Chapter 2
Bits, Data Types, and Operations

Slides based on set prepared by
Gregory T. Byrd, North Carolina State University
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

We’ll see examples of these circuits in the next chapter.
Representing Data with Physical Artifacts

Recognize this photo?
• Hanging chad

Was a vote cast or not?
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, …
- **Text** – characters, strings, …
- **Images** – pixels, colors, shapes, …
- **Sound**
- **Logical** – true, false
- **Instructions**
- …

Data type:
- *representation* and *operations* within the computer

We’ll start with numbers…
Unsigned Integers

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

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

\[
\begin{align*}
329 &= 3 \times 10^2 + 2 \times 10^1 + 9 \times 10^0 \\
101 &= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0
\end{align*}
\]

most significant

least significant
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>Value</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

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

\[\begin{align*}
10111 + 111 & = 101111
\end{align*}\]

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 bit
  
  $00101 = 5$

**Negative integers**

- sign-magnitude – set top 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 two one’s complement numbers?
    - e.g., try 4 + (-3)

Two’s complement representation developed to make circuits easy for arithmetic.

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

\[
\begin{array}{c}
00101 \quad (5) \\
+ \quad 11011 \quad (-5) \\
\hline
00000 \quad (0)
\end{array} \quad \begin{array}{c}
01001 \quad (9) \\
+ \quad \_\_\_\_\_ \quad (-9) \\
\hline
00000 \quad (0)
\end{array}
\]
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

\[
\begin{array}{c}
011010000 \\
100101111 \quad (1’s \ comp) \\
+ 1 \\
100110000 \quad (flip) \\
\end{array}
\]
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>(\text{Decimal Value})</th>
<th>(-2^3)</th>
<th>(2^2)</th>
<th>(2^1)</th>
<th>(2^0)</th>
<th>(\text{Decimal Value})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-8</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-7</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>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.

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

Assuming 8-bit 2’s complement numbers.
More Examples

<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>

$X = 00100111_{\text{two}}$

$$= 2^5 + 2^2 + 2^1 + 2^0 = 32 + 4 + 2 + 1$$

$$= 39_{\text{ten}}$$

$X = 11100110_{\text{two}}$

$-X = 00011010$

$$= 2^4 + 2^3 + 2^1 = 16 + 8 + 2$$

$$= 26_{\text{ten}}$$

$X = -26_{\text{ten}}$

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

First Method: *Division*

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

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

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

\[ X = 01101000_{\text{two}} \]
Converting Decimal to Binary (2’s C)

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>( 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

\[
\begin{array}{c@{}c@{}c@{}c@{}c@{}c@{}c@{}c@{}c}
0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
+ & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
\hline
0 & 1 & 0 & 1 & 1 & 0 & 0 & 0
\end{array}
\]

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

\[
\begin{align*}
01101000 & \quad 11110110 \quad (-10) \\
- 00010000 & \quad + \quad \quad \quad \quad (-9) \\
01101000 & \quad 11110110 \quad (-10) \\
+ 11110000 & \quad + \quad \quad \quad \quad (9) \\
01011000 & \quad (88) \quad \quad \quad \quad (-1)
\end{align*}
\]

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 (4)</td>
<td>000000100 (still 4)</td>
</tr>
<tr>
<td>1100 (-4)</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 (4)</td>
<td>000000100 (still 4)</td>
</tr>
<tr>
<td>1100 (-4)</td>
<td>111111100 (still -4)</td>
</tr>
</tbody>
</table>
Overflow

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

\[
\begin{align*}
01000 & \quad (8) \\
+ 01001 & \quad (9) \\
\underline{10001} & \quad (-15)
\end{align*}
\quad 
\begin{align*}
11000 & \quad (-8) \\
+ 10111 & \quad (-9) \\
01111 & \quad (+15)
\end{align*}
\]

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

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

<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>1</td>
<td>0</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>1</td>
<td>0</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>AND</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000101</td>
<td>00000111</td>
<td>11000101</td>
</tr>
<tr>
<td>AND</td>
<td>00000111</td>
<td>OR</td>
</tr>
<tr>
<td>00000101</td>
<td>11001111</td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>11000101</td>
<td>00111010</td>
</tr>
</tbody>
</table>
**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>
<th>Binary</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
<td>1010</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
<td>1011</td>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>1100</td>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>1101</td>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>1110</td>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<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

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & & & \\
\end{array}
\]

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

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

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

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):

$N = -1^S \times 1.fraction \times 2^{exponent - 127}$, $1 \leq exponent \leq 254$

$N = -1^S \times 0.fraction \times 2^{-126}$, exponent = 0
Floating Point Example

Single-precision IEEE floating point number:

\[ 10111111010000000000000000000000 \]

- 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 $3.07 \times 10^{12} + 9.11 \times 10^{8}$?)
# ASCII Characters

