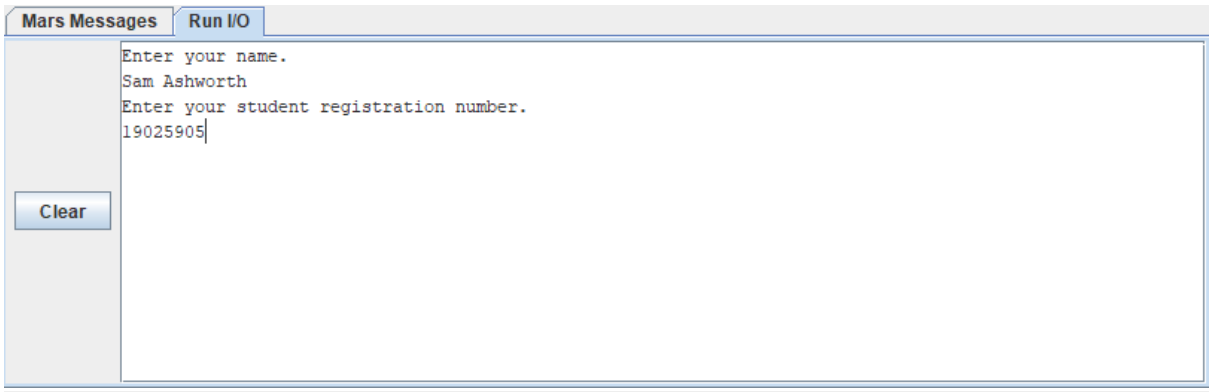
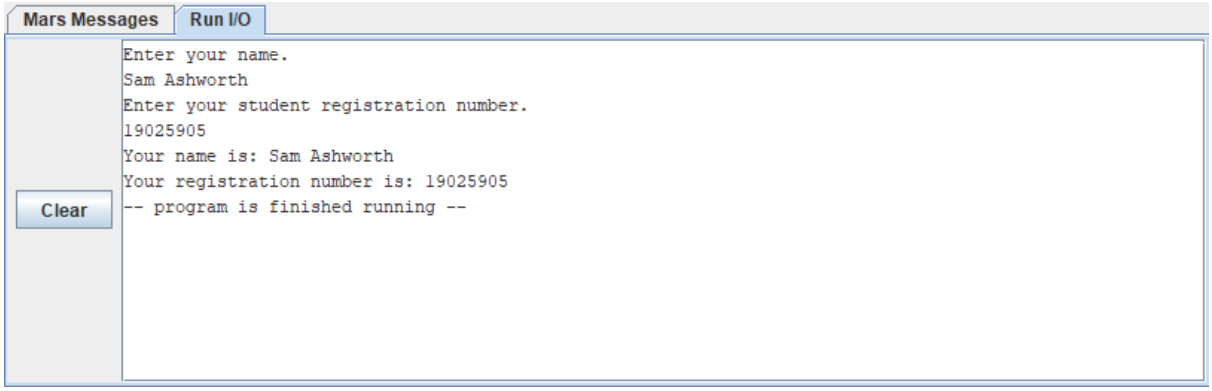
MIPS Assembly Language Assignment

Task A

This program allows the user to input their name and registration number and then outputs them. It does this by outputting prompts asking for the user to input the values and then awaiting inputs. The program will await a string for the name and an integer for the registration number. By loading the register $v0 with 8, it knows to expect a string and will store it in the variable address held in $a0. It also checks that the length of the string input is no longer than the space available in the address (30 characters) by having $a1 contain the number 30. For the integer input, $v0 must be set to 5. The integer is then stored in $v0 but must be moved to the register $t0 so that $v0 can be used for other system calls. The program then outputs both user values using other system calls where $a0 must hold the variables.

