CSEE E6863 FORMAL VERIF HW SW SYSTEMS

Final Project Report

Group Members: Weijie Wang (ww2739), Zhenqi Li (zl3508), Siyuan Li (sl5590)

1 Introduction

In this project, our team attempted to formally verify an ALU (Arithmetic Logic Unit) with the FIFO (First In, First Out) function. We refer to alu_fifo_design.sv from Appledore22's Github repository [1] as the source code to do verifications and further modifications on it. ALU is a fundamental component in digital systems and performs arithmetic and logical operations. Integrating a FIFO function enhances the ALU's capability by efficiently managing data flow, which is crucial in pipeline architectures and buffering applications. This project aims to verify the functionality and performance of an ALU with an integrated FIFO to ensure it meets design specifications and operates reliably under various conditions.[1]

2 Assertions for the Pipelined Arithmetic Processing System

The top module implements a pipelined arithmetic processing system consisting of two FIFOs sandwiching an ALU. When data enters the system, it arrives in a 10-bit format where bits `[9:8]` specify the operation (add, subtract, multiply, or divide), bits `[7:4]` contain the second operand, and bits `[3:0]` contain the first operand. This data is initially buffered in the input FIFO, processed by the ALU, and then the 9-bit result is stored in the output FIFO before being presented to the outside world. This architecture allows for efficient handling of data streams, especially when input, processing, and output operate at different rates.

The assertions in this design play a crucial role in verifying the correct behavior of the system. Assertions are well known in formal verification as properties that must hold true during operation, acting as continuous monitors to catch potential design flaws or implementation errors. In the top module, these assertions are implemented as follows:

2.1 FIFO-ALU Handshaking Verification

Assertion Name: `TOP_fifo_alu_valid`

```
assert property (@(posedge clk)
    f_in.valid_out && !alu1.ready |-> ##1 !f_in.empty);
```

This assertion verifies that when the input FIFO has valid data (`valid_out`) but the ALU isn't ready, the FIFO shouldn't be empty in the next cycle. This ensures proper handshaking between the FIFO and ALU, preventing data loss scenarios where the FIFO might prematurely discard data before the ALU can process it.

[1] `https://github.com/weijiekwang/FIFO_ALU_Formal_Verification`

2.2 ALU-Output FIFO Handshaking Verification

Assertion Name: `TOP_alu_fifo_ready`

```
assert property (@(posedge clk)
    alu1.ready && !f_out.full |-> ##1 f_out.empty);
```

This assertion verifies that when the ALU is ready with data and the output FIFO isn't full, the FIFO should be empty in the next cycle to accept the new result. This prevents potential data overflow situations where the ALU might try to send data when there's nowhere to store it.

2.3 Data Integrity Verification

Assertion Name: `TOP_data_flow`

```
assert property (@(posedge clk)
    f_in.valid_out |-> (f_in.data_out[3:0] == alu1.data1 &&
                        f_in.data_out[7:4] == alu1.data2 &&
                        f_in.data_out[9:8] == alu1.operand));
```

This assertion ensures that data is correctly routed from the input FIFO to the ALU. It also verifies that when the input FIFO has valid data, the operands and operation code are correctly connected to the ALU's inputs. This catches any potential signal routing errors or timing misalignments.

2.4 Result Propagation Verification

Assertion Name: `TOP_result_prop`

```
assert property (@(posedge clk)
    alu1.ready |-> (alu1.result == f_out.data_in));
```

This assertion ensures that when the ALU has completed its operation (`ready`), its result is correctly captured by the output FIFO. This prevents scenarios where results might be corrupted or lost during transfer between modules.

2.5 Reset Behavior Verification

Assertion Name: `TOP_reset`

```
assert property (@(posedge reset)
    (f_in.empty && f_out.empty && alu1.ready));
```

This assertion checks that during reset, both FIFOs are emptied and the ALU returns to its ready state. Proper reset behavior is critical for system initialization and recovery from error conditions.

