Number Representation & Strings

A. What is the value of the `signed char` **`0x9E`** in decimal?

-128+16+8+4+2 = -98

B. What is the value of the `unsigned char` **`37`** in binary?

0b00100101

C. If **`a = 0x2C,`** complete the *bitwise* C statement so that **`b = 0x1F.`**

```
  0b 0010 1100
^ 0b 0011 0011
  ------------
  0b 0001 1111
```

```
b = a ^ 0x33
```

For the following problems we are working with a floating point representation that follows the same conventions as IEEE 754 except using 7 bits split into the following fields:

Sign (1)	Exponent (3)	Mantissa (3)

D. What is the *magnitude* of the **bias** of this new representation?

$2^{3-1}-1 = 3$

E. What is the decimal value encoded by **`0b1110101`** in this representation?

S = 1, E = 0b110 = 6, M = 0b101

Value = $(-1)^1 \times 1.101_2 \times 2^{6-3} = -1.101 \times 2^3 = -1101_2 = -13$

F. What value will be read after we try to store **-18** in this representation? (Circle one)

-16 -NaN -∞ -18

$-18 = -(16 + 2) = -(2^4+2^1) = -1.001_2 \times 2^4$. Largest normalized exponent we can encode is 0b110, which gives Exp = 3. As a result this causes overflow, resulting in -∞ being stored (as 0b1111000)

For the following problem, assume we are working with C strings encoded in ASCII. Consider the declaration:

```
char str[] = "Hello!";
```

G. What will be stored in the array str? (Bytes given in hex)

48	65	6C	6C	6F	21	0

Pointers & Memory

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:

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

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

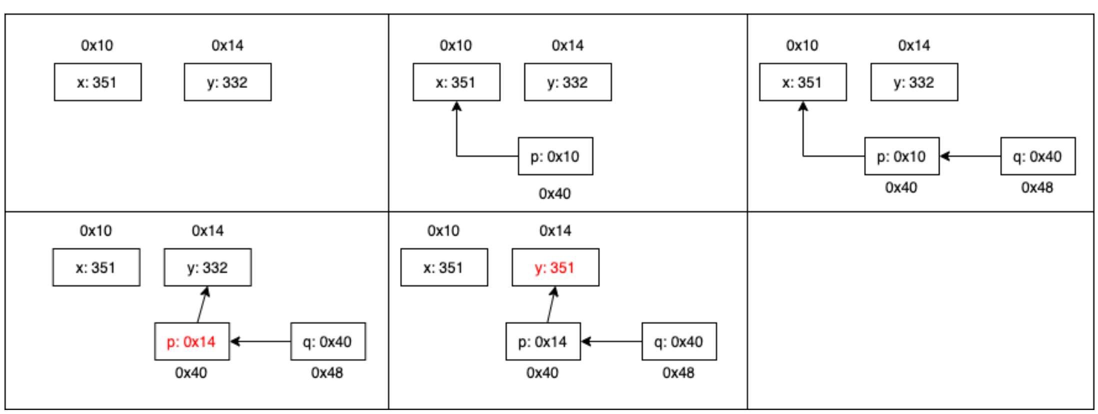
A. Using the values shown above, fill in the C type and hex value for each of the following C expressions. Leading zeros are not required for the hex values.

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

charP: 0xD + 6 (scaled by sizeof(char) = 1) yields 0x13. Address 0x13 holds the char 0xCA.
shortP: 0x1E – 2 (scaled by sizeof(int*) = 8) yields 0xE.

B. For the following snippet of C code, draw out a box-and-arrow diagram for the allocated memory.

```
int x = 351, y = 332;
int *p = &x;
int **q = &p;
*q = &y;
*(*q) = x;
```



C & Assembly

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?
   `char*, unsigned char*` is also acceptable due to zero-extension.
   Line 5: we read a byte out of memory by dereferencing the value in %rdi

B. What **variable type** would the third argument be in the corresponding C program?
   `char`
   Line 8: %dl (lowest byte of %rdx) is compared to the byte read out of memory.

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

```
                    al
                 al != 0
                   *rdi              al != dl
while ( _*rdi != 0__ && _*rdi != dl_ )
```

   Conditional 1 is from Lines 6-7, which exit the loop if `%al = 0`.
   Conditional 2 is from Lines 8-9, which loop back if `%al - %dl != 0`.

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).
   Adds a null terminator (char with value 0) to the end of *rsi (the destination string).

E. Describe at a high level what you think this function *accomplishes* (not line-by-line).
   It copies all of the characters from a source string (in %rdi) to a destination string (in %rsi) until it sees a specified character (in %dl) or the end of the source string. The destination String is then null-terminated.

