Arithmetic Operations Implemented Using MIPS Logic and Design

Diljot Singh
San Jose State University
diljot.singh@sjsu.edu

***Abstract* – This project report describes the implementation of arithmetic operations in MIPS assembly language using a) calls to the MIPS instruction set (normal procedure) and b) logical design (logical procedure) through Boolean logic, bit manipulation, logical gates, both conducted through the MARS integrated development environment.**

I. INTRODUCTION

MIPS (Microprocessor without Interlocked Pipelined Stages) is an instruction architecture which supports what is known as "MIPS Assembly Language." This assembly language gets translated to machine code through an assembler, where it can then be executed by the computer. In this project, we will be using a tool known as "MARS MIPS simulator," an integrated development environment which supports the MIPS processor to simulate the functionality of the arithmetic logic unit (ALU) in the processor.

Specifically, we will be implementing mathematical operations such as addition, subtraction, multiplication, and division through both their normal implementations (calls to MIPS instructions) and their logical implementations (using bitwise operations and Boolean logic). The objectives of this project are to:

1. Install and setup the MARS simulator environment.
2. Implement arithmetic operations using both built in MIPS instructions (such as add, sub, mult, div) and logical operations (logic gates and bit manipulation).
3. Test both implementations against each other in the MARS IDE to ensure full functionality and computability.
4. Better understand the computer architecture system components both holistically and individually.

II. REQUIREMENTS (Installation and Setup)

A. *Installation of tools and source files*

1) The MARS MIPS Simulator environment can be downloaded online at https://courses.missouristate.edu/KenVollmar/MARS/download.htm. There may be additional requirements such as having the Java SDK (software development kit) installed, which are mentioned on the MARS installation page.

2) The project file structure is provided through a .zip file and can be downloaded through Canvas at https://sjsu.instructure.com/courses/1324477/files/54503907/download?wrap=1

Move the downloaded zip file into a desired directory and unzip it. Verify that it contains the following files:

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

B. *Launching the project*

Launch MARS and navigate to “File” (located in the upper left-hand corner), as shown in Figure 1.

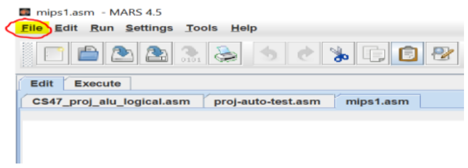
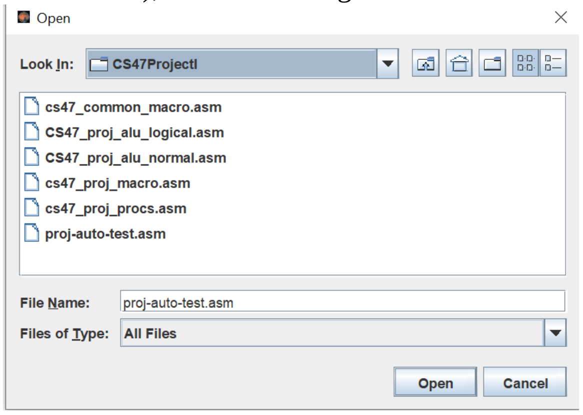


Fig 1. Opening files in MARS (1)

Click on “File” and then click “Open.” This will display various directories and file structures; navigate to where the unzipped files are saved on the user’s computer system. Select one of the files and then click open (in the bottom right-hand corner), as shown in Figure 2.



Look In: CS47Projectl

cs47_common_macro.asm
CS47_proj_alu_logical.asm
CS47_proj_alu_normal.asm
cs47_proj_macro.asm
cs47_proj_procs.asm
proj-auto-test.asm

File Name: proj-auto-test.asm
Files of Type: All Files

Fig 2. Opening files (2)

This will redirect the user back to the MARS environment with the selected file now open.

Repeat procedure until all project files have been opened.

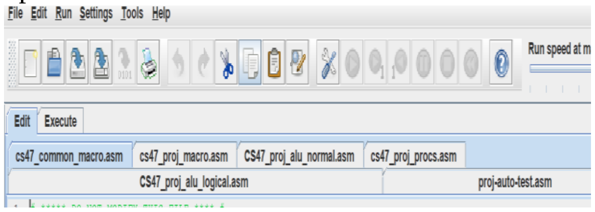


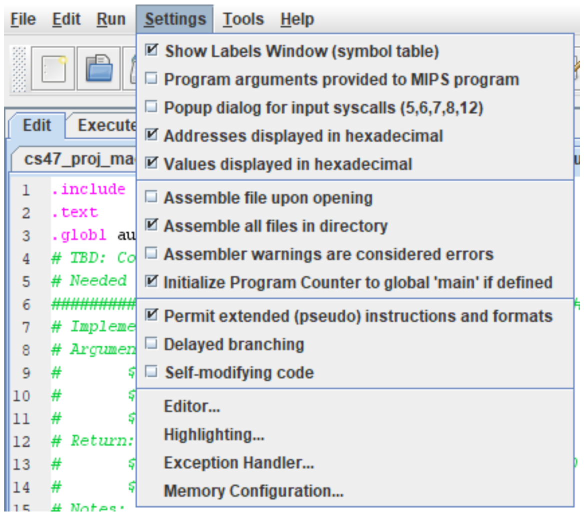
Fig 3. Project file access

As depicted in Figure 3, each file tab can be selected to be displayed in the MARS environment. The currently selected file (in this case, cs47_common_macro.asm) is highlighted blue while the other tabs are not. The three main files we will be working with are:

1) cs47_proj_alu_logical.asm
   This file will contain the logical implementation of the mathematical operations.
2) cs47_proj_alu_normal.asm
   This file will contain the normal implementation (MIPS instruction set) of the mathematical operations.
3) cs47_proj_macro.asm
   This file will contain any additional macros needed to assist in the implementations.

C. Initialization

Note: In MARS settings, make sure the boxes next to “Assembles all files in directory” and “Initialize program counter to global main if defined” are checked, as depicted in Figure 4. This will ensure that the program starts where the “main” label is in the code; not doing so could lead to unexpected errors.



☑ Show Labels Window (symbol table)
☐ Program arguments provided to MIPS program
☐ Popup dialog for input syscalls (5,6,7,8,12)
☑ Addresses displayed in hexadecimal
☑ Values displayed in hexadecimal
☐ Assemble file upon opening
☑ Assemble all files in directory
☐ Assembler warnings are considered errors
☑ Initialize Program Counter to global 'main' if defined
☑ Permit extended (pseudo) instructions and formats
☐ Delayed branching
☐ Self-modifying code

```
.include
.text
.globl au
# TBD: Co
# Needed
##########
# Impleme
# Argumen
#        $
#        $
#        $
# Return:
#        $
#        $
# Notes:
```

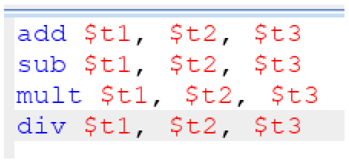
Fig 4. Initializing settings

III. REQUIREMENTS OF ARITHMETIC PROCEDURES

As mentioned, there will be two implementations of the mathematical operations: normal and logical. The normal procedure will utilize the built-in MIPS instruction set and the logical procedure will incorporate Boolean logic gates and bit manipulation. We will discuss both implementations in more detail in the “Design and Implementation” section.

A. *Normal procedure*
The normal procedure will be implemented in the cs47_proj_alu_normal.asm file and is referred to as au_normal. The procedure will take three arguments:
    1) Register $a0 – The first operand in the mathematical expression.
    2) Register $a1 – The second operand in the mathematical expression.
    3) Register $a2 – The operator (“+”, “-“, “*”, “/”), indicating which operation is to be performed (addition, subtraction, multiplication, division).

To compute the result of each mathematical operation in the normal procedure, the MIPS instruction set can be used. For example, to add two numbers, one can simply type “add _, _, _” which will call the built in add instruction to compose the sum of the last two arguments and store it into the first argument. Figure 5 displays examples of MIPS instructions utilized in the normal procedures.



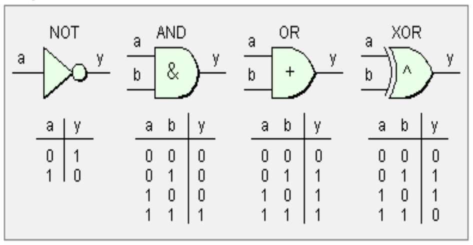
```
add $t1, $t2, $t3
sub $t1, $t2, $t3
mult $t1, $t2, $t3
div $t1, $t2, $t3
```

Fig 5. MIPS Instruction Set call example

For addition and subtraction, the results should be stored in the $v0 register. For multiplication, the low order of 32 bits (Lo) should be stored in $v0 and the high order of 32 bits (Hi) should be stored in $v1. For division, the quotient should be stored in $v0 and the remainder in $v1.

B. *Logical procedure*
The logical procedure will be implemented in the cs47_proj_alu_logical.asm file and is referred to as au_logical. In contrast with the normal procedure, the MIPS instruction set cannot be used to directly generate the result for this implementation. Instead, we must use logical operations such as AND, OR, NOT, and XOR.

