Arithmetic Implemented By MIPS Logic Operations

Hiep Tran

San Jose State University

[Hieptran1208@gmail.com](mailto:Hieptran1208@gmail.com)

**Abstract**—This report explains the implementation of basic math operations, including addition, subtraction, multiplication and division, using the MIPS logical and normal procedures in Mars. They are compared side by side. The code is provided for better understanding

1. INTRODUCTION

The use of binary logical operation is crucial in computer science. It is implemented in various ways in the field. Understanding how binary logical operations work helps learn computer science better.

The purpose of logical operators is to calculate various expressions, which is the basis of digital circuits in computer hardware. The calculator is written in MIPS (microprocessor without Interlocked Pipeline Stages), which is one of many types of assembly language that can perform addition, subtraction, multiplication, and division through the usage of logical operators and the the operations that are provided by MIPS.

1. REQUIREMENTS

This section discusses the necessary software needed to write the basic mathematical calculator and also provides the needed background information to create the calculator and understand how it works.

1. Software requirements:

To run MIPS assembly language, one must use the MARS software as their interactive development environment (IDE). MARS simulates the runtime environment and execution of MIPS without needing to operate in a low-level environment, as well as providing the same instruction sets and pseudo-instruction sets for use. MARS can be downloaded on Missouri State University website as it is developed by them.

1. Knowledge of Boolean Logic and Binary System:

The machine language in computer s is just series of 0 and 1. The order they are arranged determines the value of each bit pattern. There are many ways to interact those bit patterns together. Some operations such as AND, OR, XOR, NOT, etc. will be represented in the tables below. They express the relations of those bit patterns.

Truth Table for Logical AND

|  |  |  |
| --- | --- | --- |
| A | B | A.B |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Truth Table for Logical OR

|  |  |  |
| --- | --- | --- |
| A | B | A+B |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

Truth Table for Logical XOR

|  |  |  |
| --- | --- | --- |
| A | B | A XOR B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

1. Binary System

A number with a counting base of two is known as a binary number. Binary is a number system with base 2, unlike the decimal system, which uses base 10. The first number from the right side has power 0 and increases 1 if it moves more to the left. In the counting system, once a digit reaches 1, it becomes 0 and rolls over to the next larger digit and places a 1 there. For example, 3 is express as ..011. Elementary hand-paper arithmetic algorithms work for both base 10 and base 2 systems. Moreover, inputs that are equivalent in both systems will yield an output that are equivalent as well. This fact can be used to ensure that the program produces correct outputs.

In computers, integers are represented with binary. To distinguish positive and negative numbers, we look at the MSB in a bit pattern. It is a positive number if the MSB is 0 and it is a negative number if MSB is 1. As an example, the number 4 represented with 6 bits would be 000100 while -2 would be 111100. To convert between a number and its complement (assuming the number is in range of its signed counterpart), invert the bits and add 1 to the number.

1. DESIGN AND IMPLEMENTATION

The following section describes the design and implementation of the logical calculator. The operators implemented by the calculator and described by the paper are addition, subtraction, multiplication, and division.

1. Design

This subsection describes the design of the basic operations of the calculator.

1. Addition and Subtraction

The design of the addition and subtraction implementation go together because of the similarities of the operation. The subtraction operation is the addition operation with the second input complemented to form its negated counterpart. The addition is done by getting the carry bit and the result by logic XOR to bits in 2 numbers. These operations are performed 32 times (the number of bits available in a register). To determine the nth digit of the final output, the logical XOR operation is performed on the nth digits of the first and second inputs as well as the carry bit. The carry bit is initialized to 0, and is calculated by taking the OR operation of the AND of the nth digits of the two inputs with the AND operation of the current carry bit value and XOR operation with the nth digits of the inputs. If the carry bit is 1, we just switch the second number by using NOT and then extract the 0th bit which is a 1 and do the addition as normal. Once the operations are performed 32 times (once for each bit in the registers), the procedure ends.

1. Multiplication

For multiplication, the logic is shifting and adding. We extract 0-32nd bit in a number, if the bit is 0, we shift the multiplicant to the left and shift the multiplier 1 bit to the right. If the bit happened to be 0, we add the multiplicand to the result, then continue shifting the multiplier and the multiplicand. After 32 instruction, we extract the numbers from hi and lo to registers t0 and t1. We move those registers to v0, v1 and call out the result. In short, binary multiplication is simply repeated addition. What we must know is that 0 x 0 = 0, 1 x 0 = 0, and 1 x 1 = 1.

