MIPS Implementation of ALU with Logic Operations

Aditya Nair
San Jose State University
aditya.nair@sjsu.edu

Abstract- This report is a detailed explanation which covers the binary implementation of an ALU using logical operations. This paper explains the high-level conceptual idea of an ALU's logical operations and also dives into the code to show how this can be implemented. The operations which will be covered are addition, subtraction, multiplication, and division.

I. INTRODUCTION

In this project, we will use MARS MIPS simulator tobuild out own ALU. We will use basic logical operation AND, OR, XOR, and NOT to create our own implementation of addition, subtraction, multiplication, and division. To do this, we first have to fully understand how we can implement these operations with logical operations. Once we understand the strategy, we can use the MARS MIPS simulator to implement it, and make sure it works on several test cases.

II. REQUIREMENTS AND SETUP

A. Software Introduction

We will implement this ALU with MARS, a MIPS assembly language programing IDE which simulates a MIPS assembler and its runtime. This simulator is made by Missouri State University. In this simulation, we can run any MIPS functions. It should be noted that MARS does not actually run this on your computer CPU. It does not use actual registers on your computer for storage, and it does not actually use the assembler in your computer. Instead, because it a simulator, it only simulates an assembly language environment on your computer.

B. Software Installation

To install MARS, navigate on any browser to http://courses.missouristate.edu/kenvollmar/mars/ and find the link "Download MARS 4.5 software!" From there, follow any instruction to install the application appropriately on your operating system.

C. MARS Settings

There are some very important and useful settings to use with MARS for our ALU implementation project. See Figure 1 for exact recommended settings. But crucially, we want to set the checkmark to "Assemble all files in directory" and "Initialize Program Counter to global 'main' if defined"

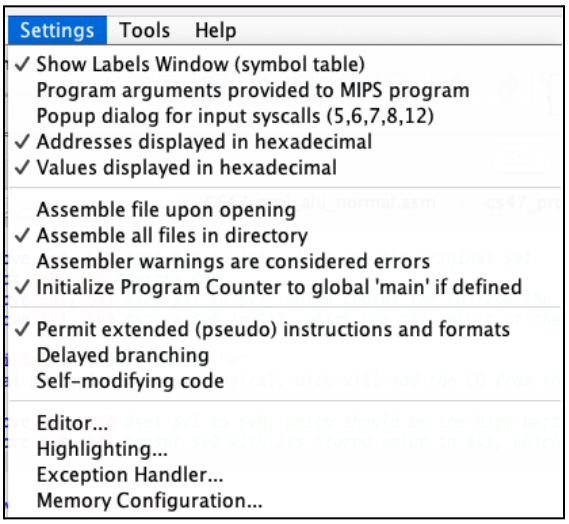

Fig. 1. MARS Recommended Settings

D. Importing Files

We need to imporort all the starter files into our project so we can get started. There are 5 files that we need to import. We can download all these files at the following link:
https://sjsu.instructure.com/courses/1474044/files/66532978/download?wrap=1
Once we download the files, there should be 5 files in a .zip format. We can unzip the files to reveal the following files.
The files should be:

1. CS47_proj_alu_logical.asm
2. CS47_proj_alu_normal.asm
3. cs47_proj_macro.asm
4. cs47_proj_procs.asm

5. proj-auto-test.asm

Here is the general explanation for each file respectivly:

1. Here is where we code our logical implementation of our ALU
2. Here is where we code a normal implementation of our ALU, built with commands such as add, sub, mult, and div. This file will be used for later testing.
3. This file is used for coding any useful macros we might use. We will come back to this later.
4. This file is not to be touched; it contains code for printing the results of tests run on our ALU
5. This file contains several test cases, meant to run and test our logical ALU. To check for accuracy, it will compare results from CS47_proj_alu_logical.asm to CS47_proj_alu_normal.asm.

E. Opening Files in MARS

Once the files are downloaded, we can open them in the MARS IDE. Once MARS is opened, click on File→Open and select each of the 5 files one by one. The files should open up in MARS, and will be connected to the original files so we can later save the files with File→Save or your Operating System equivalent.

F. Basic MARS Tutorial

MARS is a MIPS simulator environment, so we can use any MIPS command. Therefore, it might be very useful to keep a MIPS cheat sheet with you as you code to reference so you don't have to remember what specific commands do and which regesters are needed. Once the code is finished, we can assemble by clicking the screwdriver/wrench button in fig. 2:

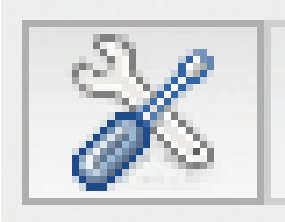

Fig. 2. MARS Assemble Button

If the settings have been properly set in accordance to step II - D of this report, assembling one file should save an assemble all files involved. Once our program is assembled, we can run it by clicking the start button in fig. 3:

Fig. 3. MARS Run Button

III. BOOLEAN LOGIC

A. Intro to Boolean Logic

Unlike humans, computers use a base-2 number system, in which all number or data is represented as what is knows as bits. A bit can either be a 0, representing false, or 1, representing true. For the computer to run virtually any operation, it needs to be able to run basic arithmetic operations with these bits. These operation are “OR”, “AND,”, “XOR”, and “NOT”. We can use these simple and logical operations and manipulate them to do an incredible amount.

