Today

- Representing cards
- Representing integers
  - Sign and magnitude
  - Two's complement
  - Arithmetic algorithms
- Shifting
  - For arithmetic
  - For fun and profit

Integer Representation

- 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
  - "One-hot" encoding
  - Drawbacks:
    - Two 32-bit words
    - 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
  - Fits in one 32-bit word
  - Easier to compare suits and values
  - Still space-inefficient
Two better representations

- Binary encoding of all 52 cards – only 6 bits needed
  - Number each card
  - 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
  - Number each suit
  - Number each value
  - Fits in one byte
  - Easy suit, value comparisons

Example code:

```java
byte hand[5];       // represents a 5-card hand
byte card1, card2;  // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( sameSuit(card1, card2) ) { ... }

static final SUIT_MASK = 0x30;
boolean sameSuit(byte card1, byte card2) {
    return 0 != (card1 & SUIT_MASK) ^ (card2 & SUIT_MASK);
}
```

Compare Card Suits

- Works even if value is stored in high bits

Example code:

```java
byte hand[5];       // represents a 5-card hand
byte card1, card2;  // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( sameSuit(card1, card2) ) { ... }
```

Encoding Integers

- The hardware (and C) supports two flavors of integers:
  - `unsigned` (\(\subset\)) – non-negatives only
  - `signed` (\(\subset\)) – 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: \(0 \ldots 2^W\)
  - Signed values: \(-2^{W-1} \ldots 2^{W-1}\)

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.

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

- 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
    - \( 0x80 = 10000000 \) is negative
    - \( 0x80 = 10000000 \) is negative...

Sign-and-Magnitude Negatives

- How should we represent -1 in binary?
  - 10000001
    - Use the MSB for + or - , and the other bits to give magnitude.
    - Unfortunate side effect?

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

Two’s Complement Negatives

- How should we represent -1 in binary?

- How do we solve these problems?
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 \( i < w-1 \): \( b_i = 1 \text{ adds } 2^i \text{ to the value.} \)

  \[ \begin{array}{c|c|c}
  \text{Positive} & \text{Two's Complement} \\
  \hline
  0000 & 1111 \\
  0001 & 1110 \\
  0010 & 1101 \\
  0011 & 1100 \\
  0100 & 1011 \\
  0101 & 1010 \\
  0110 & 1001 \\
  0111 & 1000 \\
  1000 & 0111 \\
  1001 & 0110 \\
  1010 & 0101 \\
  1011 & 0100 \\
  1100 & 0011 \\
  1101 & 0010 \\
  1110 & 0001 \\
  1111 & 0000 \\
  \end{array} \]

  - E.g., \( 10_{10} \): unsigned:
    - Two's complement:

  - All negative integers still have MSB = 1.
  - Single zero, simple arithmetic.
  - Cool rules:
    - \( x + \neg x = 0 \)
    - \( \neg x + 1 = -x \)

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

  \[ \begin{array}{c|c|c}
  \text{Positive} & \text{Two's Complement} \\
  \hline
  0000 & 1111 \\
  0001 & 1110 \\
  0010 & 1101 \\
  0011 & 1100 \\
  0100 & 1011 \\
  0101 & 1010 \\
  0110 & 1001 \\
  0111 & 1000 \\
  1000 & 0111 \\
  1001 & 0110 \\
  1010 & 0101 \\
  1011 & 0100 \\
  1100 & 0011 \\
  1101 & 0010 \\
  1110 & 0001 \\
  1111 & 0000 \\
  \end{array} \]

  - E.g., \( 10_{10} \): unsigned:
    - Two's complement:

  - "modular" addition: result is sum modulo \( 2^w \) for \( W \) bits

Two's Complement Arithmetic

- The same addition procedure works for both unsigned and two's complement integers
  - Simple hardware
  - Design principle: simplicity favors regularity
  - Algorithm: simple addition, discard the highest carry bit

  \[ \begin{array}{c|c|c|c}
  \hline
  4 & 0100 & -4 & 1100 \\
  3 & 0011 & 3 & 0111 \\
  \hline
  \text{Examples:} & \text{drop carry} & = 0001 \end{array} \]
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?

\[
\begin{array}{c}
00000001 \\
+ \\
00000000 \\
\hline
00000010 \\
00000011 \\
+ \\
00000000 \\
\hline
00000000
\end{array}
\]

Unsigned & Signed Numeric Values

- Overflow
  - If you compute a number that is too big (positive), it wraps:
    \[ 6 + 4 = 7 \] \[ 15U + 2U = ? \]
  - If you compute a number that is too small (negative), it wraps:
    \[ -7 + 3 = 0U - 2U = ? \]
  - Answers are only correct \( \mod 2^w \)

- MIPS: overflow exception

- C and Java cruise along silently when overflow occurs... Oops?

