Using MIPS Logic Operations to Create a Basic Calculator

Justin Thai
CS 47, Section 1
San Jose State University
San Jose, CA, USA
justin.thai@sjsu.edu

***Abstract*—This report explains the design and implementation of a MIPS program that can execute basic math operations. The MIPS calculator can perform addition, subtraction, multiplication, and division by using normal and logical operations.**

I. INTRODUCTION

In assembly language programming, a combination of logic operations can create procedures that do basic mathematic operations (addition, subtraction, multiplication, division). Each mathematical operation will have a specific combination of logic operations that are implemented to create it.

The objective of this project is to create a basic logic calculator by implementing logic operations. By using the assembly language MIPS and MARS, a simulator for MIPS, a basic calculator can be assembled with the implementation of MIPS logic operations.

II. REQUIREMENTS

A. MARS Simulator

MARS (MIPS Assembler and Runtime Simulator) is an interactive development environment (IDE) created by Missouri State University. MARS is needed to program MIPS assembly language. This simulator can be acquired through Missouri State University's website (https://courses.missouristate.edu/KenVollmar/MARS/).

B. CS47 Project ASM Files

The CS47 Project ASM Files is where the actual implementation of the logic calculator will go to. All of the ASM files can be acquired through the CS47ProjectI.zip file from the CS47 Project Canvas page.

In this ZIP file, the following ASM files should be present:

1) cs47_common_macro.asm – Contains the common macros needed for the MIPS program to run. This file should not be modified.

2) CS47_proj_alu_logical.asm – This file is where the logical implementation of a basic calculator is done.

3) CS47_proj_alu_normal.asm – This file is where the normal implementation of a basic calculator is done. A normal calculator is needed to compare the results of the logic calculator.

4) cs47_proj_macro.asm – This file will contain any macros that are needed to implement into the MIPS program.

5) cs47_proj_procs.asm – Contains the project procedures needed for the MIPS program to function. This file should not be modified.

6) proj_auto_test.asm – Contains the testing program to see if the implementation of a basic calculator is correctly done. This file should not be modified.

After unzipping the project's ZIP file, the ASM files can be opened and modified (if instructed) using MARS.

III. DESIGN & IMPLEMENTATION

A. Design

To understand how to implement a basic calculator using logic operations, a basic design of each mathematical operation must be created.

1) Addition and Subtraction

In logical addition, the sum of two values is computed by creating a procedure that uses logic operations, specifically a full adder, to determine the sum bit and carry bit. This procedure will be done 32 times, one for each of the 32 bits for both values. After this process iterates 32 times, the sum, which has 32 bits total, of the two values is determined.

Subtraction is just the addition of a negative value, so its logical design is very similar to addition. The only change that is needed for subtraction is to negate the second value before adding it to the first value.

2) Multiplication

For logical multiplication, two values will be taken, the first value is the multiplicand and the second value is the multiplier. In multiplication for MIPS, the product will be 64 bits long, so the result will need to be split into Hi (the higher part of the product) and Lo (the lower part of the product).

First, the first bit (or LSB) of the multiplier is used to multiply with the multiplicand, this will give you a partial product. Then, the multiplier is shifted to the right by 1 and the multiplicand is shifted to the left by 1. Once the shifting is done, the shifted multiplier is multiplied with the shifted multiplicand which will result in another partial product. Both the partial products that have been computed will then be added together to get the partial sum of the full product.

This process will be done 32 times, and, in the end, a 64-bit product should be produced.

3) Division

For logical division, two values will be taken, the first value is the dividend and the second value is the divisor. After the division process, a 32-bit quotient and 32-bit remainder should be produced.

In the division process, the divisor is first aligned with the last bit (or MSB) of the end of the dividend. The section of the dividend that is aligned with the divisor (which will be called X) is then compared to the divisor (which will be called Y). If X is greater than or equal to Y, the quotient bit is 1 and X is subtracted from Y. Otherwise, the quotient bit is 0 and no subtraction is done. The divisor is then shifted one bit to the right and the alignment and comparison process is repeated.

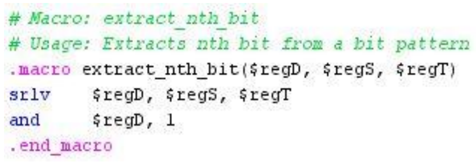
After this division process has iterated 32 times, the quotient and remainder, both of which are 32-bits each, will be computed.

B. Implementation

1) Utility Macros

These macros were created in cs47_proj_macro.asm to help with the implementation of logic math operations.

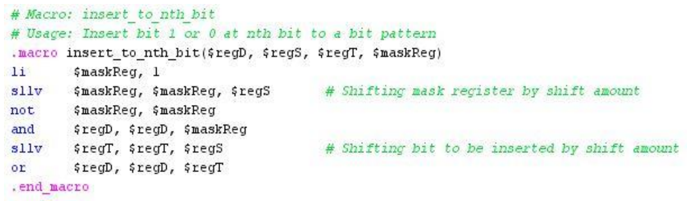
a) extract_nth_bit



```
# Macro: extract_nth_bit
# Usage: Extracts nth bit from a bit pattern
.macro extract_nth_bit($regD, $regS, $regT)
srlv    $regD, $regS, $regT
and     $regD, 1
.end_macro
```

Extract_nth_bit takes the indicated register ($regS) and shifts the register to the right by the indicated amount ($regT) using srlv. The macro will then take the bit at the point after the shifting and save it to $regD.

b) insert_to_nth_bit

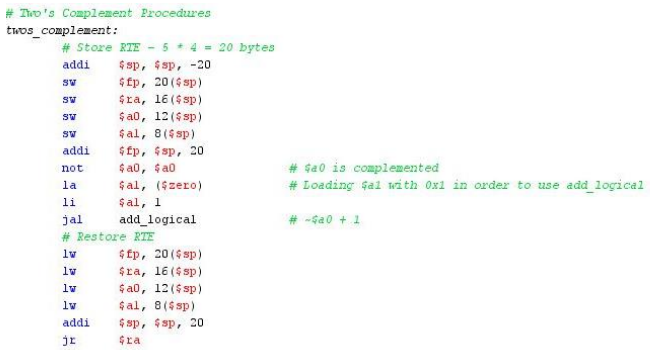


```
# Macro: insert_to_nth_bit
# Usage: Insert bit 1 or 0 at nth bit to a bit pattern
.macro insert_to_nth_bit($regD, $regS, $regT, $maskReg)
li      $maskReg, 1
sllv    $maskReg, $maskReg, $regS       # Shifting mask register by shift amount
not     $maskReg, $maskReg
and     $regD, $regD, $maskReg
sllv    $regT, $regT, $regS             # Shifting bit to be inserted by shift amount
or      $regD, $regD, $regT
.end_macro
```

Insert_to_nth_bit takes a bit ($regT) and inserts it into a register ($regD) at the indicated position ($regS). First, the mask register ($maskReg) is shifted to the left by the indicated amount ($regS) and inverted. Then, the AND operation is used for $regD and $maskReg and the result is stored at $regD. $regT is then shifted to the left by the indicated amount and saved to itself. Finally, $regT will be inserted to $regD and saved to $regD.