The truth tables for these logical gates mentioned are shown for reference in Figure 6.



a	y
0	1
1	0

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

Fig 6. Logical Gates and their Truth Tables

We are also permitted to use bit manipulation through macros, shifts, loops, which will be discussed more in the “Design and Implementation” section.

The logical procedure will take three arguments:
    1) Register $a0 – The first operand in the mathematical expression.
    2) Register $a1 – The second operand in the mathematical expression.
    3) Register $a2 – The operator (“+”, “-“, “*”, “/”), indicating which operation is to be performed.

Due to the added complexity of the logical implementation, multiple procedures will be needed to supplement the mathematical computations, which will also be discussed in the next section.

IV. DESIGN AND IMPLEMENTATION

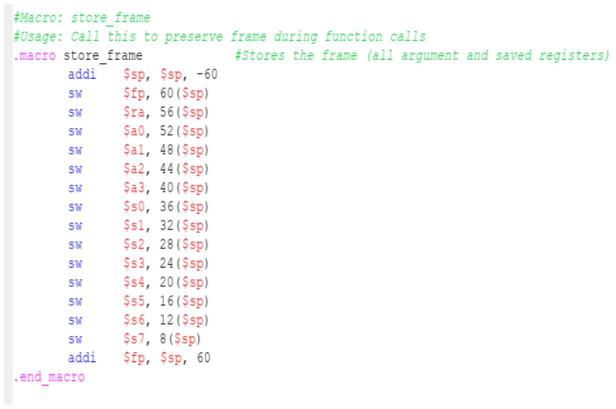
A. *Macros*

Before delving into the implementations, let us discuss the macros utilized in the “cs47_proj_macro.asm” file.

Macros, short for “macroinstruction,” contain a specific sequence of instructions which can be called using the macro’s name. This allows users to save both space and time, especially for extremely repetitive or extensive segments of code. Rather than typing the same instructions over and over again, the programmer can call the macro which contains those instructions.

The macros utilized for this project’s implementations are:

1) store_frame



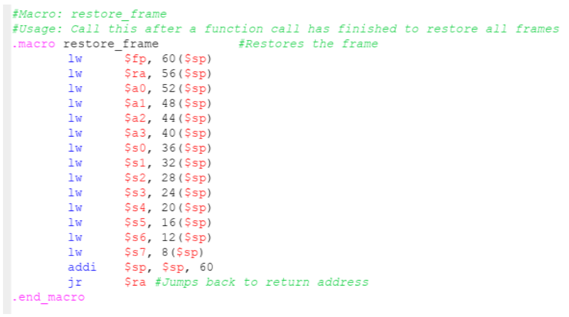
```
#Macro: store_frame
#Usage: Call this to preserve frame during function calls
.macro store_frame              #Stores the frame (all argument and saved registers)
        addi    $sp, $sp, -60
        sw      $fp, 60($sp)
        sw      $ra, 56($sp)
        sw      $a0, 52($sp)
        sw      $a1, 48($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
```

Fig 7. Store_frame macro

Stack frames are used in MIPS to preserve memory and prevent data loss as a result of subroutine calls in procedures. It is a good implementation design to store the frame first in every procedure in order to allocate memory for registers which may be needed across multiple calls.

In this macro, shown in Figure 7, we are storing $sp (stack pointer), $fp (frame pointer), $ra (return address), all argument registers ($a0 - $a3), and all saved registers ($s0 - $s7). By simply typing “store_frame” (the name of the macro) in any procedure, all of these instructions are executed.

2) restore_frame

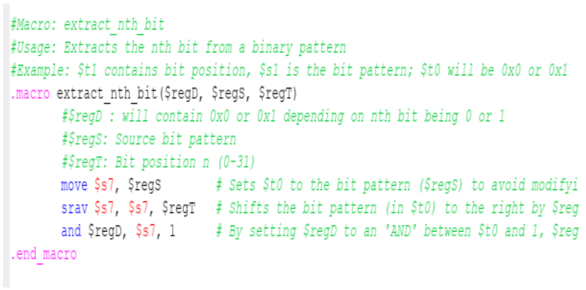


```
#Macro: restore_frame
#Usage: Call this after a function call has finished to restore all frames
.macro restore_frame            #Restores the frame
        lw      $fp, 60($sp)
        lw      $ra, 56($sp)
        lw      $a0, 52($sp)
        lw      $a1, 48($sp)
        lw      $a2, 44($sp)
        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 #Jumps back to return address
.end_macro
```

Fig 8. Restore_frame macro

Similar to the store_frame macro, we create a “restore_frame” macro which will be called at the end of every procedure, depicted in Figure 8. This restores the stack space allocated when we called the store_frame macro.

3) extract_nth_bit



```
#Macro: extract_nth_bit
#Usage: Extracts the nth bit from a binary pattern
#Example: $t1 contains bit position, $s1 is the bit pattern; $t0 will be 0x0 or 0x1
.macro extract_nth_bit($regD, $regS, $regT)
        #$regD : will contain 0x0 or 0x1 depending on nth bit being 0 or 1
        #$regS: Source bit pattern
        #$regT: Bit position n (0-31)
        move $s7, $regS         # Sets $t0 to the bit pattern ($regS) to avoid modifyi
        srav $s7, $s7, $regT    # Shifts the bit pattern (in $t0) to the right by $reg
        and $regD, $s7, 1       # By setting $regD to an 'AND' between $t0 and 1, $reg
.end_macro
```

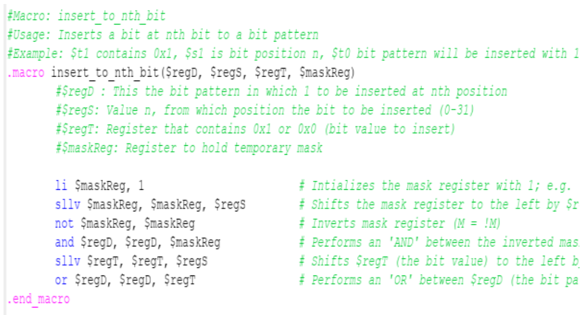
Fig 9. Extract_nth_bit Macro

The extract_nth_bit macro is used to access specific bits in a bit pattern, its implementation depicted in Figure 9. It has three arguments:

- $regD – the bit we extract
- $regS – the source bit pattern
- $regT – the bit position (0-31)

We will use this macro extensively in our alu_logical implementation to access specific bits and perform operations on them.

4) insert_to_nth_bit



```
#Macro: insert_to_nth_bit
#Usage: Inserts a bit at nth bit to a bit pattern
#Example: $t1 contains 0x1, $s1 is bit position n, $t0 bit pattern will be inserted with 1
.macro insert_to_nth_bit($regD, $regS, $regT, $maskReg)
	#$regD : This the bit pattern in which 1 to be inserted at nth position
	#$regS: Value n, from which position the bit to be inserted (0-31)
	#$regT: Register that contains 0x1 or 0x0 (bit value to insert)
	#$maskReg: Register to hold temporary mask

	li $maskReg, 1                     # Intializes the mask register with 1; e.g.
	sllv $maskReg, $maskReg, $regS     # Shifts the mask register to the left by $r
	not $maskReg, $maskReg             # Inverts mask register (M = !M)
	and $regD, $regD, $maskReg         # Performs an 'AND' between the inverted mas
	sllv $regT, $regT, $regS           # Shifts $regT (the bit value) to the left b
	or $regD, $regD, $regT             # Performs an 'OR' between $regD (the bit pa
.end_macro
```

Fig 10. Insert_nth_bit Macro

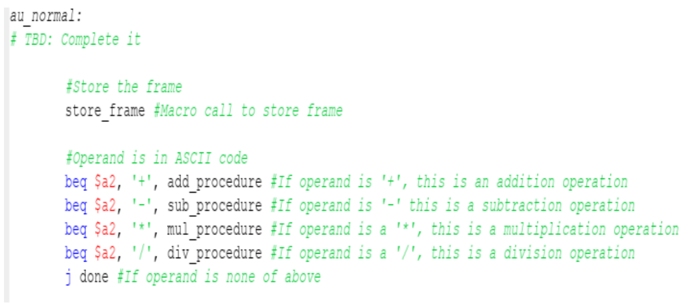
While the extract_nth_bit macro is used to access specific bits from patterns, the insert_nth_bit macro is used to insert bits into patterns at the nth location, being shown in Figure 10. This macro takes four arguments:

- $regD – the bit pattern in which insertion will take place
- $regS – the bit position n where we will insert
- $regT – the register that will contain the bit we want to insert
- $maskReg – a temporary mask register used to perform the insertion

Again, while this macro is not used in the alu_normal implementation, we use it extensively in the alu_logical implementation to modify bit patterns.

