Integer Representation

Representation of integers: unsigned and signed
Sign extension
Arithmetic and shifting
Casting

But first, encode deck of cards.

52 cards in 4 suits
How do we encode suits, face cards?
What operations should be easy to implement?
Get and compare rank
Get and compare suit

Two possible representations

52 cards – 52 bits with bit corresponding to card set to 1
“One-hot” encoding
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
**Compare Card Suits**

```
static final SUIT_MASK = 0x30;

boolean sameSuit(char card1, char card2) {
  return 0 == ((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK));
}
```

**Compare Card Values**

```
static final VALUE_MASK = 0x0F;

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

**Encoding Integers in a fixed number of bits**

Two flavors:
- **unsigned**: non-negatives only
- **signed**: both negatives and non-negatives

Positional representation, fixed # of positions.

Only $2^W$ distinct bit patterns...

Cannot represent all the integers
- **Unsigned values**: 0 ... $2^W$-1
- **Signed values**: $-2^{W-1}$ ... $2^{W-1}$-1

Terminology:
- “Most-significant” or “high-order” bit(s)
- “Least-significant” or “low-order” bit(s)

**Unsigned modular arithmetic, overflow**

Examples in 4-bit unsigned representation.

```
11 + 2 = 13 + 5 = 1110 + 0010 = 0000
```

$x + y$ in N-bit unsigned arithmetic is $(x + y) \mod 2^N$ in math

Unsigned overflow = "wrong" answer = wrap-around
  = carry 1 out of MSB = math answer too big to fit
Overflow: Unsigned

Addition overflows if and only if a carry bit is dropped.

15
+ 2
---
17

Overflow.

Signed Integers: Sign-Magnitude?

Most-significant bit (MSB) is sign bit
0 means non-negative
1 means negative
Rest of bits are an unsigned magnitude

8-bit sign-and-magnitude:
0x00 = 00000000 represents
0x7F = 11111111 represents
0xFF = 10000001 represents
0x80 = 10000000 represents

Max and min for N-bit sign-magnitude?

What is weird about sign-magnitude representation?

Sign-Magnitude Negatives

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

0100
-1011

What about zero?

Two’s complement representation

for signed integers

w-bit representation

-2^(w-1) ... 2^i ... 2^3 2^2 2^1 2^0

Positional representation, but most significant position has negative weight.
8-bit representations

\[
\begin{align*}
00001001 & \quad 10000001 \\
11111111 & \quad 00100111
\end{align*}
\]

Two’s complement: addition Just Works

<table>
<thead>
<tr>
<th>Input</th>
<th>2</th>
<th>+3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>0010</td>
<td>+011</td>
<td>0101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>1</th>
<th>+3</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>1110</td>
<td>+011</td>
<td>1001</td>
</tr>
</tbody>
</table>

4-bit unsigned vs. 4-bit two’s complement

\[
\begin{align*}
1011 & \quad 1101 \\
1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 & \quad 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
11 & \quad (\text{math difference} = 16 = 2^4) & \quad -5
\end{align*}
\]

Overflow: Two’s Complement

Addition overflows if and only if the inputs have the same sign but the output does not. If and only if the carry in and out of the sign bit differ.

<table>
<thead>
<tr>
<th>Input</th>
<th>1</th>
<th>+3</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>1111</td>
<td>+0011</td>
<td>0110</td>
</tr>
</tbody>
</table>

Modular Arithmetic

Some CPUs raise exceptions on overflow. C and Java cruise along silently... Oops?
A few reasons two’s complement is awesome

Better be true!
\[ x + -x = 0 \]

N-bit negative one is N ones.

Complement rules:
\[ x + \sim x = -1 \]
\[ -x + 1 = -x \]

Subtraction is just addition:
\[ 4 - 3 = 4 + (-3) \]

Great news for hardware.

Another view

How should we represent 8-bit negatives?
- For all positive integers \( x \) and widths \( n \), the n-bit representations of \( x \) and \(-x\) must sum to zero.
- Arithmetic should be “the same.”

\[
\begin{align*}
00000001 & + 00000010 & + 00000011 \\
00000000 & + 00000000 & + 00000000
\end{align*}
\]

- Find a rule to represent \(-x\) where that works...

Conversion Visualized

Two’s Complement → Unsigned

Ordering Inversion
Negative → Big Positive

Unsigned Range

Values To Remember

Unsigned Values

| UMin | 0 |
| TMin | \(-2^{n-1}\) |

Two’s Complement Values

