Binary Representation

CSE 410
Lecture 03
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
First: Why Binary?

- Electronic implementation
  - Easy to store
  - Reliably transmitted on noisy and inaccurate wires

- Other bases possible:
  - Distinguish more voltage levels
  - DNA data storage (base 4: A, C, G, T)
Bit

- A bit is a single binary value
- “Binary” means there are (only) two distinct values
  - in computers, high and low voltage
- We can map the two values to any other pair of values
  - Orange vs Apple; Up vs Down; 8 vs 10; 0 vs 1; true vs false
- Of these, the last two have many attractive properties
  - 0 and 1 → base-2 (binary) integers
  - true and false → Boolean circuits
## Bit Operations

- **Unary operation**
  - not
    - $\sim 1 = 0$
    - $\sim 0 = 1$

- **Binary operations**
  - and
    - $0 \& 0 = 0$
    - $0 \& 1 = 0$
    - $1 \& 0 = 0$
    - $1 \& 1 = 1$

*Operators are written as in C (and many other languages)*

*Note that operator & is different from operator &&*
Bit Operations

- Binary Operations
  - or
    - 0 | 0 == 0
    - 0 | 1 == 1
    - 1 | 0 == 1
    - 1 | 1 == 1
  - xor (“exclusive or”)
    - 0 ^ 0 == 0
    - 0 ^ 1 == 1
    - 1 ^ 0 == 1
    - 1 ^ 1 == 0
Bit Strings

- A concatenation of bits
  - Example 0 1 0 1 0 1 1 1

- The bit operators can be applied to bit strings
  - 0 1 0 1 0 1 1 1
  - & 1 1 0 0 0 1 1 0
    - ---------------
    - 0 1 0 0 0 1 1 0
  - Similarly for |, ^, and ~
Bit String Shift

- **Left shift: <<**
  - $01010101 << 1 == 10101010$
  - $01010101 << 3 == 10101000$

- **Right shift arithmetic: >>**
  - $01010101 >> 1 == 00101010$
  - $01010101 >> 3 == 00001010$

- **Note: right shift has two form, arithmetic and logical**
  - Arithmetic propagates the high order bit
    - $01010101 >> 3 == 00001010$
    - $10101010 >> 3 == 11110101$
  - Logical shifts in zeros from the left
Bit Masks: and

“and masks” turn off bits wherever the mask has a 0

- Example mask: 0 0 0 0 0 0 1
  - And’ed with another 8 bit string, it copies the low order bit of the other string and sets everything else to zero
    
    \[
    \begin{array}{cccccccc}
    1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
    \& & 0 & 0 & 0 & 0 & 0 & 1 \\
    \hline
    & & 0 & 0 & 0 & 0 & 0 & 1
    \end{array}
    \]
  - Other masks:
    - 0 0 0 0 0 1 1 => copy two low order bits
    - 0 0 0 0 1 1 0 0 => copy bits 2 and 3
    - etc.
Bit masks: or

- or masks turn on bits (wherever the mask has a 1)
  - Example mask: 0 0 0 0 1 0 0 1
    
    1 0 1 0 1 0 1 0
    | 0 0 0 0 1 0 0 1
    |------------------
    1 0 1 0 1 0 1 1
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
Integers and Integer Representations

- What is 7061?
- It’s not an integer, it’s a “place value” representation of an integer
  - We could equally write $7 \times 10^3 + 0 \times 10^2 + 6 \times 10^1 + 1 \times 10^0$ but that’s a lot less convenient
  - We could write $353 \times 20 + 1$, but that’s even dumber (why?)
- What about 70000000000000000000061?
  - It might be handier to write $7 \times 10^{22} + 61$
- There is no “right representation” there are just ones that are more convenient than others
Place value representation

- We write \( n \) consecutive digits, numbering them 0 to \( n-1 \) starting from the right. Place \( j \) has value \( b^j \) for some base \( b \).

- We write in each place a *digit*. There are \( b \) digits, representing the numbers 0, 1, 2, ..., \( b-1 \).

\[
\begin{array}{cccc}
  d_3 & d_2 & d_1 & d_0 \\
  b^3 & b^2 & b^1 & b^0
\end{array}
\]

- The place value string represents the integer
  \( d_{n-1}b^{n-1} + d_{n-2}b^{n-2} + \ldots + d_0b^0 \)
Example: 1024

- **b=10 (decimal)**
  - Digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - 1024 means $1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$

- **b=2 (binary)**
  - Digits are 0, 1
  - 10000000000 means $1 \times 2^{10}$ (plus a lot of “zero times x” terms)
    - Which is 1024 in base 10
Simplifying representations

