CS47 Project

Bitwise Calculations: Addition, Subtraction, Multiplication, and Division

Ryan Tran
College of Science: Computer Science
San José State University
San José, United States
tranryanp@gmail.com

***Abstract*—This report explains the usage of the MIPS processing architecture to implement addition, subtraction, multiplication, and division. It is implemented through given MIPS operands and through logical, bitwise operands.**

I. Introduction

This project is the final assignment of CS47. The objective is to create an ALU to calculate results for addition, subtraction, multiplication, and division through two methods: normal MIPS instructions and logical MIPS instructions. MIPS is a processing architecture that is being simulated in MARS. In doing both normal and logical ways, students will not only understand how systems compute these everyday calculations, students will also be challenged with implementing the algorithm themselves. This report contains the requirements, designs, and test results of such tasks.

II. Normal vs Logical

Before we can delve into each operation, normal and logical need to first be explained. The difference between normal and logical is the type of MIPS operations used. Operations that calculate the results on its own can be considered normal. Examples of these are 'add,' 'sub,' 'mult,' and 'div.' However, logical operations are what these example operations are doing under the surface. Using lots of bitwise operations to compute the same result, logical is essentially just normal but with every single step along the way.

III. Storing Frames

Storing frames, or storing registers, is vital for the algorithm to work. Registers in use may be overwritten and it will disrupt the process. By storing these used registers, it allows the program to run without problems. For my implementation, I stored '$a0 - $a2,' '$s0 - $s2,', and '$t0-$t9,' just to be safe. The storing takes place at the beginning of the file. At the end, these same registers are then loaded back to their respective registers.

IV. Determining Which Operation

For the algorithm to perform the proper operation, it needs to be able to determine which operation needs to be performed. Right after storing the frames, 'beq' is used to check if '$a2," the argument containing which operation must be performed, is equal to either a plus sign, minus sign, asterisk, or forward slash. This decision code is known as a multiplexer.

```
beq     $a2, '+', addition
beq     $a2, '-', subtract
beq     $a2, '*', multiply
beq     $a2, '/', divide
j end
```

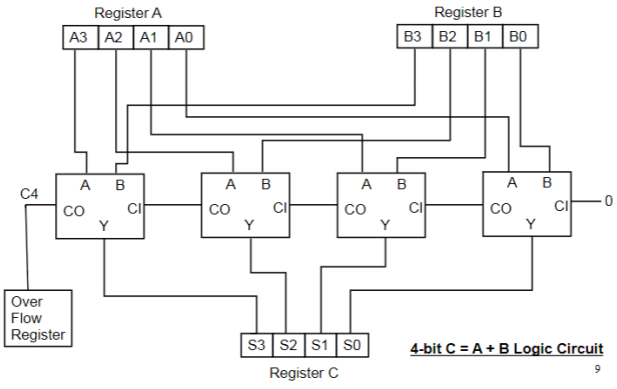
The 'j end' line of code at the end makes sure that if '$a2' is not equal to any of the signs, it skips to the end instead of moving to the code below, which is addition in my file.

V. Addition

Addition is the first operation that must be implemented. Becoming vital in later operations, it is necessary to implement addition first.

A. *Requirement*

Execute addition through normal and logical MIPS instructions. Implement logical addition through Binary Ripple Carry adder [1].



B. *Normal*

Normal addition is done with the normal MIPS operations: 'add,' 'addi,' 'addu,' or 'addiu.' For the sake of simplicity for the normal ALU, I chose to use 'add.'

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

By using ‘add’ to store the results of adding the first value and second value, this is the simplest way to complete the task.

C. *Logical*

Since the code is much longer than the normal version, I will be pasting it instead of screenshotting it. The same will be done for all other logical operations.

```
addition:
    ori     $t0, $zero, 1       # $t0 is counter
    ori     $t4, $zero, 0       # $t4 is Cin
    ori     $t5, $zero, 0       # $t5 is sum

addition_loop:
    and     $t1, $s0, $t0       # $t1 is $t0th bit of $s0
    and     $t2, $s1, $t0       # $t2 is $t0th bit of $s1

    xor     $t3, $t1, $t2       # A xor B | $t3 is sum of
                                $t0th bits
    xor     $t3, $t3, $t4       # (A xor B) xor Cin | adds
                                Cin to $t3
    or      $t5, $t5, $t3       # $t5 is current sum

    and     $t7, $t1, $t2       # A.B
    xor     $t8, $t1, $t2       # A xor B
    and     $t9, $t4, $t8       # Cin.(A xor B)
    or      $t4, $t9, $t7       # Cin.(A xor B) + A.B |
                                $t4 is Cout

    sll     $t4, $t4, 1         # shift $t4 left once
    sll     $t0, $t0, 1         # shift $t0 left once
    beqz    $t0, add_end        # if $t0 is 0, jump to
                                add_end

    j addition_loop

add_end:
    ori     $v0, $t5, 0
    ori     $v1, $t4, 0
    j end
```

