University of Washington – Computer Science & Engineering

Autumn 2019 Instructor: Justin Hsia 2019-12-10

CSE351 FINAL

Last Name:		
First Name:		
Student ID Number:		
Name of person to your Left | Right		
All work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CSE351 who haven't taken it yet. Violation of these terms could result in a failing grade. (please sign)		

Do not turn the page until 12:30.

Instructions

- This exam contains 14 pages, including this cover page. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- The last page is a reference sheet. Please detach it from the rest of the exam.
- The exam is closed book (no laptops, tablets, wearable devices, or calculators). You are allowed two pages (US letter, double-sided) of *handwritten* notes.
- Please silence and put away all cell phones and other mobile or noise-making devices. Remove all hats, headphones, and watches.
- You have 110 minutes to complete this exam.

Advice

- Read questions carefully before starting. Skip questions that are taking a long time.
- Read *all* questions first and start where you feel the most confident.
- Relax. You are here to learn.

Question	M1	M2	M3	M4	M5	F6	F7	F8	F9	F10	Total
Possible Points	16	4	16	23	12	10	19	18	14	18	**150**

Question M1: Numbers [16 pts]

(A) Take the 32-bit numeral (*i.e.* bit pattern) **`0xFF800000`**. Circle the number representation below that has the *smallest magnitude* (*i.e.* closest to 0) for this numeral. [4 pt]

Floating Point   Two's Complement   Unsigned   Two's AND Unsigned

(B) What value will be read after we try to store $\mathbf{2^{-120} - 2^{-200}}$ in a `float`? (Circle one) [4 pt]

2^{-120} NaN   0 $2^{-120}–2^{-200}$

(C) Complete the following C function that returns the *signed* value of the exponent (not the `E` field) of a 32-bit floating point numeral (as an `unsigned int` argument as in Lab 1b). Ignore floating point special cases for this question. [4 pt]

```
int getExp(unsigned int fp) {

    return ________________________________________;
}
```

(D) Dubs claims that the expression **`(x != (float) x)`** will return True/1 if there was *data loss* during the cast of `int x`. Do you agree? *Briefly* explain why or why not. [4 pt]

Works? (circle one): Yes No
Explanation:

Question M2: Design Question [4 pts]

(A) Assume we decided to store/encode **object files** as *text* files instead of *binary* files. Name one advantage and one disadvantage of this design decision. [4 pt]

Advantage:
Disadvantage:

SID: ______________

Question M3: Pointers & Memory [16 pts]

Assume a 64-bit x86-64 machine (**little endian**). Below is the buggy `pcount_r` function disassembly from the midterm, *showing where the code is stored in memory.* Hint: read the questions before the assembly!

```
00000000004005d7 <pcount_r>:
  4005d7:  b8 00 00 00 00  movl   $0x0,%eax
  4005dc:  48 85 ff        testq  %rdi,%rdi
  4005df:  75 02           jne    4005e3 <pcount_r+0xc>
  4005e1:  f3 c3           repz retq
  4005e3:  48 d1 ef        shrq   $1, %rdi
  4005e6:  e8 ec ff ff ff  callq  4005d7 <pcount_r>
  4005eb:  83 e7 01        andl   $0x1,%edi
  4005ee:  48 01 f8        addq   %rdi,%rax
  4005f1:  c3              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. [8 pt]

```
leaq (%rdi,%rsi,2), %rax
addb 2(%rdi), %sil
```

Register	Value (hex)
%rdi	0x 0000 0000 0040 05e0
%rsi	0x 0000 0000 0000 0007
%rax	0x
%sil	0x

(B) Complete the C code below to fulfill the behaviors described in the inline comments using pointer arithmetic. Let **short* shortP = 0x4005e2**. [8 pt]

```
long* v1 = (long*)((____________*)shortP + 4);  // set v1 = 0x4005f2

short v2 = shortP[______];                       // set v2 = -1
```

Question M4: Procedures & The Stack [23 pts]

A *Caesar cipher* takes a string and shifts each character by the same amount (*i.e.* **`*str += shift;`** while `*str != '\0'`). For example, `"justin"` shifted by 2 becomes `"lwuvkp"`. Below is the disassembly for an *inefficient* recursive implementation `caesar` that returns the length of the string:

```
0000000000400547 <caesar>:
  400547:    0f b6 07          movzbl (%rdi),%eax      # get *str
  40054a:    84 c0             test   %al,%al
  40054c:    75 06             jne    400554 <caesar+0xd>
  40054e:    b8 00 00 00 00    mov    $0x0,%eax        # base case
  400553:    c3                retq                    #   returns 0
  400554:    53                push   %rbx
  400555:    8b 5c 24 10       mov    0x10(%rsp),%ebx  # get shift
  400559:    01 d8             add    %ebx,%eax
  40055b:    88 07             mov    %al,(%rdi)
  40055d:    48 83 c7 01       add    $0x1,%rdi        # next char
  400561:    53                push   %rbx
  400562:    e8 e0 ff ff ff    callq  400547 <caesar>
  400567:    83 c0 01          add    $0x1,%eax        # length += 1
  40056a:    5b                pop    %rbx
  40056b:    5b                pop    %rbx
  40056c:    c3                retq
```

(A) Which of the following *generates* the **labels** used in the disassembly above? Circle one. [2 pt]

Compiler   Assembler   Linker   Loader

(B) What is the return address to `caesar` that gets stored on the stack during a recursive call? Answer in hex. [2 pt]

0x

(C) Of the 16 instructions shown in the disassembly, how many of them access memory? [4 pt]

READ from memory: ______ instructions

WRITE to memory: ______ instructions

(D) *Briefly* explain the purpose of the push at `0x400554`. [2 pt]

(E) *Briefly* explain the purpose of the push at `0x400561`. Hint: what value are we pushing and where did we get it from? Read the comments! [2 pt]

(F) Assume `main` calls `caesar` on the string **`"cse"`** with a `shift` of **1**. Fill in the stack snapshot below (in hex) as this call returns to `main`. For unknown words, write "`0x unknown`". [8 pt]

Address	Value
0x7fffffffdcc8	<ret addr to main>
0x7fffffffdcc0	<original rbx>
0x7fffffffdcb8	0x
0x7fffffffdcb0	0x
0x7fffffffdca8	0x
0x7fffffffdca0	0x
0x7fffffffdc98	0x
0x7fffffffdc90	0x
0x7fffffffdc88	0x
0x7fffffffdc80	0x

(G) Name a way that we can reduce the memory usage of this function (either in amount of memory or number of memory accesses) *while maintaining correct behavior and keeping it recursive* and explain why the change helps. [3 pt]

Change:

Why it helps:

Question M5: C & Assembly [12 pts]

Answer the questions below about the following x86-64 assembly function, which *uses a struct* with two fields named **one** and **two**, declared in that order:

```
mystery:
        movl    $0, %eax        # Line 1
.L2:    testq   %rdi, %rdi      # Line 2
        je      .L5             # Line 3
        cmpb    %sil, (%rdi)    # Line 4
        je      .L1             # Line 5
        addl    $1, %eax        # Line 6
        movq    8(%rdi), %rdi   # Line 7
        jmp     .L2             # Line 8
.L5:    movl    $-1, %eax       # Line 9
.L1:    rep ret                 # Line 10
```

(A) `%rdi` contains a pointer to an instance of the struct. What **variable type** is field one? [2 pt]

____

(B) Based on Line 7, give a more intuitive name for the field `two` in the struct. [1 pt]

(C) This function fits into the following code skeleton. Fill in the corresponding parts below, using register names as variable names (*e.g.* `al` for the value in `%al`). None should be blank. Remember that the struct fields are named `one` and `two`. [9 pt]

```
int mystery( mysteryStruct* rdi, char sil ) {

   for ( ____________________; ____________________; ____________________ ) {

      if ( ____________________ ) {
         return eax;
      }

      ____________________;
   }
   return -1;
}
```

SID: ______________

Question F6: Structs [10 pts]

For this question, assume a 64-bit machine and the following C struct definition.

