Arithmetic Operations Implemented Using MIPS Fundamental Logic and Design

Diljot Singh

San Jose State University

diljot.singh@sjsu.edu

***Abstract* – write this last, summarizing what you’ve said and done through this project / report**

1. INTRODUCTION

MIPS (Microprocessor without Interlocked Pipelined Stages) is an instruction architecture which supports what is known as “MIPS Assembly Language.” This assembly language gets translated to machine code through an assembler, where it can then be executed by the computer. In this project, we will be using a tool known as “MARS MIPS simulator,” an integrated development environment which supports the MIPS processor to simulate the functionality of the arithmetic logic unit (ALU) in the processor.

Specifically, we will be implementing mathematical operations such as addition, subtraction, multiplication, and division through both their normal implementations (calls to MIPS library) and their logical implementations (using bitwise operations and Boolean logic). The project objectives are listed below:

1. Install and setup the MARS simulator environment.
2. Design and implement arithmetic calculations using both built in MIPS operations (such as add, sub, mult, div) and logical operations (Boolean logic & algebra and bit manipulation).
3. Test both implementations against each other in the MARS IDE to ensure full functionality and computability.
4. Better understand the computer architecture system components both holistically and individually.
5. REQUIREMENTS (Installation and Setup)
6. *Installation of tools and source files*

1) The MARS MIPS Simulator environment can be downloaded online at <https://courses.missouristate.edu/KenVollmar/MARS/download.htm>. There may be additional requirements such as having the Java SDK (software development kit) installed, which are mentioned on the MARS installation page.

2) The project file structure is provided through a .zip file and can be downloaded through Canvas at <https://sjsu.instructure.com/courses/1324477/files/54503907/download?wrap=1>

Move the file into a desired directory and unzip it. Verify that it contains the following files:

1. cs47\_common\_macro.asm
2. cs47\_proj\_alu\_logical.asm
3. cs47\_proj\_alu\_normal.asm
4. cs47\_proj\_macro.asm
5. cs47\_proj\_procs.asm
6. proj-auto-test.asm
7. *Launching the project*

Launch MARS and navigate to “File” (located in the upper left-hand corner).

A screenshot of a cell phone

Description automatically generated

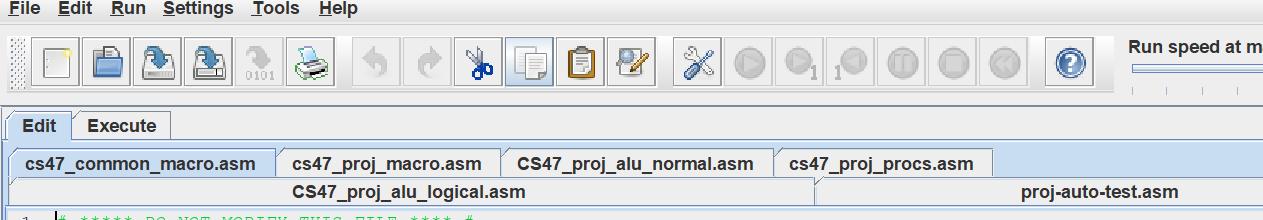
*Fig 1. Opening files (1)*

Click on “File” and then click “Open.” This will display various directories and file structures; navigate to where the unzipped files are saved on the user’s computer system. Select one of the files and then click open (in the bottom right-hand corner). A screenshot of a cell phone

Description automatically generated

*Fig 2. Opening files (2)*

This will redirect the user back to the MARS environment with the selected file now open.

Repeat procedure until all project files have been opened, as shown below:

*Fig 3. Project file access*

As depicted above, each file tab can be selected to be displayed in the MARS environment. The currently selected file (in this case, cs47\_common\_macro.asm) is highlighted blue while the other tabs are not. The three main files we will be working with are:

1. cs47\_proj\_alu\_logical.asm

*This file will contain the logical implementation of the mathematical operations.*

1. cs47\_proj\_alu\_normal.asm

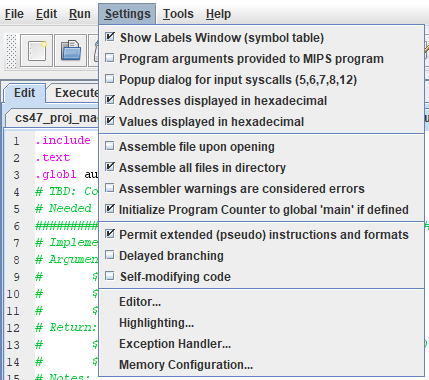
*This file will contain the normal implementation (MIPS instruction set) of the mathematical operations.*

1. cs47\_proj\_macro.asm

*This file will contain any additional macros needed to assist in both implementations.*

1. *Initialization*

**Note:** In MARS settings, make sure the boxes next to “Assembles all files in directory” and “Initialize program counter to global main if defined” are checked.



