Binary Floating Point Fused Multiply Add Unit

by

Eng. Walaa Abd El Aziz Ibrahim

A Thesis Submitted to the

Faculty of engineering at Cairo University

in partial Fulfillment of the

Requirement for the Degree of

MASTER OF SCIENCE

in

ELECTRONICS AND COMMUNICATIONS ENGINEERING

FACULTY OF ENGINEERING, CAIRO UNIVERSITY

GIZA, EGYPT

2012

Binary Floating Point Fused Multiply Add Unit

by

Eng. Walaa Abd El Aziz Ibrahim

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Under the Supervision of

Ahmed Hussien Khalil

Associate Professor
Electronics and Communications
Engineering
Cairo University

Hossam Aly Fahmy

Associate Professor
Electronics and Communications
Engineering
Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY

GIZA, EGYPT

2012

Acknowledgment

First, I want to thank God for helping me in finishing this work. I want also to thank my supervisors Dr. Ahmed Hussein and Dr. Hossam Fahmy for giving me information, support and experience which I need in research and writing this thesis.

I would like also to thank Eng. Ahmed el Shaf3y, Eng. Mervat and Eng. Ahmed Mohee for their co-operation with me and a specially thank to Eng. Amr Abd el-Fatah for helping me in testing my designs.

Finally, I want to thank my parent, sisters, fiancé and friends for giving me support and confidence.

ABSTRACT

Floating-point unit is an integral part of any modern microprocessor. The fused multiply add (FMA) operation is very important in many scientific and engineering applications. It is a key feature of the floating-point unit (FPU), which greatly increases the floating-point performance and accuracy. With the recent advancement in FPGA architecture and area density, latency has been the main focus of attention in order to improve performance. Many floating-point fused multiply add algorithms are developed to reduce the overall latency. The greatest deviation from the original IBM RS/6000 architecture comes from a paper by T. Lang and J.D. Bruguera on a reduced latency fused multiplier-adder. The paper claims an estimated 15-20% reduction in latency as compared to a standard fused multiply add. This result is calculated theoretically, and the actual architecture has yet to be implemented in either a synthesized or a custom CMOS silicon design.

The main objective of our work is to implement this algorithm but with little change to facilitate the implementation and on the other hand do not affect the performance. The implementation includes full design of blocks are not included before in Lang/Bruguera algorithm like sign detection module. Both the proposed and the basic architecture are designed using the Verilog hardware description language and then synthesized, placed and routed for Cyclone II FPGA device using Quartus II 9.1. The proposed architecture achieves a delay improvement about 25.5% as compared to the basic architecture. The increase of area in the proposed architecture is about 6.2%.

Contents

List of Figures

List of Tables

List of Abbreviations

ARM	Application Response Measurement
CDC	Control Data Corporation
CPA	Carry Propagate Adder
CSA	Carry Save Adder
DEC	Digital Equipment corporation
DSP	Digital Signal Processing
DUT	Design Under Test
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FMA	Fused Multiply Add
FPGA	Field Programmable Gate Arrays
FPU	Floating-Point Unit
HA	Half Adder
HP	Hewlett-Packard
IBM	International Business Machine
RS/6000	Reduced instruction set computing System 6000
IEEE	Institute of Electrical and Electronics Engineers
LSB	Least Significant Bit
LZA	Leading Zero Anticipator

LZC	Leading Zero Counter
LZD	Leading Zero Detector
MIPS	Microprocessor without Interlocked Pipeline Stages
MSB	Most Significant Bit
NaN	Not a Number
RI	Round to Infinity
Rn	Round towards +∞
RN	Rounding to Nearest even
RP	Round towards -∞
RZ	Round towards 0
SRAM	Static Random Access Memory
VLSI	Very Large Scale Integrated

Chapter 1 : Introduction

Floating-point unit (FPU) is one of the most important custom applications needed in most hardware designs as it adds accuracy and ease of use. Recently, the floating-point units of several commercial processors like IBM PowerPC, Intel/HP Itanium, MIPS-compatible Loongson-2F and HP PA-8000 [8], [24] have included a floating-point fused multiply add (FMA) unit to execute the double-precision fused multiply add operation $A+(B\times C)$ as an indivisible operation, with no intermediate rounding.

The FMA operation is very important in many scientific and engineering applications like digital signal processing (DSP), Finite impulse response (FIR) filters, graphics processing, fast Fourier transform (FFTs) [4], [11] division and argument reduction [20], [28]. The first FMA is introduced in 1990 by IBM RS/6000 [12], [13]. After that FMA is implemented by several companies like HP, MIPS, ARM and Intel. It is a key feature of the floating-point unit because it greatly increases the floating-point performance and accuracy since rounding is performed only once for the result $A+(B\times C)$ rather than twice for the multiplier and then for the adder. It also realizes reduction in the latency and hardware cost. FMA can be used instead of floating-point addition and floating-point multiplication by using constants e.g., $0.0\ +\ (B\ \times C)$ for multiplication and $A+\ (B\ \times\ 1.0)$ for addition.

A Field Programmable Gate Array, FPGA, provides a versatile and inexpensive way to implement and test VLSI designs. It is mostly used in low volume applications that cannot afford silicon fabrication or designs which require frequent changes or upgrades.

In FPGAs, the bottleneck for designing efficient floating-point units has mostly been area. With advancement in FPGA architecture, there is a significant increase in FPGA densities so latency has been the main focus of attention in order to improve performance.

1.1 Motivation and Contribution

Floating-point fused multiply add unit is one of the most important blocks that exist in floating-point unit as it increases its accuracy and performance. It is useful in many computations which involve the accumulation of products such as scientific and engineering applications. Many algorithms are developed on floating-point fused multiply add unit to decrease its latency [2], [10], [18], [25], [31]. The biggest deviation from the original IBM RS/6000 architecture comes from a paper by T. Lang and J.D. Bruguera [10] on a reduced latency fused multiply add units. However, to the best of our knowledge, Lang/ Bruguera algorithm is not implemented till now.

The main contribution and objective of our work is to implement the architecture which is proposed by Lang/Bruguera but with little change to facilitate the implementation. This thesis also shows the full design of some blocks which are not included in Lang/Bruguera work like sign detection module. This algorithm and the basic algorithm are implemented in the Verilog hardware description language, and then are synthesized, placed and routed for Cyclone II FPGA device using Quartus II 9.1. Area and timing information for each design approach is reported and analyzed.

1.2 Outline

This thesis is structured as follows. Chapter 2 gives an overview of IEEE floating-point standard and brief information on fused multiply-add unit.

Chapter 3 shows previous architectures of the fused multiply add and the design details of the components used to implement them. Chapter 4 shows the proposed fused multiply add fused unit and the design details of its components. Chapter 5 goes over the testing procedure, simulation, and results. Chapter 6 concludes the thesis and provides recommendations for further research.

Chapter 2 : Standard Floating Point Representations and Fused Multiply-Add Algorithm

2.1 Fraction representation

Beside integer's representation, there is a need to represent fractions. A radix point is used to divide the number into two parts, an integer part (on the left to the radix point) with using positive powers and a fraction part (on the right to radix point) with using negative powers. As an example, the binary number $(10.01)_2$ is interpreted to $(1x2^1 + 0x2^0 + 0x2^{-1} + 1x2^{-2})$ which equals to (2.25) in decimal. However an exact representation of all fractions is impossible in binary notation. For example consider the number 1/3, it can be written as a decimal by (0.333333…..) for a finite number assigned for the fractional part the representation is not exact. Fixed and floating point representations are used to handle fractions in computers.

2. 1.1 Fixed point representation

Fixed point representation is a simple and easy way to express fractional numbers using a fixed number of bits. The precision of a fixed point number is the total number of bits for the fractional part of the number. The location of the radix point is variable and non-compulsory so the precision of the fixed point can be any number up to and including all bits used. The dynamic range of fixed point is the ratio of the largest representable number to the smallest non-zero representable number. Accordingly, The more precision the less dynamic range we have.

Fixed point representation suffers from lack of dynamic range where very large and very small numbers cannot be represented together in the same representation.

2. 1.2 Floating point representation

Floating-point representation basically represents real numbers in scientific notation. Scientific notation represents numbers as a base number and an exponent. In general, a floating-point number consists of three main parts: sign (S), mantissa (M) and exponent (E). Its value can be given by $(-1)^{\mathrm{s}} \times \mathrm{M} \times \mathrm{base}^{\mathrm{E}}$ the base is one of the most important aspects needed to know when using floating point representation. It is equal 2 for binary, 10 for decimal and 16 for hexadecimal numbers.

The main feature of floating point representation is its large dynamic range. This is because the largest representable number is approximately equal to the base raised to the power of maximum positive exponent and the smallest non-zero number is approximately equal to the base raised to the power of the maximum negative exponent.

The precision is another important factor of floating-point representation. It equals the number of bits used to represent the mantissa (significand). It determines how close two adjacent floating point numbers can be.

2.2 IEEE floating point representation

A floating point system defines the base of the number system, the location of the fractional point , precision and whether the numbers are normalized or not. Many formats appeared to specify these issues such as IBM, CDC and DEC formats [34]. These formats suffered from compatibility and conversion problems.

Thus there is a need to standardize the formats Regardless of manufacturers and programming languages. The IEEE standard is developed for this need.

IEEE 754 is an IEEE Standard for Floating-Point Arithmetic which is developed for binary floating point arithmetic in 1985. In 1987 a second complementary standard (IEEE 854) is developed for radix independent floating point arithmetic. The current version is (IEEE 754–2008) which appeared in 2008. It is a revised version which combines two previous standards in addition to the inclusion of decimal floating point and its operations.

The standard specifies basic and extended floating-point number formats, arithmetic operations, conversions between various number formats, rounding algorithms, and floating-point exceptions. The standard also includes extensive recommendations for advanced exception handling.

Formats in IEEE standards describe sets of floating-point data and encodings for interchanging them. A given format comprises finite numbers, including its base and three fields needed to specify sign, significand and exponent, Two infinities +∞ and −∞ and Two kinds of Not a Number (NaN) quiet and signaling.

2.2.1 Basic formats

The standard defines five basic formats, see Table (2-1), the first three formats named single, double and quadruple precision basic formats are used to encode binary floating point numbers and use 32, 64 and 128 bits respectively. The last two formats are used for decimal floating point numbers and use 64 and 128 bits to encode them.

All the basic formats may be available in both hardware and software implementations.

This thesis concerns only on binary floating point with double precision format, so it will be discussed in more details.

Table 2-1: five basic formats of IEEE standard

Name	Common name	Base	Digits	E $_{min}$	E $_{max}$
binary 32	Single precision	2	23+1	-126	+127
binary 64	Double precision	2	52+1	-1022	+1023
binary 128	Quadruple precision	2	112+1	-16382	+16383
decimal 64		10	16	-383	+384
decimal 128		10	34	-6143	+6144

Double precision format

Double -precision format uses 1-bit for sign bit, 11-bits for bias exponent and 52-bits to represent the fraction as shown in Figure (2-1).

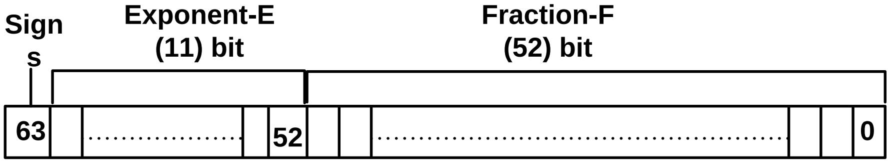


Figure 2-1: IEEE 754 double precision format

The double- precision floating-point number is calculated as $(-1^{s}) \times 1.F \times 2^{(E-1023)}$. The sign bit (s) is either 0 for non-negative number or 1 for negative numbers. The exponent field represents both positive and negative exponents. To do this, a bias is added to the actual exponent. For IEEE double-precision format, this value is 1023, for example, a stored value of 2011 indicates an exponent of (2011-1023), or 988. The mantissa or significand is composed of an implicit leading bit and the fraction bits, and represents the precision bits of the number.

Exponent values (biased) of all 0’s and all 1’s are reserved to encode special numbers such as zero, denormalized numbers, infinity, and NaNs.

I. Normalized numbers

A floating-point number is said to be normalized if the exponent field contains the real biased exponent other than all 0’s and all 1’s. For all the normalized numbers, the first bit just left to the decimal point is considered to be 1 and not encoded in the floating-point representation and thus also called the implicit or the hidden bit.

Assuming M_{max} and E_{max} to be the largest mantissa and exponent respectively, we can represent the largest normalized positive number for double precision format Max as:

$$Max = M_{max} \times 2^{E_{max}} \qquad (2\text{-}1)$$

Similarly, we get the minimum positive representable number Min from the minimum normalized mantissa M_{min} and the minimum exponent E_{min} :

$$Min = M_{min} \times 2^{E_{min}} \qquad (2\text{-}2)$$

Table (2-2) shows the maximum and minimum representable numbers which are encoded to approximate decimal values.

II. Denormalized numbers

A floating-point number is considered to be denormalized if the biased exponent field contains all 0’s and the fraction field doesn’t contain all 0’s. The implicit or the hidden bit is always set to 0. Denormalized numbers fill in the gap between zero and the lowest normalized number.

Table 2-2 : The maximum and minimum representable numbers and there's approximate decimal values.

	Fraction F	Mantissa M (with hidden one)	Exponent E (not biased)	Approximate Decimal
Min	All zeros	1	-1022	$2^{-1022} \approx 10^{-307.66}$
Max	All ones	$1 + (1-2^{-52})$	1023	$2^{1024} \approx 10^{+308.25}$

III. Infinity

In double-precision format, infinity is represented by biased exponent field of all 1's and the whole fraction field of 0's.

IV.Not a Number (NaN)

In double-precision format, NaN is similar to infinity in biased exponent field but the fraction field doesn't include all 0's.

V. Zero

In double-precision format, zero is represented by biased exponent field of all 0's and the whole fraction field of 0's. The sign bit represents -0 and +0, respectively.

The mapping from an encoding of a double-precision floating-point number to the number's value is summarized in Table (2-3).

2.2.2 Rounding modes

The result of an operation or function on floating point numbers may not be exactly a floating point number so it has to be rounded. The IEEE 754-2008 standard specifies five rounding modes.

Table 2-3 : Summary of corresponding values for double precision format.

	Fraction (F)	**Biased Exponent (E)**	**Value**
± Zero	All zeros	All zeros	±0
Denormalized number	Non zero	All zeros	$(-1)^{s} \times 0.F \times 2^{-1022}$
Normalized Numbers	**From** all zeros **to** all ones	**Not** all ones **and not** all zeros	$(-1)^{s} \times 1.F \times 2^{(E-1023)}$
± Infinity	All zeros	All ones	±∞
Not a number	Non zeros	All ones	± NaN

I. Round toward −∞ (Rn):

Rn(x): is the largest floating-point number (possibly −∞) less than or equal to x.

II. Round toward +∞ (Rp):

Rp(x): is the smallest floating-point number (possibly +∞) greater than or equal to x.

III. Round toward zero (RZ):

RZ(x): is the closest floating-point number to x that is no greater in magnitude than x.

IV. Round to nearest even (RN):

RN(x): is the floating-point number that is the closest to x. A tie-case happens when x falls exactly halfway between two consecutive floating-point numbers. It is rounded to the nearest value with an even (zero) least significant bit.

V. Round to nearest ties away from zero (RNZ):

RNZ(x): is the floating-point number that is the closest to x. When a tie case happens it is rounded to the nearest value with larger magnitude.

2.2.3 Exceptions

An exception can be signaled along with the result of an operation in IEEE standard. When exception occurs a status flag is raised which can be checked after the end of the operation. It may be with or replaced by some trap mechanism.

The standard defines five exceptions:

I. Invalid operation

This exception is signaled when an input is invalid for the function. The result is a quiet Not a Number qNaN.

Examples: $(+\infty) - (+\infty)$, 0/0.

II. Divide By Zero

This exception is signaled when an exact infinite result is defined for a function on finite inputs.

Examples: $^1/_0$ and $\log \mathrm{B}(0)$.

III. Overflow

This exception is signaled when the rounded result with an unbounded exponent range would have an exponent larger than E_{max} .

IV.Underflow

This exception is signaled when a tiny nonzero result is detected. This can be done before rounding or after rounding. If the result of an operation is exact, then the underflow flag is not raised.

V. Inexact

This exception is signaled when the exact result is not exactly representable.

2.3 Standard floating point fused multiply-add Algorithm

A fused multiply add unit performs the multiplication $\mathrm{B} \times \mathrm{C}$ followed immediately by an addition of product and a third operand A so that the calculation of $(\mathrm{A} + \mathrm{B} \times \mathrm{C})$ is done as a single and indivisible operation. It is also capable of performing multiplies only, by setting $\mathrm{A} = 0.0$, and adds (or subtract) only by setting for example $\mathrm{B} = 1.0$.

The fused multiply add (FMA) operation was introduced in 1990 on the IBM RS/6000 [12], [13] for the single instruction execution of the equation $(\mathrm{A} + \mathrm{B} \times \mathrm{C})$ with single and double precision floating-point operands. Because of its use in many applications as DSP, graphics processing, FFTs, FIR filters and division applications many algorithms are introduced after that for it.

This operation reduces the number of interconnections between floating point units, the number of adders and normalizers, and provides additional accuracy compared to separate multiply and add units. That is because of performing a single, instead of two, round /normalizes steps. On the other hand it increases the precision and delay of the adder and requires more complex normalizers.

2.3.1 Basic Algorithm

Let A, B and C be the operands represented by $(\mathrm{M_a}, \mathrm{E_a})$, $(\mathrm{M_b}, \mathrm{E_b})$ and $(\mathrm{M_c}, \mathrm{E_c})$ respectively. The significand are signed and normalized, and the result W is given by:

$$\mathrm{W} = \mathrm{A} + (\mathrm{B} \times \mathrm{C}) \qquad (2\text{-}3)$$

Where W is represented by $(\mathrm{M_w}, \mathrm{E_w})$, where $\mathrm{M_w}$ is also signed and normalized. The high level description of this operation is composed of the following five steps:

1. Multiply significand M_b and M_c, add exponents E_b and E_c, and determine the alignment shift and shift M_a, produce the intermediate result exponent $E_w = max(E_a, E_b + E_c)$.
2. Add the product and the aligned $\mathrm{M_a}$.
3. Normalize the adder output and update the result exponent.
4. Round.
5. Determine the exception flags and special values.

2.3.2 Basic implementation

Using the above algorithm, the standard floating point fused multiply-add was designed. The organization of a FMA unit is shown in Figure (2-2). We can see that the addition of the multiplier product and the aligned addend is done firstly by carry save adder (CSA) then a carry propagate adder (CPA). The sticky bit calculation which is needed for rounding and the anticipation of the leading zeros which is needed for normalization are performed parallel with CPA. The standard architecture is the baseline algorithm for floating-point fused multiply-add in any kind of hardware and software design.

2.4 Conclusion

The aim of this thesis is to implement double-precision binary floating point fused multiply add unit so in this chapter we give an overview of IEEE standard of floating point arithmetic with focusing in double-precision binary floating point format. We also briefly explain the standard algorithm of the fused multiply add operation and give the basic block of it.

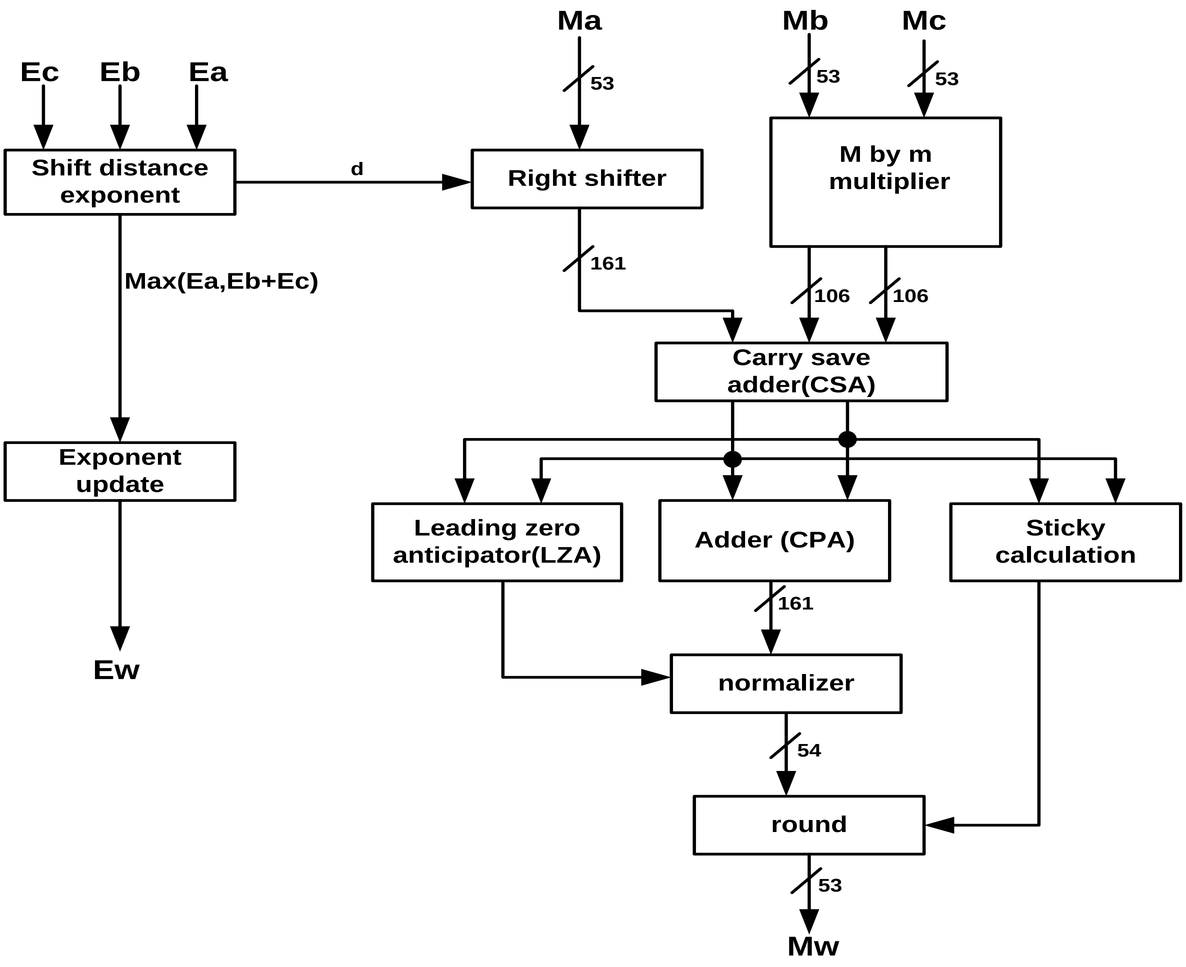


Figure 2-2 Basic implementation of FMA operation

Chapter 3 : Previous Work on the Floating-Point Fused Multiply Add Architecture

3.1 Basic architecture IBM RS/6000

As cited before in chapter 2, the first implementations of fused multiply add unit (FMA) was in 1990 by IBM RS/6000 [12], [13]. Several general-purpose processors like IBM PowerPC, the HP PA-8000, and the HP/Intel Itanium implemented FMA unit after that [8], [24] Their implementations were modified versions from the original implementation of IBM RS/6000. Figure (3-1) shows the original fused multiply-add architecture.

3.2 A fused multiply add unit with reduced latency

The greatest deviation from the original IBM RS/6000 architecture comes from a paper by T. Lang and J.D. Bruguera on a reduced latency fused multiply add unit [10]. This paper introduces a new approach to improve the performance of the fused multiply add unit. The idea depends on the combination of the addition and rounding stages into one stage. Although this idea is used before in floating-point adder and floating-point multiplier architectures as in [17], [33] it is more difficult to apply in fused multiply add unit. That is because of the postponement of normalization after add/round module is not possible in FMA operation because the rounding position is not known until the normalization has been performed; on the contrary, in case of addition it is possible because no rounding is required when massive normalization occurs. Lang and Bruguera describe that in order to combine the addition and rounding stages in a fused multiply add unit, the normalization must be done first.

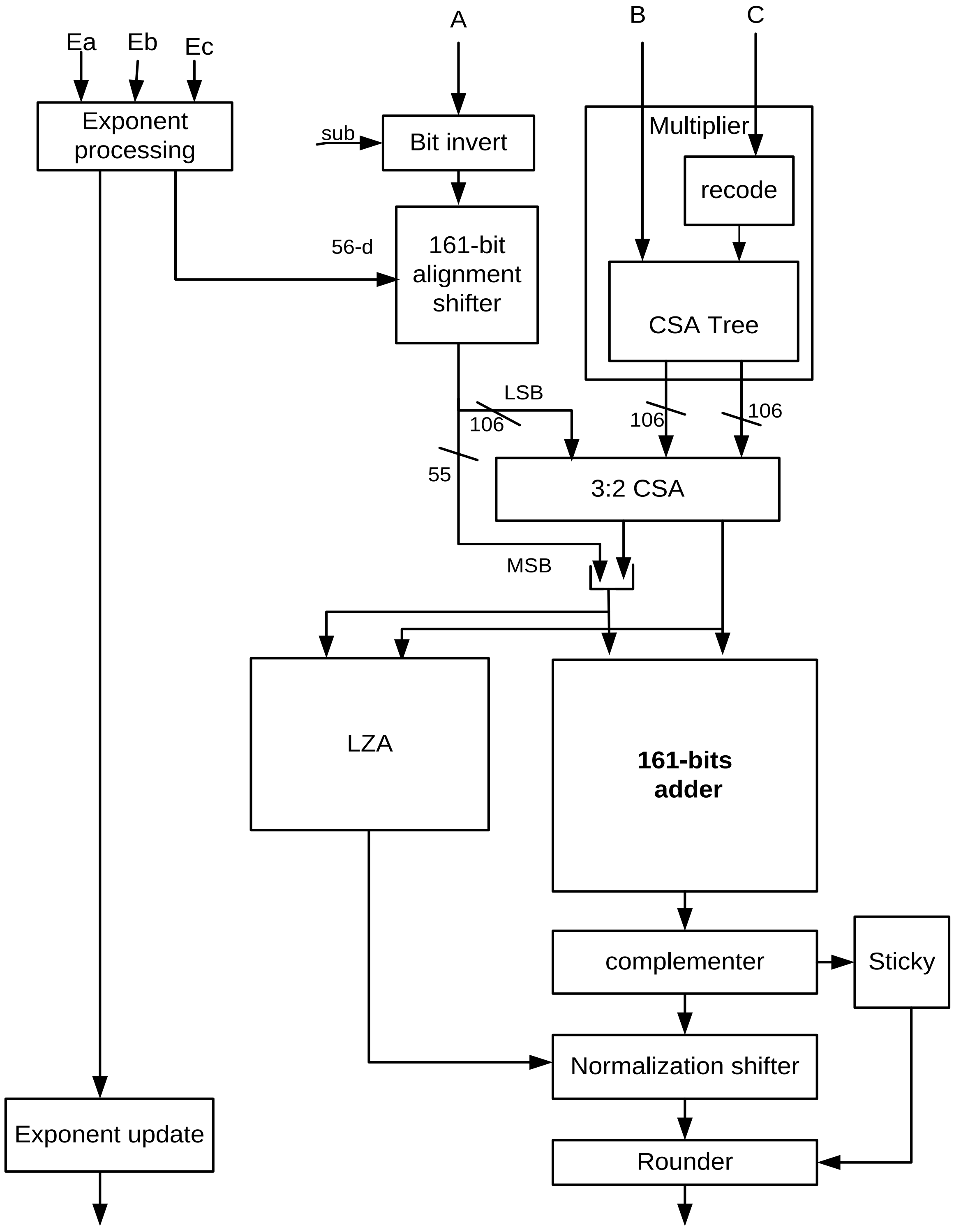


Figure 3-1: original fused multiply-add architecture.