To break down the code, it starts off by setting a counter that works as the bit location. It shifts left by increments of one to access each bit. CarryIn is denoted as Cin and tracks the number to be added on top. Lastly, the sum is set at zero.

Running in a loop until the counter becomes zero, it starts by grabbing a bit from the first value and a bit from the second value through ‘and’. The bit that is accessed is dependent on the counter. The two bits are then added together along with Cin with ‘xor’ and stored in sum. The new Cin is calculated and denoted as Cout, or CarryOut. The counter as well as Cout are then shifted left once. Before looping, ‘beqz’ checks if counter is equal to zero. If it is, addition is complete and it moves to its final steps.

Instead of following the diagram exactly and adding bit by bit, I decided to instead iterate through all the bits one at a time and shift left to supplement this. It is the same premise, but with less lines of instructions as I do not need to shift left to access a bit and then shift back right to have it be in the least significant bit (LSB) position, only to shift it left once more to add it to the final sum.

At the end, the sum is then moved to its respective return register and the remaining Cin is stored in the unused return register. This remaining Cin will be used in later operations, but is not necessary to store for addition on its own.

D. *Macro*

Creating a macro makes using addition much easier for later operations. The simple macro is as shown:

```
.macro  add_logical($num, $ber)
or      $a0, $zero, $num
or      $a1, $zero, $ber
ori     $a2, $zero, '+'
jal au_logical
.end_macro
```

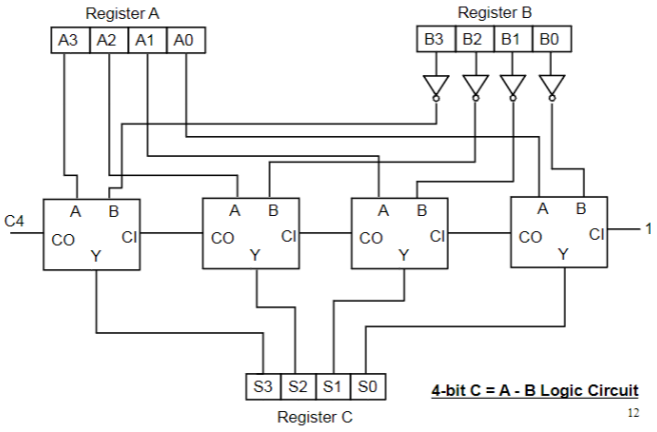
This macro simply changes the arguments and then ‘jal’s back into the file to do the necessary algorithm. It is important to note that registers in use must be stored otherwise using the macro will result in registers being overwritten and losing data.

VI. Subtraction

Subtraction is the second operation that must be implemented. It is the first operation to reuse the logical addition algorithm.

A. *Requirement*

Execute subtraction through normal and logical MIPS instructions. Implement logical subtraction by reusing the logical addition through Binary Ripple Carry Subtractor [1].



B. *Normal*

Normal subtraction is done with the normal MIPS operations: ‘sub,’ ‘subi,’ ‘subu,’ or ‘subiu,’ similar to addition. Again, I chose to use the simplest operation: ‘sub.’

```
subtract:
        sub     $v0, $a0, $a1
        j end
```

Similar to normal addition, by using ‘sub’ to store the results of subtracting the first value and second value, this is the simplest way to complete the task.

C. *Logical*

subtract:

```
    not     $s1, $s1

    ori     $t0, $zero, 1       # $t0 is counter
    ori     $t4, $zero, 1       # $t4 is Cin
    ori     $t5, $zero, 0       # $t5 is sum

    j addition_loop
```

In class, it was taught that to subtracting a number was equal to adding said number, but in 2’s complement form. The pattern for 2’s complement is to invert all the bits and to add one. We can take advantage of this pattern by reusing the logical addition algorithm. Firstly, the value that needs to be subtracted is inverted with ‘not.’ After that, addition is prepped by creating a counter and sum. However, the Cin is now one instead of zero because it is necessary to add one to make an inverted number into 2’s complement. Once those values are set up, the code jumps into the addition loop and the rest carries out the same as logical addition.