2) Utility Procedures

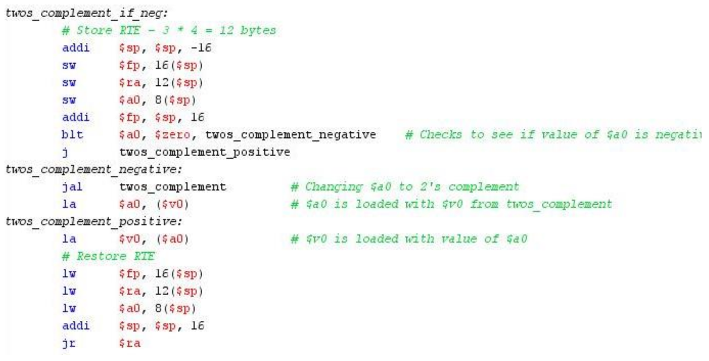
a) twos_complement



```
# Two's Complement Procedures
twos_complement:
        # Store RTE - 5 * 4 = 20 bytes
        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                # $a0 is complemented
        la      $a1, ($zero)            # Loading $a1 with 0x1 in order to use add_logical
        li      $a1, 1
        jal     add_logical             # ~$a0 + 1
        # Restore RTE
        lw      $fp, 20($sp)
        lw      $ra, 16($sp)
        lw      $a0, 12($sp)
        lw      $a1, 8($sp)
        addi    $sp, $sp, 20
        jr      $ra
```

Twos_complement is used to take a value ($a0) and converting it into 2’s complement form (~$a0+1). A frame for the RTE is first created, then $a0 is complemented. The value 1 is then loaded to $a1 to add it with ~$a0. Add_logical is then called to help add ~$a0 and 1 and the result will be stored to $v0 (in add_logical). The RTE frame is restored after add_logical is finished and the $v0, the 2's complement form of the original value, is returned.

b) twos_complement_if_neg



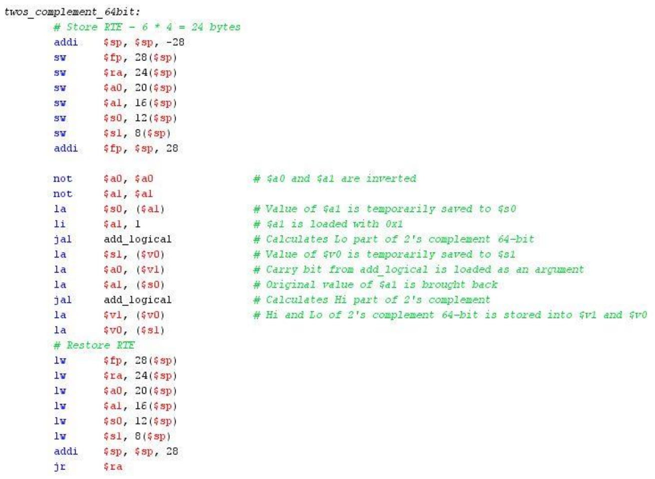
```
twos_complement_if_neg:
        # Store RTE - 3 * 4 = 12 bytes
        addi    $sp, $sp, -16
        sw      $fp, 16($sp)
        sw      $ra, 12($sp)
        sw      $a0, 8($sp)
        addi    $fp, $sp, 16
        blt     $a0, $zero, twos_complement_negative    # Checks to see if value of $a0 is negati
        j       twos_complement_positive
twos_complement_negative:
        jal     twos_complement         # Changing $a0 to 2's complement
        la      $a0, ($v0)              # $a0 is loaded with $v0 from twos_complement
twos_complement_positive:
        la      $v0, ($a0)              # $v0 is loaded with value of $a0
        # Restore RTE
        lw      $fp, 16($sp)
        lw      $ra, 12($sp)
        lw      $a0, 8($sp)
        addi    $sp, $sp, 16
        jr      $ra
```

Twos_complement_if_neg is used to check the sign of a value ($a0). If the value is negative, twos_complement will be called to convert the value to 2's complement form. If the value is positive, the procedure will return the value as is.

A frame for the RTE is created first, then $a0 is checked to see if it is negative using blt. If negative, the procedure will jump to twos_complement for the conversion, and the result from twos_complement is stored to $a0. If positive, the procedure will jump to the return step and frame restoration.

After the value is in its desired form, the $a0 is loaded to $v0 for returning and the RTE frame is restored, ending the procedure.

c) twos_complement_64bit



```
twos_complement_64bit:
        # Store RTE - 6 * 4 = 24 bytes
        addi    $sp, $sp, -28
        sw      $fp, 28($sp)
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $s0, 12($sp)
        sw      $s1, 8($sp)
        addi    $fp, $sp, 28

        not     $a0, $a0                # $a0 and $a1 are inverted
        not     $a1, $a1
        la      $s0, ($a1)              # Value of $a1 is temporarily saved to $s0
        li      $a1, 1                  # $a1 is loaded with 0x1
        jal     add_logical             # Calculates Lo part of 2's complement 64-bit
        la      $s1, ($v0)              # Value of $v0 is temporarily saved to $s1
        la      $a0, ($v1)              # Carry bit from add_logical is loaded as an argument
        la      $a1, ($s0)              # Original value of $a1 is brought back
        jal     add_logical             # Calculates Hi part of 2's complement
        la      $v1, ($v0)              # Hi and Lo of 2's complement 64-bit is stored into $v1 and $v0
        la      $v0, ($s1)
        # Restore RTE
        lw      $fp, 28($sp)
        lw      $ra, 24($sp)
        lw      $a0, 20($sp)
        lw      $a1, 16($sp)
        lw      $s0, 12($sp)
        lw      $s1, 8($sp)
        addi    $sp, $sp, 28
        jr      $ra
```

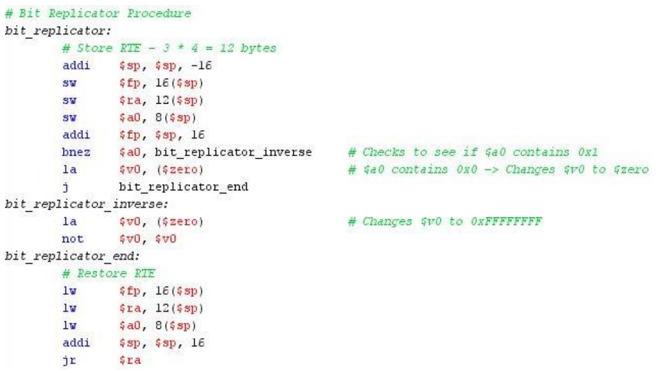
Twos_complement_64bit is used to convert a 64-bit value into 2's complement form. This procedure is primarily used by the logical multiplication as the product of the function will be in 64-bit format.

The procedure will take two arguments, the Lo part of the value ($a0) and the Hi part of the value ($a1). After the frame creation, both $a0 and $a1 will be inverted first to prepare for the conversion. $a0 will then be added with the value 1 (so ~$a0+1) using add_logical to produce the Lo part of the 2's complement 64-bit value.

