6

Arithmetic in MIPS

Objectives

After completing this lab you will:

- know how to do integer arithmetic in MIPS
- know how to do floating point arithmetic in MIPS
- know about conversion from integer to floating point and from floating point to integer.

Introduction

The definition of the R2000 architecture includes all integer arithmetic within the actual CPU. Floating point arithmetic is done by one of the four possible coprocessors, namely coprocessor number 1.

Integer arithmetic

Addition and subtraction are performed on 32 bit numbers held in the general purpose registers (32 bit each). The result is a 32 bit number itself. Two kinds of instructions are included in the instruction set to do integer addition and subtraction:

- instructions for signed arithmetic: the 32 bit numbers are considered to be represented in 2's complement. The execution of the instruction (addition or subtraction) may generate an overflow.
- instructions for unsigned arithmetic: the 32 bit numbers are considered to be in standard binary representation. Executing the instruction will never generate an overflow error even if there is an actual overflow (the result cannot be represented with 32 bits).

Multiplication and division may generate results that are larger than 32 bits. The architecture provides two special 32 bit registers that are the destination for multiplication and division instructions. These registers are called **`hi`** and **`lo`** as to indicate that they hold the higher 32 bits of the result and the lower 32 bits respectively. Special instructions are also provided to move data from these registers into the general purpose ones (**`$0`** to **`$31`**).

Instruction	Effect
`mfhi Rdest`	`Rdest` ← `hi`
`mflo Rdest`	`Rdest` ← `lo`

Instruction	Effect
`mthi Rsrc`	`hi` $\leftarrow$ `Rsrc`
`mtlo Rsrc`	`lo` $\leftarrow$ `Rsrc`

Two kinds of instructions are included in the instruction set to do integer multiplication and division:

- instructions for signed arithmetic: the 32 bit numbers are considered to be represented in 2's complement. An integer multiplication will never generate an overflow. Division may overflow. Note however that the instruction will not signal the overflow: this must be done in software.
- instructions for unsigned arithmetic: the 32 bit numbers are considered to be in standard binary representation. Executing the instruction will never generate an overflow error even if there is an actual overflow (this is the case for division).

Floating point arithmetic

Floating point arithmetic is performed by the MIPS' coprocessor 1. The coprocessor has 32 registers, numbered from 0 to 31 (their names are **`$f0`** to **`$f31`**). Each register is 32 bit wide. To accommodate doubles, registers are grouped together (0 with 1, 2 with 3,..., 30 with 31). To simplify things, floating-point operations use only even numbered registers. Arithmetic is performed on single-precision floating-point numbers (32 bit representation), and on double-precision floating-point numbers (64 bit representation). The odd numbered registers are used to hold the least significant 32 bits of a double precision number.

Floating point arithmetic resembles the IEEE-754 floating-point standard. A very brief description of number formats in this standard is given in Appendix A.

Floating-point addition, subtraction, multiplication and division may overflow. An overflow means that the exponent is too large to be represented in the exponent field.

The MIPS instruction set provides instructions that, beside floating-point operations, do floating-point comparisons, branching, load and store from/to memory and conversions between floating point formats and between integer and floating-point numbers. Unlike the integer unit where comparisons explicitly set some destination register, in the floating point unit, a comparison will *implicitly* set a condition flag. The condition flag can then be tested by branch instructions.

A summary of native instructions is presented in the table below. The `x` suffix for some instructions should be replaced by a `s` (e.g. `add.s`) to indicate the instruction should operate on single precision floating-point numbers, or by a (e.g. `add.d`) to indicate double precision operation.

