Sp17 Midterm Q1

1. Integers and Floats (7 points)

- a. In the card game Schnapsen, 5 cards are used (Ace, Ten, King, Queen, and Jack) from 4 suits, so 20 cards in total. What are the minimum number of bits needed to represent a single card in a Schnapsen deck?
- b. How many <u>negative</u> numbers can we represent if given 7 bits and using two's complement?

Consider the following pseudocode (we've written out the bits instead of listing hex digits):

int a = 0b0100 0000 0000 0000 0000 0011 1100 0000 int b = (int)(float)a int m = 0b0100 0000 0000 0000 0001 0000 0000 int n = (int)(float)m

c. Circle one: True or False:

a == b

- d. Circle one: True or False:
 - m == n
- e. How many IEEE single precision floating point numbers are in the range [4, 6) (That is, how many floating point numbers are there where $4 \le x \le 6$?)

SID: _____

Au17 Final M3

Question M3: Pointers & Memory [8 pts]

leal 2(%rdi, %rsi), %eax

movw (%rdi,%rsi,4), %bx

For this problem we are using a 64-bit x86-64 machine (little endian). Below is the count_nz function disassembly, showing where the code is stored in memory.

0000000000	40053	86	<co< th=""><th>ount</th><th>t_nz</th><th>>:</th><th></th></co<>	ount	t_nz	>:	
400536:	85 f	6				testl	%esi,%esi
400538:	7e 1	b				jle	400555 <count_nz+0x1f></count_nz+0x1f>
40053a:	53					pushq	%rbx
40053b:	8b 1	f				movl	(%rdi),%ebx
40053d:	83 e	ee	01			subl	\$0x1,%esi
400540:	48 8	33	c7	04		addq	\$0x4,%rdi
400544:	e8 e	ed	ff	ff	ff	callq	400536 <count_nz></count_nz>
400549:	85 d	lb				testl	%ebx,%ebx
40054b:	0f 9	95	c2			setne	%dl
40054e:	0f b	6	d2			movzbl	%dl,%edx
400551:	01 d	10				addl	%edx,%eax
400553:	eb O	6				jmp	40055b <count_nz+0x25></count_nz+0x25>
400555:	b8 0	0	00	00	00	movl	\$0x0,%eax
40055a:	с3					retq	
40055b:	5b					popq	%rbx
40055c:	с3					retq	

(A) What are the values (in hex) stored in each register shown after the following x86 instructions are executed? Use the appropriate bit widths. <u>Hint</u>: what is the *value* stored in %rsi? [4 pt]

	Register	Value (hex)		
	%rdi	0x 0000 0000 0040 0544		
ĺ	%rsi	Ox FFFF FFFF FFFF FFFF		
ſ	%eax	0x		
Γ	%bx	0x		

(B) Complete the C code below to fulfill the behaviors described in the inline comments using pointer arithmetic. Let char* charP = 0x400544. [4 pt]

char v1 = *(charP + ___); // set v1 = 0xDB
int* v2 = (int*)((_____*)charP - 2); // set v2 = 0x400534

Au18 Midterm Q5

Question 5: Procedures & The Stack [24 pts]

The recursive function $sum_r()$ calculates the sum of the elements of an int array and its x86-64 disassembly is shown below:

```
int sum_r(int *ar, unsigned int len) {
    if (!len) {
        return 0;
    else
        return *ar + sum_r(ar+1,len-1);
}
```

0000000000	40050	7 <sum_r< th=""><th>:>:</th><th></th><th></th></sum_r<>	:>:		
400507:	41 5	3	1	pushq	%r12
400509:	85 f	6	١	testl	%esi,%esi
40050b:	75 0	7	:	jne	400514 <sum_r+0xd></sum_r+0xd>
40050d:	b8 0	0 00 00	00 1	movl	\$0x0,%eax
400512:	eb 11	2	:	jmp	400526 <sum_r+0x1f></sum_r+0x1f>
400514:	44 81	b lf	I	movl	(%rdi),%r12d
400517:	83 e	e 01	:	subl	\$0x1,%esi
40051a:	48 8	3 c7 04	ä	addq	\$0x4,%rdi
40051e:	e8 e	4 ff ff	ff (callq	400507 <sum_r></sum_r>
400523:	44 0	1 d8	ä	addl	%r12d,%eax
400526:	41 51	b	1	popq	%r12
400528:	с3		1	retq	

