Dual-Path Architecture of Floating-Point Dot Product Computation

Yao Tao An Jianfeng Gao Deyuan Fan Xiaoya
Department of Computer Science and Engineering
Northwestern Polytechnical University
Xi'an, P.R.China
e-mail: ytpaul@mail.nwpu.edu.cn {Gaody, Fanxy}@nwpu.edu.cn

***Abstract* —Dot product computation is widely used in many algorithms, such as FFT and DCT. This paper proposes a floating-point dot product architecture based on the multiple-path method. This architecture could perform A × B+C × D as a single operation. The speed of the dual-path architecture implemented in single precision format is faster by 32% and 5.45% than the speed of a network approach using traditional adders and multipliers and the speed of the basic dot product architecture, respectively.**

Keywords- dot product computation; single operation; multple-path method

I. INTRODUCTION

In many fields such as DSP, Graphic processing and scientific computing, the floating-point (FP) dot product computation is one of frequently used operations. Specially, FFT, DCT algorithms and complex numbers' multiplications require the computer to support the 2 terms dot product computation well. Dot product computation is considered as one of basic arithmetic operations, like addition, subtraction, multiplication, division [1].

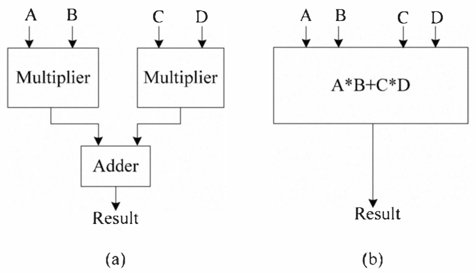

Figure 1. Two types of architectures to perform dot product. (a) Network architecture (b) Fused architecture.

Traditional method to perform dot product computation is using a network of one adder and two multipliers shown in Fig. 1 (a). The network method's latency is larger and could accumulate more rounding errors because there are three rounding operations performed in the computation. The fused architecture proposed in [2] [3] [4] shown in Fig. 1 (b) could perform the dot product computation as a single operation, thus only once rounding operation is required. The fused architecture is faster and more accurate than the network method does. In this paper, we try to speed up the fused architecture of dot product computation.

The multiple-path approach divides the critical path into several paths according to the exponent difference. Because every path only needs dealing with a less complex scenario, the delay could be saved over the traditional implementations. Multiple paths approach is widely used to speedup the FP adder [5] [6] and MAF (Multiply-Addition Fused) [7] [8] effectively.

In this paper, we propose a dual-path architecture of dot product computation according to the multiple-path philosophy. The new architecture divides the alignment shift and normalization shift into two different paths to reduce the critical path's delay. Moreover, a novel sticky bit computing method based on the pattern detection is presented to reduce the cost of our architecture. The architecture could support all four IEEE rounding modes [9], and produce the correctly rounded result. We implement the dual-path architecture in single precision data format. Compared to the network of traditional adders and multipliers, the dual path architecture is 32% faster. Compared to the basic fused dot product architecture, the speed of dual path architecture is faster by 5.45%.

The rest of the paper is organized as follows: Section II give the notation introduction. Section III introduces the basic fused architecture. In Section IV a detailed design of the dual-path architecture has been presented combining optimization techniques. Section V includes two parts, function verification and performance analysis, followed by conclusion in Section VI.

II. NOTATION

The 2 terms FP dot product computation (FDP2) is described in formula (1):

$$Fr = A \times B + C \times D \tag{1}$$

Every FP number in the formula (1) is formed by a triple(s, e, m), where s is the sign of the datum, e is the biased exponent, m represents the significand (including the hided leading one) The width of significand is f. The FP number is defined in formula (2):

$$F = (-1)^s \times 2^{(e-bias)} \times m \tag{2}$$

where *bias* is the exponent bias defined by IEEE Std.754[9].

