Implementation of a Basic Calculator with MIPS Assembly Language

Benjamin Reyes, Computer Scientist, San Jose University

Abstract -- This document explains the process in which a MIPS Assembly processor can be utilized to design a basic calculator that handles addition, subtraction, multiplication (signed and with overflow), and division. The high level code and design will be illustrated through code snippets to implement all the functions stated.

I. Introduction

Fundamentally, computers are capable of processing and understanding binary instructions through the advent of logic gates. These logic gates are capable of taking inputs of 0's and 1's and deduce a conclusion as an output. Usage of these logic gates allows for arithmetic operations (i.e. addition, subtraction, multiplication, ect.).

For this project, utilizing the MIPS processor's architecture, two programs will be created: one utilizing the built in basic instructions the MIPS processor has, and another utilizing the Pseudo Instructions on the processor. Both will create a basic calculator with the same functionality.

II. Requirements

This section defines the knowledge and digital requirements to complete the task of the basic calculator.

A) Software Requirements

The software we will be utilizing is Assembly language written and compiled within the MARS design environment. MARS can be downloaded on the Missouri State University website below however Java is required to be able to run MARS: https://courses.missouristate.edu/KenVollmar/mars/download.htm

We will be using three different source files CS47_proj_alu_logical.asm, CS47_proj_alu_normal.asm, and CS47_proj_macro.asm to implement all the operations required

B) Binary Algebra and Boolean Logic

This section will cover the required mathematical knowledge needed to execute the programs

1) Binary Algebra

Numbers can be represented in multiple ways. In day to day life we utilize a Decimal numerical system where there are 10 symbols to represent the total possible numbers (0-9). In Binary there are only two symbols 0 and 1 representing the logical on (1) and off (0). We can then represent a number as a row of 0's and 1's as below.

DECIMAL	**BINARY**
0	0
1	1
2	10
3	11
4	100
5	101
6	110

7	111
8	1000
9	1001
10	1010

With this format, computers can transfer a 1 or 0 via electrical current through wires on silicon chips. Any number in Decimal can be represented in Binary including negatives using a concept called the 2's complement. This is achieved in processors by splitting the total possible values in half where the opposite sequence plus 1 will create the negative or positive equivalent (i.e 000100 = -4 and 111100 = 4).

2) Boolean Logic

Boolean Logic is the concept of math utilizing only true and false. Much like binary, boolean logic allows for comparisons to be made between two values, true and false, to deduce some form of conclusion. This is the basis of logic gates such as AND, OR, XOR, ect. that can be arranged to create larger and more complex systems. Below are examples of commonly used logic gates.

a	b	a AND b	a OR b	a XOR b	a NOT
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Above there are two variables *a* and *b* giving input into the gates. The logic gates being used are AND, OR, XOR, and NOT. AND will check if both *a* AND *b* have a 1 and output a 1 if that is true. OR will test to see if *a* OR *b* have a 1 and will output a 1 if that is true. XOR will test to see if *a* OR *b* have a 1, but not both, and output a 1 if that is true. Finally, NOT will check the given variable if it has a 1 input, there will be a 0 output and vice versa.

III. Design and Implementation

This section will go over the different operations the calculator should achieve followed by the code that implements them.

A) Operation Designs

This subsection will go over the four different operations the calculator will achieve

1) Addition

Binary addition when using 2's complement does not require any modification to the numbers before the algorithm starts. Before the algorithm starts it is necessary to remember between each loop whether or not there is a carried bit to the next position. The carry bit will start at 0.

The addition algorithm will be achieved by executing an addition loop 32 times on the two operands. This loop will go from least significant bit to most significant bit comparing the values in both operands at each bit position. From these comparisons the loop will do a series of logic tests to determine what to insert into the position in the final number.

To start, the loop will test an XOR between the two operands selected bit to see if immediately a 1 or 0 will end up in the final number in this position before continued testing. Next it will use an AND to test if there will be a 1 in the final position and a carried bit from the previous loop. If there is a carried bit and a 1 from the previous XOR, the bit will be carried to the next loop and a 0 set in the current position, if there is a 0 in the final position and a carried bit from the previous loop: the final position will have a 1 instead and the carried bit set to 0. Finally if both operands have a 0 the final bit in the selected position will be a 0 unless there is a

carried 1 bit which will set the position to 1 and the carried bit to 0.

