Agenda

Revisiting number systems
Reviewing binary representations
Changing between number systems
Binary arithmetic

On or off? Binary numbers are natural for representing two states, which makes them the language of choice for describing operations on a computer.
Number systems

A range of number systems have been used to perform arithmetic through history:

- Our base-10 decimal system: originated in 5th century India and spread to the West through the Arab world.

- Many other numeral systems follow the conventions of the base-10 decimal system, although the symbols or numerals themselves may be different.

- However, there are many examples of cultures whose number systems actually followed a different counting base or convention.
  - Babylonian numbers followed a base-60 convention.
  - Base-5, 6, 8, and 16 number systems were also present among many cultures.
Number systems and computers

In this class, we’re more concerned with number systems used to describe data in computers.

- We’ve already reviewed binary numbers as signed and unsigned representations of integers.

- Fixed-point and floating-point conventions enable binary representations of real numbers with finite precision. Consider precision a measure of how finely quantized these approximations are and how precise arithmetic operations will be with such approximations.

While binary representations are natural considering the “on” or “off” nature of digital electronic circuits, the number of binary digits needed to represent common numbers makes them inconvenient.

- Instead we can group binary digits together in sets of 3 or 4 and describe numbers in base-8 or base-16.
- We will call these octal and hexadecimal representations, respectively.
Number systems

Compare these representations of numbers in terms of their digits:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Octal</th>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
<th>Octal</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00001</td>
<td>01</td>
<td>01</td>
<td>10</td>
<td>01010</td>
<td>12</td>
<td>0A</td>
</tr>
<tr>
<td>2</td>
<td>00010</td>
<td>02</td>
<td>02</td>
<td>11</td>
<td>01011</td>
<td>13</td>
<td>0B</td>
</tr>
<tr>
<td>3</td>
<td>00011</td>
<td>03</td>
<td>03</td>
<td>12</td>
<td>01100</td>
<td>14</td>
<td>0C</td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
<td>04</td>
<td>04</td>
<td>13</td>
<td>01101</td>
<td>15</td>
<td>0D</td>
</tr>
<tr>
<td>5</td>
<td>00101</td>
<td>05</td>
<td>05</td>
<td>14</td>
<td>01110</td>
<td>16</td>
<td>0E</td>
</tr>
<tr>
<td>6</td>
<td>00110</td>
<td>06</td>
<td>06</td>
<td>15</td>
<td>01111</td>
<td>17</td>
<td>0F</td>
</tr>
<tr>
<td>7</td>
<td>00111</td>
<td>07</td>
<td>07</td>
<td>16</td>
<td>10000</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>01000</td>
<td>10</td>
<td>08</td>
<td>17</td>
<td>10001</td>
<td>21</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>01001</td>
<td>11</td>
<td>09</td>
<td>18</td>
<td>10010</td>
<td>22</td>
<td>12</td>
</tr>
</tbody>
</table>

Hexadecimal numbers use the letters “A” – “F” as the digits for decimal values 10-15.
Number systems

Decimal (ours)
- Conventional, easy to describe larger numbers using scientific notation
- Metric system uses decimal powers of 10 to convert among convenient scales for units of measurement

Binary
- Pro: relatively simple electronic implementations of arithmetic operations
- Con: larger numbers require lots of digits, only uses two symbols (low entropy)

Octal, hexadecimal
- Pro: direct conversion between binary and octal/hexadecimal systems
- Pro: more compact representations of larger numbers, using more symbols (higher entropy)
Numbers and words

While digital computers process binary data, computers collect and process groups of digits simultaneously in groups called *words*. Generally, a basic processor instruction operates on a single word of data.

- An 8-bit or 64-bit processor forms 8-bit or 64-bit groups, respectively.

- Information is processed most efficiently when processed in whole words. So bit-by-bit manipulations of a 64-bit number are much less efficient than manipulating the entire word the same way.

