Arithmetic Operations Implemented in MIPS via Logical Operations

Michael Huang | San Jose State University | Michael.Huang.9001@gmail.com

***Abstract*—This report contains the documentation and 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

Through the MIPS Assembler and Runtime Simulator (MARS), one should be able to perform essential mathematical operations. Two versions should be available: operations performed using built-in MIPS procedures (e.g. add, sub, etc.), called **normal procedures**, and operations performed using logical procedures (e.g. and, or, xor, etc.), or **logical procedures**. For this project, one will be able to:

1. Download, install, and set up MARS.
2. Implement the MIPS procedures to perform mathematical operations.
3. Test the implementation of the MIPS procedures.

To download MARS, please visit this link:
courses.missouristate.edu/KenVollmar/mars/download.htm

Click "Download Java" to download MARS.
Be sure to download the latest version of Java:
https://www.java.com/en/

II. Project Files

To get started, please download the compressed folder containing the files to get started from this link: https://sjsu.instructure.com/courses/1208160/files/44918119/download?wrap=1
Unzip those files into a directory, there should six files in total. Then, click Mars4_5.jar file to run it.

1. *cs47_common_macro.asm* contains macros for printing out the test results.
2. *CS47_proj_alu_logical.asm* contains the logical procedures for the arithmetic operations.
3. *CS47_proj_alu_normal.asm* contains the normal procedures for the arithmetic operations.
4. *cs47_proj_macro.asm* contains the macros one will write for the logical procedures.
5. *cs47_proj_procs.asm* contains project procedures.
6. *proj-auto-test.asm* is used to test the implementation.

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

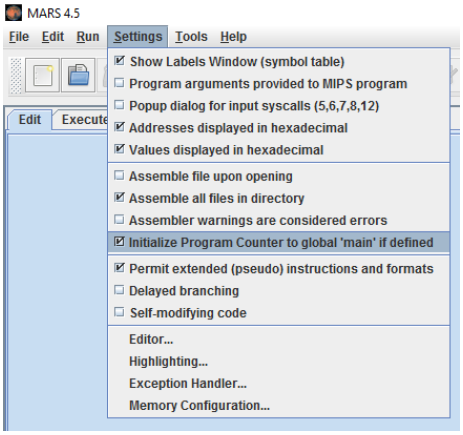

Figure 1: Optimizing MARS settings

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

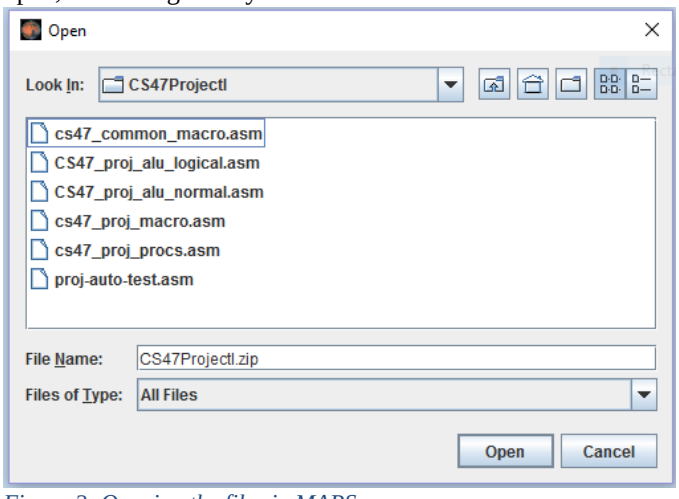

Figure 2: Opening the files in MARS

Open them one after another and this load all of the required files in MARS, and your tabs should look like this:

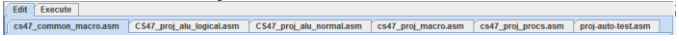

Figure 3: All files opened and loaded

III. The MIPS Procedures

As mentioned previously, the arithmetic operations will be implemented in two ways, one using MIPS normal procedures, and another using logical operations. Both normal and logical procedures will take three arguments in three registers:

1. *Register a0, or $a0* will contain the first number to be inputted into the arithmetic operation.
2. *Register a1, or $a1* will contain the second number to be inputted into the arithmetic operation.
3. *Register a2, or $a2* will contain the operator or opcode of the arithmetic operation.

$a2 will contain one of these four operators

- The "+" operator will represent an addition operation.
- The "-" operator will represent a subtraction operation.
- The "*" operator will represent a multiplication operation.
- The "/" operator will represent a division operation.

