Implementation of Arithmetic Operations in MIPS Through Logical Operations

Shayan Asgari
San Jose State University
shayan.asgari@sjsu.edu

***Abstract*—This report focuses on the implementation of basic arithmetic operations including multiplication, division, addition, and subtraction through the use of MARS(MIPS assembler and runtime simulator).**

I. Introduction

The goal of this project is to depict how the processor carries out basic arithmetic operations, mainly through logical operators. At the core of the processor stands the ALU (Arithmetic Logic Unit). By using the MARS simulator, we will be able to depict how the ALU computes basic arithmetic calculations, not only through normal MIPS native instruction implementation (mult, div, add, sub), but through the use of logical operations (NAND, XOR, OR NOT).

The program written is MIPS which is an assembly language. The software used to write MIPS is MARS (MIPS Assembler and Runtime Simulator). MIPS is an IDE developed and released by Missouri State University. In this report we will use both the MARS simulator and the MIPS assembly language to implement basic arithmetic operations.

The report includes steps regarding the setup and installation of MARS to ensure they are in accordance with the report and not prone to errors. The report will also discuss and analyze the importance of logical operators as they pertain to arithmetic operations, and depict the program works correctly by comparing the results to the tester.

II. Installation and Setup

A. *Installation of MARS IDE*

Download the MARS IDE that uses MIPS assembly language from:

http://courses.missouristate.edu/KenVollmar/MARS/

B. *Starter FIles*

Download the starter files from:

https://sjsu.instructure.com/courses/1263903/assignments/4776422?module_item_id=9632934

The folder should include the files:

1) cs47_common_macro.asm
The file contains useful macros

2) cs47_proj_macro.asm
The file contains code for calling procedures

3) proj-auto-test.asm
The file contains tester for procedure

Two other files that will be implemented as a part of report:

4) cs47_proj_alu_logical.asm

5) cs47_proj_alu_normal.asm

Note:
Windows must unzip the file manually to ensure all files are extracted correctly by navigating to file explorer, locating the file and press “Extract All” located in the top bar.

Mac OS X: Simply click on file to unzip correctly.

C. *Opening of Files in Mars*

Open the Mars4_5.jar file. Then, click on “File” and then “Open”.

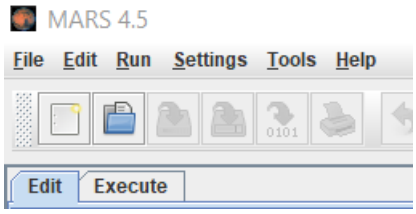

Fig 1. File Tab Location

Navigate to where you downloaded the now unzipped folder and open each .asm files one by one.

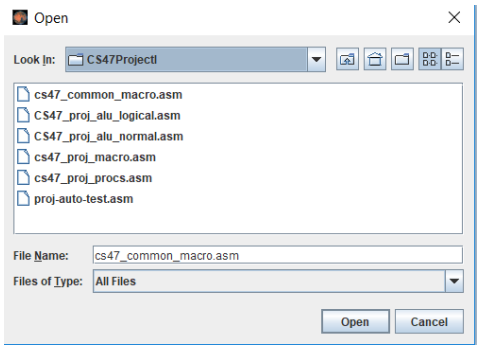

Fig 2. Opening Relative Files

After all files are opened there should be a total of five files and should look like:

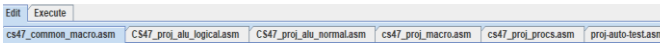

Fig 3. Expected Output Tabs

D. *Setting of MARS IDE*

Navigate to the "Settings" of the MARS IDE and in addition to the default settings, enable "Assemble all files in directory" and "Initialize Program Counter to global 'main' if defined." By assembling all files in directory, it will prevent run-time errors that would otherwise be inevitable by not assembling each individual file one by one. Next, by initializing the program counter to main, will allow the program to start at the address where 'main' occurs rather than the first line of code which in some cases will be ".include "./cs47_proj_macro.asm" and can lead to errors.

Ultimately, the settings should look like:

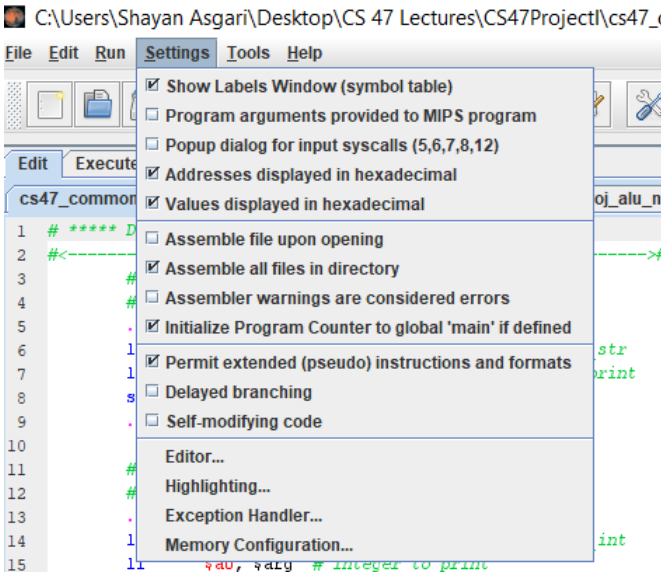

Fig 4. Required Settings

III. Implementation Of ARithmetic Operations

The two ways the arithmetic operations are implemented in this project is through a normal procedure and another using logical operators. Both functions should end up implementing basic calculator functions: multiplication, division, addition, and subtractions. However, the implementation of the two functions are significantly different and will be further explained as the report progresses.

