Arithmetic Implemented By MIPS Logic Operations

FirstName LastName
San Jose State University
email@sjsu.edu

***Abstract*—This report focuses on the implementation of the basic mathematical operations (addition, subtraction, multiplication, and division) using MIPS normal and logical procedures in MARS (MIPS Assembler and Runtime Simulator).**

I. INTRODUCTION

With the use of the MARS simulator, we will be able to calculate mathematical operations using both MIPS normal operations (add, sub, mult, and div) and logical operations (Boolean logic such as AND, OR, and NOT). In this project, we will utilize the language MIPS and the MARS simulator to perform the mathematical calculations. The project objectives are listed below:

1) To install and setup the MARS simulator.
2) To design and implement arithmetic calculations using both MIPS mathematical operations and MIPS logical operations.
3) To test the implemented procedures using MARS simulator.

This report provides steps on how to install MARS as well as discusses the design and implementation process behind the logical procedures used for the arithmetic calculations. It also observes the test cases used to ensure the procedures have been implemented correctly and potential errors.

II. INSTALLATION AND SETUP

A. Installation of the simulation tool

MARS is available for free download at the following site: http://courses.missouristate.edu/KenVollmar/MARS/. Click on the Mars4_5.jar file to start the downloading process.

B. Loading Project in the Simulator

Download the given zip file from Canvas at the site https://sjsu.instructure.com/courses/1185206/assignments/4051197 and unzip it. It should include three different files:

1) cs47_proj_macro.asm
This file contains all macros used and implemented.
2) cs47_proj_procs.asm
This file contains all procedures used and implemented.
3) proj_auto_test.asm
This file contains the testing procedure.

Open the Mars4_5.jar file, which will lead to the display of a menu on the top. Click on "File" and then "Open" to navigate to the directory in which the unzipped file is stored. Find the folder, and load each of the three files individually.

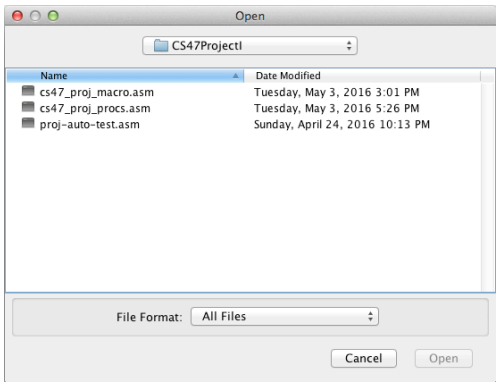

Fig. 1. Navigating to Folder

Once all files have been opened, the file names should be displayed.

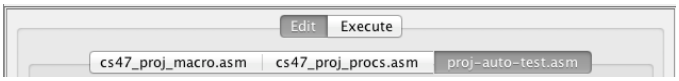

Fig. 2. Loading All Files

C. Settings of simulation tool

For convenience, check the box near the bottom of the screen to show line numbers, which will help when running the program and testing.

Also, go to the top menu and find "Settings", where a menu containing certain setting options will be checked. All defaults will be kept with the addition of one more setting. In the case that the default settings differ, check or uncheck setting options according to Fig. 3.

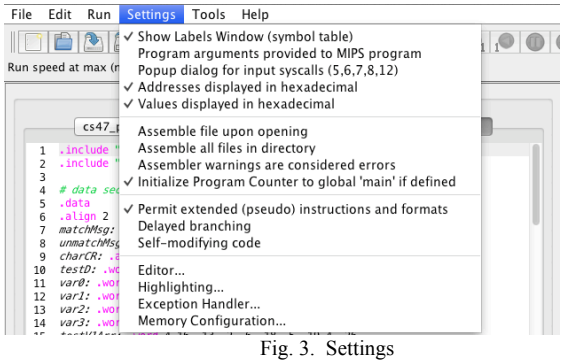

Fig. 3. Settings

1) Default Settings

The default settings include: “Show Labels Window (symbol table)”, “Addresses displayed in hexadecimal”, “Values displayed in hexadecimal”, and “Permit extended (pseudo) instructions and formats”.

2) Additional Setting

The additional setting is “Initialize Program Counter to global ‘main’ if defined”. This will let the program start at the address defined by “main” rather than the first address it encounters, which happens to be the first line of the file cs_proj_procs.asm. Starting there will make no sense to the simulator (as the testing procedure should start from its first line) and will lead to errors.

III. REQUIREMENTS OF ARITHMETIC PROCEDURES

For this project, there are two different types of procedures that need to be implemented. As mentioned previously, common mathematical procedures such as addition, subtraction, multiplication, and division, can be implemented in two different ways, which mirrors the required procedures to be written.

A. Normal Procedure

The normal procedure, which is named au_normal, will consist of calculating the mathematical expressions using MIPS mathematical operations. Such operations include add, sub, mul, and div. This procedure will take three different arguments:

1) Register $a0

This argument is the first operand in the mathematical expression.

2) Register $a1

This argument is the second operand in the mathematical expression.

3) Register $a2

This argument is the operator provided in ASCII code that determines the operation that should be performed.

