Logical Implementation of ALU Using MIPS Assembly Programming

Sanjana Nagwekar
CS 47 Section 80
San Jose State University
San Jose, USA
sanjana.nagwekar@sjsu.edu

***Abstract*— This report outlines the logical implementation of an Arithmetic Logic Unit (ALU) using MIPS assembly programming. There were two procedures that were required to be implemented, one using just normal math operations and another only using MIPS logical operations. The Arithmetic Logic Unit is responsible for taking two input operands and producing arithmetic results based on an operation code. To validate the implementation and correctness of the code, a test file with 40 cases was used as well. This report emphasizes the significance of digital hardware implementation and provides details about the logical analysis and procedures that were incorporated.**

Keywords—ALU, logical operations, MIPS, implementation

I. Introduction

The Arithmetic Logic Unit (ALU) is an essential component of any digital system, responsible for performing arithmetic and logical operations on input operands. The logical implementation of an ALU using MIPS assembly programming provides insight into the underlying working of digital arithmetic operations.

The objective of this project is to create a calculator that performs the basic mathematical functions add, subtract, multiply, and divide. This project requires the implementation of concepts that were taught throughout the entirety of the CS 47 course, primarily, program logic flow and design, assembly code, and Boolean logic. This program requires the use of MIPS assembly language with a MARS IDE to simulate a MIPS environment. This project also focuses on the implementation of two main procedures: au_normal and au_logical. After assembling this completed program, it should successfully pass all 40 test cases written in the test file.

II. Requirments

A. *Initial Requirements:* in order to get started with the project, there are many resources that are required.

1) Software Setup: the most basic requirement for this project is to first ensure that the computing device contains the software applications to support and implement this program.

a) MARS: MIPS Assembler and Runtime Simulator is an Integrated Development Environment (IDE) developed and maintained by Missouri State University that provides a platform for programming in MIPS assembly language. This IDE is capable of editing, assembling, and executing MIPS assembly code which is exactly what this project requires. The MARS IDE is compatible with the following operating systems: macOS, Windows, and Linux [1]. For installation, visit the MARS website and download the installer file appropriate for their system. Open the installer file after downloading it and follow the on-screen directions to finish the installation. Launch MARS IDE to start a brand-new project and write MIPS assembly language programs.

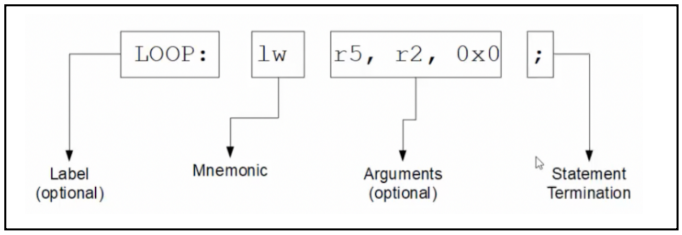
b) Project Files: For this project, six starter files were provided. Each one of these needs to be opened in a new MARS project. It is also essential to ensure that following two settings are turned on: 'Assembles all files in directory' and 'Initialize program counter to global main if defined'. The following is a brief description of each file that this project contains:

- CS47_proj_alu_normal.asm – This file is where the code for the au_normal procedure is to be written.
- CS47_proj_alu_logic.asm – This file is where the code for the au_logical procedure is to be written.
- cs47_proj_macro.asm – This file is for writing any additional macros that may be necessary throughout the project.
- cs47_proj_procs.asm – File containing code for string formatting that should not be edited
- cs47_common_macro.asm – File containing general macros that perform basic and essential functions that should also not be edited
- proj-auto-test.asm – File containing code for testing the written procedures that should not be edited

After successfully installing MARS IDE and uploading all project files, the device is equipped with the tools it needs to write and assemble this project. Without writing the two procedures, the execution of assembling and running the files would result in 0/40 test cases passed.

2) Preliminary knowledge: In order to be able to start writing the code for this project, there are many fundamental concepts that are essential to know.

a) Assembly code: MIPS assembly language is a low-level programming language that is used to write programs that can be executed by a computer's central processing unit (CPU). It is a middle language between higher-level language (such as Java or Python) and machine code (sequences of 0s and 1s). Assembly code can be broken down into smaller



parts, the Label, Mnemonic, Arguments, and Statement termination as such:

The label is an optional symbol that represents a piece of data's memory address [2]. Labels can be classified as local or global which determines the scope of which they can be visible. A local label can only be visible throughout a specific section of the code, whereas a global variable can be visible throughout an entire program.

The mnemonic is an abbreviation for a particular instruction that the processor can execute, for example, 'lw' for loading data from memory or 'sw' for storing data to memory.

