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) >= x`	Always	[✓ Sometimes]	Never
b) If `g` and `h` are both positive, then `(g + h) >= g`	[✓ Always]	Sometimes	Never
c) `(x | y) >= x`	[✓ Always]	Sometimes	Never
d) `(((unsigned int) s) >> 2) > (unsigned int) (s >> 2)`	Always	Sometimes	[✓ Never]
e) `(~s + 1) > (-1 * s)`	Always	Sometimes	[✓ Never]
f) If `s < 0`, then `(s || 1) <= (s >> 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? 3

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

x =	0	100	0101

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

y =		001	Anything

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:

```
// num is shown in memory
char* str = 0xD1;
short num = 0x15F;
int* arr[2];
```

Word Addr	+0	+1	+2	+3	+4	+5	+6	+7
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?

26 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

snow

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

Expression	**C Type**	**Hex Value**
`str + 5`	char*	0xD6
`&num - 2`	short*	0xD4
`num - 3`	short	0x15C

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`.

1,2,4,8,16

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:

1.) S is used for correct pointer arithmetic.
2.) Partial credit for mentioning regular arithmetic

(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?

6 bits

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?

4 bits

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:

Add a (1-bit) sign bit field to indicate a jar being used for candy or buttons.

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
        jl       .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:

```
__char*__ terminator( __char*__ d, __int__ s)  {
     for (int c = 0; c <= __s__; c++)  {
          if ( *(__d__ + __c__) == 0x00) {
               return d;
          }
     }
     __d__[__s__] = 0x00;
     return d;
}
```

(B) Rewrite **both** lines 2 and 3 with different assembly instructions (no `test` or `jle`!) that have the same effect as the originals.

Line 2:	`testl    %esi,%esi`	➜	cmpl $esi, $0
Line 3:	`jle      .L2`	➜	jge .L2

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

a) `imulq $5, %rax`

`leaq` __0x0__ (__%rax__, __%rax__, ___4___), ___%rax___

b) `pushq %rbx`

`subq` ___$8___ , ___%rsp____

`movq` ___%rbx___ , __(%rsp)___

c) re-order the scrambled instructions so that they swap the values of registers %rax and %rbx

`pop %rbx`	____push %rax______
`push %rax`	____push %rbx_______
`push %rbx`	____pop %rax________
`pop %rax`	____pop %rbx_________

Alternatively:

push %rbx
push %rax
pop %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`)?

12 frames

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

6 frames

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

`0x40114a & 0x40115a`

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

32 bytes

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

48 bytes

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

0x40119d

(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:

1. Pushing and popping need to be shifted to be around both call commands
2. Replace all instances of %ebx/%rbx with $r10d/$r10