Bits, Bytes and Integers

Computer Systems Organization (Spring 2015)
CSCI-UA 201, Section 3

Instructor: Joanna Klukowska
Teaching Assistants: Paige Connelly & Carlos Guzman

Slides adapted from
Andrew Case, Jinyang Li, Mohamed Zahran, Randy Bryant and Dave O’Hallaron
Today: Bits, Bytes and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
Today: Bits, Bytes and Integers

• Representing information as bits
  • Bit-level manipulations

• Integers
  • Representation: unsigned and signed
  • Conversion, casting
  • Expanding, truncating
  • Addition, negation, multiplication, shifting

• Summary
Bits - Digital Revolution

- Digital: bits of 0's and 1's
- Information stored digitally
- Interface between components of the system is digital
- Represented using
  - Punch cards
  - Voltage levels
## Bytes

- **1 byte = 8 bits** is the smallest addressable unit of memory.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 KB (1KiB) = $2^{10}$ bytes = 1,024 bytes</td>
<td>1 KB = $10^3$ bytes = 1,000 bytes</td>
</tr>
<tr>
<td>1MB (1MiB) = $2^{20}$ bytes = 1,048,576 bytes</td>
<td>1 MB = $10^6$ bytes = 1,000,000 bytes</td>
</tr>
<tr>
<td>1GB (1GiB) = $2^{30}$ bytes = 1,073,741,824 bytes</td>
<td>1 GB = $10^9$ bytes = 1,000,000,000 bytes</td>
</tr>
<tr>
<td>1TB (1TiB) = $2^{40}$ bytes = 1,099,511,627,776 bytes</td>
<td>1 TB = $10^{12}$ bytes = 1,000,000,000,000 bytes</td>
</tr>
</tbody>
</table>
Memory

• Program perspective
  • Virtual memory is a (very large) array of bytes
  • Each array location has its own address (like the index into an array)
  • Virtual address space is a set of all possible addresses

• Hardware perspective
  • Memory is a combination of random access memory (RAM), disk storage, other hardware and operating system software that creates an illusion of a monolithic byte array
Base Number Systems

- **Binary** – base 2, symbols: 0, 1
  - Actual representation used in hardware
- **Decimal** – base 10, symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Representation used by humans with 10 fingers
- **Hexadecimal** – base 16, symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - Used as a shorthand notation for binary because binary patterns are too long and decimal system does not correspond well to bit representations
Base Number Systems

Binary - Base 2 (0|1)
Decimal - Base 10 (0–9)
Hexadecimal - Base 16 (0–9|A–F)

<table>
<thead>
<tr>
<th>Bin, Dec, Hex</th>
<th>Bin, Dec, Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000, 0, 0x0</td>
<td>1000, 8, 0x8</td>
</tr>
<tr>
<td>0001, 1, 0x1</td>
<td>1001, 9, 0x9</td>
</tr>
<tr>
<td>0010, 2, 0x2</td>
<td>1010, 10, 0xa</td>
</tr>
<tr>
<td>0011, 3, 0x3</td>
<td>1011, 11, 0xb</td>
</tr>
<tr>
<td>0100, 4, 0x4</td>
<td>1100, 12, 0xc</td>
</tr>
<tr>
<td>0101, 5, 0x5</td>
<td>1101, 13, 0xd</td>
</tr>
<tr>
<td>0110, 6, 0x6</td>
<td>1110, 14, 0xe</td>
</tr>
<tr>
<td>0111, 7, 0x7</td>
<td>1111, 15, 0xf</td>
</tr>
</tbody>
</table>

Homework (DNHI): Convert the following numbers to the other two representations (by hand) and then write a C program that does the same:

Binary: 101101010, 10001110010, 10101011010001, 100001
Decimal: 255, 64, 100, 1025
Hexadecimal: 0xab, 0x123, 0xff, 0xf0
Machine Words

Every computer has a “word size”