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);
}
```

```
0000000000400507 <sum_r>:
  400507:  41 53           pushq  %r12
  400509:  85 f6           testl  %esi,%esi
  40050b:  75 07           jne    400514 <sum_r+0xd>
  40050d:  b8 00 00 00 00  movl   $0x0,%eax
  400512:  eb 12           jmp    400526 <sum_r+0x1f>
  400514:  44 8b 1f        movl   (%rdi),%r12d
  400517:  83 ee 01        subl   $0x1,%esi
  40051a:  48 83 c7 04     addq   $0x4,%rdi
  40051e:  e8 e4 ff ff ff  callq  400507 <sum_r>
  400523:  44 01 d8        addl   %r12d,%eax
  400526:  41 5b           popq   %r12
  400528:  c3              retq
```

(A) The addresses shown in the disassembly are all part of which section of memory? [2 pt]

Text or `.text` also accepted. **Instructions/Code**

(B) *Disassembly* (as shown here) is different from *assembly* (as would be found in an assembly file). Name two major differences: [4 pt]

Differences: Some possible answers include:

- No machine code (middle column) would be shown in the assembly (*i.e.* the code hasn't been assembled yet).
- Finalized addresses would not be found in the assembly (left column).
- All labels would still be symbolic/named in the assembly instructions (*e.g.* `jne`, `jmp`, `callq`).

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

The address of the instruction *after* `call`.

0x 400523

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

The instruction at address `0x400514` dereferences `%rdi` and stores the value in `%r12d`.

`*ar`

(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]

0x7fffffffde20	<ret addr to main>	sum_r(ar,3)
0x7fffffffde18	<original r12>	
0x7fffffffde10	0x **400523 <ret addr>**	sum_r(ar+1,2)
0x7fffffffde08	0x **3 <*ar>**	
0x7fffffffde00	0x **400523 <ret addr>**	sum_r(ar+2,1)
0x7fffffffddf8	0x **5 <*ar>**	
0x7fffffffddf0	0x **400523 <ret addr>**	sum_r(ar+3,0)
0x7fffffffdde8	0x **1 <*ar>**	

The base case DOES still push `%r12` onto the stack.

(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]

`0xffffffe4` = `-(0x1b + 1)` = `-28` — value (decimal): **-28**

This corresponds to the address we jump to. — address 1: 0x **400507**

This corresponds to the return address. — address 2: 0x **400523**

(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]

The issue with recursive functions is that no matter what kind of register you use to save a value (caller-saved or callee-saved), the recursive call will overwrite that value because it's an identical function! So we actually *can't* avoid pushing something to the stack without making the function iterative. So any potential saving of Stack space will come from the base case. Keep reading for two possible solution types:

Callee-saved: `%r12` is a *callee*-saved register. This means that its old value just needs to be saved before we overwrite its value; it does not need to be saved at the very top of `sum_r`.

1) Move the `pushq` instruction into the recursive case (below the `jmp` instruction).
2) Either make the `jmp` go to address `0x400528` instead OR
move the `movl $0,%eax` above the `jne` and change the `jne` to `je 0x400528`.

Caller-saved: The value we really care about saving across the recursive call (`ar` or `*ar`), already starts in a caller-saved register in `%rdi`! This value must then be saved before we make a recursive call to `sum_r` and restored once it returns:

1) Convert the `pushq %r12` to `pushq %rdi` and move it down to *replace* the `movl (%rdi),%r12d` instruction.
2) Convert the `popq %r12` to `popq %rdi` and move it right after/below the `callq`.
3) Convert the `addl %r12d,%eax` to `addl (%rdi),%eax`.

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) What values can `S` take in an x86-64 memory operand? *Briefly* describe why these choices are useful/important. [4 pt] – a memory operand is of the form `D(Rb,Ri,S)`.

Values: 1, 2, 4, 8

Importance: These values represent the different scaling factors used in pointer arithmetic based on the data type sizes.

(B) Until very recently (Java 8/9), Java did not support *unsigned* integer data types. Name one advantage and one disadvantage to this decision to omit `unsigned`. [4 pt]

Advantage: Some possible answers:

- Less confusing/more consistent arithmetic interpretations for the programmer.
- Fewer cases of implicit casting.
- Fewer data types to worry about.

Disadvantage: Some possible answers:

- Need to use larger data widths for numbers in the range (TMax, UMax] for a given width.
- More difficult to do unsigned comparisons.
- More difficult to do zero-extension.

(C) **Condition codes** are part of the *processor/CPU state*. Would our instruction set architecture (ISA) still work if we got rid of the condition codes? *Briefly* explain. [4 pt]

Circle one: Yes (No)

Explanation: Our jump and set instructions, which rely on the values of the condition codes, would no longer work. Without jump instructions, we couldn't implement most of our program's control flow.