Under the Hood: Data Representation

Computer Science 104
Lecture 2

Some slides based on those from Randy Bryant and Dave O’Hallaron

Administrivia

Generic
- My office hours: Mon 1:30-2:30, Thurs 11:0-noon
- TA: Tues 4:30-5:30, Wed 3:00-4:00
- UTAs: TBD
- Q&A we will use piazza not blackboard
  - See course web page for link to register, etc.

Homework
- Homework #1 Due Sept 12
  - Datalab
- Next Homework C programming w/ pointers, will be up soon.

Reading
- Chapter 2
Last Time: Course Overview

- Course Theme:

**Abstraction Is Good But Don’t Forget Reality**

- 5 Great Realities
  - Ints are not Integers, Floats are not Reals
  - You’ve Got to Know Assembly
  - Memory Matters
  - There’s more to performance than asymptotic complexity
  - Computers do more than execute programs
- Administrative / Logistics details

---

Today: Bits, Bytes, and Integers

- Representing information
- Bit Manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication
Binary Representations

Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0

And
- \( A \& B = 1 \) when both \( A=1 \) and \( B=1 \)

\[
\begin{array}{c|cc}
\& & 0 & 1 \\
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

Or
- \( A \mid B = 1 \) when either \( A=1 \) or \( B=1 \)

\[
\begin{array}{c|cc}
\mid & 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

Not
- \( \sim A = 1 \) when \( A=0 \)

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

Exclusive-Or (Xor)
- \( A \wedge B = 1 \) when either \( A=1 \) or \( B=1 \), but not both

\[
\begin{array}{c|c}
\wedge & 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]
Application of Boolean Algebra

- Applied to Digital Systems by Claude Shannon
  - 1937 MIT Master’s Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0

\[
A \lor \neg B \land \neg A \lor B = A \lor B
\]

General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

\[
\begin{array}{cccc}
01101001 & 01101001 & 01101001 \\
\& & 01010101 & | & 01010101 & ^ & 01010101 & \sim & 01010101 \\
01000001 & 01111101 & 00111100 & 10101010
\end{array}
\]

- All of the Properties of Boolean Algebra Apply
Representing & Manipulating Sets

■ Representation
  ▪ Width w bit vector represents subsets of \( \{0, ..., w-1\} \)
  ▪ \( a_j = 1 \) if \( j \in A \)
    - 01101001 \( \{0, 3, 5, 6\} \)
    - 76543210
    - 01010101 \( \{0, 2, 4, 6\} \)
    - 76543210

■ 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 (Char data type)
  ▪ ~0x41 \( \mapsto \) 0xBE
    - ~01000001_2 \( \mapsto \) 10111110_2
  ▪ ~0x00 \( \mapsto \) 0xFF
    - ~00000000_2 \( \mapsto \) 11111111_2
  ▪ 0x69 & 0x55 \( \mapsto \) 0x41
    - 01101001_2 & 01010101_2 \( \mapsto \) 01000001_2
  ▪ 0x69 | 0x55 \( \mapsto \) 0x7D
    - 01101001_2 | 01010101_2 \( \mapsto \) 01111110_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

- Undefined Behavior
  - Shift amount < 0 or ≥ word size
Data Representation

- Compute two hundred twenty nine minus one hundred sixty seven divided by twelve
- Compute XIX - VII + IV
- We reason about numbers many different ways
- Computers store variables (data)
  - Typically Numbers and Characters or composition of these
- The key is to use a representation that is “efficient”

More Number Systems

- Humans use decimal (base 10)
  - digits 0-9 are composed to make larger numbers
    - $11 = 1*10^1 + 1*10^0$
  - weighted positional notation
- Addition and Subtraction are straightforward
  - carry and borrow (called regrouping)
- Multiplication and Division less so
  - can use logarithms and then do adds and subtracs
Changing Base (Radix)

- Given 4 positions, what is the largest number you can represent?

Number Systems for Computers

- Today’s computers are built from transistors
- Transistor is either off or on
- Need to represent numbers using only off and on
  - two symbols
- off and on can represent the digits 0 and 1
  - BIT is Binary Digit
  - A bit can have a value of 0 or 1
- Binary representation
  - weighted positional notation using base 2
    
    \[
    11_{10} = 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 = 1011_2
    \]

\[
11_{10} = 8 + 2 + 1
\]

What is largest number, given 4 bits?
Binary, Octal and Hexadecimal numbers

- Computers can input and output decimal numbers but they convert them to internal binary representation.
- Binary is good for computers, hard for us to read
  - Use numbers easily computed from binary
- Binary numbers use only two different digits: {0,1}
  - Example: $1200_{10} = 0000010010110000_2$
- Octal numbers use 8 digits: {0 - 7}
  - Example: $1200_{10} = 04260_8$
- Hexadecimal numbers use 16 digits: {0-9, A-F}
  - Example: $1200_{10} = 04B0_{16} = 0x04B0$
  - does not distinguish between upper and lower case

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000 to 11111111
  - Decimal: 0₁₀ to 255₁₀
  - Hexadecimal 00₁₆ to FF₁₆
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write FA1D37B₁₆ in C as
      - 0xFA1D37B
      - 0xfa1d37b
    - To convert to and from hex: group binary digits in groups of four and convert according to table

