Triple-mode floating-point adder architectures

LIU DE, WANG MINGJIANG
School of Electronic and Information Engineering
Harbin Institute of Technology Shenzhen Graduate School
Xili University Town, Shenzhen, Guangdong
CHINA
liude19832006@126.com http://www.hitsz.edu.cn

Abstract: This paper presents an architecture of a triple-mode floating-point adder that supports higher precision and parallel lower precision addition. The proposed design can work in three modes: four parallel single precision or two parallel double precision or one quadruple precision addition/subtraction. The proposed triple-mode adder’s parallel computation in lower precision can be applied in SIMD application to accommodate 3D graphics, video conferencing and multimedia fields while its high precision computation can be applied in scientific applications such as supernova simulations, climate modeling and etc. To improve the performance of the triple-mode floating-point adder, the design is implemented with the improved two-path algorithm in combinational and pipeline form. To compare area, power and worst-case latency, single-mode single, double, quadruple and dual-mode quadruple precision floating-point adders are also implemented using the similar techniques. These adders and the triple-mode adder are tested and verified through extensive simulation and then synthesized with 65nm manufacturing process. The synthesis results show that the proposed triple-mode floating-point adder requires 10-16% more delay than a single-mode quadruple precision adder and saves 47-52% area compared to the combination of four single, two double and one quadruple precision adders.

Key-Words: floating-point adder, floating-point arithmetic, triple-mode adder

1 Introduction

Floating-point arithmetic is a key part of CPU, GPU and DSP chips for its freedom from overflow and underflow and ease of using to programmers. Today many general purpose processors offer hardware video decoding, image processing and 3D functionality by executing SIMD instructions such as Intel’s AVX, SSE3 and SSE4 [1,2]. Most of these SIMD instructions are floating-point arithmetic related and directly executed through two or more parallel single precision floating-point units (FPU). For image processing and 3D video game, a large number of single precision floating-point operations are executed through many parallel FPs in graphic chips [3]. NVIDIA’s GTX680 with Kepler architecture has 1536 cores in a single die and AMD’s HD7970 with GCN architecture has 2048 cores and each core comprises 5 FPUs [4]. Besides the traditional applications of SIMD, [4] uses SIMD instructions of GPU to accelerate encryption; [5] uses GPU and SIMD instruction of CPU to accelerate Viterbi decoding in wireless communication; parallel SIMDs are even used in bioinformatics [6]. Juan M. Cebrian [7] points out in his research work that power efficiency of parallel FPUs is higher than that of multi-core and .

Single-precision FPU is useful for SIMD, but low precision makes it not able to support some scientific applications. Although higher precision arithmetic can be implemented by software, it is reported that hardware implementation of a quadruple precision FPU is approximately 200 times faster than that of software implementation [8]. For scientific applications, higher precision floating-point computation is needed, which is another trend. D.H.Bailey [9] and G. H.owell [10] describes the necessity that higher precision arithmetic is useful for a variety of situations including ill-conditioned linear systems, large scale simulation etc. Although 64-bit IEEE [11] arithmetic is sufficient for those situations [10], supernova simulation and climate modeling still need higher precision such as 128-bit floating-point arithmetic. E.Schwarz [12] presents in their paper that quadruple precision (128-bit) floating-point unit can be implemented in a reasonable amount of hardware compared to double precision.

In the past decade, the software and hardware has gradually transformed from 32-bit to 64-bit. Today, from personal devices such as PC, mobile phone to enterprise workstation, from software to hardware, 64-bit has become very universal, so making FPU supporting 64-bit is also necessary.
Since the most frequent operation—floating-point addition, takes 55% of all five basic arithmetic operations specified by IEEE754-2008 [11], much research and many papers have proposed efficient floating-point addition algorithms and architectures [14-25].

Currently, most floating-point addition (subtraction) units in modern microprocessors are implemented in two-path algorithm [14-15]. With the same manufacturing technology, adder implemented with the two-path algorithm is faster but consumes more area and power, compared to single-path algorithm. As the feature size of CMOS transistor continually shrinks, the transistor becomes faster and less power consuming, so the advantage of single-path algorithm in area and power over two-path algorithm is no more important.

As described above, to support SIMD, scientific and 64-bit applications, designing a triple-mode quadruple precision floating-point adder for general purpose processor is necessary.

Several literatures present dual-mode floating-point unit including adder, multiplier, divider and multiply-add fused (MAF). A.Akkas presents a dual-mode precision floating-point adder in [26-27], a dual-mode floating-point multiplier in [28-29] and a dual-mode floating-point divider in [30], [31] presents a dual-mode double precision floating-point adder with single-path algorithm. [32] presents a 80-bits multiplier which can perform one 80-bit multiplication in 5 cycles, or one 64-bit multiplication in 4 cycles or two parallel 32-bit multiplications in 2 cycles. Ray C.C. Cheung [33] designed a dual-mode floating-point multiplier. K. Manolopoulos [34] presents a triple-mode multiplier that can perform single, double and quadruple precision multiplication. Baluni [35] presents a fully pipelined dual-mode floating-point multiplier. Libo Huang [36] and K. Manolopoulos respectively present a dual-mode floating-point MAF unit that can perform one double precision multiply-add (MA) operation or two single precision MAs. [38] presents a multi-functional MAF that can perform one double precision MA, or two single precision dot products.

In this paper, we present a single-mode and a triple-mode quadruple precision floating-point adder. Our proposed designs support all four rounding modes and exceptions specified by IEEE754-2008, but does not support sub-normal number.

Section 2 describes the proposed delay-efficient architecture of a single-mode quadruple precision adder implemented with two-path algorithm. The architecture is described in pipeline form and the specific details of circuit implementation of each component is also presented. The synthesized results of its corresponding combinational circuit is presented in Section 4.

In Section 3, we design a triple-mode quadruple precision adder by modifying the architecture of the single-mode adder in Section 2. For comparison, we also implemented single-mode, double and dual-mode quadruple precision adders using the similar techniques. All the adders with both combinational and pipeline for MAF units are implemented in Verilog-HDL, and verified through extensive simulations.

In Section 4, the synthesized results of single-mode single, double, quadruple, dual-mode quadruple and triple-mode quadruple precision floating-point adders are presented and compared.

2 Single-Mode Quadruple Precision Floating-Point Adder

The proposed design has three pipeline stages and is described in section 2.1, 2.2 and 2.3 for each stage respectively.

2.1 Stage 1

Fig.1 shows the first stage of the proposed pipelined architecture of the quadruple precision floating-point adder. D1 and D2 are the two operands, Op is the initial operation and the 2-bit signal rm is the rounding control signal. The rounding modes are as follows: rm=0, round to nearest even; rm=1, round to positive infinity; rm=2, round to negative infinity; rm=3, round to zero.

