Encoding Integers

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

**Sign Extension**
- e.g. 32 bit integer -> 64 bit integer
  - Extend sign bit for the higher bits

---

<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>(y)</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

```
short int x = 15213;    /* 2 byte long */
short int y = -15213;
```
Two’s Complement Encoding

**Unsigned**

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

**Two’s Complement**

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

**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>(2^0)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(2^1)</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>(2^2)</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>(2^3)</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>(2^4)</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>(2^5)</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>(2^6)</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>(2^7)</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>(2^8)</td>
<td>256</td>
<td>1</td>
</tr>
<tr>
<td>(2^9)</td>
<td>512</td>
<td>1</td>
</tr>
<tr>
<td>(2^{10})</td>
<td>1024</td>
<td>0</td>
</tr>
<tr>
<td>(2^{11})</td>
<td>2048</td>
<td>1</td>
</tr>
<tr>
<td>(2^{12})</td>
<td>4096</td>
<td>1</td>
</tr>
<tr>
<td>(2^{13})</td>
<td>8192</td>
<td>1</td>
</tr>
<tr>
<td>(2^{14})</td>
<td>16384</td>
<td>0</td>
</tr>
<tr>
<td>(2^{15})</td>
<td>-32768</td>
<td>0</td>
</tr>
</tbody>
</table>

**Sum**

\[ \text{Sum} \quad 15213 \quad -15213 \]

15213: 00111011 01101101

-15213: 11000100 10010011
**Encoding Integers w/ Sign**

- **One’s Complement**
  - e.g., \(7_{10} = 00111_2\) \(-7_{10} = 11000_2\)
  - \(n-n = 0\) !! but non-unique zeros
  - How many positive numbers in N bits?
  - How many negative ones?

- **Two’s Complement**
  - e.g., \(7_{10} = 00111_2\) \(-7_{10} = 11001_2\)
  - \(2^{N-1}\) non-negatives (including zero)
  - \(2^{N-1}\) negatives
  - unique zero representation
  - easy for hardware
    - leading 0 : non-negative
    - leading 1 : negative
Unsigned Values

- $U_{\text{Min}} = 0$
  - 000...0
- $U_{\text{Max}} = 2^w - 1$
  - 111...1

Two’s Complement Values

- $T_{\text{Min}} = -2^{w-1}$
  - 100...0
- $T_{\text{Max}} = 2^{w-1} - 1$
  - 011...1

Other Values

- Minus 1
  - 111...1

Values for $W = 16$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{\text{Max}}$</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Max}}$</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Min}}$</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>-1</td>
<td>FF FF</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>00 00</td>
</tr>
</tbody>
</table>

Decimal Hex Binary
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
</tr>
<tr>
<td>TMax</td>
<td>32767</td>
<td>7F FF</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80 00</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>-1 FF</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>00 00</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
- \(UMax = 2 \times TMax + 1\)

C Programming

- `#include <limits.h>`
- Declares constants, e.g.,
  - ULONG_MAX
  - LONG_MAX
  - LONG_MIN
- Values platform specific
## Unsigned & Signed Numeric Values

<table>
<thead>
<tr>
<th>x</th>
<th>B2U(x)</th>
<th>B2T(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>

- **Equivalence**
  - Same encodings for nonnegative values

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

- **Can Invert Mappings**
  - $U2B(x) = B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - $T2B(x) = B2T^{-1}(x)$
    - Bit pattern for two’s complement integer
Remark: Mappings between unsigned and two’s complement numbers: Keep bit representations and reinterpret
# Mapping Signed ↔ Unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>

T2U

U2T
# Mapping Signed ↔ Unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>
Relation btwn. Signed & Unsigned

Two’s Complement

\[ x \to T2B \to T2U \to B2U \to ux \]

Maintain Same Bit Pattern

Large negative weight

becomes

Large positive weight
Conversion Visualized

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

2’s Complement Range

TMax

TMin

0

−1

−2

UMax

UMax − 1

TMax + 1

TMax

Unsigned Range
Constants

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

Casting

- Explicit casting btwn. 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;
  ```
### Expression Evaluation

- If there is a mix of unsigned and signed in single expression, **signed values implicitly cast to unsigned**
- Including comparison operations `<`, `>`, `==`, `<=`, `>=`
- Examples (W=32):
  
  \[
  \text{TMIN} = -2,147,483,648, \quad \text{TMAX} = 2,147,483,647
  \]

<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>-2147483647-1</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>signed</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>
Summary: Casting Signed ↔ Unsigned

- **Signed and unsigned casting**
  - Bit pattern is maintained
  - But reinterpreted
  - Can have unexpected effects: adding or subtracting $2^w$

  - Expression containing signed and unsigned int
    - `int` is cast to `unsigned` !!!
**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:
  - \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \)

\( k \) copies of MSB

\( X' \)

\( X \)

\( X' \)

\( k \) copies of MSB

\( w \)

\( k \)

\( w \)
Converting from smaller to larger integer data type

C automatically performs sign extension

<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>00 00 3B 6D</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>-15213</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>
Truncating Numbers

- **Truncating a number can alter its value**
  - A form of overflow

- **Truncating an unsigned x** (w bit long) to **x’** (k bit long)
  - Truncating x to k bits is equivalent to computing $x \mod 2^k$
  - $\text{B2U}_k ([x_{k-1}, x_{k-2}, \ldots, x_0]) = \text{B2U}_w ([x_{w-1}, x_{w-2}, \ldots, x_0]) \mod 2^k$

- **Truncating a signed x** (w bit long) to **x’** (k bit long)
  - $\text{B2T}_k ([x_{k-1}, x_{k-2}, \ldots, x_0]) = \text{B2T}_w ([x_{w-1}, x_{w-2}, \ldots, x_0]) \mod 2^k$
  - $= \text{U2T}_k (\text{B2U}_w ([x_{w-1}, x_{w-2}, \ldots, x_0]) \mod 2^k)$

