Let’s start at the very beginning... when you store data you begin with ???...

- Our first requirement is to find a way to represent information (data) in a form that is mutually comprehensible by human and machine.
  - What kinds of data?
    - Integers
    - Reals
    - Text
    - ...what else
  - We will start by examining different ways of representing integers, and look for a form that suits the computer.

Recall: Datatypes

- A type is a classification of data that tells the compiler or interpreter how the programmer intends to use it.
  - the process/algorithm and result of adding two variables differs greatly according to whether they are integers, floating point numbers, or strings.

- Programming languages have a set of data types defined in the language
  - In C: `float, int, long int, unsigned int`

Number systems

- A number is a mathematical concept
  - Natural numbers, Integers, Reals, Rationals,...
- Many ways to represent a number....
  - Symbols used to create a representation
What was your first “counting” experience…
How did you learn to count? How did you express a number?

- Your first counting numbers experience
  The Unary system also used by Turing Machines

...Why?

In the CS world….

- There are 10 kinds of people in the world…
  - Those who know binary, and those who don’t

Big Point: Computer is a Binary Digital System

“Digital” (discrete) vs Binary (base two):

- Finite number of symbols
- Opposite is “analog” (continuous)
- Discrete Math vs Continuous Math

- Basic unit of information: the binary digit, or bit
- 3+ states require multiple bits
  - 2 bits → 4 possible states/strings: 00, 01, 10, 11
  - 3 bits → 8 possible states: 000, 001, 010, 011, 100, 101, 110, 111
  - In general: \( n \text{ bits} \rightarrow 2^n \text{ possible states} \)

- Why binary?

Why Binary: Because …

- Computers are electrical at base…
  - Electrical: Operates by controlling the flow of electrons
  - … and components in electrical circuits have two easily recognizable states
    - Absence of voltage and presence of voltage
    - Define one of these as 1, other as 0

<table>
<thead>
<tr>
<th>Digital Values</th>
<th>0</th>
<th>“0”</th>
<th>Illegal</th>
<th>“1”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analog Values</td>
<td>0</td>
<td>0.5</td>
<td>2.4</td>
<td>2.9 Volts</td>
</tr>
</tbody>
</table>

- Alternative: define multiple discrete values in voltage range
  - Problem: circuits would become much more complex
  - (A little) more about this in Chapter 3
“Binary Digital System”? 

- Everything is sequences of 0s and 1s
  - Text – characters, strings, …
  - Numbers – integers, reals, …
  - Images – pixels, colors, shapes, …
  - Sound
  - Instructions
  - …

- The fact that instructions are also “just 0s and 1s” makes a computer universal: programs are just another kind of data!

Bits – the universal data representation

- It is important to realize that everything that is stored or manipulated on the computer is ultimately expressed as a group of numbers and, hence, as a sequence of bits.
  - Text – individual samples represented as binary numbers/codes
  - Audio – Sounds represented as a sequence of audio samples
  - Pictures – Represented as arrays of intensity values, intensity values are stored as numbers
    - Monochrome images – 8 bits per pixel
    - Color images – 3 channels Red, Green and Blue 8 bits per channel

Hmmm……Machine Data Types

- devices that make up a computer are switches that can be on or off, i.e. at high or low voltage.
  - Thus they naturally provide us with two symbols to work with: we can call them on & off, or (more usefully) 0 and 1.

- We don’t want to keep referring to switches…
  - power of abstraction and problem transformation!

