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 co mputation in low er precision can be ap plied in SIMD applicat ion to accommodate 3D graphics, video conferencing and multim edia fields while its high precision computation can be applied in scientific applications such as supernov a simulations, climate modeling and etc. To im prove the performance of the triple-mode floating-point adder, the design is implemented with the improved two-path algorithm in co mbinational and pipeline form . To com pare area, power and worst-ca se 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 manufacture 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 chi ps for its fre edom from overflow and underflow and ease of using to programmers. Today many general purpose processors offe r 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 m ore parallel single precision floating-poi nt units (FPU). For image processing and 3D video ga me, a large number of single precision floating- point operations are executed through many parallel FPUs in graphic chips [3]. NVIDIA's GTX680 with Kep ler 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] pointes out in his resea rch work that power ef ficiency 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 s upport some scientific applications. Although higher precision arithmetic can be implemented by sof tware, it is reported that hardware im plementation of a quadruple precision FPU is approximately 200 times faster than that of softwar e implementation [8]. For scientific applications, h igher precision fl oating-point computation is needed, which is another trend. D.H.Bailey [9] and G. H owell [10] describes th e necessity that higher precision arithmetic is useful for a variet y of situatio ns including ill-conditioned linear systems, large scale si mulation etc. Although 64-bit IEEE [11] arithmetic is sufficient for those situations [10], supernova simulation and clim ate modeling still need higher precision su ch as 128-bit floating-point arithmetic. E.Schwarz [12] presents in their paper that quadr uple precision (1 28-bit) floating-point unit can be im plemented in a reasonable amount of hardware co mpared to double precision.

In the past decade, the soft ware and hardware has gradually transformed from 32-bit to 6 4-bit. Today, from personal devices such as PC, mobile phone to enterprise workstation, from software t o hardware, 64-bit has be come 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 microprocessor 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 become 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] presentes 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 divider performs one double/two single precision division. 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 [37] respectively presents 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 single, double and dual-mode quadruple precision adders using the similar techniques. All the adders with both combinational and pipeline forms 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 following: *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 FAR path, 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 mantissas. 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

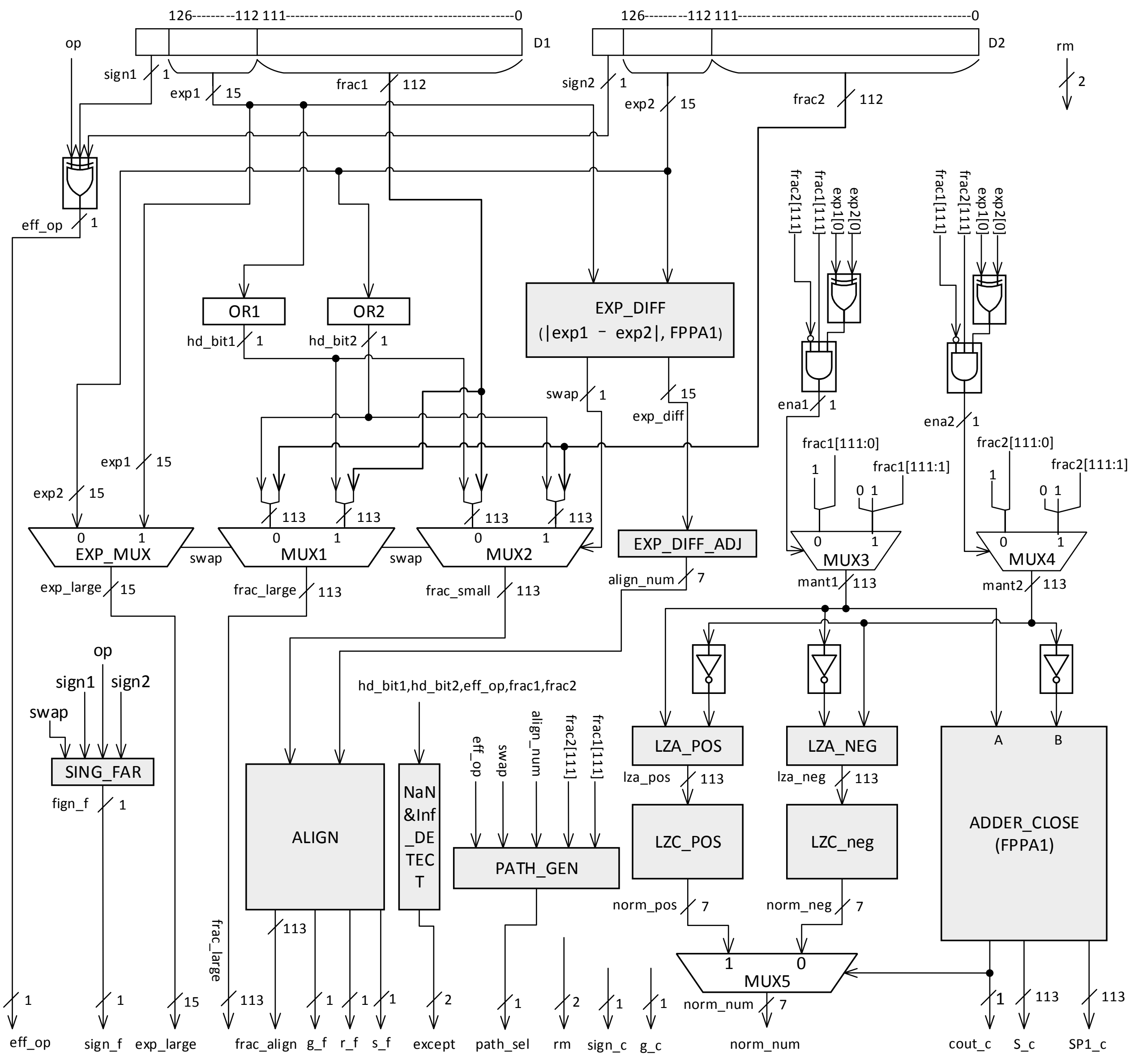


Fig. 1 The logics of the first stage of the proposed single-mode quadruple precision floating-point adder

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 num ber and adjusted to 7 bits through *EXP_DIFF_ADJ* block. The circuit o f *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 m ost 116 bits, so 7 bits is sufficient to h old a number that is greater than or equal to 116. Anot her objective of adjusting the exponent diff erence 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 alignm ent shift block *ALIGN* is generally implemented using a barrel shifter which is com posed of seve ral 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 tim es the area of one level of m ultiplexers. 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 Fi g. 2(b). T he *Nan&Inf_DETECT* block receives *eff_op*, exponents and mantissas of the two

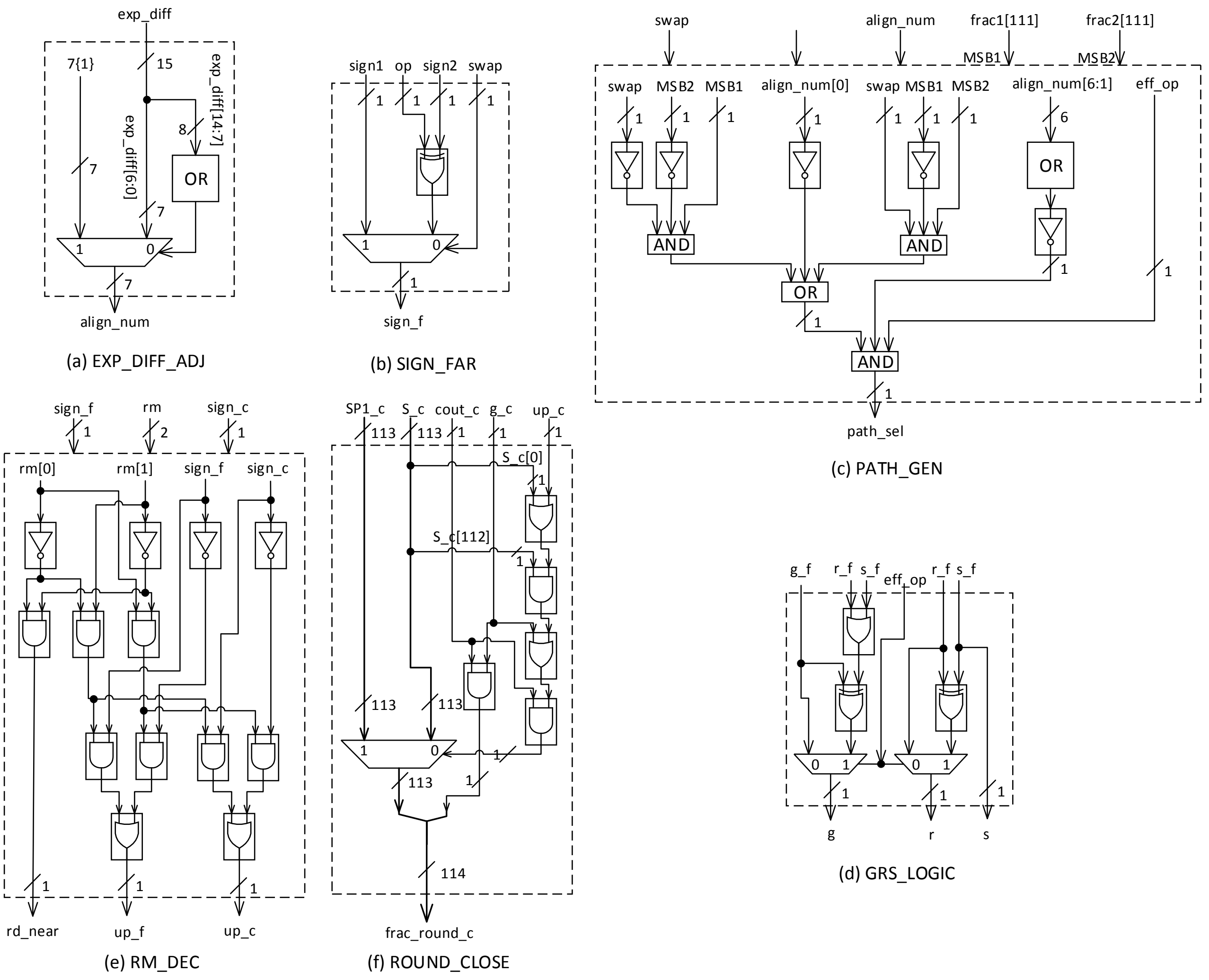


Fig. 2. The circuit of components in the first stage

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 **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≠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 *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 *mant2* and *mant1* are passed to *LZA_POS* and *ADDER_CLOSE* blocks, and the shifted and inverted mantissa *mant1* and *mant2* are passed to *LZA_NEG* block. In the case that *mant1* is less than *mant2*, the leading zero number *norm_pos* detected by *LZA_POS* and *LZC_POS* is false, but *norm_neg* is true. In the later situation, the signal *cout_c* is 0, so the correct leading zero number *norm_num* can always be obtained through *MUX5*. 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 multi-mode 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 *exp_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 consume less power and area, but the delay increases as the bit width grows. The logic level of LZA error correcting circuit in [40] is $3log_2n+8$, which is far greater than the logic level of the adder tree ($log2_n+2$) 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 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 *ADDER_CLOSE* block is implemented using a FPPA1 which computes $A+\bar{B}(A-B-1)$, $A+\bar{B}+1(A-B)$, and $B-A$. The detail of a FPPA1 is shown in [20]. When *cout_c* is 1, the output sum *S_c* of CLOSE path is $A+\bar{B}$ and the *SP1_c* equals to $S\_c+1(A+\bar{B}+1)$; when *cout_c* is 0, *S_c* is equal to *SP1_c* ($B-A$). 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 of FAR path, generate the guard, round and sticky bit of FAR path, round and normalize the difference of mantissas of CLSOE path. Fig. 3 shows the details of the second stage.

