Arithmetic Logic Unit: Part Three

Mike Orduna

August 23, 2025

Contents

Abstract

This report describes the design, implementation, and verification of a 4-bit Arithmetic Logic Unit in Verilog using a hierarchical, modular approach. Core combinational submodules implement logic gates, addition, subtraction, multiplication, and division, while a finite-state-machine controller sequences multi-cycle operations. A generic Verilog testbench exercises all opcodes and operands with GTK-Wave timing diagrams revealing a clear, operation-specific patterns such as a rising staircase remainder in division and half-wave mirror symmetry in multiplication, confirming correct synchronization between data paths and control logic. The result is a fully functional, easily extensible ALU whose design balances clarity, reusability, and testability, laying the groundwork for future enhancements.

1 Background

This report is the third part of a project that aims to develop an ALU using Verilog. The ALU could not be as functional as it is currently without the previous implementations of the ALU's most basic modules. Throughout the course of the project, design decisions were made to ensure that the ALU remained modular and easy to understand. That being said, the ALU was chosen to follow a *hierarchical design* approach, where the ALU is built from smaller modules.

Development of the ALU began with implementing its most basic functions: *Logic Gates*. Implementation starts with making modules for these basic gates, which are then used to develop more complex modules down the line. In the first report discussed, the primary focus was ensuring that these logic gates received the fed inputs correctly and produced the expected outputs. Furthermore, the development of a basic arithmetic shifter was implemented in conjunction with the logic gates, as required by the project's first step. These files can be viewed under the **src** directory, and more specifically refer to the files *bit_gates.v*, *nbit_gates.v*, and *arithmetic_operations.v*. Other modules made in this section include the *checks.v* and the *mxnbit_gates.v* files which offer additional functionality outside of the specifications of the project.

Afterwards, the development of the core functions of the ALU were implemented. This included the basic mathematical operations of an ALU: addition, subtraction, multiplication, and division. Following the guidelines developed during the planning of the project, this section of the ALU is built on many of the logic gates previously developed and make use of Verilog's *generate* statements to instantiate the appropriate modules for each operation. The modules refer to the files *arithmetic_ operations.v* and the later added *combinational_ alu.v* files. In earlier versions of this part of the project, the operations for the ALU went through many different iterations and were all shared in the same file. However, after much evaluation, a separation of concerns was established and enforced between the two core functions, leading to the split of the files.

While the initial project phases emphasized the development of the ALU's core functions, the later phases of the project focus on the refinement of the ALU's modules. Moreover, the later phases of the project focus on the introduction of a controller to the ALU, allowing for it to perform its calculations more realistically and sequentially.

2 Introduction

An ALU's effectiveness is not only determined by its ability to perform arithmetic operations, but also by its ability to manage the flow of these operations over a period of time. In earlier stages of the project, the ALU's primary focus was functionality and correctness of the operations. However, moving from the core functions of the ALU to the controller, the focus shifts to the overall control the ALU has over its operations.

The introduction of control logic to the ALU transforms it from a purely combinational circuit into one that is sequential. This means that the ALU would process instructions fed into it over multiple clock cycles. This change in fundamental processing allows for an ALU to synchronize its operations, managing its time and resources more effectively. In this report, a focus on the developmental process of the ALU controller is discussed, including the methods used to implement the controller, the challenges faced during development, and the solutions that were developed to overcome these challenges.

3 Controller Design

The controller is the most important aspect of an ALU, as it is responsible for directing the flow of data and operations within the ALU. A controller is a sequential circuit that takes in inputs and produces outputs based on the current state of the system.

3.1 Controller Inputs and Outputs

To design the controller, defining its inputs and outputs is essential. There are two types of inputs to a controller: the timing inputs and the ALU inputs. The timing inputs are the signals that are used to control the timing of the controller's operations and include the following:

- **Clock**: The clock signal is used to synchronize the operations of the ALU. It is a periodic signal that is used to trigger the controller's state transitions.
- **Reset**: The reset signal is used to initialize the controller to a known state. It is an active high signal that resets the controller's state.
- **Start**: The start signal is used to indicate that the ALU should begin processing an operation. It is an active high signal that triggers the controller to start its operation.

The ALU inputs are the signals that are used to determine and perform the operation to be performed by the ALU. These inputs include the following:

- **Opcode**: The opcode is a binary representation of the operation to be performed by the ALU. It is a multi-bit signal that is used to select the operation to be performed.
- **Input 1**: The first input to the ALU. It is a multi-bit signal that is used as the first operand for the operation.
- **Input 2**: The second input to the ALU. It is a multi-bit signal that is used as the second operand for the operation.

The primary output is split into three outputs: a high, a low, and a flag. This is to accommodate the different methods of outputting from the individual ALU operations. The outputs are as follows:

- **High Output**: The high output is a multi-bit signal that is used to indicate the high portion of outputs. Primarily used in multiplication, division, or shift operations.

- **Low Output**: The low output is a multi-bit signal that is used to indicate the low portion of outputs. This is used to represent the remaining operations such as addition or subtraction as well as the low portion of multiplication and division or the shift result.

- **Flag**: The flag is a single bit signal that is used to represent carry, borrow, or comparison flags.

The controller has an output to mark its completion labeled as *done*. This is a single bit signal that is used to indicate that the ALU has completed its operation and is ready to output the result.

3.2 Separation of Concerns

In order to facilitate the development of the controller, a separation was established between the main ALU controller and the sub-controllers.

- **Sequential Controller**: Acts as the main controller for the ALU. It is responsible for managing the overall flow of the ALU operations and directing the sub-controllers to perform their respective operations.

- **Sub-Controllers**: These are the individual controllers that are responsible for performing the specific operations of the ALU. Each sub-controller is responsible for a specific operation, such as addition, subtraction, multiplication, or division.

3.3 When to Use a Controller?

In the design of the ALU's controller, it was important to ask when to develop a controller for a specific module. Not every operation in an ALU requires a controller, as in some cases operations are done in one clock cycle as opposed to a span of cycles. Since the combinational logic was split in its core operations, it was deduced that those operations, operations with a *core* unit attached to them would need to be controlled by some central controller. This meant that modules such as the shifter, the comparator, and the logic gates would not need a controller, while remaining modules would.

Furthermore, since the ALU controller was split into two function, it would not be ideal for both the ALU controller and the sub-controllers to be sequential. Had both controllers included sequential logic, the interaction between their clock cycles would create more complicated timing mismatches, which in turn would require more complex logic to manage. So it was declared that the ALU controller would solely be combinational, while still housing the timing inputs to feed into the sub-controllers; those of which did function on the timing inputs. This design decision allowed for the separation of concerns between the different controllers developed for the ALU and allowed for a more modular design.

4 Testbench Development

To test the functionality of the controllers developed, two generic tests were designed and made, similarly to how they were discussed in the previous report, *Part Two*. These tests iterated through every potential input combination possible for the ALU. More specifically, it incremented the 3 primary inputs in the ALU: the opcode, input 1, and input 2.

4.1 Testbench Design

The tests also needed to include some method of tracking the clock and reset signals so that the ALU could be properly initialized and run for each change in input. In order to properly capture the correct change in output for each change in input, posedge detections were added into the generic tests. These posedges are used to tell the controller to update its state and output registers. However, applying a posedge requires a careful consideration for where they are placed in the generic tests to avoid saturating the system with too many posedges and that it updates its outputs accordingly. The general rules to follow when adding posedges are:

- **Posedge Between Input Change**: This is to ensure that the controller updates its state and output registers before the next input change.
- **Posedge Between Control Transitions**: This is to ensure that the controller updates its state and output registers before the next control transition.
- **Posedge After Setting Done Signals**: This is to ensure that the controller updates its state and output registers after the done signal is set.

