Introduction

All mathematical operations can be broken down into logical operations. Essentially, these logical operations serve as the foundation for computers to carry out arithmetic functions. To delve deeper into how logical operations can perform arithmetic functions, this project implements the four basic arithmetic functions: addition, subtraction, multiplication, and division, with primarily logical operations.

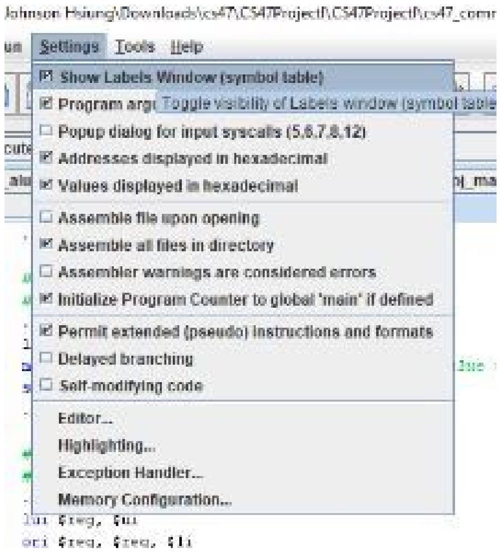
This project uses the assembly language MIPS, or microprocessor without interlocked pipeline states, in the IDE MARS (MIPS assembler and runtime simulator), developed by Missouri State University.

Requirements

Setup

The software MARS is used as an IDE to simulate the runtime environment for MIPS. With the same pseudo-instructions and instruction sets, MARS provides an exposure to the runtime environment for MIPS without any long-term consequences that could occur when programming at the assembly level. Because MARS is written in Java, a version of Java will need to be installed in order to run MARS.

After launching MARS, the files cs47_common_macro.asm, cs47_proj_macro.asm, and cs47-proj-auto_test.asm have to be accessible which can be done with a .include command with the directory. Also, under settings, check "Initialize Program Counter to global 'main' if defined” to have instruction processing begin at the correct program counter.



Knowledge

The implementation of arithmetic functions using logical operations requires understanding of both topics. For arithmetic functions, it is imperative to break down the steps of an arithmetic function into several smaller steps in order to express these smaller steps as logical operations. As for logical operations, true and false is represented by 1 and 0 respectively. Logical operators take inputs of 1’s and 0’s to produce an output. The logical operators are defined by what inputs they take and the corresponding outputs. In this implementation, it is important to understand the outputs of the main operators used to mimic arithmetic functions: AND, OR, and XOR.

The AND operation returns 1 only if both input A and input B are 1. Commonly used to extract bits.

AND Truth Table

Inputs		Output
A	B	Y = A.B
0	0	0
0	1	0
1	0	0
1	1	1

The OR operation returns 1 if A or B is 1. Commonly used to insert bits.

OR Truth Table

Inputs		Output
A	B	Y = A+B
0	0	0
0	1	1
1	0	1

The XOR operation returns 1 if only A is 1 or only B is 1. Commonly used to invert bits.

XOR Truth Table

Inputs		Output
A	B	$Y = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Binary

The base-2 numeral system is called the binary numeral system. Generally, it is expressed in terms of 1's and 0's . Just like the base-10 numeral system that we use, it uses place value. For example, an increase to the number 9 will result in 10. Broken down, the symbol reset to the first value while the position shifted to the left. In base-2, 1 will increase to 10 with 10 expressing the base-10 number 2. It can be seen that each place value shift represents an exponential increase in value. For base-ten, the first place value is 10^0, the second is 10^1, the third is 10^2, etc. Similarly for base-2, the first place value is 2^0, the second is 2^2, the third is 2^3, etc. Because place values can go on infinitely, every number has an equivalent in base-10 and base-2, so values can freely move in between numeral systems with different bases.

