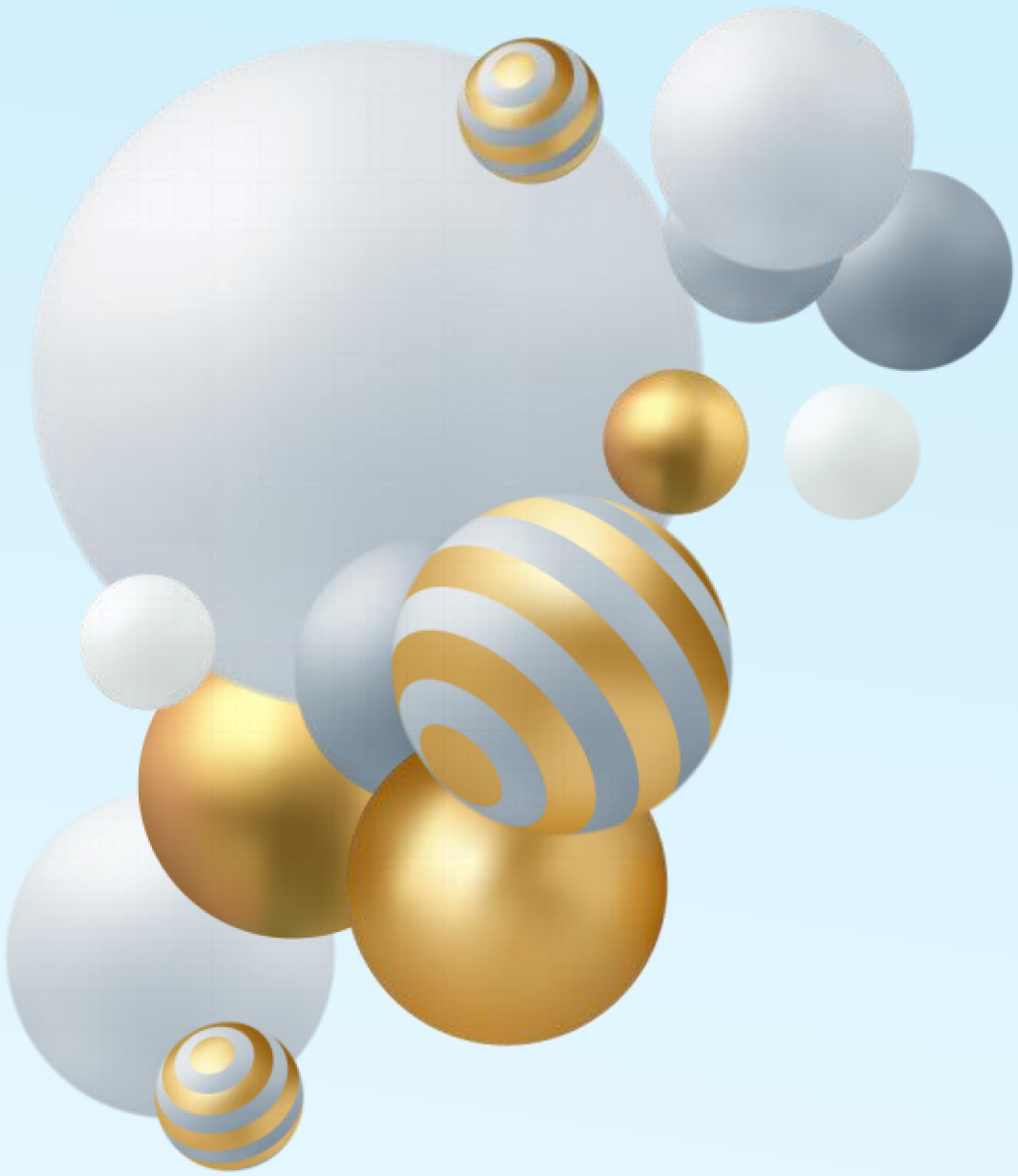
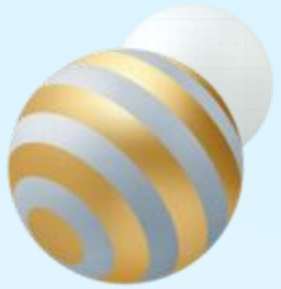