4.2 Testbench Implementation

The testbench was implemented in a separate file, *testbench_sequential.v*. This file tested two both the ALU controller and the sub-controllers.

Sub-Controller Testing

Testing was first done on the sub-controllers, as they relied on their *core* units. Developing these controllers first allowed for a more modular design to be implemented across the different sub-controllers, as we would be testing for similar functionality across the different modules. The testbench developed for the sub-controllers is shown below:

```
 `define GENERIC_CONTROL( REG1, REG2, CLK, RESET, START,
DONE ) \
 begin \
     RESET = 1'b0; START = 1'b0; \
     REG1 = { WIDTH{ 1'b0 } }; \
     repeat( BIT_STATE ) begin \
         REG2 = { WIDTH{ 1'b0 } }; \
         repeat( BIT_STATE ) begin \
             RESET = 1'b1; @( posedge CLK ); \
             RESET = 1'b0; @( posedge CLK ); \
             @( posedge CLK ); \
             START = 1'b1; @( posedge CLK ); \
             START = 1'b0; \
             wait( DONE ); \
             @( posedge CLK ); @( posedge CLK ); \
             REG2 = REG2 + 1; @( posedge CLK ); \
         end \
         REG1 = REG1 + 1; @( posedge CLK ); \
     end \
     REG1 = {WIDTH{1'b0}}; \
     REG2 = {WIDTH{1'b0}}; \
 end
```

The test was implemented using a macro to allow for a generic test to be used with any of the different controllers. The macro takes in the following parameters:

- **REG1**: The first register to be tested. This is the register that will be used to store the first input to the ALU.
- **REG2**: The second register to be tested. This is the register that will be used to store the second input to the ALU.

- **CLK**: The clock signal for the ALU. This is the signal that will be used to synchronize the operations of the ALU.
- **RESET**: The reset signal for the ALU. This is the signal that will be used to initialize the ALU to a known state.
- **START**: The start signal for the ALU. This is the signal that will be used to indicate that the ALU should begin processing an operation.
- **DONE**: The done signal for the ALU. This is the signal that will be used to indicate that the ALU has completed its operation and is ready to output the result.

Each timing input is carefully placed to ensure that the system is able to record and process the inputs correctly. One mismatch in the timing of the inputs could lead to a loss of information, where old data is saved or data that does not exist yet is used. This is why the timing of the inputs is so important in the design of the controller.

Clock Signal Implementation

Notice that the clock signal is not included in the macro directly; there is no CLK that is incremented in the macro. Rather, the clock is initialzied outside of the macro, in the testbench. There are a few standard methods for which to increment a clock depending on the application. For the purpose of this testbench, a simple implementation was used to increment the clock:

```
    initial begin
        clk = 1'b0;
        # Inverts the clock every 5 time units
        forever #5 clk = ~clk;
    end
```

This implementation increments the clock signal every 5 time units, which is set to 1ns for the purpose of this testbench. This means that the clock signal will toggle every 5ns forever, until the simulation is stopped. This is a simple implementation of a clock signal that is used to synchronize the operations of the ALU.

ALU Controller Testing

Testing for the ALU controller was done in a similar fashion to the sub-controllers, with the only difference being an extra input to the macro: the opcode. The opcode is placed at the beginning of the macro, as it is the first input to be changed. The opcode is then incremented in a loop after the remaining test is concluded. For every opcode, the generic test listed above is conducted.

4.3 Testbench Timing Diagrams

Input Timing

The testing algorithm shown in section **Sub-Controller Testing** iterates the two main register that control the inputs of each operation. They can be described and shown in the figure below:

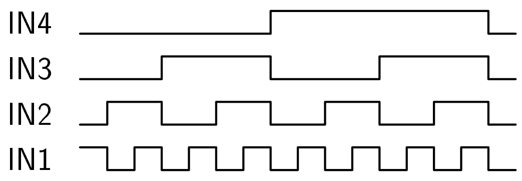

Figure 1: Timing diagram of one complete input cycle

Diagram 1 shows how `REG2` is affected by `REG1`. The generic tests follow a pattern while iterating that allow for every possible combination of inputs to be passed into the modules being tested; in this case, the ALU controller and its sub-controllers. For `REG1`, it starts at a value of 0 and increments by one after completing a full cycle of `REG2`, so that means that every change in input from `REG1` has an equivalent pattern shown in the timing diagram 1.

Control Signal Timing

At the lowest level of the testbench, the clock, reset, and start signals make up a single input change. This change is represented by the timing diagram shown below.

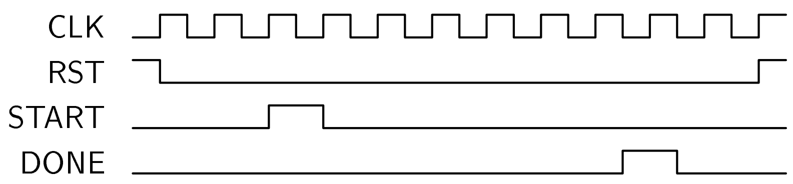

Figure 2: Timing diagram showing one ALU controller activation cycle with a reset phase, input start pulse, and synchronization to the system clock.

Diagram 2 illustrates the timing of the control signals that accompany every individual change in the input. Importantly, although Figure 1 depicts a complete input cycle of REG2, *each* increment or update in REG2 triggers the control signal sequence. Thus, every time IN2 is updated (e.g., when IN2 changes from 0 to 1, or from 1 to 2, and so on), the corresponding control signals (RST, START, and DONE) are activated, ensuring that the ALU controller processes each new input reliably and synchronously with the clock signal.

All control signals (`RST`, `START`, and `DONE`) are held high for *two full clock cycles* rather than one to ensure the controller registers each transition reliably. This design choice prevents race conditions and ensures that state changes are properly latched. The effect of this timing strategy will become more evident in later sections when the behavior of the sub-controllers is analyzed.

Furthermore, since every change in input results in the control signals in diagram 2, every input that is changed in the timing diagram 1 has these signals working in the background.

4.4 Testbench Module Instantiations

In order to use the macro in the testbench, the modules to be tested need to be instantiated in the testbench. This is done by creating instances of the modules that are to be tested and passed into the macro. One example of this is shown below, where the *Addition_ Control* module is instantiated in the testbench:

```
    /*
     * Addition_Control module instantiation
     */
    reg [ WIDTH-1:0 ] add_in1, add_in2;
    wire [ WIDTH-1:0 ] add_out;
    wire add_done, final_carry;

    Addition_Control #( .WIDTH( WIDTH ) )
   add_control_instance (
        .clk( clk ),
        .reset( reset ),
        .start( start ),
        .in1( add_in1 ),
        .in2( add_in2 ),
        .out( add_out ),
        .final_carry( final_carry ),
        .done( add_done )
    );

    // Latch the output signals for clearer waveform
    reg [WIDTH-1:0] add_latched;
    reg carry_latched;
    always @(posedge clk) begin
        if( add_done ) begin
            add_latched <= add_out;
            carry_latched <= final_carry;
        end
    end
```

Every module tested is instantiated the same way, with differences being the naming of input variables and module names.

