Implementation of a Basic Calculator through MIPS Logic Operations

Johnson Bao
San Jose State University
johnson.bao@sjsu.edu

***Abstract*—This paper will explain the implementation of the basic operations from a calculator; addition, subtraction, multiplication, and division. These operations will be implemented using both normal and MIPS logical procedures using MARS (MIPS Assembler and Runtime Simualtor).**

I. Introduction

Using a MARS 4.5 as our IDE, we can use MIPS operations (add, sub, mult, and div) to execute our mathematical operations. We can also use logical operations to have the same outcome as the provided MIPS operations by using Boolean logic such as AND, OR, and NOT. This report will show the implementation/design of these logical procedures, as well as looking at test cases to ensure they work correctly.

II. Installation and Setup

A. MARS 4.5 Download

MARS 4.5 can be downloaded from Missouri State University for free: http://courses.missouristate.edu/KenVollmar/MARS/. This will be our interactive development environment (IDE) throughout this project to create a calculator with basic mathematical operations.

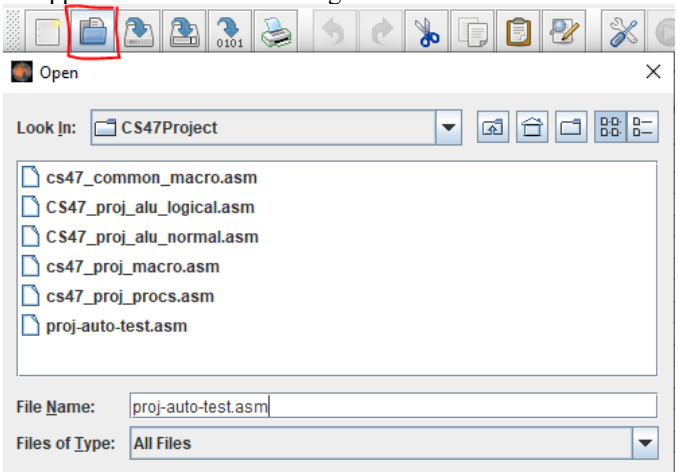
B. Downloading necessary files

Access the given zip file and extract the files in the folder. https://sjsu.instructure.com/courses/1474044/assignments/6025304. There will be six files that are included in this zip file:

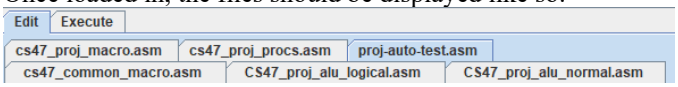
1) cs47_common_macro.asm
2) cs47_proj_procs.asm
3) proj-auto-test.asm
*These three files above should not be edited.
4) cs47_proj_macro.asm
5) cs47_proj_alu_logical.asm
6) cs47_proj_alu_normal.asm

C. Loading files into MARS IDE

Once MARS and the necessary files have been downloaded, open the Mars4_5.jar file. Locate the file opener on the top left of the simulator and find the directory of where the files were unzipped. Load in all the files given.

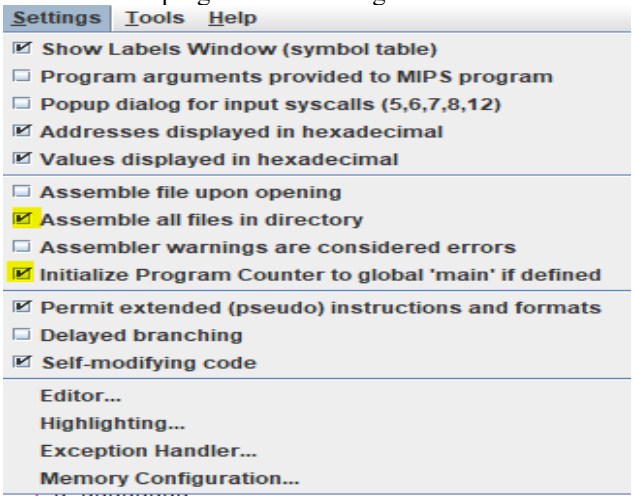


Once loaded in, the files should be displayed like so:



D. Settings used in the MIPS simulator

Go to the top of the simulator and locate the "Settings" tab. Click the tab and a dropdown menu will appear. Make sure to check the highlighted boxes: "Assembles all files in directory" and "Initialize program counter to global main if defined".



