Arithmetic Operations Implemented with MIPS Logical Operations

Eric Fonseca | San Jose State University | eric.fonseca@sjsu.edu

***Abstract*—This report details on the implementation of basic mathematical operations (namely addition, subtraction, multiplication, and division) using MIPS normal procedures, in addition to using logical procedures in MIPS Assembler and Runtime Simulator.**

I. Introduction

Using MIPS Assembler and Runtime Simulator (MARS), our goal is to be able to perform essential mathematical operations (add, sub, mult, and div). We will accomplish this in two components. **Normal procedures** using built-in MIPS procedures and **logical procedures** using logical procedures (e.g. and, or, xor, etc.). Objectives of the program are as follows:

Requirements

1. Download, install, and set up MARS.
2. Implement cs_proj_alu_normal.asm to perform logical operations
3. Implement cs_proj_alu_logical.asm to perform built in normal operations.
4. Test the implementation of the MIPS procedures.

II. INSTALLATION AND SETUP

A. *Download the simulation tool (MARS)*

To get started download our simulation tool that we will use called MARS via the link bellow:

http://courses.missouristate.edu/KenVollmar/mars/download.htm

- Make sure your machine has the latest Java SDK installed.
- If not, then click "Download Java" to download the latest version of Java: https://www.java.com/en/

B. *Download Project Files & Loading Project in Simulator*

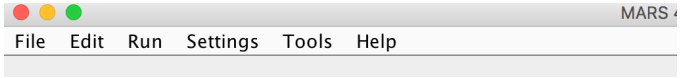
Download the compressed folder containing the files from the link below:

https://sjsu.instructure.com/courses/1228347/assignments/4372408

Unzip the files into your designated project directory. A total of six files should fill your directory. Then, click Mars4_5.jar file to run it.

1. ***proj-auto-test.asm*** will test our implementation.
2. ***cs47_proj_procs.asm*** includes project procedures.
3. ***cs47_proj_alu_normal.asm*** normal procedures for the arithmetic operations.
4. ***cs47_proj_alu_logical.asm*** logical procedures for the arithmetic operations.
5. ***cs47_common_macro.asm*** macros for printing out the test results.
6. ***cs47_proj_macro.asm*** macros written by us to use for our logical procedures.

Open MARS, and navigate to settings tab



and make sure that all the options in Fig 1. are checked.

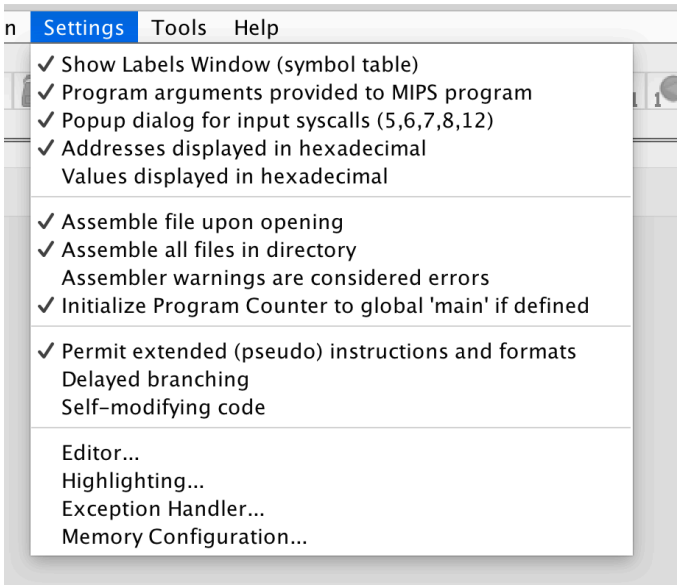


Figure 1: MARS settings

These settings will help run the project without problems.
1) "Initialize Program Counter to global 'main' if defined" is for the program to begin running at the address defined by a "main" label, as opposed to starting from the first label.
2) "Assemble all files in directory" will allow the tester to run even if a required file is not open in MARS.

Open the six required files in MARS by going to “File” and open, then navigate to your extracted folder.

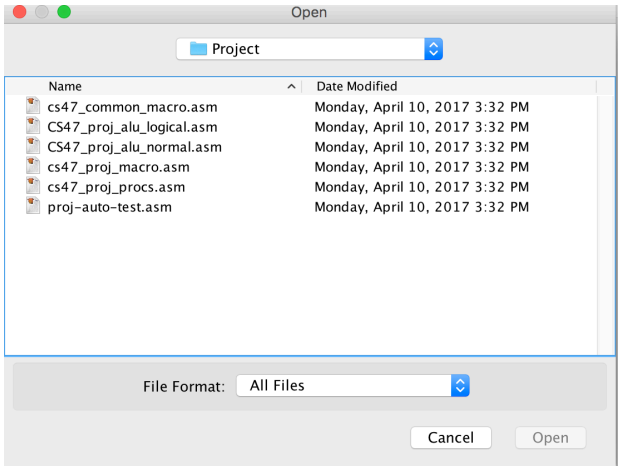


Figure 2: Opening the files in MARS

Perform this for each of the six files until you have all six opened on your simulator.

