IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 3 Issue 2, February 2016.
www.ijiset.com

ISSN 2348 – 7968

Low Power Floating-Point Multiplier Based On Vedic Mathematics

K.Prashant Gokul, M.E(VLSI Design), Sri Ramanujar Engineering College, Chennai
Prof.S.Murugeswari., Supervisor,Prof.&Head,ECE.,**SREC**.,**C**hennai-600 127.

ABSTRACT

Fast Fourier transform (FFT) coprocessor, having significant impact on the performance of communication systems, has-been a hot topic of research for many years. The FFT function consists of consecutive multiply add operations over complex numbers, dubbed as butterfly units. Applying floating-point (FP) arithmetic to FFT architectures, specifically butterfly units, has become more popular recently. It offloads compute-intensive tasks from general-purpose processors by dismissing FP concerns (e.g., scaling and overflow/underflow). However, the major downside of FP butterfly is its slowness in comparison with its fixed-point counterpart. This reveals the incentive to develop a high-speed FP butterfly architecture to mitigate FP slowness. This brief proposes a fast FP butterfly unit using a devised FP fused-dot product-add (FDPA) unit, to compute AB ± CD ±E, based on binary signed-digit (BSD) representation. The FP three-operand BSD adder and the FP BSD constant multiplier are the constituents of the proposed FDPA unit. A carry-limited BSD adder is proposed and used in the three-operand adder and the parallel BSD multiplier so as to improve the speed of the FDPA unit. Moreover, modified Booth encoding is used to accelerate the BSD multiplier. The synthesis results show that the proposed FP butterfly architecture is much faster than previous counterparts but at the cost of more area.

1. INTRODUCTION

The main objective of this thesis is to design a high-speed co-processor based onredundant number systems. In particular, the architectures of the two of most commonly used coprocessors will be redesigned based on redundant number systems to achieve improved performance. The first part of this thesis investigates the advantages and costs of designing high-speed floating-point FFT architectures using redundant number systems. New architectures are proposed and compared to previous works. The second part is devoted to proposing decimal arithmetic co-processors with architectures based on redundant number systems comparing them with previous works. A complete decimal arithmetic unit is designed accordingly, with four basic decimal arithmetic operations: addition, subtraction, multiplication and division. An architecture based on redundant number systems is also proposed for computing decimal square-root.

EXISTING SYSTEM

The main drawback of the FP operations is their slowness in comparison with the fixed-point counterparts. A way to speed up the FP arithmetic is to merge several operations in a single FP unit, and hence save delay, area, and power consumption [2]. Using redundant number systems is another well-known way of overcoming slowness, where there is no word-wide carry propagation within the intermediate operations.

LIMITATIONS

The FP operations are their slowness in comparison with the fixed-point counterparts.

PROPOSED SYSTEM

A butterfly architecture using redundant FP arithmetic, which is useful for FP FFT coprocessors and contributes to digital signal processing applications. Although there are other works on the use of redundant FP number systems, they are not optimized for butterfly architecture in which both redundant FP multiplier and adder are required. All the significant are represented in binary signed digit(BSD) format and the corresponding carry-limited adders designed. Design of FP constant multipliers for operands withes significant. Design of FP three-operand adders for operands with BSD significant. Design of FP fused-dot-product-add (FDPA) units for operands with BSD significant.

ADVANTAGE

- A way to speed up the FP arithmetic is to merge several operations in a single FP unit, and hence save delay, area, and power consumption.
- Using redundant number systems is another well-known way of overcoming FP slowness, where there is no word-wide carry propagation within the intermediate operations.

2. ARCHITECTURE

The proposed butterfly is actually a complex Fused-Multiply-Add followed by a complex addition with floating point operands. Expanding the complex numbers, Fig. 2.1 depicts the required modules for a butterfly unit. A naive approach to implement Fig. 2.2 is to cascade floating-point operations i.e., floating-point multiplication followed by two cascaded floating-point addition/subtraction. A more efficient approach is to merge the floating-point multiplication with the first floating-point addition/subtraction. This method leads to a floating-point Fused-Multiply-Add followed by a floating-point addition/subtraction.

Using the floating-point Fused-Multiply-Add, as is discussed in Section 1.5, would save time, area and power. However, the butterfly function cannot be directly implemented by Fused-Multiply-Add (i.e., A+BC). In order to circumvent this problem a Dot-Product unit is required, which is an extension to Fused-Multiply-Add operation.

