Basic Calculator Implementation in MIPS

Jennifer Luu

San Jose State University

jennifer.luu@sjsu.edu

***Abstract*— In this report, there will be explanations on how MARS (MIPS Assembler and Runtime Simulator) can process basic calculator operations such as addition, subtraction, multiplication, and division. The results computed by logical operations in MIPS are equivalent to the ones done by normal operations.**

I. Introduction

The objective of this calculator is to process binary addition, subtraction, multiplication, and division in MIPS assembly language. Boolean algebra will be the foundation for these computations. There are also other objectives that factor into completing the overall goal, such as:

1. Installing the correct software (MARS) and ensuring the right conditions
2. Designing algorithms that mimic arithmetic processes using Boolean algebra
3. Providing in-depth explanations of the algorithms
4. Testing the logical results and comparing them to the normal results for accuracy

11. Requirements

This section will focus on ensuring the appropriate conditions are met before running the calculator in MIPS.

A. Downloading MARS

To start, MARS is the required software to program in MIPS assembly language. Missouri State University offers a free download at: http://courses.missouristate.edu/KenVollmar/mars/

B. Accessing Files

A main zip file will contain all the necessary files. It can be found on Canvas under Professor Patra's class.

Once logged in, go to courses → CS 47 → Assignments → Project 1.

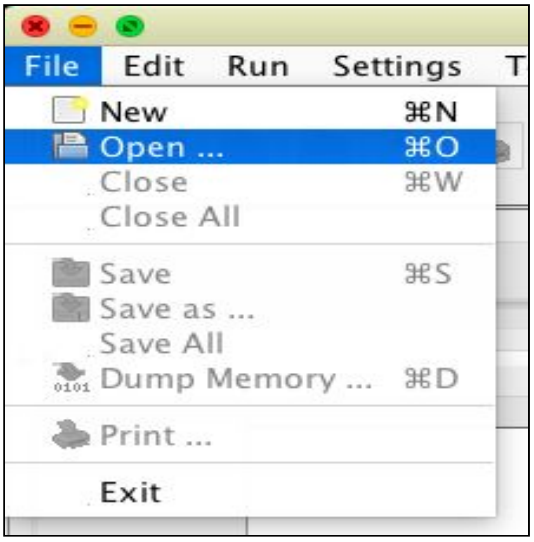
Click "CS47Project1.zip" to download. Then unzip the file. It will yield the following:

1. *cs47_common_macro.asm*
   a. This file has the most frequently used macros.
2. *cs47_proj_macro.asm*
   a. This file is for macros specifically related to the basic calculator implementation.
3. *cs47_proj_procs.asm*
   a. This file has the procedures for printing purposes.
4. *cs47_proj_alu_normal.asm*
   a. This file uses the built-in MIPS arithmetic functions.
5. *cs47_proj_alu_logical.asm*
   a. This file utilizes Boolean logic to compute binary arithmetic.
6. *proj-auto-test.asm*
   a. This file is needed for testing both the normal and logical implementation of the calculator.

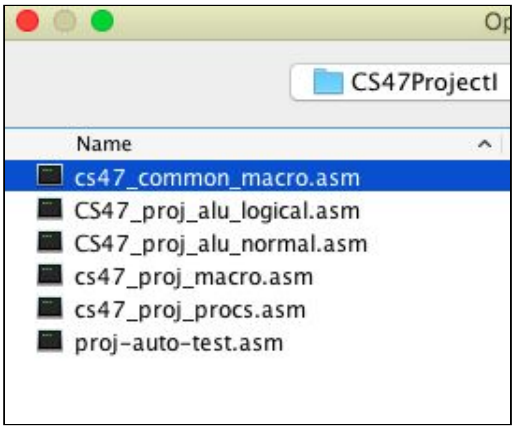
C. Assembly and Execution

Once all the necessary files are obtained, open the MARS application.

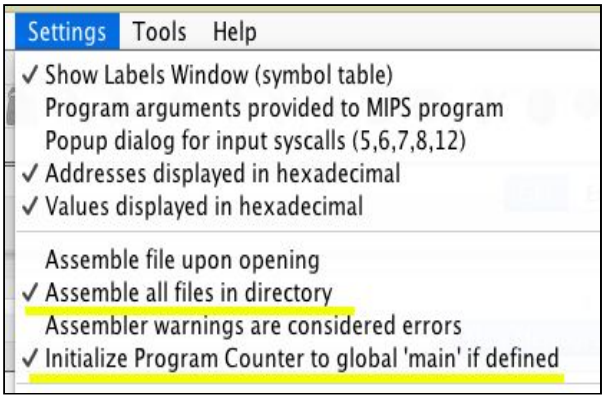
1. Go to “File” and click “Open” in the upper right hand corner.



2. Next, find the unzipped “CS47Project1” file and open all the files mentioned previously to load them into MARS.



3. Finally, go to “Settings.” Click “Assemble all files in directory.” The purpose of this is to link all the individual pieces together since they have macros that call upon each other in different locations. Additionally, click “Initialize Program Counter to global main if defined.” This is to ensure a program begins at the address of a procedure specified at the label “main” instead of at the address at the very beginning.



D. Background Knowledge of Assembly

In MIPS, registers act as storage for input and outputs. The most a register can store is 32 bits. Therefore, if the result of an operation requires more than 32 bits to hold it, then the result must be contained in separate registers. MIPS has registers $Lo to hold the low half (32 bit) and $Hi to hold the upper half (32 bits) of the result.

