CIS 2107
Chapter 2 Notes
Part 2
Representing and Manipulating Information
negative numbers?

How do we represent negative numbers?
representing negative numbers

• Sign and magnitude
• One’s complement
• Two’s complement
• Biased encoding
simple idea: sign and magnitude

• one bit for the sign
• everything else exactly the same

• So if this is $1_{10}$:
  – 0000 0000 0000 0000 0000 0000 0000 0001₂

• $-1_{10}$ is:
  – 1000 0000 0000 0000 0000 0000 0000 0001₂
sign and magnitude

• $n$ bit numbers, can represent
  – $-2^{n-1}$ to $2^{n-1}$

• Some problems:
  – Two zeros:
    • $0_{10}=0x00000000$ AND
    • $0_{10}=0x80000000$
  – Complicated hardware
another idea: back to our odometer
odometer math

\[
\begin{array}{cccccc}
  & 9 & 9 & 9 & 9 & 9 \\
+ & 2 & \hline
\end{array}
\]
odometer math

\[
\begin{array}{cccccc}
9 & 9 & 9 & 9 & 9 & 9 \\
+ & & & & & 2 \\
\hline
1/ & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccc}
9 & 9 & 9 & 9 & 9 & 9 \\
- & 9 & 9 & 9 & 9 & 8 \\
\hline
0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]
another example

\[
\begin{array}{ccccccc}
9 & 9 & 9 & 9 & 9 & 9 \\
+ & 2 & 0 & 0 & 0 & 0 \\
\hline
2 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]
another example

\[
\begin{array}{cccccc}
9 & 9 & 9 & 9 & 9 & 9 \\
+ & 2 & 0 & 0 & 0 & 0 \\
\hline
1/ & 0 & 1 & 9 & 9 & 9
\end{array}
\]
another example

\[
\begin{array}{ccccccc}
9 & 9 & 9 & 9 & 9 & 9 & 9 \\
+ & 2 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 1 & 9 & 9 & 9 & 9 & 9
\end{array}
\]

\[
\begin{array}{ccccccc}
9 & 9 & 9 & 9 & 9 & 9 & 9 \\
- & 9 & 8 & 0 & 0 & 0 & 0 \\
\hline
0 & 1 & 9 & 9 & 9 & 9 & 9
\end{array}
\]
odometer subtraction?

• (Isn’t that illegal?)

• What if we wanted to do subtraction all along?
  – subtracting 99,998 is the same as adding 2
  – subtracting 98,000 is the same as adding 2,000

• In odometer math, to subtract, what can we add?
9’s complement

• decimal 9’s complement of a number:
  – subtract number from all-9’s (same width)
• example: 9’s complement of 31,692

\[
\begin{align*}
99,999 &- 31,692 \\
68,307 &
\end{align*}
\]
10’s complement

• Simple. 9’s complement + 1.
• 10’s complement of 31,692:

\[
\begin{array}{c}
99,999 \\
- 31,692 \\
\hline
68,307 \\
+ 1 \\
\hline
68,308
\end{array}
\]
10’s complement. So What?

• In odometer math, adding 68,308 is the same as subtracting 31,692. Example:

\[
\begin{align*}
50,000 & - 31,692 \\
& \quad \quad 18,308
\end{align*}
\]

\[
\begin{align*}
50,000 & + 68,308 \\
& \quad \quad 118,308
\end{align*}
\]
Why this works

• losing the carry: same as subtracting 100,000

\[
= 50,000 - 31,692 \\
= 50,000 + (99,999 - 31,692 + 1) - 100,000 \\
= 50,000 + (99,999 - 31,692 + 1) - 100,000
\]
one’s complement

• subtract from an equal-width string of all 1’s.
• example: 1’s complement of $11000110_2$

\[
\begin{array}{c}
1111111_2 \\
\hline
11000110_2 \\
\hline
00111001_2
\end{array}
\]

• another way of getting one’s complement?
one’s complement

• subtract from an equal-width string of all 1’s.
• example: 1’s complement of $11000110_2$

\[
\begin{array}{c}
11111111_2 \\
- \quad 11000110_2 \\
\hline
00111001_2
\end{array}
\]

• another way of getting one’s complement?
  – bitwise NOT
one’s complement arithmetic

• Addition
  – add the carry bit back in

• Subtraction
  – add the complement

• notice: like signed magnitude, two zeros
two’s complement

• two’s complement = one’s complement + 1
two’s complement example

- $6_{10} = 0110_2$
- $-6_{10}?$

- take one’s complement (i.e., flip bits), add 1
  - $\text{OC}(0110_2) = 1001_2$
  - $1001_2 + 1 = 1010_2$
  - So $-6_{10} = 1010_2$
TC(TC(x))=x?

• We’ve said:
  – $6_{10}=0110_2$
  – $-6_{10} = 1010_2$
• TC(TC(0110_2))=0110_2?
• TC(1010_2)
  – take one’s complement
    • $0101_2$
  – add one: $0110_2$
$6_{10} = 0110_2; -6_{10} = 1010_2$. Check.

• So what’s $6 + (-6)$?

\[
\begin{array}{c}
0110_2 \\
+ 1010_2 \\
\hline
10000_2
\end{array}
\]
TC in C

• How could we take the two’s complement of a number in C?
TC in C

• How could we take the two’s complement of a number in C?

```c
int TC(int x) {
    ??????
}
```
TC in C

• How could we take the two’s complement of a number in C? *Easy.*

```c
int TC(int x) {
    return -x;
}
```
TC in C

• How could we take the two’s complement of a number in C? *Easy.*

```
int TC(int x) {
    return -x;
}
```

• But what if we said that you couldn’t use the negation operator?
TC in C without negation operator

- Take the one’s complement (i.e., flip bits)
- Add 1
- How do we do this in C?
TC in C without negation operator

• Take the one’s complement (i.e., flip bits)
• Add 1
• How do we do this in C?

```c
int TC(int x) {
    return ~x + 1;
}
```
unsigned numbers
signed numbers
know the key places

-1

smallest negative

highest positive
key places for 16 bit words

<table>
<thead>
<tr>
<th></th>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>0xFFFF</td>
<td>0b1111111111111111</td>
</tr>
<tr>
<td>TCMx</td>
<td>0x7FFF</td>
<td>0b0111111111111111</td>
</tr>
<tr>
<td>TCMn</td>
<td>0x8000</td>
<td>0b1000000000000000</td>
</tr>
<tr>
<td>0</td>
<td>0x0000</td>
<td>0b0000000000000000</td>
</tr>
<tr>
<td>-1</td>
<td>0xFFFF</td>
<td>0b1111111111111111</td>
</tr>
</tbody>
</table>
# key places table