Both procedures will also return the result of the arithmetic operation in specific return registers:

1. *Register v0, or $v0*:
   a. For addition, $v0 will hold the result of the addition operation involving $a0 and $a1.
   b. For subtraction, $v0 will hold the result of the subtraction operation involving $a0 and $a1.
   c. For multiplication, $v0 will hold the LO part of the result of the multiplication operation involving $a0 and $a1.
   d. For division, $v0 will hold the quotient of the result of the division operation involving $a0 and $a1.
2. *Register v1, or $v1:*
   a. For addition and subtraction, this will not be used.
   b. For multiplication, $v1 will hold the HI part of the result of the multiplication operation involving $a0 and $a1.
   c. For division, $v1 will hold the remainder of the result of the division operation involving $a0 and $a1.

A. About Normal Procedures

The normal procedures will be implemented using common MIPS mathematical operations, primarily consisting of add, sub, mul, and div. The normal procedures will be implemented in *CS47_proj_alu_normal.asm* under the *au_normal* label.

B. About Logical Procedures

The logical procedures will be implemented using digital logic operations such as AND, OR, XOR, and NOT. A part of the project requirement is to disallow the use of common MIPS mathematical operations such as add, sub, mul, and div for arithmetic procedures. However, the usage of add or any of its derivatives are allowed to be used to increment numbers for counters and setting certain values. The purpose of this of this restriction is to simulate digital circuits in a MIPS processor. The logical procedures will be implemented in *CS47_proj_alu_logical.asm* under the *au_logical* label.

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

The normal procedures in au_normal are straightforward. Based on which operator is in $a2, we branch to the procedure that will perform the operator's desired operation.

```
au_normal:
        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

        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       au_normal_end
```

Figure 4: au_normal start and branch

"+" in $a2 will result in au_normal branching to the "addition" label and perform addition between $a0 and $a1, 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 return registers.

```
addition:
        add     $v0, $a0, $a1
        j       au_normal_end

subtraction:
        sub     $v0, $a0, $a1
        j       au_normal_end

multiplication:
        mult    $a0, $a1
        mflo    $v0
        mfhi    $v1
        j       au_normal_end

division:
        div     $a0, $a1
        mflo    $v0
        mfhi    $v1
        j       au_normal_end
```

Figure 5: au_normal arithmetic procedures

After the mathematical procedures are complete, au_normal will return control back to the label's original caller with the au_normal_end label.

```
au_normal_end:
        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_normal_end procedure

B. *Logical Procedures*

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

```
au_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

        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       au_logical_end
```

Figure 7: 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       au_logical_end

subtraction:
        jal     sub_logical
        j       au_logical_end

multiplication:
        jal     mul_signed
        j       au_logical_end

division:
        jal     div_signed
        j       au_logical_end

au_logical_end:
        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 8: au_logical calling procedures

1. *Addition and Subtraction*

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:
        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
        jal     add_sub_logical
        j       au_logical_end

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, 0xFFFFFFFF
        jal     add_sub_logical
        j       au_logical_end
```

Figure 9: add_logical and sub_logical

The idea behind add_sub_logical is simple: use $a2 as a submode operator, with 0x00000000 representing addition, and 0xFFFFFFFF representing subtraction.

To add a number in MIPS, one must use a binary system. One can only add one bit of the operands at a time, and must consider carry-out of the binary adding operations.

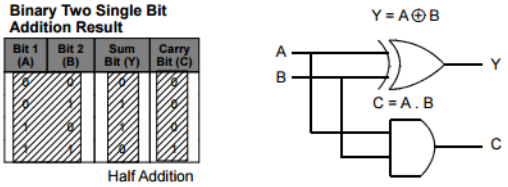

Figure 10: Half Adder [1]

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, a full adder that adds the all of the operands' bits are required.

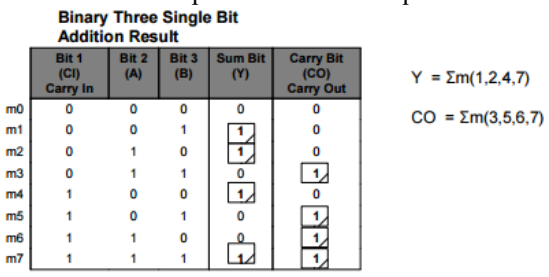

Figure 11: Full Adder truth table[1]

One needs to consider the carry-out bit for adding the next bit as the carry-in bit for the AND operations. Using the truth table of the Full Adder, one can simplify and deduce its design using a Karnaugh map, or K-map.

