30TH IEEE INTERNATIONAL SYMPOSIUM ON COMPUTER ARITHMETIC 2023

ENHANCED FLOATING-POINT MULTIPLY-ADD WITH FULL DENORMAL SUPPORT

Jongwook Sohn, David K. Dean, Eric Quintana and Wing Shek Wong

9/6/2023

Introduction

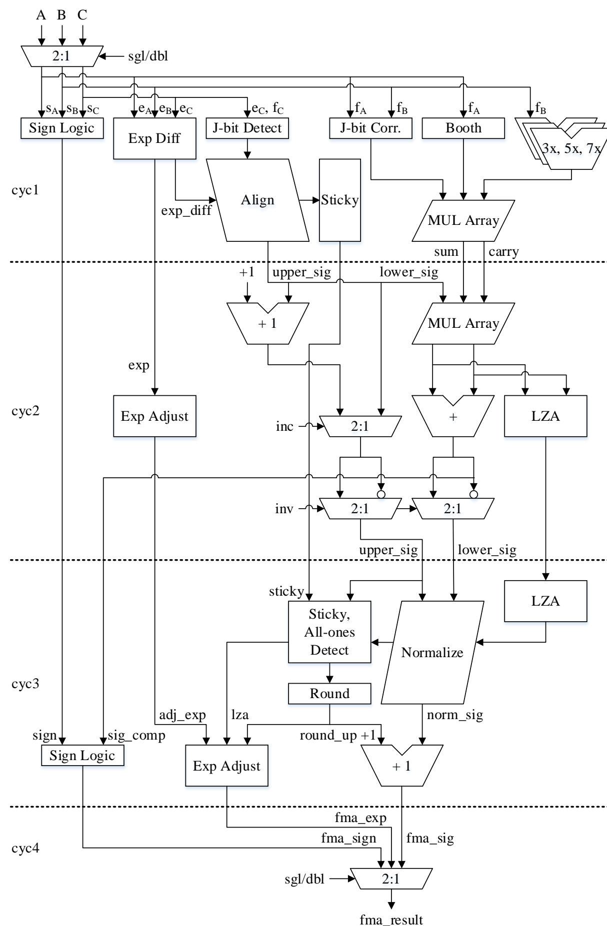
Denormal (Subnormal) Numbers?

- Exponent == 0

```
Ex)  sign   exponent           significand
      0     00000001   10000000000000000000000 = +2^-126 * 1.100…0 (normal)
      0     00000000   10000000000000000000000 = +2^-126 * 0.100…0 (denormal)
```

- $E_{denormal}$ == E_{min}, 1-bit adjustment is needed
- J-bit – implicit 1-bit above the MSB of significands
- Hardware denormal support – no exception handler

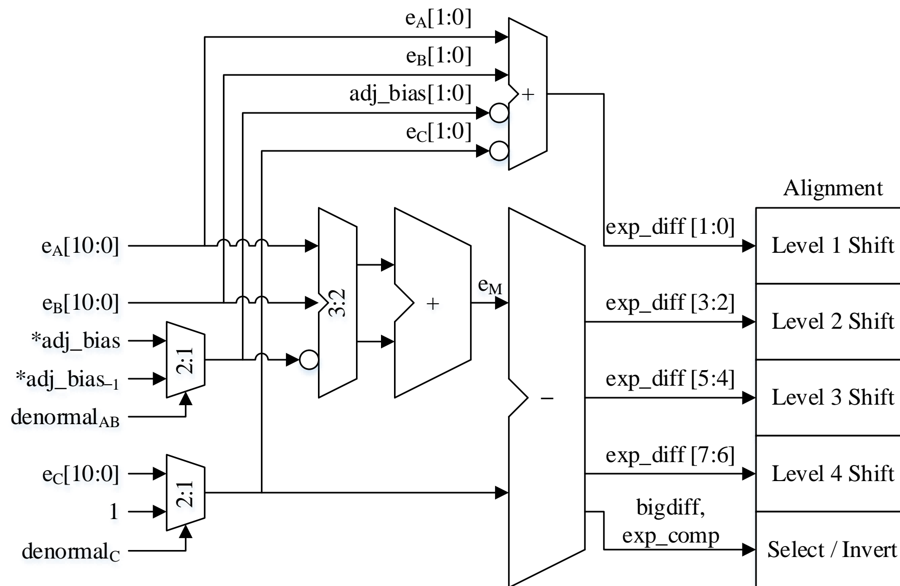
Enhanced FMA

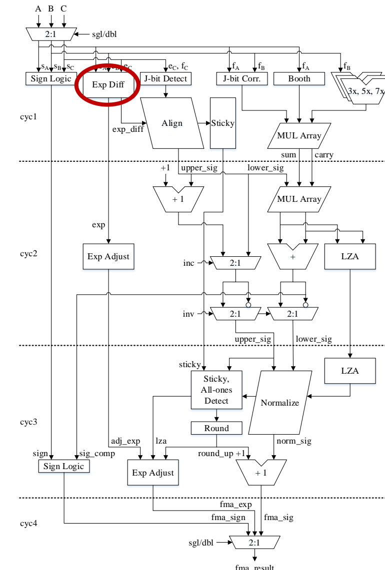


Exponent Difference

Adjusted Exponent Difference

- $e_M = e_A + e_B - adj_bias$
- $exp_diff = e_M - e_C$
- $exp_comp = e_M > e_C$
- $bigdiff = exp_diff \leq 0$ or $exp_diff \geq maxdiff$





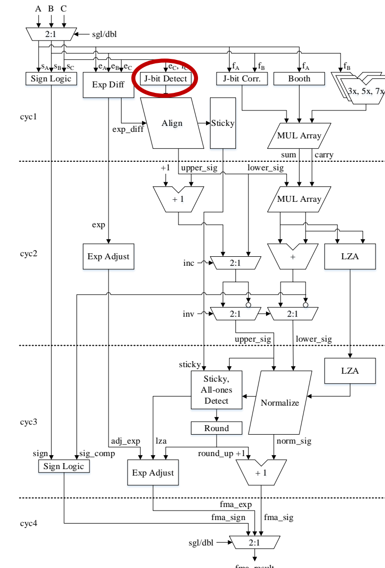
J-bit Detection

Denormal Support

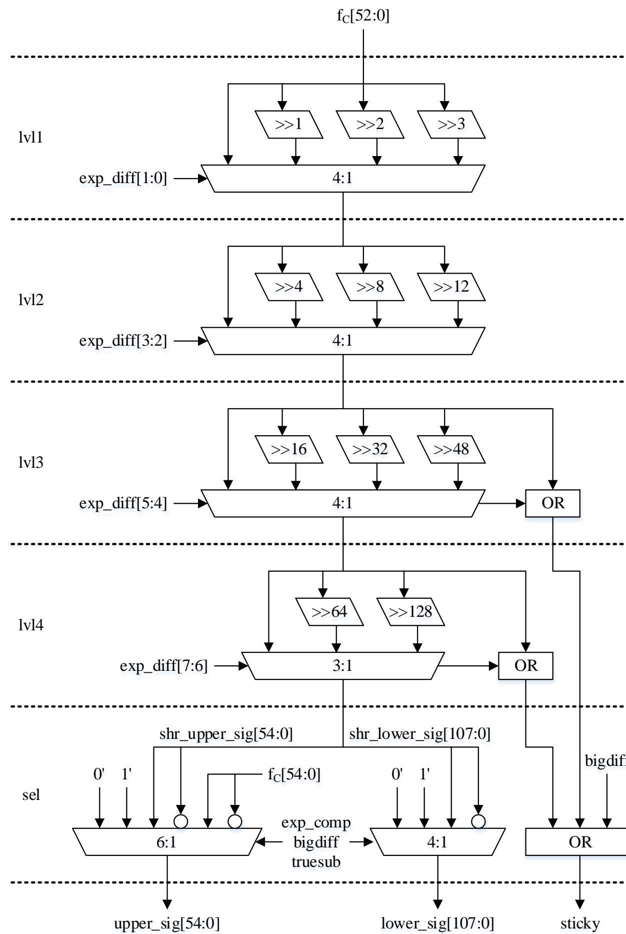
- J-bit detection in parallel with exp_diff
    - J-bit = (exp ≠ 0)
- 1-bit denormal adjustment in exp_diff
    - exp_diff is adjusted by ±1

