Arithmetic Implemented By MIPS Logic Operations

Jason Tang
San Jose State University
jason.t.tang@sjsu.edu

***Abstract*—This report focuses on the implementation of a basic calculator that is capable of addition, subtraction, multiplication, and division using MIPS instructions and bit manipulation in MARS (MIPS Assembler and Runtime Simulator).**

I. Introduction

Running the program in MIPS, the program will run addition, subtraction, multiplication, and division on the given operands and compare the results of MIPS instructions and bit manipulated logical operations following Boolean logic. This project uses MIPS instructions in the Mars simulator to calculate and compare values. The project goals are:

1) To Acquire a working version of the MARS simulator .
2) To write functions that will calculate addition, subtraction, multiplication, and division using MIPS commands and bit manipulation
3) To debug and test the program until it runs in the MARS simulator.

This report provides steps on how to install MARS as well as discusses the design and implementation process behind the logical procedures used for the arithmetic calculations. It also observes the test cases used to ensure the procedures have been implemented correctly and potential errors.

II. Installation and Setup

A. Acquire MARS

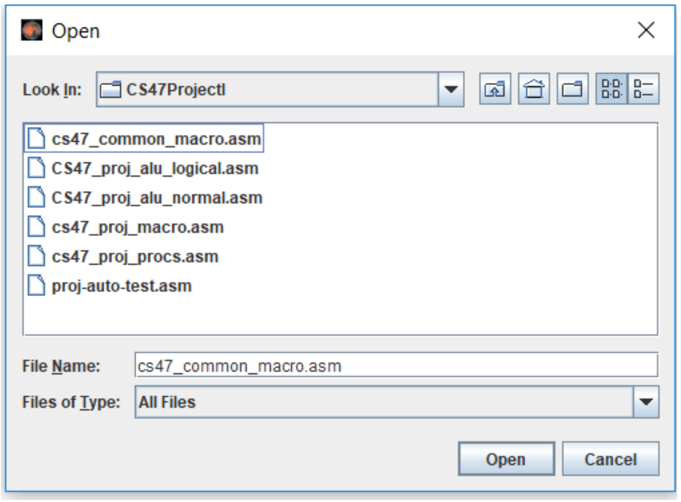
If MARS is not already on the device being used, download MARS from the site url below:
http://courses.missouristate.edu/KenVollmar/mars/download.htm

B. Acquire Starter Code

The starter code can be found on Canvas at this link https://sjsu.instructure.com/courses/1263903/assignments/4776422?module_item_id=9632934 inside of a zip file that must be unzipped and should include the following.

1) cs47_proj_macro.asm
The future location of implemented macros.
2) cs47_proj_procs.asm
A file for the tester of the code that should not be modified.
3) proj_auto_test.asm
The testing procedure which should not be modified
4) *CS_47_proj_alu_normal.asm*
The future location of the MIPS instructed operations
5) *CS_47_proj_alu_logical.asm*
The future location of the bit manipulating operations
6) *cs47_common_macro.asm*
The tester's macros that should not be modified.

Start the MARS simulator and open the unzipped folder containing the codes. To open files, navigate to the top left of the screen and click File -> open then navigate to unzipped project folder. It should look like this

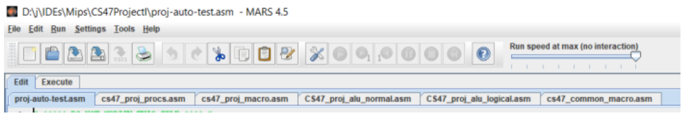


cs47_common_macro.asm
CS47_proj_alu_logical.asm
CS47_proj_alu_normal.asm
cs47_proj_macro.asm
cs47_proj_procs.asm
proj-auto-test.asm

File Name: cs47_common_macro.asm
Files of Type: All Files

Fig. 1. Opened directory of the Project

Open all of the files until all 6 can be seen in MARS navigation bar above the code.



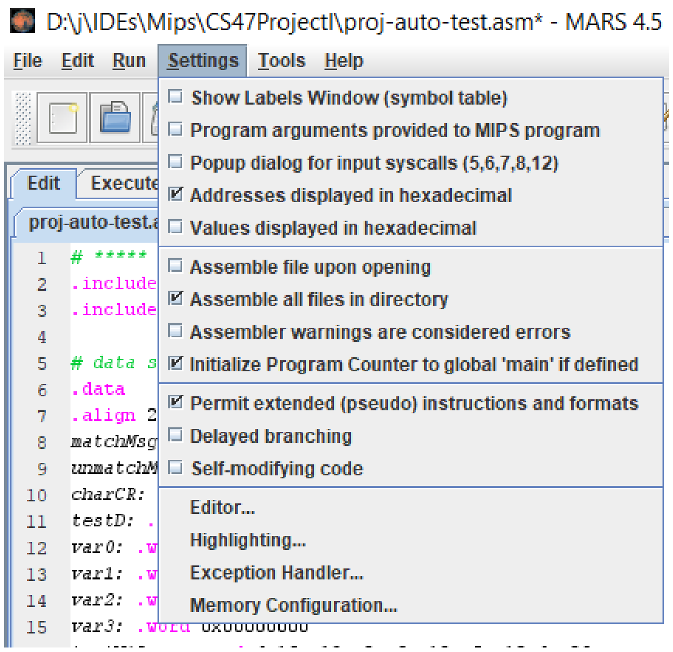
D:\j\IDEs\Mips\CS47ProjectI\proj-auto-test.asm - MARS 4.5
Run speed at max (no interaction)
proj-auto-test.asm | cs47_proj_procs.asm | cs47_proj_macro.asm | CS47_proj_alu_normal.asm | CS47_proj_alu_logical.asm | cs47_common_macro.asm

Fig. 2. Loading All Files

C. Settings of simulation tool

At the bottom left of the code window, there will be an optional setting, "Show Line Numbers". This setting is for user convenience and should be checked off

The program was written and tested with the following settings:


D:\j\IDEs\Mips\CS47ProjectI\proj-auto-test.asm* - MARS 4.5
File Edit Run Settings Tools Help
Edit Execute
proj-auto-test.a

- ☐ 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...

Fig. 3. Settings

1) *Addresses displayed in Hexadecimal*
   Toggles between Hexadecimal and Decimal display of memory address
2) *Assemble all files in directory*
   All files in current directory will be assembled when assemble operation is selected
3) *Initialize Program Counter to global 'main' if defined*
   Program Counter set to text address locally defined as 'main' if defined
4) *Permit extended (pseudo) instructions and formats*
   Allows for pseudo instructions and formats in the program

III. REQUIREMENTS OF ARITHMETIC PROCEDURES

For the Project, MIPS instruction based addition, subtraction, multiplication, and division, hereby referred to as the "normal" procedure as well as bit manipulated calculations hereby referred to as "logical" are required

A. Normal Procedure

This procedure, located in CS47_proj_alu_normal.asm, named au_normal, will consist of MIPS instructions that calculate the operations automatically. These operations are performed by add, sub, mul, and div. The procedure takes in three arguments:

1) Register $a0
First Number
2) Register $a1
Second Number
3) Register $a2
The operation to be performed, determined by the ASCII code of '+','-','*', and '/'

The procedure returns in two registers. The values of these registers differs based on which operation has been called on it.

1) *Register $v0*

Returns the result of the addition or subtraction. When multiplying, returns the lower 32 bits of the result. In the case of division, returns the quotient

2)*Register $v1*

Is only used in multiplication to return the upper 32 bits of the result and returns the remainder of the division

B. Logical Procedures

These procedures, located in CS47_proj_alu_logical, named au_logical, contains multiple procedures that calculate using logical bit manipulation operations, namely AND, OR, NOT, and XOR. add, sub, mul, and div are only used to increment counters in loops. The procedure takes in the same registers as au_normal:

1) Register $a0
First Number
2) Register $a1
Second Number
3) Register $a2
The operation to be performed, determined by the ASCII code of '+','-','*', and '/'

The procedure also returns in two registers, the exact same for au_normal. The values of these registers differs based on which operation has been called on it.

1) *Register $v0*

Returns the result of the addition or subtraction. In the case of multiplication, returns the lower 32 bits of the result. In the case of division, returns the quotient

2)*Register $v1*

Is only used in multiplication to return the upper 32 bits of the result and returns the remainder of the division

The procedure is also supplemented by other procedures which each handle an operation.