2.6 Conclusion

These assertions are essential because they help catch design bugs early in the verification process. They continuously monitor the design during simulation and can be used in formal verification tools to mathematically prove that these properties hold under all possible conditions. Without these assertions, subtle bugs in the handshaking, data transfer, or reset logic might go undetected until the system fails in actual operation. The assertions serve both as documentation of expected behavior and as active verification monitors, making the design more robust and reliable.

3 FIFO Verification

3.1 FIFO Structure

In this section, we aim to verify the `fifo_in` and `fifo_out` modules, which are responsible for storing the input and output data of the ALU module, respectively. Since they are symmetric, we only need to insert assertions for verification and modification within the FIFO module definition. The five external inputs are defined: `clk`, `reset`, `data_in[9:0]`, `ready` (which receives the output ready signal from the ALU to control the FIFO's data reading and forwarding to the ALU for computation), and `valid` (a valid signal that controls writing data into the FIFO from external sources). Four external outputs are defined: `data_out[9:0]`, `valid_out` (output valid signal), `empty`, and `full`.

Internally, an 8-bit memory array `mem[8]` is defined to store the data, and `wptr[2:0]` and `rptr[2:0]` are used to control the write and read operations. These pointers increment after each operation. The finite state machine, defined as `state[1:0]`, is used to manage the states: `send[0]` is the state for sending data, `wait_ready[1]` is an intermediate state where the system waits until the external ready signal is received, and `end_tx` indicates the completion of data transmission, where FIFO enters the end state and is ready to receive new data. The FIFO operates as follows:

When the `valid` signal is high, indicating valid input data, the FIFO writes the data into memory and updates the `write pointer` (`wptr`). If the FIFO is full, the `full` signal is set high, indicating that no more data can be written.

When the `ready` signal is high and the FIFO contains data, the FIFO outputs the data and updates the `read pointer` (`rptr`). If the FIFO is empty, the `empty` signal is set high, indicating that there is no data to read.

3.2 Verification analysis

During the formal verification of the FIFO design using JasperGold, we focused on key functionalities, including reset behavior, signal handling for valid and ready conditions, management of full and empty states, and the correct increment of read and write pointers. Assertions were applied to validate these functionalities, and several issues were detected due to logical flaws in the original implementation. These issues included multiple-driven conflicts on the ‘empty‘ signal, deadlock during the reset phase, synchronization delays between pointer updates (‘wptr‘, ‘rptr‘) and state signals (‘empty‘, ‘full‘), and

assertion validation errors. Through waveform analysis of the assertion failures, we identified problems in the read and write logic, which caused incorrect pointer increments and improper handling of full and empty conditions. After analysis, we modified the source code logic, resolved synchronization delays, and added mechanisms to handle deadlock scenarios. Verification runs confirmed the correctness of the updated design, with all assertions passing successfully. The following sections introduce the five main issues encountered, their causes, and the solutions implemented.

3.2.1 Problem 1: Multiple-Driven Conflict on `empty`

During the verification process, we detected a multiple-driven conflict on the `empty` signal. The root cause of this conflict lies in the behavior of two blocks operating simultaneously in the same clock cycle: In the first `always` block, the write pointer (`wptr`) was updated, and `empty` was assigned a value of `0`. Simultaneously, in the second `always` block, since both `wptr` and `rptr` were reset to `0`, `empty` was also assigned a value of `1`.

```
always@(posedge clk or posedge reset)
  begin
    if (reset)
      begin
        wptr <= 0;
        rptr <= 0;
        valid_out <= 0;
        full <= 0;
        empty <= 1;
        state <= 0;
      end
    else
      if(valid)
          begin
            if(wptr == (rptr -1))
              begin
                full <= 1;
                valid_out <= 0;
                empty <= 0;
              end
            else
              begin
                  mem[wptr] <= data_in;
                  wptr <= wptr + 1;
                    empty <= 0;
              end
          end
  end
```

```
always@(posedge clk)
  begin
    if(reset)
      rptr <= 0;
    else
      begin
        if(rptr == wptr)
          empty <= 1;
      if(!empty)
      begin
      case (state)
        send:begin
                    data_out <= mem[rptr];
                    valid_out <= 1;
                    rptr <= rptr+1;
          if(ready)
            state <= end_tx;
          else
                  state <= wait_ready;
                end
        wait_ready:begin
                    if(ready)
                      state<= end_tx;
                    else
                      state <= wait_ready;
                end
          end_tx:begin
                valid_out <= 0;
            state <= send;
          end

      endcase
      end
      end
  end
```

Figure 1: The original code of FIFO

As Figure 1 shows, this simultaneous assignment led to conflicting values for `empty`.

