o);");
				outputStream.newLine();
				outputStream.write("\tinitial");
				outputStream.newLine();
				outputStream.write("\t\tbegin");
				outputStream.newLine();
```

```
                        outputStream.write("$monitor (\"square = %d\", o);");
                } else if (multi == 3) {
                        outputStream.write("\treg [4:0] control;");
                        outputStream.newLine();
                        outputStream.write("\twire [" + (bitWidth - 1) + ":0] o;");
                        outputStream.newLine();
                        outputStream.write("\twire [4:0] flag;");
                        outputStream.newLine();
                        outputStream.write("\t fpmul2 crct (i, o, control, flag);");
                        outputStream.newLine();
                        outputStream.write("\tinitial");
                        outputStream.newLine();
                        outputStream.write("\t\tbegin");
                        outputStream.newLine();
                        outputStream.write("$monitor (\"square = %d flag = %d\",
                                                o, flag);");
                } else {
                        outputStream.write("\twire [" +(bitWidth * 2 - 1)+":0] o;");
                        outputStream.newLine();
                        outputStream.write("\t" + args[0] + "_" + args[1] +
                                                " crct (i,o);");
                        outputStream.newLine();
                        outputStream.write("\tinitial");
                        outputStream.newLine();
                        outputStream.write("\t\tbegin");
                        outputStream.newLine();
                        outputStream.write("$monitor (\"square = %d\", o);");
                }
                outputStream.newLine();
                changeValues(outputStream, bitWidth, multi);
                outputStream.write("\t\t\t$finish;");
                outputStream.newLine();
                outputStream.write("\t\tend");
                outputStream.newLine();
                outputStream.write("endmodule");
                outputStream.flush();
                outputStream.close();
        } catch (IOException e) {
                e.printStackTrace();
        }

    }

}
```

Appendix B

VERILOG FILES

B.1 PARTIAL SQUARE GENERATOR

```
/*
* A 2X1 13-bit MUX that sets z to i2 if c is 1
* and to i2 otherwise.
* Inputs: i1 and i2
* Control: c
* Output: z
*/
module mux21_13 (i1, i2, c, z);
        input[12:0] i1, i2;
        input c;
        output[12:0] z;
        reg[12:0] z;

        always @ (c or i1 or i2)
        begin
                if (c)
                begin
                        z = i2;
                end
                else
                begin
                        z = i1;
                end
        end
endmodule

/*
* Left shifts a 13-bit input i
* one place to the left for a
* 13-bit output o.
* Input: i
* Output: o
*/
module lshift13_1 (i, o);
        input[12:0] i;
        output[12:0] o;
        reg [12:0] o;

        always @ (i)
        begin
                o = i << 1;
        end
endmodule

/*
* Negates the 13-bit input i if
* c is 1.
* Input: i
* Output: o
*/
module codNeg13 (i, c, o);
        input[12:0] i;
        input c;
        output[12:0] o;
        reg[12:0] o;
        integer j;

        always @ (i or c)
        begin
                for (j = 0; j < 13; j = j + 1)
                begin
                        o[j] = c ^ i[j];
                end
        end

endmodule

/*
 * pps16_1
 * Calculates a 16-bit partial square from a
 * 15-bit input.
 * Input: x
 * Output: y
 */
module pps16_1 (x, y);
        input [14:0] x;
        output [15:0] y;
        wire [15:0] y;
        wire c1, c2, c3, c4;
        wire [12:0] t1, t2, t3, t4;
        xnor xn1 (c1, 0, x[0]);
        xnor xn2 (c2, x[0],x[1]);
        and a1 (c3, c2, c1);
        not n1 (c4, c3);
        and a2 (y[2], c4, c1);
        assign y[1] = 0;
        nor no1 (y[0], c1, c3);
        codNeg13 cn1 (x[14:2], x[1], t1);
        lshift13_1 ls1 (t1, t2);
        mux21_13 m1 (t1, t2,  c1, t3);
        assign t4 = 0;
        mux21_13 m2 (t3, t4, c3, y[15:3]);
endmodule
```

B.2 INTERGER MULTIPLIER

```
/* Calculates the 16-bit square
 * from a 16-bit input
 * Input: i
 * Output: o
 */
module radix4_16 (i, o);
        input[15:0] i;
        output[15:0] o;
        reg[15:0] o;
        wire[15:0] t1;
        wire[11:0] t2;
        wire[7:0] t3;
        wire[3:0] t4;
        reg[11:0] t5;
        reg[7:0] t6;

