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.

Digital system:
- finite number of symbols

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

Basic unit of information is the binary digit, or bit.

Values with more than two 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, ...
- **Logical** – true, false
- **Text** – characters, strings, ...
- **Instructions (binary)** – LC-3, x-86 ..
- **Images** – jpeg, gif, bmp, png ...
- **Sound** – mp3, wav..
- ...

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

\[
\begin{align*}
\text{329} &= 3 \times 100 + 2 \times 10 + 9 \times 1 = 329 \\
\text{101} &= 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = 5
\end{align*}
\]
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>
<th></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>
Unsigned Binary Arithmetic

Base-2 addition – just like base-10!

- add from right to left, propagating carry

\[
\begin{align*}
10010 & + 1001 & 10010 & + 1011 & 1111 & + 1 \\
11011 & & 11101 & & 10000 & \\
\end{align*}
\]

carry

\[
\begin{align*}
10111 & + 111 & \\
+ 111 & \\
\end{align*}
\]

Subtraction, multiplication, division,…
**Signed Integers**

With n bits, we have $2^n$ distinct values.

- assign about half to positive integers (1 through $2^{n-1}$) and 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: formats**

- sign-magnitude – set MS 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
Two’s Complement

Problems with sign-magnitude and 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

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

\[
\begin{array}{c}
00101 \quad (5) \\
+ \quad 11011 \quad (-5) \\
00000 \quad (0)
\end{array}
\quad
\begin{array}{c}
01001 \quad (9) \\
+ \quad \underline{\hspace{1cm}} \quad (-9) \\
00000 \quad (0)
\end{array}
\]
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

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

\[
\begin{align*}
01001 & \quad (9) \\
11010 & \quad (1’s \ comp) \\
+1 & \\
11011 & \quad (-9)
\end{align*}
\]
Two’s Complement Shortcut

To take the two’s complement of a number:

- copy bits from right to left until (and including) the first “1”
- flip remaining bits to the left

\[
\begin{align*}
011010000 & \quad \text{(flip)} \\
100101111 & \quad \text{(1’s comp)} \\
+ & \quad 1 \\
100110000 & \quad \text{(copy)} \\
\end{align*}
\]
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 (\(-2^{n-1}\)) has no positive counterpart.

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

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

Assuming 8-bit 2’s complement numbers.

\[
\begin{array}{c|c}
 n & 2^n \\
\hline
 0 & 1 \\
 1 & 2 \\
 2 & 4 \\
 3 & 8 \\
 4 & 16 \\
 5 & 32 \\
 6 & 64 \\
 7 & 128 \\
 8 & 256 \\
 9 & 512 \\
 10 & 1024 \\
\end{array}
\]
More Examples

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

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

\[
X = -26_{\text{ten}}
\]

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

First Method: Division

1. Find magnitude of decimal number. (Always positive.)
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; if original number was negative, take two’s complement.

\[ X = 104_{\text{ten}} \]

\[
\begin{align*}
104/2 &= 52 \text{ r0 bit 0} \\
52/2 &= 26 \text{ r0 bit 1} \\
26/2 &= 13 \text{ r0 bit 2} \\
13/2 &= 6 \text{ r1 bit 3} \\
6/2 &= 3 \text{ r0 bit 4} \\
3/2 &= 1 \text{ r1 bit 5} \\
1/2 &= 0 \text{ r1 bit 6}
\end{align*}
\]

\[ X = 01101000_{\text{two}} \]
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.

<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>

$X = 104_{\text{ten}}$

\[
\begin{align*}
104 - 64 &= 40 & \text{bit 6} \\
40 - 32 &= 8  & \text{bit 5} \\
8 - 8 &= 0   & \text{bit 3}
\end{align*}
\]

$X = 01101000_{\text{two}}$
Operations: Arithmetic and Logical

Recall:
a data type includes *representation* and *operations*.

We now have a good representation for signed integers, so let’s look at some arithmetic operations:

- Addition
- Subtraction
- Sign Extension

We’ll also look at overflow conditions for addition.

