Chris Carr

Manchester metropolitian university

6g4z1102 – 1cwk50 part 2

Table of Contents

[Task A 2](#_Toc535070103)

[Screenshots 2](#_Toc535070104)

[Code 3](#_Toc535070105)

[Task B 5](#_Toc535070106)

[Screenshots 5](#_Toc535070107)

[Code 7](#_Toc535070108)

[Task C 8](#_Toc535070109)

[Floating Point Arithmetic in MIPS 8](#_Toc535070110)

[Flow Chart Diagram 9](#_Toc535070111)

[Program Screenshots 10](#_Toc535070112)

[Code 11](#_Toc535070113)

[Task D 12](#_Toc535070114)

[Flowchart 13](#_Toc535070115)

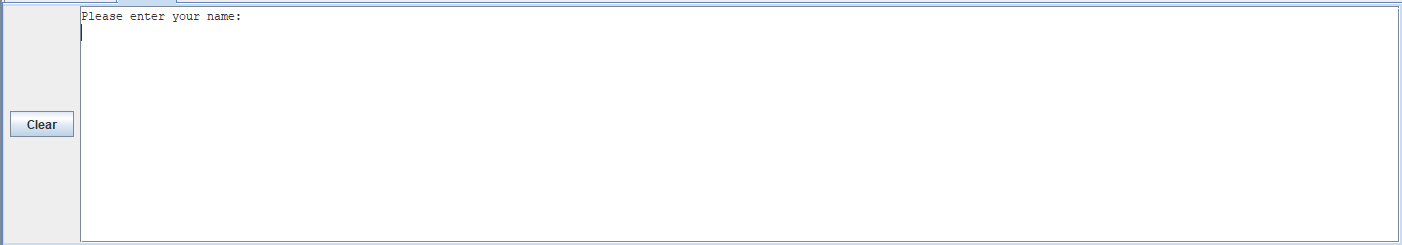
[Code 14](#_Toc535070116)

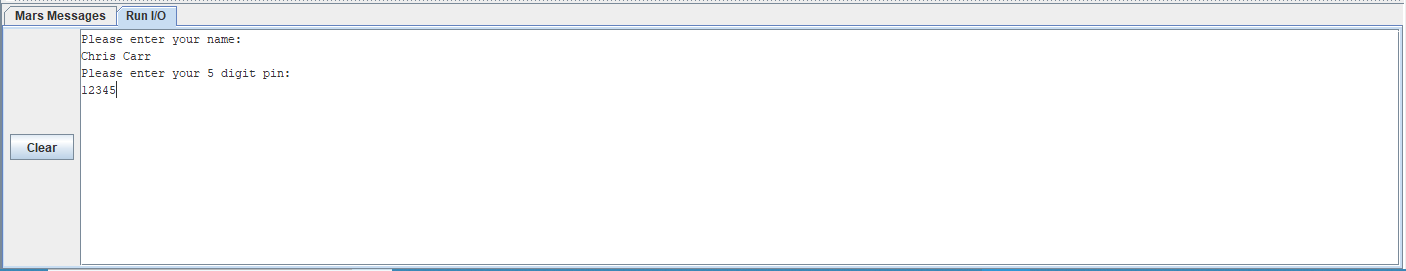
# Task A

Task A was a good way of reinforcing the information learned through university lectures/labs. It highlighted areas where my basic knowledge was lacking, and I had to resolve these issues with self-learning, rather than asking a tutor for help. It was good to begin the assignment with a basic task. One thing that I noticed throughout this task was that I needed to keep checking the reference sheets (<http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html>), rather than knowing which syscall commands to use off by heart (which the tutors do!). I identified 3 problems with the program:

* The entered string is not validated. For example: names shouldn’t contain numbers or specific special characters (i.e @)
* When the name String is read from the input buffer, a newline is added to the end (\n). This stopped me from displaying the output on a single line, as first intended.
* No validation that the integer is 5 digits long

## Screenshots







(*Revision 1*) la changed for move when moving variables. This should improve efficiency. Input variables are stored in temporary register rather than permanent registers. (stored in $t1 instead of $a1). Allocated space for String input prior to accepting the String.

