CS 31: Intro to Systems
Binary Arithmetic

Kevin Webb
Swarthmore College
January 29, 2015
Reading Quiz
Unsigned Integers

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

$$252 = 11111100$$
$$253 = 11111101$$
$$254 = 11111110$$
$$255 = 11111111$$
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”.
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?

Modular arithmetic: Here, all values are modulo 256.
Unsigned Addition (4-bit)

• Addition works like grade school addition:

```
  1
+ 0110  6
+ 0100  + 4
_____
1010   10
```

Four bits give us range: 0 - 15
Unsigned Addition (4-bit)

• Addition works like grade school addition:

\[
\begin{array}{ccccccc}
1 & 0110 & + & 0100 & + & 1010 & + & 1010 \\
& 6 & + & 4 & + & 12 & + & 10 \\
& 1010 & 10 & & 10110 & 6 \\
\end{array}
\]

^carry out

Four bits give us range: 0 - 15

Overflow!
Suppose we want to support negative values too (-127 to 127). Where should we put -1 and -127 on the circle?

A: -127 (11111111)

B: -1 (11111111)

C: Put them somewhere else.
Signed Magnitude

• One bit (usually left-most) signals:
  • 0 for positive
  • 1 for negative

For one byte:

  1 = 00000001,  -1 = 10000001

Pros: Negation is very simple!
Signed Magnitude

- One bit (usually left-most) signals:
  - 0 for positive
  - 1 for negative

For one byte:

\[ 0 = 00000000 \]

What about 10000000?

Major con: Two ways to represent zero.
Two’s Complement

- Borrow nice property from number line:

0
-1 1

Only one instance of zero!
Implies: -1 and 1 on either side of it.
Two’s Complement

- Borrow nice property from number line:

Only one instance of zero!
Implies: -1 and 1 on either side of it.
Two’s Complement

• Only one value for zero
• With N bits, can represent the range:
  • \(-2^{N-1}\) to \(2^{N-1} - 1\)
• First bit still designates positive (0) / negative (1)

• Negating a value is slightly more complicated:
  
  \[
  1 = 00000001, \quad -1 = 11111111
  \]

From now on, unless we explicitly say otherwise, we’ll assume all integers are stored using two’s complement! This is the standard!
Two’s Compliment

• Each two’s compliment number is now:

\[-2^{n-1}d_{n-1} + 2^{n-2}d_{n-2} + ... + 2^1d_1 + 2^0d_0\]

Note the negative sign on just the first digit. This is why first digit tells us negative vs. positive.
What is 11001 in decimal?

- Each two’s compliment number is now:
  \[-2^{n-1}d_{n-1} + 2^{n-2}d_{n-2} + ... + 2^1d_1 + 2^0d_0\]

A. -2  
B. -7  
C. -9  
D. -25
Two’s Complement Negation

• To negate a value $x$, we want to find $y$ such that $x + y = 0$.

• For $N$ bits, $y = 2^N - x$
Negation Example (8 bits)

- For N bits, $y = 2^N - x$
- Negate 00000010 (2)
  - $2^8 - 2 = 256 - 2 = 254$

- Our wheel only goes to 127!
  - Put -2 where 254 would be if wheel was unsigned.
  - 254 in binary is 11111110

Given 11111110, it’s 254 if interpreted as unsigned and -2 interpreted as signed.
Negation Shortcut

• A much easier, faster way to negate:
  • Flip the bits (0’s become 1’s, 1’s become 0’s)
  • Add 1

• Negate 00101110 (46)
  • $2^8 - 46 = 256 - 46 = 210$
  • 210 in binary is 11010010
Addition & Subtraction

• Addition is the same as for unsigned
  • One exception: different rules for overflow
  • Can use the same hardware for both

• Subtraction is the same operation as addition
  • Just need to negate the second operand...

• $6 - 7 = 6 + (-7) = 6 + (\sim 7 + 1)$
  • $\sim 7$ is shorthand for “flip the bits of 7”
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 --------------------------------> possible bit flipper --> ADD CIRCUIT ---> result

input 2 --> possible +1 input-------->
Overflow, Revisited

Unsigned:

192 128 64

Signed:

-128 -127 0 1 127 128

Danger Zone
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):

\[
\begin{align*}
2 + (-1) &= 1 & 2 + (-2) &= 0 & 2 + (-4) &= -2 \\
0010 + 1111 &= 10001 & 0010 + 1110 &= 10000 & 0010 + 1100 &= 1110
\end{align*}
\]

