Problem 1

Fill in the blanks of the assembly code generated from the following C function and explain what the function does. Assume 64 bit operations. Write your answer for blanks to the right of the letters.

```assembly
function_asm:
    cmpq $1, %rdi
    jle .E2 a. jle
    movq %rdi, %rax
    cqto
    movq $2, %rsi
    idivq %rsi
    movq %rax, %r11 b. movq
    .L1:
    cmpq %r11, %rsi
    jg .E1 c. jg
    movq %rdi, %rax
    cqto
    idivq %rsi
    cmpq $0, %rdx d. cmpq
    je .E2
    addq $1, %rsi e. $1
    jmp .L1
    .E1:
    movq $1, %rax f. $1
    ret g. ret
    .E2:
    movq $0, %rax h. movq
    ret i. ret
```

What does the function do?

The function takes in an integer \( n \) and checks to see if it is prime. It computes the remainder of the number by different values \( i \) between 2 and \( n/2 \); if the remainder is ever 0, that means that \( i \) divides \( n \) and \( n \) is not prime, so it returns 0 to reflect that \( n \) is not prime. continually mods the number in a loop and if it mods to the number to a 0, it returns 0 to reflect that it is not a prime number. If it reaches the end of the while loop, it returns 1 to reflect that the number passed in is a prime number.

Problem 2

Consider the table below, which shows the initial contents of some registers and memory locations:

<table>
<thead>
<tr>
<th>Registers</th>
<th>Values</th>
<th>Memory</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>16</td>
<td>0x3FF0</td>
<td>10</td>
</tr>
<tr>
<td>rdx</td>
<td>32</td>
<td>0x3FF8</td>
<td>100</td>
</tr>
<tr>
<td>rcx</td>
<td>2</td>
<td>0x4000</td>
<td>210</td>
</tr>
<tr>
<td>rbx</td>
<td>0x3FF8</td>
<td>0x4008</td>
<td>24</td>
</tr>
</tbody>
</table>

a. Fill in Table 1 showing the results if the following machine code is run from the initial state:
movq $1, %rax
subq $24, %rdx
addq %rcx, %rax
shlq $3, %rdx

<table>
<thead>
<tr>
<th>Table 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>rax</td>
</tr>
<tr>
<td>rdx</td>
</tr>
<tr>
<td>rcx</td>
</tr>
<tr>
<td>rbx</td>
</tr>
</tbody>
</table>

b. Fill in Table 2 showing the results if instead the following machine code is run from the initial state:

leaq (%rbx, %rcx, 4), %rax // rax = 0x4000
movq %rdx, 8(%rax) // 0x4008 = 32
subq $8, %rbx // rbx becomes 0x3FF0
subq $10, (%rbx) // subtract 10 from 10 = 0
subq $16, %rax // rax = 0x4000 - 16 = 0x3FF0

<table>
<thead>
<tr>
<th>Table 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>rax</td>
</tr>
<tr>
<td>rdx</td>
</tr>
<tr>
<td>rcx</td>
</tr>
<tr>
<td>rbx</td>
</tr>
</tbody>
</table>

2
Problem 3

This is the assembly associated with the function long function_A(long n):

```assembly
function_A:
    movq $-1, %rax
    movq $0, %rcx
    cmpq %rcx, %rdi
    jl .L5
    movq $1, %rax
    movq $1, %rdx
    jmp .L3

.L4
    imulq $3, %rax
    addq $1, %rdx

.L3
    cmpq %rdi, %rdx
    jle .L4

.L5
    ret
```

A. Write C code that corresponds to the assembly given above. Give the variables meaningful names, not the names of registers.

```c
long function_A(long power){
    long result = 1;
    if(power < 0){
        return -1;
    }
    else {
        for(int i = 1; i <= power; i++){
            result = result*3;
        }
    }
    return result;
}
```

B. Explain in a sentence or two what this function does.

Returns 3 to the power of a nonnegative integer. Returns -1 otherwise.

Problem 4

(Based on the textbook problem 2.87.)

Just for fun, we define a new floating point standard, called UMN-20, which contains 20 bits. This format has 1 sign bit, 6 exponent bits (k=6), and 13 fraction bits (n=13). The exponent bias is \(2^{6-1} - 1 = 31\).