<table>
<thead>
<tr>
<th>word size</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>0xFF</td>
<td>0xFFFF</td>
<td>0xFFFFFFFF</td>
<td>0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>255₁₀</td>
<td>65, 535₁₀</td>
<td>2³² − 1</td>
<td>2⁶⁴ − 1</td>
</tr>
<tr>
<td>TCMax</td>
<td>0x7F</td>
<td>0x7FFF</td>
<td>0x7FFFFFFFF</td>
<td>0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF</td>
</tr>
<tr>
<td>TCMin</td>
<td>0x80</td>
<td>0x8000</td>
<td>0x80000000</td>
<td>0x80000000000000000000000000000000</td>
</tr>
<tr>
<td>0</td>
<td>0x00</td>
<td>0x0000</td>
<td>0x00000000</td>
<td>0x0000000000000000000000000000000000</td>
</tr>
<tr>
<td>-1</td>
<td>0xFF</td>
<td>0xFFFF</td>
<td>0xFFFFFFFF</td>
<td>0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF</td>
</tr>
</tbody>
</table>

- know these in binary
unsigned numbers

- another way of describing unsigned nums

\[ B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \]
What’s B2U(1010)?

\[ B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \]

\[ B2U(1010_2) = (1)(8) + (0)(4) + (1)(2) + (0)(1) = 10_{10} \]
same for two’s complement

\[ B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \]
What’s TC(1010)?

\[ B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \]

\[ B2T(1010) = (1)(-8) + (0)(4) + (1)(2) + (0)(1) \]
\[ = -6 \]
Bias Encoding. Bias = $-2^{\text{width}-1}-1$

- # zeros?
- # positives?
- order vs TC order
<table>
<thead>
<tr>
<th>binary</th>
<th>unsigned</th>
<th>TC</th>
<th>OC</th>
<th>SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</td>
<td>-7</td>
<td>-0</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-7</td>
<td>-6</td>
<td>-1</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
<td>-5</td>
<td>-2</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
<td>-4</td>
<td>-3</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
<td>-2</td>
<td>-5</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
<td>-1</td>
<td>-6</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
<td>-0</td>
<td>-7</td>
</tr>
</tbody>
</table>
C integer types

• char, short int, int, long int (C99 long long int)
• unsigned versions of each

• C spec doesn’t say, but most implementations TC
How big are each?

<table>
<thead>
<tr>
<th>C data type</th>
<th>typical 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

Is this sort of thing an issue in Java?
constants

integers
• 123 (32-bit int)
• 123L (32-bit long int)
• 123u (32-bit unsigned int),
• 123uL (32-bit unsigned long)
• 123LL (64-bit signed long long)

octal and hex
• 0100 (100 octal = 64 decimal)
• 0x100 (100 hex = 256 decimal)
• 0xf000 (15 unsigned long).

floating point
• 12.3 (32-bit float),
• 123e-1 (32-bit float)
• 12.3f (32-bit float),
• 12.3L (64-bit long double)

'x' = character constant (0 to 127).
• In ASCII ' ' = '\040' = 'x20' = 32
• In ASCII '0' = '\060' = 'x30' = 48
• In ASCII 'A' = '\101' = 'x41' = 65
• In ASCII 'a' = '\141' = 'x61' = 97
## What range can be stored?

<table>
<thead>
<tr>
<th>width</th>
<th>signed?</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>unsigned</td>
<td>0 to $2^8 - 1$</td>
</tr>
<tr>
<td></td>
<td>signed</td>
<td>$-2^7$ to $2^7 - 1$</td>
</tr>
<tr>
<td>16 bits</td>
<td>unsigned</td>
<td>0 to $2^{16} - 1$</td>
</tr>
<tr>
<td></td>
<td>signed</td>
<td>$-2^{15}$ to $2^{15} - 1$</td>
</tr>
<tr>
<td>32 bits</td>
<td>unsigned</td>
<td>0 to $2^{32} - 1$</td>
</tr>
<tr>
<td></td>
<td>signed</td>
<td>$-2^{31}$ to $2^{31} - 1$</td>
</tr>
<tr>
<td>64 bits</td>
<td>unsigned</td>
<td>0 to $2^{64} - 1$</td>
</tr>
<tr>
<td></td>
<td>signed</td>
<td>$-2^{63}$ to $2^{63} - 1$</td>
</tr>
</tbody>
</table>
/* Number of bits in a ‘char’. */
#define CHAR_BIT 8

/* Minimum and maximum values a ‘signed char’ can hold. */
#define SCHAR_MIN (-128)
#define SCHAR_MAX 127

/* Maximum value an ‘unsigned char’ can hold. (Minimum is 0.) */
#define UCHAR_MAX 255

/* Minimum and maximum values a ‘signed short int’ can hold. */
#define SHRT_MIN (-32768)
#define SHRT_MAX 32767

/* Maximum value an ‘unsigned short int’ can hold. (Minimum is 0.) */
#define USHRT_MAX 65535

/* Minimum and maximum values a ‘signed int’ can hold. */
#define INT_MIN (-INT_MAX - 1)
#define INT_MAX 2147483647

/* Maximum value an ‘unsigned int’ can hold. (Minimum is 0.) */
#define UINT_MAX 4294967295U

/* Minimum and maximum values a ‘signed long int’ can hold. */
#define LONG_MAX 2147483647L
#define LONG_MIN (-LONG_MAX - 1L)
C99 `<stdint.h>`

- for all widths \( W \) that the machine supports
  - exact width types: `intW_t`, `uintW_t`
    - *e.g.*, `int8_t`, `uint32_t`.
  - minimum width types: `int_leastW_t`, `uint_leastW_t`
  - `<limits.h>`-style macros for these types:
    - *e.g.* `INTW_MIN`, `UINTW_MAX`, ...
take a look at some of these with gdb

```c
int p1 = 37;
unsigned int p2 = 37;
int n1 = -37;
```

• remember to compile:
  – with the –g switch
  – don’t turn on optimization (no –O)

• in gdb, to print binary, use p/t

• confirm you get the expected when you try:
  – p/d (~n1+1)
What happens?

```cpp
int x=-1;
unsigned int ux = (unsigned int) x;
```
What happens?

```c
int x=-1;
unsigned int ux = (unsigned int) x;
```

(gdb) p/t x
$1 = 11111111111111111111111111111111
(gdb) p/d x
$2 = -1
(gdb) p/t ux
$3 = 11111111111111111111111111111111
(gdb) p/u ux
$4 = 4294967295
casting from signed to unsigned

