CSE351 Spring 2012 – Midterm Exam (30 April 2012)

Please read through the entire examination first! We designed this exam so that it can be completed in 50 minutes and, hopefully, this estimate will prove to be reasonable.

There are 3 problems for a total of 100 points. The point value of each problem is indicated in the table below. Write your answer neatly in the spaces provided. If you need more space (you shouldn't), you can write on the back of the sheet where the question is posed, but please make sure that you indicate clearly the problem to which the comments apply. Do NOT use any other paper to hand in your answers. If you have difficulty with part of a problem, move on to the next one. They are independent of each other.

The exam is CLOSED book and CLOSED notes. Please do not ask or provide anything to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers.

Name: ____________________

ID#: ____________________

Problem	**Max Score**	**Score**
1	30	
2	35	
3	35	
TOTAL	**100**	

1. Number Representation (30 points)

(15 pts) Part A: Integers

We are converting from an old 16-bit machine to a new 64-bit architecture. Here is a C function that given 64 bits representing four 16-bit signed integers (`old4ints`), extracts the integer requested (specified by `int_number` ranging from 0 for most significant – leftmost, to 3 for least significant - rightmost) and converts it to a 64-bit signed integer (the return value). **Remember**: shifts are logical for unsigned integers and arithmetic for signed integers.

```
int64_t extract (uint64_t old4ints, int64_t int_number) {
     int64_t newint;
     newint = old4ints >> ( int_number << 4 );
     return newint & 0xFFFF;
}
```

(5 pts) There is something terribly wrong with this function. Describe the error(s) in a couple of sentences.

(10 pts) Write a correct version of the function.

```
int64_t extract (uint64_t old4ints, int64_t int_number) {
     int64_t newint;
```

(15 pts) Part B: Floating point numbers

A new pizzeria has opened on the Ave. It is mysteriously called “Pizza 0x40490FDB”. Given that you have just completed CSE351, you have a hunch what the mystery might be. Consider the string of hex digits as a 32-bit IEEE floating point number (8-bit exponent and 23-bit fraction). Fill in the hexadecimal digits in the bytes below and then translate them to individual bits.

8 hex digits in 4 bytes: **4 0** **4 9** **0 F** **D B**

32 bits: _

(2 pts) Is this number positive or negative (circle one)? **Positive** **Negative**

(4 pts) What is the exponent? _____
(exponents are biased in this representation so make sure to make this adjustment)

(4 pts) What is the significand? _____
(only use the first 7 bits of the fraction, ignore the lower-order 16 bits)

(4 pts) What is the decimal number represented? _____
(only show two decimal digits after the decimal point)

(1 pt) What is the pizzeria’s mystery name? ____________________________

2. Writing Assembly Code (35 points)

Below is a short C function that, given a pointer to a 4-character array consisting of only ASCII numbers (0 through 9 are represented in ASCII as 0x30 through 0x39), converts them to an integer representing a year A.D. (between 0 and 9999). It assumes the year is coded in big-endian order.

```
int charCodedYear2Integer (char *codedyr) {

      int intyr = 0;

// get the first char and mask everything but the digit
// then multiply by 10 and add the second digit (after masking)
// then multiply by 10 and add the third digit (after masking)
// then multiply by 10 and add the fourth digit (after masking)

      intyr =              (int) *codedyr       & 0x0F;
      intyr = 10*intyr + ( (int) *(codedyr + 1) & 0x0F );
      intyr = 10*intyr + ( (int) *(codedyr + 2) & 0x0F );
      intyr = 10*intyr + ( (int) *(codedyr + 3) & 0x0F );

      return intyr;
}
```

Complete the IA32 assembly code corresponding to this C function using <u>only</u>:
`add`, `and`, `lea`, and `mov` instructions or any of their variants.

