Project Report of MIPS Logical Arithmetic Calculator (May 2017)

Yecheng Liang
San Jose State University
Yecheng.Liang@sjsu.edu

***Abstract*—This is the report for the project one that simulate a basic arithmetic calculator using only logical operation in MIPS.**

***Index Terms*— logical computing, project 1, MARS**

I. Introduction

THIS project is a arithmetic calculator in MARS IDE (MIPS assembler and runtime simulator). The program is divided into two parts each using a different method to compute the result, then compare the result with the other part. The first part of the project uses normal math operations of MIPS to compute the result of addition, subtraction, multiplication and division. The second part of the project uses MIPS logic operations only to compute the addition, subtraction, multiplication and division.

II. Requirement

The requirement of the project is complete the project using specific operations and compute the result correctly. The first part of the project, which uses existing arithmetic operation in MIPS, should be completed and tested first and provided testing for the second part of the project. The second part of project should only use MIPS logic operation to compute the result.

The project was developed based on MARS IDE 4.5. It is required to install MARS IDE of the same version to run the project properly. MARS IDE and its installation procedure could be found in the link below:

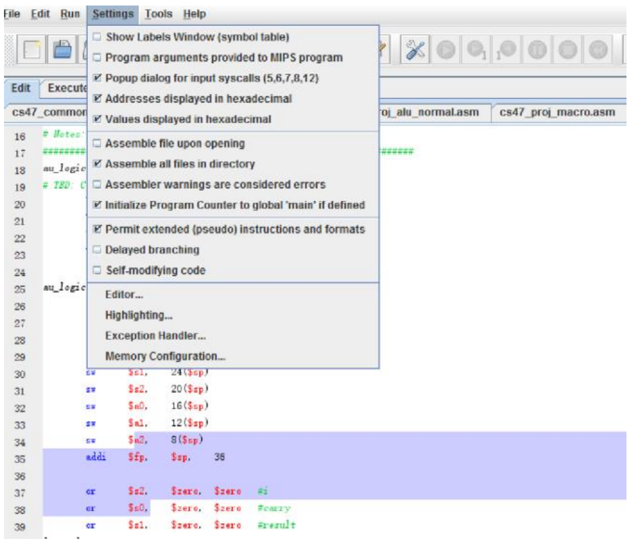
http://courses.missouristate.edu/KenVollmar/MARS/

Several required files of the project are already provided and are required to use in the project. Those files will act as an interface between our tester and emulation tester and make sure the program functions properly on different system. The files required by the project are listed below:

1) *cs47_common_macro.asm*
2) *cs47_proj_procs.asm*
3) *proj-auto-test.asm*

Those files should be placed in the same folder as the project files. Also, following option should be turned on in MARS simulator:

1) *Assembles all files in directory*
2) *'Initialize program counter to global main if defined*



- ☐ Show Labels Window (symbol table)
- ☐ Program arguments provided to MIPS program
- ✓ Popup dialog for input syscalls (5,6,7,8,12)
- ✓ Addresses displayed in hexadecimal
- ✓ Values displayed in hexadecimal
- ☐ Assemble file upon opening
- ✓ Assemble all files in directory
- ☐ Assembler warnings are considered errors
- ✓ Initialize Program Counter to global 'main' if defined
- ✓ Permit extended (pseudo) instructions and formats
- ☐ Delayed branching
- ☐ Self-modifying code
- Editor...
- Highlighting...
- Exception Handler...
- Memory Configuration...

```
# Notes
########
au_logic
# TBD: C

au_logic
        sw      $s1,    24($sp)
        sw      $s2,    20($sp)
        sw      $s0,    16($sp)
        sw      $a1,    12($sp)
        sw      $a2,    8($sp)
        addi    $fp,    $sp,    36

        or      $s2,    $zero,  $zero   #i
        or      $s0,    $zero,  $zero   #carry
        or      $s1,    $zero,  $zero   #result
```

Fig.1 MARS IDE settings

III. Basic Design For Part 1 Normal Procedure

The part one of the project will implement basic addition, subtraction, multiplication and division calculation using arithmetic operation provided in MIPS. The procedures are written in file:

1) *CS47_proj_alu_normal.asm*

The procedure au_normal is the interface procedure which accepted mathematical expressions and return the result of the expressions. It accepts three register as input arguments and return one or two registers as output depends on the type of expression. The arguments registers are the following:

2) *Register $a0*

The register $a0 is the first operand in the mathematical expression. It represented first addend in addition operation, minuend in subtraction operation, multiplicand in multiplication operation and dividend in division operation.

