CSE 351 Winter 2020 – Midterm Exam (Feb 10, 2020)

Please read through the entire examination first!

- You have 50 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. You do NOT need to submit the reference sheet with your 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 50 points. The point value of each problem is indicated in the table below. Write your answer neatly in the spaces provided.

- You may leave once you are done with the exam, however during the last 5 minutes of the exam we will ask that everyone remain in the room until the bell rings.
- You MUST stop writing once the bell rings. Points will be deducted if you are writing beyond the bell.

Good Luck!

Your Name:___________________________

UWNet ID (email):___________________________

Problem	Topic	Max Score
1	Integers & Floats	12
2	Pointers & Memory	8
3	Hardware to Software	6
4	C & Assembly	12
5	Stack Discipline	12
TOTAL		**50**

1. Integers and Floats (12 points total)

a) (1 pt) If we have only 9 bits and are using **two's complement representation**, how many **positive, non-zero** numbers can we represent?

b) (1 pt) If we have only 9 bits and are using **sign-magnitude representation**, how many **positive, non-zero** numbers can we represent?

c) (6 pt) Given the following in C: **`signed char x = 0b 0101 0001`**

i. (2 pts) What is the value of **`x`** in decimal? You may represent your answer as the sum of powers of 2.

ii. (4 pts) For each of the following expressions, indicate whether it will result in a positive, negative or a zero result. (Circle one)

• **`x >> 3`**	Positive	Negative	Zero
• **`x + 0x62`**	Positive	Negative	Zero
• **`x << 1`**	Positive	Negative	Zero
• **`!(x & 0x1)`**	Positive	Negative	Zero

d) (4 pts) Assume we have a floating point representation that follows the same conventions as IEEE 754, except that is uses 9 bits. 1 bit is for the sign, 4 bits are used for the exponent and 4 bits are used for the mantissa.

i. What is the bias for this representation

ii. What is the decimal value encoded by the bit pattern: 1 1001 1110
For any credit, show your work.

Is this number (circle one):

positive or

negative

2. Pointers, Memory & Registers (8 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
0x30	51	32	43	7A	3B	FA	E4	76
0x38	48	22	00	88	9A	B2	CD	27
0x40	4F	17	B3	2B	A0	A7	BC	F9
0x48	40	03	08	15	A9	8B	F2	3F
0x50	AA	BB	CC	DD	EE	FB	01	02

```
char* cp = 0x30;
long* qp = 0x48;
int* ip = 0x3C;
```

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`		
`cp + 13`		
`qp[-2] + 1`		
`*((char*) qp)`		
`*(((short*) ip) - 3)`		

b) (3 pts) What are the values (in hex) stored in each register shown after the following x86 instructions are executed? Refer to the state of memory shown above. If a value cannot be determined from the given information answer UNKNOWN. *Remember to use the appropriate bit widths.*

```
leaq (%rsi,%rsi,4), %rbx
movw 10(%rax), %cx
movsbl -2(%rax,%rsi,2), %edi
```

Register	Value (in hex)
%rax	0x0000 0000 0000 0040
%rsi	0x0000 0000 0000 0002
%rbx	
%cx	
%edi	

3. Hardware to Software (6 points total)

Ruth placed a bet with your TAs that they couldn't make and sell CPUs that implement the x86-64 instruction set better than Intel does. Your TAs came up with the following list of suggested changes. For each modification, circle TRUE if the new architecture still implements the x86-64 instruction set architecture, or FALSE if not**. Also very briefly explain your answer in the space provided.**

a) Make **`%r10`** and **`%r11`** additional return value registers instead of their original function.
This change would still implement the x86-64 instruction set:

TRUE FALSE Why?____

b) Reorganize all of the registers on the physical chip.
This change still implements the x86-64 instruction set:

TRUE FALSE Why?____

c) Re-implement the **`add`** instruction to be 200% faster.
This change still implements the x86-64 instruction set:

TRUE FALSE Why?____

4. C and Assembly (12 points total)

Consider the following function given in x86-64 assembly:

```
fun:
      movl   $0, %eax                  # Line 1
      jmp    .L2                       # Line 2
.L3:
      addq   $1, %rsi                  # Line 3
.L2:
      cmpq   %rdx, %rsi                # Line 4
      jge    .L5                       # Line 5
      movl   (%rdi,%rsi,4), %ecx       # Line 6
      testl  %ecx, %ecx                # Line 7
      jns    .L3                       # Line 8
      addl   %ecx, %eax                # Line 9
      jmp    .L3                       # Line 10
.L5:
      rep ret                          # Line 11
```

a) (4 pts) Fill in the function's C signature with the correct **C types**:

```
_________ fun(_________ arg1, _________ arg2, _________ arg3)
```

b) (2 pts) This function contains a while loop. What is the loop condition? Feel free to use register names as variables.

```
while (_____________________________________)
```

c) (2 pts) Rewrite the conditional jump on lines 7 and 8 using **`cmpl`** instead of **`testl`**. Write correct assembly code that could be substituted for line 7 & line 8.

```
        cmpl _______________, _______________      # Line 7

_______________ ________________________________   # Line 8
```

d) (4 pts) Briefly describe what you think this function accomplishes. What is the value returned by this function and how it is computed? (at a high level, not line-by-line)

5. Stack Discipline (12 points total)

Examine the following recursive function:

```
long dubs(long x, int* y) {
  if (x > 2) {
    return x + *y + dubs(x - 2, y);
  } else {
    return 3 * x + *y;
  }
}
```

Here is the x86_64 assembly for the same function:

```
0000000000400507 <dubs>:
  400507:      cmp    $0x2,%rdi
  40050b:      jg     400518 <dubs+0x11>
  40050d:      lea    (%rdi,%rdi,2),%rax
  400511:      movslq (%rsi),%rdx
  400514:      add    %rdx,%rax
  400517:      retq
  400518:      push   %rbx
  400519:      movslq (%rsi),%rax
  40051c:      lea    (%rax,%rdi,1),%rbx
  400520:      sub    $0x2,%rdi
  400524:      callq  400507 <dubs>
  400529:      add    %rbx,%rax
  40052c:      pop    %rbx
  40052d:      retq
```

Breakpoint

We call **dubs** from **main()**, with registers %**rsi** = **0x7ff**…**ffbd8** and %**rdi** = 5. The value stored at address **0x7ff**…**ffbd8** is the int value 4 (0x4). We set a breakpoint at "**return 3 * x + *y**" (i.e. we are just about to return from **dubs()** without making another recursive call). We have executed the **add** instruction at **400514** 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 for %**rdi**. Add more rows to the table as needed.

** *** ** *** ** DON'T FORGET! Also, fill in the box on the next page to include the value this call to **dubs** will *finally* return to **main**. ** *** ** *** **

Register	Original Value	Value **at Breakpoint**
`rdx`	`8`	
`rsp`	`0x7ff…ffbd0`	
`rdi`	`5`	
`rsi`	`0x7ff…ffbd8`	
`rbx`	`678`	
`rax`	`33`	

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

Memory address on stack	Name/description of item	Value
`0x7fffffffffffbd8`	Local var in **`main`**	`0x4`
`0x7fffffffffffbd0`	Return address back to **`main`**	`0x400986`
`0x7fffffffffffbc8`		
`0x7fffffffffffbc0`		
`0x7fffffffffffbb8`		
`0x7fffffffffffbb0`		
`0x7fffffffffffba8`		
`0x7fffffffffffba0`		
`0x7fffffffffffb98`		
`0x7fffffffffffb90`		
`0x7fffffffffffb88`		
`0x7fffffffffffb80`		
`0x7fffffffffffb78`		
`0x7fffffffffffb70`		
`0x7fffffffffffb68`		

********************* SCRATCH PAPER *********************