E. Familiarity with Binary

The binary number represents numbers in two unique digits: 0 and 1. This is also referred to a base-2 number system. It is most commonly used in circuit design, with 0 denoting “off” and 1 denoting “on.”

F. Familiarity with Boolean Algebra

Boolean Algebra utilizes the binary number system to represent specific functions. They may be represented in truth tables and circuits. The following three functions are the foundation for the logical operations in the calculator

1. The OR operation symbol can be represented with a “+” and it is inclusive. This is not to be confused with the common symbol for addition. In the case of two inputs, if at least one of them contains 1, the output will be 1. However, if both inputs contain 0, then the output will also be 0. Meaning, the only way to obtain 1 as an output is to

have the following combinations: 0 + 1, 1 + 0, and 1 + 1. Likewise, the output is 0 when 0 + 0.

a. OR truth table:

Y = A + B		
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

2. The AND operation can be expressed as "." and it is exclusive in how it requires both inputs to be 1 in order for the output to be a 1. In other words, the following combinations will yield 0 as an output: 0.1, 1.0, and 0.0.

a. AND truth table:

Y = A.B		
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

3. The XOR operation can be represented as "⊕" and it stands for an exclusive or. In other words, the output can only be 1 if only one of the inputs are 1. In other words, the inputs cannot be 1 at the same time. The following combinations will yield 1 as an output: 1 ⊕ 0 and 0 ⊕ 1. This operation mimics binary addition and subtraction.

a. XOR truth table

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

III. Design and Implementation

There will be two methods to implement the operations of a basic calculator: normal and logical. The normal method will serve as a comparison for testing purposes. It will utilize already present functions in MIPS for arithmetic processes. The logical method is the primary implementation. It will utilize algorithms based on Boolean algebra and digital logic for arithmetic computations.

A. Normal design

This procedure requires three different registers as its arguments:

1. $a0 - This is the first operand.
2. $a1 - This is the second operation.
3. $a2 - This is the operation symbol.
    a. It is based on traditional addition (+), subtraction (-), multiplication (*), and division (/) symbols.

It will return one output if the operation is addition or subtraction or two outputs if the operation is multiplication or division.

1. $v0
    a. It will return the result of $a0 - $a1.
    b. It will return the Lo 32 bits of $a0*$a1.
    c. It will return the quotient of $a0/$a1.
2. $v1
    a. It will return the Hi 32 bits of $a0*$a.
    b. It will return the remainder of $a0/$a1.

B. Normal implementation

1. To begin, it is important to allocate space for the stack for push or pop operations. Then we create different labels for the respective arithmetic operations based on their symbols.

a.
```
au_normal:
# TBD: Complete it
addi $sp, $sp, -24
sw $fp, 24($sp)
sw $ra, 20($sp)
sw $a0, 16($sp)
sw $a1, 12($sp)
sw $a2, 8($sp)
addi $fp, $sp, 24

beq $a2, '+', add
beq $a2, '-', sub
beq $a2, '*', mul
beq $a2, '/', div
```

2. The branch for addition utilizes temporary register $t0 and $t1, which takes the arguments $a0 and $a1 respectively. The "add" function in MIPS is sufficient and the result is stored in $t2, which will be returned in $v0.

a.
```
add:
move $t0, $a0
move $t1, $a1
add $t2, $t0, $t1
move $v0, $t2

j au_normal_done
```

3. The branch for subtraction is similar to addition. The only difference is that the "sub" function is used.

a.
```
sub:
move $t0, $a0
move $t1, $a1
sub $t2, $t0, $t1
move $v0, $t2

j au_normal_done
```

4. The branch for multiplication will store $a0 and $a1 in $t0 and $t1 respectively. The "mul" function is called and the $Lo will contain the lower 32 bits and the $Hi will contain the upper 32 bits. Both results will be moved to $v0 and $v1 respectively.

a.
```
mul:
move $t0, $a0
move $t1, $a1

mult $t0, $t1
mflo $v0
mfhi $v1

j au_normal_done
```

5. The division branch is similar to multiplication. The difference that the "div" function will be used instead of "mul." The $Lo will store the quotient and the $Hi will store the remainder. Likewise, both results will be moved to $v0 and $v1 respectively.

a.
```
div:
move $t0, $a0
move $t1, $a1

div $t0, $t1
mflo $v0
mfhi $v1

j au_normal_done
```

6. At the end of the normal procedure, it is necessary to restore the frame to free the space in the stack.

a.
```
au_normal_done:
lw $fp, 24($sp)
lw $ra, 20($sp)
lw $a0, 16($sp)
lw $a1, 12($sp)
lw $a2, 8($sp)
addi $sp, $sp, 24

jr      $ra
```

C. Logical design

The logical design of the calculator utilizes the similar thought processing as the normal design. However, it requires many more steps. In the logic implementation, the ability to make decisions for binary operations cannot be summarized in a simple and concise function such as "add" or "mul." Instead, the thought processes must be expanded into multiple,

meticulous steps to accomodate for various situations.

