Mathematical Arithmetic By MIPS Logic Operations

Joseph Nguyen
San Jose State University
College of Science, Computer Science
joseph.q.nguyen@sjsu.edu

***Abstract*—This report explores the implementation of mathematical arithmetic operations using normal arithmetic and logical. The program that will be portrayed is written in MIPS Assembly language using MARS.**

I. INTRODUCTION

The goal of this project is to use MARS MIPS Simulator in order to write and execute a program using MIPS Assembly language that can display the results of addition, subtraction, multiplication and division between integers. The results will first be displayed using normal operations which uses standard MIPS instructions such as add, sub, mul, and div. Following the normal results will be the results obtained using logical operations such as AND, NOT, OR and XOR. The logical operations will execute on a bit-by-bit basis.

II. GETTING STARTED

A. Installing MARS

MARS is an IDE that is used to write MIPS Assembly and assemble programs. It can be downloaded for free at http://courses.missouristate.edu/KenVollmar/mars/. Click on the download link for MARS 4.5 and follow the instructions to complete installation.

B. Setting up the Project

Once MARS is installed, download and unzip the given zip file, CS47Project1.zip, from the following canvas link, https://sjsu.instructure.com/courses/1255102/assignments/4598117. After opening the zip file, there should be the following asm files:

1) cs47_common_macro.asm
2) CS47_proj_alu_logical.asm
3) CS47_proj_alu_normal.asm
4) cs47_proj_macro.asm
5) cs47_proj_procs.asm
6) proj-auto-test.asm

Launch Mars4_5.jar, once the program is opened, go to the top left and click on File, then click Open. Locate the previously downloaded files and open all of them. We will modify CS47_proj_alu_logical.asm and CS47_proj_alu_normal.asm and cs47_proj_macro.acm. The other files will be used for testing. For the program to work we must also make sure that the following settings are checked: 'Assemble all files in directory' and 'Initialize program counter to global main if defined.' This can be done by clicking the Settings tab and checking the relevant boxes. We are now ready to program.

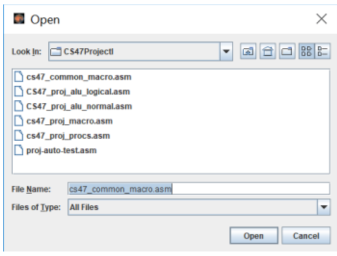

Fig. 1. Open window in MARS 4.5 with the relevent files

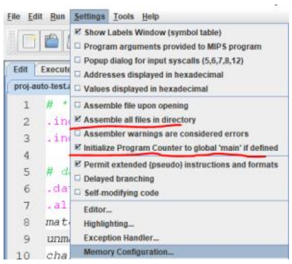

Fig. 2. Settings window in MARS 4.5 with relevent settings checked

III. EXPLANATION OF MAIN PROCEDURES

The goal of the program is to develop two main procedures, au_normal and au_logical, that will be called by the tester program.

A. au_normal

1) *Arguments:*
   a. $a0 – The first operand
   b. $a1 – The second operand
   c. $a3 – The operation symbol ('+','-','*','/')
2) *Return values (based on regular arithmetic):*
   a. $v0 – The results from operation (will contain LO for multiplication and quotient for division)
   b. $v1 – Other results (carry value for addition and subtraction, HI for multiplication and remainder for division)

B. au_logical

3) *Arguments:*
   a. $a0 – The first operand
   b. $a1 – The second operand

c. $a3 – The operation symbol ('+','-','*','/')

4) *Return values (based on logical arithmetic):*

a. $v0 – The results from operation (will contain LO for multiplication and quotient for division)

b. $v1 – Other results (carry value for addition and subtraction, HI for multiplication and remainder for division)

IV. Implementation

In order to help implement the procedures, there will be a number of utility macros and procedures that we will create and use.