Instruction	Comment
`mfc1 Rdest, FPsrc`	Move the content of floating-point register `FPsrc` to `Rdest`
`mtc1 Rsrc, FPdest`	Integer register `Rsrc` is moved to floating-point register `FPdest`
`mov.x FPdest, FPsrc`	Move floating-point register `FPsrc` to `FPdest`
`lwc1 FPdest, address`	Load word from `address` in register `FPdest`[a]
`swc1 FPsrc, address`	Store the content of register `FPsrc` at `address`[b]
`add.x FPdest, FPsrc1, FPsrc2`	Add single precision

Instruction	Comment
`sub.x FPdest, FPsrc1, FPsrc2`	Subtract `FPsrc2` from `FPsrc1`
`mul.x FPdest, FPsrc1, FPsrc2`	Multiply
`div.x FPdest, FPsrc1, FPsrc2`	Divide `FPsrc1` by `FPsrc2`
`abs.s FPdest, FPsrc`	Store the absolute value of `FPsrc` in `FPdest`
`neg.x FPdest, FPsrc`	Negate number in `FPsrc` and store result in `FPdest`
`c.eq.x FPsrc1, FPscr2`	Set the floating-point condition flag to true if the two registers are equal
`c.le.x FPsrc1, FPsrc2`	Set the floating-point condition flag to true if `FPsrc1` is less than or equal to `FPsrc2`
`c.lt.x FPsrc1, FPsrc2`	Set the floating-point condition flag to true if `FPsrc1` is less than `FPsrc2`
`bc1t label`	Branch if the floating-point condition flag is true
`bc1f label`	Branch if the floating-point condition flag is false
`cvt.x.w FPdest, FPsrc`	Convert the integer in `FPsrc` to floating-point
`cvt.w.x FPdest, FPsrc`	Convert the floating-point number in `FPsrc` to integer
`cvt.d.s FPdest, FPsrc`	Convert the single precision number in `FPsrc` to double precision and put the result in `FPdest`
`cvt.s.d FPdest, FPsrc`	Convert the double precision number in `FPsrc` to single precision and put the result in `FPdest`

a. A single-precision floating-point number has the same size as a word (32 bits). Inside coprocessor 1 a word will be treated as a single-precision number. A synthetic instruction is available to load a double from memory.
b. Stores in memory a single precision number. A synthetic instruction is available to store a double.

Laboratory 6: Prelab Exercise #1

Date ______________________ Section ______________________

Name ____

Integer Arithmetic in MIPS

During the prelab you will become familiar with integer arithmetic in MIPS. Since you have already used addition and subtraction in the previous labs, we will focus on issues related to the difference between signed and unsigned operations, and overflows. An *overflow* is a situation in which the result of an operation can not be represented using the assigned number of bits for that result.

Integer addition and subtraction

Unsigned integers are positive integers. With *n* bits for representation, the smallest unsigned is zero and the largest is $2^n - 1$. In a C/C++ program, integer variables which can not possibly be negative, should be declared as `unsigned int` (though you will see many programs where all integers are declared as plain `int`, i.e. signed integers)

Signed integers can be both positive and negative. With *n* bits, in 2's complement representation, the smallest integer is -2^{n-1} and the largest is $2^{n-1} - 1$

At the C/C++ programming level the difference between unsigned and signed integers is sometimes ignored. Note, however that declaring an integer variable that is always positive as `int` (when it should in fact be declared as `unsigned int`), reduces the range of that variable in half.

Ex 1:

```
// Assume that integers are 32 bit wide
{
unsigned int ex1;       // values for ex1 between 0 and 4,294,967,295
int ex2;    // values for ex2 between -2,147,483,648 and 2,147,483,647
}
```

If the variable `ex2` is always positive, then its possible values will be in the range 0 to 2,147,483,647, half of the range an unsigned integer has. ■

Q 1:

A negative number is represented as a bit pattern whose most significant bit is 1 (in 2's complement representation). The same bit pattern, when looked at as an unsigned integer, will be a large number. Fill out the following table (assume 8 bit number representation for simplicity)

Bit pattern	Value if bit pattern represents a signed integer	Value if bit pattern represents an unsigned integer
`00000000`		
`01111111`		
`10000000`		
`11111111`		

At the assembly language level the difference between signed and unsigned is more subtle. Most instructions that do arithmetic on signed numbers may overflow and the overflow will be signaled. Thus the trap handler[1] can take appropriate action to deal with the situation: ignoring an overflow may result in the computation of results that are completely wrong.

Instructions that do arithmetic on unsigned integers on the other hand, will not signal an overflow even if the operation itself yields a result which is not representable with the given number of bits for the result.