*The other checked boxes are optional, but useful.

III. OTHER REQUIREMENTS

A. Boolean Algebra / Boolean Logic

To implement these math operations using logic, one must understand the idea of Boolean Logic. In the world of circuitry, there are only two numbers that are represented, 0 and 1. In this case, 0 will stand for false while 1 will stand for true. There are different Boolean expressions that are used and each one has a different output depending on the operation being used.

Table 1 shows the truth table for the OR operation. The OR operation will return 0 if both inputs are 0. If either input is 1, the output will become 1.

Table 1: Truth Table for OR (+)

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

Table 2 shows the truth table for the AND operation. The AND operation will return 1 if and only if both inputs are 1. If either output contains a 0, the output will become 0.

Table 1: Truth Table for AND (.)

A	B	A . B
0	0	0
0	1	0
1	0	0
1	1	1

Table 3 shows the truth table for the XOR operation. The XOR operation will return 1 if and only if the inputs are not equal. Having the inputs be the same will result in the output returning 0.

Table 3: Truth Table for XOR ($\oplus$)

A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0

B. Binary Number System

Unlike our normal decimal system that uses base 10, the binary system uses base 2. Because the base of this number system is two, the symbols used to represent the numbers are 0 and 1. With the decimal number system, our numbers "roll over" to the next digit's place after it reaches 10. For example, the digit after 9 will roll over to 10, 19 to 20, 99 to 100. However with the binary number system, our numbers "roll over" to the next digit's place after it reaches 2. An example would be 1 to 10, as the binary system can only hold 0 and 1. The difference between number systems can be denoted by the subtext of the base that follows a number. For example, the number 10 in decimal would be 10_{10}, while it would be 1010_2 in binary. It is important to know the difference between what base is being used so that we do not get confused with the numbers.

To handle negative numbers in binary, we must use "two's complement", where if the MSB (Most Significant Bit) represents if the number is negative or positive. If the MSB is 0, then the number is positive. But if the MSB is 1, then the number is negative. For example, the number 3_{10} in two's complement would be 0011_2, while -3_{10} would equal 1101. The way to find the two's complement form of a negative number is to inverse all of the positive number's bits, and then add 1. Using the previous example, our positive number 3_{10} is 0011_2. We would inverse all these bits, making it 1100_2, then add 1 to get the negative number (1101_2)

IV. DESIGN AND IMPLEMENTATION

This section will talk about the design and implementation of both the normal and logical procedures to perform the mathematical operations.

A. Normal Procedure/Implementation

In MIPS, there are operations that are created for us to use to perform these calculations. These operations include; add, sub, mult, and div. We start by checking the sign to ensure that we are using the correct mathematical operation.

1) Addition

If the $a2 register's value equals to $t0 (set to '+'), then the procedure will jump to the addition section.

2) Subtraction

If the $a2 register's value equals to $t0 (set to '-'), then the procedure will jump to the addition section.

3) Multiplication

If the $a2 register's value equals to $t0 (set to '*'), then the procedure will jump to the multiplication section. However, this will have the answer be split into two different segments. The Hi register will move into the $v1 register, while Lo will move into the $v0 register.

4) Division

If the $a2 register's value equals to $t0 (set to '/'), then the procedure will jump to the division section. However, this will have the answer be split into two different segments. The Hi register will move into the $v1 register, while Lo will move into the $v0 register.

```
au_normal:
        store_all_frame
        li      $t0, '+'
        beq     $t0, $a2, addition_normal
        li      $t0, '-'
        beq     $t0, $a2, subtraction_normal
        li      $t0, '*'
        beq     $t0, $a2, multiplication_normal
        li      $t0, '/'
        beq     $t0, $a2, division_normal

addition_normal:
        add     $v0, $a0, $a1
        j       exit

subtraction_normal:
        sub     $v0, $a0, $a1
        j       exit

multiplication_normal:
        mult    $a0, $a1
        mflo    $v0
        mfhi    $v1
        j       exit

division_normal:
        div     $a0, $a1
        mflo    $v0
        mfhi    $v1
        j       exit

exit:
        restore_all_frame
        jr      $ra
```

