Bits, Data Types, and Operations

Computer Programming

Programming Language
- Is telling the computer how to do something
- Wikipedia Definition: Applies specific programming languages to solve specific computational problems with solutions

High-level Programming Language
- A programming language with strong abstraction from the details of the computer
  - Isolates the execution semantics of a computer architecture compared to assembly/low-level
  - Advantage: Makes the process of developing a program simpler and more understandable

What does the Computer Understand?
At the lowest level, a computer has electronic “plumbing”
- Operates by controlling the flow of electrons through very fast tiny electronic devices called transistors

The devices react to presence or absence of voltage
- Could react actual voltages but designing electronics then becomes complex

Symbolically we represent
1. Presence of voltage as “1”
2. Absence of voltage as “0”

What does the Computer process & store?
An electronic device can represent uniquely only one of two things
- Each “0” and Each “1” is referred to as a Binary Digit or Bit
  - Fundamental unit of information storage

To represent more things we need more bits
- E.g. 2 bits can represent four unique things: 00, 01, 10, 11
- k bits can distinguish $2^k$ distinct items

Combination binary bits together can represent some information or data. E.g. 01000001 can be
1. Decimal value 65
2. Alphabet (or character) ’A’ in ASCII notation
3. Command to be performed e.g. Performing Add operation
Computer is a binary digital system

Binary (base two) system:
- Has two states: 0 and 1

Digital system:
- Finite number of symbols or bits
- If we want represent 3 or more values then we require multiple bits

All computers are characterized by number of bits the can store and process
- Example: 4, 8, 16, 32...
- Currently at 64-bit

Data

What kinds of data do we need to represent?
- Numbers — signed, unsigned, integers, real, floating point, complex, rational, irrational, ...
- Text — characters, strings, ...
- Logical — true, false
- Images — pixels, colors, shapes, ...

Data type:
- "A particular representation is data type if there are operations in the computer that can operate on the information that is encoded in the representation." (by Patt & Patel)
- Most programming languages provide basic data types

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

Unsigned Integers (cont.)

An n-bit unsigned integer represents $2^n$ values
- From 0 to $2^n - 1$

<table>
<thead>
<tr>
<th>$2^0$</th>
<th>$2^1$</th>
<th>$2^2$</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>
Base-2 Addition on Unsigned Data

Signed Integers

Positive integers
- Just like unsigned with zero in most significant bit
  - $00101 = 5$

Negative integers
- Sign-magnitude: set high-order bit to show negative, other bits are the same as unsigned
  - $10101 = -5$
- One’s complement: flip every bit to represent negative
  - $11010 = -5$
- In either case, MS bit indicates sign: 0=positive, 1=negative
- Both sign-magnitude and 1’s complement have problem

Problem

Sign-magnitude problem
- Two representations of zero (+0 and –0)

1’s complement problem
- Addition is complex
  - Add two sign-magnitude numbers?
    - e.g., try -12 + (13)
      - $10011$ (-12)
      - $+01101$ (13)
      - $100000$ (0)
  - Need to add add back the carry into least significant bit (LSB)
  - Which will result in the correct result, -12+13 = 1

Two’s Complement

Idea
- Find representation to make arithmetic simple and consistent

Specifics
- For each positive number (X), assign value to its negative (-X), such that $X + (-X) = 0$ with “normal” addition, ignoring carry out

<table>
<thead>
<tr>
<th>Positive</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00101$</td>
<td>$01001$</td>
</tr>
<tr>
<td>$+11011$</td>
<td>$10111$</td>
</tr>
</tbody>
</table>

CIT 593
Two's Complement (cont.)

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

$00101$ (5) $11010$ (1's comp) $01001$ (9) $10110$ (1's comp)

\[ + \ 1 \]

\[ 11011 \] (5) $10111$ (9)

Two's Complement Signed Integers

Range of an n-bit number: $-2^{n-1}$ through $2^{n-1} - 1$
- Note: most negative number ($-2^{n-1}$) has no positive counterpart

<table>
<thead>
<tr>
<th>$2^3$</th>
<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>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

CIT 593 13

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

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

\[ X = 11100110 \text{two} \]
\[ -X = 00011010 \]
\[ = 2^5 + 2^3 + 2^1 = 16 + 8 + 2 \]
\[ = 26 \text{ten} \]
\[ X = -26 \text{ten} \]

Assuming 8-bit 2's complement numbers.

CIT 593 15

More Examples

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

\[ X = 11100110 \text{two} \]
\[ -X = 00011010 \]
\[ = 2^5 + 2^3 + 2^1 = 16 + 8 + 2 \]
\[ = 26 \text{ten} \]
\[ X = -26 \text{ten} \]

Assuming 8-bit 2's complement numbers.