Computer Organization

Lab7 Floating-Point Number Processing

Floating-Point

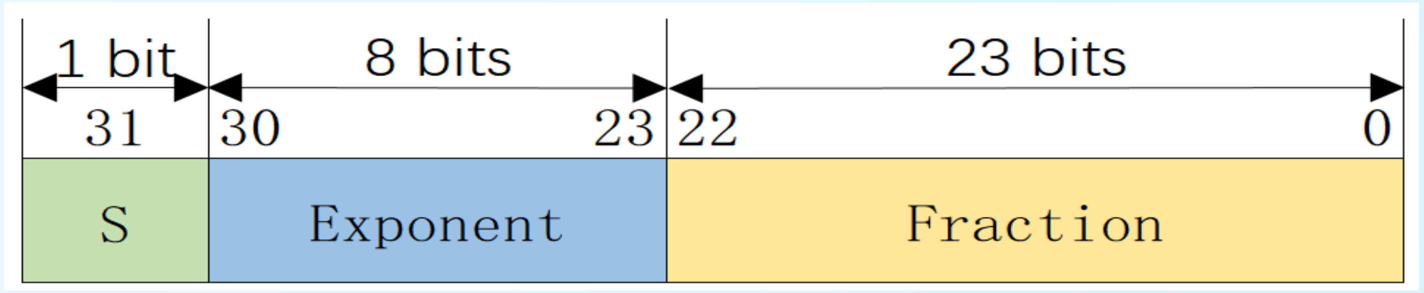


Topics

- Floating-Point Number
  - ✓ IEEE 754 On Single-Precision Floating-Point Number
  - ✓ IEEE 754 On Double-Precision Floating-Point Number
  - ✓ Conversion between Floating-Point and decimal Number
  - ✓ IEEE 754 Single-Precision Floating-Point Number Classification
  - ✓ Infinite vs NaN (Floating-Point)
- Floating-Point Instructions
  - ✓ Floating-Point Registers in RISC-V
  - ✓ Floating-Point Instructions Classification
  - ✓ Floating-Point system Calls in Rars
- Practice

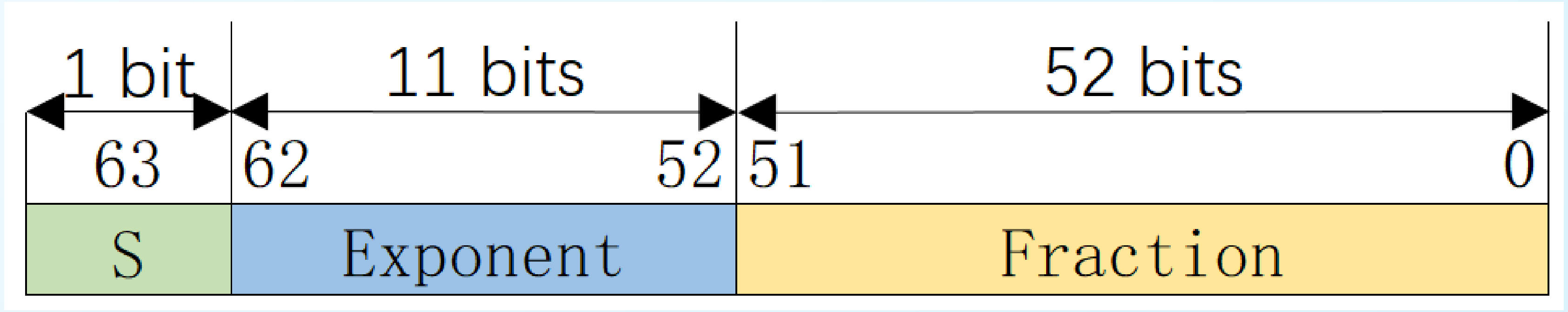
IEEE 754 On Single-Precision Floating-Point Number

