Number Representation

Prof. Harry H. Porter
Portland State University
Decimal Number Representation

Example:

4037

= 4000 + 30 + 7
= ... + 0 \cdot 10000 + 4 \cdot 1000 + 0 \cdot 100 + 3 \cdot 10 + 7 \cdot 1
= ... + 0 \cdot 10^4 + 4 \cdot 10^3 + 0 \cdot 10^2 + 3 \cdot 10^1 + 7 \cdot 10^0

Base 10:

... + X \cdot 10^4 + X \cdot 10^3 + X \cdot 10^2 + X \cdot 10^1 + X \cdot 10^0

Set of numerals (the “digits”):

\{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}
Hexadecimal Number Representation

Base 16:

\[ \ldots + X \cdot 16^4 + X \cdot 16^3 + X \cdot 16^2 + X \cdot 16^1 + X \cdot 16^0 \]
\[ \ldots + X \cdot 65536 + X \cdot 4096 + X \cdot 256 + X \cdot 16 + X \cdot 1 \]

Set of numerals:

\{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F \}

Example:

3A0F

\[ = \ldots + 0 \cdot 16^4 + 3 \cdot 16^3 + A \cdot 16^2 + 0 \cdot 16^1 + F \cdot 16^0 \]
\[ = \ldots + 0 \cdot 65536 + 3 \cdot 4096 + A \cdot 256 + 0 \cdot 16 + F \cdot 1 \]
\[ = \ldots + 0 \cdot 65536 + 3 \cdot 4096 + 10 \cdot 256 + 0 \cdot 16 + 15 \cdot 1 \]
\[ = 12,288 + 2,560 + 15 = 14,863 \text{ (in decimal)} \]
<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
</tr>
</tbody>
</table>
Hexadecimal Number Representation

Base 16:

... + $X \cdot 16^4$ + $X \cdot 16^3$ + $X \cdot 16^2$ + $X \cdot 16^1$ + $X \cdot 16^0$

... + $X \cdot 65536$ + $X \cdot 4096$ + $X \cdot 256$ + $X \cdot 16$ + $X \cdot 1$

Set of numerals:

$\{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F \}$

Example:

2CB

= ... + $0 \cdot 16^4$ + $0 \cdot 16^3$ + $2 \cdot 16^2$ + $C \cdot 16^1$ + $B \cdot 16^0$
Hexadecimal Number Representation

Base 16:

\[ \ldots + X \cdot 16^4 + X \cdot 16^3 + X \cdot 16^2 + X \cdot 16^1 + X \cdot 16^0 \]

\[ \ldots + X \cdot 65536 + X \cdot 4096 + X \cdot 256 + X \cdot 16 + X \cdot 1 \]

Set of numerals:

\{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F \}

Example:

2CB

\[ = \ldots + 0 \cdot 16^4 + 0 \cdot 16^3 + 2 \cdot 16^2 + C \cdot 16^1 + B \cdot 16^0 \]

\[ = \ldots + 0 \cdot 65536 + 0 \cdot 4096 + 2 \cdot 256 + C \cdot 16 + B \cdot 1 \]

\[ = \ldots + 0 \cdot 65536 + 0 \cdot 4096 + 2 \cdot 256 + 12 \cdot 16 + 11 \cdot 1 \]

\[ = 512 + 192 + 11 \quad = \quad 715 \text{ (in decimal)} \]
Binary Number Representation

Base 2:

\[ ... + X \cdot 2^5 + X \cdot 2^4 + X \cdot 2^3 + X \cdot 2^2 + X \cdot 2^1 + X \cdot 2^0 \]
\[ ... + X \cdot 32 + X \cdot 16 + X \cdot 8 + X \cdot 4 + X \cdot 2 + X \cdot 1 \]

Set of numerals:
\[ \{ 0, 1 \} \]

**Example:**

110101

\[ = \ldots + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \]
\[ = \ldots + 1 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 \]
\[ = 32 + 16 + 4 + 1 \]
\[ = 53 \text{ (in decimal)} \]
### Decimal Number Representation

<table>
<thead>
<tr>
<th></th>
<th>10,000</th>
<th>1,000</th>
<th>100</th>
<th>10</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

### Binary Number Representation

<table>
<thead>
<tr>
<th></th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Hex Number Representation

<table>
<thead>
<tr>
<th></th>
<th>16,777,216</th>
<th>1,048,576</th>
<th>65,536</th>
<th>4,096</th>
<th>256</th>
<th>16</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>E</td>
<td>7</td>
<td>D</td>
<td>F</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
“C” Notation

**Decimal**

48293

**Binary**

(not standard)

**Hex**

0x4E7DF20
0x4e7df20
Practice

Convert the following

10110111₂ to Base 10 = 
11011001₂ to Base 16 = 
0x2ae to Base 2 = 
0x13e to Base 10 = 
150₁₀ to Base 2 =

301₁₀ to Base 16 =

### Binary Number Representation

<table>
<thead>
<tr>
<th>128</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
</table>

### Hex Number Representation

| 16,777,216 | 1,048,576 | 65,536 | 4,096 | 256 | 16 | 1 |
Practice

Convert the following

10110111₂ to Base 10 = 128 + 32 + 16 + 4 + 2 + 1 = 183
11011001₂ to Base 16 = 0xd9
0x2ae to Base 2 = 0010 1010 1110₂
0x13e to Base 10 = 1·256 + 3·16 + 14 = 318₁₀
150₁₀ to Base 2 =
1·128 + 0·64 + 0·32 + 1·16 + 0·8 + 1·4 + 1·2 + 0·1 = 010010110₂
301₁₀ to Base 16 = 1·256 + 3·16 + 13 = 0x12d

Binary Number Representation

<table>
<thead>
<tr>
<th>128</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
</table>

Hex Number Representation

<table>
<thead>
<tr>
<th>16,777,216</th>
<th>1,048,576</th>
<th>65,536</th>
<th>4,096</th>
<th>256</th>
<th>16</th>
<th>1</th>
</tr>
</thead>
</table>
One-to-one correspondence between hex and binary;

\[
\begin{array}{cccc}
3 & A & 0 & F \\
0011 & 1010 & 0000 & 1111 \\
\end{array}
\]

**Byte (8 bits)**
- Hex: 3A
- Binary: 0011 1010

**Halfword (16 bits)**
- Hex: 3A0F
- Binary: 0011 1010 0000 1111

**Word (32 bits)**
- Hex: 3AOF 12D8
- Binary: 0011 1010 0000 1111 0001 0010 1101 1000

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

Bad match with byte alignment

```
1 1 0 1 0 1 0 0 1 0 1 1 0 0 0 1
byte

1 1 0 1 0 1 0 0 1 0 1 1 0 0 0 1
byte
```

The numbers get too long.

Word (32 bits)

Octal: 12305570426
Hex: 3A0F 12D8

Every octal looks like a decimal number (and often they get confused).

263_8 = 179_10
263_10 = 263_10
263_16 = 611_10

C Notation for octals (leading zero is significant!)

0263
# Data Representations in C

(Size in bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>IA32</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
</tr>
</tbody>
</table>

(…or any other pointer)

- short = short int
- long = long int
- long long = long long int
# Data Representations in C

(Size in bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

*(...or any other pointer)*

short = short int
long = long int
long long = long long int
# Unsigned Number Representation

**Example: 8-bits**

Always non-negative

- $0, 1, 2, \ldots, 255$
- $0, 1, 2, \ldots, 2^8 - 1$

<table>
<thead>
<tr>
<th>Value (in decimal)</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 0000</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0000 0001</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>0000 0010</td>
<td>02</td>
</tr>
<tr>
<td>3</td>
<td>0000 0011</td>
<td>03</td>
</tr>
<tr>
<td>4</td>
<td>0000 0100</td>
<td>04</td>
</tr>
<tr>
<td>5</td>
<td>0000 0101</td>
<td>05</td>
</tr>
<tr>
<td>6</td>
<td>0000 0110</td>
<td>06</td>
</tr>
<tr>
<td>7</td>
<td>0000 0111</td>
<td>07</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>252</td>
<td>1111 1100</td>
<td>FC</td>
</tr>
<tr>
<td>253</td>
<td>1111 1101</td>
<td>FD</td>
</tr>
<tr>
<td>254</td>
<td>1111 1110</td>
<td>FE</td>
</tr>
<tr>
<td>255</td>
<td>1111 1111</td>
<td>FF</td>
</tr>
</tbody>
</table>
**Unsigned Number Representation**