- (A) The addresses shown in the disassembly are all part of which section of memory? [2 pt]
- (B) Disassembly (as shown here) is different from assembly (as would be found in an assembly file). Name two major differences: [4 pt]

Difference 1:

Difference 2:

7

SID:

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

(D) What value is saved across each recursive call? Answer using a *C* expression. [2 pt]

(E) Assume main calls sum r(ar, 3) with int $ar[] = \{3, 5, 1\}$. Fill in the snapshot of memory below the top of the stack in hex as this call to sum r returns to main. For unknown words, write "0x unknown". [6 pt]

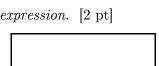
0x7ffffffde20	<ret addr="" main="" to=""></ret>
0x7ffffffde18	<original r12=""></original>
0x7fffffffde10	0x
0x7fffffffde08	0x
0x7fffffffde00	0x
0x7fffffffddf8	0x
0x7fffffffddf0	0x
0x7fffffffdde8	0x

(F) Assembly code sometimes uses *relative addressing*. The last 4 bytes of the callq instruction encode an integer (in *little endian*). This value represents the difference between which two addresses? Hint: both addresses are important to this callq. [4 pt]

	value (decimal):
0x	address 1:
0 x	address 2:

(G) What could we change in the assembly code of this function to reduce the amount of **Stack memory used** while keeping it *recursive* and *functioning properly*? [4 pt]

<original r12>



0x

Wi17 Final Q1 1. C and Assembly (15 points)

Consider the following (partially blank) x86-64 assembly, (partially blank) C code, and memory listing. Addresses and values are 64-bit, and the machine is little-endian. All the values in memory are in hex, and the address of each cell is the sum of the row and column headers: for example, address 0x1019 contains the value 0x18.

```
C code:
Assembly code:
  foo:
                                                 typedef struct person {
    movl $0, ____
                                                    char height;
                                                    char age;
  L1:
                                                   struct person* next_person;
    cmpq $0x0, %rdi
                                                 } person;
    je L2
    cmp ____, 0x1(%rdi)
                                                 int foo(person* p) {
                                                      int answer = ____;
while (_____) {
    je _
    mov 0x8(%rdi), %rdi
                                                          if (p->age == 24){
    jmp ____
                                                            answer = p->____;
  L2:
                                                            break;
                                                          }
    ret
                                                                            ___;
                                                          p =
  L3:
                                                      }
    mov (%rdi), %eax
                                                      return answer;
                                                 }
    jmp L2
```

Memory Listing	
Bits not shown are	0

	0x00	0x01	 0x05	0x06	0x07
0x1000	80	1B	 00	00	00
0x1008	80	1B	 00	00	00
0x1010	3F	18	 00	00	00
0x1018	3F	18	 00	00	00
0x1020	00	00	 00	00	00
0x1028	18	10	 00	00	00
0x1030	18	10	 00	00	00
0x1038	40	40	 00	00	00
0x1040	40	40	 00	00	00
0x1048	00	00	 00	00	00

(a) Given the code provided, fill in the blanks in the C and assembly code.

(b) Trace the execution of the call to foo((person*) 0x1028) in the table to the right. Show which instruction is executed in each step until foo returns. In each space, place the assembly instruction and the values of the appropriate registers after that instruction executes. You may leave those spots blank when the value does not change. You might not need all steps listed on the table.

Instruction	%rdi (hex)	%eax (decimal)
movl	0x1028	0
cmpq		
je		

(c) Briefly describe the value that foo returns and how it is computed. Use only variable names from the C version in your answer.

Au16 Final F5

Question F5: Caching [10 pts]

We have 16 KiB of RAM and two options for our cache. Both are two-way set associative with 256 B blocks, LRU replacement, and write-back policies. Cache A is size 1 KiB and Cache B is size 2 KiB.

