BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

CSE306 : COMPUTER ARCHITECTURE SESSIONAL

32-bit FLOATING POINT ADDER CIRCUIT

Section: B1
Group: 04
Participating Rolls:
1905065
1905067
1905069
1905076
1905077

Submitted to
Dr. Rifat Shahriyar
Md. Toufikuzzaman

February 28, 2023

1 Introduction

Floating-point addition is the most frequent floating-point operation and accounts for almost half of the scientific operations. These components demand high numerical stability and accuracy and hence are floating-point based.
A floating point number has generally three parts -

- Sign bit
- Exponent
- Fraction

These parts combines to form a floating point number in the following way-

$$(-1)^{sign} * (1 + Fraction) * 2^{Exponent-Bias}$$

This is known as the normal form of a floating point number. It ensures that there is a single digit to the left of the binary point. Biased exponent is kept so that the exponent part is always positive.
The floating point adder takes two floating point numbers as inputs and outputs the addition of two floating point numbers, as another floating point number. In case of addition, we need to remain careful about the normal form of the number, precision and do the rounding properly.

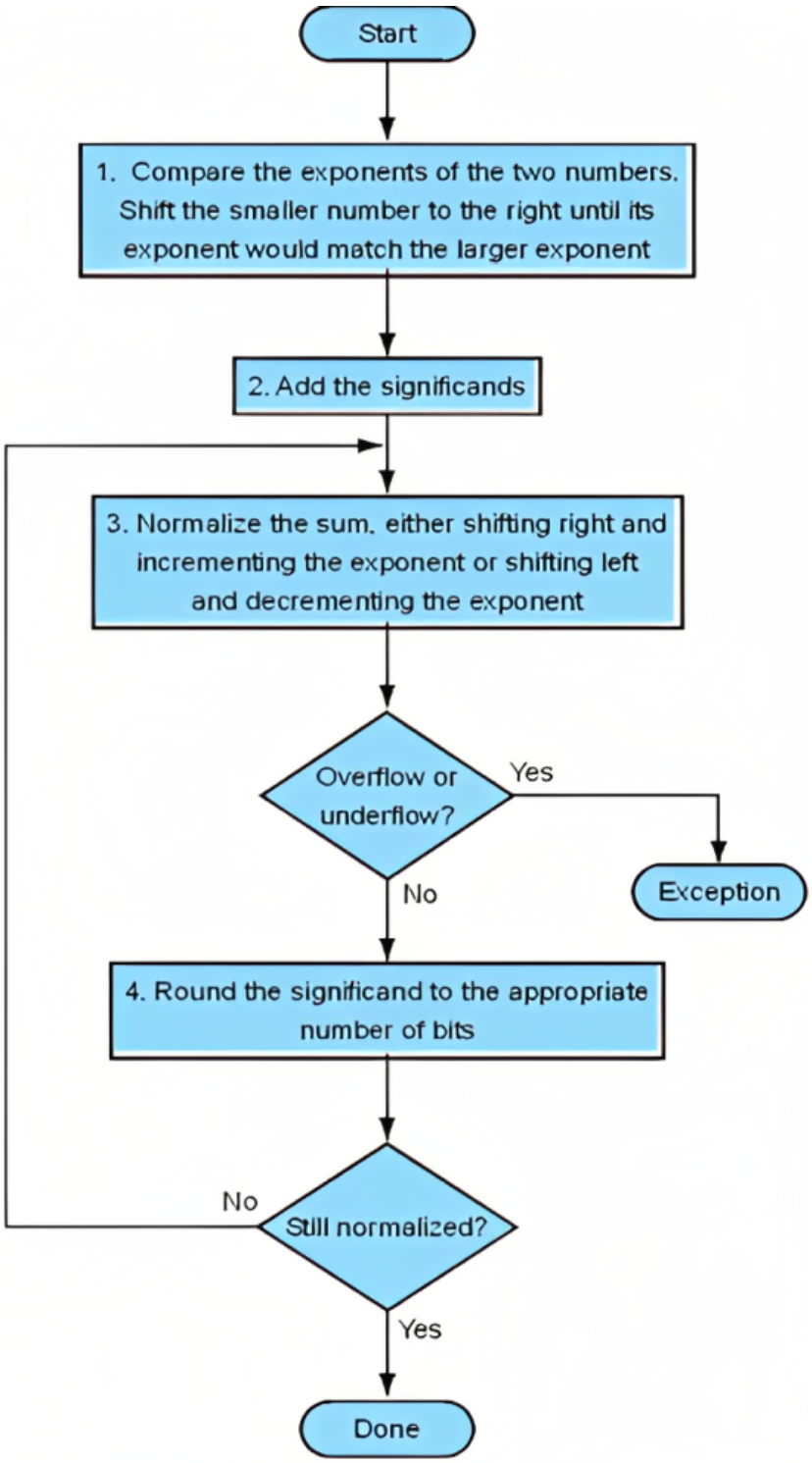
2 Problem Specification

Here we have to implement a floating point adder circuit which will take two floating point numbers. In our representation , each number is represented by 32 bits in the following way -

Sign	Exponent	Fraction
1 bit	12 bit	19 bit

The number obtained after addition must be in normal form. Besides that, we need to ensure that the number is rounded off properly.

