5

Integer Numbers

The Number Bases of Integers

Textbook Chapter 2 +
Number Systems

• Unary, or marks:
  - ////////////// = 7
  - ////////////// + ////////////// = /////////////////

• Grouping lead to Roman Numerals:
  - VII + V = XVII = XII

• Better: Arabic Numerals:
  - 7 + 5 = 12 = 1·10 + 2
Positional Number System

- The value represented by a digit depends on its *position* in the number.

Ex: 1832

- 1000
- 100
- 10
- 1
Sexagesimal: A Positional Number System

- First used over 4000 years ago in Mesopotamia
- Base 60 (Sexagesimal), alphabet: 1..60, written as 60 different groups
- But the Babylonians used only two symbols, 1 and 10, and didn’t have the zero
  - Needed context to tell 1 from 60!
- Example
  - $5,45_{60} = 5 \times 60^1 + 45 \times 60^0$
  - $300 + 45 = 345_{10}$
Babylonian Numbers
Positional Number Systems

- Select a number as the base $b$
- Define an alphabet of $b-1$ symbols plus a symbol for zero to represent all numbers
- Use an ordered sequence of 1 or more digits $d$ to represent numbers
- The represented number is the sum of all digits, each multiplied by $b$ to the power of the digit’s position $p$

$$\text{Number} = \sum_{p=0}^{\text{num digits} - 1} (d_p \cdot b^p)$$
Arabic/Indic Numerals

• Base (or radix): 10 (decimal)
  – The alphabet (digits or symbols) is 0..9
• Ours based on the Arabic symbols
  – Has the ZERO!!!
• Numerals introduced to Europe by Leonardo Fibonacci in his *Liber Abaci*
  – In 1202
  – So useful!
Arabic/Indic Numerals

- The Italian mathematician Leonardo Fibonacci
- Also known for the Fibonacci sequence
  - 1, 1, 2, 3, 5, 8, 13, 21

<table>
<thead>
<tr>
<th>European</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arabic-Indic</td>
<td></td>
<td>٠</td>
<td>١</td>
<td>٢</td>
<td>٣</td>
<td>٤</td>
<td>٥</td>
<td>٦</td>
<td>٧</td>
<td>٨</td>
</tr>
<tr>
<td>Eastern Arabic-Indic (Persian and Urdu)</td>
<td></td>
<td>٠</td>
<td>١</td>
<td>٢</td>
<td>٣</td>
<td>٤</td>
<td>٥</td>
<td>٦</td>
<td>٧</td>
<td>٨</td>
</tr>
<tr>
<td>Devanagari (Hindi)</td>
<td></td>
<td>०</td>
<td>१</td>
<td>२</td>
<td>३</td>
<td>४</td>
<td>५</td>
<td>६</td>
<td>७</td>
<td>८</td>
</tr>
<tr>
<td>Tamil</td>
<td></td>
<td>ழ</td>
<td>௨</td>
<td>௧</td>
<td>௨</td>
<td>ர</td>
<td>ய</td>
<td>ஸ</td>
<td>வ</td>
<td>ந</td>
</tr>
</tbody>
</table>

Maxwell James Dunne – Fall 2015
Numbers for Computers

- There are many ways to represent a number
- Representation does not affect computation result
- Representation affects difficulty of computing results
- Computers need a representation that works with (fast) electronic circuits
- Positional numbers work great with 2-state devices

Which base should we use for computers?
Binary Number System

- Base (radix): 2
- Digits (symbols): 0, 1
- Binary Digits, or bits
- Example:
  \[ 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1001_2 = 8 + 1 = 9_{10} \]
  \[ 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 11000_2 = 16 + 8 = 24_{10} \]
## Knowing The Powers Of Two

- Know them in your sleep

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^0$</td>
<td>1</td>
</tr>
<tr>
<td>$2^1$</td>
<td>2</td>
</tr>
<tr>
<td>$2^2$</td>
<td>4</td>
</tr>
<tr>
<td>$2^3$</td>
<td>8</td>
</tr>
<tr>
<td>$2^4$</td>
<td>16</td>
</tr>
<tr>
<td>$2^5$</td>
<td>32</td>
</tr>
<tr>
<td>$2^6$</td>
<td>64</td>
</tr>
<tr>
<td>$2^7$</td>
<td>128</td>
</tr>
<tr>
<td>$2^8$</td>
<td>256</td>
</tr>
<tr>
<td>$2^9$</td>
<td>1024</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>2048</td>
</tr>
</tbody>
</table>

- Highlighted values: 512 and 1024.
Octal Number System

- Base (radix): 8
- Digits (symbols): 0, 1, 2, 3, 4, 5, 6, 7
- \(345_8 = 3 \times 8^2 + 4 \times 8^1 + 5 \times 8^0 = 3 \times 64 + 4 \times 8 + 5 \times 1 = 192 + 32 + 5 = 229_{10}\)
- \(1001_8 = 1 \times 8^3 + 0 \times 8^2 + 0 \times 8^1 + 1 \times 8^0 = 1 \times 512 + 0 \times 64 + 0 \times 8 + 1 \times 1 = 512 + 1 = 513_{10}\)
- In C, octal numbers are represented with a leading 0 (0345 or 01001).
Hexadecimal Number System

