Problem 1

(a) Please describe what is an overflow under the context of addition and subtraction, what is the cause of it, and why it is an issue that we need to deal with.

Overflow may occur in two separate cases, when adding two numbers of the same sign or subtracting two numbers of opposite signs. Adding together two positive or negative numbers may concede a result which exceeds the number of bits of the two operands. When subtracting two numbers of opposite signs, the same is true since the operation is equivalent to that of adding two numbers of the same sign. Since the result cannot be represented by the number of bits of the operands, it is called overflow. This issue needs to be dealt with since in many cases within programming, numbers must be summed together and in a lot of cases, they both are of the same sign. If the result is interpreted and represented incorrectly, all of the proceeding calculations of the program will be erroneous and lead to different results. For this reason, the overflow problem must be worked around.

(b) Please discuss how a computer may detect overflow in the following cases involving signed operands and explain why the approach works.

1. Adding two positive integers
   - In these cases, the computer can detect overflow by examining the most significant bit of the operation's result. In this case of adding two positive numbers, if there is overflow, the most significant bit will be 1, signifying that the number is negative. If we are adding two positive numbers, the result should be positive hence, if the result is negative, there must have been overflow.
2. Adding two negative integers
   - Similar to the case of adding two positive numbers, if we add two negative numbers and the result is positive, most significant bit is 0, overflow must have occurred.
3. Subtracting a positive integer from a negative integer
   - This case is operationally the same as adding two negative numbers. The expected behavior of adding two negative numbers will be a negative number and if the result is a positive number, overflow has occurred.
4. Subtracting a negative integer from a positive integer
   - This case will be the same as adding together two positive integers. The result of the operation is expected to be positive, if the result is negative, overflow has occurred.

(c) In what scenario's would there be no overflows? Please discuss one scenario for addition, one scenario for subtraction, and explain why.

For the case of addition, when we add together numbers of different signs, there is no chance of overflow. If the numbers have different signs, the result will always tend towards zero for any two combination of numbers. There is no way for the resulted number to be

greater than the positive integer since there is a negative number being added to it, and the opposite is true for the negative integer. For this reason, there is no chance for the result to be greater than the number of bits required to represent each of the operands.

In the case of subtraction, there is no overflow when adding together numbers of the same sign. Since subtracting numbers of the same sign is equivalent to adding numbers with opposite signs, the result of the operation will always be less than the positive operand and greater than the negative operand. This leaves no way for the result to not be able to be represented with the given number of bits.

Problem 2

(a) Multiplication: If the multiplicand is in register x28, the multiplier is in register x29, and we would like the multiplication result to be in register x8 (higher 32 bits) and x9 (lower 32 bits). Please write the RISC-V instructions for multiplications in the following cases and explain what each line of code does using comments.

1. The values of x28 and x29 are both signed integers

```
mul x9, x28, x29          // x28 times x29 and place lower 32 bits in x9
mulh x8, x28, x29         // x28 times x29 and place upper 32 bits in x8
```

2. The values of x28 and x29 are both unsigned integers

```
mul x9, x28, x29          // x28 times x29 and place lower 32 bits in x9
mulhu x8, x28, x29        // x28 times x29 and place upper 32 bits in x8
```

3. The value of x28 is a signed integer and the value of x29 is an unsigned integer

```
mul x9, x28, x29          // x28 times x29 and place lower 32 bits in x9
mulhsu x8, x28, x29       // x28 times x29 and place upper 32 bits in x8
```

(b) Division: If the dividend is in the register x28, the divisor is in register x29, and we would like to have the quotient in x8 and the remainder in x9. Please write the RISC-V instructions for division in the following cases and explain what each line of code does using comments.

1. The values of x28 and x29 are both signed integers

```
div x8, x28, x29          // x28 divided by x29 and place quotient in x8
rem x9, x28, x29          // x28 divided by x29 and place remainder in x9
```

2. The values of x28 and x29 are both unsigned integers

```
divu x8, x28, x29         // x28 divided by x29 and place quotient in x8
remu x9, x28, x29         // x28 divided by x29 and place remainder in x9
```

Problem 3

(a) Please represent the number -1.2356×10^8 using the 32-bit RISC-V format.

Using a converter to convert the number to binary and using the convention to write the number in normalized form we get,

$$(-1)^1 \times 1.11010111010110000001000000 \times 2^{26}$$

Here, the sign bit is one since the number is negative and the exponent will be 26 plus the bias which is 127 due to this being single point precision. Since the fraction in this case is greater than 23 bits, exactly 23 bits after the binary point will be represented with the rest being neglected.

Sign bit: 1

Exponent: $26 + 127 = 153 = (10011001)_2$

Fraction: 11010111010110000001000

Writing this in the RISC-V format we get,

$$11001100111010111010110000001000$$

(b) What is the range, lower bound and upper bound, of floating-point numbers that the 32-bit RISC-V format can represent? What problems will be caused if the floating-point number we want to represent goes below the lower bound or above the upper bound? What would be a solution to such problems?

Since there are two reserved values for the exponent, 11111111 and 00000000, the lower bound will be where the exponent is $(00000001)_2 - (127)_{10} = (-126)_{10}$ and the fraction is 23 bits of zero. This will result in a lower bound of $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$. The upper bound would be the same except this time the exponent will be the largest it can possibly be, 11111110. In this case, the actual exponent will be $(11111110)_2 - (127)_{10} = (127)_{10}$. The largest fractional value we can have is 23 bits of one which makes the upper bound, $\pm 2.0 \times 2^{127} \approx \pm 3.4 \times 10^{38}$.

As with overflow in general, if the result exceeds the upper/lower bounds, the number will be incorrectly represented within the program and this can lead to future calculations which rely on the number to be thrown off as well. The solution would be to use a higher precision representation since double precision can represent a far more expansive range than single precision.