Arithmetic Operations: Implementation Using MIPS Logic Instructions

Nathan Durrant
San Jose State University
nathan.durrant@sjsu.edu

***Abstract*—This report illustrates the implementation of basic arithmetic operations- addition, subtraction, multiplication, and division, using only logical procedures, such as shifting and Boolean logic. This implementation is then compared to MIPS normal operations in MARS, a MIPS assembler and runtime simulator.**

I. Introduction

Utilizing the MARS simulator, we will be able to perform basic arithmetic operations with the native MIPS operations (addition, subtraction, multiplication, and division). On the other hand, strict logical operations can be applied to perform these same operations. The project objectives are as following:

1. To install and run the MARS IDE
2. To perform arithmetic calculations by applying both MIPS mathematical operations and logical operations
3. To test the defined procedures using the MARS simulator

II. Installing and Running MARS IDE

A. *Installing the Simulator*

The assembly language ide and simulator can be downloaded from the following link:

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

This project was created using version 4.5, so downloading that same version would be ideal. There should be a jar file labelled "Mars4_5" which will begin the download upon clicking it.

B. *Opening Project in MARS*

To access the project, click on the given hyperlink to download the project zip file: Project Files. After extracting, you will have the following files.

1. "CS47_proj_alu_logical.asm"

   This file is where the MIPS logical instructions will be used to implement the basic arithmetic operations.

2. "CS47_proj_alu_normal.asm"

   This file is where the MIPS standard instruction set will be used to implement the basic arithmetic operations.

3. "cs47_proj_macro.asm"

   This file is where macros related to the logical implementation of arithmetic operations is to be defined.

4. "cs47_proj_procs.asm"

   This file contains the project procedures used and implemented.

5. "proj-auto-test.asm"

   This file will be used to test the results of the two separate implementations of arithmetic operations.

6. "cs47_common_macro.asm"

   This file contains all relevant macros which are not relevant to this project, but are useful for testing.

After opening the MARS4_5.jar file, it will lead to a new screen (fig. 1). From this window, use "File" then "Open" to find the directory where the zip file is (fig. 2). By clicking on each file, they can all be opened, which should result in 6 tabs with each of the file names after they have all been opened.

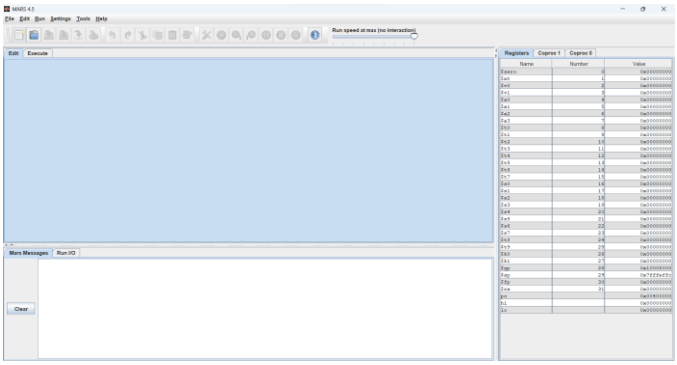

Fig. 1 MARS Window upon Opening

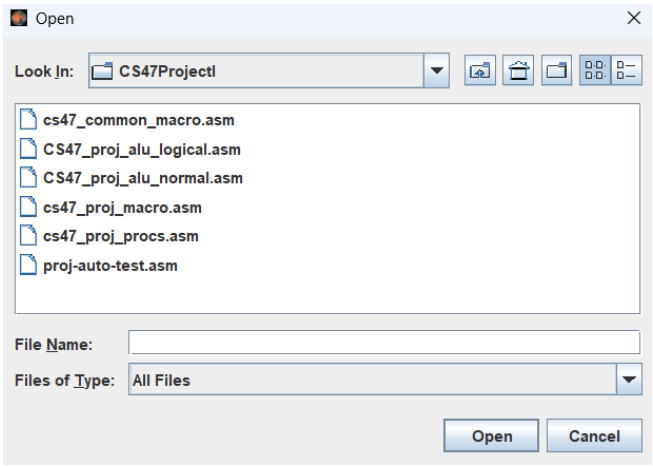

Fig 2 MARS File Window

C. *Configure MARS Settings*

For the implemented arithmetic operations to assemble and execute properly, certain settings must be chosen. Open the Settings tab in the top menu and adjust as needed so they match Fig 3.

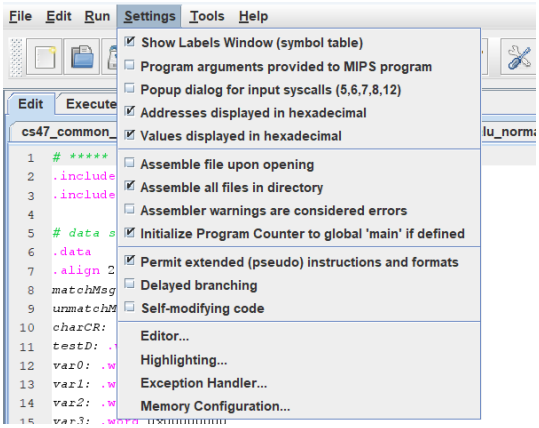

Fig 3 MARS Settings

III. Project Macros

Before the logical arithmetic procedures can be properly implemented, we must define some commonly used macros to serve as groundwork for later implantation.