- Modern computers have 64-bit architectures and many modern processors include instructions to manipulate even larger groups of bits simultaneously for common operations.

\[ X = 0111101110110111110011010001001 \]

32-bit word length
Numbers and words

Different processors represent and interpret numbers using different conventions.

- Most processors subdivide words into 8-bit groups called *bytes*.
- Some of these order the bytes most-significant (right) to least-significant (left): *big-endian*.
- Others order the bytes the other way, least-significant (right) to most-significant (left): *little-endian*.
- Many computers use little-endian representations, while many network devices use big-endian representations, so network interface hardware must be able to reorder the bytes.
- Regardless, each byte is arranged from least-significant bit on the right to most-significant on the left.

- Example: 32-bit word 10001011 01101111 01011001 00110001
  = 2,339,330,353 little-endian and
  = 827,944,843 big-endian
Binary representations

Unsigned binary numbers: nonnegative numbers

- Unsigned 8-bit version of “67”: 01000111
- Unsigned big-endian 16-bit version of “56000”: 11000000 11011010 (since 56,000 = 256*218 + 192)

Signed binary numbers: positive and negative numbers

- Signed magnitude: most significant bit indicates sign; remaining bits indicate absolute value
- Signed magnitude 8-bit version of “-14”: 10001110
Other signed binary representations:
- Offset-binary: interpret N-bit number by subtracting unsigned binary representation by $2^{N-1}$
- Offset-binary 8-bit version of “-20”: 01101100
- Two’s complement: negative numbers obtained by flipping absolute value number bits and adding 1
- Two’s complement 8-bit version of “-37”: 11011011
- Two’s complement 8-bit version of “-80”: 10110000
- What’s the range of offset-binary and two’s complement 8-bit numbers? -128 to +127
Decimal $\rightarrow$ binary representation

How do we figure out the unsigned binary representation of a decimal number (e.g., $7 \rightarrow 111$)?

1. If the number is even, set the smallest binary digit to “0”, otherwise, set to “1”. Divide number by 2, rounding down.
2. Repeat, adding new binary digits to the left.
3. Stop when we divide, rounding down, to reach zero.

Example: convert “56” to binary

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Start with 56 (even)</td>
<td>0</td>
</tr>
<tr>
<td>56 / 2 -&gt; 28 (even)</td>
<td>00</td>
</tr>
<tr>
<td>28 / 2 -&gt; 14 (even)</td>
<td>000</td>
</tr>
<tr>
<td>14 / 2 -&gt; 7 (odd)</td>
<td>1000</td>
</tr>
<tr>
<td>7 / 2 -&gt; 3 (odd)</td>
<td>11000</td>
</tr>
<tr>
<td>3 / 2 -&gt; 1 (odd)</td>
<td>111000</td>
</tr>
</tbody>
</table>
Decimal $\rightarrow$ binary representation

Another way of converting from decimal to unsigned binary:

1. Find largest power of two less than or equal to the decimal number (e.g., “100” $\rightarrow$ 64 = $2^6$).
2. Set that bit to one, and subtract off that power of two from the decimal number. (Note: rightmost bit has index zero)
3. Repeat steps 1 and 2 until zero remains.

Example: convert “56” to binary

<table>
<thead>
<tr>
<th>56 -&gt; 32 = $2^5$</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>56-32 = 24 -&gt; 16 = $2^4$</td>
<td>110000</td>
</tr>
<tr>
<td>24-16 = 8 -&gt; 8 = $2^3$</td>
<td><strong>111000</strong></td>
</tr>
</tbody>
</table>

We’ll be using mainly the first method, since it adapts easily to change between other number systems as well, without having to compute exponents.
Decimal $\rightarrow$ binary representation

These conversions work well for unsigned representations of nonnegative numbers. For signed representations of negative numbers, we need to consider the signed representation used:

- **Signed magnitude**: find unsigned representation of absolute value

- **Offset binary**: for $N$-bit offset binary representation, add $2^{N-1}$ to number to make it nonnegative, then find unsigned representation