        // Calculate partial squares
        pps16_1 p1 (i[14:0],t1);
        pps16_2 p2 (i[12:1], t2);
        pps16_3 p3 (i[10:3], t3);
        pps16_4 p4 (i[8:5], t4);
        // Added partial squares together
        always @ (t1 or t2 or t3 or t4)
        begin
                o[3:0] = t1[3:0];
                t5 = t1[15:4] + t2;
                o[7:4] = t5[3:0];
                t6[3:0] = t3[3:0];
                t6[7:4] = t3[7:4] + t4;
                o[15:8] = t6 + t5[11:4];
        end

endmodule
```

B.3 FULL PARTIAL SQUARE GENERATOR

```
/*
* A 2X1 15-bit MUX that sets z to i2 if c is 1
* and to i2 otherwise.
* Inputs: i1 and i2
* Control: c
* Output: z
*/
module mux21_15 (i1, i2, c, z);
        input[14:0] i1, i2;
        input c;
        output[14:0] z;
        reg[14:0] z;

        always @ (c or i1 or i2)
        begin
                if (c)
                begin
                        z = i2;
                end
                else
                begin
                        z = i1;
                end
        end
endmodule

/*
* Left shifts a 15-bit input i
* one place to the left for a
* 15-bit output o.
* Input: i
* Output: o
*/
module lshift15_1 (i, o);
        input[14:0] i;
        output[14:0] o;
        reg [14:0] o;

        always @ (i)
        begin
```

```
                o = i << 1;
        end
endmodule

/*
* Negates the 14-bit input i if
* c is 1.
* Input: i
* Output: o
*/
module codNeg15 (i, c, o);
        input[13:0] i;
        input c;
        output[14:0] o;
        reg[14:0] o;
        integer j;

        always @ (i or c)
        begin
                for (j = 0; j < 14; j = j+ 1)
                begin
                        o[j] = c ^ i[j];
                end
                o[14] = c ^ 0;
        end
endmodule

/*
 * codZero15
 * Sets the 15-bit output to 0
 * if c is 0 otherwise the output
 * is set to the input.
 * Input: i, c
 * Output: o
 */
module codZero15 (i, c, o);
        input[14:0] i;
        input c;
        output[14:0] o;
        reg[14:0] o;
        integer j;

        always @ (i or c)
        begin
                for (j = 0; j < 15; j = j + 1)
                begin
```

```
                    o[j] = c & i[j];
                end
        end
endmodule

/*
 * fullpps16_1
 * Calculates a 18-bit partial square
 * from a 16-bit input.
 * Input: x
 * Output: y
 */
module fullpps16_1 (x,y);
        input[15:0] x;
        output[17:0] y;
        wire[17:0] y;
        wire c1, c2, c3, c4;
        wire[14:0] t1, t2, t3;

        xnor xn1 (c1, 0, x[0]);
        xnor xn2 (c2, x[0],x[1]);
        nand n1 (c4, c2, c1);
        not n2 (c3, c4);
        and a2 (y[2], c4, c1);
        assign y[1] = 0;
        nor no1 (y[0], c1, c3);
        codNeg15 cn1 (x[15:2], x[1], t1);
        lshift15_1 ls1 (t1, t2);
        mux21_15 m1 (t1, t2, c1, t3);
        codZero15 m2 (t3, c4, y[17:3]);
endmodule
```

B.4 FULL DUAL RECODED RADIX-4 SQUARE

```
/* Calculates the 32-bit square of
 * a 16-bit input input
 * Input: i
 * Output: o
 */
module fullRadix4_16 (i,o);
        input[15:0] i;
        output[31:0] o;
        reg[31:0] o;
        wire[31:0]c1;
        wire[24:0] c2;
        wire[22:0] c3;
```

```
wire[19:0] c4;
wire[9:0] c5;
wire[7:0] c6;
wire[5:0] c7;
wire[3:0] c8;
wire[13:0] c9;
reg [27:0] c14;
reg [27:0] c16;
reg [24:0] c15;
reg [18:0] c10;
reg [19:0] c11;
reg [23:0] c12, c13;
wire x, y, z;

//Calculate partial squares
fullpps16_1 fp1 (i, c1[17:0]);
fullpps16_2 fp2 (i[15:1],c2[15:0]);
fullpps16_3 fp3 (i[15:3],c3[13:0]);
fullpps16_4 fp4 (i[15:5], c4[11:0]);
fullpps16_5 fp5 (i[15:7], c5);
fullpps16_6 fp6 (i[15:9], c2[23:16]);
fullpps16_7 fp7 (i[15:11], c3[21:16]);
fullpps16_8 fp8 (i[15:13], c4[19:16]);
signExt16 s1 (i[1], i[2], i[3],i[5],i[7],i[9],i[11],i[13], c1[31:18]);
and a1 (c3[14], i[5], i[4],i[3]);
and a2 (c4[12], i[5], i[6],i[7]);
and a3 (c4[14], i[7], i[8],i[9]);
and a4 (c2[24], i[9], i[10],i[11]);
and a5 (c3[22], i[11],i[12],i[13]);

