Computer Architecture - Lab 7

Floating Point Arithmetic on MIPS

May 10, 2019

1 Instruction

MIPS chips use the IEEE 754 floating point standard, both the 32 bit and the 64 bit versions. However these notes cover only the 32 bit instructions. The 64 bit versions are similar.

This lab topics:

- Floating point registers
- Loading and storing floating point registers
- Single and (some) double precision arithmetic
- Data movement instructions
- Reading and writing floating point

SPIM Settings for this chapter: set SPIM to allow pseudoinstructions, disable branch delays, and disable load delays.

MIPS Floating Point

Floating point on MIPS was originally done in a separate chip called coprocessor 1 also called the FPA (Floating Point Accelerator). Modern MIPS chips include floating point operations on the main processor chip. But the instructions sometimes act as if there were still a separate chip.

MIPS has 32 single precision (32 bit) floating point registers.

- The registers are named `$f0 - $f31`
- `$f0` is not special (it can hold any bit pattern, not just zero).
- Single precision floating point load, store, arithmetic, and other instructions work with these registers.
- Floating point instructions cannot use general purpose registers.
- Only floating point instructions may be used with the floating point registers.

Double Precision

MIPS also has hardware for double precision (64 bit) floating point operations. For this, it uses pairs of single precision registers to hold operands. There are 16 pairs, named `$f0, $f2, ... , $f30`. Only the even numbered register is specified in a double precision instruction; the odd numbered register of the pair is included automatically.

Some MIPS processors allow only even-numbered registers (`$f0, $f2,...`) for single precision instructions. However SPIM allows you to use all 32 registers in single precision instructions. These notes follow that usage.

Single Precision Load

Actual hardware has a delay between a load instruction and the time when the data reaches the register. The electronics of main memory handles all bit patterns in the same way, so there is the same delay no matter what the bit patterns represent.

In SPIM there is an option that disables the load delay. For this chapter, disable the load delay. (Floating point is tricky enough already).

Loading a single precision value is done with a pseudoinstruction:

```
l.s   fd,addr     # load register fd from addr
                  # (pseudoinstruction)
```

This instruction loads 32 bits of data from address `addr` into floating point register `$fd` (where `$fd` is `$f0, $f1, ... $f31`. Whatever 32 bits are located at `addr` are copied into `fd`. If the data makes no sense as a floating point value, that is OK for this instruction. Later on the mistake will be caught when floating point operations are attempted.

Single Precision Store

Sometimes the floating point registers are used as temporary registers for integer data. For example, rather than storing a temporary value to memory, you can copy it to an unused floating point register. This is OK, as long as you don't try to do floating point math with the data.

There is a single precision store pseudoinstruction:

```
s.s   fd,addr     # store register fd to addr
                  # (pseudoinstruction)
```

Whatever 32 bits are in `fd` are copied to `addr`

In both of these pseudoinstructions the address addr can be an ordinary symbolic address, or an indexed address.

Floating Point Load Immediate

There is a floating point `load immediate` pseudoinstruction. This loads a floating point register with a constant value that is specified in the instruction.

```
li.s   fd,val     # load register $fd with val
                  # (pseudoinstruction)
```

Here is a code snippet showing this:

```
li.s    $f1,1.0             # $f1 = constant 1.0
li.s    $f2,2.0             # $f2 = constant 2.0
li.s    $f10,1.0e-5         # $f10 = 0.00001
```

Example Program 1

Here is a program that exchanges (swaps) the floating point values at `valA` and `valB`. Notice how the two floating point values are written. The first in the ordinary style; the second in scientific notation.

```
## swap.asm
##
## Exchange the values in valA and valB

        .text
        .globl  main

main:
        l.s     $f0,valA        # $f0  <-- valA
        l.s     $f1,valB        # $f1  <-- valB
        s.s     $f0,valB        # $f0  --> valB
        s.s     $f1,valA        # $f1  --> valA

        li      $v0,10          # code 10 == exit
        syscall                 # Return to OS.

        .data
valA:   .float  8.32            # 32 bit floating point value
valB:   .float  -0.6234e4       # 32 bit floating point value
                                # small 'e' only
```

Full-Word Aligned

However if you want to perform floating point arithmetic, then the floating point number must be in a floating point register.