IV. DESIGN AND IMPLEMENTATION OF PROCEDURES

A. Normal Procedure

Au_normal can perform every operation given the three arguments. First it determines which math operator has been passed into register $a2. Using branching logic, branches to perform the corresponding MIPS operation. Returns results in the $v0 and $v1 registers. The $v0 and $v1 regiseters are used as the lo and hi registers for multiplication, quotient and remainder for division, and value and carry out bit for addition/subtraction.

1) Operator '+'
Calls for addition of $a0 and $a1 using add.

2) Operator '-'

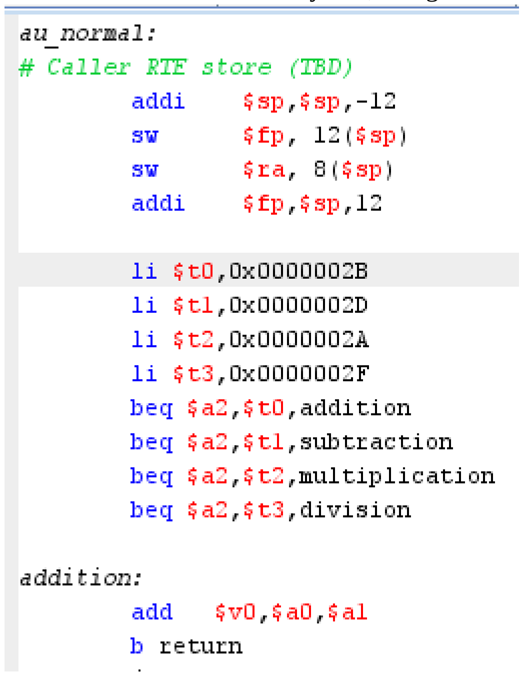
Calls for subtraction of $a0 and $a1 using sub.

3) Operator ''*

Calls for the multiplication of $a0 and $a1 using mul

4) Operator '/'

Calls for division of $a0 by $a1, using div

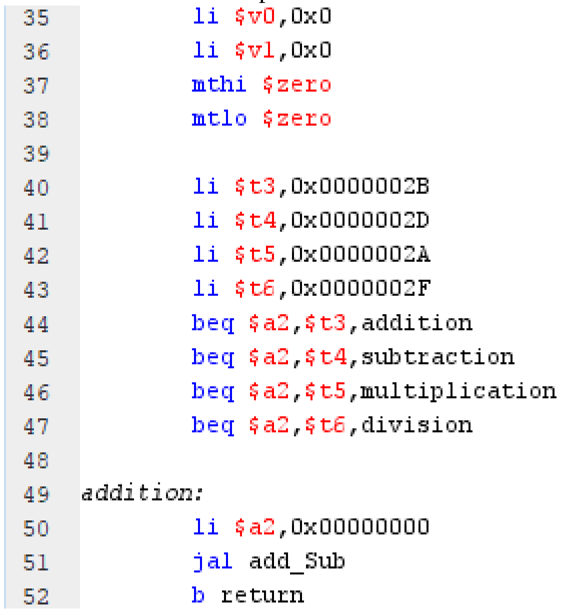


```
au_normal:
# Caller RTE store (TBD)
        addi    $sp,$sp,-12
        sw      $fp, 12($sp)
        sw      $ra, 8($sp)
        addi    $fp,$sp,12

        li $t0,0x0000002B
        li $t1,0x0000002D
        li $t2,0x0000002A
        li $t3,0x0000002F
        beq $a2,$t0,addition
        beq $a2,$t1,subtraction
        beq $a2,$t2,multiplication
        beq $a2,$t3,division

addition:
        add   $v0,$a0,$a1
        b return
```

Fig. 4. Implementation of the branching logic and the first branch for au_normal

B. Logical Procedure

Similarly, the au_logical also uses logical branching to determine what operation needs to be done, although each of its branches calls another procedure to calculate the value

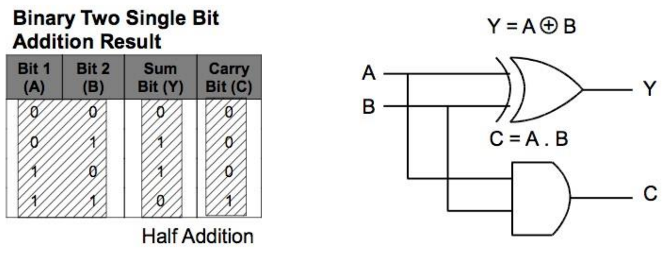


```
        li $v0,0x0
        li $v1,0x0
        mthi $zero
        mtlo $zero

        li $t3,0x0000002B
        li $t4,0x0000002D
        li $t5,0x0000002A
        li $t6,0x0000002F
        beq $a2,$t3,addition
        beq $a2,$t4,subtraction
        beq $a2,$t5,multiplication
        beq $a2,$t6,division

addition:
        li $a2,0x00000000
        jal add_Sub
        b return
```

Fig. 5. Implementation of the branching logic and the first branch for au_logical

1) add_sub_logical

The procedure will add or subtract the two numbers in the $a0 and $a1 registers based on the operation in the $a2 register, 0x00000000 for addition or 0xFFFFFFFF for subtraction. Leaves the results in $v0 and the carry in bit in $v1.



Binary Two Single Bit Addition Result

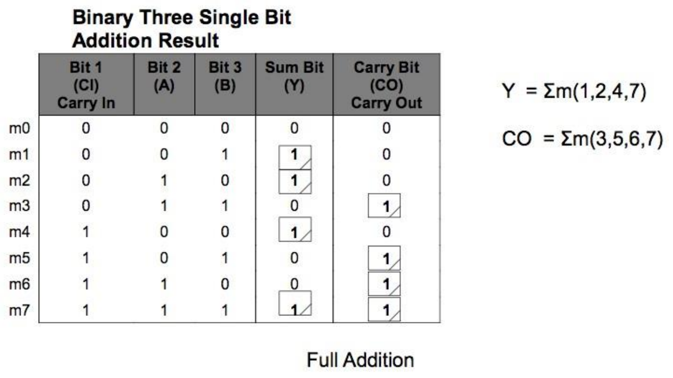
Bit 1 (A)	Bit 2 (B)	Sum Bit (Y)	Carry Bit (C)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Half Addition

Fig. 6. Half Adder [2]

Fig. 6 is the truth table and logic circuit of the Half Adder, which adds two bytes and calculates its carry bit. The sum bit is the XOR of A and B whereas the carry bit is the AND of A and B.

The full adder implements two half adder and considers the carry in and carry out bits. Fig. 7 shows that the truth table with 0 as the carry in bit is the same as the original half adder.



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

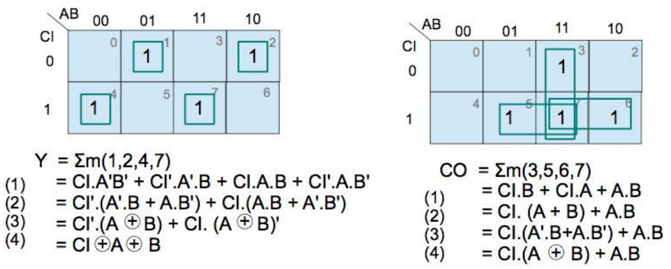
Full Addition

Y = Σm(1,2,4,7)

CO = Σm(3,5,6,7)

Fig. 7. Truth Table for Full Adder [2]

The K-map for creating the circuit design of the Full Adder is as follows.



Y = Σm(1,2,4,7)
(1) = CI.A'B' + CI'.A'.B + CI.A.B + CI'.A.B'
(2) = CI'.(A'.B + A.B') + CI.(A.B + A'.B')
(3) = CI'.(A ⊕ B) + CI. (A ⊕ B)'
(4) = CI ⊕ A ⊕ B

CO = Σm(3,5,6,7)
(1) = CI.B + CI.A + A.B
(2) = CI. (A + B) + A.B
(3) = CI.(A'.B+A.B') + A.B
(4) = CI.(A ⊕ B) + A.B

Fig. 8. K-Map for Full Adder [2]

Fig. 8 is a simplification that reuses the XOR from the half adder. The remaining expressions are used to calculate the addition.