Integer Representation

- What are you used to?
  - Decimal *Weighted positional representation*
    - Like decimal numbers you are used to: “329”
    - “3” is worth 300, because of its position (most significant)
    - “9” is only worth 9 (least significant)
  - Decimal: we use 10 symbols (0,1,…9)

<table>
<thead>
<tr>
<th>base-10 (decimal)</th>
<th>329</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^2</td>
<td>3</td>
</tr>
<tr>
<td>10^1</td>
<td>2</td>
</tr>
<tr>
<td>10^0</td>
<td>9</td>
</tr>
</tbody>
</table>

\[3 \times 100 + 2 \times 10 + 9 \times 1 = 329\]
### Radix-k Representation
- In general: we have radix-k notation
- Use k symbols – also known as k-ary numbers (radix k)
  - 0,1,2,...,k-1
  - Radix-10 (decimal) 0,1,2,...,9
  - Radix 2 (binary) 0,1
  - Radix 16 (hex) 0,1,...,9,A,B,C,D,E,F
- Weighted positional numbers – position gives “weight” of location
  - Position 0 (rightmost) has weight 1 (k^0)
  - Position 1 has weight k (k^1)
- How many different radix k numbers of length n
  - Recall Discrete Math:
    - Each of the n positions can have k values
    - How many different strings of length n, where each position has one of k values
  Therefore, we have ??

### Integer Representation
- What are you used to?
- Weighted positional representation
  - Like decimal numbers you are used to: “329”
  - “3” is worth 300, because of its position (most significant)
  - “9” is only worth 9 (least significant)

\[
\begin{array}{c|c|c}
\text{base-10 (decimal)} & \text{329} \\
10^2 & 10^1 & 10^0 \\
3 \times 100 + 2 \times 10 + 9 \times 1 = 329 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\text{base-2 (binary)} & \text{101} \\
2^2 & 2^1 & 2^0 \\
1 \times 4 + 0 \times 2 + 1 \times 1 = 5 \\
\end{array}
\]

### Radix k Representation: Definition
- radix k number with n positions
  - \( a_n a_{n-1} ... a_2 a_1 a_0 \)
    - each \( a_i \) is a value (0,1,...,k-1)
- This radix k number represents the decimal value/number
  - \( \sum_{i=0}^{n-1} a_i k^i \)
- Radix k number of length n has \( k^n \) different numbers
  - Highest value is \( (k^n - 1) \)
- Binary representation is nothing but radix 2
  - Symbols are 0, 1
  - Positions are powers of 2
  - Length n, has \( 2^n \) different numbers
  - Largest integer has value \( 2^n - 1 \)

###Unsigned Integers
- 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>\text{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>
**Question**
- What number does the binary string 1011 represent?
- What number does 10011 represent?

**Binary Representation: Summary**
- Weighted positional notation using base (radix) 2
- What are the symbols in base 2?
  - 0, 1
- k bit number: \( b_{k-1}, b_{k-2}, \ldots, b_1, b_0 \)
  - Each \( b_i \) is?
- Decimal integer \( N \) represented by this binary number is:
  \[
  b_{k-1} \cdot 2^{k-1} + b_{k-2} \cdot 2^{k-2} + \ldots + b_1 \cdot 2^1 + b_0 \cdot 2^0
  \]

**Conversion from Decimal to Binary**
```java
//input is Decimal number N, output is list of bits b, // i=0;
while N > 0 do
  b_i = N % 2; // b_i = remainder; N mod 2
  N = N / 2; // N becomes quotient of division
  i++;
end while
```
- Replace 2 by \( r \) and you have an algorithm that computes the base \( r \) representation for \( N \)

**Question:**
1. What is the binary representation of decimal number 19
   - Express 19 as a sum of numbers each a power of 2
   - Algorithm to convert decimal (base 10) to binary (base 2)
     - Generalize to convert from base \( k \) to base \( m \)
Example: Conversion of 19 to Binary

//input is Decimal number N, output is list of bits b_i
i=0;
while N > 0 do
    b_i = N % 2; // b_i = remainder; N mod 2
    N = N / 2; // N becomes quotient of division
    i++;
end while
• Iteration 1: b_0 = 19%2 =1 and N= 19/2= 9
• Iteration 2: b_1 = 9%2 = 1 and N=4
• Iteration 3: b_2 = 4%2 = 0 and N=2
• Iteration 4: b_3 = 2%2 =0 and N=1
• Iteration 5: b_4 = 1%2 =1 and N=0 so loop terminates
• Binary representation of 19 = 10011

