

CS230 System Programming
Midterm Exam
Thursday, October 20th, 2022

Name:

Student Number:

Problems	Points
1 (20 pts)	
2 (10 pts)	
3 (12 pts)	
4 (12 pts)	
5 (14 pts)	
6 (20 pts)	
7 (20 pts)	
8 (24 pts)	
Total (132 pts)	

PLEASE READ CAREFULLY BEFORE

- I. Fill your name and student numbers at every page.
- II. You will get automatically 0 credits if your handwriting is unreadable.
- III. Write your answers on this sheet

1. [20 pts: 2 pts per each] True or False Questions

1. At compile time, switch statements are translated into a series of if-else statements to optimize for performance. (T/F)
2. Floating point rounding may create overflow and the postnormalization step resolves it by dropping the overflowed MSB. (T/F)
3. %rip register always holds the address of the current instruction that the processor is running. (T/F)
4. A sign bit of "1" in the result of a 2's-complement addition indicates that negative overflow occurred. (T/F)
5. Caller stores the current address on the stack before it jumps since when the callee's computation is done, the control needs to come back to where the caller called. (T/F)
6. The main reason to use the round-to-even method is to avoid systematic bias when calculating with rounded numbers. (T/F)
7. The range of negative numbers when using a seven-bit two's complement system is -1 to -63. (T/F)
8. In x64, the first six arguments to functions are passed via registers and the orders of these six arguments determine which registers to be assigned. (T/F)
9. x86 instructions for integer arithmetic operations do not distinguish signed and unsigned numbers. (T/F)
10. Conditional move instructions eliminate the necessity of control transfer (i.e., jump) by performing computations on both branches and picking up one afterward depending on the condition. (T/F)

2. [10 pts] Datalab

[4pts] Consider the following C code for calculating the average of two integers. Fill in the blank.
(" _ " represents operand, "o" represents operation)

```
/*
 * AverageofTwo - Return [average of two integers]
 * Floor function ( floor(x) = |x| ) takes as input a real number,
 * and gives out the greatest integer less than or equal to x.
 *
 * Examples: AverageofTwo(1, 2) = 1
 *           AverageofTwo(30, 40) = 35
 *           AverageofTwo(-1, -2) = -2
 *           AverageofTwo(-30, -40) = -35
 *           AverageofTwo(-2147483648, -2147483648) = -2147483648
 *
 * Legal ops: ! ~ & ^ | + << >>
 * Constant no longer than 8-bits is allowed
 */
int AverageofTwo(int x, int y) {
    int answer = (x & y) + (( _ o _ ) o _ );
    return answer;
}
```

[6pts: 2pts each] Generally, two's complement of int x is “ $\sim x + 1$ ”. The following C code is finding two's complement without operator “ \sim ” and constants. Write down (a) ~ (c) in contexts.

```
/*
 * Tcomplement - find the two's complement of input.
 *
 * Legal ops: ! & ^ | + << >>
 * Operator priority : "!, &, +" > "<<, >>" > "&" > "|"
 * Constant is not allowed (e.g. "one = 1" is illegal)
 * No limitation on the number of operators except(b)
 */
int Tcomplement(int x) {
    int one, two, thirtyone, ret;
    one = (a);
    two = one << one;
    thirtyone = (b); // max ops is 8
    ret = (___(c)___) + one;
    return ret;
}
```

(a):

(b):

(c):

3. [12 pts: 4 pts per each] Bomblab

Solve the following problems with given C code and its assembly code. The assembly code shows phase_4, func4 of bomblab. C code only shows phase_4 function. You can ignore the blank line on phase_4+109 on assembly code.