3) *Register $a1*

The register $a1 is the second operand in the mathematical expression. It represented second addend in addition operation, subtrahend in subtraction operation, multiplier in multiplication and divisor in division operation.

4) *Register $a2*

The register $a3 is the operator of the mathematic operation. The ASCII value of the operation should be inputted. The supported operator and ASCII values are listed below:

1)	'+'	43	addition
2)	'-'	45	subtraction
3)	'*'	42	multiplication
4)	'/'	47	division

5) Register $v0

The register $v0 is the result of the mathematical expression. It represents the sum of addition operation, difference of subtraction, Lo part of the product of multiplication operation and quotient of division operation.

6) Register $v1

The register $v1 is the additional result of some mathematical expression. It represents the Hi result of the multiplication operation and reminder of the division operation.

The procedure reads the input, decided which of the following procedure should be called and jump and link to the specific procedure. The examples of the codes are following:

```
beq     $a2,    42,
au_normal_multiplication
beq     $a2,    43,     au_normal_plus
beq     $a2,    45,     au_normal_minus
beq     $a2,    47,     au_normal_division
jr      $ra
```

IV. Basic Design For Part 2 Logic Procedure

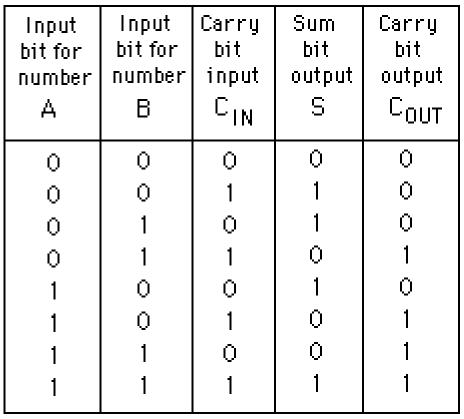
The part two of the project will implement basic addition, subtraction, multiplication and division calculation using only logic operation provided in MIPS. The procedures are written in file:

1) CS47_proj_alu_logical.asm

The procedure au_logical is the is the interface procedure which accepted mathematical expressions and return the result of the expressions. The design, inputs and outs are exact the same as those in au_normal of part one. The detailed are refenced in part III.

A) Logical addition

Logical addition is in procedure add_logical_plus, which will compute the sum of the two arguments $a0 and $a1. Logical addition is the most basic mathematical that is also used in other operations.



Input bit for number A	Input bit for number B	Carry bit input C_{IN}	Sum bit output S	Carry bit output C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

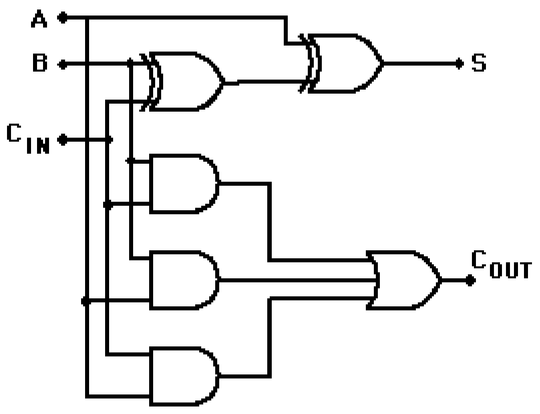


Fig.2 Full adder [1]

Fig. 2 Displays the truth table and logical design of a full adder, which takes three inputs, CI (carry-in), input A and input B.

With truth table, we can use Karnaugh map (K-Map) compute the simplify logical expression.