2) Subtraction

Subtraction uses the addition loop however requires a different setup. The subtracted operand needs to be inverted to the opposite sign in order to utilize the addition loop as subtracting a positive is like adding a negative and vice versa. This is based on the MSB of the operands where a 1 is negative and a 0 is positive. This also requires the carry bit to start at a 1 in order to fix the discrepancy in two's complement numbers when they are inverted.

3) Multiplication

Multiplication is achieved via a loop executed 32 times to read the entire mutiplier. through utilizing a HI and LO register to handle overflow, the first argument will be the multiplicand and the second will be the multiplier which is stored in the LO. The HI will be treated as a temporary final number holder. Before any multiplication can start, both operands must be checked for negative signs. If there is a negative operand, it must be inverted to a positive or the algorithm will break due to the 1 extension of negative numbers.

The loop will select the lowest bit from the LO register (multiplier) and check if it is a 1. If it is a 1 then the multiplicand will be added to the HI register and then shifted right one bit. This shift will coincide with a LO register shift to the right by 1 bit. This shift will also have the lowest value from the HI placed into the 31st significant position of the LO before the shift happens so as not to lose the value. This is to shift all the LO bits into the proper position since we are using the HI as a temporary register to store the multiplication loop results. Transfering to the 31st position is important as to save the 32nd bit in the LO from being lost due to 2's complement inversion at the end if needed. This is based on the MSB of the operands where a 1 is negative and a 0 is positive

If there is a 0 in the selected multiplier bit the registers shift without the addition, but still exchange the bits from HI to LO.

4. Division

Binary division is achieved similarly to decimal division. The first argument passed will be the dividend and the second argument will be the divisor. The division loop will subtract the divisor from the dividend if the dividend is still greater or equal to the divisor.

Division also requires negative numbers to be inverted to avoid the 1 extension caused by 2's complement numbers. This is based on the MSB of the operands where a 1 is negative and a 0 is positive. The signs will be stored in saved registers with the quotient being negative or positive based on both the dividend and divisor, but the remainder sign based on the dividend only.

B) Implementation

This section will cover the macros and procedures used to implement the operations described previously in the MIPS assembly language

1) Utility Macros

a) retrieve_bit($val, $targ, $shf)

Takes a register ($val) and moves the selected bit at $shf to the LSB of the register (clearing all the other values), and stores it in register $targ.

```
.macro retrieve_bit($val, $targ, $shf)
        li      $t6, 31                 #
        sub     $t6, $t6, $shf          #
        sllv    $t7, $val, $t6          #
        srlv    $t7, $t7, $t6           #
        srlv    $t7, $t7, $shf          #
        move    $targ, $t7              #
.end_macro
```

This macro is essential to all operations as it will extract a single bit from any given register without modifying the register. It is used to select the sign bits in operands, multiplier bits, addition bits for comparison, and bits for shifting between LO and HI for multiplication

b) set_bit($val, $targ, $pos)

This macro sets the $val bit at the given $pos in the $targ register

```
# Macro set_bit
# Usage: retrieve_bit($val, $targ, $pos)
.macro set_bit($val, $targ, $pos)
        sllv    $t7, $val, $pos         #
        or      $targ, $targ, $t7       #
.end_macro
```

This macro is used to put bits into the proper position in addition and also transferring bits from the HI register to the LO register.

2) Utility Procedures

a) invert_a0

This procedure will take the 2's complement inversion of the $a0 register and store it within $s2 which it will subsequently then store back into $a0.

```
# inverts register a0 and puts it into s2
invert_a0:
        jal     inverter                # i
        or      $s2, $v0, $zero         # s
        or      $a0, $s2, $zero         # r
        j       invert_fixed_a0         # j
```

b) invert_fixed_a0

This procedure will follow immediately after invert_a0 and test if value in register $a1 is needed to be inverted for usage in multiplication and division.

```
invert_fixed_a0:
        beqz    $s5, multi_or_divide
        or      $a0, $a1, $zero
        jal     inverter
        or      $s1, $v0, $zero
        or      $a0, $s2, $zero
        j       multi_or_divide
```

c) multi_or_divide

This procedure simply checks if the operator is a multiplication or division symbol as the inversion checkers are shared by both operations. This is to ensure the operations start on the proper path.

```
multi_or_divide:
      beq    $a2, 0x2A, multiplication_loop_initialize
      beq    $a2, 0x2F, division_loop_initialize
```