The previous program exchanged the bit patterns held at two memory locations. It could just as easily been written using general purpose registers since no arithmetic was done with the bit patterns in the registers:

```
## swap.asm
##
## Exchange the values in valA and valB

        .text
        .globl  main
```

```
main:
        lw      $t0,valA        # $t0  <-- valA
        lw      $t1,valB        # $t1  <-- valB
        sw      $t0,valB        # $t0  --> valB
        sw      $t1,valA        # $t1  --> valA

        li      $v0,10          # code 10 == exit
        syscall                 # Return to OS.

        .data
valA:   .float  8.32            # 32 bit floating point value
valB:   .float  -0.6234e4       # 32 bit floating point value
                                # small 'e' only
```

For both single precision load and store instructions (as with the general purpose load and store instructions) the memory address must be full-word aligned. It must be a multiple of four. Ordinarily this is not a problem. The assembler takes care of this.

Floating Point Services

To print a floating point value to the SPIM monitor, use service 2 (for single precision) or service 3 (for double precision). To read a floating point value from the user, use service 6 (for single precision) or service 7 (for double precision). These notes deal mostly with single precision.

Here is the complete list of SPIM exception handler services. Each I/O method uses a specific format for data. The methods for double use an even-odd pair of registers.

Service	Code in $v0	Arguments	Returned Value
print integer	1	$a0 == integer	
print float	2	$f12 == float	
print double	3	($f12, $f13) == double	
print string	4	$a0 == address of string	
read integer	5		$v0 ← integer
read float	6		$f0 ← float
read double	7		($f0, $f1) ← double
read string	8	$a0 == buffer address $a1 == buffer length	
allocate memory	9	$a0 == number of bytes	$v0 ← address
exit	10		

Mistake

Depending on the service, you may have to place arguments in other registers as well. The following example program prints out a floating point value. It first does this correctly (using system call 2). Then does it incorrectly uses the integer print service (system call 1). Of course, the 32 bits of the floating point value can be interpreted as an integer, so system call 2 innocently does what we asked it to do.

```
## print.asm
##
## Print out a 32 bit pattern, first as a float,
## then as an integer.

        .text
        .globl  main

main:
        l.s     $f12,val        # use the float as an argument
        li      $v0,2           # code 2 == print float
        syscall                 # (correct)

        li      $v0,4           # print
        la      $a0,lfeed       # line separator
        syscall

        lw      $a0,val         # use the float as a int
        li      $v0,1           # code 2 == print int
        syscall                 # (mistake)

        li      $v0,10          # code 10 == exit
        syscall                 # Return to OS.

        .data
val :   .float  -8.32           # floating point data
lfeed:  .asciiz "\n"
```

No Type Checking

This type of mistake often happens when programming in "C" where **type checking** is weak. Sometimes the wrong type can be passed to a function (such as `printf`) and odd things happen.

Compilers that keep track of the data types of values and that make sure that the correct types are used as arguments do `strong type checking`. Java is strongly typed. In assembly language type checking is largely absent.

Precision of Single Precision Floats

There are two things wrong: (1) the value -8.32 can not be represented exactly in binary, and (2) the last digit or two of the printed value are likely in error.

Single precision floats have (recall) only 24 bits of precision. This is the equivalent of 7 to 8 decimal digits. SPIM should have printed -8.3199999 to the window.

The 7 or 8 decimal digits of precision is much worse than most electronic calculators. It is usually unwise to use single precision floating point in programs. (But these chapters use it since the goal is to explain concepts, not to write production grade programs). Double precision has 15 or 16 decimal places of precision.

Single Precision Arithmetic

Instruction	Operation
abs.s fd,fs	$fd = \|$fs|
add.s fd,fs,ft	$fd = $fs + $ft
sub.s fd,fs,ft	$fd = $fs - $ft
mul.s fd,fs,ft	$fd = $fs * $ft
div.s fd,fs,ft	$fd = $fs / $ft
neg.s fd,fs	$fd = -$fs

Here are some single precision arithmetic instructions. Each of these corresponds to one machine instruction. There is a double precision version of each instruction that has a "d" in place of the "s". So `add.s` becomes `add.d` and corresponds to the machine code that adds two double precision registers.