Multiplication, division, etc., 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

```
01101000  (104)  11110110  (-10)
+ 11110000  (-16)  +_____________  (-9)
01011000  (98)  
```

Assuming 8-bit 2’s complement numbers.
Subtraction

Negate subtrahend (2nd no.) 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

\[
\begin{align*}
01101000 \ (104) & \quad 11110110 \ (-10) \\
- \underline{00010000} \ (16) & \quad - \underline{00001000} \ (-9) \\
01101000 \ (104) & \quad 11110110 \ (-10) \\
+ \underline{11110000} \ (-16) & \quad + \underline{11110110} \ (-9) \\
01011000 \ (88) & \quad 01011000 \ (88)
\end{align*}
\]

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</td>
</tr>
<tr>
<td>1100</td>
<td>00001100</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</td>
</tr>
<tr>
<td>1100</td>
<td>11111100</td>
</tr>
</tbody>
</table>

(still 4) (still -4)
Overflow

If operands are too big, then sum cannot be represented as an \( n \)-bit 2’s comp number.

\[
\begin{array}{c}
  01000 \quad (8) \\
+ 01001 \quad (9) \\
  10001 \quad (-15)
\end{array}
\begin{array}{c}
  11000 \quad (-8) \\
+ 10111 \quad (-9) \\
  01111 \quad (+15)
\end{array}
\]

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

<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>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<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>1</td>
<td>1</td>
<td></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

AND

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

OR

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

NOT

- unary operation -- one argument
- flips every bit

\[
\begin{array}{c}
\text{11000101} \\
\text{AND} \\
\text{00001111} \\
\text{00000101}
\end{array}
\]

\[
\begin{array}{c}
\text{11000101} \\
\text{OR} \\
\text{00001111} \\
\text{11001111}
\end{array}
\]

\[
\begin{array}{c}
\text{11000101} \\
\text{NOT} \\
\text{11000101} \\
\text{00111010}
\end{array}
\]
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>
</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>
Converting from Binary to Hexadecimal

Every four bits is a hex digit.

- start grouping from right-hand side

```
0111 0101 0100 1111 0100 1101 0111 01
```

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

\[
\begin{align*}
2^{-1} &= 0.5 \\
2^{-2} &= 0.25 \\
2^{-3} &= 0.125
\end{align*}
\]

\[
\begin{align*}
00101000.101 & \quad (40.625) \\
+ 11111110.110 & \quad (-1.25) \\
\hline
00100111.011 & \quad (39.375)
\end{align*}
\]

No new operations -- same as integer arithmetic.
Very Large and Very Small: Floating-Point

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):

\[
\begin{array}{c|c|c}
S & \text{Exponent} & \text{Fraction} \\
\hline
1b & 8b & 23b \\
\end{array}
\]

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

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

Single-precision IEEE floating point number:

\[
\begin{array}{c}
101111110100000000000000000000000000000000
\end{array}
\]

- Sign is 1 – number is negative.
- Exponent field is 01111110 = 126 (decimal).
- Fraction is 0.100000000000… = 0.5 (decimal).

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 $3.07 \times 10^{12} + 9.11 \times 10^8$?
Need to work with exponents )
Text: ASCII Characters

ASCII: Maps 128 characters to 7-bit code.

