Arithmetic Logic Unit: Part Two

Mike Orduna

August 23, 2025

Contents

1 Abstract

This report presents the design philosophy, implementation, and testing of **Logic Gates** and **Arithmetic Operations** in Verilog. The project outlines hierarchical and modular design principles to construct scalable circuits. Key challenges such as debugging multi-bit arithmetic and optimizing testbenches were addressed using parametric macros and systematic verification. The report provides insights into designing an ALU and lays a foundation for further hardware optimization in future iterations.

2 Introduction

The goal of this project is to explore the design and testing of fundamental digital circuits, with a focus on constructing bit-width logic gates and arithmetic operations in the HDL Verilog. These circuits to be developed are the fundamental building blocks for an arithmetic logic unit, or ALU, and will help perform complex mathematical operations in later designs. However, the implementation of bit-width logic gates and arithmetic operators first requires a foundation of basic logic gates to be developed to make the development of more complex modules more streamlined and easier to debug. Designs for these components are straightforward and are discussed more in detail in the previous report labeled: **Project Report: Part One**.

After designing, implementing, and testing the one-bit gates, the bit-width logic gates are then developed. Rigorous testing is done to the bit-width logic gates to ensure that they accurately capture their outputs in accordance with what is expected via their truth tables and algebraic equivalences. The design of the arithmetic operations is then developed which include the four basic mathematical operations:

- **Addition**
- **Subtraction**
- **Multiplication**
- **Division**

A dedicated testbench is used to test the operations listed and furthermore, the GTKWave tool is used to analyze the simulation waveforms generated from testing. With successful testing and verification of the designed

components, the mathematical operations provide a robust method of analyzing different bit-widths.

3 Design Philosophy

Before developing individual logic components, we established core design philosophies to ensure modularity, reusability, and scalability. These principles guided the construction of the logic gates and arithmetic operations and are outlined below:

- **Hierarchical Design**: The design should focus on building upon other modules
- **Modular Design**: The design should split its functions into working parts
- **Scalable Design**: The design should be capable of growth

Adding too many restrictions or rules to the design of hardware could introduce too much overhead and since the project is used as a learning opportunity, the philosophies are kept short and concise. By following these philosophies, the development process can be kept organized and allow for easier debugging from unexpected results during simulation runtimes.

3.1 nBit Design

To prevent a limited bit-width size across the hardware, modules are developed and designed to handle a **WIDTH**, or a specified bit-width that can be passed into it. Modules created using this WIDTH parameter allow for the processing of a dynamic bit-width, resulting in a more robust module.

4 Behind the Arithmetic

To create an understanding of how computer operations are performed, the Verilog code developed delves under the hood of the four main arithmetic operations:

- **Adder**: Determines the sum of two numbers

- **Subtractor**: Determines the difference between two numbers
- **Multiplier**: Determines the product of two numbers
- **Divider**: Determines the quotient of two numbers and their remainder

Each operation produces its respective mathematical results through a series of different logic gates. When discussing how these operations are performed, it is important to note that "under the hood" refers to **how** a computer performs these operations and not necessarily **why** they are performed.

A crucial step before delving into arithmetic operations is understanding how nBit gates function and their role in arithmetic computation.

4.1 nBit Gates Implementation

A brief introduction on how the nBit modules are designed is provided to give a better understanding on how the arithmetic operations use these modules.

To begin, the basic logic gates are expanded upon in the **nBit_gates.v** file. The file does this using 2 main methods, referred to as modules:

- **Gate Instantiator**: A control module that is used to instantiate the selected gate based on an OP value that is passed from an **nBit** gate module.
- **nBit Gate**: A module that instantiates its corresponding gate.