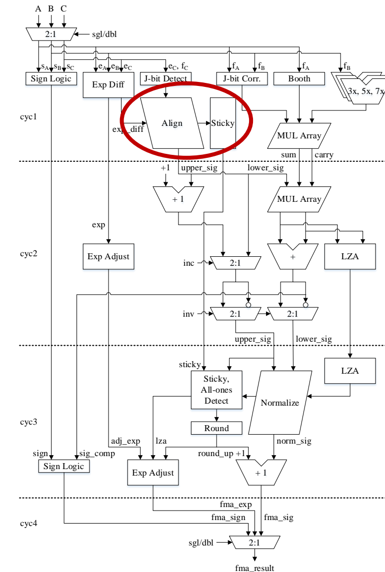
Ex) $e_A * e_B$ = 1 and e_C = 0, exp_diff = 1 → adjusted to 0

```
  1.000…0
+ 0.100…0 -> 0.010…0, need back to 0.100…0
----------
  1.100…0
```



Alignment

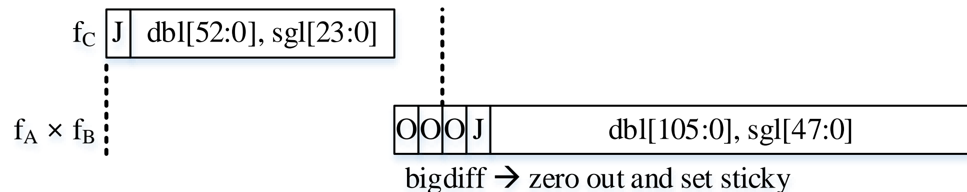




Alignment

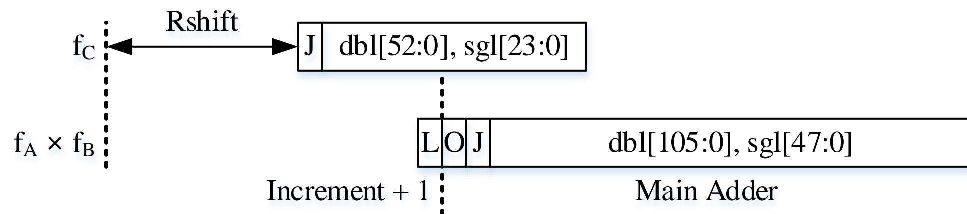
Case 1) No Shift

- exp_diff ≤ 0

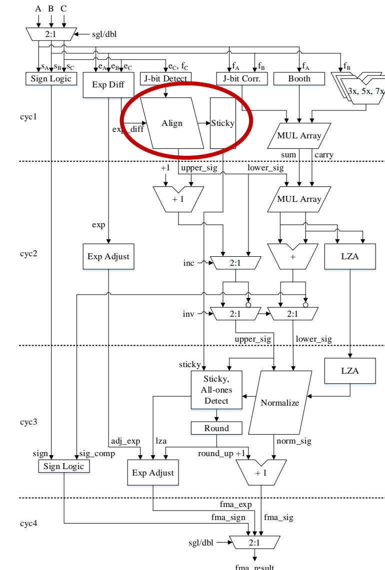


Case 2) Small Right Shift

- 0 < exp_diff ≤ *adj



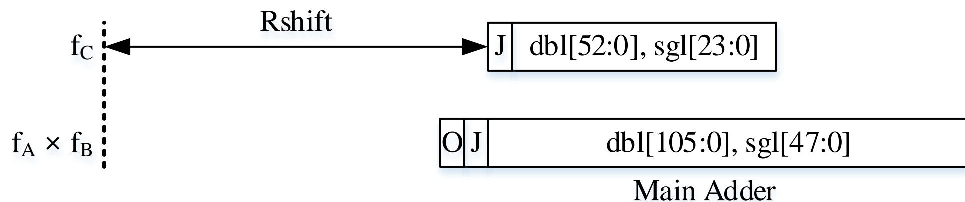
*adj = 56 (dbl), 27 (sgl)



Alignment

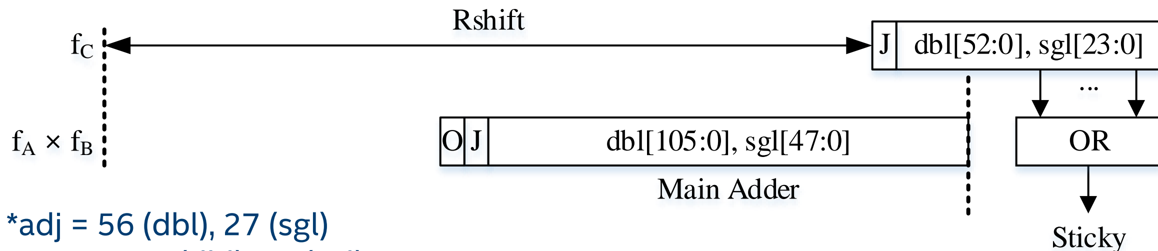
Case 3) Medium Right Shift

- *adj < exp_diff ≤ *max

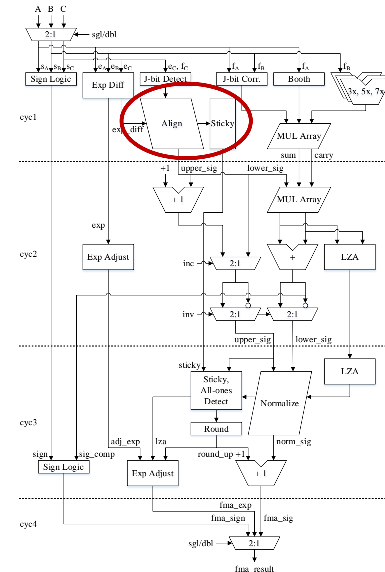


Case 4) Big Right Shift

- exp_diff > *max



*adj = 56 (dbl), 27 (sgl)
*max = 109 (dbl), 51 (sgl)



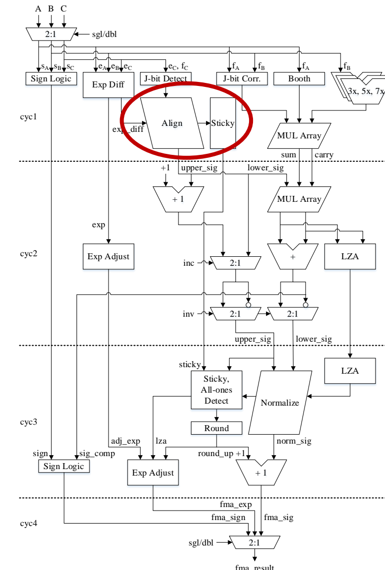
Alignment

Select Upper bits → to Incrementor

bigdiff	$e_M > e_C$	truesub	upper_sig
0	-	0	aligned f_C
0	-	1	aligned & inverted f_C
1	0	0	f_C
1	0	1	inverted f_C
1	1	0	‘0
1	1	1	‘1

