Logical Calculator Using MIPS Assembly Language

Joshua Harmon
San Jose State University
joshua.harmon@sjsu.edu

Abstract

This project report gives a detailed explanation of the MIPS emulator (MARS) environment, its features, and how it is used in this calculator. It also explains how the calculator works in full, including each function as well as the files that are involved in its implementation.

I. INTRODUCTION

In this project, the aim is to have students learn about the low-level fundamental logic that goes behind basic arithmetic by implementing these functions manually rather than using the built-in MIPS arithmetic operations. The implementation of these functions requires bit-by-bit operation of each register and value, which necessitates precise logical operations to perform correctly. To do this, one must utilize the MIPS simulator known as MARS and learn its various features and debugging features.

II. DOWNLOAD AND SETUP OF THE MARS SIMULATOR

A. Downloading the MARS simulator

The MARS simulator can be downloaded with a web browser or other download utility at this site: http://courses.missouristate.edu/KenVollmar/mars/download.htm. Please note that this simulator requires that the computer running it having at least Java J2SE 1.5 SDK or later being previously installed.

B. *Running the MARS simulator and opening project files*

Since the MARS simulator is a Java program, one will need to run it as one to open the program successfully. Once opened, the window will appear blank and present the user with the simulator interface.

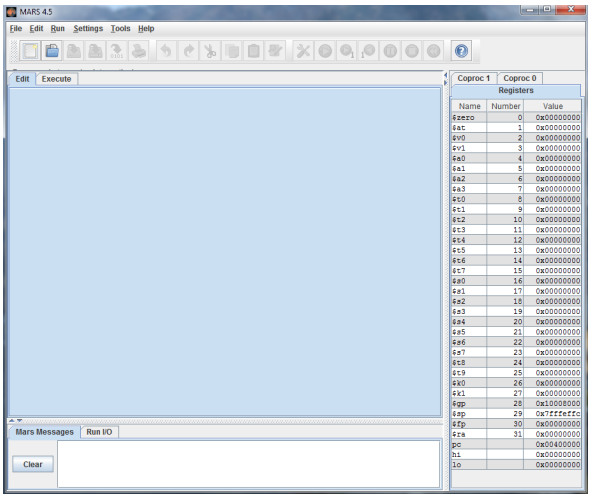

Fig. 1. The user interface

Next, make sure to download the zip archive from the Canvas page containing all appropriate assembly language files. Download this file to a specified directory that is designated for a workspace, as these files need to be easily accessible. Unzip the archive using your OS' extractor utility or use a free alternative like 7-Zip for Windows OS, for example. Once extracted, the folder will have the necessary files that can be easily accessed.

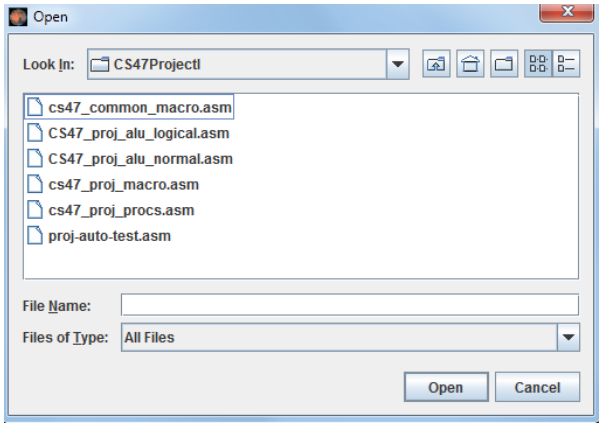

Fig. 2. The file open menu with assembly project files

1. *proj-auto-test.asm.* This file serves to test one's project with test cases and provides a score based on its analysis.

2. *cs47_proj_procs.asm.* This file has more macros that all relate to printf.

3. *cs47_proj_macro.asm.* This file is to be completed by the student with macros that assist in the operation of the calculator.

4. *CS47_proj_alu_normal.asm.* This file is to be completed by the student with code that utilizes the MIPS arithmetic functions to compute basic mathematical operations (addition, subtraction, multiplication and division).

5. *CS47_proj_alu_logical.asm.* This file is to be completed by the student with code that utilizes basic logic operations and no MIPS arithmetic functions to compute mathematical operations (addition, subtraction, multiplication and division).

6. *cs47_common_macro.asm.* This file is not to be edited by the student, as it contains basic macros that are essential to the operation of the logical calculator.

C. *Proper configuration of the MARS simulator*

For the MARS simulator to work correctly for this project, the student must enable the option 'Assembles all files in directory' so that all necessary parts of the calculator are assembled and checked for errors before running. The student must also check the 'Initialize program counter to global main if defined' option so that the simulator will start at the address as defined by main. This is critical, because the program will, by default, start at the address it encounters first instead of the correct one that main defines, resulting in an error.

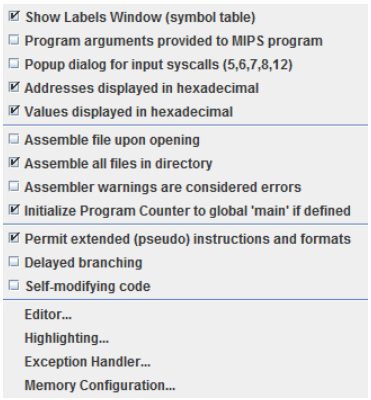

Fig. 3. The required options for successful operation

III. REQUIREMENTS FOR ARITHMETIC OPERATIONS

Two different types of implementations are required to complete this project. First, the student needs to implement a file called "CS47_proj_alu_normal", which consists of the four basic math operations, including: addition, subtraction, multiplication and division, but utilizes the built-in MIPS arithmetic operations to do so. On the other hand, the student must also implement a file named "CS47_proj_alu_logical", which consists of the same four, basic arithmetic operations, but only uses low-level logical operations to complete them, making no use of the built-in MIPS arithmetic operations in direct relation to the final answer.

