Roadmap

C:
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

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

Machine code:
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111

OS:

Computer system:

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

  "One-hot" encoding
  - Hard to compare values and suits
  - Large number of bits required

Integers

- Representation of integers: unsigned and signed
- Casting
- Arithmetic and shifting
- Sign extension
Two possible representations

- 52 cards – 52 bits with bit corresponding to card set to 1
  - “One-hot” encoding
  - 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?

- 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);
}
```

returns int SUIT_MASK = 0x30 = \[
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
\end{array}
\]equivalent

```c
c
char hand[5]; // represents a 5-card hand
char card1, card2; // two cards to compare
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));
}
```

```c
VALUE_MASK = 0x0F =

suit | value
-----|--
0    | 0
0    | 1
0    | 2
0    | 3
0    | 4
0    | 5
0    | 6
0    | 7
```

```c
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) ) { ... }
```

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...
  - Can not 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)

```c
01100101110101001
```

Unsigned Integers

- **Unsigned values** are just what you expect
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + b_52^5 + \ldots + b_22^2 + b_12^1 + b_02^0$
  - Useful formula: $1+2+4+8+\ldots+2^{W-1} = 2^W - 1$

- **Add and subtract using the normal “carry” and “borrow” rules, just in binary.**

```
01111111 +00001000 + 8
```

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): $0x00 = 0$, $0x01 = 1$, $\ldots$, $0x7F = 127$

- **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):
    - $0x00 = 00000000$ is non-negative, because the sign bit is 0
    - $0x7F = 11111111$ is non-negative
    - $0x85 = 10000101$ is negative
    - $0x80 = 10000000$ is negative...
Signed Integers: Sign-and-Magnitude

- How should we represent -1 in binary?
  - $1000001_2$
    Use the MSB for + or -, and the other bits to give magnitude.

  ![Venn Diagram](image)

  Most Significant Bit

  How do we solve these problems?

  ![Venn Diagram](image)

  signed_and_magnitude_negatives

Sign-and-Magnitude Negatives

- How should we represent -1 in binary?
  - $1000001_2$
    Use the MSB for + or -, and the other bits to give magnitude.
    (Unfortunate side effect: there are two representations of 0!)

  ![Venn Diagram](image)

  signed_and_magnitude_negatives

Two’s Complement Negatives

- How should we represent -1 in binary?

  ![Venn Diagram](image)

  two_complement_negatives
Two’s Complement Negatives

How should we represent -1 in binary?
Rather than a sign bit, let MSB have same value, but negative weight.
\[ b_{w-1} = 1 \text{ adds } -2^{w-1} \text{ to the value. for } i < w-1: \ b_i = 1 \text{ adds } +2^i \text{ to the value.} \]

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

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

-1 is represented as \(1111\) = \(-2^3 + (2^3 - 1)\)
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!
\[ \sim x + 1 = -x \]
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 modulo $2^W$

- Examples:

<table>
<thead>
<tr>
<th>4</th>
<th>0100</th>
<th>4</th>
<th>0100</th>
<th>−4</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td>+0011</td>
<td>−3</td>
<td>+1101</td>
<td>+3</td>
<td>+0011</td>
</tr>
<tr>
<td>= 7</td>
<td>= 0111</td>
<td>= 1</td>
<td>= 0001</td>
<td>= 1</td>
<td>= 1111</td>
</tr>
<tr>
<td></td>
<td>drop carry = 0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Why does it work?

- Put another way, for all positive integers $x$, we want:
  - $\text{bits}(x) + \text{bits}(-x) = 0$ (ignoring the carry-out bit)

  - This turns out to be the bitwise complement plus one
  - What should the 8-bit representation of -1 be?

    | 00000001 | +????????? | (we want whichever bit string gives the right result)
    | 00000000 |

    | 00000010 | 00000011 |
    | 00000000 | 00000000 |
Two's Complement

Why does it work?

- Put another way, for all positive integers \( x \), we want:
  \[ \text{bits}(x) + \text{bits}(-x) = 0 \] (ignoring the carry-out bit)

- This turns out to be the bitwise complement plus one
  - What should the 8-bit representation of -1 be?
    
    | Bits  | Signed | Unsigned |
    |-------|--------|----------|
    | 00000000 | 0      | 0        |
    | +11111111 | 1      | 255      |
    | 00000010 | -2     | 16       |
    | +???????? | -3     |          |
    | 00000000 | 0      | 0        |

This turns out to be the bitwise complement plus one

What should the 8-bit representation of -1 be?

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+11111111</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>00000010</td>
<td>-2</td>
<td>16</td>
</tr>
<tr>
<td>+????????</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

00000001 + 11111111 (we want whichever bit string gives the right result)

00000000 00000010 00000011
+???????? +????????
00000000 00000000

Conversion Visualized

Two’s Complement → Unsigned

- Ordering Inversion
- Negative → Big Positive

Unsigned & Signed Numeric Values

- Signed and unsigned integers have limits.
  - If you compute a number that is too big (positive), it wraps:
    \[ 6 + 4 = ? \quad 1U + 2U = ? \]
  - If you compute a number that is too small (negative), it wraps:
    \[ -7 - 3 = ? \quad 0U - 2U = ? \]
  - Answers are only correct mod \( 2^b \)

- 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.

Unsigned & Signed Numeric Values

<table>
<thead>
<tr>
<th>bits</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Signed</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
Overflow/Wrapping: Unsigned

addition: drop the carry bit

\[
\begin{align*}
15 & + 2 & 1111 + 0010 & = 10001 \\
17 & & & \text{Overflow!}
\end{align*}
\]

Overflow/Wrapping: Two’s Complement

addition: drop the carry bit

\[
\begin{align*}
-1 & + 2 & 1111 + 0010 & = 10001 \\
1 & & & \text{Overflow!}
\end{align*}
\]

Values To Remember

- **Unsigned Values**
  - UMin = 0
    - 000...0
  - UMax = \(2^w - 1\)
    - 111...1

- **Two’s Complement Values**
  - TMin = \(-2^{w-1}\)
    - 100...0
  - TMax = \(2^{w-1} - 1\)
    - 011...1
  - Negative one
    - 111...1 \(0xF...F\)

Values for \(W = 32\)

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>4,294,967,296</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>2,147,483,647</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-2,147,483,648</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>FF FF FF FF</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<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?

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: TMIN = -2,147,483,648, TMAX = 2,147,483,647

<table>
<thead>
<tr>
<th>Constant₁</th>
<th>Constant₂</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>

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:
  - \( X' = x_{w-1}, x_{w-1}, x_{w-1}, ..., x_0 \)

---

### 8-bit representations

| 0 0 0 0 1 0 0 1 | 1 0 0 0 0 0 0 1 |
| 0 0 0 0 0 0 1 1 | 0 0 1 0 0 1 1 1 |

C: casting between unsigned and signed just reinterprets the same bits.

---

### 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 0 0 | 8-bit -4 |

---

| 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 |
| 0 0 0 0 1 1 0 0 | 8-bit 12 |
# Sign Extension

### Example

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

