Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

get_mpg:
  pushq %rbp
  movq %rsp, %rbp
  ...
  popq %rbp
  ret

0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
Integers

- Representation of integers: unsigned and signed
- Casting
- Arithmetic and shifting
- Sign extension
But before we get to integers....

- Encode a standard deck of playing cards.
- 52 cards in 4 suits
  - How do we encode suits, face cards?
- What operations do we want to make easy to implement?
  - Which is the higher value card?
  - Are they the same suit?
Two possible representations

- 52 cards – 52 bits with bit corresponding to card set to 1

  low-order 52 bits of 64-bit word

- “One-hot” encoding
- Drawbacks:
  - Hard to compare values and suits
  - Large number of bits required
Two possible representations

- 52 cards – 52 bits with bit corresponding to card set to 1
  - “One-hot” encoding
  - Drawbacks:
    - Hard to compare values and suits
    - Large number of bits required

- 4 bits for suit, 13 bits for card value – 17 bits with two set to 1
  - Pair of one-hot encoded values
  - Easier to compare suits and values
    - Still an excessive number of bits

Can we do better?
Two better representations

- Binary encoding of all 52 cards – only 6 bits needed

  - Fits in one byte
  - Smaller than one-hot encodings.
  - How can we make value and suit comparisons easier?
Two better representations

- **Binary encoding of all 52 cards – only 6 bits needed**
  - Fits in one byte
  - Smaller than one-hot encodings.
  - How can we make value and suit comparisons easier?

- **Binary encoding of suit (2 bits) and value (4 bits) separately**
  - Also fits in one byte, and easy to do comparisons
**Compare Card Suits**

```c
#define SUIT_MASK 0x30

int sameSuitP(char card1, char card2) {
    return (!(card1 & SUIT_MASK) ^ (card2 & SUIT_MASK));
    //return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

- **SUIT_MASK** is defined as `0x30` or `00110000` in binary, indicating the suit part of a card.
- The function `sameSuitP` compares two cards for the same suit by using a mask to isolate the suit portion and then checking if the suit bits in both cards are the same.

---

`mask`: a bit vector that, when bitwise ANDed with another bit vector `v`, turns all **but** the bits of interest in `v` to 0.
#define SUIT_MASK 0x30

int sameSuitP(char card1, char card2) {
    return !(((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK)));
    //return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}

SUIT_MASK = 0x30 = 0 0 1 1 0 0 0 0

mask: a bit vector that, when bitwise ANDed with another bit vector v, turns all but the bits of interest in v to 0.

char hand[5];
char card1, card2;

card1 = hand[0];
card2 = hand[1];

... if ( sameSuitP(card1, card2) ) { ... }
Compare Card **Values**

```c
#define VALUE_MASK 0x0F

int greaterValue(char card1, char card2) {
    return ((unsigned int)(card1 & VALUE_MASK) >
            (unsigned int)(card2 & VALUE_MASK));
}
```

**VALUE_MASK = 0x0F =**

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

**suit** | **value**

char hand[5];       // represents a 5-card hand
char card1, card2;  // two cards to compare
card1 = hand[0];
card2 = hand[1];
...

if ( greaterValue(card1, card2) ) { ... }

**mask:** a bit vector that, when bitwise ANDed with another bit vector \( \nu \), turns all *but* the bits of interest in \( \nu \) to 0.
Encoding Integers

- The hardware (and C) supports two flavors of integers:
  - `unsigned` – only the non-negatives
  - `signed` – both negatives and non-negatives

- There are only $2^W$ distinct bit patterns of $W$ bits, so...
  - Cannot represent all the integers
  - **Unsigned values:** $0 \ldots 2^W-1$
  - **Signed values:** $-2^{W-1} \ldots 2^{W-1}-1$

- Reminder: terminology for binary representations
  - “Most-significant” or “high-order” bit(s)
  - “Least-significant” or “low-order” bit(s)
Unsigned Integers

- Unsigned values are just what you expect
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + b_52^5 + \ldots + b_12^1 + b_02^0$
  - Useful formula: $1+2+4+8+\ldots+2^{N-1} = 2^N - 1$

- Add and subtract using the normal “carry” and “borrow” rules, just in binary.
  
  \[
  \begin{array}{c}
  \text{00111111} \\
  +00001000 \\
  \hline
  01000111
  \end{array}
  \quad
  63 + 8 = 71
  \]

- How would you make signed integers?
Signed Integers: Sign-and-Magnitude

- Let's do the natural thing for the positives
  - They correspond to the unsigned integers of the same value
    - Example (8 bits): 0x00 = 0, 0x01 = 1, ..., 0x7F = 127

- But, we need to let about half of them be negative
  - Use the high-order bit to indicate negative: call it the “sign bit”
    - Call this a “sign-and-magnitude” representation
  - Examples (8 bits):
    - 0x00 = 00000000₂ is non-negative, because the sign bit is 0
    - 0x7F = 01111111₂ is non-negative
    - 0x85 = 10000101₂ is negative
    - 0x80 = 10000000₂ is negative...
Signed Integers: Sign-and-Magnitude

How should we represent -1 in binary?

- $10000001_2$
  Use the MSB for + or -, and the other bits to give magnitude.
Sign-and-Magnitude Negatives

How should we represent -1 in binary?

- $10000001_2$
  Use the MSB for + or -, and the other bits to give magnitude.
  (Unfortunate side effect: there are two representations of 0!)
How should we represent -1 in binary?

- **10000001<sub>2</sub>**
  - Use the MSB for + or -, and the other bits to give magnitude.
  - (Unfortunate side effect: there are two representations of 0!)

- Another problem: **arithmetic is cumbersome.**
  - Example: 4 - 3 != 4 + (-3)

