2.2 Signed and unsigned numbers

First, let's quickly review how a computer represents numbers. Humans are taught to think in base 10, but numbers may be represented in any base. For example, 123 base 10 = 1111011 base 2.

Numbers are kept in computer hardware as a series of high and low electronic signals, and so they are considered base 2 numbers. (Just as base 10 numbers are called decimal numbers, base 2 numbers are called binary numbers.)

A single digit of a binary number is thus the "atom" of computing, since all information is composed of *binary digits* or bits. This fundamental building block can be one of two values, which can be thought of as several alternatives: high or low, on or off, true or false, or 1 or 0.

Binary digit: Also called *binary bit*. One of the two numbers in base 2, 0 or 1, that are the components of information.

Generalizing the point, in any number base, the value of *i* th digit *d* is

$$d \times \text{Base}^i$$

where *i* starts at 0 and increases from right to left. This representation leads to an obvious way to number the bits in the doubleword: simply use the power of the base for that bit. We subscript decimal numbers with *ten* and binary numbers with *two*.

PARTICIPATION ACTIVITY 2.2.1: A base two number's value in base ten.

Start ☐ 2x speed

1011_{two}

$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{ten}$

$= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{ten}$

$= 8 + 0 + 2 + 1_{ten}$

$= 11_{ten}$

PARTICIPATION ACTIVITY 2.2.2: Binary number tool.

0	0	0	0	0	0	0	0	**0**
128	**64**	**32**	**16**	**8**	**4**	**2**	**1**	(decimal value)
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	

PARTICIPATION ACTIVITY 2.2.3: Base two and base ten numbers.

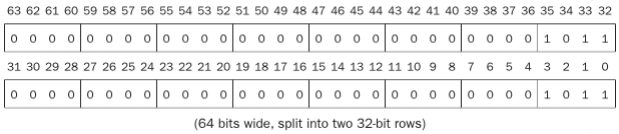
1) What is 1110_{two} in base ten?

Check **Show answer**

2) What is 3_{ten} as a *4-bit* base two number?

Check **Show answer**

We number the bits 0, 1, 2, 3, … from *right to left* in a doubleword. The drawing below shows the numbering of bits within an LEGv8 doubleword and the placement of the number 1011_{two} , (which we must unfortunately split in half to fit on the page):



(64 bits wide, split into two 32-bit rows)

Since doublewords are drawn vertically as well as horizontally, leftmost and rightmost may be unclear. Hence, the phrase *least significant bit* is used to refer to the rightmost bit (bit 0 above) and *most significant bit* to the leftmost bit (bit 63).

Least significant bit: The rightmost bit in an LEGv8 doubleword.

Most significant bit: The leftmost bit in an LEGv8 doubleword..

PARTICIPATION ACTIVITY 2.2.4: 64-bit doublewords.

1) Of a doubleword's 64 bits, what is the leftmost bit numbered?

- ○ 64
- ○

63

2) Given the following 64-bit number, what is the most significant bit's value?
1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
   - 1
   - 0

The LEGv8 doubleword is 64 bits long, so we can represent 2^{64} different 64-bit patterns. It is natural to let these combinations represent the numbers from 0 to $2^{64} - 1$ (18,446,744,073,709,551,615$_{ten}$):

$$\begin{aligned}
&00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}\\
&00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}\\
&00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}\\
&\quad\ldots\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ldots\\
&11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = 18,446,744,073,709,551,613_{ten}\\
&11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = 18,446,744,073,709,551,614_{ten}\\
&11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = 18,446,744,073,709,551,615_{ten}
\end{aligned}$$

That is, 64-bit binary numbers can be represented in terms of the bit value times a power of 2 (here *xi* means the *i* th bit of *x*):

$$(x_{63} \times 2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \ldots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

For reasons we will shortly see, these positive numbers are called unsigned numbers.

PARTICIPATION ACTIVITY 2.2.5: Unsigned numbers.

1) What is the largest base ten number representable in 4 bits (assuming the "natural" representation)?
   - 8
   - 15
   - 16

2) What is the largest base ten number representable in 8 bits (assuming the "natural" representation)?
   - 255
   - 256

3) What is the largest base ten number approximately representable by 32 bits (assuming the "natural" representation)?
   - 4 million
   - 4 billion
   - 4 trillion

4) How is the largest base ten number representable by 64 bits calculated (assuming the "natural" representation)?
   - $2^{63} - 1$
   - 2^{64}
   - $2^{64} - 1$

Hardware/Software Interface

Base 2 is not natural to human beings; we have 10 fingers and so find base 10 natural. Why didn't computers use decimal? In fact, the first commercial computer *did* offer decimal arithmetic. The problem was that the computer still used on and off signals, so a decimal digit was simply represented by several binary digits. Decimal proved so inefficient that subsequent computers reverted to all binary, converting to base 10 only for the relatively infrequent input/output events.