A. *Normal Function*

The two ways the arithmetic operations are implemented in this project is through a normal procedure and another using logical operators. Both functions should end up implementing basic calculator functions: multiplication, division, addition, and subtractions. However, the implementation of the two functions are significantly different and will be further explained as the report progresses.

The normal function named "au_normal" will include one of the ways to implement the calculator functions explained above through native MIPS assembly language instructions. These include: mult, div, add, sub. The function takes in three arguments

1) *Register $a0*
   This register contains the first number of the mathematical operation
2) *Register $a2*
   This register contains the second number of the mathematical operation
3) *Register $a3*
   This register contains the specified operation code as an ASCIIZ value(Ends with null)

The result will be placed in register $v0, apart from multiplication and division. For multiplication $v1 will contain HI and $v0 will contain LO. For division $v1 will contain H I(the remainder), and $v0 will contain LO (the quotient).

B. *Logical Function*

The logical function named "au_logic" will include another way to implement the calculator functions previously explained by using and calling multiple functions as opposed to "au_normal" which can be done much more simply. Each of the functions in "au_logic" will manipulate the Boolean logic exemplified in AND, NOT and XOR. Moreover, this function will not be using the native MIPS assembly language mathematical operations such as mult, div, add, and sub in the core of its implementation. Equivalent to "au_normal" this also takes in three arguments:

1) *Register $a0*
   This register contains the first number of the mathematical operation
2) *Register $a2*
   This register contains the second number of the mathematical operation
3) *Register $a3*
   This register contains the specified operation code as an ASCIIZ value(Ends with null)

Also, the result will be placed in register $v0, apart from multiplication and division. For multiplication $v1 will contain HI and $v0 will contain LO. For division $v1 will contain H I(the remainder), and $v0 will contain LO (the quotient).

IV. Design and Implementation

A. *Normal Function*

The normal function consists of native MIPS assembly language mathematical operations. It is important to note that the implementation of "au_normal" is at its most simplistic form. While we know the purpose of the native functions, their implementation is hidden from the user. That phenomena will be explored later in the Logical Function. Determining the

native function is dependent on register $a2 which is equivalent to the ASCIIZ value of the operation. The operations $a2 can be include:

*1) *- ASCII value of 0x2A*
This operator denotes multiplication and will be executed using the MIPS native instruction of "mult" where the result will be in LO and overflow will be in HI.

2) / - ASCII value of 0x25
This operator denotes division and will be executed using the MIPS native instruction of "div" where the quotient will be in LO and the remainder will be in HI.

3) + - ASCII value of 0x2B
This operator denotes addition and will be executed using the MIPS native instruction of "add."

4) – - ASCII value of 0x2D
This operator denotes subtraction and will be executed using the MIPS native instruction of "sub."

Through the use of statements such as beq which is a part of the core instruction set for MIPS, the function could determine the arithmetic procedure and branch to a function depending on what the procedure was.

```
au_normal:
        addi $sp, $sp, -24
        sw $fp, 24($sp)
        sw $ra, 20($sp)
        sw $a0, 16($sp)
        sw $a1, 12($sp)
        sw $a2, 8($sp)
        addi $fp, $sp, 24

        beq $a2, 0x2A, MULTIPLICATION
        beq $a2, 0x2F, DIVISION
        beq $a2, 0x2B, ADDITION
        beq $a2, 0x2D, SUBTRACTION

MULTIPLICATION: mult $a0, $a1
                mfhi $v1
                mflo $v0 ##MOST IMPORTANT IN $V0
                j au_normal_end

DIVISION: div $a0, $a1
          mflo $v0 ##MOST IMPORTANT IN $V0
          mfhi $v1
          j au_normal_end

ADDITION: add $v0,$a0, $a1
          j au_normal_end

SUBTRACTION: sub $v0, $a0, $a1
             j au_normal_end

au_normal_end:
        lw $fp, 24($sp)
        lw $ra, 20($sp)
        lw $a0, 16($sp)
        lw $a1, 12($sp)
        lw $a2, 8($sp)
        addi $sp, $sp, 24

        jr      $ra
```

Fig 5. Branching Statements/Implementation for au_normal

B. Logical Function Parameters

When looking at the implementation of the logical function, it essentially comes down to logic. In the logical function, there is the presence of logical operators including AND, OR and NOT, there is also very one very essential operator, XOR. XOR becomes extremely useful when implementing addition and subtraction. Although XOR is not a fundamental Boolean operation, it becomes very important when implementing the full adder functionality of addition/subtraction.

Similar to the normal function, the logical function takes three arguments:

1) *Register $a0*
   This register contains the first number of the mathematical operation
2) *Register $a2*
   This register contains the second number of the mathematical operation
3) *Register $a3*
   This register contains the specified operation code as an ASCIIZ value(Ends with null)

IMPORTANT NOTES

While the result for addition and subtraction will be in register $v0, for multiplication LO will hold $v0 and HI will hold $v1. Likewise, for division, $v0 (quotient) will be in LO and $v1 (remainder) will be in HI.

C. Logical Function

The start of the program consists of branching statements similar to au_normal in order to determine the proper operation that entails each test case.

```
au_logical:

        beq $a2, 0x2B, add_logical
        beq $a2, 0x2D, sub_logical
        beq $a2, 0x2A, mult_signed
        beq $a2, 0x2F, div_signed
```

Fig 6. Branching Statements au_logical

As shown above if the argument of $a2 is addition, then it will jump to the symbol of add_logical, and if $a2 is subtraction, then it will jump to sub_logical. The importance of this unveils itself when we look at the subtraction operation.