Select Lower bits → to Multiplier

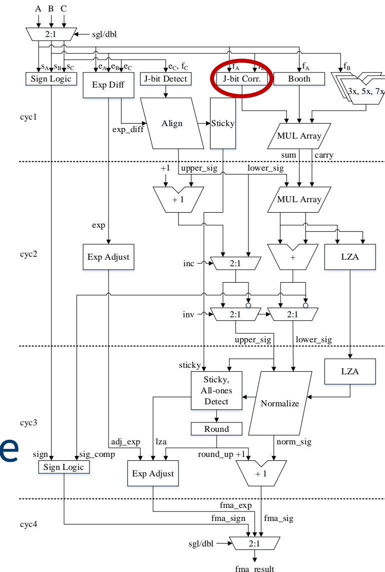
bigdiff	truesub	lower_sig
0	0	aligned f_C
0	1	aligned & inverted f_C
1	0	‘0
1	1	‘1



J-bit Correction

J-bit Correction in CSA Tree

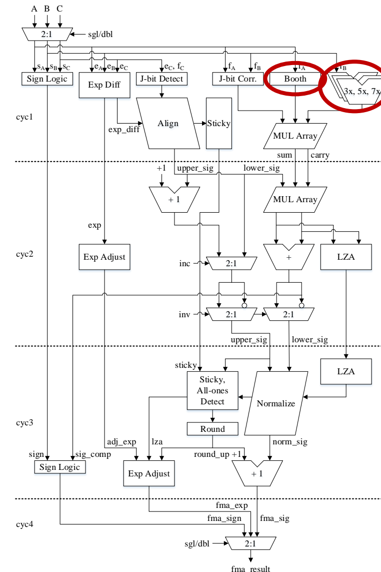
- No J-bit detection
  - f_A and f_B to multiplier with no delay
- Assume both J-bits are 1, then subtract 1 J-bit correction line in CSA tree
  - If J_A = 0, then subtract f_B
  - If J_B = 0, then subtract f_A
  - Don’t care if both J-bits are 0
  - A more partial product and a few bits for 2’s complement



Radix-16 Booth Encoding

Radix-16 Pre-computations (1x – 8x)

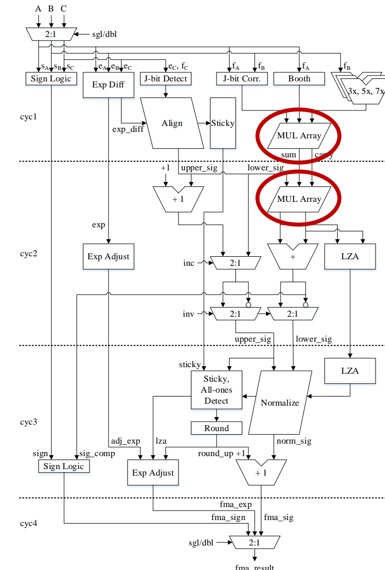
- 3 adders for pre-computations
  - $1x = f_B$
  - $2x = f_B << 1$
  - $3x = 1x + 2x$ → need an adder
  - $4x = f_B << 2$
  - $5x = 1x + 4x$ → need an adder
  - $6x = 3x << 1$
  - $7x = 8x - 1x$ → need an adder
  - $8x = f_B << 3$

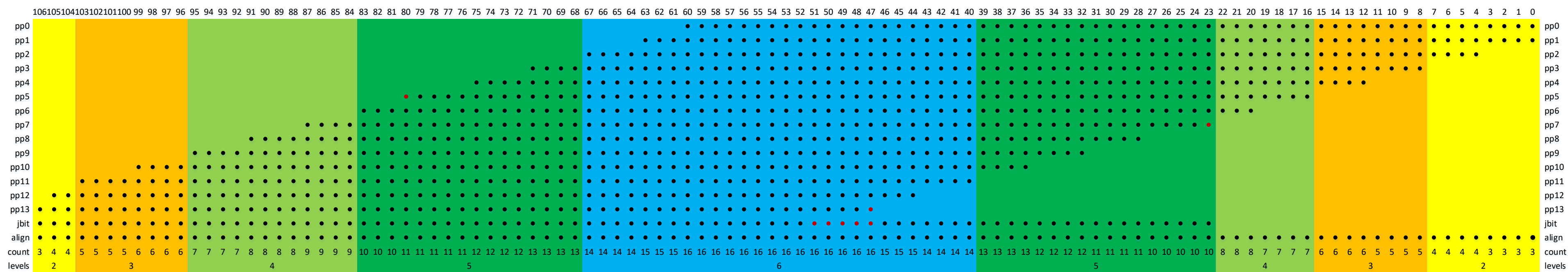


Multiplier

CSA Tree

- 6 levels of 3:2 CSA tree
  - Level 1 – 4 are in the 1st cycle
  - Level 5 – 6 are in the 2nd cycle
- 14 + 2 partial products
  - J-bit correction line and 2's complement bits are added
  - Aligned f_C is added