A. *extract_nth_bit*

This macro will extract a desired bit from a given 32-bit pattern. This takes arguments as a register for the given pattern, a register with the position of the bit n, from 0 (LSB) to 31 (MSB), and a third register that will return 0x0 or 0x1.

```
# Macro: extract nth bit from a bit pattern
.macro extract_nth_bit($regD, $regS, $regT)
# $regD will contain 0x0 or 0x1 depending on the nth bit being 0 or
# $regS: source bit pattern
# $regT: Register containing bit position n (0-31)
li $regD, 0x1
sllv $regD, $regD, $regT
and $regD, $regD, $regS
srlv $regD,$regD, $regT
.end_macro
```

Fig. 4 extract_nth_bit Implementation

B. *insert_one_to_nth_bit*

This macro may seem a little misleading by name, but rather think of it as inserting one bit to nth bit. This macro takes four arguments: the desired bit to insert, the bit position n, the bit pattern to insert the bit into, and a temporary mask register. A mask register is for a technique called masking, informally used in the previous macro (Fig. 4) as well as this macro. It involves using 0x1 in a register, shifting it to the left n times, and using the AND operation to find the value of the nth bit. This technique is extended in this macro to also insert a desired bit at that location (Fig. 5).

```
# Macro: insert bit 1 or 0 at nth bit to a bit pattern
# does NOT only insert a 1
.macro insert_one_to_nth_bit($regD, $regS, $regT, $maskReg)
# $regD: bit pattern in which 1 or 0 is to be inserted at nth position
# $regS: value n, from which position of the bit to be inserted is (0-31)
# $regT: register that contains 0x1 or 0x0 (bit value to be inserted)
# $maskReg: register to hold temporary mask
li $maskReg 0x1
sllv $maskReg, $maskReg, $regS
not $maskReg, $maskReg
and $regD, $regD, $maskReg
sllv $regT, $regT, $regS
or $regD, $regD, $regT
srlv $regT, $regT, $regS
.end_macro
```

Fig. 5 insert_one_to_nth_bit Implementation

IV. Arithmetic Procedure Guidelines

As prefaced, there are two types of procedures that need to be implemented for this project. Basic mathematical operations (addition, subtraction, multiplication, and division) can be implemented in two different ways, reflected by the procedures to be implemented.

A. *Normal*

The normal procedure is used to calculate these basic mathematical operations using MIPS built in mathematical instructions. This procedure will take three arguments, determine, and perform the desired operation. To do this, $a2 must first be checked to branch to the desired operation.

1. $a2 = ‘+;

   This refers to the addition operation, denoted by ‘+’, which will be stored in ASCII.

2. $a2 = ‘-‘

   This refers to the subtraction operation, denoted by ‘-’, which will be stored in ASCII.

3. $a2 = ‘*’

   This refers to the multiplication operation, denoted by ‘*’, which will be stored in ASCII.

4. $a2 = ‘/’

   This refers to the division operation, denoted by ‘/’, which will be stored in ASCII.

```
au_normal:
        beq $a2, '+', addition
        beq $a2, '-', subtraction
        beq $a2, '*', multiplication
        beq $a2, '/', division

addition:
        add $v0, $a0, $a1
        j return

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

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

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

Fig 6. Implementation of branches and operations

B. *Logical Procedures*

The logical procedure functions in a similar way, taking the same arguments and returning the same results, though it requires more procedures to perform these operations on a logical level. Note that at the beginning of this procedure, registers must be stored to preserve the runtime environment.