A. Implementing au_normal

We will first create macros that will use standard MIPS instructions to compute the results. These macros will take two registers as arguments. The macro names in this case will be add_norm($r1,$r2), sub_norm($r1,$r2), mul_norm($r1,$r2), and div_norm($r1,$r2).

```
.macro add_norm($r1,$r2)
add $v0,$r1,$r2
.end_macro

.macro sub_norm($r1,$r2)
sub $v0,$r1,$r2
.end_macro

.macro mul_norm($r1,$r2)
mult $r1,$r2
mflo $v0
mfhi $v1
.end_macro

.macro div_norm($r1,$r2)
div $r1,$r2
mflo $v0
mfhi $v1
.end_macro
```

Fig. 3. Macros for normal arithmetic as seen in MARS 4.5

Once the macros are done, we need to create the frame for the au_normal procedure, since we are not modifying any argument and saved registers, we do not need to include them in the frame, thus, the only registers we are saving are $ra and $fp. Once we have our frame created, all we need to do is check what symbol $a2 contains and branch to the appropriate operations. For instance, if the symbol was '*', then mul_norm($a0,$a1) will be execute. Since we are done with the operation, and we know that the macros will return the valid results in the $v0 and $v1 registers, we can simply just jump to the end of the procedure.

```
au_normal:
        addi $sp, $sp, -12
        sw $fp, 12($sp)
        sw $ra, 8($sp)
        addi $fp, $sp, 12

        beq $a2, '+', add
        beq $a2, '-', sub
        beq $a2, '*', mul
        beq $a2, '/', div
add:
        add_norm($a0,$a1)
        j au_normal_end
sub:
        sub_norm($a0,$a1)
        j au_normal_end
mul:
        mul_norm($a0,$a1)
        j au_normal_end
div:
        div_norm($a0,$a1)
        j au_normal_end

au_normal_end:
        lw $fp, 12($sp)
        lw $ra, 8($sp)
        addi $sp, $sp, 12

        jr      $ra
```

Fig. 4. Implementation of au_normal

B. Implementing au_logical

To implement au_logical we need create some utility procedures and macros in order to help us create our main procedures. The main procedures that will will create are add_logical, sub_logical, mul_signed and div_signed.

1) add_logical

To implement add_logical, we must first understand how the logical algorithm for addition works. We have all learned from elementary school how to add using the carry method. We start from the least significant bit on the right and add the numbers digit by digit. If the sum is 10, then we carry that 1 over to the next digit and repeat. The addition algorithm that we will be using is basically that, but in binary.

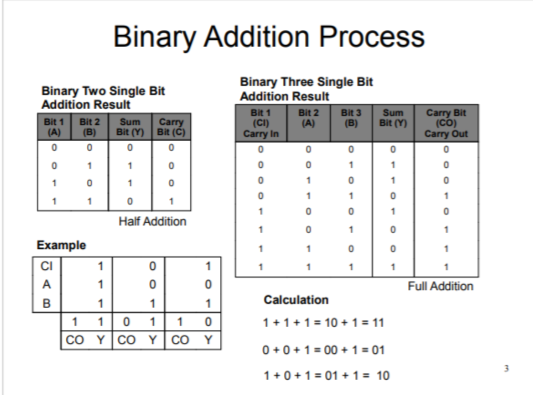

Fig. 5. Truth table of binary addition with inputs Carry In, bit2, bit3 and outputs Carry Out and Sum

In order to implement this algorithm using logic gates, we must use the truth table to get a Boolean equation for Sum and Carry Out.

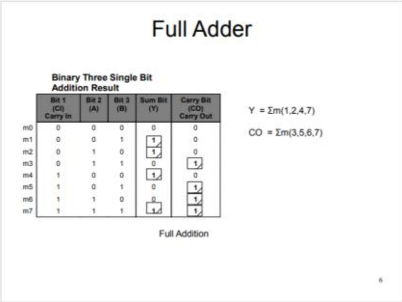

Fig. 6. Sum and Carry Out are written in terms of their sum of minterms

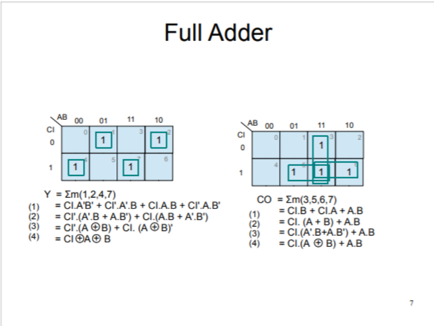

Fig. 7. Using a K-Map, and with some boolean identities, Sum and Carry Out is written out as compact boolean equations