The first instruction computes the absolute value (makes a positive value) of the value in register `$fs`

If the data in an operand register is illegal or an illegal operation is performed (such as division by zero) an exception is raised. The IEEE 754 standard describes what is done in these situations.

Data Movement

There are three instructions that move data between registers inside the processor:

These instructions merely copy bit patterns between registers. The pattern is not altered. With the mfc1 instruction, the contents of a floating point register is copied "as is" to a general purpose register. So a complicated calculation with integers can use float registers to hold intermediate results. (But the float registers cannot be used with integer instructions.) And a complicated calculation with floats can use general purpose registers the same way. (But the general purpose registers cannot be used with floating point instructions.)

Example Program 2

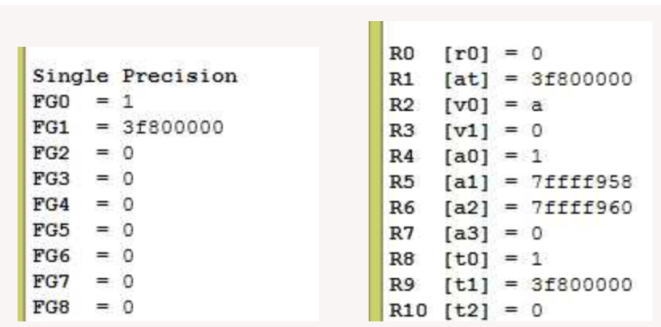
Here is a program that serves no practical purpose except as an example. The program loads a general purpose register with a two's complement 1 and copies that pattern to a floating point

Instruction	Operation
mov.s fd, fs	copy 32 bits from float register **$fs** to float register **$fd**
mtc1 rs, fd	move to coprocessor 1 Copy 32 bits from general register **$rs** to float register **$fd**. No data conversion is done. **Note:** the source register is **$rs** and the destination register is **$fd**, the reverse of the usual order.
mfc1 rd, fs	move from coprocessor 1 copy 32 bits from float register **$fs** to general register **$rd**. No data conversion is done.

register. Then it loads a floating point register with an IEEE 1.0 and moves that pattern to a general purpose register.

```
## Move data between the coprocessor and the CPU

        .text
        .globl main

main:
        li      $t0,1           # $t0 <-- 1
                                # (move to the coprocessor)
        mtc1    $t0,$f0         # $f0 <-- $t0

        li.s    $f1,1.0         # $f1 <-- 1.0
                                # (move from the coprocessor)
        mfc1    $t1,$f1         # $t1 <-- $f1

        li      $v0,10          # exit
        syscall
```

Here is what the SPIM registers contain after running the program:



The bit pattern `00000001` is the two's complement representation of one. It is now in registers `$t0` and `$f0`.

The bit pattern `3f800000` is the IEEE representation of 1.0. That pattern is now in registers `$f1` and `$t1`. (It is also in register `$at` which is used in implementing the pseudoinstruction `li.s`).

Example Program 3: Polynomial Evaluation

The example program computes the value of $ax^2 + bx + c$. It starts by asking the user for x:

```
## float1.asm -- compute ax^2 + bx + c for user-input x
##
## SPIM settings: pseudoinstructions: ON, branch delays: OFF,
##                load delays: OFF

        .text
        .globl main

        # Register Use Chart
        # $f0 -- x
        # $f2 -- sum of terms

main:   # read input
        la      $a0,prompt          # prompt user for x
        li      $v0,4               # print string
        syscall

        li      $v0,6               # read single
        syscall                     # $f0 <-- x

        # evaluate the quadratic
        l.s     $f2,a               # sum = a
        mul.s   $f2,$f2,$f0         # sum = ax
        l.s     $f4,bb              # get b
        add.s   $f2,$f2,$f4         # sum = ax + b
        mul.s   $f2,$f2,$f0         # sum = (ax+b)x = ax^2 +bx
        l.s     $f4,c               # get c
        add.s   $f2,$f2,$f4         # sum = ax^2 + bx + c

        # print the result
        mov.s   $f12,$f2            # $f12 = argument
        li      $v0,2               # print single
        syscall

        la      $a0,newl            # new line
        li      $v0,4               # print string
        syscall

        li      $v0,10              # code 10 == exit
        syscall                     # Return to OS.
```

```
##
##  Data Segment
##
        .data
a:      .float  1.0
bb:     .float  1.0
c:      .float  1.0

prompt: .asciiz "Enter x: "
blank:  .asciiz " "
newl:   .asciiz "\n"
```