// Line up bits
assign c3[15] = 0;
assign c4[13] = 0;
assign c4[15] = 0;

// Add partial squares
always @ (c1 or c2)
begin
        o[3:0] = c1[3:0];
        c16 = c1[31:4] + c2;
        o[7:4] = c16[3:0];
end

always @ (c3 or c4)
begin
```

```
                c12[3:0] = c3[3:0];
                c12[23:4] = c3[22:4] + c4;
        end

        always @ (c16 or c12)
        begin
                c13 = c12 + c16[27:4];
                o[15:8] = c13[7:0];
        end

        always @ (c5 or c13)
        begin
                o[31:16] = c13[23:8] + c5;
        end
endmodule
```

B.5 INTEGER MULTIPLCATION USING SQUARING UNITS

```
/* Calculates the 64-bit product
 * of two 64-bit inputs
 * Inputs: i1, i2
 * Output: o
 */
module squareM2_16 (i1, i2, o);
        input[15:0] i1, i2;
        output[15:0] o;
        reg[15:0] o;
        wire[15:0] t1, t2;
        reg[15:0] t3, t4;
        reg[16:0] t5, t6;
        reg[16:0] t7;
        reg[16:0] t8;
        // Add the two inputs
        // Subtract the smaller input
        // from the larger on
        always @ (i1 or i2)
        begin
                if (i1 > i2)
                begin
                        t3 = i1;
                        t4 = i2;
                end
                else
                begin
                        t3 = i2;
                        t4 = i1;
```

```
			end
			t5 = t3 - t4;
			t6 = t3 + t4;
		end
		// square the difference
		radix4_16 s1 (t5[16:1], t1);
		// square the sum
		radix4_16 s2 (t6[16:1], t2);
		// Truncation correction
		always @ (t2)
		begin
			if (t6[0] == 1)
			begin

				t7 = t2 + t6[16:1];
			end
			else
			begin
				t7[15:0] = t2;
				t7[16] = 0;
			end
		end

		always @ (t1)
		begin
			if (t5[0] == 1)
			begin

				t8 = t1 + t5[16:1];
			end
			else
			begin
				t8[15:0] = t1;
				t8[16] = 0;
			end
		end
		// Subtract the smaller square
		// from the larger one
		always @ (t7 or t8)
		begin
			o = (t7 - t8);
		end