leading 1

what happens if we cast a negative to an equal width unsigned?
<table>
<thead>
<tr>
<th>bits</th>
<th>signed</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000₂</td>
<td>00₁₀</td>
<td>00₁₀</td>
</tr>
<tr>
<td>0001₂</td>
<td>01₁₀</td>
<td>01₁₀</td>
</tr>
<tr>
<td>0010₂</td>
<td>02₁₀</td>
<td>02₁₀</td>
</tr>
<tr>
<td>0011₂</td>
<td>03₁₀</td>
<td>03₁₀</td>
</tr>
<tr>
<td>0100₂</td>
<td>04₁₀</td>
<td>04₁₀</td>
</tr>
<tr>
<td>0101₂</td>
<td>05₁₀</td>
<td>05₁₀</td>
</tr>
<tr>
<td>0110₂</td>
<td>06₁₀</td>
<td>06₁₀</td>
</tr>
<tr>
<td>0111₂</td>
<td>07₁₀</td>
<td>07₁₀</td>
</tr>
<tr>
<td>1000₂</td>
<td>−08₁₀</td>
<td>08₁₀</td>
</tr>
<tr>
<td>1001₂</td>
<td>−07₁₀</td>
<td>09₁₀</td>
</tr>
<tr>
<td>1010₂</td>
<td>−06₁₀</td>
<td>10₁₀</td>
</tr>
<tr>
<td>1011₂</td>
<td>−05₁₀</td>
<td>11₁₀</td>
</tr>
<tr>
<td>1100₂</td>
<td>−04₁₀</td>
<td>12₁₀</td>
</tr>
<tr>
<td>1101₂</td>
<td>−03₁₀</td>
<td>13₁₀</td>
</tr>
<tr>
<td>1110₂</td>
<td>−02₁₀</td>
<td>14₁₀</td>
</tr>
<tr>
<td>1111₂</td>
<td>−01₁₀</td>
<td>15₁₀</td>
</tr>
<tr>
<td>bits</td>
<td>signed</td>
<td>unsigned</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>0000₂</td>
<td>00₁₀</td>
<td>00₁₀</td>
</tr>
<tr>
<td>0001₂</td>
<td>01₁₀</td>
<td>01₁₀</td>
</tr>
<tr>
<td>0010₂</td>
<td>02₁₀</td>
<td>02₁₀</td>
</tr>
<tr>
<td>0011₂</td>
<td>03₁₀</td>
<td>03₁₀</td>
</tr>
<tr>
<td>0100₂</td>
<td>04₁₀</td>
<td>04₁₀</td>
</tr>
<tr>
<td>0101₂</td>
<td>05₁₀</td>
<td>05₁₀</td>
</tr>
<tr>
<td>0110₂</td>
<td>06₁₀</td>
<td>06₁₀</td>
</tr>
<tr>
<td>0111₂</td>
<td>07₁₀</td>
<td>07₁₀</td>
</tr>
<tr>
<td>1000₂</td>
<td>−08₁₀</td>
<td>08₁₀</td>
</tr>
<tr>
<td>1001₂</td>
<td>−07₁₀</td>
<td>09₁₀</td>
</tr>
<tr>
<td>1010₂</td>
<td>−06₁₀</td>
<td>10₁₀</td>
</tr>
<tr>
<td>1011₂</td>
<td>−05₁₀</td>
<td>11₁₀</td>
</tr>
<tr>
<td>1100₂</td>
<td>−04₁₀</td>
<td>12₁₀</td>
</tr>
<tr>
<td>1101₂</td>
<td>−03₁₀</td>
<td>13₁₀</td>
</tr>
<tr>
<td>1110₂</td>
<td>−02₁₀</td>
<td>14₁₀</td>
</tr>
<tr>
<td>1111₂</td>
<td>−01₁₀</td>
<td>15₁₀</td>
</tr>
</tbody>
</table>

signed: same as unsigned + 16
what if we cast a large unsigned positive to an equal width signed?
bit shifting for unsigned numbers

• >>, <<, >>=, <<=

• numbers fall off the end

• fill in with 0s

• math equivalent (when 1s haven’t fallen off)?
shifting left

• $x \ll j$
  
  – shift $x$ to the left $j$ bit positions
  
  – fill with 0s from the right
  
  – numbers “fall off the left end”

```c
char x=11, j;

for (j=0; j<8; j++)
    x<<=1;
```

<table>
<thead>
<tr>
<th></th>
<th>$x_{10}$</th>
<th>$x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>0b00001011</td>
</tr>
<tr>
<td>1</td>
<td>22</td>
<td>0b00010110</td>
</tr>
<tr>
<td>2</td>
<td>44</td>
<td>0b00101100</td>
</tr>
<tr>
<td>3</td>
<td>88</td>
<td>0b01011000</td>
</tr>
<tr>
<td>4</td>
<td>176</td>
<td>0b10110000</td>
</tr>
<tr>
<td>5</td>
<td>96</td>
<td>0b11000000</td>
</tr>
<tr>
<td>6</td>
<td>192</td>
<td>0b110000000</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>0b100000000</td>
</tr>
</tbody>
</table>
shifting to the right

• unsigned:
  – same as left shift

• what about signed?
  – implementation dependent
  – some fill from LHS
    • with 0
    • with sign bit
  – (why do this with signed numbers anyway?)

```c
int main(void)
{
    int i;
    char c1 = 64, c2 = -64;
    for (i=8; i>0; i--)
    {
        c1 >>= 1;
        c2 >>= 1;
    }
    return 0;
}
```
(aside) java right shift

• Java defines two:
  – >> fills from the left with the sign bit
  – >>> fills from the left with 0s
public class JavaShift {
    public static void main(String args[]) {
        int x = Integer.MIN_VALUE;  // i.e. -2**(31)
        for (int i=0; i<32; i++) {
            int ds = x >> i;
            int ts = x >>> i;
            System.out.println(x + " >> " + i + " = " +
                                Integer.toBinaryString(ds));
            System.out.println(x + " >>> " + i + " = " +
                                Integer.toBinaryString(ts));
        }
    }
}
java right shifts output

-2147483648  >>  0 = 10000000000000000000000000000000
-2147483648  >>> 0 = 10000000000000000000000000000000

-2147483648  >>  1 = 11000000000000000000000000000000
-2147483648  >>> 1 = 10000000000000000000000000000000

-2147483648  >>  2 = 11100000000000000000000000000000
-2147483648  >>> 2 = 10000000000000000000000000000000

-2147483648  >>  3 = 11110000000000000000000000000000
-2147483648  >>> 3 = 10000000000000000000000000000000

-2147483648  >>  4 = 11111000000000000000000000000000
-2147483648  >>> 4 = 10000000000000000000000000000000

-2147483648  >>  5 = 11111100000000000000000000000000
-2147483648  >>> 5 = 10000000000000000000000000000000
casting to different widths

• smaller to larger
  – no problem

• larger to smaller
  – information loss?
small to large

• What happens?
  – long x = char c
unsigned: zero extension

• unsigned numbers
  – going from small to large → zero extension
• Example: unsigned long x = unsigned char c
• What happens?
signed: sign extension

• signed numbers
  – going from small to large → sign extension
• Example: long \( x = \text{char } c \)
• What happens?
  – if \( c > 0 \)?
  – if \( c < 0 \)?
C conversion rules

• Details in K&R Appendix A
• C rules. Three types:
  – Integer promotion
  – Integer conversion rank
  – Usual arithmetic conversions
Mixing Signed and Unsigned

• What happens if we’re performing an operation:
  – One operand is signed
  – Other operand unsigned
Mixing Signed and Unsigned

#include <stdio.h>

int main(int argc, char **argv) {
    int s=-1;
    unsigned int u=0;

    if (s > u) {
        printf("%d is greater than %u\n", s, u);
    } else {
        printf("%d isn't greater than %u\n", s, u);
    }

    return 0;
}
Mixing Signed and Unsigned

#include <stdio.h>

int main(int argc, char **argv) {
    int s=-1;
    unsigned int u=0;

    if (s > u) {
        printf("%d is greater than %u\n", s, u);
    } else {
        printf("%d isn't greater than %u\n", s, u);
    }

    return 0;
}

Result:
-1 is greater than 0
Mixing Signed and Unsigned. Rules.