Hexadecimal (Base-16) Notation

• More compact and convenient than binary (base-2)
  • Fewer digits: four bits per hex digit → less error prone
  • Just a notation, not a different machine representation
  • Most languages (including C and LC-3) parse hex constants
  • Sometimes hex numbers preceded with x or 0x

<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<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>

Hexadecimal

• Any binary number can be rewritten as hexadecimal and vice versa just by translating the nibbles
  • 01101101 = 0110 1101 = 6D
  • 0011011110101110 = 0011 0111 1010 1110 = 37AE
  • 7F = 0111 1111

• Just break # up into groups of four
• Then translate each 4-bit # to decimal
  • Identical technique to go from decimal to bin.

<table>
<thead>
<tr>
<th>2^3</th>
<th>2^2</th>
<th>2^1</th>
<th>2^0</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Fun with Hex numbers

• 0xDEADC0DE
• 0xBA5EBA11
• 0xB01DFACE
• 0xBADA55
Using Binary #'s to represent any type of information

- “Encoding” data, simply means an agreed upon “mapping” of data from one representation to another
- At some point, it is the choice of an engineer to define the encoding or “mapping” of data between two forms
- Let’s take the alphabet as an “easy” example

How could we represent letters with #s?

What about a binary representation?

<table>
<thead>
<tr>
<th>Letter</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00000</td>
</tr>
<tr>
<td>B</td>
<td>00001</td>
</tr>
<tr>
<td>C</td>
<td>00010</td>
</tr>
<tr>
<td>D</td>
<td>00011</td>
</tr>
<tr>
<td>E</td>
<td>00100</td>
</tr>
<tr>
<td>F</td>
<td>00101</td>
</tr>
<tr>
<td>G</td>
<td>00110</td>
</tr>
<tr>
<td>H</td>
<td>00111</td>
</tr>
<tr>
<td>I</td>
<td>01000</td>
</tr>
<tr>
<td>J</td>
<td>01001</td>
</tr>
<tr>
<td>K</td>
<td>01010</td>
</tr>
<tr>
<td>L</td>
<td>01011</td>
</tr>
<tr>
<td>M</td>
<td>01100</td>
</tr>
</tbody>
</table>

Clearly, 1 bit, can only get us so far!

What if we use 2 bits?

<table>
<thead>
<tr>
<th>Letter</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>01</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
</tr>
</tbody>
</table>

Only 4 possible combinations with 2 bits: $2^2 = 4$

We need more bits to define encode all 26 letters!

How could we represent letters with binary #s?

To get 26 possible representations, we need at least 5 bits:

$2^5 = 32$

5 bits will do the trick, but we will have 6 extra slots

What about lowercase letters?

$26 + 26 = 52$ possibilities

$2^6 = 64$

That could work, but what about symbols?!
**ASCII Codes**

- Represent characters from keyboard
  - This encoding used to transfer characters between computer and all peripherals (keyboard, disk, network...)
- ASCII: American Standard Code for Information Interchange
  - 7 bits needed to encode all characters
  - Represent as 8 bit number
- Typing a key on keyboard = corresponding 8-bit ASCII code is stored and sent to computer