<https://github.com/jynxmagic/MIPS-Assembly-Language/commit/b6ac553f664041c003bd6809f8cd7a0213a88531>

(*Revision 2*) Correctly terminated program. <https://github.com/jynxmagic/MIPS-Assembly-Language/commit/8dfeae3e05a8908f5d59b2d359985ac745c0f626>

## Code

**.data**

**prompt\_name:** **.asciiz** "Please enter your name: \n"

**prompt\_pin:** **.asciiz** "Please enter your 5 digit pin: \n"

**output:** **.asciiz** "Name and Pin: \n"

**separator:** **.asciiz** " - \n"

**.text**

#Display prompt name

**la** $a0, **prompt\_name**

**li** $v0, 4

**syscall**

#Store name input

**li** $v0, 8

**li** $a1, 35 #allocate space for input (35 bytes - 35 Chars)

**syscall**

**move** $t1, $a0

##Probably could do some verification here (no numbers in name input) ####

#Display prompt pin

**la** $a0, **prompt\_pin**

**li** $v0, 4

**syscall**

#Store prompt pin

**li** $v0, 5

**syscall**

**move** $t2, $v0

#Output output

**la** $a0, **output**

**li** $v0, 4

**syscall**

#Output stored name

**move** $a0, $t1

**li** $v0, 4

**syscall**

#Output Seperator ("-")

**la** $a0, **separator**

**li** $v0, 4

**syscall**

#Output stored pin

**move** $a0, $t2

**li** $v0, 1

**syscall**

#end

**li** $v0, 10

**syscall**

Github commits: <https://github.com/jynxmagic/MIPS-Assembly-Language/commits/master/Task_A/Task_A.asm>

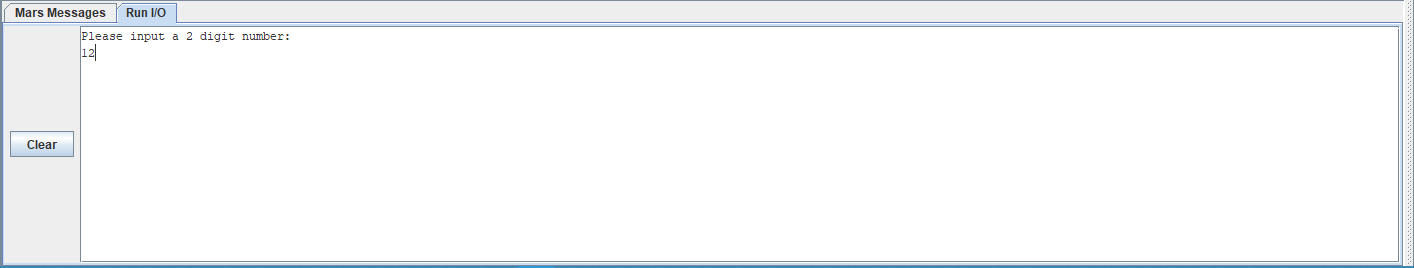
# Task B

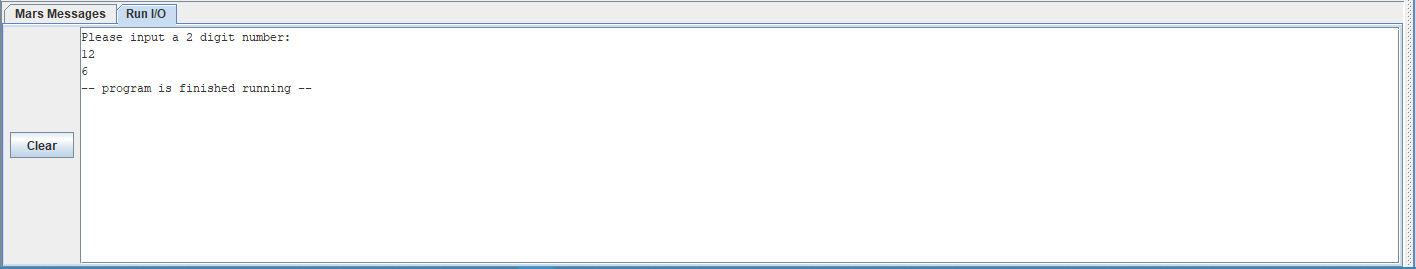
Task B introduced me to functions.

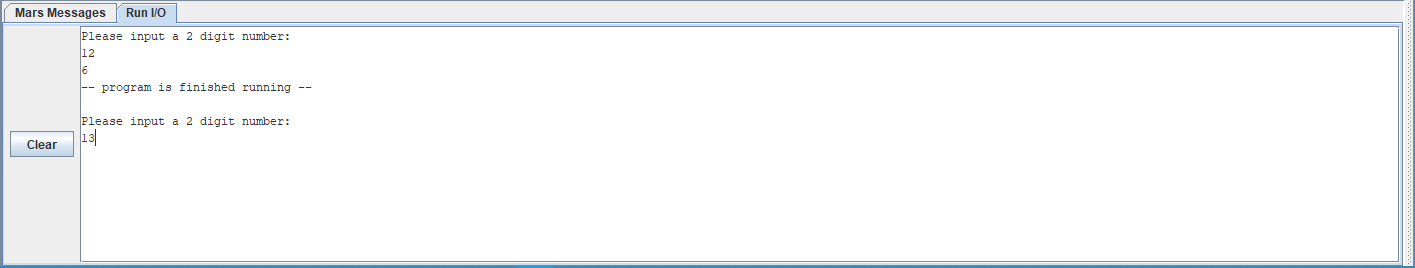
I noticed multiple functions which performed division, however I decided that the standard “div” function would suit my needs.