```
au_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

        beq $a2, '+', add_logical
        beq $a2, '-', sub_logical
        beq $a2, '*', mul_logical
        beq $a2, '/', div_logical
```

Fig 7. Branches and Runtime Environment Storage

1. add_sub_logical

This procedure calculates the sum of two numbers, given in arguments $a0 and $a1. It performs addition or subtraction based on the desired mode, denoted by the third argument, $a2.

Binary Two Single Bit Addition Result

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

Half Addition

Fig 8. Half Addition Truth Table

The logic of single bit addition, taking place in a half adder, is shown above (Fig. 8). Notice that the possible results of this addition for the sum bit (Y) and the carry-out bit (C) resemble the truth table for the XOR and the AND operation, respectively. This means single bit addition can be implemented using each to create a half adder (Fig. 9)

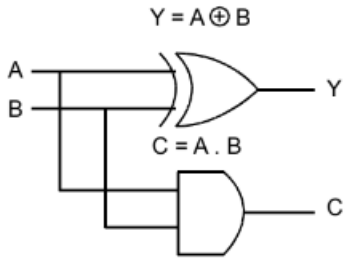

Fig. 9 Half adder

To perform addition with more bits, we must go beyond this half adder to also consider the carry-in bit, which is done in full addition (Fig 10).

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 = Σm(1,2,4,7)

CO = Σm(3,5,6,7)

Full Addition

Fig. 10 Full Addition Truth Table

To implement this in a fashion like the half adder, it is first necessary to simplify the output, given by the two SOPs. This can be done by using a K-Map.

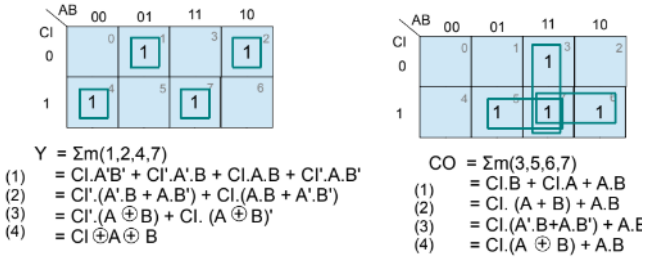

Fig. 11 Full adder K-Map

These expressions in Fig. 11 are strategically simplified in a way that will use the XOR gate again, used previously in the half adder. This provides the groundwork for a similar full adder design- making use of the previously constructed half adder (Fig. 12).

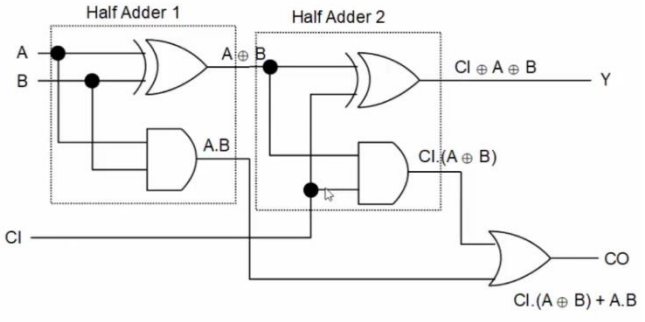

Fig. 12 Full Adder

Since the simplified sum and carry-out expressions use the same values as the result of the half adder, some of the Boolean expressions in the second half adder come from the result of the first. By chaining two half adders, a full adder can be created, with an OR operation on the carry out bits of each half adder to determine the final carry-out bit.

To extend this, a number of any number of bits can be computer using full adders. By using the final carry out bits as the carry in for the next full adder, the full adders can be chained in a ripple adder (Fig. 13).

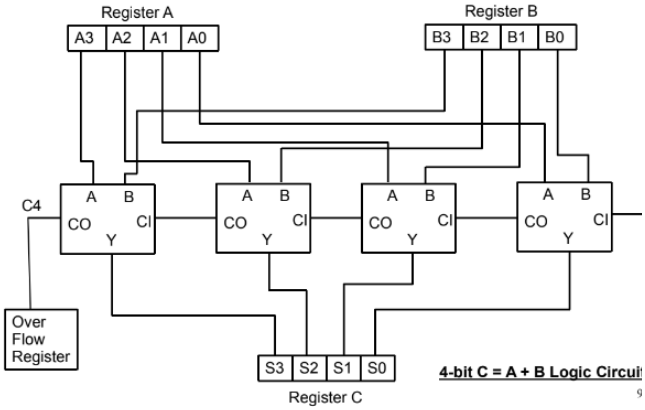

Fig. 13 Binary Ripple Carry Addition

The binary ripple adder above computes 4-bit binary addition, requiring 4 full adders. This extends to any number of n bits to be added, as they will likewise need n full adders.

It is also important to note that this is only for addition. To extend the operation to subtraction, we can reuse the addition operation, making the second operand negative. This can be done by simply taking the inverse and incrementing the second operand, taking the two's complement of it.

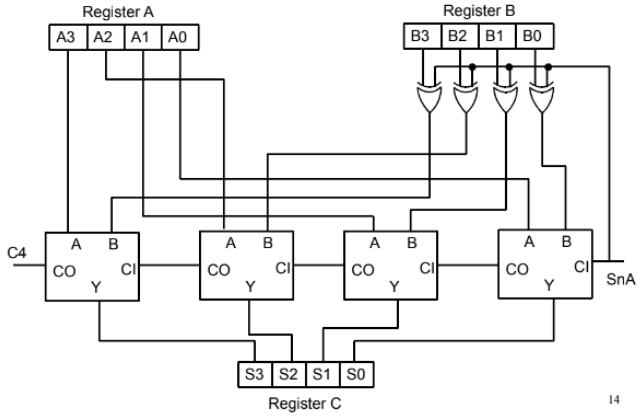

Fig. 14 Binary Ripple Carry Addition/Subtraction

Above is the extension of the binary ripple carry adder to include subtraction. The signal, SnA, is set to 1 in the case of subtraction, which will invert every bit in the second operand, and include a 1 as the initial carry in, therefore B = ~B+1.

This concept will be used to create and implement a logic-based algorithm in order to perform addition and subtraction (Fig. 15 and Fig. 16)

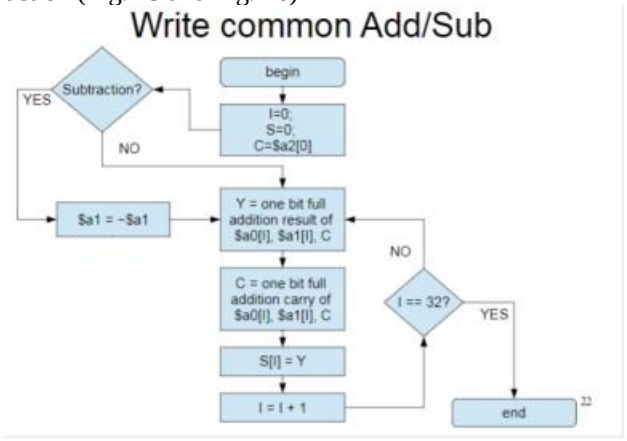

Fig. 15 Addition/Subtraction Algorithm