---

**Text: ASCII Characters**

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>ASCII</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>NUL</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>SOH</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>STX</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>ETX</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>EOT</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>ENQ</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>ACK</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>BEL</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>BS</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>HT</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>NL</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>CR</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>LF</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>VT</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>FF</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>DEL</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>!</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>&quot;</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>#</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>$</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>%</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>&amp;</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>'</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>(</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>)</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>-</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>+</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>,</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>;</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>&lt;</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>=</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>&gt;</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
<td>?</td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>`</td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>a</td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td>b</td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>c</td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>d</td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>e</td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>f</td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>g</td>
</tr>
<tr>
<td>41</td>
<td>29</td>
<td>h</td>
</tr>
<tr>
<td>42</td>
<td>2A</td>
<td>i</td>
</tr>
<tr>
<td>43</td>
<td>2B</td>
<td>j</td>
</tr>
<tr>
<td>44</td>
<td>2C</td>
<td>k</td>
</tr>
<tr>
<td>45</td>
<td>2D</td>
<td>l</td>
</tr>
<tr>
<td>46</td>
<td>2E</td>
<td>m</td>
</tr>
<tr>
<td>47</td>
<td>2F</td>
<td>n</td>
</tr>
<tr>
<td>48</td>
<td>30</td>
<td>o</td>
</tr>
<tr>
<td>49</td>
<td>31</td>
<td>p</td>
</tr>
<tr>
<td>50</td>
<td>32</td>
<td>q</td>
</tr>
<tr>
<td>51</td>
<td>33</td>
<td>r</td>
</tr>
<tr>
<td>52</td>
<td>34</td>
<td>s</td>
</tr>
<tr>
<td>53</td>
<td>35</td>
<td>t</td>
</tr>
<tr>
<td>54</td>
<td>36</td>
<td>u</td>
</tr>
<tr>
<td>55</td>
<td>37</td>
<td>v</td>
</tr>
<tr>
<td>56</td>
<td>38</td>
<td>w</td>
</tr>
<tr>
<td>57</td>
<td>39</td>
<td>x</td>
</tr>
<tr>
<td>58</td>
<td>3A</td>
<td>y</td>
</tr>
<tr>
<td>59</td>
<td>3B</td>
<td>z</td>
</tr>
</tbody>
</table>

---

**Interesting Properties of ASCII Code**

- What is the 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?

(https://www.unicode.org/)

---

**Finite Width**

- On a real computer each memory storage location can only store a finite number of bits
  - For example we can talk about a 16 bit machine, a 32 bit machine or a 64 bit machine
  - The fact that the actual storage locations are limited caps the size of the numbers that we can store and manipulate.
- These limitations also show up in programming languages where different basic types have different sizes
  - Some basic types in C
    - char – typically 8 bits
    - short int – typically 16 bits
    - int – typically 32 bits
    - long int – typically 64 bits
  - Note these sizes are not guaranteed and can change on different architectures.
Terminology

- A single binary digit is referred to as a **bit**
- A collection of 8 bits is referred to as a **byte**
- A collection of 4 bits is referred to as a **nibble**
  - Also a Hex digit
- In a computer memory each storage location can only hold a finite number of bits

```
  bit  01101010
  nibble  byte
```

Next: Machine not useful unless it can perform operations

- Arithmetic and logical operations in binary….

Binary Arithmetic

- Binary addition – just like base 10 (decimal)!
  - Add from right to left, propagating carry

```
  10010  (18)  10010  (18)  01111  (15)
  +  01001  (9)  +  01011  (11)  +  00001  (1)
  =  11011  (27)  =  11101  (29)  =  10000  (16)
```

Nice implementation of this in hardware
Subtraction, multiplication, division also similar to base 10

Question:

- 3. Add two 5-bit binary numbers 00111 and 01010
- 4. Add two 4 bit numbers: 0100 and 1100
Finite Width and Overflow

- Integers have infinite width
  - There are an infinite number of them
- Hardware integers have finite (architecture defined) width
  - Limited by hardware circuits themselves
  - 64-bit these days (2^64 integers):
  - LC3 integers are 16-bit (2^16 or ~64,000)
- Overflow: when operation result is outside type’s range
  - Example: 15 + 1 with 4-bit integers (16 needs 5 bits, 10000)

What About Negative Integers?

- Negative numbers have rights too
  - No negation without representation!
- How do we represent negative integers in decimal:
  - sign followed by value
  - -269
  - +169 is usually written as 169 (drop the + sign)
- Question: Is this a valid (as per math definition) radix 10 (decimal) representation?

Negative Integers in Binary?

- One option: sign-magnitude concept
  - What do we do with paper-and-pencil: put a ‘-’ in front
  - No ‘-’ in binary, just use a 1 in most significant bit to denote sign (0= positive, 1= negative)
    - 00101 = 5
    - 10101 = -5
- Another option: 1’s Complement
  - Simply complement bits
    - 00101 = 5
    - 11010 = -5
- Note: in both these representations, we are using an extra bit to denote the sign

What type of representation do we want?