D. *Macro*

Similar to addition, creating a macro for subtraction simplifies its later usage.

```
.macro  sub_logical($num, $ber)
or      $a0, $zero, $num
or      $a1, $zero, $ber
ori     $a2, $zero, '-'
jal au_logical
.end_macro
```

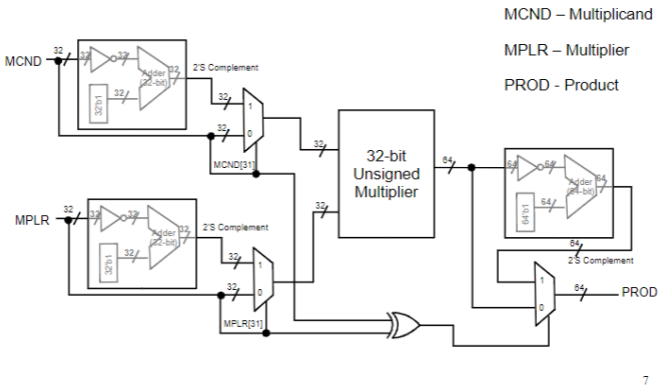
Operating the same as the addition macro, all it does is change the arguments and then ‘jal’s into the algorithm. This same macro can also calculate a single value’s two’s complement if the first argument is set to the register ‘$zero.’

VII. Multiplication

Multiplication is the third operation that must be implemented. To complete the logical multiplication algorithm, at least in my implementation, it was necessary to use subtraction. By extension, this means that addition is also necessary.

A. *Requirement*

Execute multiplication through normal and logical MIPS instructions. Implement logical multiplication through a Signed Multiplication Circuit [2].



What this circuit does compared to an unsigned multiplier is check for negative values, two’s complements them and keeps track of the negatives. After that, it goes through regular unsigned multiplication and results with using the tracked negatives to either keep the product the same or two’s complement it.

B. *Normal*

Normal multiplication is done with the normal MIPS operations: ‘mul,’ ‘mult,’ ‘mulo,’ ‘mulu,’ or ‘mulou.’ Though any of them can be used with about the same amount of lines of code, I decided to use ‘mult’ as I did not want unsigned multiplication and, in case of this project, I did have to worry too much about overflow. Since results for ‘mult’ are stored in hi and lo registers, it is important to use mfhi and mflo to move the results.

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

By using ‘mult,’ the upper 32 bits are stored in hi and the lower 32 bits are stored in lo. The values from hi and lo are then moved to their respective return variables.

C. Logical

I was unsure whether or not to past the entirety of the long code into the report, but better safe than sorry.

```
multiply:
    ori     $t0, $zero, 0x80000000    # set $t0 to
                                      1000...0000
    and     $t1, $s0, $t0             # first bit of $s0
    and     $t2, $s1, $t0             # first bit of $s1

    beqz    $t1, first_positive       # if $t1 = 0, goto
                                      first_positive
    sub_logical($zero, $s0)           # else
                                  twos_complement $s0
    ori     $s0, $v0, 0               # store value back
                                      into $s0

first_positive:
    beqz    $t2, next         # if $t2 = 0, goto
                              complement_second
    sub_logical($zero, $s1)       # else
                              twos_complement $s1
    ori     $s1, $v0, 0       # store value back into $s1



next:
    ori     $t9, $zero, 1             # set $t9 to 1
                          (placeholder/signs are opposite)
    beq     $t1, $t2, sign_equal      # if $t1 = $t2, goto
                                      sign_equal
    j multiply_start

sign_equal:
    ori     $t9, $zero, 0     # else set $t9 to 0 (both +)

multiply_start:
    ori     $t0, $zero, 1     # $t0 is the bit location
    ori     $t1, $zero, 0     # $t1 is shamt
    ori     $t5, $zero, 0     # $t5 is lo
    ori     $t6, $zero, 0     # $t6 is hi

multiply_loop:
    and     $t3, $s0, $t0     # $t3 is $t0th bit of $s1
    beqz    $t3, skip         # if $t3 = 0, goto skip

    sllv    $t4, $s1, $t1     # shift $s1 by shamt
    add_logical($t5, $t4)     # add shifted value to lo
    ori     $t5, $v0, 0       # store new value in lo

    add_logical($t6, $v1)     # add Cin to hi
    ori     $t6, $v0, 0       # store new value in hi
    sub_logical(31, $t1)      # subtract 31 by shamt
    srlv    $t3, $s1, $v0     # shift right to get hi value
    srl     $t3, $t3, 1       # shift right once more
                              because shifting by
                              32 - shamt doesn't work?
    add_logical($t6, $t3)     # add value to hi
    ori     $t6, $v0, 0       # store new value in hi

skip:
    sll     $t0, $t0, 1       # shift bit location left once
    add_logical($t1, 1)       # add 1 to shamt
    or      $t1, $zero, $v0   # store new shamt

    beqz    $t0, multiply_sign_check  # if bit location is
                                      0, goto
                                  multiply_sign_check
    j multiply_loop

multiply_sign_check:
```

```
beqz    $t9, multiply_end     # if signs are equal, goto
                                multiply_end
sub_logical($zero, $t5)       # else twos_complement lo
ori     $t5, $v0, 0           # store new value in lo
not     $t6, $t6              # inverse hi
add_logical($t6, $v1)         # add cout of lo to hi
ori     $t6, $v0, 0           # store new value in hi


multiply_end:
ori     $v0, $t5, 0                 # store lo in $v0
ori     $v1, $t6, 0                 # store hi in $v1
j end
```