Below is the code for both of these modules, with descriptive comments describing their purpose throughout each section of the module:

```
1 /*
2  * The control unit that directs the flow of the gates to be
     instantiated
3  * This streamlines the different gates that need to be
     instantiated and
4  * helps to reduce code redundancy by focusing the functions
     in one area.
5  */
6 module Gate_Instantiator #( parameter WIDTH = 4 , parameter
     OP = 0 ) (
7     input wire [ WIDTH-1:0 ] in1,
8     input wire [ WIDTH-1:0 ] in2,
```

```
    output wire [ WIDTH-1:0 ] out
);
    // Gates are instantiated based on an OP value, passed by
    the gate module
    genvar i;
    generate
        for( i = 0; i < WIDTH; i = i + 1 ) begin : gate_loop
            case( OP )
                0: begin
                    NOT not_instance (
                        .in( in1[ i ] ),
                        .out( out[ i ] )
                    );
                end
                1: begin
                    AND and_instance (
                        .in1( in1[ i ] ),
                        .in2( in2[ i ] ),
                        .out( out[ i ] )
                    );
                end
                ...
                // Other gates are similarly instantiated
                ...
            endcase
        end
    endgenerate
endmodule
```

```
/*
 * A module that instantiates its corresponding gate
 */
module nBit_AND #( parameter WIDTH = 4 ) (
    input wire [ WIDTH-1:0 ] in1,
    input wire [ WIDTH-1:0 ] in2,
    output wire [ WIDTH-1:0 ] out
);
    // Instantiate the AND gate by passing an OP of 1
    Gate_Instantiator #(.WIDTH( WIDTH ), .OP( 1 ) )
   and_instance (
        .in1( in1 ),
        .in2( in2 ),
        .out( out )
    );
endmodule
```

The purpose of the **nBit** gates and the **Gate Instantiator** is to provide a method for a larger width of bits to be processed at one time while providing a relationship between the bits. Each gate is instantiated in a similar fashion, with each OP value passed into the **Gate Instantiator** module changing according to the gate that needs to be instantiated at that time. The idea of the design was to create a separation of concerns between the instantiation of the gates and the actual gates themselves. Splitting the process into distinct tasks reduces the redundancy the modules would experience had the gates be instantiated within the **nBit** gate modules themselves.

4.2 Arithmetic Shifter Implementation

The **Arithmetic Shifter** module plays a crucial role in arithmetic operations, particularly in multiplication and division, as it enables efficient bitwise manipulation. As discussed in the previous report, the shifter takes in two inputs:

- **in**: The input to be shifted
- **shift**: The control input that dictates how the input is to be shifted

The control input is decomposed into 3 parts:

- **shift_dir**: The MSB which dictates the direction of the shift
- **fill**: The LSB which dictates the fill of the shifted bits
- **shift_amt**: The remaining bits control the shift amount

Furthermore, the module is designed to handle two types of shifts:

- **Logical Shift**: Shifts the input bits and fills the shifted bits with the fill value
- **Arithmetic Shift**: Shifts the input bits and fills the shifted bits with the sign bit

The code for the **Arithmetic Shifter** module is shown below:

```
/*
 * A module that takes two inputs:
 * - The input to be shifted
 * - The control input that dictates how the input is to be shifted
 */
module nBit_Shift #( parameter WIDTH = 4, parameter OP = 0 ) (
    input wire [ WIDTH-1:0 ] in,
    input wire [ WIDTH-1:0 ] shift,
    output reg [ WIDTH-1:0 ] out,
    output reg [ WIDTH-1:0 ] overflow
);
    // Wires are created to decompose the shift input
    wire shift_dir = shift[ 0 ];
    wire [ WIDTH-2:0 ] shift_amt = shift[ WIDTH-2:1 ];
    wire fill = shift[ WIDTH-1 ];

    always @(*) begin
        // Initializes the outputs (necessary for an output in an always block)
        out = { WIDTH{ 1'b0 } };
        overflow = { WIDTH{ 1'b0 } };
        if( OP == 0 ) begin
            if( shift_dir == 1'b0 ) begin
                out = ( in << shift_amt ) | ( fill << ( shift_amt - 1 ) );
                overflow = in >> ( WIDTH - shift_amt );
            end
            else begin
                out = ( in >> shift_amt ) | ( fill << ( WIDTH - shift_amt ) );
                overflow = in & ( ( 1 << shift_amt ) - 1 );
            end
        end
        else if( OP == 1 ) begin
            if( shift_dir == 1'b0 ) begin
                out = in << shift_amt;
                overflow = in >> ( WIDTH - shift_amt );
            end
            else begin
                out = $signed( in ) >>> shift_amt;
                overflow = in & ( ( 1 << shift_amt ) - 1 );
            end
        end
```

```
    end
endmodule
```

The Arithmetic Shifter enables flexible bitwise shifting and filling, adapting dynamically to different computational needs. Furthermore, the shifter can be briefly described as a tool that can either take the **square** or **square root** of an input.