- Which is bigger, \(231237943432586732275839\) or \(23123794343584332235839\)? \((Both\ are\ base\ 10)\)
- We (humans) prefer representations with fewer digits
- We can reduce the number of digits a factor of \(k\) by raising the base by a power of \(k\).
  - E.g., instead of base 10, use base 1000
    - Of course, we now need a 1000 different symbols for digits
Simplifying binary

- Start with binary:
  00000001001000110100010101100111

- Octal: Raise the base by a power of 3 (so, base 8)
  00 000 001 001 000 110 100 010 101 100 111
  0   0   1   1   0   6   4   2   5   4   7

- Hexadecimal: Raise the base by a power of 4 (base 16)
  0000 0001 0010 0011 0100 0101 0110 0111
  0   1   2   3   4   5   6   7
Hexadecimal

- Grouping by four bits is handy
  - Memories are always a multiple of 8 bits in length
    - We’ll see later why

- Hex digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - Correspond to values in base 10 of 0, 1, ..., 9, 10, 11, 12, 13, 14, 15
  - Case insensitive
# Hex ↔ Binary

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

What is FFFF in binary?

Is 237E even or odd?

We should specify what base we’re using when writing integers. In C:
- 123 is a decimal constant
- 0123 is an octal constant
- 0X0123 is a hex constant
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
Addition with Place Value Representations

- Addition is easy with the standard algorithm (carry ripple)

\[
\begin{array}{ccc}
1 & 1 \\
0 & 0 & 1 & 0 & 2 \\
0 & 1 & 1 & 1 & 7 \\
1 & 0 & 0 & 1 & 9 \\
\end{array}
\]

- One problem: what about addition of negative numbers?
  \[
  \begin{array}{c}
  24 \\
  +(-7) \\
  \end{array}
  \]

- Another problem: Hey, what about negative numbers at all?

- Last problem: Overflow
Overflow

- A fixed amount of space is allocated for each value on a computer
  - For integers, usually 1, 2, 4, or 8 bytes (8, 16, 32, or 64 bits)
- **Q:** What if the result is too big to fit in that much space?
- **A:** Too bad. The highest order digit is thrown away.
- That’s called **overflow**

```
 1 1 1
1 1 0 1 5
1 0 1 1 11
1 0 0 0 0
```

```plaintext
1 1 1
0 1 0 1 5
1 0 1 1 11
1 0 0 0 0
```
Two’s Complement Binary Integers

- Two’s complement is a representation for positive and negative integers
  - Addition is always addition, even if one or both values are negative
  - About half the bit strings are negative and half are positive

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
</tbody>
</table>

Verify that $x + -x = 0$
Properties

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

- If you count up from 0 by 1, you *wrap* from the largest positive integer to the smallest negative integer.
- If the high order bit is 0, the number is non-negative. If it’s 1, the number is negative.
- You get the same bit string result adding bit strings as signed values as you do adding them as unsigned.
Overflow

- Overflow occurs when the result doesn’t fit in the limited number of bits you have
  - 0001 + 0111 => 1001
    1 + 7 = -8
- You can overflow when subtracting or multiplying as well
- Unsigned integers also overflow
  - 0001 + 0111 = 1000
    1 7 8 [no overflow]
  - 0001 + 1111 = 0000
    1 15 0
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
Floating Point Overview

- We have only 32 bits, so we have only $2^{32}$ different values we can represent.

- We’re going to do the binary version of scientific notation: $2.357 \times 10^{14}$.

- Different representations have different:
  - range – roughly, how big the exponent can be
  - precision – basically the number of significant digits in the fraction.
Floating Point: Limited Digits

- Let’s start in decimal to see the issues
- Suppose you have enough space in a word to hold 8 decimal digits
  - How many digits should you use to represent the exponent and how many the fraction
- At the extremes
  - $1 \times 10^{2345678}$
  - $1.234567 \times 10^8$
- So
  - Want some balance in range vs precision
  - Also need a way to represent negative values
  - Also need a way to represent negative exponents
32-bit Binary Floats

- Called “single precision” floats
- Value is \([+/-] \text{[fraction]} \times 2^{\text{exponent}}\)
- The 32 bits are used as:
  - High order bit is the sign of the value: 1 for negative, 0 for non-negative
  - The next 8 bits are the signed (two’s complement) value for the exponent: 127 to -128
  - The remaining 23 bits are the fraction
- Range: approximately \(2.0 \times 10^{38}\) to \(2.0 \times 10^{-38}\)
- Numbers can overflow: exponent gets too big
- Numbers can underflow: exponent gets too small
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
Character Representation

- We simply agree on a mapping from bit strings to characters
  - “Everyone” knows what the mapping is
  - The compiler inserts the agreed bit string when you write ‘A’
  - The output system writes A when it sees that bit string