```
+ 0
+ 1
+ 2
+ 3
+ 4
+ 5
+ 6
+ 7
- 1
- 2
- 3
- 4
- 5
- 6
- 7
```

How do we solve these problems?
Two’s Complement Negatives

How should we represent -1 in binary?
Two’s Complement Negatives

- How should we represent -1 in binary?

Rather than a sign bit, let MSB have same value, but **negative weight**.

\[ b_{w-1} = 1 \text{ adds } -2^{w-1} \text{ to the value.} \]

for \( i < w-1 \): \( b_i = 1 \) adds \( +2^i \) to the value.
Two’s Complement Negatives

How should we represent \(-1\) in binary?

Rather than a sign bit, let MSB have same value, but *negative weight*.

\[ b_{w-1} = 1 \text{ adds } -2^{w-1} \text{ to the value. } \]

for \( i < w-1 \): \( b_i = 1 \text{ adds } +2^i \text{ to the value. } \)

\[
\text{e.g. unsigned } 1010_2:\]
\[ 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10_{10} \]

\text{2’s compl. } 1010_2:\]
\[ -1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = -6_{10} \]
Two’s Complement Negatives

- How should we represent -1 in binary?
  
  Rather than a sign bit, let MSB have same value, but **negative weight**.

  \[ b_{w-1} = 1 \text{ adds } -2^{w-1} \text{ to the value.} \]

  \[ \text{for } i < w-1: \quad b_i = 1 \text{ adds } +2^i \text{ to the value.} \]

  e.g. unsigned \( 1010_2 \):
  \[ 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10_{10} \]

  2’s compl. \( 1010_2 \):
  \[ -1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = -6_{10} \]

  -1 is represented as \( 1111_2 = -2^3 + (2^3 - 1) \)

  All negative integers still have MSB = 1.

  - **Advantages:** single zero, simple arithmetic

  - To get negative representation of any integer, take bitwise complement and then add one!

    \[ \sim x + 1 \equiv -x \]
4-bit Unsigned vs. Two’s Complement

\[2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1\]

\[-2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1\]
4-bit Unsigned vs. Two’s Complement

-2^3 x 1 + 2^2 x 0 + 2^1 x 1 + 2^0 x 1

(math) difference = 16 = 2^4
4-bit Unsigned vs. Two’s Complement

1 0 1 1

$2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1$

$-2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1$

(math) difference = 16 = $2^4$
Two’s Complement Arithmetic

- The same addition procedure works for both unsigned and two’s complement integers
  - Simplifies hardware: only one algorithm for addition
  - Algorithm: simple addition, discard the highest carry bit
    - Called “modular” addition: result is sum \( \mod 2^w \)

- Examples:

<table>
<thead>
<tr>
<th></th>
<th>0100</th>
<th>0100</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>+ 3</td>
<td>4</td>
<td>- 4</td>
</tr>
<tr>
<td></td>
<td>0011</td>
<td>- 3</td>
<td>+ 3</td>
</tr>
<tr>
<td>= 7</td>
<td>0111</td>
<td>= 1</td>
<td>= 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>drop carry</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>= 0001</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>
Two’s Complement

Why does it work?

- Put another way, for all positive integers $x$, we want:

  \[
  \text{Bit representation of } x + \text{Bit representation of } -x = 0 \quad \text{(ignoring the carry-out bit)}
  \]

- This turns out to be the **bitwise complement plus one**
  
  - What should the 8-bit representation of -1 be?
    
    \[
    \begin{array}{c}
    \text{00000001} \\
    +?????????? \quad \text{(we want whichever bit string gives the right result)} \\
    \hline
    \text{00000000}
    \end{array}
    \]

    \[
    \begin{array}{c}
    \text{00000010} \\
    +?????????? \quad \text{00000011} \\
    \hline
    \text{00000000}
    \end{array}
    \]
    
    \[
    \begin{array}{c}
    \text{00000000} \quad \text{00000000}
    \end{array}
    \]
Two’s Complement

Why does it work?

- Put another way, for all positive integers x, we want:

  \[
  \text{Bit representation of } x + \text{Bit representation of } -x + 0 \quad \text{(ignoring the carry-out bit)}
  \]

- This turns out to be the \textit{bitwise complement plus one}

  - What should the 8-bit representation of -1 be?
    
    \[
    \begin{array}{c}
    00000001 \\
    +11111111 \\
    \hline
    100000000
    \end{array}
    \]
    
    (we want whichever bit string gives the right result)
Two’s Complement

Why does it work?

- Put another way, for all positive integers $x$, we want:
  
  $\begin{align*}
  &\text{Bit representation of } x \\
  &+ \text{Bit representation of } -x \\
  &= 0 \quad (\text{ignoring the carry-out bit})
  \end{align*}$

- This turns out to be the \textit{bitwise complement plus one}

  - What should the 8-bit representation of -1 be?
    
    \[
    \begin{array}{c}
    00000001 \\
    +11111111 \\
    100000000
    \end{array}
    \quad \text{(we want whichever bit string gives the right result)}
    \]

    \[
    \begin{array}{c}
    00000010 \\
    +11111110 \\
    100000000
    \end{array}
    \quad \begin{array}{c}
    00000011 \\
    +11111101 \\
    100000000
    \end{array}
    \]
Unsigned & Signed Numeric Values

- Signed and unsigned integers have limits.
  - If you compute a number that is too big (positive), it wraps:
    \[ 6 + 4 = ? \quad 15U + 2U = ? \]
  - If you compute a number that is too small (negative), it wraps:
    \[ -7 - 3 = ? \quad 0U - 2U = ? \]

- The CPU may be capable of “throwing an exception” for overflow on signed values.
  - It won't for unsigned.

- But C and Java just cruise along silently when overflow occurs... Oops.
Conversion Visualized

<table>
<thead>
<tr>
<th>Two’s Complement → Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordering Inversion</td>
</tr>
<tr>
<td>Negative → Big Positive</td>
</tr>
</tbody>
</table>

2’s Complement Range

Unsigned Range

TMax

UMax

UMax - 1

TMax + 1

TMax

0

0

MinMax

-2

-1

0

2’s Complement

Range

Ordering Inversion

Negative → Big Positive
Overflow/Wrapping: Unsigned

