Floating Point Representation

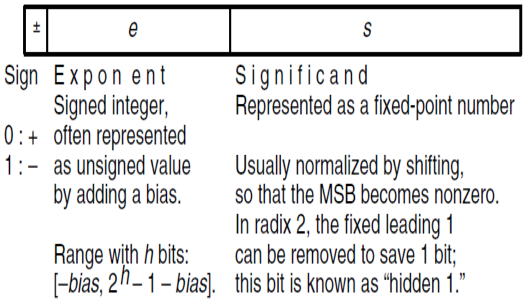
By Kumar Sai Reddy Guntaka

Introduction: Floating Point arithmetic is by far the most used way of approximating real number arithmetic for performing numerical calculations on modern computers. Each computer had a different arithmetic for long time: bases, significant and exponents' sizes, formats, etc. Each company implemented its own model, and it hindered the portability between different equipment's until IEEE 754 standard appeared defining a single and universal standard. The floating-point arithmetic is clearly the most efficient way of representing real numbers in computers. Representing an infinite, continuous set (real numbers) with a finite set (machine numbers) is not an easy task: some compromises must be found between speed, accuracy, ease of use and implementation and memory cost. Floating Point Arithmetic represent a very good compromise for most numerical applications.

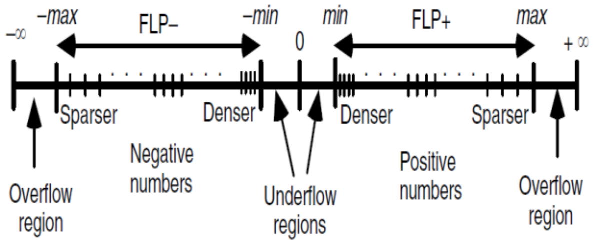
Fixed-point number systems: Provide a restricted range and/or accuracy. To guarantee that values remain representable and do not lose too much accuracy, computations must be "scaled."

A typical floating-point representation format:

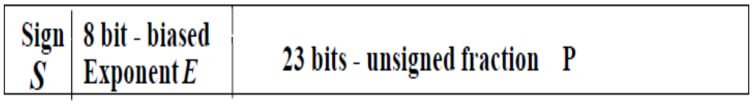
1. 1. The significand, also known as the number sign, represents a positive or negative floating-point number and is often represented by a distinct sign bit (signed-magnitude convention).
2. The exponent sign, which is frequently contained in the biased exponent, denotes a high or small number. When the bias is a power of two (for example, 128 with an exponent of 8 bits), the exponent sign is the complement of the most-significant bit(MSB).



±
e
s
Sign
0 : +
1 : –
Exponent
Signed integer,
often represented
as unsigned value
by adding a bias.
Range with h bits:
[–bias, 2^h – 1 – bias].
Significand
Represented as a fixed-point number
Usually normalized by shifting,
so that the MSB becomes nonzero.
In radix 2, the fixed leading 1
can be removed to save 1 bit;
this bit is known as "hidden 1."

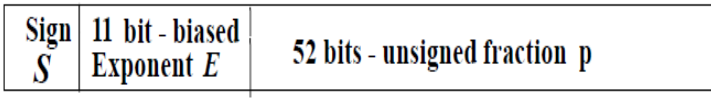


IEEE Single Precision Format:



Sign S	8 bit - biased Exponent E	23 bits - unsigned fraction P

IEEE Double Precision Format:



Sign S	11 bit - biased Exponent E	52 bits - unsigned fraction p

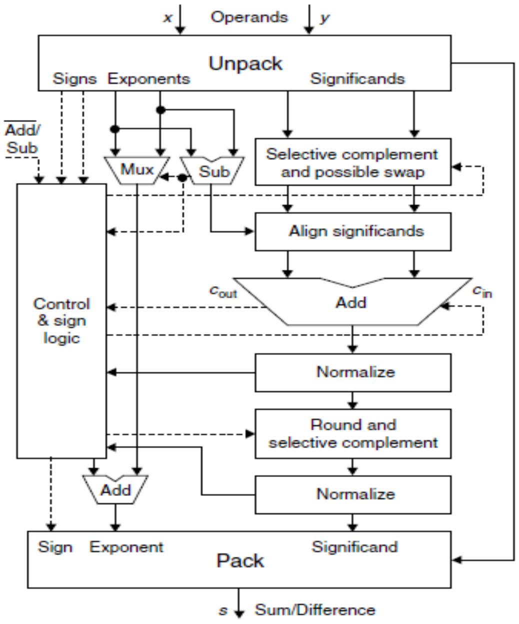
Conversion from decimal to IEEE format:

1. **Choose single or double precision.** When writing a number in single or double precision, the steps to a successful conversion will be the same for both, the only change occurs when converting the exponent and mantissa.
   - First we must understand what single precision means. In floating point representation, each number (0 or 1) is considered a “bit”. Therefore single precision has 32 bits total that are divided into 3 different subjects. These subjects consist of a sign (1 bit), an exponent (8 bits), and a mantissa or fraction (23 bits).
   - Double precision, on the other hand, has the same setup and same 3 parts as single precision; the only difference is that it will be larger and more precise number. In this case, the sign will have 1 bit, the exponent will have 11 bits and the mantissa will have 52 bits.
3. **Separate the whole and the decimal part of the number.** Take the number that you would like to convert, and take apart the number so you have a whole number portion and a decimal number portion.
4. **Convert the whole number into binary.**
5. **Convert the decimal portion into binary.**
6. **Combine the two parts of the number that have been converted into binary.**
7. **Convert the binary number into base 2 scientific notation.** You can convert the number into base 2 scientific notation by moving the decimal point over to the left until it is to the right of the first bit. These numbers are normalized which means the leading bit will always be 1. As for the exponent, the number of times that you moved the decimal will be your exponent in base 2 scientific notation. Remember that moving the decimal to the left will result in a positive exponent while moving the decimal to the right will result in a negative exponent.
8. **Determine the sign of the number and display in binary format.** You will now determine if your original number is positive or negative. If the number is positive, you will record that bit as 0, and if it is negative, you will record that bit as 1.
9. **Get the exponent based on precision.** There are set biases for both single and double precision. The exponent bias for single precision is **127**, which means we must add the base 2 exponent found

previously to it. Thus, the exponent you will use is **127+6** which is **133**. Double precision as perceived from the name is more precise and can hold larger numbers. Therefore its exponent bias is **1023**. The same steps used for single precision apply here, so the exponent you can use to find double precision is 1029.

10. **Turn the exponent into binary.** After you determine your final exponent, you will need to convert it into binary so that it could be used in the IEEE 754 conversion.
11. **Determine the mantissa.** The mantissa aspect, or the third part of the IEEE 754 conversion, is the rest of the number after the decimal of the base 2 scientific notation. You will just drop the 1 in the front and copy the decimal portion of the number that is being multiplied by 2.

FLOATING-POINT ADDERS/SUBTRACTORS:



As shown in Figure, the two operands entering the floating-point adder are first unpacked. Unpacking involves: Separating the sign, exponent, and significand for each operand and reinstating the hidden 1. Converting the operands to the internal format, if different (e.g., single-extended, or double-extended).

Testing for special operands and exceptions (e.g., recognizing not-a-number inputs and bypassing the adder). Floating-point arithmetic with subnormal's can be found elsewhere.

The difference of the two exponents is used to determine the amount of alignment right shift and the operand to which it should be applied.

To economize the hardware pre-shifting capability is often provided for only one of the two operands, with the operands swapped if the other one needs to be shifted.

Since the computed sum or difference may have to be shifted to the left in the post normalization step, several bits of the right-shifted operand, which normally would be discarded as they moved off the right end, may be kept for the addition.

If both operands have the same sign, the common sign can be ignored in the addition process and later attached to the result.