After the calculation of the Lo part, the sum of ~$a0+1 will be saved to $s1 and the carry will be saved to $a0 since it will be needed to calculate the Hi part of the value's 2's complement 64-bit form. $a0 (the carry bit) will then be added to $a1 (Hi part of the value) using add_logical, and the sum of the operation will be saved to $v1.

Once the 2's complement of Hi and Lo are computed, $s1 (Lo) will be loaded back to $v0 for returning, and Hi is already loaded to $v1. The RTE frame is restored and the procedure ends.

d) bit_replicator

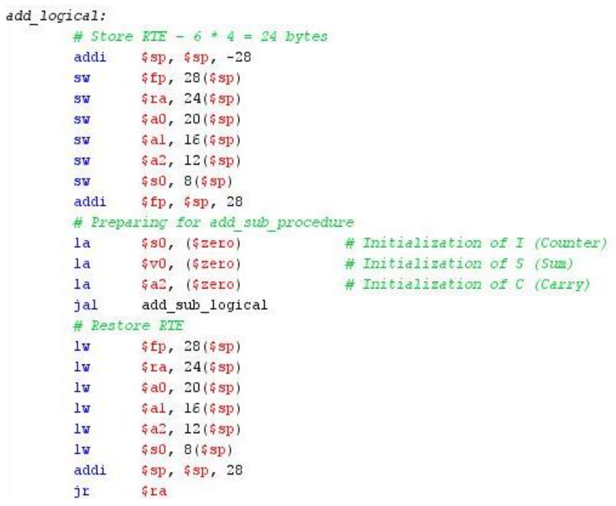


```
# Bit Replicator Procedure
bit_replicator:
        # Store RTE - 3 * 4 = 12 bytes
        addi    $sp, $sp, -16
        sw      $fp, 16($sp)
        sw      $ra, 12($sp)
        sw      $a0, 8($sp)
        addi    $fp, $sp, 16
        bnez    $a0, bit_replicator_inverse     # Checks to see if $a0 contains 0x1
        la      $v0, ($zero)                    # $a0 contains 0x0 -> Changes $v0 to $zero
        j       bit_replicator_end
bit_replicator_inverse:
        la      $v0, ($zero)                    # Changes $v0 to 0xFFFFFFFF
        not     $v0, $v0
bit_replicator_end:
        # Restore RTE
        lw      $fp, 16($sp)
        lw      $ra, 12($sp)
        lw      $a0, 8($sp)
        addi    $sp, $sp, 16
        jr      $ra
```

Bit replicator is a procedure used to replicate the desired bit ($a0) 32 times. The procedure first creates an RTE frame and checks to see if $a0 is 0 or 1. If $a0 is 0, then $v0 is loaded with all 0's and jumps to the end of the procedure. If $a0 is 1, then the procedure jumps to bit_replicator_inverse in which $v0 is loaded with 0's and then inverted. The RTE frame is then stored and the procedure ends.

3) Addition and Subtraction Procedures

a) add_logical



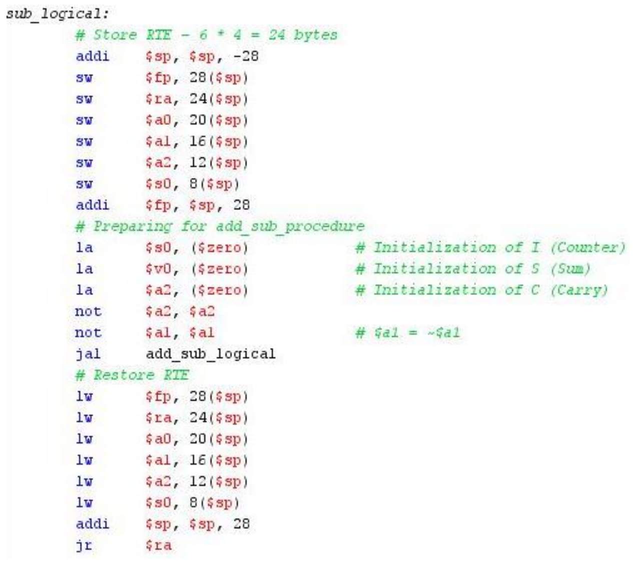
```
add_logical:
        # Store RTE - 6 * 4 = 24 bytes
        addi    $sp, $sp, -28
        sw      $fp, 28($sp)
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $a2, 12($sp)
        sw      $s0, 8($sp)
        addi    $fp, $sp, 28
        # Preparing for add_sub_procedure
        la      $s0, ($zero)            # Initialization of I (Counter)
        la      $v0, ($zero)            # Initialization of S (Sum)
        la      $a2, ($zero)            # Initialization of C (Carry)
        jal     add_sub_logical
        # Restore RTE
        lw      $fp, 28($sp)
        lw      $ra, 24($sp)
        lw      $a0, 20($sp)
        lw      $a1, 16($sp)
        lw      $a2, 12($sp)
        lw      $s0, 8($sp)
        addi    $sp, $sp, 28
        jr      $ra
```

Add_logical helps prepare for the addition of two values using logic operations. This procedure takes two arguments ($a0 and $a1) and returns the sum ($a0 + $a1) in $v0 and carry in $v1.

The RTE frame is first created and the preparation for the computation procedure (add_sub_logical) starts. During the preparation, all the variables needed for add_sub_logical is loaded as $zero. The counter (I) is saved to $s0, the sum (S) is saved to $s0, and the carry (C) is saved to $a2.

After preparations, the procedure jumps to add_sub_logical to add $a0 and $a1. When add_sub_logical returns to add_logical, the frame is restored and the sum ($v0) and carry ($v1) are returned.

b) sub_logical



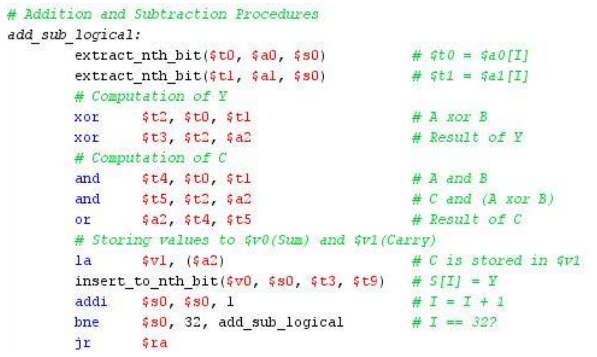
```
sub_logical:
        # Store RTE - 6 * 4 = 24 bytes
        addi    $sp, $sp, -28
        sw      $fp, 28($sp)
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $a2, 12($sp)
        sw      $s0, 8($sp)
        addi    $fp, $sp, 28
        # Preparing for add_sub_procedure
        la      $s0, ($zero)            # Initialization of I (Counter)
        la      $v0, ($zero)            # Initialization of S (Sum)
        la      $a2, ($zero)            # Initialization of C (Carry)
        not     $a2, $a2
        not     $a1, $a1                # $a1 = ~$a1
        jal     add_sub_logical
        # Restore RTE
        lw      $fp, 28($sp)
        lw      $ra, 24($sp)
        lw      $a0, 20($sp)
        lw      $a1, 16($sp)
        lw      $a2, 12($sp)
        lw      $s0, 8($sp)
        addi    $sp, $sp, 28
        jr      $ra
```

Similar to add_logical, sub_logical is to help prepare for the subtraction process done in add_sub_logical and returns the difference. The procedure takes two values ($a0 and $a1), subtract them using logic operations, and returns the difference ($a0 - $a1) in $v0 and carry in $v1.

