Name:_____

CSE351 Winter 2016, Midterm Examination
February 8, 2016

Please do not turn the page until 11:30.

Rules:

- The exam is closed-book, closed-note, etc.
- **Please stop promptly at 12:20.**
- There are **110 (not 100) points**, distributed **unevenly** among **7** questions (all with multiple parts):
- **The exam is printed double-sided.**
- The last two pages of the exam have useful reference information.

Advice:

- Read questions carefully. Understand a question before you start writing.
- **Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.**
- The questions are not necessarily in order of difficulty. **Skip around.** Make sure you get to all the questions.
- If you have questions, ask.
- Relax. You are here to learn.

Name:_____

1. (**20** points) 32-bit integers

   Hint/note: If you forget the exact details of little-endian, do not panic: You can probably still get a lot of this question correct.

   In writing your answers, be sure to put the byte with the lowest address on the left (and the byte with the highest address on the right).

   (a) Write the decimal number 13 in binary (base-2) as a 32-bit big-endian int.

   (b) Write the decimal number 13 in hexadecimal (base-16) as a 32-bit big-endian int.

   (c) Write the decimal number 13 in binary (base-2) as a 32-bit little-endian int.

   (d) Write the decimal number 13 in hexadecimal (base-16) as a 32-bit little-endian int.

   (e) For each of the following, answer *same* if the big-endian and little-endian representations are the same, else answer *different*. Assume all numbers are 32-bit ints.

      i. The minimum unsigned number
      ii. The maximum unsigned number
      iii. +1 in twos-complement
      iv. -1 in twos-complement
      v. The maximum twos-complement number
      vi. The minimum (most-negative) twos-complement signed number

   (f) Give in hexadecimal (base-16) a 32-bit value that represents the same *unsigned* number in big-endian or little-endian and is not one of the choices in the previous problem.

   Solution:
   (Spaces have no significance.)

   (a) 0000 0000 0000 0000 0000 0000 0000 1101

   (b) 00 00 00 0D

   (c) 0000 1101 0000 0000 0000 0000 0000 0000

   (d) 0D 00 00 00

   (e)
      i. same
      ii. same
      iii. different
      iv. same
      v. different
      vi. different

   (f) Many possible choices – here is one: FF 00 00 FF (need the two outside bytes the same and the two inside bytes the same)

Name:_____

2. (**16** points) Pointers and C

Consider this C code:

```
int a[6];
int * b = a;
for(int i=0; i < 6; ++i) {
   a[i] = i*7;
}
a[2] = 3;
a += 3;
a[2] = 5;
a[-2] += 1;
```

Errata: array "a" cannot be modified with +=. This example should have used another pointer instead of the array, like so:

```
int a[6];
int * b = a;
int * c = a;
for (int i=0; i < 6; ++i) {
  c[i] = i*7;
}
c[2] = 3;
c += 3;
c[2] = 5;
c[-2] += 1;
```

Suppose the space for array `a` is allocated beginning at address 500 (in base-10).

(a) Fill in the boxes below to indicate the contents of memory at the addresses indicated immediately after all the code above executes. The addresses are written in base-10 (decimal) and your answers should be written in base-10 (decimal).

someFunction(c,b)

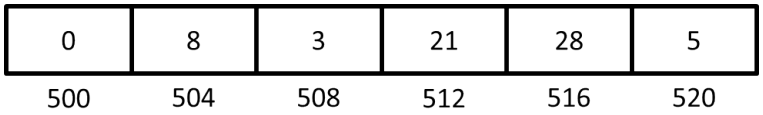
(b) Suppose we call ~~someFunction(a,b);~~ immediately after the code above executes.

i. What value does the callee receive for the first argument? (Answer in base-10.)

ii. What value does the callee receive for the second argument? (Answer in base-10.)

iii. What is the *size in bytes* of each argument passed to `someFunction` (on x86-64)?

Solution:



(a)

(b) i. 512

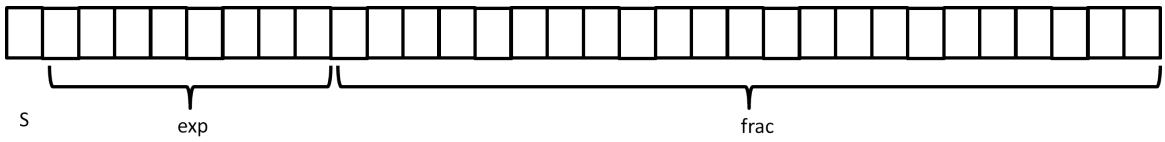
ii. 500

iii. 8 bytes each

Name:_____

3. (**19** points) Floating Point

(a) Consider the decimal number 1.25. Give the IEEE-754 representation of this number as a 32-bit floating-point number by filling in the diagram below. Hint: Remember bias and any implicit bits. Consider explaining your work for potential partial credit but explanations not necessary for correct answers.



