Arithmetic Implemented By MIPS Logic Operation

Mebin Skaria

San Jose State University

Mebin.skaria@sjsu.edu

***Abstract*— This report focuses on the implementation of basic arithmetic's such as addition, subtraction, multiplication, and division with logical procedures such as binary shifting, masking, Boolean logic (no basic add, sub, mult, div instructions). These logical procedures results outputs are then compared to that with the results of standard (add, sub, mult, div) instructions in MARS (MIPS Assembler and Runtime Simulator).**

I. Introduction

MARS will allow for easy computation on each of the arithmetic operations with MIPS protocol. Addition, subtraction, multiplication and division will have two different operations, one done by MIPS standard instruction arithmetic set and one done by the self-made logical operations which very closely ties to how the processor does this on a hardware level. The project objectives are listed as so:

1. Successfully execute the MARS program.
2. Implement two modules that successfully achieves Arithmetic operations, one with MIPS Standard Instruction Set and another with only logical operations including binary shifting, masking, Boolean logic.
3. Test the procedures created and verify the results using MAR simulator.

To all the project objectives, provided are simple steps to successfully execute MARS, implement these two modules, and finally run and test these implemented procedures.

II. Preparation for Implementation of Arithmetic Operations

A. Installation and Execution

MARS simulator can easily be attained through the provided by first opening the link provided on any of the well-known and popularly used browsers available:
http://courses.missouristate.edu/KenVollmar/MARS/

While this report was being written, the version used is MARS 4.5. While it is most likely to not change the implementation of the arithmetic modules, it should be noted that there are possibilities of small differences which could cause confusion, the best option is to also download version 4.5 to follow along.

B. Loading Project Files into the MARS Environment

Provided in the hyperlinked Link: Arithmetic Project Files will download the file "CS47ProjectI.zip" which when extracted will give you a folder containing the files

1.) *"cs47_common_macro.asm"*
This file contains all the necessary macros which are not relevant to the two modules that need to be created.

2.) *"CS47_proj_alu_logical.asm"*
This file will be where the logic for all logical arithmetic procedures will be implemented and held.

3.) *"CS47_proj_alu_normal.asm"*
This file will be where the logic for all MIPS standard instruction set (or if mentioned from now on, will be synonymous with "normal") arithmetic procedures will be implemented and held.

4.) *"cs47_proj_macro.asm"*
This file will be where all the macros necessary for arithmetic operations are implemented and held.

5.) *"cs47_proj_procs.asm"*
This file contains all printf procedures.

6.) *"proj-auto-test.asm"*
This file will be used to test the results of the two arithmetic modules.

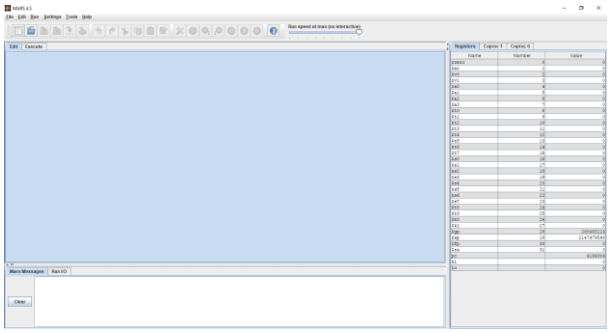

Fig. 1 MARS Starting Window

Open the MARS version downloaded, visually represented will be version Mars4_5.jar. From this window (fig. 1), individually open (fig. 2) all the files by going to file and the directory where you extracted the zip file and clicking each file.

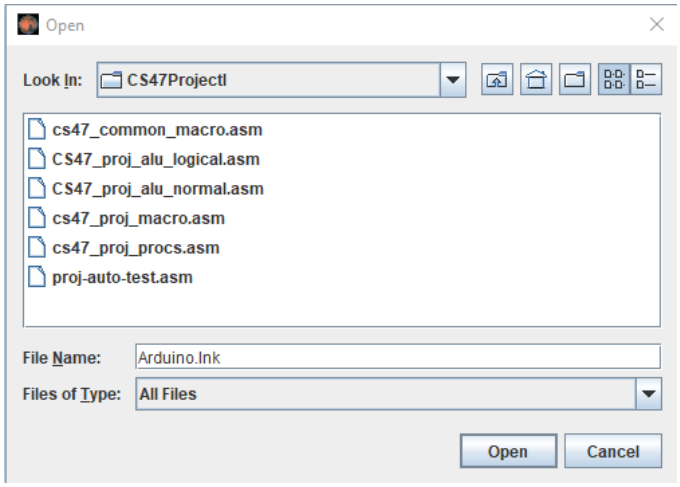

Fig 2 MARS Open File Window

If sucessfuly opened all 6 files names should have its own tab in the MARS environment.