addition: drop the carry bit

\[
\begin{array}{c}
15 \\
+ 2 \\
\hline
17
\end{array} \quad \begin{array}{c}
1111 \\
+ 0010 \\
\hline
10001
\end{array}
\]

Modular Arithmetic
Overflow/Wrapping: Two’s Complement

addition: drop the carry bit

\[
\begin{array}{c}
\text{-1} \\
+ 2 \\
\hline
\text{1}
\end{array}
\quad
\begin{array}{c}
\text{1111} \\
+ \text{0010} \\
\hline
\text{10001}
\end{array}
\]

\[
\begin{array}{c}
\text{6} \\
+ 3 \\
\hline
\text{9}
\end{array}
\quad
\begin{array}{c}
\text{0110} \\
+ \text{0011} \\
\hline
\text{1001}
\end{array}
\]

Modular Arithmetic
Values To Remember

- **Unsigned Values**
  - $U_{\text{Min}} = 0$
    - 000...0
  - $U_{\text{Max}} = 2^w - 1$
    - 111...1

- **Two’s Complement Values**
  - $T_{\text{Min}} = -2^{w-1}$
    - 100...0
  - $T_{\text{Max}} = 2^{w-1} - 1$
    - 011...1
  - Negative one
    - 111...1 0xF...F

**Values for W = 32**

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{\text{Max}}$</td>
<td>4,294,967,296</td>
<td>FF FF FF FF FF</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Max}}$</td>
<td>2,147,483,647</td>
<td>7F FF FF FF FF</td>
<td>01111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Min}}$</td>
<td>-2,147,483,648</td>
<td>80 00 00 00</td>
<td>10000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF FF FF FF</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 00</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>
Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Use “U” suffix to force unsigned:
    - \( 0U, 4294967259U \)
Signed vs. Unsigned in C

- Casting
  - `int tx, ty;`
  - `unsigned ux, uy;`
  - **Explicit** casting between signed & unsigned:
    - `tx = (int) ux;`
    - `uy = (unsigned) ty;`
  - **Implicit** casting also occurs via assignments and function calls:
    - `tx = ux;`
    - `uy = ty;`
    - The gcc flag `-Wsign-conversion` produces warnings for implicit casts, but `-Wall` does not!
- How does casting between signed and unsigned work?
- What values are going to be produced?
Signed vs. Unsigned in C

Casting

- `int tx, ty;`
- `unsigned ux, uy;`

- Explicit casting between signed & unsigned:
  - `tx = (int) ux;`
  - `uy = (unsigned) ty;`

- Implicit casting also occurs via assignments and function calls:
  - `tx = ux;`
  - `uy = ty;`
  - The gcc flag `-Wsign-conversion` produces warnings for implicit casts, but `-Wall` does not!

- How does casting between signed and unsigned work?
- What values are going to be produced?

- *Bits are unchanged,* just interpreted differently!
Casting Surprises

Expression Evaluation

- If you mix unsigned and signed in a single expression, then *signed values are implicitly cast to unsigned*.
- Including comparison operations `<`, `>`, `==`, `<=`, `>=`
- Examples for $W = 32$: $T_{MIN} = -2,147,483,648$    $T_{MAX} = 2,147,483,647$

<table>
<thead>
<tr>
<th>Constant$_1$</th>
<th>Constant$_2$</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>==</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483648</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483648</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
Casting Surprises

- If you mix unsigned and signed in a single expression, then *signed values are implicitly cast to unsigned* (The bit pattern does not change, bits are just interpreted differently.)
- Examples for $W = 32$:

Reminder: $T\text{MIN} = -2,147,483,648$ $T\text{MAX} = 2,147,483,647$

<table>
<thead>
<tr>
<th>Constant$_1$</th>
<th>Constant$_2$</th>
<th>Relation</th>
<th>Interpret the bits as:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>0U 0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>==</td>
<td>Unsigned</td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>0 0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>&lt;</td>
<td>Signed</td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>0U 0000 0000 0000 0000 0000 0000 0000 0000</td>
<td>&gt;</td>
<td>Unsigned</td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>-2147483648 1000 0000 0000 0000 0000 0000 0000 0000</td>
<td>&gt;</td>
<td>Signed</td>
</tr>
<tr>
<td>2147483647U 0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>-2147483648 1000 0000 0000 0000 0000 0000 0000 0000</td>
<td>&lt;</td>
<td>Unsigned</td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>-2 1111 1111 1111 1111 1111 1111 1111 1110</td>
<td>&gt;</td>
<td>Signed</td>
</tr>
<tr>
<td>(unsigned) -1 1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>-2 1111 1111 1111 1111 1111 1111 1111 1110</td>
<td>&gt;</td>
<td>Unsigned</td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>2147483648U 1000 0000 0000 0000 0000 0000 0000 0000</td>
<td>&lt;</td>
<td>Unsigned</td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111 1111</td>
<td>(int) 2147483648U 1000 0000 0000 0000 0000 0000 0000 0000</td>
<td>&gt;</td>
<td>Signed</td>
</tr>
</tbody>
</table>
Sign Extension

What happens if you convert a 32-bit signed integer to a 64-bit signed integer?
Sign Extension

**Task:**
- Given \( w \)-bit signed integer \( x \)
- Convert it to \( w+k \)-bit integer *with same value*

**Rule:**
- Make \( k \) copies of sign bit:
- \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \)

\( k \) copies of MSB
Sign Extension

0 0 1 0  
0 0 0 0 0 0 1 0 
1 1 0 0 
? ? ? ? ? ? 1 1 0 0 

4-bit 2 
8-bit 2 
4-bit -4 
8-bit -4
Sign Extension

Just adding zeroes to the front does not work
Sign Extension

4-bit 2: 0 0 1 0
8-bit 2: 0 0 0 0 0 0 1 0
4-bit -4: 1 1 0 0
8-bit -116: 1 0 0 0 1 1 0 0

Just making the first bit=1 also does not work
Sign Extension

0 0 1 0 4-bit 2
0 0 0 0 0 0 1 0 8-bit 2
1 1 0 0 4-bit -4
1 1 1 1 1 1 0 0 8-bit -4

Need to extend the sign bit to all “new” locations
Sign Extension Example

- Converting from smaller to larger integer data type
- C automatically performs sign extension (Java too)