- We would like the same arithmetic ‘algorithms’ work for negative numbers
  - Keeps hardware circuits simple
- Example: We want the same addition algorithm
  - Add starting with rightmost (least significant) bit and propagate the carry bit to the left
Signed Magnitude

- Leading bit is the **sign** bit

<table>
<thead>
<tr>
<th>( N )</th>
<th>10100</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>10100</td>
</tr>
<tr>
<td>-3</td>
<td>10011</td>
</tr>
<tr>
<td>-2</td>
<td>10010</td>
</tr>
<tr>
<td>-1</td>
<td>10001</td>
</tr>
<tr>
<td>0</td>
<td>10000</td>
</tr>
<tr>
<td>+1</td>
<td>00001</td>
</tr>
<tr>
<td>+2</td>
<td>00010</td>
</tr>
<tr>
<td>+3</td>
<td>00011</td>
</tr>
<tr>
<td>+4</td>
<td>00100</td>
</tr>
</tbody>
</table>

\[ Y = \text{"abc"} = (-1)^a (b.2^1 + c.2^0) \]

Range is: \(-2^{N-1} + 1 < i < 2^{N-1} - 1\)

**Problems: ?**

One's Complement

- Invert all bits

<table>
<thead>
<tr>
<th>( N )</th>
<th>11011</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>11011</td>
</tr>
<tr>
<td>-3</td>
<td>11100</td>
</tr>
<tr>
<td>-2</td>
<td>11101</td>
</tr>
<tr>
<td>-1</td>
<td>11110</td>
</tr>
<tr>
<td>0</td>
<td>11111</td>
</tr>
<tr>
<td>+1</td>
<td>00001</td>
</tr>
<tr>
<td>+2</td>
<td>00010</td>
</tr>
<tr>
<td>+3</td>
<td>00011</td>
</tr>
<tr>
<td>+4</td>
<td>00100</td>
</tr>
</tbody>
</table>

If msb (most significant bit) is 1 then the number is negative (same as signed magnitude)

Range is: \(-2^{N-1} + 1 < i < 2^{N-1} - 1\)

**Problems: ?**

**Question**

- What is the representation of -19 in
  - Signed magnitude binary
  - 1's complement binary

- A and B are signed magnitude binary nos.
  - \( A = 1010 \) and \( B = 0011 \)
  - What is \( A + B \)

- A and B are 1’s complement binary nos.
  - \( A = 1010 \) and \( B = 0011 \)
  - What is \( A + B \)

**Problems with Signed-Mag and 1C**

- Addition does not work!!
  - Can design a different algorithm (more complex)

- two representations for zero
  - We can represent one less number
  - \( N \) bits should represent \( 2^N \) different numbers
Two’s Complement Representation

- viewed as weighted position: weight of most significant bit is \(-2^{N-1}\)
- 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

\[
\begin{array}{c}
00101 \text{ (5)} \\
11010 \text{ (1's comp)} \\
+ 1 \\
11011 \text{ (-5)}
\end{array}
\quad\begin{array}{c}
01001 \text{ (9)} \\
01001 \text{ (1's comp)} \\
+ 1 \\
11010 \text{ (-9)}
\end{array}
\]

Question

- Two 2’s Complement numbers
  - A = 1010
  - B = 0011
- What is A+B

Two’s Complement (2C) – why does it work

- Representation designed to allow us to store and manipulate both positive (aka: +ve) and negative (aka: –ve) numbers
- To represent a number X we actually compute and store \((2^n + X)\)
- Recall \(2^n\) in binary will be a 1 followed by \(n\) zeros

Why does this work?