1. Division

First, we set the remainder be 64 bits with the upper 32 bits be 0s and the lower 32 bits be the dividend. The divisor is also 64 bits with the upper part is the divisor and the lower part is just 0s. We must align the divisor with the MSB bits of the dividend. Compare these bits to the divisor. If these bits are greater than the divisor, the quotient bit is set to 1 and then subtraction is performed with MSB bits – divisor. If the MSB bits are less than the divisor, the quotient bit is set to 0 and subtraction is not performed. The divisor is then shifted one bit to the right and we compare the MSB bits to the divisor again. This continues until division is over.

1. Implementation Details

Before the logical calculator operations can be implemented through MIPS, and handful of utility procedures and macroes are defined to assist in the implementation.

1. Utility Macroes

The additional macroes defined in the program are extract\_nth\_bit($regD, $regS, $regT), insert\_to\_nth\_bit($regD, $regS, $regT, $maskReg).

1. Extract\_nth\_bit

Extract\_nth\_bit returns the bit value of a given position for a given number. The defined macro takes three arguments in the following order: the register to store the result, register containing source bit pattern, and register holding the value of the bit position. The macro creates a register with value of 1, then it shift that 1 to the right position. After that, it uses logic and to make and save the result back to that register. It shifts that bit back by using sllv.

.macro extract\_nth\_bit($regD, $regS, $regT)

#regD = result 0 or 1 (of the position regT in the array regS)

#regS = the bit pattern

#regT = position in the bit pattern

li $regD, 0x1

sllv $regD, $regD, $regT

and $regD, $regS, $regD

srlv $regD, $regD, $regT

.end\_macro

1. Insert\_to\_nth\_bit

Insert\_to\_nth\_bit takes four arguments in the following order: register of the bit pattern in which to insert to, register containing the position in which to insert, register containing the value to insert (0 or 1), and a temporary register to create a mask. Firstly it creates a mask with a value of 1. It move that 1 to the right nth position and flip it. 0 becomes 1 and 1 becomes 0. Once done, AND is called on the mask and the bit pattern in which to modify. The bit to insert is then shifted left by n and OR is called with the bit pattern.

.macro insert\_to\_nth\_bit($regD, $regS, $regT, $maskReg)

# regD: bit pattern in which to be inserted at nth position

# regS: position n ( 0-31)

# regT: the bit to insert ( 0x0 or 0x1)-----------regD in the above macro

# maskReg: temporary mask

li $maskReg, 0x1

sllv $maskReg, $maskReg, $regS # maskReg = 0000000..1..0

not $maskReg, $maskReg # maskReg = 1111..0..1

and $regD, $regD, $maskReg

# change the bit at nth position of regD to 0

sllv $regT, $regT, $regS

# bring the bit in regT to the right position. regT = 0000..x..0

or $regD, $regD, $regT

# add the bit to that position of the bit pattern

.end\_macro

1. Store\_stack and load\_stack

The purpose of these two functions is to simply store and load the proper registers on call. They take no arguments and will store/load the registers listed. Specifically, the registers stored and loaded are $fp, $ra, $a0-$a3, and $s0-$s7.

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

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

1. Utility Procedures

i. twos\_complement and twos\_

complement\_if\_neg

Twos\_complement takes an argument in $a0 and returns its complement in $v0. In the twos\_complement subroutine, the NOT of $a0 is taken and stored back in $a0. $a1 has value of 1 and both $a0 and $a1 are passed into the loop to calculate the addition.

twos\_compliment:

# the idea to get a 2's compliment number is invert and add 1 to it

not $a0, $a0

li $a1, 1

#save the results onto the stack

addi $sp, $sp, -20

sw $fp, 20($sp)

sw $ra, 16($sp)

sw $t0, 12($sp)

sw $t1, 8($sp)

addi $fp, $sp, 20

li $t0, 0 # index

li $v0, 0 # sum

li $t1, 0 # carry bit

jal add\_loop

# add the invert and 1

# restore the stack and get the result in v0

lw $fp, 20($sp)

lw $ra, 16($sp)

lw $t0, 12($sp)

lw $t1, 8($sp)

addi $sp, $sp, 20

jr $ra

Twos\_complement\_if\_neg will return the argument if it is more than 0, and branch to twos\_complement if the argument is less than 0.

twos\_compliment\_if\_neg:

bltz $a0, twos\_compliment

la $v0, ($a0)

jr $ra

1. Addition/Subtraction Implementation
2. add\_logical / sub\_logical

Initially, we need to determine what operation we do by checking with register $a2. If it is a ‘+’, we jump to addition label and convert the register a2 to have a value of 0. We extract the 0th position bit out and save it into a temporary register. If a2 is ‘-‘, we perform subtraction and convert a2 to have a value of ffffffff. We again extract the oth bit out and save into a temporary register. With subtraction, we also invert the second number. We move on and do the loop process