(A) Calculate the TIO address breakdown for **Cache B**: [1.5 pt]

Tag bits	Index bits	Offset bits

(B) The code snippet below accesses an integer array. Calculate the Miss Rate for Cache A if it starts cold. [3 pt]

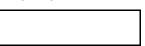
```
#define LEAP 4
#define ARRAY_SIZE 512
int nums[ARRAY_SIZE];  // &nums = 0x0100 (physical addr)
for (i = 0; i < ARRAY_SIZE; i+=LEAP)
    nums[i] = i*i;</pre>
```

(C) For each of the proposed (independent) changes, write MM for "higher miss rate", NC for "no change", or MH for "higher hit rate" to indicate the effect on Cache A for the code above:[3.5 pt]

 Direct-mapped
 Increase block size

 Double LEAP
 Write-through policy

(D) Assume it takes 200 ns to get a block of data from main memory. Assume Cache A has a hit time of 4 ns and a miss rate of 4% while Cache B, being larger, has a hit time of 6 ns. What is the worst miss rate Cache B can have in order to perform as well as Cache A? [2 pt]



Au17 Final F7

Question F7: Processes [9 pts]

(A) The following function prints out four numbers. In the following blanks, list three possible outcomes: [3 pt]

<pre>void concurrent(void) {</pre>
<pre>int x = 3, status;</pre>
if (fork()) {
if (fork() == 0) {
x += 2;
printf("%d",x);
} else {
<pre>wait(&status);</pre>
<pre>wait(&status);</pre>
x -= 2;
}
}
printf("%d",x);
exit(0);
}

(1) ______
(2) ______
(3) ______

(B) For the following examples of exception causes, write "N" for intentional or "U" for unintentional from the perspective of the user process. [2 pt]

System call

Segmentation fault

Mouse clicked

(C) Briefly define a **zombie** process. Name a process that can *reap* a zombie process. [2 pt]

Zombie process:	
Reaping process:	

(D) In the following blanks, write "Y" for yes or "N" for no if the following need to be updated when execv is run on a process. [2 pt]

Page table	PTBR	Stack	Code

Sp17 Final Q3 3. Virtual Memory (9 points)

Assume we have a virtual memory detailed as follows:

- 256 MiB Physical Address Space
- 4 GiB Virtual Address Space
- 1 KiB page size
- A TLB with 4 sets that is 8-way associative with LRU replacement

For the following questions it is fine to leave your answers as powers of 2.

a) How many bits will be used for:

Page offset? _____

Virtual Page Number (VPN)? _____ Physical Page Number (PPN)? _____

TLB index?

TLB tag? _____

b) How many entries in this page table?

c) We run the following code with an empty TLB. Calculate the TLB <u>miss</u> rate for data (ignore instruction fetches). Assume **i** and **sum** are stored in registers and **cool** is page-aligned.

```
#define LEAP 8
int cool[512];
... // Some code that assigns values into the array cool
... // Now flush the TLB. Start counting TLB miss rate from here.
int sum;
for (int i = 0; i < 512; i += LEAP) {
   sum += cool[i];
}</pre>
```

TLB <u>Miss</u> Rate: (fine to leave you answer as a fraction)

SID: _____

Au16 Final Q7 Question F7: Virtual Memory [10 pts]

Our system has the following setup:

- 24-bit virtual addresses and 512 KiB of RAM with 4 KiB pages
- A 4-entry TLB that is fully associative with LRU replacement
- A page table entry contains a valid bit and protection bits for read (R), write (W), execute (X)

(A) Compute the following values: [2 pt]

Page offset width _____ PPN width _____

Entries in a page table _____ TLBT width _____

(B) Briefly explain why we make the page size so much larger than a cache block size. [2 pt]

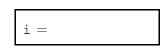
(C) Fill in the following blanks with "A" for always, "S" for sometimes, and "N" for never if the following get updated during a page fault. [2 pt]