- Base (radix): 16
- Digits (symbols): 0-9, A–F (a–f)
- In C: leading “0x” (e.g., 0xa3)
- In LC-3: leading “x” (e.g., “x3000”)
- Hexadecimal is also known as “hex” for short

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>
Examples of Converting Hex to Decimal

- \( A3_{16} = A \times 16^1 + 3 \times 16^0 \)
  \[ = 10 \times 16 + 3 \times 1 \]
  \[ = 160 + 3 \]
  \[ = 163_{10} \]

- \( 3E8_{16} = 3 \times 16^2 + E \times 16^1 + 8 \times 16^0 \)
  \[ = 3 \times 256 + 14 \times 16 + 8 \times 1 \]
  \[ = 768 + 224 + 8 \]
  \[ = 1000 \]
Base Conversion

Three cases:

I. From any base $b$ to base 10
II. From base 10 to any base $b$
III. From any base $b$ to any other base $c$
From Base $b$ to Base 10

- Base (radix): $b$
- Digits (symbols): 0 ... $(b - 1)$
- $S_{n-1}S_{n-2}...S_2S_1S_0$

$$\text{Value} = \sum_{i=0}^{n-1} (S_i b^i)$$

Use summation to transform any base to decimal
From Base $b$ to Base 10

- Example: $1234_5 = ?_{10}$

\[
\begin{align*}
(-3)^0 & \quad 1 \times 5^3 + 2 \times 5^2 + 3 \times 5^1 + 4 \times 5^0 \\
(-3)^1 & \quad 1 \times 125 + 2 \times 25 + 3 \times 5 + 4 \times 1 \\
(-3)^2 & \quad 125 + 50 + 15 + 4 \\
& \quad 194
\end{align*}
\]
From Base 10 to Base $b$ – Method 1

- Use successive divisions
- Remember the remainders
- Divide again with the quotients

```
```

\[
\begin{align*}
q &= N/b \\
r &= N \% b\\nN &= q\\n\text{if } q \neq 0 \\
&\quad \text{then } i++ \\
&\quad r = \text{ith digit of } Nb \\
&\quad N = q \\
&\text{end if} \\
&\text{end while } q \neq 0 \\
\text{END}
\end{align*}
\]
From Base 10 to Base $b$ – Method 1

$i = 0$

- Example: $2010_{10} = ?_5$

$2010 / 5 = 402$
$402 / 5 = 80 \text{ remainder } 2$
$80 / 5 = 16 \text{ remainder } 0$
$16 / 5 = 3 \text{ remainder } 1$
$3 / 5 = 0 \text{ remainder } 3$

$31020_5$

Algorithm:

- $i = 0$
- $q = N / b$
- $r = N \% b$
- $N = q$
- $r$ is the $i$th digit of $N_b$
- $++i$
- $F$: $q = 0$;
- $T$: END
From Base 10 to Base $b$ – Method 2

- Know your powers of the second base
- Subtract out the largest power of second base that fits
- Multiple by scalar, in case of binary, only a 1, so easy
- Put 1 in position for binary, scalar in position for other bases
- Repeat with remainder

$$7_{10} = ?_3$$

21
From Base 10 to Base $b$ – Method 2

- Example: $210_{10} = ?_2$
  
  \[
  210 - 128 = 82 \\
  82 - 64 = 18 \\
  18 - 16 = 2
  \]

- Example: $57_{10} = ?_3$
  
  \[
  57 - 27 \times 3 = 0 \\
  27 - 27 = 0
  \]
\[ 2 \times 3^3 + 0 \times 3^2 + 1 \times 3^1 + 0 \times 3^0 \]
From Base $b$ to Base $c$

- Use a known intermediate base
- The easiest way is to convert from base $b$ to base 10 first, and then from 10 to $c$
- Or, in some cases, it is easier to use base 2 as the intermediate base (we’ll see them soon)
Positional Multiplication

\[
\begin{array}{c}
113_5 \\
42_5 \\
\hline
231 \\
10120 \\
\hline
10401
\end{array}
\]

\[
22_5 = \overset{2}{1} \overset{2}{0}
\]

\[
2 \times 5^1 + 2 \times 5^0 = 12
\]

(a lot easier!)
Decimal To Binary Conversion: Method 1

• Divide decimal value by 2 until the value is 0
• Example: $444_{10}$
  – Divide 444 by 2; what is the remainder?
  – Divide 222 by 2; what is the remainder?
  – ...
  – Result is 0: done

• Write the remainders starting from the least significant position (the right to the left)
Decimal To Binary Conversion: Method 2

• Know your powers of two and subtract
  ... 256 128 64 32 16 8 4 2 1

• Example: 61\textsubscript{10}
  – What is the biggest power of two that fits?
  – What is the remainder?
  – What fits?
  – What is the remainder?
  – ...
  – What is the binary representation?
Binary to Octal Conversion