Selective complementation, and the determination of the sign of the result, are also affected by the Add/Sub control input of the floating-point adder/subtractor, which specifies the operation to be performed.

Rounding the result may necessitate another normalizing shift and exponent adjustment.

To improve the speed, adjusted exponent values can be precomputed and the proper value selected once the normalization results become known.

The significand adder is almost always a fast logarithmic time 1’s- or 2’s-complement adder, usually with carry-lookahead design. When the resulting significand is negative, it must be complemented to form the signed-magnitude output. Combining the sign, exponent, and significand for the result and removing the hidden 1.

Addition/Subtraction Steps:

1. Extracting signs, exponents and mantissas of both A and B numbers.

2. Treating the special cases:
   Operations with A or B equal to zero.
   Operations with +/- ∞.
   Operations with NaN.

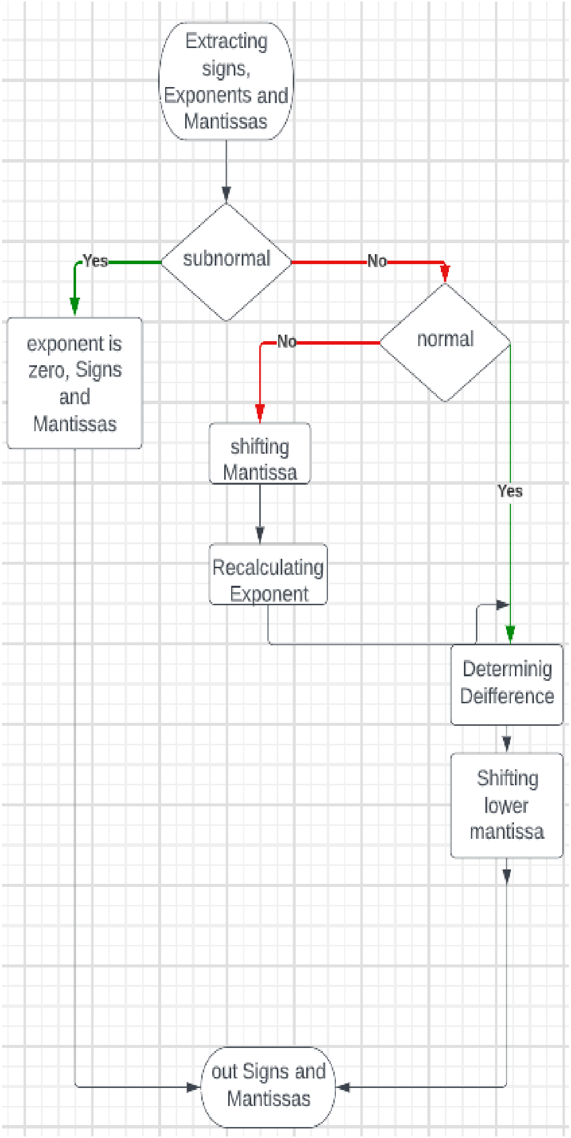
3. Finding out what type of numbers are given:
   Normal
   Subnormal
   Mixed

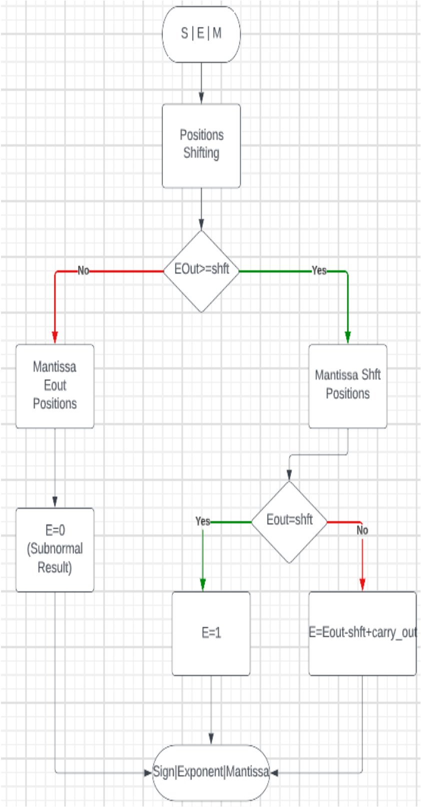
4. Shifting the lower exponent number mantissa to the right [*Exp*1- *Exp*2] bits.