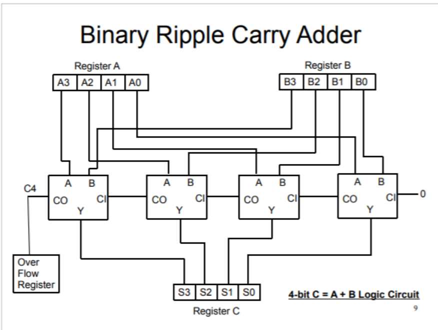

Fig. 8. Diagram of binary addition algorithm

Now lets create some macros to help us do the operations. First, we need to find a way to access a specific bit given an index in a bit pattern. This is where the macro extract_nth_bit comes in to play. It takes arguments $return, $pattern and $index. To extract the bit at $index, we use a masking technique in which we shift the pattern to the right by $index and then do an AND operation on the new bit pattern with a bit pattern 1.

```
.macro extract_nth_bit($return, $pattern, $index)
srlv $t0,$pattern,$index
andi $t0,$t0,1
add $return,$zero,$t0
.end_macro
```

Fig. 9. Implementation of extract_nth_bit macro

We also need a macro to insert the bit 1 into a specified index of a bit pattern. This macro allows us to reconstruct bit patterns such as the resulting sum of two other bit patterns. The macro is insert_to_nth_bit($pattern,$num,$index) and it takes bit pattern $num, which can be 0 or 1 and shifts it left by $index amount and then uses an OR operation with the bit pattern to get the resulting bit with the inserted number.

```
.macro insert_to_nth_bit($pattern,$num,$index)
add $t0,$zero,$num
sllv $t0,$t0,$index
or $pattern,$pattern,$t0
.end_macro
```

Fig. 10. Implementation of insert_to_nth_bit macro

Now that we have our macros, we can start writing the sub procedure add_logical. We will be using three saved registers. $s0 for index of iteration, $s1 for carry bit, and $s3 for the sum bit pattern. We will be looping throught the pattern 32 times since that is how many bits a word contains. All three registers will be initialized to 0. While $s0 < 32, compute the outputs, Carry Out and Sum, using the boolean equations and reassign $s1 and $s2 to the new values. Increment $s0 at the end of itereation. Once loops is complete, move $s2 to $v0 since it's the result. $v1 will contain the carry bit so move $s1 to $v1. The addition procedure is complete.

```
add_logical:
        addi $sp, $sp, -24
        sw $fp, 24($sp)
        sw $ra, 20($sp)
        sw $s0, 16($sp)
        sw $s1, 12($sp)
        sw $s2, 8($sp)
        addi $fp, $sp, 24

        add $s0,$zero,$zero # counter for bit index
        add $s1,$zero,$zero # Carry value starts at 0
        add $s2,$zero,$zero # bit pattern for sum
add_loop:
        beq $s0,32,add_logical_end
        extract_nth_bit($t1, $a0, $s0)
        extract_nth_bit($t2, $a1, $s0)

        xor $t3,$s1,$t1 # CarryIn xor t1
        xor $t3,$t3,$t2 # (CarryIn xor t1) xor t2, t3 = sum bit

        xor $t4,$t1,$t2
        and $t5,$s1,$t4
        and $t6,$t1,$t2
        or $s1,$t5,$t6  # s1 = CO = CI.(t1 xor t2) + t1.t2

        insert_to_nth_bit($s2,$t3,$s0)
        addi $s0,$s0,1
        j add_loop

add_logical_end:
        add $v0,$zero,$s2       # return sum
        add $v1,$zero,$s1       # return carry
```

Fig. 11. Implementation of add_logical (excluding frame restoration)

2) sub_logical

To implement sub_logical, we simply just convert the second operand and call add_logical on the first operand and the new second operand. Essentially, what we are doing is just converting A-B to A+(-B). The inverse of B can be obtained by using a NOT operation on the bit pattern and then adding 1 to it. We will create a utility procedure for inverting a number called twos_complement. It takes arguments $a0 and inverts it and stores the result in $v0