To address this issue, we consolidated the logic for `empty` and `full` state determination into a single `always` block. By doing so, we prevented simultaneous assignments and ensured consistent evaluation of `empty` and `full`. However, this change introduced a secondary issue related to the reset phase, which we will explain in Problem 2.

3.2.2 Problem 2: Deadlock After Reset

After we consolidated the `empty`, `full`, and `wptr` logic into a single `always` block, the system exhibited a deadlock state during reset. After reset, both `wptr` and `rptr` were set to `0`, causing the FIFO to remain stuck in the `empty` state. Since `rptr` depends on the `!empty` condition, it could not proceed, leading to a frozen state. The result of a conflict driven by multiple players, as Figure 2 shows.

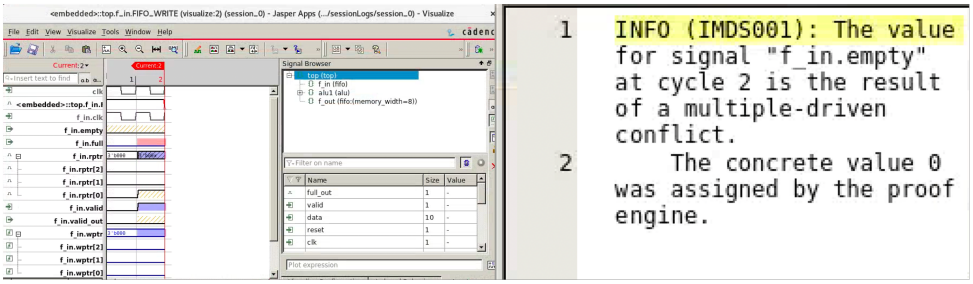

Figure 2: The multiple-driven conflict of empty

```
if (reset) begin
    wptr[0] <= 0;
    wptr[1] <= 0;
    wptr[2] <= 0;
    rptr <= 0;
    valid_out <= 0;
    full <= 0;
    empty <= 1;
    state <= 0;
    reset_done <= 0;
    update_done <=0;
    for (int i = 0; i < 8; i++) begin
       mem[i] <= 0;
    end
end
else if (!reset_done && !update_done) begin
        if (valid) begin
              reset_done <= 1;
              mem[wptr] <= data_in;
              empty <= 0;
              if (wptr == (depth -1)) begin
                  wptr <=0;
                  end else begin
                  wptr <= wptr+1;
              end
        end
end
```

Figure 3: The code of `reset_done`.

To address this issue, we introduced an internal signal `reset_done`, which was initialized to `0` during reset. After reset, we forced a write pointer (`wptr`) operation to ensure proper initialization. Once the forced pointer operation was completed, we set `reset_done` 3 to `1` to enable normal FIFO operations. This adjustment resolved the deadlock issue and ensured that both `rptr` and `wptr` could operate correctly after reset.

3.2.3 Problem 3: Multiple-Driven Conflicts on `valid_out` and `rptr`

During the verification process, we observed multiple-driven conflicts on the `valid_out` and `rptr` signals. These issues arose because:

- Both signals were driven by multiple assignments across two `always` blocks.
- Additionally, `rptr` was redundantly defined during the reset phase.

To address these issues, we cleaned up the redundant logic. We removed duplicate assignments to `valid_out` and `rptr` and consolidated their assignments into a single `always` block. This adjustment eliminated the multiple-driven conflicts and ensured the correct behavior of both signals.