B. *Normal Procedures*

The normal procedures are relatively straight forward in terms of the implementation. First, we start off with four branch statements which test the $a2 (operator) register. Based on which statement is true, the program will redirect to that procedure accordingly, as shown in Figure 11.



```
au_normal:
# TBD: Complete it

	#Store the frame
	store_frame #Macro call to store frame

	#Operand is in ASCII code
	beq $a2, '+', add_procedure #If operand is '+', this is an addition operation
	beq $a2, '-', sub_procedure #If operand is '-' this is a subtraction operation
	beq $a2, '*', mul_procedure #If operand is a '*', this is a multiplication operation
	beq $a2, '/', div_procedure #If operand is a '/', this is a division operation
	j done #If operand is none of above
```

Fig 11. Normal Procedure Branches

We will now explore each of the procedure implementations (add_procedure, sub_procedure, mul_procedure, and div_procedure) individually.

1) Addition Implementation

```
#Addition procedure, adds $a0 and $a1
#Return value is in $v0
add_procedure:
store_frame #Macro call to store frame
add $v0, $a0, $a1 #$v0 = $a0 + $a1
restore_frame #Macro call to restore frame
j done #Done
```

Fig 12. Normal Addition Implementation

The normal addition procedure is shown in Figure 12 and is implemented as follows:

- Store the frame (standard in any procedure call)
- Use the 'add' instruction discussed earlier to compute $a0 + $a1, storing that computation in $v0
- Restore the frame (standard in any procedure call)
- A jump to the 'done' label, which simply goes to the end of the program, signifying completion

Note: The "store_frame" and "restore_frame" macros are discussed earlier.

2) Subtraction Implementation

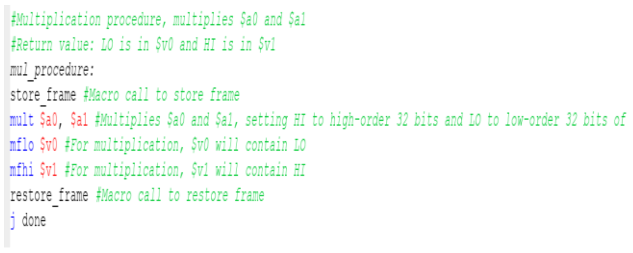
```
#Subtraction procedure, subtracts $a1 from $a0
#Return value is in $v0
sub_procedure:
store_frame #Macro call to store frame
sub $v0, $a0, $a1 #$v0 = $a0 - $a1
restore_frame #Macro call to restore frame
j done #Done
```

Fig 13. Normal Subtraction Implementation

Similar to the addition implementation, the subtraction procedure is implemented as follows, also being shown in Figure 13:

- Store the frame (standard in any procedure call)
- Use the ‘sub’ instruction discussed earlier to compute $a0 - $a1, storing that computation in $v0
- Restore the frame (standard in any procedure call)
- A jump to the ‘done’ label, which simply goes to the end of the program, signifying completion

3) Multiplication Implementation



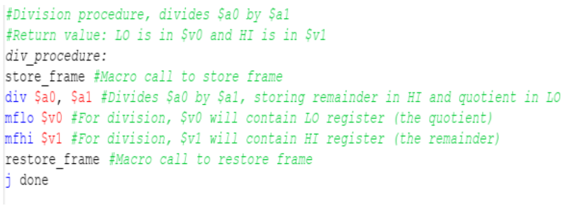
```
#Multiplication procedure, multiplies $a0 and $a1
#Return value: LO is in $v0 and HI is in $v1
mul_procedure:
store_frame #Macro call to store frame
mult $a0, $a1 #Multiplies $a0 and $a1, setting HI to high-order 32 bits and LO to low-order 32 bits of
mflo $v0 #For multiplication, $v0 will contain LO
mfhi $v1 #For multiplication, $v1 will contain HI
restore_frame #Macro call to restore frame
j done
```

Fig 14. Normal Multiplication Implementation

For the normal multiplication procedure, the implementation is as follows, also shown in Figure 14:

- Store the frame
- Use the ‘mult’ instruction to compute $a0 * $a1, which automatically stores the high-order 32 bits in the “Hi” register and the low-order 32 bits in the “Lo” register
- We must use special instructions to access Hi and Lo since these are special registers. We use the instruction “mflo” (move from LO) to move the contents of the Lo register into $v0
- We use the instruction “mfhi” (move from HI) to move the contents of the Hi register into $v1
- Restore the frame
- Jump to the done label

4) Division Implementation



```
#Division procedure, divides $a0 by $a1
#Return value: LO is in $v0 and HI is in $v1
div_procedure:
store_frame #Macro call to store frame
div $a0, $a1 #Divides $a0 by $a1, storing remainder in HI and quotient in LO
mflo $v0 #For division, $v0 will contain LO register (the quotient)
mfhi $v1 #For division, $v1 will contain HI register (the remainder)
restore_frame #Macro call to restore frame
j done
```

Fig 15. Normal Division Implementation

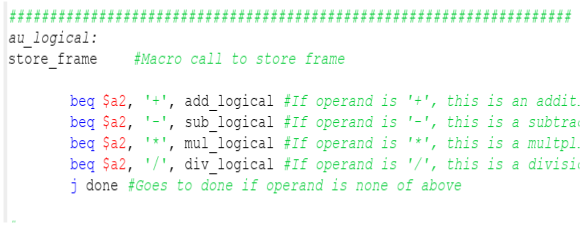
For the normal division procedure, we implement it as follows:

- Store the frame
- Call the “div” instruction from MIPS instruction set. This instruction automatically stores the remainder in the Hi register and the quotient in the Lo register
- Use “mflo” to move the quotient into the $v0 register
- Use “mfhi” to move the remainder into the $v1 register
- Restore the frame
- Jump to the done label

The implementation in MIPS is portrayed in Figure 15.

C. *Logical Procedures*

Similar to the normal procedures, we include the branch statements to determine which procedure the program should execute, based on the operator in the $a2 register.



```
#################################################################
au_logical:
store_frame    #Macro call to store frame

       beq $a2, '+', add_logical #If operand is '+', this is an addit.
       beq $a2, '-', sub_logical #If operand is '-', this is a subtra
       beq $a2, '*', mul_logical #If operand is '*', this is a multpl.
       beq $a2, '/', div_logical #If operand is '/', this is a divisi
       j done #Goes to done if operand is none of above
```

Fig 16. Logical Branch statements

Before continuing to the logical implementations however, it is necessary to discuss the utility procedures which substantiate the logic procedures. These utility procedures are called extensively throughout the alu_logical implementations and perform functions such as computing negatives, performing the mathematical operation logic, and controlling the program flow.

1) **Utility**: Common add_sub_logical

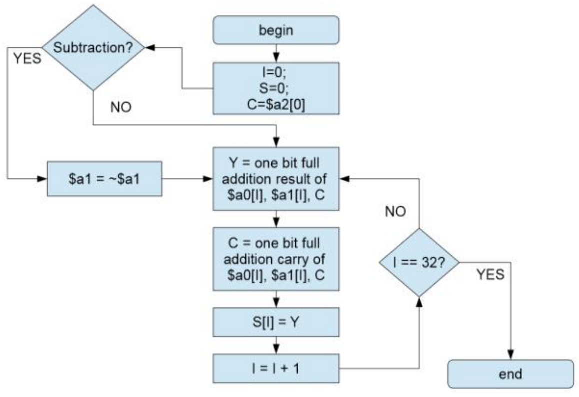
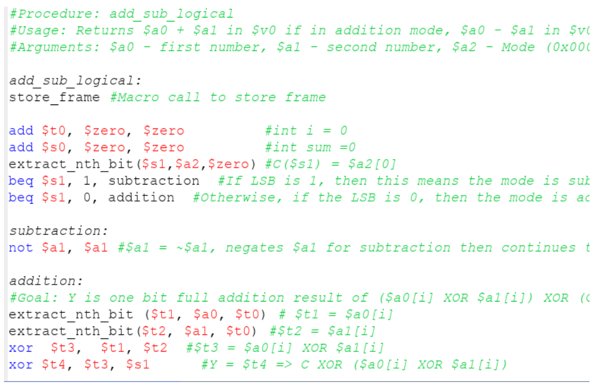

begin
I=0; S=0; C=$a2[0]
Subtraction?
YES
NO
$a1 = ~$a1
Y = one bit full addition result of $a0[I], $a1[I], C
C = one bit full addition carry of $a0[I], $a1[I], C
S[I] = Y
I = I + 1
I == 32?
NO
YES
end

