Midterm Review

15-213: Introduction to Computer Systems
Recitation 8: Monday, Oct. 14, 2013
Marjorie Carlson
Section A
Agenda

- Midterm Logistics

- A Non-Comprehensive Review of Midterm Topics:
  - Some skills you should have
  - A sample exam problem
Midterm

- **Wed Oct 16th to Sun Oct 20th**
  - Check website for times & locations
  - Intended to take 80 minutes; you’ll have 4 hours

- **Cheat sheet – ONE double-sided 8 ½ x 11 paper**
  - Cannot include pre-worked problems

- **What to study?**
  - Chapters 1-3 and Chapter 6
  - Lectures through cache

- **How to study?**
  - Read each chapter (preferably multiple times) & do problems
  - Do problems from previous exams (incl. cache from midterm 2)
  - Office hours tonight & tomorrow only

- **More info can be found** [here](#).
Agenda

- Midterm Logistics

- A Non-Comprehensive Review of Midterm Topics:
  - Ints & two’s complement
  - Floats
  - Structs (sizes & alignment of data)
  - Assembly loops
  - The stack
  - Array access
  - Overview of caching
Bits, Bytes & Integers

- Some suggested skills:
  - Know how to do basic bitshifting by hand.
  - If you represent numbers in \( w \) bits:
    - what’s UMax (the largest unsigned number)? \( 2^w - 1 \)
    - what’s Tmax (the largest signed number)? \( 2^{w-1} - 1 \)
    - what’s Tmin (the most negative signed number)? \( -2^{w-1} \)
  - How do you calculate \(-x\) in two’s complement? \( \sim x + 1 \)
For this problem, assume the following:

- We are running code on an 8-bit machine using two’s complement arithmetic for signed integers.
- `short` integers are encoded using 4 bits.
- Sign extension is performed whenever a `short` is cast to an `int`.

The following definitions are used in the table below:

```c
short sa = -6;
int b = 2*sa;
short sc = (short)b;
int x = -64;
unsigned ux = x;
```

Fill in the empty boxes in the table. If the expression is cast to or stored in a `short`, use a 4-bit binary representation. Otherwise assume an 8-bit binary representation. The first 2 lines are given to you as examples, and you need not fill in entries marked with “—”.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Decimal Representation</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>(short)0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>−</td>
<td>−17</td>
<td></td>
</tr>
<tr>
<td>−</td>
<td></td>
<td>0010 1001</td>
</tr>
<tr>
<td>sa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ux</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tmax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tmax – Tmin</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Floating Point

**Some suggested skills:**

- Know how to represent a fraction in binary scientific notation.

- Be able to convert between this format:
  \[ (-1)^s \times M \times 2^E \]

  and this format:

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
</table>

- Be able to compute bias: \( 2^{k-1} - 1 \) where \( k \) is the number of exp bits.

- Know how to convert normalized to denormalized & vice versa, when to round up/down/to even and when numbers round to 0/\( \infty \).
Floating Point

<table>
<thead>
<tr>
<th>Represents:</th>
<th>Normalized</th>
<th>Denormalized</th>
<th>Special Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Most numbers</td>
<td>Tiny numbers</td>
<td>Infinity, NaN</td>
</tr>
<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 =</td>
</tr>
<tr>
<td>M =</td>
<td>1.frac</td>
<td>.frac</td>
<td>000...000; otherwise NaN</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 are 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>
Sizes & Alignment

Know all the basic data types’ sizes, alignment rules (32-bit and 64-bit), and assembly suffixes:

- char
- short
- int
- float
- long
- pointer
- long long
- double
- long double
typedef struct {
    char a;
    long b;
    float c;
    char d[3];
    int *e;
    short *f;
} foo;

1. Show how `foo` would be allocated in memory on an x86-64 Linux system. Label the bytes with the names of the various fields and clearly mark the end of the struct. Use an X to denote space that is allocated in the struct as padding.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

2. Rearrange the elements of `foo` to conserve the most space in memory. Label the bytes with the names of the various fields and clearly mark the end of the struct. Use an X to denote space that is allocated in the struct as padding.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| | | | | | | | | | | | | | | | | |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Fall 2012; answers here.
Assembly Loops

- **Some suggested skills:**
  - Know the basic assembly instructions and jump codes.
  - Get comfortable with the order of operands!
    - `sub %edx,%eax`  ➔  `%eax = %eax-%edx`
  - Get comfortable with dereferencing and addresses:
    - `%edx` vs. `(%edx)`
    - `D(Rb, Ri, S)` ➔  `Mem[Reg[Rb] + S * Reg[Ri] + D]`
  - Hint: if statements often end up “backwards” in Assembly
    ```
    if (n < 5)
      do something;
    else
      do something else;
    ```
    ```
    cmp $0x5,%edx
    jge <something else>
    [do something]
    ```
Assembly Loops

void mystery(int *array, int n)
{
    int i;
    for (________; _______; _______)
    {
        if (__________________)
            _________;
    }
}
The Stack

- **Some suggested skills:**
  - Know how arguments are passed to a function.
    - x86-32?
    - x86-64?
  - Know where to find the return value from a function.
    - x86-32?
    - x86-64?
  - Know how these instructions modify the 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: 0xfffffd858
Return address in function that called foo: 0x080483c9

Stack addresss
arguments for foo()
+-----------------------------------+
| 0xfffffd850 | 5
+-----------------------------------+
| 0xfffffd84c |
+-----------------------------------+
| 0xfffffd848 |
+-----------------------------------+
| 0xfffffd844 |
+-----------------------------------+
| 0xfffffd840 |
+-----------------------------------+
| 0xfffffd83c |
+-----------------------------------+
| 0xfffffd838 |
+-----------------------------------+
| 0xfffffd834 |
+-----------------------------------+
| 0xfffffd830 |
+-----------------------------------+

Fall 2012; answers here.
Problem 8. (10 points):

Stack discipline.

Consider the following C code and its corresponding 32-bit x86 machine code. Please complete the stack diagram on the following page.

```c
int bar (int a, int b) {
    return a + b;
}

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

```asm
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:

Understand how assembly turns a 2D array into a 1D array by using the width of the array as a multiplier.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0][0] = [0]</td>
<td>[0][1] = [1]</td>
<td>[0][2] = [2]</td>
<td>[0][3] = [3]</td>
</tr>
<tr>
<td>[1][0] = [4]</td>
<td>[1][1] = [5]</td>
<td>[1][2] = [6]</td>
<td>[1][3] = [7]</td>
</tr>
<tr>
<td>[2][0] = [8]</td>
<td>[2][1] = [9]</td>
<td>[2][2] = [10]</td>
<td>[2][3] = [11]</td>
</tr>
</tbody>
</table>

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

\[[i][j] = i * \text{width of array} + j\]
Problem 3. (6 points): Accessing arrays.

Consider the C code below, where $H$ and $J$ are constants declared with `#define`.

```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:

```assembly
# 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,%rdi,4), %edx
movl %edx, array2(%rax,%rax,4)
movl $1, %eax
ret
```

What are the values of $H$ and $J$?

$H =$

$J =$
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/s, E, B/b, m**
  - Find which address maps to which set
  - Is it a hit or miss? Is there an eviction?
  - What’s the hit rate or miss rate for a given piece of C code?
Questions/Advice

- Work past exams!
- Once you’re in the exam: relax! You have time.
- Email us: 15-213-staff@cs.cmu.edu
- Office hours tonight and tomorrow