1. add\_loop

The loop function will logically calculate the sum of the two inputs in $a0 and $a1 bit by bit. The loop first uses the extract macro to extract 2 bits out and xor them with the initial carry bit. The result will use insert macro to store that bit into a destination register. The carry bit is also calculated and stored in $t3. It will check if there were 32 iterations. If not, we go back and start the loop again with an increment of the index t2 by 1. Afterward, the insert macro is called to place the sum bit into its proper place in $v0. Once the index equals 32, the procedure branches to the exit procedure and returns to the calling procedure

beq $a2, '+', addition

beq $a2, '-', substraction

beq $a2, '\*', multiplication

beq $a2, '/', division

addition:

li $a2, 0x00000000

extract\_nth\_bit($t3, $a2, $zero)

j add\_sub\_logic

substraction:

li $a2, 0xffffffff

extract\_nth\_bit($t3, $a2, $zero)

not $a1, $a1

j add\_sub\_logic

add\_sub\_logic:

li $t2, 0x0 # index

li $t0, 0x0 # result

add\_loop:

extract\_nth\_bit($t4, $a0, $t2)

extract\_nth\_bit($t5, $a1, $t2)

# Value of Y

xor $t6, $t4, $t5

xor $t9, $t3, $t6

insert\_to\_nth\_bit($t0, $t2, $t9, $s0)

# Value of C-out

and $t7, $t4, $t5

and $t8, $t3, $t6

or $t3, $t7, $t8

add $t2, $t2, 1

blt $t2, 32, add\_loop

j end

1. Multiplication Implementation
2. multiplication

The argument initializes all the variables that would be used through out the calculation. Because the result of 2 32 bit numbers is a 64 bit number, we need 2 additional registers to store the bits.

X

A3

A0

A1

T0

T1

# make a0 contain hi and lo (2 registers with lo is a0 and hi is a3)

li $a3, 0

# hi in multiplicand.

li $t0, 0 # lo

li $t1, 0 # hi

li $t2, 0

# index for the mul

li $t3, 0

# carry bit

li $t4, 0

# index for the add loop

li $s7, 31

# the 31rd bit

1. mul\_loop

The procedure will extract the 0th bit of the multiplier a0 and see if that bit is 0 or 1. If it is 0, we jump to shift procedure where we will shift the multiplicand to the left 1 position and shift the multiplier to the right 1 position. When we shift the multiplicand, we also extract and insert the MSB to a temp hi register, which is a3.

If the bit is 1, we add the multiplicand with the result and store it back to the result. To do this, we add lo with lo and hi with hi.

1. additionMul\_LO, additionMul\_Hi

The 2 procedures are derived from the add\_loop procedure in addition. However, in here, we assign temporary registers to save the result before we modify it.

additionMul\_LO: # add the lo multiplicand register and the lo result register

extract\_nth\_bit($s2, $a0, $t4)

extract\_nth\_bit($s3, $t5, $t4)

# Value of Y

xor $s4, $s3, $s2

xor $t9, $t3, $s4

insert\_to\_nth\_bit($t0, $t4, $t9, $s0)

# Value of C-out

and $s5, $s2, $s3

and $s6, $t3, $s4

or $t3, $s5, $s6

addi $t4, $t4, 1

blt $t4, 32, additionMul\_LO

li $t4, 0

move $t6, $t1

additionMul\_Hi: # add the hi multiplicand register and the hi result register

extract\_nth\_bit($s2, $a3, $t4)

extract\_nth\_bit($s3, $t6, $t4)

# Value of Y

xor $s4, $s3, $s2

xor $t9, $t3, $s4

insert\_to\_nth\_bit($t1, $t4, $t9, $s0)

# Value of C-out

and $s5, $s2, $s3

and $s6, $t3, $s4

or $t3, $s5, $s6

addi $t4, $t4, 1

blt $t4, 32, additionMul\_Hi

1. shift

The purpose of shift is to ensure that the final product contains the right sign after the unsigned procedure is called. The shift procedure move 1 bit and jump back to the mul\_loop procedure

extract\_nth\_bit($t8, $a0, $s7) # extract the MSB and move it to hi register

insert\_to\_nth\_bit($a3, $zero, $t8, $s0) # insert that bit to hi register

sll $a0, $a0, 1

sll $a3, $a3, 1

srl $a1, $a1, 1

addi $t2, $t2, 1

blt $t2, 32, mul\_loop

j end

1. Division Implementation
2. Division

The method begins with the index counter $s7 at 0. $a0, which holds the Quotient, is set to $s1. $a1, which holds the dvsr, is set to $s2. And finally $s3, which is the remainder, is set to 0.