Keep in mind that the binary bit patterns above are simply *representatives* of numbers. Numbers really have an infinite number of digits, with almost all being 0 except for a few of the rightmost digits. We just don't normally show leading 0s.

Hardware can be designed to add, subtract, multiply, and divide these binary bit patterns. If the number that is the proper result of such operations cannot be represented by these rightmost hardware bits, ***overflow*** is said to have occurred. It's up to the programming language, the operating system, and the program to determine what to do if overflow occurs.

Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative. The most obvious solution is to add a separate sign, which conveniently can be represented in a single bit. A ***sign and magnitude***

representation is a signed number representation where a single bit is used to represent the sign, and the remaining bits represent the the magnitude.

Alas, sign and magnitude representation has several shortcomings. First, it's not obvious where to put the sign bit. To the right? To the left? Early computers tried both. Second, adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the proper sign will be. Finally, a separate sign bit means that sign and magnitude has both a positive and a negative zero, which can lead to problems for inattentive programmers. Because of these shortcomings, sign and magnitude representation was soon abandoned.

In the search for a more attractive alternative, the question arose as to what would be the result for unsigned numbers if we tried to subtract a large number from a small one. The answer is that it would try to borrow from a string of leading 0s, so the result would have a string of leading 1s.

Given that there was no obvious better alternative, the final solution was to pick the representation that made the hardware simple: leading 0s mean positive, and leading 1s mean negative. This convention for representing signed binary numbers is called *two's complement* representation:

$$\begin{aligned}
&00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}\\
&00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}\\
&00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}\\
&\quad\ldots\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ldots\\
&01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = 9,223,372,036,854,775,805_{ten}\\
&01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = 9,223,372,036,854,775,806_{ten}\\
&01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = 9,223,372,036,854,775,807_{ten}\\
&10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = -9,223,372,036,854,775,808_{ten}\\
&10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = -9,223,372,036,854,775,807_{ten}\\
&10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = -9,223,372,036,854,775,806_{ten}\\
&\quad\ldots\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ldots\\
&11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = -3_{ten}\\
&11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = -2_{ten}\\
&11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = -1_{ten}
\end{aligned}$$

The positive half of the numbers, from 0 to $9,223,372,036,854,775,807_{ten}$ ($2^{63} - 1$), use the same representation as before. The following bit pattern ($1000\ldots0000_{two}$) represents the most negative number $-9,223,372,036,854,775,808_{ten}$ (-2^{63}). It is followed by a declining set of negative numbers: $-9,223,372,036,854,775,807_{ten}$ ($1000\ldots0001_{two}$) down to -1_{ten} ($1111\ldots1111_{two}$).

Two's complement: A signed number representation where a leading 0 indicates a positive number and a leading 1 indicates a negative number. The complement of a value is obtained by complementing each bit (0 → 1 or 1 → 0), and then adding one to the result (explained further below).

Two's complement does have one negative number that has no corresponding positive number: $-9,223,372,036,854,775,808_{ten}$. Such imbalance was also a worry to the inattentive programmer, but sign and magnitude had problems for both the programmer and the hardware designer. Consequently, every computer today uses two's complement binary representations for signed numbers.

Two's complement representation has the advantage that all negative numbers have a 1 in the most significant bit. Thus, hardware needs to test only this bit to see if a number is positive or negative (with the number 0 is considered positive). This bit is often called the sign bit. By recognizing the role of the sign bit, we can represent positive and negative 64-bit numbers in terms of the bit value times a power of 2:

$$(x_{63} \times -2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \ldots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

The sign bit is multiplied by -2^{63}, and the rest of the bits are then multiplied by positive versions of their respective base values.

Example 2.2.1: Binary to decimal conversion (two's complement representation).

What is the decimal value of this 64-bit two's complement number?

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 1111110$$

Answer

Substituting the number's bit values into the formula above:

$$\begin{aligned}
&(1 \times -2^{63}) + (1 \times 2^{62}) + (1 \times 2^{61}) + \ldots + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)\\
&= -2^{63} + 2^{62} + 2^{61} + \ldots + 2^2 + 0 + 0\\
&= -9,223,372,036,854,775,808_{ten} + 9,223,372,036,854,775,804_{ten}\\
&= -4_{ten}
\end{aligned}$$

We'll see a shortcut to simplify conversion from negative to positive soon.