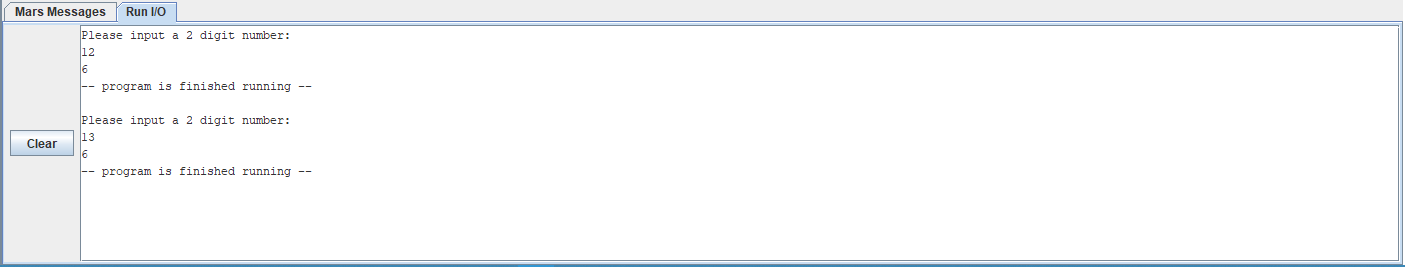
I may have used too many variables in this assignment. If I wanted to improve this program, my first step would be to reduce the number of variables used, and repeatedly use $a0 more often.

## Screenshots









(Revision 1) la replaced with move. Deleted code which rounds result – the coursework specifically states to output the quotient.

Github commit: <https://github.com/jynxmagic/MIPS-Assembly-Language/commit/f808677e33a0514665adcf2f0e7c558720e0310a#diff-f51c786fcb7d0d04bd18421eda028a57>

(*Revision 2*) Correctly terminated the program.

[*https://github.com/jynxmagic/MIPS-Assembly-Language/commit/8714e6f2582afa47c5eaf5547d7bb62cc74d8604*](https://github.com/jynxmagic/MIPS-Assembly-Language/commit/8714e6f2582afa47c5eaf5547d7bb62cc74d8604)

## Code

**.data**

**question:** **.asciiz** "Please input a 2 digit number: \n"

**.text**

#Display Question

**la** $a0, **question**

**li** $v0, 4

**syscall**

#Store user input

**li** $v0, 5

**syscall**

**move** $a1, $v0

#divide input by 2

**div** $a2, $a1, 2

#output integer result

**output:**

**move** $a0, $a2

**li** $v0, 1

**syscall**

#end

**li** $v0, 10

**syscall**

Github commits: <https://github.com/jynxmagic/MIPS-Assembly-Language/commits/master/Task_B/Task_B.asm>

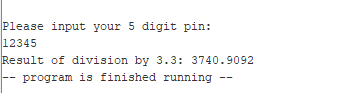
# Task C

## Floating Point Arithmetic in MIPS

In MIPS, floating point arithmetic is done in a separate component to the processor, called the co-processor. The arithmetic uses a standard for the arithmetic called IEEE 754 (latest revision IEEE 754-2008). This standard covers all the basis of arithmetic including: format, rounding rules, operations & exception handling.

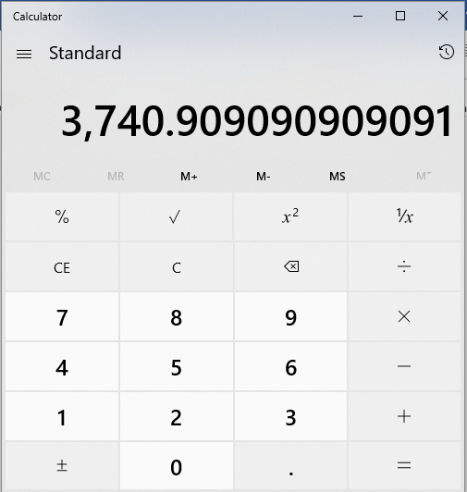
The co-processor includes a new set of registers, labelled from $f0-$f31. These registers come in pairs, as the second register is used to store the extra data for Doubles (covered later). Along with the new registers comes a new set of instructions to move values in and out of the co-processors’ registers, and instructions to perform arithmetic.