The results of the addition and subtraction will be stored in register $v0. However, for multiplication and division, two registers $v0 and $v1 will be used. In doing multiplication, $v0 will contain the LO portion of the result as $v1 will contain the HI part. For division, $v0 will contain the quotient while $v1 will hold the remainder.

B. Logical Procedures

The logical procedure, which is named au_logical, will consist of multiple procedures that calculate each mathematical procedure using logical operations. This includes operations such as AND, OR, NOT, and XOR. The use of the MIPS mathematical operations (add, sub, mul, and div) are not permitted in these procedures, aside from using them to increment counters in loops. Similar to the normal procedure, the procedure will take three arguments:

1) Register $a0

This argument is the first operand in the mathematical expression.

2) Register $a1

This argument is the second operand in the mathematical expression.

3) Register $a2

This argument is the operator provided in ASCII code that determines which operation that should be performed.

The results of the addition and subtraction will be stored in register $v0. However, for multiplication and division, two registers $v0 and $v1 will be used. In doing multiplication, $v0 will contain the LO portion of the result as $v1 will contain the HI part. For division, $v0 will contain the quotient while $v1 will hold the remainder.

It is noted that completing this procedure requires the creation of multiple procedures unlike the au_normal procedure, which can be sufficiently completed alone.

IV. DESIGN AND IMPLEMENTATION OF PROCEDURES

A. Normal Procedure

The completion of the au_normal procedure alone is sufficient enough to perform the mathematical calculations. Given the three arguments, determine which math operator the symbol in register $a2 refers to. Depending on the operator and using branches, use the corresponding MIPS normal mathematical operations to calculate the expression.

1) Operator ‘+’

This corresponds to addition, where “add” can be used to determine the sum of the two operands.

2) Operator ‘-’

This corresponds to subtraction, where “sub” can be used to determine the difference between the two operands.

3) Operator ‘’*

This corresponds to multiplication, where “mul” can be used to determine the product of the two operands.

4) Operator ‘/’

This corresponds to division, where “div” can be used to determine the quotient and remainder from dividing the dividend by the divisor.

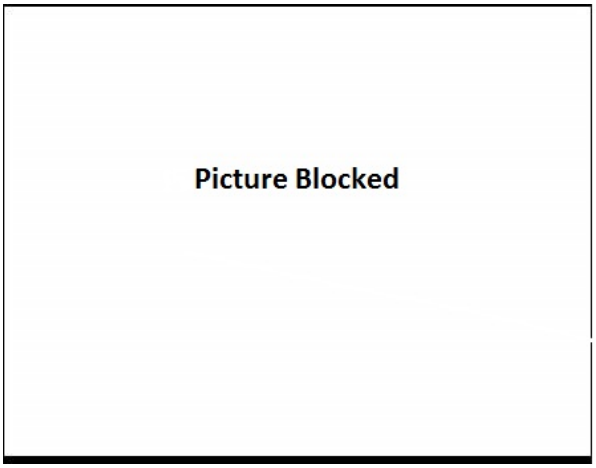

Fig. 4. Implementation of A Branch and its Operation

B. Logical Procedure

Similarly, the au_logical procedure mirrors the normal procedure; however, it will consist of calling multiple procedures. Through au_logical, other procedures will be called correspondingly to the necessary mathematical operation that will be performed.

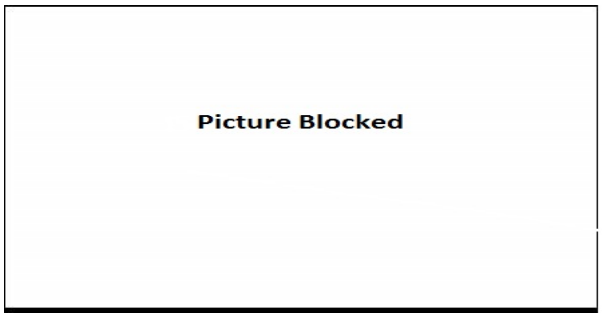

Fig. 5. Branches to Determine Corresponding Procedures to Call

1) add_sub_logical

This procedure will calculate the sum of its first two arguments $a0 and $a1 (first operand and second operand respectively). The third argument will determine its mode: addition or subtraction.

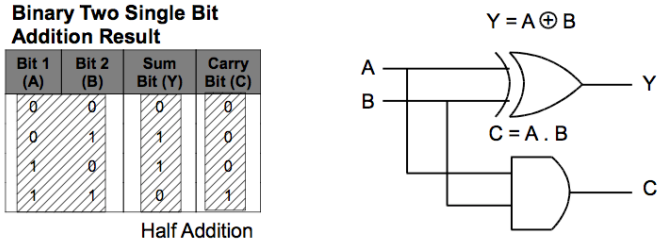

Fig. 6. Half Adder [2]

Fig. 6 displays the logical design of a half adder, which does single bit addition only considering its carry-out bit. It is noted that the sum can be determined through an AND operation, and its carry-out bit can be determined through an XOR operation.

