DESIGN OF EFFICIENT 32-BIT FLOATING POINTMULTIPLIER USING STANDARD IEEE 754

Prof. Sachin S Patil[1], Prof. Sujata S Kamate[2]

Assistant Professor[1&2]

Department of Electronics & Communication Engineering[1&2]

Hirasugar Institute of Technology, Nidasoshi-591313, India[1&2]

Abstract— The IEEE 754 standard provides the format for representation of Binary Floating point numbers. The Binary Floating point numbers are represented in Single and Double formats. The Single consist of 32 bits and the Double consist of 64 bits. The formats are composed of 3 fields: Sign, Exponent and Mantissa. In case of Single, the Mantissa is represented in 23 bits and 1 bit is added to the MSB for normalization, Exponent is represented in 8 bits which is biased to 127, actually the Exponent is represented in excess 127 bit format and MSB of Single is reserved for Sign bit. When the sign bit is 1 that means the number is negative and when the sign bit is 0 that means the number is positive. In 64 bits format the Mantissa is represented in 52 bits, the Exponent is represented in 11 bits which is biased to 1023 and the MSB of Double is reserved for sign bit. The main object of this paper is to reduce the power consumption and to increase the speed of execution by implementing certain algorithm for multiplying two floating point numbers.

These Lab-Oriented work and Activities have been carried out into two parts. First Half is the Floating Point Representation Using IEEE-754 Format (32 Bit Single Precision) and second Half is simulation, synthesis of Design using HDLs and Software Tools. The Binary representation of decimal floating-point numbers permits an efficient implementation of the proposed radix independent IEEE standard for floating-point arithmetic.

Keywords— **IEEE-754 Format, Simulation, Synthesis.**

1. INTRODUCTION

Floating point numbers are one possible way of representing real numbers in binary format; the IEEE 754 standard presents two different floating point formats, Binary interchange format and Decimal interchange format. Multiplying floating point numbers is a critical requirement for DSP applications involving large dynamic range. This paper focuses only on single precision normalized binary interchange format. Fig. 1.1 shows the IEEE 754 single precision binary format representation; it consists of a one bit sign (S), an eight bit exponent (E), and a twenty three bit fraction (M or Mantissa). An extra bit is added to the fraction to form what is called the significand 1. If the exponent is greater than 0 and smaller than 255, and there is 1 in the MSB of the significand then the number is said to be a normalized number; in this case the real number is represented by (1).

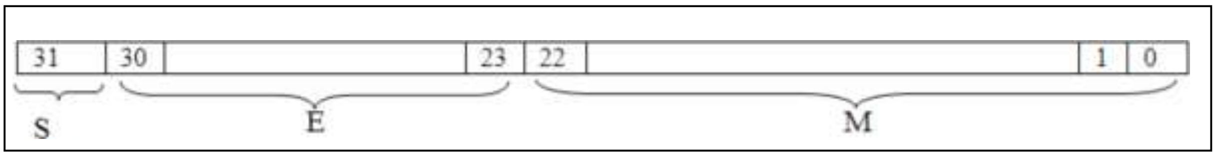


Figure 1.1 IEEE single precision floating point format

Z = (-1S) * 2 (E - *Bias*) * (1.M) □ □□□

Where M = m22 2-1 + m21 2-2 + m20 2-3+…+ m1 2-22+ m0 2-23, Bias = 127.

Significand is the mantissa with an extra MSB bit.

Multiplying two numbers in floating point format is done by

1- adding the exponent of the two numbers then subtracting the bias from theirresult 2-

multiplying the significand of the two numbers,and

3- calculating the sign by XORing the sign of the two numbers.

In order to represent the multiplication result as a normalized number there should be 1 in the MSB of the result (leading one).

Floating-point implementation on FPGAs has been the interest of many researchers. In 2008, an IEEE 754 single precision pipelined floating point multiplier was implemented onmultiple FPGAs (4 Actel A1280). In 1995, a custom 16/18 bit three stage pipelined floating point multiplier that doesn't support rounding modes was implemented. In 1996, a single precision floating point multiplier that doesn't support rounding modes was implemented using a digit-serial multiplier: using the Altera FLEX 8000 it achieved 2.3 MFlops. In 2001, a parameterizable floating point multiplier was implemented using the software-like language Handel-C, using the Xilinx XCV1000 FPGA; a five stages pipelined multiplier achieved 28MFlops. In 2002, a latency optimized floating point unit using the primitives of Xilinx Virtex II FPGA was implemented with a latency of 4 clock cycles. The multiplier reached a maximum clock frequency of 100 MHz.

Our discussion of floating point will focus almost exclusively on the IEEE floating- point standard (IEEE 754) because of its rapidly increasing acceptance. Although floating- point arithmetic involves manipulating exponents and shifting fractions, the bulk of the time in floating-point operations is spent operating on fractions using integer algorithms.

2. ARCHITECTURAL DETAILS

2.1IMPLEMENTATION OF PROPOSEDARCHITECTURE

The Architecture has sign calculator, exponent calculator, mantissa calculator, which works parallel, and a normalization unit. It takes two IEEE 754 format single precision floating point numbers and produces the multiplied output. It also supports the features like underflow, overflow and invalid operations. The implementation of Floating point multiplier Unit consists of two stages of multiplication calculation and Normalization. First stage includes three blocks which work inparallel.

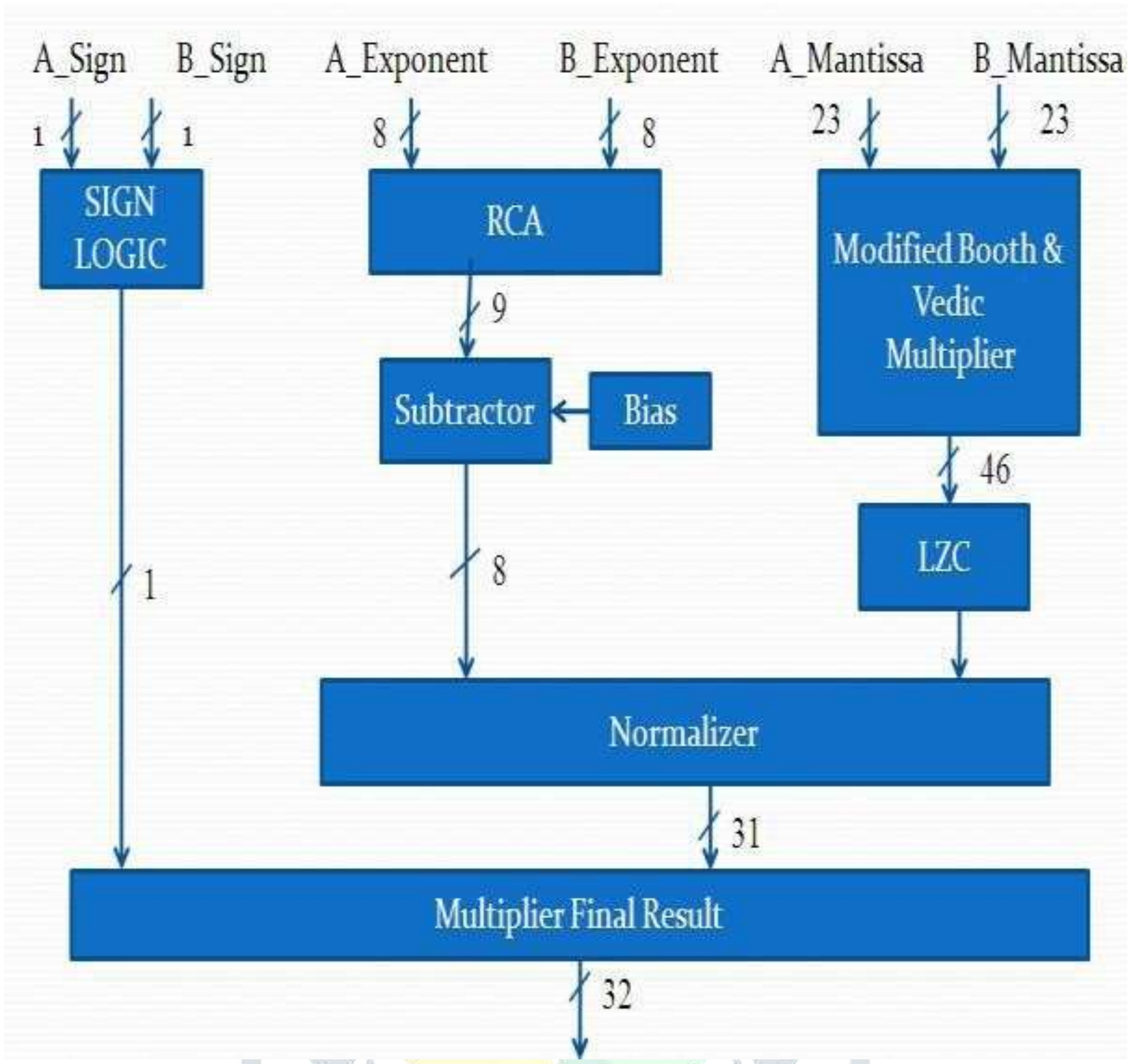


Figure 2.1: Floating Point Multiplier Architecture Using Booth and Vedic techniques

A. Sign Calculator: The Output Sign is the exclusive or of two sign bitinputs

B. Exponent Calculator: The input exponents are added using Ripple Carry Adder (RCA) and the bias is subtracted using Ripple Carry Subtractor (RCS) to produce the exponent of Output.