- both printable and non-printable (ESC, DEL, ...) characters

<table>
<thead>
<tr>
<th></th>
<th>ASCII Code</th>
<th>Character</th>
<th>ASCII Code</th>
<th>Character</th>
<th>ASCII Code</th>
<th>Character</th>
<th>ASCII Code</th>
<th>Character</th>
<th>ASCII Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>nul</td>
<td>00</td>
<td>10</td>
<td>dle</td>
<td>20</td>
<td>sp</td>
<td>30</td>
<td>0</td>
<td>40</td>
<td>@</td>
</tr>
<tr>
<td>01</td>
<td>soh</td>
<td>11</td>
<td>dc1</td>
<td>21</td>
<td>!</td>
<td>31</td>
<td>1</td>
<td>41</td>
<td>A</td>
<td>51</td>
</tr>
<tr>
<td>02</td>
<td>stx</td>
<td>12</td>
<td>dc2</td>
<td>22</td>
<td>&quot;</td>
<td>32</td>
<td>2</td>
<td>42</td>
<td>B</td>
<td>52</td>
</tr>
<tr>
<td>03</td>
<td>etx</td>
<td>13</td>
<td>dc3</td>
<td>23</td>
<td>#</td>
<td>33</td>
<td>3</td>
<td>43</td>
<td>C</td>
<td>53</td>
</tr>
<tr>
<td>04</td>
<td>eot</td>
<td>14</td>
<td>dc4</td>
<td>24</td>
<td>$</td>
<td>34</td>
<td>4</td>
<td>44</td>
<td>D</td>
<td>54</td>
</tr>
<tr>
<td>05</td>
<td>enq</td>
<td>15</td>
<td>nak</td>
<td>25</td>
<td>%</td>
<td>35</td>
<td>5</td>
<td>45</td>
<td>E</td>
<td>55</td>
</tr>
<tr>
<td>06</td>
<td>ack</td>
<td>16</td>
<td>syn</td>
<td>26</td>
<td>&amp;</td>
<td>36</td>
<td>6</td>
<td>46</td>
<td>F</td>
<td>56</td>
</tr>
<tr>
<td>07</td>
<td>bel</td>
<td>17</td>
<td>etb</td>
<td>27</td>
<td>'</td>
<td>37</td>
<td>7</td>
<td>47</td>
<td>G</td>
<td>57</td>
</tr>
<tr>
<td>08</td>
<td>bs</td>
<td>18</td>
<td>can</td>
<td>28</td>
<td>(</td>
<td>38</td>
<td>8</td>
<td>48</td>
<td>H</td>
<td>58</td>
</tr>
<tr>
<td>09</td>
<td>ht</td>
<td>19</td>
<td>em</td>
<td>29</td>
<td>)</td>
<td>39</td>
<td>9</td>
<td>49</td>
<td>I</td>
<td>59</td>
</tr>
<tr>
<td>0a</td>
<td>nl</td>
<td>1a</td>
<td>sub</td>
<td>2a</td>
<td>*</td>
<td>3a</td>
<td>:</td>
<td>4a</td>
<td>J</td>
<td>5a</td>
</tr>
<tr>
<td>0b</td>
<td>vt</td>
<td>1b</td>
<td>esc</td>
<td>2b</td>
<td>+</td>
<td>3b</td>
<td>;</td>
<td>4b</td>
<td>K</td>
<td>5b</td>
</tr>
<tr>
<td>0c</td>
<td>np</td>
<td>1c</td>
<td>fs</td>
<td>2c</td>
<td>,</td>
<td>3c</td>
<td>&lt;</td>
<td>4c</td>
<td>L</td>
<td>5c</td>
</tr>
<tr>
<td>0d</td>
<td>cr</td>
<td>1d</td>
<td>gs</td>
<td>2d</td>
<td>-</td>
<td>3d</td>
<td>=</td>
<td>4d</td>
<td>M</td>
<td>5d</td>
</tr>
<tr>
<td>0e</td>
<td>so</td>
<td>1e</td>
<td>rs</td>
<td>2e</td>
<td>.</td>
<td>3e</td>
<td>&gt;</td>
<td>4e</td>
<td>N</td>
<td>5e</td>
</tr>
<tr>
<td>0f</td>
<td>si</td>
<td>1f</td>
<td>us</td>
<td>2f</td>
<td>/</td>
<td>3f</td>
<td>?</td>
<td>4f</td>
<td>O</td>
<td>5f</td>
</tr>
</tbody>
</table>
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?

Unicode: 128 characters are not enough. 1990s Unicode was standardized, Java used Unicode.

No new operations -- integer arithmetic and logic.
Other Data Types

Text strings
- sequence of characters, terminated with NULL (0)
- typically, no hardware support

Image
- array of pixels
  - monochrome: one bit (1/0 = black/white)
  - color: red, green, blue (RGB) components (e.g., 8 bits each)
  - 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, etc., in the software that we write.