Midterm Review

15-213: Introduction to Computer Systems
Recitation 8: Monday, Oct. 13, 2014
Lou Clark
Agenda

- Midterm Logistics
- Brief Overview of *some* topics
- Practice Questions
Midterm

■ Tues Oct 14\textsuperscript{th} to Fri Oct 17\textsuperscript{th}.
  ▪ Duration – Designed to be take in 80min, but you have up to 4 hrs
  ▪ If you have not signed up for a slot online, \textbf{do so now}.
    ▪ You will only be allowed to take it during your slot

■ Cheat Sheet – ONE double sided 8 ½ x 11 paper
  ▪ No worked out problems in that sheet

■ No office hours after Monday
  ▪ After that, you can still email the list
  ▪ Responses might be slow due to volume, so be proactive, and read the book/lectures slides carefully beforehand
Midterm

What to study?
- Chapters 1-3 and Chapter 6

How to Study?
- Read each chapter 3 times, work practice problems and do problems from previous exams.
- Online practice exam allows you to get a feel for the format of the exam
Bits, Bytes & Integers

- Know how to do basic bit operations by hand
  - Shifting, addition, negation, and, or, xor, etc.

- If you have w bits
  - What are the largest/smallest representable signed numbers?
  - What are the largest/smallest representable unsigned numbers?
  - What happens to the bits when casting signed to unsigned (and vice versa)?

- Distinguish between logical and bitwise operators

- What happens in C if you do operations on mixed types (either different size, or signedness?)
Floating Point (IEEE Format)

- Sign, Exponent, Mantissa
  - $(-1)^s \times M \times 2^E$
  - $s$ – sign bit
  - $M$ – Mantissa/Fraction bits
  - $E$ – Determined by (but not equal to) exponent bits

- Bias ($2^{k-1} - 1$)

- Three main categories of floats
  - Normalized: Large values, not near zero
  - Denormalized: Small values close to zero
  - Special Values: Infinity/NaN
Floating Point (IEEE Format)

<table>
<thead>
<tr>
<th>Represents:</th>
<th>Normalized</th>
<th>Denormalized</th>
<th>Special Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent bits:</td>
<td>Not those →</td>
<td>000...000</td>
<td>111...111</td>
</tr>
<tr>
<td>E =</td>
<td>exp – bias</td>
<td>1 – bias</td>
<td>+/− ∞ if frac = 000...000; otherwise NaN</td>
</tr>
<tr>
<td>M =</td>
<td>1.frac</td>
<td>.frac</td>
<td></td>
</tr>
</tbody>
</table>

- **Floating Point Rounding**
  - Round-up – if the spilled bits are greater than half
  - Round-down – if the spilled bits are less than half
  - Round to even – if the spilled bits is exactly equal to half
Floating point encoding. In this problem, you will work with floating point numbers based on the IEEE floating point format. We consider two different 6-bit formats:

Format A:

- There is one sign bit $s$.
- There are $k = 3$ exponent bits. The bias is $2^{k-1} - 1 = 3$.
- There are $n = 2$ fraction bits.

Format B:

- There is one sign bit $s$.
- There are $k = 2$ exponent bits. The bias is $2^{k-1} - 1 = 1$.
- There are $n = 3$ fraction bits.

For formats A and B, please write down the binary representation for the following (use round-to-even). Recall that for denormalized numbers, $E = 1 - \text{bias}$. For normalized numbers, $E = e - \text{bias}$.

<table>
<thead>
<tr>
<th>Value</th>
<th>Format A Bits</th>
<th>Format B Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0 000 00</td>
<td>0 00 000</td>
</tr>
<tr>
<td>One</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1/2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$11/8$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assembly Loops

- Recognize common assembly instructions
- Know the uses of all registers in 32 and 64 bit systems
- Understand how different control flow is turned into assembly
  - For, while, do, if-else, switch, etc
- Be very comfortable with pointers and dereferencing
  - The use of parens in mov commands.
    - %eax vs. (%eax)
  - The options for memory addressing modes:
    - R(Rb, Ri, S)
    - lea vs. mov
Assembly Loop

void mystery(int *array, int n)
{
    int i;
    for(_______: _______; _______)
    {
        if(___________ == 0)
            _________;
    }
}
Assembly – Stack

- **How arguments are passed to a function**
  - IA-32
  - X86-64

- **Return value from a function**

- **How these instructions modify stack**
  - call
  - leave
  - ret
  - pop
  - push
Given assembly code of foo() and bar(), draw a detailed picture of the stack, starting with the caller invoking foo(3, 4, 5).