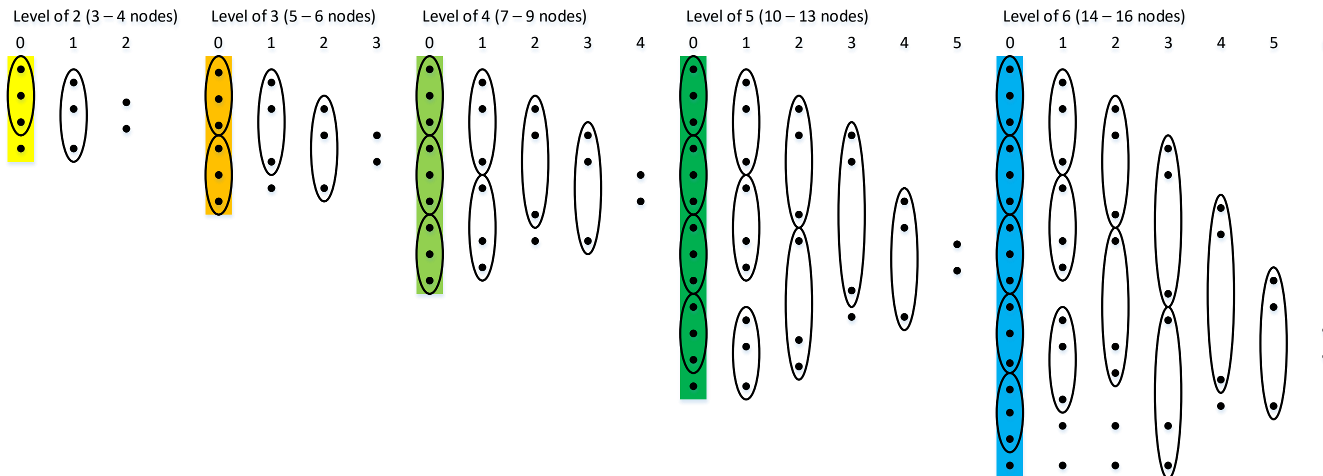


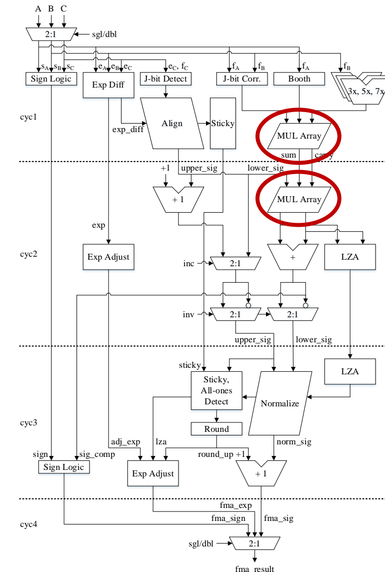


Multiplier

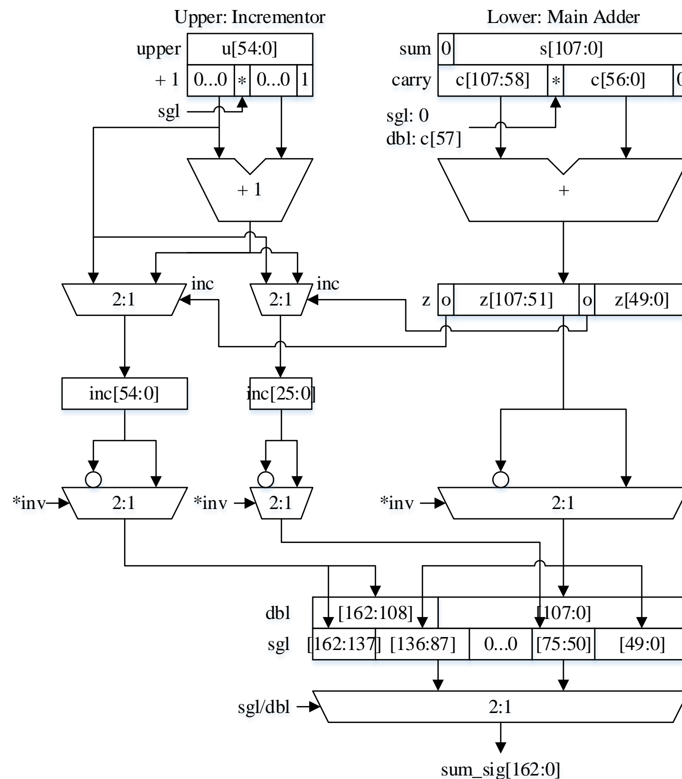
CSA Tree

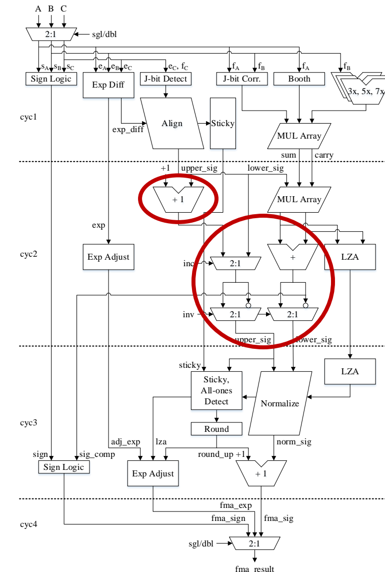
- Levels of 3:2 CSAs depending on # of nodes to add





Main Adder





Main Adder

Case 1) X + Y, Trueadd – Nothing to do

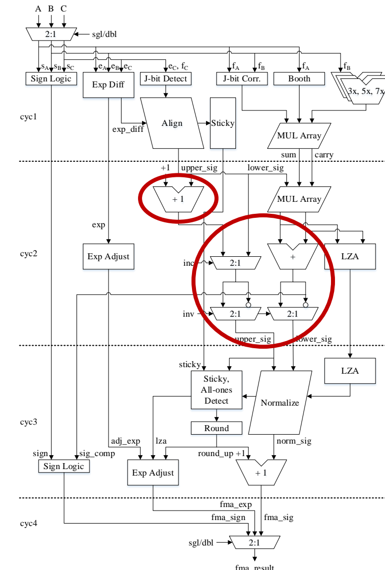
```
Ex)      X     1.0110
       + Y     1.0001
     ---------------
         Z = 10.0111
```

Case 2) X – Y (X < Y), Truesub negative – Inversion is required

```
Ex)      X     1.0001
       + ~Y    0.1001
     ----------------
         Z = 01.1010 -> 0.0101
```

Case 3) X – Y (X > Y), Truesub positive – 2's complement is required → merged with rounding

```
Ex)      X     1.0110
       + ~Y    0.1110
     ----------------
         Z = 10.0100 -> 0.0101
```



LZA

Leading Zero Anticipator (LZA)

- $sum = \{s_n, s_{n-1}, \ldots, s_0\}, carry = \{c_n, c_{n-1}, \ldots, c_0\}$
- $g_i = s_i \wedge c_i$
- $z_i = \overline{s_i \vee c_i}$
- $f^{pos}{}_i = (g_i \vee z_i) \wedge \overline{z_{i-1}}$
- $f^{neg}{}_i = (g_i \vee z_i) \wedge \overline{g_{i-1}}$
    - Selected based on the inversion

Ex) S 1.1111101

+ C 0.0000110

----

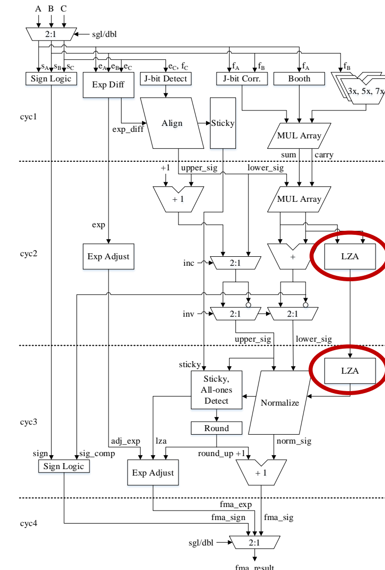
g_i = 0.0000100

z_i = 0.0000000

$g_i \vee z_i$ = 0.0000100

$\overline{z_{i-1}}$ = 1.111110

f_i = 0.000010 -> 5-bit left shift for normalization

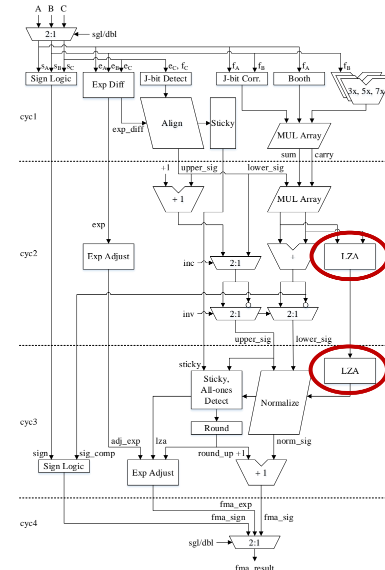


Modified LZA

Masking Underflow

- Mask bit stops normalization shift when exp < 0
  - $m_1 = m_0{}^{64} m_{64}{}^{64}$
  - $m_2 = m_0{}^{16}\, m_{16}{}^{16} m_{32}{}^{16}\, m_{48}{}^{16}$
  - $m_3 = m_0{}^{4} m_4{}^{4} m_8{}^{4} m_{12}{}^{4} \ldots m_8{}^{4} m_{12}{}^{4}$
  - $m_4 = m_0 m_1 m_2 m_3 \ldots m_2 m_3$
    - ➢ $m_k{}^n$ is set if exp has k bit and repeated n times, 2-bit decoders
  - $m = m_1 \wedge m_2 \wedge m_3 \wedge m_4 \wedge (exp < 128)$
  - $f_i = (g_i \vee z_i) \wedge \overline{z_{i-1}} \vee m_i$

```
Ex) eA*eB = 3 and eC = 3,
    100.0…1
  - 100.0…0
  ----------
    000.0…1 -> 0.0…100, 2-bit left shift
```