C. *Configuring the MARS Environment Properly*

To allow proper compiling and execution of the to be implemented arithmetic modules, turn on 'Assembles all files in directory' and 'Initialize program counter to global main if defined' options in MARS settings tab (fig. 3)

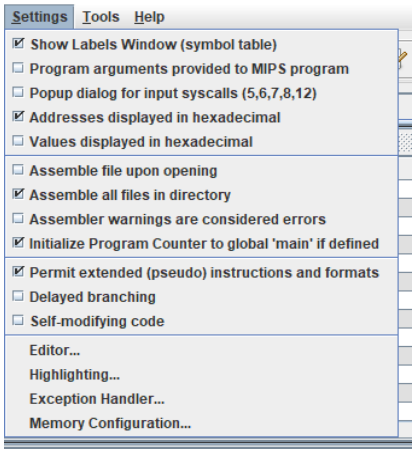

Fig 3 MARS Setting Tab

III. Arguments and Outputs of Arithmetic Modules

Now that the MARS system is downloaded, opened with all the project files, and configured with the appropriate settings, we successfully achieved the first objective of the project. Now we move on to the second objective of the project which is to implement normal and logical arithmetic operations.

A. *Normal Procedure*

The normal procedure will be implemented in the file "CS47_proj_alu_normal.asm" which takes in three arguments and returns two outputs.

1.) *Register $a0*
Holds the value for the First Operand
2.) *Register $a1*
Holds the value for the Second Operand
3.) Register $a2
Holds the operation which is being done as the appropriate ASCII symbol ('+','- ','*','/')

The output will be $v0 and $v1, $v0 will be used as the result in the Addition and Subtraction operations (+, -). $v0 and $v1 will be used together as a 64-bit register which is required when multiplying two 32 bit numbers or as $v0 – LO and $v1 – HI in the Multiplication operation (*). In the Division operation $v0 is used as the Quotient register and $v1 is used as the Remainder register. Although achieved differently from the normal procedure, the logical procedure will have the exact same arguments and outputs.

IV. Implementation of Normal Arithmetic Procedure

It should be stated that in more than 99% of the time one programs in MIPS the standard arithmetic instruction set should suffix for calculations, in which case the one percent being projects such as this one which attempts to achieve the same operations without this instruction set.

With the three arguments $a2 must be checked first and will allow the procedure to jump to the appropriate arithmetic operation desired by the user.

1.) *$a2 register equals '+'*
If the $a2 register's value is equivalent to '+' in ascii, the procedure should jump to the addition section of the program.

2.) *$a2 register equals '-'*
If the $a2 register's value is equivalent to '-' in ascii, the procedure should jump to the subtraction section of the program.

3.) *$a2 register equals '*'*
If the $a2 register's value is equivalent to '*' in ascii, the procedure should jump to the multiplication section of the program. After it should move Hi to $v1, and Lo to $v0

4.) *$a2 register equals '/'*
If the $a2 register's value is equivalent to '/' in ascii, the procedure should jump to the division section of the program. After it should move Hi to $v1 and Lo to $v0

```
au_normal:

        operation_check:

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

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

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

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

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

        end:
        jr      $ra
```

Fig 4 Implementation of au_normal

V. Implementation of Logical Arithmetic Procedure

A) *Setting up the Logical Arithmetic procedure*

We begin by using the operation branch we implemented in au_normal, but removing all the MIPS arithmetic instruction set, and instead leaving it for blank as is (Fig. 5). Also make sure to change the label from au_normal to au_logic.

```
au_logical:
    operation_check:

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

    addition:
    j end

    subtraction:
    j end

    multiplication:
    j end

    division:

    end:
    jr  $ra
```

Fig 6 Copying the Logic from au_normal and removing the instructions.

we need to consider the possible registers that will need to be stored and restored to and from the stack, this is because we want to keep the registers the same as before the procedure was called when the procedure returns its output for the developer to not have unexpected results. In the declaration of procedure, we said the outputs will be $v0 and $v1, which should be the only registers that change after the procedure is called. To achieve this, we need to create a store frame code and load frame code.

To store $fp, $ra, $a0, $a1, $a2, shift $sp to the end of the stack, and shift $fp to be the top of the stack (Fig 6.).

```
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)
    add $fp, $sp, 24
```

Fig 6 Implementation of the Store frame

Similarly, we do the same with load frame code, by restoring $fp, $ra, $a0, $a1, $a2, and shifting $sp back up to the previous stack (Fig 7.).

```
multiplication:
j end

division:

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
```

Fig 7 Implementation of the Load frame

B) *Addition Operation and Subtract Operation*

With the frame properly implemented we can continue to implement the addition operation with logical operations. But first let's cover the logic.

In binary, addition needs to take in account a carry in/carry out(Ci/o), the first operand, and the second operand. The (Ci/o) is necessary when you are doing addition and it results in requiring another bit, for example 9 + 1 in decimal results in a Co of 1 and a sum of 0 in current index, resulting in 10 when adding the Co. So same as in decimal, in binary when you add 1 + 1 it requires a Co of 1 and a sum of 0 in that current index (Fig. 8 and Fig. 9)