This concept is known as Boolean Logic or Boolean Algebra, and is needed to understand how we are going to implement out ALU with MIPS. Boolean Logic has 4 main functions, also known as logic gates. Each of these function takes 2 inputs, and returns one output:

1) AND
2) OR
3) XOR
4) NOT

Through basic electronics, each of these functions can be easily similar with wires and gates, hence the term logic gates.

To understand how each of these work, we can look at the truth table for each. For demonstration purposes, A will represent input 1, and B will represent input 2.

OR: Returns true if either A is true, OR if B is true. Either one of these can be true for the output to return true. Even if both inputs are true, the output is true. The only time OR will return false is when both inputs start as false. OR is represented with a plus sign (+).

TABLE I

Logical OR Truth Table

Input 1	Input 2	Output
A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

Here is the symbol used to represent the OR gate when drawing circuits:

Fig. 4. OR Gate Representation

AND: Returns true only if A is true AND B is true. If any of the two, or both A and B, are false, the output is also false. AND is represented with a period (.).

TABLE II
Logical AND Truth Table

Input 1	Input 2	Output
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

Here is the symbol used to represent the AND gate when drawing circuits:

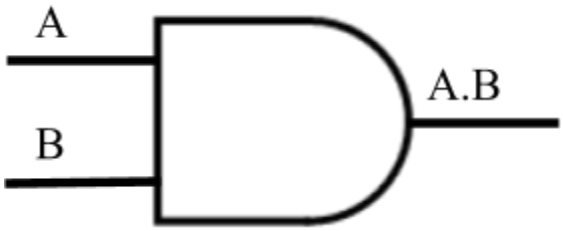

Fig. 5. OR Gate Representation

XOR: XOR is also known as exclusive or. It returns true if either A is true, or B is true, but not both. Either one of these can be true for the output to return true. The difference between OR and XOR can be seen in when both inputs are 1. In this case, OR would return true, but XOR returns false. XOR can be implemented as a combination of the other 3 gates. XOR is represented with a plus sign in a circle (⊕).

TABLE III
Logical XOR Truth Table

Input 1	Input 2	Output
A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

Here is the symbol used to represent the XOR gate when drawing circuits:

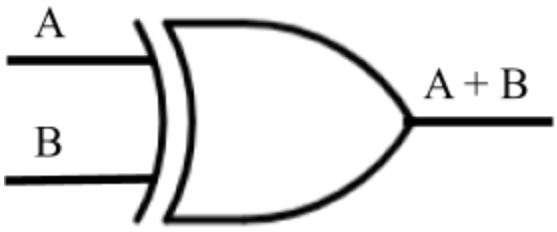

Fig. 6. XOR Gate Representation

NOT: The NOT gate simply negates whatever value is given to it. Instead of taking 2 inputs, NOT only takes one input. Without any exceptions, NOT simply

negates the input and returns that value as the output. NOT is represented with an apostrophe sign (‘).

TABLE IV
Logical NOT Truth Table

Input 1	Output
A	A’
0	1
1	0

Here is the symbol used to represent the NOT gate when drawing circuits:

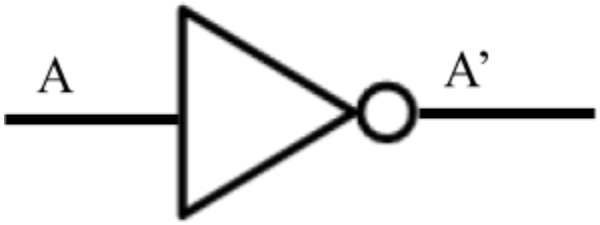

Fig. 7. NOT Gate Representation

IV. BINARY OPERATIONS IMPLEMENTATION STRATEGY

A. Intro to Binary Operations

Before we dive into MARS and implement all 4 arithmetic procedures, it is important that we thoroughly understand the logic and strategy behind the binary implementation of arithmetic operations. For demonstration purposes, A will be input 1, B will be input 2, and C will be the output for all operations.

B. Binary Addition

Binary addition is implemented in a similar manner as normal pen-and-paper addition. Starting from the right, we will add each digit to result in a resulting digit. If that resulting value is higher than the base number system, than the second digit will be considered the “carry over” and will be carried over to the addition of the next two digits. This process continues as every respective digit is added from both inputs.

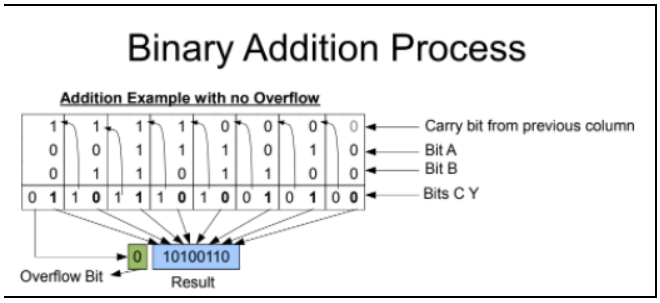

Fig. 8. Binary Addition Representation

C. Binary Subtraction

Binary subtraction is very similar to binary addition. In fact, binary subtraction is just binary addition with a few extra steps. Instead of thinking of it as subtraction, we will think of it as adding a positive number with a negative number. To do this, we will negate the number being subtracted. Since our operation is A - B, we will negate B. This will essentially be like adding A + (-B) which results in A - B. In addition, we need to make sure the original carry in bit is 1 as opposed to the 0 we use for binary addition.