```
short int x = 12345;
int ix = (int) x;
short int y = -12345;
int iy = (int) y;
```

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>12345</td>
<td>01100000 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>12345</td>
<td>00000000 00000000 01100000 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-12345</td>
<td>11001111 11000111</td>
</tr>
<tr>
<td>iy</td>
<td>-12345</td>
<td>11111111 11111111 11001111 11000111</td>
</tr>
</tbody>
</table>
Shift Operations

- **Left shift:** \( x \ll n \)
  - Shift bit vector \( x \) left by \( n \) positions
  - Throw away extra bits on left
  - Fill with 0s on right

- **Right shift:** \( x \gg n \)
  - Shift bit-vector \( x \) right by \( n \) positions
  - Throw away extra bits on right
  - **Logical shift** (for unsigned values)
    - Fill with 0s on left
  - **Arithmetic shift** (for signed values)
    - Replicate most significant bit on left
    - Maintains sign of \( x \)

\[ x \gg 9 ? \]

The behavior of \( \gg \) in C depends on the compiler! It is *arithmetic* shift right in GCC. In Java: \( >>> \) is logical shift right; \( \gg \) is arithmetic shift right.
Shift Operations

- **Left shift:** \( x \ll n \)
  - Shift bit vector \( x \) left by \( n \) positions
    - Throw away extra bits on left
    - Fill with 0s on right

- **Right shift:** \( x \gg n \)
  - Shift bit-vector \( x \) right by \( n \) positions
    - Throw away extra bits on right
  - **Logical shift** (for unsigned values)
    - Fill with 0s on left
  - **Arithmetic shift** (for signed values)
    - Replicate most significant bit on left
    - Maintains sign of \( x \)

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>00100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Logical: ( x \gg 2 )</td>
<td>00001000</td>
</tr>
<tr>
<td>Arithmetic: ( x \gg 2 )</td>
<td>00001000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Logical: ( x \gg 2 )</td>
<td>00101000</td>
</tr>
<tr>
<td>Arithmetic: ( x \gg 2 )</td>
<td>11101000</td>
</tr>
</tbody>
</table>

\( x \gg 9 \)?

Shifts by \( n < 0 \) or \( n \geq \) size of \( x \) are undefined

- **e.g.** if \( x \) is a 32-bit int, shifts by \( \geq 32 \) bits are undefined.

The behavior of \( >> \) in C depends on the compiler! It is *arithmetic* shift right in GCC.

In Java: \( >>> \) is logical shift right; \( >> \) is arithmetic shift right.
What happens when...

- $x >> n$?
- $x << n$?
What happens when...

- \( x >> n \): divide by \( 2^n \)
- \( x << n \): multiply by \( 2^n \)

Shifting is faster than general multiply or divide operations
Shifting and Arithmetic Example #2

signed

x = -101;

1 0 0 1 1 0 1 1

y = x << 2;

1 1 1 0 1 1 0

y == 108

overflow

\[ \frac{x}{2^n} \]

arithmetically shift right:
shift in copies of most significant bit from the left

\[ x \times 2^n \]

x*2^n

logical shift left:
shift in zeros from the right

\[ \frac{x}{2^n} \]

rounding (down)

signed

x = -19;

y = x >> 2;

y == -5

Shifts by n < 0 or n >= size of x are undefined

Integers & Floats
Using Shifts and Masks

- Extract the 2nd most significant byte of an integer?

<table>
<thead>
<tr>
<th>x</th>
<th>01100001</th>
<th>01100010</th>
<th>01100011</th>
<th>01100100</th>
</tr>
</thead>
</table>

Winter 2016

Integers & Floats
Using Shifts and Masks

- Extract the 2nd most significant byte of an integer:
  - First shift, then mask: \(( x \gg 16 ) \& 0xFF\)

<table>
<thead>
<tr>
<th></th>
<th>01100001</th>
<th>01100010</th>
<th>01100111</th>
<th>01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x \gg 16</td>
<td>00000000</td>
<td>00000000</td>
<td>01100001</td>
<td>01100010</td>
</tr>
<tr>
<td>( x \gg 16) &amp; 0xFF</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>11111111</td>
</tr>
<tr>
<td></td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>01100010</td>
</tr>
</tbody>
</table>

- Extract the sign bit of a signed integer?
Using Shifts and Masks

- Extract the sign bit of a signed integer:
  - \((x >> 31) & 1\) - need the "& 1" to clear out all other bits except LSB

<table>
<thead>
<tr>
<th>(x)</th>
<th>11100001 01100010 01100011 01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x &gt;&gt; 31)</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>((x &gt;&gt; 31) &amp; 0x1)</td>
<td>\textbf{00000000 00000000 00000000 00000001} (\overline{\text{mask}})</td>
</tr>
<tr>
<td></td>
<td>\textbf{00000000 00000000 00000000 00000001} (\overline{\text{result}})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(x)</th>
<th>01100001 01100010 01100011 01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x &gt;&gt; 31)</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>((x &gt;&gt; 31) &amp; 0x1)</td>
<td>\textbf{00000000 00000000 00000000 00000001} (\overline{\text{mask}})</td>
</tr>
<tr>
<td></td>
<td>\textbf{00000000 00000000 00000000 00000000} (\overline{\text{result}})</td>
</tr>
</tbody>
</table>
Using Shifts and Masks

- **Conditionals as Boolean expressions (assuming x is 0 or 1)**
  - In C: if \((x) a=y\) else \(a=z\); which is the same as \(a = x \ ? \ y : z\);
    - If \(x=1\) then \(a=y\), otherwise \(x=0\) and \(a=z\)
  - Can be re-written (assuming arithmetic right shift) as:
    \[a = ( (x << 31) >> 31) \& y ) \mid ( ( \!x) << 31 ) >> 31 ) \& z\);

<table>
<thead>
<tr>
<th>x = 1</th>
<th>00000000 00000000 00000000 00000000 00000000 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x &lt;&lt; 31)</td>
<td>10000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>(((x &lt;&lt; 31)&gt;&gt; 31))</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>(y = 257)</td>
<td>00000000 00000000 00000001 00000001</td>
</tr>
<tr>
<td>((x &lt;&lt; 31) &gt;&gt; 31) &amp; y)</td>
<td>00000000 00000000 00000001 00000001</td>
</tr>
</tbody>
</table>

If \(x=1\), then \(\!x = 0\) and \((\!x) << 31 \) \(\geq 31\) = 00..0; so: (00..0 \& z) = 0. So:
\[a = (00000000 00000000 00000001 00000001) | (00...00)\) (in other words \(a = y\))

If \(x=0\), then \(\!x = 1\) and instead \(a = z\).
One of two sides of the \(\mid\) will always be all zeroes.
Multiplication

- What do you get when you multiply 9 x 9?

- What about $2^{30} \times 3$?

- $2^{30} \times 5$?

- $-2^{31} \times -2^{31}$?
Unsigned Multiplication in C

Operands: $w$ bits

True Product: $2^w$ bits

Discard $w$ bits: $w$ bits

- **Standard Multiplication Function**
  - Ignores high order $w$ bits

- **Implements Modular Arithmetic**
  \[ \text{UMult}_w(u, v) = u \cdot v \mod 2^w \]
**Power-of-2 Multiply with Shift**

- **Operation**
  - \( u << k \) gives \( u \times 2^k \)
  - Both signed and unsigned

  **Operands:** \( w \) bits

  **True Product:** \( w+k \) bits

  **Discard** \( k \) bits: \( w \) bits

- **Examples**
  - \( u << 3 \) \[==\] \( u \times 8 \)
  - \( u << 5 - u << 3 \) \[==\] \( u \times 24 \)

  Most machines shift and add faster than multiply
  - Compiler generates this code automatically
/* Kernel memory region holding user-accessible data */
define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void* user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528
void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf(“%s\n”, mybuf);
}
Malicious Usage

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void* user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
}
```