*Fig 4. Initializing settings*

1. REQUIREMENTS OF ARITHMETIC PROCEDURES

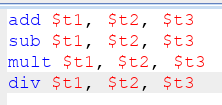
As mentioned, there will be two implementations of the mathematical operations: normal and logical. The normal procedure will utilize the built-in MIPS instruction set and the logical procedure will incorporate Boolean logic gates and bit manipulation.

1. *Normal procedure*

The normal procedure will be implemented in the cs47\_proj\_alu\_normal.asm file. The procedure will take three arguments:

1. Register $a0 – The first operand in the mathematical expression.
2. Register $a1 – The second operand in the mathematical expression.
3. Register $a2 – The operator (“+”, “-“, “\*”, “/”), indicating which operation is to be performed.

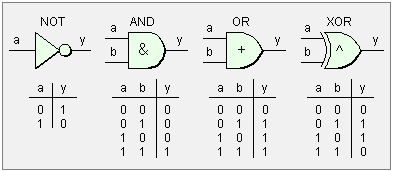
To compute the result of each mathematical operation, the MIPS instruction set can be used. For example, to add two numbers in the normal procedure, one can simply type “add \_, \_, \_,” which will call the built in add instruction to compose the sum of the last two arguments and store it into the first argument.

  *Fig 5. MIPS Instruction Set call example*

For addition and subtraction, the results should be stored in the $v0 register. For multiplication, the low order of 32 bits (Lo) should be stored in $v0 and the high order of 32 bits (Hi) should be stored in $v1. For division, the quotient should be stored in $v0 and the remainder in $v1.

1. *Logical procedure*

The logical procedure will be implemented in the cs47\_proj\_alu\_logical.asm file. In contrast with the normal procedure, the MIPS instruction set cannot be used to directly generate the result for this implementation. Instead, we must use logical operations such as AND, OR, NOT, and XOR.



*Fig 6. Logical Gates and their Truth Tables*

In addition, we are also permitted to use bit manipulation through macros, shifts, and loops. The logical procedure will also take three arguments:

1. Register $a0 – The first operand in the mathematical expression.
2. Register $a1 – The second operand in the mathematical expression.
3. Register $a2 – The operator (“+”, “-“, “\*”, “/”), indicating which operation is to be performed.

Due to the added complexity of the logical implementation, multiple procedures will be needed to supplement the mathematical computations, which will be discussed in the next section.

1. DESIGN AND IMPLEMENTATION
2. *Macros*

Before delving into the implementations, let us discuss the macros utilized in the “cs47\_proj\_macro.asm” file.

Macros, short for “macroinstruction,” contains a specific sequence of instructions which can be called using the macro’s name. This allows users to save both space and time, especially for extremely repetitive or extensive segments of code. Rather than typing the same instructions over and over again, the programmer can call the macro which contains those instructions.

The macros utilized for this project’s implementations are:

1. store\_frame

A screenshot of a cell phone

Description automatically generated

*Fig 7. Store\_frame macro*

Stack frames are used in MIPS to preserve memory and prevent data loss as a result of subroutine calls in procedures. It is a good implementation design to store the frame first in every procedure in order to allocate memory for registers which may be needed across multiple calls. In this macro, we are saving $sp (stack pointer), $fp (frame pointer), $ra (return address), all argument registers ($a0 - $a3), and all saved registers ($s0 - $s7). By simply typing “store\_frame” (the name of the macro), all these instructions are executed.

1. restore\_frame

A screenshot of a social media post

Description automatically generated *Fig 8. Restore\_frame macro*

Similar to the store\_frame macro, we create a “restore\_frame” macro which will be called at the end of every procedure. This restores the stack space allocated when we called the store\_frame macro.

1. extract\_nth\_bit

A screenshot of a cell phone

Description automatically generated *Fig 9. Extract\_nth\_bit Macro*

The extract\_nth\_bit macro is used to access specific bits in a bit pattern. It has three arguments:

* $regD – the bit we extract
* $regS – the source bit pattern
* $regT – the bit position (0-31)

We will use this macro extensively in our alu\_logical implementation to access specific bits and perform operations on them.