```
subtraction_mode:
        not $a1, $a1

addition_mode:
        extract_nth_bit($t0, $a0, $t2)
        extract_nth_bit($t1, $a1, $t2)
        xor $t5, $t0, $t1 # A XOR B
        and $t6, $t0, $t1 # A AND B
        and $t7, $t3, $t5 # CI AND (A XOR B)
        xor $t4, $t3, $t5 # Intermediate sum- CI XOR (A XOR B)
        or $t3, $t7, $t6 # COUT- CI AND (A XOR B) OR (A AND B)
        insert_one_to_nth_bit($v0, $t2, $t4, $t9)
        addi $t2, $t2, 0x1
        blt $t2, 32, addition_mode
        # return final carryout in $v1
        move $v1, $t3
        j au_logical_return
```

Fig. 16 add_logical and sub_logical Implementation

With addition and subtraction fully implemented in out logical procedures, we can now move on to performing multiplication. Before this can be implemented, however, additional procedures must first be implemented that will be useful later when it comes to multiplication.

2. twos_complement

This procedure will convert any input into twos complement form. $a0 is the only argument in this procedure, and is the desired number to convert.

```
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
        li $a1, 1
        li $a2, '+'
        jal au_logical

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

Fig. 17 twos_complement Implementation

3. twos_complement_64bit

This procedure computes the twos compliment form of a 64-bit number, taking two arguments for the upper and lower half of the pattern. It first inverts the lower half, Lo, while adding one. It then inverts the higher half, Hi, and adds the carry out from the previous addition with Lo.

```
twos_complement_64bit:
        addi $sp, $sp, -28
        sw $fp, 28($sp)
        sw $ra, 24($sp)
        sw $a0, 20($sp)
        sw $a1, 16($sp)
        sw $a2, 12($sp)
        sw $s1, 8($sp)
        addi $fp, $sp, 28

        move $s1,$a1 #store Hi
        not $a0,$a0 #inverse Lo
        li $a2, '+'
        li $a1, 1
        jal au_logical #inverse Lo + 1
        move $a0,$s1 #store Hi
        move $s1,$v0 #store result
        move $a1,$v1 #store carryout
        not $a0,$a0
        jal au_logical #inverse Hi + carryout
        move $v1,$v0 #return both results
        move $v0,$s1

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

Fig. 18 twos_complement_64bit Implementation

4. bit_replicator

This procedure does exactly as it sounds, it replicates a bit. If the given argument is 1, it returns a 32-bit pattern of all ones, 0xFFFFFFFF, and will likewise return 0x00000000 if it is 0.

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

        beqz $a0, bit_replicator_zero

        li $v0, 0xFFFFFFFF
        j bit_replicator_return

bit_replicator_zero:
        li $v0, 0x00000000

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

Fig. 19 bit_replicator Implementation

5. mul_unsigned

With the previous procedures added to the au_logical program, multiplication can now be effectively implemented. This procedure, like the MIPS normal instruction, will take two arguments, the multiplicand and multiplier, returning the 64-bit result in two registers.