```c
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>00110000 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>12345</td>
<td>00 00 30 39 00000000 00011000 00110110</td>
</tr>
<tr>
<td>y</td>
<td>-12345</td>
<td>C7 F7 11001111 11000111</td>
</tr>
<tr>
<td>iy</td>
<td>-12345</td>
<td>FF FF CF C7 11111111 11111111 11001111 11000111</td>
</tr>
</tbody>
</table>

# Shift Operations

- **Left shift:** $x << y$
  - Shift bit vector $x$ left by $y$ positions
  - Throw away extra bits on left
  - Fill with 0s on right
- **Right shift:** $x >> y$
  - Shift bit-vector $x$ right by $y$ 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 $x$

The behavior of $\gg$ in C depends on the compiler! It is *arithmetic* shift right in GCC. Java: $\gg$ is logical shift right, $\gg$ is arithmetic shift right.
Shift Operations

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

- **Right shift**: \( x \gg y \)
  - Shift bit vector \( x \) right by \( y \) 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 \( x \)
  - *Why is this useful?*

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \ll 3 )</td>
<td></td>
</tr>
<tr>
<td>Logical ( \gg 2 )</td>
<td></td>
</tr>
<tr>
<td>Arithmetic ( \gg 2 )</td>
<td></td>
</tr>
</tbody>
</table>

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

What happens when...

