Chapter 5: Computer Arithmetic

Now that binary numbers are discussed, it's time to look at how their arithmetic works. But there are two basic binary number representations so far. **Unsigned binary numbers** and binary numbers that are represented using the **sign and magnitude method**.

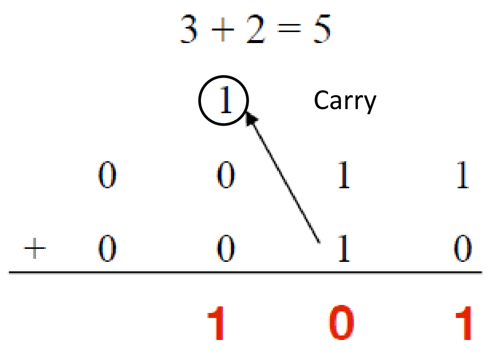
⇨ Addition and subtraction for unsigned binary integers

1. Addition Rule

1	0	+	1
0	1		1
0	0		0
1	1		10

0 with a carry of 1

Example:



2. Subtraction Rule

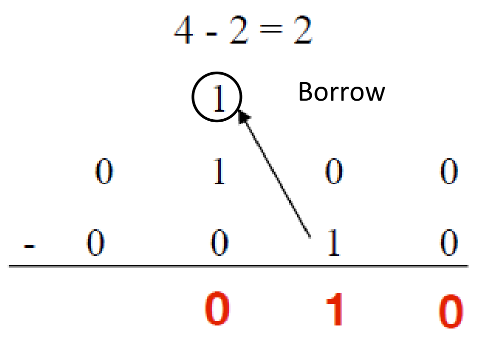
1	0	-	1
1	1		0
0	0		0
0	1		1

1 with a borrow of 1

Making it

[10-1=1]

Example:



⇨ Addition & subtraction for signed binary integers (using sign and magnitude)

If the binary numbers are signed then addition and subtraction can be expressed using just the addition operation. Take a look at the following equations.

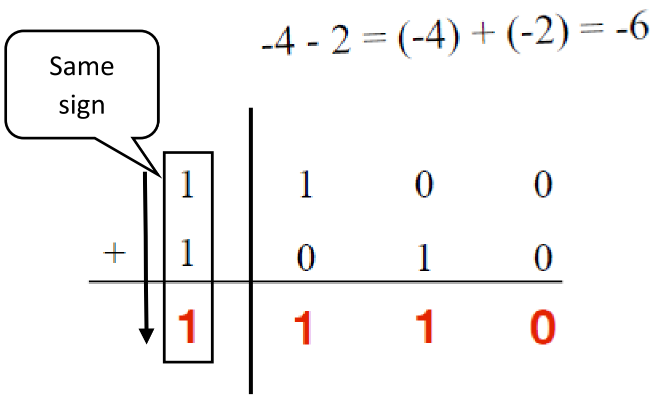
- X – Y = X + (- Y)
- - X – Y = (- X) + (- Y)

Recap: The first bit of the binary number indicates sign, 0 for positive and 1 for negative

Rules:

1. Identify the sign of each binary number [**the sign bit is excluded for the calculation**]
   A. If they both have the same sign
      - Add the two binaries.
      - Apply the sign to the result.