III. The MIPS Procedures

As we stated in the introduction, the arithmetic operations will be implemented in two components. First component will utilize MIPS normal procedures, and the second component will be using logical operations. Both components will take three arguments in three registers:

A. *Normal Procedure*

Named cs47_proj_alu_normal.asm will calculate our mathematical expression using built in MIPS procedures (addition, subtraction, multiplication, division). The normal procedures will be implemented in *cs47_proj_alu_normal.asm* under the *au_normal* label.

1. *Register $a0* is the first operand in our mathematical operation.
2. *Register $a1* is the second number in our mathematical operation.
3. *Register $a2* will contain the operator or Opcode of our mathematical operation.
   (“+”addition, “ – ” subtraction, “ * ” multiplication, “ / ” division)

The results of the arithmetic operation will be stored in the following registers:

1. ***Register $v0*:**
   a. For addition and subtraction, $v0 will hold the result of the addition or subtraction operation of $a0 and $a1.
   b. For multiplication, $v0 will hold the LO part of the result.
   c. For division, $v0 will hold the quotient.

2. *Register v1, or $v1:*
   a. For multiplication, $v1 will hold the HI part of the result.
   b. For division, $v1 will hold the remainder of the result.

B. *Logical Procedures*

The logical procedure will simulate digital circuits in a MIPS processor. It will be implemented using digital logic gates such as AND, OR, XOR, and NOT. However, we will use add or any of its derivatives to increment numbers for counters and setting certain values. The logical procedures will be implemented in *CS47_proj_alu_logical.asm* under the *au_logical* label. Similar to the normal procedure, the logical procedure will take three arguments:

1. $ao – First operand in our mathematical expression
2. $a1 – Second operand in our mathematical expression.
3. $a2 – The Opcode that will determine what operation will execute.

The results of the arithmetic operation will be stored in the following registers:

3. ***Register $v0*:**
   d. For addition and subtraction, $v0 will hold the result of the addition or subtraction operation of $a0 and $a1.
   e. For multiplication, $v0 will hold the LO part of the result and $v1 will contain the HI.
   f. For division, $v0 will hold the quotient., while $v1 will contain the remainder

IV. Designing and Implementing MIPS Procedures

The design behind the procedures is based on operator recognition and branching to a procedure that corresponds to the operator in $a2.

A. *Normal Procedures*

Determine which operator is in $a2, then we branch to the procedure that will perform the operator’s desired operation.

```
au_normal:


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

        beq     $a2, $t0, addition
        beq     $a2, $t1, subtraction
        beq     $a2, $t2, multiplication
        beq     $a2, $t3, division


addition:
```

Figure 3: au_normal start and branch

For example “+” in $a2 will result a branching to the “addition” label and addition between $a0 and $a1 will be performed, yielding the result of the addition in $v0. “-“, “*”, or “/” in $a2 will call the subtraction, multiplication, and

division labels respectively, and yield the result(s) in their respective registers.

```
addition:
        add     $v0, $a0, $a1
        jr      $ra

subtraction:
        sub     $v0, $a0, $a1
        jr      $ra

multiplication:
        mult    $a0, $a1
        mflo    $v0
        mfhi    $v1
        jr      $ra

division:
        div     $a0, $a1
        mflo    $v0
        mfhi    $v1
        jr      $ra
```

Figure 4: au_normal arithmetic procedures

After the procedures are complete, au_normal will return control back to the label's original caller jr $ra.

B. Logical Procedures

The logical procedures in au_logical uses the concept of branching to different procedures based on the operator in $a2.

```
au_logical:
        #store RTE
        subi    $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
logic_conditions:
        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       End_Logic
```

Figure 5: au_logical start and branching

However, unlike the normal procedures, the logical procedures will call additional in-depth arithmetic procedures.

```
addition:
        jal     add_logical
        j       End_Logic

subtraction:
        jal     sub_logical
        j       End_Logic

multiplication:
        jal     mul_signed
        j       End_Logic

division:
        jal     div_signed
        j       End_Logic

End_Logic:
        lw      $fp, 24($sp)
        lw      $ra, 20($sp)
        lw      $a0, 16($sp)
        lw      $a1, 12($sp)
        lw      $a2, 8($sp)
        addi    $sp, $sp, 24
        jr      $ra
```

Figure 6: au_logical calling procedures

1. *add_sub_logical*

The addition and subtraction labels will call their respective processes of add_logical and sub_logical. However, add_logical and sub_logical both call a process known as add_sub_logical.

add_logical- This procedure will call add_sub_logical procedure to perform addition. Thus, the argument $a2, which determines the operation, will contain 0x00000000.

sub_logical - This procedure will call add_sub_logical procedure to perform subtraction. Thus, the argument $a2, which determines the operation, will contain 0xFFFFFFFF.

```
add_logical:
        subi    $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, $zero
        addi    $a2, $a2, 0
        jal     add_sub_logical
        j       End_Logic
###################################
###################################

sub_logical:
        subi    $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, $zero
        addi    $a2, $a2, 1
        jal     add_sub_logical
        j       End_Logic
```

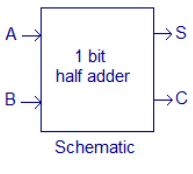
Figure 7: add_logical and sub_logical

The idea behind add_sub_logical is the following: use $a2 to determine the mode (additions or subtraction), if 0x00000000 then branch to addition, if 0xFFFFFFFF then branch to subtraction.

