CSE 351 Autumn 2015 – Midterm Exam (4 November 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 95 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 (as it appears on your ID):__Sample Solution __________

Student Number:___________________________

UWNet ID:___________________________

Problem	Max Score	Score
1	25	
2	10	
3	10	
4	20	
5	30	
TOTAL	**95**	

1. Integers and Floats (25 points total)

A. (21 points) Given the following declarations:

Note: order of questions was different on different exams.

```
int x1 = …; // x1 > 0
int x2 = …; // x2 < 0
float f = …;
double d1 = …;
double d2 = …;
```

Assume neither **d1, d2** nor **f** is NaN.

For each of the following, indicate if it is TRUE for all possible values of the given variables (note that **x1** and **x2** have specific ranges). **If not, select FALSE and give a BRIEF one sentence justification for your answer** – BE SPECIFIC. You do not need to give a justification for true answers.

	Circle One	
1) `d1 == (double)(float) d1`	TRUE	FALSE

Float cannot represent as much precision or range as double.

2) `x1 == (int)(float) x1`	TRUE	FALSE

Float only has 23 bits for precision vs. 32 bits in int

3) `x2 == (int)(double) x2`	TRUE	FALSE
4) `d1 == -(-d1)`	TRUE	FALSE

Note: this just requires flipping the sign bit twice.

5) `(d1 + d2) - d1 == d2`	TRUE	FALSE

Floating point operations are not associative. If d1 is very big and d2 is very small, then we might "lose" d2 when adding it to d1. d1 + d2 could also overflow and become NaN.

6) `x1 + x2 will never overflow`	TRUE	FALSE

Note: x1 > 0, x2 < 0

7) `f == (float)(double) f`	TRUE	FALSE

B. (4 points) What is the largest positive number we can represent with a 10-bit signed two's complement integer?

Bit pattern in binary:

01 1111 1111

Value in decimal:

$$2^8 + \ldots + 2^0 = 2^9 - 1 = 512 - 1 = 511$$

2. C to Assembly (10 points)

Given the following C function:

```
long happy(long *x, long y, long z) {
  if (y > z)
    return z + y;
  else
    return *x;
}
```

Write **x86-64** bit assembly code for this function here. Comments are not required but could help for partial credit. We are not judging you on the efficiency of your code, just the correctness. It is fine to leave off the size suffixes if you prefer to (e.g. b, w, l, q).

```
happy:
       cmp %rdx, %rsi # y:z
       jle .else
       leaq (%rsi, %rdx), %rax   # y > z    %rax = z + y
       ret
.else:
       movq (%rdi), %rax         # y <= z   %rax = *x
       ret
```

Also fine to swap the if and else clauses:

```
happy:
       cmp %rdx, %rsi # y:z
       jg .else
       movq (%rdi), %rax         # y <= z   %rax = *x
       ret
.else:
       leaq (%rsi, %rdx), %rax   # y > z    %rax = z + y
       ret
```

3. C to Assembly (10 points)

Given the following C function:

```
long silly(long *z, long index){
   z = z + 2;
   return z[index] - 5;
}
```

Write **x86-64** bit assembly code for this function here. Comments are not required but could help for partial credit. We are not judging you on the efficiency of your code, just the correctness. It is fine to leave off the size suffixes if you prefer to (e.g. b, w, l, q).

```
silly:
         movq 16(%rdi, %rsi, 8), %rax
         subq $5, %rax
         ret
```

Fine to do this with more instructions.

4. Assembly to C (20 points)

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

```
long sunny(long *z, long counter){
  long temp = *z;
  while (counter > 1) {
    temp = temp * 8;
    counter--;
  }
  return temp;
}
```

Note: order of questions was different on different exams.

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

```
a)          movq    (%rdi), %rax
            jmp .L10
.L11:
            leaq    8(%rax), %rax
.L10:
            subq    $1, %rsi
            jg  .L11
            rep ret
```

Circle One:

Correct **<u>Incorrect</u>**

<u>If Incorrect, give Reason</u>:

- **Adds 8 instead of multiplying.**

```
b)
            movq    (%rdi), %rax
            jmp .L10
.L11:
            salq    $3, %rax
            subq    $1, %rsi
.L10:
            cmpq    $1, %rsi
            jg  .L11
            rep ret
```

Circle One:

<u>Correct</u> Incorrect

<u>If Incorrect, give Reason</u>:

c)

```
            movq    %rdi, %rax
            jmp .L10
.L11:
            salq    $3, %rax
            subq    $1, %rsi
.L10:
            cmpq    $1, %rsi
            jg  .L11
            rep ret
```

Circle One:

Correct **Incorrect**

If Incorrect, give Reason:

First movq needs to put *z, not z into rax.

d)

```
            movq    (%rdi), %rax
.L11:       cmpq    $1, %rsi
            jle .L10

            leaq    (,%rax,8), %rax
            subq    $1, %rsi
            jmp .L11
.L10:
            rep ret
```

Circle One:

Correct Incorrect

If Incorrect, give Reason:

e)

```
            leaq    (%rdi), %rax
            jmp .L10
.L11:
            salq    $3, %rax
            subq    $1, %rsi
.L10:
            cmpq    $1, %rsi
            jg  .L11
            rep ret
```

Circle One:

Correct **Incorrect**

If Incorrect, give Reason:

First instruction should be movq, not leaq. Needs to move *z not &z into rax.