The functionality of the first stage is to compare exponents of D1 and D2, swap mantissas of D1 and D2, determine the sign of the result and the effective operation eff_op in FA, and compute the mantissa difference in CLOSE path. When eff_op equals to 1, the effective operation is subtraction.

In FAR path, the EXP_DIFF block in this stage produces two signals: swap and exp_diff which is the absolute value of the difference of exp1 and exp2. OR1 and OR2 blocks are two OR gates to generate the hidden leading bit of mantissa. If the exponent equals to zero, the hidden leading bit hd_bit1 (or hd_bit2) is 0, otherwise is 1. The EXP_DIFF block is implemented using a flagged parallel prefix adder (FPPA1) which can compute \(|A-B|\). The EXP_DIFF block does not contain a comparator. The details of a FPPA1 is shown in [20]. When swap equals to 0, exp1 is less than exp2, the
greater mantissa \{hd_bit2, frac2\} is selected through MUX1 to obtain frac_large; the smaller mantissa \{hd_bit1, frac1\} is selected through MUX2 to obtain frac_small. When swap equals to 1, the opposite operation occurs as illustrated in Fig. 1. The greater exponent is multiplexed through EXP_MUX.

The exp_diff is a 15-bit number and adjusted to 7 bits through EXP_DIFF_ADJ block. The circuit of EXP_DIFF_ADJ block is shown in Fig. 2(a). If the high order bits exp_diff[14:7] is not 0, which means exp_diff is greater than 127, then align_num is 127, otherwise align_num is exp_diff[6:0]. In the process of alignment, frac_small can be right shifted at most 116 bits, so 7 bits is sufficient to hold a number that is greater than or equal to 116. Another objective of adjusting the exponent difference is to decrease the delay of ALIGN block. The ALIGN block accomplishes the task of right shifting frac_small by the align_num bits. The alignment shift block ALIGN is generally implemented using a barrel shifter which is composed of several levels of multiplexers. When the width of the shifting number is 15 bits, the delay of a barrel shifter is 15 times the delay of a multiplexer and the area is 15 times the area of one level of multiplexers. That is why we need to adjust the 15-bit exp_diff to 7-bits align_num. The SIGN_FAR block is used to produce the sign signal sign_f of FAR path and its circuit is shown in Fig. 2(b). The NaN&Inf_DETECT block receives eff_op, exponents and mantissas of the two
operands to determine whether the result is infinity or a NaN. To share logic, the \(hd\_bit1\) and \(hd\_bit2\) are passed to this block. When \(D1\) and \(D2\) are both infinity and the effective operation is subtraction \((eff\_op=1)\), the result is a NaN \((res\_is\_nan=1)\); when one of the two operands is NaN, the result is a NaN \((res\_is\_nan=1)\); when one operand is infinity and the other is a normal number, the result is infinity \((res\_is\_inf=1)\). To make the Figure clear, we use a 2-bit signal \(except = \{res\_is\_nan, res\_is\_inf\}\) to represent exceptions as illustrated in Fig. 1.

In \textit{CLOSE} path:
1) When \(exp1=exp2, exp1[0]\) definitely equals to \(exp2[0]\), then \(ena1\) and \(ena2\) is 0, both mantissas \(\{1,frac1\}\) and \(\{1,frac2\}\) remain unchanged through \(MUX3\) and \(MUX4\).
2) When \(\{frac1[111], frac2[111]\}\) is 01 and \(exp1\neq exp2, ena2\) is 1, \(ena1\) is 0, \(\{1,frac1\}\) remains unchanged through \(MUX3\), \(\{1,frac2\}\) is right shifted by one bit through \(MUX4\), the difference \(S_c\) (or \(SP1\_c\)) of the two mantissas has more than two leading zeros: (A) if \(exp1-exp2=1, S_c\) is the right result, so \(path\_sel\) turns into 1; (B) if \(exp1-exp2=-1\), the right result should be selected from \(FAR\) path, \(path\_sel\) turns into 0; in this case, \(frac\_large=\{1,frac2\}, frac\_small=\{1,frac1\}\), after aligning \(frac\_small\), the difference of \(frac\_large\) and \(frac\_align\) has at most two leading zeros.
3) If \(\{frac1[111], frac2[111]\}\) is 10, the mechanism is just the opposite in step 2).

The \textit{PATH\_GEN} block is shown in Fig. 2(c). When the exponents difference \(align\_num\) is 0 or 1 and \(eff\_op\) is 1 and \(\{swap, frac1[111], frac2[111]\}\) is 101 or 010, \(path\_sel\) is 1. The reason is explained in step 1, 2) and 3) above.
In our proposed design, we used two-way leading zero detection (LZD) to count the number of leading zeros. The LZD functionality is implemented through a leading zero anticipating (LZA) and a leading zero counting (LZC) logic. To obtain the difference of two mantissas, a 113-bit flagged parallel prefix adder (FPPA1) is used. As illustrated in Fig.1, the shifted and inverted mantissa \( \text{mant2} \) and \( \text{mant1} \) are passed to \( \text{LZA}_\text{POS} \) and \( \text{ADDER}_\text{CLOSE} \) blocks, and the shifted and inverted mantissa \( \text{mant1} \) and \( \text{mant2} \) are passed to \( \text{LZA}_\text{NEG} \) block. In the case that \( \text{mant1} \) is less than \( \text{mant2} \), the leading zero number \( \text{norm}_\text{pos} \) detected by \( \text{LZA}_\text{POS} \) and \( \text{LZC}_\text{POS} \) is false, but \( \text{norm}_\text{neg} \) is true. In the later situation, the signal \( \text{cout}_\text{c} \) is 0, so the correct leading zero number \( \text{norm}_\text{num} \) can always be obtained through \( \text{MUX}_5 \). The LZC circuits are described in detail in [22, 24-25] and [21], but we choose [21]’s method for LZC logic since it is easier to be modified to implement multimode leading zero detection. We choose [23]’s method for LZA logic since it is faster and less area consuming compared to [22, 24-25]. If using one-way LZA-LZC, in the case of \( \text{exp}_\text{diff} \) equaling to 0, we have to use a 113-bit comparator to compare the two mantissas before LZA, which would introduce a large amount of delay and area. The correct leading zero number of our used method is always one less than or equal to the exact result. To correct the LZA error, the mantissas difference after normalization needs to be left shifted by one or zero bit. Paper [39] proposed LZA circuit which can obtain the exact leading zero number and need no LZA error correction, but have 25% more delay and 67% more area than its parallel adder in 128-bit. Paper [40] proposed a LZA error correcting circuit that consumes less power and an area, but the delay increases as the bit width grows. The logic level of LZA error correcting circuit in [40] is \( \text{log}_2 n + 8 \), which is far greater than the logic level of the adder tree \( (\text{log}_2 n + 2) \) in CLOSE path. Paper [25] proposed a faster LZA correction circuit, but its area is about two times of one LZD (LZA+LZC). So the area of one LZD in [25] exceeds the area of two-way LZD. Compared to [25, 39-40], our proposed two-way LZD can obtain the best trade-off in delay and area.

