Arithmetic Operations Implemented Using MIPS Logic and Design

Diljot Singh

San Jose State University

diljot.singh@sjsu.edu

***Abstract* – This project report describes the implementation of arithmetic operations in MIPS assembly language using a) calls to the MIPS instruction set (normal procedure) and b) logical design (logical procedure) through Boolean logic, bit manipulation, logical gates, both conducted through the MARS integrated development environment.**

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 instructions) and their logical implementations (using bitwise operations and Boolean logic). The objectives of this project are to:

1. Install and setup the MARS simulator environment.
2. Implement arithmetic operations using both built in MIPS instructions (such as add, sub, mult, div) and logical operations (logic gates 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 downloaded zip file into a desired directory and unzip it. Verify that it contains the following files:

1. cs47\_proj\_alu\_logical.asm
2. cs47\_proj\_alu\_normal.asm
3. cs47\_proj\_macro.asm
4. cs47\_common\_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), as shown in Figure 1.

A screenshot of a cell phone

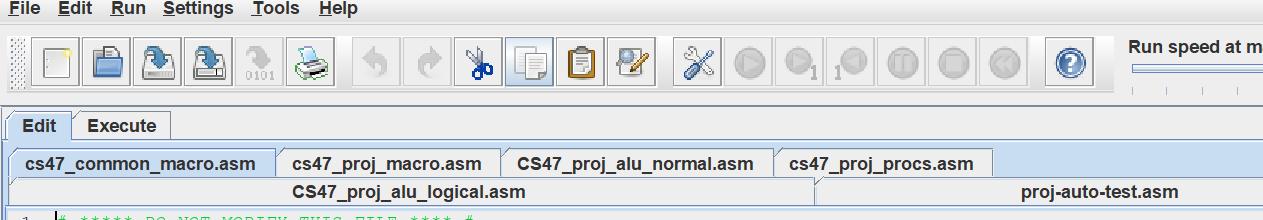
Description automatically generated

*Fig 1. Opening files in MARS (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), as shown in Figure 2.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.*Fig 3. Project file access*

As depicted in Figure 3, 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

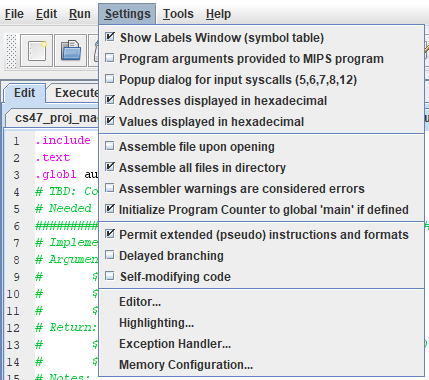
*This file will contain the normal implementation (MIPS instruction set) of the mathematical operations.*

1. cs47\_proj\_macro.asm

*This file will contain any additional macros needed to assist in the 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, as depicted in Figure 4. This will ensure that the program starts where the “main” label is in the code; not doing so could lead to unexpected errors.



*Fig 4. Initializing settings*

1. REQUIREMENTS OF ARITHMETIC PROCEDURES

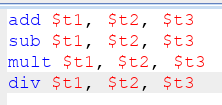
As mentioned, there will be two implementations of the mathematical operations: normal and logical. The normal procedure will utilize the built-in MIPS instruction set and the logical procedure will incorporate Boolean logic gates and bit manipulation. We will discuss both implementations in more detail in the “Design and Implementation” section.

1. *Normal procedure*

The normal procedure will be implemented in the cs47\_proj\_alu\_normal.asm file and is referred to as au\_normal. 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 (addition, subtraction, multiplication, division).

To compute the result of each mathematical operation in the normal procedure, the MIPS instruction set can be used. For example, to add two numbers, 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. Figure 5 displays examples of MIPS instructions utilized in the normal procedures.