C. Mantissa Calculator: Output Mantissa is calculated by multiplying themantissa's using multiplier (Modified Booth/Vedic techniques)

Second stage performs Normalization of the first stage output. It first calculates how much amount the mantissa needs to be left shifted using LZC (Leading Zero Counter) and finally produces the multiplier output.

2.2 MULTIPLIERS USED IN THEARCHITECTURE

2.2.1 Vedicmultiplier:

For Mantissa calculations, Vedic and modified booth multipliers are used in the implementation. The design of Vedic Multiplier starts with 2x2 bit multiplier. Here, "Urdhva Tiryakbhyam Sutra" (Vertically and Crosswise Algorithm) has been used for multiplication to develop multiplierarchitecture.

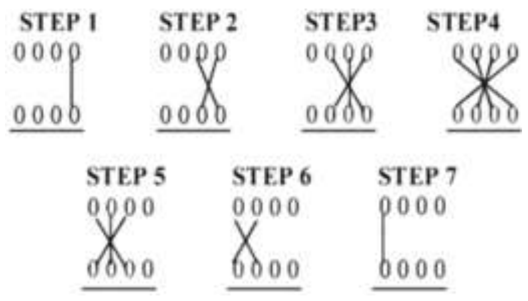


Figure 2.2: 4 X 4 bit Vedic multiplication

The expressions of the partial products obtained by multiplying A=A3A2A1A0 and B= B3B2B1B0 are P1=A0B0 and Carry=C0 P2=A1B0+B1A0+C0 and Carry=C1 P3=A2B0+B2A0+A1B1+C1 and Carry=C2 P4=A3B0+A0B3+A2B1+A1B2+C2 &Carry=C3

P5=A3B1+A2B2+A1B3+C3 and Carry=C4 P6=A3B2+A2B3+C4 and Carry=C5
P7=A3B3+C5 This Sutra shows how to handle multiplication of a larger number (N x N, of N bits each) by breaking it into smaller numbers of size (N/2 = n, say) and these smaller numbers can again be broken into smaller numbers (n/2 each) till 2 × 2 basic multiplier block. Hence, whole multiplication process is to be simplified. First the basic block, 2×2 multipliers have been made then, using these blocks, 4×4 block and thereby using 4×4 block, 8×8 block and then finally 16×16 bit Multiplier has beenmade.

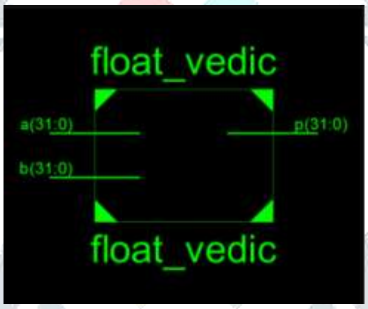


Figure 2.3: Black box view of single precision floating point vedic multiplier

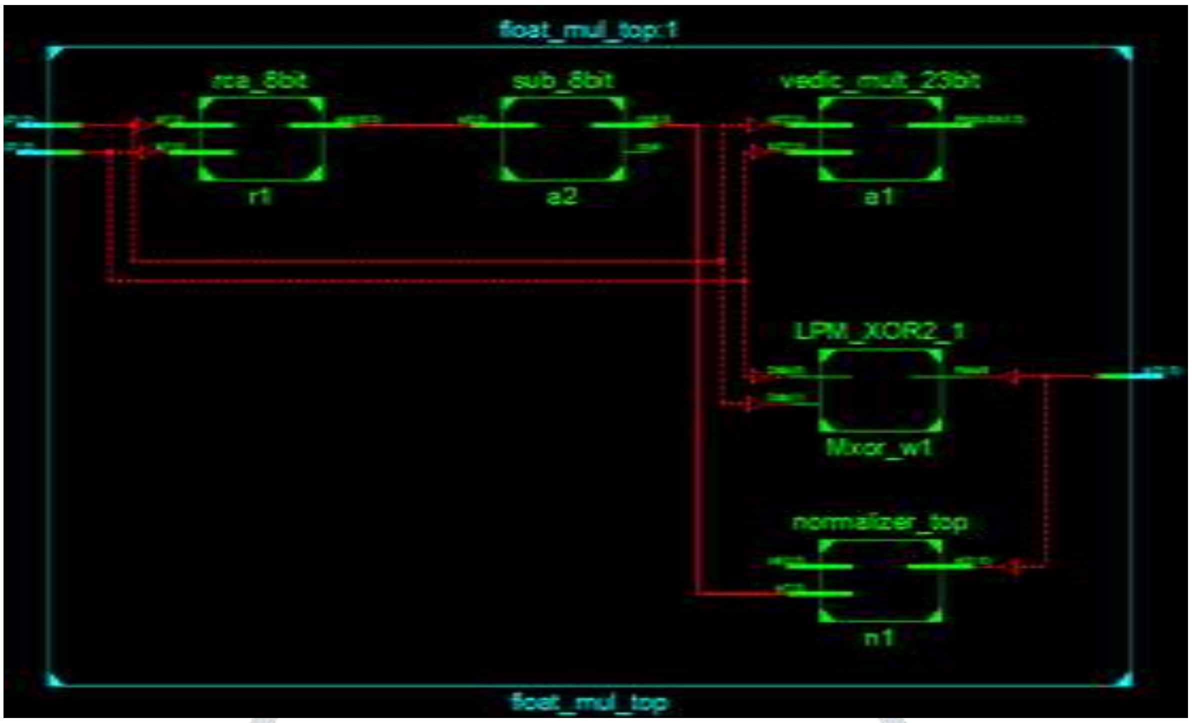


Figure 2.4: single precision floating point vedic multiplier

2.2.2 Modified BoothMultiplier

Booth multiplication is smaller, faster multiplication algorithm through encoding the signed numbers to 2’s complement, which is also a standard technique used in chip design, and provides significant improvements by reducing the number of partial product to half over “long multiplication” techniques. Modified Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. Modified Booth algorithm reduces the number of partial products generated in a multiplication process through encoding the signed numbers to 2’s complement according to the table shown below.

TABLE 2.1 Modified Booth Encoding Table

b2b1b0	Operations
000	All zero
001	Same number
010	Same number
011	Single left shift
100	2's complement and left shift
101	2's complement
110	2's complement
111	All zero's

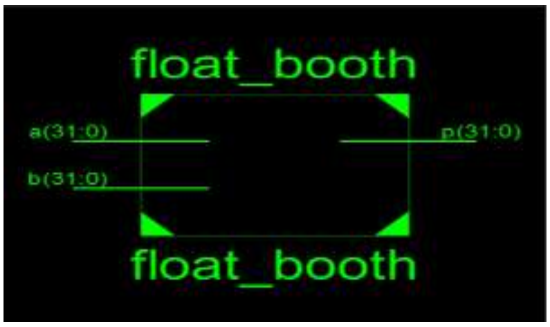


Figure 2.5: Black box view of single precision floating point modified booth multiplier