- Consider adding two 2C numbers:

\[
(2^n + X) + (2^n + Y) = 2^n + (2^n + (X+Y))
\]

Extra overflow bit (discarded)

In practice:

- The Most Significant Bit (MSB) in N-bit 2C representation has a weight of \(-2^{(N-1)}\)
Encoding Integers: Formal Definition

<table>
<thead>
<tr>
<th>Unsigned</th>
<th>Two’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ B_{\text{U}}(x) = \sum_{i=0}^{w} x_i \cdot 2^i ]</td>
<td>[ B_{\text{T}}(x) = -x_0 \cdot 2^{w-1} + \sum_{i=1}^{w} x_i \cdot 2^i ]</td>
</tr>
</tbody>
</table>

- \( \text{short int} x = 15213 \)
- \( \text{short int} y = -15213 \)

- \( \text{C} \) short 2 bytes long

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

- Sign Bit
  - For 2’s complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

2C Summary

- If you have the positive binary representation for a number, to find the 2C representation, simply:
  - Flip all the bits and add 1
    - OR
  - Copy bits from right to left up to and including the first ‘1’
    - Flip remaining bits
  - Techniques work in reverse as well!

Arithmetic Overflow - Summary

- For **unsigned** numbers
  - Any addition that produces an ‘extra bit’ is a problem
- For **2C signed** numbers
  - Sometimes addition or subtraction produce an extra bit – **this is not necessarily a problem.**
  - Arithmetic overflow can occur when you are adding 2 positive or 2 negative numbers – in this case if the sign of the result is different from the sign of the addends you have an arithmetic overflow
    - (this is the key to determining overflow condition in 2C)
  - Note: most CPU architectures today, use 2C representation

Sign Extension

- Suppose we have a number which is stored in a four bit register
- We wish to add this number to a number stored in an eight bit register
- We have a device which will do the addition and it is designed to add two 8 bit numbers
- What issues do we need to deal with?
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>

Question to think about: why does this work?

Arithmetic Operations...

- Addition: we’ve seen this
  - Same as decimal…add and propagate carry
- Subtraction: A – B
  - Negate B: compute 2’s complement of B
  - Add to A
- Multiplication – we’ve seen this in decimal…
  - Shift and add

Shift
- What happens if we add a number to itself?
  - (0011) + (0011) = ??

Read notes….

Shifting Bit Fields

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Pattern x</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X &lt;&lt; 1 – Left Shift by 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>X &lt;&lt; 2 – Left Shift by 2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Original Pattern x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X &gt;&gt; 1 – Shift Right (logical) by 1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>X &gt;&gt;&gt; 1 – Shift Right (arithmetic) by 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- Shift Left:
  - Move all #’s to the left, fill in empty spots with a 0
- Shift Right (2 kinds):
  - shift right logical (SRL) >>
    - shift 0’s in from the left
  - shift right arithmetic (SRA) >>>
    - replicate the sign bit, (very useful for sign extension!)
**Shifts**
- Powers of 2 are everywhere …
- … and so is multiplication by (small) powers of 2
- Another use of the \(2^n = 2 \times 2^{n-1}\) binary identity
  - Shift left by \(n\) (pushing in 0s) is the same as multiplying by \(2^n\)
  - Use \(<<\) to construct both hardware and software multipliers
  - What about shift right?
  - Think of it like multiplying by 10. Say you have 5*10, isn’t that just shifting 5 to the ten’s place?
    - 5*100, just shifting the 5 to the hundred’s place?
- Most important use of “shifting circuits”…
  - To implement multiplication in a computer (recall shift & add?)

**Comparison**
- Another useful operation is comparison
  - \(==\) (equals), \(!=\) (not equals), \(>, <, >=, <=\)
- Comparison via subtraction, \(A - B\), if result is …
  - Zero \(\rightarrow A == B\), not zero \(\rightarrow A != B\)
  - Positive \(\rightarrow A > B\), not positive \(\rightarrow A <= B\)
  - Negative \(\rightarrow A < B\), not negative \(\rightarrow A >= B\)
- Pitfall: comparison is explicitly signed or unsigned
  - +/- are not, “result” is same either way
  - Comparison interprets numbers in a way +/- don’t
  - Example, which is bigger 0110 or 1010?