A. *Normal ALU Operation*

1. *Register $a0.* This register receives the input of the first argument to eventually use in an arithmetic operation.

2. *Register $a1.* This register receives the input of the second argument to eventually use in an arithmetic operation.

3. *Register $a2.* This register receives the input for the desired arithmetic operation as a single character code, such as '+' for addition, '-' for subtraction, '*' for multiplication, and '/' for division.

4. *Register $v0.* This register will store the output of the addition and subtraction operations, as well as the 'LO' register in multiplication and the quotient for division.

5. *Register $v1.* This register will store some of the output from multiplication and division operations and the overflow bit for addition. For multiplication, it will store the 'HI' register and for division, it will store the remainder.

Depending on the type of arithmetic operation, the output of the result will be stored in certain registers. For addition and subtraction, the result will be stored in the $v0 register and $v1 will hold the overflow bit if applicable. For multiplication and division, two registers will be used, $v0 and $v1. Regarding multiplication, since the answer will be in 64-bit and MIPS is a 32-bit system, the answer must be split into 'HI' and 'LO' registers, each containing 32-bits. The LO register will be stored in the $v0 register and the HI register will be stored in the $v1 register, so there

will be two outputs. Regarding division, the result of the quotient will be stored in the $v0 register and the remainder will be stored in the $v1 register.

B. *Logical ALU Operation*

1. *Register $a0.* This register receives the input of the first argument to eventually use in an arithmetic operation.

2. *Register $a1.* This register receives the input of the second argument to eventually use in an arithmetic operation.

3. *Register $a2.* This register receives the input for the desired arithmetic operation as a single character code, such as '+' for addition, '-' for subtraction, '*' for multiplication, and '/' for division.

4. *Register $v0.* This register will store the output of the addition and subtraction operations, as well as the 'LO' register in multiplication and the quotient for division.

5. *Register $v1.* This register will store some of the output from multiplication and division operations. For multiplication, it will store the 'HI' register and for division, it will store the remainder.

While the end result of both normal and logical ALU operations will be identical, the logical operation will require many low-level logic operations to complete the same task in the normal implementation of the ALU.

IV. THE DESIGN OF THE CALCULATOR

A. *Normal ALU Operation*

Completing the normal ALU operation requires that basic mathematical operations are completed using the built-in MIPS arithmetic. As described before, the two operands will be input into the two registers, $a0 and $a1, while the desired operation will be input into the $a2 register. In the next column is the implementation of the normal ALU operation.

1. *funcsum_norm.* This corresponds to the '+' character code for the register $a2, which will output the sum of the two operands, storing it in $v0.

2. *funcdiff_norm.* This corresponds to the '-' character code for the register $a2, which will output the difference of the two operands, storing it in $v0.

3. *funcprod_norm.* This corresponds to the '*' character code for the register $a2, which will output the LO register of the result to $v0 and the HI register of the result to $v1.

4. *funcdiv_norm.* This corresponds to the '/' character code for the register $a2, which will output the quotient to $v0 and the remainder to $v1.

```
beq $a2 '+' funcsum_norm
beq $a2 '-' funcdiff_norm
beq $a2 '*' funcprod_norm
beq $a2 '/' funcdiv_norm

# Arithmetic using normal MIPS operations

funcsum_norm:
add     $v0 $a0 $a1
j       exitproc_norm

funcdiff_norm:
sub     $v0 $a0 $a1
j       exitproc_norm

funcprod_norm:
mult    $a0 $a1
mfhi    $v1
mflo    $v0
j       exitproc_norm

funcdiv_norm:
div     $a0 $a1
mfhi    $v1
mflo    $v0
j       exitproc_norm
```

Fig. 4. Implementation of normal ALU operations using built-in MIPS arithmetic

B. *Logical ALU Operation*

The operation of the logical ALU will require the use of some student-written macros, since the student cannot utilize the built-in arithmetic operations in MIPS. Since the logical ALU will require changing and looking into specific bits, a macro that can read a specific bit and one that inserts a bit to the desired location will be required.

1. *extract_nth_bit.* This macro will retrieve the desired bit at a certain location from a source bit pattern.

```
.macro extract_nth_bit($regDest, $regSrc, $regTarg)
        srlv    $regSrc $regSrc $regTarg        # Shifts $regSrc by the target amount and stores it back :
        li      $regDest 1                      # Sets $regDest to 1
        and     $regDest $regSrc $regDest
.end_macro
```

Fig. 5. Implementation of the extract_nth_bit macro

2. *insert_to_nth_bit.* This macro will insert the desired bit pattern at a certain location into the source bit pattern.

```
.macro insert_one_to_nth_bit($regDest, $regSrc, $regTarg, $maskReg)
        move    $maskReg $regT                  # Moves value in $regT to $maskReg
        sllv    $maskReg $maskReg $regSrc       # Shifts $maskReg by $regSrc
        or      $regDest $regDest $maskReg
.end_macro
```

Fig. 6. Implementation of the insert_to_nth_bit macro

Now that the proper macros have been implemented, the necessary procedures for operating the basic operations can now be implemented.