• Group into 3 starting at least significant bit
  – Why 3?
  – Add leading 0 as needed
    • Why not trailing 0s?
• Write one octal digit for each group

\[ 8 = 2^3 \]
Binary to Octal Conversion: Examples

\[\begin{align*}
2^3 & = 1 \\
2^2 & = 2 \\
2^1 & = 4 \\
\text{• 100 010 111} & \text{ (binary)} \\
\end{align*}\]

\[\begin{align*}
4 & = 4 \\
2 & = 2 \\
7 & = 1 \\
\text{4 + 2 + 1} & \text{ (octal)} \\
\end{align*}\]

\[\begin{align*}
\text{• 010 101 110} & \text{ (binary)} \\
\end{align*}\]

\[\begin{align*}
2^2 & = 4 \\
2^0 & = 1 \\
\text{4 + 1} & \text{ (octal)} \\
\end{align*}\]

Maxwell James Dunne – Fall 2015
Octal to Binary Conversion

- Write down the 3-bit binary code for each octal digit
- Example:
  - 047_

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

• Group into 4 starting at least significant bit
  – Why 4? $\left( 1 \leq 2^4 \right)$
  – Add leading 0 if needed
• Write one hex digit for each group

byte 8 bits

nibble 4 bits
Binary to Hex Conversion: Examples

- 1001 1110 0111 0000 (binary)
  \[
  \begin{array}{cccc}
  9 & E & 7 & 0 \\
  \\end{array}
  \] (hex)

- 0001 1111 1010 0011 (binary)
  \[
  \begin{array}{cccc}
  1 & F & A & 3 \\
  \\end{array}
  \] (hex)
Hex to Binary Conversion

- Write down the 4-bit binary code for each hex digit
- Example:

  \[
  \begin{align*}
    0 & \rightarrow 0000 \\
    3 & \rightarrow 0011 \\
    9 & \rightarrow 1001 \\
    c & \rightarrow 1100 \\
    8 & \rightarrow 1000 \\
    1 & \rightarrow 0001 \\
    9 & \rightarrow 1001 \\
    a & \rightarrow 1010 \\
    2 & \rightarrow 0010 \\
    b & \rightarrow 1011 \\
    3 & \rightarrow 0011 \\
    c & \rightarrow 1100 \\
    4 & \rightarrow 0100 \\
    d & \rightarrow 1101 \\
    5 & \rightarrow 0101 \\
    e & \rightarrow 1110 \\
    6 & \rightarrow 0110 \\
    f & \rightarrow 1111
  \end{align*}
  \]
# Conversion Table

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Octal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>10</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>11</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>12</td>
<td>1010</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>13</td>
<td>1011</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>14</td>
<td>1100</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>15</td>
<td>1101</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>16</td>
<td>1110</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>17</td>
<td>1111</td>
</tr>
</tbody>
</table>
More Conversions

- Hex → Octal
  - Do it in 2 steps
  - Hex → binary → octal
- Decimal → Hex
  - Do it in 2 steps
  - Decimal → binary → hex
- So why use hex and octal and not just binary and decimal?
Largest Number

- What is the largest number that we can represent in $n$ digits...
  - In base 10? $99 \cdot 10^{n-1} + 10^n - 1$
  - In base 2? $2^n - 1$
  - In octal? $8^n - 1$
  - In hex? $16^n - 1$
  - In base $b$? $b^n - 1$

- How many different numbers can we represent with $n$ digits in base $b$?
Data Representation

Using binary numbers to represent information
Data Representation

- Goal: Store numbers, characters, sets, database records in the computer.
- What we have: Circuit that stores 2 voltages, one for logic 0 (0 volts) and one for logic 1 (ex: 3.3 volts).
  - DRAM – uses a single capacitor to store and a transistor to select.
  - SRAM – typically uses 6 transistors.
Definition: A unit of information. It is the amount of information needed to specify one of two equally likely choices.

- Example: Flipping a coin has 2 possible outcomes, heads or tails. The amount of info needed to specify the outcome is 1 bit.
Storing Information

<table>
<thead>
<tr>
<th>Value Representation</th>
<th>Value Representation</th>
<th>Value Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>H 0</td>
<td>False 0</td>
<td>1e-4 0</td>
</tr>
<tr>
<td>T 1</td>
<td>True 1</td>
<td>5 1</td>
</tr>
</tbody>
</table>

- Use more bits for more items
- Three bits can represent 8 things: 000, 001, ..., 111
- $N$ bits can represent $2^N$ things

<table>
<thead>
<tr>
<th>N bits</th>
<th>Can represent</th>
<th>Which is approximately</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>65,536</td>
<td>65 thousand (64K where K=1024)</td>
</tr>
<tr>
<td>32</td>
<td>4,294,967,296</td>
<td>4 billion</td>
</tr>
<tr>
<td>64</td>
<td>$1.8446... \times 10^{19}$</td>
<td>20 billion billion</td>
</tr>
</tbody>
</table>
# Storing Information

Most computers today use:

<table>
<thead>
<tr>
<th>Type</th>
<th># of bits</th>
<th>Name for storage unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>8-16</td>
<td>byte (ASCII) – 16b Unicode (Java)</td>
</tr>
<tr>
<td>Integers</td>
<td>32-64</td>
<td>word (sometimes 8 or 16 bits)</td>
</tr>
<tr>
<td>Reals</td>
<td>32-64</td>
<td>word or double word</td>
</tr>
</tbody>
</table>

\( \geq 8 \)
Character Representation

Memory location for a character usually contains 8 bits:
- 00000000 to 11111111 (binary)
- 0x00 to 0xff (hexadecimal)

Which characters?
- A, B, C, ..., Z, a, b, c, ..., z, 0, 1, 2, ..., 9
- Punctuation (,,:{...)
- Special (\n \O ...)