1) Useful Maros

For the entirety of this project, two macros turn out to be extremely useful in the extraction and insertion of any bit into a register. For that reason, they are given their own place in a separate file called, "cs47_proj_macro.asm."

```
#$regD is what is returned as final
#$regS is what is to be extracted from
#regT is the bit position to be extracted
.macro extract_nth_bit($regD, $regS, $regT)
        addi $t8, $zero, 1
        sllv $t8, $t8, $regT
        and $regD, $regS, $t8
        srlv $regD, $regD, $regT
.end_macro

#$regD original bit pattern that 0 or 1 will be inserted in at $regS position
#$regS position to be modified [0-31]
#regT register that contains bit to insert which will be 0x0 or 0x1
#$maskReg register that is the ne
.macro insert_to_nth_bit($regD, $regS, $regT, $maskReg)
        addi $maskReg, $zero, 1
        sllv $maskReg, $maskReg, $regS
        not $maskReg, $maskReg
        and $regD, $regD, $maskReg
        sllv $regT, $regT, $regS
        or $regD, $regD, $regT
.end_macro
```

Fig 7. Extraction and Insertion Macros

2) add_logical

If $a2 is addition, we are going to load into $a2 the hexadecimal code of 0x0000000. Loading in a value into $a2 is mainly to decipher between the two operation so that prior to entering any computational logical operator, our program knows how to go about a certain operation; in this case it is addition.

We also have to store and restore the frame because we are changing the value of $a2 and according to MIPS argument passing registers must be preserved throughout calls.

```
add_logical:
        addi $sp, $sp, -20
        sw $fp, 20 ($sp)
        sw $ra, 16($sp)         #Have to save frame again because we will be changing $a2 based on operation
        sw $a2, 12($sp)
        sw $s1, 8($sp)
        addi $fp, $sp, 20

        lui $a2, 0x0000  #Set $a2 to 0x00000000 through lui and ori (MIPS Native Instructions)
        ori $a2 0x0000
        jal add_sub_logical

        lw $fp, 20 ($sp)
        lw $ra, 16($sp)         #Have to save frame again because we will be changing $a2 based on operation
        lw $a2, 12($sp)
        lw $s1, 8($sp)
        addi $fp, $sp, 20
        jr $ra
```

Fig 8. add_logical

Once $a2's values are loaded in, we jump and link to add_sub_logical. This is mainly to ensure our frame will be later restored up jumping register unconditionally.

3) sub_logical

If $a2 is subtraction, we are going to load into $a2 the hexadecimal code of 0xFFFFFFFF. Now, when our program enters our computational function it will know what action to take in changing one of the operands which we will later see.

We also have to store and restore the frame because we are changing the value of $a2 and according to MIPS argument passing registers must be preserved throughout calls.

```
sub_logical:
        addi $sp, $sp, -20
        sw $fp, 20 ($sp)
        sw $ra, 16($sp)         #Have to save frame again because we will be changing $a2 based on operation
        sw $a2, 12($sp)
        sw $s1, 8($sp)
        addi $fp, $sp, 20

        lui $a2, 0xFFFF  #Set $a2 all to zeros which signals addition then we can check if $a1 is negative
        ori $a2 0xFFFF  # $a2 indicates to perform addition
        jal add_sub_logical

        lw $fp, 20 ($sp)
        lw $ra, 16($sp)         #Have to save frame again because we will be changing $a2 based on operation
        lw $a2, 12($sp)
        lw $s1, 8($sp)
        addi $sp, $sp, 20
        jr $ra
```

```
RESTORE:
        move $v0, $s1 #Move the final answer to $v0
        move $v1, $t1 #Move carry in bit to $v1 *** Will be necessary
        lw      $fp, 32($sp)
        lw      $ra, 28($sp)
        lw      $a0, 24($sp)
        lw      $a1, 20($sp)
        lw      $a2, 16($sp)
        lw      $s0, 12($sp)
        lw      $s1, 8($sp)
        addi    $sp, $sp, 32

        jr $ra
```

Fig 9. sub_logical

A. add_sub_logical

This function computes the sum or difference between the operand of $a0 and $a1 depending on $a2 which is the hexadecimal code that

The understanding of addition and subtraction within the ALU first depends on the understanding of binary addition between two bits. A half adder shown below depicts the addition result of two single bits and a carry-out bit.

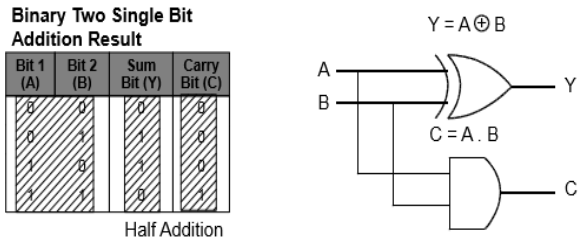

Fig 10. Half Adder Between Two Single Bits [1]

What we can see is that a half adder's sum is nothing more than an XOR between A and B, and the carry is out is just the AND between A and B. However, to come up with the proper implementation of addition and subtraction, we must consider a full adder because the carry-in bit becomes important when deciphering the two operations.