When adding a number in MIPS, we can only add one bit of the operands at a time, and must consider carry-out of the binary adding operations. This is where we implement our half adder.

Inputs		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Truth table



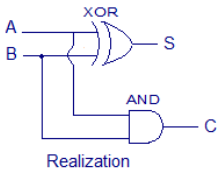
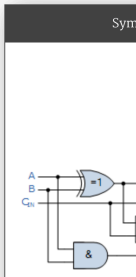


Figure 8: Half Adder

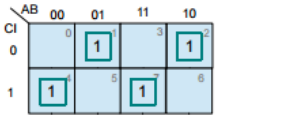
The Half Adder design adds two binary bits together, and stores a carry-out bit for adding the next bit of the operand. The sum of the binary bits is determined through an AND operation, and its carry-out bit is determined through a XOR operation. To determine a full number addition and consider both the carry-in bit and the carry-out bit we need to use a full adder , a full adder that adds the all of the operands' bits are required.



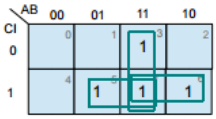
Truth Table				
C-in	B	A	Sum	C-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 9: Full Adder truth table

Using the truth table of the Full Adder, we can simplify our expression and using a Karnaugh map, or K-map we can determine its design.



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

Figure 10: Full Adder Karnaugh map

The expressions obtained by simplification via K-map will serve as the basis for the logical design of the Full Adder.

Determining the sum that results from adding two numbers involves using a XOR gate that takes the carry-in bit from both numbers. Next, we use a OR gate for our carry-out bits that results from the two half adders.

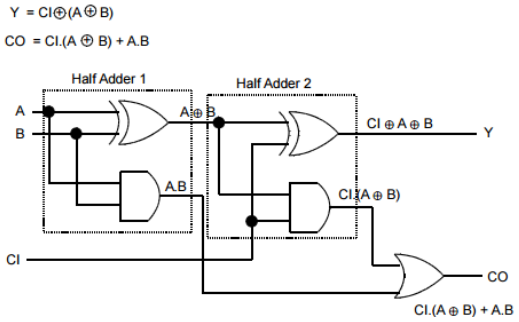


Figure 11: Full Adder circuit diagram

add_sub_logical utilizes the Full Adder design and to perform addition and subtraction, as subtraction is also equivalent to the addition of a negative operand ($a0 + ~($a1)) which can be viewed as an addition. So this is why in order to perform subtraction we would need to convert our second operand into two's complement by inverting using NOT $a1, and adding 1 to it. The following snippet of code illustrates *twos_complement* implementation in code.

```
twos_complement:
        subi    $sp, $sp, 28
        sw      $fp, 28($sp)
        sw      $ra, 24($sp)
        sw      $a0, 20($sp)
        sw      $a1, 16($sp)
        sw      $s0, 12($sp)
        sw      $s1, 8($sp)
        addi    $fp, $sp, 28

        not     $a0, $a0
        li      $a1, 1
        jal     add_logical
        # Restore stack frame
        lw      $fp, 28($sp)
        lw      $ra, 24($sp)
        lw      $a0, 20($sp)
        lw      $a1, 16($sp)
        lw      $s0, 12($sp)
        lw      $s1, 8($sp)
        addi    $sp, $sp, 28
        jr      $ra
```

Figure 12: twos_complement

By doing these steps, we can determine if $a1's two's complement is needed. To fully add two 32-bit numbers together, one must loop through the operands' bits, and use the Full Adder to add their bits together and factor in their carry bits.

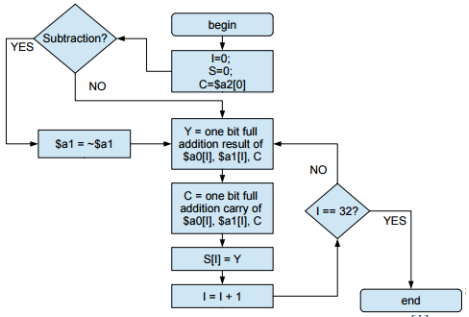


Figure 13: add_sub_logical flowchart[1]

