Implementation of a Basic Calculator through Digital Logic in MIPS

FirstName LastName, Computer Scientist, San Jose State University

***Abstract*—This work thoroughly explains the implementation of a basic calculator through digital logic in the MIPS assembly language. The paper first describes the software and knowledge needed to understand the design and implementation of the calculator. The paper then describes the upper level design of the implementation followed by details on the actual implementation of the program. Code snippets are provided to provide a better understanding of the program. The program is tested with a pre-written test file which compares the calculator's results with MIPS arithmetic instruction sets.**

I. Introduction

The use of logical operators to calculate expressions is a fundamental concept that serves as the foundation for digital circuits. To understand how digital logic can be used to carry out basic arithmetic, the objective of this project is to implement a basic calculator mainly through the use of logical operators. Functions supported by the calculator include the addition, subtraction, multiplication, and division of two integers.

The calculator is written in MIPS (microprocessor without Interlocked Pipeline Stages), which is one of many types of assembly language. The software used to simulate a MIPS environment and write the implementation is MARS (MIPS Assembler and Runtime Simulator), an IDE developed and released by Missouri State University.

II. Requirements

This sections describes both the software required to implement the calculator as well as the preliminary knowledge to understand the design and implementation of the calculator.

A. Software Requirements

To simulate the runtime environment of the MIPS assembly language, the software MARS is used as an 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. Note that since MARS is written in Java, the user will require an installation of Java on the work station.

Once MARS is launched, some settings will have to change for the program to run correctly. Under Settings, select "Initialize Program Counter to global 'main' if defined." The three files required for the calculator are cs_47_proj_macros.asm," cs_47_proj_procs.asm, and proj-auto_test.asm.

B. Knowledge of Boolean Logic and Binary System

The following subsections describe the preliminary concepts required to implement the logical calculator.

1) Boolean Logic

The implementation of digital logic requires a solid understanding of both Boolean logic and binary arithmetic. In digital circuits, the expressions true and false are represented as a 1 or 0 respectively. From here, truth tables are constructed to determine the output of a Boolean expression given two truth values.

The AND operation will essentially return 0 if its two inputs are neither 1. Table I illustrates the output of the AND operation given two inputs.

TABLE I
TRUTH TABLE FOR LOGICAL AND

A	*B*	*A.B*
0	0	0
0	1	0
1	0	0
1	1	1

The OR operation will essentially return 0 only if its two inputs are both 0. Table II illustrates the output of the OR operation given two inputs.

TABLE II
TRUTH TABLE FOR LOGICAL OR

A	*B*	*A+B*
0	0	0
0	1	1
1	0	1
1	1	1

The XOR operation will return 0 if its two inputs match, and will return 1 if its two inputs do not match. Table III illustrates the output of the XOR operation given two inputs.

TABLE III
TRUTH TABLE FOR LOGICAL XOR

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

2) Binary System

A number with a counting base of two is known as a binary number. Typically, the alphabet consists of 0 and 1, symbols which conveniently coincide with the expressions of Boolean logic. In the counting system, a digits place will count to 1 before "rolling over" the digit to the next higher place. For example in the base 10 system, the next digit after 9 will roll over to form 10, 19 will roll over to form 20, and 99 will roll over to form 100. In binary, 1 is represented as 1, which rolls over to 10 when incremented to 2.

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 between negative and positive numbers, the most significant bit determines the sign of the number. As an example, the number 2 represented with 6 bits would be 000010 while -2 would be 111110. 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.

III. 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.

A. Design of Calculator

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

1) Addition and Subtraction

The design of the addition and subtraction implementation is lumped together because of the similarities of the operation. In fact, the subtraction operation is simply the addition operation with the second input complemented to form its negated counterpart.

The addition logic involves performing logical operations to determine the digit of the final output as well as a "carry bit" to carry over the number for the next logical operation. 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. The figure below illustrates the logical equations formally.

