

## COE 181.1 Lab Report

**Lab Number:** Lab 5

**Lab Title:** INTEGER MULTIPLICATION AND DIVISION

**Student Name:** Janna Joyce E. Jumao-as

**Student Email:** jannajoyce.jumao-as@g.msuiit.edu.ph

**Student Phone Number:** 09309321199

### Objectives:

- Understand binary multiplication and division
- Understand the MIPS multiply and divide instructions
- Write MIPS programs that use integer multiplication and division

### Procedures:

**Task 1:** Write MIPS code to perform the following integer multiplications. What is the value of the LO and HI registers?

a)  $98765 \times 54321$  using the *multu* instruction

b)  $-98765 \times -54321$  using the *mult* instruction

- To access the results of the multiplication, use mflo and mfhi to move the values from the LO and HI registers into general-purpose registers.

**Task 2:** Write MIPS code to perform the following integer divisions. What is the value of the LO and HI registers?

a)  $98765 / 54321$  using the *divu* instruction

b)  $-98765 / -54321$  using the *div* instruction

- To access the results of the multiplication, use mflo and mfhi to move the values from the LO and HI registers into general-purpose registers.

**Task 3:** Factorial Calculation: Using the *mul* instruction, write a MIPS program that computes the factorial of a number *n* input by the user, and display the result on the screen. Run your code and record the maximum 32-bit factorial value that can be computed without errors.

- Ask the user to enter an input.
- Implement factorial loop for factorial calculation.
- Display overflow message since MIPS uses 32-bit registers only.

**Task 4:** The string-to-integer program presented in "Applications of Integer Multiply and Divide Instructions" section converts a string of decimal digits to an unsigned integer using successive multiplications by **10** and additions. It is also possible to convert a string of digits in any radix system to an integer, using successive multiplications by the radix value and additions. Rewrite the string-to-integer program asking the user to input a radix value between **2** and **10** and a string of digits in the specified radix system. For example, if the radix value

is 8 then the string can only have octal digit characters from '0' to '7'. Convert the string of digit characters into an unsigned integer and display the value of the unsigned integer.

- Ask the user to input a radix value between 2 and 10.
- Ask the user to input a string of digits in the specified radix system.
- Loop through each character of the string, converting each character (which represents a digit) to its integer value.
- Multiply the current result by the radix and add the current digit value
- Print the resulting unsigned integer.

**Task 5:** *The integer-to-string program presented in Section 6.5 converts an unsigned integer to string format using successive division by 10 and storing the remainder digit characters in a string. It is also possible to convert the unsigned integer to any radix using successive divisions by the radix value. Rewrite the integer-to-string program asking the user to input an unsigned integer and a radix value between 2 and 10. Do the radix conversion and then print the string. Make sure that the string has sufficient space characters, especially when converting to radix 2.*

- Ask the user to input an integer.
- Ask the user to input a radix value between 2 and 10.
- Use successive divisions by the radix to extract each digit.
- Store each remainder (the digit in the current radix) in a string.
- Reverse the string at the end since we process the least significant digit first.
- Ensure that when converting to binary (radix 2), the string is padded to an appropriate length.
- After the conversion, print the resulting string.

**Task 6:** *Fraction computation: Using successive integer multiplications and divisions, write a MIPS program that divides an integer  $x$  by another integer  $y$  that are read as input. The result of the division should be in the form:  $a.b$ , where  $a$  is the integer part and  $b$  is the fractional part. Compute the fraction  $b$  with 8 digits after the decimal point. Display the result in the form  $a.b$ .*

- Read two integers  $x$  (numerator) and  $y$  (denominator) from the user.
- Converts numerator and denominator to floating-point registers.
- To get the fractional part  $b$ , multiply the remainder of the division by 10 repeatedly to extract each decimal place.
- Print the result.

## Results and Analysis:

### Task 1a:

```
li $v0, 4           # syscall code for print string
la $a0, msg          # load msg
syscall             # print msg

li $t0, 98765        # load 98765 into $t0
li $t1, 54321        # load 54321 into $t1
multu $t0, $t1       # perform unsigned multiplication (98765 * 54321)

mflo $t2             # move LO register result to $t2
mfhi $t3             # move HI register result to $t3

# Display LO result
li $v0, 4           # syscall code for print string
la $a0, lo_result   # load lo_result address
syscall             # print "LO register: "
li $v0, 1           # syscall code for print integer
move $a0, $t2       # load LO result into $a0
syscall             # print LO value

# Display HI result
li $v0, 4           # syscall code for print string
la $a0, hi_result   # load hi_result address
syscall             # print "HI register: "
li $v0, 1           # syscall code for print integer
move $a0, $t3       # load HI result into $a0
syscall             # print HI value
```

By using the multu instruction, both numbers are treated as unsigned integers. As a result, the multiplication of  $98765 \times 54321$  is performed without considering any signs, yielding a straightforward result.