Screenshots





Code

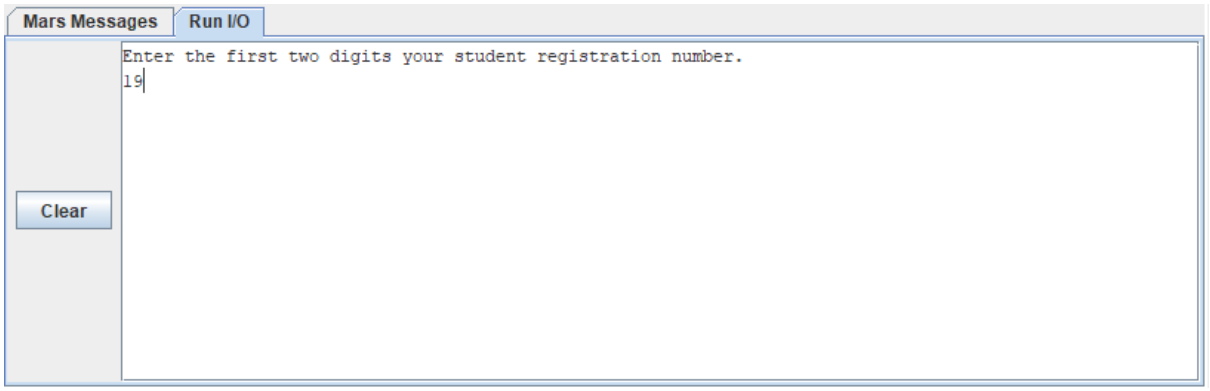
```
.data
#-----------------------------
#This is the data segment
#-----------------------------

strEnterName: .asciiz "Enter your name. \n"
strEnterReg: .asciiz "Enter your student registration number. \n"
strName: .space 30
strNameRes: .asciiz "Your name is: "
strRegRes: .asciiz "Your registration number is: "

.text
#-------------------------------
#This is the body of the code
#-------------------------------

main:

#
#Output strEnterName
#

li $v0, 4
la $a0, strEnterName
syscall

#
#Accept string input
#

li $v0, 8
li $a1 30
la $a0, strName
syscall

#
#Output strEnterReg
#

li $v0, 4
la $a0, strEnterReg
syscall

#
#Accept integer input
#
li $v0, 5
syscall

#
#Send user integer to $t0
```

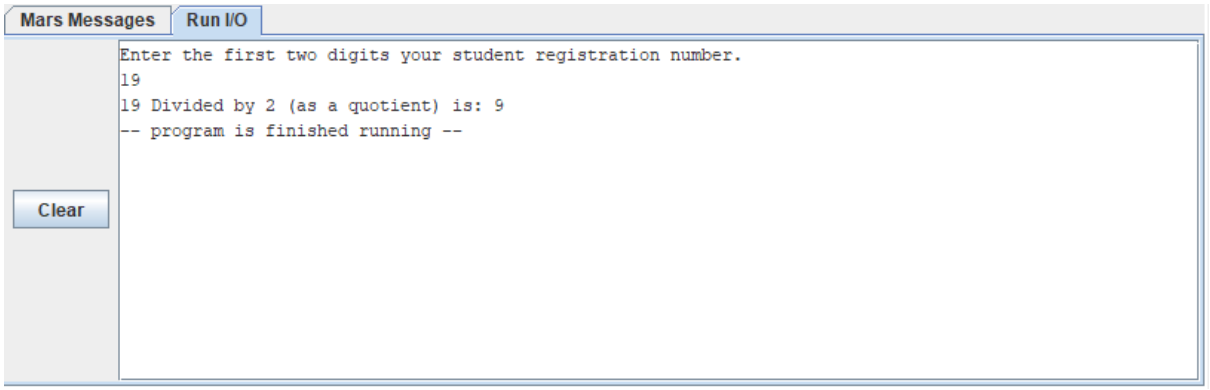
```
#

move $t0, $v0

#
#Output user's name
#

li $v0, 4
la $a0, strNameRes
syscall
la $a0, strName
syscall

#
#Output user's registration number
#

la $a0, strRegRes
syscall
li $v0, 1
move $a0, $t0
syscall

#
# System call code for exit = 10
#

li $v0, 10
syscall
```