3 Floating Point Addition Algorithm



Floating Point Addition Algorithm

4 Device Description

- `MUX`x`bit2x1` : takes two x-bit inputs and outputs the one selected by a one bit selector, constructed using cascading IC74157. Labeled by `M`x

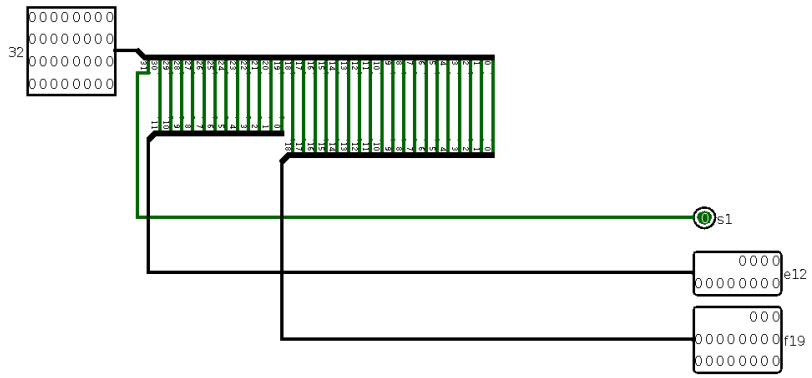
- AND$x$bit$y$x1 : takes y x-bit inputs and outputs their bitwise AND, constructed using cascading IC7408. Labeled by ANDx
- OR$x$bit$y$x1 : takes y x-bit inputs and outputs their bitwise OR, constructed using cascading IC7432. Labeled by ORx
- XOR$x$bit$y$x1 : takes y x-bit inputs and outputs their bitwise XOR, constructed using cascading IC7486. Labeled by XORx
- NOT$x$bit : takes one x bit input and outputs its bitwise NOT, constructed using cascading IC7404. Labeled by NOTx
- Adder$x$bit : takes two x bit inputs and a carry in bit and outputs their bitwise sum and the carry out bit, constructed using cascading IC7483. Labeled by Ax
- inputSplitter : splits an incoming 32 bit data into 3 outgoing wires 1, 12, 19 bits each.
- outputSplitter : combines 3 incoming 1, 12 and 19 bit data into a single 32 bit data.
- makeSignificand : converts an incoming 19 bit data into 32 bit outgoing data, with msb=1 and the lsb's=0.
- selectInverse : takes a 32 bit input and a selector, outputs the same input if selector is set to 0, otherwise send the bitwise not (1's complement).
- incrementExponent : takes a 12bit input and outputs the one-increment of it.
- CompareWith4 : takes a 3 bit input and outputs whether the input is greater, equal or less than 4.
- StickyAdderBit : looks at the 4 lsb of a number and outputs the value that should be added to the new lsb.
- CustomLeftShifter : takes a 32bit input and a 12bit shiftAmount, outputs the input left shifted by that amount, with the shifted bits replaced by the shift bit.
- CustomRightShifter : takes a 32bit input and a 12bit shiftAmount, outputs the input right shifted by that amount, with the shifted bits replaced by the shift bit.
- PriorityEncoder : takes a 32bit input and outputs the position of the first 1 in the input, plus one.
- Zero32bit : 32 bit output that is all zero's.
- Control : check section-5.2
- Normalizer : check section-5.6
- Rounder : check section-5.7

5 Detailed Design Steps

We shall describe the design steps by listing and detailing out each of the major individual components.

5.1 Splitter

First, two 32 bit inputs, A and B will be inserted into `inputSplitter`, this will split the 32 bits into three individual lines of 1-bit, 12-bit and 19-bit each i.e., separating the sign, exponent and fraction parts. We denote them as A_s, A_e, A_f, B_e, B_f and B_g.



`inputSplitter`

5.2 Control

Inside `Control`, we find the difference between A_e, B_e and A_f, B_f. We use their result to output three values:

- The absolute difference between A_e and B_e (Denoted as D_e)
- The larger value between absolute values of A and B ($S_{|A|>=|B|}$)
- Whether the absolute values of A and B are identical ($S_{|A|=|B|}$)

These values will be later used to shift fraction values and used as selector bits for numerous MUXs.

5.3 Decider

Using five 2x1 MUX and ($S_{|A|>=|B|}$, we figure out G_s, G_e, G_f, L_s, L_f, such that, $G = max(|A|, |B|)$ and $L = min(|A|, |B|)$.
The observation here is that we must always right shift the addend with the lower exponent value and make it equal to the exponent of the larger value. Hence, we must determine which addend is larger which one is smaller.
The second observation is, before addition, we should treat both addends to be having the same exponent value, which is that of the larger addend. Hence, we don't need the value of the exponent of the lower exponent in the future.
Devices Used: `two M1, one M12, two M19`

5.4 Appropriator

Few things that must be done before the addition:

- Determine whether to add or subtract, for this we take the XOR value of G_e and L_e.
- None of the fraction parts so far have the leading 1 in their current representation, we use the device `makeSignificand` to put a leading one at the leftmost bit of each fraction, and to also convert the 20 bit fractions into 32 bit values.