The \( \text{ADDER}_\text{CLOSE} \) block is implemented using a FPPA1 which computes \( A + B (A-B-1), A + B + 1 (A-B), \) and \( B - A \). The detail of a FPPA1 is shown in [20]. When \( \text{cout}_\text{c} = 1 \), the output sum \( S_c \) of CLOSE path is \( A + B \) and the \( SP_1 \) equals to \( S_c + 1 (A + B + 1) \); when \( \text{cout}_\text{c} = 0 \), \( S_c \) is equal to \( SP_1 (B-A) \). A and B are the two inputs of FPPA1.

### 2.2 Stage 2

The functionality of the second stage is to compute the sum/difference of mantissas as of FAR path, generate the guard, round and sticky bit of FAR path, and normalize the difference of mantissas of CLSOE path. Fig. 3 shows the details of the second stage.

The \( \text{ADDER}_\text{FAR} \) block is used to compute the sum/difference. When the effective operation is addition and the rounding mode is rounding toward positive or negative infinity, \( A + B + 2 \) also need to be computed [18]. We designed a new flagged parallel prefix adder (FPPA2) that is different to the one presented in [20]. Our designed FPPA2 can compute \( A + B, A + B + 1, A + B + 2, A + B, A + B + 1, A + B + 2 \) and \( B - A \). For two operands \( A = \{a_n, k, a_0\} \) and \( B = \{b_n, k, b_0\} \), the flag signal \( \text{flag} \) for obtaining \( A + B + 2 \) is produced from the two signals \( G = \{g_n, k, g_0\} \) and \( P = \{p_{n-1}, k, p_{b_0}\} \). The sign \( \text{sign}_f \) is obtained using the following formula:

\[
S + 2 = S + \text{flag}_f (s_f) \]

The “+” means AND operation and “\(^{\land}\)” means XOR operation. The flag signal \( \text{flag}_f \) for obtaining \( A + B + 1 \) is the same as described in [20] and \( S + 1 = S + \text{flag}_f \). To the author’s knowledge, the FPPA2 is the first time designed and used in floating-point adder. The literature [14-18, 26-27, 31] all used a compound adder to implement \( S + 2 \). Compared to compound adder, our designed FPPA2 saves the extra row of halfadders, which in turn decreases the delay of the critical path. The \( S, S + 1 \) and \( S + 2 \) of FAR path is denoted as \( S_f, SP_1_f, SP_2_f \). See Fig. 3.

The \( \text{GRS}_\text{LOGIC} \) block is used to generate the guard, round and sticky bit of the final result and its circuit is shown in Fig. 2(d). The \( \text{RM}_\text{DEC} \) block is used to decode the rounding mode. When \( \text{rm} \) equals to 0, the signal \( rd \_near \) indicating round toward nearest even is activated; when rounding mode is round toward positive infinity \( rm = 1 \) and the sign is positive \( \text{sign}_c = 0 \) or \( \text{sign}_f = 0 \), or round toward negative infinity \( rm = 2 \) and the sign is negative \( \text{sign}_c = 1 \) or \( \text{sign}_f = 1 \), the signal \( up \_f \) indicating rounding toward infinity is activated. The circuit of \( \text{RM}_\text{DEC} \) is shown in Fig. 2(e).

Since the result \( S + 1 (SP_1_c) \) and the result \( S (S_c) \) of CLOSE path have been computed, the rounding
process is executed through \textit{ROUND\_CLOSE} block before normalization. \cite{18} presents the rounding mode. The rounded result is a 114-bit signal and denoted as \textit{frac\_round\_c}. The circuit of \textit{ROUND\_CLOSE} block is shown in Fig. 2(f). Since the mantissa of CLOSE path is right shifted by one or zero bit through \textit{MUX3} and \textit{MUX4}, the round and sticky bit is 0, the guard bit is just the LSB of mantissa \{1,frac1\} (\{1,frac2\}) or 0. \textit{g\_c} is the guard bit and is not drawn in stage1 for clarity.

The \textit{NORMALIZATION} block is used to normalize the rounded result of CLOSE path and implemented with a traditional barrel shifter. As mentioned earlier, the normalized result need an extra 1 bit left shifting to correct the LZA error. As illustrated in Fig. 3, if the MSB of \textit{frac\_norm—MSB\_c} is 0, \textit{frac\_norm} is left shifted by one bit through \textit{MUX6}, otherwise the mantissa remains unchanged.

2.3 Stage 3

In the third stage, the exponents of both paths are adjusted, the mantissa sum of FAR path is rounded and exception is determined. Fig. 4 shows the details of the third stage.

The rounding of mantissa sum of FAR path is completed through the \textit{ROUND\_FAR} block. As we stated in stage1 step 2), when the effective operation is subtraction and the two high bits of the rounded result is 00, the rounded result need to be left shifted by two bits. This leads to a different rounding mode compared to general rounding mode \cite{18} in FAR path. For example, When \textit{grs}=011, no carry can be propagated to the \textit{LSB} of \textit{S\_f} because \textit{g} equals to 0, the high order bits is selected as \textit{S}. In this situation, if \textit{MSB}=1, the result \textit{S\_f} needs no left shifting, there is no carry propagated to \textit{LSB} of \textit{S}; if \textit{MSB}=0, \textit{s\_MSB}=1, the result \textit{S\_f} needs left shifting by one bit, after left shifting, \textit{g} become the new least significant bit of the result, and cause \textit{rs}=11, there is a carry propagated to \textit{g} which turns \textit{g} from 0 to 1; if \textit{MSB}=0, \textit{s\_MSB}=0, the result \textit{S} needs left shifting by two bit, \textit{g} and \textit{r} are shifted into the result, and cause \textit{s}=1 and \textit{r}=1, there is a carry propagated to \textit{g} which turns \textit{g} from 0 to 1 and \textit{r} from 1 to 0. When \textit{up\_f} equals to 0 or rounding toward zero (\textit{rd\_zero}) is active, no rounding is needed and the two bits shifted in keep the value of \textit{r} and \textit{s}. The rounding mode is summarized in Table 1 and 2, and the circuit of \textit{ROUND\_FAR} block is shown in Fig. 5(a). In Table 1, \textit{LSB} is the least significant bit of \textit{S\_f}, \textit{MSB} is the most significant bit of \textit{S\_f}, \textit{s\_MSB} is the bit right next to \textit{MSB}, \textit{C} is the carry out of FAR path—\textit{cout\_f},
SP1 is SP1_f, SP2 is SP2_f. The rounded result is left shifted twice depending on the value of MSB and sMSB, through MUX7 and MUX8. The reason why frac_f1 need left shifting at most two bits is explained previously in case (B) of step 2).

