Arithmetic Logic Unit: Part One

Mike Orduna

August 23, 2025

Contents

1 Abstract

This report presents the design and verification process of the development of fundamental digital circuits. The work is structured in a modular, hierarchical fashion, beginning with the implementation and testing of the basic logic gates, progressing into more complex modules such as the arithmetic shifter developed later on. Detailed simulations are conducted using dedicated testbenches and the GTKWaveform tool to verify that each module functions accordingly and expectedly. While the logic gates demonstrated a predictable and consistent behavior, the arithmetic shifter received multiple iterations to its functions to ensure an optimized and clean design. Overall, the project solidified our understanding of digital circuits and provides a strong foundation for future ALU development.

2 Introduction

The goal of this project is to explore the design and verification of fundamental digital circuits, focusing on constructing basic logic gates and an arithmetic shifter. These circuits form the foundation for a larger Arithmetic Logic Unit (ALU), which will eventually perform fundamental mathematical operations. To achieve this, we first implemented one-bit logic gates such as NOT, AND, and OR, adopting a modular and hierarchical approach to ensure scalability and reusability. After designing these components, we rigorously tested their functionality using a dedicated testbench. Simulations were conducted to verify expected behavior, with results analyzed using the GTKWaveform tool. With successful verification of our initial circuit designs, we have established a foundation for further development, including extending the ALU's functionality and optimizing its performance.

3 Design

Before constructing an Arithmetic Logic Unit, we first developed its fundamental components, beginning with basic logic gates. We structured the project into modular stages, where each stage builds upon the previous one, forming a hierarchical design. Our first step was to implement basic logic gates. These gates are essential in digital circuits and were implemented using

Boolean algebra principles to ensure reusability and consistency throughout the design.

3.1 Basic Logic Gates

Each gate plays a pivotal role in digital computation. The AND and OR gates facilitate logical decision-making, the NOT gate allows for signal inversion, and the XOR gate is fundamental in arithmetic circuits, particularly in half-adders and full-adders, which are used for binary addition. More complex gates, such as NAND and XNOR, were constructed by instantiating simpler gates, aligning with the modular approach we outlined. Among these, the NAND and NOR gates are particularly significant, as they are functionally universal and can be used to construct any other logic gate. This design choice streamlined the debugging process and provided a scalable framework for representing digital logic. These gates establish a scalable foundation for implementing more advanced arithmetic and logic operations in future stages of the project.

4 Verilog Code

The Verilog code for part one of the project is organized into two files. The first file implements the basic logic gates:

- **NOT** - Inverts the input signal.
- **AND** - Outputs high if both inputs are high.
- **OR** - Outputs high if either input is high.
- **NAND** - Outputs low if both inputs are high.
- **NOR** - Outputs low if either input is high.
- **XOR** - Outputs high if the inputs differ.
- **XNOR** - Outputs high if the inputs are the same.

4.1 Basic Logic Gates

Each module follows a hierarchical design, ensuring reusability and scalability for more complex operations.

```
/*
 * Each module has a header that describes the modules
 * inputs and outputs.
 */
module AND (
    /* Each module has two inputs and one output */
    input in1,
    input in2,
    output out
);
    /*
     * Each module then assigns the output to the logical
     * operation of the inputs
     */
    assign out = in1 & in2;
endmodule
```

Derived gates, such as NAND and NOR, are implemented by instantiating their base gates and inverting the output, ensuring modularity and consistency. This approach streamlines debugging and maintains a systematic design by constructing complex logic from fundamental operations.

```
module NAND (
    input in1,
    input in2,
    output out
);
    /* An output for the instantiated gate is made */
    wire and_result;
    AND and_instance (
        .in1( in1 ),
        .in2( in2 ),
        .out( and_result )
    );
    /* The output is then inverted with the '~' operator */
    assign out = ~and_result;
endmodule
```

