### Topics

- **Numeric Encodings**
  - Unsigned & Two’s complement

- **Programming Implications**
  - C promotion rules

- **Basic operations**
  - Addition, negation, multiplication

- **Programming Implications**
  - Consequences of overflow
  - Using shifts to perform power-of-2 multiply/divide

### Encoding Integers

#### Unsigned

\[ B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \]

#### Two’s Complement

\[ B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \]

- C short 2 bytes long

### C Puzzles

- Taken from old exams
- Assume machine with 32 bit word size, two’s complement integers
- For each of the following C expressions, either:
  - Argue that is true for all argument values
  - Give example where not true

<table>
<thead>
<tr>
<th>Expression</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &lt; 0 )</td>
<td>( (x*2 &lt; 0) )</td>
</tr>
<tr>
<td>( ux &gt; = 0 )</td>
<td>( x &lt; = 0 )</td>
</tr>
<tr>
<td>( x &amp; 7 == 7 )</td>
<td>( (x&lt;&lt;30) &lt; 0 )</td>
</tr>
<tr>
<td>( x &gt; -1 )</td>
<td>( -x &lt; -y )</td>
</tr>
<tr>
<td>( x * x &gt;= 0 )</td>
<td>( x &gt; 0 ) &amp; ( y &gt; 0 )</td>
</tr>
<tr>
<td>( x &gt;= 0 )</td>
<td>( -x &lt;= 0 )</td>
</tr>
<tr>
<td>( x &lt;= 0 )</td>
<td>( -x &gt;= 0 )</td>
</tr>
</tbody>
</table>

### Encoding Example (Cont.)

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213: 00111011 01101101</th>
<th>-15213: 11000101 10010011</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sum</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15213</td>
<td>00111011 01101101</td>
</tr>
</tbody>
</table>
Numeric Ranges

Unsigned Values
- \( UMin = 0 \)
- \( 000...0 \)
- \( UMax = 2^w - 1 \)
- \( 111...1 \)

Two’s Complement Values
- \( Tmin = -2^{w-1} \)
- \( 100...0 \)
- \( Tmax = 2^{w-1} - 1 \)
- \( 011...1 \)

Other Values
- Minus 1
- \( 111...1 \)

Values for \( W = 16 \)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>\texttt{FF FF 11111111 11111111}</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>\texttt{7F FF 01111111 11111111}</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>\texttt{80 00 10000000 00000000}</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>\texttt{FF FF 11111111 11111111}</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>\texttt{00 00 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>65,535</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
- \(|TMin| = Tmax + 1\)
  - Asymmetric range

C Programming
- \#include <limits.h>
- K&R App. B11
- Declares constants, e.g.,
  - \texttt{ULONG_MAX}
  - \texttt{LONG_MAX}
  - \texttt{LONG_MIN}
- Values platform-specific

Casting Signed to Unsigned

C Allows Conversions from Signed to Unsigned

\[
\text{short int } x = 15213; \\
\text{unsigned short int } ux = (\text{unsigned short}) x; \\
\text{short int } y = -15213; \\
\text{unsigned short int } uy = (\text{unsigned short}) y;
\]

Equivalence
- Same encodings for nonnegative values

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

\( \Rightarrow \) Can Invert Mappings
- \( \text{B2U}(x) = \text{B2U}^{-1}(x) \)
  - Bit pattern for unsigned integer
- \( \text{B2T}(x) = \text{B2T}^{-1}(x) \)
  - Bit pattern for two’s comp integer

Resulting Value
- No change in bit representation
- Nonnegative values unchanged
  - \( ux = 15213 \)
- Negative values change into (large) positive values
  - \( uy = 50323 \)
Relation between Signed & Unsigned

Two’s Complement

Maintain Same Bit Pattern

\[ x' = 2^{-w+1} - 2^{-w-1} = 2 \times 2^{-1} = 2^w \]

\[ u_x = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases} \]

Signed vs. Unsigned in C

Constants
- By default are considered to be signed integers
- Unsigned if have “U” as suffix
  
  0U, 4294967259U

Casting
- Explicit casting between signed & unsigned same as U2T and T2U
  
  int tx, ty;
  unsigned ux, uy;
  tx = (int) ux;
  uy = (unsigned) ty;
- Implicit casting also occurs via assignments and procedure calls
  
  tx = ux;
  uy = ty;