To be brief about it, the first four labels deals with negative values. It keeps track of negative values and also two's complements values if they are negative.

At the start of the multiplication process, a bit location, shift amount, lo, and hi registers are initialized. Though different from the method taught in class, this multiplication code still follows the same train of thought, checking values and adding to lo and hi when needed. The 'skip' label is used for when a value is zero and thus can skip several lines of code.

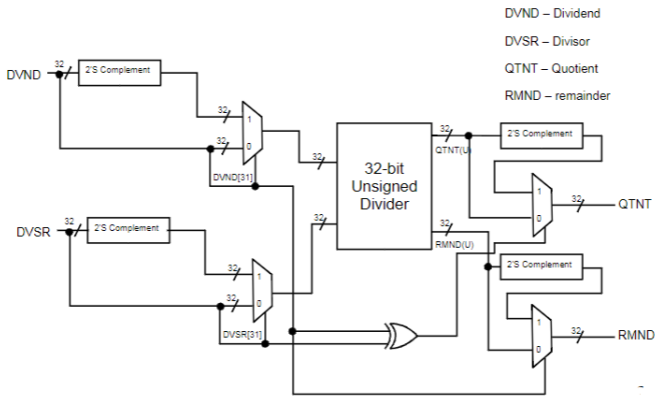
At the end, 'multiply_sign_check' compares the tracked values that determine positive and negatives. If both are positive or both are negative, nothing is done and the product is moved to its respective return registers. Otherwise if the two values denote that the multiplied values were opposite signs, the entire product is two's complemented and then returned in its respective registers.

VIII. Division

Division is the fourth and final operation that must be implemented. Requiring both logical addition and logical subtraction like logical multiplication did, this final operation wraps up the project.

A. *Requirement*

Execute division through normal and logical MIPS instructions. Implement logical division through a Signed Division Circuit [3].



In regards to the positive and negative values, it is the same as the Signed Multiplication Circuit as it keeps tracks of negative values and two's complements them. Aside from that, the only difference is that unsigned division is performed instead of unsigned multiplication.

B. *Normal*

Normal division is done with the normal MIPS operations: 'div' or 'divu.' As I wanted signed division, 'div' was the only option to use. Similar to multiplication, results for 'div' are stored in hi and lo registers, so again, using mfhi and mflo is necessary.

```
divide:
        div     $a0, $a1
        mflo    $v0
        mfhi    $v1
```

Similar to normal multiplication, the quotient is stored in lo and the remainder is stored in hi. The quotient and remainder are then moved to their respective return variables.