4.2 Arithmetic Shifter

The second file implements an arithmetic shifter, which performs left or right shifts on a width-bit input. The width is parameterized and can be set during instantiation, ranging from 1 to 1024 bits. Errors are triggered for extreme values, and a warning is issued at 256 bits.

```
1  module New_nBit_Shift #( parameter WIDTH = 4, parameter OP =
       0 ) (
2      input wire [ WIDTH-1:0 ] in,
3      input wire [ WIDTH-1:0 ] shift,
4      output reg [ WIDTH-1:0 ] out,
5      output reg [ WIDTH-1:0 ] overflow
6  );
7      /*
8       * A check is made to verify the parameters used when
9       * the module is instantiated (if at all specified)
10      */
11     generate
12         if( WIDTH < 2 ) begin
13             initial begin
14                 $error( "WIDTH must be at least 2" );
15             end
16         end
17         else if( OP < 0 || OP > 1 ) begin
18             initial begin
19                 $error( "OP must be between 0 or 1" );
20             end
21         end
22     endgenerate
23
24     /* Wires are created to disect the shift input */
25     wire shift_dir = shift[ 0 ];
26     wire [ WIDTH-2:0 ] shift_amt = shift[ WIDTH-2:1 ];
27     wire fill = shift[ WIDTH-1 ];
28
29     /* The magic happends here */
30     always @(*) begin
31         out = { WIDTH{ 1'b0 } };
32         overflow = { WIDTH{ 1'b0 } };
33
34         if( OP == 0 ) begin /* Logical shift */
35             if( shift_dir == 1'b0 ) begin /* Left shift */
36                 out = ( in << shift_amt ) | ( fill << (
   shift_amt - 1 ) );
```

```
37                  overflow = in >> ( WIDTH - shift_amt );
38              end
39              else begin  /* Right shift */
40                  out = ( in >> shift_amt ) | ( fill << ( WIDTH
    - shift_amt ) );
41                  overflow = in & ( ( 1 << shift_amt ) - 1 );
42              end
43          end
44          else if( OP == 1 ) begin    /* Arithmetic shift */
45                  if( shift_dir == 1'b0 ) begin   /* Left shift
    */
46                      out = in << shift_amt;
47                      overflow = in >> ( WIDTH - shift_amt );
48                  end
49                  else begin  /* Right shift */
50                      out = $signed( in ) >>> shift_amt;
51                      overflow = in & ( ( 1 << shift_amt ) - 1
   );
52                  end
53          end
54          else begin
55              /*
56              * A default case is included to
57              * catch any errors in the OP parameter
58              */
59              $error( "Error: Default case succeeded where it
    shouldn't. \n" );
60          end
61      end
62  endmodule
```

To facilitate the shifting process, the shift input is decomposed into three distinct components:

- **Shift Direction** - Determines whether the shift is left or right.
- **Shift Amount** - Specifies how many positions the input is shifted.
- **Fill Bit** - Used in logical shifts to maintain bit alignment.

4.3 Testbench Development

To verify correctness, separate testbenches were created for each module. These testbenches systematically iterate through all possible input combina-

tions to ensure accurate functionality. To improve readability and maintainability, test inputs were controlled using a dedicated task. However, testing the arithmetic shifter was significantly more complex than testing the basic logic gates due to its multiple parameters and shift operations. Despite this, both testbenches followed a similar structured approach to ensure reliable verification.

```
    // Internal wires to store the output of the module
    reg [ 0:0 ] in, in1, in2;
    wire [ 0:0 ] NOT_out, AND_out, OR_out, NAND_out, NOR_out,
    XOR_out, XNOR_out;

    // Each gate is then instantiated such as:
    AND and_instance (
        .in1( in1 ),
        .in2( in2 ),
        .out( AND_out )
    );
    OR or_instance (
        .in1( in1 ),
        .in2( in2 ),
        .out( OR_out )
    );
    // And so on...

    // A task is then created to handle the test cases
    task test_bit;
        begin
            // Initialize the first input signal
            in1 = { 1'b0 };

            repeat( 2 ) begin
                // Initialize the second input signal
                in2 = { 1'b0 };
                repeat( 2 ) begin
                    #10;
                    in2 = in2 + 1;
                end
                in1 = in1 + 1;
            end
        end
    endtask

    // Finally, the initial block is used to call the task
    initial begin
```

```
38          $dumpfile( "waveform1.vcd" );
39          $dumpvars( 0, testbench_bit );
40
41              test_bit;
42
43          $finish;
44      end
```

5 Waveform Analysis

5.1 GTKWave Overview

Each test required a method of evaluation to verify expected results. Hence, the utilization of the GTKWaveform tool was necessary. The tool allowed for the visualization of the simulation results, which were then compared to the expected results. Waveforms were stored in a .vcd file and analyzed using GTKWave to compare actual vs. expected results.

