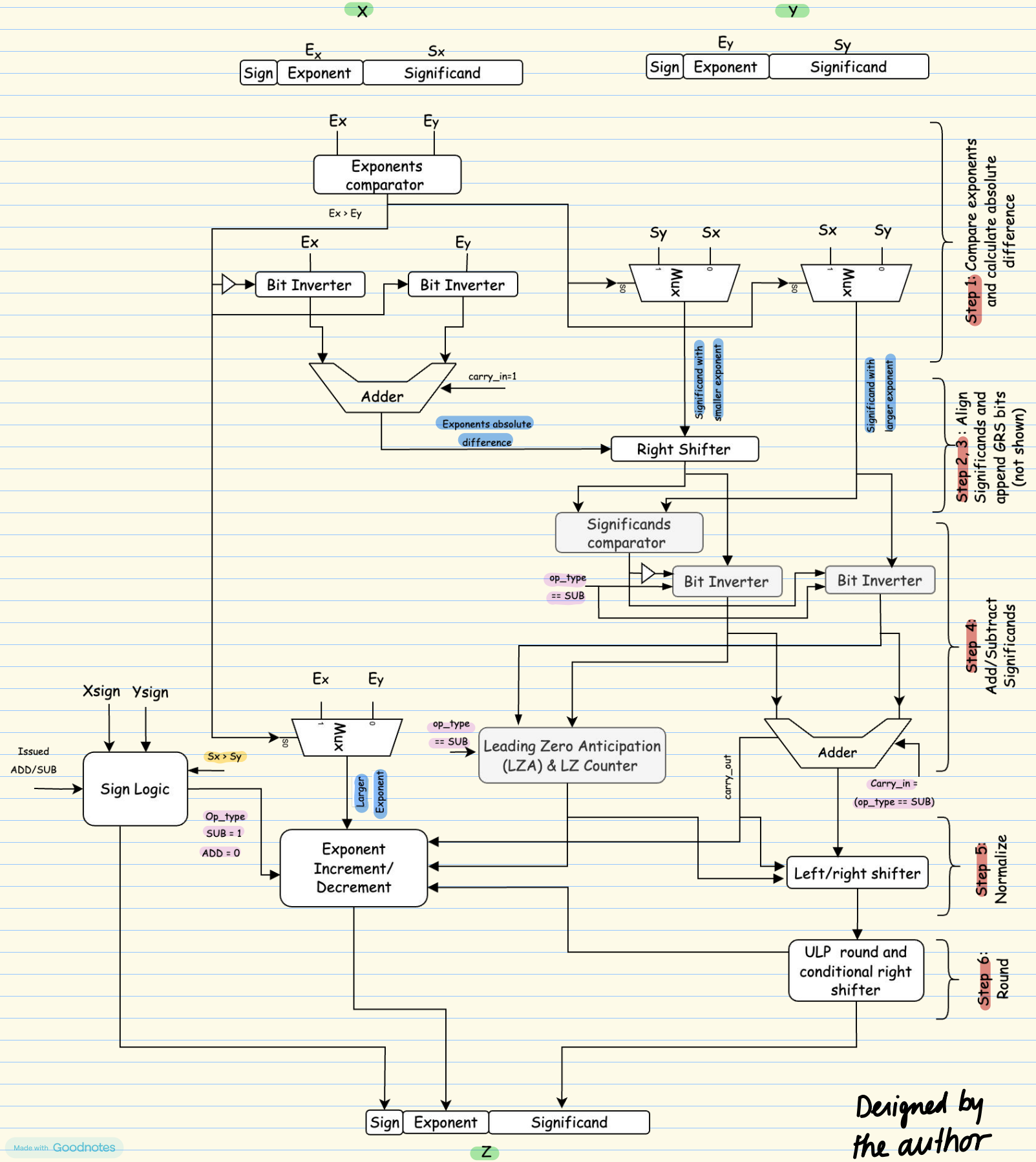
Floating-point Adder/ subtractor



Designed by the author

Made with Goodnotes

Floating Point addition/subtraction Steps

let's take two decimal numbers 6.5 and 1.75 and add them in FP16.

