Bits, Bytes, and Integers

15-213: Introduction to Computer Systems
3rd Lectures, May 28th, 2013

Instructors:
Greg Kesden
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
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:
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$

```
\[ X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \]
```

- Diagram:
  - $X$ is the original $w$-bit integer.
  - $X'$ is the extended $w+k$-bit integer.
  - $k$ copies of the MSB are made.
  - $X'$ has $k$ copies of the MSB at the beginning.

```
\[ k \text{ copies of MSB} \]
```
Sign Extension Example

- Converting from smaller to larger integer data type
- C automatically performs sign extension
Expanding, Truncating: Basic Rules

- **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
Lets run some tests

printf("%d\n", getValue());

- 50652
- 1500
- 9692
- 26076
- 17884
- 42460
- 34268
- 50652
Let's run some tests

```c
int x = getValue(); printf("%d %08x
", x, x);
```

- 50652 0000c5dc
- 1500 000005dc
- 9692 000025dc
- 26076 000065dc
- 17884 000045dc
- 42460 0000a5dc
- 34268 000085dc
- 50652 0000c5dc

Those darn engineers!
Only care about least significant 12 bits

```c
int x=getValue();
x=(x & 0x0fff);
printf("%d\n",x);
```
Only care about least significant 12 bits

```c
int x = getValue();
x = x(&0x0fff);
printf("%d\n", x);
```

`hmm?`

```c
printf("%x\n", x);
```
Must sign extend

```c
int x=getValue();
x=(x&0x00fff) | (x&0x0800?0xfffff000:0);
printf("%d\n",x);
```

There is a better way.
Because you graduated from 213

```c
int x=getValue();
x=(x&0x00fff) | (x&0x0800?0xffffffff:0);
printf("%d\n",x);
```

huh?
int x=getValue();
x=(x&0x00fff) | (x&0x0800?0x0ff0000000:0);
printf("%d\n",x);
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

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

True Sum: \( w+1 \) bits

| \( u + v \) | \( \cdots \) | \( \cdots \) | \( \cdots \) | \( \cdots \) |

Discard Carry: \( w \) bits

| UAdd\(_w\)(\( u \), \( v \)) | \( \cdots \) | \( \cdots \) | \( \cdots \) | \( \cdots \) |

## 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

![Integer Addition Graph](image.png)
Visualizing Unsigned Addition

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

---

**True Sum**
- $2^{w+1}$
- $2^w$
- 0

**Modular Sum**

**Overflow**

$UAdd_4(u, v)$

Diagram showing the visualization of unsigned addition.
Two’s Complement Addition

Operands: $w$ bits

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

True Sum: $w+1$ bits

\[
\text{u} + \text{v}
\]

Discard Carry: $w$ bits

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

- **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

```
0111...1  2^w - 1
0100...0  2^(w-1) - 1
0000...0  0
1011...1  -2^w
1000...0  -2^w
```

- **True Sum**
- **TAdd Result**
  - PosOver
  - NegOver
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 \( 2^w \) 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: \( x \times y \geq (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1} \)
  - 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
  - is 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**

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

### Examples
- \( u \ll 3 \) \( == \) \( u \times 8 \)
- \( u \ll 5 \) \(-\) \( u \ll 3 \) \( == \) \( u \times 24 \)
- 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>$u$</th>
<th>$2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0\ldots 010\ldots 00$</td>
<td></td>
</tr>
</tbody>
</table>

**Division:**

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

**Result:**

<table>
<thead>
<tr>
<th>$\lfloor u / 2^k \rfloor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0\ldots 00\ldots$</td>
</tr>
</tbody>
</table>

<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
  - \( x \gg k \) gives \( \lfloor x / 2^k \rfloor \)
  - Uses arithmetic shift
  - Rounds wrong direction when \( x < 0 \)

![Diagram of division process]

<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 \gg 1 )</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>( y \gg 4 )</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>( y \gg 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**
  - 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**

Dividend:
\[
\begin{array}{c|c|c|c}
| & & & \\
|---|---|---|---|
| u & \cdots & 0 & \cdots 0 0 \\
\hline
+2^k - 1 & 0 & \cdots & 0 0 1 \cdots 1 1 \\
\hline
\end{array}
\]

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

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

Case 2: Rounding

Dividend: \[ x \]

\[ +2^k - 1 \]

\[ \frac{x}{2^k} \]

Divisor: \[ \frac{1}{2^k} \]

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 Just Because Number Nonnegative**
  - 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 arithmetic

- **Do Use When Using Bits to Represent Sets**
  - Logical right shift, no sign extension
Integer C Puzzles

- Assume 32-bit word size, two’s complement integers
- For each of the following C expressions: true or false? Why?

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

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
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
  - Most current machines use 32 bits (4 bytes) as word size
    - Limits addresses to 4GB ($2^{32}$ bytes)
    - Becoming too small for memory-intensive applications
      - leading to emergence of computers with 64-bit word size
  - 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)
For other data representations too ...

<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>
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
    - 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**

```
0x100 0x101 0x102 0x103
```

```
01 23 45 67
```

**Little Endian**

```
0x100 0x101 0x102 0x103
```

```
67 45 23 01
```
Representing Integers

int A = 15213;

long int C = 15213;

int B = -15213;

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3 B 6 D

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

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

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

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

**Result (Linux):**

```c
int a = 15213;
0x11fffffffcb8 0x6d
0x11fffffffcb9 0x3b
0x11fffffffcba 0x00
0x11fffffffcb 0x00
```
Reading Byte-Reversed Listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code

- **Example Fragment**
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00</td>
<td>cmpl $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

- **Deciphering Numbers**
  - Value: 0x12ab
  - Pad to 32 bits: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse: ab 12 00 00
Representing Pointers

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

Different compilers & machines assign different locations to objects
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

```c
char S[6] = "18243";
```
**Code Security Example**

- **SUN XDR library**
  - Widely used library for transferring data between machines

```c
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```

```c
malloc(ele_cnt * ele_size)
```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
        /* malloc failed */
        return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
XDR Vulnerability

malloc(ele_cnt * ele_size)

- What if:
  - ele_cnt = $2^{20} + 1$
  - ele_size = 4096 = $2^{12}$
  - Allocation = ??

- How can I make this function secure?