Chapter 2
Bits, Data Types, and Operations

How do we represent data in a computer?
- At the lowest level, a computer is an electronic machine.
  - works by controlling the flow of electrons
- Easy to recognize two conditions:
  1. presence of a voltage – we’ll call this state “1”
  2. absence of a voltage – we’ll call this state “0”
- Could base state on value of voltage, but control and detection circuits more complex.
  - compare turning on a light switch to measuring or regulating voltage

Computer is a binary digital system.
- Basic unit of information is the binary digit, or bit.
- Values with >2 states require multiple bits.
  - A collection of two bits has four possible states:
    - 00, 01, 10, 11
  - A collection of three bits has eight possible states:
    - 000, 001, 010, 011, 100, 101, 110, 111
  - A collection of n bits has \(2^n\) possible states.

What kinds of data do we need to represent?
- Numbers – signed, unsigned, integers, floating point, complex, rational, irrational, …
- Text – characters, strings, …
- Logical – true, false
- Images – pixels, colors, shapes, …
- Sound – wave forms
- Instructions
- …
- Data type:
  - representation and operations within the computer
- We’ll start with numbers…
Unsigned Integers

Non-positional notation
- could represent a number ("5") with a string of ones ("11111")
- problems?

Weighted positional notation
- like decimal numbers: "329"
- "3" is worth 300, because of its position, while "9" is only worth 9

\[
329 = 3 \times 10^2 + 2 \times 10^1 + 9 \times 10^0
\]

Unsigned Integers (cont.)

An n-bit unsigned integer represents \(2^n\) values:
from 0 to \(2^n-1\).

<table>
<thead>
<tr>
<th>2^2</th>
<th>2^1</th>
<th>2^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Unsigned Binary Arithmetic

Base-2 addition – just like base-10!
- add from right to left, propagating carry

\[
\begin{array}{c@{\hspace{1cm}}c@{\hspace{1cm}}c@{\hspace{1cm}}c@{\hspace{1cm}}c}
10010 & + & 1001 & + & 1111 \\
11011 & + & 1011 & + & 10000 \\
& + & 1011 & + & 111 \\
\end{array}
\]

Subtraction, multiplication, division: remember integer math!

Signed Integers

With n bits, we have \(2^n\) distinct values.
- assign about half to positive integers (1 through \(2^{n-1}\))
- assign about half to negative (-\(2^{n-1}\) through -1)
- that leaves two values: one for 0, and one extra

Positive integers
- just like unsigned – zero in most significant (MS) bit

\[
00101 = 5
\]

Negative integers
- sign-magnitude – set sign bit to show negative

\[
\begin{array}{c@{\hspace{1cm}}c@{\hspace{1cm}}c}
10101 & = & -3 \\
11010 & = & -5 \\
\end{array}
\]
- one's complement – flip every bit to represent negative

\[
\begin{array}{c@{\hspace{1cm}}c@{\hspace{1cm}}c}
10101 & = & -3 \\
11010 & = & -5 \\
\end{array}
\]
- in either case, MS bit indicates sign: 0=pos., 1=neg.
Two’s Complement

Problems with sign-magnitude, 1’s complement
- two representations of zero (+0 and –0)
- arithmetic circuits are complex
  - How to add two sign-magnitude numbers?
    - e.g., try 2 + (-3)
  - How to add to one’s complement numbers?
    - e.g., try 4 + (-3)

Two’s complement representation developed to make circuits easy for arithmetic.
- for each positive number (X), assign value to its negative (-X), such that X + (-X) = 0 with “normal” addition, ignoring carry out

\[
\begin{align*}
00101 & \quad (5) \\
+ & \quad 11011 \quad (-5) \\
00000 & \quad (0) \\
\end{align*}
\]

Two’s Complement Representation

If number is positive or zero,
- normal binary representation, zeroes in upper bit(s)
If number is negative,
- start with positive number
- flip every bit (i.e., take the one’s complement)
- then add one

\[
\begin{align*}
00101 & \quad (5) \\
11010 & \quad (1’s \ comp) \\
+ 1 & \quad -5 \\
11011 & \quad (-5) \\
\end{align*}
\]

Two’s Complement Shortcut

To take the two’s complement of a number:
- copy bits from right to left until (and including) first “1”
- flip remaining bits to the left

\[
\begin{align*}
011010000 & \quad (1’s \ comp) \\
+ 1 & \quad (flip) \\
100110000 & \quad (copy) \\
\end{align*}
\]
Two’s Complement Signed Integers

- MS bit is sign bit – it has weight \(-2^{n-1}\).
- Range of an n-bit number: \(-2^{n-1}\) through \(2^{n-1} - 1\).
  - The most negative number has no positive counterpart.