C Code

```
void phase_4(char *input) {
    int user_val, user_sum, result, target_val, numScanned, simple_sum;

    numScanned = sscanf(input, "%d %d %d", &simple_sum, &user_val, &user_sum);
    if ((numScanned != 3) || user_val < 1 || user_val > 10 || simple_sum <= 0) {
        explode_bomb();
    }

    if(simple_sum != ____(a)__)
        explode_bomb();

    target_val = 1;
    result = func4(user_val, target_val);

    if (result != user_sum) {
        explode_bomb();
    }
}
```

Assembly Code

```
(gdb) disas phase_4
Dump of assembler code for function phase_4:
0x0000000000001449 <+0>: sub $0x28,%rsp
0x000000000000144d <+4>: mov %fs:0x28,%rax
0x0000000000001456 <+13>: mov %rax,0x18(%rsp)
0x000000000000145b <+18>: xor %eax,%eax
0x000000000000145d <+20>: lea 0xc(%rsp),%rcx
0x0000000000001462 <+25>: lea 0x14(%rsp),%rdx
0x0000000000001467 <+30>: lea 0x10(%rsp),%r8
0x000000000000146c <+35>: lea 0x177f(%rip),%rsi # 0x2bf2
```

```
0x0000000000000001473 <+42>: callq 0xf50 <__isoc99_sscanf@plt>
0x0000000000000001478 <+47>: cmp $0x3,%eax
0x000000000000000147b <+50>: jne 0x1489 <phase_4+64>
0x000000000000000147d <+52>: mov 0xc(%rsp),%eax
0x0000000000000001481 <+56>: sub $0x1,%eax
0x0000000000000001484 <+59>: cmp $0x9,%eax
0x0000000000000001487 <+62>: jbe 0x14c6 <phase_4+125>
0x0000000000000001489 <+64>: callq 0x1980 <explode_bomb>
0x000000000000000148e <+69>: cmpl $0x347,0x14(%rsp)
0x0000000000000001496 <+77>: jne 0x14cf <phase_4+134>
0x0000000000000001498 <+79>: mov $0x1,%esi
0x000000000000000149d <+84>: mov 0xc(%rsp),%edi
0x00000000000000014a1 <+88>: callq 0x1402 <func4>
0x00000000000000014a6 <+93>: cmp %eax,0x10(%rsp)
0x00000000000000014aa <+97>: je 0x14b1 <phase_4+104>
0x00000000000000014ac <+99>: callq 0x1980 <explode_bomb>
0x00000000000000014b1 <+104>: mov 0x18(%rsp),%rax
0x00000000000000014b6 <+109>: _____, _____ # ignore this line
0x00000000000000014bf <+118>: jne 0x14d6 <phase_4+141>
0x00000000000000014c1 <+120>: add $0x28,%rsp
0x00000000000000014c5 <+124>: retq
0x00000000000000014c6 <+125>: cmpl $0x0,0x14(%rsp)
0x00000000000000014cb <+130>: jg 0x148e <phase_4+69>
0x00000000000000014cd <+132>: jmp 0x1489 <phase_4+64>
0x00000000000000014cf <+134>: callq 0x1980 <explode_bomb>
0x00000000000000014d4 <+139>: jmp 0x1498 <phase_4+79>
0x00000000000000014d6 <+141>: callq 0xeb0 <__stack_chk_fail@plt>
End of assembler dump.
```

(gdb) disas func4

Dump of assembler code for function func4:

```
0x0000000000000001402 <+0>: mov $0x1,%eax
0x0000000000000001407 <+5>: mov %edi,%edx
0x0000000000000001409 <+7>: or %esi,%edx
0x000000000000000140b <+9>: je 0x1447 <func4+69>
0x000000000000000140d <+11>: sub $0x8,%rsp
0x0000000000000001411 <+15>: cmp %esi,%edi
0x0000000000000001413 <+17>: jg 0x142b <func4+41>
0x0000000000000001415 <+19>: cmp %esi,%edi
```

```
0x000000000000001417 <+21>: jl 0x1439 <func4+55>
0x000000000000001419 <+23>: sub $0x1,%esi
0x00000000000000141c <+26>: sub $0x1,%edi
0x00000000000000141f <+29>: callq 0x1402 <func4>
0x000000000000001424 <+34>: add %eax,%eax
0x000000000000001426 <+36>: add $0x8,%rsp
0x00000000000000142a <+40>: retq
0x00000000000000142b <+41>: sub $0x1,%edi
0x00000000000000142e <+44>: callq 0x1402 <func4>
0x000000000000001433 <+49>: lea 0x1(%rax,%rax,1),%eax
0x000000000000001437 <+53>: jmp 0x1426 <func4+36>
0x000000000000001439 <+55>: sub $0x1,%esi
0x00000000000000143c <+58>: callq 0x1402 <func4>
0x000000000000001441 <+63>: lea -0x1(%rax,%rax,1),%eax
0x000000000000001445 <+67>: jmp 0x1426 <func4+36>
0x000000000000001447 <+69>: repz retq
```

End of assembler dump.