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

        not $a0,$a0                #INV($a0)
        addi $a1,$zero,1

        jal add_logical

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

Fig. 12. Implementation of two_complement procedure

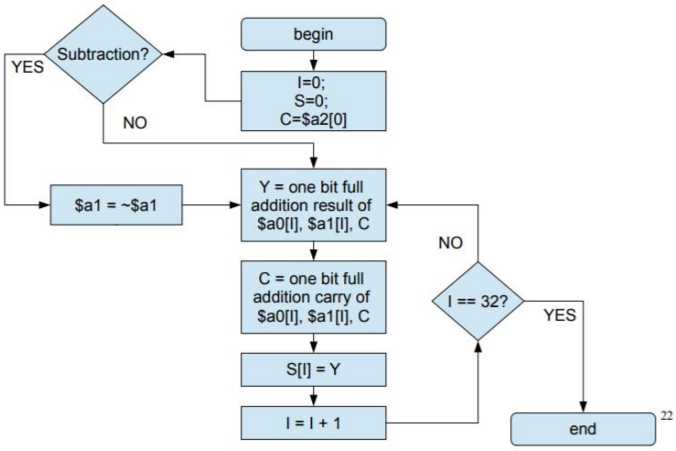

Fig. 13. This flow chart shows us the relationship between add_logical and sub_logical. sub_logical is simply just a modified add_logical

```
sub_logical:
        addi $sp,$sp, -24
        sw $fp, 24($sp)
        sw $ra, 20($sp)
        sw $a1, 16($sp)
        sw $a0, 12($sp)
        sw $s1, 8($sp)
        addi $fp, $sp, 24

        move $s1,$a0              # Store first operand
        move $a0,$a1
        jal twos_complement       # Invert second operand
        move $a1,$v0
        move $a0,$s1
        jal add_logical           # Add them together

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

Fig. 14. Implementation of sub_logical

3) mul_signed

To implement mul_signed, we must first create a mul_unsigned procedure. The algorithm for unsigned binary multiplication is very similar to elementary multiplication. Note that the multiplication of binary bits is equivalent to the AND operation between them.

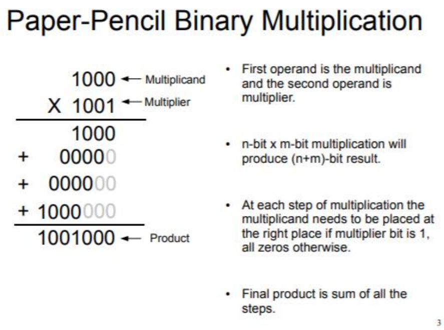

Fig. 15. Elementary Paper-Penci Binary Multiplication

During the muplication algorithm, a single multiplier bit is multiplied with the rest of the bits of the multiplicand. This is essentailly the replication of the multiplier bit and the AND operation between the replication and multiplicand. We can create a macro that returns a bit pattern 0x00000000 if the bit is 0 and 0xFFFFFFFF if it is 1.

```
.macro bit_replicator($return,$num)
beq $num,0,return_zero
addi $return,$zero,0xFFFFFFFF
j end
return_zero:
addi $return,$zero,0x0
end:
.end_macro
```

Fig. 16. Implemenation of bit_replicator macro

Since the multiplication algorithm produces a 64-bit product, we must represent the product as two 32-bit HI and LO registers. We set HI as 0 and LO as the multiplier. We then set HI = HI + X where X is the AND operation between the replicated multiplier bit and the multiplicand. We then shift LO to the right by one and set its left most bit to the first bit of HI. We then shift HI to the right by one. What we are simulating here is shifting a 64-bit pattern. We will do this process 32 times to get our product.

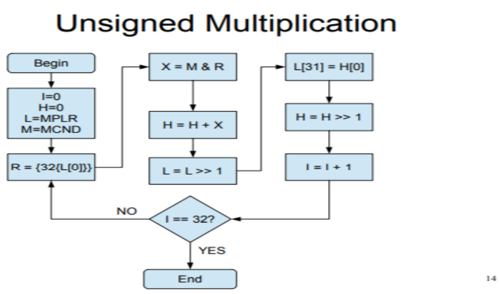

Fig. 17. Flow chart for unsigned multiplication

For signed multiplication, create a procedure called twos_complement_if_neg. This procedures takes a number a gives its complement if it is negative. $v0 will return the absolute value of the number.

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

        addi $t0,$zero,31
        add $v0,$zero,$a0
        extract_nth_bit($t0,$a0,$t0)
        beq $t0,0,twos_complement_if_neg_end
        jal twos_complement

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

Fig. 18. Implementation of twos_complement_if_neg

Create another procedure, twos_complement_64bit. This procedure will take two arguments, a LO bit pattern and a HI bit pattern. First invert both numbers and add 1 to the inverted LO. If there is a carry overflow then add 1 to the inverted HI. What is being simulated here is the complement of a 64 bit number.