The EXP_ADJ_FAR block is a simple common adder to increment or decrement exp_large. The functionality of ADDEND block is to determine the addend which is added to exp_large. When both MSB and sMSB are 0 and eff_op is 1, the addend is assigned to 111111111111111 (-2); when MSB is 0 and sMSB is 1 and eff_op is 1, the addend is assigned to 111111111111111 (-1); when MSB is 1 and eff_op is 1, the addend is assigned to 000000000000000; when cout_f is 1 and eff_op is 0, the addend is assigned to 00 0000000000001 (+1); when cout_f is 0, MSB is 1 and eff_op is 0, the addend is assigned to 000000000000000. The circuit of ADDEND block is shown in Fig. 5(d). If the adjusted exponent exp_f_tmp is the maximum value (111111111111111), exp_inf turns into 1, overflow occurs.

The sign, exponent and fraction of the result is selected through MUX10, MUX11 and MUX12 according to path selection signal path_sel.

The EXACT and EXCEPTION blocks are used to detect exceptions. When the bits rounded off is 0, the result is exact as shown in Fig. 5(b). When one of the input operand is a NaN (except=10), the result is invalid. When the result is infinity (except=01) or the exponent reaches its maximum value (exp_inf=1), overflow occurs. When no overflow, underflow or invalid occurs, inexact turns into the complement of exact_tmp. The circuit of EXCEPTIONS block is shown in Fig. 5(e). Since
res_is_nan and res_is_inf are mutually exclusive and except={res_is_nan, res_is_inf}, except could not be 11.

The final exponent and fraction are selected through MXU13 and MUX14 in Fig. 4.

### 3 Triple-Mode Quadruple Precision Floating-Point Adder

In this section, a triple-mode quadruple precision floating-point adder is designed with the architecture of the improved two-path algorithm.

#### 3.1 Stage 1

Pipeline stage 1 is shown in Fig. 6. For clarity the rounding mode (rm), and precision mode (op_mode) signals are not drawn in Fig. 6.

---

In Fig. 6, S1, S2, S3, S4, S5, S6, S7, S8 are all 32-bit floating-point numbers. When op_mode equals to 0, the adder operates in quadruple precision mode, Q1 consists of S1, S2, S3 and S4, and Q2 consists of S5, S6, S7 and S8 as illustrated in Fig. 6. When op_mode equals to 1, the adder operates in double precision mode, D1 consists of S1, S2, D2 consists of S3, S4, D3 consists of S5, S6, and D4 consists of S7, S8. D1, D2, D3 and D4 are all 64-bit floating-point numbers. In other cases, the adder operates in single precision mode (op_mode equals to 2 or 3). The combination of various precision modes and operations is listed in Table 3.

The signs, exponents and fractions of each operands in various precision modes is shown in Fig. 6. For example, the sign of Q1 is S1[30:16], denoted as exp_q1; the fraction of Q1 is \{ S1[15:0], S2[31:0], S3[31:0], S4[31:0] \}, denoted as frac_q1. In this paper, without

---

#### Table 1. The rounding mode with effective subtraction operation

<table>
<thead>
<tr>
<th>R</th>
<th>rd_near</th>
<th>up_f</th>
<th>zero</th>
<th>rd_near</th>
<th>up_f</th>
<th>zero</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LSB=0</td>
<td>LSB=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>0</td>
<td>--</td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

#### Table 2. The rounding mode with effective addition operation

<table>
<thead>
<tr>
<th>R</th>
<th>rd_near</th>
<th>up_f</th>
<th>zero</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSB=1</td>
<td>sMSB=1</td>
<td>MSB=0</td>
</tr>
<tr>
<td>000</td>
<td>SP1,00</td>
<td>SP1,00</td>
<td>SP1,00</td>
</tr>
<tr>
<td>111</td>
<td>S,10</td>
<td>S,10</td>
<td>S,10</td>
</tr>
<tr>
<td>110</td>
<td>S,10(*)</td>
<td>S,10(*)</td>
<td>S,10(*)</td>
</tr>
<tr>
<td>101</td>
<td>S,01</td>
<td>S,01</td>
<td>S,01</td>
</tr>
<tr>
<td>001</td>
<td>S,00</td>
<td>S,00</td>
<td>S,00</td>
</tr>
</tbody>
</table>
special specification, "{}" means string concatenation.

The **EXP_DIFF** block is used to compute the exponent differences of operands and obtain the larger exponents in various precision modes. It consists of four FPPA1s as illustrated in Fig. 7(a). \(\text{exp}_d1\) is extended to \{0000, \text{exp}_d1\} and \(\text{exp}_d3\) is extended to \{0000, \text{exp}_d3\}. \(\text{exp}_s1\) is extended to \{0000000, \text{exp}_s1\} and \(\text{exp}_s5\) is extended to \{0000000, \text{exp}_s5\}. In a similar way, exponents of the second pair of single precision operands are extended to 11 bits by placing three 0s in their high order bits. \(\text{MUX}_1, \text{MUX}_2, \text{MUX}_3\) and \(\text{MUX}_4\) are used to select the correct exponents based on the corresponding operation mode. The \(\text{Exp}_\text{Diff}_j\) \((j=1,2,3,4)\) is similar to the one used in single-mode adder previously described, and produces the exponent difference signal \(\text{edi} (i=1,2,3,4)\) and the

---

**Fig. 5.** The circuits of components in the third stage
The exponent difference \( \text{exp}_{i} \) \((i=1,2,3,4)\) of each pair of operands are modified to appropriate bit width and the reason is explained in Section 2.1. The \text{EXP_DIFF_ADJ} is consist of four exponent difference adjust logics shown in Fig. 2(a). The \text{SIGN_LOGIC}, \text{PATH_LOGIC}, \text{EFF_OP_LOGIC} blocks are respectively consist of four \text{SIGN_FARs}, four \text{PATH_GENs} and four XOR gates illustrated in Fig. 2(b) and (c).

The \text{HIDDEN_BITS} block in Fig. 6 is used to determine the hidden leading bit for each mantissa.
The functionality of MANT_SWAP in Fig. 6 is to swap the mantissas of each pair of operands in different precision modes. This unit is easily implemented by an amount of multiplexors. When \( \text{swap}[i] \) equals to 0, the second mantissa of each pair of operands is selected. The circuit of MANT_SWAP block is shown in Fig. 7(c). The data structure of the swapped mantissas \( \text{frac}_\text{large} \) and \( \text{frac}_\text{small} \) are shown in Fig. 8. The benefit of this structure is that the carry out bit from the addition of lower order bits will not be propagated to higher order bits, because of the zeros between each mantissas.

The NaN&INF block is similar to the one described in section 2.