To consider both the carry-in bit and the carry-out bit in single bit addition, the full adder is used. The full adder is simply two half adders. By looking at Fig. 7, it is noted that the half of the truth table is the same as the original half adder, and the rest considers the presence of a carry-in bit.

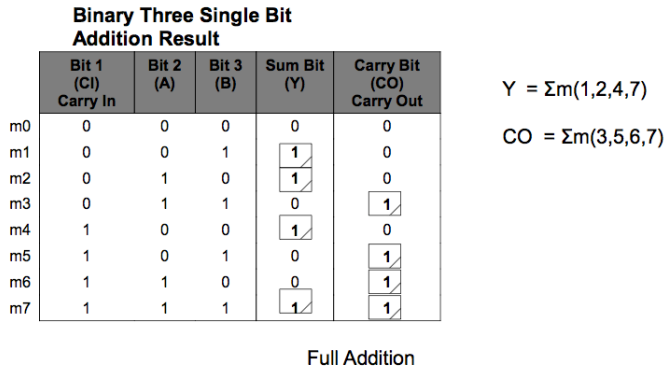

Fig. 7. Truth Table for Full Adder [2]

To determine the logical design of the full adder, a K-Map can be used.

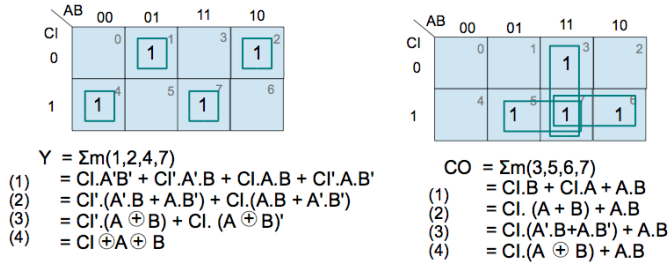

Fig. 8. K-Map for Full Adder [2]

Fig. 8 demonstrates the simplifying of the expressions to reuse the XOR operation already used in the half adder. These expressions will be used to determine the logical design for addition.

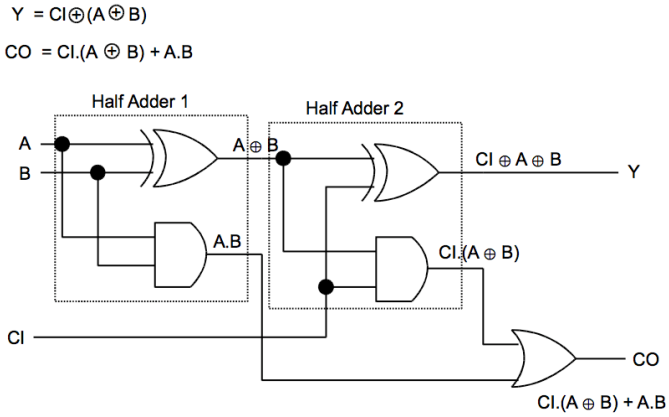

Fig. 9. Logical Design for Addition/Subtraction [2]

The sum is determined by performing the XOR operation on the carry-in bit, the first bit of the first operand, and the first bit of the second operand. The final carry-out bit is determined by the OR operation between the carry-out bits from the two half adders. Note that part of the Boolean expressions come from the expressions determined by the single half adder.

The sum, however, can mean either addition or subtraction. Subtraction is also equivalent to the addition of a negative operand. Thus, to perform subtraction, the second operand should be converted to its two's complement form. The following equation will convert a number to its corresponding two's complement form.

$$\$a0 = \sim \$a0 + 1. \quad (1)$$

The argument $a2 will determine which of the two operations should be performed by the value it holds. If $a2 holds 0, addition is performed; if $a2 holds 1, subtraction is performed. Subtraction is performed by inverting the second operand and adding in a carry-in bit. On the contrary, addition assumes the carry-in bit is 0. This means the full adder can be reused for subtraction as shown when comparing Fig. 10 and 11.

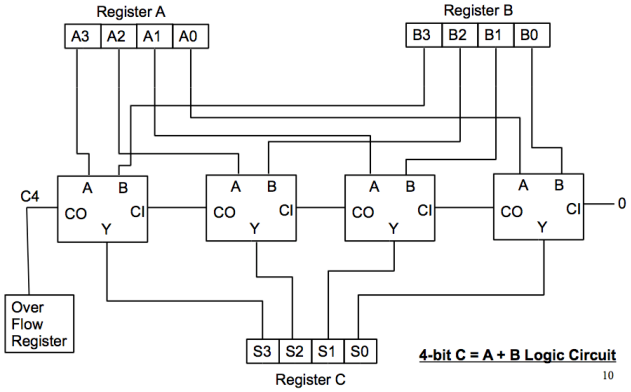

Fig. 10. Multi-bit Addition Only [2]

The only differences are the potential for the presence of a carry-in bit (for addition, the carry-in bit can be ignored) and an XOR operation. If the carry-in bit holds 1, the XOR operation between the bit and B will invert B for subtraction; else, it will carry on with no carry-in bit and pass whatever value B is holding for addition.