Which bit patterns for which characters?
- Want a standard!!!
- Want a standard to help sort strings of characters.
Character Representation

- **ASCII (American Standard Code for Information Interchange)**
- Defines what character is represented by each sequence of bits.
- **Examples:**
  - `0100 0001` is `0x41` (hex) or 65 (decimal). It represents “A”.
  - `0100 0010` is `0x42` (hex) or 66 (decimal). It represents “B”.
- Different bit patterns are used for each different character that needs to be represented.
ASCII Properties

ASCII has some nice properties.
• If the bit patterns are compared, (pretending they represent integers), then
  “A” < “B”
  65 < 66
• This is good because it helps with sorting things into alphabetical order.
• But...:
  • ‘a’ (61 hex) is different than ‘A’ (41 hex)
  • ‘8’ (38 hex) is different than the integer 8
  • ‘0’ is 30 (hex) or 48 (decimal)
  • ‘9’ is 39 (hex) or 57 (decimal)
# 7-bit ASCII Table

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hx Oct</th>
<th>Char</th>
<th>Dec</th>
<th>Hx Oct</th>
<th>Html</th>
<th>Char</th>
<th>Dec</th>
<th>Hx Oct</th>
<th>Html</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>NUL (null)</td>
<td>32</td>
<td>2040</td>
<td>32</td>
<td>Space</td>
<td>64</td>
<td>40100</td>
<td>64</td>
<td>96</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>SOH (start of heading)</td>
<td>65</td>
<td>4101</td>
<td>65</td>
<td>A</td>
<td>97</td>
<td>1410</td>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>0002</td>
<td>STX (start of text)</td>
<td>66</td>
<td>42102</td>
<td>66</td>
<td>B</td>
<td>98</td>
<td>12402</td>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>0003</td>
<td>ETX (end of text)</td>
<td>67</td>
<td>43103</td>
<td>67</td>
<td>C</td>
<td>99</td>
<td>13403</td>
<td>99</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>0004</td>
<td>EOT (end of transmission)</td>
<td>68</td>
<td>44104</td>
<td>68</td>
<td>D</td>
<td>100</td>
<td>14404</td>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>0005</td>
<td>ENQ (enquiry)</td>
<td>69</td>
<td>45105</td>
<td>69</td>
<td>E</td>
<td>101</td>
<td>14505</td>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>0006</td>
<td>ACK (acknowledge)</td>
<td>70</td>
<td>46106</td>
<td>70</td>
<td>F</td>
<td>102</td>
<td>14606</td>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>0007</td>
<td>BEL (bell)</td>
<td>71</td>
<td>47107</td>
<td>71</td>
<td>G</td>
<td>103</td>
<td>14707</td>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>0100</td>
<td>BS (backspace)</td>
<td>72</td>
<td>48110</td>
<td>72</td>
<td>H</td>
<td>104</td>
<td>150104</td>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>0110</td>
<td>TAB (horizontal tab)</td>
<td>73</td>
<td>49111</td>
<td>73</td>
<td>I</td>
<td>105</td>
<td>151105</td>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>10</td>
<td>0120</td>
<td>LF (NL line feed, new line)</td>
<td>74</td>
<td>4A112</td>
<td>74</td>
<td>J</td>
<td>106</td>
<td>152106</td>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>11</td>
<td>0130</td>
<td>VT (vertical tab)</td>
<td>75</td>
<td>4B113</td>
<td>75</td>
<td>K</td>
<td>107</td>
<td>153107</td>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>0140</td>
<td>FF (NP form feed, new page)</td>
<td>76</td>
<td>4C114</td>
<td>76</td>
<td>L</td>
<td>108</td>
<td>154108</td>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>0150</td>
<td>CR (carriage return)</td>
<td>77</td>
<td>4D115</td>
<td>77</td>
<td>M</td>
<td>109</td>
<td>155109</td>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>0160</td>
<td>SO (shift out)</td>
<td>78</td>
<td>4E116</td>
<td>78</td>
<td>N</td>
<td>110</td>
<td>156110</td>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>0170</td>
<td>SI (shift in)</td>
<td>79</td>
<td>4F117</td>
<td>79</td>
<td>O</td>
<td>111</td>
<td>157111</td>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>16</td>
<td>1020</td>
<td>DLE (data link escape)</td>
<td>80</td>
<td>50120</td>
<td>80</td>
<td>P</td>
<td>112</td>
<td>160112</td>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>1121</td>
<td>DC1 (device control 1)</td>
<td>81</td>
<td>51121</td>
<td>81</td>
<td>Q</td>
<td>113</td>
<td>161113</td>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>1222</td>
<td>DC2 (device control 2)</td>
<td>82</td>
<td>52122</td>
<td>82</td>
<td>R</td>
<td>114</td>
<td>162114</td>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>1323</td>
<td>DC3 (device control 3)</td>
<td>83</td>
<td>53123</td>
<td>83</td>
<td>S</td>
<td>115</td>
<td>163115</td>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>1424</td>
<td>DC4 (device control 4)</td>
<td>84</td>
<td>54124</td>
<td>84</td>
<td>T</td>
<td>116</td>
<td>164116</td>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>1525</td>
<td>NAK (negative acknowledge)</td>
<td>85</td>
<td>55125</td>
<td>85</td>
<td>U</td>
<td>117</td>
<td>165117</td>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>1626</td>
<td>SYN (synchronous idle)</td>
<td>86</td>
<td>56126</td>
<td>86</td>
<td>V</td>
<td>118</td>
<td>166118</td>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>1727</td>
<td>ETB (end of trans. block)</td>
<td>87</td>
<td>57127</td>
<td>87</td>
<td>W</td>
<td>119</td>
<td>167119</td>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>24</td>
<td>1830</td>
<td>CAN (cancel)</td>
<td>88</td>
<td>58130</td>
<td>88</td>
<td>X</td>
<td>120</td>
<td>170120</td>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>1931</td>
<td>EM (end of medium)</td>
<td>89</td>
<td>59131</td>
<td>89</td>
<td>Y</td>
<td>121</td>
<td>171121</td>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>0320</td>
<td>SUB (substitute)</td>
<td>90</td>
<td>5A132</td>
<td>90</td>
<td>Z</td>
<td>122</td>
<td>172122</td>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>0330</td>
<td>ESC (escape)</td>
<td>91</td>
<td>5B133</td>
<td>91</td>
<td>[</td>
<td>123</td>
<td>173123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>0340</td>
<td>FS (file separator)</td>
<td>92</td>
<td>5C134</td>
<td>92</td>
<td>\</td>
<td>124</td>
<td>174124</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>0350</td>
<td>GS (group separator)</td>
<td>93</td>
<td>5D135</td>
<td>93</td>
<td>]</td>
<td>125</td>
<td>175125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>0360</td>
<td>RS (record separator)</td>
<td>94</td>
<td>5E136</td>
<td>94</td>
<td>^</td>
<td>126</td>
<td>176126</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>0370</td>
<td>US (unit separator)</td>
<td>95</td>
<td>5F137</td>
<td>95</td>
<td>_</td>
<td>127</td>
<td>177127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: www.asciitable.com
Integer Representation