Arguments are input values that are used for the instruction that can be but not limited to registers, immediate values, or memory locations. A register is user-accessible temporary storage that MIPS provides that can be accessed very easily. There are two main types of registers: general purpose registers and special purpose registers. The following are just a few general purpose registers that are heavily used within this project:

- $v0-$v1 – values for function results and expression evaluation
- $a0-$a3 – arguments
- $t0-$t9 – temporaries (values that are not preserved across procedure calls)
- $s0-$s7 –saved temporaries (preserved values across procedure calls)
- $fp and $ra – frame pointer and return address (used for procedure calls)

The following are special purpose registers that are also necessary to know for this project:

- PC – program counter (points to the next executed instruction)
- Hi/Lo – used to hold results of multiplication/division operations

Another important aspect of MIPS assembly code is macros which are pre-written pieces of code that may be added into a program using a single instruction, similar to functions in other high-level programming languages. Macros are defined by '.macro' followed by its name, any arguments that it requires, and then the actual instructions.

Lastly, it is important to note that there are three different types of instructions in MIPS assembly code: register (R type), immediate (I type), and jump (J type).

- R-type instructions are used for arithmetic and logical operations involving two registers,
- I-type instructions are used for operations involving an immediate value and a register
- J-type instructions are used to jump to a different memory address.

All three of these types of instructions work together and are heavily used throughout this project.

b) Boolean logic: In digital electronics and computer science, boolean logic is a fundamental idea that deals with operations on binary variables (i.e., variables that can only take on one of two potential values: 0 or 1). Binary variables are frequently expressed in Boolean logic by logical phrases that use logical operators like AND, OR, and NOT.

AND is a logical operator that takes in two binary input variables and produces an output of 1 only when both input variables are 1; otherwise, it produces an output of 0. The truth table below represents this AND operation.

Inputs		Output
A	B	Y = A.B
0	0	0
0	1	0
1	0	0
1	1	1

OR is a logical operator that takes in two binary input variables and produces an output of 1 if at least one of the input variables is 1; otherwise, it produces an output of 0. The truth table below represents this OR operation.

Inputs		Output
A	B	Y = A+B
0	0	0
0	1	1
1	0	1
1	1	1

NOT is a unary logical operator that takes in a single binary input and produces a single binary output which negates the input variable (it produces an output of 0 if the input is 1 and vice versa).

Logic gates are also essential ways to represent logic operations, fundamentally, NAND (not of AND), NOR (not of OR), and NOT. These gates can be used in combinations to represent more complex logical operations and create digital circuits.

B. Project Requirements

1) Normal Procedure: The normal procedure is supposed to take three arguments ($a0, $a1, $a2) and use normal math operations in MIPS to compute a result. $a0 and $a1 represent two terms/values for computation and $a2 represents an mathematical operator, and returns the result in $v0 and $v1 accordingly.

2) Logical Procedure: The logical procedure also takes in the same three arguments, but instead of using MIPS mathematical operations, it uses MIPS logic operations. In other words, this procedure requires a strong use of the boolean operators mentioned above (AND, OR, NOT) along with multiple procedures to perform these calculations without using add, subtract, multiply, and divide. For this procedure it is important to consider all types of test cases such as negative numbers, and signed vs unsigned numbers.

3) Test Cases: The last requirment for this project is for all 40 test cases to match after assembling the procedures.

III. Design and Implementation

A. Normal Procedure

This MIPS assembly code below describes the function "au_normal," which performs the arithmetic operations addition, subtraction, multiplication, and division, using three arguments: $a0 for the first operand, $a1 for the second operand, and $a2 for the operation code. Because MIPS assembly language does not support arguments that are character literals such, '+' for addition, '-' for subtraction, '*' for multiplication, and '/' for division, it is important to convert these symbols into values based on their ASCII codes. The ASCII character for addition '+' is equal to 43, subtraction '-' is 45, for multiplication '*' is 42, and for division '/' is 47.

```
au_normal:
# TBD: Complete it
    add $t0, $zero, $a0   # Move the first operand to temporary register $t0
    add $t1, $zero, $a1   # Move the second operand to temporary register $t1

    # Perform the arithmetic operation based on its code in $a2
        # Addition sign = 43
        # Subtraction sign = 45
        # Multiplication sign = 42
        # Division sign = 47
    beq $a2, 43, ADD_operation   # Addition
    beq $a2, 45, SUB_operation   # Subtraction
    beq $a2, 42, MUL_operation   # Multiplication
    beq $a2, 47, DIV_operation   # Division

ADD_operation:
    add $v0, $t0, $t1     # Store addition result of $t0 and $t1 in $v0
    j end

SUB_operation:
    sub $v0, $t0, $t1     # Store subtraction result of $t0 and $t1 in $v0
    j end

MUL_operation:
    mul $v0, $t0, $t1     # Store multiplication result of $t0 and $t1 in $v0
    mfhi $v1              # Move the HI result to $v1
    j end

DIV_operation:
    div $t0, $t1
    mflo $v0              # Move the quotient of $t0 and $t1 to $v0
    mfhi $v1              # Move the remainder to $v1

end:
    jr $ra                # Return to the calling function
```

The first and second operands are transferred to temporary registers $t0 and $t1, respectively, in the code's first two lines. The branch if equal (beq) instruction is used in the second portion of the code to decide which arithmetic operation should be carried out based on the operation code ($a2). For example, the code branches to the ADD_operation section if $a2 equals the ASCII character for addition (43), and the same goes for the subtraction, multiplication, and division operations.