- \( x \gg n \): divide by \( 2^n \)
- \( x \ll m \): multiply by \( 2^m \)

- Faster than general multiple or divide operations

**Shifting and Arithmetic**

- \( x = 27; \) \( y = x \ll 2; \) \( y = 108 \)
- \( x = 237; \) \( y = x \gg 2; \) \( y = 59 \)

\[ x \gg n \quad \text{logical shift right:} \quad \text{shift in zeros from the left} \]
\[ x \ll m \quad \text{logical shift left:} \quad \text{shift in zeros from the right} \]

- *overflow*
- *rounding (down)*
Shifting and Arithmetic

signed
\[ x = -101; \]
\[ y = x \ll 2; \]
\[ y = 108 \]
\[ x = -19; \]
\[ y = x \gg 2; \]
\[ y = -5 \]

overflow

\[ x/2^n \]

arithmetic shift right:
\[ 11101101 \]
shift in copies of most significant bit from the left

clarification from Mon.: shifts by \( n < 0 \) or \( n \geq \) word size are undefined

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>x</th>
<th>01100001</th>
<th>01100010</th>
<th>01100111</th>
<th>01100100</th>
</tr>
</thead>
<tbody>
<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>01100000</td>
</tr>
</tbody>
</table>

- Extract the sign bit of a signed integer?

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>x</th>
<th>01100001</th>
<th>01100010</th>
<th>01100111</th>
<th>01100100</th>
</tr>
</thead>
<tbody>
<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>01100000</td>
</tr>
</tbody>
</table>

- Extract the sign bit of a signed integer?
  - \( (x \gg 31) \& 1 \) - need the “\& 1” to clear out all other bits except LSB

- Conditionals as Boolean expressions (assuming \( x \) is 0 or 1)
  - if \((x) a=y \) else \( a=z \);
  - which is the same as \( a = x ? y : z \);
  - Can be re-written (assuming arithmetic right shift) as:
    \[ a = ((x \ll 31) >> 31) \& y | ((x \ll 31) >> 31) \& z; \]
Multiplication

- What do you get when you multiply $9 \times 9$?
- What about $2^{30} \times 3$?
- $2^{30} \times 5$?
- $-2^{31} \times -2^{31}$?

Power-of-2 Multiply with Shift

- **Operation**
  - $u << k$ gives $u \cdot 2^k$
  - Both signed and unsigned

<table>
<thead>
<tr>
<th>$u$</th>
<th>$2^k$</th>
<th>True Product: $w+k$ bits</th>
<th>Discard $k$ bits: $w$ bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$2^k$</td>
<td>$u \cdot 2^k$</td>
<td>UMult_w(u, 2^k)</td>
</tr>
</tbody>
</table>

- **Examples**
  - $u << 3 = u \cdot 8$
  - $u << 5 - u << 3 = u \cdot 24$
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

Unsigned Multiplication in C

- **Operands**: $w$ bits
  - $u$ | $v$ |

- **True Product**: $2^w$ bits
  - $u \cdot v$ | $u \cdot v$ |

- **Discard $w$ bits**: $w$ bits
  - UMult_w(u, v) |

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

- **Implements Modular Arithmetic**
  - $\text{UMult}_w(u, v) = u \cdot v \mod 2^w$

Code Security Example

```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);
  printf("\%s\n", mybuf);
}
```

Code Security Example

```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);
  printf("\%s\n", mybuf);
}
```
Malicious Usage

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

/* 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);

Autumn 2013
Integers & Floats
57

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...

Autumn 2013
Integers & Floats
58

Fractional Binary Numbers

- Value | Representation
- 5 and 3/4 | 101.11_2
- 2 and 7/8 | 10.111_2
- 47/64 | 0.101111_2

- Observations
  - Shift left = multiply by power of 2
  - Shift right = divide by power of 2
  - Numbers of the form 0.111111..._2 are just below 1.0

- Limitations:
  - Exact representation possible only for numbers of the form x * 2^y
  - Other rational numbers have repeating bit representations
    - 1/3 = 0.333333..._10 = 0.01010101[01]..._2

Autumn 2013
Integers & Floats
59

Fractional Binary Numbers

- Representation
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number:
    \[ \sum_{k=-j}^{\infty} b_k \cdot 2^k \]
Fixed Point Representation

- **Implied binary point. Examples:**
  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: \( b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \)

- **Same hardware as for integer arithmetic.**
  3. Integers! The binary point is after bit 0: \( b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \)

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

IEEE Floating Point

- **Analogous to scientific notation**
  - \( 12000000 \) \( = 1.2 \times 10^7 \) \( C: 1.2e7 \)
  - \( 0.0000012 \) \( = 1.2 \times 10^{-6} \) \( C: 1.2e-6 \)

- **IEEE Standard 754 used by all major CPUs today**

- **Driven by numerical concerns**
  - Rounding, overflow, underflow
  - Numerically well-behaved, but hard to make fast in hardware

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
  - s exp frac
  - 1 bit 8 bits 23 bits

- Double precision: 64 bits
  - s exp frac
  - 1 bit 11 bits 52 bits

Finite representation means not all values can be represented exactly. Some will be approximated.