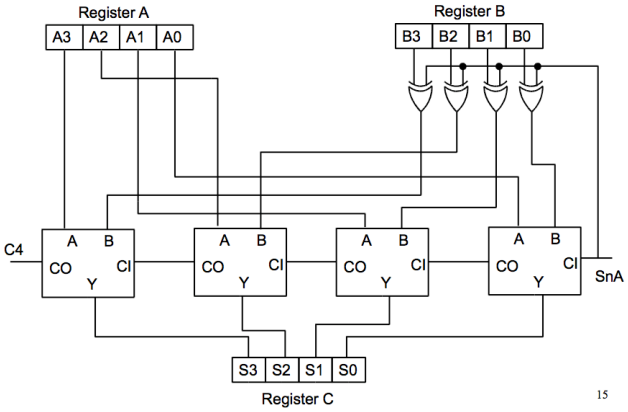

Fig. 11. Multi-bit Addition/Subtraction [2]

To implement this in the software as shown by Fig. 12, a loop is created to run 32 times (since there are 32-bits). A branch is also created to determine whether addition or subtraction is to be performed, where the second operand should be inverted if the mode is subtraction. By following the Boolean expressions of the full adder, the sum (considering both the carry-in and carry-out bit) can be performed.

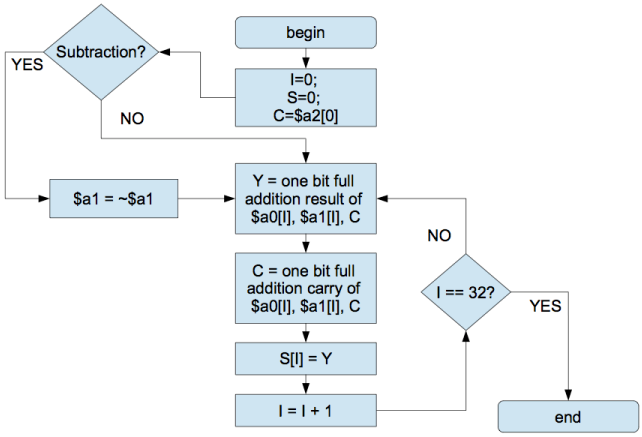

Fig. 12. Code Flowchart for Addition/Subtraction [2]

The implementation is rather straightforward from the flowchart.

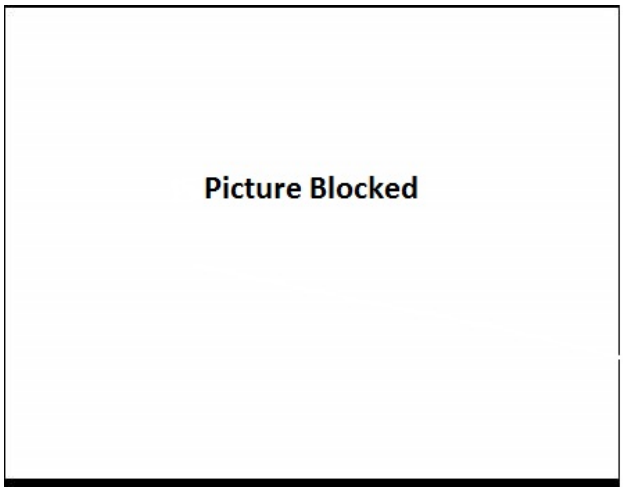

Fig. 13. Implementation of Loop Used in Addition/Subtraction

2) add_logical

This procedure will call add_sub_logical procedure to perform addition. Thus, the argument $a2, which determines the operation, will contain 0x00000000.

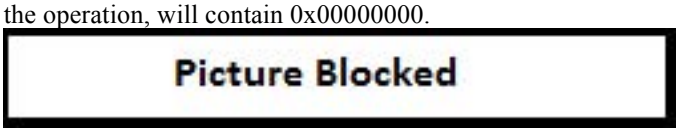

Fig. 14. Implementation for Addition Only

3) sub_logical

This procedure will call add_sub_logical procedure to perform subtraction. Thus, the argument $a2, which determines the operation, will contain 0xFFFFFFFF.

Fig. 15. Implementation for Subtraction Only

4) twos_complement

This procedure will be used to convert any operand to its corresponding two's complement form. It will take one argument $a0 as the number to be converted. To convert any number a0, (1) should be performed.

The NOT operation and the full adder will be used for this process.

Fig. 16. Implementation of Two's Complement

5) twos_complement_64bits

This procedure will be used to convert the contents of two 32-bit registers into a 64-bit result. Two arguments will be passed: $a0 contains the LO portion of the total result, and the HI portion of the result will be in $a1. First, convert both contents into their corresponding two's complement forms. Add 1 to $a0, and add its carry-out with $a1. Note that converting to a 64-bit result is not the same as converting two

separate 32-bit numbers to two's complement. The full adder should be used for the addition process.