Once the operations are performed 32 times (once for each bit in the registers), the procedure ends. Note that there is a possibility that the carry bit is not carried over if the operations are performed 32 times. In this case, the overflow can be stored in another register to return.

For subtraction, simply find the complement of the second input before performing the procedure.

2) *Multiplication*

The unsigned multiplication logic involves creating a mask from the least significant bit of the current multiplier, using the mask on the multiplicand and adding the result to the final output (initialized to 0).

The mask is created by replicating the first bit of the current multiplier. If the bit is 0, the mask is 0; if the bit is 1, the mask is -1. The mask and the multiplicand are AND together and added to a hi counter. The multiplier and hi counter are shifted to the left by 1, with the hi counter's dropped off bit inserted into the multiplier's most significant bit. These operations are performed 32 times to receive the final output (in place of multiplier).

For signed multiplication, the multiplier and multiplicand are made positive if either are negative and the unsigned multiplication is performed. The final sign is determined by performing the XOR operation on the most significant bits, and complementing the results of the unsigned multiplication if the sign value is 1.

3) Division

The unsigned division logic first shifts the remainder (initialized to 0) to the left, grabbing the most significant bit from the current dividend storing the bit in the least significant bit in the remainder. The dividend is then shifted to the left.

The divisor is then subtracted from the remainder value. If the result is more than or equal to 0, the remainder is made to equal the result, and the bit 1 is inserted in the dividend's least significant bit. If the result is less than 0, the loop simply continues. Once the steps are performed 32 times, the operation ends with the quotient stored in the dividend's place.

For signed division, the sign of the quotient is determined similarly as the product's sign was determined in the signed

multiplication logic. The sign of the remainder is determined by checking the most significant bit of the dividend. If the sign is 1, complement the remainder.

B. 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, insert_to_nth_bit, store_stack, and load_stack.

i. Extract_nth_bit

Extract_nth_bit returns the bit value of a given position for a given number.

```
  #$regD contains 0 or 1 depending on nth bit 0
or 1
  #$regS source bit pattern
  #$regT bit position


  <CODE REMOVED>
```

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 functions by shifting the source bit pattern by the bit position number, moving the desired bit to the least significant bit position. From there, AND is called with 1 to extract the desired bit and stored.

ii. Insert_to_nth_bit

Insert_to_nth_bit loads a given bit value into the nth position of a value and returns it.

```
  #$regD bit pattern in which to insert to
  #$regS position of bit to insert (0-31)
  #$regT register containing 0 or 1 to insert
  #$maskReg register to hold mask
  .macro insert_to_nth_bit($regD, $regS, $regT,
$maskReg)

  <CODE REMOVED>



  .end_macro
```

The defined macro 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.

The macro functions first by setting the mask value to 1. The mask is then shifted to the left by n. The OR operation is then called with the mask and -1 to invert its bits. 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.

iii. 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 regardless of whether they were used or not. Though the proper use of the stack involves storing and loading only the registers that are used, macroes were defined to cut down on the number of lines in the code and increase readability. Specifically, the registers stored and loaded are $fp, $ra, $a0-$a3, and $s0-$s7.

2) Utility Procedures

i. twos_complement and twos_complement_if_neg

```
twos_complement:

  <CODE REMOVED>



twos_complement_if_neg:

  <CODE REMOVED>
```

Twos_complement takes an argument in $a0 and returns its complement in $v0.

The complement is calculated by calling XOR on the argument and -1. Add_logical is then called to add the XOR'd result and 1.

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.

ii. Twos_complement_64bit

This procedure takes the lo and hi values as arguments from the results mul_unsigned and returns the complement of the two.