The binary system is so commonly used because electronically, information is processed as a series of on and off signals which can be represented as a base-2 numeral system . Additionally, the logical states true and false can also be represented by 1's (true) and 0's (false). The binary system connects how the computer fundamentally works (on and off signals) with numbers in base-10 and logical states.

The highest place value number or most significant bit of a binary number determines if it is a positive or negative number. Zero means positive while one means negative. This system is called 2's complement. The smallest negative number is when only the MSB is 1, and the number increases in value based on its corresponding positive value. For example, the 4-bit numbers 1000 is -16 in base ten while 1001 is -15 because the value increased by one.

Design

Addition

The logical implementation of addition is similar to addition by hand with operations done on one bit at a time. For example, when adding 10 and 23, 0 is added with 3 for the first position and 1 is added with 2 for the second position. Then, there are three values considered for every bit operation: the nth bit from the two numbers and the carryout from the previous operation. These three values produce the sum bit and the carryout for the next operation. The sum bit (the bit of the sum) is calculated using the formula $A \oplus B \oplus CI$ where A and B are the bits from the numbers, CI is the carryout bit from the previous operation, and $\oplus$ is the XOR operation. For addition, the carry bit for the first operation is always zero. The carry out bit is calculated using the formula $CI * (A \oplus B) + A*B$ with * representing the AND operation and + the OR operation. These series of operations are repeated for all 32 bits in the register to produce the sum.

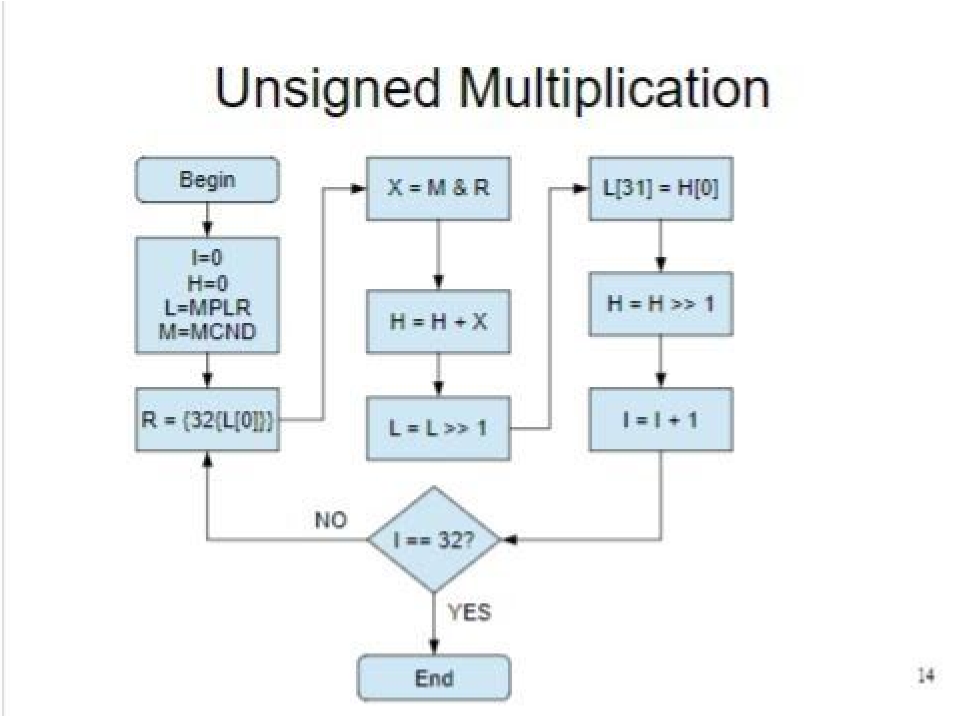
Subtraction

Subtraction works very similarly to addition logically. Essentially, the subtrahend value sign is flipped to mimic the subtraction operation. For example, 5 - 3 is the same as 5 + (-3). This allows for the addition implementation to be reused. To inverse a 2's complement number in binary, simply change all 1's to 0's and vice versa, then add one. This can be done by taking the XOR operation with the number to be inverted and 1 for each bit and then setting the first carry bit to one. Setting the first carry bit to one mimics the add one step.