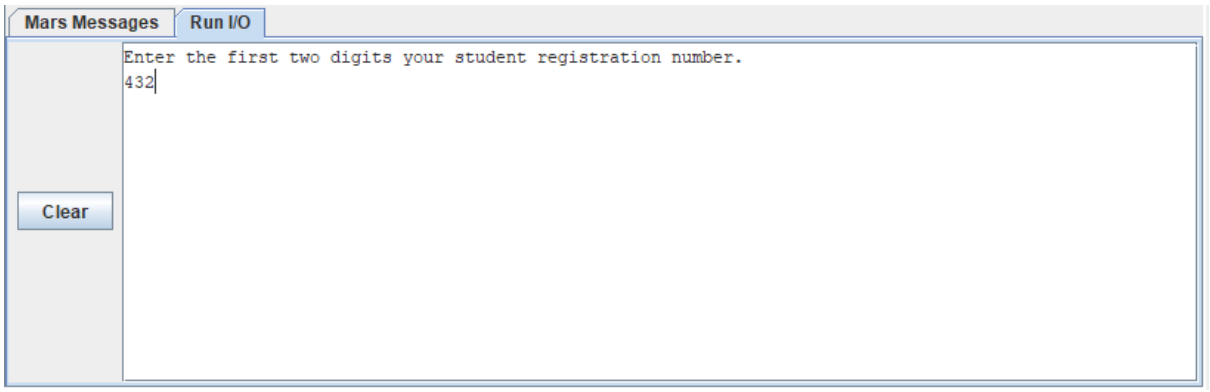
Task B

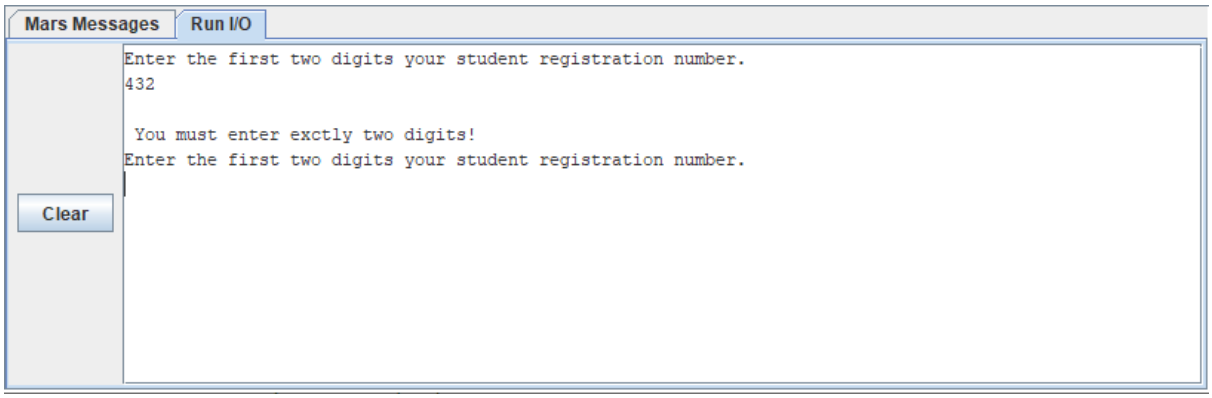
This program allows the user to enter the first two digits of their registration number (if more or fewer than two digits are entered, an error message is output and the program resets), divides it by 2 and then outputs the result. It does this by outputting a prompt asking the user to input these digits and then reads them in as an integer. The program then moves the input from $v0 to $t0 so that $v0 can later be overwritten to be used for other system calls. The “div” instruction is used to divide the user integer (stored in $t0) by 2 and store the quotient in $t1. The program then outputs the result in a sentence. The actual quotient is output by storing the contents of $t1 in $a0 and using a system call with $v0 set to 1.