/* Declaration of library function memcpy */
void* memcpy(void* dest, void* src, size_t n);
Floating point topics

- Background: fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

There are many more details that we won’t cover
- It’s a 58-page standard...
Summary

- As with integers, floats suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like ints
  - Some “simple fractions” have no exact representation (e.g., 0.2)
  - Can also lose precision, unlike ints
    - “Every operation gets a slightly wrong result”

- Mathematically equivalent ways of writing an expression may compute different results
  - Violates associativity/distributivity

- **Never** test floating point values for equality!

- **Careful** when converting between ints and floats!
Fractional Binary Numbers

- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number: \[ \sum_{k=-j}^{i} b_k \cdot 2^k \]
Fractional Binary Numbers

- **Value**
  - 5 and 3/4
  - 2 and 7/8
  - 47/64

- **Representation**
  - 101.11₂
  - 10.111₂
  - 0.101111₂

- **Observations**
  - Shift left = multiply by power of 2
  - Shift right = divide by power of 2
  - Numbers of the form $0.11111...₂$ are just below 1.0
    - $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^i} + ... \rightarrow 1.0$
  - Use notation $1.0 - \varepsilon$
Limits of Representation

Limitations:

- Even given an arbitrary number of bits, can only exactly represent numbers of the form \( x \times 2^y \) (\( y \) can be negative)
- Other rational numbers have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.333333...</td>
</tr>
<tr>
<td></td>
<td>0.01010101[01]...2</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011[0011]</td>
</tr>
<tr>
<td></td>
<td>0.001100110011[0011]</td>
</tr>
<tr>
<td>1/10</td>
<td>0.0001100110011</td>
</tr>
<tr>
<td></td>
<td>0.0001100110011[0011]2</td>
</tr>
</tbody>
</table>
Fixed Point Representation

- **Implied binary point.** Two example schemes:
  
  #1: the binary point is between bits 2 and 3
  \[b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ [. \ b_2 \ b_1 \ b_0\]
  
  #2: the binary point is between bits 4 and 5
  \[b_7 \ b_6 \ b_5 \ [. \ b_4 \ b_3 \ b_2 \ b_1 \ b_0\]

  …..

- Wherever we put the binary point, with fixed point representations there is a trade off between the amount of range and precision we have.

- Fixed point = fixed *range* and fixed *precision*
  
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers

- Hard to pick how much you need of each!
Floating Point

- Analogous to scientific notation
  - In Decimal:
    - Not 12000000, but $1.2 \times 10^7$  In C: 1.2e7
    - Not 0.0000012, but $1.2 \times 10^{-6}$  In C: 1.2e-6
  - In Binary:
    - Not 11000.000, but $1.1 \times 2^4$
    - Not 0.000101, but $1.01 \times 2^{-4}$

- We have to divvy up the bits we have (e.g., 32) among:
  - the sign (1 bit)
  - the significand
  - the exponent
IEEE Floating Point

- **IEEE 754**
  - Established in 1985 as uniform standard for floating point arithmetic
  - Main idea: make numerically sensitive programs portable
  - Specifies two things: representation and result of floating operations
  - now supported by all major CPUs

- **Driven by numerical concerns**
  - Numerical analysts predominated over hardware designers in defining standard
  - Nice standards for rounding, overflow, underflow, but...
  - But... hard to make fast in hardware
  - Float operations can be an order of magnitude slower than integer
Floating Point Representation