Table 3. Combination of various precision modes and operations

<table>
<thead>
<tr>
<th>( op[i] )</th>
<th>quadruple</th>
<th>double</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( Q1+Q2 )</td>
<td>( D1+D3 )</td>
<td>( S1+S5 )</td>
</tr>
<tr>
<td>1</td>
<td>( Q1-Q2 )</td>
<td>( D1-D3 )</td>
<td>( S1-S5 )</td>
</tr>
<tr>
<td>0</td>
<td>--</td>
<td>( D2+D4 )</td>
<td>( S2+S6 )</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
<td>( D2-D4 )</td>
<td>( S2-S6 )</td>
</tr>
<tr>
<td>0</td>
<td>--</td>
<td>( -- )</td>
<td>( S3+S7 )</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
<td>( -- )</td>
<td>( S3-S7 )</td>
</tr>
<tr>
<td>0</td>
<td>--</td>
<td>( -- )</td>
<td>( S4+S8 )</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
<td>( -- )</td>
<td>( S4-S8 )</td>
</tr>
</tbody>
</table>
described in single-mode adder. It produces four exception signals for each precision mode: except[i]={res_is_nan[i],res_is_inf[i]} (i=1,2,3,4).

The MANT_SHIFT_MUX block in Fig. 6 is used to right shift the mantissas by one or zero bit in various precision modes for CLOSE path. This block consists of seven MS blocks shown in Fig. 9. The techniques of MS block is similar to the one described in Section 2, and redrawn in the right lower corner in Fig. 9 and W is the bit width of mantissa. The data structure of the shifted mantissa frac1 and frac2 is shown in Fig. 8.

In Fig. 6, for CLOSE path, the shifted mantissa frac1 and frac2 are passed to MANT_ADD&LZD block. The functionality of this block is to compute the difference and leading zero number of each pair of mantissas in various precision modes. The implementation of MANT_ADD&LZD block is shown in Fig. 10(a). The ADDER_CLOSE block is a 113-bit FPPA1 and used to compute the difference of mantissas in CLSOE path. The flag signal generating SP1_c and carry out signals are slightly different from the one used in single-mode adder, which depends on the precision mode as following:

\[
\text{quadruple:} \quad \text{flag1}[0]=1, \text{flag1}[i]=A[i-1]^B[i-1] \text{ flag1}[i-1];
\]
cout\_c[1] = \text{Count} (\text{Cout is the carry out of } A+B)

double:
\begin{align*}
\text{flag1}[0] &= 1, \text{flag1}[56] = 1, \\
\text{flag1}[i] &= A[i-1]^B[i-1] \text{flag1}[i-1] (i \neq 0, 56) \\
\text{cout\_c[1]} &= S\_c[109], \text{cout\_c[2]} = S\_c[53]
\end{align*}

single:
\begin{align*}
\text{flag1}[0] &= 1, \text{flag1}[28] = 1, \text{flag1}[56] = 1, \text{flag1}[84] = 1, \\
\text{flag1}[i] &= A[i-1]^B[i-1] \text{flag1}[i-1] (i \neq 0, 28, 56, 84) \\
\text{cout\_c[1]} &= S\_c[107], \text{cout\_c[2]} = S\_c[80], \\
\text{cout\_c[3]} &= S\_c[52], \text{cout\_c[4]} = S\_c[24]
\end{align*}

The \( S\_c = frac1 + frac2, \ SP1\_c = S\_c^*\text{flag1} \).

As described in Section 2, the LZA\_LZC\_NEG is used to obtain the correct number of leading zeros in case that \( frac1 \) is smaller than \( frac2 \) in various precision modes. In Fig. 10(a), when the first single precision result is negative (\( cout\_c[1] \) equals to 0), \( ns1 \) of LZA\_LZC\_NEG is selected through MUX1 and MUX5; when the second double precision result is positive (\( cout\_c[2] \) equals to 1), \( nd2 \) of LZA\_LZC\_POS is selected through MUX3 and MUX6; when the quadruple precision result is positive, \( nq \) of LZA\_LZC\_POS is selected through MUX2 and MUX5. The signals \( \text{norm\_numi} \) \((i=1,2,3,4) \) are the leading zero numbers detected and used in the normalization in stage2 in various precision modes.

To support three operating modes, the LZA\_LZC\_POS(NEG) logic in Fig. 10(a) is slightly different from the one described in Section 2. The details of LZA, LZC and LZD logic are described in [23, 21], so here we only give the modification of these logics. Fig. 10(b) shows the implementation of LZA-LZC used in our designs. As shown in Fig. 10(b), the 113-bit LZA unit consists of four LZAs and three 1-bit multiplexers. MUX1, MUX2 and MUX3 are used to connect the four LZAs as a single 113-bit LZA in quadruple precision mode, and the outputs of four LZAs are concatenated and extended with 15 trailing zeros in double precision mode, the LZA29 is connected to the first LZA28 by MUX1 to anticipate the number of leading zeros of the first 53-bit mantissa, the second and third LZA28 together with MUX3 are used for the second 53-bit mantissa. The low order 25 bits of \( lza1 \) (\( lza[24:0] \)) and \( lza2 \) are concatenate and extended with 11 trailing zeros, and the same as \( lza3 \) and \( lza4 \). In single precision mode, each LZA block is used to anticipate the number of leading zeros of its corresponding mantissa. The low order 24 bits of \( lza1 \) (\( lza[1:23] \)) and \( lza2 \) is extended with 8 trailing zeros. Each LZC32 unit produces a 5-bit number from its corresponding leading zero anticipating string, two LZC32s plus one LZD Logic unit constitutes a 64-bit LZC unit and produces a 6-bit number. Finally two 64-bit LZC unit constitutes a...
128-bit LZC unit and produces a 7-bit number. As shown in Fig. 10(b), $nq$ represents the leading zero number in quadruple precision mode in the third pipeline stage, $nd1$ and $nd2$ in double precision mode, and $ns1$, $ns2$, $ns3$, $ns4$ in single precision mode.