1. *func_sum_logi*. This algorithm will take care of the addition of two arguments. It will make use of the half-adder and full-adder mechanisms.

Binary Two Single Bit Addition Result

Bit 1 (A)	Bit 2 (B)	Sum Bit (Y)	Carry Bit (C)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Half Addition

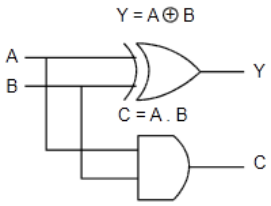

Fig. 8. The Half-Adder Mechanism[1]

Of course, since this operation will require the use of more than just one sum with only one bit, a full adder mechanism will need to be used. Full adders can handle a sum, a carry in bit and a carry out bit, making it capable of performing a more complex addition operation.

Binary Three Single Bit Addition Result

	Bit 1 (CI) Carry In	Bit 2 (A)	Bit 3 (B)	Sum Bit (Y)	Carry Bit (CO) Carry Out
m0	0	0	0	0	0
m1	0	0	1	1	0
m2	0	1	0	1	0
m3	0	1	1	0	1
m4	1	0	0	1	0
m5	1	0	1	0	1
m6	1	1	0	0	1
m7	1	1	1	1	1

$Y = \Sigma m(1,2,4,7)$

$CO = \Sigma m(3,5,6,7)$

Full Addition

Fig. 9. The truth table and minterms for sums and carry out bits[1]

In Fig. 9, one can see that the top half of the truth table is the same as a half adder's and that the table correctly adds the two bits considering every combination of carry in bit, and the two states of bit A and B. In half of the scenarios, there will be no carry out bit and in the other half, there will be.

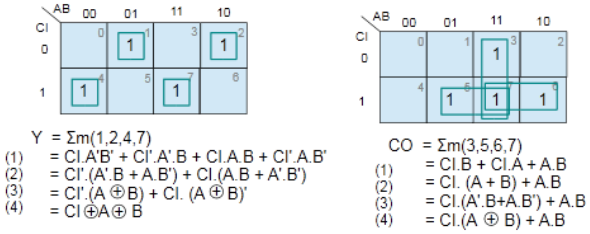

Fig. 10. The K-Map representation of the minterms in Fig. 9.[1]

As the K-Maps in Fig. 10 show, the sum bits can be simplified to bitwise XOR operations and the carry bits can be simplified to one bitiwse XOR and one bitwise AND operation.

$Y = CI \oplus (A \oplus B)$

$CO = CI.(A \oplus B) + A.B$

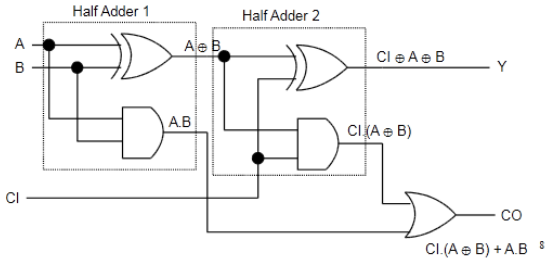

Fig. 11. Drawing of the full-adder mechanism[1]

The full-adder above in Fig. 11. comprises of the two half-adders as well as another gate for the carry out bit. However, one must use a more complicated circuit using multiple full-adders to be able to carry out many-bit sum operations.

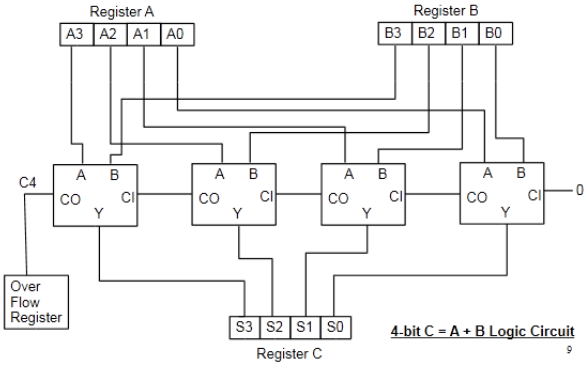

Fig. 12. The multi-bit adder, comprised of multiple full-adders[1]

In Fig. 12, we can see many single-bit full-adder mechanisms put together to compute multi-bit sums.

They are interconnected and allow for the carry out bit to be transferred to the carry in bit of the next adder. For subtraction, one can utilize the same mechanism, except to invert the second argument into two's complement, making it negative. This can be done by simply negating it and adding one.

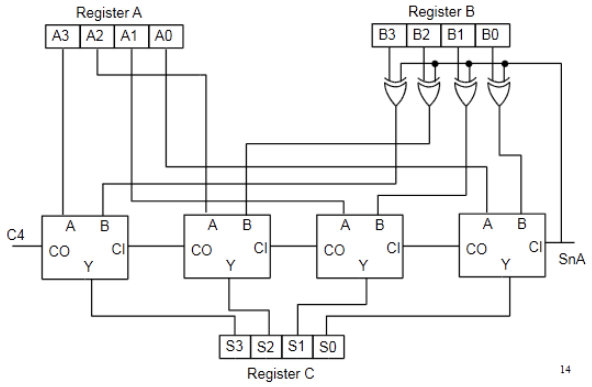

Fig. 13. The multi-bit adder/subtractor, similar to Fig. 12.[1]

Now for the multi-bit adder/subtractor mechanism in Fig. 13, multiple XOR gates are utilized for each operation that's needed. This process will loop thirty-two times since the MIPS system is 32-bit. To make the main code cleaner, a macro to implement the half and full-adder are quite convenient.

```
# Half-Add Util
.macro half_add_util($argl, $arg2, $sum, $car)
        xor     $sum $argl $arg2
        and     $car $argl $arg2
.end_macro
```