Casting Surprises

Expression Evaluation
- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations <, >, ==, <=, >=
- Examples for \( W = 32 \)

<table>
<thead>
<tr>
<th>Constant₁</th>
<th>Constant₂</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>==</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483648</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483648</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
**Explanation of Casting Surprises**

2’s Comp. → Unsigned
- Ordering Inversion
- Negative → Big Positive

![Diagram of 2's Complement to Unsigned Conversion]

**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:
- \( X' = \underbrace{x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_0}_{k \text{ copies of MSB}} \)

![Diagram illustrating sign extension]

**Sign Extension Example**

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

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

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

**Justification For Sign Extension**

**Prove Correctness by Induction on \( k \)**
- Induction Step: extending by single bit maintains value

![Diagram illustrating induction step]

- Key observation: \(-2^{w-1} = -2^w + 2^w \)
- Look at weight of upper bits:
  - \( x = -2^{w-1} x_{w-1} \)
  - \( x' = -2^w x_{w-1} + 2^w x_{w-1} = -2^{w-1} x_{w-1} \)
Why Should I Use Unsigned?

Don’t Use Just Because Number Nonzero
- C compilers on some machines generate less efficient code
  unsigned i;
  for (i = 1; i < cnt; i++)
    a[i] += a[i-1];
- Easy to make mistakes
  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 Need Extra Bit’s Worth of Range
- Working right up to limit of word size

Negating with Complement & Increment

Claim: Following Holds for 2’s Complement
\[ \sim x + 1 = -x \]

Complement
- Observation: \[ \sim x + x = 1111...11 = -1 \]

\[ x = 100110101 \]
+ \[ \sim x = 01100010 \]
\[ 11111111 \]

Increment
- \[ \sim x + x + (\sim x + 1) = -x + (\sim x + 1) \]
- \[ \sim x + 1 = -x \]

Warning: Be cautious treating int’s as integers

Comp. & Incr. Examples

\( x = 15213 \)

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>3B 6D</td>
<td>00111011 01101101</td>
<td>15213</td>
</tr>
<tr>
<td>-15214</td>
<td>11000110 10011010</td>
<td>-15214</td>
</tr>
<tr>
<td>-15213</td>
<td>11000101 10010011</td>
<td>-15213</td>
</tr>
<tr>
<td>C4 93</td>
<td>11000101 10010111</td>
<td>C4 93</td>
</tr>
<tr>
<td>00 00</td>
<td>00000000 00000000</td>
<td>0</td>
</tr>
<tr>
<td>FF FF</td>
<td>11111111 11111111</td>
<td>-1</td>
</tr>
<tr>
<td>00 00</td>
<td>00000000 00000000</td>
<td>0</td>
</tr>
</tbody>
</table>

Unsinged Addition

Operands: \( w \) bits

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

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

Standard Addition Function
- Ignores carry output

Implements Modular Arithmetic
\[ s = UAdd_w(u, v) = \begin{cases} 
  u + v & u + v < 2^w \\
  u + v - 2^w & u + v \geq 2^w 
\end{cases} \]
Visualizing Integer Addition

**Integer Addition**
- 4-bit integers $u, v$
- Compute true sum $\text{Add}_4(u, v)$
- Values increase linearly with $u$ and $v$
- Forms planar surface

**Mathematical Properties**

**Modular Addition Forms an Abelian Group**
- Closed under addition
  
- Commutative
  
- Associative
  
- 0 is additive identity
  
- Every element has additive inverse
  
- Signed vs. unsigned addition in C:
    
    ```
    int s, t, u, v;
    s = (int) ((unsigned) u + (unsigned) v);
    t = u + v
    ```

Two's Complement Addition

**Operands**: $w$ bits

**True Sum**: $w+1$ bits

**Discard Carry**: $w$ bits

**TAdd and UAdd have Identical Bit-Level Behavior**
- Will give $s == t$
characterizing TAdd

functionality
- true sum requires \( w+1 \) bits
- drop off MSB
- treat remaining bits as 2's comp. integer

\[ T\text{Add}(u, v) \]

\[
\begin{cases}
> 0 & \text{PosOver} \\
< 0 & \text{NegOver}
\end{cases}
\]

true sum

\[
\begin{array}{c|c}
0 \ldots 1 & 1 \ldots 1 \\
0 \ldots 0 & 0 \ldots 0 \\
0 \ldots 0 & 100\ldots 0 \\
1 \ldots 0 \ldots 0 & 000\ldots 0 \\
1 \ldots 0 \ldots 0 & 100\ldots 0 \\
1 \ldots 0 \ldots 0 & 011\ldots 1 \\
1 \ldots 0 \ldots 0 & 111\ldots 1 \\
\end{array}
\]