5. Stack Discipline (30 points)

Examine the following recursive function:

```
long magic(long x, long *y) {
  long temp;
  if (x < 2) {
    return *y;
  } else {
    temp = *y + 1;
    return x + magic(x-3, &temp);
  }
}
```

Here is the x86_64 assembly for the same function:

```
4005f6 <magic>:
4005f6:   cmp    $0x1,%rdi
4005fa:   jg     0x400600 <magic+10>
4005fc:   mov    (%rsi),%rax
4005ff:   retq
400600:   push   %rbx
400601:   sub    $0x10,%rsp
400605:   mov    %rdi,%rbx
400608:   mov    (%rsi),%rax
40060b:   add    $0x1,%rax
40060f:   mov    %rax,0x8(%rsp)
400614:   lea    -0x3(%rdi),%rdi
400618:   lea    0x8(%rsp),%rsi
40061d:   callq  0x4005f6 <magic>
400622:   add    %rbx,%rax
400625:   add    $0x10,%rsp
400629:   pop    %rbx
40062a:   retq
```

Suppose we call **`magic`** from **`main()`**, with registers %**`rsi`** = **`0x7ff`**…**`ffbaa`** and %**`rdi`** = 7. The value stored at address **`0x7ff`**…**`ffbaa`** is the long value 3. We set a <u>breakpoint</u> at "**`return *y`**" (i.e. we are just about to return from **`magic()`** without making another recursive call). We have executed the **`mov`** instruction at **`4005fc`** but have not yet executed the **`retq`**.

Fill in the register values on the next page and **draw what the stack will look like <u>when the program hits that breakpoint</u>**. 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. <u>Also, fill in the box on the next page to include the value this call to **`magic`** will *finally* return to **`main`**.</u>

Version 1

Register	Original Value	Value <u>at Breakpoint</u>
rsp	0x7ff…ffad0	0x7fffffffffffa90
rdi	7	1
rsi	0x7ff…ffbaa	0x7fffffffffffaa0
rbx	2	4
rax	9	5

DON'T FORGET → What value is **finally** returned to `main` by this call? **16**

Memory address on stack	Name/description of item	Value
0x7fffffffffffad0	Return address back to `main`	0x400827
0x7fffffffffffac8	Old rbx	2
0x7fffffffffffac0	temp	4
0x7fffffffffffab8	Unused	-------
0x7fffffffffffab0	Return address	0x400622
0x7fffffffffffaa8	Old rbx	7
0x7fffffffffffaa0	temp	5
0x7fffffffffffa98	Unused	-------
0x7fffffffffffa90	Return address	0x400622
0x7fffffffffffa88		
0x7fffffffffffa80		
0x7fffffffffffa78		
0x7fffffffffffa70		
0x7fffffffffffa68		
0x7fffffffffffa60		

Version 2

Register	Original Value	Value at Breakpoint
rsp	0x7ff…ffad0	0x7fffffffffffa90
rdi	6	0
rsi	0x7ff…ffbaa	0x7fffffffffffaa0
rbx	1	3
rax	8	4

DON'T FORGET → What value is **finally** returned to **main** by this call? 13

Memory address on stack	Name/description of item	Value
0x7fffffffffffad0	Return address back to **main**	0x400827
0x7fffffffffffac8	Old rbx	1
0x7fffffffffffac0	temp	3
0x7fffffffffffab8	Unused	-------
0x7fffffffffffab0	Return address	0x400622
0x7fffffffffffaa8	Old rbx	6
0x7fffffffffffaa0	temp	4
0x7fffffffffffa98	Unused	-------
0x7fffffffffffa90	Return address	0x400622
0x7fffffffffffa88		
0x7fffffffffffa80		
0x7fffffffffffa78		
0x7fffffffffffa70		
0x7fffffffffffa68		
0x7fffffffffffa60		

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
`sal` shift data in the dst to the left (arithmetic shift) by the number of bits specified in 1st operand

`jmp` jump to address
`jg` conditional jump to address if greater than
`jle` conditional jump to address if less than or equal
`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