A Dot-Product unit computes AB+CD or AB-CD. This unit is capable of saving more time, area and power than Fused-Multiply-Add. The reason lies in the fact that a Dot-Product unit combines more floating-point operands and hence eliminating more intermediate Normalization, Rounding and Leading-Zero-Detection. Combining floating-point operations, although seems interesting and an easy way of saving time, area and power, leads to precision loss which need to be taken care of, meticulously.

For example, in a Fused-Multiply-Add unit the combination of the multiplication with the addition removes the intermediate Rounding, Normalization and Leading-Zero-Detection after the multiplication. However, it is important to pass wider operands (more number of bits) to the floating-point addition such that the required precision can be recovered at the end of the addition.

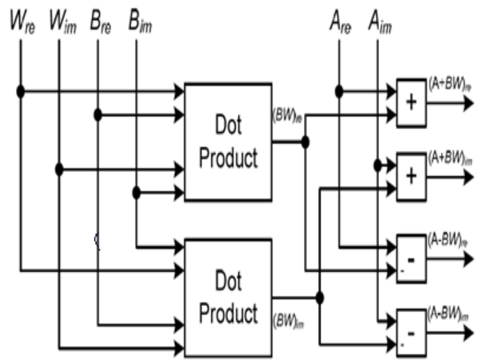

Fig 2.1 Butterfly Architecture with Dot-Product Units

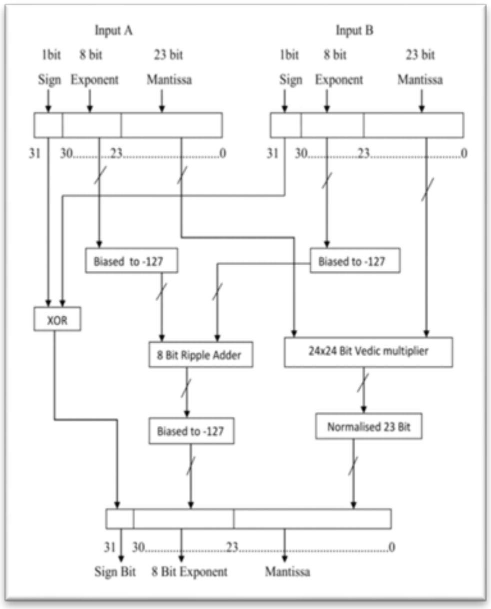

Figure 2.2 Vedic Multiplier Block Diagram

3. ARCHITECTURE DESCRIPTION

The Proposed Redundant Floating-Point Multiplier

Floating-pointmultiplication,consists of operations on the exponents and those on the significands. The former is just a simple addition of the exponent; although, there may be a need for exponent adjustment in the normalization and rounding phase.
The latter, however, is the most time-consuming part of a floating-point multiplier. Multiplication over significands consists of three major steps called: 1) Partial Product Generation, 2) Partial Product Reduction and 3) Final Addition.

The proposed multiplier, likewise other parallel multipliers, consists of two major steps, namely, partial product generation (PPG) and partial product reduction (PPR). However, contrary to conventional multipliers, the proposed multiplier keeps the product in redundant format and hence there is no need for the final carry-propagating adder.

Partial Product Generation (PPG)

The partial product generation, in a 2's complement representation of the multiplicand and multiplier, consists of arrays of AND operation such that each bit of the multiplier is ANDed to the whole bits of the multiplicand. This is not the case if the operands are represented in redundant format and/or Booth encoding. For example, if the multiplier is represented in the modified Booth encoding, the partial product generation looks like the circuitry shown in Fig. 3.1

Partial product generation of a redundant multiplier is even more complicated, since the cardinality of the multiplier's digit-set is more than the radix. Generating the multiples of the multiplicand is easy (shift and negation) for ±2x and ±1x; however, ±3x and ±5x (if exists) involve an addition. Fig. 3.1 shows how these multiples are generated. Consequently, higher redundancy factor leads to more complicated partial product generation; although it provides faster redundant addition.