Q 2:

In what cases can the operation overflow? In the 'Addition' and 'Subtraction' columns of the following table use a 'Y' to indicate that the operation can overflow, and 'N' to indicate that the operation can not overflow.

Possible cases of overflow for integer addition and subtraction

Operand_1	Operand_2	Addition	Subtraction[a]
positive[b]	positive		
positive	negative[c]		
negative	positive		
negative	negative		
0	positive		
0	negative		
positive	0		
negative	0		

a. Operand_2 is subtracted from Operand_1
b. Strictly greater than 0
c. Strictly smaller than 0

1. More about trap handlers in Lab #7

Step 1

Write a program, called *lab6.1.c* which:

- declares two integer variables, `a` and `b`. The initial value for `a` and `b` is the maximum possible value for an integer (INT_MAX if you include the *limits.h* file)
- adds `a` and `b` and prints the result

Q 3:

Do you think the addition in *lab6.1.c* overflows? Explain.

Step 2

Compile and run the executable created from *lab6.1.c*.

`a`	`b`	Printed result

Q 4:

Is the result your program prints correct? Explain.

Most compilers generate code that does not signal integer overflows. An unsuspecting programmer may easily generate incorrect output without any error message to indicate it.

Step 3

Create the program *lab6.1.asm* based on the following description:

- in 'main' prompts the user to enter two integers; store them in **`$t0`** and **`$t1`**
- adds the two numbers using the native signed instruction; store the result in **`$t2`**
- prints the result

Run the program and fill out the following test plan. Wherever possible use numbers that would make the addition overflow. In the 'Overflows' column use a 'Y' to indicate the operation overflows and a 'N' to indicate it does not. In the 'Comment' column mark with a star those cases when an overflow error is actually reported.

Note: carefully choose the numbers you enter for test. The system function that reads an integer from the user (read_int) will truncate a very big number as to fit an integer register (32 bits). What you have in the register may be something you did not expect (try for example to enter a number like 8,589,934,593 and then look in the register where you store the number to see what has actually been stored there).

Test plan for *lab6.1.asm* **(signed addition)**

Operand_1	Operand_2	Printed result	Overflows	Comment
+2000000000	+2000000000		Y	
+	-			
-	+			
-	-			

Step 4

Create the program *lab6.2.asm* based on the following description:

- in 'main' prompts the user to enter two integers; store them in **`$t0`** and **`$t1`**
- subtracts the second number from the first using the native signed instruction; store the result in **`$t2`**
- prints the result

Run the program and fill out the following test plan. Wherever possible use numbers that would make the subtraction overflow. In the 'Overflows' column use a 'Y' to indicate the operation overflows and a 'N' to indicate it does not. In the 'Comment' column mark with a star those cases when an overflow error is actually reported.

Test plan for *lab6.2.asm* **(signed subtraction)**

Operand_1	Operand_2	Printed result	Overflows	Comment
+	+			
+	-			
-	+			
-	-			

Step 5

Testing unsigned addition and subtraction requires some attention. The SPIM simulator always prints the content of a general purpose register as a *signed* integer. The fact that a large valid unsigned integer is printed as a negative number may confuse you. Unsigned numbers larger or equal to 2^{n-1} (but smaller than 2^n) have the most significant bit 1. Therefore they will be interpreted as negative numbers by the printing function.

Create the program *lab6.3.asm* based on the following description:

- in 'main' prompts the user to enter two integers; store them in **`$t0`** and **`$t1`**
- adds the two numbers using the native unsigned instruction; store the result in **`$t2`**
- prints the result

Run the program and fill out the following test plan. Wherever possible use numbers that would make the addition overflow. In the 'Overflows' column use a 'Y' to indicate the operation overflows and a 'N' to indicate it does not. In the 'Comment' column mark with a star those cases when an overflow error is actually reported.