The reduced latency fused multiply-add architecture is shown in Figure (3-2). In this design, the aligned addend is combined with the multiplier product in the same way as in the IBM RS/6000. The anticipation of leading zeros is done first in LZA block parallel to the reduction of vectors by using carry save adder (CSA) and a detection of output sign in sign detection module. Because the normalization shifter has to begin its job after LZA block and the LZA block is on the critical path authors propose to overlap the operation of LZA with the operation of the normalization shifter. It is done by designing a new LZA encoder which produces control signals which control the normalization shifter once they are calculated. However, despite the overlap, a time gap still exists between LZA block and the normalization shifter. To overcome this gap a part of dual adder (next stage) is anticipated before the normalization.

When the data exits the normalization stage, it enters the add/round module. The data is split into two groups, a 51-bit in dual adder and a 108-bit in carry/sticky block. The carry/sticky block creates and passes the rounding information bits to rounding control, which then selects the correct augmented adder output. The data are post-normalized, and the fused multiply-add is complete.

The paper claims an estimated 15-20% reduction in latency as compared to a standard fused multiply-add [10]. This result is calculated theoretically, and the actual architecture has yet to be implemented in either a synthesized or a custom CMOS silicon design [19].

3.3 Basic floating point Fused Multiply-Add components

This part introduces design details of blocks used in basic architecture.

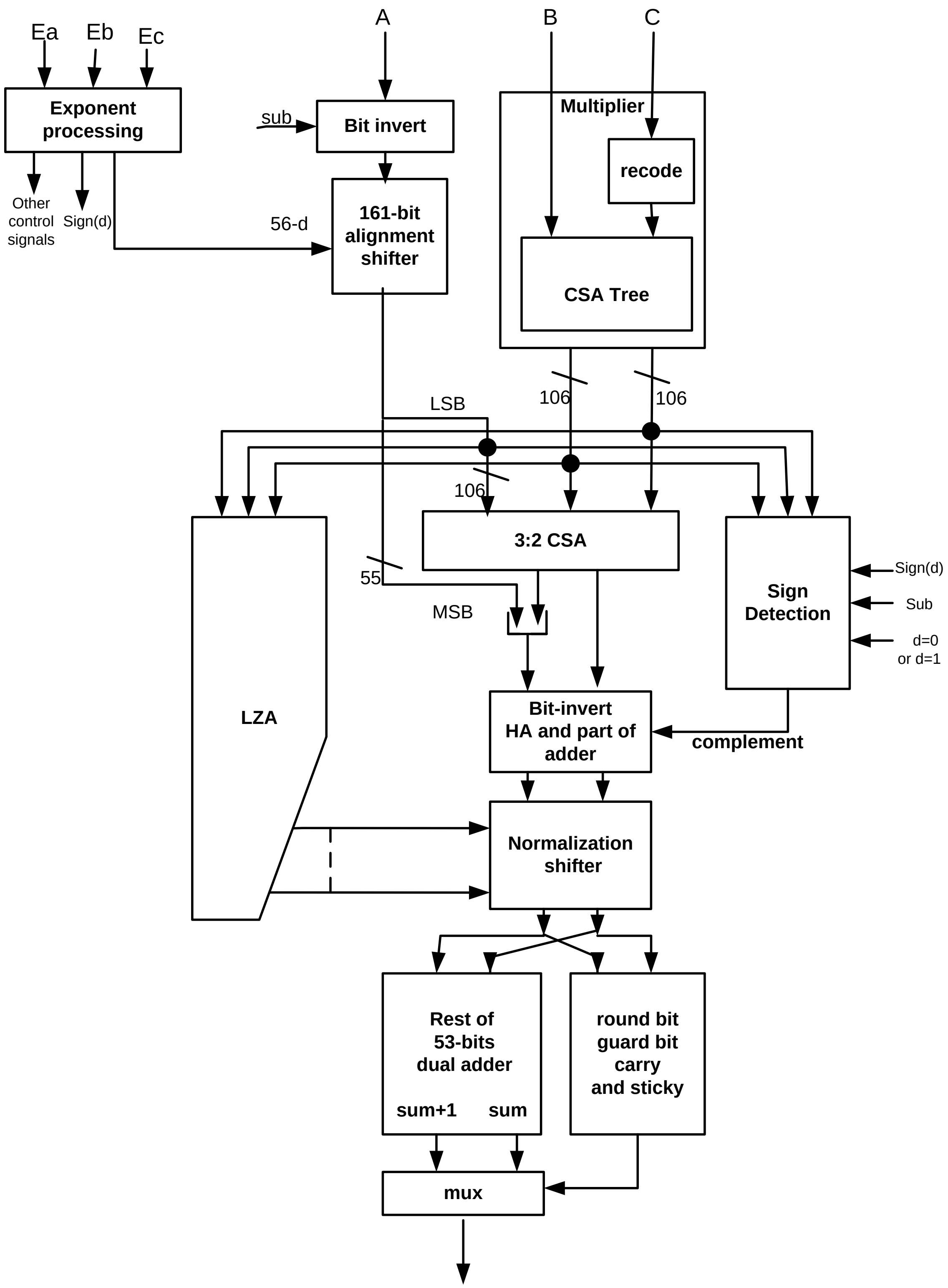


Figure 3-2: Lang/Bruguera combined addition/rounding stage fused multiply-add

3.3.1 The multiplier

Multiplication is the process of generation and addition of the partial products. Multiplication algorithms differ in how the generate partial products and how the partial products are added together to produce the final result.

3.3.1.1Partial Product Generation

Floating point fused multiply add unit includes a multiplier which uses a modified Booth's algorithm to generate partial products. In Booth's algorithm the multiplier operand C is often recoded into a radix higher than 2 in orders to reduce the number of partial products. The most common recoding is radix-4 recoding (modified booth's recoding) with the digit set {−2, −1, 0, 1, 2} is shown in table (3-1). For a series of consecutive 1's, the recoding algorithm converts them into 0's surrounded by a 1 and a (−1), which has the potential of reducing switching activity.

Each three consecutive bits of the multiplier C represent the input to booth recoding block and the output from this block selects the right operation on the multiplicand B which may be "shift and invert" or "invert" or "equal zero" or "no operation" or "shift " (-2B, -B, 0, B, 2B) respectively due to the bits of multiplier. Also there is an output bit $sign$ to indicate the sign and complete the 2's complement if the partial product is negative. Figure (3-3) shows the generation of one partial product. For double precision format the multiplier has 53 bits. By using modified booth recoding the number of partial products are reduced to 27 partial products. Figure (3-4) shows the generation of all 27 partial products.

Table 3-1: Modified Booth's recoding

Bits of multiplier C			Encoding Operation on multiplicand B
C_{i+1}	C_i	C_{i-1}	
0	0	0	+0
0	0	1	+B
0	1	0	+B
0	1	1	+2B
1	0	0	-2B
1	0	1	-B
1	1	0	-B
1	1	1	-0

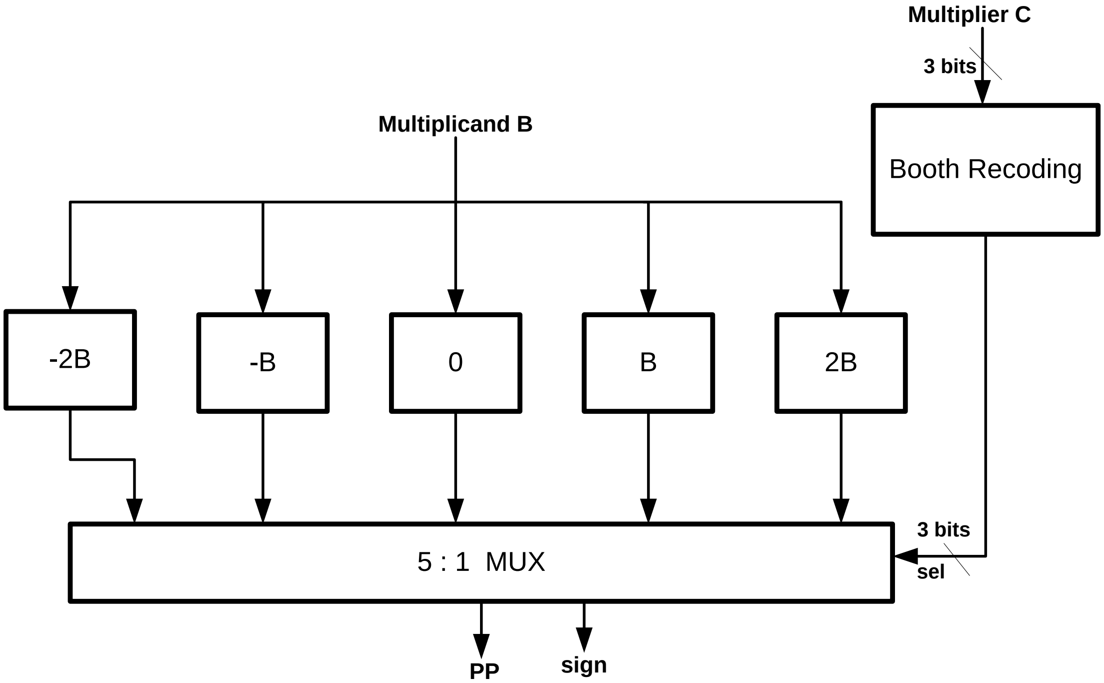


Figure 3-3: One partial product generation

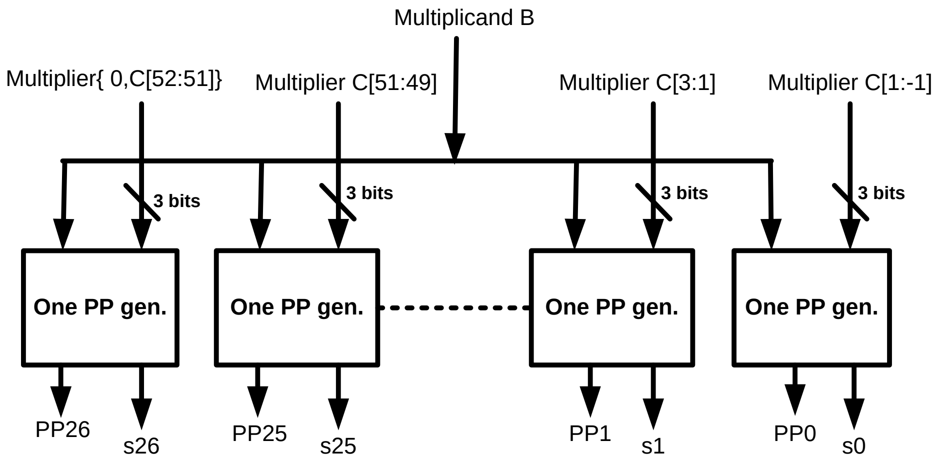


Figure 3-4: All partial products generation

3.3.1.2 Partial product reduction

After generation of the partial products, the partial Products begin compression using 3-2 CSA tree. The reduction occurs by rows where each three partial product in same level will be input to CSA adder and output 2 operands (i.e. partial products) to the next level, and so on. For 27 partial products, 8 stages are required to produce a product in carry-save, or a carry vector and sum vector that need only to be added for a complete multiply. Figure (3-5) shows the block diagram of CSA reduction tree.

3.3.2 The alignment shifter

To reduce the latency in a fused multiply add unit, the inversion and alignment of the significand of A is done in parallel with the multiplication. The inversion provides the one's complement of A for an effective subtraction. The shift amount for the alignment depends on the value of $\mathrm{d} = \mathrm{E_a} - (\mathrm{E_b} + \mathrm{E_c})$, where $\mathrm{E_a}$, $\mathrm{E_b}$ and $\mathrm{E_c}$ are the exponents of the A, B and C operands, respectively.

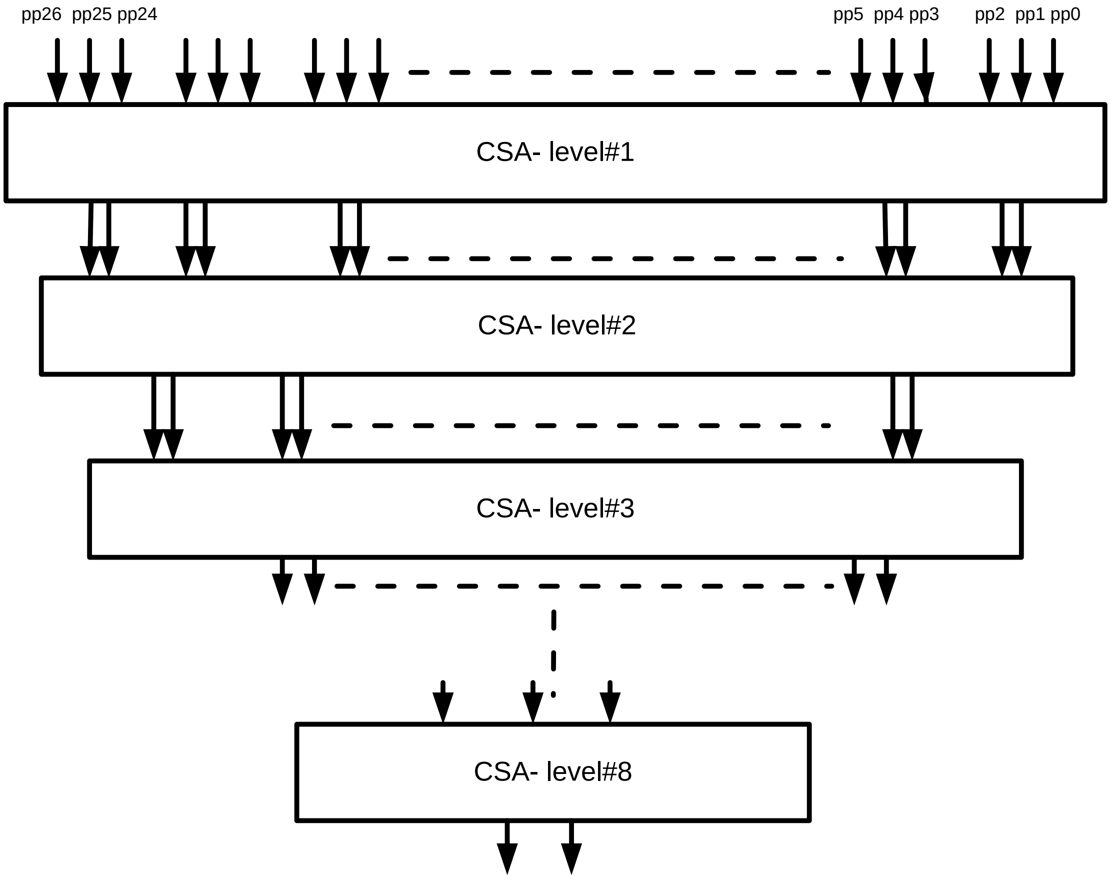
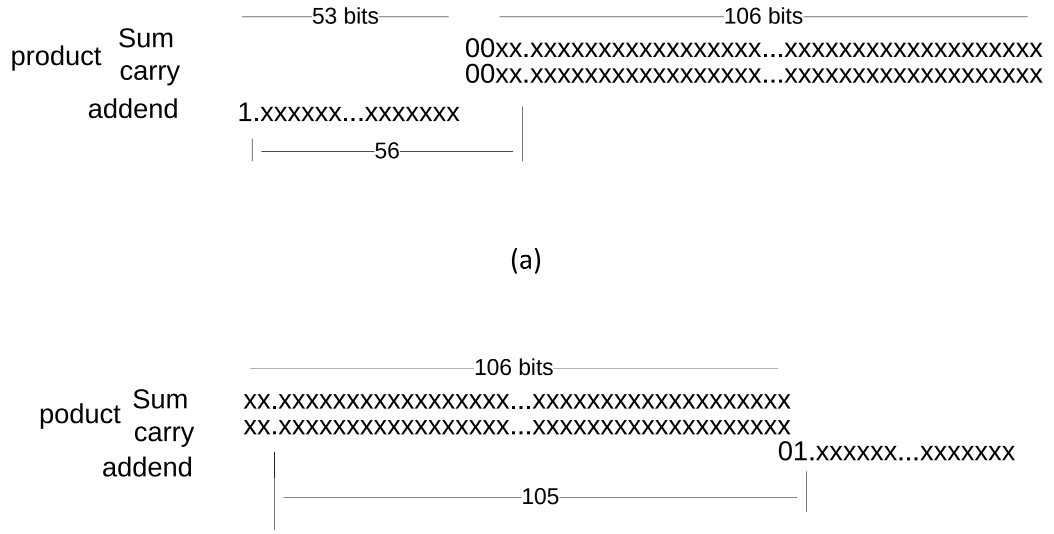


Figure 3-5: The block diagram of CSA reduction tree.

When $d \geq 0$ (i.e. $E_a > (E_b + E_c)$), in a conventional alignment, $B \times C$ would have to be aligned with a right shift of d bits. Instead of that, shift the addend A to the left to perform the alignment parallel with multiplication. For double precision format the maximum left alignment shift would be 56 bits, see Figure (3-6) (a). When $d \geq 56$, $B \times C$ is placed to the right of the least significant bit of A; in this case, $B \times C$ affects only the calculation of the sticky bit. The maximum left shift is obtained by observing that the guard (position 53) and the round (position 54) bits are 0 when the result significand corresponds to the addend. Consequently, two additional positions are included, resulting in the shift of 56 positions.

When $\mathrm{d} < 0$, the addend A would have to be aligned with a right shift of d bits. In this case, the maximum alignment shift would be 105 bits for double precision format see Figure (3-6) (b).


53 bits
106 bits
product
Sum
carry
00xx.xxxxxxxxxxxxxxxxx...xxxxxxxxxxxxxxxxxxx
00xx.xxxxxxxxxxxxxxxxx...xxxxxxxxxxxxxxxxxxx
addend
1.xxxxxx...xxxxxxx
56
106 bits
poduct
Sum
carry
xx.xxxxxxxxxxxxxxxxx...xxxxxxxxxxxxxxxxxxx
xx.xxxxxxxxxxxxxxxxx...xxxxxxxxxxxxxxxxxxx
addend
01.xxxxxx...xxxxxxx
105

(a)

(b)

Figure 3-6: Position of addend using bidirectional shift (a) Maximum left shift. (b) Maximum right shift.

For shift amounts larger than 105, $\mathrm{d} < -105$, the operand A is placed to the right of the least-significant bit of $\mathrm{B} \times \mathrm{C}$, affecting only the calculation of the sticky bit.

To avoid bidirectional shifter the alignment is implemented as a right shift by placing the addend A to the left of the most significant bit of the product $\mathrm{B} \times \mathrm{C}$ by 56 bits. This is shown in Figure (3-7) (a). Two extra bits are placed between the addend A and the product $\mathrm{B} \times \mathrm{C}$ to allow correct rounding when A is not shifted. For $\mathrm{d} \geq 0$ with this implementation, A is right shifted $(56 - \mathrm{d})$bits; then, the shift amount is $\text{shift amount} = \max\{0{,}56 - \mathrm{d}\}$, see Figure 3-7 (b).

For $\mathrm{d} < 0$, A is right shifted $56 - \mathrm{d}$ bits, see Figure (3-7) (c), then $\text{shift amount} = \min\{161, 56 - \mathrm{d}\}$.By combining both cases, the shift amount is in the range [0:161], requiring a 161-bit right shifter. Moreover, the shift amount is computed as $\text{shift amount} = 56 - \mathrm{d}$

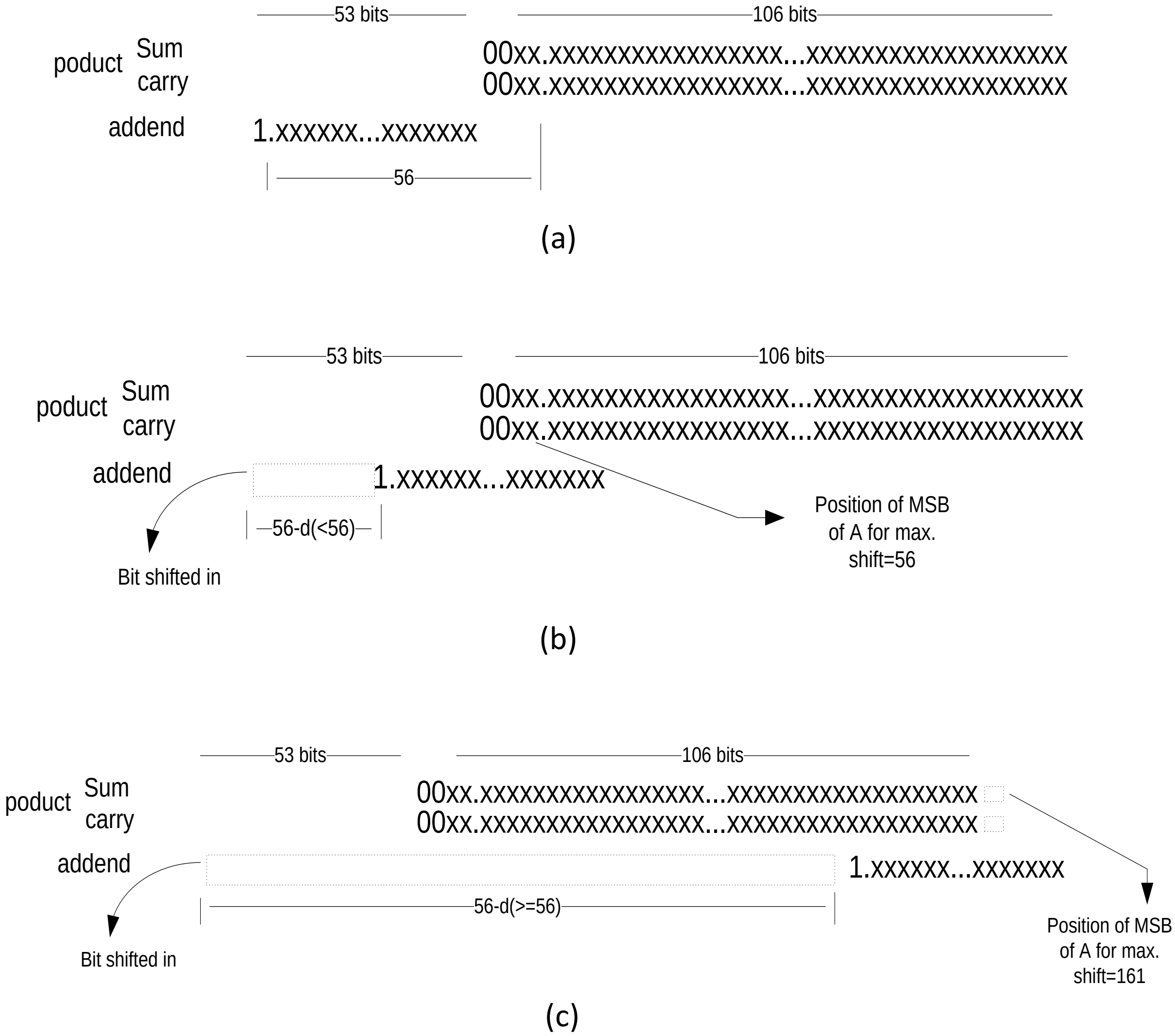

53 bits
106 bits
poduct
Sum
carry
00xx.xxxxxxxxxxxxxxxxx...xxxxxxxxxxxxxxxxxxx
00xx.xxxxxxxxxxxxxxxxx...xxxxxxxxxxxxxxxxxxx
addend
1.xxxxxx...xxxxxxx
56
(a)
53 bits
106 bits
poduct
Sum
carry
00xx.xxxxxxxxxxxxxxxxx...xxxxxxxxxxxxxxxxxxx
00xx.xxxxxxxxxxxxxxxxx...xxxxxxxxxxxxxxxxxxx
addend
1.xxxxxx...xxxxxxx
56-d(<56)
Bit shifted in
Position of MSB of A for max. shift=56
(b)
53 bits
106 bits
poduct
Sum
carry
00xx.xxxxxxxxxxxxxxxxx...xxxxxxxxxxxxxxxxxxx
00xx.xxxxxxxxxxxxxxxxx...xxxxxxxxxxxxxxxxxxx
addend
1.xxxxxx...xxxxxxx
56-d(>=56)
Bit shifted in
Position of MSB of A for max. shift=161
(c)

Figure 3-7: Alignment of A. (a) before alignment. (b) Alignment with d ≥ 0. (c) Alignment with d < 0.

3.3.3 3:2 CSA

The multiplier produce 106-bit sum and carry vectors that are reduced together with the aligned A using 3:2 CSA. Although the output of the multiplier must be positive number because we multiply two positive numbers (sign and magnitude representation), one of the two output vectors of the multiplier (sum and carry) may be negative because of using booth algorithm which use negative sets {-1,-2} which convert a positive number with sign and magnitude representation to a negative number with two's complement representation. The addition of sum and carry vectors must be a positive number but one of them, not both, may be negative.

Instead of using 161-bit CSA, Only the 106 least-significant bits of the aligned A are needed as input to the 3:2 CSA, because the product (i.e. sum and carry vectors) has only 106 bits and The 55 most-significant bits will be sign extension bits which have two cases {0, 0} if both sum and carry vectors are positive or {0, 1} if one of them is negative. For the 55 most significant bits, we use two multiplexers, one to select between A and inverted A as a sum vector and the second one to select between zeros and A as a carry vector by Xor-ing sign extension bits then the outputs of the two multiplexers are concatenated at the output of the CSA to obtain the 161-bit sum and carry words, see Figure (3-8).