3.2.4 Problem 4: Synchronization Delay Between Pointer Updates and State Signals

The fourth issue we encountered was the delay between the updates of the `wptr` and `rptr` pointers and the evaluation of the `empty` and `full` states. Since both `wptr` and `rptr` are updated on the rising edge of the clock, the `empty` and `full` states require one additional cycle to reflect these updates accurately.

```
// verify the full
FIFO_full0: assert property (@(posedge clk) valid && wptr == (rptr - 1) |-> full && !empty);

// verify the full
FIFO_full: assert property (@(posedge clk) $past(valid && wptr == (rptr - 1)) |-> full && !empty);
```

Figure 4: Asseartion comparsion

To address this, we used the `$past` operator in our assertions to reference the previous states of `wptr` and `rptr` as Figure 4 shows. By doing so, we ensured that the `empty` and `full` states were evaluated correctly based on the pointer values from the previous clock cycle. This approach resolved the timing mismatch and aligned the state evaluations with pointer updates.

3.2.5 Problem 5: Logic Error in Pointer Updates and State Evaluations

The fifth issue we encountered was a logic error in the updates of `rptr` and the evaluation of the `empty` and `full` states. In the original code, `rptr` was updated in one `always` block, while `empty`, `full`, and `wptr` were handled in another block. This separation caused problems because the states were being checked before `rptr` was updated.

```
end else if(update_done) begin
    if(wptr == (rptr -1)) begin
        full <= 1;
        empty <= 0;
    end
    else if (rptr == wptr) begin
        empty <= 1;
        full  <= 0;
    end
update_done <= 0;
end
```

Figure 5: The code of `update_done`.n

To address this, we merged the two `always` blocks into one. In the updated logic, both `wptr` and `rptr` are updated first, followed by the evaluation of the `empty` and `full` states. Additionally, we introduced a new internal signal called `update_done` as Figure 5 shows, which ensures proper sequencing. When `update_done` is set to `0`, the FIFO performs pointer updates for `wptr` and `rptr`. Once the updates are complete, we set `update_done` to `1`, allowing the evaluation of the `empty` and `full` states. After the evaluation is finished, we reset `update_done` to `0`, returning the system to the pointer update phase.

This restructuring ensured that the states are only evaluated after the pointers have been updated, effectively resolving the synchronization issue.

3.3 Verification Results

After we addressed the aforementioned issues, we conducted formal verification using JasperGold. We used assertions and cover properties to validate the reset behavior, ensuring that all signals, including `empty`, `full`, `wptr`, and `rptr`, were correctly initialized. We also verified the correct evaluation of the `empty` and `full` states, confirming that these states behaved as expected under different operating conditions. Additionally, we ensured proper synchronization between the updates of the `rptr` and `wptr` pointers, verifying that pointer updates and state evaluations occurred in the correct order. Cover properties were employed to guarantee that the updates of `wptr` and `rptr` happened as intended. The results in Figure 6 and Figure 7 showed that all assertions and cover properties passed successfully, confirming the correctness and robustness of the revised FIFO design.

```
// reset: fifo ->reset
fifo_reset1: assert property (@(posedge reset) (empty && !full && (wptr == 0) && (rptr == 0) &&
(state == 0) && valid_out == 0));

// reset: fifo -> !valid  && !ready
fifo_reset2: assert property (@(posedge reset) (!valid && !ready));

// verify the full
FIFO_full: assert property (@(posedge clk) $past(valid && wptr == (rptr - 1)) |-> full && !empty);

// verify the empty
FIFO_empty: assert property (@(posedge clk) $past(valid && (wptr == rptr) && reset_done) |-> !full &&
empty);

//cover wptr add
FIFO_wptr: cover property (@(posedge clk)
    (wptr != 7) |-> (wptr == $past(wptr) + 1));

//cover rptr add
FIFO_rptr: cover property (@(posedge clk)
    (rptr != 7) |-> (rptr == $past(rptr) + 1));

//fifo full -> no change
FIFI_write_full: assert property (@(posedge clk) full && valid |-> (wptr == $past(wptr)));

// fifo empty -> no change
FIFO_read_emtpy: assert property (@(posedge clk) empty && ready |-> (rptr == $past(rptr)));

// fifo wptr_add
FIFO_wptr_add: assert property (@(posedge clk) $past(reset, 2) && valid && !empty && !full |-> (wptr
== $past(wptr) + 1));

//fifo read_add
FIFO_rptr_add: assert property (@(posedge clk) $past(reset, 2) && !empty |-> rptr == ($past(rptr) +
1));
```