PARTICIPATION ACTIVITY 2.2.6: Two's complement representation.

1) Sign and magnitude representation and two's complement representation are used about equally in modern computers.
   - ○ True

- False

2) In two's complement, is the following number positive or negative?
11110000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
   - Positive
   - Negative

3) In two's complement, is the following number positive or negative?
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001111
   - Positive
   - Negative

4) Knowing that 2^{63} is 9,223,372,036,854,775,808, what is the base ten value of the following two's complement number?
10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
   - -9,223,372,036,854,775,808
   - -1

5) How is 0 represented in two's complement?
All 0's: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
or
All 1's: 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
   - All 0's
   - All 1's

6) In a two's complement representation, the magnitude of the largest negative value is one greater than the magnitude of the largest positive number.
   - True
   - False

Just as an operation on unsigned numbers can overflow the capacity of hardware to represent the result, so can an operation on two's complement numbers. Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits to the left (the sign bit is incorrect): a 0 on the left of the bit pattern when the number is negative or a 1 when the number is positive.

PARTICIPATION ACTIVITY 2.2.7: Two's complement: Overflow.

Indicate if the binary operation resulted in overflow. The numbers presented are 32-bit values; 64-bit values do not fit in the space, but the concepts are identical no matter the number of bits.

1)

```
  1000 1111 0000 0000 0000 0000 0000 0000
+ 1000 0000 1111 1111 1111 1111 1111 0000
-----------------------------------------
  0000 1111 1111 1111 1111 1111 1111 0000
```

   - Overflow
   - No overflow

2)

```
  0000 1111 0000 0000 0000 0000 0000 0000
+ 0111 0000 0000 0000 0000 0000 0000 0000
-----------------------------------------
  0111 1111 0000 0000 0000 0000 0000 0000
```

   - Overflow
   - No overflow

3)

```
  0111 0000 0000 0000 0000 0000 0000 0000
```

```
+ 1111 0000 0000 0000 0000
0000 0000 0000
------------------------------
------------
  0110 0000 0000 0000 0000
0000 0000 0000
```

- ○ Overflow
- ○ No overflow

Hardware/Software Interface

Signed versus unsigned applies to loads as well as to arithmetic. The function of a signed load is to copy the sign repeatedly to fill the rest of the register, known as a ***sign extension***. The purpose of a signed load is to place a correct representation of the number within that register. Unsigned loads simply fill with 0s to the left of the data, since the number represented by the bit pattern is unsigned.

When loading a 64-bit doubleword into a 64-bit register, the point is moot; signed and unsigned loads are identical. ARMv8 does offer two flavors of byte loads: *load byte* (`LDURB`) treats the byte as an unsigned number and thus zero-extends to fill the leftmost bits of the register, while *load byte signed* (`LDURSB`) works with signed integers. Since C programs almost always use bytes to represent characters rather than consider bytes as very short signed integers, `LDURB` is used practically exclusively for byte loads.

Hardware/Software Interface

Unlike the signed numbers discussed above, memory addresses naturally start at 0 and continue to the largest address. Put another way, negative addresses make no sense. Thus, programs want to deal sometimes with numbers that can be positive or negative and sometimes with numbers that can be only positive. Some programming languages reflect this distinction. C, for example, names the former integers (declared as `long long int` in the program) and the latter *unsigned integers* (`unsigned long long int`). Some C style guides even recommend declaring the former as `signed long long int` to keep the distinction clear.

Let's examine two useful shortcuts when working with two's complement numbers. The first shortcut is a quick way to negate a two's complement binary number. Simply invert every 0 to 1 and every 1 to 0, then add one to the result. This shortcut is based on the observation that the sum of a number and its inverted representation must be 111 … 111_{two}, which represents −1. Since $x + \bar{x} = -1$, therefore $x + \bar{x} + 1 = 0$ or $\bar{x} + 1 = -x$. (We use the notation $\bar{x}$ to mean invert every bit in x from 0 to 1 and vice versa.)

Example 2.2.2: Negation shortcut.

Negate 2_{ten}, and then check the result by negating -2_{ten}.