- Word size determines the number of bits used to storing a memory address (or a pointer)
  - This determines the maximum size of virtual memory (virtual address space)
- Until recently, most machines used 32-bit (4-byte) words
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- These days, most new systems use 64-bit (8-byte) words
  - Potential address space = 8 Petabytes (that’s 8*1024 Terabytes)
  - x86-64 machines support 48-bit addresses: 256 Terabytes
32-bit Word

- **32-bit machine**: 32 bits are used for addresses
  - There are \(2^{32}\) possible addressable bytes
  - Pointers have to be 4 bytes (i.e. 32 bits) in size
  - What is the largest memory that a 32-bit machine can address?

\[
2^{32} \text{ bytes} = \frac{2^{32} \text{ bytes}}{2^{30} \text{ bytes/GB}} = 2^2 \text{ GB} = 4 \text{ GB}
\]
64-bit Words

• **64-bit machine**: 64 bits are used for addresses
  • There are $2^{64}$ possible addressable bytes
  • Pointers have to be 8 bytes (i.e. 64 bits) in size
  • What is the largest memory that a 64-bit machine can address?

\[
2^{64} \text{ bytes} = \frac{2^{64} \text{ bytes}}{2^{40} \text{ bytes/TB}} = 2^{14} \text{ TB} = 16384 \text{ TB} = 8 \text{ PT}
\]
# Data Type Sizes in C

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

Programs should never assume a specific size of any type!
Byte Ordering
Multi-byte object representation issues

• Where should the bytes be? (consecutive locations or not)
  • Consecutive locations used

• Which byte address should be the address of the object? (first, last, somewhere in the middle)
  • Smallest address is commonly used

• How should the bytes be ordered
  • Big Endian: most Sun & IBM machines
    Least significant byte has highest address
  • Little Endian: most Intel compatible machines
    Least Significant byte has lowest address

• Why does it matter?
  • Machines with different approaches have to “talk” to one another
  • Reading assembly code
Byte Ordering Example

Variable \( x \) has 4-byte representation, it contains \( 0x01234567 \) and its address \( \&x \) is \( 0x100 \).

Big Endian

\[
\begin{array}{c|c|c|c}
0x100 & 0x101 & 0x102 & 0x103 \\
\hline
01 & 23 & 45 & 67 \\
\end{array}
\]

Little Endian

\[
\begin{array}{c|c|c|c}
0x100 & 0x101 & 0x102 & 0x103 \\
\hline
67 & 45 & 23 & 01 \\
\end{array}
\]
Byte Ordering Example
Deciphering Numbers

- Number in decimal: 321560
- Number in hex: 0x4E818
- Pad to 32-bits: 0x0004E818
- Split into bytes: 00 04 E8 18
- Big Endian byte order: 00 04 E8 18
- Little Endian byte order (reverse bytes, not the content of bytes!): 18 E8 04 00

Homework (DNHI): For each of the following decimal numbers show how they would be stored as bytes using Big Endian and Little Endian conventions. Assume that the word size is 32 bits.
5789021, 10, 1587, 989795, 341, 2491
Examining Data Representation

- The following C code can be used to print byte representation of data (casting pointer to unsigned char* creates a byte array)
- See source code examples for use

```c
typedef unsigned char *pointer;

void show_bytes(byte_pointer start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%.2x", start[i]);
    printf("\n");
}

void show_int(int x) {
    show_bytes((byte_pointer) &x, sizeof(int));
}

void show_float(float x) {
    show_bytes((byte_pointer) &x, sizeof(float));
}

void show_pointer(void *x) {
    show_bytes((byte_pointer) &x, sizeof(void *));
}

void show_string(char *x) {
    show_bytes((byte_pointer) x, strlen(x));
}
```
Examining Data Representation using `show_data`

- Running `show_int( 1 )` on a PC with Intel Core i7 processor, produces the following output:
  
  `01 00 00 00`

- Running `show_int( 1 )` on a Sun Solaris with Dual UltraSparc IIIi processor, produces the following output:
  
  `00 00 00 01`

- What does this tell you about the byte ordering used by the two machines?