The appropriate mathematical operation is carried out by the ADD_operation, SUB_operation, MUL_operation, and DIV_operation sections, which then store the outcome in $v0 (and $v1 for multiplication and division, respectively). For multiplication, the mfhi instruction moves the HI result to $v1, and for division, the mflo instruction moves the quotient of $t0 and $t1 to $v0.

The end label is used to jump to the instruction after the jal (jump and link) instruction and the jr (jump register) instruction returns to the calling function.

B. Logical Procedure

1) Macros: There are two different macros that are defined and used in order to simplify the code for the other procedures.

a) Insert: This macro is needed to insert a single bit at the nth position in a bit pattern. This macro takes four arguments: the destination register (the bit pattern that it is inserted into), source register (the n position value), shift register (bit value being inserted), and mask register (holds temporary mask).

```
.macro insert($destReg, $srcReg, $shiftReg, $maskReg)
        # Load the immediate value 1 into `maskReg`
        li $maskReg, 1
        # Shift left logical `maskReg` by `srcReg` bits to create a mask with a 1 in the
        sllv $maskReg, $maskReg, $srcReg
        # Negate the mask to create a mask with a 0 in the `srcReg`th position
        nor $maskReg, $maskReg, $maskReg
        # Clear the `shiftReg`th bit of `destReg` using the mask
        and $destReg, $destReg, $maskReg
        # Shift `shiftReg` left logical by `srcReg` bits
        sllv $shiftReg, $shiftReg, $srcReg
        # Set the `shiftReg`th bit of `destReg` using the shifted value of `shiftReg`
        or $destReg, $destReg, $shiftReg
        .end_macro
```

The way that this macro works is that the value 1 is initially loaded into the maskReg. To generate a mask with a 1 in the srcRegth place, it then shifts maskReg left logical by srcReg bits. To produce a mask with a 0 in the srcRegth location, the mask is negated using the nor instruction. The and instruction is then used to clear the shiftRegth bit of the destReg using this mask. The shiftReg is then shifted left logically by the srcReg bits, and using the or instruction, the shifted value is utilized to set the shiftRegth bit of the destReg. The .end_macro directive marks the end of the insert macro.

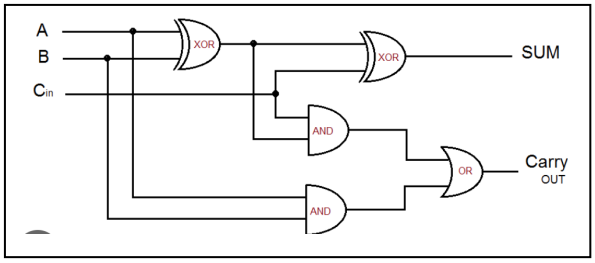
b) Extract: This macro is needed to extract the nth from a bit pattern. This macro takes three arguments: the destination register ($destReg), the source register ($srcReg), and the shift register ($shiftReg). To extract the desired bit value, this macro starts by shifting right logical srcReg by shiftReg bits. The andi command is then used to mask the result's least significant bit (bit 0). The .end_macro directive marks the end of the extract macro.

```
.macro extract($destReg, $srcReg, $shiftReg)
        # Shift right logical `srcReg` by `shiftReg` bits
        srlv $destReg, $srcReg, $shiftReg
        # Mask the least significant bit (bit 0) of the result
        andi $destReg, $destReg, 1
        .end_macro
```

2) Procedures: There are several additional procedures that are crucial in order to successfully implement the au_logical procedure.

a) add_sub_logical: Procedure that uses logical operations to calculate the sum of its first two parameters, $a0 and $a1, and then stores the result in $v0, where the third argument $a2, stands for the initial carry-in bit

This procedure implements full adder logic for two binary numbers A and B. The figure below serves as a visual demonstration of the logical implementation for this procedure.



In this figure, the binary bits being added are represented by the inputs A and B, and Cin is the carry input. While Cout represents the carry output, the output S represents the total of A, B, and Cin.