Made with Goodnotes

To start, let's convert decimal to FP16

	$X = 6.5$	$Y = 1.75$
Binary:	110.10	1.11
Normalize:	1.1010×2^2	1.11×2^0
Exponent:	$2 + 15 = 17$ (bias)	$0 + 15 = 15$

FP16 : $\underbrace{0}_{\text{sign}}\ \underbrace{10001}_{\text{exponent}}\ \underbrace{1010000000}_{\text{significand}}$ $\quad$ $\underbrace{0}_{\text{sign}}\ \underbrace{01111}_{\text{exponent}}\ \underbrace{1100000000}_{\text{significand}}$

Made with Goodnotes

Step 1 : compare exponents

→ first step in floating point addition is to compare exponents and Calculate the difference, if any.

Larger exponent : 17

Exponents difference : 17 - 15 = 2

Made with Goodnotes

Step 2: Align the significands

→ the significand of the number that has the smaller exponent has to get shifted right by the difference of the exponents.

Thus, shift the significand of y by 2 bits to the right

$$y = 1.1100000000 \times 2^{0}$$

$$\gg 2$$

Shift right

$$= 0.0111000000000 \times 2^{2}$$

↳ Now, these are 12 bits.
but we only have 10 bits for Significand!

Made with Goodnotes

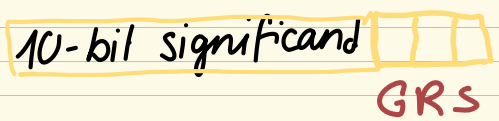
Step 3: Append G, R, S bits for rounding

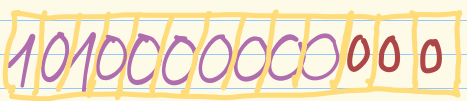
Because aligning significands will lead to bits shifted out, we append 3 bits for rounding.

→ Guard bit (G): the most significant bit of the shifted out bits.

→ Round bit (R): the 2nd most significant bit of the shifted out bits.

→ Sticky bit (S): Reduction OR of all remaining shifted out bits, default zero. That is, it becomes 1 if any 1 is shifted into it.

So it will look like → 

x significand becomes → 

y significand becomes → 

(The two shifted out bits were zero, and s defaults to zero since no bits were shifted into it.)

Made with Goodnotes

Step 4: Add the Significands

Now, significands are ready to be added.

We consider the leading implicit bit, the 10-bit significand, and the GRS in the addition.

lets call the sum result Z:

binary addition +

1.1010000000 0 ← x significand

0.0111000000 0 0 ← y significand

10.0001000000 0 0 ← Z significand

carry out ↖

With the exponent, Z becomes

$10.0001000000\,0\,0 \times 2^{2}$

Made with Goodnotes

Step 5: Normalize

→ The normalization step shifts the Sum so that the leading bit is 1.

→ The direction and the amount of the shift and what we look for in the normalization step depends on whether the operation is addition or subtraction.

Addition

we look for **Carry out**.

if it exists, we shift the significand to the **right** by 1, and **increment** the exponent by 1.

Subtraction

we look for **leading zeros**

and we shift the significand to the **left** till the leading bit is 1. For each shift-left, we **decrement** the exponent by 1.

Sum **before** normalization

$$10.0001000000000 \times 2^2$$

carry-out

Sum **after** normalization (shifted right to have leading 1)

$$= 1.0000100000000\,0 \times 2^3$$

→ the sticky bit is the reduction OR of any bit shifted into it. Since all bits shifted into it are 0, it stays 0

$$= 1.0000100000000 \times 2^3$$

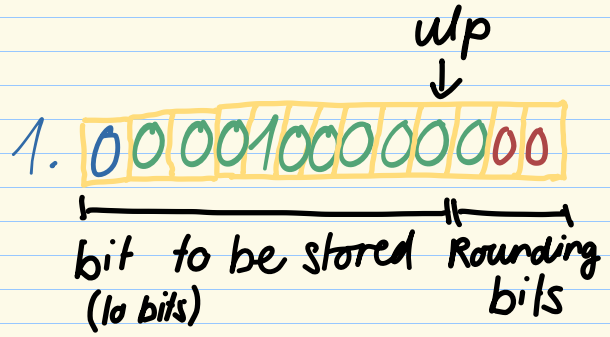
Made with Goodnotes

Step 6: Round

→ Since the significand of the sum is now 13 bits but FP16 has only 10 bits to store the significand, we have to round it.

