Implementation of Basic Arithmetic Operations using MIPS Boolean Logic Operations

Narinder Singh

San Jose State University
narinder.p.singh@sjsu.edu

***Abstract*—The purpose of this report is to implement basic arithmetic operations i.e. (addition, subtraction, multiplication, and division) with the Boolean Logical Operations (and, or, not etc.) provided in MIPS assembly language. Instead of implementing them on a real MIPS processor, an IDE (Integrated Development Environment) MARS (MIPS Assembler and Runtime Simulator) is used.**

I. INTRODUCTION

Since MIPS provides its own basic arithmetic operations (add, sub, mul, div), the results obtained by the implantation of these operation using Boolean logical operations can be compared against the arithmetic operations provided in MIPS. The project can be implanted using the following steps.

1) Installing the MARS IDE.
2) Implementing the arithmetic operations which are already provided in MIPS
3) Designing and implementing arithmetic operations using only Boolean logical operations provided in MIPS
4) Finally comparing the results obtained in step 2 and step 3.

II. INSTALLATING MARS

A. *Downloading MARS*

MARS can be downloaded for free using the following website:
https://courses.missouristate.edu/KenVollmar/MARS/download.htm

B. *Downloading Project Files from SanJose State University*

Project Files can be downloaded file from Canvas at the site
https://sjsu.instructure.com/courses/1185206/assignments/4051197

Files Included are:

1) cs47_proj_macro.asm
macros that will be defined and used.

2) cs47_proj_procs.asm
Procedures that will be helpful in the project.

3) proj_auto_test.asm
For comparing the results of normal MIPS operations and implemented operations.

C. *Settings of MARS IDE*

The main options that needs to be check ON for the project files are:

1. Assemble all files in directory
2. Initialize Program Counter to global 'main' if defined

Other options that can be turned on for convenience, are shown in the Figure 1

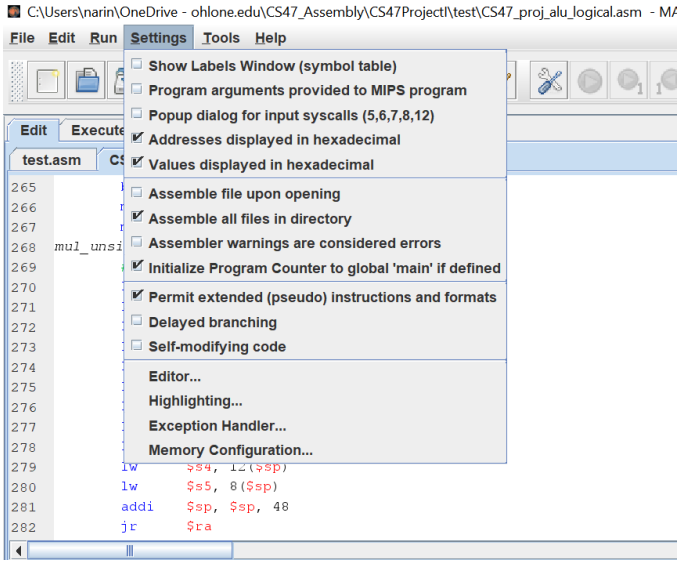

Fig. 1. Settings

III. TOOLS NEEDED FOR ARITHMETIC PROCEDURES

This project will be implemented using the procedure discussed below. As discussed before results obtained from Boolean logical operations will be compared with the results of arithmetic operations provided in MIPS. Following procedures will be helpful in achieving the target

A. au_normal

The au_normal procedure will be used to calculate the results using the MIPS provided arithmetic operations. It takes in the following arguments.

1. *Register $a0*

For the first operand of the arithmetic expression.

2. *Register $a1*

For the second operand of the arithmetic expression.

3. *Register $a2*

For the operator that is given in ASCII code to tell the procedure which operation to perform.

The results obtained from this procedure can be given as follows:

1. *Register $v0*
   a. *For addition and subtraction*
      The result will be returned in $v0
   b. *For multiplication and division*
      Lower bits of the result with multiplication will be returned in $v0. Quotient will be returned the case of Division.
2. *Register $v1*
   a. *For addition and subtraction*
      Not applied
   b. *For multiplication and division*
      Higher bits of the result will be returned in $v1 when multiplying and remainder will be returned the case of division.

B. au_logical

This is the heart of the program as it will calculate the arithmetic expressions using Boolean logical operations such as AND, OR, NOT, XOR, NOR etc. The arguments and return types are exactly as the au_normal procedure.