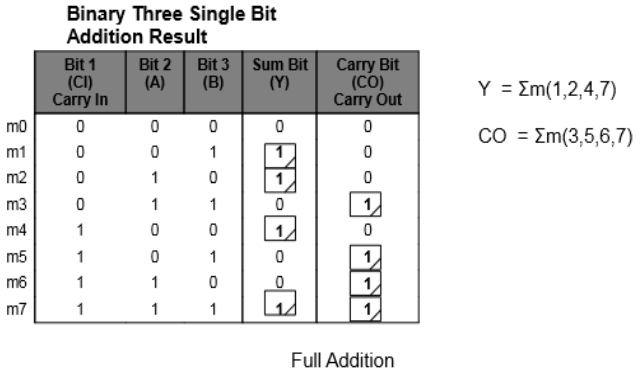

Fig 11. Full Adder Between Three Single Bits [1]

As Fig 10. depicts, the main difference between a half adder and a full adder is that a full adder considers a carry in bit which becomes essential when talking about subtraction. For subtraction, it is essential that the carry in bit starts at 1, while for addition the carry in bit starts at 0.

Once the minterms of the full adder are determined a K-Map is used to organize and reduce the equation to then be optimized

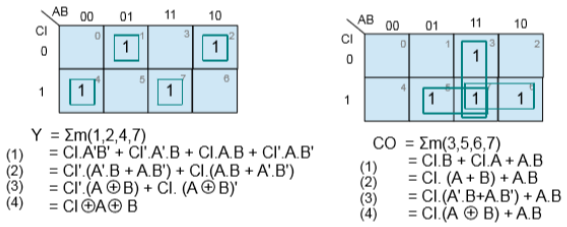

Fig 12. Karnaugh Map of Full Adder [1]

Once optimized, a circuit is constructed which effectively represents the logical circuit design for a full adder.

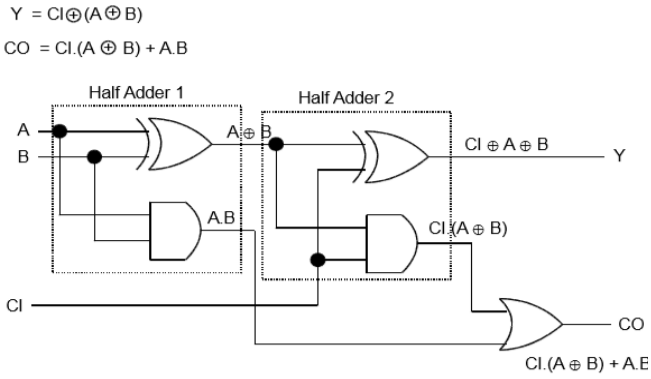

Fig 13. Logical Circuit Design of Full Adder [1]

As seen in Fig. 12, the implementation of a full adder depends on the implementation of two half adders. Also, the sum of Y is determined from doing a XOR between the carry-in bit, the first bit of the first operand and the first bit of the second operand. Lastly, the carry-out bit found from an OR of the bits produced from the two half adders.

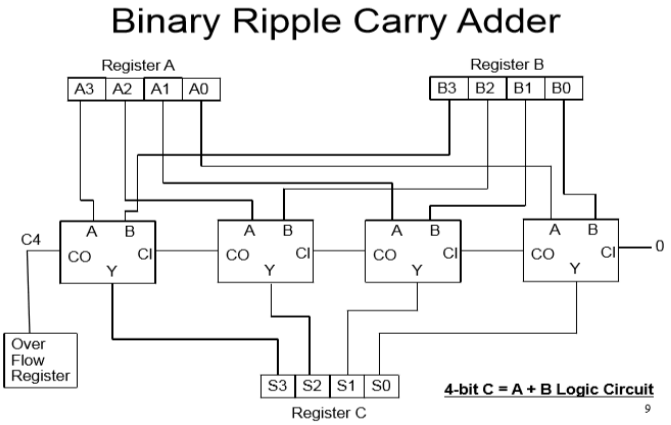

Fig 14. Logical Circuit Design of Full Adder [1]

Fig 12. Shows the logical circuit design of a full adder for a 4-bit integer. Note that the carry in is 0. In order to replicate this process for subtraction it is essential to represent each of the numbers of $a0 and $a1 in 2's complement form so that we are able to re-use the full adder circuit as a subtraction circuit.

The reason for using 2's complement is that it allows us to use the same circuit while our operation is not necessarily addition. Essentially, subtraction is addition.

If S = A – B, then S = A + INV(B) + 1. The inversion of B and -1 is taken from mathematical deduction and through applying different properties. Assuming that both A and B are in 2's complement, an INV(B) will simply give us an inversion of each bit in B giving us 1's complement. Moreover, the addition of 1 to INV(B) will give us the 2's complement of B which will allow us to reuse the same circuit for either operation.

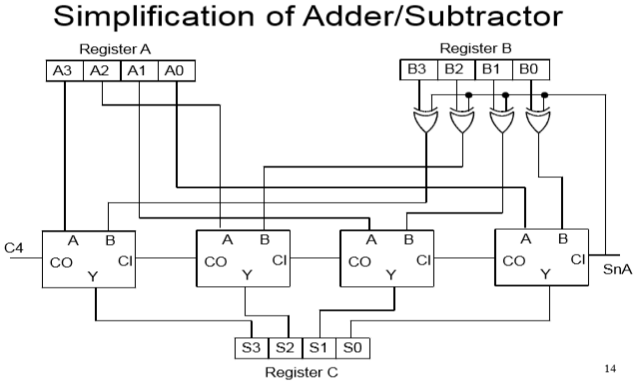

Fig 15. Logical Circuit Design of Adder/Subtractor [2]

Fig 12. shows the importance of changing $a2(Fig. 7/ Fig.8) in order to make necessary changes so that the circuit can work efficiently for either addition or subtraction. If $a2 is 0x00000000, then the carry in bit will be 0 and B will remain the same. However, if $a2 is 0xFFFFFFFF, a XOR operation will take place and invert the bit pattern for B, and the carry in bit will be 1 which will take care of INV(B) + 1.

