The Hardware/Software Interface
CSE351 Winter 2013

Integers

Today’s Topics
- Representation of integers: unsigned and signed
- Casting
- Arithmetic and shifting
- Sign extension

But before we get to integers…
- How about encoding 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:
    - 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
  - "Two-hot" (?) encoding
  - Easier to compare suits and values
  - Still an excessive number of bits

Two better representations

- Binary encoding of all 52 cards – only 6 bits needed
  - Fits in one byte
  - Smaller than one-hot or two-hot encoding, but 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

Some basic operations

- Checking if two cards have the same suit:
  - \#define SUIT_MASK 0x30
  - char array[5]; // represents a 5 card hand
  - char card1, card2; // two cards to compare
  - card1 = array[0];
  - card2 = array[1];
  - \( \text{SUIT\_MASK} = 0x30; \)
  - \( \begin{array}{c}
  0 & 0 & 1 & 1 & 0 & 0 & 0 \\
  \end{array} \)
  - if sameSuitP(card1, card2) {
    - suit
    - value
  - }

bool sameSuitP(char card1, char card2) {
  return !( (card1 & SUIT\_MASK) ^ (card2 & SUIT\_MASK));
  //return (card1 & SUIT\_MASK) == (card2 & SUIT\_MASK);
}

- Comparing the values of two cards:
  - \#define SUIT\_MASK 0x30
  - \#define VALUE\_MASK 0x0F
  - char array[5]; // represents a 5 card hand
  - char card1, card2; // two cards to compare
  - card1 = array[0];
  - card2 = array[1];
  - \( \text{VALUE\_MASK} = 0x0F; \)
  - \( \begin{array}{c}
  0 & 0 & 0 & 1 & 1 & 1 \\
  \end{array} \)
  - if greaterValue(card1, card2) {
    - suit
    - value
  - }

bool greaterValue(char card1, char card2) {
  return ((unsigned int)(card1 & VALUE\_MASK) >
    (unsigned int)(card2 & VALUE\_MASK));
}
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’t represent all the integers
  -Unsigned values are $0 \ldots 2^W - 1$
  -Signed values are $-2^{W-1} \ldots 2^{W-1} - 1$

Reminder: terminology for binary representations:

"Most-significant" or "high-order" bit(s)  "Least-significant" or "low-order" bit(s)

011001011010100

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: $1+2+4+8+\ldots+2^{W-1} = 2^W - 1$

- You add/subtract them using the normal “carry/borrow” rules, just in binary

Signed Integers

- Let’s do the natural thing for the positives
  - They correspond to the unsigned integers of the same value
    - Example (8 bits): Ox00 = 0, Ox01 = 1, ..., Ox7F = 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):
    - Ox00 = 00000000, is non-negative, because the sign bit is 0
    - Ox7F = 11111111, is non-negative
    - Ox85 = 10000101, is negative
    - Ox80 = 10000000, is negative...

Sign-and-Magnitude Negatives

- How should we represent -1 in binary?
  - Sign-and-magnitude: 10000001,
    Use the MSB for + or -, and the other bits to give magnitude
Sign-and-Magnitude Negatives

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

Two’s Complement Negatives

- How should we represent -1 in binary?
  - Rather than a sign bit, let MSB have same value, but negative weight
    - W-bit word: Bits 0, 1, ..., W-2 add $2^0$, $2^1$, ..., $2^{W-2}$ to value of integer
    when set, but bit W-1 adds $-2^{W-1}$ when set
    - e.g. unsigned 1010: $1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10_{10}$
      2’s comp. 1010: $-1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = -6_{10}$
  - So -1 represented as 1111; all negative integers still have MSB = 1
  - Advantages of two’s complement:
    - only one zero, simple arithmetic
  - To get negative representation of any integer, take bitwise complement and then add one!
    $$\neg x + 1 = -x$$

Sign-and-Magnitude Negatives

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

Two’s Complement Arithmetic

- The same addition procedure works for both unsigned and two’s complement integers
  - Simplifies hardware: only one adder needed
  - Algorithm: simple addition, discard the highest carry bit
    - Called “modular” addition: result is sum modulo $2^W$