- There is more than one agreed representation

- ASCII
  - Historically the agreed mapping
  - Fixed, 8-bit long strings

- Unicode
  - Variable length encoding: 8, 16, or 32 bits per character
  - Many, many more bit strings, so many, many more characters
<table>
<thead>
<tr>
<th>ASCII Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;NUL&gt;</td>
</tr>
<tr>
<td>1</td>
<td>&lt;SOH&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;STX&gt;</td>
</tr>
<tr>
<td>3</td>
<td>&lt;ETX&gt;</td>
</tr>
<tr>
<td>4</td>
<td>&lt;EOT&gt;</td>
</tr>
<tr>
<td>5</td>
<td>&lt;ENQ&gt;</td>
</tr>
<tr>
<td>6</td>
<td>&lt;ACK&gt;</td>
</tr>
<tr>
<td>7</td>
<td>&lt;BEL&gt;</td>
</tr>
<tr>
<td>8</td>
<td>&lt;BS&gt;</td>
</tr>
<tr>
<td>9</td>
<td>&lt;TAB&gt;</td>
</tr>
<tr>
<td>10</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>11</td>
<td>&lt;VT&gt;</td>
</tr>
<tr>
<td>12</td>
<td>&lt;FF&gt;</td>
</tr>
<tr>
<td>13</td>
<td>&lt;CR&gt;</td>
</tr>
<tr>
<td>14</td>
<td>&lt;SO&gt;</td>
</tr>
<tr>
<td>15</td>
<td>&lt;SI&gt;</td>
</tr>
<tr>
<td>16</td>
<td>&lt;DLE&gt;</td>
</tr>
<tr>
<td>17</td>
<td>&lt;DC1&gt;</td>
</tr>
<tr>
<td>18</td>
<td>&lt;DC2&gt;</td>
</tr>
<tr>
<td>19</td>
<td>&lt;DC3&gt;</td>
</tr>
<tr>
<td>20</td>
<td>&lt;DC4&gt;</td>
</tr>
<tr>
<td>21</td>
<td>&lt;NAK&gt;</td>
</tr>
<tr>
<td>22</td>
<td>&lt;SYN&gt;</td>
</tr>
<tr>
<td>23</td>
<td>&lt;ETB&gt;</td>
</tr>
<tr>
<td>24</td>
<td>&lt;CAN&gt;</td>
</tr>
<tr>
<td>25</td>
<td>&lt;EM&gt;</td>
</tr>
<tr>
<td>26</td>
<td>&lt;SUB&gt;</td>
</tr>
<tr>
<td>27</td>
<td>&lt;ESC&gt;</td>
</tr>
<tr>
<td>28</td>
<td>&lt;FS&gt;</td>
</tr>
<tr>
<td>29</td>
<td>&lt;GS&gt;</td>
</tr>
<tr>
<td>30</td>
<td>&lt;RS&gt;</td>
</tr>
<tr>
<td>31</td>
<td>&lt;US&gt;</td>
</tr>
</tbody>
</table>

This table provides the ASCII codes and corresponding characters.
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
C Language Pointers

- `int x;`  // x names a 32-bit string we’ll use as an int

- `int *p;`  // p names a bit string that can hold a
  // memory address

- `p = &x;`  // set p to the address of x

- `*p = 4;`  // sets the word of memory pointed at by
  // to 4 (i.e., x = 4)
C Language Pointers

```c
int x;
int *p;
p = &x;
*p = 4;

.addi x2, x0, x      # x2 = &x
.sw x2, p           # p = &x
.addi x3, x0, 4     # 4
.sw x3, 0(x2)       # *p = 4

.text

...  
x: .word 0
...  
p: .word 0
```
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
Arrays

- Arrays are just consecutive words of memory
  - The CPU doesn’t know anything about “arrays”
- The array name is the “base address” of the array
- The index is the offset from that base address


```
+3
A[3]
```

```
+10
A[10]
```
Arrays

- `int A[10];`

```
.text
addi   x1, x0, 4       # 4
addi   x2, x0, A       # base address of A
sw     x1, 3(x2)       # store at A[3]
```
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
Structure Representation

- struct person {
  int id;
  int department;
}

struct person *p

- ... p->department = 10

This defines a type. It doesn’t allocate memory.
“id” and “department” are offsets from the base of a struct person.
They have values 0 and 1 respectively.

It’s a similar idea for objects. They’re hunks of consecutive memory. Field names are offsets into those hunks.
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
- Strings
String Representation

- RISC-V hardware doesn’t know anything about “string”
- Strings are inventions of the programming language
- In C, strings are zero terminated arrays of characters

```csharp
char str[17];
str[0] = 'T';
str[1] = 'h';
str[2] = 'i';
str[3] = 's';
str[4] = ' ';
...

The C standard library provides functions that make this easier:

```c
strcpy(str, "This is a string");
```