- Single-Precision Floating-Point Number (32-bit width)
  - ✓ Signed bit (符号位): 1 bit, the most significant bit
    - 0: positive number
    - 1: negative number
  - ✓ Exponent (阶码): 8 bits
    - Used to represent indices with a base of 2
    - Using frame shift code (移码), and the **Bias** is 127 (0111_1111)
    - The real value is: **Exponent - Bias**
    - 0000_0000 and 1111_1111 are reserved
  - ✓ Fraction (尾数): 23 bits
    - Representing the decimal part under binary
    - Using true code (原码), and the real value is: **1. Fraction**
  - ✓ The real value of the floating-point number
    - $X = (-1)^S \times (1.\text{Fraction}) \times 2^{(\text{Exponent - Bias})}$



IEEE 754 On Double-Precision Floating-Point Number

- Double-Precision Floating-Point Number (64-bit width)
  - ✓ Signed bit: 1 bit
  - ✓ Exponent: 11 bits
  - ✓ Fraction: 52 bits
  - ✓ Shift code Bias: 1023



Conversion between Floating-Point and decimal Number (1)

- Convert 408.6875_{ten} to IEEE 754 single-precision floating-point number
  - ✓ Binary: 110011000.1011 $\longrightarrow$ $2^8 \times 1.100110001011$
  - ✓ Signed bit: 0
  - ✓ Exponent: $8 + 127 = 135_{ten} = 10000111_{two}$
  - ✓ Fraction:100110001011
  - ✓ Floating number:

0	10000111	10011000101100000000000

  - ✓ Convert binary to hexadecimal: 0x43CC5800

Conversion between Floating-Point and decimal Number (2)

- Suppose 0xC1830000 is the hexadecimal machine code for a IEEE 754 single precision floating-point number, convert it to its decimal value.

Binary: 1100 0001 1000 0011 0000 0000 0000 0000

Signed bit: 1

Exponent: $10000011_{two} = 131_{ten}$

The real value: $131 – 127 = 4_{ten}$

Fraction: 000 0011 0000 0000 0000 0000

The real value: 1.0000011

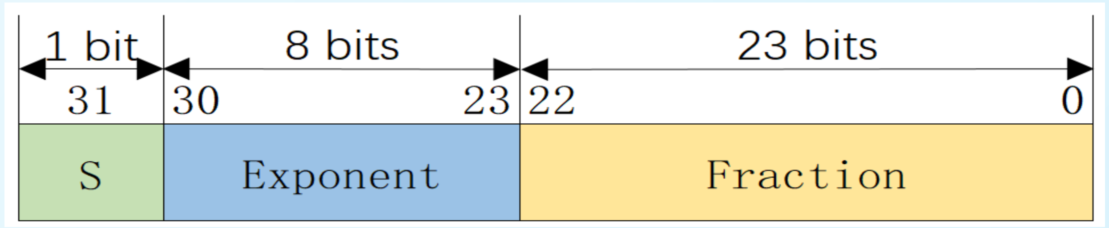
The true value in decimal:

$X = (-1)^S \times (1.\text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

$= (-1)^1 \times 1.0000011 \times 2^4$

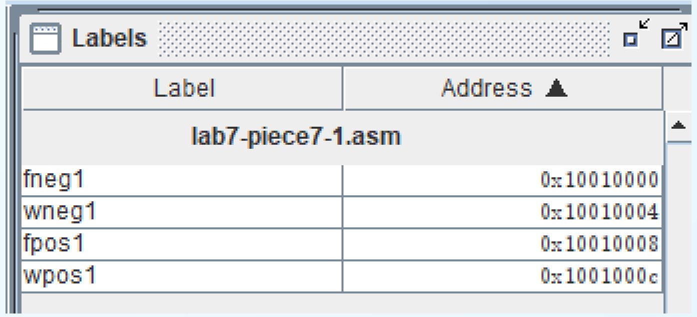
$= - 10000.011$

$= -16.375$



Conversion between Floating-Point and decimal Number (3)

