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”.

00000000
Unsigned Addition (4-bit)

• Addition works like grade school addition:

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

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

- Addition works like grade school addition:

\[
\begin{array}{ccc}
\text{1} & \text{0110} & \text{6} \\
+ & \text{0100} & +4 \\
\hline
\text{1010} & \text{10} \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{1100} & \text{12} \\
+ & \text{1010} & \text{10} \\
\hline
\text{10110} & \text{6} \\
\text{^carry out} & \\
\end{array}
\]

Four bits give us range: 0 - 15

Overflow!
What About Subtraction?

Suppose we want to subtract 5.

- Adding 16 makes us do a complete turn
- Adding 16-5 = 11 will make us land five spaces before the original point

→ This is how we will subtract 5

- Does this look familiar?
Two’s Complement

The Encoding comes from Definition of the 2’s complement of a number:

*2’s complement of an N bit number, \( x \), is its complement with respect to \( 2^N \)*

Can use this to find the bit encoding, \( y \), for the negation of \( x \):

For N bits, \( y = 2^N - x \)

<table>
<thead>
<tr>
<th>X</th>
<th>-X</th>
<th>( 2^4 - X )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>10000 - 0000 = 0000     (only 4 bits)</td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>10000 - 0001 = 1111</td>
</tr>
<tr>
<td>0010</td>
<td>1110</td>
<td>10000 - 0010 = 1110</td>
</tr>
<tr>
<td>0011</td>
<td>1101</td>
<td>10000 - 0011 = 1101</td>
</tr>
</tbody>
</table>
What About Subtraction?

Suppose we want to subtract 5.

- Adding 16 makes us do a complete turn
- Adding $16 - 5 = 11$ will make us land five spaces before the original point

→ This is how we will subtract 5

- This is two’s complement!
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)
  • $2^8 - 2 = 256 - 2 = 254$
  • 254 in binary is 11111110
Negation Example (8 bits)

• For N bits, \( y = 2^N - x \)
• Negate 00000010 (2)
  • \(2^8 - 2 = 256 - 2 = 254\)
  • 254 in binary is 11111110

Given 11111110, it’s 254 if interpreted as unsigned and -2 interpreted as signed.
Negation Example (8 bits)

• For N bits, \( y = 2^N - x \)
• Negate 00000010 (2)
  • \( 2^8 - 2 = 256 - 2 = 254 \)
  • 254 in binary is 11111110

• Negate 00101110 (46)
  • \( 2^8 - 46 = 256 - 46 = 210 \)
  • 210 in binary is 11010010
Negation Shortcut

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

• Negate 00101110 (46)
  • Flip the bits: 11010001
  • Add 1: 11010010
Subtraction Hardware

Negate and add 1 to second operand:
Can use the same circuit for add and subtract:

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

- \( \sim 7 \) is shorthand for “flip the bits of 7”

input 1 -------------------------------\( \sim 7 \) --> possible bit flipper --> ADD CIRCUIT ---> result
input 2 --> possible bit flipper --> possible +1 input------->
Signed Addition & Subtraction

• Addition is the same as for unsigned
  • Can use the same hardware for both

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

• One exception
By using two’s complement, do we still have this value “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!
Signed Addition & Subtraction

• Addition is the same as for unsigned
  • Can use the same hardware for both

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

• One exception: different rules for overflow
Overflow, in More Details

Unsigned

- Danger Zone
- 255
- 0
- 192
- 64
- 128

Signed

- Danger Zone
- -1
- 0
- 1
- -127
- 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!
  - The result will look incorrect

Signed addition (and subtraction):

\[
\begin{array}{ccc}
2+(-1) &=& 1 \\
0010 &+& 0001 &\rightarrow & 1001 \\
\end{array}
\]

\[
\begin{array}{ccc}
2+(-2) &=& 0 \\
0010 &+& 0000 &\rightarrow & 1001 \\
\end{array}
\]

\[
\begin{array}{ccc}
2+(-4) &=& -2 \\
0010 &+& 0001 &\rightarrow & 1001 \\
\end{array}
\]

\[
\begin{array}{ccc}
2+7 &=& -7 \\
0010 &+& 1111 &\rightarrow & 1001 \rightarrow -7 \\
\end{array}
\]

\[
\begin{array}{ccc}
-2+7 &=& 7 \\
1001 &+& 1111 &\rightarrow & 1001 \rightarrow -7 \\
\end{array}
\]

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

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

Signed addition (and subtraction):

<table>
<thead>
<tr>
<th>Expression</th>
<th>Binary</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2+-1=1</td>
<td>0010</td>
<td>0011</td>
</tr>
<tr>
<td>2+-2=0</td>
<td>0010</td>
<td>0000</td>
</tr>
<tr>
<td>2+-4=-2</td>
<td>0010</td>
<td>1110</td>
</tr>
<tr>
<td>2+7=-7</td>
<td>0010</td>
<td>1101</td>
</tr>
<tr>
<td>-2+-7=7</td>
<td>1110</td>
<td>0000</td>
</tr>
</tbody>
</table>

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 + ~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>Carry-in</th>
<th>Carry-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 + 11</td>
<td>1001 + 1011 + 0</td>
<td>1 0100</td>
</tr>
<tr>
<td>9 + 6</td>
<td>1001 + 0110 + 0</td>
<td>0 1111</td>
</tr>
<tr>
<td>3 + 6</td>
<td>0011 + 0110 + 0</td>
<td>0 1001</td>
</tr>
</tbody>
</table>

Subtraction (carry-in = 1)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Carry-in</th>
<th>Carry-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 - 3</td>
<td>0110 + 1100 + 1</td>
<td>1 0011</td>
</tr>
<tr>
<td>3 - 6</td>
<td>0011 + 1010 + 1</td>
<td>0 1101</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>Operation</th>
<th>carry-in</th>
<th>carry-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition (carry-in = 0)</td>
<td>1001 + 1011 + 0 = 1' 0100 = 4</td>
<td></td>
</tr>
<tr>
<td>9 + 11</td>
<td>1001 + 0110 + 0 = 0 1111 = 15</td>
<td></td>
</tr>
<tr>
<td>9 + 6</td>
<td>0011 + 0110 + 0 = 0 1001 = 9</td>
<td></td>
</tr>
<tr>
<td>Subtraction (carry-in = 1)</td>
<td>0110 + 1100 + 1 = 1 0011 = 3</td>
<td></td>
</tr>
<tr>
<td>6 - 3</td>
<td>0011 + 1010 + 1 = 0 1101 = 13</td>
<td></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 signed 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, expand the smaller to equivalent larger size:

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

z = z + y;  // z = z + x;
```

<table>
<thead>
<tr>
<th>z = z + y;</th>
<th>z = z + x;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000001010</td>
<td>0000000000000101</td>
</tr>
<tr>
<td>+ 00000010</td>
<td>+ 11110011</td>
</tr>
<tr>
<td>0000000000000010</td>
<td>11111111111110011</td>
</tr>
</tbody>
</table>

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>

<table>
<thead>
<tr>
<th>01010101</th>
<th>01101010</th>
<th>10101010</th>
<th>~10101111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00100001</td>
<td>&amp; 10111011</td>
<td>^ 01101001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>01110101</th>
<th>00101010</th>
<th>11000011</th>
</tr>
</thead>
</table>
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
Up Next

• C programming
Hello World

<table>
<thead>
<tr>
<th>Python</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td># hello world                                                         // hello world</td>
<td></td>
</tr>
<tr>
<td>import math                                                          #include &lt;stdio.h&gt;</td>
<td></td>
</tr>
<tr>
<td>def main():                                                           int main() {</td>
<td></td>
</tr>
<tr>
<td>print “hello world”</td>
<td>printf(“hello world\n”);</td>
</tr>
<tr>
<td>main()</td>
<td>return 0;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

#: single line comment                                                 //: single line comment
import libname: include Python libraries                              \#include<libname>: include C libraries
Blocks: indentation                                                     Blocks: {   } (indentation for readability)
print: statement to printout string                                    printf: function to print out format string
statement: each on separate line                                       statement: each ends with ;
def main(): : the main function definition                             int main( ) : the main function definition
                                                                  (int specifies the return type of main)
“White Space”

• Python cares about how your program is formatted. Spacing has meaning.

• C compiler does NOT care. Spacing is ignored.
  – This includes spaces, tabs, new lines, etc.
  – Good practice (for your own sanity):
    • Put each statement on a separate line.
    • Keep indentation consistent within blocks.
These are the same program...

```c
#include <stdio.h>

int main() {
    int number = 7;
    if (number > 10) {
        do_this();
    } else {
        do_that();
    }
}
```

```c
#include <stdio.h>

int main() { int number = 7; if (number > 10) { do_this();
    } else {
    do_that();}}
```
Curly Bracket Etiquette

The most important thing is being consistent throughout your program.
Types

• Everything is stored as bits.

• Type tells us how to interpret those bits.

• “What type of data is it?”
  – integer, floating point, text, etc.
Types in C

• All variables have an explicit type!

• You (programmer) must declare variable types.
  – Where: at the beginning of a block, before use.
  – How: `<variable type> <variable name>;`

• Examples:
  
  int humidity;
  float temperature;
  humidity = 20;
  temperature = 32.5
We have to explicitly declare variable types ahead of time? Lame! Python figured out variable types for us, why doesn’t C?

A. C is old.
B. Explicit type declaration is more efficient.
C. Explicit type declaration is less error prone.
D. Dynamic typing (what Python does) is imperfect.
E. Some other reason (explain)
Numerical Type Comparison

Integers (int)
- Example:
  ```c
  int humidity;
  humidity = 20;
  ```
- Only represents integers
- Small range, high precision
- Faster arithmetic
- (Maybe) less space required

Floating Point (float, double)
- Example:
  ```c
  float temperature;
  temperature = 32.5;
  ```
- Represents fractional values
- Large range, less precision
- Slower arithmetic

I need a variable to store a number, which type should I use?

Use the one that fits your specific need best…
Up Next

- more C programming