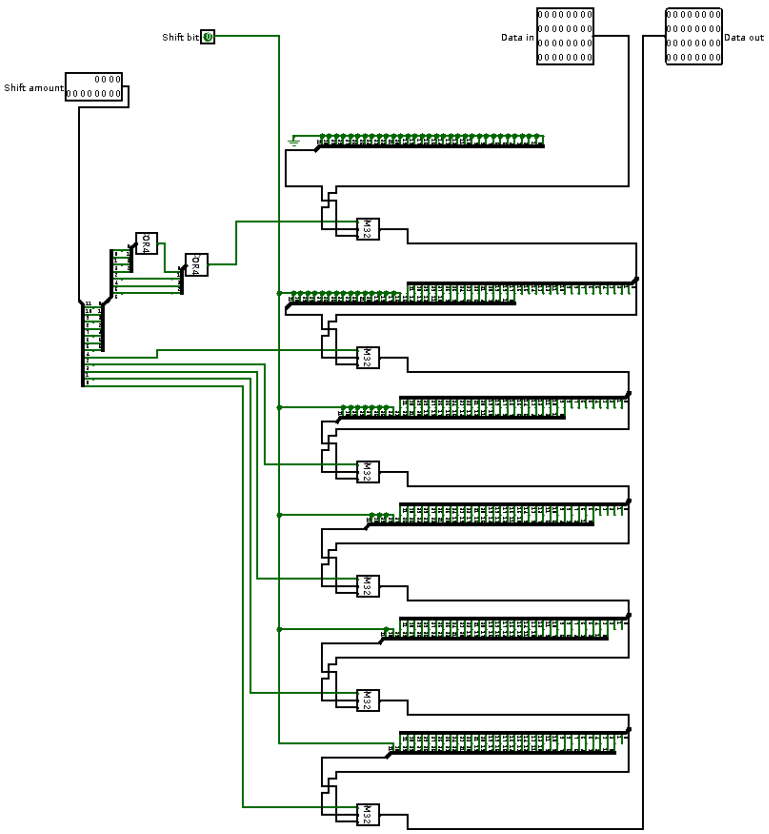


Figure 1: `CustomRightShifter`

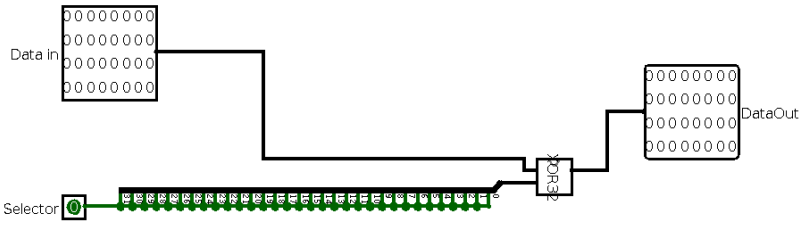


Figure 2: `selectInverse`

- `CustomRightShifter` takes a 32 bit input, a 12 bit shift amount value, and outputs a value which has been right shifted by that amount. Thus, inserting L_f and D_e will shift L_f by an amount of D_e, i.e., make $G_e = L_e$.
- For subtraction, no matter what the signs are, we shall always subtract L from G, that way, we won't need to invert that result after addition. The sign of the result will simply be G_s. In order to accomplish this, we used the device `selectInverse` that outputs the 2's complement of the right shifted L_f only when $G_e \oplus L_e$ is 1, otherwise send L_f as it is.

