Integers

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: September 25, 2017 at 10:23
Topics of this Slideset

- Numeric Encodings: Unsigned and two’s complement
- Programming Implications: C promotion rules
- Basic operations:
  - addition, negation, multiplication
  - Consequences of overflow
  - Using shifts to perform power-of-2 multiply/divide
Assume a machine with 32-bit, two’s complement integers.
For each of the following, either:
- Argue that is true for all argument values;
- Give an example where it’s not true.

\[
\begin{align*}
\text{x < 0} & \quad \rightarrow \quad (x*2 < 0) \\
\text{ux >= 0} & \quad \rightarrow \quad (x << 30 < 0) \\
(x \& 7) == 7 & \quad \rightarrow \quad (x << 30 < 0) \\
\text{ux > -1} & \quad \rightarrow \quad -x < -y \\
\text{x > y} & \quad \rightarrow \quad -x < -y \\
\text{x * x >= 0} & \quad \rightarrow \quad x + y > 0 \\
\text{x > 0 && y > 0} & \quad \rightarrow \quad x + y > 0 \\
\text{x >= 0} & \quad \rightarrow \quad -x <= 0 \\
\text{x <= 0} & \quad \rightarrow \quad -x >= 0
\end{align*}
\]
For unsigned integers, we treat all values as non-negative and use *position notation* as with non-negative decimal numbers.

Assume we have a \( w \) length bit string \( X \).

**Unsigned:**  \( B2U_w(X) = \sum_{i=0}^{w-1} X_i \times 2^i \)
Two’s complement is a way of encoding integers, including some positive and negative values. It’s exactly like unsigned except the high order bit is given negative weight.

\[
\text{Two’s complement: } B2T_w(X) = -X_{w-1} \times 2^{w-1} + \sum_{i=0}^{w-2} X_i \times 2^i
\]

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

**Sign Bit:**
For 2’s complement, the most significant bit indicates the sign.
- 0 for nonnegative
- 1 for negative
Encoding Example

\[ x = 15213: \quad 00111011 \quad 01101101 \]
\[ y = -15213: \quad 11000100 \quad 10010011 \]

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td>2</td>
<td>0 0</td>
<td>1 2</td>
</tr>
<tr>
<td>4</td>
<td>1 4</td>
<td>0 0</td>
</tr>
<tr>
<td>8</td>
<td>1 8</td>
<td>0 0</td>
</tr>
<tr>
<td>16</td>
<td>0 0</td>
<td>1 16</td>
</tr>
<tr>
<td>32</td>
<td>1 32</td>
<td>0 0</td>
</tr>
<tr>
<td>64</td>
<td>1 64</td>
<td>0 0</td>
</tr>
<tr>
<td>128</td>
<td>0 0</td>
<td>1 128</td>
</tr>
<tr>
<td>256</td>
<td>1 256</td>
<td>0 0</td>
</tr>
<tr>
<td>512</td>
<td>1 512</td>
<td>0 0</td>
</tr>
<tr>
<td>1024</td>
<td>0 0</td>
<td>1 1024</td>
</tr>
<tr>
<td>2048</td>
<td>1 2048</td>
<td>0 0</td>
</tr>
<tr>
<td>4096</td>
<td>1 4096</td>
<td>0 0</td>
</tr>
<tr>
<td>8192</td>
<td>1 8192</td>
<td>0 0</td>
</tr>
<tr>
<td>16384</td>
<td>0 0</td>
<td>1 16384</td>
</tr>
<tr>
<td>-32768</td>
<td>0 0</td>
<td>1 -32768</td>
</tr>
</tbody>
</table>

Sum \[ 15213 \] \[ -15213 \]

Integers
**Unsigned Values**

\[
\begin{align*}
\text{UMin} &= 0 & 000 \ldots 0 \\
\text{UMax} &= 2^w - 1 & 111 \ldots 1 \\
\end{align*}
\]

**Two’s Complement Values**

\[
\begin{align*}
\text{TMin} &= -2^{w-1} & 100 \ldots 0 \\
\text{TMax} &= 2^{w-1} - 1 & 011 \ldots 1 \\
\end{align*}
\]

**Values for \( w = 16 \)**

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF 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>
Values for Different Word Sizes

<table>
<thead>
<tr>
<th>w</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>255</td>
<td>65,525</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Tmax</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>Tmin</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>

Observations

- \(|TMin| = TMax + 1\)
- \(UMax = 2 \times TMax + 1\)

