# CSEE E6893 Formal Verification of Hardware and Software Systems
# Final Project Report
# Formal Verification of a 16-bit SIMD Processor

**Sai Vittal Battula (sb4701); Manish Shankar (mr4264)**

**Note: No conflicts in contribution. The project was done collaboratively.**

## Introduction:

Formal verification is the process of mathematically proving the correctness of a hardware or software design. In the context of a simple SIMD (Single Instruction Multiple Data) processors written in Verilog, formal verification involves using assertions to specify properties that the design must satisfy, and then using a formal verification tool to check whether these properties hold for all possible inputs and behaviors of the design.

The approach used by us to formally apply verification techniques on our SIMD processor in Verilog included the following steps:

1. Write a high-level specification: This specification could include the input and output formats for the processor, as well as any constraints on the data being processed.
2. Write Verilog code for the processor design, including any necessary modules and functions.
3. Write assertions in Verilog that capture the properties specified in the high-level specification. These assertions could be placed at key points in the design, such as at the inputs and outputs of modules, to check that the design is behaving as expected.
4. Use the formal verification tool to check the correctness of the design. This involved running the tool on the Verilog code and assertions and examining the results to see if any errors or violations of the specification are detected.
5. If any errors or violations are found, debug the design and re-run the formal verification tool until the design is deemed correct.

## Design Under Test:

The design at the focus of our project is a simple SIMD processor that has a 16-bit SIMD ALU [1] at its core, which performs 2's complement calculations. These calculations require two clock cycles to complete: the first cycle is used to load values into the registers, and the second cycle performs the actual operations. The processor uses 6-bit opcodes to select the desired function, and the instruction code (including the opcode) is 18 bits in total. The ALU is integrated into a simple processor with a 5-stage (no pipeline), with each stage having a delay of 1 cycle to match the delay of the ALU.

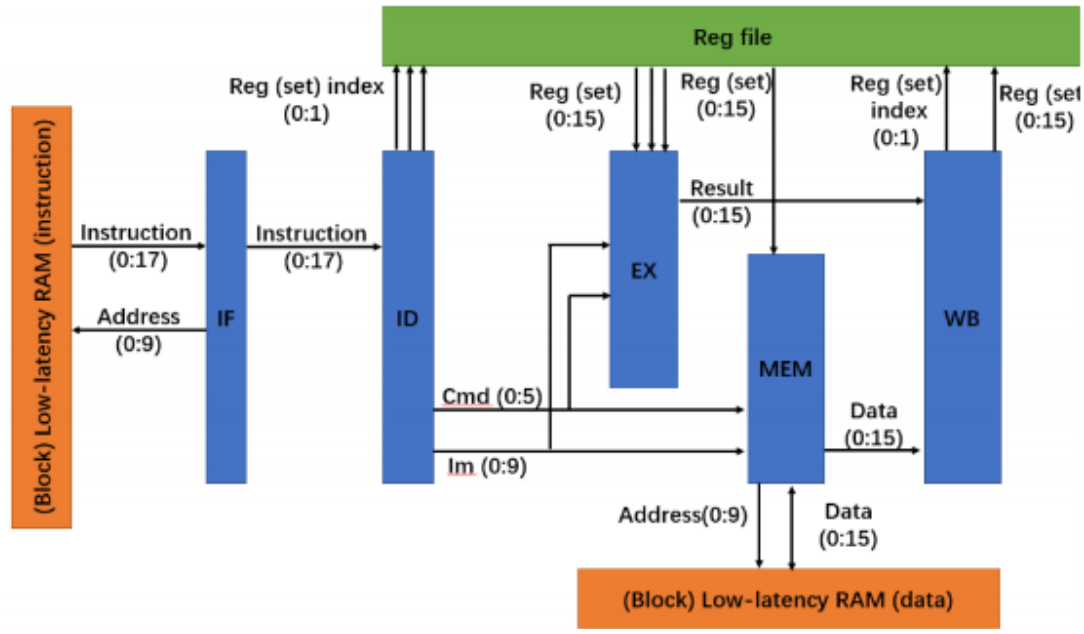



Fig 1. SIMD Architecture

## Architecture:

The five typical stages are IF, ID, EX, MEM, and WB, without pipeline. In stage IF, a 10-bit address will be sent to an instruction Block-RAM (BRAM) to fetch 18-bit instructions. In the stage ID, the instruction will be decoded and some of the control registers will be set to control the following stage. In stage EX, ALU will process data in registers or implement some control commands, e.g. jump. In stage MEM, if the instruction is “store” or “load”, data would be read from/ written to data BRAM, based on instruction and

address. Finally, in stage WB, data will be written back to the register. The pins of the clock, reset, address, data and BRAM enable will be exposed on the interface of the processor. The architecture of the processor is shown in the figure above.