This procedure works by repeatedly iterating over each bit of the input numbers A and B, the procedure puts this logic into action. It takes the A and B bits that correspond to each bit location, uses the entire adder logic to calculate the sum and carry, and then adds the sum bit to the result number. The output is then returned as the outcome.

```
#PROCEDURE 1
add_sub_logical: # calculate the sum of its first two arguments $a0 and $a1
    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

   li $t0, 0 #this one is i
   li $v0, 0 #This one is S
   extract($v1, $a2, $zero)

addition_loop:
        beq $t0, 32, exit
        extract($t2, $a0, $t0) #Get ith bit of A and place it in t2
        extract($t3, $a1, $t0) #Get ith bit of B and place it in t3
        xor $t4, $t2, $t3 #Let t4 carry the result of Xor A and B
        xor $t5, $t4, $v1 #Let t5 carry the result of Ci xor t4, which is Y
        and $t8, $t2, $t3 #Let t8 carry the result of A and B
        and $t7, $t4, $v1 #Let t7 carry the result of Cin and t4
        or  $t9, $t8, $t7 #Let t9 carry the result of t5 or t8, which is the C
        move $v1, $t9 #Change the content of Cin to CO
        insert($v0, $t0, $t5, $t4)
        addi $t0, $t0, 1
        j addition_loop
exit:

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

Using a loop, each bit in the two input arguments is iterated over, which then applies logical operations to find the appropriate bit in the output sum. The addition additionally takes into account the carry-in ($a2). The final sum is returned in $v0 once all 32 bits have been processed, after which the loop terminates. Before handing back control to the calling function, the procedure recovers the original values of $fp, $ra, $a0, $a1, and $a2 by using a stack to hold temporary values during the calculation.

b) add_logical: Procedure that calls add_sub_logical to perform an addition operation.

```
#PROCEDURE 2
add_logical:  # calls add_sub_logical procedure to perform addition.
        # Allocate space on the stack for saved registers and function arguments
        addi    $sp, $sp, -24
        sw    $fp, 24($sp) # Save frame pointer
        sw    $ra, 20($sp) # Save return address
        sw    $a0, 16($sp) # Save function argument 1
        sw    $a1, 12($sp) # Save function argument 2
        sw    $a2,  8($sp) # Save function argument 3
        addi    $fp, $sp, 24  # Set up a new stack frame by setting the frame pointer to the current stack pointer plus 24

        # Load the value 0x00000000 into register $a2, which is used as a temporary variable to store the result of the ad
        li $a2, 0x00000000
        # Call the add_sub_logical subroutine to perform the actual addition operation
        jal add_sub_logical

        # Restore saved registers and function arguments from the stack
        lw    $fp, 24($sp) # Restore frame pointer
        lw    $ra, 20($sp) # Restore return address
        lw    $a0, 16($sp) # Restore function argument 1
        lw    $a1, 12($sp) # Restore function argument 2
        lw    $a2,  8($sp) # Restore function argument 3

        # Deallocate stack space by adding 24 to the stack pointer
        addi    $sp, $sp, 24
        jr $ra    # Jump back to the return address
```

To store registers and function arguments, space is first allocated on the stack. The frame pointer, return address, and three function parameters are then stored onto the stack. Initiated with the value 0x00000000, the third function parameter ($a2) serves as a temporary variable to hold the outcome of the addition operation. It then invokes the "add_sub_logical" procedure, supplying the three function arguments as parameters, to complete the addition operation.

The procedure deallocates the space on the stack by adding 24 to the stack pointer when the operation is finished. It also restores the saved registers and function arguments from the stack. The return address of the calling program is then reached.

c) sub_logical: Procedure that calls add_sub_logical to perform a subtration operation.

```
#PROCEDRURE 3
sub_logical: # Performs subtraction by calling add_sub_logical procedure
        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

        # Negate the second argument by performing a bitwise NOT operation on it and store the result in $a1
        not $a1, $a1
        # Load the value 0xFFFFFFFF into register $a2, which is used as a temporary variable to store the result of
        li $a2, 0xFFFFFFFF
        # Call the add_sub_logical subroutine to perform the actual subtraction operation
        jal add_sub_logical
        # Restore the values of $fp, $ra, $a0, and $a1 from the stack
        lw    $fp, 24($sp)
        lw    $ra, 20($sp)
        lw    $a0, 16($sp)
        lw    $a1, 12($sp)
        lw    $a2,  8($sp)
        addi    $sp, $sp, 24    # Deallocate the space on the stack for local variables
        jr $ra
```

The implementation of this procedure is similar to add_logical. However, in this procedure, the second argument is negated by applying a bitwise NOT operation to it, and the result is saved in register $a1. After that, it calls add_sub_logical to carry out the actual subtraction operation and puts the value 0xFFFFFFFF into register $a2, which is used as a temporary variable to record the subtraction result.

d) twos_complement: Procedure that converts any operand into its corresponding two's complement form. The technique first creates a new stack frame, allots space on the stack for function arguments and stored registers, and negates the operand to determine its one's complement. It then uses the "add_logical" procedure to add 1 to the one's complement in order to obtain the two's complement after setting the second parameter to 1. Then it deallocates the stack space, jumps back to the return address, and restores the saved registers and function arguments from the stack.

In order to convert the operand into its two's complement, the "twos_complement_if_neg" function first determines whether it the operand is negative. If it is, it then runs the "twos_complement" subroutine. If the operand isn't negative, it simply returns it.