CIT 593 16
Converting Decimal to Binary (2’s C)

First Method: Division
1. Change to positive decimal number
2. Divide by two – remainder is least significant bit
3. Keep dividing by two until answer is zero, recording remainders from right to left
4. Append a zero as the MS bit; if original number negative, take two’s complement

<table>
<thead>
<tr>
<th>Division Steps</th>
<th>Remainder</th>
<th>Binary Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 104₁₀</td>
<td></td>
<td></td>
</tr>
<tr>
<td>52/2 = 26</td>
<td>r0</td>
<td>bit 0</td>
</tr>
<tr>
<td>26/2 = 13</td>
<td>r2</td>
<td>bit 2</td>
</tr>
<tr>
<td>13/2 = 6</td>
<td>r3</td>
<td>bit 3</td>
</tr>
<tr>
<td>6/2 = 3</td>
<td>r5</td>
<td>bit 5</td>
</tr>
<tr>
<td>3/2 = 1</td>
<td>r1</td>
<td>bit 4</td>
</tr>
<tr>
<td>1/2 = 0</td>
<td>r1</td>
<td>bit 6</td>
</tr>
</tbody>
</table>

X = 01101000₂

Second Method: Subtract Powers of Two
1. Change to positive 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

<table>
<thead>
<tr>
<th>n</th>
<th>2ⁿ</th>
<th>X</th>
<th>Subtract</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>104</td>
<td>104 - 64</td>
<td>40</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>40</td>
<td>40 - 32</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8 - 8</td>
<td>0</td>
</tr>
</tbody>
</table>

X = 01101000₂

Operations: Arithmetic and Logical

Recall: A data type includes representation and operations

Operations for signed integers:
- Addition
- Subtraction
- Sign Extension

Logical operations are also useful:
- AND
- OR
- NOT

And . . .
- Overflow conditions for addition

Addition

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
  - Assuming 8-bit 2’s complement numbers

01101000₁₀ + 1111011₁₀ = 11110101₁₀ (19)

01110000₁₀ - 1111111₁₁₁₁ = 1110110₁₀ (19)

01010100₀₁₀ + 1111011₁₁₁₁ = 1110110₁₀ (19)

carry(discard)
**Subtraction**

Negate 2nd operand and 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

| 01101000 (104) | 11110110 (−10) |
| 00010000 (16)  | −11110111 (−9) |

\[-00010000 (−16) − 11110111 (−9)\]

| 01101000 (104) | 11110110 (−10) |
| +11110000 (−16) | + 00001001 (9) |
| 01011000 (88)  | 11111111 (−1) |

Assuming 8-bit 2’s complement numbers.

---

**Sign Extension**

To get correct results
- Must represent numbers with same number of bits

What 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 (4)</td>
<td>00000100 (still 4)</td>
</tr>
<tr>
<td>1100 (−4)</td>
<td>00001100 (12, not −4)</td>
</tr>
</tbody>
</table>

Now, let’s replicate the MSB (the sign bit)

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

---

**Logical Operations**

Operations on logical TRUE or FALSE
- Two states: TRUE=1, FALSE=0

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>(A \text{ AND } B)</th>
<th>(A \text{ OR } B)</th>
<th>(A \text{ NOT } A)</th>
</tr>
</thead>
<tbody>
<tr>
<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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

View \(n\)-bit number as a collection of \(n\) logical values
- Operation applied to each bit independently (bitwise operators)
  - Lot of use in water marking, cryptography, and new application that requires bit manipulation

---

**Examples of Logical Operations**

### AND
- Useful for clearing bits
  - \(A \text{ AND } 0\)
  - \(A \text{ AND } 1\)

<table>
<thead>
<tr>
<th>(A \text{ AND } 0)</th>
<th>(A \text{ AND } 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000101</td>
<td>11000101</td>
</tr>
</tbody>
</table>

### OR
- Useful for setting bits
  - \(A \text{ OR } 0\)
  - \(A \text{ OR } 1\)

<table>
<thead>
<tr>
<th>(A \text{ OR } 0)</th>
<th>(A \text{ OR } 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000101</td>
<td>00001111</td>
</tr>
</tbody>
</table>

### NOT
- Unary operation -- one argument
  - Flips every bit

<table>
<thead>
<tr>
<th>(A \text{ NOT } A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00111010</td>
</tr>
</tbody>
</table>
Overflow

Overflow is said to occur if result is too large to fit in the number of bits used in the representation.