Assume our representation has a fixed number of bits $n$ (e.g. 32 bits).

- Which 4 billion integers do we want?
  - There are an infinite number of integers less than zero and an infinite number greater than zero.

- What bit patterns should we select to represent each integer AND where the representation:
  - Does not affect the result of calculation
  - Does dramatically affect the ease of calculation

- Convert to/from human-readable representation as needed.
Integer Representation

Usual answers:

1. Represent 0 and consecutive positive integers
   - Unsigned integers
2. Represent positive and negative integers
   - Signed magnitude
   - One’s complement
   - Two’s complement
   - Biased

Unsigned and two’s complement the most common
Unsigned Integers

• Integer represented is binary value of bits:
  
  0000 -> 0, 0001 -> 1, 0010 -> 2, ...

• Encodes only positive values and zero
• Range: 0 to $2^n - 1$, for n bits
Unsigned Integers

If we have 4 bit numbers:

To find range make $n = 4$. Thus $2^4 - 1$ is 15
Thus the values possible are 0 to 15
$[0:15] = 16$ different numbers

7 would be 0111
17 not represent able
-3 not represent able

For 32 bits:

Range is 0 to $2^{32} - 1 = [0: 4,294,967,295]$
Which is 4,294,967,296 different numbers
Signed Magnitude Integers

- A human readable way of getting both positive and negative integers.
- Not well suited to hardware implementation.
- But used with floating point.
Signed Magnitude Integers

Representation:

- Use 1 bit of integer to represent the sign of the integer
  - Sign bit is msb: 0 is “+”, 1 is “−”
- Rest of the integer is a magnitude, with same encoding as unsigned integers.
- To get the additive inverse of a number, just flip (invert, complement) the sign bit.
- Range: $-(2^{n-1} - 1)$ to $2^{n-1} - 1$
Signed Magnitude - Example

If 4 bits then range is:
\[-2^3 + 1 \text{ to } 2^3 - 1\]
which is -7 to +7

Questions:
• 0101 is ?
• -3 is ?
• +12 is ?
• \([-7, \ldots, -1, 0, +1, \ldots, +7] = 7 + 1 + 7 = 15 < 16 = 2^4\]
  • Why?
  • What problems does this cause?
One’s Complement

- Historically important (in other words, not used today!!!)
- Early computers built by Semour Cray (while at CDC) were based on 1’s complement integers.
- Positive integers use the same representation as unsigned.
  - 0000 is 0
  - 0111 is 7, etc
- Negation is done by taking a bitwise complement of the positive representation.
  - Complement = Invert = Not = Flip = \{0 \rightarrow 1, 1 \rightarrow 0\}
  - A logical operation done on a single bit
- Top bit is sign bit
One’s Complement Representation

To get 1’s complement of \(-1\)
  • Take +1: 0001
  • Complement each bit: 1110
  • Don’t add or take away any bits.
Another example (4-bits):
  • 1100
  • This must be a negative number. To find out which, find the inverse!
  • 0011 is +3
  • 1100 in 1’s Complement must be?
Properties of 1’s complement:
  • Any negative number will have a 1 in the MSB
  • There are 2 representations for 0; 0000 and 1111
Two’s Complement

• Variation on 1’s complement that does not have 2 representations for 0.
• This makes the hardware that does arithmetic simpler and faster than the other representations.
• The negative values are all “slid” by one, eliminating the –0 case.
• How to get 2’s complement representation:
  • Positive: just as if unsigned binary
  • Negative:
    • Take the positive value
    • Take the 1’s complement of it
    • Add 1
Two’s Complement

Example, what is -5 in 4-bit 2SC?