Normalization

sum_sig[162:0]

lvl1

<<64

<<128

lza_lvl1

3:1

OR

lvl2

<<16

<<32

<<48

lza_lvl2

4:1

OR

lvl3

<<4

<<8

<<12

lza_lvl3

4:1

OR

lvl4

[53:50]

<<1

<<2

<<3

lza_lvl4

4:1

4:1

OR

post

>>1

align_sticky

post_norm

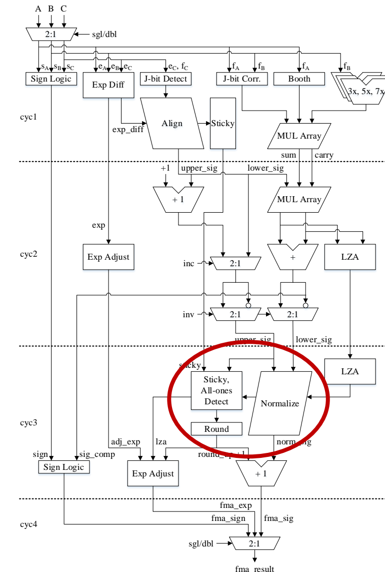
2:1

OR

norm_sig[52:0]

sticky

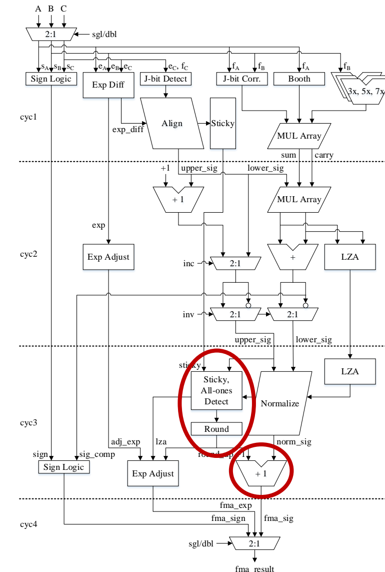
allones



Rounding

Rounding Control

- 4 rounding modes are reduced to 2
  - $RNE \wedge Guard \wedge (LSB \vee Sticky)$
  - $*RINF \wedge (Guard \vee Sticky)$
    - $*RINF = (RINF- \wedge \overline{sign}) \vee (RINF+ \wedge sign)$
    - RZ is omitted by using AOI
- Force round up for 2’s complement
  - 2’s complement is propagated only if the bits below LSBs are all-ones
- Significand overflow after rounding
  - $ov_rndup = allones \wedge roundup$ → eliminates the re-normalization



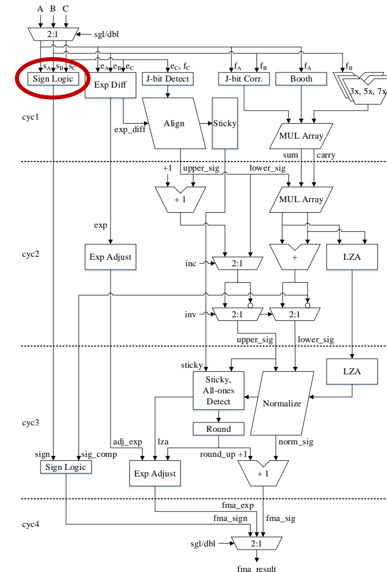
Sign Logic

Effective Sign

- $s_{Meff} = s_A \oplus s_B \oplus is_neg$
- $s_{Ceff} = s_C \oplus is_sub$

Sign Becomes 1 (Negative), otherwise 0 (Positive)

- $s_{Meff} = 1, s_{Ceff} = 1$
- $s_{Meff} = 1, s_{Ceff} = 0, and\ A*B > C$
- $s_{Meff} = 0, s_{Ceff} = 1, and\ A*B < C$
- $s_{Meff} \neq s_{Ceff}, A*B = C$, and round_to_-infinity



Exponent Logic

Intermediate FMA Exp

- $e_M = e_A + e_B - adj_bias$
- $(e_M > e_C) \rightarrow exp = e_M$

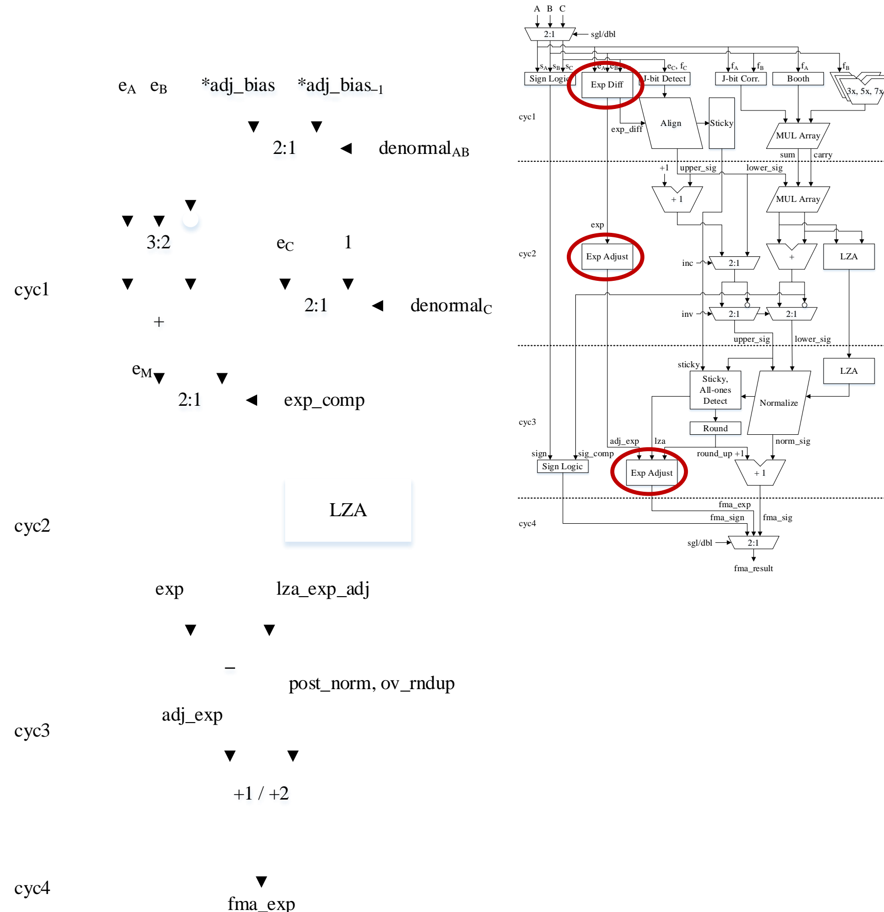
  $(e_M < e_C) \rightarrow exp = e_C$

Adjusted FMA Exp

- $adj_exp = exp - lza$
- $\overline{(post_norm \vee ov_rndup)} \rightarrow fma_exp = adj_exp + 0$

  $(post_norm \oplus ov_rndup) \rightarrow fma_exp = adj_exp + 1$

  $(post_norm \wedge ov_rndup \rightarrow fma_exp = adj_exp + 2$



*adj_bias = dbl: 0x3C7, sgl: 0x64

$^*adj_bias_{-1}$ = dbl: 0x3C6, sgl: 0x63

Results (Latency)