- Numerical form:

\[ V_{10} = (-1)^s \times M \times 2^E \]

- Sign bit \( s \) determines whether number is negative or positive
- Significand (mantissa) \( M \) normally a fractional value in range \([1.0, 2.0)\)
- Exponent \( E \) weights value by a (possibly negative) power of two
Floating Point Representation

- **Numerical form:**

  \[ V_{10} = (-1)^s \times M \times 2^E \]

  - Sign bit \( s \) determines whether number is negative or positive
  - Significand (mantissa) \( M \) normally a fractional value in range \([1.0, 2.0)\)
  - Exponent \( E \) weights value by a (possibly negative) power of two

- **Representation in memory:**

  - MSB \( s \) is sign bit \( s \)
  - \( \text{exp} \) field encodes \( E \) (but is not equal to \( E \))
  - \( \text{frac} \) field encodes \( M \) (but is not equal to \( M \))
Precisions

- **Single precision:** 32 bits

  - 1 bit for the sign (s)
  - 8 bits for the exponent (exp)
  - 23 bits for the fraction (frac)

- **Double precision:** 64 bits

  - 1 bit for the sign (s)
  - 11 bits for the exponent (exp)
  - 52 bits for the fraction (frac)

- Finite representation means not all values can be represented exactly. Some will be approximated.
Normalization and Special Values

\[ V = (-1)^s \times M \times 2^E \]

- “Normalized” = \( M \) has the form 1.xxxxx
  - As in scientific notation, but in binary
  - 0.011 \( \times 2^5 \) and 1.1 \( \times 2^3 \) represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it

- How do we represent 0.0? Or special or undefined values like 1.0/0.0?
Normalization and Special Values

\[ V = (-1)^s \times M \times 2^E \]

- **“Normalized”** = \( M \) has the form 1.xxxxxx
  - As in scientific notation, but in binary
  - 0.011 \( \times 2^5 \) and 1.1 \( \times 2^3 \) represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it.

- **Special values:**
  - **zero:** \( s = 0 \) \( \exp = 00...0 \) \( \frac{}{\exp} = 00...0 \)
  - **+ \( \infty \), - \( \infty \):** \( \exp = 11...1 \) \( \frac{}{\exp} = 00...0 \)
    \[
    1.0/0.0 = -1.0/-0.0 = +\infty, \quad 1.0/-0.0 = -1.0/0.0 = -\infty
    \]
  - **NaN ("Not a Number"):** \( \exp = 11...1 \) \( \frac{}{\exp} \neq 00...0 \)
    Results from operations with undefined result: \( \sqrt{-1} \), \( \infty - \infty \), \( \infty \times 0 \), etc.
  - **Note:** \( \exp=11...1 \) and \( \exp=00...0 \) are reserved, limiting \( \exp \) range...
Normalized Values

\[ V = (-1)^S \times M \times 2^E \]

- **Condition**: \( \exp \neq 000...0 \) and \( \exp \neq 111...1 \)

- **Exponent coded as biased value**: \( E = \exp - \text{Bias} \)
  - \( \exp \) is an unsigned value ranging from 1 to \( 2^{k-2} \) (\( k = \# \text{ bits in } \exp \))
  - \( \text{Bias} = 2^{k-1} - 1 \)
    - Single precision: 127 (so \( \exp \): 1...254, \( E \): -126...127)
    - Double precision: 1023 (so \( \exp \): 1...2046, \( E \): -1022...1023)
  - These enable negative values for \( E \), for representing very small values

- **Significand coded with implied leading 1**: \( M = 1.000...0 \)
  - \( \text{xxx...x} \): the \( n \) bits of \( \text{frac} \)
  - Minimum when \( 000...0 \) (\( M = 1.0 \))
  - Maximum when \( 111...1 \) (\( M = 2.0 - \varepsilon \))
  - Get extra leading bit for “free”
Normalized Encoding Example

\[ V = (-1)^s \cdot M \cdot 2^E \]

**Value:** float \( f = 12345.0 \);
- \( 12345_{10} = 11000000111001_2 \)
  \[ = 1.1000000111001 \times 2^{13} \quad \text{(normalized form)} \]

**Significand:**
- \( M = 1.1000000111001_2 \)
- \( \text{frac} = 10000011100100000000 \)

**Exponent:** \( E = \exp - \text{Bias} \), so \( \exp = E + \text{Bias} \)
- \( E = 13_{10} \)
- \( \text{Bias} = 127_{10} \)
- \( \exp = 140_{10} = 10001100_2 \)

**Result:**
- \( V = (-1)^s \cdot M \cdot 2^E \)
  \[ = (-1)^0 \cdot 1.5069580078125_{10} \times 2^{13_{10}} \]
Distribution of Values

■ 6-bit IEEE-like format
  ■ e = 3 exponent bits
  ■ f = 2 fraction bits
  ■ Bias is $2^{3-1}-1 = 3$

■ Notice how the distribution gets denser toward zero.
Floating Point Operations

- Unlike the representation for integers, the representation for floating-point numbers is not exact.
Floating Point Operations: Basic Idea

\[ V = (-1)^S \times M \times 2^E \]

- \[ x +_f y = \text{Round}(x + y) \]
- \[ x \times_f y = \text{Round}(x \times y) \]

**Basic idea for floating point operations:**
- First, compute the exact result
- Then, *round* the result to make it fit into desired precision:
  - Possibly overflow if exponent too large
  - Possibly drop least-significant bits of significand to fit into \( \text{frac} \)
Floating Point Addition

\((-1)^{s_1} M_1 2^{E_1} + (-1)^{s_2} M_2 2^{E_2}\)

Assume \(E_1 > E_2\)

**Exact Result: \((-1)^s M 2^E\)**

- Sign \(s\), significand \(M\):
  - Result of signed align & add
- Exponent \(E\): \(E_1\)

