Bits, Bytes, and Integers

Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating

Binary Representations

Encoding Byte Values

- Byte = 8 bits
- Binary 00000000 to 11111111
- Decimal: 0\(_{10}\) to 255\(_{10}\)
- Hexadecimal 00\(_{16}\) to FF\(_{16}\)
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
  - Write FA1D37B\(_{16}\) in C as
    - 0xFA1D37B
    - 0xFA1D37B
Byte-Oriented Memory Organization

- Programs Refer to Virtual Addresses
  - Conceptually very large array of bytes
  - Actually implemented with hierarchy of different memory types
  - System provides address space private to particular "process"
    - Program being executed
    - Program can clobber its own data, but not that of others
- Compiler + Run-Time System Control Allocation
  - Where different program objects should be stored
  - All allocation within single virtual address space

Machine Words

- Machine Has “Word Size”
  - Nominal size of integer-valued data
    - Including addresses
  - Most current machines use 32 bits (4 bytes) words
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - High-end systems use 64 bits (8 bytes) words
    - Potential address space = 1.8 X 10^19 bytes
    - x86-64 machines support 48-bit addresses: 256 Terabytes
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always one or multiples 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)

Data Representations

<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>1012</td>
<td>1016</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte Ordering

- How should bytes within a multi-byte word be ordered in memory?
- Big Endian (Sun, PPC Mac, Internet)
  - Most significant byte has lowest address
- Little Endian (Intel x86)
  - Least significant byte has lowest address
- Example
  - Variable x has 4-byte representation 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>

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 ab
  - Reverse: ab 12 00 00

Representing Integers

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

Representing Strings

- Strings in C
  - Represented by array of characters
  - Each character encoded in ASCII
    - Standard 7-bit encoding of characters
    - Character "0" has code 0x30
    - Digit / has code 0x30 + i
  - String should be null-terminated
    - Final character = 0
  - Compatibility
    - Byte ordering not an issue

char S[6] = "18243";

Linux/Alpha

<table>
<thead>
<tr>
<th>31</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating

Boolean Algebra

- Algebraic representation of logic
- Encode "True" as 1 and "False" as 0

And

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 1</td>
</tr>
</tbody>
</table>

Or

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

Not

<table>
<thead>
<tr>
<th>~A</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 1</td>
</tr>
<tr>
<td>1</td>
<td>0 0</td>
</tr>
</tbody>
</table>

Exclusive-Or (Xor)

<table>
<thead>
<tr>
<th>^</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0</td>
</tr>
<tr>
<td>1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

Boolean Algebra on Bit Vectors

- Operate on Bit Vectors
- Operations applied bitwise

| 01101001 & 01010101 | 01111101 ^ 01010101 | 00111100 ~ 01010101 |
| 01000001 | 01111110 | 10101010 |

Representing & Manipulating Sets

- Representation
  - Width w bit vector represents subsets of \{0, ..., w−1\}
  - \(a_j = 1\) if \(j \in A\)
    - \(01101001\) \{0, 3, 5, 6\}
    - \(01010101\) \{0, 2, 4, 6\}
  - \(01010101\) \{0, 2, 4, 6\}
  - \(01010101\) \{0, 2, 4, 6\}

- Operations
  - \& Intersection \(01000001\) \{0, 6\}
  - | Union \(01111101\) \{0, 2, 3, 4, 5, 6\}
  - ^ Symmetric difference \(00111100\) \{2, 3, 4, 5\}
  - ~ Complement \(10101010\) \{1, 3, 5, 7\}
Bit-Level Operations in C

- Operations &, |, ~, ^ available in C
- Apply to any "integral" data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
- Arguments applied bit-wise

Examples
- \(~0x41 = 0xBE\)
- \(~01000001_2 = 10111110_2\)
- \(~0x00 = 0xFF\)
  - \(00000001_2 \rightarrow 11111110_2\)
- \(0x69 \& 0x55 = 0x41\)
- \(01101001_2 \& 01010101_2 = 01000001_2\)
- \(0x69 | 0x55 = 0x7D\)
- \(01101001_2 | 01010101_2 = 01111101_2\)

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
- View 0 as "False"
- Anything nonzero as "True"
- Always return 0 or 1
- Early termination