In FAR path, the block $ALIGN$ in Fig. 6 is used to complete the alignment task in various precision modes. To support single, double and quadruple precision mode, the $ALIGN$ block is different from a traditional barrel shifter. The $ALIGN$ block receives $frac\_small$ and the adjusted exponent differences $shift1$, $shift2$, $shift3$, $shift4$ as its input operands and produces a shifted 113-bit number $frac\_align$ and three 4-bit signals: $g\_far$, $r\_far$ and $s\_far$ as shown in Fig. 6. The architecture of the $ALIGN$ block we proposed is illustrated in Fig. 11. If the operation is quadruple precision mode ($op\_mode$ equals to 0), the first column of multiplexers ($MUX1$ to $MUX19$) is used to right shift the high order 57 bits of $q$, the second column of multiplexers ($MUX5$ to $MUX20$) is used to right shift the low order 56 bits of $q$, and the third column of multiplexers ($MUX6$ to $MUX21$)
plus MUX3 are used to concatenate the low and high order bits. In double precision mode (op_mode equals to 1), 0 is passed to g2 through MUX2 and MUX3, then the first column and the second column of multiplexers works independently to right shift two 53-bits numbers. In single precision mode (op_mode equals to 2 or 3), shift3 is multiplexed through MUX22 and shift4 is multiplexed through MUX23, and the fourth single mantissa is passed to d7 through MUX2, MUX3 and MUX6. The first single precision mantissa is contained in the high order 29 bits of d1 and the third single precision mantissa is contained in the high order 28 bits of d6. The first, second and third column of multiplexers work independently to accomplish the right shifting of the first, third and fourth single precision mantissa. Since there are only three columns of multiplexers, we use another traditional barrel shifter with the width of 24 bits to right shift the second single precision mantissa as illustrated in Fig. 11. The logics of producing guard, rounding and sticky bits for various operating modes are simple and not drawn in Fig. 11 for simplicity.

3.2 Stage 2

The architecture of stage 2 is shown in Fig. 12. In pipeline stage 2, the aligned mantissa s frac_align and frac_large in FAR path are passed through ADDER_FAR block. Also the guard (g), round (r) and sticky (s) bits of mantissa result are determined in this stage. GRSi and RMi (i=1,2,3,4) blocks are similar to the GRS_LOGIC block in Fig. 2(d) and the RM_DEC block in Fig. 2(e). RMi blocks are used to obtain the rounding enabling signal rd_near, up_f and up_c. In FAR path, if the effective operation is subtraction (eff_op[i]=1), each part of frac Align is complemented through the XORi (i=1,2,3,4,5,6,7) gate and multiplexed through MUXi in various precision modes, and then added to frac_large through ADDER_FAR block. The ADDER_FAR block is a FPPA2 but slightly different from the one described in Section 2. The flag signal flag1 generating SP1_f and carry out signals cout_f are similar to that described in Stage 1 in this Section. The flag signal flag2 generating SP2_f is as following:

quadruple:
flag2[0]=0, flag2[1]=1,
flag2[i]=P[i-1]^G[i-2]*flag2[i-1] (i>1);
double:
flag2[0]=0, flag2[1]=1,

flag2[56]=0, flag2[57]=1,
flag2[i]=P[i-1]^G[i-2]*flag2[i-1] (i≠0,1,56,57);
single:
flag2[0]=0, flag2[1]=1,
flag2[28]=0, flag2[29]=1,
flag2[56]=0, flag2[57]=1,
flag2[84]=0, flag2[85]=1,
flag2[i]=P[i-1]^G[i-2]*flag2[i-1]
(i≠0,1,28,29,56,57,84,85);

S_f=frac_large+frac_tmp, SP1_f=S_f^flag1, SP2_f=S_f^flag2.

The mantissa results S_c and SP1_c of CLOSE path in Stage 1 are rounded through R_Ci (i=1,2,3,4,5,6,7) blocks and multiplexed through MUX2, then normalized through the NORMALIZATION block. The R_Ci logic is identical to the rounding logic in Fig. 2(f). The NORMALIZATION block is similar to the ALIGN block we described in Fig. 11 except that its shifting direction is left. In Fig. 12, the multiplexors from MUX3 to MUX9 are used to correct the LZA error. The mechanism of LZA error correction in various precision modes is exactly the same as that described in single-mode adder. Then according to precision mode, the results and MSBs are extended and multiplexed through MUX10 and MUX11.

3.3 Stage 3

The third pipeline stage of the triple-mode quadruple precision floating-point adder is shown in Fig. 13 and Fig. 14. In stage 3, the computing results of mantissas of FAR path are rounded, the exponents of both FAR and CLOSE path are adjusted, and the exceptions are detected. In Fig. 13, R_F1 block is used to round the mantissa result of quadruple precision operands, MUX1 and MUX8 are used to left shift the rounded result by at most 2 bits. R_F2, MUX3, MUX10 and R_F3, MUX4 and MUX11 are used for double precision mode. Similarly, from R_F4 to R_F7, and MUX4 to MUX14 are for single precision mode. The implementation of each rounding logic R_{Fi} (i=1,2,3,4,5,6,7) is totally identical to the logic in Fig. 5(a). The details of rounding mechanism is presented in Section 2. The seven rounded mantissa results are extended and multiplexed through MUX15 according to precision mode. The 113-bit signal frac_f in Fig. 13 is the final mantissa computing result of FAR path. The input signals of R_F block are shown at the top of Fig. 13.

The four EXACTi (i=1,2,3,4) blocks and the four EXCEPTIONi blocks are identical to the ones in Fig.
Fig. 12. The second pipeline stage of the triple-mode quadruple precision floating-point adder.

5(b)(c) and used to produce the invalid, inexact and overflow signals. The details of mechanism is described in Section 2 stage 3. The signal $expj_{inf}$ ($j=1,2,3,4$) is generated in the process of adjusting exponents of FAR path shown in Fig. 14.

In Fig. 14, like the method used in single-mode adder, four groups of exponents adjusting logic ($ADDENDi$, $ADDERi$) are used to adjust the exponents of FAR path. The seven AND gates are used to detect whether the mantissa result is zero or not. For example in quadruple mode, when $frac_c$ is not 0 and $exp\_cout$ is 0 which means the adjusted exponent $t$ is less than or equal to 0, since our design does not support subnormal number, underflow occurs and the signal $expj_{inf}$ ($j=1,2,3,4$) turns into 1. For example, in double precision mode, if $exp\_f$ is $xxxx1111111111$, $exp\_inf$ is asserted through $MUX20$. The $ADDENi$ block is completely identical to the one in Fig. 5(d).

For CLOSE path in Fig. 14, four $FPPA1s$ are used to subtract $norm\_uni$ from $exp\_large$ ($i=1,2,3,4$) to obtain the adjusted exponents. The seven OR gates are used to detect whether the mantissa result is zero or not. For example in quadruple mode, when $frac_c$ is not 0 and $exp\_cout$ is 0 which means the adjusted exponent $t$ is less than or equal to 0, since our design does not support subnormal number, underflow occurs and the signal $expj_{inf}$ ($j=1,2,3,4$) turns into 1. For example, in double precision mode, if $exp\_f$ is $xxxx1111111111$, $exp\_inf$ is asserted through $MUX20$. The $ADDENi$ block is completely identical to the one in Fig. 5(d).