TAdd

\[
\begin{cases}
0 & \text{PosOver} \\
10 & \text{NegOver}
\end{cases}
\]

\[
\begin{align*}
T\text{Add}_w(u,v) = & \begin{cases}
\begin{align*}
u + v + 2^w & \text{if} \ u + v < T\text{Min}_w \text{ (NegOver)} \\
u + v & \text{if} \ T\text{Min}_w \leq u + v \leq T\text{Max}_w \\
u + v - 2^w & \text{if} \ T\text{Max}_w < u + v \text{ (PosOver)}
\end{align*}
\end{cases}
\end{align*}
\]

visualizing 2's comp. addition

values
- 4-bit two's comp.
- range from -8 to +7

wraps around
- if sum \( \geq 2^{w-1} \)
  - becomes negative
  - at most once
- if sum \( < -2^{w-1} \)
  - becomes positive
  - at most once

detecting 2's comp. overflow

task
- given \( s = T\text{Add}_w(u, v) \)
- determine if \( s = \text{Add}_w(u, v) \)
- example

```c
int s, u, v;
s = u + v;
```

claim
- overflow iff either:
  - \( u, v < 0, s \geq 0 \) (NegOver)
  - \( u, v \geq 0, s < 0 \) (PosOver)
- \( \text{ovf} = (u<0 == v<0) \& \& (u<0 != s<0) \);

mathematical properties of TAdd

isomorphic algebra to UAdd
- \( T\text{Add}_w(u, v) = U2T(U\text{Add}_w(T2U(u), T2U(v))) \)
  - since both have identical bit patterns

two's complement under TAdd forms a group
- closed, commutative, associative, 0 is additive identity
- every element has additive inverse
  - let \( T\text{Comp}_w(u) = U2T(U\text{Comp}_w(T2U(u)) \)
  - \( T\text{Add}_w(u, T\text{Comp}_w(u)) = 0 \)

\[
T\text{Comp}_w(u) = \begin{cases}
-u & u \neq T\text{Min}_w \\
T\text{Min}_w & u = T\text{Min}_w
\end{cases}
\]
Multiplication

Computing Exact Product of \( w \)-bit numbers \( x, y \)

- Either signed or unsigned

Ranges

- **Unsigned**: \( 0 \leq x \cdot y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1 \)
  - Up to 2w bits
- **Two’s complement min**: \( x \cdot y \geq (-2^{w-1})(2^{w-1}-1) = -2^{2w-2} + 2^{w-1} \)
  - Up to 2w-1 bits
- **Two’s complement max**: \( x \cdot y \leq (2^{w-1})^2 = 2^{2w-2} \)
  - Up to 2w bits, but only for \((TMin_w)^2\)

Maintaining Exact Results

- Would need to keep expanding word size with each product computed
- Done in software by “arbitrary precision” arithmetic packages

Unsigned Multiplication in C

Operands: \( w \) bits

<table>
<thead>
<tr>
<th>( u )</th>
<th>( \cdot )</th>
<th>( v )</th>
</tr>
</thead>
</table>

True Product: \( 2^w \) bits

\[ u \cdot v \]

Discard \( w \) bits: \( w \) bits

\[ \text{UMult}_w(u, v) \]

Standard Multiplication Function

- Ignores high order \( w \) bits

Implements Modular Arithmetic

\[ \text{UMult}_w(u, v) = u \cdot v \mod 2^w \]

Unsigned vs. Signed Multiplication

Unsigned Multiplication

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

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

Two’s Complement Multiplication

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

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

Unsigned vs. Signed Multiplication

Unsigned Multiplication

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

Two’s Complement Multiplication

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

Relation

- Signed multiplication gives same bit-level result as unsigned
- \( up == (\text{unsigned}) \ p \)
Power-of-2 Multiply with Shift

Operation
- $u \ll k$ gives $u \ast 2^k$
- Both signed and unsigned

Examples
- $u \ll 3 = u \ast 8$
- $u \ll 5 - u \ll 3 = u \ast 24$
- Most machines shift and add much faster than multiply
  - Compiler generates this code automatically