C. *Logical*

```
divide:
    ori     $t0, $zero, 0x80000000    # set $t0 to
                                        1000...0000 (32 bit)
    and     $t1, $s0, $t0             # first bit of $s0
    and     $t2, $s1, $t0             # first bit of $s1

    beqz    $t1, dividend_positive    # if $t1 = 0, goto
                                        dividend_positive
    sub_logical($zero, $s0)           # else
                                        twos_complement $s0
    ori     $s0, $v0, 0               # store value back
                                        into $s0
```

```
        ori     $t8, $zero, 1           # sets $t8 to 1
                                        (dividend is negative)
dividend_positive:
        beqz    $t2, divide_start       # if $t2 = 0, goto
                                        divide_start
        sub_logical($zero, $s1)         # twos_complement $s1
        ori     $s1, $v0, 0             # store value back
                                        into $s1
        ori     $t9, $zero, 1           # sets $t9 to 1
                                        (divisor is negative)

divide_start:
        ori     $t0, $zero, 1           # $t0 is counter
        ori     $t4, $s1, 0             # $t4 is divisor
        ori     $t5, $s0, 0             # $t5 is
                                        dividend/quotient
        ori     $t6, $zero, 0           # $t6 is remainder

divide_loop:
        sll     $t6, $t6, 1             # shift remainder
                                        left once
        ori     $t2, $zero, 0x80000000  # last bit
        and     $t2, $t5, $t2           # retrieves last bit
                                        of dividend
        srl     $t2, $t2, 31            # shifts last bit
                                        to lsb
        or      $t6, $t6, $t2           # inserts last bit of
                                        dividend into first
                                        bit of remainder

        sll     $t5, $t5, 1             # shifts dividend
                                        left once

        sub_logical($t6, $t4)           # difference
                                        between
                                        remainder and
                                        divisor
        blt     $v0, $zero, divide_loop_repeat      # if < 0,
                                        goto divide_loop_repeat
        ori     $t6, $v0, 0             # else difference
                                        is new remainder
        or      $t5, $t5, 1             # sets first bit of
                                        dividend as one

divide_loop_repeat:
        sll     $t0, $t0, 1         # increment counter
        beqz    $t0, divide_sign_check  # if = 0, goto
                                        divide_sign_check
        j       divide_loop

divide_sign_check:
        beq     $t8, $t9, signs_equal   # if both are same
                                        sign, goto sign_equal
        beqz    $t8, other_case         # else if not same
                                        sign and $t8 = 0,
                                        goto other_case
        sub_logical($zero, $t5)         # else 2's
                                        complement quotient
        ori     $t5, $v0, 0             # store
        sub_logical($zero, $t6)         # 2's complement
                                        remainder
        ori     $t6, $v0, 0             # store
        j divide_end

signs_equal:
        beqz    $t8, divide_end         # if both are = 0,
                                        goto divide_end

        sub_logical($zero, $t6)         # else 2's
                                        complement remainder
        ori     $t6, $v0, 0             # store
        j divide_end

other_case:
        sub_logical($zero, $t5)         # 2's complement
```

```
                                    quotient
    ori     $t5, $v0, 0             # store

divide_end:
    ori     $v0, $t5, 0             # store lo in $v0
    ori     $v1, $t6, 0             # store hi in $v1
```

For the first two labels, similar to logical multiplication, the dividend and divisor are checked for being negative and two's complemented if they are negative. Once again, the negatives are tracked, but specifically for division, it is specifically tracked whether the dividend or divisor or both or neither are negative.

After that, the code starts with initializing a counter, divisor, dividend that eventually becomes the quotient, and a remainder. The loop continuously shifts the remainder and quotient left and checks if the difference between the new remainder between a newly inserted bit and the divisor is less than zero. If it is, the counter increases, but the value is not kept. If it is greater or equal to zero, remainder takes on the value of the difference and the dividend's LSB becomes one.

By the end, the code goes through a similar negative sign check like in multiplication. However, the difference is that depending on where the negative lies, the dividend or divisor, affects the quotient and remainder's signs in different ways. All the different cases are accounted for and finally, the quotient and remainder are returned in their respective registers.

IX. Testing

Provided with the project is a file named 'proj-auto-test.asm.' Its purpose is to compute different operations on different sets of values using both the normal and logical ALU. The two results are then compared. With 40 calculations to be done, a 40/40 means that all calculations of all operations work. Anything below 40/40 means there was some error in one or more operations. The results have been pasted below.

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


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

As shown by my testing results, I achieved a 40/40, showing that I have passed all the tests given by 'proj-auto-test.asm" file. This means that all my operations are functional and return the correct output.

X. Conclusion

At the end of this project and report, I have learned how to implement mathematical operations through bitwise, logical MIPS instructions. With a 100% success rate (at least on the given test cases), I am greatly satisfied with my project and its final form, though it may still be somewhat messy to others. With that, I have completed the CS47 project which consisted of the bitwise operations: addition, subtraction, multiplication, and division.

References

[1] K. Patra, Class Lecture, Topic: "Addition Subtraction Logic," San José State University, San José, Apr., 18, 2019.

[2] K. Patra, Class Lecture, Topic: "Multiplication Logic," San José State University, San José, Apr., 23, 2019.

[3] K. Patra, Class Lecture, Topic: "Division Logic," San José State University, San José, Apr., 25, 2019.