d) inverter

This procedure will take the number within the $a0 register and take the 2's complement inversion (with a 1 added to it) and store it into $v0 for usage.

```
inverter:
        addi    $sp, $sp, -24
        sw      $a0, 0($sp)
        sw      $a1, 4($sp)
        sw      $a2, 8($sp)
        sw      $fp, 12($sp)
        sw      $ra, 16($sp)
        addi    $fp, $sp, 24
        li      $a1, 1
        nor     $a0, $zero, $a0
        li      $a2, 0x2B
        jal     au_logical
        lw      $a0, 0($sp)
        lw      $a1, 4($sp)
        lw      $a2, 8($sp)
        lw      $fp, 12($sp)
        lw      $ra, 16($sp)
        addi    $sp, $sp, 24
        jr      $ra
```

This is achieved by storing the frame of all the given arguments as to not lose them when returning. It will then take the bitwise NOR of what's in $a0 and then assign a 1 to $a1 in order to fix the natural discrepancy in 2's complement numbers.

Next the value 0x2B (ascii '+') is stored in $a2 and au_logical is called. This will add the 1 previously stated as required and store the inverted and fixed value into $v0. The frame is then restored and the PC returned to the previous code

e) restore_frame_return

This takes all values from $s0-$s5, $a0-$a3, $fp, and $ra and restores them to the values given when au_logical was called.

```
# does what it says
restore_frame_return:
        lw      $a0, 0($sp)
        lw      $a1, 4($sp)
        lw      $a2, 8($sp)
        lw      $a3, 12($sp)
        lw      $fp, 16($sp)
        lw      $s0, 20($sp)
        lw      $s1, 24($sp)
        lw      $ra, 28($sp)
        lw      $s2, 32($sp)
        lw      $s3, 36($sp)
        lw      $s4, 40($sp)
        lw      $s5, 44($sp)
        addi    $sp, $sp, 52
        jr      $ra
```

3) Addition/Subtraction Implementation

The following code snippet will utilize the algorithm described above for addition and subtraction.

a) addition_l_loop

```
# $s0 --> final number
# $s1 --> carry bit
# $t0 --> bit selected
# utilizes full adder design from book
addition_l_loop:
        slti    $t3, $t0, 32    #keep loopin
        beqz    $t3, end_addition_loop
        retrieve_bit($a0, $t1, $t0)
        retrieve_bit($a1, $t2, $t0)
        xor     $t4, $t2, $t1   # check if c
        and     $t5, $t4, $s1   # check if w
        xor     $t4, $s1, $t4   # check if i
        and     $t6, $t1, $t2   # check for
        or      $s1, $t5, $t6   # check if t
        set_bit($t4, $s0 $t0)   # place bit
        addi    $t0, $t0, 1     #increment p
        j       addition_l_loop
```

As can be seen, the essential logic elements are there with $t1 and $t2 being the operands, the current selected bit is $t0, and $s0 is the carry bit between loops. $t4 is also essential in that it is the value that will be put in the selected position $t0.

Following is the initialization of either a subtraction or addition loop

b) subtract_start/addition_start

```
#same thing as adding a negative
subtract_start:
        nor     $a1, $a1, $zero
        li      $s1, 1
        j       addition_l_loop

#set up to do addition normally
addition_start:
        li      $s1, 0          # s
        j       addition_l_loop
```

With addition, the only setup required is to set our carry bit $s1 to 0. With subtraction however, in order to use the addition loop the second operand ($a1) must have it's sign flipped. Also the one discrepancy caused when inverting 2's complement

numbers must be handled thus the carry bit ($s1) is initialized with a 1.