**Fixing**

- If \(M \geq 2\), shift \(M\) right, increment \(E\)
- If \(M < 1\), shift \(M\) left \(k\) positions, decrement \(E\) by \(k\)
- Overflow if \(E\) out of range
- Round \(M\) to fit \textit{frac} precision

Line up the binary points
Floating Point Multiplication

\((–1)^{s_1} M_1 2^{E_1} * (–1)^{s_2} M_2 2^{E_2}\)

- **Exact Result:** \((–1)^s M 2^E\)
  - Sign \(s\): \(s_1 \wedge s_2\)
  - Significand \(M\): \(M_1 * M_2\)
  - Exponent \(E\): \(E_1 + E_2\)

- **Fixing**
  - If \(M \geq 2\), shift \(M\) right, increment \(E\)
  - If \(E\) out of range, overflow
  - Round \(M\) to fit \(\text{frac}\) precision
Rounding modes

**Possible rounding modes (illustrate with dollar rounding):**

<table>
<thead>
<tr>
<th>Mode</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>Round-toward-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-nearest</td>
<td>$1</td>
<td>$2</td>
<td>??</td>
<td>??</td>
<td>??</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>

**Round-to-even avoids statistical bias in repeated rounding.**

- Rounds up about half the time, down about half the time.
- Default rounding mode for IEEE floating-point.
Mathematical Properties of FP Operations

- Exponent overflow yields $+\infty$ or $-\infty$

- Floats with value $+\infty$, $-\infty$, and NaN can be used in operations
  - Result usually still $+\infty$, $-\infty$, or NaN; sometimes intuitive, sometimes not

- Floating point operations are not always associative or distributive, due to rounding!
  - $(3.14 + 1e10) - 1e10 \neq 3.14 + (1e10 - 1e10)$
  - $1e20 * (1e20 - 1e20) \neq (1e20 * 1e20) - (1e20 * 1e20)$
Floating Point in C

- C offers two levels of precision
  - `float` single precision (32-bit)
  - `double` double precision (64-bit)

- `#include <math.h>` to get `INFINITY` and `NAN` constants

- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results
  - Just avoid them!
Floating Point in C

- **Conversions between data types:**
  - Casting between *int*, *float*, and *double* changes the bit representation.
  - *int* → *float*
    - May be rounded; overflow not possible
  - *int* → *double* or *float* → *double*
    - Exact conversion (32-bit ints; 52-bit frac + 1-bit sign)
  - *long int* → *double*
    - Rounded or exact, depending on word size
  - *double* or *float* → *int*
    - Truncates fractional part (rounded toward zero)
      - E.g. 1.999 -> 1, -1.99 -> -1
    - “Not defined” when out of range or NaN: generally sets to Tmin (even if the value is a very big positive)
Number Representation Really Matters

- **1991: Patriot missile targeting error**
  - clock skew due to conversion from integer to floating point

- **1996: Ariane 5 rocket exploded ($1 billion)**
  - overflow converting 64-bit floating point to 16-bit integer

- **2000: Y2K problem**
  - limited (decimal) representation: overflow, wrap-around

- **2038: Unix epoch rollover**
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to $TMin$ in 2038

- **other related bugs**
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown “smart” warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)
#include <stdio.h>

int main(int argc, char* argv[]) {

    float f1 = 1.0;
    float f2 = 0.0;
    int i;
    for ( i=0; i<10; i++ ) {
        f2 += 1.0/10.0;
    }

    printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
    printf("f1 = %10.8f\n", f1);
    printf("f2 = %10.8f\n", f2);

    f1 = 1E30;
    f2 = 1E-30;
    float f3 = f1 + f2;
    printf ("f1 == f3? %s\n", f1 == f3 ? "yes" : "no");

    return 0;
}

$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119
f1 == f3? yes
Summary

- As with integers, floats suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like ints
  - Some “simple fractions” have no exact representation (e.g., 0.2)
  - Can also lose precision, unlike ints
    - “Every operation gets a slightly wrong result”

- Mathematically equivalent ways of writing an expression may compute different results
  - Violates associativity/distributivity

- Never test floating point values for equality!
- Careful when converting between ints and floats!
Many more details for the curious...

- Denormalized values – to get finer precision near zero
- Distribution of representable values
- Floating point multiplication & addition algorithms
- Rounding strategies

- We won’t be using or testing you on any of these extras in 351.
Denormalized Values

- **Condition:** $exp = 000...0$

- **Exponent value:** $E = exp - Bias + 1$ (instead of $E = exp - Bias$)

- **Significand coded with implied leading 0:** $M = 0 . xxx...x_2$
  - $xxx...x$: bits of $frac$

- **Cases**
  - $exp = 000...0, frac = 000...0$
    - Represents value 0
    - Note distinct values: +0 and −0 (why?)
  - $exp = 000...0, frac \neq 000...0$
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - Equispaced
Special Values

- **Condition:** \( \text{exp} = 111...1 \)

- **Case:** \( \text{exp} = 111...1, \frac{\text{frac}}{000...0} \)
  - Represents value \( \infty \) (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., \( 1.0/0.0 = -1.0/-0.0 = +\infty, \ 1.0/-0.0 = -1.0/0.0 = -\infty \)

- **Case:** \( \text{exp} = 111...1, \frac{\text{frac}}{\neq 000...0} \)
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., \( \sqrt{-1}, \infty - \infty, \infty \times 0 \)
Visualization: Floating Point Encodings

-∞  -Normalized  +∞  -Denorm  +Denorm  +Normalized  0  NaN  NaN

Integers & Floats
Tiny Floating Point Example

8-bit Floating Point Representation
- the sign bit is in the most significant bit.
- the next four bits are the exponent, with a bias of 7.
- the last three bits are the frac