After the frame creation, preparations for the subtraction process is done. The counter, difference, and carry are initialized (exactly like the initialization of the same variables in add_logical). The carry is then inverted since, in subtraction, the initial carry bit is 1. The argument $a1 is inverted as well since the subtraction process is essentially addition with a negative number.

Sub_logical will jump to add_sub_logical for the computation process after the preparation of variables. Add_sub_logical will return the difference ($v0) and carry ($v1) after it is done, and the RTE frame is restored afterward.

c) add_sub_logical



```
# Addition and Subtraction Procedures
add_sub_logical:
        extract_nth_bit($t0, $a0, $s0)          # $t0 = $a0[I]
        extract_nth_bit($t1, $a1, $s0)          # $t1 = $a1[I]
        # Computation of Y
        xor     $t2, $t0, $t1                   # A xor B
        xor     $t3, $t2, $a2                   # Result of Y
        # Computation of C
        and     $t4, $t0, $t1                   # A and B
        and     $t5, $t2, $a2                   # C and (A xor B)
        or      $a2, $t4, $t5                   # Result of C
        # Storing values to $v0(Sum) and $v1(Carry)
        la      $v1, ($a2)                      # C is stored in $v1
        insert_to_nth_bit($v0, $s0, $t3, $t9)   # S[I] = Y
        addi    $s0, $s0, 1                     # I = I + 1
        bne     $s0, 32, add_sub_logical        # I == 32?
        jr      $ra
```

Add_sub_logical is the computation procedure that produces the sum (for add_logical) or difference (for sub_logical) using a combination of logic operations. In this procedure, the arguments $a0 and $a1 are taken for computation (using the variables initialized in add_logical or sub_logical), and the sum/difference is returned in $v0 and carry in $v1.

First, the bit at a position indicated by the counter ($s0) is extracted from the bit patterns $a0 and $a1 and stored in $t0 and $t1 respectively. After extraction, add_sub_logical moves to the computation of the sum bit (Y).

The XOR operation is used between $t0 and $t1, and the result is stored in $t2. XOR is used again for $t4 and the carry ($a2), and the result is stored in $t3. $t3 will be the computation of Y.

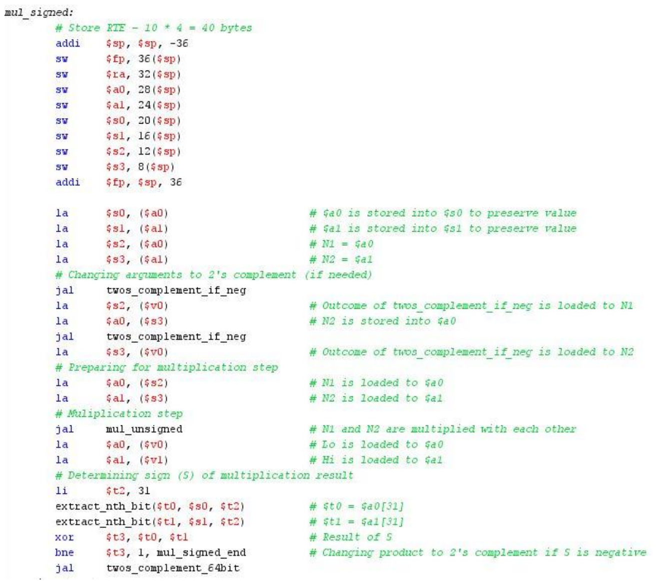
The carry bit (C) is then computed with the AND and OR operations. AND is first used between $t0 and $t1 and saved to $t4. Then, AND is used between $t2 and the carry bit ($a2) and stored in $t5. Lastly, the OR operation is used between $t4 and $t5 and saved to $a2.

After the computation of the sum and carry bits, they are stored in the intended registers. The carry bit will be simply loaded to $v1. For the sum bit, the macro insert_to_nth_bit is needed to insert the bit at the indicated position ($s0).

This process is repeated 32 times in total and returned to the procedure that called it.

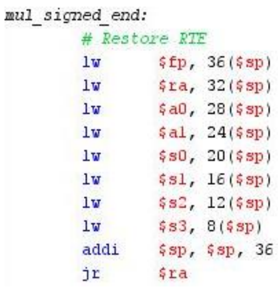
4) Multiplication Procedures

a) mul_signed



```
mul_signed:
        # Store RTE - 10 * 4 = 40 bytes
        addi    $sp, $sp, -36
        sw      $fp, 36($sp)
        sw      $ra, 32($sp)
        sw      $a0, 28($sp)
        sw      $a1, 24($sp)
        sw      $s0, 20($sp)
        sw      $s1, 16($sp)
        sw      $s2, 12($sp)
        sw      $s3, 8($sp)
        addi    $fp, $sp, 36

        la      $s0, ($a0)                      # $a0 is stored into $s0 to preserve value
        la      $s1, ($a1)                      # $a1 is stored into $s1 to preserve value
        la      $s2, ($a0)                      # N1 = $a0
        la      $s3, ($a1)                      # N2 = $a1
        # Changing arguments to 2's complement (if needed)
        jal     twos_complement_if_neg
        la      $s2, ($v0)                      # Outcome of twos_complement_if_neg is loaded to N1
        la      $a0, ($s3)                      # N2 is stored into $a0
        jal     twos_complement_if_neg
        la      $s3, ($v0)                      # Outcome of twos_complement_if_neg is loaded to N2
        # Preparing for multiplication step
        la      $a0, ($s2)                      # N1 is loaded to $a0
        la      $a1, ($s3)                      # N2 is loaded to $a1
        # Muliplication step
        jal     mul_unsigned                    # N1 and N2 are multiplied with each other
        la      $a0, ($v0)                      # Lo is loaded to $a0
        la      $a1, ($v1)                      # Hi is loaded to $a1
        # Determining sign (S) of multiplication result
        li      $t2, 31
        extract_nth_bit($t0, $s0, $t2)          # $t0 = $a0[31]
        extract_nth_bit($t1, $s1, $t2)          # $t1 = $a1[31]
        xor     $t3, $t0, $t1                   # Result of S
        bne     $t3, 1, mul_signed_end          # Changing product to 2's complement if S is negative
        jal     twos_complement_64bit
```



```
mul_signed_end:
        # Restore RTE
        lw      $fp, 36($sp)
        lw      $ra, 32($sp)
        lw      $a0, 28($sp)
        lw      $a1, 24($sp)
        lw      $s0, 20($sp)
        lw      $s1, 16($sp)
        lw      $s2, 12($sp)
        lw      $s3, 8($sp)
        addi    $sp, $sp, 36
        jr      $ra
```

Mul_signed is a procedure that uses logic operations to multiply two values together and produce the product. This procedure takes two arguments ($a0 and $a1) and returns the product in two parts. The Hi part of the product will be saved to $v1 and the Lo part will be saved to $v0. Mul_signed's main purpose is to ensure the arguments are in the correct form before the multiplication process and returning the product of the arguments.