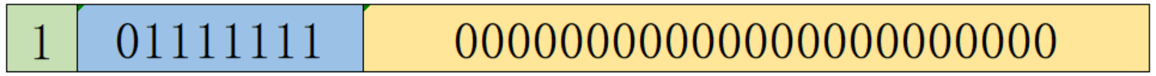
```
# Piece 7-1
.data
    fneg1:      .float    -1
    wneg1:      .word     -1
    fpos1:      .float    1
    wpos1:      .word     1
```



Label	Address ▲
lab7-piece7-1.asm	
fneg1	0x10010000
wneg1	0x10010004
fpos1	0x10010008
wpos1	0x1001000c

- **-1** = $(-1)^1$ x (1.**0**) x 2^0

  s: 1;   exponent: 0 + 0111_1111;   fraction: 0

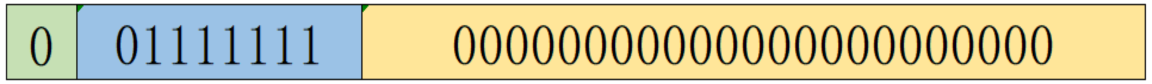


1	01111111	00000000000000000000000

  0xBF800000

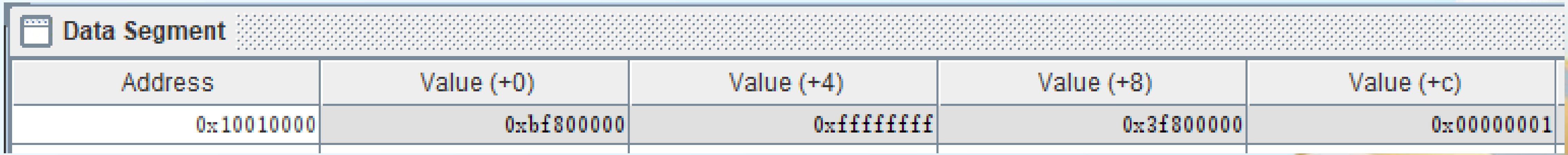
- **1** = $(-1)^0$ x (1.**0**) x 2^0

  s: 0;   exponent: 0 + 0111_1111;   fraction: 0



0	01111111	00000000000000000000000

  0x3F800000



Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0xbf800000	0xffffffff	0x3f800000	0x00000001

IEEE 754 Single-Precision Floating-Point Number Classification

Classification	Sign	Exponent (E)	Fraction (M)	Real value
Positive zero	0	0 (all 0s)	0	+0
Negative zero	1	0 (all 0s)	0	-0
Positive infinity	0	255 (all 1s)	0	+∞
Negative infinity	1	255 (all 1s)	0	-∞
NaN	0 or 1	255 (all 1s)	M ≠ 0	Not a number
Normalize positive numbers	0	$1 \leq E \leq 254$	M	$(-1)^0 \times (1.M) \times 2^{(E - 127)}$
Normalize negative numbers	1	$1 \leq E \leq 254$	M	$(-1)^1 \times (1.M) \times 2^{(E - 127)}$
Denormalize positive numbers	0	0 (all 0s)	M ≠ 0	$(-1)^0 \times (0.M) \times 2^{-126}$
Denormalize negative numbers	1	0 (all 0s)	M ≠ 0	$(-1)^1 \times (0.M) \times 2^{-126}$

Infinite vs NaN (Floating-Point)