There are 2 data types accepted by the co-processor: Float and Double. Floats contain 32-bits of data, whereas doubles (conveniently named) contain double this – 64-bits of data. Doubles are extremely important for mathematical work where extreme precision is required.



***Figure A***

In my own work, I ran into a problem with using the 32-bit float data types. In Task C, I ran the division 12345/3.3, which is illustrated in *Figure A*. As you can see, the result of this is: 3740.9092. Next, to check that this was the correct result, I performed this addition on the windows calculator, which is illustrated in *Figure B*. The result of the division on the windows calculator was 3740.909090… . I noticed that this means there is a huge problem with the rounding of Single Precision Floats. The rounding of 90909 would never result in 9092; the rounding of 90909 should result in 9091 – in my program it becomes 9092. This is not an issue with the amount of data that is stored in Floats, rather it is an issue with the aforementioned IEEE 754 standards.



***Figure B***

(*Revision 1*) Cleaned up the comments. <https://github.com/jynxmagic/MIPS-Assembly-Language/commit/c5e8fd95381d9dcd8b74fa0cc75fab5fe701bb42>

## Flow Chart Diagram

Registers used: $a0, $v0, $a1

Instructions used: Load Address (la), Load Immediate (li), Syscall

Registers used: $f2, $f0, $f12

Instructions used: Load float 32 bit - single (l.s), Divide float 32 bit – single (div.s)

