Arithmetic Operations Implemented by MIPS Digital Logic

Benjamin Garcia
San Jose State University
benjamin.garcia@sjsu.edu

***Abstract* – This report focuses on explaining how a basic calculator can be implemented through the use of: 1) normal math operations from standard MIPS instructions and 2) logic operations in MIPS assembly language to perform mathematical operations (addition, subtraction, multiplication, and division).**

I. INTRODUCTION

By learning how to use logical operators to calculate expressions, a foundation is laid down for digital circuits. More advanced hardware and software revolves around the basic mathematical operations implemented by the ALU. In doing this project, we design an ALU with the use of standard MIPS instructions as well as by using only logical procedures to create a simple calculator that can perform all the necessary arithmetic operations.

The calculator is written in MIPS assembly code and tested using MARS (MIPS Assembler and Runtime Simulator) which can be downloaded at:

II. REQUIREMENTS

A. *Project Setup*

Download MARS (MIPS Simulator) at: http://courses.missouristate.edu/KenVollmar/mars/ and the provided project file "CS47Project1.zip" in SJSU Canvas at: https://sjsu.instructure.com/courses/1311483/files/52626319/download?wrap=1

Unzip the file and it will include the following files:

1) cs47_common_macro.asm
2) cs47_proj_alu_logical.asm
3) cs47_proj_alu_normal.asm
4) cs47_proj_macro.asm
5) cs47_proj_procs.asm
6) proj-auto-test.asm

Open MARS and press "File" then "Open" and navigate to where your unzipped folder is stored. Now open the three files that we will be working with (you will have to repeat this step for each file as MARS does not allow you to select more than one file at a time):

1) cs47_proj_alu_logical.asm
   This file will hold the implementation of a basic calculator using only logical operations.
2) cs47_proj_alu_normal.asm
   This file will hold the implementation of a calculator using MIPS basic instruction set.
3) cs47_proj_macro.asm
   This file contains all project specific macros used and implemented.

On the MARS settings, make sure to turn on 'Assembles all files in directory' and 'Initialize program counter to global main if defined' option.

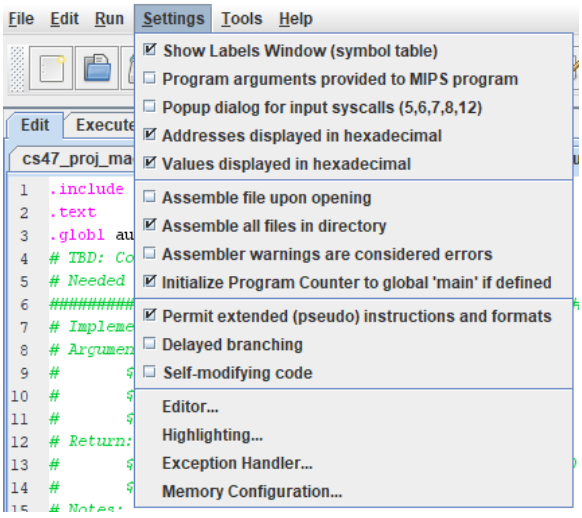

Figure 1: MARS Settings Menu

B. *Boolean Algebra*

A Boolean variable assumes two values, TRUE (denoted by 1) or FALSE (denoted by 0). With these variables, there are basic Boolean algebraic operations which are AND (or conjunction denoted by a '.'), OR (or disjunction denoted by a '+'), and NOT (or negation denoted by a ''). The truth tables for each operation is below.

Table 1: Truth Table for Logical AND

A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

Note: The AND operation will always return 0 unless all inputs are 1.

Table 2: Truth Table for Logical OR

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

Note: The OR operation will always return 1 unless all inputs are 0.

Table 3: Truth Table for Logical NOT

A	A'
0	1
1	0

Although these are the basic Boolean algebraic operations, there are different operations when moving to logic gates. Logic gates are physical implementations of fundamental logic operations and the fundamental logic gates are NAND (not and), NOR (not OR), and NOT. The reason for this change is because at the hardware level, an extra transistor is needed to make a logical AND/OR, therefore the fundamental logic gate is NAND/NOR and not AND/OR.

Even though these are the fundamental logic gates, there is one very important gate in digital logic and circuit design which is an XOR gate (denoted by $\oplus$). Its truth table is shown and is used a lot when implementing the calculator in MIPS assembly.