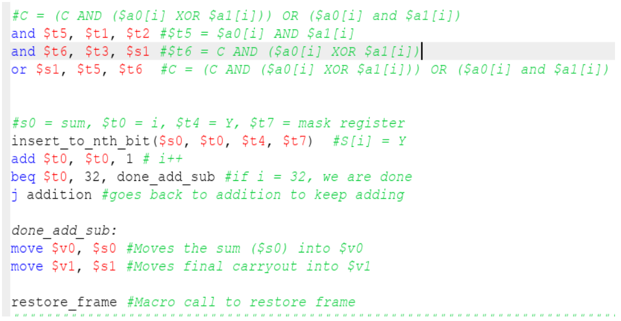
Fig 17. Add_sub_logical flowchart

The add_sub_logical utility procedure is used in both the add_logical and sub_logical procedures. The idea of the procedure is that it uses a loop to compute the sums and carries of every bit until every bit has been accessed. Following the flow chart in Figure 17, we can construct an implementation as follows, shown in Figure 18.



```
#Procedure: add_sub_logical
#Usage: Returns $a0 + $a1 in $v0 if in addition mode, $a0 - $a1 in $v
#Arguments: $a0 - first number, $a1 - second number, $a2 - Mode (0x00

add_sub_logical:
store_frame #Macro call to store frame

add $t0, $zero, $zero          #int i = 0
add $s0, $zero, $zero          #int sum =0
extract_nth_bit($s1,$a2,$zero) #C($s1) = $a2[0]
beq $s1, 1, subtraction  #If LSB is 1, then this means the mode is sub
beq $s1, 0, addition  #Otherwise, if the LSB is 0, then the mode is ac

subtraction:
not $a1, $a1 #$a1 = ~$a1, negates $a1 for subtraction then continues t

addition:
#Goal: Y is one bit full addition result of ($a0[i] XOR $a1[i]) XOR (
extract_nth_bit ($t1, $a0, $t0) # $t1 = $a0[i]
extract_nth_bit($t2, $a1, $t0) #$t2 = $a1[i]
xor  $t3,  $t1, $t2  #$t3 = $a0[i] XOR $a1[i]
xor $t4, $t3, $s1      #Y = $t4 => C XOR ($a0[i] XOR $a1[i])
```

Fig 18. Add_sub_logical implementation



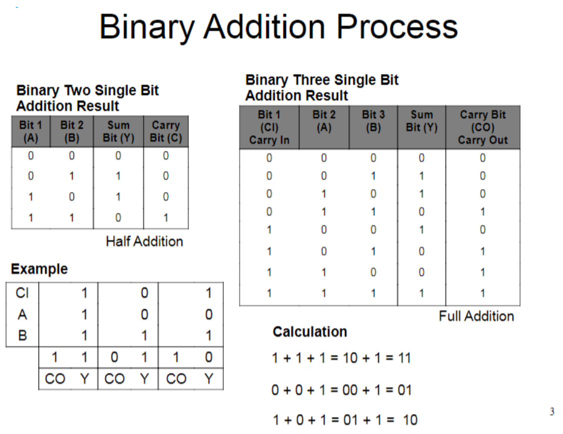
```
#C = (C AND ($a0[i] XOR $a1[i])) OR ($a0[i] and $a1[i])
and $t5, $t1, $t2 #$t5 = $a0[i] AND $a1[i]
and $t6, $t3, $s1 #$t6 = C AND ($a0[i] XOR $a1[i])
or $s1, $t5, $t6  #C = (C AND ($a0[i] XOR $a1[i])) OR ($a0[i] and $a1[i])


#s0 = sum, $t0 = i, $t4 = Y, $t7 = mask register
insert_to_nth_bit($s0, $t0, $t4, $t7)  #S[i] = Y
add $t0, $t0, 1 # i++
beq $t0, 32, done_add_sub #if i = 32, we are done
j addition #goes back to addition to keep adding

done_add_sub:
move $v0, $s0 #Moves the sum ($s0) into $v0
move $v1, $s1 #Moves final carryout into $v1

restore_frame #Macro call to restore frame
```

Fig 18.1 Add_sub_logical implementation

This utility procedure has three arguments: $a0 – the first operand, $a1 – the second operand, and $a2 – the mode (either addition or subtraction). If the mode is addition, then the $a2 register will be equal to 0x00000000, according to our convention. On the contrary, if the $a2 register contains 0xFFFFFFFF, this signifies that the procedure is to perform subtraction. The only difference between the addition and subtraction modes are that in subtraction we invert $a1 first before continuing to the implementation (since A – B = A + (-B)). Therefore, our implementation can support both addition and subtraction.

We extract one bit from both operands ($a0 and $a1) starting from the LSB side. By performing an XOR between these two bits, we are essentially performing a sum and trying to compute the carry. We then perform another XOR between the carry and the two bits, following with two AND's and one OR to get the final carry. Once this is obtained, we insert the sum to the $s0 register (which is storing the sum) and continue looping.

This implementation will summate every corresponding bit one at a time and move their sums into the $s0 register. Once $t0 (the loop counter) is equal to 32, that signifies that we have accessed every bit and we are done. Once the loop has completed, we will set $v0 (the return value) to the sum we calculated in our loop ($s0). Also, the final carryout computed will be saved in $v1 for enhanced functionality in other procedures.



Binary Addition Process

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

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
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Full Addition

Example

CI		1		0		1
A		1		0		0
B		1		1		1
	1	1	0	1	1	0
	CO	Y	CO	Y	CO	Y

Calculation

1 + 1 + 1 = 10 + 1 = 11

0 + 0 + 1 = 00 + 1 = 01

1 + 0 + 1 = 01 + 1 = 10

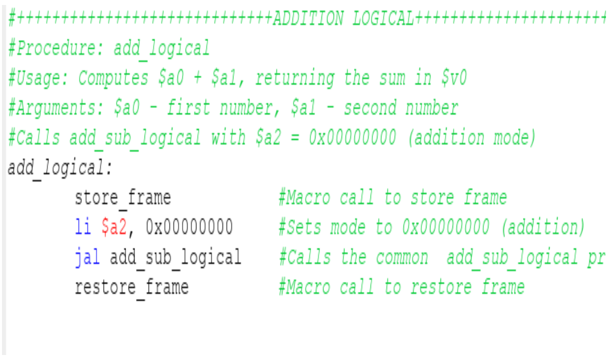
3

Fig 19. Binary addition process

Figure 19 depicts the process of computing the carries and sums of every bit using both a half adder and full adder, the same logic being implemented in our common add_sub_logical. In particular, we pay close attention to the carry out (CO) and carry in (CI) bits, which we can use to simplify both addition and subtraction, as depicted in Figure 22.

2) add_logical

Once the implementation of the common add_sub_logical has been completed (see Figure 18), all the add_logical must do is simply make a call to the utility procedure with the mode set to addition, as depicted in Fig 20.



```
#++++++++++++++++++++++++++++++ADDITION LOGICAL++++++++++++++++++++++
#Procedure: add_logical
#Usage: Computes $a0 + $a1, returning the sum in $v0
#Arguments: $a0 - first number, $a1 - second number
#Calls add_sub_logical with $a2 = 0x00000000 (addition mode)
add_logical:
        store_frame             #Macro call to store frame
        li $a2, 0x00000000      #Sets mode to 0x00000000 (addition)
        jal add_sub_logical     #Calls the common  add_sub_logical pr
        restore_frame           #Macro call to restore frame
```

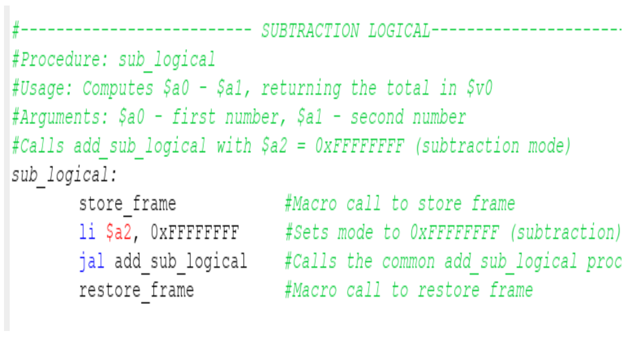
Fig 20. add_logical procedure

In add_logical, the steps we take are as follows:

- Store the frame
- Load the mode register ($a2) with the immediate value 0x00000000, signifying addition.
- Jump and link ('jal') to the common add_sub_logical which will compute the sum and return it in register $v0
- Restore the frame

3) sub_logical

Similar to add_logical, the sub_logical utilizes the common add_sub_logical utility procedure created.



```
#------------------------- SUBTRACTION LOGICAL---------------------
#Procedure: sub_logical
#Usage: Computes $a0 - $a1, returning the total in $v0
#Arguments: $a0 - first number, $a1 - second number
#Calls add_sub_logical with $a2 = 0xFFFFFFFF (subtraction mode)
sub_logical:
        store_frame             #Macro call to store frame
        li $a2, 0xFFFFFFFF      #Sets mode to 0xFFFFFFFF (subtraction)
        jal add_sub_logical     #Calls the common add_sub_logical proc
        restore_frame           #Macro call to restore frame
```

Fig 21. sub_logical procedure

The steps we take in sub_logical are as follows:

- Store the frame
- Load the mode register ($a2) with the convention we set for subtraction: 0xFFFFFFFF
- Jump and link to the add_sub_logical utility procedure, which will return the total in $v0
- Restore the frame