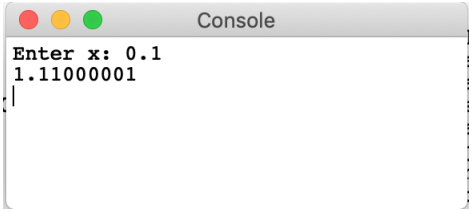
After the syscall the floating point value from the user is in `$f0`. The next section of the program does the calculation.

The assembler objects to the symbolic address "b" (because there is a mnemonic "b", for branch) so use "bb" instead.

The polynomial is evaluated from left to right. First $ax + b$ is calculated. Then that is multiplied by x and c is added in, giving $axx + bx + c$.

The value $x2$ is not explicitly calculated. This way of calculating a polynomial is called **Horner's Method**. It is useful to have in your bag of tricks.

As we have seen before, the results are not quite accurate. You would expect this because 0.1 cannot be represented accurately.



2 Exercises

1. **Exercise 1 — Arithmetic Expression (30pts)**
   Write a program that computes value of the following arithmetic expression for values of x and y entered by the user:

$$5.4xy - 12.3y + 18.23x - 8.23$$

2. **Exercise 2 — Harmonic Series (30pts)**
   Write a program that computes the sum of the first part of the harmonic series:

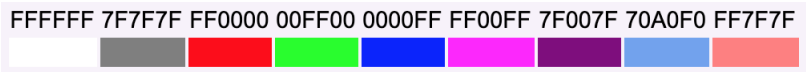
$$1/1 + 1/2 + 1/3 + 1/4 + ...$$

This sum gets bigger and bigger without limit as more terms are added in. Ask the user for a number of terms to sum, compute the sum and print it out.

3. **Exercise 3 — Web Page RGB Colors (20pts)**
Colors on a Web page are often coded as a 24 bit integer as follows:

$$RRGGBB$$

In this, each R, G, or B is a hex digit $0..F$. The R digits give the amount of red, the G digits give the amount of green, and the B digits give the amount of blue. Each amount is in the range 0..255 (the range of one byte). Here are some examples:



Another way that color is sometimes expressed is as three fractions 0.0 to 1.0 f or each of red, green, and blue. For example, pure red is $(1.0, 0.0, 0.0)$, medium gray is $(0.5, 0.5, 0.5)$ and so on.

Write a program that has a color number declared in the data section and that writes out the amount of each color expressed as a decimal fraction. Put each color number in 32 bits, with the high order byte set to zeros:

```
            .data
color:      .word  0x00FF0000      # pure red, (1.0, 0.0, 0.0)
```

For extra fun, write a program that prompts the user for a color number and then writes out the fraction of each component.

4. **Exercise 4 — Polynomial Evaluation (Horner's Method, yet Again!) (20 pts)**
Write a program that computes the value of a polynomial using Horner's method. The coefficients of the polynomial are stored in an array of single precision floating point values:

```
            .data
n:          5
a:          .float 4.3, -12.4, 6.8, -0.45, 3.6
```

The size and the values in the array may change. Write the program to initialize a sum to zero and then loop n times. Each execution of the loop, with loop counter j, does the following:

$$sum = sum * x + a[j]$$

To test and debug this program, start with easy values for the coefficients.

3 References

1. https://chortle.ccsu.edu/AssemblyTutorial/Chapter-31/ass31_1.html