In the first slide packet on binary representation, we saw how non-negative integer values like 97 are represented in memory.

What if, instead of having 97, we had −97?

We need a way to represent negative integers.

**Representing Negativity**

For starters, we need a way to represent whether an integer is negative or positive. We can think of this as a binary question: a number is either negative or nonnegative.

So, we can simply pick a bit in the binary representation of the integer and decide that it’s going to be the sign bit.

Which bit should we pick? Well, we want to pick the bit that we’re least likely to use in real life, so that it’s not a big waste to use it as a sign bit.

In real life, we’re much more likely to deal with very small numbers (e.g., 0, 1, 13, 97) than very large numbers (e.g., 4001431453). So, we pick the leftmost bit, called the *most significant bit*, and decide that it’ll be our sign bit.
Representations of Negative Integers: Sign-Value

Okay, now we have our sign bit. So here are three ways to represent negative integers.

Sign-Value: To get the negative version of a positive number, set the sign bit to 1, and leave all other bits unchanged.

\[ +97 = \begin{array}{cccccccccc} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ -97 = \end{array} \]

An unfortunate feature of Sign-Value representation is that there are two ways of representing the value zero: all bits set to zero, and all bits except the sign bit set to zero.

\[ 0 = \begin{array}{cccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 = \end{array} \]

\[ 0 = \begin{array}{cccccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 = \end{array} \]

This makes the math a bit confusing.

More importantly, when performing arithmetic operations, we need to treat negative operands as special cases.

Representations of Negative Integers: One’s Complement

One’s Complement: To get the negative version of a positive number, invert (complement) all bits; i.e., all 1’s become 0’s and vice versa.

\[ +97 = \begin{array}{cccccccccc} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ -97 = \sim \begin{array}{cccccccccc} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ -97 = \end{array} \Rightarrow \]

Note that the tilde symbol \( \sim \) is the bitwise inversion operator.

Just as in Sign-Value representation, in One’s Complement there are two ways of representing the value zero: all bits set to zero, or all bits set to one.

\[ 0 = \begin{array}{cccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 = \end{array} \]

\[ 0 = \begin{array}{cccccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 = \end{array} \]

This approach has the same disadvantages as sign-value.
Representations of Negative Integers

Two’s Complement

Two’s Complement: To get the negative version of a positive number, invert all bits and then add 1; if the addition causes a carry bit past the most significant bit, discard the high carry:

\[
\begin{align*}
+97 &= 011000001 \\
-97 &= \sim 011000001 + 1 \\
-97 &= 100111110 \\
-97 &= 100111111 \\
\end{align*}
\]

Note that, in Two’s Complement representation, the value zero is uniquely represented by having all bits set to zero:

\[
\begin{align*}
+0 &= 000000000 \\
-0 &= \sim 000000000 + 1 \\
-0 &= 111111111 \\
-0 &= 000000000 (\text{most significant carry discarded}) \\
\end{align*}
\]

Since negating zero produces all zeros, the representation of zero is unique (and intuitive).

A Curious Property of Two’s Complement

When you perform an arithmetic operation (e.g., addition, subtraction, multiplication, division) on two signed integers in Two’s Complement representation, you can use exactly the same method as if you had two unsigned integers (i.e., nonnegative integers with no sign bit), except that you throw away the high carry (or the borrow for subtraction).

\[
\begin{align*}
+45 &= 1011010 \\
+14 &= 0100110 \\
-14 &= \sim 0100110 + 1 \\
-14 &= 101110100001 + 1 \\
\end{align*}
\]

This property of Two’s Complement representation is so incredibly handy that virtually every general-purpose computer available today uses Two’s Complement.

Why? Because, with Two’s Complement, we don’t need special algorithms for arithmetic operations that involve negative values.
Range of Two’s Complement Values

When we represent negative integers in Two’s Complement notation, the range of numbers that can be represented in \( b \) bits is:

\[-(2^{b-1}) \ldots (2^{b-1} - 1)\]

For example, the range of numbers that can be represented in 8 bits is:

\[-(2^7) \ldots (2^7 - 1) = -128 \ldots 127\]

Likewise, the range of numbers that can be represented in 16 bits is:

\[-(2^{15}) \ldots (2^{15} - 1) = -32768 \ldots 32767\]

How do we know this?

Well, the greatest positive integer that can be represented in Two’s Complement is the number represented by having all bits except the sign bit set to one:

\[
\begin{array}{llllllll}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

What value is this? Well, it’s one less than the unsigned **positive** integer represented by

\[
\begin{array}{llllllll}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Notice that the latter value is \( 2^8 - 1 \).

Therefore, the former value is \( 2^8 - 1 - 1 \).