1. What is 5? 0101
2. Invert all the bits: 1010 (basically find the 1SC)
3. Add one: 1010 + 1 = 1011 which is -5 in 2SC

To get the additive inverse of a 2’s complement integer

1. Take the 1’s complement
2. Add 1
Two’s Complement

Number of integers representable is $-2^{n-1}$ to $2^{n-1}-1$

So if 4 bits:

$$[-8, ..., -1, 0, +1, ..., +7] = 8 + 1 + 7 = 16 = 2^4$$

numbers

With 32 bits:

$$[-2^{31}, ..., -1, 0, +1, ..., (2^{31}-1)] = 2^{31} + 1 + (2^{31}-1) = 2^{32}$$

numbers

$$[-2147483648, ..., -1, 0, +1, ..., 2147483647] \sim \pm 2B$$
A Little Bit on Adding

Simple way of adding 1 in binary:

• Start at LSB, for each bit (working right to left)
  • While the bit is a 1, change it to a 0.
  • When a 0 is encountered, change it to a 1 and stop.
  • Can combine with bit inversion to form 2’s complement.

\[
\begin{array}{c}
100111 \\
+ 1 \\
\hline
101000
\end{array}
\]
A Little Bit on Adding

More generally, it's just like decimal!!

\[
\begin{align*}
0 + 0 &= 0 \\
1 + 0 &= 1 \\
1 + 1 &= 2, \text{ which is 10 in binary, sum is 0, carry is 1.} \\
1 + 1 + 1 &= 3, \text{ sum is 1, carry is 1.}
\end{align*}
\]

\[
\begin{array}{c}
x \quad 0011 \\
+ y \quad +0001 \\
\hline 
\text{sum} \quad 0100
\end{array}
\]
## A Little Bit on Adding

### Truth Table for a full Adder

<table>
<thead>
<tr>
<th>Carry in</th>
<th>A</th>
<th>B</th>
<th>Sum</th>
<th>Carry out</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>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Biased Representation

An integer representation that skews the bit patterns so as to look just like unsigned but actually represent negative numbers.

Example: 4-bit, with BIAS of $2^3$ (or called Excess 8)

- True value to be represented: 3
- Add in the bias: +8
- Unsigned value: 11

The bit pattern of 3 in biased-8 representation will be 1011
Biased Representation

Suppose we were given a biased-8 representation, 0110, to find what the number represented was:

- Unsigned 0110 represents 6
- Subtract out the bias -8
- True value represented -2

Operations on the biased numbers can be unsigned arithmetic but represent both positive and negative values.

How do you add two biased numbers? Subtract?
Exercises, what are these in decimal?

\[ 25_{10} \text{ in excess 100 is:} \]
\[ 52_{10, \text{excess 127}} \text{ is:} \]
\[ 101101_{2, \text{excess 31}} \text{ is:} \]
\[ 1101_{2, \text{excess 31}} \text{ is:} \]
Biased Representation

Where is the sign “bit” in excess notation? Bias notation used in floating-point exponents.

Choosing a bias:
To get an ~ equal distribution of values above and below 0, the bias is usually $2^{n-1}$ or $2^{n-1} - 1$.

Range of bias numbers?
Depends on bias, but contains $2^n$ different numbers.
Sign Extension

How to change a number with a smaller number of bits into the same number (same representation) with a larger number of bits?

This must be done frequently by arithmetic units.
Sign Extension - unsigned

Unsigned representation:

Copy the original integer into the LSBs, and put 0’s elsewhere.

Thus for 5 bits to 8 bits:

xxxxx -> 000xxxxxx
Sign Extension – signed magnitude

Signed magnitude:

Copy the original integer’s magnitude into the LSBs & put the original sign into the MSB, put 0’s elsewhere.

