Arithmetic Implemented By MIPS Logic Operations

Joshua Liang, San Jose State University, joshua.liang@sjsu.edu

***Abstract*—This report contains the information and implementation of the basic mathematical operations (addition, subtraction, multiplication, and division) in MIPS normal procedures and logical procedures in MIPS Assembler and Runtime Simulator (MARS).**

I. Introduction

We will use MARS to perform and calculate mathematical operations in two different methods. One method will be to use MIPS normal procedures such as addition, subtraction, multiplication, and division. The second method will be to use logical procedures such as Boolean logic (AND, OR, NOT). The three main goals of this project are:

1. Download, install, and set up MARS.
2. Implement arithmetic operations using MIPS normal procedures and MIPS logical procedures.
3. Test the implementation of the MIPS procedures.

II. Requirements

A. MARS Installation

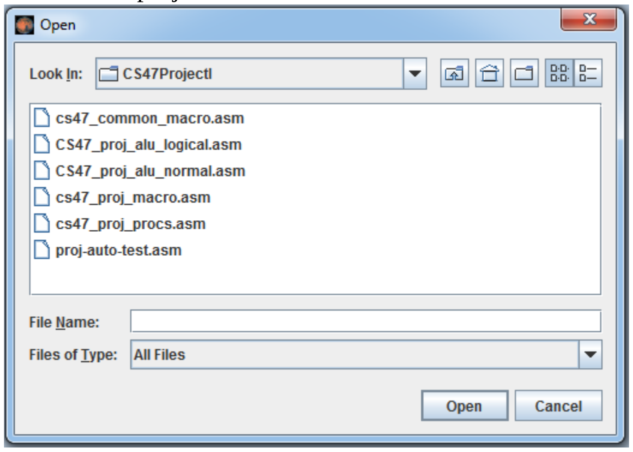
Visit the following website to download MARS:
http://courses.missouristate.edu/KenVollmar/mars/download.htm
Click on the "Download MARS" button to begin downloading the simulator.

B. Project Files

Download CS47ProjectI.zip and unzip it from the following site:
https://sjsu.instructure.com/courses/1242233/assignments/4498130
The following six files should be unzipped into a directory:

1. *cs47_common_macro.asm*
   This contains macros for printing out test results
2. *cs47_proj_alu_logical.asm*
   This contains logical operations for arithmetic operations
3. *cs47_proj_alu_normal.asm*
   This contains normal procedures arithmetic operations
4. *cs47_proj_macro.asm*
   This contains macros that will be written for logical procedures
5. *cs47_proj_procs.asm*
   This contains project procedures
6. *proj-auto-test.asm*
   This contains the testing procedure

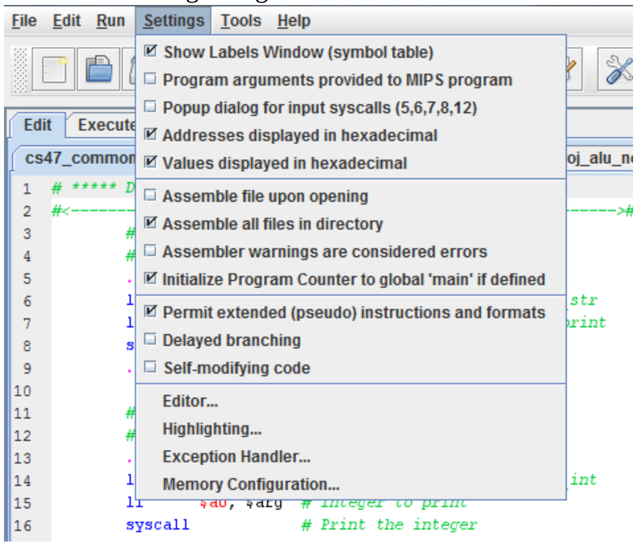
Open MARS and click on "File" at the top left corner. Then click "open" and navigate to the extracted folder that contains the six project files. Load all of the files.



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

Figure 1. Opening Project Files

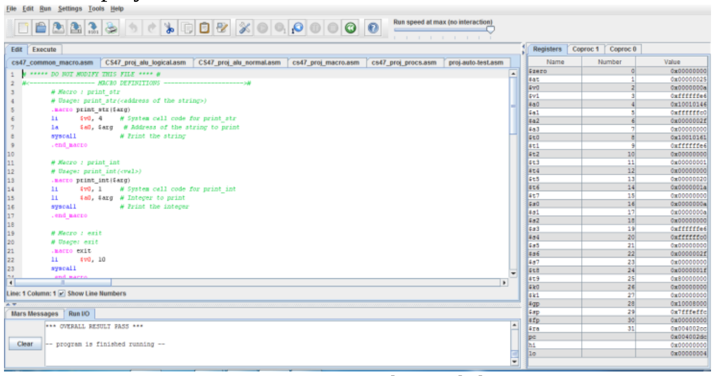
After loading the project files, go to the top of the menu and select "settings". Make sure that everything is checked or unchecked according to Figure 2 shown below.



- ✓ 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

Figure 2. Settings

After loading all project files and configuring the settings, each file can be seen and editable. The file names should also be displayed.


File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
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
Line: 1 Column: 1 Show Line Numbers
Mars Messages Run I/O
Clear
Registers Coproc 1 Coproc 0

```
# ***** DO NOT MODIFY THIS FILE **** #
#<------------------ MACRO DEFINITIONS ---------------------->#
        # Macro : print_str
        # Usage: print_str(<address of the string>)
        .macro print_str($arg)
        li      $v0, 4     # System call code for print_str
        la      $a0, $arg   # Address of the string to print
        syscall            # Print the string
        .end_macro

        # Macro : print_int
        # Usage: print_int(<val>)
        .macro print_int($arg)
        li      $v0, 1     # System call code for print_int
        li      $a0, $arg  # Integer to print
        syscall            # Print the integer
        .end_macro

        # Macro : exit
        # Usage: exit
        .macro exit
        li      $v0, 10
        syscall
```

```
*** OVERALL RESULT PASS ***

-- program is finished running --
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000025
$v0	2	0x0000000a
$v1	3	0xffffffe6
$a0	4	0x10010146
$a1	5	0xffffffc0
$a2	6	0x0000002f
$a3	7	0x00000000
$t0	8	0x10010161
$t1	9	0xffffffe6
$t2	10	0x00000000
$t3	11	0x00000001
$t4	12	0x00000000
$t5	13	0x00000020
$t6	14	0x0000001a
$t7	15	0x00000000
$s0	16	0x0000000a
$s1	17	0x0000000a
$s2	18	0x00000000
$s3	19	0xffffffe6
$s4	20	0xffffffc0
$s5	21	0x00000000
$s6	22	0x0000002f
$s7	23	0x00000000
$t8	24	0x0000001f
$t9	25	0x80000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x004002cc
pc		0x004002dc
hi		0x00000000
lo		0x00000004

Figure 3. Files Loaded

III. ARITHMETIC OPERATION DESCRIPTION

The arithmetic operations will be implemented in two methods, in normal operations and in logical operations.

A. *Normal Procedures*

The normal procedure which is named au_normal (in CS47_proj_alu_ normal.asm). It takes three arguments:

1) *$a0 (First operand)*
2) *$a1 (Second operand)*
3) *$a2 (Operation code '+', '-', '*', '/' - ASCII code).*

It returns result in $v0 and $v1 (for multiplication $v1 it will contain HI, for division $v1 will contain remainder). This procedure uses normal math operations of MIPS to compute the result (add, sub, mul and div).

B. *Logical Procedures*

The logical procedures which is named au_logical (in CS47_proj_alu_logical.asm). It takes three arguments:

1) *$a0 (First operand)*
2) *$a1 (Second operand)*
3) *$a2 (Operation code '+', '-', '*', '/' - ASCII code)*

It returns result in $v0 and $v1 (for multiplication $v1 it will contain HI, for division $v1 will contain remainder). The evaluation of mathematical operations should use MIPS logic operations only (result should not be generated directly using MIPS mathematical operations). The implementation needs to follow the digital algorithm implemented in hardware to implement the mathematical operations.

IV. DESIGN AND IMPLEMENTATION OF OPERATIONS.

A. *Normal Procedure*

Depending on the operator in $a2 refers to, we will branch out to the procedures that will perform the corresponding calculations.