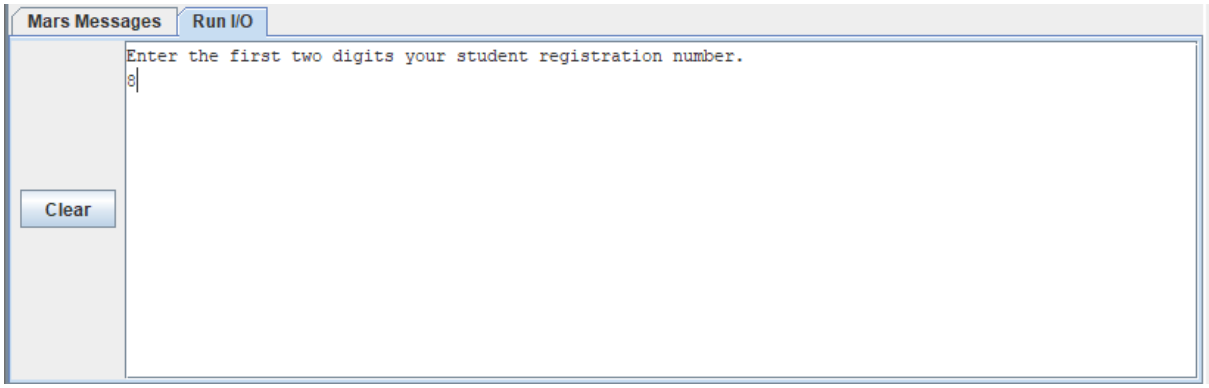
Screenshots

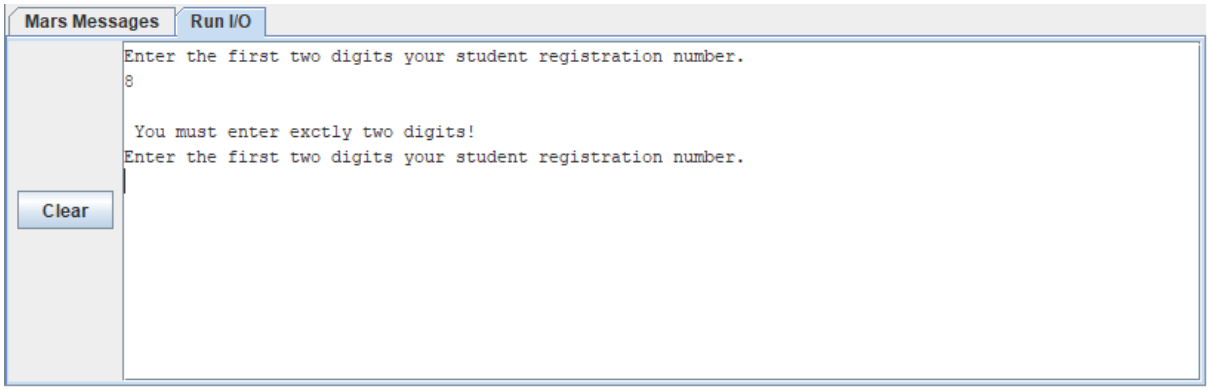












Code

```
.data
#-----------------------------
#This is the data segment
#-----------------------------
strNums: .asciiz "Enter the first two digits of your student registration number. \n"
strResult: .asciiz " divided by 2 (as a quotient) is: "
strError: .asciiz "\n You must enter exactly two digits! \n"

.text
#------------------------------
#This is the body of the code
#------------------------------

main:

#
#Output strNums
#

li $v0, 4
la $a0, strNums
syscall

#
#Accept integer input
#
```

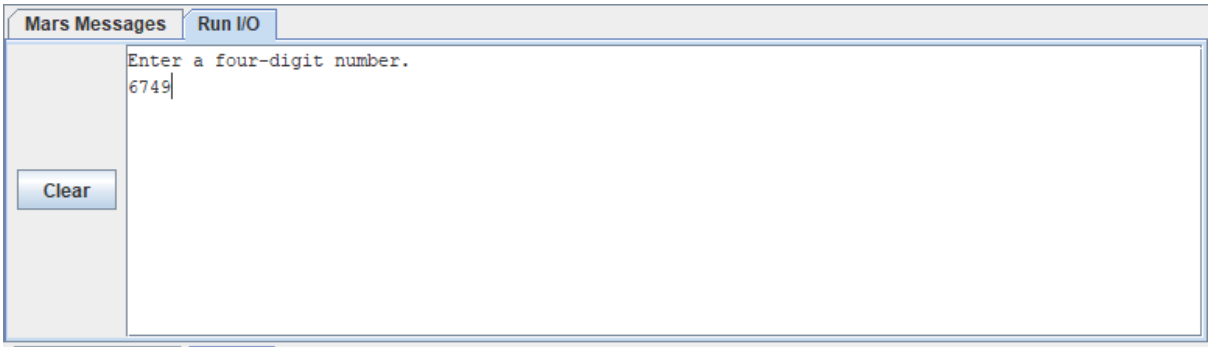
```
li $v0, 5
syscall

hiCheck:

#
#Check if the program has less than 3 digits then branch to next check
#

ble $v0, 99, lowCheck
b invalidNum
lowCheck:

#
#Check if the program has more than 1 digit then branch to continue program
#

bge $v0, 10, validNum

invalidNum:
#
#Output strError and restart program
#

li $v0, 4
la $a0, strError
syscall
b main

validNum:

#
#Send user integer to $t0
#

move $t0, $v0

#
#Divide user integer by 2 and store result in $t1
#

div $t1, $t0, 2

#
#Output user integer
#

li $v0, 1
move $a0, $t0
syscall

#
```

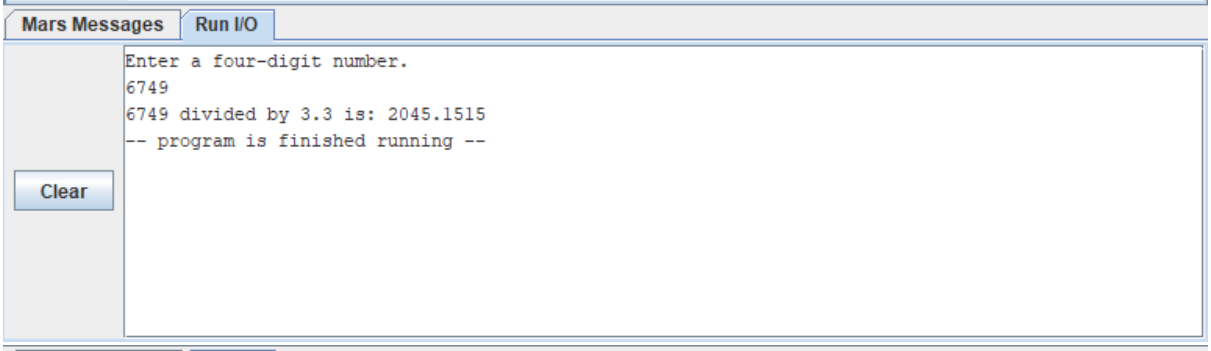
```
#Output strResult
#

li $v0, 4
la $a0, strResult
syscall

#
#Output result of division
#

li $v0, 1
move $a0, $t1
syscall

#
# System call code for exit = 10
#

li $v0, 10
syscall
```

