

CSE 351 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 17:30.

Instructions

- This exam contains 6 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). You are allowed one page (US letter/A4, double-sided) of handwritten notes. Scientific calculators are allowed.
- 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. You can do it! 🍀

Question	1	2	3	4	5	Total
Possible Points	20	20	20	20	20	100

Question 1: Number Representation [20 pts]

(A) Given the following variable declarations:

```
unsigned int x;
unsigned int y;
float g;
float h;
signed int s;
```

CIRCLE ONE of the options for each statement below:

a) $(x + y) \geq x$	Always	Sometimes	Never
b) If g and h are both positive, then $(g + h) \geq g$	Always	Sometimes	Never
c) $(x y) \geq x$	Always	Sometimes	Never
d) $((\text{unsigned int}) s) \gg 2 > (\text{unsigned int}) (s \gg 2)$	Always	Sometimes	Never
e) $(\sim s + 1) > (-1 * s)$	Always	Sometimes	Never
f) If $s < 0$, then $(s 1) \leq (s \gg 31)$	Always	Sometimes	Never

(B) Consider the following small 8-bit floating-point encoding scheme:

S (1 bit)	E (3 bits)	M (4 bits)
-----------	------------	------------

a) What is the bias for this encoding?

b) Given a floating point number $x = 10.5$ encoded in the above scheme:

$x =$

0	110	0101
---	-----	------

Give an encoding for a **normalized** (not a special-case encoding) floating point number y such that $x + y$ results in **rounding**:

$y =$

--	--	--

Question 2: Pointers and Memory [20 pts]

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

	Word Addr	+0	+1	+2	+3	+4	+5	+6	+7
<pre>// num is shown in memory char* str = 0xD1; short num = 0x15F; int* arr[2];</pre>	0xC0	D1	00	01	5F	64	18	12	C4
	0xC8	82	55	72	AB	CE	0A	B1	00
	0xD0	77	73	6E	6F	77	00	79	61
	0xD8	5F	01	90	71	00	F5	1F	D2
	0xE0	89	9C	10	F5	6B	3C	F2	10

(A) How many bytes are allocated by the declarations and initializations in the code above?

 bytes

(B) What is the result of `printf("%s", str)`, printing `str` as a string? A partial ASCII chart is provided for reference.

Hex	0x6E	0x6F	0x70	0x71	0x72	0x73	0x74	0x75	0x76	0x77	0x78	0x79	0x7A
Char	n	o	p	q	r	s	t	u	v	w	x	y	z

(C) Consider the following C expressions. What is their **C type** and (hexadecimal) **value**?

Expression	C Type	Hex Value
<code>str + 5</code>		
<code>&num - 2</code>		
<code>num - 3</code>		

Question 3: Design Questions [20 pts]

(A) Suppose x86-64 is extended to be compatible with a new primitive data type called `longer`, which takes up **128 bits**.

- a) List all the possible values of `S` in the x86-64 memory operand `D(Rb, Ri, S)` after the addition of `longer`.

- b) In 1-2 sentences, explain the purpose of `S` in the x86-64 memory operand. Why are the above values of `S` useful?

Explanation:

(B) Jorge wants to make an encoding scheme for a candy jar. It should store:

- The number of pieces of candy currently in the jar (`numPieces` field).
- A signed value representing how many pieces of candy were added or removed the last time someone accessed the jar (`lastAccess` field).

- a) If a jar needs to be able to hold at least 50 pieces of candy, what is the minimum number of bits required for the `numPieces` field?

- b) If Jorge wants to represent changes in candy from -7 to +7 pieces, what is the minimum number of bits required for the `lastAccess` field?

- c) Occasionally, Jorge's mom likes to use jars to store her button collection. What can we add to this encoding scheme to tell us whether a jar is being used to store candy or buttons? (1-2 sentences)

Change:

Question 4: C & Assembly [20 pts]

```
terminator:
    movq    %rdi,%rax           # Line 1
    testl   %esi,%esi          # Line 2
    jle     .L2                 # Line 3
    movl    $0x0, %ecx         # Line 4
.L1:      movslq   %ecx,%rcx     # Line 5
    cmpb    $0x0, (%rdi,rcx,1) # Line 6
    je      .L3                 # Line 7
    addl    $0x1,%ecx         # Line 8
    cmpl    %ecx,%esi         # Line 9
    jg      .L1                 # Line 10
.L2:      movslq   %esi,%rsi     # Line 11
    movb    $0x0, (%rdi,%rsi,1) # Line 12
.L3:      ret                  # Line 13
```

