More about Binary

9/6/2016
Unsigned vs. Two’s Complement

8-bit example:

1 1 0 0 0 0 0 1 1

$$2^7 + 2^6 + 2^1 + 2^0 = 128 + 64 + 2 + 1 = 195$$

$$-2^7 + 2^6 + 2^1 + 2^0 = -128 + 64 + 2 + 1 = -61$$

Why does two’s complement work this way?
The traditional number line

Addition

... -1 0 1 ...

...
Unsigned ints on the number line

00000000 11111111

0 2^N - 1
Unsigned Integers

• Suppose we had one byte
  • Can represent $2^8$ (256) values
  • If unsigned (strictly non-negative): 0 – 255

252 = 11111100
253 = 11111101
254 = 11111110
255 = 11111111

What if we add one more?

Car odometer “rolls over”.

99999999
00000000
Unsigned Overflow

If we add two N-bit unsigned integers, the answer can’t be more than $2^N - 1$.

\[
\begin{array}{c}
11111010 \\
+ 00001100 \\
\hline
\text{X}00000110
\end{array}
\]

When there should be a carry from the last digit, it is lost. This is called **overflow**, and the result of the addition is incorrect.
In cs31, the number line is a circle

This means that all arithmetic is modular. With 8 bits, arithmetic is mod $2^8$; with N bits arithmetic is mod $2^N$.

$255 + 4 = 259 \mod 256 = 3$
Suppose we want to support negative values too (-127 to 127). Where should we put -1 and -127 on the circle? Why?

A: -127 (11111111)

B: -1 (11111111)

C: Put them somewhere else.
Option B is Two’s Complement

• Borrows nice properties from the number line:

Only one instance of zero, with -1 and 1 on either side of it.

Addition: moves to the right

"Like wrapping number line around a circle"
Does two’s complement, solve the “rolling over” (overflow) problem?

A. Yes, it’s gone.

B. Nope, it’s still there.

C. It’s even worse now.

This is an issue we need to be aware of when adding and subtracting!
Overflow, Revisited

**Unsigned Number Line**

- Danger Zone endpoints of unsigned number line
- 0
- 255

**Signed Number Line**

- Danger Zone endpoints of signed number line
- -128
- -127
- 0
- 127
- 128
If we add a positive number and a negative number, will we have overflow? (Assume they are the same # of bits)

A. Always

B. Sometimes

C. Never
Signed Overflow

• Overflow: IFF the sign bits of operands are the same, but the sign bit of result is different.
  • Not enough bits to store result!

Signed addition (and subtraction):

\[
\begin{align*}
2 + (-1) &= 1 \\
2 + (-2) &= 0 \\
2 + (-4) &= -2
\end{align*}
\]

\[
\begin{align*}
0010 + 1111 &= 10001 \\
0010 + 1110 &= 10000 \\
0010 + 1100 &= 1110
\end{align*}
\]

No chance of overflow here - signs of operands are different!
Signed Overflow

- Overflow: IFF the sign bits of operands are the same, but the sign bit of result is different.
  - Not enough bits to store result!

Signed addition (and subtraction):

\[
\begin{align*}
2 + (-1) &= 1 & 2 + (-2) &= 0 & 2 + (-4) &= -2 & 2 + 7 &= -7 & -2 + (-7) &= 7 \\
0010 & & 0010 & & 0010 & & 0010 & & 1110 \\
+1111 & & +1110 & & +1100 & & +0111 & & +1001 \\
1 0001 & & 1 0000 & & 1110 & & \text{Overflow here!} & & 1 0111
\end{align*}
\]

Operand signs are the same, and they don’t match output sign!
Overflow Rules

• Signed:
  • The sign bits of operands are the same, but the sign bit of result is different.

• Can we formalize unsigned overflow?
  • Need to include subtraction too, skipped it before.
Recall Subtraction Hardware

Negate and add 1 to second operand:
Can use the same circuit for add and subtract:
\[ 6 - 7 = 6 + \neg 7 + 1 \]

input 1 ------------------------------>
input 2 --> possible bit flipper --> ADD CIRCUIT ---> result
possible +1 input --------->

Let’s call this +1 input: “Carry in”
How many of these unsigned operations have overflowed?

4 bit unsigned values (range 0 to 15):

<table>
<thead>
<tr>
<th>carry-in</th>
<th>carry-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition (carry-in = 0)</td>
<td></td>
</tr>
<tr>
<td>9 + 11 = 1001 + 1011 + 0 = 1 0100</td>
<td></td>
</tr>
<tr>
<td>9 + 6  = 1001 + 0110 + 0 = 0 1111</td>
<td></td>
</tr>
<tr>
<td>3 + 6  = 0011 + 0110 + 0 = 0 1001</td>
<td></td>
</tr>
<tr>
<td>Subtraction (carry-in = 1)</td>
<td></td>
</tr>
<tr>
<td>6 − 3  = 0110 + 1100 + 1 = 1 0011</td>
<td></td>
</tr>
<tr>
<td>3 − 6  = 0011 + 1010 + 1 = 0 1101</td>
<td></td>
</tr>
</tbody>
</table>

A. 1  
B. 2  
C. 3  
D. 4  
E. 5
How many of these **unsigned** operations have overflowed?

4 bit unsigned values (range 0 to 15):