**Example: 32-bits**

Always non-negative

0, 1, 2, ... 4,294,967,295

0, 1, 2, ... $2^{32} - 1$

<table>
<thead>
<tr>
<th>Value (in decimal)</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1</td>
<td>0000 0000 0000 0000 0000 0000 0000 0001</td>
<td>0000 0001</td>
</tr>
<tr>
<td>2</td>
<td>0000 0000 0000 0000 0000 0000 0000 0010</td>
<td>0000 0002</td>
</tr>
<tr>
<td>3</td>
<td>0000 0000 0000 0000 0000 0000 0000 0011</td>
<td>0000 0003</td>
</tr>
<tr>
<td>4</td>
<td>0000 0000 0000 0000 0000 0000 0000 0100</td>
<td>0000 0004</td>
</tr>
<tr>
<td>5</td>
<td>0000 0000 0000 0000 0000 0000 0000 0101</td>
<td>0000 0005</td>
</tr>
<tr>
<td>6</td>
<td>0000 0000 0000 0000 0000 0000 0000 0110</td>
<td>0000 0006</td>
</tr>
<tr>
<td>7</td>
<td>0000 0000 0000 0000 0000 0000 0000 0111</td>
<td>0000 0007</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>4,294,967,292</td>
<td>1111 1111 1111 1111 1111 1111 1111 1100</td>
<td>FFFF FFFC</td>
</tr>
<tr>
<td>4,294,967,293</td>
<td>1111 1111 1111 1111 1111 1111 1111 1101</td>
<td>FFFF FFFD</td>
</tr>
<tr>
<td>4,294,967,294</td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
<td>FFFF FFFE</td>
</tr>
<tr>
<td>4,294,967,295</td>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>FFFF FFFF</td>
</tr>
</tbody>
</table>
Unsigned Number Representation

Largest Number Representable

**Byte (8-bits)**
\[2^8 - 1\]
\[= 255\]
\[= \text{FF (in hex)}\]

**Halfword (16-bits)**
\[2^{16} - 1\]
\[= 65,535\]
\[= 64K - 1\]
\[= \text{FFFF (in hex)}\]

**Word (32-bits)**
\[2^{32} - 1\]
\[= 4,294,967,295\]
\[= 4G - 1\]
\[= \text{FFFF FFFF (in hex)}\]
### Signed Number Representation

#### Example: 8-bits

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Unsigned Value</th>
<th>Signed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000 0001</td>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0000 0010</td>
<td>02</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0111 1101</td>
<td>7D</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>0111 1110</td>
<td>7E</td>
<td>126</td>
<td>126</td>
</tr>
<tr>
<td>0111 1111</td>
<td>7F</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>1000 0000</td>
<td>80</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>1000 0001</td>
<td>81</td>
<td>129</td>
<td>129</td>
</tr>
<tr>
<td>1000 0010</td>
<td>82</td>
<td>130</td>
<td>130</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111 1101</td>
<td>FD</td>
<td>253</td>
<td>253</td>
</tr>
<tr>
<td>1111 1110</td>
<td>FE</td>
<td>254</td>
<td>254</td>
</tr>
<tr>
<td>1111 1111</td>
<td>FF</td>
<td>255</td>
<td>255</td>
</tr>
</tbody>
</table>
Signed Number Representation

Example: 8-bits

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Unsigned Value</th>
<th>Signed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000 0001</td>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0000 0010</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0111 1101</td>
<td>7D</td>
<td>125</td>
<td>125 2^7–3</td>
</tr>
<tr>
<td>0111 1110</td>
<td>7E</td>
<td>126</td>
<td>126 2^7–2</td>
</tr>
<tr>
<td>0111 1111</td>
<td>7F</td>
<td>127</td>
<td>127 2^7–1</td>
</tr>
<tr>
<td>1000 0000</td>
<td>80</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>1000 0001</td>
<td>81</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>1000 0010</td>
<td>82</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1111 1101</td>
<td>FD</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td>1111 1110</td>
<td>FE</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>1111 1111</td>
<td>FF</td>
<td>255</td>
<td></td>
</tr>
</tbody>
</table>
### Signed Number Representation

Example: 8-bits

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Unsigned Value</th>
<th>Signed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000 0001</td>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0000 0010</td>
<td>02</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0111 1101</td>
<td>7D</td>
<td>125</td>
<td>125 2^7-3</td>
</tr>
<tr>
<td>0111 1110</td>
<td>7E</td>
<td>126</td>
<td>126 2^7-2</td>
</tr>
<tr>
<td>0111 1111</td>
<td>7F</td>
<td>127</td>
<td>127 2^7-1</td>
</tr>
<tr>
<td>1000 0000</td>
<td>80</td>
<td>128</td>
<td>-128 -(2^7)</td>
</tr>
<tr>
<td>1000 0001</td>
<td>81</td>
<td>129</td>
<td>-127 -(2^7-1)</td>
</tr>
<tr>
<td>1000 0010</td>
<td>82</td>
<td>130</td>
<td>-126 -(2^7-2)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111 1101</td>
<td>FD</td>
<td>253</td>
<td>-3</td>
</tr>
<tr>
<td>1111 1110</td>
<td>FE</td>
<td>254</td>
<td>-2</td>
</tr>
<tr>
<td>1111 1111</td>
<td>FF</td>
<td>255</td>
<td>-1</td>
</tr>
</tbody>
</table>
### Signed Number Representation

**Example: 8-bits**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Unsigned Value</th>
<th>Signed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000 0001</td>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0000 0010</td>
<td>02</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0111 1101</td>
<td>7D</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>0111 1110</td>
<td>7E</td>
<td>126</td>
<td>126</td>
</tr>
<tr>
<td>0111 1111</td>
<td>7F</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>1000 0000</td>
<td>80</td>
<td>128</td>
<td>-128</td>
</tr>
<tr>
<td>1000 0001</td>
<td>81</td>
<td>129</td>
<td>-127</td>
</tr>
<tr>
<td>1000 0010</td>
<td>82</td>
<td>130</td>
<td>-126</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111 1101</td>
<td>FD</td>
<td>253</td>
<td>-3</td>
</tr>
<tr>
<td>1111 1110</td>
<td>FE</td>
<td>254</td>
<td>-2</td>
</tr>
<tr>
<td>1111 1111</td>
<td>FF</td>
<td>255</td>
<td>-1</td>
</tr>
</tbody>
</table>

**“Two’s complement” number representation**

Most significant bit
0 means $\geq$ zero (in hex: 0..7)
1 means < zero (in hex: 8..F)
### Signed Number Representation

**Example: 8-bits**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Unsigned Value</th>
<th>Signed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000 0001</td>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0000 0010</td>
<td>02</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0111 1101</td>
<td>7D</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>0111 1110</td>
<td>7E</td>
<td>126</td>
<td>126</td>
</tr>
<tr>
<td>0111 1111</td>
<td>7F</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>1000 0000</td>
<td>80</td>
<td>128</td>
<td>-128</td>
</tr>
<tr>
<td>1000 0001</td>
<td>81</td>
<td>129</td>
<td>-127</td>
</tr>
<tr>
<td>1000 0010</td>
<td>82</td>
<td>130</td>
<td>-126</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111 1101</td>
<td>FD</td>
<td>253</td>
<td>-3</td>
</tr>
<tr>
<td>1111 1110</td>
<td>FE</td>
<td>254</td>
<td>-2</td>
</tr>
<tr>
<td>1111 1111</td>
<td>FF</td>
<td>255</td>
<td>-1</td>
</tr>
</tbody>
</table>

"Two’s complement" number representation

**Most significant bit**
- 0 means ≥ zero (in hex: 0..7)
- 1 means < zero (in hex: 8..F)

Always one more negative number than positive numbers:

\[-128, \ldots, -1, 0, 1, \ldots, +127\]

\[2^7 = 128 \text{ values} + 2^7 = 128 \text{ values} = 2^8 = 256 \text{ values}\]
### Signed Number Representation

**Example: 32-bits**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Unsigned Value</th>
<th>Signed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000...0000</td>
<td>0000 0000</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0000...0001</td>
<td>0000 0001</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0000...0010</td>
<td>0000 0002</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0111...1101</td>
<td>7FFF FFFD</td>
<td>2,147,483,645</td>
<td></td>
</tr>
<tr>
<td>0111...1110</td>
<td>7FFF FFFE</td>
<td>2,147,483,646</td>
<td></td>
</tr>
<tr>
<td>0111...1111</td>
<td>7FFF FFFF</td>
<td>2,147,483,647</td>
<td></td>
</tr>
<tr>
<td>1000...0000</td>
<td>8000 0000</td>
<td>2,147,483,648</td>
<td></td>
</tr>
<tr>
<td>1000...0001</td>
<td>8000 0001</td>
<td>2,147,483,649</td>
<td></td>
</tr>
<tr>
<td>1000...0010</td>
<td>8000 0002</td>
<td>2,147,483,650</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111...1101</td>
<td>FFFF FFFD</td>
<td>4,294,967,294</td>
<td></td>
</tr>
<tr>
<td>1111...1110</td>
<td>FFFF FFFE</td>
<td>4,294,967,295</td>
<td></td>
</tr>
<tr>
<td>1111...1111</td>
<td>FFFF FFFF</td>
<td>4,294,967,296</td>
<td></td>
</tr>
</tbody>
</table>