Figure 6: The assertions and covers

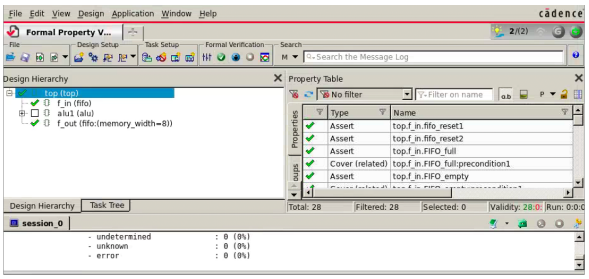

Figure 7: The screenshot of result

3.4 Verification Challenge with `rptr_add`

During the verification process, we observed that the assertion for `rptr_add` did not pass successfully. To gain further insights, we added a cover property to monitor the behavior of `rptr_add` under different conditions. However, when comparing the waveform results from the cover property and the assertion, we noticed discrepancies between the two.

We suspect that the issue lies in the design of our assertion conditions, which might not fully capture the intended behavior of `rptr_add`. Despite multiple attempts to refine the assertion logic, we were

unable to resolve the mismatch. Below, we present the waveform comparison between the cover property and the assertion results for `rptr_add`, as Figure 8 shows.

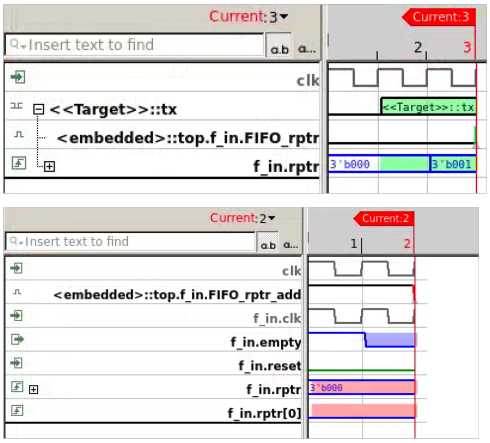

Figure 8: Waveform Comparison Between Cover Property and Assertion for `rptr_add`

This comparison reveals potential areas where the assertion logic may need further refinement. Future work will focus on carefully analyzing the cover property behavior and aligning it more accurately with the assertion conditions to ensure successful verification of `rptr_add`.

4 ALU Verification

4.1 ALU Structure

The ALU is implemented as a hardware module with the following inputs, outputs, and internal components:

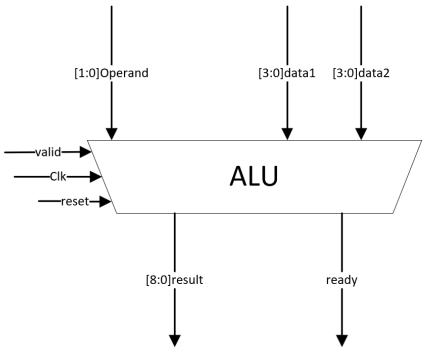

Figure 9: ALU Data-path

Inputs and Outputs

Inputs:

- **clk**: Clock signal for synchronous operations.

- **reset**: Resets internal states and registers.
- **valid**: Indicates when the input data is valid.
- **data1** (4 bits): First operand for arithmetic operations.
- **data2** (4 bits): Second operand for arithmetic operations.
- **operand** (2 bits): Specifies the operation to perform:
  - `00`: Addition
  - `01`: Subtraction
  - `10`: Multiplication
  - `11`: Division

Outputs:

- **result** (9 bits): Stores the operation output.
- **ready**: Indicates when the computation is complete.