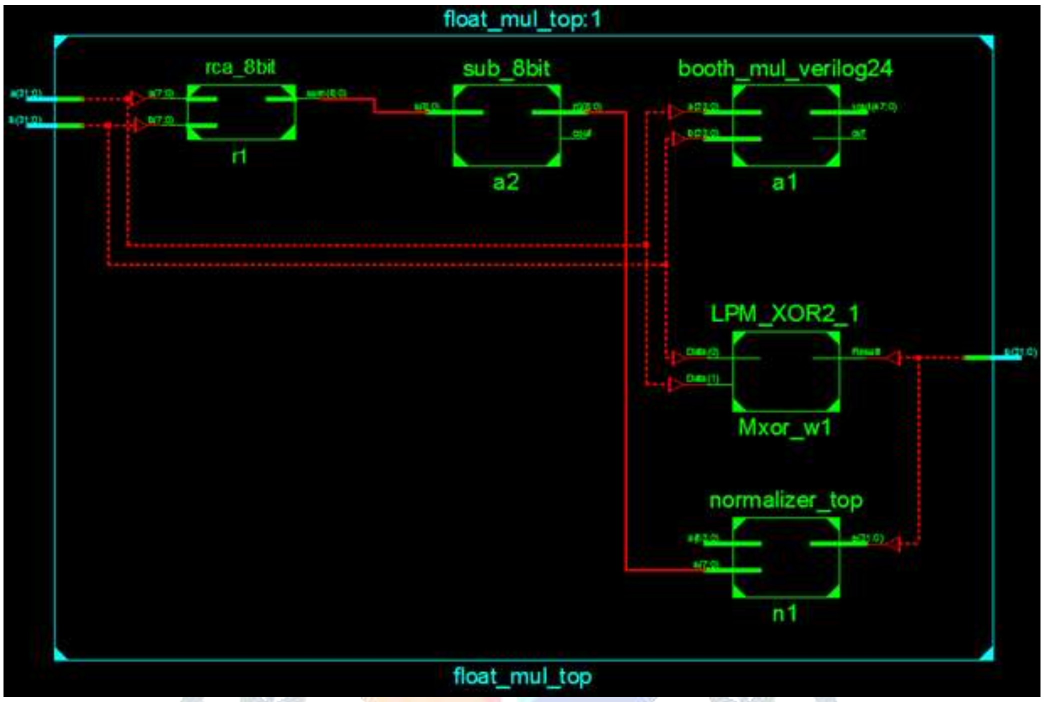


Figure 2.6: single precision floating point modified booth multiplier

2.2.3 DELAY RESULTS:

Multiplier	Delay(ns)
Modified Booth algorithm	121.737
Vedic multiplier	94.770

3. COMPARISON WITH DIFFERENT MULTIPLICATION METHODS

This is the most important stage, product of the mantissa bits is calculated. The multiplication of mantissa bits is performed in the following stages.

3.1 Generation of PartialProducts

The Booth multiplier makes use of Booth encoding algorithm in order to reduce the number of partial products by considering certain number of bits of the multiplier at a time, thereby achieving a speed advantage over other multiplier architectures. This algorithm is valid for both signed and unsigned numbers. It can handle signed binary multiplication by using 2's complement representation. For generating the partial products Radix-8 Modified Booth's Algorithm is used. Since the multiplier and multiplicand comprises of 24 bits, this algorithm will generate 8 partial products. The shortcoming of Radix 2 Booth algorithm is that it becomes inefficient when there are isolated 1's. For example, 001010101(decimal 85) gets reduced to 01-11- 11-11-1(decimal 85), requiring eight instead of four operations.001010101(0) recoded as 011111111, requiring 8 instead of 4operations.

3.1.1 Radix-8 Modified Booth'sAlgorithm

Recoding extended to 3 bits at a time - overlapping groups of 4 bits each. Radix-8 recoding applies the same algorithm as radix-4, but now we take quartets of bits instead of triplets.
Consequently, a multiplier based on this radix-8 scheme generates fewer partial products than a radix-4 multiplier, but the computation of each partial product is more complex. In particular, a partial product corresponding to an encoding x=+3 requires the computation of 3x, and therefore a full addition. Each quartet is codified as a signed-digit using the table 3.1.

Table 3.1: Recoding in Booth Radix-8 Algorithm.

Quartet value	Signed-digit value	Quartet value	Signed-digit value
0000	0	1000	-4
0001	+1	1001	-3
0010	+1	1010	-3
0011	+2	1011	-2
0100	+2	1100	-2
0101	+3	1101	-1
0110	+3	1110	-1
0111	+4	1111	0

3.1.2 Synthesis Results onFPGA

Table 3.2 shows the synthesis report on Xilinx for generation of partial products.

Table 3.2: Synthesis Report of Partial Product Generation

	Spartan 3E xc3s500E	Virtex 4 xc4vlx15
No. of Slices	2003/4656 (43%)	1913/6144 (31%)
No. of LUTs	3653/9312 (39%)	3794/12288 (30%)
Minimum Period	10.344ns	6.537ns
Maximum Frequency	96.676MHz	152.986MHz

3.2 Partial Product Reduction

8 Partial products are generated using Radix-8 Modified Booth's Algorithm. They are reduced using 4:2compressors.

3.2.1 Carry Save Adder

A Carry-Save Adder is just a set of one-bit full adders, without any carry-chaining. The most important application of a carry-save adder is to calculate the partial products in integer multiplication. 4:2 compressors are used as carry save adders. The 4:2 compressor structure actually compresses five partial products bits into three. The architecture is connected in such a way that four of the inputs are coming from the same bit position of the weight j while one bitis fed from the neighboring position j- 1(known as carry-in). The outputs of 4:2 compressor consists of one bit in the position j and two bits in the position j+1.

A 4:2 compressor can also be built using 3:2 compressors. It consists of two 3:2 compressors (full adders) in series and involves a critical path of 4 XOR delays as shown in Figure 3[8]. The output Cout, being independent of the input Cin accelerates the carry save summation of the partial products. 4:2 compressor is made from 2 full adders. The final carry is saved and hence is called carry save adder. The delay of 4:2 compressor is equal that of 4 xor gates.

Initially two 4:2 compressors are used to reduce each 4 partial products pair to generate the pair of sum and carry. Then these final 4 partial products generated from above two 4:2 compressors are further reduced to generate final sum and carry. The final sum and carry are added in next Carry Propagate adder.

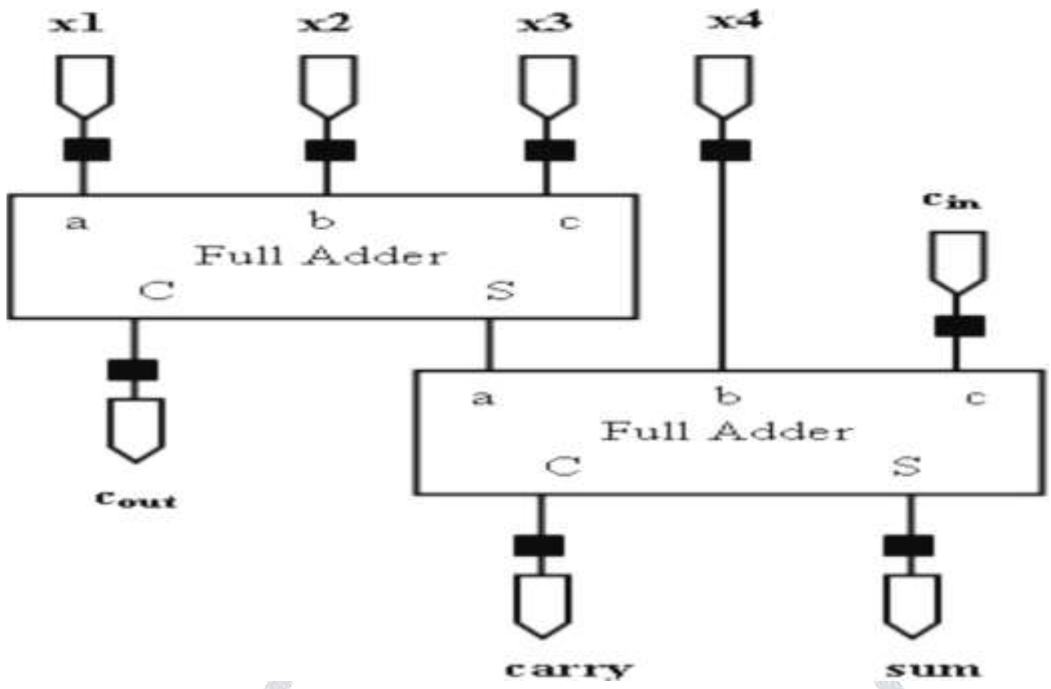


Figure 3.1: 4:2 Compressor Design using Full Adders

3.2.2 Synthesis Results onFPGA

Table 3.3 shows the synthesis report on Xilinx for Partial Product Addition.

Table 3.3: Synthesis Report of Partial Product Addition

	Spartan 3E xc3s500E	Virtex 4 xc4vlx15
No. of Slices	501/4656 (10%)	502/6144 (8%)
No. of LUTs	706/9312 (7%)	702/12288 (5%)
Minimum Period	5.184ns	3.075ns
Maximum Frequency	192.905MHz	325.256MHz

3.3 Final stage Carry PropagateAdder

Further the partial products generated through carry save adders are further reduced by using Ripple Carry Adder.