- Sum cannot be represented as $n$-bit 2's comp number

$$
\begin{array}{c}
01000 \ (8) \\
+01001 \ (9) \\
\hline
10111 \ (-9)
\end{array}

\begin{array}{c}
10001 \ (-15) \\
-01111 \ (+15)
\end{array}
$$

We have overflow if

- Signs of both numbers are the same, and Sign of sum is different
- If Positive number is subtracted from a Negative number, result is positive and vice versa

Any application involving numeric calculations should take of overflow situation otherwise consequences can be dire

- E.g. Control software malfunction of Ariane 5 Flight 501 spacecraft

Hexadecimal Notation

It is often convenient to write binary (base-2) numbers as hexadecimal (base-16) numbers instead

- Fewer digits: four bits per hex digit
- Less error prone: easy to corrupt 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>

Number System

People like to use decimal numbers

Computers use binary numbers

- E.g. Programming Languages
  - Translates decimal numbers into binary
  - The computer does all its arithmetic in binary
  - The languages translates binary results back into decimal

You occasionally have to use numbers in other number systems

- In Java, you can write numbers as octal, decimal, or hexadecimal but not binary
- In HTML, colors are usually specified in hexadecimal notation:
  - #FF0000, #669966,
- In C, you can write all notations

Hexadecimal Conversions

Binary to Hex

- Every group of four bits is a hex digit
- Start grouping from right-hand side

$$
\begin{array}{cccccccc}
0011 & 1010 & 1000 & 1111 & 0100 & 1101 & 0111 \\
3 & A & 8 & F & 4 & D & 7
\end{array}
$$

Hex to Decimal

$$1AC_{16} = 1 \times 16^2 + 10 \times 16^1 + 12 \times 16^0 = 428_{10}$$

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

- 3 digits per every octal group

<table>
<thead>
<tr>
<th>Binary</th>
<th>Octal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Octal to Binary Conversion

- One octal digit equals three binary digits

\[ 10110101110010100001011 \]

5 5 3 4 5 0 1 3

Octal to Decimal

\[ 173_8 = 1 \times 8^2 + 7 \times 8^1 + 3 \times 8^0 \]
\[ = 123_{10} \]

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

Problem
- Large values: 6.023 \times 10^{23} -- requires 79 bits
- Small values: 6.626 \times 10^{-34} -- requires >110 bits

Use equivalent of “scientific notation”: F \times 2^E

Need to represent F (fraction), E (exponent), and sign

IEEE 754 Floating-Point Standard (32-bits):

\[
\begin{array}{c}
S \quad E \quad 2^{23b}
\end{array}
\]

\[
N = (-1)^S \times 1.fraction \times 2^{Exponent - 127}, 1 \leq \text{exponent} \leq 254
\]

\[
N = (-1^{5}) \times 0.fraction \times 2^{-126}, \text{exponent} = 0
\]
**Floating Point Example**

Single-precision IEEE floating point number

\[
1 \ 01111110 \ 10000000000000000000000
\]

- **Sign**: 1: number is negative
- **Exponent**: field is 01111110 = 126 (decimal)
- **Fraction**: is 100000000000… = \(2^{-1} = 1/2 = 0.5\) (decimal)

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

**Floating Point (contd..)**

\[N = -1^{j} \times 1.\text{fraction} \times 2^{\text{exponent} - 127}, \ 1 \leq \text{exponent} \leq 254 \rightarrow \text{Normalized}\]

\[N = -1^{j} \times 0.\text{fraction} \times 2^{-127}, \ \text{exponent} = 0 \rightarrow \text{Denormalized}\]

- **Zero**: Exponent field & fraction field is all 0’s
- **Infinity (positive & negative)**: Exponent all 1’s and Fraction all 0’s
- **NaN (Not a Number)**: Exponent all 1’s and Fraction all 0’s
  - When does this occur?
  - An invalid operation is also not the same as an
    - Arithmetic overflow (which might return an infinity)
    - An arithmetic underflow (which would return the smallest normal number, a denormal number, or zero)

**Fun Experiment**

Try this experiment: Open up a new file in Notepad and type – “Four score and seven years ago”

You find the file to be 30 bytes in size

- Notepad stores the sentence in a file on disk
- File is nothing but sequence of characters
- The file will also contain 1 byte per character (note that space is also a character)
Storage Space for Numerics

Numeric types in Java are characterized by their **size**: how much memory they occupy.
- 1 byte = 8 bits

### Integral types

<table>
<thead>
<tr>
<th>type</th>
<th>size</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1 byte</td>
<td>-128: 127</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>-32768: 32767</td>
</tr>
<tr>
<td>char</td>
<td>2 bytes</td>
<td>0: 65535</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>-2147483648: 2147483647</td>
</tr>
<tr>
<td>long</td>
<td>8 bytes</td>
<td>...</td>
</tr>
</tbody>
</table>

### Floating point types

<table>
<thead>
<tr>
<th>type</th>
<th>size</th>
<th>largest</th>
<th>smallest &gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>3.4E38</td>
<td>1.4E-45</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>1.7E308</td>
<td>4.9E-324</td>
</tr>
</tbody>
</table>

What about C Language?
- Are platform dependent

Other Data Types

#### Text strings
- Sequence of characters, terminated with NULL (0)
- Typically, no hardware support

#### Image
- Array of pixels
  - Monochrome: one bit (0/1 = black/white)
  - Color: red, green, blue (RGB) components (e.g., 8 bits each)
  - Typically no hardware support

#### Sound
- Sequence of fixed-point numbers