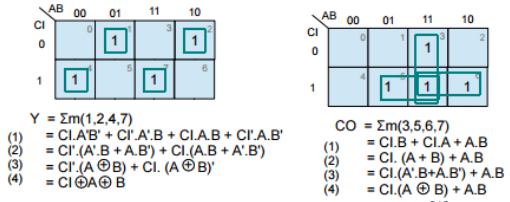

Figure 12: Full Adder Karnaugh map[1]

The expressions resulted from the K-map reduction will serve as the basis for the logical design of the Full Adder.

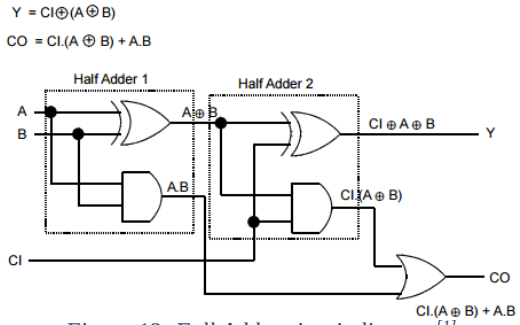

Figure 13: Full Adder circuit diagram[1]

The sum of adding each of the operands' bits is determined by a XOR operation involving the carry-in bit, the bits of the first and second operand. The final carry-out bit is determined by an OR operation involving the carry-out bits from the two half adders indicated. add_sub_logical utilizes the Full Adder design and to perform addition and subtraction, as subtraction is merely $a0 + ~($a1), which can be viewed as an addition. This is the reason behind using $a2 as a submode operator. One can obtain the two's complement form of $a1 by inverting using NOT $a1, and adding 1 to it.

```
twos_complement:
        subi    $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, $zero
        or      $a1, 0x1
        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 14: twos_complement

By treating the subtraction as addition, 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.

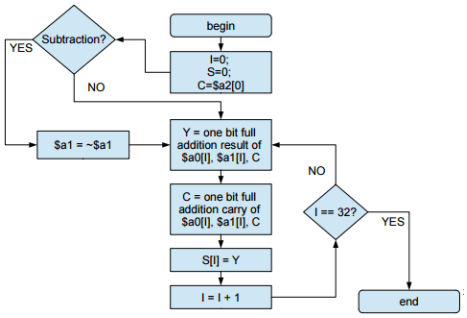

Figure 15: add_sub_logical flowchart[1]

From the flowchart, one can implement the logic behind the Full Adder and the loop to add two 32-bit numbers together.

```
add_sub_logical:
        subi    $sp, $sp, 40
        sw      $fp, 40($sp)
        sw      $ra, 36($sp)
        sw      $a0, 32($sp)
        sw      $a1, 28($sp)
        sw      $a2, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        addi    $fp, $sp, 40

        or      $t0, $zero, $zero
        or      $t1, $zero, $zero
        or      $t2, $zero, $zero
        extract_nth_bit($t2, $a2, $zero)
        beq     $a2, 0x00000000, add_sub_logical_loop
        not     $a1, $a1

add_sub_logical_loop:
        beq     $t0, 0x20, add_sub_logical_end
        extract_nth_bit($t3, $a0, $t0)
        extract_nth_bit($t4, $a1, $t0)
        xor     $s4, $t3, $t4
        xor     $s5, $t2, $s4
        and     $s6, $t3, $t4
        and     $s7, $t2, $s4
        or      $t2, $s6, $s7
        insert_to_nth_bit($v0, $t0, $s5, $t9)
        addi    $t0, $t0, 0x1
        j       add_sub_logical_loop

