Implementing Basic Mathematical Operations Through Digital Logic

Sam Vuong
San Jose State University

***Abstract*—This report discusses the implementation of basic mathematical operations using both MIPS normal math operations and MIPS logic operations.**

I. Introduction

Computers are logic crunching machines. Any implementation or algorithm eventually boils down to the fundamental logical operations (NAND, NOR, NOT) that are processed by hardware circuits. Basic mathematical operations, such as addition, subtraction, multiplication, and division are no exception. To understand how these arithmetical operations are completed, we use MIPS digital logic operations to imitate what hardware does. To simulate a MIPS processor, the MARS IDE is used in this project.

II. Requirements

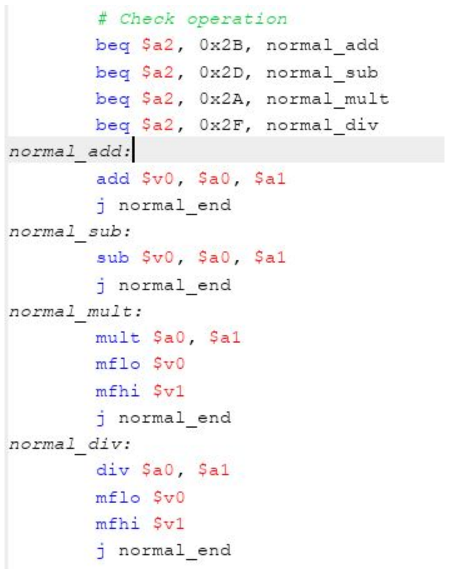
This project features two different ways in which these basic mathematical operations can be implemented. The first way is to use normal math operations found in the MIPS assembly language, which will be carried out in what is called the normal procedure, named au_normal. The second way is to use MIPS digital logic operations, which will be carried out in what is called the logical procedure, named au_logical. Both procedures are to take three arguments: $a0 (first operand), $a1 (second operand), and $a2 (operation code). Both procedures are also to return the result in $v0 and $v1 (if performing multiplication/division).

III. Design and Implementation

This section features any procedures or macros used to fully implement this calculator.

A. Normal Procedure

Basic mathematical operations are fully implemented through the procedure au_normal. The operation code given in argument $a2 is used to determine which normal math operations are to be performed with the values stored in $a0 and $a1 as operands. As shown in the following code snippet, branching is used to jump to the correct label that contains the MIPS math operation corresponding to the operation code.



```
        # Check operation
        beq $a2, 0x2B, normal_add
        beq $a2, 0x2D, normal_sub
        beq $a2, 0x2A, normal_mult
        beq $a2, 0x2F, normal_div
normal_add:
        add $v0, $a0, $a1
        j normal_end
normal_sub:
        sub $v0, $a0, $a1
        j normal_end
normal_mult:
        mult $a0, $a1
        mflo $v0
        mfhi $v1
        j normal_end
normal_div:
        div $a0, $a1
        mflo $v0
        mfhi $v1
        j normal_end
```

.

B. Utility Macros

To assist with this implementation, three macros were used.

1. extract_nth_bit

This macro retrieves the bit (0 or 1) stored in the specified bit position from a specified bit pattern. The macro takes three arguments: a register to store the extracted bit, a register containing the source bit pattern, and a register indicating the bit position to extract from.

```
# $regD contains extracted bit, $regS contains source bit pattern, $regT contains bit position
.macro extract_nth_bit($regD, $regS, $regT)
srlv $regS, $regS, $regT
and $regD, $regS, 1
.end_macro
```

By shifting the source bit pattern by the number of times indicated by the bit position, the bit we are looking to extract is now at the LSB (least significant bit) position. Thus, performing a logical AND operation on this modified bit pattern with 1 gives us the extracted bit.

2. Insert_to_nth_bit

This macro inserts a specified bit (0 or 1) to the indicated bit position of a given bit pattern. The macro takes four arguments: a register containing the bit pattern to be modified, a register holding the bit position to insert into, a register containing the bit value to insert, and a register that can be used for masking.

```
# $regD contains bit pattern to be modified, $regS contains bit
.macro insert_to_nth_bit($regD, $regS, $regT, $maskReg)
li $maskReg, 1 # Mask
sllv $maskReg, $maskReg, $regS
add $t0, $zero, 0xFFFFFFFF
xor $maskReg, $maskReg, $t0 # Invert Mask
and $regD, $regD, $maskReg
sllv $regT, $regT, $regS
or $regD, $regD, $regT
.end_macro
```

Several logical operations are performed in order to manipulate the mask and source bit pattern in such a way that when the bit value to be inserted, $regT, is shifted left to the desired bit position, a logical OR operation can finally be executed to successfully insert the bit value.