• Convert the signed operand to unsigned
  – Seems opposite of what you’d expect
• Bit string stays the same
• **BUT** we interpret it differently
• Could mean difference of $2^{\text{width}}$
Back to our example

#include <stdio.h>

int main(int argc, char **argv) {
    int s=-1;
    unsigned int u=0;

    if (s > u) {
        printf("%d is greater than %u\n", s, u);
    } else {
        printf("%d isn't greater than %u\n", s, u);
    }

    return 0;
}

Result:
-1 is greater than 0
C Conversion Rules: Integer Promotion

• Integer types smaller than int promoted to int.
• example:
  – char result, a=100, b=10, c=20
casting large to small. truncation

    int x1 = 0x00001234;
    int x2 = 0x12345678;
    short s1 = (short)x1;
    short s2 = (short)x2;
    printf("x1=0x%08x, ", x1);
    printf("x2=0x%08x, ", x2);
    printf("s1=0x%08x, ", s1);
    printf("s2=0x%08x\n", s2);

we get:

    x1=0x00001234, x2=0x12345678, s1=0x00001234, s2=0x00005678
truncation of unsigned numbers

- unsigned int x
- truncating to $k$ bits equivalent to $x \mod 2^k$
integer arithmetic

• “odometer” effect
• modular arithmetic
• unsigned addition: addition modulo width
• for others, see details in BO
unsigned addition

• addition modulo the width
Floating Point
#include <stdio.h>

int main(int argc, char **argv) {
    float f=0.1;
    if (f==0.1)
        printf("It’s 0.1\n");
    else
        printf("It’s not 0.1\n");
    return 0;
}
What happens here?

```c
#include <stdio.h>

int main(int argc, char **argv) {
    float f=0.1;
    if (f==0.1)
        printf("It’s 0.1\n");
    else
        printf("It’s not 0.1\n");
    return 0;
}
```

```
bash-3.2$ gcc -Wall -o StrangeFloat02 StrangeFloat02.c
bash-3.2$ ./StrangeFloat02
It’s not 0.1
```
What about here?

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    float f1=0.1,
        f2=0.2,
        sum;

    sum=f1+f2;
    printf("%.8f+%.8f=%.8f\n", f1, f2, sum);
    return 0;
}
```
What about here?

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    float f1=0.1,
         f2=0.2,
         sum;

    sum=f1+f2;
    printf("%.8f+%.8f=%.8f\n", f1, f2, sum);
    return 0;
}
```

bash-3.2$ gcc -Wall -o StrangeFloat StrangeFloat.c
bash-3.2$ ./StrangeFloat
0.10000000+0.20000000=0.300000001
Strange on 32-bit machines

```c
#include <stdio.h>

int main(void) {
    float a = 30000000;
    float b = 3;
    float c;

    c = a + b - a;
    printf("%f\n", c);
    c = a + b;
    c = c - a;
    printf("%f\n", c);

    return 0;
}
```
Strange on 32-bit machines

```c
#include <stdio.h>

int main(void) {
    float a = 30000000;
    float b = 3;
    float c;
    c = a + b - a;
    printf("%f\n", c);
    c = a + b;
    c = c - a;
    printf("%f\n", c);
    return 0;
}
```

Output:

3.000000
4.000000
```c
#include <stdio.h>

int main(int argc, char **argv) {
    int x = 2147483647; /* largest signed 32-bit integer */
    float f = x;

    printf("x=%d, f=%f\n", x, f);
    printf("f-64=%f\n", (f-64));
    printf("f-65=%f\n", (f-65));

    return 0;
}
```

More Strange Behavior
More Strange Behavior

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int x = 2147483647; /* largest signed 32-bit integer */
    float f = x;

    printf("x=%d, f=%f\n", x, f);
    printf("f-64=%f\n", (f-64));
    printf("f-65=%f\n", (f-65));

    return 0;
}
```

Output:

```
x=2147483647, f=2147483648.0
```
More Strange Behavior

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int x = 2147483647; /* largest signed 32-bit integer */
    float f = x;

    printf("x=%d, f=%f\n", x, f);
    printf("f-64=%f\n", (f-64));
    printf("f-65=%f\n", (f-65));

    return 0;
}
```

**Output:**

```
x=2147483647, f=2147483648.0
f-64=2147483648.000000
```
#include <stdio.h>

int main(int argc, char **argv) {
    int x = 2147483647; /* largest signed 32-bit integer */
    float f = x;

    printf("x=%d, f=%f\n", x, f);
    printf("f-64=%f\n", (f-64));
    printf("f-65=%f\n", (f-65));

    return 0;
}

Output:

x=2147483647, f=2147483648.0
f-64=2147483648.000000
f-65=2147483520.000000
Recall Binary Representation of ints

$$11011001_2 =$$

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

$$= (1)(128) + (1)(64) + (0)(32) + (1)(16) + (1)(8) + (0)(4) + (0)(2) + (1)(1)$$
$$= 128 + 64 + 16 + 8 + 1$$
$$= 217$$
What about fractions?

.11011001₂ =

<table>
<thead>
<tr>
<th>.</th>
<th>2⁻¹</th>
<th>2⁻²</th>
<th>2⁻³</th>
<th>2⁻⁴</th>
<th>2⁻⁵</th>
<th>2⁻⁶</th>
<th>2⁻⁷</th>
<th>2⁻⁸</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

= 2⁻¹ + 2⁻² + 2⁻⁴ + 2⁻⁵ + 2⁻⁸

= 0.5 + 0.25 + 0.0625 + 0.03125 + 0.00390625

= 0.84765625
some powers of $2 \leq 1$

<table>
<thead>
<tr>
<th>$2^0$</th>
<th>1.000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-1}$</td>
<td>0.500000</td>
</tr>
<tr>
<td>$2^{-2}$</td>
<td>0.250000</td>
</tr>
<tr>
<td>$2^{-3}$</td>
<td>0.125000</td>
</tr>
<tr>
<td>$2^{-4}$</td>
<td>0.062500</td>
</tr>
<tr>
<td>$2^{-5}$</td>
<td>0.031250</td>
</tr>
<tr>
<td>$2^{-6}$</td>
<td>0.015625</td>
</tr>
<tr>
<td>$2^{-7}$</td>
<td>0.0078125</td>
</tr>
<tr>
<td>$2^{-8}$</td>
<td>0.00390625</td>
</tr>
</tbody>
</table>
Decimal to binary?

• How do we convert the representation of:
  – a fraction in decimal
  – to a fraction in binary
recall how we did numbers > 1

• Convert $119_{10}$ to binary

  \[
  119 = 59 \times 2 + 1 \\
  59 = 29 \times 2 + 1 \\
  29 = 14 \times 2 + 1 \\
  14 = 7 \times 2 + 0 \\
  7 = 3 \times 2 + 1 \\
  3 = 1 \times 2 + 1 \\
  1 = 0 \times 2 + 1
  \]

• solution: $1110111_2$
converting fractions

1 while (fraction part != 0) {
2     multiply number by 2
3     record the integer part for later
4     subtract the integer part
5 }

6

7 recorded integer parts are the binary rep
example. $0.6953125_{10}$?

