**Name:**

**Advanced Programming in C++**

**Lab Exercise 3/21/2024**

**Computer Number Systems**

**Storing Numbers in the Computer**

Integers are stored using sign-magnitude format while floating point numbers are stored using the IEEE 754 standard.

|  |  |
| --- | --- |
| 1 bit | 31 bits or 63 bits (depending on machine operating system) |
| Sign | Magnitude |

Note: sign bit is 0 for positive numbers and 1 for negative numbers

IEEE 754 single precision uses 32 bits while double precision uses 64 bits. There is also an 80 bit data implementation specified by IEEE 754.

**Single Precision**

|  |  |  |
| --- | --- | --- |
| 1 bit | 8 bits (bias = 127) | 23 bits |
| Sign | Exponent | Mantissa |

**Double Precision**

|  |  |  |
| --- | --- | --- |
| 1 bit | 11 bits (bias = 1023) | 52 bits |
| Sign | Exponent | Mantissa |

Note: sign bit is 0 for positive numbers and 1 for negative numbers

**Converting to IEEE 754 Form**

*Put 0.085 in single-precision format*

1. **The first step is to look at the sign of the number.**  
   Because 0.085 is positive, the sign bit =0.

**Note:** A negative number would have a sign bit of 1

1. **Write 0.085 in base-2 scientific notation.**  
   This means that we must factor it into a number in the range [1 <= n < 2] and a power of 2.  
     
   0.085 / 2power = (1+fraction)  
     
   So we can divide 0.085 by a power of 2 to get the (1 + fraction).   
     
   0.085 / 2-1 = 0.17  
   0.085 / 2-2 = 0.34  
   0.085 / 2-3 = 0.68  
   **0.085 / 2-4 = 1.36**  
     
   Therefore, 0.085 = 1.36 \* 2-4

Note: Only the 0.36 part of this number is stored. The 1 is stored implicitly which allows us to have one more bit in the mantissa which doubles the precision.

1. **Find the exponent.**  
   The power of 2 is -4, and the bias for the single-precision format is 127. This means that the exponent = 123ten, or 01111011bin

4. **Write the fraction in binary form**  
The fraction = 0.36 . Unfortunately, this is not a "pretty" number, like those shown in the book. The best we can do is to approximate the value. Single-precision format allows 23 bits for the fraction.  
  
Binary fractions look like this:  
  
0.1 = (1/2) = 2-1  
0.01 = (1/4) = 2-2  
0.001 = (1/8) = 2-3  
  
To approximate 0.36, we can say:  
  
0.36 = (0/2) + **(1/4)** + (0/8) + **(1/16)** + **(1/32)** +...  
0.36 = 2-2 + 2-4 + 2-5+...   
  
0.36ten ~ 0.01011100001010001111011bin .  
  
The binary string we need is: 01011100001010001111011.  
  
It's important to notice that you will not get 0.36 exactly. This is why floating-point numbers have error when you put them in IEEE 754 format.

1. **Now put the binary strings in the correct order -**  
   1 bit for the sign, followed by 8 for the exponent, and 23 for the fraction. The answer is:

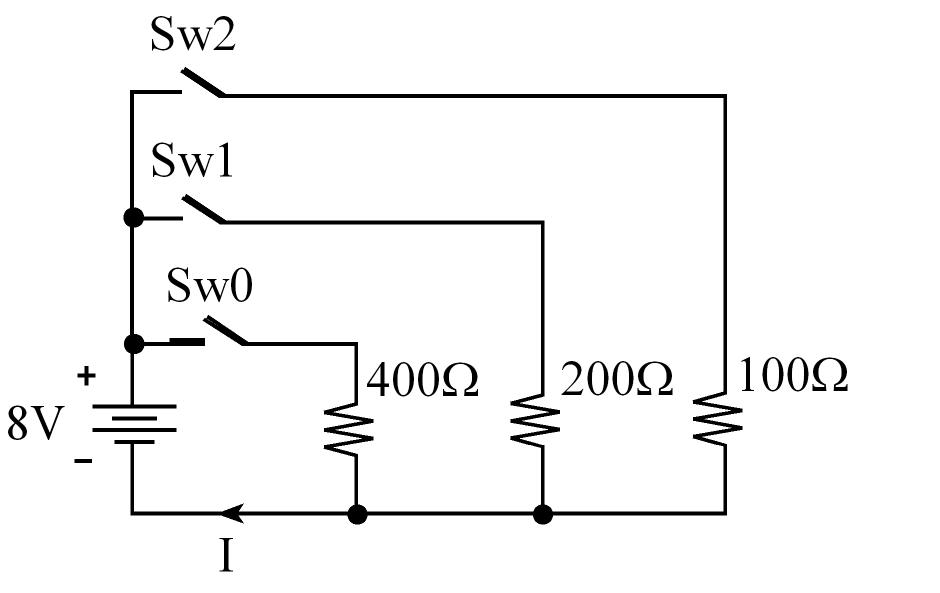
|  |  |  |  |
| --- | --- | --- | --- |
|  | **Sign** | **Exponent** | **Fraction** |
| **Decimal** | 0 | 123 | 0.36 |
| **Binary** | **0** | **01111011** | **01011100001010001111011** |

**Exercises**

1. Write a program that allows the user to enter an integer. The program should convert the integer into a binary number and display it. For example, if the user entered the number 37 it should output 100101.
2. Write a program that allows the user to enter a floating point number less than 1 and converts it into a binary number. Of example, if the user entered 0.75 it should output 0.11. For simplicity, limit your binary digits displayed to 12 bits to the right of the radix point.
3. Write a program that allows the user to enter a floating point number and converts it into a binary number. Of example, if the user entered 7.75 it should output 111.11. For simplicity, limit your binary digits to the right of the radix point displayed to 12 bits.
4. I have written a program that will allows the user to enter a floating point number and converts and displays the IEEE754 representation of that number (how it is actually stored in an IEEE754 compliant microprocessor). Run the program with a few numbers (positive, negative, and 0).

As computers are purely digital devices, they often are required to interface with our analog world. For example, to drive a speaker to listen to music, you would need to convert digital values to analog signals. This would require a device called a digital to analog converter (DAC). Real DACs are usually 16 or 24 bit which can represent 64 thousand or 16 million signal levels respectively.

For simplicity, consider this 3-bit digital to analog converter. We define the logic state of each switch as 0 or 1, where 0 means not on and 1 means on. Define a 3-bit number n (0 to 7) which specifies the three switch positions. n = 0 means none are on. n = 1 means Sw0 is on. n = 2 means Sw1 is on. n = 3 means Sw1 and Sw0 are on. n = 4 means Sw2 is on. n = 5 means Sw2 and Sw0 are on. n = 6 means Sw2 and Sw1 are on. n = 7 means all are pushed.



Derive a relationship between the current I and the number n.

Hint 1: Make a table of n versus I.

Hint 2: Ohms Law is the relationship between current (I), voltage (V), and resistance (R)

Answer: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_