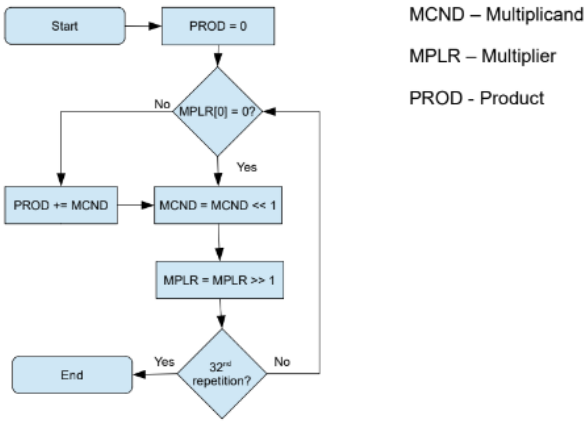

Fig. 20 Binary Multiplication Algorithm

As multiplication is simply a repeated addition procedure, we will be able to use the above algorithm to implement multiplication on a logical level (Fig. 21).

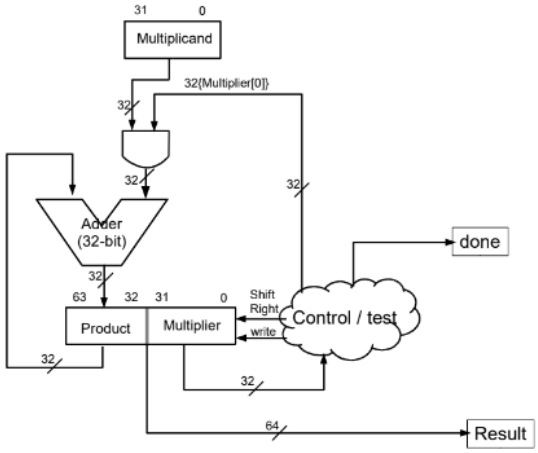

Fig. 21 Unsigned Sequential Multiplier

The sequential multiplier will generate a 64-bit product, meaning that it requires two 32-bit registers to store the output. As it checks the multiplier and adds to the product accordingly, the procedure will shift the multiplier to the right. Once it has completed the process, and shifted to the right 32 times, the multiplier will eventually be shifted out of the register, which will be entirely taken up by the 64-bit product.

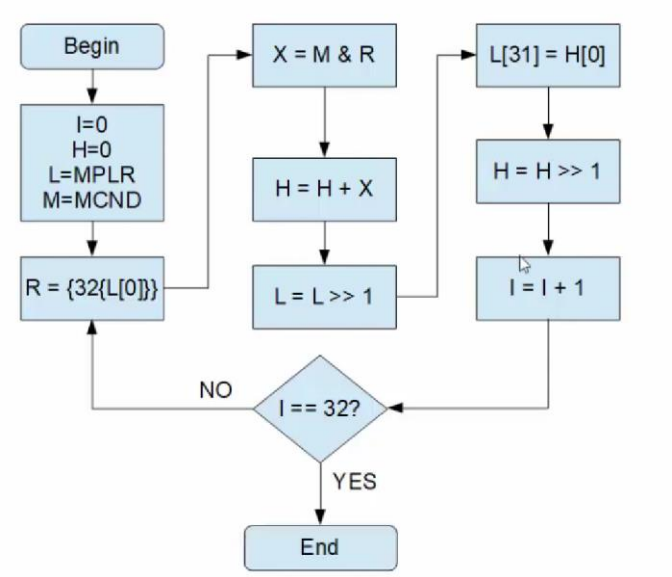

Fig. 22 Unsigned Multiplication Algorithm

Taking the concept of the sequential multiplier (Fig. 21), an algorithm can be generated to implement this using MIPS logical instructions (Fig. 22). This checks if the current bit in the multiplier (L[0]) is equal to one using an AND gate, and if so will add the multiplicand to the product, or the upper half of the multiplier register. The Hi and Lo of the combined registers are both shifted, and the algorithm repeats until it has completed 32 cycles. At this point, the 64-bit register will be the product, returning $v0, or Hi, and $v1, or Lo.

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

        li $s0, 0 # I=0
        li $v1, 0 # H=0
        move $v0, $a1 # L = MLPR
mul_loop:
        extract_nth_bit($t0, $v0, $zero)
        move $s1, $v0
        move $s2, $a0
        move $a0, $t0
        jal bit_replicator # R={32{L[0]}}
        move $t0,$v0
        move $v0,$s1
        move $a0,$s2
        and $t1, $t0, $a0 # X = M & R
        move $s1, $v0
        move $s2, $a0
        move $a0, $v1
        move $a1, $t1
        li $a2, '+' # H = H + X
        jal au_logical
        move $v1, $v0
        move $v0, $s1
        move $a0, $s2
        srl $v0, $v0, 1 # L = L >> 1
        extract_nth_bit($t2, $v1, $zero) # H[0]
        li $t3, 31
        insert_one_to_nth_bit($v0, $t3, $t2, $t4) # L[31] = H[0]
        srl $v1, $v1, 1 # H = H >> 1
        addi $s0, $s0, 1 # I++
        blt $s0, 32, mul_loop

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

Fig. 23 Unsigned Multiplication Implementation

6. mul_signed

The mul_unsigned procedure adds a lot of functionality to the logical arithmetic operations, though it needs to be extended to support signed multiplication. For signed multiplication, the absolute value of each operand, found with the twos_complement_if_neg, which can be used for mul_unsigned, and the sign of the output can be found separately. This is done with the use of an XOR gate between the MSB of the multiplier and multiplicand. If it is 1, then the output must be negative, which the twos_complement_64bit procedure can be called to convert the product (Fig. 24)

