Project Report 1

Yousef Alaa Awad

yousef.awad@ucf.edu

EEL3801: Computer Organization

Due Date: 23rd June, 2025
Submission Date: 23rd June, 2025

1.0 Project Description
The point of this project is to develop, and teach how to make, a MIPS assembly program that performs multiplication, division, and modulus operations all without using the already built instructions such as *mul*, *div*, and *rem*. The project, specifically, is divided into two parts. Part A being to implement the following two functions: F = 5*B*D + A and G = D*D – C*A and then displays them to the user in binary and decimal formats. Part B then expands upon these formula and then was to compute the following functions: H = (F+2) / (G-2) and I = (F+25) % (H's Remainder), of which again then displays them in both decimal and binary formats. The entire goal of this project, at its core, was therefore to simply understand how loops and conditionals work in MIPS assembly.

2.0 Program Design

Part A
Again, for Part A, we were to design the following two functions: F = 5*B*D+A and G = D*D-C*A, but only using loops and conditionals so that we do not use any prebuilt *mul*, *div*, or *rem* assembly instructions. Now, since multiplication is just the collective additions of one term by the count of the other term, we can easily create a loop.

Specifically, say, for computing B*D in F, we use a loop (specifically mulBD in the assembly) and use a temporary counter register (for which is $t6) to increment additions of B until it reaches D times. Then, after this, we then continue this by resetting said temporary counter register to 0 and then do a repeated addition of B*D's value onto itself until the counter reaches the value of 5, thereby giving us the result of 5*B*D (of which is all done in the loop mul5, with the label mul5prep resetting the counter). Now to add the final A value onto the 5*B*D we simply just add it using the simple *add* inbuilt command/opcode of MIPS, and therefore the entirety of F is complete.

Now for the function of G = D*D-C*A we used a differing approach. That being, that we used an additional loop to subtract C, A times. And, thankfully, the repeated subtractions works exactly the same as for the calculations of F with additions above with B*D and 5*B*D and, as such, will not be discussed here for the sake of brevity.

Now to printing out the actual results to the use. That is simply done with loading the command register, that being $v0, with the either 1 or 35 (for either decimal or binary printing to the user) then loading the the text register, $a0, with the decimal number location (that being $t4 for F and $t5 for G).

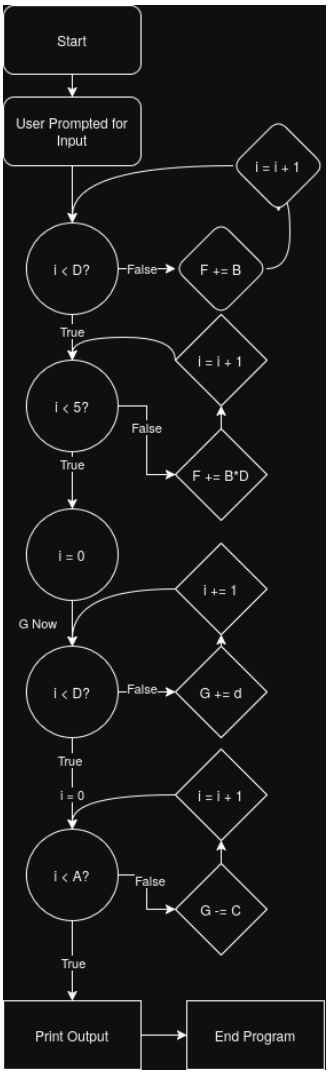
Part B
And now for Part B, of which will contain a lot of similarities to Part A and therefore will contain less text overall due to, once again, wanting to have a brief and concise report of the project. For this part we specifically want to perform division and modulus operations. In this, I have taken the exact inputs of Part A, thereby copying it over as it was implemented above assuming a user input of A, B, C, and D, and then extended it – as well as gotten rid of the print statements of F and G as they are now unneeded – to calculate the above define H and I.

To compute H, of which has the formula of H = (F+2) / (G-2), I took the values of F and G of earlier that were computed in Part A and added 2 to F and subtracted 2 from G. After this, I then did a loop, as using the inbuilt *div* instruction was not allowed, and simply continuously subtracted the value of G-2 from F+2, tracking each time that the "division" occurs via the temporary counter register (again $t6) until it failed the conditional check of F+2 being greater than G-2 (as division is therefore no longer possible). This approach also means that the remainder is also subsequently calculated, that being the ending value of the F+2 register.