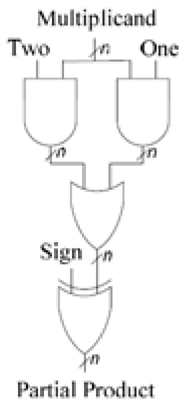

Figure 3.1 Patial Product Generation –Booth Encoding

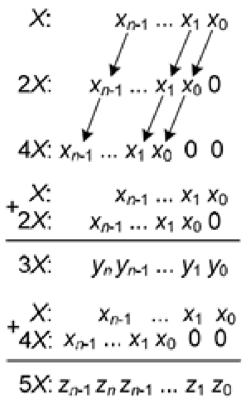
X: x_{n-1} ... x_1 x_0
2X: x_{n-1} ... x_1 x_0 0
4X: x_{n-1} ... x_1 x_0 0 0
X: x_{n-1} ... x_1 x_0
+2X: x_{n-1} ... x_1 x_0 0
3X: y_n y_{n-1} ... y_1 y_0
X: x_{n-1} ... x_1 x_0
+4X: x_{n-1} ... x_1 x_0 0 0
5X: z_{n-1} z_n z_{n-1} ... z_1 z_0

Figure 3.2 Generating the Multiples of the Multiplicand

Partial Product Reduction (PPR)

Partial product reduction phase is actually a multi-operand addition of the partial products generated in the partial product generation phase.

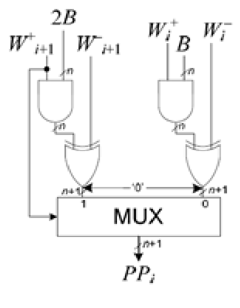

Figure 3.3 Generation of the i-th partial product

Generalizing this approach for *p* operands (i.e., [*p*:1] reduction block) is described next. Reduction of *p* operands can be divided into two parts, each of which reduces *p*/2 operands (i.e., [*p*/2:1] reduction block); and then add the outputs together.

Fig. 3.2 shows the reduction of *p* operands using reduction blocks for *p*/2 operands. Each of those [p/2:1] modules could be further divided into two sub modules. Continuing this approach leads to a [2:1] reduction block which is known as a carry-propagating adder. Consequently, log(*p*) steps are required to reduce *p* operands to 1.

Reduction by columns is done by modules called counters or compressors. These modules take *p* bits, all at the same weighted position, and generate *q* bits of adjacent weights.

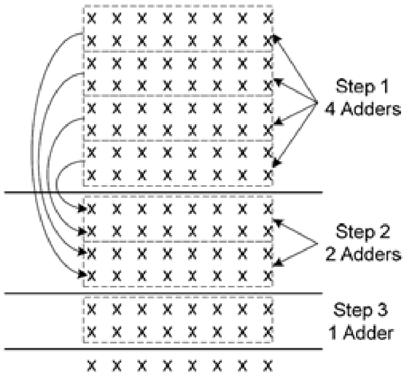

Figure 3.4 Partial Product Reduction by Rows

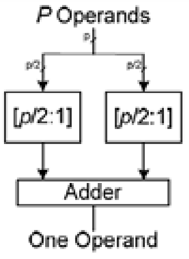

Figure 3.5 [p:1] reduction block based on [p/2:1] blocks

Larger counters can be built using full-adders and half-adders (i.e., [2:2] counter). For example, Fig. 3.3 depicts a [7:3] counter implemented by full-adders. The numbers next to each wire show the weight of that input/output. It should be noted that in a counter all same weighted outputs can be added together using a full-adder. The final output is a 3-bit number which counts the number of 1s in the input *p*.

Having multiple counters working in parallel leads to a multi-column counter which can be used to reduce several columns. In a multi-column counter, there are multiple counters each of which performing on a single weighted position and passes the carries to the next higher weighted column. For example, Fig. 3.4 shows a [7:2] compressor which passes the carries to the next higher weighted position and receives input carries from the lower weighted position. It should be noted that this module is called [7:2] compressor, because 2 bits are not enough to count 7 bits.

Therefore, partial product reduction phase of a multiplier can be taken care of by multi-column compressors. These modules take *p* bits of a single column (same weights) and reduce it into two bits per column. For example, a 54-by-54 bit multiplier generates 54 partial products. In order to reduce these partial products one could design a [54:2] compressor to reduce them to only two operands. Then a carry-propagating adder, over the two operands, generates the final product. Large compressors/counters can be built based on smaller ones. For instance, Fig. 3.6 shows a [16:2] compressor based on [4:2] compressors (Fig. 3.5). The PPR step of the proposed

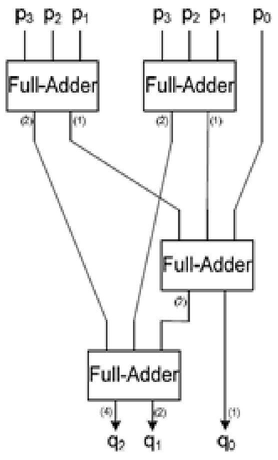

Figure 3.6 [7:3] Counter by Full Adders

Given that partial products are all represented in a redundant encoding i.e., Binary-Signed-Digit, there is a need for an adder/counter that works on BSD digits. This carry-limited addition circuitry is shown in Fig. 3.6, where capital (small) letters symbolize negabits (posibits). The white dots are logical NOT operators required over negabit signals [6]. The critical path delay of this adder consists of three full-adders.