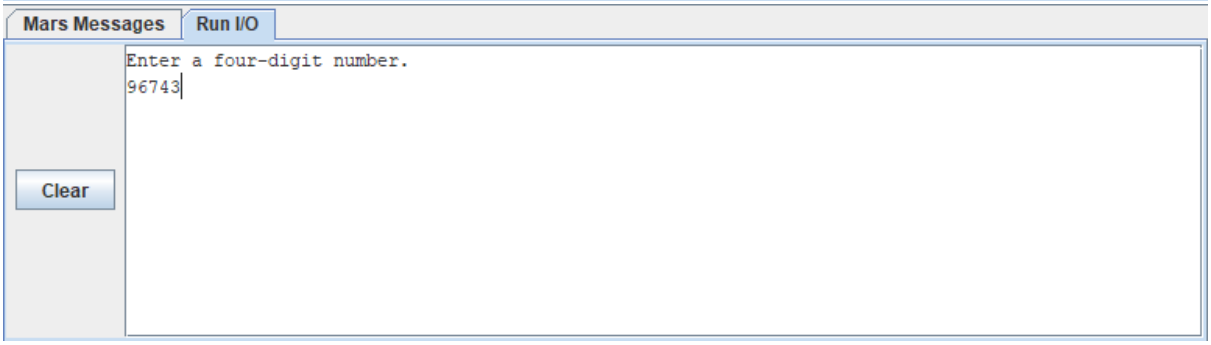
Task C

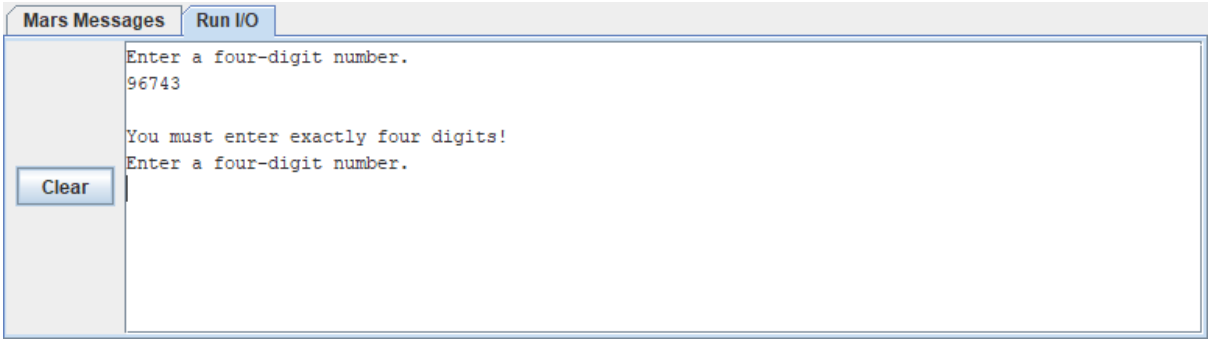
This program allows the user to enter a four-digit integer and divides it by 3.3. It does this by storing 3.3 in the data segment as a float called num1 and then loading this into Coprocessor 1 at $f4. It then asks the user to input a four-digit integer (and outputs an error and resets the program if the integer is not exactly four digits). This integer is then sent to $f0 in Coprocessor 1 and converted to a single precision float. The user float is then divided by 3.3 (stored in $f4) and the result is stored in $f12 in Coprocessor 1 (where it can be used as an argument for a syscall). The result is then output as part of a sentence. In order to perform any floating-point arithmetic, both operands must be of the same data type (in this case, both the user number and 3.3 are stored as singles). This involves using Coprocessor 1 to hold the operands and the result of the division. Outputting a float using a system call, uses the contents of $f12 as an argument, so this is where the result of the division must be stored before it can be displayed to the user.