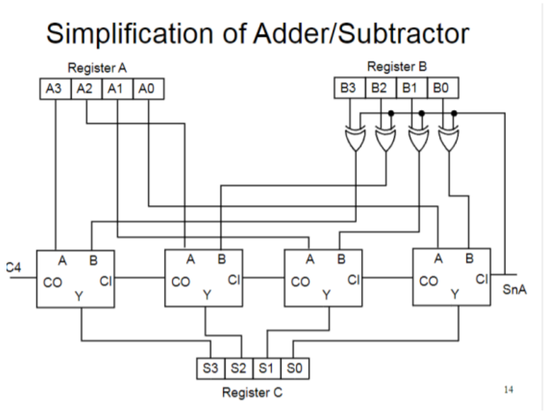
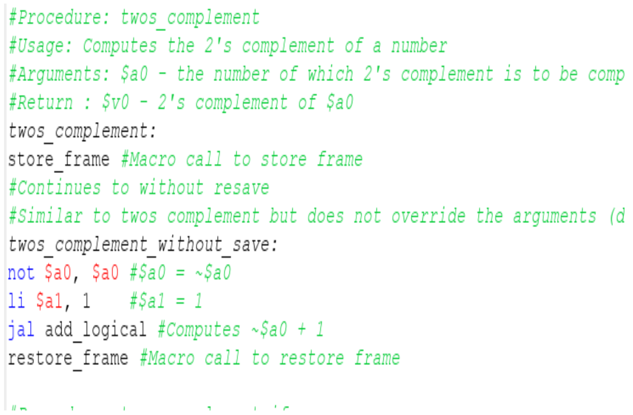


Fig 22. Combination of adder / subtractor

Figure 22 portrays the simplification of the adder and subtractor digital circuits through their combination. We can see how through designating registers for the carry in (CI), carry out (CO), and input/output, both subtraction and addition can be designed, which is what we strove to achieve through our common add_sub_logical procedure.

4) **Utility:** twos_complement

This utility procedure is used to compute the 2's complement of a number. 2's complement is used to represent both positive and negative numbers in binary. If the MSB of a 2's complement number is 0, the number is positive, and if the MSB is 1, the number is negative. We implement the twos_complement as such:



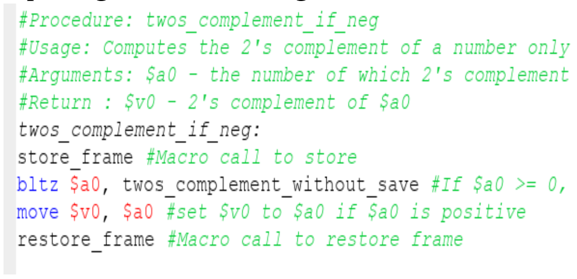
```
#Procedure: twos_complement
#Usage: Computes the 2's complement of a number
#Arguments: $a0 - the number of which 2's complement is to be comp
#Return : $v0 - 2's complement of $a0
twos_complement:
store_frame #Macro call to store frame
#Continues to without resave
#Similar to twos complement but does not override the arguments (d
twos_complement_without_save:
not $a0, $a0 #$a0 = ~$a0
li $a1, 1    #$a1 = 1
jal add_logical #Computes ~$a0 + 1
restore_frame #Macro call to restore frame
```

Fig 23. Twos_complement utility procedure

In this procedure, we are receiving one argument: $a0, which is the number we are trying to compute the complement of. The complement of the number is returned in the register $v0. The steps we follow are:

- Compute the inverse of our input (Set $a0 to ~$a0).
- Load the $a1 register with the immediate value "1" for our next procedure call
- Call the add_logical procedure to compute ~$a0 + 1. This effectively calculates the 2's complement and stores it in register $v0
- Restore the frame

5) **Utility:** twos_complement_if_neg
This utility procedure is a slightly tweaked version of our earlier twos_complement. Now, the 2's complement is only computed if the input argument $a0 is negative.



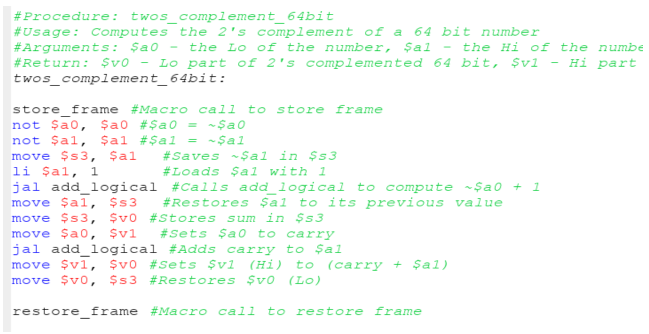
```
#Procedure: twos_complement_if_neg
#Usage: Computes the 2's complement of a number only
#Arguments: $a0 - the number of which 2's complement
#Return : $v0 - 2's complement of $a0
twos_complement_if_neg:
store_frame #Macro call to store
bltz $a0, twos_complement_without_save #If $a0 >= 0,
move $v0, $a0 #set $v0 to $a0 if $a0 is positive
restore_frame #Macro call to restore frame
```

Fig 24. Twos_complement_if_neg procedure

The only difference is that we have added a conditional check to see if $a0 is negative: "bltz $a0, twos_complement_without_save." What this statement signifies is that if the $a0 register is less than zero, only then do we call the twos_complement procedure created earlier. Otherwise, we simply move the $a0 argument into $v0 right away.

6) **Utility:** twos_complement_64bit

While computing the 2's complement of a single number is relatively simple, it is not the case for a 64 bit value. This is because the value will be stored in two separate registers, such as Hi and Lo, and the 2's complement should be computed considering the entire value, not two separate values. However, this can be still done through a series of procedure calls, as shown in the implementation in Figure 25.



```
#Procedure: twos_complement_64bit
#Usage: Computes the 2's complement of a 64 bit number
#Arguments: $a0 - the Lo of the number, $a1 - the Hi of the numbe
#Return: $v0 - Lo part of 2's complemented 64 bit, $v1 - Hi part
twos_complement_64bit:

store_frame #Macro call to store frame
not $a0, $a0 #$a0 = ~$a0
not $a1, $a1 #$a1 = ~$a1
move $s3, $a1   #Saves ~$a1 in $s3
li $a1, 1       #Loads $a1 with 1
jal add_logical #Calls add_logical to compute ~$a0 + 1
move $a1, $s3   #Restores $a1 to its previous value
move $s3, $v0 #Stores sum in $s3
move $a0, $v1  #Sets $a0 to carry
jal add_logical #Adds carry to $a1
move $v1, $v0 #Sets $v1 (Hi) to (carry + $a1)
move $v0, $s3 #Restores $v0 (Lo)

restore_frame #Macro call to restore frame
```

Fig 25. Twos_complement_64bit

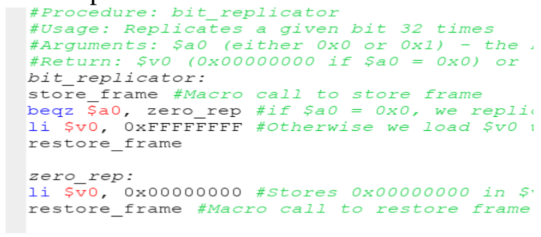
In this procedure we compute the 64 bit 2's complement as follows:

- Invert both $a0 and $a1 (the Lo and Hi of the number, respectively)
- Save the inverted $a1 in register $s3
- Load the $a1 register with the immediate value 1
- Calls the add_logical procedure to compute ~$a0 + 1
- Restore the inverted $a1
- Move the calculated 2's complement to $v0
- Move the carry to $a0
- Call the add_logical once more to compute carry + $a1
- Restore the Lo register with its 2's complement
- Restore the frame

What this implementation does is that it calls the 2's complement twice using the carry from the add_logical. This allows both the Hi and Lo registers to be complemented while being treated as a single value, due to the fact we are obtaining the carry from the Lo register and using this to complement the Hi register.

7) **Utility:** bit_replicator
The purpose this utility procedure is to replicate a given bit thirty-two times. It receives one argument: $a0, which is the bit we want to replicate (either 0x0 or 0x1). We should store the replicated pattern in $v0, which will be 0x00000000 if $a0 = 0x0, or 0xFFFFFFFF if $a0 = 1.

The implementation is as follows:



```
#Procedure: bit_replicator
#Usage: Replicates a given bit 32 times
#Arguments: $a0 (either 0x0 or 0x1) - the
#Return: $v0 (0x00000000 if $a0 = 0x0) or
bit_replicator:
store_frame #Macro call to store frame
beqz $a0, zero_rep #if $a0 = 0x0, we repli
li $v0, 0xFFFFFFFF #Otherwise we load $v0
restore_frame

zero_rep:
li $v0, 0x00000000 #Stores 0x00000000 in $
restore_frame #Macro call to restore frame
```

Fig 26. Bit_replicator

As shown in Figure 26, this procedure is created using a branch statement redirecting to the corresponding replication code. If the $a0 is detected as being equal to zero, we branch to the zero_rep label and load the $v0 register with the immediate value 0x00000000. Otherwise, we load the $v0 register with the value 0xFFFFFFFF.