Range of Two’s Complement Values

(continued)

We said that when we represent negative integers in Two’s Complement notation, the range of numbers that can be represented in \( b \) bits is:

\[-(2^{b-1}) \ldots (2^{b-1} - 1)\]

We’ve now proven that the greatest positive integer that can be represented in Two’s Complement is the number that is represented by having all bits except the sign bit set to one, which is \( 2^8 - 1 \):

\[
\begin{array}{llllllllll}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

So what’s the smallest negative integer (i.e., the negative integer with the greatest absolute value)?

Well, can we represent the negative of \( 2^8 - 1 \)?

\[
\begin{array}{llllllllll}
+ (2^{8-1} - 1) = & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
- (2^8 - 1 - 1) = & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 + 1 \Rightarrow \\
- (2^8 - 1 - 1) = & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 + 1 \Rightarrow \\
- (2^8 - 1 - 1) = & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
\]
Range of Two’s Complement Values (continued)

We know that, in Two’s Complement representation in \( b \) bits, we can represent \(-2^{b-1} \):

\[-(2^{b-1} - 1) = \begin{array}{c}
1 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
1
\end{array}\]

What happens if we subtract 1 from this? Well, we get:

\[-(2^{b-1} - 1) - 1 = \begin{array}{c}
1 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0
\end{array}\]

Notice that

\[-(2^{b-1} - 1) - 1 = -2^{b-1} + 1 - 1 = -2^{b-1}\]

What happens if we subtract 1 from \(-2^{b-1}\)?

\[-2^{b-1} - 1 = \begin{array}{c}
1 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0
\end{array}\]

\[-2^{b-1} - 1 = \begin{array}{c}
0 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1
\end{array}\]

But this result, in \( b \)-bit Two’s Complement representation, is a positive value, which we can tell because its most significant bit is zero.

Therefore, \(-2^{b-1}\) is the smallest negative integer (i.e., the negative integer with the greatest absolute value) that can be represented in \( b \) bits in Two’s Complement representation.

Overflow

When we’re working with a value that’s near the limit of what can be represented in Two’s Complement for the given number of bits, we sometimes perform an operation that should result in a positive value but instead produces a negative value.

Such an event is called overflow.

For example, consider the following addition in 8-bit Two’s Complement representation:

\[
127 \equiv 0\quad1\quad1\quad1\quad1\quad1\quad1\quad1
\]

\[+\quad1 \equiv 0\quad0\quad0\quad0\quad0\quad0\quad0\quad1
\]

\[128? \equiv 1\quad0\quad0\quad0\quad0\quad0\quad0\quad0
\]

Notice that the result should be 128, but instead, in 8-bit Two’s Complement representation, it’s actually -128. (You can tell it’s negative because its leftmost bit is set to 1.) So this is overflow.
Underflow

Just as we can have overflow, we can also have underflow, which occurs when we're working with a value that's near the limit of what can be represented in Two's Complement for the given number of bits, and we perform an operation that should result in a negative value but instead produces a positive value.

\[
\begin{align*}
-128 & \equiv 10000000 \\
-1 & \equiv 00000000000000000000000000000001 \\
-129? & \equiv 00000000001111111111111111111111
\end{align*}
\]

Notice that the result should be -129, but instead, in 8-bit Two's Complement representation, it's actually 127. (You can tell it's positive because its leftmost bit is cleared to 0.) So this is underflow.

Floating Point Representation

Okay, so now that we understand how integer values are represented in memory, how are real values represented in memory?

Well, to help us understand floating point representation, let's take a look at scientific notation.

We said, a while back, that a real value can be represented in scientific notation. Here's some examples from mathematics:

\[
\begin{align*}
6,300,000,000,000,000 & = 6.3 \times 10^{18} \\
0,000,000,000,000,000,000,000,000,000,000 & = 2.71 \times 10^{-11}
\end{align*}
\]

And here's their equivalents in Fortran 90:

\[
\begin{align*}
6,300,000,000,000,000 & = 6.3E+18 \\
0,000,000,000,000,000,000,000,000,000,000 & = 2.71E-11
\end{align*}
\]

Floating point representation uses scientific notation to represent floating point numbers. For example, a 32 bit integer might be split up this way:

\[
\begin{array}{cccc}
\text{1} & \text{11111111111111111111111111111111}
\end{array}
\]

In this case, the first bit is the sign bit, which tells us whether the floating point number is negative or positive. The next set of bits represent the exponent, and the last set of bits are the fraction.

This is one of many different ways to represent floating point numbers; for example, sometimes the order is switched, or the number of bits for the exponent and for the fraction differ, and so on:

\[
\begin{array}{cccc}
\text{1} & \text{11111111111111111111111111111111}
\end{array}
\]

\textbf{Jargon}: the fraction is sometimes called the mantissa.