Example:

```
1100 0010 0110 0111 0100 1111 1101 0101₂
C 2 6 7 4 F D 5₁₆
```
Issues for Binary Representation

- Complexity of arithmetic operations
- Negative numbers
- Maximum representable number
- Choose representation that’s easy for machine not easy for humans

Binary Integers

**Unsigned Integers**
- \( i = 100101_2 \); \( i = 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 \)
- 4 bits => max number is 15

**Sign Magnitude**
- Add a sign bit
  - Example: \( 010110_2 = 22_{10} \); \( 110110_2 = -22_{10} \)
- Advantages:
  - Simple extension of unsigned numbers.
  - Same number of positive and negative numbers.
- Disadvantages:
  - Two representations for 0: 0=000000; -0=100000.
  - Algorithm (circuit) for addition depends on the arguments’ signs.
1’s Complement Representation

- Key is to use largest positive binary numbers to represent negative numbers.
- Find $-x$ represent with $i$:
  - $i = 2^n - 1 - x$
- Simply invert each bit (0->1, 1->0)
- Disadvantage: Two zeros

6-bit examples:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>1's Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000</td>
</tr>
<tr>
<td>1</td>
<td>000001</td>
</tr>
<tr>
<td>-1</td>
<td>111111</td>
</tr>
</tbody>
</table>


2’s Complement Representation

- Still use large positives to represent negatives

$\quad \quad i = 2^n - x$

- This is 1’s complement + 1

$\quad \quad i = 2^n - 1 - x + 1$

- So, invert bits and add 1

6-bit examples:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>2's Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000</td>
</tr>
<tr>
<td>1</td>
<td>000001</td>
</tr>
<tr>
<td>-1</td>
<td>111111</td>
</tr>
</tbody>
</table>


2’s Complement

- Advantages:
  - Only one representation for 0: $0 = 000000$
  - Addition algorithm independent of sign bits.

- Disadvantage:
  - One more negative number than positive:
    - Example: 6-bit 2’s complement number.
    - $100000_2 = -32_{10}$, but $32_{10}$ could not be represented

2’s Complement Negation

- To negate a number do:
  - Step 1. complement the digits
  - Step 2. add 1

Example

$14_{10} = 001110_2$
$-14_{10} = 110001_2$

$$+ 1$$
$$110010_2$$
Complement & Increment Examples

\( x = 15213 \)

<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 00111011 01101101</td>
</tr>
<tr>
<td>( \sim x )</td>
<td>-15214</td>
<td>C4 92 11000100 10010010</td>
</tr>
<tr>
<td>( \sim x + 1 )</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
</tbody>
</table>

\( y = -15213 \)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
</tbody>
</table>

\( x = 0 \)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00000000 00000000</td>
</tr>
<tr>
<td>( \sim 0 )</td>
<td>-1</td>
<td>FF FF 11111111 11111111</td>
</tr>
<tr>
<td>( \sim 0 + 1 )</td>
<td>0</td>
<td>00 00 00000000 00000000</td>
</tr>
</tbody>
</table>

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>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
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:
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$

In C, automatic sign extension for converting from smaller to larger integer data type.

Example

```c
short int x = 15213;
int ix = (int) x;
short int y = -15213;
int iy = (int) y;
```

<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>

- Converting from smaller to larger integer data type
- C automatically performs sign extension
Summary: 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

Mapping Between Signed & Unsigned

- Mappings between unsigned and two’s complement numbers: *keep bit representations and reinterpret*
Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix
    0U, 4294967259U

- **Casting**
  - Explicit casting between signed & unsigned same as U2T and T2U
    ```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

Expression Evaluation
- If there is a mix of unsigned and signed in single expression,
  *signed values implicitly cast to unsigned*
- Including comparison operations \(<\), \(\ge\), \(<=\), \(>=\)
- Examples for \(W = 32\): \(TMIN = -2,147,483,648\), \(TMAX = 2,147,483,647\)

<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>

Code Security Example

```c
/* 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;
}
```

- Similar to code found in FreeBSD’s (Unix) implementation of getpeername
- There are legions of smart people trying to find vulnerabilities in programs
Typical Usage

```c
/* 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;
}
```

```c
#define MSIZE 528
void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

What could go wrong?

Malicious Usage

```c
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

```c
/* 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;
}
```

```c
#define MSIZE 528
void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
}
```
Summary
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`!

Unsigned Addition (UAdd)

Operands: $w$ bits

```
  \[ u \quad \ldots \quad \ldots \quad \ldots \quad v \]
```

True Sum: $w+1$ bits

```
  \[ u + v \quad \ldots \quad \ldots \quad \ldots \quad \ldots \]
```

Discard Carry: $w$ bits

```
  UAdd_w(u, v) \quad \ldots \quad \ldots \quad \ldots \quad \ldots
```

- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic
  - $s = \text{UAdd}_w(u, v) = u + v \mod 2^w$

Potential Problems?
- 4-bit values: $14 + 2 = ??
Two’s Complement Addition (TAdd)

Operands: \( w \) bits

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

True Sum: \( w+1 \) bits

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

Discard Carry: \( w \) bits

- \text{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
Next time

- Multiplication
- Floating Point
- Memory
  - Pointers
  - Arrays
  - Strings

- Machine Level Representation

Reading
- Chapter 2