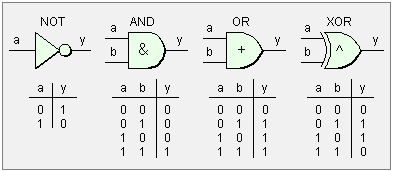
  *Fig 5. MIPS Instruction Set call example*

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 and is referred to as au\_logical. 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.

The truth tables for these logical gates mentioned are shown for reference in Figure 6.



*Fig 6. Logical Gates and their Truth Tables*

We are also permitted to use bit manipulation through macros, shifts, loops, which will be discussed more in the “Design and Implementation” section.

The logical 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.

Due to the added complexity of the logical implementation, multiple procedures will be needed to supplement the mathematical computations, which will also 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,” contain 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, shown in Figure 7, we are storing $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) in any procedure, all of 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, depicted in Figure 8. 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, its implementation depicted in Figure 9. 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, being shown in Figure 10. 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 used 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, as shown in Figure 11.

A screenshot of a cell phone

Description automatically generated

*Fig 11. Normal Procedure Branches*

We will now explore each of the procedure implementations (add\_procedure, sub\_procedure, mul\_procedure, and div\_procedure) individually.

1. Addition Implementation

A screenshot of a cell phone

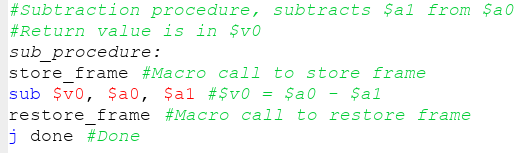
Description automatically generated *Fig 12. Normal Addition Implementation*

The normal addition procedure is shown in Figure 12 and 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, also being shown in Figure 13:

* 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, also shown in Figure 14:

* 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 the instruction “mflo” (move from LO) to move the contents of the Lo register into $v0
* We use the instruction “mfhi” (move from HI) to move the contents of the Hi register into $v1
* Restore the frame
* Jump to the done label

4) 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

The implementation in MIPS is portrayed in Figure 15.

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.

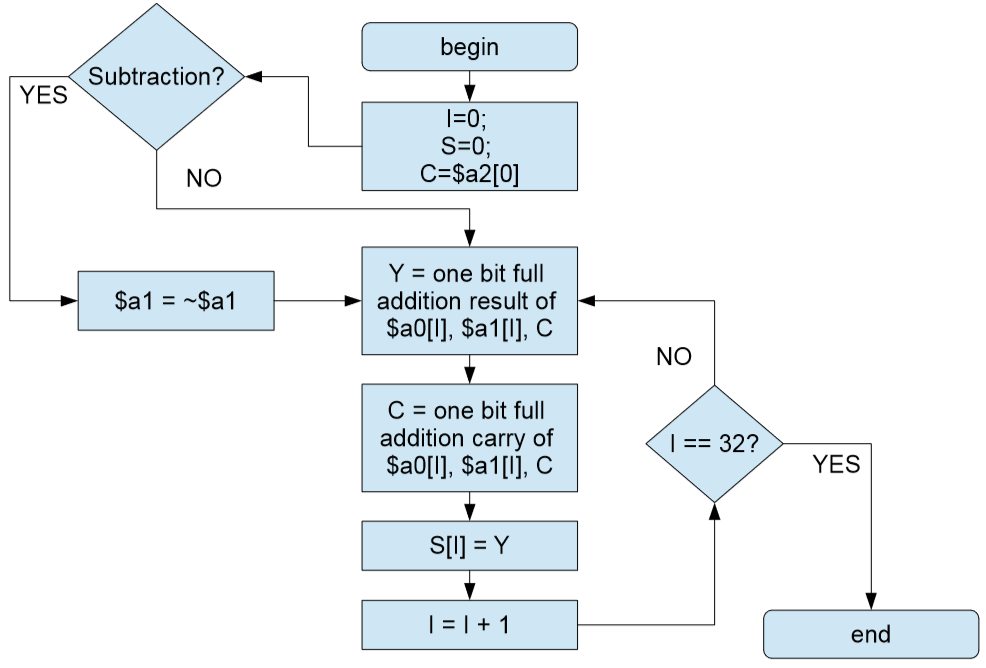
A screenshot of a cell phone

Description automatically generated

*Fig 16. Logical Branch statements*

Before continuing to the logical implementations however, it is necessary to discuss the utility procedures which substantiate the logic 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 sums and carries of every bit until every bit has been accessed. Following the flow chart in Figure 17, we can construct an implementation as follows, shown in Figure 18.