- **Two’s complement**: form unsigned representation of absolute value, flip bits, add one

**Example**: find 8-bit offset-binary representation of “-110”:
Converting from binary to decimal

Example: unsigned binary 1000101 =

Example: signed-magnitude binary 10010111 =
Binary $\rightarrow$ octal or hexadecimal

Converting between binary and octal/hexadecimal systems is easy because 8 and 16 are powers of two. Thus, we just need to collect binary digits:

Groups of three binary digits: $10011011 \rightarrow 10011011 = 233$ in octal (base-8)

Groups of four binary digits: $10011011 \rightarrow 10011011 = 9B$ in hexadecimal ("B" = 11)

Another example: $0101101101 \rightarrow$ hexadecimal:
Octal and hexadecimal notation

We can convert from octal or hexadecimal digit-by-digit:

Octal (base-8) 471 → 100 111 001 = 100111001 in binary

Hexadecimal 2B3 → 0010 1011 0011 = 1010110011 in binary

Question: what is octal 471 in decimal?
Converting from decimal $\rightarrow$ octal

Revise our original decimal $\rightarrow$ binary procedure using modulo arithmetic:

1. Divide the number by 8, storing the remainder as the least significant digit.
2. Repeat on the quotient, adding new octal digits to the left.
3. Stop when the quotient reaches zero.

Example: convert “74” to octal

\[
\begin{array}{c|c}
74 &= 8 \times 9 + 2 & 2 \\
9 &= 8 \times 1 + 1 & 12 \\
1 &= 8 \times 0 + 1 & 112 \\
\end{array}
\]
Converting from decimal → hexadecimal

Revise again using division by 16:

1. Divide the number by 16, storing the remainder as the least significant digit. Use letters A-F for remainders 10-15.
2. Repeat on the quotient, adding new hexadecimal digits to the left.
3. Stop when the quotient reaches zero.

Example: convert “74” to hexadecimal

<table>
<thead>
<tr>
<th>Calculation</th>
<th>74 = 16 * 4 + 10</th>
<th>4 = 16 * 0 + 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>A</td>
<td>4A</td>
</tr>
</tbody>
</table>
The same procedure works for other bases. Try computing a base-3 representation of “29”:

We can also go in reverse order using powers of 3

<table>
<thead>
<tr>
<th>Decimal</th>
<th>2^3 = 27</th>
<th>2^2 = 9</th>
<th>2^1 = 3</th>
<th>2^0 = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quotient</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Binary arithmetic

Binary addition can be represented via moving along a cycle:

For a 4-bit unsigned binary representation:
1. The first number is the starting point on the cycle.
2. The second number is the number of steps to take around the cycle.

Example: “3” + “7” = 0011 + 0111 = 1010 = “10”
Binary arithmetic

Negative numbers using two’s complement: addition with a negative number is equivalent to subtraction.

For the 4-bit two’s complement cycle, subtraction involves moving counter-clockwise around the cycle.

Example: “4” + “-3” = 0100 + 1101 = 0001 = “1”
Overflow and underflow

What happens when we go around the half-circle? The result won’t match the usual decimal result. This is called overflow (when we go clockwise) or underflow (when we go counter-clockwise).

For example: $0110 + 0100 = 1010$

“6” + “4” = “-6”? This is overflow!
Your turn

Let’s use 8-bit two’s complement representations of “-27” and “15”. Add them together:

“-27” =

“15” =

“-27” + “15” =
Binary multiplication

We can multiply binary numbers much like we can multiply decimal numbers:

Ex:  

```
1101  "13"
× 0101  "5"
```

```
  1101
 1101
1101
0000
```

```
+ 1101
 1000001  "65"
```

Note that computers don’t do multiplication this way; more efficient algorithms are possible…
Your turn

Try multiplying unsigned binary numbers 101 and 110 together:
Announcements

Next class: Introduction to encryption

Homework 7 due today