Table 4: Truth Table for XOR Logic Gate

A	B	A$\oplus$B
0	0	0
0	1	1
1	0	1
1	1	1

Note: The XOR gate returns 0 if its inputs match and returns 1 if they differ. This is a KEY POINT to realize when implementing the calculator.

C. Binary Knowledge

Binary is a number system with base 2 unlike our traditional counting system, the decimal system, which uses base 10. Base 2 means that it consists of only 2 symbols, 0 and 1. In the base 10 number system, a number can contain digits ranging from 0 to 9 and when that position maxes out at 9, it goes back to 0 and the next higher position increases. For example, the next digit after 9 will "roll over" to become 10 and 99 will roll over to form 100. However, in the binary system, once 1 is reached, the next number goes back to 0 and the 1 rolls over to the next position. For example, the next few digits after 1 will be 10, 11, 100, 101.

D. Negative Representation in Binary

When using our traditional number system, a negative number can easily be represented with a "-" symbol in front of the number. However, in binary, we are limited to the symbols 0 and 1. So, a system had to be put in place called "two's complement". Essentially what it does is reserve the left most bit of a number to represent the "sign" of the number. A number 1 in the most significant bit (left most bit) of a number represents it is a negative number and a 0 represents a positive number. For example, 3 in two's complement form is 0011 but -3 is 1101. The positive of a number is simply it's normal binary value but to get the negative number, you invert all the bits of it's corresponding positive number and add 1. When flipping the number 3, we get 1100 and then add one to get 1101 (-3).

The two's complement system takes the value of the MSB, sets that to be negative and then add the value after the initial 1. For example, with the number 1101, it has a 1 so it means it is negative. The highest bit value (1000 = 2^3) is 8 and the value after the initial 1 is 101 which is 5 so the value of 1101 is -8 + 5 = -3.

III. Design and Implementation

To complete this project, there are two different types of procedures that must be implemented: the normal procedure using MIPS' provided basic instructions and the logical way using logic gates. For each procedure, addition, subtraction, multiplication, and division must be implemented.

A. Normal Procedure

The normal procedure is to be completed in the "cs47_proj_alu_normal.asm" file and is named au_normal. The normal procedure calculates mathematical expressions by using MIPS mathematical operations provided in the basic instructions. This procedure takes in three different arguments:

1) Register $a0
   This argument is the first operand in the mathematical expression.
2) Register $a1
   This argument is the second operand in the mathematic expression.
3) Register $a2
   This argument is the operator provided in ASCII code that determines whether to do an addition, subtraction, multiplication, or division operation on the two operands.

This procedure also returns two outputs:

1) Register $v0
   This register will contain the result of an addition or subtraction operation. It will also hold the "LO" result of a multiplication operation since the result is a 64-bit answer. Or it can hold the quotient of a division operation.
2) Register $v1
   This register will contain the "HI" result of a multiplication operation or the remainder of a division operation.

The normal procedure relies on MIPS basic instruction which provides "add", "sub", "mult", and "div". By using branches to jump to the right operation, we can compute the result using those provided instructions and store them in the proper register.

```
au_normal:
        beq $a2, '+', add_op
        beq $a2, '-', sub_op
        beq $a2, '*', mul_op
        beq $a2, '/', div_op
```

Figure 2: Normal Procedure Branches

This shows the "branch equals" instruction checking register $a2 which contains the operator and branching to the proper level to do the mathematical procedure.

1) Addition Implementation

```
add_op:
        stack_save              # call stack_save macro
        add $v0, $a0, $a1
        stack_restore           # call stack_restore macro
```

Figure 3: Addition Normal Procedure

The addition in the normal procedure is implemented by first saving the stack frame using a macro which will be discussed later, using the "add" instruction to add the values in $a0 and $a1 then storing the result in $v0. Lastly, the stack frame is restored using a macro being discussed later.

2) Subtraction Implementation

```
sub_op:
        stack_save              # call stack_save macro
        sub $v0, $a0, $a1
        stack_restore           # call stack_restore macro
```

Figure 4: Subtraction Normal Procedure

The subtraction in the normal procedure is implemented by first saving the stack frame using a macro which will be discussed later, using the "sub" instruction to subtract the values in $a0 and

$a1 then storing the result in $v0. Lastly, the stack frame is restored using a macro being discussed later.