B. Logical Procedure/Implementation

Similar to the normal implementation, we will start by checking the sign.

```
au_logical:
        store_all_frame

        li      $t0, '+'
        beq     $t0, $a2, addition_logical
        li      $t0, '-'
        beq     $t0, $a2, subtraction_logical
        li      $t0, '*'
        beq     $t0, $a2, multiplication_logical
        li      $t0, '/'
        beq     $t0, $a2, division_logical
        j       exit

addition_logical:
        jal     add_logical
        j       exit

subtraction_logical:
        jal     sub_logical
        j       exit

multiplication_logical:
        jal     mul_signed
        j       exit

division_logical:
        jal     signed_div
        j       exit

exit:
        restore_all_frame
        jr      $ra
```

But before we continue, we need to implement some macros and procedures to complete our task.

1) Utility Macros

Some macros that we will need to create will be useful in implementation:

i. extract_nth_bit

This macro gets the bit value of any given position for a given number.

```
.macro extract_nth_bit($regD, $regS, $regT)
srlv    $regS, $regS, $regT         # Shift right
li      $regD, 1                    # regD = 1
and     $regD, $regS, $regD         # Check the right most bit of $regS
.end_macro
```

Extract_nth_bit takes 3 different arguments, $regD, $regS, and $regT. $regD stores the result, $regS stores the source bit pattern, and $regT holds the bit position value. The macro works by shifting the source bit pattern to the right by the bit position number given. This allows for the

bit we want to extract to become the LSB. Afterwards, we call the AND operation with 1 to extract the bit we want and store it.

ii. insert_to_nth_bit

This macro will load a given bit into the nth bit position of a value.

```
.macro insert_to_nth_bit($regD, $regS, $regT, $maskReg)
move    $maskReg, $regT             # Move 1 in regT into maskReg
sllv    $maskReg, $maskReg, $regS   # Shift left
or      $regD, $regD, $maskReg      # regD = regD or maskReg
.end_macro
```

Insert_to_nth_bit takes in 4 arguments, $regD, $regS, $regT, and $maskReg. $regD is the bit pattern we want to insert, $regS is the register that contains the position we want to insert, $regT contains the value we want to insert (0 or 1), and $maskReg is used as a temporary register. The macro works by first moving the value we want to insert into $maskReg and then shifting the mask to the left by n ($regS). Afterwards, the OR operation will then look at the bit pattern we wanted to insert at and the temporary register that contains the value to be inserted.

iii. half_adder / full_adder

```
.macro half_adder($bit_A, $bit_B, $answer, $carry)
xor     $answer, $bit_A, $bit_B         # Answer = bit_A xor bit_B
and     $carry, $bit_A, $bit_B          # Carry = bit_A and bit_B
.end_macro

.macro full_adder($bit_A, $bit_B, $answer, $AB, $carry_in, $carry_out)
half_adder($bit_A, $bit_B, $answer, $AB)
and     $carry_out, $carry_in, $answer  # Carry out = carry in and (bit_A xor bit_B)
xor     $answer, $carry_in, $answer     # Answer = carry in xor (bit_A xor bit_B)
xor     $carry_out, $carry_out, $AB     # Carry out = (carry in and (bit_A xor bit_B)) :
                                        # (bit_A and bit_B)
.end_macro
```

Half_adder takes 2 inputted bits and provides the output bit in one sum bit and then a carry bit. Full_adder takes 2 half_adders and takes two other arguments, a carry_in and a carry_out.

iv. store_all_frame / restore_all_frame

These 2 macros are to store and load all registers necessary. Unlike the previous macros, these do not take any arguments and will store and restore all registers that need to be saved, regardless of being used or not. Both of these macros are used for convenience to shorten the amount of code. The registers being stored are $fp, $ra, $a0-$a3, and $s0-$s7.

```
.macro store_all_frame
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
.end_macro
.macro restore_all_frame
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
.end_macro
```