*“Two’s complement” number representation*
## Signed Number Representation

**Example: 32-bits**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Unsigned Value</th>
<th>Signed Value</th>
<th>&quot;Two’s complement&quot; number representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000...0000</td>
<td>0000 0000</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0000...0001</td>
<td>0000 0001</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0000...0010</td>
<td>0000 0002</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0111...1101</td>
<td>7FFF FFFD</td>
<td>2,147,483,645</td>
<td>2,147,483,645</td>
<td>2&lt;sup&gt;31&lt;/sup&gt;–3</td>
</tr>
<tr>
<td>0111...1110</td>
<td>7FFF FFFE</td>
<td>2,147,483,646</td>
<td>2,147,483,646</td>
<td>2&lt;sup&gt;31&lt;/sup&gt;–2</td>
</tr>
<tr>
<td>0111...1111</td>
<td>7FFF FFFF</td>
<td>2,147,483,647</td>
<td>2,147,483,647</td>
<td>2&lt;sup&gt;31&lt;/sup&gt;–1</td>
</tr>
<tr>
<td>1000...0000</td>
<td>8000 0000</td>
<td>2,147,483,648</td>
<td>-2,147,483,648</td>
<td>-(2&lt;sup&gt;31&lt;/sup&gt;)</td>
</tr>
<tr>
<td>1000...0001</td>
<td>8000 0001</td>
<td>2,147,483,649</td>
<td>-2,147,483,647</td>
<td>-(2&lt;sup&gt;31&lt;/sup&gt;–1)</td>
</tr>
<tr>
<td>1000...0010</td>
<td>8000 0002</td>
<td>2,147,483,650</td>
<td>-2,147,483,646</td>
<td>-(2&lt;sup&gt;31&lt;/sup&gt;–1)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1111...1101</td>
<td>FFFF FFFD</td>
<td>4,294,967,294</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>1111...1110</td>
<td>FFFF FFFE</td>
<td>4,294,967,295</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>1111...1111</td>
<td>FFFF FFFF</td>
<td>4,294,967,296</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>
### Signed Number Representation

**Example: 32-bits**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Unsigned Value</th>
<th>Signed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000...0000</td>
<td>0000 0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000...0001</td>
<td>0000 0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0000...0010</td>
<td>0000 0002</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0111...1101</td>
<td>7FFF FFFD</td>
<td>2,147,483,645</td>
<td>2,147,483,645</td>
</tr>
<tr>
<td>0111...1110</td>
<td>7FFF FFFE</td>
<td>2,147,483,646</td>
<td>2,147,483,646</td>
</tr>
<tr>
<td>0111...1111</td>
<td>7FFF FFFF</td>
<td>2,147,483,647</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>1000...0000</td>
<td>8000 0000</td>
<td>2,147,483,648</td>
<td>-2,147,483,648</td>
</tr>
<tr>
<td>1000...0001</td>
<td>8000 0001</td>
<td>2,147,483,649</td>
<td>-2,147,483,647</td>
</tr>
<tr>
<td>1000...0010</td>
<td>8000 0002</td>
<td>2,147,483,650</td>
<td>-2,147,483,646</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111...1101</td>
<td>FFFF FFFD</td>
<td>4,294,967,294</td>
<td>-3</td>
</tr>
<tr>
<td>1111...1110</td>
<td>FFFF FFFE</td>
<td>4,294,967,295</td>
<td>-2</td>
</tr>
<tr>
<td>1111...1111</td>
<td>FFFF FFFF</td>
<td>4,294,967,296</td>
<td>-1</td>
</tr>
</tbody>
</table>

Always one more negative number than positive numbers:

$-2,147,483,648, \ldots , -1, 0, 1, \ldots + 2,147,483,647$

$2^{31} \text{ values} + 2^{31} \text{ values} = 2^{32} \text{ values}$
### Ranges of Numbers Using “Signed” Values

...in the “two’s complement” system of number representation:

<table>
<thead>
<tr>
<th>Total Number of Values</th>
<th>Byte (8-bits)</th>
<th>Halfword (16-bits)</th>
<th>Word (32-bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$2^8$</td>
<td>$2^{16}$</td>
<td>$2^{32}$</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>64K</td>
<td>4G</td>
</tr>
<tr>
<td></td>
<td></td>
<td>65,536</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Ranges of Numbers Using “Signed” Values

...in the “two’s complement” system of number representation:

<table>
<thead>
<tr>
<th></th>
<th>Total Number of Values</th>
<th>Largest Positive Number</th>
<th>Most Negative Number</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte</strong> (8-bits)</td>
<td>(2^8)</td>
<td>(2^7-1)</td>
<td>(-(2^7))</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>127</td>
<td>-128</td>
</tr>
<tr>
<td><strong>Halfword</strong> (16-bits)</td>
<td>(2^{16})</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64K</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>65,536</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Word</strong> (32-bits)</td>
<td>(2^{32})</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4G</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4,294,967,296</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Ranges of Numbers Using “Signed” Values

...in the “two’s complement” system of number representation:

<table>
<thead>
<tr>
<th></th>
<th>Total Number of Values</th>
<th>Largest Positive Number</th>
<th>Most Negative Number</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte</strong> (8-bits)</td>
<td>2^8</td>
<td>2^7-1</td>
<td>-(2^7)</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>127</td>
<td>-128</td>
</tr>
<tr>
<td><strong>Halfword</strong> (16-bits)</td>
<td>2^16</td>
<td>2^15-1</td>
<td>-(2^15)</td>
</tr>
<tr>
<td></td>
<td>64K</td>
<td>32K-1</td>
<td>-32K</td>
</tr>
<tr>
<td></td>
<td>65,536</td>
<td>32,767</td>
<td>-32,768</td>
</tr>
<tr>
<td><strong>Word</strong> (32-bits)</td>
<td>2^32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4G</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4,294,967,296</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Ranges of Numbers Using “Signed” Values

...in the “two’s complement” system of number representation:

<table>
<thead>
<tr>
<th></th>
<th>Total Number of Values</th>
<th>Largest Positive Number</th>
<th>Most Negative Number</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte</strong> (8-bits)</td>
<td>$2^8$</td>
<td>$2^7-1$</td>
<td>$-(2^7)$</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>127</td>
<td>-128</td>
</tr>
<tr>
<td><strong>Halfword</strong> (16-bits)</td>
<td>$2^{16}$</td>
<td>$2^{15}-1$</td>
<td>$-(2^{15})$</td>
</tr>
<tr>
<td></td>
<td>64K</td>
<td>32K-1</td>
<td>-32K</td>
</tr>
<tr>
<td></td>
<td>65,536</td>
<td>32,767</td>
<td>-32,768</td>
</tr>
<tr>
<td><strong>Word</strong> (32-bits)</td>
<td>$2^{32}$</td>
<td>$2^{31}-1$</td>
<td>$-(2^{31})$</td>
</tr>
<tr>
<td></td>
<td>4G</td>
<td>2G-1</td>
<td>-2G</td>
</tr>
<tr>
<td></td>
<td>4,294,967,296</td>
<td>2,147,483,647</td>
<td>-2,147,483,648</td>
</tr>
</tbody>
</table>
Leading Zeros Can Be Ignored

**Decimal**

1,234
... 000,000,000,000,001,234

**Hex**

3A0F1C
... 0000 0000 0000 003A 0F1C

**Binary**

101011010
... 0000 0000 0000 0001 0101 1010
Two's Complement Numbers:
Leading Ones Can Be Ignored

Decimal
−9,099

Hex
DC75
... FFFF FFFF FFFF FFFF DC75

Binary
1101 1100 0111 0101
... 1111 1111 1111 1111 1111 1101 1100 0111 0101
Sign Extension

Increase the size of a number by copying the “Sign Bit”
Sign Extension

*Increase the size of a number by copying the “Sign Bit”*

**Binary - Positive**

0110 1010
0000 0000 0110 1010
0000 0000 0000 0000 0000 0000 0110 1010

**Binary - Negative**

1100 0101
1111 1111 1100 0101
1111 1111 1111 1111 1111 1111 1100 0101

**Hex - Positive**

6A = 106₁₀
006A
0000 006A

**Hex - Negative**

C5 = −59₁₀
FFC5
FFFF FFC5

0-7: *positive or zero*

8-F: *negative*
Reducing the Size

Eliminate the leading bits.
Must not change the sign bit!!!
Must not eliminate significant bits!!!

**Binary**

\[
0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0110 \ 1010 = 106_{10} \\
0110 \ 1010 \\
1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1100 \ 0101 = -59_{10}
\]

**Hex**

\[
0000 \ 06A = 106_{10} \\
FFFF \ FFC5 = -59_{10}
\]
Reducing the Size

Eliminate the leading bits.
Must not change the sign bit!!!
Must not eliminate significant bits!!!

**Binary**

0000 0000 0000 0000 0000 0000 1100 0101 = $197_{10}$

1111 1111 1111 1111 1111 1111 0110 1010 = $-150_{10}$

**Hex**

0000 00C5 = $197_{10}$

FFFF FF6A = $-150_{10}$
Reducing the Size

Eliminate the leading bits.
Must not change the sign bit!!!
Must not eliminate significant bits!!!