IV. DESIGN OF THE PROCEDURES

A. au_normal

Relatively simple this procedure will calculate the results directly using MIPS provided arithmetic operations. To check which operation is to be performed on the operands the operators are passed into the argument register ($a2). A branch structure is required to check which operator is passed in as an argument which can be done using the Branch instruction provided in MIPS. Figure 2 shows how this is done. The four operators that will be employed in this job are as follows

1. *Addition*

- Operator "+" will be used when addition is to be performed on the operands. The MIPS instruction that will perform this operation is 'add'.

2. *Subtraction*

- Operator "-" will be used when subtraction is to be performed on the operands. The MIPS instruction that will perform this operation is 'sub.

3. *Multiplication*

- Operator "*" will be used when multiplication is to be performed on the operands. The MIPS instruction that will perform this operation is 'mul.

4. *Division*

- Operator "/" will be used with division is to be performed on the operands. The MIPS instruction that will perform this operation is 'div.

```
28
29          # Check which operation to perform
30          beq     $t2, '+', add
31          beq     $t2, '-', subtract
32          beq     $t2, '*', multiply
33          beq     $t2, '/', divide
34          j       else
35  add:
36          add     $v0, $t0, $t1
37          j       else
38  subtract:
39          sub     $v0, $t0, $t1
40          j       else
41  multiply:
42          mul     $v0, $t0, $t1
43          mfhi    $v1
44          j       else
45  divide:
46          div     $t0, $t1
47          mflo    $v0
48          mfhi    $v1
49  else:
50          #Return to the caller
51          jr      $ra
52
```

Fig. 2. Implementing Branch

B. *au_logical*

au_logical uses a similar architecture for choosing between which operation is to be done on the passed operands. However, since it was very easy to just call the MIPS instructions in the case of au_normal, we did not have to modularize the code. In the case of au_logical it would be really hard to debug the code if the code is not broken up in different components. Hence, the branch mechanism of au_logical will call other routines to achieve its targets. The Figure 3 show how the code will look like once implemented.

```
35          # Check which operation to perform
36          beq     $s2, '+', add
37          beq     $s2, '-', subtract
38          beq     $s2, '*', multiply
39          beq     $s2, '/', divide
40          j       au_logical_ret   #if any other symbol is passed exit the procedure
41
42          #add     $s1, $zero, $zero # store argument index
43  add:
44          and     $a2, 0x00000000
45          jal     add_sub_logic
46          j       au_logical_ret
47  subtract:
48          ori     $a2, 0xffffffff
49          jal     add_sub_logic
50          j       au_logical_ret
51  multiply:
52          move    $a0, $s0
53          jal     twos_complement_if_neg
54          move    $s0, $v0
55          move    $a0, $s1
56          jal     twos_complement_if_neg
57          move    $s1, $v0
```

Fig. 3. Routines are called based on the branch mechanism

Procedure that will be called inside au_logical are as follows.

1) add_sub_logical

This is the engine from addition and subtraction operations of the au_logical procedure. The two operands are passed into $a0 and $a1 respectively and the mode is passed into $a2. If zero is passed in $a2 add_sub_logical will perform an addition operation on the operands and in the case when 0xFFFFFFFF is passed it will perform subtraction. Passing 32 ones is the key for subtraction as it is used as the initial carry bit when the addition is done between first operand and inverted second operand.

Binary Two Single Bit Addition Result

Bit 1 (A)	Bit 2 (B)	Sum Bit (Y)	Carry Bit (C)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Half Addition

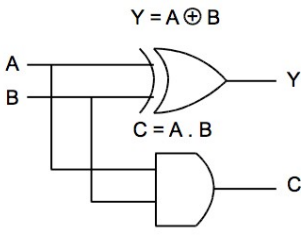

Fig. 4. Half Adder [2]

Figure 4 shows the diagram of Half adder which is chained to form a full adder to obtain multi-bit addition. A and B are passed into an XOR gate and AND gate to obtain Y which is the sum bit and C with is the carry bit.