5.5 Adder

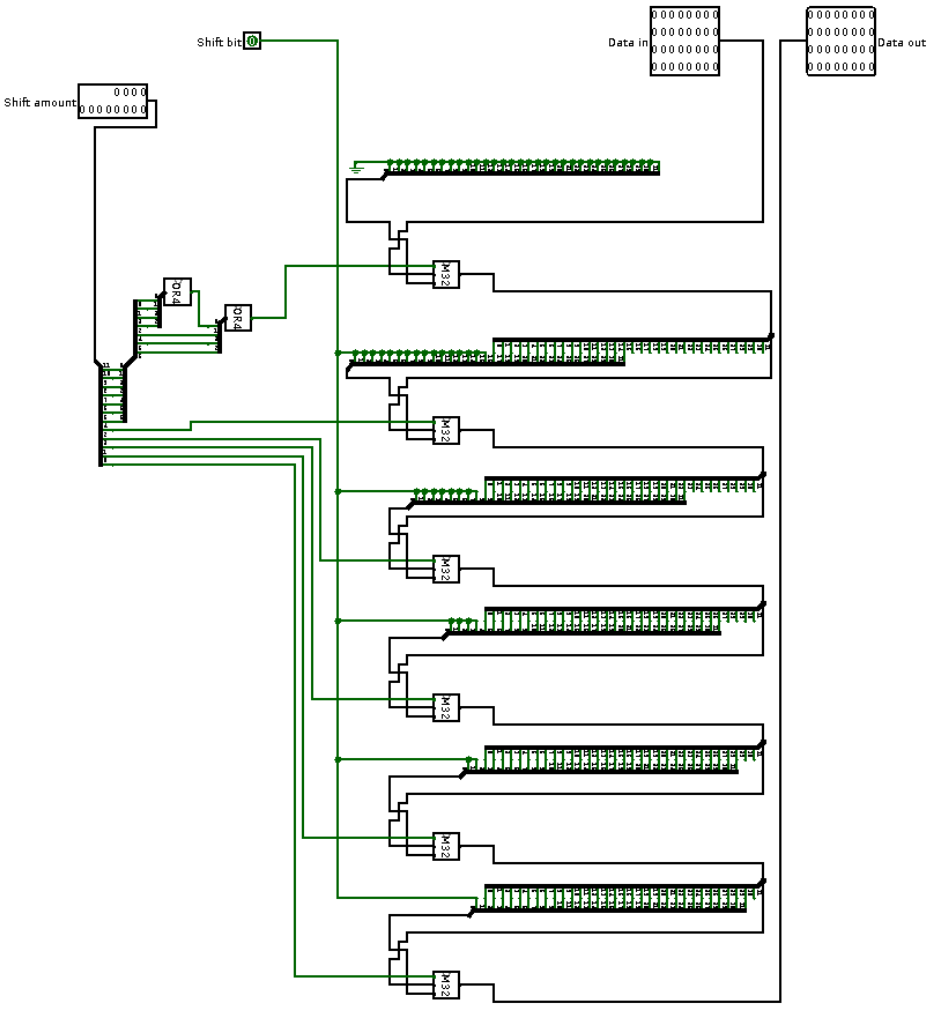
Here we used a simple 32 bit adder. The value of C_{in} should be 1 when we are doing a subtraction operation.
Besides, we also did $F_e = G_e + 1$ using `incrementExponent`.

5.6 Normalizer

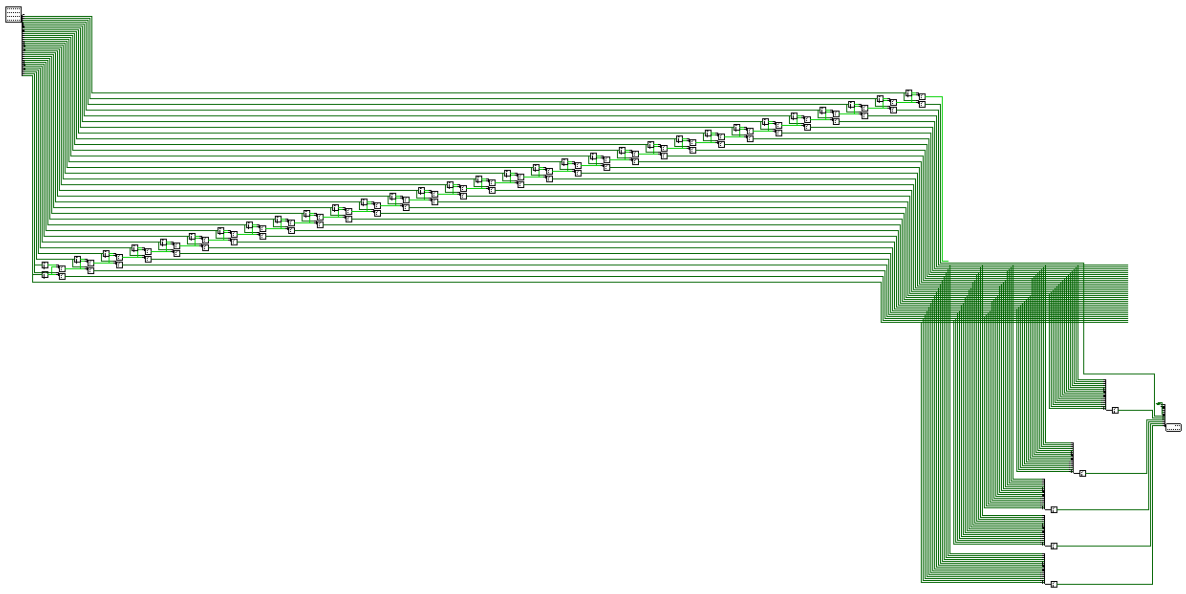
Before normalizing two cases of the result that should be considered:

1. There is a carry out: this indicates that the leftmost 1 is already outside the result fraction (we hereby denote it as F_{f32}), hence, we don't need to right shift it anymore.
2. Otherwise, we need to left shift the result until the left most 1 is outside the 32 bits.

No matter if C_{out} of `A32` is set or not, we send the result (F_{f32}) and F_e inside `Normalizer`. Here, using a `PriorityEncoder`, we first determine the amount of shift we need to do (D_{norm}). We send D_{norm} and F_{f32} to the `CustomLeftShifter`, and subtract D_{norm} from F_e.
Using another pair of MUX, we determine using C_{out} of `A32` whether we need to send the original value to the output or the shifted value.
Thus, we get the final normalized unrounded 1+12+32 = 45-bit result.



