Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Computer Architecture and Assembly Language Lab

Fall 2016

Lab 3

Arithmetic Operations (Basic and float) and Combinatorial Logic

Goal

You have been taught in Labs 1 and 2 how to read the input variables and print them as an output and how to write programs with loops. In this laboratory, you will learn how to do arithmetic operations and combinatorial logic. You will also gain knowledge about floating point operations. After some simple examples you have to write a program that requires knowledge about reading and printing, loop and branch, arithmetic and floating point operations.

Preparation

You must carefully read the second and third chapters of the textbook (Patterson and Hennessy, 5th edition) and also the SPIM instructions in Appendix A.10 in the textbook before attending the lab.

You should also look at the tutorial on ***QtSpim*** under "resources" on Sakai lab page. The other required materials will be uploaded on Sakai. The ***QtSpim*** program needed to do the lab exercises has been installed on the ECE 103 computers. To acquire this software for your personal computer, you can download it from **http://sourceforge.net/projects/spimsimulator/files/ or http://pages.cs.wisc.edu/~larus/spim.html .**

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

You have to study the lab manual carefully before attending the lab and prepare some pseudo codes for the exercises.

Introduction [1, 2]

As you have seen in the first and second Lab, ***Qtspim*** is a simulator that runs MIPS 32 programs. Clicking on the icon Qtspim.exe on the desktop screen will cause two windows to appear. The first one is a ***Qtspim*** window that you can run your program in and the second one is the console window that displays your results.

For running your program, you have to do the following:

1. Create a new text file, write your code and save it as **x.asm** or **x.txt.**
2. Click on the **File** button and choose **Reinitialize and Load File** and load your program
3. You can run your program all at once using **Run** icon from **Simulator** on the menu bar or use function key **F5.** For running your program line by line, you can use function key **F10.**
4. You can follow the results by looking at the registers in the left window.

Logical operations

A bitwise instruction operates on one or more bit patterns or binary numbers at the level of their individual bits. In MIPS we can do bitwise operations by using some instructions like **`not`**, **`and`**, **`or, nor`**. Below, you can see a sample that uses MIPS logical bitwise operations. Copy the code below to a text file named as **Ex1**, **reinitialize** and run. *Before* running the program, calculate the output on paper.

```
# Registers used:
# t0 - used to hold the original number
# t1 - used to hold NOT of the original number
.text
```

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

```
main:

li $t0, -106            # place the binary equivalent of -106 into
register $t0

not $t1, $t0            # place NOT of -106 into register t1


li $v0,10               # place the binary equivalent of 10 into
                        #register $v0

syscall                 # run syscall function for exit


# end of program
```

Look at the content of registers and compare them to your results.

Shift operations

A shift operation moves every bit of a register to the right or to the left (depending on which shift operation is used). For example a left shift would move each bit to the left by the number specified. The vacated bits will be filled by 0s. An example would be

```
original:         0011 0100 1101 1110
shift left 1:     0110 1001 1011 1100
shift left 2:     1101 0011 0111 1000
shift left 3:     1010 0110 1111 0000
shift left 4:     0100 1101 1110 0000
```

The instruction *sll $dest, $src, amount* can be used in MIPS to shift the contents of register $src by *amount* and store the result in the *$dest* register. *srl $dest, $src* can be used for a right shift. *sll* (shift left logical) is equivalent to multiplication, *srl* (shift right logical) is equivalent to division. Both instructions are R-type.

Floating point numbers

Besides integer operations, MIPS provides several special instructions and registers for reading and printing floating point numbers. Floating point numbers are not exact.

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

There are two different methods to store a floating point number in memory:

- **Single-Precision floating point format:** in this representation a floating point number occupies 32 bits. Single precision signed floating point numbers have one sign bit, 8 bits for exponent, which can be positive or negative and 23 bits for the fraction. You can see the organization of bits in the following example:

sign	exponent (8 bits)	fraction (23 bits)
0	01110101	00101100000000000000000

To calculate a number in this representation we use the following formula:

$$number = (-1)^{sign}(1 + \sum_{i=1}^{23} b_{-i}2^{-i}) \times 2^{(e-127)}$$

- **Double-Precision floating point format:** a double-precision floating point number is stored in 8 bytes (two registers), which is for covering a larger range of numbers. In this representation we have one sign bit, 11 bits for exponent and 52 bits for fraction. So a double precision floating point number occupies 64 bits (two registers or two memory rows).