**Binary**

\[
\begin{align*}
0000 & \ 0000 \ 0000 \ 0000 \ 0000 & 1100 \ 0101 & = 197_{10} \\
1100 & \ 0101 & = -59_{10} \\
1111 & \ 1111 \ 1111 \ 1111 \ 1111 \ 0110 \ 1010 & = -150_{10} \\
0110 & \ 1010 & = 106_{10}
\end{align*}
\]

**Hex**

\[
\begin{align*}
0000 \ 00C5 & = 197_{10} \\
C5 & = -59_{10} \\
FFFF \ FF6A & = -150_{10} \\
6A & = 106_{10}
\end{align*}
\]
Reducing the Size

Eliminate the leading bits.
Must not change the sign bit!!!
Must not eliminate significant bits!!!

**Binary**

- 0000 0000 0000 0000 0000 0000 1100 0101 = 197\_10
- 1100 0101 = –59\_10
- 1111 1111 1111 1111 1111 1111 0110 1010 = –150\_10
- 0110 1010 = 106\_10

**Hex**

- 0000 00C5 = 197\_10
- C5 = –59\_10
- FFFF FF6A = –150\_10
- 6A = 106\_10

**OOPS!**
What does C do?

```c
char c;
short s;
int i;

c = -123;
s = c;
i = c;

printf ("i = %d\n", i);
printf ("s = %d\n", s);
printf ("c = %d\n", c);
```

Output:

```
i = -123
s = -123
c = -123
```
What does C do?

```c
char c;
short s;
int i;

c = -123;
s = c;
i = c;

printf ("i = %d\n", i);
printf ("s = %d\n", s);
printf ("c = %d\n", c);
```

Output:

```
i = -123
s = -123
c = -123
```

Sign extended; no problem

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>85</td>
<td>-123</td>
</tr>
<tr>
<td>FF85</td>
<td>-123</td>
</tr>
<tr>
<td>FFFF FF85</td>
<td>-123</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 0101</td>
<td>-123</td>
</tr>
<tr>
<td>1111 1111 1000 0101</td>
<td>-123</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1000 0101</td>
<td>-123</td>
</tr>
</tbody>
</table>
What does C do?

```c
char c;
short s;
int i;

i = -59;
s = i;
c = i;

printf ("i = %d\n", i);
printf ("s = %d\n", s);
printf ("c = %d\n", c);
```

```
Output:
i = -59
s = -59
c = -59
```

Sometimes, truncation does not change the value…
What does C do?

```c
char c;
short s;
int i;

i = 100000;
s = i;
c = i;

printf ("i = %d\n", i);
printf ("s = %d\n", s);
printf ("c = %d\n", c);
```

Output:

```
i = 100000
s = -31072
c = -96
```

Sometimes it is a problem!
What does C do?

char c;
short s;
int i;

i = 100000;
s = i;
c = i;

printf ("i = %d\n", i);
printf ("s = %d\n", s);
printf ("c = %d\n", c);

Output:
i = 100000
s = -31072
c = -96

Sometimes it is a problem!
Casting in C

The default is “signed”
Use unsigned keyword if you want it.

Casting from signed to unsigned:

```
short int x = 15213;
unsigned short int ux = (unsigned short) x;
short int y = -15213;
unsigned short int uy = (unsigned short) y;
```

Copies bits without processing.
Does not change bit values.

Non-negative values will be unchanged.
Negative values change into large positive values

Why? Sign bit becomes the most significant bit.
Casting in C

The default is “signed”
Use unsigned keyword if you want it.

Casting from signed to unsigned:

```
short int x = 15213;
unsigned short int ux = (unsigned short) x;
short int y = -15213;
unsigned short int uy = (unsigned short) y;
```

Copies bits without processing.
Does not change bit values.

Non-negative values will be unchanged.
Negative values change into large positive values
  Why? Sign bit becomes the most significant bit.
Signed vs. Unsigned

Constants are assumed to be signed.
If you want a signed constant…

```
0U
123456U
```

You can cast between signed and unsigned.

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

Implicit casting
  Occurs in assignment statements
  Occurs when arguments are passed to functions
Compilers often warn about possible issues
  Always pay attention to compiler warnings and fix your code!
Casting Surprises

What relation do you expect these to have?
When both are mixed, the signed value is cast to unsigned.

0 == 0U   unsigned
-1 < 0    signed
-1 > 0U   unsigned
2147483647 > -2147483648   signed
2147483647U < -2147483648   unsigned
-1 > -2   signed
(unsigned) -1 > -2   unsigned
2147483647 < 2147483648U   unsigned
2147483647 > (int) 2147483648U   signed
An Example Bug

Code for determining which string is longer. What goes wrong here?

```c
size_t strlen(const char*);

int strlonger(char *s, char *t) {
    return (strlen(s) - strlen(t)) > 0;
};
```

Imagine we use this to test to see if something will fit in a buffer. It returns true. We copy the string in to the buffer. Buffer overrun → system security violation
Logical Functions

1 = True
0 = False

*Input:* Two Bits (two logical values)
*Output:* One Bit (one logical value)

The Logical Functions:

**AND**
True if and only if both inputs are true.
Output 1 iff both inputs are 1.

**OR**
True if and only if either input is true.
Output 1 if either or both inputs are 1.

**XOR (exclusive-or)**
True if and only if exactly one input is true.
Output 1 iff the inputs are different

**NOT**
Only one input
Flip the bit; output the opposite value.
Logical Function: AND

1 = True
0 = False

Input: Two Bits (two logical values)
Output: One Bit (one logical value)

The output is 1 iff both inputs are 1.
If either input is 0, then the output is 0.
Both have to be TRUE to make the output TRUE.

AND

“Apples are red.”  1  TRUE
“Lemons are green.”  0  FALSE
“Apples are red AND lemons are green”  0  FALSE

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
</tr>
</tbody>
</table>
Logical Function: OR

1 = True
0 = False

**Input:** Two Bits (two logical values)
**Output:** One Bit (one logical value)

The output is 1 if either input is 1.
If both inputs are 0, then the output is 0.
Both have to be FALSE to make the output FALSE.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

“Apples are red.”
“Lemons are green.”
“Apples are red OR lemons are green”

1 TRUE
0 FALSE
1 TRUE
Logical Function: XOR

1 = True
0 = False

**Input:** Two Bits (two logical values)
**Output:** One Bit (one logical value)

The output is 1 if one input is 1.
If both inputs are 0 (or both are 1),
then the output is 0.
Both have to be DIFFERENT to make
the output TRUE.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
</tr>
</tbody>
</table>

“Apples are red.”
“Lemons are green.”
“Apples are red XOR lemons are green”

1  TRUE
0  FALSE
1  TRUE
Logical Function: NOT

1 = True
0 = False

Input: Two Bits (two logical values)
Output: One Bit (one logical value)

If the input is 0, the output is 1.
If the input is 1, the output is 0.
The input is flipped.

XOR

"Lemons are green."
"Lemons are NOT green"

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Performing Logical Operations on Larger Values

AND &

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\]
Performing Logical Operations on Larger Values

OR

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
\end{array}
\]
Performing Logical Operations on Larger Values

XOR

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\hline
0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
\end{array}
\]
Performing Logical Operations on Larger Values

NOT \sim

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
\end{array}
\]
Performing Logical Operations on Larger Values

EQUAL

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\hline
1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
\end{array}
\]
Performing Logical Operations on Larger Values

**EQUAL**

<table>
<thead>
<tr>
<th>1 1 1 0 1 1 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>1 0 1 1 1 0 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 1 1 0 1 1 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

|=|=|

<table>
<thead>
<tr>
<th>1 1 1 0 1 1 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1 0 0</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 1</td>
</tr>
</tbody>
</table>
Performing Logical Operations on Larger Values

**EQUAL**

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

**Not-XOR**

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

---

© Harry H. Porter, 2014
**Addition**

**Decimal:**

\[
\begin{array}{c}
1 & 1 & 1 \\
3 & 8 & 5 & 3 \\
\hline
+ & 9 & 3 & 7 & 4 \\
\hline
1 & 3 & 2 & 2 & 7 \\
\end{array}
\]

**Binary:**

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

#### Decimal:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>9</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Binary:

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 + 0 = 0
1 + 0 = 1
1 + 1 = 10
1 + 1 + 1 = 11

e tc.
**Addition:**

The algorithm is the same for SIGNED and UNSIGNED.

**8-bit Unsigned:**

\[
\begin{align*}
1110 \ 1100 &= 236 \\
+ \ 1010 \ 1010 &= 170 \\
\hline
1 \ 1001 \ 0110 &= 406
\end{align*}
\]

**8-bit Signed:**

\[
\begin{align*}
1110 \ 1100 &= -20 \\
+ \ 1010 \ 1010 &= -86 \\
\hline
1 \ 1001 \ 0110 &= -106
\end{align*}
\]
**Addition:**

The algorithm is the same for SIGNED and UNSIGNED. Overflow detection is slightly different.

**8-bit Unsigned:**

\[
\begin{align*}
1110\ 1100 & = 236 \\
+\ 1010\ 1010 & = 170 \\
\hline
1\ 1001\ 0110 & = 406
\end{align*}
\]

**8-bit Signed:**

\[
\begin{align*}
1110\ 1100 & = -20 \\
+\ 1010\ 1010 & = -86 \\
\hline
1\ 1001\ 0110 & = -106
\end{align*}
\]

*Overflow! (max value = 255)*
Addition:
The algorithm is the same for SIGNED and UNSIGNED. Overflow detection is slightly different.