`CustomLeftShifter`



PriorityEncoder

5.7 Rounder

Using the rounding truth table of section 5.1, we constructed `StickyAdderBit`, which takes 4 consecutive bits as input and outputs 1 bit. The idea here is that by looking at lsb and GRS, we need to make GRS = 000 and decide whether to add 1 to lsb or not, which is exactly what `StickyAdderBit` does.
Since we are always either truncating or adding, the only case where we need to normalize after rounding is when rounding causes the leftmost 1 to overflow. This can only happen when every leading bit is 1, and adding 1 with the lsb causes every bit to become 0. Further right shifting in this case will increase the exponent value by 1. Hence, after each stage of rounding, we shall use a pair of MUX to decide using the C_{out} bits in every stage whether we shall send all zeros to the next stage or the current result.
Now, we need to reduce 32 bits to 19 bits using the rounding principles. We did that in 5 stages:

- reduce 32 bits to 31 bits, this required a 31 bit adder, a 31 bit MUX
- reduce 31 bits to 28 bits, this required a 28 bit adder, a 28 bit MUX
- reduce 28 bits to 25 bits, this required a 25 bit adder, a 25 bit MUX
- reduce 25 bits to 22 bits, this required a 22 bit adder, a 22 bit MUX
- reduce 22 bits to 19 bits, this required a 19 bit adder, a 19 bit MUX

Beside these, we also need a 12 bit adder and a `StickyAdderBit` circuit in every stage. Thus, in our `Rounder` circuit, we take input an unrounded 12 bit exponent and an unrounded 32 bit fraction, and output a rounded 12 bit exponent and a rounded 19 bit fraction.

5.8 Overflow and Underflow

Overflow and underflow flags were kept to signal that the output has exceeded the boundaries of normal numbers.
For any two normal numbers the highest allowable exponent is $2^{12}-2$, hence, the exponent

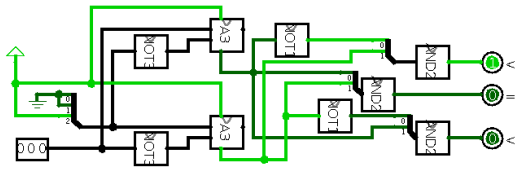


Figure 3: `CompareWith4`

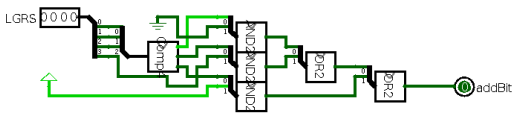


Figure 4: `StickyAdderBit`

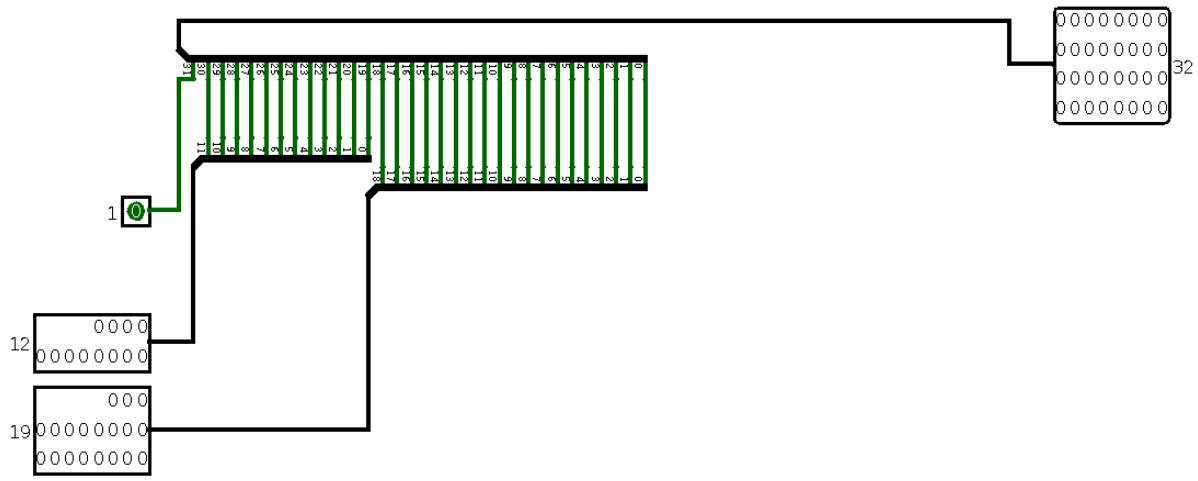
of their sum can never exceed $2^{12} - 1$. Thus, checking if adding one to the final exponent results in a carry out is enough to check for any overflow.
On the other hand, for any two normal numbers the lowest allowable exponent is 1, but since each of the fraction parts has 19 bits, the exponent of their difference can be equal to or lower than 0. Hence there are two cases:

- The exponent of the sum if zero, in that case checking if their NOR is 1 is sufficient
- The exponent of the sum has become lower than zero. In this case, we checked if the carry out bit in the exponent decrementer of the `CustomLeftShifter` is zero. If it is, this means that the sign of the exponent has changed.

Taking the OR of these two values is enough for checking whether the sum has caused an underflow.