Multiplication

The logical implementation of multiplication is also similar to multiplying by hand. For each symbol in the multiplier, multiply its value with the multiplicand and add this value to all the previous iterations. Then, increase the place value of the next operation. This serves as the basis for logical multiplication. One different aspect is the mask which is created by replicating the first bit of the current multiplier 32 times. The mask is then AND with the multiplicand and added to the sum. This step replaces multiplying bit by bit. This is how positive numbers are handled.

The algorithm for unsigned multiplication is shown below.



Patra.Kaushik.*Unsigned Multiplication*. 2014. *SJSU.*

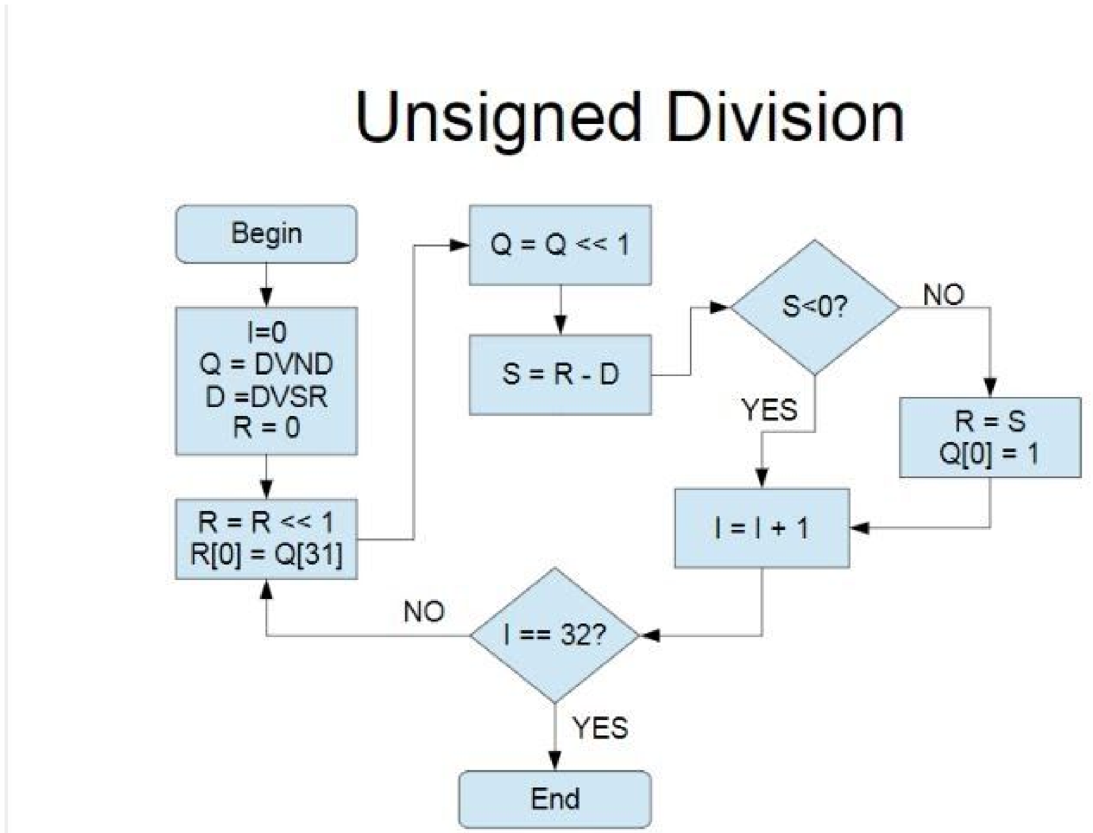
Because MARS has 32 bit registers and multiplication between two 32 bit registers results in a 64 bit value, the end product is split into two registers. H acts as the higher part of the 64 bit register while L is the low.

