9 Floating-Point

9.1 Objectives

After completing this lab, you will:

- Understand Floating-Point Number Representation (IEEE 754 Standard)
- Understand the MIPS Floating-Point Unit
- Write Programs using the MIPS Floating-Point Instructions
- Write functions that have floating-point parameters and return floating-point results

9.2 Floating-Point Number Representation

Floating-point numbers have the following representation:

S	E = Exponent	F = Fraction

The Sign bit **S** is zero (positive) or one (negative).

The Exponent field **E** is 8 bits for single-precision and 11 bits for double-precision. The exponent field is biased. The **Bias** is 127 for single-precision and 1023 for double-precision.

The Fraction field **F** is 23 bits for single-precision and 52 bits for double-precision. Floating-point numbers are normalized (except when **E** is zero). There is an implicit **1.** (not stored) before the fraction **F**. Therefore, the value of a normalized floating-point number is:

$$\textbf{Value} = \pm (1.F)_2 \times 2^{E - Bias}$$

The MARS simulator has a floating-point representation tool that illustrates single-precision floating-point numbers. Go to **Tools → Floating Point Representation**, and open the window, shown in Figure 9.1.

Now use the tool to check the binary format and the decimal value of floating-point numbers.

For example, the decimal value of: **0 10000001 10110100000000000000000** is **6.75**.

Similarly, the 32-bit representation of: **-2.7531** is **1 10000000 01100000011001011001010**.