C Programming

```c
#include <limits.h>
```

Declares various constants: ULONG_MAX, LONG_MAX, LONG_MIN, etc. *The values are platform-specific.*
**Equivalence:** Same encoding for nonnegative values

**Uniqueness:**
- Every bit pattern represents a unique integer value
- Each representable integer has unique encoding

**Can Invert Mappings:**
- inverse of \( B2U(X) \) is \( U2B(X) \)
- inverse of \( B2T(X) \) is \( T2B(X) \)
C allows conversions from signed to unsigned.

```c
short int x = 15213;
unsigned short ux = (unsigned short) x;
short int y = -15213;
unsigned short uy = (unsigned short) y;
```

**Resulting Values:**

- *The bit representation stays the same.*
- Nonnegative values are unchanged.
- Negative values change into (large) positive values.
Signed vs Unsigned in C

Constants

- By default, constants are considered to be signed integers.
- They are unsigned if they have “U” as a suffix: 0U, 4294967259U.

Casting

- Explicit casting between signed and unsigned is the same as U2T and T2U:

```c
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

- Implicit casting also occurs via assignments and procedure calls:

```c
tx = ux;
uy = ty;
```
### Expression Evaluation

- If you mix unsigned and signed in a single expression, signed values implicitly cast to unsigned.
- This includes when you compare using `<`, `>`, `==`, `<=`, `>=`.

<table>
<thead>
<tr>
<th>Const 1</th>
<th>Const 2</th>
<th>Rel.</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td><code>==</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td><code>&lt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483648</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483648</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned) 1</td>
<td>-2</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
</tbody>
</table>
Sign Extension

**Task:** Given a $w$-bit signed integer $x$, convert it to a $w+k$-bit integer with the *same value*.

**Rule:** Make $k$ copies of the sign bit:

$$x' = x_{w-1}, \ldots x_{w-1}, x_{w-2}, \ldots, x_0$$

Why does this work?

- zero-extend extends unsigned values to wider mode
- sign-extend extends signed values to wider mode
Sign Extension Example

```c
short int x = 15213;
int ix = (int) x;
short int y = -15213;
int iy = (int) y;
```

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 3B 6D</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

In converting from smaller to larger signed integer data types, C automatically performs sign extension.
Why Use Unsigned?

Don’t use just to ensure numbers are nonzero.

- Some C compilers generate less efficient code for unsigned.

```c
unsigned i;
for (i=1; i < cnt; i++)
    a[i] += a[i-1]
```

- It’s easy to make mistakes.

```c
for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1]
```

Do use when performing modular arithmetic.

- multiprecision arithmetic
- other esoteric stuff

Do use when you need extra bits of range.
To find the negative of a number in two’s complement form: complement the bit pattern and add 1:

\[ \sim x + 1 = -x \]

Example:

10011101 = 0x9C = \(-99_{10}\)

Complement:

01100010 = 0x62 = 98_{10}

Add 1:

01100011 = 0x63 = 99_{10}

Try it with: 11111111 and 00000000.
## Complement and Increment Examples

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>~x</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>~x+1</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>~0</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>~0+1</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Unsigned Addition

Given two $w$-bit unsigned quantities $u$, $v$, the true sum may be a $w+1$-bit quantity.

Discard the carry bit and treat the result as an unsigned integer.

Thus, unsigned addition implements **modular addition**.

$$UAdd_w(u, v) = (u + v) \mod 2^w$$

$$UAdd_w(u, v) = \begin{cases} 
  u + v & u + v < 2^w \\
  u + v - 2^w & u + v \geq 2^w 
\end{cases}$$
Task: Determine if \( s = \text{UAdd}_w(u, v) = u + v \).

Claim: We have overflow iff:

\[
  s < u \text{ and } s < v.
\]

On the machine, this causes the **carry flag** to be set.
Properties of Unsigned Addition

W-bit unsigned addition is:

- Closed under addition:
  \[ 0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1 \]

- Commutative
  \[ \text{UAdd}_w(u, v) = \text{UAdd}_w(v, u) \]

- Associative
  \[ \text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v) \]

- 0 is the additive identity
  \[ \text{UAdd}_w(u, 0) = u \]

- Every element has an additive inverse
  Let \( \text{UComp}_w(u) = 2^w - u \), then
  \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]
Two’s Complement Addition

Given two \( w \)-bit signed quantities \( u, v \), the true sum may be a \( w+1 \)-bit quantity.

**Discard the carry bit** and treat the result as a two’s complement number.

\[
\text{TAdd}_w(u, v) = \begin{cases} 
  u + v + 2^w & u + v < \text{TMin}_w \text{ (NegOver)} \\
  u + v & \text{TMin}_w < u + v \leq \text{TMax}_w \\
  u + v - 2^w & \text{TMax}_w < u + v \text{ (PosOver)} 
\end{cases}
\]
Two’s Complement Addition

TAdd and UAdd have identical bit-level behavior.