<table>
<thead>
<tr>
<th>Addition (carry-in = 0)</th>
<th>carry-in</th>
<th>carry-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 + 11 = 1001 + 1011 + 0 = 1 0100 = 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 + 6 = 1001 + 0110 + 0 = 0 1111 = 15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 + 6 = 0011 + 0110 + 0 = 0 1001 = 9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subtraction (carry-in = 1)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6 − 3 = 0110 + 1100 + 1 = 1 0011 = 3  (\sim3)</td>
<td></td>
</tr>
<tr>
<td>3 − 6 = 0011 + 1010 + 1 = 0 1101 = 13  (\sim6)</td>
<td></td>
</tr>
</tbody>
</table>

A. 1  
B. 2  
C. 3  
D. 4  
E. 5  

**What’s the pattern?**
Overflow Rule Summary

• Signed overflow:
  • The sign bits of operands are the same, but the sign bit of result is different.

• Unsigned: overflow
  • The carry-in bit is different from the carry-out.

<table>
<thead>
<tr>
<th>$C_{in}$</th>
<th>$C_{out}$</th>
<th>$C_{in}$ XOR $C_{out}$</th>
<th>$C_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

So far, all arithmetic on values that were the same size. What if they’re different?
Suppose we have a signed 8-bit value, 00010110 (22), and we want to add it to a signed 4-bit value, 1011 (-5). How should we represent the four-bit value?

A. 1101 (don’t change it)
B. 00001101 (pad the beginning with 0’s)
C. 11111011 (pad the beginning with 1’s)
D. Represent it some other way.
Sign Extension

- When combining signed values of different sizes, expand the smaller to equivalent larger size:

```c
char y=2, x=-13;
short z = 10;

z = z + y;
```

```
0000000000001010
+ 000000010
000000000000010
```

```
0000000000000101
+ 11110011
1111111111110011
```

Fill in **high-order bits** with **sign-bit** value to get same numeric value in larger number of bytes.
Let’s verify that this works

4-bit signed value, sign extend to 8-bits, is it the same value?

0111  ---->  0000 0111  obviously still 7
1010  ---->  1111 1010  is this still -6?

-128 + 64 + 32  + 16  +  8  +  0  +  2  +  0  =  -6  yes!
Operations on Bits

• For these, doesn’t matter how the bits are interpreted (signed vs. unsigned)

• Bit-wise operators (AND, OR, NOT, XOR)

• Bit shifting
## Bit-wise Operators

- bit operands, bit result (interpret as you please)

<table>
<thead>
<tr>
<th>&amp; (AND)</th>
<th></th>
<th>(OR)</th>
<th>~ (NOT)</th>
<th>^ (XOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A &amp; B</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| 01010101 | 01101010 | 10101010 | ~10101111 |
| 01110101 | ^ 01010000 |

| 01010101 | 01101010 | 10101010 | ~10101111 |
| 01110101 | ^ 01010000 |

| 01110101 | 00101010 | 11000011 | 1010100000 |

| 01110101 | 00101010 | 11000011 |

| 01110101 | 00101010 | 11000011 |

| 01110101 | 00101010 | 11000011 |

| 01110101 | 00101010 | 11000011 |

| 01110101 | 00101010 | 11000011 |
More Operations on Bits

- Bit-shift operators: \(<<\) left shift, \(>>\) right shift

```
01010101 << 2  is 01010100
   2 high-order bits shifted out
   2 low-order bits filled with 0

01101010 << 4 is 10100000
01010101 >> 2  is 00010101
01101010 >> 4  is 00000110

10101100 >> 2  is 00101011 (logical shift)
   or 11101011 (arithmetic shift)
```

Arithmetic right shift: fills high-order bits w/sign bit
C automatically decides which to use based on type:
   signed: arithmetic, unsigned: logical
Floating Point Representation

1 bit for sign  |  exponent |  fraction |
8 bits for exponent
23 bits for precision

\[
\text{value} = (-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{(\text{exponent} - 127)}
\]

let's just plug in some values and try it out

0x40ac49ba: 0 10000001 01011000100100110111010
sign = 0 exp = 129 fraction = 2902458

\[
= 1 \times 1.2902458 \times 2^2 = 5.16098
\]

Think of scientific notation: 1.933e-4 = 1.933 \times 10^{-4}
Character Representation

- Represented as one-byte integers using ASCII.
- ASCII maps the range 0-127 to letters, punctuation, etc.
Characters and strings in C

```c
char c = 'J';
char s[6] = "hello";
s[0] = c;
printf("%s\n", s);
```

Will print: Jello

- Character literals are surrounded by single quotes.
- String literals are surrounded by double quotes.
- Strings are stored as arrays of characters.
Discussion question: how can we tell where a string ends?

A. Mark the end of the string with a special character.

B. Associate a length value with the string, and use that to store its current length.

C. A string is always the full length of the array it’s contained within (e.g., char name[20] must be of length 20).

D. All of these could work (which is best?).

E. Some other mechanism (such as?).
What will this snippet print?

cchar c = ‘J’;
cchar s[6] = “hello”;
s[5] = c;
printf(“%s\n”, s);

A. Jello
B. hellJ
C. helloJ
D. Something else, that we can determine.
E. Something else, but we can’t tell what.