5 Testbench Development

The development for the testing environment follows a similar fashion to what was discussed in the previous report. A major improvement over the previous testbench design is the introduction of parametric macros, allowing streamlined testing across multiple modules without redundancy. The example below showcases this improvement:

```
    `define GENERIC_HALF( REG1, REG2 ) \
        begin \
            REG1 = { 1'b0 }; \
            repeat( 2 ) begin \
                REG2 = { 1'b0 }; \
                repeat( 2 ) begin \
                    #10; \
                    REG2 = REG2 + 1; \
                end \
                REG1 = REG1 + 1; \
            end \
        end

    `define GENERIC_FULL( REG1, REG2 ) \
        begin \
            REG1 = { WIDTH{ 1'b0 } }; \
            repeat( BIT_STATE ) begin \
                REG2 = { WIDTH{ 1'b0 } }; \
                repeat( BIT_STATE ) begin \
                    #10; \
                    REG2 = REG2 + 1; \
                end \
                REG1 = REG1 + 1; \
            end \
            REG1 = { WIDTH{ 1'b0 } }; \
            REG2 = { WIDTH{ 1'b0 } }; \
        end
```

The testbench utilizes the macros **GENERIC_HALF** and **GENERIC_FULL** to test any variation of modules, so long as they either have one input or two inputs. This significantly reduced the repetition many other implementations would experience and helped to streamline the functionality of the tests in one, centralized location. The actual testing of each individual module however is developed in a similar fashion as the previous report. An example of the testbench for the **Half Adder** module is shown below:

```
    // Half Adder
    reg half_in1, half_in2;
    wire half_adder_out, half_adder_carry_out;

    Half_Adder adder_instance (
        .in1( half_in1 ),
        .in2( half_in2 ),
        .out( half_adder_out ),
        .carry_out( half_adder_carry_out )
    );
```

Every module is developed in a similar fashion and when testing is done, the registers are passed into the generic tests such like:

```
    initial begin
        $dumpfile( "waveform6.vcd" );
        $dumpvars( 0, testbench_arithmeticOP );

        `GENERIC_HALF( half_in1, half_in2 );

        #50 $finish;
    end
```

One last note is that the timescale for the testbench is set to 1ns, which has been a standard for the project. This will be important when analyzing the waveforms using the GTKWave tool.

6 Half & Full Adder

The first series of arithmetic operations to be developed are the **Half Adder** and the **Full Adder**. The modules provide distinct methods of handling single-bit addition and multi-bit addition, respectively. Using fundamental logic gates, the adders are designed to provide the sum of two inputs and a carry out bit if the sum exceeds the bit-width of the input.

6.1 Half Adder

The **Half Adder** is the simplest form of the adding function that a computer can perform. It takes two inputs and outputs their sum, with a carry bit if the sum exceeds the width of the bit. For instance, adding single bits together can yield four possible results:

A	B	Sum
0	0	0
0	1	1
1	0	1
1	1	0

Figure 1: Truth Table for Half Adder

The truth table in figure 1 can be decomposed into three main cases:

- **Adding Two Low Bits**: Where the sum is 0 and the carry bit is 0
- **Adding a High and Low Bit**: Where the sum is 1 and the carry bit is 0
- **Adding Two High Bits**: Where the sum is 0 and the carry bit is 1

Using these cases, the half adder can be represented by a logic diagram that shows the relationship between the inputs and outputs:

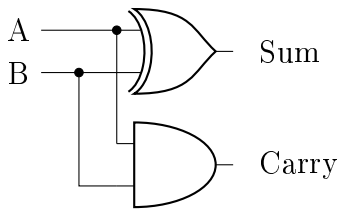

Figure 2: Half Adder Logic Diagram

Referring to the logic diagram in figure 5, the half adder is implemented using the XOR module and the AND module. The complete implementation of the code can be viewed in the **half adder code**, submitted in the **arithmetic_operations.v** file.