A screenshot of text

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

A screenshot of text

Description automatically generated *Fig 18.1 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 that the procedure is to perform subtraction. The only difference between the addition and subtraction modes are that in subtraction we invert $a1 first before continuing to the implementation (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.

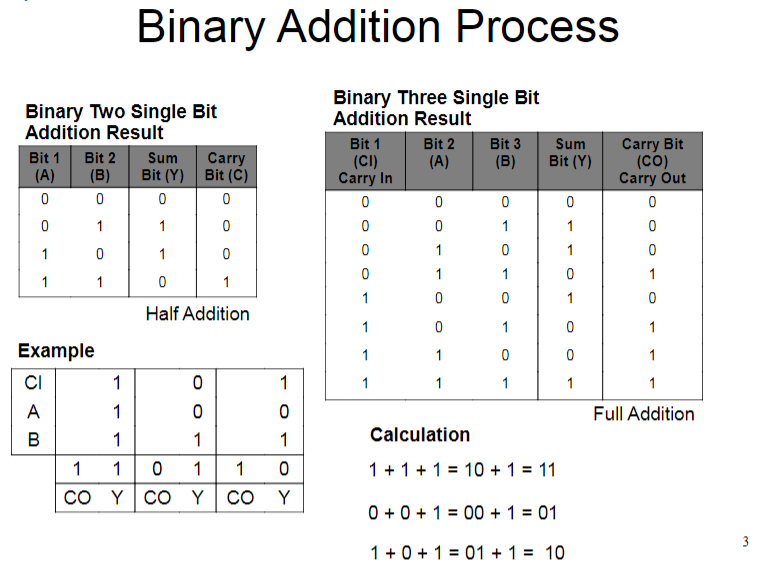
This implementation will summate every corresponding bit one at a time and move their sums into the $s0 register. Once $t0 (the loop counter) is equal to 32, that signifies that we have accessed every bit and we are done. Once the loop has completed, we will set $v0 (the return value) to the sum we calculated in our loop ($s0). Also, the final carryout computed will be saved in $v1 for enhanced functionality in other procedures. *Fig 19. Binary addition process*

Figure 19 depicts the process of computing the carries and sums of every bit using both a half adder and full adder, the same logic being implemented in our common add\_sub\_logical. In particular, we pay close attention to the carry out (CO) and carry in (CI) bits, which we can use to simplify both addition and subtraction, as depicted in Figure 22.

1. **add\_logical**

Once the implementation of the common add\_sub\_logical has been completed (see Figure 18), all the add\_logical must do is simply make a call to the utility procedure with the mode set to addition, as depicted in Fig 20.

A screenshot of a cell phone

Description automatically generated *Fig 20. 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 21. 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 22. Combination of adder / subtractor*

Figure 22 portrays the simplification of the adder and subtractor digital circuits through their combination. We can see how through designating registers for the carry in (CI), carry out (CO), and input/output, both subtraction and addition can be designed, which is what we strove to achieve through our common add\_sub\_logical procedure.

1. **Utility:** twos\_complement

This utility procedure is used to compute the 2’s complement of a number. 2’s complement is used to represent both positive and negative numbers in binary. If the MSB of a 2’s complement number is 0, the number is positive, and if the MSB is 1, the number is negative. We implement the twos\_complement as such:

A screenshot of a cell phone

Description automatically generated *Fig 23. 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 24. Twos\_complement\_if\_neg 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 signifies is that if the $a0 register is less than zero, only then do we call the twos\_complement procedure created earlier. 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 the case 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 computed considering the entire value, not two separate values. However, this can be still done through a series of procedure calls, as shown in the implementation in Figure 25.