The outputs of multiply array are represented as carry-save form in formula (3):

$$\begin{cases} ma \times mb = ps_1 + pc_1 \\ mc \times md = ps_2 + pc_2 \end{cases} \tag{3}$$

where *ma*, *mb*, *mc*, *md* are significands of *A*, *B*, *C*, *D* shown in formula (1), respectively. (ps_1, pc_1) is the first multiply array's outputs. (ps_2, pc_2) is the second multiply array's outputs.

The exponent difference $\delta = (ea+eb) - (ec+ed)$, where *ea*, *eb*, *ec*, *ed* are exponents of *A*, *B*, *C*, *D*, respectively. *sign_d* is the sign of δ. *d* is the absolute value of δ, namely $d = |\delta|$.

We define data (ms_1, mc_1) and (ms_2, mc_2) in carry-save form shown in formula (4) and (5):

$$(ms_1, mc_1) = \begin{cases} (ps_1, pc_1) & \textit{if } \delta \geq 0, \\ (ps_2, pc_2) & \textit{others}, \end{cases} \tag{4}$$

$$(ms_2, mc_2) = \begin{cases} (ps_1, pc_1) & \textit{if } \delta < 0, \\ (ps_2, pc_2) & \textit{others}. \end{cases} \tag{5}$$

A exclusively or B is represented as A^B.

III. Basic Architecture

The basic architecture [2] [3] of dot product computation is based on the algorithm of traditional FP adder. The computing includes following steps:

1) Unpacking. Unpack the input FP values to triple representations including the sign, exponent, and significand
2) Significands' multiplication and exponent processing. The multiply array fed with *ma* and *mb* outputs (ps_1, pc_1), another array fed with *mc* and *md* outputs (ps_2, pc_2). The exponent difference δ is got by $\delta = (ea+eb) - (ec+ed)$. δ is represented by *sign_d* and *d* mentioned above. Meanwhile, the *emax* is produced by *emax*= maximum (*ea*+*eb*, *ec*+*ed*) − *bias*.
3) According to the *sign_d*, swap (ps_1, pc_1) and (ps_2, pc_2), swap's results are (ms_1, mc_1) and (ms_2, mc_2) described in formula (4) and (5) respectively. Perform the alignment shift to (ms_2, mc_2) by *d*. If the effective operation which is decided by input operation type and signs of FP numbers is subtraction, results of alignment shift will be inverted.
4) Compression. (ms_1, mc_1), and aligned (ms_2, mc_2) are compressed by the compression logic into data represented in carry-save form.
5) Internal addition and leading zero anticipation. The compound adder is employed to avoid the requirement of complement logic for negative results.
6) Normalization.
7) Performing Rounding operation according to the rounding type. Then packing and outputting the final result.

The basic fused architecture of 2-term dot product computation is shown in Fig. 2.

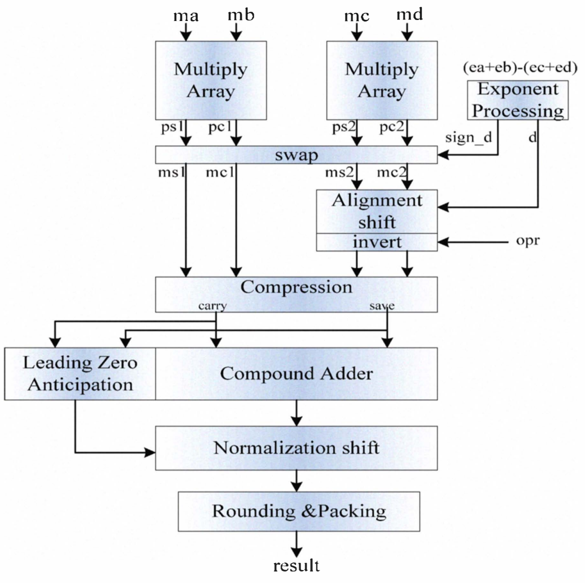

Figure 2. The basic scheme of FDP2

IV. Dual-Path Architecture

A. Overview

In this section, the dual-path architecture is presented to speedup the dot product computation. 2 terms dot product