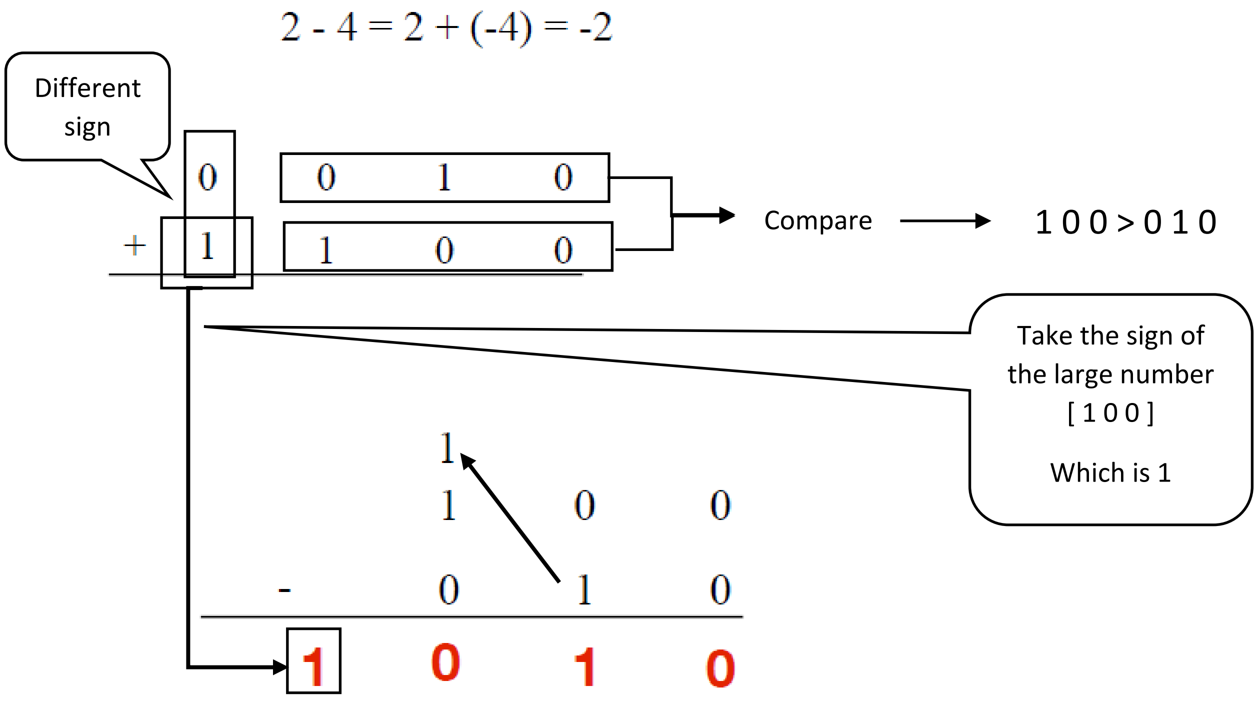
Example:



B. If they have different signs

- Compare the numbers
- Subtract small from large.
- Apply the sign of the large number to the result.

Example:



Note: ***When calculating binary addition for integers represented using sign and magnitude, overflow may occur.***

0101 + 0110 = 1011 => 5 + 6 = -5 [Discuss Correction]

⇨ Multiplication and Division for unsigned binary integers

Multiplication Rules:

1	0	*	0
0	1		0
0	0		0
1	1		1

Multiplication for unsigned integers is straight forward,

- Find out the number of bits needed **n**
- Represent both numbers with **n**
- apply decimal multiplication rule
- apply the unsigned binary addition rule
- cut-off unnecessary bits

Example:

4 * 2 = 8

Number of bits needed -> the immediate next 2^n for 8 is $16=2^4$ so <u>4 bits</u>

$$\begin{array}{cccccccccc}
 & 0 & 1 & 0 & 0 & * & 0 & 0 & 1 & 0 \\
\hline
 & & & 0 & 0 & 0 & 0 & & & \\
 & & 0 & 1 & 0 & 0 & & & & \\
 & 0 & 0 & 0 & 0 & & & & & \\
0 & 0 & 0 & 0 & & & & & & \\
\hline
\multicolumn{3}{c}{\boxed{0\ \ 0\ \ 0}} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & & &
\end{array}$$

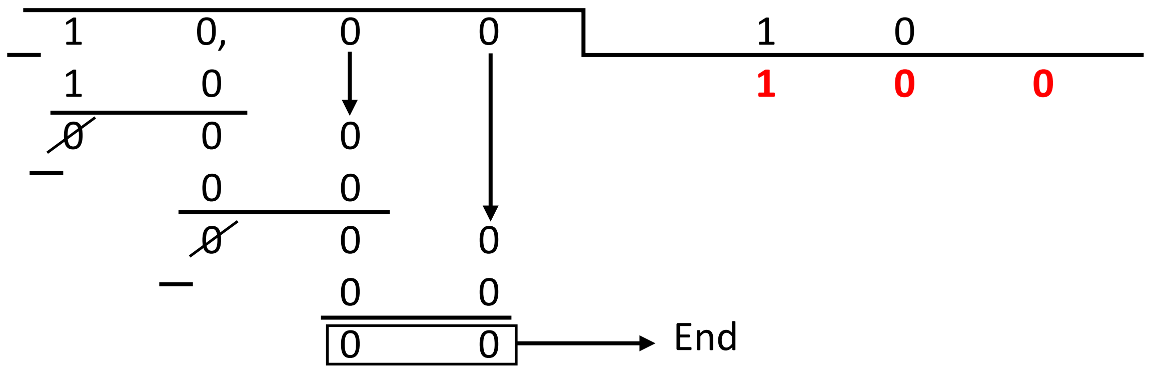
Cut off

Division Rules:

- Assumption
    - **n**= number of bits of the divisor **db1**
    - **db2**= dividend
- cut **n** from **db2** to create a smaller binary **b**
- if **b** is less than **db1** then multiply **db1** by 1
- if **b** is greater than **db1** then multiply **db1** by 0
- subtract and continue until the dividend is exhausted
    - apply the subtraction rule for unsigned integers
    - trim unnecessary bits

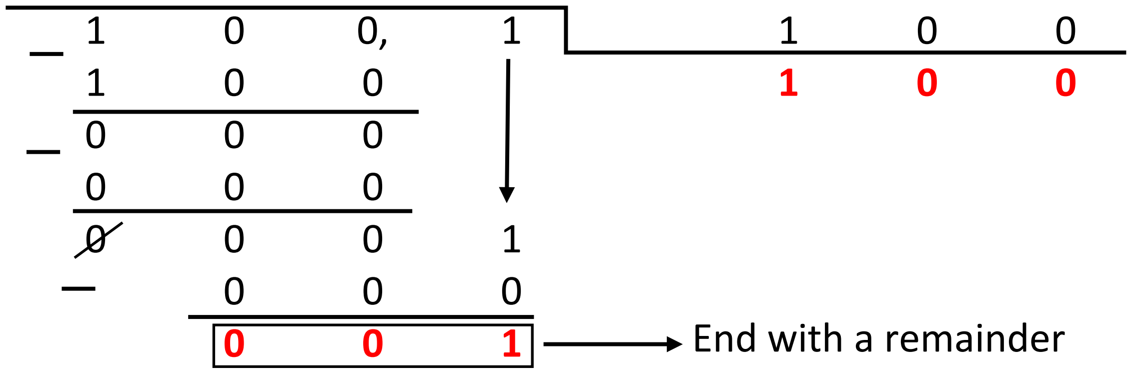
Example 1:

8 / 2 = 4



Example 1:

9 / 4 = 2 with a reminder of 1



⇨ **Multiplication and Division for signed binary integers (using sign and magnitude)**

Rules:

- Look at the sign bits of each binary integers
- Exclude the sign bits from the calculation
- Perform the normal multiplication/division
    - If the signs are **different** assign **1** as the sign bit for the result
    - If the signs are **the same** assign **0** as the sign bit for the result

⇨ Binary Complements

Complements are used to represent negative integers in a binary form. Positive integers are represented in the normal binary from.

- **1’s complement**

Simply convert 0s to 1s and 1s to 0s.

$()_{10}$	$()_2$	Notice that the negative integers always start with 1 and the positive once start with a 0	1’s C $()_2$	$()_{10}$
0	000		111	0
+1	001		110	-1
+2	010		101	-2
+3	011		100	-3
+4	010		101	-4
+5	0101		1010	-5

Take +1 (001), to find -1 all that is needed to be done to convert 0s to 1s and 1s to 0s so

$$\begin{tikzcd}[column sep=tiny] 0 \arrow[d] & 0 \arrow[d] & 1 \arrow[d] \\ 1 & 1 & 0 \end{tikzcd}$$

- **Subtraction using 1’s complement**

Since every negative integer can be represented uniquely subtraction can be replaced with addition. Like previously X – Y = X + (-Y).

Rule: If there is a remainder at the end it’s added to the result.

Example:

$4 - 2 = 4 + (-2) = 2$

$(4)_{10} == (0100)_2$ $\quad$ $-(2)_{10} = -(\,(2)_{10}\,) == -(\,(0010)_2\,) = (1101)_2$

$$\begin{array}{rr} \bigcirc\!\!\!\!\!1 & 1 \\ & 0100 \\ + & 1101 \\ \hline & 0001 \\ + & 1 \\ \hline & \textcolor{red}{\mathbf{0010}} \end{array}$$

Last Carry

This Method is much better and more effective than the sign-magnitude method because there is no need to isolate the sign bit making the calculation more natural. So It’s mathematically less complicated.

Discussion: What problems do you observe in 1’s complement and it’s subtraction? [Use -2-2]

- **2’s compliment**

This is a more refined version of 1’s compliment. It solves the problem of 1’s complement.

To find the 2’s complement of a negative integer, first find the 1’s complement the add 1.

$()_{10}$	$()_2$	1's C $()_2$	2's C $()_2$	$()_{10}$
0	000	111	000	0
+1	001	110	111	-1
+2	010	101	110	-2
+3	011	100	101	-3
+4	010	101	110	-4
+5	0101	1010	1011	-5

Discussion: How does 2's complement solve the problem of 1's complement?

- **Subtraction using 2's complement**

Rule: If there is a remainder at the end it's dropped and the process ends.

Example:

$4 - 2 = 4 + (-2) = 2$

$(4)_{10} == (0100)_2$ $\qquad$ $-(2)_{10} = -(\ (2)_{10}\ ) == -(\ (0010)_2\ \ ) = (1101 + 1)_2$

$$\begin{array}{lr} \cancel{1} & 1 \\ & 0\,1\,0\,0 \\ \text{Drop} \downarrow \quad + & 1\,1\,1\,0 \\ \hline & 0\,0\,1\,0 \end{array}$$

$$\begin{array}{r} 1 \\ 1\,1\,0\,1 \\ +\qquad 1 \\ \hline 1\,1\,1\,0 \end{array}$$

⇨ **Multiplication using 1's complement**

Observe the following example:

$2 * -2 = -4$ using 4 bits

$$\begin{array}{ccccccccc} 0 & 0 & 1 & 0 & * & 1 & 1 & 0 & 1 \\ \hline & & & & & 0 & 0 & 1 & 0 \\ & & & & 0 & 0 & 0 & 0 & \\ & & & 0 & 0 & 1 & 0 & & + \\ & & 0 & 0 & 1 & 0 & & & \\ \hline & & 0 & 0 & 1 & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{0} \end{array}$$

This is incorrect. To check perform –(-x)=x using 1's complement -(1010)=(0101)=5 but -2*2 = -4 and –(-4)=4 not 5

Discussion: Why is this happening? How can it be solved?

⇨ Multiplication using 2’s complement

The benefit of 2’s complement is that the mathematics is very straight forward, hence applying the normal binary multiplication is enough to get the accurate result.

2 * -2 = -4 0010 * 1110

$$
\begin{array}{ccccccccc c}
0 & 0 & 1 & 0 & * & 1 & 1 & 1 & 0 & \\
\hline
 & & & & & 0 & 0 & 0 & 0 & \\
 & & & & 0 & 0 & 1 & 0 & & \\
 & & & 0 & 0 & 1 & 0 & & & + \\
 & & 0 & 0 & 1 & 0 & & & & \\
\cline{3-9}
 & & 0 & 0 & 1 & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{0} &
\end{array}
$$

⇨ Division using 2’s complement

Observe the following statements.

X / -Y = -X / Y = -(X / Y)
-X / -Y = X / Y

With these rules set, division can be interpreted to subtraction. For example: 16/4 means

16 – 4 = 12 =========> 1 step

12 – 4 = 8 ==========> 1 step

8 – 4 = 4 ===========> 1 step

4 – 4 = 0 ===========> 1 step

There are **4** steps needed to reach 0, which indicates the end of the process. Hence 16/4 = 4

Rules:

- Check the sign of each binary.
- Apply the above rule
- Perform 2’s comp’ subtraction.

Example:

16 / -4 using 6 bits.

- (16 / 4) = -(4) 16 = 0 1 0 0 0 0 4 = 0 0 0 1 0 0

Translate division to subtraction in 2’s complement.

0 1 0 0 0 0 – (0 0 0 1 0 0) == 0 1 0 0 0 0 + 1 1 1 1 0 0 = **0 0 1 1 0 0**

0 0 1 1 0 0 – (0 0 0 1 0 0) == 0 0 1 1 0 0 + 1 1 1 1 0 0 = **0 0 1 0 0 0**

0 0 1 0 0 0 – (0 0 0 1 0 0) == 0 0 1 0 0 0 + 1 1 1 1 0 0 = **0 0 0 1 0 0**

0 0 0 1 0 0 – (0 0 0 1 0 0) == 0 0 0 1 0 0 + 1 1 1 1 0 0 = $\boxed{\mathbf{0\,0\,0\,0\,0\,0}} \longrightarrow$ End

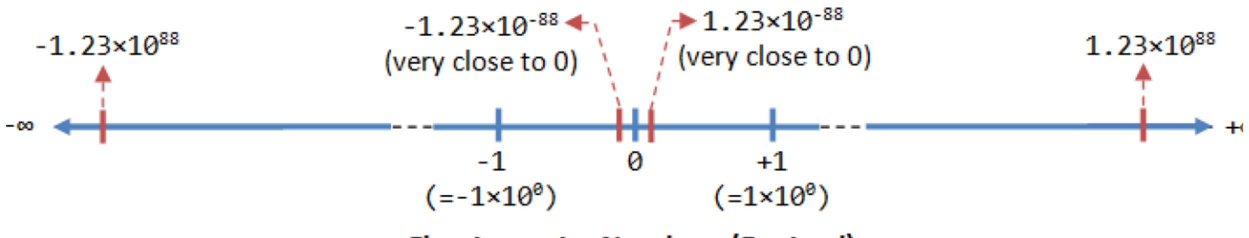
It took 4 steps to reach 0 hence the quotient 4 [0 0 0 1 0 0] but the sign of the result is negative as established at the beginning so

– (0 0 0 1 0 0) so **1 1 1 1 0 0**

Discussion: How would 2's complement division work for integers that divide to give out a remainder? [Use 7 / -3]

⇨ Floating-Point Number Representation

A floating-point number (or real number) can represent a very large (1.23×10^88) or a very small (1.23×10^-88) value. It could also represent very large negative number (-1.23×10^88) and very small negative number (-1.23×10^88), as well as zero, as illustrated:



Floating-point Numbers (Decimal)

A floating-point number is typically expressed in the scientific notation, with a fraction (F), and an exponent (E) of a certain radix (r), in the form of F×r^E. Decimal numbers use radix of 10 (F×10^E); while binary numbers use radix of 2 (F×2^E).

Representation of floating-point number is not unique. For example, the number 55.66 can be represented as 5.566×10^1, 0.5566×10^2, 0.05566×10^3, and so on. The fractional part can be normalized. In the normalized form, there is only a single non-zero digit before the radix point. For example, decimal number 123.4567 can be normalized as 1.234567×10^2; binary number 1010.1011B can be normalized as 1.0101011B×2^3.

It is important to note that floating-point numbers suffer from loss of precision when represented with a fixed number of bits (e.g., 32-bit or 64-bit). This is because there are infinite number of real numbers (even within a small range of says 0.0 to 0.1). On the other hand, a n-bit binary pattern can represent a finite 2^n distinct numbers. Hence, not all the real numbers can be represented. The nearest approximation will be used instead, resulted in loss of accuracy. It is also important to note that floating number arithmetic is very much less efficient than integer arithmetic. It could be speed up with a so-called dedicated floating-point co-processor. Hence, use

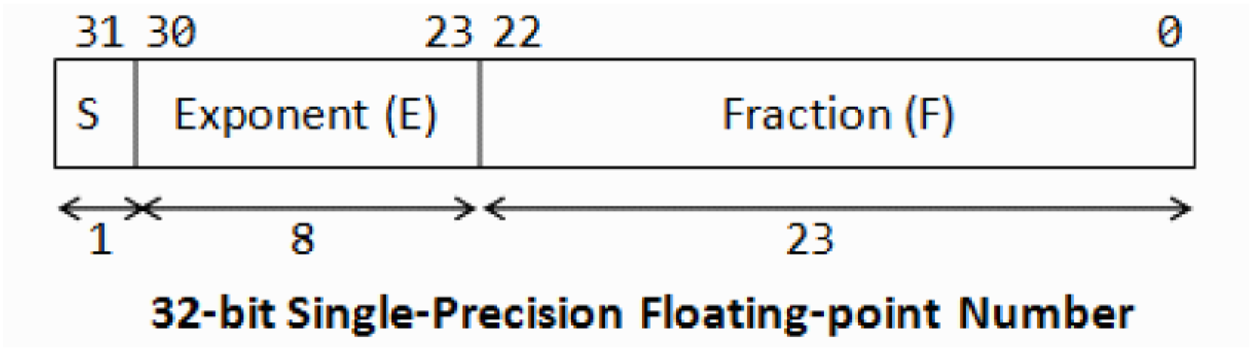
integers if your application does not require floating-point numbers. In computers, floating-point numbers are represented in scientific notation of fraction (F)

and exponent (E) with a radix of 2, in the form of F×2^E. Both E and F can be positive as well as negative. Modern computers adopt IEEE 754 standard for representing floating-point numbers. There are two representation schemes: 32-bit single-precision and 64-bit double-precision.

- **IEEE-754 32-bit Single-Precision Floating-Point Numbers**

In 32-bit single-precision floating-point representation:

- The most significant bit is the sign bit (S), with 0 for positive numbers and 1 for negative numbers.
- The following 8 bits represent exponent (E).
- The remaining 23 bits represents fraction (F).



- **Normalized Form**

Let's illustrate with an example, suppose that the 32-bit pattern is 1 1000 0001 011 0000 0000 0000 0000 0000, with:

- S = 1
- E = 1000 0001
- F = 011 0000 0000 0000 0000 0000

In the normalized form, the actual fraction is normalized with an implicit leading 1 in the form of 1.F. In this example, the actual fraction is 1.011 0000 0000 0000 0000 0000 = 1 + 1×2^-2 + 1×2^-3 = 1.375D. The sign bit represents the sign of the number, with S=0 for positive and S=1 for negative number. In this example with S=1, this is a negative number, i.e., -1.375D. In normalized form, the actual exponent is E-127 (so-called excess-127 or bias-127). This is because we need to represent both positive and negative exponent. With an 8-bit E, ranging from 0 to 255, the excess-127 scheme could provide actual exponent of -127 to 128.

In this example, E-127=129-127=2D.

Hence, the number represented is -1.375×2^2=-5.5D