A close up of a piece of paper

Description automatically generated*Fig 25. 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 register $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 while being treated as a single value, due to the fact we are obtaining the carry from the Lo register and using this to complement the Hi register.

1. **Utility:** bit\_replicator

The purpose this utility procedure is to replicate a given bit thirty-two times. It receives one argument: $a0, which is the bit we want to replicate (either 0x0 or 0x1). We should store the replicated pattern in $v0, which will be 0x00000000 if $a0 = 0x0, or 0xFFFFFFFF if $a0 = 1.

The implementation is as follows:

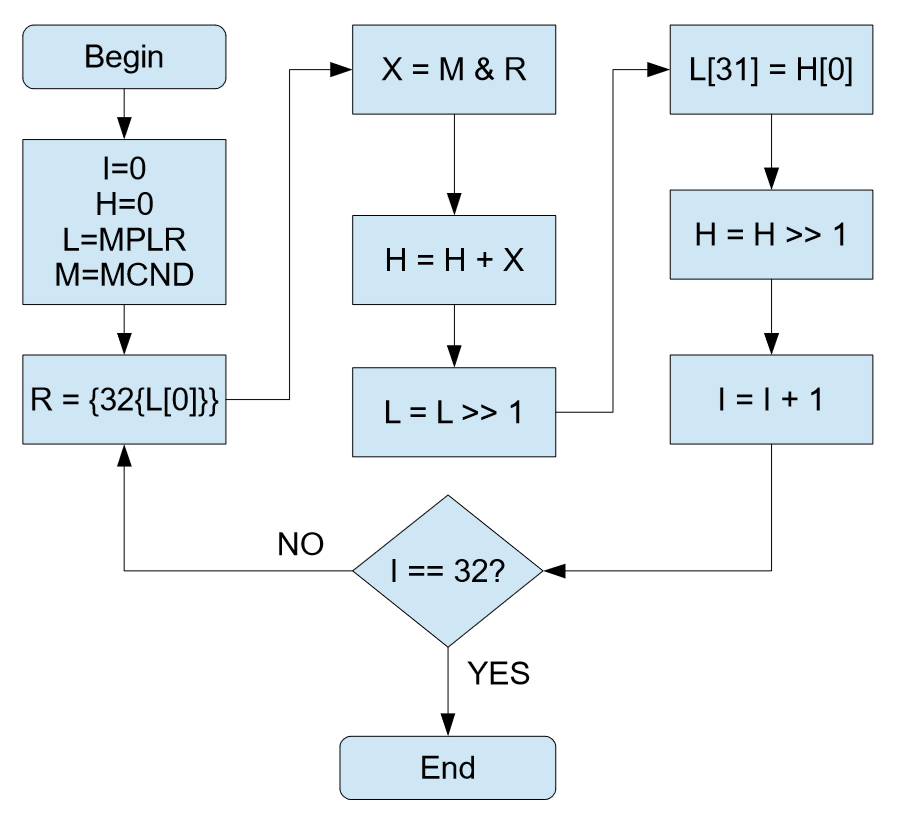
A screenshot of a cell phone

Description automatically generated *Fig 26. Bit\_replicator*

As shown in Figure 26, this procedure is created using a branch statement redirecting to the corresponding replication code. If the $a0 is detected as being equal to zero, we branch to the zero\_rep label and load the $v0 register with the immediate value 0x00000000. Otherwise, we load the $v0 register with the value 0xFFFFFFFF.

1. **Utility:** mul\_unsigned

Mul\_unsigned is used to compute the result of multiplying two unsigned numbers. It takes two arguments: $a0 (the first operand) and $a1 (the second operand). After the multiplication is complete, it will return the Lo part of the result in $v0 and the Hi component in $v1.