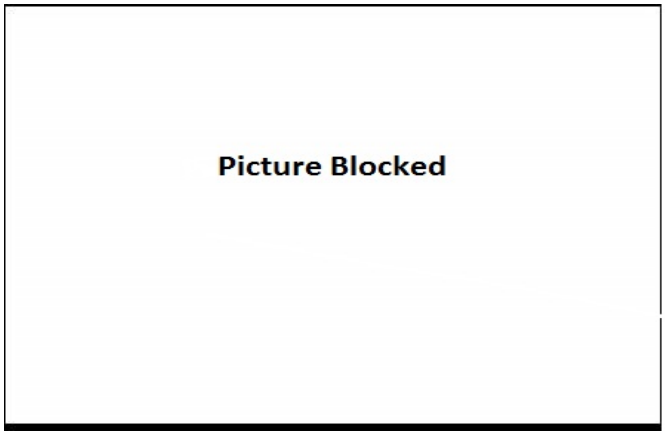

Fig. 17. Implementation of Two's Complement 64 Bits

6) bit_replicator

This procedure will be used to replicate a certain individual bit 32 times to become 32 bits. It will come to use in the multiplication procedure. This procedure takes one argument, which will be the bit value (either 0 or 1) to replicate. The value returned will be a 32-bit number consisting of the value replicated 32 times.

Fig. 18. Implementation of Bit Replicator

7) mul_unsigned

This procedure will perform unsigned multiplication with two arguments $a0 and $a1, multiplicand and multiplier respectively. Fig. 19 shows the logical design of multiplication.

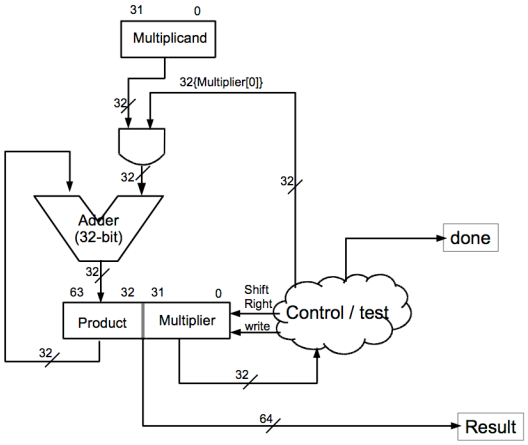

Fig. 19. Logical Design of Unsigned Multiplication [3]

The operands will be stored in 32-bit registers whereas the result will be in a 64-bit register (since multiplication is a 64-bit operation). Instead, we can have the two 32-bit registers holding the product and multiplier simulate the 64-bit register instead.

The multiplicand is AND-ed with the 0^{th} bit of the multiplier replicated 32 times (due to 32-bit register). This is because the AND operation between binary patterns corresponds to how binary multiplication works.

TABLE I
AND OPERATION FOR BINARY MULTIPLICATION

A	B	A AND B	A*B
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

After, the result is added together to the total product, where the full adder can be utilized again. The multiplier will be shifted to the right to denote that the certain individual bit multiplication is done. This process continues until the multiplier has disappeared, and the full product fills up the two 32-bit registers.

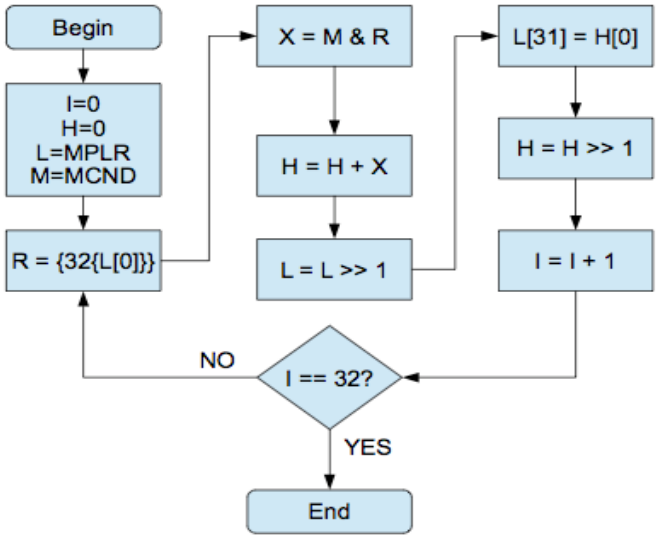

Fig. 20. Flowchart for Unsigned Multiplication [3]

To mimic this in the software as seen in Fig. 20, a loop will be created to perform the necessary commands 32 times (since there are 32 bits). To simulate the 64-bit register with two 32-bit registers, the contents in the multiplier register will be shifted to the right once. The first bit in the product register will be extracted and inserted into the most significant position (31^{st}) of the multiplier register. Then, the product register will be shifted. Therefore, this mimics the shifting done in a regular 64-bit register.

Picture Blocked

Fig. 21. Implementation of Loop in Unsigned Multiplication

8) mul_signed

This procedure will take care of all multiplication performed, where its two arguments $a0 (multiplicand) and $a1 (multiplier), either signed or unsigned, are taken into consideration.

