15-213
“The course that gives CMU its Zip!”

Integers
Sep 3, 2002

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

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

• \( x < 0 \) \( \Rightarrow (x \times 2) < 0 \)
• \( \text{ux} \geq 0 \)
• \( x \& 7 == 7 \) \( \Rightarrow (x<<30) < 0 \)
• \( \text{ux} > -1 \)
• \( x > y \) \( \Rightarrow -x < -y \)
• \( x \times x \geq 0 \)
• \( x \geq 0 \) \( \& \& y > 0 \) \( \Rightarrow x + y > 0 \)
• \( x \geq 0 \) \( \Rightarrow -x \leq 0 \)
• \( x < 0 \) \( \Rightarrow -x \geq 0 \)

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

Encoding Integers

**Unsigned**

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

**Two’s Complement**

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

```c
short int x = 15213;
short int y = -15213;
```

- C short 2 bytes long

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

Sign Bit
- For 2’s complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

Encoding Example (Cont.)

\( x = 15213: 00111011 01101101 \)
\( y = -15213: 11000100 10010011 \)

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</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>

Sum

\( x = 15213: 00111011 01101101 \)
\( y = -15213: 11000100 10010011 \)

\( x + y = 00111011 01101101 + 11000100 10010011 = 01111111 11111111 \)

\( x = 15213: 00111011 01101101 \)
\( y = -15213: 11000100 10010011 \)

\( x + y = 00111011 01101101 + 11000100 10010011 = 01111111 11111111 \)

\( x = 15213: 00111011 01101101 \)
\( y = -15213: 11000100 10010011 \)

\( x + y = 00111011 01101101 + 11000100 10010011 = 01111111 11111111 \)
### Numeric Ranges

**Unsigned Values**
- **U\text{Min} = 0**
  - 000...0
- **U\text{Max} = 2\text{w} - 1**
  - 111...1

**Two’s Complement Values**
- **T\text{Min} = -2^{\text{w}-1}**
  - 100...0
- **T\text{Max} = 2^{\text{w}-1} - 1**
  - 011...1

**Other Values**
- **Minus 1**
  - 111...1

### 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>U\text{Max}</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>T\text{Max}</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>T\text{Min}</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
- | T\text{Min} | = | T\text{Max} + 1 |
  - Asymmetric range
- **U\text{Max} = 2 * T\text{Max} + 1**

### C Programming
- `#include <limits.h>`
- K&R App. B11
- Declares constants, e.g.,
  - U\text{LONG_MAX}
  - L\text{ONG_MAX}
  - L\text{ONG_MIN}
- Values platform-specific

### Casting Signed to Unsigned

#### C Allows Conversions from Signed to Unsigned

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

#### 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

<table>
<thead>
<tr>
<th>x</th>
<th>T2B</th>
<th>B2U</th>
<th>ux</th>
</tr>
</thead>
</table>

Maintain Same Bit Pattern

\[ x = x - 2^w \]

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

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

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>Constant_1</th>
<th>Constant_2</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>signed</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>unsigned</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

#### 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:
  \[ X' = x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_0 \]
- $k$ copies of MSB

**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>

- 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

\[
X' = \begin{cases} 
-2^{w-1} & \text{if } x > 0 \\
0 & \text{if } x = 0 \\
-2^{w-1} x_{w-1} + 2^{w-1} x_{w-2} & \text{if } x < 0 
\end{cases}
\]

- **Key observation:**
  \[ -2^{w-1} = -2^w + 2^{w-1} \]
- **Look at weight of upper bits:**
  \[
  X' = -2^w x_{w-1} + 2^{w-1} x_{w-2}
  \]
Why Should I Use Unsigned?

Don’t Use Just Because Number Nonzero
- C compilers on some machines generate less efficient code
  ```c
  unsigned i;
  for (i = 1; i < cnt; i++)
    a[i] += a[i-1];
  ```
- 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 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

- \( x + 1 = -x \)

Complement
- Observation: \( x + x = 1111...11 = -1 \)
  ```c
  x 100111101
  + ~x 01100010
  = -1 11111111
  ```

Increment
- \( ~x + x + (-x + 1) = -x + (-x + 1) \)
- \( ~x + 1 = -x \)

Warning: Be cautious treating int’s as integers

Comp. & Incr. Examples

<table>
<thead>
<tr>
<th>x = 15213</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>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<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

Operands: \( w \) bits

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

Standard Addition Function
- Ignores carry output

Implements Modular Arithmetic

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

\[
UAdd_w(u, v) = \begin{cases} 
  u + v & u + v < 2^w \\
  u + v - 2^w & u + v \geq 2^w
\end{cases}
\]
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

Visualizing Unsigned Addition

Wraps Around
- If true sum $\geq 2^w$
- At most once

Mathematical Properties

Modular Addition Forms an $\textit{Abelian Group}$
- 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 additive identity
  $\text{UAdd}_w(u, 0) = u$
- Every element has additive inverse
  - Let $\text{UComp}_w(u) = 2^w - u$
  - $\text{UAdd}_w(u, \text{UComp}_w(u)) = 0$

Two’s Complement Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

$\text{TAdd}_w(u, v)$

$\text{TAdd}$ and $\text{UAdd}$ have Identical Bit-Level Behavior
- Signed vs. unsigned addition in C:
  ```c
  int s, t, u, v;
  s = (int) ((unsigned) u + (unsigned) v);
  t = u + v
  ```
  Will give $s == t$
Characterizing TAdd

Functionality
- True sum requires $w+1$ bits
- Drop off MSB
- Treat remaining bits as 2’s comp. integer

True Sum

TAdd Result

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 $s = TAdd_w(u, v)$
- Determine if $s = Add_w(u, v)$
- Example
  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)
- $ovf = (u<0 == v<0) && (u<0 != s<0)$;

