Under the Hood: Data Representation

Computer Science 104
Lecture 2

Administrivia

Generic
■ My office hours Mon & Thurs: 2:30-3:30
■ UTAs: Daphne Erez and Alex Kritchevsky

Homework
■ Homework #1 Due Jan 24
  ▪ One C program, linked list manipulation
■ Next Homework Due Jan 31, 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
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication
Representations (Group task)

- Form partners
- Using only the three symbols @ # $ specify a representation for the following:
  - All integers from 0 to 10
  - Commands to 1) walk, 2) turn, 3) sit, 4) raise right arm, 5) raise left arm

- Using only your representation write down series of commands & integers (if appropriate, e.g., raise left arm-3, turn-2)
  - Must have at least 5 commands

What You Know Today

**C**

```c
... int result;
double score;

double curve(double score) {
    return(score * 0.22124);
}

int main()
{
    int x;
    ...
    result = x + result;
    printf("Score is %d\n", curve(80));
    ...
}
```

**JAVA**

```java
... System.out.println("Please Enter In Your First Name: ");
String firstName = bufRead.readLine();
System.out.println("Please Enter In The Year You Were Born: ");
String bornYear = bufRead.readLine();
System.out.println("Please Enter In The Current Year: ");
String thisYear = bufRead.readLine();
int bYear = Integer.parseInt(bornYear);
int tYear = Integer.parseInt(thisYear);
int age = tYear - bYear;
System.out.println("Hello " + firstName + ", You are " + age + " years old");
```
High Level to Assembly

High Level Language (C, C++, Fortran, Java, etc.)
- Statements
- Variables
- Operators
- Methods, functions, procedures

Assembly Language
- Instructions
- Registers
- Memory

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
    \[ 11 = 1 \times 10^1 + 1 \times 10^0 \]
  ▪ 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 subtracts

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
    \[ 11_{10} = 1\times2^3 + 1\times2^1 + 1\times2^0 = 1011_2 \]
    \[ 11_{10} = 8 + 2 + 1 \]

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 $00000000_2$ to $11111111_2$
  - 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
    - To convert to and from hex: group binary digits in groups of four and convert according to table

Example:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

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:
- $010110_2 = 22_{10}; 101001_2 = -22_{10}$
- $0_{10} = 000000_2; 0 = 111111_2$
- $1_{10} = 000001_2; -1_{10} = 111110_2$
- $1010 = 10 - 101 = -5$
- $1011 = 10 - 101 = -4$
- $1100 = 10 - 1001 = -3$
- $1101 = 10 - 1011 = -2$
- $1110 = 10 - 1111 = -1$
- $1111 = 10 - 1111 = -0$
2’s Complement Representation

- Still use large positives to represent negatives
  \[ i = 2^n - x \]
- This is 1’s complement + 1
  \[ i = 2^n - 1 - x + 1 \]
- So, invert bits and add 1

6-bit examples:
- \(010110_2 \) = \(22_{10}\)
- \(100101_2 \) = \(-22_{10}\)
- \(0_{10} = 000000_2\)
- \(1_{10} = 000001_2, -1_{10} = 111111_2\)

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

\[
\begin{align*}
14_{10} &= 001110_2 \\
-14_{10} &= 110001_2 \\
+1 &
\end{align*}
\]

Complement & Increment Examples

\[
\begin{array}{cccc}
\text{Decimal} & \text{Hex} & \text{Binary} \\
15213 & 3B 6D & 00111011 \text{ } 01101101 \\
-15214 & C4 92 & 11000100 \text{ } 10010010 \\
-15213 & C4 93 & 11000100 \text{ } 10010011 \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{Decimal} & \text{Hex} & \text{Binary} \\
0 & 0 & 00000000 \text{ } 00000000 \\
-0 & -1 & FF \text{ } FF \text{ } 1111111 \text{ } 1111111 \\
-0+1 & 0 & 00 \text{ } 00 \text{ } 00000000 \text{ } 00000000 \\
\end{array}
\]
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:
    - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$
Sign Extension Example