Conversion Visualized

Two’s Complement \( \rightarrow \) Unsigned

- Ordering Inversion
- Negative \( \rightarrow \) Big Positive

Values To Remember

- Unsigned Values
  - \( U_{\text{Min}} = 0 \)
  - \( U_{\text{Max}} = 2^w - 1 \)

- Two’s Complement Values
  - \( T_{\text{Min}} = -2^{w-1} \)
  - \( T_{\text{Max}} = 2^w - 1 \)
  - Negative one

Values for \( W = 32 \)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,294,967,296</td>
<td>FF FF FF FF</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>2,147,483,647</td>
<td>FF FF FF FF</td>
<td>10111111 10111111 10111111 10111111</td>
</tr>
<tr>
<td>-1</td>
<td>7F FF FF FF</td>
<td>01010101 01010101 01010101 01010101</td>
</tr>
<tr>
<td>-2</td>
<td>80 00 00 00</td>
<td>10000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>0</td>
<td>00 00 00 00</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>
Sign Extension

- What happens if you convert a 16-bit signed integer to a 32-bit signed integer?

Sign Extension Example

- Converting from smaller to larger integer data type
- Java and C automatically perform sign extension.

```java
short x = 12345;
int ix = (int) x;
short 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>1x</td>
<td>00110000 01101101</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>-12345</td>
<td>11001111 11000111</td>
</tr>
<tr>
<td>iy</td>
<td>-12345</td>
<td>11001111 11000111</td>
</tr>
</tbody>
</table>

Do we really need immediates?

(How to make nothing from anything)

```java
int makeZero(int any) {
    return
}

int makeOne(int any) {
    int zero = makeZero(any);
    return
}

int makeTwo(int any) {
    int one = makeZero(any);
    return
}
```

See also: Church Numerals, for an even more impressive (non-CS240) and vaguely similar idea: http://en.wikipedia.org/wiki/Church_encoding
### Shift Operations (Java syntax)

**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**
  - Fill with 0s on left
- **Arithmetic shift**
  - Replicate most significant bit on left
- **Why is this useful?**

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 01100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &lt;&lt; 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>( x &gt;&gt;&gt; 2 )</td>
<td>( 00011000 )</td>
</tr>
<tr>
<td>( x &gt;&gt; 2 )</td>
<td>( 00011000 )</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 else happens when...

- \( x >> n \)?
- \( x << m \)?

### What happens when...

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

Faster than general multiply or divide operations

### Using Shifts and Masks

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

```
x 01100001 01100010 01100011 01100100
```
Using Shifts and Masks

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

<table>
<thead>
<tr>
<th>x</th>
<th>01100000 01100010 01100100 01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x ( \gg 16 )</td>
<td>00000000 00000000 01100010 01100010</td>
</tr>
<tr>
<td>( (x \gg 16) &amp; \text{0xFF} )</td>
<td>00000000 00000000 00000000 11111111</td>
</tr>
</tbody>
</table>

- Extract the sign bit of a signed integer?

\[
\begin{align*}
(x \gg 31) \& 1 & \quad \text{need the “& 1” to clear out all other bits except LSB}
\end{align*}
\]

<table>
<thead>
<tr>
<th>x</th>
<th>01100001 01100010 01100011 01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x ( \gg 16 )</td>
<td>00000000 00000000 01100010 01100010</td>
</tr>
<tr>
<td>( (x \gg 16) &amp; \text{0xFF} )</td>
<td>00000000 00000000 00000000 12222222</td>
</tr>
</tbody>
</table>

Unsigned Multiplication in C

- Standard Multiplication Function
  - Ignores high order \( w \) bits
  - Implements Modular Arithmetic
    \[
    \text{UMult}((u,v)) = (u \cdot v) \mod 2^w
    \]
Power-of-2 Multiply with Shift

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

<table>
<thead>
<tr>
<th>$u$</th>
<th>$u \times 2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{Operands: } w \text{ bits}$</th>
<th>$\text{True Product: } w+k \text{ bits}$</th>
<th>$\text{Discard } k \text{ bits: } w \text{ bits}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$u \times 2^k$</td>
<td>$u$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

- **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

---

Two’s complement review

- **4-bit Unsigned vs. Two’s Complement**

| 1 | 0 | 1 | 1 |
| 2 | 1 | 0 | 1 |

- **True Product:**
  - $2^3 \times 1 + 2^2 \times 1 = 1011$
  - $-2^3 \times 1 + 2^2 \times 1 = 1011$

- **Difference:**
  - $11 - (-5) = 16 = 2^4$

- **Math:**
  - $11 - (-5) = 16 = 2^4$
### 4-bit Unsigned vs. Two's Complement

- **1011**
- $2^3 + 2^2 + 2^1 + 2^0 = 16$
- **(math) difference** = $16 = 2^4$

### 8-bit representations

<table>
<thead>
<tr>
<th>4-bit unsigned</th>
<th>8-bit unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>00000010</td>
</tr>
<tr>
<td>1100</td>
<td>00000100</td>
</tr>
</tbody>
</table>

### Sign Extension

<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>00000010</td>
</tr>
<tr>
<td>1100</td>
<td>00000100</td>
</tr>
<tr>
<td>?????1100</td>
<td>00001100</td>
</tr>
</tbody>
</table>

---

C: Casting between unsigned and signed just reinterprets the same bits.
Sign Extension

0010 4-bit 2

00000010 8-bit 2

1100 4-bit -4

10001100 8-bit -116

Overflow/Wrapping: Unsigned

addition: drop the carry bit

15 + 2 17

1111 + 0010 1001

Modular Arithmetic

Overflow/Wrapping: Two’s Complement

addition: drop the carry bit

-1 + 2 1 + 0010 1001

6 + 3 9 + 0011 1001

-7 Modular Arithmetic
Shifting and Arithmetic

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y = 108 \]