5.9 Joiner

Finally using `outputSplitter` we combine the 1 bit sign bit, 12 bit exponent and 19 bit fraction into a 32 bit output.



`outputSplitter`

6 Truth Tables Used

6.1 Equation for `StickyAdderBit`

G	R	S	**Add with lsb**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	lsb
1	0	1	1
1	1	0	1
1	1	1	1

Implemented using comparator and combinatorial 3x1 mux.

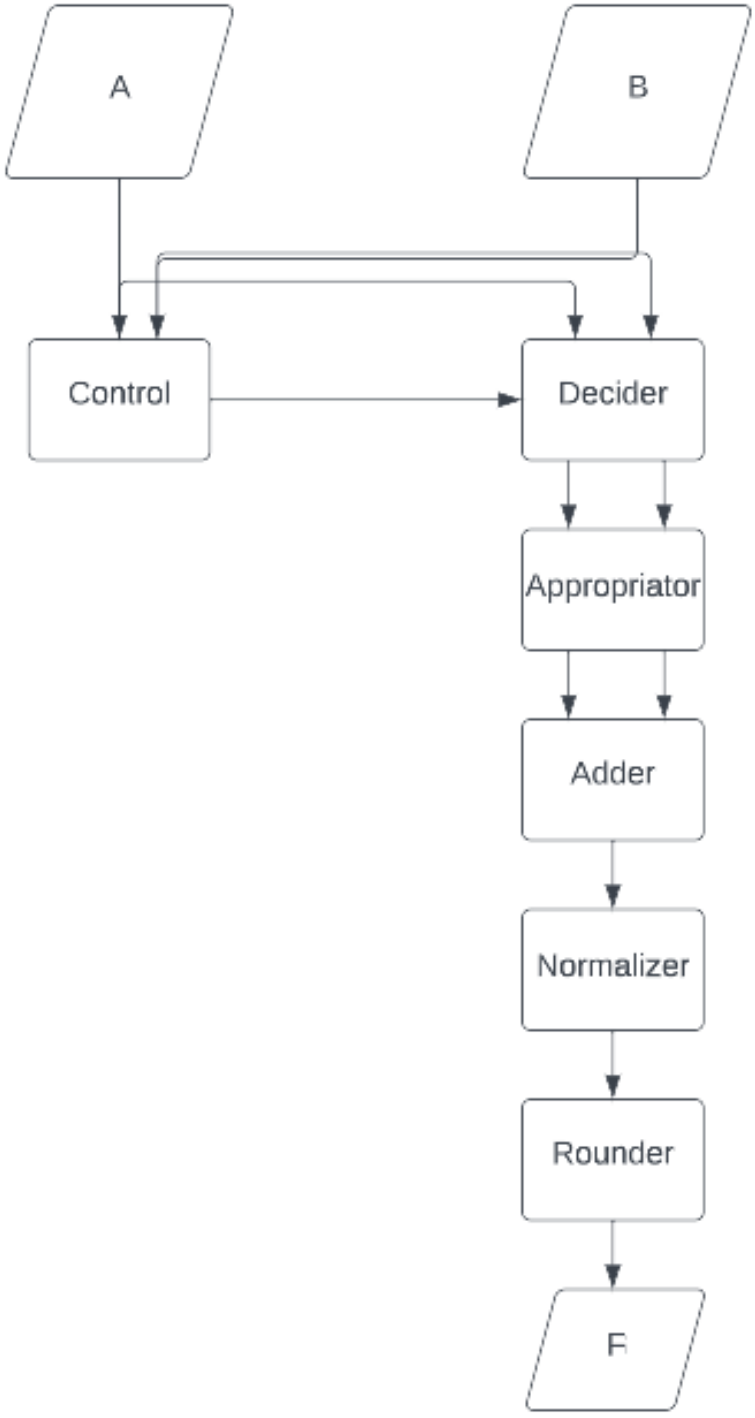
6.2 Normalize or Not

G_s	L_s	C_{out}	**Normalize or not (No-1 Yes-0)**
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