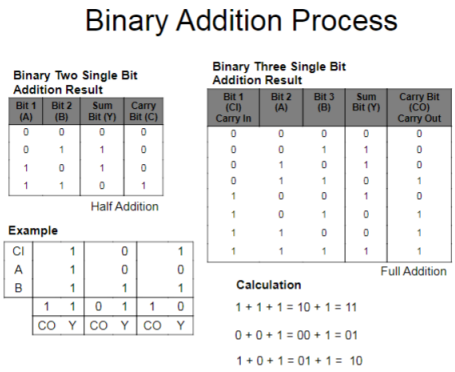

Fig 8 The Binary Addition Truth Table

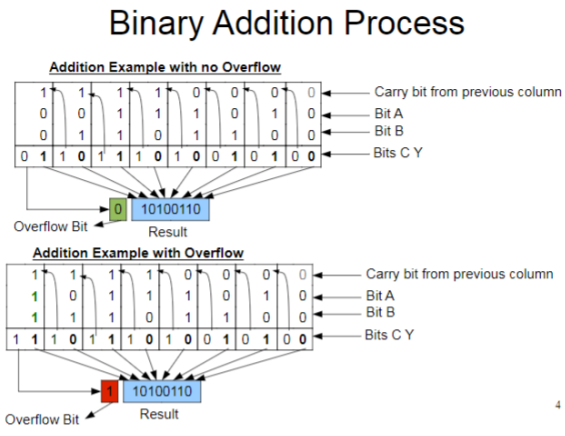

Fig 9 The Binary Addition Process

From this truth table we can create SOP forms of the Summation(Y) and the Co, which in this case will become Y = Em(1,2,4,7) and Co = Em(3,5,6,7) (Fig. 10)

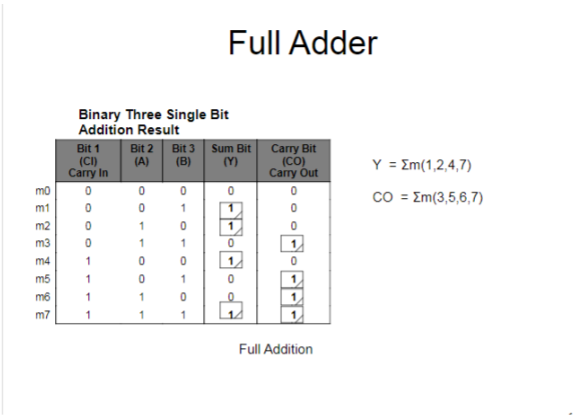

Fig 10 SOP of Y and CO

With these two SOP sets, we can achieve Boolean Logic by first KMAP reducing each set and then reducing it to as small of an expression as possible (shown in Fig. 11)

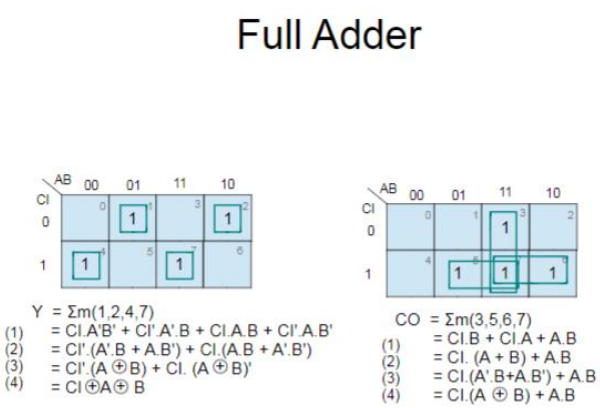

Fig 11 KMAP Reduction of Y and CO Boolean Algebra

Now we can see that both Y and CO can be explained in both programming and electronic circuitry by CI XOR A XOR B for Y and CI AND (A XOR B) OR A AND B for Co (Fig 12), this results in what is called a Full Adder schematic for binary addition.

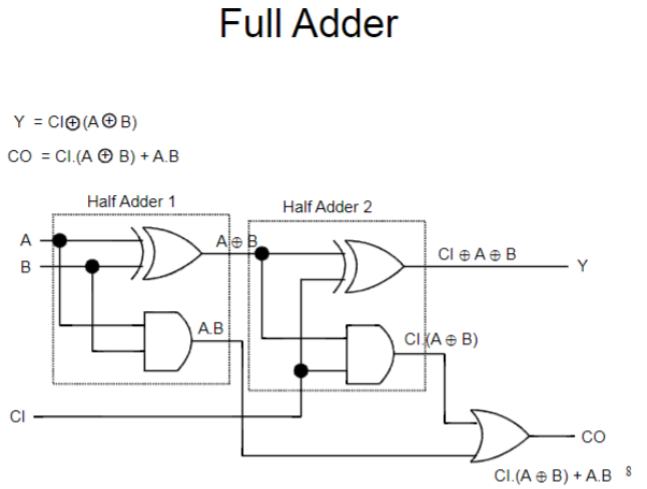

Fig 12 Full Adder Schematic

This circuit and logic is used for one single bit addition, for an Integer, we will need to chain 32 bits worth of Full Adders together and use an initial Ci of 0 (shown in Fig. 13).

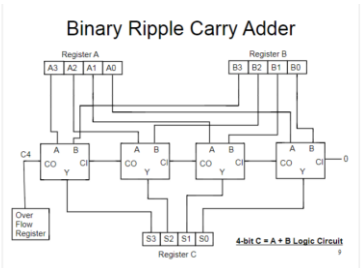

Fig 13 Binary Ripple Carry Adder used to add x > 1 bits

For addition we will need three helper macros which will quicken the process and prevent errors.