**Implementing Comparison**
- How are signed and unsigned comparison implemented?
  - Let’s look at 0110 (6) and 1010 (–6 or 10) in 4-bit representation
  - If this is a signed comparison, subtraction result is positive (12)
  - If unsigned, subtraction result is negative (–4)
  - Potential problem: 12 overflows 4-bit signed representation
  - What to do? Extend to 5-bit representation, check “new” MSB
    - Signed comparison? Sign extend
    - Unsigned comparison? Zero extend

<table>
<thead>
<tr>
<th>Address</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Binary Representation Summary**
- Every storage locations stores a finite sequence of bits
  - 8-bit, 16-bit, 32-bit etc.
- The same bit string can mean different things depending on how the program wants to look at it.

Unsigned: +129
2C: -127
2C: 109
ASCII: ‘m’
Logical Operators

Propositional Logic – sound familiar?
- each variable has True (T) or False (F) value
- Use logical connectives to build more complex propositions (i.e., logic statements)
  - Connectives: AND, OR, NOT, ...
- (A AND B) is True if A is True and B is True...
- Build “truth table” for propositional ‘formula’

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A OR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>NOT A</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Another use for bits: Logic
- Beyond numbers
  - logical variables can be true or false, on or off, etc., and so are readily represented by the binary system.
  - A logical variable A can take the values false = 0 or true = 1 only.
    - Logical Variables = Propositions in propositional logic
    - Example proposition: “George is late to class” – can only be True or False
  - The manipulation of logical variables is known as Boolean Algebra, and has its own set of operations - which are not to be confused with the arithmetic operations of the previous section.
  - Some basic operations: NOT, AND, OR, XOR
  - Boolean function: function over Boolean variables and using the Boolean operators (i.e., logic operations)

Boolean Logical Operations
- Represent propositions using binary representation
- Operations on logical TRUE or FALSE variables
  - Boolean variables
    - 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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A OR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>NOT A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
**Exclusive OR**

- $(A \text{ XOR } B)$ is true if exactly one of $A$ or $B$ is true; else false

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>A XOR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Logic Operations..more examples**

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

**Basic Logic Operations**

- **Equivalent Notations**
  - $\sim A = A' = A$
  - $A \land B = A \cdot B = A \cap B$, $A \lor B = A \cup B$
- **Other common logic operations:**
  - $\land\lor = \sim(A \land \lor)$
  - $\lor\land = \sim(A \lor \land)$
  - $\land\neg = \sim(A \land \neg)$
  - $\neg\lor = \sim(A \lor \neg)$
  - $\neg\land\lor = \sim(A \land \lor)$
  - $\neg\lor\land = \sim(A \lor \land)$

**Bitwise Logical Operations**

- **View n-bit field as a collection of n logical values**
  - Apply operation to each bit independently

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

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

- Computers don't support individual bits as a data type
  - Just use least significant of n-bit integer
  - Integers are generally more useful

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11000101</td>
<td>00001111</td>
<td>00001010</td>
<td>10001111</td>
<td>10001111</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11000101</td>
<td>00001111</td>
<td>00001010</td>
<td>10001111</td>
<td>10001111</td>
<td></td>
</tr>
</tbody>
</table>

11
a little bit of “reality”

- look at how some of the concepts we have studied take shape in ‘real life’
  - C programming and O/S

- Go through the lecture notes posted on my webpage
  - As you learn C, try out the operators discussed in the notes
  - I will loop back to this material and summarize in 3 weeks

Logical Operations in C

- C supports both bitwise and boolean logic operations
  - x & y    bitwise logic operation
  - x && y   boolean operation: output is boolean value

- What’s going on here?
  - In boolean operation the result has to be TRUE (1) or FALSE (0)
  - Treats any non-zero argument as TRUE and returns only TRUE (1) or FALSE (0)

- In C: logical operators do not evaluate their second argument if result can be obtained from first
  - a && 5/a  can we get divide by zero error?

Bitwise Review

- Can only be applied to integral operands
- that is, char, short, int and long
- (signed or unsigned)
  &  Bitwise AND
  |  Bitwise OR
  ^  Bitwise XOR
  <<  Shift Left
  >>  Shift Right
  ~   1’s Complement (Inversion)

Question: Bitwise operations in C

- assume 4 bit
- What is (4 & 6): 0100 & 0110 ?