$0.6953125_{10} \times 2 = 1.390625_{10}$

$0.390625_{10} \times 2 = 0.78125_{10}$

$0.78125_{10} \times 2 = 1.5625_{10}$

$0.5625_{10} \times 2 = 1.125_{10}$

$0.125_{10} \times 2 = 0.25_{10}$

$0.25_{10} \times 2 = 0.5_{10}$

$0.5_{10} \times 2 = 1.0_{10}$

solution: $0.6953125_{10}=0.1011001_{2}$
double check result

• $0.1011001_2 = 0.6953125_{10}$

\[
0.1011001_2 = 2^{-1} + 2^{-3} + 2^{-4} + 2^{-7} \\
= 0.5_{10} + 0.125_{10} + 0.0625_{10} + 0.0078125_{10}
\]

\[
\begin{array}{c}
0.5_{10} \\
0.125_{10} \\
0.0625_{10} \\
+ 0.0078125_{10} \\
\hline
0.6953125_{10}
\end{array}
\]
Does it always work?

• Remember the pigeonhole?
  – infinite number of floating-point numbers
  – storing in register of finite size
Does it always work? Repeating

- We have repeating decimal fractions
- We also have repeating binary fractions.
  - \( 0.1_{10} = 0.0001100110011_2 \ldots \)
  - \( 0.2_{10} = 0.001100110011_2 \ldots \)
- The more places we have, the closer we are to the value we’re trying to represent
representing $0.2_{10}$

<table>
<thead>
<tr>
<th>base 2 float</th>
<th>base 10 frac</th>
<th>base 10 float</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0₂</td>
<td>0₈</td>
<td>0₈</td>
</tr>
<tr>
<td>0.00₂</td>
<td>0₁₀</td>
<td>0₁₀</td>
</tr>
<tr>
<td>0.001₂</td>
<td>1/₈₀</td>
<td>0.12₅₀</td>
</tr>
<tr>
<td>0.0011₂</td>
<td>3/₁₆₀</td>
<td>0.1₈₇₅₀</td>
</tr>
<tr>
<td>0.0011₀₂</td>
<td>3/₁₆₀</td>
<td>0.1₈₇₅₀</td>
</tr>
<tr>
<td>0.0011₀₀₂</td>
<td>3/₁₆₀</td>
<td>0.1₈₇₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₂</td>
<td>25/₁₂₈₀</td>
<td>0.1₉₅₃₁₂₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₁₂</td>
<td>5₁/₂₅₆₀</td>
<td>0.₁₉₉₂₁₈₇₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₁₀₂</td>
<td>5₁/₂₅₆₀</td>
<td>0.₁₉₉₂₁₈₇₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₁₀₀₂</td>
<td>5₁/₂₅₆₀</td>
<td>0.₁₉₉₂₁₈₇₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₁₀₀₁₂</td>
<td>4₀₉/₂₀₄₈₀</td>
<td>0.₁₉₉₇₀₇₀₃₁₂₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₁₀₀₁₁₂</td>
<td>₈₁₉/₄₀₉₆₀</td>
<td>0.₁₉₉₉₅₁₁₇₁₈₇₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₁₀₀₁₁₀₂</td>
<td>₈₁₉/₄₀₉₆₀</td>
<td>0.₁₉₉₉₅₁₁₇₁₈₇₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₁₀₀₁₁₀₀₂</td>
<td>₈₁₉/₄₀₉₆₀</td>
<td>0.₁₉₉₉₅₁₁₇₁₈₇₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₁₀₀₁₁₀₀₁₂</td>
<td>₆₅₅₃/₃₂₇₆₈₀</td>
<td>0.₁₉₉₉₈₁₆₈₉₄₅₃₁₂₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₁₀₀₁₁₀₀₁₁₂</td>
<td>₁₃₁₀₇/₆₅₅₃₆₀</td>
<td>0.₁₉₉₉₉₆₉₄₈₂₄₂₁₈₇₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₁₀₀₁₁₀₀₁₁₀₂</td>
<td>₁₃₁₀₇/₆₅₅₃₆₀</td>
<td>0.₁₉₉₉₉₆₉₄₈₂₄₂₁₈₇₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₁₀₀₁₁₀₀₁₁₀₀₂</td>
<td>₁₃₁₀₇/₆₅₅₃₆₀</td>
<td>0.₁₉₉₉₉₆₉₄₈₂₄₂₁₈₇₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₁₀₀₁₁₀₀₁₁₀₀₁₂</td>
<td>₁₀₄₈₅₇/₅₂₄₂₈₈₀</td>
<td>0.₁₉₉₉₉₈₈₅₅₅₉₀₈₂₀₃₁₂₅₀</td>
</tr>
<tr>
<td>0.00110₀₁₁₀₀₁₁₀₀₁₁₀₀₁₁₂</td>
<td>₂₀₉₇₁₅/₁₀₄₈₅₇₆₀</td>
<td>0.₁₉₉₉₉₉₈₀₉₂₆₅₁₃₆₇₁₈₇₅₀</td>
</tr>
</tbody>
</table>
representing $0.1_{10}$

<table>
<thead>
<tr>
<th>base 2 float</th>
<th>base 10 frac</th>
<th>base 10 float</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.002</td>
<td>$0_{10}$</td>
<td>$0_{10}$</td>
</tr>
<tr>
<td>0.0002</td>
<td>$0_{10}$</td>
<td>$0_{10}$</td>
</tr>
<tr>
<td>0.00012</td>
<td>$1/16_{10}$</td>
<td>$0.0625_{10}$</td>
</tr>
<tr>
<td>0.000112</td>
<td>$3/32_{10}$</td>
<td>$0.09375_{10}$</td>
</tr>
<tr>
<td>0.0001102</td>
<td>$3/32_{10}$</td>
<td>$0.09375_{10}$</td>
</tr>
<tr>
<td>0.00011002</td>
<td>$3/32_{10}$</td>
<td>$0.09375_{10}$</td>
</tr>
<tr>
<td>0.000110012</td>
<td>$25/256_{10}$</td>
<td>$0.09765625_{10}$</td>
</tr>
<tr>
<td>0.0001100112</td>
<td>$51/512_{10}$</td>
<td>$0.099609375_{10}$</td>
</tr>
<tr>
<td>0.00011001102</td>
<td>$51/512_{10}$</td>
<td>$0.099609375_{10}$</td>
</tr>
<tr>
<td>0.000110011002</td>
<td>$51/512_{10}$</td>
<td>$0.099609375_{10}$</td>
</tr>
<tr>
<td>0.0001100110012</td>
<td>$409/4096_{10}$</td>
<td>$0.099853515625_{10}$</td>
</tr>
<tr>
<td>0.00011001100112</td>
<td>$819/8192_{10}$</td>
<td>$0.0999755859375_{10}$</td>
</tr>
<tr>
<td>0.000110011001102</td>
<td>$819/8192_{10}$</td>
<td>$0.0999755859375_{10}$</td>
</tr>
<tr>
<td>0.0001100110011002</td>
<td>$819/8192_{10}$</td>
<td>$0.0999755859375_{10}$</td>
</tr>
<tr>
<td>0.00011001100110012</td>
<td>$6553/65536_{10}$</td>
<td>$0.0999908447265625_{10}$</td>
</tr>
<tr>
<td>0.000110011001100112</td>
<td>$13107/131072_{10}$</td>
<td>$0.09999847412109375_{10}$</td>
</tr>
<tr>
<td>0.0001100110011001102</td>
<td>$13107/131072_{10}$</td>
<td>$0.09999847412109375_{10}$</td>
</tr>
<tr>
<td>0.00011001100110011002</td>
<td>$13107/131072_{10}$</td>
<td>$0.09999847412109375_{10}$</td>
</tr>
<tr>
<td>0.000110011001100110012</td>
<td>$104857/1048576_{10}$</td>
<td>$0.09999942779541015625_{10}$</td>
</tr>
<tr>
<td>0.0001100110011001100112</td>
<td>$209715/2097152_{10}$</td>
<td>$0.099999904632568359375_{10}$</td>
</tr>
</tbody>
</table>
can’t represent everything