8) **Utility:** mul_unsigned

Mul_unsigned is used to compute the result of multiplying two unsigned numbers. It takes two arguments: $a0 (the first operand) and $a1 (the second operand). After the multiplication is complete, it will return the Lo part of the result in $v0 and the Hi component in $v1.

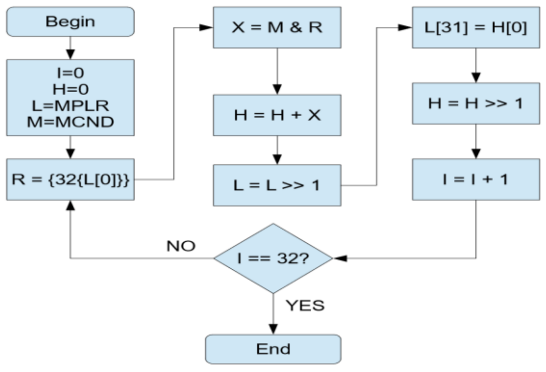
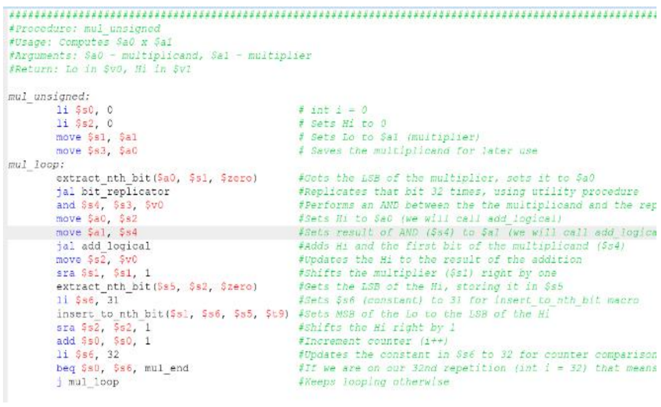


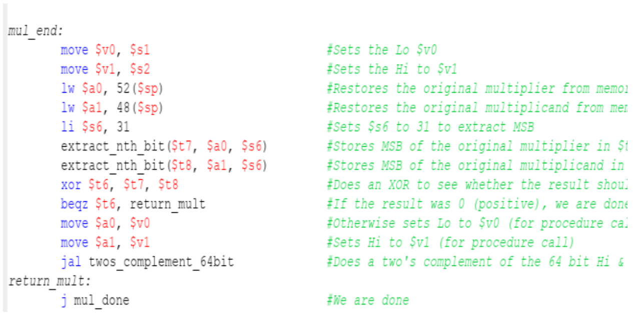
Fig 27. Unsigned multiplication algorithm

Due to the fact that the operands are 32 bits but the result is 64 bits, we must account for both the Hi and Lo registers in our code using right shifts and bit insertions. Using the algorithm flowchart depicted in Figure 27, we create an implementation as such:



```
############################################################################################
#Procedure: mul_unsigned
#Usage: Computes $a0 x $a1
#Arguments: $a0 - multiplicand, $a1 - multiplier
#Return: Lo in $v0, Hi in $v1

mul_unsigned:
        li $s0, 0                               # int i = 0
        li $s2, 0                               # Sets Hi to 0
        move $s1, $a1                           # Sets Lo to $a1 (multiplier)
        move $s3, $a0                           # Saves the multiplicand for later use
mul_loop:
        extract_nth_bit($a0, $s1, $zero)        #Gets the LSB of the multiplier, sets it to $a0
        jal bit_replicator                      #Replicates that bit 32 times, using utility procedure
        and $s4, $s3, $v0                       #Performs an AND between the the multiplicand and the rep
        move $a0, $s2                           #Sets Hi to $a0 (we will call add_logical)
        move $a1, $s4                           #Sets result of AND ($s4) to $a1 (we will call add_logica
        jal add_logical                         #Adds Hi and the first bit of the multiplicand ($s4)
        move $s2, $v0                           #Updates the Hi to the result of the addition
        sra $s1, $s1, 1                         #Shifts the multiplier ($s1) right by one
        extract_nth_bit($s5, $s2, $zero)        #Gets the LSB of the Hi, storing it in $s5
        li $s6, 31                              #Sets $s6 (constant) to 31 for insert_to_nth_bit macro
        insert_to_nth_bit($s1, $s6, $s5, $t9)   #Sets MSB of the Lo to the LSB of the Hi
        sra $s2, $s2, 1                         #Shifts the Hi right by 1
        add $s0, $s0, 1                         #Increment counter (i++)
        li $s6, 32                              #Updates the constant in $s6 to 32 for counter comparison
        beq $s0, $s6, mul_end                   #If we are on our 32nd repetition (int i = 32) that means
        j mul_loop                              #Keeps looping otherwise
```

Fig 28. mul_unsigned implementation

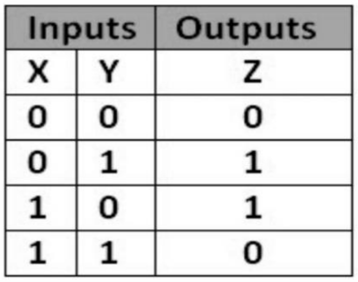


```
mul_end:
        move $v0, $s1                           #Sets the Lo $v0
        move $v1, $s2                           #Sets the Hi to $v1
        lw $a0, 52($sp)                         #Restores the original multiplier from memo
        lw $a1, 48($sp)                         #Restores the original multiplicand from me
        li $s6, 31                              #Sets $s6 to 31 to extract MSB
        extract_nth_bit($t7, $a0, $s6)          #Stores MSB of the original multiplier in $
        extract_nth_bit($t8, $a1, $s6)          #Stores MSB of the original multiplicand in
        xor $t6, $t7, $t8                       #Does an XOR to see whether the result shou
        beqz $t6, return_mult                   #If the result was 0 (positive), we are don
        move $a0, $v0                           #Otherwise sets Lo to $v0 (for procedure ca
        move $a1, $v1                           #Sets Hi to $v1 (for procedure call)
        jal twos_complement_64bit               #Does a two's complement of the 64 bit Hi &
return_mult:
        j mul_done                              #We are done
```

Fig 28.1 mul_unsigned implementation

Following the algorithm outlined, we can see how the necessary components of decimal and binary multiplication (such as right shifting the multipliers and having addition calls between every product) are present.

Furthermore, by implementing the "jal twos_complement_64bit" at the end of the multiplication loop, we can forgo having to create a separate procedure for signed multiplication and can simply do a test to see whether the multiplicand and multiplier were negative or not. This requires the usage of the "XOR" logical operation prior to the complementation, utilizing the fact that the XOR will return a 0 only when the MSB of the multiplier and multiplicand were the same. What this signifies in terms of the 2's complement in multiplication is that if both the multiplier and multiplicand were negative or positive (had the same MSB), then the product will be positive. However, if and only if there is exactly one negative operand in the procedure, XOR will return 1, signifying a negative product, which is where the "twos_complement_64bit" procedure is invoked. The inputs and outputs in relation to the XOR logical operation is shown in Figure 29.

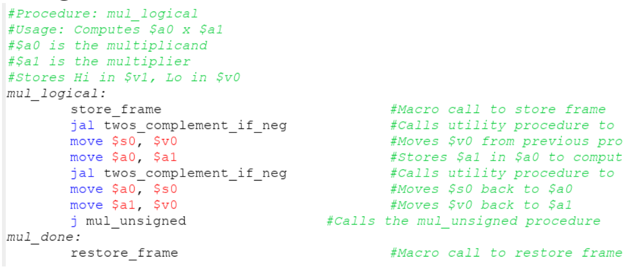


Inputs		Outputs
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

Fig 29. XOR Truth Table

9) mul_logical

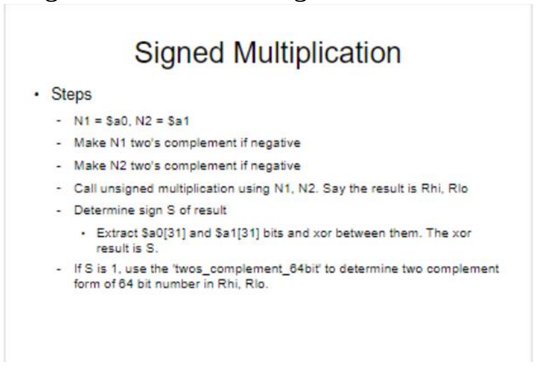
Similar to the add_logical and sub_logical procedures, the mul_logical procedure will make a call to the utility procedure "mul_unsigned" discussed earlier. The implementation is shown in Figure 30 below.



```
#Procedure: mul_logical
#Usage: Computes $a0 x $a1
#$a0 is the multiplicand
#$a1 is the multiplier
#Stores Hi in $v1, Lo in $v0
mul_logical:
        store_frame                             #Macro call to store frame
        jal twos_complement_if_neg              #Calls utility procedure to
        move $s0, $v0                           #Moves $v0 from previous pro
        move $a0, $a1                           #Stores $a1 in $a0 to comput
        jal twos_complement_if_neg              #Calls utility procedure to
        move $a0, $s0                           #Moves $s0 back to $a0
        move $a1, $v0                           #Moves $v0 back to $a1
        j mul_unsigned                      #Calls the mul_unsigned procedure
mul_done:
        restore_frame                           #Macro call to restore frame
```

Fig 30. Mul_logical procedure

Due to the fact that we are computing the multiplication of both signed and unsigned numbers in one procedure, we use the utility procedure "twos_complement_if_neg" to convert both operands to be positive, if they are not already so for the time being. Then, we call our mul_unsigned procedure which will take care of the multiplication logic and complement the numbers at the end if needed. This algorithm design is referenced in Figure 31.