| UMin | 0 |
| TMin | \(-2^{n-1}\) |

Values for \( W = 32 \)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>4,294,967,296</td>
<td>FF FF FF FF</td>
</tr>
<tr>
<td>Tmax</td>
<td>2,147,483,647</td>
<td>FF FF FF FF</td>
</tr>
<tr>
<td>Tmin</td>
<td>-2,147,483,648</td>
<td>80 00 00 00</td>
</tr>
<tr>
<td>-1</td>
<td>FF FF FF FF</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 00 00 00</td>
<td></td>
</tr>
</tbody>
</table>
Sign Extension

<table>
<thead>
<tr>
<th>0 0 0 0 0 0 1 0</th>
<th>8-bit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0 0 1 0</td>
<td>16-bit 2</td>
</tr>
<tr>
<td>1 1 1 1 1 0 0</td>
<td>8-bit -4</td>
</tr>
<tr>
<td>? ? ? ? ? ? ? ? 1 1 1 1 1 0 0</td>
<td>16-bit -4</td>
</tr>
</tbody>
</table>

Try some possibilities...

Shift Operations

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

Right shift: \( x >> y \)
- Shift vector \( x \) right by \( y \) positions
- Throw away extra bits on right
- Fill with ??? on left

Logical shift
- Fill with 0s on left

Arithmetic shift
- Replicate most significant bit on left
- Why is this useful? Rain check!

Shift gotchas

For a type represented by \( n \) bits, shift by no more than \( n-1 \).

C: shift by \( <0 \) or \( \geq \) (bits in type) is undefined.
- means anything could happen, including computer catching fire

Java: shift value is used modulo number of bits in shifted type
- given int \( x \):
  \( x << 34 == x << 2 \)

in C: meaning of \( >> \) on signed types is compiler-defined! GCC: arithmetic shift
in Java: \( >> \) is arithmetic, \( >>> \) is logical
Using Shifts and Masks

Extract 2nd most significant byte from a 32-bit integer:

\[
x = 01100001 01100010 01100011 01100100
\]

Extract the sign bit of a signed integer:

Shifting and Arithmetic: signed

\[
x = -101;\quad y = x \ll 2;\quad y = 108
\]

Signed:
- For \( x > 0 \):
  - arithmetic shift right:
    - shift in copies of MSB from the left
  - For \( x < 0 \) it rounds the opposite direction!

- For \( x < 0 \):
  - arithmetic shift right:
    - shift in copies of MSB from the left

Shifting and Arithmetic: unsigned

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

(unsigned)
- logical shift left:
- shift in zeros from the right

\[
x = 237;\quad y = x \gg 2;\quad y = 59
\]

(unsigned)
- logical shift right:
- shift in zeros from the left

Multiplication

Operands: \( w \) bits

True Product: \( 2^w \) bits

Discard \( w \) bits: \( w \) bits

unted: \( u * v = (u \cdot v) \mod 2^w \)

- overflow iff any bit \( b_y >> w \neq 0 \)

- signed: overflow iff any bit \( b_y >> w \neq b_{w-1} \)

For all programming languages

- More generally true about loss of value when casting to smaller types.
- High bits are just discarded.
Power-of-2 Multiply with Shift

Operation
\[ u \cdot 2^k = u \cdot 2^k \]

Both signed and unsigned

Operands: \( w \) bits

\[ u \quad \begin{array}{|c|c|c|c|c|c|c|} \hline & & & & & \strut \end{array} \]

* \( 2^k \)

True Product: \( w+k \) bits

\[ u \cdot 2^k \quad \begin{array}{|c|c|c|c|c|c|c|} \hline & & & & & \strut \end{array} \]

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

\[ (\text{UMult}, u \cdot 2^k) \quad \begin{array}{|c|c|c|c|c|c|c|} \hline & & & & & \strut \end{array} \]

\[ (\text{TMult}, u \cdot 2^k) \quad \begin{array}{|c|c|c|c|c|c|c|} \hline & & & & & \strut \end{array} \]

Examples

\[ u \ll 3 \quad == \quad u \cdot 8 \]
\[ (u \ll 5) - (u \ll 3) \quad == \quad u \cdot 24 \]

Signed vs. Unsigned in C

Casting: bits unchanged, just interpreted differently.

\[ \text{int } tx, ty; \]
\[ \text{unsigned } ux, uy; \]

Explicit casting:

\[ tx = (\text{int}) ux; \]
\[ uy = (\text{unsigned}) ty; \]

Implicit casting via assignments and function calls:

\[ tx = ux; \]
\[ uy = ty; \]

gcc flag -Wsign-conversion warns about implicit casts; -Wall does not!

Signed vs. Unsigned in C

Constants

By default are considered to be signed integers

Use "U" suffix to force unsigned:

\[ 0U, 4294967259U \]

C 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 \( W = 32 \):  

\[ \text{TMIN} = -2,147,483,648 \quad \text{TMAX} = 2,147,483,647 \]

<table>
<thead>
<tr>
<th>Constant\text{(_1)}</th>
<th>Constant\text{(_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>unsigned</td>
</tr>
<tr>
<td>2147483647</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>