- What is (4 ^ 6): 0100 ^ 0110 ?

- What is ( ~4)

- What is (4 && 6 ): 0100 && 0110 ?
Limitations of integer representations?.. do we need anything else?

- Most numbers are not integer!
  - Even with integers, there are other considerations

- Range:
  - The magnitude of the numbers we can represent is determined by how many bits we use:
    - e.g. with 32 bits the largest number we can represent is about +/- 2 billion, far too small for many purposes.

- Precision:
  - The exactness with which we can specify a number:
    - e.g. a 32 bit number gives us 31 bits of precision, or roughly 9 figure precision in decimal representation.

- How to deal with Real numbers...We need other data types!

How to deal with complicated real numbers...Some History...

- The Indiana Legislature once introduced legislation declaring that the value of $\pi$ was exactly 3.2

Scientific Notation

- $-6.023 \times 10^{-23}$

IEEE-754
Fixed-Point

- How can we represent fractions?
  - “binary point” separate positive from negative powers of two
    - Analogous to “decimal point”: .75 = (7/10)+(5/100)
  - 2's addition and subtraction still work
    - If binary points are aligned (“fixed-point”)

\[\begin{align*}
2^{-1} &= 0.5 \\
2^{-2} &= 0.25 \\
2^{-3} &= 0.125 \\
00101000.101 &= (40.625) \\
+11111110.110 &= (-1.25) \\
00100111.011 &= (39.375)
\end{align*}\]

Very Large and Very Small: Floating-Point

- Problem
  - Large values: \(6.022 \times 10^{23}\) requires 79 bits
  - Small values: \(6.626 \times 10^{-34}\) requires >110 bits

- Solution: use equivalent of “scientific notation”: \(F \times 2^E\)
  - Need to represent \(F\) (fraction), \(E\) (exponent), and \(S\) (sign)

IEEE 754 Floating-Point Standard

\[\begin{array}{c|c}
S & Exponent & Fraction \\
\hline
0 & 00000000 & 00000000000000000000000 \\
1 & exponent & mantissa (significand) \\
\hline
(-1)^S \times 1.M \times 2^{E-127}
\end{array}\]

IEEE 754 Floating-Point Standard

- 32-bit (“single-precision” or float)
  - 8-bit exponent, 23-bit fraction
  - \(X = -1^S \times 1.fraction \times 2^{exponent-127}, 1 \leq exponent \leq 254\)
    - Exponent representation is called “excess notation”

- 64-bit (“double-precision” or double)
  - 11-bit exponent, 52-bit fraction
  - \(X = -1^S \times 1.fraction \times 2^{exponent-1023}, 1 \leq exponent \leq 2046\)

- Representation must be “normalized” (just like decimal)
  - \(1 \leq Fraction < 2\) (fraction to left of binary point must be 1)
    - This 1 is implicit in Fraction
Floating-Point Example

- What is this?
- \[ \begin{array}{c}
  10111111010000000000000000000000 \\
  \hline
  \text{sign exponent fraction}
\end{array} \]

- Sign is 1: number is negative
- Exponent is 01111110 = 126 (decimal)
- Fraction is 0.100000000000… = 1/102 = 0.5 (decimal)
- Value = \(-1.5 \times 2^{(126-127)} = -1.5 \times 2^{-1} = -0.75\)

More floating point

- Reading assignment – Chapter 2

What next..

- The hardware building blocks and their operations – Chapter 3
- Digital Logic structures
  - Basic device operations: CMOS transistor
  - Combinational Logic circuits
    - Gates (NAND, OR, NOT), Decoder, Multiplexer
    - Adders, multipliers
  - Sequential circuits—concept of memory
    - Finite state machines, memory organization
    - Basic storage elements: latches, flip-flops
- Labs – build logic circuits
- HW1 posted – Due next Tuesday 2pm
- Quiz 1 – on Tuesday