add_sub_logical_end:
        move    $v1, $t2

        lw      $fp, 40($sp)
        lw      $ra, 36($sp)
        lw      $a0, 32($sp)
        lw      $a1, 28($sp)
        lw      $a2, 24($sp)
        lw      $s4, 20($sp)
        lw      $s5, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7, 8($sp)
        addi    $sp, $sp, 40
        jr      $ra
```

Figure 16: add_sub_logical

add_logical will call add_sub_logical with 0x00000000 in $a2, and sub_logical will call add_sub_logical with 0xFFFFFFFF in $a2. Since $a2 is 0xFFFFFFFF in sub_logical, add_sub_logical will call two's_complement and obtain $a1's two's complement for addition.

2. *Multiplication*

To perform multiplication, multiplying bits is not as simple as addition. Multiplying negative numbers can be tricky, so it is better to split the work into two camps: one procedure for handling unsigned number operations and another one exclusively used to determine the sign of the result. Hence, we will split the work up into mul_unsigned and mul_signed.

The logical design for signed multiplication is listed below:

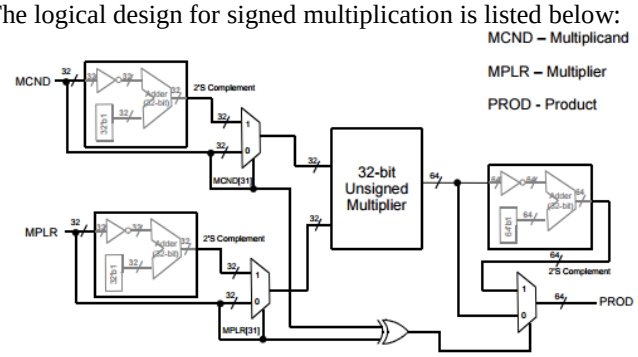

Figure 17: Signed Multiplication design[2]

Firstly, we will obtain the twos complement of the multiplicand in $a0 if it is negative and the multiplier in $a1 if it is negative. This will allow us to perform unsigned multiplication and append the signed bits after unsigned multiplication. Before returning it in the results we must make sure the results of our signed multiplication result in two 32-bit results in $v0 and $v1, and will be done via sign extension through obtaining the twos_complement in 64 bits. The results in $v0 and $v1 will have their signs determined by a XOR operation between original multiplicand and multiplier's signs. For a 32-bit unsigned multiplier, the logical design for it is listed below:

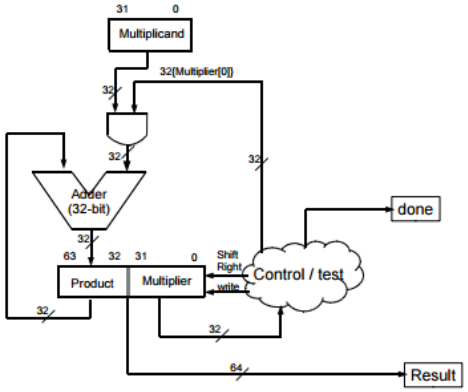

Figure 18: Unsigned Multiplication design[2]

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:

MCND	MPLR	MCND AND MPLR
0	0	0
0	1	0
1	0	0
1	1	1

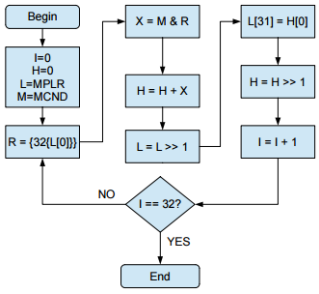

Figure 19: Unsigned Multiplication flowchart[2]

Following the flowchart, the AND operations will occur 32 times, with the multiplicand and multiplier both shifting to ensure the correct bits are ANDed and the result stored in a different bit. Below is the implementation of mul_unsigned:

```
mul_unsigned:
        subi    $sp, $sp, 40
        sw      $fp, 40($sp)
        sw      $ra, 36($sp)
        sw      $a0, 32($sp)
        sw      $a1, 28($sp)
        sw      $a2, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        addi    $fp, $sp, 40

        or      $t5, $zero, $zero
        or      $t6, $zero, $zero
        move    $s4, $a0
        move    $s5, $a1
        or      $s6, $zero, $zero
        or      $s7, $zero, $zero

mul_unsigned_loop:
        beq     $t5, 0x20, mul_unsigned_end

        extract_nth_bit($a0, $s4, $zero)
        jal     bit_replicator
        move    $s6, $v0
        and     $s7, $s5, $s6
        move    $a0, $t6
        move    $a1, $s7
        jal     add_logical
        move    $t6, $v0
        srl     $s4, $s4, 0x1

        extract_nth_bit($t7, $t6, $zero)

        li      $t8, 0x1F
        insert_to_nth_bit ($s4, $t8, $t7, $t9)
        srl     $t6, $t6, 0x1

        addi    $t5, $t5, 0x1
        j       mul_unsigned_loop

mul_unsigned_end:
        move    $v0, $s4
        move    $v1, $t6

        lw      $fp, 40($sp)
        lw      $ra, 36($sp)
        lw      $a0, 32($sp)
        lw      $a1, 28($sp)
        lw      $a2, 24($sp)
        lw      $s4, 20($sp)
        lw      $s5, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7, 8($sp)
        addi    $sp, $sp, 40
        jr      $ra
```

Figure 20: 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:

```
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
        move    $a2, $a0
        move    $s5, $a1
        move    $a3, $a1

        jal     twos_complement_if_neg
        move    $s4, $v0
        move    $a0, $s5
        jal     twos_complement_if_neg
        move    $s5, $v0

        move    $a0, $s4
        move    $a1, $s5
        jal     mul_unsigned
        move    $s4, $v0
        move    $s5, $v1

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

        xor     $t9, $s6, $s7
        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 21: mul_signed

For the result, recall that $v0 will contain the LO parts of the operation, and $v1 will contain the HI parts. The reason behind this is because multiplying two 32-bit numbers will result in a 64-bit result, and cannot be contained in 1 MIPS register, which is 32-bit. Therefore, we must first ensure the result complies with $v0 and $v1 in their correct form.

```
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 22: 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 the work will be split between a signed procedure and unsigned procedure. The dividend will be located in $a0 while the divisor will be located in $a2. The quotient will be returned in $v0 and remainder returned in $v1.

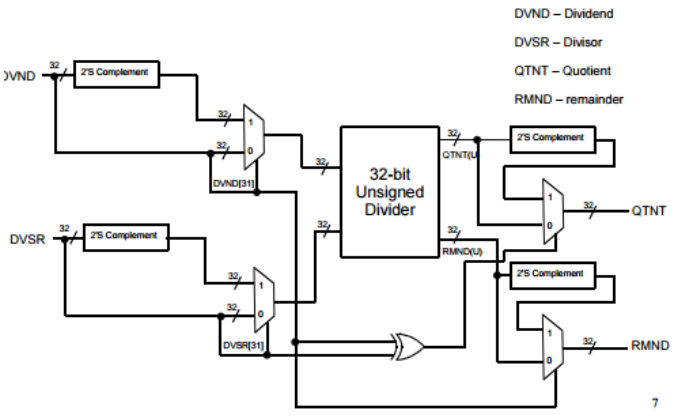

Figure 23: Signed Division design[3]

Once again, we will obtain the two's complement of the dividend and the divisor if they are negative. Then we will plug them in for unsigned division. Finally, the signs of the quotient and remainder will be determined by the dividend and divisor's original signs via a XOR operation.

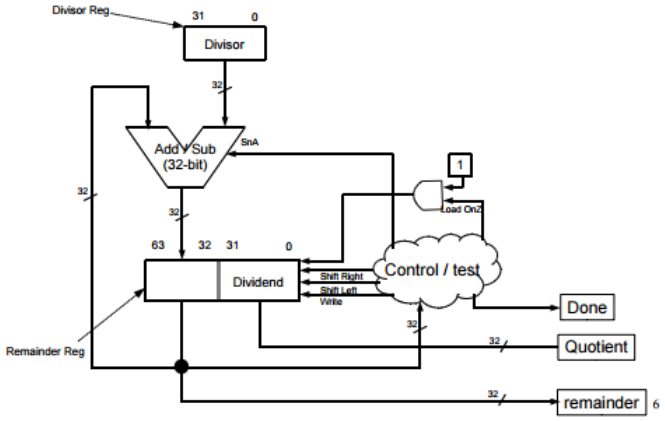

Figure 24: Unsigned Division design[3]

Here is a flowchart of the unsigned division procedure:

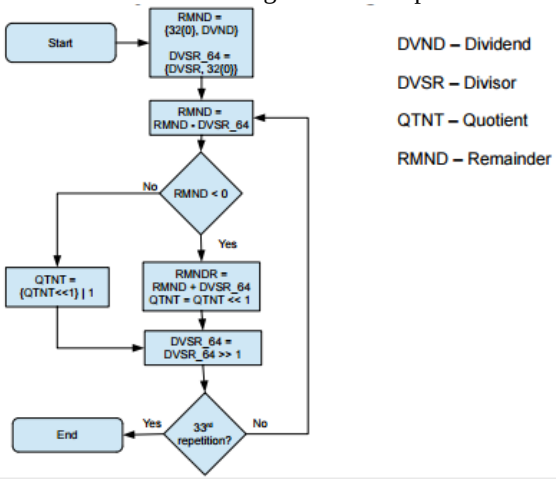

Figure 25: Unsigned Division flowchart[3]

The division is similar to the multiplication part. A loop that occurs for 32 repetitions will be utilized. To simulate a 64-bit number being divided, the number in the remainder register will be shifted to the left. This will allow us to insert the 31st bit of the quotient register into its 0th position. Subsequently, the contents of the quotient register will be shifted to the left. The dividend will eventually shift enough to contain the remainder and the quotient.

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