```
        li      $t0, '+'
        li      $t1, '-'
        li      $t2, '*'
        li      $t3, '/'

        beq     $a2, $t0, ADDITION
        beq     $a2, $t1, SUBTRACTION
        beq     $a2, $t2, MULTIPLCATION
        beq     $a2, $t3, DIVISION

        j       RETURN
```

Figure 4. Branch and Operation Implementation

The four following operators are what we will be branching out to:

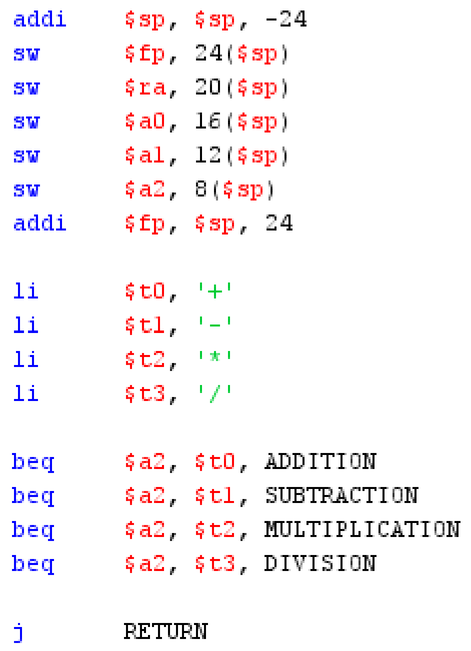
1) *$t0 which is Operator '+'* Addition
2) *$t1 which is Operator '-'* Subtraction
3) *$t2 which is Operator '*'* Multiplication
4) *$t3 which is Operator '/'* Division

```
ADDITION:
        addu    $v0, $a0, $a1
        j       RETURN

SUBTRACTION:
        subu    $v0, $a0, $a1
        j       RETURN

MULTIPLCATION:
        mult    $a0, $a1
        mflo    $v0
        mfhi    $v1
        j       RETURN

DIVISION:
        div     $a0, $a1
        mflo    $v0
        mfhi    $v1
        j       RETURN
```

Figure 5. Normal Procedures Implementation

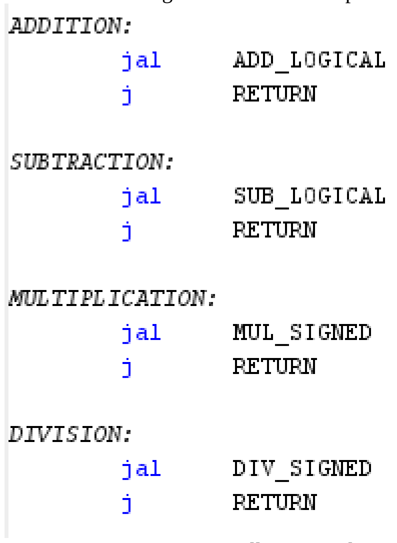
B. *Logical Procedure*

The au_logical procedure is very similar to the normal procedure. The difference is that it will be calling multiple procedures and these other procedures will be called with the appropriate math operations that will be used.



```
addi    $sp, $sp, -24
sw      $fp, 24($sp)
sw      $ra, 20($sp)
sw      $a0, 16($sp)
sw      $a1, 12($sp)
sw      $a2, 8($sp)
addi    $fp, $sp, 24

li      $t0, '+'
li      $t1, '-'
li      $t2, '*'
li      $t3, '/'

beq     $a2, $t0, ADDITION
beq     $a2, $t1, SUBTRACTION
beq     $a2, $t2, MULTIPLICATION
beq     $a2, $t3, DIVISION

j       RETURN
```

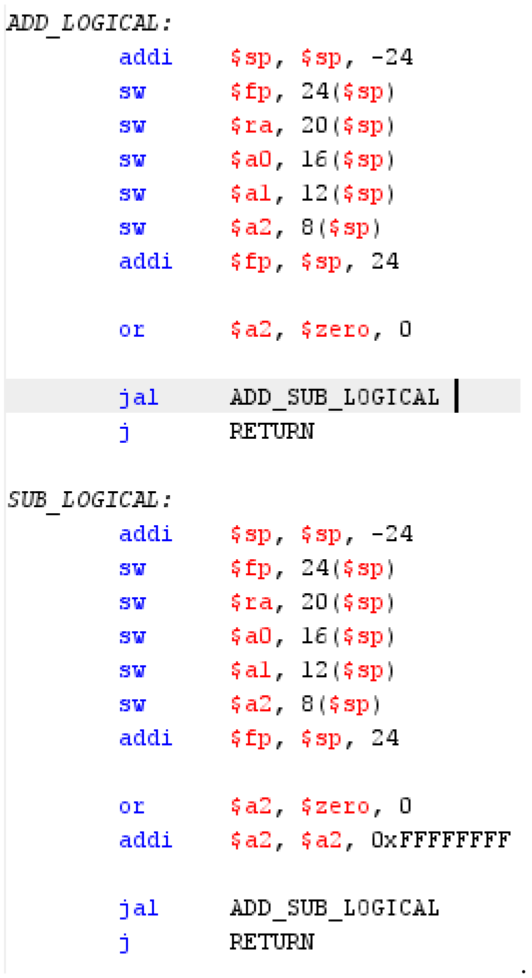
Figure 6. Branch and Operation Implementation



```
ADDITION:
        jal     ADD_LOGICAL
        j       RETURN

SUBTRACTION:
        jal     SUB_LOGICAL
        j       RETURN

MULTIPLICATION:
        jal     MUL_SIGNED
        j       RETURN

DIVISION:
        jal     DIV_SIGNED
        j       RETURN
```

Figure 7. Calling Procedures Implementation

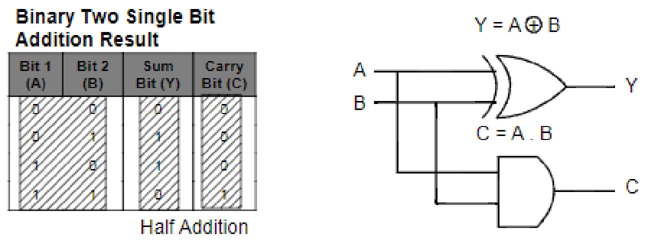
1) *Addition and Subtraction*
ADD_LOGICAL and SUB_LOGICAL both call ADD_SUB_LOGICAL as shown in Figure 8.