This flowchart illustrates the logic behind the Full Adder and the loop to add two 32-bit numbers together. With it we can implement it in code. As shown bellow

```
add_sub_logical:
        #Store frame
        addi    $sp, $sp, -32
        sw      $fp, 32($sp)
        sw      $ra, 28($sp)
        sw      $a0, 24($sp)
        sw      $a1, 20($sp)
        sw      $a2, 16($sp)
        sw      $s0, 12($sp)
        sw      $s1, 8($sp)
        addi    $fp, $sp, 32
        beq $a2, 0, Logic_Add_Branch
        beq $a2, 1, Logic_Sub_Branch
Logic_Sub_Branch:
        not     $a1, $a1
        li      $t0, 0
        li      $v0, 0
        and     $s0, $a2, 0x1
        beqz    $a2, Logic_Add_Loop
Logic_Add_Branch:
        li      $t0, 0
        li      $v0, 0
        and     $s0, $a2, 0x1
        beqz    $a2, Logic_Add_Loop
Logic_Add_Loop:
        extract_nth_bit($t2, $a0, $t0)
        extract_nth_bit($t3, $a1, $t0)
        xor     $t4, $t2, $t3
        xor     $t6, $s0, $t4
        and     $t7, $s0, $t4
        and     $t8, $t2, $t3
        or      $s0, $t7, $t8
        insert_to_nth_bit($v0, $t0, $t6, $t9)
        addi    $t0, $t0, 0x1
        bne     $t0, 32, Logic_Add_Loop
        beq     $t0, 32, end_add_sub_logical
end_add_sub_logical:
        move $v1, $s0
        # Restore
        lw      $fp, 32($sp)
        lw      $ra, 28($sp)
        lw      $a0, 24($sp)
        lw      $a1, 20($sp)
        lw      $a2, 16($sp)
        lw      $s0, 12($sp)
        lw      $s1, 8($sp)
        addi    $sp, $sp, 32
        jr      $ra
```

Figure 14: add_sub_logical

2. *Multiplication*

Multiplication is more complicated then addition. There are two cases that we must keep in mind.

1. Multiplication of an unsigned integer
    a. mul_unsigned for handling unsigned numbers.
2. Multiplication of a signed integer
    a. mul_signed exclusively used to determine the sign of the result

1. mul_signed

For a 32-bit unsigned multiplier, the logical design for it is listed below:

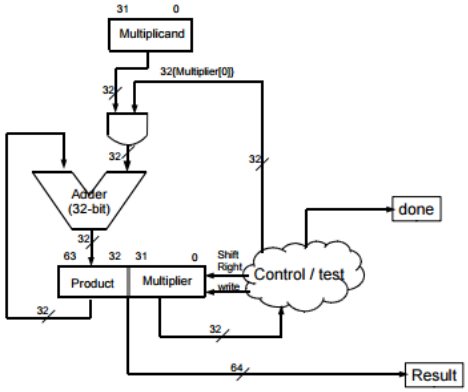


Figure 15: Unsigned Multiplication design[2]

The result of the unsigned multiplication multiplicand can be obtained by an AND operation between the multiplicand and the multiplier. However, since this is multiplication, one must shift the multiplier right by 1 to AND the correct bits, similar to how hand-worked multiplication works. Additionally, we must AND the multiplicand and the multiplier's bits in the correct format 32 times, due to the fact that the multiplicand and multiplier occupy $a0 and $a1, which are both 32-bit registers.

One must realize that the AND operations between the multiplicand and multiplier will result in this truth table:

TABLE I
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

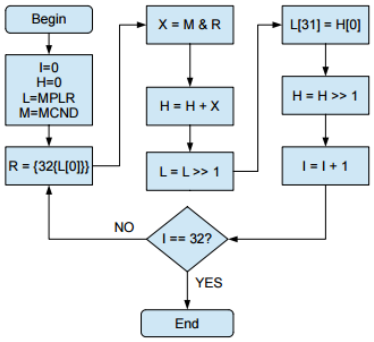


Figure 16: Unsigned Multiplication flowchart

In the above flowchart the AND operations will occur 32 times, with the multiplier and multiplicand shifting to ensure the correct bits are AND'ed. Below is the implementation of mul_unsigned:

```
mul_unsigned:
        subi    $sp, $sp, 52
        sw      $fp, 52($sp)
        sw      $ra, 48($sp)
        sw      $a0, 44($sp)
        sw      $a1, 40($sp)
        sw      $a2, 36($sp)
        sw      $s0, 32($sp)
        sw      $s1, 28($sp)
        sw      $s2, 24($sp)
        sw      $s3, 20($sp)
        sw      $s4, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7,  8($sp)
        addi    $fp, $sp, 52
        li      $s7, 0
        li      $s4, 0
        move    $s1, $a1
        move    $s0, $a0
mul_unsigned_loop:
        beq     $t5, 0x20, mul_unsigned_end
        extract_nth_bit($s2, $s1, $zero)
        move    $a0, $s2
        jal     bit_replicator
        move    $s2, $v0
        and     $s3, $s0, $s2
        move    $a0, $s4
        move    $a1, $s3
        jal     add_logical
        move    $s4, $v0
        srl     $s1, $s1, 1
        extract_nth_bit($t0, $s4, $zero)
        li      $s6, 0x1f
        insert_to_nth_bit($s1, $s6, $t0, $t9)
        srl     $s4, $s4, 1
        addi    $s7, $s7, 1
        bne     $s7, 32, mul_unsigned_loop
mul_unsigned_end:
        move    $v0, $s1
        move    $v1, $s4
        # Restore Frame
        lw      $fp, 52($sp)
        lw      $ra, 48($sp)
        lw      $a0, 44($sp)
        lw      $a1, 40($sp)
        lw      $a2, 36($sp)
        lw      $s0, 32($sp)
        lw      $s1, 28($sp)
        lw      $s2, 24($sp)
        lw      $s3, 20($sp)
        lw      $s4, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7,  8($sp)
        addi    $sp, $sp, 52
        jr      $ra
```