3.3.4 Carry propagate adder (CPA) and leading zero anticipator (LZA)

The output vectors of 3:2 CSA are now input to a 161-bit carry propagate adder (CPA) and a leading zero anticipator (LZA) in the same stage.

We use the prefix adder to implement the carry propagate adder in this thesis because it is very efficient in binary addition due to its regular structure and fast performance.

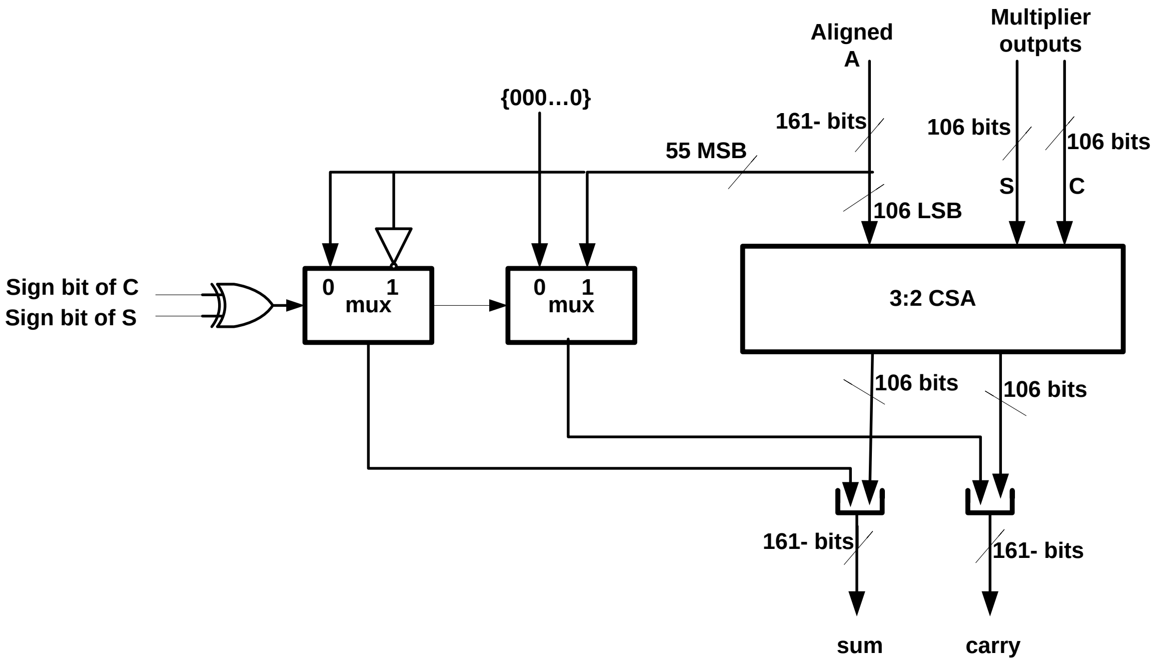


Figure 3-8: Selection of MSB of sum and carry words of CSA by sign bits.

After the addition operation is done the adder output is left shifted by the number of leading zeros to obtain a normalized result. It is called the normalization step. Instead of waiting for the adder output to determine the number of leading zeros it can be anticipated in parallel with the adder using a leading zero anticipator (LZA) to eliminate this operation from the critical path. LZA unit anticipates the number of leading zeros from the operands of the adder. Since the shift amount is already determined by LZA the normalization operation is performed once the result of addition is obtained.

The leading zero anticipator (LZA) has two main parts: the encoding of the leading-one position (detection module) and the correction of this position (correction module). The detection module are divided into two parts the first one is called LZA logic and it determines the number of leading zeros (i.e. the position of the leading one) by producing a string of zeros and ones where the position of the most significant 1 is the position of the leading one. The second part, called leading zero detector (LZD), counts the number of zero digits from the left-most position until the first nonzero digit is reached (i.e. leading one position). since the detection is done from most significant bit to least significant bit (from left to right) regardless of the carry that may come from the least significant bit, the detection of leading one position may be off by one bit.

The LZA logic takes two input strings and uses a set of logical equations given in [23], cited in Equation (3-1), to predict the bit position of the leading '1' after a subtraction that causes massive cancellation. If the bits are numbered such that bit 0 is the most significant, then, the indicator f_i is equal to one when:

$$\mathrm{f_i = t_{i-1}.(g_i.z_{i+1} + z_i.\bar{g}_{i+1}) \oplus \bar{t}_{i+1}(z_i.\bar{z}_{i+1} + g_i.\bar{g}_{i+1}),\ i > 0} \quad (3\text{-}1)$$

$$\mathrm{f_0 = \bar{t}_0.t_1}$$

Where

$$t_i = a_i \oplus b_i$$

$$g_i = a_i\ .\ b_i \quad (3\text{-}2)$$

$$z_i = \bar{a}_i\ .\ \bar{b}_i$$

After LZA logic LZD is used to drive the normalization shifter by encoding the position of leading one to its weighted binary representation. The LZD unit assumes n bits as input and produces $\log_2 \mathrm{n}$ bits of the leading one position. The LZD proposed by Oklobdzija [15] is used to implement the LZD for the floating points fused multiply add of IBM RS/6000 in this thesis. Table (3-2) shows the truth table of 2-bits LZD. By using two 2-bit LZD's we can get 4-bit LZD as shown in Figure (3-9) (a), the logic structure of 4-bit LZD is shown in Figure (3-9) (b). Following the same concept we can get LZD with higher number of output using hierarchical structure, as an example a 16-bit LZD is shown in Figure (3-10). It can be included that the numbers of levels needed are $\log_2 \mathrm{n}$ level.

Number of levels can be reduced by one by implementing 4-bits LZD directly from original bits. Figure (3-11) shows direct implementation of 4-bit LZD using original bits using truth table shown in Table (3-3).

Table 3-2: Truth table of 2-bit LZD

pattern	Position	Valid
1x	0	Yes
01	1	Yes
00	X	No

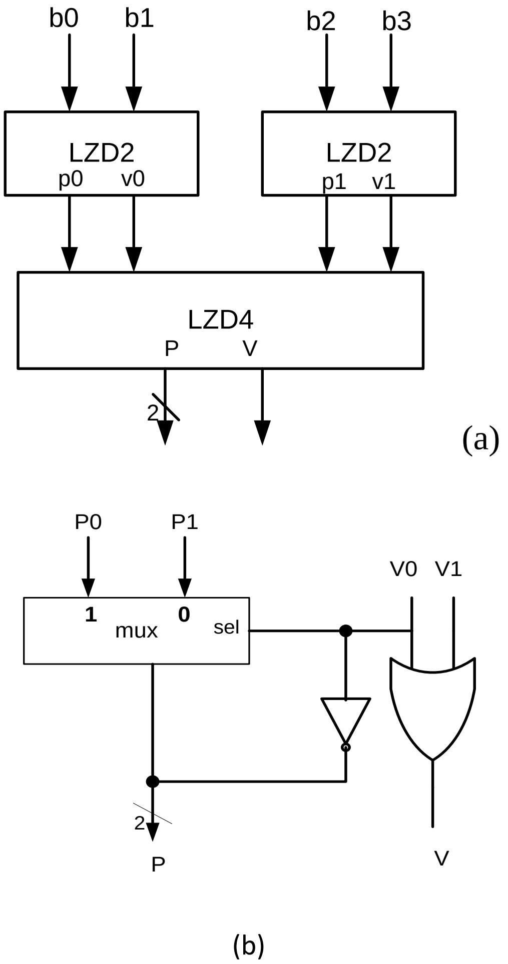


Figure 3-9: 4-bit LZD (a) using two 2-bits LZD's. (b) the logic structure of 4-bit LZD block.

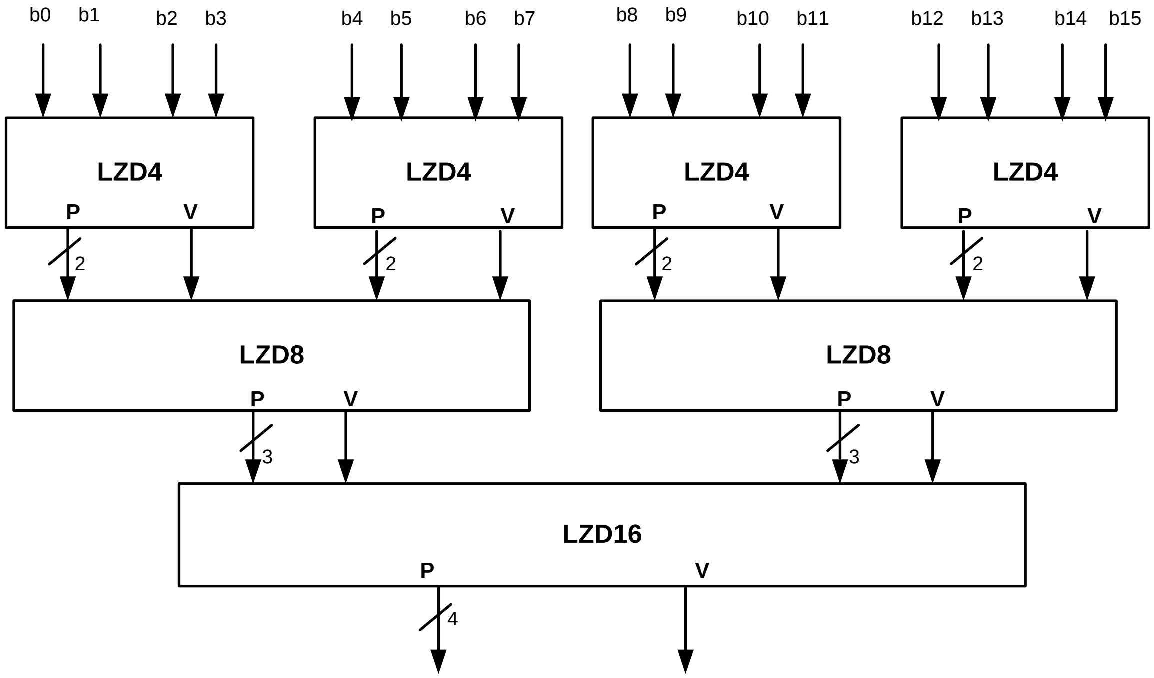


Figure 3-10: Structure of 16-bit LZD

Table 3-3: Truth table of 4-bit LZD

Pattern **$b_0b_1b_2b_3$**	**Position**	**Position (binary)** **P_1P_0**
1011	0	00
0100	1	01
0011	2	10
0001	3	11
0000	X	XX

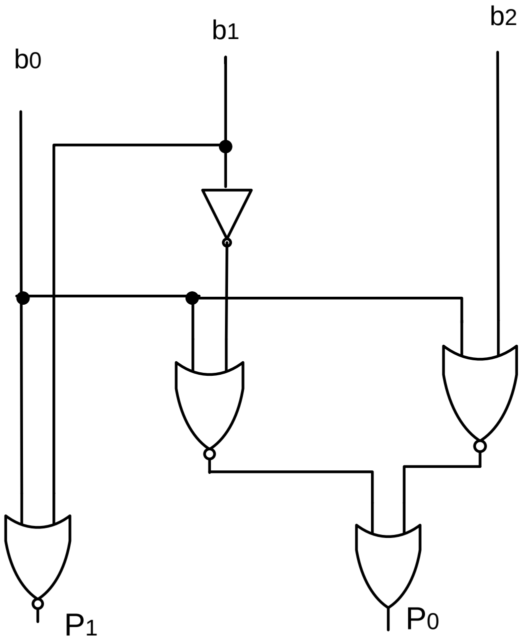


Figure 3-11 : One level implementation of 4 bit LZD

As cited before, the prediction of the position of the leading digit may differ from that of the true result by one. Then, the result is not correctly normalized and an additional left shift by one position is needed. The exponent should be also decreased by one. Many methods have been proposed in [3] so far to handle the one-bit error of the LZA logic. One of these methods, which are used here in this thesis, compares the position of the leading one in one hot representation of the anticipated leading digit; vector contains zeros except the digit indicates the position of the leading one, and adder output [3], [21]. The comparison is done by a bitwise AND of the one-hot vector with the adder output.

If all the bits of the derived word are equal to zero then the predicted position of the leading digit is not correct.

A transformation is needed to obtain the one hot vector from LZA logic output. To derive this one-hot representation, an intermediate S string is produced at first. The S vector is produced where its bits that follow the leading digit are set to one, while the other more-significant bits remain to zero. For example, for the vector 00110101 the S vector is equal to 00111111. Assume the vector output from LZA logic is defined as $\mathrm{A} = \mathrm{A}_{\mathrm{n}-1}\mathrm{A}_{\mathrm{n}-2} \ldots\ldots \mathrm{A}_0$ where $\mathrm{A}_{\mathrm{n}-1}$ is the most significant bit, the i^{th} bit of S denoted as s_{i}, is defined as follow:

$$\mathrm{S}_{\mathrm{i}} = \mathrm{A}_{\mathrm{n}-1} + \mathrm{A}_{\mathrm{n}-2} + \cdots + \mathrm{A}_{\mathrm{i}+1} + \mathrm{A}_{\mathrm{i}} \qquad (3\text{-}3)$$

Where

$$0 \leq \mathrm{i} \leq \mathrm{n} - 1$$

The one-hot representation of the leading digit (L word) is produced from S by detecting the case of different consecutive bits. Hence

$$\mathrm{L}_{\mathrm{i}} = \bar{\mathrm{S}}_{\mathrm{i}+1}.\mathrm{S}_{\mathrm{i}} \qquad \text{For} \qquad 0 \leq \mathrm{i} \leq \mathrm{n} - 2 \qquad (3\text{-}4)$$

And

$$\mathrm{L}_{\mathrm{n}-1} = \mathrm{S}_{\mathrm{n}-1}$$

The circuit that computes the L word is called a priority encoder. This transformation is done in parallel with the operation of the LZD then there is no increase in delay, in addition the LZA is not in the critical path as Shown in Figure (3-12).

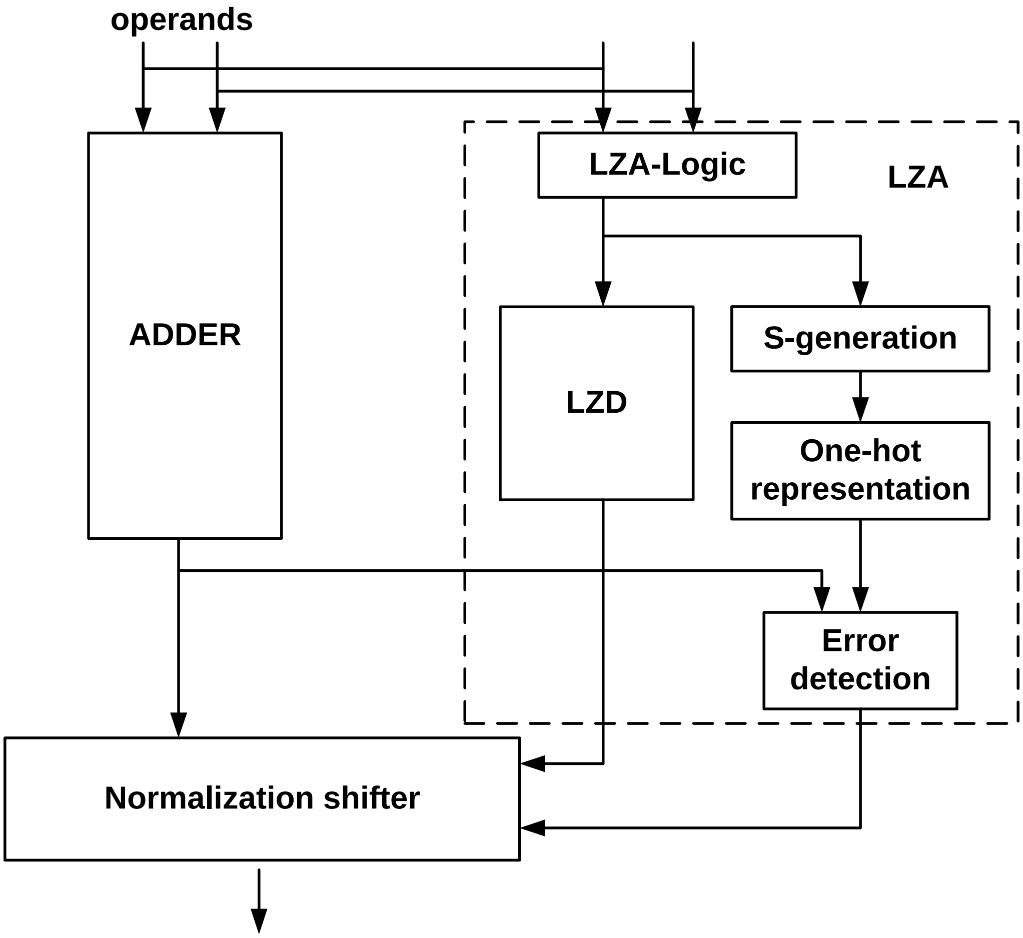


Figure 3-12: block diagram of LZA module.

3.3.5 The normalization

Using the results from the LZD, the result from the adder is shifted left to normalize the result. That means now the first bit is 1.The normalizer is mostly a large shifter. The shifter has more than one stage. As shown in Figure (3-13), the stages are organized from the coarsest to the finest. The last stage performs a shift by one or two due to correction signal. This should have a negligible effect on the delay of the last stage.

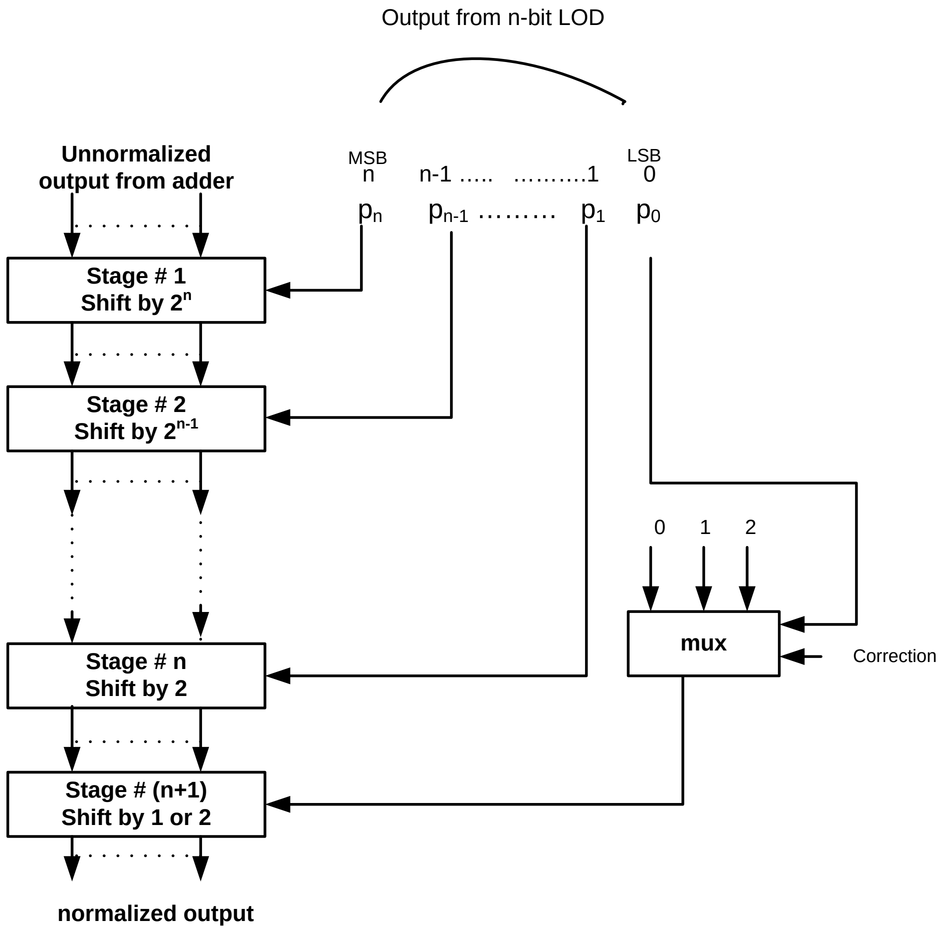


Figure 3-13: n-bits normalization shifter

3.3.6 The rounding

The rounding block rounds the result to nearest floating point number due to the round mode and performs post-normalization in case of an overflow. The round decision is taken by knowing also sticky and round bits. The sticky bit is calculated from the result by OR-ing all least significant bits after the round bit.

3.4 Conclusion

We discuss in this chapter previous works in fused multiply add architectures and concern two main architectures which are the basic architecture IBM RS/6000 and the fused multiply add with reduced latency proposed by Bruguera/Lang .

That is because the proposed architecture has a lot of details like the fused multiply add proposed by Bruguera/Lang and then is compared with basic architecture. We maintain also design details of blocks used in basic fused multiply add architecture like multiplier, alignment shifter, LZA, normalizer and rounder.

Chapter 4：Implementation of Fused Multiply Add Unit

4.1 Proposed Fused Multiply Add

The proposed fused multiply add is actually the first implementation of the Lang/ Bruguera fused multiply add algorithm. The implementation is done with slight change in Lang/ Bruguera architecture.

In Lang/Bruguera design the LZA and the sign detection modules have three inputs coming from the output of the multiplier and the aligned addend, but in the proposed design the inputs to these modules come from the output from 3:2 CSA after combining the multiplier outputs and the aligned addend.

The sign detection module is used to detect negative sign of output. If the sign is negative the output of the 3:2 CSA is complemented. Comparison is needed between the output of the multiplier $\mathrm{B} \times \mathrm{C}$ (i.e. two output vectors from CSA reduction tree) and the aligned A. To compare between these three vectors there are two solutions, adding the two vectors output from the multiplier first by a carry propagate adder then comparing its output with aligned A or reducing these three vectors using 3:2 CSA then comparing the output of this CSA. The second solution is preferred to eliminate using of carry propagate adder in order to decrease its delay because the output of this block, $comp$ signal, is used after that to select inputs to normalization shifter which are needed to be available as soon as possible before shift amount of normalization shifter is calculated.

Lang/Bruguera fused multiply-add architecture provides three inputs to the LZA block. The three inputs may be combined with a 3:2 CSA before entering the LZA logic unit. The Xiao-Lu paper [30] presents new equations that allow this three input string to predict the leading ‘1’.

A three input LZA removes the requirement for a 3:2 CSA and therefore decreases the number of logic stages. In our case it is suitable to use 3:2 CSA as used in sign detection module, as shown in Figure (4-1), because both blocks are in critical path and affect the overall delay.

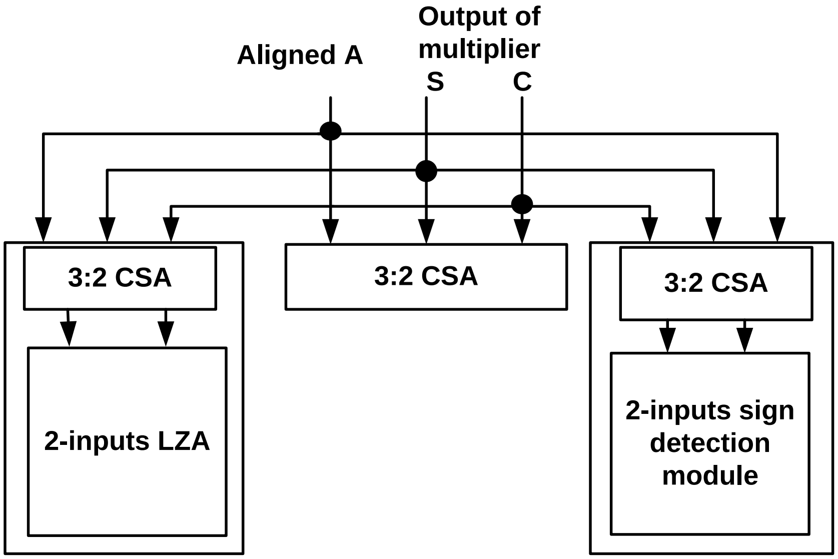


Figure 4-1: Three inputs LZA and sign detection module.

Instead of using three 3:2 CSA, we use only one in the proposed fused multiply add and then take its output to two inputs sign detection and LZA blocks. This change will not increase the overall delay as the delay of CSA already exists in the critical path. The proposed architecture is shown in Figure (4-2).

4.2 Design details of blocks used in proposed FMA

To implement this architecture there is a need to fully design the sign detection module with two inputs and redesign add/round module to include the correction of LZA error. In the remaining part of this chapter we will include the design of these blocks and the design of blocks that were modified from the basic architecture to be suitable in this implementation like LZA block and the anticipated part of the adder.

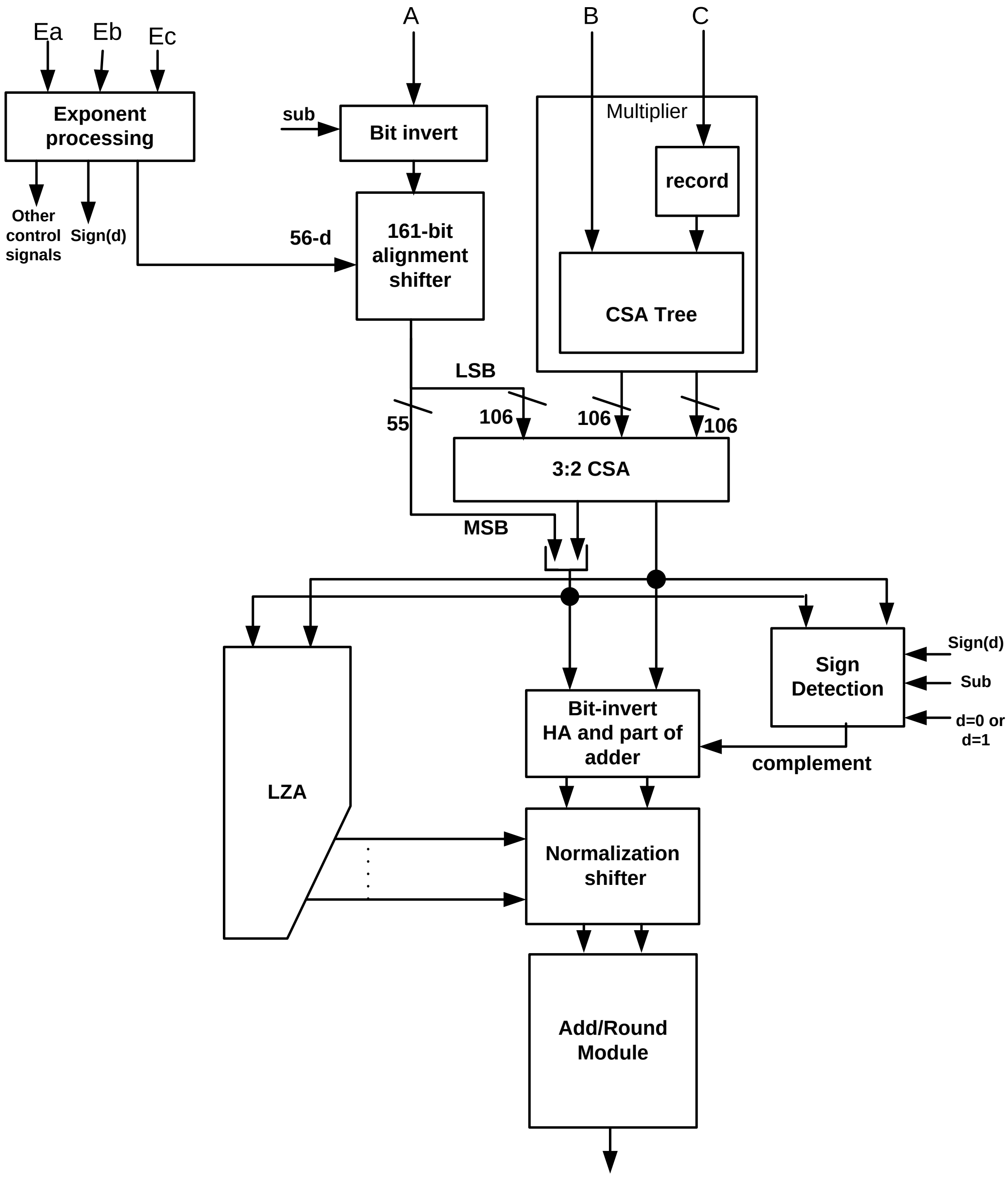


Figure 4-2: The proposed Fused Multiply Add.