1) Extract_nth_bit($regD, $regS, $regT) | Extract_nth_bit_d($regD, $regS, $regT)

One helpful macro will be one that returns the value at a certain index in a bit pattern($regS). To do this we have to create a Mask ($regD) which will store 0x1 which will be shifted to the index($regT) which is to be looked at. After, this mask then is AND with the bit pattern and the result will returned to mask resulting in at the index a 0 or 1 which represents in the value at the index in the bit pattern. To return we shift the mask to the right index times. Extract_nth_bit_d does the same but instead of a register, the macro accepts a constant for the $regT position allowing to skip the step of setting a register to a certain value first (Fig. 14).

```
.macro extract_nth_bit($regD,$regS,$regT)
li $regD, 0x1 #Extract
sllv $regD, $regD,$regT
and $regD, $regS,$regD
srlv $regD, $regD,$regT
.end_macro
.macro extract_nth_bit_d($regD,$regS,$regT
li $regD, 0x1 #Extract
sll $regD, $regD,$regT
and $regD, $regS,$regD
srl $regD, $regD,$regT
.end_macro
```

Fig 14 Implementation of extract_nth_bit and extract_nth_bit_d

2) Insert_to_nth_bit($regD, $regS, $regT, $maskReg)

3)

The last helpful macro will be one that allows you to insert a 0 or 1 into a bit pattern at one of its 32 indexes range from (0, 31). This can be achieved by setting $maskReg (mask) as 0x1 and then shifting it to the left by the position ($regS). Invert the mask after and mask it with $regD (the bit pattern), then shift the value ($regT) to the left by the same index position and OR the bit pattern with the value to be inserted (Fig. 15).

```
.macro insert_to_nth_bit($regD,$regS,$regT,$m
li $maskReg, 0x1 #INSERT
sllv $maskReg, $maskReg,$regS
not $maskReg,$maskReg
and $regD,$regD,$maskReg
sllv $regT, $regT, $regS
or $regD,$regT,$regD
.end_macro
```

Fig 15 Implementation of insert_to_nth_bit

It is good to note that with the addition operation to achieve subtraction it is equivalent to inverting the subtractor and adding 1 which will give us (-1 * subtractor). In other words, any subtraction expression can be represented as a positive with a negative right-hand operand, or (3 + 3 = 3 – (-3)). With this logic the only thing we need to different in subtraction is invert the right-hand operand and do addition once again.

On a circuit level this is done with a signal which is then XOR with the right-hand operand in the Binary Ripple Carry. In MARS we simulate this by changing the operation code from '+' or '-' to a 0x00000000 or 0xFFFFFFFF as shown in order. This is done through a jump to an add_logical or a sub_logical which are labels that need to be created. Both of these labels will then jump to another label that needs to be created which is called add_sub_logical which will change the right-hand operand either into the inverse or not depending on if the first bit of the operation code value is 1 or 0, were 1 would inverse and 0 would not (Fig. 16 and Fig 17).

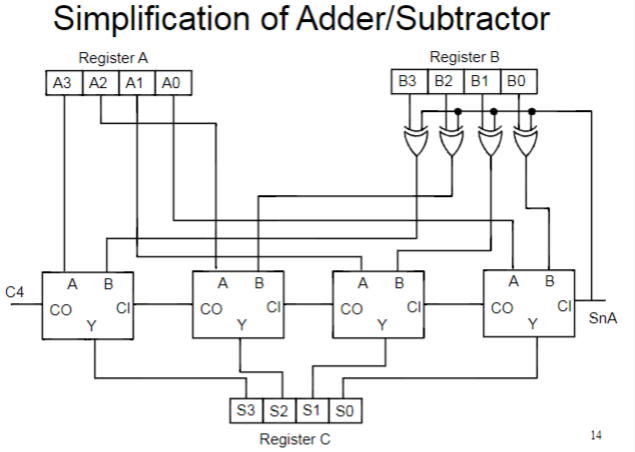

Fig 16 Circuit of Addition and Subtraction

```
add_logical:
li $a2, 0x0
j add_sub_logical

sub_logical:
li $a2, 0xFFFFFFFF

add_sub_logical:
#t0 - COUNTER
#t1 - A
#t2 - B
#t8 - CARRY IN CARRY OUT
#t9 - SUM
li $t0, 0
li $v0, 0
li $t9, 0
extract_nth_bit_d($t8,$a2,0) # SIGNAL
beqz $t8, add_operation
sub_inverse:
not $a1,$a1

add_operation:
```

Fig 17 Implementation of add_logical, sub_logical, and jump to add_sub_logical

By following the Reduced KMAP algebraic expressions for both the sets, and the model of the binary ripple carrier, we can generate this flowchart and implement it in MARS as so (Fig. 18 and Fig. 19).