```
ADD_LOGICAL:
        addi    $sp, $sp, -24
        sw      $fp, 24($sp)
        sw      $ra, 20($sp)
        sw      $a0, 16($sp)
        sw      $a1, 12($sp)
        sw      $a2, 8($sp)
        addi    $fp, $sp, 24

        or      $a2, $zero, 0

        jal     ADD_SUB_LOGICAL
        j       RETURN

SUB_LOGICAL:
        addi    $sp, $sp, -24
        sw      $fp, 24($sp)
        sw      $ra, 20($sp)
        sw      $a0, 16($sp)
        sw      $a1, 12($sp)
        sw      $a2, 8($sp)
        addi    $fp, $sp, 24

        or      $a2, $zero, 0
        addi    $a2, $a2, 0xFFFFFFFF

        jal     ADD_SUB_LOGICAL
        j       RETURN
```

Figure 8. ADD_LOGICAL & SUB_LOGICAL

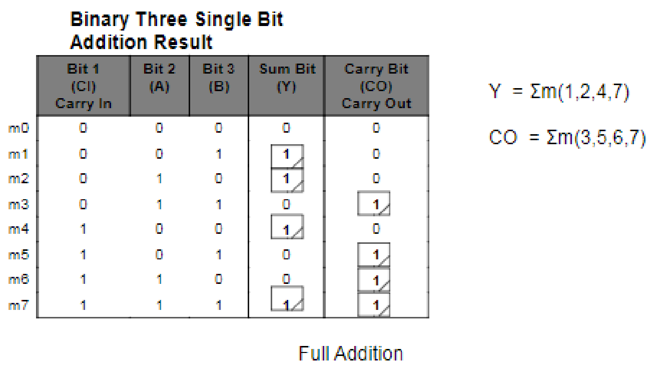
ADD_SUB_LOGICAL calculates the sum of arguments $a0 and $a1. The third argument determines whether it will be addition or subtraction. Addition will be determined by 0, and subtraction will be determined by 0xFFFFFFFF. In MIPS, we use a binary system to add numbers. We use the Half Adder design and the Full Adder design to do so.



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

Figure 9. Half Adder Design

The Half Adder design performs single bit addition while only using the carry-out bit. The diagram shows that the sum can be determined with an AND operation, and the carry-out can be determined with an XOR operation.



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

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

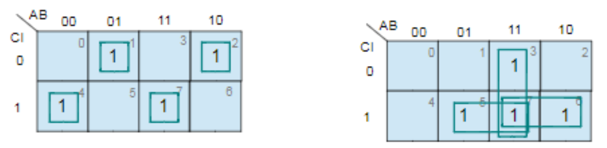
$CO = \Sigma m(3,5,6,7)$

Full Addition

Figure 10. Full Adder Design

The Full Adder design will take both the carry-in bit and the carry-out bit into consideration, as opposed to the Half Adder design. Figure 10 above shows that the Full Adder design is just two Half Adders. The truth table is the same as the Half Adder, but considers the carry-in bit as well.

The logical design of the Full Adder must be determined with the use of a K-MAP. A K-MAP will reduce the terms for simplification. Figure 11 below show us how the K-MAP is used.



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

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

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

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

(1) $= CI.B + CI.A + A.B$

(2) $= CI.\ (A + B) + A.B$

(3) $= CI.(A'.B+A.B') + A.B$

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

Figure 11. K-MAP

These reduced terms will then help us draw our logical design for addition and subtraction. Figure 12. below shows us the logical diagram.

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

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

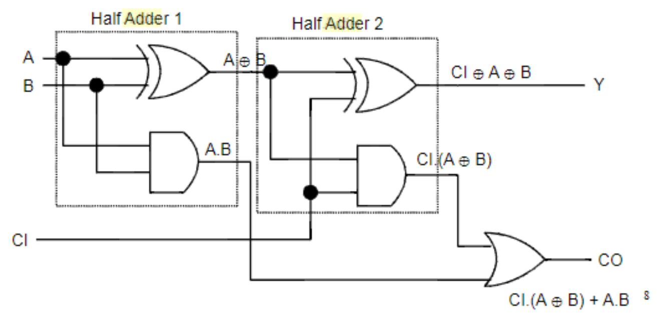
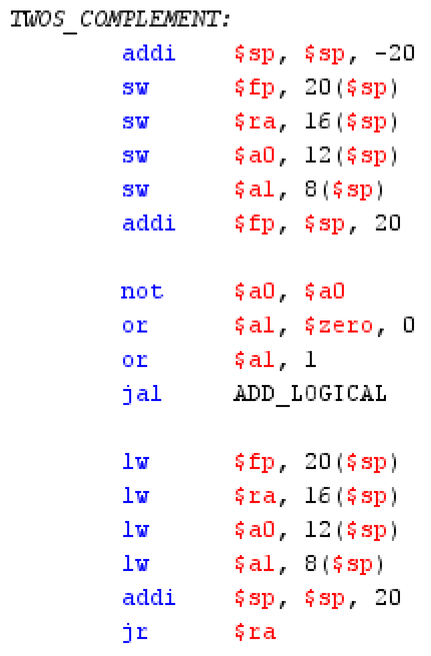


Figure 12. Logic Diagram

The sum of the operands' bits is determined by the XOR operation involving the carry-in bit and the first bit of both the the first operand and the second operand. The final carry-out bit is determined by the OR operation involving the carry-out bits from both the two Half Adders. The sum can be either addition or subtraction because subtraction is the same as the addition of a negative number. Therefore, the two's complement of the second operand will be required to perform subtraction . The equation below will return the two's complement of a number:

$a0 = ~$a0 + 1

Then, $a2 is simply used as a submode operator. The two's complement form of $a1 can be determined by adding 1 to NOT $a1 as shown in Figure 13 below.



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

        not     $a0, $a0
        or      $a1, $zero, 0
        or      $a1, 1
        jal     ADD_LOGICAL

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

Figure 13. TWOS_COMPLEMENT

To determine if $a1's two's complement is needed, we simply just treat the subtraction as an addition. A loop will be run through the operands' bits to add two 32-bit number. The loop will use the Full Adder design to add their bits and factor in their carry bits. This is shown in Figure 14 below.

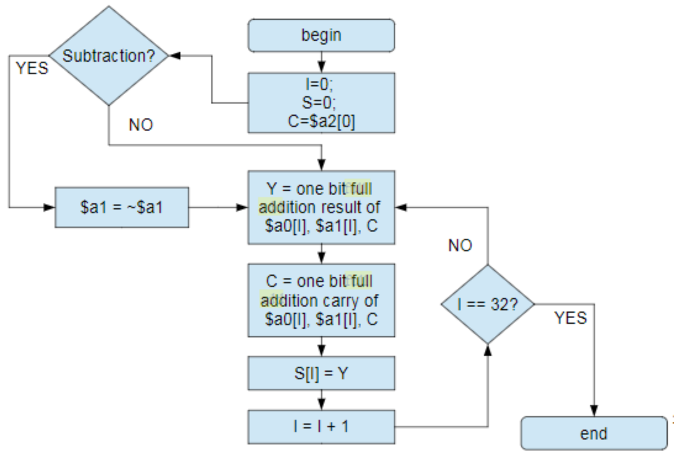

begin
I=0;
S=0;
C=$a2[0]
Subtraction?
YES
NO
$a1 = ~$a1
Y = one bit full addition result of $a0[I], $a1[I], C
C = one bit full addition carry of $a0[I], $a1[I], C
S[I] = Y
I = I + 1
I == 32?
NO
YES
end

Figure 14. Addition/Subtraction Flowchart

We can copy this flowchart to create our implementation of ADD_SUB_LOGICAL as seen in Figure 15 below.