(b) The IEEE-754 representation for 0 as a 32-bit floating-point number is a "special case" in the standard.

i. What is the bit-representation of (positive) 0?

ii. If this were not a special case in the standard, what floating-point number would these bits represent? Give your answer in the form $a \cdot 2^b$.

(c) If `x` and `y` have type `float`, give *two* (very) different reasons that `(x+2*y)-y == x+y` might evaluate to `0` (i.e., false). Describe each reason in roughly 2 English sentences: what is the term used to describe the issue, what happens as a result, and what sort of values for `x` and `y` could cause it? You do *not* need to provide exact examples for `x` and `y` that demonstrate the issue.

Solution:

(a) 0 0111 1111 0100 0000 0000 0000 0000 000

(b) i. All 0s

ii. $1 \cdot 2^{-127}$

(c) (In either order)

- *Overflow:* If `x` and `y` are certain (very large) values, then `x+2*y` might produce the special value "infinity" (or negative infinity is possible too) and then subtracting `y` still produces infinity while `x+y` is still small enough not to overflow.
- *Rounding error:* If `x` and `y` are the right number of orders of magnitude apart, we might due to rounding get that `x+y` is still `x` while `x+2*y-y` is slightly more than `x`.

Name:_____

4. (**9** points) Computer-Architecture Design

   (a) In roughly one English sentence, give a reason that it is better to have *fewer* registers in an instruction-set architecture.

   (b) In roughly one English sentence, give a reason that it is better to have *many* registers in an instruction-set architecture.

   (c) Yes or no: If we decided to change the x86-64 calling convention to make `%rbx` caller-saved, would the implementation of the CPU need to change?

   Solution:

   (a) We can implement the CPU with faster access to the registers, and we can design instruction encodings with fewer bits for identifying a register. (One reason is enough for full credit.)

   We also gave partial credit for saying there are fewer registers to save across a function call. This really is not a correct answer because unused registers do not take any effort to save, but the intuition on an open-ended question is good. And there is a related issue when different programs/threads take turns executing.

   (b) It is easier for humans or the compiler to write code without having to use the slower and harder-to-use memory on the stack for temporary variables.

   (c) No (it's just a convention)

Name:_____

5. (**18** points) x86-64 Programming

In this problem, you will show that x86-64 does not *need* many of its instructions (they are provided for convenience and performance).

Hint: `%rsp` is the stack pointer and `%rip` is the program counter.

(a) Suppose there were no `call` instruction. Show how you could replace `call foo` with two assembly instructions and one new label.

(b) Suppose there were no `ret` instruction. Show how you could replace `ret` with one other assembly instruction. (Significant partial credit for using two instructions instead.)

(c) Suppose there were no `push` (i.e., `pushq`) instruction. Show how you could replace `pushq %rax` with two assembly instructions.

(d) Suppose there were no `pop` (i.e., `popq`) instruction. Show how you could replace `popq %rax` with two assembly instructions.

(e) In roughly 1 English sentence, explain why we cannot replace `pushq (%rax)` or `popq (%rax)` with two assembly instructions.

Solution:

(a)
```
        pushq L
        jmp foo
    L:
```

(b)
```
        popq %rip
```

or in two instructions

```
        addq $8,%rsp
        jmp -8(%rsp)
```

(c)
```
        subq $8,%rsp
        movq %rax,(%rsp)
```

(d)
```
        movq (%rsp),%rax
        addq $8,%rsp
```

(e) Because the `movq` instruction can have at most one memory operand and `(%rsp)` is a memory operand, so we need two instructions to move the data and one to update the value of `%rsp`.

Name:_____

6. (**11** points) Stack Layout

Suppose before the assembly below is executed, the value of `%rsp` is `0xFFFF8888`. (The address of each instruction precedes it.)

```
0x00002f:   pushq $7
0x000031:   pushq $5
0x000033:   addq $2, 8(%rsp)
0x000039:   callq someOtherFunction
0x00003e:   ...
```

Immediately after the `callq` instruction executes:

(a) What is the value of `%rsp` in base-16 (hexadecimal)?

(b) Show the contents of the stack for the range from your answer to part (a) up to (but not including) `0xFFFF8888`. You can show each 8 bytes together as a single number on one "line" — for each such "line" show the address and the contents. For both, use base-16 (hexadecimal).