Mathematical Properties of TAdd

Isomorphic Algebra to UAdd
- $TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v)))$
- 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 $TComp_w(u) = U2T(UComp_w(T2U(u)))$
  - $TAdd_w(u, TComp_w(u)) = 0$

$$TComp_w(u) = \begin{cases} 
-u & u \neq TMin_w \\
TMin_w & u = TMin_w
\end{cases}$$
### 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**: \( x \cdot y \geq (-2^{w-1})^2(2^{w-1}-1) = -2^{2w-2} + 2^{w-1} \)
  - 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

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

- **True Product**: \( 2^w \) bits
- **Operands**: \( w \)-bit numbers \( u, v \)
- **Discard** \( w \) bits: \( w \)-bit number \( u \cdot v \)

#### Standard Multiplication Function
- Ignores high order \( w \) bits

#### Implements Modular Arithmetic

\[
UMult_w(u, v) = u \cdot v \mod 2^w
\]

---

### Unsigned vs. Signed Multiplication

#### Unsigned Multiplication

- \( \text{unsigned} \ ux = (\text{unsigned}) \ x; \)
- \( \text{unsigned} \ uy = (\text{unsigned}) \ y; \)
- \( \text{unsigned} \ up = ux \cdot uy \)
- **Truncates product to \( w \)-bit number**: \( up = UMult_w(u, v) \)
- **Modular arithmetic**: \( up = ux \cdot uy \mod 2^w \)

#### Two's Complement Multiplication

- \( \text{int} \ x, y; \)
- \( \text{int} \ p = x \cdot y; \)
- **Compute exact product of two \( w \)-bit numbers**: \( x, y \)
- **Truncate result to \( w \)-bit number**: \( p = TMult_w(x, y) \)

---

### Unsigned vs. Signed Multiplication

#### Unsigned Multiplication
- \( \text{unsigned} \ ux = (\text{unsigned}) \ x; \)
- \( \text{unsigned} \ uy = (\text{unsigned}) \ y; \)
- \( \text{unsigned} \ up = ux \cdot uy \)

#### Two's Complement Multiplication

- \( \text{int} \ x, y; \)
- \( \text{int} \ p = x \cdot y; \)
- **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 \( u \times 2^k \)
- Both signed and unsigned

<table>
<thead>
<tr>
<th>Operands:</th>
<th>True Product:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w ) bits</td>
<td>( u \times 2^k ) bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Discard ( k ) bits:</th>
<th>( w ) bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMult(_k(u, 2^k))</td>
<td>TMult(_k(u, 2^k))</td>
</tr>
</tbody>
</table>

**Examples**
- \( u \ll 3 \implies u \times 8 \)
- \( u \ll 5 - u \ll 3 \implies u \times 24 \)
- 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

<table>
<thead>
<tr>
<th>Operands:</th>
<th>Division:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u \gg k )</td>
<td>( u / 2^k )</td>
</tr>
</tbody>
</table>

**Quotient of Signed by Power of 2**
- \( x \gg k \) gives \( \lfloor x / 2^k \rfloor \)
- Uses arithmetic shift
- Rounds arithmetic shift when \( u < 0 \)

**Correct Power-of-2 Divide**

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>dividend</td>
<td>dividend</td>
<td>dividend</td>
<td>dividend</td>
</tr>
</tbody>
</table>

### Signed Power-of-2 Divide with Shift

<table>
<thead>
<tr>
<th>Operands:</th>
<th>Division:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \gg k )</td>
<td>( x / 2^k )</td>
</tr>
</tbody>
</table>

**Case 1: No rounding**
- Dividend: \( x \gg k \)
  - \( +2^k -1 \)
  - \( \lfloor (x + (1<<k -1)) / 2^k \rfloor \)
  - In C: \( x + (1<<k -1) >> k \)
  - Biases dividend toward 0

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y \gg 1 )</td>
<td>-7606.5</td>
<td>7606</td>
<td>1D B6</td>
</tr>
<tr>
<td>( y \gg 4 )</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
</tr>
</tbody>
</table>

**Biasesing has no effect**
Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>[x + 2^k - 1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x / 2^k]</td>
<td>Incremented by 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>[/ 2^k]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x / 2^k]</td>
<td>Incremented by 1</td>
</tr>
</tbody>
</table>

Biasing adds 1 to final result

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
  \[\text{UMult}_u(t, \text{UMult}_u(u, v)) = \text{UMult}_u(t, \text{UMult}_u(u, v))\]
- 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.,
  \[u > 0 \Rightarrow u + v > v\]
  \[u > 0, v > 0 \Rightarrow u \cdot v > 0\]
- These properties are not obeyed by two’s comp. arithmetic
  \[T_{\text{Max}} + 1 = T_{\text{Min}}\]

C Puzzle Answers

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

- \(x < 0 \Rightarrow ((x * 2) < 0)\)
  False: \(T_{\text{Min}}\)

- \(ux >= 0\)
  True: \(U_{\text{Min}}\)

- \(x & 7 == 7\)
  \((x << 30) < 0\)
  True: \(x_i = 1\)

- \(ux > -1\)
  False: 0

- \(x > y\)
  \(-x < -y\)
  False: \(-1, T_{\text{Min}}\)

- \(x * x >= 0\)
  False: 30426

- \(x > 0 && y > 0\)
  \(x + y > 0\)
  False: \(T_{\text{Max}}, T_{\text{Max}}\)

- \(x >= 0\)
  \(-x <= 0\)
  True: \(-T_{\text{Max}} < 0\)

- \(x <= 0\)
  \(-x >= 0\)
  False: \(T_{\text{Min}}\)