Internal Registers

- **state** (1 bit): Tracks the current state:
  - `compute`: Active state for processing operations.
  - `wait_state`: Intermediate state for multi-cycle operations.
- **count** (5 bits): Tracks the number of cycles elapsed during multi-cycle operations.
- **operand_latch** (2 bits): Latches the operation for multi-cycle computations.
- **data1_latch, data2_latch** (4 bits): Latches the input operands for multi-cycle computations.

Operation Flow

Operation Modes:

- **Addition** (`operand = 00`): Executes in a single cycle, setting `result` and `ready` immediately.
- **Subtraction** (`operand = 01`): Executes in a single cycle, setting `result` and `ready` immediately.
- **Multiplication** (`operand = 10`):
  - Enters `wait_state` and latches inputs for multi-cycle computation.
  - Completes after the configured number of cycles, updating `result` and setting `ready`.

- **Division** (`operand = 11`):
  - If `data2` is non-zero, behaves like multiplication with multi-cycle support.
  - If `data2` is zero, outputs `result = 0`.

State Transitions:

- In `compute` state, operations are processed based on the `operand`.
- In `wait_state`, the `count` register increments until the required cycles are completed, transitioning back to `compute`.

4.2 Verification Analysis

To verify the correct functionality of the Arithmetic Logic Unit (ALU), assertions are implemented for each arithmetic operation. These assertions ensure that the ALU produces the expected results under the specified conditions.

```
// Assertion to verify arithmetic functionality in ALU
ADD_OPERATION_CHECK: assert property (@(posedge clk) disable iff (reset)
    (valid && operand == 2'b00 && state == 0) |-> ##1 (result == data1 + data2));

SUB_OPERATION_CHECK: assert property (@(posedge clk) disable iff (reset)
    (valid && operand == 2'b01 && state == 0) |-> ##1 (result == data1 - data2));

MUL_OPERATION_CHECK: assert property (@(posedge clk) disable iff (reset)
    (valid && operand == 2'b10 && state == 0 && count == cycles - 1) |-> ##1 (result == data1_latch * data2_latch));

DIV_OPERATION_CHECK: assert property (@(posedge clk) disable iff (reset)
    (valid && operand == 2'b11 && data2 != 0 && count == cycles - 1) |-> ##1 (result == data1_latch / data2_latch));
```

Figure 10: ALU Assertions

Addition Operation Assertion

Explanation:

- Ensures that when the operation is addition (`operand == 00`), the result is computed as `data1 + data2`.
- The assertion triggers if the input is valid, the ALU is in the `compute` state (`state == 0`), and the clock edge occurs.
- The result must be available in the next clock cycle (`1`).
- Addition is a single-cycle operation, so no intermediate states or delays are involved.

Subtraction Operation Assertion

Explanation:

- Similar to addition, but ensures the result is computed as `data1 - data2`.
- Subtraction is also a single-cycle operation, so the result is ready in the next clock cycle.

Multiplication Operation Assertion

Explanation:

- Verifies the multiplication operation when `operand == 10`.
- Ensures the computation completes after a configured number of cycles (`cycles`).
- The data inputs (`data1` and `data2`) are latched to `data1_latch` and `data2_latch` to prevent changes during the multi-cycle operation.
- The assertion checks that after the required number of cycles, the result is correctly computed and stored in `result`.

Division Operation Assertion

Explanation:

- Verifies the division operation when `operand == 11`.
- Checks that the divisor (`data2`) is non-zero to avoid illegal division.
- Similar to multiplication, the operation completes after the configured number of cycles, using latched data inputs.
- The result is validated after the last cycle (`count == cycles - 1`).
- For division by zero, additional logic ensures `result == 0`, but this is not covered in this assertion.

4.3 Verification Results

From the 11 verification results, it can be observed that the assertions for Addition (ADD) and Subtraction (SUB) operations passed verification (green checkmarks), while the assertions for Multiplication (MUL) and Division (DIV) operations failed (red crosses). This indicates that there might be issues in the implementation of multiplication and division functionalities.