2) Utility Procedures

i. twos_complement and twos_complement_if_neg

```
twos_complement:
        store_all_frame

        not     $a0, $a0
        li      $al, 1
        jal     add_logical

        restore_all_frame
        jr      $ra

twos_complement_if_neg:
        store_all_frame

        li      $t1, 31
        move    $t2, $a0
        extract_nth_bit($t0, $t2, $t1)
        beqz    $t0, greater_than
        jal     twos_complement
        j       done

greater_than:
        move    $v0, $a0

done:
        restore_all_frame
        jr      $ra
```

As explained earlier in the binary system section, we can use twos_complement as a procedure to convert a value into a negative number if needed by inversing $a0 (our value) and adding one. Twos_complement_if_neg is there in case our number is already negative, then it turns the number into a positive number by calling twos_complement.

ii. twos complement 64bit

```
twos_complement_64bit:
        store_all_frame

        not     $a0, $a0
        not     $al, $al
        move    $s0, $al
        li      $al, 1
        jal     add_logical
        # First add
        move    $a0, $v1
        move    $al, $s0
        move    $s1, $v0
        jal     add_logical
        # Second add
        move    $v1, $v0
        move    $v0, $s1

        restore_all_frame
        jr      $ra
```

Similar to the other two, this procedure converts a number to two's complement. However, this procedure requires 2 registers to store the entire value. We start by inversing the 2 values we are converting and then storing each of the values separately in different registers.

3) Addition / Subtraction Implementation

```
add_logical:
        store_all_frame

        li      $t0, 0
        li      $t1, 0
        li      $t2, 0
        li      $t4, 0
        li      $t5, 0
        li      $t6, 0
        add     $s0, $zero, $zero

addition_loop:
        beq     $t2, 32, end_addition_loop
        extract_n_bit($t0, $a0)
        extract_n_bit($t1, $al)
        full_adder($t0, $t1, $t5, $t4, $t6, $t8)
        insert_to_nth_bit($s0, $t2, $t5, $t6)
        move    $t6, $t8
        addi    $t2, $t2, 1
        j       addition_loop

end_addition_loop:
        move    $v0, $s0
        move    $v1, $t8

        restore_all_frame
        jr      $ra


sub_logical:
        store_all_frame

        neg     $al, $al
        jal     add_logical

        restore_all_frame
        jr      $ra
```

i. add_logical / sub_logical

Add_logical prepares both operations by setting all the needed registers to 0. We begin by setting $t0 to 0 in order to hold $a2 and $t1 to hold $a1. $t2 is a position pointer for $s1 while $t4 is the carry holder. $t5 is the bit answer holder which we will put into $s2 later on and $t6 is the carry in for the full adder. We also have $s0 ready to save the final result.

Sub_logical is the same, but it negates the second integer.

ii. addition_loop:

First we check to see if $t2 is equal to 32 since that is the maximum amount of bits we can hold. If so, we end the loop and move the values to Lo and Hi so that they can store any overflow that happens. If not, we take the first bit of the two integers and use our full_adder macro to find the bit answer and carry out. We then insert our bit answer holder $t5 to the position pointer $t2 where $s0 is. We then make our carry in be our carry out and then increment the loop by 1. This continues until $t2 equals 32 and we have no more space.