D. Binary Multiplication

Binary multiplication is implemented similar to normal pen-and-paper multiplication. As seen in Fig. 6, we have a multiplicand (a) and a multiplier (b). For each digit in the multiplier, starting from the right, we will multiply the digit at that multiplier with the entire multiplicand. However, with binary multiplication, since each digit is either 1 or 0 we do not need to worry about multiplying each digit witht eh multiplicand. Instead, if the digit of the multiplier is a 0, our result for that line will be 0, and if it is a 1, our result will be the entire multiplicand. As we move on the the next digit of the multiplier, we do the same procedure, except we shift the result to the left by one digit. For each next digit in the multiplier, we shift the result one more digit to the left. Once we have gone through every digit of the multiplier, we can simply add all the results using binary addition to get the final product.

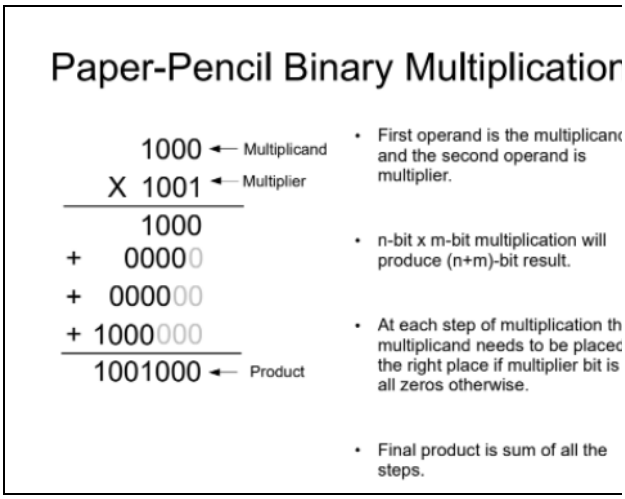

Fig. 9. Binary Multiplication Representation

E. Binary Division

Binary division, our final arithmetic operation, is again implemented similar to normal pen-and-paper division. To do this division, as we can see in Fig. 10, we keep dividing the dividend by the divisor to generate a quotient as well as a remainder. The quotient of that division will be our final quotient, and we will again divide the remainder by the divisor and add the result to the quotient, shifted one digit to the right each time. We will continue this process until the remained can no longer be divided by the divisor. At that point, the quotient is the final quotient for our division.

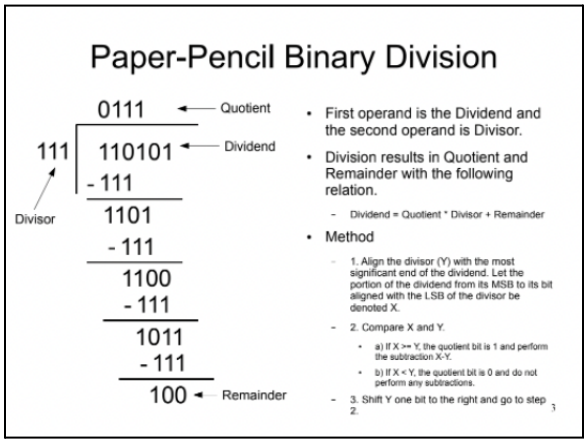

Fig. 10. Binary Division Representation

V. LOGICAL MIPS ALU IMPLIMENTATION

A. Intro to Logical Mips Implimentation

Before we start coding each logical operation, lets take a look at some general conversion, and which registers we will use.

1) $a0 - First Input
2) $a1 - Second Input
3) $a2 - Operation Code
    a) Addition: '+'
    b) Subtraction: '-'
    c) Division: '/'
    d) Multiplication: '*'
4) $v0 - First Output
    a) Addition: Sum
    b) Subtracting: Difference
    c) Multiplication: Product (Lo)
    d) Division: Quotient
5) $v1 - Second Output
    a) Multiplication: Product (Hi)
    b) Division: Remainder

B. Macros

In dealing with almost all of our operations, there are a few macros that can greatly recude the amount of code needed due to repetitivness. These two macros are extract_nth_bit and insert_to_nth_bit.

Extract_nth_bit will do exactly that, given an "n", this macro will return the bit at position n of a given bit string. This macro will take 3 inputs, $regD, $regS, $regT

1) $regD will contain the result, or output. In other words, it will contain the digit of the given bit string at position n.
2) $regS will be our bit string or bit pattern.
3) $regT will be n, the bit position of our bit string that we are extracting our bit from.