**ALU Components:**

The ALU consists of three SIMD basic computation units: SIMD adder, SIMD multiplier, and SIMD shifter. These three components can be reused for the computation of data with different widths (4-bit, 8-bit, or 16-bit) and can handle all the instructions (listed in Section 4) in the design specification. Each of them is controlled by three input ports, H (for 16-bit), O (for 8-bit), and Q (for 4-bit), indicating the kind of input data, so that the unit can handle the data properly

**SIMD Adder:**

The SIMD adder is implemented based on 4 4-bit adders. To reuse the adder for data with different widths, the input signals, H, O, and Q, control the forwarding of the carries between the adders. Moreover, the adder can also support subtraction, by flipping the second operand and adding one to it when the signal sub is set.

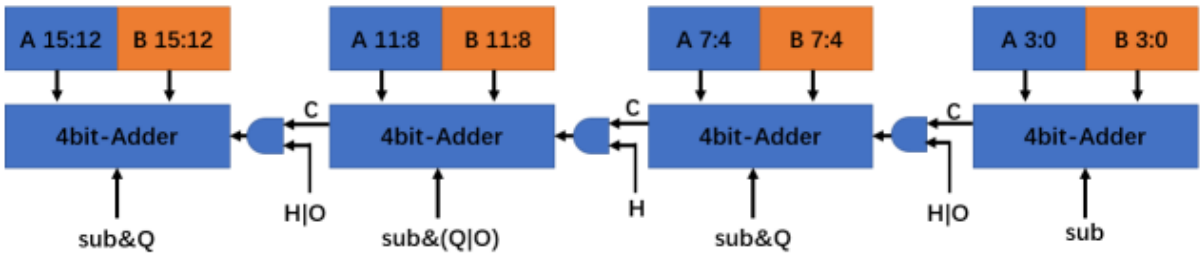



Fig 2. SIMD Adder Data Path

**SIMD Shifter:**

The SIMD shifter in the processor can handle data of different widths by using two 16-bit shifters and the necessary overstep-correct logic. An example of how this shifter operates is shown in Figure 3, which demonstrates a logical left shift. The shifter uses the input signals H and O to determine whether the Most

Significant Bit (MSB) of a 4-bit block should be set to 0 or inherited from the Least Significant Bit (LSB) of the 4-bit block in front of it. This allows the SIMD shifter to perform shift operations on data of various widths