Fig. 13. The Half-Adder implementation

```
# Full-Add Util
.macro full_add_util($argl, $arg2, $sum, $prevcar, $carin, $carout)
       half_add_util($argl $arg2 $sum $prevcar)
       and     $carout $carin $sum
       xor     $sum $carin $sum
       xor     $carout $carout $prevcar
.end_macro
```

Fig. 14. The Full-Adder implementation

In Fig. 13, the code is a translation of the logic gates in Fig. 8, doing a bitwise XOR of the two arguments to find the sum and a bitwise AND operation to get the carry. In Fig. 14, the half-adder is called using the same parameters as the full-adder and the operation is conducted in a similar fashion as the half-adder. The logic for combining both. Using this these techniques, my implementation of the addition function is as follows:

```
li      $s0 0 # Final Answer
li      $t0 0 # Index counter
li      $t3 0 # Carry-in
li      $t4 0 # Carry-out

# $t3 = Carry out
# $t4 = Carry in

funcsum_loop:
beq     $t0 32 funcsum_loop_end

retrieve_first_bit($s1 $a0)
retrieve_first_bit($s2 $a1)

full_add_util($s1 $s2 $s3 $s4 $t4 $t3)
put_one_to_nth_bit($s0 $t0 $s3 $t4)

move    $t4 $t3
addi    $t0 $t0 1
j       funcsum_loop

funcsum_loop_end:
move    $v0 $s0
move    $v1 $t3
```

Fig. 15. My implementation of the addition function

First, all of the temporary registers are initialized to zero so that they have a fresh start, then the addition loop is initiated. Before anything, a check is made to the counter to see if it has looped thirty-two times, if so, it must be done and will jump to the end to output the answer. However, if the counter is not at thirty-two, then the procedure will fetch the first bit of each of the operands. This is the first step in the addition process, so it is fed into the arguments of the full-adder macro and it utilizes the half-adder mechanisms to compute the final sum. The carry out of the last operation is then moved to the carry-in of the next operation until the entire operation is finished at thirty-two loops. The sum is output to the $v0 register and any possible overflow after is output to the $v1 register.

2. *funcdiff_logi*. funcdiff_logi is a simple operation for subtraction that simply inverts all of the bits in $a1 and then jumps to the funcsum_logi function. This works because the second operand is turned into a "negative" number in two's complement so the addition of a positive number and negative number is the same as subtracting the second operand from the first. My implementation is as follows:

```
neg     $a1 $a1         # Invert $a1
jal     funcsum_logi
```

Fig. 16. My implementation of the subtraction function

3. *funcprod_logi_signed*. The multiplication function is multitudes more complex than the addition and subtraction, requiring compatibility with two thirty-two bit values to output a sixty-four bit value with the lower bits output to $v0 and the higher bits output to $v1. First, I will go over all of the utilities that my function depends on before I go into the main function.

3A. *conv_2s_comp*. This simple function takes in an $a0 argument and outputs the two's complement version of it. It negates the argument, sets the $a1 argument to 1 and then jump and links to the funcsum_logi function.

```
not     $a0 $a0
addi    $a1 $zero 1
jal     funcsum_logi
```

Fig. 17. My implementation of the two's complement utility

3B. *conv_2s_comp_neg*. This function first checks if the argument passed in is greater than zero, in which case the number is already in two's complement form and should be preserved. This number would be output immediately and the function would end. However, if it is less than zero, or a negative number, then converting to two's complement is necessary. The function jump and links to the conv_2s_comp utility, and after that is done, it jumps straight to the end of the function to output the now-two's complement number.

Fig. 18. My implementation of the two's complement if negative utility

```
bgez    $a0 preserve_2s_comp
jal     conv_2s_comp
j       conv_2s_comp_neg_done

preserve_2s_comp:
move    $v0 $a0

conv_2s_comp_neg_done:
lw      $a0 16($sp)
lw      $ra 12($sp)
lw      $fp 8($sp)
addi    $sp $sp 16

jr      $ra
```

3C. *conv_2s_comp_sixfour*. This function considers the first operand as the "LO" register and the second operand the "HI" register. It negates both of these then saves them to saved registers for safe storage. It then sets $a1 to 1 and calls the funcsum_logi function to see if the LO will overflow into the HI register. If it does, the 1 will be saved in the overflow $v1 from the funcsum_logic function. Then it saves the sum of the call to $s1 and sets $a0 to the overflow bit, which is either 0 or 1. After this, it then sets the HI to the previously saved LO value. Finally, it calls the funcsum_logi to add everything all together. Lastly, it sets the result from the operation to the $v1 and the sum to $s1.

```
not     $a0 $a0         # Negates LO
not     $a1 $a1         # Negates HI
move    $s3 $a1         # Saves HI to $s3
move    $s0 $a0         # Saves LO to $s0

addi    $a1 $zero 1     # Sets HI to 1
jal     funcsum_logi
move    $s1 $v0         # Sets sum to $s1
move    $a0 $v1         # Sets LO to overflow bit

move    $a1 $s3         # Sets HI to saved LO

jal     funcsum_logi

move    $v1 $v0
move    $v0 $s1
```

Fig. 19. My implementation of the sixty-four bit two's complement adding

Now, since the signed multiplication function relies on the unsigned function, I will go over this one first.

