Name	
Student ID	

UW CSE 351, Summer 2013 Midterm Exam

Instructions:

- Make sure that your exam is not missing any of the 10 pages, then write your full name and UW student ID on the front.
- Read over the entire exam before starting to work on the problems! The last page is a reference page that you may tear off to use during the exam; it does not have to be turned in.
- Feel free to use the backs of pages for working on problems, but *write your answers in the space provided below each problem.* If you make a mess, clearly indicate your final answer. Be sure to answer all parts of all questions.
- Do not spend too much time on a problem if there are other easy problems that you haven't solved yet.
- **No books, notes, or electronic devices may be used during the exam.** You may not communicate with other students during the exam, but please ask the instructor if you need clarification for some problem.
- If you read this far, smile and have fun!

Section	**Awarded**	**Possible**
Section 1		**5**
Section 2		**25**
Section 3		**40**
Section 4		**30**
Total		**100**

1. Instruction Set Architectures (5 points)

(a) The original x86 processor, the Intel 8086, had a *word size* of 16 bits. What does *word size* mean and what does it determine in a computer system? (one or two sentences)

(b) Circle all of the features below that are part of the *instruction set architecture* of a machine:

i. The number of registers.
ii. The number of machine cycles to execute a single instruction.
iii. The effect that a certain instruction has on memory and registers when executed.
iv. The condition codes and what causes them to be set.
v. The available memory addressing modes.
vi. The machine word size.

2. Numbers and Bits (25 points)

(a) (3 points) Is there anything wrong with the code below? If so, name one thing. Assume that `n` is the number of elements in the `values` array. (Write one or two sentences.)

```
double product_except(int n, float values[], float skip) {
    float prod = 1.0;
    int i;
    for (i = 0; i < n; i++) {
        if (values[i] != skip) {
            prod = prod * values[i];
        }
    }
    return (double)prod;
}
```

(b) (8 points) Consider a machine with 6-bit integers. (While we generally use powers of two today, 36-bit words with 6-bit characters were commonplace several decades ago.)

i. What are the binary *and* decimal representations of the sum of these *two's complement signed* integers represented in binary? (Please write the sum both in binary and in decimal.)

100101 + 011101 =

binary:

decimal:

ii. What are the binary *and* decimal value of the sum of the 6-bit *two's complement* representations of these signed integers (shown here in decimal)? (Please write the sum both in binary and in decimal.)

(-24) + (-12) =

binary:

decimal:

iii. *Either* your answer for (i) *or* your answer for (ii) should be different than the answer for arithmetic on the equivalent mathematics integers. Which one? What is the name of the condition that causes it to be different?

(c) (4 points) What are two disadvantages to *sign and magnitude* integer representation as compared to *two's complement*? (one sentence or phrase each)

(d) (10 points) Design a space-efficient encoding of the latitude, longitude, and altitude of a satellite's position relative to Earth using only a single 32-bit C int to store *all* of this information at once. It must be easy to retrieve the individual dimensions from the encoded position. Signed latitudes range from -90 to +90 in increments of 1 and signed longitudes range from -180 to +179 in increments of 1. The satellite altitude (in your favorite units) ranges from zero to some maximum in increments of 1.

i. Allocate the 32 available bits below by bracketing and labeling which bits belong to which dimensions. *Put the latitude in the highest order bits, followed by the longitude, followed by the altitude in the lowest order bits.* Be sure to maximize the altitude you can represent.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
```

ii. What is the maximum altitude your encoding can represent? You may write an expression involving exponents and/or arithmetic.

iii. Implement a function to retrieve the longitude as a signed 32-bit `int` given an `int` encoded using your answer to (i). Only bitwise operators, shifts, and addition/subtraction are allowed, for efficiency. You may use local variables, but they are not necessary.

```
int get_longitude(int position) {
```

3. Reverse Engineering Stacks and Procedures (40 points)

There are questions that use the code below on the following pages. You may tear this page out carefully if you want to avoid flipping back for reference. You must turn this page in, but it does it not need to be attached to the rest of your exam.

Running short on sleep after a research paper deadline, your instructor accidentally removed several important files for a C program! Fortunately, the compiled machine code survived, but he needs your help reverse-engineering it. He remembers that the code uses a linked list of integers, with each node in the list represented by the following type:

```
typedef struct list_node {
  int value;
  struct list_node * next;
} list_node;
```

Each `list_node` stores an `int value` and a pointer (`next`) to the next `list_node` in the list. The last element in a list stores the address `0x0` in its `next` field, showing that there is no next element after the last element.

Additionally, we know that `mystery` looks like this:
`int mystery(list_node* current) { ... }`

The following machine code has been recovered and disassembled:

```
0x080483a0 <mystery>:
   0x080483a0 <+0>:     push   %ebp
   0x080483a1 <+1>:     mov    %esp,%ebp
   0x080483a3 <+3>:     sub    $0x8,%esp
   0x080483a6 <+6>:     mov    0x8(%ebp),%edx
   0x080483a9 <+9>:     mov    0x4(%edx),%eax
   0x080483ac <+12>:    test   %eax,%eax     <-- Pause for (c).
   0x080483ae <+14>:    jne    0x80483b4 <mystery+20>
   0x080483b0 <+16>:    mov    (%edx),%eax
   0x080483b2 <+18>:    jmp    0x80483bc <mystery+28>
   0x080483b4 <+20>:    mov    %eax,(%esp)
   0x080483b7 <+23>:    call   0x80483a0 <mystery>
   0x080483bc <+28>:    leave                <-- Stop for (d).
   0x080483bd <+29>:    ret

0x080483bb <intrigue>:
   ...                                       <-- Start.
   0x080483eb <+48>:    call   0x80483a0 <mystery>
   0x080483f0 <+53>:    mov    %eax,0x2c(%esp)
   ...
```

Section 3 continues on the next page.

3. [Continued] Reverse Engineering Stacks and Procedures (40 points)

We have provided you with the state of the heap and the stack *just before executing the call instruction in* `intrigue` *at 0x080483eb*. ***Questions (a) through (e) on the next page will guide you through filling the blanks in this diagram.*** Feel free to make notes in the margins, but make sure all labels requested in (a) through (e) are clearly visible. (Hints: you should not need more stack space than we have drawn and the heap contents will not change.)

Stack **initial `%ebp = 0xffffff58`** **initial `%esp = 0xffffff4c`**

Address	Value	Description of value
0xffffff4c	0x0b000000	
0xffffff48		
0xffffff44		
0xffffff40		
0xffffff3c		
0xffffff38		
0xffffff34		
0xffffff30		
0xffffff2c		

Heap

Address	Value	Description of these 8 bytes
0x0b00001c	0x0	
0x0b000018	351	
...	...	...
0x0b000004	0x0b000018	
0x0b000000	341	

Section 3 continues on the next page.

3. [Continued] Reverse Engineering Stacks and Procedures (40 points)

These questions refer to the diagrams and code on the previous two pages.

(a) (3 points) Based on `mystery`'s signature and the initial state of the stack, the heap, and the `%ebp` and `%esp` registers immediately before the call instruction at 0x080483eb is executed, label the *type* of what should be stored in the 8 bytes of memory at 0x0b000000 and the 8 bytes of memory at 0x0b000018.

(b) (10 points) Next, simulate the execution of the program *starting just before the call instruction at 0x080483eb. Stop just before you reach the test instruction at 0x080483ac.* Fill in the stack diagram as your simulation progresses, writing each value stored on the stack and a description of what this value represents. Write "unused" if the location is not used. *You may find it helpful to look ahead to part (e) of this problem to consider as you simulate the program.*

(c) (4 points) What is the combined purpose of the `test` instruction at 0x080483ac and the following `jne` instruction at 0x080483ae? (one sentence or phrase)

(d) (10 points) Continue simulating execution and filling in the stack diagram until just before you execute a `leave` instruction, and then label where the frame pointer ("$\leftarrow$%ebp") and stack pointer ("$\leftarrow$%esp") point.

(e) (10 points) We just heard that the original C code had the following structure! Using what you now know about `mystery`, fill in the blank lines with C code, using no variables except `current` in your expressions. Do not write register names.

```
int mystery(list_node* current) {

  if(________________________________) {

    return ___________________________________;
  }
  return ___________________________________;
}
```

(f) (3 points) Congratulations, you recovered the lost code! Explain what `mystery` does with its linked list argument. (one or two sentences)

4. Translating C Arrays to Memory and Assembly (30 points)

The following two C functions appear to perform the same operations: setting all elements in the diagonal of a matrix to their index within the diagonal. For example, `matrix[3][3]` gets set to 3. ***However, they are subtly different.***