4) Multiplication Implementation

a) multiplication_start

```
# $s0 --> hi saved
# $s1 --> lo saved and multiplier
# $s2 --> multiplicand
# $s3 --> loop count
# $s4 and $s5 --> two's compliment si
multiply_start:
        or      $s1, $a1, $zero
        or      $s2, $a0, $zero
        li      $s0, 0
        li      $t0, 31
        retrieve_bit($a0, $s4, $t0)
        retrieve_bit($a1, $s5, $t0)
        beqz    $s4, invert_fixed_a0
        j       invert_a0
```

Here the registers $s1 and $s2 will be loaded with the proper values of $a1 (multiplier) and $a0 (multiplicand) respectively. Afterwards the most significant sign bit will be saved for both operands in $s4 and $s5 and the negative tests will be called.

b) multiplication_loop_initialize

```
multiplication_loop_initialize:
li      $s0, 0                  #
li      $s3, 32                 #
j       multiplication_loop
```

This simply sets the counter up to 32 and clears the HI register ($s0) for usage in the multiplication_loop

The code snippet below will show the main multiplication loop.

c) multiplication_loop

```
# multiplication is achieved by selectin
# based on multiplier bit location.
multiplication_loop:
        beqz    $s3, fix_signs          #
        retrieve_bit($s1, $t1, $zero)  #
        beqz    $t1, shift_for_next_bit#
        or      $a1, $s0, $zero         #
        li      $a2, 0x2B               #
        jal     au_logical
        or      $s0, $v0, $zero         #
        j       shift_for_next_bit
```

As the algorithm showed, the next multiplier bit will be retrieved from the LO register and the test to see if it is a 1 is executed. If there is a 0 in the retrieved bit a call to shift_for_next_bit is made, but if there is a 1 in the multiplier bit a call is made to au_logical in order to add the value inside $a0 to the saved HI in $s0. Then there will be a load putting the output in $v0 back into $s0 followed by a call to shift_for_next_bit.

Next the code snippet will show the shifting of registers and values after the multiplication_loop

d) shift_for_next_bit

```
# after every loop of multiplication (wh
# multiplier bit) the register needs to
shift_for_next_bit:
        srl     $s1, $s1, 1             #
                                        #
        retrieve_bit($s0, $t1, $zero)  #
        li      $t3, 31                 #
        set_bit($t1, $s1, $t3)          #
        srl     $s0, $s0, 1             #
        addi    $s3, $s3, -1            #
        j       multiplication_loop
```

This label follows the algorithm stated above for multiplication. Immediately the LO register ($s1) will shift out the last used multiplier bit. Next the LSB of the HI register will be saved into $t1 then set into the LO register ($s1) at the 31st significant bit. This is to keep the most significant sign bit positive for inversion later. Next the HI register is shifted and

the counter is reduced by 1. The loop continues with a call back to multipication_loop.

Next is the fix_signs that will restore a negative value to the final output if need be.

e) fix_signs

```
# based on saved signs ($s4, $s5) it w
# number stored in lo and hi($s1, $s0)
fix_signs:
        xor     $t0, $s4, $s5
        beqz    $t0, multi_return
        or      $a0, $s1, $zero
        jal     inverter
        or      $s1, $v0, $zero
        nor     $s0, $s0, $zero
        j       multi_return
```

Previously, the MSB sign bit was saved in $s4 and $s5 for arguments $a0 and $a1 when they were first sent to the au_logical. Here the XOR test will give a 0 if the output is to be positive and 1 if the final value will be negative. If there is no need to invert the final value the mutli_return will be called to end the multiplication.

The inverter is called only once as the HI and LO represent a single number so a 1 is only required to be added once to fix the 2's complement discrepancy. The LO will have the inverter called on it while the HI will only have a bitwise inversion.

f) multi_return

```
# loads the lo and hi with the corre
multi_return:
        or      $v0, $s1, $zero
        or      $v1, $s0, $zero
        j       restore_frame_return
```

Loads the $s1 register into $v0 (LO) and $s0 to $v1 (HI) and calls a restore_frame_return.

5. Division Implementation

This snippet will show the start and initialization of the division operation.

```
# $a0-->$s2 --> starting dividend
# $a1-->$s1 --> divisor
# $s3--> counter/quotient
divide_start:
        or      $s1, $a1, $zero         # 
        or      $s2, $a0, $zero
        li      $t1, 31                 # 
        retrieve_bit($a0, $s4, $t1)     # 
        retrieve_bit($a1, $s5, $t1)
        beqz    $s4, invert_fixed_a0    # 
        j       invert_a0

division_loop_initialize:
        or      $a1, $s1, $zero         # 
        li      $a2, 0x2D               # 
        li      $s3, 0                  # 
        j       division_loop
```

a) divide_start

Here the division_start will share the same utility procedures as multiplication_start. The dividend ($a0) will be loaded into $s2 and the divisor ($a1) will be loaded into $s1. From here the sign bits will be saved in $s4 and $s5 then the negative tests will be called.