The *ADDER_FAR* block is used to compute the sum/difference. When the effective operation is addition and the rounding mode is rounding toward to 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 $A+B$, $A+B+1$, $A+B+2$, $A+\bar{B}$, $A+\bar{B}+1$, $A+\bar{B}+2$ and $B-A$. For two operands $A=\{a_{n-1},..,a_2,a_1,a_0\}$ and $B=\{b_{n-1},..,b_2,b_1,b_0\}$, the flag signal $flag2=\{f_{n-1},..,f_2,f_1,f_0\}$ for obtaining $A+B+2$ is produced from the two signals $G=\{g_{n-1},..,g_2,g_1,g_0\}$ and $P=\{p_{n-1},..,p_2,p_1,p_0\}$: $g_i=a_i{\cdot}b_i$, $p_i=a_i\wedge b_i$ $(i=0,1,2\ldots)$; $f_0=0$, $f_1=1$, $f_2=p_1\wedge g_0$, $f_i=(p_{i-1}\wedge g_{i-2}){\cdot}f_{i-1}$ $(i>2)$. Suppose the sum of A and B is S ($S=A+B$), then $S+2$ can be obtained using the following formula: $S+2=S\wedge flag2$ ($s_i\wedge f_i$). The "·" means AND operation and "^" means XOR operation. The flag signal *flag1* for obtaining $A+B+1$ is the same as described in [20] and $S+1=S\wedge flag1$. 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 half-adders, 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*, *SP1_f* and *SP2_f* in Fig. 3.

The *GRS_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 *RM_DEC* block is used to decode the rounding mode. When *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 (*sign_c*=0 or *sign_f*=0), or round toward negative infinity (*rm*=2) and the sign is negative (*sign_c*=1 or *sign_f*=1), the signal *up_f* indicating rounding toward infinity is activated. The circuit of *RM_DEC* is shown in Fig. 2(e).

Since the result $S+1$(*SP1_c*) and the result S (*S_c*) of CLOSE path has been computed, the rounding

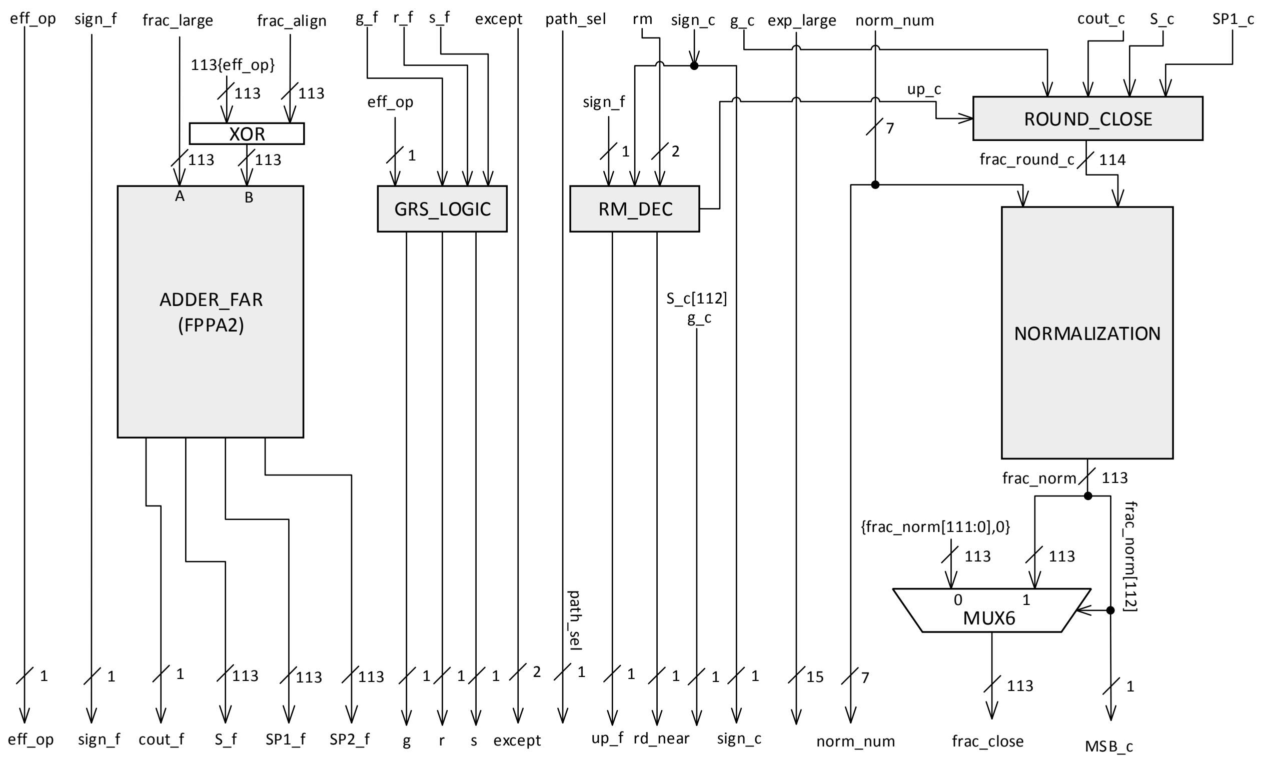


Fig. 3 The logics of the second stage of the proposed single-mode quadruple precision floating-point adder

process is executed through *ROUDN_CLOSE* block before normalization. [18] presents the rounding mode. The rounded result is a 114-bit signal and denoted as *frac_round_c*. The circuit of *ROUDN_CLOSE* block is shown in Fig. 2(f). Since the mantissa of CLOSE path is right shifted by one or zero bit through *MUX3* and *MUX4*, the round and sticky bit is 0, the guard bit is just the LSB of mantissa {*1,frac1*} ({*1,frac2*}) or 0. *g_c* is the guard bit and is not drawn in stage1 for clarity.

The *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 *frac_norm*—*MSB_c* is 0, *frac_norm* is left shifted by one bit through *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 *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 [18] in FAR path. For example, When *grs*=011, no carry can be propagated to the *LSB* of *S_f* because *g* equals to 0, the high order bits is selected as *S*. In this situation, if *MSB*=1, the result *S_f* needs no left shifting, there is no carry propagated to *LSB* of *S*; if *MSB*=0, *sMSB*=1, the result *S_f* needs left shifting by one bit, after left shifting, *g* become the new least significant bit of the result, and cause *rs*=11, there is a carry propagated to *g* which turns *g* from 0 to 1; if *MSB*=0, *sMSB*=0, the result *S* needs left shifting by two bits, *g* and *r* are shifted into the result, and cause *s*=1 and *r*=1, there is a carry propagated to g which turns *g* from 0 to 1 and *r* from 1 to 0. When *up_f* equals to 0 or rounding toward zero (*rd_zero*) is active, no rounding is needed and the two bits shifted in keep the value of *r* and *s*. The rounding mode is summarized in Table 1 and 2, and the circuit of *ROUND_FAR* block is shown in Fig. 5(a). In Table 1, *LSB* is the least significant bit of *S_f*, *MSB* is the most significant bit of *S_f*, *sMSB* is the bit right next to *MSB*, *C* is the carry out of FAR path— *cout_f*,

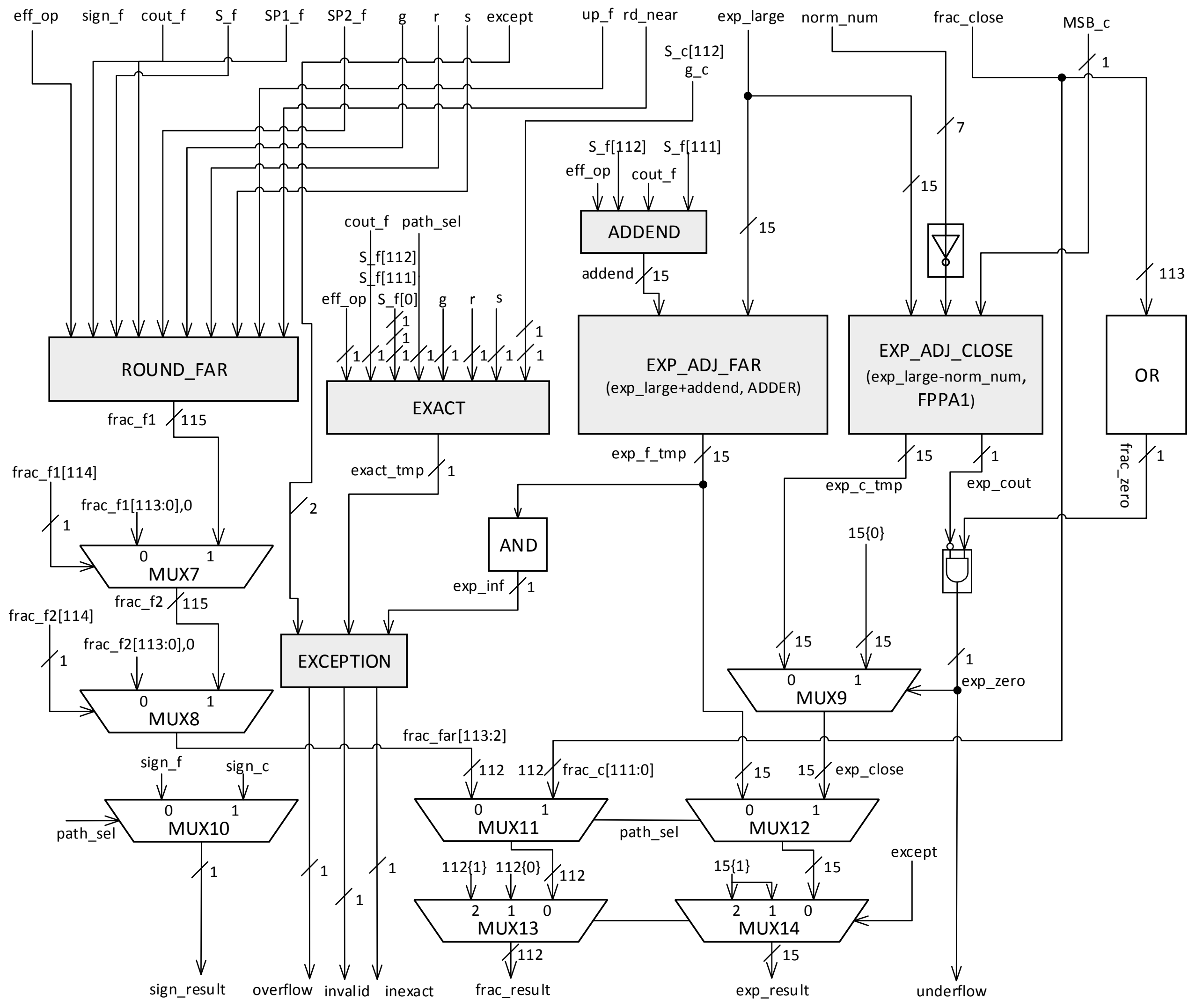


Fig. 4 The logics of the third stage of the proposed single-mode quadruple precision floating-point adder