The development of the half adder allows for a central location for the logic of single bits. From this module, the full adder can be developed to handle multi-bit addition.

6.2 Full Adder

The **Full Adder** is an extension of the half adder, providing a method for adding two inputs of some bit-width together. Similar to the half adder, the full adder provides a sum and a carry out bit if the sum exceeds the bit-width of the input. The full adder can have a variable of combinations depending on the width of the input, but for the purposes of this project, the width is set to four bits. This means that there are sixteen possible combinations of inputs that can be added together. The truth table for the full adder follows the same logic as the half adder but with additional inputs, but the logic for full adder can be represented as a similar logic diagram as shown in figure 5.

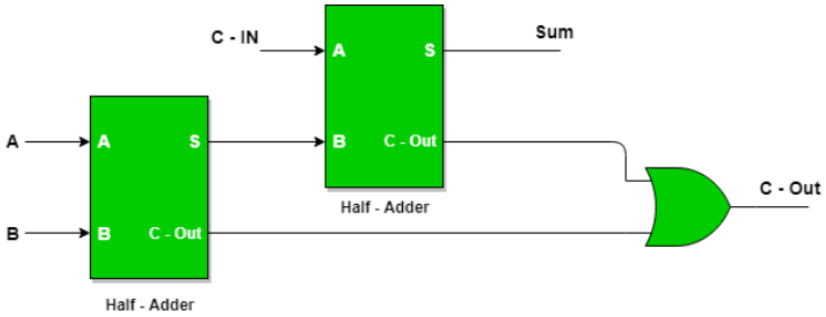

Figure 3: Full Adder Logic Diagram

The **Full Adder** uses two half adders and an OR gate to perform its calculations for a bit-width of 2. For each bit added to the width, an additional half adder is added to the diagram. The complete implementation of the code can be viewed in the **full adder code**, submitted in the **arithmetic_operations.v** file.

Using previously developed modules, the full adder is capable of adding a variable width of bits together. The **Full Adder** module is comprised of one major function: the addition loop. This loop iterates through the width of the input and adds the bits together, storing the results in the **temp_out** and **temp_carry_out** wires. The carry out bit is then added to the next bit in the loop, with the final carry out bit being stored in the **final_carry** wire. By utilizing the half adders to compute the sum of the individual bits in an input and the OR gate to compute any carry out bits, the full adder is capable of adding a variable width of bits together.

7 Half & Full Subtractor

The next series of arithmetic operations to be developed are the **Half Subtractor** and the **Full Subtractor**. The modules provide distinct methods of handling single-bit subtraction and multi-bit subtraction, respectively. Using fundamental logic gates, the subtractors are designed to provide the difference of two inputs and a borrow out bit if the difference exceeds the bit-width of the input.

7.1 Half Subtractor

The **Half Subtractor** is the simplest form of the subtracting function that a computer can perform. It takes two inputs and outputs their difference, with a borrow bit if the difference exceeds the bit-width of the input. For instance, subtracting single bits together can yield four possible results:

A	B	Difference
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4: Truth Table for Half Subtractor

The truth table in figure 4 can be decomposed into three main cases:

- **Subtracting Two Low Bits**: Where the difference is 0 and the borrow bit is 0
- **Subtracting a High and Low Bit**: Where the difference is 1 and the borrow bit is 0
- **Subtracting Two High Bits**: Where the difference is 0 and the borrow bit is 1

Using these cases, the half subtractor can be represented by a logic diagram that shows the relationship between the inputs and outputs:

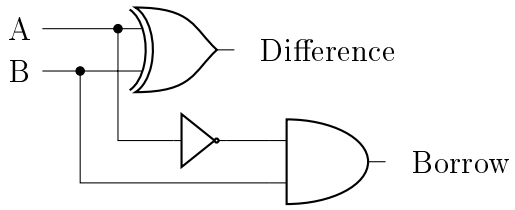

Figure 5: Half Adder Logic Diagram

Referring to the logic diagram in figure 5, the half subtractor is implemented using the XOR module, the AND module, and the NOT module. The complete implementation of the code can be viewed in the **half subtractor code**, submitted in the **arithmetic_operations.v** file.