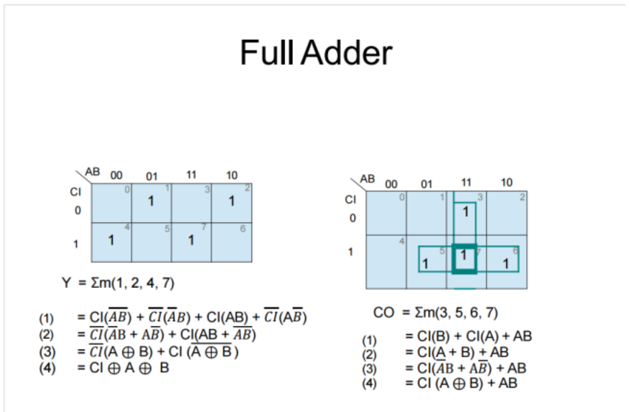


Fig.3 K-Map and logical expression for Full Adder [2]

The Fig.3 shows the simplifying of expression. The expression 4, which is Y = CI xor A xor B, CO = CI (A xor B) + AB is the expression used in the project. The sample codes are as follow:

```
xor     $t4,    $t1,    $t2
#A xor B

and     $t6,    $t1,    $t2
#A and B

and     $t7,    $s0,    $t4
#CI (A xor B)
```

```
xor     $t5,    $s0,    $t4
#CI xor (A xor B)

or      $s0,    $t6,    $t7
#carry over,    CI (A xor B) + AB
```

By connecting multiple full adder in ripple carry form, the sum of a multi-bit numbers in the system can be calculate. Fig. 4 shows an example of 4-bit adder. In this project, an extended 32-bit ripple carry adder is used which follows the same pattern.

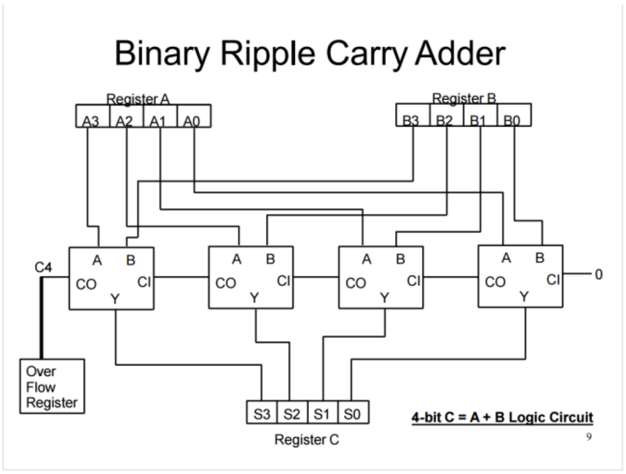


Fig.4 Multi-bit Binary Ripple Carry Adder [2]

B) Logical Subtraction

Mathematic subtraction is equal to adding the invited numbers, thus logical subtraction is equivalent to adding the inverted subtrahend in two’s complement form. The procedure au_logical_minus is used to process subtrahend then pass the argument to logical addition procedure.

C) Logical Multiplication

Logical multiplication is performed in two parts. The first parts saved the signed notion of the both arguments and inverted then into unsigned numbers. Then the procedure pass the arguments into unsigned multiplication procedure where unsigned multination is performed. The output is then sent back to the first procedure where it will restore the signed notation to the result and move the outputs to final register.

1) au_logical_multiplication

Procedure au_logical_multiplication is the procedure that performs signed logical multiplication operation. The procedure process the argument and then pass them into the unsigned multiplication procedure.