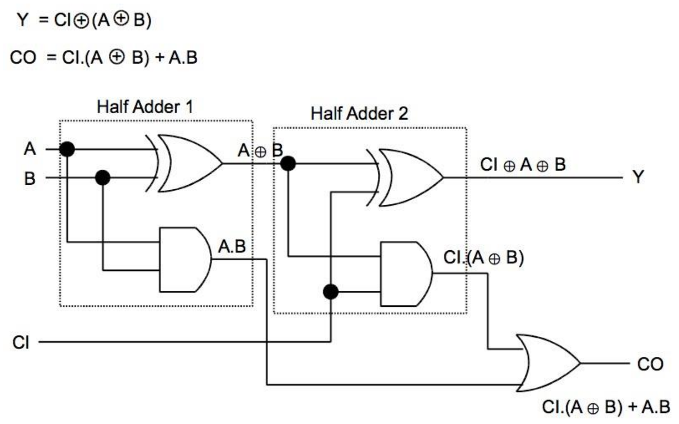


Fig. 9. Logical Design for Addition/Subtraction [2]

The sum is calculated by the XOR operation with the carry-in bit, first bit of the first number, and the first bit of the second number. The final carry-out bit is the OR operation of the carry-out bits. This means that the or operation takes in the result of one of the half adders and the other result is a product of both half adders.

The add_Sub procedure can be used for subtraction as well as addition. Subtraction is addition with a negative number. To do subtraction, the second operand should be converted to two's complement form like so:

$$\$a0 = \sim \$a0 + 1. \quad (1)$$

The value of $a2 will determines which operation is applied. If $a2 contains 0x00000000, addition is performed; if $a2 holds 0xFFFFFFFF, subtraction is performed. Subtraction is the same as addition, except subtraction inverts the second operand and takes the carry in bit from $a2, which should return a 1 instead of a 0. Fig. 10 shows the addition circuit and the reused addition circut in Fig.11.

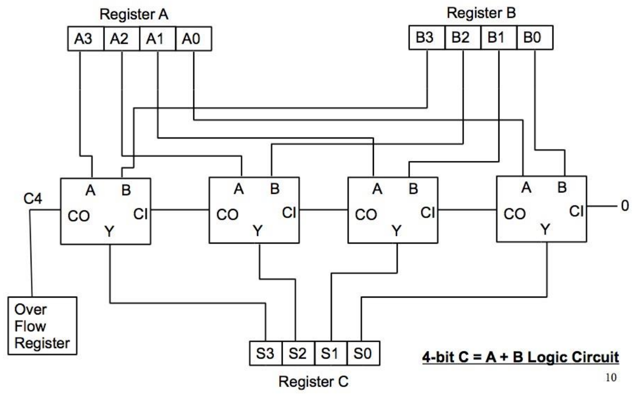


Fig. 10. Multi-bit Addition Only [2]

If the carry in bit is 1, B must have been inverted for subtraction; otherwise, it will carry on with no carry-in bit and pass whatever value B is holding for addition.

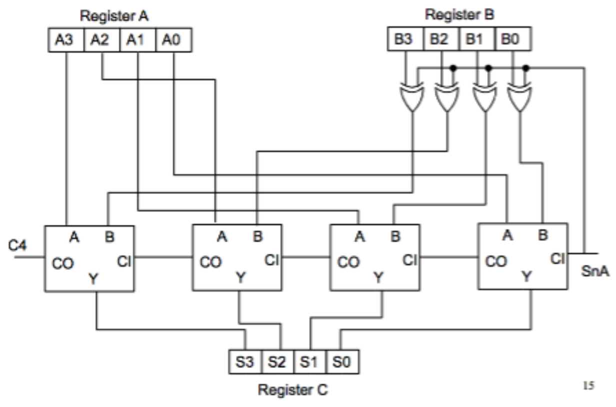


Fig. 11. Multi-bit Addition/Subtraction [2]

To implement this in the software as shown by Fig. 12, loop 32 times. Use branch logic to determine whether to do addition or subtraction and invert the second number when necessary. The Boolean expressions calculate the value of the sum.

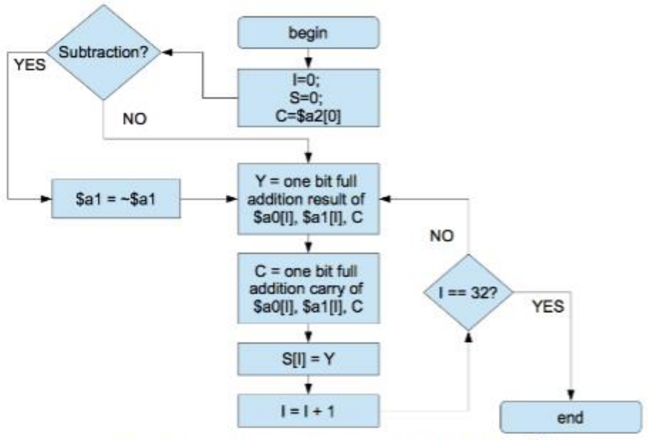
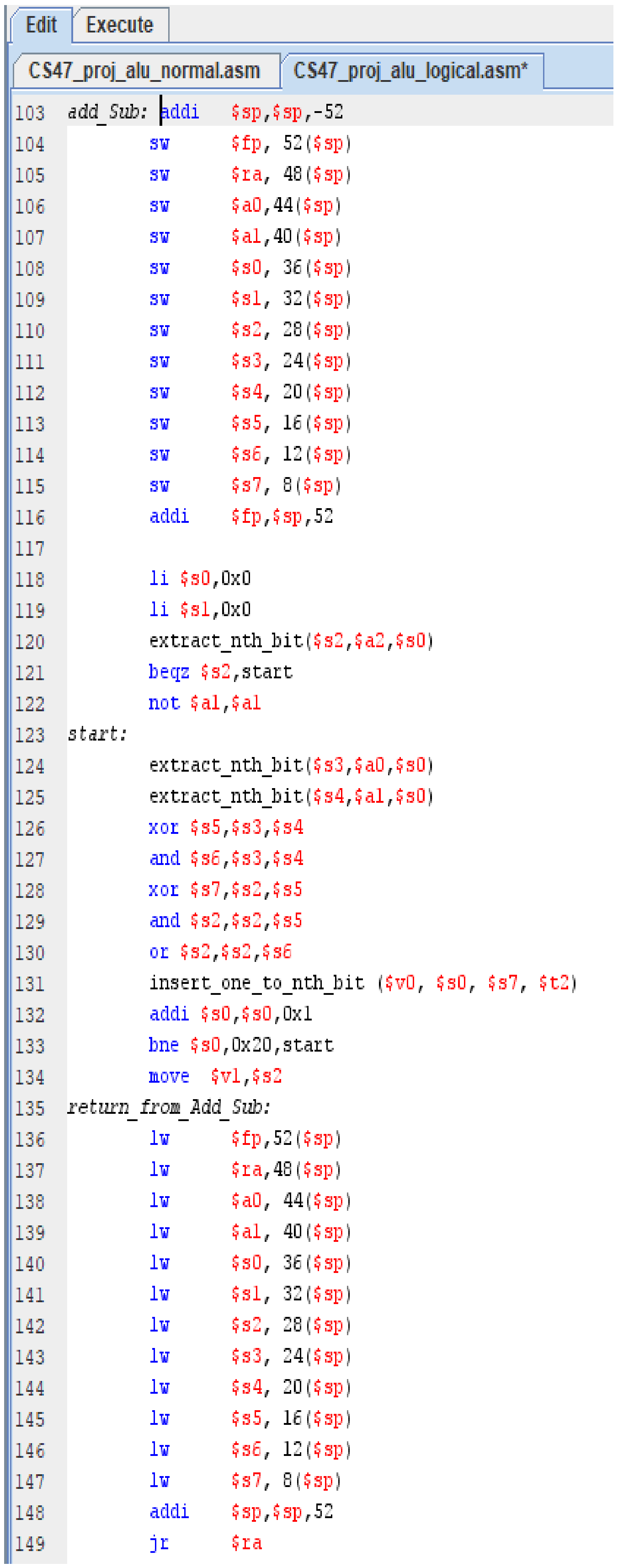

begin
I=0;
S=0;
C=$a2[0]
Subtraction?
YES
NO
$a1 = ~$a1
Y = one bit full addition result of $a0[I], $a1[I], C
C = one bit full addition carry of $a0[I], $a1[I], C
S[I] = Y
I = I + 1
I == 32?
NO
YES
end