1. insert\_to\_nth\_bit

A screenshot of a cell phone

Description automatically generated *Fig 10. Insert\_nth\_bit Macro*

While the extract\_nth\_bit macro is used to access specific bits from patterns, the insert\_nth\_bit macro is used to insert bits into patterns at the nth location. This macro takes four arguments:

* $regD – the bit pattern in which insertion will take place
* $regS – the bit position n where we will insert
* $regT – the register that will contain the bit we want to insert
* $maskReg – a temporary mask register used to perform the insertion

Again, while this macro is not needed in the alu\_normal implementation, we use it extensively in the alu\_logical implementation to modify bit patterns.

1. *Normal Procedures*

The normal procedures are relatively straight forward in terms of the implementation. First, we start off with four branch statements which test the $a2 (operator) register. Based on which statement is true, the program will redirect to that procedure accordingly.

A screenshot of a cell phone

Description automatically generated

*Fig 11. Normal Procedure Branches*

We will now explore each procedure implementation individually.

1. Addition Implementation

A screenshot of a cell phone

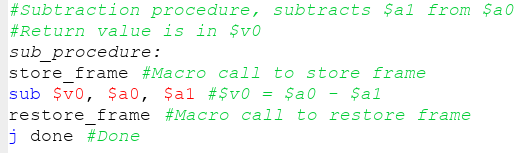
Description automatically generated *Fig 12. Normal Addition Implementation*

The normal addition procedure is implemented as follows:

* Store the frame (standard in any procedure call)
* Use the ‘add’ instruction discussed earlier to compute $a0 + $a1, storing that computation in $v0
* Restore the frame (standard in any procedure call)
* A jump to the ‘done’ label, which simply goes to the end of the program, signifying completion

**Note:** The “store\_frame” and “restore\_frame” macros are discussed earlier.

1. Subtraction Implementation



*Fig 13. Normal Subtraction Implementation*

Similar to the addition implementation, the subtraction procedure is implemented as follows:

* Store the frame (standard in any procedure call)
* Use the ‘sub’ instruction discussed earlier to compute $a0 - $a1, storing that computation in $v0
* Restore the frame (standard in any procedure call)
* A jump to the ‘done’ label, which simply goes to the end of the program, signifying completion

1. Multiplication Implementation

A screenshot of a cell phone

Description automatically generated

*Fig 14. Normal Multiplication Implementation*

For the normal multiplication procedure, the implementation is as follows:

* Store the frame
* Use the ‘mult’ instruction to compute $a0 \* $a1, which automatically stores the high-order 32 bits in the “Hi” register and the low-order 32 bits in the “Lo” register
* We must use special instructions to access Hi and Lo since these are special registers. We use “mflo” (move from lo) to move the contents of the Lo register into $v0
* We use “mfhi” (move from hi) to move the contents of the Hi register into $v1
* Restore the frame
* Jump to the done label

1. Division Implementation

A screenshot of a cell phone

Description automatically generated *Fig 15. Normal Division Implementation*

For the normal division procedure, we implement it as follows:

* Store the frame
* Call the “div” instruction from MIPS instruction set. This instruction automatically stores the remainder in the Hi register and the quotient in the Lo register
* Use “mflo” to move the quotient into the $v0 register
* Use “mfhi” to move the remainder into the $v1 register
* Restore the frame
* Jump to the done label

1. *Logical Procedures*

Similar to the normal procedures, we include the branch statements to determine which procedure the program should execute based on the operator in the $a2 register.