4.2.1 Leading zero anticipator (LZA) and Normalization shifter

As cited before in chapter 3 the LZA block is used to determine the normalization amount. It is composed of two main modules the detection and the correction modules. The detection module has two main parts, the LZA logic which determines the position of the leading one by getting a string of bits having the same number of leading zeros as the sum and LZD which encodes the number of leading zeros in a binary representation.

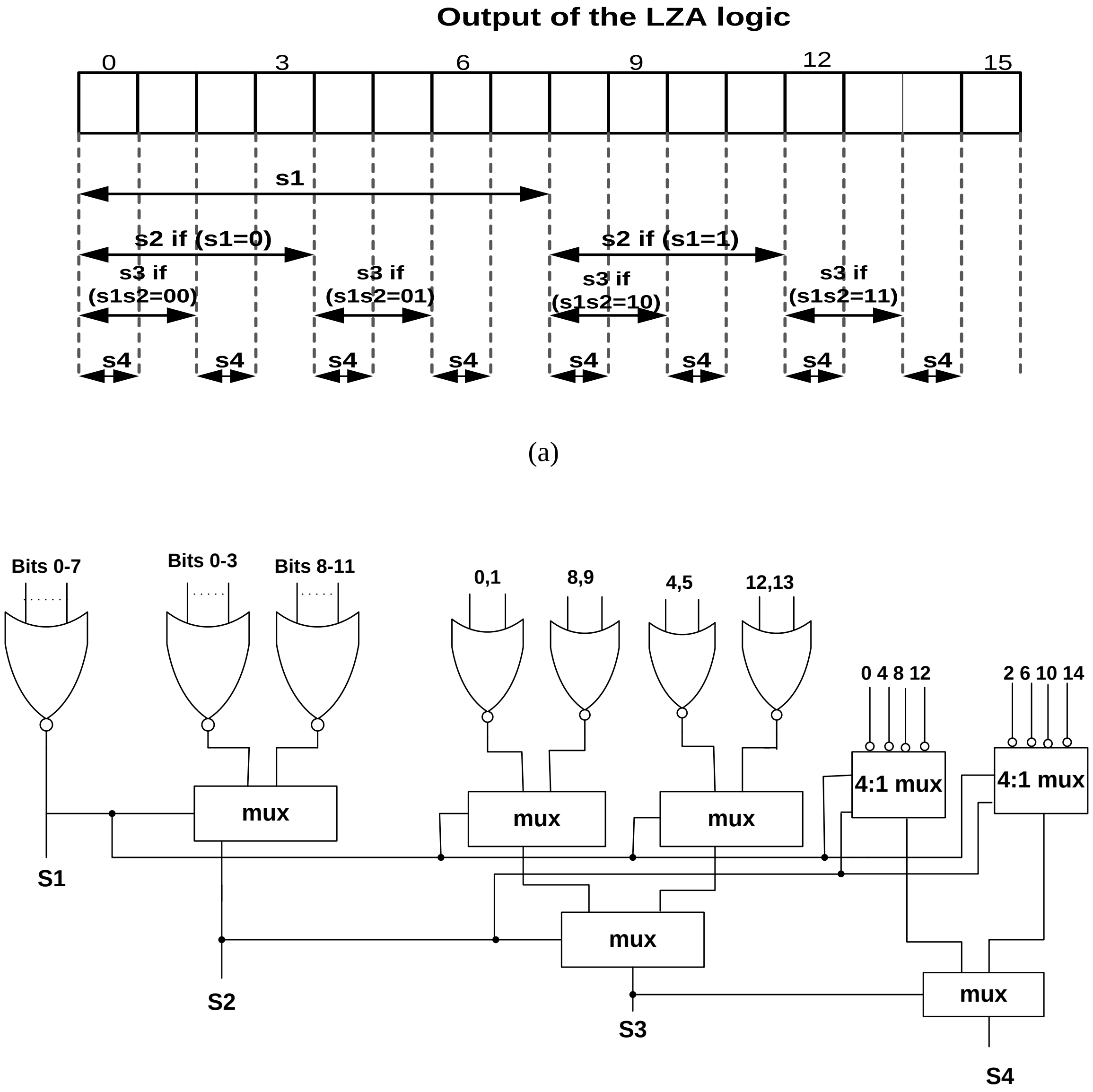
In the reduced latency Fused Multiply Add the LZA block is in the critical path. So to reduce its delay there is a solution proposed by Lang/Bruguera paper [10] by overlapping part of the LZA with the normalization shifter in such a way that the shift amount is obtained starting from the most-significant- bit and, once the first bit (MSB) is obtained, the normalization shift can start. It is contrary to what happen in conventional LZA where the normalization shifter has to begin its job after getting output from LZD.

Lang/Bruguera made this modification by replacing the LZD by a set of gates and Multiplexers. The basic idea of the encoding is to check the existence of the leading one in different groups of bits in LZA logic output where the number and the position of the checked bits depend on the weight of the output bit [1].

Figure (4-3) (a) shows which groups are explored to get each bit for a 16-bit normalization shift, for example, to get the most significant bit s1 the 8 most-significant bits of LZA logic output are checked and if all of them are zeros (s1=1) a normalization shift is needed by 8 bits. To get s2 two different groups of four bits are checked and then selected by s1 and so on. Figure (4-3) (b) shows the implementation of this algorithm. NOR gates are used to determine if there is some 1 in different groups of the string. Each bit si+1 is obtained after a delay of 2:1 Multiplexer from the previous bit si.

A Multiplexer with more than two inputs can be implemented using a tree of 2:1 Multiplexer’s.

It is clear that the most significant bits are computed first so the normalization operation can be started with the most significant bit after a delay of computing s1 only.



(a)

(b)

Figure 4-3: Encoding of the 16-bit normalization shift amount. (a) Bits checked to determine the shift amount. (b) Encoding of the shift amount.

Lang/Bruguera proposed another approach to overlap the operation of LZA and the operation of normalization shifter. They split the 162-bit normalization shifter into two stages a 54-bit normalization shifter followed by a 108-bit normalization shifter [10]. It is possible because the shift amount is greater than 56 if d is negative (d<0) which makes the 56-most significant bits are zero and a normalization shift of 54-bit,at least, can be performed in this case. LZA controls only the 108-bit normalization shift.

The second module of the LZA is the correction module which is responsible for the detection and correction of the error which may occur in the determination of the position of the leading one. Many approaches are proposed to correct this error without adding a large delay to the critical path.

The simplest method used is to check the most significant bit of the output of the normalization shifter, as shown in Figure (4-4) (a), and if it equals zero an error exists and it is corrected by adding a compensation shifter after the normalization stage [29]. The second method is proposed in [6], [16] and shown in Figure (4-4) (b). It uses the carries from the adder and checks the output of the LZD by a carry select circuit, then uses its output to feed the shift correction circuit. The third one is used before in chapter 3 and shown in Figure (4-4) (c). It uses a LZA that generates a one-hot vector having the same number of bits as the sum. It corrects the error by comparing the one hot vector with the sum from the adder in parallel with the shifter. A method proposed by Hinds and Lutz [5] uses a correction circuit which decides if the location of the leading one in the sum which may be in either an odd or even bit position agrees with the oddness of the predicted shift count from LZD. This is shown in figure (4-4) (d). Bruguera and Lang [9] have developed a scheme, shown in figure (4-4) (e), to detect the error in parallel with the LZA logic and generate a correction signal which may be used in parallel with the coarse shifter to generate a fine correction shift signal.

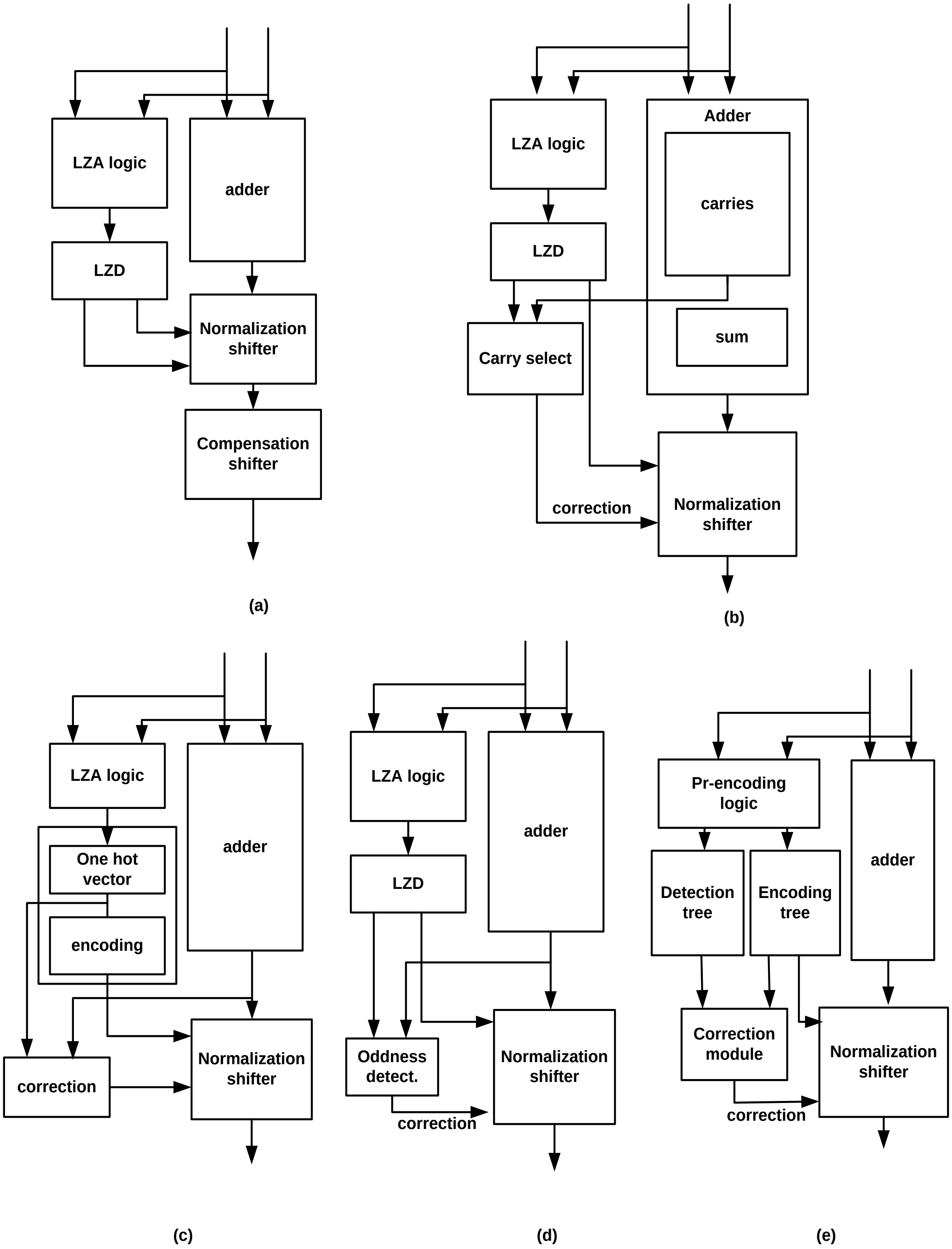


Figure 4-4: Correction of error of LZA based on (a) compensation shifter correction. (b) Carry select correction. (c) One hot vector. (d) Oddness detection. (e) Parallel correction.

All these methods except the last one use information from the adder and provide that the normalization step is performed after the addition but our architecture is based on the combination of the addition and rounding (using a dual adder) and the anticipation of the normalization before the addition. So we can’t use these schemes in our design.

The last design proposed by Bruguera/Lang [9] is only applicable and needed when the effective operation is a subtraction and operates on positive significand. The two significand are in sign-and-magnitude representation. It is not applicable here where the inputs to the LZA are from the 3:2 CSA outputs which are in two’s complement representation and may be positive or negative. So in this architecture, we postpone the correction of the error to add / round module.

4.2.2 Sign detection module

The function of the sign detection module is to detect the sign of the adder output and complement the outputs of the CSA when the result is negative. The two’s complement of the CSA output is performed by inverting the sum and carry words and adding two in the least-significant position.

The result can be negative only for effective subtraction. Moreover, since, in effective subtraction, we always complement the significand of A, the result can be negative only for $d \geq 0$ (i.e. Ea $> (Eb + Ec)$). When $d \geq 2$, the result is always negative, While for d=0 or d=1 with overflow in multiplication (i.e. case of equal exponents) the result may be positive or negative. A magnitude (unsigned) comparison between the two vectors output from CSA has to be performed. In this case the shift amount is equal 56 or 55 so at least the 55 most significant bits of aligned A are sign extension. Also the sign bit in the multiplier output appears in bit 108, so we need only 109-bit magnitude comparator to include at least one sign bit in the two vectors. It is shown in Figure (4-5).

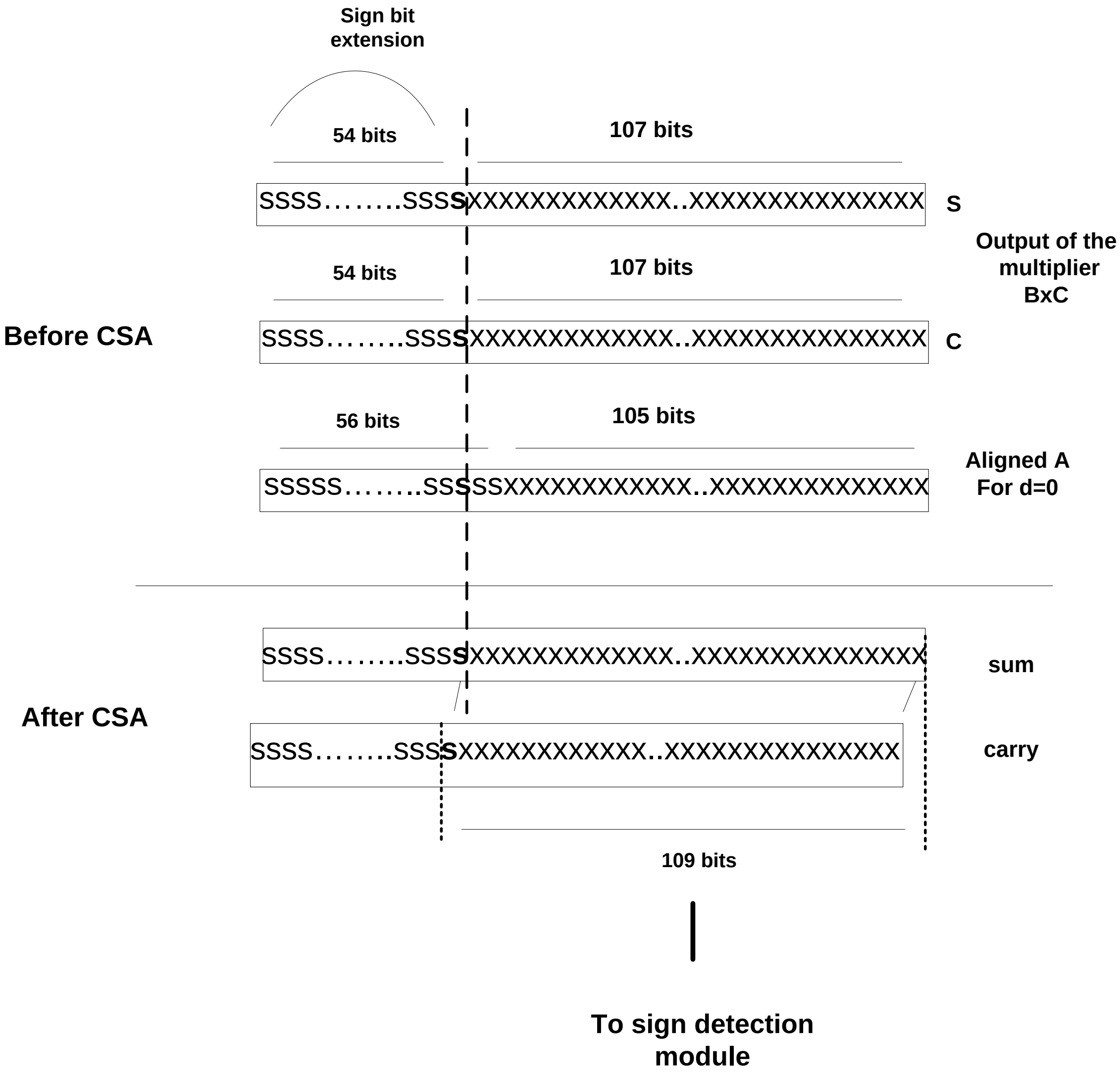

Sign bit extension
54 bits
107 bits
SSSS……..SSSSXXXXXXXXXXXXXX..XXXXXXXXXXXXXXX
S
Output of the multiplier BxC
54 bits
107 bits
Before CSA
SSSS……..SSSSXXXXXXXXXXXXXX..XXXXXXXXXXXXXXX
C
56 bits
105 bits
SSSSS……..SSSSSXXXXXXXXXXXXX..XXXXXXXXXXXXXX
Aligned A For d=0
SSSS……..SSSSXXXXXXXXXXXXXX..XXXXXXXXXXXXXXX
sum
After CSA
SSSS……..SSSSXXXXXXXXXXXXX..XXXXXXXXXXXXXXX
carry
109 bits
To sign detection module

Figure 4-5: Inputs to sign detection module to include sign extension bit.

The sign bit of aligned A is always ‘1’ when the effective operation is subtraction and the sign bits of the output of the multiplier are (0, 0), (0, 1) or (1, 0), as cited before in part 3.3.3, when adding the three sign bits in CSA the sign bits of the result will be (0, 1) or (1, 0) as shown in Table (4-1). That means in the case of equal exponents the results of CSA are a positive and a negative numbers. The magnitude comparator compares them and determines which is greater in magnitude. To make magnitude comparison, the negative number has to be converted to its positive value by getting the two’s complement value. These are done by making inversion then add one.

Table 4-1: Sign bits of the output of CSA.

Sign bit of aligned A	Sign bit of multiplier output		Sign bit of CSA output	
	S	C	Sum	Carry
1	0	0	1	0
1	1	0	0	1
1	0	1	0	1

A binary tree comparator proposed in [7] is used here. The design is based on a technique previously used for operand normalization [1]. In the first stage of our magnitude comparator, 2-bit pairs from the first operand, X, are compared in parallel with the corresponding 2-bit pairs from the second operand, Y, to determine if the two bits from X are equal , greater than, or less than the two bits from Y. If the bits pairs in X and Y are denoted as $\mathrm{X}[2\mathrm{i} + 1, 2\mathrm{i}]$ and $\mathrm{Y}[2\mathrm{i} + 1, 2\mathrm{i}]$, then the greater than $\mathrm{GT}[\mathrm{i}]$ and less than, $\mathrm{LT}[\mathrm{i}]$ signals are computed as:

$$GT[i] = (X[2i+1].\bar{Y}[2i+1]) + (X[2i+1].X[2i].\bar{Y}[2i]) + (X[2i].\bar{Y}[2i+1].\bar{Y}[2i])$$

$$LT[i] = (\bar{X}[2i+1].Y[2i+1]) + (\bar{X}[2i+1].\bar{X}[2i].Y[2i]) + (\bar{X}[2i].Y[2i+1].Y[2i])$$

(4-1)

For $(0 \le \mathrm{i} \le \mathrm{n}/2 - 1)$, where n is the operand size and n = 109 for the proposed implementation. The relationships between $\mathrm{X}[2\mathrm{i} + 1, 2\mathrm{i}]$ and $\mathrm{Y}[2\mathrm{i} + 1, 2\mathrm{i}]$ for the different values of GT[i] and LT[i] are shown in Table (4-2).

Table 4-2: 2-Bit Magnitude Comparison.

GT[i]	LT[i]	Description
0	**0**	X[2i+1: 2i] = Y[2i+1: 2i]
0	**1**	X[2i+1: 2i] < Y[2i+1: 2i]
1	**0**	X[2i+1: 2i] > Y[2i+1: 2i]
1	**1**	Invalid

In subsequent comparator stages, a similar process is used, but $\mathrm{X[i]}$ signal is replaced by $\mathrm{GT[i]_j}$ signals and $\mathrm{Y[i]}$ signal is replaced by $\mathrm{LT[i]_j}$ signals, where the subscript ”j” indicates comparator stage j. Furthermore, since $\mathrm{GT[i]_j}$ and $\mathrm{LT[i]_j}$ cannot both be one simultaneously, the greater than and less than equations can be simplified to:

$$\mathrm{GT[i]_{j+1}} = \mathrm{GT[2i+1]_j} + \mathrm{GT[2i]_j}.\overline{\mathrm{LT}}\mathrm{[2i+1]_j} \qquad (4\text{-}2)$$

$$\mathrm{LT[i]_{j+1}} = \mathrm{LT[2i+1]_j} + \overline{\mathrm{GT}}\mathrm{[2i+1]_j}.\mathrm{LT[2i]_j}$$

For $1 \le \mathrm{j} \le (\log_2 \mathrm{n} - 1)$ and $0 \le \mathrm{i} \le (\mathrm{n}/2^{\mathrm{i}+1} - 1)$ after a total of $\log_2(\mathrm{n})$ comparator stages, in our implementation n = 109 so 7 stages are needed. A block diagram of 109-bit tree comparator is shown in Figure (4-6).

The output of the sign detection module is the complement signal, *comp* and it equals one if d is positive (the most significant bit is one), but in case of equal exponents it takes the sign of the input which has greater magnitude. When the comparator output is greater than $(\mathrm{GT} = 1$ and $\mathrm{LT} = 0)$ the *comp* signal takes the sign of the sum vector S and if it is less than the *comp* signal takes sign of the carry vector C.

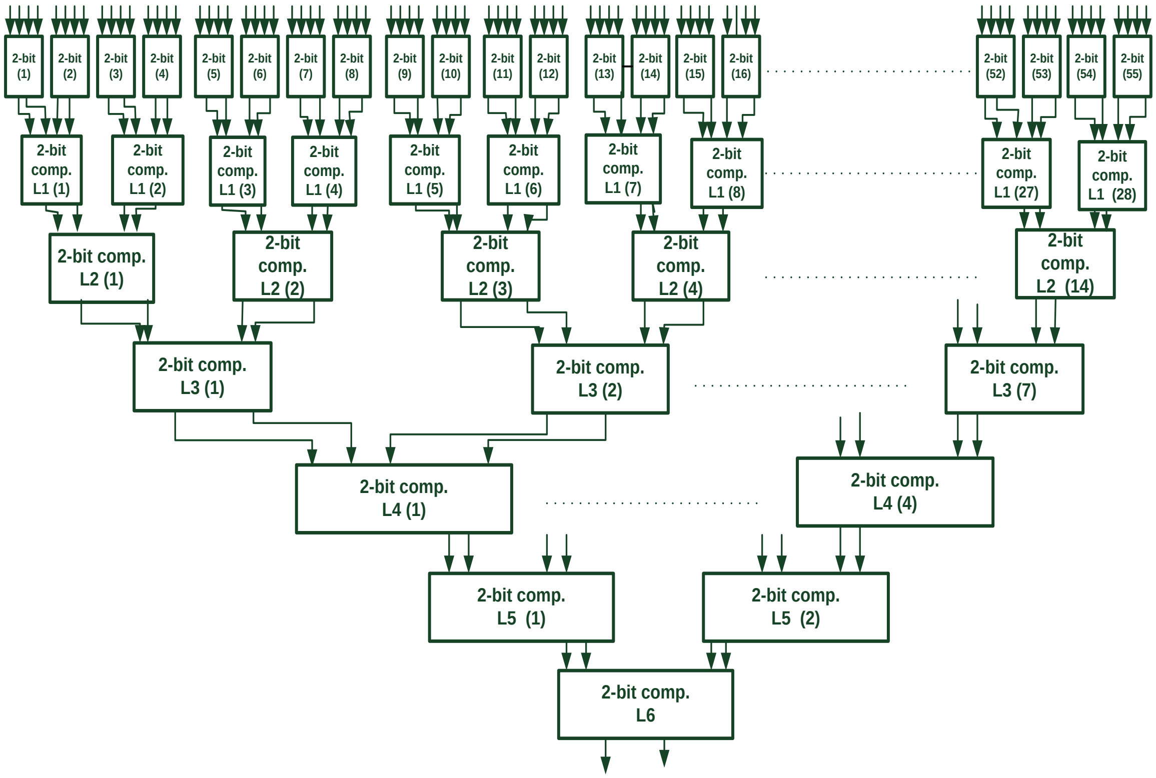


Figure 4-6: A 109-bit tree comparator.

The complement signal, comp can be deduced in the following equation.

$$\text{comp} = \left(\left(\text{GT}.\,\text{S}_{\text{msb}} + \text{LT}.\,\text{C}_{\text{msb}}\right).\,\text{E} + \text{d}_{\text{msb}}\,.\,\overline{\text{E}}\right).\,\text{sub} \qquad (4\text{-}3)$$

The E signal indicates the case of equal exponents. And sub signal indicates effective subtraction.