b) division_loop_initialize

The $a1 register will be set to the divisor and a subtraction call will set into $a2. The quotient counter $s3 will be set to 0. All of these operations are to set up for usage inside the division_loop.

c) division_loop

```
division_loop:
        slt     $t0, $s2, $s1           # 
        bnez    $t0, division_set_values
        or      $a0, $s2, $zero         # 
        jal     au_logical
        or      $s2, $v0, $zero         # 
        addi    $s3, $s3, 1             # 
        j       division_loop
```

The division loop will check for the condition that the dividend ($s2) left is less than the divisor ($s1) and break off to division_set_values if that is true. Otherwise the dividend ($s2) will be loaded into $a0 and a subtraction call will be made to au_logical to subtract the divisor from the dividend. There will also be an increase in the final quotient ($s3) by one after saving the new value from the subtraction back into $s2.

```
# check to see if remainder or quotient need t
division_set_values:
        xor     $t3, $s4, $s5
        beqz    $t3, fixed_inversion_quotient
        or      $a0, $s3, $zero
        jal     inverter
        or      $s3, $v0, $zero
        j       fixed_inversion_quotient

# checks to see if remainder needs to be inver
fixed_inversion_quotient:
        beqz    $s4, fixed_inversion_remainder
        or      $a0, $s2, $zero
        jal     inverter
        or      $s2, $v0, $zero
        j       fixed_inversion_remainder

# assign to proper registers for output v1 --
fixed_inversion_remainder:
        or      $v1, $s2, $zero
        or      $v0, $s3, $zero
        j       restore_frame_return
```

d) division_set_values

This is the first test to see if inversion is required on the quotient by using the previously saved sign bits $s4 and $s5. If one or the other (not both) was negative a XOR will return a 1 and and immediately invert the quotient and move on to fixed_inversion_quotient or immediately transition if the XOR returns a 0

e) fixed_inversion_quotient

Once the quotient has been tested, there needs to be a check if the remainder will be negative based on if the sign bit from the dividend was negative at the start. If it wasn't $s4 will be 0 and fixed_inversion_remainder will be called. If $s4 is 1, the remainder will be converted back to its negative sign and move onto the final stage of division: fixed_inversion_remainder.

f) fixed_inversion_remainder

This is the final step of division. $s2 will be loaded into $v1 for the remainder and $s3 will be loaded into $v0 for the quotient. Then the restore_frame_return is called.

IV. Testing

In order to test the calculator, a series of calls to the operations must be made testing all potential combinations of positive and negative signs as well as with potential different remainders for division.

```
(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 ***

-- program is finished running --
```

A) Testing Implementation

The CS47_proj_alu_normal.asm is utilized to compare the values output from CS47_proj_alu_logical.asm where the previous code snippets have been implemented.

1) addition and subtraction

These two procedures are within the au_normal and will utilize the Pseudoinstruction *add* and *sub* to call addition and subtraction, respectively, on the two operands and store them into register $v0 as output.

2) multiplication

This will call the *mult* Pseudoinstruction to multiply the 2 operands together and place bits greater than 2^{31} into the HI register. All of the bits less than or equal 2^{31} will be stored in the LO register. These two will be output separately and compared separately to the au_logical counterparts.

3) division

This will utilize the *div* Pseudoinstruction to divide operand in $a0 by operand $a1. This will then place the quotient in the LO register and the remainder in the HI register and be output separately to compare to au_logical counterparts.

B) proj_auto_test

This program takes two arrays of numbers loaded into .data and utilizes each array to input values into $a0 and $a1. From here every pair of numbers will be tested in the au_normal and au_logical procedures for all possible operations within them. As it does each operation it will look into the predetermined location output registers and compare the values between the two programs resulting in a [matched] or [not matched] if indeed the values are equal.

V. Conclusion

The objective of the project was to create a simple calculator capable of addition, subtraction, multiplication, and division. So far the au_logical procedure is capable of resulting in correct outputs between two operands with signed numbers on all of these operations.

The next steps for this program would be to reduce duplicate code within the program as well as turning multi-use procedures into macros that could be used regardless of what assembly file they are in. Furthermore, more complicated mathematical procedures such as exponential, logarithmic, factorial, and square root functions could be implemented into the au_logical. However, it was essential to create these basic functions as they create the groundwork for more complicated functions.