Test plan for *lab6.3.asm* (unsigned addition)

Operand_1	Operand_2	Printed result	Overflows	Comment
+	+			

Step 6

Create the program *lab6.4.asm* based on the following description:

- in 'main' prompts the user to enter two integers; store them in **`$t0`** and **`$t1`**
- subtracts the second number from the first using the native unsigned instruction; store the result in **`$t2`**
- prints the result

Run the program and fill out the following test plan. Wherever possible use numbers that would make the subtraction overflow. In the 'Overflows' column use a 'Y' to indicate the operation overflows and a 'N' to indicate it does not. In the 'Comment' column mark with a star those cases when an overflow error is actually reported.

Test plan for *lab6.4.asm* (unsigned subtraction)

Operand_1	Operand_2	Printed result	Overflows	Comment
+	+			
0	+			

Laboratory 6: Prelab Exercise #2

Date ______________________ Section ______________________

Name ____

Integer Arithmetic in MIPS (cont' d)

During this prelab exercise you will become familiar with integer multiplication and division in MIPS.

Integer multiplication

Integer multiplication can be done using both signed and unsigned numbers. The architecture specifies two special registers, **`hi`** and **`lo`** (32 bit each), which are the destination for integer multiplication and division.

Multiplying two n-bit unsigned integers may yield a result that requires $2 \cdot n$ bits to be represented.

Ex 1:

Assume 2 bit unsigned integers. Then the largest integer that can be represented using 2 bits is 3 (its binary representation is 11).

	Multiplication	Comment
Decimal	$3 \cdot 3 = 9$	
Binary	$11 \cdot 11 = 1101$	Result requires 4 bits for representation

■

For signed multiplication the two numbers to be multiplied are in 2's complement representation. The result is also in 2's complement representation. The result of a signed multiplication may require $2 \cdot n - 1$ bits for representation: there are $2 \cdot (n-1)$ bits for the magnitude and one bit for the sign.

Native instructions for multiplication

Instruction	Comment
`mult Rsrc1, Rsrc2`	Multiply the signed numbers in `Rsrc1` and `Rsrc2`. The higher 32 bits of the result go in register **`hi`**. The lower 32 bits of the result go in register **`lo`**
`multu Rsrc1, Rsrc2`	Multiply the unsigned numbers in `Rsrc1` and `Rsrc2`. The higher 32 bits of the result go in register **`hi`**. The lower 32 bits of the result go in register **`lo`**

Since both signed and unsigned integer multiplication never require more than 64 bits for the result, an overflow will never occur during integer multiplication using the native multiply instructions.

The virtual machine provides multiply instructions whose result is the size of a word (the destination register is some general purpose integer register). Since the destination register is only 32 bit wide, these synthetic instructions may overflow.

Synthetic instructions for multiplication

Instruction	Comment
`mul Rdest, Rsrc1, Rsrc2`	Multiply the signed numbers in `Rsrc1` and `Rsrc2`. The lower 32 bits of the result go in register `Rdest`.
`mulo Rdest, Rsrc1, Rsrc2`	Multiply the signed numbers in `Rsrc1` and `Rsrc2`. The lower 32 bits of the result go in register `Rdest`. Signal overflow
`mulou Rdest, Rsrc1, Rsrc2`	Multiply the unsigned numbers in `Rsrc1` and `Rsrc2`. The lower 32 bits of the result go in register `Rdest`. Signal overflow

Integer division

The result of an integer division is a quotient (stored in register **`lo`**) and a remainder (stored in register **`hi`**), both integer numbers.

- There is one complication related to integer division: the sign of the remainder. There are two approaches to this problem:
- follow the division theorem from mathematics
- use the computer science view

By dividing an integer *DD* called dividend, by a *positive* integer *DR* called divisor, the following conditions are true (the division theorem):

$$DD = DR \cdot Q + RM \qquad (1)$$
$$0 \le RM < DR \qquad (2)$$

where *Q* (the quotient) and *RM* (the remainder) are unique integers. Note that the remainder is always positive.

Ex 2:

What is the result of dividing -22 by 3?

$DD = -22$, $DR = 3$, $Q = -8$, $RM = +2$

Note that relation (1) could be satisfied by $Q' = -7$, $RM' = -1$. However this solution is not valid since it does not satisfy the condition that the remainder must be positive (2).

What is the result of dividing +22 by 3?