```
# PROCEDURE 4
twos_complement: # Converts any operand to its corresponding two's complement form.
        addi    $sp, $sp, -20
        sw      $fp, 20($sp)
        sw      $ra, 16($sp)
        sw      $a0, 12($sp)
        sw      $a1,  8($sp)
        addi    $fp, $sp, 20
        # Compute the one's complement of the operand by negating it
        not $a0, $a0
        # Set $a1 to 1 to add 1 to the one's complement to get the two's complement
        li $a1, 1
        # Call the add_logical subroutine to add 1 to the one's complement
        jal add_logical
        lw $fp, 20($sp)
        lw $ra, 16($sp)
        lw $a0, 12($sp)
        lw $a1,  8($sp)
        addi $sp, $sp, 20
        jr $ra
twos_complement_if_neg:
        addi    $sp, $sp, -16
        sw      $fp, 0($sp)
        sw      $ra, 4($sp)
        sw      $a0, 8($sp)
        # Check if the operand is negative (i.e., less than zero)
        blt     $a0, $zero, skip_call
        # If the operand is non-negative, simply return it
        move    $v0, $a0
        j       end_now
skip_call:
        # If the operand is negative, call the twos_complement subroutine to convert it to two's comp
        jal     twos_complement
end_now:
        # Restore the values of $a0, $ra, and $fp from the stack
        lw      $a0, 8($sp)
        lw      $ra, 4($sp)
        lw      $fp, 0($sp)
        addi    $sp, $sp, 16
        jr      $ra
```

e) twos_complement_64bits: Procedure to convert the contents of two 32-bit registers into a 64-bit two's complement result.

```
#PROCEDURE 5
twos_complement_64bit: # Convert the contents of two 32-bit registers into a 64-b
        addi    $sp, $sp, -24
        sw      $fp, 20($sp)
        sw      $ra, 16($sp)
        sw      $a0, 12($sp)
        sw      $a1, 8($sp)
        addi    $fp, $sp, 20

        not $a0, $a0          # apply bitwise NOT to Lo of the number
        not $a1, $a1          # apply bitwise NOT to Hi of the number
        move $s3, $a1         # save original Hi of the number to register s3
        li $a1, 1             # set the Hi argument to 1

        jal add_logical       # call add_logical to add 1 to the 2's complement o

        move $s4, $v0         # save the Lo part of the result to register s4
        move $a1, $v1         # set the Hi argument to the Hi result of add_logic
        move $a0, $s3         # set the Lo argument to the original Hi of the num

        jal add_logical       # call add_logical to add the original Hi of the nu

        move $v1, $v0         # move the Hi part of the result to v1
        move $v0, $s4         # move the Lo part of the result to v0

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

To implement this procedure, the bitwise NOT operation is used to first negate the values of the Lo and Hi registers. The add_logical subroutine is then called to add one to the number's two-component complement. The Hi component of the result register $v1 and the Lo part of the result register $v0 each contain the outcome of the addition. The add_logical procedure is called once again after setting the parameters to add the number's original Hi to the outcome. Finally, the values of $fp, $ra, $a0, and $a1 are restored from the stack before returning. The Hi part of the result is put to $v1 and the Lo part is moved to $v0.

f) bit_replicator: Procedure to replicate an individual but 32 times to become 32 bits. The first step of the process involves allocating memory on the stack and storing the values for the current frame pointer and return address. The algorithm then determines whether the replicated bit that was specified is equal to 0. It sets all 32 bits to 0 if the bit is 0. Otherwise, it uses a mask (0xFFFFFFFF) with all bits set to 1 to set all 32 bits to 1. The procedure restores the previously saved values after the replication is finished and then connects back to the caller.

```
# PROCEDURE 6
bit_replicator: #Replicates a certain individual bit 32 times to b
        addi $sp, $sp, -16
        sw $fp, 12($sp)
        sw $ra, 8($sp)
        sw $a0, 4($sp)
        addi $fp, $sp, 12

        # check if bit to replicate is 0
        beq $a0, 0, case_zero
        li $v0, 0xFFFFFFFF # set all bits to 1

        # jump to end of replication
        j case_end

case_zero:
        li $v0, 0 # set all bits to 0

case_end:
        lw $fp, 12($sp)
        lw $ra, 8($sp)
        lw $a0, 4($sp)
        addi $sp, $sp, 16
        jr $ra
```