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

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

- Two’s Complement
  - T2U
  - T2B
  - B2U

- Unsigned
  - U2T
  - U2B
  - B2T

Maintain Same Bit Pattern

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

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>-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>
### 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
    - `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 `<`, `>`, `==`, `<=, `>=`
- 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>2147483647U</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

\[
\begin{array}{c}
\begin{array}{c}
\mu \\
+ \nu \\
\end{array}
\end{array}
\]

True Sum: $w+1$ bits

\[
\begin{array}{c}
\begin{array}{c}
\mu + \nu \\
\text{UAdd}_w(u, v) \\
\end{array}
\end{array}
\]

Discard Carry: $w$ bits

- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic
  \[
s = \text{UAdd}_w(u, v) = u + \nu \mod 2^w
\]
- Potential Problems?
  - 4-bit values: $14 + 2 = ??$
Visualizing Unsigned Addition

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

Two's Complement Addition (TAdd)

<table>
<thead>
<tr>
<th>Operands: w bits</th>
<th>$u$</th>
<th>$v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Sum: w+1 bits</td>
<td>$u + v$</td>
<td></td>
</tr>
<tr>
<td>Discard Carry: w bits</td>
<td>TAdd$_w(u, v)$</td>
<td></td>
</tr>
</tbody>
</table>

- 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

\[
\begin{align*}
\text{True Sum} & \quad \text{TAdd Result} \\
000...0 & \quad 011...1 \\
011...1 & \quad 000...0 \\
100...0 & \quad 011...1 \\
111...1 & \quad \text{-2}^w \\
\text{-2}^w & \quad \text{-2}^w \\
\text{PosOver} & \quad \text{NegOver}
\end{align*}
\]

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
Addition

- Example: A = 0x0ABC; B = 0x0FEB.

- Compute: A + B and A - B in 16-bit 2's complement arithmetic.

- Give answer in HEX

Answer

- A + B = 0x1AA7
- A - B = 0xFAD1


**Multiplication**

- Computing Exact Product of $w$-bit numbers $x, y$
  - Either signed or unsigned
  - How many bits are required, in general?

---

**Unsigned Multiplication in C**

<table>
<thead>
<tr>
<th>Operands: $w$ bits</th>
<th>$u$</th>
<th>$*$</th>
<th>$v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Product: $2w$ bits</td>
<td>$u \cdot v$</td>
<td>$*$</td>
<td>$v$</td>
</tr>
<tr>
<td>Discard $w$ bits: $w$ bits</td>
<td>$\text{UMult}_w(u, v)$</td>
<td>$*$</td>
<td>$v$</td>
</tr>
</tbody>
</table>

- 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

\[
\begin{array}{c|c}
\mu & \ast \\
\hline
\nu & \ast \\
\end{array}
\]

True Product: \( 2^w \) bits

\[
\begin{array}{c|c}
\mu \cdot \nu & \ast \\
\hline
\text{TMult}_w(\mu, \nu) & \ast \\
\end{array}
\]

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 \( u \ast 2^k \)
  ▪ Both signed and unsigned

Operands: \( w \) bits

\[
\begin{array}{c|c|c}
\mu & \ast & k \\
\hline
\nu & \ast & \nu \ll k \\
\end{array}
\]

True Product: \( w+k \) bits

\[
\begin{array}{c|c|c}
\mu \cdot 2^k & \ast & \nu \ll k \\
\hline
\text{UMult}_w(\mu, 2^k) & \ast & \text{TMult}_w(\mu, 2^k) \\
\end{array}
\]

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

■ Examples
  ▪ \( u \ll 3 \) \( \equiv \) \( u \ast 8 \)
  ▪ \( u \ll 5 - u \ll 3 \equiv u \ast 24 \)
  ▪ Many machines shift and add faster than multiply
    ▪ Compiler generates this code automatically
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 (Next Lecture)*
  - Logical right shift, no sign extension

Next time

- Bitwise operations
- Memory
  - Pointers
  - Arrays
  - Strings

- Machine Level Representation

Reading
- Chapter 2