```
div_unsigned:
        subi    $sp, $sp, 40
        sw      $fp, 40($sp)
        sw      $ra, 36($sp)
        sw      $a0, 32($sp)
        sw      $a1, 28($sp)
        sw      $a2, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        addi    $fp, $sp, 40

        or      $t5, $zero, $zero
        or      $t6, $zero, $zero
        move    $s4, $a0
        move    $s5, $a1
        or      $s6, $zero, $zero
        or      $s7, $zero, $zero


div_unsigned_loop:
        beq     $t5, 0x20, div_unsigned_end

        sll     $t6, $t6, 0x1
        li      $t8, 0x1F
        extract_nth_bit($s7, $s4, $t8)
        insert_to_nth_bit ($t6, $zero, $s7, $t9)
        sll     $s4, $s4, 0x1
        move    $a0, $t6
        move    $a1, $s5
        jal     sub_logical
        move    $s6, $v0

        bltz    $s6, div_unsigned_loop_end
        move    $t6, $s6
        li      $t8, 0x1
        insert_to_nth_bit($s4, $zero, $t8, $t9)

div_unsigned_loop_end:
        addi    $t5, $t5, 0x1
        j       div_unsigned_loop

div_unsigned_end:
        move    $v0, $s4
        move    $v1, $t6

        lw      $fp, 40($sp)
        lw      $ra, 36($sp)
        lw      $a0, 32($sp)
        lw      $a1, 28($sp)
        lw      $a2, 24($sp)
        lw      $s4, 20($sp)
        lw      $s5, 16($sp)
        lw      $s6, 12($sp)
        lw      $s7, 8($sp)
        addi    $sp, $sp, 40
        jr      $ra
```

Figure 26: 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.

```
div_signed:
        subi    $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    $s4, $a0
        move    $a2, $a0
        move    $s5, $a1
        move    $a3, $a1

        jal     twos_complement_if_neg
        move    $s4, $v0
        move    $a0, $s5
        jal     twos_complement_if_neg
        move    $s5, $v0
        move    $a0, $s4
        move    $a1, $s5
        jal     div_unsigned

        move    $s4, $v0
        move    $s5, $v1
```

Figure 27: div_signed start

As one can see, twos_complement_if_neg_is called just as it is called in mul_signed.

```
determine_sign_of_Q:
        li      $t8, 0x1F
        extract_nth_bit($s6, $a2, $t8)
        extract_nth_bit($s7, $a3, $t8)

        xor     $s0, $s6, $s7
        move    $s1, $s4
        beq     $s0, $zero, determine_sign_of_R

        move    $a0, $s1
        jal     twos_complement
        move    $s1, $v0

determine_sign_of_R:
        li      $t8, 0x1F
        extract_nth_bit($s0, $a2, $t8)
        move    $s2, $s5
        beq     $s0, $zero, div_signed_end

        move    $a0, $s5
        jal     twos_complement
        move    $s2, $v0
```

Figure 28: Determining the signs of the quotient and remainder

The signs of the quotient and remainder will be determined the aforementioned XOR operation between the original operands.

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

        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 29: div_signed_end

Please refer to diagram 23 to refresh the information provided in reading the code.

V. Common Mistakes

A very common blunder when writing such code would be not taking the original signs of the operands into account when determining the signs of the result. When writing this code at one point, a XOR operation was performed with the two's compliment versions of the operands resulted in incorrect signs for the signed multiplication and division results.

One must also consider the fact that most procedures take $a0 and $a1 as parameters and results are saved in $v0 and $v1. A user must not forget to move the desired contents of a register to $a0 and $a1 to be used in a process such as add_logical. One must also remember to move the results out of $v0 and $v1 and store it in another register if one wishes to use those values later.

Saving and restoring the stack frame is also very important for every procedure as an incorrect frame pointer would lead to erroneous results. Another big blunder in the restoration of the frame is the absence of jr $ra. Without jr $ra, the procedure would not return to the caller properly and may yield incorrect results as the memory location is not returned and observed properly by the caller.

VI. Macros and other Procedures Used