```
void diagonalA(int n, int* matrix[]) {
  int i;
  for (i = 0; i < n; i++) {
    matrix[i][i] = i;
  }
}
void diagonalB(int n, int matrix[4][4]) {
  int i;
  for (i = 0; i < n; i++) {
    matrix[i][i] = i;
  }
}
```

For each statement below, create a true statement by filling the blank with one of:

- "Only A" if the statement is only true of `diagonalA`;
- "Only B" if the statement is only true of `diagonalB`;
- "Neither" if it is true of neither; or
- "Both" if it is true of both.

Additionally, where asked, explain how or why this is the case ***in a brief sentence or phrase***.

(a) (5 points) __________ uses multi-dimensional (2D) arrays, while __________ uses multi-level (2-level) arrays. What is the difference and how can you tell from the type?

(b) (2 points) __________ could lead to a segmentation fault. How/why?

(c) (5 points) __________ could generate memory accesses to this sequence of addresses if the base address of the matrix array is `0x4000` and `n` = 4:
`0x4000, 0x4014, 0x4028, 0x403c`. How/why?

Section 4 continues on the next page.

(d) (5 points) __________ could generate memory accesses to this sequence of addresses if the base address of the `matrix` array is `0x4000` and `n` = 4:

`0x4000, 0x4800, 0x4004, 0x568c, 0x4008, 0x4600, 0x400c, 0x4800`

How/why?

(e) (3 points) __________ could lead to a buffer overflow in the current stack frame, corrupting the return address so that when `diagonalA/B` returns, it starts executing an attacker's code instead. How/why?

(f) (10 points) __________ could be correctly compiled to this assembly code (question below):

```
080483f4 <diagonalMystery>:
 80483f4: 55              push   %ebp
 80483f5: 89 e5           mov    %esp,%ebp
 80483f7: 53              push   %ebx
 80483f8: 8b 4d 08        mov    0x8(%ebp),%ecx
 80483fb: 8b 5d 0c        mov    0xc(%ebp),%ebx
 80483fe: 85 c9           test   %ecx,%ecx
 8048400: 7e 12           jle    8048414 <diagonalMystery+0x20>
 8048402: b8 00 00 00 00  mov    $0x0,%eax
 8048407: 8b 14 83        mov    (%ebx,%eax,4),%edx
 804840a: 89 04 82        mov    %eax,(%edx,%eax,4)
 804840d: 83 c0 01        add    $0x1,%eax
 8048410: 39 c8           cmp    %ecx,%eax
 8048412: 75 f3           jne    8048407 <diagonalMystery+0x13>
 8048414: 5b              pop    %ebx
 8048415: 5d              pop    %ebp
 8048416: c3              ret
```

What was your key observation?

Reference

1. **Hex Digits**

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

2. **Powers of Two**

2^0 = 0x0001 = 1	2^6 = 0x0040 = 64
2^1 = 0x0002 = 2	2^7 = 0x0080 = 128
2^2 = 0x0004 = 4	2^8 = 0x0100 = 256
2^3 = 0x0008 = 8	2^9 = 0x0200 = 512
2^4 = 0x0010 = 16	2^{10} = 0x0400 = 1024
2^5 = 0x0020 = 32	

3. **Assembly Code Instructions**

`pushl`	push a 4-byte value onto the stack
`pushq`	push a 8-byte value onto the stack
`popq`	pop a 8-byte value from the stack
`call`	push the address of the next instruction onto the stack and jump to target
`leave`	restore ebp from the stack
`ret`	pop return address from stack and jump there
`leal`	compute an address in first operand, put result in second
`movl`	move 4 bytes between values, registers and memory
`movq`	move 8 bytes between values, registers and memory
`movzbl`	move zero-extended value to long
`incl`	increment operand by 1
`decl`	decrement operand by 1
`addl`	(4 bytes) add first operand to second, put result in second
`addq`	(8 bytes) add first operand to second, put result in second
`subl`	subtract first operand from second, put result in second
`imull`	signed multiply of first operand and second, put result in second
`andl`	logical AND of first operand with second, put result in second
`sall`	arithmetic left shift of first operand, put result in second
`sarl`	arithmetic right shift of first operand
`cmpl`	subtract first operand from second; set flags
`testl`	logical AND of first operand with second; set flags (4 bytes)
`testb`	logical AND of first operand with second; set flags (1 byte)
`jmp`	jump to address
`jg`	conditional jump to address if last comparison was greater than
`je`	conditional jump to address if result of last comparison was zero
`jne`	conditional jump to address if result of last comparison was not zero
`jle`	conditional jump to address if result of last comparison was zero or negative