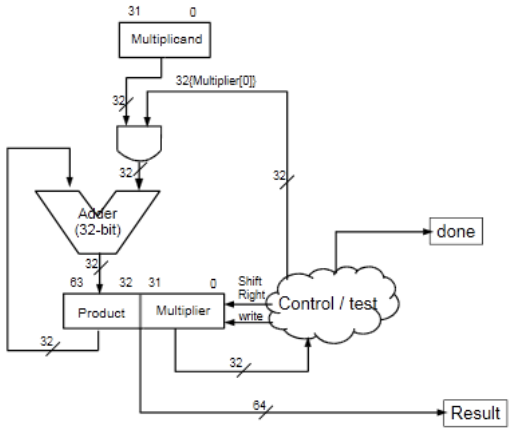

Fig. 20. The basis of the multiply function in circuit form[2]

Fig. 20. shows the essential structure of the multiplication function in a logical circuit format and is condensed as much as possible to reduce load. A flow chart is also another great way to visualize the process going on behind the multiplication function, which can be represented as such:

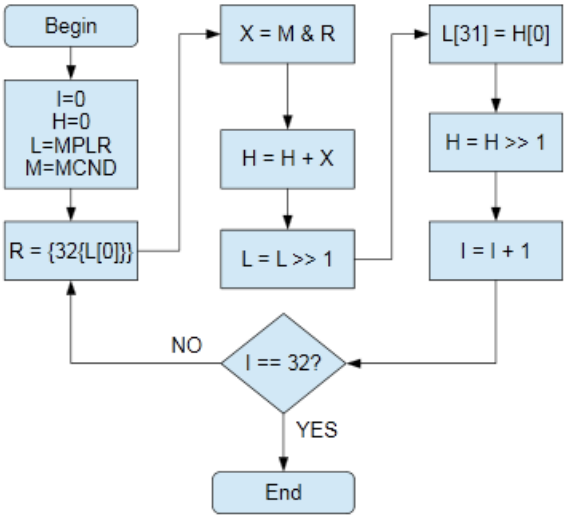

Fig. 21. The simplified flow-chart of the unsigned multiplication procedure[2]

Looking at this representation, it is much easier to see how the underlying process works and how one can implement it in MIPS. I will now walk through my implementation of the multiplication function, called funcprod_logi:

```
move    $s2 $a0 # Stores multiplicand
move    $s0 $a1 # Stores multiplier

# Initialize variables
add     $s1 $zero $zero
add     $s3 $zero 1
sll     $s3 $s3 31
add     $s5 $zero $zero
add     $s6 $zero $zero

funcprod_logi_unsigned_looper:
beq     $s1 33 funcprod_logi_unsigned_done

retrieve_first_bit($t0 $s0) # Get first bit of current multiplier
srl $s5 $s5 1 # Shift LO register to make room for incoming HI register bit
retrieve_first_bit($t1, $s6) # Get first bit of HI
beq $t1 1 add_to_LO
continue_1:
beq $t0 1 add_to_HI
continue_2:
addi $s1 $s1 1 # Increment counter
j       funcprod_logi_unsigned_looper

add_to_LO:
add $s5 $s5 $s3 # Adds 31 bit-shifted to left 1 value
j       continue_1

add_to_HI:
add $s6 $s6 $s2 # Adds multiplicand to HI if LSB of multiplier is 1
j       continue_2

funcprod_logi_unsigned_done:
move    $v0 $s5
move    $v1 $s6
```

Fig. 22. My implementation of multiplication

To begin, I labeled the necessary registers I needed, like for storing the multiplicand and the multiplier so that the originals do not get modified. I then initialize some essential variables I need, like a register that holds a value of one shifted left to the 31st position, which I will explain shortly. Registers for the LO and the HI final answers are also needed in my function. I initiate a loop that will repeat thirty-three times starting from a counter value of zero. The first task is to retrieve the least significant bit (LSB) of the current multiplier and store it. I then shift the LO register to the right one bit since it must be shifted, but also to make room for an incoming bit if the LSB of the HI register must be transferred to the highest bit of the LO register since they are linked. This is where the register with the one value in the 31st bit spot comes into play. Once the LO register is shifted to the right once, a zero is padded to the leftmost side, and if the LSB of the HI must also be shifted to the right, then simply adding this value to the LO acts as "transferring" it to the LO register, hence why if the LSB of the HI is one, I add this value to the LO register. If the LSB of the HI is not one, then it simply passes to the next check, which sees if the first bit of the current multiplier is one or zero. If it is one, then the multiplicand needs to be added to the HI. If it is not equal to one, then the function will simply increment the counter and jump back to the beginning of the loop. This will repeat until the final answer in sixty-four bits is calculated and output correctly to $v0 and $v1.

4. *funcprod_logi_signed*. This function takes into account negative numbers being input into the operands. A circuit diagram of the process is shown here:

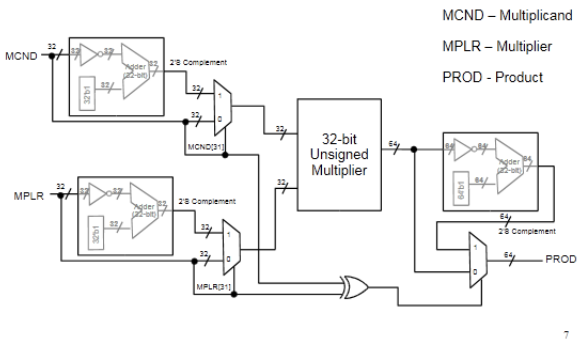

Fig. 23. The signed multiplication circuit diagram[2]

The signed multiplication function first saves both operands into two saved registers and two temporary registers. Then I set another saved register as 31 to be able to get the MSB of the multiplier and multiplicand to check for sign type. Once I obtain the sign types based on the MSB of both operands, I then proceed to run the operands through the "two's complement if negative" procedure utilizing jump and link to return back where it left off. After both operands are run through the procedures, they are ready to be put through the regular, unsigned multiplication procedure.