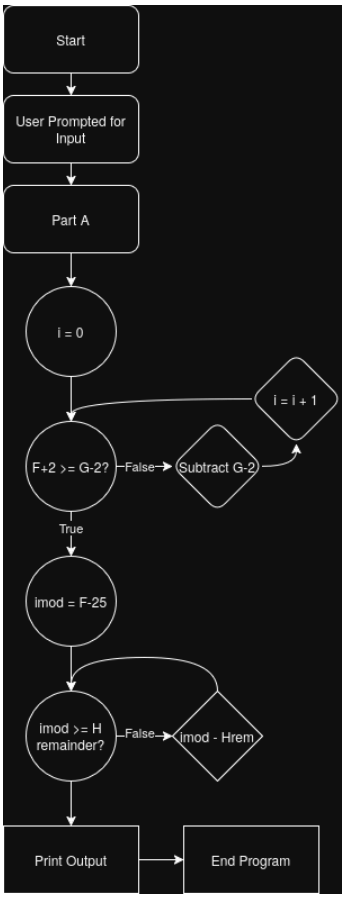
Now, with the H remainder value being known, we can calculate I = (F+25) % (H's Remainder). Now, to do modulus (of which only gives you the remainder of a division and not the amount of subtractions you can do to

it), I simply did an infinite loops of subtractions from F+25 by H's Remainder while F+25 was greater than H's Remainder. The result of F+25, after it failed the condition and became smaller than H's Remainder, is then the value of I.

The program then, just as in Part A, prints out the values of H, H's Remainder, and I, all in only decimal as wanted.



Flowchart for Part A



Flowchart for Part B

3.0 Symbol Table

Register	Usage	Part's Used In
$t0	Holds number A	A & B
$t0	Holds H's Remainder	B
$t1	Holds number B	A & B
$t1	Holds I's value	B
$t2	Holds number C	A & B
$t2	Holds H's value	B
$t3	Holds number D	A & B
$t4	Holds end result of F	A & B
$t5	Holds end result of G	A & B
$t6	Temporary Loop Counter	A & B
$t7	Holds value 4 for multiplying by 5	A & B
$t8	Holds original value of B*D	A & B
$v0	Declares which syscall is used	A & B
$a0	Declares secondary parameter used	A & B

Label	Usage	Part's Used In
Main	Declares start of MIPS program	A & B
MulBD	For calculating B*D	A & B
Mul5prep	For exiting B*D	A & B
Mul5	Loop for 5*B*D	A & B
ExitmulF	Exiting when F is calculated	A & B
MulDD	For calculating D*D	A & B
Mulsubprep	For exiting mulDD	A & B
MulsubCA	Loop to subtract C from D*D, A	A & B

Label	Usage	Part's Used In
	times	
PrintRest	Used to start print results sequence	A & B
DoneA	Used to start Part B	B
DivFG	For calculating F+2 / G+2	B
DonDiv	For when F+2 < G+2	B
ModI	For calculating (F-25)%H Remainder	B

4.0 Learning Coverage

This project therefore covers the following:

- Using conditional statements and branch statements in Assembly/MIPS
- Practice using the various simplistic functions of MIPS (such as *add*, or *sub*)
- Creation of loops based on variable iteration and checking
- Printing values, strings, and variables in both binary and decimal
- Storing variables, and manipulating registers all the while using them in functions
-

5.0 Prototype in C-Language

Part A

```
#include <stdio.h>

int main() {
  int A = 0, B = 0, C = 0, D = 0; // inputs
  int F = 0, G = 0;               // outputs
  int i = 0;                      // counter

  // getting user input
  printf("Enter 4 integers for A, B, C, D respectively:\n");
  scanf("%d", &A);
  scanf("%d", &B);
  scanf("%d", &C);
  scanf("%d", &D);
  printf("a = %d\nb = %d\nc = %d\nd = %d\n", A, B, C, D);

  // computing B*D
  do {
    F += B;
    i += 1;
  } while (i < D);

  i = 0;
  int Ftemp = F; // storing F temporarily like in MIPS

  // 5*(B*D)
  do {
    F += Ftemp;
    i += 1;
  } while (i < 4);

  F += A;
```

```
  i = 0;

  // Now for G
  // Computing D*D
  do {
    G += D;
    i += 1;
  } while (i < D);

  i = 0;

  // Computing D*D - C*A
  do {
    G -= C;
    i += 1;
  } while (i < A);

  // printing result
  printf("f_ten = %d\nf_two = %b\ng_ten = %d\ng_two = %b\n", F, F, G, G);
}
```