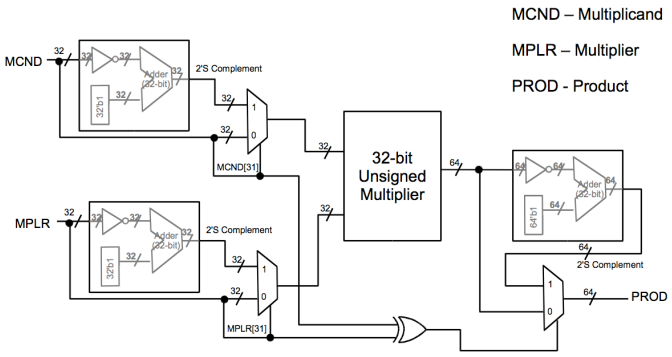

Fig. 22. Logical Design of Signed Multiplication [3]

First, operands will be converted to its corresponding two's complement form if they are negative, where they will be passed to mul_unsigned to perform unsigned multiplication.

Picture Blocked

Fig. 23. Checking for Negative Operands

After, the sign of the product is determined by looking at the original operands. Given two positive or two negative operands, the product will be positive. Given one positive and one negative operand, the product will be negative. This corresponds to the XOR operation.

TABLE II
XOR OPERATION FOR SIGNED NUMBERS

A	B	A XOR B	SIGNS
0	0	0	- * - = +
0	1	1	- * + = -
1	0	1	+ * - = -
1	1	0	+ * + = +

Then, the product is converted into its two's complement 64-bit form if negative.

Picture Blocked

Fig. 24. Implementation of Signed Multiplication

9) div_unsigned

This procedure is similar to mul_unsigned in that it will perform unsigned division. It also takes two arguments $a0 as dividend and $a1 as divisor.

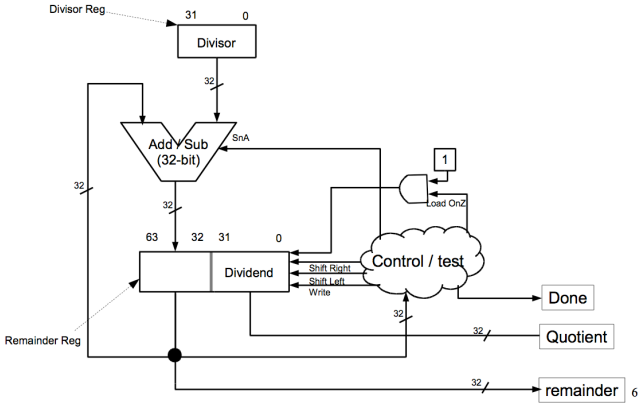

Fig. 25. Logical Design of Unsigned Division [4]

Division is also a 64-bit operation, so the registers used to contain the quotient and the dividend can simulate a 64-bit register as the dividend will eventually be shifted out. The dividend is subtracted by the divisor, starting at the most significant bit position (left-most). If the subtraction leads to a positive result, this will result in a 1 in the corresponding bit position in the result. If the subtraction cannot be done and thus leads to a negative result, the operation will be reversed. Instead, a 0 will be placed in the bit position of the result instead. This process will continue as the dividend is shifted out and replaced by the quotient.

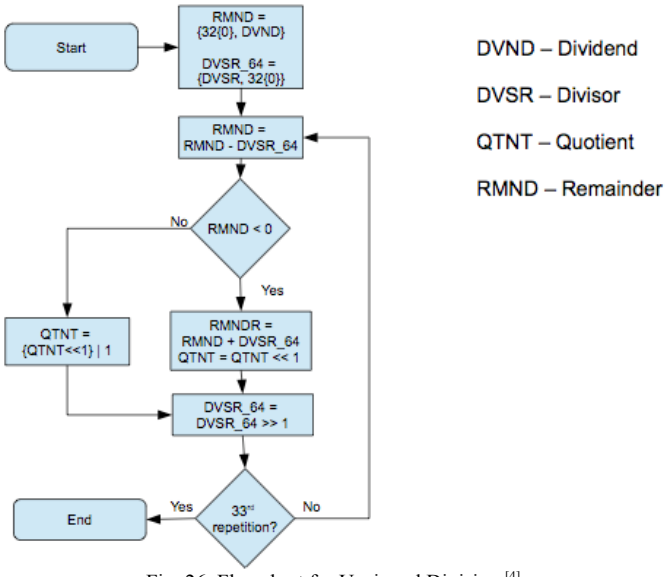

Fig. 26. Flowchart for Unsigned Division [4]

To mimic this in the software as seen in Fig. 28, a loop will be created to run the necessary commands 32 times. For mirroring the 64-bit register, shift the contents in the remainder register to the left to allow insertion of the most significant bit (31st) of the quotient register into its 0th position. Then, shift the contents of the quotient register to the left. Eventually, the dividend will be shifted out to hold both the quotient and the remainder instead. Note that the register holding the dividend and the register holding the quotient is the same one.

