Under the Hood: Data Representations, Memory and Bit Operations

Computer Science 104
Lecture 3

Administrivia
- Homework #1 Due Tonight
- Homework #2 Due Next Wednesday night

Outline
- Review/finish data representations (Floating Point)
- Arrays
- Pointers
- Pointer Arithmetic
- Bitwise operations (AND, OR)

Reading
  Chapter 2
Integers

- Unsigned vs. signed (2’s complement)
- Extension (e.g. short to int)
- Overflow
  - Not enough bits to represent number
  - Unsigned: large positives
  - 2’s complement: large positives and large negatives

Review: 2’s Complement Negation and Addition

- To negate a number (given x what is \(-x\)):
  - Step 1. complement the digits
  - Step 2. add 1

Example

\[
\begin{align*}
14_{10} &= 001110_2 \\
-14_{10} &= 110001_2 \\
+1_{110001_2} &= 110010_2
\end{align*}
\]

- To add signed numbers use regular addition but disregard carry out

Example

\[
\begin{align*}
18_{10} - 14_{10} &= 18_{10} + (-14_{10}) = 4_{10} \\
010010_2 + 110010_2 &= 000100_2
\end{align*}
\]
Multiplication

- Computing Exact Product of 32-bit numbers $x, y$
  - Either signed or unsigned
  - How many bits are required?

Power-of-2 Multiply with Shift

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

<table>
<thead>
<tr>
<th>$u$</th>
<th>$2^k$</th>
<th>$0$</th>
<th>$00$</th>
<th>$000$</th>
<th>$000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$2^k$</td>
<td>$0$</td>
<td>$00$</td>
<td>$000$</td>
<td>$000$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>True Product: $w+k$ bits</th>
<th>$u \times 2^k$</th>
<th>$0$</th>
<th>$00$</th>
<th>$000$</th>
<th>$000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discard $k$ bits: $w$ bits</td>
<td>$\text{UMult}(u, 2^k)$</td>
<td>$0$</td>
<td>$00$</td>
<td>$000$</td>
<td>$000$</td>
</tr>
<tr>
<td></td>
<td>$\text{TMult}(u, 2^k)$</td>
<td>$0$</td>
<td>$00$</td>
<td>$000$</td>
<td>$000$</td>
</tr>
</tbody>
</table>

- Examples
  - $u \ll 3$ == $u \times 8$
  - $u \ll 5 - u \ll 3$ == $u \times 24$
  - 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
    - unsigned i;
    - for (i = cnt-2; i >= 0; i--)
    - a[i] += a[i+1];
  - Can be very subtle
    - #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

What About Non-integer Numbers?

- There are infinitely many real numbers between two integers
- Many important numbers are real
  - speed of light $\approx 3 \times 10^8$
  - 3.145...
- Fixed number of bits limits range of integers
  - Can’t represent some important numbers
- Humans use Scientific Notation
  - 1.3x104
Floating Point Representation

Numbers are represented by:

\[ X = (-1)^s \times 1 \times F \times 2^{E-127} \]

- Only store S, E & F
  
  \( S := 1 \)-bit field; Sign bit

  \( E := 8 \)-bit field; Exponent: Biased integer, \( 0 \leq E \leq 255 \).

  \( F := 23 \)-bit field; Fraction: Normalized fraction with hidden 1 (don’t actually store it)

- Single precision floating point number uses 32-bits for representation (type float)

<table>
<thead>
<tr>
<th>31 30 22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>23-bit</td>
</tr>
<tr>
<td>( s )</td>
<td>( \text{exp} )</td>
</tr>
</tbody>
</table>

Example:

\[ X = -0.75_{10} \text{ in single precision } \]

\[ (-1/2 + 1/4) \]

\[ -0.75_{10} = -0.11_2 = (-1) \times 1.1_2 \times 2^{-1} = (-1) \times 1.1_2 \times 2^{126-127} \]

\( S = 1 \); \( \text{Exp} = 126_{10} = 01111110_2 \);

\( F = 100000000000000000000000_2 \)

\[ X = \begin{array}{c}
01111110100000000000000000000000000000000
\end{array} \]

\( s \) \( \text{E} \) \( \text{F} \)
Floating Point Representation

- Example:
- What floating-point number is: 0xC1580000?

\[
\begin{array}{c}
\begin{array}{c}
\text{s=1}
\end{array} \\
\begin{array}{c}
\text{E=128+2-127 = 3}
\end{array} \\
\begin{array}{c}
\text{F=1011}
\end{array} \\
\begin{array}{c}
-1.1011 \times 2^3 = -1101.1 = -13.5
\end{array}
\end{array}
\]

Answer

What floating-point number is 0xC1580000?

1100 0001 0101 1000 0000 0000 0000 0000

\[
X = \begin{bmatrix}
1 \\
1000 \\
0010 \\
101 \\
1000 \\
0000 \\
0000 \\
0000 \\
0000 \\
0000
\end{bmatrix}
\]

- \( s = 1 \)
- \( E = 128 + 2 - 127 = 3 \)
- \( F = 1011 \)
- \(-1.1011 \times 2^3 = -1101.1 = -13.5 \)
Floating Point Representation

- Double Precision Floating point:

  64-bit representation: 1-bit sign, 11-bit (biased) exponent; 52-bit fraction (with hidden 1).

  \[ X = (-1)^s \times 1.F \times 2^{E-1023} \]

**Double precision floating point number (type double)**

<table>
<thead>
<tr>
<th>S</th>
<th>Exp</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11-bit</td>
<td>20 - bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 - bit</td>
</tr>
</tbody>
</table>

What about characters/strings?

- Many important things stored as strings...
  - Your name
- What is a string?
- How should we represent strings?
### ASCII Character Representation

<table>
<thead>
<tr>
<th>Oct. Chr.</th>
<th>000</th>
<th>001</th>
<th>002</th>
<th>003</th>
<th>004</th>
<th>005</th>
<th>006</th>
<th>007</th>
</tr>
</thead>
<tbody>
<tr>
<td>nul</td>
<td>000</td>
<td>soh</td>
<td>stx</td>
<td>etx</td>
<td>eot</td>
<td>enq</td>
<td>ack</td>
<td>bel</td>
</tr>
<tr>
<td>bs</td>
<td>010</td>
<td>hlt</td>
<td>nl</td>
<td>vt</td>
<td>sp</td>
<td>cr</td>
<td>so</td>
<td>si</td>
</tr>
<tr>
<td>dc1</td>
<td>020</td>
<td>dcl</td>
<td>dc2</td>
<td>dc3</td>
<td>dc4</td>
<td>nak</td>
<td>syh</td>
<td>etb</td>
</tr>
<tr>
<td>em</td>
<td>030</td>
<td>em</td>
<td>sub</td>
<td>esc</td>
<td>fs</td>
<td>gs</td>
<td>us</td>
<td>can</td>
</tr>
<tr>
<td>sp</td>
<td>040</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>'</td>
</tr>
<tr>
<td>(</td>
<td>050</td>
<td>)</td>
<td>+</td>
<td>,</td>
<td>-</td>
<td>.</td>
<td>/</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>060</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>070</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>080</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>090</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>100</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>40</td>
<td>110</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>48</td>
<td>120</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
</tr>
<tr>
<td>56</td>
<td>130</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
</tr>
<tr>
<td>64</td>
<td>140</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
</tr>
<tr>
<td>72</td>
<td>150</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
</tr>
<tr>
<td>80</td>
<td>160</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
</tr>
<tr>
<td>88</td>
<td>170</td>
<td>89</td>
<td>90</td>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
</tr>
</tbody>
</table>

- Each character is represented by a 7-bit ASCII code.
- It is packed into 8-bits

### Unicode

- Many types
- UTF-8: variable length encoding backward compatible with ASCII
  - Linux
- UTF-16: variable length
  - Windows, Java
- UTF-32: fixed length
Summary of Data Representations

- Computers operate on binary numbers (0s and 1s)
- Conversion to/from binary, oct, hex
- Signed binary numbers
  - 2’s complement
  - arithmetic, negation
- Floating point representation
  - hidden 1
  - biased exponent
  - single precision, double precision
- ASCII code for characters

Computer Memory

- What is Computer Memory?
- What does it “look like” to the program?
- How do we find things in computer memory?
A Program’s View of Memory

- What is Memory? a bunch of bits
- Looks like a large linear array
- Find things by indexing into array
  - unsigned integer
- Most computers support byte (8-bit) addressing
  - Each byte has a unique address (location).
  - Byte of data at address 0x100 and 0x101
- Machine Has “Word Size”
  - Nominal size of integer-valued data
    - Including addresses
  - 32-bit vs. 64-bit addresses
  - 32-bit word of data at address 0x100 and 0x104

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)
Byte Ordering (Buzzword: Endian)

- How should bytes of a multi-byte word be ordered in memory?
- Big Endian
  - Least significant byte has highest address
- Little Endian
  - Least significant byte has lowest address (Intel x86)
- Example
  - An int has a 4-byte representation 0x01234567

Big Endian:
0x100 0x101 0x102 0x103
 01  23  45  67

Little Endian:
0x100 0x101 0x102 0x103
 67  45  23  01

Buzz Word Definition: Alignment

- **Alignment**: require that objects fall on address that is multiple of their size.
- 32-bit integer
  - Aligned if address \( \% \ 4 = 0 \)
- 64-bit integer?
  - Aligned if ?
Memory Partitions

- Text for instructions
  - `add res, src1, src2`
  - `mem[res] = mem[src1] + mem[src2]`
- Data
  - static (constants, globals)
  - dynamic (heap, `new` allocated)
  - grows up
- Stack
  - local variables
  - grows down
- Variables are names for memory locations
  - `int x;`

---

A Simple Program’s Memory Layout

```c
... int result; // global var main() {
    int x;
    ...
    result = x + result;
    ...
}
mem[0x208] = mem[0x400] + mem[0x208]
```
Reference (handle) vs. Pointer

Java

- “The value of a reference type variable, in contrast to that of a primitive type, is a reference to (an address of) the value or set of values represented by the variable”
  http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html

- Cannot manipulate the value of the reference

C

- A pointer is a memory location that contains the address of another memory location
- Can manipulate the value of pointer (double edge sword)

Pointers

- “address of” operator &
  - don’t confuse with bitwise AND operator (later today)

Given

```c
int x; int *p;
p = &x;
```

Then

```c
*p = 2; and x = 2; produce the same result
```

- What happens for `p = 2;`:

On 32-bit machine, `p` is 32-bits

```
x 0x26cf0
... *
p 0x26d00
```

- `p` is 32-bits, `x` is 32-bits, and the address of `x` is stored in `p`.

- Manipulating `p` does not affect `x` directly, but it allows accessing the memory location that contains `x`'s address.

- On a 32-bit machine, `p` is represented by 32 bits, while `x` is also 32 bits, indicating that the address of `x` is stored in `p`. This allows dynamic memory allocation and manipulation, which is a powerful feature of pointers.
Vector (ArrayList) Class vs. Arrays

- Vector Class
  - insulates programmers
  - array bounds checking
  - automagically growing/shrinking when more items are added/deleted

- How are Vectors implemented?
  - real understanding comes when more levels of abstraction are understood (the ridiculous...)

- Programming close to HW
  - (e.g., operating system, device drivers, etc.)

- Arrays can be more efficient
  - but be leery of claims that C-style arrays required for efficiency

- Can talk about memory easier in terms of arrays
  - pointer to a vector?

Arrays

- In C
  - malloc(nbytes);
  - free(ptr);

- No automatic growing or shrinking
Address Calculation

- x is a pointer, what is x+33?
- A pointer, but where?
  - what does calculation depend on?
- Result of adding an int to a pointer depends on size of object pointed to
- Result of subtracting two pointers is an int

\[(d + 3) - d = \text{blank}\]

```
int * a = new int[100];
a = (int*) malloc(100*sizeof(int));  
\begin{array}{|c|c|c|c|}
\hline
0 & 1 & 32 & 33 \\
\hline
98 & 99 & & \hline
\end{array}
```

- a[33] is the same as *(a+33)
- if a is 0x00a0, then a+1 is 0x00a1
- a+2 is 0x00a8
  - (decimal 160, 164, 168)

```
double * d = new double[200];
d = (double*) malloc(100*sizeof(double));
\begin{array}{|c|c|c|c|}
\hline
0 & 1 & 33 & 99 \\
\hline
32 & 98 & & \hline
\end{array}
```

- *(d+33) is the same as d[33]
- if d is 0x00b0, then d+1 is 0x00b1
- d+2 is 0x00c0
  - (decimal 176, 184, 192)

More Pointer Arithmetic

- address one past the end of an array is ok for pointer comparison only
- what’s at *(begin+44)?
- what does begin++ mean?
- how are pointers compared using < and using ==?
- what is value of end - begin?

```
char * a = new char[44];
char * begin = a;
char * end = a + 44;

while (begin < end)
{
    *begin = 'z';
    begin++;
}
```
More Pointers & Arrays

- `int * a = new int[100];`
- a is a pointer
- *a is an int
- a[0] is an int (same as *a)
- a[1] is an int
- a+1 is a pointer
- a+32 is a pointer
- *(a+1) is an int (same as a[1])
- *(a+99) is an int
- *(a+100) is trouble

Array Example

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int *a;
    int k;
    a = (int *) malloc(100*sizeof(int));
    int *p = a; // do this after malloc

    for (k = 0; k < 100; k++)
    {
        *p = k;
        p++;
    }

    printf("Entry 3 is %d\n",a[3]);
    return(0);
}
```
C Array of Structures -> Linked List

```c
#include <stdio.h>
#include <stdlib.h>

struct node {
  int me;
  struct node *next;
};

int main() {
  struct node *ar;
  struct node *p;
  int k;
  ar = (struct node *) malloc (10*sizeof(struct node));
  p = ar;
  for (k = 0; k < 9; k++) {
    p->me = k;
    p->next = &ar[k+1];
    p++;
  }
  p->me = 9;
  p->next = NULL;
  p = &ar[0];
  while (p != NULL) {
    printf("%d 0x%x 0x%lx\n", p->me, (unsigned long) p, (unsigned long) p->next);
    p = p->next;
  }
  return(0);
}
```

Given `ar = 0x10000`, what does memory layout look like?

Memory Layout

<table>
<thead>
<tr>
<th>Output</th>
<th>Memory Address</th>
<th>Memory Contents</th>
<th>Source Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Me</td>
<td>0x26ca8</td>
<td>0</td>
<td>me</td>
</tr>
<tr>
<td>p</td>
<td>0x26cb0</td>
<td>0x26cb0</td>
<td>next</td>
</tr>
<tr>
<td>p-&gt;next</td>
<td>0x26cb8</td>
<td>0</td>
<td>ar[0]</td>
</tr>
<tr>
<td></td>
<td>0x26cc0</td>
<td>0x26cc0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x26cc8</td>
<td>0x26cc8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x26cd0</td>
<td>0x26cd0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x26cd8</td>
<td>0x26cd8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x26ce0</td>
<td>0x26ce0</td>
<td>me</td>
</tr>
<tr>
<td></td>
<td>0x26ce8</td>
<td>0x26ce8</td>
<td>next</td>
</tr>
<tr>
<td></td>
<td>0x26cf0</td>
<td>0</td>
<td>ar[9]</td>
</tr>
<tr>
<td></td>
<td>0x26cf8</td>
<td>0x26cf8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x26cfo</td>
<td>0x0</td>
<td></td>
</tr>
</tbody>
</table>
Memory Manager (Heap Manager)

- `malloc / new (C/C++)`
- Library routines that handle memory management for data segment (allocation / deallocation)
- Java has garbage collection (reclaim memory of unreferenced objects)
- C/C++ must use `free/delete, else memory leak`

Strings as Arrays

- A string is an array of characters with '\0' at the end
- Each element is one byte, ASCII code
- '\0' is null (ASCII code 0)
Strlen()

- `strlen()` returns the # of characters in a string
  - same as # elements in char array?

```c
int strlen(char * s)
// pre: '\0' terminated
// post: returns # chars
{
    int count=0;
    while (*s++)
        count++;
    return count;
}
```

Outline

- Memory
  - Important concept: bits in memory can represent anything
  - Data (char, int, float, int*, char*, etc.)
  - Instructions (the commands of the machine)
- Bit Manipulations
Bit Manipulations

Problem
- 32-bit word contains many values
  - e.g., input device, sensors, etc.
  - current x,y position of mouse and which button (left, mid, right)
- Assume x, y position is 0-255
- How many bits for position?
- How many for button?

Goal
- Extract position and button from 32-bit word
- Need operations on individual bits of word

Bitwise AND / OR

- & operator performs bitwise AND
- | operator performs bitwise OR
- Per bit
  - 0 & 0 = 0
  - 0 & 1 = 0
  - 1 & 0 = 0
  - 1 & 1 = 1
- For multiple bits, apply operation to individual bits in same position

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>011010</td>
<td>011010</td>
</tr>
<tr>
<td>101110</td>
<td>101110</td>
</tr>
<tr>
<td>001010</td>
<td>111110</td>
</tr>
</tbody>
</table>
Mouse Example

- 32-bit word with x, y and button fields
  - bits 0-7 contain x position
  - bits 8-15 contain y position
  - bits 16-17 contain button (0 = left, 1 = middle, 2 = right)
- To extract value need to clear all other bits
- How do I use bitwise operations to do this?

```
button  y   x
0x1a34c = 01 1010 0011 0100 1100
```

Mouse Solution

- AND with a bit mask
  - specific values that clear some bits, but pass others through
- To extract x position use mask 0x000ff

```
xpos = 0x1a34c & 0x000ff
```

```
button  y   x
0x1a34c = 01 1010 0011 0100 1100
0x000ff = 00 0000 0000 1111 1111
0x0004c = 00 0000 0000 0100 1100
```
More of the Mouse Solution

- To extract y position use mask 0x0ff00
  ypos = 0x1a34c & 0x0ff00
- Similarly, button is extracted with mask 0x30000
  button = 0x1a34c & 0x30000
- Not quite done...why?

<table>
<thead>
<tr>
<th>button</th>
<th>y</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1a34c</td>
<td>01</td>
<td>1010 0011 0100 1100</td>
</tr>
<tr>
<td>0x000ff</td>
<td>00</td>
<td>1111 1111 0000 0000</td>
</tr>
<tr>
<td>0x0a300</td>
<td>00</td>
<td>1010 0011 0000 0000</td>
</tr>
</tbody>
</table>

The SHIFT operator

- >> is shift right, << is shift left, operands are int and number of positions to shift
- (1 << 3) is ...000001 -> ...0001000 (it’s 2^3)
- 0xff00 is 0xff << 8, and 0xff is 0xff00 >> 8
- So, true ypos value is
  ypos = (0x1a34c & 0x0ff00) >> 8
  button = (0x1a34c & 0x30000) >> 16
Extracting Parts of Floating Point Number

- See web page for full code
- x is 32-bit word

```c
#define EXP_BITS 8
#define FRACTION_BITS 23
#define SIGN_MASK 0x80000000
#define EXP_MASK 0x7f800000
#define FRACTION_MASK 0x007fffff
struct myfloat {
    int sign;
    unsigned int exp;
    unsigned int fraction;
};
float x;

num->sign = (x & SIGN_MASK) >> (EXP_BITS + FRACTION_BITS);
num->exp = (x & EXP_MASK) >> FRACTION_BITS;
num->fraction = x & FRACTION_MASK;
```

Summary

- Homework #2 Wednesday February 2
- Computer memory is linear array of bytes
- Pointer is memory location that contains address of another memory location
- Bitwise operations
- Code examples are linked to course web page
- We’ll visit these topics again throughout semester

Next Time
- Instruction set architecture (ISA)
  Reading: Chapter 3