Examples (char data type)
- \(!0x41 = 0x00\)
- \(!0x00 = 0x01\)
- \(!0x41 = 0x01\)
- \(0x69 \& 0x55 = 0x01\)
- \(0x69 || 0x55 = 0x01\)
- \(p \& \& *p\) (avoids null pointer access)

Shift Operations

- Left Shift: \(x << y\)
  - Shift bit-vector \(x\) left \(y\) positions
  - Throw away extra bits on left
  - Fill with 0’s on right

- Right Shift: \(x >> y\)
  - Shift bit-vector \(x\) right \(y\) positions
  - Throw away extra bits on right
  - Logical shift
    - Fill with 0’s on left
  - Arithmetic shift
    - Replicate most significant bit on right

Bits, Bytes, and Integers

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

Unsigned

$$B_2(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

$$B_2(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Encoding Example (Cont.)

$$x = 15213: \begin{array}{c|c|c} \text{Decimal} & \text{Hex} & \text{Binary} \\ \hline x & 15213 & 00111011 \ 01101101 \\ y & -15213 & C4 \ 93 \\ \end{array}$$

For 2's complement, most significant bit indicates sign
- 0 for nonnegative, 1 for negative
- What is the bit representation of -1?

Numeric Ranges

Unsigned Values
- $$U_{Min} = 0$$
- $$U_{Max} = 2^w - 1$$

Two's Complement Values
- $$T_{Min} = -2^{w-1}$$
- $$T_{Max} = 2^{w-1} - 1$$

Values for $$W = 16$$

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>11111111 \ 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>1111 \ 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>1111 \ 11111111</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
</tr>
<tr>
<td>0</td>
<td>0</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>

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

\[ \Rightarrow \text{Can Invert Mappings} \]

- \( U2B(x) = B2U^{-1}(x) \)
  - Bit pattern for unsigned integer
- \( T2B(x) = B2T^{-1}(x) \)
  - Bit pattern for two’s comp integer

### Mapping Between Signed & Unsigned

- Mappings between unsigned and two’s complement numbers:
  - keep bit representations and reinterpret

### Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating

### 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>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>

\[ \Rightarrow \text{Can Invert Mappings} \]

- \( U2B(x) = B2U^{-1}(x) \)
  - Bit pattern for unsigned integer
- \( T2B(x) = B2T^{-1}(x) \)
  - Bit pattern for two’s comp integer

\[ +/- 16 \]
### Relation between Signed & Unsigned

<table>
<thead>
<tr>
<th>T2U</th>
<th>T2B</th>
<th>B2U</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Maintain Same Bit Pattern

\[ \text{ux} = \begin{cases} 
  x & x \geq 0 \\
  x + 2^w & x < 0 
\end{cases} \]

Large negative weight becomes Large positive weight

### Conversion Visualized

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

### Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have "U" as suffix
  - Examples: `0U`, `4294967259U`

- **Casting**
  - Explicit casting between 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;`

### Casting Surprises

- **Expression Evaluation**
  - If there is a mix of unsigned and signed in single expression, `signed values implicitly cast to unsigned`
  - Examples for `W = 32`:
    - `TMIN = -2,147,483,648`, `TMAX = 2,147,483,647`
    - `Constant_1  Constant_2  Relation  Evaluation`
      - `0 0U  ==  unsigned`
      - `-1 0  <  signed`
      - `-1 0U  >  unsigned`
      - `2147483647 -2147483647-1  >  signed`
      - `2147483647U -2147483647-1  <  unsigned`
      - `-1 -2  >  signed`
      - `(unsigned)-1 -2  >  unsigned`
      - `2147483647 2147483648U  <  unsigned`
      - `2147483647 (int) 2147483648U  >  signed`
Security Vulnerability

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}

Real example: similar to code found in FreeBSD's implementation of getpeername

Malicious Usage

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
}

—MSIZE, when interpreted as unsigned int by memcpy, becomes a very large integer

Casting Signed ↔ Unsigned: Basic Rules

- 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!!

Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
**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:
    \[ X' = x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_0 \]

- **Example**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ) = 15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>( ix ) = (int) ( x )</td>
<td>00 00 3B 6D</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>( y ) = (-15213 )</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>( iy ) = (int) ( y )</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

**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

**Disclaimer**

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.