The development of the half subtractor allows for a central location for the logic of single bits. From this module, the full subtractor can be developed to handle multi-bit subtraction.

7.2 Full Subtractor

Similar to the full adder, the **Full Subtractor** is an extension of the half subtractor, providing a method for subtracting two inputs of some bit-width together. The full subtractor provides a difference and a borrow out bit if the difference exceeds the bit-width of the input. The full subtractor can have a variable of combinations depending on the width of the input, but for the purposes of this project, the width is set to four bits. This means that there are sixteen possible combinations of inputs that can be subtracted together. The truth table for the full subtractor follows the same logic as the half subtractor but with additional inputs, but the logic for full subtractor can be represented as a similar logic diagram as shown in figure 5.

The **Full Subtractor** uses two half subtractors and an OR gate to perform its calculations for a bit-width of 2. For each bit subtracted from the width, an additional half subtractor is added to the diagram. The complete implementation of the code from the **full subtractor code** can be viewed from the submitted files under the **arithmetic_operations.v** file.

Unfortunately, the logic is very similar between the full adder and the full subtractor. The implementation of the full subtractor was unable to make use of a modified version of the full adder due to the nature of the subtraction operation. The addition of a not gate **in the middle** of the full subtractor prevented any sort of modification to be done to the full adder

module. Nevertheless, both the implementation of the full adder and the full subtractor provide a methods for adding and subtracting a variable width of bits together.

Another note is that the full subtractor is not capable of handling negative numbers as of yet. The exact implementation of negative numbers is not discussed in this report, but the full subtractor is capable of handling the subtraction of two positive numbers, which can be seen in the **Testbench Development** section.

8 Multiplier

The **Multiplier** is a module that takes two inputs and outputs their product in a high and low output. The multiplier is capable of multiplying two inputs of some bit-width together. Binary arithmetic performs multiplication in a series of steps that are repeated for as many bits as the input has. This process is referred to as the **shift and add** method and is how the design of the multiplier is implemented. Below are the steps an input goes through to be multiplied by another input:

- **Step 1**: Two inputs are multiplied together
- **Step 2.1**: For every high bit in the multiplier, the multiplicand is shifted to the left.
- **Step 2.2**: For every low bit in the multiplier, the multiplicand is set to 0.
- **Step 3**: After every bit in the multiplier is processed, the results are added together. The result is the product of the two inputs.

However, for a computer, the process of multiplication is not quite captured in these steps. Rather, the computer needs a few additional steps in between that calculate the product over a period of time. The complete implementation of the **Multiplier** module can be seen in the **multiplier code** submitted in the **arithmetic_operations.v** file.

The implementation above for the multiplier has a few key components that are used to calculate the product of two inputs:

- **Partial Sums**: Two arrays are created that store the partial sums of the product over time

- **Shift and Add Method**: The multiplicand is shifted to the left by the multiplier and added to the partial sum
- **Shifter Module**: The multiplicand is shifted to the left by the multiplier
- **Full Adder Module**: The low output is added to the shift result and the high output is added to the overflow from the shift result

This implementation is essentially how a computer performs multiplication such as using ***** in a programming language, although the nuances of the lowest level of the computer are not actually shown (aka the assembly code). However, the multiplier module highlights the level of complexity that is required to perform multiplication in a computer.