→ The most commonly used rounding method is round to nearest, ties to even.

→ Here's how it works: the least significant bit in the significand is called unit in the last place (ULP)



if the value of the rounding bits is less than ½ the weight of ULP, the nearest to the significand would be the significand itself, so it stays unchanged.

Made with Goodnotes

if the value of the rounding bits is more than ½ the weight of ULP, the nearest to the significand would be rounding up, we add 1 to the ulp if the addition results into carryout we go to step 5.

Rounding-to-nearest case only depends on the rounding bits.

Rounding bits	Rounding action
0xx	No change to ulp
11x	ulp + 1
101	ulp + 1

(X means don't care, could be 0 or 1)

→ The case in our example

1. 00001000000

10 bits to store after rounding

if the value of the rounding bits is equal to ½ the weight of ULP, it's a tie and we round to the nearest even. if ulp is 0, then the significad is already even and it stays unchanged, if is 1 we add 1 to it to round to the narest even

Ties-to-even case, the value of ulp matters.

ULP

Rounding bits	ulp	Rounding action
100	0	No change to ulp
100	1	ulp + 1

Made with Goodnotes

Result

The result offer all steps is:

10-bit significand: 1. 0000100000
(implicit bit; 10 bits to store)

5-bit exponent: 10010 (binary equivalent of 18 Since exponent is 3 and the bias is 15)

1-bit sign: 0

= 0 10010 0000100000 F-P16 to store
(sign exponent significand)

In Decimal

$$(-1)^{sign} \times 1.\text{Significand} \times 2^{(\text{exponent}-\text{bias})}$$

$$= 1 \times 1.0000100000 \times 2^{3}$$

$$= \left(1 + \frac{1}{2^5}\right) \times 2^3 = 2^3 + 2^{-2} = 8 + 0.25$$

$$= 8.25$$

(which is the correct result of 6.5 + 1.75)

Made with Goodnotes

hardware considerations

A few notes to consider to help understand the hardware:

- A subtractor in hardware is basically an adder with the subtrahend represented in two's complement by inverting its bits and setting the carry-in of the adder to 1. Additionaly, to guarantee that the result of the subtraction is always +ve, we apply two's complement conversion (inversion) to the smaller operand. we determine the smaller operand by using an unsigned comparator, which works because the operands are stored as unsigned magnitude. Guaranteeing that the subtraction result is always positive eliminates the need for checking the carry out and converting the negative result from two's complement to unsigned magnitude. it also allows the use of leading zero anticipation logic that requires the subtrahend to be the smaller-magnitude operand. LZA doesn't wait for the subtraction result, reducing the latency of the critical path.

Made with Goodnotes

→ Regarding Sign Control logic: if the stored sign bits are the same, an issued addition Performs addition in hardware, and an issued subtraction Performs addition in hardware. The resulting sign bit is the same as the operands'. However, if the stored sign bits are different, an issued addition performs a subtraction in hardware and vice versa. The resulting sign is the sign of the larger-magnitude significand.