### Task 1b:

```
li $v0, 4          # syscall code for print string
la $a0, msg         # load msg address
syscall            # print msg

li $t0, -98765      # load -98765 into $t0
li $t1, -54321      # load -54321 into $t1
mult $t0, $t1       # perform signed multiplication (-98765 * -54321)

mflo $t2           # move LO register result to $t2
mfhi $t3           # move HI register result to $t3

# Display LO result
li $v0, 4          # syscall code for print string
la $a0, lo_result  # load lo_result address
syscall            # print "LO register: "

li $v0, 1          # syscall code for print integer
move $a0, $t2      # load LO result into $a0
syscall            # print LO value

# Display HI result
li $v0, 4          # syscall code for print string
la $a0, hi_result  # load hi_result address
syscall            # print "HI register: "

li $v0, 1          # syscall code for print integer
move $a0, $t3      # load HI result into $a0
syscall            # print HI value
```

By using the mult instruction, the numbers are treated as signed integers, meaning the multiplication considers whether the values are positive or negative. For example,  $-98765 \times -54321$  results in a positive value because the two negatives cancel each other out. However, if a positive number were multiplied by a negative one, the result would be negative, and the values in the HI and LO registers would differ.

**Task 2a:**

```
li $t0, 98765      # Load 98765 into $t0
li $t1, 54321      # Load 54321 into $t1

divu $t0, $t1      # Unsigned division of $t0 by $t1
mflo $a0           # Move quotient (LO) to $a0
mfhi $a1           # Move remainder (HI) to $a1

li $v0, 4          # Load syscall for print string
la $a0, quo        # Load address of message
syscall            # Print "Quotient (LO): "

li $v0, 1          # Load syscall for print integer
move $a0, $a0      # Move quotient to $a0 for printing
syscall            # Print quotient

li $v0, 4          # Load syscall for print string
la $a0, remainder  # Load address of message
syscall            # Print "\nRemainder (HI): "

li $v0, 1          # Load syscall for print integer
move $a0, $a1      # Move remainder to $a0 for printing
syscall            # Print remainder
```

### Task 2b:

```
li $t0, -98765      # Load -98765 into $t0
li $t1, -54321      # Load -54321 into $t1

div $t0, $t1        # Signed division of $t0 by $t1
mflo $a0            # Move quotient (LO) to $a0
mfhi $a1            # Move remainder (HI) to $a1

li $v0, 4           # Load syscall for print string
la $a0, quo         # Load address of message
syscall            # Print "Quotient (LO): "

li $v0, 1           # Load syscall for print integer
move $a0, $a0       # Move quotient to $a0 for printing
syscall            # Print quotient

li $v0, 4           # Load syscall for print string
la $a0, remainder   # Load address of message
syscall            # Print "\nRemainder (HI): "

li $v0, 1           # Load syscall for print integer
move $a0, $a1       # Move remainder to $a0 for printing
syscall            # Print remainder
```

In tasks 2a and 2b, the quotient remains the same in both cases (268500992) because dividing two positive or two negative numbers results in a positive quotient. However, the remainder is affected by the sign in signed division. In Task 2a, the remainder is positive (44444) since both values are positive in unsigned mode. In Task 2b, the signed division keeps the sign of the remainder consistent with the signed inputs, resulting in -44444.

### Task 3:

```
factorial_loop:

    beq $t2, 0, display_result    # if $t2 is 0, factorial is complete

    mul $t3, $t1, $t2             # $t3 = $t1 * $t2

    div $t4, $t3, $t2             # $t4 = $t3 / $t2
    mflo $t4                      # move quotient to $t4
    bne $t4, $t1, overflow        # if $t4 != $t1, overflow occurred

    move $t1, $t3                 # $t1 = $t3 (new factorial result)
    sub $t2, $t2, 1              # decrement $t2

    j factorial_loop
```

The code iteratively calculates the factorial of a given number and checks for overflow during each multiplication step. The loop begins by checking if the counter register \$t2 has reached zero, indicating that the factorial calculation is complete, in which case it jumps to display\_result. Otherwise, it multiplies the current factorial result stored in \$t1 by \$t2, with the product stored in \$t3. To detect overflow, the code divides \$t3 by \$t2 and stores the quotient in \$t4. If \$t4 does not equal the original value of \$t1, it means an overflow has occurred, and the program jumps to the overflow label to handle it. If no overflow is detected, the code updates \$t1 with the new factorial result stored in \$t3, decrements \$t2 by one, and loops back to repeat the process until \$t2 reaches zero.