All the operations regarding floating point numbers are using the registers **$f0-$f32**. After reading a floating point number from input, it will place it in the register **$f0**. For printing a floating point number, it has to be written to register **$f12**. In order to understand floating point number operations, run the following program. You can always convert an integer number to floating point and vice versa. To do this you need to use the instructions **cvt.s.w** and **cvt.w.s** in combination with move to coprocessor 1 **mtc1** and move from coprocessor 1 **mfc1** instructions. The instruction **cvt.s.w** converts the contents of register **$f0** from 2's complement integer to a single precision floating point number into **$f1**.

```
mtc1 $t0, $f0
cvt.s.w $f1, $f0
```

The instruction **cvt.w.s** converts a single precision floating point in register **$f1** to a 2's complement number in register **$f0**.

```
cvt.w.s $f0, $f1
mfc1 $t0, $f0
```

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Also, as the floating point registers are implemented in co-processor 1, we should move the content of their registers to register file registers like `$t0` using the **`mfc1`** instruction and the reverse action can be done by the **`mtc1`** instruction.

```
# float_division.asm-- A program that reads and writes a float number
# Registers used:
# $f0 - used to hold the input value.
# $f12 - used to hold the output value.
# $v0 - syscall parameter and return value.
.text
main:


li $v0, 6            # load syscall read_float into $v0. And after that
the input value will go to the $f0
syscall              # make the syscall.
mov.s $f1, $f0       # move the number to print into $f1.


li $v0, 6            # load syscall read_float into $v0.
syscall              # make the syscall.
mov.s $f2, $f0       # move the number to print into $f2.


div.s $f12, $f1, $f2  # divide two floating-points and move the
                      # quotient to $f12
## Print out $f12.
li $v0, 2              # load syscall print_float into $v0. And after
that the value in the $f12 will be ready for printing
```

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

```
syscall                 # make the syscall.

li $v0, 10              # syscall code 10 is for exit.

syscall                 # make the syscall.

# end of read_write_float.asm.
```

There is an instruction for floating point division which gives us the quotient of the division. However if we need to get the remainder of a division along with its quotient we should use two other instructions known as **mfhi** and **mflo**. After a division instruction, i.e. **div.s $f12, $f1, $f0**, the quotient will be moved to a register, named **LO** and the remainder to a register, named **HI**. Then, to get the quotient and remainder of the division we should use **mfhi** and **mflo** instruction as follows:

```
div.s $f12, $f1, $f0      # divide two single precision floating

                          #point-numbers and place result in $f12
mflo $f2  # $f2 now contains the quotient of the
                          #division
mfhi $f3                  # $f3 now contains the remainder of the
                          #division
```

Here is a table of useful single precision floating point instructions you may use in yourassignments.

Table 1: instructions for using in floating point number

l.s $f0, 16(t0)	Load a single precision floating point number from a memory location into $f0
s.s $f0, 16(t0)	Store $f0 to a memory location single precision
li.s $f0, 1.5	Load immediate number single precision in $f0
Arithmetic Operations	
abs.s $f0, $f1	Return the absolute value
add.s $f0, $f1, $f2	Add two float numbers, single precision
sub.s $f0, $f1, $f2	Subtract two float numbers in single precision
mul.s $f0, $f1, $f2	Multiply two float numbers, $f0=$f1*$f2
div.s $f0, $f1, $f2	Divide two float numbers, $f0=$f1/$f2
neg.s $f0, $f1	Negate the float number

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Move operations	
mov.s $f0, $f1	copy $f1 to $f0 in single precision
Mfc1 $t0, $f0	copy $f0 to $t0
Mtc1 $t0, $f0	copy $t0 to $f0
Branch operations	
c.lt.s $f0,$f1	set condition bit in coprocessor 1 as true if $f0 < $f1, else set it as false. The flag will stay until set or cleared next time
c.le.s $f0,$f1	set condition bit as true if $f0 <= $f1, else clear it
bc1t L1	branch to L1 if the condition bit is true
bc1f L1	branch to L1 if the condition bit is false