3. invert_bit_pattern

This macro is used to obtain the inversion of a given bit pattern. It takes two arguments: a register containing the source bit pattern and a register that can be used for masking.

```
# Inverts bit pattern given a bit pattern in $reg and a register $maskReg to hold mask
.macro invert_bit_pattern($reg, $maskReg)
li $maskReg, 0xFFFFFFFF
xor $reg, $reg, $maskReg
.end_macro
```

The inverted bit pattern is obtained by performing a logical XOR operation on the source bit pattern and the mask containing all 1's.

C. Utility Procedures

Various utility procedures are used in this project.

1. Twos_complement & twos_complement_if_neg

These procedures are used to obtain the two's complemented form of a bit pattern. Only one argument is taken, stored in $a0, which is the source bit pattern. The only difference between the two procedures is that the procedure twos_complement_if_neg only returns the two's complement form of the provided bit pattern only if it is negative.

```
        not $a0, $a0
        li $a1, 1
        jal add_logical # called with ~$a0 and value 1 as $a1
```

The code snippet above is taken from the procedure twos_complement. By calling the procedure add_logical with argument $a0 being the inverted bit pattern, and argument $a1 with value 1, the two's complement is obtained.

2. Twos_complement_64bit

Unlike the previous two procedures relating to obtaining the two's complemented form, this procedure is to be used in order to obtain the two's complement form of a 64 bit result. This is necessary when multiplying and dividing because 32 bit multiplication/division results are 64 bit, that is, 32 bits are stored in one register and another 32 bits are stored in another. This procedure takes two arguments: $a0 (lower 32 bits) and $a1 (higher 32 bits).

```
    not $a0, $a0
    not $a1, $a1
    move $s0, $a1
    li $a1, 1
    jal add_logical
    move $s1, $v0 # Final result for Lo part of 2's complemented 64 bit
    move $a0, $s0 # Use higher 32 bits as first number to add
    move $a1, $v1 # Final carry out, use as second number to add
    jal add_logical
    move $s2, $v0 # Final result for Hi part of 2's complemented 64 bit
    move $v1, $s2 # Save Hi in $v1
    move $v0, $s1 # Save Lo in $v0
```

In this procedure, it is necessary to call add_logical twice because the two's complement form of the lower 32 bits must first be taken. There may be overflow from converting the lower 32 bits into its two's complement form, and if so, it must be placed in $a1 and used in the second call to add_logical with the higher 32-bits.

3. bit_replicator

This procedure is used to replicate a given bit value 32 times. It only takes one argument, $a0, which contains the bit to replicate, and returns the result (either 0x0 or 0xFFFFFFFF) in $v0.

```
        # Body
        beqz $a0, bit_replicator_zero
        # Replicate '1' 32 times
        li $a0, 1
        jal twos_complement # 2's complement of 1 is 0xFFFFFFFF
        j bit_replicator_end
bit_replicator_zero:
        move $v0, $a0
```

If the bit to be replicated is 0, the program immediately jumps to the label bit_replicator_zero where the argument is moved to $v0 where it is returned. Otherwise, the two's complement form of the value 1 is obtained (-1), which is 0xFFFFFFFF.

D. Addition/Subtraction

The digital logic implementation of addition/subtraction consists of three separate procedures, add_sub_logical, add_logical, and sub_logical.

1. Add_sub_logical

This is the common procedure used for the core implementation of addition and subtraction. As subtraction is essentially the addition of a positive number and a negative number, the same common procedure can be used as long as minor modifications are made. This procedure takes three arguments: $a0 (first number), $a1 (second number) and $a2 (mode—indicates whether addition or subtraction should be performed). The procedure returns the final result in $v0, and the final carryout in $v1.

```
        li $t0, 0 # holds bit from first number
        li $t1, 0 # holds bit from second number
        li $t2, 0 # holds $t0 XOR $t1
        li $t3, 0 # holds sum bit
        li $t4, 0 # holds maskReg
        li $t5, 0 # holds first number
        li $t6, 0 # holds second number
        add $s0, $zero, $zero # sum
        add $s1, $zero, $zero # index
        add $s2, $zero, $zero # will contain 0 or 1 depending on mode
        extract_nth_bit($s2, $a2, $s1) # Extract MSB from second number
        beqz $s2, add_routine # after inversion, if necessary, goes into add_routine regardless
        invert_bit_pattern($a1, $t7) # Subtraction, so invert second number
        addi $a1, $a1, 1 # Add 1 to obtain final inverted bit pattern
        li $s2, 0 # Reset $s2, have already used it to determine addition/subtraction
add_routine:
        move $t5, $a0 # Reset value of first number after each iteration
        move $t6, $a1 # Reset value of second number after each iteration
        extract_nth_bit($t0, $t5, $s1) # extract bit from first number at index
        extract_nth_bit($t1, $t6, $s1) # extract bit from second number at index
        xor $t2, $t0, $t1 # $t2 contains t0 XOR t1
        xor $s3, $t2, $s2 # s3 contains this iteration's sum bit
        and $s2, $s2, $t2
        and $t3, $t0, $t1
        or $s2, $s2, $t3
        insert_to_nth_bit($s0, $s1, $s3, $t4) # Insert sum bit to result
        addi $s1, $s1, 1 # Increase index
        bne $s1, 32, add_routine # Loop condition
        move $v0, $s0 # Result
        move $v1, $s2 # Final carryout
```