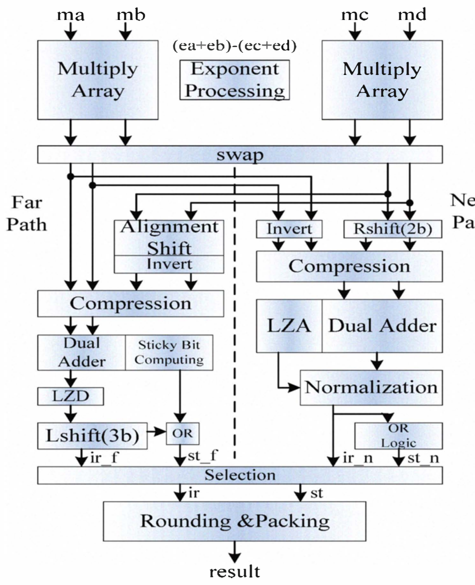

Figure 3. The dual-path scheme of FDP2.

computation is actually the addition of two products of multiply arrays. So the dual-path addition technique [5] which is effectively used in FP adder could also be employed in the dot product computation. However, in the first stage of computing, the multiply array is located in the critical path of computation, and then the exponent processing which is performed parallel with multiplication does not need to be divided into two paths. After the multiplication and swap of products by the *d_sign*, the computation path is divided into two parallel paths exclusively used for the different scenarios. Because there are two bits in the integer part of every product, then if $d \leq 2$, the data cancellation maybe happens. Additionally, the near path is the critical path because the near path contains a large internal adder than far path does shown in the following part, so we just deal with effective subtraction in the near path to simplify the computation in the critical path. The partition criterions are:

(1) Near path: if $d \leq 2$ and the operation is effective subtraction.

(2) Far path: the remaining cases.

In the near path, because the exponent difference is so small that the data cancellation maybe happens which leads to a lot of leading zero produced, then a large left shift is required for the purpose of normalization. But, because $d \leq 2$, only a 2 bits right shift is needed to perform alignment.

On the contrary, in the far path, there are only 3 bits of leading zero produced in the worst case because of the large exponent difference, so the normalization shift could finish quickly. However, the normalization in the far path requires a large right shifter.

Both the far path and near path could produce one sticky bit and f+1 bits of accurate intermediate results which contain one round bit. The two paths are combined before the rounding stage. The dual-path architecture of dot product computation is shown in Fig 3.