```
mul_signed:
        addi $sp, $sp, -32
        sw $fp, 32($sp)
        sw $ra, 28($sp)
        sw $s0, 24($sp)
        sw $s1, 20($sp)
        sw $s2, 16($sp)
        sw $a0, 12($sp)
        sw $a1, 8($sp)
        addi $fp, $sp, 32
        move $s1, $a0
        move $s2, $a1
        jal twos_complement_if_neg # Make N1 two's complement if negative
        move $s0, $v0
        move $a0, $a1
        jal twos_complement_if_neg # Make N2 two's complement if negative
        move $a1, $v0
        move $a0, $s0
        jal mul_unsigned
        li $t2, 31
        extract_nth_bit($t0, $s1, $t2) # Determine sign S of result
        extract_nth_bit($t1, $s2, $t2)
        xor $t0, $t0, $t1 # S = $a0[31] XOR $a1[31]
        move $a0, $v0
        move $a1, $v1
        beqz $t0, mul_signed_return
        jal twos_complement_64bit # If S is 1, use the 'twos_complement_64bit' to determine twos comp
mul_signed_return:
        lw $fp, 32($sp)
        lw $ra, 28($sp)
        lw $s0, 24($sp)
        lw $s1, 20($sp)
        lw $s2, 16($sp)
        lw $a0, 12($sp)
        lw $a1, 8($sp)
        addi $sp, $sp, 32
        jr $ra
```

Fig. 24 Signed Multiplication Implementation

7. div_unsigned

Division can likewise be implemented starting with an unsigned procedure. It will take two arguments, the dividend and the divisor. This operation also returns a 64-bit value, comprised of the quotient and the remainder.

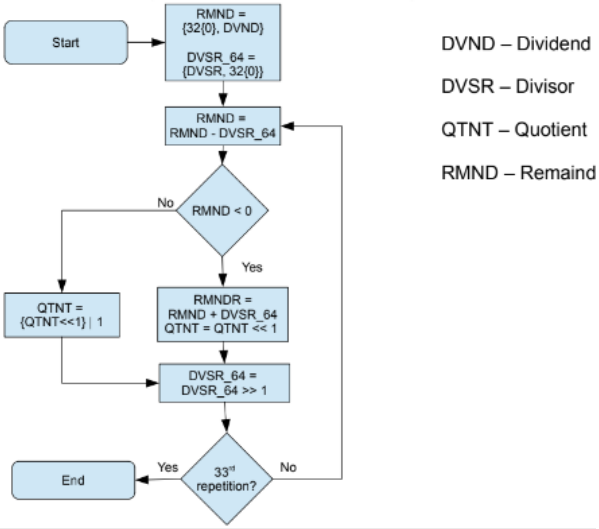

Fig. 25 Binary Division Algorithm

This algorithm divides the dividend by the greatest number possible, repeating 32 times similarly to the multiplication algorithm, which will output the quotient and remainder.

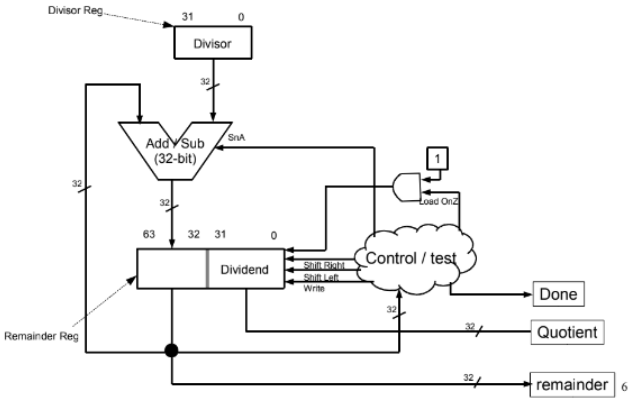

Fig. 26 Binary Division Circuit

The division circuit (Fig. 26) directly reflects the algorithm binary division algorithm (Fig. 25). These serve as the basis for an unsigned division algorithm that can be implemented with MIPS (Fig. 27).

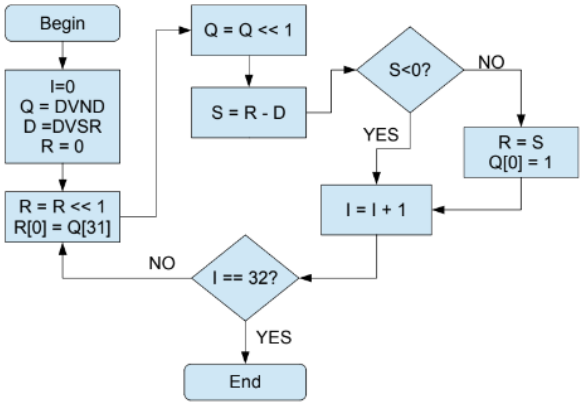

Fig. 27 Unsigned Division Algorithm

This algorithm emulates a 64-bit register- as the remainder is shifted to the left, the MSB is pulled from the quotient. After 32 repeitions of the algorithm, the dividend will be completely shifted of the register to contain the quotient and remainder.