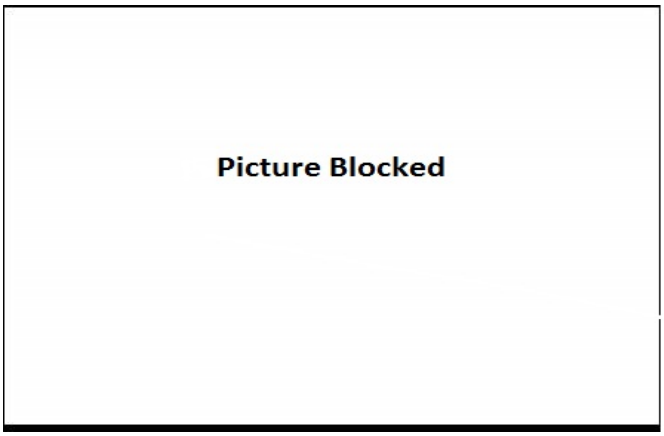

Fig. 27. Implementation of Loop in Unsigned Division

10) div_signed

Likewise, this procedure will be used to perform division for all arguments (signed and unsigned). It follows similar steps taken in the mul_signed procedure.

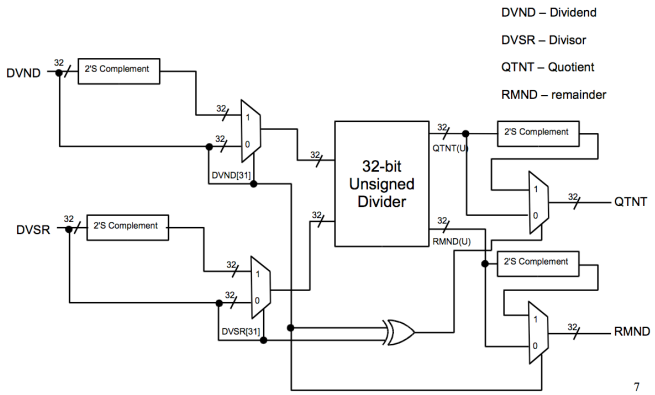

Fig. 28. Logical Design of Signed Division [4]

The operands will be tested for their sign and converted to their corresponding two's complement form if they are negative. They will be passed to div_unsigned for unsigned division, where the quotient and remainder are left in registers $v0 and $v1 respectively. After, the signs of the quotient and remainders are determined by looking at the original arguments' signs. Like mul_signed, the quotient's sign is determined by the XOR-ing of the signs from the original operands and converted to its two's complement form if necessary (see Table II). The original dividend's sign determines the sign of the remainder.

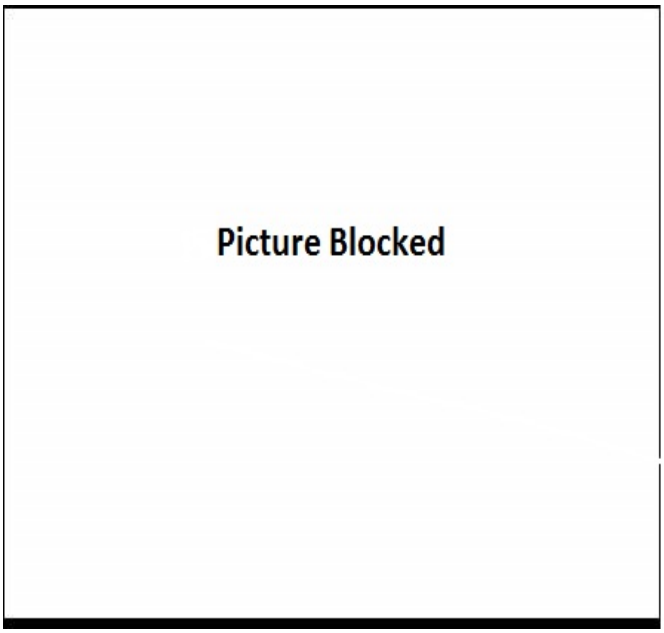

Fig. 29. Implementation of Signed Division

C. Macros

1) Extracting an Individual Bit

All procedures involve determining what value is in a certain bit position; therefore, creating this macro would be useful and convenient.

A technique called masking will be used, where a register will hold the value 1 shifted left into the same position as the desired value to be extracted from the bit pattern. By conducting the AND operation between the bit pattern and the

mask, the result will contain the value of the desired bit. Then, it should be shifted back right by the same initial amount to obtain the bit value (either 0 or 1).

Fig. 30. Implementation of Single Bit Extraction

2) Inserting into a Bit Position

Likewise, all procedures involve placing a specific value in a certain bit position; therefore, creating this macro would also be useful and convenient.

Masking will be used again. A masking register will hold the value 1 shifted left into the same position as the one that will hold the inserted bit in the pattern. Invert the mask and do the AND operation with the original bit pattern. This results in the same bit pattern with a 0 in the desired slot. Another register will hold the desired value to be inserted shifted left into the desired position. By conducting the OR operation between this register and the previous result, the value will be inserted into the correct position.

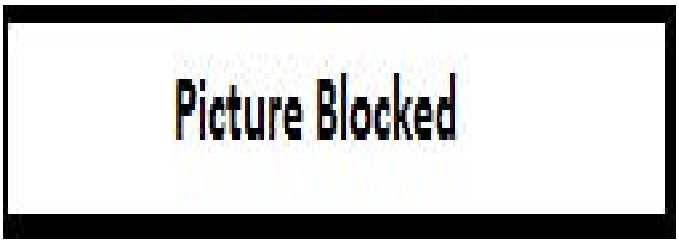

Fig. 31. Implementation of Single Bit Insertion

D. Putting it All Together

The relationships between the procedures and its operations or corresponding procedures can be seen in Fig. 32 and 33. The arrows in the normal procedure indicate branches.