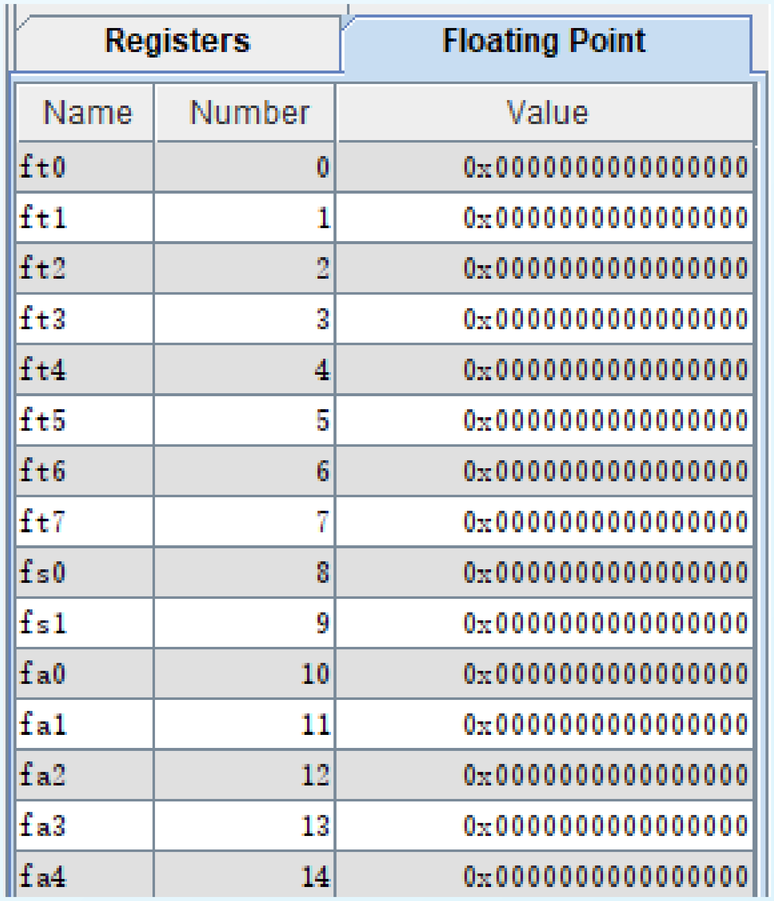
- Q1. What are the results of piece 7-2 and 7-3?
- Q2. Which piece of codes will get an infinite value, 7-2 or 7-3?
- Q3. Is the result positive infinity or negative infinity? What's the machine code of this infinity value?
- Q4. Which piece of codes will get the NaN, 7-2 or 7-3? What's the machine code of this NaN value?

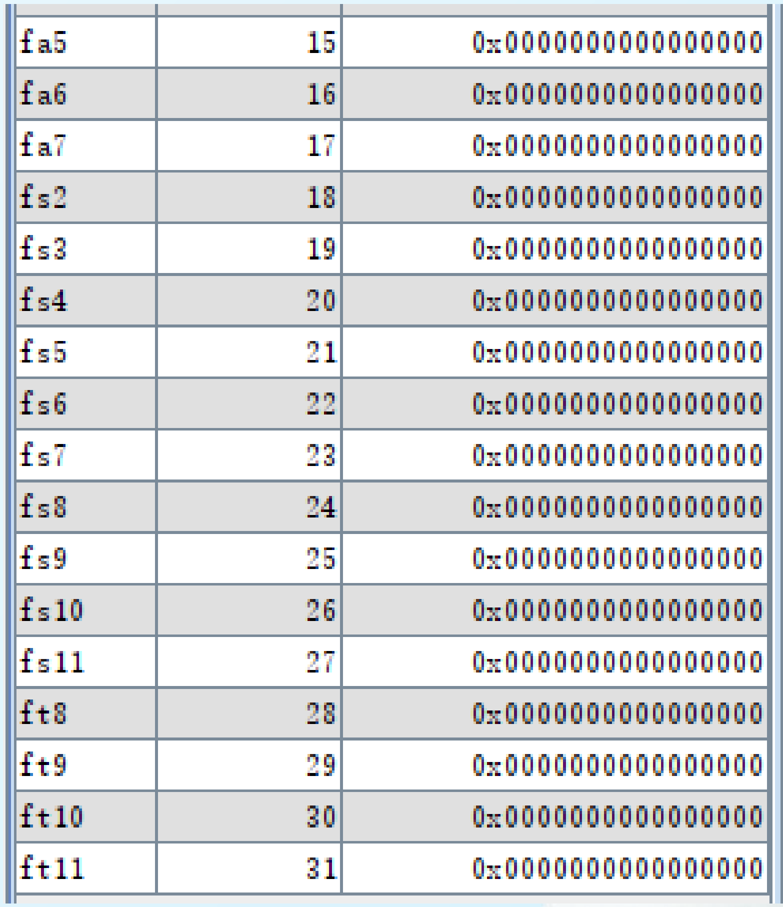
```
# Piece 7-2
.include "macro_print_str.asm"
.data
	sdata: .word 0xff7f7fff
	fneg1: .float -1
.text
	la t0, sdata
	flw ft0, (t0)
	fmul.s fa0, ft0, ft0
	li a7, 2
	ecall

	print_string("\n")

	la t0, fneg1
	flw ft0, (t0)
	fmul.s fa0, ft0, ft0
	li a7, 2
	ecall

	li a7, 10
	ecall
```

```
# Piece 7-3
.include "macro_print_str.asm"
.data
	sdata: .word 0xffff7fff
	fneg1: .float -1
.text
	la t0, sdata
	flw ft0, (t0)
	fmul.s fa0, ft0, ft0
	li a7, 2
	ecall

	print_string("\n")

	la t0, fneg1
	flw ft0, (t0)
	fmul.s fa0, ft0, ft0
	li a7, 2
	ecall

	li a7, 10
	ecall
```

