CSE 351 Spring 2019 – Midterm Exam (3 May 2019)

Please read through the entire examination first!

- You have 60 minutes for this exam. Don't spend too much time on any one problem!
- The last page is a reference sheet. Feel free to detach it from the rest of the exam.
- The exam is CLOSED book and CLOSED notes (no summary sheets, no calculators, no mobile phones).

There are 5 problems for a total of 53 points. The point value of each problem is indicated in the table below. Write your answer neatly in the spaces provided.

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!

Your Name:____Sample Solution________

UWNet ID:___________________________

Problem	**Topic**	**Max Score**
1	Integers & Floats	13
2	Hardware to Software	8
3	C & Assembly	11
4	Stack Discipline	12
5	Pointers & Memory	9
TOTAL		**53**

1. Integers and Floats (13 points total)

We define two new types as follows:

`Ten_int`s are 10-bit signed two's complement integers.

`Ten_float`s are 10-bit floating point numbers with 4 bits for the exponent, 5 bits for the fraction, and 1 bit for the sign. **`Ten_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) (2 pts) What is the most negative number we can represent with **`Ten_int`**s?

Give the bit pattern in binary:

1000000000

Give the value in decimal:

$-(2^9) = -512$

-512

b) (4 pts) Convert the following **`Ten_float`**s bit pattern into decimal.

Bit pattern in binary: 0 1011 11000

Give the bias for **`Ten_float`**

7

$2^{4-1} - 1 = 8 - 1 = 7$

Give the value in decimal:

28

Exponent = 11 – 7 = 4
Value = 0b1.11 * 2^4 = 0b11100 = 16 + 8 + 4 = 28

c) (2 pts) What is the result in binary if you add 1 to the maximum positive **`Ten_int`**s number?

0 1111 1111 + 1 = (the most negative number)

10000 00000

In a sentence, describe what is happening in this case:

This is overflow: adding a positive number to a positive number, resulting in a negative number.

d) (5 pts) Assuming rules similar to those for conversions between IEEE floats and ints and addition in C, **indicate whether the following statements are True or False**.

- **TRUE** / **FALSE**: Given a **`Ten_float`** that is negative, it is possible to lose precision when converting from a **`Ten_float`** to a **`Ten_int`**.

- **TRUE** / **FALSE**: The largest positive number representable as a **`Ten_int <`** the largest non-infinite positive number representable as a **`Ten_float`**.

- **TRUE** / **FALSE**: Given a **`Ten_int`** that is positive, it is possible to lose precision when converting it to a **`Ten_float`**.

- **TRUE** / **FALSE**: Adding a negative **`Ten_float`** to a positive **`Ten_float`** will **never** result in a loss of precision.

- **TRUE** / **FALSE**: Adding a negative **`Ten_int`** to a positive **`Ten_int`** will **never** result in overflow.

2. Hardware to Software (8 points total)

a) (2 pts) Give **one disadvantage** of sign magnitude representation of ints compared to two's complement representation of ints.

- **Two representations of 0 (bad for checking equality)**
- **Arithmetic is cumbersome. Example: 4-3 != 4+(-3)**

b) (2 pts) What does having a standardized calling convention enable that would not be possible otherwise?

Code generated by one compiler can be called by code generated by another compiler. This makes it possible to call library code.

c) (4 pts) Your friend proposes that the next generation of x86 machines only allow you to refer to the full 64-bit versions of registers (we could no longer refer to only a portion of a register and instructions would all have the suffix `q`). For example, instructions like `movb (%rax), $dil` would instead have to be: `movq (%rax), $rdi`

Give **one drawback** and **one benefit** of this approach.

Drawbacks:

- **All datatypes are now the same size, overall memory use by your data is likely to be more**
- **X86 programs already written may no longer work anymore, no backwards compatibility**

Benefits:

- **No need to worry about alignment of different type sizes (since there is in effect only one size)**
- **Addresses could refer to 8 bytes instead of 1 byte, since there is no need or ability to access smaller chunks of memory, so your address space could become 8x larger without changing the bit width of an address**
- **Machine Encoding of instructions may be smaller – no need to specify suffix or as many possible registers**

3. C and Assembly (11 points total)
You are given the following x86-64 assembly function:

```
mystery:
        movl    $0, %edx
        movl    $0, %eax
.L3:
        cmpl    %esi, %edx
        jge     .L1
        movslq  %edx, %rcx
        addl    (%rdi,%rcx,4), %eax
        addl    $1, %edx
        jmp     .L3
.L1:
        rep ret
```

a) (1 pt) What variable type would `%rdi` be in the corresponding C program?

int*

b) (1 pt) What variable type would `%rsi` be in the corresponding C program?

int

c) (7 pts) Fill in the missing C code that is equivalent to the x86-64 assembly above:

```
______int_____ mystery( (answer to a) rdi, (answer to b) rsi) {
    ___ int ______ eax = ___0 ;________

    for (int edx = 0; edx < rsi; edx++) {
        eax += rdi[edx];
    }

    return eax;
}
```

d) (2 pts) In 1 sentence, describe what this function is doing?