9 Divider

The **Divider** is a module that takes two inputs and outputs their quotient and remainder. The divider is capable of dividing two inputs of some bit-width together. Binary arithmetic performs division in a series of steps that are repeated for as many bits as the input has. This process is referred to as the **shift and subtract** method and is how the design of the divider is implemented. Below are the steps an input goes through to be divided by another input:

- **Step 1**: Two inputs are divided together
- **Step 2**: Align the MSB of the divisor with the MSB of the dividend
- **Step 3**: Subtract the divisor from the dividend
- **Step 4**: If the result is negative, shift the divisor to the right and add 0 to the quotient
- **Step 5**: If the result is positive, shift the divisor to the right and add 1 to the quotient
- **Step 6**: Replace the dividend with the result and repeat the process until the divisor is greater than the dividend

This is a general overview of how division is implemented when working with binary arithmetic. However, similar for the multiplier, the computer needs a few additional steps in between that calculate the quotient and remainder over a period of time.

An important note to mention regarding the divider is that the implementation of the divider is split into 2 modules: the **Divider** module and the **Division_Alignment** module. The **Divider** module is responsible for the actual division of the inputs, while the **Division_Alignment** module is responsible for aligning the divisor with the dividend. The complete implementation of the **Divider** and the **Division_Alignment** module are shown in the **divider code** and the **division alignment code** from the **arithmetic_operations.v** file respectively.

Unfortunately, the implementation of the divider is not as straightforward as the multiplier. Because of this, the divider is quite complex in its design and very verbose in its implementation. However, the divisors functions can be split into a few distinct parts:

- **Partial Quotient and Remainder**: Two arrays are created that store the partial quotient and remainder of the division over time
- **Division Alignment**: The divisor is aligned with the dividend based on the leading bits of the inputs
- **Full Subtractor Module**: The aligned divisor is subtracted from the dividend to get the initial remainder
- **Quotient Calculation**: The quotient is shifted based on the position of the leading bits of the inputs
- **Division Loop**: The new dividend is then assigned based on the remainder and repeated until the divisor is greater than the dividend

10 Challenges and Solutions

Capturing the essence of the arithmetic operations in a computer was a challenging task. The implementation of the arithmetic operations required a deep understanding of the logic gates and how they interact with each other. The development of the modules was a process that required a lot of

trial and error to get the correct output. The challenges faced during the development of the arithmetic operations are as follows:

- **Complexity of the Modules**: The arithmetic operations are complex in their design and implementation. The modules require a lot of logic gates and wires to perform the calculations. This complexity made it difficult to understand how the modules worked and how they interacted with each other.
- **Debugging the Modules**: Debugging the modules was a challenging task. The modules are designed to perform a series of calculations over time, which made it difficult to pinpoint where the error was occurring. The use of testbenches and waveform analysis was essential in debugging the modules.
- **Understanding the Logic Gates**: The arithmetic operations are built using logic gates such as AND, OR, XOR, and NOT. Understanding how these gates work and how they interact with each other was essential in developing the modules. The logic gates are the building blocks of the arithmetic operations and understanding how they work was crucial in the development process.

Despite the challenges faced during the development of the arithmetic operations, the solutions to these challenges were found through a combination of research, trial and error, and collaboration. The solutions to the challenges are as follows:

- **Research**: Researching the logic gates and how they work was essential in understanding how the arithmetic operations were implemented. The research provided valuable insights into how the modules were designed and how they interacted with each other.
- **Trial and Error**: Trial and error was a key component in the development of the arithmetic operations. Testing different inputs and analyzing the outputs was essential in understanding how the modules worked and how they could be improved.
- **Collaboration**: Collaboration with peers was essential in developing the arithmetic operations. Working together to solve problems and share insights was crucial in overcoming the challenges faced during the development process.

11 Conclusion

The development of the arithmetic operations was a challenging but rewarding task. The implementation of the arithmetic operations required a deep understanding of the logic gates and how they interact with each other. The development of the modules was a process that required a lot of trial and error to get the correct output. The challenges faced during the development of the arithmetic operations were overcome through a combination of research, trial and error, and collaboration. The solutions to the challenges provided valuable insights into how the modules were designed and how they interacted with each other.

Overall, implementing the actual hardware configurations of the arithmetic operations was rewarding and gave a deeper insight to how computers perform arithmetic when using symbols such as +, -, *, and /. In future, further enhancements to the readability and efficiency of the modules developed can be made to improve the overall performance of the arithmetic operations to provide a smoother and more efficient experience.