3.3.1 Ripple Carry Adder

Ripple Carry Adder is used to obtain the final sum and the output carry by adding the partial products from the carry save adders. It creates a logical circuit using multiple full adders to add N-bit numbers. Each full adder inputs a Cin, which is the Cout of the previous adder. This kind of adder is a ripple carry adder, since each carry bit "ripples" to the next full adder. The 48-bit sum and carry outputs obtained from the partial product accumulator are added in the final stage adder to give the product of the mantissas. As shown in Figure 4 a ripple carry adder is a chain of cascaded full adders and one half adder; each full adder has three inputs (A, B, Ci) and two outputs (S, Co). The carry out (Co) of each adder is fed to the next full adder (i.e each carry bit "ripples" to the next fulladder).

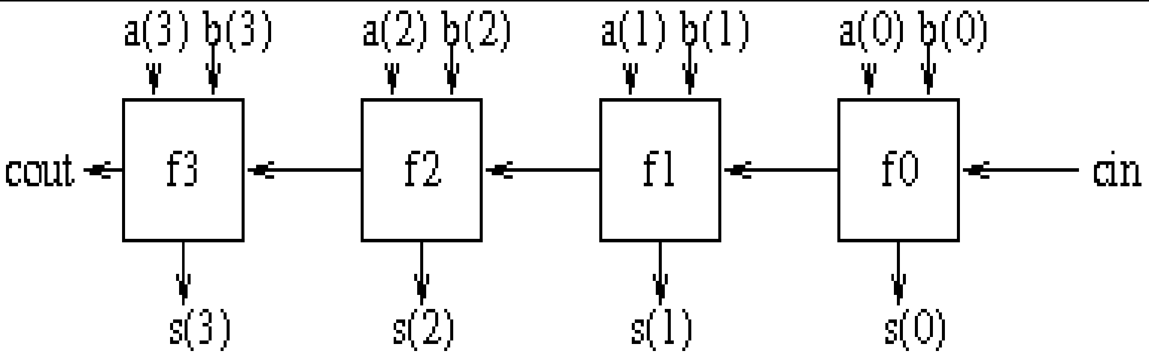


Figure 3.2: Ripple Carry Adder

3.3.2 Synthesis Results on FPGA

Table 3.4 shows the synthesis report on Xilinx for Final 48- bit Stage Carry Propagate Adder.

Table 3.4: Synthesis Report of 48-bit Ripple Carry Adder

Device	Spartan 3E xc3s500E	Virtex 4 xc4vlx15
No. of Slices	150/4656 (3%)	146/6144 (2%)
No. of LUTs	246/9312 (2%)	250/12288 (2%)
Minimum Period	8.235ns	5.206ns
Maximum Frequency	121.438MHz	192.095MHz

3.4 PIPELINING

A pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. It is divided into segments and each segment can execute its operation concurrently with the other segments. When a segment completes an operation, it passes the result to the next segment in the pipeline and fetches the next operation fromthepreceding segment. The final results of each instruction emerge at the end of the pipeline in rapid succession. The pipeline technique is widely used to improve the performance of digital circuits. As the number of pipeline stages is increased, the path delays of each stage are decreased and the overall performance of the circuit is improved.

3.4.1 5-Stage Pipelining

In order to enhance the performance of the multiplier, five pipelining stages are used to divide the critical path thus increasing the maximum operating frequency of the multiplier. Five pipelining stages mean that there is latency in the output by five clocks[8]. The pipelining stages are embedded at the following locations:

i. After the Pre-processing of the Multiplicand and Multiplier.

ii. After the Exponent Adder and Generation of 8 Partial Products.

iii. After subtracting the Bias and compressing the partial Products to4.

iv. After Compressing the Partial Products to2.

v. After Normalization and Final Carry Propagate Adder.

Table 3.5 shows the synthesis result. Comparing with 3-stage pipelined multiplier, the frequency is increased as the pipeline stages are increased.

3.4.2 Synthesis Results on FPGA

Table 3.5: Synthesis Report of 5-Stage Floating Point Multiplier

Device	Spartan 3E xc3s500E	Virtex 4 xc4vlx15
No. of Slices	2045/4656 (43%)	2067/6144 (33%)
No. of LUTs	3893/9312 (41%)	3900/12288 (31%)
Minimum Period	12.154ns	6.532ns
Maximum Frequency	82.279MHz	153.081MHz

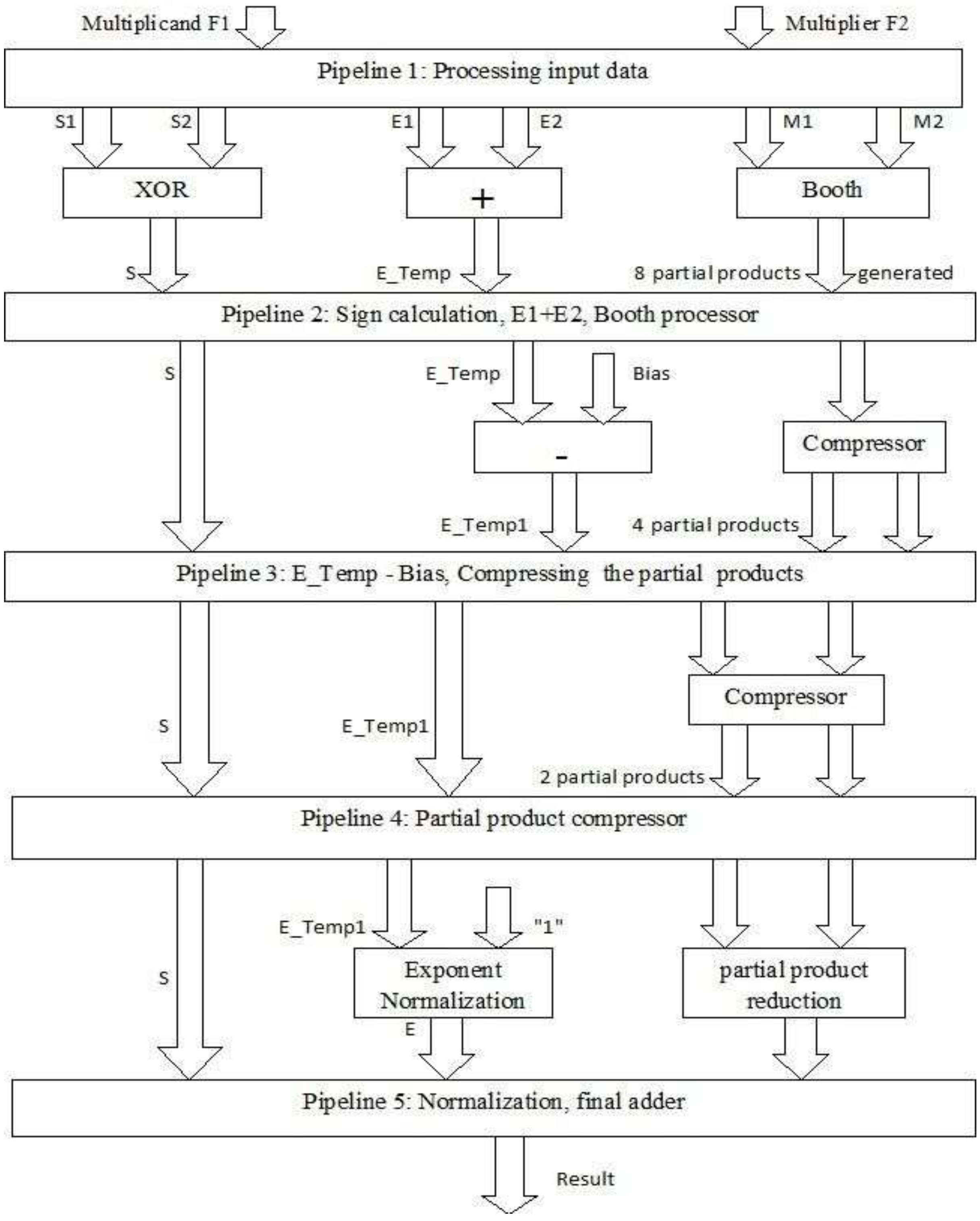

Multiplicand F1
Multiplier F2
Pipeline 1: Processing input data
S1
S2
E1
E2
M1
M2
XOR
+
Booth
S
E_Temp
8 partial products generated
Pipeline 2: Sign calculation, E1+E2, Booth processor
S
E_Temp
Bias
-
Compressor
E_Temp1
4 partial products
Pipeline 3: E_Temp - Bias, Compressing the partial products
S
E_Temp1
Compressor
2 partial products
Pipeline 4: Partial product compressor
E_Temp1
"1"
S
Exponent Normalization
partial product reduction
E
Pipeline 5: Normalization, final adder
Result

Figure 3.3 shows the various pipeline stages in the Multiplier.