Thus for 6 bits to 8 bits

sxxxxx -> s00xxxxx
Sign Extension – 1SC and 2SC

1’s and 2’s complement:

1. Copy the original n-1 bits into the LSBs
2. Take the MSB of the original and copy it elsewhere

Thus for 6 bits to 8 bits:

sxxxxx $\rightarrow$ sssxxxxx
Sign Extension – 2SC

To make this less clear... 😊

• In 2’s complement, the MSB (sign bit) is the \(-2^{n-1}\) place.
• It says “subtract \(2^{n-1}\) from \(b_{n-2}\ldots b_0\)”.
• Sign extending one bit
  • Adds a \(-2^n\) place
  • Changes the old sign bit to a \(+2^{n-1}\) place
• \(-2^n + 2^{n-1} = -2^{n-1}\), so the number stays the same
Sign Extension

What is -12 in 8-bit 2’s complement form
Arithmetic and Logical Operations

(Ch 2 & 3)
Logical Operations

Operate on raw bits with 1 = true and 0 = false

| In1 | In2 | & | | | ~(&) | ~(|) | ^ | ~(^) |
|-----|-----|---|---|---|-------|-------|---|-------|
| 0   | 0   | 0 | 0 | 1 | 1     | 0     | 1 |       |
| 0   | 1   | 0 | 1 | 1 | 0     | 1     | 0 |       |
| 0   | 1   | 0 | 1 | 1 | 0     | 1     | 0 |       |
| 1   | 0   | 0 | 1 | 1 | 0     | 1     | 0 |       |
| 1   | 1   | 1 | 1 | 0 | 0     | 0     | 0 | 1     |

The symbols are C bitwise operators
"bit-wise" logical operations are done in parallel for corresponding bits

Example:
\[
X = \begin{array}{c}
0 \\
0 \\
1 \\
1 \\
\end{array}
\quad X \text{ AND } Y = \begin{array}{c}
0 \\
0 \\
1 \\
0 \\
\end{array}
\quad Y = \begin{array}{c}
0 \\
0 \\
1 \\
0 \\
\end{array}
\]

So how do an OR? How about an XOR?
Logical Operations

Shifts and Rotates

Logical right
- Move bits to the right, same order
- Throw away the bit that pops off the LSB
- Introduce a 0 into the MSB
  
  \[ 00110101 \rightarrow 00011010 \]  
  (shift right by 1)

Logical left
- Move bits to the left, same order
- Throw away the bit that pops off the MSB
- Introduce a 0 into the LSB
  
  \[ 00110101 \rightarrow 11010100 \]  
  (shift left by 2)
Arithmetic right shift

- Move bits to the right, same order
- Throw away the bit that pops off the LSB
- Reproduce the original MSB into the new MSB
- Alternatively, shift the bits, and then do sign extension

\[
\begin{align*}
00110101 & \rightarrow 00011010 \quad \text{(right by 1)} \\
1100 & \rightarrow 1111 \quad \text{(right by 2)}
\end{align*}
\]

Arithmetic left shift

- Move bits to the left, same order
- Throw away the bit that pops off the MSB
- Introduce a 0 into the LSB

\[
\begin{align*}
00110101 & \rightarrow 01101010 \quad \text{(left by 1)}
\end{align*}
\]
Logical Operations: Shifts and Rotates

Rotate left

- Move bits to the left, same order
- Put the bit(s) that pop off the MSB into the LSB
- No bits are thrown away or lost
  
  \[
  \begin{align*}
  00110101 & \rightarrow 01101010 \quad \text{(rotate by 1)} \\
  1100 & \rightarrow 1001 \quad \text{(rotate by 1)}
  \end{align*}
  \]

Rotate right

- Move bits to the right, same order
- Put the bit that pops off the LSB into the MSB
- No bits are thrown away or lost
  
  \[
  \begin{align*}
  00110101 & \rightarrow 10011010 \quad \text{(rotate by 1)} \\
  1101 & \rightarrow 0111 \quad \text{(rotate by 2)}
  \end{align*}
  \]
Binary Addition

\[
\begin{array}{c}
x \quad 0011 \\
+ y \quad +0001 \\
\text{sum} \quad 0100
\end{array}
\]

Or in tabular form...

<table>
<thead>
<tr>
<th>Carry In</th>
<th>A</th>
<th>B</th>
<th>Sum</th>
<th>Carry Out</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>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Binary Addition

And as a full adder...

4-bit Ripple-Carry adder:
- Carry values propagate from bit to bit
- Like pencil-and-paper addition
- Time proportional to number of bits
- “Lookahead-carry” can propagate carry proportional to \( \log(n) \) with more space.
Addition: unsigned

Just like the simple addition given earlier:

Examples:

\[
\begin{array}{c}
100001 \quad (33) \\
+011101 \quad (29) \\
\hline \\
(62)
\end{array}
\]

\[
\begin{array}{c}
00001010 \quad (10) \\
+00001110 \quad (14) \\
\hline \\
(24)
\end{array}
\]

(we are ignoring overflow for now)
Addition: 2’s complement

- Just like unsigned addition
- Assume 6-bit and observe:

\[
\begin{array}{cccccc}
000011 & (3) & 101000 & (-24) & 111111 & (-1) \\
+111100 & (-4) & +010000 & (16) & +001000 & (8) \\
\hline
\multicolumn{2}{c}{(-1)} & \multicolumn{2}{c}{(-8)} & \multicolumn{2}{c}{(7)}
\end{array}
\]

- Ignore carry-outs (overflow)
- Sign bit is in the \(2^{n-1}\) bit position
- What does this mean for adding different signs?
Addition: 2’s complement

More examples: Convert to 2SC and do the addition

-20 + 15

5 + 12

-12 + -25
Addition: sign magnitude

- Add magnitudes only, just like unsigned addition
- Do not carry into the sign bit
- If a carry out of the MSB of magnitude then overflowed
- Add only integers of like sign ("+ to +" OR "- to -")
- Sign of the result is same as sign of the addends
Addition: sign magnitude