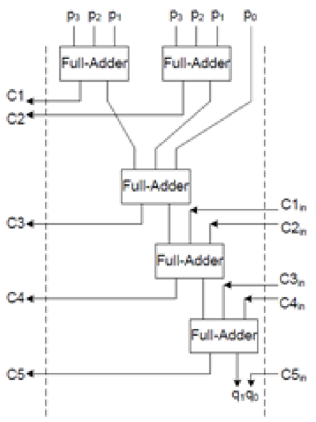

Figure 3.7 [7:2] Compressor by Full Adders

Since the BSD adder is actually a carry-limited adder, reducing the partial product using this adder can be deemed as a reduction-by-column or reduction-by-row approach. In either case, the major constituent of the PPR step is the proposed carry-limited addition over the operands represented in BSD format.

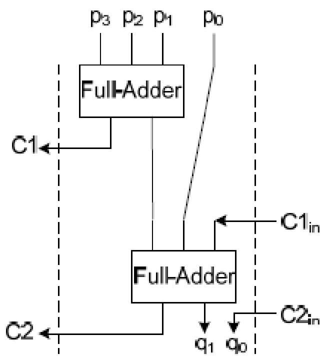

Figure 3.14: [4:2] Compressor by Full-Adders

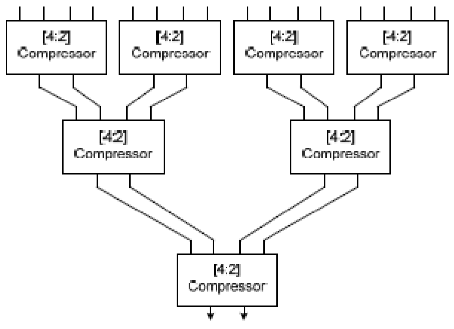

Figure 3.15: [16:2] Compressor by [4:2] Compressors

44

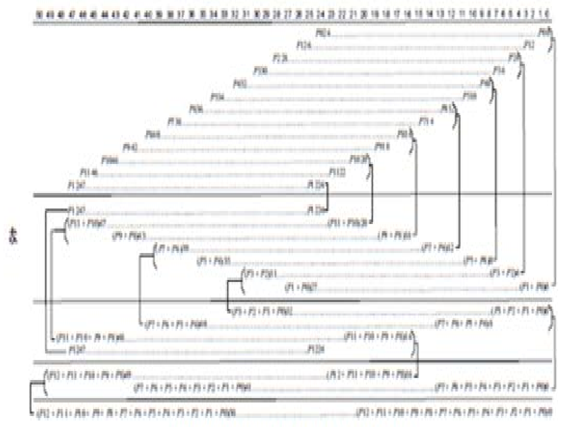

Figure 3.10 Partial Product Reduction Tree

The numbers in the first row of Fig. 3.8 show the bit positions of each partial product. For example, P024 symbolizes bit position 24 of first partial product (PP0). Note that the last partial product has 24-(binary position) width (instead of 25), given that the 26th bit of *W* is always 1 (hidden bit).

Now that the product is generated, it is required to determine how many bits are required to be passed to the three-operand adder so as to meet the precision requirements. As is shown in Fig. 3.8, the final product is a 51-bit number represented in Binary-Signed-Digit encoding. Given the normalized single precision formats of the inputs (*B* is in ±[1, 2) and *W* in [1, 2)), the final product is in ±[1, 4).

If the multiplier's operands were represented in standard IEEE format (i.e., each with 24 bits), the final product would fit into 48 binary positions (47…0). Consequently, positions 45 down to 0 would be fractions (see Fig. 3.9). However, the product out of the proposed BSD multiplier has 51 binary positions. Given that the value of this number is the same as that of the standard product, the 45 least significant positions are fractions.

MULTIPLICATION STEPS OF URDHVATRIYAKBHYAM

The Vedic multiplication system is based on 16 Vedic sutras or aphorisms, which describes natural ways of solving a whole range of mathematical problems. Out of these 16 Vedic

Sutras the Urdhva-triyakbhyam sutra is suitable for this purpose.