```
.macro extract_nth_bit($regD, $regS, $regT)
        addi    $t9, $zero, 0x1
        sllv    $t9, $t9, $regT
        and     $regD, $regS, $t9
        srlv    $regD, $regD, $regT
.end_macro
.macro insert_to_nth_bit ($regD, $regS, $regT, $maskReg)
        addi    $maskReg, $zero, 0x1
        sllv    $maskReg, $maskReg, $regS
        not     $maskReg, $maskReg
        and     $regD, $regD, $maskReg
        add     $maskReg, $zero, $regT
        sllv $maskReg, $maskReg, $regS
        or   $regD, $regD, $maskReg
.end_macro
```

Figure 30: Bit-inserting and Extracting Macros

These macros are used to extract bits of an operand to be used to AND the other operands to determine their unsigned bits, as well as their results' signs. extract_nth_bit takes a $regD, which will be a resulting bit of 0 or 1, $regS is the source of the bit to extract, and $regT is the index of which $regS to extract. Likewise, insert_to_nth_bit takes a $regD, which is the register of a bit to insert, $regS is the source of the number to insert, $regT is the index of which bit of $regS to insert, and $maskReg is a temporary register to help shift and save the bit to insert.

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

        or      $v0, $a0, 0x00000000
        beq     $a0, $zero, bit_replicator_end
        li      $v0, 0xFFFFFFFF

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

Figure 31: bit_replicator

A bir 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:
        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

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

Figure 32: twos_complement_if_neg

This is the aforementioned gathering and obtainment of the two's complement of the operands for signed multiplication and signed division if the operands are negative. A cautionary step before these procedures is to save extra copies of the operands to XOR them and obtain the final result's signs.

VII. Results

Assemble and run the provided proj-auto-test.asm to check the implementation against actual results. The results of the logical procedures will be compared to the normal procedures.

```
(4 + 2)         normal=> 6      logcal=> 6      [matched]
(4 - 2)         normal=> 2      logcal=> 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     logcal=> 13     [matched]
(16 - -3)       normal=> 19     logcal=> 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     logcal=> -8     [matched]
(-13 - 5)       normal=> -18    logcal=> -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    logcal=> -10    [matched]
(-2 - -8)       normal=> 6      logcal=> 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    logcal=> -12    [matched]
(-6 - -6)       normal=> 0      logcal=> 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      logcal=> 0      [matched]
(-18 - 18)      normal=> -36    logcal=> -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     logcal=> -3     [matched]
(5 - -8)        normal=> 13     logcal=> 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    logcal=> -16    [matched]
(-19 - 3)       normal=> -22    logcal=> -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      logcal=> 7      [matched]
(4 - 3)         normal=> 1      logcal=> 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    logcal=> -90    [matched]
(-26 - -64)     normal=> 38     logcal=> 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 --
```

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

VIII. Conclusion

I've learned quite a lot through this project. The sheer amount of procedure and registers to use is staggering. The conventions, procedures, and branching methods used in this project are very useful knowledge in the path of compiler design for any future course in that regard. During the debugging process, I have learned to double and triple check

all registers used to make sure the correct copies of a variable or operand is saved, and is able to be used for a later process such as the sign check process of multiplication and division. This project made me reflect of how difficult digital circuits are to implement. Higher level languages such as Java and Scala have their work hidden from the user as the frame storing and restoration is handled by the compiler invisible to the user. I have deeply appreciated the knowledge I have gain in this project to allow dissection of other programming languages and discover their inner workings.

References:

[1] K. Patra. CS 47. Class Lecture, Topic: “Addition Subtraction Logic.” San Jose State University, San Jose, CA, November 14, 2016.

[2] K. Patra. CS 47. Class Lecture, Topic: “Multiplication Logic.” San Jose State University, San Jose, CA, November 16, 2016.

[3] K. Patra. CS 47. Class Lecture, Topic: “Division Logic.” San Jose State University, San Jose, CA, November 21, 2016.