#### Task 4:

```
li    $v0, 8           # syscall for read string
la    $a0, input_buffer # load address of input buffer
li    $a1, 32          # maximum number of characters
syscall

li    $t0, 0           # $t0 will hold the final integer result

la    $t2, input_buffer # Load the starting address of input buffer

convert_loop:
lb     $t3, ($t2)       # Load a character from the input buffer
beq    $t3, 0x0A, display_result # Newline (end of input), proceed to display result
beq    $t3, 0, display_result  # Null terminator, end of input

# Check if character is within the valid range for the specified radix
sub     $t3, $t3, '0'    # Convert ASCII to integer value
blt     $t3, 0, invalid_digit_msg # Check if less than 0
bge     $t3, $t1, invalid_digit_msg # Check if greater than or equal to radix

# Multiply the current result by the radix and add the new digit
mul     $t0, $t0, $t1    # result = result * radix
add     $t0, $t0, $t3    # result += current digit

# Move to the next character in the string
addi    $t2, $t2, 1      # Advance to the next character
j       convert_loop     # Repeat the loop
```

Each character from the input string is processed by converting it from ASCII to an integer. It checks if the character is valid for the specified radix. If valid, it updates the result by multiplying the current value by the radix and adding the digit. The loop continues until a newline or null terminator is encountered, at which point it moves to display the result.



### Task 5:

```
    la    $t2, input_buffer      # Address of the output buffer
    li    $t3, 0                 # Counter for number of digits

convert_loop:
    beqz   $t0, finish_conversion # If integer is 0, exit loop

    divu   $t4, $t0, $t1         # Divide $t0 by $t1
    mfhi   $t5                   # $t5 = remainder (digit)
    mflo   $t0                   # $t0 = quotient

    addi   $t5, $t5, '0'         # Convert to ASCII ('0' = 48)
    sb     $t5, ($t2)            # Store ASCII character in buffer
    addiu  $t2, $t2, 1           # Move buffer pointer
    addiu  $t3, $t3, 1           # Increment digit count
    j      convert_loop          # Repeat for the next digit

finish_conversion:

    la     $t2, input_buffer      # Reset $t2 to the start of the string
    add    $t4, $t2, $t3          # $t4 = end of string (t2 + digit count)
    addi   $t4, $t4, -1           # Point to the last character

reverse_loop:
    bge    $t2, $t4, display_result # If start >= end, done reversing
    lb     $t6, ($t2)             # Load character from start
    lb     $t7, ($t4)             # Load character from end
    sb     $t6, ($t4)             # Swap characters
    sb     $t7, ($t2)
    addiu  $t2, $t2, 1            # Move start pointer forward
    addiu  $t4, $t4, -1           # Move end pointer backward
    j      reverse_loop           # Repeat
```

The integer in \$t0 is repeatedly divided by the radix \$t1. The remainder is stored as an ASCII character in the buffer, and the quotient becomes the new value of \$t0. This continues until the integer is reduced to 0. After conversion, the buffer is reversed by swapping characters from the start and end pointers until they meet, preparing the result for display.

### Task 6:

```
# Perform division
mtcl $t0, $f0          # Move integer numerator to $f0
mtcl $t1, $f1          # Move integer denominator to $f1
cvt.s.w $f0, $f0       # Convert to float
cvt.s.w $f1, $f1       # Convert to float
div.s $f2, $f0, $f1    # Divide $f0 by $f1, result in $f2

# Prepare result for output
li $v0, 4              # syscall for print_string
la $a0, output_msg     # load address of output message
syscall

li $v0, 2              # syscall for print_float
mov.s $f12, $f2        # Move result to $f12 for printing
syscall

li $v0, 4              # syscall for print_string
la $a0, newline        # load address of newline
syscall

j finish              # Jump to finish
```

The integer values in registers \$t0 and \$t1 are first moved into floating-point registers \$f0 and \$f1, respectively. They are then converted to floating-point numbers using `cvt.s.w`. After the conversion, the two floating-point numbers are divided (`div.s`), and the result is stored in \$f2. The result is then printed by first outputting a string message (`output_msg`), followed by the floating-point result stored in \$f2. A newline is printed afterward, and the program finishes by jumping to the `finish` label.

