Integers

Topics

- Representations of Integers
  - Basic properties and operations
  - Implications for C

Slides with GREY titles will not be covered in class
Having been through the computer engineering courses, it is expected you are familiar enough with twos-compliment. You can read through the immediately following slides as reference and review. Except note this new information for C:

C Programming

- `#include <limits.h>`
  - K&R App. B11
- Declares constants defining range of builtin variable types, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values are platform-specific, using them increases code portability and readability
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

\begin{tabular}{|c|c|c|c|}
\hline
Decimal & Hex & Binary \\
\hline
x & 15213 & 3B 6D 00111011 01101101 \\
\hline
y & -15213 & C4 93 11000100 10010011 \\
\hline
\end{tabular}

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

<table>
<thead>
<tr>
<th>Sum</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### Numeric Ranges

#### Unsigned Values
- $U_{Min} = 0$
  - 000…0
- $U_{Max} = 2^w - 1$
  - 111…1

#### Two’s Complement Values
- $T_{Min} = -2^{w-1}$
  - 100…0
- $T_{Max} = 2^{w-1} - 1$
  - 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>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>65,535</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>`
  - K&R App. B11
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values platform-specific
Unsigned & Signed Numeric Values

**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 comp integer

<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>
conversions are defined by reinterpretation of the same bits.

- Define mappings between unsigned and two’s complement numbers based on their bit-level representations
### 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 $\leftrightarrow$ 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>

$+16$
Relation between Signed & Unsigned

Two’s Complement

Unsigned

Maintain Same Bit Pattern

\[ ux = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases} \]
Integer and Float Literal Constants

- By default integer numerical literals are considered to be signed integers
- Unsigned if have “U” as suffix
  - 0U, 4294967259U
- Long and unsigned long use suffix L and UL
  - 123L, 123UL
- Floating point numbers can use suffix F or L for long doubles along with a decimal or an E
  - 12.3F, 123E-1F
Signed vs. Unsigned in C

Casting

- Explicit casting between signed & unsigned same as U2T and T2U conversions are defined by dumb *reinterpretation* of the same bits.
  
  ```
  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;
  ```
(Implicit) 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>-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>
Explanation of Casting Surprises

2’s Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive

2’s Comp. Range

Unsigned Range

UMax
UMax - 1

TMax + 1
TMax

TMin

0

-2

-1
0

TMax
Sign Extension (CE Review)

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

\( w \) bits

\( X \)

\( X' \)

\( k \) bits

\( w \) bits
Sign Extension Example

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</td>
<td>15213</td>
<td>3B 6D</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 3B 6D</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93</td>
</tr>
</tbody>
</table>

- Converting from smaller to larger integer data type, C automatically performs sign extension
Why Should I Use Unsigned?

For larger max number, otherwise avoid use to avoid pitfalls

Be aware of pitfalls

- Easy to make mistakes
  ```c
  unsigned i;
  for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];
  ```

- Can be very subtle
  ```c
  #define DELTA sizeof(int)
  int i;
  for (i = CNT; i-DELTA >= 0; i-= DELTA)
    ...,
  ```

Do Use When Performing Modular Arithmetic

- Multiprecision (arbitrary resolution) arithmetic

Do Use When Using Bits to Represent Sets/Lists of 1’s and 0’s

- Logical right shift, no sign extension
Complement & Increment

Claim: Following Holds for 2’s Complement

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

Complement

- Observation: \( \sim x + x == 1111...11_2 == -1 \)

\[ x \begin{array}{c|c|c|c|c} 1 & 0 & 0 & 1 & 1 \\ \hline \sim x \end{array} \]

\[ + \begin{array}{c|c|c|c|c} 0 & 1 & 1 & 0 & 0 \\ \hline \sim x \end{array} \]

\[ \begin{array}{c|c|c|c|c} 1 & 1 & 1 & 1 & 1 \\ \hline -1 \end{array} \]

Increment

- \( \sim x + x == -1 \)
- \( \sim x + x + (\sim x + 1) == -1 + (-x + 1) \)
- \( \sim x + 1 == -x \)

Warning: Be cautious treating int’s as integers

- OK here
## Comp. & Incr. Examples

### $x = 15213$

<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>$\sim x$</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>$\sim x + 1$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011 1</td>
</tr>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