For negative numbers, it is always converted to its positive form before multiplying. Recall that if the MSB is 1, it is a negative number. Thus, the MSB is checked to see if it is one. The sign of the product is then determined by the signs of the two arguments. If one was negative, the product is converted to its negative value; if both were negative or positive, the product remains positive.

Division

In division, the end result has two parts: the quotient and the remainder. Just like in multiplication, this 64 bit result is split into two registers with the divisor in the upper and the dividend in the lower. Note that the upper and lower eventually becomes the remainder and quotient respectively. The core idea of division is to subtract the divisor from the remainder (initially 0). If the result is negative, add the divisor back and shift the quotient to the left. If the result is positive, shift the quotient to the left and insert 1 to its LSB. This is done for all 32 bits.

The algorithm for unsigned division is shown below.



Patra.Kaushik.*Unsigned Division*. 2014. *SJSU.*

For negative numbers, the quotient follows the same pattern as multiplication: positive if signs match, negative otherwise. And just like multiplication, the 2 arguments are always converted to positive before the computation begins. For the sign of the remainder, it is always the same sign as the dividend.

Utility Macros

Utility macros were implemented to easily reuse code. These macros were mainly used in multiplication and division implementation where extraction and insertion were frequent.

```
.macro extract_bit($tar, $reg, $arg) #goes into $t0
srl $tar, $reg, $arg # shift right $reg by amount $arg
andi $tar, $tar, 1
.end_macro

.macro insert_bit($tar, $ins_tar, $reg, $arg) # uses $t0
sll $t0, $reg, $arg
or $tar, $t0, $ins_tar
.end_macro

.macro replicate_bit($tar, $reg)
andi $t0, $reg, 1
beq $t0, $zero, replicate_zero
li $tar, 0xFFFFFFFF
j end
replicate_zero:
move $tar, $zero
end:
.end_macro
```

extract_bit ($tar, $reg, $arg) - extracts the bit in index $arg of $reg and puts it into $tar.

insert_bit($tar, $ins_tar, $reg, $arg) - inserts the bit in $reg into the $arg index of $ins_tar. Places that result in $tar.

replicate_bit($tar, $reg) - replicates the bit in $reg 32 times to fill up all of $tar with that bit.

Implementation

Most lines of code are commented to clearly state the purpose of each line of code. Thus, the explanation will not walk through each line of code.

Addition

The code for addition is presented below. It carries out the algorithm described in the design section. Some notable lines:

Line 46 and 49: Retrieves the bit based on the iteration of the for-loop because for each index of the for loop i, it handles the i'th bit.

Line 62: This operation decides what each bit of the sum should be.

```
37  start_add:
38          li $t9, 32 # for-loop upper bound
39          li $t8, 0 # loop index
40          li $s1, 0 # initalize first carry bit
41          li $v0, 0 # initalize return register
42
43  for_start_add:
44          beq $t8, $t9, for_end_add
45
46          srlv $t0, $a0, $t8 # shift right a by amount of for loop
47          andi $t0, $t0, 1
48
49          srlv $t1, $a1, $t8 # shift right b by amount of for loop
50          andi $t1, $t1, 1  # now the bit at ith position should be the first value of $t0 and $t1
51
52
53          xor $s0, $t0, $t1 # the bits for summation: a xor b
54          and $s3, $t0, $t1 # the bits for AB
55          xor $s2, $s0, $s1 # for Y: CI xor a xor b
56          and $t0, $s1, $s0 # for carryout: CI(A xor B)
57          or $s1, $t0, $s3 # for carryout: CI(A xor B) + AB
58
59
60          move $t2, $s2 # move Y to $t2
61          sllv $t2, $t2, $t8 # shift it left by for loop index for insertion
62          or $v0, $v0, $t2 # combine current $v0 with shifted value
63          addi $t8, $t8, 1 # for loop counter ++
64          j for_start_add
65
66  for_end_add:
67          j end
68
```

Because multiplication and division will need a logical implementation of addition, this section of the code is replicated in the procedure call plus_logical_procedure with frame store and restoration.