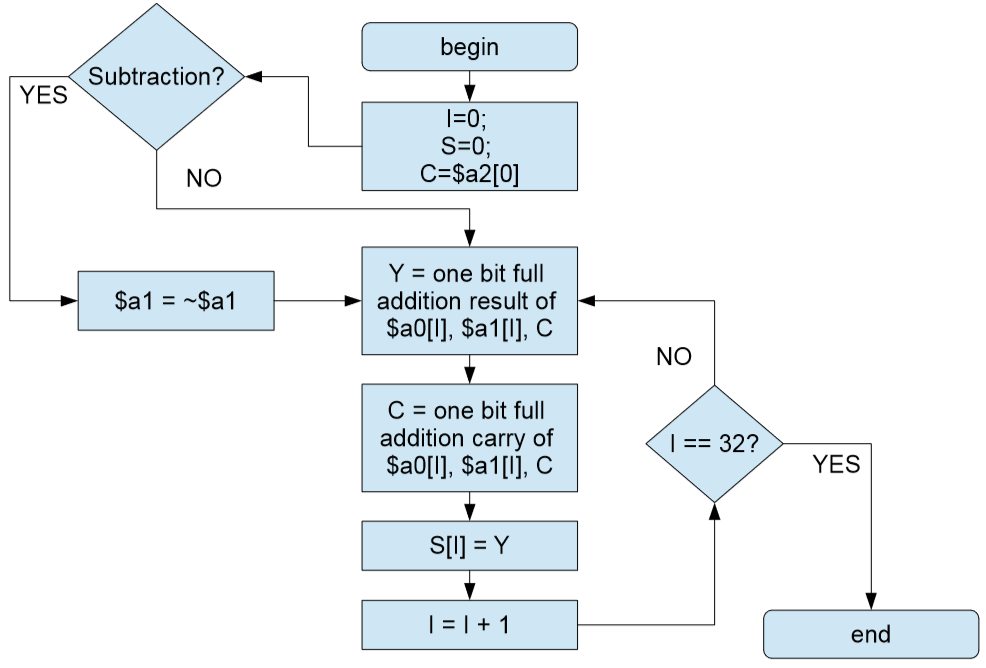
A screenshot of a cell phone

Description automatically generated

*Fig 16. Logical Branch statements*

Before continuing to the logical implementations however, it is necessary to discuss the utility procedures which are the essence of the logical procedures. These utility procedures are called extensively throughout the alu\_logical implementations and perform functions such as computing negatives, performing the mathematical operation logic, and controlling the program flow.

1. **Utility**: Common add\_sub\_logical



*Fig 17. Add\_sub\_logical flowchart*

The add\_sub\_logical utility procedure is used in both the add\_logical and sub\_logical procedures. The idea of the procedure is that it uses a loop to compute the sum’s and carry’s of every bit until every bit has been accessed. Following the flow chart in Fig 16., we can construct a implementation as follows:

A screenshot of text

Description automatically generated

A screenshot of text

Description automatically generated *Fig 18. Add\_sub\_logical implementation*

This utility procedure has three arguments: $a0 – the first operand, $a1 – the second operand, and $a2 – the mode (either addition or subtraction). If the mode is addition, then the $a2 register will be equal to 0x00000000, according to our convention. On the contrary, if the $a2 register contains 0xFFFFFFFF, this signifies subtraction mode. The only difference between the addition and subtraction modes are that in subtraction we invert $a1 and proceed as normal (since A – B = A + (-B)). Therefore, our implementation can support both addition and subtraction.

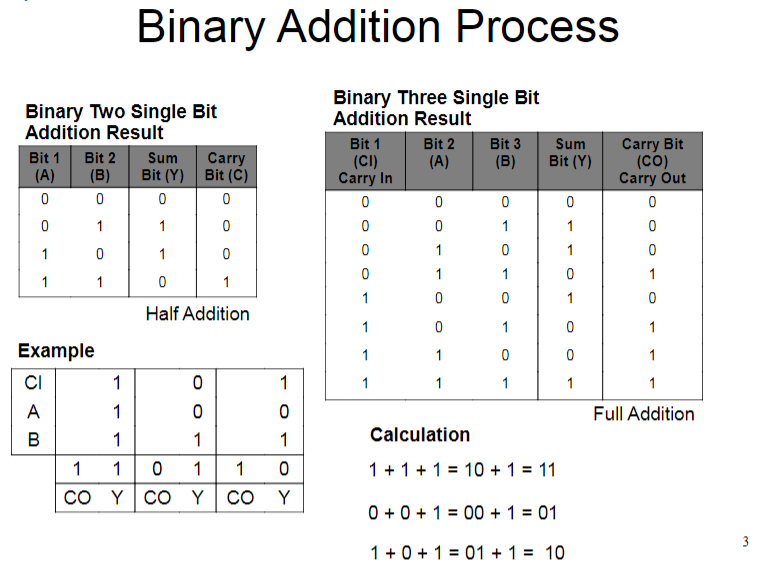
We extract one bit from both operands ($a0 and $a1) starting from the LSB side. By performing an XOR between these two bits, we are essentially performing a sum and trying to compute the carry. We then perform another XOR between the carry and the two bits, following with two AND’s and one OR to get the final carry. Once this is obtained, we insert the sum to the $s0 register (which is storing the sum) and continue looping. This implementation will sum every corresponding bit one at a time and move their sums into the $s0 register. Once $t0 (the loop counter) is equal to 32, that signifies that we have accessed every bit and we are done. Once the loop has completed, we will set $v0 (the return value) to the sum we calculated in our loop ($s0). In addition, the final carryout computed will be saved in $v1 for additional functionality. *Fig 18. Binary addition process*

Figure 18 depicts the process of computing the carries and sums of every bit using both a half adder and full adder.