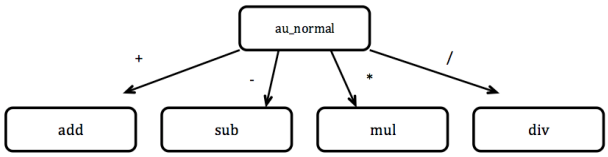

Fig. 32. Normal Procedure and its Operations

In contrast, the arrows in the logical procedure indicate the existence of a relationship between the calling procedure and the called procedure.

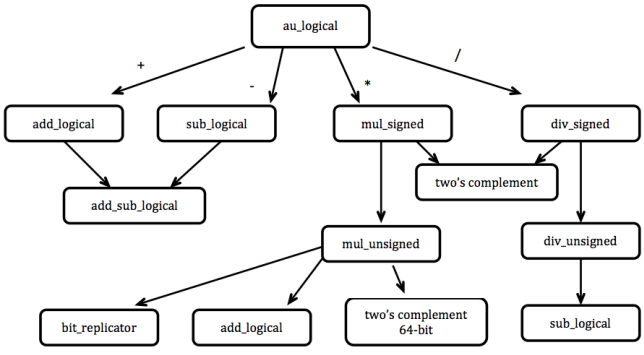

Fig. 33. Logical Procedure and its Corresponding Procedures

V. TESTING

To test the procedures, all files must be saved. Assemble the proj_auto_test.asm (which should not be altered), and run the program by clicking on the "Run" button in the icons menu. The purpose of creating both an au_normal and au_logical procedure is to match their results since they should both be giving the same answers. Should the procedures run correctly and properly, the calculations of 40 different mathematical expressions (containing both signed and unsigned numbers) by both au_normal and au_logical should pass with a 40/40:

```
(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]
(-2 - -8)       normal=> 6      logcal=> 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    logcal=> -12    [matched]
(-6 - -6)       normal=> 0      logcal=> 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      logcal=> 0      [matched]
(-18 - 18)      normal=> -36    logcal=> -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     logcal=> -3     [matched]
(5 - -8)        normal=> 13     logcal=> 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    logcal=> -16    [matched]
(-19 - 3)       normal=> -22    logcal=> -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      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 --
```

Fig. 34. Test Results

Common possible errors when completing the procedures include:

1) The use of temporary registers

Use temporary registers with caution because they are not guaranteed to be preserved within a procedure. This is a problem when procedures are called within other procedures. Should the same temporary register be used in both the calling and called procedure, the calling procedure will lose any value held in that register after it calls the procedure that reuses it. When using temporary registers, it is best to use unique registers in certain procedures instead of reusing them to avoid this problem.

2) Storing and restoring frame

Storing and restoring frame is very important; failure to do so correctly results in infinite loops, runtime exceptions, or "invalid program counter value". Common mistakes include not saving saved registers in the frame and moving the frame pointer instead of the stack pointer when restoring the frame.

3) Omitting jr $ra

Omitting this line after restoring the frame will result in 40/40 in testing because the procedures will instead be

calculated through au_normal (given that au_normal works properly). That, however, does not fulfill the requirement and should be modified so that the procedure returns to the caller's next instruction.

4) Registers holding results

Any operations done in a procedure can be done using any register. However, there are only certain designated registers ($v0 and/or $v1) that should hold the end result. Therefore, it must be noted that any register used to store the result besides the designed ones should store their contents into these result registers. Without doing so, it is possible that the procedures are correct, but the correct results do not show up because they are not available to be fetched from the result registers.

VI. CONCLUSION

By completing this project, I learned a lot about MARS and MIPS operations. The protocols (procedures, frame storing, branches, commands) are all things I have become familiar with. I have come upon many different errors, which has enabled me to learn and figure out their reasons. This project has shed some light on all the work a computer must do for a simple calculation. As humans, we are used to very simple commands, such as adding and subtracting, with calculators and in our minds; however, we don't seem to realize that devices need to go through a more complicated process to do those exact calculations. Something we think as easy and trivial is actually the opposite for a computer, and this project has allowed me to go through that process. It was interesting and enlightening to do so.

REFERENCES

[1] D. A. Patterson and J. L. Hennessy, Computer Organization and Design (5th Edition). Waltham, MA: Morgan Kaufman, 2013, pp. 178-195.

[2] K. Patra. CS 47. Class Lecture, Topic: "Addition Subtraction Logic." San Jose State University, San Jose, CA, April 14, 2016.

[3] K. Patra. CS 47. Class Lecture, Topic: "Multiplication Logic." San Jose State University, San Jose, CA, April 19, 2016.

[4] K. Patra. CS 47. Class Lecture, Topic: "Division Logic." San Jose State University, San Jose, CA, April 21, 2016.