To eliminate the need of the carry propagate adder used in two's complement conversion in order to decrease the delay of the sign detection module a new scheme is proposed. This scheme uses the magnitude minus one (one's complement) value in comparison instead of the magnitude (two's complement) of the negative number.

The comparison using one’s complement (magnitude-1) makes error in two values. It can be shown using this example, assume a negative number (-5) by comparing it with any positive numbers two general cases appear, see Figure (4-7).

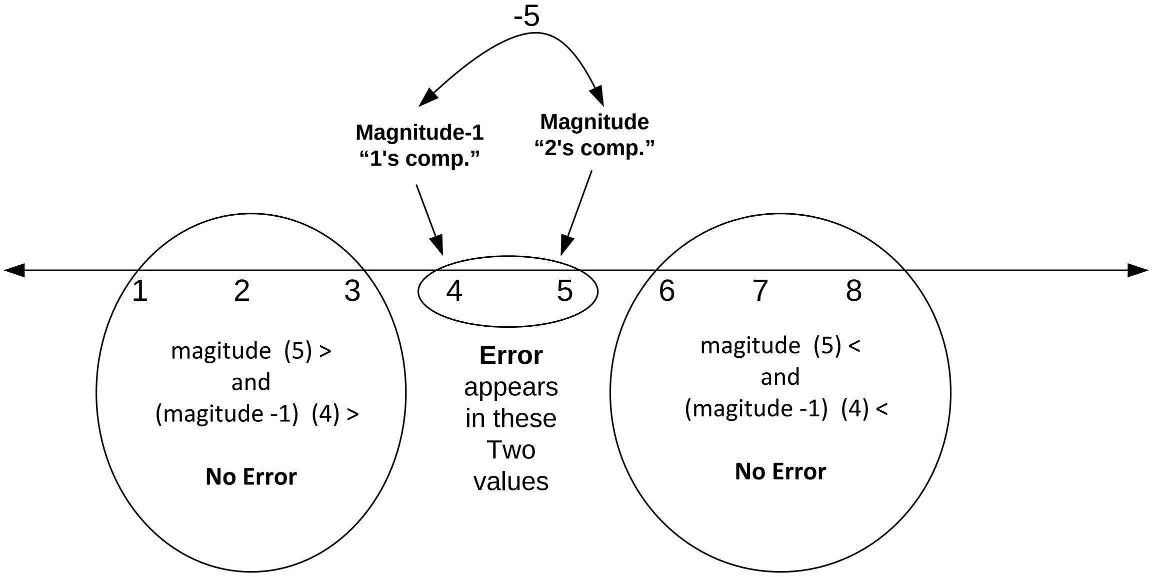


Figure 4-7: Comparison (magnitude) and (magnitude-1) of -5 with any positive number.

1- For positive numbers which are less and greater than both magnitude (5) and magnitude minus one (4) , (1,2,3) and (6,7,8,..) respectively , the result be the same whether we use magnitude or magnitude minus one in comparison and no error occurs.
2- There is an error when comparing -5 with its magnitude and magnitude minus one, 5 and 4 respectively.

This error can be overcome by the following discussion. When comparing the negative number -5 with +4 using the conventional comparison which use magnitude of -5 (+5) the comp signal equals 1 by using eq. (4-3), but when using magnitude minus one (+4) instead the equate signal equals one (GT=0 and LT=0).

We can overcome this error by making comp=1 if the output of the comparator is equal signal.

When comparing the negative number -5 with its magnitude (+5) the conventional output of the comparator has to be equal and comp=0 by using eq. (4-3). In our case as we compare +4 with +5 the comp signal takes the sign of greater magnitude number (+5) so comp=0 as in conventional comparison so no modification is needed.

A complete block of sign detection module is shown in Figure (4-8). The eq. (4-3) can be modified to include equal signal to correct the error.

$$comp = \left((GT.S_{msb} + LT.C_{msb} + \overline{GT}.\overline{LT}).E + d_{msb}.\bar{E}\right).sub \tag{4-4}$$

4.2.3 Add/Round module

The add/round module implements only the four round modes defined in the standard (IEEE 754-1985), round to nearest/ even (RN), round to + ∞ (Rp) , round to -∞ (Rn) and round to zero (RZ). The round to nearest/even mode is obtained by rounding to nearest/up first then the least -significant bit (LSB) is corrected [22]. Round to nearest/up produces the same result as round to nearest/even except when a tie occurs; then, the sticky bit identifies the tie case and the correct rounded to nearest/even result is obtained by forcing the LSB to 0.

The add/round module for the proposed FMA differs from the corresponding modules for floating-point addition [14], [27]. The difference is that now there can be a carry propagating from the 108 least significant bits to the 53 more-significant bits. This carry has to be added with the rounding bits to obtain the 53-bit rounded result. On the other hand, in floating point addition, as both operands are 53-bit wide and one of them is aligned, there is no carry from the least-significant part, which corresponds to the positions that are right shifted out in the alignment.

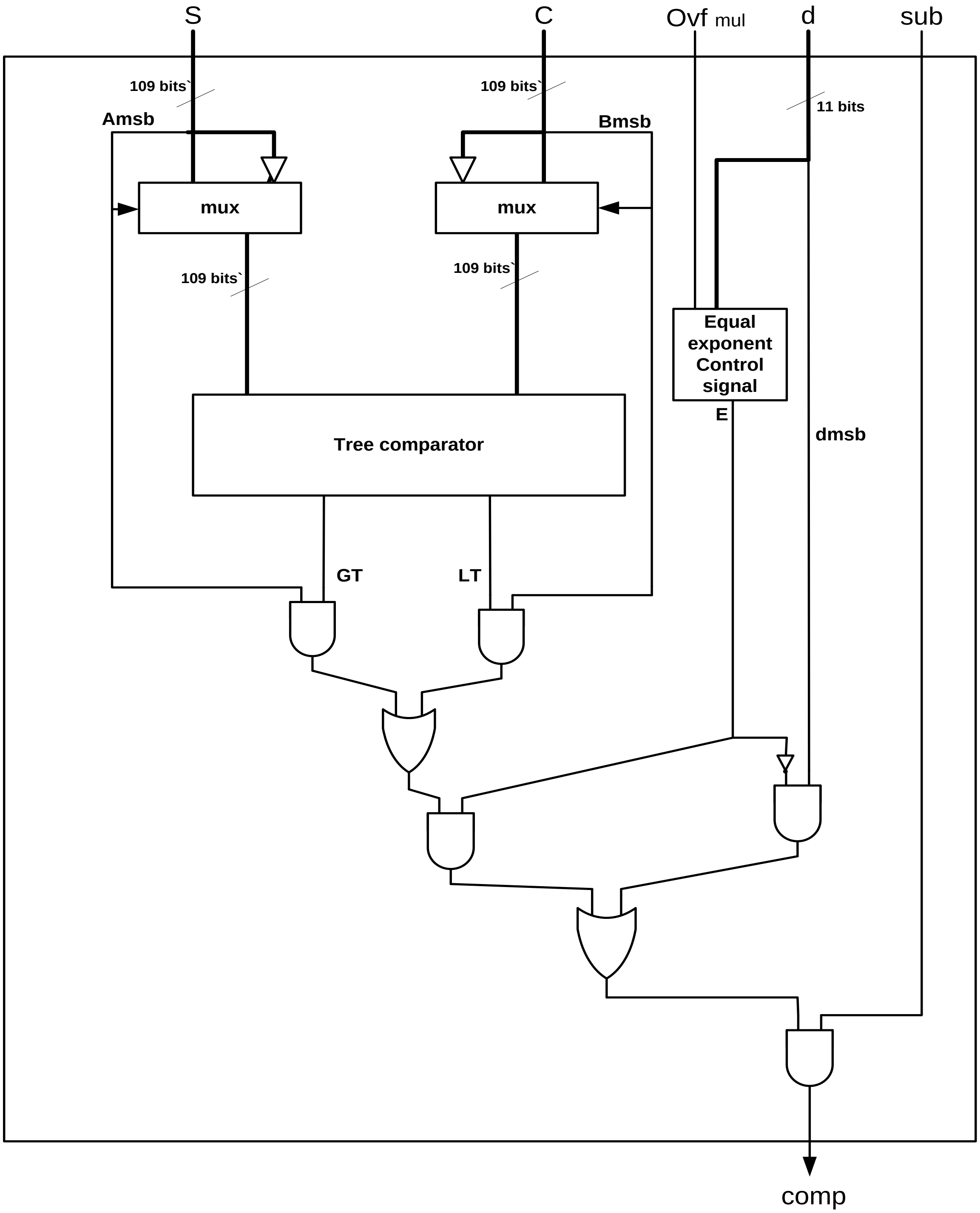


Figure 4-8: A complete block of sign detection module.

The fused multiply add operation is similar to floating-point multiplication in rounding algorithm where the rounded output is computed from a carry-save representation. From the three schemes that were proposed in [22], [26], [32],[33] for the add/round module in multiplication, we choose the one called The YZ rounding algorithm described in [26],[33] and edit it to be used in proposed FMA.

The add/round module accepts two numbers in carry save form. When these numbers are added there are two cases appearing, see Figure (4-9).

1- An error occurs in LZA, it means there is no overflow ($ovf\ = 0$). The mantissa from bit 1 to bit 53 and the round bit is bit 54.

2- No error occurs, it means there is an overflow ($ovf\ = 1$). The mantissa is taken from bit 0 to bit 52 and bit 53 will be the round bit. Also a right shift is needed and the exponent of the result is increased by one.

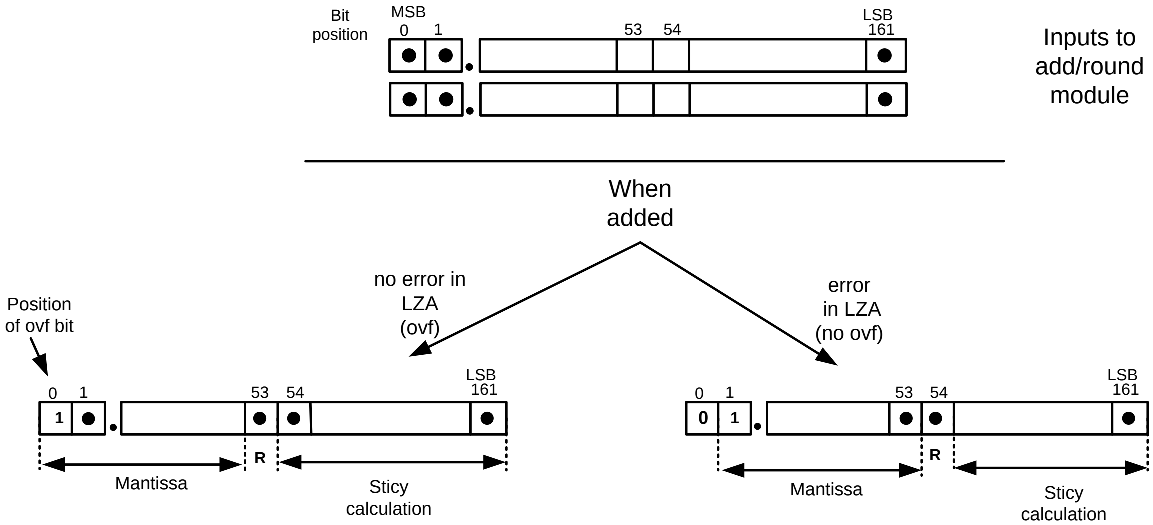


Figure 4-9: cases appear due to one bit error in lZA.

As in the YZ algorithm [26] the sum and carry-strings are divided into a high part $[0: 54]$ and a low part $[55: 161]$from the low part the carry bit $c[54]$ and the sticky bit are computed. The higher part is input to a line of half adders then is divided into two parts: positions $[51: 54]$ are added with the carry bit $c[54]$ and the rounding bits created from round decision block. The position of the rounding bit depends on the value of the overflow bit so the processing of this part is split into two paths; one working under the assumption that the result will not overflow (i.e. less than 2), and the other path working under the assumption that the result will overflow. Positions $[0\!:50]$ are fed into the dual adder, which outputs the sum Y0 which assumes no carry is propagated from least significant bits $[51\!:54]$ and the incremented sum Y1 which assumes a carry is propagated. The correct rounded result is chosen by Multiplexers and control signals computed in select decision block. The block diagram of the add/round module is shown in Figure (4-10).

4.2.3.1Select decision and bits selection

The select decision box has three outputs $\mathrm{O},$ ovf and inc signals. O Signal decides if the case before rounding is an overflow or no overflow and selects between Zo and Zn to produce Z vector. If the case before rounding is an overflow it will be the same after rounding in both $Y0$ and $Y1$ except when all bits of $Y1$ are 1's which causes a carry propagation till the most significant bit of $Y1[0]$ (bit of overflow) and set $Y1[-1]$ to be one, see Figure(4-11) (a), similarly in the case of no overflow it will remain the same after rounding in both $Y0$ and $Y1$ except in case of all ones in $Y1$ where the overflow bit $Y1[0]$ will be one, it is shown in Figure (4-11)(b).

The O signal can be computed from the following equation

$$\mathrm{O} = \mathrm{Y0}[0] + \mathrm{Y1}[0] + \mathrm{Y1}[-1] + \left(\overline{\mathrm{Y1}[0].\overline{\mathrm{Y1}}[1].\overline{\mathrm{Y1}}[2]\ldots\overline{\mathrm{Y1}}[50]}\right) \qquad (4\text{-}10)$$

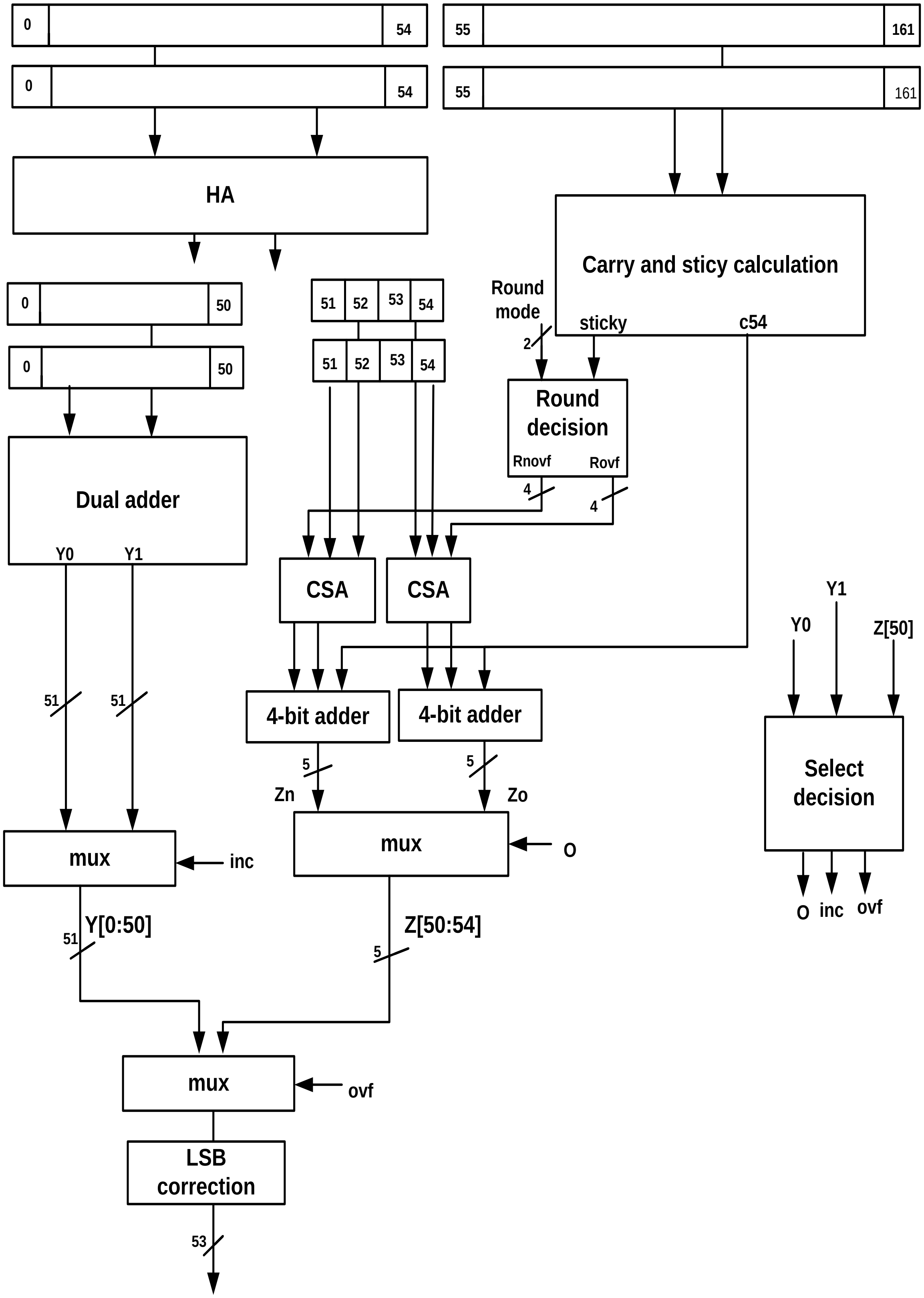


Figure 4-10: Block diagram of the add/round module.

The inc signal selects between outputs from dual adder, the sum $\mathrm{Y0}\ [0:50]$ and the incremented sum $\mathrm{Y1}\ [0:50]$ to produce Y vector.

It can be determined by the most significant bit of the Z vector. The inc signal is computed by eq. (4-11).

$$\mathrm{inc} = \mathrm{Z}[50] \qquad (4\text{-}11)$$

The ovf signal determines the correct rounded result, Figure (4-12), and updates the exponent. It decides if the 50 most significant bits are $\mathrm{Y}\ [0:49]$ or $\mathrm{Y}\ [1:50]$ and consequently decides the 3 least significant bits. It can be done by checking the most significant bit of Y. ovf signal is computed as follows:

$$\mathrm{ovf} = \mathrm{Y}[0] \qquad (4\text{-}12)$$

4.2.3.2Sticky bit calculation and round decision

The sticky bit which is needed to perform the correct rounding is composed of two components: $\mathrm{st1}$, obtained by ORing the bits shifted out during the alignment of A, and $\mathrm{st2}$, obtained from the add/round module. The final sticky bit is obtained by OR-ing both of them $\mathrm{st} = \mathrm{st1} + \mathrm{st2}$. To obtain the partial sticky bit $\mathrm{st2}$, typically, the lower 107 bits of the sum and carry vectors are summed and OR'ed together. However, as proposed in [33] the sticky bit can be computed directly from carry save form without the need for an adder to generate the sum. We define:

$$\mathrm{P_i} = \mathrm{S_i} \oplus \mathrm{C_i} \qquad (4\text{-}5)$$

$$\mathrm{h_i} = \mathrm{S_i} + \mathrm{C_i}$$

$$\mathrm{t_i} = \mathrm{P_i} \oplus \mathrm{h_{i+1}}$$

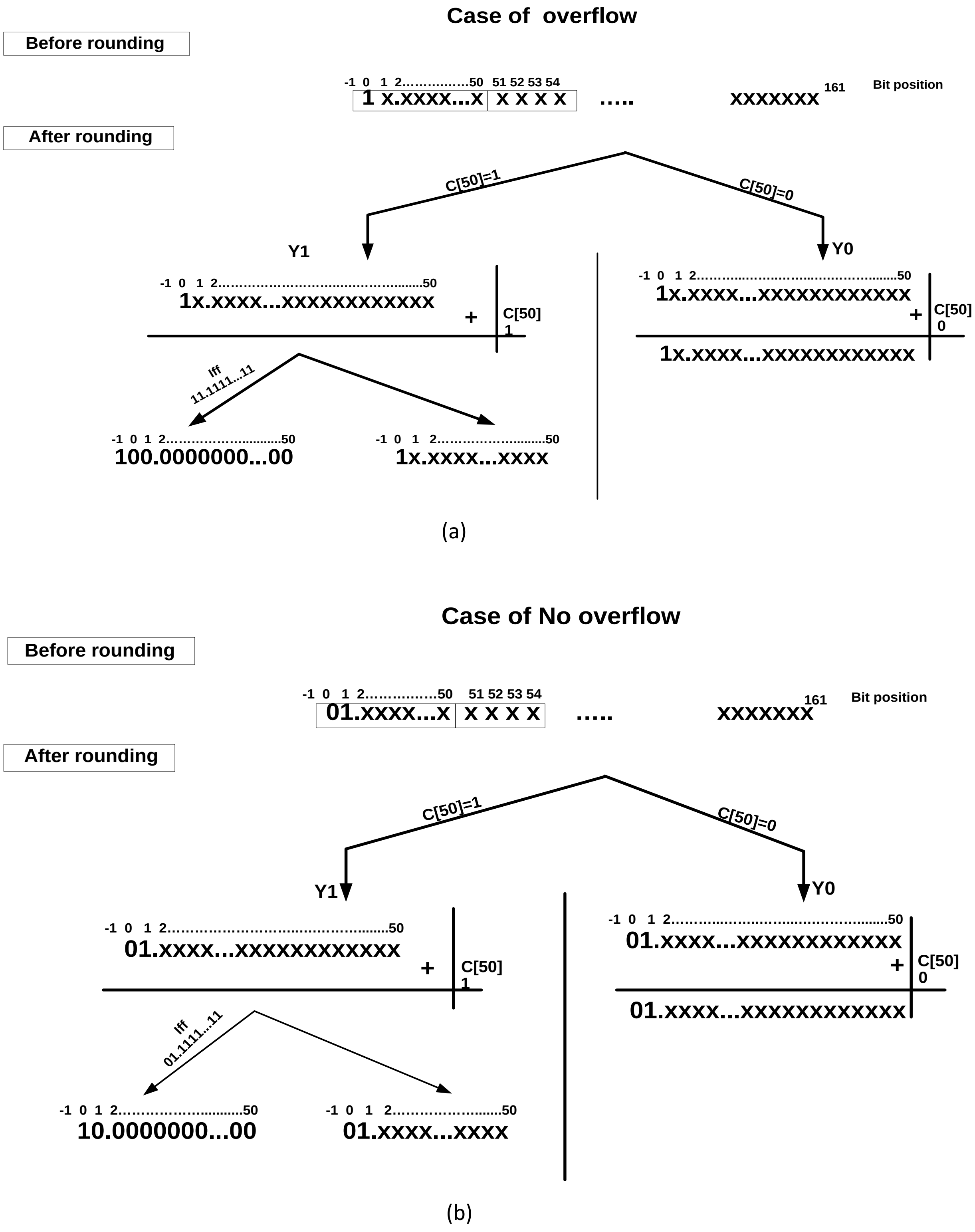


Figure 4-11: The rounded result in case of (a) overflow, (b) No overflow.

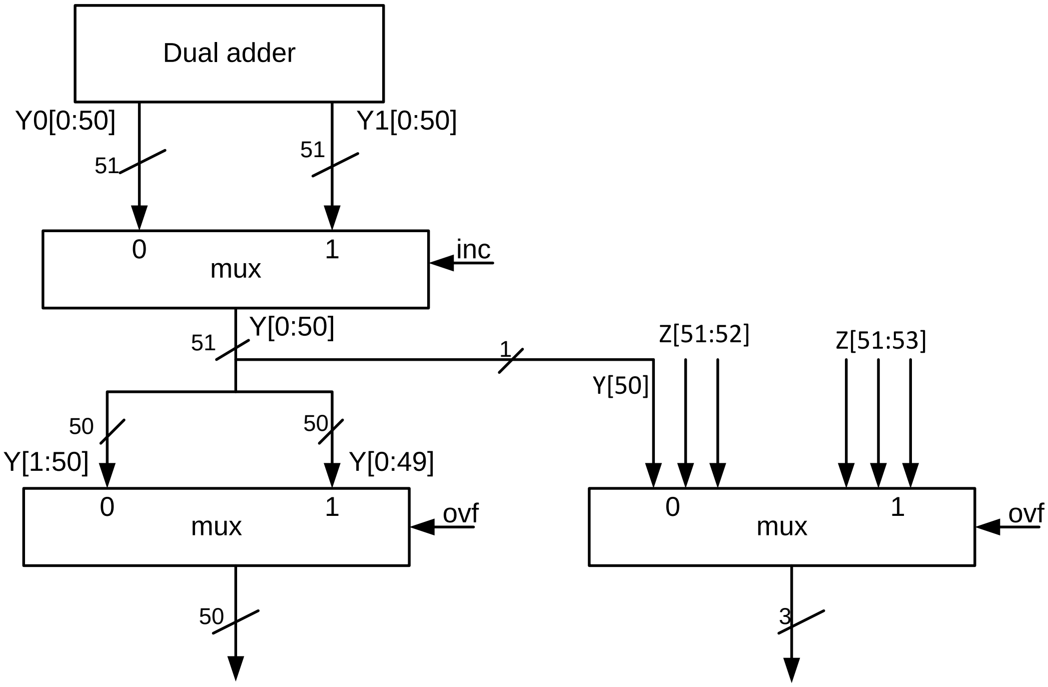


Figure 4-12: Selection of rounded result using $\boldsymbol{inc}$ and $\boldsymbol{ovf}$ signals.

Where $\mathrm{S_i}$ and $\mathrm{C_i}$ are the sum and carry output from CSA.$55 \leq \mathrm{i} \leq 161$, The sticky bit is computed directly by the eq. (4-6).

$$\mathrm{st2} = \mathrm{t_{55}} + \mathrm{t_{56}} + \mathrm{t_{57}} + \cdots + \mathrm{t_{161}} \tag{4-6}$$

For the overflow path bit 54 should be included in sticky calculation, eq. (4-7) shows how to compute sticky bit in case of overflow

$$\mathrm{t_{54}} = \mathrm{P_{54}} \oplus \mathrm{h_{55}} \tag{4-7}$$

$$\mathrm{st2_{ovf}} = \mathrm{t_{54}} + \mathrm{st2}$$

The final sticky bits for both the overflow and no overflow cases are input now to the round decision block with the rounding mode to determine the number to be added to the right position of the round bit. We have four rounding modes so need two bits (r [0] r [1]) to identify them; it is shown in Table (4-3). For rounding to zero, RZ, the mantissa is truncated and no bits are added. In case of rounding to nearest even, RN, a 1 is always added to the round bit then a correction is made in case of a tie case. For rounding toward +∞, Rp, in case of positive result if all the bits to the right of the LSB of the desired result are 0 then the result is correct. If any of these bits are a 1, (i.e. R=l or sticky=l) then a 1 should be added to the LSB of the result. If the result is negative it should be truncated. For rounding toward -∞, Rn, the exact opposite holds.

Table 4-3: Rounding modes.