### Screenshots (Output):

#### Task 1a:

```
98765 * 54321

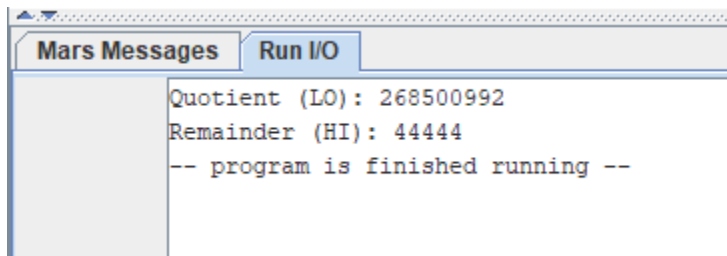
LO register: 1070046269
HI register: 1
-- program is finished running --
```

### Task 1b:

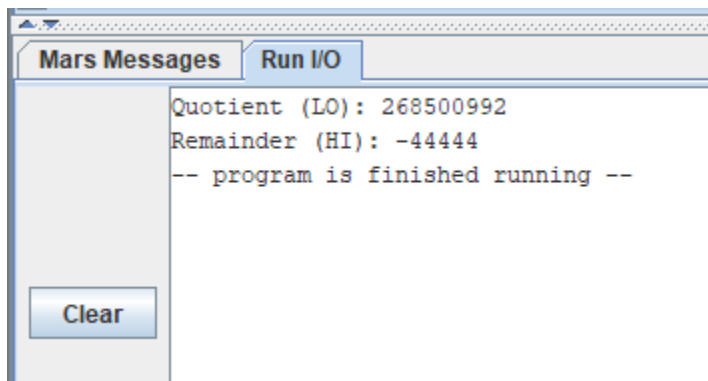
```
-98765 * -54321

LO register: 1070046269
HI register: 1
-- program is finished running --
```

### Task 2a:



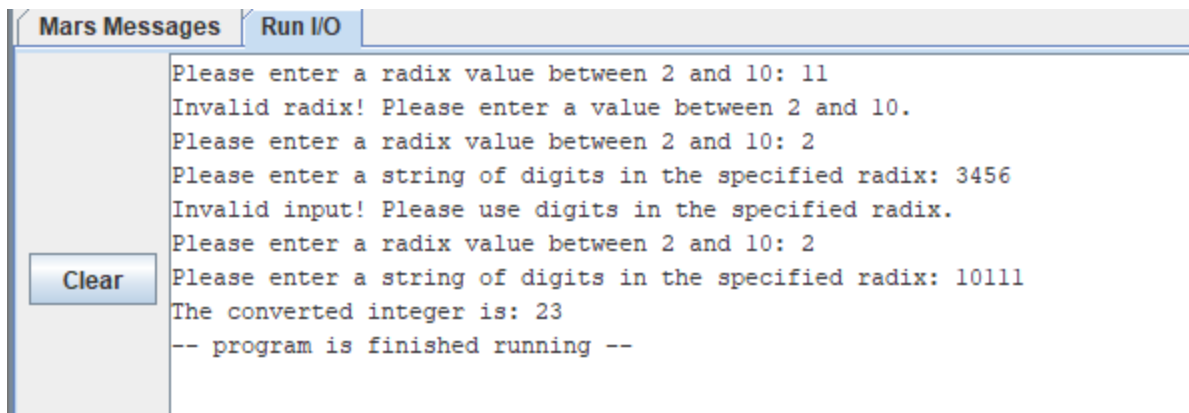
### Task 2b:



### Task 3:

```
Please enter a positive integer: 5
The factorial is: 120
Please enter a positive integer: 7
The factorial is: 5040
Please enter a positive integer: 9
The factorial is: 362880
Please enter a positive integer: 12
The factorial is: 479001600
Please enter a positive integer: 13
The factorial exceeds 32-bit limit.
-- program is finished running --
```