 *Fig 27. Unsigned multiplication algorithm*

Due to the fact that the operands are 32 bits but the result is 64 bits, we must account for both the Hi and Lo registers in our code using right shifts and bit insertions. Using the algorithm flowchart depicted in Figure 27, we create an implementation as such:

A screenshot of a social media post

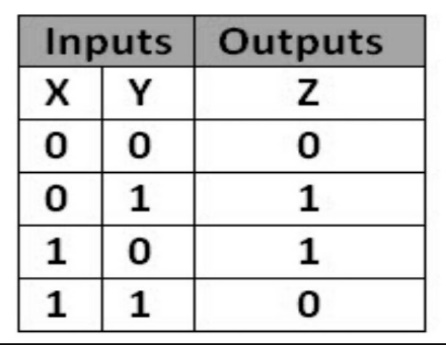
Description automatically generated *Fig 28. mul\_unsigned implementation*

A screenshot of a cell phone

Description automatically generated *Fig 28.1 mul\_unsigned implementation*

Following the algorithm outlined, we can see how the necessary components of decimal and binary multiplication (such as right shifting the multipliers and having addition calls between every product) are present.

Furthermore, by implementing the “jal twos\_complement\_64bit” at the end of the multiplication loop, we can forgo having to create a separate procedure for signed multiplication and can simply do a test to see whether the multiplicand and multiplier were negative or not. This requires the usage of the “XOR” logical operation prior to the complementation, utilizing the fact that the XOR will return a 0 only when the MSB of the multiplier and multiplicand were the same. What this signifies in terms of the 2’s complement in multiplication is that if both the multiplier and multiplicand were negative or positive (had the same MSB), then the product will be positive. However, if and only if there is exactly one negative operand in the procedure, XOR will return 1, signifying a negative product, which is where the “twos\_complement\_64bit” procedure is invoked. The inputs and outputs in relation to the XOR logical operation is shown in Figure 29.

 *Fig 29. XOR Truth Table*

1. **mul\_logical**

Similar to the add\_logical and sub\_logical procedures, the mul\_logical procedure will make a call to the utility procedure “mul\_unsigned” discussed earlier. The implementation is shown in Figure 30 below.A screenshot of a cell phone

Description automatically generated *Fig 30. Mul\_logical procedure*

Due to the fact that we are computing the multiplication of both signed and unsigned numbers in one procedure, we use the utility procedure “twos\_complement\_if\_neg” to convert both operands to be positive, if they are not already so for the time being. Then, we call our mul\_unsigned procedure which will take care of the multiplication logic and complement the numbers at the end if needed. This algorithm design is referenced in Figure 31.A screenshot of a cell phone

Description automatically generated *Fig 31. Signed multiplication algorithm*

1. **Utility:** div\_unsigned

Similar to mul\_unsigned, div\_unsigned will also take two arguments: $a0 (the first operand), and $a1 (the second operand). This procedure assumes the two operands are unsigned and after performing division, it will store the quotient in register $v0 and remainder in $v1.