The lower half of this chunk of code (indicated by the label add_routine) performs the brunt of the work. The code prior to add_routine is used for ensuring that registers start off with values at zero, and to obtain the two's complement of the second operand if subtraction is being performed. Here, twos_complement is not used, but rather, the macro invert_bit_pattern. Add_routine signifies the start of a loop, which is iterated 32 times. Depending on which iteration of the loop it is, the appropriate bit is extracted from each operand, and logical operations are performed in order to obtain the correct sum bit and carry-out bit.

2. Add_logical

This procedure is only used to set the correct value in $a2, 0x0, to indicate that addition should be performed prior to calling add_sub_logical.

```
add $a2, $zero, $zero
jal add_sub_logical
```

3. Sub_logical

Similar to add_logical, this procedure is only used to set the correct value in $a2, 0xFFFFFFFF, to indicate that subtraction should be performed prior to calling add_sub_logical.

```
li $a2, 0xFFFFFFFF # subtraction mode
jal add_sub_logical
```

E. Multiplication

1. Mul_unsigned

This procedure performs multiplication, taking in two arguments, $a0 (multiplicand) and $a1 (multiplier). The result is returned in $v0 (lower 32 bits) and $v1 (higher 32 bits).

```
        # Body
        li $s0, 0 # Index
        li $s1, 0 # Final sum or Hi of product
        move $s2, $a1 # Multiplier or Lo of product
        move $s3, $a0 # Multiplicand
mul_unsigned_loop:
        beq $s0, 32, mul_unsigned_loop_end
        li $t0, 0
        li $a0, 0 # Will contain current LSB of multiplier
        extract_nth_bit($a0, $s2, $t0) # Using $t0 first as a register indicating 0th position
        jal bit_replicator
        move $t0, $v0 # Now contains replicated bit pattern
        li $t1, 0 # Will contain bit pattern of this step
        and $t1, $s3, $t0
        move $a0, $s1
        move $a1, $t1
        jal add_logical # Increase sum by this step's sum
        move $s1, $v0
        srl $s2, $s2, 1
        li $t2, 0 # Will contain LSB of current sum (Hi)
        extract_nth_bit($t2, $s1, $t2)
        li $t3, 31 # Contains value of 31 to be used to indicate bit position to insert to
        insert_to_nth_bit($s2, $t3, $t2, $t4)
        srl $s1, $s1, 1
        addi $s0, $s0, 1
        j mul_unsigned_loop
mul_unsigned_loop_end:
        move $v0, $s2
        move $v1, $s1
```

As can be seen in the code snippet above, registers indicating the index and Hi are initialized. The original arguments $a0 and $a1 are also saved in two different registers as $a0 and $a1 are later used as arguments in extract_nth_bit, which changes their bit pattern. The label mul_unsigned_loop indicates the start of a loop that is iterated 32 times. Depending on the iteration, the bit in the multiplier is replicated. Then, the logical operation AND is used to obtain the partial sum from the multiplicand and replicated bit. Partial sums are then added to obtain the final result.

2. Mul_signed

This procedure computes the product of two numbers in which one or more operands may be negative. The procedure takes the same arguments as mul_unsigned, and similarly, returns the result in $v0 and $v1.

```
        # Body
        move $s6, $a0 # Contains old multiplicand
        move $s7, $a1 # Contains old multiplier
        jal twos_complement_if_neg
        move $s0, $v0 # Will contain new multiplicand (or stay the same if non-negative)
        move $a0, $s7 # Move multiplier to $a0 before testing if negative
        jal twos_complement_if_neg
        move $s1, $v0 # Will contain new multiplier (or stay the same if non-negative)
        move $a0, $s0 # New multiplicand
        move $a1, $s1 # New multiplier
        jal mul_unsigned
        li $t0, 31 # Contains value indicating MSB
        li $t1, 0 # Will contain MSB of old multiplicand
        li $t2, 0 # Will contain MSB of old multiplier
        li $t3, 0 # Will contain sign S of result
        extract_nth_bit($t1, $s6, $t0)
        extract_nth_bit($t2, $s7, $t0)
        xor $t3, $t1, $t2 # Contains sign S of result
        beqz $t3, mul_signed_end
        move $a0, $v0
        move $a1, $v1
        jal twos_complement_64bit
```