Binary Three Single Bit Addition Result

	Bit 1 (CI) Carry In	Bit 2 (A)	Bit 3 (B)	Sum Bit (Y)	Carry Bit (CO) Carry Out
m0	0	0	0	0	0
m1	0	0	1	1	0
m2	0	1	0	1	0
m3	0	1	1	0	1
m4	1	0	0	1	0
m5	1	0	1	0	1
m6	1	1	0	0	1
m7	1	1	1	1	1

Full Addition

$Y = \Sigma m(1,2,4,7)$

$CO = \Sigma m(3,5,6,7)$

Fig. 5 Full Adder truth table [2]

This K-Map shown in Figure 6 can be used to design the full adder.

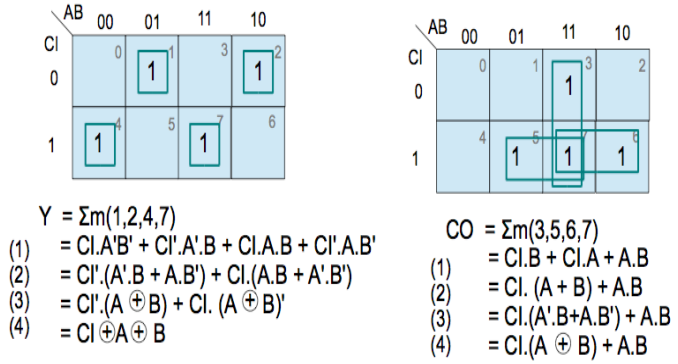

Fig. 6. K-Map for Full Adder [2]

The expressions used for Half adder can be simplified to design Full Adder as it make the reuse of XOR possible which is the key for reducing manufacturing cost when implementing on a silicon chip.

$Y = CI \oplus (A \oplus B)$

$CO = CI.(A \oplus B) + A.B$

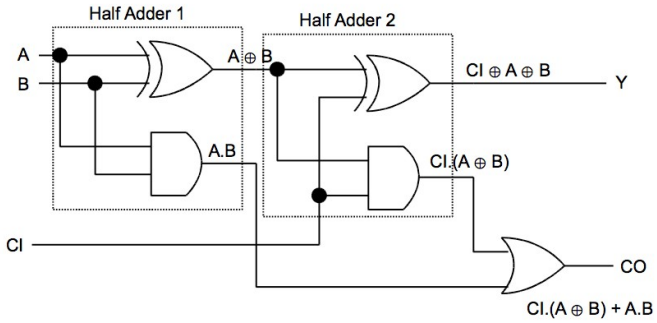

Fig. 7. Logical Design for Addition/Subtraction [2]

XOR operation is performed between the carry in-bit and the LSB of first operand. Then the result is and XOR with the LSB of second operand. OR operation is employed to calculate the final carry-out bit by using it between the carry-out bits from the two half adders.

To perform the subtraction operation the second operand is converted to its two's complement form because subtracting A from B is same as adding - A to B. For converting the second operand to its 2's complement we can use

$$A = \sim A + 1 \qquad (1)$$

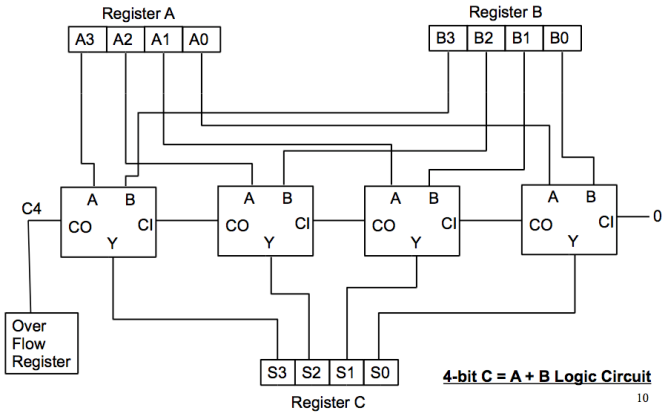

Fig. 8. Multi-bit Addition Only [2]

The carry in bit for addition can be ignored however when the performing subtraction it can serve as a switch that will invert the second operand. Fig. 8 shows the diagram of full adder when performing addition and the same circuit can be converted to an addition-subtraction by using an XOR in front of second operand B which will invert the contents of B depending on the state of carry in bit as shown in Figure 9.