Rounding bits		**Rounding**
r[0]	**r[1]**	**mode**
0	0	RZ
0	1	Rp
1	0	Rn
1	1	RN

Rounding toward +∞ (Rp) can be considered two separate modes ,rounding to infinity (RI) in case of positive numbers and round to zero (RZ) if the number is negative. This can applied in rounding toward -∞ also. There are three general rounding modes are considered in this thesis, rounding to nearest even (RN), rounding to zero (RZ) and rounding to infinity (RI). They can be generated from rounding bits (r [0] r [1]) as shown in eq. (4-8).

$$RN = r[0].r[1]$$

$$RI = (\overline{sign}.\bar{r}[0].r[1]) + (sign.r[0].\bar{r}[1]) \qquad (4\text{-}8)$$

$$RZ = (\overline{RN + RI})$$

The round decision block generates two bits Rd and Rd1 to determine numbers to be added to the round bit. Rd equals one if the mode is RN or RI and

Rd1 equals one in case of RI mode and sticky bit is set. The logic equation of Rd and Rd1 is shown in eq. (4-9).

$$\mathrm{Rd} = \mathrm{RN} + \mathrm{RI}$$

$$\mathrm{Rd1} = \mathrm{RI.St} \tag{4-9}$$

The two bits are added first in HA then the results are added to bits [53:54] of the sum and carry vectors in case of no overflow path and to bits [52:53] in case of overflow path using CSA. It is shown in Figure (4-13).

If the mode is RN one is added to the round bit ($Rd = 1, Rd1 = 0$) and when the mode is RI with sticky bit is not set and round bit is set ($Rd = 1, Rd1 = 0$) a one is added to the round bit and a carry is propagated to the LSB. If sticky bit is set then Rd1=1 and a carry is generated and propagated to the LSB. In case of RZ, $Rd = 0$ and $Rd1 = 0$.

4.2.3.3 LSB correction

A LSB correction is needed in case of round to nearest /even. It was stated earlier that round to nearest /even is done by rounding to nearest/up which produces exactly the same result as round to nearest/even except when a tie occurs. A tie can only occur when the result is exactly halfway between two numbers of representable precision.

The bit to be added to the round bit (R) for correct round to nearest/even is based upon the least significant (L), round (R), and sticky (S) bits as shown in Table (4-4). In contrast, round to nearest up assumes that the bit to be added to the R bit for correct rounding is always one. The only case where round to nearest/up mode produces a different result from round to nearest/even mode is shown in Table (4-4) (bold raw). In this case round to nearest up changes only the L bit from a 0 to a 1 where no carry propagates after that. While round to nearest even leaves the L bit unchanged ($L = 0$). This means that the correct round to nearest even result can be obtained from the round to nearest up result by restoring the L bit to a 0.

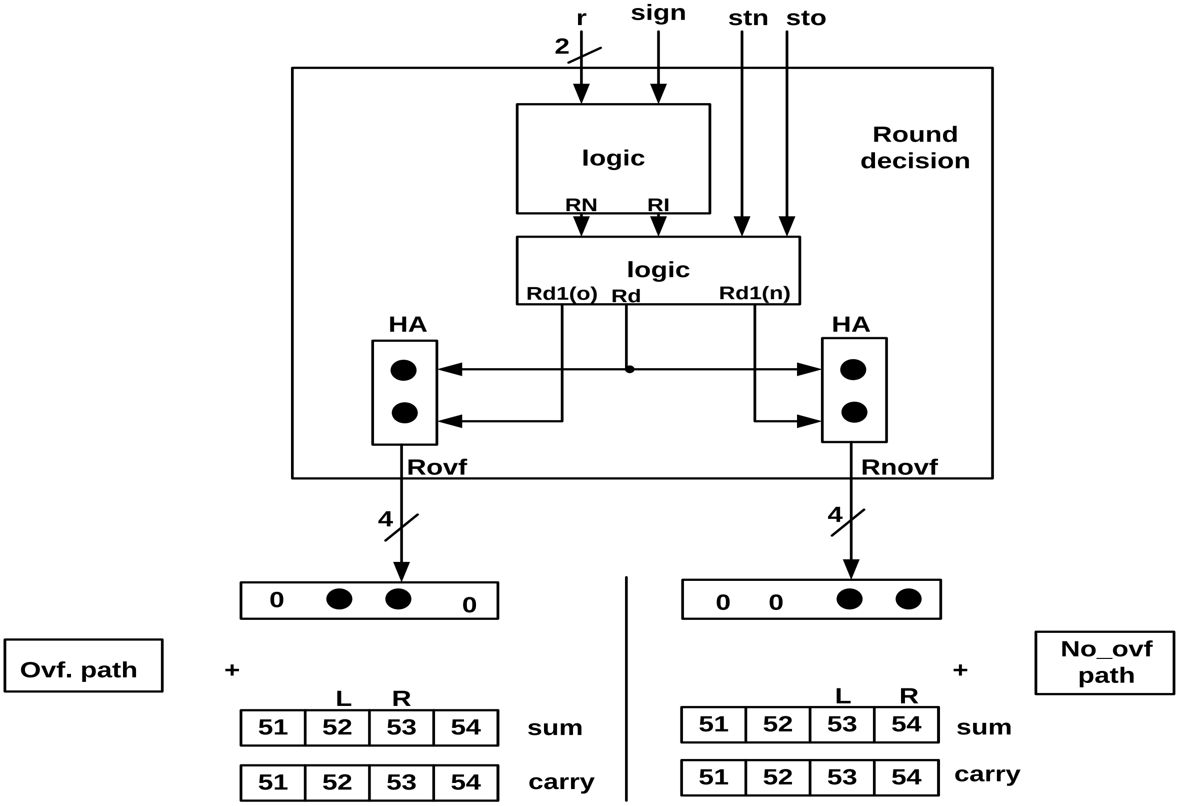


Figure 4-13: Block diagram of round decision

Table 4-4: Round to Nearest/even versus Round to Nearest/up.

Before rounding			**Bit added to R**		**L after rounding**	
L	R	S	Nearest/even	nearest/up	Nearest/even	nearest /up
X	0	0	0	1	X	X
X	0	1	0	1	X	X
0	**1**	**0**	**0**	**1**	**0**	**1**
1	1	0	1	1	0	0
X	1	1	1	1	$\bar{X}$	$\bar{X}$

In a LSB correction block sticky bits and round bits are needed to be computed to know if there is a tie case. The sticky bits are already computed in sticky calculation part as cited before. The round bits are not available and can be computed only after carry propagate addition of positions $[51:54]$ of sum and carry vectors then take the positions 53 and 54 of the result as the round bits for overflow and no overflow paths respectively.

Round bits can be computed from a carry save form without using carry propagate adder by using eq.'s (4-10) and (4-11). It is equivalent to compute the result of adding bits 54 and 53 of sum and carry vectors using propagate and generate signals as used in prefix adder. Once round bits are computed a tie case can be determined if $\mathrm{R} = 1$ and $\mathrm{S} = 0$ then the LSB is forced to zero using logic gates, see Figure (4-12).

$$P_{54} = sum[54] \oplus carry[54] \tag{4-10}$$
$$P_{53} = sum[53] \oplus carry[53]$$
$$g_{54} = sum[54] \, . carry[54]$$

$$R_{novf} = P_{54} \oplus C_{54} \tag{4-11}$$
$$R_{ovf} = P_{53} \oplus (g_{54} + P_{54} \, . C_{54})$$

4.2.4 Anticipated part of the adder

As referred before in Lang/Bruguera algorithm [10] there are some approaches used to decrease the delay of FMA. The first approach, cited before, is to overlap the operation of the LZA and the operation of normalization shifter. However, despite the overlap, the operation of the normalization shifter cannot begin at the same time as the LZA. The normalization shifter has to wait for computation of first digit of LZA (S1).

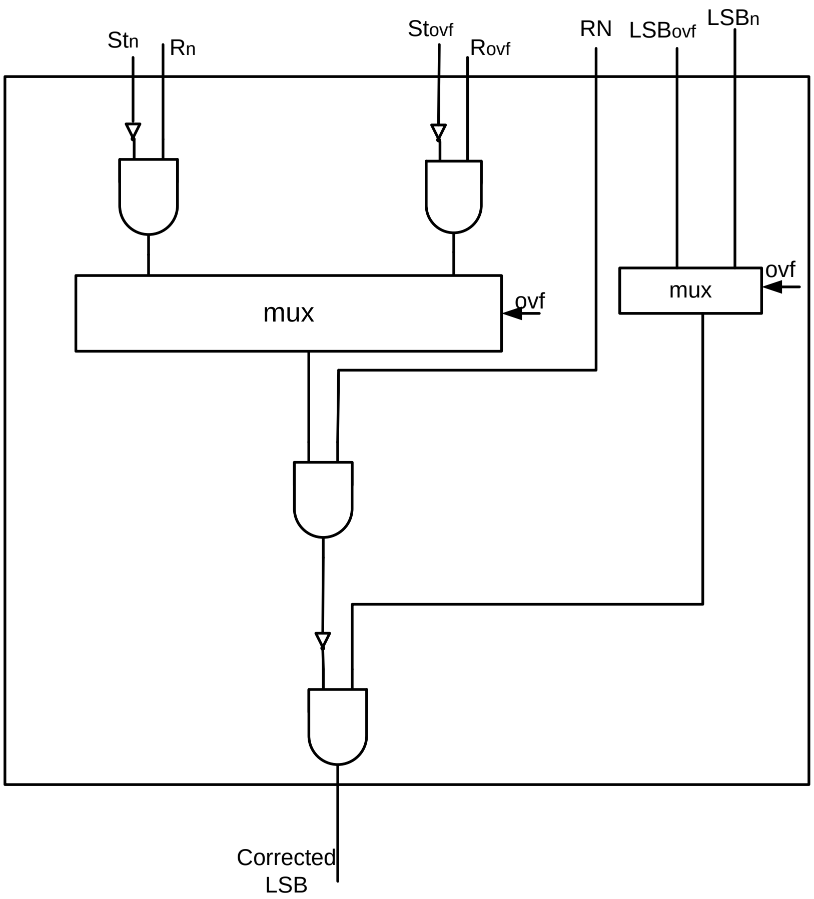


Figure 4-14: LSB correction block.

The second approach is to anticipate part of dual adder in the time gap between LZA and the normalization shifter. The anticipated part of the dual adder may be the half adder (HA) and the generation of propagate and generate signals.

As shown in part 4.2.3, a row of HA is used in higher part of data $[0:54]$ to limit the maximum value in bits 51 to 54 and assure that only one carry is produced to bit 50 during the rounding and the assimilation of the lower part. To anticipate HA before the normalization operation it is obligatory to include HA for the 162 bits because before the normalization the position of most significant bit of the result is unknown.

The HA is duplicated, with and without inverted inputs, to make bit inversion of sum and carry words if the comp signal is one (negative result) to avoid additional delay. However, the sum word is the same both with inverted inputs and without inverted inputs as shown in table (4-6); then, it is only necessary to replicate the calculation of the carry word.

The two 1s needed to complete the two's complement of the inputs of the HA have to be added in the least-significant position of these inputs. The first 1 is incorporated as the least significant bit of the carry word and the second 1 is added in the add/round module.

Some part of the dual adder can be anticipated also before the normalization; moreover, the anticipated part of the dual adder has to be replicated, considering as input the inverted and non-inverted HA outputs.

Table 4-5: Truth table of H.A with inverted and non inverted inputs.

Inputs to HA		**Outputs of HA**		**Inverted inputs to HA**		**Outputs due to inverted inputs**	
A	B	S	C	$\overline{A}$	$\overline{B}$	S	C
0	**0**	**0**	**0**	**1**	**1**	**0**	**1**
0	**1**	**1**	**0**	**1**	**0**	**1**	**0**
1	**0**	**1**	**0**	**0**	**1**	**1**	**0**
1	**1**	**0**	**1**	**0**	**0**	**0**	**0**

It seems reasonable that, at most, the calculation of the bit carry generate gi and bit carry propagate pi can be anticipated. However, some more logic could be placed before the normalization; for example, the calculation of carry generated and carry propagated signals of groups of 2 or 4 bits, but it will make the design of the add/round module more complex, also the anticipation of the first level only of the dual adder does not imply changes in its design due to the effect of the shift over the already calculated carry generate and carry-propagate signals, G and P. Figure (4-13) shows the anticipated part of dual adder.

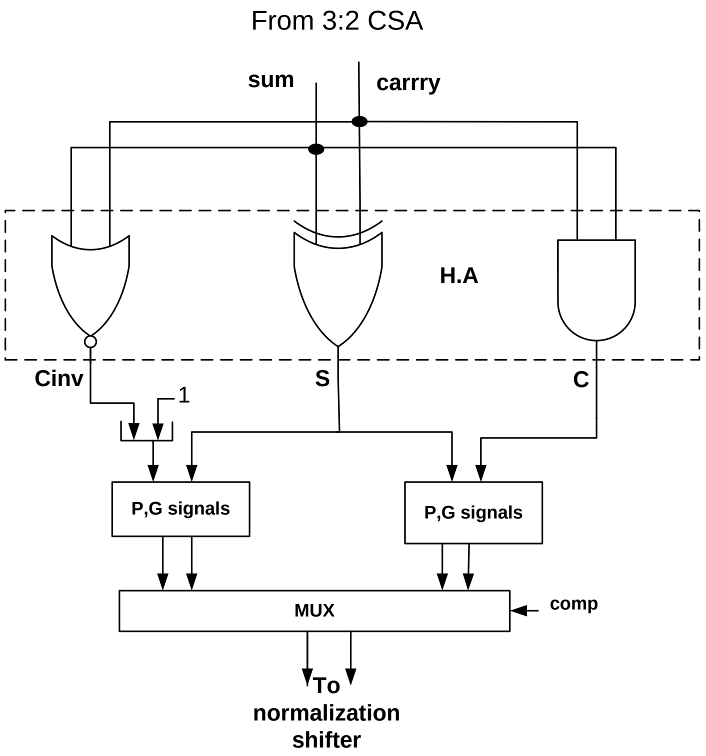


Figure 4-15: Anticipated part of HA and bit carry propagate and generate signals selection.

4.3 Conclusion

We discuss in this chapter design of blocks used to implement the proposed fused multiply add unit like sign detection and add round modules. Sign detection module produces the *comp* signal which is used after that to select inputs to the normalization shifter. These inputs are needed to be available as soon as possible before shift amount of normalization shifter is calculated. Because of this reason we use a new scheme to implement sign detection module to reduce its latency. We use magnitude minus one (1’s complement) instead of magnitude (2’s complement) in magnitude comparator to eliminate the need of carry propagate adder.

Chapter 5：Simulation Verification and Results

5.1 Simulation verification

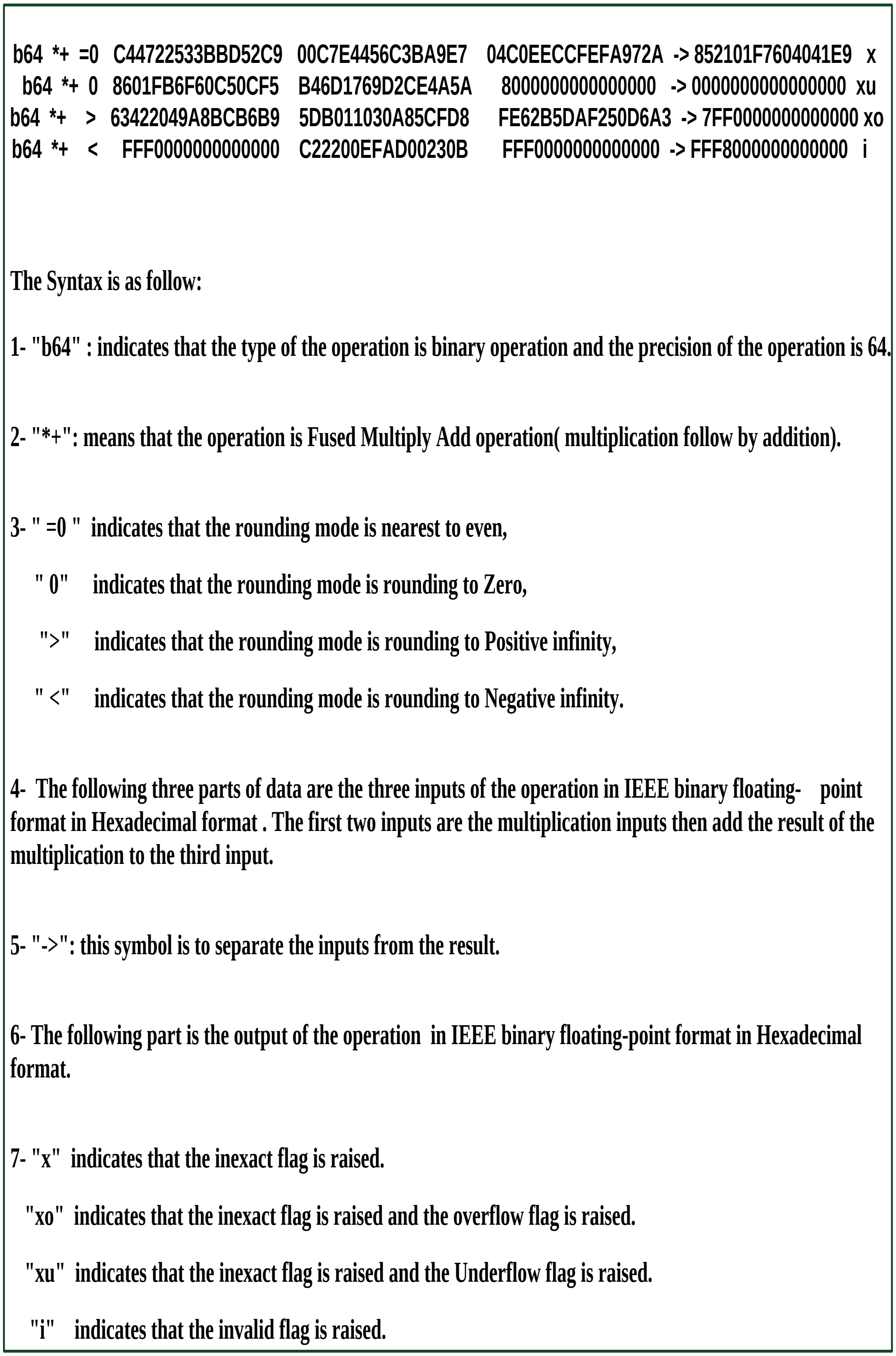
Both the proposed and the basic architecture of the fused multiply add (IBM/6000) were implemented in the Verilog hardware description language, some of them are shown in appendix. ModelSim10.0c is a used to compile Verilog codes and to simulate them. The simple way to test the design is to write a test bench that exercises various features of the design. Test vectors are used, some examples of these test vectors and their syntax are shown in Figure (5-1), these test vectors are divided into two text files one for the inputs of the design under test (DUT) and the other is for the expected (right) results.

The input text file is read by test bench using test bench verilog syntax $readmemh then the output is written to a text file using test bench verilog syntax $fopen, $fdisplay and $fclose. The output text file is now compared to the already exist output file (right outputs) using Active File Compare program. An example of a comparison is shown in Figure (5-2).

About 12,800 test vectors are used. A part from them includes special cases which raise overflow, underflow and invalid operation flags. 100% accuracy was reported for all designs for normalized and denormalized numbers. An example test run is shown in Figure (5-3).

5.2 Simulation Results

The two implemented architectures of fused multiply add are synthesized, placed and routed for Cyclone II FPGA device using Quartus II 9.1.



```
b64  *+  =0   C44722533BBD52C9   00C7E4456C3BA9E7   04C0EECCFEFA972A  -> 852101F7604041E9   x
  b64  *+  0   8601FB6F60C50CF5   B46D1769D2CE4A5A     8000000000000000   -> 0000000000000000  xu
b64  *+   >   63422049A8BCB6B9   5DB011030A85CFD8     FE62B5DAF250D6A3  -> 7FF0000000000000 xo
b64  *+   <    FFF0000000000000   C22200EFAD00230B     FFF0000000000000  -> FFF8000000000000   i
```

The Syntax is as follow:

1- "b64" : indicates that the type of the operation is binary operation and the precision of the operation is 64.

2- "*+": means that the operation is Fused Multiply Add operation(multiplication follow by addition).

3- " =0 " indicates that the rounding mode is nearest to even,

" 0"   indicates that the rounding mode is rounding to Zero,

">"   indicates that the rounding mode is rounding to Positive infinity,

" <"   indicates that the rounding mode is rounding to Negative infinity.

4- The following three parts of data are the three inputs of the operation in IEEE binary floating-   point format in Hexadecimal format . The first two inputs are the multiplication inputs then add the result of the multiplication to the third input.

5- "->": this symbol is to separate the inputs from the result.

6- The following part is the output of the operation in IEEE binary floating-point format in Hexadecimal format.

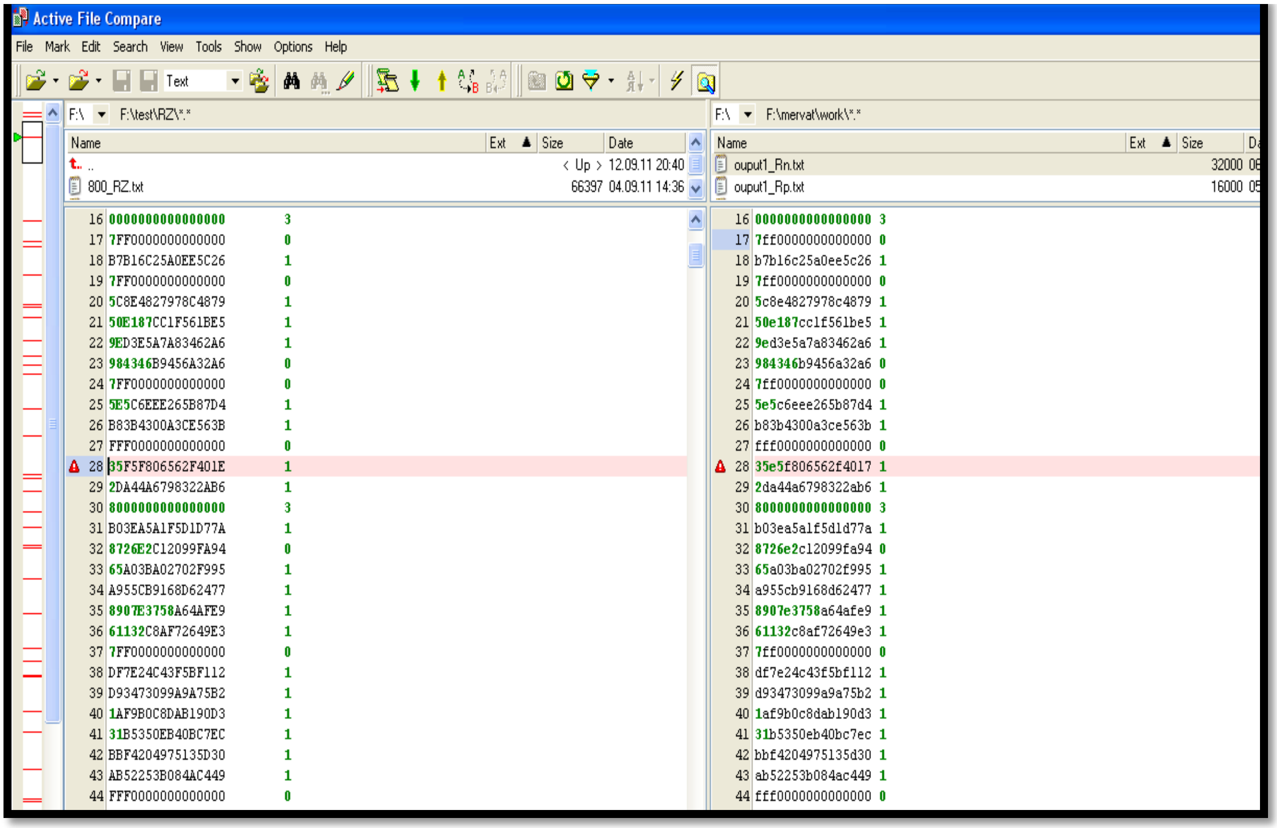
7- "x" indicates that the inexact flag is raised.

"xo" indicates that the inexact flag is raised and the overflow flag is raised.

"xu" indicates that the inexact flag is raised and the Underflow flag is raised.

"i"   indicates that the invalid flag is raised.

Figure 5-1: Test vectors examples and their syntax.

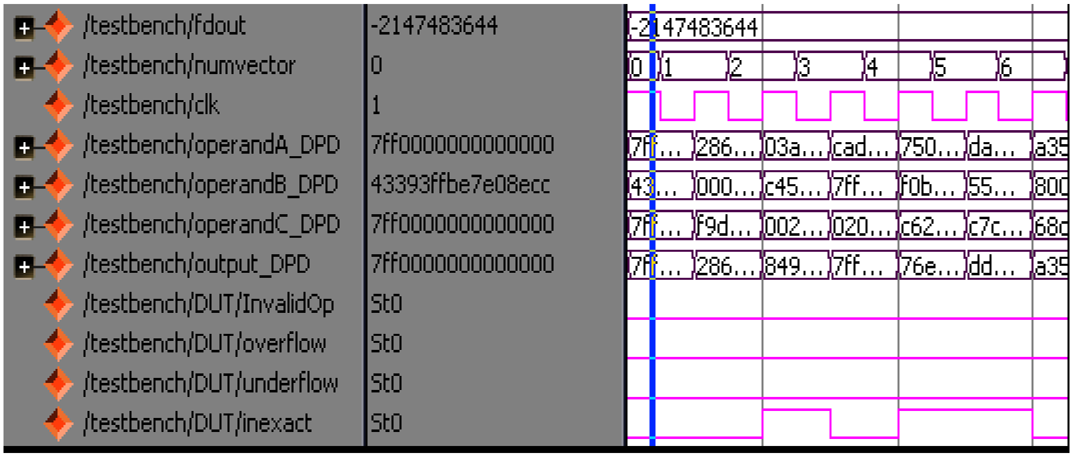


```
800_RZ.txt                       66397 04.09.11 14:36
```

```
ouput1_Rn.txt                    32000
ouput1_Rp.txt                    16000
```

```
16 0000000000000000     3        16 0000000000000000 3
17 7FF0000000000000     0        17 7ff0000000000000 0
18 B7B16C25A0EE5C26     1        18 b7b16c25a0ee5c26 1
19 7FF0000000000000     0        19 7ff0000000000000 0
20 5C8E4827978C4879     1        20 5c8e4827978c4879 1
21 50E187CC1F561BE5     1        21 50e187cc1f561be5 1
22 9ED3E5A7A83462A6     1        22 9ed3e5a7a83462a6 1
23 984346B9456A32A6     0        23 984346b9456a32a6 0
24 7FF0000000000000     0        24 7ff0000000000000 0
25 5E5C6EEE265B87D4     1        25 5e5c6eee265b87d4 1
26 B83B4300A3CE563B     1        26 b83b4300a3ce563b 1
27 FFF0000000000000     0        27 fff0000000000000 0
28 35F5F806562F401E     1        28 35e5f806562f4017 1
29 2DA44A6798322AB6     1        29 2da44a6798322ab6 1
30 8000000000000000     3        30 8000000000000000 3
31 B03EA5A1F5D1D77A     1        31 b03ea5a1f5d1d77a 1
32 8726E2C12099FA94     0        32 8726e2c12099fa94 0
33 65A03BA02702F995     1        33 65a03ba02702f995 1
34 A955CB9168D62477     1        34 a955cb9168d62477 1
35 8907E3758A64AFE9     1        35 8907e3758a64afe9 1
36 61132C8AF72649E3     1        36 61132c8af72649e3 1
37 7FF0000000000000     0        37 7ff0000000000000 0
38 DF7E24C43F5BF112     1        38 df7e24c43f5bf112 1
39 D93473099A9A75B2     1        39 d93473099a9a75b2 1
40 1AF9B0C8DAB190D3     1        40 1af9b0c8dab190d3 1
41 31B5350EB40BC7EC     1        41 31b5350eb40bc7ec 1
42 BBF4204975135D30     1        42 bbf4204975135d30 1
43 AB52253B084AC449     1        43 ab52253b084ac449 1
44 FFF0000000000000     0        44 fff0000000000000 0
```

Figure 5-2: A window of Active File Compare Program.



```
/testbench/fdout             -2147483644
/testbench/numvector         0
/testbench/clk               1
/testbench/operandA_DPD      7ff0000000000000
/testbench/operandB_DPD      43393ffbe7e08ecc
/testbench/operandC_DPD      7ff0000000000000
/testbench/output_DPD        7ff0000000000000
/testbench/DUT/InvalidOp     St0
/testbench/DUT/overflow      St0
/testbench/DUT/underflow     St0
/testbench/DUT/inexact       St0
```

Figure 5-3: Example ModelSim run for floating-point fused multiply add unit.