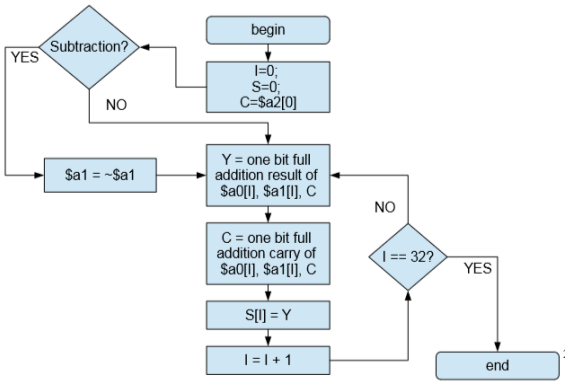

Fig 16. Flowchart for add_sub_logical [1]

The flowchart above depicts an effective procedure for carrying out the necessary actions in add_sub_logical. Prior to entering a loop that will run for 32 times, based on the mathematical operation, the second argument ($a1) will stay the same for addition or be inverted for subtraction, and the carry in bit will remain 0 for addition or be 1 for subtraction. Each time the loop is entered, each bit from each argument is extracted, added, then inserted to the register will holds the result.

Fig 14. shows useful macros that help extract each bit from the arguments and later insert a bit into the final answer.

```
add_sub_logical:
        addi    $sp, $sp, -32
        sw      $fp, 32($sp)
        sw      $ra, 28($sp)
        sw      $a0, 24($sp)
        sw      $a1, 20($sp)
        sw      $a2, 16($sp)
        sw      $s0, 12($sp)
        sw      $s1, 8($sp)
        addi    $fp, $sp, 32

        addi $t0, $zero, 0      # index
        addi $s1, $zero, 0      #setting our final answer to 0b0000
        addi $t1, $zero, 0      #If addition the carry in will be 0
        beq $a2, 0x00000000, loop #Checking if it is addition, if n
        not $a1, $a1    #Negating the bit pattern
        addi $t1, $zero, 1      #Setting the CI to 1
```

Fig 17. Implementation of add_sub_logical

Depending on the operation of $a2, the bit pattern of $a1 will be inverted and the carry in represented in $t1 will be set to 1 and fall through to the loop. Otherwise, $a1 will remain the same and the carry in of $t1 will be set to 0 and jump to the loop.

```
loop:
                extract_nth_bit($t2, $a0, $t0)  #Extracting first bit
                extract_nth_bit($t3, $a1, $t0)  #Extracting first bit
                xor $t4, $t2, $t3       #A XOR B
                xor $s0, $t1, $t4       #(A XOR B)XOR CI

                insert_nth_bit($s1, $t0, $s0, $t7)      #Inserting sum
                and $t6, $t4, $t1       #Doing calculations to find ne
                and $t5, $t2, $t3       #Doing calculations to find ne
                or $t1, $t5, $t6        #CI =  CI.(A XOR B) + A.B
                addi $t0, $t0, 1        #Incrementing index of loop
                blt $t0, 32, loop       #Until $t0 does not equal 32,
                                        #If it does, then it w
```

Fig 18. Implementation of add/sub loop

Once each bit is added it is then inserted into $s1 which holds our final bit pattern for the result of the corresponding mathematical operation. It is later moved to $v0 as Fig 16. shows.

```
RESTORE:
        move $v0, $s1 #Move the final answer to $v0
        move $v1, $t1 #Move carry in bit to $v1 *** Will be necessary
        lw      $fp, 32($sp)
        lw      $ra, 28($sp)
        lw      $a0, 24($sp)
        lw      $a1, 20($sp)
        lw      $a2, 16($sp)
        lw      $s0, 12($sp)
        lw      $s1, 8($sp)
        addi    $sp, $sp, 32

        jr $ra
```

Fig 19. Implementation of add/sub loop

As for $v1, it holds the carry which will later be useful for calculating the final answer for multiplication if the answer is negative.

B. *Utility functions for Multiplication*

1) *twos_complement*

This utility function takes in an argument of $a0, and converts the number into its 2's complement form by inverting the bit pattern, then calling add_logical to add 1 to that bit pattern. This becomes useful when trying to convert a number to its unsigned form when the argument is originally negative.

```
twos_complement:
        addi $sp, $sp, -24
        sw $fp, 24($sp)
        sw $ra, 20($sp)
        sw $a0, 16($sp)
        sw $a1, 12($sp)
        sw $a2, 8($sp)
        addi $fp, $sp, 24

        not $a0, $a0    #Do an inversion of bit pattern
        addi $a1, $zero, 1      #Add one to the inversion by calling add_logical
        jal add_logical # $v0 will hold result

        lw $fp, 24($sp)
        lw $ra, 20($sp)
        lw $a0, 16($sp)
        lw $a1, 12($sp)
        lw $a2, 8($sp)
        addi $sp, $sp, 24

        jr $ra
```

Fig 20. twos_complement

2) *twos_complement_if_neg*

This utility function takes in an argument of $a0, and checks to see if first bit is a 1 or 0. If the first bit is a 1, it will call twos_complement. Else, nothing will change and the pointer will return back to the caller.

```
twos_complement_if_neg:
        blt $a0, 0, twos_complement     # If $a0 is negative, jump
        move $v0, $a0           #Else, since twos_complement return
        jr $ra
```