$DD = +22$, $DR = 3$, $Q = +7$, $RM = +1$

In this approach $-\left(\frac{22}{7}\right) \neq \left(\frac{-22}{7}\right)$. If this was implemented on computers, then programming would definitely be more fun than it is now. ■

The way division is usually implemented (the 'computer science way'), the following relations are true:

$$DD = DR \cdot Q + RM \qquad (3)$$
$$|RM| < |DR| \qquad (4)$$
$$sign(RM) = sign(DD) \qquad (5)$$

If the dividend and the divisor have the same sign then the quotient is positive; otherwise it is negative. A nonzero remainder has always the same sign as the dividend.

Q 1:

What are the results of the following integer divisions?

Dividend	Divisor	Quotient	Remainder
22	7		
-22	7		
22	-7		
-22	-7		

In MIPS, if one of the operands in a division is negative, then the remainder is **unspecified**. The SPIM simulator will return a result based on the conventions on the machine it is run on.

To obtain the correct result for division, extra steps need to be performed:

- convert both operand to positive integers
- perform the division
- set the result to its actual representation based on the initial signs of the dividend and divisor.

A signed division may overflow since the 2's complement representation of integers is asymmetric: there is one negative number more than positive numbers. Note however that the overflowing division does not signal the overflow. It is the compiler's or programmer's task to generate proper code that detects the overflow.

Dividing an integer by zero is an illegal operation. However, even if the divisor is zero, the division will report no error. It is again the compiler's or programmer's task to generate proper code that detects the illegal operation.

Native instructions for division

Instruction	Comment
`div Rsrc1, Rsrc2`	Divide the signed integer in `Rsrc1` by the signed integer in `Rsrc2`. The quotient (32 bits) goes in register **`lo`**. The remainder goes in register **`hi`**. Can overflow.

Native instructions for division

Instruction	Comment
`divu Rsrc1, Rsrc2`	Divide the unsigned integer in `Rsrc1` by the unsigned integer in `Rsrc2`. The quotient (32 bits) goes in register **`lo`**. The remainder goes in register **`hi`**. Does not overflow.

The virtual machine provides divide instructions whose result is stored in a general purpose register.

Synthetic instructions for division

Instruction	Comment
`div Rdest, Rsrc1, Rsrc2`	Divide the signed integer in `Rsrc1` by the signed integer in `Rsrc2`. Store the quotient (32 bits) in register `Rdest`. Can overflow.
`divu Rdest, Rsrc1, Rsrc2`	Divide the unsigned integer in `Rsrc1` by the unsigned integer in `Rsrc2`. Store the quotient in Rdest. Does not overflow.
`rem Rdest, Rsrc1, Rsrc2`	Divide the signed integer in `Rsrc1` by the signed integer in `Rsrc2`. Store the remainder[a] (32 bits) in register `Rdest`. Can overflow.
`remu Rdest, Rsrc1, Rsrc2`	Divide the signed integer in `Rsrc1` by the signed integer in `Rsrc2`. Store the remainder[a] (32 bits) in register `Rdest`. Does not overflow.

a. If one of the operands is negative then the remainder is undefined.

Step 1

Write a program, called *lab6.2.c* which:

- declares two integer variables, `a` and `b`. The initial value for `a` and `b` is the maximum possible value for an integer (INT_MAX if you include the *limits.h* file)
- multiplies `a` and `b` and prints the result

Q 2:

Do you think the multiplication in *lab6.2.c* overflows? Explain.

Step 2

Compile and run the executable created from *lab6.2.c*.

`a`	`b`	Printed result

Q 3:

Is the result your program prints correct? Explain.

Most compilers generate code that does not signal integer overflows. An unsuspecting programmer may easily generate incorrect output without any error message to indicate it.

Step 3

Create the program *lab6.5.asm* based on the following description:

- in 'main' prompts the user to enter two integers; store them in **`$t0`** and **`$t1`**
- multiplies the two numbers using the native signed instruction
- prints the result

Run the program and fill out the following test plan. Use the `print` command in the simulator to see the content of registers **`hi`** and **`lo`**. In the last two rows of the table enter the numbers that represent the largest possible signed integer and the smallest possible one respectively.

