CS 449 Fall 2019 – Midterm Exam

Please read through the entire examination first! We designed this exam so that it can be completed in 75 minutes and, hopefully, this estimate will prove to be reasonable.

There are 6 problems for a total of 90 points. The point value of each problem is indicated in the table below. **Please write your answer neatly in the answer spaces provided**. If you need more space, you can write on the space on the sheet where the question is posed. Do NOT use any other paper to hand in your answers. 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.

And... Relax. You are here to learn!

Last Name: Peiffer

First Name: Avery

Pitt ID#: 4042056

I certify that all work is my own. I had no prior knowledge of exam contents nor will I share the contents with any student in CS 449 who has not yet taken the exam. Violation of these terms may result in a failing grade. ***(Please sign below.)***

Signature: Avery Peiffer

Problem	Max Score	Score
1	15	
2	10	
3	20	
4	20	
5	15	
6	10	
TOTAL	**90**	

1. Warm-up (15 points)

A. (5 points) If we have six (6) bits in which to represent integers, what is largest unsigned number and what is largest 2s complement number we can represent (**in decimal**)?

Largest unsigned number: 63

$2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$

$32 + 16 + 8 + 4 + 2 + 1$

Largest 2s complement number: 31

(largest magnitude (abs. value) is -32 (100 000)

100000

011111

B. (10 points) If %eax stores x and %ebx stores y, what do the following lines of assembly compute? Note that the result is in %eax.

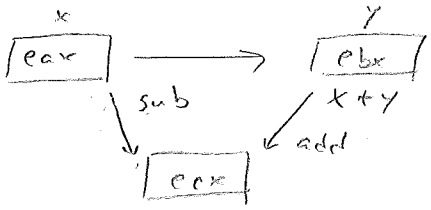
```
    mov     %ebx, %ecx
    add     %eax, %ebx
    je      .L1
    sub     %eax, %ecx
    je      .L1
    xor     %eax, %eax
    jmp     .L2
L1:
    mov     $1, %eax
L2:

    ...
```

mov S, D

add S, D

sub S, D



x + y = 0 ? return 1 :

Write below succinctly (1-2 sentences) what expression the x86 code above computes:

If (x + y) == 0 or (x - y) == 0, the result (%eax), is set to 1. Otherwise, a different branch is executed.

Avery Peiffer
4042056

2. Floating Point Representation (10 points)

Suppose we have 16-bit floating point numbers where 6 bits are assigned to the exponent and 9 bits to the fraction and 1 to the sign bit.

A. (2 points) What is the bias for this float (**in decimal**)?

$2^{6-1} - 1 = 2^5 - 1$

$= 31$

31

B. (8 points) Given the decimal number 3.625, calculate the fraction (frac) and exponent (exp) that would appear in the floating point representation. (Note: you may leave your answer in decimal for the exponent.)

3.625

↓

$s = 0$

$M = 1.625 = 1.101\,000\,000$

Implied

1.8125×2^1

$.5 + .25 + 0 + .0625$

1.75

+ .0625

1.8125

frac = | 110100000 |

exp = | 100000 | 32

Complete Bit pattern = | 0 100000 110100000 |

$e = 32$

$exp = 32 - 31 = 1$

$M = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} = 1.8125$

0 100000 10 1000 000

32 1.625

$1.625 \cdot 2^1$

$3.625 = M \cdot 2^1$

1.8125

2 | 3.625

-2

16

-16

02

-0

10

1

1.8125

+ 1.8125

3.6250

3. Pointers & Memory (20 points)

For this problem we are using a 64-bit x86-64 machine (**little endian**). The current state of memory (values in hex) is shown below:

Word Addr	+0	+1	+2	+3	+4	+5	+6	+7
0x00	AC	AB	03	01	BA	5E	BA	11
0x08	5E	00	68	0C	BE	A7	CE	FA
0x10	1D	B0	99	DE	AD	60	BB	40
0x18	14	1D	EC	AF	EE	FF	CO	70
0x20	BA	B0	41	20	80	AA	BE	EF

```
char* charP = 0x12
int*  intP  = 0x8
long* longP = 0x30
```

(A) Using the values shown above, complete the C code below to fulfill the behaviors described in the comments using pointer arithmetic. [8 pt]

```
char v1 = *(charP + ___9___);     // set v1 = 0xAF
int* v2 = &intP[___3___];         // set v2 = 0x14
```

(B) What are the values (in hex) stored in each register shown after the following x86-64 instructions are executed? We are still using the state of memory shown above.
Remember to use the appropriate bit widths. [12 pt]

```
leab  7(%rsi),      %r9b

movl  (%rdi,%rsi),  %eax

movzbq -2(,%rsi,8), %r8
```

Register	Data (hex)
%rdi	0x 0000 0000 0000 0019
%rsi	0x 0000 0000 0000 0003
%r9b	0x 0A
%eax	0x 70 C0 FF EE
%r8	0x 0000 0000 0000 0000

Avery Peiffer
4042056

4. C Programming (20 points)

Your task is to implement a simple stack adding machine that uses a stack data structure (this is independent of the stack for calling/returning from subroutines). For example, Push 2, Push 3, PopAdd yields 5 in the top of the stack. Following this with Push 1, PopAdd would yield 6. Fill in the code for functions **push** and **popadd** so they meet the specifications stated in the comments. Do not make other code modifications. Calls to **malloc** always return a valid address. You may not need all the lines for your code solution.

```
/* Each item on the stack is represented by a pointer
to the previous element (NULL if none) and its value. */
typedef struct stack_el {
    struct stack_el *prev;
    double val;
} stack_el;

/* PUSH: Push new value to top of stack.
Return pointer to new top of stack. */
stack_el* push(stack_el *top_of_stack, double v) {
    stack_el * top = (stack_el*) malloc (sizeof (stack_el));
    top -> prev = top_of_stack;
    top -> val = v;
    return top;
}
/* POPADD: Pop top stack element and add its value to the new top's value. Return new
top of stack. Free no longer used memory.  Do not change the stack if it has fewer
than 2 elements. */
stack_el* popadd(stack_el *top_of_stack) {
    if (top_of_stack -> prev == NULL) {
        return top_of_stack;
    }
    stack_el * new_top = top_of_stack -> prev;
    new_top -> val += top_of_stack -> val;
    free (top_of_stack);
    return new_top;
}
```