1. add\_logical

Once the implementation of the common add\_sub\_logical has been completed, all the add\_logical has to do is simply make a call to the utility procedure with the mode set to addition, as depicted in Fig 19.

A screenshot of a cell phone

Description automatically generated *Fig 19. add\_logical procedure*

In add\_logical, the steps we take are as follows:

* Store the frame
* Load the mode register ($a2) with the immediate value 0x00000000, signifying addition.
* Jump and link (‘jal’) to the common add\_sub\_logical which will compute the sum and return it in register $v0
* Restore the frame

1. sub\_logical

Similar to add\_logical, the sub\_logical utilizes the common add\_sub\_logical utility procedure created.

A screenshot of a cell phone

Description automatically generated *Fig 20. sub\_logical procedure*

The steps we take in sub\_logical are as follows:

* Store the frame
* Load the mode register ($a2) with the convention we set for subtraction: 0xFFFFFFFF
* Jump and link to the add\_sub\_logical utility procedure which will return the total in $v0
* Restore the frame

A close up of text on a white background

Description automatically generated *Fig 21. Combination of adder / subtractor*

1. **Utility:** twos\_complement

This utility procedure is used to compute the 2’s complement of a number. What is special about 2’s complement is that it can be used to represent both negative and positive numbers in binary by denoting the MSB to be a sign indicator. If the MSB of a 2’s complement number is 0, it is positive, and if it is 1, it is negative. We implement the twos\_complement as such:

A screenshot of a cell phone

Description automatically generated *Fig 22. Twos\_complement utility procedure*

In this procedure, we are receiving one argument: $a0, which is the number we are trying to compute the complement of. The complement of the number is returned in the register $v0. The steps we follow are:

* Compute the inverse of our input (Set $a0 to ~$a0).
* Load the $a1 register with the immediate value “1” for our next procedure call
* Call the add\_logical procedure to compute ~$a0 + 1. This effectively calculates the 2’s complement and stores it in register $v0
* Restore the frame

1. **Utility:** twos\_complement\_if\_neg

This utility procedure is a slightly tweaked version of our earlier twos\_complement. Now, the 2’s complement is only computed if the input argument $a0 is negative.

A screenshot of a cell phone

Description automatically generated *Fig 23. Twos\_complement\_if\_neg utility procedure*

The only difference is that we have added a conditional check to see if $a0 is negative: “bltz $a0, twos\_complement\_without\_save.” What this statement is signifying is that if the $a0 register is less than zero, only then do we call the twos\_complement procedure. Otherwise, we simply move the $a0 argument into $v0 right away.

1. **Utility:** twos\_complement\_64bit

While computing the 2’s complement of a single number is relatively simple, it is not as easy for a 64 bit value. This is because the value will be stored in two separate registers, such as Hi and Lo, and the 2’s complement should be of the value as a whole. However, this can be still done through a series of procedure calls as shown in the implementation in Fig 24.