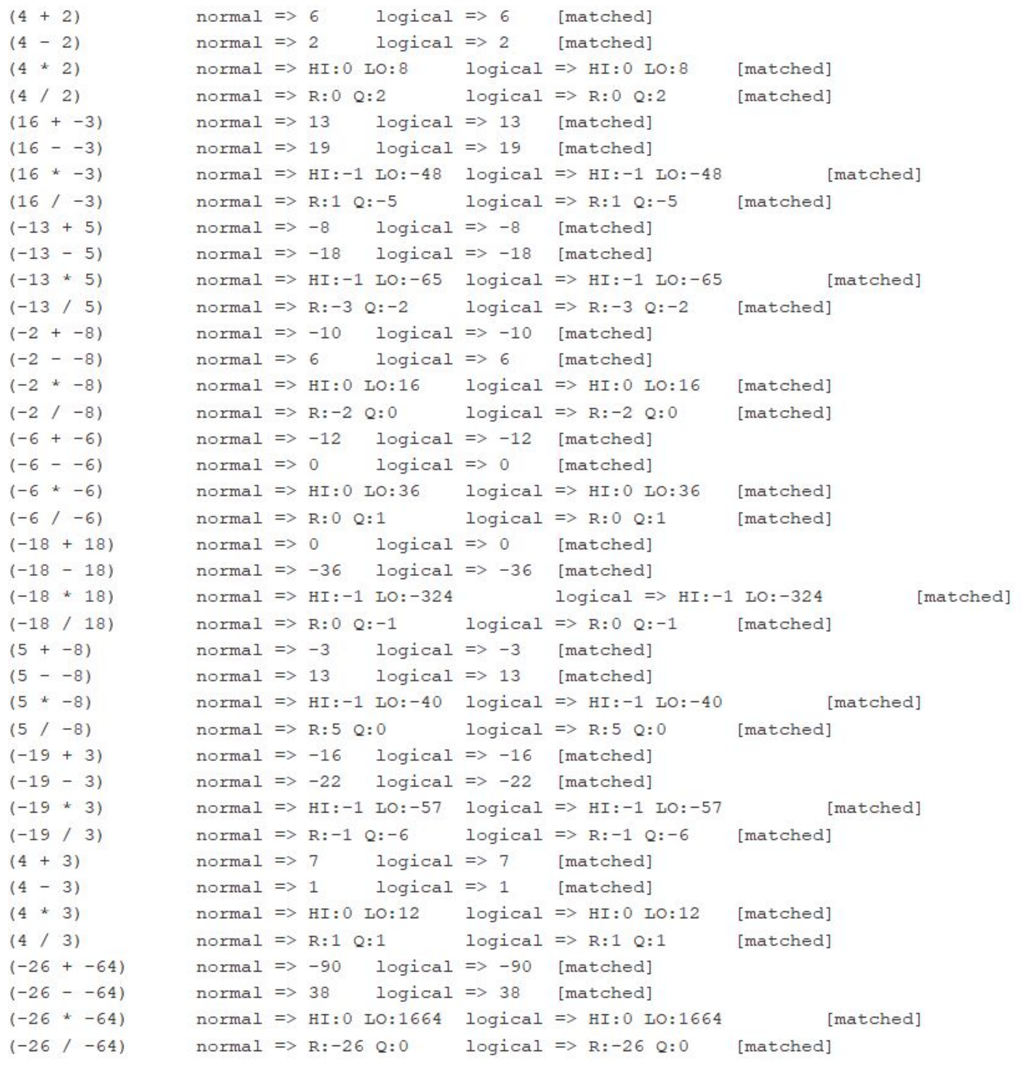
The procedure obtains the two’s complement form of the operands if they’re negative by calling twos_complement_if_neg. Mul_unsigned is then called with the two’s complement form of any negative operands.

The XOR logic operation is used at the end to obtain the sign of the result, to determine whether or not twos_complement_64bit needs to be called.

F. Division

IV. Testing

The following results are obtained after implementing all required procedures, and assembling/running proj_auto_test.asm.



```
(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]
```

IV. Conclusion

Coding in MIPS assembly language is definitely no easy task, with strict guidelines to follow to ensure that desired values are preserved in registers. Implementing these basic arithmetical operations using digital logic really helped me better understand how computers simplify more complex algorithms into logic.