Output prompt to screen

User input pin

Divide by 3.3

Output Result

Registers used: $v0, $f12

Instructions used: Load Immediate (li), syscall

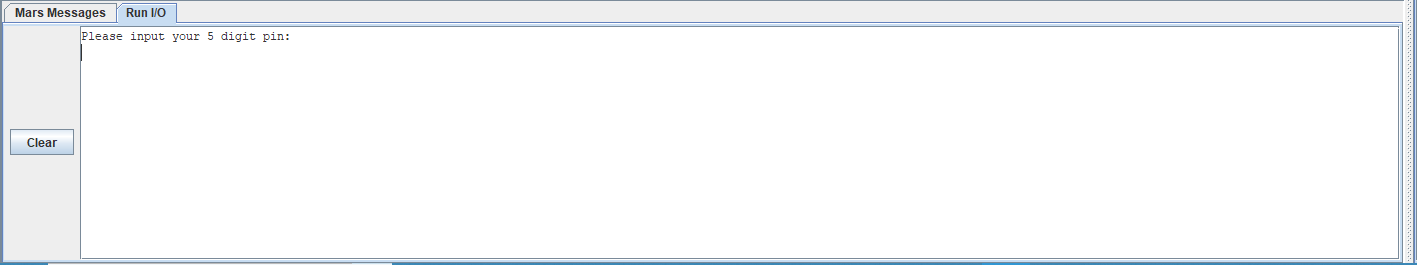
Registers used: $f12, $v0

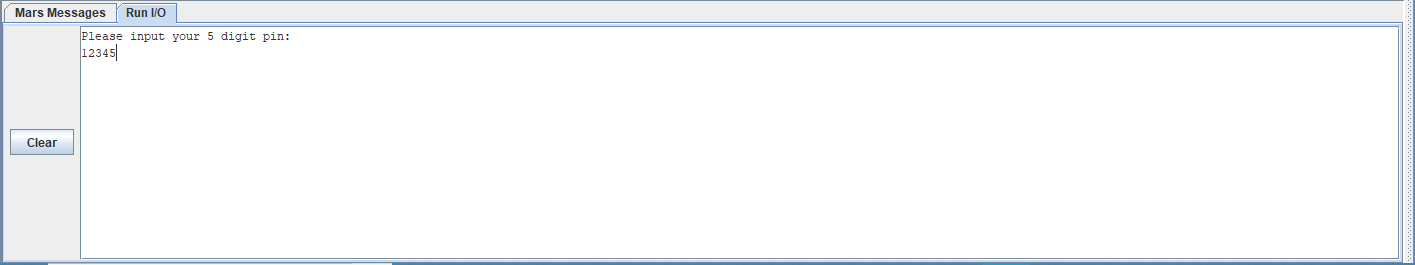
Instructions used: li, syscall

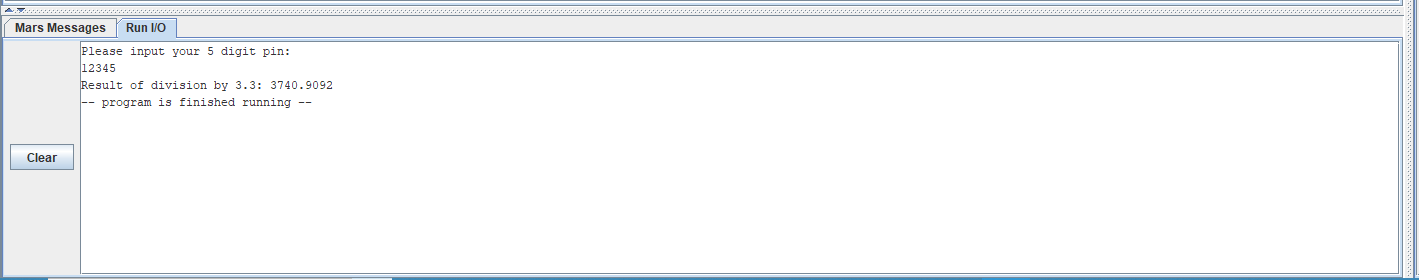
Registers used: $v0

Instructions used: $lli, syscall

## Program Screenshots







## Code

**.data**

**prompt:** **.asciiz** "Please input your 5 digit pin: \n"

**result:** **.asciiz** "Result of division by 3.3: "

**divisor:** **.float** 3.3

**.text**

## User input ##

#output text to screen

**la** $a0, **prompt**

**li** $v0, 4

**syscall**

#get user input

**li** $v0, 6

**syscall**

## Division ##

**l.s** **$f2**, **divisor**

#divide by 3.3 in co processor

**div.s** **$f12**, **$f0**, **$f2**

## Output ##

#output result string

**la** $a0, **result**

**li** $v0, 4

**syscall**

#output result

**li** $v0, 2

**syscall**

#end

**li** $v0, 10

**syscall**

Github commits: <https://github.com/jynxmagic/MIPS-Assembly-Language/commits/master/Task_C/Task_C.asm>

# Task D

In Task D, basic loop and function (called “branching” in MiPS) procedures were implemented. There are multiple ways to implement loops within MiPS. The methodology which was implemented into this particular program was using an instruction call “bne”, which stands for “Branch if Not Equal”. The program uses an iterative counter and the “bne” statement to determine if the dice has been rolled 8 times; after the dice has been rolled 8 times, the sum is outputted. Another methodology of fulfilling the objectives of this program would be to use the “jump” instruction in conjunction with the “$ra” register. Both achieve the same result, however the “branching” functionality is a lot easier to implement.

## Flowchart

Generate random number

(1-6)

Output random number

Increment counter

Add random number to sum of numbers

Has counter reached 8?

Output sum of numbers

**NO**

**Yes**

Registers used: $v0, $a0, $a1

Instructions used: li, la, addi, syscall

Registers used: $t0

Instructions used: addi

Registers used: $v0

Instructions used: li, syscall

Registers used: $t1, $a0

Instructions used: add

Registers used: $t0

Instructions used: bne

Registers used: $a0, $t1, $v0

Instructions used: move, li, syscall

Registers used: $v0

Instructions used: li, syscall

## Code

**.data**

**seed:** **.asciiz** "Machine code is awesome, so is Chris!"

**sum\_text:** **.asciiz** "Sum of all numbers = "

**newline:** **.asciiz** "\n"

**.text**

**li** $t0, 0 #counter

**li** $t1, 0 #value of all added sums

#random generator setup

## Set Seed ##

**li** $v0, 40

**li** $a0, 1 # id of random # generator

**la** $a1, **seed** # set seed for generator

**syscall**

**random\_int\_loop:**

**addi** $t0, $t0, 1 #increment counter

## Generate random number ##

**li** $v0, 42

**li** $a0, 1 #set id of random # generator

**li** $a1, 5 #set upper bound to 5

**syscall**

## Add 1 to generated number - dice never roll 0 ##

**addi** $a0, $a0, 1

## Print the number we just generated ##

**li** $v0, 1

**syscall**

## Add the number we just generated to our TOTAL value ##

**add** $t1, $t1, $a0

## Print newline ##

**la** $a0, **newline**

**li** $v0, 4

**syscall**

## Check if rolled 8 times, if not, roll again ##

**bne** $t0, 8, **random\_int\_loop**

## Print Sum of All numbers ##

#print sum text

**la** $a0, **sum\_text**

**li** $v0, 4

**syscall**

#print sum

**move** $a0, $t1

**li** $v0, 1

**syscall**

## End ##

**li** $v0, 10

**syscall**

Github commits: <https://github.com/jynxmagic/MIPS-Assembly-Language/commits/master/Task_D/Task_D.asm>