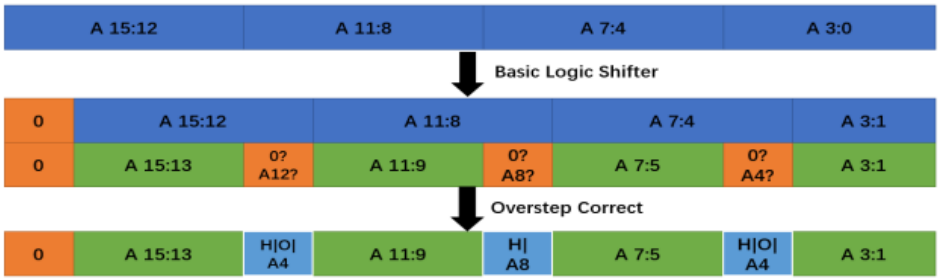



Fig 3. SIMD Shifter

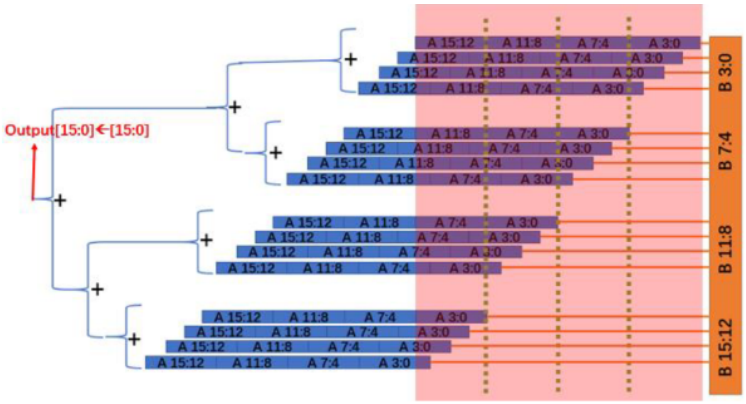
**SIMD Multiplier:**

The SIMD multiplier in the processor is implemented using adders and shifters, and it is able to perform multiplication on data of different widths by using the input signals H, O, and Q to control the input operands of the adders and select the appropriate output for the target data width. For example, a 1x16x16 multiplication is implemented using a typical multiplier structure with an adder tree, as shown in Figure 4. The least significant 16 bits of the sum produced by this multiplier are the output of the multiplication. This allows the SIMD multiplier to perform a wide range of multiplication operations on data of various widths.

Fig 4.

1x16x16 Multiplication

on the processor



## Our Approach:

We went about the project, by first identifying a model that would allow us to do a wide range of model-checking techniques, and provide us with a better understanding of the tool as well as the entire process of model checking of a design using the software.

At first, we chose to verify the design of an 8-bit microcomputer that was written on Verilog HDL, but when going through the base code we realized that the code needed an assembly language compiler written on python for the memory initialization and this was causing us issues when we were trying to integrate it into the model checking tool.

Hence, we switched to the SIMD processor verification for our project, and the first few weeks were spent on understanding the tool and familiarizing ourselves with how assertions work and writing assertions in SVA. And, binding the SVA assertions to our Verilog design. We also put in the time to set up the design repository and worked around with the provided tutorials to run Questa PropCheck.

After this, we utilized the Questa tutorials to understand the basics of the tool. We set up the design as directed by the documents and created directives and other files to compile the final Verilog code. Then SystemVerilog assertions (SVA) were bounded to the Verilog design with relevant inputs. Later, assertions were coded, and analysis was run in the CLI mode, which formed the basis for the debugging of the files. Finally, Questa PropCheck was run in GUI to see the properties and their waveforms.

We first started off by checking simple properties to understand the assertions application better and to ensure that the model we found is operating correctly then after the initial checkpoint we developed more assertions to do an overall model checking assessment. We have also done manual testing/simulation of our design.

We have worked on formally verifying the following properties:

The default clocking period for all assertions is the positive edge of the global clock.

## 1. Functional properties:

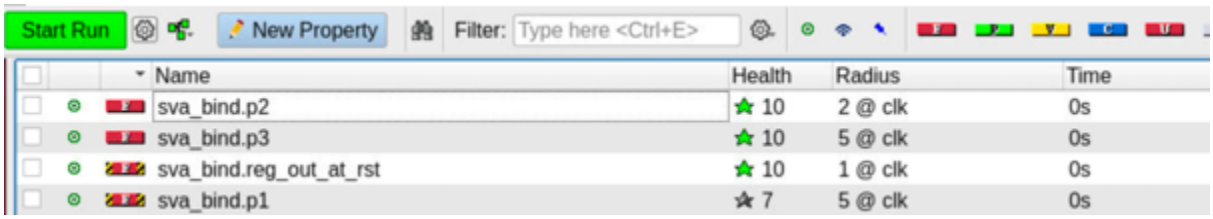
Functional correctness formal verification is the process of using mathematical proofs and other techniques to formally verify that a system or software meets its functional requirements. This process typically involves proving that the system or software meets certain correctness criteria, such as safety, liveness, and security properties. Formal verification can be used to ensure that a system has the desired behavior before it is deployed in production. It is often used in safety-critical applications where a single bug could lead to catastrophic consequences.