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 111111111111110 (-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 000000000000001 (+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(c). Since

Table 1. The rounding mode with effective subtraction operation

R		*C*=1					*C*=0				
		rd_near		*up_f*		*zero*	*rd_near*		*up_f*		*zero*
		LSB=0	*LSB*=1	1	0	--	*LSB*=0	*LSB*=1	1	0	--
grs	000	*C,S*	*C,SP1*(*sLSB*=1) *C,S*(*sLSB*=0)	*C,SP2*(*LSB*=1) *C,S*(*LSB*=0)	*C,S*	*C,S*	*S*	*S*	*S*	*S*	*S*
	001	*C,S*	*C,SP1*	*C,SP2*	*C,S*	*C,S*	*S*	*S*	*SP1*	*S*	*S*
	010	*C,S*	*C,SP1*	*C,SP2*	*C,S*	*C,S*	*S*	*S*	*SP1*	*S*	*S*
	011	*C,S*	*C,SP1*	*C,SP2*	*C,S*	*C,S*	*S*	*S*	*SP1*	*S*	*S*
	100	*C,S*	*C,SP1*	*C,SP2*	*C,S*	*C,S*	*S*	*SP1*	*SP1*	*S*	*S*
	101	*C,S*	*C,SP1*	*C,SP2*	*C,S*	*C,S*	*SP1*	*SP1*	*SP1*	*S*	*S*
	110	*C,S*	*C,SP1*	*C,SP2*	*C,S*	*C,S*	*SP1*	*SP1*	*SP1*	*S*	*S*
	111	*C,S*	*C,SP1*	*C,SP2*	*C,S*	*C,S*	*SP1*	*SP1*	*SP1*	*S*	*S*

Table 2. The rounding mode with effective addition operation

R		*rd_near*			*up_f*				*zero*
					1			0	
		MSB=1	*MSB*=0 *sMSB*=1	*MSB*=0 *sMSB*=0	*MSB*=1	*MSB*=0 *sMSB*=1	*MSB*=0 *sMSB*=0	--	--
grs	000	*SP1,00*	*SP1,00*	*SP1,00*	*SP1,00*	*SP1,00*	*SP1,00*	*SP1,00*	*SP1,00*
	111	*SP1,00*	*SP1,00*	*SP1,00*	*SP1,00*	*SP1,00*	*SP1,00*	*S,11*	*S,11*
	110	*SP1,00*	*SP1,00*	*S,11*	*SP1,00*	*SP1,00*	*S,11*	*S,11*	*S,11*
	101	*SP1,00*	*S,10*	*S,10*	*SP1,00*	*SP1,00*	*S,10*	*S,10*	*S,10*
	100	*SP1,00*(*LSB*=1)	*S,10*	*S,10*	*SP1,00*	*S,10*	*S,10*	*S,10*	*S,10*
	011	*S,10*(*)	*S,10*(*)	*S,10*(*)	*SP1,00*	*S,10*(*)	*S,10*(*)	*S,01*	*S,01*
	010	*S,01*	*S,01*	*S,01*	*SP1,00*	*S,01*	*S,01*	*S,01*	*S,01*
	001	*S,00*	*S,00*	*S,00*	*SP1,00*	*S,00*	*S,00*	*S,00*	*S,00*

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

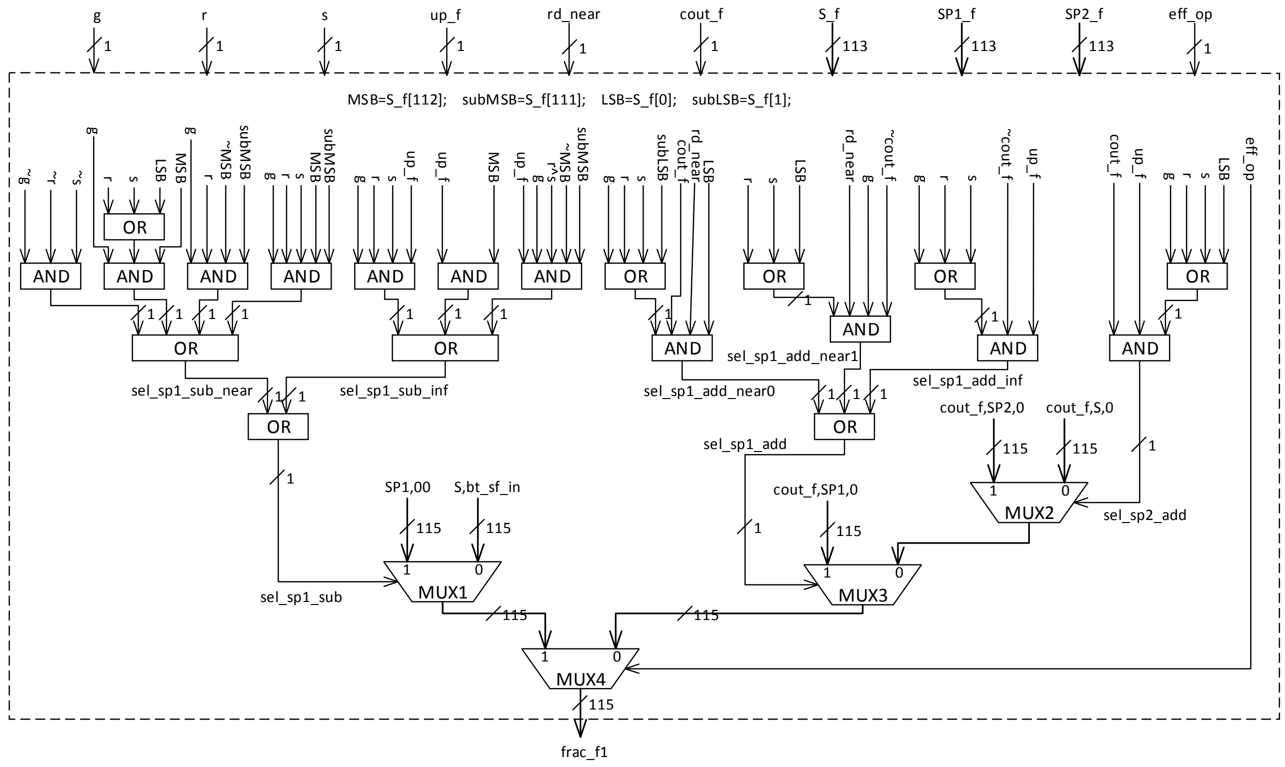
In this section, a triple-mode quadruple precision floating-point adder is designed with the architecture of the improved two-path algorithm.

3.1 Stage 1

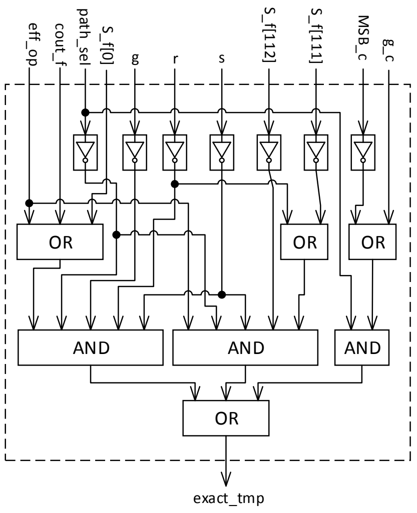
Pipeline stage 1 is shown in Fig. 6. For clarity the rounding mode (*rm*), and precision mode (*op_mode*) signals are not drawn in Fig. 6.

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.

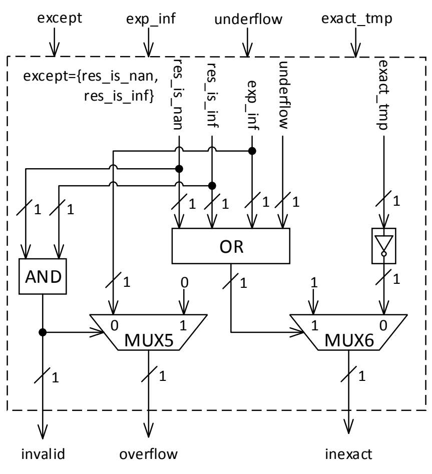
The signs, exponents and fractions of each operands in various precision modes is shown in Fig. 6. For example, the sign of *Q1* is the MSB of *S1*; the exponent of *Q1* is *s1[30:16]*, denoted as *exp_q1*; the fraction of *Q1* is *{ S1[15:0], S2[31:0], S3[31:0], S4[31:0]}*, denoted as *frac_q1*.In this paper, without