	PowerPC FMA	Lang's FMA	Proposed FMA
Multiplier (gate levels)	Radix-4 Booth (6)	Radix-4 Booth (6)	Radix-16 Booth & 53-bit Adder (10)
	8 Levels of 3:2 CSAs (12)	8 Levels of 3:2 CSAs (12)	6 Levels of 3:2 CSAs (9)
Main Adder (gate levels)	3:2 CSA (2)	3:2 CSA (2)	No 3:2 CSA (0)
	106-bit Adder (12)	Part of 162-bit Adder (10)	106-bit Adder (12)
	106-bit Incrementor (10)	No Complement (0)	No Complement (0)
Normalization (gate levels)	162-bit Shifter (8)	162-bit Shifter (8)	162-bit Shifter (8)
Rounding (gate levels)	53-bit Incrementor (8)	Rest of 162-bit Adder & Rounding (13)	53-bit Incrementor (8)
	1-bit Shifter (2)	1-bit Shifter (2)	No Shifter (0)
Total Gate Levels	60	53	47, -20%/-10%

Results (Area)

	PowerPC FMA	Lang's FMA	Proposed FMA
Exp Diff & Alignment (gate count)	11-bit Adder × 2 (200)	11-bit Adder × 2 (200)	11-bit Adder × 2 (200)
	162-bit Shifter (2,400)	162-bit Shifter (2,400)	162-bit Shifter (2,400)
	Sticky (100)	Sticky (100)	Sticky (100)
Multiplier (gate count)	Radix-4 Booth (200)	Radix-4 Booth (200)	Radix-16 Booth (400) 53-bit Adder × 3 (1,500)
	1,500-bit 3:2 CSAs (9,000)	1,500-bit 3:2 CSAs (9,000)	900-bit 3:2 CSAs (5,400)
Main Adder (gate count)	106-bit 3:2 CSAs (600)	106-bit 3:2 CSAs (600)	No 3:2 CSA (0)
	106-bit Adder (1,200)	Part of 162-bit Adder × 2 (1,200)	106-bit Adder (1,200)
	56-bit Incrementor (200)	No Upper Incrementor (0)	56-bit Incrementor (200)
	106-bit Incrementor (400)	No Complement (0)	No Complement (0)
Normalization (gate count)	162-bit Shifter (2,400)	162-bit Shifter × 2 (4,800)	162-bit Shifter (2,400) Sticky & All-ones Detection (200)
Rounding (gate count)	56-bit Incrementor (200)	Rest of 162-bit Adder × 2 (1,200) Rounding (1,400)	56-bit Incrementor (200)
	1-bit Shifter (100)	1-bit Shifter (100)	No Shifter (0)
Total Gate Count	19,000	23,200	16,600, -10%/-30%

References

[1] IEEE Standard for Floating-Point Arithmetic, ANSI/IEEE Standard 754-2008, New York: IEEE, Inc., 2008.

[2] How 12th Gen's Intel® Core™ Hybrid Technology Works, https://www.intel.com/content/www/us/en/gaming/resources/how-hybrid-design-works.html, Intel Corp., 2022.

[3] E. M. Schwarz, M. Schmookler, S. Dao Trong, "FPU Implementations with Denormalized Numbers," IEEE Trans. on Computers, vol.54, no. 7, pp. 825-836, July 2005.

[4] D. R. Lutz, "Fused multiply-add microarchitecture comprising separate early-normalizing multiply and add pipelines," Proc. 20th IEEE Symp. Computer Arithmetic, pp. 123-128, July 2011.

[5] J. Sohn, D. K. Dean, E. Quintana and W. S. Wong, "Enhanced Floating-Point Adder with Full Denormal Support," Proc. 29th IEEE Symp. Computer Arithmetic, pp. 35-42, September 2022.

[6] E. Hokenek, R. Montoye, and P.W. Cook, "Second-Generation RISC Floating Point with Multiply-Add Fused," IEEE Journal of Solid-State Circuits, vol. 25, no. 5, pp. 1207-1213, 1990.

[7] S. D. Trong, M. Schmookler, E. M. Schwarz, and M. Kroener, "P6 Binary Floating-Point Unit", Proc. 18th IEEE Symp. Computer Arithmetic, pp. 77-86, 2007.

[8] T. Lang, J.D. Bruguera. Floating-Point Fused Multiply-Add with Reduced Latency. IEEE Trans. on Computers, Vol. 53, No. 8, pp. 988-1003, August 2004.

References

[9] M. Schmookler and K. Nowka, “Leading Zero Anticipation and Detection – A Comparison of Methods, Proc. 15th IEEE Symp. Computer Arithmetic, pp. 7-12, 2001.

[10] G. W. Bewick, Fast Multiplication: Algorithms and Implementation, PhD dissertation, Stanford University, 1994.

[11] V.G. Oklobdzija, D. Villeger, and S.S. Liu, “A Method for Speed Optimized Partial Product Reduction and Generation of Fast Partial Multipliers Using an Algorithmic Approach,” IEEE Trans. on Computers, vol. 45, no. 3, pp. 294–306, Mar. 1996.

[12] R. M. Jessani and M. Putrino, “Comparison of single- and dual-pass multiply-add fused floating-point units,” IEEE Trans. on Computers, vol. 47, no. 9, pp. 927-937, 1998.

[13] G. Even and P.M. Seidel, “A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication,” IEEE. Trans. on Computers, vol. 49, no. 7, pp. 638-650, July 2000.

Bios of Authors

Jongwook Sohn received his B.S. degree in electrical engineering from Korea University, in 2009, M.S. and Ph.D. degrees in electrical and computer engineering from the University of Texas at Austin in 2011 and 2013, respectively. Since 2011, he has been with Intel, where he has been working on floating-point unit design for several E-Core CPUs.

David K. Dean received his B.S. degree in computer engineering with a minor in Mathematics from Iowa State University in 1994. Over the course of 25 years, he has worked at Motorola and Intel on various arithmetic and floating-point units. He joined Intel in 2002 and is now on the E-core floating-point unit design team. David also maintains the Arisim C++ floating-point reference library for E-Cores.

Bios of Authors

Eric Quintana received his B.S. degree in electrical engineering from Texas A&M University in 1984. Over the course of 25 years, he has worked at Motorola on the floating-point unit design of several well-known CPUs, and then later worked at Freescale on the floating-point design of several low-power embedded core CPUs. He joined Intel in 2010 and is now the E-core floating-point design team technical leader. Eric also enjoys writing software and teaching at a local church.

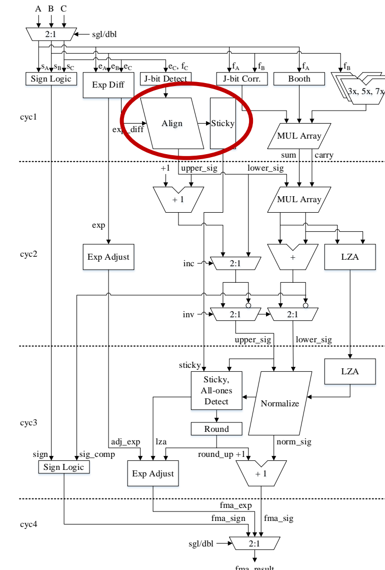
Wing Shek Wong received his M.S. degree in computer engineering from the University of Texas at Austin in 2002. He worked at AMD as design engineer for several CPU generations. He has been with Intel since 2008. He worked as an RTL designer for server product and later joined E-Core CPU team in 2009. He is now architect for E-Core Integer execution unit and Floating-point execution unit. He worked on integer and floating-point unit as well as scheduler design for several E-core CPUs.