• with finite length binary strings, can only approximate numbers that can’t be written as:

\[(x)(2^y)\]

• (remember that y can be positive or negative)
Other “interesting” numbers

<table>
<thead>
<tr>
<th>base 2 float</th>
<th>base 10 frac</th>
<th>base 10 float</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1₂</td>
<td>1/2₁₀</td>
<td>0.5₁₀</td>
</tr>
<tr>
<td>0.11₂</td>
<td>3/4₁₀</td>
<td>0.75₁₀</td>
</tr>
<tr>
<td>0.11₁₁₂</td>
<td>7/8₁₀</td>
<td>0.875₁₀</td>
</tr>
<tr>
<td>0.11₁₁₁₁₂</td>
<td>15/16₁₀</td>
<td>0.937₅₁₀</td>
</tr>
<tr>
<td>0.11₁₁₁₁₁₂</td>
<td>31/32₁₀</td>
<td>0.9687₅₁₀</td>
</tr>
<tr>
<td>0.11₁₁₁₁₁₁₂</td>
<td>63/6₄₁₀</td>
<td>0.9843₇₅₁₀</td>
</tr>
<tr>
<td>0.11₁₁₁₁₁₁₁₂</td>
<td>12₇/₁₂₈₁₀</td>
<td>0.9921₈₇₅₁₀</td>
</tr>
<tr>
<td>0.11₁₁₁₁₁₁₁₁₂</td>
<td>2₅₅/2₅₆₁₀</td>
<td>0.9₉₆₀₉₃₇₅₁₀</td>
</tr>
<tr>
<td>0.11₁₁₁₁₁₁₁₁₁₂</td>
<td>5₁₁/₅₁₂₁₀</td>
<td>0.₉₉₈₀₄₆₈₇₅₁₀</td>
</tr>
<tr>
<td>0.11₁₁₁₁₁₁₁₁₁₁₂</td>
<td>1₀₂₃/1₀₂₄₁₀</td>
<td>0.₉₉₉₀₂₃₄₃₇₅₁₀</td>
</tr>
</tbody>
</table>
machine representation of floats

• represent as:

\[ \pm m \times b^e \]

- \( m \): mantissa
- \( b \): base
- \( e \): exponent

• but, base is always 2
machine representation of floats

• Recall: $1234567.0_{10}$ can be written:
  - $123456.7 \times 10$
  - $12345.67 \times 10^2$
  - $1234.567 \times 10^3$
  - $123.4567 \times 10^4$
  - $12.34567 \times 10^5$
  - $1.234567 \times 10^6$
machine representation of floats

• $110010.0010_2$ can be written:
  – $110010.0010_2 \times 2^0$
  – $11001.00010_2 \times 2^1$
  – $1100.100010_2 \times 2^2$
  – $110.0100010_2 \times 2^3$
  – $11.00100010_2 \times 2^4$
  – $1.100100010_2 \times 2^5$

• Shift until radix after first 1 called *normalizing*
machine representation of floats

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent</th>
<th>mantissa</th>
</tr>
</thead>
</table>

### floating point representation

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent</th>
<th>mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 bits</td>
<td>23 bits</td>
</tr>
<tr>
<td>11 bits</td>
<td>52 bits</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C type</th>
<th>exponent</th>
<th>mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>8 bits</td>
<td>23 bits</td>
</tr>
<tr>
<td>double</td>
<td>11 bits</td>
<td>52 bits</td>
</tr>
</tbody>
</table>
floating point representation

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent</th>
<th>mantissa</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>C type</th>
<th>exponent</th>
<th>mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>8 bits</td>
<td>23 bits</td>
</tr>
<tr>
<td>double</td>
<td>11 bits</td>
<td>52 bits</td>
</tr>
</tbody>
</table>

On Intel – “extended precision” (depending on compiler)

<table>
<thead>
<tr>
<th>C type</th>
<th>exponent</th>
<th>mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>long double</td>
<td>15 bits</td>
<td>64 bits</td>
</tr>
</tbody>
</table>
How?

• How are floating-point numbers represented?
• There are three cases.

--- let’s do the most common case first
The common case: *Normalized*

- Set the **sign bit**. 0 for positive, 1 for negative
- Write num in fixed-pt binary
- Normalize (radix pt is just to the right of the first 1)
- **mantissa**: the values to the right of radix point
- what’s stored in **exponent** field?
  - the actual exponent + *bias value*
bias value = $2^{\text{exponent field width}-1-1}$

<table>
<thead>
<tr>
<th>C data type</th>
<th>bits exponent field</th>
<th>bias value</th>
<th>exponent range</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>8</td>
<td>$2^{8-1}-1=127$</td>
<td>-126 to 127</td>
</tr>
<tr>
<td>double</td>
<td>11</td>
<td>$2^{11-1}-1=1023$</td>
<td>-1022 to 1023</td>
</tr>
</tbody>
</table>
Why bias?

• We’d like to store a value which could be positive or negative
• We’re storing it in a single bit string.
• We’ve done this before:
  – Two’s complement
  – One’s complement
  – Sign and magnitude
• Idea is the same: *biased encoding*
The exponent. Recall Two’s Complement
Bias Encoding. Bias = \(2^{\text{width}-1}-1\)
Bias Encoding. Bias = $2^{\text{width}-1}-1$

- # zeros?
- # positives?
- order vs TC order
- Spin the two’s complement wheel *almost*
Bias Encoding. Bias = \(2^{\text{width}-1}-1\)

Diagram showing the mapping of binary codes to integer values with a note indicating 'no flip' for certain codes.
How? Example -15.375

• Set the sign bit. 0 for positive, 1 for negative
  – sign bit 1
• Write num in fixed-pt binary:
  – 15.375 = 1111.011₂
• Normalize (so radix point is just to the right of the first 1)
  – 1.111011₂ * 2³
• m is the values to the right of radix point
  – 111011
• calculate e: exponent+127
  – exponent was 3₁₀ (11₂).
  – 3₁₀+127₁₀ = 130₁₀ or 1000 0010₂
• Final result:
  
  1  1000 0010  111 0110 0000 0000 0000 0000
Another example

• -118.625
Another example

• -118.625
double check: try in gdb

-118.625? \[\begin{array}{c} 1 \\ 1000 \ 0101 \end{array} \ \begin{array}{c} 110 \ 1101 \ 0100 \ 0000 \ 0000 \ 0000 \end{array} \]

• Check.