```
ADD_SUB_LOGICAL:
        addi    $sp, $sp, -40
        sw      $fp, 40($sp)
        sw      $ra, 36($sp)
        sw      $a0, 32($sp)
        sw      $a1, 28($sp)
        sw      $a2, 24($sp)
        sw      $s0, 20($sp)
        sw      $s1, 16($sp)
        sw      $s2, 12($sp)
        sw      $s3, 8($sp)
        addi    $fp, $sp, 40

        or      $t0, $zero, 0
        or      $t1, $zero, 0
        or      $t2, $zero, 0
        extract_nth_bit($t2, $a2, $zero)
        beq     $a2, 0, ADD_SUB_LOGICAL_1
        not     $a1, $a1

ADD_SUB_LOGICAL_1:
        beq     $t0, 32, ADD_SUB_LOGICAL_EXIT
        extract_nth_bit($t3, $a0, $t0)
        extract_nth_bit($t4, $a1, $t0)
        xor     $s0, $t3, $t4
        xor     $s1, $t2, $s0
        and     $s2, $t3, $t4
        and     $s3, $t2, $s0
        or      $t2, $s2, $s3
        insert_to_nth_bit($v0, $t0, $s1, $t9)
        addi    $t0, $t0, 1

        j       ADD_SUB_LOGICAL_1

ADD_SUB_LOGICAL_EXIT:
        move    $v1, $t2

        lw      $fp, 40($sp)
        lw      $ra, 36($sp)
        lw      $a0, 32($sp)
        lw      $a1, 28($sp)
        lw      $a2, 24($sp)
        lw      $s0, 20($sp)
        lw      $s1, 16($sp)
        lw      $s2, 12($sp)
        lw      $s3, 8($sp)
        addi    $sp, $sp, 40
        jr      $ra
```

Figure 15. ADD_SUB_LOGICAL

ADD_LOGICAL calls ADD_SUB_LOGICAL with $a2 as 0. On the contrary, SUB_LOGICAL will do the same, except call $a2 as 0xFFFFFFFF. Since $a2 is 0xFFFFFFFF, ADD_SUB_LOGICAL will call the TWOS_COMPLEMENT procedure and return $a1's two's complement for subtraction.

2) *Multiplication*

For multiplication, we will split it into two methods as multiplying positive and negative numbers are very different.

1) *MUL_UNSIGNED*
   Unsigned Multiplication
2) *MUL_SIGNED*
   Signed Multiplication

In our MUL_UNSIGNED implication we will use BIT_REPLICATOR procedure. This procedure will replicate 1 bit to 32 bits. It takes an argument of a bit value (0 or 1), and returns a 32 bit number with the original value replicated 32 times. This is shown in our implementation in Figure 16 below.

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

        or      $v0, $a0, 0
        beq     $a0, 0, BIT_REPLICATOR_EXIT
        li      $v0, 0xFFFFFFFF
```

Figure 16. BIT_REPLICATOR Implementation

The process of unsigned multiplication is shown below in Figure 17.

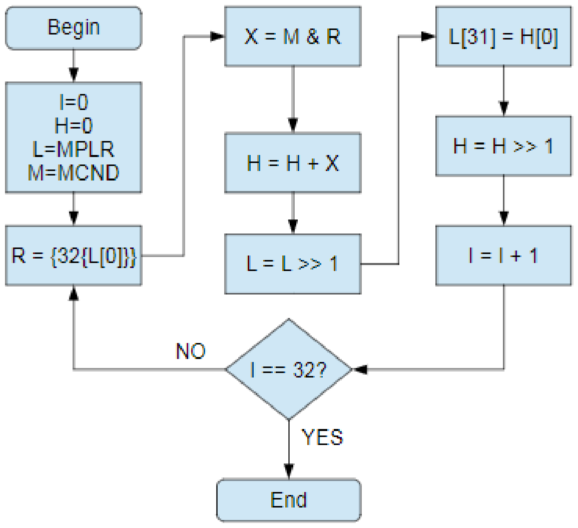
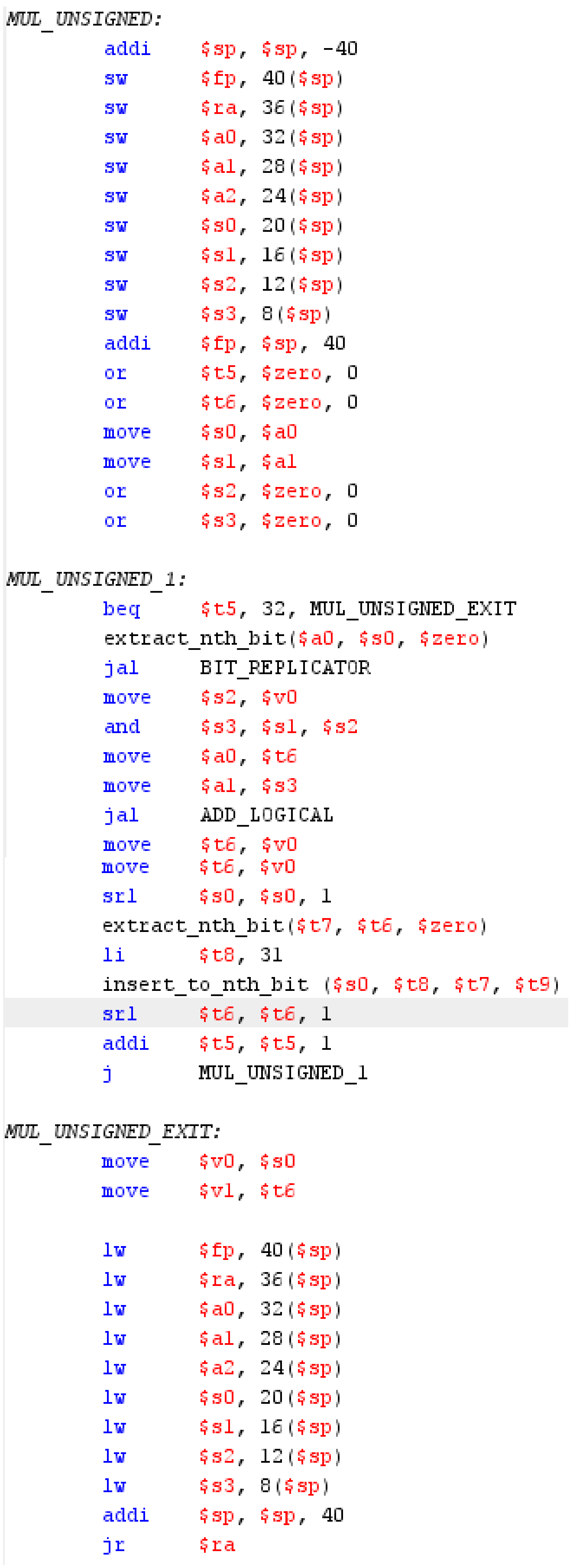