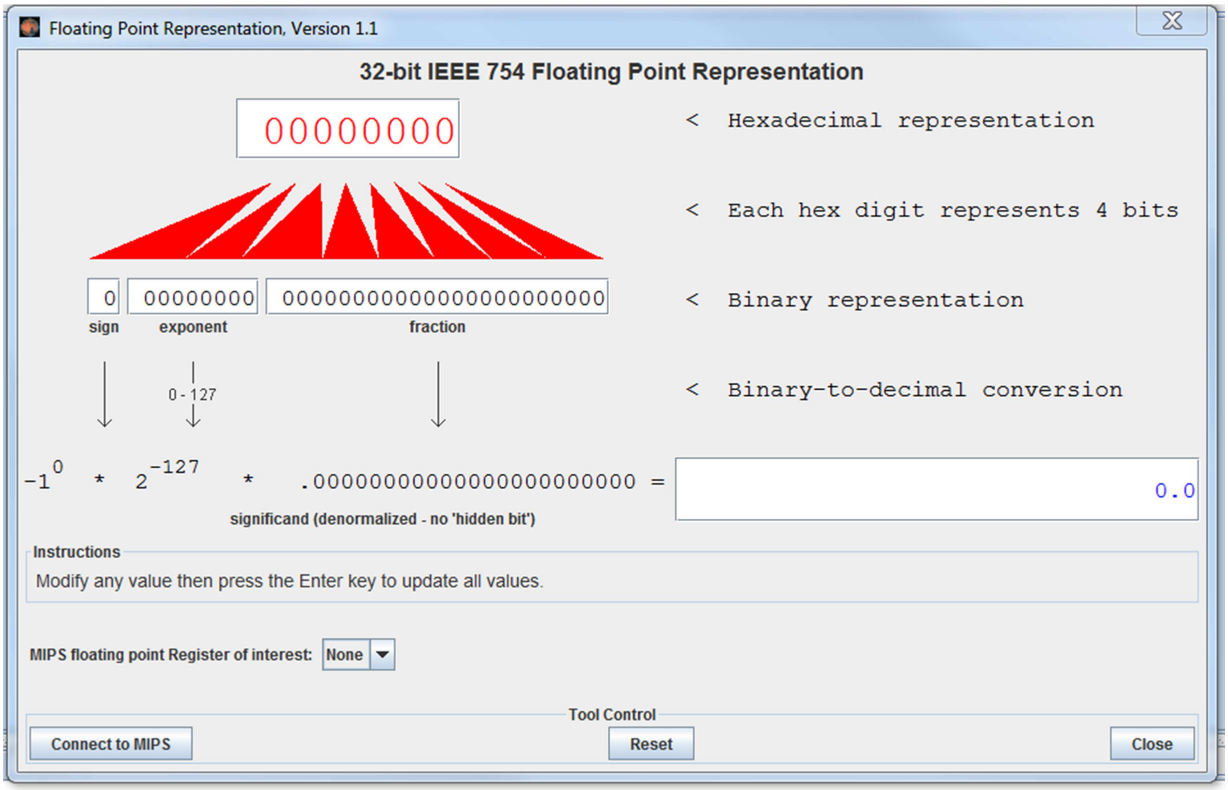
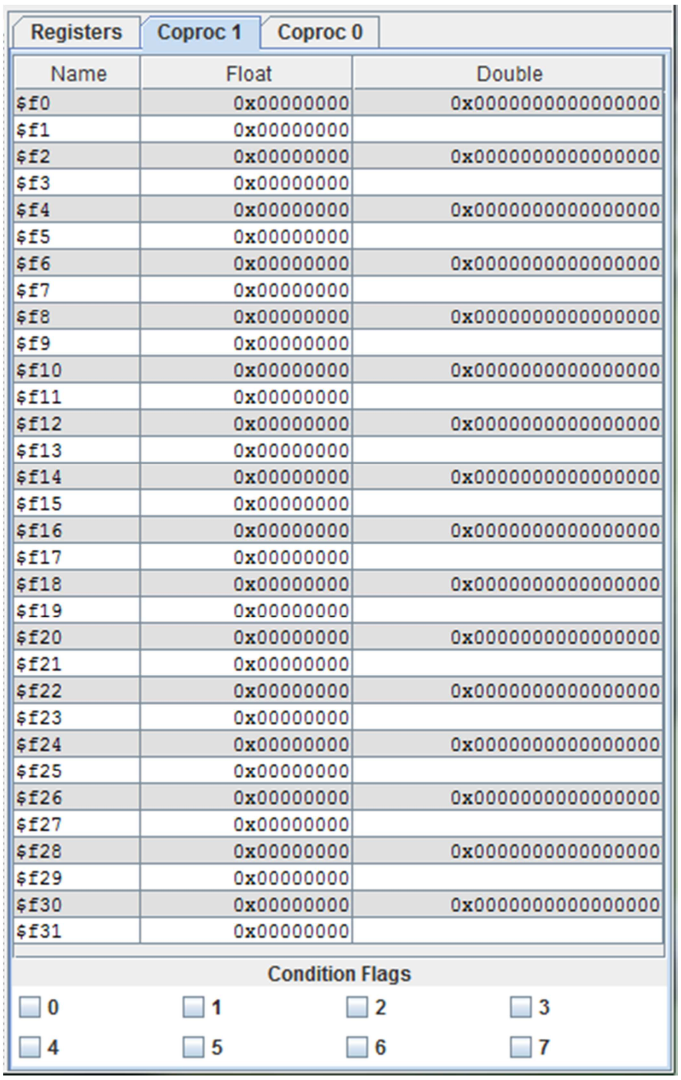


Figure 9.1: Floating-Point Representation tool supported by MARS

9.3 MIPS Floating-Point Registers

The floating-point unit (called coprocessor 1) has 32 floating-point registers. These registers are numbered as **`$f0`**, **`$f1`**, …, **`$f31`**. Each register is 32 bits wide. Thus, each register can hold one single-precision floating-point number. How can we use these registers to store 64-bit double-precision floating-point numbers? The answer is that the 32 single-precision registers are grouped into 16 double-precision registers. The double-precision number is stored in an even-odd pair of registers, but we only refer to the even-numbered register. For example, when we store a double-precision number in **`$f0`**, it is actually stored in registers **`$f0`** and **`$f1`**.

In addition, there are 8 condition flags, numbered from 0 to 7. These condition flags are used by floating-point compare and branch instructions. These are shown in Figure 9.2.