<table>
<thead>
<tr>
<th>8-bit Unsigned:</th>
<th>8-bit Signed:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 1100</td>
<td>1110 1100</td>
</tr>
<tr>
<td>+ 1010 1010</td>
<td>+ 1010 1010</td>
</tr>
<tr>
<td>1 1001 0110</td>
<td>1 1001 0110</td>
</tr>
<tr>
<td>1110 1100</td>
<td>1110 1100</td>
</tr>
<tr>
<td>+ 1010 1010</td>
<td>+ 1010 1010</td>
</tr>
<tr>
<td>1 1001 0110</td>
<td>1 1001 0110</td>
</tr>
</tbody>
</table>

Overflow! (max value = 255)

Subtraction:
The algorithm is the same for SIGNED and UNSIGNED. Overflow detection is slightly different.
Addition:
The algorithm is the same for SIGNED and UNSIGNED. Overflow detection is slightly different.

8-bit Unsigned:
\[
\begin{array}{c|c}
1110 1100 & = 236 \\
+ 1010 1010 & = 170 \\
\hline
1 1001 0110 & = 406 \\
\end{array}
\]

8-bit Signed:
\[
\begin{array}{c|c}
1110 1100 & = -20 \\
+ 1010 1010 & = -86 \\
\hline
1 1001 0110 & = -106 \\
\end{array}
\]

Subtraction:
The algorithm is the same for SIGNED and UNSIGNED. Overflow detection is slightly different.

Multiplication:
Two algorithms.

8-bit Signed:
\[
\begin{array}{c|c|c}
1111 1110 & = -2 \\
\times 1111 1110 & = -2 \\
\hline
0000 0000 0000 0100 & = +4 \\
\end{array}
\]

8-bit Unsigned:
\[
\begin{array}{c|c}
1111 1110 & = 254 \\
\times 1111 1110 & = 254 \\
\hline
1111 1100 0000 0100 & = 64,516 \\
\end{array}
\]

(Note: Result may be twice as long as operands.)
**Addition:**
The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

8-bit Unsigned:  
1110 1100 = 236  
+ 1010 1010 = 170  
1 1001 0110 = 406

8-bit Signed:  
1110 1100 = -20  
+ 1010 1010 = -86  
1 1001 0110 = -106

**Subtraction:**
The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

**Multiplication:**
Two algorithms.

8-bit Signed:  
1111 1110 = -2  
\times 1111 1110 = -2  
0000 0000 0000 0100 = +4

8-bit Unsigned:  
1111 1110 = 254  
\times 1111 1110 = 254  
1111 1110 1111 0110 = 64,516

(NOTE: Result may be twice as long as operands.)

**Division:**
Two algorithms.
Arithmetic Negation

The Algorithm to Negate a Signed Number:
Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

```
0000 0010
```

complementing:
add 1:

```
= 2
```
Arithmetic Negation

The Algorithm to Negate a Signed Number:
Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

\[
\begin{align*}
0000 0010 &= 2 \\
\text{complementing:} &\quad 1111 1101 \\
\text{add 1:} &\quad 1111 1010
\end{align*}
\]
Arithmetic Negation

The Algorithm to Negate a Signed Number:
Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

0000 0010 = 2
complementing: 1111 1101
add 1: +0000 0001
Arithmetic Negation

The Algorithm to Negate a Signed Number:
Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

0000 0010  =  2
complementing: 1111 1101
add 1:          +0000 0001
1111 1110  =  -2
**Arithmetic Negation**

**The Algorithm to Negate a Signed Number:**
Bitwise complement (i.e., logical NOT)
Followed by “add 1”

**Example:**

\[
\begin{align*}
\text{0000 0010} & = 2 \\
\text{complementing: 1111 1101} & \\
\text{add 1: +0000 0001} & \\
\text{1111 1110} & = -2
\end{align*}
\]

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.
Arithmetic Negation

The Algorithm to Negate a Signed Number:
Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

\[
\begin{align*}
\text{0000 0010} & = 2 \\
\text{complementing: 1111 1101} \\
\text{add 1:} & +0000 0001 \\
& \quad 1111 1110 = -2
\end{align*}
\]

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

\[
\begin{align*}
\text{1000 0000} & = -128 \\
\text{complementing:} \\
\text{add 1:}
\end{align*}
\]
Arithmetic Negation

The Algorithm to Negate a Signed Number:
Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

\[ \begin{align*}
0000 \ 0010 &= 2 \\
\text{complementing:} & \quad 1111 \ 1101 \\
\text{add 1:} & \quad +0000 \ 0001 \\
& \quad 1111 \ 1110 = -2
\end{align*} \]

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

\[ \begin{align*}
1000 \ 0000 &= -128 \\
\text{complementing:} & \quad 0111 \ 1111 \\
\text{add 1:} & \quad 0111 \ 1111
\end{align*} \]
**Arithmetic Negation**

**The Algorithm to Negate a Signed Number:**
Bitwise complement (i.e., logical NOT)
Followed by “add 1”

**Example:**

```
0000 0010 = 2
complementing: 1111 1101
add 1: +0000 0001
1111 1110 = -2
```

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

**8-Bit Example:**

```
1000 0000 = -128
complementing: 0111 1111
add 1: +0000 0001
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:
Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

\[
\begin{array}{c|c}
0000\ 0010 & = \ 2 \\
\text{complementing:} & 1111\ 1101 \\
\text{add 1:} & +0000\ 0001 \\
& 1111\ 1110 = -2 \\
\end{array}
\]

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

\[
\begin{array}{c|c}
1000\ 0000 & = -128 \\
\text{complementing:} & 0111\ 1111 \\
\text{add 1:} & +0000\ 0001 \\
& 1000\ 0000 = -128 \\
\end{array}
\]
Arithmetic Negation

The Algorithm to Negate a Signed Number:
Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

\[
\begin{align*}
0000\ 0010 & = 2 \\
\text{complementing:} & \quad 1111\ 1101 \\
\text{add 1:} & \quad +0000\ 0001 \\
& \quad 1111\ 1110 = -2
\end{align*}
\]

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

\[
\begin{align*}
1000\ 0000 & = -128 \\
\text{complementing:} & \quad 0111\ 1111 \\
\text{add 1:} & \quad +0000\ 0001 \\
& \quad 1000\ 0000 = -128
\end{align*}
\]

The most negative 32-bit number, 0x80000000
Hex: 8 0 0 0 0 0 0 0 0 0 0
Binary: 1000 0000 0000 0000 0000 0000 0000 0000
Decimal: -2,147,483,648
# Storing Numbers In Memory

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit Size</th>
<th>Byte Size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte</strong></td>
<td></td>
<td>8 bits</td>
</tr>
<tr>
<td><strong>Halfword</strong></td>
<td></td>
<td>16 bits = 2 bytes</td>
</tr>
<tr>
<td><strong>Word</strong></td>
<td></td>
<td>32 bits = 4 bytes</td>
</tr>
<tr>
<td><strong>Doubleword</strong></td>
<td></td>
<td>64 bits = 8 bytes</td>
</tr>
<tr>
<td><strong>Quadword</strong></td>
<td></td>
<td>128 bits = 16 bytes</td>
</tr>
</tbody>
</table>
Byte Ordering

Big Endian
Sparc (Sun), PowerPC

Little Endian
Intel (Macs, PCs)

Example: 32-bit Integer Value
0x01234567

Big Endian:

Little Endian:
Main Memory Organization

![Diagram of memory organization with hexadecimal values and byte labels.](image-url)
Main Memory Organization

Addresses are 32 bits (up to $2^{32} = 4$ Gbytes)
Main Memory Organization

Addresses are 32 bits (up to $2^{32} = 4$ Gbytes)

A WORD

Address of this word

Low Memory

High Memory

byte
Main Memory Organization

Address of this word

Low Memory

MSB (Most significant byte)

A WORD

Address of this word

LSB (Least significant byte)

ASMS (Least significant byte)

Addresses are 32 bits (up to $2^{32} = 4$ Gbytes)

Low Memory

High Memory
Main Memory Organization

Address of this word

Addresses are 32 bits (up to $2^{32} = 4$ Gbytes)

Big Endian Architecture (e.g., SPARC)

Low Memory

MSB (Most significant byte)

A WORD

LSB (Least significant byte)

High Memory
Main Memory Organization

Address of this word

Addresses are 32 bits (up to $2^{32} = 4$ Gbytes)

Little Endian Architecture (e.g., Intel)

Low Memory

LSB (Least significant byte)

MSB (Most significant byte)

High Memory
# Program to Show Byte Order

```c
#include <stdio.h>
#include <string.h>

void show_bytes(unsigned char * start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf(" %2.2x", start[i]);
    printf("\n");
}

int i=0x01020304;
float f=123.456;
int *ip=&i;
char *s = "ABCDEF";

int main() {
    show_bytes ((char *) &i, sizeof(int));
    show_bytes ((char *) &f, sizeof(float));
    show_bytes ((char *) &ip, sizeof(char *));
    show_bytes (s, strlen(s));
}
```

**Output (Sun, Big Endian):**