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):

\[
\begin{array}{cccccc}
2+(-1) = 1 & 2+(-2) = 0 & 2+(-4) = -2 & 2+7 = -7 & -2+(-7) = 7 \\
0010 & 0010 & 0010 & 0010 & 1110 \\
+1111 & +1110 & +1100 & +0111 & +1001 \\
1 0001 & 1 0000 & 1110 & 1001 & 1 0111 \\
\end{array}
\]

Overflow here! 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:

\[ 6 - 7 = 6 + \sim 7 + 1 \]

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):

Addition (carry-in = 0)

<table>
<thead>
<tr>
<th>Operation</th>
<th>4-bit Value 1</th>
<th>4-bit Value 2</th>
<th>Carry-in</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 + 11</td>
<td>1001</td>
<td>1011</td>
<td>0</td>
<td>1 0100</td>
</tr>
<tr>
<td>9 + 6</td>
<td>1001</td>
<td>0110</td>
<td>0</td>
<td>0 1111</td>
</tr>
<tr>
<td>3 + 6</td>
<td>0011</td>
<td>0110</td>
<td>0</td>
<td>0 1001</td>
</tr>
</tbody>
</table>

Subtraction (carry-in = 1)

<table>
<thead>
<tr>
<th>Operation</th>
<th>4-bit Value 1</th>
<th>4-bit Value 2</th>
<th>Carry-in</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 - 3</td>
<td>0110</td>
<td>1100</td>
<td>1</td>
<td>1 0011</td>
</tr>
<tr>
<td>3 - 6</td>
<td>0011</td>
<td>1001</td>
<td>1</td>
<td>0 1101</td>
</tr>
</tbody>
</table>

A. 1  
B. 2  
C. 3  
D. 4  
E. 5
How many of these 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</td>
<td>1001</td>
<td>1011</td>
</tr>
<tr>
<td></td>
<td>+ 0</td>
<td>+ 0</td>
</tr>
<tr>
<td>1001 + 1011 + 0 = 10100</td>
<td>1</td>
<td>0100 = 4</td>
</tr>
<tr>
<td>9 + 6</td>
<td>1001</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td>+ 0</td>
<td>+ 0</td>
</tr>
<tr>
<td>1001 + 0110 + 0 = 1111</td>
<td>0</td>
<td>1111 = 15</td>
</tr>
<tr>
<td>3 + 6</td>
<td>0011</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td>+ 0</td>
<td>+ 0</td>
</tr>
<tr>
<td>0011 + 0110 + 0 = 1001</td>
<td>0</td>
<td>1001 = 9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subtraction (carry-in = 1)</th>
<th>carry-in</th>
<th>carry-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 - 3</td>
<td>0110</td>
<td>1100</td>
</tr>
<tr>
<td></td>
<td>+ 1</td>
<td>+ 1</td>
</tr>
<tr>
<td>0110 + 1100 + 1 = 0011</td>
<td>1</td>
<td>0011 = 3</td>
</tr>
<tr>
<td>3 - 6</td>
<td>0011</td>
<td>1001</td>
</tr>
<tr>
<td></td>
<td>+ 1</td>
<td>+ 1</td>
</tr>
<tr>
<td>0011 + 1001 + 1 = 1101</td>
<td>0</td>
<td>1101 = 13</td>
</tr>
</tbody>
</table>

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

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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

So far, all arithmetic on values that were the same size. What if they’re different?
Suppose I have an 8-bit value, 00010110 (22), and I want to add it to a signed four-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 expanded smaller to equivalent larger size:

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

z = z + y;                z = z + x;
0000000000001010          0000000000000101
+                  +           000000010          11110011
0000000000000010       11111111111110011
```

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>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>01010101</th>
<th>01101010</th>
<th>10101010</th>
<th>~10101111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100001</td>
<td>10111011</td>
<td>01101001</td>
<td>01010000</td>
</tr>
</tbody>
</table>

| 01110101 | 00101010 | 11000011 |
More Operations on Bits

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

\[
01010101 \quad << \quad 2 \quad is \quad 01010100
\]
2 high-order bits shifted out
2 low-order bits filled with 0

\[
01101010 \quad << \quad 4 \quad is \quad 10100000
\]

\[
01010101 \quad >> \quad 2 \quad is \quad 00010101
\]

\[
01101010 \quad >> \quad 4 \quad is \quad 00000110
\]

\[
10101100 \quad >> \quad 2 \quad is \quad 00101011 \text{ (logical shift)}
\]
or \[
11101011 \text{ (arithmetic shift)}
\]

Arithmetic right shift:  fills high-order bits w/sign bit

C automatically decides which to use based on type:
signed: arithmetic, unsigned: logical
Up Next

• Digital Circuits