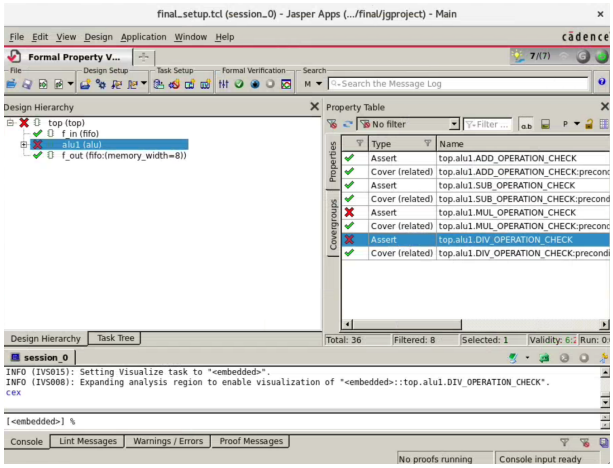

Figure 11: ALU Verification Results

The verification results indicate failures in the multiplication operation for two test cases:

- **Case 1:** Multiplication of `0000 * 0000` produced an incorrect result of `9'h01a` (decimal 26), whereas the expected result is `0`.

Figure 12: ALU Multiplication Verification Waveform 1

- **Case 2:** Multiplication of `1000 * 0100` (decimal `8 * 4`) produced an incorrect result of `9'b0001_1110` (decimal 30), whereas the correct result should be `32`.

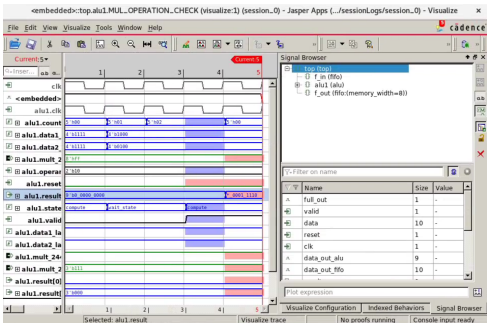

Figure 13: ALU Multiplication Verification Waveform 2

The testing process aimed to verify the correctness of the multiplication operation in the ALU. Initially, the test case `0000 * 0000` was used, which produced an incorrect result of `9'h01a` (decimal 26).

Suspecting that the input `0000 * 0000` might interfere with the assertion mechanism or internal computation, the assertion was modified to ensure that the inputs could not be `0`. Following this change, a second test case `1000 * 0100` (decimal `8 * 4`) was executed. However, the result was still incorrect, yielding `9'b0001_1110` (decimal 30), indicating that the issue was not related to zero inputs but likely a more fundamental problem in the multiplication logic.

A similar issue was observed in the division operation, where test cases also failed to produce the expected results, suggesting that the root cause may affect both multiplication and division functionalities.

5 Extra Work

In preparation for the project, the tutorials from Lars Fischer and Michael Theobald were completed and learned. Several questions were solved to get more understanding of the Formal Verification methodology, including how to write assertions and how to analyze them through the visualization on the Jasper.

6 Appendix

Weijie Wang contributed in the early stage of writing assertions for the FIFO and ALU modules, finding that there might be bugs in the source code. He is also responsible for writing the assertions in the top module to make sure the transitions between FIFO and ALU, and vice versa, go through smoothly.

Zhenqi Li contributed to the entire model analysis, FIFO verification, and the early stage of ALU verification, finding the logic issues in the FIFO and ALU models. He is also responsible for debugging the bugs in the FIFO source code and trying to find the transition between FIFO and ALU.

Siyuan Li contributed to the entire model analysis, ALU verification, and the early stage of FIFO verification, finding the issues in the ALU model. She is also responsible for debugging the bugs in the ALU source code and trying to find the transition between FIFO and ALU.

References

[1] appledore22. alu_fifo_design.sv. `https://github.com/appledore22/ALU_FIFO/blob/master/alu_fifo_design.sv`, 2024. Accessed: 2024-12-24.