```
twos_complement_64bit:
        addi $sp, $sp, -28
        sw $fp, 28($sp)
        sw $ra, 24($sp)
        sw $s1, 20($sp)
        sw $s0, 16($sp)
        sw $a1, 12($sp)
        sw $a0, 8($sp)
        addi $fp, $sp, 28

        not $s0,$a0             # Invert $a0
        not $s1,$a1             # Invert $a1

        add $a0,$zero,$s0
        addi $a1,$zero,1
        jal add_logical         # Add 1 to ~$a0
        add $a0,$zero,$s1
        add $a1 $zero,$v1
        add $s1,$zero,$v0
        jal add_logical         # Add the Carry Out from previous calculation to ~$a1
        add $v1,$zero,$v0
        add $v0,$zero,$s1       # Return the values
```

Fig. 19. Implementation of twos_complement_64bit (excluding frame restoration)

Now implement mul_signed. This procedure will take two numbers and convert them to positive using twos_complement_if_neg. The product of these positive numbers is computed using mul_unsigned. The signs of the original numbers will be considered. To determine the sign of the numbers, use extract_nth_bit on index 31 for both numbers. If they are both the same, then the product is positive. If they are different, then the result must be inverted to negative using twos_complement_64bit. Note that determining the signs of the product is equivalent to doing an XOR operation.

```
add $s2,$zero,$a0                  # Store first number
add $s3,$zero,$a1                  # Store second number

jal twos_complement_if_neg         # Absolute value of first number
add $s0,$zero,$v0
add $a0,$zero,$s3
jal twos_complement_if_neg         # Absolute value of second number
add $s1,$zero,$v0

add $a0,$zero,$s0
add $a1,$zero,$s1
jal mul_unsigned                   # Positive product

addi $t0,$zero,31
extract_nth_bit($t1,$s2,$t0)
addi $t0,$zero,31
extract_nth_bit($t2,$s3,$t0)
xor $t0,$t1,$t2                    # Determine sign of product 0 if postive, 1 if negative

beq $t0,0,mul_signed_end           # If postive, done, else compute 64bit complement
add $a0,$zero,$v0
add $a1,$zero,$v1
jal twos_complement_64bit
```

Fig. 20. Implementation of mul_signed (excluding frame)

4) div_signed

To implement div_signed, we must first create a div_unsigned procedure. The algorithm for unsigned binary division is very similar to paper-pencil division.

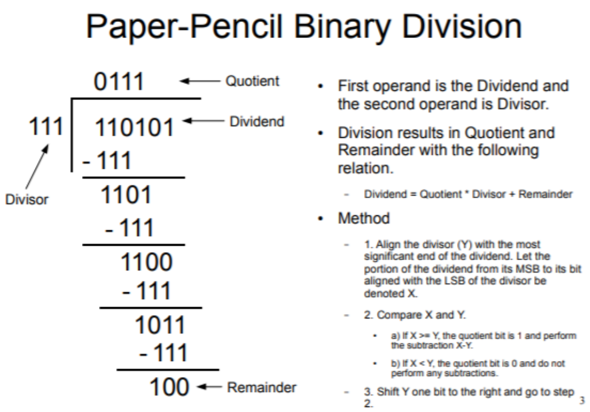

Fig. 21. Paper-pencil binary division

The binary division is also a 64-bit operation with one register being the quotient and the other being the remainder. First start by initializing i to 0, Q as dividend, D as divisor and R as 0. For the start of the iteration, R is left shifted by one and R[0] is equal to Q[31], then Q is left shifted by one. Let S = R – D. If S is less than 0 then just increment i, otherwise, R = S and Q[0] = 1. Repeat this 32 times.

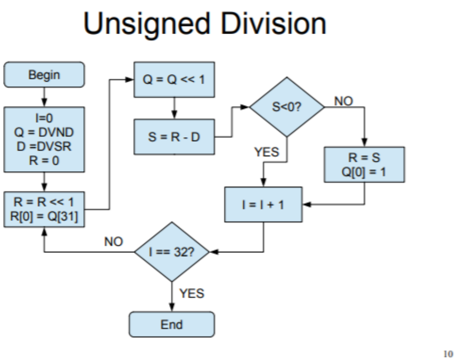

Fig. 22. Flowchart for unsigned division

Following the flowchart, div_unsigned is easily implemented.