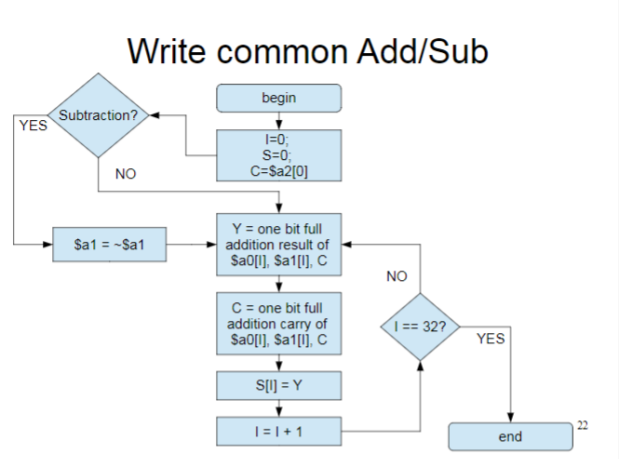

Fig 18 Add Sub Logical Operation Flow Chart

```
add_operation:
#t3 A XOR B
#t4 A AND B
#t5, TEMP MASK
#t6 CI AND (A XOR B)
#t8 CI AND (A XOR B) + (A AND B) << COUNTER
#t9 CI XOR (A XOR B) << SUM
extract_nth_bit($t1,$a0,$t0) # A
extract_nth_bit($t2,$a1,$t0) # B
xor $t3, $t1,$t2 # A XOR B
and $t4, $t1,$t2 # A AND B
and $t6, $t8, $t3 # CI AND (A XOR B)
xor $t9, $t3, $t8 # CI XOR (A XOR B) << SUM
or $t8, $t6, $t4 # CI AND (A XOR B) + (A AND B) << COUT

insert_to_nth_bit($v0,$t0,$t9,$t5)
add $t0,$t0,1
blt $t0,32,add_operation
move $v1,$t8 #LAST Co in $v1
add_mode_end:
j au_logical_end
```

Fig 19 Implementation of add_sub_logic

C) Multiplication Operation

Congratulations! We achieved the power of addition. Note, that this result will cause the size to be double the amount which is the purpose of having a HI and LO register in MIPS protocol. We will first have to create a few helpful procedures, create unsigned multiplication and implement signed multiplication. The reason why we do unsigned multiplication first is because it is simple to check if the result of a signed multiplication should be negative or positive by XOR of the MSB in both the operands. A multiplication operation can be processed as shown in Fig.

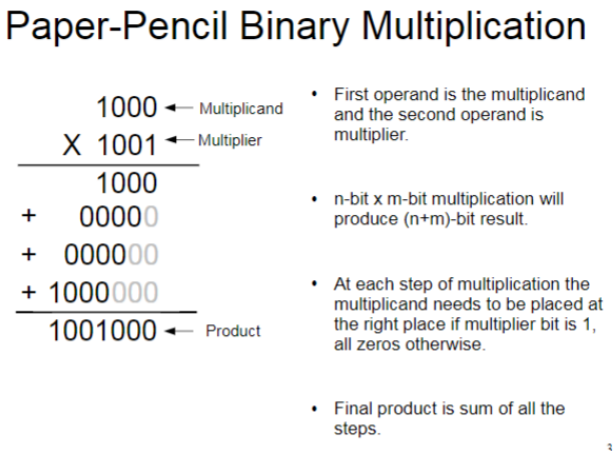

Fig 20 Multiplication by hand for Binary Multiplication
With this concept we can generate the flowchart necessary for Binary Multiplication Algorithm as shown in Fig 21.

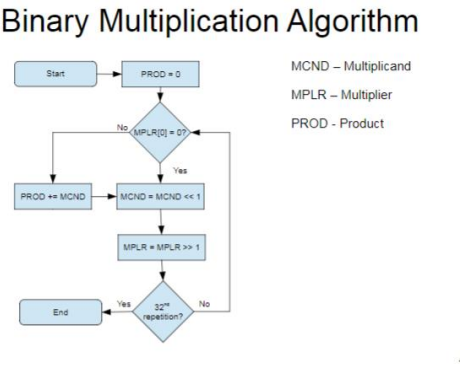

Fig 21 Flowchart for Binary Multiplication

Which when is integrated into a Circuit will look like that as of Fig 22. Some things to note is that while the Multiplicand and Multiplier are 2 32-bit numbers, the Product and Multiplier are joined together as two registers resulting in 1 64-bit register. This is done because the product will grow by 1 bit every iteration and by the 31st iteration the product will have consumed the whole register.

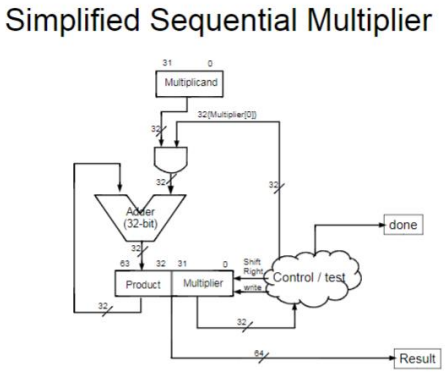

Fig 22 Unsigned Multiplier Circuit

With this we have enough knowledge to implement the Multiplication Operation into MARS in the MIPS protocol. We will begin by creating 4 helpful procedures.

1) twos_complement

This procedure will inverse a number from positive to negative or negative to positive. This can be achieved by first NOT the number and then adding 1 with the add_logical section of au_logic.

```
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 23 Implementation of twos_complement

2) twos_complement_if_neg.

This procedure will inverse a number from negative to positive and only negative to positive. This can be achieved by checking if the number is negative, and if it is then calling twos_complement.