Examples:

\[
\begin{align*}
0 & \quad 0101 \quad (5) \\
+ & \quad 0 \quad 0011 \quad (3) \\
\hline
\quad & \quad 0110 \quad (8)
\end{align*}
\]

\[
\begin{align*}
1 & \quad 1010 \quad (-10) \\
+ & \quad 1 \quad 0011 \quad (-3) \\
\hline
\quad & \quad 1110 \quad (-13)
\end{align*}
\]

\[
\begin{align*}
0 & \quad 01011 \quad (11) \\
+ & \quad 1 \quad 01110 \quad (-14) \\
\hline
\quad & \quad 10001 \quad (-3)
\end{align*}
\]

Not addition! This is subtraction
Subtraction

General rules:

1 - 1 = 0
0 - 0 = 0
1 - 0 = 1
10 - 1 = 1
0 - 1 = need to borrow!

• Or replace $(x - y)$ with $x + (-y)$
• Can replace subtraction with additive inverse and addition
Subtraction: 2’s complement

Don’t. Just use addition:

\[ x - y \rightarrow x + (-y) \]

Example:

\[
\begin{array}{c}
10110 \quad (-10) \\
- \quad 00011 \quad (3)
\end{array}
\]

\[
\begin{array}{c}
10110 \quad (-10) \\
+ \quad 11101 \quad (-3)
\end{array}
\]

\[
\begin{array}{c}
\underline{10011} \quad (-13)
\end{array}
\]
Can also flip bits of bottom # and add an LSB carry in, so for -10 - 3 we get:

\[ \begin{array}{c}
1 \\
10110 \\
+ 11100 \\
10011 \\
\end{array} \]

“add 1”

“flip bits of bottom number”

(throw away carry out)

Addition and subtraction are simple in 2’s complement, just need an adder and inverters.
Subtraction: unsigned

For n-bits use the 2’s complement method and overflow if negative

\[
\begin{align*}
11100 \ (+28) \\
- \ 10110 \ (+22)
\end{align*}
\]

Becomes

\[
\begin{align*}
1 \\
11100 \\
+01001 \\
\hline
00110
\end{align*}
\]

Only take 5 bits of result
Subtraction: sign magnitude

• If signs are different, then change the problem to addition
• If the signs are the same then do subtraction
  • compare magnitudes
  • subtract smaller from larger
  • if the order was switched, then switch the sign of the result
Subtraction: sign magnitude

For example:

\[
\begin{array}{c}
0 \, 00111 \ (7) \ \text{becomes} \ \ 0 \, 11000 \ (24) \\
- \ 0 \, 11000 \ (24) \\
\hline
1 \, 10001 \ (-17)
\end{array}
\]

Need to switch sign since the order of the subtraction was reversed

\[
\begin{array}{c}
1 \, 11000 \ (-24) \\
- \ 1 \, 00010 \ (-2) \\
\hline
1 \, 10110 \ (-22)
\end{array}
\]

Evaluation of the sign bit is not part of the arithmetic, it is determined by comparing magnitudes
Overflow in Addition

Unsigned: When there is a carry out of the MSB

\[
\begin{align*}
1000 & \quad (8) \\
+1001 & \quad (9) \\
\underline{+1001} & \quad (1) \\
10001 & \quad (1)
\end{align*}
\]
Signed magnitude: When there is a carry out of the MSB of the magnitude

```
  1 1000  (-8)
+1 1001  (-9)
  110001  (-1)
```

carry out from MSB of magnitude
2's complement: When the signs of the addends are the same, but the sign of the result is different

\[
\begin{array}{c}
0011 \ (3) \\
+ \ 0110 \ (6) \\
\hline
1001 \ (-7)
\end{array}
\]

Adding 2 numbers of opposite signs never overflows

Why?
Overflow in Subtraction

Unsigned: if result would be negative

Signed magnitude: never happens when actually doing subtraction

2's complement: never do subtraction, so use the addition rule on the addition operation done.
Unsigned Binary Multiplication

The *multiplicand* is multiplied by the *multiplier* to produce the *product*, the sum of *partial products*

\[
\text{multiplicand} \times \text{multiplier} = \text{product}
\]

\[
\begin{array}{c}
0011 \ (+3) \\
x 0110 \ (+6) \\
\hline \\
0000 \\
0011 \\
0011 \\
0000 \\
\hline \\
0010010 \ (+18)
\end{array}
\]

- Longhand, it looks just like decimal
- Result can require twice as many bits as the larger multiplicand (why?)
2's Complement Multiplication

- If negative multiplicand, just sign-extend it.
- If negative multiplier, take 2SC of both multiplicand and multiplier (-7 x -3 = 7 x 3, and 7 x -3 = -7 x 3), also known as the additive inverse.

0011 (3)  \times 1011 (-5)
\underline{1101 (-3)  \times 0101 (+5)}
\underline{11111101}
00000000
111101
\underline{11110001} (-15)

Only need 8 bits for result
Division

Only required to know for unsigned binary

Just like you do with decimal long hand

\[ \frac{14}{2} = 10 \overline{1110} \]