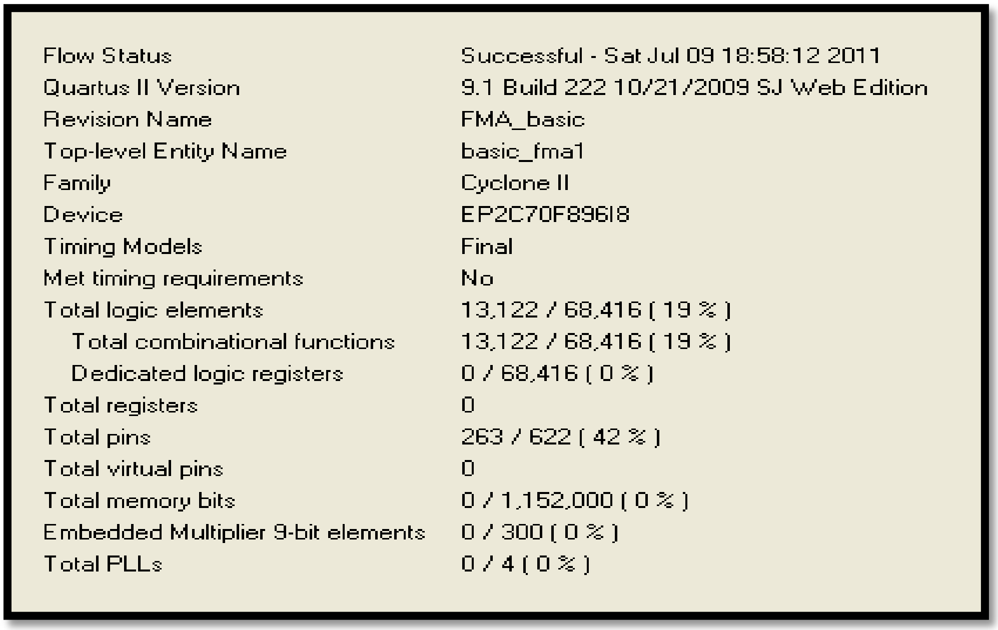
In the FPGA’s technology logic functions are implemented by truth tables defined by programming the SRAM that defines the FPGA’s functionality. Each of the truth tables has a delay which contributes to the delay from inputs to outputs of the function being implemented. In addition, the connections in the FPGA between the inputs, truth tables, and outputs pass through buffers, multiplexers and pass transistors as determined by the circuit specification and the routing paths determined by the implementation tools. The decomposition into truth tables combined with the routing of the interconnections between them yields considerable uncertainty in the propagation delay from input to output of an implemented circuit. The worst case delay which occurs in the circuit from any combinational logic input to any combinational logic output is determined by adding up the maximum expected delays through the combinational circuit including both logic and interconnections. To decrease uncertainty we use constraints to specify the maximum delay allowable, forcing the tools to attempt to meet or better this delay.

The worst case delay of both the basic and the proposed architectures of fused multiply add unit is shown in Figure (5-4). It is clear that the delay improvement is 25.5%.

	Worest-case delay (ns)
Basic architecture	95.345
Proposed architecture	70.985

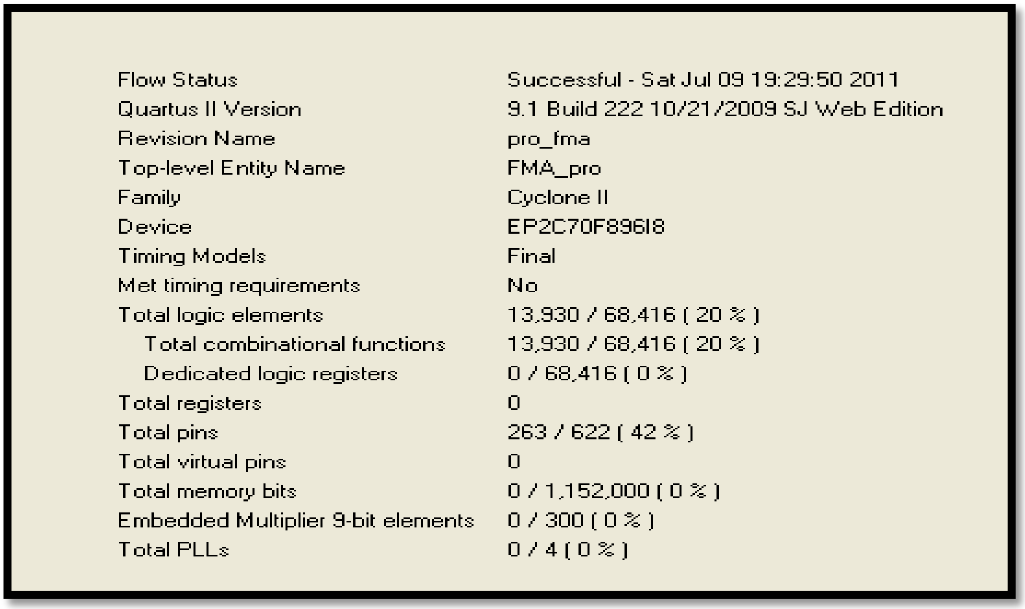
Figure 5-4: Worst case delay of basic and proposed architectures.

The total number of logic elements that are used by FPGA is an important issue in comparing the performance in addition to the worst case delay measurement. A summary of the compilation report is shown in Figure (5-5) (a, b) for both the basic and the proposed architectures.



```
Flow Status                           Successful - Sat Jul 09 18:58:12 2011
Quartus II Version                    9.1 Build 222 10/21/2009 SJ Web Edition
Revision Name                         FMA_basic
Top-level Entity Name                 basic_fma1
Family                                Cyclone II
Device                                EP2C70F896I8
Timing Models                         Final
Met timing requirements               No
Total logic elements                  13,122 / 68,416 ( 19 % )
   Total combinational functions      13,122 / 68,416 ( 19 % )
   Dedicated logic registers          0 / 68,416 ( 0 % )
Total registers                       0
Total pins                            263 / 622 ( 42 % )
Total virtual pins                    0
Total memory bits                     0 / 1,152,000 ( 0 % )
Embedded Multiplier 9-bit elements    0 / 300 ( 0 % )
Total PLLs                            0 / 4 ( 0 % )
```

(a)



```
Flow Status                           Successful - Sat Jul 09 19:29:50 2011
Quartus II Version                    9.1 Build 222 10/21/2009 SJ Web Edition
Revision Name                         pro_fma
Top-level Entity Name                 FMA_pro
Family                                Cyclone II
Device                                EP2C70F896I8
Timing Models                         Final
Met timing requirements               No
Total logic elements                  13,930 / 68,416 ( 20 % )
   Total combinational functions      13,930 / 68,416 ( 20 % )
   Dedicated logic registers          0 / 68,416 ( 0 % )
Total registers                       0
Total pins                            263 / 622 ( 42 % )
Total virtual pins                    0
Total memory bits                     0 / 1,152,000 ( 0 % )
Embedded Multiplier 9-bit elements    0 / 300 ( 0 % )
Total PLLs                            0 / 4 ( 0 % )
```

(b)

Figure 5-5: Compilation report of (a) basic architecture (b) proposed architecture.

From compilation reports it is clear that the basic architecture occupies 13,122 form 68,416 total logic elements which are corresponding to occupying 19% of total logic elements. The proposed architecture occupies 13,930 logic elements; about 20% of total logic elements. The increment in the number of logic gates in the proposed architecture is about 6.2% only.

Both basic and proposed architectures are similar in upper part (multiplier, bit invert and 3:2 CSA) and differ in the lower part, CPA, complementer, normalizer and rounder in basic architecture, three parallel paths (i.e. sign detection ,LZA, normalization shifter and anticipated part of adder) and add round module for proposed architecture. Each block of lower part in two architectures is synthesized separately; Table (5-1) shows the worst case delay and the number of logic gates of each individual block.

Table 5-1: worst case delay and number of logic gates of lower part of(a) basic (b) proposed architectures.

	Worst Case Delay (ns)	**Number of logic gates**
Carry propagate adder	19.434	1998
Complementer	52.308	220
Normalizer	17.468	1693
Rounder	22.971	106

(a)

	Worst Case Delay (ns)	**Number of logic gates**
Three parallel paths	27.124	1700
Add-round module	24.932	712

(b)

5.3 Conclusion

Both basic and proposed architectures are implemented in the Verilog hardware description language and then synthesized for Cyclone II FPGA device using Quartus II 9.1. For the proposed architecture the overall delay is 70.985 ns. It occupies 13,930 from 68,416 total logic elements which are corresponding to occupying 20% of total logic elements. The basic architecture has an overall delay 95.345 ns and occupies 13,122 logic elements, about 19% of total logic elements. The proposed architecture achieves a delay improvement about 25.5% as compared to the basic architecture. The increase of area in the proposed architecture is about 6.2% which is not a big matter. Each block of lower part of two architectures is individually synthesized. The best delay is the delay of normalizer which equals 17.468 ns but the rounder has better area delay product.

Chapter 6： Conclusion

Floating-point unit is an integral part of any modern microprocessor. Floating point units are available in forms of intellectual property for FPGAs to be bought and used by customers. The fused multiply add (FMA) operation was introduced in 1990 on the IBM RS/6000 for the single instruction execution of the equation A+ (B x C) with single and double precision floating-point operands. This hardware unit was designed to reduce the latency of dot product calculations and provided greater floating-point arithmetic accuracy since only a single rounding is performed. FMA is implemented by several commercial processors like IBM, HP, MIPS, ARM and Intel. FMA can be used instead of floating-point addition and floating-point multiplication by using constants e.g., 0.0 + (B x C) for multiplication and A+ (B x 1.0) for addition.

Many approaches are developed on floating-point fused multiply add unit to decrease its latency. The greatest deviation from the original IBM RS/6000 architecture comes from a paper by Lang and Bruguera on a reduced latency fused multiply add. Theirs proposal claims to achieve a significant increase in fused multiply add unit performance by the combination of the addition and rounding stage into one block. The paper claims an estimated 15-20% reduction in latency as compared to a standard fused multiply add. This result is calculated theoretically, and the actual architecture has yet to be implemented in either a synthesized or a custom CMOS silicon design.

The main objective of our work is to implement this algorithm with making some changes in the architecture to facilitate the implementation and on the other hand do not affect the performance. The change is to take the inputs to sign detection and LZA blocks from 3:2 CSA instead of the multiplier outputs and the aligned addend A. This is more efficient because using three inputs (multiplier output and aligned addend A) to sign detection module makes the need of a carry propagate adder is obligatory to add multiplier ouput before making comparison. This change will not increase the overall delay as the delay of CSA already exists in the critical path. The implementation includes full design of blocks are not used before like sign detection module. The basic fused multiply add (IBM RS/6000) architecture is implemented also to make relative comparison. Each block of two architectures is designed using the Verilog hardware description language, and then synthesized, placed and routed for Cyclone II FPGA device using Quartus II 9.1. Area and timing information for each design approach and algorithm is reported and analyzed.

For the proposed Fused multiply add unit the overall delay is 70.985 ns. It occupies 13,930 form 68,416 total logic elements which are corresponding to occupying 20% of total logic elements. The basic fused multiply add architecture has an overall delay 95.345 ns and occupies 13,122 logic elements, about 19% of total logic elements. It is clear that the proposed architecture achieves a delay improvement about 25.5% as compared to the basic architecture. The increase of area in the proposed architecture is about 6.2% which is not a big matter.

6.1 Future Work

In order to expand our research further some of the works proposed are converting the Verilog codes so that it can accommodate any exponent and mantissa length. This will give the design more versatility to use any precision of IEEE binary format also it can be designed for decimal floating point.

BIBLIOGRAPHY

[1] Antelo, E., Bruguera, J., and Zapata, E., “A Novel Design for a Two Operand Normalization Circuit,” IEEE Transactions on VLSI Systems, vol. 6, no. 1, March 1998, Pages: 173 - 176.

[2] Chen, C., Chen, L., and Cheng, J., “Architectural Design of a Fast Floating-Point Multiplication-Add Fused Unit Using Signed-Digit Addition,” Proceedings of the IEE Computer Digital Technology, vol. 149, no. 4, July 2002, Pages: 113 – 120.

[3] Dimitrakopoulos, G., Galanopoulos, K., Mavrokefalidis, C., and Nikolos, D., “Low-Power Leading-Zero Counting and Anticipation Logic for High-Speed Floating Point Units,” IEEE Transactions on VLSI Systems, vol. 16, no. 7, July 2008, Pages: 837 – 850.

[4] Hinds, C., “An Enhanced Floating Point Coprocessor for Embedded Signal Processing and Graphics Applications,” Proceedings of the 33th Asilomar Conference on Signals, Systems, and Computers, vol. 1, October 1999, Pages: 147 – 151.

[5] Hinds, C., and Lutz, D., “A Small and Fast Leading One Predictor Corrector Circuit,” Proceedings of 39th Asilomar Conference on Signals, Systems and Computers, October 2005, pages: 1181– 1185.

[6] Hokenek, E., and Montoye, R., “Leading-Zero Anticipator (LZA) in the IBM RISC System/6000 Floating Point Execution Unit,” IBM Journal of Research and Development, vol. 34, no.1, January1990, pages: 71-77.

[7] James, E., Michael J., “A Combined Two’s Complement and Floating-Point Comparator” Proceedings of the IEEE International Symposium on Circuits and Systems, vol. 1, May 2005, pages: 89 – 92.

[8] Kumar, A., “THE HP PA-8000 RISC CPU,” IEEE computer society, vol. 17, April 1997, pages: 27-32.

[9] Lang, T., and Bruguera, "Leading-One Prediction with Concurrent Position Correction," IEEE Transactions on Computers, vol. 48, no. 10, October 1999, Pages: 1083 – 1097.

[10] Lang, T., and Bruguera, J., "Floating-Point Fused Multiply-Add with Reduced Latency," IEEE Transactions on Computers, vol. 53, no. 8, August 2004, Pages: 42 – 51.

[11] Linzer, E., "Implementation of Efficient FFT Algorithms on Fused Multiply-Add Architectures," IEEE Transactions on Signal Processing, vol. 41, no.1, January 1993, Pages: 93.

[12] Montoye, E., "Floating Point Unit for Calculating A=XY+Z Having Simultaneous Multiply and Add," United States patent, no. 4969118, November 1990.

[13] Montoye, R., Hokenek, E., and Runyon, S., "Second –Generation RISC Floating Point with Multiply-Add Fused," IEEE journal of solid-state circuits, vol. 25, no. 5, October 1990, Pages: 1207 – 1213.

[14] Oberman, S., Al-Twaijry, H., and Flynn, M., "The SNAP Project: Design of Floating-Point Arithmetic Units," Proceeding of the 13th IEEE Symposium on Computer Arithmetic, July 1997, pages: 156-165.

[15] Oklobdzija, V., "An algorithmic and novel design of a leading zero detector circuit: comparsion with logic synthesis," IEEE Transactions on VLSI Systems, vol. 2, no. 1, March 1994, Pages: 124 – 128.

[16] Quach, N., and Flynn, M., "Leading One Prediction – Implementation, Generalization, and Application," Technical Report CSL-TR-91-463, Stanford University, March 1991.

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

[18] Quinnell, et al., "Three-path Fused Multiply Adder Circuit," United States Patent Application Publication, no. 0256150, October 2008.

[19] Quinnel, et al., Swartzlander, E., and Lemonds, C., “Floating-Point Fused Multiply-Add Architectures.” Proceeding of the 41th Asilomar Conference on Signals, Systems and Computers, November 2007, Pages: 42 – 51.

[20] Robison, A., “N-Bit Unsigned Division Via N-Bit Multiply-Add,” Proceedings of the 17th IEEE Symposium on Computer Arithmetic, Cape Cod, Massachusetts, USA, June 2005, Pages: 131 – 139.

[21] Rogenmoser, R., and O’Donnell, L., “Method and apparatus to correct leading one prediction,” United States Patent application, no. 0165887, November 2002.

[22] Santoro, M., Bewick, G., and Horowitz, M.A., “Rounding Algorithms for IEEE Multipliers,” Proceedings of the 9th IEEE Symposium on Computer Arithmetic, Santa Monica, California, USA, September 1989, Pages: 176 – 183.

[23] Schmookler, M., and Nowka, K., “Leading Zero Anticipation and Detection -- A Comparison of Methods,” Proceedings of the 15th IEEE Symposium on Computer Arithmetic, Vail, Colorado, June 2001, Pages: 7 – 12.

[24] Schmookler, M., Trong, S.D., Schwarz, E., and Kroener M., “P6 Binary Floating-Point Unit,” Proceedings of the 18th IEEE Symposium on Computer Arithmetic, Montpellier, France, June 2007, pages: 77-86.

[25] Seidel, P., “Multiple Path IEEE Floating-Point Fused Multiply-Add”, Proceedings of the 46th IEEE International Midwest Symposium on Circuits and Systems, 2003. MWSCAS '03, Pages: 1359 – 1362.

[26] Seidel, P., and Even, G., “A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication,” Proceedings of the 14th IEEE Symposium on Computer Arithmetic, Adelaide, Australia, vol. 49, no. 7, April 1999. Pages: 638 – 650.

[27] Seidel, P., and Even, G., “How Many Logic Levels Does Floating- Point Addition Require?” Proceedings of Int’l Conference Computer Design (ICCD), October 1998, pages: 142 – 149.

[28] Smith, R. A., “A Continued-Fraction Analysis of Trigonometric Argument Reduction,” IEEE Transactions on Computers, vol. 44, no. 11, November 1995, Pages: 1348 – 1351.

[29] Suzuki, H., Morinaka, H., Makino, H., Nakase, Y., Mashiko, K., and Sumi, T., “Leading-zero Anticipatory Logic for High-speed Floating Point Addition,” IEEE Journal of Solid State Circuits, vol. 31, no. 8, August 1996, pages: 1157 – 1164.

[30] Xiao-Lul, M., “Leading Zero Anticipation for Latency Improvement in Floating-Point Fused Multiply-Add Units,” 6th International Conference on ASIC, vol. 2, October 2005, pages: 53 – 56.

[31] Yeh, H. “Fast method of floating point multiplication and accumulation,” United States Patent, no. 5867413, February 1999.

[32] Yu, R., and Zyner, G. “Method and apparatus for partially supporting subnormal operands in floating point multiplication,” United State patent, no. 5602769, February 1997.

[33] Yu, R., and Zyner, G., “167 MHz Radix-4 Floating-Point Multiplier,” Proceedings of the 12th IEEE Symposium on Computer Arithmetic, Bath, England, UK. July 1995, Pages: 149 – 154.

[34] http://www.quadibloc.com/comp/cp0201.htm.

APPENDIX

Sign Processing

```
module sign_processing (Sa,Sb,Sc,o,comp,Sw,eff_o);

input  Sa,Sb,Sc,o,comp;  //o:sign of operation if add o='0',if sub o='1'

output Sw;

output eff_o;  //eff_o is effective operation

wire Sbc;  //sign of bxc

assign Sbc=Sb^Sc;

assign eff_o=o^Sa^Sbc;

assign Sw=(comp&Sa)|(~comp&(Sbc^o));

endmodule

module
```