```c
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

This will give `s == t`.
Task: Determine if $s = T\text{Add}_w(u, v) = u + v$.

Claim: We have overflow iff either:

- $u, v < 0$ but $s \geq 0$ (NegOver)
- $u, v \geq 0$ but $s < 0$ (PosOver)

Can compute this as:

$$\text{ovf} = (u<0 == v<0) \&\& (u<0 != s<0);$$

On the machine, this causes the overflow flag to be set.

Why don’t we have to worry about the case where one input is positive and one negative?
Properties of TAdd

**TAdd is Isomorphic to UAdd.**
This is clear since they have identical bit patterns.

\[ \text{Tadd}_w(u, v) = \text{U2T(UAdd}_w(\text{T2U}(u), \text{T2U}(v))) \]

**Two’s Complement under TAdd forms a group.**

- Closed, commutative, associative, 0 is additive identity.
- Every element has an additive inverse:

  Let \( \text{TComp}_w(u) = \text{U2T(UComp}_w(\text{T2U}(u)), \text{then} \text{TAdd}_w(u, \text{UComp}_w(u)) = 0 \)

\[
\text{TComp}_w(u) = \begin{cases} 
-u & u \neq \text{TMin}_w \\
\text{TMin}_w & u = \text{TMin}_w
\end{cases}
\]
Computing the exact product of two $w$-bit numbers $x$, $y$. This is the same for both signed and unsigned.

Ranges:

- **Unsigned:** $0 \leq x \cdot y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$, requires up to $2w$ bits.
- **Two’s comp. min:** $x \cdot y \geq (-2^{w-1}) \cdot (2^w - 1) = -2^{2w-2} + 2^{w-1}$, requires up to $2w - 1$ bits.
- **Two’s comp. max:** $x \cdot y \leq (-2^{w-1})^2 = 2^{2w-2}$, requires up to $2w$ (but only for $TMin^2_w$).

Maintaining the exact result

- Would need to keep expanding the word size with each product computed.
- Can be done in software with “arbitrary precision” arithmetic packages.
Given two $w$-bit unsigned quantities $u$, $v$, the true sum may be a $2w$-bit quantity.

**We just discard the most significant $w$ bits**, treat the result as an unsigned number.

Thus, unsigned multiplication implements *modular multiplication*.

$$\text{UMult}_w(u, v) = (u \times v) \mod 2^w$$
Unsigned Multiplication

```
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
unsigned up = ux * uy;
```

- Truncates product to \(w\)-bit number: \(up = \text{UMult}_w(ux, uy)\)
- Modular arithmetic: \(up = (ux \cdot uy) \mod 2^w\)

Two’s Complement Multiplication

```
int x, y;
int p = x * y;
```

- Compute exact product of two \(w\)-bit numbers \(x, y\).
- Truncate result to \(w\)-bit number: \(p = \text{TMult}_w(x, y)\)
Unsigned Multiplication

\[
\begin{align*}
\text{unsigned } \ ux &= (\text{unsigned}) \ x; \\
\text{unsigned } \ uy &= (\text{unsigned}) \ y; \\
\text{unsigned } \ up &= ux \times uy;
\end{align*}
\]

Two’s Complement Multiplication

\[
\begin{align*}
\text{int } \ x, \ y; \\
\text{int } \ p &= x \times y;
\end{align*}
\]

Relation

- Signed multiplication gives same bit-level result as unsigned.
  - \( up == (\text{unsigned}) \ p \)
A left shift by $k$, is equivalent to multiplying by $2^k$. This is true for both signed and unsigned values.

$$u \ll 1 \rightarrow u \times 2$$
$$u \ll 2 \rightarrow u \times 4$$
$$u \ll 3 \rightarrow u \times 8$$
$$u \ll 4 \rightarrow u \times 16$$
$$u \ll 5 \rightarrow u \times 32$$
$$u \ll 6 \rightarrow u \times 64$$

Compilers often use shifting for multiplication, since shift and add is much faster than multiply (on most machines).

$$u \ll 5 - u \ll 3 = u \times 24$$
Two useful functions on real numbers are the *floor* and *ceiling* functions.