<table>
<thead>
<tr>
<th>(-2^n)</th>
<th>(2^n)</th>
<th>(2^{n-1})</th>
<th>(2^{n-2})</th>
<th>(2^{n-3})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(-8)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(-7)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>(-6)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>(-5)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>(-4)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>(-3)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>(-2)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>(-1)</td>
</tr>
</tbody>
</table>

Converting Binary (2’s C) to Decimal

1. If leading bit is one, take two’s complement to get a positive number.
2. Add powers of 2 that have “1” in the corresponding bit positions.
3. If original number was negative, add a minus sign.

<table>
<thead>
<tr>
<th>(n)</th>
<th>(2^n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
</tbody>
</table>

Converting Decimal to Binary (2’s C)

First Method: Division

1. Find magnitude of decimal number
2. Divide by two – remainder is least significant bit.
3. Keep dividing by two until answer is zero, writing remainders from right to left.
4. Append a zero as the MS bit; for negative, take two’s complement.

<table>
<thead>
<tr>
<th>(X)</th>
<th>(2^n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
</tbody>
</table>

More Examples

\[X = 00100111_{\text{two}}\]
\[= 2^5 + 2^2 + 2^1 + 2^0 = 32 + 4 + 2 + 1\]
\[= 39_{\text{ten}}\]

\[X = 11100110_{\text{two}}\]
\[-X = 00011010\]
\[= 2^4 + 2^3 + 2^1 = 16 + 8 + 2\]
\[= 26_{\text{ten}}\]

\[X = \text{two’s complement of } 11100110_{\text{two}}\]
\[= 00011010_{\text{two}}\]
\[= 2^4 + 2^3 + 2^1 = 16 + 8 + 2\]
\[= 26_{\text{ten}}\]

\[X = 01101000_{\text{two}}\]
\[= 2^6 + 2^5 + 2^3 = 64 + 32 + 8\]
\[= 104_{\text{ten}}\]

Assuming 8-bit 2’s complement numbers.
Converting Decimal to Binary (2’s C)

Second Method: **Subtract Powers of Two**

1. Find magnitude of decimal number.
2. Subtract largest power of two less than or equal to number.
3. Put a one in the corresponding bit position.
4. Keep subtracting until result is zero.
5. Append a zero as MS bit; if original was negative, take two’s complement.

\[
X = 104_{10} = 11110110_{2} = 16 + 32 + 8 + 4 + 2 + 0
\]

\[
X = 104_{10} = 01101000_{2}
\]

---

Operations: Arithmetic and Logical

Recall: data types include representation and operations.

2’s complement is a good representation for signed integers, now we need arithmetic operations:

- **Addition (including overflow)**
- **Subtraction**
- **Sign Extension**

Multiplication and division can be built from these basic operations.

Logical operations are also useful:

- **AND**
- **OR**
- **NOT**

---

Addition

As we’ve discussed, 2’s comp. addition is just binary addition.

- assume all integers have the same number of bits
- ignore carry out
- for now, assume that sum fits in n-bit 2’s comp. representation

\[
\begin{array}{c|c|c}
 & 01101000 & 11110110 \\ 
\hline
11110000 & -16 & -9 \\ 
01011000 & 98 & -19 \\ 
\end{array}
\]

Assuming 8-bit 2’s complement numbers.

---

Subtraction

Negate second operand, then add.

- assume all integers have the same number of bits
- ignore carry out
- for now, assume that difference fits in n-bit 2’s comp. representation

\[
\begin{array}{c|c|c}
 & 01101000 & 11110110 \\ 
\hline
-00010000 & -16 & -9 \\ 
01011000 & 98 & -19 \\ 
\end{array}
\]

Assuming 8-bit 2’s complement numbers.
Sign Extension

- To add two numbers, we must represent them with the same number of bits.
- If we just pad with zeroes on the left:
  
<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>00000100 (still 4)</td>
</tr>
<tr>
<td>1100</td>
<td>00001100 (12, not -4)</td>
</tr>
</tbody>
</table>

- Instead, replicate the MS bit -- the sign bit:

<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>00000100 (still 4)</td>
</tr>
<tr>
<td>1100</td>
<td>11111100 (still -4)</td>
</tr>
</tbody>
</table>

Overflow

- If operands are too big, then sum cannot be represented as an $n$-bit 2's comp number.

<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>00000100 (still 4)</td>
</tr>
<tr>
<td>1001</td>
<td>00001100 (12, not -4)</td>
</tr>
</tbody>
</table>

- We have overflow if:
  - signs of both operands are the same, and
  - sign of sum is different.