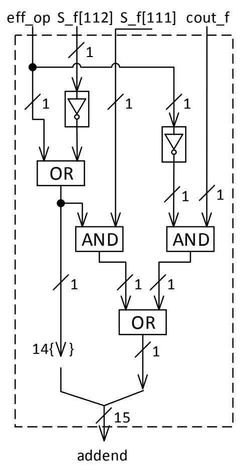
(a) ROUDN_FAR



(b) EXACT



(c) EXCEPTION

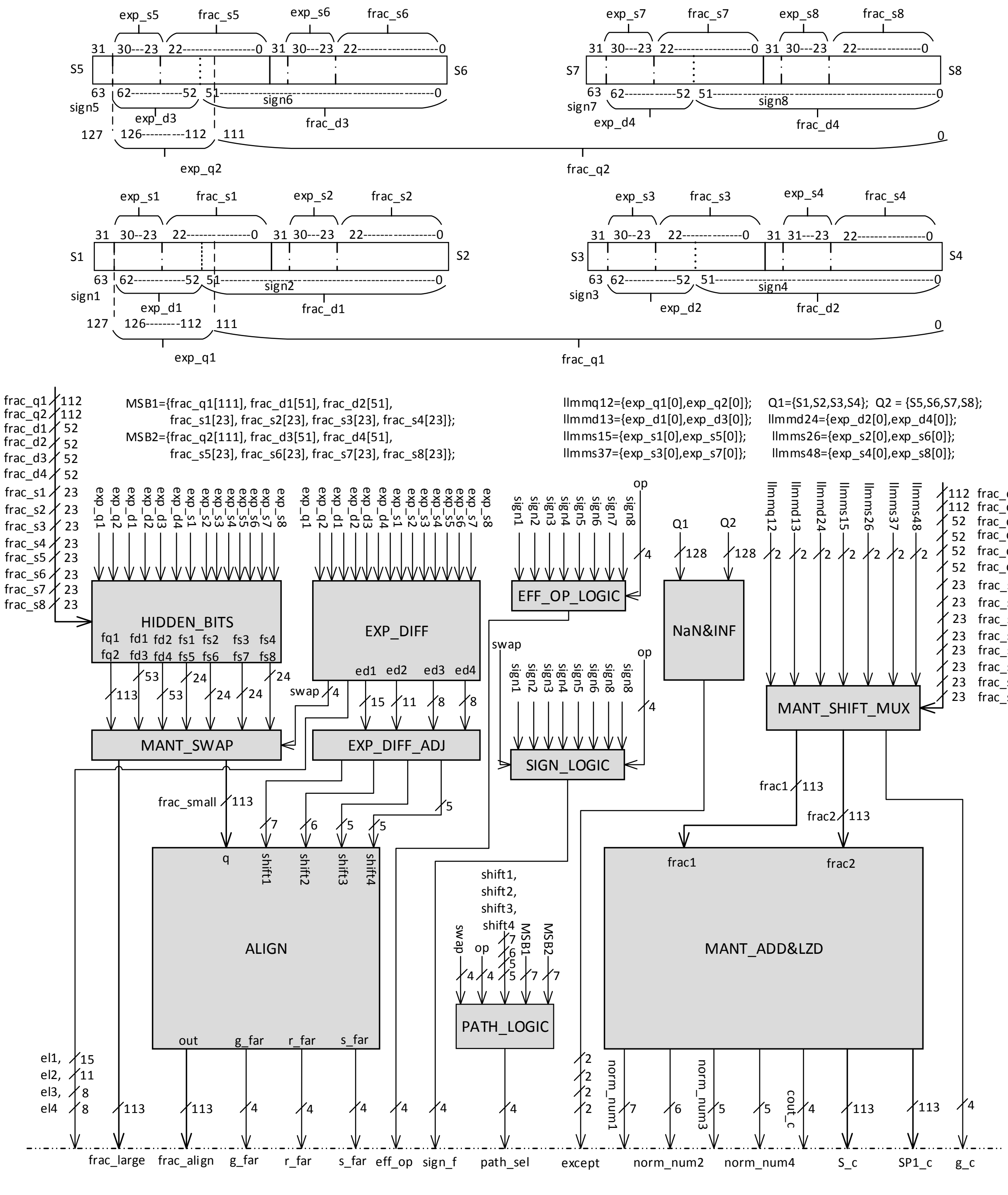


(d) ADDEND

Fig. 5. The circuits of components in the third stage

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). *exp_d1* is extended to *{0000, exp_d1}* and *exp_d3* is extended to *{0000, exp_d3}*. *exp_s1* is extended to *{0000000, exp_s1}* and *exp_s5* is extended to *{0000000, 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. *MUX1, MUX2, MUX3* and *MUX4* are used to select the correct exponents based on the corresponding operation mode. The *Exp_Diffj* (*j=1,2,3,4*) is similar to the one used in single-mode adder previously described, and produces the exponent difference signal *edi* (*i=1,2,3,4*) and the



exp_s5 frac_s5 exp_s6 frac_s6
exp_s7 frac_s7 exp_s8 frac_s8
S5 S6 S7 S8
sign5 sign6 sign7 sign8
exp_d3 frac_d3 exp_d4 frac_d4
exp_q2 frac_q2
exp_s1 frac_s1 exp_s2 frac_s2
exp_s3 frac_s3 exp_s4 frac_s4
S1 S2 S3 S4
sign1 sign2 sign3 sign4
exp_d1 frac_d1 exp_d2 frac_d2
exp_q1 frac_q1
MSB1={frac_q1[111], frac_d1[51], frac_d2[51], frac_s1[23], frac_s2[23], frac_s3[23], frac_s4[23]};
MSB2={frac_q2[111], frac_d3[51], frac_d4[51], frac_s5[23], frac_s6[23], frac_s7[23], frac_s8[23]};
llmmq12={exp_q1[0],exp_q2[0]}; Q1={S1,S2,S3,S4}; Q2 = {S5,S6,S7,S8};
llmmd13={exp_d1[0],exp_d3[0]}; llmmd24={exp_d2[0],exp_d4[0]};
llmms15={exp_s1[0],exp_s5[0]}; llmms26={exp_s2[0],exp_s6[0]};
llmms37={exp_s3[0],exp_s7[0]}; llmms48={exp_s4[0],exp_s8[0]};
HIDDEN_BITS
EXP_DIFF
EFF_OP_LOGIC
NaN&INF
MANT_SHIFT_MUX
MANT_SWAP
EXP_DIFF_ADJ
SIGN_LOGIC
ALIGN
PATH_LOGIC
MANT_ADD&LZD
frac_small
shift1 shift2 shift3 shift4
swap op MSB1 MSB2
frac1 frac2
el1, el2, el3, el4
frac_large frac_align g_far r_far s_far eff_op sign_f path_sel except norm_num2 norm_num4 S_c SP1_c g_c
norm_num1 norm_num3 cout_c

Fig. 6. The first pipeline stage of the triple-mode quadruple precision floating-point adder.

swap signal *swap[i]* (*i=1,2,3,4*). The larger exponent *eli* (*i=1,2,3,4*) is m ultiplexed through the multiplexor under each FPPA1.

The exponent difference *edi* (*i=1,2,3,4*) of each pair of operands are m odified to appropriate bit width and t he reason is explained in Section 2.1. The *EXP_DIFF_ADJ* is consist of four exponen t difference adjust logics shown in Fig. 2(a). The *SIGN_LOGIC*, *PATH_LOGIC*, *EFF_OP_LOGIC* blocks are respectively consist of four *SIGN_FAR*s, four *PATH_GEN*s and four *XOR* gates illustrated in Fig. 2(b) and (c).

The *HIDDEN_BITS* block in Fig. 6 is used t o determine the hidden leadin g bit for each mantissa

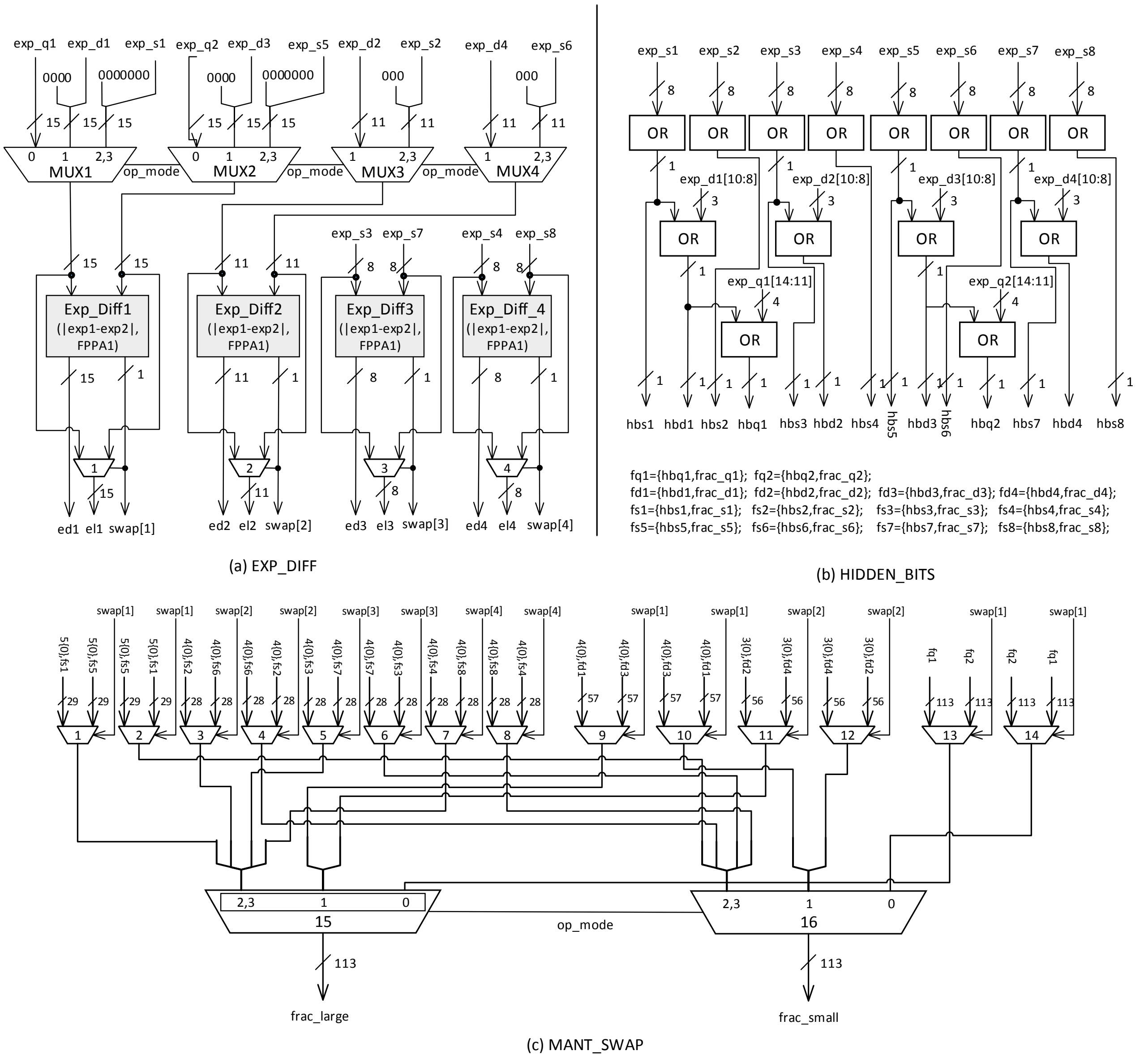

MUX1
MUX2
MUX3
MUX4
op_mode
Exp_Diff1
Exp_Diff2
Exp_Diff3
Exp_Diff_4
(|exp1-exp2|, FPPA1)
ed1 el1 swap[1]
ed2 el2 swap[2]
ed3 el3 swap[3]
ed4 el4 swap[4]
(a) EXP_DIFF
OR
hbs1 hbd1 hbs2 hbq1 hbs3 hbd2 hbs4 hbs5 hbd3 hbs6 hbq2 hbs7 hbd4 hbs8
fq1={hbq1,frac_q1}; fq2={hbq2,frac_q2};
fd1={hbd1,frac_d1}; fd2={hbd2,frac_d2}; fd3={hbd3,frac_d3}; fd4={hbd4,frac_d4};
fs1={hbs1,frac_s1}; fs2={hbs2,frac_s2}; fs3={hbs3,frac_s3}; fs4={hbs4,frac_s4};
fs5={hbs5,frac_s5}; fs6={hbs6,frac_s6}; fs7={hbs7,frac_s7}; fs8={hbs8,frac_s8};
(b) HIDDEN_BITS
15
16
op_mode
frac_large
frac_small
(c) MANT_SWAP

Fig. 7. The circuits of *EXP_DIFF*, *HIDDEN_BITS* and *MANT_SWAP*

Table 3. Combination of various precision modes and operations

		quadruple	double	single
op[1]	0	*Q1+Q2*	*D1+D3*	*S1+S5*
	1	*Q1-Q2*	*D1-D3*	*S1-S5*
op[2]	0	--	*D2+D4*	*S2+S6*
	1	--	*D2-D4*	*S2-S6*
op[3]	0	--	--	*S3+S7*
	1	--	--	*S3-S7*
op[4]	0	--	--	*S4+S8*
	1	--	--	*S4-S8*

in different precision modes. The circuit of this block is shown in Fig. 7(b). The mechanism is described in section 2.