- (1) Find the correct simple_sum (a) value in C code to avoid explosion of bomb. Write your answer in hexadecimal format.

- (2) Calculate user_sum value in C code to avoid explosion of bomb when user_val is 1. The target_val is 1 as shown on the C code.

- (3) Count how many times the func4 is invoked when both user_val and target_val are 2 on C code. Assume the simple_sum is correct to avoid the explosion.

4. [12 pts: 2 pts per each] Consider the following C code and fill out the blanks in the expected printf outputs.

```
union foo {  
    int x;  
    float f;  
    unsigned char c[4];  
};  
void endianness_little2big(union foo *bar) {  
    unsigned char g;  
    g = bar->c[0];  
    bar->c[0] = bar->c[3];  
    bar->c[3] = g;  
    g = bar->c[1];  
    bar->c[1] = bar->c[2];  
    bar->c[2] = g;  
}  
void endianness_question() {  
    union foo bar;  
  
    bar.x = 2268;  
    printf("Integer decimal value: %d\n",bar.x);  
    printf("Integer hex value before endian swap: 0x%08x\n",bar.x);  
    endianness_little2big(&bar);  
    printf("Integer hex value after endian swap: 0x%08x\n", bar.x);  
  
    bar.x = 0x419c0000;  
    printf("Float decimal value: %f\n",bar.f);  
    printf("Float hex value before endian swap: 0x%08x\n",bar.x);  
    endianness_little2big(&bar);  
    printf("Float hex value after endian swap: 0x%08x\n", bar.x);  
}  
int main() { endianness_question(); return 0; }
```

Integer decimal value: (1) _____

Integer hex value before endian swap: (2) _____

Integer hex value after endian swap: (3) _____

Float decimal value: (4) _____

Float hex value before endian swap: (5) _____

Float hex value after endian swap: (6) _____

5. [14 pts: 2 pts per each] Consider the following C code and corresponding assembly. Fill in the missing instructions (one instruction per a blank line).

C Code	Assembly Code
<pre>int func(int *valptr, int n) { int val = 10; *valptr += 1; if (n < 2) return 1; else return n + func(&val, n - 1); }</pre>	<pre>func: (1) _____ subq \$16, %rsp movq %fs:40, %rax movq %rax, 8(%rsp) xorl %eax, %eax movl \$10, 4(%rsp) addq \$1, (%rdi) cmpl \$1, %esi jg .L6 (2) _____ .L1: (3) _____ (4) _____ jne .L7 addq \$16, %rsp (5) _____ ret .L6: movl %esi, %ebx (6) _____ (7) _____ call func addq %rbx, %rax jmp .L1 .L7: call __stack_chk_fail@PLT</pre>

6. [20 pts] Consider an 8-bit floating point representation with a 3-bit significand, a 4-bit exponent, a sign bit, and a bias value of 7. Assume that the implementation supports the IEEE standard (both normalized and denormalized values, and uses “nearest even” rounding). Fill in the empty boxes in the following table. (Note that if there are multiple possible answers, you can just fill one possible one)

Credits	Description	Value	sign	exponent	significand
2 pts	Zero	0.0			
2 pts	Largest denormalized	-			
2 pts	Smallest positive normalized	-			
2 pts	Largest finite number	-			
2 pts	NaN	-			
2 pts	Negative infinity	-			
4 pts	-5	-5.0			
4 pts	-	$2^{-4} - 2^{-6}$			

7. [20 pts: 2 pts per each] Consider the following x86-64 assembly code for a procedure poli():

Based on the assembly code, fill in the blanks below in its corresponding C source code. (Note: you may only use symbolic variables a, b, and result, from the source code in your expressions below — do not use register names.)