Floating-Point Registers in RISC-V

- 32 floating-point registers.
- Each register is 64-bit width.
- View in “Floating Point” window in Rars.





Name	Number	Value
ft0	0	0x0000000000000000
ft1	1	0x0000000000000000
ft2	2	0x0000000000000000
ft3	3	0x0000000000000000
ft4	4	0x0000000000000000
ft5	5	0x0000000000000000
ft6	6	0x0000000000000000
ft7	7	0x0000000000000000
fs0	8	0x0000000000000000
fs1	9	0x0000000000000000
fa0	10	0x0000000000000000
fa1	11	0x0000000000000000
fa2	12	0x0000000000000000
fa3	13	0x0000000000000000
fa4	14	0x0000000000000000
fa5	15	0x0000000000000000
fa6	16	0x0000000000000000
fa7	17	0x0000000000000000
fs2	18	0x0000000000000000
fs3	19	0x0000000000000000
fs4	20	0x0000000000000000
fs5	21	0x0000000000000000
fs6	22	0x0000000000000000
fs7	23	0x0000000000000000
fs8	24	0x0000000000000000
fs9	25	0x0000000000000000
fs10	26	0x0000000000000000
fs11	27	0x0000000000000000
ft8	28	0x0000000000000000
ft9	29	0x0000000000000000
ft10	30	0x0000000000000000
ft11	31	0x0000000000000000

Floating-Point Instructions Classification

- RV32F, RV32D
- Instructions usually start with “f”
- Some basic type
  - ✓ Load and Store: e.g. flw, fld, fsw, fsd
  - ✓ Move data: e.g. fmv.s.x, fmv.x.s
  - ✓ Computational: e.g. fadd.s, fadd.d, fmadd.s, fmadd.d, fmax.d, fmax.s, fsqrt.s, fsqrt.d
  - ✓ Relational: e.g. fle.s, fle.d, flt.s, flt.d
  - ✓ Convert: e.g. fcvt.d.s, fcvt.d.w, fcvt.d.wu
- When the operands are single-precision float number, use “.s” as suffix; when the operands are double-precision float number, use “.d” as suffix.
  - ✓ floating add: fadd.s f1, f2, f3 # assign f1 to f2 + f3
  - ✓ floating add (64 bits): fadd.d f1, f2, f3 # assign f1 to f2 + f3
- Also supports pseudo instructions.