Figure 17: mul_unsigned

In the implementation, there is a loop that occurs 32 times. Inside the loop, once can see add_logical as the Full Adder can be utilized in this operation. Once can also see the extraction of the product's bits and inserted into the lower bits as a working virtual 64-bit register. This would mimic the shifting that would occur in hand-worked multiplication.

For signed multiplication, this is the implementation:

1. ***mul_signed***

The logical design for signed multiplication is listed below:

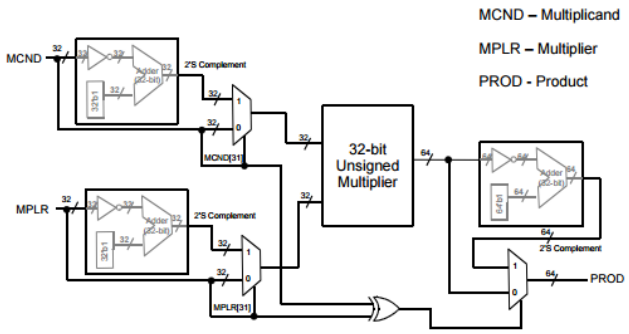


Figure 18: Signed Multiplication design[2]

First, our operands will be converted to its corresponding two's complement form if they are negative. This will allow us to go into case 1 (mul_unsigned) and perform unsigned multiplication.

TABLE I
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

It is important the results of our signed multiplication gives us two 32-bit results in registers $v0 and $v1. This can be done via sign extension and converted into its twos_complement 64 bit form. The results in $v0 and $v1 will have their signs determined by a XOR operation between original multiplicand and multiplier's signs.

```
mul_signed:

        subi    $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      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        addi    $fp, $sp, 44

        move    $s4, $a0        # s4 = copy
        move    $a2, $a0        # Extra cop
        move    $s5, $a1        # s5 = copy
        move    $a3, $a1        # Extra cop

        jal     twos_complement_if_neg
        move    $s4, $v0        # Store two
        move    $a0, $s5        # Now do th
        jal     twos_complement_if_neg
        move    $s5, $v0        # Store two

        move    $a0, $s4        # Move s4 i
        move    $a1, $s5        # Move s5 i
        jal     mul_unsigned

        move    $s4, $v0        # s4 = lo o
        move    $s5, $v1        # s5 = hi o

        li      $t8, 0x1F
        extract_nth_bit($s6, $a2, $t8)  #
        extract_nth_bit($s7, $a3, $t8)

        xor     $t9, $s6, $s7   # Sign = XO
        beq     $t9, $zero, mul_signed_end

        move    $a0, $s4
        move    $a1, $s5
        jal     twos_complement_64bit


mul_signed_end:
        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      $s4, 20($sp)
        lw      $s5, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7, 8($sp)
        addi    $sp, $sp, 44
        jr      $ra
#########################################
```

Figure 19: mul_signed

To make sure sure that we have $v0 and $v1 in their correct form, $v0 will contain the LO parts of the operation, and $v1 will contain the HI parts.

this is because multiplying two 32-bit numbers will result in a 64-bit result, which cannot be contained in 1 MIPS register.

```
twos_complement_64bit:
        subi    $sp, $sp, 36
        sw      $fp, 36($sp)
        sw      $ra, 32($sp)
        sw      $a0, 28($sp)
        sw      $a1, 24($sp)
        sw      $a2, 20($sp)
        sw      $s4, 16($sp)
        sw      $s5, 12($sp)
        sw      $s6, 8($sp)
        addi    $fp, $sp, 36

        not     $a0, $a0
        not     $a1, $a1
        move    $s4, $a1

        or      $a1, $zero, 0x1
        jal     add_logical

        move    $s5, $v0
        move    $s6, $v1

        move    $a0, $s4
        move    $a1, $s6
        jal     add_logical

        move    $v1, $v0
        move    $v0, $s5
        lw      $fp, 36($sp)
        lw      $ra, 32($sp)
        lw      $a0, 28($sp)
        lw      $a1, 24($sp)
        lw      $a2, 20($sp)
        lw      $s4, 16($sp)
        lw      $s5, 12($sp)
        lw      $s6, 8($sp)
        addi    $sp, $sp, 36
        jr      $ra
```

Figure 20: twos_complement_64bit

This will ensure the results of mul_signed in $v0 and $v1 will be 32-bit, and will be returned by mul_signed.

3. *Division*

The design for division is very similar to multiplication in that we will have two components, signed procedure and unsigned procedure.

