CSE351 Spring 2014 – Midterm Exam (5 May 2014)

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 90 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 (no summary sheets, no calculators, no mobile phones, no laptops). 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: _____***Solution Key***__ ______

ID#: _________________________

Problem	Max Score	Score
1	15	
2	10	
3	20	
4	30	
5	15	
TOTAL	**100**	

1. Warm-up (15 points)

A. If we have six (6) bits in which to represent integers, what is largest unsigned number and what is largest 2s complement number we can represent (in decimal)?

Largest unsigned number: ________63_______

Largest 2s complement number: ________31_______

B. If %eax stores x and %ebx stores y, what do the following lines of assembly compute? Note that the result is in %eax.

```
    mov    %ebx, %ecx
    add    %eax, %ebx
    je     .L1
    sub    %eax, %ecx
    je     .L1
    xor    %eax, %eax
    jmp    .L2
L1:
    mov    $1, %eax
L2:
   …
```

|x| == |y| or a logical comparison of the absolute values of x and y. The first line merely copies y so that is can be reused. The second line compute x + y. If the result is 0 then we jump to L1 (this indicates x and y are negatives of each other) where eax is set to 1 (true). If not, then we compute y – x. Again, if the result is 0 then we jump to L1 (this indicates x and y have the same positive or same negative values) where %eax is set to 1 (true). If not, then we clear %eax (false) and finally jump around the statement that set %eax to 1.

2. Floating Point Representation (10 points)

Suppose we have 16-bit floating point numbers where 6 bits are assigned to the exponent and 9 bits to the fraction and 1 to the sign bit.

A. What is the bias for this float?

Bias = 2 ^ (6-1) – 1 = 31

B. Given the decimal number 3.625, calculate the fraction (frac) and exponent (exp) that would appear in the floating point representation. (Note: you may leave your answer in decimal for the exponent.)

3 in binary is 11. The decimal fraction can be represented as a sum of binary fractions.

0..625
- 0.50000 $1/2^1$
0.125
- 0.125000 $1/2^3$
0.0

Thus, the binary fraction is 0.101. Altogether the mantissa is 11.101. To normalize, we move the decimal place until there is only a 1 ahead of the decimal point (a value between 1 and 2), and then drop it as it is implicit in our floating point number representation. Thus,

frac = 1101

In the process of normalizing, the mantissa was divided by 2^1 (1 binary place), so the signed exponent (E) is 1. Thus, with

exp = bias + E = bias + 1 = 31 + 1 = 32 = 100000_2

The complete bit pattern for our number is 0 100000 110100000.

3. C and Assembly Code (20 points)

Given the C code for the function foo, determine which IA32 and x86-64 code snippet corresponds to a correct implementation of foo.

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

     int c = x << (y + 3);

     if (x != 0) {
          return c;
     } else {
          return 1;
     }

}
```

A. Which of the following IA-32 implementations is correct for foo()? Circle the correct one and give at least one reason why the other two are not correct.

```
i)    push   %ebp
      mov    %esp, %ebp
      mov    0xc(%ebp), %ecx
      add    $0x3, %ecx
      mov    0x8(%ebp), %eax
      shl    %eax, %ecx
      mov    %ecx, %eax
      cmp    $0x8(%ebp), $0
      jne    $0x808472 // two lines down to leave
      mov    $0x1, %eax
      leave
      ret
```

```
ii)   push   %ebp
      mov    %esp, %ebp
      mov    0xc(%ebp), %ecx
      add    $0x3, %ecx
      mov    0x8(%ebp), %eax
      shl    %ecx, %eax
      cmp    $0x8(%ebp), $0
      jne    $0x808472 // two lines down to leave
      mov    $0x1, %eax
      leave
      ret
```

```
iii)  push   %ebp
      mov    %esi, %ecx
      add    $0x3, %ecx
      mov    %edi, %eax
      shl    %ecx, %eax
      test   %edi, $0
      jne    $0x808472 // two lines down to leave
      mov    $0x1, %eax
      leave
      ret
```

i) *has a logical error that shifts y+3 by x rather than x by y+3.*

ii) *is the correct implementation.*

iii) *assumes the arguments are in registers rather than on the stack which is the x86-64 calling convention.*

B. Which of the following x86-64 implementations is correct for foo()? Circle the correct one and give at least one reason why the other two are not correct.

```
i)    add    $0x3, %rsi
      mov    %rdi, %rax
      shl    %rsi, %rax
      test   %rdi, %rdi
      jne    $0x808472 // two lines down to leave
      mov    $0x1, %rax
      leave
      ret


ii)   push   %rbx
      mov    %rsi, %rbx
      add    $0x3, %rbx
      mov    %rdi, %rax
      shl    %rbx, %rax
      test   %rdi, %rdi
      jne    $0x808472 // two lines down to leave
      mov    $0x1, %rax
      leaveq
      ret


iii)  mov    %rdi, %rdx
      add    $0x3, %rdx
      mov    %rsi, %rax
      shl    %rdx, %rax
      test   $0, %rdi
      jne    $0x808472 // two lines down to leave
      mov    $0x1, %rax
      ret
```

i) *is the correct implementation.*

ii) *%rbx is not popped from stack at end of procedure and leaveq is used which is really part of IA32 calling conventions.*

iii) *the arguments are out of order in the %rdi and %rsi registers, the test instruction is a bit-wise AND that sets condition codes, by one of the arguments being $0, the result will always be 0 and the jne conditional jump will never be taken.*

4. Stack Discipline (30 points)

The following function recursively computes the greatest common divisor of the integers a, b:

```
int gcd(int a, int b) {
     if (b == 0) {
          return a;
     } else {
          return gcd(b, a % b);
     }
}
```

Here is the x86_64 assembly for the same function:

```
4006c6 <gcd>:
4006c6:   sub       $0x18, %rsp
4006ca:   mov       %edi, 0x10(%rsp)
4006ce:   mov       %esi, 0x08(%rsp)
4006d2:   cmpl      $0x0, %esi
4006d7:   jne       4006df <gcd+0x19>
4006d9:   mov       0x10(%rsp), %eax
4006dd:   jmp       4006f5 <gcd+0x2f>
4006df:   mov       0x10(%rsp), %eax
4006e3:   cltd
4006e4:   idivl     0x08(%rsp)
4006e8:   mov       0x08(%rsp), %eax
4006ec:   mov       %edx, %esi
4006ee:   mov       %eax, %edi
4006f0:   callq     4006c6 <gcd>
4006f5:   add       $0x18, %rsp
4006f9:   retq
```

Note: `cltd` is an instruction that sign extends %eax into %edx to form the 64-bit signed value represented by the concatenation of [%edx | %eax].

Note: `idivl <mem>` is an instruction divides the 64-bit value [%edx | %eax] by the long stored at <mem>, storing the quotient in %eax and the remainder in %edx.

A. Suppose we call gcd(144, 64) from another function (i.e. main()), and set a breakpoint just before the statement “return a”. When the program hits that breakpoint, what will the stack look like, starting at the top of the stack and going all the way down to the saved instruction address in main()? Label all return addresses as "ret addr", label local variables, and leave all unused space blank.

Memory address on stack	**Value (8 bytes per line)**	
`0x7fffffffffffad0`	Return address back to main	<-%rsp points here at start of procedure
`0x7fffffffffffac8`	*1st of 3 local variables on stack (argument a = 144)*	
`0x7fffffffffffac0`	*2nd of 3 local variables on stack (argument b = 64)*	
`0x7fffffffffffab8`	*3rd of 3 local variables on stack (unused)*	
`0x7fffffffffffab0`	Return address back to gcd(144, 64)	
`0x7fffffffffffaa8`	*1st of 3 local variables on stack (argument a = 64)*	
`0x7fffffffffffaa0`	*2nd of 3 local variables on stack (argument b = 16)*	
`0x7fffffffffffa98`	*3rd of 3 local variables on stack (unused)*	
`0x7fffffffffffa90`	Return address back to gcd(64,16)	
`0x7fffffffffffa88`	*1st of 3 local variables on stack (argument a = 16)*	
`0x7fffffffffffa80`	*2nd of 3 local variables on stack (argument b = 0)*	
`0x7fffffffffffa78`	*3rd of 3 local variables on stack (unused)*	*<-%rsp at “return a” in 3rd recursive call*
`0x7fffffffffffa70`		

B. How many total bytes of local stack space are created in each frame (in decimal)?

______*32*_______ *24 allocated explicitly and 8 for the return address.*

C. When the function begins, where are the arguments (a, b) stored?

They are stored in the registers %rdi and %rsi, respectively.

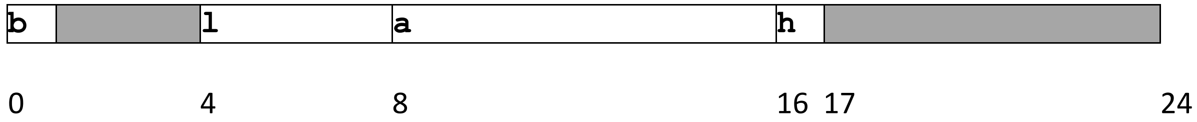
D. From a memory-usage perspective, why are iterative algorithms generally preferred over recursive algorithms?

Recursive algorithm continue to grow the stack for the maximum number of recursions which may be hard to estimate.

5. Structs (15 points)

A. Draw a picture of the following struct, specifying the byte offset of each of the struct's fields and the size of any areas of fragmentation. Assume a 64-bit architecture.

```
typedef struct blah {
     char b;
     int l;
     char *a;
     char h;
} blahblahblah;
```



B. How many bytes of internal fragmentation does the struct contain? External fragmentation?

Intenal fragmentation:_____*3 bytes after b*____

External fragmentation:_____*7 bytes after h*____

C. Reorder the fields of the struct to minimize fragmentation:

```
typedef struct blah {

        _______________       char *a;

        _______________       int l;

        _______________       char b;

        _______________       char h;

} blahblahblah;
```

D. What is the size of the reordered struct (including external fragmentation)?

_____16 bytes_____

E. How many bytes of internal fragmentation does the struct contain? External?

Intenal fragmentation:____*none*________

External fragmentation:____*2 bytes after h*____

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 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 (1st operand) to dst (2nd) with result stored in dst (2nd)
`sub`	subtract src (1st operand) from dst (2nd) with result stored in dst (2nd)
`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 1st operand
`jmp`	jump to address
`jne`	conditional jump to address if zero flag is not set
`cmp`	subtract src (1st operand) from dst (2nd) 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