**Homework (DNHI):** Use the above code to verify the answers from the previous DNHI. Try “deciphering” a negative number. What happens? Are the results as expected?
Today: Bits, Bytes and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
Representing Code

- As with everything else, the code of programs is represented as binary
- Instruction code on different machines is different
  - The binary code generated by different machines is different
  - A program created on one machine cannot, generally, be moved to another machine and run – it needs to be recompiled and rebuilt.
# Boolean Algebra (the basics)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>C bitwise operator</th>
<th>Truth table</th>
<th>C logical operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬</td>
<td>NOT</td>
<td>~</td>
<td>( \sim 0 = 0 )</td>
<td>!</td>
</tr>
<tr>
<td>∧</td>
<td>AND</td>
<td>&amp;</td>
<td>0 &amp; 0 = 0</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>∨</td>
<td>OR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⊕</td>
<td>EXCLUSIVE OR</td>
<td>^</td>
<td>0 ^ 0 = 0</td>
<td></td>
</tr>
</tbody>
</table>
Operations on Bit Vectors

• To operate on vectors of bits, a Boolean operation is applied to bits at corresponding positions

• Example:

  \[ \sim 1100 = 0011 \]

  \[ 0110 \ & \ 1010 = 0010 \]

  \[ 0110 \ | \ 1010 = 1110 \]

  \[ 0110 \ ^\ ^\ 1010 = 1100 \]

• These operators in C are called bitwise operators
Using values as bit-vectors for in-place swap

• Swapping values of two variables normally requires a temporary storage

• Using the bitwise exclusive or operator we can actually do this using only the storage of the two bits

```c
void inplace_swap ( int * x, int * y ) {
    *y = *x ^ *y;
    *x = *x ^ *y;
    *y = *x ^ *y;
}
```

Homework (DNHI): Try it on paper with several different values to convince yourself that this works and how.
**Logical Operations in C**
(are different than the bitwise operations)

- There is no data type to represent Boolean values
- The logical operators `&&`, `||` and `!` view
  - 0 (or a bit vector represented by all zeroes) as **false**
  - anything else as **true**

Their return value is always 0 or 1 (numerical).

- Example:
  - `!0x41 = 0x00`
  - `!0x00 = 0x01`
  - `!!0x41 = 0x01`
  - `0x69 && 0x55 = 0x01`
  - `0x69 || 0x55 = 0x01`
Shift Operations

Left Shift:  \( x << y \)
- Shift bit-vector \( x \) to the left by \( y \) positions
- Throw away extra bits on left
- Fill with 0’s on right
- Equivalent to multiplying by \( 2^y \)
- Undefined Behavior when shift amount \( < 0 \) or \( \geq \) word size

Right Shift:  \( x >> y \)
- Shift bit-vector \( x \) to the right by \( y \) positions
- Throw away extra bits on right
- **Logical shift**
  - Fill with 0’s on left
- **Arithmetic shift**
  - Replicate most significant bit on right
  - Equivalent to dividing by \( 2^y \)
- Undefined Behavior when shift amount \( < 0 \) or \( \geq \) word size
Shift Operations - Examples

- **Example 1:**  
  \(x\) is 01100010  
  \(x \ll 3\) is 00010000  
  (logical) \(x \gg 2\) is 00011000  
  (arith.) \(x \gg 2\) is 00011000

- **Example 2:**  
  \(x\) is 10100010  
  \(x \ll 3\) is 00010000  
  (logical) \(x \gg 2\) is 00101000  
  (arith.) \(x \gg 2\) is 11101000
Today: Bits, Bytes and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
Encoding Integers

- **Unsigned**

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

- **Two's Complement** (Signed)

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B6D 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C493 11000100 10010011</td>
</tr>
</tbody>
</table>

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

- Unsigned
  - \( U_{\text{min}} = 0 \)
  - \( U_{\text{max}} = (2^w) - 1 \)
- Two's Complement
  - \( T_{\text{min}} = -2^{(w-1)} \)
  - \( T_{\text{max}} = 2^{(w-1)} - 1 \)