```
% a.out
01 02 03 04
42 f6 e9 79
00 00 00 01 00 10 12 00
41 42 43 44 45 46
```
# Program to Show Byte Order

```c
#include <stdio.h>
#include <string.h>
void show_bytes(unsigned char * start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%2.2x", start[i]);
    printf("\n");
}

int i=0x01020304;
float f=123.456;
int *ip=&i;
char *s = "ABCDEF"

int main() {
    show_bytes ((char *) &i, sizeof(int));
    show_bytes ((char *) &f, sizeof(float));
    show_bytes ((char *) &ip, sizeof(char *));
    show_bytes (s, strlen(s));
}
```

**Output (Mac, Little Endian):**

```
% a.out
04 03 02 01
79 e9 f6 42
28 f0 63 0d 01 00 00 00
41 42 43 44 45 46
```
Data Alignment

Data stored in memory must be “aligned” according to the length of the data

**Byte Data**
- can go at any address

**Halfword Data**
- must be “halfword aligned”
- addresses must be even numbers

**Word Data**
- must be “word aligned”
- addresses must be divisible by 4

**Doubleword Data**
- must be “doubleword aligned”
- addresses must be divisible by 8
byte

word / int (32 bits)

halfword / short (16 bits)

double (64 bits)
Halfword addresses end in an even number:
0, 2, 4, 6, 8, a, c, e
In binary:
---- ---- ---- ---- ---- ---- ---- ---0
Word addresses end in a number divisible by 4: 0, 4, 8, c

In binary:
---- ---- ---- ---- ---- ---- ---- --00
Doubleword addresses end in a number divisible by 8: 0, 8
In binary: ---- ---- ---- ---- ---- -000
Fixed-Point Numbers

Decimal

\[ 123.456 \]

\[ \ldots 10^2 10^1 10^0 10^{-1} 10^{-2} 10^{-3} \ldots \]

\[ \ldots 100 10 1 \frac{1}{10} \frac{1}{100} \frac{1}{1000} \ldots \]

Binary

\[ 101.0101 \]

\[ \ldots 2^2 2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4} \ldots \]

\[ \ldots 4 2 1 \frac{1}{2} \frac{1}{4} \frac{1}{8} \frac{1}{16} \ldots \]

What is this number? \[ 4 + 1 + \frac{1}{4} + \frac{1}{16} = 5 \frac{5}{16} = 5.3125 \]
“Every binary fraction can be represented exactly with a decimal fraction.”

\[ 1001.01_2 = 9.25_{10} \]

(And the decimal representation will use no more decimal digits to the right of “.” than the binary number has bits.)

“Some decimal fractions cannot be represented exactly using binary fractions.”

\[ 0.3_{10} = 0.0100110011001100110011..._2 \]

\[ = 0.010011_2 \]

(of finite length)
Floating Point Numbers

Decimal

\[
123.456 = 1.2345 \times 10^2
\]
\[
6.0225 \times 10^{23}
\]

Limited precision: Rounded to the nearest \(10^{19}\)
The leading digit will always be 1,2,3, ..., 9 (never 0).
Floating Point Numbers

**Decimal**

123.456

= 1.2345 \times 10^2

6.0225 \times 10^{23}

Limited precision: Rounded to the nearest $10^{19}$
The leading digit will always be 1,2,3, ..., 9 (never 0).

**Binary**

101.0101

= 1.010101 \times 2^2

= 1.328125 \times 4 = 5.3125

Note: The leading bit will always be “1” (never “0”).
No need to store the first bit!
Single Precision Floating Point
Number Representation

\[ N = (-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{\text{exp}} \]

8 bit exponent
23 bit fraction

Sign bit 0=pos, 1=neg

Range: -126..127
Single Precision Floating Point
Number Representation

\[ N = (-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{\text{exp}} \]

The exponent is an 8 bit number interpreted as follows...

- 0000 0000  “sub-normal”
- 0000 0001  -126
- 0111 1110  -1
- 0111 1111  0
- 1000 0000  1
- 1111 1110  127
- 1111 1111  “not a number”

Sign bit  0=pos, 1=neg
Range: -126..127
Single Precision Floating Point
Number Representation

\[ N = (-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{\text{exp}} \]

The exponent is an 8 bit number interpreted as follows...

- 0000 0000  "sub-normal"
- 0000 0001  -126
- ...  ...
- 0111 1110  -1
- 0111 1111  0
- 1000 0000  1
- ...  ...
- 1111 1110  127
- 1111 1111  "not a number"

Single-Precision

Smallest non-zero number: 1.17549435 \times 10^{-38}
Largest number: 3.40282347 \times 10^{+38}
About 9 digits of accuracy!
Zero

31 30 23 22 0

8 bit exponent 23 bit fraction

Sign bit 0=pos, 1=neg

Sign = 0 (positive)
1 (negative)

Exp = 00000000

Fraction = 000000000000000000000000000000000

0x0000 00000 (= positive zero)
0x8000 00000 (= negative zero)

Similar for double precision.
Other Special Cases

When

\[
\text{exp} = 1111\ 1111
\]

a special meaning arises

**Not A Number** “NaN”

\[
0xFFFF\ FFFF
\]

(= -1 as a signed number)

Will cause an exception when used.

**Positive Infinity** “+inf”

\[
+\infty
0x7F80\ 0000
\]

**Negative Infinity** “-inf”

\[
-\infty
0xFF80\ 0000
\]

Divide \( \frac{1}{0} \) \( \Rightarrow +\infty \)

Divide \( -\frac{1}{0} \) \( \Rightarrow -\infty \)

You can compare other numbers to \( +\infty \) and \( -\infty \).
Distribution of Numbers

Example: 6-bit floating point numbers
  Exponent: 3 bits
  Fraction: 2 bits

Notice how the density is greater close to zero:
NaN Details

Representation:

\[
\begin{array}{c}
\text{31} \\
\text{30} \\
\text{29} \\
\text{28} \\
\text{27} \\
\text{26} \\
\text{25} \\
\text{24} \\
\text{23} \\
\text{22} \\
\text{21} \\
\text{20} \\
\text{19} \\
\text{18} \\
\text{17} \\
\text{16} \\
\text{15} \\
\text{14} \\
\text{13} \\
\text{12} \\
\text{11} \\
\text{10} \\
\text{9} \\
\text{8} \\
\text{7} \\
\text{6} \\
\text{5} \\
\text{4} \\
\text{3} \\
\text{2} \\
\text{1} \\
\text{0}
\end{array}
\]

- **8 bit exponent**
- **Sign bit**
- **23 bit fraction (any value)**

The fraction is ignored.

Any value

from

\[\text{FF80 0000}\]

to

\[\text{FFFF FFFF}\]

Indicates an “invalid result.”

\[0 \div 0 \Rightarrow \text{NaN}\]

Operands preserve Nan

\[3.75 + \text{NaN} \Rightarrow \text{NaN}\]
Denormalized Numbers

Normalized:

\[ N = (-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{\text{exp}} \]

smallest value \( 1.000\ldots000 \times 2^{-126} \)

Denormalized:

\[ N = (-1)^{\text{sign}} \times 0.\text{fraction} \times 2^{-126} \]

largest value \( 0.111\ldots111 \times 2^{-126} \)

smallest value \( 0.000\ldots001 \times 2^{-126} \)

positive zero \( 0.000\ldots000 \times 2^{-126} \)
Denormalized Numbers

Denormalized values are very close to zero. They have reduced precision. +0.0 and -0.0 are special cases of denormalized numbers.

Example (using floats with only 6 bits)
Floating Point Properties

Addition

Commutative: \( x + y = y + x \)

Not associative: \( (x + y) + z \neq x + (y + z) \)
due to rounding

Example:
\[
(3.14 + 1e10) - 1e10 = 0.0, \text{ due to rounding}
3.14 + (1e10 - 1e10) = 3.14
\]

Multiplication

Not associative

Multiplication does not distribute over addition

Example:
\[
1e20 \times (1e20 - 1e20) = 0.0
(1e20 \times 1e20) - (1e20 \times 1e20) = \text{NaN}
\]
Double Precision Floating Point
Number Representation

\[ N = (-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{\text{exp}} \]

**Double-Precision**

**Smallest non-zero number:**
\[ 2.225,073,858,507,201,4 \times 10^{-308} \]

**Largest number:**
\[ 1.797,693,134,862,315,7 \times 10^{+308} \]

About 17 digits of accuracy!
Logical Functions

\[
\begin{align*}
\text{and} & \quad 0 & \quad 0 & \quad 0 & \quad 0 \\
\text{or} & \quad 0 & \quad 1 & \quad 1 & \quad 1 \\
\text{xor} & = (x \neq y) & \quad 0 & \quad 1 & \quad 1 & \quad 0
\end{align*}
\]

Instructions work on all 64 bits at once:

\[
\text{andq } \%rcx, \%rax
\]

\[
\begin{align*}
\%rax & \rightarrow 0011 \ 1100 \ \ldots \ 1010 \\
\%rcx & \rightarrow 1010 \ 1101 \ \ldots \ 1001 \\
\%rax & \rightarrow 0010 \ 1100 \ \ldots \ 1000
\end{align*}
\]
Logical Functions in C

Operate on integer data
char, int, short, long long
Each operand is a bit vector

**And**
\( x = y \& z; \)
\[
\begin{array}{cccccccc}
1010 & 1100 & 0110 & 0010 \\
0101 & 0111 & 0101 & 1010 \\
0000 & 0100 & 0100 & 0010 \\
\end{array}
\]

**Or**
\( x = y | z; \)
\[
\begin{array}{cccccccc}
1010 & 1100 & 0110 & 0010 \\
0101 & 0111 & 0101 & 1010 \\
1111 & 1111 & 0111 & 1010 \\
\end{array}
\]

**Exclusive-Or**
\( x = y \^{} z; \)
\[
\begin{array}{cccccccc}
1010 & 1100 & 0110 & 0010 \\
0101 & 0111 & 0101 & 1010 \\
1111 & 1011 & 0011 & 1000 \\
\end{array}
\]
To turn on bits in a word...
Use the “or” instruction and a “mask” word
\[ x \text{ or mask} \rightarrow \text{result} \]
Turn on bits in x wherever the mask has a 1 bit

Example: Turn on every other bit in 3A0F
\[
\begin{array}{cccccc}
0011 & 1010 & 0000 & 1111 & \rightarrow & 3A0F \\
0101 & 0101 & 0101 & 0101 & \rightarrow & \text{mask} \\
0111 & 1111 & 0101 & 1111 & \rightarrow & \text{result}
\end{array}
\]

To turn off bits in a word...
Use the “and” instruction and a mask
\[ x \text{ and mask} \rightarrow \text{result} \]
Turn off bits in x wherever the mask has a 0 bit

To flip (or “toggle”) bits in a word...
Use the “xor” instruction and a mask
\[ x \text{ xor mask} \rightarrow \text{result} \]
Change the bits in x wherever the mask has a 1 bit
Shifting Instructions

**shl**

“Shift Left” $\ll$

\[
\text{shl} \quad \text{int,reg}
\]

A fast way to multiply by $2^N$...

**Example:** Multiply by 32 ($= 2^5$)

\[
\text{shl} \quad $5,\%eax
\]

0000 0000 0000 0011 = 3
0000 0000 0110 0000 = 64+32 = 96

**shr**

“Shift Logical Right” $\gg$

\[
\text{shr} \quad \text{int,reg}
\]

**sar**

“Shift Arithmetic Right” $\gg$

\[
\text{sar} \quad \text{int,reg}
\]

A fast way to divide by $2^N$, rounding toward $-\infty$...

\[
\text{sar} \quad $3,\%eax
\]
Testing

cmp reg1, reg2

Compare operand1 to operand2

Set integer condition codes accordingly

The next instruction will normally be a conditional branch

Example:

cmp $73, %rax  ! if x <= 73 goto loop
ble loop      ! .

Branch if the condition codes indicate \( op2 \leq op1 \)
Pointers

**A pointer is a memory address.**

Pointers are “typed”
...the type of the object at that address

*Pointers are typed in order to determine what gets returned when the pointer is dereferenced.*

Use “*” to declare a pointer type