Fig 21. twos_complement_if_neg

3) *twos_complement_64bit*

This utility function takes in an argument of $a0 and $a1. After preforming unsigned multiplication, the first bits of the original arguments may be positive and negative, or negative and positive which mean a negative product. Thus, this means that the LO and HI portion of the product must be converted into 64 bit 2's complement to determine the final product if negative.

```
twos_complement_64bit:
        addi    $sp, $sp, -40
        sw      $fp, 40($sp)
        sw      $ra, 36($sp)
        sw      $a0, 32($sp)
        sw      $a1, 28($sp)
        sw      $a2, 24($sp)
        sw      $s0, 20($sp)
        sw      $s1, 16($sp)
        sw      $s2, 12($sp)
        sw      $s3, 8($sp)
        addi    $fp, $sp, 40

        not $a0, $a0    #Invert both arguments
        not $a1, $a1
        move $s1, $a1 #Have to save at least one of the registers for the second call for add_logic
        li $a1, 1
        jal add_logical #Adding 1($a1) to $a0
        move $s2, $v0   #Moving result in $s2
        move $a0,$s1
        move $a1, $v1   #TAKING THE CARRY FROM PREVIOUS CALL AND STORING IT AS FIRST OPERAND IN upc
        jal add_logical
        move $s3, $v0
        move $v0, $s2 #New LO(32-bit)
        move $v1, $s3 #New HI(32-bit)
```

Fig 22. twos_complement_64bit

4) bit_replicator

This utility function takes in an argument of $a0 and does a 32-bit sign extension of the bit of $a0; this will be useful and make multiplication easier to implement.

```
bit_replicator:
        addi $sp, $sp, -16
        sw $fp, 16 ($sp)
        sw $ra, 12($sp)
        sw $a0, 8($sp)
        addi $fp, $sp, 16

        beq $a0, 0x1, bit_replicator_negative   #If extracted bit from $a0 is 1, it is negative
        lui $a0, 0x0000 #Else, $a0 is positive and load 0x00000000 into $a0
        ori $a0, 0x0000
        move $v0, $a0   #returning $v0, so move $a0's contents to $v0
        j restore_bit_replicator

bit_replicator_negative:
        lui $a0, 0xFFFF #Loading bit pattern of $a0 with all ones which is 0xFFFFFFFF
        ori $a0, 0xFFFF
        move $v0, $a0   #returning $v0, so move $a0's contents to $v0

restore_bit_replicator:
        lw $fp, 16 ($sp)
        lw $ra, 12($sp)
        lw $a0, 8($sp)
        addi $sp, $sp, 16
        jr $ra
```

Fig 23. bit_replicator

C. *Multiplication*

1) mult_signed

This function takes in the argument $a0 and $a1. If any of the operands are negative, the twos_complement utility function will be called so that calling mult_unsigned will not be a problem.

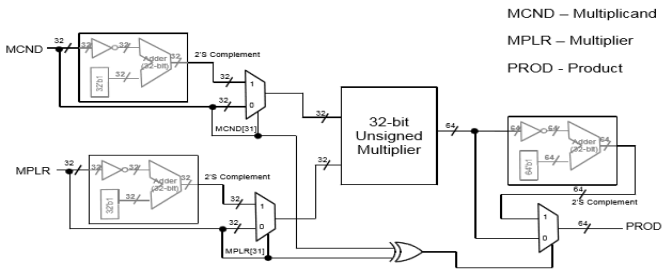

Fig 24. Signed Multiplication Circuit Drawing

Upon entering this circuit, it is important to note that either of the operands can be positive or negative, however, they are converted to their 2's complement form so that 32-bit unsigned multiplication can be done. Only after this, then is the sign of the final product determined through twos_complement_64bit.

2) mult_unsigned

This function takes in two arguments, $a0 and $a1, and preform multiplication through a loop that corresponds to the flowchart below. It is important to note that although both of the operands are 32-bit integers, the product must be 64 bit. This is the reason why there are LO and HI registers, where the low holds the product and the high holds any overflow.

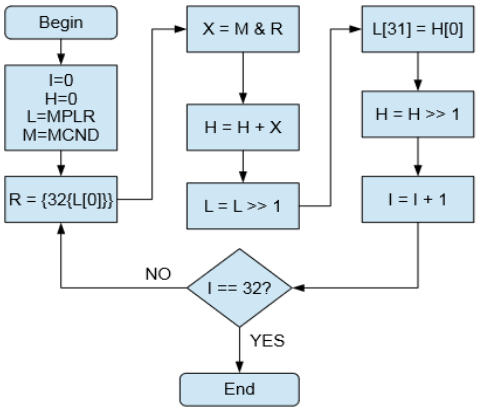

Fig 25. Signed Multiplication Circuit Drawing [2]

This shift function that occurs in Fig 22. is to capture the two bit product. The multiplier is first shifted to the right by one which is the same as dividing by a power of 2^1. Then the most significant bit of H[0] will be placed as the most significant bit in the LO register. Finally, the high register will be shifted to the left by one and the counter for the loop will be incremented. This process will continue until the loop counter equals 32.

After the product is determined, we then have to go back and observe the original operands before twos_complement is called (if it was necessary) and extract the MSB of each of the operands to determine the sign of the final product. It turns out, a XOR operation mimics the sign output of any multiplication input. In the truth table below, 0 represents a positive number, and 1 represents a negative number.