(A) Fill in the missing C code that is equivalent to the x86-64 assembly above:

```
_____ terminator( _____ d, _____ s) {
    for (int c = 0; c < _____; c++) {
        if ( *(_____ + _____) == 0x00) {
            return d;
        }
    }
    _____[_____] = 0x00;
    return d;
}
```

(B) Rewrite line 2 (shaded) with a different assembly instruction (not test!) that has the same effect as the original.

```
Line 2:  testl    %esi,%esi    →
```

(C) For the following instructions, write an equivalent line(s) of assembly using the replacement instruction(s) suggested.

a) `imulq $5, %rax`

`leaq _____ (_____, _____, _____), _____`

b) `pushq %rbx`

`subq _____, _____`

`movq _____, _____`

(D) Re-order the scrambled instructions so that they swap the values of registers `%rax` and `%rbx`.

`pop %rbx` _____

`push %rax` _____

`push %rbx` _____

`pop %rax` _____

Question 5: Procedures & The Stack [20 pts]

Consider the following C code, which is used to compute a fibonacci sequence:

```
int fib(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fib(n-1) + fib(n-2);
    }
}

char* call_fib(int num) {
    int count = fib(num);
    if (count % 2 == 0) {
        return "even";
    }
    return "odd";
}

int main() {
    int num = 4;
    char* result = call_fib(num);
    printf("%s\n", result);
}
```

Also consider the full disassembly for fib:

```
0000000000401126 <fib>:
 401126: 53                push   %rbx
 401127: 48 83 ec 10       sub    $0x10,%rsp
 40112b: 89 7c 24 0c       mov    %edi,0xc(%rsp)
 40112f: 83 7c 24 0c 01    cmpl  $0x1,0xc(%rsp)
 401134: 7f 06            jg     40113c <fib+0x16>
 401136: 8b 44 24 0c       mov    0xc(%rsp),%eax
 40113a: eb 20            jmp   40115c <fib+0x36>
 40113c: 8b 44 24 0c       mov    0xc(%rsp),%eax
 401140: 83 e8 01         sub    $0x1,%eax
 401143: 89 c7            mov    %eax,%edi
 401145: e8 dc ff ff ff   call  401126 <fib>
 40114a: 89 c3            mov    %eax,%ebx
 40114c: 8b 44 24 0c       mov    0xc(%rsp),%eax
 401150: 83 e8 02         sub    $0x2,%eax
 401153: 89 c7            mov    %eax,%edi
 401155: e8 cc ff ff ff   call  401126 <fib>
 40115a: 01 d8            add    %ebx,%eax
 40115c: 48 83 c4 10       add    $0x10,%rsp
 401160: 5b              pop    %rbx
 401161: c3              ret
```

And partial disassemblies for `call_fib` and `main`:

```
0000000000401162 <call_fib>:
 401162: 48 83 ec 28          sub    $0x28,%rsp
 401166: 89 7c 24 0c          mov    %edi,0xc(%rsp)
 40116a: 8b 44 24 0c          mov    0xc(%rsp),%eax
 40116e: 89 c7                mov    %eax,%edi
 401170: e8 b1 ff ff ff      call  401126 <fib>
...
 401190: 48 83 c4 28          add    $0x28,%rsp
 401194: c3                  ret
```

```
0000000000401195 <main>:
 401195: 48 83 ec 18          sub    $0x18,%rsp
 401199: c7 44 24 0c 04 00 00  movl  $0x4,0xc(%rsp)
 4011a0: 00
 4011a1: 8b 44 24 0c          mov    0xc(%rsp),%eax
 4011a5: 89 c7                mov    %eax,%edi
 4011a7: e8 b6 ff ff ff      call  401162 <call_fib>
...
 4011c1: 48 83 c4 18          add    $0x18,%rsp
 4011c5: c3                  ret
```

(A) If `main` is executed as it is written, with a call to `call_fib(4)`, how many total stack frames are created (including `main`)?

(B) What is the **maximum depth** of the stack frames generated for this program?

(C) What address(es) are saved on the stack for the recursive calls of `fib`?

(D) How large is `fib(4)`'s stack frame right before it makes its first recursive call to `recur(3)` on line 401145?

(E) How large (in bytes) is `call_fib`'s stack frame right before it makes the call to `fib`?

(F) What is the address of the immediate `$0x4` within `main`'s disassembly?

(G) In which region of memory are the two possible return values of `call_fib` stored?

stack heap static data literals instructions

(H) If we wanted to use `%r10d` instead of `%ebx` in `fib`, how else would the assembly code need to change to follow proper x86-64 conventions? Your response should be no more than 4 sentences.

Changes:

This page is purposely left blank.