Multiplier

```
module multiplier (A,B,s,c);

input [52:0] A,B;

output [107:0] s,c;

wire [106:0] pp0,pp1,pp2,pp3,pp4,pp5,pp6,pp7,pp8,pp9,pp10,pp11,pp12,pp13,pp14,pp15,pp16,pp17,pp18,pp19,pp20,pp21,pp22,pp23,pp24,pp25,pp26;

wire  [80:0] Ab;

booth_vector ua3 (A,Ab);

ppg ua1 (Ab,B,pp0,pp1,pp2,pp3,pp4,pp5,pp6,pp7,pp8,pp9,pp10,pp11,pp12,pp13,pp14,pp15,pp16,pp17,pp18,pp19,pp20,pp21,pp22,pp23,pp24,pp25,pp26);

red_tree2 u2 (pp0,pp1,pp2,pp3,pp4,pp5,pp6,pp7,pp8,pp9,pp10,pp11,pp12,pp13,pp14,pp15,pp16,pp17,pp18,pp19,pp20,pp21,pp22,pp23,pp24,pp25,pp26, s,c);

endmodule

module booth (x,b);

input [2:0] x;    //x[0]=z  x[1]=y , x[2]=x

output [2:0] b;    //b[2]=a , b[1]=b  ,b[0]=c

assign b[2]=x[2]&(~x[1]|~x[0]);

assign b[1]=x[2]^(x[1]&x[0]);

assign b[0]=x[1]^x[0];

endmodule

module booth_vector (A,Ab);

input [52:0] A;

output [80:0] Ab;

booth u1 ({A[1:0],1'b0},Ab[2:0]);

booth u2 (A[3:1],Ab[5:3]);

booth u3 (A[5:3],Ab[8:6]);

booth u4 (A[7:5],Ab[11:9]);

booth u5 (A[9:7],Ab[14:12]);

booth u6 (A[11:9],Ab[17:15]);

booth u7 (A[13:11],Ab[20:18]);

booth u8 (A[15:13],Ab[23:21]);

booth u9 (A[17:15],Ab[26:24]);

booth u10 (A[19:17],Ab[29:27]);

booth u11 (A[21:19],Ab[32:30]);

booth u12 (A[23:21],Ab[35:33]);

booth u13 (A[25:23],Ab[38:36]);

booth u14 (A[27:25],Ab[41:39]);

booth u15 (A[29:27],Ab[44:42]);
booth u16 (A[31:29],Ab[47:45]);
booth u17 (A[33:31],Ab[50:48]);
booth u18 (A[35:33],Ab[53:51]);
booth u19 (A[37:35],Ab[56:54]);
booth u20 (A[39:37],Ab[59:57]);
booth u21 (A[41:39],Ab[62:60]);
booth u22 (A[43:41],Ab[65:63]);
booth u23 (A[45:43],Ab[68:66]);
booth u24 (A[47:45],Ab[71:69]);
booth u25 (A[49:47],Ab[74:72]);
booth u26 (A[51:49],Ab[77:75]);
booth u27 ({1'b0,A[52:51]},Ab[80:78]);
endmodule
module ppg (Ab,B,pp0,pp1,pp2,pp3,pp4,pp5,pp6,pp7,pp8,pp9,pp10,pp11,pp12,pp13,pp14,pp15,pp16,pp17,pp18,pp19,pp20,pp21,pp22,pp23,pp24,pp25,pp26);
input [52:0] B;
input [80:0] Ab;
output [106:0] pp0,pp1,pp2,pp3,pp4,pp5,pp6,pp7,pp8,pp9,pp10,pp11,pp12,pp13,pp14,pp15,pp16,pp17,pp18,pp19,pp20,pp21,pp22,pp23,pp24,pp25,pp26;
wire [26:0] y;
wire [105:0] p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,p17,p18,p19,p20,p21,p22,p23,p24,p25,p26;
pp u1 (B,Ab[2:0],y[0],p0);
pp u2 (B,Ab[5:3],y[1],p1);
pp u3 (B,Ab[8:6],y[2],p2);
pp u4 (B,Ab[11:9],y[3],p3);
pp u5 (B,Ab[14:12],y[4],p4);
pp u6 (B,Ab[17:15],y[5],p5);
pp u7 (B,Ab[20:18],y[6],p6);
pp u8 (B,Ab[23:21],y[7],p7);
pp u9 (B,Ab[26:24],y[8],p8);
pp u10 (B,Ab[29:27],y[9],p9);
pp u11 (B,Ab[32:30],y[10],p10);
pp u12 (B,Ab[35:33],y[11],p11);
pp u13 (B,Ab[38:36],y[12],p12);
pp u14 (B,Ab[41:39],y[13],p13);
pp u15 (B,Ab[44:42],y[14],p14);
pp u16 (B,Ab[47:45],y[15],p15);
pp u17 (B,Ab[50:48],y[16],p16);
pp u18 (B,Ab[53:51],y[17],p17);
pp u19 (B,Ab[56:54],y[18],p18);
pp u20 (B,Ab[59:57],y[19],p19);
pp u21 (B,Ab[62:60],y[20],p20);
pp u22 (B,Ab[65:63],y[21],p21);
pp u23 (B,Ab[68:66],y[22],p22);
pp u24 (B,Ab[71:69],y[23],p23);
pp u25 (B,Ab[74:72],y[24],p24);
pp u26 (B,Ab[77:75],y[25],p25);
pp u27 (B,Ab[80:78],y[26],p26);
assign pp0={p0[105],p0};
assign pp1={p1[104:0],1'b0,y[0]};
assign pp2={p2[102:0],1'b0,y[1],2'b0};
assign pp3={p3[100:0],1'b0,y[2],4'b0};
assign pp4={p4[98:0],1'b0,y[3],6'b0};
assign pp5={p5[96:0],1'b0,y[4],8'b0};
assign pp6={p6[94:0],1'b0,y[5],10'b0};
assign pp7={p7[92:0],1'b0,y[6],12'b0};
assign pp8={p8[90:0],1'b0,y[7],14'b0};
assign pp9={p9[88:0],1'b0,y[8],16'b0};
assign pp10={p10[86:0],1'b0,y[9],18'b0};
assign pp11={p11[84:0],1'b0,y[10],20'b0};
assign pp12={p12[82:0],1'b0,y[11],22'b0};
assign pp13={p13[80:0],1'b0,y[12],24'b0};
assign pp14={p14[78:0],1'b0,y[13],26'b0};
assign pp15={p15[76:0],1'b0,y[14],28'b0};
assign pp16={p16[74:0],1'b0,y[15],30'b0};
assign pp17={p17[72:0],1'b0,y[16],32'b0};
assign pp18={p18[70:0],1'b0,y[17],34'b0};
assign pp19={p19[68:0],1'b0,y[18],36'b0};
assign pp20={p20[66:0],1'b0,y[19],38'b0};
assign pp21={p21[64:0],1'b0,y[20],40'b0};
assign pp22={p22[62:0],1'b0,y[21],42'b0};
assign pp23={p23[60:0],1'b0,y[22],44'b0};
assign pp24={p24[58:0],1'b0,y[23],46'b0};
assign pp25={p25[56:0],1'b0,y[24],48'b0};
assign pp26={p26[54:0],1'b0,y[25],50'b0}
endmodule
module  pp (A,bt,s,p);
input [52:0] A;   //A=multiplicand
input [2:0] bt;  //bt=booth encoding
output s;
output [105:0] p;  //o/p PP
wire h;
wire [105:0] B,x,y;
assign h=~bt[1]&~bt[0];
assign B={53'b0,A};
assign x=bt[0]?B:{B[104:0],1'b0};
assign y=bt[2]?~x:x;
assign p=h?106'b0:y;
assign s=h?1'b0:bt[2];
endmodule
module red_tree2
(p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,
p13,p14,p15,p16,p17,p18,p19,p20,p21,p22,p
23,p24,p25,p26, s,c)
input [106:0]
p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,
p13,p14,p15,p16,p17,p18,p19,p20,p21,p22,p
23,p24,p25,p26;
output [107:0] s,c;
wire [106:0]
s1,s2,s3,s4,s5,s6,s7,s8,c1,c2,c3,c4,c5,c6,c7,c8,
pp24,pp25,pp26;
wire [106:0]
ss1,ss2,ss3,ss4,ss5,ss6,cc1,cc2,cc3,cc4,cc5,cc6
,ppp26;
wire [106:0]
sss1,sss2,sss3,sss4,ccc1,ccc2,ccc3,ccc4,px26;
wire [106:0] sv1,sv2,sv4,cv1,cv2,cv4,pv26;
wire [106:0] sx1,sx4,cx1,cx2,cx4,pxx26;
wire [106:0] sy1,cy1,sy4,cy4,py26;
wire [106:0] su1,cu1,su4,pu26;
wire [106:0] sn1,cn1,pn26;

L1_m u1
(p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,
p13,p14,p15,p16,p17,p18,p19,p20,p21,p22,p
23,p24,p25,p26,s1,s2,s3,s4,s5,s6,s7,s8,c1,c2,c
3,c4,c5,c6,c7,c8,pp24,pp25,pp26);

L2_m u2
(s1,s2,s3,s4,s5,s6,s7,s8,c1,c2,c3,c4,c5,c6,c7,c
8,pp24,pp25,pp26,ss1,ss2,ss3,ss4,ss5,ss6,cc1,
cc2,cc3,cc4,cc5,cc6,ppp26);

L3_m u3
(ss1,ss2,ss3,ss4,ss5,ss6,cc1,cc2,cc3,cc4,cc5,cc
6,ppp26,sss1,sss2,sss3,sss4,ccc1,ccc2,ccc3,cc
c4,px26);

L4_m u4
(sss1,sss2,sss3,sss4,ccc1,ccc2,ccc3,ccc4,px26,
sv1,sv2,sv4,cv1,cv2,cv4,pv26);

L5_m u5
(sv1,sv2,sv4,cv1,cv2,cv4,pv26,sx1,sx4,cx1,cx2
,cx4,pxx26);

L6_m u6
(sx1,sx4,cx1,cx2,cx4,pxx26,sy1,cy1,sy4,cy4,py
26);

L7_m u7
(sy1,cy1,sy4,cy4,py26,su1,cu1,su4,pu26);

L8_m u8 (su1,cu1,su4,pu26,sn1,cn1,pn26);

L9_m u9 (sn1,cn1,pn26,s,c);

endmodule

module L1_m
(pp0,pp1,pp2,pp3,pp4,pp5,pp6,pp7,pp8,pp9,
pp10,pp11,pp12,pp13,pp14,pp15,pp16,pp17,
pp18,pp19,pp20,pp21,pp22,pp23,pp24,pp25,
pp26,
s1,s2,s3,s4,s5,s6,s7,s8,c1,c2,c3,c4,c5,c6,c7,c8,
p24,p25,p26);

input [106:0]
pp0,pp1,pp2,pp3,pp4,pp5,pp6,pp7,pp8,pp9,p
p10,pp11,pp12,pp13,pp14,pp15,pp16,pp17,p
p18,pp19,pp20,pp21,pp22,pp23,pp24,pp25,p
p26;

output [106:0]
s1,s2,s3,s4,s5,s6,s7,s8,c1,c2,c3,c4,c5,c6,c7,c8,
p24,p25,p26;

csaa u1 (pp0,pp1,pp2,s1,c1);

csaa u2 (pp3,pp4,pp5,s2,c2);

csaa u3 (pp6,pp7,pp8,s3,c3);

csaa u4 (pp9,pp10,pp11,s4,c4);

csaa u5 (pp12,pp13,pp14,s5,c5);

csaa u6 (pp15,pp16,pp17,s6,c6);

csaa u7 (pp18,pp19,pp20,s7,c7);

csaa u8 (pp21,pp22,pp23,s8,c8);

assign p24=pp24;

assign p25=pp25;

assign p26=pp26;

endmodule

module L2_m
(ss1,ss2,ss3,ss4,ss5,ss6,ss7,ss8,cc1,cc2,cc3,cc
4,cc5,cc6,cc7,cc8,pp24,pp25,pp26,s1,s2,s3,s4
,s5,s6,c1,c2,c3,c4,c5,c6,p26);

input [106:0]
ss1,ss2,ss3,ss4,ss5,ss6,ss7,ss8,cc1,cc2,cc3,cc4
,cc5,cc6,cc7,cc8,pp24,pp25,pp26;

output [106:0]
s1,s2,s3,s4,s5,s6,c1,c2,c3,c4,c5,c6,p26;

csaa u1 (ss1,ss2,{cc1[105:0],1'b0},s1,c1);

csaa u2
({cc2[105:0],1'b0},ss3,{cc3[105:0],1'b0},s2,c2)
;

csaa u3 (ss4,{cc4[105:0],1'b0},ss5,s3,c3);
csaa u4 ({cc5[105:0],1'b0},{cc6[105:0],1'b0},ss6,s4,c4);

csaa u5 (ss7,{cc7[105:0],1'b0},ss8,s5,c5);

csaa u6 (pp24,{cc8[105:0],1'b0},pp25,s6,c6);

assign p26=pp26;

endmodule

module L3_m (ss1,ss2,ss3,ss4,ss5,ss6,cc1,cc2,cc3,cc4,cc5,cc6,pp26, s1,s2,s3,s4,c1,c2,c3,c4,p26);

input [106:0] ss1,ss2,ss3,ss4,ss5,ss6,cc1,cc2,cc3,cc4,cc5,cc6,pp26;

output [106:0]  s1,s2,s3,s4,c1,c2,c3,c4,p26;

csaa u1 (ss1,ss2,{cc1[105:0],1'b0},s1,c1);

csaa u2 ({cc2[105:0],1'b0},{cc3[105:0],1'b0},ss3,s2,c2);

csaa u4 (ss4,ss5,{cc4[105:0],1'b0},s3,c3);

csaa u3 ({cc6[105:0],1'b0},ss6,{cc5[105:0],1'b0},s4,c4);

assign p26=pp26;

endmodule

module L4_m ( ss1,ss2,ss3,ss4,cc1,cc2,cc3,cc4,pp26, s1,s2,s4,c1,c2,c4,p26);

input [106:0] ss1,ss2,ss3,ss4,cc1,cc2,cc3,cc4,pp26;

output [106:0] s1,s2,s4,c1,c2,c4,p26;

 csaa u1 (ss1,ss2,{cc1[105:0],1'b0},s1,c1);

csaa u2 (ss3,{cc2[105:0],1'b0},{cc3[105:0],1'b0},s2,c2);

assign s4=ss4;

assign c4=cc4;

assign p26=pp26;

endmodule

module L5_m (ss1,ss2,ss4,cc1,cc2,cc4,pp26,

            s1,s4,c1,c2,c4,p26);

input [106:0] ss1,ss2,ss4,cc1,cc2,cc4,pp26;

output [106:0] s1,s4,c1,c2,c4,p26;

csaa u1 (ss1,ss2,{cc1[105:0],1'b0},s1,c1);

assign c2=cc2;

assign s4=ss4;

assign c4=cc4;

assign p26=pp26;

endmodule

module L6_m (ss1,ss4,cc1,cc2,cc4,pp26,

            s1,c1,s4,c4,p26);

input [106:0] ss1,ss4,cc1,cc2,cc4,pp26;

output [106:0] s1,c1,s4,c4,p26;

csaa u1 ({cc1[105:0],1'b0},ss1,{cc2[105:0],1'b0},s1,c1);

assign s4=ss4;

assign c4=cc4;

assign p26=pp26;

endmodule

module L7_m ( ss1,cc1,ss4,cc4,pp26,

            s1,c1,c4,p26);

input [106:0] ss1,cc1,ss4,cc4,pp26;
```

```
output [106:0] s1,c1,c4,p26;

csaa u1 ({cc1[105:0],1'b0},ss1,ss4,s1,c1);

assign c4=cc4;

assign p26=pp26;

endmodule

module L8_m ( ss1,cc1,cc4,pp26,

            s1,c1,p26);

input [106:0]  ss1,cc1,cc4,pp26;

output [106:0] s1,c1,p26;

csaa u1 ({cc1[105:0],1'b0},ss1,{cc4[105:0],1'b0},s1,c1);

assign p26=pp26;

endmodule

module L9_m ( ss1,cc1,pp26,s,c);

input [106:0]  ss1,cc1,pp26;

output [107:0] s,c;

wire [106:0] s1,c1;

csaa u1 ({cc1[105:0],1'b0},ss1,pp26,s1,c1);

assign s={s1[106],s1};

assign c={c1,1'b0};

endmodule
```

Bit invert

```
module bit_invert (A,Ainv,sub);

input [0:52] A;

input sub;

output [0:160] Ainv;

assign Ainv=sub?{~A,108'hfffffffffffffffffffffffffff}:{A,108'b0};

endmodule
```

Alignment Shifter & Sticky bit st1 calculation

```
module align_shifter (A,sh_amount,sub,As,st1);

input [0:160] A;

input sub;

input [0:7] sh_amount;

output [0:160] As;

output st1; //sticky bit

wire [0:160] A1,A2,A3,A4,A5,A6,A7;

wire st_1,st_2,st_3,st_4,st_5,st_6,st_7,st_8,t1,t2,t3,t4,t5,t6,t7,t8;

align_sh1 u1 (A,sh_amount[0],sub,A1,st_1); //sh1=shift_amount[0]--> msb

align_sh2 u2 (A1,sh_amount[1],sub,A2,st_2);

align_sh3 u3 (A2,sh_amount[2],sub,A3,st_3);

align_sh4 u4 (A3,sh_amount[3],sub,A4,st_4);

align_sh5 u5 (A4,sh_amount[4],sub,A5,st_5);

align_sh6 u6 (A5,sh_amount[5],sub,A6,st_6);

align_sh7 u7 (A6,sh_amount[6],sub,A7,st_7);

align_sh8 u8 (A7,sh_amount[7],sub,As,st_8); //sh8=shift_amount[7] -->lsb

assign t1=sh_amount[0]&st_1;

assign t2=sh_amount[1]&st_2;

assign t3=sh_amount[2]&st_3;

assign t4=sh_amount[3]&st_4;

assign t5=sh_amount[4]&st_5;

assign t6=sh_amount[5]&st_6;
assign t7=sh_amount[6]&st_7;
assign t8=sh_amount[7]&st_8;
assign st1=t1|t2|t3|t4|t5|t6|t7|t8;
endmodule
module align_sh1 (a,sh1,sub,b,st_1);
input [0:160] a;
input sh1,sub;     //shift by 128 to right
output [0:160] b;
output st_1;
wire [0:160] g;
assign g=sub?{128'hffffffffffffffffffffffffffffffff,a[0:32]}:{128'b0,a[0:32]};
assign b=sh1? g:a;
assign st_1=sub? ~(!(~a[33:160])):~(!a[33:160]);  //if sub=0 check if all bits=0,if sub=1 check if all bits =1
endmodule
module align_sh2 (a,sh2,sub,b,st_2);
input [0:160] a;
input sh2,sub;     //shift by 64 to right
output [0:160] b;
output st_2;
wire [0:160] g;
assign g=sub? {64'hffffffffffffffff,a[0:96]}:{64'b0,a[0:96]};
assign b=sh2? g:a;
assign st_2=sub? ~(!(~a[97:160])):~(!a[97:160]);
endmodule
```

```
module align_sh3 (a,sh3,sub,b,st_3);
input [0:160] a;
input sh3,sub;   //shift by 32 to right
output [0:160] b;
output st_3;
wire [0:160] g;
assign g=sub? {32'hffffffff,a[0:128]}:{32'b0,a[0:128]};
assign b=sh3?g:a;
assign st_3=sub? ~(!(~a[129:160])):~(!a[129:160]);
endmodule
module align_sh4 (a,sh4,sub,b,st_4);
input [0:160] a;
input sh4,sub;   //shift by 16 to right
output [0:160] b;
output st_4;
wire [0:160] g;
assign g=sub? {16'hffff,a[0:144]}:{16'b0,a[0:144]};
assign b=sh4? g:a;
assign st_4=sub? ~(!(~a[145:160])):~(!a[145:160]);
endmodule
module align_sh5 (a,sh5,sub,b,st_5);
input [0:160] a;
input sh5,sub;   //shift by 8 to right
output [0:160] b;
output st_5;
```

```
wire [0:160] g;

assign g=sub? {8'hff,a[0:152]}:{8'b0,a[0:152]};

assign b=sh5? g:a;

assign st_5=sub? ~(!(~a[153:160])):~(!a[153:160]);

endmodule

module align_sh6 (a,sh6,sub,b,st_6);

input [0:160] a;

input sh6,sub;   //shift by 4 to right

output [0:160] b;

output st_6;

wire [0:160] g;

assign g=sub? {4'hf,a[0:156]}:{4'b0,a[0:156]};

assign b=sh6? g:a;

assign st_6=sub? ~(!(~a[157:160])):~(!a[157:160]);

endmodule

module align_sh7 (a,sh7,sub,b,st_7);

input [0:160] a;

input sh7,sub;   //shift by 2 to right

output [0:160] b;

output st_7;

wire [0:160] g;

assign g=sub? {2'b11,a[0:158]}:{2'b0,a[0:158]};

assign b=sh7?g:a;

assign st_7=sub?~(!(~a[159:160])):~(!a[159:160]);

endmodule

module align_sh8 (a,sh8,sub,b,st_8);

input [0:160] a;

input sh8,sub;   //shift by 1 to right

output [0:160] b;

output st_8;

wire [0:160] g;

assign g=sub?{1'b1,a[0:159]}:{1'b0,a[0:159]};

assign b=sh8? g:a;

assign st_8=sub? ~a[160]:a[160];

endmodule
```

LZA

```
module lza (a,b,f);

input [0:107] a,b;

output [0:107] f;

wire [0:107] t,g,z,g1,z1,t1,x;

assign t=a^b;

assign g=a&b;

assign z=~a&~b;

assign g1={g[1:107],1'b0};

assign z1={z[1:107],1'b0};

assign t1={1'b0,t[0:106]};

assign x=(t1&((g&~z1)|(z&~g1)))|(~t1&((z&~z1)|(g&~g1)));

assign f[0]=~t[0]&t[1];

assign f[1:107]=x[1:107];

endmodule
```

Multiplixer

```
module mux0 (a,b,s,c);
input [161:0] a,b;
input s;
output [161:0] c;
assign c=s? b:a;
```

Shift amount

```
module encode_norm (f,S1,S2,S3,S4,S5,S6,S7);
input [0:107] f;
output S1,S2,S3,S4,S5,S6,S7;
S1 U1(f,S1);
S2 u2 (f,S1,S2);
S3 u3 (f,S1,S2,S3);
S4 u4 (f,S1,S2,S3,S4);
S5 u5 (f,S1,S2,S3,S4,S5);
S6 u6 (f,S1,S2,S3,S4,S5,S6);
S7 u7 (f,S1,S2,S3,S4,S5,S6,S7);
Endmodule
module S1 (A,S1);
input [0:107] A;
output S1;
assign S1=!A[0:63];
endmodule
module S2 (A,S1,S2);
input [0:107] A;
input S1;
output S2;
wire x1,x2;
assign x1=!A[0:31];
assign x2=!A[64:95];
assign S2=S1?X2:X1;
endmodule
module S3 (A,S1,S2,S3);
input [0:107] A;
input S1,S2;
output S3;
wire x1,x2,x3,x4;
wire f1,f2;
assign x1=!A[0:15];
assign x2=!A[64:79];
assign x3=!A[32:47];
assign x4=!A[96:107];
mux0 u1 (x1,x2,S1,f1);
mux0 u2 (x3,x4,S1,f2);
mux0 u3 (f1,f2,S2,S3);
endmodule
module S4 (A,S1,S2,S3,S4);
input [0:107] A;
input S1,S2,S3;
output S4;
wire x1,x2,x3,x4,x5,x6,x7;
wire f1,f2,f3,f4,f5;
assign x1=!A[0:7];
assign x2=!A[64:71];
assign x3=!A[32:39];
assign x4=!A[96:103];
assign x5=!A[16:23];

assign x6=!A[80:87];

assign x7=!A[48:55];

mux0 u1 (x1,x2,S1,f1);

mux0 u2 (x3,x4,S1,f2);

mux0 u3 (x5,x6,S1,f3);

mux0 u4 (f1,f2,S2,f4);

mux0 u5 (f3,x7,S2,f5);

mux0 u6 (f4,f5,S3,S4);

endmodule

module S5 (A,S1,S2,S3,S4,S5);

input [0:107] A;

input S1,S2,S3,S4;

output S5;

wire x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14;

wire f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12;

assign x1=!A[0:3];

assign x2=!A[64:67];

assign x3=!A[32:35];

assign x4=!A[96:99];

assign x5=!A[16:19];

assign x6=!A[80:83];

assign x7=!A[48:51];

assign x8=!A[8:11];

assign x9=!A[72:75];

assign x10=!A[40:43];

assign x11=!A[104:107];

assign x12=!A[24:27];

assign x13=!A[88:91];

assign x14=!A[56:59];

mux0 u1 (x1,x2,S1,f1);

mux0 u2 (x3,x4,S1,f2);

mux0 u3 (x5,x6,S1,f3);

mux0 u4 (x8,x9,S1,f4);

mux0 u5 (x10,x11,S1,f5);

mux0 u6 (x12,x13,S1,f6);

mux0 u7 (f1,f2,S2,f7);

mux0 u8 (f3,x7,S2,f8);

mux0 u9 (f4,f5,S2,f9);

mux0 u10 (f6,x14,S2,f10);

mux0 u11(f7,f8,S3,f11);

mux0 u12 (f9,f10,S3,f12);

mux0 u13 (f11,f12,S4,S5);

endmodule

module S6 (A,S1,S2,S3,S4,S5,S6);

input [0:107] A;

input S1,S2,S3,S4,S5;

output S6;

wire x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17,x18,x19,x20,x21,x22,x23,x24,x25,x26,x27;

wire f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11;

wire k1,k2,k3,k4,k5,k6,k7,k8;

wire m1,m2,m3,m4;
wire n1,n2;

assign x1=!A[0:1];

assign x2=!A[64:65];

assign x3=!A[32:33];

assign x4=!A[96:97];

assign x5=!A[16:17];

assign x6=!A[80:81];

assign x7=!A[48:49];

assign x8=!A[8:9];

assign x9=!A[72:73];

assign x10=!A[40:41];

assign x11=!A[104:105];

assign x12=!A[24:25];

assign x13=!A[88:89];

assign x14=!A[56:57];

assign x15=!A[4:5];

assign x16=!A[68:69];

assign x17=!A[36:37];

assign x18=!A[100:101];

assign x19=!A[20:21];

assign x20=!A[84:85];

assign x21=!A[52:53];

assign x22=!A[12:13];

assign x23=!A[76:77];

assign x24=!A[44:45];

assign x25=!A[28:29];

assign x26=!A[92:93];

assign x27=!A[60:61];
```

```
mux0 u1 (x1,x2,S1,f1);

mux0 u2 (x3,x4,S1,f2);

mux0 u3 (x5,x6,S1,f3);

mux0 u4 (x8,x9,S1,f4);

mux0 u5 (x10,x11,S1,f5);

mux0 u6 (x12,x13,S1,f6);

mux0 u7 (x15,x16,S1,f7);

mux0 u8 (x17,x18,S1,f8);

mux0 u9 (x19,x20,S1,f9);

mux0 u10 (x22,x23,S1,f10);

mux0 u11 (x25,x26,S1,f11);

mux0 u12 (f1,f2,S2,k1);

mux0 u13 (f3,x7,S2,k2);

mux0 u14 (f4,f5,S2,k3);

mux0 u15 (f6,x14,S2,k4);

mux0 u16(f7,f8,S2,k5);

mux0 u17 (f9,x21,S2,k6);

mux0 u18 (f10,x24,S2,k7);

mux0 u19 (f11,x27,S2,k8);

mux0 u20 (k1,k2,S3,m1);

mux0 u21 (k3,k4,S3,m2);

mux0 u22 (k5,k6,S3,m3);

mux0 u23 (k7,k8,S3,m4);

mux0 u24 (m1,m2,S4,n1);

mux0 u25 (m3,m4,S4,n2);

mux0 u26 (n1,n2,S5,S6);

endmodule

module S7 (f,S1,S2,S3,S4,S5,S6,S7);
```

```
input [0:107] f;
input S1,S2,S3,S4,S5,S6;
output S7;
wire [0:107] A;
wire f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14,f15,f16,f17,f18,f19,f20,f21,f22;
wire k1,k2,k3,k4,k5,k6,k7,k8,k9,k10,k11,k12,k13,k14,k15,k16;
wire m1,m2,m3,m4,m5,m6,m7,m8;
wire n1,n2,n3,n4;
wire g1,g2;
assign A=~f;
mux0 u1 (A[0],A[64],S1,f1);
mux0 u2 (A[32],A[96],S1,f2);
mux0 u3 (A[16],A[80],S1,f3);
mux0 u4 (A[8],A[72],S1,f4);
mux0 u5 (A[40],A[104],S1,f5);
mux0 u6 (A[24],A[88],S1,f6);
mux0 u7 (A[4],A[68],S1,f7);
mux0 u8 (A[36],A[100],S1,f8);
mux0 u9 (A[20],A[84],S1,f9);
mux0 u10 (A[12],A[76],S1,f10);
mux0 u11 (A[28],A[92],S1,f11);
mux0 u12 (A[2],A[66],S1,f12);
mux0 u13 (A[34],A[98],S1,f13);
mux0 u14 (A[18],A[82],S1,f14);
mux0 u15 (A[10],A[74],S1,f15);
mux0 u16 (A[42],A[106],S1,f16);
mux0 u17 (A[26],A[90],S1,f17);
mux0 u18 (A[6],A[70],S1,f18);
mux0 u19 (A[38],A[102],S1,f19);
mux0 u20 (A[22],A[86],S1,f20);
mux0 u21 (A[14],A[78],S1,f21);
mux0 u23 (A[30],A[94],S1,f22);
mux0 u24 (f1,f2,S2,k1);
mux0 u25 (f3,A[48],S2,k2);
mux0 u26 (f4,f5,S2,k3);
mux0 u27 (f6,A[56],S2,k4);
mux0 u28(f7,f8,S2,k5);
mux0 u29 (f9,A[52],S2,k6);
mux0 u30 (f10,A[44],S2,k7);
mux0 u31 (f11,A[60],S2,k8);
mux0 u32 (f12,f13,S2,k9);
mux0 u33 (f14,A[50],S2,k10);
mux0 u34 (f15,f16,S2,k11);
mux0 u35 (f17,A[58],S2,k12);
mux0 u36 (f18,f19,S2,k13);
mux0 u37 (f20,A[54],S2,k14);
mux0 u38 (f21,A[46],S2,k15);
mux0 u39 (f22,A[62],S2,k16);
mux0 u40 (k1,k2,S3,m1);
mux0 u41 (k3,k4,S3,m2);
mux0 u42 (k5,k6,S3,m3);
mux0 u43 (k7,k8,S3,m4);
mux0 u44 (k9,k10,S3,m5);
```

```
mux0 u45 (k11,k12,S3,m6);
mux0 u46 (k13,k14,S3,m7);
mux0 u47 (k15,k16,S3,m8);
mux0 u48 (m1,m2,S4,n1);
mux0 u49 (m3,m4,S4,n2);
mux0 u50 (m5,m6,S4,n3);
mux0 u51 (m7,m8,S4,n4);
mux0 u52 (n1,n2,S5,g1);
mux0 u53 (n3,n4,S5,g2);
mux0 u54 (g1,g2,S6,S7);
endmodule
```

Round decision

```
module Round_decision (RI,RN,S,Rd,Rd1);
input RI,RN,S;
output Rd,Rd1;
assign Rd=RI|RN;
assign Rd1=RI&S;
endmodule
```

LSB correction

```
module LSB_correct (RN,N,Sn,Sun,Rn,Run,LSB,Lc);
input RN,Sn,Sun,Rn,Run,LSB,N;
output Lc;
wire x1,x2,x3,x4;
assign x1=Rn&~Sn;  //tie case for normalized Rn=Ro
assign x2=Run&~Sun;  //tie case for unnor. Run=Rn
assign x3=N?x1:x2;
assign x4=x3&RN;
assign Lc=LSB&~x4;
endmodule
```