5.2 Basic Gate Signal Results

The basic logic gates have two primary input signals, in1 and in2. The outputs for each gate are shown in a unique signal that is designated for that gate. To clarify, the signals received and outputted by the AND and NAND gates are represented in figure 1:

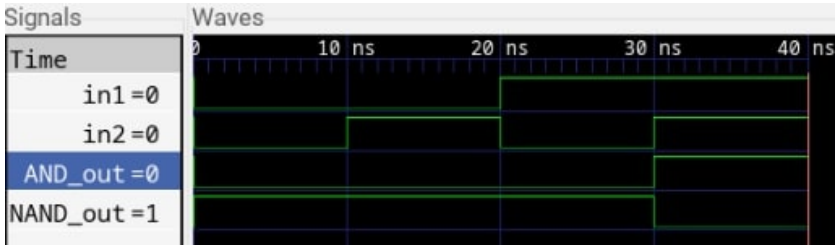

Figure 1: The outputs for the AND and NAND gates reflect the inverse of one another

From the basic logic gates test, and more specifically from the AND and NAND gates, it is noted that the simulation lasts 40ns. Each increment of in2 applies a 10ns delay to simulate propagation effects. Our inputs span

across all available iterations (00, 01, 10, 11) where in1 is the driver input and in2 is the iterative input. That being said, for every instance of in1, in2 iterates through all of its available inputs. Furthermore, it can be seen that the NAND and AND gates are direct inverses of each other, which aligns with their functionality conceptually.

For the AND gate, the output signal remains low up until the point when it receives two high input signals, to which it outputs a high signal. For the NAND gate, it exhibits inverse properties to that of the AND gate. That being said, the output signals remain high up until it receives two high input signals, to which it outputs a low signal.

5.3 Arithmetic Shifter

The arithmetic shifter is more complex than basic gates. It uses two inputs but additionally tracks specific bit positions, shifts the input by a defined amount, and captures overflowed bits.

5.4 Shift Signal Analysis

To accurately track changes in shift signals, we carefully monitored bit increments and transitions. Figure 2 illustrates the hierarchical control of shift signals.

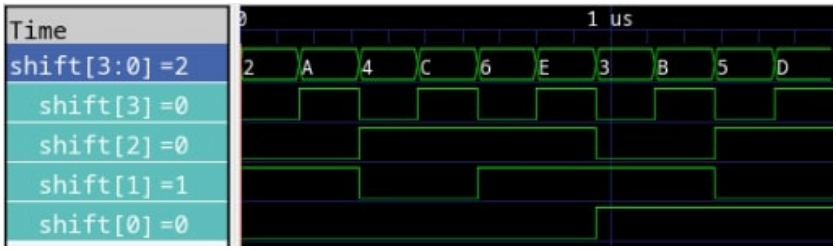

Figure 2: The shift signals are controlled hierarchically by the fill bit, shift amount, and shift direction

Each bit updates based on its position relative to other bits in the signal. The Shift direction is the leading bit that controls the overall flow of the signal and is the least significant bit. The shift amount is then controlled afterwards and controls the flow of the fill bit. The fill bit is the last bit in

the signal to change, hence why it alters the most frequently. Furthermore, right shifts involve fewer operations but risk data loss when shifting too far. To mitigate this, a stricter limit is applied.

5.5 Arithmetic Shifter Simulation Results

For the arithmetic shifters results, there are two distinct outputs that are tested for: the logical shift and the arithmetic shift. Both shifts have similar properties, but the arithmetic shift has an additional operation that is performed to inputs that are shifted to the right. To clarify the operations, the outputs for the arithmetic shifter are shown in figure 3:

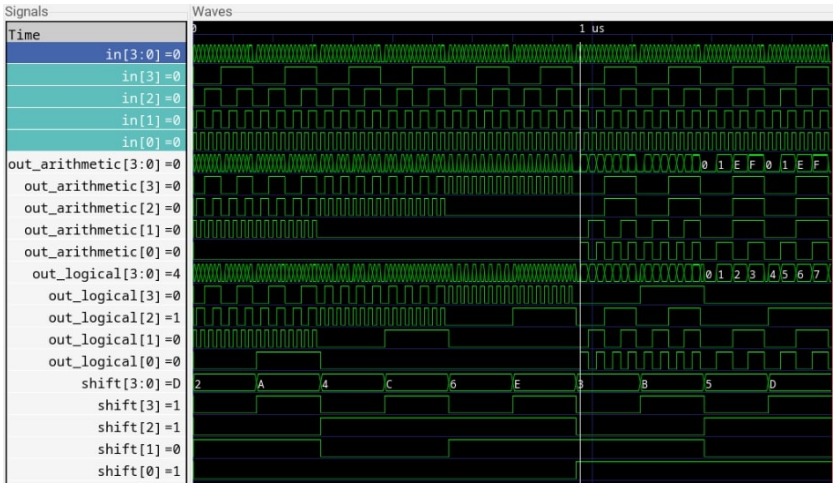

Figure 3: The logical and arithmetic shifts capture how the input is shifted

Both outputs showcase a shift of the input either **up** (which signifies a left shift) or **down** (which signifies a right shift). However, there a few key differences between the two outputs.