```c
int x = 50323; // 0x0000C493
short int sx = (short) x; // -15213 = -(2^{16} - 50323)
int y = sx; // -15213
```
Summary - Expanding, Truncating

- **Expanding (e.g., short int to int)**
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- **Truncating (e.g., unsigned to unsigned short)**
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behavior
Two’s Complement Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

TAdd and 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 $s == t$
**Functionality**

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

**True Sum**

<table>
<thead>
<tr>
<th>True Sum</th>
<th>TAdd Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^w-1$</td>
<td>011...1</td>
</tr>
<tr>
<td>0</td>
<td>000...0</td>
</tr>
<tr>
<td>$-2^w-1$</td>
<td>100...0</td>
</tr>
<tr>
<td>$-2^w$</td>
<td>1000...0</td>
</tr>
</tbody>
</table>

**Diagram:**
- PosOver: $011...1$
- NegOver: $100...0$
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
Goal: Computing Product of $w$-bit numbers $x$, $y$

- Either signed or unsigned

But, exact results can be bigger than $w$ bits

- Unsigned: up to $2^w$ bits
  - Result range: $0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
- Two’s complement min (negative): Up to $2^w$-1 bits
  - Result range: $x \times y \geq (-2^{w-1}) \times (2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
- Two’s complement max (positive): Up to $2^w$ bits, but only for $(TMin_w)^2$
  - Result range: $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$

So, maintaining exact results...

- would need to keep expanding word size with each product computed
- done in software, if needed
  - e.g., by “arbitrary precision” arithmetic packages
Unsigned Multiplication in C

Operands: $w$ bits

True Product: $2^w$ bits

Discard $w$ bits: $w$ bits

- **Standard Multiplication Function**
  - Ignores high order $w$ bits

- **Implements Modular Arithmetic**
  
  $\text{UMult}_w(u, v) = u \cdot v \mod 2^w$
Signed Multiplication in C

Operands: \( w \) bits

\[ \begin{array}{c}
\text{True Product: } 2^*w \text{ bits } \\
\bullet \bullet \bullet \\
\text{Discard } w \text{ bits: } w \text{ bits }
\end{array} \]

\[ TMult_w(u, v) \]

\[ u \star v \]

\[ u \bullet v \]

\[ u \cdot v \]

\[ \text{True Product: } 2^*w \text{ bits } \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]

\[ TMult_w(u, v) \]

\[ \text{Discard } w \text{ bits: } w \text{ bits } \]

\[ u \]

\[ v \]

\[ u \cdot v \]
## Operation

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

Operands: $w$ bits

True Product: $w+k$ bits

Discard $k$ bits: $w$ bits

---

### Examples

- $u \ll 3 \quad == \quad u \times 8$
- $(u \ll 5) - (u \ll 3) \quad == \quad u \times 24$
- Most machines shift and add 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

**Operands:**

<table>
<thead>
<tr>
<th>Division:</th>
<th>( u \gg k )</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>15213</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>( x \gg 1 )</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>( x \gg 4 )</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>( x \gg 8 )</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
Signed Power-of-2 Mult./Div.

❖ Two’s complement multiplication by a power of 2
  - int \( s \), unsigned \( k \) (\( 0 \leq k < w \))
  - \( s \ll k \) (in C program) yields \( s \cdot 2^k \)

❖ Two’s complement division by a power of 2
  - int \( s \), unsigned \( k \) (\( 0 \leq k < w \))
    - \( s / 2^k \) in C computes quotient using \textbf{round toward zero}
    - rounding down (for \( s \geq 0 \))
      - \( s \gg k \) yields \( \lfloor s / 2^k \rfloor \)
    - rounding up (for \( s < 0 \))
      - \((s + \text{bias}) \gg k \) yields \( \lceil s / 2^k \rceil \)
    - \( \text{bias} = 2^k - 1 = (1 \ll k) - 1 \)

❖ (NOTE) \( \gg \) should be an arithmetic shift-right
Division

❖ Terms

- \( \frac{A}{B} = Q, \ A \mod B = R \)
- \( A = B \times Q + R \)
- **A**: dividend, **B**: divisor, **Q**: quotient, **R**: remainder

❖ Sign relations

- **R** is either zero or the same signed number as **A**
- **B \times Q** is either zero or the same signed number as **A**
- E.g., **A/3, A/-3**

<table>
<thead>
<tr>
<th>A</th>
<th>-8</th>
<th>-7</th>
<th>-6</th>
<th>-5</th>
<th>-4</th>
<th>-3</th>
<th>-2</th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A/3</td>
<td>Q</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>-8</td>
<td>-7</td>
<td>-6</td>
<td>-5</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>A/-3</td>
<td>Q</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Summary – Add, Multiply, Divide

❖ **Add, Multiply**
  - Signed and unsigned computations are the same at bit level behavior

❖ **Power-of-2 Multiply**
  - Unsigned:  \( u << k = u \times 2^k \)
  - Signed:  \( s << k = s \times 2^k \)
  - same bit level behavior

❖ **Power-of-2 divide**
  - Unsigned:  \( u >> k = \lfloor u / 2^k \rfloor \) (logical shift)
  - Signed:  \( s >> k = \lfloor s / 2^k \rfloor \) (s ≥ 0, arithmetic shift)
  \( (s + \text{bias}) >> k = \lceil s / 2^k \rceil \) (s < 0, arithmetic shift)
  \( \text{bias} = 2^k - 1 = (1 << k) - 1 \)