1. *div_unsigned*

Procedure similar to mul_unsigned, it will perform unsigned division, taking as input two arguments $a0 (Dividend) and $a1 (Divisor).

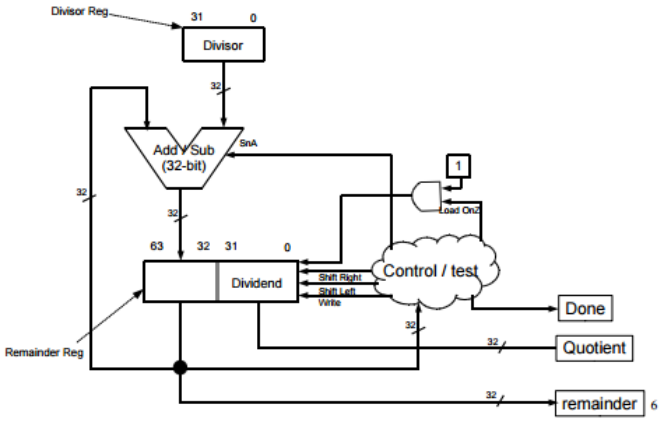


Figure 21: Unsigned Division design[3]

Similar to the multiplication part. A loop that occurs for 32 repetitions. To simulate a 64-bit number being divided, the number in the remainder register will be shifted to the left to allow for for insertion of the 31st bit of quotient register into 0th index. This will eventually happen enough to contain the remainder and quotient.

Here is a flowchart of the unsigned division procedure:

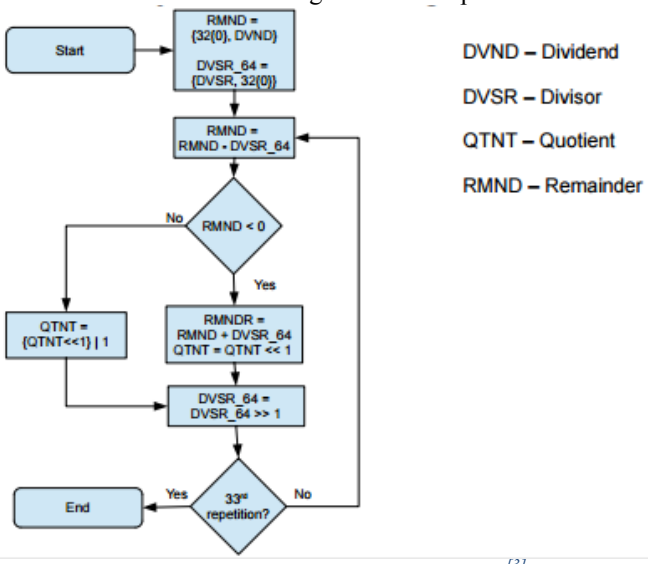


Figure 22: Unsigned Division flowchart[3]

The implementation is relatively straightforward.
Below is the implementation of the unsigned division process:

```
div_unsigned:
        # Store frame
        addi    $sp, $sp, -36
        sw      $fp, 36($sp)
        sw      $ra, 32($sp)
        sw      $a0, 28($sp)
        sw      $a1, 24($sp)
        sw      $s0, 20($sp)
        sw      $s1, 16($sp)
        sw      $s2, 12($sp)
        sw      $s3,  8($sp)
        addi    $fp, $sp, 36

        or      $s0, $zero, $zero
        or      $s3, $zero, $zero
        move    $s1, $a0                ;
        move    $s2, $a1                ;

div_unsigned_loop:
        beq     $t5, 32, div_unsigned_loop_end

        sll     $s3, $s3, 1             ;
        li      $t0, 31
        extract_nth_bit($t3, $s1, $t0)

        insert_to_nth_bit($s3, $zero, $t3, $t9)
        sll     $s1, $s1, 1             ;
        move    $a0, $s3                ;
        move    $a1, $s2                ;
        jal     sub_logical

        move    $t6, $v0                ;

        bltz    $t6, Equals_Zero
        move    $s3, $t6                ;
        li      $t5, 1
        insert_to_nth_bit($s1, $zero, $t5, $t9)

Equals_Zero:
        add     $s0, $s0, 1             ;
        beq     $s0, 32, div_unsigned_loop_end
        j       div_unsigned_loop

div_unsigned_loop_end:
        move    $v0, $s1                ;
        move    $v1, $s3                ;

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

Figure 23: div_unsigned

And as explained before, the unsigned division results will be plugged in for signed division to determine the quotient and remainder's signs.

1. ***div_signed***

Likewise, this procedure will be used to perform division for all arguments (signed and unsigned). It follows similar steps taken in the mul_signed procedure.

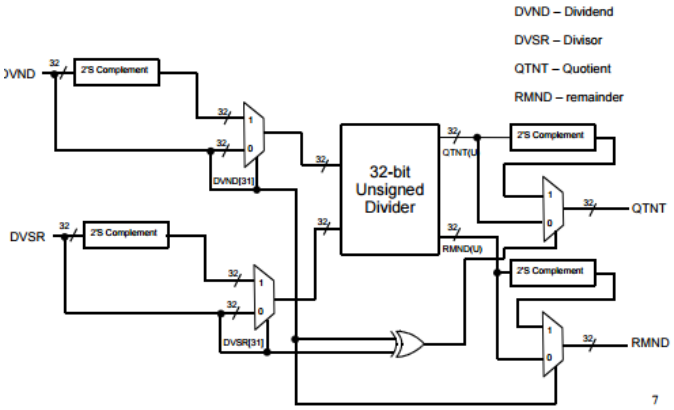


Figure 24: Signed Division design[3]

Once again, we will investigate for the operands sign and if negative, then convert them to their corresponding two's complement. Then we will plug them in div_unsigned. Finally, the signs of the quotient and remainder will be determined by the dividend and divisor's original signs via a XOR operation.

