CSE 351 Midterm - Spring 2015

May 1, 2015

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 5 problems for a total of 100 points, and one 10 point extra credit problem. 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 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, and that you write your name on **all** pages. 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 (no summary sheets, no mobile phones, no laptops, and simple calculators only). 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.

Good luck and have fun!

Name: ________Solution Guide________

Student ID: ____________________________

Problem	Max Score	Score
1	10	
2	20	
3	30	
4	30	
5	10	
TOTAL	100	
EC	10	

1 Number Representation(10 points)

Let `x=0xE` and `y=0x7` be integers stored on a machine with a word size of **4bits**. Show your work with the following math operations. **The answers—including truncation—should match those given by our hypothetical machine with 4-bit registers.**

A. (2pt) What hex value is the result of adding these two numbers?

In hex: 0xE + 0x7 = 0x15 $\rightarrow$ 0x5
In binary converted back to hex: 0xE + 0x7 = 1110 + 0111 = 10101 $\rightarrow$ 0101 = 0x5
Half credit for not truncating to the appropriate value.

B. (2pt) Interpreting these numbers as unsigned ints, what is the decimal result of adding $x + y$?

In unsigned decimal: 0xE + 0x7 = 14 + 7 = 21 % 16 = 5
Half credit for not truncating to the appropriate value or incorrect conversion.
No credit for computing in signed decimal

C. (2pt) Interpreting x and y as two's complement integers, what is the decimal result of computing $x-y$?

In signed decimal: 0xE - 0x7 =¿ -2 - 7 = -9 $\rightarrow$ 7
Half credit for not truncating to the appropriate value, or incorrect conversion.
No credit for computing in unsigned decimal

D. (2pt) In one word, what is the phenomenon happening in 1B?

Overflow.

E. (2pt) Circle all statements below that are **TRUE** on a **32-bit architecture**:
Half point each.

- It is possible to lose precision when converting from an int to a float. True
- It is possible to lose precision when converting from a float to an int. True
- It is possible to lose precision when converting from an int into a double. False
- It is possible to lose precision when converting from a double into an int. True

2 IA32 ASM to C (20 points)

A function 'mystery' has the following overall structure:

```
int mystery (int x, int y){

    int result;

    for (________________;________________; result++){

        ________________________;

        ________________________;

    }
    ________________________;

    return result;
}
```

The GCC C compiler generates the following x86 (IA32) assembly code (x is at %ebp+8, y at %ebp+12)

```
            pushl   %ebp
            movl    %esp, %ebp
            movl    8(%ebp), %ecx
            movl    12(%ebp), %edx
            movl    $0, %eax
            test    %ecx, %ecx
            jz      .L3
     .L6
            addl    %ecx, %edx
            subl    $1, %ecx
            addl    $1, %eax
            cmpl    $0, %ecx
            jg      .L6
     .L3
            addl    %edx, %eax
            popl    %ebp
            ret
```

Fill in the blanks in mystery based on the assembly code above. You may only use the symbolic variables x, y, and result in your expressions. Do not use register names.

Answers to blanks, in order:

```
result = 0;
x > 0;        //  Also accept x != 0
y += x;
x--;
result += y
```

3 C to ASM (30 points)

Write **x86-64** assembly instructions (see the reference sheet for the list of instructions that you can use on this exam) that might be generated by the following function foo. It may be a good idea to consult the register chart provided on the reference sheet.

```
int foo (int a, int b){
    int c, d;
    c = a / 16;
    d = b * 64;
    if (c > d)
        return a;
    else
        return b;
}
```

Place the assembly code for function `foo` here (you should need fewer than 15 instructions), and a comment for each line of your code. **You may only use the instructions that are on reference sheet!**

```
.FOO
    movl %edi, %e10      # ( may use another register, but must be 32 bit)
    sar  $4, $e10        # ( no credit for anything other than shift)
    movl %esi, %e11      # ( may use another register, but must be 32 bit)
    shl  6, %e11         # ( no credit for anything other than shift)
    cmpl %e11, %e10      # ( accept opposite order, if next line matches)
    jle  $.L1            # ( two for instruction, 2 for useful label OR arrow OR address)
    movl $edi, %eax      #
    jmp  $.END           # ( also accept ret here instead of jump to end)
.L1
    movl %esi, %eax      #
.END
    ret                  # (must be present: all control flow must go through a ret)
```

4 Stack Discipline (30 points)

Given the C function

```
int proc ( void ){
    int a[3];
    scanf("%x %x %x", &a[1], &a[0], &a[2]);
    return a[2];
}
```