First, both the arguments are inverted by calling XOR and -1. The inverted lo value and 1 is passed into the add_logical function, which returns the complemented lo value in $v0, and the overflowed carry bit in $v1. This bit is then added to the inverted hi to get its complement. The complemented lo and hi are returned in $v0 and $v1 respectively.

iii. bit_replicator and replicate

```
bit_replicator:
  <CODE REMOVED>

replicate:
  <CODE REMOVED>
```

These two procedures assist in “replicating” a given bit to occupy all 32 registers.

Bit_replicator will take an argument in $a0 and return 0 in $v0 if the argument is 0. The argument will branch to replicate if the argument is 1.

For replicate, the procedure simply returns -1 in $v0.

iv. Exit

This procedure will simply call the load_stack macro and jump back to the return address. The procedure is called after the end of a calculator operation.

3) Addition/Subtraction Implementation

```
add_logical:

  <CODE REMOVED>

sub_logical:

   <CODE REMOVED>

   <CODE REMOVED>


add_loop:

  <CODE REMOVED>
```

i. add_logical / sub_logical

Initially, the carry bit, final sum, and index are initialized to 0 in add_logical. For sub_logical, the first argument is stored, and the second argument is moved to $a0 to be complemented with the twos_complement procedure. The arguments are moved back to place and the proper variables are initialized. From here, both procedures jump to add_loop.

ii. add_loop

The add_loop function will logically calculate the sum of the two inputs in $a0 and $a1. The loop first checks if the counter is equal to 32. If not, the procedure will call macroes to extract the nth position of the argument. Using the XOR operation, the sum bit is calculated. The carry bit is also calculated and stored in $v1. Afterward, the insert macro is called to place the sum bit into its proper place in $v0. Finally, the index is incremented by 1 and the procedure jumps to itself. Once the index equals 32, the procedure branches to the exit procedure and returns to the calling procedure.

4) Multiplication Implementation

```
mul_logical:

   <CODE REMOVED>
```

i. mul_logical

The arguments are initially stored twice to free space for the complement procedure and to check for the sign in the later procedure. The two arguments are checked and complemented if negative. From here, the results are passed into the argument registers and mul_unsigned is called.

ii. mul_unsigned

The procedure initializes the index, the hi counter, and stores the arguments. The multiplicand is stored in $s4 while the multiplier is stored in $s5. The procedure then calls mul_loop

```
<CODE REMOVED>
```

iii. mul_loop

The loop initially checks to see if the index is equal to 32. Then the procedure extracts the first bit of the multiplier, which is then replicated after loaded into $a0. The result of the replicator and the multiplicand are ANDed together and stored into $s7. The current hi counter value and $s7 are loaded into arguments and add_logical is called.

The result of the add_logical is moved back to the hi counter. The multiplier is then shifted to the right, while the least significant bit of the hi counter is extracted and inserted into the most significant bit of the multiplier. The hi counter is then shifted to the right. The results of the lo (corresponding with the multiplier's register) and hi counter are stored in $v0 and $v1 respectively. Finally, the index is incremented and the procedure calls itself.

Once the index reaches 32, mul_signed is called to place the appropriate signs.

iv. mul_signed

The purpose of mul_signed is to ensure that the final product contains the right sign after the unsigned procedure is called. Recall that mul_logical stored the original arguments and complemented the arguments if either were negative. Here, the most significant bits of both the multiplier and multiplicand are extracted with the macro and XOR'd together to receive the sign. If the sign is 0, the procedure calls exit. If the sign is not 0, the lo and hi results of mul_loop are loaded into the argument variables and passed into the procedure twos_complement_64bit. Afterward, the procedure exits.

5) Division Implementation

```
div_logical:

<CODE REMOVED>
```

i. div_logical

Much like mul_logical, the arguments are stored twice and complemented if negative. Afterward, the arguments, complemented or otherwise, are loaded into argument registers. The procedure div_unsigned is called.

```
div_loop:

<CODE REMOVED>
```

ii. div_unsigned

The index and remainder registers are initialized to 0. The dividend and divisor are stored and div_loop is called.

div_loop and branch

The procedure first branches if the index is equal to 32. The remainder is then shifted to the left by 1. The extract macro is called on the 0^{th} position of the dividend and inserted into the new position of the remainder variable with the insert macro. The dividend is the shifted to the left.