3) Multiplication Implementation

```
mul_op:
        stack_save              # call stack_save macro
        mult $a0, $a1
        mflo $v0
        mfhi $v1
        stack_restore           # call stack_restore macro
```

Figure 5: Multiplication Normal Procedure

The multiplication in the normal procedure is implemented by first saving the stack frame using a macro which will be discussed later, using the "mult" instruction which multiplies the values in $a0 and $a1 and stores the lower half of the 64-bit in the lo register and the upper half in the hi register. The "mflo" instruction is then used to move the lower half of the result into $v0 and the "mfhi" instruction is used to move the upper half of the result into $v1. Lastly, the stack frame is restored using a macro being discussed later.

4) Division Implementation

```
div_op:
        stack_save              # call stack_save macro
        div $a0, $a1
        mflo $v0
        mfhi $v1
        stack_restore           # call stack_restore macro
```

Figure 6: Division Normal Procedure

The division in the normal procedure is implemented by first saving the stack frame using a macro which will be discussed later, using the "div" instruction which divides the values in $a0 and $a1 and stores the quotient in the lo register and the remainder in the hi register. The "mflo" instruction is then used to move the quotient into $v0 and the "mfhi" instruction is used to move the remainder into $v1. Lastly, the stack frame is restored using a macro being discussed later.

B. Project Macros

Code that is commonly used should be stored as a macro to prevent copy and pasting repetitive lines of code which allows for more areas for error. There is a provided file to add macros for use in out project called "cs47_proj_macro.asm". The macros I created were mainly to help with the logical implementation but I did create stack saving and restoring macros since every procedure must do those two operations.

1) stack_save

```
.macro stack_save
        addi    $sp, $sp, -60
        sw      $fp, 60($sp)
        sw      $ra, 56($sp)
        sw      $a0, 52($sp)
        sw      $a1, 48($sp)
        sw      $a2, 44($sp)
        sw      $a3, 40($sp)
        sw      $s0, 36($sp)
        sw      $s1, 32($sp)
        sw      $s2, 28($sp)
        sw      $s3, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        sw      $s7, 8($sp)
        addi    $fp, $sp, 60
.end_macro
```

Figure 7: stack_save Macro

This macro moves the stack pointer down, saves the frame pointer ($fp), the return address ($ra), all of the argument registers ($a0 - $a3), all of the save registers ($s0 - $s7), and then moves the frame pointer to the stack pointer's original position.

2) stack_restore

```
.macro stack_restore
        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
.end_macro
```

Figure 7: stack_restore Macro

This macro loads the frame pointer ($fp), the return address ($ra), all of the argument registers ($a0 - $a3), all of the save registers ($s0 - $s7), moves the stack pointer to the same position as the frame pointer, and then jumps back to the caller using jump return to the return address. It essentially reverts the stack_save macro.

3) extract_nth_bit

```
.macro extract_nth_bit($regD, $regS, $regT)
        # $regD : will contain 0x0 or 0x1 de
        # $regS: Source bit pattern
        # $regT: Bit position n (0-31)

        la $t0, ($regS)          # load bit p
        srav $t0, $t0, $regT     # right shif
        andi $regD, $t0, 1       # set $regD
.end_macro
```

Figure 8: extract_nth_bit Macro

This macro takes in 3 arguments: $regD (a register to return what the bit is at a certain position), $regS (a register holding the bit pattern to be extracted from), and $regT (a register holding the position number we want to extract).

The macro loads the bit pattern into a temporary register ($t0) so the original source register is not messed up, right shifts it by the numerical value of the bit position that we are trying to extract the bit from and stored back in $t0, and then performs a logical AND operation on that right shifted pattern and the number 1 to get the value of the bit at the position stored in $regT. This extracted value then gets placed into whatever register is in the $regD argument when the macro is called. This works because after the right shifts, the right most bit is now the bit we want to extract. We then AND it with 1 and if that right most bit is 0, the result of the AND operation will be 0; if it is 1, then the result of the AND operation between that right most bit and 1 will be 1. This properly extracts the bit we want.