Figure 3.3: Floating point multiplier with 5 pipelining stages

4. HARDWARE OF FLOATING POINT MULTIPLIER

4.1 Sign bit calculation

Multiplying two numbers results in a negative sign number if one of the multiplied numbers is of a negative value. By the aid of a truth table we find that this can be obtained by XORing the sign of twoinputs.

4.2 Unsigned Adder (for exponent addition)

This unsigned adder is responsible for adding the exponent of the first input to the exponent of the second input and subtracting the Bias (127) from the addition result (i.e. A_exponent + B_exponent - Bias). The result of this stage is called the intermediate exponent. The add operation is done on 8 bits, and there is no need for a quick result because most of the calculation time is spent in the significand multiplication process (multiplying 24 bits by 24 bits); thus we need a moderate exponent adder and a fast significandmultiplier.

An 8-bit ripple carry adder is used to add the two input exponents. As shown in Fig. 4.1 a ripple carry adder is a chain of cascaded full adders and one half adder; each full adder has three inputs (A, B, C_i) and two outputs (S, C_o). The carry out (C_o) of each adder is fed to the next full adder (i.e each carry bit "ripples" to the next full adder).

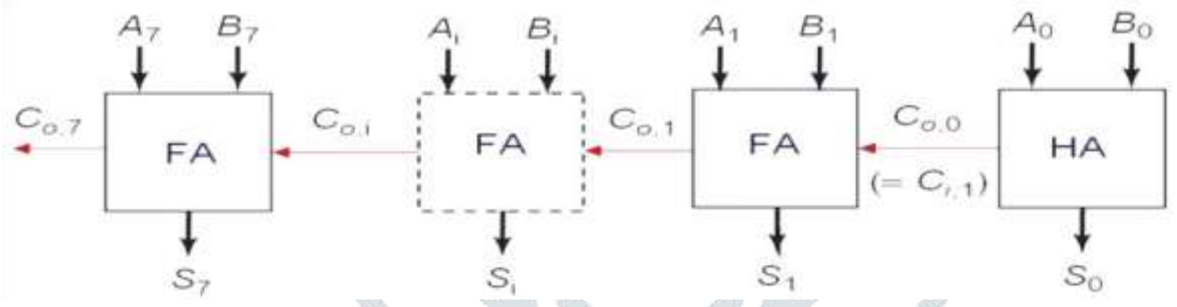


Figure 4.1 Ripple Carry Adder

The addition process produces an 8 bit sum (S_7 to S_0) and a carry bit ($C_{o,7}$). These bits are concatenated to form a 9 bit addition result (S_8 to S_0) from which the Bias is subtracted. The Bias is subtracted using an array of ripple borrow subtractors. A normal subtractor has three inputs(minuend (S), subtrahend (T), Borrow in (B_i)) and two outputs (Difference (R), Borrow out (B_o)). The subtractor logic can be optimized if one of its inputs is a constant value which is our case, where the Bias is constant ($127|_{10} = 001111111|_2$). Table 4.1 shows the truth table for a 1-bit subtractor with the input T equal to 1 which we will call “one subtractor (OS)”.

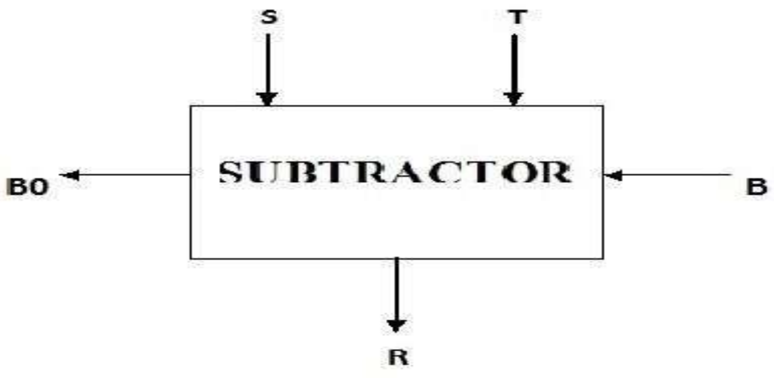


Figure 4.2: 1-Bit Subtractor

Table 4.1: 1-Bit Subtractor With The Input T=1

S	T	Bi	Difference(R)	B0
0	1	0	1	1
1	1	0	0	0
0	1	1	0	1
1	1	1	1	1

The Boolean equations (2) and (3) represent this subtractor:

$$Difference\ (R) = \overline{S \oplus B_i}(2)$$

$$Borrow_{out}(B_o) = \overline{S} + B_i(3)$$

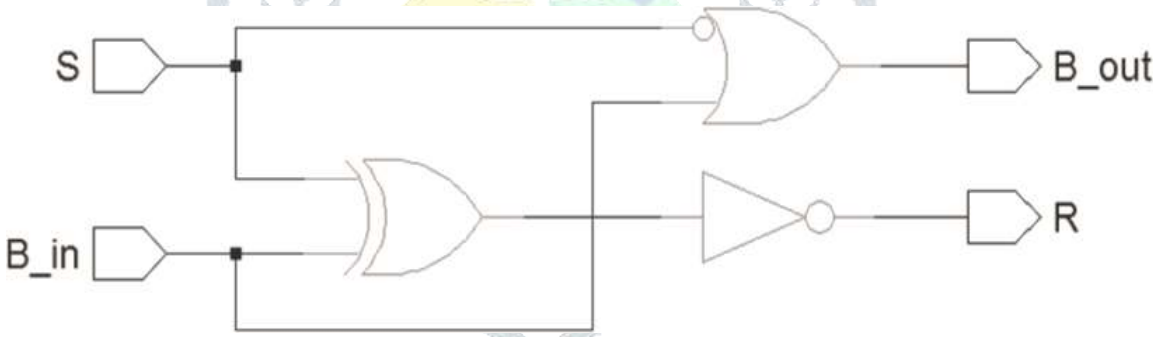


Figure 4.3: 1-bit subtractor with the input T = 1

Table 4.2 shows the truth table for a 1-bit subtractor with the input T equal to 0 which we will call “zero subtractor (ZS)” .

Table 4.2: 1-Bit Subtractor With The Input T=0

S	T	Bi	Difference(R)	B0
0	0	0	0	0
1	0	0	1	0
0	0	1	1	1
1	0	1	0	0

The Boolean equations (4) and (5) represent this subtractor:

$$Difference\ (R) = S \oplus B_i(4)$$

$$Borrow_{out}(B_o) = \overline{S} \cdot B_i(5)$$

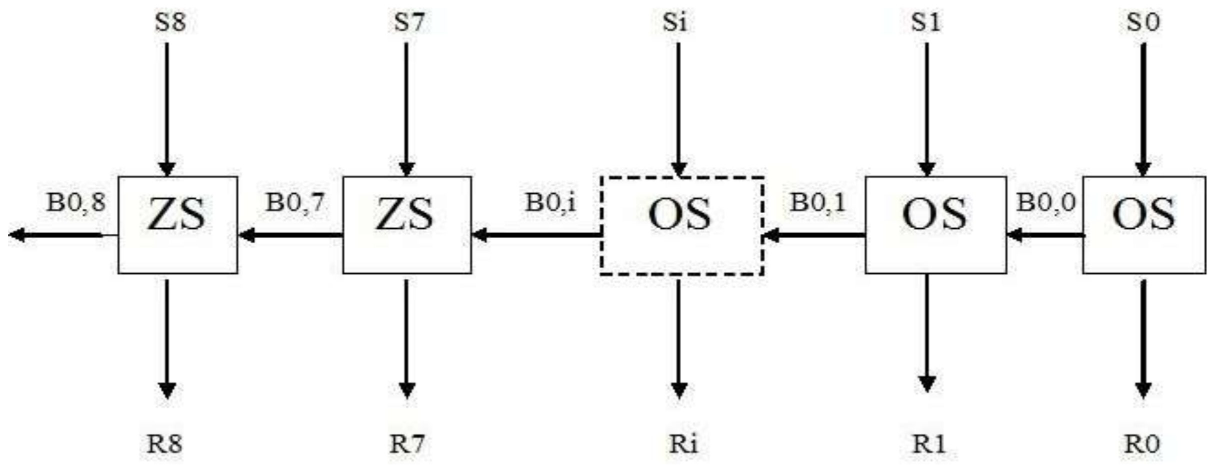


Figure 4.4: 1-bit subtractor with the input T = 0

Fig. 4.4 shows the Bias subtractor which is a chain of 7 one subtractors (OS) followed by 2 zero subtractors (ZS); the borrow output of each subtractor is fed to the next subtractor. If an underflow occurs then $E_{result} < 0$ and the number is out of the IEEE 754 single precision normalized numbers range; in this case the output is signaled to 0 and an underflow flag is asserted.