A close up of a map

Description automatically generated *F­ig 32. Binary Division Logical Circuit*

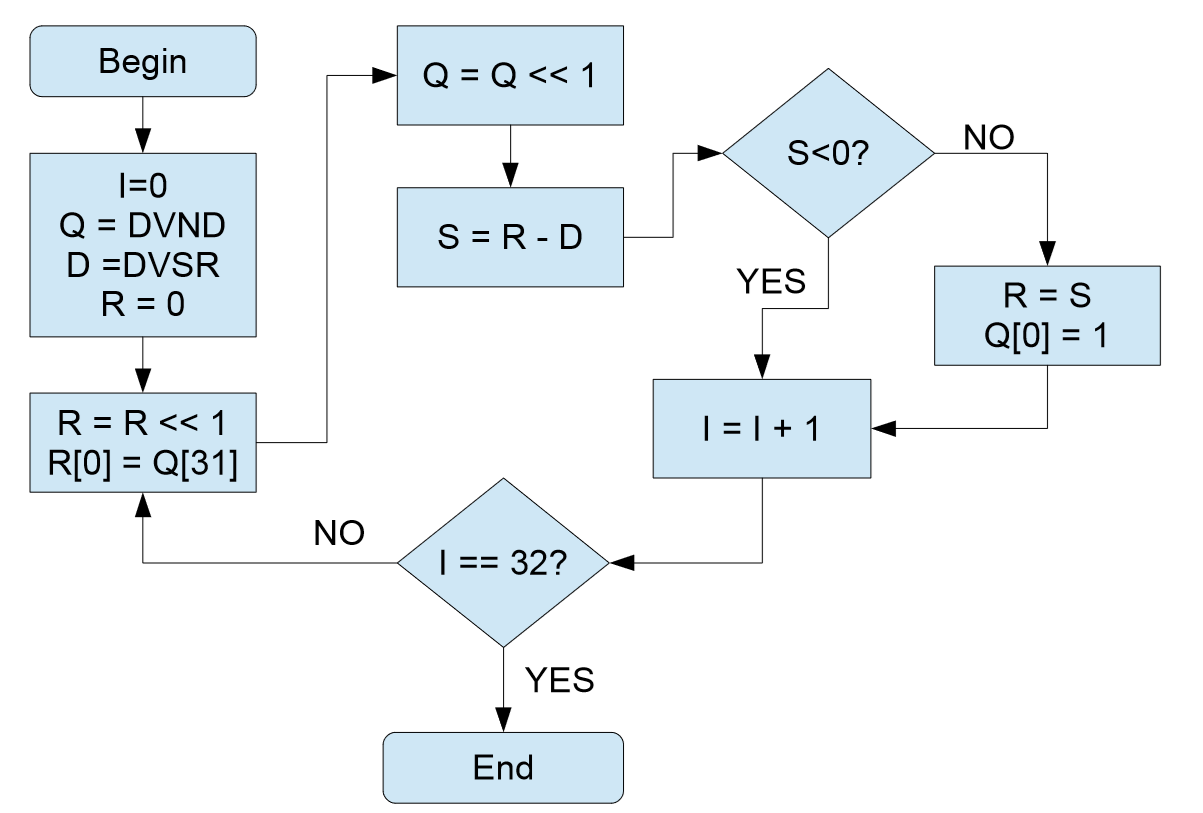
As shown in the division logical circuit in Figure 32, division is a 64-bit operation which takes a 64-bit divisor and a 64-bit dividend. The control unit issues a right shift for the divisor and subtracts the divisor from the dividend. This will yield either a positive (+ve) or a negative (-ve) result.

If negative, the addition operation is called and will write the result into the register designated for the remainder. Then, the OnZ signal is assigned a value of 0, followed by the quotient register being shifted left by one bit.

Otherwise, if the result was positive, there will be no addition and the OnZ signal will be assigned a value of 1 instead. Then, the quotient will be shifted to the left by one bit as normal.

This will continue for thirty two repetitions total, with the quotient register containing the quotient and the remainder register containing the remainder at the end.

The algorithm we will follow to implement this digital design in MIPS is shown in Figure 33.

 *Fig 33. Unsigned division algorithm*

Similar to multiplication, we will need to run a loop that repeats thirty-two times and designates registers from the beginning for the quotient (being initialized with the dividend) and the divisor. Furthermore, we initialize the remainder register with the value 0.

Following the algorithm flowchart in Figure 33, we implement the logic design MIPS as such:

A screenshot of text

Description automatically generated

A screenshot of a social media post

Description automatically generated *Fig 34. Div\_unsigned implementation*

As shown in Figure 34, the implementation is relatively simple, with occasional calls to the sub\_logical procedure (to subtract the divisor from the dividend) and the left shifts on the remainder and quotient registers. At the end of the division loop, denoted by the label div\_unsigned\_end, we follow a similar complement computation as seen in the mul\_unsigned procedure (refer to Figure 35 if needed). By extracting the most significant bits of the original dividend and divisor, followed by a logical XOR between the two, we can determine whether the result will be negative or positive (Refer to Figure 29 for XOR Truth Table computation).