Solution:

`0xFFFF8888 - 0x18`

(a) ~~`0xFFFF8888-0x16`~~ (i.e., 24 bytes) = 0xFFFF8870

(b)
```
0xFFFF8880: 0x9 # (7+2)

0xFFFF8878: 0x5

0xFFFF8870: 0x3e # (the return address)
```

Name:_____

7. (**17** points) Control Flow

Consider this C switch-statement and the *incorrect* assembly, perhaps from a really buggy compiler, for it:

```
int f(int y) {
  switch(y) {
  case 1:
          y=17;
          break;
  case 9:
  case 10:
  case 3:
          y=14;
  case 4:
          y++;
          break;
  case 6:
          y=9;
          break;
  default:
          return -5;
  }
  return y;
}
```

```
          .section        .rodata
.L4:
          .quad   .L8
          .quad   .L3
          .quad   .L8
          .quad   .L5
          .quad   .L6
          .quad   .L8
          .quad   .L7
          .quad   .L8
          .quad   .L8
          .quad   .L8
          .quad   .L8
          .text
f:
          cmpl    $11, %edi
          ja      .L8
          movl    %edi, %eax
          jmp     *.L4(,%rax,8) # hint: this line is correct
.L3:
          movl    $17, %eax
          ret
.L5:
          movl    $14, %edi
.L6:
          leal    1(%rdi), %eax
          ret
.L7:
          movl    $9, %eax
.L8:
          movl    $-5, %eax
          ret
```

(a) The assembly code for `f` returns the same answer as the C code for `f` for some values of `y` but not others. For which of the following values is the assembly code ***incorrect***? No explanations required. (More than one is *incorrect*.)

15 11 9 6 3 1 -2

(b) For one of your answers to part (a), a segmentation fault is likely. Which one? No explanation required.

Solution:

(a) Incorrect for:

- 11 (explanation: The `cmpl` has an off-by-one error – it should use `$10`, not `$11`)
- 9 (explanation: The jump table has the wrong entry – it should jump to .L5)

- 6 (explanation: We should have a `ret` before .L8 rather than "falling through")

(b) 11 (we read off the end of the jump table and use "whatever bits are there" as the jump target)

REFERENCES

Powers of 2:

$2^0 = 1$	
$2^1 = 2$	$2^{-1} = .5$
$2^2 = 4$	$2^{-2} = .25$
$2^3 = 8$	$2^{-3} = .125$
$2^4 = 16$	$2^{-4} = .0625$
$2^5 = 32$	$2^{-5} = .03125$
$2^6 = 64$	$2^{-6} = .015625$
$2^7 = 128$	$2^{-7} = .0078125$
$2^8 = 256$	$2^{-8} = .00390625$
$2^9 = 512$	$2^{-9} = .001953125$
$2^{10} = 1024$	$2^{-10} = .0009765625$

Assembly Code Instructions:

`push`	push a value onto the stack (including subtracting from the stack pointer)
`pop`	pop a value from the stack (including adding to the stack pointer)
`call`	jump to a procedure after pushing a return address onto the stack
`ret`	pop return address from the stack and jump there
`mov`	move a value
`lea`	compute effective address and store in a dst
`add`	add src (1^{st} operand) to dst (2^{nd}) with result stored in dst (2^{nd})
`sub`	subtract src (1^{st} operand) from dst (2^{nd}) with result stored in dst (2^{nd})
`and`	bit-wise AND of src and dst with result stored in dst
`or`	bit-wise OR of src and dst with result stored in dst
`sar`	shift data in the dst to the right (arithmetic shift) by the number of bits specified in 1^{st} operand
`sal`	shift data in the dst to the left (arithmetic shift) by the number of bits specified in 1^{st} operand
`jmp`	jump to address
`jne`	conditional jump to address if zero flag is not set
`jg`	conditional jump to address if (strictly) greater than (signed comparison)
`jle`	conditional jump to address if less than or equal (signed comparison)
`ja`	conditional jump to address if (strictly) greater than (unsigned comparison)
`cmp`	subtract src (1^{st} operand) from dst (2^{nd}) and set flags
`test`	bit-wise AND src and dst and set flags

Register map for x86-64:

Note: all registers are caller-saved except those explicitly marked as callee-saved, namely, `rbx`, `rbp`, `r12`, `r13`, `r14`, and `r15`. `rsp` is a special register.

`%rax`	Return value
`%rbx`	Callee saved
`%rcx`	Argument #4
`%rdx`	Argument #3
`%rsi`	Argument #2
`%rdi`	Argument #1
`%rsp`	Stack pointer
`%rbp`	Callee saved

`%r8`	Argument #5
`%r9`	Argument #6
`%r10`	Caller saved
`%r11`	Caller Saved
`%r12`	Callee saved
`%r13`	Callee saved
`%r14`	Callee saved
`%r15`	Callee saved