Registers | Coproc 1 | Coproc 0

Name	Float	Double
$f0	0x00000000	0x0000000000000000
$f1	0x00000000	
$f2	0x00000000	0x0000000000000000
$f3	0x00000000	
$f4	0x00000000	0x0000000000000000
$f5	0x00000000	
$f6	0x00000000	0x0000000000000000
$f7	0x00000000	
$f8	0x00000000	0x0000000000000000
$f9	0x00000000	
$f10	0x00000000	0x0000000000000000
$f11	0x00000000	
$f12	0x00000000	0x0000000000000000
$f13	0x00000000	
$f14	0x00000000	0x0000000000000000
$f15	0x00000000	
$f16	0x00000000	0x0000000000000000
$f17	0x00000000	
$f18	0x00000000	0x0000000000000000
$f19	0x00000000	
$f20	0x00000000	0x0000000000000000
$f21	0x00000000	
$f22	0x00000000	0x0000000000000000
$f23	0x00000000	
$f24	0x00000000	0x0000000000000000
$f25	0x00000000	
$f26	0x00000000	0x0000000000000000
$f27	0x00000000	
$f28	0x00000000	0x0000000000000000
$f29	0x00000000	
$f30	0x00000000	0x0000000000000000
$f31	0x00000000	

Condition Flags

0 1 2 3

4 5 6 7

Figure 9.2: MIPS Floating-Point Registers and Condition Flags

9.4 MIPS Floating-Point Instructions

The FPU supports several instructions including floating-point load and store, floating-point arithmetic operations, floating-point data movement instructions, convert, and branch instructions. We start this section with the floating-point load and store instructions. These instructions load into or store a floating-point register. However, they use the same base-displacement addressing mode used with integer instructions. Notice that the base address register is an integer (not a floating-point) register.

Instruction	Example	Meaning
`lwc1` or `l.s`	`lwc1 $f1,0($sp)`	Load a word from memory to a single-precision floating-point register: `$f1 = MEM[$sp]`
`ldc1` or `l.d`	`ldc1 $f2,8($t1)`	Load a double word from memory to a double-precision register: `$f2 = MEM[$t1+8]`

Instruction	Example	Meaning
`swc1 or s.s`	`swc1 $f5,4($t2)`	Store a single-precision floating-point register in memory: `MEM[$t2+4] = $f5`
`sdc1 or s.d`	`sdc1 $f6,16($t3)`	Store a double-precision floating-point register in memory: `MEM[$t3+16] = $f6`

The floating-point arithmetic instructions are listed next. The `.s` extension is used for single-precision arithmetic instructions, while the `.d` is used for double-precision instructions.

Instruction	Example	Meaning
`add.s`	`add.s $f0,$f2,$f4`	`$f0 = $f2 + $f4 (single-precision)`
`add.d`	`add.d $f0,$f2,$f4`	`$f0 = $f2 + $f4 (double-precision)`
`sub.s`	`sub.s $f0,$f2,$f4`	`$f0 = $f2 - $f4 (single-precision)`
`sub.d`	`sub.d $f0,$f2,$f4`	`$f0 = $f2 - $f4 (double-precision)`
`mul.s`	`mul.s $f0,$f2,$f4`	`$f0 = $f2 × $f4 (single-precision)`
`mul.d`	`mul.d $f0,$f2,$f4`	`$f0 = $f2 × $f4 (double-precision)`
`div.s`	`div.s $f0,$f2,$f4`	`$f0 = $f2 / $f4 (single-precision)`
`div.d`	`div.d $f0,$f2,$f4`	`$f0 = $f2 / $f4 (double-precision)`
`sqrt.s`	`sqrt.s $f0, $f2`	`Square root (single-precision)`
`sqrt.d`	`sqrt.d $f0, $f2`	`Square root (double-precision)`
`abs.s`	`abs.s $f0, $f2`	`Absolute value (single-precision)`
`abs.d`	`abs.d $f0, $f2`	`Absolute value (double-precision)`
`neg.s`	`neg.s $f0, $f2`	`Negative value (single-precision)`
`neg.d`	`neg.d $f0, $f2`	`Negative value (double-precision)`

The data movement instructions move data between general-purpose and floating-point registers, or between floating-point registers.

