Topic1: Signed number representations

In computing, **signed number representations** are required to encode negative numbers in binary number systems.

In mathematics, negative numbers in any base are represented by prefixing them with a minus sign ("−"). However, in computer hardware, numbers are represented only as sequences of bits, without extra symbols. The four best-known methods of extending the binary numeral system to represent signed numbers are: sign-and-magnitude, ones' complement, two's complement, and offset binary. Some of the alternative methods use implicit instead of explicit signs, such as negative binary, using the base −2. Corresponding methods can be devised for other bases, whether positive, negative, fractional, or other elaborations on such themes.

There is no definitive criterion by which any of the representations is universally superior. For integers, the representation used in most current computing devices is two's complement, although the Unisys ClearPath Dorado series mainframes use ones' complement.

Representation of Signed Binary Numbers

The Most Significant Bit MSBMSB of signed binary numbers is used to indicate the sign of the numbers. Hence, it is also called as **sign bit**. The positive sign is represented by placing '0' in the sign bit. Similarly, the negative sign is represented by placing '1' in the sign bit.
If the signed binary number contains 'N' bits, then N−1N−1 bits only represent the magnitude of the number since one bit MSBMSB is reserved for representing sign of the number.

There are three **types of representations** for signed binary numbers

- Sign-Magnitude form
- 1's complement form
- 2's complement form

Representation of a positive number in all these 3 forms is same. But, only the representation of negative number will differ in each form.

Example

Consider the **positive decimal number +108**. The binary equivalent of magnitude of this number is 1101100. These 7 bits represent the magnitude of the number 108. Since it is positive number, consider the sign bit as zero, which is placed on left most side of magnitude.

$$+108+108_{10} = 01101100011011002$$

Therefore, the **signed binary representation** of positive decimal number +108 is **01101100**. So, the same representation is valid in sign-magnitude form, 1's complement form and 2's complement form for positive decimal number +108.

Sign-Magnitude form

In sign-magnitude form, the MSB is used for representing **sign** of the number and the remaining bits represent the **magnitude** of the number. So, just include sign bit at the left most side of unsigned binary number. This representation is similar to the signed decimal numbers representation.

Example

Consider the **negative decimal number -108**. The magnitude of this number is 108. We know the unsigned binary representation of 108 is 1101100. It is having 7 bits. All these bits represent the magnitude.

Since the given number is negative, consider the sign bit as one, which is placed on left most side of magnitude.

$$-108-108_{10} = 11101100111011002$$

Therefore, the sign-magnitude representation of -108 is **11101100**.

1's complement form

The 1's complement of a number is obtained by **complementing all the bits** of signed binary number. So, 1's complement of positive number gives a negative number. Similarly, 1's complement of negative number gives a positive number.

That means, if you perform two times 1's complement of a binary number including sign bit, then you will get the original signed binary number.

Example

Consider the **negative decimal number -108**. The magnitude of this number is 108. We know the signed binary representation of 108 is 01101100.

It is having 8 bits. The MSB of this number is zero, which indicates positive number. Complement of zero is one and vice-versa. So, replace zeros by ones and ones by zeros in order to get the negative number.

$$-108-108_{10} = 10010011100100112$$

Therefore, the **1's complement of** 108108_{10} is 10010011100100112.

2's complement form

The 2's complement of a binary number is obtained by **adding one to the 1's complement** of signed binary number. So, 2's complement of positive number gives a negative number. Similarly, 2's complement of negative number gives a positive number.

That means, if you perform two times 2's complement of a binary number including sign bit, then you will get the original signed binary number.

Example

Consider the **negative decimal number -108**.

We know the 1's complement of $(\mathbf{108})_{10}$ is $(\mathbf{10010011})_2$

2's compliment of 108108_{10} *= 1's compliment of* 108108_{10} *+ 1.*

= 10010011 + 1

= 10010100

Therefore, the **2's complement of** 108108_{10} is 1001010010010100_2.

Topic 2 : Fixed and Floating point representations

Topic2: Fixed-Point Representation:

This representation has fixed number of bits for integer part and for fractional part. For example, if given fixed-point representation is IIII.FFFF, then you can store minimum value is 0000.0001 and maximum value is 9999.9999. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.

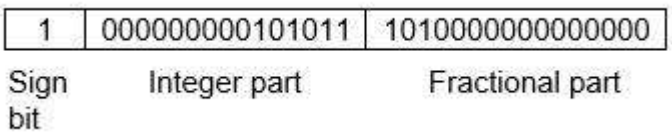
We can represent these numbers using:

- Signed representation: range from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for k bits.
- 1's complement representation: range from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for k bits.
- 2's complementation representation: range from $-(2^{(k-1)})$ to $(2^{(k-1)}-1)$, for k bits.

2's complementation representation is preferred in computer system because of unambiguous property and easier for arithmetic operations.

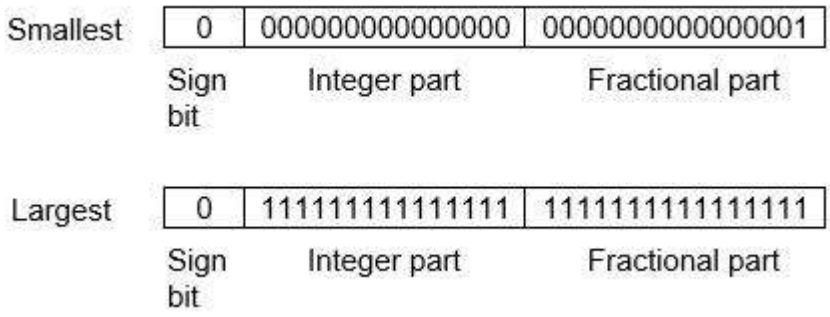
Example: Assume number is using 32-bit format which reserve 1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part.

Then, -43.625 is represented as following:



Where, 0 is used to represent + and 1 is used to represent. 000000000101011 is 15 bit binary value for decimal 43 and 1010000000000000 is 16 bit binary value for fractional 0.625.

The advantage of using a fixed-point representation is performance and disadvantage is relatively limited range of values that they can represent. So, it is usually inadequate for numerical analysis as it does not allow enough numbers and accuracy. A number whose representation exceeds 32 bits would have to be stored inexactly.



These are above smallest positive number and largest positive number which can be store in 32-bit representation as given above format. Therefore, the smallest positive number is $2^{-16} \approx 0.000015$ approximate and the largest positive number is $(2^{15}-1)+(1-2^{-16})=2^{15}(1-2^{-16}) =32768$, and gap between these numbers is 2^{-16}.

We can move the radix point either left or right with the help of only integer field is 1.

Topic3 : What is Ripple Carry Adder?

A structure of multiple full adders is cascaded in a manner to gives the results of the addition of an n bit binary sequence. This adder includes cascaded full adders in its structure so, the carry will be generated at every full adder stage in a ripple-carry adder circuit. These carry output at each full adder stage is forwarded to its next full adder and there applied as a carry input to it. This process continues up to its last full adder stage. So, each carry output bit is rippled to the next stage of a full adder. By this reason, it is named as “RIPPLE CARRY ADDER”. The most important feature of it is to add the input bit sequences whether the sequence is 4 bit or 5 bit or any.

“One of the most important point to be considered in this carry adder is the final output is known only after the carry outputs are generated by each full adder stage and forwarded to its next stage. So there will be a delay to get the result with using of this carry adder”.

There are various types in ripple-carry adders. They are:

- 4-bit ripple-carry adder
- 8-bit ripple-carry adder
- 16-bit ripple-carry adder

First, we will start with 4-bit ripple-carry-adder and then 8 bit and 16-bit ripple-carry adders.

4-bit Ripple Carry Adder

The below diagram represents the 4-bit ripple-carry adder. In this adder, four full adders are connected in cascade. Co is the carry input bit and it is zero always. When this input carry 'Co' is applied to the two input sequences A1 A2 A3 A4 and B1 B2 B3 B4 then output represented with S1 S2 S3 S4 and output carry C4..

8-bit Ripple Carry Adder

- It consists of 8 full adders which are connected in cascaded form.
- Each full adder carry output is connected as an input carry to the next stage full adder.
- The input sequences are denoted by (A1 A2 A3 A4 A5 A6 A7 A8) and (B1 B2 B3 B4 B5 B6 B7 B8) and its relevant output sequence is denoted by (S1 S2 S3 S4 S5 S6 S7 S8).
- The addition process in an 8-bit ripple-carry-adder is the same principle which is used in a 4-bit ripple-carry-adder i.e., each bit from two input sequences are going to added along with input carry.
- This will use when the addition of two 8 bit binary digits sequence.

16-bit Ripple Carry Adder

- It consists of 16 full adders which are connected in cascaded form.
- Each full adder carry output is connected as an input carry to the next stage full adder.
- The input sequences are denoted by (A1 ….. A16) and (B1 …… B16) and its relevant output sequence is denoted by (S1 …….. S16).
- The addition process in a 16-bit ripple-carry-adder is the same principle which is used in a 4-bit ripple-carry adder i.e., each bit from two input sequences are going to add along with input carry.
- This will use when the addition of two 16 bit binary digits sequence.

Topic5: Carry-lookahead adder

A carry-lookahead adder (CLA) or fast adder is a type of electronics adder used in digital logic. A carry-lookahead adder improves speed by reducing the amount of time required to determine carry bits. It can be contrasted with the simpler, but usually slower, ripple-carry adder (RCA), for which the carry bit is calculated alongside the sum bit, and each stage must wait until the previous carry bit has been calculated to begin calculating its own sum bit and carry bit. The carry-lookahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger-value bits of the adder.

A ripple-carry adder works in the same way as pencil-and-paper methods of addition. Starting at the rightmost (least significant) digit position, the two corresponding digits are added and a result obtained. It is also possible that there may be a carry out of this digit position (for example, in pencil-and-paper methods, "9 + 5 = 4, carry 1"). Accordingly, all digit positions other than the rightmost one need to take into account

the possibility of having to add an extra 1 from a carry that has come in from the next position to the right.

This means that no digit position can have an absolutely final value until it has been established whether or not a carry is coming in from the right. Moreover, if the sum without a carry is 9 (in pencil-and-paper methods) or 1 (in binary arithmetic), it is not even possible to tell whether or not a given digit position is going to pass on a carry to the position on its left. At worst, when a whole sequence of sums comes to …99999999… (in decimal) or …11111111… (in binary), nothing can be deduced at all until the value of the carry coming in from the right is known, and that carry is then propagated to the left, one step at a time, as each digit position evaluated "9 + 1 = 0, carry 1" or "1 + 1 = 0, carry 1". It is the "rippling" of the carry from right to left that gives a ripple-carry adder its name, and its slowness. When adding 32-bit integers, for instance, allowance has to be made for the possibility that a carry could have to ripple through every one of the 32 one-bit adders.

Carry-lookahead depends on two things:

1. Calculating for each digit position whether that position is going to propagate a carry if one comes in from the right.
2. Combining these calculated values to be able to deduce quickly whether, for each group of digits, that group is going to propagate a carry that comes in from the right.

Topic6: Booth's algorithm

Booth's Multiplication Algorithm

Booth's algorithm is a multiplication algorithm that multiplies two signed binary numbers in 2's compliment notation.

PROCEDURE:

1. Let M is the multiplicand.
2. Let Q is the multiplier.
3. Consider a 1-bit register Q_{-1} and initialize it to 0.
4. Consider a register A and initialize it to 0.

CONDITIONS:

1. If $Q_0\,Q_{-1}$ are same i.e. 00 or 11 then, perform arithmetic right shift by 1 bit.
2. If $Q_0\,Q_{-1}$ = 10 then perform
   A←A-M
   And then perform arithmetic right shift.
3. If $Q_0\,Q_{-1}$ = 01 then perform
   A←A+M
   And then perform arithmetic right shift.

For example:

Consider two numbers 6 and 2 and we have to perform their multiplication by using Booth's algorithm.

Here 6 is multiplicand (M) and 2 is multiplier (Q).

Now write 6 and 2 in binary form.

M = 6 = 0 1 1 0

Q = 2 = 0 0 1 0 (Q_3,Q_2,Q_1,Q_0)

Booth's algorithm calculates the product in n steps where n is the number of bits used to represent the numbers.

INITIALISE	A	B	Q_{-1}	OPERATIONS
	0 0 0 0 ↓↘↘↘↘	0 0 1 0 ↘↘↘↘	0	
Step 1.	0 0 0 0	0 0 0 1 ↓	0 ↓	Arithmetic right shift
Step 2.	1 0 1 0 ↓↘↘↘↘ 1 1 0 1	0 0 0 1 ↘↘↘↘ 0 0 0 0	0 1	A←A-M Then shift
Step 3.	0 0 1 1 ↓↘↘↘↘ 0 0 0 1 ↓↘↘↘↘	0 0 0 0 ↘↘↘↘ 1 0 0 0 ↘↘↘↘	1 0	A←A+M Then shift
Step 4.	0 0 0 0	1 1 0 0 In binary, 12 = 1100 Hence 3*2 = 12	0	Arithmetic right shift

Carry Save Array Multiplier The carry-save array multiplier uses an array of carry-save adders for the accumulation of partial product. It uses a carry-propagate adder for the generation of the final product. This reduces the critical path delay of the multiplier since the carry-save adders pass the carry to the next level of adders rather than the adjacent ones.

Topic7 : Division restoring and non-restoring techniques

Restoring Division Algorithm For Unsigned Integer

A division algorithm provides a quotient and a remainder when we divide two number. They are generally of two type **slow algorithm and fast algorithm**. Slow division algorithm are restoring, non-restoring, non-performing restoring, SRT algorithm and under fast comes Newton–Raphson and Goldschmidt.
In this article, will be performing restoring algorithm for unsigned integer. Restoring term is due to fact that value of register A is restored after each iteration.

Let's pick the step involved:

- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Then the content of register A and Q is shifted left as if they are a single unit
- **Step-3:** Then content of register M is subtracted from A and result is stored in A
- **Step-4:** Then the most significant bit of the A is checked if it is 0 the least significant bit of Q is set to 1 otherwise if it is 1 the least significant bit of Q is set to 0 and value of register A is restored i.e the value of A before the subtraction with M
- **Step-5:** The value of counter n is decremented
- **Step-6:** If the value of n becomes zero we get of the loop otherwise we repeat from step 2
- **Step-7:** Finally, the register Q contain the quotient and A contain remainder

Non-Restoring Division For Unsigned Integer

In earlier post Restoring Division learned about restoring division. Now, here perform Non-Restoring division, it is less complex than the restoring one because simpler operation are involved i.e. addition and subtraction, also now restoring step is performed. In the method, rely on the sign bit of the register which initially contain zero named as A.

Let's pick the step involved:

- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Check the sign bit of register A
- **Step-3:** If it is 1 shift left content of AQ and perform A = A+M, otherwise shift left AQ and perform A = A-M (means add 2's complement of M to A and store it to A)
- **Step-4:** Again the sign bit of register A
- **Step-5:** If sign bit is 1 Q[0] become 0 otherwise Q[0] become 1 (Q[0] means least significant bit of register Q)
- **Step-6:** Decrements value of N by 1
- **Step-7:** If N is not equal to zero go to **Step 2** otherwise go to next step
- **Step-8:** If sign bit of A is 1 then perform A = A+M
- **Step-9:** Register Q contain quotient and A contain remainder