Logical Shift Logical shifts use a fill bit at the most significant position, maintaining alignment when shifting left or right.

Arithmetic Shift As opposed to the logical shift, the arithmetic shift has no fill bit used in its shift operation. Arithmetic shifts behave like logical shifts for left operations but use sign extension for right shifts, preserving the sign bit.

5.6 Arithmetic Shifter Overflow Results

Since both shift operations may discard bits, an overflow signal captures and preserves these lost values. This is shown by the following figure 4:

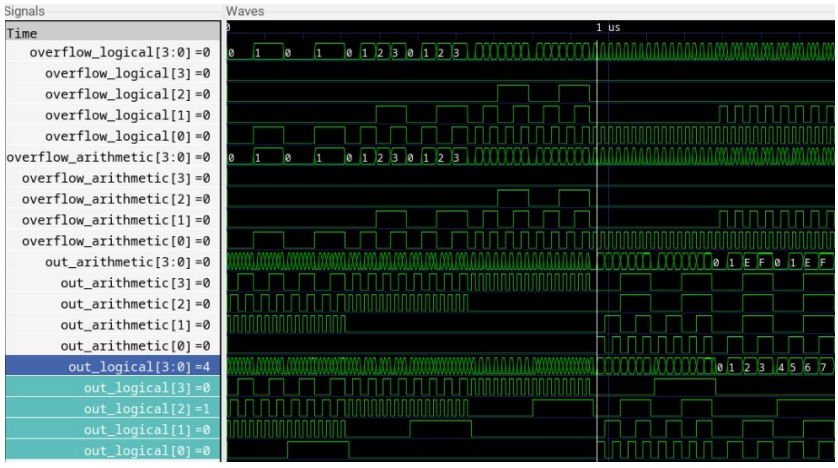

Figure 4: the overflow signals capture the bits that are shifted out of the output

From the overflow signals, it can be seen that the overflow signal captures the bits that are shifted out as the input is shifted. The capture of these bits allows for the arithmetic shifter to maintain the integrity of the input signal, even as bits are shifted out of the output. Overflow behavior remains consistent across logical and arithmetic shifts, capturing bits discarded during the shift process.

6 Discussion

6.1 Design and Development

The development of the basic logic gates and arithmetic shifter was both challenging and rewarding. Designing these modules required a strong foundation in Boolean algebra and digital logic principles. A structured methodology and hierarchical design approach were crucial, allowing us to build complex circuits from simple components. Developing the Verilog code required careful planning to ensure scalability and reusability. Each module underwent

iterative improvements to enhance functionality and flexibility. One major improvement was the addition of parameters, which allowed for greater adaptability in testing and implementation.

6.2 Challenges and Solutions

Initially, modules were designed with fixed bit-width inputs and outputs. However, as testing progressed, a more flexible system became essential. To address this, we introduced parameters, enabling dynamic bit-width configuration. This allowed for thorough testing across various input sizes, ensuring robust functionality. To prevent excessive resource usage, constraints were added to the parameters, ensuring the system remained both flexible and efficient. Without these limits, the system risked inefficiencies due to an overly broad scope.

6.3 Arithmetic Shifter Challenges

Few challenges were as significant as those encountered in developing the arithmetic shifter. Since it relied on multiple control inputs, testing required generating a wide range of input combinations. To simplify testing, we combined the three main inputs into a single shift signal, encapsulating all shift control parameters. Additionally, to reduce redundant code in the testbench, tasks were implemented to organize test cases, improving readability and maintainability. Throughout development, continuous improvements in organization and optimization enhanced efficiency, resulting in a more manageable and scalable system.

7 Conclusion

The completion of the first phase of this project marks a significant milestone in our journey toward designing a fully functional ALU. The development of basic logic gates and the arithmetic shifter has established a strong foundation for future expansion. Our hierarchical design approach and modular structure have proven effective in creating scalable and reusable components. Rigorous testing and waveform analysis validated the functionality of each module, ensuring they meet the project's requirements. Moving forward, we will extend this foundation by developing more complex modules, such

as arithmetic and logic operations, to fully realize the ALU's capabilities. The knowledge and experience gained in this phase will enable us to tackle future challenges effectively and refine our design for greater efficiency and performance.