Latch Implementation

It is important to note the use of a *latch* for the output signals. The outputs from each module vary or oscillate with the periodic clock due to the internal combinational circuitry, which can produce intermediate values between clock edges. To avoid an oscillating output, the output signal can be latched using a flip-flop register. The latching is done using a simple always block that captures the output of the module when the `DONE` signal is asserted. This ensures a stable output value is captured until the next operation is completed.

5 ALU Controller

The ALU controller is the top-level controller for the ALU, as it designates the flow of data and operations within the ALU. The inputs and outputs of the ALU controller are discussed in the *Controller Inputs and Outputs* section.

5.1 ALU Operations

The ALU controller directs the flow of data using a finite state machine, which manages operation selection and output coordination. Although FSMs are typically sequential, the ALU FSM was designed to act as a purely combinational block to mitigate timing mismatches between the controller and sub-controllers. In order to maintain the modularity and hierarchical design, the ALU instantiates the sub-controllers before entering its FSM. Because of this, every operation in the ALU is computed per input. However, once the FSM is entered, only the specified operation is transferred and outputted by the ALU controller module.

ALU Controller States

The ALU controller is designed to handle the following operations with the assigned opcodes:

Operation	**Opcode**
`Addition`	0000
`Subtraction`	0001
`Multiplication`	0010
`Division`	0011
`Logical Shift`	0100
`Arithmetic Shift`	0101
`Greater Than`	0110
`Less Than`	0111
`Equal To`	1000
`Bitwise AND`	1001
`Bitwise OR`	1010
`Bitwise XOR`	1011
`Bitwise NOT`	1100

Figure 3: ALU Opcode Table

One thing to note about the ALU operations is the difference between the logical and arithmetic shifts. *Logical shifts* are used to shift the bits of a number to the left or right, filling the empty bits with zeros. This is useful for unsigned numbers, where the sign of the number does not matter. *Arithmetic shifts*, on the other hand, are used to shift the bits of a signed number to the left or right, filling the empty bits with the sign bit. This is useful for signed numbers, where the sign of the number matters. Both of these shifts carry their own control signals within them and are used to determine the direction of the shift, the amount of bits to shift, and the fill bit to use.

Sub-Controller Instantiations

The ALU controller instantiates the sub-controllers for each operation to be performed. This is a feature of the ALU controller that allows for the modularity of the design to be maintained more easily. This also allows for the ALU to pass down the responsibility of the operations to the sub-controllers, rather than having all the logic in the ALU controller. However, this does impact the performance of the ALU, as the ALU controller must wait for the sub-controllers to complete their operations before it can proceed, even when the operation is not needed.

Each sub-controller is instantiated such like:

```verilog
    // Addition Controller
    wire [ WIDTH-1:0 ] add_out;
    wire add_done, final_carry;

    Addition_Control #( .WIDTH( WIDTH ) ) adder_instance (
        .clk( clk ),
        .reset( reset ),
        .start( start ),
        .in1( in1 ),
        .in2( in2 ),
        .out( add_out ),
        .final_carry( final_carry ),
        .done( add_done )
    );
```

Most of the sub-controllers are instantiated in a similar fashion, with the differences being the naming conventions and the output signals. The instantiations allow for a modular and cleaner design, as the ALU controller becomes a simple wrapper around the sub-controllers. This allows for the ALU controller to be easily modified or extended in the future, as new operations can be added by simply adding a new sub-controller and updating the ALU controller to include the new operation. This is a key feature of the hierarchical design approach taken in this project.

ALU Finite State Machine

The finite state machine for the ALU controller is designed to be purely combinational. It selects and coordinates which of the sub-controllers outputs to use based on the opcode. The FSM for the ALU was originally designed to be sequential, but due to timing mismatches between the ALU controller and the sub-controllers, it was changed to be purely combinational. This change allows for easier debugging and readability at the cost of performance. Sequential logic is typically easier to debug and understand than combinational logic, since the flow of the data using case statements and a clock is easier to follow than a looped combinational circuit. However, this change does not impact the functionality of the ALU, as the ALU controller is still able to perform its operations correctly. The FSM is shown below:

```
    always @(*) begin
        // Initialize the output signals
        out_high = { WIDTH{ 1'b0 } };
        out_low = { WIDTH{ 1'b0 } };
        flag = 1'b0;
        done = 1'b0;

        case( opcode )
            ADD: begin
                out_low = add_out;
                flag = final_carry;
                done = add_done;
            end
            SUB: begin
                out_low = sub_out;
                flag = final_borrow;
                done = sub_done;
            end
            ... and so on
```

This pattern continues for all available operations, where the different outputs are assigned based on which operation is selected. Not every output in the ALU controller is assigned due to either a limited representation of the output or the lack of need for the output. Due to how the operations performed have varying output sizes, the outputs from the ALU remain flexible.

6 Sub-Controllers

The sub-controllers are the individual controllers of their respective operation. Of the operations performed by the ALU, there are only four operations that required sub-controllers:

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

Each of these operations performed their calculations in a series of steps, and originally used loops to perform their operations. However, after some refactoring of the modules, the operations were changed to accommodate the new design of the ALU. Each sub-controller now incorporates a finite state machine (FSM) with a well-defined state transition process to manage the step-by-step execution of its operation. This in turn added additional clarity to the design of the ALU, and furthermore, allowed the operations to be performed in a sequential manner rather than combinational. It is important to note that this is in accordance to how real hardware operates, as hardware takes time to produce results.

6.1 Sub-Controller Operation

Each sub-controller runs in a sequential manner. The sub-controller is activated by the ALU controller, which then runs through its FSM to perform the operation. The FSM is broken down into parts that are responsible for the different steps of the operations and all remain relatively similar in design. The general structure of the FSM is as follows:

- **IDLE**: The initial state of the FSM. The FSM waits for the start signal to be asserted before proceeding to the next state.
- **INIT**: The initialization state of the FSM. The FSM initializes the input registers and sets the output registers to zero.
- **STEP**: The main state of the FSM. The FSM performs the operation and updates the output registers.

- **LOAD**: The load state of the FSM. The FSM loads the output registers with the final result.
- **DONE**: The final state of the FSM. The FSM sets the done signal and resets the state to IDLE.

The done signal is asserted by sub-controllers to indicate completion of their operation. The ALU controller monitors these flags to determine when to latch the outputs and re-enter the IDLE state. This handshake ensures synchronization between sequential and combinational logic blocks.

