Bit String to Single Precision Floating Number

Due: June 13, 2025

Problem Description

You are given an ASCII file whose size is smaller than 1024 bytes. The file contains some binary strings separated by spaces. Each binary string contains exactly 32 bits which encode a 32-bit 2's complement integer number or a single precision floating-point number. There can be more than one line in the file and a line may not have any bit strings. The input file name is example2.txt. In Homework 2, you were asked to write a RISC-V program to read the file, convert each 32-bit string into the corresponding integer number in decimal format, and display the decimal number on the monitor. In Homework 3, you are asked further to extend the program to convert a 32-bit string into a single precision floating-point number based on IEEE-std 754, 1985 and display it on the monitor. Here, you surely have an opportunity to use some floating-point instructions. Below are the tasks that you need do for a given 32-bit string:

1. Display the floating part of the corresponding single precision floating-point number calculated based on its mantissa and sign bit. Taking the first 32-bit string 10000000000000000000000000000001 in the example output as an instance, the floating part is equal to $-1.1920929E-7 = -2^{-23}$. For the second bit string, it is equal to $0.75000024 = 2^{-1} + 2^{-2} + 2^{-22}$.
2. Display the exponent part of the corresponding floating-point number. Taking the first 32-bit string 10000000000000000000000000000001 in the example output as an instance, the exponent part is 2^{-126} because the bit string represents a denormalized number. For the third bit string, it is equal to 2^{-42}.
3. Display the decimal value of the 32-bit string, which is equal to the product of the floating part and the exponent part. Taking the first bit string in the example output as an instance, its decimal value is $-1.4E-45 = -1.1920929E-7 \times 2^{-126}$.

You should use RARS instruction set simulator to develop and execute the assembly code since the ecall "Print Float" of Jupiter ISA simulator does not function correctly. It is very easy to convert a piece of RISC-V code for Jupiter to the one for RARS. The difference is mainly on the use of ecalls and directives. Some directives available in Jupiter are not available in RARS. However, for those not available in RARS, you can always find a replacement. For example, *.zero* is not available in RARS but *.space* can be used instead. Yet another example, *.rodata* is not available in RARS, but *.data* can be used instead. As to the usage difference in ecalls, Jupiter uses a0, a1, a2, … to hold the arguments for an ecall whereas RARS uses a7, a0, a1, a2,… to hold the arguments for an ecall; Jupiter uses a0 to hold an ecall code whereas RARS uses a7 to hold an ecall number (i.e., ecall code). Certainly, some ecall codes for the same task may also be different. For example, Jupiter uses ecall code 16 for closing a file whereas RARS uses ecall number 57.

Get RARS: https://github.com/TheThirdOne/rars?tab=readme-ov-file
What to download: **rars1_6.jar** on https://github.com/TheThirdOne/rars/releases/tag/v1.6
How to run RARS: Open a DOS command window and get to the directory where the Java archived file **rars1_6.jar** is located. Then, on the command line just type **java -jar rars1_6.jar.** Waiting for a few seconds, you will be given a beautiful RARS GUI window. You can edit, assemble, debug, and execute your RISC-V program there. It is very easy to use.
RARS directives: https://github.com/TheThirdOne/rars/wiki/Assembler-Directives
RARS ecalls: https://github.com/TheThirdOne/rars/wiki/Environment-Calls

Output Format

Your output should include all the lines printed for Homework 2. Besides, for each 32-bit string being treated as a single precision floating-point number, a few extra lines should be printed. If it is ***not a number***, one extra line containing "This is not a number: NaN" should be printed. If it is a number, three extra lines should be printed. The first extra line displays the floating part, the second extra line displays the exponent part, and the third extra line displays the decimal value of the corresponding floating-point number. Refer to the example output for details. Note that a denormalized number has a value of $x = (-1)^S \times (0 + Fraction) \times 2^{-(Bias-1)}$.