Subtraction

The code for subtraction is shown below. It is very similar to addition except for a few multiple exceptions:

Line 72: The carry bit is initialized to 1 instead of 0. This is to emulate adding 1 to a number when inverting it.

Line 83: The subtrahend bit is XOR with 1, inverting the bit. Another method is to invert all the bits first before entering the for-loop. However, this way was easier as there was no need to load another register.

```
69  start_minus:
70          li $t9, 32 # for-loop upper bound
71          li $t8, 0 # loop index
72          li $s1, 1 # initalize first carry bit
73          li $v0, 0 # initalize return register
74
75  for_start_minus:
76          beq $t8, $t9, for_end_minus
77
78          srlv $t0, $a0, $t8 # shift right a by amount of for loop
79          andi $t0, $t0, 1
80
81          srlv $t1, $a1, $t8 # shift right b by amount of for loop
82          andi $t1, $t1, 1  # now the bit at ith position should be the first value of $t0 and $t1
83          xor $t1, $t1, 1 # xor with 1 to invert the bits
84
85
86          xor $s0, $t0, $t1 # the bits for summation: a xor b
87          and $s3, $t0, $t1 # the bits for AB
88          xor $s2, $s0, $s1 # for Y: CI xor a xor b
89          and $t0, $s1, $s0 # for carryout: CI(A xor B)
90          or $s1, $t0, $s3 # for carryout: CI(A xor B) + AB
91
92
93          move $t2, $s2 # move Y to $t2
94          sllv $t2, $t2, $t8 # shift it left by for loop index for insertion
95          or $v0, $v0, $t2 # combine current $v0 with shifted value
96          addi $t8, $t8, 1 # for loop counter ++
97          j for_start_minus
98
99  for_end_minus:
100
101         j end
102 start_multiply:
```

A logical implementation of subtraction is needed for division, so this section of the code is replicated in minus_logical_procedure with frame store and restoration.

Multiplication

The implementation below follows the basic algorithm for multiplication. Some notable lines:

Line 125-128: This is a recurring block of code anytime addition needs to be called. Here, inverting a number requires 1 to be added.