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 *swap[i]* (*i=1,2,3,4*) 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 *frac_large* and *frac_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

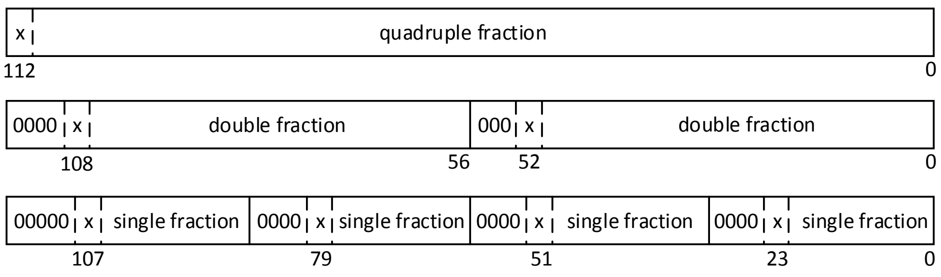


Fig. 8. The data structure of mantissa for various operation modes

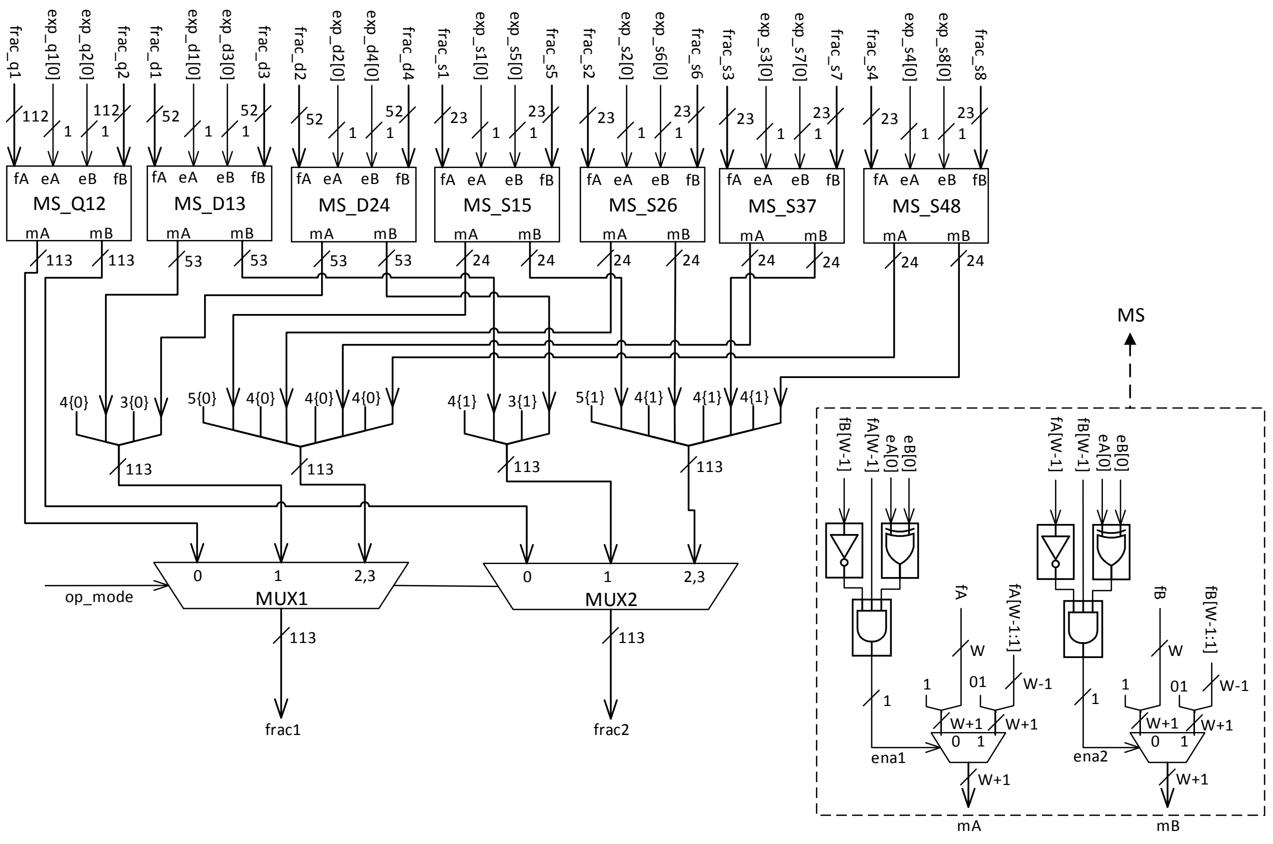


Fig. 9. The circuit of *MANT_SHIFT_MUX* block

described in single- mode adder. It pr oduces four exception signals for each pre cision 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 bloc k consists of seven *MS* blocks shown in Fig. 9. T he techniques of *MS* block is si milar to the one described in Section 2, and redrawn in the right lower corner in Fig. 9 a nd *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 pat h, 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 u sed in single-m ode adder, which depends on the precision mode as following:

quadruple:

flag1[0]=1,flag1[i]=A[i-1]^B[i-1] flag1[i-1];

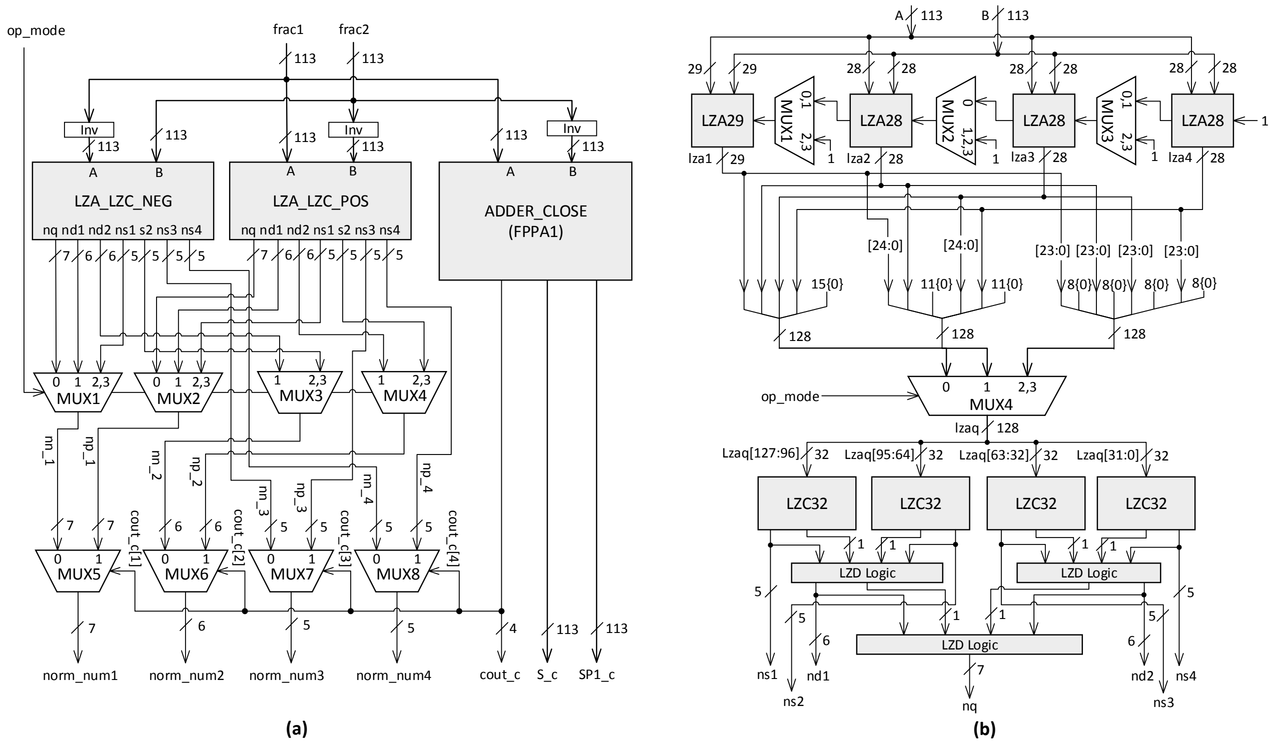


Fig. 10. The circuit of *MANT_ADD&LZD* block

cout_c[1]=Cout (*Cout* is the carry out of *A+B*)

double:

flag1[0]=1,flag1[56]=1,

flag1[i]=A[i-1]^B[i-1]·flag1[i-1] (*i≠0,56*)

cout_c[1]=S_c[109], *cout_c[2]=S_c[53]*

single:

flag1[0]=1, *flag1[28]=1*, *flag1[56]=1*, *flag1[84]=1*,

flag1[i]=A[i-1]^B[i-1]·flag1[i-1] (*i≠0,28,56,84*)

cout_c[1]=S_c[107], *cout_c[2]=S_c[80]*,

cout_c[3]=S_c[52], *cout_c[4]=S_c[24]*

The sum $S\_c = frac1 + \overline{frac2}$, $SP1\_c = S\_c\text{^}flag1$.

As described in Section 2, the *LZA_LZC_NEG* is used to obtain the correct number of leading zeros in case of 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 *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 concatenated 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 *lzai* (*i=1,2,3,4*) , *lzai[23:0]*, is extended with 8 trailing zeros. Each *LZC32* unit produces a 5-bit number from its corresponding leading zero anticipating string, two *LZC32*s plus one *LZD Logic* unit constitutes a 64-bit LZC unit and produces a 6-bit number. Finally two 64-bit LZC unit constitute a