```
div_signed:
        # Store frame
        subi    $sp, $sp, 36
        sw      $fp, 36($sp)
        sw      $ra, 32($sp)
        sw      $a0, 28($sp)
        sw      $a1, 24($sp)
        sw      $s1, 20($sp)
        sw      $s2, 16($sp)
        sw      $s4, 12($sp)
        sw      $s5,  8($sp)
        addi    $fp, $sp, 36
        # Instantiate variables to be used

        move    $s1, $a0                #
        move    $s2, $a1                #


        li      $t7, 31         # $t7 = 31
        extract_nth_bit($s4, $s1, $t7)  #
        extract_nth_bit($s5, $s2, $t7)  #
        xor     $s5, $s4, $s5           #

        # Make $s1 twos_complement_if_neg
        jal     twos_complement_if_neg
        move    $a0, $s1
        jal     twos_complement_if_neg
        move    $s1, $v0                #

        # Make $s2 twos_complement_if_neg
        jal     twos_complement_if_neg
        move    $a0, $s2                #
        jal     twos_complement_if_neg
        move    $s2, $v0                #

        move    $a0, $s1                #
        move    $a1, $s2                #
        jal     div_unsigned

        move    $s1, $v0                #
        move    $s2, $v1                #

        # $s5 is S of Q
        beqz    $s5, IfZero             #
        move    $a0, $s1                #
        jal     twos_complement
        move    $s1, $v0                #

IfZero: # $s4 is S of R
        beqz    $s4, elseIF             #
        move    $a0, $s2                #
        jal     twos_complement
        move    $s2, $v0                #

elseIF:
        move    $v0, $s1                #
        move    $v1, $s2                #

        # Restore frame
        lw      $fp, 36($sp)
        lw      $ra, 32($sp)
        lw      $a0, 28($sp)
        lw      $a1, 24($sp)
        lw      $s1, 20($sp)
        lw      $s2, 16($sp)
        lw      $s4, 12($sp)
        lw      $s5,  8($sp)
        addi    $sp, $sp, 36
        jr      $ra
```

Figure 25: div_signed start

twos_complement_if_neg_is called similarly as in mul_signed.

```
# Make $s1 twos_complement_if_n
jal     twos_complement_if_neg
move    $a0, $s1
jal     twos_complement_if_neg
move    $s1, $v0

# Make $s2 twos_complement_if_n
jal     twos_complement_if_neg
move    $a0, $s2
jal     twos_complement_if_neg
move    $s2, $v0

move    $a0, $s1
move    $a1, $s2
jal     div_unsigned

move    $s1, $v0
move    $s2, $v1
```

Figure 26: Determining the signs of the quotient and remainder

The signs of the quotient and remainder will be determined, as shown in the snippet above of the XOR operation between the original operands.

V. Common Mistakes

A common mistake when writing such code would be to forget to move the results out of $v0 and $v1 and store it in another register if one wishes to use those values later. Also, saving and restoring the stack frame is very important.

Another common mistake would be in the restoration of the frame is the absence of jr $ra. This would cause the procedure to not return to the caller properly and may give incorrect results.

Finally, when dealing with signed and unsigned multiplication and division, one must not forget to take into account of the original sign of the operands.

VI. Macros and other Procedures Used

```
.macro extract_nth_bit($regD, $regS, $regT)
srlv    $regD, $regS, $regT
andi    $regD, 0x1
.end_macro

.macro insert_to_nth_bit($regD, $regS, $regT, $maskReg)
addi    $maskReg, $maskReg, 0x1
sllv    $maskReg, $maskReg, $regS
not     $maskReg, $maskReg
and     $regD, $maskReg, $regD
sllv    $regT, $regT, $regS
or      $regD, $regD, $regT
.end_macro
```

Figure 27: Bit-inserting and Extracting Macros

These macros are used to extract bits of an operand to be used to AND other operands. This is done in to determine their unsigned bits, as well as their results' signs. extract_nth_bit takes as input $regD, which will result in wither 0 or 1. $regS is from where we will extract the bit, and $regT is the index. Similarly, insert_to_nth_bit takes $regD as input and as register of inserted bit. The source of number to insert is $regS,The index of the bit from $regS to insert, and $maskReg is a temporary register to help shift and save the bit to insert.