Line 143: addu calls are commented out because it acted as placeholder for the logical implementation of add. This means the code can be tested without having to worry about the correct implementation of add as a procedure call.

```
108 start_multiply:
109         li $s2, 0 # I = 0
110         li $s3, 0 # H = 0
111         move $s1, $a1 # L = Multiplier
112         move $s0, $a0 # M = Multiplicand
113         li $s7, 32 # for loop
114         move $s6, $zero # for loop
115         extract_bit($t5, $s1, 31) # extract 31st bit of multiplier to check for negativity
116         extract_bit($t6, $s0, 31) # multiplicand
117         beq $t5, 1, invert_multiplier # if multiplier is negative, jump to invert_multiplier
118         beq $t6, 1, invert_multiplicand # if multiplicand is negative, jump to invert_multiplicand
119         j for_start_multiply # nothing needs to be inverted, so jump to for loop
120 
121 invert_multiplier:
122         lui $t2, 0xFFFF
123         ori $t2, $t2, 0xFFFF # t2 is now 0xFFFFFFFF
124         xor $s1, $s1, $t2 # xor with 0xFFFFFFFF will invert all bits in the register
125         move $a0, $s1
126         li $a1, 1
127         jal plus_logical_procedure
128         move $s1, $v0
129         beq $t6, 1, invert_multiplicand # check to see if second number is also negative
130         j for_start_multiply
131 invert_multiplicand:
132         lui $t2, 0xFFFF
133         ori $t2, $t2, 0xFFFF
134         xor $s0, $s0, $t2 # invert bits of multiplicand
135         move $a0, $s0
136         li $a1, 1
137         jal plus_logical_procedure
138         move $s0, $v0
139 for_start_multiply:
140         beq $s7, $s6, for_end_multiply # for loop condition
141         replicate_bit($s4, $s1) # replicate the bit
142         and $s5, $s0, $s4 # X = M $ R
143         #addu $s3, $s3, $s5 # H = H + X
144         move $a0, $s3
145         move $a1, $s5
146         jal plus_logical_procedure
147         move $s3, $v0
```

```
        srl $s1, $s1, 1 # L = L >> 1
        extract_bit($t1, $s3, 0) # H[0]
        insert_bit($s1, $s1, $t1, 31) # L[31] = H[0]
        srl $s3, $s3, 1 # H = H >> 1
        addi $s6, $s6, 1 # index++
        j for_start_multiply
for_end_multiply:
        xor  $t7, $t5, $t6 # $t5 and $t6 was the information about
                           # the negativity of the multiplicand and multiplier.
                           # XOR checks to see if product should be negative.
        beq  $t7, 0, positive_product # If XOR resulted in 0, it should be positive.
        xor $s1, $s1, $t2 # inverts Low
        xor $s3, $s3, $t2 # inverts Hi
        move $a0, $s1
        li $a1, 1
        jal plus_logical_procedure
        move $s1, $v0
        #addi $s1, $s1, 1 # adds one
positive_product:
        move $v0, $s1 # transfer value to return
        move $v1, $s3
        j end
```

Division

The implementation below follows the basic algorithm for division. Like multiplication, it converts its arguments to positive first.