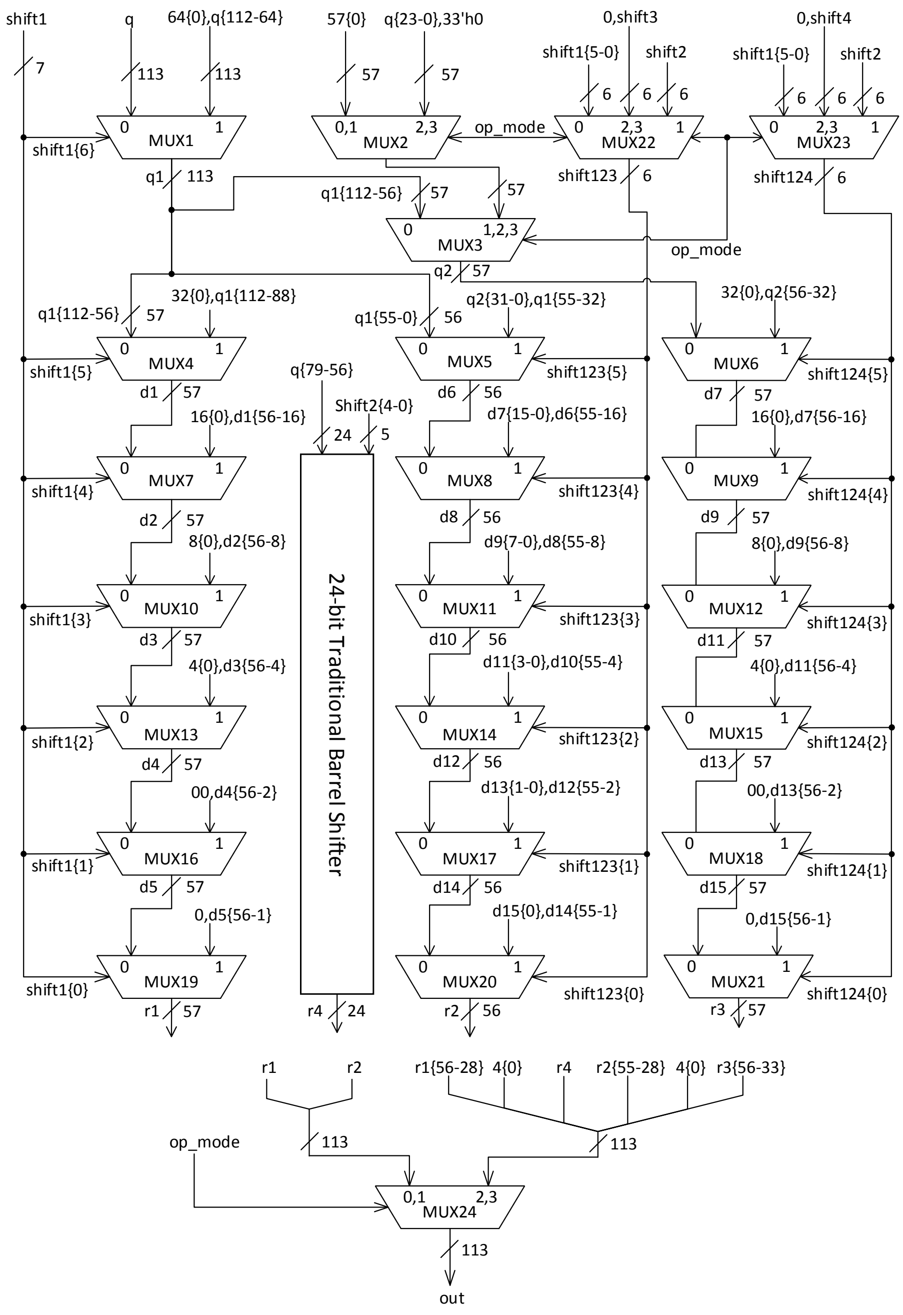


Fig. 11. The ALIGN block supports single, double and quadruple mode

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 *q2* 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 works 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 mantissas *frac_align* and *frac_large* in FAR path are added/subtracted 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 *MUX1* 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 previous 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 Stage1 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 is 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 mantissas 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_Fi* block are shown at the top of Fig. 13.

The four *EXACTi* (*i=1,2,3,4*) blocks and four *EXCEPTIONi* blocks are identical to the ones in Fig.

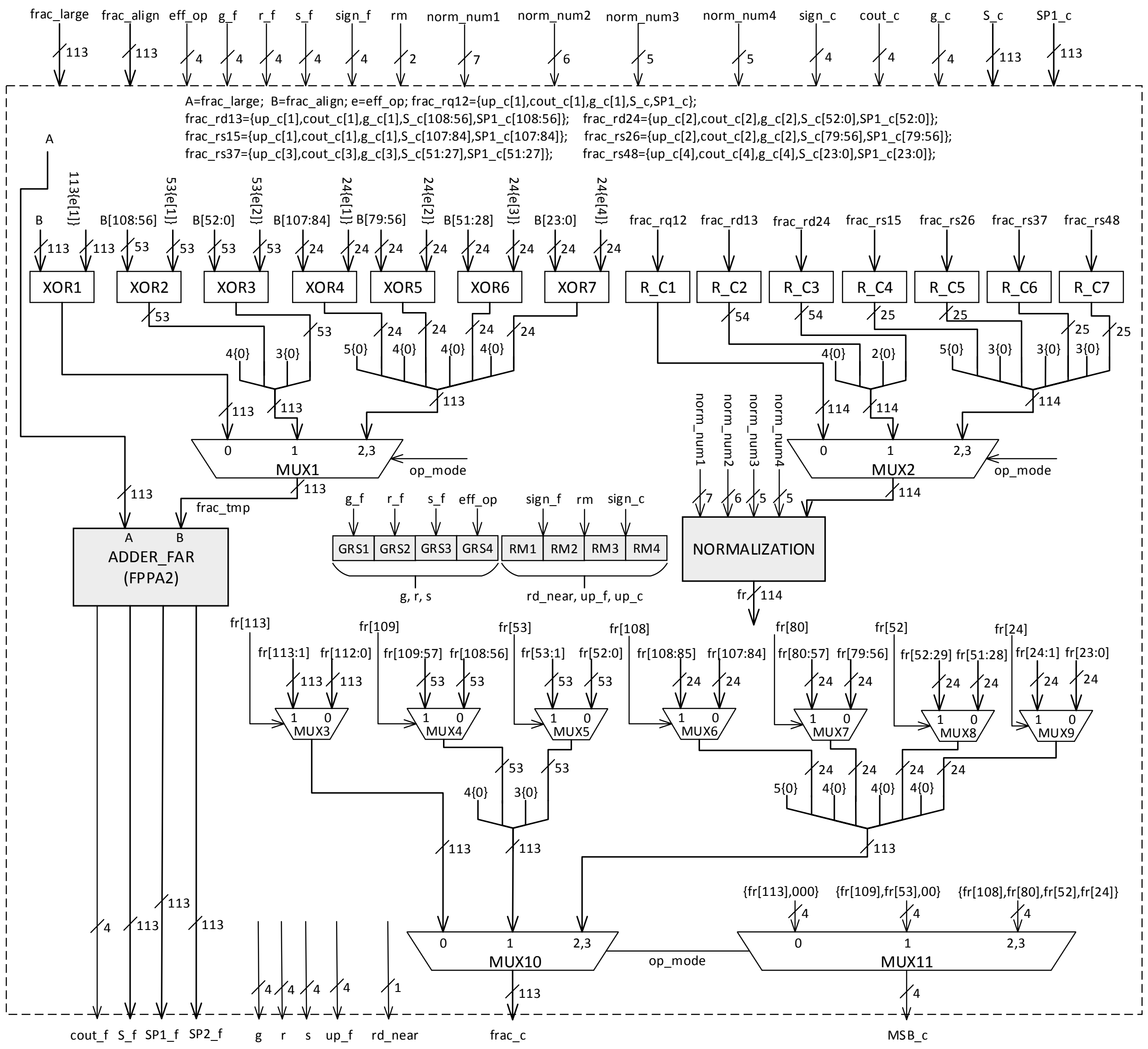

A=frac_large; B=frac_align; e=eff_op; frac_rq12={up_c[1],cout_c[1],g_c[1],S_c,SP1_c};
frac_rd13={up_c[1],cout_c[1],g_c[1],S_c[108:56],SP1_c[108:56]}; frac_rd24={up_c[2],cout_c[2],g_c[2],S_c[52:0],SP1_c[52:0]};
frac_rs15={up_c[1],cout_c[1],g_c[1],S_c[107:84],SP1_c[107:84]}; frac_rs26={up_c[2],cout_c[2],g_c[2],S_c[79:56],SP1_c[79:56]};
frac_rs37={up_c[3],cout_c[3],g_c[3],S_c[51:27],SP1_c[51:27]}; frac_rs48={up_c[4],cout_c[4],g_c[4],S_c[23:0],SP1_c[23:0]};
ADDER_FAR (FPPA2)
MUX1
MUX2
NORMALIZATION
MUX10
MUX11
op_mode
cout_f S_f SP1_f SP2_f g r s up_f rd_near frac_c MSB_c

Fig. 12. The second pipeline stage of the triple-mode quadruple precision floating-point adder.

5(b)(c) and used to produc e 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. 1 4, like the method used in sin gle-mode adder, four groups of exponents adjusting logic (*ADDENDi*, *ADDERi*) are used to adjust the exponents of FAR path. The seven *AND* gates ar e used to detect whether all the bits of the adjusted exponents are 1 in vario us precision m odes. If all the bits of th e adjusted exponents are 1, overflo w occurs and the signal *expj_inf* (*j=1,2,3,4*) turns into 1. For example, in double precision mode, if *exp1_f* is *xxxx11111111111*, *exp1_inf* is asserted through *MUX20*. The *ADDENDi* block is completely identical to the one in Fig. 5(d).

For CLOSE path in Fig. 14, four *FPPA1*s are used to subtract *norm_numi* from *expi_large* (*i=1,2,3,4*) to obtain the adjusted ex ponents. The seven *OR* gates are used to detect whether the mantissa result is zero or not. F or example in quadruple mode, when *frac_c* is not 0 and *exp1_cout* is 0 which means the adjusted exponen t is less than o r equal to 0, since our des ign does not support subnormal number, underflow occurs (*uf1*=1) and *15{0}* is passed to *exp1_c* as the exponent of CLOSE path. *MUX22* and *MUX23* are used to select the right underflow signals in various precision modes.