1. *Binary Addition*
    a. In base 10, addition involves two operands. Addition starts at the "1's place," which is the leftmost side of the digits. The sum may reside in its original place or it may be carried to the next position to the right if it exceeds 10. This is also commonly known as "carrying the 1."
    b. As previously mentioned before, the binary number system utilizes two distinct digits, 0 and 1. Addition in base 2 is quite similar to addition in base 10: the same rules apply. As in, 0 + 0 = 0 and 0 + 1 = 1. Carrying over the 1 applies when 1 + 1 because 2 is not a valid number in base 2. Therefore, the next addition operation must take into account of the left over 1. For instance, 0011 + 0001 = 0100.
    c. The XOR operation of the nth digits of the two operands mimics binary addition. The AND operation of the same nth digits obtains the carry value.
    d. Moreover, there must be a case to handle negative integers as well. Unsigned integers have 0 as their most significant bit (MSB), while signed integers have a 1. The remaining bits of the signed integer are the inversion of its complement. For instance, 4 is represented as 0100 and -4 is represented as 1011. To obtain this result, we invert the bits in 4 and add 1.

1. *Binary Subtraction*
    a. The procedure in binary addition applies to binary subtraction. Meaning, subtraction is the addition of an inverted number. For example, 5 - 3 = 5 + (-3).

1. *Binary Multiplication*
    a. Multiplication requires a mask, which is a value that determines what bits to keep or clear in another register. It is used for bitwise operations, which is the act of applying XOR, AND, or OR operations to change the bits in a register.
    b. The procedure takes in two arguments: the multiplicand and the multiplier. The LSB of the multiplier is used to create the mask. The mask will be used in an AND operation with the multiplicand since AND resembles multiplication between 0 and 1. Then the result will be replicated 32 times and stored in the overall product. The multiplier is shifted 1 bit to the right for the next LSB to repeat the process until all the bits of the multiplier have been used.

1. *Binary Division*
    a. Similarly, binary division takes in two arguments: the dividend and the divisor. To start, the divisor is placed under the dividend starting at its MSB. If the number from the MSB of the dividend to the LSB of the LSB of the divisor is greater than the divisor, division cannot happen because subtraction would yield a negative number.
    b. Therefore, if the result is negative then it is crucial to shift the divisor 1 bit to the left to find a

positive sum. Once a positive sum is found, it becomes the remainder and the process is continued until 32 bits have filled the quotient.

D. Logical Implementation

1. Store frame
    a. To start, it is necessary to store the entire frame to allocate space for the registers. Registers $fp, $ra, $a0-$a3, and $s0 - $s7 are stored. Register $sp is implicitly stored. After storing the registers, there are branches specifically designated for the four different operations based on their symbols.

```
au_logical:
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

beq $a2, '+', add_logical
beq $a2, '-', sub_logical
beq $a2, '*', mult_signed
beq $a2, '/', div_signed
```

2. Utility macros

extract_nth_bit

i. The purpose of this macro is to obtain a bit at the nth position of a source register and hold that bit in another register.

ii. This macro has three registers as arguments that contain the destination, source, and bit position.

```
# Put $regS in $s0 for manipulation
# Right shift $s0 by $regT
# Andi 1 to remove and store in $regD
.macro extract_nth_bit($regD, $regS, $regT)
move  $s0, $regS
srlv  $s0, $s0, $regT
andi $regD, $s0, 0x1
.end_macro
```

insert_to_nth_bit

iii. The purpose of this macro is to insert a specific bit from a source register, either 0 or 1, into a designated register

iv. It requires four registers as arguments that contain the: destination, source, bit position, and mask.

```
.macro insert_to_nth_bit($regD, $regS, $regT, $maskReg)
# Preparing $maskReg = 1
# Shifting $maskReg by variable amount in $regS
# Invert $maskReg
# Mask $regD with $maskReg to remove bits
# Shif left $regT by variable amount in $regS
# OR $regD with $regD to add bits
addi $maskReg, $maskReg, 0x1
sllv $maskReg, $maskReg, $regS
not $maskReg, $maskReg
and $regD, $regD, $maskReg
sllv $regT, $regT, $regS
or $regD, $regD, $regT
.end_macro
```

3. Utility procedures

twos_complement

i. This procedure follows the standard algorithm to convert from negative to positive and vice versa:

1. ~$a0 + 1

ii. It takes one argument, $a0, as its input. $a0 is then inverted and 1 is added to gain its complement.

```
twos_complement:
addi $sp, $sp, -28
sw $fp, 28($sp)
sw $ra, 24($sp)
sw $a0, 20($sp)
sw $a1, 16($sp)
sw $a2, 12($sp)
sw $s0, 8($sp)
addi $fp, $sp, 28

not $a0, $a0
li $a1, 0x1

jal add_logical

lw $fp, 28($sp)
lw $ra, 24($sp)
lw $a0, 20($sp)
lw $a1, 16($sp)
lw $a2, 12($sp)
lw $s0, 8($sp)
addi $sp, $sp, 28
jr $ra
```

twos_complement_if_neg

iii. This procedure calls upon the previous one if the integer is a negative number. This time, the result is returned in $v0.

```
twos_complement_if_neg:
bltz $a0, twos_complement

addi $sp, $sp, -28
sw $fp, 28($sp)
sw $ra, 24($sp)
sw $a0, 20($sp)
sw $a1, 16($sp)
sw $a2, 12($sp)
sw $s0, 8($sp)
addi $fp, $sp, 28

move $v0, $a0

lw $fp, 28($sp)
lw $a0, 20($sp)
lw $a1, 16($sp)
lw $a2, 12($sp)
lw $s0, 8($sp)
addi $sp, $sp, 28
jr $ra
```

twos_complement_if_64bit

iv. Similarly, this procedure obtains the two's complement of an integer, expect it is reserved for a 64-bit representation. This is needed for multiplication and division since those operations require registers $Lo and $Hi, which both store 32 bits. The $Lo result will be returned in $v0 and the $Hi in $v1.