Test plan for *lab6.5.asm* **(signed multiplication): register's content**

Operand_1	Operand_2	Register **`hi`** (hexadecimal)	Register **`lo`** (hexadecimal)
`2`	`1`		
2	-1		
262,144 (=2^{18})	16,384 (=2^{14})		

Q 4:

In the second row of this test plan the register **`hi`** is all 1s. Why is that?

Step 4

Run the program again and fill out the next test plan. Use the same numbers you have used in the previous step.

Test plan for *lab6.5.asm* (signed multiplication): printed results

Operand_1	Operand_2	Expected result	Printed result

Q 5:

Some of the results your program prints are not correct. Why?

Step 5

Create the program *lab6.6.asm* based on the following description:

- in 'main' prompts the user to enter two integers; store them in **`$t0`** and **`$t1`**
- multiplies the two numbers using the native unsigned instruction

Run the program and fill out the following test plan.

Test plan for *lab6.6.asm* (unsigned multiplication): register's content

Operand_1	Operand_2	Register **`hi`** (hexadecimal)	Register **`lo`** (hexadecimal)
`2`	`1`		
2	-1		
262,144 (=2^{18})	16,384 (=2^{14})		

Q 6:

What numbers are in reality multiplied when the input numbers are 2 and -1? Remember that a negative integer will mean a large positive unsigned integer.

Number	Bit pattern	Value if bit pattern represents an unsigned integer
`2`		

Number	Bit pattern	Value if bit pattern represents an unsigned integer
`-1`		

Step 6

Create the program *lab6.7.asm* based on the following description:

- in 'main' prompts the user to enter two integers; store them in **`$t0`** and **`$t1`**
- divides the first number by the second one, using the native signed instruction
- prints the quotient and the remainder

Run the program and fill out the following test plan. Use small numbers that would generate a nonzero remainder. For the last row use a pair of numbers that would make the division overflow. Note that division by zero (which is an illegal operation) and overflow are different things. In the 'Comment' column mark with a star those cases where an error is reported.

Test plan for *lab6.7.asm* **(signed division)**

Operand_1	Operand_2	Printed Quotient	Printed remainder	Comment
+	+			
+	-			
-	+			
-	-			
	0			

Highlight those cells that contain incorrect results.

Step 7

Create the program *lab6.8.asm* based on the following description:

- in 'main' prompts the user to enter two integers; store them in **`$t0`** and **`$t1`**
- divides the first number by the second one, using the native unsigned instruction
- prints the quotient and the remainder

Run the program and fill out the following test plan. Use small numbers that would generate a nonzero remainder. For the last row use a pair of numbers that would make the division overflow. In the 'Comment' column mark with a star those cases where an error is reported.

Test plan for *lab6.7.asm* (unsigned division)

Operand_1	Operand_2	Printed Quotient	Printed remainder	Comment
+	+			
+	-			
-	+			
-1	+2			
-	-			
	0			

Highlight divisions where incorrect results are printed.

Q 7:

What numbers are in reality divided when the input numbers are -1 and +2?

Number	Bit pattern	Value if bit pattern represents an unsigned integer
`-1`		
`+2`		

Laboratory 6: Prelab Exercise #3

Date ____________________ Section ____________________________

Name ____

Floating Point Arithmetic in MIPS

During this exercise you will use floating-point instructions. Since the SPIM simulator does implement only partially the IEEE-754 standard, this exercise will ignore most issues related to the standard, like ways to handle overflows, underflows, denormalized representation, rounding, etc. The focus will be on floating point number representation.

Floating point numbers

The IEEE-754 standard reserves several bit patterns to have special meaning. In other words not all bit patterns represent some number.

Special bit patterns in IEEE-754

Sign bit	Exponent	Significand	Comment
`x`	`0..0`	`0..0`	Zero
`x`	`0..0`	`not all zeros`	Denormalized number
`0`	`1..1`	`0..0`	Plus infinity (+inf)
`1`	`1..1`	`0..0`	Minus infinity (-inf)
`x`	`1..1`	`not all zeros`	Not a Number (NaN)

Infinity does not mean mathematically infinite, rather something too big to be represented. An overflow can return an +inf or a -inf (though the standard also provides a mechanism to determine the correct result in case of overflows). Some operations on infinity return yet another infinity as a result.