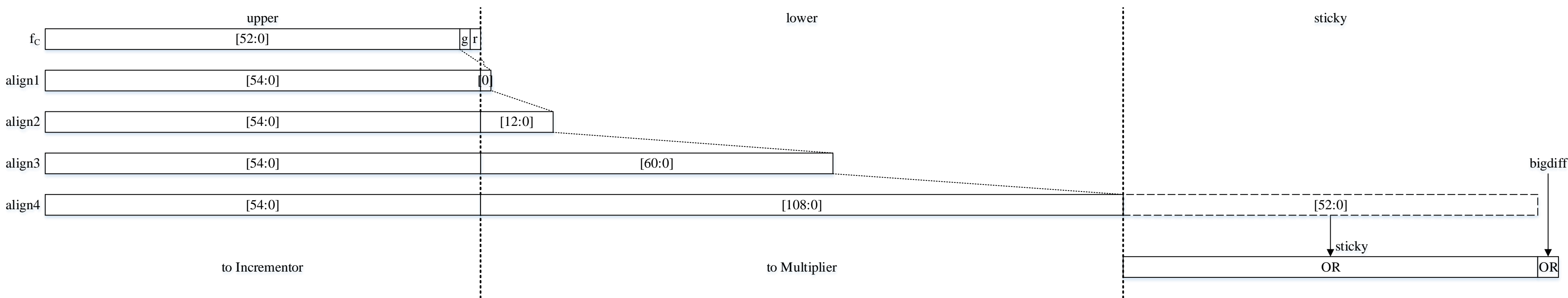
Q&A

Backup - Alignment

Sticky Logic

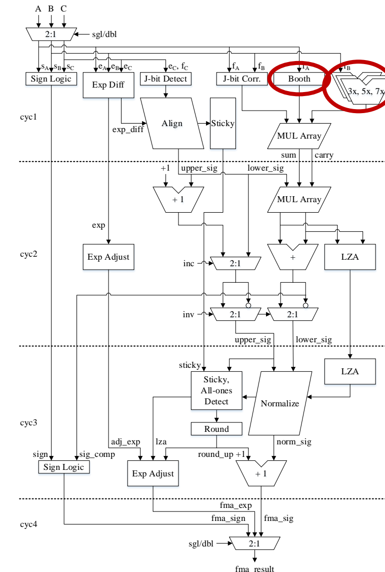
- Sticky logic is in parallel with alignment in each level
- Shifted bits under the round bit in each level are ORed to generate sticky bit
- Sticky bits in each level are ORed to generate the final sticky bit





Backup - Radix-16 Booth Encoding

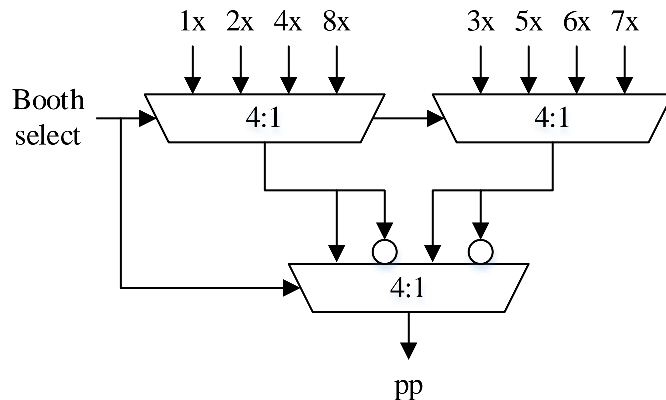
f_A 5-bit grouping (hex)	Radix-16 encode	f_A 5-bit grouping (hex)	Radix-16 encode
00000 (0x00)	+0x	10000 (0x10)	-8x
00001 (0x01)	+1x	10001 (0x11)	-7x
00010 (0x02)	+1x	10010 (0x12)	-7x
00011 (0x03)	+2x	10011 (0x13)	-6x
00100 (0x04)	+2x	10100 (0x14)	-6x
00101 (0x05)	+3x	10101 (0x15)	-5x
00110 (0x06)	+3x	10110 (0x16)	-5x
00111 (0x07)	+4x	10111 (0x17)	-4x
01000 (0x08)	+4x	11000 (0x18)	-4x
01001 (0x09)	+5x	11001 (0x19)	-3x
01010 (0x0A)	+5x	11010 (0x1A)	-3x
01011 (0x0B)	+6x	11011 (0x1B)	-2x
01100 (0x0C)	+6x	11100 (0x1C)	-2x
01101 (0x0D)	+7x	11101 (0x1D)	-1x
01110 (0x0E)	+7x	11110 (0x1E)	-1x
01111 (0x0F)	+8x	11111 (0x1F)	-0x