4) insert_to_nth_bit

```
.macro insert_to_nth_bit($regD, $regS, $regT, $maskReg)
        # $regD : This the bit pattern in which 1 to be
        # $regS: Value n, from which position the bit to
        # $regT: Register that contains 0x1 or 0x0 (bit
        # $maskReg: Register to hold temporary mask

        li $maskReg, 1                  # initia
        sllv $maskReg, $maskReg, $regS  # left s
        not $maskReg, $maskReg          # invert
        and $regD, $regD, $maskReg      # set $r
        sllv $regT, $regT, $regS
        or $regD, $regD, $regT          # set $r
.end_macro
```

Figure 9: insert_to_nth_bit Macro

This macro takes in 4 arguments: $regD (contains the bit pattern in which the proper bit will be inserted into), $regS (contains the insertion position), regT (contains what bit will be inserted, either 0x1 or 0x0), and $maskReg (any temporary register in which a mask will be created that is necessary for accomplishing this task).

The mask is initialized at 1 and then shifted left by the amount in $regS (the insertion position) and stored back in the temporary mask register. All bits in the mask register are then inverted using the logical NOT operation and stored back in $maskReg. The AND operation is used on the mask register and the bit pattern that we want to insert which forces a 0 into the position in which we are inserting. $regT which contains the bit being inserted is then shifted left by the amount in $regS to get it in the right position. An OR operation is then used on $regD and $regT and the result is stored back in $regD which completes the insertion process.

C. Utility Procedures

Aside from the macros used in the project, there are some procedures that are added to complement the logical procedures.

1) twos_complement_if_neg, twos_complement

```
twos_complement_if_neg:
        stack_save
        bltz $a0, twos_complement_no_resave
        la $v0, ($a0)
        stack_restore
twos_complement:
        stack_save
twos_complement_no_resave:
        not $a0, $a0
        li $al, 1
        jal add_logical
        stack_restore
```

Figure 10: twos_complement_if_neg and twos_complement Utility Procedures

This procedure takes an argument in the $a0 register and returns it's two's complement in the $v0 register.

These are two different procedures referenced in the logical implementation, but the twos_complement_if_neg relies on the normal twos_complement procedure. The twos_complement_if_neg procedure will compare the value in the argument register $a0 and if it is less than 0, jump to the body of the twos_complement procedure and return the two's complement of that value in $v0. If the value is not less than 0, the original value is put in $v0 and returned to the caller. The main twos_complement procedure calculates the two's complement of the value in the $a0 register by flipping all the bits using the NOT operation and then adding 1 by using the jump and link instruction to calculate the result through the add logical procedure.

2) twos_complement_64bit

```
twos_complement_64bit:
        stack_save
        not $a0, $a0
        not $al, $al
        la $a3, ($al)
        li $al, 1
        jal add_logical
        la $s0, ($v0)
        la $a0, ($v1)
        la $al, ($a3)
        jal add_logical
        la $v1, ($v0)
        la $v0, ($s0)
        stack_restore
```

Figure 11: twos_complement_64bit Utility Procedure

This utility procedure returns the complement of the lo and hi registers that result from a multiplication operation. The "LO" is passed in the $a0 register and the "HI" is passed in the $a1 register. The "LO" part of the result of the two's complement procedure is returned in the $v0 register and the "HI" part is returned in the $v1 register.

The procedure works by inverting both the lo and the hi registers and saved back in their respective registers. The new lo register is then added with 1 using the add logical procedure to get the "LO" part of the two's complemented value. The carry bit from that addition procedure is then added to the new inverted hi register to get the "HI" part of the two's complemented value. The new lo and hi results are then stored in $v0 and $v1 respectively.

3) bit_replicator

```
bit_replicator:
        stack_save
        beqz $a0, replicate_zero
        li $v0, 0xFFFFFFFF
        stack_restore
replicate_zero:
        li $v0, 0x00000000
        stack_restore
```

Figure 12: bit_replicator Utility Procedure

This procedure takes whatever bit is in $a0 and replicates it to be a 32 bit value of the bit passed in. The result is loaded into $v0.

The procedure checks if the argument in $a0 is zero and if it is, it will branch to the replicate_zero label where 0x00000000 is loaded into $v0. If it is not 0 (meaning it must be 1), then 0xFFFFFFFF is loaded into $v0.