Instruction	Example	Meaning
`mfc1`	`mfc1 $t0, $f2`	Move data from a floating-point register to a general-purpose register.
`mtc1`	`mfc1 $t0, $f2`	Move data from a general-purpose register to a floating-point register.
`mov.s`	`mov.s $f0, $f1`	Move single-precision data between two floating-point registers.
`mov.d`	`mov.d $f0, $f2`	Move double-precision data between two floating-point registers (move even-odd pair of registers).

The convert instructions convert the format of data in floating-point registers. Three data formats are supported: **.s** = single-precision float, **.d** = double-precision, and **.w** = integer word.

Instruction	Example	Meaning
`cvt.s.w`	`cvt.s.w $f0,$f2`	`$f0` = convert `$f2` from word to single-precision
`cvt.s.d`	`cvt.s.d $f0,$f2`	`$f0` = convert `$f2` from double to single-precision
`cvt.d.w`	`cvt.d.w $f0,$f2`	`$f0` = convert `$f2` from word to double-precision
`cvt.d.s`	`cvt.d.s $f0,$f2`	`$f0` = convert `$f2` from single to double-precision
`cvt.w.s`	`cvt.w.s $f0,$f2`	`$f0` = convert `$f2` from single-precision to word
`cvt.w.d`	`cvt.w.d $f0,$f2`	`$f0` = convert `$f2` from double-precision to word
`ceil.w.s`	`ceil.w.s $f0,$f2`	`$f0` = Integer ceiling of single-precision float in `$f2`
`ceil.w.d`	`ceil.w.d $f0,$f2`	`$f0` = Integer ceiling of double-precision float in `$f2`
`floor.w.s`	`floor.w.s $f0,$f2`	`$f0` = Integer floor of single-precision float in `$f2`
`floor.w.d`	`floor.w.d $f0,$f2`	`$f0` = Integer floor of double-precision float in `$f2`
`trunc.w.s`	`trunc.w.s $f0,$f2`	`$f0` = Truncate single-precision float in `$f2`
`trunc.w.d`	`trunc.w.d $f0,$f2`	`$f0` = Truncate double-precision float in `$f2`

The floating-point compare instructions compare floating-point registers for equality, less than, and less than or equal. The FP compare instructions set the condition flags **0** to **7** to true (1) or false(0).

Instruction	Example	Meaning
`c.eq.s`	`c.eq.s $f2,$f3`	if `($f2 == $f3)` set flag **0** to true else false
`c.eq.d`	`c.eq.s 3,$f4,$f6`	Compare equal double-precision. Result in flag **3**
`c.lt.s`	`c.eq.s 4,$f5,$f8`	if `($f5 < $f8)` set flag **4** to true else false
`c.lt.d`	`c.lt.d 7,$f4,$f6`	Compare less-than double. Result in flag **7**
`c.le.s`	`c.le.s $f10,$f11`	if `($f10 <= $f11)` set flag **0** to true else false
`c.le.d`	`c.le.d $f14,$f16`	Compare less or equal double. Result in flag **0**

The floating-point branch instructions (**bc1t** and **bc1f**) branch to the target address based on the value of the specified condition flag (true or false).

Instruction	Example	Meaning
bc1t	**bc1t label**	Branch to **label** if condition flag **0** is true
bc1t	**bc1t 1, label**	Branch to **label** if condition flag **1** is true
bc1f	**bc1f label**	Branch to **label** if condition flag **0** is false
bc1f	**bc1f 4, label**	Branch to **label** if condition flag **4** is false

9.5 System Call Services for Floating-Point Numbers

The MARS tool provides the following **syscall** service numbers (passed in **$v0**) to print and read single-precision and double-precision floating-point numbers:

Service	$v0	Arguments	Result
Print Float	**2**	**$f12** = float to print	
Print Double	**3**	**$f12** = double to print	
Read Float	**6**		Float is returned in $f0
Read Double	**7**		Double is returned in **$f0**

9.6 MIPS Floating-Point Register Usage Convention

Compilers follow the MIPS register usage convention when translating functions and procedures into MIPS assembly-language code. The following table shows the MIPS software convention for floating-point registers. Not following the MIPS software usage convention can result in serious bugs when passing parameters, getting results, or using registers across function calls.