This is equivalent to $\overline{G_s \oplus L_s} \cdot C_{out}$

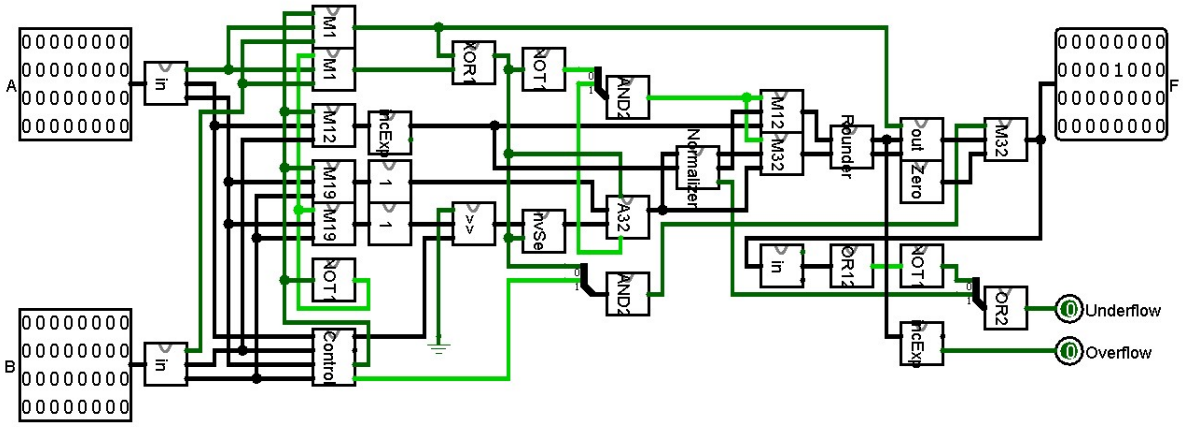
7 Diagrams

7.1 Block Diagram

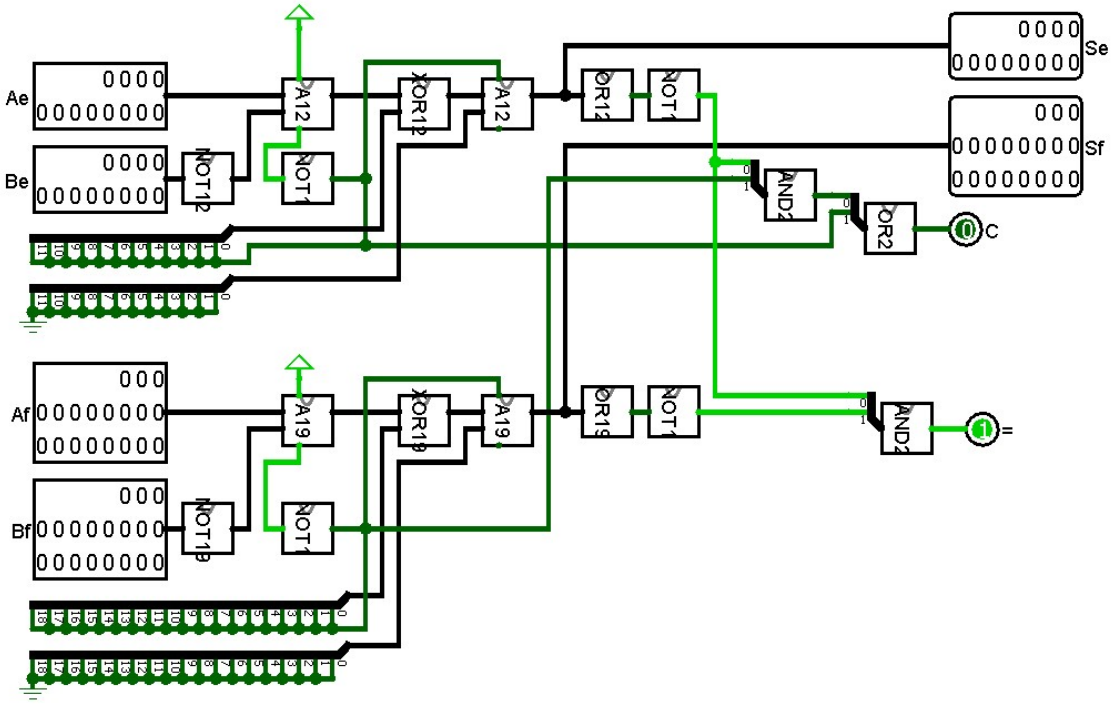


High Level Architecture Block Diagram

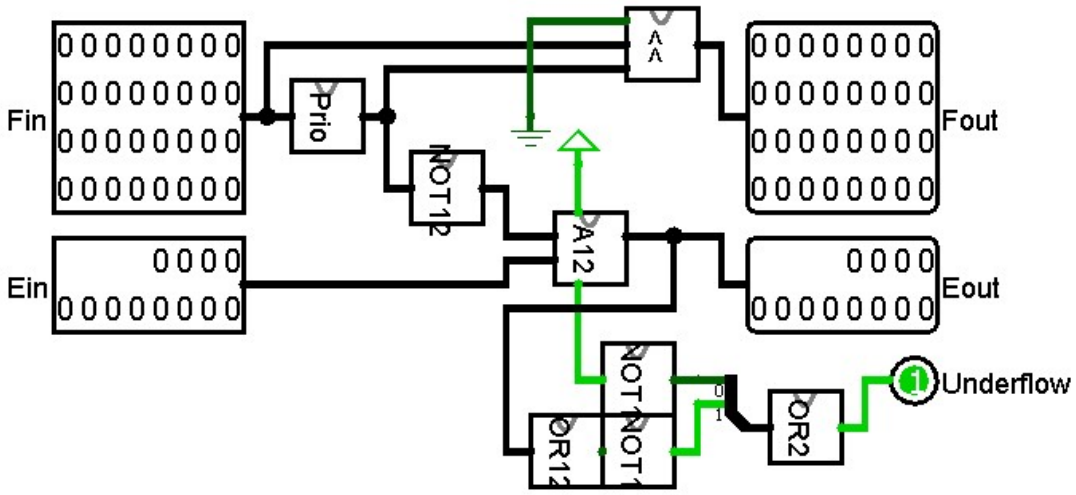
7.2 Circuit Diagram



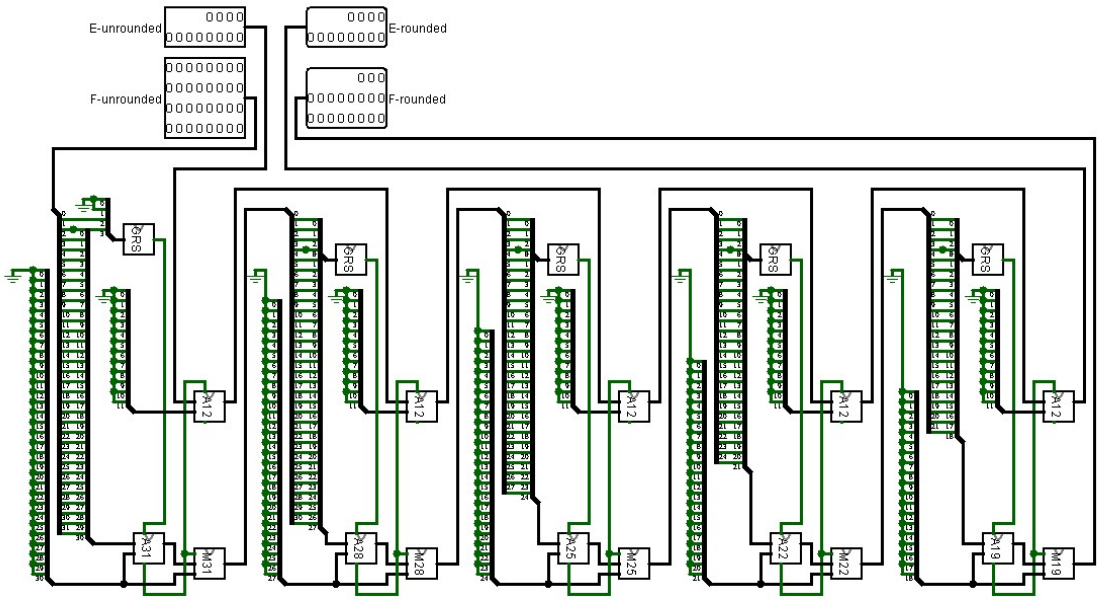
Circuit Diagram



Control



Normalizer



Rounder

8 Tools and Apparatus

8.1 Integrated Circuits

IC Number	IC Name	Count
SN74HC04N	Hex Inverter	67
SN74HC08N	Quad 2-Input AND	65
SN74HC32N	Quad 2-Input OR	130
SN74HC86N	Quad 2-Input XOR	17
SN74HC83N	4bit Binary Full Adder	91
SN74HC157N	Quad 2 to 1 MUX	163

8.2 Simulator

Software: Logisim
Version: logisim-win-2.7.1

9 Discussion

1. The whole circuit was constructed using 7400-library integrated circuits.
2. Individual components were encapsulated using smaller components, thus, little care was given to optimization of IC numbers.
3. The whole circuit has is combinatorial i.e., no sequential or memory components such as register, or flipflops were used, only basic logic gates were used. Measures taken to make it possible has been discussed below:
    - In order to make both addend's exponent equal, `CustomRightShifter` was used. This device uses six 32-bit 2x1MUX that check each bit of the shift amount value and decide whether to send the shifted or the unshifted value of the input. For shift amount values higher than 31, the OR of the seven most significant bits of shift amount were taken. If the result of this value is 1, then the circuit sends a full zero value.