Below are two state diagrams describing how their respective controllers flow during runtime:

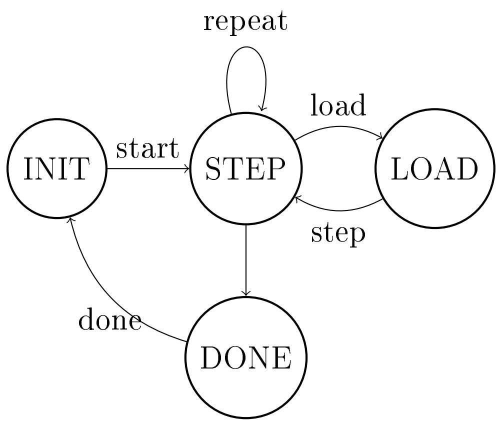

Figure 4: State diagram of the Addition and Subtraction Controllers

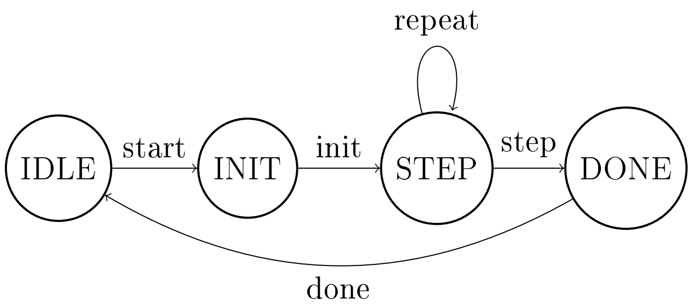

Figure 5: State diagram of the Multiplication and Division Controllers

While most of the states are similar across the different sub-controllers, the *INIT* and *LOAD* states are featured in their own unique manner. The *INIT* state is used to initialize the input registers in the *addition* and *subtraction* operations while the *LOAD* state is used to load registers in the *multiplication* and *division* operations. This is due to the designs of each respective operation and will be discussed in more detail in the following sections. However, the FSM logic for the sub-controllers are all designed to move through each state as the operation is performed.

Furthermore, every operation receives a section before the FSM case logic is entered that initializes any internal wires and output registers used in the operation. This helps each input to be independant from one another and avoid any mismatches in the timing of the inputs. This is denoted in the source files as a check against `RESET` during runtime.

6.2 Adder Controller

The Adder controller is the first of the sub-controllers developed and is responsible for performing the addition operation. It is composed of the 2 main parts:

- **Adder Core**: The core unit that performs the addition operation. It is a simple combinational circuit that takes in two bits and produces a sum and a carry bit.
- **Adder Controller**: The controller that manages the flow of the addition operation. It is a sequential circuit that directs the *Adder Core* to perform the addition of the inputs over multiple clock cycles.

6.3 Adder Core

The *adder core* is the combinational circuit refactored from the previous report. It performs its functions on a bit-by-bit basis, where each bit is added together and the carry is added and passed to the next bit. The difference between the previous implementation and the current implementation is that the logic of the bit-by-bit process is now separated from the combinational logic it was in before. This allows for it to be used either in a combinational or sequential manner, depending on the needs of the design.

The *adder core* remains the same, as it is composed of two half adders, one for each input and one for the carry bits, and an OR gate to determine the final carry. The *adder core* is shown below:

```verilog
module Addition_Core #( parameter WIDTH = 4 ) (
    input wire in1,
    input wire in2,
    input wire carry_in,
    output wire out,
    output wire carry_out
);
    // Internal wires
    wire temp_out, temp_carry_out, carry_overflow;

    // Add the inputs and store the output
    Half_Adder input_adder_instance (
        .in1( in1 ),
        .in2( in2 ),
        .out( temp_out ),
        .carry_out( temp_carry_out )
    );

    // Add the carry
    Half_Adder carry_adder_instance (
        .in1( temp_out ),
        .in2( carry_in ),
        .out( out ),
        .carry_out( carry_overflow )
    );

    // Determine the final carry
    OR or_instance (
        .in1( temp_carry_out ),
        .in2( carry_overflow ),
        .out( carry_out )
    );
endmodule
```

6.4 Adder Controller FSM

From the *adder core*, the *adder controller* is built. It uses an FSM to manage the different states the addition operation can be in and uses the *adder core* to perform each step of the addition of two inputs. The addition FSM first moves into the `INIT` state, where the input registers are initialized. It then moves into a switching between `STEP` and `LOAD` states, where the addition is performed and then is saved to the output registers and for the next carry bit. The FSM is shown below:

```
    case( state )
        INIT: begin // Initialize the addition operation
            done <= 1'b0;
            if( start ) begin
                step_counter <= 0;
                current_in1 <= in1[ 0 ];
                current_in2 <= in2[ 0 ];
                carry_in <= 1'b0;
                state <= LOAD;
            end
        end
        STEP: begin // Perform the addition operation
            if( step_counter < WIDTH ) begin
                current_in1 <= in1[ step_counter ];
                current_in2 <= in2[ step_counter ];
                carry_in <= carry_out;
                state <= LOAD;
            end else begin
                out <= final_out;
                final_carry <= carry_out;
                state <= DONE;
            end
        end
        LOAD: begin // Load the output registers
            final_out[ step_counter ] <= current_out;
            state <= STEP;
            step_counter <= step_counter + 1;

        end
        DONE: begin // Finish the addition operation
            done <= 1'b1;
            state <= INIT;
        end
    endcase
```

There is an additional step that is taken before entering the main case logic in the FSM which simply initializes all the temporary wires and output registers. This is done any time the FSM enters the the *posedge reset* block. This is to ensure that the output registers are properly initialized before every new operation begins.

6.5 Subtractor Controller

The subtractor controller is the second of the sub-controllers developed and is responsible for performing the subtraction operation. It is composed similarly to the adder controller, with the following parts:

- **Subtractor Core**: The core unit that performs the subtraction operation. It is a simple combinational circuit that takes in two bits and produces a difference and a borrow bit.
- **Subtractor Controller**: The controller that manages the flow of the subtraction operation. It is a sequential circuit that directs the *Subtractor Core* to perform the subtraction of the inputs over multiple clock cycles.

Subtractor Core

Like the adder controller, the subtractor controller features a *core* unit that performs its operations in a bit-by-bit manner, with carry bits being passed into and from the core unit. The subtractor core is similar to the adder core, with the differences being the composition of the logic. The subtractor core is shown below:

```
module Subtraction_Core #( parameter WIDTH = 4 ) (
    input wire in1,
    input wire in2,
    input wire borrow_in,
    output wire out,
    output wire borrow_out
);
    // Internal wires
    wire temp_out, temp_borrow_out, borrow_overflow;

    // Subtract the inputs and store the output
    Half_Subtractor input_subtractor_instance (
        .in1( in1 ),
        .in2( in2 ),
        .out( temp_out ),
        .borrow_out( temp_borrow_out )
    );

    // Subtract the borrow
    Half_Subtractor borrow_subtractor_instance (
            .in1( temp_out ),
            .in2( borrow_in ),
            .out( out ),
            .borrow_out( borrow_overflow )
        );

        // Determine the final borrow
        OR or_instance (
            .in1( temp_borrow_out ),
            .in2( borrow_overflow ),
            .out( borrow_out )
        );
    endmodule
```

As opposed to the *adder core*, the *subtractor core* uses a *half subtractors* instead of *half adders*. The half subtractor is a combinational circuit that takes in two bits and produces a difference and a borrow bit. The half subtractor is similar to the half adder, with the only difference being the logic used to produce the output. The *subtractor core* uses an OR gate to determine the borrow bits and any borrow overflow that may occur.

Subtractor Controller FSM

The FSM used for the subtractor controller is identical to the one used for the adder controller. This stark similarity is due to the relationship between the two operations, as they are both feature a similar structure in their core units. Furthermore, the choice to use two distinct controllers for the two operations was made to maintain the modularity of the design and ensure that the operations could be performed independently of one another. Both operations could be implemented together in a unified controller, but there would be a loss of modularity and clarity in the design, since the operations would be conjoined into a single controller. Thus, to avoid this issue, two controllers for the the operations were made and kept separate.