Fig. 12. Code Flowchart for Addition/Subtraction [2]

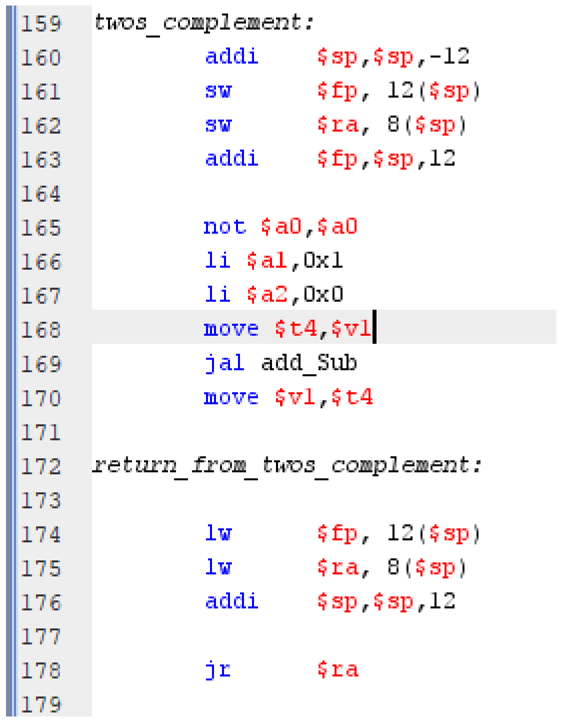


```
add_Sub: addi   $sp,$sp,-52
         sw     $fp, 52($sp)
         sw     $ra, 48($sp)
         sw     $a0,44($sp)
         sw     $a1,40($sp)
         sw     $s0, 36($sp)
         sw     $s1, 32($sp)
         sw     $s2, 28($sp)
         sw     $s3, 24($sp)
         sw     $s4, 20($sp)
         sw     $s5, 16($sp)
         sw     $s6, 12($sp)
         sw     $s7, 8($sp)
         addi   $fp,$sp,52

         li $s0,0x0
         li $s1,0x0
         extract_nth_bit($s2,$a2,$s0)
         beqz $s2,start
         not $a1,$a1
start:
         extract_nth_bit($s3,$a0,$s0)
         extract_nth_bit($s4,$a1,$s0)
         xor $s5,$s3,$s4
         and $s6,$s3,$s4
         xor $s7,$s2,$s5
         and $s2,$s2,$s5
         or $s2,$s2,$s6
         insert_one_to_nth_bit ($v0, $s0, $s7, $t2)
         addi $s0,$s0,0x1
         bne $s0,0x20,start
         move  $v1,$s2
return_from_Add_Sub:
         lw     $fp,52($sp)
         lw     $ra,48($sp)
         lw     $a0, 44($sp)
         lw     $a1, 40($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,52
         jr     $ra
```

Figure 13. Implementation of Addition/Subtraction in MAR

2) twos_complement

This procedure converts $a0's number to its twos complement form by calculating the not value and using add logical to add 1 to the value and returns the value to $v0

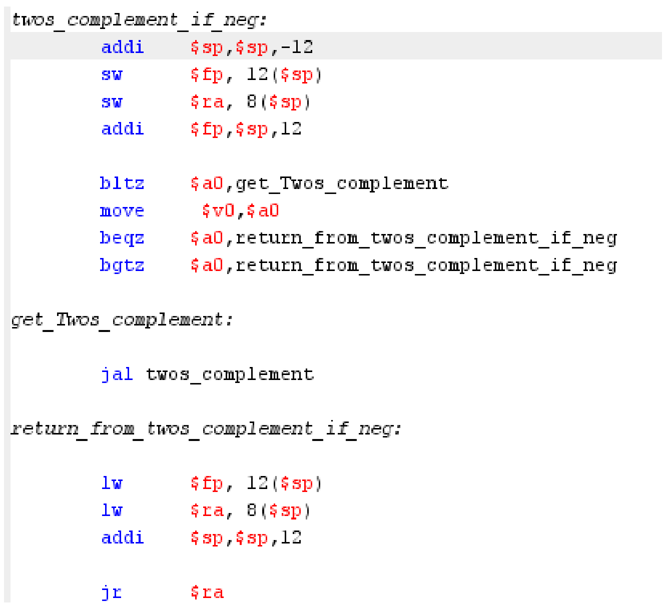


```
twos_complement:
        addi    $sp,$sp,-12
        sw      $fp, 12($sp)
        sw      $ra, 8($sp)
        addi    $fp,$sp,12

        not $a0,$a0
        li $a1,0x1
        li $a2,0x0
        move $t4,$v1
        jal add_Sub
        move $v1,$t4

return_from_twos_complement:

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

        jr      $ra
```

Figure 14. Implementation of twos_complement in MARS

3)twos_complement_if_neg

This procedure converts a number in register $a0 to its positive two's complement form in $v0. Functionally converts negative numbers to positive.

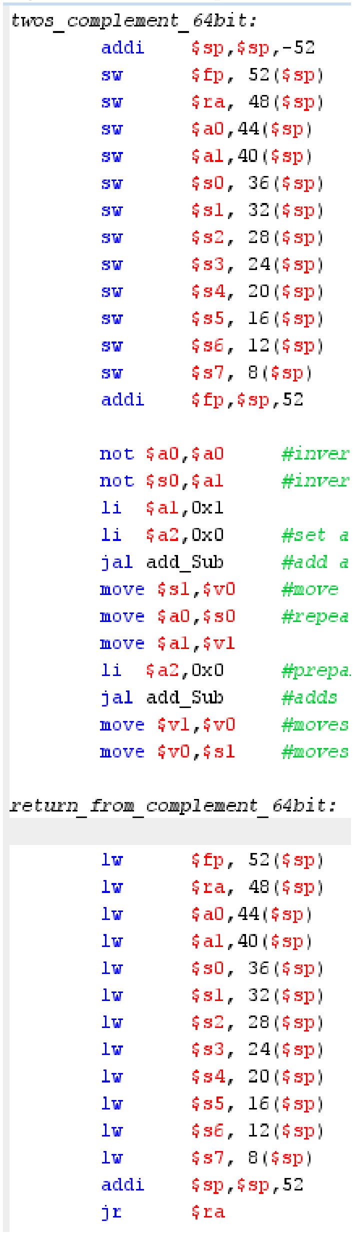


```
twos_complement_if_neg:
        addi    $sp,$sp,-12
        sw      $fp, 12($sp)
        sw      $ra, 8($sp)
        addi    $fp,$sp,12

        bltz    $a0,get_Twos_complement
        move     $v0,$a0
        beqz    $a0,return_from_twos_complement_if_neg
        bgtz    $a0,return_from_twos_complement_if_neg

get_Twos_complement:

        jal twos_complement

return_from_twos_complement_if_neg:

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

        jr      $ra
```

Fig 15. Implementation of twos_complement_if_neg

4)*twos_complement_64bit*

Takes in $a0 and $a1 as the Lo and Hi registers of a 64 bit number and returns $v0 and $v1 as the Lo and Hi registers of the complemented 64 bit value

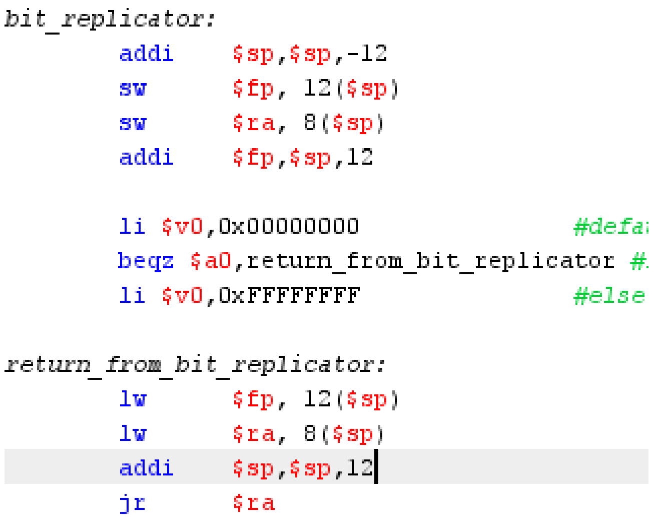


```
twos_complement_64bit:
        addi    $sp,$sp,-52
        sw      $fp, 52($sp)
        sw      $ra, 48($sp)
        sw      $a0,44($sp)
        sw      $a1,40($sp)
        sw      $s0, 36($sp)
        sw      $s1, 32($sp)
        sw      $s2, 28($sp)
        sw      $s3, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        addi    $fp,$sp,52

        not $a0,$a0       #inver
        not $s0,$a1       #inver
        li  $a1,0x1
        li  $a2,0x0       #set a
        jal add_Sub       #add a
        move $s1,$v0      #move
        move $a0,$s0      #repea
        move $a1,$v1
        li  $a2,0x0       #prepa
        jal add_Sub       #adds
        move $v1,$v0      #moves
        move $v0,$s1      #moves

return_from_complement_64bit:

        lw      $fp, 52($sp)
        lw      $ra, 48($sp)
        lw      $a0,44($sp)
        lw      $a1,40($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,52
        jr      $ra
```

