Bits, Bytes, and Integers

CSE 238/2038/2138: Systems Programming

Instructor:  
Fatma CORUT ERGİN

Slides adapted from Bryant & O’Hallaron’s slides
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
- Representations in memory, pointers, strings
- Summary
Unsigned Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

\[
\begin{align*}
\text{True Sum:} & \quad u + v \\
\text{Discard Carry:} & \quad \text{UAdd}_w(u, v)
\end{align*}
\]

- **Standard Addition Function**
  - Ignores carry output

- **Implements Modular Arithmetic**

\[
s = \text{UAdd}_w(u, v) = u + v \mod 2^w
\]
Visualizing (Mathematical) 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

True Sum

$2^w$

$2^{w+1}$

Modular Sum

Overflow

Overflow
Two’s Complement Addition

Operands: $w$ bits

\[
\begin{array}{c}
\text{u} \\
\hline
+ \text{v} \\
\hline
\text{u + v}
\end{array}
\]

True Sum: $w+1$ bits

\[
\begin{array}{c}
\text{u + v} \\
\hline
\text{TAdd}_w(u, v)
\end{array}
\]

Discard Carry: $w$ bits

\[
\begin{array}{c}
\text{TAdd}_w(u, v) \\
\hline
\text{u + v}
\end{array}
\]

- **TAdd and UAdd have Identical Bit-Level Behavior**
  - Signed vs. unsigned addition in 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

```
True Sum

<table>
<thead>
<tr>
<th></th>
<th>0 111...1</th>
<th>2^{w-1}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 100...0</td>
<td>2^{w-1}-1</td>
</tr>
<tr>
<td></td>
<td>0 000...0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1 011...1</td>
<td>-2^{w-1}</td>
</tr>
<tr>
<td></td>
<td>1 000...0</td>
<td>-2^w</td>
</tr>
</tbody>
</table>
```

```
TAdd Result

<table>
<thead>
<tr>
<th></th>
<th>011...1</th>
<th>000...0</th>
<th>100...0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PosOver</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NegOver</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Visualizing 2’s Complement 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
Multiplication

- **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 $2w$ 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 $2w-1$ bits
    - Result range: $x \times y \geq (-2^{w-1})(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two’s complement max (positive): Up to $2w$ 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
  - is done in software, if needed
    - e.g., by “arbitrary precision” arithmetic packages
Unsigned Multiplication in C

Operands: $w$ bits

True Product: $2w$ 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

True Product: $2w$ bits

Discard $w$ bits: $w$ bits

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

**Operands:** $w$ bits

<table>
<thead>
<tr>
<th>$u$</th>
<th>$2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$0 \cdots 010 \cdots 00$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$u \times 2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \cdots 0 \cdots 0$</td>
</tr>
</tbody>
</table>

**True Product:** $w+k$ bits

**Discard $k$ bits:** $w$ bits

<table>
<thead>
<tr>
<th>$u \ll k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \cdots 0 \cdots 0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$u \times 2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \cdots 0 \cdots 0$</td>
</tr>
</tbody>
</table>

**Examples**
- $u \ll 3$ $\equiv$ $u \times 8$
- $(u \ll 5) - (u \ll 3)$ $\equiv$ $u \times 24$
- 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
  - \( u >> k \) gives \( \lfloor u / 2^k \rfloor \)
  - Uses logical shift

### Division

<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 Power-of-2 Divide with Shift

- Quotient of Signed by Power of 2
  - \( x >> k \) gives \( \lfloor x / 2^k \rfloor \)
  - Uses arithmetic shift
  - Rounds wrong direction if \( x < 0 \)

\[
\begin{array}{ccccccc}
\text{Operands:} & x & \ldots & \ldots & \ldots & \ldots & \ldots \\
\text{Division:} & & & & & & \\
\text{Result:} & \text{RoundDown}(x / 2^k) & & & & & \\
\end{array}
\]

<table>
<thead>
<tr>
<th>( x )</th>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-15213)</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
<td></td>
</tr>
<tr>
<td>(-7606.5)</td>
<td>-7607</td>
<td>E2 49</td>
<td>11100010 01001001</td>
<td></td>
</tr>
<tr>
<td>(-950.8125)</td>
<td>-951</td>
<td>FC 49</td>
<td>11111100 01001001</td>
<td></td>
</tr>
<tr>
<td>(-59.4257813)</td>
<td>-60</td>
<td>FF C4</td>
<td>11111111 11000100</td>
<td></td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

Quotient of Negative Number by Power of 2

- Want \( \lfloor x/2^k \rfloor \) (Round Toward 0)
- Compute as \( \lfloor (x + 2^k - 1)/2^k \rfloor \)
  - In C: \((x + (1<<k) - 1) >> k\)
  - Biases dividend toward 0

Case 1: No rounding

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>1</th>
<th>⋯</th>
<th>0</th>
<th>⋯</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 2^k-1</td>
<td>0</td>
<td>⋯</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>⋯</td>
</tr>
<tr>
<td>1</td>
<td>⋯</td>
<td>1</td>
<td>⋯</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>0</th>
<th>⋯</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>⋯</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>⋯</td>
<td>1</td>
<td>⋯</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Case 2: Rounding

Dividend: $x + 2^k - 1$

Divisor: $\frac{x}{2^k}$

Biasing adds 1 to final result
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
- Representations in memory, pointers, strings
Arithmetic: Basic Rules

- **Addition:**
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod $2^w$
    - Mathematical addition + possible subtraction of $2^w$
  - Signed: modified addition mod $2^w$ (result in proper range)
    - Mathematical addition + possible addition or subtraction of $2^w$

- **Multiplication:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod $2^w$
  - Signed: modified multiplication mod $2^w$ (result in proper range)
Why Should I Use Unsigned?

- *Don’t* use without understanding implications
  - 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)
        . . .
    ```
Counting Down with Unsigned

- **Proper way to use unsigned as loop index**
  
  ```
  unsigned i;
  for (i = cnt-2; i < cnt; i--)
    a[i] += a[i+1];
  ```

- **See Robert Seacord, Secure Coding in C and C++**
  
  - C Standard guarantees that unsigned addition will behave like modular arithmetic
    - \(0 - 1 \rightarrow UMax\)

- **Even better**
  
  ```
  size_t i;
  for (i = cnt-2; i < cnt; i--)
    a[i] += a[i+1];
  ```
  
  - Data type `size_t` defined as unsigned value with length = word size
  - Code will work even if `cnt = UMax`
  - What if `cnt` is signed and < 0?
Why Should I Use Unsigned? (cont.)

- **Do Use When Performing Modular Arithmetic**
  - Multiprecision arithmetic

- **Do Use When Using Bits to Represent Sets**
  - Logical right shift, no sign extension
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
- Representations in memory, pointers, strings
Byte-Oriented Memory Organization

- Programs refer to data by address
  - Conceptually, envision it as a very large array of bytes
    - In reality, it’s not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

- Note: system provides private address spaces to each “process”
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others
Machine Words

Any given computer has a “Word Size”

- Nominal size of integer-valued data
  - and of addresses

- Until recently, most machines used 32 bits (4 bytes) as word size
  - Limits addresses to 4GB ($2^{32}$ bytes)

- Increasingly, machines have 64-bit word size
  - Potentially, could have 18 EB (exabytes) of addressable memory
  - That’s $18.4 \times 10^{18}$

- Machines still support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes
Word-Oriented Memory Organization

- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
# Example Data Representations

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Typical 64-bit</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>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>–</td>
<td>–</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte Ordering

- So, how are the bytes within a multi-byte word ordered in memory?

- Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address
Byte Ordering Example

- **Example**
  - Variable `x` has 4-byte value of 0x01234567
  - Address given by `&x` is 0x100

**Big Endian**

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

**Little Endian**

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
Representing Integers

int A = 15213;

int B = -15213;

long int C = 15213;

Two’s complement representation
Examining Data Representations

- Code to Print Byte Representation of Data
  - Casting pointer to unsigned char * allows treatment as a byte array