Registers	Usage
$f0 - $f3	Floating-point procedure results
$f4 - $f11	Temporary floating-point registers, NOT preserved across procedure calls
$f12 - $f15	Floating-point parameters, NOT preserved across procedure calls. Additional floating-point parameters should be pushed on the stack.
$f16 - $f19	More temporary registers, NOT preserved across procedure calls.
$f20 - $f31	Saved floating-point registers. Should be preserved across procedure calls.

9.7 In-Lab Tasks

1. Convert by hand the number **-123456789** into its 32-bit single-precision binary representation, and then use the floating-point representation tool presented in Section 9.2 to verify your answer. Show your work for a full mark.
2. Convert by hand the floating-point number **1 10010100 10011000001100000000000** (shown in binary) into its corresponding decimal value, and then use the floating-point representation tool presented in Section 9.2 to verify your answer. Show your work for a full mark.
3. Trace the following program by hand to determine the values of registers **$f0** thru **$f9**. Notice that **array1** and **array2** have the same elements, but in a different order. Comment on the sums of **array1** and **array2** elements computed in registers **$f4** and **$f9**, respectively. Now use the MARS tool to trace the execution of the program and verify your results. What conclusion can be made from this exercise?

```
.data
  array1: .float 5.6e+20, -5.6e+20, 1.2
  array2: .float 1.2, 5.6e+20, -5.6e+20
.text
  la      $t0, array1
  lwc1    $f0, 0($t0)
  lwc1    $f1, 4($t0)
  lwc1    $f2, 8($t0)
  add.s   $f3, $f0, $f1
  add.s   $f4, $f2, $f3
  la      $t1, array2
  lwc1    $f5, 0($t1)
  lwc1    $f6, 4($t1)
  lwc1    $f7, 8($t1)
  add.s   $f8, $f5, $f6
  add.s   $f9, $f7, $f8
```

4. Write an interactive program that inputs an integer **sum** and an integer **count**, computes, and displays the **average = (float) sum / (float) count** as a single-precision floating-point number. Hint: use the proper convert instruction to convert **sum** and **count** from integer word into single-precision float.
5. Write an interactive program that inputs the coefficient of a quadratic equation, computes, and displays the roots of the quadratic equation. All input, computation, and output should be done using double-precision floating-point instructions and registers. The program should handle the case of complex roots and displays the results properly.

6. Square Root Calculation: Newton's iterative method can be used to approximate the square root of a number **`x`**. Let the initial **`guess`** be **`1`**. Then each new **`guess`** can be computed as follows:

```
guess = ((x/guess) + guess) / 2;
```

Write a function called **`square_root`** that receives a double-precision parameter **`x`**, computes, and returns the approximated value of the square root of **`x`**. Write a loop that repeats 20 times and computes 20 **`guess`** values, then returns the final **`guess`** after 20 iterations. Use the MIPS floating-point register convention (Section 9.6) to pass the parameter **`x`** and to return the function result. All computation should be done using double-precision floating-point instructions and registers. Compare the result of the **`sqrt.d`** instruction against the result of your **`square_root`** function. What is the error in absolute value?

9.8 Bonus Problems

7. The sine function can be approximated by the following series:

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \quad \text{for all } x$$

Write a function that computes the sine of a parameter **`x`**. Use the MIPS floating-point register convention (Section 9.6) to pass the parameter **`x`** and to return the function result. All computation should be done using double-precision floating-point instructions and registers. Limit your computation to the first 20 terms of the series.

8. Converting a string into a floating-point number.

Write a function to convert a string, such as: **`"-13.232e-5"`** into a double-precision floating-point number. The address of the string should be passed in register **`$a0`**. The function should return the double-precision floating-point number in `$f0`. Conversion should terminate if the end of the string is reached (NULL byte), or an invalid character is encountered, such as a space, comma, etc.