6.6 Multiplier Controller

The multiplier controller is the third of the sub-controllers developed and is responsible for performing the multiplication operation. The conception of the multiplier controller was a bit different than the previous two controllers due to the nature of the multiplication operation. Instead of working bit-by-bit, the controller would work on shifting the inputs, aligning them accordingly based on the multiplication algorithm developed.

Multiplier Core

The *multiplier core* is the heart of the multiplication operation. It performs its operations on an entire input rather than in single bits. Furthermore, the *multiplier core* features the use of Full Adders, a shifter, and a comparator to perform and complete each step of the multiplication process. The *multiplier core* is shown below:

```verilog
 module Multiplier_Core #( parameter WIDTH = 4 ) (
     input wire [ WIDTH-1:0 ] in1,
     input wire [ WIDTH-1:0 ] in2,
     input wire [ WIDTH-1:0 ] partial_high,
     input wire [ WIDTH-1:0 ] partial_low,
     input wire [ WIDTH-2:0 ] step_counter,
     output wire [ WIDTH-1:0 ] out_high,
     output wire [ WIDTH-1:0 ] out_low,
     output wire is_equal
 );
     // Internal wires
     wire [ WIDTH-1:0 ] shift, shift_result, shift_overflow, combined_overflow;
     wire final_carry;
     assign shift = { step_counter, 1'b0 };  // Construct the shift signal

     // Shift the multiplicand according to the step
     nBit_Shift #( .WIDTH( WIDTH ), .OP( 0 ) ) shift_instance (
         .in( in1 ),
         .shift( shift ),
         .out( shift_result ),
         .overflow( shift_overflow )
     );

     // Add the shifted multiplicand to the low side of the partial sum
     Full_Adder #( .WIDTH( WIDTH ) ) adder_low_instance (
         .in1( partial_low ),
         .in2( shift_result ),
         .out( out_low ),
         .final_carry( final_carry )
```

```verilog
    );

    // Add the overflow from the shift to the high side of the partial sum
    Full_Adder #( .WIDTH( WIDTH ) ) adder_overflow_instance (
        .in1( partial_high ),
        .in2( shift_overflow ),
        .out( combined_overflow ),
        .final_carry(  )
    );

    // Add the overflow from the two addition operations to the high side of the partial sum
    Full_Adder #( .WIDTH( WIDTH ) ) adder_final_instance (
        .in1( combined_overflow ),
        .in2( { { ( WIDTH-1 ){ 1'b0 } }, final_carry } ),
        .out( out_high ),
        .final_carry(  )    // Final carry is not used
    );

    // Determine if the current bit being calculated is is equal to 1
    Equal_To #( .WIDTH( 1'b1 ) ) equal_instance (
        .in1( in2[ step_counter ] ),
        .in2( 1'b1 ),
        .out( is_equal )
    );
endmodule
```

The *multiplier core* determines a shift based on the current step of the multiplication. This is composed during the initial stages of the multiplication step that is then used by the shifter. It then undergoes a series *Full Adders* to add the shifted multiplicand to the low side of the partial sum and any overflow from the shift is added to the high side of the partial sum. This process is then repeated for every step of the multiplication, which is determined by the number of bits in the inputs. The *multiplier core* also features a comparator to determine if the current bit being calculated is equal to 1. This is used by the *multiplier controller* to determine if the current step should be performed or not. The design of the *multiplier core* is similar to those of the *adder core* and *subtractor core*, but that work on a whole input rather than a single bit for one clock cycle. This does not capture the true nature of how a clock signal may pass through the hardware. If the design were to be implemented in hardware, the *multiplier core* would perform each step if the *Full Adders* across multiple clock cycles while the shifting would remain mostly the same. However, the implementation of a feature like this one would need to allow for the clock cycles between the *multiplier controller* and the *addition controllers* to be in sync with one another, which is more difficult to align. Hence, for simplicity and modularity, the *multiplier core* was designed to work in a single clock cycle rather than across multiple clock cycles.

Multiplier Controller FSM

The FSM for the multiplier controller is slightly different than the previous two controllers developed ad it features an `IDLE` step and removes the `LOAD` step. The `IDLE` step is used to initialize the output registers without affecting the remaining registers. This is because how the *multiplier core* tracks its partial sums and output registers. In essence, the *multiplier core* builds the output registers as it goes, rather than having to load them in between steps. Hence, the `LOAD` step is removed as well. The FSM for the multiplier controller is shown below:

```
 case( state )
     IDLE: begin // Initialize the multiplication
operation
         done <= 1'b0;
         if( start ) begin
             current_out_high <= { WIDTH{ 1'b0 } };
             current_out_low <= { WIDTH{ 1'b0 } };
             state <= INIT;
         end
     end
     INIT: begin // Perform first multiplication step
         if( is_equal ) begin
             current_out_high <= { WIDTH{ 1'b0 } };
             current_out_low <= in1;
             state <= STEP;
         end else begin
             current_out_high <= { WIDTH{ 1'b0 } };
             current_out_low <= { WIDTH{ 1'b0 } };
             state <= STEP;
         end
         step_counter <= 1;
```

```
        end
        STEP: begin // Remaining multiplication steps
            if( step_counter < WIDTH ) begin
                if( is_equal ) begin
                    current_out_high <= mul_high;
                    current_out_low <= mul_low;
                end

                step_counter <= step_counter + 1;
            end else begin
                out_high <= current_out_high;
                out_low <= current_out_low;
                state <= DONE;
            end
        end
        DONE: begin
            done <= 1'b1;
            state <= IDLE;
        end
    endcase
```

The FSM for the multiplier controller utilizes its `INIT` state slightly differently than the other controllers. The `INIT` state is used to perform the first multiplication step, where the inputs are initialized and then determined based on the assertion of the least significant bit of the second input. Since the `STEP` uses a *partial sum* to add to and from, the `INIT` state is needed to initialize that partial sum before the first step and after can be performed. The `STEP` state is used to perform the remaining multiplication steps, where the inputs are added to the partial sum and the output registers are updated. Once all steps are completed, the `DONE` state is entered, where the output registers are updated and the FSM is reset to the `IDLE` state. From here, the *multiplier controller* takes back control and the ALU controller can proceed to the next operation.

6.7 Divider Controller

The divider controller is the fourth of the sub-controllers developed and is responsible for performing the division operation. The design and implementation of the division was the most complex of the four operations, as it required multiple intermediary steps to be performed before each step could be done. To avoid conflating the main module of the division with too many operations, it was integral to separate the division into 3 parts:

- **Signal Decomposition**: The first intermediary step performed in the division logic that is used to decompose an input signal into its most significant components.
- **Signal Alignment**: The second intermediary step performed in the division logic that is used to align an input to another input based on the MSB of its second input.
- **Divider Core**: The main method of the divider which performs subtraction operation and assigns the output registers.

Divider Core

Most of the changes made to the division operation from the previous report is the addition of the *divider core* module. It is a refactored version of previous combinational logic that has been isolated into its own module. This is in order to promote a modular design and allow for the division module to be used across multiple logic blocks. Similar to how the *multiplier core* was designed, the *divider core* performs its operations on a whole input rather than a single bit for one clock cycle.