```
#########################################################
# Implement twos_complement_if_neg
# Argument:
#   $a0: Number of which 2's complement to be computed
# Return:
#   $v0: Two's complement of $a0 of $a0 is negative
# Notes:
#########################################################
twos_complement_if_neg:
    addi    $sp, $sp, -20
    sw  $fp, 20($sp)
    sw  $ra, 16($sp)
    sw  $a0, 12($sp)
    sw  $a1, 8($sp)
    addi    $fp, $sp, 20

    bgez $a0,positive

    jal twos_complement
    j twos_complement_if_neg_end
    positive:
    move $v0,$a0
    twos_complement_if_neg_end:
    lw  $fp, 20($sp)
    lw  $ra, 16($sp)
    lw  $a0, 12($sp)
    lw  $a1, 8($sp)
    addi    $sp, $sp, 20
jr $ra
```

Fig 24 Implementation of twos_complement_if_neg

3) twos_complement_64bit

This procedure will inverse a 64-bit number, which is two other registers. This is achieved by first inverse the LO register and adding 1, after taking the Co we inverse the HI register and adding the Co.

```
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 #keep hold of arg 2
    not $a0,$a0 #inverse arg 1
    li $a2, '+' #set operation command to add
    li $a1, 1 # change a1 to 1 to add 1
    jal au_logical #~arg1 + 1 = $v0
    move $a1,$v1 #move the carry out of ~arg1 + 1 to $a1
    move $a0,$s1 #copy the value of arg2 to $a0
    not $a0,$a0 #inverse arg 2
    move $s1,$v0 #copy the ~arg1+1 to $s1
    jal au_logical # ~arg2 + carryout
    move $v1,$v0 #move addition of ~arg2 + carryout to $v1
    move $v0,$s1 #move ~arg1+1 to $v0
    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 25 Implementation of twos_complement_64bit

4) bit_replicator

This procedure will accept a 0x1 or 0x0 valued bit pattern and repeat the 1 or 0 causing a result of 0x00000000 or 0Xffffffff.

```
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

        bit_replicator_one:
        li $v0, 0xFFFFFFFF
        j bit_replicator_end

        bit_replicator_zero:
        li $v0, 0x00000000
        bit_replicator_end:

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

Fig 26 bit_replicator

With the circuit we can generate a flow chart which is compatible with MIPS (Fig. 27)

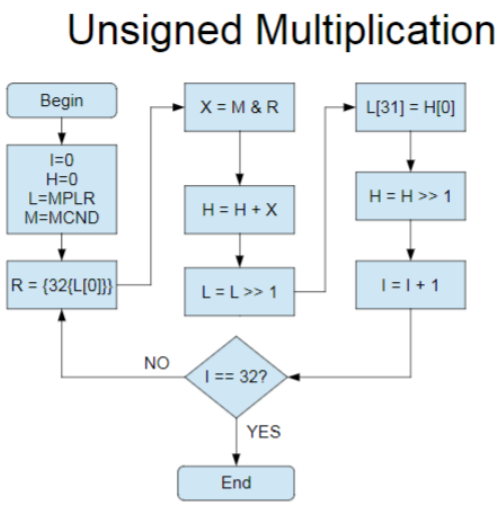

Fig 28 Unsigned Multiplication Flowchart

This unsigned multiplication works by first having the upper half (in theory but another register in MIPS) of the multiplier register equal 0 and then AND the multiplier with the LSB of the multiplier which needs to be repeated ® and becomes X = M & R. with this it is added to the upper half of the multiplier register. LO (multiplier register) is shifted to the right and the MSB of LO will equal the LSB of HI. HI is then shifted which erases the LSB. The counter is incremented and repeated another 31 times. The result will return a 64 bit register which is HI and LO or $v0, and $v1 which has been agreed upon as the outputs before (Fig. 29).

```
mul_unsigned:
    addi    $sp, $sp, -32
    sw  $fp, 32($sp)
    sw  $s2, 28($sp)
    sw  $ra, 24($sp)
    sw  $a0, 20($sp)
    sw  $a1, 16($sp)
    sw  $s0, 12($sp)
    sw  $s1, 8($sp)
    addi    $fp, $sp, 32
    li $s0, 0 #counter i = 0
    li $v1, 0 #H = 0
    move $v0, $a1 #move multiplier into lo register
    mul_loop:
    extract_nth_bit($t1,$v0,$zero)
    move $s1, $v0
    move $s2, $a0
    move $a0,$t1
    jal bit_replicator # R = {32{L[0]}}
    move $t1,$v0
    move $v0,$s1
    move $a0,$s2
    and $t2,$t1,$a0 #index of lsb with $a0 X = M & R
    move $s1, $v0 #stores lo
    move $s2, $a0 #stores multiplicand
    move $a0, $v1 # argument for au logical - produc
    move $a1, $t2 # argument for au logical - 0 or 1
    li $a2, '+'
    jal au_logical # H = H+X
    li $a2, '+'
    jal au_logical # H = H+X
    move $v1, $v0 # moving product back to product r
    move $v0,$s1  # moving lo back to lo register
    move $a0,$s2  # moving back multiplicand
    srl $v0,$v0,1 #L = L >> 1
    li $t5, 31
    extract_nth_bit_d($t6,$v1,0) #H[0]
    insert_to_nth_bit($v0,$t5,$t6,$t7)#L[31] = H[0]
    srl $v1,$v1,1 # H = H >> 1
    addi $s0, $s0,1 #I = I +1
    blt $s0, 32, mul_loop
    mul_loop_end:
    lw  $fp, 32($sp)
    lw  $s2, 28($sp)
    lw  $ra, 24($sp)
    lw  $a0, 20($sp)
    lw  $a1, 16($sp)
    lw  $s0, 12($sp)
    lw  $s1, 8($sp)
    addi    $sp, $sp, 32
jr $ra
```

Fig 29 Implementation of Unsigned Multiplication

As mentioned before to figure out if the result should become a negative or a positive we use XOR the MSB of both the original arguments. If the XOR returns a 1, then we will make the HI and LO become twos complement or we keep the result the same and return as is. We do the twos complement with the twos_complement_64bit procedure (Fig. 30)