```
start_divide:
        li $s7, 32 # for loop
        move $s6, $zero # for loop
        move $s0, $a0 # $s0 = Q = DVND
        move $s1, $a1 # $s1 = D = DVSR
        move $s2, $zero # initialize R = 0

        extract_bit($t5, $s0, 31) # extract 31st bit of DVND to check for negativity
        extract_bit($t6, $s1, 31) # DVSR
        beq $t5, 1, invert_DVND # if DVND is negative, jump to invert_DVND
        beq $t6, 1, invert_DVSR # if DVSR is negative, jump to invert_DVSR
        j for_start_divide # nothing needs to be inverted, so jump to for loop

invert_DVND:
        lui $t2, 0xFFFF
        ori $t2, $t2, 0xFFFF # t2 is now 0xFFFFFFFF
        xor $s0, $s0, $t2 # xor with 0xFFFFFFFF will invert all bits in the register
        #addi $s0, $s0, 1 # add 1
        move $a0, $s0
        li $a1, 1
        jal plus_logical_procedure
        move $s0, $v0
        beq $t6, 1, invert_DVSR # check to see if second number is also negative
        j for_start_divide
invert_DVSR:
        lui $t2, 0xFFFF
        ori $t2, $t2, 0xFFFF
        xor $s1, $s1, $t2 # invert bits of multiplicand
        #addi $s1, $s1, 1 # add 1
        move $a0, $s1
        li $a1, 1
        jal plus_logical_procedure
        move $s1, $v0
```

```
for_start_divide:
        beq $s7, $s6, for_end_divide
        sll $s2, $s2, 1 # R = R << 1
        extract_bit($t1, $s0, 31) # $t1 = Q[31]
        insert_bit($s2, $s2, $t1, 0) # R[0] = Q[31]
        sll $s0, $s0, 1 # Q = Q << 1
        move $a0, $s2 # load these registers to call logical_minus
        move $a1, $s1
        jal minus_logical_procedure # S = R - D
        move $s3, $v0 # this procedure puts return value into $v0, so extract it
        bltz $s3, for_increase_divide # if S < 0, increase index
        move $s2, $s3 # R = S
        ori $s0, $s0, 1 # Q[0] = 1
for_increase_divide:
        addi $s6, $s6, 1
        j for_start_divide
for_end_divide:
        xor  $t7, $t5, $t6 # $t5 and $t6 was the information about
                           # the negativity of the divisor and dividend.
                           # XOR checks to see if quotient should be negative.
        beq  $t7, 0, positive_quotient # If XOR resulted in 0, it should be positive.
        lui $t2, 0xFFFF # $t2 was used before, so it needs to be reloaded
        ori $t2, $t2, 0xFFFF # t2 is now 0xFFFFFFFF
        xor $s0, $s0, $t2 # inverts Low
        #addi $s0, $s0, 1 # adds one
        move $a0, $s0
        li $a1, 1
        jal plus_logical_procedure
        move $s0, $v0
positive_quotient:
        beqz $t5, positive_remainder # checks to see if dividend was positive
        lui $t2, 0xFFFF # dividend was not positive so convert remainder to negative
        ori $t2, $t2, 0xFFFF # t2 is now 0xFFFFFFFF
        xor $s2, $s2, $t2
        #addi $s2, $s2, 1
        move $a0, $s2
        li $a1, 1
        jal plus_logical_procedure
        move $s2, $v0
positive_remainder:
        move $v0, $s0
        move $v1, $s2
        j end
end:
```

