Binary and decimal representations

Although we typically write numbers in decimal (the base 10 number system), computers represent all values in binary, in base 2. So it is important to know how to understand numbers written in both of these representations. The way we interpret $106_{10}$ is as a sum of powers of 10. For example,

$$106_{10} = 1 \times 10^2 + 0 \times 10^1 + 6 \times 10^0$$

The position of the digit determine the power of 10 used for each term.

Similarly, we interpret a binary number as a sum of powers of 2. For example,

$$1101010_2 = 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

If we carry out this calculation in decimal, we obtain 106. This provides a simple recipe to convert between the binary and decimal representations of a number.

To find the binary representation of a number written in decimal, we need to take a different route: we repeatedly divide by 2. The remainder of each step read from bottom to top is its binary representation. Let’s convert 106 back to binary in this way:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>106</td>
<td>1101010</td>
</tr>
<tr>
<td>53</td>
<td>101001</td>
</tr>
<tr>
<td>26</td>
<td>11010</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Checkpoint 0

What is the decimal representation of $1111010_2$?

Using the right side of the table above, what is the binary representation of $49_{10}$?

Although it is important that you are able to do these conversions yourself, there are a number of good conversion tools online. For example, we like this one for binary to decimal (and more).

Hexadecimal notation

Hexadecimal represents numbers in base 16. Every hex digit corresponds to exactly 4 binary digits (bits). Thus, a 32-bit int can be written using 8 hex digits. We do so in C0 by using the prefix 0x: we enter the hex number $7F2C_{16}$ in C0 as $0x7F2C$. The hexadecimal representation allows us to “see” the bit structure of an int without having to write out 32 1’s and 0’s. In fact, C0 does not support writing numbers in binary at all — only hex and decimal are supported.

Find the hex representation of the binary number $001111010101101_2$.

Find the decimal representation of the hexadecimal number $0x20_{16}$.

Why wouldn’t it make sense to write a C0 function that converts hex numbers to decimal numbers?
Two’s complement

So far we have only considered non-negative numbers, but C0’s \texttt{int} type represents integers in the range \([-2^{31}, 2^{31})\). C0 uses two’s complement to determine which 32-bit \texttt{int’s} it considers negative. In two’s complement, the most significant bit has negative place value. The figure to the right shows how two’s complements partitions the 4-bit numbers into positive and negative. Note that the leftmost bit is always 1 for negative numbers and always 0 for positive numbers (and zero). Because of this, we call this bit the sign bit.

The same can be done for numbers that are represented using \(k\) bits for any value of \(k\), including 32 which is what C0 uses.

To convert a number in two’s complement to decimal, we apply a variant of the powers of 2’s method we saw earlier: we make the term for sign bit negative. For example, in a 4-bit world with two’s complement,

\[
1011_2 = 1 \times -2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0
\]

which evaluates to \(-5_{10}\).

What is the range of the values that can be represented using \(k\) bits using two’s complement?

---

Checkpoint 1

Assume you are in a 7-bit world (where numbers are written as 6 bits):

What is \(1101010_2\) in decimal? __________________________

What is \(0011101_2\) in decimal? __________________________

Notice that the first value you computed here consists of the same binary digits as the example at the beginning of this recitation, but now that we’re in a two’s complement world, you interpret the binary representation differently!

Checkpoint 2

In C0’s 32-bit world, what are (in binary and hex):

\texttt{int\_min}: __________________________ \[2\], __________________________ \[16\]
\texttt{int\_max}: __________________________ \[2\], __________________________ \[16\]
\texttt{-1}: __________________________ \[2\], __________________________ \[16\]

Remember that in C0, integers are always 32-bits!

Checking for overflow

Because \texttt{int’s} are just 32 bits long, numbers that are not in the range \([-2^{31}, 2^{31})\) cannot be entered in C0. Such numbers can however emerge as the result of an arithmetic operation, something that is called overflow. For example, \(4 \times 2^{30}\) overflows since its mathematical value is \(2^{32}\) which is outside the range \([-2^{31}, 2^{31})\).
Checkpoint 3

We want to write a function that adds to numbers only when no overflow can occur (in other words, when the result we get in C0 is the same as the result we get with true integer arithmetic) and fail a contract otherwise. Here’s our first attempt:

```c
int no_overflow_add(int a, int b)
/*@requires (a > 0 && b > 0 && a + b <= int_max())
|| (a < 0 && b < 0 && a + b >= int_min())
|| (a <= 0 && b >= 0)
|| (a >= 0 && b <= 0);
@*/
{
    return a + b;
}
```

This doesn’t quite work as intended. Explain what is wrong with the precondition as written. How would you correct the precondition so that it fails whenever there would be an overflow?