Signed Multiplication

- Steps
  - N1 = $a0, N2 = $a1
  - Make N1 two's complement if negative
  - Make N2 two's complement if negative
  - Call unsigned multiplication using N1, N2. Say the result is Rhi, Rlo
  - Determine sign S of result
    - Extract $a0[31] and $a1[31] bits and xor between them. The xor result is S.
  - If S is 1, use the 'twos_complement_64bit' to determine two complement form of 64 bit number in Rhi, Rlo.

Fig 31. Signed multiplication algorithm

10) **Utility:** div_unsigned

Similar to mul_unsigned, div_unsigned will also take two arguments: $a0 (the first operand), and $a1 (the second operand). This procedure assumes the two operands are unsigned and after performing division, it will store the quotient in register $v0 and remainder in $v1.

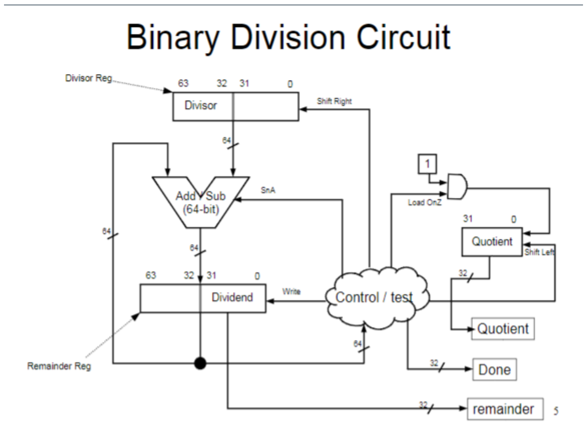


Fig 32. Binary Division Logical Circuit

As shown in the division logical circuit in Figure 32, division is a 64-bit operation which takes a 64-bit divisor and a 64-bit dividend. The control unit issues a right shift for the divisor and subtracts the divisor from the dividend. This will yield either a positive (+ve) or a negative (-ve) result.

If negative, the addition operation is called and will write the result into the register designated for the remainder. Then, the OnZ signal is assigned a value of 0, followed by the quotient register being shifted left by one bit.

Otherwise, if the result was positive, there will be no addition and the OnZ signal will be assigned a value of 1 instead. Then, the quotient will be shifted to the left by one bit as normal.

This will continue for thirty two repetitions total, with the quotient register containing the quotient and the remainder register containing the remainder at the end.

The algorithm we will follow to implement this digital design in MIPS is shown in Figure 33.

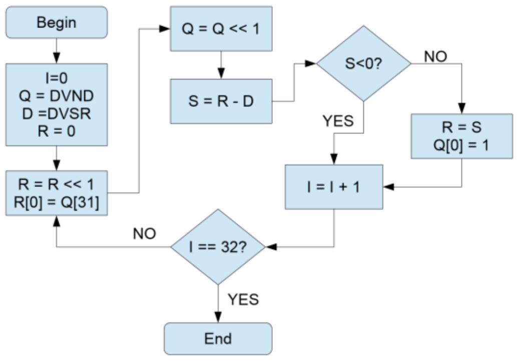
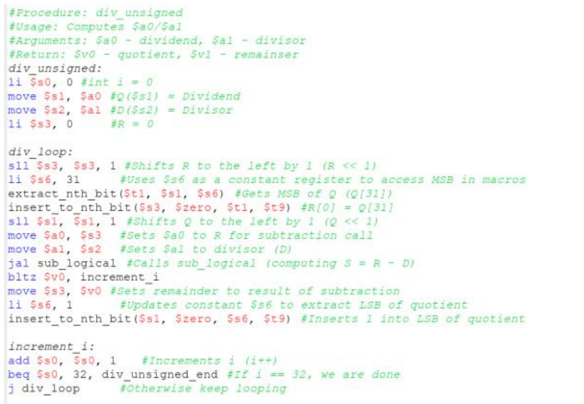


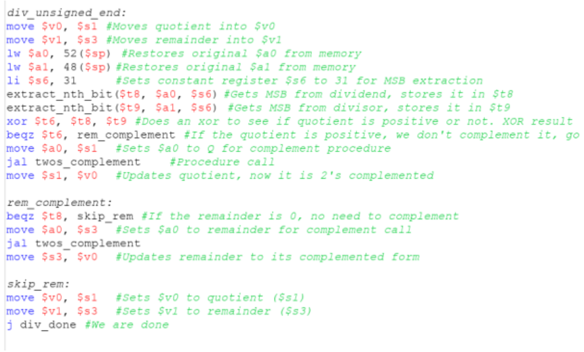
Fig 33. Unsigned division algorithm

Similar to multiplication, we will need to run a loop that repeats thirty-two times and designates registers from the beginning for the quotient (being initialized with the dividend) and the divisor. Furthermore, we initialize the remainder register with the value 0.

Following the algorithm flowchart in Figure 33, we implement the logic design MIPS as such:



```
#Procedure: div_unsigned
#Usage: Computes $a0/$a1
#Arguments: $a0 - dividend, $a1 - divisor
#Return: $v0 - quotient, $v1 - remainser
div_unsigned:
li $s0, 0 #int i = 0
move $s1, $a0 #Q($s1) = Dividend
move $s2, $a1 #D($s2) = Divisor
li $s3, 0     #R = 0

div_loop:
sll $s3, $s3, 1 #Shifts R to the left by 1 (R << 1)
li $s6, 31      #Uses $s6 as a constant register to access MSB in macros
extract_nth_bit($t1, $s1, $s6) #Gets MSB of Q (Q[31])
insert_to_nth_bit($s3, $zero, $t1, $t9) #R[0] = Q[31]
sll $s1, $s1, 1 #Shifts Q to the left by 1 (Q << 1)
move $a0, $s3   #Sets $a0 to R for subtraction call
move $a1, $s2   #Sets $a1 to divisor (D)
jal sub_logical #Calls sub_logical (computing S = R - D)
bltz $v0, increment_i
move $s3, $v0 #Sets remainder to result of subtraction
li $s6, 1       #Updates constant $s6 to extract LSB of quotient
insert_to_nth_bit($s1, $zero, $s6, $t9) #Inserts 1 into LSB of quotient

increment_i:
add $s0, $s0, 1    #Increments i (i++)
beq $s0, 32, div_unsigned_end #If i == 32, we are done
j div_loop      #Otherwise keep looping
```

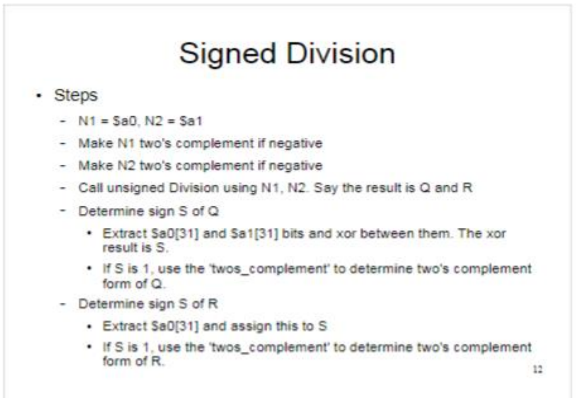


```
div_unsigned_end:
move $v0, $s1 #Moves quotient into $v0
move $v1, $s3 #Moves remainder into $v1
lw $a0, 52($sp) #Restores original $a0 from memory
lw $a1, 48($sp) #Restores original $a1 from memory
li $s6, 31      #Sets constant register $s6 to 31 for MSB extraction
extract_nth_bit($t8, $a0, $s6) #Gets MSB from dividend, stores it in $t8
extract_nth_bit($t9, $a1, $s6) #Gets MSB from divisor, stores it in $t9
xor $t6, $t8, $t9 #Does an xor to see if quotient is positive or not. XOR result
beqz $t6, rem_complement #If the quotient is positive, we don't complement it, go
move $a0, $s1   #Sets $a0 to Q for complement procedure
jal twos_complement    #Procedure call
move $s1, $v0   #Updates quotient, now it is 2's complemented

rem_complement:
beqz $t8, skip_rem #If the remainder is 0, no need to complement
move $a0, $s3   #Sets $a0 to remainder for complement call
jal twos_complement
move $s3, $v0   #Updates remainder to its complemented form

skip_rem:
move $v0, $s1   #Sets $v0 to quotient ($s1)
move $v1, $s3   #Sets $v1 to remainder ($s3)
j div_done #We are done
```

Fig 34. Div_unsigned implementation

As shown in Figure 34, the implementation is relatively simple, with occasional calls to the sub_logical procedure (to subtract the divisor from the dividend) and the left shifts on the remainder and quotient registers. At the end of the division loop, denoted by the label div_unsigned_end, we follow a similar complement computation as seen in the mul_unsigned procedure (refer to Figure 35 if needed). By extracting the most significant bits of the original dividend and divisor, followed by a logical XOR between the two, we can determine whether the result will be negative or positive (Refer to Figure 29 for XOR Truth Table computation).