Assembly Code	C Code
<pre> poli: cmpq \$7, %rdi ja .L2 jmp *.L4(%rdi,8) .L4: .quad .L3 .quad .L2 .quad .L5 .quad .L2 .quad .L6 .quad .L7 .quad .L2 .quad .L5 .text .L6: leaq (%rdi,%rdi,2), %rsi jmp .L2 .L7: leaq (%rsi,%rsi,4), %rdx .L3: salq \$6, %rdx movq %rdx, %rsi .L2: movq %rsi, (%rcx) ret .L5: addq %rdx, %rsi salq \$5, %rsi jmp .L2 </pre>	<pre> void poli(long a, long b, long c, long *dest) { long val; switch (a) { case (1) _____: c = (2) _____; case (3) _____: val = (4) _____; break; case (5) _____: case (6) _____: val = (7) _____; break; case (8) _____: val = (9) _____; break; default: val = (10) _____; } *dest = val; } </pre>

8. [24 pts: 2 pts per each] Consider the following C code and corresponding assembly. Fill the assembly code in the blanks, based on the corresponding c statements.

C Code

```
long** matmul(int MAT_SIZE, long** mat1, long** mat2, long** res_mat){  
    int i, j, k;  
    for (i = 0 ; i < MAT_SIZE; i++) {  
        for (j = 0 ; j < MAT_SIZE; j++) {  
            for (k = 0 ; k < MAT_SIZE; k++){  
                res_mat[i][j] += mat1[i][k] * mat2[k][j];  
            }  
        }  
    }  
    return res_mat;  
}
```

Assembly Code

```
0x00000000000000078a <+0>: push %r12
0x00000000000000078c <+2>: push %rbp
0x00000000000000078d <+3>: push %rbx
0x00000000000000078e <+4>: mov %rcx,%rax
0x000000000000000791 <+7>: mov $0x0,%r12
0x000000000000000797 <+13>: jmp (1)
0x000000000000000799 <+15>: mov %r12,%r9
0x00000000000000079c <+18>: mov %rbp,%rbx
0x00000000000000079f <+21>: lea 0x0(%rbx,8),%r8
0x0000000000000007a7 <+29>: add (%rax,%r9,8), (2)
0x0000000000000007ab <+33>: mov (%rsi,%r9,8),%rcx
0x0000000000000007af <+37>: mov %r10,%r11
0x0000000000000007b2 <+40>: mov (%rdx, (3),8),%r9
0x0000000000000007b6 <+44>: mov (%r9, (4),8),%r9
0x0000000000000007ba <+48>: imul ((5),%r11,8),%r9
0x0000000000000007bf <+53>: add %r9, ((6))
0x0000000000000007c2 <+56>: add (7),%r10
0x0000000000000007c6 <+60>: cmp %rdi,%r10
0x0000000000000007c9 <+63>: (8) 0x799
0x0000000000000007cb <+65>: add (9),%rbp
0x0000000000000007ce <+68>: cmp %rdi,%rbp
0x0000000000000007d0 <+70>: (10) 0x7da
0x0000000000000007d2 <+72>: mov $0x0,%r10
0x0000000000000007d8 <+78>: jmp 0x7c6
0x0000000000000007da <+80>: add (11),%r12
0x0000000000000007de <+84>: cmp %rdi,%r12
0x0000000000000007e1 <+87>: (12) 0x7ea
0x0000000000000007e3 <+89>: mov $0x0,%rbp
0x0000000000000007e8 <+94>: jmp 0x7ce
0x0000000000000007ea <+96>: pop %rbx
0x0000000000000007eb <+97>: pop %rbp
0x0000000000000007ec <+98>: pop %r12
0x0000000000000007ee <+100>: ret
```

(1)	
(2)	
(3)	
(4)	
(5)	
(6)	
(7)	
(8)	
(9)	
(10)	
(11)	
(12)	

***** Description of jX instructions**

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~ (SF^OF) & ~ZF	Greater (Signed)
jge	~ (SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF) ZF	Less or Equal (Signed)
ja	~CF & ~ZF	Above (unsigned)
jb	CF	Below (unsigned)