V0

A0

A1

…………

V1

division:

li $s7, 0

# index = 1

move $s1, $a0

# dvnd put in s1

move $s2, $a1

# dvsr put in s2

li $s3, 0

# remainder set to 0

1. Div\_loop

In div\_loop, the remainder is shifted left by 1 bit. Then the 31st bit of the quotient ($s1) is put into the 0th bit position of the remainder ($s3). The quotient is then shifted left by one bit. Then, we have a temp S = Remainder - divisor. If S is 0, it means the Dvsr is greater than the remainder. If it is less than 0, we jump to increment counter. If S is not greater than 0 , we set R to equal S and make the 0th bit of Q a 1. Then we increase the index and go back to the loop again.

li $s4, 31

sll $s3, $s3, 1 # move remainder 1 to the left.

extract\_nth\_bit($s5, $s1, $s4)

insert\_to\_nth\_bit($s3, $zero, $s5, $s6)

sll $s1, $s1, 1

move $a0, $s3 # save the remainder and the quotient

move $a1, $s2

la $t0, 0 # index = 0

li $v1, 0 # carry bit = 0

li $v0, 0 # sum = 0

jal add\_loop # S = v0, S = R - D

bltz $v0, increase\_index # v0 < 0, increment the index

move $s3, $v0

li $t0, 1

insert\_to\_nth\_bit($s1, $zero, $t0, $v1) # q[0] = 1

1. increase\_index

Increase the index by 1 and compare with 32. If index < 32, go back to left\_shift\_and\_extract otherwise we end it.

addi $s7, $s7, 1

beq $s7, 32, end\_division

j div\_loop

1. end\_division

End\_division determines the sign of the quotient and remainder. The 31st bits of the original inputs are extracted and XOR is called on both of them. If the XOR is 1, that means only one number is negative, meaning that the sign bit should be negative. If it equals 1, we figure out the correct signs to put on the remainder and the quotient by using various twos\_complements calls. If the XOR’d answer is equal to 0, we branch to positive\_div.

move $v0, $s1

lw $fp, 20($sp)

lw $ra, 16($sp)

lw $a0, 12($sp)

lw $a1, 8($sp)

addi $sp, $sp, 20

move $s6, $a0

move $s7, $a1

li $t8, 31

extract\_nth\_bit($t1,$a0, $t8) # take out the 2 31st bits to compare

extract\_nth\_bit($t2, $a1, $t8)

xor $t6, $t1, $t2 #if xor is 1, it is negative

beqz $t6, positive\_div

move $a0, $s1 # return quotient to a0

jal twos\_compliment

move $s1, $v0 # temp to store the 2's compliment

move $a0, $s3 # remainder to a0

extract\_nth\_bit($t1, $s7, $t8)

bgtz $t1, skip\_secondtwos

jal twos\_compliment

la $v1, ($v0)

la $v0, ($s1)

j end

1. TESTING

Along with “proj\_alu\_logical”, Another class, called “proj\_alu\_normal” is created. This class uses MIPS inbuilt arithmetic operations such as “add”, “sub”, “mul”, and “div” to calculate the functions.

1. Implementation
2. addition\_au\_normal

Addition\_au\_normal calls the MIPS add instruction to add the two inputs. Arguments are passed through the method in $a0 and $a1, and the result is stored in $v0.

add $t0, $a0, $a1

j end

1. subtraction\_au\_normal

Similarly to addition\_au\_normal, the arguments are passed through the method in $a0 and $a1, and the result is stored in $v0.

sub $t0, $a0, $a1

j end

1. Multiplication\_au\_normal

Multiplication\_au\_normal calls the MIPS “mul” instruction and stores the result in $v0. The hi value is then moved into the $v1 register.

mul $t0, $a0, $a1

mfhi $t1

j end

1. Division\_au\_normal

Division\_au\_normal calls the MIPS “div” instruction on $a0 and $a1, which are the two inputs. The hi value contains the remainder and the lo value contains the quotient. The remainder is moved into $v1 and the quotient is moved into $v0.

div $a0, $a1

mflo $t0

mfhi $t1

j end

end:

move $v0, $t0

move $v1, $t1

1. Proj-auto-test

An assembly file was created to test both of the classes to compare the results. 40/40 is passing and otherwise failing. The screenshot below is the result.

1. CONCLUSION

The purpose of this project is to learn how binary logics work. It uses hardware simulation to create software which is 4 basic operations. It helps learn about the logic and the interaction behind every operation. The program was written using the MIPS assembly language and tested with a provided testing file which matched the outputs of the logical calculator operations with the MIPS arithmetic instruction set.