	endmodule
```

REFERENCES

[1] A.D. Booth, “A Signed Binary Multiplication Technique,” *Quarterly J. Mechanical and Applied Math.*, vol. 4, no. 2, pp. 236-240, 1951.

[2] D.W. Matula, “A Radix-4 dual recoded Squaring Procedure For Use in Design of Application Specific Arithmetic Circuits”, SRC Report, 4/12/2007

[3] A. Pirson, J.-M. Bard, and M. Daoudi, Squaring Circuit for Binary Numbers, United States Patent, No. 5,629,885, May 13, 1997.

[4] L.P. Rubinfeld, “A Proof of the Modified Booth's Algorithm for Multiplication,” *IEEE Trans. Computers*, vol. 24, no. 10, pp. 1014-1015 Oct. 1975.

[5] J. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W.R. Davis, P.D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, R. Jenkal, FreePDK: An Open-Source Variation-Aware Design Kit. Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education, 2007.

[6] A. Strollo, E. Napoli, and D. Caro, “New Design of Squarer Circuits Using Booth Encoding and Folding Techniques”, Proceedings of the 8th ICECS, pp. 193-196, Sep. 02, 2001.

[7] K. Wires, W. Schulte, L. Marquette, and P. Balzola, “Combined Unsigned and Two’s Complement Squarers”, Proceedings of the Thirty Third

Asilomar Conference on Signals, Systems, and Computers, pp. 1215-1219., 1999

[8] F. Yul and A. Willson, Jr., “Multirate Digital Squarer Architectures”, Proceedings of the 8th ICECS, pp. 177-180, Sep. 02, 2001.

[9] A. Fit-Florea, D.W. Matula, and M.A. Thornton, “Additive Bit-serial Algorithm for the Discrete Logarithm Modulo 2^k”, *IEE Electronics Letters*, vol. 41, no. 2, January 2005, pp. 57-59.

[10] L. Li, A. Fit-Florea, M.A. Thornton, and D.W. Matula "Hardware Implementation of an Additive Bit-Serial Algorithm for the Discrete Logarithm Modulo 2^k ", *Proc. IEEE Ann. Sym. on VLSI*, May 2005, pp.130-135.

[11] L. Li, M.A. Thornton, and D.W. Matula, “A Digit Serial Algorithm for the Integer Power Operation”, *Proc. of ACM/IEEE Great Lakes Symposium on VLSI (GLSVLSI),* April 2006, pp. 302-307.

[12] P. M. Seidel, D.W. Matula, and L.D. McFearin, "Secondary Radix Recodings for Higher Radix Multipliers", *IEEE Trans. on Comp.*, vol 54, 2005, pp. 111-123.

[13] J. Moore, M.A. Thornton, and D.W. Matula, “A Low Power Radix-4 Dual Recoded Integer Squaring Implementation for use in Design of Application Specific Arithmetic Circuits”, *IEEE Asilomar Conference on Signals, Systems, and Computers (ASILOMAR)*, October 26-29, 2008, pp. 1819-1822.

[14] IEEE *Standard for Binary Floating-Point Arithmetic*. New York: ANSI/IEEE 754-1985, 1985.

[15] M. E. Phair, “Free Floating-Point Madness: Multiplier”, http://www.hmc.edu/chips/fpmul.html, May 14, 2002

[16] N. Quach, N. Takagi, and M. Flynn, "On Fast IEEE Rounding," Technical Report CSL-TR-91-459, Stanford University, January 1991.

[17] G. Even and P-M. Seidel, "A comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication." *IEEE Transactions on Computers*, Vol. 49 No. 7 July 2000.

[18] J. Stohmann and E. Barke, "A Universal Pezaris Array Multipler Generator for SRAM-Based FPGAs", *Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings., 1997 IEEE International Conference on*, pp.489,495, 12-15 Oct 1997

[19] A. D. Booth, "A signed binary multiplication technique", *Quart. J. Mech. Appl. Math.* ***4*** (1951), 236–240.

[20] J. Hensley, A. Lastra, and M. Singh, "An area- and energy-efficient asynchronous Booth multiplier for mobile devices," *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on* , vol., no., pp.18,25, 11-13 Oct. 2004

[21] A. S. Prabhu, and V. Elakya, "Design of modified low power booth multiplier," *Computing, Communication and Applications (ICCCA), 2012 International Conference on* , vol., no., pp.1,6, 22-24 Feb. 2012

[22] S. Chen, C. Liu, T. Wu, and A. Tsai, "Design and Implementation of High-Speed and Energy-Efficient Variable-Latency Speculating Booth Multiplier (VLSBM)," *IEEE Trans. On Ciruits and Systems 60-I*, pp. 2631-2643, Oct. 2013

[23] J. Moore, M. Thornton, and D.W. Matula, Low Power Floating-Point Multiplication and Squaring Units with Shared Circuitry, *IEEE Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 1395-1398, August 4, 2013

[24] L. Dadda, "Some schemes for parallel multipliers". *Alta Frequenza* 34: 349–356, March 1965

[25] B. Jeevan, S. Narender, C.V.K. Reddy, and K. Sivani, "A High Speed binary Floating Point Multiplier Using Dadda Algorithm", *Automation, Computing, Communication, Control and Compressed Sensing (iMac4s), 2013 International Multi-Conference on* , vol., no., pp.455,460, 22-23 March 2013

[26] H. Eriksson, P. Larsson-Edefors, M. Sheeran, M. Sjalander, D. Johansson, and M. Scholin, "Multiplier Reduction Tree with Logarithmic Logic Depth and Regular Connectivity", *2006 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 4-8, May 2006

[27] C.S. Wallace, "A Suggestion for a Fast Multiplier", *IEEE Transactions on Electronic Computers Vol: EC-13 Issue: 1*, pp. 14-17, Feb. 1964

[28] S. Rajaram and K. Vanithamani, "Improvement of Wallace Multipliers using Parallel Prefix Adders", *2011 International Conference on Signal Processing, Communication, Computing, and Networking Technologies (ICSCCN)*, pp. 781-784, July 2011

[29] M.E. Robinson and E. Swartzlander Jr., "A Reduction Scheme to Optimize the Wallace Multiplier", *International Conference on Computer Design: VLSI in Computers and Processors 1998 (ICCD '98)*, pp. 122-127, Oct. 1998

[30] N. Sureka, R. Porselvi, and K. Kumuthapriya, "An Efficient High Speed Wallace Tree Multiplier", *2013 International Conference on Information Communication and Embedded Systems (ICICES)*, pp. 1023-1026, Feb. 2013

[31] W. Dong-Hui, "Scan Test in 18x18 bits Booth Coding-Wallace Tree Multiplier", *4th International Conference on ASIC 2001*, pp. 624-627, Oct. 2001