```
typedef struct {
  char* title;     // title (e.g. "HW SW INTERFACE")
  char  dept[3];   // dept (e.g. "CSE")
  short num;       // course number (e.g. 351)
  int   enrolled;  // students enrolled
} course;
```

(A) How much memory, in bytes, does an instance of `course` use? How many of those bytes are *internal* fragmentation and *external* fragmentation? [6 pt]

sizeof(course)	**Internal**	**External**

(B) Assume that an instance **`course`** `c` is allocated on the stack and an array **`char`** `ar[]` is allocated 40 bytes below `c` (*i.e.* `&ar + 0x28 == (char*)&c`). Fill in the blanks below with the new ASCII characters stored in `c.dept` after the following loop is executed. Hint: recall that the values 0x30 to 0x39 correspond to the ASCII characters `'0'` to `'9'`. [4 pt]

```
for (int i = 0; i < 52; ++i) {
   ar[i] = i;
}
```

`c.dept[0]:`	'___'
`c.dept[1]:`	'___'
`c.dept[2]:`	'___'

Question F7: Caching [19 pts]

We have 256 KiB of RAM and a 4-KiB L1 data cache that is 2-way set associative with 32-byte blocks and random replacement, write-back, and write allocate policies.

(A) Calculate the TIO address breakdown: [3 pt]

Tag bits	Index bits	Offset bits

(B) The code snippet below accesses two arrays of `doubles`. Assuming `i` is stored in a register and the cache starts *cold*, give the memory access pattern (read or write to which elements/addresses) and compute the **miss rate**. [6 pt]

```
#define SIZE 128
double src[SIZE];     // &src = 0x08000 (physical addr)
double dst[SIZE];     // &dst = 0x0E000 (physical addr)
for (int i = 0; i < SIZE; i += 1) {
    dst[i] = src[i];
    src[i] = i;
}
```

Per Iteration:	Access 1:	Access 2:	Access 3:
(circle) →	R / W to	R / W to	R / W to
(fill in) →	_______ [i]	_______ [i]	_______ [i]
			Code Miss Rate: _________

(C) For each of the proposed (independent) changes, draw ↑ for "increased", — for "no change", or ↓ for "decreased" to indicate the effect on the **miss rate from Part B** for the code above: [8 pt]

Use `float` instead ______ Double the cache size ______

Half the associativity ______ No-write allocate ______

(D) Assume it takes 160 ns to get a block of data from main memory. If our L1 data cache has a hit time of 5 ns and a miss rate of 5%, what is our average memory access time (AMAT)? [2 pt]

ns

SID: ____________

Question F8: Processes [18 pts]

(A) The following function prints out four numbers. In the following blanks, list three possible outcomes: [6 pt]

```
void concurrent(void) {
   int n = 5;
   if (fork()) {
      n++;
      if (fork()) {
        n++;
        wait();
      }
      printf("%d, ", n);
      exit(0);
   } else {
      printf("%d, ", n);
   }
   printf("%d, ", n);
   exit(0);
}
```

(1) ________________

(2) ________________

(3) ________________

(B) For the following examples of exception causes, write "**S**" for synchronous or "**A**" for asynchronous from the perspective of the user process. [4 pt]

System call ______ Divide by zero ______

Segmentation fault ______ Key pressed ______

(C) Fill in the following blanks with "**A**" for always, "**S**" for sometimes, and "**N**" for never if the following would be different when **context switching** to a *different* process? [4 pt]

Process ID _____ Program _____ PTBR _____ Condition Codes _____

(D) Is the following statement True or False? Provide a *brief* justification: a single process can execute multiple programs simultaneously. [4 pt]

Circle one: True / False

Justification:

Question F9: Virtual Memory [14 pts]

Our system has the following setup:

- 15-bit virtual addresses and 2 KiB of RAM with 256-byte pages
- A 4-entry fully-associative TLB with LRU replacement
- A PTE contains bits for valid (V), dirty (D), read (R), write (W), and execute (X)

(A) Compute the following values: [8 pt]

page offset width ________ # of TLB sets ________

# of virtual pages ________ minimum width of PTBR ________

(B) Assuming that the TLB is in the state shown (permission bits: 1 = allowed, 0 = disallowed), give example addresses that will fulfill the following scenarios: [6 pt]

TLBT	PPN	Valid	D	R	W	X
0x20	0xc	1	0	1	0	0
0x7f	0xa	1	0	1	1	0
0x7e	0xf	1	0	1	1	0
0x04	0xe	1	0	1	1	1

A value in `%rip` that causes a TLB Hit and no exception:	0x
A *write* address that causes a TLB Hit and segmentation fault:	0x

SID: ______________

Question F10: Memory Allocation [18 pts]

(A) In the following code, briefly identify the TWO memory issues and their fixes. [6 pt]