However, I check for the ending result of the sign for the product by doing a bitwise XOR on the two MSB's of the two operands. If both are negative (one and one), then the product will be positive with an MSB of 0; however, if there is any other case except both positive, then the final product will be negative. I then execute the unsigned multiplication procedure with the new two's complement arguments. After this, I save the results, $v0 and $v1, to two saved registers that serve as the holders for the final answers. At this point, I check if the result from the bitwise XOR is zero or one. If it is zero, then the procedure is finished, and can output the answers, but if it is one, then the answer will need to be turned to negative. I transfer the two saved registers that have the outputs of the previous multiplication operation into $a0 and $a1 to utilize the "conv_2s_comp_sixfour" function. After this, the answers are output.

4. *funcdiv_unsigned.* This procedure will have two functions related to it. It will call the signed version of the division, which itself will call the unsigned version, similar to how the multiplication function works. Since it is so similar, I will explain it like so, with unsigned coming first.

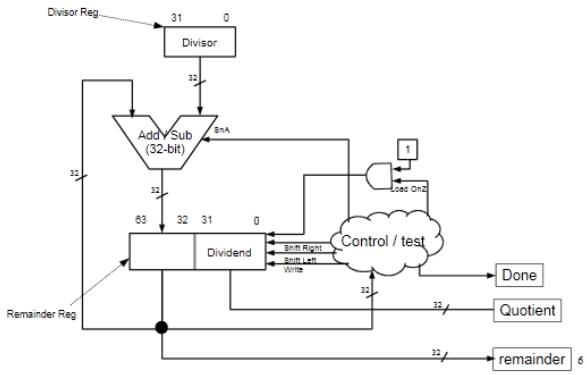

Fig. 24. The circuit diagram for the overall division function[3]

In Fig. 24, the basic division operation here is laid out logically into a circuit diagram with various logic gates and it relies on the addition and subtraction functions working correctly as well. The divisor is put into a thirty-two bit register that never changes and the remainder and quotient registers are linked similar to a HI and LO register setup. However, the dividend is first loaded into the quotient register, which is the "LO" part. As such, in the case that the MSB of the quotient is one, there must be check for the 31st bit to make sure that data is not lost. If the MSB of the quotient is one, then an operation is made to add one to the remainder register *after* it has been shifted already. This way there is no conflict in the LSB of the remainder. This will run thirty-two iterations until it reaches the final answer for the quotient and the remainder. The translation of this circuit diagram can be seen in this flow chart:

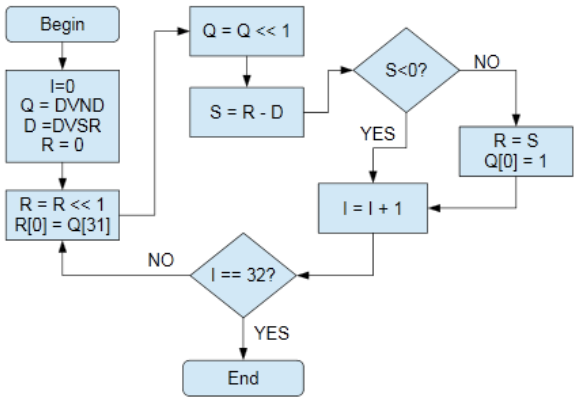

Fig. 25. Flow chart representation of the logic circuit in Fig. 24.[3]

Using this flow chart as a guide, this is my implementation of the unsigned division function.

```
        add     $s0 $zero $zero          # Initialize remainder to 0
        add     $s1 $zero $zero          # Initialize quotient to 0
        add     $s2 $zero $zero          # Initialize divisor to 0
        add     $s3 $zero $zero          # Initialize counter to 0
        add     $s4 $zero $zero          # Initialize temporary storage variable to 0
        add     $s5 $zero 31

        move    $s1 $a0                  # Copies dividend to quotient register
        move    $s2 $a1                  # Copies divisor to divisor register

        funcdiv_logi_unsigned_looper:
        beq     $s3 32 func_logi_unsigned_done          # Check if counter has hit 33

        sll     $s0 $s0 1                # Shifts remainder to the left by one to make room
        move    $t3 $s1                  # Make copy of quotient to check MSB
        sll     $s1 $s1 1                # Shifts quotient by 1 to the left
        retrieve_nth_bit($t1 $t3 $s5)
        beq     $t1 1 add_one_to_remainder              # Jumps out if MSB of quotient is 1 for special case
        continue_on:
        move    $a0 $s0                  # Set $a0 to current remainder to get ready for subtraction
        jal     funcdiff_logi            # Subtract divisor from remainder
        move    $s4 $v0                  # Copy partial subtraction to temp storage to check
        bltz    $s4 funcdiv_logi_loop_end               # If the difference is negative, branch to shift the divisor right away

        shift_divisor_right_pos:
        move    $s0 $s4                  # Set remainder as result of positive difference
        addi    $s1 $s1 1                # Adds one to quotient register
        j       funcdiv_logi_loop_end

        add_one_to_remainder:
        addi    $s0 $s0 1                # Adds 1 to remainder when MSB of quotient is 1
        j       continue_on              # Jump back to negative case

        funcdiv_logi_loop_end:
        addi    $s3 $s3 1                # Increments counter by 1
        j       funcdiv_logi_unsigned_looper

func_logi_unsigned_done:
        move    $v0 $s1
        move    $v1 $s0
```