```
bit_replicator:
        # Store frame
        subi    $sp, $sp, 16
        sw      $fp, 16($sp)
        sw      $ra, 12($sp)
        sw      $a0, 8($sp)
        addi    $fp, $sp, 16

        beq     $a0, 0, POSITIVE
        beq     $a0, 1, NEGATIVE

POSITIVE:
        li      $v0, 0x00000000
        j       bit_replicator_end

NEGATIVE:
        li      $v0, 0xFFFFFFFF
        j       bit_replicator_end

bit_replicator_end:
        lw      $fp, 16($sp)
        lw      $ra, 12($sp)
        lw      $a0,  8($sp)
        addi    $sp, $sp, 16
        jr      $ra
```

Figure 28: bit_replicator

A bit replicator is used as sign extension for multiplication's LO and HI results to ensure both results are indeed 32-bit, together forming a 64-bit number in $v0 and $v1. It is similar to a mask.

```
twos_complement_if_neg:
        # Store frame
        subi    $sp, $sp, 16
        sw      $fp, 16($sp)
        sw      $ra, 12($sp)
        sw      $a0,  8($sp)
        addi    $fp, $sp, 16

        move    $v0, $a0

        bgt     $a0, $zero, twos_complement_if_neg_end
        jal     twos_complement

Execute_twos_complement:
        jal     twos_complement
        move    $a0, $v0                        # $a0 =

twos_complement_if_neg_end:
        # Restore frame
        lw      $fp, 16($sp)
        lw      $ra, 12($sp)
        lw      $a0,  8($sp)
        addi    $sp, $sp, 16
        jr      $ra
```

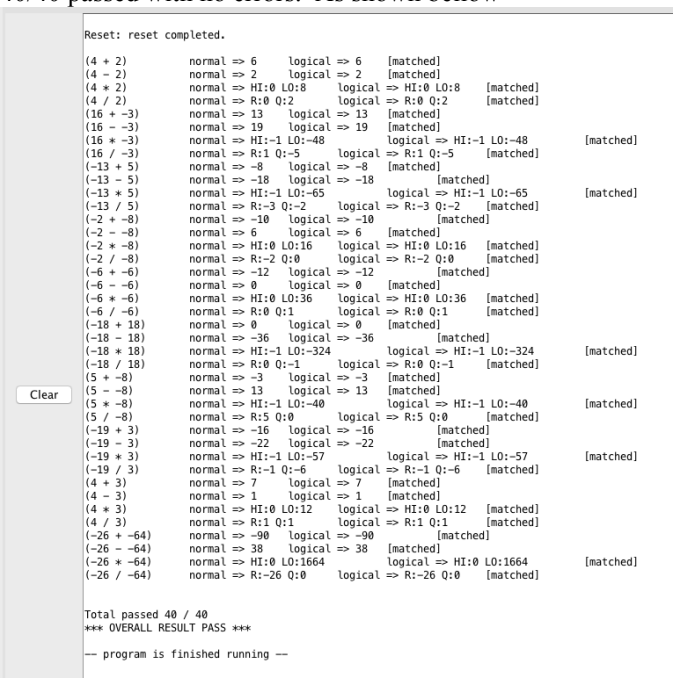
Figure 29: twos_complement_if_neg

The snippet of code above illustrates how we obtain the two's complement of the operands for signed multiplication and signed division when the operands are negative.

VII. Results

Once you have all methods implemented then it is time to assemble and run the provided proj-auto-test.asm. The results of the logical procedures will be compared to the normal procedures and if they match then you will see [matched] if not then you will instead see [not matched].

If your implementation is correct, your result should show 40/40 passed with no errors. As shown bellow



VIII. Conclusion

In completing this project, I gained valuable knowledge of a computers ability to perform complex task with only simple, rudimentary procedures and commands. In the process I gained first hand experience with MIPS simulator (MARS) and its protocols. As well discovered how tedious the work involved with a low level language such as MIPS can be. With so many registers, procedures, and branching methods it is easy to get lost and confused but debugging and keeping track of your registers and procedures is something that helped me enormously. Looking back at the project as a whole it was definitely difficult and at times seemed impossible, but with attention to detail and problem solving skills it was absolutely possible. I now have a better understanding of just how much work a high level language like java does for us, and just how exactly it does those things allows me to better understand what my code is really doing when I create it.

References:

[1] CS 47. Class Lecture, Topic: “Addition Subtraction Logic.” San Jose State University, San Jose, CA, May 1, 2017.

[2] CS 47. Class Lecture, Topic: “Multiplication Logic.” San Jose State University, San Jose, CA, May 1, 2017.

[3] CS 47. Class Lecture, Topic: “Division Logic.” San Jose State University, San Jose, CA, May 1, 2017.

[4] [Online].: http://www.circuitstoday.com/half-adder , San Jose, CA, May 1, 2017.

[5] [Online]: http://www.electronics-tutorials.ws/combination/comb_7.html , San Jose, CA, May5 16, 2016.

[6] [Online]:https://sub.allaboutcircuits.com/images/04116.png, San Jose, CA, November 21, 2016.