Using double precision floating point numbers is not very different from single precision numbers. The main difference is the change of .s at the end of each instruction to .d. The second difference and most important is that the floating point (FP) registers are used in pairs. Every double precision number when stored in the FP registers uses the $rdest and $rdest+1, e.g. if there is a double precision number in register $f8 it will also occupy register $f9. The following table has the equivalent double precision instructions of Table 1 for double precision.

Table 2: instructions for using in double floating point number

l.d $f0, 16(t0)	Load a double precision floating point number from a memory location into $f0
s.d $f0, 16(t0)	Store $f0-$f1 to a memory location double precision
li.d $f0, 1.5	Load immediate number double precision in $f0 and $f1
Arithmetic Operations	
abs.d $f0, $f2	Return the absolute value
add.d $f0, $f2, $f4	Add two float numbers, double precision ($f2, $f3 and $f4, $f5)
sub.d $f0, $f2, $f4	Subtract two float numbers in double precision
mul.d $f0, $f2, $f4	Multiply two float numbers, $f0, ($f1=$f2, $f3)*($f4, $f5)
div.d $f0, $f2, $f4	Divide two float numbers, $f0=$f2/$f4 that is $f0, $f1= ($f2, $f3)/($f4, $f5)
neg.d $f0, $f2	Negate the float number
Move operations	
mov.d $f0, $f2	copy $f2, $f3 to $f0, $f1 in double precision
Mfc1.d $t0, $f0	copy $f0,$f1 to $t0,$t1

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Branch operations	
`c.lt.d $f0,$f2`	`set condition bit in coprocessor 1 as true if $f0, $f1 < $f2, $f3 else set it as false. The flag will stay until set or cleared next time`
`c.le.d $f0,$f2`	`set condition bit as true if $f0, $f1 <= $f2, $f3 else clear it`

Assignments

Assignment 1

Write a simple program in assembly language to read three numbers from the input with the appropriate message, calculates the following logic function and prints the output to the console along with a message.
Suppose the three numbers are A, B, C the logic function is the following:

F=(A OR C)' AND (B AND C)'

Assignment 2

Write a program in assembly language that asks the user for two integers and calculates their product. You can only use shift, add or sub instructions to calculate the multiplication (not the mult instruction)

Assignment 3

A. Calculate the number represented by

sign	exponent (8 bits)	fraction (23 bits)
1	00011110	00110101111100110011000

Using the formula:

$$number = (-1)^{sign}(1 + \sum_{i=1}^{23} b\_i 2^{-i}) \times 2^{(e-127)}$$

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

B. Represent number 7.56_{10} by the binary string and fill the following table

sign	exponent (8 bits)	fraction (23 bits)

Compare the way that signed and unsigned numbers are stored.

Assignment 4

Write a program in assembly language to calculate the cube root of a float positive number N, using the Newton method for cube root. This method uses the recursive formula

$$x_{i+1} = (2 * x_i + N/x_i^2)/3$$

X_1 is the initial guess for the cube root and can be started from N or some other values say 1.
This procedure must be repeated until $| x_{i+1} - x_i | < \varepsilon$, where ε is some small number, you can use 0.00001 for this assignment.
Your program must request the user for a float number with an appropriate message and the print the cube root of that number.

.

Assignment 5

Write a program that calculates the Volume of a circular cone. The radius and height are arbitrary floating point values which are entered by the user and the output will be shown on the screen with an appropriate message. This is the way you can calculate the volume of a circular cone:

Volume= (1/3)*pi*radius^{2}height

Assignment 6

Write a program in assembly that calculates the sum of all the numbers of a floating point array that are bigger than the user input. Assume you defined an array with 10 floating point elements in your program and you ask the user for a real number. The result will be shown in the output with an appropriate message.
If there are no bigger numbers than the user input in your array provide an appropriate message.