A close up of a piece of paper

Description automatically generated

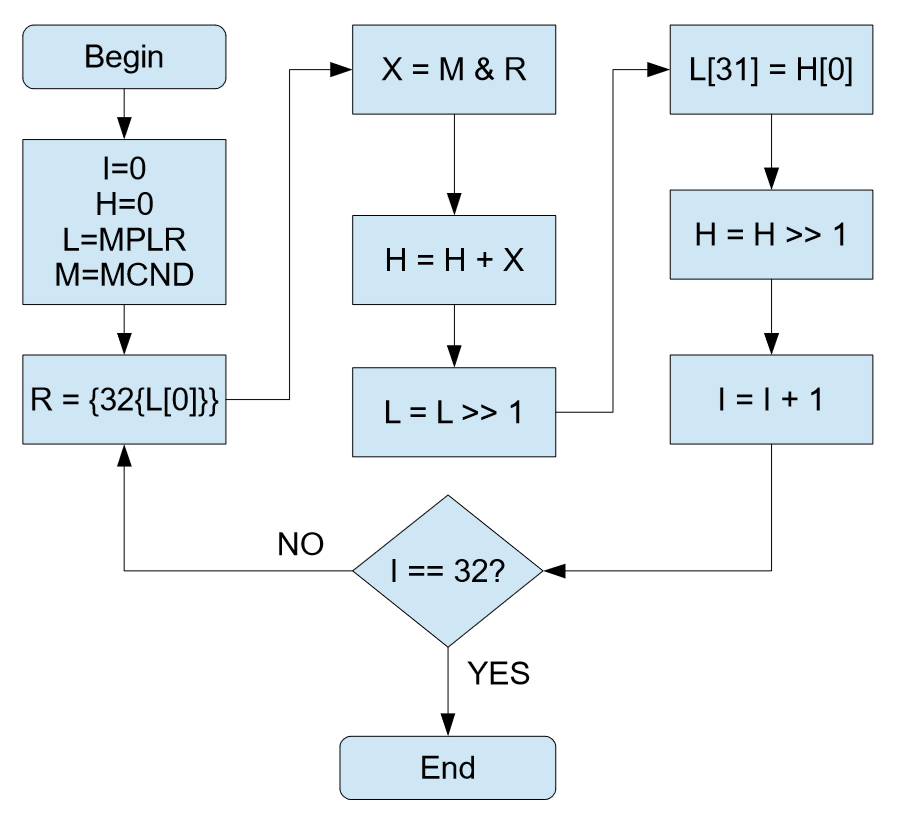
*Fig 24. Twos\_complement\_64bit*

In this procedure we compute the 64 bit 2’s complement as follows:

* Invert both $a0 and $a1 (the Lo and Hi of the number, respectively)
* Save the inverted $a1 in $s3
* Load the $a1 register with the immediate value 1
* Calls the add\_logical procedure to compute ~$a0 + 1
* Restore the inverted $a1
* Move the calculated 2’s complement to $v0
* Move the carry to $a0
* Call the add\_logical once more to compute carry + $a1
* Restore the Lo register with its 2’s complement
* Restore the frame

What this implementation does is that it calls the 2’s complement twice using the carry from the add\_logical. This allows both the Hi and Lo registers to be complemented as a whole value rather than each separately added to one another.