Overall, we are extracting the bit $regD from $regS at position $regT. We will do this by loading $regD with 1, and then temporarily shifting regD to the left until the position of 1 in $regD is the same as the position of the bit that we need in $regS. Then, using AND on $regD and $regS, we can set the value at the 1 of $regD to the proper value of $regS, before undoing the first shift to the left to return our final value.

```
.macro extract_nth_bit($regD, $regS, $regT)

        li $regD, 0x1
        sllv $regD, $regD, $regT
        and $regD, $regS, $regD
        srlv $regD, $regD, $regT

.end_macro
```

Fig. 11. Extract_nth_bit Implementation

The other macro, insert_to_nth_bit, will again do exactly that. This macro will insert a digit to a bit string at a position n. This macro will take 4 inputs, $regD, $regS ,$regT, and $maskReg.

1) $regD will be the bit string which we want to insert a bit into.
2) $regS will be n, the position of our bit string $regD what we want to insert our digit into.
3) $regT will be the digit that we want to insert, being either a 0x0 or 0x1.
4) $maskReg will be a temporary register used store a mask.

Overall, we are inserting $regT into $regD at position $regT. We will do this in a similar approach as extract_nth_bit. We will shift $maskReg to the left until its at the value that we want to insert, and negate it, using AND to update the value of $regD to the new value. At this point, the value at the position N will be a 1. Now, we can temporarily shift $reegT to the left before using OR to add $regT to $regD before shifting is back. Shifting will allow us to insert $regT at the proper location instead of at the end.

```
.macro insert_to_nth_bit($regD, $regS, $regT, $maskReg)

        li $maskReg, 0x1
        sllv $maskReg, $maskReg, $regS
        not $maskReg, $maskReg
        and $regD, $regD, $maskReg
        sllv $regT, $regT, $regS
        or $regD, $regT, $regD
        srlv $regT,$regT,$regS

.end_macro
```

Fig. 12. Insert_to_nth_bit Implementation

A. Au_logical and end

Au_logical is the main function of our code. This is where any operation call will start from. It is here that we will branch to the appropriate operation. In au_logical, we can start by saving the run time envoronment. Then, we can do simple branching using beq to branch to the 4 following functions:

1) log_add
2) log_sub
3) log_div
4) log_mul

```
au_logical:

        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, '+', log_add
        beq $a2, '-', log_sub
        beq $a2, '/', log_div
        beq $a2, '*', log_mul
```

Fig. 13. Au_logical Implementation

It will also be useful to add an “end” function to take care of the run time environment, and so any function can skip straight to the end.

```
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. 14. End Implementation

B. log_add and log_sub

We are now ready to implement the logical add and subtract functions.

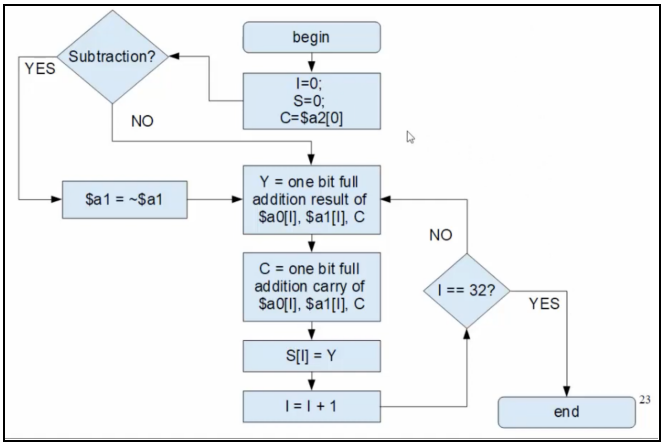

Fig. 15. log_add and log_sub strategy

The strategy for implementing logical add and subtract can be seen in figure 11.We will start by resetting all the temp registers we will use as well as the output $v0. We will also designate registers to act as a counter, carry in / carry out, and the sum. Next, if we are dealing with subtraction, we will negate $a1, which was the second input, and we will set our first carry bit to 1 instead of 0. Once the exceptions for subtraction have been dealt with, we will precede with normal addition. The code for this can be seen below in Fig. 16

```
log_add:


li $a2, 0x0
j add_or_sub_reset

log_sub:
li $a2, 0xFFFFFFFF

add_or_sub_reset:
li $v0, 0
li $t0, 0
li $t4, 0
extract_nth_bit($t3,$a2,$zero)
beqz $t3, add_helper