g) mul_unsigned: Procedure to perform unsigned multiplicaiton of two 32-bit arguments. This procedure takes two arguments (multiplicand and multiplier) and utilzes the two's complement procedures and bit_replicator along with logical operator XOR to calculate a product.

```
#PROCEDURE 7
mul_unsigned: # Performs unsigned multiplication of two arguments
        addi    $sp, $sp, -60
        sw      $fp, 60($sp)
        sw      $ra, 56($sp)
        sw      $a0, 52($sp)
        sw      $a1, 48($sp)
        sw      $a2, 44($sp)
        sw      $a3, 40($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, 60
        li $s0, 0 # Counter i
        li $s1, 0 # Hi part of the result
        move $s2, $a1 # L, the multiplier
        move $s3, $a0 # M, the multiplicand

loop:
         # Check if all 32 bits of the multiplier have been processed
        beq $s0, 32, Exit
         # Extract the least significant bit of the multiplier
        extract($a0, $s2, $zero)
        jal bit_replicator # Call the bit_replicator subroutine to replicate the bit and store

        move $s4, $v0 # $s4 now has the replicated bit
         # Calculate the product of the multiplicand and the replicated bit
        and $s5, $s3, $s4 # X = M & R
        move $a1, $s5 # H + X
        la $a0, ($s1) # Store the sum in $a1

        jal add_logical # Call the add_logical subroutine to add $a0 and $a1 and store the resu

        move $s1, $v0 # Assign the value of the result to H
         # Shift the multiplier right logical by 1
        srl $s2, $s2, 1
         # Extract the least significant bit of H and store it in $s7
        extract($s7, $s1, $zero)
         # Assign the least significant bit of L to the value of $s7
        li $t1, 31
        insert($s2, $t1  $s7, $t9)
        srl $s1, $s1, 1 #Shift the value of H right logical by 1
        addi $s0, $s0, 1 #Increment the counter
        j loop

Exit:
        move $v0, $s2 #Assign the value of $v0 by L, which is the Lo part
        move $v1, $s1 #Assign the value of $v1 by H, which is the hi part

        lw      $fp, 60($sp)
        lw      $ra, 56($sp)
        lw      $a0, 52($sp)
        lw      $a1, 48($sp)
        lw      $a2, 44($sp)
        lw      $a3, 40($sp)
        lw      $s0, 36($sp)
        lw      $s1, 32($sp)
        lw      $s2, 28($sp)
        lw      $s3, 24($sp)
        lw      $s4, 20($sp)
        lw      $s5, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7,  8($sp)
        addi    $sp, $sp, 60
        jr      $ra
```

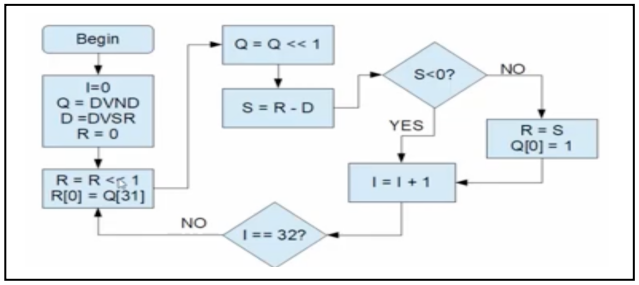
In this code, a counter and variables that will hold the high and low portions of the result are first initialized. After that, it creates a loop that runs 32 times, one for each multiplier bit. The least significant bit of the multiplier is taken out and replicated by a program called bit_replicator, which then stores the result in $v0, for each iteration of the loop. After computing the multiplicand and replicated bit's product, the procedure stores the result in $a1 and invokes the add_logical subroutine to add $a0 and $a1 before storing the result in $v0. The program then modifies the result's high portion and moves the multiplier one logical position to the right. This process is repeated until the low and high parts of the product are stored in $v0 and $v1.

h) mul_signed: Procedure to perform multiplication of signed or unsigned arguments. This procedure starts by producing a new frame pointer and stores the register contents into the stack. The first and second parameters, $a0 and $a1, are then transferred into the registers $s6 and $s5. If either argument is negative, the procedure runs a different function called "twos_complement_if_neg" to obtain their two's complement. To conduct the actual multiplication, "mul_unsigned," is then called. The function then performs