After the frame creation, the arguments $a0 and $a1 are loaded to $s0 and $s1 respectively in order to preserve their values for later use. $a0 and $a1 are then saved to $s2 and $s3 for the multiplication step.

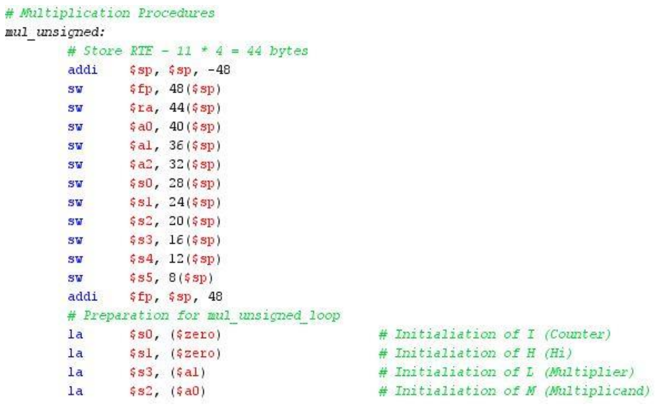
Before the multiplication process, twos_complement_if_neg is used to check if each argument needs to be converted to 2's complement form. The result of twos_complement_if_neg is stored in $s2 ($a0) and $s3 ($a1).

The values received from twos_complement_if_neg is then loaded to $a0 and $a1 before calling mul_unsigned to multiply the values together. Once mul_unsigned is completed, the Lo part of the product is saved to $a0 and the Hi part to $a1 for the sign determination.

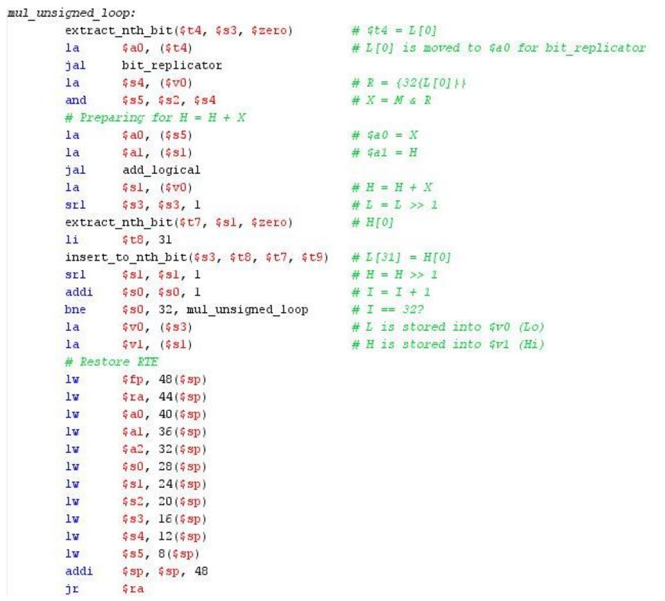
After the multiplication computation, the sign of the product must be determined. Using the macro extract_nth_bit to take the sign bit (or MSB) of each argument, both bits are XOR'ed between each other and the result is saved to $t3.

The sign is then compared to the value 1 to see if the product is negative or positive. If positive, mul_signed jumps to mul_signed_end for frame restoration and the returning of the product values. If negative, mul_signed jumps to twos_complment_64bit to make the product into 2's complement form before moving to mul_signed_end.

b) mul_unsigned



```
# Multiplication Procedures
mul_unsigned:
        # Store RTE - 11 * 4 = 44 bytes
        addi    $sp, $sp, -48
        sw      $fp, 48($sp)
        sw      $ra, 44($sp)
        sw      $a0, 40($sp)
        sw      $a1, 36($sp)
        sw      $a2, 32($sp)
        sw      $s0, 28($sp)
        sw      $s1, 24($sp)
        sw      $s2, 20($sp)
        sw      $s3, 16($sp)
        sw      $s4, 12($sp)
        sw      $s5, 8($sp)
        addi    $fp, $sp, 48
        # Preparation for mul_unsigned_loop
        la      $s0, ($zero)                    # Initialiation of I (Counter)
        la      $s1, ($zero)                    # Initialiation of H (Hi)
        la      $s3, ($a1)                      # Initialiation of L (Multiplier)
        la      $s2, ($a0)                      # Initialiation of M (Multiplicand)
```



```
mul_unsigned_loop:
        extract_nth_bit($t4, $s3, $zero)        # $t4 = L[0]
        la      $a0, ($t4)                      # L[0] is moved to $a0 for bit_replicator
        jal     bit_replicator
        la      $s4, ($v0)                      # R = {32{L[0]}}
        and     $s5, $s2, $s4                   # X = M & R
        # Preparing for H = H + X
        la      $a0, ($s5)                      # $a0 = X
        la      $a1, ($s1)                      # $a1 = H
        jal     add_logical
        la      $s1, ($v0)                      # H = H + X
        srl     $s3, $s3, 1                     # L = L >> 1
        extract_nth_bit($t7, $s1, $zero)        # H[0]
        li      $t8, 31
        insert_to_nth_bit($s3, $t8, $t7, $t9)   # L[31] = H[0]
        srl     $s1, $s1, 1                     # H = H >> 1
        addi    $s0, $s0, 1                     # I = I + 1
        bne     $s0, 32, mul_unsigned_loop      # I == 32?
        la      $v0, ($s3)                      # L is stored into $v0 (Lo)
        la      $v1, ($s1)                      # H is stored into $v1 (Hi)
        # Restore RTE
        lw      $fp, 48($sp)
        lw      $ra, 44($sp)
        lw      $a0, 40($sp)
        lw      $a1, 36($sp)
        lw      $a2, 32($sp)
        lw      $s0, 28($sp)
        lw      $s1, 24($sp)
        lw      $s2, 20($sp)
        lw      $s3, 16($sp)
        lw      $s4, 12($sp)
        lw      $s5, 8($sp)
        addi    $sp, $sp, 48
        jr      $ra
```

In mul_unsigned, the process of using logic operations to compute the product of two values is executed. The procedure will take two arguments ($a0 and $a1) and return the product in $v0 for the Lo part and $v1 for the Hi part.

Before the multiplication process (mul_unsigned_loop), the initialization of variables needed for the process is done. $zero is loaded to $a0 (counter, represented by I) and Hi $s1 (Hi, represented by H). The argument of $a0 is saved to the multiplier ($s3), while argument $a1 is saved to the multiplicand ($s2). After initialization, the multiplication process will take place.

In mul_unsigned_loop, the computation of the product is done. The LSB of the multiplier (L) is first extracted with extract_nth_bit and saved to $t4. $t4 will then be replicated 32 times using bit_replicator and saved to $s4 (R). The AND operation is then used between R and the multiplicand (M) and the result is saved to $s5 (X).