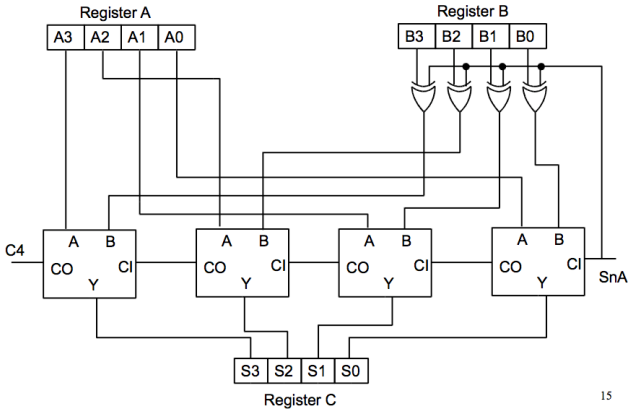

Fig. 9. Multi-bit Addition/Subtraction [2]

MIPS implementation of add_sub_logical requires the following flow chart as shown in Figure 10

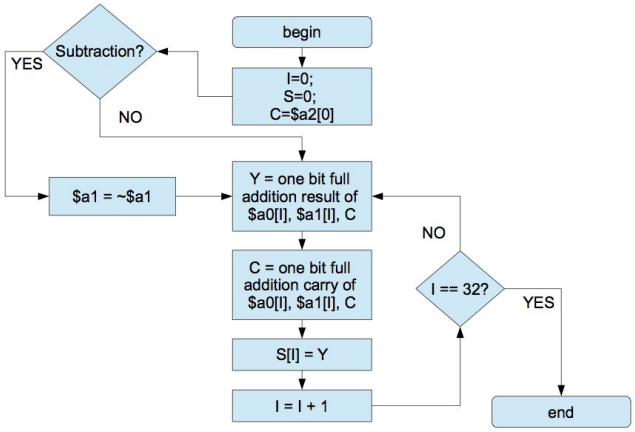

Fig. 10. Code Flowchart for Addition/Subtraction [2]

```
        # body
        and     $s2, $zero, $zero # $t0 = 0 --> index(I)
        and     $t1, $zero, $zero # $t1 = 0 --> sum(S)
        extract_nth_bit($t2, $a2, $t0) # $t2 = $a2[0] --> carry(C)
        bne     $t2, 1, add_loop
        not     $a1, $a1
add_loop:
        and     $t3, $zero, $zero # $t3 = 0 --> Y
        extract_nth_bit($s0, $a0, $s2) # $s0 = $a0[I]
        extract_nth_bit($s1, $a1, $s2) # $s1 = $a1[I]
        add     $t4, $zero, $zero
        xor     $t4, $s0, $s1  # $t3(Y) = one bit add result of $a0[I] and $a1[I]
        xor     $t3, $t2, $t4  # $t3(Y) = one bit add result of $a0[I], $a1[I] and $t2->carry(C)
        and     $s1, $s1, $s0
        and     $s0, $t4, $t2
        or      $t2, $s0, $s1
        insert_to_nth_bit($t1, $s2, $t3, $t4)
        #move   $t1, $t3       # $t1(S) = $t3(Y)
        addi    $s2, $s2, 1    # $t0 += 1 --> index(I)
        bne     $s2, 32, add_loop
        move    $v0, $t1       # return $t1(Sum) in $v0
        move    $v1, $t2       # return $t2(final Carry Out) in $v1
add_sub_logic_ret:
        #restore RTE
```

Fig. 11. Implementation of Loop Used in Addition/Subtraction

2) add_logical

This can be done in the branch form to avoid any confutions

```
        #add     $s1, $zero, $zero # store argument index
add:
        and     $a2, 0x00000000
        jal     add_sub_logic
        j       au_logical_ret
subtract:
        ori     $a2, 0xffffffff
        jal     add_sub_logic
        j       au_logical_ret
```

Fig. 12. Implementation for Addition Only

3) sub_logical

Refer to Figure 12 to see how subtraction is performed by passing 0xFFFFFFFF to the add_sub_logic

4) twos_complement

Since we are converting operands to their two's complement form quite often, so it is worth the time to make a routine that can take care of this task for us.