The *divider core* uses both the *signal decomposition* and *signal alignment* as well as a *Full Subtractor* and a few *comparators* to perform and complete each step of the dicision process. The *divider core* is shown below:

```
    module Divider_Core #( parameter WIDTH = 4 ) (
        input wire [ WIDTH-1:0 ] in1,
        input wire [ WIDTH-1:0 ] in2,
        output wire [ WIDTH-1:0 ] out,
        output wire [ WIDTH-1:0 ] remainder,
        output wire is_zero,
        output wire is_less,
        output wire has_borrow
    );
        // Internal wires
        wire [ WIDTH-1:0 ] alignment_result;
        wire [ WIDTH-2:0 ] pos;
        wire final_borrow;

        // Align the dividend and divisor
        Signal_Alignment #( .WIDTH( WIDTH ) )
    alignment_instance (
            .in1( in1 ),
            .in2( in2 ),
            .out( alignment_result ),
            .out_pos( pos )
        );

        // Subtract the aligned divisor from the dividend
        Full_Subtractor #( .WIDTH( WIDTH ) )
    initial_subtractor_instance (
            .in1( in1 ),
            .in2( alignment_result ),
            .out( remainder ),
            .final_borrow( final_borrow )
        );

        // Assign the quotient based on the current position
    being worked on
        nBit_Shift #( .WIDTH( WIDTH ), .OP( 0 ) )
    shift_quotient_instance (
            .in( { {(WIDTH-1){1'b0}}, 1'b1 } ),
            .shift( { pos, 1'b0 } ),
            .out( out ),
            .overflow(  )   // Overflow is not used
        );

        // Check if the divisor is zero
        Equal_To #( .WIDTH( WIDTH ) ) equal_instance (
            .in1( in2 ),
            .in2( { WIDTH{ 1'b0 } } ),
            .out( is_zero )
        );

        // Check if in1 < in2
        Less_Than #( .WIDTH( WIDTH ) ) less_than_instance (
            .in1( in1 ),
            .in2( in2 ),
            .out( is_less )
        );

        // Check if there is a final borrow
        Equal_To #( .WIDTH( 1 ) ) final_borrow_instance (
            .in1( final_borrow ),
            .in2( 1'b0 ),
            .out( has_borrow )
        );
    endmodule
```

The *divider core* functions by first extracting the most significant bits using the *signal decomposition* and then aligning the inputs based on the position of that MSB in the *signal alignment* step. From there, the aligned divisor is subtracted from the dividend using a *Full Subtractor*, where a quotient is then placed based on the position determined by the *signal decomposition* step using a *nBit Shifter*. Upon completing its operation, the *divider core* issues a few *comparators* checks to determine some key features of the inputs being calculated. The *comparators* are used by the *divider controller* to set rules for the division operation and determine if the operation is valid or not. The *comparators* used in the *divider core* are:

- **First Equal To Comparator**: The first comparator featured in the final checks of the *divider core* is used to determine if a divisor is zero. This is used by the *divider controller* to determine if the division operation performed is valid or not. If the divisor is zero, the division operation fails and the output registers are not updated.
- **Less Than Comparator**: The second comparator featured is used to determine if the dividend is less than the divider. This is used by the *divider controller* to determine if the division operation is valid or not. As opposed to the zero check, if this comparator fails, then the divisor does not fit into the dividend and a remainder can be set.
- **Second Equal To Comparator**: The third comparator featured is used to determine if there is a final borrow from the subtraction operation. This is used by the *divider controller* to determine if another step is needed. So long as this comparator is true, the division operation will continue in a loop.

Divider Controller FSM

The FSM for the divider controller is identical to the one used for the multiplier controller, with only a slight difference in the `INIT` state. In the `INIT` state, the *divider core* needs to have its quotient and remainder initialized based on the current inputs. Although this step is not strictly necessary the first time the division process works, it helps to ensure that the remaining steps are performed and initialized correctly based on a step counter passed down from the *divider controller* to the *divider core*.

Once the division operation is finished, a `DONE` state is entered, where the output registers are updated and the FSM is reset to the `IDLE` state. From here, the *divider controller* takes back control and the ALU controller can proceed to the next operation.

7 Waveform Analysis

Every operation performed by the ALU produces an output observable in waveform form. These waveforms are tied to the ALU controller's output registers, with each register reflecting part of the computed result. The three key output registers are:

- **p_high_latch**: The high portion of the output register that is used to store the final result of operations with sectional outputs.
- **p_low_latch**: The low portion of the output register that is used to store the final result of most standard operations.
- **flag_latch**: The flag register that is used to represent any single bit results of the operations.

Every output register utilizes the latch discussed in the subsection **Latch Implementation** in order to create a stable output wave to analyze.

Every operation from the opcodes are processed in one continuous waveform that can be represented with these three registers and can be deconstructed to analyze how the different operations are outputted.

7.1 Addition Waveform

The first waveform discussed is the simplest to understand, since it follows a strict pattern that can be easily analyzed. Because of the method used to iterate through the operations, a distinct pattern can be formed that shows a progressive growth of two output signals. The output signals used in the addition operation are as follows:

- **flag_latch**: Represents any carry bits produced by the addition operation.
- **p_low_latch**: Represents the output of the addition operation.

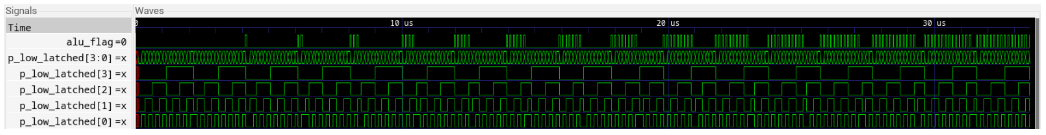

Figure 6: Addition waveform from the ALU from GTKWave

Addition Timing Diagram

The waveform can be decomposed into sectional outputs, where each section represents a single bit, such as what is shown in figure 6. As the addition operation produces a single-segment result, the `p_high_latch` remains inactive. Only `p_low_latch` is engaged, representing a 4-bit result, while `flag_latch` captures overflow through the carry bit. The output of `p_low_latch` is observed bit-wise, labeled `OUT1` through `OUT4`, with bit significance increasing from LSB to MSB.

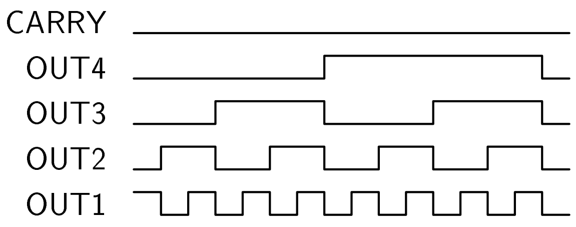

Figure 7: Timing diagram of the addition operation, showing the first complete input cycle

The output of the addition operation follows the same general pattern for its `OUT` signals with the only register showing any changes over time being the `CARRY` signal. At the end of the first complete input cycle, the `CARRY` signal briefly rises, increasing its duration with each successive addition of larger numbers.

