Converting between binary and decimal

To easily convert a number represented in binary notation, such as \(10100_2\), we can employ Horner’s algorithm. At each step, we multiply the previous result by \(2\), and add the next bit in the number. To convert in the other direction, we divide by \(2\) and write the remainder at each step from bottom to top. We can see the conversion between \(10100_2\) and 20 (or \(20_{10}\) to be extra-decimaly) below.

\[
\begin{array}{cccc}
\times 2 + & & & \\
\times 2 + & & & \\
\times 2 + 1 & = & 1 & \\
1 & & & 2 \\
\times 2 + 1 & = & 5 & \\
5 & & & 10 \\
\times 2 + 0 & = & 20 & \\
\end{array}
\]

\[
\begin{array}{cccc}
\times 2 + & & & \\
\times 2 + & & & \\
\times 2 + & & & \\
\end{array}
\]

Checkpoint 0

What is the decimal representation of \(1111010\)\(_2\)? ____________

What is the binary representation of \(49\)\(_{10}\)? ____________

Hexadecimal notation

Hex is useful because every hex digit corresponds to exactly 4 binary digits (bits). Base 8 (octal) is similarly useful: each octal digit corresponds to exactly 3 binary digits. However, hex more evenly divides up a 32-bit integer. In C0 we indicate we are using base 16 with an 0x prefix, so \(7f2c\)\(_{16}\) is \(0x7f2c\).

```
   Hex       0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
  Bin.  0000  0001  0010  0011  0100  0101  0110  0111 1000 1001 1010 1011 1100 1101 1110 1111
 Dec.  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Convert the binary number 101111010101101\(_2\) to hex. _________________

Convert the 0x20 to decimal. _________________

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

Bit manipulation

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

There are also shift operators. They take a number and shift it left (or right) by the specified number of bits. In C0, right shifts sign extend. This means that if the first digit was a 1, then 1s will be copied in as we shift.

\(1101\ 1111\ 0101\ 0010\)\(_2\) \(\gg\ 8 = 1111\ 1111\ 1101\ 1111\)\(_2\)
ARGB representation of color

We usually use 32-bit integers in C0 to represent a single integer. However, it’s possible to use the bits in other ways: as 32 separate Boolean values or as 4 separate 8-bit numbers in the range $[0, 255)$. This lets us represent a color (red, green, and blue intensities, plus transparency or “alpha”), as 32-bit C0 integer.

Two’s complement

Because C0’s int type only represents integers in the range $[-2^{31}, 2^{31})$, addition and multiplication are defined in terms of modular arithmetic. As a result, adding two positive numbers may give you a negative number!

Checkpoint 1

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 getSignBit(int x)
{
    return 0;
}
```

Checkpoint 2

What assertion would you need to write to ensure that an addition would give a result without overflowing (in other words, to ensure that the result you get in C0 is the same as the result you get with true integer arithmetic).

```c
int safe_add(int a, int b)
/*
@requires a >= 0 && b >= 0 &&
*/
{
    return a + b;
}
```

What about multiplication? For the sake of simplicity, you can assume both numbers are non-negative.

```c
int safe_mult(int a, int b)
/*
@requires a >= 0 && b >= 0 &&
*/
{
    return a * b;
}
```