$a0 takes in the argument that is needed to be converted into its two's complement form and the result is returned in $v0.

```
twos_complement:
        #store RTE - 3 *4 = 12 bytes
        addi    $sp, $sp, -20
        sw      $fp, 20($sp)
        sw      $ra, 16($sp)
        sw      $a0, 12($sp)
        sw      $s0, 8($sp)
        addi    $fp, $sp, 20
        # body
        move    $s0, $a0
        not     $a0, $a0
        ori     $a1, $zero, 1
        and     $a2, $zero, $zero
        jal     add_sub_logic
twos_complement_ret:
        #restore RTE
```

Fig. 13. Implementation of Two's Complement

5) twos_complement_64bits

```
185         # body
186         move    $s0, $a0        # $s0 = $a0
187         move    $s1, $a1        # $s1 = $a1
188         not     $a0, $a0        # $a0 = ~$a0
189         ori     $a1, $zero, 1   # a1 = 1
190         and     $a2, $zero, $zero       # a2 = 0
191         jal     add_sub_logic
192         move    $s0, $v0        # store $v0 in $s0
193         not     $a0, $s1        # $a0 = ~upperbits
194         move    $a1, $v1        # $a1 = final carry out from add_sub_logic
195         and     $a2, $zero, $zero # $a0 = 0 telling add_sub_logic to add $a0 and $a1
196         jal     add_sub_logic
197         move    $v0, $s0        # returning lower bits calculated previously in $v0
198                                 # $v1 will automatically have the upper bits of 2s complement
199 twos_complement_64bit_ret:
200         #restore RTE
```

Fig. 14. Implementation of Two's Complement 64 Bits

6) bit_replicator

```
212 bit_replicator:
213             #store RTE - 3 *4 = 12 bytes
214             addi    $sp, $sp, -20
215             sw      $fp, 20($sp)
216             sw      $ra, 16($sp)
217             sw      $a0, 12($sp)
218             sw      $s0, 8($sp)
219             addi    $fp, $sp, 20
220             # body
221             move    $s0, $a0
222             beqz    $s0, bit_zero
223             ori     $v0, $zero, 0xffffffff
224             j       bit_replicator_ret
225 bit_zero:
226             or      $v0, $zero, $zero
227 bit_replicator_ret:
228             #restore RTE
229             lw      $fp, 20($sp)
230             lw      $ra, 16($sp)
231             lw      $a0, 12($sp)
232             lw      $s0, 8($sp)
233             addi    $sp, $sp, 20
234             jr      $ra
```

Fig. 15. Implementation of Bit Replicator

7) mul_unsigned

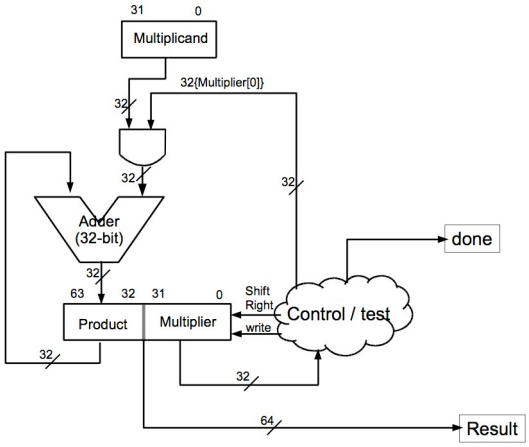

Fig. 16. Logical Design of Unsigned Multiplication [3]