### $0$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>$\sim 0$</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$\sim 0 + 1$</td>
<td>0</td>
<td>0</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Long/short unsigned/signed Conversion Rules

The rules are “simple”:

**Step 1:** extend or truncate
- Going from longer to shorter, upper bits are truncated
- Going from shorter to longer, zero or sign extension is done depending on source type being unsigned or signed respectively

**Step 2:** bit copy
- To/From signed/unsigned is just bit copying, no other smart manipulations or conversions are done

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned long = unsigned long</td>
<td>unsigned long</td>
</tr>
<tr>
<td>unsigned long = signed long</td>
<td>unsigned long</td>
</tr>
<tr>
<td>signed long = unsigned long</td>
<td>signed long</td>
</tr>
<tr>
<td>signed long = signed long</td>
<td>signed long</td>
</tr>
<tr>
<td>unsigned long = unsigned short</td>
<td>unsigned long</td>
</tr>
<tr>
<td>unsigned long = signed short</td>
<td>unsigned long</td>
</tr>
<tr>
<td>signed long = unsigned short</td>
<td>signed long</td>
</tr>
<tr>
<td>signed long = signed short</td>
<td>signed long</td>
</tr>
<tr>
<td>unsigned short = unsigned long</td>
<td>unsigned long</td>
</tr>
<tr>
<td>unsigned short = signed long</td>
<td>signed long</td>
</tr>
<tr>
<td>signed short = unsigned long</td>
<td>signed long</td>
</tr>
<tr>
<td>signed short = signed long</td>
<td>signed long</td>
</tr>
<tr>
<td>unsigned short = unsigned short</td>
<td>unsigned short</td>
</tr>
<tr>
<td>unsigned short = signed short</td>
<td>signed short</td>
</tr>
<tr>
<td>signed short = unsigned short</td>
<td>signed short</td>
</tr>
<tr>
<td>signed short = signed short</td>
<td>signed short</td>
</tr>
</tbody>
</table>
Summary of points

Two compliment addition of two N-bit numbers can require up to N+1 bits to store a full result.

Two’s compliment addition can only overflow if signs of operands are the same (likewise for subtraction the signs must be different).

Result of N-bit addition with overflow is dropping of MSBits’s: \( A+B = (A+B) \mod (2^N) \)

For multiplication, multiplying two N-bit numbers requires up to 2N bits to store the operand. Multiplying a N-bit with a M-bit requires up to N+M bits.
Unsigned Addition

Operands: \( w \) bits

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

**Standard Addition Function**

- Ignores carry output for result in C (though it is stored in carry bit in machine)

**Implements Modular Arithmetic**

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

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

Operands: $w$ bits

$$u \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot$$

$$+ \quad v \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot$$

True Sum: $w+1$ bits

$$u + v \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot$$

Discard Carry: $w$ bits

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

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 $s == t$
TAdd Overflow

Functionality

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

<table>
<thead>
<tr>
<th>True Sum</th>
<th>TAdd Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 111...1</td>
<td>011...1</td>
</tr>
<tr>
<td>0 100...0</td>
<td>000...0</td>
</tr>
<tr>
<td>0 000...0</td>
<td>100...0</td>
</tr>
<tr>
<td>1 011...1</td>
<td>-2(w-1)-1</td>
</tr>
<tr>
<td>1 000...0</td>
<td>-2^w</td>
</tr>
</tbody>
</table>
**Characterizing TAdd**

**Functionality**

- True sum requires $w+1$ bits
- Drop off MSB
- Treat remaining bits as 2’s comp. integer
- Can only overflow if signs of operands are the same

\[
TAdd_w(u, v) = \begin{cases} 
  u + v - 2^{w-1} & u + v < TMin_w \quad (\text{PosOver}) \\
  u + v & TMin_w \leq u + v \leq TMax_w \\
  u + v + 2^{w-1} & TMax_w < u + v \\
\end{cases}
\]
Multiplication

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

- Either signed or unsigned

Ranges

- **Unsigned:** \( 0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1 \)
  - Up to \( 2^w \) bits