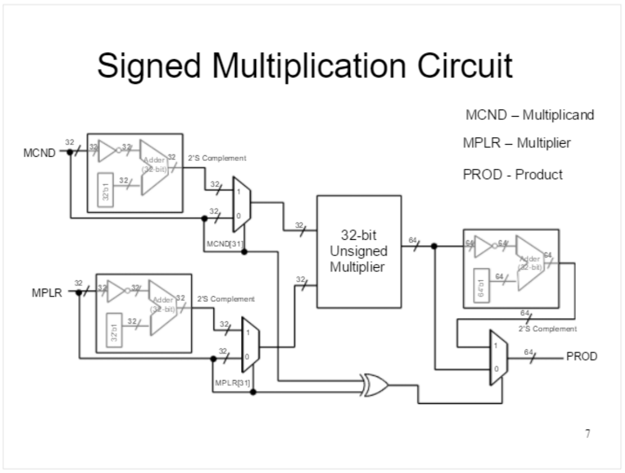
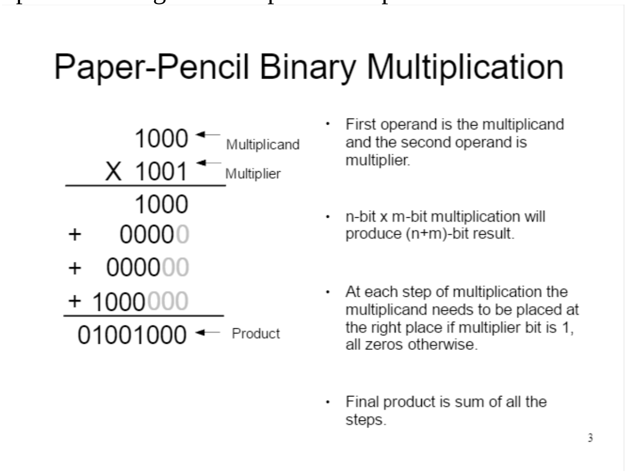


Fig.5 Signed Multiplication Circuit [3]

2) Unsigned multiplication

The procedure au_logical_multiplication used the Paper-Pencil Binary Multiplication algorithm to perform unsigned multiplication operation.



Paper-Pencil Binary Multiplication

```
     1000  ← Multiplicand
   X 1001  ← Multiplier
     1000
+   00000
+  000000
+ 1000000
 01001000  ← Product
```

- First operand is the multiplicand and the second operand is multiplier.
- n-bit x m-bit multiplication will produce (n+m)-bit result.
- At each step of multiplication the multiplicand needs to be placed at the right place if multiplier bit is 1, all zeros otherwise.
- Final product is sum of all the steps.

3

Fig.6 Paper-Pencil Binary Multiplication [3]

Fig.5 shows the procedure of paper-pencil binary multiplication. In each steps of the multiplication, if the numbers in multiplier is 1, a left-shifted multiplicand is added to the product. The shift amount is decided by the steps of the operation.

The product of two 32-bit number is a 64-bit number, thus in 32-bit MIPS the product is stored in two register. $v1 is Hi, which is the higher part of the number, while $v0 is Lo, which is the lower part of the number.

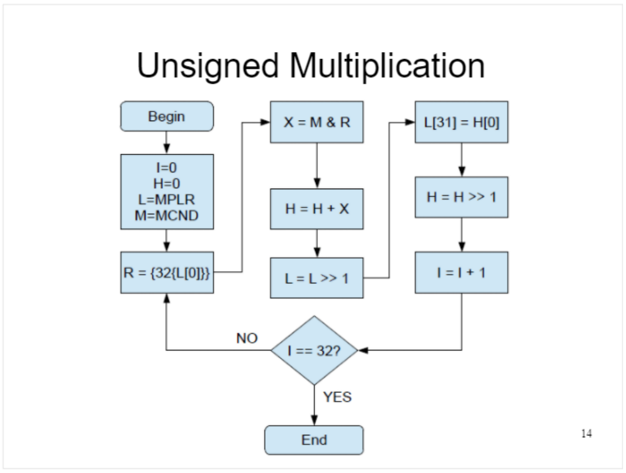


Fig.7 Simplified unsigned multiplication [3]

Fig.7 shows simplified unsigned multiplication. The codes of paper-pencil binary multiplication and division of Hi and Lo number are showed below:

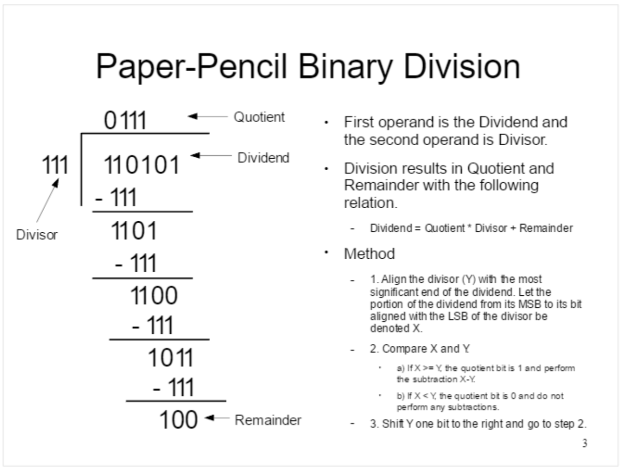
```
loop_multipication:
        beq       $s2,      32,       end_multipication
        extract_nth_bit($s7,          $s2)
        beq       $v0,      $zero,    mul_zero

        move      $s4,      $s6
        or        $s3,      $zero,    $zero     #j
        or        $s5,      $zero     $zero
################################################
#Divided number into Hi and Lo
###############################################
        move      $a0,      $s0
        move      $a1,      $s5
        jal       au_logical_plus
        move      $s0,      $v0

        move      $a0,      $s1
        move      $a1,      $s4
        jal       au_logical_plus
        move      $s1,      $v0

mul_zero:
        addi      $s2,      $s2,      1         #i++
        j         loop_multipication
```

The procedure use the au_logical_plus procedure to perform addition calculation.

D) Logical Division

The procedure au_logical_division perform signed division operation. The procedure will process the arguments, save the signed notation of the arguments, invert the arugments into unsigned number then pass into unsigned division procedure au_logical_unsigned_division. Then the procedure uses paper-pencil algorithm to perform unsigned logical division.

When unsigned calculation complete, the procedure will restore the sign notation of the result. The reminder will have the sign of the dividend and the quotient have the sign depends on both dividend and divider.



Paper-Pencil Binary Division
0111
Quotient
111
110101
Dividend
- 111
Divisor
1101
- 111
1100
- 111
1011
- 111
100
Remainder
3

- First operand is the Dividend and the second operand is Divisor.
- Division results in Quotient and Remainder with the following relation.
  - Dividend = Quotient * Divisor + Remainder
- Method
  - 1. Align the divisor (Y) with the most significant end of the dividend. Let the portion of the dividend from its MSB to its bit aligned with the LSB of the divisor be denoted X.
  - 2. Compare X and Y.
    - a) If X >= Y, the quotient bit is 1 and perform the subtraction X-Y.
    - b) If X < Y, the quotient bit is 0 and do not perform any subtractions.
  - 3. Shift Y one bit to the right and go to step 2.

Fig.8 Paper-Pencil Binary Division [4]

Fig.8 Shows the procedure of paper-pencil binary division algorithm. It use the subtraction procedure in B).

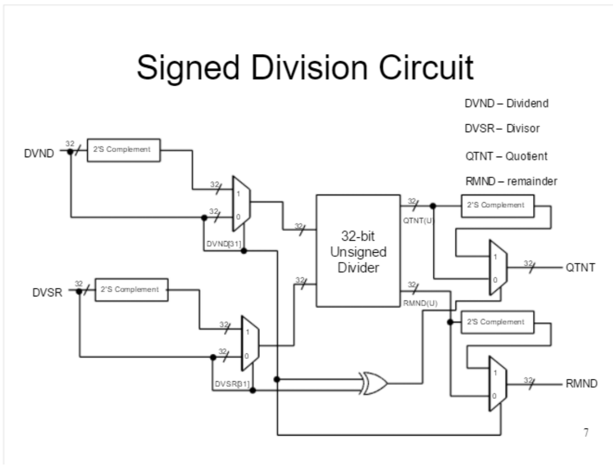


Fig.9 Signed Division Circuit [4]

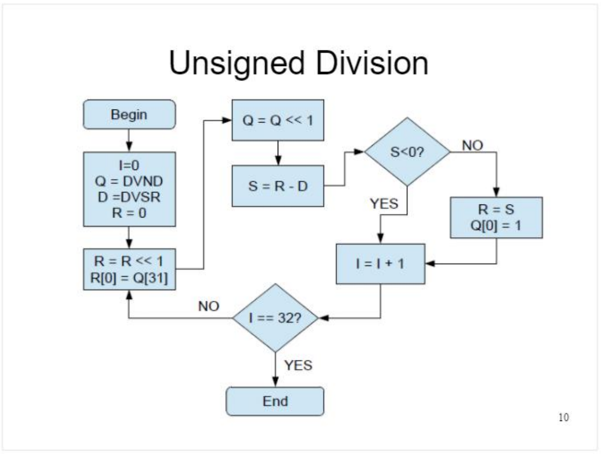


Fig.10 Unsigned Division [4]