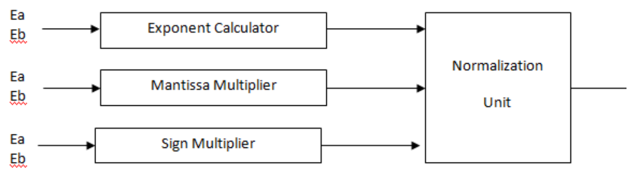

Figure 3.10 Floating Point Multiplier

The pipelining stages are imbedded at the following locations:

1) In the middle of the significand multiplier, and in the middle of the exponent adder (before the bias subtraction).

2) After the significand multiplier, and after the exponent adder.

3) At the floating point multiplier outputs (sign, exponent and mantissa bits). Three pipelining stages mean that there is latency in the output by three clocks. The synthesis tool "retiming" option was used so that the synthesizer uses its optimization logic to better place the pipelining registers across the critical path.

4. OUTPUT WAVEFORM

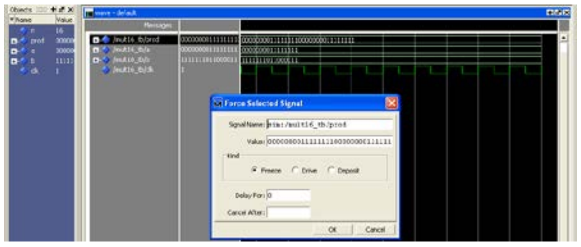

Fig 4.1 a Output

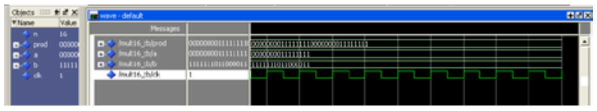

Fig 4.1b Output

5. CONCLUSION

High-speed FP butterfly architecture, which is faster than previous works is presented in the Phase I. The reason for this speed improvement is two-fold: 1) BSD representation of the significant which eliminates carry-propagation and 2) the new FDPA unit proposed in this Phase I. The unit combines multiplications and additions required in FP butterfly; thus higher speed is achieved by eliminating extra LZD, normalization, and rounding units.

6. FUTURE SCOPE

In the future work, further research will be done by applying dual-path FP architecture to the three-operand FP adder and using other redundant FP representations. Use of improved techniques in the termination phase of the design (i.e., redundant LZD, normalization, and rounding) would lead to faster architectures, though higher area costs are expected.

7. REFERENCES

[1] Amir Kaivani and SeokbumKo , "Floating-Point Butterfly Architecture Based on
Binary Signed-Digit Representation" ,Apr. 2015
[2] J. Sohn and E. E. Swartzlander, Jr., "Improved architectures for a floating-point fused dot product unit," in Proc. IEEE 21st Symp.Comput. Arithmetic, Apr. 2013, pp. 41–48.
[3] IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754-2008,
Aug. 2008, pp. 1–58.
[4] B. Parhami, Computer Arithmetic: Algorithms and Hardware Designs, 2nd ed. New York, NY, USA: Oxford Univ. Press, 2010.
[5] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," Math.Comput., vol. 19, no. 90, pp. 297–301, Apr. 1965.
[6] A. F. Tenca, "Multi-operand floating-point addition," in Proc. 19th IEEE Symp. Comput. Arithmetic, Jun. 2009, pp. 161–168.
[7] Y. Tao, G. Deyuan, F. Xiaoya, and R. Xianglong, "Three-operand floating-point adder," in Proc. 12th IEEE Int. Conf. Comput. Inf. Technol., Oct. 2012, pp. 192–196.
[8] A. M. Nielsen, D. W. Matula, C. N. Lyu, and G. Even, "An IEEE compliant floating-point adder that conforms with the pipeline packetforwardingparadigm," IEEE Trans. Comput., vol. 49, no. 1, pp. 33–47, Jan. 2000.
[9] P. Kornerup, "Correcting the normalization shift of redundant binary representations," IEEE Trans. Comput., vol. 58, no. 10, pp. 1435–1439, Oct. 2009.

[10] 90 nm CMOS090 Design Platform, STMicroelectronics, Geneva, Switzerland, 2007.
[11] E. E. Swartzlander, Jr., and H. H. Saleh, “FFT implementation with fused floating-point operations,” IEEE Trans. Comput., vol. 61, no. 2, pp. 284–288, Feb. 2012.