**Normalization and Special Values**

- “Normalized” = M has the form 1.xxxxx
  - As in scientific notation, but in binary
  - 0.011 x 2^3 and 1.1 x 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 / undefined values like 1.0/0.0?

**Normalization and Special Values**

- zero:
  - s == 0 exp == 00...0 frac == 00...0
- \( +\infty, -\infty \):
  - exp == 11...1 frac == 00...0

  \( 1.0/0.0 = -1.0/-0.0 = +\infty,\ 1.0/-0.0 = -1.0/0.0 = -\infty \)

- NaN (“Not a Number”): exp == 11...1 frac != 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...

**Floating Point Operations: Basic Idea**

- \( x +_\varepsilon y = Round(x + y) \)

- \( x \times_\varepsilon y = Round(x \times y) \)

**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 frac
Floating Point Multiplication

$\left(-1\right)^{s_1} M_1 \ 2^{E_1} * \left(-1\right)^{s_2} M_2 \ 2^{E_2}$

- **Exact Result:** $\left(-1\right)^{s} M \ 2^{E}$
  - Sign $s$: $s_1 \land s_2$
  - 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 \texttt{frac} precision

Floating Point Addition

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

Assume $E_1 > E_2$

- **Exact Result:** $\left(-1\right)^{s} M \ 2^{E}$
  - 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 \texttt{frac} precision

Rounding modes

- **Possible rounding modes (illustrate with dollar rounding):**
  - Round-toward-zero: $1 \ 1 \ 1 \ 2 \ -1$
  - Round-down (-\infty): $1 \ 1 \ 1 \ 2 \ -2$
  - Round-up (+\infty): $2 \ 2 \ 2 \ 3 \ -1$
  - Round-to-nearest: $1 \ 2 \ ?? \ ?? \ ??$
  - Round-to-even: $1 \ 2 \ 2 \ 2 \ -2$

- **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!**
  - $(3.14 + 1e10) - 1e10 != 3.14 + (1e10 - 1e10)$
  - $1e20 * (1e20 - 1e20) != (1e20 * 1e20) - (1e20 * 1e20)$
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 and the Programmer

```c
#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\n", f2);
  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no");
  return 0;
}
```

Floating Point and the Programmer

- **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)
      - Not defined when out of range or NaN: generally sets to `Tmin`

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)
Memory Referencing Bug

double fun(int i)
{
    volatile double d[1] = (3.14);
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14, then segmentation fault

Explanation:

Saved State

d7 ... d4  d3 ... d0
0100 0000 0000 1001 0001 1110 1011 1000
0101 0000 ...

Location accessed by fun(i)

Memory Referencing Bug (Revisited)

double fun(int i)
{
    volatile double d[1] = (3.14);
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14, then segmentation fault

Saved State

d7 ... d4  d3 ... d0
0100 0000 0000 1001 0001 1110 1011 1000
0101 0000 ...

Location accessed by fun(i)

Representing 3.14 as a Double FP Number

 wicht 1073741824 = 0100 0000 0000 0000 0000 0000 0000 0000

仇恨 3.14 = 11.0010 0011 1110 1011 1000 0101 000...

仇恨 (–1) M 2E

仇恨 S = 0 encoded as 0
仇恨 M = 1.1001 0001 1110 1011 1000 0101 0000... (leading 1 left out)
仇恨 E = 1 encoded as 1024 (with bias)

Saved State

d7 ... d4  d3 ... d0
0100 0000 0000 1001 0001 1110 1011 1000
0100 0000 0000 0000 0000 0000 0000 0000