GCC generates the following code:

```
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        pushl   %esi
        subl    $0x20, %esp
        leal    -20(%ebp), %eax
        movl    $0, %esi
        leal    (%eax,%esi, 4), %ebx
        movl    %ebx, 8(%esp)
        addl    $1, %esi
        leal    (%eax,%esi, 4), %ebx
        movl    %ebx, 4(%esp)
        addl    $1, %esi
        leal    (%eax,%esi, 4), %ebx
        movl    %ebx, 12(%esp)
        movl    $.LC0, (%esp)     #Pointer to string "%x %x %x"
        call    scanf             <== here
        movl    (%ebx), %eax
        addl    $0x20, %esp
        popl    %esi
        popl    %ebx
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Draw a picture depicting the stack frame of `proc` immediately before the call to `scanf` (labeled ”here” above). Draw labeled arrows indicating where the stack and frame pointers are. If needed, you can assume that `%esp = 0x800040` and `%ebp = 0x800060` just before `proc` is called. The next page is left blank to give you more room.

Note: though not necessary to solve the problem, `scanf` is much like the `sscanf` you saw in Lab 2 (matching an input string to some format), except it reads the input string from `stdin` (the terminal).

Address	Value	Comment	
0x800040	??	where %esp used to point	
0x80003C	??	ret addr	
0x800038	0x800060	old ebp	<-- ebp
0x800034	??	saved ebx	
0x800030	??	saved esi	
0x80002C	??	a[2]	
0x800028	??	a[1]	
0x800024	??	a[0]	
0x800020	--	wasted space	
0x80001C	0x80002C	&a[2]	
0x800018	0x800024	&a[0]	
0x800014	0x800028	&a[1]	
0x800010	??	$.LC0 (pointer to format string)	<-- esp

Grading Notes:
First two lines in table are optional. Need to have the other 11.
Comment column and pointer columns are required. Address and value are optional
If addresses are used, they must increment by the correct values
Any values provided &a[0],&a[1],&a[2],old ebp, must be correct

5 Structs (10 points)

Suppose you are given the following struct definition for an x86-64 architecture which is used to implement a linked list of all tweets in Katelin's SuperTwitter implementation.

```
typedef struct Super_Tweet{
    char super_tweeter[21];
    int num_retweets;
    int num_favorites;
    long id;
    tweet* next;
    int datetime_encoded;             //seconds since SuperTwitter was launched
} tweet
```

A. (1/2pt each) Given the above definition, fill in the following table:

Field Name	**Offset**	**Size of Field (bytes)**
super tweeter	0	21
(wasted space)	21	3
num retweets	24	4
num favorites	28	4
id	32	8
next	40	8
datetime encoded	48	4
(wasted space)	52	4

B. (1pt) What is the size of the struct? 56 bytes

C. (1/2pt) How much internal fragmentation does this struct have? 3 bytes.

D. (1/2pt)How much external fragmentation does this struct have? 4 bytes.

6 Arrays (10 points, extra credit)

In the space below, draw the memory layout on a 32-bit machine for:

```
char a[2][3] = {{'a', 'b', 'c'}, {'d','e','f'}}
```

Half point, each box, +1 for correct ordering

0x00	'a'	'b'	'c'	'd'
0x04	'e'	'f'		
0x08				
0x0C				
0x10				
0x14				
0x18				
0x1C				

```
char *b[2] = {"foo", "bar"};
```

Hint: you may place "foo" and "bar" somewhere in memory, to get an address.

Half point, each character box, 1 point each pointer. Solution assumes little endian, big endian also okay.

0x00				
0x04	'f'	'o'	'o'	'\0'
0x08				
0x0C				
0x10	'b'	'a'	'r'	'\0'
0x14				
0x18	04	00	00	00
0x1C	10	00	00	00

References

Powers of 2:

$2^0 = 1$	
$2^1 = 2$	$2^{-1} = 0.5$
$2^2 = 4$	$2^{-2} = 0.25$
$2^3 = 8$	$2^{-3} = 0.125$
$2^4 = 16$	$2^{-4} = 0.0625$
$2^5 = 32$	$2^{-5} = 0.03125$
$2^6 = 64$	$2^{-6} = 0.015625$
$2^7 = 128$	$2^{-7} = 0.0078125$
$2^8 = 256$	$2^{-8} = 0.00390625$
$2^9 = 512$	$2^{-9} = 0.001953125$
$2^{10} = 1024$	$2^{-10} = 0.0009765625$

Hex help:

```
0x00 = 0
0x0A = 10
0x0F = 15
0x20 = 32
0x28 = 40
0x2A = 42
0x2F = 47
```

Assembly Code Instructions:

`push`	push a value onto the stack and decrement the stack pointer
`pop`	pop a value from the stack and increment the stack pointer
`call`	jump to a procedure after first pushing a return address onto the stack
`ret`	pop return address from stack and jump there
`mov`	move a value between registers and memory
`lea`	compute effective address and store in a register
`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) by the number in 1^{st} operand
`shl`	shift data in the dst to the left by the number of bits specified in 1^{st} operand
`jmp`	jump to address
`jg`	conditional jump to address if not zero flag and not sign flag
`jle`	conditional jump to address if zero flag or sign flag
`jne`	conditional jump to address if zero flag is not set
`jns`	conditional jump to address if sign flag is not set
`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	`%r8`	Argument #5
`%rbx`	Callee Saved	`%r9`	Argument #6
`%rcx`	Argument #4	`%r10`	Caller Saved
`%rdx`	Argument #3	`%r11`	Caller Saved
`%rsi`	Argument #2	`%r12`	Callee Saved
`%rdi`	Argument #1	`%r13`	Callee Saved
`%rsp`	Stack Pointer	`%r14`	Callee Saved
`%rbp`	Callee Saved	`%r15`	Callee Saved