- **Two’s complement min:** \( x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^w \)
  - Up to \( 2^{w-1} \) bits

- **Two’s complement max:** \( x \times y \leq (-2^{w-1})^2 = 2^{2w-2} \)
  - Up to \( 2^w \) bits, but only for \((T_{\text{Min}})_w)^2\)

Maintaining Exact Results

- Would need to keep expanding word size (approximately add the number of bits of each operand) with each product computed

- Done in software 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

\[
\begin{array}{c}
\hspace{0.5cm} u \\
\times \hspace{0.5cm} v \\
\hspace{0.5cm} u \cdot v
\end{array}
\]

True Product: \( 2^w \) bits

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

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

**Standard Multiplication Function**

- Ignores high order \( w \) bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same
Power-of-2 Multiply with Shift

Operation

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

Examples

- $u << 3$ == $u \times 8$
- Important “trick” for a CMPE to know:
  - To implement $u \times 24$ can use $u << 4 + u << 3$
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically
Compiled Multiplication Code

C Function

```c
int mul12(int x)
{
    return x * 12;
}
```

Compiled Arithmetic Operations

<table>
<thead>
<tr>
<th>leal (%eax,%eax,2), %eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>sall $2, %eax</td>
</tr>
</tbody>
</table>

Explanation

```c
t <- x + x * 2
return t << 2;
```

- C compiler automatically generates shift/add code when multiplying by constant
Unsigned Power-of-2 Divide with Shift

Quotient of Unsigned by Power of 2

- \( u >> k \) gives \( \lfloor u / 2^k \rfloor \)
- Uses logical shift

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</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 &gt;&gt; 1 )</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>( x &gt;&gt; 4 )</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>( x &gt;&gt; 8 )</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
Compiled Unsigned Division Code

C Function

```c
unsigned udiv8(unsigned x) {
    return x / 8;
}
```

Compiled Arithmetic Operations

```
shrl $3, %eax
```

Explanation

```
# Logical shift
return x >> 3;
```

- Uses logical shift for unsigned
**Signed Power-of-2 Divide with Shift**

**Quotient of Signed by Power of 2**

- \( x >> k \) gives \( \lfloor x / 2^k \rfloor \) instead of the round-towards-zero that standard integer division provides \(-3/2 = -1\)
- Uses arithmetic shift
- Rounds wrong direction when \( u_k < 0 \)

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>( y &gt;&gt; 1)</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>( y &gt;&gt; 4)</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>( y &gt;&gt; 8)</td>
<td>-59.4257813</td>
<td>FF C4</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

Quotient of Negative Number by Power of 2
For negative number,

- Want \[ \left\lfloor \frac{x}{2^k} \right\rfloor \] (Round Toward 0)
- Compute as \[ \left\lfloor \frac{x + 2^k - 1}{2^k} \right\rfloor \]
  
  - In C: `(x + (1<<k)-1) >> k`
  
  - Biases dividend toward 0

Case 1: No rounding

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>[ u ]</th>
<th>[ +2^k - 1 ]</th>
<th>[ \left\lfloor \frac{u + 2^k - 1}{2^k} \right\rfloor ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>1 ... 0</td>
<td>0 ... 01</td>
<td>1 ... 11</td>
</tr>
<tr>
<td>[ +2^k - 1 ]</td>
<td>0 ... 0</td>
<td>0 ... 11</td>
<td>0 ... 01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>[ \left\lfloor \frac{u}{2^k} \right\rfloor ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \left\lfloor \frac{u}{2^k} \right\rfloor ]</td>
<td>1 ... 11 ... 11</td>
</tr>
</tbody>
</table>

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

Case 2: Rounding

Dividend: $x + 2^k - 1$

Divisor: $2^k$

Biasing adds 1 to final result
Compiled Signed Division Code

C Function

```c
int idiv8(int x)
{
    return x/8;
}
```

Compiled Arithmetic Operations

```
testl %eax, %eax
js L4
L3:
sarl $3, %eax
ret
L4:
addl $7, %eax
jmp L3
```

Explanation

```
#add bias for negatives
if x < 0
    x += 7;
# Arithmetic shift
return x >> 3;
```

- Uses arithmetic shift for int
Arbitrary Precision Arithmetic for Longer numbers

For 16 bit calculations, an 8-bit architecture may support double-register arithmetic.

