

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): _____

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:

```
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.

	<u>Circle One</u>	
1) <code>d1 == (double)(float) d1</code>	TRUE	FALSE
2) <code>x1 == (int)(float) x1</code>	TRUE	FALSE
3) <code>x2 == (int)(double) x2</code>	TRUE	FALSE
4) <code>d1 == -(-d1)</code>	TRUE	FALSE
5) <code>(d1 + d2) - d1 == d2</code>	TRUE	FALSE
6) <code>x1 + x2</code> will never overflow	TRUE	FALSE
7) <code>f == (float)(double) f</code>	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:

Value in decimal:

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:

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:

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;
}
```

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

a)

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

Circle One:

Correct

Incorrect

If Incorrect, give Reason:

b)

```
    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:

```

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:

```

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:

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 breakpoint 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 when the program hits that breakpoint. 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. Also, fill in the box on the next page to include the value this call to `magic` will *finally* return to `main`.

Register	Original Value	Value at <u>Breakpoint</u>
rsp	0x7ff...ffad0	
rdi	7	
rsi	0x7ff...ffbaa	
rbx	2	
rax	9	

DON'T FORGET



What value is **finally** returned to **main** by this call?

Memory address on stack	Name/description of item	Value
0x7fffffffad0	Return address back to main	0x400827
0x7ffffffffffac8		
0x7ffffffffffac0		
0x7ffffffffffab8		
0x7ffffffffffab0		
0x7ffffffffffaa8		
0x7ffffffffffaa0		
0x7ffffffffffa98		
0x7ffffffffffa90		
0x7ffffffffffa88		
0x7ffffffffffa80		
0x7ffffffffffa78		
0x7ffffffffffa70		
0x7ffffffffffa68		
0x7ffffffffffa60		

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

<code>%rax</code>	Return value	<code>%r8</code>	Argument #5
<code>%rbx</code>	Callee saved	<code>%r9</code>	Argument #6
<code>%rcx</code>	Argument #4	<code>%r10</code>	Caller saved
<code>%rdx</code>	Argument #3	<code>%r11</code>	Caller Saved
<code>%rsi</code>	Argument #2	<code>%r12</code>	Callee saved
<code>%rdi</code>	Argument #1	<code>%r13</code>	Callee saved
<code>%rsp</code>	Stack pointer	<code>%r14</code>	Callee saved
<code>%rbp</code>	Callee saved	<code>%r15</code>	Callee saved