D. Logical Addition and Subtraction Design and Implementation

Addition and subtraction is combined because their operations are very similar. The subtraction operation is just an addition operation with the second input complemented to form its negative counterpart.

$$A - B = A + (-B)$$

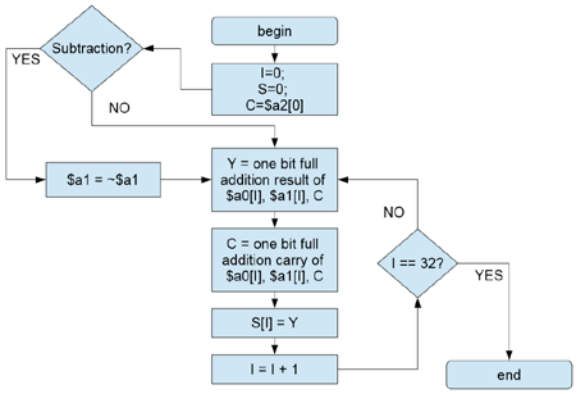

Figure 13: Flowchart for Addition and Subtraction

This flowchart shows how to convert the addition and subtraction implementation into a single piece. This is mainly as a result of the carry in bit being initialized to the MSB of the second argument (0 for addition because it is a positive number, 1 for subtraction because it is a negative number). This is because when doing a subtraction operation, it is essentially an addition operation with the second argument in it's two's complement form. To convert to two's complement as previously stated, the bits are flipped and 1 is added to that. So, to get to subtraction, the bits are flipped and the carry in bit becomes 1. This essentially combines the addition and subtraction procedure.

```
# returns $v0 = final sum and $v1 = carry out

# $s0 = index counter
# $v1 = carry in bit for upcoming addition (i
# $s1 = constant (32) to compare against coun
# $s2 = nth bit from first argument
# $s3 = nth bit from second argument
# $s4 = result of A XOR B (bits for partial s
# $s5 = result of CI XOR A XOR B (result of p
# $s6 = A.B
# $s7 = CI.(A XOR B)
# $t9 = temporary register to use for mask
add_logical:
        stack_save
        li $s0, 0
        li $v1, 0
        li $v0, 0
        jal full_adder_loop
        stack_restore

full_adder_loop:
        li $s1, 32
        beq $s0, $s1, exit_add_loop
        extract_nth_bit($s2, $a0, $s0)
        extract_nth_bit($s3, $a1, $s0)
        xor $s4, $s2, $s3
        xor $s5, $v1, $s4
        and $s6, $s2, $s3
        and $s7, $v1, $s4
        or $v1, $s7, $s6
        insert_to_nth_bit($v0, $s0, $s5, $t9)
        addi $s0, $s0, 1
        j full_adder_loop

exit_add_loop:
        jr $ra
```

Figure 13: Logical Addition Procedure

The addition logic is done by performing operations to determine the digit of a partial summation between one bit of each operand starting with the LSB side, taking the "carry out" of that partial sum, and then adding it with the next bit of each operand being added. This is repeated 32 times to add all 32 bits together.

For addition, the stack is first saved, the counter is initialized to 0, the carry in bit is initialized to 0, and the final sum is set to 0. Then, the "jal" instruction is called to jump and link into

the full_adder_loop which computes 32 partial sums of the two bits of each operand being added and the carry out bit from the previous operation and continuously inserts the results into the $v0 register holding the final sum. At the end of each loop, the counter is increased by 1 and jumped back to the beginning where it is compared against the constant 32 to see if the procedure has been executed 32 times. If it has, it will jump to the exit_add_loop label which contains a jump return to the return address ($ra) where the stack is restored. After 32 times in this loop, the result will be stored in $v0.

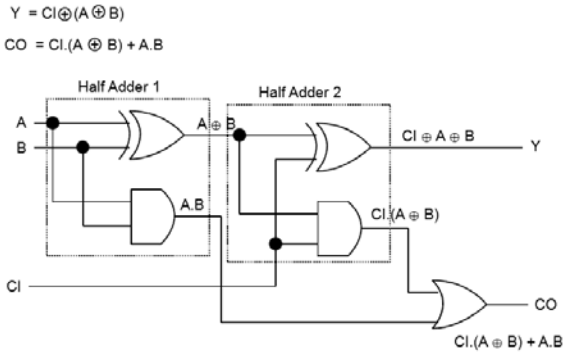

Figure 14: Full Adder Circuit

The full_adder_loop in the logical addition procedure goes off the full adder circuit designed at the hardware level shown above. It uses the extract_nth_bit macro to extract each bit in the position of the counter and XOR's them giving an A⊕B result. That result is then XOR'd with the carry in bit to give the final value represented by CI⊕A⊕B. This final result is then inserted to the $v0 register holding the final result using the insert_to_nth_bit macro.