Unsigned Power-of-2 Divide with Shift

Quotient of Unsigned by Power of 2
- $u \gg k$ gives $\lfloor u / 2^k \rfloor$
- Uses logical shift

Examples
- $x \gg 1 = 7606.5 \gg 1 = 7606$
- $x \gg 4 = 950.8125 \gg 4 = 950$
- $x \gg 8 = 59.4257813 \gg 8 = 59$

Signed Power-of-2 Divide with Shift

Quotient of Signed by Power of 2
- $x \gg k$ gives $\lfloor x / 2^k \rfloor$
- Uses arithmetic shift
- Rounds wrong direction when $u < 0$
  - Compiler generates this code automatically

Correct Power-of-2 Divide

Quotient of Negative Number by Power of 2
- Want $\lceil x / 2^k \rceil$ (Round Toward 0)
- Compute as $\lceil (x+2^k-1) / 2^k \rceil$
  - In C: $(x + (1<<k)-1) \gg k$
  - Biases dividend toward 0

Case 1: No rounding
- $u \gg k$
- $u \gg k + 2^k-1$
- $\lceil u / 2^k \rceil$

Biased addition of $+2^k-1$ has no effect
Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

Dividend:

\[ x = \begin{array}{|c|c|c|c|c|}
\hline
& \cdots & \cdots & \cdots & 1 \\
\hline
+2^{k-1} & 0 & \cdots & 0 & 1 \\
\hline
\end{array} \]

Divisor:

\[ \begin{array}{|c|c|c|c|c|c|}
\hline
& \cdots & \cdots & 0 & 1 \\
\hline
/ 2^k & 0 & \cdots & 0 & 1 \\
\hline
\end{array} \]

\[ \left\lfloor \frac{x}{2^k} \right\rfloor = \begin{array}{|c|c|c|c|c|}
\hline
& \cdots & \cdots & 1 & 0 \\
\hline
\end{array} \]

Binary Point

Incremented by 1

Biasing adds 1 to final result

Incremented by 1

Properties of Unsigned Arithmetic

Unsigned Multiplication with Addition Forms

Commutative Ring

- Addition is commutative group
- Closed under multiplication
  \[ 0 \leq {\text{UMult}}_u(u, v) \leq 2^w - 1 \]
- Multiplication Commutative
  \[ {\text{UMult}}_u(u, v) = {\text{UMult}}_u(v, u) \]
- Multiplication is Associative
  \[ {\text{UMult}}_u(t, {\text{UMult}}_u(u, v)) = {\text{UMult}}_u({\text{UMult}}_u(t, u), v) \]
- 1 is multiplicative identity
  \[ {\text{UMult}}_u(u, 1) = u \]
- Multiplication distributes over addition
  \[ {\text{UMult}}_u(t, {\text{UAdd}}_u(u, v)) = {\text{UAdd}}_u({\text{UMult}}_u(t, u), {\text{UMult}}_u(t, v)) \]

Properties of Two’s Comp. Arithmetic

Isomorphic Algebras

- Unsigned multiplication and addition
  - Truncating to \( w \) bits
- Two’s complement multiplication and addition
  - Truncating 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.,
  \[ u > 0 \implies u + v > v \]
  \[ u > 0, v > 0 \implies u \cdot v > 0 \]
- These properties are not obeyed by two’s comp. arithmetic

\[ TMax + 1 = = TMin \]

C Puzzle Answers

- Assume machine with 32 bit word size, two’s comp. integers
- \( TMin \) makes a good counterexample in many cases

- \( x < 0 \) \quad \Rightarrow \quad (x*2) < 0 \quad False: \( TMin \)
- \( ux >> 0 \)
- \( x & 7 == 7 \) \quad \Rightarrow \quad (x<<30) < 0 \quad True: \( x_1 = 1 \)
- \( ux > -1 \)
- \( x > y \) \quad \Rightarrow \quad -x < -y \quad False: -1, \( TMin \)
- \( x * x >= 0 \)
- \( x > 0 \&\& y > 0 \) \quad \Rightarrow \quad x + y > 0 \quad False: \( TMax, TMax \)
- \( x >= 0 \) \quad \Rightarrow \quad -x <= 0 \quad True: -TMax < 0
- \( x <= 0 \) \quad \Rightarrow \quad -x >= 0 \quad False: \( TMin \)

- 15213 * 30426 = = -10030 (16-bit words)