v. It takes two arguments, $a0 and $a1, which are the Lo and Hi bits respectively.

vi. To begin, invert both $a0 and $a1. In order to perform ~$a0 + 1, it is necessary to temporarily move $a1 into a separate register and load 1 into $a1. Addition will be performed using another procedure, add_logical, instead of the "add" function in MIPS. Move the result from $v0 to $s2 to temporarily save it.

vii. Move the original value of $a1 from the separate register to $a0 for the next addition step. Similarly, load the carry value from the previous addition operation into $a1. Once again, use add_logical to perform addition. Load the result into a different register, $s3, for temporary storage.

viii. Finally, move $s2 back into $v0. This will be the two's complement of the Lo 32 bits. Likewise, move $s3 back into $v1. This will be the two's complement of the Hi 32 bits.

```
twos_complement_64bit:
addi $sp, $sp, -40
sw $fp, 40($sp)
sw $ra, 36($sp)
sw $a0, 32($sp)
sw $a1, 28($sp)
sw $a2, 24($sp)
sw $s0, 20($sp)
sw $s1, 16($sp)
sw $s2, 12($sp)
sw $s3, 8($sp)
addi $fp, $sp, 40

not $a0, $a0
not $a1, $a1

move $s1, $a1
li $a1, 0x1
jal add_logical

move $s2, $v0

move $a0, $v1
move $a1, $s1

jal add_logical

move $s3, $v0

move $v0, $s2
move $v1, $s3

lw $fp, 40($sp)
lw $ra, 36($sp)
lw $a0, 32($sp)
lw $a1, 28($sp)
lw $a2, 24($sp)
lw $s0, 20($sp)
lw $s1, 16($sp)
lw $s2, 12($sp)
lw $s3, 8($sp)
addi $sp, $sp, 40
jr $ra
```

bit_replicator

ix. The purpose of this procedure is to replicator a given bit 32 times to fill a register. It will be used for multiplication.

x. It takes one argument, $a0, which is the bit to replicated. $a0 will be either 0 or 1.

xi. If $a0 = 0, then $v0 will return the values 0x00000000. If $a1 = 1, then $v0 will return the values $v0 = 0xFFFFFFFF.

```
bit_replicator:
addi $sp, $sp, -28
sw $fp, 28($sp)
sw $ra, 24($sp)
sw $a0, 20($sp)
sw $a1, 16($sp)
sw $a2, 12($sp)
sw $s0, 8($sp)
addi $fp, $sp, 28

beq $a0, $zero, zero_rep
beq $a0, 0x1, one_rep

one_rep:
li $v0, 0XFFFFFFFF
j bit_replicator_done

zero_rep:
li $v0, 0x00000000
j bit_replicator_done

bit_replicator_done:
lw $fp, 28($sp)
lw $ra, 24($sp)
lw $a0, 20($sp)
lw $a1, 16($sp)
lw $a2, 12($sp)
lw $s0, 8($sp)
addi $sp, $sp, 28
jr $ra
```

4. add_sub_logical

This procedure computes 32-bit binary addition. It takes three operands: $a0, $a1, and $a2.

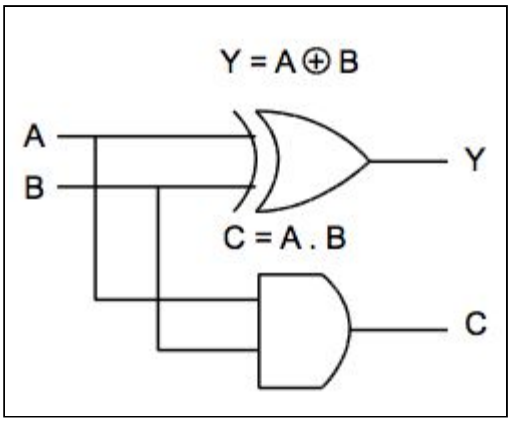
i. $a0 is the first number.

ii. $a1 is the second number.

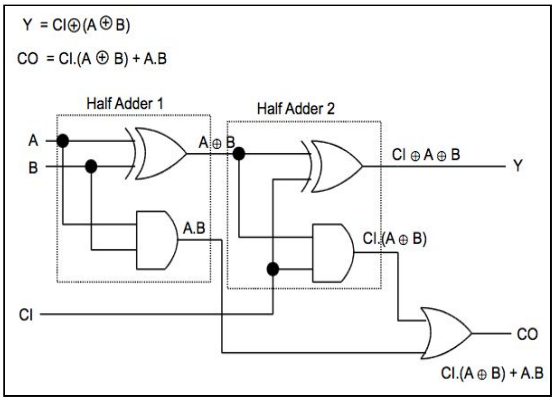
iii. $a2 is the symbol the denotes addition or subtraction.

The sum will be returned in $v0.

This process will begin with the half adder, which is addition that takes 2 input bits only. By performing XOR on the 2 inputs, the sum is obtained. By performing AND on the 2 inputs, the first carry-out bit is obtained.