Floating-Point system Calls in Rars

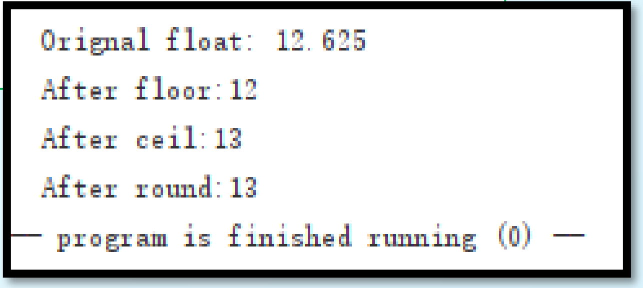
Name	Number	Description	Inputs	Ouputs
PrintFloat	2	Prints a floating point number	fa0 = float to print	N/A
PrintDouble	3	Prints a double precision floating point number	fa0 = double to print	N/A
ReadFloat	6	Reads a float from input console	N/A	fa0 = the float
ReadDouble	7	Reads a double from input console	N/A	fa0 = the double
RandFloat	43	Get a random float	a0 = index of pseudorandom number generator	fa0 = uniformly randomly selected from from [0,1]
RandDouble	44	Get a random double from the range 0.0-1.0	a0 = index of pseudorandom number generator	fa0 = the next pseudorandom

Demo(1)

- Run the codes of piece 7-4, and tell the function of it.
- Change float1 to value of 2147483647.825, what're the outputs, are they correct? Why?

```
# Piece 7-4-1
.include "macro_print_str.asm"
.data
        float1: .float 12.625
        float2: .float 0.5
.text
        la t0, float1
        flw ft0, (t0)
        la t0, float2
        flw ft1, (t0)

        print_string("Orignal float: ")
        print_float(ft0)

        print_string("\nAfter floor:")
        # floor operation
        fsub.s  ft2, ft0, ft1# ft2 = ft0 - 0.5
        # conver the result to a 32-bit integer
        fcvt.w.s  a0, ft2    # a0 = (int32_t)ft2
        li a7, 1
        ecall
```

```
# Piece 7-4-2

        print_string("\nAfter ceil:")
        # ceil operation
        fadd.s  ft2, ft0, ft1# ft2 = ft0 + 0.5
        # conver the result to a 32-bit integer
        fcvt.w.s  a0, ft2    # a0 = (int32_t)ft2
        li a7, 1
        ecall

        print_string("\nAfter round:")
        # round operation
        fcvt.w.s  a0, ft0    # a0 = (int32_t)ft0
        li a7, 1
        ecall

        end
```



```
Orignal float:  12.625
After floor: 12
After ceil: 13
After round: 13
— program is finished running (0) —
```

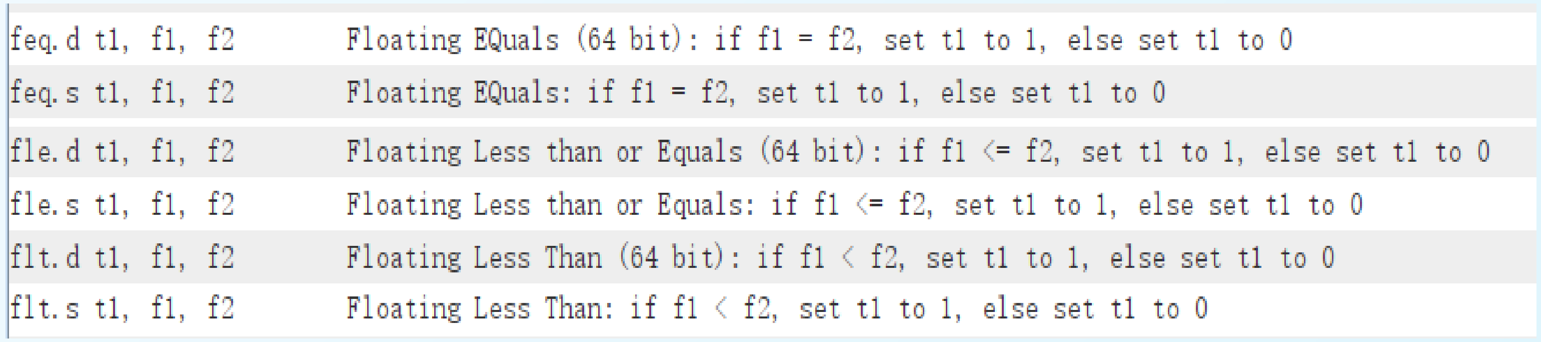
```
#Add the content to
"macro_print_str.asm"

.macro print_float(%fr)
        addi sp, sp, -8
        fsw fa0, 4(sp)
        sw a7, 0(sp)

        fmv.s fa0, %fr
        li a7, 2
        ecall

        lw a7, 0(sp)
        flw fa0, 4(sp)
        addi sp, sp, 8
.end_macro
```