For example, the next input cycle following the logic adds `0001` instead of `0000` to all possible combinations of `in2`. This results in an output shown as:

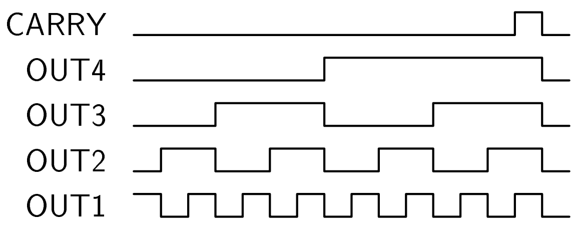

Figure 8: Timing diagram of the addition operation, showing the second complete input cycle

The next input cycle would then be `0010` instead of `0001`, and results in the following output:

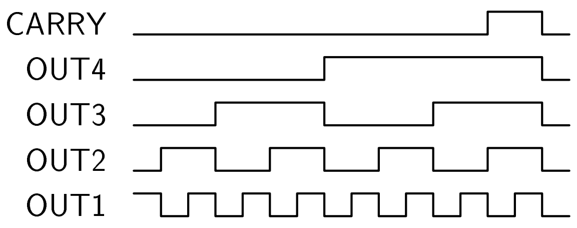

Figure 9: Timing diagram of the addition operation, showing the third complete input cycle

This is the overall growth and pattern that follows the addition operation logic as larger and larger numbers are added.

7.2 Subtraction Waveform

The second waveform discussed is the subtraction operation. Subtraction reuses the same latches as addition but interprets the flag register as a borrow bit and exhibits a decaying, rather than growing, flag waveform.

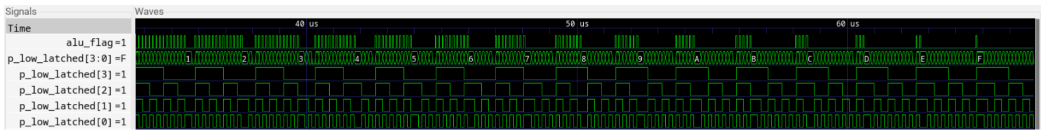

Figure 10: Subtraction waveform from the ALU

Subtraction Timing Diagram

The subtraction operation is decomposed identically as the addition operation was, with the same output signals being used. However, the `CARRY` signal is now replaced with a `BORROW` signal, which is used to represent any borrow bits produced by the subtraction operation. Furthermore, the subtraction operation experiences a decay in its `BORROW` signal instead of a growth like the addition operation.

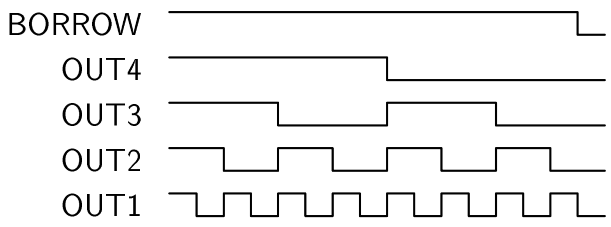

Figure 11: Timing diagram of the subtraction operation, showing the first complete input cycle

Negative results are produced via two's-complement: the output bits are inverted, then the borrow bit is asserted to complete the +1 step. Essentially, the outputs are the opposite from that seen in the addition operation. So instead of the output being `0000` for the first input cycle, it is now `1111` with a borrow of `1`. For the remaining cycles, the output is the same, but the borrow signal decays in length.

Using the same example as before, the output from the next input cycle is:

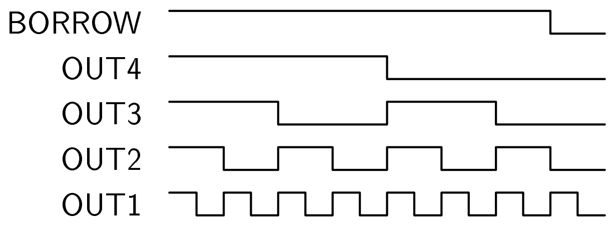

Figure 12: Timing diagram of the subtraction operation, showing the second complete input cycle

This is the overall decay pattern that follows the subtraction operation logic as larger and larger numbers are subtracted.

7.3 Multiplication Waveform

The third waveform discussed is the multiplication operation. Multiplication in the ALU differs from addition and subtraction because it produces twice as many bits, which are split into two outputs to maintain uniformity:

- **p_high_latch**: Represents the high output bits of the multiplication operation.
- **p_low_latch**: Represents the low output bits of the multiplication operation.

This is to keep the size of bits similar across the system and is due to the increased size of the output produced by multiplying two numbers.

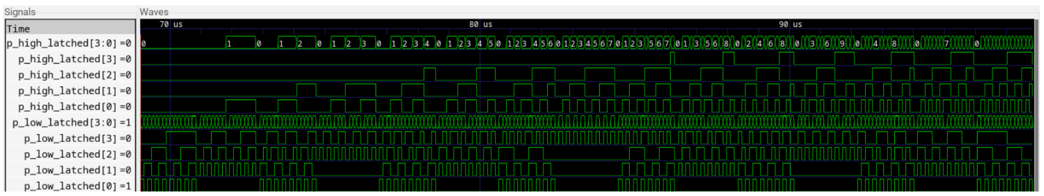

Figure 13: Multiplication waveform from the ALU

Multiplication Timing Diagram

The waveform can be decomposed into sectional outputs, similar to how they are demonstrated before, and is shown in figure 13. Since the output representation has changed, the `p_high_latch` becomes active and the `flag_latch` is now disabled. Both high and low latches capture a predesignated 4-bit result, with the latch method remaining consistent. Both high and low outputs are also observed bit-wise and are labeled `OUT1` through `OUT4` for both outputs respectively. Furthermore, their bit significance increases the same way as before: by increments of one for inputs one and two.

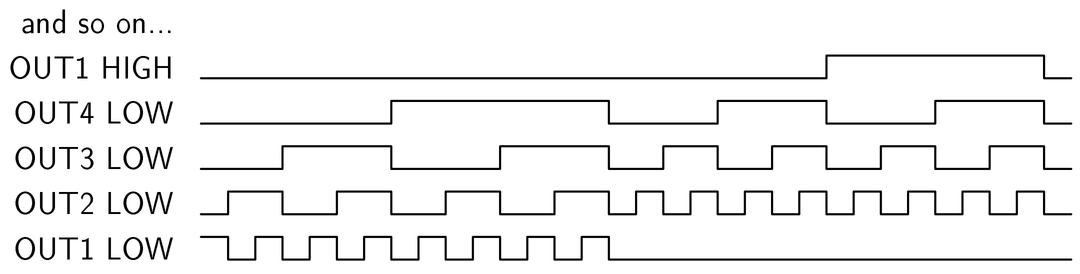

Figure 14: Timing diagram of the multiplication operation, showing the next two complete input cycles after the first

The timing diagram above starts with an input of `0001` instead of `0000` since the output produces no significant information; the output is 0. Furthermore, it is observed that the waveform is being shifted up the output register. This is due to the method used to solve the multiplication in binary, where the output is shifted based on the bits of the multiplicand. However, this pattern does not remain consistent across the entire multiplication operation as shown below:

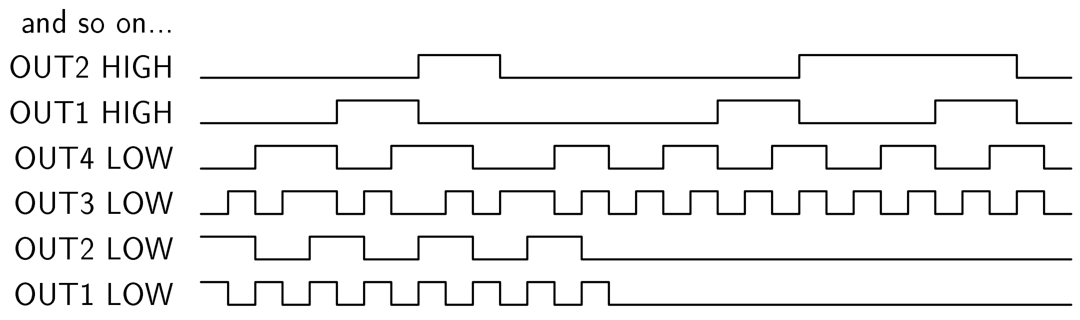

Figure 15: Timing diagram of the multiplication operation, showing the next two complete input cycles after the previous

As the multiplication operation continues, the upper output registers are utilized more frequently and present a growth similar to that seen in the addition operation. The waveforms slowly grow into the pattern observed in the input cycles in these upper bits, like how 1 shows.