Fig. 16 Implementation of twos_complement_64bit

5) *bit_replicator*

Takes in a single bit 0x0 or 0x1 in $a0 and returns 0x00000000 or 0xFFFFFFFF respectively in $v0



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

        li $v0,0x00000000                  #defa
        beqz $a0,return_from_bit_replicator #
        li $v0,0xFFFFFFFF                  #else

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

Fig.17 Implementation of bit_replicator

6) *mul_unsigned*

Multiplies the 32 bit numbers passed in through $a0 and $a1 using unsigned multiplication. It returns the Lo 32 bits in $v0 and the Hi 32 bits in $v1. The logical design of the procedure is as follows:

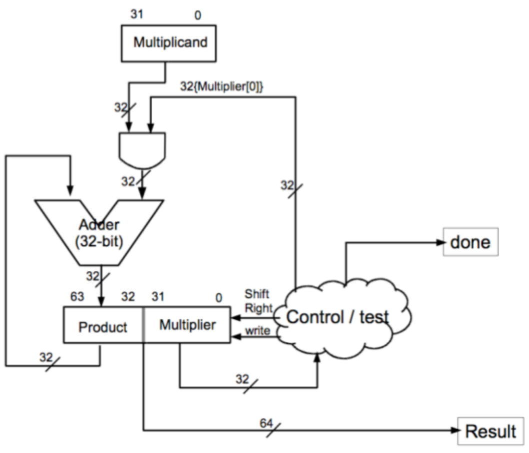


Fig.18 Logical implementation of mul_unsigned

The multiplicand is AND'd with the 0th bit of the multiplier 32 times to get every bit in the register . Since binary multiplication is logically equivalent to the and operation

AND OPERATION FOR BINARY MULTIPLICATION

A	B	A AND B	A*B
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

The result of the 32 and operations are all added up using the full adder. Each time the multiplier finishes a round of 32 multiplications, it is right shifted to remove the used bit, until all 32 bits of the multiplier are used and removed and the result of the multiplication is 64 bits.

The figure below shows the flowchart for the loop:

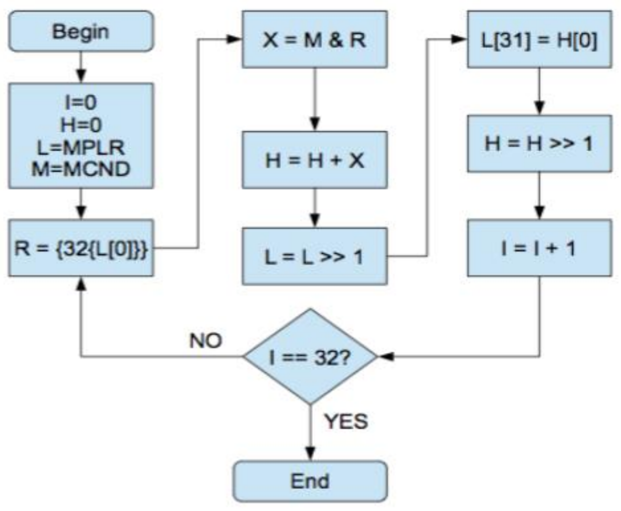
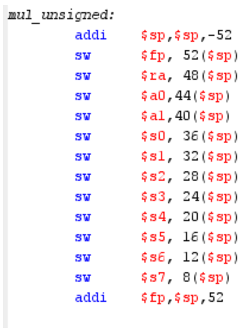


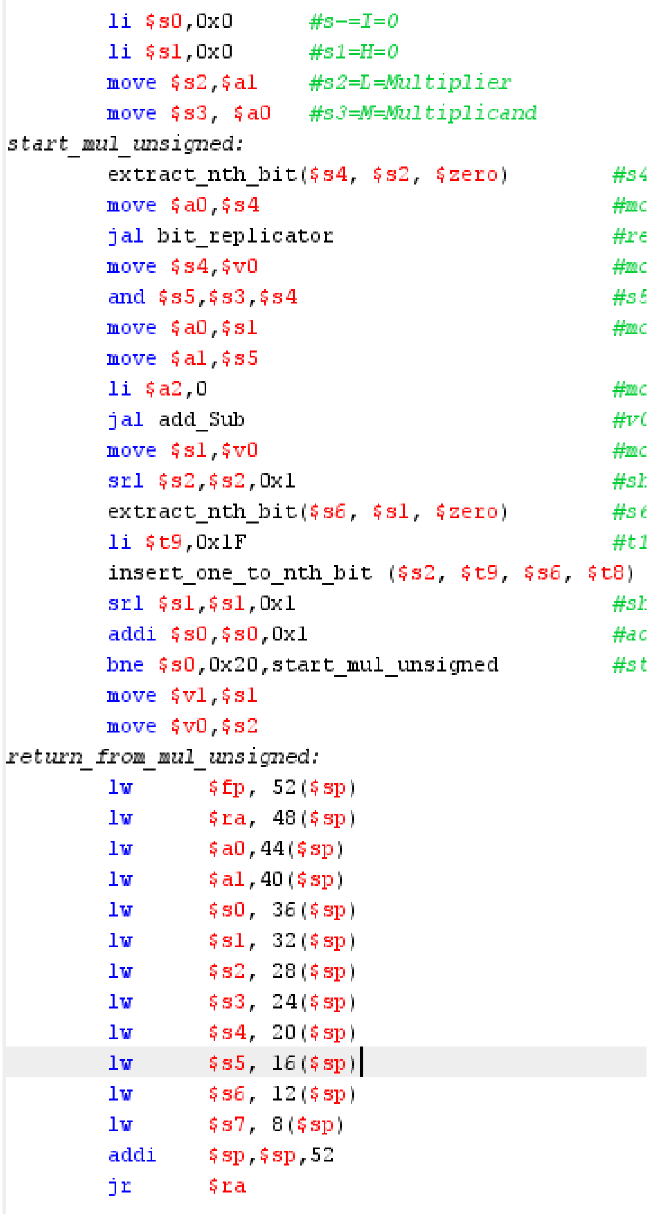
Figure 19. Flowchart for mul_unsigned

The multiplier must loop 32 times and store its results in two 32 bit registers, $v0 and $v1. The loop runs by doing extending the first bit of the multiplier to 32 bits, ANDing it with the multiplicand, and then adding it to the result register. Then it right shifts the multiplier to remove the last used bit and then inserts the LSB of the product into the MSB of the multiplier, to use the freed space in the multiplier. It then continues the loop for 32 times, using until the L Multiplier register is the Lo register for the product.