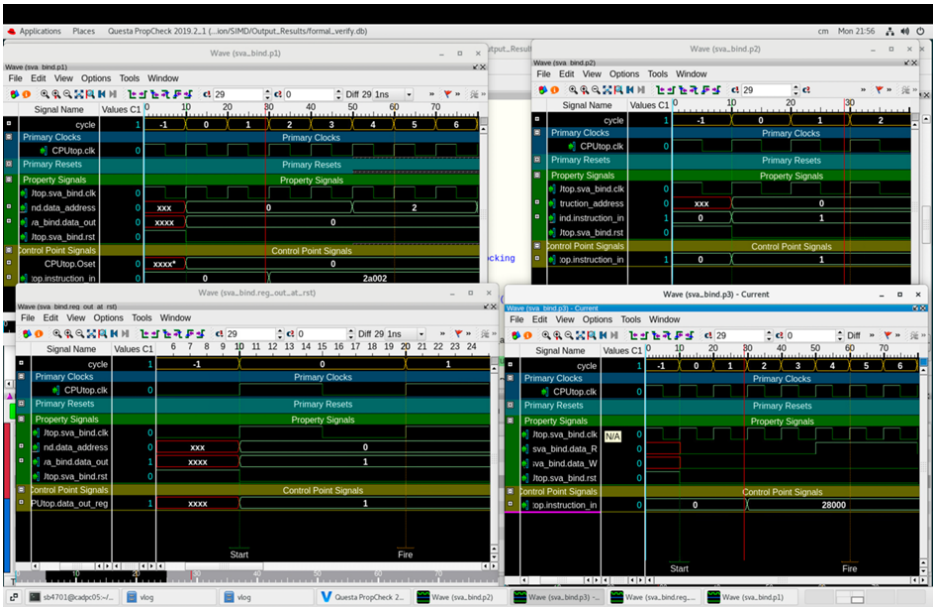
**Initial Checkpoint**

We were giving assertions to test the design. Basic assertions were inserted. We still needed to add more complex assertions to do the functional check and verify if the system is performing operations correctly as described. But we were facing some issues with the assertions failing, so we had to check if the model was failing or if we were doing something wrong with our assertions scripts.

```
// Registers and outputs at reset
reg_out_at_rst: assert property (disable iff(rst) (!rst |-> data_out === '0 && data_address === '0));

// Assertions
p1: assert property (disable iff(rst) (data_address |-> data_out));
p2: assert property (disable iff(rst) (instruction_in |=> instruction_address));
p3: assert property (disable iff(rst) (data_R |-> data_W));
```





## Final Progress

We have written the following properties to functionally verify our SIMD.

```
default clocking c0 @(posedge clk); endclocking

// Functional properties
data_R_positive: assert property (disable iff(rst) @c0 (data_R >= 0));
data_W_positive: assert property (disable iff(rst) @c0 (data_W >= 0));
data_in_positive: assert property (disable iff(rst) @c0 (data_in >= 0));
data_out_positive: assert property (disable iff(rst) @c0  (data_out >= 0));
data_address_positive: assert property (disable iff(rst) @c0 (data_address >= 0));
instruction_in_positive: assert property (disable iff(rst) @c0 (instruction_in >= 0));
instruction_address_positive: assert property (disable iff(rst) @c0 (instruction_address >= 0));
opcode_range: assert property (disable iff(rst) @c0 (opcode >= 0 && opcode <= 63));
addition: assert property (disable iff(rst) @c0 (CMD_addition <= (opcode <= 5)));
substraction: assert property (disable iff(rst) @c0 (CMD_substruction <= (opcode <= 11)));
multiplication: assert property (disable iff(rst) @c0 (CMD_multiplication <= (opcode <= 17)));
mul_accumulation: assert property (disable iff(rst)@c0 (CMD_mul_accumulation <= (opcode <= 20)));
logic_shift_left: assert property (disable iff(rst) @c0 (CMD_logic_shift_left <= (opcode <= 23)));
logic_shift_right: assert property (disable iff(rst) @c0 (CMD_logic_shift_right <= (opcode <= 26)));
and_check: assert property (disable iff(rst) @c0 (CMD_and <= (opcode <= 29)));
or_check: assert property (disable iff(rst) @c0 (CMD_or <= (opcode <= 32)));
not_check: assert property (disable iff(rst) @c0 (CMD_not <= (opcode <= 35)));
loopjump: assert property (disable iff(rst) @c0 (CMD_loopjump <= (opcode <= 36)));
setloop: assert property (disable iff(rst) @c0 (CMD_setloop <= (opcode <= 37)));
load: assert property (disable iff(rst) @c0 (CMD_load <= (opcode <= 40)));
store: assert property (disable iff(rst) @c0 (CMD_store <= (opcode <= 43)));
set: assert property (disable iff(rst) @c0 (CMD_set <= (opcode <= 46)));
```

From **data_R_positive → instruction_address_positive**, these properties check whether the input to the SIMD is correct as per the design.