```
typedef unsigned char *pointer;

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

Printf directives:
%p: Print pointer
%x: Print Hexadecimal
show_bytes Execution Example

int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));

Result (Linux x86-64):

int a = 15213;
0x7fffffff71dbc 6d
0x7fffffff71dbd 3b
0x7fffffff71dbe 00
0x7fffffff71dbf 00
Representing Pointers

```c
int B = -15213;
int *P = &B;
```

Different compilers & machines assign different locations to objects

Even get different results each time run program
Representing Strings

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character “0” has code 0x30
      - Digit $i$ has code 0x30+$i$
  - String should be null-terminated
    - Final character = 0

- **Compatibility**
  - Byte ordering not an issue

```
char S[6] = "18213";
```
Integer C Puzzles

Initialization

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

- `x < 0 → ((x*2) < 0)` False
- `ux >= 0` True
- `x & 7 == 7 → (x<<30) < 0` True
- `ux > -1` False
- `x > y → -x < -y` False
- `x * x >= 0` False
- `x > 0 && y > 0 → x + y > 0` False
- `x >= 0 → -x <= 0` True
- `x <= 0 → -x >= 0` False
- `(x|−x)>>31 == -1` False
- `ux >> 3 == ux/8` True
- `x >> 3 == x/8` False
- `x & (x−1) != 0` False
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
- Representations in memory, pointers, strings