The implementation in code is shown to the right.



```
mul_unsigned:
        addi    $sp,$sp,-52
        sw      $fp, 52($sp)
        sw      $ra, 48($sp)
        sw      $a0,44($sp)
        sw      $a1,40($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,52
```



```
        li $s0,0x0       #s-=I=0
        li $s1,0x0       #s1=H=0
        move $s2,$a1     #s2=L=Multiplier
        move $s3, $a0    #s3=M=Multiplicand
start_mul_unsigned:
        extract_nth_bit($s4, $s2, $zero)        #s4
        move $a0,$s4                            #mc
        jal bit_replicator                      #re
        move $s4,$v0                            #mc
        and $s5,$s3,$s4                         #s5
        move $a0,$s1                            #mc
        move $a1,$s5
        li $a2,0                                #mc
        jal add_Sub                             #v(
        move $s1,$v0                            #mc
        srl $s2,$s2,0x1                         #sl
        extract_nth_bit($s6, $s1, $zero)        #s6
        li $t9,0x1F                             #t1
        insert_one_to_nth_bit ($s2, $t9, $s6, $t8)
        srl $s1,$s1,0x1                         #sl
        addi $s0,$s0,0x1                        #ac
        bne $s0,0x20,start_mul_unsigned         #st
        move $v1,$s1
        move $v0,$s2
return_from_mul_unsigned:
        lw      $fp, 52($sp)
        lw      $ra, 48($sp)
        lw      $a0,44($sp)
        lw      $a1,40($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,52
        jr      $ra
```

Fig 20. Implementation of mul_unsigned

7) *mul_signed*

The procedure does multiplication and takes into account the possibility of positive and negative values being passed in through $a0 and $a1 by running twos_complement_if_negative to convert them to positive values and then running 32 bit unsigned multiplier to get the value. To properly finish the calculation, it runs twos_complement_64 bit if the result needs to be negative

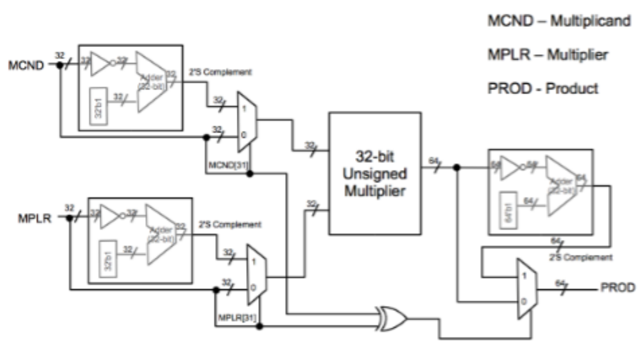


Fig. 21 Logic Design of mul_signed

Below is the logic for converting negative operands

```
move $s6,$a0
move $s7,$a1
li $t4,0x1F
extract_nth_bit($s3, $a0, $t4)
extract_nth_bit($s4, $a1, $t4)
xor $s5,$s3,$s4

jal twos_complement_if_neg
move $s0,$v0
move $a0,$s7
jal twos_complement_if_neg
move $s1,$v0
```

Fig.22 Implementation of converting negative operands

The sign of the 64 bit product can be determined by taking the XOR of the MSB of the two operands in registers $a0 and $a1 because the MSB determines the sign of the number, 1 means its negative and 0 means its positive. This mirrors the XOR operation's truth table

XOR OPERATION FOR DETERMINING SIGNS

A	B	A XOR B	SIGNS
0	0	0	- * - = +
0	1	1	- * + = -
1	0	1	+ * - = -
1	1	0	+ * + = +

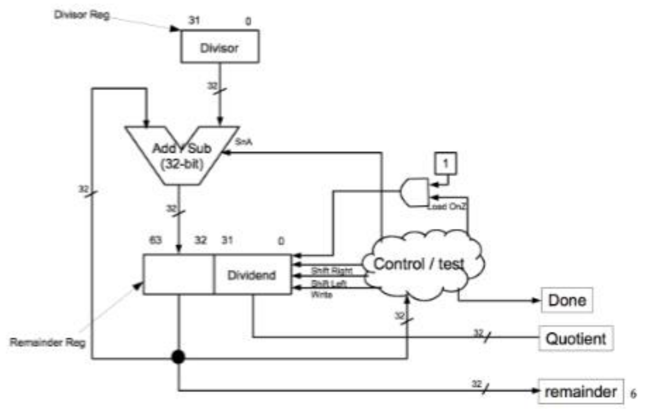
To the right is the full implementation of mul_signed

```
mul_signed:
        addi    $sp,$sp,-52
        sw      $fp, 52($sp)
        sw      $ra, 48($sp)
        sw      $a0,44($sp)
        sw      $a1,40($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,52
        move $s6,$a0
        move $s7,$a1
        li $t4,0x1F
        extract_nth_bit($s3, $a0, $t4)
        extract_nth_bit($s4, $a1, $t4)
        xor $s5,$s3,$s4

        jal twos_complement_if_neg
        move $s0,$v0
        move $a0,$s7
        jal twos_complement_if_neg
        move $s1,$v0

        move $a0,$s0
        move $a1,$s1
        jal mul_unsigned

        beqz $s5,return_from_mul_signed
        move $a0,$v0
        move $a1,$v1
        jal twos_complement_64bit

return_from_mul_signed:

        lw      $fp, 52($sp)
        lw      $ra, 48($sp)
        lw      $a0,44($sp)
        lw      $a1,40($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,52
        jr      $ra
```

Fig. 22 Implementation of mul_signed

8)*div_unsigned*
The procedure takes in $a0 as the dividend and $a1 as the divisor. It returns $v0 as the quotient and $v1 as the remainder, all in 32 bits.

Below is the logical design of div_unsigned:



div_unsigned uses two registers to simulate a 64 bit register by shifting out the dividend and shifting in the quotient. The loop itself subtracts the divedend from the divisor from the MSB. If the subtraction is positive, the quotient gets a inserted into it and if it is negative, a 0 is inserted. This continues until the divedend is shifted out and replaced by the quotient.

Below is the flowchart of div_unsigned:

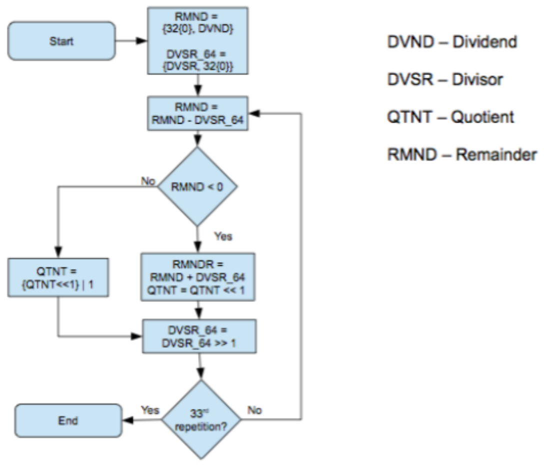


Fig 24 Flowchart of div_unsigned

A loop that runs 32 times is created that shifts out the dividend in the soon to be remainder register to the left and shifts in the quotient's MSB to simulate a 64 bit register. The result ends up in one register with the remainder and quotient in one.