**opcode_range**: This property checks the range of the opcodes.

From **addition → set**, these properties check the functionality of the relevant operations depending on the opcodes as per the design.

| | | |
|---|---|---|
| sva_bind.data_R_positive | ☆ 0 | 3s |
| sva_bind.data_W_positive | ☆ 7 | 3s |
| sva_bind.data_in_positive | ☆ 7 | 3s |
| sva_bind.data_out_positive | ☆ 7 | 3s |
| sva_bind.data_address_positive | ☆ 7 | 3s |
| sva_bind.instruction_in_positive | ☆ 7 | 3s |
| sva_bind.instruction_address_positive | ☆ 7 | 3s |
| sva_bind.opcode_range | ☆ 0 | 3s |
| sva_bind.addition | ☆ 7 | 3s |
| sva_bind.substraction | ☆ 0 | 3s |
| sva_bind.multiplication | ☆ 0 | 3s |
| sva_bind.mul_accumulation | ☆ 0 | 3s |
| sva_bind.logic_shift_left | ☆ 0 | 3s |
| sva_bind.logic_shift_right | ☆ 0 | 3s |
| sva_bind.and_check | ☆ 0 | 3s |
| sva_bind.or_check | ☆ 0 | 3s |
| sva_bind.not_check | ☆ 0 | 3s |
| sva_bind.loopjump | ☆ 0 | 3s |
| sva_bind.setloop | ☆ 0 | 3s |
| sva_bind.load | ☆ 0 | 3s |
| sva_bind.store | ☆ 7 | 3s |
| sva_bind.set | ☆ 7 | 3s |

2. **Coverage properties:**

The coverage property in formal verification is a measure of how well a verification tool has tested the system by determining how many lines of code or other elements have been tested. Coverage analysis helps to identify areas of the system that are not being tested adequately, so they can be addressed before deployment. It also helps to identify potential bugs and errors that may have been overlooked during development. Coverage analysis can provide valuable insights

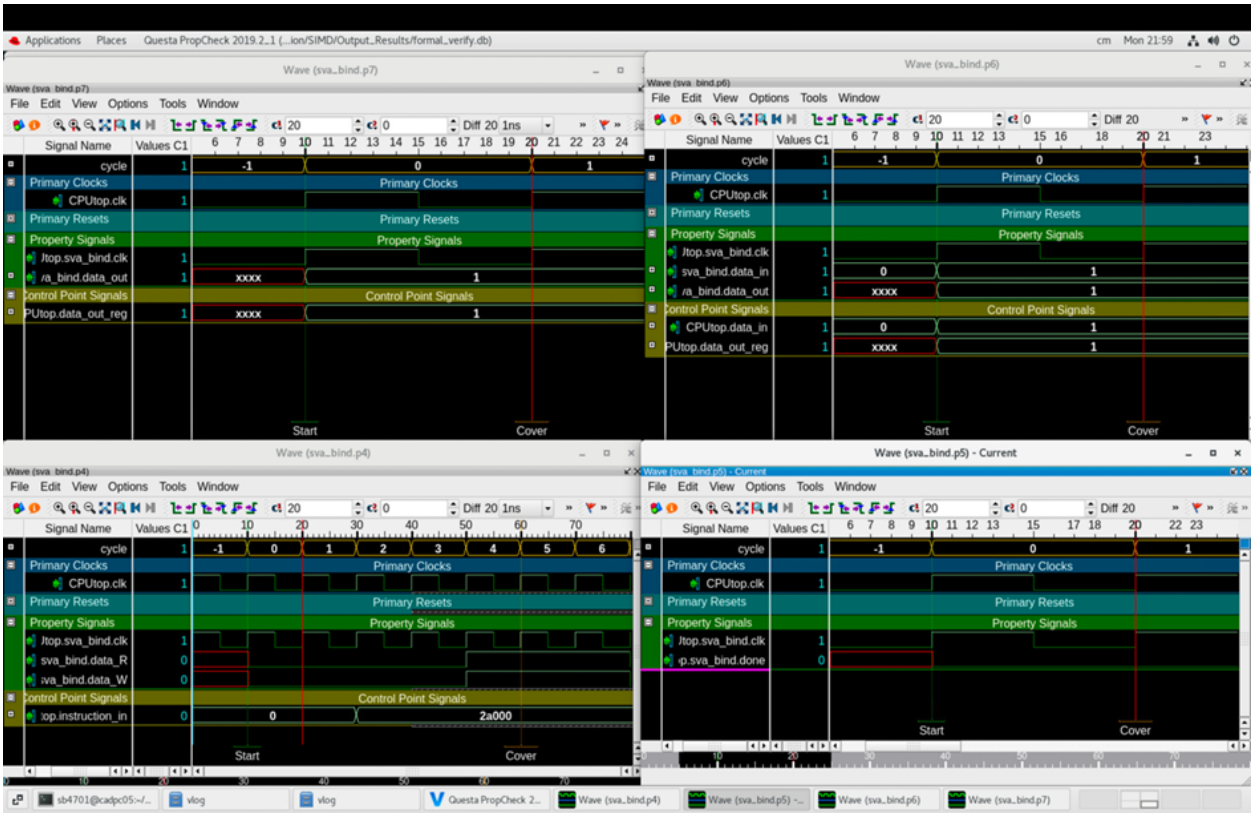
into the robustness of the system and help to reduce the risk of unexpected behavior when deployed [4].