Part B

```
#include <stdio.h>

int main() {
  int A = 0, B = 0, C = 0, D = 0; // inputs
  int F = 0, G = 0;               // outputs
  int i = 0;                      // counter

  // getting user input
  printf("Enter 4 integers for A, B, C, D respectively:\n");
  scanf("%d", &A);
  scanf("%d", &B);
  scanf("%d", &C);
  scanf("%d", &D);
  printf("a = %d\nb = %d\nc = %d\nd = %d\n", A, B, C, D);

  // computing B*D
  do {
    F += B;
    i += 1;
  } while (i < D);

  i = 0;
  int Ftemp = F; // storing F temporarily like in MIPS

  // 5*(B*D)
  do {
    F += Ftemp;
    i += 1;
  } while (i < 4);

  F += A;
```

```
  i = 0;

  // Now for G
  // Computing D*D
  do {
    G += D;
    i += 1;
  } while (i < D);

  i = 0;

  // Computing D*D - C*A
  do {
    G -= C;
    i += 1;
  } while (i < A);

  // Now for Part B
  i = 0;        // will signify the quotient of H
  Ftemp = F + 2; // now storing remainder of H

  // Calculating H
  do {
    Ftemp -= G - 2;
    i += 1;
  } while (Ftemp >= G - 2);

  int imod = F - 25;

  do {
    imod -= Ftemp;
  } while (imod >= Ftemp);

  // printing result
  printf(
      "\nf_ten = %d\ng_ten = %d\nh_quotient = %d\nh_remainder = %d\ni_mod = %d\n",
      F, G, i, Ftemp, imod);
}
```

6.0 Test Plan

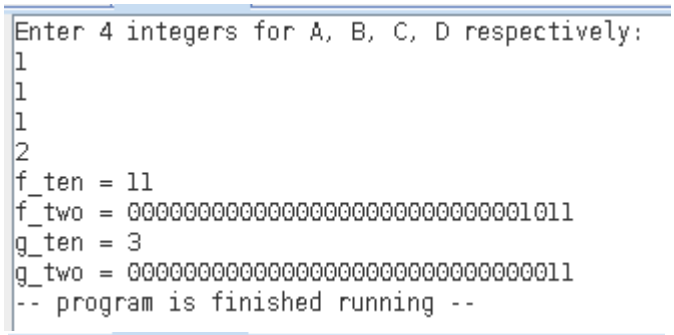
To test the functionality of the MIPS code, I used the following three cases:

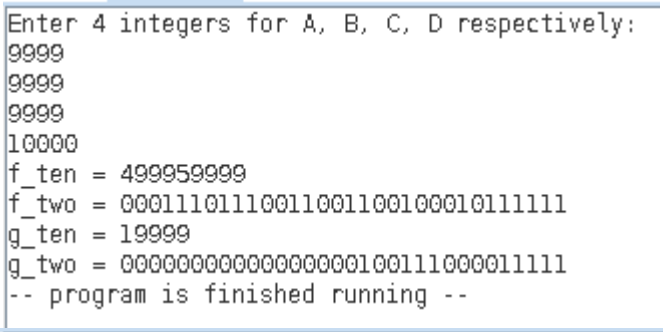
Test Case Name	Input	Output (Part A)	Output (Part B)
Minimum	1 1 1 2	f_ten = 11 f_two = 00000000000000000000000000001011 g_ten = 3 g_two = 00000000000000000000000000000011	f_ten = 11 g_ten = 3 h_quotient = 13 h_remainder = 0 i_mod = -14
Maximum	9999 9999 9999 10000	f_ten = 499959999 f_two = 00011101110011001100100010111111 g_ten = 19999 g_two =	f_ten = 499959999 g_ten = 19999 h_quotient = 25001 h_remainder = 15004 i_mod = 11690

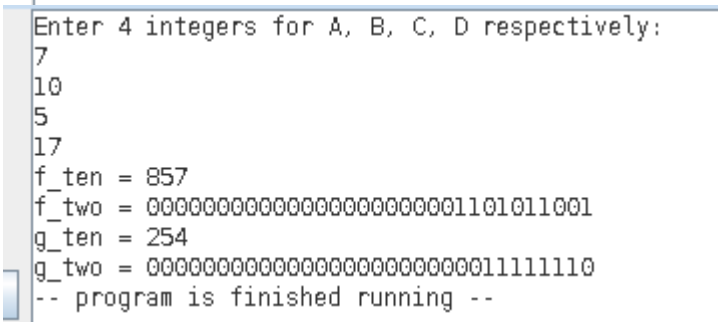
Test Case Name	Input	Output (Part A)	Output (Part B)
		00000000000000000100111000011111	
Default	7	f_ten = 857	f_ten = 857
	10	f_two =	g_ten = 254
	5	00000000000000000000001101011001	h_quotient = 3
	17	g_ten = 254	h_remainder = 103
		g_two =	i_mod = 8
		00000000000000000000000011111110	

The test cases specifically show the broadest range, both in the miminum and maximum and of a real test case of the middle that could be hand computed to double check, of potential inputs, all the while still not taking long to compute (or causing integer overflow). The most important test case of this was determined to be the default case, simply due to the fact that the edge cases are rarely ever going to be computed on daily usage.