```c
char* cp;  // cp points to a character in memory
int* ip;   // ip points to an integer in memory
```

“&” operator gives address of object

```c
int x;
int* p;
```

What is the data type of `p`?
What is the data type of `*p`?
What is the data type of `&x`?
Pointers

Given the following code...

```c
main() {
    int B = -15213;
    int* P = &B;
}
```

Suppose
The address of \( B \) is \( 0xbfffff8d4 \)
The address of \( P \) is \( 0xbfffff8d0 \)

What is the value of \( P \)?
What is the size of \( P \)?
Write the value of each byte of \( P \) in order as they appear in memory.
Pointers

Given the following code...

```c
main() {
    int B = -15213;
    int* P = &B;
}
```

Suppose
- The address of `B` is `0xbffff8d4`
- The address of `P` is `0xbffff8d0`

What is the value of `P`?
What is the size of `P`?
Write the value of each byte of `P` in order as they appear in memory.
Pointers

Given the following code...

```c
main() {
    int B = -15213;
    int* P = &B;
}
```

Suppose

- The address of `B` is 0xbfffff8d4
- The address of `P` is 0xbfffff8d0

What is the value of `P`?
What is the size of `P`?
Write the value of each byte of `P` in order as they appear in memory.
Pointers

Given the following code...

```c
main() {
    int B = -15213;
    int* P = &B;
}
```

Suppose
- The address of `B` is 0xbffff8d4
- The address of `P` is 0xbffff8d0

What is the value of `P`?
What is the size of `P`?
Write the value of each byte of `P` in order as they appear in memory.
Given the following code...

```c
main() {
    int B = -15213;
    int* P = &B;
}
```

Suppose

- The address of \( B \) is \( 0xbfffff8d4 \)
- The address of \( P \) is \( 0xbffфффf8d0 \)

What is the value of \( P \)?
What is the size of \( P \)?
Write the value of each byte of \( P \) in order as they appear in memory.
Pointers

Given the following code...

```c
main() {
    int B = -15213;
    int* P = &B;
}
```

Suppose

- The address of `B` is `0xbfffff8d4`
- The address of `P` is `0xbfffff8d0`

What is the value of `P`?
What is the size of `P`?
Write the value of each byte of `P` in order as they appear in memory.
Dereferencing pointers

Returns the data that is stored in the memory location

The unary operator *

Used to dereference a pointer variable

```c
int x = 1, y = 2, z[10];
int* ip = &x;
y = *ip; // y is now 1
*ip = 0; // x is now 0
```

Dereferencing uninitialized pointers: What happens?

```c
int* ip;
*ip = 3;
```

Segmentation fault (or worse: nothing so obvious!)
**Pointer Arithmetic**

Type determines what is returned when “dereferenced”
Also: pointer arithmetic is based on the type of data referenced.

- Incrementing an `int *` adds 4 to the pointer.
- Incrementing a `char *` adds 1 to the pointer.
- Incrementing a `int * *` adds 4 or 8 to the pointer.

**Example:**

```c
char* cp=0x100;
int* ip=0x200;
float* fp=0x300;
double* dp=0x400;
int i=0x500;
```

What are the hexadecimal values of each after each of these commands?

```c
cp++;
ip++;
fp++;
dp++;
i++;"```
Pointers and Arrays

Arrays are stored in one contiguous block of memory.
An array is a collection of values
   All the same type
   Indexed (or “accessed”) by number

Example

```c
int a[20];
```

The first element is
```
a[0]
```
The last element is
```
a[19]
```
The variable “a” is a pointer to int.

Similar to:
```
int * a;
```
```
i = *a;   i = a[0];
j = *(a+3);   j = a[3];
b = a+3;
```

Really adds 12
Example

```
#include <stdio.h>
main() {
    char* str="abcdefg\n";
    char* x;
    x = str;

    printf("str[0]: %c\n", str[0]);
    printf("str[1]: %c\n", str[1]);
    printf("str[2]: %c\n", str[2]);
    printf("str[3]: %c\n", str[3]);

    printf("x: %x  *x: %c\n", x, *x);
    x++;
    printf("x: %x  *x: %c\n", x, *x);
    x++;
    printf("x: %x  *x: %c\n", x, *x);
    x++;
    printf("x: %x  *x: %c\n", x, *x);
}
```
#include <stdio.h>

main() {
    int numbers[10], *num, i;

    for (i=0; i < 10; i++)
        numbers[i]=i;

    num = (int *) numbers;
    printf("num: %x  *num: %d\n", num, *num);   num++;
    printf("num: %x  *num: %d\n", num, *num);   num++;
    printf("num: %x  *num: %d\n", num, *num);   num++;
    printf("num: %x  *num: %d\n", num, *num);

    num = (int *) numbers;
    printf("numbers=%x      num=%x\n", numbers, num);
    printf("&numbers[7]=%x num+7=%x\n", &numbers[7], num+7);
    printf("numbers[7]=%d   *(num+7)=%d\n", numbers[7], *(num+7));
}

num: 5833fba0  *num: 0
num: 5833fba4  *num: 1
num: 5833fba8  *num: 2
num: 5833fbac  *num: 3
numbers=5833fba0 num=5833fba0
&numbers[7]=5833fbbc num+7=5833fbbc
numbers[7]=7  *(num+7)=7
Representing Strings

In C:

ASCII encoding of characters
Each character takes 1 byte
   (There are other encodings)
Character “0” has code **0x30**
Digit \(i\) has code **0x30+i**
The string should be “null terminated”
   **0x00 = NUL = ‘\0’**
Byte ordering is not an issue
   Big Endian = Little Endian
Text files are usually platform independent

But line termination can be a problem.