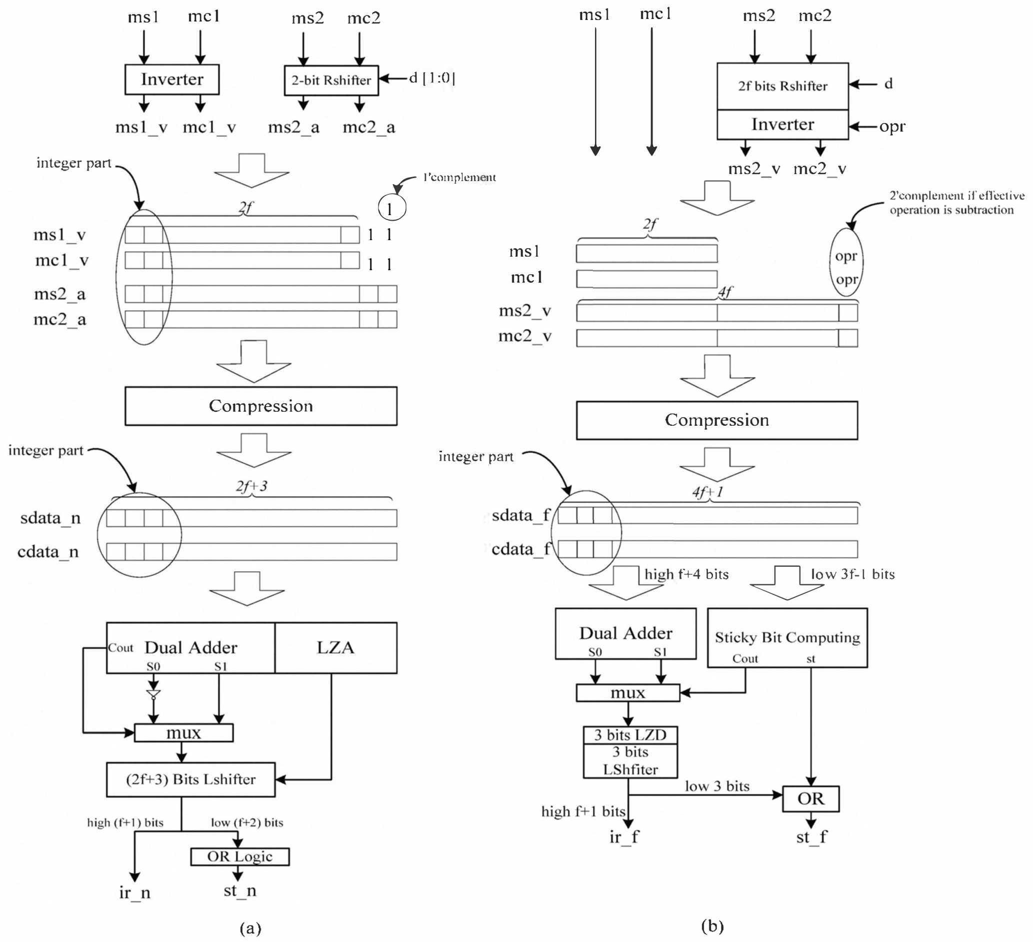

Figure 4. (a) Near Path (b) Far Path

B. Near Path

The detailed near path is given in Fig. 4 (a). There is only effective subtraction performed in the near path and the exponent difference is not greater than two. The computation in the near path includes following steps:

1) The smaller product represented by (ms_2, mc_2) is aligned by low two bits of d. At same time, the invert is performed to the bigger product (ms_1, mc_1). We denote aligned product (ms_2, mc_2) as(ms_2_a, mc_2_a) which width is $2f$+2 bits. The inverted product (ms_1, mc_1) is denoted as (ms_1_v, mc_1_v) which width are $2f$ bits.
2) For the purpose of avoiding complement logic which is used to deal with the negative results, the internal addition in the near path employs one's complement subtraction and the dual adder [10] which could produce both the addition's result $S0$ and $S0$+1(denoted as S1). Assume n=$2f$+2,

$$\begin{aligned} & ms_2\_a + mc_2\_a - ms_1 - mc_1 \\ & = ms_2\_a + mc_2\_a - ms_1 - mc_1 + 2^{n+1} - 2^{n+1} \\ & = ms_2\_a + mc_2\_a + (2^n - ms_1) + (2^n - mc_1) - 2^{n+1} \\ & = (ms_2\_a + mc_2\_a + ms_1\_v + mc_1\_v + 1) - (2^{n+1} - 1) \end{aligned} \quad (6)$$

Assume $ms_2\_a + mc_2\_a + ms_1\_v + mc_1\_v + 1 = \{Cout, S0\}$ and $S0\_v$ is the inverter of $S0$. Then,

$$ms_2\_a + mc_2\_a - ms_1 - mc_1 = \begin{cases} S0\_v & if\ Cout = 0, \\ S1 & others \end{cases} \quad (7)$$

So, after the internal adder, we could get the absolute value of subtraction.

Before the compression, we need add additional one to the tail of $ms2\_a$ for the one's complement subtraction. The outputs of compression logic ($sdata\_n$, $cdata\_n$) are $2f$+3 bits. The dual adder's carry bit $Cout$ is used to choose the correct result.

3) Perform the normalization by a $2f$+3 bits left shifter according to the output of leading zero anticipation (LZA). The high f+1 bits of normalized result are the intermediate result of near path (ir_n), and low f+2 bits are fed into OR logic to produce the sticky bit (st_n).