### 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>FFFF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>( T_{\text{max}} )</td>
<td>32767</td>
<td>7FFF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>( T_{\text{min}} )</td>
<td>-32768</td>
<td>8000</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>(-1)</td>
<td>-1</td>
<td>FFFF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>( 0 )</td>
<td>0</td>
<td>0000</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
**Values for different word sizes**

<table>
<thead>
<tr>
<th></th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>UMax</td>
<td>255</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
</tr>
</tbody>
</table>

- Observe:
  - $|T_{min}| = T_{max} + 1$
  - $U_{max} = T_{max} + 1$
  - Unsigned and two's complements use same encoding for non-negative values
  - Every bit pattern represent a unique integer in its encoding

- In C `limits.h` declares limits for different types:
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
  - ...

These values are platform specific
Today: Bits, Bytes and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
# Mapping Signed <-> Unsigned

Keep bit representation the same, but interpret either as a signed or unsigned value.

<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>
Signed and Unsigned in C

• Constants
  • By default, signed integers
  • Unsigned with “U” as suffix: 0U, 4294967259U

• Casting
  • Explicit casting between signed & unsigned
    ```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;
    ```
Casting surprises

- If there is a mix of unsigned and signed in single expression, **signed values implicitly cast to unsigned**, including expressions with logical comparison operations <, >, ==, <=, >=.

- What will the following code print?

```c
if ( -1 < 0 )
    printf ("-1 < 0 is true" );
else
    printf ("-1 < 0 is false" );

if ( -1 < 0U )
    printf ("-1 < 0U is true" );
else
    printf ("-1 < 0U is false" );
```
Possible security issue
Today: Bits, Bytes and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
Expanding

- Task:
  - Convert \( w \)-bit signed integer \( x \) to \( w+k \)-bit integer with same value

- Solution:
  - Make \( k \) copies of sign bit (0's if the sign bit is zero, 1's if the sign bit is one)
Signed extension - example

C automatically performs signed extension when converting from “smaller” to “larger” data type.

<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>
Truncating

- Example: from int to short (i.e. from 32-bit to 16-bit)
- High-order bits are truncated
- Value is altered and must be reinterpret
- This non-intuitive behavior can lead to buggy code!
Today: Bits, Bytes and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
Today: Bits, Bytes and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
Negation: complement & increment

- Following holds for 2's complement number representation:
  \[ \overline{x} + 1 = -x \]

- Observe that for any number \( x \)
  \[ \overline{x} + x = 1111\ldots111 \equiv -1 \]
## Negation - 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>$\neg x$</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>$\neg x + 1$</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></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>0</td>
<td>0</td>
<td>0000000000 0000000000</td>
</tr>
<tr>
<td>$\neg 0$</td>
<td>-1</td>
<td>FF  FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$\neg 0 + 1$</td>
<td>0</td>
<td>0</td>
<td>0000000000 0000000000</td>
</tr>
</tbody>
</table>
**Unsigned addition**

Operands: $w$ bits

<table>
<thead>
<tr>
<th>$u$</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$</td>
<td>$v$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

True Sum: $w+1$ bits

<table>
<thead>
<tr>
<th>$u$</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$</td>
<td>$v$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Discard Carry: $w$ bits

| $u+v$ |   |   |   |   |

$UAdd_w(u, v)$

- Standard addition function ignores carry output and implements modular arithmetic:

$$s = UAdd(u, v) = (u + v) \mod 2^w$$
# Two's complement addition

Operands: $w$ bits

$$u \quad \cdots \quad \cdots \quad \cdots \quad v$$

True Sum: $w+1$ bits

$$u + v$$

Discard Carry: $w$ bits

$\text{TAddw}(u, v)$

- If $\text{sum} \geq 2^{(w-1)}$, then sum becomes negative (positive overflow)
- If $\text{sum} < -2^{(w-1)}$, then sum becomes positive (negative overflow)
Multiplication

- Task:
  - Computing Exact Product of w-bit numbers x, y (either signed or unsigned)