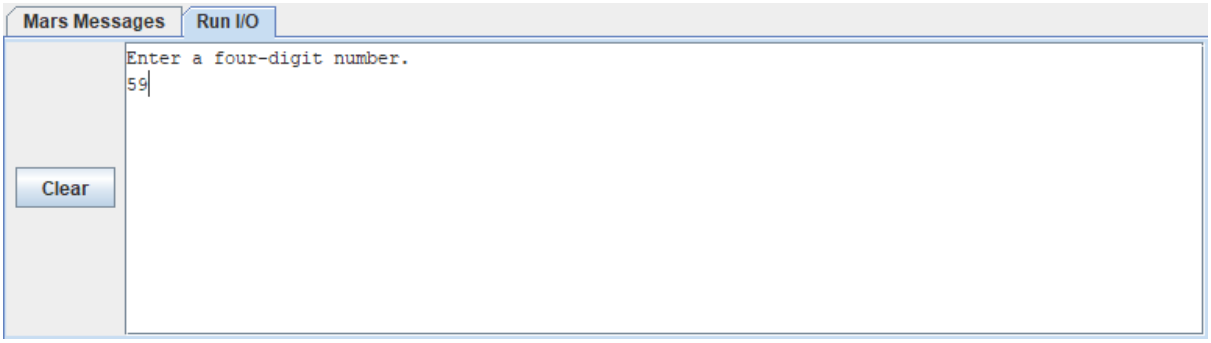
Screenshots

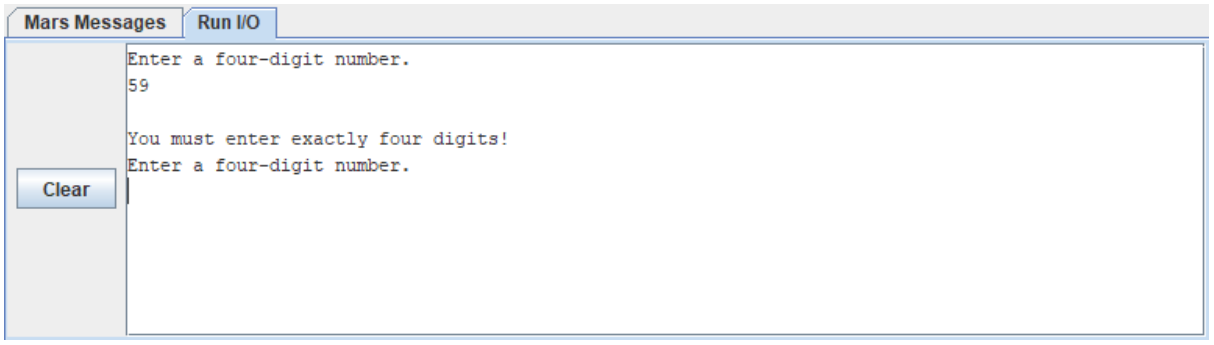




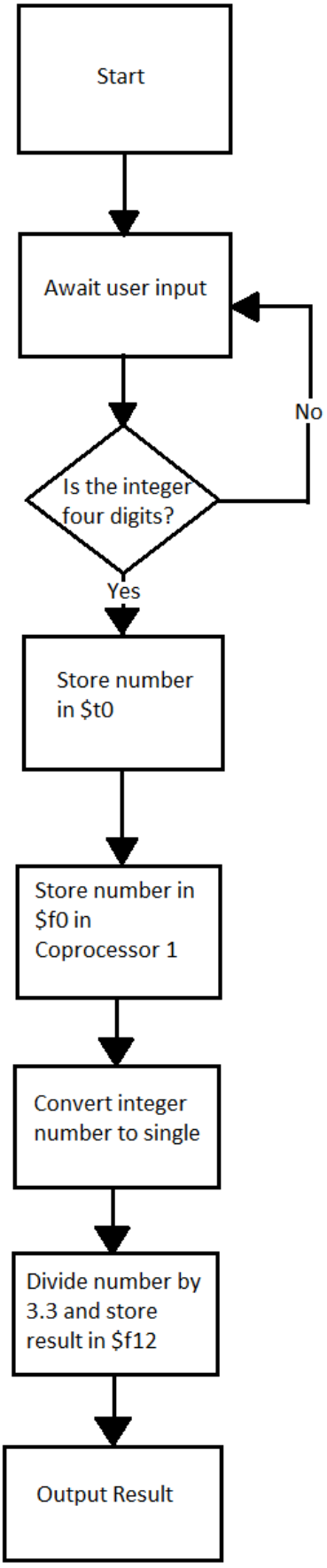







Mars Messages
Run I/O
Clear
Enter a four-digit number.
59
You must enter exactly four digits!
Enter a four-digit number.

