CSE351 Autumn 2014 – Midterm Exam (29 October 2014)

(Version A)

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 4 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 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. 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.

Good Luck!

Name: ____________________________

UWNet ID: ____________________________

Quiz Section: ____________________________

Problem	**Max Score**	**Score**
1	20	
2	10	
3	30	
4	30	
TOTAL	**90**	

1. Integers and Floats (20 points)

We define two new types as follows:

`Nine_int`s are 9-bit signed two's complement integers.

`Nine_float`s are 9-bit floating point numbers with 4 bits for the exponent, 4 bits for the fraction, and 1 bit for the sign. **`Nine_float`**s are similar to IEEE floating point as far as layout of sign, exponent and fraction and represent special values (e.g. 0, pos and neg infinity, NAN) similar to how they are represented in 32 bit IEEE floating point.

A. What is the <u>largest</u> positive number we can represent with **`Nine_int`**s?

Bit pattern in binary:

Value in decimal:

B. What is the <u>bias</u> for **`Nine_float`**?

C. What is the <u>largest</u> positive number we can represent with **`Nine_float`**s?

Bit pattern in binary:

Value in decimal:

D. Assuming rules similar to those for conversions between IEEE floats and ints and addition in C, **circle all the statements below that are <u>TRUE</u>**.

a. It is possible to lose precision when converting from **`Nine_int`**s to **`Nine_float`**s.

b. It is possible to lose precision when converting from **`Nine_float`**s to **`Nine_int`**s.

c. The smallest negative number representable as a **`Nine_int`** **<** The smallest negative number representable as a **`Nine_float`**. (Reminder: -4 < -3)

d. Adding a negative **`Nine_float`** to a positive **`Nine_float`** will **<u>not</u>** result in a loss of precision.

2. Arrays (10 points)

Given the following C function:

```
long int sum_pair(long int z[16], long int dig)
{
   return z[dig] + z[dig + 1];
}
```

Write **x86-64** bit assembly code for this function here. You can assume that 0 <= **dig** < 15. Comments are not required but could help for partial credit.

sum_pair:

3. **Assembly to C (30 points)**

Given the C code for the function **`trick()`**, determine which IA32 and x86-64 code snippet corresponds to a correct implementation of **`trick()`**.

```
int trick (int *x, int y) {
  int temp = *x * 5;
  int result = temp & y;
  return result - y;
}
```

A. Circle **<u>all of</u>** the IA-32 implementations below that **correctly** implement **`trick()`** (there could be more than one). For implementations that are **not** correct give at least one reason each why it is not correct. (You do not need to give reasons why the correct ones are correct.)

i)

```
pushl   %ebp
movl    %esp, %ebp
leal    8(%ebp), %eax
movl    %eax, %edx
sall    $2, %eax
addl    %edx, %eax
andl    12(%ebp), %eax
subl    12(%ebp), %eax
popl    %ebp
ret
```

<u>Reason</u>:

ii)

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %eax
movl    (%eax), %edx
movl    %edx, %eax
addl    %eax, %edx
addl    %edx, %eax
andl    12(%ebp), %eax
subl    12(%ebp), %eax
popl    %ebp
ret
```

<u>Reason</u>:

iii)

```
pushl   %ebp
movl    %esp, %ebp
movl    12(%ebp), %edx
movl    8(%ebp), %eax
movl    (%eax), %eax
leal    (%eax,%eax,4), %eax
andl    %edx, %eax
subl    %edx, %eax
popl    %ebp
ret
```

<u>Reason</u>:

B. Circle **<u>all of</u>** the x86-64 implementations below that **correctly** implement **`trick()`** (there could be more than one). For implementations that are **not** correct give at least one reason each why it is not correct. (You do not need to give reasons why the correct ones are correct.)

```
i)          movl    (%rdi), %eax
            addl    (%rax), %eax
            addl    %eax, %eax
            andl    %esi, %eax
            subl    %esi, %eax
            ret
```

<u>Reason</u>:

```
ii)         movl    (%rdi), %eax
            leal    (%rax,%rax,4), %eax
            andl    %esi, %eax
            subl    %esi, %eax
            ret
```

<u>Reason</u>:

```
iii)        movl    (%rdi), %eax
            leal    (%eax,%eax,2), %eax
            addl    %eax, %eax
            andl    %esi, %eax
            subl    %esi, %eax
            ret
```

<u>Reason</u>:

4. Stack Discipline (30 points)

Examine the following recursive function:

```
long int treat(long int a, long int *b) {
  if (a <= 0) {
    return *b;
  } else {
    return treat(a-*b, b);
  }
}
```

Here is the x86_64 assembly for the same function:

```
4005fc <treat>:
4005fc:   sub    $0x18,%rsp
400600:   mov    %rdi,0x8(%rsp)
400605:   mov    %rsi,(%rsp)
400609:   cmpq   $0x0,0x8(%rsp)
40060f:   jg     0x40061a <treat+30>
400611:   mov    (%rsp),%rax
400615:   mov    (%rax),%rax
400618:   jmp    0x400638 <treat+60>
40061a:   mov    (%rsp),%rax
40061e:   mov    (%rax),%rax
400621:   mov    0x8(%rsp),%rdx
400626:   sub    %rax,%rdx
400629:   mov    (%rsp),%rax
40062d:   mov    %rax,%rsi
400630:   mov    %rdx,%rdi
400633:   callq  0x4005fc <treat>
400638:   add    $0x18,%rsp
40063c:   retq
```

Suppose we call **treat(7, &val)** from **main()**, with registers %**rsi** = **0x7ff**…**ffb00** and %**rdi** = 7. The value stored at address **0x7ff**…**ffb00** is the long int value 5. We set a breakpoint just before the statement "**return *b**" executes (i.e. we are just about to return from **treat()** without making another recursive call but have not yet executed the **add** instruction before **retq**). Draw what the stack will look like when the program hits that breakpoint. Start at the top of the stack and go all the way down to the return address back to **main()** shown currently on the stack. Give both a description of the item stored at that location and the value stored at that location. If a location on the stack is not used, write "unused" in the Description for that address and put "-----" for its Value. You may list the Values in hex or decimal. Unless preceded by **0x** we will assume decimal. It is fine to use **f**…**f** for sequences of **f**'s as shown above for %**rsi**. Add more rows to the table as needed.

%**rsp** points here at start of procedure →

Memory address on stack	Name/description of item	Value
0x7fffffffffffad0	Return address back to **main**	0x400827
0x7fffffffffffac8		
0x7fffffffffffac0		
0x7fffffffffffab8		
0x7fffffffffffab0		
0x7fffffffffffaa8		
0x7fffffffffffaa0		
0x7fffffffffffa98		
0x7fffffffffffa90		
0x7fffffffffffa88		
0x7fffffffffffa80		
0x7fffffffffffa78		
0x7fffffffffffa70		
0x7fffffffffffa68		
0x7fffffffffffa60		

B. What is the value stored in register **`$rsp`** at the start of the procedure (in hex or decimal)?

C. What is the value stored in register **`$rsp`** when the breakpoint is reached (in hex or decimal)?

D. What value is returned by **`treat(7, &val)`**?

E. Where will that return value be found?

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