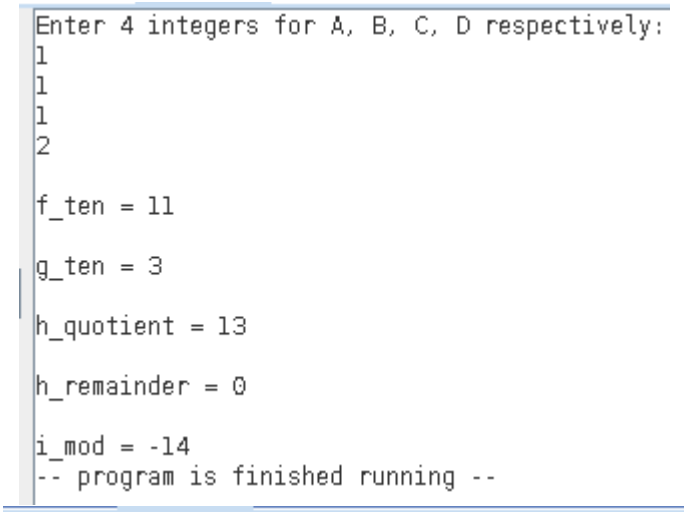
7.0 Test Results
Part A (in order of listed in Section 6.0):

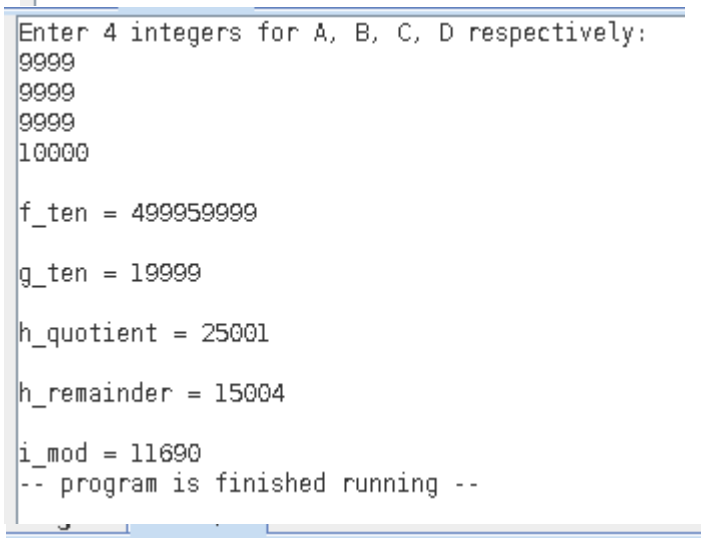


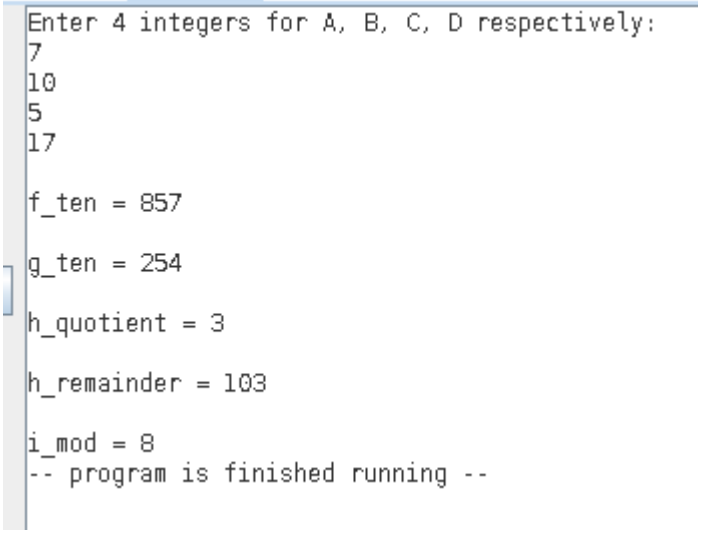




Part B (in order of listen in Section 6.0):







8.0 References

8.1 MARS Simulator

The MARS Simulator for MIPS processors, available at:
http://courses.missouristate.edu/kenvollmar/mars/

and MARS syscall functions listed at:
http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html