**Value of ebp when foo is called:** 0xffffd858  
**Return address in function that called foo:** 0x080483c9

<table>
<thead>
<tr>
<th>Stack address</th>
<th>The diagram starts with the arguments for foo()</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffd850</td>
<td></td>
</tr>
<tr>
<td>0xffffd850</td>
<td>5</td>
</tr>
<tr>
<td>0xffffd84c</td>
<td></td>
</tr>
<tr>
<td>0xffffd848</td>
<td></td>
</tr>
<tr>
<td>0xffffd844</td>
<td></td>
</tr>
<tr>
<td>0xffffd840</td>
<td></td>
</tr>
<tr>
<td>0xffffd83c</td>
<td></td>
</tr>
<tr>
<td>0xffffd838</td>
<td></td>
</tr>
<tr>
<td>0xffffd834</td>
<td></td>
</tr>
<tr>
<td>0xffffd830</td>
<td></td>
</tr>
</tbody>
</table>
int bar (int a, int b) {
    return a + b;
}

int foo(int n, int m, int c) {
    c += bar(m, n);
    return c;
}

08048374 <bar>:
8048374:  55          push  %ebp
8048375:  89 e5       mov  %esp,%ebp
8048377:  8b 45 0c    mov  0xc(%ebp),%eax
804837a:  03 45 08    add  0x8(%ebp),%eax
804837d:  5d          pop  %ebp
804837e:  c3          ret

0804837f <foo>:
804837f:  55          push  %ebp
8048380:  89 e5       mov  %esp,%ebp
8048382:  83 ec 08    sub  $0x8,%esp
8048385:  8b 45 08    mov  0x8(%ebp),%eax
8048388:  89 44 24 04 mov  %eax,0x4(%esp)
804838c:  8b 45 0c    mov  0xc(%ebp),%eax
804838f:  89 04 24    mov  %eax,(%esp)
8048392:  e8 dd ff ff ff call 8048374 <bar>
8048397:  03 45 10    add  0x10(%ebp),%eax
804839a:  c9          leave
804839b:  c3          ret
Array Access

- A suggested method for these problems:
  - Start with the C code
  - Then look at the assembly Work backwards!
  - Understand how in assembly, a logical 2D array is implemented as a 1D array, using the width of the array as a multiplier for access

| [0][0] = [0] | [0][1] = [1] | [0][2] = [2] | [0][3] = [3] |
| [1][0] = [4] | [1][1] = [5] | [1][2] = [6] | [1][3] = [7] |
| [2][0] = [8] | [2][1] = [9] | [2][2] = [10] | [2][3] = [11] |

\[
[0][2] = 0 \times 4 + 2 = 2 \\
[1][3] = 1 \times 4 + 3 = 7 \\
[2][1] = 2 \times 4 + 1 = 9 \\
\]

\([i][j] = i \times \text{width of array} + j\)
```c
int array1[H][J];
int array2[J][H];

int copy_array(int x, int y) {
    array2[y][x] = array1[x][y];
    return 1;
}
```

Suppose the above C code generates the following x86-64 assembly code:

```
# On entry:
#   %edi = x
#   %esi = y
#
# copy_array:
movslq   %esi,%rsi
movslq   %edi,%rdi
movq     %rsi, %rax
salq     $4, %rax
subq     %rsi, %rax
addq     %rdi, %rax
leaq     (%rdi,%rdi,2), %rdi
addq     %rsi, %rdi
movl     array1(,%rdi,4), %edx
movl     %edx, array2(,%rax,4)
movl     $1, %eax
ret
```
Caching Concepts

- **Dimensions: S, E, B**
  - S: Number of sets
  - E: Associativity – number of lines per set
  - B: Block size – number of bytes per block (1 block per line)

- **Given Values for S,E,B,m**
  - Find which address maps to which set
  - Is it a Hit/Miss. Is there an eviction
  - Hit rate/Miss rate

- **Types of misses**
  - Which types can be avoided?
  - What cache parameters affect types/number of misses?
Questions/Advice

- Relax!
- Work Past exams!
- Email us - (15-213-staff@cs.cmu.edu)