- Another test -- easy for hardware:
  - carry into MS bit does not equal carry out

Logical Operations

- Operations on logical TRUE or FALSE
  - two states -- takes one bit to represent: TRUE=1, FALSE=0

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
<th>A</th>
<th>B</th>
<th>A OR B</th>
<th>A</th>
<th>NOT A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- View $n$-bit number as a collection of $n$ logical values
  - operation applied to each bit independently

Examples of Logical Operations

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000101</td>
<td>11000101</td>
<td>11000101</td>
</tr>
</tbody>
</table>

- useful for clearing bits
  - AND with zero = 0
  - AND with one = no change

- useful for setting bits
  - OR with zero = no change
  - OR with one = 1

- unary operation -- one argument
  - flips every bit

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000101</td>
<td>00000101</td>
<td>01111010</td>
</tr>
</tbody>
</table>

- useful for clearing bits
  - AND with zero = 0
  - AND with one = no change

- useful for setting bits
  - OR with zero = no change
  - OR with one = 1

- unary operation -- one argument
  - flips every bit
Hexadecimal Notation

- It is often convenient to write binary (base-2) numbers in hexadecimal (base-16) instead.
  - fewer digits - four bits per hex digit
  - less error prone - no long string of 1's and 0's

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>

Converting from Binary to Hexadecimal

- Every four bits is a hex digit.
  - start grouping from right-hand side

0111 1010 1000 1111 0100 1101 0111

3 A 8 F 4 D 7

This is not a new machine representation,
just a convenient way to write the number.

Fractions: Fixed-Point

- How can we represent fractions?
  - Use a "binary point" to separate positive from negative powers of two — just like "decimal point."
  - 2's comp addition and subtraction still work (if binary points are aligned)

2^1 = 0.5
2^2 = 0.25
2^3 = 0.125

00101000.101 (40.625)
+ 11111110.110 (-1.25)
00100111.011 (39.375)

No new operations — same as integer arithmetic.

Very Large and Very Small: Floating-Point

- Large values: 6.023 x 10^{23} -- requires 79 bits
- Small values: 6.626 x 10^{-34} -- requires >110 bits
- Use equivalent of "scientific notation": F x 2^E
- Must have F (fraction), E (exponent), and sign.
- IEEE 754 Floating-Point Standard (32-bits):

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1b</td>
<td>8b</td>
</tr>
<tr>
<td>23b</td>
<td></td>
</tr>
</tbody>
</table>

N = (-1)^E x 1.fraction x 2^{Exponent-127}, 1 ≤ Exponent ≤ 254
N = (-1)^E x 0.fraction x 2^{-126}, exponent = 0
Floating Point Example

- Single-precision IEEE floating point number:
  \[
  \begin{array}{c|c|c|c}
  \text{sign} & \text{exponent} & \text{fraction} \\
  1 & 01111110 & 10000000000000000000000 \\
  \end{array}
  \]

- Sign is 1 – number is negative.
- Exponent field is 01111110 = 126 (decimal).
- Fraction is \(\frac{1}{2^{23}}\) = 1.5 (decimal).

\[
\text{Value} = -1.5 \times 2^{(126-127)} = -1.5 \times 2^{-1} = -0.75
\]

Floating-Point Operations

- Will regular 2's complement arithmetic work for Floating Point numbers?
- (Hint: In decimal, how do we compute 3.07 \times 10^{12} + 9.11 \times 10^{8}?)

Text: ASCII Characters

- ASCII: Maps 128 characters to 7-bit code.
  - printable and non-printable (ESC, DEL, …) characters

Interesting Properties of ASCII Code

- What is relationship between a decimal digit ('0', '1', ...) and its ASCII code?
- What is the difference between an upper-case letter ('A', 'B', ...) and its lower-case equivalent ('a', 'b', ...)?
- Given two ASCII characters, how do we tell which comes first in alphabetical order?
- Are 128 characters enough?
  (http://www.unicode.org/)
Other Data Types

- **Text strings**
  - array of characters, terminated with null character ('\0')
  - typically, no hardware support

- **Image**
  - array of pixels
    - monochrome: one bit (1/0 = black/white)
    - color: red, green, blue (RGB) components
    - other properties: transparency
  - hardware support:
    - typically none, in general-purpose processors
    - MMX -- multiple 8-bit operations on 32-bit word

- **Sound**
  - sequence of fixed-point numbers

LC-3 Data Types

- Some data types are supported directly by the instruction set architecture.

- For LC-3, there is only one hardware-supported data type:
  - 16-bit 2’s complement signed integer
  - Operations: ADD, AND, NOT

- Other data types are supported by interpreting 16-bit values as logical, text, fixed-point, floating-point, etc., in the software that we write.