```
div_unsigned:
        addi $sp, $sp, -32
        sw $fp, 32($sp)
        sw $ra, 28($sp)
        sw $s0, 24($sp)
        sw $s1, 20($sp)
        sw $s2, 16($sp)
        sw $a0, 12($sp)
        sw $a1, 8($sp)
        addi $fp, $sp, 32
        li $s0, 0 # I = 0
        move $s1, $a0 # Q = DVND
        li $s2, 0 # R = 0
div_loop:
        sll $s2, $s2, 1 # R = R << 1
        li $t0, 31
        extract_nth_bit($t1, $s1, $t0)
        insert_one_to_nth_bit($s2, $zero, $t1, $t9) # R[0] = Q[31]
        sll $s1, $s1, 1 # Q = Q << 1
        move $a0, $s2
        li $a2, '-'
        jal au_logical # S = R - D
        bltz $v0, div_loop_return
        move $s2, $v0 # R = S
        li $t2, 1
        insert_one_to_nth_bit($s1, $zero, $t2, $t9) # Q[0] = 1
div_loop_return:
        addi $s0, $s0, 1
        blt $s0, 32, div_loop
        move $v0, $s1
        move $v1, $s2
        lw $fp, 32($sp)
        lw $ra, 28($sp)
        lw $s0, 24($sp)
        lw $s1, 20($sp)
        lw $s2, 16($sp)
        lw $a0, 12($sp)
        lw $a1, 8($sp)
        addi $sp, $sp, 32
        jr $ra
```

Fig. 28 Unsigned Division Implementation

8. div_signed

The same logic as the mul_signed procedure can be applied to likewise add signed functionality to the logical division operations.

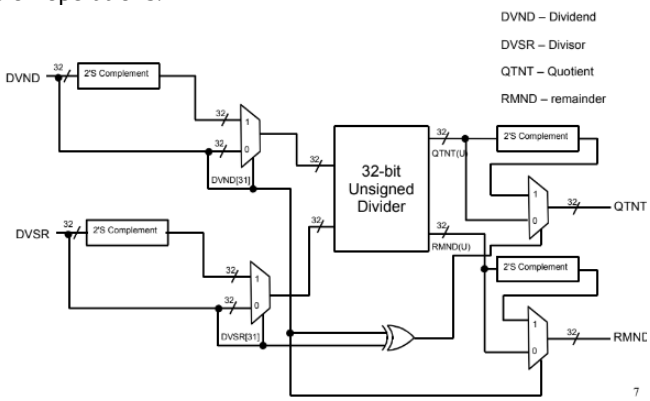

Fig 29 Signed Divison Circuit

The absolute value of the dividend and divisor will be similarly passed to the unsigned division procedure. After this has been completed, the original operand's signs will be tested to see if the product must be converted to twos complement form, once again using an XOR gate to find.

```
div_signed:
        addi $sp, $sp, -40
        sw $fp, 40($sp)
        sw $ra, 36($sp)
        sw $s0, 32($sp)
        sw $s1, 28($sp)
        sw $s2, 24($sp)
        sw $s3, 20($sp)
        sw $s4, 16($sp)
        sw $a0, 12($sp)
        sw $a1, 8($sp)
        addi $fp, $sp, 40
        move $s0, $a0 # N1
        move $s1, $a1 # N2
        jal twos_complement_if_neg
        move $s2, $v0
        move $a0, $a1
        jal twos_complement_if_neg
        move $a1, $v0
        move $a0, $s2
        jal div_unsigned
        move $a0, $v0 # Q
        move $s3, $v0
        move $a1, $v1 # R
        move $s4, $v1
        # Determine S of Q
        li $t0, 31
        extract_nth_bit($t1, $s0, $t0)
        extract_nth_bit($t2, $s1, $t0)
        move $s2, $t1
        xor $t0, $t1, $t2 # $a0[31] xor $a1[31]
        beqz $t0, remainder_check # If S is 1, two's complement of Q
        move $a0, $s3
        jal twos_complement
        move $s3, $v0
remainder_check:
        beqz $s2, div_signed_return # If S is 1, two's complement of R
        move $a0, $s4
        jal twos_complement
        move $s4, $v0
div_signed_return:
        move $v0, $s3
        move $v1, $s4

        lw $fp, 40($sp)
        lw $ra, 36($sp)
        lw $s0, 32($sp)
        lw $s1, 28($sp)
        lw $s2, 24($sp)
        lw $s3, 20($sp)
        lw $s4, 16($sp)
        lw $a0, 12($sp)
        lw $a1, 8($sp)
        addi $sp, $sp, 40
        jr $ra
```

Fig 30 Signed Division Implementation

V. TESTING

Now addition, subtraction, multiplication, and division has been successfully added to the algorithm using both normal and logical procedures. It is now possible to test and compare both implementations. To do this, assemble and run "proj-auto-test.asm" (Fig. 31).