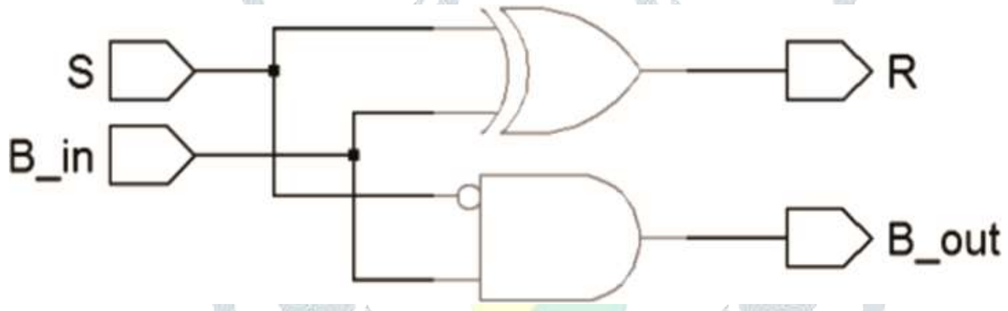


Figure 4.5: Ripple Borrow Subtractor

4.3 Unsigned Multiplier (for significand multiplication)

This unit is responsible for multiplying the unsigned significand and placing the decimal point in the multiplication product. The result of significand multiplication will be called the intermediate product (IP). The unsigned significand multiplication is done on 24 bit. Multiplier performance should be taken into consideration so as not to affect the whole multiplier's performance. A 24x24 bit carry save multiplier architecture is used as it has a moderate speed with a simple architecture. In the carry save multiplier, the carry bits are passed diagonally downwards (i.e. the carry bit is propagated to the next stage). Partial products are made by ANDing the inputs together and passing them to the appropriate adder.Carry save multiplier has three main stages:

1- The first stage is an array of halfadders.
2- The middle stages are arrays of full adders. The number of middle stages is equal to the significand size minustwo.
3- The last stage is an array of ripple carry adders. This stage is called the vector merging stage. The number of adders (Half adders and Full adders) in each stage is equal to the significand

size minus one. For example, a 4x4 carry save multiplier is shown in Fig. 6.5 and it has the followingstages:

1- The first stage consists of three halfadders.

2- Two middle stages; each consists of three fulladders.

3- The vector merging stage consists of one half adder and two full adders.

The decimal point is between bits 45 and 46 in the significand multiplier result. The multiplication time taken by the carry save multiplier is determined by its critical path. The critical path starts at the AND gate of the first partial products (i.e. a_1b_0 and a_0b_1), passes through the carry logic of the first half adder and the carry logic of the first full adder of the middle stages, then passes through all the vector merging adders. The critical path is marked in bold in Fig. 4.6.

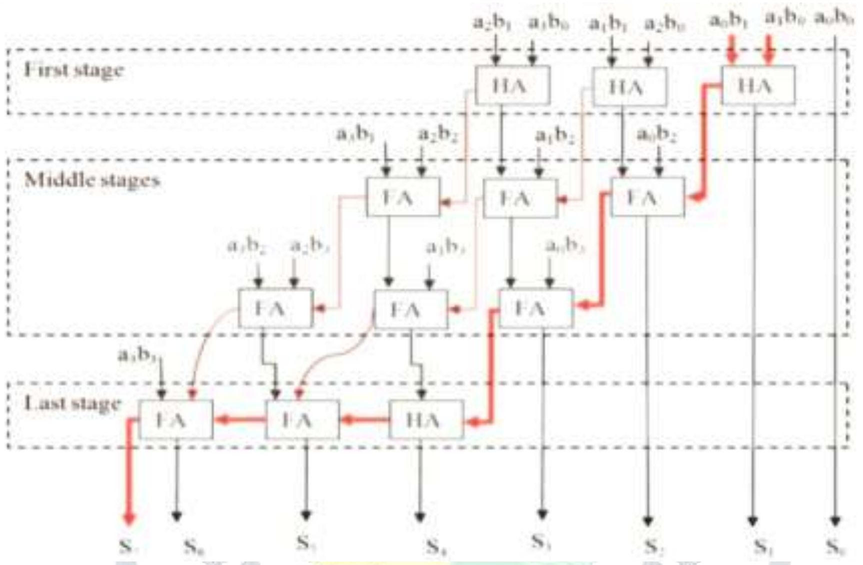

First stage
Middle stages
Last stage
HA
FA
a_2b_1 a_3b_0 a_1b_1 a_2b_0 a_0b_1 a_1b_0 a_0b_0
a_3b_1 a_2b_2 a_1b_2 a_0b_2
a_3b_2 a_2b_3 a_1b_3 a_0b_3
a_3b_3
S_7 S_6 S_5 S_4 S_3 S_2 S_1 S_0

Figure 4.6: 4x4 bit Carry Save multiplier

1- Partial product: $a_ib_j = a_i$ andb_j

2- HA: halfadder

3- FA: fulladder

4.4 Normalizer

The result of the significand multiplication (intermediate product) must be normalized to have a leading '1' just to the left of the decimal point (i.e. in the bit 46 in the intermediate product). Since the inputs are normalized numbers then the intermediate product has the leading one at bit 46 or47

1- If the leading one is at bit 46 (i.e. to the left of the decimal point) then the intermediate product is already a normalized number and no shift isneeded.

2- If the leading one is at bit 47 then the intermediate product is shifted to the right and the exponent is incremented by1.

The shift operation is done using combinational shift logic made by multiplexers.

4.5 Underflow/Overflow Detection

Overflow/underflow means that the result's exponent is too large/small to be represented in the exponent field. The exponent of the result must be 8 bits in size, and must be between 1 and 254 otherwise the value is not a normalized one. An overflow may occur while adding the two exponents or during normalization. Overflow due to exponent addition may be compensated during subtraction of the bias; resulting in a normal output value (normal operation). An underflow may occur while subtracting the bias to form the intermediate exponent. If the intermediate exponent < 0 then it's an underflow that can never be compensated; if the intermediate exponent = 0 then it's an underflow that may be compensated during normalization by adding 1 toit.

When an overflow occurs an overflow flag signal goes high and the result turnsto ±Infinity (sign determined according to the sign of the floating point multiplier inputs). When an underflow occurs an underflow flag signal goes high and the result turns to ±Zero (sign determined according to the sign of the floating point multiplier inputs). Denormalized numbers are signaled to Zero with the appropriate sign calculated from the inputs and an

underflow flag is raised. Assume that E_1 and E_2 are the exponents of the two numbers A and B respectively; the result's exponent is calculated by (6)

$$E_{result} = E_1 + E_2 - 127 \qquad (6)$$

E_1 and E_2 can have the values from 1 to 254; resulting in E_{result} having values from -125 (2-127) to 381 (508-127); but for normalized numbers, E_{result} can only have the values from 1 to 254. Table III summarizes the E_{result} different values and the effect of normalization on it.

Table 4.3 Normalization Effect On Result's Exponent And Overflow/Underflow Detection

Eresult	**Category**	**Comments**
$-125 \leq E_{result} < 0$	Underflow	Can't be compensated during normalization
$E_{result} = 0$	Zero	May turn to normalized number during normalization (by adding 1 to it)
$1 < E_{result} < 254$	Normalized number	May result in overflow during normalization
$255 \leq E_{result}$	Overflow	Can't be compensated

4.6 Implementation AndTesting

The whole multiplier (top unit) was tested against the Xilinx floating point multiplier core generated by Xilinx coregen. Xilinx core was customized to have two flags to indicate overflow and underflow, and to have a maximum latency of three cycles. Xilinx core implements the "round to nearest" rounding mode.

A test bench is used to generate the stimulus and applies it to the implemented floating point multiplier and to the Xilinx core then compares the results. The floating point multiplier code was also checked using Design Checker .

Design Checker is a linking tool which helps in filtering design issues like gated clocks, unused/undriven logic, and combinational loops. Also the speed of Xilinx core is affected by the fact that it implements the round to nearest rounding mode.