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Use array **A=[1.35 2.67 3.566 4.56 5.98 9.43 12.34 15.54 23.87 34.33]** as the floating point array.

For example:
Please give a real number:
16.75
The answer is:
58.2

Lab report

Write a proper report using MS Word and include the results and discussions of your results in the report. **This report must include the pseudo-codes and the codes in MIPS assembly language**. It is better to add some screen shots from ***QtSpim*** showing intermediate results as the programs run.

The lab report must be handed at the start of the next lab (2 weeks). Please upload your code files on Sakai as well.

References

1. Patterson and Hennessy, "Computer Organization and Design: The Hardware / Software interface", 5th Edition.
2. Daniel J. Ellard, "MIPS Assembly Language Programming: CS50 Discussion and Project Book", September 1994.

Appendix

You can see here some useful MIPS assembly language instructions and their descriptions.

Table 2: Basic MIPS Instructions

Instruction	Operand	Description
li	des, const	Load the constant const into des.
lw	des, addr	Load the word at addr into des.
add	des, src1, src2	des gets src1 + src2.
sub	des, src1, src2	des gets src1 - src2.
move	des, src1	Copy the contents of src1 to des.
div	src1, reg2	Divide src1 by reg2, leaving the quotient in register lo and the

		remainder in register hi.
div	des, src1, src2	des gets src1 / src2.
mfhi	des	Copy the contents of the hi register to des.
mflo	des	Copy the contents of the lo register to des.
mthi	src1	Copy the contents of the src1 to hi.
mtlo	src1	Copy the contents of the src1 to lo.
mul	des, src1, src2	des gets src1 _ src2
mult	src1, reg2	Multiply src1 and reg2, leaving the low-order word in register lo and the high-order word in register hi.
rem	des, src1, src2	des gets the remainder of dividing src1 by src2.
syscall		Makes a system call(refer to table 2 for more information)

Table 3: Basic MIPS logical operation Instructions

Instruction	Operand	Description
abs	des, src1	des gets the absolute value of src1.
and	des, src1, src2	des gets the bitwise and of src1 and src2.
neg	des, src1 des	gets the negative of src1
nor	des, src1, src2	des gets the bitwise logical nor of src1 and src2.
not	des, src1	des gets the bitwise logical negation of src1.
or	des, src1, src2	des gets the bitwise logical or of src1 and src2.
xor	des, src1, src2	des gets the bitwise exclusive or of src1 and src2.
sll	des, src1, src2	des gets src1 shifted left by src2 bits.
srl	des, src1, src2	Right shift logical.

Table 4: SPIM syscall

Service	Code	Arguments
print_int	1	$a0
print_float	2	$f12
print_double	3	$f12
print_string	4	$a0
read_int	5	none
read_float	6	none

read_double	7	none
read_string	8	$a0 (address), $a1 (length)
sbrk	9	$a0(length)
exit	10	none

Table 5: Branch Instructions

Instruction	Operand	Description
b	lab	Unconditional branch to lab.
beq	src1, src2, lab	Branch to lab if src1 = src2.
bne	src1, src2, lab	Branch to lab if src1 ≠ src2.
bge	src1, src2, lab	Branch to lab if src1 ≥ src2.
bgt	src1, src2, lab	Branch to lab if src1 > src2.
ble	src1, src2, lab	Branch to lab if src1 ≤ src2.
blt	src1, src2, lab	Branch to lab if src1 < src2.
beqz	src1, lab	Branch to lab if src1 = 0.
bnez	src1, lab	Branch to lab if src1 ≠ 0.
bgez	src1, lab	Branch to lab if src1 ≥0.
bgtz	src1, lab	Branch to lab if src1 > 0.
blez	src1, lab	Branch to lab if src1 _ 0.
bltz	src1, lab	Branch to lab if src1 < 0.
bgezal	src1, lab	If src1 _ 0, then put the address of the next instruction into $ra and branch to lab.
bgtzal	src1, lab	If src1 > 0, then put the address of the next instruction into $ra and branch to lab.
bltzal	src1, lab	If src1 < 0, then put the address of the next instruction into $ra and branch to lab.