A. Fill in the table that follows for each of the numbers given, with the following instructions for each column:

- **Hex**: the four hexadecimal digits describing the encoded form.
- **M**: the value of the significand. This should be a number of the form \(x \text{ or } x/2 \text{ where } x \text{ is an integer and } y \text{ is an integral power of } 2\). Examples include 0, 67/64, and 1/256
- **E**: the integer value of the exponent.
- **V**: the numeric value represented. Use the notation \(x \times 2^z\), where \(x\) and \(z\) are integers.
- **D**: the (possibly approximate) numeric value, rounding to 3 bits and rounding towards 0.

```
**Example:** to represent the number \(3/4\) we would have \(s=0\), \(M=3/2\), and \(E=-1\). Our number would therefore have an exponent field of 0111102 (decimal value of \(31 - 1 = 30\)) and a significand field of 10000000000002, giving a hex representation 3C000. The numerical value is 0.75. You need not fill in entries marked –.

<table>
<thead>
<tr>
<th>Description</th>
<th>Hex</th>
<th>M</th>
<th>E</th>
<th>V</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>3/4</td>
<td>3D000</td>
<td>3/2</td>
<td>-1</td>
<td>3/4</td>
<td>0.75</td>
</tr>
<tr>
<td>100</td>
<td>4B200</td>
<td>25/16</td>
<td>6</td>
<td>25 (*2^2)</td>
<td>100.0</td>
</tr>
<tr>
<td>Largest value &lt; -2</td>
<td>C0001</td>
<td>-8193/8192</td>
<td>1</td>
<td>-8193 (*2^{-12})</td>
<td>-2.000244</td>
</tr>
<tr>
<td>Smallest positive normalized value</td>
<td>02000</td>
<td>1</td>
<td>-30</td>
<td>1 (*2^{-30})</td>
<td>0.000000000031</td>
</tr>
<tr>
<td>Number with hex 12340</td>
<td>12340</td>
<td>141/128</td>
<td>-22</td>
<td>141 (*2^{-29})</td>
<td>0.000000262</td>
</tr>
<tr>
<td>NaN</td>
<td>7E001</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

100 :

**Step 1:** 100 can be written as \(64 + 32 + 4\), which in binary, is 11001002.

**Step 2:** From Step 1, we can rewrite as \(1.100100 \times 2^6\), which is the same as \(1.100100 \times 2^{37-31}\). Compare to the formula \(M \times 2^E\). Therefore, \(M = 1.100100\), which is \(1 + \frac{1}{2} + \frac{1}{16} = 15/16\). \(E = 6\). We also change this expression into the notation of \(V\), as \(25/16 = 25 \times 2^2\). Therefore, \(D\) as computed from \(V\) is 100.0.

**Step 3:** From Step 2, compare \(1.100100 \times 2^6\) to the formula \(1.frac \times 2^{exp-31}\). Therefore, the exponent bits have the value 37, or 100101 in binary. The fractional bits are 100100 expanded to 13 bits, which is 1001000000000 in binary.

**Step 4:** From Step 3, the hex representation \{sign bits\}{exp bits\}{fractional bits\}, i.e \{0\} \{100101\} \{1001000000000\}. Grouping by 4 bits at a time, we have, 0100 1011 0010 0000 0000, i.e. 0xB200 in hex.

Largest value less than -2 :

**Step 1:** The value -2 can be expressed as \(-102\) in binary.

**Step 2:** From Step 1, we can rewrite \(-102\) as \(-1.0 \times 2^4\), which is \(-1.0 \times 2^{32-31}\). The next number which is smaller than this is \(-1.00000000000001 \times 2^4\). Compare to the formula \(M \times 2^E\). Therefore \(M = -1.00000000000001 = -8193/8192\), and \(E = 1\). The notation of \(V\) is obtained by simplifying to \(-8193/8192\) \(* 2^1 = -8193 \times 2^{-12}\). \(D\) can be computed from \(V\), as -2.000244.

**Step 3:** From Step 2, compare \(-1.00000000000001 \times 2^{32-31}\) to the formula \(1.frac \times 2^{exp-31}\). The fractional bits are 00000000000012 in binary. The sign bit is 1 because it is a negative number. The exp bits have value 32, which is 1000002 in binary.

**Step 4:** From Step 3, the hex representation is \{sign bits\}{exp bits\}{fractional bits\}, i.e \{1\} \{100000\} \{0000000000001\}. In groups of four this is 1100 0000 0000 0000 0001, which is 0xC0001 in hex.

Smallest positive normalized value :

