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? **~ 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:

```
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 "—".

Expression	Decimal Representation	Binary Representation
Zero	0	0000 0000
(short)0	0	0000
—	−17	
—		0010 1001
sa		
b		
sc		
ux		
TMax		
TMax − TMin		

Floating Point

- **Some suggested skills:**
  - Know how to represent a fraction in binary scientific notation.
  - Be able to convert between this format:

$$(-1)^s * M * 2^E$$

    and this format:

s	exp	frac

  - 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/ ∞.

Floating Point

	Normalized	Denormalized	Special Values
Represents:	Most numbers	Tiny numbers	Infinity, NaN
Exponent bits:	Not those →	000…000	111…111
E =	exp – bias	1 – bias	$^{+}/_{-}$ ∞ if frac = 000…000; otherwise NaN
M =	1.frac	.frac	

- 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.

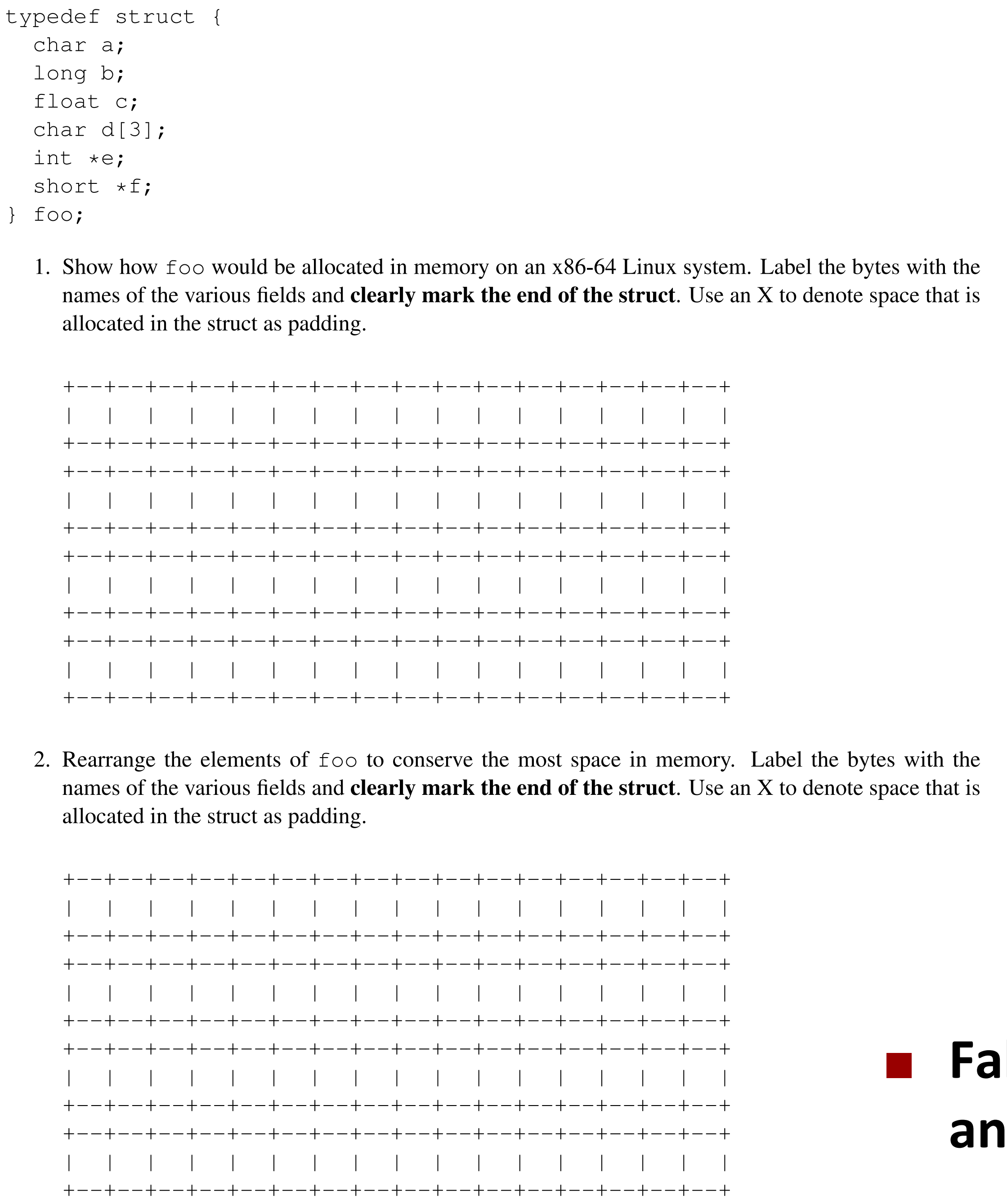
For formats A and B, please write down the binary representation for the following (use round-to-even). Recall that for denormalized numbers, $E = 1 -$ bias. For normalized numbers, $E = e -$ bias.

Value	Format A Bits	Format B Bits
Zero	0 000 00	0 00 000
One		
$1/2$		
$11/8$		

- **Fall 2012; answers here.**

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

```
080483a0 <mystery>:
 80483a0:   push   %ebp
 80483a1:   mov    %esp,%ebp
 80483a3:   push   %ebx
 80483a4:   mov    0xc(%ebp),%ebx
 80483a7:   mov    0x8(%ebp),%ecx
 80483aa:   test   %ebx,%ebx
 80483ac:   jle    80483c5 <mystery+0x25>
 80483ae:   xor    %eax,%eax
 80483b0:   mov    (%ecx,%eax,4),%edx
 80483b3:   add    $0x1,%edx
 80483b6:   cmp    $0x2,%edx
 80483b9:   ja     80483be <mystery+0x1e>
 80483bb:   mov    %edx,(%ecx,%eax,4)
 80483be:   add    $0x1,%eax
 80483c1:   cmp    %ebx,%eax
 80483c3:   jne    80483b0 <mystery+0x10>
 80483c5:   pop    %ebx
 80483c6:   pop    %ebp
 80483c7:   ret
```

```
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: 0xffffd858
Return address in function that called foo: 0x080483c9

Stack      The diagram starts with the
addresss   arguments for foo()
           +-----------------------------------+
0xffffd850 |                5                  |
           +-----------------------------------+
0xffffd84c |                                   |
           +-----------------------------------+
0xffffd848 |                                   |
           +-----------------------------------+
0xffffd844 |                                   |
           +-----------------------------------+
0xffffd840 |                                   |
           +-----------------------------------+
0xffffd83c |                                   |
           +-----------------------------------+
0xffffd838 |                                   |
           +-----------------------------------+
0xffffd834 |                                   |
           +-----------------------------------+
0xffffd830 |                                   |
           +-----------------------------------+
```

- **Fall 2012; answers here.**

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

[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 * 4 + 2 = 2

[1][3] = 1 * 4 + 3 = 7

[2][1] = 2 * 4 + 1 = 9

[i][j] = i * width of array + j

```
int array1[H][J];
int array2[J][H];

int copy_array(int x, int y) {
    array2[y][x] = array1[x][y];

    return 1;
}
```

Find H & J

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
```

- **Fall 2010; answers here.**

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