Fig. 26. My implementation of the unsigned division function

My implementation first labels all registers with what I will be using them for. One is dedicated to the remainder, another is for the quotient, and lastly one for the divisor. I then set a counter to zero and have a register hold thirty-one so that I can check the MSB of a register if needed. I then make saved copies of the dividend and the divisor to manipulate later. The main loop is then started, first checking if the counter has hit

thirty-two loops. I first shift the remainder one bit to the left since it is required to, but it also serves to make room for any possible incoming HI bit in the MSB. Since the extraction of a certain bit shifts the target register, I make a temporary copy of the quotient to evaluate. If the MSB of the quotient is one, then I add one to the remainder to transfer it over. After this mini-procedure, it jumps back to where it left off and sets the current remainder to $a0 to get it ready for subtraction using funcdiff_logi. funcdiff_logi is then executed and the difference is stored into $s4 for evaluation. It is checked to see if it is negative or above zero, branching to the appropriate procedure if it is the former. If the difference is positive, then the result of that difference is then made the remainder and one is added to the quotient register, upon which the iteration has finished. This will loop thirty-two times and then copy the quotient to $v0 and the remainder to $v1.

5. *funcdiv_signed*. This function performs a similar operation to the signed version of the signed multiplication where it first makes the two operands two's complement if it is necessary, except additional operations are done on the resulting quotient and remainder after the unsigned division function is called.

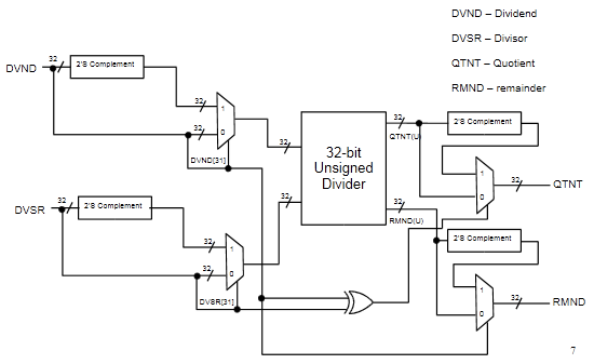

Fig. 27. The signed division circuit diagram[3]

This signed division circuit illustrates the process and flow of the numbers through the function. After computing the quotient and remainder, a process is used to determine whether to modify them or not. First, I check the MSB of the dividend and divisor and store them into saved registers. After this, I XOR them and save the result into another register. The result will dictate whether the quotient should be converted to two's complement or not. If the result is one, then this means one of the operands is positive and the other is negative, meaning that the quotient should be negative, so the quotient is put through the two's complement procedure. After this is finished, it is saved to the final answer and the remainder is now evaluated. The MSB of the dividend is then evaluated and if it is one (negative), then the remainder needs to also be negative, so it is converted to two's complement. If the dividend's MSB is zero, then it is left alone and passed to the final answer output.

V. TESTING ALL FUNCTIONS

After all of the functions are implemented, all necessary files originally in the project folder need to be assembled, which should happen automatically once the assemble function is clicked in the MARS simulator, assuming that all proper settings are set. The testing file will then execute and compare the results of the au_normal output and the results of the au_logical output, which should be identical if implemented properly.

```
(4 + 2)         normal => 6     logical => 6    [matched]
(4 - 2)         normal => 2     logical => 2    [matched]
(4 * 2)         normal => HI:0 LO:8     logical => HI:0 LO:8    [matched]
(4 / 2)         normal => R:0 Q:2       logical => R:0 Q:2      [matched]
(16 + -3)       normal => 13    logical => 13   [matched]
(16 - -3)       normal => 19    logical => 19   [matched]
(16 * -3)       normal => HI:-1 LO:-48          logical => HI:-1 LO:-48         [matched]
(16 / -3)       normal => R:1 Q:-5      logical => R:1 Q:-5     [matched]
(-13 + 5)       normal => -8    logical => -8   [matched]
(-13 - 5)       normal => -18   logical => -18          [matched]
(-13 * 5)       normal => HI:-1 LO:-65          logical => HI:-1 LO:-65         [matched]
(-13 / 5)       normal => R:-3 Q:-2     logical => R:-3 Q:-2    [matched]
(-2 + -8)       normal => -10   logical => -10          [matched]
(-2 - -8)       normal => 6     logical => 6    [matched]
(-2 * -8)       normal => HI:0 LO:16    logical => HI:0 LO:16   [matched]
(-2 / -8)       normal => R:-2 Q:0      logical => R:-2 Q:0     [matched]
(-6 + -6)       normal => -12   logical => -12          [matched]
(-6 - -6)       normal => 0     logical => 0    [matched]
(-6 * -6)       normal => HI:0 LO:36    logical => HI:0 LO:36   [matched]
(-6 / -6)       normal => R:0 Q:1       logical => R:0 Q:1      [matched]
(-18 + 18)      normal => 0     logical => 0    [matched]
(-18 - 18)      normal => -36   logical => -36          [matched]
(-18 * 18)      normal => HI:-1 LO:-324         logical => HI:-1 LO:-324        [matched]
(-18 / 18)      normal => R:0 Q:-1      logical => R:0 Q:-1     [matched]
(5 + -8)        normal => -3    logical => -3   [matched]
(5 - -8)        normal => 13    logical => 13   [matched]
(5 * -8)        normal => HI:-1 LO:-40          logical => HI:-1 LO:-40         [matched]
(5 / -8)        normal => R:5 Q:0       logical => R:5 Q:0      [matched]
(-19 + 3)       normal => -16   logical => -16          [matched]
(-19 - 3)       normal => -22   logical => -22          [matched]
(-19 * 3)       normal => HI:-1 LO:-57          logical => HI:-1 LO:-57         [matched]
(-19 / 3)       normal => R:-1 Q:-6     logical => R:-1 Q:-6    [matched]
(4 + 3)         normal => 7     logical => 7    [matched]
(4 - 3)         normal => 1     logical => 1    [matched]
(4 * 3)         normal => HI:0 LO:12    logical => HI:0 LO:12   [matched]
(4 / 3)         normal => R:1 Q:1       logical => R:1 Q:1      [matched]
(-26 + -64)     normal => -90   logical => -90          [matched]
(-26 - -64)     normal => 38    logical => 38   [matched]
(-26 * -64)     normal => HI:0 LO:1664          logical => HI:0 LO:1664         [matched]
(-26 / -64)     normal => R:-26 Q:0     logical => R:-26 Q:0    [matched]

Total passed 40 / 40
*** OVERALL RESULT PASS ***

-- program is finished running --
```