TODAY: MULT, DIV, AND OTHER UTILITIES, ADD REFERENCES



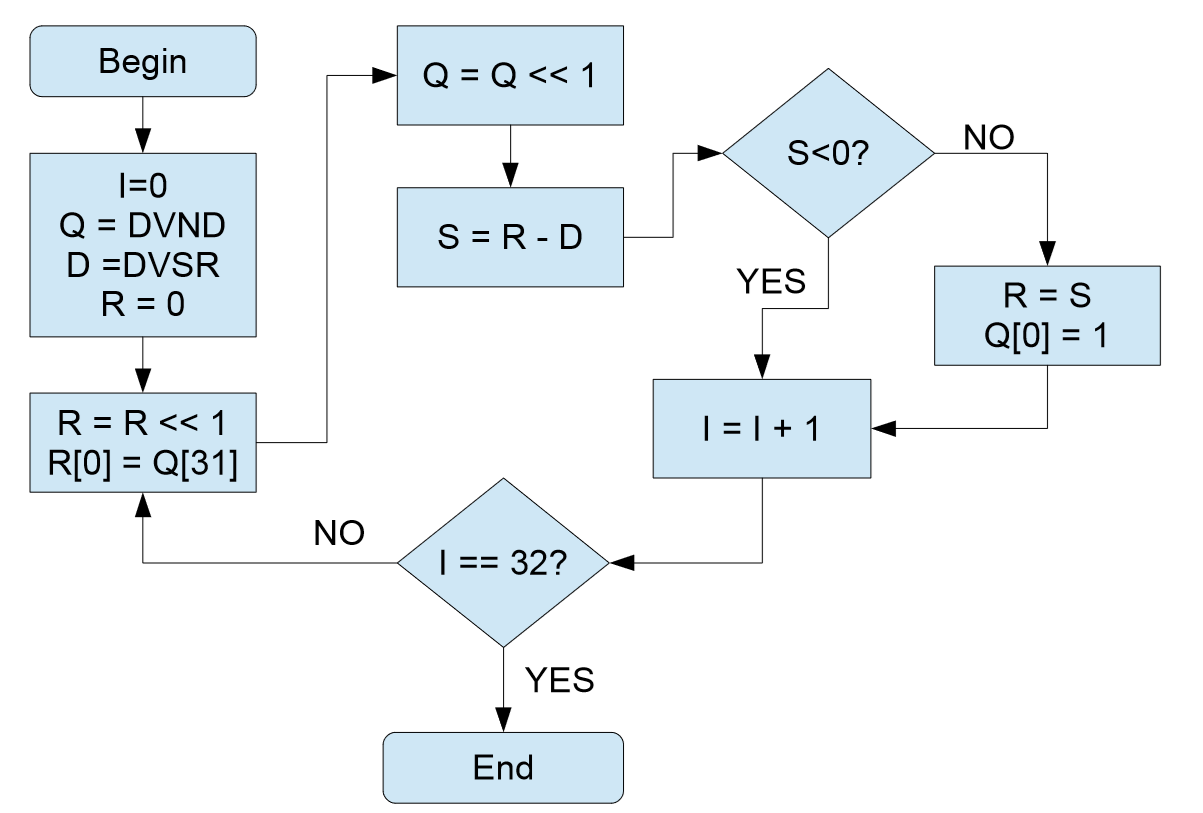


Figure 18: Division Flow Chart

Figure 19: Logical Division Implementation (part 1)

The first part of the division procedure is getting all the operands to be positive by utilizing the utility procedure “twos\_complement\_if\_neg”. Once all operands are positive, the arguments are passed into $a0 and $a1 and proceeds into the div\_unsigned procedure.

Figure 20: Logical Division Implementation (part 2)

The div\_unsigned procedure first initializes the counter to 0, the quotient to the dividend, saves the divisor for modification, and initializes the remainder to 0 and then proceeds into the loop labeled “div\_loop”. First, the remainder is left shifted by 1 and the MSB of the quotient is extracted and inserted into the LSB of the remainder using the extract\_nth\_bit and insert\_to\_nth\_bit macros. The quotient is then left shift by 1.

The remainder and divisor are subtracted and if the result is less than zero, the procedure will jump to the counter\_increment label where the counter is incremented by 1, compared against the constant 32, and exits the loop if it equals to 32, otherwise it jumps to the beginning of div\_loop. If the result of the subtraction is greater than or equal to zero, the remainder is set to the result of the subtraction and the bit 1 is inserted into the LSB of the quotient and then proceeds to the counter\_increment label.

Once the loop reaches 32 times and jumps to the div\_exit label, then the signs are calculated. The below image shows the rules for calculating the signs of a division operation.