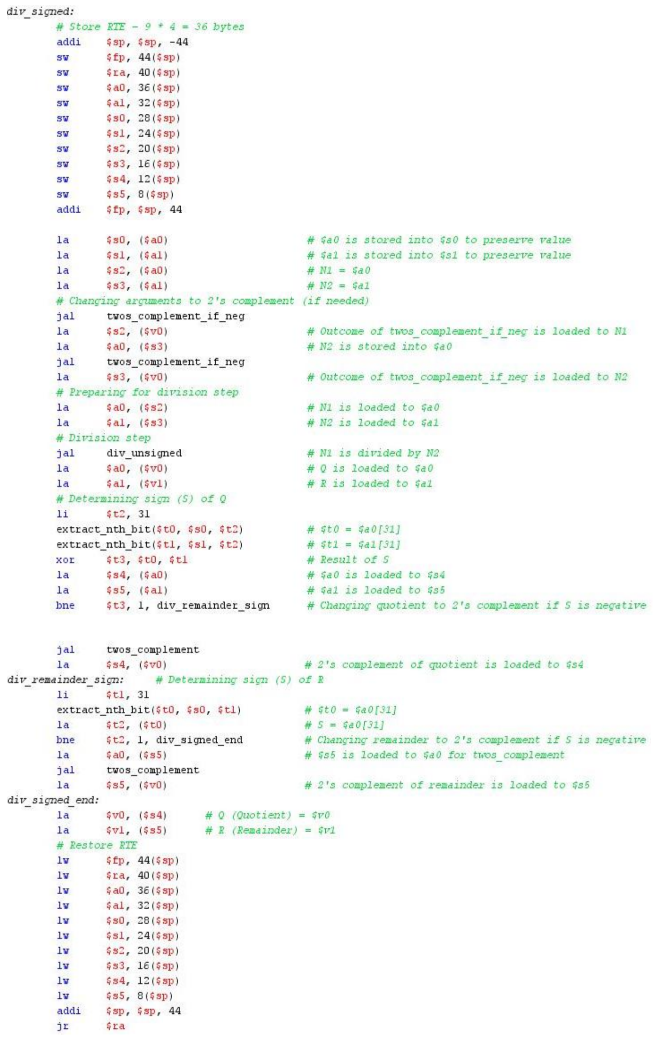
X and H are then saved to $a0 and $a1 to have add_logical take them and compute the sum of the two values. The result of this addition is then saved to H. L is then shifted to the right by 1.

Using extract_nth_bit, the LSB of H is then saved to $t7 and inserted to the MSB of L with insert_to_nth_bit. H is then shifted to the right by 1.

This process is repeated 32 times before saving the product into $v0 (for Lo) and $v1 (for Hi).

5) Division Procedures

a) div_signed



```
div_signed:
        # Store RTE - 9 * 4 = 36 bytes
        addi    $sp, $sp, -44
        sw      $fp, 44($sp)
        sw      $ra, 40($sp)
        sw      $a0, 36($sp)
        sw      $a1, 32($sp)
        sw      $s0, 28($sp)
        sw      $s1, 24($sp)
        sw      $s2, 20($sp)
        sw      $s3, 16($sp)
        sw      $s4, 12($sp)
        sw      $s5, 8($sp)
        addi    $fp, $sp, 44

        la      $s0, ($a0)                      # $a0 is stored into $s0 to preserve value
        la      $s1, ($a1)                      # $a1 is stored into $s1 to preserve value
        la      $s2, ($a0)                      # N1 = $a0
        la      $s3, ($a1)                      # N2 = $a1
        # Changing arguments to 2's complement (if needed)
        jal     twos_complement_if_neg
        la      $s2, ($v0)                      # Outcome of twos_complement_if_neg is loaded to N1
        la      $a0, ($s3)                      # N2 is stored into $a0
        jal     twos_complement_if_neg
        la      $s3, ($v0)                      # Outcome of twos_complement_if_neg is loaded to N2
        # Preparing for division step
        la      $a0, ($s2)                      # N1 is loaded to $a0
        la      $a1, ($s3)                      # N2 is loaded to $a1
        # Division step
        jal     div_unsigned                    # N1 is divided by N2
        la      $a0, ($v0)                      # Q is loaded to $a0
        la      $a1, ($v1)                      # R is loaded to $a1
        # Determining sign (S) of Q
        li      $t2, 31
        extract_nth_bit($t0, $s0, $t2)          # $t0 = $a0[31]
        extract_nth_bit($t1, $s1, $t2)          # $t1 = $a1[31]
        xor     $t3, $t0, $t1                   # Result of S
        la      $s4, ($a0)                      # $a0 is loaded to $s4
        la      $s5, ($a1)                      # $a1 is loaded to $s5
        bne     $t3, 1, div_remainder_sign      # Changing quotient to 2's complement if S is negative


        jal     twos_complement
        la      $s4, ($v0)                      # 2's complement of quotient is loaded to $s4
div_remainder_sign:     # Determining sign (S) of R
        li      $t1, 31
        extract_nth_bit($t0, $s0, $t1)          # $t0 = $a0[31]
        la      $t2, ($t0)                      # S = $a0[31]
        bne     $t2, 1, div_signed_end          # Changing remainder to 2's complement if S is negative
        la      $a0, ($s5)                      # $s5 is loaded to $a0 for twos_complement
        jal     twos_complement
        la      $s5, ($v0)                      # 2's complement of remainder is loaded to $s5
div_signed_end:
        la      $v0, ($s4)      # Q (Quotient) = $v0
        la      $v1, ($s5)      # R (Remainder) = $v1
        # Restore RTE
        lw      $fp, 44($sp)
        lw      $ra, 40($sp)
        lw      $a0, 36($sp)
        lw      $a1, 32($sp)
        lw      $s0, 28($sp)
        lw      $s1, 24($sp)
        lw      $s2, 20($sp)
        lw      $s3, 16($sp)
        lw      $s4, 12($sp)
        lw      $s5, 8($sp)
        addi    $sp, $sp, 44
        jr      $ra
```

The main purpose of div_signed is to prepare two values for the division process (div_unsigned) using logic operations and determine the signs of the result. Div_signed takes two arguments ($a0 and $a1) and returns the quotient in $v0 and the remainder in $v1.

Similar to mul_signed, $a0 and $a1 are saved to $s0 and $s1 respectively for value preservation and $s2 and $s3 respectively for the division process. Additionally, each value is checked with twos_complement_if_neg and converted to 2’s complement form if they are negative.

The values are then loaded to $a0 and $a1 for div_unsigned, and the quotient and remainder are returned in $v0 and $v1 respectively.