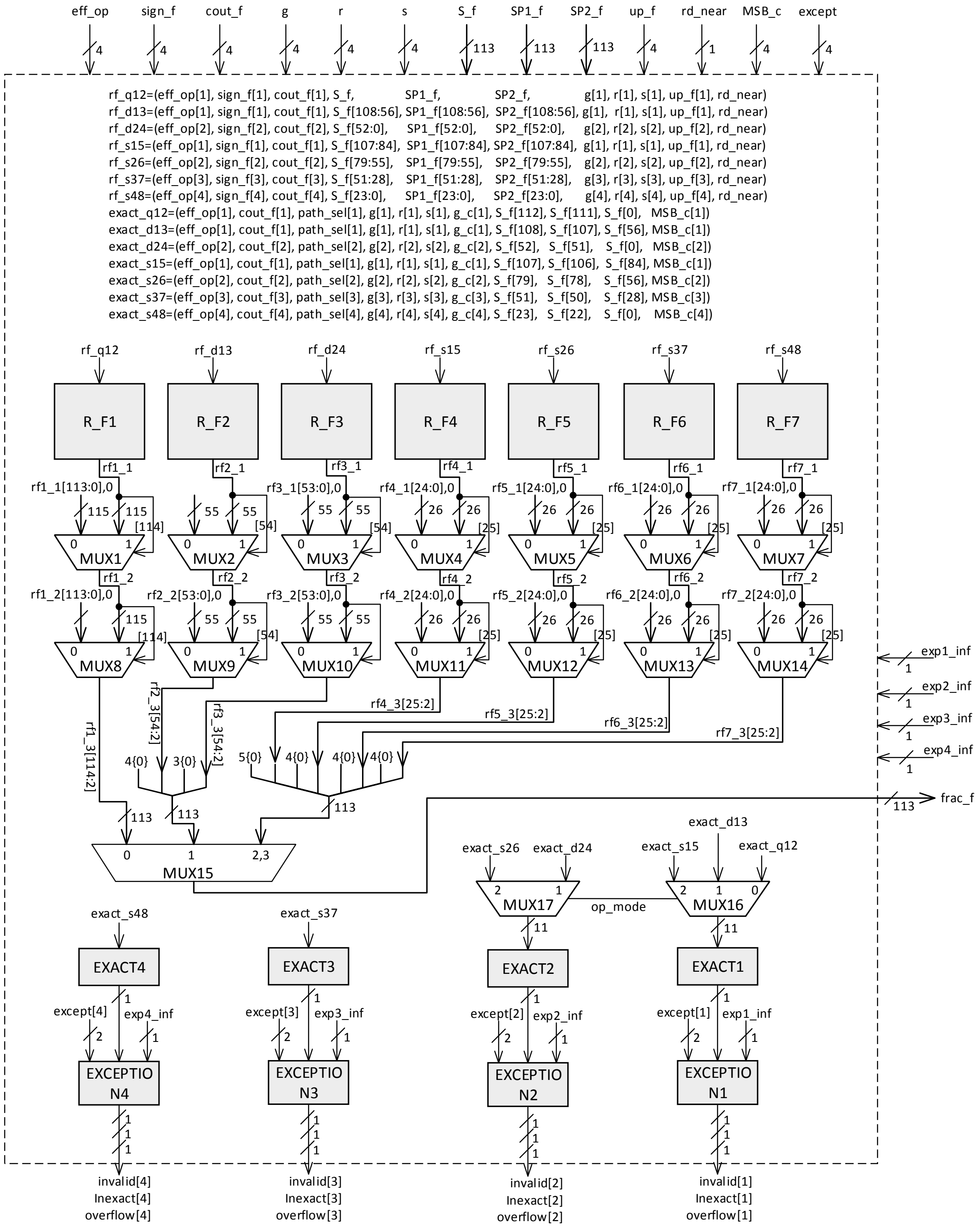

eff_op
sign_f
cout_f
g
r
s
S_f
SP1_f
SP2_f
up_f
rd_near
MSB_c
except
rf_q12=(eff_op[1], sign_f[1], cout_f[1], S_f, SP1_f, SP2_f, g[1], r[1], s[1], up_f[1], rd_near)
rf_d13=(eff_op[1], sign_f[1], cout_f[1], S_f[108:56], SP1_f[108:56], SP2_f[108:56], g[1], r[1], s[1], up_f[1], rd_near)
rf_d24=(eff_op[2], sign_f[2], cout_f[2], S_f[52:0], SP1_f[52:0], SP2_f[52:0], g[2], r[2], s[2], up_f[2], rd_near)
rf_s15=(eff_op[1], sign_f[1], cout_f[1], S_f[107:84], SP1_f[107:84], SP2_f[107:84], g[1], r[1], s[1], up_f[1], rd_near)
rf_s26=(eff_op[2], sign_f[2], cout_f[2], S_f[79:55], SP1_f[79:55], SP2_f[79:55], g[2], r[2], s[2], up_f[2], rd_near)
rf_s37=(eff_op[3], sign_f[3], cout_f[3], S_f[51:28], SP1_f[51:28], SP2_f[51:28], g[3], r[3], s[3], up_f[3], rd_near)
rf_s48=(eff_op[4], sign_f[4], cout_f[4], S_f[23:0], SP1_f[23:0], SP2_f[23:0], g[4], r[4], s[4], up_f[4], rd_near)
exact_q12=(eff_op[1], cout_f[1], path_sel[1], g[1], r[1], s[1], g_c[1], S_f[112], S_f[111], S_f[0], MSB_c[1])
exact_d13=(eff_op[1], cout_f[1], path_sel[1], g[1], r[1], s[1], g_c[1], S_f[108], S_f[107], S_f[56], MSB_c[1])
exact_d24=(eff_op[2], cout_f[2], path_sel[2], g[2], r[2], s[2], g_c[2], S_f[52], S_f[51], S_f[0], MSB_c[2])
exact_s15=(eff_op[1], cout_f[1], path_sel[1], g[1], r[1], s[1], g_c[1], S_f[107], S_f[106], S_f[84], MSB_c[1])
exact_s26=(eff_op[2], cout_f[2], path_sel[2], g[2], r[2], s[2], g_c[2], S_f[79], S_f[78], S_f[56], MSB_c[2])
exact_s37=(eff_op[3], cout_f[3], path_sel[3], g[3], r[3], s[3], g_c[3], S_f[51], S_f[50], S_f[28], MSB_c[3])
exact_s48=(eff_op[4], cout_f[4], path_sel[4], g[4], r[4], s[4], g_c[4], S_f[23], S_f[22], S_f[0], MSB_c[4])
R_F1
R_F2
R_F3
R_F4
R_F5
R_F6
R_F7
MUX1 … MUX14
MUX15
MUX16
MUX17
op_mode
exp1_inf
exp2_inf
exp3_inf
exp4_inf
frac_f
EXACT1
EXACT2
EXACT3
EXACT4
EXCEPTION1
EXCEPTION2
EXCEPTION3
EXCEPTION4
invalid[1] Inexact[1] overflow[1]
invalid[2] Inexact[2] overflow[2]
invalid[3] Inexact[3] overflow[3]
invalid[4] Inexact[4] overflow[4]

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

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

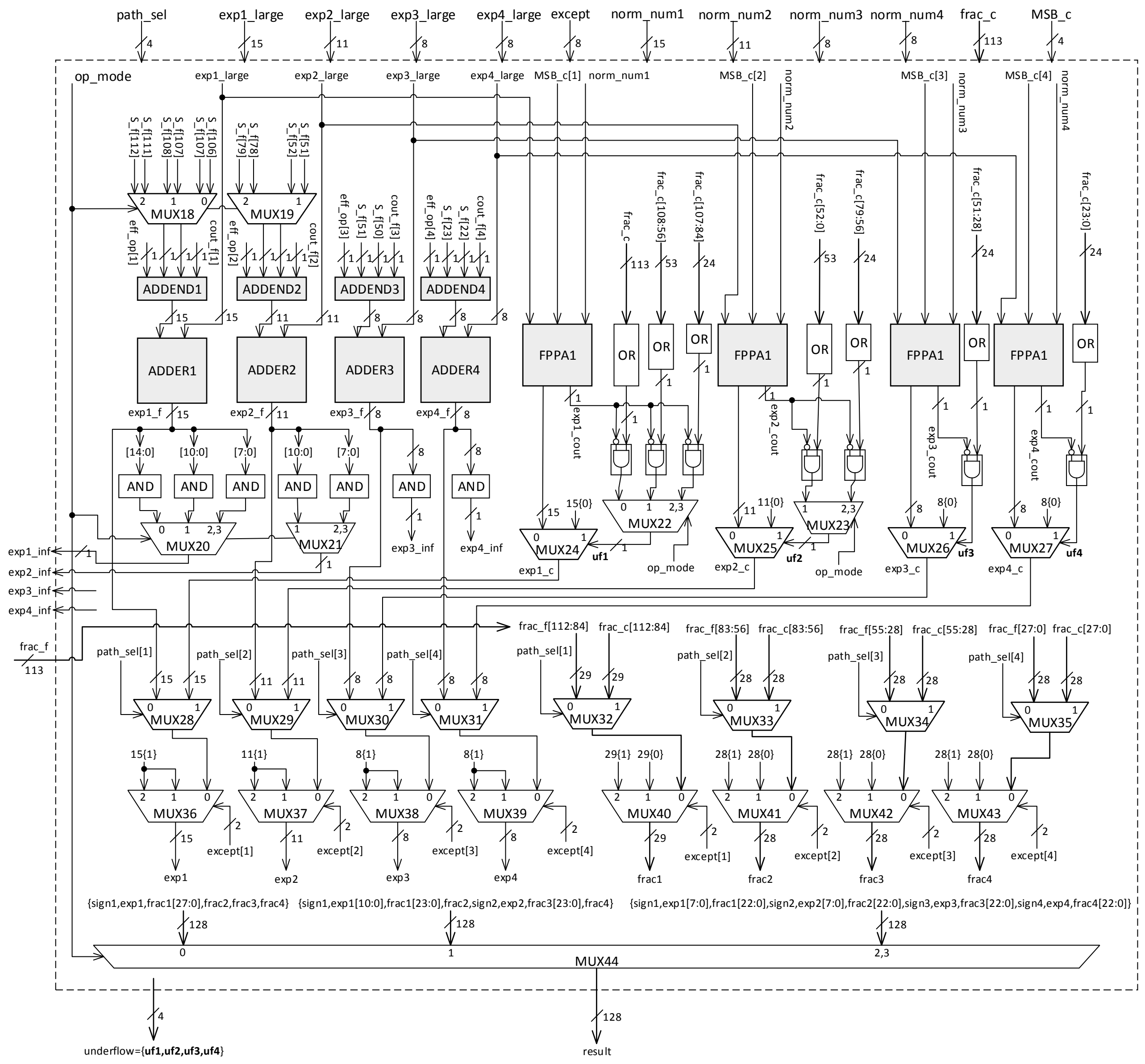


Fig. 14. Exponent adjustment of both FAR and CLOSE path in Stage 3

final 128-bit *result* is multiplexed through *MUX42* according to the precision mode.

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 pipeline 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's Topographic technique, which can obtain the best

Table 4. Testing vector patterns including exceptions and corners

operand	exception and corners, 100 random vectors for normal number in each case									40000 random vectors
A	*NaN*	*NaN*	*INF*	*INF*	*NaN*	*normal*	*INF*	*normal*	*0*	*equal*	*normal*
B	*NaN*	*INF*	*NaN*	*INF*	*normal*	*NaN*	*normal*	*INF*	*0*	*equal*	*normal*

Table 5. Latency of the proposed designs

LATENNCY			Single	Double	Quad	Quad (Dual-Mode)	Quad (Tri-Mode)	[27]Quad (Dual-Mode) (0.11μm)	[31]Double (Dual-Mode) (0.18μm)
COMB		(ns)	1.34(74%)	1.54(85%)	1.81(100%)	1.90(105%)	2.00(110%)	--	7.84
		FO4	41	47	55	58	62	--	87
PIPE	1		0.66(0.67)	0.78(0.8)	.0.89(0.9)	0.92(0.95)	1.09(1.10)	0.74	--
	2		0.50(0.67)	0.62(0.8)	0.71(0.9)	0.91(0.95)	1.06(1.10)	1.19	--
	3		0.65(0.67)	0.74(0.8)	0.84(0.9)	0.92(0.95)	1.08(1.10)	1.59	--
	worst	(ns)	0.66(74%)	0.78(88%)	0.89(100%)	0.92(113%)	1.09(122%)	1.59	2.56(2.56*4)
		FO4	20.6	24.6	27.7	29.2	33.8	31.8	28.4(4 stages)
PIPE (RR)	1,2,3		0.58(0.6)	0.69(0.7)	0.78(0.8)	0.83(0.85)	0.90(0.93)	--	--
	worst	(ns)	0.58(75%)	0.69(88%)	0.78(100%)	0.83(106%)	0.90(116%)	--	--
		FO4	17.8	21.5	24.0	26.2	28.6	--	--

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 form, COMB denotes the design is in combinational form and PIPE(RR) means that the design in pipeline form is synthesized with **R**egister **R**epositioning 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 μm^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 adder is 128 bits, and the dual-mode adder of [31] is 64 bits. By applying register repositioning technique, our designed single-mode, dual-mode and triple-mode quadruple precision adder can run at 1250MHz, 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 μm^2 respectively. After register repositioning, the area changes to 85013 μm^2. The gate number of our designed 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 6. Area estimation of the proposed designs