The remainder and divisor are loaded into $a0 and $a1 respectively and sub_logical is called. The result is then moved into $s2. A branch is called if the subtraction result (stored in $s2)is greater than or equal to 0. If the value of $s2 greater than or equal to 0, the remainder is assigned the value of $s2, and the bit 1 is inserted into the first bit of the divisor.

If $s2 is less than 0, the branch is not called. In either case, the index is incremented by one. The quotient (corresponding with the dividend) and remainder are loaded into $v0 and $v1 respectively.

Once the index is equal to 32, div_signed is called to handle the signs of the quotient and remainder.

iii. div_signed, signed_quotient, and signed_remainder

These procedure handles the signs of the remainder and quotient.

Div_signed first stores the results of the div_loop procedure. The procedure then extracts the 31^{st} bits of both the dividend and divisor and store the results in $s2 and $s3 respectively. $s2 and $s3 are then XOR'd and the sign is then stored in $s4.

The procedure the branches depending on whether the quotient or remainder needs to be complemented. If $s4 is not equal to 0, the procedure calls signed_quotient. If $s2 (first bit of dividend) is not equal to 0, the procedure calls signed_remainder. Otherwise, the procedure exits.

In signed_quotient, the quotient (stored in $s6 by div_signed) is passed into $a0 and twos_complement is called. The result is re-stored in $s6. If $s2 is not equal to 0, the procedure will call signed_remainder. Otherwise, the result of the complement and remainder is loaded into $v0 and $v1 and the procedure exits.

In signed_remainder, the remainder (stored in $s7 by, div_signed) is passed into $a0 and twos_complement is called. The result of the complement and the quotient are loaded into $v1, and $v0 respectively.

IV. TESTING

To test that the calculator's procedures operate correctly, control procedures are written using MIPS's inbuilt arithmetic instructions. The procedure au_normal is written to parse the input and call the respective instruction.

A. *Testing Implementation*

1) Add_normal and sub_normal

Add_normal and sub_normal will call MIPS instructions add and sub respectively. The arguments are passed into the parameters and the result is stored in $v0.

2) *Mul_normal*

Mul_normal will call the instruction mul and store the results in $v0. The hi result of the instruction is moved from the hi counter to $v1.

3) *Div_normal*

Div_normal will call the instruction div, which stores the results in the quotient in the lo counter and remainder in the hi counter. These results are moved to $v0 and $v1 respectively.

B. *Proj-auto-test*

An assembly file was written and provided to easily test that the calculator operations work as expected. The file provides sample inputs and matches the results of au_normal with au_logical to ensure that the outputs are correct. The snippet above is a sample of the output of this test.

```
(4 + 2)  normal=> 6   logcal=> 6   [matched]
(4 - 2)  normal=> 2   logcal=> 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    logcal=> 13    [matched]
(16 - -3)  normal=> 19    logcal=> 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    logcal=> -8    [matched]
(-13 - 5)  normal=> -18   logcal=> -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   logcal=> -10   [matched]

                          .
                          .
                          .
                          .
                          .
                          .
                          .

 (-19 / 3)   normal=> R:-1 Q:-6   logical=> R:-1
Q:-6   [matched]
(4 + 3)  normal=> 7   logcal=> 7   [matched]
(4 - 3)  normal=> 1   logcal=> 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   logcal=> -90
[matched]
(-26 - -64)  normal=> 38    logcal=> 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 --
```

V. Conclusion

The objective of this project was to implement a basic calculator through the use of logical operators. The project provided insight on how digital circuits operate and how the logic behind the circuits are implemented. 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.

Next possible steps include implementing more complex expressions such as exponentiation, square rooting, decimal calculations, and more.