Mirror Symmetry Effect

A key observation of the lower output registers in the multiplication operation timing diagrams is that the pattern between the multiplicand of any number M and $16-M$ are bit-wise complements of one another. For this to occur, `M` would need to be greater than `0` and less than the `width` of the bits; here the `width` is assigned to `4`. In essence, the output of one half of the waveform is a mirror image of its other half. This is a property of **half-wave symmetry**, where the output of the waveform is a mirror image of itself across a vertical line drawn through the midpoint of a waveform. This mirror image can be described as the two's complement of the first half of the waveform.

In the waveform shown in 13, the mirror can be observed at the midpoint `8`.

7.4 Division Waveform

The fourth waveform discussed is the division operation. Division in the ALU is similar to multiplication in that it utilizes the two same output registers as the multiplication operation:

- **p_high_latch**: Represents the quotient of the division operation.
- **p_low_latch**: Represents the remainder of the division operation.

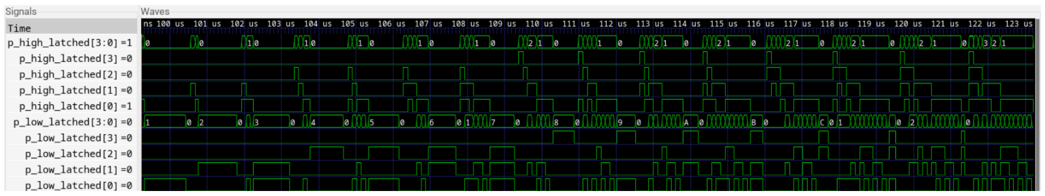

Figure 16: Division waveform from the ALU

Division Timing Diagram

The waveform is decomposed into similar sectional outputs as before, with the `p_high_latch` representing the quotient calculated and the `p_low_latch` representing any remainder. Thus, both the `p_high_latch` and the `p_low_latch` are still enabled during this operation. The same method of incrementing is applied here as it was done previously.

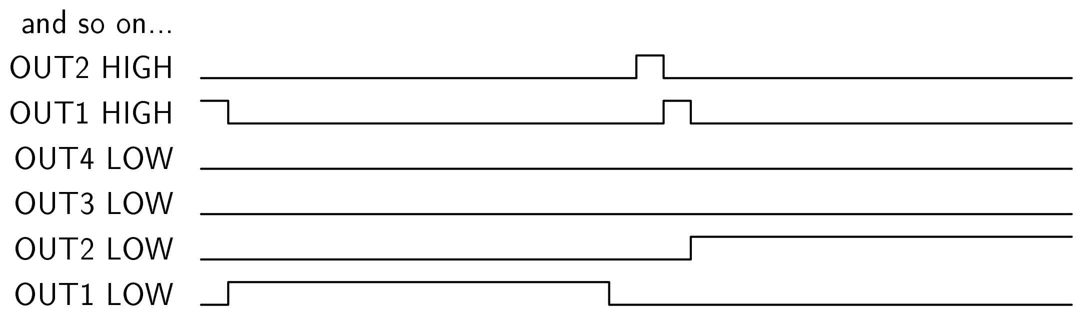

Figure 17: Timing diagram of the division operation, showing the next two complete input cycles after the first.

The diagram begins at the second input cycle, when `in1` is `0001` to avoid dividing numbers by `0`. Furthermore, it shows both the remaining bits and the quotient from the operations `0001` and `0010`, capturing their results as `in2` is iterated upon. The pattern showcased maintains a consistent and predictable flow, similarly seen in the addition operation.

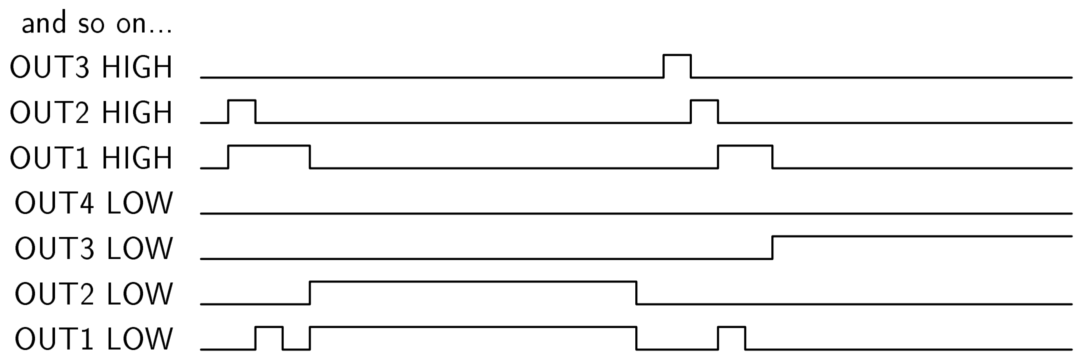

Figure 18: Timing diagram of the division operation, showing the next two complete input cycles after the previous.

Every iteration produces a step on `p_high_latch` to the quotient and updates `p_low_latch` to the new remainder, which remains constant for the following shift phase. This alternating pulse-and-hold pattern yields a rising-staircase on the remainder bus, since successive subtractions accumulate interleaved with single-cycle high pulses on the quotient bus. When the partial remainder falls below the divisor, the quotient bit is `0`, and the `p_high_latch` remains low. Furthermore, the `p_low_latch` shifts the previous remainder forward. Only when the remainder is greater than the divisor is when a non-zero stair-step occurs.

8 Conclusion

Throughout the project, a successful 4-bit ALU in Verilog is built by applying a clean, hierarchical design methodology. Simple combinational cores for each arithmetic and logic function, driven by a central FSM-based controller allowed for a robust ALU with greater capabilities as its iterations were expanded on. Further separating core modules from their subcontrollers made the codebase more maintainable and simplified timing-mismatches encountered while debugging. Extensive testing using testbenchs helped to cover every possibility in the ALU and GTKWave waveform analysis demonstrated that every operation behaves as expected. The addition and subtraction operations exhibited a growth/decay pattern in their carry/borrow flags, multiplication operations showcase a form of half-wave symmetry, and division operations producing a distinctive sawtooth-esq remainder pattern.

Looking ahead, there are several avenues for enhancement:

- **Signed-Number Support:** Extend addition, subtraction, and comparison logic to handle two's-complement operands.
- **Floating-Point Unit:** Integrate IEEE-754 compliant modules for real-number arithmetic.
- **FPGA Synthesis & Timing Closure:** Port the design to an FPGA platform, optimize for setup/hold times, and verify performance under real-world clock constraints.
- **Performance Optimizations:** Explore look-ahead carry adders, non-restoring division algorithms, or pipelined control to reduce cycle counts.

These extensions will deepen the ALU's capabilities and further validate the scalability of the modular, controller-driven architecture.