To calculate the carry out bit which becomes the carry in bit for the next partial addition, the two extracted bits are first AND'd giving A.B. Then, the carry in bit is AND'd with the A⊕B result giving CI.(A⊕B). These two results are then OR'd giving CI.(A⊕B) + A.B which represents the carry out bit of that partial addition.

```
sub_logical:
        stack_save
        la $s0, ($a0)
        la $a0, ($a1)
        jal twos_complement
        la $a0, ($s0)
        la $a1, ($v0)
        jal add_logical
        stack_restore
```

Figure 14: Logical Subtraction Procedure

Implementing subtraction is easy since it is simply based on addition. The bulk of the code lies in the previous addition procedure. However, before we get to it, we must calculate the two's complement of the second operand being subtracted so we can add the two together and get the subtraction result. This is done by calling our two's complement procedure which inverts the bits on our second operand and adds 1 to it (using the add_logical) procedure. Then, the two arguments are passed into the add_logical procedure (the first argument, and the two's complement of the second argument).

E. Logical Multiplication Design and Implementation

The multiplication procedure will take two signed arguments in $a0 and $a1, multiply them and output the "LO" part of the result in $v0 and the "HI" part of the result in $v1.

The unsigned multiplication procedure involves the use of creating a mask from the LSB of the multiplier, using it on the multiplicand, and then adding the result to the final output.

The flow chart below shows how the unsigned multiplication procedure works using logical operations.

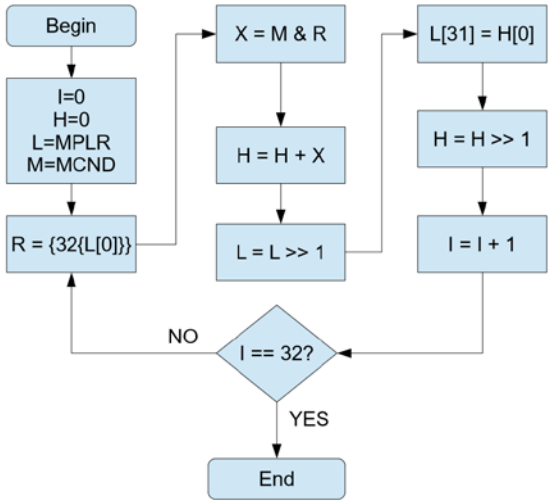

Figure 15: Multiplication Flow Chart

```
mul_logical:
# returns signed $v0 = lo and signed $v1 = hi

# $a0 = multiplicand MCND -> twos complement of
# $a1 = multiplier MPLR -> twos complement of m
        stack_save
        jal twos_complement_if_neg
        la $s0, ($v0)
        la $a0, ($a1)
        jal twos_complement_if_neg
        la $a0, ($s0)
        la $a1, ($v0)
mul_unsigned:
# unsigned multiplication of $a0 (MCND), $a1 (M
# $s0 = counter
# $s1 = unsigned lo
# $s2 = unsigned hi
# $s3 = MCND (save of $a0) to compare against
# $s4 = AND of MCND.1stBitOfMPLR
# $s5 = LSB of hi
# $s6 = CONSTANT (switches between 31 for bit i
# $t6 = XOR between $t7 and $t8 to determine si
# $t7 = MSB from original multiplicand MCND
# $t8 = MSB from original multiplier MPLR
# $t9 = temporary register to use for mask
```

Figure 16: Logical Multiplication Procedure (part 1)