5. Addition/Subtraction of the numbers and detection of mantissa overflow (carry bit).

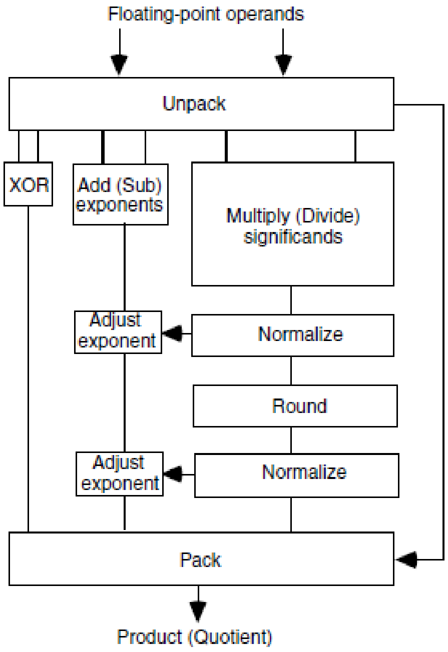
6. Standardizing mantissa shifting it to the left up the first one will be at the first position and updating the value of the exponent according with the carry bit and the shifting over the mantissa.

7. Detecting exponent overflow or underflow.





FLOATING-POINT MULTIPLIERS AND DIVIDERS:



$$(\pm s1 \times b^{e1}) \times (\pm s2 \times b^{e2}) = (\pm s1 \times s2) \times b^{e1+e2}$$

A floating-point multiplier consists of a fixed-point multiplier for the significands, plus peripheral and support circuitry to deal with the exponents and special values (±0, ±∞, NaNs and subnormal's). Depicts a generic block diagram for a floating-point multiplier. The role of unpacking is exactly as discussed for floating-point adders. Similarly, the final packing of the result is done as for floating-point adders. The sign of the product is obtained by XORing the signs of the two operands.

With the IEEE 754-2008 short format, subtracting the bias of 127 can be easily accomplished by providing a carry-in of 1 into the exponent adder and subtracting 128 from the sum.

Since multiplying the significands is the most complex part of floating-point multiplication, there is ample time for such computations. Also, rounding need not be a separate step at the end. With proper design, it may be possible to incorporate the bulk of the rounding process in the multiplication hardware.

To see how, note that most multipliers produce the least-significant half of the product earlier than the rest of the bits. So, the bits that will be used for rounding are produced early in the multiplication cycle. However, the need for normalization right shift becomes known at or near the end.

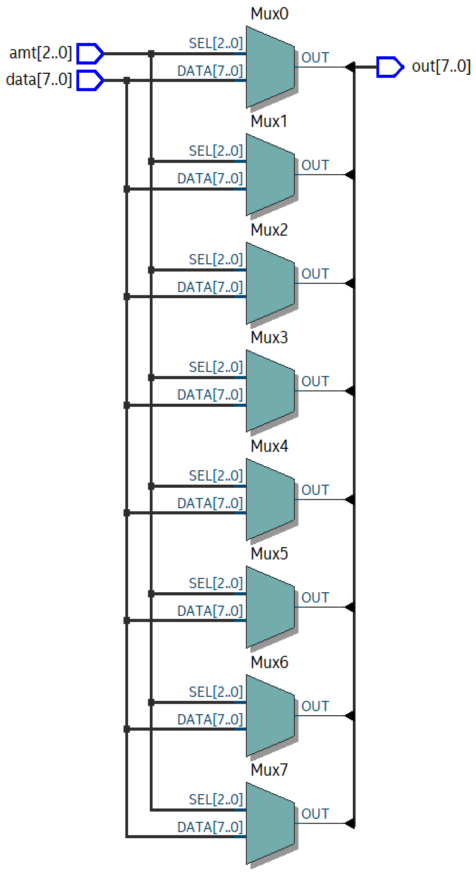
Since the significand divider's output may have to be left-shifted by 1 bit for normalization, the quotient must be developed with an extra 2 bits that serve as the guard and round bits (see the discussion of rounding for floating-point addition.

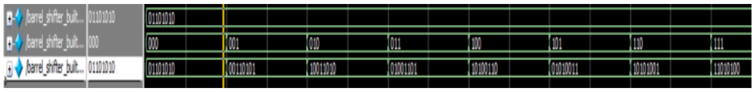
Barrel Shifters:

- A barrel shifter is a digital device that can shift a data word by a specific number of bits using just pure combinational logic and no sequential logic. It contains a control input that defines how many bit places it shifts. The Shift Register (Multi-bit) is identical to the Barrel Shifter, except that the bits moved out of the register are shifted back into the opposite end of the register. In right shift operations, for example, the LSBs relocated out of the register are shifted into the MSBs.
- Barrel shifters can be used in digital signal processors and processors. A barrel shifter can be implemented as a series of multiplexers, with the output of one multiplexer coupled to the input of the next multiplexer in a fashion that relies on the shift distance. When a barrel shifter is implemented using a series of shift multiplexers, each shifts a word by 2k bit positions (1,2,4,8,16,32...) for varying values of k. The number of stages of multiplexing is proportional to the breadth of the input vector.

Simple 8-bit right rotator:

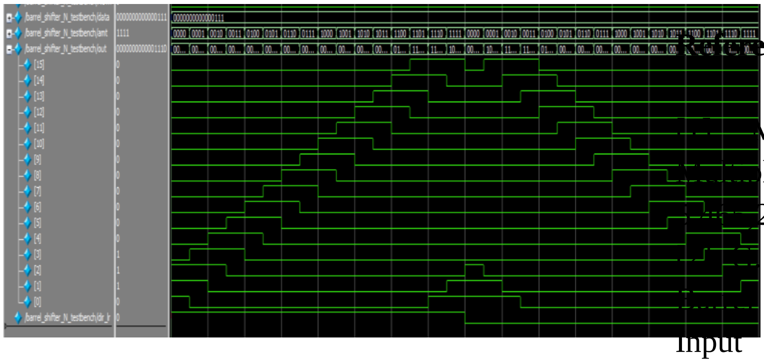
- A simple 8-bit barrel shifter that rotates and arbitrary number of bits to the right. The circuit has an 8-bit data input, data, and a 3-bit control signal, amt, which specifies the amount to be rotated.





```
/barrel_shifter_bult...  01101010  01101010
/barrel_shifter_bult...  000       000       001       010       011       100       101       110       111
/barrel_shifter_bult...  01101010  01101010  00110101  10011010  01001101  10100110  01010011  10101001  11010100
```

Multifunction Barrel Shifter:



Conclusion:

The importance and usefulness of floating-point format nowadays does not allow any discussion. Any computer or electronic device which operates with real numbers implement this type of representation and operation.

This design works with all the numbers defined by the standard: normal and subnormal. Furthermore, all the exceptions are taken into account as NaN, zero or infinity.

References:

[1] M. Seckora, \Barrel Shifter or Multiply/Divide IC Structure," U.S. Patent 5,465,222, November 1995.

[2] G. F. Burns, \Method for Generating Barrel Shifter Result Flags Directly from Input Data," U.S. Patent 6,009,451, December 1999.

[3] H. S. Lau and L. T. Ly, \Left Shift Overflow Detection," U.S. Patent 5,777,906, July 1998.

[4] M. Diamondstein and H. Srinivas, \Fast Conversion Two's Complement Encoded Shift Value for a Barrel Shifter," U.S. Patent 5,948,050, September 1999.

[5] IEEEStandard754-1985forBinary Floating-Point Arithmetic, IEEE.