Answer

$$2_{ten} = 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 000$$

Negating this number by inverting the bits and adding one,

$$\begin{array}{ll} & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 1 \\ + & \\ \hline = & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 1 \\ = & -2_{ten} \end{array}$$

Going the other direction,

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{t}$$

is first inverted and then incremented:

$$\begin{array}{ll} & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 0 \\ + & \\ \hline = & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 0 \\ = & 2_{ten} \end{array}$$

PARTICIPATION ACTIVITY 2.2.8: Negation shortcut for two's complement representation.

1) If $+3_{ten}$ is 00000011_{two}, what is -3_{ten} in an 8-bit two's complement representation?

Check Show answer

2) What is -9_{ten} in an 8-bit two's complement representation?

Check Show answer

3) Assuming a 64-bit two's-complement representation, what is 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111100$_{two}$ in base ten?

Check Show answer

Our next shortcut tells us how to convert a binary number represented in *n* bits to a number represented with more than *n* bits. The shortcut is to take the most significant bit from the smaller quantity—the sign bit—and replicate it to fill the new bits of the larger quantity. The old nonsign bits are simply copied into the right portion of the new doubleword. This shortcut is commonly called *sign extension*.

PARTICIPATION ACTIVITY 2.2.9: Example of sign extension shortcut.

Start 2x speed

0000 0000 0000 0000 0000 0000 0000 0010 $_{two}$ = 2 $_{ten}$

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0010 $_{two}$ = 2 $_{ten}$

1111 1111 1111 1111 1111 1111 1111 1110 $_{two}$ = -2 $_{ten}$

1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110 $_{two}$ = -2 $_{ten}$

This trick works because positive two's complement numbers really have an infinite number of 0s on the left and negative two's complement numbers have an infinite number of 1s. The binary bit pattern representing a number hides leading bits to fit the width of the hardware; sign extension simply restores some of them.

PARTICIPATION ACTIVITY 2.2.10: Negation shortcut for two's complement representation.

1) Given -5 in 8-bit two's complement: 11111011.
Extending to 16 bits yields: _____

- ○ 00000000 11111011
- ○ 11111111 11111011

Summary

The main point of this section is that we need to represent both positive and negative integers within a computer, and although there are pros and cons to any option, the unanimous choice since 1965 has been two's complement.

Elaboration

For signed decimal numbers, we used "–" to represent negative because there are no limits to the size of a decimal number. Given a fixed data size, binary and hexadecimal bit strings can encode the sign; therefore, we do not normally use "+" or "–" with binary or hexadecimal notation.

Elaboration

Two's complement gets its name from the rule that the unsigned sum of an n-bit number and its n-bit negative is 2^n; hence, the negation or complement of a number x is $2^n - x$, or its "two's complement."

A third alternative representation to two's complement and sign and magnitude is called one's complement. The negative of a one's complement is found by inverting each bit, from 0 to 1 and from 1 to 0, or $\bar{x}$. This relation helps explain its name since the complement of x is $2^n - x - 1$. It was also an attempt to be a better solution than sign and magnitude, and several early scientific computers did use the notation. This representation is similar to two's complement except that it also has two 0s: 00 … 00$_{two}$ is positive 0 and 11 … 11$_{two}$ is negative 0. The most negative number, 10 … 000$_{two}$, represents –2,147,483,647$_{ten}$, and so the positives and negatives are balanced. One's complement adders did need an extra step to subtract a number, and hence two's complement dominates today.

A final notation, which we will look at when we discuss floating point in COD Chapter 3 (Arithmetic for Computers), is to represent the most negative value by 00 … 000$_{two}$ and the most positive value$_{two}$ by 11 … 11$_{two}$, with 0 typically having the value 10 … 00$_{two}$. This representation is called a biased notation, since it biases the number such that the number plus the bias has a non-negative representation.

One's complement: A notation that represents the most negative value by 10 … 000_{two} and the most positive value by 01 … 11_{two}, leaving an equal number of negatives and positives but ending up with two zeros, one positive (00 … 00_{two}) and one negative (11 … 11_{two}). The term is also used to mean the inversion of every bit in a pattern: 0 to 1 and 1 to 0.

Biased notation: A notation that represents the most negative value by 00 … 000_{two} and the most positive value by 11 … 11_{two}, with 0 typically having the value 10 … 00_{two}, thereby biasing the number such that the number plus the bias has a non-negative representation.

PARTICIPATION ACTIVITY 2.2.11: Check yourself: Two's complement.

1) Decimal value of the following 16-bit two's complement number:
1111 1111 1111 1000$_{two}$
   - -4_{ten}
   - -8_{ten}
   - -16_{ten}
   - $65{,}528_{ten}$

Provide feedback on this section