Testing

Au_normal

Au_normal branches into the 4 basic arithmetic operations. The arguments are loaded into $a0, $a1, and $a2, and the results are returned into $v0 and $v1. Executes when $a2 contains “+.”

Start_add and start_minus

Simply calls the basic MIPS arithmetic functions for add and subtract and returns into $v0. Executes when $a2 contains “+” and “-” respectively.

Start_multiply

Calls the MIPS function mul. Then, copies the result of the HI register into $v1. Executes when $a2 contains “*.”

Start_divide

Calls the MIPS function div. Copies the quotient from lo to $v0 and the remainder from hi to $v1. Executes when $a2 contains “/.”

Au_logical

Au_logical is the logical implementation of the 4 arithmetic operations with the same arguments and return values as Au_normal. The details of how each operation is done can be found in the Implementation section.

Proj-auto-test

An assembly program to test the logical implementations of the 4 arithmetic operations. It compares the results of Au_normal and Au_logical to see if the logical implementations are correct. It tests 10 sets of values with each operation, resulting in 40 total tests. Below is the output for the program.

```
(4 + 2)         normal => 6     logical => 6    [matched]
(4 - 2)         normal => 2     logical => 2    [matched]
(4 * 2)         normal => HI:0 LO:8     logical => HI:0 LO:8    [matched]
(4 / 2)         normal => R:0 Q:2       logical => R:0 Q:2      [matched]
(16 + -3)       normal => 13    logical => 13   [matched]
(16 - -3)       normal => 19    logical => 19   [matched]
(16 * -3)       normal => HI:-1 LO:-48          logical => HI:-1 LO:-48         [matched]
(16 / -3)       normal => R:1 Q:-5      logical => R:1 Q:-5     [matched]
(-13 + 5)       normal => -8    logical => -8   [matched]
(-13 - 5)       normal => -18   logical => -18          [matched]
(-13 * 5)       normal => HI:-1 LO:-65          logical => HI:-1 LO:-33         [not matched]
(-13 / 5)       normal => R:-3 Q:-2     logical => R:-3 Q:-2    [matched]
(-2 + -8)       normal => -10   logical => -10          [matched]
(-2 - -8)       normal => 6     logical => 6    [matched]
(-2 * -8)       normal => HI:0 LO:16    logical => HI:0 LO:16   [matched]
(-2 / -8)       normal => R:-2 Q:0      logical => R:-2 Q:0     [matched]
(-6 + -6)       normal => -12   logical => -12          [matched]
(-6 - -6)       normal => 0     logical => 0    [matched]
(-6 * -6)       normal => HI:0 LO:36    logical => HI:0 LO:4    [not matched]
(-6 / -6)       normal => R:0 Q:1       logical => R:0 Q:1      [matched]
(-18 + 18)      normal => 0     logical => 0    [matched]
(-18 - 18)      normal => -36   logical => -36          [matched]
(-18 * 18)      normal => HI:-1 LO:-324         logical => HI:-1 LO:-324        [matched]
(-18 / 18)      normal => R:0 Q:-1      logical => R:0 Q:-1     [matched]
(5 + -8)        normal => -3    logical => -3   [matched]
(5 - -8)        normal => 13    logical => 13   [matched]
(5 * -8)        normal => HI:-1 LO:-40          logical => HI:-1 LO:-40         [matched]
(5 / -8)        normal => R:5 Q:0       logical => R:5 Q:0      [matched]
(-19 + 3)       normal => -16   logical => -16          [matched]
(-19 - 3)       normal => -22   logical => -22          [matched]
(-19 * 3)       normal => HI:-1 LO:-57          logical => HI:-1 LO:-49         [not matched]
(-19 / 3)       normal => R:-1 Q:-6     logical => R:-1 Q:-6    [matched]
(4 + 3)         normal => 7     logical => 7    [matched]
(4 - 3)         normal => 1     logical => 1    [matched]
(4 * 3)         normal => HI:0 LO:12    logical => HI:0 LO:12   [matched]
(4 / 3)         normal => R:1 Q:1       logical => R:1 Q:1      [matched]
(-26 + -64)     normal => -90   logical => -90          [matched]
(-26 - -64)     normal => 38    logical => 38   [matched]
(-26 * -64)     normal => HI:0 LO:1664          logical => HI:0 LO:1664         [matched]
(-26 / -64)     normal => R:-26 Q:0     logical => R:-26 Q:0    [matched]
```

```
Total passed 37 / 40
*** OVERALL RESULT FAILED ***
```

Mismatches

The mismatches come from a bug in logical multiplication. It appeared when the plus_logical_procedure replaced every instance of the basic instruction add, specifically in the for loop.

```
#addu $s3, $s3, $s5 # H = H + X
move $a0, $s3
move $a1, $s5
jal plus_logical_procedure
move $s3, $v0
```

Although the procedure call to plus_logical procedure works for every other instance, it does not work sometimes inside this for-loop. If the addu call replaced the call to plus_logical_procedure at these lines, multiplication would work as intended with 40/40 results passed. This means the error comes from one of these lines. However, I could not figure out how or why this bug occured. I tried looking for patterns in the mismatches and checking all the frame stores and restorations, but did not see any patterns or problems. Plus_logical_procedure is the exact same implementation as the plus implementation in au_logical which works fine. Because of this, I had no idea how to approach this bug.

Conclusion

This project explored the usage of logical operations in arithmetic functions. It revealed how digital circuits can be manipulated to mimic arithmetic functions. Using MIPS assembly language in MARS, logical implementations of the four basic arithmetic functions were programmed. The program compared the results of these implementations with the MIPS arithmetic instructions to verify its accuracy. As floating point representation is the next topic in CS 47, the next step could be to add decimal numbers to the functionality of this program.

Working on this project helped me appreciate all the work of amazing people. It is because of people like them that 1 + 1 is as easy as typing 3 inputs on the keyboard. I cannot even imagine how beautifully complex things like monitors, projectors, keyboards, planes, etc. must be. I learned a lot through this project. It was like playing a game at the beginning: I had to be constantly aware of which registers are being used and what each register represents through each progression of my implementation. Each time I ran the tester, I held my breath, praying for results to still be somewhat correct. However, as I worked more and more on the project, keeping track of registers got easier and testing the code was less nerve wracking. I developed a methodical approach to errors: try to pinpoint the cause of the error and constantly analyze implemented code. Although a bug remains, I am happy I finished this project and improved my problem solving.