**Step 1:** This number has no fractional bits and the smallest possible exponent. The sign bits are 0, because it is positive. Therefore, we can write the binary value as \{sign bit\}{exp bits\}{fractional bits\}, which is \{0\} \{000001\} \{0000000000000\}. In groups of 4, this is 0000 0010 0000 0000 0000, or 0x02000.

**Step 2:** We use the formula \(sign \times 1.frac \times 2^{exp-31}\), and replace the variables to get \(1.0000000000000000 \times 2^{-30}\). By matching this to the formula \(M \times 2^E\), therefore, \(M = 1\) and \(E = -30\).

**Step 3:** In the notation of \(V\), the value is \(1 \times 2^{-30}\). The decimal value \(D\) works out to 0.0000000000931.

Number with hex 12340 :

**Step 1:** The number 12340 can be written as 0001 0010 0011 0100 0000 in binary groups of 4, or using a different grouping as 0 001001 0001101000000, where these groups represent the sign, exp bits and fractional bits respectively. We note that the exp bits have value 9.

**Step 2:** We use the formula \(sign \times 1.frac \times 2^{exp-31}\) to obtain the number as 1.0001101 \(\times 2^{9-31}\). This can be rewritten as \((1 + \frac{1}{16} + \frac{1}{32} + \frac{1}{128}) \times 2^{-22}\) or \((141/128) \times 2^{-22}\). Therefore, matching the formula \(M \times 2^E\), this yields \(M = 141/128\) and \(E = 2^{-22}\).
Step 3: From Step 2, we simplify the expression into the notation of $V$, as $141 \times 1/128 \times 2^{-22} = 141 \times 2^{-7} \times 2^{-22} = 141 \times 2^{-29}$. The decimal value $D$, as computed from $V$, is 0.000000262.

NaN:
This number is represented as \{sign\}{exp bits}{fractional bits}. The only requirement is that the exp bits should be all ones and the fractional bits should be anything other than all zeros. One possible answer is \{0\}{111111}{0000000000001}, but there can be many other answers which satisfy the above criteria. The groups of 4 representation is 0111 1110 0000 0000 0001, i.e. 0x7E001 in hex. NaN means “not a number”, hence there is no $M, E, V, D$ value.

B. Floating point numbers in general, and in this case specifically the UMN-20 format, support addition and subtraction, but this operation is not necessarily associative. To illustrate this, please fill in the following table and briefly comment on what you observe.

<table>
<thead>
<tr>
<th>Computation</th>
<th>Value</th>
<th>Computation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1 = 2^{31} + 2^{31}$</td>
<td>$Inf$</td>
<td>$R_1 = 2^{31}$</td>
<td>$2^{31}$</td>
</tr>
<tr>
<td>$L_2 = 2^{31}$</td>
<td>$2^{31}$</td>
<td>$R_2 = 2^{31} - 2^{31}$</td>
<td>$0$</td>
</tr>
<tr>
<td>$L = L_1 - L_2$</td>
<td>$Inf$</td>
<td>$R = R_1 + R_2$</td>
<td>$2^{31}$</td>
</tr>
</tbody>
</table>

Does $L$ equal $R$?
Ans: No, $L$ is $Inf$ whereas $R$ is $2^{31}$.

Problem 5
(Based on textbook problem 3.60)

The assembly for the function was produced with GCC.

```
pushq %rbp
movq %rsp, %rbp
subq $32, %rsp
movq %rdi, -24(%rbp)
movl $0, -8(%rbp)
.L6:
cmpl $25, -8(%rbp)
jg .L7
movl $26, -4(%rbp)
.L5:
cmpl $25, -4(%rbp)
jle .L4
call rand
movl %eax, -4(%rbp)
andl $31, -4(%rbp)
jmp .L5
.L4:
movq -24(%rbp), %rdx
movl -4(%rbp), %ecx
movl -8(%rbp), %eax
movl %ecx, %esi
movl %eax, %edi
call swap
addl $1, -8(%rbp)
jmp .L6
```
void create_shuffle(char *table){
    for (int i=_0_; i<_26_; i++){
        int j = _26_;
        while ( j >= _26_){
            j = _rand();
            j= j & _31_;
        }
        swap(_i_, _j_, table);
    }
}

.L7:
        nop
        leave
        ret

Fill in the blanks for the C code, which was compiled to obtain this function.