Same general form as IEEE Format
- normalized, denormalized
- representation of 0, NaN, infinity
# Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>000–6</td>
<td>0</td>
<td>0.0000</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>001–6</td>
<td>1/8*1/64 = 1/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>010–6</td>
<td>2/8*1/64 = 2/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>110–6</td>
<td>6/8*1/64 = 6/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>111–6</td>
<td>7/8*1/64 = 7/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>000–6</td>
<td>8/8*1/64 = 8/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>110–1</td>
<td>14/8*1/2 = 14/16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>111–1</td>
<td>15/8*1/2 = 15/16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>0000</td>
<td>8/8*1 = 1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>0010</td>
<td>9/8*1 = 9/8</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>0100</td>
<td>10/8*1 = 10/8</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>110 7</td>
<td>14/8*128 = 224</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>1117</td>
<td>15/8*128 = 240</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>000n/a</td>
<td>n/a</td>
<td>inf</td>
</tr>
</tbody>
</table>

- **Denormalized numbers**: Numbers closest to zero.
- **Normalized numbers**: Numbers closest to 1.
- **Normalized numbers**: Numbers closest to 1 below.
- **Normalized numbers**: Numbers closest to 1 above.
- **Normalized numbers**: Largest norm.
- **Normalized numbers**: Denormalized numbers.
Distribution of Values

- **6-bit IEEE-like format**
  - $e = 3$ exponent bits
  - $f = 2$ fraction bits
  - Bias is $2^{3-1}-1 = 3$

- Notice how the distribution gets denser toward zero.
Distribution of Values (close-up view)

- **6-bit IEEE-like format**
  - $e = 3$ exponent bits
  - $f = 2$ fraction bits
  - Bias is 3

- Denormalized
- Normalized
- Infinity
## Interesting Numbers

<table>
<thead>
<tr>
<th>Description</th>
<th>exp</th>
<th>frac</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>00...00</td>
<td>00...00</td>
<td>0.0</td>
</tr>
<tr>
<td>Smallest Pos. Denorm.</td>
<td>00...00</td>
<td>00...01</td>
<td>$2^{-127} \times 2^{-1022}$</td>
</tr>
<tr>
<td>Single</td>
<td>≈ 1.4 * $10^{-45}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>≈ 4.9 * $10^{-324}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td>$(1.0 - \varepsilon) \times 2^{-127}$</td>
</tr>
<tr>
<td>Single</td>
<td>≈ 1.18 * $10^{-38}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>≈ 2.2 * $10^{-308}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smallest Pos. Norm.</td>
<td>00...01</td>
<td>00...00</td>
<td>$1.0 \times 2^{-127}$</td>
</tr>
<tr>
<td>Just larger than largest denormalized</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One</td>
<td>01...11</td>
<td>00...00</td>
<td>1.0</td>
</tr>
<tr>
<td>Largest Normalized</td>
<td>11...10</td>
<td>11...11</td>
<td>$(2.0 - \varepsilon) \times 2^{127}$</td>
</tr>
<tr>
<td>Single</td>
<td>≈ 3.4 * $10^{38}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>≈ 1.8 * $10^{308}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Special Properties of Encoding

- Floating point zero (0⁺) exactly the same bits as integer zero
  - All bits = 0

- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider 0⁻ = 0⁺ = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity
Floating Point Multiplication

\[ (-1)^{s_1} M_1 \ 2^{E_1} \ * \ (-1)^{s_2} M_2 \ 2^{E_2} \]

- **Exact Result:** \((-1)^s M \ 2^E\)
  - Sign s: \(s_1 \ ^\oplus\ s_2\) // xor of s1 and s2
  - Significand M: \(M_1 \ * \ M_2\)
  - Exponent E: \(E_1 + E_2\)

- **Fixing**
  - If \(M \geq 2\), shift M right, increment E
  - If E out of range, overflow
  - Round M to fit frac precision
Floating Point Addition

\[ (-1)^{s_1} M_1 \ 2^{E_1} + (-1)^{s_2} M_2 \ 2^{E_2} \]

Assume \( E_1 > E_2 \)

**Exact Result:** \((-1)^s \ M \ 2^E\)

- Sign \( s \), significand \( M \):
  - Result of signed align & add
  - Exponent \( E \): \( E_1 \)

**Fixing**

- If \( M \geq 2 \), shift \( M \) right, increment \( E \)
- if \( M < 1 \), shift \( M \) left \( k \) positions, decrement \( E \) by \( k \)
- Overflow if \( E \) out of range
- Round \( M \) to fit frac precision
Closer Look at Round-To-Even

- **Default Rounding Mode**
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under-estimated

- **Applying to Other Decimal Places / Bit Positions**
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth
    - 1.2349999 1.23 (Less than half way)
    - 1.2350001 1.24 (Greater than half way)
    - 1.2350000 1.24 (Half way—round up)
    - 1.2450000 1.24 (Half way—round down)
Rounding Binary Numbers

- **Binary Fractional Numbers**
  - “Half way” when bits to right of rounding position = $100...2$

- **Examples**
  - Round to nearest 1/4 (2 bits right of binary point)

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Rounded</th>
<th>Action</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3/32</td>
<td>10.00011₂</td>
<td>10.00₂</td>
<td>(&lt;1/2—down)</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.00110₂</td>
<td>10.01₂</td>
<td>(&gt;1/2—up)</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.11100₂</td>
<td>11.00₂</td>
<td>( 1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.10100₂</td>
<td>10.10₂</td>
<td>( 1/2—down)</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>
Floating Point Puzzles

For each of the following C expressions, either:

- Argue that it is true for all argument values
- Explain why not true

```
int x = ...;
float f = ...;
double d = ...;
double d2 = ...;
```

1) \( x == (\text{int})(\text{float}) x \)
2) \( x == (\text{int})(\text{double}) x \)
3) \( f == (\text{float})(\text{double}) f \)
4) \( d == (\text{double})(\text{float}) d \)
5) \( f == -(\neg f) \)
6) \( 2/3 == 2/3.0 \)
7) \( (d+d2)-d == d2 \)

Assume neither \( d \) nor \( f \) is NaN