University of Washington – Computer Science & Engineering

Autumn 2019 Instructor: Justin Hsia 2019-10-28

CSE351 MIDTERM

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 5:30.

Instructions

- This exam contains 5 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 one page (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 70 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	1	2	3	4	5	Total
Possible Points	20	20	12	24	24	**100**

Question 1: Number Representation [20 pts]

(A) Convert the decimal number **-12** into *5-bit* two's complement. Answer in binary. [2 pt]

`0b`

(B) If **`signed char a = 0x88`**, complete the *bitwise* C statement so that **`b = 0xF1`**. The first blank should be an operator and the second blank should be a numeral. [4 pt]

`b = a ___ 0x_____`

(C) Find the *largest 8-bit unsigned numeral* `c` (answer in hex) such that **`c + 0x80`** causes NEITHER *signed* nor *unsigned* overflow in 8 bits. [4 pt]

`0x`

For the rest of this problem we are working with a floating point representation that follows the same conventions as IEEE 754 except using 8 bits split into the following fields:

Sign (1)	Exponent (5)	Mantissa (2)

(D) What is the *magnitude* of the **bias** of this new representation? [2 pt]

(E) What is the decimal value encoded by **`0b 1100 1001`** in this representation? [4 pt]

(F) What is the **smallest positive integer** that *can't be represented* in this floating point encoding scheme? Hint: for what integer will the "one's digit" get rounded? [4 pt]

SID: _____

Question 2: Pointers & Memory [20 pts]

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	20	F6	EF	EA	A2	5E	9F	1A
0x08	A2	D0	4F	C4	A0	0C	F7	27
0x10	B8	BD	1A	CA	35	95	CB	80
0x18	84	3F	02	4F	8E	F3	F6	E5
0x20	CD	4A	F6	48	1A	6F	7E	63

```
char*  charP  = 0xD;
short* shortP = 0x1E;
```

(A) Using the values shown above, fill in the C type and hex value for each of the following C expressions. Leading zeros are not required for the hex values. [8 pt]

C Expression	C Type	Hex Value
`*(charP + 6)`		0x
`(int**)shortP - 2`		0x

(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 data widths.* [12 pt]

```
leal   2(,%rsi,8),     %r10d
movswq (%rdi,%rdi,2),  %r11
cmpb   0x19(%rsi),     %dil
```

Register	Data (hex)
%rdi	0x 0000 0000 0000 0004
%rsi	0x 0000 0000 0000 0001
%r10d	0x
%r11	0x
%dil	0x

Question 3: Design Questions [12 pts]

Answer the following questions in the boxes provided with a **single sentence fragment**. Please try to write as legibly as possible.

(A) While the floating point special cases seem arbitrary, there is a method to the madness. *Briefly* describe why the following choices were made: [4 pt]

The `E` encoding for ∞:
Multiple encodings for NaN:

(B) When we cast between an integer data type and a floating point one, the conversion is done by encoding the original *value*, changing the stored bits. Imagine if we instead did the conversion by leaving the bits the same, but interpreting them differently. Name an advantage and a disadvantage of this change. [4 pt]

Advantage:
Disadvantage:

(C) Assume we have an address space of $\mathbf{2^w}$ **bytes**. If we decided to **assign an address to every 4 bits** instead of every byte, what is the new width of an address? Also, name one change we would need to make to the existing x86-64 instruction syntax. [4 pt]

New address width: _____ bits

Instruction syntax change:

SID: _____

Question 4: C & Assembly [24 pts]

Answer the questions below about the following x86-64 assembly function:

```
mystery:
        jmp     .L2                 # Line 1
.L4:    addq    $1, %rdi            # Line 2
        movb    %al, (%rsi)         # Line 3
        leaq    1(%rsi), %rsi       # Line 4
.L2:    movzbl  (%rdi), %eax        # Line 5
        testb   %al, %al            # Line 6
        je      .L3                 # Line 7
        cmpb    %dl, %al            # Line 8
        jne     .L4                 # Line 9
.L3:    movb    $0, (%rsi)          # Line 10
        retq                        # Line 11
```

(A) What **variable type** would `%rdi` be in the corresponding C program? [4 pt]

_____

(B) What **variable type** would the 3rd argument be in the corresponding C program? [4 pt]

_____

(C) This function uses a `while` loop. Fill in the two conditionals below, using register names as variable names (no declarations necessary). [8 pt]

```
while ( _____ && _____ )
```

(D) Taking the variable types into account, describe at a high level what the *purpose* of Line 10 is (not just what it does mechanically). [4 pt]

(E) Describe at a high level what you think this function *accomplishes* (not line-by-line). [4 pt]

Question 5: Procedures & The Stack [24 pts]

The recursive function `pcount_r()` from lecture calculates the "popcount" of `x`, returning the number of 1's in the unsigned binary representation. However, a faulty compiler has produced the **BUGGY** x86-64 disassembly shown below:

CORRECT

```
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    return (x & 1) + pcount_r(x >> 1);
}
```

BUGGY

```
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 is the address of the code that comes after the *function* `pcount_r` (not the label) in memory? [2 pt]

0x

(B) Circle one: The variable `x` will show up in which table(s) in the object file? [2 pt]

Symbol Table **Relocation Table** **Both Tables** **Neither Table**

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

0x

(D) To see what's going wrong with this implementation, trace the execution for `pcount_r(1)`. What are the expected and actual return values? [4 pt]

Expected: Actual:

(E) Assume `main` calls our buggy `pcount_r(5)`. Fill in the snapshot of memory below the top of the stack **in hex** as this call to `pcount_r` returns to `main`. For unknown words, write "`0x unknown`". [4 pt]

0x7fffffffdc98	<ret addr to main>
0x7fffffffdc90	0x
0x7fffffffdc88	0x
0x7fffffffdc80	0x
0x7fffffffdc78	0x
0x7fffffffdc70	0x

(F) Now let's go about fixing the assembly code. During your execution trace in part D, think about when you encountered a value that you didn't expect. Which register did this happen to? As a *C expression*, what value was supposed to be held in this register? Is this caller- or callee-saved? [4 pt]

Register: _____ Expression: _____ Type: call___-saved

(G) Now we need to add a `pushq` and `popq` instruction for the register you identified above. Pay attention to the register saving convention you identified above. Give the addresses of the instruction you would place the new instruction *just before*. For example, write "`0x 4005e1`" if you wanted to place an instruction just before the `repz retq`. [6 pt]

`pushq` address:	0x
`popq` address:	0x

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 (Midterm)

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

IEEE 754 FLOATING-POINT STANDARD

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

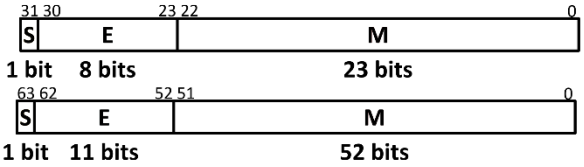
Bit fields: $(-1)^S \times 1.M \times 2^{(E-\text{bias})}$

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 `a` to 0 or 1 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

Sizes

C type	x86-64 suffix	Size (bytes)
char	b	1
short	w	2
int	l	4
long	q	8