**ASCII:** Maps 128 characters to 7-bit code.

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

<table>
<thead>
<tr>
<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 nul</td>
<td>0</td>
<td>01 soh</td>
<td>1</td>
<td>02 stx</td>
<td>2</td>
</tr>
<tr>
<td>03 etx</td>
<td>3</td>
<td>04 eot</td>
<td>4</td>
<td>05 enq</td>
<td>5</td>
</tr>
<tr>
<td>06 ack</td>
<td>6</td>
<td>07 bel</td>
<td>7</td>
<td>08 bs</td>
<td>8</td>
</tr>
<tr>
<td>09 ht</td>
<td>9</td>
<td>0a nl</td>
<td>10 dle</td>
<td>0b vt</td>
<td>11 dc1</td>
</tr>
<tr>
<td>0c np</td>
<td>12 dc2</td>
<td>0d cr</td>
<td>13 dc3</td>
<td>0e so</td>
<td>14 dc4</td>
</tr>
<tr>
<td>0f si</td>
<td>15 nak</td>
<td>01 soh</td>
<td>16 syn</td>
<td>02 stx</td>
<td>17 etb</td>
</tr>
<tr>
<td>03 etx</td>
<td>18 can</td>
<td>04 eot</td>
<td>19 em</td>
<td>05 enq</td>
<td>20 sp</td>
</tr>
<tr>
<td>06 ack</td>
<td>21 !</td>
<td>07 bel</td>
<td>22 &quot;</td>
<td>08 bs</td>
<td>23 #</td>
</tr>
<tr>
<td>09 ht</td>
<td>24 $</td>
<td>0a nl</td>
<td>25 %</td>
<td>0b vt</td>
<td>26 &amp;</td>
</tr>
<tr>
<td>0c np</td>
<td>27 '</td>
<td>0d cr</td>
<td>28 (</td>
<td>0e so</td>
<td>29 )</td>
</tr>
<tr>
<td>0f si</td>
<td>30 0</td>
<td>01 soh</td>
<td>31 1</td>
<td>02 stx</td>
<td>32 2</td>
</tr>
<tr>
<td>03 etx</td>
<td>33 3</td>
<td>04 eot</td>
<td>34 4</td>
<td>05 enq</td>
<td>35 5</td>
</tr>
<tr>
<td>06 ack</td>
<td>36 6</td>
<td>07 bel</td>
<td>37 7</td>
<td>08 bs</td>
<td>38 8</td>
</tr>
<tr>
<td>09 ht</td>
<td>39 9</td>
<td>0a nl</td>
<td>40 @</td>
<td>0b vt</td>
<td>41 A</td>
</tr>
<tr>
<td>0c np</td>
<td>42 B</td>
<td>0d cr</td>
<td>43 C</td>
<td>0e so</td>
<td>44 D</td>
</tr>
<tr>
<td>0f si</td>
<td>45 E</td>
<td>01 soh</td>
<td>46 F</td>
<td>02 stx</td>
<td>47 G</td>
</tr>
<tr>
<td>03 etx</td>
<td>48 H</td>
<td>04 eot</td>
<td>49 I</td>
<td>05 enq</td>
<td>50 P</td>
</tr>
<tr>
<td>06 ack</td>
<td>51 Q</td>
<td>07 bel</td>
<td>52 R</td>
<td>08 bs</td>
<td>53 S</td>
</tr>
<tr>
<td>09 ht</td>
<td>54 T</td>
<td>0a nl</td>
<td>55 U</td>
<td>0b vt</td>
<td>56 V</td>
</tr>
<tr>
<td>0c np</td>
<td>57 W</td>
<td>0d cr</td>
<td>58 X</td>
<td>0e so</td>
<td>59 Y</td>
</tr>
<tr>
<td>0f si</td>
<td>60 `</td>
<td>01 soh</td>
<td>61 a</td>
<td>02 stx</td>
<td>62 b</td>
</tr>
<tr>
<td>03 etx</td>
<td>63 c</td>
<td>04 eot</td>
<td>64 d</td>
<td>05 enq</td>
<td>65 e</td>
</tr>
<tr>
<td>06 ack</td>
<td>66 f</td>
<td>07 bel</td>
<td>67 g</td>
<td>08 bs</td>
<td>68 h</td>
</tr>
<tr>
<td>09 ht</td>
<td>69 i</td>
<td>0a nl</td>
<td>70 p</td>
<td>0b vt</td>
<td>71 q</td>
</tr>
<tr>
<td>0c np</td>
<td>72 r</td>
<td>0d cr</td>
<td>73 s</td>
<td>0e so</td>
<td>74 t</td>
</tr>
<tr>
<td>0f si</td>
<td>75 u</td>
<td>01 soh</td>
<td>76 v</td>
<td>02 stx</td>
<td>77 w</td>
</tr>
<tr>
<td>03 etx</td>
<td>78 x</td>
<td>04 eot</td>
<td>79 y</td>
<td>05 enq</td>
<td>80 del</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?

Are 128 characters enough?
(http://www.unicode.org/)

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-2/LC-3 Data Types

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

For LC-2, there is only one 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.