For even longer numbers results can be calculated a piece at a time and overflow bits (add/sum) or overflow registers (multiply) can be used to compute larger results. The built-in C variable types are usually automatically handled by the compiler. If even longer types are needed, find an arbitrary precision arithmetic software library.
Floating Point Math and Fixed-Point Math

If no floating point unit (FPU) is available, you can find a floating point software library. This will likely be slow.

Another option is fixed-point math. You can write or use a library or just do it as needed inline....
Fixed-point arithmetic

Want to add 0011.1110 and 0001.1000

Store 0011.1110 as 00111110 and
Store 0001.1000 as 00011000

Add 00011000 + 00111110 = 01010110

Interpret 01010110 as 0101.0110
Fixed-point arithmetic

Up to you to determine number of bits to use for whole and fraction parts depending on range and precision needed

Then, you get a scalar factor $S$ that converts is to a whole number

$$1101.0000 \times 16 = 11010000 \quad S=16$$
$$01.011000 \times 64 = 01011000 \quad S=64$$
Fixed-point arithmetic

Addition:

A+B computed as A*S+B*S=C*S   Divide result by S
to obtain answer C
(powers of two are efficient choices for S)

Subtraction:

A-B computed as A*S-B*S=C*S   Divide result by S
to obtain answer C
Fixed-point arithmetic

Multiplication:

A*B computed as \((A*S)*(B*S)=C*S^2\)

- Divide result by \(S^2\) to obtain answer \(C\)
- Divide result by \(S\) to obtain scaled answer \(C*S\) which you can use further

Unfortunately, the intermediate result \(C * S^2\) required more storage than the scaled result \(C*2\)
Fixed-point arithmetic

Division:

A/B could be computed as \((A*S)/(B*S)=C\)

Scales cancel. Which is fine if you only wanted an integer answer

Would need to multi by \(S\) to obtain scaled result \(C*S\) for further math

…but this is less accurate since the lower bits have already been lost

Better to prescale one of the operands \(((A*S)*S)/(B*S)=C*S\)

Unfortunately, the intermediate term \(((A*S)*S)\) required more storage
Rounding Errors float to int

float to int always truncates fractional digits, effectively rounding towards zero

5.7 -> 5
-5.7 -> -5

Need Nearest Integer Rounding?
Add +/- .5 before truncation depending on sign

5.7+.5 = 6.2 -> 6
5.4+.5 = 5.9 -> 5

-5.7+.5 = -5.2 -> -5 doesn’t work the same
For negative numbers, need to subtract
-5.7-.5 = -6.2 -> 6
-5.4-.5 = -5.9 -> 5
Rounding errors

A/B gives floor(A/B) rather than round(A/B)

So (A+B/2)/B is how we get rounding with integer-only arithmetic

If B is odd, we need to choose to round B/2 up or down depending on application

Example:

Want to set serial baud rate using clock divider
i.e. BAUD_RATE=CLK_RATE/CLK_DIV

Option 1:
#define CLK_DIV (CLK_RATE/BAUD_RATE)

Option 2: BETTER CHOICE IS
#define CLK_DIV (CLK_RATE+ (BAUD_RATE/2)) /BAUD_RATE
Choices for order of operations

In general, if intermediate result will fit in the allowed integer word size, apply integer multiplications before integer divisions to avoid loss of precision.

Example 1:
int i = 660, x = 54, int y = 23;

want i/x*y : true answer is 255.555...

\[
i/x*y \quad \text{gives 253 good}
\]
\[
i*y/x \quad \text{gives 255 better}
\]
\[
(i*y+x/2)/x \quad \text{gives 256 best}
\]

Example 2:
unsigned int c = 7000, x=10,y=2;

want c*x/y which is truly 35000

c*=x; \quad \text{overflows c since (c*x) > 65535 resulting in x = 4465}
c/=y; \quad \text{get 2232 here}

c/=y;
c*=x; \quad \text{gives 35000 !!!!}

In general, if intermediate result will fit in the allowed integer word size, apply integer multiplications before integer divisions to avoid loss of precision.