University of Washington — Computer Science & Engineering

CSE 351, Summer 2019 — Midterm Exam

Friday, July 26th, 2019

Name: ____________________

UW NetID: ____________________ @uw.edu

Name of student to your left: ____________________

Name of student to your right: ____________________

I certify that all work is my own. I had no prior knowledge of exam contents nor will I share the contents with any student in CSE 351 who has not yet taken the exam. Violation of these terms may result in a failing grade. *(Please sign below.)*

Signature: ____________________

Instructions

- You may fill out this page, but **do not turn the page until 10:50am.**
- This exam is closed-book, except for one *handwritten* double-sided 8.5×11" note sheet. Cell phones, smart watches, notes written underneath your sleeves, Google Glasses, Hololens, neural links, and any other futuristic devices are not allowed.
- You have 60 minutes to complete the exam. Please stop promptly at 11:50am.
- The last page of the exam is a reference sheet. Please detach it before turning in your exam.
- Write your UW NetID (*not* your student ID number) on the top-right corner of each page.
- We will scan your exams to grade them. Please write *clearly* and *legibly*.
- There are 6 questions, totaling 80 points, across 8 pages (including this one).

Advice

- Read the questions thoroughly before answering.
- Write down your thoughts and intermediate steps so that you can get partial credit. But be sure to clearly indicate your final answer.
- Questions are not necessarily in order of difficulty. Skip around or read ahead. Make sure you have a chance to attempt all the questions.
- *Relax*. You are here to learn ☺.

Question:	1	2	3	4	5	6	Total
Points:	17	15	15	16	5	12	80

1. (17 points) Number Representation

In an effort to save space (and sanity), we've invented a new integer number type called `int_ten` that is only 10 bits wide. For this question, you may write your answers as a sum of powers of two unless otherwise specified.

Suppose we define a new `int_ten` variable:

```
int_ten x = 0b1110001001;
```

(a) (1 point) Write down `x` in *hexadecimal*.

> **Solution:** `0x389`

(b) (1 point) Interpreting `x` as an *unsigned* 10-bit integer, what is its decimal value?

> **Solution:** 905. $(2^9 + 2^8 + 2^7 + 2^3 + 2^0)$

(c) (2 points) Interpreting `x` as a *(signed) two's complement* 10-bit integer, what is its decimal value?

> **Solution:** -119. $(-2^9 + 2^8 + 2^7 + 2^3 + 2^0)$

Now we've defined another new floating-point number type, `float_ten`, that is also 10 bits wide. This floating-point type uses 1 bit for the sign, 3 bits for the exponent, and 6 bits for the mantissa. The layout of sign, exponent, and mantissa, and representation of special values, is the same as for a 32-bit IEEE floating-point number.

(d) (2 points) What is the bias for `float_ten` numbers?

> **Solution:** The bias is $2^{E-1} - 1$. The exponent is 3 bits, so the bias is $2^2 - 1 = 3$.

(e) (3 points) What *decimal* number does the bit pattern `0b1110001001` represent in this floating point encoding?

> **Solution:** E is 110_2, which is 6_{10}. We subtract the bias, so the exponent is $6 - 3 = 3$. The sign bit is 1 so the number is negative. In binary scientific notation: $-1.001001 * 2^3$. $-1001.001_2 = -9.125_{10}$.

Consider this silly C code:

```
1 #include <math.h>
2
3 void gillyweed() {
4     float hagrid = (float) (1 << 24);
5     while (hagrid < INFINITY) {
6         hagrid += 1.0;
7     }
8 }
```

(f) (2 points) On line 4, what *decimal* value is `hagrid` set to? Explain in 1-2 sentences.

> **Solution:** `hagrid` will be set to the value 2^{24}, since when casting from an integer value to a floating-point value, the bits will be changed so that the value remains the same.

(g) (3 points) Will `gillyweed` ever return? Explain in 1-2 sentences.