- Ranges
  - Unsigned multiplication requires up to 2w bits to store result:
    \[ 0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1 \]
  - Two’s complement smallest possible value requires up to 2w-1 bits:
    \[ x \times y \geq (-2^{w-1}) \times (2^{w-1}-1) = -2^{2w-2} + 2^{w-1} \]
  - Two’s complement largest possible value requires up to 2w bits (in one case):
    \[ x \times y \leq (-2^{w-1})^2 = 2^{2w-2} \]

- Maintaining Exact Results
  - Would need to keep expanding word size with each product computed
  - Done in software by “arbitrary precision” arithmetic packages
Unsigned/Signed multiplication

Operands: $w$ bits

True Product: $2w$ bits

Discard $w$ bits: $w$ bits

- Standard multiplication function ignores high order $w$ bits
Power-of-2 multiplication using shift

Operands: \( w \) bits

\[
\begin{array}{c}
u \\
* 2^k\\
\end{array}
\]

True Product: \( w+k \) bits

\[
\begin{array}{c}
u * 2^k\\
\end{array}
\]

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

\[
\begin{array}{c}
\text{UMult}(u, 2k)\\
\end{array}
\]

\[u \ll k \text{ is equivalent to } u * 2^k\]

\[
\begin{align*}
\bullet & \ u \ll 3 \ = \ u * 8 \\
\bullet & \ (u \ll 5) \ - \ (u \ll 3) \ = \ u * 24
\end{align*}
\]

Most machines shift and add faster than multiply

\[
\begin{align*}
\bullet & \ \text{Compiler generates this code automatically}
\end{align*}
\]
Compiled multiplication

```c
int mult_by_12 ( int x ) {
    return x * 12;
}
```

Compiled Arithmetic Operations

leal (%eax,%eax,2), %eax
sadd $2, %eax

Explanation

t ← x + x * 2
return t << 2;

C compiler automatically generates shift/add code when multiplying by a constant.
Unsigned division by power-of-2 using shift

- $u \gg k$ is equivalent to $\text{floor}(u/2^k)$

<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>
Signed division by power-of-2 using shift

Operands:

\[ x \]

\[ / 2^k \]

Division:

\[ x \] \[ / 2^k \]

Result: \( \text{RoundDown}(x / 2^k) \)

- \( x \gg k \) is equivalent to \( \text{floor}(x/2^k) \)
- Problem: incorrect rounding for \( x < 0 \) because arithmetic shift is used

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-15213</td>
<td>C4</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>1</td>
<td>-7606.5</td>
<td>E2</td>
<td>10100010 01001001</td>
</tr>
<tr>
<td>4</td>
<td>-950.8125</td>
<td>FC</td>
<td>101011100 01001001</td>
</tr>
<tr>
<td>8</td>
<td>-59.4257813</td>
<td>FF</td>
<td>101101100 01001001</td>
</tr>
</tbody>
</table>
Trick to correct signed division when shifting

- When $x$ is negative compute
  \[(x + 2^k - 1) / 2^k\]
  instead of
  \[x / 2^k\]
  to get correct rounding.

- This uses property that
  \[
  \text{ceil}(x / y) = \text{floor}( (x+y-1)/y )
  \]
Given the initialization below state if each of the statements to the right is true or not.

(  =>  means “implies”)

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

- \( x < 0 \)  \( \Rightarrow \)  \((x*2) < 0\)
- \( ux >= 0 \)
- \( x & 7 == 7 \)  \( \Rightarrow \)  \((x<<30) < 0\)
- \( ux > -1 \)
- \( x > y \)  \( \Rightarrow \)  \(-x < -y\)
- \( x * x >= 0 \)
- \( x > 0 && y > 0 \)  \( \Rightarrow \)  \(x + y > 0\)
- \( x >= 0 \)  \( \Rightarrow \)  \(-x <= 0\)
- \( x <= 0 \)  \( \Rightarrow \)  \(-x >= 0\)
- \( (x|-x)>>31 == -1\)
- \( ux >> 3 == ux/8\)
- \( x >> 3 == x/8\)
- \( x & (x-1) != 0\)