```
mul_signed:

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

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

    jal twos_complement_if_neg
    move $s0, $v0
    move $a0, $a1

    jal twos_complement_if_neg
    move $a1,$v0
    move $a0,$s0

    jal mul_unsigned

    extract_nth_bit_d($t0,$s1,31)
    extract_nth_bit_d($t1,$s2,31)
    move $a0,$v0
     move $a1,$v1
     xor $t0,$t0,$t1
     beqz $t0,mul_signed_restore
     jal twos_complement_64bit
     mul_signed_restore:


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

Fig 30 Implementation of Signed Multiplication

Division Operation

With completion of the Addition, Subtraction and Multiplication operation. The theory for using Division is similar but instead of Addition and shifting to the right, we will now use Subtraction and shift to the left. The process is straightforward as simple division with decimals (Fig. 31)

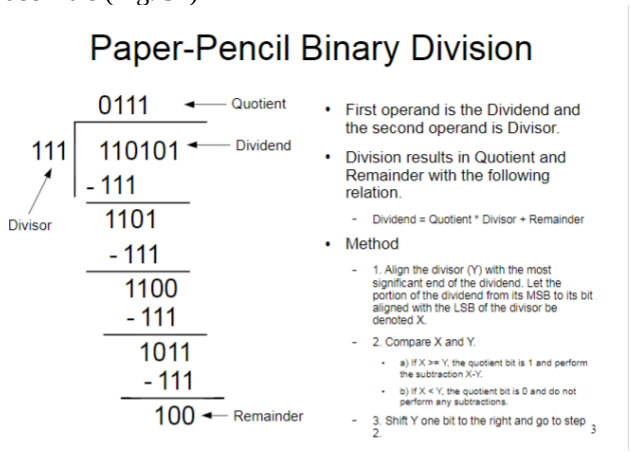

Fig 31 Binary Division done by hand

From figure 31 we can derive a binary division algorithm as shown in figure 32.

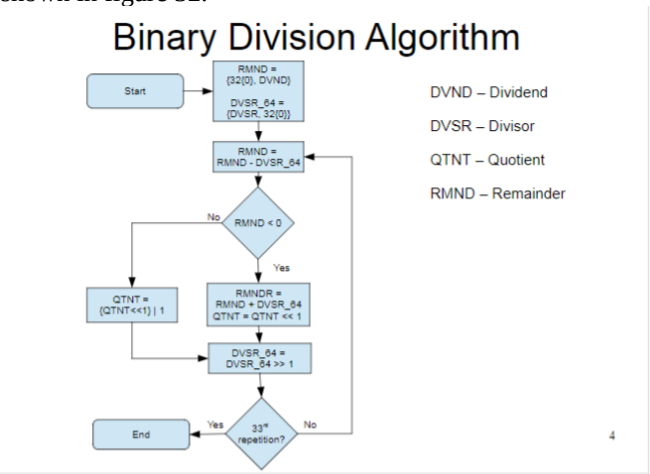

Fig 32 Binary Division Algorithm

We will need one 32 bits register and a 64-bit register (2 32 registers) which will equal the Quotient and the remainder. The algorithm tries to find the biggest number it is able to divide and then repeats it 32 times which will result in a output of a Quotient and a Remainder.

This algorithm can be used to generate the circuit (Fig. 33)

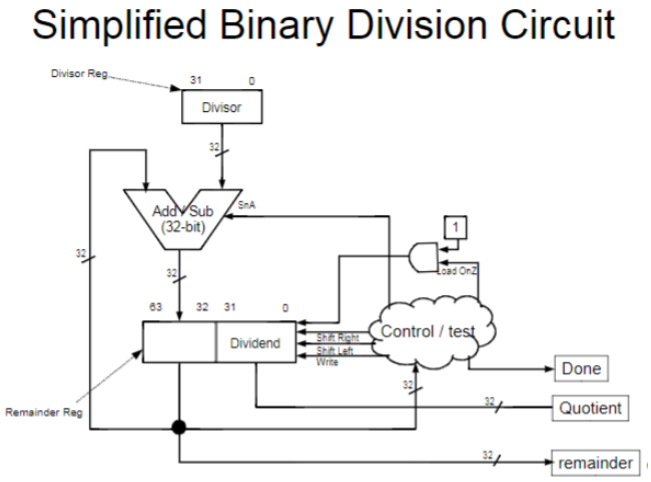

Fig 33 Binary Division Circuit

With this circuit and the Binary Division Algorithm a possible algorithm is designed to do Unsigned Division which is compatible with MIPS (Fig. 34)