Fig. 10 shows the procedure of an unsigned division. The procedure extract 1-bit from the dividend from left to right, shift-left and insert the extracted bit into the reminder and try to subtract the divider. When it reaches the end of dividend, anything remains in the reminder register is output as the reminder. The example codes are as follow:

```
loop_div:
  beq     $s2,     32,      end_unsigned_division
  sll     $s3,     $s3,     1          #          R = R << 1

  li      $s4,     31
  extract_nth_bit($s0,      $s4)
  #insert_to_nth_bit($s3,   $zero,    $v0)
  #move   $s3,     $v0
  or      $s3,     $s3,     $v0
  sll     $s0,     $s0,     1

  move    $a0,     $s3
  move    $a1,     $s1
  jal     au_logical_minus
  #move   $s4,     $v0      #          S = R - D
  bltz    $v0,     skip     #          if D < 0
  move    $s3,     $v0
  li      $t0,     1
  insert_to_nth_bit($s0,    $zero,    $t0)
  move    $s0,     $v0
skip:
  addi    $s2,     $s2,     1          #i++
  j       loop_div
```

E) Utility Macros and Procedures

There are several utility macros and procedure in the project to for code use and simplify purpose. Each of them severed a single purpose and is used many times in different part of the project. The Macros and procedure are as follows:

1) extract_nth_bit($regS,$regT)

This macro will extract the specific number from a number. It is used to extract bit from arguments. It takes two register, $regS is a number and $regT is which number to extract.

2) insert_to_nth_bit($regD,$regS,$regT)

This macro will insert the specific number into number in a specific place. It is used to insert bit into arguments. It takes three register, $regD is the number should be inserted into; $regS is the position the number should be inserted into; $regT is the inserted number. The result is stored in $v0.

3) Twos_complement_invertor

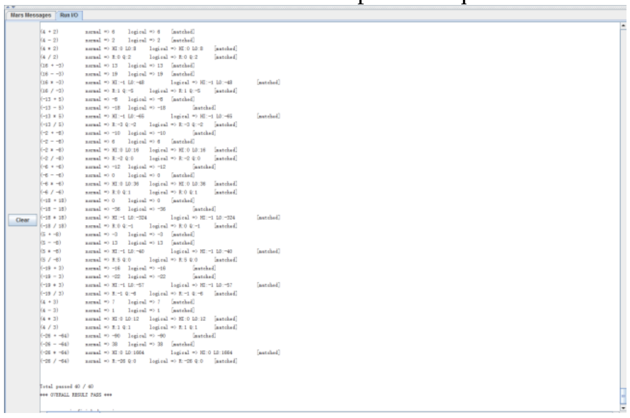
This procedure takes a number and outputs it's inverted two 'complement number. The output is stored in $v0.

V. Testing

The basic testing program is provided in

proj-auto-test.asm

However, it only compares the outputs of both part of the project without verifying them, thus it is possible that both parts of the project are incorrect when tester report pass. Manual verification of the outputs is required.



```
(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 ***
```

Fig.11 Tester Outputs

Fig.11 shows the example outputs of the project tester which has been manually tested and verified.

VI. Conclusion

This project compute four types of basic arithmetic expression in MIPS in both MIPS normal instruction or only logical operation. After completing the project, I learned the how logical circuit and circuit calculator works to compute binary numbers. Now I can build logical calculator in games such as Minecraft and in even in real life. This experience would be the millstone for me to study more about what drive the computer that we use every day.

References

[1] Nave, R. (2017). The Full-Adder. [online] Hyperphysics.phy-astr.gsu.edu. Available at: http://hyperphysics.phy-astr.gsu.edu/hbase/Electronic/fulladd.html#c1 [Accessed 11 May 2017].

[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