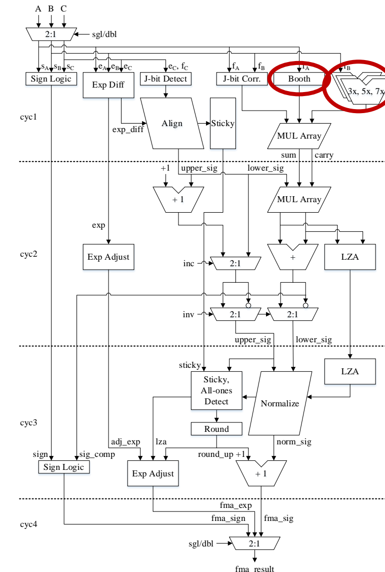


Backup - Radix-16 Booth Encoding

Partial Product Generation

- Select multiples, hi/lo and polarity
- 0x is zeroed out by AOI

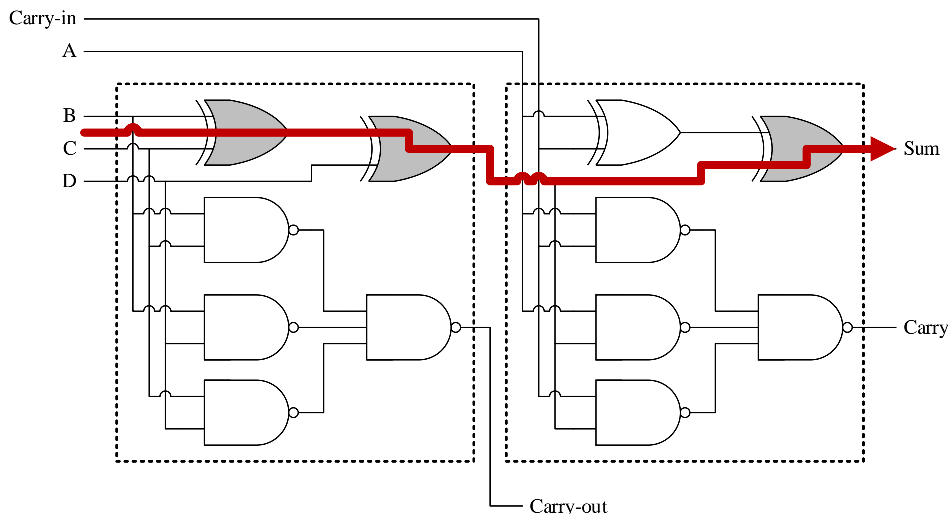


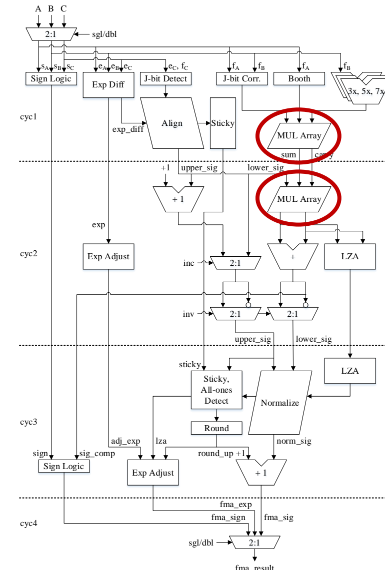


Backup - Multiplier

CSA Interconnection

- 4:2 CSA – back-to-back 3:2 CSAs
  - Timing path – 3 XORs instead of 4

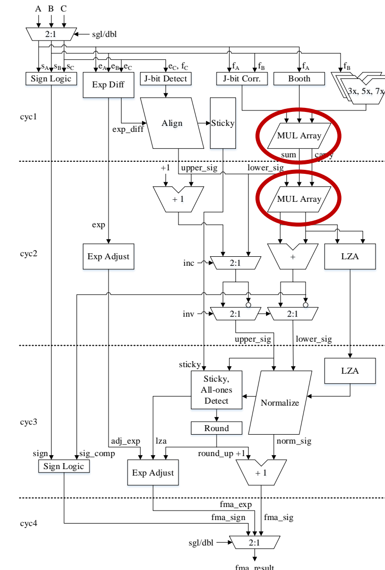




Backup - Multiplier

Radix-4 vs. Radix-16 (Double Precision)

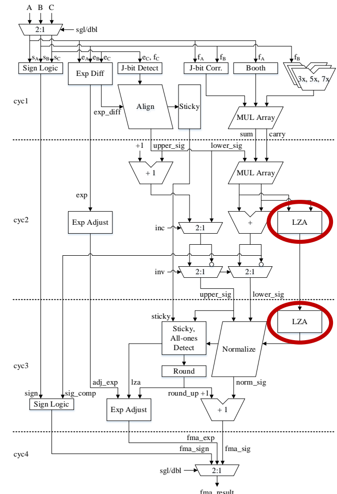
	Radix-4	Radix-16
Latency (gate levels)	3-bit Encoding (6)	5-bit Encoding & 53-bit Adder (10)
	8 Levels of 3:2 CSAs (12)	6 Levels of 3:2 CSAs (9)
	3:2 CSA (2)	No 3:2 CSA (0)
	Total (20)	Total (19), -1
Area (gate count)	3-bit Encoding (200)	5-bit Encoding (400) 53-bit Adder × 3 (1,500)
	27 Partial Products 1,500-bit 3:2 CSAs (9,000)	14 + 2 Partial Products 900-bit 3:2 CSAs (4,800)
	106-bit 3:2 CSAs (600)	No 3:2 CSA (0)
	Total (9,800)	Total (6,700), -30%

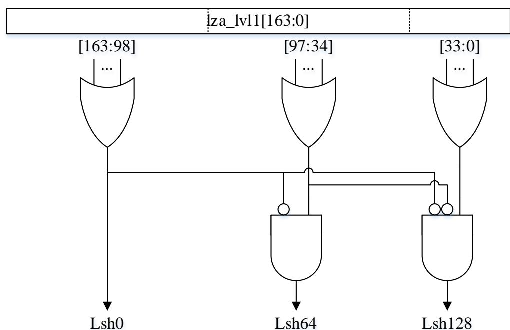


Backup - LZA

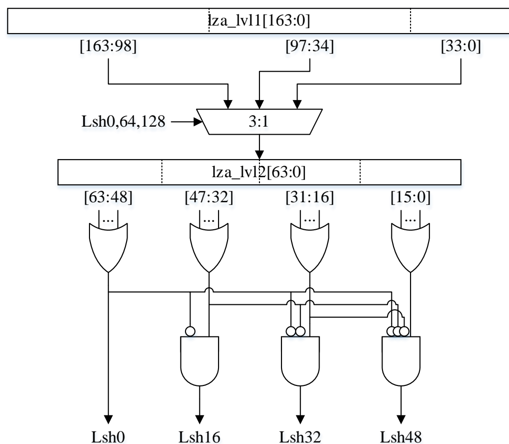
Leading Zero Detection (LZD)

- Four levels of LZD
  - Level 1 – 2 are in the 2nd cycle
  - Level 3 – 4 and post are in the 3rd cycle

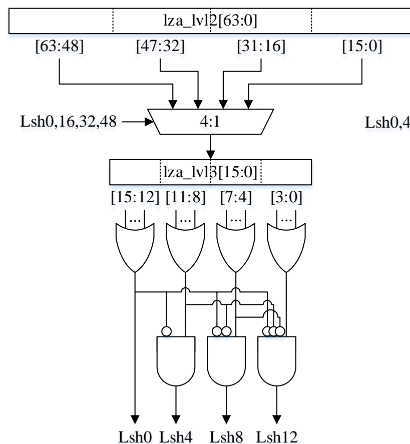




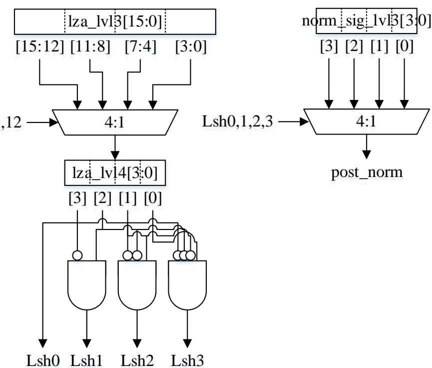
LZA level 1



LZA level 2



LZA level 3



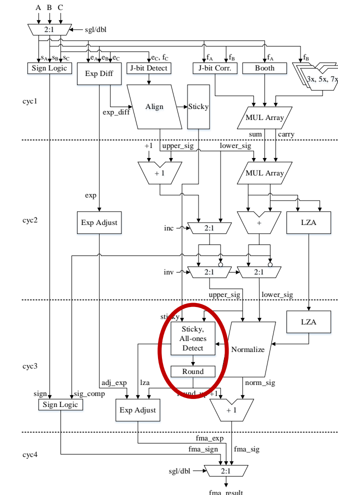
LZA level 4

Post-normaliztion

Backup - Normalization

Sticky Logic

- Sticky logic is in parallel with normalization in each level
- Bits under the round bit in each level are ORed to generate sticky bits
- Sticky bits in each level are ORed to generate the final sticky bit



sum | sum_sig[163:0]

lvl1 | [163:25] | [24:0]

lvl2 | [163:89] | [88:25]

lvl3 | [163:105] | [104:89]

lvl4 | [163:109] | [108:105]

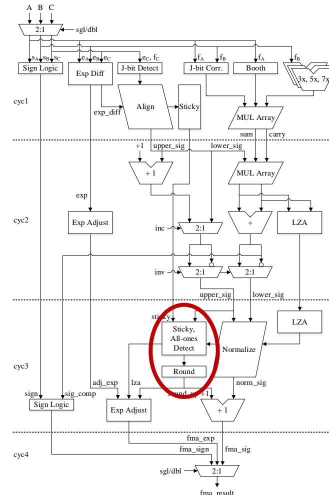
sticky4 sticky3 sticky2 sticky1 align sticky

post_norm & [109] → OR | OR | OR | OR | OR | OR → sticky

Backup - Normalization

All-ones Detection

- All-ones detection is in parallel with normalization in each level
- All-ones is detected by ANDing all bits in the significand range in each level
- All-ones in each level are ANDed to generate the final all-ones



sum [163:0]

lvl1 [163:0]

lvl2 [163:147] [146:62]

lvl3 [163:159] [158:147] [146:110] [109:98]

lvl4 [163] [162:159] [158:110] [109:107]

ones4 ones3 ones2 ones3 ones4

post_norm & [163] → AND AND AND AND AND → all-ones