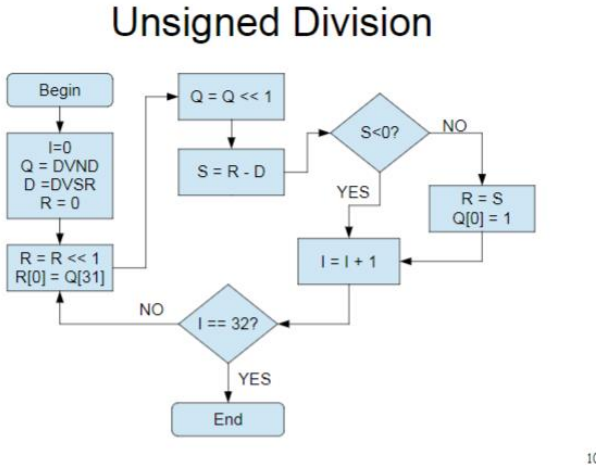

Fig 34 Unsigned Division Algorithm

This algorithm works by shifting the remainder to the left and then pulling the MSB from the Dividend. From there the Dividend is shifted to the left to erase the MSB. We then create a variable that holds the result of the Remainder subtracted by the Divisor and checks if it is less than 0. If it is not less than zero, as in it's a positive number, the result will be placed in the Remainder Register and the Quotients LSB will equal 1. From there the counter is updated and is repeated for another 31 iterations. Note the same theory of what we did with Multiplication with the unsigned to signed logic can be repeated with the unsigned division to signed division.

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

Fig 35. Implementation of Unsigned Division

As mentioned with the same logic as before we can create the code necessary for Signed Division (Fig. 36)

```
div_signed:
    addi    $sp, $sp, -40
    sw  $fp, 40($sp)
    sw  $s4, 36($sp)
    sw  $s3, 32($sp)
    sw  $s2, 28($sp)
    sw  $ra, 24($sp)
    sw  $a0, 20($sp)
    sw  $a1, 16($sp)
    sw  $s0, 12($sp)
    sw  $s1, 8($sp)
    addi    $fp, $sp, 40
    move $s0,$a0
    move $s1,$a1
    jal twos_complement_if_neg  # Make N1 Positive
    move $s2,$v0 #N1 is in T0
    move $a0,$a1
    jal twos_complement_if_neg # Make N2 Positive
    move $a1,$v0
    move $a0,$s2
    jal div_unsigned #DIV OF N1 N2 return to V0 V1
    move $a0, $v0 #Q
    move $s3, $v0 #Q
    move $a1, $v1 #R
    move $s4,$v1 #R
    extract_nth_bit_d($t0,$s0,31) # Extract A0[31]
    extract_nth_bit_d($t1,$s1,31) # Extract A1[31]
    xor $t2,$t1,$t0 # A1[31] XOR A0[31]
    xor $t2,$t1,$t0 # A1[31] XOR A0[31]
    move $s2,$t0 #A0[31]
    beqz $t2,tc_q_end #If S is 1 two complement Q
    move $a0,$s3
    jal twos_complement
    move $s3, $v0
    tc_q_end:
    beqz $s2,tc_r_end #If S is 1 two complement Q
    move $a0,$s4
    jal twos_complement
    move $s4, $v0
    tc_r_end:
    move $v0,$s3
    move $v1,$s4
    lw  $fp, 40($sp)
    lw  $s4, 36($sp)
    lw  $s3, 32($sp)
    lw  $s2, 28($sp)
    lw  $ra, 24($sp)
    lw  $a0, 20($sp)
    lw  $a1, 16($sp)
    lw  $s0, 12($sp)
    lw  $s1, 8($sp)
    addi    $sp, $sp, 40
jr $ra
```

Fig 36. Implementation of Signed Division

VI. TESTING

And with the completion of Addition, Subtraction, Multiplication, Division we have successfully achieved the second project objective. Now what is left is to test the Project. Switch to the “proj-auto-test.asm” file and compile and run (Fig. 37).

```
(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 ***
```

Fig 37. Result of the Tester

A variety of test cases are streamed into the au_normal and au_logic to see if the code returns the same result. If they return the same result, then a “Total passed 40 / 40 *** OVERALL RESULT PASS ***” is returned in the console. As we know that the implementation of the au_normal is correct we can also claim that the results of the tester is also correct, therefore the implementations of the au_logic is also correct!

VII. Conclusion

Overall the modules implemented by the Arithmetic Standard Instruction Set and only logical operations gave me a better understanding of how: MIPS, Assembly, Stack Frames (storing, loading), Electrical Engineering (Boolean Algebra, KMAPS, REDUCTION), ALU are used not only for Arithmetic Operations but also in general. This project reinforced almost all of the theories we learned in class and gave reinforced my base knowledge in the field of Low Level Computer Science.

References

[1] K. Patra. CS 47. Class Lecture, Topic: "Addition Subtraction Logic." San Jose State University, San Jose, CA, April 14, 2017.
[2] K. Patra. CS 47. Class Lecture, Topic: "Multiplication Logic." San Jose State University, San Jose, CA, April 19, 2017.
[3] K. Patra. CS 47. Class Lecture, Topic: "Division Logic." San Jose State University, San Jose, CA, April 21, 2017
[4] Chapter 3 of 'Computer Organization & Design by Hennesy, Patterson
[5] Chapter 4 of 'Logic and Computer Design Fundamentals' by Mano, Kime