4) Multiplication Implementation

```
mul_unsigned:
        store_all_frame

        move    $s6, $a1
        li      $s0, 0
        li      $s1, 0
        li      $s2, 0
        jal     twos_complement_if_neg
        move    $s3, $v0
        move    $a0, $s6
        jal     twos_complement_if_neg
        move    $s5, $v0
        li      $s4, 0

unsigned_mul_loop:
        beq     $s4, 32, stop_unsigned_mul
        beqz    $s5, stop_unsigned_mul
        extract_n_bit($t0, $s5)
        beqz    $t0, shift_1
        move    $a0, $s0
        move    $a1, $s1
        move    $a2, $s2
        move    $a3, $s3
        jal     adder_64bit
        move    $s0, $v0
        move    $s1, $v1

shift_1:
        move    $a0, $s2
        move    $a1, $s3
        jal     multiplicand_64_shift_left
        move    $s2, $v0
        move    $s3, $v1

        addi    $s4, $s4, 1
        j       unsigned_mul_loop

stop_unsigned_mul:
        move    $v0, $s1
        move    $v1, $s0

        restore_all_frame
        jr      $ra
```

```
mul_signed:
        store_all_frame
        move    $t1, $a0
        move    $t2, $a1
        li      $t3, 31
        extract_nth_bit($t0, $t1, $t3)
        extract_nth_bit($t4, $t2, $t3)


        # Compare signs, same = positive; Different = 2's comp
        xor     $s6, $t0, $t4
        jal     mul_unsigned
        move    $s0, $v0
        move    $s1, $v1

        beqz    $s6, end_mul_signed
        move    $a0, $s0
        move    $a1, $s1
        jal     twos_complement_64bit
        move    $s0, $v0
        move    $s1, $v1

end_mul_signed:
        move    $v0, $s0
        move    $v1, $s1

        restore_all_frame
        jr      $ra
```

i. mul_unsigned

Mul_unsigned begins by saving $s0 and $s1 for the Hi and Lo section of the products. We also save $s2 and $s3 for the multiplicand, but we make sure that the multiplicand is positive by using two's complement. We then save $s5 so we can hold the multiplier and $s4 for a loop counter. The multiplier must also be positive because we are using unsigned multiplication.

ii. unsigned_mul_loop

Unsigned_mul_loop first checks if the loop counter is equal to 32 so it does not go over the 32-bit limit for unsigned. If not, we then check the LSB of the multiplicand and extract it and shifting it left.

iii. mul_signed

For mul_signed, we start by getting the MSB for $a0 (multiplicand) and $a1 (multiplier). We then extract that using extract_nth_bit. Afterwards, we compare the two signs and save the result in $s6. If the signs are the same, we know it is positive, but if it is different then we need to use 64 bit two's complement.

```
adder_64bit:
        store_all_frame
        move    $s0, $a0
        move    $s1, $a1
        move    $s2, $a2
        move    $s3, $a3

        # add Los
        move    $a0, $s2
        move    $a1, $s0
        jal     add_logical
        move    $s2, $v0
        move    $t0, $v1

        # first add Hi with carry out
        move    $a0, $s3
        move    $a1, $t0
        jal     add_logical
        move    $s3, $v0

        # add hi part
        move    $a0, $s3
        move    $a1, $s1
        jal     add_logical
        move    $s3, $v0

        # end
        move    $v0, $s2
        move    $v1, $s3

        restore_all_frame
        jr      $ra

multiplicand_64_shift_left:
        store_all_frame

        # get MSB of low part
        move    $s0, $a1
        li      $t1, 31
        extract_nth_bit($t0, $s0, $t1)
        sll     $a0, $a0, 1
        insert_to_nth_bit($a0, $zero, $t0, $t1)
        sll     $a1, $a1, 1
        move    $v0, $a0
        move    $v1, $a1
        restore_all_frame
        jr      $ra
```

iv. adder_64bit

adder_64bit is a helpful implementation that assists in adding 64 bit numbers. We first separate registers to hold different values for the 64 bit answer. $s0 and $s2 will hold the Lo and Lo result respectively, as $s1 and $s3 will hold Hi and the Hi result. We first add the Los together using our previous add_logical implementation and store it in $s2. Afterwards, we do the same but with the registers that we saved for Hi. When both are done, we then add the two answers together to get the 64 bit answer.

v. multiplicand_64_shift_left

multiplicand_64_shift_left converts a single bit input to a 32 bit output. We start by getting the MSB of the low part using our extract_nth_bit and shifting by 31. We then shift again to make room for the high part of the output and insert the MSB of the low part into that area.