> **Solution:** No. 2^{24} is more than 23 bits away from $1 = 2^0$ and we only have 23 bits in the mantissa, so it will be rounded off.

(h) (3 points) If we change `hagrid` to be a `double` instead of a `float`, will `gillyweed` ever return? Explain in 1-2 sentences.

> **Solution:** No. Though the 1 would not be *immediately* rounded off, continuously adding 1 to any number will eventually cause it to be rounded off as the number grows sufficiently large.

Why do assembly programmers need to wear scuba masks? . . .

2. (15 points) Pointers & Memory

For this question, refer to the C assignments and memory diagram below, with addresses increasing left-to-right and top-to-bottom. Remember that x86-64 machines are little endian.

```
int *i = 0x10;
char *c = 0x2c;
long *l = 0x08;
```

Address	+0	+1	+2	+3	+4	+5	+6	+7
0x00	1e	00	00	00	00	00	00	00
0x08	aa	bb	cc	dd	ee	ff	00	11
0x10	08	07	06	05	04	03	02	01
0x18	53	61	6d	20	69	73	20	73
0x20	75	70	65	72	20	63	6f	6f
0x28	6c	2e	9d	ab	b6	2d	e7	99

(a) (10 points) Fill in the C type and hex value for each of the following C expressions. Assume that `0x00` is a valid memory address (i.e., not a null pointer).

C Expression	C Type	Hex Value
`*i`	`int`	`0x05060708`
`l+1`	`long *`	`0x10`
`*(c+2)`	`char`	`0xe7`
`i[-2]`	`int`	`0xddccbbaa`
`** ((short **) (l-1))`	`short`	`0x7320`

(b) (5 points) Determine the final value, in hex, of each of these registers after executing the instructions shown on the left. Assume that all registers start with the value `0x0`, except `%rdi`, which initially has the value `0xc`. Write out bytes to fill out the *entire width* of the specified register.

Instruction	Register	Hex Value
	`%rdi`	`0x000000000000000c`
`movw %di, %bx`	`%bx`	`0x000c`
`leal (%edi,%edi,2), %eax`	`%eax`	`0x00000024`
`movswl (%rdi), %edx`	`%rdx`	`0x00000000ffffffee`

. . . because they work below C level! ☺

3. (15 points) C & Assembly

You are given the following mysterious-looking function in x86-64 assembly:

```
mystery:
.L4:
        movzbl  (%rdi), %eax
        testb   %al, %al
        je      .L2
        leal    -97(%rax), %edx
        cmpb    $25, %dl
        ja      .L3
        subl    $32, %eax
        movb    %al, (%rdi)
.L3:
        addq    $1, %rdi
        jmp     .L4
.L2:
        rep ret
```

(a) (2 points) What variable type is `%rdi` in the corresponding C program?

(a) char *

(b) (9 points) Fill in the missing parts of the C code that is equivalent to the assembly above:

```
void mystery((answer to a) x) {
    while (*x != '\0' or 0 ) {
        if (*x >= 97 && *x <= 122 ) {
            *x = *x - 32 ;
        }
        x++ ;
    }
}
```

(c) (4 points) On a high level, what does this function *accomplish*? Explain in 1-2 sentences. **Hint**: the ASCII character code for the letter `'a'` is 97, and the code for `'A'` is 65.

Solution: It converts letters in a string from lowercase to uppercase. (There's room for partial credit here.)

4. (16 points) Procedures & The Stack

The recursive function `fact` calculates the factorial of its argument `n`. This function, along with its x86-64 assembly, is shown below.

```
long fact(long n) {
    if (n < 2)
        return 1; // BREAKPOINT HERE
    else
        return n * fact(n-1);
}
```

```
00000000004004b7 <fact>:
    4004b7:    cmp     $0x1,%rdi
    4004bb:    jg      4004c3 <fact+0xc>
    4004bd:    mov     $0x1,%eax
    4004c2:    retq    # BREAKPOINT HERE
    4004c3:    push    %rbx
    4004c4:    mov     %rdi,%rbx
    4004c7:    lea     -0x1(%rdi),%rdi
    4004cb:    callq   4004b7 <fact>
    4004d0:    imul    %rbx,%rax
    4004d4:    pop     %rbx
    4004d5:    retq
```

(a) (2 points) The addresses shown above are part of which section of memory?