\n  **0x0A**
\r  **0x0D**
\n\r  **0x0A0D**

```
char mystr[6] = "15213";
```
ASCII Character Set

! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
# ASCII Character Set

All printable characters with decimal codes

<table>
<thead>
<tr>
<th>ASCII Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>!</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
</tr>
<tr>
<td>39</td>
<td>'</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
</tr>
<tr>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>69</td>
<td>E</td>
</tr>
<tr>
<td>81</td>
<td>Q</td>
</tr>
<tr>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
</tr>
<tr>
<td>58</td>
<td>:</td>
</tr>
<tr>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>82</td>
<td>R</td>
</tr>
<tr>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
</tr>
<tr>
<td>59</td>
<td>;</td>
</tr>
<tr>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>95</td>
<td>_</td>
</tr>
<tr>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>72</td>
<td>H</td>
</tr>
<tr>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>96</td>
<td>~</td>
</tr>
<tr>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>61</td>
<td>=</td>
</tr>
<tr>
<td>73</td>
<td>I</td>
</tr>
<tr>
<td>85</td>
<td>U</td>
</tr>
<tr>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>112</td>
<td>y</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>63</td>
<td>?</td>
</tr>
<tr>
<td>75</td>
<td>K</td>
</tr>
<tr>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>99</td>
<td>c</td>
</tr>
<tr>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>64</td>
<td>@</td>
</tr>
<tr>
<td>76</td>
<td>L</td>
</tr>
<tr>
<td>88</td>
<td>X</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>77</td>
<td>M</td>
</tr>
<tr>
<td>89</td>
<td>Y</td>
</tr>
<tr>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>90</td>
<td>Z</td>
</tr>
<tr>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>55</td>
<td>7</td>
</tr>
<tr>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>79</td>
<td>O</td>
</tr>
<tr>
<td>91</td>
<td>[</td>
</tr>
<tr>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>124</td>
<td></td>
</tr>
</tbody>
</table>
### ASCII Chart

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Character</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>NUL</td>
<td>Control characters</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1F</td>
<td>31</td>
<td>(space)</td>
<td>Punctuation</td>
</tr>
<tr>
<td>20</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>33</td>
<td>!</td>
<td>Punctuation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>48</td>
<td>0</td>
<td>Digits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>57</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3A</td>
<td>58</td>
<td>:</td>
<td>Punctuation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>65</td>
<td>A</td>
<td>Uppercase</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5A</td>
<td>90</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5B</td>
<td>91</td>
<td>[</td>
<td>Punctuation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>97</td>
<td>a</td>
<td>Lowercase</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7A</td>
<td>122</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>7B</td>
<td>123</td>
<td>{</td>
<td>Punctuation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7F</td>
<td>127</td>
<td>DEL</td>
<td>Backspace</td>
</tr>
<tr>
<td>80</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Not used</td>
</tr>
<tr>
<td>FF</td>
<td>255</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Unicode

ASCII is only suitable for Roman / Latin alphabet. Unicode supports
  Russian, Greek, Chinese, math symbols, etc.

Unicode is the default for newer software
  Java
The C library contains some support.

Each characters is encoded with 32 bits per character.
  4 bytes
  Characters are called “code points”.

There are several “encodings”.
  UTF-8
  UTF-16
  UTF-32
## Unicode Examples

### ASCII/Latin-1  U+0000 – U+007F  (0–127)

<table>
<thead>
<tr>
<th>!</th>
<th>5</th>
<th>A</th>
<th>a</th>
<th>k</th>
</tr>
</thead>
</table>

### Latin-1 supplement  U+0080 – U+00FF  (128–255)

<table>
<thead>
<tr>
<th>¥</th>
<th>¢</th>
<th>€</th>
<th>¼</th>
<th>Ñ</th>
<th>ñ</th>
</tr>
</thead>
<tbody>
<tr>
<td>ü</td>
<td>×</td>
<td>÷</td>
<td>æ</td>
<td>©</td>
<td></td>
</tr>
</tbody>
</table>

### Cyrillic  U+0400 – U+04FF  (1024–1279)

<table>
<thead>
<tr>
<th>љ</th>
<th>ѳ</th>
<th>Ѥ</th>
<th>ѫ</th>
<th>Ж</th>
</tr>
</thead>
</table>

### Greek  U+0370 – U+03FF  (880–1023)

<table>
<thead>
<tr>
<th>Θ</th>
<th>Γ</th>
<th>Δ</th>
<th>Σ</th>
<th>Ψ</th>
</tr>
</thead>
</table>

### Misc.

<table>
<thead>
<tr>
<th>∀</th>
<th>☢</th>
<th>≦</th>
<th>≿</th>
<th>☵</th>
<th>☯</th>
<th>☒</th>
<th>☛</th>
<th>☛</th>
</tr>
</thead>
</table>

**Unicode 7.0 (June 2014) encodes 113,021 characters.**
UTF-32

Every character is given a 32-bit number

UTF-32 is very simple.
  Each number is stored in 4 bytes.

C standard library type:
  \texttt{32-bit wchar\_t}

Not all combinations are allowed.
  \[2^{21} = 2,097,152\] possible characters
  (\textit{Only 21 bits are actually needed for the code points.})

There is a waste of memory
  11 bits (out of 32) are never used.
  Many of the characters are very rare.

\textit{A better (variable-length) encoding is desirable.}

  UTF-8 \textit{Requires one byte for most common characters}
  UTF-16 \textit{Requires two bytes for most other characters}
UTF-8

A variable-length, byte-based encoding

Preserves ASCII transparency.
   All of the ASCII characters (0..127) are unchanged.
   *ASCII text is also UTF-8 text.*

All other characters are encoded with **multibyte sequences**.

See next slide…

The first byte indicates the number of bytes that follow.
   The leading byte is in the range $C0_{16}$ to $FD_{16}$.
   The trailing bytes are in the range $80_{16}$ to $BF_{16}$.
   The byte values $FE_{16}$ and $FF_{16}$ are never used.

UTF-8 is relatively compact for encoding text in European scripts.
   Uses 50% more space than UTF-16 for East Asian text.

Characters up to $7F_{16}$ take one byte
Characters up to $7FF_{16}$ take two bytes
Characters up to $FFFF_{16}$ take three bytes
Other characters take 4-6 bytes
UTF-8
How the bits of the “code point” are encoded
Uses between 1 and 6 bytes per character.

<table>
<thead>
<tr>
<th>Bits of code point</th>
<th>First code point</th>
<th>Last code point</th>
<th>Bytes in sequence</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
<th>Byte 5</th>
<th>Byte 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>U+0000</td>
<td>U+007F</td>
<td>1</td>
<td>0xxxxxx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>U+0080</td>
<td>U+07FF</td>
<td>2</td>
<td>110xxxx</td>
<td>10xxxxx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>U+0800</td>
<td>U+FFFF</td>
<td>3</td>
<td>1110xxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>U+10000</td>
<td>U+1FFFFF</td>
<td>4</td>
<td>11110xx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>U+200000</td>
<td>U+3FFFFFF</td>
<td>5</td>
<td>111110x</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>U+4000000</td>
<td>U+7FFFFFFF</td>
<td>6</td>
<td>1111110x</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
</tr>
</tbody>
</table>

Credit: Wikipedia / Ken Thompson

ASCII characters are 0-127; start with a zero.
ASCII characters are encoded without any change
UTF-16

This is a commonly used encoding; good for all languages.

Can encode code points \(00000000_{16}\) through \(0010FFFF_{16}\)

The first 1,114,112 code points.

Most common characters are in the range of 0 to \(FFFF_{16}\).

These are encoded exactly as-is.

A text file is a sequence of 16-bits numbers.

\[
0000 \quad 0100 \quad 0001 \quad 0110
\]

These characters are encoded with two 16-bit numbers:

\(10000_{16}\) to \(10FFFF_{16}\)

Character values \(D800_{16}\) to \(DFFF_{16}\) are set aside for the encoding mechanism

(These values will never be assigned to actual characters)

Subtract \(10000_{16}\) to get a number \(00000_{16}\) to \(FFFFF_{16}\) (20 bits)

The first 2 bytes must be in the range \(D800_{16}\) to \(DBFF_{16}\)

The second 2 bytes must be in the range \(DC00_{16}\) to \(DFFF_{16}\).
UTF-16
Uses 16-bit (2-byte) number units.
   Endian-ness is now a problem!

Big Endian is assumed.

The text file may begin with this character:
   \textbf{0xFEFF}
   Called the “Byte Order Mark” (BOM)
   This is invisible
       A “zero-width, non-breaking space”

The character \textbf{0xFFF} is invalid and reserved. It should never be used.

If the software reads \textbf{0xFFF} as the first character…
   It should flip the bytes for all remaining 16-bit numbers.
Glyphs vs. Characters

“We need to distinguish between characters and glyphs. A character is the smallest semantic unit in a writing system. It is an abstract concept such as the letter A or the exclamation point. A glyph is the visual presentation of one or more characters, and is often dependent on adjacent characters.

There is not always a one-to-one mapping between characters and glyphs. In many languages (Arabic is an example), the way a character looks depends heavily on the surrounding characters. Standard printed Arabic has as many as four different printed representations (glyphs) for every letter of the alphabet. In many languages, two or more letters may combine together into a single glyph (called a ligature), or a single character might be displayed with more than one glyph.”

http://userguide.icu-project.org/unicode#TOC-Overview-of-UTF-16