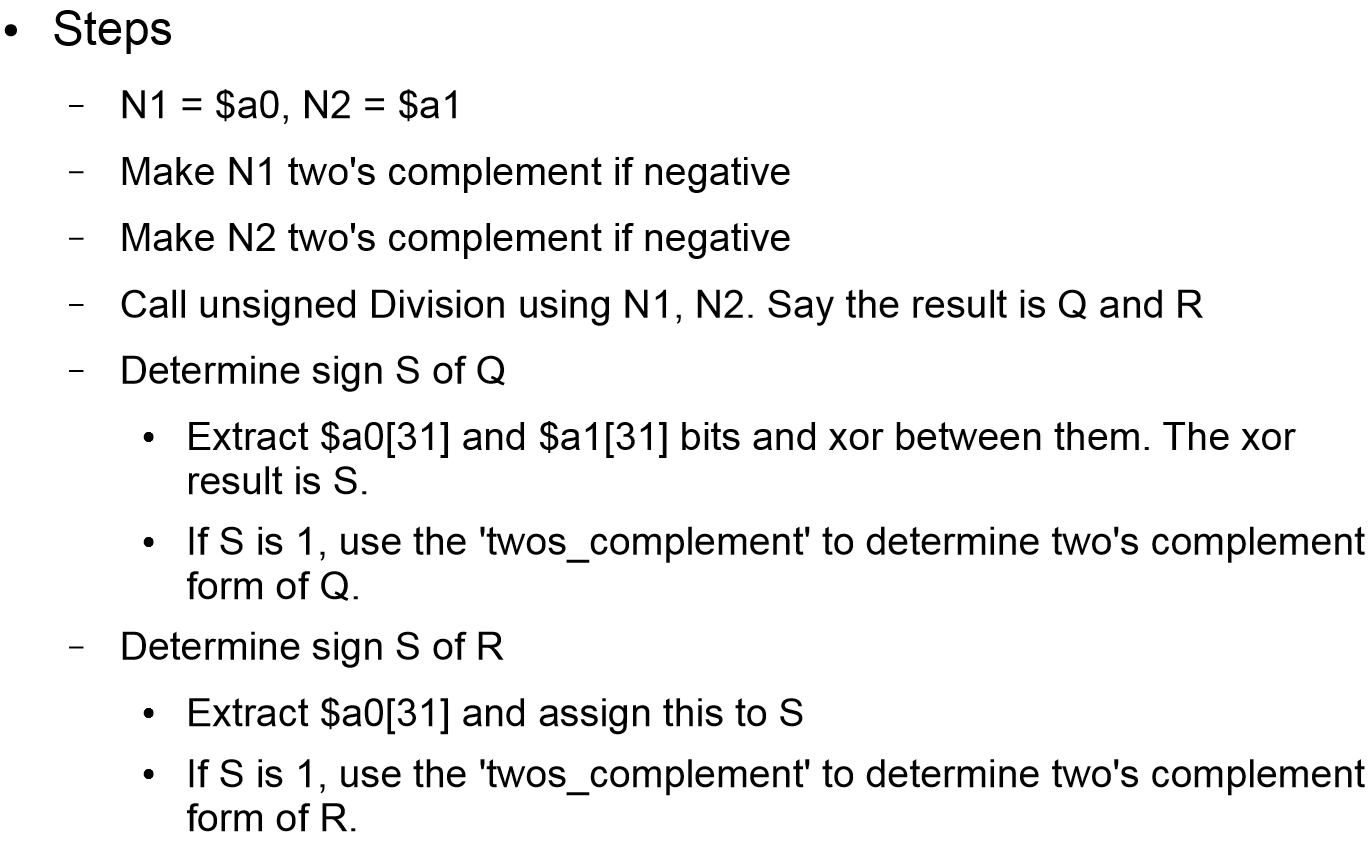


Figure 21: Calculating Sign of Quotient and Remainder

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 MSB of the dividend. If it is 1, then the remainder is its complemented value otherwise it is the current value.

1. TESTING IMPLEMENTATION

After a lot of back and forth with revising my code and debugging, I was finally able to get a 40/40 result on my program. The main problems I ran into were using the same temporary register across procedures and having the values mixed up. I have manually gone through and checked all the values of the output of my program and from my testing see that it is all accurate and correct. I assembled the code in the “proj-auto-test.asm” file which contains the tester code provided by the professor and everything ran perfectly with a 40/40 output. I modified the numbers here and there to further test my output and everything remained fully functional.

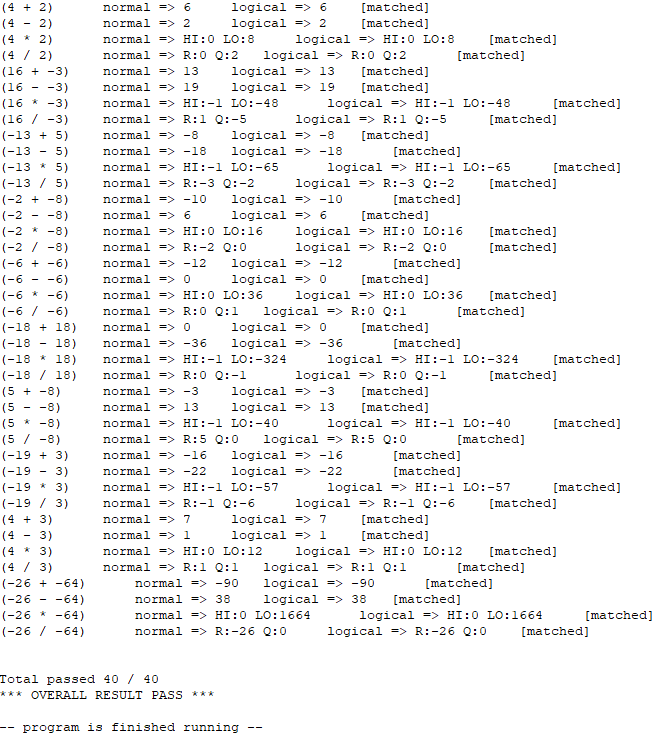


Figure 22: Output of Program

1. CONCLUSION

Overall, this project was a very tough, tedious, and time consuming one. Before learning about the logical operations, I didn’t think a calculator could be created just using AND, OR, NOT, and XOR operations and not using the normal add, subtract, multiply, and divide functions we are so used to. It’s clear that this level of code is underappreciated since this is how the ALU has to be implemented when working with only 0’s and 1’s. I feel like I have a lot better understanding of how the hardware works and the underlying process of arithmetic operations. After completing this project, I feel a lot more comfortable working in MIPS assembly code and even debugging in assembly code.