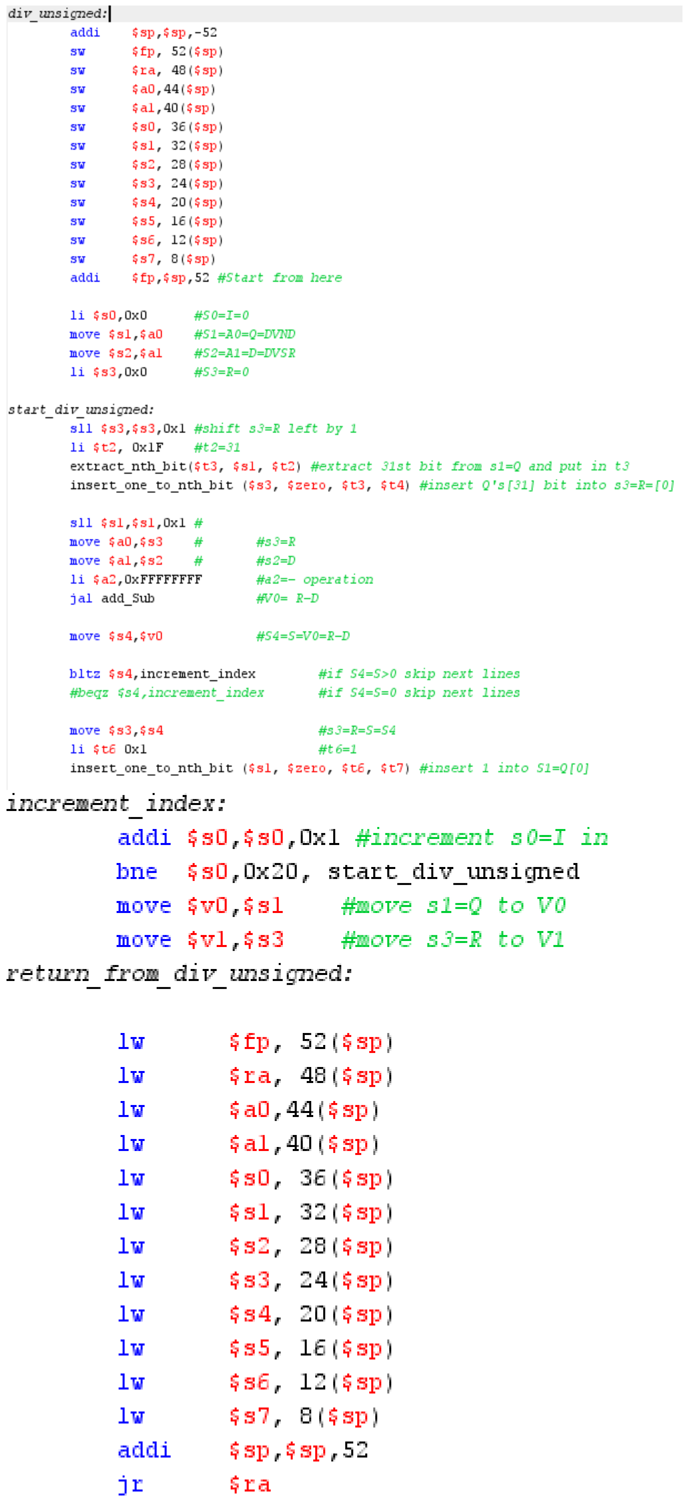

```
div_unsigned:
        addi    $sp,$sp,-52
        sw      $fp, 52($sp)
        sw      $ra, 48($sp)
        sw      $a0,44($sp)
        sw      $a1,40($sp)
        sw      $s0, 36($sp)
        sw      $s1, 32($sp)
        sw      $s2, 28($sp)
        sw      $s3, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        addi    $fp,$sp,52 #Start from here

        li $s0,0x0      #S0=I=0
        move $s1,$a0    #S1=A0=Q=DVND
        move $s2,$a1    #S2=A1=D=DVSR
        li $s3,0x0      #S3=R=0

start_div_unsigned:
        sll $s3,$s3,0x1 #shift s3=R left by 1
        li $t2, 0x1F    #t2=31
        extract_nth_bit($t3, $s1, $t2) #extract 31st bit from s1=Q and put in t3
        insert_one_to_nth_bit ($s3, $zero, $t3, $t4) #insert Q's[31] bit into s3=R=[0]

        sll $s1,$s1,0x1 #
        move $a0,$s3    #       #s3=R
        move $a1,$s2    #       #s2=D
        li $a2,0xFFFFFFFF       #a2=- operation
        jal add_Sub             #V0= R-D

        move $s4,$v0            #S4=S=V0=R-D

        bltz $s4,increment_index        #if S4=S>0 skip next lines
        #beqz $s4,increment_index       #if S4=S=0 skip next lines

        move $s3,$s4                    #s3=R=S=S4
        li $t6 0x1                      #t6=1
        insert_one_to_nth_bit ($s1, $zero, $t6, $t7) #insert 1 into S1=Q[0]
increment_index:
        addi $s0,$s0,0x1 #increment s0=I in
        bne  $s0,0x20, start_div_unsigned
        move $v0,$s1    #move s1=Q to V0
        move $v1,$s3    #move s3=R to V1
return_from_div_unsigned:

        lw      $fp, 52($sp)
        lw      $ra, 48($sp)
        lw      $a0,44($sp)
        lw      $a1,40($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,52
        jr      $ra
```

Fig. 25. Implementation of Loop in Unsigned Division

10) div_signed

This procedure takes in $a0 and $a1 as 32 bit dividend and divisor, which can be negative or positive, and divides them, storing the resulting quotient in $v0 and $v1

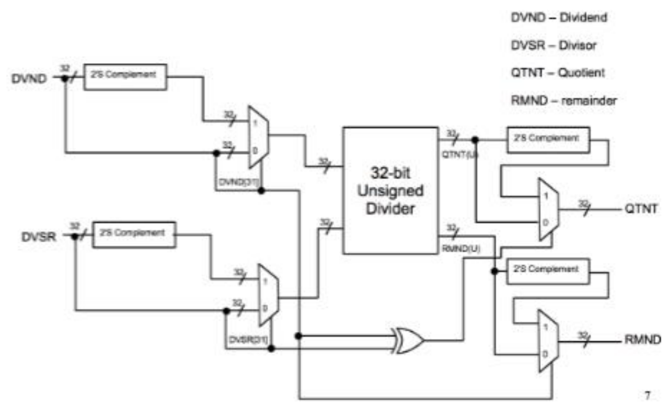
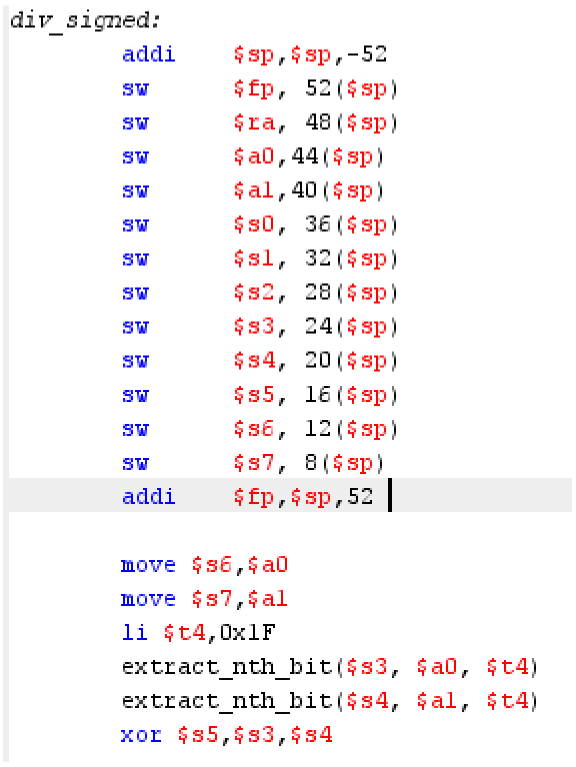
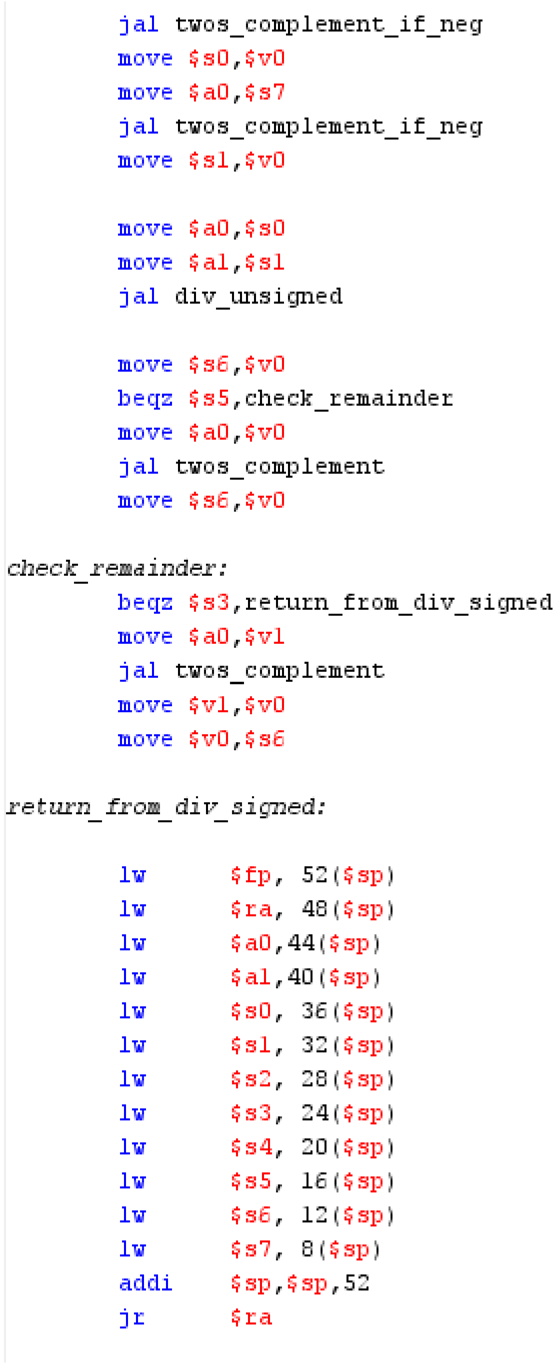