- Examples:
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
<td>+ 3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>= 7</td>
</tr>
<tr>
<td>-4</td>
<td>1100</td>
<td>+ 3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>- 4</td>
</tr>
<tr>
<td>- 4</td>
<td>1100</td>
<td>1</td>
</tr>
<tr>
<td>- 4</td>
<td>1100</td>
<td>drop carry</td>
</tr>
<tr>
<td>- 1</td>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>
Two’s Complement

Why does it work?
- Put another way: given the bit representation of a positive integer, we want the negative bit representation to always sum to 0 (ignoring the carry-out bit) when added to the positive representation.
- This turns out to be the *bitwise complement plus one*.
  - What should the 8-bit representation of -1 be?
    - (we want whichever bit string gives the right result)
      - \[
        \begin{array}{c}
          00000000 \\
          +\text{????????} \\
          \hline
          00000000
        \end{array}
      \]
- What should the 8-bit representation of -1 be?
  - (we want whichever bit string gives the right result)
    - \[
      \begin{array}{c}
        00000000 \\
        +\text{11111111} \\
        \hline
        00000000
      \end{array}
    \]
- What should the 8-bit representation of -1 be?
  - (we want whichever bit string gives the right result)
    - \[
      \begin{array}{c}
        00000000 \\
        +\text{11111111} \\
        \hline
        00000000
      \end{array}
    \]
Same W bits interpreted as signed vs. unsigned:

- Two's complement (signed) addition: x and y are W bits wide

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

Two’s Complement Values
- $T_{\text{Min}} = -2^{w-1}$
- $T_{\text{Max}} = 2^{w-1} - 1$

Observations
- $|T_{\text{Min}}| = T_{\text{Max}} + 1$
- Asymmetric range
- $U_{\text{Max}} = 2 \times T_{\text{Max}} + 1$

C Programming
- #include <limits.h>
- Declares constants, e.g.: ULONG_MAX, LONG_MAX, LONG_MIN
- Values are platform specific
- See: /usr/include/limits.h on Linux

Values for different word sizes:

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>$U_{\text{Max}}$</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>$T_{\text{Max}}$</td>
<td>127</td>
<td>32,767</td>
<td>2,147,483,647</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>$T_{\text{Min}}$</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,372,036,854,775,808</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, 0x4294967259U

Values to remember:

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{\text{Max}}$</td>
<td>65,535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Max}}$</td>
<td>32,767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Min}}$</td>
<td>-32,768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>0F FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
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!

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
  - Equivalent to multiplying by 2^y (if no bits lost)
- **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
  - Equivalent to dividing by 2^y
  - Correct rounding (towards 0) requires some care with signed numbers

Using Shifts and Masks

- **Extract the 2nd most significant byte of an integer:**
  - First shift, then mask: `(x >> 16) & 0xFF`
    - Argument x
      - 01100010
      - 00111000
      - 00011000
    - Logical >> 2
      - 00011000
    - Arithmetic >> 2
      - 00011000

- **Extract the sign bit of a signed integer:**
  - `(x >> 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 << 31) >> 31 ) & y + ((x) << 31 ) >> 31 ) & z;

Casting Surprises

- **Expression Evaluation**
  - If you mix signed and unsigned in a single expression, then
    - signed values implicitly cast to unsigned
  - Including comparison operations `<=, >=`:
    - Examples for W = 32: `TMIN = 2,147,483,648` `TMAX = 2,147,483,647`
  - Constant_1 Constant_2 Relation Evaluation
    - 0 0U == unsigned
    - -1 0 < signed
    - -1 0U > unsigned
    - 2147483647 -2147483648 > unsigned
    - 2147483647 < unsigned
    - (unsigned)-1 2 > unsigned
    - 2147483647U 2 < unsigned
    - (int) 2147483648U > signed
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:
  \[ X' = x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_0 \]

Sign Extension Example

- Converting from smaller to larger integer data type
- C automatically performs sign extension

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>12345</td>
<td>00000000 01101101</td>
</tr>
<tr>
<td>lx</td>
<td>0000 30 39</td>
<td>00000000 00000000 00110000 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-12345</td>
<td>FF CF C7</td>
</tr>
<tr>
<td>iy</td>
<td>-12345</td>
<td>11111111 11111111 11001111 11000111</td>
</tr>
</tbody>
</table>