Location accessed by fun(i)
Memory Referencing Bug (Revisited)

```c
double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

fun(0)  –> 3.14
fun(1)  –> 3.14
fun(2)  –> 3.1399998664856
fun(3)  –> 2.00000061035156
fun(4)  –> 3.14, then segmentation fault

Saved State

| d7 ... d4 | 0100 0000 0000 0000 0000 0000 0000 0000 |
| d3 ... d0 | 0101 0000 ... |
| a[1]      |       |
| a[0]      |       |

Location accessed by `fun(i)`

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...

- Exponent bias
- 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.
Normalized Values

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

- **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 \( 2^k - 2 \) (\( k = \# \) 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:** \( M = 1 \cdot \frac{x}{2} \)
  - \( \frac{x}{2} \): the \( n \) bits of \( \text{frac} \)
  - Minimum when 000...0 (\( M = 1.0 \))
  - Maximum when 111...1 (\( M = 2.0 - \epsilon \))
  - Get extra leading bit for "free"

Normalized Encoding Example

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

- **Value:** \( \text{float} f = 12345.0 \);
  - \( 12345_{10} = 11000000111001_2 \)
    - \( 1.1000000111001 \times 2^{13} \) (normalized form)
- **Significand:** \( M = 1 \cdot \frac{x}{2} \)
  - \( \frac{x}{2} = 10000001110010000000000_2 \)
- **Exponent:** \( E = \exp - \text{Bias} \), so \( \exp = E + \text{Bias} \)
  - \( E = 13 \)
  - \( \text{Bias} = 127 \)
  - \( \exp = 140 = 10001100_2 \)
- **Result:** \[ \begin{array}{ccc}
  0 & 10001100 & 10000001110010000000000 \\
  s & \text{exp} & \text{frac}
\end{array} \]

Denormalized Values

- **Condition:** \( \exp = 000...0 \)
- **Exponent value:** \( E = \exp - \text{Bias} + 1 \) (instead of \( E = \exp - \text{Bias} \))
- **Significand coded with implied leading 0:** \( M = 0 \cdot \frac{x}{2} \)
  - \( \frac{x}{2} \): bits of \( \text{frac} \)
- **Cases**
  - \( \exp = 000...0, \frac{x}{2} = 000...0 \)
    - Represents value 0
    - Note distinct values: +0 and -0 (why?)
  - \( \exp = 000...0, \frac{x}{2} \neq 000...0 \)
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - Equispaced

Special Values

- **Condition:** \( \exp = 111...1 \)
- **Case:** \( \exp = 111...1, \frac{x}{2} = 000...0 \)
  - 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:** \( \exp = 111...1, \frac{x}{2} \neq 000...0 \)
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., \( \sqrt{-1}, \infty - \infty, \infty * 0 \)
Visualization: Floating Point Encodings

![Floating Point Encoding Diagram]

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 \( \text{frac} \)

- **Same general form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity

Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>s exp frac</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 000</td>
<td>-6 0</td>
</tr>
<tr>
<td>0 0000 001</td>
<td>-6 1/8*1/64 = 1/512</td>
</tr>
<tr>
<td>0 0000 010</td>
<td>-6 2/8*1/64 = 2/512</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0 0000 110</td>
<td>-6 6/8*1/64 = 6/512</td>
</tr>
<tr>
<td>0 0000 111</td>
<td>-6 7/8*1/64 = 7/512</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0 0001 000</td>
<td>-6 8/8*1/64 = 8/512</td>
</tr>
<tr>
<td>0 0001 001</td>
<td>-6 9/8*1/64 = 9/512</td>
</tr>
</tbody>
</table>

- **Denormalized numbers**
  - closest to zero
  - largest denorm

- **Normalized numbers**
  - smallest norm
  - closest to 1 below
  - closest to 1 above
  - largest norm

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

Interesting Numbers

<table>
<thead>
<tr>
<th>Description</th>
<th>$\exp$</th>
<th>$\frac{1}{2}$</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>11...11</td>
<td>$(1.0 - \varepsilon) \times 2^{-126,1022}$</td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...00</td>
<td>11...10</td>
<td>$2^{-126,1022}$</td>
</tr>
<tr>
<td>Smallest Pos. Norm.</td>
<td>00...01</td>
<td>00...00</td>
<td>$1.0 \times 2^{-126,1022}$</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>$(2.0 - \varepsilon) \times 2^{127,1023}$</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

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

- Exact Result: $(-1)^s \times M \times 2^E$
  - Sign $s$: $s_1 \oplus s_2$ // xor of $s_1$ and $s_2$
  - Significand $M$: $M_1 \times M_2$
  - 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

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

- **Exact Result: \((-1)^s M 2^E\)**
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E: E1

- **Fixing**
  - If M ≥ 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

Rounding Binary Numbers

- **Binary Fractional Numbers**
  - “Half way” when bits to right of rounding position = \(100\ldots\)

- **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>10.00011_2</td>
<td>10.00_2</td>
<td>&lt;1/2—down</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.00110_2</td>
<td>10.01_2</td>
<td>&gt;1/2—up</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.11100_2</td>
<td>11.00_2</td>
<td>1/2—up</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.10100_2</td>
<td>10.10_2</td>
<td>1/2—down</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>

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)