Practice 1

- Conversion between hexadecimal floating-point numbers and decimal numbers.
  - ✓ Convert **409.2675**$_{\textbf{ten}}$ to IEEE 754 hexadecimal single precision floating-point numbers.
  - ✓ Suppose **0xc1a6fae1** is the hexadecimal machine code for a IEEE 754 single precision floating-point number, please calculate its corresponding decimal value.
  - ✓ Convert **-409.2675**$_{\textbf{ten}}$ to IEEE 754 hexadecimal double precision floating-point numbers. (optional)
  - ✓ Suppose **0xc0611bf1a9fbe76d** is the hexadecimal machine code for a IEEE 754 double precision floating-point number, please calculate its corresponding decimal value. (optional)
  - ✓ Tip: you can use Rars to get help.

Practice 2

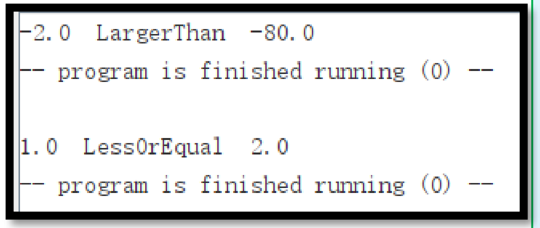
- Complete the codes.
- Tip1: There is not an instruction in RISC-V to compare two float numbers and then branch directly. We can use two steps to complete this function.
  - ✓ 1. Compare two float numbers and set an integer register as 1 or 0.
  - ✓ 2. Compare the register and a specific value (0 or 1) to check whether to branch or not.



feq.d t1, f1, f2	Floating EQuals (64 bit): if f1 = f2, set t1 to 1, else set t1 to 0
feq.s t1, f1, f2	Floating EQuals: if f1 = f2, set t1 to 1, else set t1 to 0
fle.d t1, f1, f2	Floating Less than or Equals (64 bit): if f1 <= f2, set t1 to 1, else set t1 to 0
fle.s t1, f1, f2	Floating Less than or Equals: if f1 <= f2, set t1 to 1, else set t1 to 0
flt.d t1, f1, f2	Floating Less Than (64 bit): if f1 < f2, set t1 to 1, else set t1 to 0
flt.s t1, f1, f2	Floating Less Than: if f1 < f2, set t1 to 1, else set t1 to 0

- Tip 2: When comparing two float numbers, they should both be single-precision or double-precision.

```
# Piece 7-5
.include "macro_print_str.asm"
.data
        fd1: .float    1.0
        dd1: .double       2.0
.text
        la t0, fd1
        flw ft0, (t0)
        la t0, dd1
        fld ft1, (t0)

        fmv.s fa0, ft0
        li a7, 2                    # print fd1
        ecall

        ##complete code here##
        li t2, 1
        beq t1, t2, printLe         # if(t1 == 1)
        j printGt
printLe:
        print_string("  LessOrEqual  ")
        j printSecondData
printGt:
        print_string("  LargerThan  ")

printSecondData:
        fmv.d fa0, ft1
        li a7, 3                    # print dd1
        ecall
        end
```



```
-2.0  LargerThan  -80.0
-- program is finished running (0) --

1.0  LessOrEqual  2.0
-- program is finished running (0) --
```

Practice 3

- Calculate the value of e from the infinite series:
  - Input a double-precision float number which represents a precision threshold.
  - Your program should terminate when the difference between two successive iterations is smaller than the precision threshold.
  - Print the value of e (as double-precision float).

$$\sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \cdots$$

Practice 4

- Given a single-precision float number 'x' and a positive integer 'r’. Round up ‘x’ to a number which keeps 'r' digits after the decimal point.

  For example, suppose 'x' is 1.5671

  if 'r' is 2, print 1.57

  if 'r' is 0, print 2

  if 'r' is 3, print 1.567