```
        addi $s4,$zero,0        # i = 0
        add $s5,$zero,$a0       # Q = Dividend
        add $s6,$zero,$a1       # D = Divisor
        addi $s7,$zero,0        # R = remainder starts at zero

div_loop:
        beq $s4,32,div_unsigned_end         # Check i==32
        sll $s7,$s7,1                       # R << 1
        addi $t0,$zero,31
        extract_nth_bit($t0,$s5,$t0)        # Q[31]
        addi $t1,$zero,0
        insert_to_nth_bit($s7,$t0,$t1)      # R[0] = Q[31]
        sll $s5,$s5,1                       # Q << 1
        move $a0,$s7
        move $a1,$s6
        jal sub_logical                     # S = R - D
        blt $v0, 0, increment_i             # if (s < 0) then branch
        move $s7,$v0                        # R = S
        addi $t2,$zero,0
        addi $t3,$zero,1
        insert_to_nth_bit($s5,$t3,$t2)      # Q[0] = 1
increment_i:
        addi $s4,$s4,1
        j div_loop
```

Fig. 23. Implementation of div_unsigned (Excluding frame restoration)

Once we have div_unsigned, we can implement div_signed. The operands are converted to absolute value. Then the unsigned division between them is computed. To determine sign of quotient, compute the XOR value of original operands. If both are the same, then just determine sign for remainder. If both are different, then invert the quotient so that is has the correct sign and then determine sign of remainder. For the remainder check the sign of the first operand. If it is negative, invert the remainder, otherwise leave it.

```
        add $s2,$zero,$a0               # Store $a0
        add $s3,$zero,$a1               # Store $a1

        jal twos_complement_if_neg      # |$a0|
        add $s0,$zero,$v0
        add $a0,$zero,$s3
        jal twos_complement_if_neg      # |$a1|
        add $s1,$zero,$v0
        add $a0,$zero,$s0
        add $a1,$zero,$s1
        jal div_unsigned                # |$a0/$a1|
        move $s0,$v0                    # Store quotient
        move $s1,$v1                    # Store remainder
        addi $t0,$zero,31
        extract_nth_bit($t1,$s2,$t0)
        addi $t0,$zero,31
        extract_nth_bit($t2,$s3,$t0)
        xor $t0,$t1,$t2
        beq $t0,0,sign_of_R             # If signs of $a0 and $a1 are not the same, invert the quotient
        move $a0,$s0
        jal twos_complement
        move $s0,$v0
sign_of_R:
        addi $t0,$zero,31
        extract_nth_bit($t1,$s2,$t0)
        beq $t1,0,div_signed_end        # If sign of $a0 is negative, invert it
        move $a0,$s1
        jal twos_complement
        move $s1,$v0
div_signed_end:
        move $v0,$s0
        move $v1,$s1
```

Fig. 24. Implementation of div_signed (excluding frame)

5) au_logical

All main procedures have been completed. Now finish au_logical by calling the approriate procedures based on symbol.

```
au_logical:
        addi $sp, $sp, -12
        sw $fp, 12($sp)
        sw $ra, 8($sp)
        addi $fp, $sp, 12

        beq $a2, '+', add
        beq $a2, '-', sub
        beq $a2, '*', mul
        beq $a2, '/', div
add:
        jal add_logical
        j au_logical_end
sub:
        jal sub_logical
        j au_logical_end
mul:
        jal mul_signed
        j au_logical_end
div:
        jal div_signed
        j au_logical_end

au_logical_end:
        lw $fp, 12($sp)
        lw $ra, 8($sp)
        addi $sp, $sp, 12
        jr      $ra
```

Fig. 25. Implementation of au_logical

V. Testing

Now that both au_normal and au_logical have been completed, assemble all files and make sure that there are no errors. Then click on proj-auto-test.asm and run it. 40 arithmetic operations should be displayed. The program should run compute almost instantly, if not then restart MARS or computer. If done properly, the message should say:

Total passed 40 / 40

*** OVERALL RESULT PASS ***

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


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

-- program is finished running --
```

Fig. 26. The successful results should display as above

VI. CONCLUSION

This project was a very tough one to do. I spent so much time debugging and thinking that I got really stressed out. However; this project has taught me the fundamentals of basic arithmetic. I have always wondered how calculators are made, and how programming languages support basic arithmetic operations. I never would have thought that it all would all boil down to basic logic gates and shifting. Actually writing the code was very tiring, so I have learned to really appreciate these algorithms as they are so useful for computing. I now have a better understand about how basic calculations work, that even though it calculations may be simple; the computer does a lot of tedious work just to get it done. This project has given me a lot of insight as a programmer.