Problem 1

When introducing RISC-V instructions, we have seen several required formats that do not seem to follow a straightforward or convenient design.

(a) All arithmetic operations must have 3 operands, which makes the addition of n numbers requiring $n - 1$ instructions.

(b) The S-type RISC-V instruction format has the 12-bit immediate field split into two fields, which makes the format hard to read and understand.

(c) There is no bitwise NOT instruction provided, which makes it inconvenient to write the code for NOT operations.

For each of the three cases above, explain why RISC-V follows this design.

(a) The reason for each instruction having three operands is to coincide with the design principle that simplicity favors regularity. This means that the simplest solution is the one that is most regular. In the first case, the addition instruction is limited to having three operands due to the fact that this allows each of the instructions to be of the same length. If we have an addition operation that deals with the sum of 4 numbers, and one that only deals with 2, the computer would have to write different instructions for each of the two sums since they have a different of terms. With the required format however, the computer will only have to deal with three operands at one time, every time, for the practice of addition. Despite having different numbers of instructions, it is still the most efficient since the computer does not have to deal with multiple formats each time it has to do addition, it can simply follow one convention each time.

(b) For the S-type instructions the immediate field is split into two separate sections for the same reason as before. Since this type of instruction has 2 source registers, in order to line the bits up with another type of instruction that uses 2 source registers, the R-type instructions, the immediate field is split into separate sections. This leads the source register fields, funct3 field, and the opcode field, to all be in the same place. This once again satisfies the regularity principle as the fields in the different types of instructions are lined up in the same place allowing the computer to recognize them more quickly.

(c) For the last case, if we look at a different design principle, good design demands good compromises, we can see why there is no need for a specific NOT instruction. With the use of the XOR instruction, the NOT operation can be achieved and for this reason there is no need to design another instruction simply for a specific case when the case is already covered by another instruction. For this reason, it is more efficient to simply use an existing operation to get the desired behavior.

Problem 2

We have the following C code:

```
A[6] = A[6] + 3;
```

Assuming the base address of array `A` in memory is stored in register x22, and the memory is byte addressed, please convert the C code to the following lower-level code:

(a) RISC-V assembly code

(b) Machine code in binary

For each line of assembly code, write a comment explaining what the code is doing and for each line of machine code explain which instruction format is used.

(a)

```
lw x9, 24(x22)       // load from memory A[6] into register x9
addi x9, x9, 3       // perform addition A[6] + 3
sw x9, 24(x22)       // store into memory at A[6] the value of A[6] + 3
```

(b)

```
00000001100010110010010010000011    I-type Instructions
00000000001101001000010010010011    I-type Instructions
00000000100110110010110000100011    S-type Instructions
```

Problem 3

Assuming variable `i` is in register x5, the base address of array `A` in memory is in register x22, and variable `s` is in register x23:

(a) Please convert the following C code to RISC-V assembly code.

```
for (i = 99; i >= 0; i = i - 1){
    A[i] = A[i] + s;
}
```

(b) Now the C code is updated to the following, which would produce the same results as in part a.

```
for (i = 99; i >= 0; i = i - 2){
    A[i] = A[i] + s;
    A[i-1] = A[i-1] + s;
}
```

Convert this loop to RISC-V and discuss the pros and cons compared to the version in a.

(a)

```
       addi x5, x0, 99       // initialize value of i
Loop:  ssli x10, x5, 2       // shifts i by 4 bytes
       add x10, x10, x22     // address of A[i]
       lw x9, 0(x10)         // x9 = A[i]
       blt x5, x0, Exit      // exit if i < 0
       add x9, x9, x23       // A[i] = A[i] + s
       sw x9, 0(x10)         // store A[i] + s into A[i]
       addi x5, x5, -1       // set i = i - 1
       beq x0, x0, Loop      // loop condition
Exit:  ...
```

(b)

```
        addi x5, x0, 99         // initialize value of i
Loop:   ssli x10, x5, 2         // shifts i by 4 bytes
        add x10, x10, x22       // address of A[i]
        lw x9, 0(x10)           // x9 = A[i]
        addi x4, x5, -1         // x4 = i - 1
        ssli x11, x4, 2         // shifts i-1 by 4 bytes
        add x11, x11, x22       // address of A[i-1]
        lw x8, 0(x11)           // x8 = A[i-1]
        blt x5, x0, Exit        // exit if i < 0
        add x9, x9, x23         // A[i] = A[i] + s
        sw x9, 0(x10)           // store A[i] + s into A[i]
        add x8, x8, x23         // A[i-1] = A[i-1] + s
        sw x8, 0(x11)           // store A[i-1] + s into A[i-1]
        addi x5, x5, -2         // set i = i - 2
        beq x0, x0, Loop        // loop condition
Exit:   ...
```

From the two sets of assembly code it can be seen that one is significantly longer than the other. And this leads to the main advantages and disadvantages of the two loops written in C code. The code for part b requires far more designation of resources by the CPU as it requires the use of more registers and more lines of code to assemble is general. For this reason, the loop in part a is much more optimized and will much more efficiently than the loop in part b. Although both of the loops provide the same function, since the loop in part a is less lines of assembly code, the program will be a smaller size than its counterpart in part b and therefore, it will be more efficient, making it the better option.