#### Task 4:

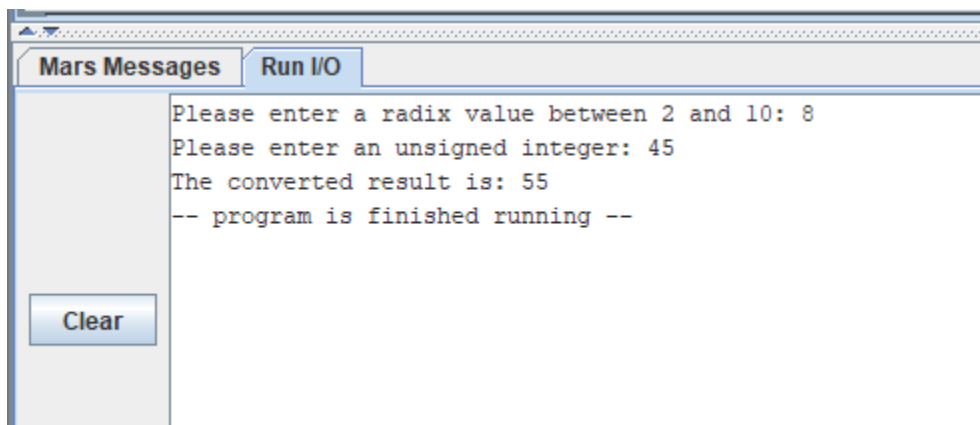


The screenshot shows a window titled 'Mars Messages' with a 'Run I/O' button. The text area contains the following messages:

```
Please enter a radix value between 2 and 10: 11
Invalid radix! Please enter a value between 2 and 10.
Please enter a radix value between 2 and 10: 2
Please enter a string of digits in the specified radix: 3456
Invalid input! Please use digits in the specified radix.
Please enter a radix value between 2 and 10: 2
Please enter a string of digits in the specified radix: 10111
The converted integer is: 23
-- program is finished running --
```

A 'Clear' button is located at the bottom left of the text area.

#### Task 5:

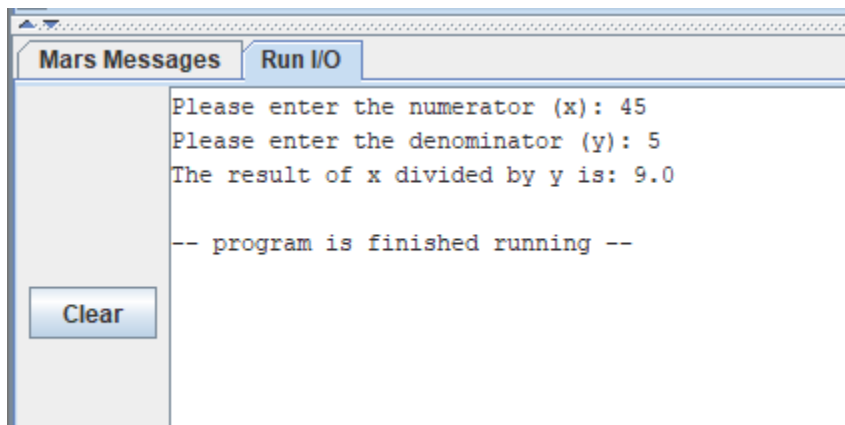


The screenshot shows a window titled 'Mars Messages' with a 'Run I/O' button. The text area contains the following messages:

```
Please enter a radix value between 2 and 10: 8
Please enter an unsigned integer: 45
The converted result is: 55
-- program is finished running --
```

A 'Clear' button is located at the bottom left of the text area.

#### Task 6:



The screenshot shows a window titled 'Mars Messages' with a 'Run I/O' button. The text area contains the following messages:

```
Please enter the numerator (x): 45
Please enter the denominator (y): 5
The result of x divided by y is: 9.0

-- program is finished running --
```

A 'Clear' button is located at the bottom left of the text area.

#### Conclusion:

- I have gained hands-on experience in key areas of MIPS assembly programming, including handling the **HI** and **LO** registers for multiplication and division, performing **factorial calculations**, and converting **digits into unsigned integers**. I have also learned **radix conversion** to change numbers between bases and worked with **fraction computation**, applying floating-point arithmetic for precise division. This experience

has strengthened my understanding of low-level programming, memory management, and mathematical operations in MIPS, providing a solid foundation for more advanced topics.

- In tasks 4 and 5, I encountered challenges in obtaining the unsigned integer equivalent of the radix and performing the radix conversion to string. These issues made me realize the importance of careful memory access. Specifically, an "address out of range" runtime exception occurred, highlighting the need to properly manage memory boundaries and ensure that data is accessed correctly. This experience underscored the importance of attention to detail when working with memory in low-level programming.
- In task 6, I encountered a challenge with printing the fractional part because I didn't divide the fractional part (scaled by 100000000) by 10 to extract each digit. Additionally, I didn't update the remainder and continue dividing to obtain each successive digit.

**Additional Notes/Observations:**

- In Task 4-6, buffer plays an important role. It is temporarily storing data during processing. It holds the converted digits of an integer as ASCII characters. It is used to store data during string manipulation (e.g., reversing the digits) and facilitates input/output operations, such as reading user input or displaying the result. It ensures smooth data handling by providing space for intermediate steps like conversion and string manipulation before final output.