Recall that `%eax`, `%ecx`, and `%edx` are caller-save registers.
The variable `intyr` corresponds to `%eax` as it is also the return value.
Use the following page to write your code.
Make sure to comment each line.

```
<charCodedYear2Integer>:
     pushl   %ebp                    Save old %ebp
     movl    %esp,%ebp               Initialize new %ebp
     movl    8(%ebp),%ecx            Get addr of first char
     xorl    %eax,%eax               Clear %eax to use for intyr


     <your assembly code (on next page) goes here>


     leave                           Restore old %ebp
     ret                             Return
```

Write your assembly code here (should be no somewhere in the range of 15 to 20 lines):

Hints: Start by moving the first character into a register. Multiply using additions (`lea` and/or `add` instructions) instead of multiply instructions.

3. Stack and Procedures (35 points)

(18 pts) Part A: Return values

Consider a return value from a procedure that needs to be a struct rather than a simple value. A struct can't be placed in a register (such as %rax that is used to transfer return values). For each of the situations described below, provide at most a couple of sentences to explain your approach.

(a) Consider the case when the struct is allocated statically in memory at compile time. The callee wants to return the struct which contains an arbitrary number of fields and doesn't fit in %rax. How could it return the struct?

(b) Consider the case when the struct is allocated on the stack by the **caller**. Are there any different issues in that case?

(c) Consider the case when the struct is created and allocated on the stack by the **callee**. Are there any different issues in that case?

(15 pts) Part B: Stack frames

We have a three procedure program with functions `main`, `proc_a`, and `proc_b` compiled for x86-64. `main` calls `proc_a` or `proc_b` based on the values of some inputs and the two procedures can call each other. `proc_a` has 7 arguments, while `proc_b` has only 2 arguments. `proc_a` also has a register it must save on the stack (a callee-save register) before it calls any other procedures and restore before it itself returns and it also allocates one 64-bit int on the stack. Thus four types of values can end up on the stack: return address (RET ADDR), argument (ARG), allocated value (ALLOCATED), and callee-saved register (CALLEE).

Here are some tiny assembly code snippets:

```
0000000000405060 <main>:
        …
        405068:   call <proc_a>
        40506d:   …
        …

0000000000405132 <proc_a>:
        …
        405155:   call <proc_b>
        40515a:   …
        …

0000000000406354 <proc_b>:
        …
        40637c:   call <proc_a>
        406381:   …
        …
```

(cont'd on next page)

Given the following stack contents, complete the Type column below by entering one of the four types of values the entry represents (RET ADDR, ARG, ALLOCATED, or CALLEE). Also complete the third column specifying which procedure placed the ALLOCATED value or CALLEE-saved register on the stack, or if an ARG, which procedure placed it on the stack for which other procedure, or if a RET ADDR, the name of the procedure that covers that address.

VALUES IN STACK MEMORY	TYPE	PROCEDURE USING VALUE OR CONTAINING ADDRESS	
…		…	
0000000000000001	ARG	*main for proc_a*	
000000000040506d			
0000000000000002			
0000000000000003			
000000000040515a			
0000000000000004			
0000000000406381			
0000000000406380			
0000000000000006			← %rsp

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:

pushl	push a 32-bit value onto the stack
leave	restore ebp from the stack
ret	pop return address from stack and jump there
movl	move 4 bytes between immediate values, registers and memory
movzbl	move 1 byte into the low-order byte of a long word, filling the other 3 bytes with 0s.
movsbl	move 1 byte into the low-order byte of a long word, filling the other 3 bytes by sign-extending the low-order byte that was moved
addl	add 1st operand to 2nd with result stored in 2nd
subl	subtract 1st operand from 2nd with result stored in 2nd
andl	logical bitwise AND of 1st operand with 2nd with result stored in 2nd
xorl	logical bitwise XOR of 1st operand with 2nd with result stored in 2nd
jmp	jump to address
je	conditional jump to address if zero flag set
jne	conditional jump to address if zero flag is not set
cmpl	subtract 1st operand from 2nd and set flags
testl	logical bitwise AND of 1st and 2nd operands to set flags