```
(4 + 2)          normal => 6     logical => 6    [matched]
(4 - 2)          normal => 2     logical => 2    [matched]
(4 * 2)          normal => HI:0 LO:8     logical => HI:0 LO:8    [matched]
(4 / 2)          normal => R:0 Q:2       logical => R:0 Q:2      [matched]
(16 + -3)        normal => 13    logical => 13   [matched]
(16 - -3)        normal => 19    logical => 19   [matched]
(16 * -3)        normal => HI:-1 LO:-48  logical => HI:-1 LO:-48         [matched]
(16 / -3)        normal => R:1 Q:-5      logical => R:1 Q:-5     [matched]
(-13 + 5)        normal => -8    logical => -8   [matched]
(-13 - 5)         normal => -18   logical => -18  [matched]
(-13 * 5)         normal => HI:-1 LO:-65  logical => HI:-1 LO:-65         [matched]
(-13 / 5)         normal => R:-3 Q:-2     logical => R:-3 Q:-2    [matched]
(-2 + -8)         normal => -10   logical => -10  [matched]
(-2 - -8)         normal => 6     logical => 6    [matched]
(-2 * -8)         normal => HI:0 LO:16    logical => HI:0 LO:16   [matched]
(-2 / -8)         normal => R:-2 Q:0      logical => R:-2 Q:0     [matched]
(-6 + -6)         normal => -12   logical => -12  [matched]
(-6 - -6)         normal => 0     logical => 0    [matched]
(-6 * -6)         normal => HI:0 LO:36    logical => HI:0 LO:36   [matched]
(-6 / -6)         normal => R:0 Q:1       logical => R:0 Q:1      [matched]
(-18 + 18)        normal => 0     logical => 0    [matched]
(-18 - 18)        normal => -36   logical => -36  [matched]
(-18 * 18)        normal => HI:-1 LO:-324         logical => HI:-1 LO:-324        [matched]
(-18 / 18)        normal => R:0 Q:-1      logical => R:0 Q:-1     [matched]
(5 + -8)          normal => -3    logical => -3   [matched]
(5 - -8)          normal => 13    logical => 13   [matched]
(5 * -8)          normal => HI:-1 LO:-40  logical => HI:-1 LO:-40         [matched]
(5 / -8)          normal => R:5 Q:0       logical => R:5 Q:0      [matched]
(-19 + 3)         normal => -16   logical => -16  [matched]
(-19 - 3)         normal => -22   logical => -22  [matched]
(-19 * 3)         normal => HI:-1 LO:-57  logical => HI:-1 LO:-57         [matched]
(-19 / 3)         normal => R:-1 Q:-6     logical => R:-1 Q:-6    [matched]
(4 + 3)           normal => 7     logical => 7    [matched]
(4 - 3)           normal => 1     logical => 1    [matched]
(4 * 3)           normal => HI:0 LO:12    logical => HI:0 LO:12   [matched]
(4 / 3)           normal => R:1 Q:1       logical => R:1 Q:1      [matched]
(-26 + -64)       normal => -90   logical => -90  [matched]
(-26 - -64)       normal => 38    logical => 38   [matched]
(-26 * -64)       normal => HI:0 LO:1664  logical => HI:0 LO:1664         [matched]
(-26 / -64)       normal => R:-26 Q:0     logical => R:-26 Q:0    [matched]


Total passed 40 / 40
*** OVERALL RESULT PASS ***

-- program is finished running --
```

Fig. 31 Auto Test Output

VI. Conclusion

Over the course of this project, I learned a lot about different MIPS operations, as well as logical programming techniques. The electrical engineering concepts, like Boolean algebra, KMAPs, and ALUs, bolstered my understanding of computer systems. With how far computers have come today, we might neglect the barebone basics of computer operations. The low-level computer science concepts may seem like less relevant at first, but I can confidently say with my firsthand experience that applying these concepts can be enriching and fulfilling, deepening my appreciation for computer systems in its entirety.

References

[1] K. Patra. CS 47. Class Lecture, Topic: “Addition / Subtraction Logic.” San Jose State University, San Jose, CA, November 6, 2023.

[2] K. Patra. CS 47. Class Lecture, Topic: “Multiplication Logic.” San Jose State University, San Jose, CA, November 8, 2023.

[3]] K. Patra. CS 47. Class Lecture, Topic: “Division Logic.” San Jose State University, San Jose, CA, November 13, 2023.

[4] Chapter 3 of ‘Computer Organization & Design by Hennesy, Patterson.

[5] Chapter 4 of ‘Logic and Computer Design Fundamentals’ by Mano, Kime