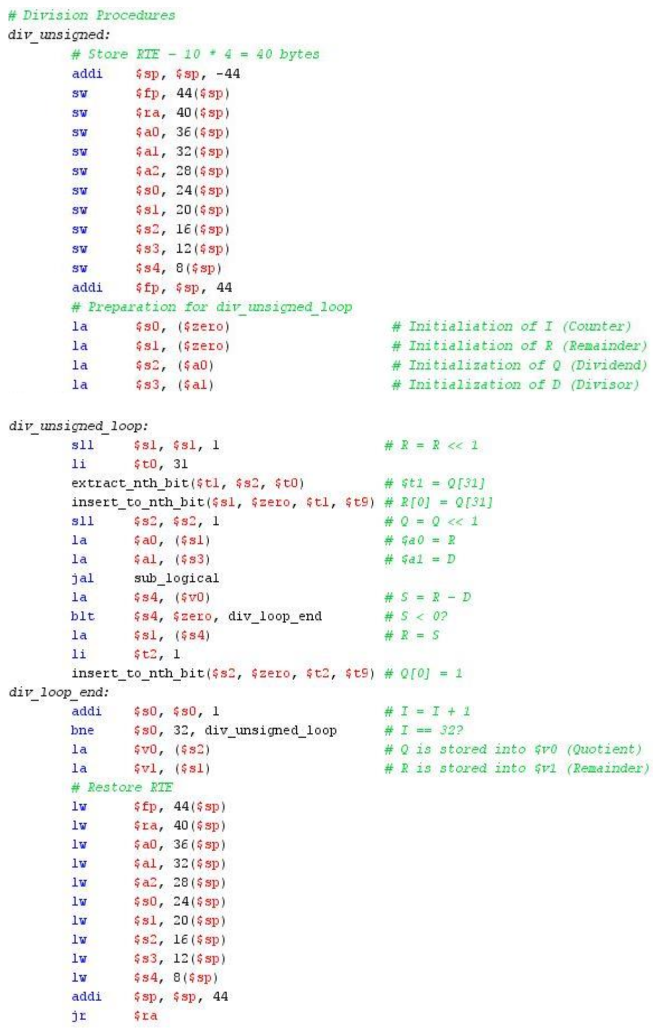
Once the quotient and remainder are computed, the signs for each one must be determined. For the quotient, the

MSB’s of the arguments are taken and the XOR operation is used between them to compute the sign S. Depending on the result of S, div_signed will jump to twos_complement if S is negative or move on to the sign determination of the remainder if S is positive.

For the determination of the remainder’s sign, the MSB of dividend is taken as S. S will then be converted to 2’s complement form by twos_complement if needed.

The finalized values for the quotient and remainder are saved to $v0 and $v1 respectively, and the RTE frame is restored.

b) div_unsigned



```
# Division Procedures
div_unsigned:
        # Store RTE - 10 * 4 = 40 bytes
        addi    $sp, $sp, -44
        sw      $fp, 44($sp)
        sw      $ra, 40($sp)
        sw      $a0, 36($sp)
        sw      $a1, 32($sp)
        sw      $a2, 28($sp)
        sw      $s0, 24($sp)
        sw      $s1, 20($sp)
        sw      $s2, 16($sp)
        sw      $s3, 12($sp)
        sw      $s4, 8($sp)
        addi    $fp, $sp, 44
        # Preparation for div_unsigned_loop
        la      $s0, ($zero)                     # Initialiation of I (Counter)
        la      $s1, ($zero)                     # Initialiation of R (Remainder)
        la      $s2, ($a0)                       # Initialization of Q (Dividend)
        la      $s3, ($a1)                       # Initialization of D (Divisor)

div_unsigned_loop:
        sll     $s1, $s1, 1                     # R = R << 1
        li      $t0, 31
        extract_nth_bit($t1, $s2, $t0)          # $t1 = Q[31]
        insert_to_nth_bit($s1, $zero, $t1, $t9) # R[0] = Q[31]
        sll     $s2, $s2, 1                     # Q = Q << 1
        la      $a0, ($s1)                      # $a0 = R
        la      $a1, ($s3)                      # $a1 = D
        jal     sub_logical
        la      $s4, ($v0)                      # S = R - D
        blt     $s4, $zero, div_loop_end        # S < 0?
        la      $s1, ($s4)                      # R = S
        li      $t2, 1
        insert_to_nth_bit($s2, $zero, $t2, $t9) # Q[0] = 1
div_loop_end:
        addi    $s0, $s0, 1                     # I = I + 1
        bne     $s0, 32, div_unsigned_loop      # I == 32?
        la      $v0, ($s2)                      # Q is stored into $v0 (Quotient)
        la      $v1, ($s1)                      # R is stored into $v1 (Remainder)
        # Restore RTE
        lw      $fp, 44($sp)
        lw      $ra, 40($sp)
        lw      $a0, 36($sp)
        lw      $a1, 32($sp)
        lw      $a2, 28($sp)
        lw      $s0, 24($sp)
        lw      $s1, 20($sp)
        lw      $s2, 16($sp)
        lw      $s3, 12($sp)
        lw      $s4, 8($sp)
        addi    $sp, $sp, 44
        jr      $ra
```

Div_unsigned is responsible for the division process of the logic calculator. This procedure takes two arguments ($a0 and $a1) and returns the quotient in $v0 and remainder in $v1.

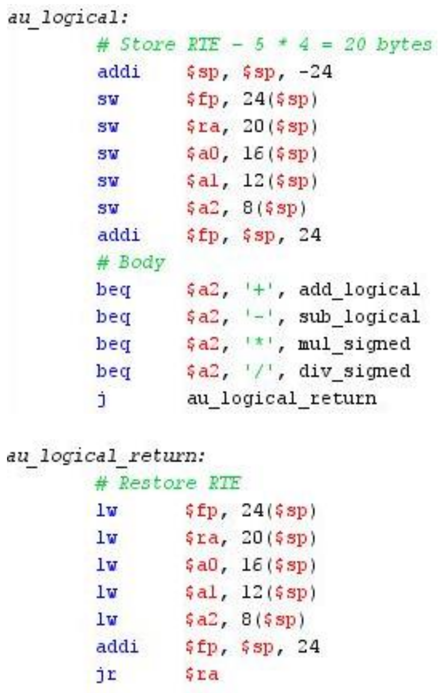
Once the RTE frame is created, the counter (I) and remainder (R) are initialized with $zero and saved to $s0 and $s1 respectively. The value of $a0 is then initialized to the dividend Q (in $s2) and the value of $a1 is saved to the divisor D (in $s3).

In the division process of div_unsigned_loop, R is shifted to the left by 1. Then, The MSB of Q is extracted with extract_nth_bit and inserted to the LSB of R with insert_to_nth_bit. Q is shifted to the left by 1 afterward.

R is loaded to $a0 and D is loaded to $a1 to use sub_logical and produce R – D. The difference between R and D will be saved to S ($s4). If S is greater than or equal to 0, then S will be the new remainder and the LSB of Q is set to 1. If not, div_unsigned_loop will move on to the next iteration.

Div_unsigned will iterate 32 times before saving the quotient to $v0 and remainder to $v1.