Fig. 26. Logic design of Signed Division

The divisor and dividend are converted to positive numbers using twos_complement_if_negative and the results of div_unsigned are converted to their proper sign. Quotient by taking the XOR of the MSB of the divisor and dividend, using a similar logic to mul_signed. If the dividend is negative, the remainder is negative.



```
div_signed:
        addi    $sp,$sp,-52
        sw      $fp, 52($sp)
        sw      $ra, 48($sp)
        sw      $a0,44($sp)
        sw      $a1,40($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,52

        move $s6,$a0
        move $s7,$a1
        li $t4,0x1F
        extract_nth_bit($s3, $a0, $t4)
        extract_nth_bit($s4, $a1, $t4)
        xor $s5,$s3,$s4
```



```
        jal twos_complement_if_neg
        move $s0,$v0
        move $a0,$s7
        jal twos_complement_if_neg
        move $s1,$v0

        move $a0,$s0
        move $a1,$s1
        jal div_unsigned

        move $s6,$v0
        beqz $s5,check_remainder
        move $a0,$v0
        jal twos_complement
        move $s6,$v0

check_remainder:
        beqz $s3,return_from_div_signed
        move $a0,$v1
        jal twos_complement
        move $v1,$v0
        move $v0,$s6

return_from_div_signed:

        lw      $fp, 52($sp)
        lw      $ra, 48($sp)
        lw      $a0,44($sp)
        lw      $a1,40($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,52
        jr      $ra
```

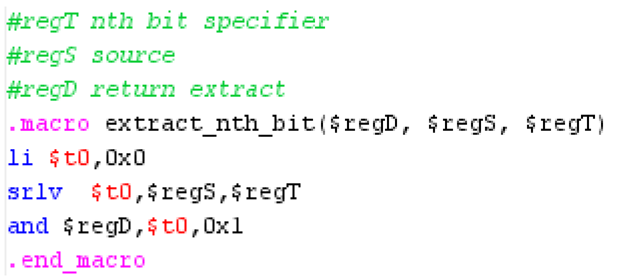
Fig. 27. Implementation of Signed Division

C. Macros

Every procedure/design requires extraction of bits and insertion of bits, so the following two macros were created

1) Extract_nth_bi

Right shift the value in $regS by the value in $regT and masks it using a 0x1 as an immediate mask and assigns it to $regD

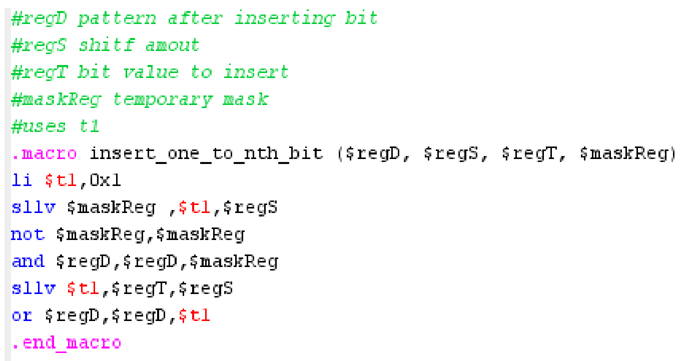


```
#regT nth bit specifier
#regS source
#regD return extract
.macro extract_nth_bit($regD, $regS, $regT)
li $t0,0x0
srlv  $t0,$regS,$regT
and $regD,$t0,0x1
.end_macro
```

Fig. 28 Implementation of extract_nth_bit

2) *Insert_one_to_nth_bit*

Creates a mask by putting 1 into a register and shifting it to the left to create a mask of the correct length. Then it inverts the mask and uses the inverted mask to get every value except the target bit to insert. Then it left shifts the $regT by $regS and then OR's the result to insert the bit into the pattern



```
#regD pattern after inserting bit
#regS shitf amout
#regT bit value to insert
#maskReg temporary mask
#uses t1
.macro insert_one_to_nth_bit ($regD, $regS, $regT, $maskReg)
li $t1,0x1
sllv $maskReg ,$t1,$regS
not $maskReg,$maskReg
and $regD,$regD,$maskReg
sllv $t1,$regT,$regS
or $regD,$regD,$t1
.end_macro
```

Fig. 29 Implementation of insert_one_to_nth bit

D. Overall Diagram

Au_normal uses the following procedures

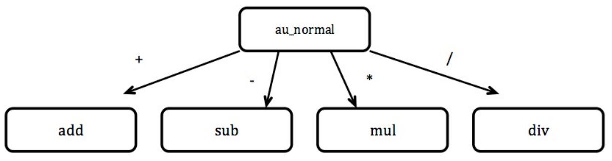


Fig. 30. Normal Procedure and its Operations

Au_logical does the following:

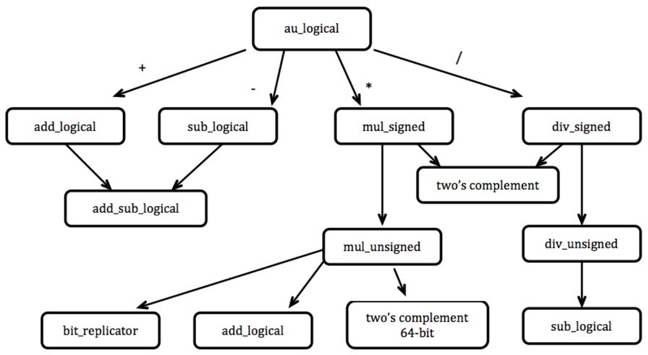
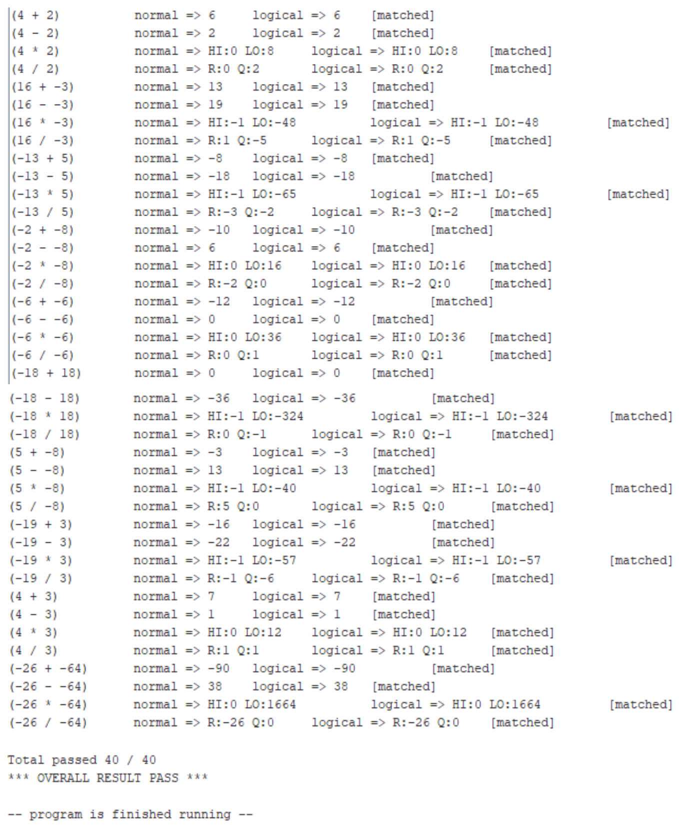


Fig. 31. Logical Procedure and its Corresponding Procedures

V.TESTING

Save all files, assemble proj_auto and click the run button. There are 40 operations to run with 10 combinations of operands that should result in a 40/40 if normal and logical match each other



```
(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. 32. Successful Test Results

Common errors are include:

1) The use of temporary registers

Temporary registers may be overwritten by macros, tracking temporary register use and limiting temporary register usage can decrease the likelihood of encountering this problem

2) Storing and restoring frame

Improper frame storage can result in $pc counter errors or overwriting values return registers. Proper frame storage conventions can fix this error.

3) Errors in branching logic

If the branching logic has a condition swapped, the values can be the opposite of what is expected, and once these errors are compounded, they can be very hard to find.

VI. CONCLUSION

This project taught me about using MARS and debugging using the MARS simulator. Using stack and frame restores have become easier with practice and I have learned the importance of tracking your registers and checking your overwrites. Debugging these errors has taught me about how unchecked MIPS is compared to high level coding and has given me a new found understanding of binary addition, multiplication, division, and subtraction

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.