 Page table
 Swap space
 TLB
 Cache

(D) The TLB is in the state shown when the following code is executed. Which iteration (value of i) will cause the protection fault (segfault)? Assume sum is stored in a register.
Recall: the hex representations for TLBT/PPN are padded as necessary. [4 pt]

```
long *p = 0x7F0000, sum = 0;
for (int i = 0; 1; i++) {
    if (i%2)
        *p = 0;
    else
        sum += *p;
    p++;
}
```

TLBT	PPN	Valid	R	W	Χ
0x7F0	0x31	1	1	1	0
0x7F2	0x15	1	1	0	0
0x004	0x1D	1	1	0	1
0x7F1	0x2D	1	1	0	0



Au16 Final Q8 Question F8: Memory Allocation [9 pts]

(A) Briefly describe one drawback and one benefit to using an *implicit* free list over an *explicit* free list. [4 pt]

Implicit benefit:	
	Implicit benefit:

(B) The table shown to the right shows the value of the header for the block returned by the request: (int*)malloc(N*sizeof(int))
 What is the alignment size for this dynamic memory allocator? [2 pt]

Ν	header value
6	33
8	49
10	49
12	65

- (C) Consider the C code shown here. Assume that the malloc call succeeds and foo is stored in memory (not just in a register). Fill in the following blanks with ">" or "<" to compare the values returned by the following expressions just before return 0. [3 pt]
 - ZERO _____ &ZERO
 - foo _____ &foo
 - foo _____ &str

```
#include <stdlib.h>
int ZERO = 0;
char* str = "cse351";
int main(int argc, char *argv[]) {
    int *foo = malloc(8);
    free(foo);
    return 0;
}
```

Name:__

Wi16 Final Q10

```
10. C vs. Java (11 points) Consider this Java code (left) and somewhat similar C code (right) running on x86-64:
```

```
public class Foo {
                               struct Foo {
  private int[] x;
                                  int x[6];
  private int y;
                                 int y;
  private int z;
                                  int z;
  private Bar b;
                                 struct Bar * b;
                               };
  public Foo() {
     x = null;
     b = null;
                               struct Foo * make_foo() {
  }
                                 struct Foo * f = (struct Foo *)malloc(sizeof(struct Foo));
}
                                 f \rightarrow x = NULL;
                                 f \rightarrow b = NULL;
                                 return f;
                               }
```

- (a) In Java, new Foo() allocates a new object on the heap. How many bytes would you expect this object to contain for holding Foo's fields? (Do not include space for any header information, vtable pointers, or allocator data.)
- (b) In C, malloc(sizeof(struct Foo)) allocates a new object on the heap. How many bytes would you expect this object to contain for holding struct Foo's fields? (Do *not* include space for any header information or allocator data.)
- (c) The function make_foo attempts to be a C variant of the Foo constructor in Java. One line fails to compile. Which one and why?
- (d) What, if anything, do we know about the values of the y and z fields after Java creates an instance of Foo?
- (e) What, if anything, do we know about the values of the y and z fields in the object returned by make_foo?

Au17 Final Q9

Question F9: Memory Allocation [9 pts]

(A) In a free list, what is a **footer** used for? Be specific. Why did we not need to use one in allocated blocks in Lab 5? [2 pt]

Footer:

Lab 5:

(B) We are designing a dynamic memory allocator for a 64-bit computer with 4-byte boundary tags and alignment size of 4 bytes. Assume a footer is always used. Answer the following questions: [4 pt]

 Maximum tags we can fit into the header (ignoring size):
 _______ tags

 Minimum block size if we implement an *explicit* free list:
 ______ bytes

 Maximum block size (leave as expression in powers of 2):
 ______ bytes

(C) Consider the C code shown here. Assume that the malloc call succeeds and foo is stored in memory (not just in a register). Fill in the following blanks with ">" or "<" to compare the values returned by the following expressions just before return 0. [3 pt]

&foo	 &ZERO
&str	 ZERO

&main _____ str

```
#include <stdlib.h>
int ZERO = 0;
char* str = "cse351";
int main(int argc, char *argv[]) {
    int *foo = malloc(8);
    free(foo);
    return 0;
}
```