    - Inside `Normalizer`, two key components were used. The first one is a `PriorityEncoder`, which takes the value of the 32-bit fraction and outputs the position of the left-most 1. This value is later used in `CustomLeftShifter` to shift the same 32-bit fraction by that amount, and is also subtracted from the exponent value.
    - Inside `Rounder`, we've cascaded 5 sets of 3 bit truncators, each of which truncates 3 bits from the fraction input from `Normalizer`. The exponent value is also rounded accordingly inside `Rounder` using adders.
4. For selection/sorting of values, MUXs were used in plentiful. Carry out bits from different stages function as selector bit of MUX in numerous occasions.
5. To preserve even the lowest bit of information and prevent any data loss, we took a number of steps:

- We've considered the carry out after the addition operation as being the leading one of the result, that way, we only need to shift the sum leftwards. Since we are avoiding any right shifting of the result, no data will be lost.
- To avoid preserving one bit for sign, we handled the decision of addition or subtraction using control. The sign bit of the larger addend was used as the sign bit of the sum. For subtraction, the addend with the smaller absolute value is always subtracted from the addend from the larger absolute value.
- Rounding has been done rigorously using five cascading rounder devices.

6. The circuit has been constructed keeping in mind that the addends will always be in normalized form i.e., no Zero, denormalized, NaN or infinity values should be provided as inputs.
7. In case the result is an unnormalized number, Overflow and Underflow flags have been constructed to function accordingly using the nature of the output and values of carry out bits from different stages.