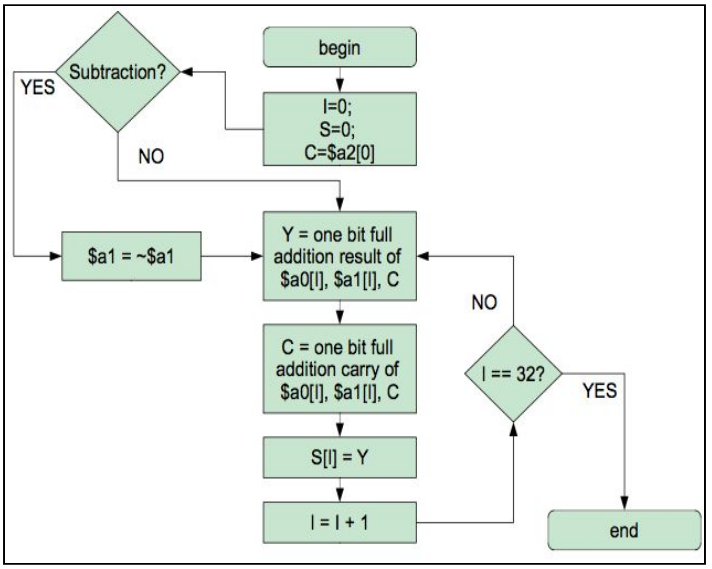
Then, full addition is implemented. The first carry-out bit will become the carry-in bit for the following addition operation.



The new sum is denoted as Y. It is computed by the XOR of the next 2 digits, as well as the XOR with the carry in bit. Here, this operation is essentially adding three bits together.

To compute the new carry-out, there must be the AND of the 2 digits, as well as their XOR. There must also be an AND operation with the carry-in bit and the already XOR bit of the 2 digits. Afterwards, OR must be performed on the result and the AND of the 2 digits.

For the implementation, there is first a branch to determine if $a2 is for addition or subtraction. If $a2 denotes subtraction, then $a1 must be inverted to proceed with addition of a negative number. The half adder is then calculated, which will yield the first bit of the total sum and a carry-out bit. The loop is repeated 32 times to fill the register and the carry-out bit becomes the carry-in bit. The total sum is returned in $v0.



j. The implementation is translated from the diagram and circuits to the following code:

```
add_sub_logical:
addi $sp, $sp, -28
sw $fp, 28($sp)
sw $ra, 24($sp)
sw $a0, 20($sp)
sw $a1, 16($sp)
sw $a2, 12($sp)
sw $s0, 8($sp)
addi $fp, $sp, 28

add $t0, $zero, $zero
extract_nth_bit($t1, $a2, $zero)
add $t2, $zero, $zero

beq $a2, 0xFFFFFFFF, subtraction
j add_sub_main

subtraction:
not $a1, $a1

add_sub_main:
beq $t0, 32, add_sub_exit
extract_nth_bit($t3, $a0, $t0)
extract_nth_bit($t4, $a1, $t0)

xor $t5, $t3, $t4
xor $t6, $t5, $t1

and $t7, $t3, $t4
and $t1, $t1, $t5

or $t1, $t1, $t7
```

```
insert_to_nth_bit($t2, $t0, $t6, $t8)
addi $t0, $t0, 1
j add_sub_main

add_sub_exit:
move $v0, $t2
move $v1, $t1

lw $fp, 28($sp)
lw $a0, 20($sp)
lw $a1, 16($sp)
lw $a2, 12($sp)
lw $s0, 8($sp)
addi $sp, $sp, 28
jr $ra
```

5. add_logical

   This procedure is a branch specifically for addition that is specified if $a2 is equal to 0x00000000. It calls add_sub_logical.

```
add_logical:
addi $sp, $sp, -16
sw $fp, 16 ($sp)
sw $ra, 12($sp)
sw $a2, 8($sp)
addi $fp, $sp, 16

li $a2, 0x00000000
jal add_sub_logical

lw $fp, 16 ($sp)
lw $ra, 12($sp)
lw $a2, 8($sp)
addi $sp, $sp, 16

jr $ra
```

6. sub_logical

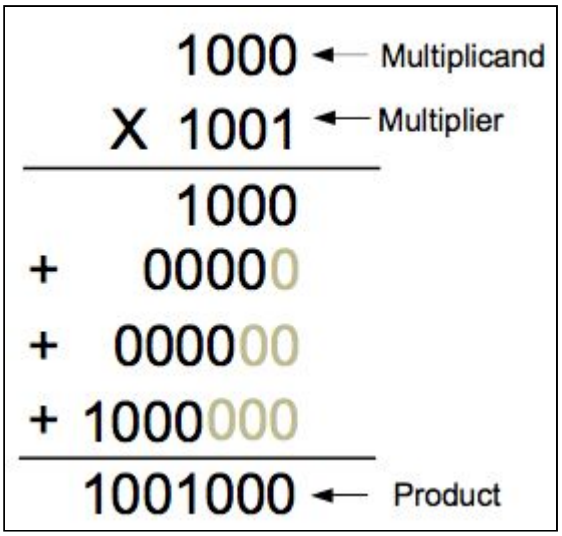
   This procedure is a branch specifically for subtraction that is specified if $a2 equals 0xFFFFFFFF. It calls add_sub_logical as well.

```
sub_logical:
addi $sp, $sp, -16
sw $fp, 16 ($sp)
sw $ra, 12($sp)
sw $a2, 8($sp)
addi $fp, $sp, 16

li $a2, 0xFFFFFFFF
jal add_sub_logical

lw $fp, 16 ($sp)
lw $ra, 12($sp)
lw $a2, 8($sp)
addi $sp, $sp, 16
jr $ra
```