6) Logic Operations Procedure



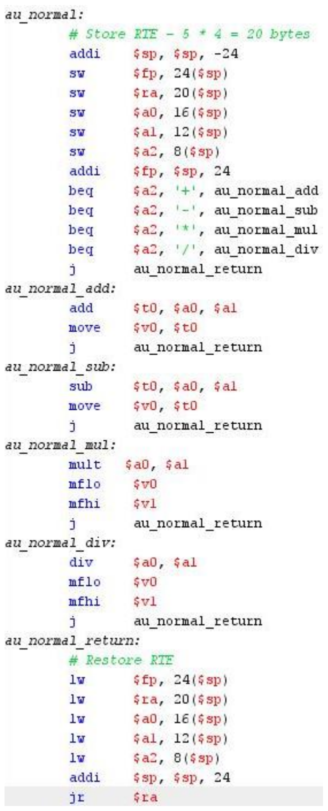
```
au_logical:
        # Store RTE - 5 * 4 = 20 bytes
        addi    $sp, $sp, -24
        sw      $fp, 24($sp)
        sw      $ra, 20($sp)
        sw      $a0, 16($sp)
        sw      $a1, 12($sp)
        sw      $a2, 8($sp)
        addi    $fp, $sp, 24
        # Body
        beq     $a2, '+', add_logical
        beq     $a2, '-', sub_logical
        beq     $a2, '*', mul_signed
        beq     $a2, '/', div_signed
        j       au_logical_return

au_logical_return:
        # Restore RTE
        lw      $fp, 24($sp)
        lw      $ra, 20($sp)
        lw      $a0, 16($sp)
        lw      $a1, 12($sp)
        lw      $a2, 8($sp)
        addi    $fp, $sp, 24
        jr      $ra
```

Au_logical is used to determine which mathematic operation is being called upon and then jumps to the indicated logic math operation. First, a frame for the run-time environment (RTE) is created. Then, the procedure compares the sign indicated by $a2 to the mathematic signs (+, –, *, /) and branches to the designated operation. If there are no matching signs (which should not happen at all), the procedure jumps to au_logical_return.

Au_logical_return is used for restoring the RTE after the logic math operations are completed.

7) Normal Operations Procedure



```
au_normal:
        # Store RTE - 5 * 4 = 20 bytes
        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, '+', au_normal_add
        beq     $a2, '-', au_normal_sub
        beq     $a2, '*', au_normal_mul
        beq     $a2, '/', au_normal_div
        j       au_normal_return
au_normal_add:
        add     $t0, $a0, $a1
        move    $v0, $t0
        j       au_normal_return
au_normal_sub:
        sub     $t0, $a0, $a1
        move    $v0, $t0
        j       au_normal_return
au_normal_mul:
        mult   $a0, $a1
        mflo    $v0
        mfhi    $v1
        j       au_normal_return
au_normal_div:
        div     $a0, $a1
        mflo    $v0
        mfhi    $v1
        j       au_normal_return
au_normal_return:
        # Restore RTE
        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
```

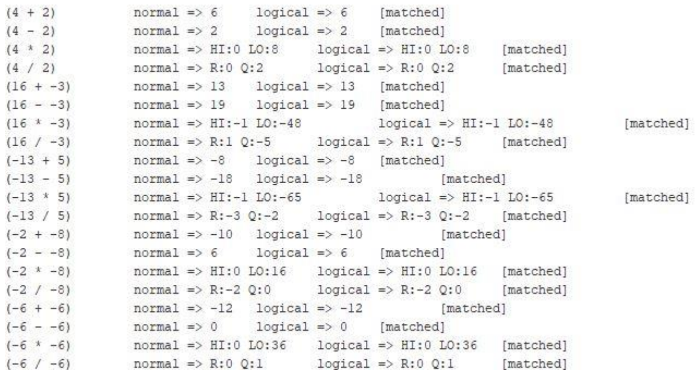
The normal mathematic operations are done in the au_normal procedure found in CS47_proj_alu_normal.asm. The main purpose of this procedure is to check the accuracy of the logic mathematic operations.

Similar to au_logical, a frame for the RTE is first created and then the sign of the operation ($a2) is compared to the mathematic operations. The procedure then branches to the designated normal math operation and the math operation is done. The result of the operation is then saved to the designated register ($v0 for addition and subtraction, $v0 and $v1 for multiplication and division). Once the result is saved, the RTE frame is restored and the procedure ends.

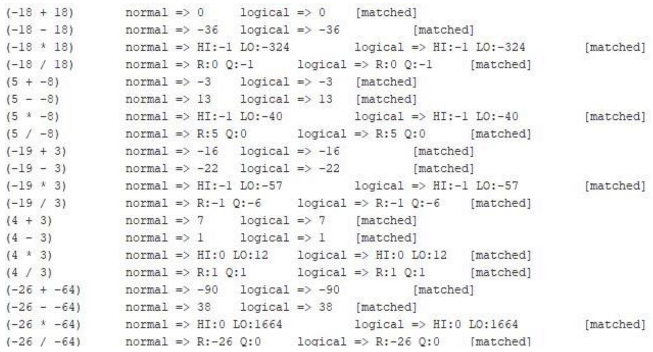
IV. TESTING

After all the logical implementations of a basic calculator have been completed, the implementation must be tested using CS47_proj_alu_normal.asm and the testing file proj_auto_test.asm. All of the files that have been modified must be first saved before assembling the program. Once assembled, the program can be run for testing.

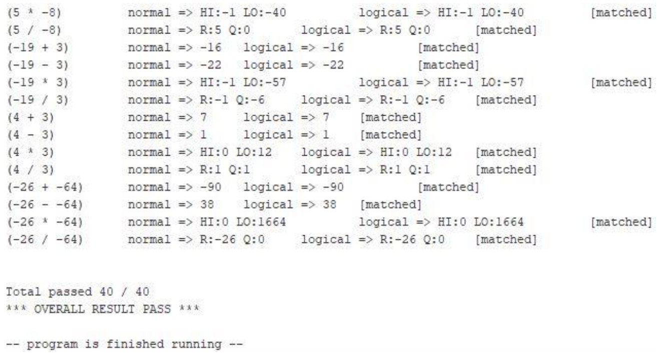
If the logical implementation of a basic calculator is done correctly, the program should return the following results:



```
(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]
```



```
(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 --
```

Proj_auto_test.asm works by taking test values from testV1Arr and testV2Arr (both in the file itself) and calling au_normal to compute the expected value of a specified math operation. The same math operation from au_logical would then be called and the result of it will be compared to the expected value. To pass the test, every result (40 total) from au_logical must match with its corresponding expected value.

V. CONCLUSION

In this project, the implementation of a calculator that computes basic math operations must be done with only logic operations. With the given ASM files and using MIPS simulator MARS, the implementation of this logic calculator can be achieved.

The next step that can be taken is to go through the program and see if any improvements can be made. Doing this will create a more efficient and effective program that can compute results in even less time.

Although some parts of this project proved to be difficult, I now have a better understanding of MIPS assembly language and how it can be used to create programs like the basic calculator. This project also helped me understand more about logic gates and the significance of them in computer hardware. Overall, I have learned about several aspects of assembly language and computer hardware, and I hope I can use MIPS for future programming projects.