```
int N = 32;
long* func(long src[]) {
   long* p = (long*) malloc(N * sizeof(long));
   for (int i = 0; i < N; i++) {
      p[i] += src[i];
   }
}
```

Error 1:
Fix 1:
Error 2:
Fix 2:

(B) We are using a dynamic memory allocator on a **64-bit machine** with an **explicit free list**, **16-byte boundary tags**, and **8-byte alignment**. Assume that a footer is always used. [6 pt]

Request	block addr	return value	block size	internal fragmentation in this block
`p = malloc(9);`	`0x628`	`0x`_______	_______ bytes	_______ bytes

(C) Consider the C code shown here. Assume that the `malloc` call succeeds and that all variables are stored in memory (not registers). In the following groups of expressions, **circle the one** whose returned *value* (assume just before `return 0`) is **largest**. [6 pt]

```
#include <stdlib.h>
long glob = 10;
char* str = "351";

int main() {
    short* sp = malloc(8);
    int ONE = 1;
    free(sp);
    return 0;
}
```

Group 1:	`&sp`	`sp`	`&str`
Group 2:	`&glob`	`main`	`str`
Group 3:	`glob`	`ONE`	`*str`

End of Exam

Did you write your Student ID Number on the top-right corner of every odd page?

THIS PAGE PURPOSELY LEFT BLANK

CSE 351 Reference Sheet (Final)

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1024

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi

Sizes

C type	Suffix	Size
`char`	`b`	1
`short`	`w`	2
`int`	`l`	4
`long`	`q`	8

IEEE 754 FLOATING-POINT STANDARD

Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$

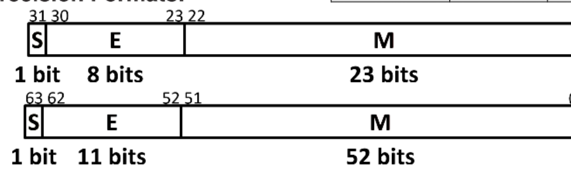
Bit fields: $(-1)^S \times 1.M \times 2^{(E-\text{bias})}$

where Single Precision Bias = 127, Double Precision Bias = 1023.

IEEE 754 Symbols

E	M	Meaning
all zeros	all zeros	± 0
all zeros	non-zero	± denorm num
1 to MAX-1	anything	± norm num
all ones	all zeros	± ∞
all ones	non-zero	NaN

IEEE Single Precision and Double Precision Formats:



Assembly Instructions

`mov a, b`	Copy from `a` to `b`.
`movs a, b`	Copy from `a` to `b` with sign extension. Needs *two* width specifiers.
`movz a, b`	Copy from `a` to `b` with zero extension. Needs *two* width specifiers.
`lea a, b`	Compute address and store in `b`. *Note:* the scaling parameter of memory operands can only be 1, 2, 4, or 8.
`push src`	Push `src` onto the stack and decrement stack pointer.
`pop dst`	Pop from the stack into `dst` and increment stack pointer.
`call <func>`	Push return address onto stack and jump to a procedure.
`ret`	Pop return address and jump there.
`add a, b`	Add from `a` to `b` and store in `b` (and sets flags).
`sub a, b`	Subtract `a` from `b` (compute `b-a`) and store in `b` (and sets flags).
`imul a, b`	Multiply `a` and `b` and store in `b` (and sets flags).
`and a, b`	Bitwise AND of `a` and `b`, store in `b` (and sets flags).
`sar a, b`	Shift value of `b` *right* (*arithmetic*) by `a` bits, store in `b` (and sets flags).