If the quotient is positive, we do not complement it and instead jump to a similar check on the remainder. After all needed complements have been performed (using the twos_complement utility procedure), we move the quotient (stored in $s1) to register $v0, and the remainder (stored in $s3) to register $v1.



Signed Division

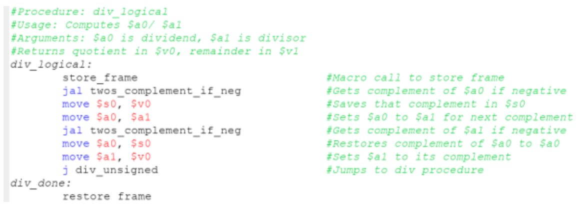
- Steps
  - N1 = $a0, N2 = $a1
  - Make N1 two's complement if negative
  - Make N2 two's complement if negative
  - Call unsigned Division using N1, N2. Say the result is Q and R
  - Determine sign S of Q
    - Extract $a0[31] and $a1[31] bits and xor between them. The xor result is S.
    - If S is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

12

Fig 35. Signed division algorithm

11) div_logical

Once the div_unsigned procedure has been implemented (with complement functionality), the div_logical is able to proceed in a similar fashion as mul_logical:



```
#Procedure: div_logical
#Usage: Computes $a0/ $a1
#Arguments: $a0 is dividend, $a1 is divisor
#Returns quotient in $v0, remainder in $v1
div_logical:
        store_frame                             #Macro call to store frame
        jal twos_complement_if_neg              #Gets complement of $a0 if negative
        move $s0, $v0                           #Saves that complement in $s0
        move $a0, $a1                           #Sets $a0 to $a1 for next complement
        jal twos_complement_if_neg              #Gets complement of $a1 if negative
        move $a0, $s0                           #Restores complement of $a0 to $a0
        move $a1, $v0                           #Sets $a1 to its complement
        j div_unsigned                          #Jumps to div procedure
div_done:
        restore frame
```

Fig 36. Div_logical procedure

We complement both operands ($a0 and $a1) if needed, and then call our div_unsigned procedure. This will perform the division logic as described in Figure 33 and will also complement the quotient and remainder at the end as needed.

D. *Results*

After implementing all of the macros, utility procedures, and logical procedures described, the program should now be fully functional (in design) and ready for testing. In the next section we will explore optimal test results and common errors (with some solutions).

V. TESTING

A. *Ideal Situation after Running the Tests*

After saving all of the files, proceed to open the file "proj-auto-test.asm" in MARS. Click the assemble button as indicated in Figure 36 and run the program by clicking the large green play button to the right of the assemble button.

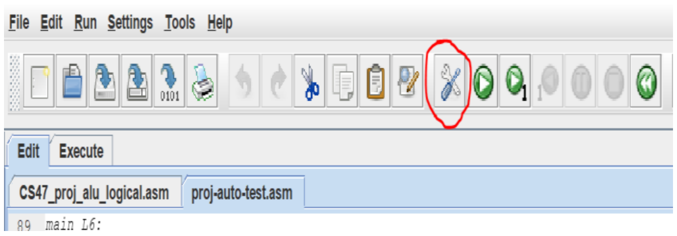
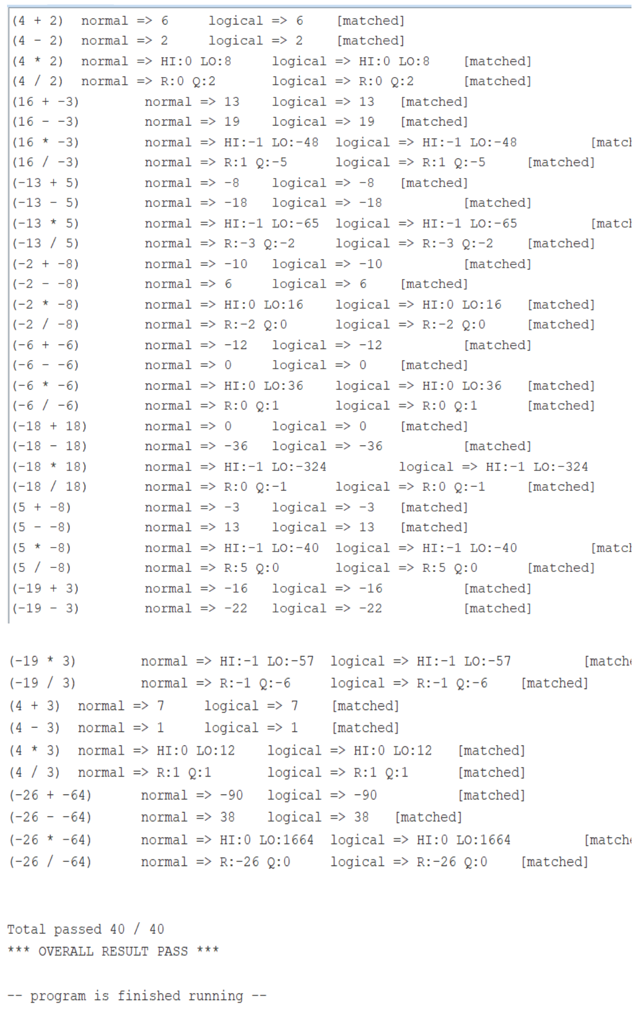


Fig 36. Assembling and running proj-auto-test

If all has been correctly, the MARS screen should display a screen similar to Figure 37, printing "Total passed 40/40" when the program has finished running. What the program is doing is that it is making calls to both your normal procedures and logical procedures, testing the results against one another. If a mismatch is to appear, it is most likely to be as a result of an error in the logical implementation, not the normal implementation.



```
(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         [matc
(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         [matc
(-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
(-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         [matc
(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         [match
(-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         [match
(-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 37. Test Results

B. Common Errors

If the program indicates that not all tests were passed, there is no need to worry. Some common errors and bugs include:

1) A simple typo somewhere in the program.
    a. This can be a mistake such as mistyping the name of a macro or procedure, leading to those instructions never being called
2) The misuse of temporary registers
    a. Temporary registers are not saved across multiple procedure calls and thus can result in something such as an infinite loop if not implemented correctly.
3) Not storing or restoring the frame
    a. As described in the Macros section under "Design and Implementation," the stack frame is extremely important to preserve registers across multiple subroutines in procedure calls. Not storing and restoring the frame correctly can lead to registers being overridden or simply not saved.

If the program is still giving errors, verify that the implementations were followed correctly and refer to the logical flowchart diagrams.

VI. CONCLUSION

Completing this project helped me understand the digital logic and design of the ALU and processor overall, especially through MIPS assembly language. After observing how the algorithms in the flowcharts could be applied to bit values through simple bit manipulation, logical operations, shifts, and loops, I now have a better understanding of how computation is performed in computers, compared to as in humans.

One factor that helped me understand the functionality of arithmetic operations through MIPS Logic better would be that I implemented the logical procedures in two different ways. Originally, I had implemented each logical procedure using very simple loops in combination with logical operations such as XOR's, AND's, and shifts. Although this was relatively simple for addition and subtraction, I began to run into difficulties for the multiplication and division implementations due to the fact these operation results were 64-bits and in two different registers. Therefore, I reimplemented every logical procedure from scratch using the recommended implementations. Although this was both a difficult and time-consuming task, I gained a more comprehensive understanding as a result.

To me, it was simply astounding how through Boolean values (1’s and 0’s), so much could be accomplished. It is both hard and awe-inspiring to imagine that simply through the manipulation of electronic signals, entire computations — faster than any human — could be performed. I am much more appreciative of computers after seeing just how complex something such as addition, subtraction, multiplication, and division could be, and will use my newfound knowledge to remember just how extraordinary Computer Science is.

REFERENCES

[1] D.A. Patterson and J.L Hennessy, Computer Organization and Design (5th edition). Waltham, MA: Morgan Kaufman, 2014, pp. 178 – 195.

[2] K. Patra. CS 47. Class Lecture, Topic: "Addition Subtraction Logic." San Jose State University, San Jose, CA, April 14, 2016.

[3] K. Patra. CS 47. Class Lecture, Topic: "Multiplication Logic." San Jose State University, San Jose, CA, April 19, 2016.

[4] K. Patra. CS 47. Class Lecture, Topic: "Division Logic." San Jose State University, San Jose, CA, April 21, 2016.