For CLOSE path in Fig. 14, four $FPPA1s$ are used to subtract $norm\_uni$ from $exp\_large$ ($i=1,2,3,4$) to obtain the adjusted exponents. The seven OR gates are used to detect whether the mantissa result is zero or not. For example in quadruple mode, when $frac_c$ is not 0 and $exp\_cout$ is 0 which means the adjusted exponent $t$ is less than or equal to 0, since our design does not support subnormal number, underflow occurs and the signal $expj_{inf}$ ($j=1,2,3,4$) turns into 1. For example, in double precision mode, if $exp\_f$ is $xxxx1111111111$, $exp\_inf$ is asserted through $MUX20$. The $ADDENi$ block is completely identical to the one in Fig. 5(d).
According to the path selection signal $path\_sel$, the correct exponents are selected through MUX28 ($29,30,31$), and the correct mantissas are selected through MUX32 ($33,34,35$) in various precision modes. Then the exponents and mantissas are multiplexed through MUX36 ($37,38,39,40,41,42,43$). These eight multiplexors are used to process exceptions. For example in double precision mode, $except[1]=except[2]$: if $except[1]=\{res\_is\_nan[1],res\_is\_inf[1]\}=01$, then $15[1]$ is passed to $exp1$, $29[0]$ is passed to $frac1$ and $28[0]$ is passed to $frac2$. This means the computing result of the first pair of double precision operands ($D1$ and $D3$) is an infinity. At last, the

Fig. 13. Rounding of FAR path and exception detection in Stage 3
4 Synthesis Results

To make a complete comparison, the triple-mode quadruple, dual-mode quadruple, single-mode quadruple, double and single precision floating-point adders are implemented in Verilog-HDL using our proposed architecture. All the adders are in two forms: combinational and pipelined with three stages. Our proposed designs are validated by functional verification, performing a simulation with 40000 random normal vectors plus corner/exception vectors. The vectors combination in three precision modes are presented in Table 4. A, B are two input floating point numbers, normal in Table 4 denotes a random normal number, INF denotes infinity and equal is also a random normal number but indicating A=B. Because subnormal number is not supported, the test vector has no subnormal number in Table 4. The EDA tools we used is Synopsys VCS-2014.03. To evaluate the designs, all the designs in combinational and pipelined form are entirely synthesized using Synopsys Design Compiler 2013.12-SP5.

For we mostly concern about performance, so the logic synthesis criteria is in terms of delay. In synthesis process, we applied Synopsys y’s Topographic technique, which can obtain the best
correspondence between synthesis and placement route results. Table 5 shows the delay, Table 6 shows the area and Table 7 shows the power of all the adders we implemented. In all the three tables, PIPE means the design is in pipeline for m, COMB denotes the design is in combinational form and PIPE(RR) means that the design in pipeline form is synthesized with Register Repositioning technique. The delay, area and power are also compared to the results that presented in [27] and [31]. The delay is represented in nanosecond and FO4, and the area is represented in square micrometer and the number of gates. The logic synthesis library we used is TSMC 65 nm CMOS standard cell library. The area of the minimum inverter gate in our used library is 1.44 $\mu m^2$, and the delay of FO4 is roughly 0.0325 ns. The decimal number in parentheses in each row of Table 5 is the period of constraining clock.

In Table 5, for pure combinational circuits, the triple-mode adder has roughly 10% more delay (2.0 ns VS 1.81 ns), compared to single-mode quadruple precision adder. Since there is no triple-mode floating-point adder in previous literature, we use dual-mode adder implemented with the proposed architecture to compare performance with previous research work. The delay in FO4 of our proposed dual-mode adder is 58, just 67% of the delay (87) presented in [31]. With pipeline form, the delay of stage 1, stage 2 and stage 3 is 1.09ns, 1.06ns and 1.08ns respectively for triple-mode adder, and 0.92ns, 0.91ns, 0.92ns for dual-mode quadruple precision adder. The triple-mode adder in pipeline form has roughly 22% more delay than single-mode quadruple precision adder, and the quadruple precision dual-mode adder has 13% more delay. The worst delay of our proposed dual-mode adder is 29.2 in FO4 compared to 31.8 presented in [27]. The dual-mode adder presented in [27] has no exception processing circuit. Taking into this account, our proposed architecture is better than that of [27]. Compared to the total latency 113.6(28.4*4) in FO4 presented in [31], the dual-mode adder we designed has a faster speed which is 87.6 (29.2*3) in FO4. The total latency of the triple-mode adder we designed is 101.4 (33.8*3) is also smaller than that in [31]. Besides that, our proposed dual-mode and triple-mode quadruple precision adder can run at 125 0MHz, 1176MHz and 1075MHz respectively (clock period is 0.8ns, 0.85ns and 0.93ns respectively).

In Table 6, the area of our designed quadruple precision triple-mode adder in combinational and pipeline form is 66916 and 71290 $\mu m^2$ respectively. After register repositioning, the area changes to 85013 $\mu m^2$. The gate number of our design quadruple precision triple-mode adder in combinational and pipeline form is 46469 and 49507 respectively. From Table 6, we can conclude that the area of higher precision adder is roughly

### Table 4. Testing vector patterns including exceptions and corners

<table>
<thead>
<tr>
<th>operand</th>
<th>exception and corners, 100 random vectors for normal number in each case</th>
<th>40000 random vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NaN NaN INF INF NaN normal INF normal 0 equal normal</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>NaN INF NaN INF normal NaN normal INF 0 equal normal</td>
<td></td>
</tr>
</tbody>
</table>

### Table 5. Latency of the proposed designs

<table>
<thead>
<tr>
<th>LATENCY</th>
<th>Single</th>
<th>Double</th>
<th>Quad</th>
<th>Quad (Dual-Mode)</th>
<th>Quad (Tri-Mode)</th>
<th>[27]Quad (Dual-Mode)</th>
<th>[31]Double (Dual-Mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMB (ns)</td>
<td>1.34(74%)</td>
<td>1.54(85%)</td>
<td>1.81(100%)</td>
<td>1.90(105%)</td>
<td>2.00(110%)</td>
<td>--</td>
<td>7.84</td>
</tr>
<tr>
<td>FO4</td>
<td>41</td>
<td>47</td>
<td>55</td>
<td>58</td>
<td>62</td>
<td>--</td>
<td>87</td>
</tr>
<tr>
<td>PIPE</td>
<td>1</td>
<td>0.66(0.67)</td>
<td>0.78(0.8)</td>
<td>0.89(0.9)</td>
<td>0.92(0.95)</td>
<td>1.09(1.10)</td>
<td>0.74</td>
</tr>
<tr>
<td>2</td>
<td>0.50(0.67)</td>
<td>0.62(0.8)</td>
<td>0.71(0.9)</td>
<td>0.91(0.95)</td>
<td>1.06(1.10)</td>
<td>1.19</td>
<td>--</td>
</tr>
<tr>
<td>3</td>
<td>0.65(0.67)</td>
<td>0.74(0.8)</td>
<td>0.84(0.9)</td>
<td>0.92(0.95)</td>
<td>1.08(1.10)</td>
<td>1.59</td>
<td>--</td>
</tr>
<tr>
<td>worst (ns)</td>
<td>0.66(74%)</td>
<td>0.78(88%)</td>
<td>0.89(100%)</td>
<td>0.92(113%)</td>
<td>1.09(122%)</td>
<td>1.59</td>
<td>2.56(2.56*4)</td>
</tr>
<tr>
<td>FO4</td>
<td>20.6</td>
<td>24.6</td>
<td>27.7</td>
<td>29.2</td>
<td>33.8</td>
<td>31.8</td>
<td>28.4(d stages)</td>
</tr>
<tr>
<td>PIPE (RR)</td>
<td>1,2,3</td>
<td>0.58(0.6)</td>
<td>0.69(0.7)</td>
<td>0.78(0.8)</td>
<td>0.83(0.85)</td>
<td>0.90(0.93)</td>
<td>--</td>
</tr>
<tr>
<td>worst (ns)</td>
<td>0.58(75%)</td>
<td>0.69(88%)</td>
<td>0.78(100%)</td>
<td>0.83(106%)</td>
<td>0.90(116%)</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