**Definition:** The floor function $\lfloor r \rfloor$, is the greatest integer less than or equal to $r$.

\[
\lfloor 3.14 \rfloor = 3 \\
\lfloor -3.14 \rfloor = -4 \\
\lfloor 7 \rfloor = 7
\]

**Definition:** The ceiling function $\lceil r \rceil$, is the smallest integer greater than or equal to $r$.

\[
\lceil 3.14 \rceil = 4 \\
\lceil -3.14 \rceil = -3 \\
\lceil 7 \rceil = 7
\]
A right shift by $k$, is (approximately) equivalent to dividing by $2^k$, but the effects are different for the unsigned and signed cases. Quotient of unsigned value by power of 2.

$$u \gg k = \left\lfloor \frac{x}{2^k} \right\rfloor$$

Uses logical shift.

<table>
<thead>
<tr>
<th></th>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>15213</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>$y \gg 1$</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>$y \gg 4$</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>$y \gg 8$</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
Quotient of signed value by power of 2.

\[ u \gg k = \lfloor x/2^k \rfloor \]

- Uses arithmetic shift. What does that mean?
- Rounds in wrong direction when \( u < 0 \).

<table>
<thead>
<tr>
<th></th>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>( y \gg 1 )</td>
<td>-7606.5</td>
<td>-7607</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>( y \gg 4 )</td>
<td>-950.8125</td>
<td>-951</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>( y \gg 8 )</td>
<td>-59.4257813</td>
<td>-60</td>
<td>FF C4</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
We’ve seen that right shifting a negative number gives the wrong answer because it rounds away from 0.

\[ x \gg k = \lfloor x/2^k \rfloor \]

We’d really like \( \lceil x/2^k \rceil \) instead.

You can compute this as: \( \lfloor (x + 2^k - 1)/2^k \rfloor \). In C, that’s:

\[
(x + (1<<k) - 1) \gg k
\]

This biases the dividend toward 0.
Properties of Unsigned Arithmetic

Unsigned multiplication with addition forms a **Commutative Ring**.

- Addition is commutative
- Closed under multiplication
  \[ 0 \leq \text{UMult}_w(u, v) \leq 2^w - 1 \]
- Multiplication is commutative
  \[ \text{UMult}_w(u, v) = \text{UMult}_w(v, u) \]
- Multiplication is associative
  \[ \text{UMult}_w(t, \text{UMult}_w(u, v)) = \text{UMult}_w(\text{UMult}_w(t, u), v) \]
- 1 is the multiplicative identity
  \[ \text{UMult}_w(u, 1) = u \]
- Multiplication distributes over addition
  \[ \text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v)) \]
**Isomorphic Algebras**

- Unsigned multiplication and addition: truncate to \( w \) bits
- Two’s complement multiplication and addition: truncate to \( w \) bits

*Both form rings isomorphic to ring of integers mod \( 2^w \)*

**Comparison to Integer Arithmetic**

- Both are rings
- Integers obey ordering properties, e.g.
  
  \[ u > 0 \rightarrow u + v > v \]
  
  \[ u > 0, v > 0 \rightarrow u \cdot v > 0 \]

- These properties are not obeyed by two’s complement arithmetic.
  
  \[ \text{TMax} + 1 \equiv \text{TMin} \]
  
  \[ 15213 \times 30426 \equiv -10030 \] (for 16-bit words)
Assume a machine with 32-bit word size, two’s complement integers.

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- `x < 0` → `((x*2) < 0` False: TMin
- `ux >= 0` → `(x<<30) < 0` True: 0 = UMin
- `(x & 7) == 7` → `(x<<30) < 0` True: \( x_1 = 1 \)
- `ux > -1` False: 0
- `x > y` → `−x < −y` False: \(-1, \text{TMin}\)
- `x * x >= 0` → `x + y > 0` False: \( 30426 \)
- `x > 0 && y > 0` → `−x <= 0` False: \( \text{TMax}, \text{TMax} \)
- `x >= 0` → `−x >= 0` True: \(-\text{TMax} < 0\)
- `x <= 0` False: TMin