Hint

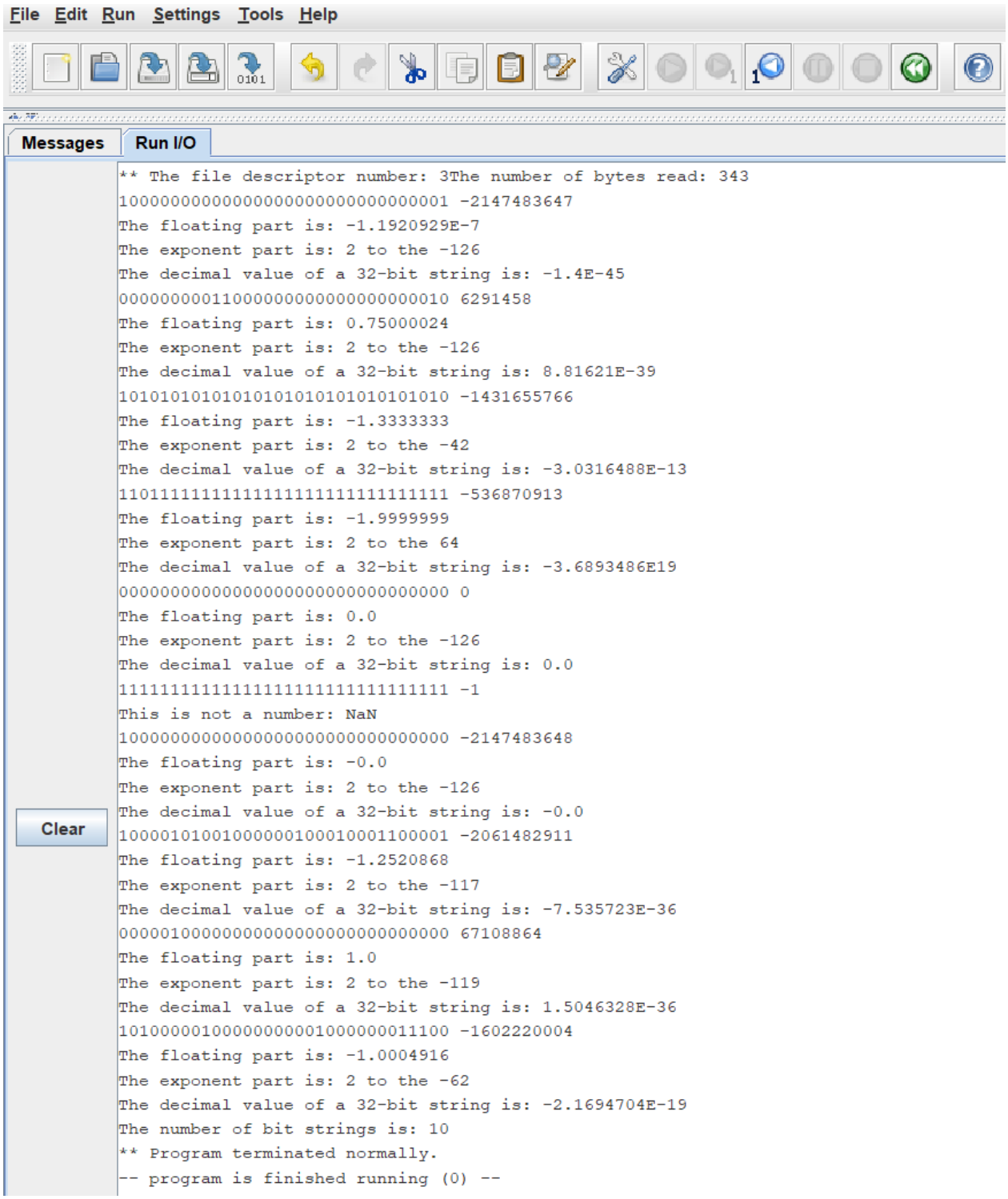
Assume that you have already read a 32-bit string and stored it in a register. To display the floating part and the exponent part, you have to shift the bits in the register to the right to extract respectively the mantissa, the excess exponent, and the sign bit. Hence, some logical operations such as AND, etc. may be used to mask out some bits in the register. Refer to Chapter 3's slides for details. Prior to calculating the floating part from the mantissa and the sign bit, you may need to know whether the underlying floating-point number is a denormalized number or not. Based on this piece of information, the exact floating part and exponent part can be calculated. As to displaying the decimal value of a 32-bit string, the simplest way is to move it from a 32-bit integer register to a floating-point register and then print out the floating-point register. Certainly, you can still iterate through the bits in the register to obtain the corresponding decimal value.

What Should Be Handed in

- A file contains the assembly code, *which should have a header same as the one in Homework 1*. The file name should be **sID.s** where ID is your student ID number. A valid file name must look like s1091111.s .
- A clip like the one shown in the example output below. Save the clip as a file called **sID.png**, where ID is your student ID number. A valid file name for an output clip must look like s1091111.png .
- The homework will not be graded if you do not follow the above rules.

Example Output

Given the bit strings in the file provided along with the problem sheet, your program's output except the file descriptor number should be exactly the same as the example output shown below.




```
** The file descriptor number: 3The number of bytes read: 343
10000000000000000000000000000001 -2147483647
The floating part is: -1.1920929E-7
The exponent part is: 2 to the -126
The decimal value of a 32-bit string is: -1.4E-45
00000000011000000000000000000010 6291458
The floating part is: 0.75000024
The exponent part is: 2 to the -126
The decimal value of a 32-bit string is: 8.81621E-39
10101010101010101010101010101010 -1431655766
The floating part is: -1.3333333
The exponent part is: 2 to the -42
The decimal value of a 32-bit string is: -3.0316488E-13
11011111111111111111111111111111 -536870913
The floating part is: -1.9999999
The exponent part is: 2 to the 64
The decimal value of a 32-bit string is: -3.6893486E19
00000000000000000000000000000000 0
The floating part is: 0.0
The exponent part is: 2 to the -126
The decimal value of a 32-bit string is: 0.0
11111111111111111111111111111111 -1
This is not a number: NaN
10000000000000000000000000000000 -2147483648
The floating part is: -0.0
The exponent part is: 2 to the -126
The decimal value of a 32-bit string is: -0.0
10000101001000000100010001100001 -2061482911
The floating part is: -1.2520868
The exponent part is: 2 to the -117
The decimal value of a 32-bit string is: -7.535723E-36
00000100000000000000000000000000 67108864
The floating part is: 1.0
The exponent part is: 2 to the -119
The decimal value of a 32-bit string is: 1.5046328E-36
10100000100000000001000000011100 -1602220004
The floating part is: -1.0004916
The exponent part is: 2 to the -62
The decimal value of a 32-bit string is: -2.1694704E-19
The number of bit strings is: 10
** Program terminated normally.
-- program is finished running (0) --
```