`shr a, b`	Shift value of `b` *right* (*logical*) by `a` bits, store in `b` (and sets flags).
`shl a, b`	Shift value of `b` *left* by `a` bits, store in `b` (and sets flags).
`cmp a, b`	Compare `b` with `a` (compute `b-a` and set condition codes based on result).
`test a, b`	Bitwise AND of `a` and `b` and set condition codes based on result.
`jmp <label>`	Unconditional jump to address.
`j* <label>`	Conditional jump based on condition codes (*more on next page*).
`set* a`	Set byte based on condition codes.

Conditionals

Instruction			(op) s, d	test a, b	cmp a, b
je	sete	"Equal"	d (op) s == 0	b & a == 0	b == a
jne	setne	"Not equal"	d (op) s != 0	b & a != 0	b != a
js	sets	"Sign" (negative)	d (op) s < 0	b & a < 0	b-a < 0
jns	setns	(non-negative)	d (op) s >= 0	b & a >= 0	b-a >= 0
jg	setg	"Greater"	d (op) s > 0	b & a > 0	b > a
jge	setge	"Greater or equal"	d (op) s >= 0	b & a >= 0	b >= a
jl	setl	"Less"	d (op) s < 0	b & a < 0	b < a
jle	setle	"Less or equal"	d (op) s <= 0	b & a <= 0	b <= a
ja	seta	"Above" (unsigned >)	d (op) s > 0U	b & a > 0U	b-a > 0U
jb	setb	"Below" (unsigned <)	d (op) s < 0U	b & a < 0U	b-a < 0U

Registers

Name	Convention	Name of "virtual" register		
		Lowest 4 bytes	Lowest 2 bytes	Lowest byte
%rax	Return value – Caller saved	%eax	%ax	%al
%rbx	Callee saved	%ebx	%bx	%bl
%rcx	Argument #4 – Caller saved	%ecx	%cx	%cl
%rdx	Argument #3 – Caller saved	%edx	%dx	%dl
%rsi	Argument #2 – Caller saved	%esi	%si	%sil
%rdi	Argument #1 – Caller saved	%edi	%di	%dil
%rsp	Stack Pointer	%esp	%sp	%spl
%rbp	Callee saved	%ebp	%bp	%bpl
%r8	Argument #5 – Caller saved	%r8d	%r8w	%r8b
%r9	Argument #6 – Caller saved	%r9d	%r9w	%r9b
%r10	Caller saved	%r10d	%r10w	%r10b
%r11	Caller saved	%r11d	%r11w	%r11b
%r12	Callee saved	%r12d	%r12w	%r12b
%r13	Callee saved	%r13d	%r13w	%r13b
%r14	Callee saved	%r14d	%r14w	%r14b
%r15	Callee saved	%r15d	%r15w	%r15b

C Functions

void* `malloc(`**size_t** `size):`
Allocate size bytes from the heap.

void* `calloc(`**size_t** `n,` **size_t** `size):`
Allocate n*size bytes and initialize to 0.

void `free(`**void*** `ptr):`
Free the memory space pointed to by ptr.

size_t `sizeof(`**type**`):`
Returns the size of a given type (in bytes).

char* `gets(`**char*** `s):`
Reads a line from stdin into the buffer.

pid_t `fork():`
Create a new child process (duplicates parent).

pid_t `wait(`**int*** `status):`
Blocks calling process until any child process exits.

int `execv(`**char*** `path,` **char*** `argv[]):`
Replace current process image with new image.

Virtual Memory Acronyms

MMU	Memory Management Unit	**VPO**	Virtual Page Offset	**TLBT**	TLB Tag
VA	Virtual Address	**PPO**	Physical Page Offset	**TLBI**	TLB Index
PA	Physical Address	**PT**	Page Table	**CT**	Cache Tag
VPN	Virtual Page Number	**PTE**	Page Table Entry	**CI**	Cache Index
PPN	Physical Page Number	**PTBR**	Page Table Base Register	**CO**	Cache Offset