## Initial Checkpoint

We had written some basic cover properties to understand how cover properties work as we weren't initially familiar with them in prior but when we studied them, we deemed that it was an essential part of our model checking project.

```
// Coverage
p4: cover property (data_R |-> data_W);
p5: cover property (~done);
p6: cover property (data_in |-> data_out);
p7: cover property (data_out);
```

| | | | |
|---|---|---|---|
| sva_bind.p4 | 7 | 5 @ clk | 0s |
| sva_bind.p5 | 10 | 1 @ clk | 0s |
| sva_bind.p6 | 10 | 1 @ clk | 0s |
| sva_bind.p7 | 10 | 1 @ clk | 0s |



## Final Progress

We have written the following coverage properties for our SIMD.

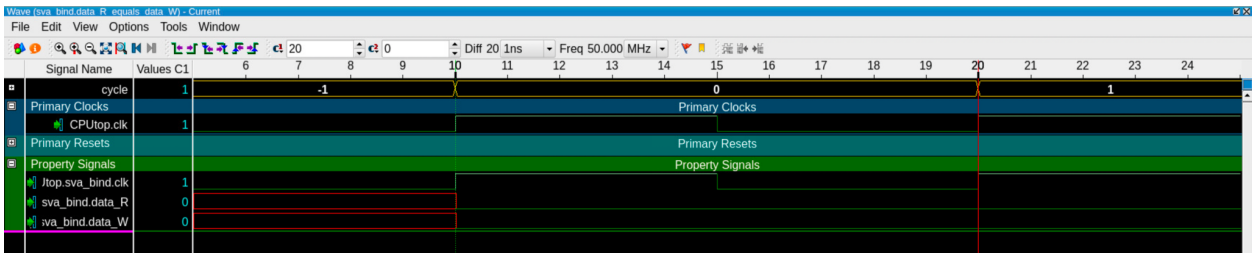
```
// Coverage properties
data_in_data_out: cover property (disable iff(rst) @c0 (data_in |-> $past(data_out)));
data_in_equals_data_out: cover property (disable iff(rst) @c0 (data_in == data_out));
data_R_equals_data_W: cover property ( @c0 (data_R == data_W));
final_state_not_done: cover property (@c0 (~done));
data_not_at_output: cover property (@c0 (~data_out));
read_not_enabled: cover property (@c0 (~rdata_en));
write_not_enabled: cover property (@c0 (~wdata_en));
```

**data_in_data_out**: This property will covers if the data_out flows after the data_in.

From **data_in_equals_data_out → data_R_equals_data_W**, these properties cover whether the antecedent variable is equal to the precedent variable.

From **final_state_not_done → write_not_enabled**, these properties cover whether the declared variable is not enabled.

| | | | | |
|---|---|---|---|---|
| sva_bind.data_R_equals_data_W | 10 | 1 @ clk | 3s |
| sva_bind.final_state_not_done | 10 | 1 @ clk | 3s |
| sva_bind.read_not_enabled | 10 | 1 @ clk | 3s |
| sva_bind.write_not_enabled | 10 | 1 @ clk | 3s |
| sva_bind.data_in_data_out | 10 | 1 @ clk | 3s |
| sva_bind.data_in_equals_data_out | 10 | 1 @ clk | 3s |
| sva_bind.data_not_at_output | 10 | 1 @ clk | 3s |




One of the waveforms for the data_R_equals_data_W cover property.

### 3. Fairness/Liveness properties:

Fairness in formal verification means that all stakeholders involved in the verification process are treated equally. This includes developers, testers, customers, and other stakeholders. Fairness

implies that each stakeholder has an equal say in the decision-making process and that their opinions are taken into consideration when making decisions about the system.