C. Far Path

In the far path, the effective subtraction if d>2 and the effective addition are performed. Because the exponent difference is large, there is always a positive result produced by internal adder. The detailed computation in far path shown in Fig.4 (b) includes following steps:

1) The smaller product (ms_2, mc_2) is aligned by a $2f$ bits right shifter and then inverted to (ms_2_v, mc_2_v) which width are $4f$ bits.
2) The 2'complement representation is used for the effective subtraction and the complement logic is not required because there is no negative result produced in the far path. The outputs of compression logic ($sdata\_f$, $cdata\_f$) are $4f$+1 bits.
3) The far path should output f+1 accurate bits in which f bits for the significand and one bit is round bit. Because there may be 3 leading zero bits in the far path, we just need perform f+4 bits addition. The rest bits of ($sdata\_f$, $cdata\_f$) are responsible for producing the sticky bit and carry bit fed to the high part's addition. This task is finished by the sticky bit computing logic. The detail of sticky bit computing logic will be introduced in section 4.4. Because there may be a carry bit from low $3f$-1 bits, the high f+4 bits' addition employs a dual adder. After the addition in the dual adder is over we choose the correct result by the carry bit from the sticky bit computing logic.
4) A 3 bits leading zero detect logic and a 3 bits left shifter are used to normalize the output of the internal adder in the far path. The low 3 bits of normalized result and the output of sticky bit computing logic st are fed into OR logic to produce the sticky bit in the far path(st_f). The high f+1 bits of normalized result is output as the intermediate result in the far path (ir_f).

D. Sticky Bit Computing

The sticky bit computing logic produces the carry bit and the sticky bit based on the pattern detection. The generation of carry bit could be achieved by parallel-prefix addition technology [10]. So in this section, we just focus on the generation of sticky bit.

Considering two inputs with magnitude representation: A and B as follows:

$A = a_{n-1}a_{n-2}...a_1a_0$, $B = b_{n-1}b_{n-2}...b_1b_0$,

and define functions:

$p_i = a_i \wedge b_i$, $y_i = a_ib_i$, $z_i = \overline{a_i + b_i}$.

When the result is positive, then sticky bit is zero if and only if:

(1) All bits of inputs are zero, namely A=B=0, or

(2) Inputs are not zero, but the addition of inputs produce a carry bit and the rest of result are zero, namely A+B=2^n.

If condition (1) is fulfilled, the pattern is $zzz...zzz$.

If condition (2) is fulfilled, the pattern is $p*yz*$, where '*' indicates zero or more instances.

A binary tree is used to match those patterns. A node of the binary tree has three possible value p, z, y. The matching rules are defined in (8):

$$\begin{cases} p = p^l p^r, \\ z = z^l z^r, \\ y = p^l y^r + y^l z^r, \end{cases} \quad (8)$$

where (z^l, p^l, y^l) and (z^r, p^r, y^r) represent the left input and the right input, respectively. Figure 5 show a example how the binary tree works to match the pattern $p*yz*$.

So, the sticky bit *st* is generated by the final node's values *y* and *z* in formula (9)

$$st = \overline{y+z} \tag{9}$$

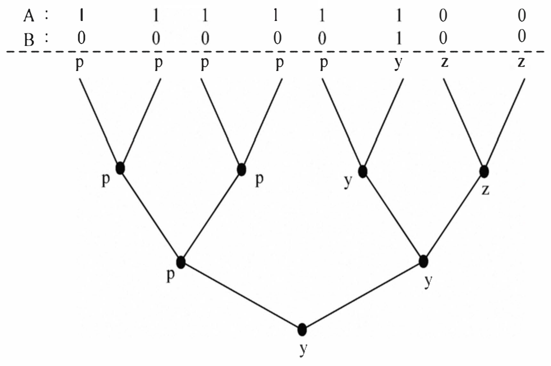

Figure 5. An example of pattern detection by a binary tree.

V. Implementation And Verification

A. Performance Analysis

An implementation of proposed dual path architecture of FP dot product computation has been finished with FP single-precision data format. For comparison purpose, the network architecture and the basic fused dot product unit are also implemented. All architectures are coded by Verilog HDL and synthesized by Design Compiler of Synopsys with a SIMC 180nm CMOS standard cell library. The implementation results are shown in Figure 6. Detailed data are shown in Table 1, where all the data are normalized by the data of network architecture.