Bit patterns

In C0, we use 32-bit `int`’s to represent a single integer. However, it’s possible to use these 32 bits to encode other information using the 32 bits of an `int`.

It makes little sense to use arithmetic operations on such bit patterns (`int`’s we think of as representing information other than a single number). Instead, we manipulate bit patterns using a dedicated set of operations on `int`’s. These are the bitwise operations and the shifts.

What operators to use on a value of type `int` depends on what we see this value as.

- If we view it as a number, we should exclusively use arithmetic operators on it.
- If instead we view it as a bit pattern, we should manipulate it exclusively with bitwise operators and shifts.

Bitwise operators

C0 has four bitwise operations. They are called bitwise because they manipulate each bit in an `int` independently of the bits around it. Here’s how they work on a single bit:

<table>
<thead>
<tr>
<th></th>
<th>and</th>
<th>or</th>
<th>xor</th>
<th>complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

In C0, the bitwise operators apply to entire `int`’s: they apply the above tables to each of the 32 positions of their operands.
Checkpoint 4

Assume we are in the 5-bit world.

What does \( (01101_{[2]} \& 10101_{[2]} \mid (01010_{[2]} \wedge 10110_{[2]}) \) evaluate to?

Given an int \( p \), write an expression that preserves the 16 most significant bits while setting the least significant bits to all 1's. For example, for \( p = 0\times12345678 \), this expression would evaluate to \( 0\times1234FFFF \).

What is the difference between the logical operators !, && and | | on one hand and bitwise operators ~, & and | on the other?

Shifts

The two shift operators, \( x << k \) and \( x >> k \), move bits around an int \( x \). They take an int understood as a bit pattern and shift it left or right, respectively, by the specified \( k \) bits. The left shift \( x << k \) always sets the rightmost \( k \) bits of the result to 0. Instead, the right shift \( x >> k \) copies the sign bit of \( x \) to the leftmost \( k \) bits of the result — this is called sign extension. Here are some 32-bit world examples:

- \( 1101 1111 0101 0010 1101 1111 0101 0010_{[2]} << 9 = 1010 0101 1011 1110 1010 0100 0000 0000_{[2]} \)
- \( 0101 1111 0101 0010 0101 1111 0101 0010_{[2]} >> 9 = 0000 0000 0010 1111 1010 1001 0110 1111_{[2]} \)
- \( 1101 1111 0101 0010 1101 1111 0101 0010_{[2]} >> 9 = 1111 1111 1110 1111 1010 1001 0110 1111_{[2]} \)
- \( 80ABCDEF_{[16]} >> 9 = FFC055E0_{[16]} \)

Mixing bitwise and arithmetic operators

We said earlier that int’s seen as numbers should be manipulated exclusively with arithmetic operators, while int’s seen as bit patterns should be manipulated exclusively with bitwise operators and shifts.

There are very few exceptions to this rule. One is when using the left shift operation to quickly compute powers of two. It leverages this property:

\[ x << k = x \times 2^k \]

(The right shift corresponds to a variant of division that always rounds towards negative infinity.)

Another exception is when we are dealing with numbers but are interested in aspects of their binary representation, like the value of the sign bit.
Checkpoint 5

Write a function that returns 1 if the sign bit is 1, and 0 otherwise. That is, write a function that returns the sign bit shifted to be the least significant bit. Your solution can use any of the bitwise operators, but will not need all of them.

```c
int get_sign_bit(int x)
//@ensures \result == 0 || \result == 1;
{
    return .........................................................;
}
```

Fun facts about integers...

...that may come handy.

- `int_max() + 1 == int_min()`
- `int_min() - 1 == int_max()`
- `-int_min() == int_min()`
- `-x == ~x + 1`

Take-home exercise: prove that each of the above equalties hold in two’s complement.