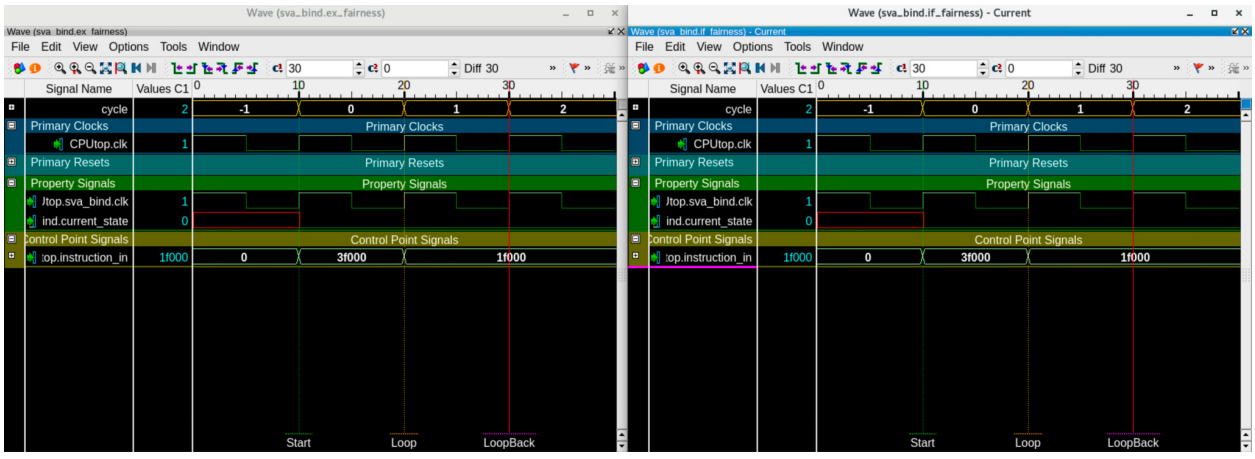
We have written the following fairness/liveness properties for our SIMD [2].

```
// Fairness/Liveness properties
idle_fairness: assert property (s_eventually (current_state == STATE_IDLE));
if_fairness: assert property (s_eventually (current_state == STATE_IF));
id_fairness: assert property (s_eventually (current_state == STATE_ID));
ex_fairness: assert property (s_eventually (current_state == STATE_EX));
mem_fairness: assert property (s_eventually (current_state == STATE_MEM));
halt_fairness: assert property (s_eventually (current_state == STATE_HALT));
```

From **idle_fairness → halt_fairness**, all the properties assert that eventually, the current state will reach the declared states (check the image below) in the future.



The above two properties are getting fired in the simulation. Though IF and EX should be executed in our SIMD design, we were not able to figure out and debug this [4].



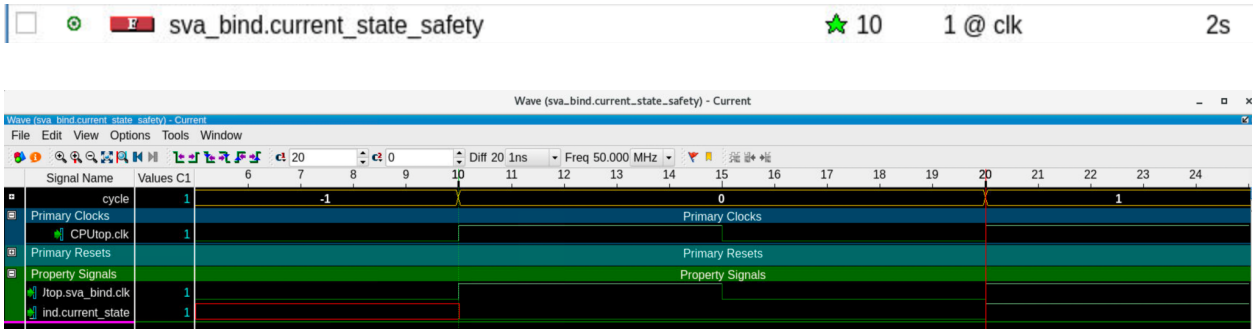
4. **Safety properties:**

We have written the following safety properties for our SIMD.

```
// Safety properties
current_state_safety: assert property (always (current_state));
```

**current_state_safety**: This property asserts that there will always be a current state.

The above property is getting fired in the simulation. Though there will be current_state declared in our SIMD design, we were not able to figure out and debug this [4].