INPUTS		OUTPUTS
A	B	Y = A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

Fig 26. Signed Multiplication Circuit Drawing

3) twos_complement in mult_signed

Upon entering mult_signed, before entering mult_unsigned both arguments have to be checked for negativity. First twos_complement_if neg will be called, then if they are negative then the arguments will be modified in the twos_complement function call.

```
move $s0,$a0    # Saved argument $a0 in $s0 for using twos_complement_64bit later if needed
move $s1,$a1    # Saved argument $a1 in $s1 for using twos_complement_64bit later if needed
jal     twos_complement_if_neg  #Invert bit pattern and add one if $a0 is negative
move $s2, $v0   #Store result in $s2

move $a0, $s1                   #$s1 is really $a1, but we are loading $s0 in $a0
jal twos_complement_if_neg      #Invert bit pattern and add one if $a0 is negative, store in
move $s3, $v0   #Store result in $s3

move $a0,$s2    #Loading in new arguments in to do mult_unsigned
move $a1,$s3
jal     mult_unsigned
move $s4, $v0   #Save LO result into $s4
move $s5, $v1   #Save HI result into $s5
```

Fig 27. Checking for Negativity [2]

4) mult_unsigned implementation

The implementation for the mult_unsigned and its loop in Fig. 25 are implemented accordingly to the flowchart in Fig.22.

```
        addi $s0, $zero, 0 #I
        move $s1, $a1 #MPLR L
        addi $s2, $zero, 0x00000000 #H
        move $s3, $a0 #MCND M

mult_loop:
        extract_nth_bit($t3,$s1, $zero) #Extract first bit of LO
        move $a0, $t3           #Preparing to call bit_replicator by moving $t3 to $a0
        jal bit_replicator
        move $t3, $v0           #Move the result back to $t3(R)

        and $t4, $t3, $s3       # $t4 = M & R *** $t4 = X
        move $a1, $t4           #Preparing to do H = H + X ** $a1 = X
        move $a0, $s2           #$a0 = H
        jal add_logical
        move $s2, $v0           #s2 = H + X
        srl $s1,$s1,1    # L>>1

        extract_nth_bit($t5, $s2, $zero) # Extracting H[0], putting it into $t5
        addi $t6, $zero, 31
        insert_to_nth_bit($s1, $t6, $t5, $t7) #Inserting $t5 to L[31]

        srl $s2, $s2, 1 # H>>1
        addi $s0, $s0, 1 # I++

        blt $s0, 32, mult_loop
```

Fig 28. Implementation for mult_unsigned and mult_loop

5) mult_signed_restoration.

Once the product has been determined and the frame has of mult_unsigned has been restored, the caller will return back to mult_unsigned, and now determine the sign of the final product.

```
        jal     mult_unsigned
        move $s4, $v0   #Save LO result into $s4
        move $s5, $v1   #Save HI result into $s5

        li $t9, 31
        extract_nth_bit($t0,$s0,$t9) #Extract MSB bit from $a0
        extract_nth_bit($t1,$s1,$t9) #Extract MSB bit from $a1
        xor $s6, $t0, $t1
        beq $s6, 0, RESTORE_MULT_SIGNED #IF the XOR == 0, then go restore, as answer is positive
        move $a0,$s4    #ELSE, do twos_complement_64bit
        move $a1,$s5
        jal     twos_complement_64bit
        j       RESTORE_MULT_SIGNED

RESTORE_MULT_SIGNED:
        lw $fp, 48($sp)         #it does not mess up on frame creation
        lw $ra, 44($sp)
        lw $a0, 40($sp)
        lw $a1, 36($sp)
        lw $s0, 32($sp)
        lw $s1, 28($sp)
        lw $s2, 24($sp)
        lw $s3, 20($sp)
        lw $s4, 16($sp)
        lw $s5, 12($sp)
        lw $s6, 8($sp)
        addi $sp, $sp, 48
        jr $ra
```

Fig 29. Adjusting Product to Necessary Sign

To determine the sign of the product, the 31st bit of the original bit pattern of the arguments of $a0 and $a1 are determined; then, a XOR operation is done between those two bits. If the XOR results in a 1 as Fig 26. shows, then the product has to undergo the twos_complement_64-bit call. Else, it will restore the frame for mult_signed.

D. div_signed

Similar to multiplication, this function call is split into two parts: div_signed and div_unsigned.

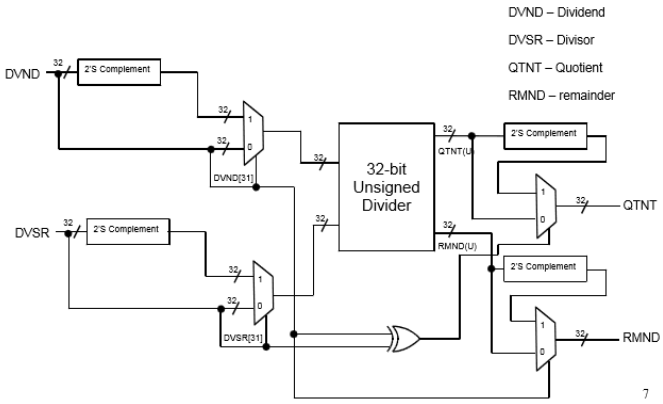

Fig 30. Circuit Design for div_signed [3]

It is also important to note that division is a 32-bit operation because between two unsigned integers that are natural numbers, the highest number of bits that can be in the product is 32-bits. In this case $v0 is a 32 bit register that holds the quotient, while 32-bits is contained in the remainder. However, unlike multiplication, if the result (quotient in this case) is negative, it has nothing to do with the result of the remainder. Instead, the remainder's sign depends on the MSB bit of $a0 (the first argument). If $a0[31] is a 1, then the remainder will undergo two's complement as well. This is different from multiplication in the sense that the XOR does not determine the result for both the LO and HI registers; instead, the Quotient(LO) and the Remainder(HI), must undergo two separate checks.