• In a program where we have:
  - `float fl`
  - `(gdb) set fl=-118.625`
  - `(gdb) x/t &fl`
  - `0x7fff5fbff68c:`
    1100001011101101010000000000000000
How?

• How are floating-point numbers represented?
• There are three cases:
  – *Normalized* values (what we just did)
  – *De-normalized* values
    • Numbers “close” to 0.0
    • Exponent field is all zeros
  – “Special” cases
    • +/- ∞, NaN
    • Exponent field is all 1s.
Visualizing Floating-point Range
How?

• How are floating-point numbers represented?
• There are three cases:
  – *Normalized* values (what we just did)
  – *De-normalized values*
    • Numbers “close” to 0.0
    • Exponent field is all zeros
  – “Special” cases
    • +/- $\infty$, NaN
    • Exponent field is all 1s.
De-Normalized Values

• Why?
  – Used to represent values “close” to 0.0

• Exponent field – all 0s.
  – Fraction represented has exponent of 1-Bias

• Mantissa field
  – we don’t assume a leading 1

• Sign field
  – As usual, can be 1 or 0.
  – Means that we can also have +0.0 or -0.0
Example

• We’ll use 7-bit floats, consisting of:
  – A sign bit
  – 3 bits for the exponent
  – 3 bits for the mantissa

• What is: 0 101 000, where:
  – 0 is the sign bit
  – 101 are the bits for the mantissa
  – 000 are the bits for the exponent
Example: 0 101 000

• We’ll use 7-bit floats, consisting of:
  – A sign bit
  – 3 bits for the exponent
  – 3 bits for the mantissa

• What is: 0 101 000, where:
  – 0 is the sign bit
  – 101 are the bits for the mantissa
  – 000 are the bits for the exponent

• Bias is $2^{(3-1)}-1 = 2^2-1 = 3$
• Exponent is 1-bias = -2
• Mantissa:
  – $\frac{1}{2}+0/4+1/8 = 5/8$
• Final result:
  – Mantissa * $2^{\text{exponent}}=$
  – $5/8*2^{-2}=$
  – $5/8*1/4=$
  – $5/32=$
  – 0.15625
Another de-normalized example

• 0 110 000 (110 is the mantissa)
Another de-normalized example

- 0 110 000 (110 is the mantissa)
- Bias is \(2^{(3-1)}-1 = 2^2-1 = 3\)
- Exponent is 1-bias = -2
- \(M = \frac{1}{2} + \frac{1}{4} + 0/8 = \frac{3}{4}\)
- Final result = \(M \times 2^{\text{exponent}} = \)
  - \(\frac{3}{4} \times 2^{-2} = \)
  - \(3/4 \times 1/4 = \)
  - \(3/16 = 0.1875\)
How are floating-point numbers represented?

• There are three cases:
  – *Normalized* values (what we just did)
  – *De-normalized* values
    • Numbers “close” to 0.0
    • Exponent field is all zeros
  – “*Special*” cases
    • +/- ∞, NaN
    • Exponent field is all 1s.
“Special” Cases

• The exponent field is all 1s.
• If the fraction field is all 0s:
  – +/- infinity
  – Can use +/- infinity when:
    • overflow has occurred
    • divide by 0.
• Otherwise:
  – NaN, *e.g.* sqrt(-1)
"Special" Cases

Special Cases

+infinity

-infinity

NaN

Not all 0s
“Simple” examples

• 32-, 64-, or 80-bit widths: tough to see
• keep it simple for now: 8-bit widths.
  – 1 bit for sign
  – 4 bits for the exponent
  – 3 bits for the fraction
Bias

• With a 4-bit exponent, what will be the bias?
Bias

• With a 4-bit exponent, what will be the bias?
• $\text{Bias} = 2^{\text{width}-1} - 1 = 2^{4-1} - 1 = 2^3 - 1 = 7$
Table of “Simple” Values

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>000</td>
<td>-6</td>
<td>0</td>
<td>zero</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>001</td>
<td>-6</td>
<td>1/8 * 1/64 = 1/512</td>
<td>closest to zero</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>010</td>
<td>-6</td>
<td>2/8 * 1/64 = 2/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>110</td>
<td>-6</td>
<td>6/8 * 1/64 = 6/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>111</td>
<td>-6</td>
<td>7/8 * 1/64 = 7/512</td>
<td>largest denormalized</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>000</td>
<td>-6</td>
<td>8/8 * 1/64 = 8/512</td>
<td>smallest norm (rem. leading 1)</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>001</td>
<td>-6</td>
<td>9/8 * 1/64 = 9/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>110</td>
<td>-1</td>
<td>14/8 * 1/2 = 14/16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>111</td>
<td>-1</td>
<td>15/8 * 1/2 = 15/16</td>
<td>closest to 1 from below</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>000</td>
<td>0</td>
<td>8/8 * 1 = 1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>001</td>
<td>0</td>
<td>9/8 * 1 = 9/8</td>
<td>closest to 1 from above</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>010</td>
<td>0</td>
<td>10/8 * 1 = 10/8</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>110</td>
<td>7</td>
<td>14/8 * 128 = 224</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>111</td>
<td>7</td>
<td>15/8 * 128 = 240</td>
<td>largest normalized</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>000</td>
<td>n/a</td>
<td>infinity</td>
<td></td>
</tr>
</tbody>
</table>

- Please convince yourselves that these make sense
Table of “Simple” Values

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>000</td>
<td>-6</td>
<td>0</td>
<td>zero</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>001</td>
<td>-6</td>
<td>1/8 * 1/64 = 1/512</td>
<td>closest to zero</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>010</td>
<td>-6</td>
<td>2/8 * 1/64 = 2/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>110</td>
<td>-6</td>
<td>6/8 * 1/64 = 6/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>111</td>
<td>-6</td>
<td>7/8 * 1/64 = 7/512</td>
<td>largest denormalized</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>000</td>
<td>-6</td>
<td>8/8 * 1/64 = 8/512</td>
<td>smallest norm (rem. leading 1)</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>001</td>
<td>-6</td>
<td>9/8 * 1/64 = 9/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>110</td>
<td>-1</td>
<td>14/8 * 1/2 = 14/16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>111</td>
<td>-1</td>
<td>15/8 * 1/2 = 15/16</td>
<td>closest to 1 from below</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>000</td>
<td>0</td>
<td>8/8 * 1 = 1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>001</td>
<td>0</td>
<td>9/8 * 1 = 9/8</td>
<td>closest to 1 from above</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>010</td>
<td>0</td>
<td>10/8 * 1 = 10/8</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>110</td>
<td>7</td>
<td>14/8 * 128 = 224</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>111</td>
<td>7</td>
<td>15/8 * 128 = 240</td>
<td>largest normalized</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>000</td>
<td>n/a</td>
<td>infinity</td>
<td></td>
</tr>
</tbody>
</table>