Compare to the network method, the proposed architecture could speedup dot product computation 1.32 times with 37.5% area penalty. The proposed architecture is also faster 5.45% and has 1% less area than the basic one.

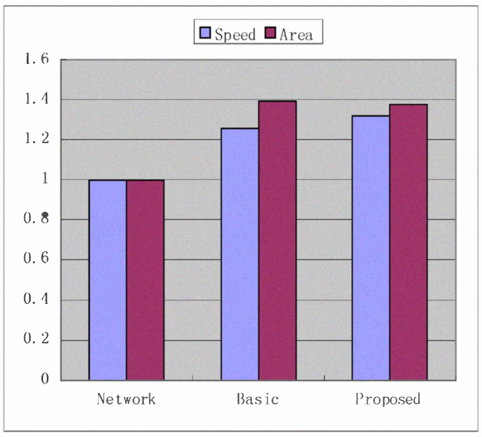

Figure 6. The speedup and area cost of every architecture.

B. Function Verification

We build a accumulator architecture which width is equal to $(2*2^{el}+1)$ to verify our design, where *el* is the bit width of exponent. Our architecture is tested by random data and corner cases.

TABLE I. Implementation Data

	network	*Basic*	*Proposed*
Area(μm^2)	304891.1	423597.09	419252.75
Delay(ns)	15.02	12.00	11.38

VI. Conclusion

Two-term FP dot product computation is one of most frequently used operation. In this paper dual-path architecture to perform $A\times B+C\times D$ is presented. Besides, this architecture could also execute multiplication, addition, and MAF operation. Because only one rounding operation is performed, there is less rounding errors is introduced compared with the network method. The implementation result shows that the proposed architecture is faster than both the network method and the basic fused architecture.

Acknowledge

This work is supported by grants from National Nature Science Foundation of China (No: 60773223, 61003037, 60736012) and National High Technology Research and Development Program of China (No: 2009AA01Z110).

Reference

[1] U.W. Kulisch and L.Miranker, "The Arithmetic of the Digital Computer: A New Approach", SIAM Review vol.28, no.1, March 1986.

[2] H. H. Saleh and E. E. Swartzlander, Jr. "A Floating-Point Fused Dot-Product Unit". IEEE International Conference on Computer Design (ICCD), Lake Tahoe, CA, 2008. pp. 427-431.

[3] Swartzlander E. Saleh, H. "FFT Implementation with Fused Floating-Point Operations". IEEE Transactions on computers. 2010. vol.PP(99), pp.1.

[4] Mustafa Gok, Metin Mete Ozbilen. "Multi-functional floating-point MAF designs with dot product support". Microelectronics Journal. January 2008. Vol. 39(1): pp. 30-43.

[5] Peter-Michael Seidel and Guy Even, "Delay-Optimized Implementation of IEEE Floating-Point Addition" IEEE Transactions on computers,Vol. 53, No. 2, February, 2004.

[6] P. Farmwald, "On the Design of High Performance Digital Arithmetic Units," PhD thesis, Stanford Univ., Aug. 1981.

[7] P.M. Seidel, "Multiple Path IEEE Floating-Point Fused Multiply-Add," Proceedings of the 46th IEEE International Midwest Symposium on Circuits and Systems, pp. 1359- 1362, 2003.

[8] Quinnell, E.; Swartzlander, E.E.; Lemonds, C.; "Floating-Point Fused Multiply-Add Architectures," Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on, pp.331-337, 4-7 Nov. 2007, doi: 10.1109/ACSSC.2007.4487224.

[9] IEEE Computer Society, "IEEE Standard for Floating-Point Arithmetic", IEEE Std. 754-2008, Aug. 2008.

[10] R. Zimmermann, "Binary Adder Architectures for Cell-Based VLSI and their Synthesis". PhD thesis, Swiss Federal Institute of Technology (ETH) Zurich, Hartung-Gorre Verlag, 1998.