To summarize, the quotient's sign depends on a XOR between the first bit of each of the two arguments, while the remainder has to undergo a separate check which relies on the sign of $a0[31].

Prior to entering div_unsigned, each of the arguments must be checked for negativity and undergo the twos_complement call if they are negative, similar to multiplication.

```
div_signed:
        addi    $sp, $sp, -56
        sw      $fp, 56($sp)
        sw      $ra, 52($sp)
        sw      $a0, 48($sp)
        sw      $a1, 44($sp)
        sw      $a2, 40($sp)
        sw      $s0, 36($sp)
        sw      $s1, 32($sp)
        sw      $s2, 28($sp)
        sw      $s3, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        addi    $fp, $sp, 56

        move $s0,$a0    # Saved argument $a0 in $s0 for using/testing twos_complement_quotient and twos_complement_remainder later
        move $s1,$a1    # Saved argument $a1 in $s1 for using/testing twos_complement_quotient later
        jal     twos_complement_if_neg  #Invert bit pattern and add one if $a0 is negative
        move $s2, $v0   #Store result in $s2

        move $a0, $s1                   #$s1 is really $a1, but we are loading $s0 in $a0
        jal twos_complement_if_neg      #Invert bit pattern and add one if $a0 is negative, store in
        move $s3, $v0   #Store result in $s3

        move $a0,$s2    #Loading in new arguments in to do div_unsigned
        move $a1,$s3
        jal     div_unsigned
        move $s4, $v0   #Save LO result into $s4
        move $s5, $v1   #Save HI result into $s5
```

Fig 31. Checking for Negativity

After, the arguments have been turned into their unsigned representation, there will be a jump and link call to div_unsigned, where the loop is located. The steps taken in the loop correspond to this flowchart which shows how division works.

```
div_loop:

        sll $s1, $s1, 1 #Shifting R to left by 1
        addi $t0, $t0, 31
        extract_nth_bit($s4, $s2, $t0) #Extracting Q[31] and putting it into $s4
        insert_to_nth_bit($s1, $zero, $s4, $t9) #Inserting bit of $sit into $s1[0] which is R[0]
        sll $s2, $s2, 1 #Shifting Q to left by 1
        move $a0, $s1 #Moving R into first argument
        move $a1, $s3 #Moving D into first argument
        jal sub_logical
        move $s5, $v0 #S = R - D

        bgt $s5, -1, div_greater
        addi $s0, $s0, 1
        blt $s0, 32, div_loop

        j RESTORE_DIV_UNSIGNED

div_greater:
        move $s1, $s5  #R = S
        addi $t1, $zero, 1
        insert_to_nth_bit($s2, $zero, $t1, $t9)
        addi $s0, $s0, 1
        blt $s0, 32, div_loop
```

Fig 32. div_unsigned loop

Once the loop has completed, the call will jump back to div_signed and determine the sign of the quotient and remainder.

```
        move $a0,$s2    #Loading in new arguments in to do div_unsigned
        move $a1,$s3
        jal     div_unsigned
        move $s4, $v0   #Save LO result into $s4
        move $s5, $v1   #Save HI result into $s5

        addi $t9, $zero, 31
        extract_nth_bit($t0, $s0, $t9)
        extract_nth_bit($t1, $s1, $t9)
        xor $s6, $t0, $t1

        move $s7, $t0 #USED LATER TO CHANGE REMAINDER TO NEGATIVE if $s7 is
        beq $s6, 1, twos_complement_quotient
        j twos_complement_remainder

        j RESTORE_DIV_SIGNED

twos_complement_quotient:
        move $a0, $s4
        jal twos_complement
        move $s4, $v0

twos_complement_remainder:
        bne $s7, 1, RESTORE_DIV_SIGNED
        move $a0, $s5
        jal twos_complement
        move $s5, $v0

RESTORE_DIV_SIGNED:
        move $v0, $s4
        move $v1, $s5
```

Fig 33. Determining Signs of Quotient and Remainder

The result from the XOR in Fig 33. will determine if a sign change of the quotient is necessary to its negative form. Once that is completed, a separate check is taken place to determine the sign for the remainder. If the first bit of the $a0, the divisor, is 1. This implies that the remainder must be negative. For that reason, the remainder must undergo a twos_complement call.

V. Testing

After all macros, functions have been implemented accordingly, a test shall be done to determine if the work completed is correct. In order to run the program, each and every file must be saved and assembled. If a highlighted yellow bar pops up, make sure to have implemented everything without any syntax errors. After this has been done, click the green arrow with a tooltip text of "Run the current program." The program should produce a total of 40 results with each operation having a result produced from au_normal and another from au_logical. Each of the results should align and ultimately product a score of 40/40.

```
(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 34. Program Result

VI. Conclusion

I have learned a lot from completing this project. Arithmetic operations are more than just literally asking your computer or application to do division. It is an extremely complicated procedure that uses Boolean operations to complete its task. While many people may be oblivious to this, it is important to realize that these operations have a lot more complexity than we may think. Not only was this a valuable experience, but it made me realize that sometimes the most "simplest" tasks we give a computer are not so simple after all.

[1] K.Patra. CS 47. Class Lecture, Topic : “Addition Subtraction Logic.” San Jose State University, San Jose, CA, November 14, 2018.

[2] K.Patra. CS 47. Class Lecture, Topic : “Addition Subtraction Logic.” San Jose State University, San Jose, CA, November 14, 2018.

[3] K.Patra. CS 47. Class Lecture, Topic : “Addition Subtraction Logic.” San Jose State University, San Jose, CA, November 14, 2018.