sub_helper:
not $a1, $a1
```

Fig. 16. Setup for Logical Add/Sub

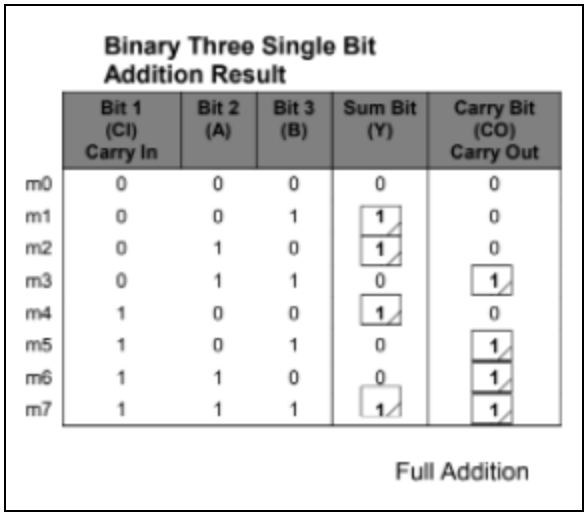

Fig. 17. Full Adder Truth Table

From the truth table in Fig. 17, we can visualize how we should implement addition. We can see how two bits are added, resulting in a sum bit and a carry out bit. This carry out bit is then the carry in bit for the following line. The next line adds both bits and the carry bit. This process continues for every bit in a and b.

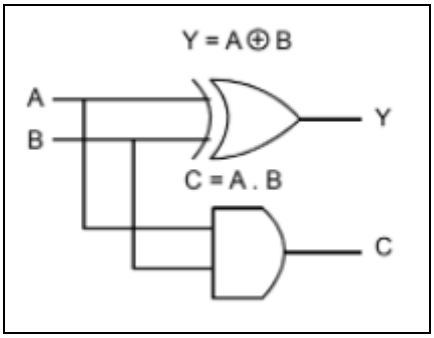

Fig. 18. Half Adder Logical Visualization

In this half adder in Fig. 18, we can visualize the truth table from Fig 11. In a logical manner. We can see that the carry bit is just a result from an AND operation of input bits A and B. Additionaly, the sum bit Y is just the result from an XOR operation of input bits A and B.

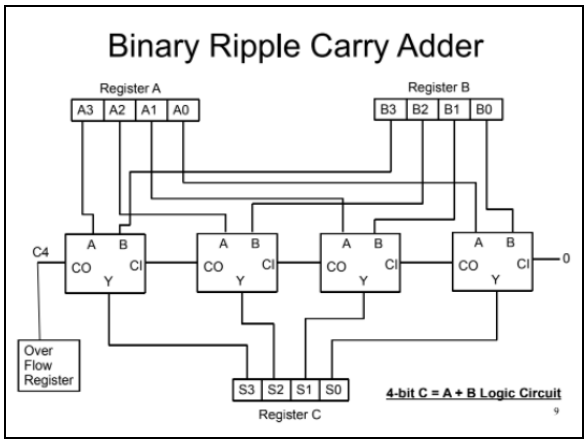

Fig. 19. Binary Ripple Carry Adder

When everything is put together, we get the full Binary Ripple Carry Adder. We can see a combination of multiple "half adders." We can see how the carry out bit of one half adder becomes the carry in bit for the next half adder. Our goal is to, using a loop, simulate these half adders using MIPS with the same logic structure.

B. log_add and log_sub implementation

In Fig. 20, we can see a MIPS implementation of the Carry Adder. In a loop where a counter goes from 0 to 32, we will first set out addends a and b the corresponding value based ont eh counter. We will then add both a and b and the carry-in using XOR, and then we will set the carry-out bit using AND.

```
add_helper:
extract_nth_bit($t1,$a0,$t0)
extract_nth_bit($t2,$a1,$t0)

xor $t5, $t1, $t2
and $t6, $t1, $t2

and $t7, $t3, $t5
xor $t4, $t5, $t3
or $t3, $t7, $t6

insert_to_nth_bit($v0,$t0,$t4,$t5)

add $t0,$t0,1
blt $t0,32,add_helper
move $v1,$t3


add_helper_end:
j end
```

Fig. 20. log_add Implementation

C. Multiplication functions

There are a few necessary functions we need to implement before multiplication. The first is the bit replicated. This will replicate either F's or 0's depending on the value of $a0. The code for this can be seen in Fig. 21

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

        beqz $a0,repeating_zero

        li $v0, 0xFFFFFFFF #Load
        j skip_to_end_replicator

        repeating_zero:
        li $v0, 0x00000000 #Load


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

Fig. 21. Bit Replicator Implementation

Next, we need to implement two's complement. We will need 3 different functions. The first one is two's compliment, which will convert a given bit string into two's compliments. See Fig. 22 for the code for twos_complement.

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

        not $a0,$a0 #negate $a
        li $a1, 1 #set $a1 to
        li $a2, '+' #set $a2 t
        jal au_logical #call a

        lw      $fp, 20($sp)
        lw      $ra, 16($sp)
        lw      $a0, 12($sp)
        lw      $a1, 8($sp)
        addi    $sp, $sp, 20
        jr $ra
```

Fig. 22. Two's Compliment Implementation

We also need a function that will run two's compliment after taking care of any negative numbers that may exist.

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

        bltz $a0, numIsNeg #ski
        move $v0,$a0 #since $a0
        j end_twos_comp #skip t

        numIsNeg:
        jal twos_complement #ru

        end_twos_comp:
        lw      $fp, 20($sp)
        lw      $ra, 16($sp)
        lw      $a0, 12($sp)
        lw      $a1, 8($sp)
        addi    $sp, $sp, 20
        jr $ra
```

Fig. 23. Two's Compliment Implementation

Lastly, we need a function that will run two's compliment on number with a bit of 64.