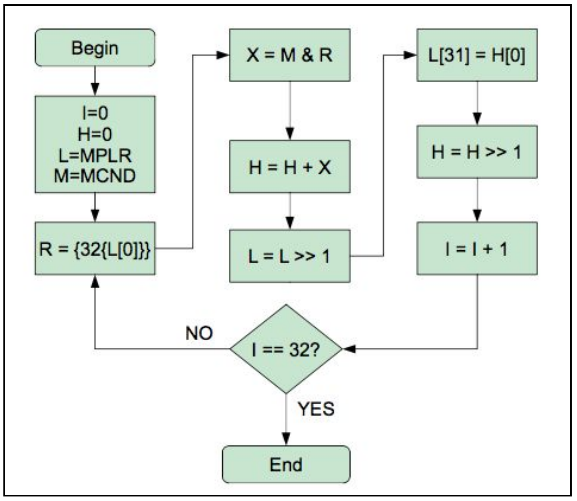
7. mult_unsigned

   Binary multiplication follows the same rules as decimal multiplication. It is a 64-bit operation that takes in two arguments, $a0 and $a1. $a0 is the multiplicand while $a1 is the multiplier.

   $v0 returns the Lo 32 bits, while $v1 returns the Hi 32 bits.



The final product is initialized to 0. The LSB of the multiplier gets AND with the rest of the multiplicand. The result is a partial product, which is added to the final product. The next closest bit to the right of the LSB is now used to AND the multiplicand. Likewise, the partial product is added to the final product along with an extra 0 as a place holder, like in normal multiplication. The process continues until each bit of the multiplier has been used.



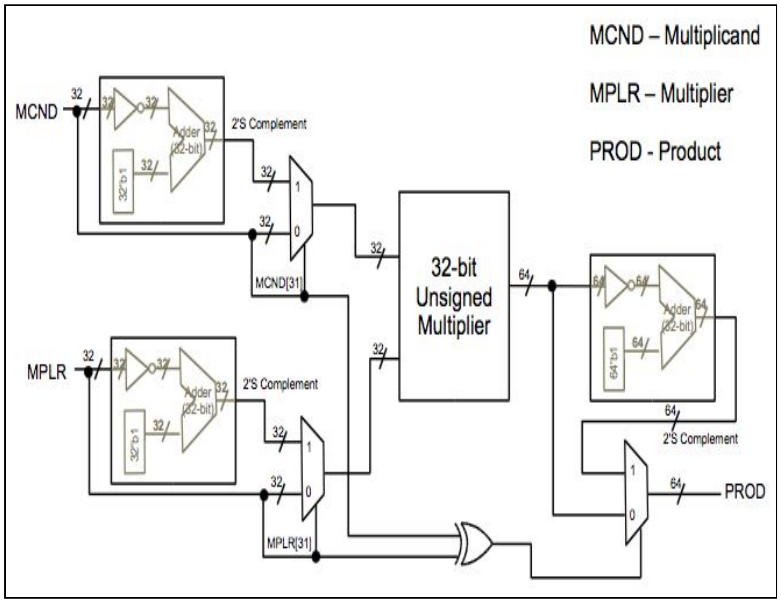
The code implementation of the diagram is as follows:

```
mult_unsigned:
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

#begin
add $s1, $zero, $zero
add $s2, $zero, $zero
move $s3, $a1
move $s4, $a0

mult_unsigned_main:
beq $s1, 32, mult_unsigned_done
extract_nth_bit($a0,$s3,$zero)
jal     bit_replicator
move $s5, $v0

and  $s6, $s4, $s5
```

```
move $a0, $s2
move $a1, $s6

jal add_logical

move $s2, $v0
srl  $s3, $s3, 1

extract_nth_bit($s7, $s2, $zero)
add $t0, $zero, 31

insert_to_nth_bit($s3, $t0, $s7, $t1)

srl $s2, $s2, 1
addi $s1, $s1, 1
j mult_unsigned_main

mult_unsigned_done:
la $v0, ($s3)
la $v1, ($s2)

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

jr $ra
```

8. mul_signed

   This procedure works in the same way mul_unsigned does, expect it handles the case when there are negative integers.



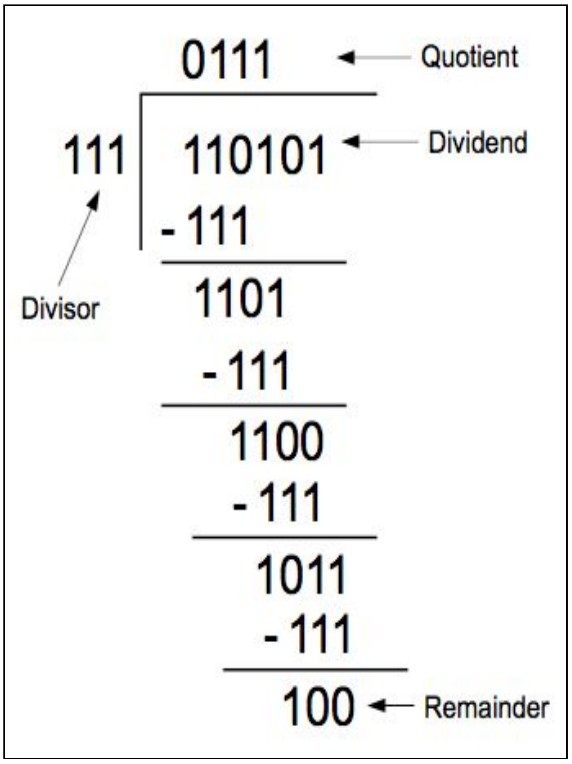
It first obtains the two's complement of the numbers and calls mult_unsigned to get the product of them. The 64-bit product is split in half and each 32-bit part is inverted to obtain their original signs.

```
jal twos_complement_if_neg
move $s3, $v0

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

move $a0, $s3
move $a1, $s4

jal mult_unsigned

move $s5, $v0
move $s6, $v1

add $t0, $zero, 31

extract_nth_bit($t1,$s1,$t0)
extract_nth_bit($t2,$s2,$t0)

xor $t3, $t1, $t2

beq $t3, 0x1, twos_comp_64

j mult_signed_done
```

```
twos_comp_64:
move $a0, $s5
move $a1, $s6
jal twos_complement_64bit

move $s5, $v0
move $s6, $v1

mult_signed_done:
la $v0, ($s5)
la $v1, ($s6)

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
```