- Please convince yourselves that these make sense
### Table of “Simple” Values

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>000</td>
<td>-6</td>
<td>0</td>
<td>zero</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>001</td>
<td>-6</td>
<td>$1/8 \times 1/64 = 1/512$</td>
<td>closest to zero</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>010</td>
<td>-6</td>
<td>$2/8 \times 1/64 = 2/512$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>110</td>
<td>-6</td>
<td>$6/8 \times 1/64 = 6/512$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>111</td>
<td>-6</td>
<td>$7/8 \times 1/64 = 7/512$</td>
<td>largest denormalized</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>000</td>
<td>-6</td>
<td>$8/8 \times 1/64 = 8/512$</td>
<td>smallest norm (rem. leading 1)</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>001</td>
<td>-6</td>
<td>$9/8 \times 1/64 = 9/512$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>110</td>
<td>-1</td>
<td>$14/8 \times 1/2 = 14/16$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>111</td>
<td>-1</td>
<td>$15/8 \times 1/2 = 15/16$</td>
<td>closest to 1 from below</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>000</td>
<td>0</td>
<td>$8/8 \times 1 = 1$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>001</td>
<td>0</td>
<td>$9/8 \times 1 = 9/8$</td>
<td>closest to 1 from above</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>010</td>
<td>0</td>
<td>$10/8 \times 1 = 10/8$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>110</td>
<td>7</td>
<td>$14/8 \times 128 = 224$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>111</td>
<td>7</td>
<td>$15/8 \times 128 = 240$</td>
<td>largest normalized</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>000</td>
<td>n/a</td>
<td>n/a</td>
<td>infinity</td>
</tr>
</tbody>
</table>

- Please convince yourselves that these make sense
Normalized

<table>
<thead>
<tr>
<th>s</th>
<th>not all 0s or all 1s</th>
<th>anything</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>exponent</td>
<td>mantissa</td>
</tr>
</tbody>
</table>

De-normalized

<table>
<thead>
<tr>
<th>s</th>
<th>0 0 0 0 0 0 0 0 0</th>
<th>anything</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>exponent</td>
<td>mantissa</td>
</tr>
</tbody>
</table>

Special Cases

+infinity

<table>
<thead>
<tr>
<th></th>
<th>0 1 1 1 1 1 1 1</th>
<th>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>exponent</td>
<td>mantissa</td>
</tr>
</tbody>
</table>

-infinity

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

NaN

<table>
<thead>
<tr>
<th>s</th>
<th>1 1 1 1 1 1 1 1</th>
<th>not all 0s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>exponent</td>
<td>mantissa</td>
</tr>
</tbody>
</table>
Floating-point cheat sheet

- **e == 0**
  - No? normalized
  - Yes? denormalized

- **e all 1s**
  - No? 
  - Yes?

- **f = (-1)^s * 2^{e-bias} * 1.M**
  - Note: E=e-bias

- **M == 0?**
  - No? 
  - Yes?

- **NaN**

- **S == 0?**
  - No? 
  - Yes?

- **-∞**
- **+∞**
“important” numbers

<table>
<thead>
<tr>
<th>description</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>00...00</td>
<td>00...00</td>
</tr>
<tr>
<td>smallest de-normalized</td>
<td>00...00</td>
<td>00...01</td>
</tr>
<tr>
<td>largest de-normalized</td>
<td>00...00</td>
<td>11...11</td>
</tr>
<tr>
<td>smallest normalized</td>
<td>00...01</td>
<td>00...00</td>
</tr>
<tr>
<td>one</td>
<td>01...11</td>
<td>00...00</td>
</tr>
<tr>
<td>largest normalized</td>
<td>11...10</td>
<td>11...11</td>
</tr>
</tbody>
</table>
## Rounding

<table>
<thead>
<tr>
<th>Method</th>
<th>$1.40</th>
<th>$1.60</th>
<th>$1.50</th>
<th>$2.50</th>
<th>$-1.50</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>towards zero</strong></td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>-$1</td>
</tr>
<tr>
<td><strong>round down</strong> (-(\infty))</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>-$2</td>
</tr>
<tr>
<td><strong>round up</strong> (+(\infty))</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>$3</td>
<td>-$1</td>
</tr>
<tr>
<td><strong>round to even</strong></td>
<td>$1</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>-$2</td>
</tr>
</tbody>
</table>
### Rounding

<table>
<thead>
<tr>
<th>Method</th>
<th>$1.40</th>
<th>$1.60</th>
<th>$1.50</th>
<th>$2.50</th>
<th>-$1.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>towards zero</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>-$1</td>
</tr>
<tr>
<td>round down (-∞)</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>-$2</td>
</tr>
<tr>
<td>round up (+∞)</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>$3</td>
<td>-$1</td>
</tr>
<tr>
<td>round to even</td>
<td>$1</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>-$2</td>
</tr>
</tbody>
</table>

- Don’t be confused by round to even:
  - Round to the closest
  - when you’re half-way between two possibilities, choose the even one
Why Round to Even?

- Rounding in same direction → skew?
- Round to even:
  - sometimes up
  - sometimes down
addition and multiplication

- commutative: yes
  \[ f_1 + f_2 = f_2 + f_1 \]

- associative: no
  \[ f_1 + (f_2 + f_3) \text{ might not equal } (f_1 + f_2) + f_3 \]
  \[ f_1 \times (f_2 \times f_3) \text{ might not equal } (f_1 \times f_2) \times f_3 \]

- distributive: no
  \[ f_1 \times (f_2 + f_3) \text{ might not equal } (f_1 \times f_2) + (f_1 \times f_3) \]
The moral of the story ... back to our first example. Why false?

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    float f=0.1;

    if (f==0.1)
        printf("It’s 0.1\n");
    else
        printf("It’s not 0.1\n");

    return 0;
}
```
The moral of the story ... back to our first example. Why false?

```
#include <stdio.h>

int main(int argc, char **argv)
{
    float f=0.1;
    if (f==0.1)
        printf("It’s 0.1\n");
    else
        printf("It’s not 0.1\n");
    return 0;
}
```

- Can we store $0.1_{10}$ without rounding?
- Will there be less round error if we use a double?
- What happens if the compiler uses a double for the 0.1 in line 7? Will the values be the same?
These give us what we expect.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    float f = 0.1;
    if (f==0.1f) {
        printf("It’s 0.1\n");
    } else {
        printf("It’s not 0.1\n");
    }
    return 0;
}
```
## Floats in C. Casting rules.

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int → float</code></td>
<td>no overflow</td>
</tr>
<tr>
<td></td>
<td>possible rounding</td>
</tr>
<tr>
<td><code>(int or float) → double</code></td>
<td>no problem.</td>
</tr>
<tr>
<td></td>
<td>double’s range is higher</td>
</tr>
<tr>
<td><code>double → float</code></td>
<td>possible overflow to +/- infinity</td>
</tr>
<tr>
<td></td>
<td>possible rounding</td>
</tr>
<tr>
<td><code>(float or double) → int</code></td>
<td>round towards zero</td>
</tr>
<tr>
<td></td>
<td><em>e.g.</em>, <code>1.9999 → 1</code></td>
</tr>
<tr>
<td></td>
<td><code>−1.999 → −1</code></td>
</tr>
<tr>
<td></td>
<td>overflow possible</td>
</tr>
<tr>
<td></td>
<td>C standard doesn’t specify result</td>
</tr>
</tbody>
</table>