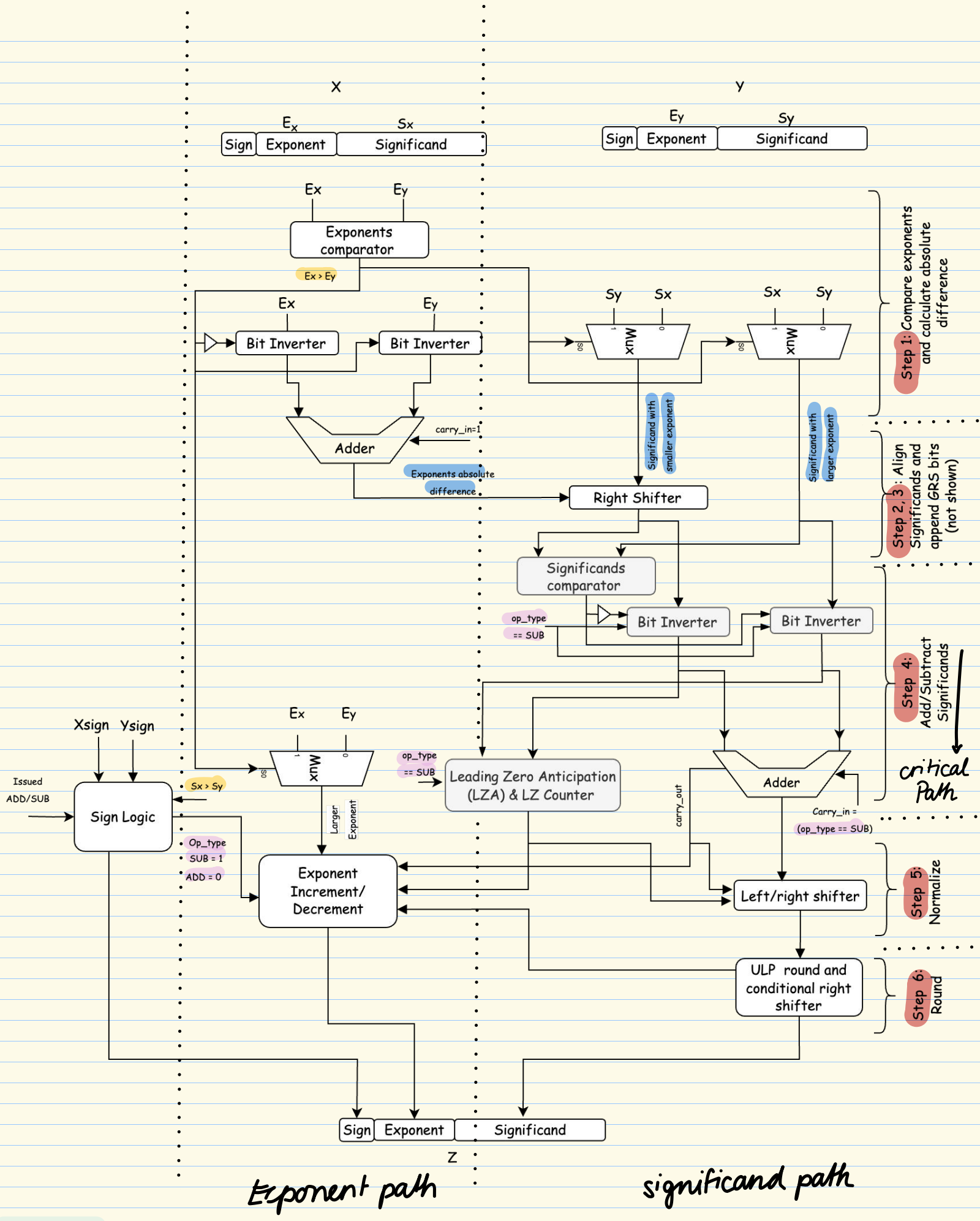
→ while won't be shown, floating point adders have control hardware for cases like equal significands exponents to skip steps, and also for special values like zero, Inf, NAN, and denormal numbers.

- Exponents comparison and difference calculation could be done faster than shown here by using 2 subtractors in Parallel and selecting the result with carry out. while faster, it requires more hardware.

→ the leading zero anticipation logic, leading zero Counter, significands comparators, and bit inverters are skipped when the operation is addition.

let's end the notes by another look at the hardware!

Made with Goodnotes



Made with Goodnotes



www.asicbrew.com

by

Mohamed Abdelgawad

AI Hardware Design Engineer

Made with Goodnotes