an XOR operation on both input numbers, extracts the most significant bit, and verifies the outcome to determine if the result of multiplication is negative or not. If the result is negative, the two's complement of the result is obtained by calling "twos_complement_64bit".

```
mul_signed: # Performs multiplication of either signed or unsigned arguments
        addi    $sp, $sp, -60
        sw      $fp, 60($sp)
        sw      $ra, 56($sp)
        sw      $a0, 52($sp)
        sw      $a1, 48($sp)
        sw      $a2, 44($sp)
        sw      $a3, 40($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, 60

move $s6, $a0 # let N1, which will be $s6, be $a0. That is, N1 = $a0
move $s5, $a1 # let N2, which will be $s5, be $a1. That is, N2 = $a1


jal twos_complement_if_neg #v0 contains the two complement of $a0 if $a0 is negative
move $a3, $v0 #a3 will hold the twos complement if $a0 was negative
move $t6, $s5 #let N2, which will be $t6, be $a1. That is, N2 = $a1
move $a0, $t6 #Move N2 into a0. This way, when twos_complement_if_neg is called, a0 
jal twos_complement_if_neg
move $a1, $v0
move $a0, $a3
jal mul_unsigned

li $t0, 31
extract($t1, $s6, $t0) #extract $a0[31] and place it in $t1
extract($t2, $t6, $t0) #extract $a1[31] and place it in $t2
xor $t3, $t1, $t2 #Xor between t2 and t1, place it in t3, and this is the sign of th
bne $t3, 1, cont1
move $a0, $v0
move $a1, $v1
jal twos_complement_64bit
cont1:

        lw      $fp, 60($sp)
        lw      $ra, 56($sp)
        lw      $a0, 52($sp)
        lw      $a1, 48($sp)
        lw      $a2, 44($sp)
        lw      $a3, 40($sp)
        lw      $s0, 36($sp)
        lw      $s1, 32($sp)
        lw      $s2, 28($sp)
        lw      $s3, 24($sp)
        lw      $s4, 20($sp)
        lw      $s5, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7,  8($sp)
        addi    $sp, $sp, 60
        jr      $ra
```

i) div_unsigned: Procedure that performs unsigned division of two 32 bit integers. After saving the register values on the stack, this procedure allocates memory for local variables and initializes variables such as $s0 to zero and $s1 to the remainder. The division process is carried out by the technique using a loop. The procedure shifts the dividend left by one bit while in the loop, extracts the 31st bit of the dividend and sets it in the remainder's 0th bit, shifts the dividend left by one bit, and then uses the "sub_logical" procedure to subtract the divisor from the remainder. When subtracting, the procedure increases the remainder if the difference is negative. The technique ends the loop after 32 iterations if the quotient is still not found. Lastly, the register values are restored and the quotient is returned in $v0 and the remainder in $v1. The figure below outlines the logical implementation of this procedure [3].




```
#PROCEDURE 9
div_unsigned: # Performs unsigned division
        addi    $sp, $sp, -60
        sw      $fp, 60($sp)
        sw      $ra, 56($sp)
        sw      $a0, 52($sp)
        sw      $a1, 48($sp)
        sw      $a2, 44($sp)
        sw      $a3, 40($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, 60

li $s0, 0 # i
li $s1, 0 # Remainder
move $s2, $a0 # Dividend
move $s3, $a1 # Divisor

div_loop:
   # Check if i >= 32
        beq $s0, 32, Exit_Div
        # Shift remainder left logical by 1
        sll $s1, $s1, 1
        li $t1, 31
        extract($t2, $s2, $t1) # Extract the 31st bit of divident and place it in $t2
        insert($s1, $zero, $t2, $t9)  # Insert the 31st bit of dividend into the zeroth bit
        # Shift dividend left logical by 1
        sll $s2, $s2, 1
        # Call sub_logical to subtract dividend from remainder
        move $a0, $s1 # Move remainder into $a0
        move $a1, $s3 # Move divisor into $a1
        jal sub_logical
        blt $v0, $zero, increment
        move $s1, $v0 # Move the difference into remainder

        li $t1, 1
        insert($s2, $zero, $t1, $t9)
increment:
        addi $s0, $s0, 1
        j div_loop

Exit_Div:
move $v0, $s2 # Move quotient to $v0
move $v1, $s1 # Move remainder to $v1


        lw      $fp, 60($sp)
        lw      $ra, 56($sp)
        lw      $a0, 52($sp)
        lw      $a1, 48($sp)
        lw      $a2, 44($sp)
        lw      $a3, 40($sp)
        lw      $s0, 36($sp)
        lw      $s1, 32($sp)
        lw      $s2, 28($sp)
        lw      $s3, 24($sp)
        lw      $s4, 20($sp)
        lw      $s5, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7,  8($sp)
        addi    $sp, $sp, 60
        jr      $ra
```

j) div_signed: Procedure that performs division of signed and unsigned arguments. As usual it starts by storing the registers on the stack. The two input arguments, N1 and N2, are then moved into registers $s6 and $s5, respectively. By invoking the twos_complement_if_neg function, it is determined whether N1 is negative and stores the outcome in register $a3. By running the same function again, it is determined whether N2 is negative and stores the outcome in register $v0. The div_unsigned function is then called using the absolute values of N1 and N2. By using the logical XOR operation on the sign bits of N1 and N2, it identifies the sign of the outcome and puts it in register $t3. If the outcome is negative, it calls the twos_complement function to convert it to two's complement and stores the result in register $s4. If either N1 or N2 was negative, then it converts the quotient to its two's complement and is stored in $v0 and $v1.