Fig. 28. The final results of testing

VI. ISSUES ENCOUNTERED

A. *Not putting jr $ra after stack unload*

Towards the beginning of the project, when I was testing the addition and subtraction functions, I saw that the result of the test was somehow a 40/40 even though I had not implemented the multiplication or division functions. Immediately realizing that something was wrong, I realized that I had not put 'jr $ra' after each stack unload, which led it to go back to

compute au_normal as the au_logical results, leading to a perfect score even though au_logical does not implement all of the arithmetic functions yet.

B. *Not initializing and using temporary registers when not appropriate*

Another issue that I ran into was storing important values into temporary registers as well as not initializing them to zero before storing values. Since I use the add function to store values into the temporary registers (nothing related to computing the final result), if the temporary registers already have a value in them (which they usually do), the register will not hold the desired value. Therefore, it is right to initialize the temporary registers to zero before setting them to any desired value. Furthermore, I stored values into these registers that would be wiped/changed when I need to use them, leading to an incorrect result. With certain stored values that are important, a saved register that does not clear would be more appropriate.

C. *Not using jump and link correctly*

When using the 'jal' instruction, it takes the current arguments and uses them as inputs in the procedure being called, then outputs the result to $v0. I had not known that the jal instruction worked this way, leading to many incorrect computations. The correct way to use this instruction is to copy the desired operands into the $a0/$a1 registers (whichever is applicable) and then call jal. Then, copy the result stored in $v0/$v1 (whichever is applicable) to any desired stored/temporary variables.

D. *Not commenting each register/line with its function/ purpose*

Towards the beginning of the project, I neglected to label the registers and non-obvious functions with their purpose. Since I do not have the best memory, I quickly lost track of which lines were doing what operations and which registers stored what values. After labeling these, continuing through the project became much easier.

E. *Retrieving bit macros shifting target register*

A frequently occurring issue I had when retrieving bits from registers was that after retrieving the desired bit and saving it to a register, the register that I was looking at got shifted to the right, meaning any data in the LSB was cut off. This resulted in many of my registers being zero or another undesirable value. To remedy this, I copied the source register to a temporary variable and used the temporary variable as the source to retrieve the desired bit, then the original register is not shifted and still retains the original value.

F. *Losing data in the MSB or LSB when shifting*

Whenever a function required the use of two registers being linked like HI and LO, it is imperative to check if the LSB or the MSB is set to any value depending on the shift direction and situation so that it can be inserted into the connected register. When I first implemented the multiplication function and shifted the HI and LO registers to the right, I failed to check the LSB of the HI and lost data that should have been transferred to the MSB of the LO. Therefore, I first did a check of the LSB of the HI by retrieving the bit and branched to a mini-procedure to add one to the MSB of the LO, then shift the HI register to the right. I then used this logic in every procedure where it was needed.

VII. CONCLUSION

All in all, this project utilized absolutely every concept that has been taught in this course thus far, it was truly comprehensive. I spent hours looking at the screen and going through loops in my head to see how the computer would walk through the procedure as well as utilizing the debugger very frequently to keep track of saved values and see how operations were executing to make sure there were doing what I had designed them to do. I certainly reinforced concepts that I was familiar with and became well-versed in concepts that I was not very knowledgeable about. This project revealed to me the ins and outs of the MIPS assembly language and how to use it to its fullest potential. Every error I ran into taught me something new about how MIPS handles various components of its registers and operations. At the end of this project, I have thoroughly strengthened my skills in MIPS and programming in general, and it has made me a better programmer in every aspect.

REFERENCES

[1] K. Patra. CS 47. Class Lecture, Topic: “Addition Subtraction Logic.” San Jose State University, San Jose, CA, November 25, 2018.
[2] K. Patra. CS 47. Class Lecture, Topic: “Multiplication Logic.” San Jose State University, San Jose, CA, November 27, 2018.
[3] K. Patra. CS 47. Class Lecture, Topic: “Division Logic.” San Jose State University, San Jose, CA, November 29, 2018