```
twos_complement_64bit:
        #$t0 is $a0, $t1 is a1
        addi    $sp, $sp, -28
        sw      $fp, 28($sp)
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $a2, 12($sp)
        sw      $s1, 8($sp)
        addi $fp, $sp, 28

        not $a0, $a0 #negate $
        move $s1, $a1 #store $
        li $a1, 1 #set $a1 to

        li $a2, '+' #set $a2 t
        jal au_logical #run au

        move $a0, $s1 #set $a0
        not $a0, $a0 #negate $
        move $a1, $v1 #set $a1
        move $s1, $v0 #store $

        li $a2, '+' #set $a2 t
        jal au_logical #run au

        move $v1, $v0 #set $v1
        move $v0, $s1 #reset $

        lw      $fp, 28($sp)
        lw      $ra, 24($sp)
        lw      $a0, 20($sp)
        lw      $a1, 16($sp)
        lw      $a2, 12($sp)
        lw      $s1, 8($sp)
        addi    $sp, $sp, 28
        jr $ra
```

Fig. 24. Two's Compliment 64 Bit Implementation

D. log_mul

Lets first look at the strategy to implement binary multiplication. Refer to Fig. 25 to see the strategy layed out. We will start by setting the product to 0. Then, for every digit in the multiplier, if the LSB is 1 then we will add the multiplicant to the product. We will then shift the multiplier to the right by one, and the multiplicand to the left by 1, before repeating the same cycle. Using this method, we can using the addition we coded earlier to massibly help out in taking care of our multiplication.

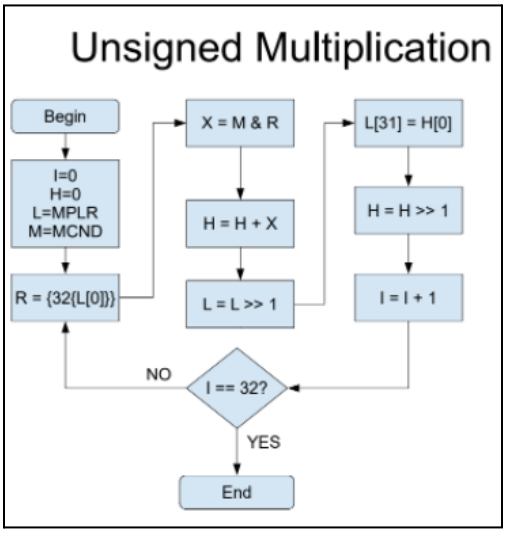

Fig. 25. Log_mul logic

Now, we need to translate this basic logic into MIPS. In Fig. 26, we can see the binary implementation of the above logic from Fig. 25.

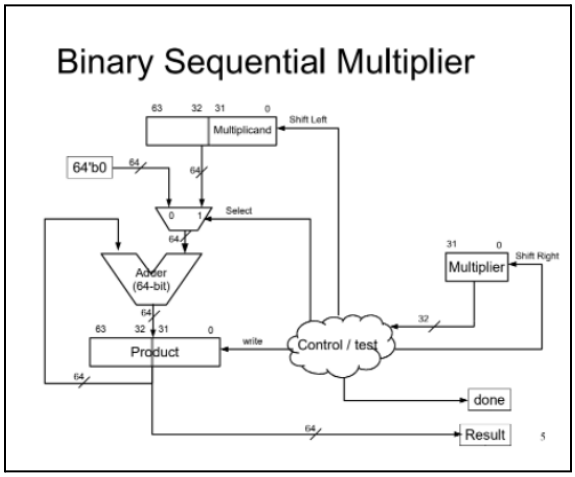

Fig. 26. Binary Sequential Multiplier

Here is the code to implement both the unsigned and signed multiplication.

```
mul_unsigned:

        addi    $sp, $sp, -40
        sw      $s4, 40($sp) #Hi
        sw      $s3, 36($sp)
        sw      $fp, 32($sp)
        sw      $s2, 28($sp)#multiplier
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $s0, 12($sp)
        sw      $s1, 8($sp) #multiplicand M
        addi    $fp, $sp, 3

        li $s4, 0
        li $s3, 0

        li $s0, 0
        li $v1, 0

        mult_helper:
        extract_nth_bit($t1,$s2,$zero)

        move $a0, $t1
        jal bit_replicator

        move $t1, $v0

        and $s3, $t1, $s1
        move $a0, $s4
        move $a1, $s3
        li $a2,

        jal au_logical

        move $s4, $v0

        srl $s2, $s2, 1
        li $t0, 0
        extract_nth_bit($t1, $s4, $t0)
        srl $s4, $s4, 1

        li $t0, 31
        li $t5, 0
        insert_to_nth_bit($s2, $t0, $t1, $t5)
        addi $s0, $s0, 1
        bne $s0, 32, mult_helper

        move $v0, $s2


        lw      $s4, 40($sp)
        lw      $s3, 36($sp)
        lw      $fp, 32($sp)
        lw      $s2, 28($sp)
        lw      $ra, 24($sp)
        lw      $a0, 20($sp)
        lw      $a1, 16($sp)
        lw      $s0, 12($sp)
        lw      $s1, 8($sp)
        addi    $sp, $sp, 40
        jr $ra
```