Table 1: Some operations with infinity

Operation	Result	Comment
`x + ( inf)`	`inf`	x finite
`x - (+inf)`	`-inf`	x finite
`(+inf) + (+inf)`	`+inf`	
`(-inf) + (-inf)`	`-inf`	
`x * (+inf)`	`+inf if x>0, -inf otherwise`	x nonzero

There can be a positive zero if the sign bit is 0 and a negative zero (the sign bit is 1). Denormalized numbers are included in IEEE-754 to handle cases of exponent underflow (very small numbers).

A *NaN* (sometimes denoted by *nan*) is used to represent an indeterminate result. There are two kinds of NaNs, signaling and quiet: the actual bit pattern in the significand field is used to differentiate between them, and it is implementation-dependent. A signaling NaN can be used for instance for uninitialized variables: attempting to operate on a signaling NaN can cause a trap. Note that any operation on a signaling NaN will have as a result a quiet NaN. Operating on a quiet NaN simply returns another NaN without generating any exception.

Table 2: Some operations which produce a quiet NaN

Operation	Comment
`x + (NaN)`	Any operation on a quiet NaN (addition in this example)
`(+inf) + (-inf)`	
`0 * ( inf)`	
`0/0`	
`inf/inf`	
`x%0`	The remainder of division by 0
$\sqrt{x}$, `x < 0`	

Step 1

Create a program named *lab6.9.asm* which:

- declares the variables *Zero.s*, *PlusInf.s*, *MinusInf.s*, *PlusNaN.s*, *MinusNaN*, of size word, initialized with the bit patterns corresponding to zero, plus infinity, minus infinity, positive Not a Number, and negative Not a Number respectively, in single precision representation
- declares the variables *Zero.d*, *PlusInf.d*, *MinusInf.d*, *NaN.d*, initialized with the bit patterns corresponding to zero, plus infinity, minus infinity, and Not a Number respectively, in double precision
- loads these variables in floating-point registers, starting with **`$f0`**
- prints, starting with **`$f0`**, the contents of those registers where variables have been loaded; print a newline (\n) character after each value.

Run the program. Fill out the following table

Printed output when running *lab6.9.asm*

Variable name	Printed output

Printed output when running *lab6.9.asm*

Variable name	Printed output

Step 2

Create the program *lab6.10.asm* which:

- declares the same single precision variables as *lab6.9.asm*
- declares a float called *MyNum*, initialized to the first two digits of your Social Security Number
- performs an operation which uses *MyNum* and which generates infinity; print the result. Choose some operation from Table 1
- performs an operation that generates NaN and print the result; choose some operation from Table 2

Run the program. Fill out the following table

Printed output when running *lab6.10.asm*

MyNum	Operation implemented	Printed result

Q 1:

What is the bit pattern for the largest possible single precision floating point number? Write it down in hexadecimal.

Laboratory 6: Inlab

Date ______________________ Section ______________________

Name ____

Floating Point Arithmetic in MIPS (cont' d)

This inlab session will focus on doing floating-point arithmetic and on various conversions.

Step 1

During Lab #4 you created a program named *lab4.5.asm* which computes the factorial of an integer number. All operations were performed on integer numbers. You are to create a new program named *lab6.11.asm* which computes the factorial of an integer using floating point numbers:

- in 'main' prompt the user to enter an integer
- check if the number entered is negative: if it is negative, then print a message saying so and prompt the user again for a number
- call the procedure named 'FactorialSingle' whose parameter will be the number read from the user, converted to single precision floating-point; the procedure returns the factorial (a single precision floating-point) of that number
- use the MIPS convention for procedure call
- prints a message and the value returned by 'FactorialSingle'

Run *lab4.5.asm* and the new program to complete the following test plan. Use scientific notation for the output printed by *lab6.11.asm* (i.e. 3.141592e0).

Test plan for *lab6.11.asm*

Input number	*lab4.5.asm* output	*lab6.11.asm* output
`0`		
`5`		
`10`		
`15`		
`20`		
`40`		

Q 1:

Why some of the results printed by *lab4.5.asm* are negative?'