AREA		Single	Double	Quad	Quad (Dual-Mode)	Quad (Tri-Mode)	[27]Quad (Dual-Mode) ($0.11\mu m$)	[31]Double (Dual-Mode) ($0.18\mu m$)
CO MB	Area(μm^2)	10639	21885	40936	55628	66916	--	164000*2
	Gate Count	7388	15198	28428	38631	46469	--	10288*2
PIPE	Area(μm^2)	12921	24243	50268	63528	71290	357399	172000*2
	Gate Count	8793	16835	34908	44323	49507	$86663\#=\frac{124794}{1.44}$	10794*2
PIPE (RR)	Area(μm^2)	14618	29284	61293	68895	85013	--	--
	Gate Count	10151	20336	42565	47844	59037	--	--

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

POWER		Single	Double	Quad	Quad (Dual-Mode)	Quad (Tri-Mode)	[27]Quad (Dual-Mode) ($0.11\mu m$)	[31]Double (Dual-Mode) ($0.18\mu m$)
CO MB	combinational	2.53	4.22	7.12	8.64	9.93	--	--
	registers	0	0	0	0	0	--	--
	total	2.53	4.22	7.12	8.64	9.93	--	9.76
PIPE	combinational	3.29	5.19	8.72	10.88	11.73	--	--
	registers	7.59	12.46	22.16	22.09	21.13	--	--
	total	10.88	17.64	30.88	32.97	32.86	--	48.38
PIPE (RR)	combinational	3.3126	5.07	10.68	11.48	12.72	--	--
	registers	9.7653	20.17	28.57	30.56	29.36	--	--
	total	13.08	25.24	39.25	42.03	42.08	--	--

two times of that of lower precision adder. For example, when the circuit is pure combinational, the area of 32-bit adder is 10639μm^2 while the area of 64-bit adder is 21885μm^2; when the circuit is pipelined, the area of 64-bit adder is 24243μm^2 while the area of 128-bit adder is 50268μm^2. 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 number of dual-mode adder we designed is 44323 which is greater than 2*10794. So the two-path algorithm is not suitable for area-efficient design. The gate count of our proposed dual-mode quadruple precision adder is only 51.1% of that in [27], which proves that the proposed architecture is more area-efficient than [27]. Compared to a combination of four single precision, two double precision and one quadruple precision adder, the area saving of the proposed triple-mode adder is 47.4%, 52.6% and 52.3% in combinational, pipeline and register repositioning form respectively. The area saving is computed using the following equation:

$$100\% - [(4*S+2*D+Q)-T]/(4*S+2*D+Q)*100\%,$$

S, D, Q and T are the area of single-mode single, double, quadruple precision and triple-mode adder.

The power of the triple-mode adder in combinational circuit is 9.93mW, as shown in Table 7. For pipelining and register repositioning circuit, the power of the triple-mode adder is 32.86mW and 42.08mW. As seen in Table 7, the registers of pipelined and register-repositioning circuits consumes 21.13mW and 29.36mV respectively. The aggressive rising in power is mainly contributed by registers which dissipate much internal power even when clock is deactivated.

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:

[1] (2012, Jan.), Avoiding AVX-SSE transition penalties, Intel, [Online]. Available: http://software.intel.com/en-us/articles

[2] (2014 Sep.) Intel64 and IA-32 architectures optimization reference manual, Intel, [Online]. Available: http://www.intel.com/content/www/us/en/

[3] C. L. Yang and B. Sano, "Exploiting parallelism in geometry processing with general purpose processors and floating-point SIMD instruction," *IEEE Transactions on Computers*, vol. 49, no. 9, 2000, pp. 934-946

[4] Naoki NISHIKAWA and Keisuke IWAI, "Throughput and Power Efficiency Evaluation of Block Ciphers on Kepler and GCN GPUs Using Micro-Benchmark Analysis," *IEICE Transactions on Information and Systems*, vol. E97-D, no. 6, 2014, pp. 1506-1515

[5] Li Rongchun, Dou Yong, "Efficient parallel implementation of three-point Viterbi decoding algorithm on CPU, GPU and FPGA," *Concurrency and Computation-Practice & Experience*, vol. 26, no. 3, 2014, pp. 821-840

[6] M.Ferreira, N. Roma and Luis MS Russo, "Cache Oblivious parallel SIMD Viterbi decoding for sequence search in HMMER," *BMC Bioinformatics*, vol. 15:165, 2014

[7] Juan M. Cebrian, Lasse Natvig and J. C. Meyer, "Performance and energy impact of parallelization and vectorization techniques in modern microprocessors," *Computing*, vol. 96, no. 12, 2014, pp. 1179-1193

[8] A. Akkas, "Instruction Set Enhancements for Reliable Computations," Ph.D. dissertation, Lehigh University, 2001.

[9] D.H. Bailey, R. Barrio, J.M. Borwein, "High-precision computation: Mathematical physics and dynamics," *Applied Mathematics and Computation*, vol. 218, no. 20, 2012, pp. 10106-10121

[10] G. Howell, G.A. Geist, "Necessity of high precision arithmetic for large-scale computations," in Proc. NPSC, 1995 , pp. 219–222. Jun. 2012.

[11] IEEE Standard for Floating-Point Arithmetic, ANSI/IEEE Standard 754-2008, Aug. 29, 2008.

[12] E. Schwarz, R. Smith, C. Krygowski, "The S/390 G5 floating point unit supporting hex and binary architecture," *14th IEEE Symposium on Computer Arithmetic*, 1999, pp. 258-265.

[13] S. Oberman, "Design Issues in High-Performance Floating-Point Arithmetic Units," Ph.D. dissertation, Dept. Elect. Eng., Stanford University, Stanford, 1996.

[14] A. Beaumont-Smith, N. Burgess, " Reduced latency IEEE floating-point standard adder architectures," *14th IEEE Symposium on Computer Arithmetic*, 1999, pp. 35–43.

[15] P. M. Seidel, G. Even, "Delay-optimized implementation of IEEE floating-point addition," IEEE Transactions on Computers, vol. 53, no. 2, 2004, pp. 97–113

[16] M. Farmwald, "On the Design of High-Performance Digital Arithmetic Units," Ph.D. Dissertation, Stanford University, Stanford, 1981.

[17] S. Oberman, H. Al-Twaijry, M. Flynn, "A SNAP project: design of floating-point arithmetic units," *13th IEEE Symposium on Computer Arithmetic*, 1997, pp. 156-165.

[18] J. Bruguera and T. Lang, "Rounding in floating-point addition using a compound adder," Techinical report, University of Santiago de Compostela, 2000.

[19] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," IEEE

Transactions on Computers, vol. C-22, no. 8, 1973, pp. 786–793

[20] N. Bruguera, "The flagged prefix adder for dual addition," *Proceeding of SPIE - The International Society for Optical Engineering*, 1998, pp. 567-575

[21] G. Oklobdzija, "An Alg orithmic and Novel Design of a Leading Zero Detector Circuit: Comparison with Logic Synthesis," *IEEE Transactions on VLSI Systems*, vol. 2, no. 1, 1994, pp. 124-128

[22] V. Oklobdzija, "Comment on "Leading-zero anticipatory logic for high-speed floating point addition"," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 2, 1997, pp. 292–293

[23] H.Suzuki and H. Morinaka, "Leading-Zero Anticipatory Logic for High-Speed Floatin g Point Addition," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 8, 1996, pp. 1157-1164

[24] G. Dimitrakopoulos, K. Galanopoulos, C. Mavrokefalidis and D. Nikolos, "Low -Power Leading-Zero Counting and Anticipation Logic for High-Speed Floating Point Units," *IEEE Transactions on VLSI Systems*, vol. 16, no. 7 , 2008, pp. 837-850

[25] J. D. Brugu era and T. Lang, "Leading-one prediction with concurrent position correction," IEEE Transactions on Com puters, vol. 48, no. 10, 1999, pp.1083–1097

[26] A. Akkas, "Dual-m ode quadruple precision floating-point adder," *9th Euromicro Conference on Digital System Design*, Cavtat, CROATIA, 2006, pp. 211-220

[27] Akkas, "Dual-mode floating-point adder architectures," *Journal of Systems Architecture*, vol. 54, no. 12, 2008, pp. 1129-1142

[28] Akkas and M. Schulte, "Dual-mode floating-point multiplier architectures with parallel operations," *Journal of Systems Architecture*, vol. 52, no. 10, 2006, pp. 549-562

[29] Akkas and M. J. Schulte, "A Quadruple Precision and Dual Double Precision Floating-Point Multiplier," *Euromicro Symposium on Digital System Design*, 2003, pp. 76-81.

[30] Isseven and A. Akka s, "A Dual- mode quadruple precision floating-point divider," *40th Asilomar Conference on Signals, Systems and Computers*, 2006, pp. 1697-1701.

[31] M. K. Jaiswal and Ray C.C. Cheung, "Unified Architecture for Double /Two-Parallel Single Precision Floating Poi nt Adder," *IEEE Transactions on Circuits and Systems II-Express Briefs*, vol. 61, no. 7, 2014, pp. 521-525

[32] D. Tan, C. E. Lemonds and M . J. Schulte, "Low-Power Multiple-Precision Iterative Floating-Point Multiplier with SIMD Support ," *IEEE Transactions on Computers*, vol. 58, n o. 2, 2009, pp. 175-187

[33] M. K. Jaiswal and Ray C.C. C heung, "ConFigurable Architecture for Double/Two-Parallel Single Precision Floating Point Division," *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, 2014, pp. 332-337.

[34] K. Manolopoulos, D. Reisis, "An Efficient Multiple Precision Floating-Point Multiplier," *18th IEEE International Conference on Electronics, Circuits and Systems*, 2011, pp. 153-156.

[35] A. Baluni and F. Merchant, "A Full y Pipelined Modular Multiple Precision Floating Point Multiplier With Vector Support," *International Symposium on Electronic System Design*, 2011, pp. 45-50.

[36] Libo Huang and Li Shen, "A Ne w Architecture for Multiple Precision Floating-Point Multiply-Add Fused Unit Design," *18th IEEE Symposium on Computer Arithmetic*, 2007, pp. 69-76.

[37] K. Manolopoulos and D. Reisis, "An Efficient Dual-Mode Floating-Point Multiply-Add Fused Unit," in Proc. *17th IEEE International Conference on Electronics, Circuits and Systems*, 2010, pp. 5-8.

[38] M. Gok and M. M. Ozbilen, "Multi-functional floating-point MAF desig ns with d ot product support," *Microelectronics Journal*, vol. 39, pp. 30-43, 2007.

[39] A. Verma and A. K. Verma, "Hy brid LZA: A Near Optimal Implementation of the Leading Zero Anticipator," *14th Asia and South Pacific Design Automation Conference*, 2009, pp. 203-209.

[40] N. K. Reddy, M. C. Sekhar, "A Nov el Low Power Error Detection Logic for I nexact Leading Zero Anticipator in Floating Point Units," *27th International Conference on VLSI Design*, 2014, pp. 128-132.