9. div_unsigned

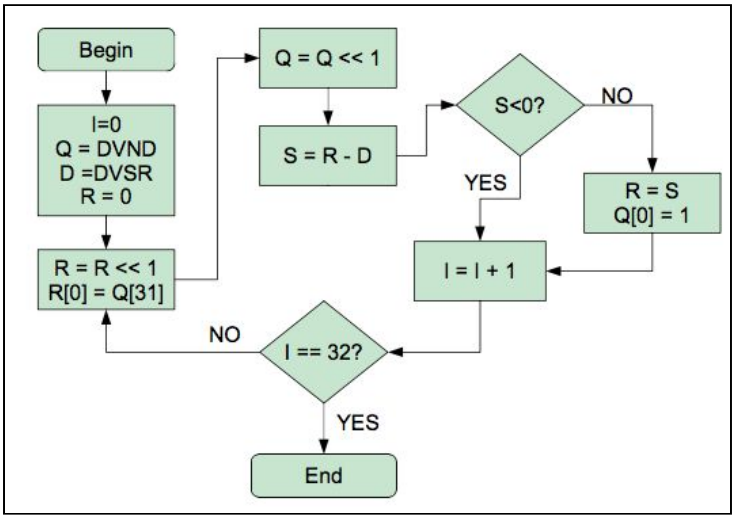
Binary division is a 64-bit operation, like multiplication. The first input is $a0, which is the dividend and the second input is $a1, which is the divisor. Similar to all the previous procedures, binary division also resembles its normal counterpart.



Suppose X is the portion from the the MSB of the dividend to the LSB for the divisor when they are aligned starting at the right. If X < divisor, then the result of X - divisor is negative. This is not a valid answer, so the next step is to place a 0 in the quotient and shift the divisor 1 bit to the left.

X is updated with an extra bit that comes from shifting X to the left as well. Once there is a positive result from X - divisor, then the quotient bit will have a 1. This procedure continues until the divisor can no longer be subtracted from the last remainder.

The quotient is returned in $v0, and the remainder is returned in $v1.



The implementation from the diagram is translated to the following code:

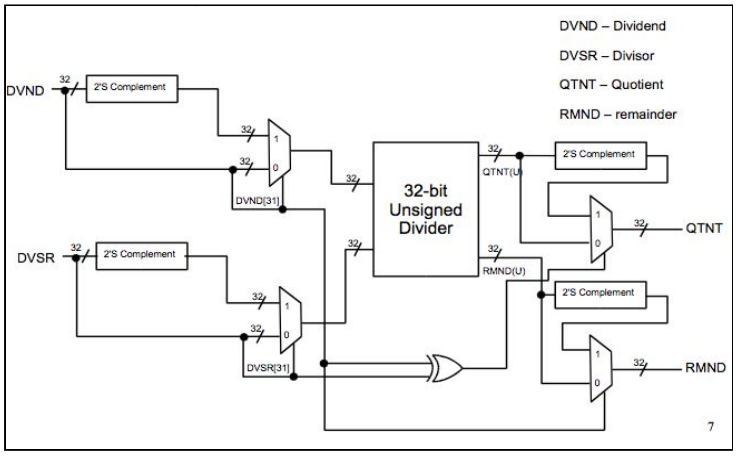
```
div_unsigned:
addi $sp, $sp, -60
sw $fp, 60($sp)
sw $ra, 56($sp)
sw $a0, 52($sp)
sw $a1, 48($sp)
sw $a2, 44($sp)
sw $a3, 40($sp)
sw $s0, 36($sp)
sw $s1, 32($sp)
sw $s2, 28($sp)
sw $s3, 24($sp)
sw $s4, 20($sp)
sw $s5, 16($sp)
sw $s6, 12($sp)
sw $s7, 8($sp)
addi $fp, $sp, 60

#begin
add $s1, $zero, $zero
move $s2, $a0
move $s3, $a1
add $s4, $zero, $zero

div_unsigned_main:
beq $s1, 32, div_unsigned_done
sll $s4, $s4, 0x1
add $s5, $zero, 31
```

```
extract_nth_bit($s6, $s2, $s5)

add $t9, $zero, $zero
insert_to_nth_bit($s4, $zero, $s6, $t9)

sll $s2, $s2, 0x1

move $a0, $s4
move $a1, $s3

jal sub_logical
move $s7, $v0

bltz $s7, sum_less_than
move $s4, $s7
add $t3, $zero, 0x1
add $t4, $zero, $zero
insert_to_nth_bit($s2, $zero, $t3, $t4)


sum_less_than:
addi $s1, $s1, 0x1
jal div_unsigned_main

div_unsigned_done:
move $v0, $s2
move $v1, $s4
```

```
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
```

10. div_signed

This procedure handles the case when there are negative integers for the dividend or divisor.