Q 2:

What are the maximum values of the input for which correct output is still printed?'

Integer	n =
Single precision floating-point	n =

Step 2

Using some other method, calculate the exact value of 20! On a UNIX platform you can use bc (the "arbitrary-precision arithmetic language"). If no software is available to you to do the job, then you will have to use the old-good pencil and paper method.

The exact value for 20!

```
20!=
```

Run again *lab6.11.asm* and write down the value printed for 20!

Value printed by *lab6.11.asm* for 20!

```
20!=
```

Q 3:

Why do the two values differ? '

Q 4:

How many digits are exact in the result printed by *lab6.11.asm*?

exact_digits =

Q 5:

Based on the answer to the previous question, can you make a rough estimation of the number of digits that should be printed for a floating-point single precision number?

digits_to_print =

Step 3

In step 1 you have used the integer to floating point conversion (`cvt.s.w`). Let's now try a conversion from float to integer. Modify *lab6.11.asm* (save the new program as *lab6.12.asm*):

- after printing the value returned by 'FactorialSingle', convert that value to an integer and print it too.

Run the program and complete the next test plan. Write down the value printed for the factorial as is, don't use scientific notation this time.

Test plan for *lab6.12.asm*

Input number	Floating-point output	Integer Output
0		
5		
10		
11		
12		
13		
14		
15		

Highlight those cases where the integer output is a number different than the floating-point output.

Q 6:

Do you think the conversion operation yields a signed integer or an unsigned one? Explain.

Q 7:

As you can see the conversion instruction does not signal any error even if the conversion itself results in a wrong value for the result. What is, in your opinion, the reason the architecture does not specify the conversion instructions should report errors? '

Q 8:

What would be the sequence of instructions that would emulate a signaling conversion? Do this only for the float to integer conversion.

Laboratory 6: Postlab

Date ______________________ Section ______________________

Name ____

Step 1

During the prelab exercises you have been, at the very least, annoyed by the fact that you could not correctly print unsigned integers. By using `print_int`, a large unsigned integer (larger or equal to 2^{31}) would print as a negative number. You now want to correct this by providing future users with a procedure that prints unsigned integers. You will write a procedure, named 'PrintUnsigned' which:

- receives the unsigned number to print in **`$a0`**
- converts the number to a null terminated ASCII string (the null character is 0x00)
- prints the string
- returns in $v0 the number of non-null characters in the printed ASCII string

Here is the C code for the integer to ASCII conversion function, `itoa()`. You will have to slightly modify it and use it in your implementation of 'PrintUnsigned'.

P.1:

```
/* code for itoa() from "The C Programming Language" by Kernigan and Richie
*/

void reverse(char *s)
{
       int c, i, j;

       for( i=0, j=strlen(s)-1; i<j; i++, j--) {
              c=s[i];
              s[i]=s[j];
              s[j]=c;
       }
}

void itoa(int n, char *s)
{
       int i, sign;

       if ((sign=n) < 0)
              n = -n;
       i=0;
       do {
              s[i++] = n%10 + '0';
       } while((n/=10) > 0);
```

```
        if (sign < 0)
                s[i++] = '-';
        s[i] = '\0';
        reverse(s);
}
```

Hints:

- you will need to reserve space in 'PrintUnsigned' for the string you generate;
- make sure you use the appropriate (signed/unsigned) arithmetic functions

Step 2

Create a program called *lab6.13.asm* which you use to test 'PrintUnsigned', as follows:

- the program prints six times (in a loop) the value of a counter initialized to $2^{31} - 3$;
- at each iteration the program prints the counter using `print_int` and then the same counter using 'PrintUnsigned'
- at each iteration the counter is incremented by one using an unsigned operation

Run the program and complete the following test plan.

Test plan for *lab6.13.asm*

Counter	Printed by print_int	Printed by 'PrintUnsigned'

Step 3

Return to your lab instructor a copy of *lab6.13.asm* together with this postlab description. Ask your lab instructor whether copies of programs must be on paper (hardcopy), e-mail or both.