Figure 17. Unsigned Multiplication Flowchart

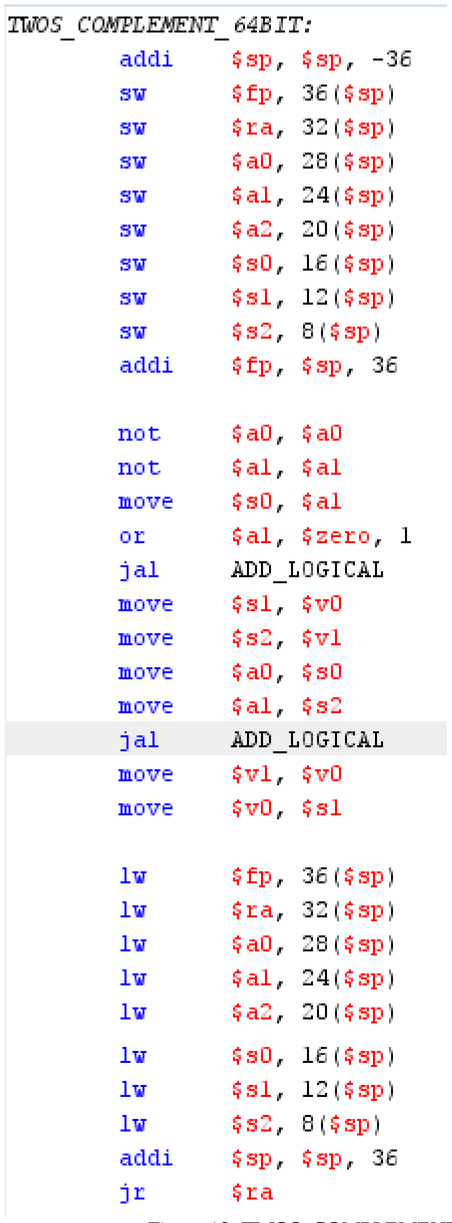
While copying the flowchart in our implementation in Figure 18, a loop will run 32 times because there are 32 bits. The multiplier will shift to the right and the 1st bit in the product register will be extracted and inserted into the 31st register. Following, the product register will also shift.

For MUL_SIGNED, this procedure will take its multiplicand and multiplier and check if they are negative by converting them to their two’s complement form and then calling the MUL_UNSIGNED procedure.
$v0 will contain the LO parts and $v1 will contain the HI parts. A 64-bit number will result from two 32-bit numbers multiplying. A 64-bit number can’t be stored in 1 MIPS register. Thus, the 64-bit must be stored into 2 registers, $v0 and $v1. To do this, we will create a TWOS_COMPLEMENT_64BIT procedure. This procedure will store the results of MUL_SIGNED into 2 registers, $v0 and $v1. The implementation is shown below in Figure 19.



```
MUL_UNSIGNED:
        addi    $sp, $sp, -40
        sw      $fp, 40($sp)
        sw      $ra, 36($sp)
        sw      $a0, 32($sp)
        sw      $a1, 28($sp)
        sw      $a2, 24($sp)
        sw      $s0, 20($sp)
        sw      $s1, 16($sp)
        sw      $s2, 12($sp)
        sw      $s3, 8($sp)
        addi    $fp, $sp, 40
        or      $t5, $zero, 0
        or      $t6, $zero, 0
        move    $s0, $a0
        move    $s1, $a1
        or      $s2, $zero, 0
        or      $s3, $zero, 0

MUL_UNSIGNED_1:
        beq     $t5, 32, MUL_UNSIGNED_EXIT
        extract_nth_bit($a0, $s0, $zero)
        jal     BIT_REPLICATOR
        move    $s2, $v0
        and     $s3, $s1, $s2
        move    $a0, $t6
        move    $a1, $s3
        jal     ADD_LOGICAL
        move    $t6, $v0
        move    $t6, $v0
        srl     $s0, $s0, 1
        extract_nth_bit($t7, $t6, $zero)
        li      $t8, 31
        insert_to_nth_bit ($s0, $t8, $t7, $t9)
        srl     $t6, $t6, 1
        addi    $t5, $t5, 1
        j       MUL_UNSIGNED_1

MUL_UNSIGNED_EXIT:
        move    $v0, $s0
        move    $v1, $t6

        lw      $fp, 40($sp)
        lw      $ra, 36($sp)
        lw      $a0, 32($sp)
        lw      $a1, 28($sp)
        lw      $a2, 24($sp)
        lw      $s0, 20($sp)
        lw      $s1, 16($sp)
        lw      $s2, 12($sp)
        lw      $s3, 8($sp)
        addi    $sp, $sp, 40
        jr      $ra
```