(a) **Instructions or code**

(b) (2 points) During a recursive call to `fact`, what return address is pushed on to the stack? Answer in hex.

(b) 0x4004d0

(c) (2 points) Where in this code is `n` saved before `fact` makes a recursive call? Give the address of the corresponding *assembly instruction*.

(c) 0x400c4

As a matter of `fact`, this question continues on the next page. . . ☺

(d) (10 points) Assume that `main` calls `fact(4)`. Fill in a memory diagram of the stack when we hit the breakpoint shown above (on C line 3, or assembly line 5). Include a brief description (1-3 words) of the each entry, as well as its value (if known). *You may not need all the lines provided.*

Address	Description	Value
0x7fffffffdce8	return to `main`	*unknown*
0x7fffffffdce0	saved `%rbx` from `main`	*unknown*
0x7fffffffdcd8	**return to** `fact`	0x4004d0
0x7fffffffdcd0	**saved** `%rbx`	4
0x7fffffffdcc8	**return to** `fact`	0x4004d0
0x7fffffffdcc0	**saved** `%rbx`	3
0x7fffffffdcb8	**return to** `fact`	0x4004d0
0x7fffffffdcb0	***unknown***	***unknown***

5. (5 points) Building an Executable

(a) (1 point) Give an example of a *valid* assembly instruction that the *assembler* cannot fully translate to completed machine code.

(a) `call`, **jumps, labels**

(b) (1 point) Which table in an object file holds information about the methods, global variables, and other data defined in that file?

(b) **symbol table**

(c) (3 points) In order, what four steps are required to produce and run a completed binary from C source files?

Solution: Compiling, Assembling, Linking, Loading (0.5 pt for each step, 1 pt for correct order)

6. (12 points) System & Architecture Design

   Your intrepid instructor is founding a new company, WolfBytes™, where he plans to sell CPUs that implement the x86-64 instruction set—but *even better* than Intel does! He needs your help to figure out how to design these chips!

   (a) (3 points) Sam decides that Intel hasn't put enough registers in their chips. Therefore, he decides to build in separate registers for each different data size (e.g., `%rax` and `%eax` now refer to entirely different registers and don't share any space). This will allow compilers to have so much more rapidly-accessible space in the CPU! *Does this still implement the x86-64 instruction set? Explain briefly (1-2 sentences).*

   Solution: No. Existing code will not run correctly with these changes, since it will expect that referring to registers by their different names will still give you (partial) views of the same data.

   (b) (3 points) Instead, Sam decides to double the size of each register, so that we can store larger data types. He gives these new 128-bit registers new names, and doesn't change any of the existing register names (e.g., `%rdi` still refers to a 64-bit register, etc.). *Will this remain compatible with x86-64 programs? Explain briefly (1-2 sentences).*

   Solution: Yes. Existing x86-64 code will still run correctly, it just won't take advantage of the larger registers.

   (c) (4 points) Try as he might, Sam simply cannot figure out how Intel made their `imul` (integer multiply) instruction run so quickly. He decides that he will have to implement multiplication in his chips by using addition instead. Therefore, his `imul` instruction does produce the right results, but it runs much more slowly. *Is this still a valid x86-64 implementation? Explain briefly (1-2 sentences).*

   Solution: Yes, the architecture doesn't specify anything about speed.

   (d) (2 points) What is your favorite text editor?

   ○ `emacs` ○ `ubertext` ○ `ed` √ `wolfedit` ○ `pico` ○ `vim` ○ `nano`

Did you write your UW NetID on the top-right corner of each page? 🐱