5. SOFTWAR E DETAILS

5.1 Algorithm:

As stated in the introduction, normalized floating point numbers have the form of Z= (-1^{S}) * $2^{(E-\underline{Bias})}$ * (1.M). To multiply two floating point numbers the following is done:

Step 1: Multiplying the significand; i.e. ($1.M_1*1.M_2$)

Step 2: Placing the decimal point in the result

Step 3: Adding the exponents; i.e. ($E_1 + E_2$ – *Bias)*

Step 4: Obtaining the sign; i.e. s_1 xor s_2

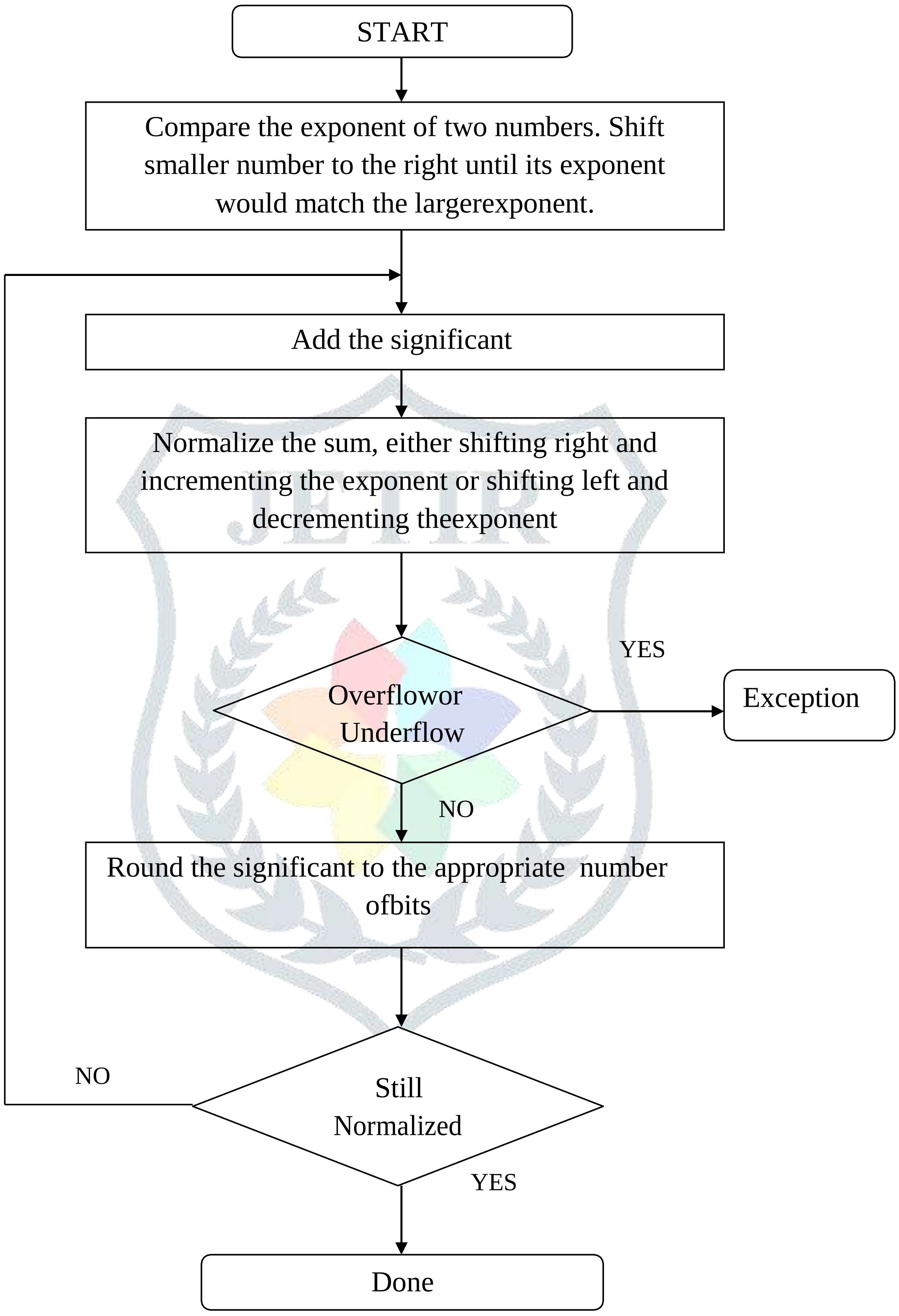
Step 5: Normalizing the result; i.e. obtaining 1 at the MSB of the results'

Significand

Step 6: Rounding the result to fit in the available bits

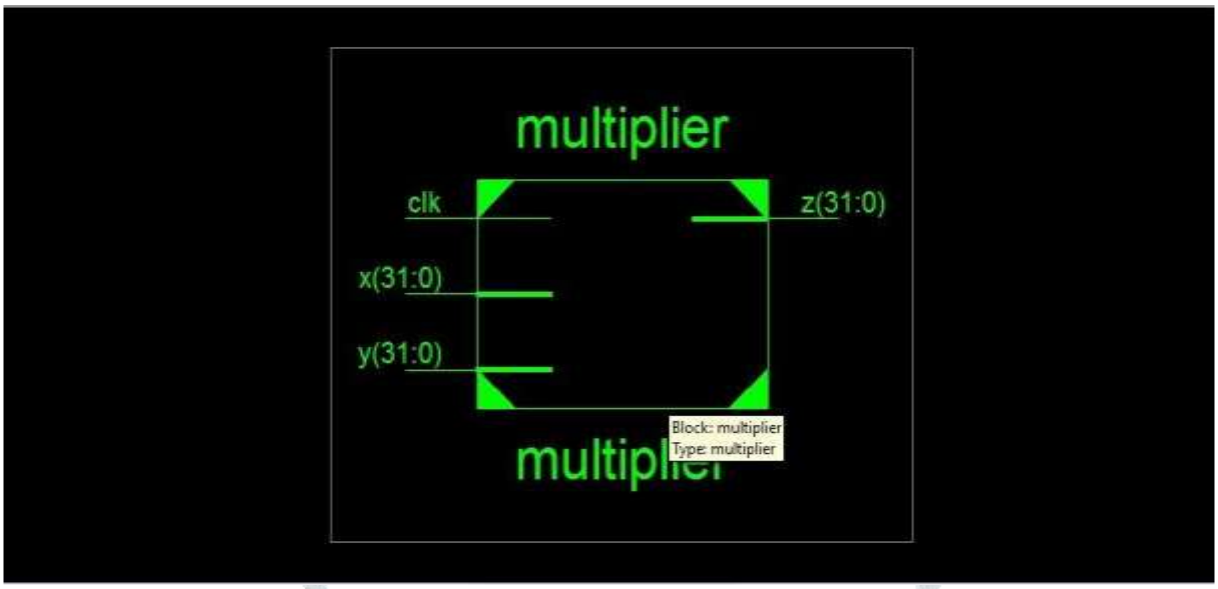
Step 7: Rounding the result to fit in the available bits