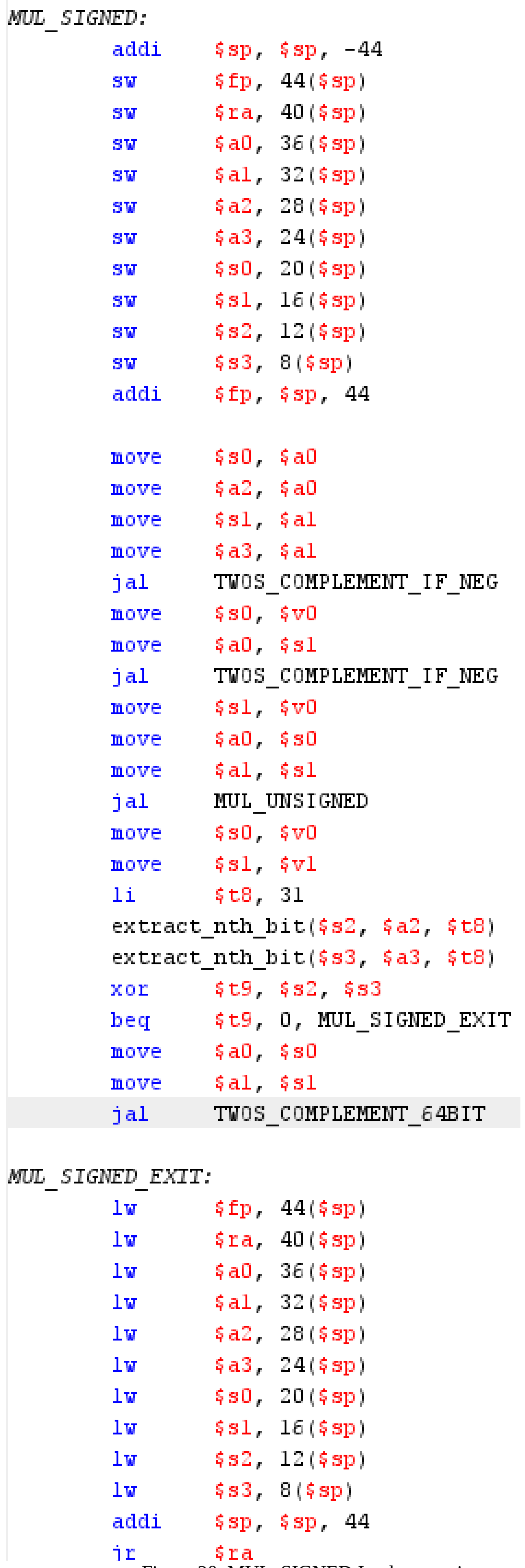
Figure 18. MUL_UNSIGNED Implementation



```
TWOS_COMPLEMENT_64BIT:
        addi    $sp, $sp, -36
        sw      $fp, 36($sp)
        sw      $ra, 32($sp)
        sw      $a0, 28($sp)
        sw      $a1, 24($sp)
        sw      $a2, 20($sp)
        sw      $s0, 16($sp)
        sw      $s1, 12($sp)
        sw      $s2, 8($sp)
        addi    $fp, $sp, 36

        not     $a0, $a0
        not     $a1, $a1
        move    $s0, $a1
        or      $a1, $zero, 1
        jal     ADD_LOGICAL
        move    $s1, $v0
        move    $s2, $v1
        move    $a0, $s0
        move    $a1, $s2
        jal     ADD_LOGICAL
        move    $v1, $v0
        move    $v0, $s1

        lw      $fp, 36($sp)
        lw      $ra, 32($sp)
        lw      $a0, 28($sp)
        lw      $a1, 24($sp)
        lw      $a2, 20($sp)
        lw      $s0, 16($sp)
        lw      $s1, 12($sp)
        lw      $s2, 8($sp)
        addi    $sp, $sp, 36
        jr      $ra
```

Figure 19. TWOS_COMPLEMENT_64BIT

Signed Multiplication implementation is shown below.



```
MUL_SIGNED:
        addi    $sp, $sp, -44
        sw      $fp, 44($sp)
        sw      $ra, 40($sp)
        sw      $a0, 36($sp)
        sw      $a1, 32($sp)
        sw      $a2, 28($sp)
        sw      $a3, 24($sp)
        sw      $s0, 20($sp)
        sw      $s1, 16($sp)
        sw      $s2, 12($sp)
        sw      $s3, 8($sp)
        addi    $fp, $sp, 44

        move    $s0, $a0
        move    $a2, $a0
        move    $s1, $a1
        move    $a3, $a1
        jal     TWOS_COMPLEMENT_IF_NEG
        move    $s0, $v0
        move    $a0, $s1
        jal     TWOS_COMPLEMENT_IF_NEG
        move    $s1, $v0
        move    $a0, $s0
        move    $a1, $s1
        jal     MUL_UNSIGNED
        move    $s0, $v0
        move    $s1, $v1
        li      $t8, 31
        extract_nth_bit($s2, $a2, $t8)
        extract_nth_bit($s3, $a3, $t8)
        xor     $t9, $s2, $s3
        beq     $t9, 0, MUL_SIGNED_EXIT
        move    $a0, $s0
        move    $a1, $s1
        jal     TWOS_COMPLEMENT_64BIT

MUL_SIGNED_EXIT:
        lw      $fp, 44($sp)
        lw      $ra, 40($sp)
        lw      $a0, 36($sp)
        lw      $a1, 32($sp)
        lw      $a2, 28($sp)
        lw      $a3, 24($sp)
        lw      $s0, 20($sp)
        lw      $s1, 16($sp)
        lw      $s2, 12($sp)
        lw      $s3, 8($sp)
        addi    $sp, $sp, 44
        jr      $ra
```

Figure 20. MUL_SIGNED Implementation

Notice how the implementation in Figure 20 utilizes the TWOS_COMPLEMENT_64BIT procedure. Together, the TWOS_COMPLEMENT_64BIT procedure and the MUL_SIGNED procedure correctly follow process of the Signed Multiplication Circuit Diagram.

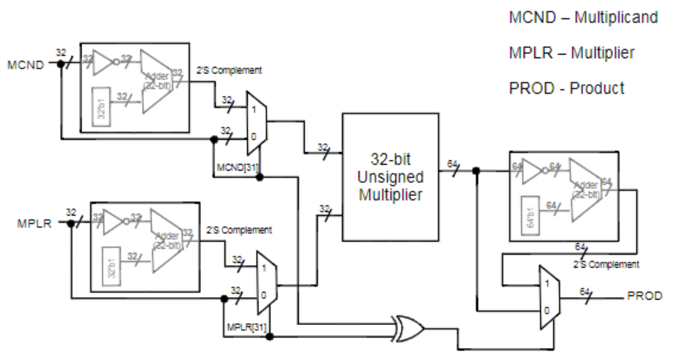


Figure 21. Signed Multiplication Circuit Diagram

3) Division

Division is very similar to multiplication. Division will also be split into two methods:

1) *DIV_UNSIGNED*
   Unsigned Division
2) *DIV_SIGNED*
   Signed Division

Unsigned division also takes two arguments $a0 and $a1 just like unsigned multiplication. $a0 will be the dividend and $a1 will be the divisor. The quotient will return as $v0 and the remainder will return as $v1.