Flow Diagram



Code

```
.data
#-----------------------------
#This is the data segment
#-----------------------------
strNums: .asciiz "Enter a four-digit number. \n"
strResult: .asciiz " divided by 3.3 is: "
strError: .asciiz "\nYou must enter exactly four digits! \n"
num1: .float 3.3

.text
#-------------------------------
#This is the body of the code
#-------------------------------

main:

#
#Load 3.3 into Coprocessor 1 at $f4 as a single
#

l.s $f4, num1

#
#Output strNums
#

li $v0, 4
la $a0, strNums
syscall

#
#Accept integer input
#

li $v0, 5
syscall

hiCheck:

#
#Check if the program has less than 3 digits then branch to next check
#

ble $v0, 9999, lowCheck
b invalidNum

lowCheck:

#
```

```
#Check if the program has more than 1 digit then branch to continue program
#

bge $v0, 1000, validNum

invalidNum:
#
#Output strError and restart program
#

li $v0, 4
la $a0, strError
syscall
b main

validNum:

#
#Send user integer to $t0
#

move $t0, $v0

#
#Send user integer to $f0 in Coprocessor1
#

mtc1 $v0, $f0

#
#Convert user integer to a single
#

cvt.s.w $f0, $f0

#
#Divide user single by 3.3 and store result in $f12
#

div.s $f12, $f0, $f4

#
#Output user integer
#

li $v0, 1
move $a0, $t0
syscall

#
#Output strResult
#
```

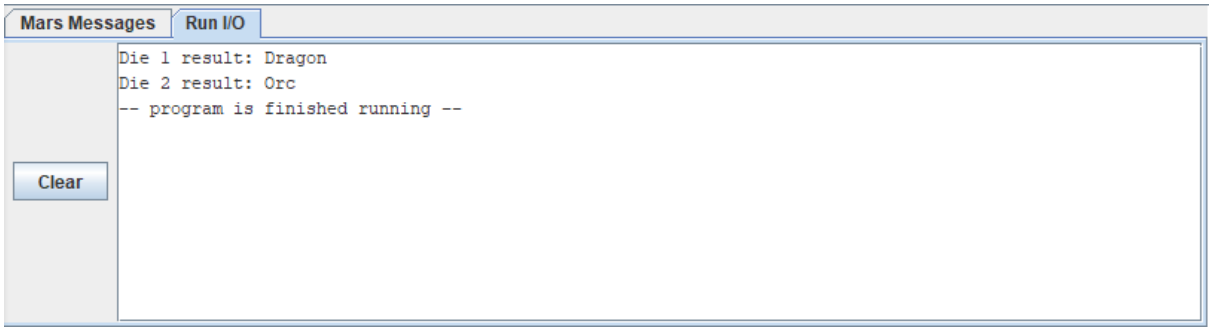
```
li $v0, 4
la $a0, strResult
syscall

#
#Output result of division as a single
#

li $v0, 2
syscall

#
# System call code for exit = 10
#

li $v0, 10
syscall
```