5.2 FlowChart:

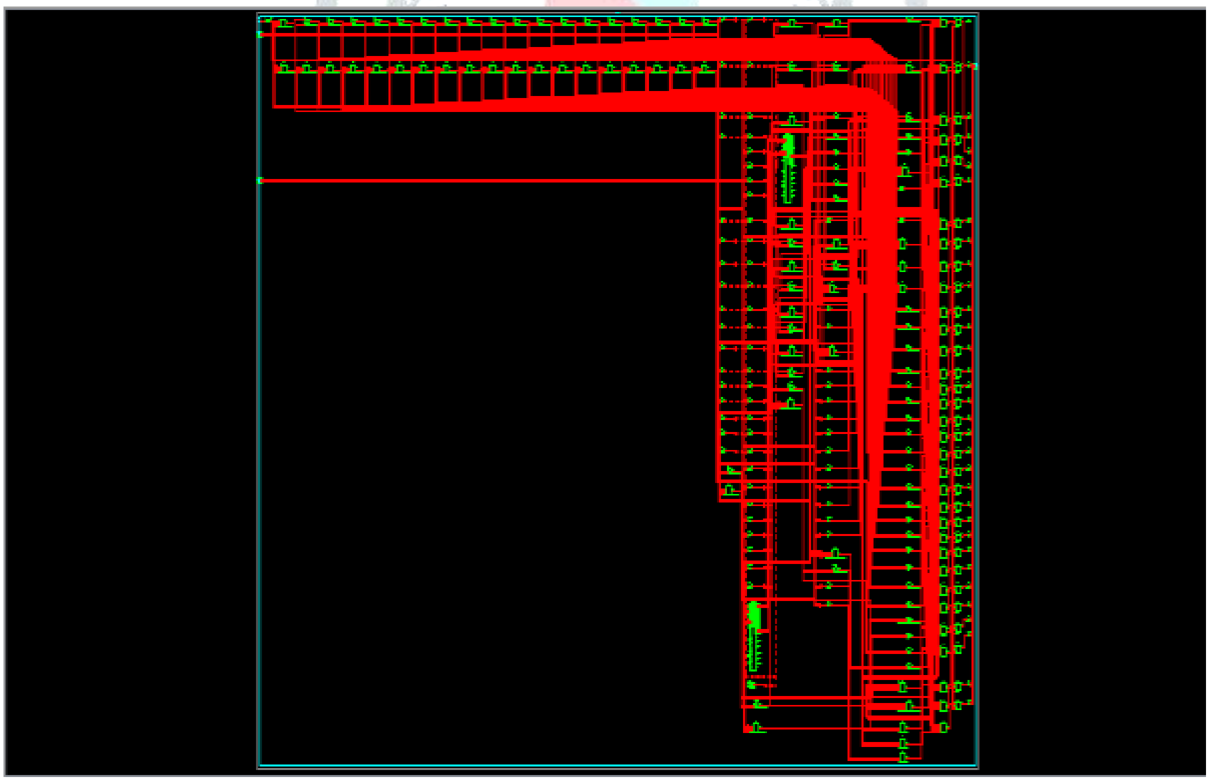

START
Compare the exponent of two numbers. Shift smaller number to the right until its exponent would match the largerexponent.
Add the significant
Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing theexponent
Overflowor Underflow
YES
Exception
NO
Round the significant to the appropriate number ofbits
NO
Still Normalized
YES
Done

6. RESULTS

6.1 RTL View



6.2 Technological View

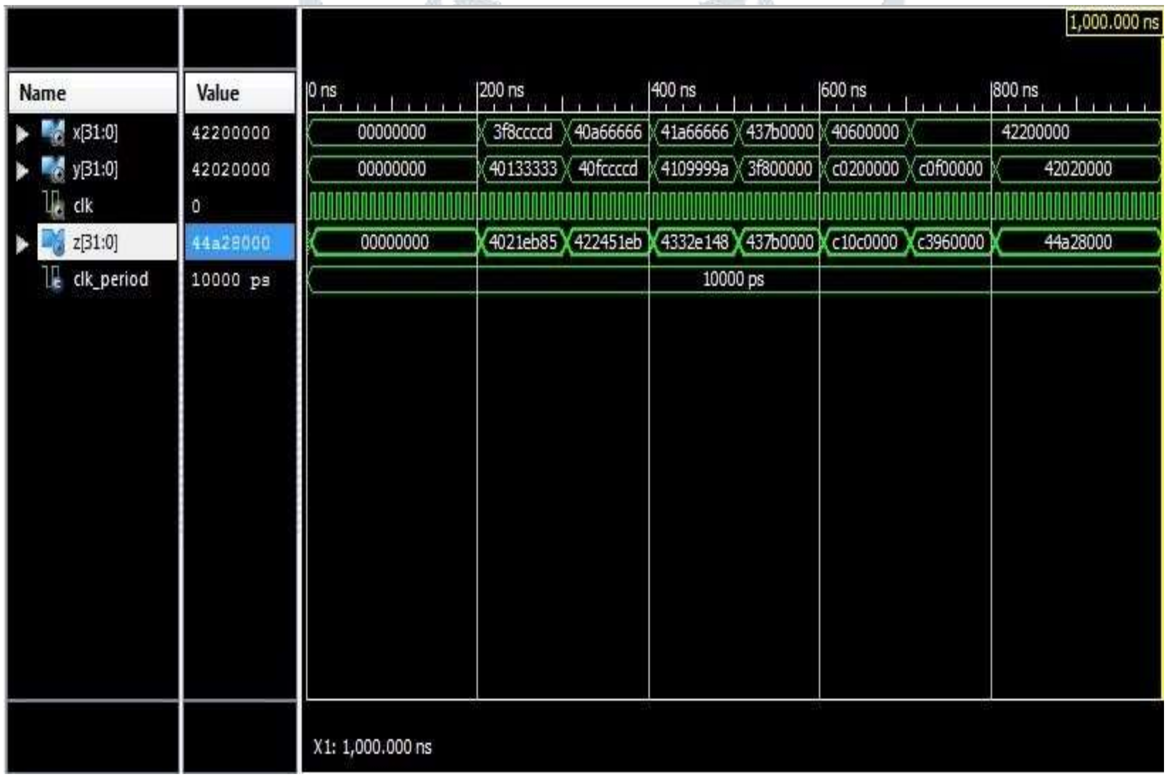


6.3 Synthesis Report

Below Table Shows the Synthesis for final 32-bit floating point multiplier.

	Spartan 6 XC6SLX4	Virtex 7 XC7V285T
No. of Slices	1/4800(1%)	1/357600(1%)
No. of LUTs	72/2400(3%)	72/178800(1%)
No. of Bonded IOBs	97/102(95%)	97/600(16%)
Minimum Period	3ns	2.95ns
Maximum Frequency	333.33MHz	338.98MHz
Power Dissipation	14.4mW	545.7mW
Total memory usage	197MB	695MB

6.4 Simulation Results



```
x[31:0]     42200000   00000000 | 3f8ccccd | 40a66666 | 41a66666 | 437b0000 | 40600000 |  | 42200000
y[31:0]     42020000   00000000 | 40133333 | 40fccccd | 4109999a | 3f800000 | c0200000 | c0f00000 | 42020000
clk         0
z[31:0]     44a28000   00000000 | 4021eb85 | 422451eb | 4332e148 | 437b0000 | c10c0000 | c3960000 | 44a28000
clk_period  10000 ps   10000 ps
```

7. ADVANTAGES & DISADVANTAGES

7.1 Advantages:

- They can represent values between integers.
- Because of the scaling factor, they can represent a much greaterrange ofvalues.

7.2 Disadvantages:

- Rounds off largenumbers.

8. APPLICATIONS

8.1 Floating point multiplication is a most widely used operation in DSP/Math processors, digitalcomputers.

8.2 Floating Point Arithmetic is extensively used in the field of banking, tax calculation, currency conversion, and other financialareas.

9. CONCLUSION

Single Precision multiplier is designed and implemented using VHDL/Verilog and simulated using Modelsim Simulator. One of the important aspects of the presented and implemented design method is that it can be applicable to all kinds of floating-point multipliers. The presented design is compared with conventional multipliers via synthesis. The synthesis results showed that the proposed design is much faster and more efficient than conventional multipliers. The designed multiplier conforms to IEEE 754 standard for floating point numbers. In this implementation default rounding mode is used. The design is also verified for overflow and underflow cases and other exceptions defined by IEEE standard.

10. FUTURE SCOPE

The future scopes of this project are to implement the proposed floating point arithmetic unit using Field-Programmable Gate Arrays (FPGAs). This project presents an implementation of a floating point multiplier that supports the IEEE 754-2008 binary interchange format, the multiplier doesn’t implement rounding and just presents the significand multiplication result as is (48 bits). This gives better precision if the whole 48 bits are utilized in another unit i.e. a floating point adder to form a MACunit.

11. REFERENCES

1.M Al-Ashrafy, A Salem and W Anis, “An efficient implementation of floating point multiplier”, in Proc. IEEE Electronics, Communications and Photonics Conference (SIECPC), Saudi International, April 24-262011.

2. Tejaswini H N, Dr. Ravishankar C V, “Single Precision Floating point Numbers Multiplication using standard IEEE 754” International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE) Volume 4, Issue 6, June2015.

3. Surendra Singh Rajpoot, Nidhi Maheshwari, D S Yadav, “ Design and implementation of efficient 32-bit floating point multiplier "International Journal Of Engineering And Computer Science ISSN:2319-7242 Volume 2 Issue 6 June 2013 Page No.2098-2101.

4. Priyanka Koneru, M Tinnanti Sreenivasu, Addanki Purna Ramesh, "Asynchronous Single Precision Floating Point Multiplier Using Verilog HDL" International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE) Volume 2, Issue 11, November2013.

5. Mr. S.S.Mohanasundaram, A.Nirmal kumar, T.Arul prakash, "Design of Floating Point Multiplier Using Vedic Mathematics" IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 2 Issue 1, January2015.