The two's complement for the signed numbers are taken, and the updated inputs are sent to div_unsigned for division. Once again, the complemented numbers go back to their signed state to maintain equality.

```
div_signed:
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

move $s3, $a0
move $s4, $a1

jal twos_complement_if_neg
move $s1, $v0

move $a0, $a1
```

```
move $a0, $a1

jal twos_complement_if_neg
move $s2, $v0

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

jal div_unsigned

move $s1, $v0
move $s2, $v1

# determine sign S of Q
add $t0, $zero, 31
extract_nth_bit($s5, $s3, $t0)
extract_nth_bit($s6, $s4, $t0)

xor $s7, $s5, $s6

beq $s7, 0x1, twos_comp_q
bne $s7, 0x1, twos_comp_r

twos_comp_q:
move $a0, $s1
jal twos_complement
move $s1, $v0
```

```
j twos_comp_r

# determine sign of R
twos_comp_r:
move $s7, $s5
bne $s7, 0x1, div_signed_done
move $a0, $s2
jal twos_complement
move $s2, $v0
j div_signed_done

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

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

jr  $ra
```

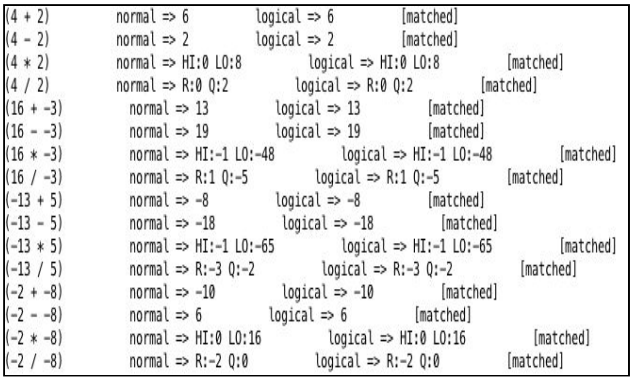
11. Restore frame

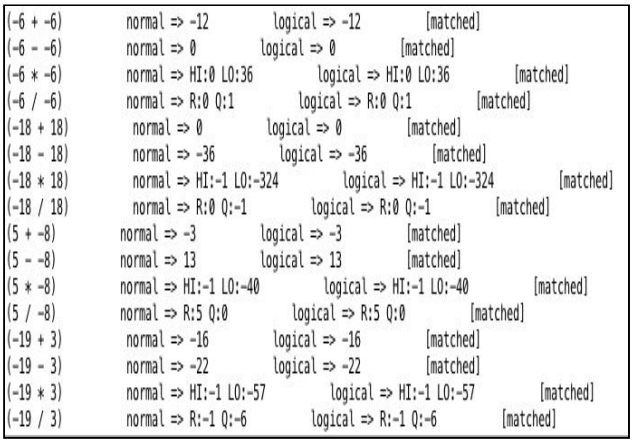
Throughout these operations and procedures, it is necessary to continually store and restore the frame to avoid errors such as runtime exceptions. Though they may be tedious to include, it is needed for the labels and loops to function smoothly.

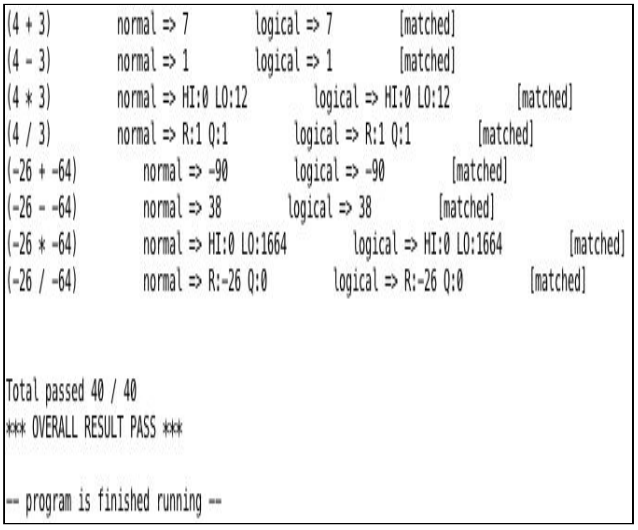
```
lw $fp, 60($sp)
lw $ra, 56($sp)
lw $a0, 52($sp)
lw $a1, 48($sp)
lw $a2, 44($sp)
lw $a3, 40($sp)
lw $s0, 36($sp)
lw $s1, 32($sp)
lw $s2, 28($sp)
lw $s3, 24($sp)
lw $s4, 20($sp)
lw $s5, 16($sp)
lw $s6, 12($sp)
lw $s7, 8($sp)
addi $sp, $sp, 60

jr $ra
```

1V. TESTING

Testing is a crucial aspect in designing any kind of algorithm because it checks for faulty implementations. Therefore, at the final stage the normal results must be compared to the logical results. To do so, assemble all files and then run the program. The output should yield the following:







V. Conclusion

This project required meticulous and tedious planning in order to implement the algorithms correctly. Something as common and seemingly simple as 1+1 had to be broken down into multiple small steps in assembly language. It is easy for people to do multiplication and division in their heads in a few seconds without much thought. However, that is the opposite in assembly language. Omitting even a single step or not fully understanding an algorithm can make the difference between code that works and code that does not. Not only did this demonstrate how computers aren't inherently as "smart" as they are portrayed, but it also emphasized that it is the human who writes these instructions for the computer to follow. Ultimately, it is a person behind a computer who indirectly does these calculations.