```
mul_signed:

        addi    $sp, $sp, -32
        sw      $fp, 32($sp)
        sw      $s2, 28($sp)
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $s0, 12($sp)
        sw      $s1, 8($sp)
        addi    $fp, $sp, 32

        move $s1, $a0
        move $s2, $a1

        bgez $a0,skip1
        jal twos_complement
        move $s0, $s1

        move $s1, $v0

        skip1:

        move $a0, $a1

        bgez $a0,skip2
        jal twos_complement
        move $a1,$s2
        move $s2, $v0

        skip2:
        move $a0, $s0

        jal mul_unsigned

        move $t5, $v0

        li $t2, 31
        extract_nth_bit($t0, $a0, $t2)
        extract_nth_bit($t1, $a1, $t2)
        move $a0, $v0
        move $a1, $v1
        xor $t0, $t0, $t1
        beqz $t0, mul_finish

        jal twos_complement_64bit

        mul_finish:
        lw      $fp, 32($sp)
        lw      $s2, 28($sp)
        lw      $ra, 24($sp)
        lw      $a0, 20($sp)
        lw      $a1, 16($sp)
        lw      $s0, 12($sp)
        lw      $s1, 8($sp)
        addi    $sp, $sp, 32
        jr $ra
```

Fig. 27. Mul_unsigned and Mul_signed implementation

E. log_div

Lets first look at the strategy to implement binary division. Refer to Fig. 28 to see the strategy layed out. To start, we load the divisor in the upper half of a 64-bit register, and the dividend in the lower half of a separate remainder register. Next, we will run a loop 32 times. In this loop, we will subtract the divisor from the remainder. If the result of this subtraction is negative, we will undo the subtraction by adding the divisor to the remainder, shifting the quotient one bit to the left, inserting 0 into the LSB. Otherwise, if the subtraction is a positive number, we will keep the result as is, shifting the quotient resiter to the left one time, but this time inserting 1 into the LSB.

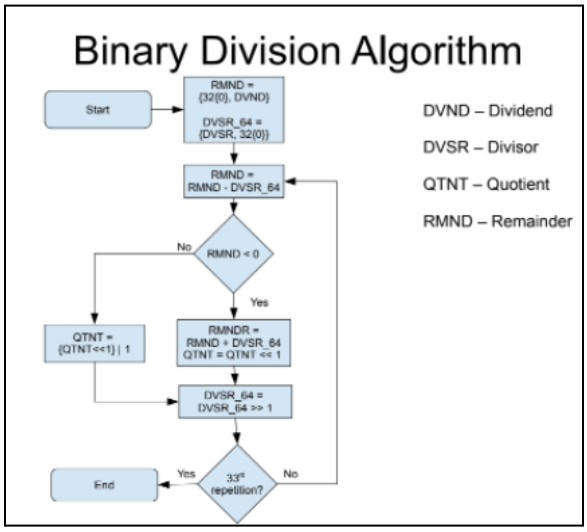

Fig. 28. Log_div logic

Now, we need to translate this basic logic into MIPS. In Fig. 29, we can see the binary implementation of the above logic from Fig. 28.

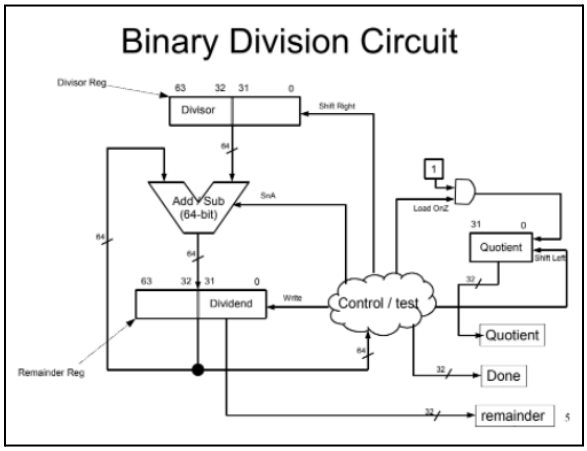

Fig. 29. Binary Sequential Multiplier

```
div_unsigned:

        addi    $sp, $sp, -40
        sw      $s4, 40($sp)
        sw      $s3, 36($sp)
        sw      $fp, 32($sp)
        sw      $s2, 28($sp)
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $s0, 12($sp)
        sw      $s1, 8($sp)
        addi    $fp, $sp, 40

        #s0 - quotient, $s1 - divisor, $s2 - rema
        li $s3, 0


        div_loop:
        sll $s2, $s2, 1

        li $t0, 31
        extract_nth_bit($t1, $s0, $t0)
        li $t0, 0
        insert_to_nth_bit($s2, $0, $t1, $t0)

        sll $s0, $s0, 1
        move $a0, $s2
        move $a1, $s1
        li $a2, '-'
        jal au_logical

        move $s4, $v0
        bltz $s4, skip5

        move $s2, $s4
        li $t0, 1
        li $t2, 0
        insert_to_nth_bit($s0, $zero, $t0, $t2)

        skip5:
        addi $s3, $s3, 1
        bne $s3, 32, div_loop

        move $v0, $s0
        move $v1, $s2

        lw      $s4, 40($sp)
        lw      $s3, 36($sp)
        lw      $fp, 32($sp)
        lw      $s2, 28($sp)
        lw      $ra, 24($sp)
        lw      $a0, 20($sp)
        lw      $a1, 16($sp)
        lw      $s0, 12($sp)
        lw      $s1, 8($sp)
        addi    $sp, $sp, 40
        jr $ra
```

Fig. 29. Div_unsigned implementation

Here is the code to implement both the unsigned and signed division.