5) Division Implementation

```
div_unsigned:

        store_all_frame

        move    $s1, $a1
        jal     twos_complement_if_neg
        move    $s0, $v0
        move    $a0, $s1
        jal     twos_complement_if_neg
        move    $s1, $v0
        li      $s2, 0
        li      $s3, 0
        li      $s4, 0


        move    $s2, $s0

div_loop:
        beq     $s4, 32, stop_div
        sll     $s3, $s3, 1
        li      $t1, 31
        move    $t2, $s2

        # Get the MSB from dividend, insert at
        extract_nth_bit($t0, $t2, $t1)
        insert_to_nth_bit($s3, $zero, $t0, $t3)
        sll     $s2, $s2, 1

        move    $a0, $s3
        move    $a1, $s1
        jal     sub_logical
        move    $t3, $v0

        bltz    $t3, cont
        move    $s3, $t3

        li      $t0, 1
        insert_to_nth_bit($s2, $zero, $t0, $t2)

cont:
        addi    $s4, $s4, 1
        j       div_loop

stop_div:
        move    $v0, $s2
        move    $v1, $s3
```

i. div_unsigned

First, we make save a register for our divisor, in which we save $s1 and $a1. We also save $s0 and $a0 for the dividend. The other registers we want to save are $s2, $s3, and $s4 which are our quotient, remainder, and loop counter respectively.

ii. div_loop

We begin by checking if our loop is equal to 32 bits, if so we stop the loop and exit. If not, we shift the remainder to the left by 1 and make $t1 equal 31 and $t2 equal to our quotient. This is so that we can get the MSB from the dividend and insert it at the LSB of our remainder. After we insert, we shift left by one. We then start to our remainder by using sub_logical, subtracting the remainder and divisor until we have no remainder left, which is indicated when the remainder is larger than the divisor. We then insert one into the quotient at the MSB.

```
div_signed:
        store_all_frame

        move    $t1, $a0
        move    $t2, $a1
        li      $t3, 31
        extract_nth_bit($s3, $t1, $t3)
        extract_nth_bit($s4, $t2, $t3)


        xor     $s0, $s3, $s4

        jal     div_unsigned
        move    $s1, $v0
        move    $s2, $v1

        beqz    $s0, check_remainder
        move    $a0, $s1
        jal     twos_complement
        move    $s1, $v0

check_remainder:
        beqz    $s3, stop_div_signed
        move    $a0, $s2
        jal     twos_complement
        move    $s2, $v0

stop_div_signed:
        move    $v0, $s1
        move    $v1, $s2

        restore_all_frame
        jr      $ra
```

iii. div_signed

Similar to mul_signed, we start by getting the MSB for $a0 (dividend) and $a1 (divisor). We then extract that using extract_nth_bit. We compare the two signs and save the result in $s6. After comparing, we know it is positive if they're the same, but if it is different then we need to use two's complement with 64 bits.

iv. check_remainder

We use this to check the remainder and see if it is in two's complement form or not. If it is, then we stop and move on. But if it is not, we turn the remainder into two's complement form and then store it.

V. TESTING

In order to see if the calculator operations functions correctly, we can use the built in arithmetic operations that MIPS provides us. We will be using the normal procedures that we implemented earlier in our "proj_alu_normal" file. We will then compare these values to our logical values using "add", "sub", "mult", and "div".

A. Testing Implementation

1) *addition_normal / subtraction_normal*

Addition_normal and subtraction_normal call the MIPS instructions "add" and "sub". They have arguments passed through $a0 and $a1, and their values are stored in $v0.

2) *multiplication_normal*

Multiplication_normal calls the MIPS instruction "mult" and stores the values in two different registers, $v0 and $v1 representing Lo and Hi respectively.

3) *division_normal*

Division_normal calls the MIPS instruction "div" and stores the results in the values in the same way multiplication normal stores the values. $v0 represents the quotient and Lo while $v1 represents the remainder and Hi.

B. proj-auto-test.asm

Given the proj-auto-test.asm, we can test and see if our logical calculator matches with the normal operations. They compare both the normal operations and logical operations and look for matching outputs and returns a score based on how many are correct. Here is an example of what should be outputted if done correctly.

```
(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 --
```

VI. CONCLUSION

The project at hand was to implement a basic calculator using logical operations. Written in MIPS assembly language, we used our knowledge of Boolean logic and the binary system in order to achieve these results. Although it is a basic operations calculator, we are capable of continuing to other operations, such as square roots, exponents, etc.