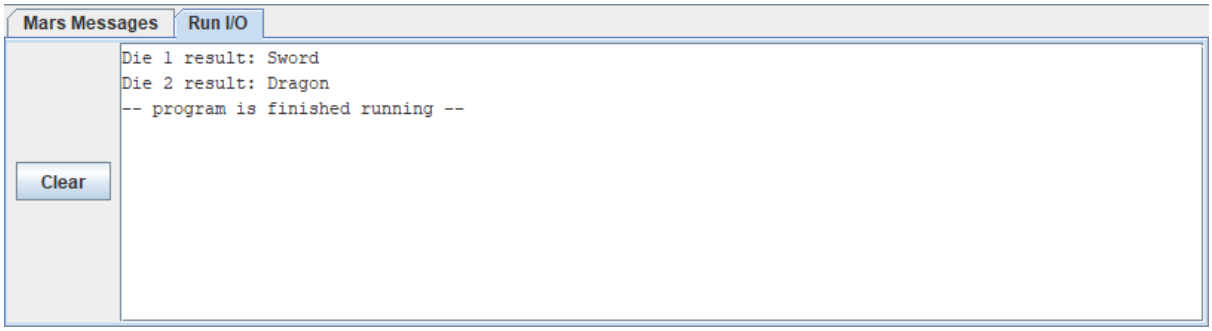
Task D

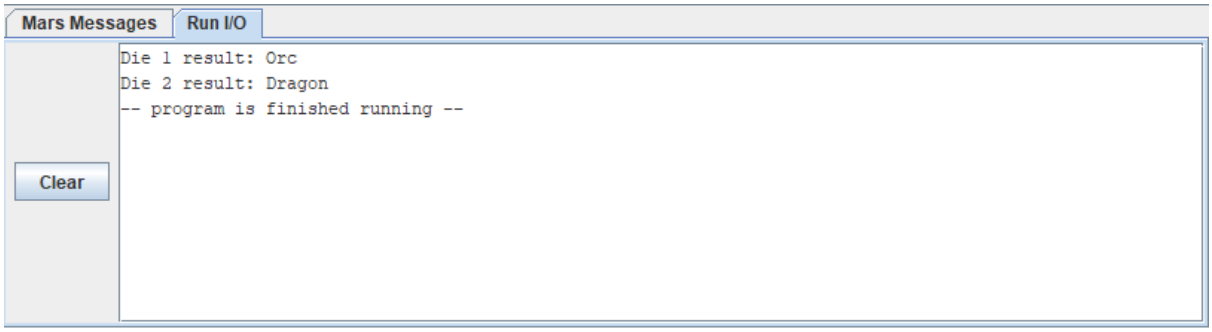
This program simulates the throwing of two twelve-sided dice. It achieves this result by using syscall 42 with an argument of 12 to generate a random number between 0 and 11. It then adds 1 to this number and stores it. The reason for adding 1 is so that it can loop the random number generation twice, saving the results in two different registers, by checking that they don't contain 0. The program then relates these numbers to the symbol they represent (1-5 being Dragon, 6-8 being Sword and 9-12 being Orc) by branching to different areas of the code and looping back to repeat the process. For example, the six main labels that can be branched to are dragFirstVal, dragSecondVal, swordFirstVal, swordSecondVal, orcFirstVal and orcSecondVal. The piece of code that runs depends on the randomly generated number and whether it's the first or second die being checked. Once the correct values have been stored, they are displayed to the user with a few system calls for string outputs.

Screenshots

Below are the results for three separate simulations of the dice being rolled.







Code

```
.data
#-----------------------------
#This is the data segment
#-----------------------------
strFirstDie: .asciiz "Die 1 result: "
strSecondDie: .asciiz "\nDie 2 result: "
strDragon: .asciiz "Dragon"
strOrc: .asciiz "Orc"
strSword: .asciiz "Sword"
strConnect: .asciiz " and "

.text
#-----------------------------
#This is the body of the code
```

```
#------------------------------

main:
#
#Initialise the registers by setting them to 0
#

li $t0, 0
li $t1, 0
li $t2, 0
li $t3, 0
li $t4, 0

generate:
#
#Generate a random number between 0 and 11
#

li $v0, 42
li $a1, 12
syscall

#
#Branch to firstVal if $t0 contains 0
#

beqz $t0, firstVal

secondVal:
#
#Store the random number in $t1 and add 1 then exit loop
#

add $t1, $a0, 1
b result

firstVal:
#
#Store the random number in $t0 and add 1 and branch to generate
#

add $t0, $a0, 1
b generate

result:
#
#Store the value from $t0 in $t2
#

move $t2, $t0

check:
```

```
#
#Check value of dice and relate to strings
#

ble $t2, 5, dragon
ble $t2, 8, sword
ble $t2, 12, orc
b display

dragon:
#
#Check if $t3 has been used, branch to dragFirstVal if not
#

beqz $t3, dragFirstVal

dragSecondVal:
#
#Store strDragon in $t4 and branch back to check. Use 13 in $t2 as a terminating condition
#

la $t4, strDragon
li $t2, 13
b check

dragFirstVal:
#
#Store strDragon in $t3
#

la $t3, strDragon
move $t2, $t1
b check

sword:
#
#Check if $t3 has been used, branch to swordFirstVal if not
#

beqz $t3, swordFirstVal

swordSecondVal:
#
#Store strSword in $t4 and branch back to check
#

la $t4, strSword
li $t2, 13
b check

swordFirstVal:
#
```

```
#Store strSword in $t3
#

la $t3, strSword
move $t2, $t1
b check

orc:
#
#Check if $t3 has been used, branch to orcFirstVal if not
#

beqz $t3, orcFirstVal

orcSecondVal:
#
#Store strOrc in $t4 and branch back to check
#

la $t4, strOrc
li $t2, 13
b check

orcFirstVal:
#
#Store strOrc in $t3
#

la $t3, strOrc
move $t2, $t1
b check

display:
#
#Output result of first die
#

la $a0, strFirstDie
li $v0, 4
syscall
move $a0, $t3
li $v0, 4
syscall

#
#Output result of second die
#

la $a0, strSecondDie
li $v0, 4
syscall
move $a0, $t4
```

```
li $v0, 4
syscall

end:
#
# System call code for exit = 10
#

li $v0, 10
syscall
```