| FO4 | 17.8 | 21.5 | 24.0 | 26.2 | 28.6 | -- | -- |
two times of that of lower precision adder. For example, when the circuit is pure combinational, the area of 32-bit adder is 10639 μm² while the area of 64-bit adder is 21885 μm²; when the circuit is pipelined, the area of 64-bit adder is 40936 μm² while the area of 128-bit adder is 55628 μm². Since the dual-mode adder presented in [31] is 64 bits, the number of gates is approximately estimated to be 2*10794 when extended to 128 bits. The gate count of our proposed dual-mode quadruple precision adder is 44323 which is greater than 2*10794. So the two-path algorithm is not suitable for area-efficient design.

Table 6. Area estimation of the proposed designs

<table>
<thead>
<tr>
<th>AREA</th>
<th>Single</th>
<th>Double</th>
<th>Quad (Dual-Mode)</th>
<th>Quad (Tri-Mode)</th>
<th>[27]Quad (Dual-Mode) (0.11μm)</th>
<th>[31]Double (Dual-Mode) (0.18μm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO MB</td>
<td>Area(μm²)</td>
<td>10639</td>
<td>21885</td>
<td>40936</td>
<td>55628</td>
<td>66916</td>
</tr>
<tr>
<td></td>
<td>Gate Count</td>
<td>7388</td>
<td>15198</td>
<td>28428</td>
<td>38631</td>
<td>46469</td>
</tr>
<tr>
<td>PIPE</td>
<td>Area(μm²)</td>
<td>12921</td>
<td>24243</td>
<td>50268</td>
<td>63528</td>
<td>71290</td>
</tr>
<tr>
<td></td>
<td>Gate Count</td>
<td>8793</td>
<td>16835</td>
<td>34908</td>
<td>44323</td>
<td>49507</td>
</tr>
<tr>
<td>PIPE (RR)</td>
<td>Area(μm²)</td>
<td>14618</td>
<td>29284</td>
<td>61293</td>
<td>68895</td>
<td>85013</td>
</tr>
<tr>
<td></td>
<td>Gate Count</td>
<td>10151</td>
<td>20336</td>
<td>42565</td>
<td>47844</td>
<td>59037</td>
</tr>
</tbody>
</table>

The data with a “#” is computed using scaled area: Area(65nm)=Area(110nm)*(65/110)^2=357399*(65/110)^2=124794

Table 7. Power estimation of the proposed designs

<table>
<thead>
<tr>
<th>POWER</th>
<th>Single</th>
<th>Double</th>
<th>Quad (Dual-Mode)</th>
<th>Quad (Tri-Mode)</th>
<th>[27]Quad (Dual-Mode) (0.11μm)</th>
<th>[31]Double (Dual-Mode) (0.18μm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO MB</td>
<td>combinational</td>
<td>2.53</td>
<td>4.22</td>
<td>7.12</td>
<td>8.64</td>
<td>9.93</td>
</tr>
<tr>
<td>registers</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
</tr>
<tr>
<td>total</td>
<td>2.53</td>
<td>4.22</td>
<td>7.12</td>
<td>8.64</td>
<td>9.93</td>
<td>--</td>
</tr>
<tr>
<td>PIPE</td>
<td>combinational</td>
<td>3.29</td>
<td>5.19</td>
<td>8.72</td>
<td>10.88</td>
<td>11.73</td>
</tr>
<tr>
<td>registers</td>
<td>7.59</td>
<td>12.46</td>
<td>22.16</td>
<td>22.09</td>
<td>21.13</td>
<td>--</td>
</tr>
<tr>
<td>total</td>
<td>10.88</td>
<td>17.64</td>
<td>30.88</td>
<td>32.97</td>
<td>32.86</td>
<td>--</td>
</tr>
<tr>
<td>PIPE (RR)</td>
<td>combinational</td>
<td>3.3126</td>
<td>5.07</td>
<td>10.68</td>
<td>11.48</td>
<td>12.72</td>
</tr>
<tr>
<td>registers</td>
<td>9.7653</td>
<td>20.17</td>
<td>28.57</td>
<td>30.56</td>
<td>29.36</td>
<td>--</td>
</tr>
<tr>
<td>total</td>
<td>13.08</td>
<td>25.24</td>
<td>39.25</td>
<td>42.03</td>
<td>42.08</td>
<td>--</td>
</tr>
</tbody>
</table>

5 Conclusion

This paper presents an architecture of improved two-path algorithm for floating-point adders. By using flagged parallel prefix adder (FPPA) to replace comparator and compound adder, the rounding process is simplified and delay is decreased. Using two ways of simple LZA-LZC in [21, 23] instead of exact LZA [39-40] not only
decreases the latency but also keep the area in a reasonable range.

Also this paper shows how to modify the proposed architecture to support multiple precision addition/subtraction. The proposed triple-mode quadruple precision floating-point adder can perform four parallel single precision or two parallel double precision or a quadruple precision addition/subtraction. To support multiple precision, we designed a triple-mode normalization logic, a triple-mode alignment logic and a triple-mode FPPA. We also modified the main components of our proposed architecture including leading-zero detection logics. The triple-mode normalization and alignment logic require a very small increase in delay and a relatively reasonable increase in area compared to single-mode adder. On the other hand, the extra multiplexors introduced to support triple-mode operations result in a slightly increase in delay and area.

The synthesis results show that the proposed triple-mode quadruple precision adder requires 10-16% more delay than the single-mode quadruple precision adder. The proposed triple-mode adder saves 47-52% area and is very useful for SIMD and scientific applications. To the author’s knowledge, this is the first triple-mode floating-point adder.

References:


