Under the Hood: Memory and Bit Operations

CPS 104
Lecture 3

Administrivia

• Homework #1 Due September 12
• Memory
• Bitwise operations

Outline
• Review data representations
• Arrays
• Pointers
• Pointer Arithmetic
• Bitwise operations (AND, OR)

Reading
Chapter 3 (next few lectures)
Review: Binary Integers

• **Unsigned Binary numbers (only positive)**
  - Base 2 numbers, only two digits \{0, 1\}
  - \( i = 100101_2 = 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 \)

• **Sign-Magnitude** := Highest order bit is the sign bit
  - Example: 010110_2 = 22_{10}; 110110_2 = -22_{10}

• **2's Complement**
  - \( i = -a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots a_02^0 \)
  - Example: 010110_2 = 22_{10}; 101010_2 = -22_{10} (6-bit 2's comp.)
  - Examples: 0_{10} = 000000_2; 1_{10} = 000001_2; -1_{10} = 111111_2

Review: Binary, Octal and Hexidecimal 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

• Hexidecimal numbers use 16 digits: \{0-9, A-F\}
  - Example: 1200_{10} = 04B0_{16} = 0x04B0
### Review: 2’s Complement Negation and Addition

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

**Example**

\[
\begin{align*}
14_{10} &= 001110_2 \\
-14_{10} &= 110011_2 \\
+1_{10} &= 110010_2
\end{align*}
\]

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

**Example**

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

### Review: Floating Point Representation

Numbers are represented by:

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

- **S := 1-bit field; Sign bit**
- **E := 8-bit field; Exponent: Biased integer, 0 \leq E \leq 255.**
- **M := 23-bit field; Mantissa: Normalized fraction with hidden 1.**

**Single precision floating point number**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>exp</td>
<td>23-bit</td>
<td>Mantissa</td>
</tr>
</tbody>
</table>
Floating Point Representation

• The mantissa represents a fraction using binary notation:
  \[ M = . s_1, s_2, s_3 \ldots = 1.0 + s_1*2^{-1} + s_2*2^{-2} + s_3*2^{-3} + \ldots \]

• Example: \( X = -0.75_{10} \) in single precision \((-1/2 + 1/4)\)
  
  \[-0.75_{10} = -0.11_2 = (-1) \times 1.1_2 \times 2^{-1} = (-1) \times 1.1_2 \times 2^{126-127} \]
  \[ S = 1; \ Exp = 126_{10} = 0111 1110_2; \]
  \[ M = 100 0000 0000 0000 0000 0000_2 \]

```
X = 31 30 23 22 0
  s  E  M
  0 111 1110 100 0000 0000 0000 0000 0000 0000
```

Floating Point Representation

Example:
What floating-point number is: \( 0xC1580000 \)?
Floating Point Representation

- **Double Precision Floating point:**

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

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

Double precision floating point number

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

32-bit

---

ASCII Character Representation

| Oct.Chr. | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 120 | 121 | 122 | 123 | 130 | 131 | 132 | 133 | 140 | 141 | 142 | 143 | 144 | 145 | 146 | 147 | 150 | 151 | 152 | 153 | 154 | 155 | 156 | 157 | 160 | 161 | 162 | 163 | 170 | 171 | 172 | 173 | 174 | 175 | 176 | 177 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 000 | nul | 001 | soh | 002 | stx | 003 | etx | 004 | eot | 005 | enq | 006 | ack | 007 | bel | 008 | sp  | 009 | !   | 010 | "   | 011 | #   | 012 | $   | 013 | %   | 014 | &   | 015 | '   | 016 | (   | 017 | )   | 018 | *   | 019 | +   | 020 | ,   | 021 | -   | 022 | .   | 023 | /   | 024 | 0   | 025 | 1   | 026 | 2   | 027 | 3   | 028 | 4   | 029 | 5   | 030 | 6   | 031 | 7   | 032 | 8   | 033 | 9   | 034 | :   | 035 | ;   | 036 | <   | 037 | =   | 038 | >   | 039 | ?   | 040 | @   | 041 | A   | 042 | B   | 043 | C   | 044 | D   | 045 | E   | 046 | F   | 047 | G   | 048 | H   | 049 | I   | 050 | J   | 051 | K   | 052 | L   | 053 | M   | 054 | N   | 055 | O   | 056 | P   | 057 | Q   | 058 | R   | 059 | S   | 060 | T   | 061 | U   | 062 | V   | 063 | W   | 064 | X   | 065 | Y   | 066 | Z   | 067 |

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

Bit: 0, 1

Bit String: sequence of bits of a particular length
- 4 bits is a nibble
- 8 bits is a byte
- 16 bits is a half-word
- 32 bits is a word
- 64 bits is a double-word

Character:
- ASCII 7 bit code
- Decimal (BCD code)
  - digits 0-9 encoded as 0000 thru 1001
two decimal digits packed per 8 bit byte

Integers:
- 2's Complement (32-bit representation).

Floating Point:
- Single Precision (32-bit representation).
- Double Precision (64-bit representation).
- Extended Precision (128-bit representation).

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
  — Word of data at address 0x100 and 0x104
• 32-bit v.s. 64-bit addresses
  — we will assume 32-bit for rest of course, unless otherwise stated

Buzz Word Definition: Endianess

Byte Order
• **Big Endian:** byte 0 is 8 most significant bits IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
• **Little Endian:** byte 0 is 8 least significant bits Intel 80x86, DEC Vax, DEC Alpha
Buzz Word Definition: Alignment

- **Alignment**: require that objects fall on address that is multiple of their size.
- 32-bit integer
  - Aligned if address \( \text{addr} \mod 4 = 0 \)
- 64-bit integer?
  - Aligned if \?

<table>
<thead>
<tr>
<th>Byte #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Aligned</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Not Aligned</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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;
main()
{
    int *x;
    ...
    result = x + result;
    ...
}
mem[0x208] = mem[0x400] + mem[0x208]
```

Pointers

- A pointer is a memory location that contains the address of another memory location
- “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
```

On 32-bit machine, p is 32-bits
Vector Class vs. Arrays

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

• How are Vectors implemented?
  - real understanding comes when more levels of abstraction are understood

• 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++ allocate using array form of new
  
  ```
  int *a = new int[100];
  double *b = new double[300];
  ```

• new [] returns a pointer to a block of memory
  - how big? where?

• Size of chunk can be set at runtime

• delete [] a; // storage returned

• In C
  
  ```
  malloc(nbytes);
  free(ptr);
  ```
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
  - \((d + 3) - d = \) _______

int * a = new int[100]

```
0 1 32 33 98 99
```

a[33] is the same as *(a+33)
if a is 0x00a0, then a+1 is 0x00a4, a+2 is 0x00a8
(decimal 160, 164, 168)

double * d = new double[200];

```
0 1 33 199
```

*(d+33) is the same as d[33]
if d is 0x00b0, then d+1 is 0x00b8, 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];

0 1 32 33 98 99

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

```cpp
#include <iostream.h>

main()
{
    int *a = new int[100];
    int *p = a;
    int k;
    for (k = 0; k < 100; k++)
    {
        *p = k;
        p++;
    }
    cout << "entry 3 = " << a[3] << endl;
}
```
Array of Classes (Linked List)

```cpp
#include <iostream>
class node {
public:
    int me;
    node *next;
};
main()
{
    node *ar = new node[10];
    int k;
    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) {
        cout << p->me << " " <<
        hex << p << " " << p->next <<
        endl;
        p = p->next;
    }
    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 p</td>
<td>p-&gt;next</td>
<td>me is int (4 bytes)</td>
<td>next is node* (4 bytes)</td>
</tr>
<tr>
<td>0x26ca8 0x26cb0</td>
<td>0</td>
<td>ar[0]</td>
<td></td>
</tr>
<tr>
<td>0x26cb0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26cb8 0x26cc0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26cc0 0x26cc8</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26cc8 0x26cd0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26cd0 0x26cd8</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26cd8 0x26ce0</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26ce0 0x26ce8</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26ce8 0x26cf0</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26cf0</td>
<td>9</td>
<td>ar[9]</td>
<td></td>
</tr>
</tbody>
</table>
Array of Classes with Inheritance

#include <iostream.h>
class course {
public:
int number;
int score;
float average;
};
class node : public course {
public:
int me;
node *next;
};
main()
{
ode *ar = new node[10];
node *p = ar;
int *num_ptr, *me_ptr;
int k;
for (k = 0; k < 9; k++)
{
p->me = k;
p->number = 104;
p->score = k*20;
p->average = 0.96;
p->next = &ar[k+1];
p++;
}
p->me = 9;
p->number = 104;
p->score = k*20;
p->average = 0.96;
p->next = NULL;
num_ptr = &p->number;
me_ptr = &p->me;
cout << p->me << " " << *me_ptr
<< " " << p->number << " " <<
*num_ptr << endl;
}

Memory Layout

```
0x2ad40 104 number

0x2ad54 score

0x2ad54 average

0x2ad54 me

0x2adf4 next

p

num_ptr

me_ptr

...```

for (k = 0; k < 9; k++)
{
p->me = k;
p->number = 104;
p->score = k*20;
p->average = 0.96;
p->next = &ar[k+1];
p++;
}
p->me = 9;
p->number = 104;
p->score = k*20;
p->average = 0.96;
p->next = NULL;
num_ptr = &p->number;
me_ptr = &p->me;
cout << p->me << " " << *me_ptr
<< " " << p->number << " " <<
*num_ptr << endl;
}
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)

```
0 1 15 16 42 43
```

Strlen()

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

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

- Memory
- 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 binary number
**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 0x00ff**
  
  \[
  xpos = 0x1a34c \ & \ 0x000ff
  \]

<table>
<thead>
<tr>
<th>button</th>
<th>y</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1a34c</td>
<td>01 1010 0011 0100 1100</td>
<td></td>
</tr>
<tr>
<td>0x000ff</td>
<td>00 0000 0000 1111 1111</td>
<td></td>
</tr>
<tr>
<td>0x0004c</td>
<td>00 0000 0000 0100 1100</td>
<td></td>
</tr>
</tbody>
</table>

### 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 1010 0011 0100 1100</td>
<td></td>
</tr>
<tr>
<td>0x000ff</td>
<td>00 1111 1111 0000 0000</td>
<td></td>
</tr>
<tr>
<td>0xa300</td>
<td>00 1010 0011 0000 0000</td>
<td></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
  
  $$\text{ypos} = (0x1a34c \ & \ 0x0ff00) \ >> \ 8$$

  $$\text{button} = (0x1a34c \ & \ 0x30000) \ >> \ 16$$

Extracting Parts of Floating Point Number

- See web page for full code
  
  ```cpp
  #define EXP_BITS 8
  #define MANTISSA_BITS 23
  #define SIGN_MASK 0x80000000
  #define EXP_MASK 0x7f800000
  #define MANTISSA_MASK 0x007fffff

  class myfloat {
    public:
      int sign;
      unsigned int exp;
      unsigned int mantissa;
  };

  num->sign = (x & SIGN_MASK) >> (EXP_BITS + MANTISSA_BITS);
  num->exp  = (x & EXP_MASK) >> MANTISSA_BITS;
  num->mantissa = x & MANTISSA_MASK;
  ```

x is 32-bit word
## Summary

- **Homework #1 September 12**
- 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