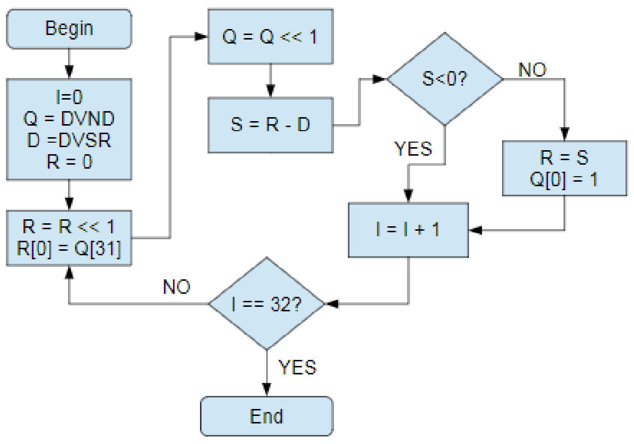


Figure 22. Unsigned Division Flowchart

To implement DIV_UNSIGNED, we will just follow the process of the flowchart in Figure 22 above. A loop will run the commands 32 times. For the 64 bit register, the remainder register will be shifted left so that we can insert 31st bit of the quotient register into its 0th position. Afterwards, the quotient register shifts left and the dividend will shift out to hold the quotient and the remainder. The implementation is shown below in Figure 23.

```
DIV_UNSIGNED:
        addi    $sp, $sp, -40
        sw      $fp, 40($sp)
        sw      $ra, 36($sp)
        sw      $a0, 32($sp)
        sw      $a1, 28($sp)
        sw      $a2, 24($sp)
        sw      $s0, 20($sp)
        sw      $s1, 16($sp)
        sw      $s2, 12($sp)
        sw      $s3, 8($sp)
        addi    $fp, $sp, 40
        or      $t5, $zero, 0
        or      $t6, $zero, 0
        move    $s0, $a0
        move    $s1, $a1
        or      $s2, $zero, 0
        or      $s3, $zero, 0
DIV_UNSIGNED_L1:
        beq     $t5, 32, DIV_UNSIGNED_EXIT
        sll     $t6, $t6, 1
        li      $t8, 31
        extract_nth_bit($s3, $s0, $t8)
        insert_to_nth_bit ($t6, $zero, $s3, $t9)
        sll     $s0, $s0, 1
        move    $a0, $t6
        move    $a1, $s1
        jal     SUB_LOGICAL
        move    $s2, $v0
        bltz    $s2, DIV_UNSIGNED_L2
        move    $t6, $s2
        li      $t8, 1
        insert_to_nth_bit($s0, $zero, $t8, $t9)
DIV_UNSIGNED_L2:
        addi    $t5, $t5, 1
        j       DIV_UNSIGNED_L1
DIV_UNSIGNED_EXIT:
        move    $v0, $s0
        move    $v1, $t6

        lw      $fp, 40($sp)
        lw      $ra, 36($sp)
        lw      $a0, 32($sp)
        lw      $a1, 28($sp)
        lw      $a2, 24($sp)
        lw      $s0, 20($sp)
        lw      $s1, 16($sp)
        lw      $s2, 12($sp)
        lw      $s3, 8($sp)
        addi    $sp, $sp, 40
        jr      $ra
```

Figure 23. DIV_UNSIGNED Implementation

DIV_SIGNED will be performed for all division whether (both signed and unsigned).

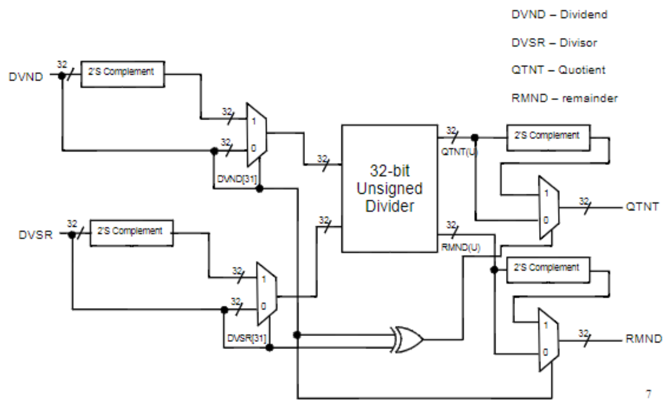


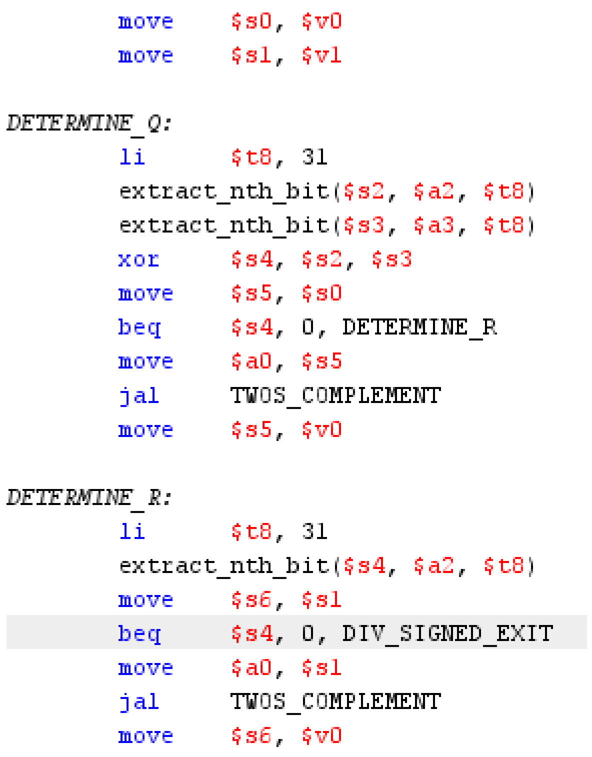
Figure 24. Signed Division Diagram

The process for this is very similar to the MUL_SIGNED process as in the operands will also be checked for their signage by converting to their two's complement form. DIV_UNSIGNED will be called for unsigned division. Quotient will be saved into $v0 and the remainder will be saved into $v1. The signs of the quotient and remainder will be determined by the XOR operation between the original operands. The implementation will be shown below in Figure 25, Figure 26, and Figure 27.

```
DIV_SIGNED:
        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

        move    $s0, $a0
        move    $a2, $a0
        move    $s1, $a1
        move    $a3, $a1
        jal     TWOS_COMPLEMENT_IF_NEG
        move    $s0, $v0
        move    $a0, $s1
        jal     TWOS_COMPLEMENT_IF_NEG
        move    $s1, $v0
        move    $a0, $s0
        move    $a1, $s1
        jal     DIV_UNSIGNED
```

Figure 25. DIV_SIGNED Implementation

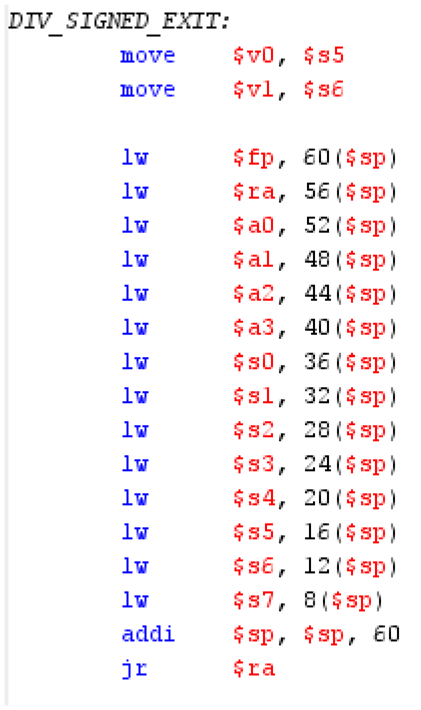
In Figure 25 above, we see TWOS_COMPLEMENT_IF_NEG being called to check the by converting to the two's complement form



```
        move    $s0, $v0
        move    $s1, $v1

DETERMINE_Q:
        li      $t8, 31
        extract_nth_bit($s2, $a2, $t8)
        extract_nth_bit($s3, $a3, $t8)
        xor     $s4, $s2, $s3
        move    $s5, $s0
        beq     $s4, 0, DETERMINE_R
        move    $a0, $s5
        jal     TWOS_COMPLEMENT
        move    $s5, $v0

DETERMINE_R:
        li      $t8, 31
        extract_nth_bit($s4, $a2, $t8)
        move    $s6, $s1
        beq     $s4, 0, DIV_SIGNED_EXIT
        move    $a0, $s1
        jal     TWOS_COMPLEMENT
        move    $s6, $v0
```

Figure 26. DIV_SIGNED Implementation Continued

In Figure 26 above, DETERMINE_Q and DETERMINE_R are procedures used to determine the signage of the quotient and remainder with the use of XOR.



```
DIV_SIGNED_EXIT:
        move    $v0, $s5
        move    $v1, $s6

        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
        jr      $ra
```

Figure 27. DIV_SIGNED Implementation Continued

V. Macros

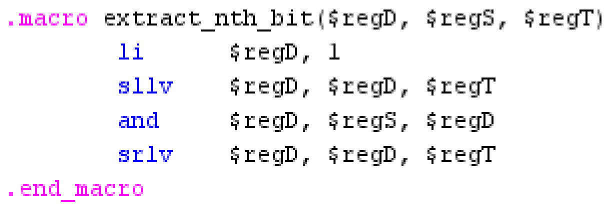
1) Bit Extraction

This macro is very important because it will help determine what value is in what specific bit position. A register that holds the value will shift left into the same position as the desired value. Then the AND operation will be used between the mask and the bit pattern to return the value of the desired bit. Afterwards, it will shift right by the original amount. Three registers will be used:

1) $regD : will contain 0x0 or 0x1 depending on nth bit being 0 or 1

2) $regS: Source bit pattern

3) $regT: Bit position n (031)



```
.macro extract_nth_bit($regD, $regS, $regT)
        li      $regD, 1
        sllv    $regD, $regD, $regT
        and     $regD, $regS, $regD
        srlv    $regD, $regD, $regT
.end_macro
```

Figure 28. extract_nth_bit implementation

2) Bit Insertion

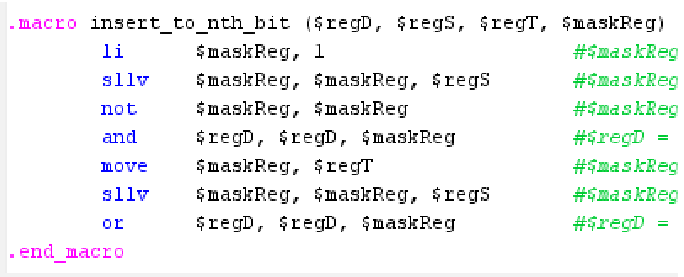
This macro is also very important because our procedures need to place values in specific positions. A mask register with the value of 1 will shift left into the position that will hold the inserted bit. The mask will be inverted. An AND operation will be performed between the inverted mask and the original bit pattern. A second register will hold the desired inserted value and will be shifted left into the desired position. An OR operation will be used between this register and the last result to insert the value into the desired position. Four registers will be used.

1) $regD : This the bit pattern in which 1 to be inserted at nth position

2) $regS: Value n, from which position the bit to be inserted (031)

3) $regT: Register that contains 0x1 or 0x0 (bit value to insert)

4) $maskReg: Register to hold temporary mask

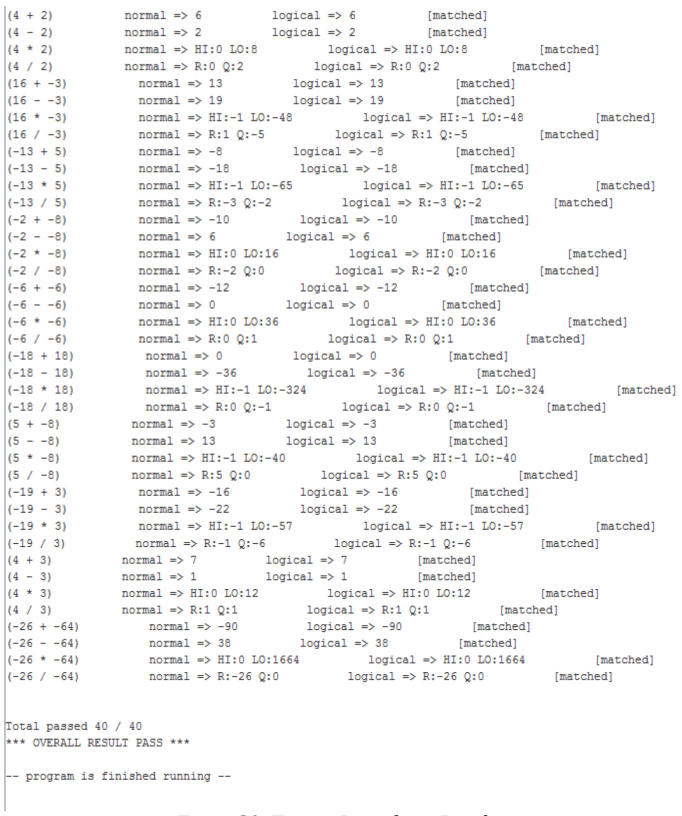


```
.macro insert_to_nth_bit ($regD, $regS, $regT, $maskReg)
        li      $maskReg, 1                             #$maskReg
        sllv    $maskReg, $maskReg, $regS               #$maskReg
        not     $maskReg, $maskReg                      #$maskReg
        and     $regD, $regD, $maskReg                  #$regD =
        move    $maskReg, $regT                         #$maskReg
        sllv    $maskReg, $maskReg, $regS               #$maskReg
        or      $regD, $regD, $maskReg                  #$regD =
.end_macro
```

Figure 29. insert_to_nth_bit implementation

VI. Testing

After completing all procedures, save all of the files and assemble proj_auto_test.asm by clicking on the icon with two wrenches. Then, run it by clicking on the icon with a play button next to the assemble button. If the procedures run correctly, au_normal and au_logical should overall pass with a 40/40, meaning that all 40 math expressions gave the same answer. Figure 30 below shows the output if all procedures are correct.



```
(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 --
```

Figure 30. Testing Procedures Result

VII. Conclusion

This project has taught me a lot about MARS, MIPS, and computer architecture. In addition, I learned a lot about math such as binary, decimal, and hex conversion and how this is used to work with 32-bit and 64-bit systems. The more complex sections that I learned were frame storage and branches. Syntax and commands are very easy to learn as it is just memorization, but one must truly understand the concept and logic behind frame storage to use it over and over again. Creating the frame storage and accessing it with stack points and frame pointers for each procedure taught me a lot. Moving the wrong stack pointer or frame pointer and storing with the wrong temporary register can be very problematic. I made the mistake of using the same temporary registers in both the calling and called procedures. This caused my calling procedure's temporary register to be overwritten and lost its original value. The CS club tutors in Maquarie Hall Room 226 at San Jose State University help me a lot. Computer architecture shows me that computers actually process things in a very difficult manner. For example, simple mathematical procedures such as addition, subtraction, multiplication, and division become very complex for computers as computers as we have seen with the sub procedures (ADD_SUB_LOGICAL,TWOS_COMPLEMENT, etc.).

REFERENCES:

[1] K. Patra. CS 47. Class Lecture, Topic: "Addition Subtraction Logic." San Jose State University, San Jose, CA, November 13, 2016.

[2] K. Patra. CS 47. Class Lecture, Topic: "Multiplication Logic." San Jose State University, San Jose, CA, November 15, 2016.

[3] K. Patra. CS 47. Class Lecture, Topic: "Division Logic." San Jose State University, San Jose, CA, November 20, 2016.