5. **False positive check:**

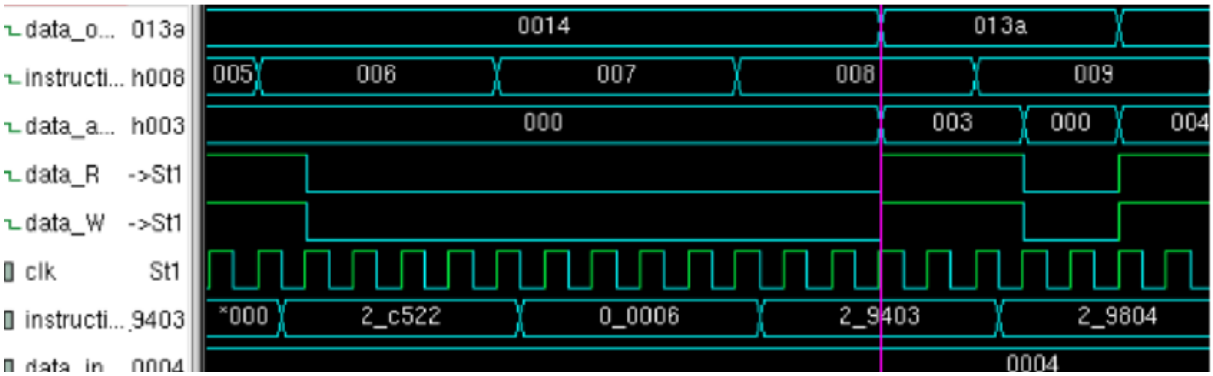
We have done manual testing/simulation for our design-under-test and we can conclude that no false positives were found. We have found by manually checking all the values of the registers for all stages to see if there was any overflow or any input that was hindering the functionality of the SIMD [2].

6. **Latency insensitive check:**

Latency-insensitive check is an important step in formal verification. This check verifies that the design does not have any timing violations caused by signal latency. It involves verifying that all the signals in the design are received and processed in the same clock cycle, regardless of any

delays caused by routing or other factors. The verification process ensures that the design does not contain any timing-related errors which could lead to unexpected behavior or system failure. This check is especially important for designs with multiple clocks since it ensures that all signals are synchronized properly. Latency-insensitive checks also help to reduce power consumption and improve system performance.

From the current waveforms that have been obtained from the design simulation, we can conclude that the design is not latency insensitive because some of the modules are dependent on the inputs of the other functional modules. Therefore, the current SIMD processor is not latency insensitive [3].




## Setbacks and Scope of Improvement

Initially, setting up the tool and understanding the crux of assertions. We had a lot of struggles in understanding the syntax for assertions and implementing them in our design. A lot of our time was consumed in understanding the material and the Questa tool, we went through the tasks assigned on the example FIFO to understand the model but chose to implement the requirements on our project as we wanted to learn to tackle fundamental issues on our project so it would provide us an opportunity to go through the functionality of the chosen model.

We were also facing issues in implementing the assertions to check the correctness of the jump functions that are being used in the model. We could see that it was being used by the system, however, we were unable to write an assertion for it, we were able to conclude this because the functional assertions were failing even though it seemed to us that it was being used.

We also are still having issues with a fairness check of the IF state and the EX state, these are the states where the instruction is fetched and the data computation is executed, so we still need to look into why this issue is occurring. These 2 states have to occur for the system to operate, and through the waveform generated by Questasim, we could see that we were getting the right outputs.

## **Instructions for the files:**

The source files are attached to the report as well. It includes two directories, *SIMD* and *Basic-SIMD-Processor-Verilog-Tutorial*.

*Basic-SIMD-Processor-Verilog-Tutorial* directory consists of the original source of the design-under-test.

*SIMD* directory consists our work with the assertions in the src file and etc. To run the simulation, open this directory in the terminal and run ```***make run***```.

## References:


1. https://github.com/zslwyuan/Basic-SIMD-Processor-Verilog-Tutorial.
2. Seligman, Erik, et al. Formal Verification: An Essential Toolkit for Modern VLSI Design. 2nd ed., Academic Press, 2023.
3. Todd M Austin. 2000. DIVA: A Dynamic Approach to Microprocessor Verification. https://www.eecg.utoronto.ca/~moshovos/ACA06/readings/JILP-diva.pdf.
4. Ying, Mingsheng, and Yuan Feng. Model Checking Quantum Systems: Principles and Algorithms. 1st ed., Cambridge UP, 2021.