```
div_signed:

        addi    $sp, $sp, -40
        sw      $fp, 40($sp)
        sw      $s4, 36($sp)
        sw      $s3, 32($sp)
        sw      $s2, 28($sp)
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $s0, 12($sp)
        sw      $s1, 8($sp)
        addi    $fp, $sp, 40

        move $s0, $a0 #S0 - divident / q
        move $s1, $a1 #S1 - divisor

        li $s2, 0 #$s2 - remainder

        move $s3, $a0

        bgtz $s0, skip3
        jal twos_complement
        move $s0, $v0

        skip3:

        bgtz $s1, skip4
        move $a0, $s1
        jal twos_complement
        move $s1, $v0
        move $a0, $s3

        skip4:

        jal div_unsigned
        move $s2, $v0
        move $s3, $v1

        li $t0, 31
        extract_nth_bit($s0, $a0, $t0)

        extract_nth_bit($t2, $a1, $t0)
        xor $t3, $s0, $t2

        bne $t3, 1, skip6
        move $a0, $s2
        jal twos_complement

        move $s2, $v0

        skip6:
        bne $s0, 1, skip7
        move $a0, $s3
        jal twos_complement
        move $s3, $v0
        skip7:
        move $v0, $s2
        move $v1, $s3

        lw      $fp, 40($sp)
        lw      $s4, 36($sp)
        lw      $s3, 32($sp)
        lw      $s2, 28($sp)
        lw      $ra, 24($sp)
        lw      $a0, 20($sp)
        lw      $a1, 16($sp)
        lw      $s0, 12($sp)
        lw      $s1, 8($sp)

        addi    $sp, $sp, 40
        jr $ra
```

Fig. 30. Div_signed implementation

VI. TESTING

A. Normal ALU MIPS Implementation

To test out our logical ALU, we need some sort of reference to check if the test values are accurate. While this can be done by hand, it is much better form to implement a normal ALU for higher accuracy. This normal ALU can use normal MIPS commands such as add, sub, mult, and div. We can implement this normal ALU in CS47_proj_alu_normal.asm. Similar to our logical alu, we will figure out what symbol register $a2 contains, and branch to the appropriate operation. Additionally, all the input and output registers will be the same our logical ALU. The two inputs will be given in $a0 and $a1, and the outputs should go to $v0, and $v1 if the operation is multiplication or division.

```
au_normal:
        beq $a2, '+', add
        beq $a2, '-', sub
        beq $a2, '/', div
        beq $a2, '*', mul

        add:
        add $v0, $a0, $a1
        jr $ra

        sub:
        sub $v0, $a0, $a1
        jr $ra

        mul:
        mult $a0, $a1
        mflo $v0
        mfhi $v1
        jr $ra

        div:
        div $a0, $a1
        mflo $v0
        mfhi $v1
        jr $ra
```

Fig. 32. Normal ALU Implementation

B. Correct Output

Once our normal ALU has been completed, the file proj-auto-test.asm will run both logical.asm and normal.asm and compare the results. It will run 40 different tests of addition, subtraction, multiplication, and division, and return a score showing how many are accurate, along with the results for each individual test. If everything is correct and all the test results are passed, the following output will be printed, as seen in Fig. 31. If not all test cases are passing, the score out of 40 will represent how many test cases are working so know exactly which test cases are not working properly:

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

-- program is finished running --
```

Fig. 33. Testing Passed Output

VII. CONCLUSION

This project helped me thoroughly understand the logic behind the the logical implementation of mathematical operations using MIPS. Not only did I learn a lot more about how MIPS can be used effectively and cleanly, but I learned a lot about the logic and intuition behind binary addition, subtraction, multiplication, and division. This project helped me reinforce concepts that we learned throughout CS 47, and was an excellent way to use everything we learned in one final project.

REFERENCES

[1] D. A. Patterson and J. L. Hennessy, Computer Organization and Design (5th Edition). Waltham, MA: Morgan Kaufman, 2013, pp. 178-195.

[2] K. Patra. CS 47. Class Lecture, Topic: “Addition Subtraction Logic.” San Jose State University, San Jose, CA, April 14, 2016.

[3] K. Patra. CS 47. Class Lecture, Topic: “Multiplication Logic.” San Jose State University, San Jose, CA, April 19, 2016.

[4] K. Patra. CS 47. Class Lecture, Topic: “Division Logic.” San Jose State University, San Jose, CA, April 21, 2016.

[5] Vollmar, K. (2017, December 7). MARS (MIPS Assembler and Runtime Simulator). Mars mips simulator. Retrieved May 5, 2022, from courses.missouristate.edu/kenvollmar/mars/