The first part of the multiplication procedure is getting all the operands to be positive by utilizing the utility procedure "twos_complement_if_neg". Once all operands are positive, the arguments are passed into $a0 and $a1 and proceeds into the mul_unsigned procedure.

```
        li $s0, 0
        la $s1, ($a1)
        li $s2, 0
        la $s3, ($a0)
extract_beginning_mplr:
        extract_nth_bit($a0, $s1, $zero)
        jal bit_replicator
        and $s4, $s3, $v0
        la $a0, ($s2)
        la $a1, ($s4)
        jal add_logical
        la $s2, ($v0)
        srl $s1, $s1, 1
        extract_nth_bit($s5, $s2, $zero)
        li $s6, 31
        insert_to_nth_bit($s1, $s6, $s5, $t9)
        srl $s2, $s2, 1
        addi $s0, $s0, 1
        li $s6, 32
        beq $s0, $s6, exit_mult
        j extract_beginning_mplr
exit_mult:
        la $v0, ($s1)
        la $v1, ($s2)
        lw $a0, 52($sp)
        lw $a1, 48($sp)
        li $s6, 31
        extract_nth_bit($t7, $a0, $s6)
        extract_nth_bit($t8, $a1, $s6)
        xor $t6, $t7, $t8
        beqz $t6, return_mult
        la $a0, ($v0)
        la $a1, ($v1)
        jal twos_complement_64bit
return_mult:
        stack_restore
```

Figure 17: Logical Multiplication Procedure (part 2)

In the mul_unsigned procedure, the counter is initialized to 0, the lo is initialized to the multiplier, the hi is initialized to 0, and the multiplicand is duplicated into $s3 for modification.

We then enter the loop labeled extract_beginning_mplr which is named this because the mask is made by extracting the LSB of the multiplier and replicating it using the bit_replicator. The mask and the multiplicand are then AND'd together and added as an operand for an addition procedure. The other operand is the current "HI" value. The result of the addition is

then set as the new “HI” value. The multiplier is then shifted right by 1 and the LSB of the “HI” is extracted and inserted into the MSB of the “LO” value. The “HI” value is then right shifted by 1 and the counter is increased by 1. The loop then compares the counter to the constant 32 to see if it has completed this loop 32 times and jumps back to the beginning if it has not. If it has, it will continue to the exit_mult label where the sign of the lo and hi result are calculated.

The final sign is determined by performing an XOR operation on the MSBs of the multiplicand and the multiplier. If the result of the XOR operation is 0, then the result is positive, and the current values are returned. If the result of the XOR operation is 1, then the result must be negative and the “twos_complement_64bit” utility procedure is called to convert the 64-bit result into it’s two’s complement form and then it is returned.

F. Logical Division Design and Implementation

The division procedure will take two signed arguments in $a0 and $a1, divide them and output the quotient part of the result in $v0 and the remainder part of the result in $v1.

The flow chart below shows how the unsigned division procedure works using logical operations.

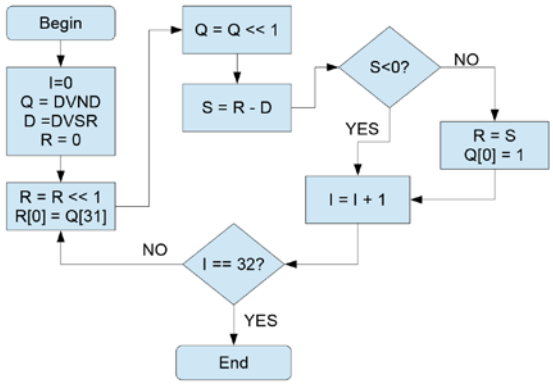

Figure 18: Division Flow Chart

```
div_logical:
# unsigned division of $a0 (dividend), $a1 (d
# returns $v0 = quotient and $v1 = remainder
        stack_save              # call stack_
        jal twos_complement_if_neg
        la $s0, ($v0)
        la $a0, ($a1)
        jal twos_complement_if_neg
        la $a0, ($s0)
        la $a1, ($v0)
div_unsigned:
# $s0 = counter
# $s1 = quotient
# $s2 = divisor
# $s3 = remainder
# $s4 = constant (31 for bit extraction/32 fo
# $s5 = MSB of quotient for insertion into LS
# $s6 = 1 to be inserted into LSB of quotient
# $t6 = XOR between $t7 and $t8 to determine
# $t7 = MSB from original dividend
# $t8 = MSB from original divisor
# $t9 = temporary register to use for mask
```

Figure 19: Logical Division Implementation (part 1)