Summing the first rsi elements of the int array starting at rdi

4. Stack Discipline (12 points total)

Examine the following recursive function:

```
long husky(long *x, long y) {
   long woof = 0;
   if (y < 3) {
    return *x * 3;
  } else {
    woof = y + 1;
    return woof + husky(&woof, y - 1);
  }
}
```

Here is the x86_64 assembly for the same function:

```
000000000040057a <husky>:
  40057a:      cmp    $0x2,%rsi
  40057e:      jg     400588 <husky+0xe>
  400580:      mov    (%rdi),%rax
  400583:      lea    (%rax,%rax,2),%rax
                                                   Breakpoint
  400587:      retq
  400588:      push   %rbx
  400589:      sub    $0x10,%rsp
  40058d:      lea    0x1(%rsi),%rbx
  400591:      mov    %rbx,0x8(%rsp)
  400596:      sub    $0x1,%rsi
  40059a:      lea    0x8(%rsp),%rdi
  40059f:      callq  40057a <husky>
  4005a4:      add    %rbx,%rax
  4005a7:      add    $0x10,%rsp
  4005ab:      pop    %rbx
  4005ac:      retq
```

We call **husky** from **main()**, with registers %**rdi** = **0x7ff**…**ffbd8** and %**rsi** = 4. The value stored at address **0x7ff**…**ffbd8** is the long value 16 (0x10). We set a <u>breakpoint</u> at "**return *x * 3**" (i.e. we are just about to return from **husky()** without making another recursive call). We have executed the **lea** instruction at **400583** 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 for %**rdi**. 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 **husky** will *finally* return to **main**.</u>

Register	Original Value	Value at Breakpoint
rsp	0x7ff…ffbd0	0x7fffffffffffb90
rdi	0x7ff…ffbd8	0x7fffffffffffba0
rsi	4	2
rbx	9	4
rax	7	12

DON'T FORGET → What value is **finally** returned to **main** by this call? **21**

Memory address on stack	Name/description of item	Value
0x7fffffffffffbd8	Local var in **main**	0x10
0x7fffffffffffbd0	Return address back to **main**	0x400975
0x7fffffffffffbc8	**Old rbx**	*9*
0x7fffffffffffbc0	***woof***	*5*
0x7fffffffffffbb8	*Unused*	*Unknown*
0x7fffffffffffbb0	**Return address back to husky**	**0x4005a4**
0x7fffffffffffba8	*Old rbx*	5
0x7fffffffffffba0	***woof***	4
0x7fffffffffffb98	*Unused*	*Unknown*
0x7fffffffffffb90	**Return address back to husky**	**0x4005a4**
0x7fffffffffffb88		
0x7fffffffffffb80		
0x7fffffffffffb78		
0x7fffffffffffb70		
0x7fffffffffffb68		
0x7fffffffffffb60		

5. Pointers, Memory & Registers (9 points total)

Assuming a 64-bit x86-64 machine (little endian), you are given the following variables and initial state of memory (values in hex) shown below:

Address	+0	+1	+2	+3	+4	+5	+6	+7
0x00	AB	EE	1E	AC	D5	8E	10	E7
0x08	42	84	32	2D	A5	F2	3A	CA
0x10	83	14	53	B9	70	03	F4	31
0x18	01	20	FE	34	46	E4	FC	52
0x20	4C	A8	B5	C3	D0	ED	53	17

```
long* yp = 0x18;
int* ip = 0x10;
short* sp = 0x08;
```

a) (5 pts) Fill in the type and value for each of the following C expressions. If a value cannot be determined from the given information answer UNKNOWN.

Expression (in C)	Type	Value (in hex)
`*(ip + 2)`	int	0x34FE2001
`sp + 3`	short*	0x0E
`yp[-1]`	long	0x31F40370B9531483
`(*sp) + 1`	short	0x8443
`&(ip[1])`	int*	0x14

b) (4 pts) What are the values (in hex) stored in each register shown after the following x86 instructions are executed? *Remember to use the appropriate bit widths.*

Instruction	Register	Value (in hex)
	`%rax`	0x0000 0000 0000 0002
	`%rsi`	0x0000 0000 0000 0010
`movw (,%rax,4), %bx`	`%bx`	0x8442
`movswq 4(,%rax,2), %rcx`	`%rcx`	0xffffffffffff8442
`leaq (%rsi,%rax,8), %rdi`	`%rdi`	0x0000000000000020
`addl 4(%rax,%rax), %esi`	`%esi`	0x2D328452