If the quotient is positive, we do not complement it and instead jump to a similar check on the remainder. After all needed complements have been performed (using the twos\_complement utility procedure), we move the quotient (stored in $s1) to register $v0, and the remainder (stored in $s3) to register $v1.

A screenshot of a social media post

Description automatically generated *Fig 35. Signed division algorithm*

1. **div\_logical**

Once the div\_unsigned procedure has been implemented (with complement functionality), the div\_logical is able to proceed in a similar fashion as mul\_logical:

A screenshot of a cell phone

Description automatically generated *Fig 36. Div\_logical procedure*

We complement both operands ($a0 and $a1) if needed, and then call our div\_unsigned procedure. This will perform the division logic as described in Figure 33 and will also complement the quotient and remainder at the end as needed.

1. *Results* After implementing all of the macros, utility procedures, and logical procedures described, the program should now be fully functional (in design) and ready for testing. In the next section we will explore optimal test results and common errors (with some solutions).
2. TESTING
3. *Ideal Situation after Running the Tests*

After saving all of the files, proceed to open the file “proj-auto-test.asm” in MARS. Click the assemble button as indicated in Figure 36 and run the program by clicking the large green play button to the right of the assemble button.

A screenshot of a social media post

Description automatically generated *Fig 36. Assembling and running proj-auto-test*

If all has been correctly, the MARS screen should display a screen similar to Figure 37, printing “Total passed 40/40” when the program has finished running. What the program is doing is that it is making calls to both your normal procedures and logical procedures, testing the results against one another. If a mismatch is to appear, it is most likely to be as a result of an error in the logical implementation, not the normal implementation.

A screenshot of text

Description automatically generated

A screenshot of a cell phone

Description automatically generated  *Fig 37. Test Results*

1. *Common Errors*

If the program indicates that not all tests were passed, there is no need to worry. Some common errors and bugs include:

1. A simple typo somewhere in the program.
   1. This can be a mistake such as mistyping the name of a macro or procedure, leading to those instructions never being called
2. The misuse of temporary registers
   1. Temporary registers are not saved across multiple procedure calls and thus can result in something such as an infinite loop if not implemented correctly.
3. Not storing or restoring the frame
   1. As described in the Macros section under “Design and Implementation,” the stack frame is extremely important to preserve registers across multiple subroutines in procedure calls. Not storing and restoring the frame correctly can lead to registers being overridden or simply not saved.

If the program is still giving errors, verify that the implementations were followed correctly and refer to the logical flowchart diagrams.

1. CONCLUSION

Completing this project helped me understand the digital logic and design of the ALU and processor overall, especially through MIPS assembly language. After observing how the algorithms in the flowcharts could be applied to bit values through simple bit manipulation, logical operations, shifts, and loops, I now have a better understanding of how computation is performed in computers, compared to as in humans.

One factor that helped me understand the functionality of arithmetic operations through MIPS Logic better would be that I implemented the logical procedures in two different ways. Originally, I had implemented each logical procedure using very simple loops in combination with logical operations such as XOR’s, AND’s, and shifts. Although this was relatively simple for addition and subtraction, I began to run into difficulties for the multiplication and division implementations due to the fact these operation results were 64-bits and in two different registers. Therefore, I reimplemented every logical procedure from scratch using the recommended implementations. Although this was both a difficult and time-consuming task, I gained a more comprehensive understanding as a result.

To me, it was simply astounding how through Boolean values (1’s and 0’s), so much could be accomplished. It is both hard and awe-inspiring to imagine that simply through the manipulation of electronic signals, entire computations — faster than any human — could be performed. I am much more appreciative of computers after seeing just how complex something such as addition, subtraction, multiplication, and division could be, and will use my newfound knowledge to remember just how extraordinary Computer Science is.

REFERENCES

[1] D.A. Patterson and J.L Hennessy, Computer Organization and Design (5th edition). Waltham, MA: Morgan Kaufman, 2014, 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.