5. Functions & Stack Discipline (15 points)

The recursive power function `power()` calculates `base^pow` and its x86-64 disassembly is shown below:

```
int power(int base, unsigned int pow) {
  if (pow) {
    return base * power(base,pow-1);
  }
  return 1;
}
```

```
00000000004005a0 <power>:
  4005a0:  85 f6              testl  %esi,%esi
  4005a2:  74 10              je     4005b4 <power+0x14>
  4005a4:  53                 pushq  %rbx
  4005a5:  89 fb              movl   %edi,%ebx
  4005a7:  83 ee 01           subl   $0x1,%esi
  4005aa:  e8 f1 ff ff ff     call   4005a0 <power>
  4005af:  0f af c3           imull  %ebx,%eax
  4005b2:  eb 06              jmp    4005ba <power+0x1a>
  4005b4:  b8 01 00 00 00     movl   $0x1,%eax
  4005b9:  c3                 ret
  4005ba:  5b                 popq   %rbx
  4005bb:  c3                 ret
```

(A) How much space (in bytes) does this function take up in our final executable? [2 pt]

400566
− 4005a0 = 16 + 1 + 12 = 28
00016

28 bytes

(B) Which register is being saved on the stack? [2 pt]

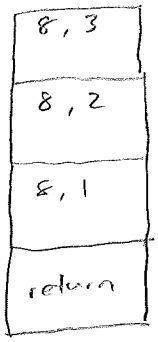
%rbx

Avery Peiffer
4041256

(C) What is the return address to `power` that gets stored on the stack? Answer in hex. [2 pt]

0x4005af

(D) Assume `main` calls `power(8,3)`. Fill in the snapshot of memory below the top of the stack **in hex** as this call to `power` returns to `main`. For unknown words, write "unknown". [6 pt]



0x7fffeca3f748	<ret addr to main>	1st frame
0x7fffeca3f740	<original rbx>	
0x7fffeca3f738	<ret addr to 4005af>	
0x7fffeca3f730	< 2nd rbx >	2nd frame
0x7fffeca3f728	< ret addr to 4005af>	
0x7fffeca3f720	< 3rd rbx >	3rd frame
0x7fffeca3f718	< ret addr to 4005af>	
0x7fffeca3f710		

(E) A student claims that we could have gotten away with not pushing a register onto the stack in `power`. Is he/she correct or not? Briefly explain. [3 pt]

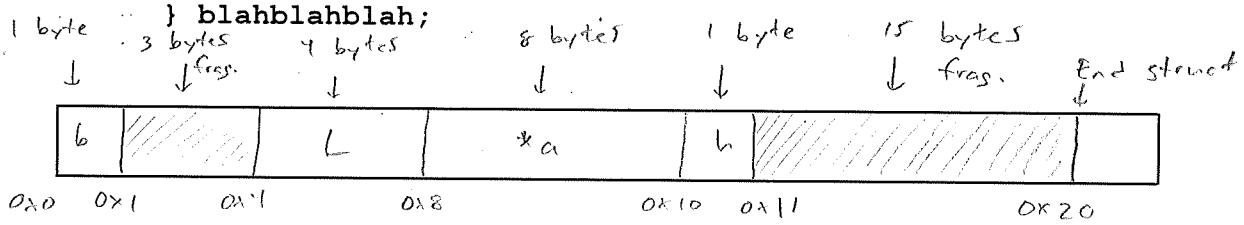
Yes - the value that %rbx holds does not change, so it can be kept track of by the original register that holds it.

6. Structs (10 points)

A. (2 pts) Draw a picture of the following struct, specifying the byte offset of each of the struct's fields and the size of any areas of fragmentation. Assume a 64-bit architecture.

```
typedef struct blah {
    char b;
    int l;
    char *a;
    char h;
} blahblahblah;
```

Has to end on address divisible by 8



B. (2 pts) How many bytes of internal fragmentation does the struct contain? External fragmentation?

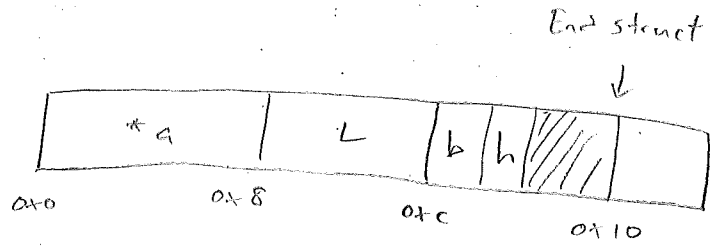
Full size = 32 bytes

Internal fragmentation: 3

External fragmentation: 15

C. (2 pts) Reorder the fields of the struct to minimize fragmentation:

```
typedef struct blah {
    char *a;
    int l;
    char b;
    char h;
} blahblahblah;
```



D. (2 pts) What is the size of the reordered struct?

16 bytes

E. (2 pts) How many bytes of internal fragmentation does the struct contain now? External?

Internal fragmentation: 0

External fragmentation: 2