In other words, this procedure divides signed and unsigned numbers by converting them to two's complements, and if necessary, executing the div_unsigned subroutine. Then, depending on the original signs of N1 and N2, converting the result and quotient back to signed or unsigned form. These activities are carried out by a variety of the procedures mentioned prior. For this reason, order of operations is very important when using MIPS assembly language, because it is important to start off coding the more basic procedures so that they could be utilzed by other more complex procedures later on.

```
#PROCEDURE 10
div_signed: # Performs division of signed and unsigned arguments
        addi    $sp, $sp, -60
        sw      $fp, 60($sp)
        sw      $ra, 56($sp)
        sw      $a0, 52($sp)
        sw      $a1, 48($sp)
        sw      $a2, 44($sp)
        sw      $a3, 40($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, 60

 # Save N1 and N2 in $s6 and $s5 respectively
move $s6, $a0
move $s5, $a1

 # Convert N1 to two's complement if it's negative
jal twos_complement_if_neg
move $a3, $v0

# Convert N2 to two's complement if it's negative
move $t6, $s5
move $a0, $t6
jal twos_complement_if_neg

# Call div_unsigned with the absolute values of N1 and N2
move $a1, $v0
move $a0, $a3
jal div_unsigned

End:
        lw      $fp, 60($sp)
        lw      $ra, 56($sp)
        lw      $a0, 52($sp)
        lw      $a1, 48($sp)
        lw      $a2, 44($sp)
        lw      $a3, 40($sp)
        lw      $s0, 36($sp)
        lw      $s1, 32($sp)
        lw      $s2, 28($sp)
        lw      $s3, 24($sp)
        lw      $s4, 20($sp)
        lw      $s5, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7,  8($sp)
        addi    $sp, $sp, 60
        jr       $ra


# Determine the sign of the result
li $t0, 31
extract($t1, $s6, $t0) # extract the sign bit of N1 and place it in $t1
extract($t2, $t6, $t0) # extract the sign bit of N2 and place it in $t2
xor $t3, $t1, $t2  # the sign of the result is the XOR of the sign bits of N1 and N

# Convert the result to two's complement if it's negative
move $a0, $v0
move $s1, $v1
bne $t3, 1, cont2
jal twos_complement
move $s4, $v0
j cont4
cont2:
move $s4, $v0
cont4:
li $t0, 31

# Convert the quotient to two's complement if N1 or N2 was negative
extract($t1, $s6, $t0)
move $a0, $s1
bne $t1, 1, cont3
jal twos_complement
move $v1, $v0
move $v0, $s4
j End
cont3:
move $v0, $s4
move $v1, $s1
j End
```

k) au_logical

```
au_logical:
        addi    $sp, $sp, -60
        sw      $fp, 60($sp)
        sw      $ra, 56($sp)
        sw      $a0, 52($sp)
        sw      $a1, 48($sp)
        sw      $a2, 44($sp)
        sw      $a3, 40($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, 60

beq     $a2, 43, add_logical
beq     $a2, 45, sub_logical
beq     $a2, 42, mul_signed
beq     $a2, 47, div_signed
```

IV. Testing

Assembling all the finished procedures and running the code will execute the project test file. This test file provides 40 different test cases, and for each test it compares the mathematical result of the au_normal procedure with the au_logical results to see if it is a match. If all procedures are implemented correctly, the output should return 'Total passed 40/40'.

```
(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 --
```

V. Conclusion

In conclusion, the implementation of the two methods au_normal and au_logical effectively demonstrates the logical implementation of the Arithmetic Logic Unit (ALU) using MIPS assembly programming. The au_logical method uses solely MIPS logic operations to implement mathematical operations while the au_normal procedure computes the output using normal math operations of MIPS. The proj-auto-test.asm program's run produced the anticipated results, validating that the ALU implementation functions properly. This project emphasizes the value of learning how the ALU is implemented logically and it provided me with an opportunity to gain more practice programming with MIPS assembly language and also applying all the concepts that I learned throughout this course. Each step of the project required me to dive deeper into these course concepts and test my level of understanding through my code. This project also emphasizes the ease of using supported functions and how much more difficult it is to truly work from scratch.

References

[1] Missouri State University. (2011). *MARS - Mips Assembly and Runtime Simulator*. Mars help idefrom https://courses.missouristate.edu /kenvollmar/mars/Help/Help_4_1/MarsHelpIDE.html

[2] Arm Developer. (n.d.). *ARM Compiler armasm User Guide Version 5.06*. Documentation – arm developer, from https://developer.arm. com/documentation/dui0473/m/symbols--literals--expressions--and-operators/labels

[3] I Khan, A. (2022, January 23). Full Adder Circuit using Logic Gates. Circuits DIY, from https://www.circuits-diy.com/full-adder-circuit-using-logic-gates/