The first part of the division procedure is getting all the operands to be positive by utilizing the utility procedure “twos_complement_if_neg”. Once all operands are positive, the arguments are passed into $a0 and $a1 and proceeds into the div_unsigned procedure.

```
        li $s0, 0
        la $s1, ($a0)
        la $s2, ($a1)
        li $s3, 0
div_loop:
        li $s4, 31
        sll $s3, $s3, 1
        extract_nth_bit($s5, $s1, $s4)
        insert_to_nth_bit($s3, $zero, $s5, $t9)
        sll $s1, $s1, 1
        la $a0, ($s3)
        la $a1, ($s2)
        jal sub_logical
        bltz $v0, counter_increment
        la $s3, ($v0)
        li $s6, 1
        insert_to_nth_bit($s1, $zero, $s6, $t9)
counter_increment:
        addi $s0, $s0, 1
        li $s4, 32
        beq $s0, $s4, div_exit
        j div_loop
div_exit:
        la $v0, ($s1)
        la $v1, ($s3)
        lw $a0, 52($sp)
        lw $a1, 48($sp)
        li $s4, 31
        extract_nth_bit($t7, $a0, $t9)
        extract_nth_bit($t8, $a1, $t9)
        xor $t6, $t7, $t8
        beqz $t6, remainder_check
        la $a0, ($s1)
        jal twos_complement
        la $s1, ($v0)
remainder_check:
        beqz $t7, skip_second
        la $a0, ($s3)
        jal twos_complement
        la $s3, ($v0)
skip_second:
        la $v0, ($s1)
        la $v1, ($s3)
        stack_restore
```

Figure 20: Logical Division Implementation (part 2)

The div_unsigned procedure first initializes the counter to 0, the quotient to the dividend, saves the divisor for modification, and initializes the remainder to 0 and then proceeds into the loop labeled “div_loop”. First, the remainder is left shifted by 1 and the MSB of the quotient is extracted and inserted into the LSB of the remainder using the extract_nth_bit and insert_to_nth_bit macros. The quotient is then left shift by 1.

The remainder and divisor are subtracted and if the result is less than zero, the procedure will jump to the counter_increment label where the counter is incremented by 1, compared against the constant 32, and exits the loop if it equals to 32, otherwise it jumps to the beginning of div_loop. If the result of the subtraction is greater than or equal to zero, the remainder is set to the result of the subtraction and the bit 1 is inserted into the LSB of the quotient and then proceeds to the counter_increment label.

Once the loop reaches 32 times and jumps to the div_exit label, then the signs are calculated. The below image shows the rules for calculating the signs of a division operation.

- Steps
  - N1 = $a0, N2 = $a1
  - Make N1 two's complement if negative
  - Make N2 two's complement if negative
  - Call unsigned Division using N1, N2. Say the result is Q and R
  - Determine sign S of Q
    - Extract $a0[31] and $a1[31] bits and xor between them. The xor result is S.
    - If S is 1, use the 'twos_complement' to determine two's complement form of Q.
  - Determine sign S of R
    - Extract $a0[31] and assign this to S
    - If S is 1, use the 'twos_complement' to determine two's complement form of R.

Figure 21: Calculating Sign of Quotient and Remainder

The sign of the quotient is determined similarly as the product’s sign was determined in the signed multiplication logic. The sign of the remainder is determined by checking the MSB of the dividend. If it is 1, then the remainder is its complemented value otherwise it is the current value.

IV. TESTING IMPLEMENTATION

After a lot of back and forth with revising my code and debugging, I was finally able to get a 40/40 result on my program. The main problems I ran into were using the same temporary register across procedures and having the values mixed up. I have manually gone through and

checked all the values of the output of my program and from my testing see that it is all accurate and correct. I assembled the code in the "proj-auto-test.asm" file which contains the tester code provided by the professor and everything ran perfectly with a 40/40 output. I modified the numbers here and there to further test my output and everything remained fully functional.

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


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

-- program is finished running --
```

Figure 22: Output of Program

V. CONCLUSION

Overall, this project was a very tough, tedious, and time consuming one. Before learning about the logical operations, I didn't think a calculator could be created just using AND, OR, NOT, and XOR operations and not using the normal add, subtract, multiply, and divide functions we are so used to. It's clear that this level of code is underappreciated since this is how the ALU has to be implemented when working with only 0's and 1's. I feel like I have a lot better understanding of how the hardware works and the underlying process of arithmetic operations. After completing this project, I feel a lot more comfortable working in MIPS assembly code and even debugging in assembly code.