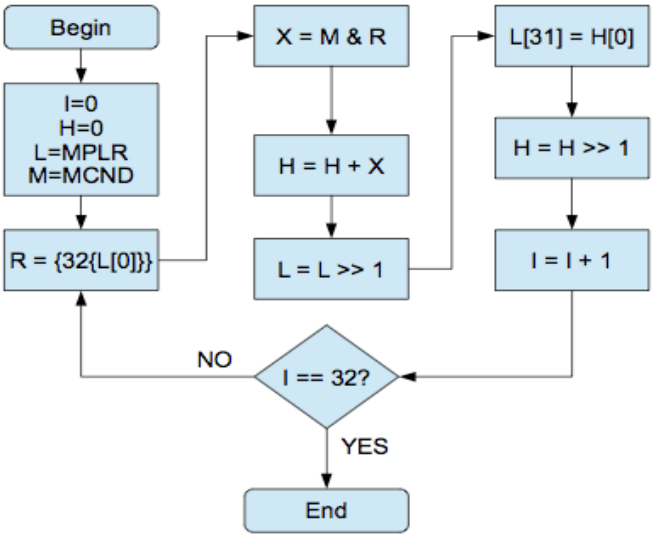

Fig. 17. Flowchart for Unsigned Multiplication [3]

```
mul_loop:
        extract_nth_bit($t0, $s0, $s2)
        move    $a0, $t0
        jal     bit_replicator
        move    $s4, $v0        # $s4(R) = 32{L[I]}
        and     $s5, $s1, $s4           # $s5(X) = $s1(M) and $s4(R)
        move    $a0, $s3        #$s3(H) = X($s5) + H($s3)
        move    $a1, $s5        #$s3(H) = X($s5) + H($s3)
        and     $a2, $zero, $zero
        jal     add_sub_logic
        move    $s3, $v0        #$s3(H) = X($t4) + H($t1)
        srl     $s0, $s0, 1     # L($s0) = L >> 1
        and     $t1, $zero, $zero       # H[temp]; temp($t1) = 0
        extract_nth_bit($t2, $s3, $t1)
        ori     $t4, $zero, 31
        insert_to_nth_bit($s0, $t4, $t2, $t5) # ($s0)L[31] = H[0] ->($t2)
        srl     $s3, $s3, 1     # H = H >> 1

        addi    $s2, $s2, 1
        bne     $s2, 32, mul_loop
        move    $v0, $s0
        move    $v1, $s3
mul_unsigned_ret:
        #restore RTE
        lw      $fp, 48($sp)
```

Fig. 18. Implementation of Loop in Unsigned Multiplication

1) *Extracting an Individual Bit*

```
# Macro : extract_nth_bit
# Usage : extract_nth_bit(<result>, <reg>, <position>)
.macro extract_nth_bit($res, $regD, $nth)
#add    $res, $zero, $zero
#addi   $res, $res, +31
#sub    $res, $res, $nth

addi    $res, $zero, 1
SLLV    $res, $res, $nth
and     $res, $regD, $res
SRLV    $res, $res, $nth
#or     $regD, $maskReg, $bit
#SLLV   $t0, $reg, $res        # Shift LEFT Logical Variable
#SRLV   $res, $reg, $res       # Shift Right Logical Variable
#SRLV   $res, $reg, $nth       # Shift Right Logical Variable
.end_macro
```

2) *Inserting into a Bit Position*

```
# Macro : insert_to_nth_bit
# Usage : insert_to_nth_bit(<regD>, <position>, <$bit>, <$maskReg>)
.macro insert_to_nth_bit($regD, $nth, $bit, $maskReg)
addi    $maskReg, $zero, +1
SLLV    $maskReg, $maskReg, $nth
not     $maskReg, $maskReg
and     $maskReg, $regD, $maskReg
SLLV    $bit, $bit, $nth
or      $regD, $maskReg, $bit
.end_macro
```

TESTING

```
(-6 / -6)         normal => R:0 Q:1        logical => R:3 Q:2       [not matched]
(-18 + 18)        normal => 0      logical => 0     [matched]
(-18 - 18)        normal => -36    logical => -36           [matched]
(-18 * 18)        normal => HI:-1 LO:-324          logical => HI:0 LO:18    [not matched]
(-18 / 18)        normal => R:0 Q:-1       logical => R:3 Q:2       [not matched]
(5 + -8)          normal => -3     logical => -3    [matched]
(5 - -8)          normal => 13     logical => 13    [matched]
(5 * -8)          normal => HI:-1 LO:-40   logical => HI:0 LO:8     [not matched]
(5 / -8)          normal => R:5 Q:0        logical => R:3 Q:2       [not matched]
(-19 + 3)         normal => -16    logical => -16           [matched]
(-19 - 3)         normal => -22    logical => -22           [matched]
(-19 * 3)         normal => HI:-1 LO:-57   logical => HI:0 LO:3     [not matched]
(-19 / 3)         normal => R:-1 Q:-6      logical => R:3 Q:2       [not matched]
(4 + 3)   normal => 7      logical => 7     [matched]
(4 - 3)   normal => 1      logical => 1     [matched]
(4 * 3)   normal => HI:0 LO:12     logical => HI:3 LO:3     [not matched]
(4 / 3)   normal => R:1 Q:1        logical => R:3 Q:2       [not 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:64    [not matched]
(-26 / -64)       normal => R:-26 Q:0      logical => R:3 Q:2       [not matched]


Total passed 20 / 40
```

V. CONCLUSION

The objective of this project was to implement a basic calculator through the use of logical operators. The project provided insight on how digital circuits operate and how the logic behind the circuits are implemented. The program was written using the MIPS assembly language and tested with a provided testing file which matched the outputs of the logical calculator operations with the MIPS arithmetic instruction set. Next possible steps include implementing more complex expressions such as exponentiation, square rooting, decimal calculations, and more

REFERENCES

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

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

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

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